

ЯЗЫК ПРОГРАММИРОВАНИЯ C# 5.0 И ПЛАТФОРМА .NET 4.5

*Совершите увлекательное путешествие
по вселенной .NET!*

Эндрю Троелсен



www.williamspublishing.com

apress®

www.apress.com

ЯЗЫК ПРОГРАММИРОВАНИЯ **C# 5.0** И ПЛАТФОРМА **.NET 4.5**

6-е издание

Эндрю Троелсен

«



Москва • Санкт-Петербург • Киев
2013

ББК 32.973.26-018.2.75

Т70

УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией С.Н. Тригуб

Перевод с английского Ю.Н. Артеменко

Под редакцией Ю.Н. Артеменко

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, <http://www.williamspublishing.com>

Троелсен, Эндрю.

Т70 Язык программирования C# 5.0 и платформа .NET 4.5, 6-е изд. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2013. — 1312 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1814-7 (рус.)

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства APress, Berkeley, CA.

Authorized translation from the English language edition published by APress, Inc., Copyright © 2012 by Andrew Troelsen.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher. Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Russian language edition is published by Williams Publishing House according to the Agreement with R&I Enterprises International. Copyright © 2013.

Научно-популярное издание

Эндрю Троелсен

Язык программирования C# 5.0 и платформа .NET 4.5 6-е издание

Верстка Т.Н. Артеменко

Художественный редактор В.Г. Павлютин

Подписано в печать 04.02.2013. Формат 70×100/16.

Гарнитура Times. Печать офсетная.

Усл. печ. л. 105.78. Уч.-изд. л. 84.5.

Тираж 2000 экз. Заказ № 3512.

Первая Академическая типография "Наука"

199034, Санкт-Петербург, 9-я линия, 12/28

ООО "И. Д. Вильямс", 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-1814-7 (рус.)

ISBN 978-1-43-024233-8 (англ.)

© Издательский дом "Вильямс", 2013

© by Andrew Troelsen, 2012

Оглавление

Часть I. Введение в C# и платформу .NET	43
Глава 1. Философия .NET	44
Глава 2. Создание приложений на языке C#	78
Часть II. Основы программирования на C#	103
Глава 3. Главные конструкции программирования на C#: часть I	104
Глава 4. Главные конструкции программирования на C#: часть II	146
Часть III. Объектно-ориентированное программирование на C#	181
Глава 5. Инкапсуляция	182
Глава 6. Понятие наследования и полиморфизма	227
Глава 7. Структурированная обработка исключений	261
Глава 8. Работа с интерфейсами	287
Часть IV. Дополнительные конструкции программирования на C#	321
Глава 9. Коллекции и обобщения	322
Глава 10. Делегаты, события и лямбда-выражения	357
Глава 11. Расширенные средства языка C#	391
Глава 12. LINQ to Objects	426
Глава 13. Время жизни объектов	455
Часть V. Программирование с использованием сборок .NET	481
Глава 14. Построение и конфигурирование библиотек классов	482
Глава 15. Рефлексия типов, позднее связывание и программирование с использованием атрибутов	528
Глава 16. Динамические типы и среда DLR	567
Глава 17. Процессы, домены приложений и объектные контексты	588
Глава 18. Язык CIL и роль динамических сборок	612
Часть VI. Введение в библиотеки базовых классов .NET	651
Глава 19. Многопоточное, параллельное и асинхронное программирование	652
Глава 20. Файловый ввод-вывод и сериализация объектов	702
Глава 21. ADO.NET, часть I: подключенный уровень	745
Глава 22. ADO.NET, часть II: автономный уровень	795
Глава 23. ADO.NET, часть III: Entity Framework	850
Глава 24. Введение в LINQ to XML	884
Глава 25. Введение в Windows Communication Foundation	899
Глава 26. Введение в Windows Workflow Foundation	953
Часть VII. Windows Presentation Foundation	987
Глава 27. Введение в Windows Presentation Foundation и XAML	988
Глава 28. Программирование с использованием элементов управления WPF	1043
Глава 29. Службы визуализации графики WPF	1098
Глава 30. Ресурсы, анимация и стили WPF	1136
Глава 31. Свойства зависимости, маршрутизируемые события и шаблоны	1165
Часть VIII. ASP.NET Web Forms	1193
Глава 32. Введение в ASP.NET Web Forms	1194
Глава 33. Веб-элементы управления, мастер-страницы и темы ASP.NET	1235
Глава 34. Управление состоянием в ASP.NET	1274
Предметный указатель	1306

Содержание

Об авторе	32
О техническом редакторе	32
Благодарности	32
Введение	33
Автор и читатели — одна команда	33
Краткий обзор книги	34
Часть I. Введение в C# и платформу .NET	34
Часть II. Основы программирования на C#	34
Часть III. Объектно-ориентированное программирование на C#	35
Часть IV. Дополнительные конструкции программирования на C#	36
Часть V. Программирование с использованием сборок .NET	37
Часть VI. Введение в библиотеки базовых классов .NET	38
Часть VII. Windows Presentation Foundation	40
Часть VIII. ASP.NET Web Forms	41
Загружаемые приложения	41
Исходный код примеров	42
От издательства	42
Часть I. Введение в C# и платформу .NET	43
Глава 1. Философия .NET	44
Начальное знакомство с платформой .NET	44
Некоторые основные преимущества платформы .NET	45
Введение в строительные блоки платформы .NET (CLR, CTS и CLS)	46
Роль библиотек базовых классов	46
Что привносит язык C#	47
Сравнение управляемого и неуправляемого кода	49
Другие языки программирования, ориентированные на .NET	49
Жизнь в многоязычном окружении	50
Обзор сборок .NET	51
Роль языка CIL	52
Роль метаданных типов .NET	55
Роль манифеста сборки	56
Понятие общей системы типов (CTS)	56
Типы классов CTS	57
Типы интерфейсов CTS	57
Типы структур CTS	58
Типы перечислений CTS	58
Типы делегатов CTS	59
Члены типов CTS	59
Встроенные типы данных CTS	59
Понятие общезыковой спецификации (CLS)	60
Обеспечение совместимости с CLS	62
Понятие общезыковой исполняющей среды (CLR)	62
Различия между сборками, пространствами имен и типами	64
Роль корневого пространства имен Microsoft	67
Доступ к пространству имен программным образом	67
Ссылка на внешние сборки	68

Исследование сборки с помощью утилиты <code>ildasm.exe</code>	69
Просмотр CIL-кода	70
Просмотр метаданных типов	70
Просмотр метаданных сборки (манифеста)	71
Независимая от платформы природа .NET	71
Несколько слов по поводу приложений Windows 8	73
Построение приложений в стиле Metro	74
Роль .NET в среде Windows 8	75
Резюме	77
Глава 2. Создание приложений на языке C#	78
Роль .NET Framework 4.5 SDK	78
Окно командной строки разработчика	79
Построение приложений C# с использованием <code>csc.exe</code>	79
Указание целевых входных и выходных параметров	80
Ссылка на внешние сборки	82
Ссылка на несколько внешних сборок	83
Компиляция нескольких файлов исходного кода	83
Работа с ответными файлами в C#	84
Построение приложений .NET с использованием Notepad++	85
Построение приложений .NET с помощью SharpDevelop	86
Создание простого тестового проекта	87
Построение приложений .NET с помощью Visual C# Express	89
Некоторые уникальные возможности Visual C# Express	89
Построение приложений .NET с помощью Visual Studio	89
Некоторые уникальные возможности Visual Studio	90
Выбор целевой версии .NET Framework в диалоговом окне New Project	91
Использование утилиты Solution Explorer	91
Утилита Class View	93
Утилита Object Browser	93
Встроенная поддержка рефакторинга кода	94
Фрагменты кода и технология Surround With	96
Утилита Class Designer	98
Интегрированная система документации .NET Framework 4.5 SDK	100
Резюме	102
Часть II. Основы программирования на C#	103
Глава 3. Главные конструкции программирования на C#: часть I	104
Структура простой программы C#	104
Вариации метода <code>Main()</code>	106
Указание кода ошибки приложения	107
Обработка аргументов командной строки	108
Указание аргументов командной строки в Visual Studio	109
Интересное отступление от темы: некоторые дополнительные члены класса <code>System.Environment</code>	110
Класс <code>System.Console</code>	111
Базовый ввод-вывод с помощью класса <code>Console</code>	112
Форматирование консольного вывода	113
Форматирование числовых данных	114
Форматирование числовых данных в приложениях, отличных от консольных	115
Системные типы данных и соответствующие ключевые слова C#	115
Объявление и инициализация переменных	117

Внутренние типы данных и операция new	118
Иерархия классов для типов данных	119
Члены числовых типов данных	120
Члены System.Boolean	121
Члены System.Char	121
Синтаксический разбор значений из строковых данных	121
Типы System.DateTime и System.TimeSpan	122
Сборка System.Numerics.dll	122
Работа со строковыми данными	124
Базовые манипуляции строками	125
Конкатенация строк	125
Управляющие последовательности	126
Определение дословных строк	127
Строки и равенство	127
Строки являются неизменяемыми	128
Тип System.Text.StringBuilder	129
Сужающие и расширяющие преобразования типов данных	130
Ключевое слово checked	133
Настройка проверки переполнения на уровне всего проекта	134
Ключевое слово unchecked	135
Понятие неявно типизированных локальных переменных	136
Ограничения неявно типизированных переменных	137
Неявно типизированные данные являются строго типизированными	138
Польза от неявно типизированных локальных переменных	138
Итерационные конструкции C#	139
Цикл for	140
Цикл foreach	140
Циклы while и do/while	141
Конструкции принятия решений и операции равенства/сравнения	142
Оператор if/else	142
Операции равенства и сравнения	142
Условные операции	143
Оператор switch	143
Резюме	145
Глава 4. Главные конструкции программирования на C#: часть II	146
Методы и модификаторы параметров	146
Стандартное поведение передачи параметров по значению	147
Модификатор out	148
Модификатор ref	149
Модификатор params	150
Определение необязательных параметров	151
Вызов методов с использованием именованных параметров	152
Понятие перегрузки методов	154
Массивы в C#	156
Синтаксис инициализации массивов C#	157
Неявно типизированные локальные массивы	158
Определение массива объектов	158
Работа с многомерными массивами	159
Использование массивов в качестве аргументов и возвращаемых значений	160
Базовый класс System.Array	161
Тип enum	162
Управление хранилищем, лежащим в основе перечисления	163

Объявление переменных типа перечисления	164
Тип <code>System.Enum</code>	165
Динамическое извлечение пар "имя/значение" перечисления	166
Типы структур	168
Создание переменных типа структур	169
Типы значений и ссылочные типы	170
Типы значений, ссылочные типы и операция присваивания	171
Типы значений, содержащие ссылочные типы	173
Передача ссылочных типов по значению	174
Передача ссылочных типов по ссылке	175
Заключительные детали относительно типов значений и ссылочных типов	176
Понятие типов, допускающих <code>null</code> , в C#	177
Работа с типами, допускающими <code>null</code>	178
Операция ??	179
Резюме	180
Часть III. Объектно-ориентированное программирование на C#	181
Глава 5. Инкапсуляция	182
Знакомство с типом класса C#	182
Размещение объектов с помощью ключевого слова <code>new</code>	184
Понятие конструкторов	185
Роль стандартного конструктора	185
Определение специальных конструкторов	186
Еще раз о стандартном конструкторе	187
Роль ключевого слова <code>this</code>	188
Построение цепочки вызовов конструкторов с использованием <code>this</code>	189
Обзор потока конструктора	192
Еще раз о необязательных аргументах	193
Понятие ключевого слова <code>static</code>	194
Определение статических полей данных	195
Определение статических методов	197
Определение статических конструкторов	198
Определение статических классов	200
Основные принципы объектно-ориентированного программирования	201
Роль инкапсуляции	201
Роль наследования	202
Роль полиморфизма	203
Модификаторы доступа C#	204
Стандартные модификаторы доступа	205
Модификаторы доступа и вложенные типы	206
Первый принцип: службы инкапсуляции C#	206
Инкапсуляция с использованием традиционных методов доступа и изменения	207
Инкапсуляция с использованием свойств .NET	209
Использование свойств внутри определения класса	212
Свойства, доступные только для чтения и только для записи	213
Еще раз о ключевом слове <code>static</code> : определение статических свойств	214
Понятие автоматических свойств	215
Взаимодействие с автоматическими свойствами	216
Замечания относительно автоматических свойств и стандартных значений	217
Понятие синтаксиса инициализации объектов	218
Вызов специальных конструкторов с помощью синтаксиса инициализации	220
Инициализация вложенных типов	221

Работа с данными константных полей	222
Понятие полей, допускающих только чтение	223
Статические поля, допускающие только чтение	224
Понятие частичных типов	224
Резюме	226
Глава 6. Понятие наследования и полиморфизма	227
Базовый механизм наследования	227
Указание родительского класса для существующего класса	228
Замечание относительно множества базовых классов	230
Ключевое слово <code>sealed</code>	230
Изменение диаграмм классов Visual Studio	231
Второй принцип ООП: подробности о наследовании	233
Управление созданием базового класса с помощью ключевого слова <code>base</code>	234
Хранение секретов семейства: ключевое слово <code>protected</code>	236
Добавление запечатанного класса	237
Реализация модели включения/делегации	237
Определения вложенных типов	239
Третий принцип ООП: поддержка полиморфизма в C#	240
Ключевые слова <code>virtual</code> и <code>override</code>	241
Переопределение виртуальных членов в IDE-среде Visual Studio	243
Запечатывание виртуальных членов	244
Абстрактные классы	244
Понятие полиморфного интерфейса	246
Скрытие членов	249
Правила приведения к базовому и производному классу	251
Ключевое слово <code>as</code>	252
Ключевое слово <code>is</code>	253
Главный родительский класс <code>System.Object</code>	253
Переопределение <code>System.Object.ToString()</code>	256
Переопределение <code>System.Object.Equals()</code>	257
Переопределение <code>System.Object.GetHashCode()</code>	258
Тестирование модифицированного класса <code>Person</code>	259
Статические члены <code>System.Object</code>	260
Резюме	260
Глава 7. Структурированная обработка исключений	261
Ода ошибкам и исключениям	261
Роль обработки исключений .NET	262
Строительные блоки обработки исключений в .NET	263
Базовый класс <code>System.Exception</code>	264
Простейший пример	265
Генерация общего исключения	267
Перехват исключений	268
Конфигурирование состояния исключения	269
Свойство <code>TargetSite</code>	269
Свойство <code>StackTrace</code>	270
Свойство <code>HelpLink</code>	271
Свойство <code>Data</code>	271
Исключения уровня системы (<code>System.SystemException</code>)	273
Исключения уровня приложения (<code>System.ApplicationException</code>)	274
Построение специальных исключений, способ первый	274

Построение специальных исключений, способ второй	276
Построение специальных исключений, способ третий	277
Обработка нескольких исключений	278
Общие операторы <code>catch</code>	280
Повторная генерация исключений	281
Внутренние исключения	282
Блок <code>finally</code>	283
Какие исключения могут генерировать методы?	283
Результат наличия необработанных исключений	284
Отладка необработанных исключений с использованием Visual Studio	285
Резюме	285
Глава 8. Работа с интерфейсами	287
Понятие интерфейсных типов	287
Сравнение интерфейсных типов и абстрактных базовых классов	288
Определение специальных интерфейсов	290
Реализация интерфейса	292
Вызов членов интерфейса на уровне объектов	294
Получение ссылок на интерфейсы: ключевое слово <code>as</code>	295
Получение ссылок на интерфейсы: ключевое слово <code>is</code>	295
Использование интерфейсов в качестве параметров	296
Использование интерфейсов в качестве возвращаемых значений	298
Массивы интерфейсных типов	298
Реализация интерфейсов с использованием Visual Studio	299
Явная реализация интерфейсов	301
Проектирование иерархий интерфейсов	303
Множественное наследование посредством интерфейсных типов	304
Интерфейсы <code>IEnumerable</code> и <code>IEnumerator</code>	306
Построение методов итератора с применением ключевого слова <code>yield</code>	308
Построение именованного итератора	310
Интерфейс <code>ICloneable</code>	311
Более сложный пример клонирования	313
Интерфейс <code>IComparable</code>	315
Указание нескольких порядков сортировки посредством <code>IComparer</code>	318
Специальные свойства и специальные типы сортировки	319
Резюме	320
Часть IV. Дополнительные конструкции программирования на C#	321
Глава 9. Коллекции и обобщения	322
Побудительные причины создания классов коллекций	322
Пространство имен <code>System.Collections</code>	324
Обзор пространства имен <code>System.Collections.Specialized</code>	326
Проблемы, связанные с необобщенными коллекциями	327
Проблема производительности	327
Проблемы с безопасностью типов	330
Первый взгляд на обобщенные коллекции	333
Роль параметров обобщенных типов	334
Указание параметров типа для обобщенных классов и структур	335
Указание параметров типа для обобщенных членов	336
Указание параметров типов для обобщенных интерфейсов	336

Пространство имен <code>System.Collections.Generic</code>	338
Синтаксис инициализации коллекций	339
Работа с классом <code>List<T></code>	340
Работа с классом <code>Stack<T></code>	342
Работа с классом <code>Queue<T></code>	342
Работа с классом <code>SortedSet<T></code>	344
Пространство имен <code>System.Collections.ObjectModel</code>	345
Работа с <code>ObservableCollection<T></code>	345
Создание специальных обобщенных методов	347
Выведение параметров типа	349
Создание специальных обобщенных структур и классов	351
Ключевое слово <code>default</code> в обобщенном коде	352
Ограничение параметров типа	353
Примеры использования ключевого слова <code>where</code>	353
Недостаток ограничений операций	355
Резюме	356
Глава 10. Делегаты, события и лямбда-выражения	357
Понятие типа делегата <code>.NET</code>	357
Определение типа делегата в <code>C#</code>	358
Базовые классы <code>System.MulticastDelegate</code> и <code>System.Delegate</code>	360
Пример простейшего делегата	362
Исследование объекта делегата	363
Отправка уведомлений о состоянии объекта с использованием делегатов	364
Включение группового вызова	367
Удаление целей из списка вызовов делегата	368
Синтаксис групповых преобразований методов	369
Понятие обобщенных делегатов	370
Обобщенные делегаты <code>Action<></code> и <code>Func<></code>	371
Понятие событий <code>C#</code>	373
Ключевое слово <code>event</code>	375
“За кулисами” событий	376
Прослушивание входящих событий	377
Упрощенная регистрация событий с использованием <code>Visual Studio</code>	378
Создание специальных аргументов событий	379
Обобщенный делегат <code>EventHandler<T></code>	381
Понятие анонимных методов <code>C#</code>	381
Доступ к локальным переменным	383
Понятие лямбда-выражений	384
Анализ лямбда-выражения	386
Обработка аргументов внутри множества операторов	387
Лямбда-выражения с несколькими параметрами и без параметров	388
Усовершенствование примера <code>PrimAndProperCarEvents</code> за счет использования лямбда-выражений	389
Резюме	390
Глава 11. Расширенные средства языка <code>C#</code>	391
Понятие методов-индексаторов	391
Индексация данных с использованием строковых значений	393
Перегрузка методов-индексаторов	394
Многомерные индексаторы	395
Определения индексаторов в интерфейсных типах	396

Понятие перегрузки операций	396
Перегрузка бинарных операций	397
А как насчет операций += и -=?	399
Перегрузка унарных операций	399
Перегрузка операций эквивалентности	400
Перегрузка операций сравнения	401
Финальные соображения относительно перегрузки операций	402
Понятие специальных преобразований типов	402
Числовые преобразования	402
Преобразования между связанными типами классов	403
Создание специальных процедур преобразования	403
Дополнительные явные преобразования для типа <code>Square</code>	406
Определение процедур неявного преобразования	406
Понятие расширяющих методов	408
Понятие анонимных типов	412
Определение анонимного типа	413
Анонимные типы, содержащие другие анонимные типы	417
Работа с типами указателей	417
Ключевое слово <code>unsafe</code>	419
Работа с операциями * и &	421
Небезопасная (и безопасная) функция обмена	421
Доступ к полям через указатели (операция ->)	422
Ключевое слово <code>stackalloc</code>	423
Закрепление типа с помощью ключевого слова <code>fixed</code>	423
Ключевое слово <code>sizeof</code>	424
Резюме	425
Глава 12. LINQ to Objects	426
Программные конструкции, специфичные для LINQ	426
Неявная типизация локальных переменных	427
Синтаксис инициализации объектов и коллекций	427
Лямбда-выражения	428
Расширяющие методы	429
Анонимные типы	429
Роль LINQ	430
Выражения LINQ строго типизированы	431
Основные сборки LINQ	431
Применение запросов LINQ к элементарным массивам	432
Решение без использования LINQ	433
Рефлексия результирующего набора LINQ	434
LINQ и неявно типизированные локальные переменные	435
LINQ и расширяющие методы	436
Роль отложенного выполнения	437
Роль немедленного выполнения	437
Возврат результата запроса LINQ	438
Возврат результатов LINQ через немедленное выполнение	440
Применение запросов LINQ к объектам коллекций	440
Доступ к содержащимся в контейнере подобъектам	441
Применение запросов LINQ к необобщенным коллекциям	442
Фильтрация данных с использованием <code>OfType<T>()</code>	442
Исследование операций запросов LINQ	443
Базовый синтаксис выборки	444
Получение подмножества данных	445

Проецирование новых типов данных	446
Получение счетчиков посредством Enumerable	447
Обращение результирующих наборов	447
Выражения сортировки	447
LINQ как лучшее средство построения диаграмм	448
Исключение дубликатов	449
Агрегатные операции LINQ	449
Внутреннее представление операторов запросов LINQ	450
Построение выражений запросов с использованием операций запросов	451
Построение выражений запросов с использованием типа Enumerable и лямбда-выражений	451
Построение выражений запросов с использованием типа Enumerable и анонимных методов	453
Построение выражений запросов с использованием типа Enumerable и низкоуровневых делегатов	453
Резюме	454
Глава 13. Время жизни объектов	455
Классы, объекты и ссылки	455
Базовые сведения о времени жизни объектов	456
Код CIL для ключевого слова new	457
Установка объектных ссылок в null	458
Роль корневых элементов приложения	459
Понятие поколений объектов	461
Параллельная сборка мусора в .NET 1.0 — .NET 3.5	462
Фоновая сборка мусора в .NET 4.0 и последующих версиях	462
Тип System.GC	463
Принудительный запуск сборщика мусора	464
Построение финализируемых объектов	466
Переопределение System.Object.Finalize()	467
Описание процесса финализации	469
Создание освобождаемых объектов	470
Повторное использование ключевого слова using в C#	472
Создание финализируемых и освобождаемых типов	473
Формализованный шаблон освобождения	474
Ленивое создание объектов	476
Настройка процесса создания данных Lazy<>	479
Резюме	480
Часть V. Программирование с использованием сборок .NET	481
Глава 14. Построение и конфигурирование библиотек классов	482
Определение специальных пространств имен	482
Устранение конфликтов между именами посредством полностью заданных имен	484
Устранение конфликтов между именами посредством псевдонимов	485
Создание вложенных пространств имен	487
Стандартное пространство имен Visual Studio	488
Роль сборок .NET	488
Сборки увеличивают возможности для повторного использования кода	489
Сборки определяют границы типов	489
Сборки являются единицами, поддерживающими версии	489
Сборки являются самоописательными	490
Сборки являются конфигурируемыми	490

Формат сборки .NET	490
Заголовок файла Windows	490
Заголовок файла CLR	492
CIL-код, метаданные типов и манифест сборки	493
Необязательные ресурсы сборки	494
Построение и использование специальной библиотеки классов	494
Исследование манифеста	497
Исследование CIL-кода	499
Исследование метаданных типов	500
Построение клиентского приложения на C#	500
Построение клиентского приложения на Visual Basic	502
Межъязыковое наследование в действии	503
Понятие закрытых сборок	504
Идентичность закрытой сборки	504
Понятие процесса зондирования	504
Конфигурирование закрытых сборок	505
Роль файла App.Config	507
Понятие разделяемых сборок	508
Глобальный кеш сборок	509
Понятие строгих имен	510
Генерация строгих имен в командной строке	511
Генерация строгих имен в Visual Studio	513
Установка строго именованных сборок в GAC	514
Использование разделяемой сборки	515
Исследование манифеста SharedCarLibClient	517
Конфигурирование разделяемых сборок	517
Фиксация текущей версии разделяемой сборки	518
Построение разделяемой сборки версии 2.0.0.0	518
Динамическое перенаправление на специфичные версии разделяемой сборки	520
Понятие сборок политик издателя	522
Отключение политики издателя	523
Элемент <codeBase>	523
Пространство имен System.Configuration	525
Документация по схеме конфигурационного файла	526
Резюме	526
Глава 15. Рефлексия типов, позднее связывание и программирование с использованием атрибутов	528
Потребность в метаданных типов	528
Просмотр (частичных) метаданных для перечисления EngineState	529
Просмотр (частичных) метаданных для типа Car	530
Исследование блока TypeRef	531
Документирование определяемой сборки	532
Документирование ссылаемых сборок	532
Документирование строковых литералов	532
Понятие рефлексии	533
Класс System.Type	534
Получение информации о типе с помощью System.Object.GetType()	534
Получение информации о типе с помощью typeof()	535
Получение информации о типе с помощью System.Type.GetType()	535
Построение специального средства для просмотра метаданных	536
Рефлексия методов	536

Рефлексия полей и свойств	537
Рефлексия реализованных интерфейсов	537
Отображение различных дополнительных деталей	538
Реализация метода Main()	538
Рефлексия обобщенных типов	540
Рефлексия параметров и возвращаемых значений методов	540
Динамически загружаемые сборки	541
Рефлексия разделяемых сборок	543
Позднее связывание	545
Класс System.Activator	546
Вызов методов без параметров	547
Вызов методов с параметрами	548
Роль атрибутов .NET	549
Потребители атрибутов	550
Применение атрибутов в C#	550
Сокращенная нотация атрибутов C#	551
Указание параметров конструктора для атрибутов	552
Атрибут [Obsolete] в действии	552
Построение специальных атрибутов	553
Применение специальных атрибутов	553
Синтаксис именованных свойств	554
Ограничение использования атрибутов	554
Атрибуты уровня сборки	555
Файл AssemblyInfo.cs, генерируемый Visual Studio	556
Рефлексия атрибутов с использованием раннего связывания	557
Рефлексия атрибутов с использованием позднего связывания	558
Возможное применение на практике рефлексии, позднего связывания и специальных атрибутов	559
Построение расширяемого приложения	560
Построение сборки CommonSnappableTypes.dll	561
Построение оснастки на C#	561
Построение оснастки на Visual Basic	562
Построение расширяемого приложения Windows Forms	563
Резюме	566
Глава 16. Динамические типы и среда DLR	567
Роль ключевого слова dynamic языка C#	567
Вызов членов на динамически объявленных данных	569
Роль сборки Microsoft.CSharp.dll	570
Область применения ключевого слова dynamic	571
Ограничения ключевого слова dynamic	572
Практическое применение ключевого слова dynamic	572
Роль исполняющей среды динамического языка (DLR)	573
Роль деревьев выражений	574
Роль пространства имен System.Dynamic	574
Динамический поиск в деревьях выражений во время выполнения	575
Упрощение вызовов с поздним связыванием посредством динамических типов	575
Использование ключевого слова dynamic для передачи аргументов	576
Упрощение взаимодействия с COM посредством динамических данных	578
Роль основных сборок взаимодействия	579
Встраивание метаданных взаимодействия	580
Общие сложности взаимодействия с COM	581

Взаимодействие с COM с использованием динамических данных C#	582
Взаимодействие с COM без использования динамических данных C#	585
Резюме	586
Глава 17. Процессы, домены приложений и объектные контексты	588
Роль процесса Windows	588
Роль потоков	589
Взаимодействие с процессами на платформе .NET	591
Перечисление выполняющихся процессов	593
Исследование конкретного процесса	594
Исследование набора потоков процесса	594
Исследование набора модулей процесса	596
Запуск и останов процессов программным образом	597
Управление запуском процесса с использованием класса <code>ProcessStartInfo</code>	598
Домены приложений .NET	599
Класс <code>System.AppDomain</code>	600
Взаимодействие со стандартным доменом приложения	601
Перечисление загруженных сборок	602
Получение уведомлений о загрузке сборок	604
Создание новых доменов приложений	604
Загрузка сборок в специальные домены приложений	606
Выгрузка доменов приложений программным образом	607
Контекстные границы объектов	608
Контекстно-свободные и контекстно-связанные типы	609
Определение контекстно-связанного объекта	609
Исследование контекста объекта	609
Итоговые сведения о процессах, доменах приложений и контекстах	611
Резюме	611
Глава 18. Язык CIL и роль динамических сборок	612
Причины для изучения грамматики языка CIL	612
Директивы, атрибуты и коды операций CIL	613
Роль директив CIL	614
Роль атрибутов CIL	614
Роль кодов операций CIL	614
Разница между кодами операций и их мнемоническими эквивалентами в CIL	615
Основанная на стеке природа CIL	615
Возвратное проектирование	617
Роль меток в коде CIL	620
Взаимодействие с CIL: модификация файла *.il	620
Компиляция CIL-кода с помощью <code>ilasm.exe</code>	622
Роль инструмента <code>peverify.exe</code>	623
Директивы и атрибуты CIL	623
Указание ссылок на внешние сборки в CIL	623
Определение текущей сборки в CIL	624
Определение пространств имен в CIL	624
Определение типов классов в CIL	625
Определение и реализация интерфейсов в CIL	626
Определение структур в CIL	627
Определение перечислений в CIL	627
Определение обобщений в CIL	627
Компиляция файла <code>CILTypes.il</code>	628

Соответствия между базовыми классами .NET, C# и CIL	629
Определение членов типов в CIL	629
Определение полей данных в CIL	630
Определение конструкторов типа в CIL	630
Определение свойств в CIL	631
Определение параметров членов	631
Исследование кодов операций CIL	632
Директива <code>.maxstack</code>	634
Объявление локальных переменных в CIL	634
Отображение параметров на локальные переменные в CIL	635
Скрытая ссылка <code>this</code>	636
Представление итерационных конструкций в CIL	636
Создание сборки .NET на CIL	637
Построение сборки <code>CILCars.dll</code>	637
Построение сборки <code>CILCarClient.exe</code>	639
Динамические сборки	641
Исследование пространства имен <code>System.Reflection.Emit</code>	642
Роль типа <code>System.Reflection.Emit.ILGenerator</code>	643
Выдача динамической сборки	644
Выдача сборки и набора модулей	645
Роль типа <code>ModuleBuilder</code>	646
Выдача типа <code>HelloClass</code> и строковой переменной-члена	647
Выдача конструкторов	648
Выдача метода <code>SayHello()</code>	649
Использование динамически сгенерированной сборки	649
Резюме	650
Часть VI. Введение в библиотеки базовых классов .NET	651
Глава 19. Многопоточное, параллельное и асинхронное программирование	652
Отношения между процессом, доменом приложения, контекстом и потоком	653
Проблема параллелизма	654
Роль синхронизации потоков	654
Краткий обзор делегатов .NET	655
Асинхронная природа делегатов	656
Методы <code>BeginInvoke()</code> и <code>EndInvoke()</code>	657
Интерфейс <code>System.IAsyncResult</code>	657
Асинхронный вызов метода	658
Синхронизация вызывающего потока	658
Роль делегата <code>AsyncCallback</code>	660
Роль класса <code>AsyncResult</code>	662
Передача и получение специальных данных состояния	663
Пространство имен <code>System.Threading</code>	664
Класс <code>System.Threading.Thread</code>	664
Получение статистики о текущем потоке выполнения	666
Свойство <code>Name</code>	666
Свойство <code>Priority</code>	667
Ручное создание вторичных потоков	668
Работа с делегатом <code>ThreadStart</code>	668
Работа с делегатом <code>ParametrizedThreadStart</code>	670
Класс <code>AutoResetEvent</code>	671
Потоки переднего плана и фоновые потоки	672

Проблемы параллелизма	673
Синхронизация с использованием ключевого слова lock языка C#	675
Синхронизация с использованием типа System.Threading.Monitor	677
Синхронизация с использованием типа System.Threading.Interlocked	678
Синхронизация с использованием атрибута [Synchronization]	679
Программирование с использованием обратных вызовов Timer	680
Пул потоков CLR	681
Параллельное программирование с использованием TPL	683
Пространство имен System.Threading.Tasks	683
Роль класса Parallel	683
Обеспечение параллелизма данных с помощью класса Parallel	684
Доступ к элементам пользовательского интерфейса во вторичных потоках	686
Класс Task	688
Обработка запроса на отмену	688
Обеспечение параллелизма задач с помощью класса Parallel	689
Запросы Parallel LINQ (PLINQ)	692
Выполнение запроса PLINQ	693
Отмена запроса PLINQ	694
Асинхронные вызовы в версии .NET 4.5	695
Знакомство с ключевыми словами async и await языка C#	695
Соглашения об именовании асинхронных методов	697
Асинхронные методы, возвращающие void	698
Асинхронные методы с множеством контекстов await	699
Модернизация примера AddWithThreads с использованием async/await	699
Резюме	701
Глава 20. Файловый ввод-вывод и сериализация объектов	702
Исследование пространства имен System.IO	702
Классы Directory (DirectoryInfo) и File (FileInfo)	703
Абстрактный базовый класс FileSystemInfo	704
Работа с типом DirectoryInfo	705
Перечисление файлов с помощью типа DirectoryInfo	706
Создание подкаталогов с помощью типа DirectoryInfo	707
Работа с типом Directory	708
Работа с типом DriveInfo	709
Работа с классом FileInfo	710
Метод FileInfo.Create()	711
Метод FileInfo.Open()	711
Методы FileOpen.OpenRead() и FileInfo.OpenWrite()	712
Метод FileInfo.OpenText()	713
Методы FileInfo.CreateText() и FileInfo.AppendText()	713
Работа с типом File	714
Дополнительные члены File	714
Абстрактный класс Stream	715
Работа с классом FileStream	716
Работа с классами StreamWriter и StreamReader	717
Запись в текстовый файл	718
Чтение из текстового файла	719
Прямое создание экземпляров классов StreamWriter/StreamReader	720
Работа с классами StringWriter и StringReader	721
Работа с классами BinaryWriter и BinaryReader	722
Программное отслеживание файлов	723

Понятие сериализации объектов	725
Роль графов объектов	727
Конфигурирование объектов для сериализации	728
Определение сериализируемых типов	728
Открытые поля, закрытые поля и открытые свойства	729
Выбор формatera сериализации	730
Интерфейсы IFormatter и IRemotingFormatter	730
Точность типов среди форматов	731
Сериализация объектов с использованием BinaryFormatter	732
Десериализация объектов с использованием BinaryFormatter	733
Сериализация объектов с использованием SoapFormatter	734
Сериализация объектов с использованием XmlSerializer	735
Управление генерацией данных XML	736
Сериализация коллекций объектов	737
Настройка процессов сериализации SOAP и двоичной сериализации	739
Углубленный взгляд на сериализацию объектов	739
Настройка сериализации с использованием ISerializable	740
Настройка сериализации с использованием атрибутов	743
Резюме	744
Глава 21. ADO.NET, часть I: подключенный уровень	745
Высокоуровневое определение ADO.NET	745
Три грани ADO.NET	746
Поставщики данных ADO.NET	747
Поставщики данных ADO.NET от Microsoft	749
О сборке System.Data.OracleClient.dll	750
Получение сторонних поставщиков данных ADO.NET	750
Дополнительные пространства имен ADO.NET	750
Типы из пространства имен System.Data	751
Роль интерфейса IDbConnection	752
Роль интерфейса IDbTransaction	752
Роль интерфейса IDbCommand	753
Роль интерфейсов IDbDataParameter и IDataParameter	753
Роль интерфейсов IDbDataAdapter и IDataAdapter	754
Роль интерфейсов IDataReader и IDataRecord	754
Абстрагирование поставщиков данных с помощью интерфейсов	755
Повышение гибкости с помощью конфигурационных файлов приложения	757
Создание базы данных AutoLot	758
Создание таблицы Inventory	758
Занесение тестовых записей в таблицу Inventory	760
Создание хранимой процедуры GetPetName ()	761
Создание таблиц Customers и Orders	761
Визуальное создание отношений между таблицами	763
Модель фабрик поставщиков данных ADO.NET	764
Полный пример фабрики поставщиков данных	765
Потенциальный недостаток модели фабрик поставщиков данных	767
Элемент <connectionStrings>	768
Понятие подключенного уровня ADO.NET	769
Работа с объектами подключения	770
Работа с объектами ConnectionStringBuilder	772
Работа с объектами команд	773

Работа с объектами чтения данных	774
Получение нескольких результирующих наборов с использованием объекта чтения данных	776
Создание многократно используемой библиотеки доступа к данным	776
Добавление логики подключения	778
Добавление логики вставки	778
Добавление логики удаления	779
Добавление логики обновления	780
Добавление логики выборки	780
Работа с параметризованными объектами команд	781
Выполнение хранимой процедуры	783
Создание приложения с консольным пользовательским интерфейсом	785
Реализация метода Main()	785
Реализация метода ShowInstructions()	787
Реализация метода ListInventory()	787
Реализация метода DeleteCar()	788
Реализация метода InsertNewCar()	788
Реализация метода UpdateCarPetName()	789
Реализация метода LookUpPetName()	789
Понятие транзакций базы данных	790
Основные члены объекта транзакции ADO.NET	791
Добавление таблицы CreditRisks в базу данных AutoLot	792
Добавление метода транзакции в InventoryDAL	792
Тестирование транзакции базы данных	793
Резюме	794
Глава 22. ADO.NET, часть II: автономный уровень	795
Понятие автономного уровня ADO.NET	795
Роль объектов DataSet	797
Основные свойства класса DataSet	797
Основные методы класса DataSet	797
Создание DataSet	798
Работа с объектами DataColumn	799
Построение объекта DataColumn	800
Включение автоинкрементных полей	801
Добавление объектов DataColumn в DataTable	801
Работа с объектами DataRow	802
Свойство RowState	803
Свойство DataRowVersion	804
Работа с объектами DataTable	805
Вставка объектов DataTable в DataSet	806
Получение данных из объекта DataSet	806
Обработка данных из DataTable с использованием объектов DataTableReader	807
Сериализация объектов DataTable и DataSet в формате XML	808
Сериализация объектов DataTable и DataSet в двоичном формате	810
Привязка объектов DataTable к графическим пользовательским интерфейсам	
Windows Forms	811
Заполнение DataTable из обобщенного List<T>	812
Удаление строк из DataTable	814
Выборка строк на основе критерия фильтрации	815
Обновление строк в DataTable	817
Работа с типом DataView	818

Работа с адаптерами данных	819
Простой пример адаптера данных	820
Отображение имен из базы данных на дружественные к пользователю имена	821
Добавление в AutoLotDAL.dll функциональности автономного уровня	822
Определение начального класса	822
Конфигурирование адаптера данных с использованием SqlCommandBuilder	823
Реализация метода GetAllInventory()	824
Реализация метода UpdateInventory()	824
Установка номера версии	825
Тестирование функциональности автономного уровня	825
Объекты DataSet с несколькими таблицами и отношения между данными	826
Подготовка адаптеров данных	827
Построение отношений между таблицами	828
Обновление таблиц базы данных	829
Навигация между связанными таблицами	829
Инструменты визуального конструирования баз данных Windows Forms	831
Визуальное проектирование элемента управления DataGridView	832
Сгенерированный файл App.config	835
Исследование строго типизированного класса DataSet	835
Исследование строго типизированного класса DataTable	836
Исследование строго типизированного класса DataRow	837
Исследование строго типизированного адаптера данных	837
Завершение приложения Windows Forms	838
Изоляция строго типизированного кода работы с базой данных в библиотеке классов	839
Просмотр сгенерированного кода	840
Выборка данных с помощью сгенерированного кода	841
Вставка данных с помощью сгенерированного кода	842
Удаление данных с помощью сгенерированного кода	843
Вызов хранимой процедуры с помощью сгенерированного кода	843
Программирование с помощью LINQ to DataSet	844
Роль библиотеки расширений DataSet	846
Получение объекта DataTable, совместимого с LINQ	846
Роль расширяющего метода DataRowExtensions.Field<T>()	848
Заполнение новых объектов DataTable из запросов LINQ	848
Резюме	849
Глава 23. ADO.NET, часть III: Entity Framework	850
Роль Entity Framework	850
Роль сущностей	852
Строительные блоки Entity Framework	854
Роль служб объектов	854
Роль клиента сущности	855
Роль файла *.edmx	856
Роль классовObjectContext и ObjectSet<T>	857
Собираем все вместе	858
Построение и анализ первой модели EDM	859
Генерация файла *.edmx	859
Изменение формы сущностных данных	862
Просмотр отображений	864
Просмотр сгенерированного файла *.edmx	865
Просмотр сгенерированного исходного кода	867
Улучшение сгенерированного исходного кода	869

Программирование с использованием концептуальной модели	869
Удаление записи	870
Обновление записи	871
Запросы с помощью LINQ to Entities	872
Запросы с помощью Entity SQL	873
Работа с объектом EntityDataReader	874
Проект AutoLotDAL версии 4, теперь с сущностями	875
Роль навигационных свойств	875
Использование навигационных свойств внутри запросов LINQ to Entity	877
Вызов хранимой процедуры	878
Привязка данных сущностей к графическим пользовательским интерфейсам	
Windows Forms	879
Добавление кода привязки данных	882
Продолжение изучения API-интерфейсов доступа к данным в .NET	882
Резюме	883
Глава 24. Введение в LINQ to XML	884
История о двух API-интерфейсах XML	884
Интерфейс LINQ to XML как лучшая модель DOM	886
Синтаксис литералов Visual Basic как наилучший интерфейс LINQ to XML	886
Члены пространства имен System.Xml.Linq	887
Осевые методы LINQ to XML	889
Избыточность XName (и XNamespace)	890
Работа с XElement и XDocument	891
Генерация документов из массивов и контейнеров	893
Загрузка и разбор XML-содержимого	894
Манипулирование XML-документом в памяти	894
Построение пользовательского интерфейса приложения LINQ to XML	895
Импорт файла Inventory.xml	895
Определение вспомогательного класса LINQ to XML	896
Связывание пользовательского интерфейса и вспомогательного класса	897
Резюме	898
Глава 25. Введение в Windows Communication Foundation	899
Собрание API-интерфейсов распределенных вычислений	899
Роль DCOM	900
Роль служб COM+/Enterprise Services	901
Роль MSMQ	902
Роль .NET Remoting	902
Роль веб-служб XML	903
Роль WCF	904
Обзор функциональных возможностей WCF	905
Обзор архитектуры, ориентированной на службы	905
WCF: заключение	906
Исследование основных сборок WCF	907
Шаблоны проектов WCF в Visual Studio	908
Шаблон проекта для построения веб-сайта со службами WCF	909
Базовая структура приложения WCF	909
Адрес, привязка и контракт в WCF	911
Понятие контрактов WCF	911
Понятие привязок WCF	912
Понятие адресов WCF	914

Построение службы WCF	916
Атрибут [ServiceContract]	917
Атрибут [OperationContract]	918
Служебные типы как контракты операций	918
Хостинг службы WCF	919
Установка ABC внутри файла App.config	920
Кодирование с использованием типа ServiceHost	921
Указание базовых адресов	921
Подробный анализ типа ServiceHost	922
Подробный анализ элемента <system.serviceModel>	924
Включение обмена метаданными	924
Построение клиентского приложения WCF	927
Генерация кода прокси с использованием svcutil.exe	927
Генерация кода прокси с использованием Visual Studio	928
Конфигурирование привязки на основе TCP	930
Упрощение конфигурационных настроек	931
Использование стандартных конечных точек	931
Открытие одной службы WCF, использующей множество привязок	932
Изменение параметров привязки WCF	934
Использование конфигурации стандартного поведения MEX	935
Обновление клиентского прокси и выбор привязки	936
Использование шаблона проекта WCF Service Library	937
Построение простой математической службы	938
Тестирование службы WCF с помощью WcfTestClient.exe	938
Изменение конфигурационных файлов с помощью SvcConfigEditor.exe	939
Хостинг службы WCF внутри Windows-службы	940
Указание ABC в коде	941
Включение MEX	943
Создание программы установки для Windows-службы	943
Установка Windows-службы	944
Асинхронный вызов службы из клиента	945
Проектирование контрактов данных WCF	947
Использование веб-ориентированного шаблона проекта WCF Service	948
Реализация контракта службы	949
Роль файла *.svc	951
Содержимое файла Web.config	951
Тестирование службы	951
Резюме	952
Глава 26. Введение в Windows Workflow Foundation	953
Определение бизнес-процесса	953
Роль WF	954
Построение простого рабочего потока	954
Исполняющая среда рабочих потоков	957
Хостинг рабочего потока с использованием класса WorkflowInvoker	957
Хостинг рабочего потока с использованием класса WorkflowApplication	960
Итоги по первому рабочему потоку	961
Знакомство с действиями рабочих потоков	961
Действия потока управления	962
Действия блок-схемы	962
Действия обмена сообщениями	963
Действия конечного автомата	963
Действия исполняющей среды и действия-примитивы	964

Действия транзакций	964
Действия над коллекциями и действия обработки ошибок	965
Построение рабочего потока в виде блок-схемы	965
Подключение действий к блок-схеме	966
Работа с действием InvokeMethod	967
Определение переменных уровня рабочего потока	968
Работа с действием FlowDecision	969
Работа с действием TerminateWorkflow	970
Построение условия True	970
Работа с действием ForEach<T>	971
Завершение приложения	973
Промежуточные итоги	974
Построение последовательного рабочего потока (в выделенной библиотеке)	975
Определение начального проекта	975
Импорт сборок и пространств имен	976
Определение аргументов рабочего потока	977
Определение переменных рабочего потока	978
Работа с действием Assign	979
Работа с действиями If и Switch	980
Построение специального действия кода	982
Использование библиотеки рабочего потока	984
Извлечение выходного аргумента рабочего потока	985
Резюме	986
Часть VII. Windows Presentation Foundation	987
Глава 27. Введение в Windows Presentation Foundation и XAML	988
Мотивация, лежащая в основе WPF	988
Унификация различных API-интерфейсов	989
Обеспечение разделения ответственности через XAML	990
Обеспечение оптимизированной модели визуализации	990
Упрощение программирования сложных пользовательских интерфейсов	991
Различные варианты приложений WPF	992
Традиционные настольные приложения	992
WPF-приложения на основе навигации	994
Приложения XBAP	994
Отношения между WPF и Silverlight	995
Исследование сборок WPF	996
Роль класса Application	998
Роль класса Window	999
Роль класса System.Windows.Controls.ContentControl	999
Роль класса System.Windows.Controls.Control	1001
Роль класса System.Windows.FrameworkElement	1002
Роль класса System.Windows.UIElement	1002
Роль класса System.Windows.Media.Visual	1003
Роль класса System.Windows.DependencyObject	1003
Роль класса System.Windows.Threading.DispatcherObject	1003
Построение приложения WPF без XAML	1003
Создание строго типизированного окна	1005
Создание простого пользовательского интерфейса	1006
Взаимодействие с данными уровня приложения	1007
Обработка закрытия объекта Window	1008
Перехват событий мыши	1009
Перехват клавиатурных событий	1010

Построение приложения WPF с использованием только XAML	1011
Определение объекта Window в XAML	1012
Определение объекта Application в XAML	1014
Обработка файлов XAML с помощью msbuild.exe	1014
Трансформация разметки в сборку .NET	1016
Отображение XAML-разметки окна на код C#	1016
Роль BAML	1018
Отображение XAML-разметки приложения на код C#	1019
Итоговые замечания о процессе трансформирования XAML в сборку	1020
Синтаксис XAML для WPF	1020
Введение в Xaml	1020
Пространства имен XAML XML и "ключевые слова" XAML	1022
Управление видимостью классов и переменных-членов	1024
Элементы XAML, атрибуты XAML и преобразователи типов	1024
Понятие синтаксиса "свойство-элемент" в XAML	1026
Понятие присоединяемых свойств XAML	1026
Понятие расширений разметки XAML	1027
Построение приложений WPF с использованием файлов отделенного кода	1029
Добавление файла кода для класса MainWindow	1029
Добавление файла кода для класса MyApp	1030
Обработка файлов кода с помощью msbuild.exe	1031
Построение приложений WPF с использованием Visual Studio	1031
Шаблоны проектов WPF	1032
Панель инструментов и визуальный конструктор/редактор XAML	1032
Установка свойств с использованием окна Properties	1034
Обработка событий с использованием окна Properties	1034
Обработка событий в редакторе XAML	1035
Окно Document Outline	1036
Просмотр автоматически сгенерированных файлов кода	1036
Построение специального редактора XAML с помощью Visual Studio	1036
Проектирование графического пользовательского интерфейса окна	1037
Реализация события Loaded	1038
Реализация события Click объекта Button	1039
Реализация события Closed	1040
Тестирование приложения	1040
Изучение документации WPF	1041
Резюме	1042
Глава 28. Программирование с использованием элементов управления WPF	1043
Обзор основных элементов управления WPF	1043
Элементы управления Ink API	1044
Элементы управления документов WPF	1045
Общие диалоговые окна WPF	1045
Подробные сведения находятся в документации	1045
Краткий обзор визуального конструктора WPF в Visual Studio	1046
Работа с элементами управления WPF в Visual Studio	1047
Работа с редактором Document Outline	1047
Управление компоновкой содержимого с использованием панелей	1048
Позиционирование содержимого внутри панелей Canvas	1051
Позиционирование содержимого внутри панелей WrapPanel	1052
Позиционирование содержимого внутри панелей StackPanel	1053
Позиционирование содержимого внутри панелей Grid	1054
Позиционирование содержимого внутри панелей DockPanel	1057

Включение прокрутки в типах панелей	1058
Конфигурирование панелей с использованием визуальных конструкторов Visual Studio	1058
Построение окна с использованием вложенных панелей	1061
Построение системы меню	1062
Построение панели инструментов	1065
Построение строки состояния	1066
Завершение проектирования пользовательского интерфейса	1066
Реализация обработчиков событий MouseEnter/MouseLeave	1067
Реализация логики проверки правописания	1067
Понятие команд WPF	1068
Объекты внутренних команд	1068
Подключение команд к свойству Command	1069
Подключение команд к произвольным действиям	1070
Работа с командами Open и Save	1071
Более глубокий взгляд на API-интерфейсы и элементы управления WPF	1073
Работа с элементом управления TabControl	1074
Построение вкладки Ink API	1075
Проектирование панели инструментов	1076
Элемент управления RadioButton	1078
Обработка событий для вкладки Ink API	1078
Элемент управления InkCanvas	1080
Элемент управления ComboBox	1082
Сохранение, загрузка и очистка данных InkCanvas	1084
Введение в интерфейс Documents API	1085
Блочные элементы и встроенные элементы	1085
Диспетчеры компоновки документа	1085
Построение вкладки Documents	1086
Наполнение FlowDocument с помощью кода	1087
Включение аннотаций и “клеяких” заметок	1088
Сохранение и загрузка потокового документа	1089
Введение в модель привязки данных WPF	1091
Построение вкладки Data Binding	1091
Установка привязки данных с использованием Visual Studio	1092
Свойство DataContext	1093
Преобразование данных с использованием IValueConverter	1094
Установка привязок данных в коде	1095
Построение вкладки DataGrid	1096
Резюме	1097
Глава 29. Службы визуализации графики WPF	1098
Службы графической визуализации WPF	1098
Варианты графической визуализации WPF	1099
Визуализация графических данных с использованием фигур	1100
Добавление прямоугольников, эллипсов и линий на поверхность Canvas	1102
Удаление прямоугольников, эллипсов и линий с поверхности Canvas	1105
Работа с элементами Polyline и Polygon	1106
Работа с элементом Path	1107
Кисти и перья WPF	1110
Конфигурирование кистей с использованием Visual Studio	1111
Конфигурирование кистей в коде	1112
Конфигурирование перьев	1113

Применение графических трансформаций	1114
Первый взгляд на трансформации	1115
Трансформация данных Canvas	1115
Работа с редактором трансформаций Visual Studio	1118
Построение начальной компоновки	1118
Применение трансформаций на этапе проектирования	1119
Трансформация холста в коде	1120
Визуализация графических данных с использованием рисунков и геометрий	1121
Построение кисти DrawingBrush с использованием геометрий	1122
Рисование с помощью DrawingBrush	1123
Включение типов Drawing в DrawingImage	1124
Роль инструмента Expression Design	1125
Экспорт файла с примером графики в виде XAML	1125
Импорт графических данных в проект WPF	1127
Взаимодействие с объектами изображения	1128
Визуализация графических данных с использованием визуального уровня	1129
Базовый класс Visual и производные дочерние классы	1129
Первый взгляд на класс DrawingVisual	1130
Визуализация графических данных в специальном диспетчере компоновки	1132
Реагирование на операции проверки попадания	1133
Резюме	1135
Глава 30. Ресурсы, анимация и стили WPF	1136
Система ресурсов WPF	1136
Работа с двоичными ресурсами	1137
Программная загрузка изображения	1139
Работа с объектными (логическими) ресурсами	1142
Роль свойства Resources	1142
Определение ресурсов уровня окна	1142
Расширение разметки {StaticResource}	1145
Расширение разметки {DynamicResource}	1145
Ресурсы уровня приложения	1146
Определение объединенных словарей ресурсов	1147
Определение сборки, включающей только ресурсы	1148
Службы анимации WPF	1150
Роль классов анимации	1150
Свойства To, From и By	1151
Роль базового класса Timeline	1152
Реализация анимации в коде C#	1152
Управление темпом анимации	1153
Запуск в обратном порядке и циклическое выполнение анимации	1154
Реализация анимации в разметке XAML	1155
Роль раскадровок	1156
Роль триггеров событий	1156
Анимация с использованием дискретных ключевых кадров	1157
Роль стилей WPF	1158
Определение и применение стиля	1158
Переопределение настроек стиля	1159
Автоматическое применение стиля с помощью TargetType	1159
Создание подклассов существующих стилей	1160
Роль неименованных стилей	1160
Определение стилей с триггерами	1161
Определение стилей с несколькими триггерами	1162

Анимированные стили	1162
Применение стилей в коде	1163
Резюме	1164
Глава 31. Свойства зависимости, маршрутизируемые события и шаблоны	1165
Роль свойств зависимости	1165
Знакомство с существующим свойством зависимости	1167
Важные замечания относительно оболочек свойств CLR	1170
Построение специального свойства зависимости	1170
Добавление процедуры проверки достоверности данных	1174
Реагирование на изменение свойства	1175
Маршрутизируемые события	1176
Роль маршрутизируемых пузырьковых событий	1177
Продолжение или прекращение пузырькового распространения	1177
Роль маршрутизируемых туннельных событий	1178
Логические деревья, визуальные деревья и стандартные шаблоны	1180
Программный просмотр логического дерева	1180
Программный просмотр визуального дерева	1181
Программный просмотр стандартного шаблона элемента управления	1183
Построение специального шаблона элемента управления с помощью инфраструктуры триггеров	1186
Шаблоны как ресурсы	1187
Встраивание визуальных подсказок с использованием триггеров	1188
Роль расширения разметки {TemplateBinding}	1189
Роль класса ContentPresenter	1190
Включение шаблонов в стили	1191
Резюме	1192
Часть VIII. ASP.NET Web Forms	1193
Глава 32. Введение в ASP.NET Web Forms	1194
Роль протокола HTTP	1194
Цикл запрос/ответ HTTP	1195
HTTP — протокол без хранения состояния	1195
Веб-приложения и веб-серверы	1195
Роль виртуальных каталогов IIS	1196
Веб-сервер разработки ASP.NET	1197
Роль языка HTML	1197
Структура HTML-документа	1198
Роль форм HTML	1199
Инструменты визуального конструктора HTML в Visual Studio	1199
Построение HTML-формы	1200
Роль сценариев клиентской стороны	1202
Пример сценария клиентской стороны	1203
Обратная отправка веб-серверу	1204
Обратные отправки в ASP.NET	1205
Обзор API-интерфейса ASP.NET	1205
Основные функциональные возможности ASP.NET 2.0 и последующих версий	1206
Основные функциональные возможности ASP.NET 3.5 (и .NET 3.5 SP1) и последующих версий	1207
Основные функциональные возможности ASP.NET 4.0 и 4.5	1208
Построение однофайловой веб-страницы ASP.NET	1208
Указание сборки AutoLotDAL.dll	1210

Проектирование пользовательского интерфейса	1210
Добавление логики доступа к данным	1211
Роль директив ASP.NET	1213
Анализ блока script	1214
Анализ объявлений элементов управления ASP.NET	1215
Построение веб-страницы ASP.NET с использованием файлов кода	1216
Ссылка на сборку AutoLotDAL.dll	1218
Изменение файла кода	1218
Отладка и трассировка страниц ASP.NET	1219
Сравнение веб-сайтов и веб-приложений ASP.NET	1220
Структура каталогов веб-сайта ASP.NET	1222
Ссылка на сборки	1222
Роль папки App_Code	1223
Цепочка наследования для типа Page	1223
Взаимодействие с входящим HTTP-запросом	1225
Получение статистики о браузере	1226
Доступ к входным данным формы	1227
Свойство IsPostBack	1227
Взаимодействие с исходящим HTTP-ответом	1228
Выдача HTML-содержимого	1229
Перенаправление пользователей	1229
Жизненный цикл веб-страницы ASP.NET	1230
Роль атрибута AutoEventWireup	1231
Событие Error	1231
Роль файла web.config	1233
Утилита администрирования веб-сайтов ASP.NET	1234
Резюме	1234
Глава 33. Веб-элементы управления, мастер-страницы и темы ASP.NET	1235
Природа веб-элементов управления	1235
Обработка событий серверной стороны	1236
Свойство AutoPostBack	1237
Базовые классы Control и WebControl	1238
Перечисление содержащихся элементов управления	1238
Динамическое добавление и удаление элементов управления	1241
Взаимодействие с динамически созданными элементами управления	1242
Функциональность базового класса WebControl	1243
Основные категории веб-элементов управления ASP.NET	1243
Несколько слов о пространстве имен System.Web.UI.HtmlControls	1245
Документация по веб-элементам управления	1245
Построение примера веб-сайта ASP.NET	1245
Работа с мастер-страницами	1246
Определение стандартной страницы содержимого Default.aspx	1252
Проектирование страницы содержимого Inventory.aspx	1254
Проектирование страницы содержимого BuildCar.aspx	1258
Роль элементов управления проверкой достоверности	1261
Включение поддержки проверки достоверности с помощью JavaScript на стороне клиента	1263
Элемент управления RequiredFieldValidator	1263
Элемент управления RegularExpressionValidator	1264
Элемент управления RangeValidator	1264
Элемент управления CompareValidator	1264

Создание итоговой панели проверки достоверности	1265
Определение групп проверки достоверности	1266
Работа с темами	1268
Файлы *.skin	1269
Применение тем ко всему сайту	1271
Применение тем на уровне страницы	1271
Свойство SkinID	1271
Программное назначение тем	1272
Резюме	1273
Глава 34. Управление состоянием в ASP.NET	1274
Проблема поддержки состояния	1274
Приемы управления состоянием ASP.NET	1276
Роль состояния представления ASP.NET	1277
Демонстрация работы с состоянием представления	1277
Добавление специальных данных в состояние представления	1279
Роль файла Global.asax	1279
Глобальный обработчик исключений “последнего шанса”	1281
Базовый класс <code>HttpApplication</code>	1282
Различие между состоянием приложения и состоянием сеанса	1282
Поддержка данных состояния уровня приложения	1282
Модификация данных приложения	1285
Обработка останова веб-приложения	1286
Работа с кешем приложения	1286
Использование кеширования данных	1287
Модификация файла *.aspx	1289
Поддержка данных сеанса	1291
Дополнительные члены класса <code>HttpSessionState</code>	1293
Cookie-наборы	1294
Создание cookie-наборов	1295
Чтение входящих cookie-данных	1296
Роль элемента <code><sessionState></code>	1297
Хранение данных сеанса на сервере состояния сеансов ASP.NET	1297
Хранение информации о сеансах в выделенной базе данных	1298
Введение в API-интерфейс ASP.NET Profile	1299
База данных <code>ASPNETDB.mdf</code>	1299
Определение пользовательского профиля в <code>web.config</code>	1300
Программный доступ к данным профиля	1301
Группирование данных профиля и сохранение специальных объектов	1303
Резюме	1305
Предметный указатель	1306

Я посвящаю эту книгу целиком и полностью двум самым
важным людям в моей жизни: жене Мэнди
и сыну Сорену Вейду Троелсену.
Не хватит слов, чтобы выразить мою любовь к вам.

Об авторе

Эндрю Троелсен с любовью вспоминает свой самый первый компьютер Atari 400, оснащенный кассетным устройством хранения и черно-белым телевизором, служащим в качестве монитора (который его родители разрешили ему поставить у себя в спальне, за что им большое спасибо).

В настоящее время Эндрю работает в Intertech (www.intertech.com) — центре обучения и консалтинга по .NET/Java, который находится в Миннеаполисе, шт. Миннесота.

Он является автором многочисленных книг, в числе которых *Developer's Workshop to COM and ATL 3.0* (Wordware Publishing, 2000 г.), *COM and .NET Interoperability* (Apress, 2002 г.) и *Visual Basic 2008 and the .NET 3.5 Platform: An Advanced Guide* (Apress, 2008 г.).

О техническом редакторе

Энди Олсен возглавляет компанию по обучению разработке программного обеспечения, находящуюся в Великобритании, которая проводит обучение .NET, Java, веб- и мобильным технологиям в Европе, США и Азии. Энди работает с .NET, начиная с самой первой бета-версии этой платформы, и занимается активным исследованием новых функциональных возможностей, которые появились в .NET 4.5. Он живет у моря в городе Суонси вместе со своей женой Джейн и детьми Эмили и Томом. Любит делать пробежки вдоль побережья (регулярно останавливаясь на чашечку кофе по пути), кататься на лыжах и наблюдать за лебедями. Связаться с ним можно по адресу:

andyo@olsensoft.com

Благодарности

Может показаться, что обновить существующую книгу проще, чем написать совершенно новую книгу с нуля. Однако, как говорит мой опыт, все обстоит с точностью до наоборот. Хотя я один отвечаю за общее содержание, эта книга никогда не была бы опубликована без вклада тех людей, которые неустанно работали вместе со мной.

Огромная благодарность моему техническому редактору Энди Олсену. Энди, как всегда, внес множество великолепных предложений (я только жалею, что не успел реализовать их все, но, может быть, это получится сделать в следующем издании).

Спасибо персоналу из Apress: вы продолжаете мне доказывать, почему я с удовольствием работаю с вами. Благодарю всех тех, кто приложил усилия по превращению моей сырой рукописи в профессиональный, высококачественный и пригодный к публикации текст.

Введение

Много лет тому назад (примерно в 2001 г.) я получил возможность написать книгу по предстоящей технологии Microsoft, которая на то время называлась NGWS (Next Generation Windows Software — программное обеспечение Windows нового поколения). Когда я начал исследовать исходный код, предоставленный Microsoft, то отметил многократные ссылки на язык программирования COOL (Common Object Oriented Language — общий объектно-ориентированный язык).

Пока я работал над своей первой рукописью книги *C# and the .NET Platform*, пользуясь предварительной альфа-сборкой (разумеется, без какой-либо документации), NGWS в конечном счете была переименована в платформу Microsoft .NET. И, как вы наверняка догадались, язык COOL нам сегодня известен под названием C#.

Первое издание этой книги вышло одновременно с версией .NET 1.0, бета 2. С тех пор я переписывал текст для учета многочисленных обновлений языка программирования C#, а также бурного появления новых API-интерфейсов в каждом новом выпуске платформы .NET.

С годами эта книга была очень хорошо принята прессой (финалист премии JOLT и книга года по программированию ReferenceWare), читателями и различными университетскими программами по вычислительной технике и разработке программного обеспечения.

Было просто замечательно общаться с читателями и преподавателями по всему миру. Спасибо вам всем за предложения, комментарии и (естественно) критику. Возможно, я не в состоянии ответить на каждое почтовое сообщение, но будьте уверены, что все они принимаются во внимание.

Автор и читатели — одна команда

Авторам книг по технологиям приходится писать для очень требовательной группы людей (я не могу не знать это, т.к. я один из них). Всем известно, что разработка программных решений с помощью любой платформы или языка очень сложна и сильно зависит от отдела, компании, клиентской базы и поставленной задачи. Кто-то работает в сфере электронных публикаций, кто-то занимается разработкой систем для правительства и региональных органов власти, а кто-то сотрудничает с NASA или военными отраслями. Что касается меня, то я занимаюсь разработкой обучающего программного обеспечения для детей (Oregon Trail/Amazon Trail), различных многоуровневых систем и проектов в медицинской и финансовой сфере. Это значит, что код, который придется писать вам, скорее всего, будет иметь мало общего с кодом, с которым приходится иметь дело мне.

Таким образом, в этой книге я специально стараюсь избегать создания демонстраций, свойственных только конкретной отрасли или направлению программирования. Учитывая это, язык C#, объектно-ориентированное программирование, среда CLR и библиотеки базовых классов .NET объясняются на примерах, не привязанных к отрасли. В частности, здесь везде применяется одна и та же тема, так или иначе, близкая каждому — автомобили.

Моя работа, как автора, состоит в том, чтобы максимально доступно объяснить вам язык программирования C# и основные концепции платформы .NET. Кроме того, я также буду делать все возможное для снабжения вас инструментами и стратегиями, которые могут потребоваться для продолжения обучения по прочтении настоящей книги.

Ваша работа, как читателя, заключается в том, чтобы усвоить всю эту информацию и научиться применять ее на практике при разработке своих программных решений. Конечно, скорее всего, проекты, которые понадобятся вам выполнять в будущем, не будут связаны с автомобилями и их дружественными именами, но именно в этом и состоит вся суть прикладных знаний.

После изучения представленных в этой книге тем и концепций вы сможете успешно строить решения .NET, удовлетворяющие требованиям конкретной среды программирования.

Краткий обзор книги

Эта книга логически разделена на восемь частей, каждая из которых содержит несколько взаимосвязанных между собой глав. Ниже приведено краткое содержание каждой из частей и глав настоящей книги.

Часть I. Введение в C# и платформу .NET

Часть I этой книги предназначена для ознакомления с природой платформы .NET и различными инструментами разработки (многие из которых распространяются с открытым исходным кодом), которые используются при построении приложений .NET.

Глава 1. Философия .NET

Первая глава выступает в качестве основы для всего остального материала. Главная ее цель заключается в том, чтобы ознакомить вас с набором строительных блоков .NET, таких как общезыковая исполняющая среда (Common Language Runtime — CLR), общая система типов (Common Type System — CTS), общезыковая спецификация (Common Language Specification — CLS) и библиотеки базовых классов. Здесь вы получите первоначальное представление о языке программирования C# и формате сборки .NET. Также вы узнаете о роли платформы .NET в рамках операционной системы Windows 8 и поймете разницу между приложением Windows 8 и приложением .NET.

Глава 2. Создание приложений на языке C#

Целью этой главы является ознакомление с процессом компиляции файлов исходного кода C# с применением различных средств и приемов. В начале главы будет показано, как использовать компилятор командной строки (`csc.exe`) и файлы ответов. Затем вы узнаете о многочисленных редакторах кода и интегрированных средах разработки, включая Notepad++, SharpDevelop, Visual C# Express и Visual Studio. Кроме того, будет показано, как установить на машине разработки локальную копию документации .NET Framework 4.5 SDK.

Часть II. Основы программирования на C#

Темы, представленные в этой части книги, исключительно важны, поскольку подходят для разработки приложений .NET любого типа (веб-приложений, настольных приложений с графическим пользовательским интерфейсом, библиотек кода или служб Windows). Здесь вы ознакомитесь с фундаментальными типами данных .NET, научитесь манипулировать текстом и узнаете о роли разнообразных модификаторов параметров C# (включая необязательные и именованные аргументы).

Глава 3. Главные конструкции программирования на C#: часть I

В этой главе начинается формальное изучение языка программирования C#. Здесь вы узнаете о роли метода `Main()` и многочисленных деталях, касающихся внутренних типов данных .NET, включая манипулирование текстовыми данными с использованием `System.String` и `System.Text.StringBuilder`. Кроме того, будут описаны итерационные конструкции и конструкции принятия решений, сужающие и расширяющие операции, а также ключевое слово `unchecked`.

Глава 4. Главные конструкции программирования на C#: часть II

В этой главе завершается рассмотрение ключевых аспектов C#. Будет показано, как создавать перегруженные методы типов и определять параметры с использованием ключевых слов `out`, `ref` и `params`. Также рассматриваются два средства C#: *именованные* и *необязательные параметры*. Кроме того, будет описано создание и манипулирование массивами данных, определение типов, допускающих `null` (и операций `?` и `??`), и показаны отличия между *типами значений* (включающими перечисления и специальные структуры) и *ссылочными типами*.

Часть III. Объектно-ориентированное программирование на C#

В этой части вы освоите ключевые конструкции языка C#, в том числе *объектно-ориентированное программирование* (ООП). Здесь также будет показано, как обрабатывать исключения времени выполнения, и каким образом работать со строго типизированными интерфейсами.

Глава 5. Инкапсуляция

В этой главе начинается рассмотрение концепций объектно-ориентированного программирования на языке C#. После введения в основные принципы ООП (инкапсуляция, наследование и полиморфизм) будет показано, как создавать надежные типы классов с применением конструкторов, свойств, статических членов, констант и полей, предназначенных только для чтения. Глава завершается объяснением частичных определений типов, синтаксиса инициализации объектов и автоматических свойств.

Глава 6. Понятие наследования и полиморфизма

Здесь вы ознакомитесь с оставшимися двумя основными принципами ООП — наследованием и полиморфизмом, — которые позволяют строить семейства связанных типов классов. Вы узнаете о роли виртуальных и абстрактных методов (а также абстрактных базовых классов) и о природе полиморфных интерфейсов. И, наконец, в главе будет рассмотрена роль главного базового класса платформы .NET — `System.Object`.

Глава 7. Структурированная обработка исключений

В этой главе рассматривается решение проблемы аномалий, возникающих в коде во время выполнения, за счет применения *структурированной обработки исключений*. Здесь описаны ключевые слова, предусмотренные для этого в C# (`try`, `catch`, `throw` и `finally`), а также отличия между исключениями уровня приложения и уровня системы. Кроме того, будут представлены различные инструменты в рамках Visual Studio, которые предназначены для отладки исключений, упущенных из виду.

Глава 8. Работа с интерфейсами

Материал этой главы предполагает наличие понимания концепций объектно-ориентированной разработки и посвящен *программированию на основе интерфейсов*. Здесь будет показано, как определять классы и структуры, поддерживающие множество по-

ведений, как обнаруживать эти поведения во время выполнения и как выборочно скрывать какие-то из них за счет явной реализации интерфейсов. В дополнение к созданию специальных интерфейсов, рассматриваются вопросы реализации стандартных интерфейсов из состава .NET и их применения для построения объектов, которые могут сортироваться, копироваться, перечисляться и сравниваться.

Часть IV. Дополнительные конструкции программирования на C#

В этой части книги вы получите возможность углубить знания языка C# за счет изучения других более сложных (но очень важных) концепций. Здесь завершается ознакомление с системой типов .NET описанием интерфейсов и делегатов. Кроме того, будет рассмотрена роль обобщений, дано краткое введение в язык LINQ (Language Integrated Query) и представлены некоторые более сложные функциональные возможности C# (такие как методы расширения, частичные методы и манипулирование указателями).

Глава 9. Коллекции и обобщения

Эта глава посвящена обобщениям. Вы увидите, что программирование с использованием обобщений позволяет создавать типы и члены типов, содержащие заполнители, которые заполняются вызывающим кодом. В целом, обобщения позволяют значительно улучшить производительность приложений и безопасность в отношении типов. В главе не только рассматриваются различные обобщенные типы из пространства имен `System.Collections.Generic`, но также показано, как строить собственные обобщенные методы и типы (с ограничениями и без).

Глава 10. Делегаты, события и лямбда-выражения

В этой главе вы узнаете, что собой представляет тип *делегата*. Делегат .NET — это объект, который *указывает* на другие методы в приложении. С помощью делегатов можно создавать системы, позволяющие многочисленным объектам взаимодействовать между собой двухсторонним образом. После изучения способов применения делегатов в .NET будет показано, как использовать ключевое слово `event` в C#, которое упрощает манипулирование делегатами. Кроме того, рассматривается роль лямбда-операции (`=>`) в C# и связь между делегатами, анонимными методами и лямбда-выражениями.

Глава 11. Расширенные средства языка C#

В этой главе описаны расширенные средства языка C#, в том числе перегрузка операций и создание специальных процедур преобразования (явных и неявных) для типов. Кроме того, вы узнаете, как строить и взаимодействовать с *индексаторами типов* и работать с *расширяющими методами*, *анонимными типами*, *частичными методами* и указателями C#, используя контекст кода `unsafe`.

Глава 12. LINQ to Objects

В этой главе начинается рассмотрение языка интегрированных запросов LINQ (Language Integrated Query). Язык LINQ позволяет создавать строго типизированные выражения запросов, которые могут применяться к многочисленным целевым объектам LINQ для манипулирования данными в самом широком смысле этого слова. Глава посвящена API-интерфейсу LINQ to Objects, который позволяет применять выражения LINQ к контейнерам данных (например, массивам, коллекциям и специальным типам). Эта информация будет полезна позже при рассмотрении других дополнительных API-интерфейсов, таких как LINQ to XML, LINQ to DataSet, PLINQ и LINQ to Entities.

Глава 13. Время жизни объектов

В финальной главе этой части объясняется, как среда CLR управляет памятью, используя сборщик мусора .NET. Вы узнаете о роли корневых элементов приложения, поколениях объектов и типе `System.GC`. После представления основ рассматриваются темы *освобождаемых объектов* (реализующих интерфейс `IDisposable`) и процесса финализации (с помощью метода `System.Object.Finalize()`). В главе также описан класс `Lazy<T>`, позволяющий определять данные, которые не будут размещаться вплоть до поступления запроса от вызывающего кода. Вы увидите, что эта возможность очень полезна, когда нежелательно загромождать кучу объектами, которые в текущий момент программе не нужны.

Часть V. Программирование с использованием сборок .NET

Эта часть книги посвящена деталям формата сборок .NET. Здесь вы узнаете не только о способах развертывания и конфигурирования библиотек кода .NET, но также о внутреннем устройстве двоичного образа .NET. Будет описана роль атрибутов .NET и определения информации о типе во время выполнения. Кроме того, рассматривается роль среды DLR (Dynamic Language Runtime — исполняющая среда динамического языка) и ключевого слова `dynamic` в C#. Наконец, объясняются более сложные темы, касающиеся сборок, такие как домены приложений, синтаксис языка CIL и построение сборок в памяти.

Глава 14. Построение и конфигурирование библиотек классов

На самом высоком уровне термин *сборка* используется для описания любого двоичного файла `*.dll` или `*.exe`, созданного с помощью компилятора .NET. Однако в действительности понятие сборки намного шире. В этой главе будет показано, чем отличаются однофайловые и многофайловые сборки, как создавать и развертывать сборки обеих разновидностей, как делать сборки закрытыми и разделяемыми с помощью XML-файлов `*.config` и специальных сборок политик издателя. Кроме того, в главе описана внутренняя структура глобального кеша сборок (Global Assembly Cache — GAC).

Глава 15. Рефлексия типов, позднее связывание и программирование с использованием атрибутов

В главе 15 продолжается изучение сборок .NET. Здесь будет показано, как обнаруживать типы во время выполнения с использованием пространства имен `System.Reflection`. Типы из этого пространства имен позволяют строить приложения, способные считывать метаданные сборки на лету. Кроме того, в главе рассматривается динамическая загрузка и создание типов во время выполнения с применением *позднего связывания*, а также роль атрибутов .NET (стандартных и специальных). Для закрепления материала в конце главы приводится пример построения расширяемого приложения Windows Forms.

Глава 16. Динамические типы и среда DLR

В версии .NET 4.0 появился новый аспект исполняющей среды .NET, который называется *исполняющей средой динамического языка*. Используя DLR и ключевое слово `dynamic`, введенное в C# 2010, можно определять данные, которые не будут распознаваться вплоть до времени выполнения. Такие возможности существенно упрощают решение некоторых очень сложных задач программирования для .NET. В этой главе вы ознакомитесь с рядом практических применений динамических данных, включая использование API-интерфейсов рефлексии .NET и взаимодействие с унаследованными библиотеками COM с минимальными усилиями.

Глава 17. Процессы, домены приложений и объектные контексты

В этой главе подробно рассказывается о внутреннем устройстве загруженной исполняемой сборки .NET. Цель главы заключается в иллюстрации отношений между процессами, доменами приложений и контекстными границами. Все эти темы подготавливают базу для изучения процесса создания многопоточных приложений в главе 19.

Глава 18. Язык CIL и роль динамических сборок

Цель этой финальной главы в данной части двояка. В первой половине главы рассматривается синтаксис и семантика языка CIL, а во второй — роль пространства имен `System.Reflection.Emit`. Типы из этого пространства имен можно использовать для построения программного обеспечения, которое способно генерировать сборки .NET в памяти во время выполнения. Формально сборки, которые определяются и выполняются в памяти, называются *динамическими сборками*.

Часть VI. Введение в библиотеки базовых классов .NET

К этому моменту вы уже должны хорошо знать основы языка C# и формат сборок .NET. Эта часть расширяет ваши знания исследованием ряда часто используемых служб, поставляемых в составе библиотек базовых классов .NET, включая создание многопоточных приложений, файловый ввод-вывод и доступ к базам данных с помощью ADO.NET. Здесь показано, как создавать распределенные приложения с применением Windows Communication Foundation (WCF) и приложения с рабочими потоками, которые используют API-интерфейс Windows Workflow Foundation (WF), а также описан API-интерфейс LINQ to XML.

Глава 19. Многопоточное, параллельное и асинхронное программирование

Эта глава посвящена построению многопоточных приложений. В ней демонстрируется ряд приемов, которые можно применять для написания кода, безопасного в отношении потоков. Глава начинается с краткого напоминания о том, что собой представляет тип делегата в .NET, и объяснения внутренней поддержки делегата для асинхронного вызова методов. Затем рассматриваются типы в пространстве имен `System.Threading`, а также библиотека параллельных задач (Task Parallel Library — TPL). С применением TPL разработчики .NET могут строить приложения, которые распределяют рабочую нагрузку по всем доступным центральным процессорам в исключительно простой манере. В главе также описана роль API-интерфейса PINQ (Parallel LINQ), который предлагает способ создания запросов LINQ, масштабируемых среди множества процессорных ядер. В завершение главы рассматриваются некоторые новые ключевые слова C#, появившиеся в .NET 4.5, которые интегрируют асинхронные вызовы методов непосредственно в язык.

Глава 20. Файловый ввод-вывод и сериализация объектов

Пространство имен `System.IO` позволяет взаимодействовать с существующей структурой файлов и каталогов. В этой главе будет показано, как программно создавать (и удалять) систему каталогов. Вы также узнаете, каким образом перемещать данные в и из разнообразных потоков (файловых, строковых и находящихся в памяти). Кроме того, в главе рассматриваются службы сериализации объектов платформы .NET. Сериализация позволяет сохранить состояние объекта (или набора связанных объектов) в потоке для последующего использования. Десериализация — это процесс извлечения объекта из потока в память для потребления в приложении. После описания основ вы узнаете, как настраивать процесс сериализации с применением интерфейса `ISerializable` и набора атрибутов .NET.

Глава 21. ADO.NET, часть I: подключенный уровень

В этой первой из трех посвященных базам данных главам представлено введение в API-интерфейс доступа к базам данных платформы .NET, который называется ADO.NET. В частности, рассматривается роль поставщиков данных .NET и взаимодействие с реляционной базой данных с использованием *подключенного уровня* ADO.NET, который представлен объектами подключения, объектами команд, объектами транзакций и объектами чтения данных. В этой главе также приведен пример создания специальной базы данных и первой версии специальной библиотеки доступа к данным (AutoLotDAL.dll), неоднократно применяемой в остальных главах книги.

Глава 22. ADO.NET, часть II: автономный уровень

В этой главе изучение способов работы с базами данных продолжается рассмотрением *автономного уровня* ADO.NET. Здесь вы узнаете о роли типа DataSet и объектов адаптеров данных. Кроме того, вы ознакомитесь с многочисленными инструментами Visual Studio, которые могут существенно упростить создание приложений, управляемых данными. Будет показано, как привязывать объекты DataTable к элементам пользовательского интерфейса, и как применять запросы LINQ к находящимся в памяти объектам DataSet, используя API-интерфейс LINQ to DataSet.

Глава 23. ADO.NET, часть III: Entity Framework

В этой главе изучение ADO.NET завершается рассмотрением роли инфраструктуры Entity Framework (EF), которая позволяет создавать код доступа к данным с использованием строго типизированных классов, напрямую отображающихся на бизнес-модель. Здесь будут описаны роли службы объектов EF, клиента сущностей и контекста объектов, а также показано устройство файла *.edmx. Кроме того, вы узнаете, как взаимодействовать с реляционными базами данных с применением LINQ to Entities. В главе также создается финальная версия специальной библиотеки доступа к данным (AutoLotDAL.dll), которая будет использоваться в нескольких оставшихся главах книги.

Глава 24. Введение в LINQ to XML

В главе 14 была представлена модель программирования LINQ — в частности, LINQ to Objects. В этой главе вы углубите свои знания языка LINQ, научившись применять запросы LINQ к XML-документам. Первым делом, будут описаны сложности с манипулированием XML-данными, которые существовали в .NET изначально, когда использовались типы из сборки System.Xml.dll. Затем будет показано, как создавать XML-документы в памяти, сохранять их на жестком диске и перемещаться по их содержимому посредством модели программирования LINQ (LINQ to XML).

Глава 25. Введение в Windows Communication Foundation

До этого места в книге все примеры приложений запускались на единственном компьютере. В этой главе вы узнаете об API-интерфейсе *Windows Communication Foundation* (WCF), который позволяет создавать распределенные приложения в симметричной манере, независимо от лежащих в их основе низкоуровневых деталей. Будет показано, как конструировать службы, хосты и клиенты WCF. Как вы увидите, службы WCF являются чрезвычайно гибкими, поскольку предоставляют клиентам и хостам возможность использования конфигурационных файлов на основе XML, в которых декларативно указываются адреса, привязки и контракты.

Глава 26. Введение в Windows Workflow Foundation

В этой главе вы узнаете о роли приложений, поддерживающих рабочие потоки, и ознакомитесь со способами моделирования бизнес-процессов с применением API-интерфейса WF, введенного в версии .NET 4.0. Кроме того, будет описана библиотека действий WF и показано, как создавать специальные действия, которые используют специальную библиотеку доступа к данным, созданную ранее в книге.

Часть VII. Windows Presentation Foundation

В .NET 3.0 программистам был предложен замечательный API-интерфейс под названием *Windows Presentation Foundation* (WPF). Он быстро стал заменой модели программирования настольных приложений Windows Forms. В сущности, WPF позволяет строить настольные приложения с векторной графикой, интерактивной анимацией и операциями привязки данных, используя декларативную грамматику разметки XAML. Более того, архитектура элементов управления WPF позволяет легко изменять внешний вид и поведение любого элемента управления с помощью правильно оформленной разметки XAML.

Глава 27. Введение в Windows Presentation Foundation и XAML

Инфраструктура WPF позволяет создавать исключительно интерактивные и многофункциональные пользовательские интерфейсы для настольных приложений (и косвенно для веб-приложений). В отличие от Windows Forms, в WPF множество ключевых служб (вроде двумерной и трехмерной графики, анимации, форматированных документов и т.п.) интегрируются в единую унифицированную объектную модель. В этой главе предлагается введение в WPF и язык XAML (Extendable Application Markup Language — расширяемый язык разметки приложений). Вы узнаете, как создавать WPF-приложения вообще без XAML, с использованием только XAML и с применением комбинации обоих подходов. В завершение главы рассматривается пример построения специального редактора XAML, который будет использоваться в остальных главах, посвященных WPF.

Глава 28. Программирование с использованием элементов управления WPF

В этой главе будет показано, как работать с предлагаемыми WPF элементами управления и диспетчерами компоновки. Вы узнаете, как создавать системы меню, окна с разделителями, панели инструментов и строки состояния. Также в главе рассматриваются API-интерфейсы (и связанные с ними элементы управления), входящие в состав WPF — Documents API, Ink API и модель привязки данных.

Глава 29. Службы визуализации графики WPF

В API-интерфейсе WPF интенсивно используется графика, и с учетом этого WPF предоставляет три подхода к визуализации графических данных: *фигуры*, *рисунки* и *геометрии* и *визуальные объекты*. В этой главе вы ознакомитесь с каждым подходом и изучите несколько важных графических примитивов (таких как кисти, перья и трансформации). Также вы узнаете, как выполнять операции проверки попадания в отношении графических данных.

Глава 30. Ресурсы, анимация и стили WPF

В этой главе освещены три важных (и связанных между собой) темы, которые позволят углубить знания API-интерфейса Windows Presentation Foundation. В первую очередь вы узнаете о роли *логических ресурсов*. Система логических ресурсов (также называемых *объектными ресурсами*) предлагает способ именования и ссылки на часто исполь-

зуемые объекты внутри WPF-приложения. Затем вы научитесь определять, выполнять и управлять анимационной последовательностью. Вы увидите, что применение анимации WPF не ограничивается видеоиграми или мультимедиа-приложениями. И, наконец, вы ознакомитесь с ролью *стилей* WPF. Подобно веб-странице, использующей CSS или механизм тем ASP.NET, приложение WPF может определять общий вид и поведение для набора элементов управления.

Глава 31. Свойства зависимости, маршрутизируемые события и шаблоны

Эта глава начинается с рассмотрения двух важных тем, связанных с созданием специальных элементов управления: *свойства зависимости* и *маршрутизируемые события*. Затем описывается роль *стандартного шаблона* и способы его программного просмотра во время выполнения. В завершение главы объясняется, как строить специальные шаблоны.

Часть VIII. ASP.NET Web Forms

Эта часть посвящена построению веб-приложений с использованием API-интерфейса ASP.NET. Технология ASP.NET предназначена для моделирования процесса создания настольных пользовательских интерфейсов путем наложения управляемых событиями, объектно-ориентированной инфраструктуры поверх стандартного запроса/ответа HTTP.

Глава 32. Введение в ASP.NET Web Forms

В этой главе начинается изучение процесса разработки веб-приложений с помощью ASP.NET. Вы увидите, что код сценариев серверной стороны теперь заменен кодом на настоящих объектно-ориентированных языках программирования (например, C# и VB.NET). Здесь рассматривается конструирование веб-страницы ASP.NET, лежащая в основе модель программирования и другие ключевые аспекты ASP.NET, такие как выбор веб-сервера и работа с файлами `web.config`.

Глава 33. Веб-элементы управления, мастер-страницы и темы ASP.NET

В то время как предыдущая глава была посвящена созданию объектов `Page` из ASP.NET, в этой главе рассказывается об элементах управления, которые наполняют внутреннее дерево элементов управления. Здесь описаны основные *веб-элементы управления*, включая элементы управления проверкой достоверности, элементы управления навигацией по сайту и различные операции привязки данных. Кроме того, рассматривается роль *мастер-страниц* и *механизма тем* ASP.NET, который является альтернативой серверной стороны традиционным таблицам стилей.

Глава 34. Управление состоянием в ASP.NET

Эта глава дополняет ваши знания ASP.NET описанием разнообразных способов управления состоянием в .NET. Как и в классическом ASP, в ASP.NET можно создавать cookie-наборы, а также переменные уровня приложения и уровня сеанса. Кроме того, в ASP.NET имеется еще один прием управления состоянием: *кеш приложения*. После исследования технологий управления состоянием с помощью ASP.NET рассматривается роль базового класса `HttpApplication` и демонстрируется динамическое переключение поведения веб-приложения с помощью файла `web.config`.

Загружаемые приложения

На сайте издательства для загрузки доступны два дополнительных приложения. Первое из них посвящено основам API-интерфейса Windows Forms, который используется в нескольких примерах построения пользовательского интерфейса, рассмотренных

в книге. Во втором приложении рассматривается независимая от платформы природа .NET на примере платформы Mono.

Загружаемое приложение А. Программирование с помощью Windows Forms

Первоначальный набор инструментов для построения настольных графических пользовательских интерфейсов, который поставлялся в рамках платформы .NET, называется *Windows Forms*. В этом приложении описана роль этой инфраструктуры пользовательских приложений и показано, как с ее помощью создавать главные окна, диалоговые окна и системы меню. Кроме того, здесь рассматриваются вопросы наследования форм и визуализации двухмерной графики с помощью пространства имен *System.Drawing*. В конце приложения приводится пример создания программы для рисования (средней сложности), иллюстрирующий практическое применение всех описанных концепций.

Загружаемое приложение Б. Независимая от платформы разработка .NET-приложений с помощью Mono

Это приложение посвящено использованию распространяемой с открытым исходным кодом реализации платформы .NET под названием *Mono*. Платформу *Mono* можно применять для построения многофункциональных приложений .NET, которые допускается создавать, развертывать и выполнять под управлением разнообразных операционных систем, включая Mac OS X, Solaris и многочисленные дистрибутивы Linux. Учитывая, что *Mono* по большому счету сравнимо с платформой .NET от Microsoft, вы уже знаете большинство из того, что она предлагает. Поэтому в данном приложении основное внимание уделяется процессу установки *Mono*, инструментам разработки *Mono*, а также механизму исполняющей среды *Mono*.

Исходный код примеров

Исходный код всех рассматриваемых в настоящей книге примеров доступен для загрузки на веб-сайте издательства.

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо, либо просто посетить наш веб-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг.

Наши координаты:

E-mail: info@williamspublishing.com

WWW: <http://www.williamspublishing.com>

Информация для писем из:

России: 127055, г. Москва, ул. Лесная, д. 43, стр. 1

Украины: 03150, Киев, а/я 152

часть I

Введение в C# и платформу .NET

В этой части

Глава 1. Философия .NET

Глава 2. Создание приложений на языке C#

Философия .NET

Платформа Microsoft .NET (и связанный с ней язык программирования C#) впервые была представлена примерно в 2002 г. и быстро стала одной из основных современных сред разработки программного обеспечения. Как было отмечено во вступительном разделе этой книги, при ее написании преследовались две цели. Первая из них — предоставление читателям глубокого и подробного описания синтаксиса и семантики языка C#. Вторая (столь же важная) цель — иллюстрация применения многочисленных API-интерфейсов .NET, включая доступ к базам данных с помощью ADO.NET и Entity Framework (EF), набор технологий LINQ, WPF, WCF, WF и разработку веб-сайтов с использованием ASP.NET. Как говорят, путешествие начинается с первого шага, который и будет сделан в главе 1.

Задача настоящей главы заключается в построении концептуальной основы для успешного освоения всего остального материала книги. Здесь рассматриваются такие связанные с .NET темы, как сборки, общий промежуточный язык (Common Intermediate Language — CIL) и оперативная компиляция (Just-In-Time Compilation — JIT). В дополнение к предварительному обзору ключевых слов языка программирования C#, вы узнаете о взаимоотношениях между различными компонентами платформы .NET, такими как общезыковая исполняющая среда (Common Language Runtime — CLR), общая система типов (Common Type System — CTS) и общезыковая спецификация (Common Language Specification — CLS).

Кроме того, в настоящей главе представлен обзор функциональности, предоставляемой в библиотеках базовых классов .NET 4.5, для обозначения которых иногда используют аббревиатуру BCL (base class library — библиотека базовых классов). Здесь вы кратко ознакомитесь с независимой от языка и платформы природой платформы .NET (это действительно так — область применения .NET не ограничивается операционной системой Windows) и узнаете о роли .NET при создании приложений, ориентированных на среду Windows 8. Как должно быть понятно, многие из этих тем будут более подробно освещаться в остальной части книги.

Начальное знакомство с платформой .NET

До того, как компания Microsoft выпустила язык C# и платформу .NET, разработчики программного обеспечения, создававшие приложения для операционных систем семейства Windows, часто применяли модель программирования COM. Технология COM (Component Object Model — модель компонентных объектов) позволяла строить библиотеки, которые можно было использовать в различных языках программирования. Например, программист на C++ мог построить библиотеку COM, которой мог пользоваться разработчик на Visual Basic. Независимая от языка природа COM, безусловно, была удобна. Однако недостатком модели COM являлась усложненная инфраструктура, хрупкая модель развертывания и возможность работы только под управлением Windows.

Несмотря на сложность и ограничения COM, с применением этой архитектуры было успешно создано буквально бесчисленное количество приложений. Тем не менее, в настоящее время большинство приложений, ориентированных на семейство операционных систем Windows, создаются без применения модели COM. Вместо этого приложения для настольных компьютеров, веб-сайты, службы операционной системы и библиотеки многократно используемой логики доступа к данным и бизнес-логики строятся с помощью платформы .NET.

Некоторые основные преимущества платформы .NET

Как уже было сказано, язык C# и платформа .NET впервые были представлены в 2002 г., и целью их создания было обеспечение более мощной, гибкой и простой модели программирования по сравнению с COM. Как можно будет увидеть в остальных материалах книги, .NET Framework — это программная платформа для построения приложений на базе семейства операционных систем Windows, а также многочисленных операционных систем производства не Microsoft, таких как Mac OS X и различные дистрибутивы Unix и Linux. Для начала ниже приведен краткий перечень некоторых ключевых средств, поддерживаемых .NET.

- *Возможность взаимодействия с существующим кодом.* Эта возможность, несомненно, является очень полезной, поскольку позволяет комбинировать существующие двоичные компоненты COM (т.е. обеспечивать взаимодействие с ними) с более новыми программными компонентами .NET и наоборот. С выходом .NET 4.0 и последующих версий возможность взаимодействия дополнительно упростилась благодаря добавлению ключевого слова *dynamic* (которое рассматривается в главе 16).
- *Поддержка многочисленных языков программирования.* Приложения .NET можно создавать с использованием любого числа языков программирования (C#, Visual Basic, F# и т.д.).
- *Общий исполняющий механизм, разделяемый всеми поддерживаемыми .NET языками.* Одним из аспектов этого механизма является наличие хорошо определенного набора типов, которые способен понимать каждый поддерживающий .NET язык.
- *Языковая интеграция.* В .NET поддерживается межъязыковое наследование, межъязыковая обработка исключений и межъязыковая отладка кода. Например, базовый класс может быть определен в C#, а затем расширен в Visual Basic.
- *Обширная библиотека базовых классов.* Эта библиотека позволяет избегать сложностей, связанных с выполнением низкоуровневых обращений к API-интерфейсам, и предлагает согласованную объектную модель, используемую всеми поддерживаемыми .NET языками.
- *Упрощенная модель развертывания.* В отличие от COM, библиотеки .NET не регистрируются в системном реестре. Более того, платформа .NET позволяет сосуществовать на одном и том же компьютере несколькими версиями одной и той же сборки *.dll.

Эти и многие другие темы будут подробно рассматриваться в последующих главах.

Введение в строительные блоки платформы .NET (CLR, CTS и CLS)

Теперь, когда вы узнали об основных преимуществах, обеспечиваемых платформой .NET, рассмотрим три ключевых (взаимосвязанных) компонента, которые делают это возможным — CLR, CTS и CLS. С точки зрения программиста .NET представляет собой исполняющую среду и обширную библиотеку базовых классов. Уровень исполняющей среды называется *общезыковой исполняющей средой* (Common Language Runtime) или, сокращенно, средой CLR. Главной задачей CLR является автоматическое обнаружение, загрузка и управление объектами .NET (вместо программиста). Кроме того, среда CLR заботится о ряде низкоуровневых деталей, таких как управление памятью, размещение приложения, координирование потоков и выполнение проверок, связанных с безопасностью (в числе прочих низкоуровневых нюансов).

Другим строительным блоком платформы .NET является *общая система типов* (Common Type System) или, сокращенно, система CTS. В спецификации CTS полностью описаны все возможные типы данных и все программные конструкции, поддерживаемые исполняющей средой. Кроме того, в CTS показано, как эти сущности могут взаимодействовать друг с другом, и указано, как они представлены в формате метаданных .NET (которые рассматриваются далее в этой главе, а более подробно — в главе 15).

Важно понимать, что отдельно взятый язык, совместимый с .NET, может не поддерживать абсолютно все функциональные средства, определенные спецификацией CTS. Поэтому существует *общезыковая спецификация* (Common Language Specification) или, сокращенно, спецификация CLS, в которой описано подмножество общих типов и программных конструкций, которое должны поддерживать все языки программирования для .NET. Таким образом, если создаваемые типы .NET предлагают только средства, совместимые с CLS, ими могут пользоваться все языки, поддерживающие .NET. И, наоборот, в случае применения типа данных или программной конструкции, выходящей за рамки CLS, нельзя гарантировать возможность взаимодействия с такой библиотекой кода .NET каждым языком программирования .NET. К счастью, как будет показано далее в этой главе, существует очень простой способ указать компилятору C# на необходимость проверки всего кода относительно совместимости с CLS.

Роль библиотек базовых классов

В дополнение к среде CLR и спецификациям CTS/CLS, платформа .NET предоставляет библиотеку базовых классов, которая доступна всем языкам программирования .NET. Эта библиотека не только инкапсулирует разнообразные примитивы, такие как потоки, файловый ввод-вывод, системы визуализации графики и механизмы взаимодействия с различными внешними устройствами, но также обеспечивает поддержку для многочисленных служб, требуемых большинством реальных приложений.

Библиотеки базовых классов определяют типы, которые можно использовать для построения программных приложений любого вида. Например, ASP.NET можно применять для построения веб-сайтов, WCF — для создания сетевых служб, WPF — для написания настольных приложений с графическим пользовательским интерфейсом и т.д. Кроме того, библиотеки базовых классов предоставляют типы для взаимодействия с XML-документами, локальным каталогом и файловой системой текущего компьютера, для коммуникаций с реляционными базами данных (через ADO.NET) и т.п. На высоком уровне отношения между CLR, CTS, CLS и библиотеками базовых классов выглядят так, как показано на рис. 1.1.

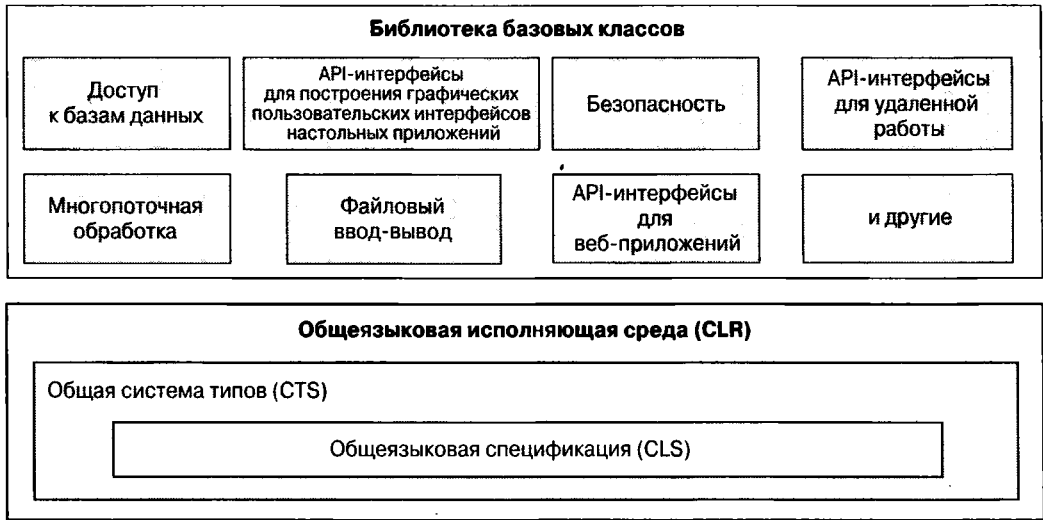


Рис. 1.1. Отношения между CLR, CTS, CLS и библиотеками базовых классов

Что привносит язык C#

Синтаксис языка программирования C# выглядит очень похожим на синтаксис языка Java. Однако называть C# клоном Java неправильно. В действительности и C#, и Java являются членами семейства языков программирования, основанного на C (куда также входят C, Objective C, C++), и потому они разделяют схожий синтаксис.

Правда заключается в том, что многие синтаксические конструкции C# смоделированы на основе разнообразных аспектов языков Visual Basic (VB) и C++. Например, подобно VB, язык C# поддерживает идею свойств класса (как противоположность традиционным методам извлечения и установки) и необязательные параметры. Подобно C++, язык C# позволяет перегружать операции, а также создавать структуры, перечисления и функции обратного вызова (посредством делегатов).

Более того, по мере изучения материала этой книги, вы очень скоро увидите, что C# поддерживает множество средств, которые традиционно встречаются в различных языках функционального программирования (например, LISP или Haskell), скажем, лямбда-выражения и анонимные типы. Вдобавок, с появлением технологии LINQ (*Language Integrated Query* — язык интегрированных запросов), язык C# стал поддерживать конструкции, которые делают его довольно уникальным в мире программирования. Несмотря на все это, наибольшее влияние на него оказали именно языки, основанные на C.

Вследствие того, что C# представляет собой гибрид из нескольких языков, он является таким же синтаксически чистым — если не чище — как и Java, почти столь же простым, как VB, и практически таким же мощным и гибким, как C++. Ниже приведен далеко не полный список ключевых особенностей языка C#, которые характерны для всех его версий.

- Указатели использовать не требуется! В программах на C# обычно не возникает нужды в манипулировании указателями напрямую (хотя в случае абсолютной необходимости можно опуститься на этот уровень, как будет показано в главе 11).
- Автоматическое управление памятью посредством сборки мусора. Учитывая это, ключевое слово `delete` в C# не поддерживается.

- Формальные синтаксические конструкции для классов, интерфейсов, структур, перечислений и делегатов.
- Аналогичная языку C++ возможность перегрузки операций для специальных типов без излишних сложностей (например, обеспечение “возврата **this*, чтобы позволить связывание в цепочку” — не ваша забота).
- Поддержка программирования на основе атрибутов. Эта разновидность разработки позволяет аннотировать типы и их членов с целью дополнительного уточнения их поведения. Например, если пометить метод атрибутом [Obsolete], при попытке использования этого члена программисты получают соответствующее специальное предупреждение.

С выходом версии .NET 2.0 (примерно в 2005 г.), язык программирования C# был обновлен для поддержки многочисленных новых функциональных возможностей, наиболее значимые из которых перечислены ниже.

- Возможность создания обобщенных типов и обобщенных членов. Используя обобщения, можно создавать очень эффективный и безопасный к типам код, который определяет множество *заполнителей*, указываемых во время взаимодействия с обобщенными элементами.
- Поддержка анонимных методов, которые позволяют предоставлять встраиваемую функцию везде, где требуется тип делегата.
- Возможность определения единственного типа в нескольких файлах кода (или, если необходимо, в виде представления в памяти) с использованием ключевого слова *partial*.

В версии .NET 3.5 (вышедшей примерно в 2008 г.) в язык программирования C# была добавлена дополнительная функциональность, включая следующие средства.

- Поддержка строго типизированных запросов (т.е. LINQ), применяемых для взаимодействия с разнообразными формами данных. Запросы LINQ вы впервые встретите в главе 12.
- Поддержка анонимных типов, которые позволяют моделировать *форму* типа, а не его поведение.
- Возможность расширения функциональности существующего типа (не создавая его подклассы) с использованием расширяющих методов.
- Включение лямбда-операции (*=>*), которая еще больше упрощает работу с типами делегатов .NET.
- Новый синтаксис инициализации объектов, который позволяет устанавливать значения свойств во время создания объекта.

Версия .NET 4.0 (вышедшая в 2010 г.) дополнила C# еще рядом средств, в том числе указанными ниже.

- Поддержка необязательных параметров, а также именованных аргументов в методах.
- Поддержка динамического поиска членов во время выполнения через ключевое слово *dynamic*. Как будет показано в главе 18, это обеспечивает универсальный подход к вызову членов на лету независимо от инфраструктуры, с помощью которой они были реализованы (COM, IronRuby, IronPython или службы рефлексии .NET).
- Работа с обобщенными типами стала намного понятнее, учитывая возможность легкого отображения обобщенных данных на универсальные коллекции *System.Object* через ковариантность и контравариантность.

Все это подводит нас к текущей версии C#, входящей в состав платформы .NET 4.5. В этой версии C# появилась пара новых ключевых слов (`async` и `await`), которые значительно упрощают многопоточное и асинхронное программирование. Те, кому приходилось работать с предшествующими версиями C#, могут припомнить, что вызов методов через вторичные потоки требовал довольно большого объема малопонятного кода и использования различных пространств имен .NET. Благодаря тому, что теперь C# поддерживает языковые ключевые слова, которые автоматически решают эти задачи, процесс вызова методов асинхронным образом почти столь же прост, как их вызов в синхронной манере. Более подробно эти темы рассматриваются в главе 19.

Сравнение управляемого и неуправляемого кода

Возможно, наиболее важный аспект, который следует знать о языке C#, заключается в том, что он порождает код, который может выполняться только в рамках исполняющей среды .NET (использовать C# для построения COM-сервера или неуправляемого приложения C/C++ не допускается). Выражаясь официально, для обозначения кода, ориентированного на исполняющую среду .NET, применяется термин **управляемый код**. Двоичный модуль, который содержит управляемый код, называется **сборкой** (сборки более подробно рассматриваются далее в этой главе). В противоположность этому, код, который не может обслуживаться непосредственно исполняющей средой .NET, называется **неуправляемым кодом**.

Другие языки программирования, ориентированные на .NET

Важно помнить, что C# — это не единственный язык, который может использоваться для построения .NET-приложений. Среда Visual Studio изначально предлагает пять управляемых языков, а именно — C#, Visual Basic, C++/CLI, JavaScript и F#.

На заметку! F# — это язык .NET, основанный на синтаксисе функциональных языков. Хотя он может применяться как чистый функциональный язык, в нем также предлагается поддержка конструкций объектно-ориентированного программирования (ООП) и библиотек базовых классов .NET. Дополнительные сведения об этом управляемом языке доступны на его официальной домашней странице по адресу <http://msdn.microsoft.com/fsharp>.

В дополнение к управляемым языкам, предлагаемым Microsoft, существуют .NET-компиляторы, которые предназначены для языков Smalltalk, Ruby, Python, COBOL и Pascal (и это далеко не полный перечень). Хотя в настоящей книге внимание сконцентрировано на языке C#, вы можете счесть небезынтересным следующий веб-сайт:

www.dotnetlanguages.net

Щелкнув на ссылке **Resources** (Ресурсы) в самом верху домашней страницы этого сайта, можно получить список всех языков программирования .NET и соответствующие ссылки, предназначенные для загрузки различные компиляторов (рис. 1.2).

Даже если вы интересуетесь в первую очередь построением программ .NET с использованием синтаксиса C#, все равно рекомендуется посетить указанный сайт, поскольку вы наверняка сочтете многие языки для .NET заслуживающими дополнительного внимания (к примеру, LISP.NET).

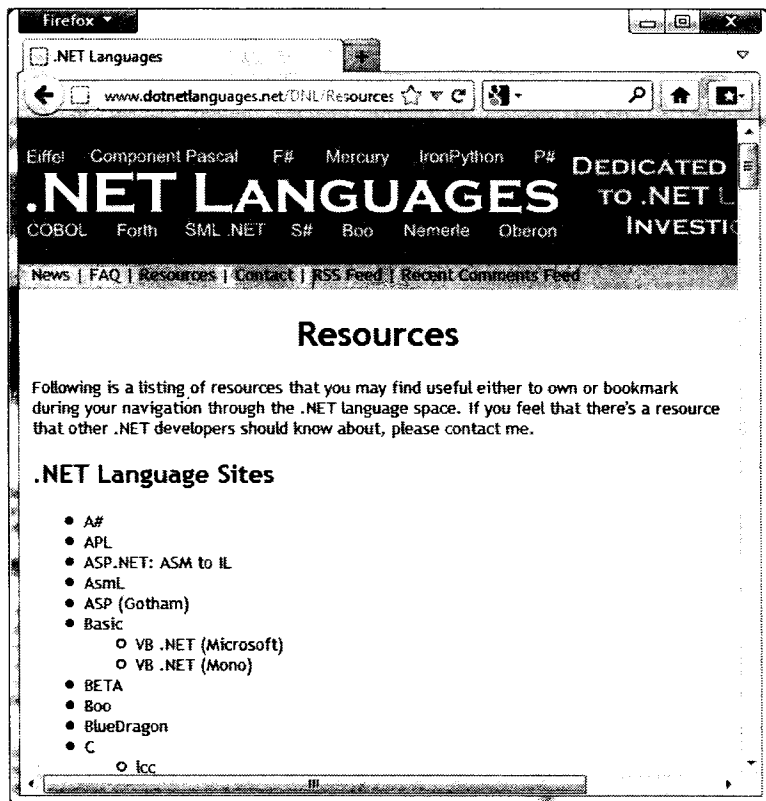


Рис. 1.2. Один из многочисленных сайтов, документирующих известные языки программирования .NET

Жизнь в многоязычном окружении

Как только разработчики приходят к осознанию независимой от языка природы .NET, у них возникает множество вопросов. Самый распространенный из них может быть сформулирован так: если все языки .NET компилируются в управляемый код, почему существует не один, а множество языков/компиляторов?

Ответить на этот вопрос можно по-разному. Программисты бывают очень привередливы, когда дело касается выбора языка программирования. Некоторые предпочитают языки с многочисленными точками с запятой и фигурными скобками, но с минимальным набором ключевых слов. Другим нравятся языки, предлагающие более читабельные синтаксические конструкции (такие как в Visual Basic). Кто-то не желает отказываться от своего опыта работы на мэйнфреймах и предпочитает перенести его на платформу .NET (выбирая компилятор COBOL.NET).

А теперь ответьте честно: если бы в Microsoft предложили единственный “официальный” язык .NET, например, на базе семейства BASIC, то все ли программисты были бы рады такому выбору? Или если бы “официальный” язык .NET основывался на синтаксисе Fortran, сколько людей в мире просто бы проигнорировали платформу .NET? Поскольку исполняющая среда .NET демонстрирует меньшую зависимость от языка, используемого для построения блока управляемого кода, программисты .NET могут, сохраняя свои синтаксические предпочтения, обмениваться скомпилированными сборками с коллегами, другими отделами и внешними организациями (и не обращать внимания на то, какой язык .NET в них применяется).

Еще одно полезное преимущество интеграции различных языков .NET в одном унифицированном программном решении вытекает из того простого факта, что каждый язык программирования имеет свои сильные и слабые стороны. Например, некоторые языки программирования обладают превосходной встроенной поддержкой для выполнения сложных математических вычислений. В других лучше реализованы финансовые или логические вычисления, взаимодействие с мейнфреймами и т.п. А когда преимущества конкретного языка программирования объединяются с преимуществами платформ .NET, выигрывают все.

Конечно, в реальности велика вероятность того, что придется тратить большую часть времени на построение программного обеспечения с помощью предпочитаемого языка .NET. Тем не менее, после освоения синтаксиса одного из языков .NET изучение синтаксиса какого-то другого языка существенно упрощается. Вдобавок это довольно выгодно, особенно тем, кто занимается консультированием по разработке программного обеспечения. Например, если предпочитаемым языком у вас является C#, то при попадании в клиентскую среду, где все построено на Visual Basic, вы все равно сможете пользоваться функциональностью .NET Framework и понимать общую структуру кодовой базы с минимальными усилиями и беспокойством.

Обзор сборок .NET

Какой бы язык .NET не был выбран для программирования, важно понимать, что хотя двоичные модули .NET имеют то же самое файловое расширение, как и неуправляемые двоичные компоненты Windows (*.dll или *.exe), внутренне они устроены совершенно по-другому. В частности, двоичные модули .NET содержат не специфические, а независимые от платформы инструкции на промежуточном языке (Intermediate Language — IL) и метаданные типов. На рис. 1.3 показано, как все это выглядит схематически.

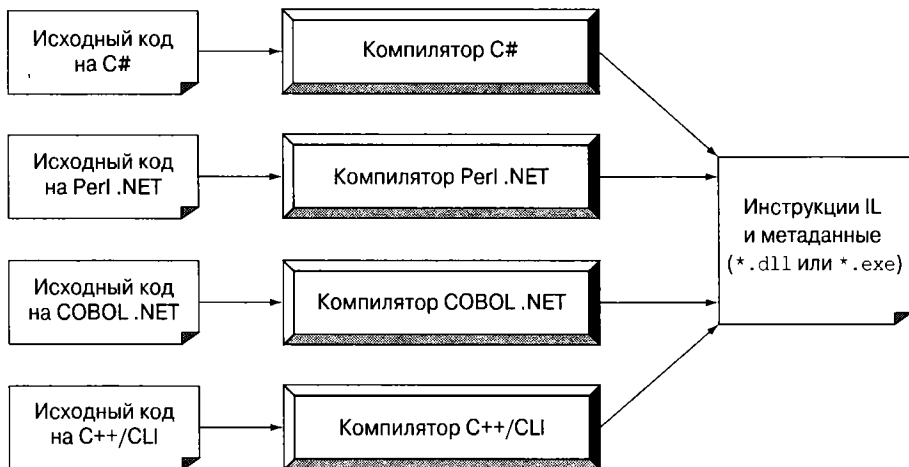


Рис. 1.3. Все .NET-компиляторы генерируют инструкции IL и метаданные

На заметку! Относительно аббревиатуры IL уместно сказать несколько дополнительных слов. IL также известен как промежуточный язык Microsoft (Microsoft Intermediate Language — MSIL) или же общий промежуточный язык (Common Intermediate Language — CIL). Поэтому при чтении литературы по .NET не забывайте о том, что IL, MSIL и CIL обозначают одну и ту же концепцию. В данной книге при ссылке на низкоуровневый набор инструкций будет использоваться аббревиатура CIL.

Когда файл *.dll или *.exe был создан с помощью .NET-компилятора, полученный большой двоичный объект называется *сборкой*. Все многочисленные детали, касающиеся сборок .NET, подробно рассматриваются в главе 14. Однако для упрощения текущей дискуссии необходимо понимать некоторые основные свойства этого нового файлового формата.

Как уже упоминалось, сборка содержит код CIL, который концептуально похож на байт-код Java тем, что не компилируется в специфичные для платформы инструкции до тех пор, пока это не станет абсолютно необходимым. Обычно “абсолютная необходимость” наступает тогда, когда на блок инструкций CIL (такой как реализация метода) производится ссылка для его использования исполняющей средой .NET.

Кроме инструкций CIL сборки также содержат *метаданные*, которые детально описывают характеристики каждого “типа” внутри двоичного модуля. Например, если имеется класс по имени SportsCar, то метаданные типа описывают такие детали, как базовый класс SportsCar, реализуемые SportsCar интерфейсы (если есть), а также полные описания всех членов, поддерживаемых SportsCar. Метаданные .NET всегда представляются внутри сборки и автоматически генерируются компилятором соответствующего языка .NET.

Наконец, помимо инструкций CIL и метаданных типов, сами сборки также описываются с помощью метаданных, которые официально называются *манифестом*. Манифест содержит информацию о текущей версии сборки, сведения о культуре (применяемые для локализации строковых и графических ресурсов) и список ссылок на все внешние сборки, которые требуются для правильного функционирования. Разнообразные инструменты, которые можно использовать для исследования типов, метаданных и манифестов сборок, рассматриваются в нескольких последующих главах.

Роль языка CIL

Теперь давайте немного более подробно посмотрим, что собой представляет код CIL, метаданные типов и манифест сборки. CIL — это язык, который находится выше любого конкретного набора специфичных для платформы инструкций. Например, приведенный ниже код C# моделирует простой калькулятор. Не углубляясь пока в детали синтаксиса, обратите внимание на формат метода Add() в классе Calc.

```
// Класс Calc.cs
using System;
namespace CalculatorExample
{
    // Этот класс содержит точку входа приложения.
    class Program
    {
        static void Main()
        {
            Calc c = new Calc();
            int ans = c.Add(10, 84);
            Console.WriteLine("10 + 84 is {0}.", ans);
            // Ожидать нажатия пользователем клавиши <Enter> перед выходом.
            Console.ReadLine();
        }
    }
    // Калькулятор на C#.
    class Calc
    {
        public int Add(int x, int y)
        { return x + y; }
    }
}
```

После компиляции этого файла кода с помощью компилятора C# (csc.exe) получается однофайловая сборка *.exe, содержащая манифест, инструкции CIL и метаданные, описывающие каждый аспект классов Calc и Program.

На заметку! В главе 2 вы найдете подробные сведения о компиляции кода с применением компилятора C#, а также об использовании графических IDE-сред, таких как Microsoft Visual Studio.

Например, открыв полученную сборку в утилите ildasm.exe (которая рассматривается далее в этой главе), можно увидеть, что метод Add() был преобразован в CIL следующим образом:

```
.method public hidebysig instance int32 Add(int32 x,
      int32 y) cil managed
{
    // Code size 9 (0x9)
    .maxstack 2
    .locals init (int32 V_0)
    IL_0000: nop
    IL_0001: ldarg.1
    IL_0002: ldarg.2
    IL_0003: add
    IL_0004: stloc.0
    IL_0005: br.s IL_0007
    IL_0007: ldloc.0
    IL_0008: ret
} // end of method Calc::Add
```

Не стоит беспокоиться, если результирующий CIL-код этого метода выглядит непонятным, потому что в главе 17 будут рассматриваться все необходимые базовые аспекты языка программирования CIL. Главное помнить, что компилятор C# выдает CIL-код, а не инструкции, ориентированные на определенную платформу.

Как упоминалось ранее, подобным образом ведут себя все компиляторы .NET. Чтобы удостовериться в этом, создадим то же самое приложение на языке Visual Basic, а не C#:

```
' Класс Calc.vb
Imports System

Namespace CalculatorExample

    ' "Модуль" VB — это класс, который
    ' содержит только статические члены.
    Module Program
        Sub Main()
            Dim c As New Calc
            Dim ans As Integer = c.Add(10, 84)
            Console.WriteLine("10 + 84 is {0}.", ans)
            Console.ReadLine()
        End Sub
    End Module

    Class Calc
        Public Function Add(ByVal x As Integer, ByVal y As Integer) As Integer
            Return x + y
        End Function
    End Class

End Namespace
```

Просмотрев CIL-код этого метода Add(), можно обнаружить похожие инструкции (лишь слегка скорректированные компилятором Visual Basic, vbc.exe):

```
.method public instance int32 Add(int32 x,
  int32 y) cil managed
{
  // Code size 8 (0x8)
  .maxstack 2
  .locals init (int32 V_0)
  IL_0000: ldarg.1
  IL_0001: ldarg.2
  IL_0002: add.ovf
  IL_0003: stloc.0
  IL_0004: br.s IL_0006
  IL_0006: ldloc.0
  IL_0007: ret
} // end of method Calc::Add
```

Исходный код. Файлы с кодом Calc.cs и Calc.vb доступны в подкаталоге Chapter 01.

Преимущества языка CIL

На этом этапе может возникнуть вопрос о том, какую выгоду приносит компиляция исходного кода в CIL, а не напрямую в набор ориентированных на конкретную платформу инструкций. Одним из самых важных преимуществ такого подхода является языковая интеграция. Как уже можно было увидеть, все компиляторы .NET генерируют примерно одинаковые CIL-инструкции. Благодаря этому все языки могут взаимодействовать в рамках четко обозначенной двоичной “арены”.

Более того, поскольку CIL не зависит от платформы, сама .NET Framework является независимой от платформы, предоставляя те же самые преимущества, к которым привыкли разработчики на Java (например, единую кодовую базу, способную работать в средах разных операционных систем). В действительности существует международный стандарт для языка C# и крупного подмножества платформы .NET, а также реализации для многих операционных систем, отличных от Windows (более подробно об этом речь пойдет в конце этой главы).

Компиляция CIL-кода в инструкции, специфичные для платформы

Из-за того, что сборки содержат CIL-инструкции, а не инструкции, специфичные для платформы, CIL-код перед использованием должен компилироваться на лету. Компонент, который отвечает за компиляцию CIL-кода в понятные процессору инструкции, называется оперативным (just-in-time — JIT) компилятором (иногда его называют Jitter). Исполняющая среда .NET использует JIT-компилятор в соответствии с целевым процессором и оптимизирует его для базовой платформы.

Например, если строится приложение .NET, предназначенное для развертывания на портативном устройстве (например, на мобильном устройстве Windows), соответствующий JIT-компилятор будет оптимизирован для функционирования в среде с ограниченным объемом памяти. С другой стороны, в случае развертывания сборки на серверной системе (где объем памяти редко представляет проблему), JIT-компилятор будет оптимизирован для функционирования в среде с большим объемом памяти. Это дает разработчикам возможность писать единственный блок кода, который будет автоматически эффективно компилироваться JIT-компилятором и выполняться на машинах с разной архитектурой.

Более того, при компиляции CIL-инструкций в соответствующий машинный код JIT-компилятор будет кешировать результаты в памяти в манере, подходящей для целевой операционной системы. В этом случае, если производится вызов метода по имени `PrintDocument()`, соответствующие CIL-инструкции компилируются в специфичные для платформы инструкции при первом вызове метода и остаются в памяти для последующего использования. Благодаря этому, при вызове `PrintDocument()` в следующий раз повторная компиляция CIL-инструкций не понадобится.

На заметку! Можно также выполнять “предварительную JIT-компиляцию” при установке приложения с помощью инструмента командной строки `ngen.exe`, который входит в состав .NET Framework 4.5 SDK. Это позволяет улучшить показатели времени запуска для приложений, насыщенных графикой.

Роль метаданных типов .NET

Кроме CIL-инструкций в сборке .NET содержатся исчерпывающие и точные метаданные, которые описывают каждый определенный в двоичном модуле тип (например, класс, структуру, перечисление), а также члены каждого типа (например, свойства, методы, события). К счастью, за генерацию новейших и наилучших метаданных по типам всегда отвечает компилятор, а не программист. Из-за того, что метаданные .NET являются настолько детальными, сборки являются целиком самоописательными сущностями.

Для иллюстрации формата метаданных типов .NET давайте рассмотрим метаданные, которые были сгенерированы для приведенного выше метода `Add()` из класса `Calc` на языке C# (метаданные для версии метода `Add()` на языке Visual Basic аналогичны):

```
TypeDef #2 (02000003)
```

```
-----
TypeDefName: CalculatorExample.Calc (02000003)
Flags       : [NotPublic] [AutoLayout] [Class]
[AnsiClass] [BeforeFieldInit] (00100001)
Extends     : 01000001 [TypeRef] System.Object
Method #1 (06000003)
```

```
-----
MethodName: Add (06000003)
Flags      : [Public] [HideBySig] [ReuseSlot] (00000086)
RVA       : 0x00002090
ImplFlags : [IL] [Managed] (00000000)
CallConvtn: [DEFAULT]
hasThis
ReturnType: I4
  2 Arguments
    Argument #1: I4
    Argument #2: I4
  2 Parameters
    (1) ParamToken : (08000001) Name : x flags: [none] (00000000)
    (2) ParamToken : (08000002) Name : y flags: [none] (00000000)
```

Метаданные используются многими аспектами самой исполняющей среды .NET, а также различными средствами разработки. Например, средство `IntelliSense`, предоставляемое такими инструментами, как `Visual Studio`, работает за счет чтения метаданных сборки во время проектирования. Метаданные также применяются различными утилитами для просмотра объектов, инструментами отладки и самим компилятором C#.

Можно с полной уверенностью утверждать, что метаданные играют критически важную роль во многих .NET-технологиях, включая Windows Communication Foundation (WCF), рефлексии, позднее связывание и сериализацию объектов. Более подробно роль метаданных .NET раскрывается в главе 15.

Роль манифеста сборки

Наконец, последний, но не менее важный момент — вспомните, что сборка .NET также содержит метаданные, которые описывают ее саму (формально они называются манифестом). Помимо прочих деталей, манифест документирует все внешние сборки, которые требуются текущей сборке для корректного функционирования, версию сборки, информацию об авторских правах и т.д. Подобно метаданным типов, за генерацию манифеста сборки всегда отвечает компилятор. Ниже приведены некоторые существенные детали манифеста, сгенерированного в результате компиляции приведенного ранее в этой главе файла кода Calc.cs (предполагается, что компилятор был инструктирован назначить сборке имя Calc.exe):

```
.assembly extern mscorlib
{
    .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
    .ver 4:0:0:0
}
.assembly Calc
{
    .hash algorithm 0x00008004
    .ver 0:0:0:0
}
.module Calc.exe
.imagebase 0x00400000
.subsystem 0x00000003
.file alignment 0x00000200
.corflags 0x00000001
```

В сущности, этот манифест документирует набор требуемых для Calc.exe внешних сборок (в директиве `.assembly extern`), а также различные характеристики самой сборки (вроде номера версии и имени модуля). Более подробно данные манифеста будут обсуждаться в главе 14.

Понятие общей системы типов (CTS)

Сборка может содержать любое количество различающихся типов. В мире .NET тип — это просто общий термин, используемый для ссылки на члены из набора {класс, интерфейс, структура, перечисление, делегат}. При построении решений на любом языке .NET, скорее всего, придется взаимодействовать со многими из этих типов. Например, сборка может определять класс, реализующий несколько интерфейсов. Возможно, метод одного из интерфейсов принимает в качестве входного параметра перечисление, а возвращает структуру.

Вспомните, что CTS является формальной спецификацией, в которой указано, как типы должны быть определены, чтобы они могли обслуживаться средой CLR. Внутренние детали CTS обычно интересуют только тех, кто занимается разработкой инструментов и/или компиляторов для платформы .NET. Однако все .NET-программисты должны уметь работать с пятью типами, определенными CTS, на выбранном языке. Краткий обзор типов приведен ниже.

Типы классов CTS

Каждый совместимый с .NET язык поддерживает, как минимум, понятие типа класса, которое играет центральную роль в ООП. Класс может включать любое количество членов (таких как конструкторы, свойства, методы и события) и элементов данных (полей). В C# классы объявляются с помощью ключевого слова `class`, примерно так:

```
// Тип класса C# с одним методом.
class Calc
{
    public int Add(int x, int y)
    {
        return x + y;
    }
}
```

Формальное знакомство с построением типов классов в C# начинается в главе 5, а пока в табл. 1.1 приведен перечень характеристик, которые свойственны типам классов.

Таблица 1.1. Характеристики классов CTS

Характеристика классов	Описание
Является ли класс запечатанным?	Запечатанные классы не могут выступать в качестве базовых для других классов
Реализует ли класс какие-то интерфейсы?	Интерфейс — это коллекция абстрактных членов, которая предоставляет контракт между объектом и пользователем объекта. CTS позволяет классу реализовывать любое количество интерфейсов
Является класс абстрактным или конкретным?	Абстрактные классы не допускают прямого создания экземпляров и предназначены для определения общих поведений для производных типов. Экземпляры конкретных классов могут создаваться напрямую
Какова видимость класса?	Каждый класс должен конфигурироваться с ключевым словом видимости, таким как <code>public</code> или <code>internal</code> . По сути, оно управляет тем, может ли класс использоваться внешними сборками или только внутри определяющей его сборки

Типы интерфейсов CTS

Интерфейсы — это всего лишь именованные коллекции определений абстрактных членов, которые могут поддерживаться (т.е. быть реализованными) в заданном классе или структуре. В C# типы интерфейсов определяются с помощью ключевого слова `interface`. По соглашению имена всех интерфейсов .NET начинаются с прописной буквы `I`, как показано в следующем примере:

```
// Тип интерфейса в C# обычно объявляется
// открытым, чтобы позволить типам в других
// сборках реализовать его поведение.
public interface IDraw
{
    void Draw();
}
```


Сами по себе интерфейсы не особенно полезны. Однако когда класс или структура реализует отдельный интерфейс уникальным образом, появляется возможность получать доступ к заданной функциональности, используя ссылку на этот интерфейс в полиморфной манере. Программирование на основе интерфейсов подробно рассматривается в главе 8.

Типы структур CTS

Концепция структуры также формализована в рамках CTS. Тем, кто имел дело с языком C, будет приятно узнать, что определяемые пользователем типы (user-defined type — UDT) сохранились в мире .NET (хотя их поведение несколько изменилось). Выражаясь просто, структуру можно считать облегченным типом класса, имеющим семантику на основе значений. Более подробно структуры рассматриваются в главе 4. Обычно структуры лучше всего подходят для моделирования геометрических и математических данных и создаются в C# с применением ключевого слова `struct`, как показано в следующем примере:

```
// Тип структуры в C#.
struct Point
{
    // Структуры могут содержать поля.
    public int xPos, yPos;

    // Структуры могут содержать параметризованные конструкторы.
    public Point(int x, int y)
    { xPos = x; yPos = y; }

    // Структуры могут определять методы.
    public void PrintPosition()
    {
        Console.WriteLine("{0}, {1}", xPos, yPos);
    }
}
```

Типы перечислений CTS

Перечисления — это удобная программная конструкция, которая позволяет группировать пары “имя-значение”. Например, предположим, что требуется создать игровое приложение, в котором игроку бы позволялось выбирать персонажа одной из трех категорий: Wizard (маг), Fighter (воин) или Thief (вор). Вместо того чтобы использовать и отслеживать числовые значения для каждого варианта, гораздо удобнее создать строго типизированное перечисление с помощью ключевого слова `enum`:

```
// Тип перечисления C#.
enum CharacterType
{
    Wizard = 100,
    Fighter = 200,
    Thief = 300
}
```

По умолчанию для хранения каждого элемента выделяется блок памяти, соответствующий 32-битному целому, однако при необходимости (например, при программировании для устройств с малым объемом памяти, таким как мобильные устройства) это можно изменить. Кроме того, спецификация CTS требует, чтобы перечислимые типы порождались от общего базового класса `System.Enum`. Как будет показано в главе 4, этот базовый класс определяет несколько интересных членов, которые позволяют извлекать,

манипулировать и преобразовывать лежащие в основе пары “имя-значение” программным образом.

Типы делегатов CTS

Делегаты являются .NET-эквивалентом безопасных к типам указателей на функции в стиле C. Главное отличие заключается в том, что делегат .NET представляет собой класс, порожденный от `System.MulticastDelegate`, а не просто указатель на низкоуровневый адрес в памяти. В C# делегаты объявляются с помощью ключевого слова `delegate`:

```
// Этот тип делегата в C# может "указывать" на любой метод,  
// возвращающий значение int и принимающий два значения int.  
delegate int BinaryOp(int x, int y);
```

Делегаты критически важны, когда требуется обеспечить объект возможностью перенаправления вызова другому объекту, и они формируют основу архитектуры событий .NET. Как будет показано в главах 11 и 19, делегаты обладают внутренней поддержкой группового вызова (т.е. перенаправления запроса множеству получателей) и асинхронного вызова методов (т.е. вызова методов во вторичном потоке).

Члены типов CTS

Теперь, когда было приведено краткое описание каждого из формализованных в CTS типов, пришла пора рассказать о том, что большинство из этих типов способно поддерживать любое количество членов. Формально член типа ограничен набором {конструктор, финализатор, статический конструктор, вложенный тип, операция, метод, свойство, индексатор, поле, поле только для чтения, константа, событие}.

В спецификации CTS описываются различные *характеристики*, которые могут быть ассоциированы с членом. Например, каждый член может иметь характеристику доступности (т.е. открытый, закрытый или защищенный). Некоторые члены могут быть объявлены как абстрактные (для обеспечения полиморфного поведения производными типами) или как виртуальные (для определения фиксированной, но допускающей переопределение реализации). Кроме того, большинство членов могут быть сконфигурированы как статические (связанные с уровнем класса) или члены экземпляра (связанные с уровнем объекта). Создание членов типов будет описано в нескольких следующих главах.

На заметку! Как будет показано в главе 9, язык C# также поддерживает создание обобщенных типов и членов.

Встроенные типы данных CTS

Финальный аспект спецификации CTS, который следует знать на текущий момент, заключается в том, что она устанавливает четко определенный набор фундаментальных типов данных. Хотя в каждом языке для объявления фундаментального типа данных обычно предусмотрено уникальное ключевое слово, ключевые слова всех языков .NET в конечном итоге соответствуют одному и тому же типу CTS, определенному в сборке `mscorlib.dll`. В табл. 1.2 показано, как ключевые типы данных из CTS выражаются в различных языках .NET.

С учетом того, что уникальные ключевые слова в любом управляемом языке являются просто сокращенными обозначениями реального типа из пространства имен `System`, больше не нужно беспокоиться ни об условиях переполнения и потери значимости для числовых данных, ни о внутреннем представлении строковых и булевских значений в различных языках.

Таблица 1.2. Встроенные типы данных CTS

Тип данных CTS	Ключевое слово VB	Ключевое слово C#	Ключевое слово C++/CLI
System.Byte	Byte	byte	unsigned char
System.SByte	SByte	sbyte	signed char
System.Int16	Short	short	short
System.Int32	Integer	int	int или long
System.Int64	Long	long	__int64
System.UInt16	UShort	ushort	unsigned short
System.UInt32	UInteger	uint	unsigned int или unsigned long
System.UInt64	ULong	ulong	unsigned __int64
System.Single	Single	float	float
System.Double	Double	double	double
System.Object	Object	object	object^
System.Char	Char	char	wchar_t
System.String	String	string	String^
System.Decimal	Decimal	decimal	Decimal
System.Boolean	Boolean	bool	bool

Ниже приведены фрагменты кода, определяющие 32-битные целочисленные переменные на C# и Visual Basic с использованием ключевых слов языка, а также формального типа CTS:

```
// Определение целочисленных переменных в C#.
int i = 0;
System.Int32 j = 0;
' Определение целочисленных переменных в VB.
Dim i As Integer = 0
Dim j As System.Int32 = 0
```

Понятие общезыковой спецификации (CLS)

Как вы уже знаете, разные языки программирования выражают одни и те же программные конструкции с помощью уникальных и специфичных для конкретного языка терминов. Например, в C# конкатенация строк обозначается с использованием операции "плюс" (+), а в VB для этого обычно служит амперсанд (&). Даже если два разных языка выражают одну и ту же программную идиому (например, функцию, не возвращающую значение), высока вероятность того, что синтаксис будет выглядеть сильно отличающимся:

```
// Ничего не возвращающий метод в C#.
public void MyMethod()
{
    // Некоторый код...
}
' Ничего не возвращающий метод в VB.
Public Sub MyMethod()
    ' Некоторый код...
End Sub
```

Ранее уже было показано, что такие небольшие синтаксические вариации для исполняющей среды .NET несущественны, учитывая, что соответствующие компиляторы (в этом случае — `csc.exe` и `vbc.exe`) генерируют схожий набор CIL-инструкций. Тем не менее, языки могут отличаться и по общему уровню функциональности. Например, какой-то язык .NET может как иметь, так и не иметь ключевое слово для представления данных без знака, а также поддерживать или не поддерживать типы указателей. С учетом таких возможных вариаций, было бы идеально располагать опорными требованиями, которым бы удовлетворяли все языки, ориентированные на .NET.

Спецификация CLS — это набор правил, подробно описывающих минимальное и полное множество характеристик, которые должен поддерживать отдельный компилятор .NET, чтобы генерировать программный код, обслуживаемый средой CLR и в то же время доступный в унифицированной манере всем языкам, ориентированным на платформу .NET. Во многих отношениях CLS можно рассматривать как *подмножество* полной функциональности, определяемой CTS.

В конечном итоге CLS является набором правил, которых должны придерживаться создатели компиляторов при желании, чтобы их продукты могли гладко функционировать в мире .NET. Каждое правило имеет простое название (например, “Правило номер 6”), и каждое правило описывает воздействие на тех, кто создает компиляторы, и на тех, кто (каким-либо образом) взаимодействует с ними. Наиболее важным в CLS является правило номер 1.

- *Правило номер 1.* Правила CLS применяются только к тем частям типа, которые видны извне определяющей сборки.

Из этого правила можно сделать корректный вывод о том, что остальные правила CLS не применяются к логике, используемой для построения внутренних рабочих деталей типа .NET. Единственными аспектами типа, которые должны соответствовать CLS, являются сами определения членов (т.е. соглашения об именовании, параметры и возвращаемые типы). Логика реализации члена может применять любое количество приемов, не согласованных с CLS, т.к. для внешнего мира это не играет никакой роли.

В целях иллюстрации ниже приведен метод `Add()` на C#, который не совместим с CLS, поскольку его параметры и возвращаемое значение используют данные без знака (что не является требованием CLS):

```
class Calc
{
    // Открытые данные без знака не совместимы с CLS!
    public ulong Add(ulong x, ulong y)
    {
        return x + y;
    }
}
```

Однако если просто работать с данными без знака внутри метода, как в следующем примере:

```
class Calc
{
    public int Add(int x, int y)
    {
        // Поскольку эта переменная ulong используется только
        // внутренне, совместимость с CLS не нарушается.
        ulong temp = 0;
        ...
        return x + y;
    }
}
```

то правила CLS по-прежнему соблюдаются и все языки .NET имеют возможность обращаться к этому методу `Add()`.

Разумеется, помимо правила 1 спецификация CLS определяет множество других правил. Например, в CLS описано, каким образом отдельный язык должен представлять текстовые строки, внутренне представлять перечисления (базовый тип, используемый для хранения), определять статические члены и т.д. К счастью, для того, чтобы стать умелым разработчиком .NET, запоминать все эти правила вовсе не обязательно. Очень хорошо разбираться в спецификациях CTS и CLS обычно должны только создатели инструментов и компиляторов.

Обеспечение совместимости с CLS

Как вы увидите в ходе изучения этой книги, в языке C# определено несколько программных конструкций, которые являются несовместимыми с CLS. Тем не менее, компилятор C# можно заставить выполнять проверку кода на предмет совместимости с CLS, используя единственный атрибут .NET:

```
// Сообщить компилятору C# о необходимости проверки на совместимость с CLS.
[assembly: CLSCompliant(true)]
```

Детали программирования на основе атрибутов более подробно рассматриваются в главе 15. А пока просто следует запомнить, что атрибут `[CLSCompliant]` заставляет компилятор C# проверять каждую строку кода на соответствие правилам CLS. В случае обнаружения нарушений любых правил CLS компилятор сообщит об ошибке с указанием проблемного кода.

Понятие общезыковой исполняющей среды (CLR)

Помимо спецификаций CTS и CLS, для получения общей картины на данный момент осталось рассмотреть еще одну аббревиатуру — CLR. С точки зрения программирования под термином исполняющая среда может пониматься коллекция служб, которые требуются для выполнения скомпилированной единицы кода. Например, когда разработчики на Java развертывают программное обеспечение на новом компьютере, они должны удостовериться в том, что на компьютере установлена виртуальная машина Java (Java Virtual Machine — JVM), которая сделает возможным выполнение их программного обеспечения.

Платформа .NET предлагает еще одну исполняющую среду. Ключевая разница между исполняющей средой .NET и упомянутыми выше средами состоит в том, что исполняющая среда .NET обеспечивает единый четко определенный уровень выполнения, который разделяется всеми языками и платформами, ориентированными на .NET.

Основной механизм CLR физически имеет вид библиотеки по имени `mscorlib.dll` (также известной как общий механизм выполнения исполняемого кода объектов (Common Object Runtime Execution Engine)). Когда на сборку производится ссылка с целью ее использования, библиотека `mscorlib.dll` загружается автоматически и, в свою очередь, загружает требуемую сборку в память. Исполняющая среда отвечает за решение множества задач. Первая и наиглавнейшая задача — определение местоположения сборки и нахождение запрошенного типа в двоичном файле за счет чтения содержащихся в нем метаданных. Затем CLR размещает тип в памяти, преобразует связанный с ним CIL-код в специфичные для платформы инструкции, производит все необходимые проверки безопасности и после этого выполняет нужный код.

В дополнение к загрузке специальных сборок и созданию специальных типов, среда CLR при необходимости будет взаимодействовать с типами, содержащимися в библиотеках базовых классов .NET. Хотя вся библиотека базовых классов разделена на отдельные сборки, главной среди них является сборка `microsoft.dll`. Эта сборка содержит большое количество основных типов, которые инкапсулируют широкий спектр общих задач программирования, а также основные типы данных, используемые всеми языками .NET. При построении решений .NET доступ к этой сборке предоставляется автоматически.

На рис. 1.4 показан рабочий поток, возникающий между исходным кодом (в котором применяются типы из библиотеки базовых классов), отдельным компилятором .NET и исполняющей средой .NET.

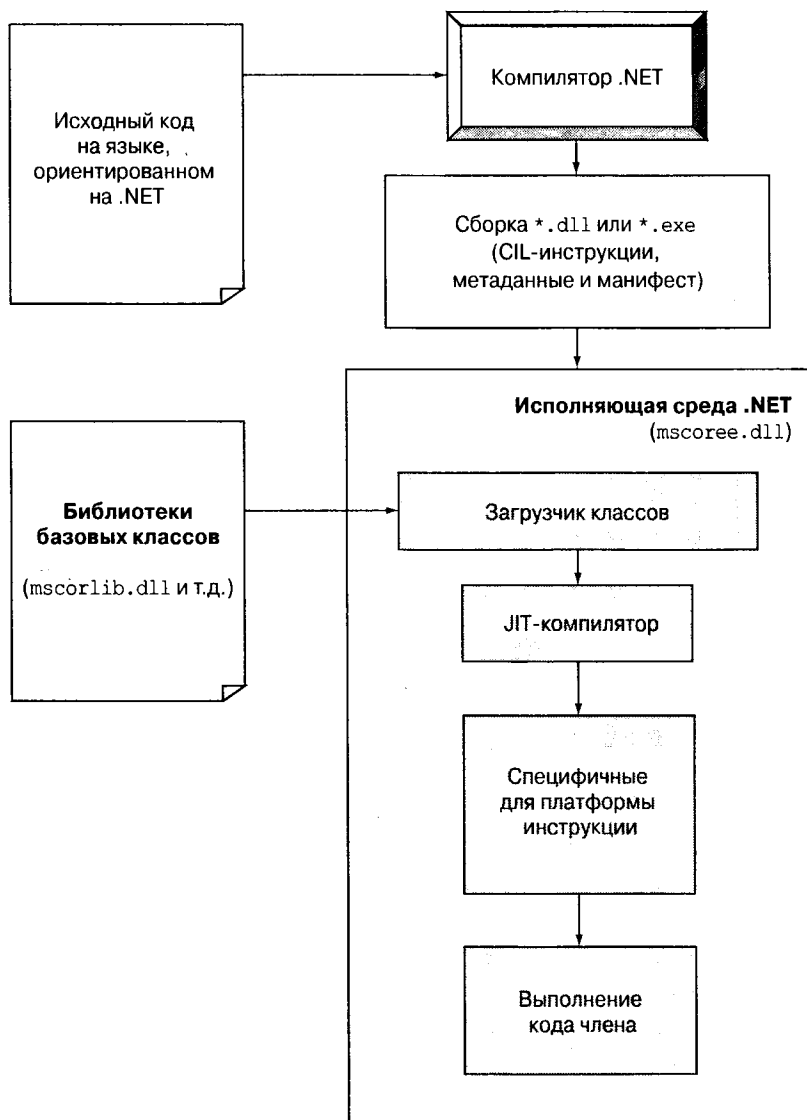


Рис. 1.4. Сборка `mscorlib.dll` в действии

Различия между сборками, пространствами имен и типами

Важность библиотек кода понимает любой из нас. Главное назначение библиотек платформы — предоставлять разработчикам четко определенный набор готового кода для использования в создаваемых приложениях. Однако С# не поставляется с какой-то специфичной для языка библиотекой кода. Вместо этого разработчики на С# пользуются нейтральными к языкам библиотеками .NET. Для поддержания всех типов внутри библиотек базовых классов в хорошо организованном виде в рамках платформы .NET широко применяется концепция пространства имен.

Пространство имен — это группа семантически связанных типов, которые содержатся в одной или нескольких связанных друг с другом сборках. Например, пространство имен `System.IO` содержит типы, имеющие отношение к файловому вводу-выводу, пространство имен `System.Data` — типы для работы с базами данных и т.д. Очень важно понимать, что в одной сборке (вроде `mscorlib.dll`) может содержаться любое количество пространств имен, каждое из которых может иметь любое число типов.

Чтобы стало понятнее, на рис. 1.5 показана копия экрана браузера объектов Visual Studio. Этот инструмент позволяет просматривать сборки, на которые имеются ссылки в текущем проекте, пространства имен внутри отдельной сборки, типы в конкретном пространстве имени и члены специфического типа. Обратите внимание, что сборка `mscorlib.dll` содержит множество разных пространств имен (таких как `System.IO`), каждое из которых имеет собственные семантически связанные типы (например, `BinaryReader`).

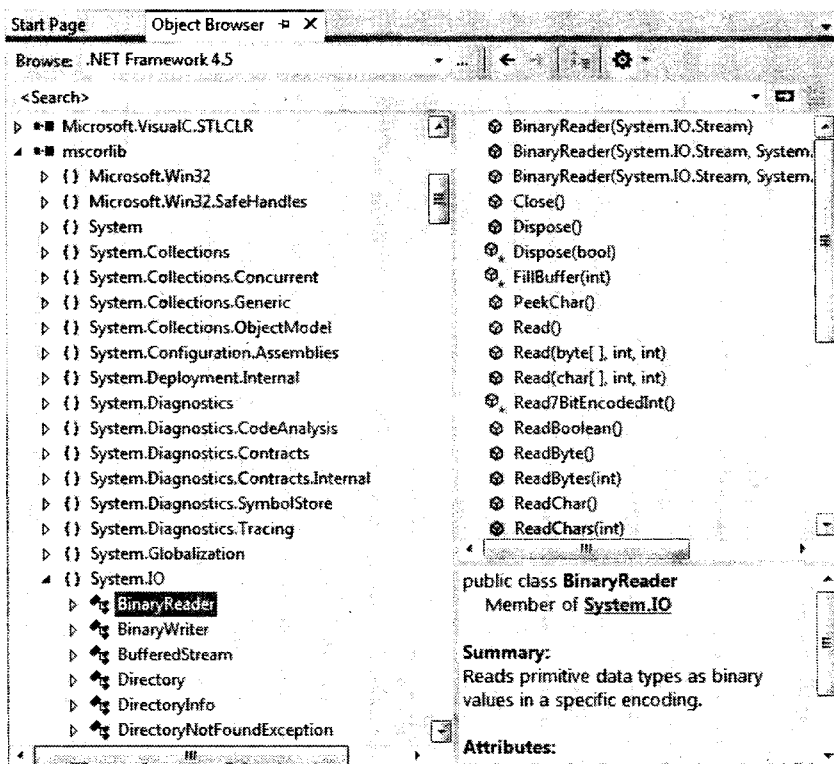


Рис. 1.5. Сборка может иметь любое количество пространств имен

Главная разница между таким подходом и специфичной для языка библиотекой заключается в том, что любой язык, ориентированный на исполняющую среду .NET, использует *те же самые* пространства имен и *те же самые* типы. Например, ниже приведен код вездесущего приложения "Hello World" на языках C#, VB и C++/CLI:

// Приложение "Hello World" на языке C#.

```
using System;

public class MyApp
{
    static void Main()
    {
        Console.WriteLine("Hi from C#");
    }
}
```

' Приложение "Hello World" на языке VB.

```
Imports System

Public Module MyApp
    Sub Main()
        Console.WriteLine("Hi from VB")
    End Sub
End Module
```

// Приложение "Hello World" на языке C++/CLI

```
#include "stdafx.h"
using namespace System;

int main(array<System::String ^> ^args)
{
    Console::WriteLine(L"Hi from C++/CLI");
    return 0;
}
```

Обратите внимание, что в коде на каждом языке применяется класс `Console`, определенный в пространстве имен `System`. Помимо очевидных синтаксических различий, эти три приложения выглядят очень похожим образом, как физически, так и логически.

Понятно, что основной целью любого разработчика для .NET является освоение обилия типов, которые определены в (многочисленных) пространствах имен .NET. Наиболее фундаментальное пространство имен, с которого следует начинать, называется `System`. Это пространство имен предоставляет набор ключевых типов, которые любой разработчик для .NET будет эксплуатировать снова и снова. Фактически создание любого маломальски функционального приложения на C# невозможно без добавления, по крайней мере, ссылки на пространство имен `System`, поскольку в нем определены все главные типы данных (например, `System.Int32`, `System.String` и т.д.). В табл. 1.3 приведен краткий список некоторых (но, конечно же, не всех) пространств имен .NET, сгруппированных на основе функциональности.

Любое пространство имен, вложенное в `Microsoft` (к примеру, `Microsoft.CSharp`, `Microsoft.ManagementConsole`, `Microsoft.Win32`), содержит типы, которые используются для взаимодействия со службами, уникальными для операционной системы `Windows`. Учитывая это, вы не должны предполагать, что эти типы могут успешно применяться в других операционных системах, поддерживающих .NET, таких как `Mac OS X`. По большей части, в настоящей книге мы не будем углубляться в детали пространств имен, вложенных в `Microsoft`, поэтому, если вам интересно, обращайтесь по этому поводу к документации .NET Framework 4.5 SDK.

Таблица 1.3. Некоторые пространства имен в .NET

Пространство имен .NET	Описание
System	Внутри пространства имен System содержится множество полезных типов, предназначенных для работы с внутренними данными, математическими вычислениями, генерацией случайных чисел, переменными среды и сборкой мусора, а также ряд часто применяемых исключений и атрибутов
System.Collections System.Collections.Generic	Эти пространства имен определяют набор контейнерных типов, а также базовые типы и интерфейсы, которые позволяют строить настраиваемые коллекции
System.Data System.Data.Common System.Data.EntityClient System.Data.SqlClient	Эти пространства имен используются для взаимодействия с базами данных через ADO.NET
System.IO System.IO.Compression System.IO.Ports	Эти пространства имен определяют множество типов, предназначенных для работы с файловым вводом-выводом, сжатием данных и портами
System.Reflection System.Reflection.Emit	Эти пространства имен определяют типы, которые поддерживают обнаружение типов во время выполнения, а также динамическое создание типов
System.Runtime.InteropServices	Это пространство имен предоставляет средства, позволяющие типам .NET взаимодействовать с неуправляемым кодом (например, DLL-библиотеками на основе C и серверами COM) и наоборот
System.Drawing System.Windows.Forms	Эти пространства имен определяют типы, применяемые для построения настольных приложений с использованием исходного инструментального набора .NET для создания пользовательских интерфейсов (Windows Forms)
System.Windows System.Windows.Controls System.Windows.Shapes	Пространство имен System.Windows является корневым для нескольких пространств имен, которые представляют инструментальный набор для построения пользовательских интерфейсов Windows Presentation Foundation (WPF)
System.Linq System.Xml.Linq System.Data.DataSetExtensions	Эти пространства имен определяют типы, применяемые во время программирования с использованием API-интерфейса LINQ
System.Web	Это одно из многих пространств имен, которые позволяют строить веб-приложения ASP.NET
System.ServiceModel	Это одно из многих пространств имен, используемых для построения распределенных приложений с помощью API-интерфейса Windows Communication Foundation (WCF)
System.Workflow.Runtime System.Workflow.Activities	Это два из многочисленных пространств имен, которые определяют типы, применяемые при построении приложений, поддерживающих рабочие потоки, с помощью API-интерфейса Windows Workflow Foundation (WF)
System.Threading System.Threading.Tasks	Это пространство имен определяет многочисленные типы для построения многопоточных приложений, которые могут распределять рабочую нагрузку по нескольким центральным процессорам
System.Security	Безопасность является неотъемлемой характеристикой мира .NET. В пространствах имен, связанных с безопасностью, содержится множество типов, которые позволяют работать с разрешениями, криптографией и т.д.
System.Xml	В пространствах имен, связанных с XML, содержатся многочисленные типы, используемые для взаимодействия с XML-данными

Роль корневого пространства имен Microsoft

При изучении перечня, приведенного в табл. 1.3, вы наверняка заметили, что `System` является корневым пространством имен для большинства вложенных пространств имен (таких как `System.IO`, `System.Data` и т.д.). Однако оказывается, что помимо `System` в библиотеке базовых классов определено несколько других корневых пространств имен наивысшего уровня, самое полезное из которых называется `Microsoft`.

На заметку! В главе 2 будет показано, как пользоваться документацией .NET Framework 4.5 SDK, в которой содержатся детальные описания всех пространств имен, типов и членов, встречающихся в библиотеках базовых классов.

Доступ к пространству имен программным образом

Полезно снова повторить, что пространство имен — это не более чем удобный способ логической организации связанных типов для упрощения работы с ними. Давайте еще раз обратимся к пространству имен `System`. С точки зрения разработчика можно считать, что конструкция `System.Console` представляет класс по имени `Console`, который содержится внутри пространства имен под названием `System`. Однако с точки зрения исполняющей среды .NET это не так. Исполняющая среда видит только единственный класс по имени `System.Console`.

В C# ключевое слово `using` упрощает процесс добавления ссылок на типы, определенные в заданном пространстве имен. Вот как оно работает. Предположим, что требуется построить графическое настольное приложение с использованием API-интерфейса WPF. Хотя изучение типов в каждом пространстве имен предполагает исследование и экспериментирование, ниже показаны некоторые из возможных кандидатов на ссылку из такой программы:

```
// Некоторые возможные пространства имен, используемые
// для построения WPF-приложения.
using System;                // Общие типы из библиотек базовых классов.
using System.Windows.Shapes; // Типы для графической визуализации.
using System.Windows.Controls; // Типы виджетов графического интерфейса
                               // Windows Forms.
using System.Data;           // Общие типы, связанные с данными.
using System.Data.SqlClient; // Типы доступа к данным MS SQL Server.
```

После указания ряда необходимых пространств имен (и установки ссылки на сборки, в которых они определены), можно свободно создавать экземпляры типов, которые в них содержатся. Например, если нужно создать экземпляр класса `Button` (определенного в пространстве имен `System.Windows.Controls`), можно написать следующий код:

```
// Явно перечислить пространства имен, используемые в этом файле.
using System;
using System.Windows.Controls;

class MyGUIBuilder
{
    public void BuildUI()
    {
        // Создать элемент управления типа кнопки.
        Button btnOK = new Button();
        ...
    }
}
```

Благодаря импортированию в файле кода пространства имен `System.Windows.Controls`, компилятор имеет возможность выяснить, что класс `Button` является членом данного пространства имен. Если не импортировать пространство имен `System.Windows.Controls`, компилятор сообщит об ошибке. При желании переменные также можно объявлять с использованием *полностью заданного имени*:

```
// Пространство имен System.Windows.Controls здесь не указано!
using System;

class MyGUIBuilder
{
    public void BuildUI()
    {
        // Использование полностью заданного имени.
        System.Windows.Controls.Button btnOK =
            new System.Windows.Controls.Button();
        ...
    }
}
```

Хотя определение типа с использованием полностью заданного имени позволяет делать код более читабельным, трудно не согласиться с тем, что применение ключевого слова `using` в C# значительно сокращает объем набора на клавиатуре. В этой книге полностью заданные имена в основном использоваться не будут (разве что для устранения установленной неоднозначности), а предпочтение отдается упрощенному подходу с применением ключевого слова `using`.

Однако всегда помните о том, что ключевое слово `using` — это просто сокращенная нотация для указания полностью заданного имени типа, поэтому любой из этих подходов дает в результате *точно* такой же CIL-код (учитывая, что в CIL-коде всегда используются полностью заданные имена) и не влияет ни на производительность, ни на размер сборки.

Ссылка на внешние сборки

В дополнение к указанию пространства имен через ключевое слово `using` в C# компилятору C# также необходимо сообщить имя сборки, содержащей действительное CIL-определение типа, на который производится ссылка. Как уже отмечалось, многие ключевые пространства имен .NET определены внутри сборки `mscorlib.dll`. Однако класс `System.Drawing.Bitmap` содержится в отдельной сборке по имени `System.Drawing.dll`. Подавляющее большинство сборок .NET Framework размещено в специальном каталоге, который называется глобальным кешем сборок (*global assembly cache* — GAC). На машине Windows по умолчанию GAC может быть найден внутри каталога `C:\Windows\Assembly`, как показано на рис. 1.6.

В зависимости от инструмента, применяемого для разработки приложений .NET, может существовать несколько способов информирования компилятора о том, какие сборки необходимо включить в цикл компиляции. Все эти способы подробно рассматриваются в главе 2, поэтому здесь их детали опущены.

На заметку! Начиная с версии .NET 4.0, в Microsoft решили выделить под сборки .NET 4.0 и .NET 4.5 специальное место, находящееся отдельно от каталога `C:\Windows\Assembly`. Более подробно об этом будет рассказываться в главе 14.

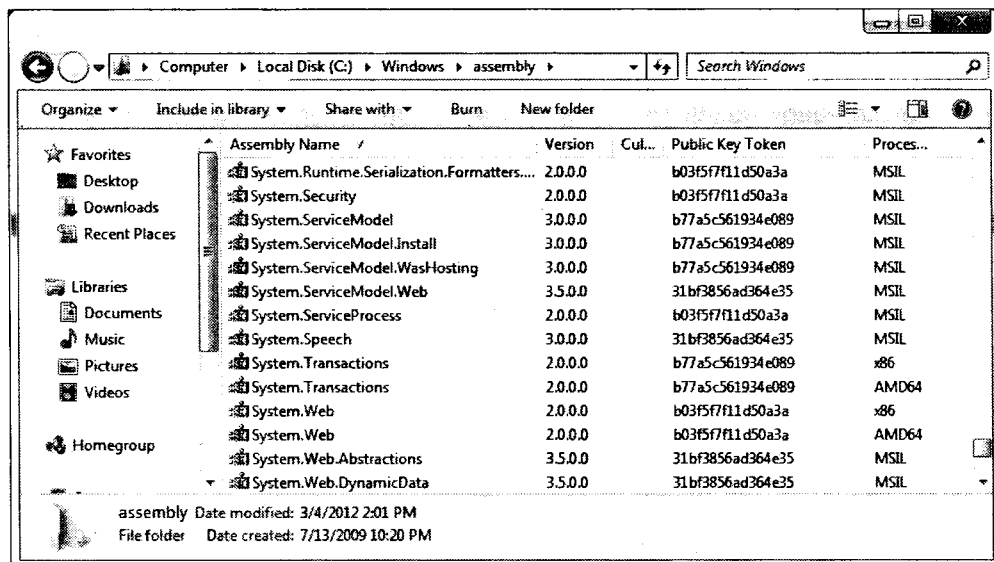


Рис. 1.6. Многие библиотеки .NET размещены в GAC

Исследование сборки с помощью утилиты ildasm.exe

Если вас начинает беспокоить мысль о необходимости освоения всех пространств имен .NET, просто вспомните о том, что уникальным пространство имен делает то, что в нем содержатся типы, которые каким-то образом связаны семантически. Поэтому если в качестве пользовательского интерфейса достаточно простого консольного приложения, можете вообще не думать о пространствах имен для построения интерфейсов настольных и веб-приложений. Если вы создаете приложение рисования, то пространства имен для работы с базами данных, скорее всего, не понадобятся. Как и в случае любого нового набора готового кода, изучение должно проходить постепенно.

Утилита ildasm.exe (Intermediate Language Disassembler — дизассемблер промежуточного языка), которая поставляется в составе .NET Framework 4.5 SDK, позволяет загрузить любую сборку .NET и изучить ее содержимое, включая ассоциированный с ней манифест, CIL-код и метаданные типов. Этот инструмент позволяет программистам более подробно разобраться, как их код C# отображается на CIL-код, и в конечном итоге помогает понять внутреннюю механику работы платформы .NET. Хотя использование ildasm.exe вовсе не обязательно для того, чтобы стать опытным программистом для .NET, настоятельно рекомендуется время от времени применять этот инструмент, чтобы лучше понять, как написанный код C# вписывается в концепции исполняющей среды.

На заметку! Чтобы запустить утилиту ildasm.exe, откройте окно Developer Command Prompt (Командная строка разработчика), введите в нем ildasm и нажмите клавишу <Enter>. В главе 2 будет показано, как открыть это "специальное" командное окно, а также другие инструменты, загружаемые из командной строки.

После запуска утилиты ildasm.exe выберите пункт меню File⇒Open (Файл⇒Открыть) и перейдите к сборке, которую требуется исследовать. В целях иллюстрации на рис. 1.7 показана сборка Calc.exe, которая была сгенерирована на основе приведенного ранее в этой главе файла Calc.cs. Утилита ildasm.exe представляет структуру любой сборки в знакомом древовидном формате.

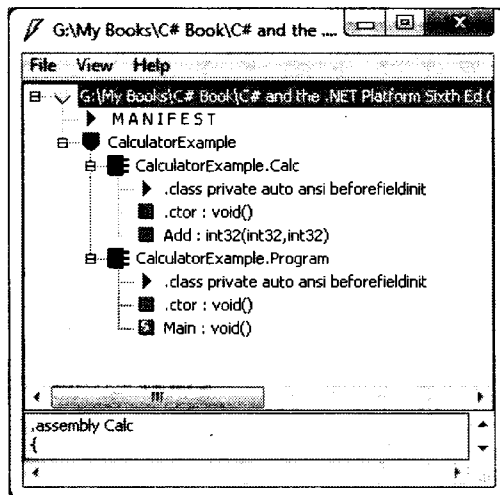


Рис. 1.7. Утилита ildasm.exe позволяет видеть содержащиеся внутри сборки .NET код CIL, манифест и метаданные типов

Просмотр CIL-кода

Кроме содержащихся в сборке пространств имен, типов и членов, утилита ildasm.exe также позволяет просматривать и CIL-инструкции для конкретного члена. Например, если дважды щелкнуть на методе Main() в классе Program, откроется отдельное окно с CIL-кодом этого метода (рис. 1.8).

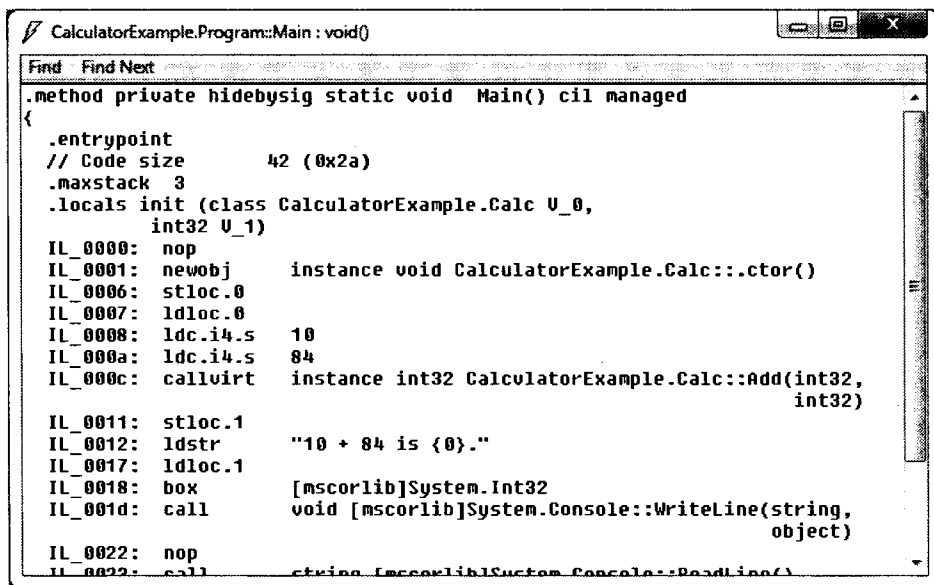


Рис. 1.8. Просмотр CIL-кода для метода

Просмотр метаданных типов

Для просмотра метаданных типов, содержащихся в загруженной в текущий момент сборке, нажмите комбинацию клавиш <Ctrl+M>. На рис. 1.9 показаны метаданные для метода Calc.Add().

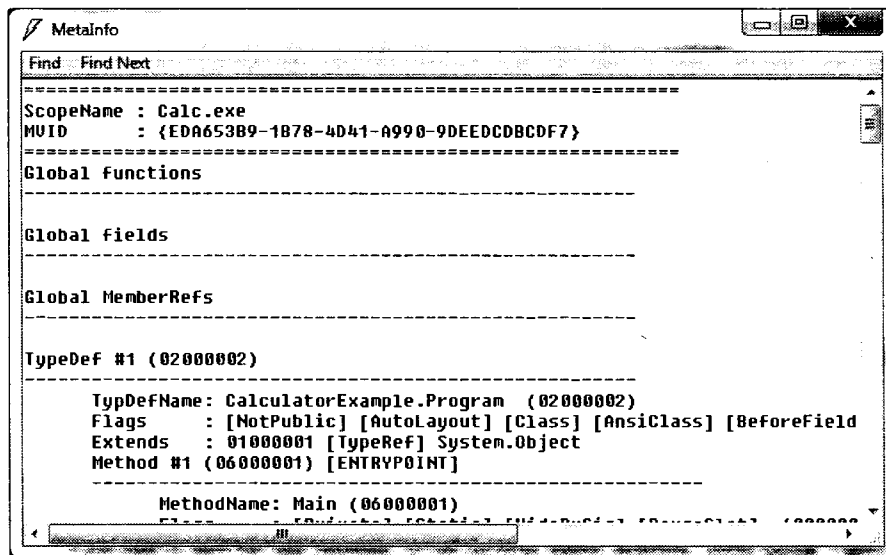


Рис. 1.9. Просмотр метаданных типов с помощью ildasm.exe

Просмотр метаданных сборки (манифеста)

Наконец, чтобы просмотреть содержимое манифеста сборки, дважды щелкните на значке MANIFEST (рис. 1.10).

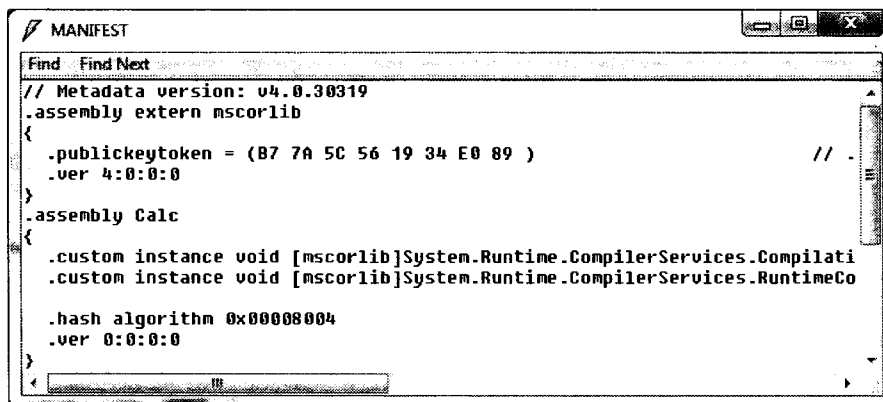


Рис. 1.10. Просмотр данных манифеста с помощью ildasm.exe

Несомненно, утилита ildasm.exe обладает намного большим набором возможностей, чем было показано здесь, и они будут демонстрироваться в книге, когда это уместно.

Независимая от платформы природа .NET

Теперь хотелось бы сказать несколько слов о независимой от платформы природе .NET. К удивлению большинства разработчиков, сборки .NET могут разрабатываться и выполняться в средах операционных систем, отличных от Microsoft, включая Mac OS X, различные дистрибутивы Linux, Solaris, а также на мобильных устройствах iOS и Android (через API-интерфейс MonoTouch). Чтобы понять, почему это возможно, необходимо рассмотреть еще одну аббревиатуру из мира .NET — CLI (Common Language Infrastructure — общезыковая инфраструктура).

Вместе с языком программирования C# и платформой .NET в Microsoft также разработали набор официальных документов с описанием синтаксиса и семантики языков C# и CIL, формата сборок .NET, ключевых пространств имен .NET и технических деталей функционирования гипотетической исполняющей среды .NET (которая называется виртуальной системой выполнения (Virtual Execution System — VES)).

Все эти документы были поданы в организацию Ecma International (www.ecma-international.org) и утверждены в качестве официальных международных стандартов. Среди них наибольший интерес представляют:

- документ ECMA-334, в котором содержится спецификация языка C#;
- документ ECMA-335, в котором содержится спецификация общезыковой инфраструктуры (CLI).

Важность этих документов становится очевидной с пониманием того факта, что они предоставляют третьим сторонам возможность создавать дистрибутивы платформы .NET для любого количества операционных систем и/или процессоров. Документ ECMA-335 является более насыщенным из этих двух спецификаций, поэтому он разбит на разделы, которые описаны в табл. 1.4.

Таблица 1.4. Разделы спецификации CLI

Раздел документа ECMA-335	Описание
Раздел I. Концепции и архитектура	Описывает общую архитектуру CLI, в том числе правила CTS и CLS и технические детали функционирования исполняющей среды .NET
Раздел II. Определение и семантика метаданных	Описывает детали метаданных и формат сборок .NET
Раздел III. Набор инструкций CIL	Описывает синтаксис и семантику кода CIL
Раздел IV. Профили и библиотеки	Предоставляет высокоуровневый обзор минимальных и полных библиотек классов, которые должны поддерживаться дистрибутивом .NET
Раздел V. Формат обмена информацией отладки	Описывает стандартный способ обмена информацией отладки между создателями и потребителями CLI
Раздел VI. Приложения	Предоставляет набор разрозненных деталей, таких как руководства по проектированию библиотек классов и особенности реализации компилятора CIL

Следует иметь в виду, что в разделе IV (Профили и библиотеки) описан лишь минимальный набор пространств имен, которые представляют ожидаемые CLI службы (например, коллекции, консольный ввод-вывод, файловый ввод-вывод, многопоточная обработка, рефлексия, доступ в сеть, основные средства защиты, манипулирование XML-данными). В CLI не определяются пространства имен, которые упрощают разработку веб-приложений (ASP.NET), доступ к базам данных (ADO.NET) и создание настольных приложений с графическим пользовательским интерфейсом (Windows Presentation Foundation или Windows Forms).

Хорошая новость состоит в том, что главные дистрибутивы .NET расширяют библиотеки CLI совместимыми с Microsoft эквивалентами реализаций ASP.NET, ADO.NET (и т.д.), чтобы обеспечивать полнофункциональные платформы для разработки приложений производственного уровня. На сегодняшний день популярностью пользуются две основные реализации CLI (помимо решения, предлагаемого Microsoft и рассчитанного на Windows). Хотя эта книга посвящена созданию .NET-приложений с помощью дистрибутива .NET от Microsoft, в табл. 1.5 приведена краткая информация о проектах Mono и Portable.NET.

Таблица 1.5. Дистрибутивы .NET с открытым кодом

Дистрибутив	Описание
www.mono-project.com	Проект Mono — это дистрибутив CLI с открытым кодом, который ориентирован на различные версии Linux (например, SuSE, Fedora), Mac OS X, устройства iOS (iPad, iPhone), устройства Android и (как ни странно) Windows
www.dotgnu.org	Проект Portable.NET — это еще один дистрибутив CLI с открытым кодом, который может функционировать под управлением множества операционных систем. Проект Portable.NET стремится охватить максимально возможное число операционных систем (например, Windows, AIX, Mac OS X, Solaris и все основные дистрибутивы Linux)

Проекты Mono и Portable.NET предоставляют ECMA-совместимый компилятор C#, исполняющую среду .NET, примеры кода, документацию, а также многочисленные инструменты для разработки приложений, которые по своим функциональным возможностям эквивалентны средствам, поставляемым в составе .NET Framework 4.5 SDK.

На заметку! Введение в процесс разработки межплатформенных .NET-приложений с использованием Mono можно найти в приложении А (доступном для загрузки на веб-сайте издательства).

Несколько слов по поводу приложений Windows 8

В завершение этой главы кратко остановимся на взаимосвязи платформы Microsoft .NET с совершенно новым набором технологий, представленных вместе с операционной системой Windows 8. Если у вас еще не было возможности ознакомиться с Windows 8, то следует отметить, что новый пользовательский интерфейс радикально отличается от предшествующих версий Microsoft Windows (фактически, можно сказать, что изменение столь же радикально, как и переход от Windows 3.11 к Windows 95 — для тех, кто еще помнит, что собой представляли эти ОС). Новый стартовый экран ОС Windows 8 показан на рис. 1.11.

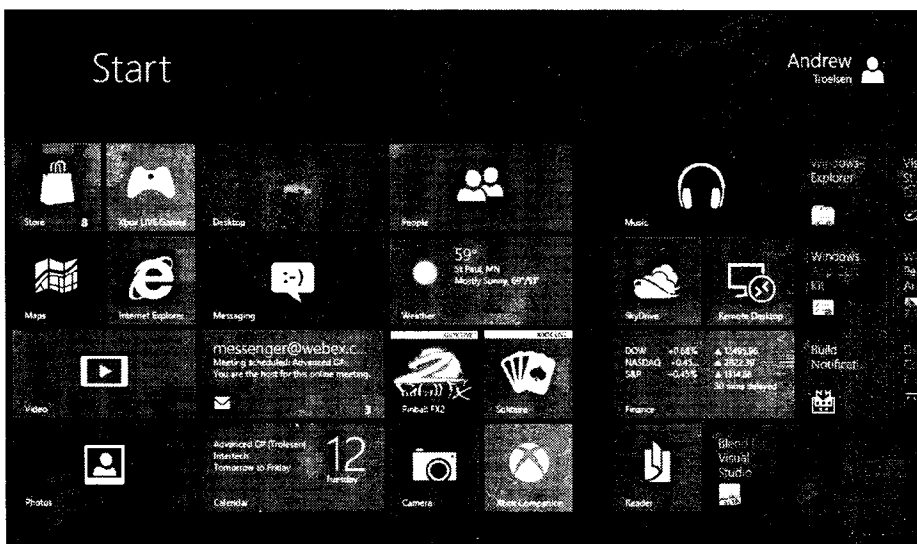


Рис. 1.11. Стартовый экран ОС Windows 8

Как видно на рис. 1.11, стартовый экран Windows 8 поддерживает мозаичную компоновку, где каждый элемент мозаики представляет приложение, установленное на компьютере. В отличие от типичного настольного Windows-приложения, приложения Windows 8 построены для взаимодействия с сенсорным экраном. Кроме того, они выполняются в полноэкранном представлении и лишены привычной “отделки” (т.е. систем меню, строк состояния и кнопок панелей инструментов), которую можно встретить во многих приложениях для настольных компьютеров. Часть экрана приложения для выдачи прогноза погоды в Windows 8 показана на рис. 1.12.

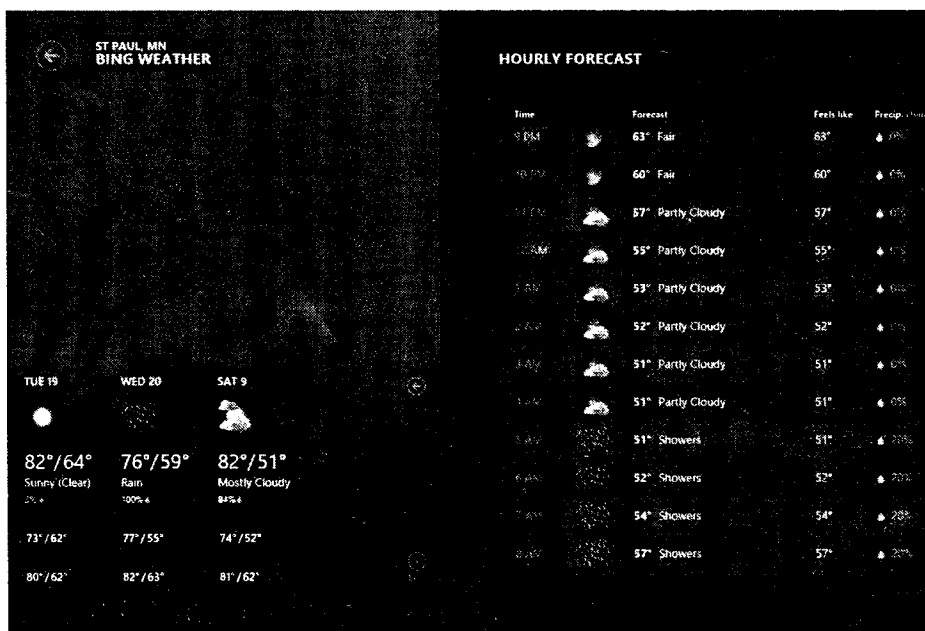


Рис. 1.12. Приложения Windows 8 представляют собой полноэкранные, насыщенные графикой настольные приложения

На заметку! Построение приложений Windows 8 требует от разработчиков овладения совершенно новым набором приемов создания пользовательских интерфейсов, методологиями хранения данных и вариантами пользовательского ввода. Приложение Windows 8 — это значительно больше, чем элемент мозаики, установленный на стартовом экране.

Построение приложений в стиле Metro

Создание и выполнение приложения Windows 8 возможно только в среде этой операционной системы и не поддерживается в среде Windows 7. Фактически, если установить Visual Studio в системе Windows 7 (или более ранней), шаблоны проектов Windows 8 даже не отобразятся в диалоговом окне New Project (Новый проект).

Программирование приложения Windows 8 требует от разработчиков перехода на совершенно новый уровень исполняющей среды, получивший название (достаточно точное) *исполняющей среды Windows* (*Windows Runtime* — WinRT). Следует постоянно помнить, что WinRT — это не то же самое, что CLR в .NET, однако она предоставляет ряд аналогичных служб, таких как сборка мусора, поддержка множества языков программирования (в том числе C#) и т.д.

Кроме того, эти приложения создаются с применением совершенно нового набора пространств имен, которые все начинаются с корневого имени Windows. Различные пространства имен WinRT, отображенные в браузере объектов Visual Studio, показаны на рис. 1.13.

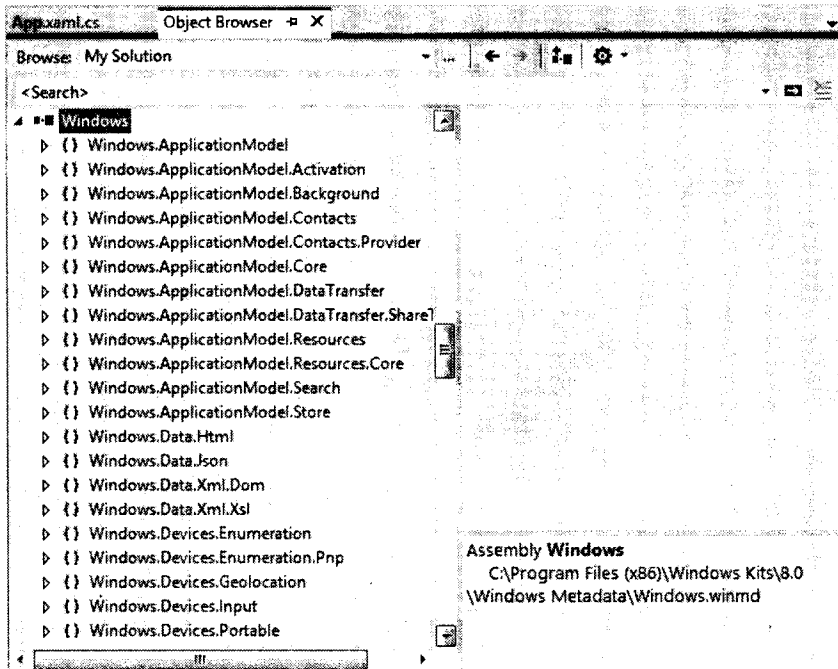


Рис. 1.13. Пространства имен Windows.* используются для построения приложений только для Windows 8

В качестве своеобразной компенсации пространства имен Windows.* предлагают функциональность, которая повторяет многие API-интерфейсы библиотек базовых классов .NET. В действительности с точки зрения программирования построение приложения для WinRT подобно построению приложения .NET для CLR.

Например, приложения Windows 8 могут быть созданы с применением языка C# (а также Visual Basic, JavaScript или C++).

Многие пространства имен Windows.* предоставляют также функциональные возможности, аналогичные (хотя и не идентичные) присутствующим в библиотеках базовых классов .NET. В качестве примера позднее в этой книге (в главах 27–31) мы рассмотрим технологию *Windows Presentation Foundation* (WPF). В этих главах вы узнаете об основанном на XML языке под названием XAML. Начав знакомство с построением приложений Windows 8, вы убедитесь, что они также создаются с применением XAML (или HTML5, если в качестве основного языка выбран JavaScript) и обладают моделью программирования, очень напоминающей модель WPF-программ, создаваемых на платформе .NET.

Роль .NET в среде Windows 8

В дополнение к пространствам имен Windows.*, приложения Windows 8 могут использовать также большой поднабор платформы .NET. Все вместе API-интерфейсы .NET для приложений в стиле Metro предоставляют поддержку обобщенных коллекций, LINQ,

обработки XML-документов, служб ввода-вывода, служб безопасности и других аспектов, описанных в этой книге. Безусловно, это радует, поскольку многие рассмотренные в этой книге темы найдут непосредственное отражение в конструировании приложений Windows 8.

Помните также, что хотя классический рабочий стол Windows не является используемым по умолчанию отображением в Windows 8, он остается доступным посредством щелчка на элементе мозаики Desktop (Рабочий стол) или просто нажатия кнопки Windows на клавиатуре. После его открытия ОС отобразит рабочий стол, который почти ничем не отличается от рабочего стола Windows 7 (рис. 1.14).

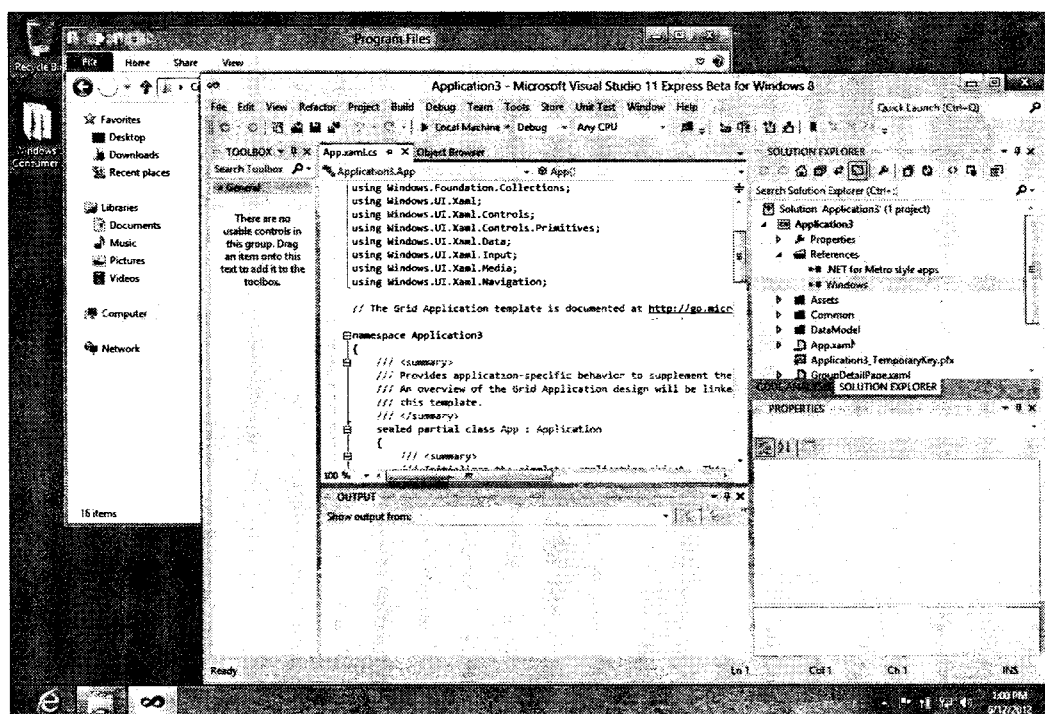


Рис. 1.14. Рабочий стол Windows 8 позволяет запускать приложения, разработанные не для Windows 8

Отсюда можно запускать все привычные приложения рабочего стола, в том числе Visual Studio, Word, Excel, Photoshop и т.п. Если приложения .NET создаются в среде Windows 8, они будут запускаться в среде рабочего стола Windows 8.

Одним словом, платформа Microsoft .NET продолжает жить и здравствовать в среде Windows 8 и остается основной частью среды разработки. Нужно только помнить, что для построения приложений Windows 8 элементы .NET не используются. Вместо этого для создания таких приложений придется применять новые библиотеки (Windows.*), новую исполняющую среду (WinRT) и новый поднабор характерных компонентов .NET (API-интерфейсы .NET).

В этой книге мы не будем рассматривать процесс построения приложений Windows 8 с помощью WinRT, а сосредоточим внимание исключительно на платформе .NET. Однако помните, что большинство рассмотренных в этой книге тем почти наверняка облегчит изучение разработки приложений Windows 8 в будущем, если возникнет такая необходимость.

Резюме

Задача настоящей главы заключалась в изложении концептуальных базовых сведений для обеспечения успешного освоения всего остального материала книги. Сначала были рассмотрены ограничения и сложности, которые существовали в технологиях, предшествовавших появлению .NET, а потом было показано, как .NET и C# упрощают существующее положение вещей.

Главную роль в .NET, по сути, играют механизм исполняющей среды (`mscoree.dll`) и библиотека базовых классов (`mscorlib.dll` вместе со связанными файлами). Общеязыковая исполняющая среда (CLR) способна обслуживать любые двоичные модули (или сборки) .NET, подчиняющиеся правилам управляемого кода. Как было показано в этой главе, в каждой сборке (помимо метаданных типов и манифеста) содержатся CIL-инструкции, которые с помощью JIT-компилятора преобразуются в инструкции, ориентированные на конкретную платформу. Кроме того, вы ознакомились с ролью общеязыковой спецификации (CLS) и общей системы типов (CTS).

После этого было рассказано о таких полезных утилитах для просмотра объектов, как `ildasm.exe` и `reflector.exe`, а также о том, как сконфигурировать машину для обслуживания приложений .NET с помощью полного и клиентского профилей. И, наконец, напоследок было кратко упомянуто о преимуществах независимой от платформы природы C# и .NET (тема, которая дополнительно рассматривается в приложении А) и о месте платформы .NET в операционной системе Windows 8.

Создание приложений на языке C#

Программистам на C# доступно множество инструментов для построения .NET-приложений. Целью этой главы является предоставление краткого обзора разнообразных доступных средств для разработки .NET-приложений, включая, естественно, Visual Studio. Глава начинается с объяснений, как работать с компилятором командной строки C#, `csc.exe`, и простейшим из всех текстовых редакторов Notepad (Блокнот), который входит в состав операционной системы Microsoft Windows, а также приложением Notepad++, доступным для бесплатной загрузки.

Хотя для изучения приведенного в книге материала вполне хватило бы компилятора `csc.exe` и простейшего текстового редактора, вас наверняка интересует работа с многофункциональными интегрированными средами разработки (integrated development environment — IDE). В связи с этим в данной главе описана также бесплатная IDE-среда с открытым кодом SharpDevelop, предназначенная для разработки .NET-приложений. Кроме того, в главе кратко рассматривается IDE-среда Visual C# Express (распространяемая бесплатно), а также ключевые средства среды Visual Studio Professional.

На заметку! В этой главе будут встречаться синтаксические конструкции C#, которые пока еще не рассматривались. Если вы с ними не знакомы, не переживайте. Формальное изучение языка C# начнется в главе 3.

Роль .NET Framework 4.5 SDK

Одним из многих мифов в области разработки .NET-приложений является уверенность в том, что для разработки приложений на C# программисты должны обязательно приобретать копию Visual Studio. В действительности строить .NET-приложения любого рода можно с использованием бесплатно загружаемого комплекта инструментов для разработки программного обеспечения .NET Framework 4.5 Software Development Kit (SDK).

Внутри SDK предоставляются многочисленные управляемые компиляторы, утилиты командной строки, примеры кода, библиотеки классов .NET и полноценная система документации. Если вы планируете применять средства Visual Studio или Visual C# Express, имейте в виду, что нет никакой необходимости вручную загружать или устанавливать .NET Framework 4.5 SDK. При установке любого из упомянутых продуктов комплект SDK устанавливается автоматически, таким образом, предоставляя все необходимое. Однако если вы не собираетесь использовать IDE-среду от Microsoft при работе с настоящей книгой, обязательно установите SDK, прежде чем двигаться дальше.

На заметку! Программа установки .NET Framework 4.5 SDK (dotNetFx45_Full_x86_x64.exe) доступна на странице загрузки .NET по адресу <http://msdn.microsoft.com/netframework>.

Окно командной строки разработчика

При установке .NET Framework 4.5 SDK, Visual Studio или Visual C# Express на локальном жестком диске создается набор новых каталогов, в каждом из которых содержатся разнообразные инструменты для разработки .NET-приложений. Многие из этих инструментов работают в режиме командной строки, и чтобы использовать их в любом каталоге, нужно сначала зарегистрировать соответствующие пути в операционной системе.

Хотя можно было бы вручную обновить переменную PATH своей машины, имеет смысл сэкономить немного времени, воспользовавшись окном командной строки разработчика (Developer Command Prompt), которое доступно из папки Start⇒All Programs⇒Microsoft Visual Studio 11⇒Visual Studio Tools (Пуск⇒Все программы⇒Microsoft Visual Studio 11⇒Инструменты Visual Studio) и показано на рис. 2.1.

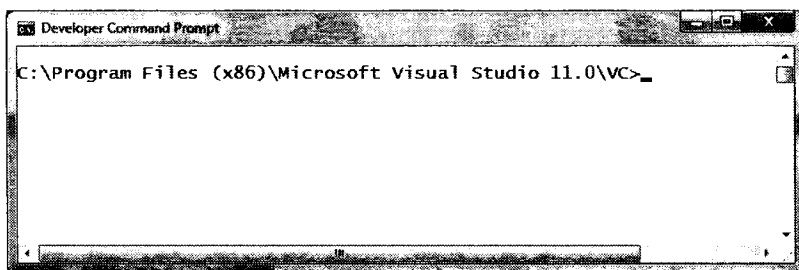


Рис. 2.1. Окно командной строки разработчика

Преимущество использования этого конкретного окна командной строки связано с тем, что оно заранее сконфигурировано для обеспечения доступа ко всем инструментам разработки под .NET. Предполагая, что на компьютере установлена среда разработки .NET, введите следующую команду и нажмите клавишу <Enter>:

```
csc -?
```

Если все в порядке, вы должны увидеть список аргументов командной строки компилятора С# командной строки. Как легко убедиться, этот компилятор командной строки предоставляет множество параметров; тем не менее, на практике для построения программ С# из командной строки требуется лишь небольшая их часть.

Построение приложений С# с использованием csc.exe

Хотя необходимость в построении крупных приложений с использованием одного лишь компилятора С# командной строки может никогда не возникнуть, очень важно понимать в целом, каким образом компилировать файлы кода вручную. Существует несколько причин, по которым вы должны освоить этот процесс.

- Самой очевидной причиной является отсутствие копии Visual Studio или другой графической IDE-среды.

- Работа выполняется в колледже или университете, где использование инструментов генерации кода и IDE-сред в учебном процессе запрещено.
- Планируется применение инструментов автоматической сборки, таких как `msbuild.exe`, которые требуют знания опций командной строки для используемых инструментов.
- Есть желание углубить свои познания C#. Когда для построения приложений применяется графическая IDE-среда, в конечном итоге все сводится к предоставлению компилятору `csc.exe` инструкций по манипулированию входными файлами кода C#. В свете этого полезно посмотреть, что происходит "за кулисами".

Еще одно преимущество подхода с использованием одного лишь компилятора `csc.exe` состоит в том, что он позволяет чувствовать себя более уверенно при работе с другими инструментами командной строки, входящими в состав .NET Framework 4.5 SDK. Как будет показано далее в книге, несколько важных утилит (вроде `gacutil.exe`, `ngen.exe`, `ildasm.exe` и `aspnet_regiis.exe`) доступны исключительно в командной строке.

Чтобы проиллюстрировать создание .NET-приложения без IDE-среды, мы построим простую исполняемую сборку по имени `TestApp.exe` с применением компилятора C# командной строки и редактора Notepad. Вначале необходимо подготовить исходный код. Откройте редактор Notepad (Блокнот), выбрав в меню `Start⇒All Programs⇒Accessories⇒Notepad` (Пуск⇒Все программы⇒Стандартные⇒Блокнот), и введите следующее типичное определение класса на языке C#:

```
// Простое приложение C#.
using System;

class TestApp
{
    static void Main()
    {
        Console.WriteLine("Testing! 1, 2, 3");
    }
}
```

По завершении ввода сохраните файл под именем `TestApp.cs` в удобном месте (например, в `C:\CscExample`). Теперь можно приступить к ознакомлению с основными опциями компилятора C#.

На заметку! Файлам кода C# назначается расширение `*.cs`. В отличие от Java, имя файла не обязано отражать определяемое в нем имя типа (или типов).

Указание целевых входных и выходных параметров

В первую очередь следует разобраться, как указывать имя и тип создаваемой сборки (например, консольное приложение `MyShell.exe`, библиотека `MathLib.dll`, приложение Windows Presentation Foundation по имени `Halo8.exe` и т.д.). Каждый компонент представляется специфическим флагом, передаваемым `csc.exe` в виде параметра командной строки (табл. 2.1).

На заметку! Параметры, передаваемые компилятору командной строки (а также большинству других утилит командной строки), могут сопровождаться префиксом в виде символа дефиса (-) или символа косой черты (/).

Таблица 2.1. Обычные выходные параметры, которые может принимать компилятор С#

Параметр	Описание
/out	Этот параметр применяется для указания имени создаваемой сборки. По умолчанию сборке назначается то же имя, что и у входного файла *.cs
/target:exe	Этот параметр позволяет построить исполняемое консольное приложение. Сборка такого типа генерируется по умолчанию, потому при создании подобного приложения данный параметр можно опускать
/target:library	Этот параметр позволяет создать однофайловую сборку *.dll
/target:winexe	Хотя приложения с графическим пользовательским интерфейсом можно создавать с применением параметра /target:exe, параметр /target:winexe позволяет предотвратить открытие окна консоли под остальными окнами

Чтобы скомпилировать TestApp.cs в консольное приложение по имени TextApp.exe, перейдите в каталог, в котором был сохранен файл исходного кода, с помощью команды cd:

```
cd C:\CscExample
```

Затем введите следующую команду (обратите внимание, что флаги должны располагаться перед именами входных файлов, а не после):

```
csc /target:exe TestApp.cs
```

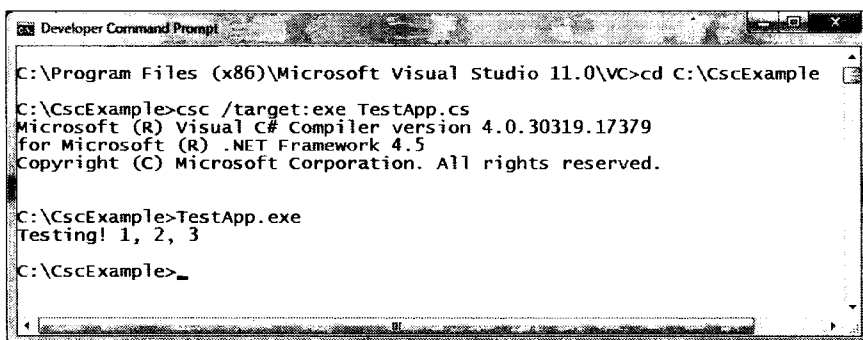
Здесь флаг /out не был явно указан, поэтому исполняемый файл получит имя TestApp.exe с учетом того, что именем входного файла является TestApp. Также помните, что почти все принимаемые компилятором С# флаги имеют сокращенные версии, наподобие /t вместо /target (полный список которых можно увидеть, введя в командной строке команду csc -?).

```
csc /t:exe TestApp.cs
```

Более того, поскольку флаг /t:exe используется компилятором как стандартный вывод, скомпилировать TestApp.cs также можно с помощью следующей простой команды:

```
csc TestApp.cs
```

Теперь можно запустить приложение TestApp.exe из командной строки, введя имя исполняемого файла, как показано на рис. 2.2.

**Рис. 2.2. Компиляция и запуск приложения TestApp.exe**

Ссылка на внешние сборки

Давайте посмотрим, как скомпилировать приложение, в котором используются типы, определенные в отдельной сборке .NET. Если вас интересует, каким образом компилятор C# воспринял ссылку на тип `System.Console`, вспомните из главы 1, что во время процесса компиляции происходит *автоматическое добавление ссылки* на `mscorlib.dll` (если по какой-то необычной причине необходимо отключить эту возможность, можно передать `csc.exe` параметр `/nostdlib`).

Модифицируем приложение `TestApp` так, чтобы в нем отображалось окно сообщения Windows Forms. Для этого откройте файл `TestApp.cs` и измените его следующим образом:

```
using System;

// Добавить следующую строку:
using System.Windows.Forms;

class TestApp
{
    static void Main()
    {
        Console.WriteLine("Testing! 1, 2, 3");

        // Добавить следующую строку:
        MessageBox.Show("Hello...");
    }
}
```

Обратите внимание на импорт пространства имен `System.Windows.Forms` с помощью поддерживаемого в C# ключевого слова `using` (о котором рассказывалось в главе 1). Вспомните, что явное перечисление пространств имен, которые используются внутри файла `*.cs`, позволяет избежать необходимости в указании полностью заданных имен для типов.

Далее в командной строке нужно проинформировать компилятор `csc.exe` о том, в какой сборке содержатся используемые пространства имен. Поскольку применяется класс `MessageBox` из пространства имен `System.Windows.Forms`, с помощью флага `/reference` (или его сокращенной версии `/r`) должна быть указана сборка `System.Windows.Forms.dll`:

```
csc /r:System.Windows.Forms.dll TestApp.cs
```

Запустив приложение снова, в дополнение к консольному выводу вы должны увидеть окно с сообщением, как показано на рис. 2.3.

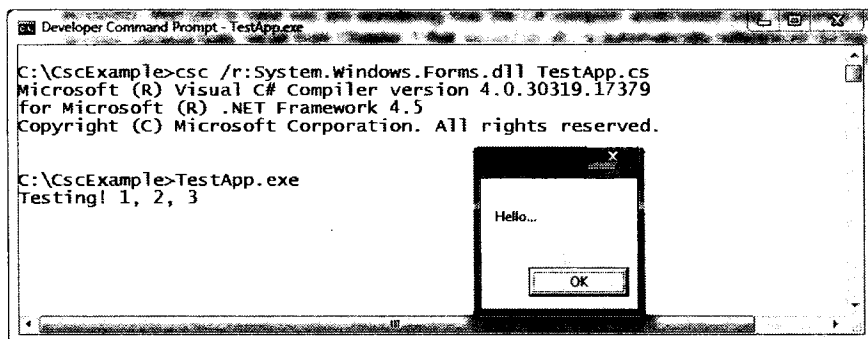


Рис. 2.3. Первое приложение с графическим пользовательским интерфейсом

Ссылка на несколько внешних сборок

Кстати, а что если нужно указать `csc.exe` несколько внешних сборок? Для этого просто перечислите все сборки через точку с запятой. В текущем примере ссылаться на множество сборок не требуется, но ниже приведена команда, в которой это сделано:

```
csc /r:System.Windows.Forms.dll;System.Drawing.dll *.cs
```

На заметку! Как будет показано позже в главе, компилятор C# будет автоматически ссылаться на набор основных сборок .NET (таких как `System.Windows.Forms.dll`), даже если они не указаны с помощью флага `/r`.

Компиляция нескольких файлов исходного кода

Текущая версия приложения `TestApp.exe` создавалась с использованием единственного файла исходного кода `*.cs`. Хотя определять все типы .NET в одном файле `*.cs` вполне допустимо, большинство проектов состоят из множества файлов `*.cs` для обеспечения большей гибкости кодовой базы. Предположим, что написан новый класс, содержащийся в отдельном файле по имени `HelloMsg.cs`:

```
// Класс HelloMessage.
using System;
using System.Windows.Forms;

class HelloMessage
{
    public void Speak()
    {
        MessageBox.Show("Hello...");
    }
}
```

Изменим начальный класс `TestApp` так, чтобы в нем применялся этот новый класс, и закомментируем предыдущую логику, связанную с Windows Forms:

```
using System;

// Эта строка больше не нужна:
// using System.Windows.Forms;

class TestApp
{
    static void Main()
    {
        Console.WriteLine("Testing! 1, 2, 3");
        // Эта строка тоже больше не нужна:
        // MessageBox.Show("Hello...");
        // Использовать класс HelloMessage!
        HelloMessage h = new HelloMessage();
        h.Speak();
    }
}
```

Скомпилировать файлы C# можно путем явного перечисления каждого входного файла:

```
csc /r:System.Windows.Forms.dll TestApp.cs HelloMsg.cs
```

В качестве альтернативы компилятор C# позволяет использовать групповой символ (*) для включения в текущую сборку всех файлов *.cs, которые содержатся в каталоге проекта:

```
csc /r:System.Windows.Forms.dll *.cs
```

Вывод, полученный в результате запуска программы, идентичен предыдущей ее версии. Единственная разница между этими двумя приложениями связана с разнесением логики по нескольким файлам.

Работа с ответными файлами в C#

Как не трудно догадаться, для построения сложного приложения C# из командной строки потребовалось бы вводить множество входных параметров для уведомления компилятора о том, как он должен обрабатывать исходный код. Для уменьшения нагрузки по вводу компилятор C# поддерживает использование *ответных файлов*.

В ответных файлах C# размещаются все инструкции, которые должны применяться во время компиляции текущей сборки. По соглашению эти файлы имеют расширение *.rsp. Чтобы посмотреть на них в действии, создадим ответный файл по имени TestApp.rsp, который содержит следующие параметры (комментарии обозначаются символом #):

```
# Это ответный файл
# для примера TestApp.exe
# из главы 2.

# Ссылки на внешние сборки:
/r:System.Windows.Forms.dll

# Параметры вывода и файлы для компиляции (используя групповой символ).
/target:exe /out:TestApp.exe *.cs
```

Теперь, предполагая, что этот файл сохранен в том же каталоге, где находятся подлежащие компиляции файлы исходного кода C#, приложение можно создать следующим образом (обратите внимание на использование символа @):

```
csc @TestApp.rsp
```

При необходимости можно также указывать несколько ответных файлов *.rsp в качестве входных (например, csc @FirstFile.rsp @SecondFile.rsp @ThirdFile.rsp). Однако при таком подходе имейте в виду, что компилятор обрабатывает параметры команд по мере их поступления. Таким образом, аргументы командной строки в более позднем ответном файле могут перезаписать параметры из предшествующего ответного файла.

Также обратите внимание, что все флаги, явно перечисленные перед ответным файлом, будут перезаписаны параметрами, указанными в этом файле. То есть в случае ввода следующей команды:

```
csc /out:MyCoolApp.exe @TestApp.rsp
```

именем сборки по-прежнему останется TestApp.exe (а не MyCoolApp.exe) из-за наличия флага /out:TestApp.exe в ответном файле TestApp.rsp. Однако если флаги перечислены после ответного файла, они перезапишут параметры, содержащиеся в этом файле.

На заметку! Флаг /reference является кумулятивным. Где бы ни указывались внешние сборки (до, после или внутри ответного файла), конечным результатом будет совокупность всех сборок, на которые производилась ссылка.

Стандартный ответный файл (*csc.rsp*)

Последним моментом, касающимся ответных файлов, о котором необходимо упомянуть, является то, что с компилятором C# ассоциирован стандартный ответный файл (*csc.rsp*), который размещен в том же самом каталоге, что и *csc.exe* (обычно в *C:\Windows\Microsoft.NET\Framework\<версия>*, где *<версия>* — это номер версии платформы). Если открыть файл *csc.rsp* в программе Notepad (Блокнот), обнаружится, что с помощью флага */r:* в нем уже указаны многие сборки .NET, в том числе различные библиотеки для разработки веб-приложений, программирования с использованием LINQ, доступа к данным и другие основные библиотеки (помимо *mscorlib.dll*).

При построении программ C# с помощью *csc.exe* ссылка на этот ответный файл добавляется автоматически, даже если указан специальный файл **.rsp*. Учитывая наличие стандартного ответного файла, текущее приложение *TestApp.exe* может быть успешно скомпилировано с применением следующей команды (т.к. *csc.rsp* содержит ссылку на *System.Windows.Forms.dll*):

```
csc /out:TestApp.exe *.cs
```

Чтобы запретить автоматическое чтение *csc.rsp*, укажите опцию */noconfig*:

```
csc @TestApp.rsp /noconfig
```

На заметку! В случае ссылки (с помощью опции */r*) на сборки, которые в действительности не используются, компилятор их проигнорирует, поэтому беспокоиться по поводу “разбухания кода” не нужно. Если производится ссылка на библиотеку, которая на самом деле не применяется, компилятор также ее проигнорирует.

Очевидно, что компилятор командной строки C# имеет множество других параметров, которые могут использоваться для управления генерацией результирующей сборки .NET. Другие важные возможности будут демонстрироваться по мере необходимости далее в книге, а полные сведения о них можно найти в документации .NET Framework 4.5 SDK.

Исходный код. Код приложения *CscExample* доступен в подкаталоге *Chapter 02*.

Построение приложений .NET с использованием Notepad++

Еще одним простым текстовым редактором, о котором следует кратко упомянуть, является свободно загружаемое приложение Notepad++. Этот инструмент можно получить по адресу <http://notepad-plus.sourceforge.net/>. В отличие от примитивного редактора Notepad (Блокнот), входящего в состав Windows, приложение Notepad++ позволяет создавать код на множестве языков и поддерживает разнообразные подключаемые утилиты. Вдобавок Notepad++ обладает рядом других замечательных достоинств, в числе которых:

- готовая поддержка ключевых слов C# (включая выделение цветом);
- поддержка *свертывания синтаксиса*, позволяющая сворачивать и разворачивать группы операторов внутри редактора (похожая на доступную в Visual Studio / C# Express);
- возможность увеличивать и уменьшать масштаб отображения текста с помощью колесика мыши при нажатой клавише *<Ctrl>*;
- настраиваемая функция автозавершения для различных ключевых слов C# и названий пространств имен .NET.

Функция автозавершения кода C# (рис. 2.4) включается нажатием комбинации клавиш <Ctrl+пробел>.

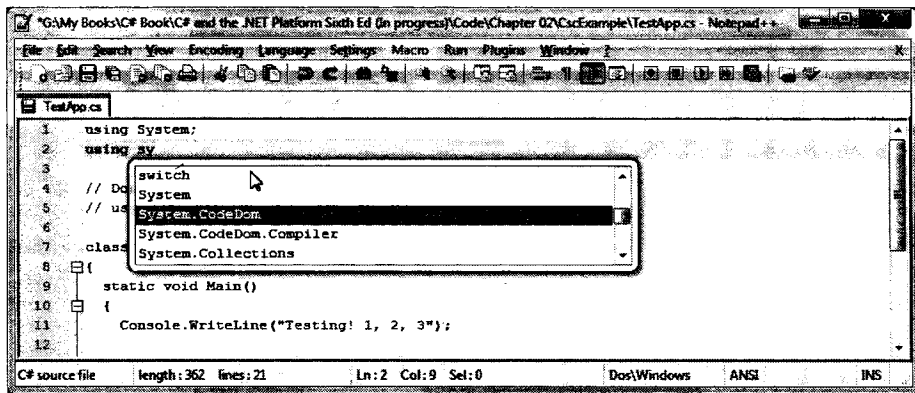


Рис. 2.4. Использование автозавершения кода в Notepad++

На заметку! Список вариантов, отображаемых в окне автозавершения, можно изменять и расширять. Для этого необходимо открыть файл `C:\Program Files\Notepad++\plugins\APIs\cs.xml` для редактирования и добавить в него любые дополнительные записи.

На этом краткий обзор приложения Notepad++ завершен. Для получения дополнительной информации воспользуйтесь встроенной справочной системой Notepad++.

Построение приложений .NET с помощью SharpDevelop

Вы наверняка согласитесь с тем, что написание кода C# с применением Notepad++ является шагом вперед по сравнению с использованием Notepad. Тем не менее, в Notepad++ отсутствуют развитые возможности IntelliSense, визуальные конструкторы для построения графических пользовательских интерфейсов, шаблоны проектов, либо утилиты для работы с базами данных. Для удовлетворения перечисленных потребностей предназначен еще один продукт для разработки .NET-приложений — SharpDevelop (также называемый *#Develop*).

Продукт SharpDevelop представляет собой многофункциональную IDE-среду с открытым кодом, которую можно применять для построения сборок .NET на языках C#, VB, IronRuby, IronPython, C++/CLI, F#, а также Boo. Помимо того, что эта IDE-среда совершенно бесплатна, интересно отметить, что она написана полностью на C#. Для установки SharpDevelop можно загрузить и скомпилировать ее файлы .cs* вручную или запустить программу `setup.exe`. Оба дистрибутива доступны на веб-сайте <http://www.sharpdevelop.com/>.

SharpDevelop предлагает множество путей улучшения продуктивности разработки. Ниже приведен список наиболее важных достоинств.

- Поддержка множества языков, версий и типов проектов .NET.
- Средство IntelliSense, автозавершение кода и фрагменты кода.
- Диалоговое окно Add Reference (Добавление ссылки) для ссылки на внешние сборки, включая сборки, развернутые в глобальном кеше сборки (GAC).

- Встроенные визуальные конструкторы графических пользовательских интерфейсов для настольных и веб-приложений.
- Встроенные утилиты для просмотра объектов и определения кода.
- Утилиты для визуального проектирования баз данных.
- Утилита для преобразования кода C# в код VB (и наоборот).

Выглядит довольно впечатляюще для бесплатной IDE-среды, не так ли? Ниже кратко рассмотрены некоторые наиболее интересные возможности.

На заметку! Текущая версия SharpDevelop (4.2) поддерживает все возможности C# и .NET 4.5. Периодически проверяйте веб-сайт SharpDevelop на предмет появления новых выпусков.

Создание простого тестового проекта

После установки SharpDevelop пункт меню File⇒New⇒Solution (Файл⇒Создать⇒Решение) позволяет выбирать тип проекта, который нужно генерировать (и язык .NET). Например, предположим, что создается решение C# Windows Application (Windows-приложение на C#) по имени MySDWinApp (рис. 2.5).

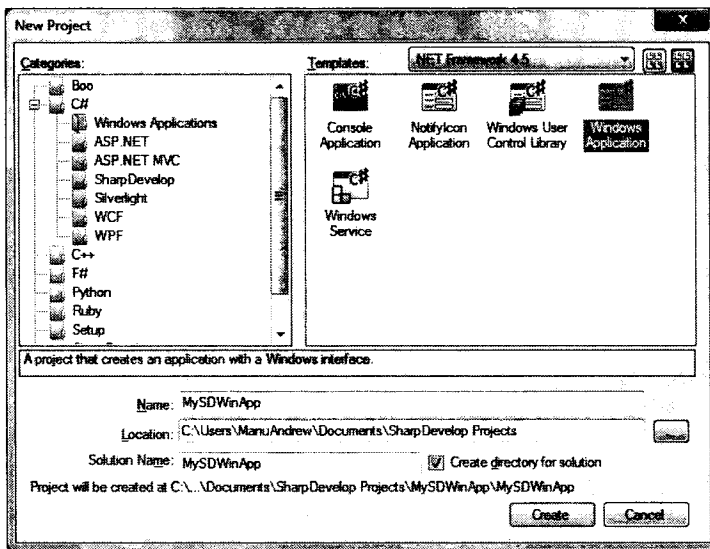


Рис. 2.5. Диалоговое окно создания нового проекта в SharpDevelop

Как и в Visual Studio, доступна панель элементов управления для визуального конструктора графических пользовательских интерфейсов Windows Forms, позволяющая перетаскивать элементы управления на поверхность конструктора, и окно Properties (Свойства), которое дает возможность настраивать внешний вид и поведение каждого элемента графического пользовательского интерфейса. На рис. 2.6 показан пример настройки кнопки; обратите внимание, что для этого был выполнен щелчок на вкладке Design (Конструктор), которая находится под открытым файлом кода.

После щелчка на вкладке Source (Исходный код) в нижней части окна конструктора форм, станут доступными средства IntelliSense, автозавершения кода и встроенной справки (рис. 2.7).

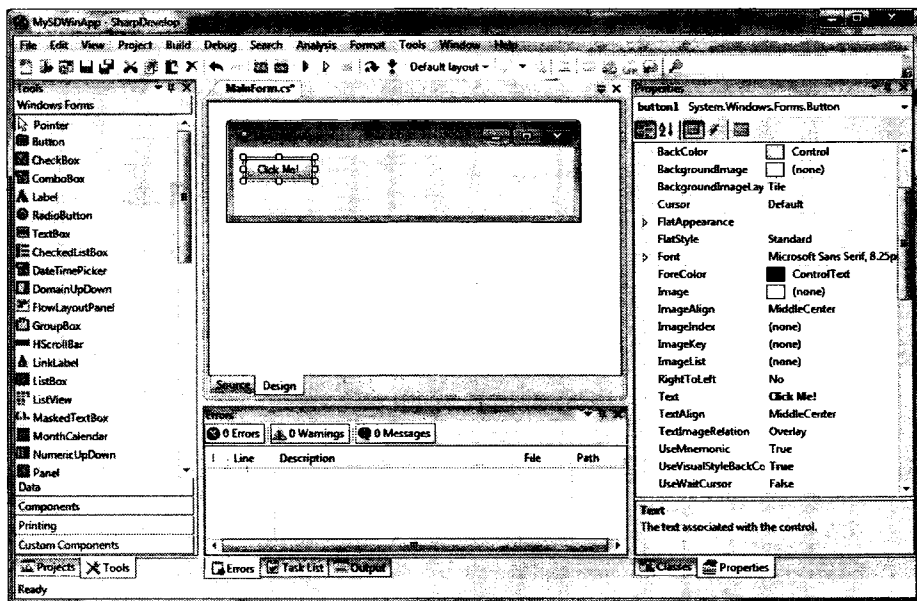


Рис. 2.6. Графическое конструирование приложения Windows Forms в SharpDevelop

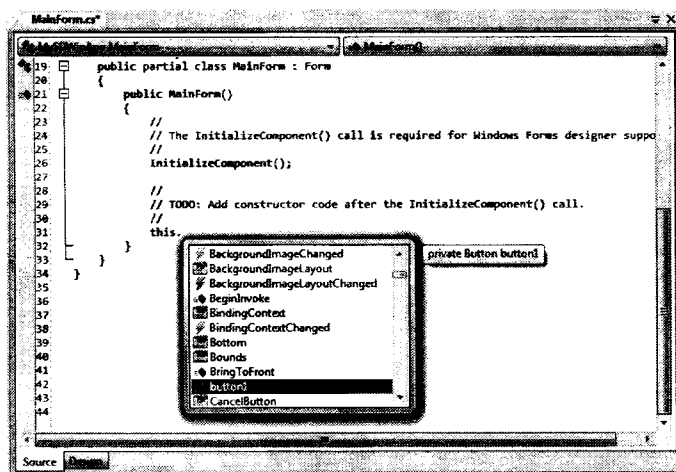


Рис. 2.7. В SharpDevelop поддерживается множество утилит генерации кода

Продукт SharpDevelop спроектирован для воспроизведения большинства функциональности, предлагаемой IDE-средами .NET производства Microsoft (которые рассматриваются следующими). Учитывая это, более подробно возможности SharpDevelop рассматриваться не будут. Для получения дополнительной информации воспользуйтесь меню Help (Справка).

На заметку! MonoDevelop — это IDE-среда с открытым кодом, основанная на кодовой базе SharpDevelop. Она является предпочтительной IDE-средой для программистов .NET, которые строят приложения, ориентированные на операционные системы Mac OS X или Linux, с применением платформы Mono. Более подробно эта IDE-среда описана на сайте <http://monodevelop.com/>.

Построение приложений .NET с помощью Visual C# Express

Компания Microsoft предоставляет линейку IDE-сред под общим названием *Express* (<http://msdn.microsoft.com/express>). К настоящему времени на рынке доступно несколько членов этого семейства (все они распространяются бесплатно и поддерживаются и обслуживаются компанией Microsoft).

- *Visual Web Developer Express*. Легковесный инструмент для построения динамических веб-сайтов ASP.NET и служб WCF.
- *Visual Basic Express*. Упрощенный инструмент для программирования, идеально подходящий новичкам, которые хотят научиться создавать приложения с применением дружественного к пользователям синтаксиса Visual Basic.
- *Visual C# Express* и *Visual C++ Express*. IDE-среды, ориентированные специально на студентов и энтузиастов, желающих обучиться основам программирования с использованием предпочитаемого синтаксиса.
- *SQL Server Express*. Система управления базами данных начального уровня, предназначенная для любителей, энтузиастов и обучающихся разработчиков.

Некоторые уникальные возможности Visual C# Express

В целом продукты Express представляют собой усеченные версии своих полнофункциональных аналогов в линейке Visual Studio и ориентированы главным образом на любителей .NET и студентов. Подобно SharpDevelop, в Visual C# Express предлагаются разнообразные инструменты для просмотра объектов, визуальные конструкторы графических пользовательских интерфейсов для настольных приложений, диалоговое окно Add Reference (Добавление ссылки), возможности IntelliSense и шаблоны расширения кода.

Тем не менее, в Visual C# Express доступно несколько (важных) средств, в настоящее время отсутствующих в SharpDevelop. К их числу относятся:

- развитая поддержка создания приложений Window Presentation Foundation (WPF) с помощью XAML;
- возможность загрузки дополнительных бесплатных шаблонов, которые позволяют разрабатывать приложения для Xbox 360, приложения WPF с интеграцией Twitter и многие другие.

Поскольку по внешнему виду и поведению IDE-среда Visual C# Express очень похожа на Visual Studio (и в некоторой степени на SharpDevelop), более подробно она здесь не рассматривается. Ее можно использовать для проработки материала этой книги, но следует иметь в виду, что Visual C# Express не поддерживает шаблоны проектов для построения веб-сайтов ASP.NET. Чтобы строить также и веб-приложения, понадобится загрузить продукт Visual Web Developer, который также доступен по адресу <http://msdn.microsoft.com/express>.

Построение приложений .NET с помощью Visual Studio

Профессиональные разработчики программного обеспечения .NET, как правило, рассматривают наиболее серьезным в этой сфере продуктом производства Microsoft, который называется Visual Studio и доступен по адресу <http://msdn.microsoft.com/vstudio>.

Этот инструмент представляет собой самую функционально насыщенную и наиболее приспособленную под использование на предприятиях IDE-среду из всех рассмотренных в настоящей главе. Разумеется, за эту мощь приходится платить, и цена варьируется в зависимости от версии Visual Studio. Как не трудно догадаться, каждая версия поставляется с уникальным набором функциональных возможностей.

На заметку! Семейство продуктов Visual Studio включает в себя множество членов. В оставшейся части книги предполагается, что для использования выбрана версия Visual Studio Professional.

Хотя далее предполагается наличие копии Visual Studio Professional, она вовсе не обязательна для проработки материала настоящей книги. В худшем случае окажется, что в используемой вами IDE-среде (такой как Microsoft C# Express или SharpDevelop) та или иная опция не доступна. Тем не менее, весь приведенный в книге код будет нормально компилироваться, какой бы инструмент не применялся для его создания.

На заметку! После загрузки исходного кода для этой книги можно открыть необходимый пример в Visual Studio (или Visual C# Express), дважды щелкнув на файле *.sln примера. Если на машине нет установленной копии Visual Studio или Visual C# Express, потребуется вручную вставить файлы *.cs примера в рабочую область проекта внутри используемой IDE-среды.

Некоторые уникальные возможности Visual Studio

Продукт Visual Studio поставляется с конструкторами графических пользовательских интерфейсов, поддержкой фрагментов кода, инструментами манипулирования базами данных, утилитами для просмотра объектов и проектов, а также встроенной справочной системой. В отличие от многих уже рассмотренных IDE-сред, Visual Studio поддерживает множество дополнительных возможностей, наиболее важные из которых перечислены ниже:

- визуальные редакторы и конструкторы XML;
- поддержка разработки приложений для мобильных устройств Windows;
- поддержка разработки приложений для Microsoft Office;
- поддержка визуального конструктора для проектов Windows Workflow Foundation;
- встроенная поддержка рефакторинга кода;
- инструменты визуального конструирования классов.

По правде говоря, Visual Studio предлагает настолько много возможностей, что для их полного описания понадобилась бы отдельная книга. В настоящей книге такие цели не преследуются. Тем не менее, в нескольких следующих разделах наиболее важные функциональные возможности рассматриваются немного подробнее, а другие по мере необходимости будут описаны далее в книге.

На заметку! Если вы пользовались предыдущей версией Microsoft Visual Studio, то быстро заметите, что внешний вид и поведение IDE-среды несколько изменились. Если стандартная тема "Dark" ("Темная") покажется слишком темной, ее можно изменить на светлую ("Light"), выбрав пункт меню Tools⇒Options (Сервис⇒Параметры) и указав нужную тему в раскрываемом списке Color Theme (Цветовая тема) внутри раздела Environment⇒General (Среда⇒Основные параметры). При получении всех снимков экрана, представленных в этой книге, применялась тема "Light".

Выбор целевой версии .NET Framework в диалоговом окне New Project

Можете сейчас создать новое консольное приложение на C# (по имени VsExample), выбрав пункт меню **File**⇒**New**⇒**Project** (Файл⇒Создать⇒Проект). Как показано на рис. 2.8, Visual Studio поддерживает возможность выбора версии .NET Framework (2.0, 3.0, 3.5, 4.0 или 4.5), для которой должно создаваться приложение, с помощью раскрывающегося списка, отображаемого по центру в верхней части диалогового окна **New Project** (Новый проект). Для всех описываемых в настоящей книге проектов в этом списке можно оставлять выбранным предлагаемый по умолчанию вариант .NET Framework 4.5.

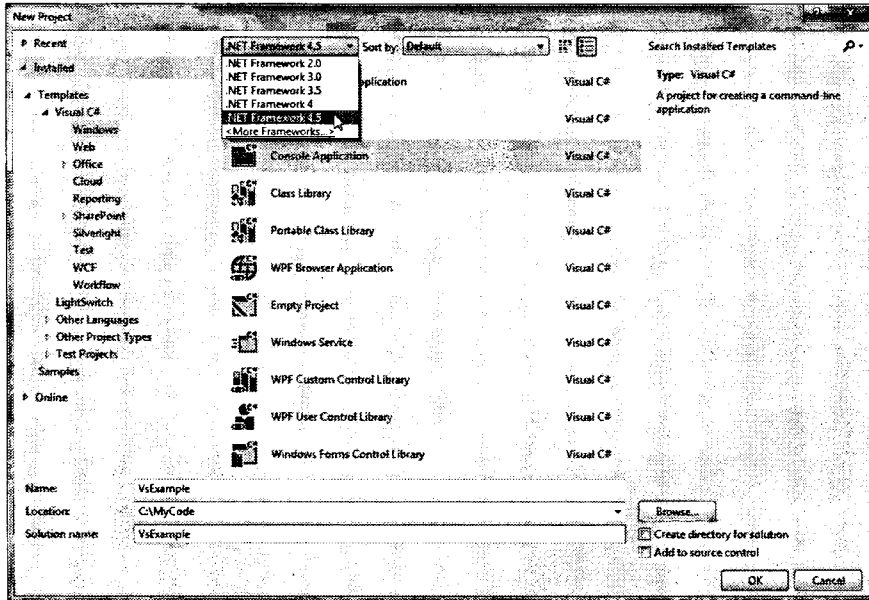


Рис. 2.8. Visual Studio позволяет выбирать целевую версию .NET Framework

Использование утилиты Solution Explorer

Утилита **Solution Explorer** (Проводник решений), доступная через меню **View** (Вид), позволяет просматривать набор всех файлов содержимого и ссылочных сборок, которые входят в состав текущего проекта (рис. 2.9). Также обратите внимание, что заданный файл (например, **Program.cs**) можно раскрыть, чтобы просмотреть определенные в нем кодовые типы. По ходу изложения материала этой книги по мере необходимости будут указываться и другие полезные особенности **Solution Explorer**. Однако при желании можете самостоятельно поэкспериментировать с каждой из опций.

Кроме того, внутри папки **References** (Ссылки) в окне **Solution Explorer** отображается список всех сборок, на которые имеются ссылки. В зависимости от типа выбираемого проекта и целевой версии .NET Framework, этот список выгля-

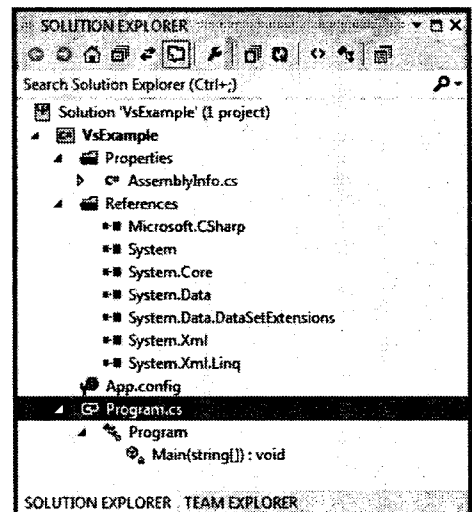


Рис. 2.9. Окно утилиты Solution Explorer

дит по-разному. Поскольку мы создали консольное приложение, набор автоматически включаемых библиотек минимален (`System.dll`, `System.Core.dll`, `System.Data.dll` и т.д.).

На заметку! Вспомните из главы 1, что основная библиотека .NET носит имя `mscorlib.dll`. Эта библиотека не будет отображена в окне Solution Explorer. Однако, несмотря на это, все содержащиеся в ней типы доступны.

Ссылка на внешние сборки

Если необходимо сослаться на дополнительные сборки, щелкните правой кнопкой мыши на папке References и выберите в контекстном меню пункт Add Reference (Добавить ссылку). После этого откроется диалоговое окно Reference Manager (Диспетчер ссылок), позволяющее выбрать желаемые сборки (в Visual Studio это аналог параметра /reference для компилятора командной строки). На вкладке Framework (Платформа) этого окна (рис. 2.10) отображается список наиболее часто используемых сборок .NET; на вкладке Browse (Обзор) предоставляется возможность найти сборки .NET, которые находятся на жестком диске. На вкладке Recent (Недавние) отслеживаются ссылки на сборки, которые применялись в других проектах.

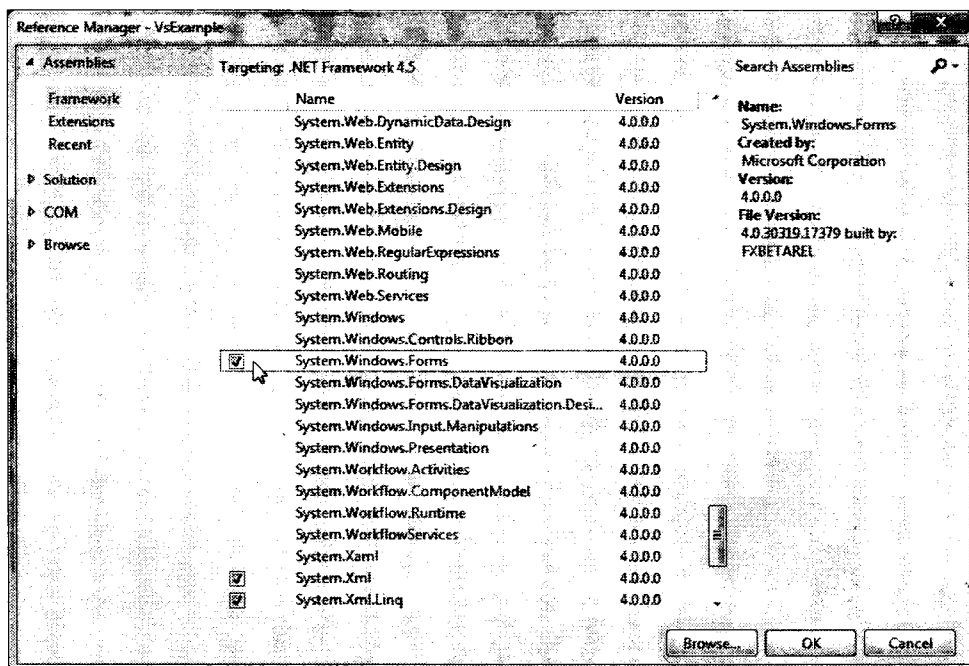


Рис. 2.10. Диалоговое окно Reference Manager

Чтобы выполнить простейший тест, найдите сборку `System.Windows.Forms.dll` в разделе Framework и отметьте связанный с ней флажок. После этого действия (и закрытия диалогового окна) данная библиотека появится в папке References в окне Solution Explorer. Эту ссылку можно удалить, выбрав ее в окне Solution Explorer и нажав клавишу <Delete> (или с помощью команды Delete (Удалить) контекстного меню, открываемого по щелчку правой кнопкой мыши).

Просмотр свойств проекта

И, наконец, обратите внимание на значок Properties (Свойства) в окне Solution Explorer. Двойной щелчок на нем приводит к открытию редактора конфигурации проекта (рис. 2.11).

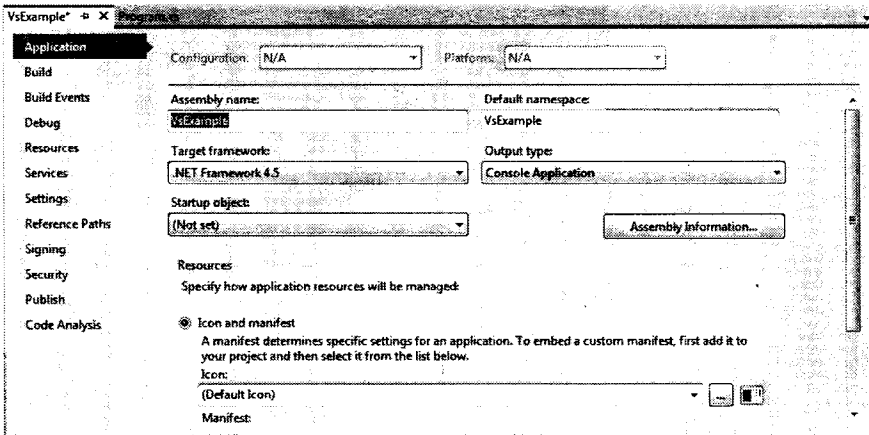


Рис. 2.11. Редактор конфигурации проекта

Различные аспекты редактора конфигурации проекта будут обсуждаться далее в книге. Здесь можно устанавливать различные параметры безопасности, назначать сборке надежное имя (глава 14), разворачивать приложение, вставлять ресурсы приложения и конфигурировать события, происходящие до и после компиляции сборки.

Утилита Class View

Следующим инструментом, с которым необходимо ознакомиться, является утилита Class View (Просмотр классов), доступ к которой также производится через меню View. Эта утилита предназначена для просмотра всех типов в текущем проекте с объектно-ориентированной точки зрения (а не с точки зрения файлов, как в Solution Explorer). В верхней панели утилиты Class View отображается список пространств имен и их типов, а в нижней панели — члены выбранного в текущий момент типа (рис. 2.12).

Двойной щелчок на типе или члене типа приводит к автоматическому открытию соответствующего файла кода C# и помещению курсора в нужную позицию. Еще одной замечательной особенностью утилиты Class View в Visual Studio является возможность открытия любой ссылочной сборки и просмотра содержащихся внутри нее пространств имен, типов и членов.

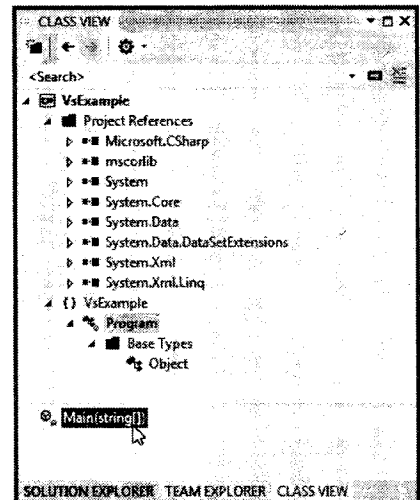


Рис. 2.12. Утилита Class View

Утилита Object Browser

В Visual Studio доступна еще одна утилита для исследования набора сборок, на которые имеются ссылки в текущем проекте. Называется эта утилита Object Browser

(Браузер объектов) и получить к ней доступ можно через меню View. После ее открытия нужно лишь выбрать сборку для исследования (рис. 2.13).

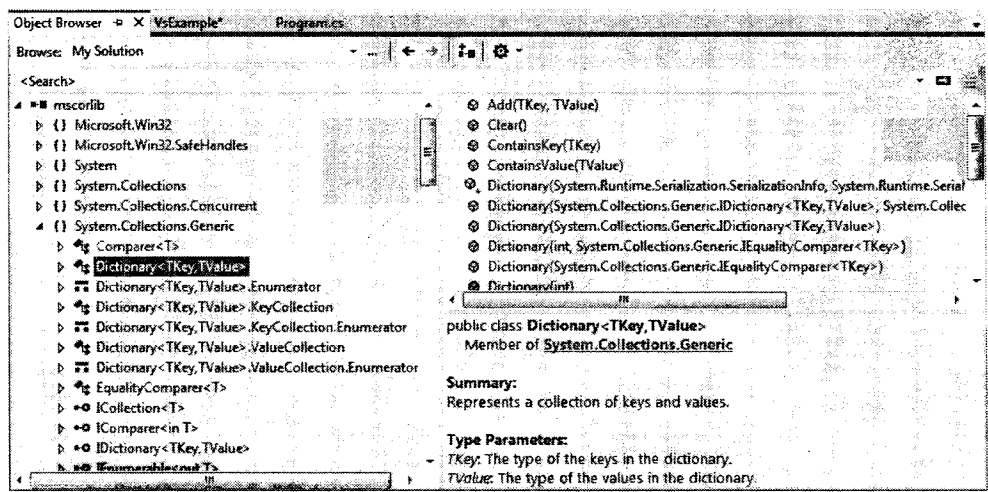


Рис. 2.13. Утилита Object Browser

Встроенная поддержка рефакторинга кода

Одним из главных функциональных средств Visual Studio является встроенная поддержка *рефакторинга* существующего кода. Если объяснять упрощенно, то рефакторинг — это формальный механический процесс улучшения существующей кодовой базы. В прежние времена рефакторинг требовал приложения массы ручных усилий. К счастью, Visual Studio позволяет значительно автоматизировать процесс рефакторинга, предлагая несколько наиболее распространенных технологий.

С помощью меню Refactor (Рефакторинг), доступного при открытом для редактирования файле кода в IDE-среде, соответствующих клавиатурных сокращений, смарт-тегов и/или чувствительных к контексту щелчков кнопкой мыши можно значительно видоизменять код, прикладывая минимальные усилия. В табл. 2.2 перечислены некоторые часто используемые приемы рефакторинга, распознаваемые Visual Studio.

Таблица 2.2. Рефакторинг в Visual Studio

Прием рефакторинга	Описание
Extract Method (Извлечение метода)	Позволяет определить новый метод на основе выбранных операторов кода
Encapsulate Field (Инкапсуляция поля)	Превращает открытое поле в закрытое поле, инкапсулированное в свойстве C#
Extract Interface (Извлечение интерфейса)	Определяет новый интерфейсный тип на основе набора существующих членов типа
Reorder Parameters (Переупорядочение параметров)	Позволяет изменять порядок следования аргументов в члене
Remove Parameters (Удаление параметров)	Удаляет заданный аргумент из текущего списка параметров
Rename (Переименование)	Позволяет переименовать кодовую конструкцию (имя метода, поле, локальную переменную и т.д.) по всему проекту

Чтобы увидеть процесс рефакторинга в действии, модифицируем метод `Main()` следующим образом:

```
static void Main(string[] args)
{
    // Настроить консольный интерфейс (CUI).
    Console.Title = "My Rocking App";
    Console.ForegroundColor = ConsoleColor.Yellow;
    Console.BackgroundColor = ConsoleColor.Blue;
    Console.WriteLine("*****");
    Console.WriteLine("***** Welcome to My Rocking App *****");
    Console.WriteLine("*****");
    Console.BackgroundColor = ConsoleColor.Black;

    // Ожидать нажатия клавиши <Enter>.
    Console.ReadLine();
}
```

Хотя в приведенном выше коде нет ничего неправильного, представим, что данное приветственное сообщение должно отображаться в различных местах по всей программе. Вместо того чтобы заново вводить ту же самую отвечающую за консольный интерфейс логику, было бы неплохо иметь вспомогательную функцию, которую можно было бы вызывать для этого. Итак, попробуем применить к существующему коду прием рефакторинга **Extract Method** (Извлечение метода).

Выделите в редакторе кода все операторы внутри `Main()` кроме последнего вызова `Console.ReadLine()` и щелкните на выделенном коде правой кнопкой мыши. Выберите в открывшемся контекстном меню пункт **Refactor** ⇒ **Extract Method** (Рефакторинг ⇒ Извлечение метода), как показано на рис. 2.14.

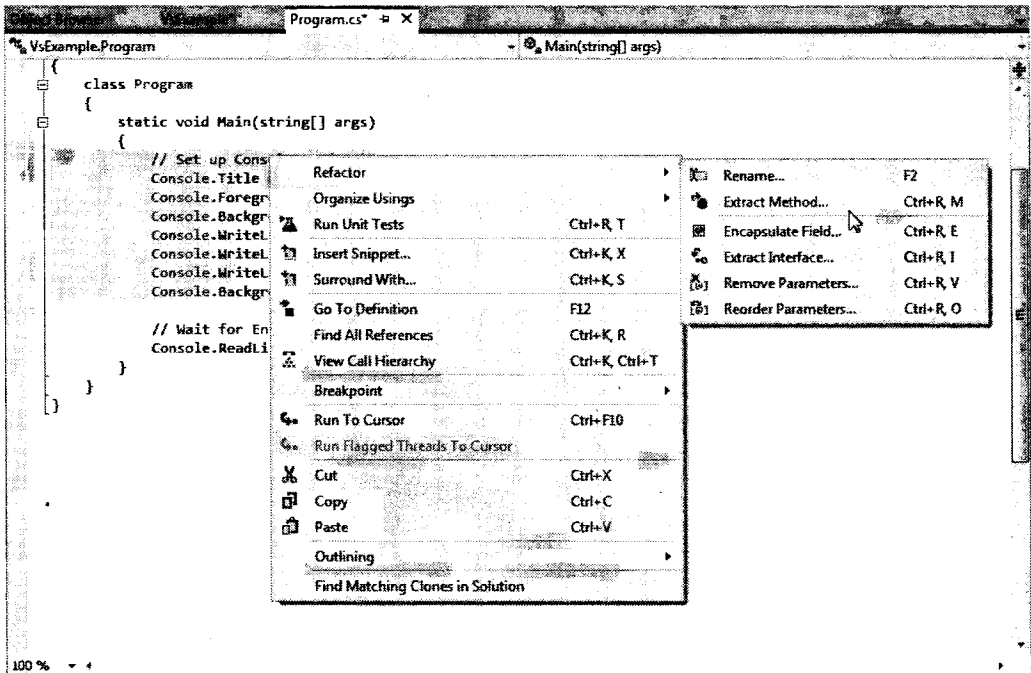


Рис. 2.14. Активизация рефакторинга кода

В открывшемся диалоговом окне назначьте новому методу имя `ConfigureCUI`. После этого метод `Main()` будет вызывать новый сгенерированный метод `ConfigureCUI()`, который содержит выделенный ранее код:

```
class Program
{
    static void Main(string[] args)
    {
        ConfigureCUI();

        // Ожидать нажатия клавиши <Enter>.
        Console.ReadLine();
    }

    private static void ConfigureCUI()
    {
        // Настроить консольный интерфейс (CUI).
        Console.Title = "My Rocking App";
        Console.ForegroundColor = ConsoleColor.Yellow;
        Console.BackgroundColor = ConsoleColor.Blue;
        Console.WriteLine("*****");
        Console.WriteLine("***** Welcome to My Rocking App *****");
        Console.WriteLine("*****");
        Console.BackgroundColor = ConsoleColor.Black;
    }
}
```

Этот лишь один простой пример использования рефакторинга в Visual Studio, и в ходе изучения настоящей книги вы встретитесь с другими примерами. Тем не менее, сейчас вы можете смело активизировать остальные опции рефакторинга, чтобы исследовать их влияние (мы не будем использовать текущий проект `VsExample` далее в книге, поэтому можете делать с ним все, что хотите).

Фрагменты кода и технология Surround With

В Visual Studio (а также в Visual C# Express) можно вставлять готовые блоки кода C# путем выбора соответствующих пунктов в меню, контекстно-чувствительных щелчков кнопкой мыши и/или использования клавиатурных сокращений. Количество доступных расширений кода впечатляет и в целом их можно поделить на две основных группы.

- *Фрагменты кода.* Эти шаблоны позволяют вставлять общие блоки кода в месте расположения курсора мыши.
- *Окружение (Surround With).* Эти шаблоны позволяют помещать блок выбранных операторов в рамки соответствующего контекста.

Чтобы посмотреть на данную функциональность в действии, предположим, что требуется последовательно просмотреть входные параметры метода `Main()` в цикле `foreach`. Вместо того чтобы вводить необходимый код вручную, можно активизировать фрагмент кода `foreach`. После выполнения этого действия IDE-среда поместит шаблон кода `foreach` в текущую позицию курсора.

В целях иллюстрации поместим курсор мыши после начальной открывающей фигурной скобки в методе `Main()`. Одним из способов активизации фрагмента кода является щелчок правой кнопкой мыши и выбор в контекстном меню пункта `Insert Snippet` (Вставить фрагмент) или `Surround With` (Окружить). Это приводит к отображению списка всех фрагментов кода данной категории (для закрытия контекстного меню необходимо нажать клавишу `<Esc>`). В качестве клавиатурного сокращения понадобится ввести имя нужного фрагмента кода, которым в данном случае является `foreach`. На рис. 2.15 видно, что значок, представляющий фрагмент кода, внешне напоминает клочок бумаги.

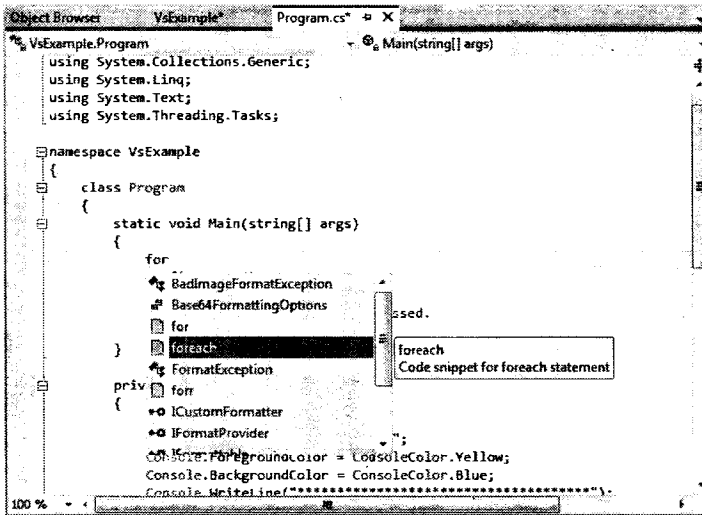


Рис. 2.15. Активизация фрагмента кода

Отыскав фрагмент кода, который требуется активизировать, два раза нажмите клавишу <Tab>. Это приведет к автоматическому завершению всего фрагмента кода и вставке набора заполнителей, которые можно заполнить для получения готового фрагмента. Нажимая клавишу <Tab>, можно циклически проходить по заполнителям и заполнять нужные пробелы (для выхода из режима редактирования фрагмента кода следует нажать клавишу <Esc>). В рассматриваемом примере первый заполнитель (ключевое слово var) можно заменить типом данных string, второй (имя переменной item) — именем arg, а последний — именем входного параметра типа string[].

В результате получается следующий код:

```
static void Main(string[] args)
{
    foreach (string arg in args)
    {
        ...
    }
}
```

Щелчок правой кнопкой мыши в открытом файле кода C# и выбор из контекстного меню пункта Surround With (Окружить) также приводит к открытию списка возможных вариантов. При использовании фрагментов Surround With обычно сначала выбирается блок операторов кода для представления того, что должно применяться для их окружения (например, блок try/catch). Обязательно уделите время изучению предопределенных шаблонов расширения кода, поскольку они могут радикально ускорить процесс разработки.

На заметку! Все шаблоны для расширения кода являются XML-описаниями кода для генерации в IDE-среде. В Visual Studio (а также в Visual C# Express) можно создавать собственные шаблоны кода. Дополнительные сведения доступны в статье "Investigating Code Snippet Technology" ("Исследование технологии применения фрагментов кода") по адресу <http://msdn.microsoft.com/en-US/library/ms379562>.

Утилита Class Designer

В Visual Studio имеется возможность конструировать классы и другие типы (такие как интерфейсы или делегаты) визуальным образом (в Visual C# Express такая функциональность отсутствует). Утилита Class Designer (Конструктор классов) позволяет просматривать и модифицировать отношения между типами (классами, интерфейсами, структурами, перечислениями и делегатами) в проекте. С помощью этой утилиты можно визуальным образом добавлять или удалять члены из типа с отражением этих изменений в соответствующем файле кода C#. Кроме того, изменения, вносимые в файл кода C#, отражаются в диаграмме классов.

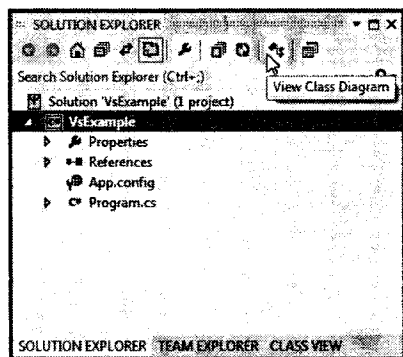


Рис. 2.16. Вставка файла диаграммы классов в текущий проект

Для работы с этим аспектом Visual Studio сначала необходимо вставить новый файл диаграммы классов. Делать это можно несколькими способами, один из которых предусматривает щелчок на кнопке View Class Diagram (Просмотр диаграммы классов) в правой части окна Solution Explorer, как показано на рис. 2.16 (удостоверьтесь, что в окне выбран проект, а не решение).

После выполнения этого действия появляются значки, которые представляют типы, найденные в текущем проекте. Щелкая на значке с изображением стрелки для заданного типа, можно отображать или скрывать его члены (рис. 2.17).

На заметку! С помощью панели инструментов утилиты Class Designer можно настраивать параметры отображения поверхности конструктора.

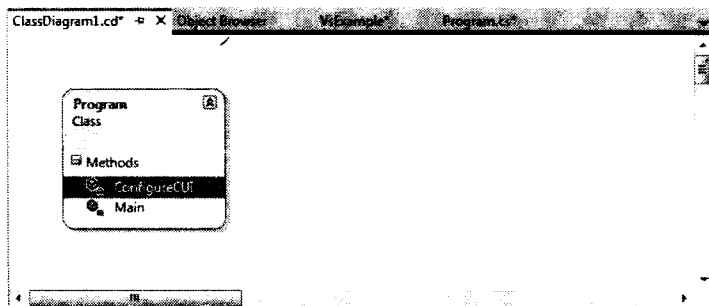


Рис. 2.17. Просмотр диаграммы классов

Утилита Class Designer работает в сочетании с двумя другими средствами Visual Studio — окном Class Details (Детали класса), которое открывается через меню View⇒Other Windows (Вид⇒Другие окна), и панелью инструментов Class Designer, отображаемой выбором пункта меню View⇒Toolbox (Вид⇒Панель инструментов). Окно Class Details не только отображает подробные сведения о текущем выбранном элементе диаграммы, но также позволяет изменять существующие члены и вставлять новые на лету (рис. 2.18).

Панель инструментов Class Designer, которая также может быть активизирована через меню View, позволяет вставлять в проект новые типы (и создавать между ними отношения) визуальным образом (рис. 2.19). (Чтобы видеть эту панель инструментов, окно диаграммы классов должно быть активным.) По мере выполнения этих действий IDE-среда автоматически создает новые определения типов на C#.

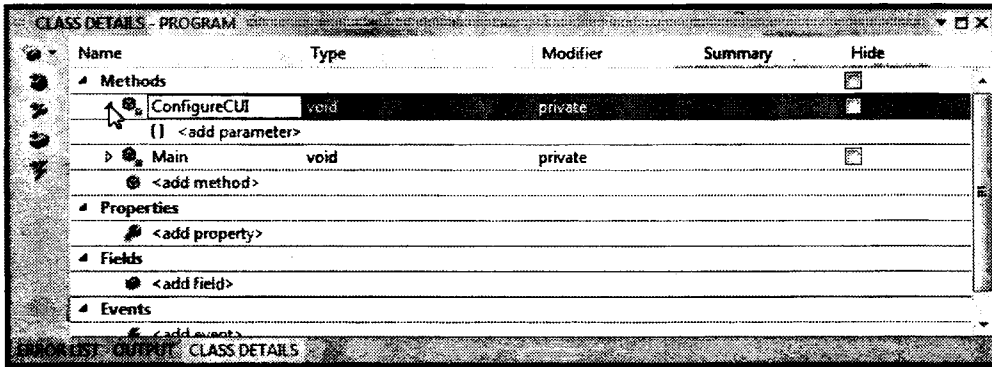


Рис. 2.18. Окно Class Details

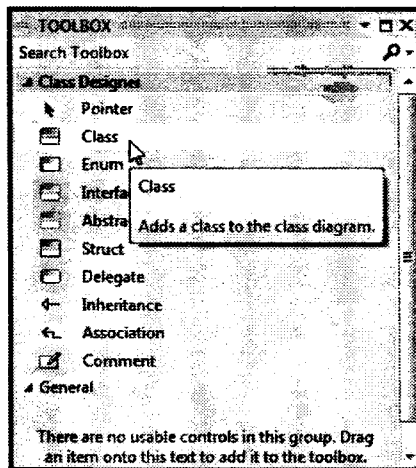


Рис. 2.19. Панель инструментов Class Designer

Для примера перетащим из панели инструментов Class Designer в окно Class Designer новый элемент Class (Класс). Затем в открывшемся диалоговом окне ему следует назначить имя Car и с помощью окна Class Details добавить открытое поле типа string по имени PetName (рис. 2.20).

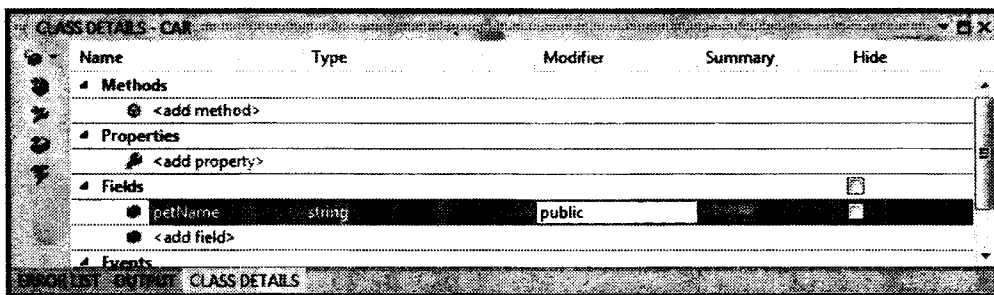


Рис. 2.20. Добавление поля с помощью окна Class Details

Если после этого взглянуть на C#-определение класса Car, можно увидеть, что оно было соответствующим образом обновлено (за исключением приведенного ниже комментария):

```
public class Car
{
    // Использовать открытые данные обычно не рекомендуется,
    // но здесь это упрощает пример.
    public string petName;
}
```

Теперь активизируем утилиту Class Designer еще раз и перетащим на поверхность конструктора другой элемент Class, назначив ему имя SportsCar. Затем выберем в панели инструментов Class Designer значок Inheritance (Наследование) и щелкнем в верхней части значка SportsCar. Далее щелкнем в верхней части значка класса Car. Если все было сделано правильно, то класс SportsCar станет производным от класса Car, как показано на рис. 2.21.

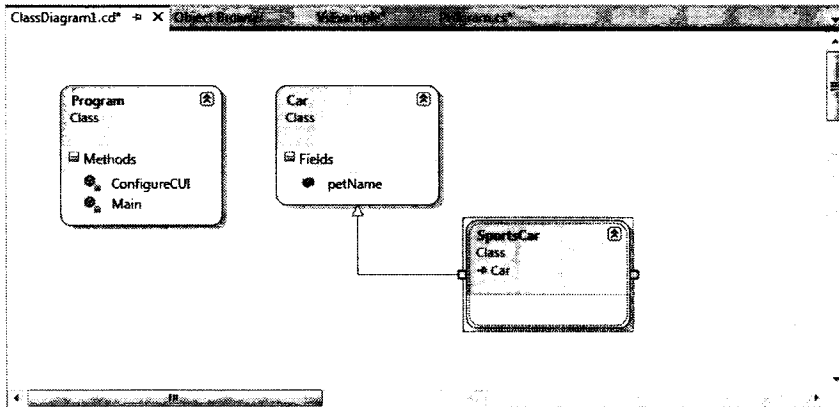


Рис. 2.21. Визуальное наследование от существующего класса

Чтобы завершить данный пример, обновим сгенерированный класс SportsCar, добавив в него открытый метод по имени GetPetName() со следующим кодом:

```
public class SportsCar : Car
{
    public string GetPetName()
    {
        petName = "Fred";
        return petName;
    }
}
```

Все эти (и другие) визуальные инструменты Visual Studio будут часто использоваться в дальнейшем материале книги. Однако вы уже должны чуть лучше понимать основные возможности этой IDE-среды.

На заметку! Концепция наследования подробно рассматривается в главе 6.

Интегрированная система документации .NET Framework 4.5 SDK

Финальным аспектом Visual Studio, с которым вы должны уметь работать с самого начала, является полностью интегрированная справочная система. Документация .NET Framework 4.5 SDK представляет собой исключительно хороший, понятный и насыщенный полезной информацией источник. Из-за огромного количества предопределенных типов .NET (а их тысячи) необходимо уделить время исследованиям предлагаемой документации, иначе вряд ли вас ожидает особый успех на поприще разработки .NET-приложений.

При наличии подключения к Интернету просматривать документацию .NET Framework 4.5 SDK можно в онлайн-режиме по следующему адресу:

<http://msdn.microsoft.com/library>

Далее, используя дерево навигации в левой части страницы, необходимо перейти к области Разработка на .NET и выбрать .NET Framework 4.5. После этого отобразится чрезвычайно важная ссылка Библиотека классов платформы .NET Framework, по которой можно найти документацию по каждому отдельному типу в каждом отдельном пространстве имен .NET (рис. 2.22).

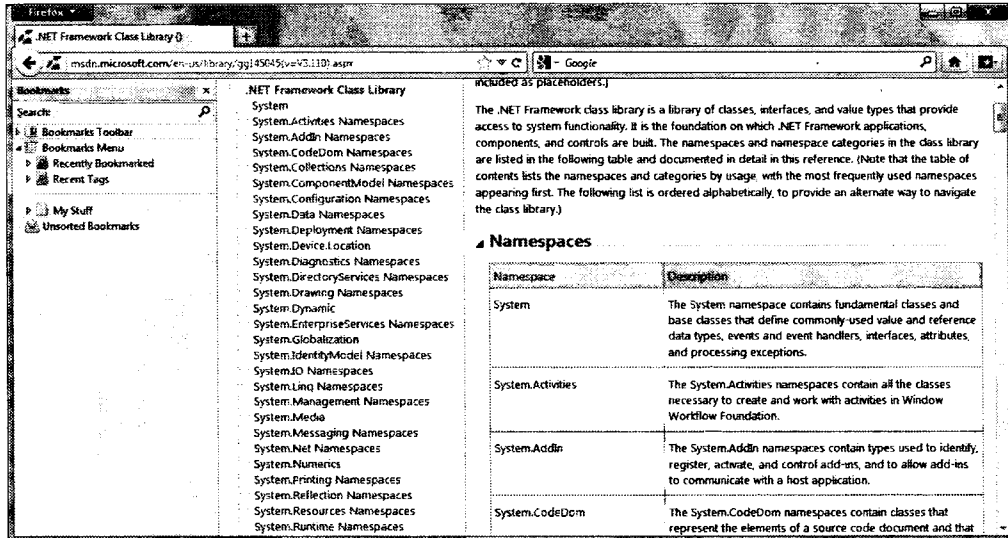


Рис. 2.22. Онлайн-просмотр документации по .NET Framework

Во время установки среды Visual Studio будет предоставлена возможность установки этой же справочной системы локально на компьютере (что может быть весьма полезно при отсутствии подключения к Интернету). Если требуется выполнить локальную установку справочной системы после установки самой среды, выберите в меню кнопки Start (Пуск) пункт All Programs ⇒ Microsoft Visual Studio 11 ⇒ Microsoft Help Viewer (Все программы ⇒ Microsoft Visual Studio 11 ⇒ Программа просмотра справки Microsoft). Затем можно приступить к добавлению интересующей справочной документации, как показано на рис. 2.23 (если места на диске достаточно, имеет смысл добавить всю возможную документацию).

Независимо от того, установлена справочная система локально или просматривается в онлайн-режиме, простейший способ взаимодействия с документацией предусматривает выделение ключевого слова C#, имени типа либо имени члена в окне кода Visual Studio и нажатие клавиши <F1>. Это приводит к открытию окна с документацией по выбранному элементу. Например, если выделить ключевое слово `string` в определении класса `Car` и нажать <F1>, появится страница справки для типа `string`.

Еще одним полезным аспектом документации является вкладка Search (Поиск). Здесь можно ввести имя любого пространства имен, типа или члена и перейти к соответствующему месту в документации. Например, если вы попытаетесь выполнить поиск для пространства имен `System.Reflection`, то сможете изучить детали этого пространства имен, исследовать содержащиеся внутри него типы, просмотреть примеры кода и т.д.

На заметку! Не лишним будет еще раз напомнить о том, насколько важно уметь пользоваться документацией .NET Framework 4.5 SDK. Ни одна книга, какой бы объемной она ни была, не способна охватить все аспекты платформы .NET. Поэтому необходимо научиться работать со справочной системой — впоследствии это окупится сторицей.

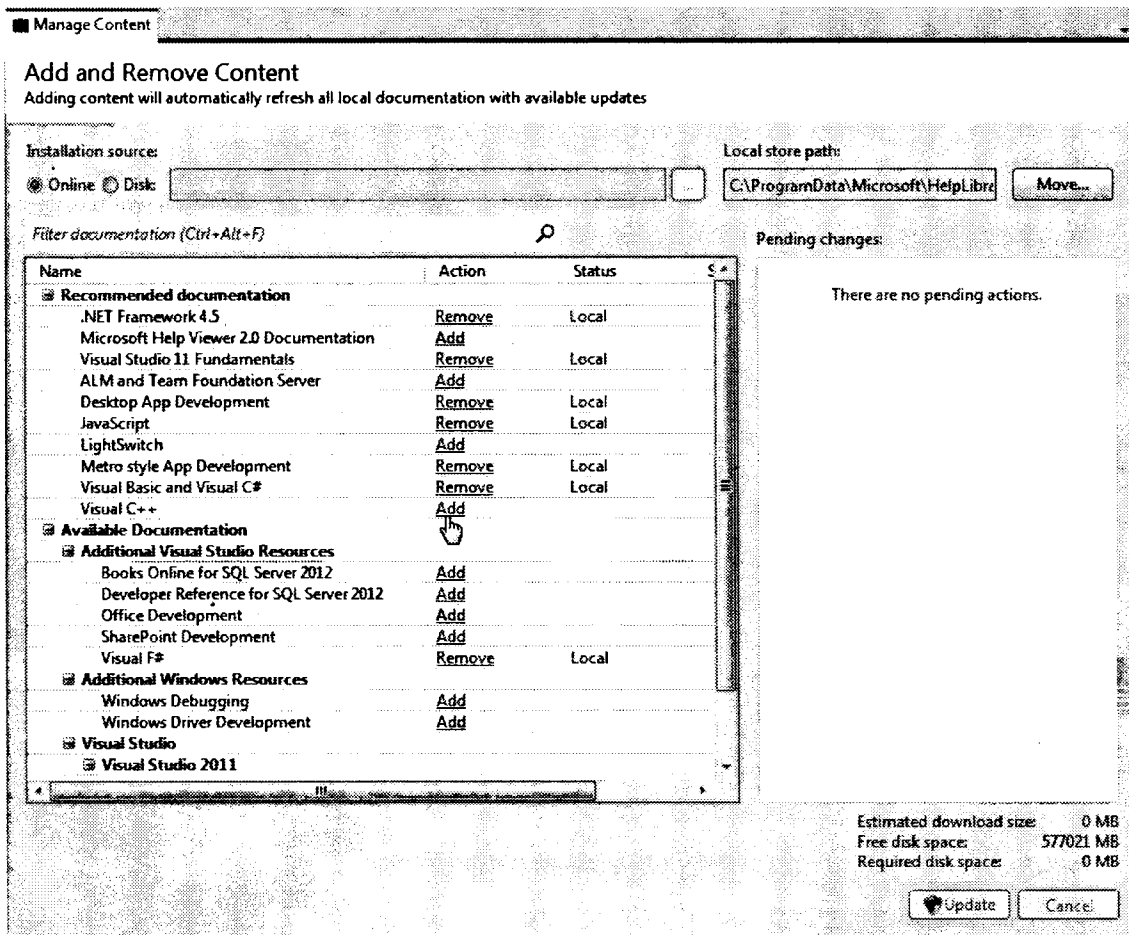


Рис. 2.23. Диспетчер библиотеки справки позволяет загрузить локальную копию документации .NET Framework 4.5 SDK

Резюме

Цель этой главы заключалась в проведении краткого экскурса в основные средства, которые программист на языке C# может использовать во время разработки. Мы начали с построения сборок .NET с помощью бесплатного компилятора C# (`csc.exe`) и приложений Notepad/Notepad++, а также исследовали процесс применения этих инструментов для редактирования и компиляции файлов кода *.cs.

В главе также были представлены три многофункциональных IDE-среды: SharpDevelop с открытым кодом, Microsoft Visual C# Express и Microsoft Visual Studio Professional. Функциональные возможности каждого из этих продуктов рассматривались лишь кратко, поэтому при желании можете заняться более детальным изучением той или иной IDE-среды (но помните, что дополнительные средства Visual Studio будут описаны в оставшихся главах книги).

часть II

Основы программирования на C#

В этой части

Глава 3. Главные конструкции программирования на C#: часть I

Глава 4. Главные конструкции программирования на C#: часть II

Главные конструкции программирования на C#: часть I

В этой главе начинается формальное изучение языка программирования C# с представления набора отдельных тем, которые необходимо знать для освоения платформы .NET Framework. В первую очередь нужно понять, как строить *объект приложения* и какова структура точки входа исполняемой программы — метода `Main()`. Затем будут описаны фундаментальные типы данных C# (и их эквиваленты в пространстве имен `System`), включая классы `System.String` и `System.Text.StringBuilder`.

После ознакомления с деталями фундаментальных типов данных .NET мы рассмотрим несколько приемов преобразования типов данных, в том числе сужающие и расширяющие операции, а также использование ключевых слов `checked` и `unchecked`.

Кроме того, в этой главе будет описана роль ключевого слова `var` языка C#, которое позволяет неявно определять локальную переменную. Как будет показано далее в книге, неявная типизация чрезвычайно удобна, а иногда и обязательна, при работе с набором технологий LINQ. Глава завершается кратким обзором ключевых слов и операций C#, которые позволяют управлять последовательностью выполняемых в приложении действий, используя различные конструкции циклов и принятия решений.

Структура простой программы C#

Язык C# требует, чтобы вся логика программы содержалась внутри определения типа (вспомните из главы 1, что *тип* — это общий термин, относящийся к любому элементу из множества {класс, интерфейс, структура, перечисление, делегат}). В отличие от многих других языков, в C# не допускается создание ни глобальных функций, ни глобальных элементов данных. Вместо этого все данные-члены и все методы должны содержаться внутри определения типа. Для начала создадим новый проект консольного приложения по имени `SimpleCSharpApp`. Как показано ниже, в исходном коде `Program.cs` нет ничего особо примечательного:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
namespace SimpleCSharpApp
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

С учетом этого модифицируем метод `Main()` класса `Program`, добавив в него следующие операторы кода:

```
class Program
{
    static void Main(string[] args)
    {
        // Вывести пользователю простое сообщение.
        Console.WriteLine("***** My First C# App *****");
        Console.WriteLine("Hello World!");
        Console.WriteLine();

        // Ожидать нажатия клавиши <Enter> перед завершением работы.
        Console.ReadLine();
    }
}
```

На заметку! C# является языком программирования, чувствительным к регистру. Следовательно, *Main* — не то же самое, что *main*, а *Readline* — не то же самое, что *ReadLine*. Запомните, что все ключевые слова C# вводятся в нижнем регистре (например, `public`, `lock`, `class`, `dynamic`), а названия пространств имен, типов и членов начинаются (по соглашению) с заглавной буквы и содержат заглавные буквы в любых вложенных в них словах (как, например, `Console.WriteLine`, `System.Windows.MessageBox` и `System.Data.SqlClient`). Как правило, при каждом получении от компилятора ошибки, связанной с неопределенными символами, в первую очередь следует проверить регистр символов и точность написания имен.

Здесь мы имеем определение типа класса, который поддерживает единственный метод по имени `Main()`. По умолчанию среда Visual Studio назначает классу, определяющему метод `Main()`, имя `Program`; однако при желании это можно изменить. Каждое исполняемое приложение C# (консольная программа, программа рабочего стола Windows или Windows-служба) должно содержать класс, определяющий метод `Main()`, который используется для обозначения точки входа в приложение.

Формально класс, который определяет метод `Main()`, называется *объектом приложения*. Хотя в одном исполняемом приложении разрешено иметь несколько объектов приложений (это может быть удобно при модульном тестировании), потребуется проинформировать компилятор о том, какой из методов `Main()` должен использоваться в качестве точки входа. Это делается либо через опцию `/main` компилятора командной строки, либо посредством раскрывающегося списка `Startup Object` (Объект запуска) на вкладке `Application` (Приложение) редактора свойств проекта Visual Studio (см. главу 2).

Обратите внимание, что сигнатура метода `Main()` снабжена ключевым словом `static`, которое будет подробно рассматриваться в главе 5. Пока же достаточно знать, что область действия статических членов охватывает уровень класса (а не уровень объектов), поэтому они могут вызываться без предварительного создания нового экземпляра класса.

Кроме ключевого слова `static` этот метод `Main()` принимает один параметр, который представляет собой массив строк `[string[] args]`. Хотя в текущий момент этот

массив никак не обрабатывается, данный параметр может содержать любое количество входных аргументов командной строки (доступ к ним будет описан чуть ниже). И, наконец, этот метод `Main()` был сконфигурирован с возвращаемым значением `void`, которое означает, что мы не определяем явно возвращаемое значение с помощью ключевого слова `return` перед выходом из области действия метода.

Логика `Program` содержится внутри самого метода `Main()`. Здесь используется класс `Console`, который определен в пространстве имен `System`. В состав его членов входит статический метод `WriteLine()`, который позволяет отправлять строку текста и символ возврата каретки на стандартное устройство вывода. Кроме того, здесь вызывается метод `Console.ReadLine()`, чтобы окно командной строки, запускаемое в IDE-среде `Visual Studio`, оставалось видимым во время сеанса отладки до тех пор, пока не будет нажата клавиша `<Enter>`. Класс `System.Console` более подробно рассматривается немного позже.

Вариации метода `Main()`

По умолчанию `Visual Studio` будет генерировать метод `Main()` с возвращаемым значением `void` и массивом `string` в качестве единственного входного параметра. Однако метод `Main()` может иметь не только такую форму. Допускается создавать собственные вариации точки входа в приложение с использованием любой из приведенных ниже сигнатур (предполагая, что они содержатся внутри определения класса или структуры `C#`):

```
// Возвращаемый тип int и массив строк в качестве параметра.
static int Main(string[] args)
{
    // Должен возвращать значение перед выходом!
    return 0;
}

// Нет ни возвращаемого типа, ни параметров.
static void Main()
{
}

// Возвращаемый тип int, но никаких параметров.
static int Main()
{
    // Должен возвращать значение перед выходом!
    return 0;
}
```

На заметку! Метод `Main()` может также быть определен как открытый, а не закрытый; последнее подразумевается, если не указан конкретный модификатор доступа. В `Visual Studio` метод `Main()` автоматически определяется как неявно закрытый.

Очевидно, что выбор способа создания метода `Main()` зависит от ответов на два вопроса. Первый из них: нужно ли возвращать значение системе после окончания выполнения метода `Main()` и завершения работы программы? Если да, то понадобится возвращать тип данных `int`, а не `void`. Второй вопрос формулируется так: требуется ли обрабатывать предоставляемые пользователем параметры командной строки? Если требуется, то они должны быть сохранены в массиве `string`. Давайте рассмотрим все возможные варианты более подробно.

Указание кода ошибки приложения

Хотя в подавляющем большинстве случаев методы `Main()` имеют `void` в качестве возвращаемого значения, возможность возврата `int` из `Main()` сохраняет согласованность C# с другими языками, основанными на C. По соглашению, возврат значения 0 свидетельствует о том, что выполнение программы прошло успешно, тогда как любое другое значение (такое как -1) представляет условие ошибки (имейте в виду, что значение 0 возвращается автоматически даже в случае, если метод `Main()` прототипирован как возвращающий `void`).

В Windows возвращаемое приложением значение сохраняется в переменной среды по имени `%ERRORLEVEL%`. Если создается приложение, которое программно запускает другой исполняемый файл (тема, рассматриваемая в главе 17), то получить значение `%ERRORLEVEL%` можно с помощью статического свойства `System.Diagnostics.Process.ExitCode`.

Учитывая, что возвращаемое значение передается системе в момент завершения работы приложения, очевидно, что приложение не может получить и отобразить финальный код ошибки во время выполнения. Однако мы покажем, как просмотреть код ошибки по завершении программы, для чего модифицируем метод `Main()` следующим образом:

```
// Обратите внимание, что теперь возвращается int, а не void.
static int Main(string[] args)
{
    // Вывести сообщение и ожидать нажатия клавиши <Enter>.
    Console.WriteLine("***** My First C# App *****");
    Console.WriteLine("Hello World!");
    Console.WriteLine();
    Console.ReadLine();
    // Возвратить произвольный код ошибки.
    return -1;
}
```

Теперь давайте захватим возвращаемое методом `Main()` значение с помощью пакетного файла. В проводнике Windows перейдите в папку, содержащую скомпилированное приложение (например, `C:\SimpleCSharpApp\bin\Debug`). Добавьте в папку Debug новый текстовый файл (по имени `SimpleCSharpApp.bat`), содержащий следующие инструкции (если раньше вам не приходилось создавать файлы `.bat`, можете не задумываться над его нюансами — это всего лишь тест):

```
@echo off

rem Пакетный файл для приложения SimpleCSharpApp.exe,
rem который захватывает возвращаемое им значение.

SimpleCSharpApp
@if "%ERRORLEVEL%" == "0" goto success

:fail
rem Выполнение этого приложения не удалось!
echo This application has failed!
echo return value = %ERRORLEVEL%
goto end

:success
rem Выполнение этого приложения прошло успешно!
echo This application has succeeded!
echo return value = %ERRORLEVEL%
goto end

:end
rem Все готово.
echo All Done.
```

Теперь откройте окно командной строки разработчика (см. главу 2) и перейдите в папку, содержащую исполняемый файл приложения и только что созданный файл *.bat. Запустите пакетный файл, набрав его имя и нажав <Enter>. После этого на экране появится показанный ниже вывод, т.к. метод Main() сейчас возвращает значение -1. Если бы он возвращал 0, вы увидели бы в окне консоли сообщение This application has succeeded!.

```
***** My First C# App *****
Hello World!

This application has failed!
return value = -1
All Done.
```

В подавляющем большинстве приложений C# (если не во всех) в качестве возвращаемого значения метода Main() будет использоваться void, что, как уже известно, подразумевает неявный возврат кода ошибки, равного 0. Поэтому все демонстрируемые далее в книге методы Main() будут возвращать void (никакие пакетные файлы для перехвата кода возврата в последующих проектах не используются).

Обработка аргументов командной строки

Теперь, когда стало более понятно, что собой представляет возвращаемое значение метода Main(), давайте посмотрим на входной массив данных string. Предположим, что теперь необходимо обновить приложение так, чтобы оно могло обрабатывать любые возможные параметры командной строки. Один из способов предусматривает использование цикла for в C# (все итерационные конструкции C# более подробно рассматриваются в конце этой главы):

```
static int Main(string[] args)
{
    ...
    // Обработать любые входные аргументы.
    for(int i = 0; i < args.Length; i++)
        Console.WriteLine("Arg: {0}", args[i]);

    Console.ReadLine();
    return -1;
}
```

Здесь с применением свойства Length класса System.Array производится проверка, содержит ли массив string какие-то элементы. Как будет показано в главе 4, все массивы C# на самом деле относятся к классу System.Array и потому разделяют общий набор членов. При проходе в цикле по массиву значение каждого элемента выводится на консоль. Предоставить аргументы в командной строке сравнительно просто:

```
C:\SimpleCSharpApp\bin\Debug>SimpleCSharpApp.exe /arg1 -arg2

***** My First C# App *****
Hello World!
Arg: /arg1
Arg: -arg2
```

В качестве альтернативы стандартному циклу for для прохода по входному массиву string можно также использовать ключевое слово foreach. Ниже приведен пример одного из возможных применений (подробности конструкций циклов будут рассмотрены далее в этой главе):

```
// Обратите внимание, что в случае использования
// цикла foreach проверять размер массива не требуется.
static int Main(string[] args)
{
    ...
    // Обработать любые входные аргументы с помощью foreach.
    foreach(string arg in args)
        Console.WriteLine("Arg: {0}", arg);

    Console.ReadLine();
    return -1;
}
```

И, наконец, доступ к аргументам командной строки можно также получать с помощью статического метода `GetCommandLineArgs()` типа `System.Environment`. Возвращаемым значением этого метода является массив `string`. Первый элемент этого массива содержит имя самого приложения, а остальные элементы — отдельные аргументы командной строки. Обратите внимание, что при таком подходе больше не обязательно определять метод `Main()` как принимающий массив `string` в качестве входного параметра, хотя никакого вреда от этого не будет.

```
static int Main(string[] args)
{
    ...
    // Получить аргументы с использованием System.Environment.
    string[] theArgs = Environment.GetCommandLineArgs();
    foreach(string arg in theArgs)
        Console.WriteLine("Arg: {0}", arg);

    Console.ReadLine();
    return -1;
}
```

Разумеется, определение того, на какие аргументы командной строки будет реагировать программа (если вообще будет), и как они должны быть сформатированы (скажем, с префиксом `-` или `/`), возлагается целиком на вас. В приведенном выше коде мы просто передаем последовательность опций, которые выводятся прямо в окно командной строки. Однако предположим, что создается новое игровое приложение, запрограммированное на обработку опции по имени `-godmode`. Тогда при запуске пользователем приложения с этим флагом можно было бы предпринимать в его отношении соответствующие действия.

Указание аргументов командной строки в Visual Studio

В реальности конечный пользователь имеет возможность предоставления аргументов командной строки при запуске программы. Однако во время разработки также может потребоваться указывать допустимые флаги командной строки в целях тестирования. Чтобы сделать это в Visual Studio, дважды щелкните на значке `Properties` (Свойства) в проводнике решений, выберите в левой части окна вкладку `Debug` (Отладка) и укажите в текстовом поле `Command line arguments` (Аргументы командной строки) желаемые аргументы (рис. 3.1).

Указанные аргументы командной строки будут автоматически передаваться методу `Main()` во время отладки или выполнения приложения в IDE-среде Visual Studio.

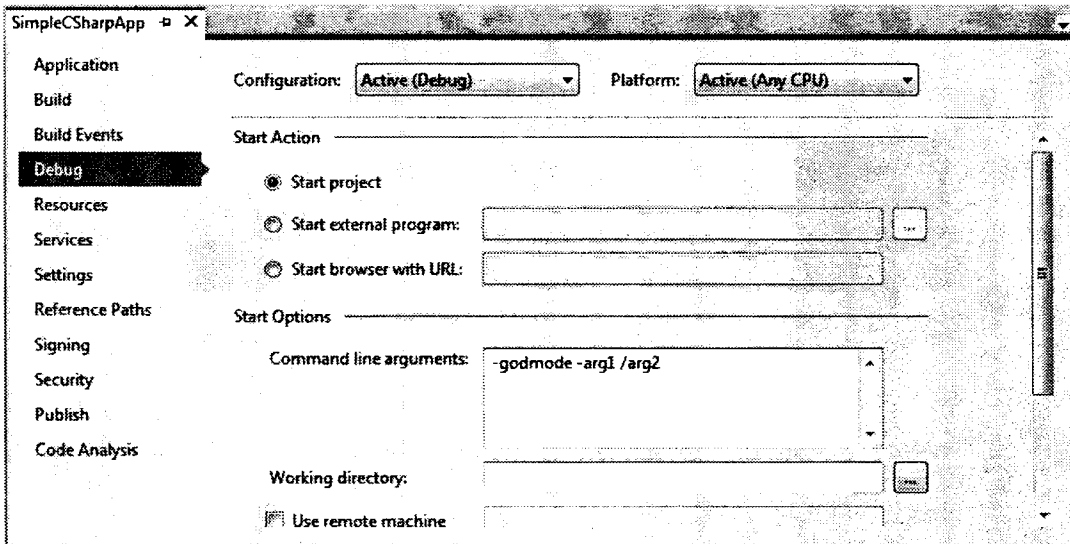


Рис. 3.1. Указание аргументов командной строки в Visual Studio

Интересное отступление от темы: некоторые дополнительные члены класса `System.Environment`

Помимо `GetCommandLineArgs()` в классе `Environment` открыто несколько других чрезвычайно полезных методов. В частности, с помощью различных статических членов этот класс позволяет получать детальные сведения, касающиеся операционной системы, под управлением которой в текущий момент выполняется .NET-приложение. В целях иллюстрации пользы класса `System.Environment` модифицируем метод `Main()` для вызова вспомогательного метода по имени `ShowEnvironmentDetails()`:

```
static int Main(string[] args)
{
    ...
    // Вспомогательный метод внутри класса Program.
    ShowEnvironmentDetails();
    Console.ReadLine();
    return -1;
}
```

Теперь реализуем этот метод в классе `Program`, в котором будут вызываться различные члены типа `Environment`:

```
static void ShowEnvironmentDetails()
{
    // Вывести информацию о дисковых устройствах
    // данной машины и другие интересные детали.
    foreach (string drive in Environment.GetLogicalDrives())
        Console.WriteLine("Drive: {0}", drive);

    Console.WriteLine("OS: {0}", Environment.OSVersion); // операционная система
    Console.WriteLine("Number of processors: {0}",
        Environment.ProcessorCount); // количество процессоров
    Console.WriteLine(".NET Version: {0}",
        Environment.Version); // версия .NET
}
```

Ниже показан возможный результат тестового запуска данного метода. Естественно, если на вкладке Debug в Visual Studio не были указаны аргументы командной строки, то в окно консоли они выводиться не будут.

```
***** My First C# App *****
Hello World!
Arg: -godmode
Arg: -arg1
Arg: /arg2

Drive: C:\
Drive: D:\
Drive: E:\
Drive: F:\
Drive: G:\
Drive: H:\
Drive: I:\
OS: Microsoft Windows NT 6.1.7601 Service Pack 1
Number of processors: 4
.NET Version: 4.0.30319.17020
```

Тип Environment определяет и другие члены кроме показанных в предыдущем примере. В табл. 3.1 перечислены некоторые интересные дополнительные свойства; исчерпывающую информацию можно найти в документации .NET Framework 4.5 SDK.

Таблица 3.1. Избранные свойства System.Environment

Свойство	Описание
ExitCode	Возвращает или устанавливает код возврата для приложения
Is64BitOperatingSystem	Возвращает булевское значение, представляющее то, работает ли данный компьютер под управлением 64-разрядной операционной системы
MachineName	Возвращает имя текущей машины
NewLine	Возвращает символ новой строки для текущей среды
SystemDirectory	Возвращает полный путь к системному каталогу
UserName	Возвращает имя пользователя, который запустил данное приложение
Version	Возвращает объект Version, который представляет версию платформы .NET

Исходный код. Проект SimpleCSharpApp доступен в подкаталоге Chapter 03.

Класс System.Console

Почти во всех примерах приложений, создаваемых на протяжении первых нескольких глав, будет интенсивно использоваться класс System.Console. И хотя в действительности консольный пользовательский интерфейс не настолько привлекателен, как графический пользовательский интерфейс либо интерфейс веб-приложения, ограничение начальных примеров до консольных программ позволяет сосредоточить все внимание на синтаксисе C# и ключевых аспектах платформы .NET, не отвлекаясь на сложные детали построения графических пользовательских интерфейсов настольных приложений или веб-сайтов.

Класс `Console` инкапсулирует средства манипуляции вводом, выводом и потоками ошибок для консольных приложений. В табл. 3.2 перечислены некоторые интересные его члены. Как легко убедиться, класс `Console` предоставляет ряд членов, которые позволяют придавать дополнительные возможности приложениям командной строки, такие как изменение цветов фона и переднего плана и выдача звуковых сигналов (различной частоты).

Таблица 3.2. Избранные члены `System.Console`

Член	Описание
<code>Beep()</code>	Этот метод заставляет консоль выдать звуковой сигнал указанной частоты и длительности
<code>BackgroundColor</code> <code>ForegroundColor</code>	Эти свойства устанавливают цвета фона и переднего плана для текущего вывода. Им может быть присвоен любой член перечисления <code>ConsoleColor</code>
<code>BufferHeight</code> <code>BufferWidth</code>	Эти свойства управляют высотой и шириной буферной области консоли
<code>Title</code>	Это свойство устанавливает заголовок текущей консоли
<code>WindowHeight</code> <code>WindowWidth</code> <code>WindowTop</code> <code>WindowLeft</code>	Эти свойства управляют размерами консоли по отношению к установленному буферу
<code>Clear()</code>	Этот метод позволяет очищать установленный буфер и область отображения консоли

Базовый ввод-вывод с помощью класса `Console`

Помимо членов, перечисленных в табл. 3.2, тип `Console` определяет методы, которые позволяют захватывать ввод и вывод; все они являются статическими и потому вызываются с предварением имени метода именем самого класса (`Console`). Как уже было указано, метод `WriteLine()` позволяет поместить в поток вывода строку текста (включая символ возврата каретки). Метод `Write()` помещает в поток вывода текст без символа возврата каретки. Метод `ReadLine()` позволяет получить информацию из потока ввода вплоть до нажатия клавиши `<Enter>`. Метод `Read()` используется для захвата из потока ввода одиночного символа.

Для демонстрации базового ввода-вывода с применением класса `Console` создадим новый проект консольного приложения по имени `BasicConsoleIO` и модифицируем метод `Main()` для вызова вспомогательного метода `GetUserData()`:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Basic Console I/O *****");
        GetUserData();
        Console.ReadLine();
    }
}
```

На заметку! Тема фрагментов кода Visual Studio уже кратко затрагивалась в главе 2. Фрагмент кода `sw` очень полезен в начальных главах этой книги, поскольку он автоматически разворачивается в метод `Console.WriteLine()`. Чтобы удостовериться в этом, введите `sw` где-либо внутри метода `Main()` и два раза нажмите клавишу `<Tab>`. К сожалению, фрагмента кода для метода `Console.ReadLine()` не предусмотрено.

Теперь реализуем этот метод в классе Program вместе с логикой, которая приглашает пользователя ввести некоторые сведения и затем отображает их на стандартном устройстве вывода. Например, мы могли бы запросить у пользователя его имя и возраст (который для простоты будет трактоваться как текстовое значение, а не привычное числовое):

```
static void GetUserData()
{
    // Получить информацию об имени и возрасте.
    Console.Write("Please enter your name: "); // Запрос на ввод имени
    string userName = Console.ReadLine();
    Console.Write("Please enter your age: "); // Запрос на ввод возраста
    string userAge = Console.ReadLine();

    // Изменить цвет переднего плана, просто ради интереса.
    ConsoleColor prevColor = Console.ForegroundColor;
    Console.ForegroundColor = ConsoleColor.Yellow;

    // Вывести полученные сведения на консоль.
    Console.WriteLine("Hello {0}! You are {1} years old.",
        userName, userAge);

    // Восстановить предыдущий цвет переднего плана.
    Console.ForegroundColor = prevColor;
}
```

После запуска этого приложения входные данные будут выводиться в окне консоли (с использованием указанного специального цвета).

Форматирование консольного вывода

В ходе первых нескольких глав вы могли заметить, что внутри различных строковых литералов часто встречались конструкции вроде {0} и {1}. В .NET для форматирования строк поддерживается стиль, немного напоминающий стиль оператора printf() языка C. Выражаясь проще, при определении строкового литерала с сегментами данных, значения которых остаются неизвестными до этапа выполнения, внутри него допускается указывать заполнитель, используя синтаксис в виде фигурных скобок. Во время выполнения на место каждого такого заполнителя подставляется передаваемое в Console.WriteLine() значение (или значения).

Первый параметр метода WriteLine() представляет строковый литерал, который содержит заполнители вида {0}, {1}, {2} и т.д. Запомните, что порядковые числа заполнителей в фигурных скобках всегда начинаются с 0. Остальные параметры WriteLine() — это просто значения, которые должны подставляться на месте заполнителей.

На заметку! Если количество уникально нумерованных заполнителей превышает количество заполняющих аргументов, во время выполнения будет сгенерировано исключение форматирования. Однако если заполняющих аргументов больше, чем заполнителей, то лишние аргументы просто игнорируются.

Один и тот же заполнитель может повторяться в пределах заданной строки. Например, для создания строки "9, Number9, Number9" можно было бы написать такой код:

```
// Вывод строки "9, Number9, Number9"
Console.WriteLine("{0}, Number {0}, Number {0}", 9);
```


Также следует знать о том, что каждый заполнитель допускается размещать в любом месте внутри строкового литерала, и вовсе не обязательно, чтобы заполнители указывались в возрастающей последовательности своих номеров, например:

```
// Выводит: 20, 10, 30
Console.WriteLine("{1}, {0}, {2}", 10, 20, 30);
```

Форматирование числовых данных

Если для числовых данных требуется более сложное форматирование, каждый заполнитель может дополнительно содержать разнообразные символы форматирования, наиболее полезные из которых перечислены в табл. 3.3.

Таблица 3.3. Символы для форматирования числовых данных в .NET

Символ форматирования	Описание
C или c	Используется для форматирования денежных значений. По умолчанию значение предваряется символом локальной валюты (например, знаком доллара (\$)) для культуры US English)
D или d	Используется для форматирования десятичных чисел. В этом флаге можно также указывать минимальное количество цифр для представления значения
E или e	Используется для экспоненциального представления. Регистр этого флага указывает, в каком регистре должна представляться экспоненциальная константа — в верхнем (E) или в нижнем (e)
F или f	Используется для форматирования с фиксированной точкой. В этом флаге можно также указывать минимальное количество цифр для представления значения
G или g	Означает <i>general</i> (общий). Этот флаг может использоваться для представления чисел в формате с фиксированной точкой или экспоненциальном формате
N или n	Используется для базового числового форматирования (с запятыми)
X или x	Используется для шестнадцатеричного форматирования. В случае символа X в верхнем регистре шестнадцатеричное представление будет содержать символы верхнего регистра

Эти символы форматирования добавляются в виде суффиксов к заполнителям после двоеточия (например, {0:C}, {1:d}, {2:X}). В целях иллюстрации модифицируем метод Main() для вызова вспомогательного метода по имени FormatNumericalData(). Реализация этого метода в классе Program форматирует фиксированное числовое значение несколькими способами.

```
// Использовать несколько дескрипторов формата.
static void FormatNumericalData()
{
    Console.WriteLine("The value 99999 in various formats:");
    Console.WriteLine("c format: {0:c}", 99999);
    Console.WriteLine("d9 format: {0:d9}", 99999);
    Console.WriteLine("f3 format: {0:f3}", 99999);
    Console.WriteLine("n format: {0:n}", 99999);

    // Обратите внимание, что использование верхнего или нижнего регистра для x
    // определяет, в каком регистре отображаются символы в шестнадцатеричном формате.
    Console.WriteLine("E format: {0:E}", 99999);
    Console.WriteLine("e format: {0:e}", 99999);
}
```

```

Console.WriteLine("X format: {0:X}", 99999);
Console.WriteLine("x format: {0:x}", 99999);
}

```

Ниже приведен результат вывода метода `FormatNumericalData()`:

The value 99999 in various formats:

```

c format: $99,999.00
d9 format: 000099999
f3 format: 99999.000
n format: 99,999.00
E format: 9.999900E+004
e format: 9.999900e+004
X format: 1869F
x format: 1869f

```

Далее в книге будут встречаться и другие примеры форматирования; исчерпывающую информацию о форматировании строк в .NET можно найти в документации .NET Framework 4.5 SDK.

Исходный код. Проект `BasicConsoleIO` доступен в подкаталоге `Chapter 03`.

Форматирование числовых данных в приложениях, отличных от консольных

Напоследок следует отметить, что использованием символов форматирования строк .NET не ограничивается только консольными приложениями. Тот же самый синтаксис форматирования можно применять при вызове статического метода `string.Format()`. Это может оказаться удобным при динамической компоновке текстовых данных для использования в приложении любого типа (например, в настольном приложении с графическим пользовательским интерфейсом, в веб-приложении ASP.NET и т.д.).

Метод `string.Format()` возвращает новый объект `string`, который форматируется в соответствии с предоставляемыми флагами. После этого текстовые данные могут использоваться любым желаемым образом. Для примера предположим, что требуется создать графическое настольное WPF-приложение и сформатировать строку, отображаемую внутри него в окне сообщения. Решение этой задачи демонстрируется в показанном ниже коде. Однако имейте в виду, что этот код не скомпилируется до тех пор, пока в проект не будет включена ссылка на сборку `PresentationFramework.dll` (добавление ссылок на библиотеки в Visual Studio было описано в главе 2).

```

static void DisplayMessage()
{
    // Использование string.Format() для форматирования строкового литерала.
    string userMessage = string.Format("100000 in hex is {0:x}", 100000);

    // Для компиляции этой строки кода требуется
    // ссылка на PresentationFramework.dll!
    System.Windows.MessageBox.Show(userMessage);
}

```

Системные типы данных и соответствующие ключевые слова C#

Подобно любому другому языку программирования, в C# определены ключевые слова для фундаментальных типов данных, которые применяются для представления ло-

кальных переменных, переменных-членов данных в классах, возвращаемых значений и параметров методов. Однако в отличие от других языков программирования, в C# эти ключевые слова представляют собой нечто большее, чем просто распознаваемые компилятором лексемы. В сущности, они являются сокращенными обозначениями полноценных типов из пространства имен `System`. В табл. 3.4 перечислены эти системные типы данных вместе с их диапазонами значений, соответствующими ключевыми словами C# и сведениями о совместимости с общезыковой спецификацией (CLS).

На заметку! Вспомните из главы 1, что совместимый с CLS код .NET может использоваться любым управляемым языком программирования. Если в программе присутствуют данные, не совместимые с CLS, другие языки могут быть не в состоянии работать с ними.

Таблица 3.4. Внутренние типы данных C#

Сокращенное обозначение в C#	Совместимость с CLS	Системный тип	Диапазон	Описание
<code>bool</code>	Да	<code>System.Boolean</code>	<code>true</code> или <code>false</code>	Признак истинности или ложности
<code>sbyte</code>	Нет	<code>System.SByte</code>	от -128 до 127	8-битное число со знаком
<code>byte</code>	Да	<code>System.Byte</code>	от 0 до 255	8-битное число без знака
<code>short</code>	Да	<code>System.Int16</code>	от -32 768 до 32 767	16-битное число со знаком
<code>ushort</code>	Нет	<code>System.UInt16</code>	от 0 до 65 535	16-битное число без знака
<code>int</code>	Да	<code>System.Int32</code>	от -2 147 483 648 до 2 147 483 647	32-битное число со знаком
<code>uint</code>	Нет	<code>System.UInt32</code>	от 0 до 4 294 967 295	32-битное число без знака
<code>long</code>	Да	<code>System.Int64</code>	от -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807	64-битное число со знаком
<code>ulong</code>	Нет	<code>System.UInt64</code>	от 0 до 18 446 744 073 709 551 615	64-битное число без знака
<code>char</code>	Да	<code>System.Char</code>	от U+0000 до U+ffff	Одиночный 16-битный символ Unicode
<code>float</code>	Да	<code>System.Single</code>	от -3.4×10^{38} до $+3.4 \times 10^{38}$	32-битное число с плавающей точкой
<code>double</code>	Да	<code>System.Double</code>	от $\pm 5.0 \times 10^{-324}$ до $\pm 1.7 \times 10^{308}$	64-битное число с плавающей точкой
<code>decimal</code>	Да	<code>System.Decimal</code>	(от -7.9×10^{28} до 7.9×10^{28}) / (10 ^{от 0 до 28})	128-битное число со знаком
<code>string</code>	Да	<code>System.String</code>	Ограничен объемом системной памяти	Набор символов Unicode
<code>object</code>	Да	<code>System.Object</code>	В переменной <code>object</code> может храниться любой тип данных	Базовый класс для всех типов в мире .NET

На заметку! По умолчанию число с плавающей точкой трактуется как `double`. Чтобы получить значение типа `float`, после числа необходимо указать суффикс `f` или `F` (например, `5.3F`), а значение типа `decimal` — суффикс `m` или `M` (например, `300.5M`). И, наконец, неформатированные целые числа по умолчанию трактуются как `int`. Чтобы получить значение типа `long`, понадобится указать суффикс `l` или `L` (например, `4L`).

Объявление и инициализация переменных

Для объявления локальной переменной (например, переменной внутри контекста члена) необходимо указать тип данных и сразу за ним имя переменной. Чтобы посмотреть, как это выглядит, создадим новый проект консольного приложения по имени `BasicDataTypes` и модифицируем класс `Program` так, чтобы в его методе `Main()` вызывался следующий вспомогательный метод:

```
static void LocalVarDeclarations()
{
    Console.WriteLine("=> Data Declarations:");
    // Локальные переменные объявляются следующим образом:
    // типДанных имяПеременной;
    int myInt;
    string myString;

    Console.WriteLine();
}
```

Имейте в виду, что использование локальной переменной до присваивания ей начального значения приведет к ошибке на этапе компиляции. С учетом этого, рекомендуется присваивать начальные значения локальным переменным во время их объявления. Делать это можно как в одной и той же строке, так и разносить объявление и присваивание на два отдельных оператора кода.

```
static void LocalVarDeclarations()
{
    Console.WriteLine("=> Data Declarations:");
    // Локальные переменные объявляются и инициализируются следующим образом:
    // типДанных имяПеременной = начальноеЗначение;
    int myInt = 0;

    // Объявлять и присваивать можно также в двух отдельных строках.
    string myString;
    myString = "This is my character data";

    Console.WriteLine();
}
```

Также разрешено объявлять множество переменных одного и того же типа в единственной строке кода, как показано ниже на примере трех переменных `bool`:

```
static void LocalVarDeclarations()
{
    Console.WriteLine("=> Data Declarations:");
    int myInt = 0;
    string myString;
    myString = "This is my character data";

    // Объявить три переменных типа bool в одной строке.
    bool b1 = true, b2 = false, b3 = b1;
    Console.WriteLine();
}
```

Поскольку ключевое слово `bool` в C# — это просто сокращенное обозначение структуры `System.Boolean`, также возможно размещать любой тип данных, используя его полное имя (естественно, это же касается всех остальных ключевых слов C#, представляющих типы данных). Ниже приведена окончательная реализация метода `LocalVarDeclarations()`, в которой демонстрируются различные способы объявления локальных переменных.

```
static void LocalVarDeclarations()
{
    Console.WriteLine("> Data Declarations:");
    // Локальные переменные объявляются и инициализируются следующим образом:
    // типДанных имяПеременной = начальноеЗначение;
    int myInt = 0;
    string myString;
    myString = "This is my character data";

    // Объявить три переменных типа bool в одной строке.
    bool b1 = true, b2 = false, b3 = b1;

    // Использовать тип данных System.Boolean для объявления булевской переменной.
    System.Boolean b4 = false;
    Console.WriteLine("Your data: {0}, {1}, {2}, {3}, {4}, {5}",
        myInt, myString, b1, b2, b3, b4);
    Console.WriteLine();
}
```

Внутренние типы данных и операция `new`

Все внутренние типы данных поддерживают так называемый стандартный конструктор (тема, которая более подробно рассматривается в главе 5). Это средство позволяет создавать переменные с использованием ключевого слова `new` и автоматически устанавливать для них стандартные значения:

- значение `false` для переменных типа `bool`;
- значение `0` для переменных числовых типов (`0.0` для типов с плавающей точкой);
- пустой символ для переменных типа `char`;
- значение `0` для переменных типа `BigInteger`;
- значение `1/1/0001 12:00:00 AM` для переменных типа `DateTime`;
- значение `null` для объектных ссылок (включая `string`).

На заметку! Тип данных `BigInteger`, упомянутый в приведенном списке, будет описан чуть позже.

Применение ключевого слова `new` во время создания переменных базовых типов дает более громоздкий код C#, что можно видеть в следующем примере:

```
static void NewingDataTypes()
{
    Console.WriteLine("> Using new to create variables:");
    bool b = new bool();           // Установить в false.
    int i = new int();             // Установить в 0.
    double d = new double();       // Установить в 0.
    DateTime dt = new DateTime();   // Установить в 1/1/0001 12:00:00 AM
    Console.WriteLine("{0}, {1}, {2}, {3}", b, i, d, dt);
    Console.WriteLine();
}
```

Иерархия классов для типов данных

Очень интересно отметить, что даже элементарные типы данных в .NET организованы в иерархию классов. Если вы не знакомы с концепцией наследования, ищите всю необходимую информацию в главе 6. Пока важно понять лишь то, что типы, находящиеся в самом верху иерархии, предоставляют определенное стандартное поведение, которое передается производным типам. На рис. 3.2 показаны отношения между ключевыми системными типами.

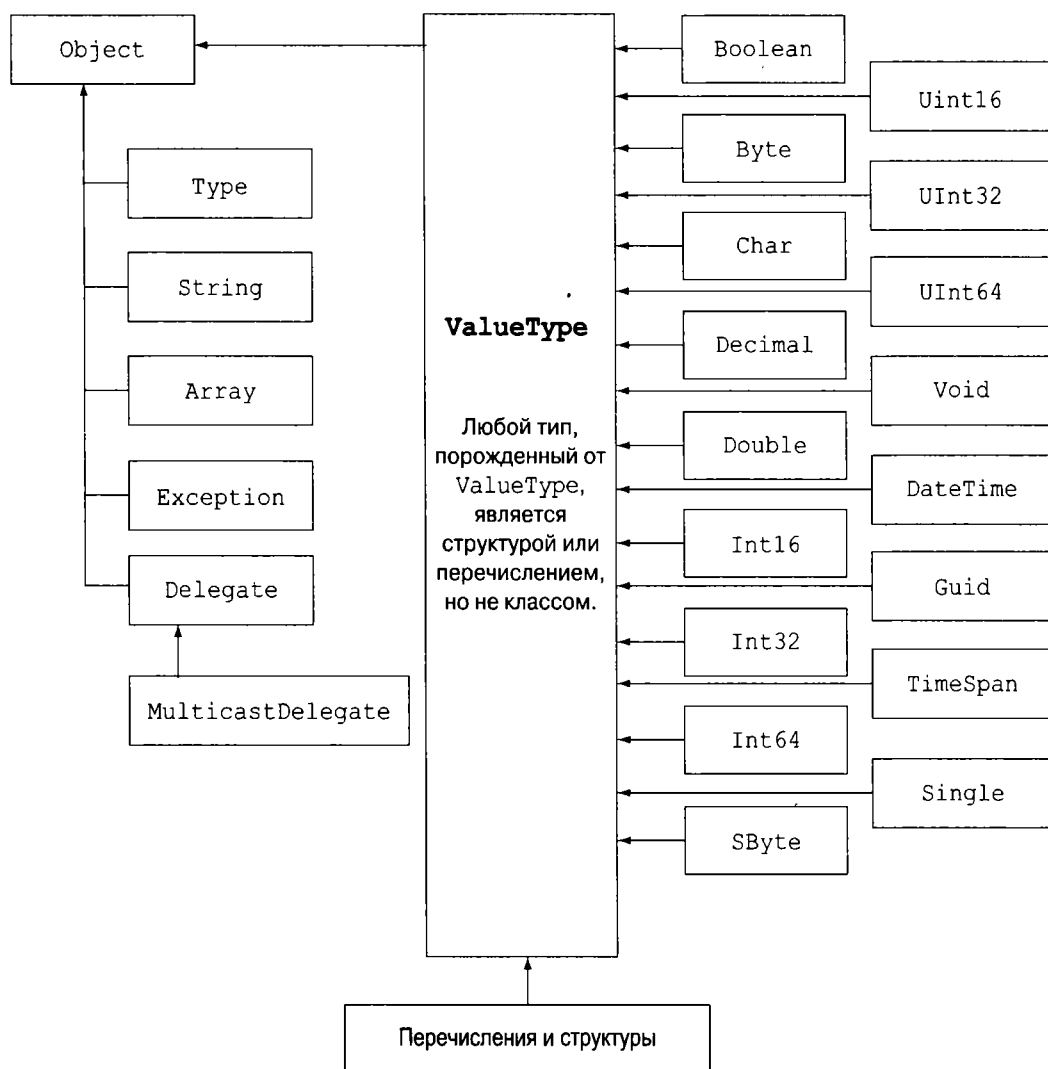


Рис. 3.2. Иерархия классов для системных типов

Обратите внимание, что каждый из этих типов в конечном итоге порожден от `System.Object`, который определяет набор методов (например, `ToString()`, `Equals()`, `GetHashCode()`), общих для всех типов в библиотеках базовых классов .NET (все эти методы подробно рассматриваются в главе 6).

Также важно отметить, что многие числовые типы данных порождены от `System.ValueType`. Потомки `ValueType` автоматически размещаются в стеке и, таким образом, имеют очень предсказуемое время жизни и являются довольно эффективными.

С другой стороны, типы, которые не имеют `System.ValueType` в своей цепочке наследования (такие как `System.Type`, `System.String`, `System.Array`, `System.Exception` и `System.Delegate`), размещаются не в стеке, а в куче с автоматической сборкой мусора. (Более подробно эти различия рассматриваются в главе 4.)

Не вдаваясь особо в детали классов `System.Object` и `System.ValueType`, важно уяснить, что поскольку любое ключевое слово C# (например, `int`) представляет собой просто сокращенное обозначение соответствующего системного типа (в этом случае `System.Int32`), приведенный ниже синтаксис является вполне допустимым. Причина в том, что тип `System.Int32` (`int` в C#) в конечном итоге порожден от `System.Object` и, таким образом, может обращаться к любому из его открытых членов, как показано в следующей вспомогательной функции:

```
static void ObjectFunctionality()
{
    Console.WriteLine("=> System.Object Functionality:");

    // Ключевое слово int в C# - это в действительности сокращение для класса
    // System.Int32, который наследует от System.Object следующие члены:
    Console.WriteLine("12.GetHashCode() = {0}", 12.GetHashCode());
    Console.WriteLine("12.Equals(23) = {0}", 12.Equals(23));
    Console.WriteLine("12.ToString() = {0}", 12.ToString());
    Console.WriteLine("12.GetType() = {0}", 12.GetType());
    Console.WriteLine();
}
```

Вызов этого метода в `Main()` дает такой вывод:

```
=> System.Object Functionality:
12.GetHashCode() = 12
12.Equals(23) = False
12.ToString() = 12
12.GetType() = System.Int32
```

Члены числовых типов данных

Продолжая эксперименты со встроенными типами данных C#, отметим, что числовые типы .NET поддерживают свойства `MaxValue` и `MinValue`, которые предоставляют информацию о диапазоне значений, хранящихся в конкретном типе. Помимо свойств `MinValue` и `MaxValue`, каждый числовой тип может определять дополнительные полезные члены. Например, тип `System.Double` позволяет получать значения для эpsilon и бесконечности (представляющие интерес для тех, кто занимается решением математических задач). В целях иллюстрации рассмотрим следующую вспомогательную функцию:

```
static void DataTypeFunctionality()
{
    Console.WriteLine("=> Data type Functionality:");

    Console.WriteLine("Max of int: {0}", int.MaxValue);
    Console.WriteLine("Min of int: {0}", int.MinValue);
    Console.WriteLine("Max of double: {0}", double.MaxValue);
    Console.WriteLine("Min of double: {0}", double.MinValue);
    Console.WriteLine("double.Epsilon: {0}", double.Epsilon);
    Console.WriteLine("double.PositiveInfinity: {0}",
        double.PositiveInfinity);
    Console.WriteLine("double.NegativeInfinity: {0}",
        double.NegativeInfinity);
    Console.WriteLine();
}
```

Члены System.Boolean

Теперь взглянем на тип данных `System.Boolean`. Допустимыми значениями, которые могут присваиваться типу `bool` в C#, являются `true` и `false`. С учетом этого, должно быть очевидным, что `System.Boolean` не поддерживает свойства `MinValue` и `MaxValue`, а вместо них определяет свойства `TrueString` и `FalseString` (которые выдают, соответственно, строки `"True"` и `"False"`). Например:

```
Console.WriteLine("bool.FalseString: {0}", bool.FalseString);
Console.WriteLine("bool.TrueString: {0}", bool.TrueString);
```

Члены System.Char

Текстовые данные в C# представляются с помощью ключевых слов `string` и `char`, которые являются сокращенными обозначениями для типов `System.String` и `System.Char` (оба они основаны на Unicode). Как уже может быть известно, `string` представляет непрерывный набор символов (например, `"Hello"`), а `char` — одиночный символ в `string` (например, `'H'`).

Кроме возможности хранить одиночный элемент символьных данных, тип `System.Char` предлагает немало другой функциональности. Используя статические методы `System.Char`, можно выяснять, является данный символ цифрой, буквой, знаком пунктуации или чем-то еще. Рассмотрим следующий метод:

```
static void CharFunctionality()
{
    Console.WriteLine("=> char type Functionality:");
    char myChar = 'a';
    Console.WriteLine("char.IsDigit('a'): {0}", char.IsDigit(myChar));
    Console.WriteLine("char.IsLetter('a'): {0}", char.IsLetter(myChar));
    Console.WriteLine("char.IsWhiteSpace('Hello There', 5): {0}",
        char.IsWhiteSpace("Hello There", 5));
    Console.WriteLine("char.IsWhiteSpace('Hello There', 6): {0}",
        char.IsWhiteSpace("Hello There", 6));
    Console.WriteLine("char.IsPunctuation('?'): {0}",
        char.IsPunctuation('?'));
    Console.WriteLine();
}
```

Как было показано в этом методе, для многих членов `System.Char` предусмотрены два соглашения относительно вызова: одиночный символ или строка с числовым индексом, указывающим позицию проверяемого символа.

Синтаксический разбор значений из строковых данных

Типы данных .NET предоставляют возможность генерировать переменную заданного типа на основе текстового эквивалента (т.е. выполнять синтаксический разбор). Этот прием может оказаться исключительно полезным при преобразовании некоторых вводимых пользователем данных (например, результата выбора в раскрывающемся списке внутри графического пользовательского интерфейса) в числовые значения. Ниже приведен пример метода `ParseFromStrings()` с логикой синтаксического разбора:

```
static void ParseFromStrings()
{
    Console.WriteLine("=> Data type parsing:");
    bool b = bool.Parse("True");
    Console.WriteLine("Value of b: {0}", b);
    double d = double.Parse("99.884");
```



```

Console.WriteLine("Value of d: {0}", d);
int i = int.Parse("8");
Console.WriteLine("Value of i: {0}", i);
char c = Char.Parse("w");
Console.WriteLine("Value of c: {0}", c);
Console.WriteLine();
}

```

Типы System.DateTime и System.TimeSpan

В пространстве имен System определено несколько полезных типов данных, для которых в C# ключевые слова не предусмотрены. К ним относятся структуры DateTime и TimeSpan (а также показанные на рис. 3.2 типы System.Guid и System.Void, которые можете исследовать самостоятельно; эти два типа данных относительно редко встречаются в приложениях).

Тип DateTime содержит данные, представляющие конкретное значение даты (месяц, день, год) и времени, которые могут форматироваться различными способами с применением членов этого типа. Структура TimeSpan позволяет легко определять и преобразовывать единицы времени, используя различные ее члены.

```

static void UseDatesAndTimes()
{
    Console.WriteLine("=> Dates and Times:");

    // Этот конструктор принимает год, месяц и день.
    DateTime dt = new DateTime(2011, 10, 17);

    // Какой это день месяца?
    Console.WriteLine("The day of {0} is {1}", dt.Date, dt.DayOfWeek);

    // Сейчас месяц декабрь.
    dt = dt.AddMonths(2);
    Console.WriteLine("Daylight savings: {0}", dt.IsDaylightSavingTime());

    // Этот конструктор принимает часы, минуты и секунды.
    TimeSpan ts = new TimeSpan(4, 30, 0);
    Console.WriteLine(ts);

    // Вычесть 15 минут из текущего TimeSpan и вывести результат.
    Console.WriteLine(ts.Subtract(new TimeSpan(0, 15, 0)));
}

```

Сборка System.Numerics.dll

В пространстве имен System.Numerics определена структура под названием BigInteger. Тип данных BigInteger может использоваться для представления огромных числовых значений, которые не ограничены фиксированным верхним или нижним пределом.

На заметку! В пространстве имен System.Numerics определена вторая структура под названием Complex, которая позволяет моделировать математически сложные числовые данные (например, мнимые числа, вещественные данные, гиперболические тангенсы). Дополнительные сведения об этой структуре можно найти в документации .NET Framework 4.5 SDK.

Хотя во многих приложениях .NET потребность в использовании структуры BigInteger может никогда не возникать, если все-таки необходимо создать большое числовое значение, в первую очередь понадобится добавить в проект ссылку на сборку System.Numerics.dll в соответствии со следующими шагами.

Важно отметить, что тип данных `BigInteger` реагирует на внутренние математические операции C#, такие как `+`, `-` и `*`. Следовательно, вместо вызова метода `BigInteger.Multiply()` для перемножения двух больших чисел можно использовать такой код:

```
BigInteger reallyBig2 = biggy * reallyBig;
```

К этому моменту вы уже должны понимать, что ключевые слова C#, представляющие базовые типы данных, имеют соответствующие типы в библиотеках базовых классов .NET, каждый из которых предлагает фиксированную функциональность. Хотя каждый член этих типов данных в книге подробно не рассматривается, не помешает освоить их самостоятельно. Подробные описания всех типов данных .NET можно найти в документации .NET Framework 4.5 SDK.

Исходный код. Проект `BasicDataTypes` доступен в подкаталоге `Chapter 03`.

Работа со строковыми данными

Класс `System.String` предлагает множество методов, вполне ожидаемых в служебном классе подобного рода, включая методы для определения длины символьных данных, поиска подстрок в текущей строке и преобразования символов между верхним и нижним регистрами. В табл. 3.5 перечислены некоторые наиболее интересные члены этого класса.

Таблица 3.5. Избранные члены `System.String`

Член <code>System.String</code>	Описание
<code>Length</code>	Свойство, которое возвращает длину текущей строки
<code>Compare()</code>	Статический метод, который позволяет сравнить две строки
<code>Contains()</code>	Метод, который позволяет определить, содержится ли в строке определенная подстрока
<code>Equals()</code>	Метод, который позволяет проверить, содержатся ли в двух строковых объектах идентичные символьные данные
<code>Format()</code>	Статический метод, позволяющий сформатировать строку с использованием других элементарных типов данных (например, числовых данных или других строк) и нотации <code>{0}</code> , которая рассматривалась ранее в этой главе
<code>Insert()</code>	Метод, который позволяет вставить строку внутрь другой определенной строки
<code>PadLeft()</code> <code>PadRight()</code>	Методы, которые позволяют дополнить строку некоторыми символами
<code>Remove()</code> <code>Replace()</code>	Методы, которые позволяют получить копию строки с соответствующими изменениями (удалением или заменой символов)
<code>Split()</code>	Метод, который возвращает массив <code>string</code> , содержащий подстроки в этом экземпляре, которые разделяются элементами из указанного массива <code>char</code> или <code>string</code>
<code>Trim()</code>	Метод, который удаляет все вхождения набора указанных символов с начала и конца текущей строки
<code>ToUpper()</code> <code>ToLower()</code>	Методы, которые создают копию текущей строки в верхнем или нижнем регистре

Базовые манипуляции строками

Работа с членами `System.String` выглядит так, как и следовало ожидать. Просто объявите переменную `string` и пользуйтесь предоставляемой типом функциональностью через операцию точки. Имейте в виду, что несколько членов `System.String` являются статическими и потому должны вызываться на уровне класса (а не объекта). В целях иллюстрации создадим новый проект консольного приложения по имени `FunWithStrings` и добавим в него следующий метод, который будет вызываться внутри `Main()`:

```
static void BasicStringFunctionality()
{
    Console.WriteLine("=> Basic String functionality:");
    string firstName = "Freddy";
    // Значение firstName.
    Console.WriteLine("Value of firstName: {0}", firstName);
    // Длина firstName.
    Console.WriteLine("firstName has {0} characters.", firstName.Length);
    // firstName в верхнем регистре.
    Console.WriteLine("firstName in uppercase: {0}", firstName.ToUpper());
    // firstName в нижнем регистре.
    Console.WriteLine("firstName in lowercase: {0}", firstName.ToLower());
    // Содержит ли firstName букву y?
    Console.WriteLine("firstName contains the letter y?: {0}",
        firstName.Contains("y"));
    // firstName после замены.
    Console.WriteLine("firstName after replace: {0}", firstName.Replace("dy", ""));
    Console.WriteLine();
}
```

Здесь объяснять особо нечего: метод просто вызывает различные члены, такие как `ToUpper()` и `Contains()`, на локальной переменной `string` для получения разнообразных форматов и выполнения преобразований. Ниже показан вывод:

```
***** Fun with Strings *****
=> Basic String functionality:
Value of firstName: Freddy
firstName has 6 characters.
firstName in uppercase: FREDDY
firstName in lowercase: freddy
firstName contains the letter y?: True
firstName after replace: Fred
```

Хотя на первый взгляд здесь нет ничего необычного, вывод, полученный в результате вызова метода `Replace()`, может несколько запутать. В действительности переменная `firstName` вообще не изменяется; вместо этого мы получаем новую переменную `string` в модифицированном формате. Чуть позже мы еще вернемся к исследованию неизменяемой природы строк.

Конкатенация строк

Переменные `string` могут быть сцеплены вместе для создания строк большего размера с помощью операции `+` языка C#. Как известно, этот прием формально называется *конкатенацией строк*. Рассмотрим следующую вспомогательную функцию:

```
static void StringConcatenation()
{
    Console.WriteLine("=> String concatenation:");
    string s1 = "Programming the ";
    string s2 = "PsychoDrill (PTP)";
    string s3 = s1 + s2;
    Console.WriteLine(s3);
    Console.WriteLine();
}
```

Возможно, будет интересно узнать, что в результате обработки компилятором символа + в C# выдается вызов статического метода `String.Concat()`. Поэтому конкатенацию строк можно также осуществлять, вызывая метод `String.Concat()` напрямую (хотя в действительности это не дает каких-то преимуществ, а лишь увеличивает объем ввода):

```
static void StringConcatenation()
{
    Console.WriteLine("=> String concatenation:");
    string s1 = "Programming the ";
    string s2 = "PsychoDrill (PTP)";
    string s3 = String.Concat(s1, s2);
    Console.WriteLine(s3);
    Console.WriteLine();
}
```

Управляющие последовательности

Как и в других основанных на C языках, строковые литералы C# могут содержать различные управляющие последовательности, которые позволяют уточнять то, как символьные данные должны выводиться в выходной поток. Каждая управляющая последовательность начинается с символа обратной косой черты, за которым следует интерпретируемый знак. В табл. 3.6 перечислены часто используемые управляющие последовательности.

Таблица 3.6. Управляющие последовательности в строковых литералах

Управляющая последовательность	Описание
\'	Вставляет в строковый литерал символ одинарной кавычки
\"	Вставляет в строковый литерал символ двойной кавычки
\\	Вставляет в строковый литерал символ обратной косой черты. Особенно полезна при определении путей к файлам и сетевым ресурсам
\a	Заставляет систему выдавать звуковой сигнал, который в консольных приложениях может служить аудио-подсказкой пользователю
\n	Вставляет символ новой строки (на платформах Windows)
\r	Вставляет символ возврата каретки
\t	Вставляет в строковый литерал символ горизонтальной табуляции

Например, чтобы вывести строку, содержащую символ табуляции после каждого слова, можно воспользоваться управляющей последовательностью `\t`. Или предположим, что нужно создать один строковый литерал с символами кавычек внутри, второй — с определением пути к каталогу и третий — со вставкой трех пустых строк после вывода

символьных данных. Для этого можно применить управляющие последовательности `\`, `\\` и `\n`. Кроме того, ниже приведен еще один пример, в котором для привлечения внимания каждый строковый литерал снабжен звуковым сигналом:

```
static void EscapeChars()
{
    Console.WriteLine("=> Escape characters:\a");
    string strWithTabs = "Model\tColor\tSpeed\tPet Name\a ";
    Console.WriteLine(strWithTabs);

    Console.WriteLine("Everyone loves \"Hello World\\\"a ");
    Console.WriteLine("C:\\MyApp\\bin\\Debug\\a ");

    // Добавить 4 пустых строки и снова выдать звуковой сигнал.
    Console.WriteLine("All finished.\\n\\n\\n\\a ");
    Console.WriteLine();
}
```

Определение дословных строк

За счет добавления к строковому литералу префикса `@` можно создавать так называемые *дословные строки*. Дословные строки позволяют отключать обработку управляющих последовательностей в литералах и выводить значения `string` в том виде, в каком они есть. Эта возможность наиболее полезна при работе со строками, представляющими пути к каталогам и сетевым ресурсам. Таким образом, вместо использования управляющей последовательности `\\` можно написать следующий код:

```
// Следующая строка воспроизводится дословно,
// с отображением всех управляющих последовательностей.
Console.WriteLine(@"C:\MyApp\bin\Debug");
```

Также обратите внимание, что дословные строки могут применяться для предохранения пробелов в строках, разделенных на несколько строк вывода:

```
// При использовании дословных строк пробельные символы предохраняются.
string myLongString = @"This is a very
    very
        long string";
Console.WriteLine(myLongString);
```

Используя дословные строки, можно также напрямую вставлять в литералы символы двойной кавычки, просто дублируя лексему `"`:

```
Console.WriteLine(@"Cerebus said ""Darrrr! Pret-ty sun-sets""");
```

Строки и равенство

Как будет подробно объясняться в главе 4, *ссылочный тип* представляет собой объект, размещаемый в управляемой куче со сборкой мусора. По умолчанию при выполнении проверки на предмет равенства ссылочных типов (с помощью операций `==` и `!=` языка C#) значение `true` будет возвращаться в том случае, если обе ссылки указывают на один и тот же объект в памяти. Несмотря на то что в действительности `string` является ссылочным типом, операции равенства для него были переопределены так, чтобы можно было сравнивать значения объектов `string`, а не ссылки на объекты в памяти.

```
static void StringEquality()
{
    Console.WriteLine("=> String equality:");
    string s1 = "Hello!";
    string s2 = "Yo!";
```

```

Console.WriteLine("s1 = {0}", s1);
Console.WriteLine("s2 = {0}", s2);
Console.WriteLine();

// Проверить строки на предмет равенства.
Console.WriteLine("s1 == s2: {0}", s1 == s2);
Console.WriteLine("s1 == Hello!: {0}", s1 == "Hello!");
Console.WriteLine("s1 == HELLO!: {0}", s1 == "HELLO!");
Console.WriteLine("s1 == hello!: {0}", s1 == "hello!");
Console.WriteLine("s1.Equals(s2): {0}", s1.Equals(s2));
Console.WriteLine("Yo.Equals(s2): {0}", "Yo!".Equals(s2));
Console.WriteLine();
}

```

Операции равенства C# выполняют в отношении объектов `string` посимвольную проверку равенства с учетом регистра. Следовательно, строки "Hello!", "HELLO!" и "hello!" не равны между собой. Кроме того, памятуя о связи между `string` и `System.String`, проверку на предмет равенства можно выполнять с помощью метода `Equals()` класса `String` и других поддерживаемых этим классом операций равенства. И, наконец, поскольку каждый строковый литерал (такой как "Yo") является допустимым экземпляром `System.String`, доступ к функциональности, ориентированной на работу со строками, можно получать также для фиксированной последовательности символов.

Строки являются неизменяемыми

Один из интересных аспектов `System.String` связан с тем, что после присваивания объекту `string` первоначального значения символьные данные *не могут быть изменены*. На первый взгляд это может показаться противоречащим действительности, ведь строкам постоянно присваиваются новые значения, а в типе `System.String` доступен набор методов, которые, похоже, только то и делают, что позволяют изменять символьные данные тем или иным образом (например, преобразуя их в верхний или нижний регистр). Тем не менее, если присмотреться внимательнее к тому, что происходит "за кулисами", то можно будет заметить, что методы типа `string` на самом деле возвращают совершенно новый объект `string` в модифицированном виде:

```

static void StringsAreImmutable()
{
    // Установить первоначальное значение для строки.
    string s1 = "This is my string.";
    Console.WriteLine("s1 = {0}", s1);

    // Преобразование s1 в верхний регистр?
    string upperString = s1.ToUpper();
    Console.WriteLine("upperString = {0}", upperString);

    // Нет! s1 по-прежнему остается в том же формате!
    Console.WriteLine("s1 = {0}", s1);
}

```

Если внимательно взглянуть на приведенный ниже вывод, легко убедиться, что исходный объект `string` (`s1`) не был преобразован в верхний регистр, когда вызывался метод `ToUpper()`. В действительности была возвращена копия переменной типа `string` в измененном формате.

```

s1 = This is my string.
upperString = THIS IS MY STRING.
s1 = This is my string.

```

Тот же самый закон неизменяемости строк действует и при использовании в C# операции присваивания. В целях иллюстрации реализуем следующий метод `StringsAreImmutable2()`:

```
static void StringsAreImmutable2()
{
    string s2 = "My other string";
    s2 = "New string value";
}
```

Скомпилируем приложение и загрузим результирующую сборку в утилиту `ildasm.exe` (см. главу 1). Ниже показан CIL-код, который будет сгенерирован для метода `StringsAreImmutable2()`:

```
.method private hidebysig static void StringsAreImmutable2() cil managed
{
    // Code size      14 (0xe)
    .maxstack 1
    .locals init ([0] string s2)
    IL_0000: nop
    IL_0001: ldstr      "My other string"
    IL_0006: stloc.0
    IL_0007: ldstr      "New string value"
    IL_000c: stloc.0
    IL_000d: ret
} // end of method Program::StringAreImmutable2
```

Хотя низкоуровневые детали CIL пока подробно не рассматривались, обратите внимание на многочисленные вызовы кода операции `ldstr` (загрузка строки). Код операции `ldstr` в CIL загружает новый объект `string` в управляемую кучу. Предыдущий объект `string`, который содержал значение "My other string", будет в конечном итоге удален сборщиком мусора.

Так что же конкретно необходимо вынести из всего этого? Если кратко: класс `string` может стать неэффективным и приводить к "разбуханию" кода при неправильном использовании, особенно при выполнении конкатенации строк. Но когда необходимо представлять базовые символьные данные, такие как номер карточки социального страхования, имя и фамилия или простые фрагменты текста, используемые внутри приложения, он является идеальным вариантом.

Тем не менее, если строится приложение, в котором будут часто изменяться текстовые данные (например, текстовый процессор), то представление обрабатываемых текстовых данных с применением объектов `string` будет очень неудачным решением, поскольку это практически наверняка (и часто косвенно) приведет к созданию ненужных копий данных `string`. Как тогда должен поступить программист? Ответ на этот вопрос вы найдете ниже.

Тип `System.Text.StringBuilder`

С учетом того, что тип `string` может оказаться неэффективным при необдуманном использовании, библиотеки базовых классов .NET предоставляют пространство имен `System.Text`. Внутри этого (относительно небольшого) пространства имен находится класс под названием `StringBuilder`. Как и `System.String`, класс `StringBuilder` определяет методы, которые позволяют, к примеру, заменять или форматировать сегменты. Чтобы использовать этот класс в файлах кода C#, первым делом понадобится импортировать следующее пространство имен в файл кода (в случае нового проекта Visual Studio это уже должно быть сделано):


```
// Здесь определен класс StringBuilder:
using System.Text;
```

Уникальным в `StringBuilder` является то, что при вызове его членов производится непосредственное изменение внутренних символьных данных объекта (делая его более эффективным), без получения копии данных в модифицированном формате. При создании экземпляра `StringBuilder` начальные значения объекта могут быть заданы через один из множества конструкторов. Если вы не знакомы с понятием конструктора, то пока достаточно знать лишь то, что конструкторы позволяют создавать объект с начальным состоянием, когда применяется ключевое слово `new`. Взгляните на следующий пример использования `StringBuilder`:

```
static void FunWithStringBuilder()
{
    Console.WriteLine("=> Using the StringBuilder:");
    StringBuilder sb = new StringBuilder("**** Fantastic Games ****");
    sb.Append("\n");
    sb.AppendLine("Half Life");
    sb.AppendLine("Morrowind");
    sb.AppendLine("Deus Ex" + "2");
    sb.AppendLine("System Shock");
    Console.WriteLine(sb.ToString());
    sb.Replace("2", " Invisible War");
    Console.WriteLine(sb.ToString());
    Console.WriteLine("sb has {0} chars.", sb.Length);
    Console.WriteLine();
}
```

Здесь создается объект `StringBuilder` с начальным значением `"**** Fantastic Games ****"`. Как видите, можно добавлять строки в конец внутреннего буфера, а также заменять или удалять любые символы. По умолчанию `StringBuilder` способен хранить строку длиной не более 16 символов (но при необходимости будет автоматически расширяться), однако значение начальной длины можно изменить через дополнительный аргумент конструктора:

```
// Создать StringBuilder с исходным размером в 256 символов.
StringBuilder sb = new StringBuilder("**** Fantastic Games ****", 256);
```

Если добавляется больше символов, чем в указанном лимите, объект `StringBuilder` скопирует свои данные в новый экземпляр и увеличит размер буфера на заданный лимит.

Исходный код. Проект `FunWithStrings` доступен в подкаталоге `Chapter 03`.

Сужающие и расширяющие преобразования типов данных

Теперь, когда вы знаете, как работать с внутренними типами данных C#, давайте рассмотрим связанную с этим тему *преобразования типов данных*. Создадим новый проект консольного приложения по имени `TypeConversions` и определим в нем следующий класс:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with type conversions *****");
    }
}
```

```
// Добавить две переменных типа short и вывести результат.
short numb1 = 9, numb2 = 10;
Console.WriteLine("{0} + {1} = {2}",
    numb1, numb2, Add(numb1, numb2));
Console.ReadLine();
}

static int Add(int x, int y)
{
    return x + y;
}
}
```

Обратите внимание, что метод `Add()` ожидает двух параметров `int`. Тем не менее, в методе `Main()` ему на самом деле передаются две переменных `short`. Хотя это может выглядеть как несоответствие типов данных, программа компилируется и выполняется без ошибок, возвращая ожидаемый результат 19.

Причина, по которой компилятор считает этот код синтаксически корректным, связана с тем, что здесь потеря данных невозможна. Поскольку максимальное значение для типа `short` (32 767) гораздо меньше максимального значения диапазона для типа `int` (2 147 483 647), компилятор неявно *расширяет* каждое значение `short` до `int`. Формально термин *расширение* применяется для определения неявного *восходящего приведения*, которое не вызывает потери данных.

На заметку! Расширяющие и сужающие преобразования, поддерживаемые для каждого типа данных C#, описаны в разделе "Type Conversion Tables" ("Таблицы преобразования типов") документации .NET Framework 4.5 SDK.

Хотя такое неявное расширение типов было полезно в предыдущем примере, в других ситуациях оно может стать источником ошибок на этапе компиляции. Например, пусть для `numb1` и `numb2` установлены значения, которые (при сложении вместе) превышают максимальное значение `short`. Кроме того, предположим, что возвращаемое значение метода `Add()` сохраняется в новой локальной переменной `short`, а не напрямую выводится на консоль.

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with type conversions *****");

    // Следующий код вызовет ошибку компиляции!
    short numb1 = 30000, numb2 = 30000;
    short answer = Add(numb1, numb2);

    Console.WriteLine("{0} + {1} = {2}",
        numb1, numb2, answer);
    Console.ReadLine();
}
```

В этом случае компилятор сообщит об ошибке:

```
Cannot implicitly convert type 'int' to 'short'. An explicit conversion exists
(are you missing a cast?)
Не удастся неявно преобразовать тип int в short. Существует явное преобразование
(возможно, пропущено приведение)
```

Проблема в том, что хотя метод `Add()` способен вернуть значение `int`, равное 60000 (что входит в допустимый диапазон для `System.Int32`), оно не может быть сохранено в переменной `short`, т.к. выходит за пределы допустимого диапазона для типа `short`. Формально это означает, что среде CLR не удастся применить *сужающую опе-*

рацию. Как не трудно догадаться, сужающая операция представляет собой логическую противоположность расширяющей операции, поскольку предусматривает сохранение большего значения внутри переменной меньшего типа данных.

Важно отметить, что все сужающие преобразования приводят к ошибкам на этапе компиляции, даже когда есть веская причина полагать, что такое преобразование должно пройти успешно. Например, следующий код также вызовет ошибку при компиляции:

```
// Снова ошибка при компиляции!
static void NarrowingAttempt()
{
    byte myByte = 0;
    int myInt = 200;
    myByte = myInt;

    Console.WriteLine("Value of myByte: {0}", myByte);
}
```

Здесь значение, содержащееся в переменной типа `int` (по имени `myInt`), безопасно вписывается в диапазон допустимых значений для типа `byte`; следовательно, можно было бы ожидать, что сужающая операция не должна привести к ошибке во время выполнения. Однако из-за того, что язык C# создавался с расчетом на безопасность к типам, мы все-таки получим ошибку на этапе компиляции.

Если нужно информировать компилятор о том, что вы готовы мириться с возможной потерей данных в результате сужающей операции, следует применить *явное приведение*, используя операцию приведения `()` языка C#. Взгляните на показанную ниже модификацию класса `Program`:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with type conversions *****");
        short numb1 = 30000, numb2 = 30000;

        // Явно привести int к short (и разрешить потерю данных).
        short answer = (short)Add(numb1, numb2);

        Console.WriteLine("{0} + {1} = {2}",
            numb1, numb2, answer);
        NarrowingAttempt();
        Console.ReadLine();
    }

    static int Add(int x, int y)
    {
        return x + y;
    }

    static void NarrowingAttempt()
    {
        byte myByte = 0;
        int myInt = 200;

        // Явно привести int к byte (без потери данных).
        myByte = (byte)myInt;
        Console.WriteLine("Value of myByte: {0}", myByte);
    }
}
```

На этот раз компиляция кода проходит успешно, однако результат сложения совершенно не корректен:

```
***** Fun with type conversions *****
30000 + 30000 = -5536
Value of myByte: 200
```

Как было только что показано, явное приведение заставляет компилятор применить сужающее преобразование, даже когда оно может вызвать потерю данных. В случае метода `NarrowingAttempt()` это не было проблемой, т.к. значение 200 вписывалось в диапазон допустимых значений для типа `byte`. Тем не менее, в ситуации со сложением двух значений типа `short` внутри `Main()` конечный результат получился совершенно не приемлемым ($30\,000 + 30\,000 = -5536$?).

Для построения приложений, в которых потеря данных не допускается, в C# предоставляются ключевые слова `checked` и `unchecked`, которые позволяют гарантировать, что потеря данных не окажется незамеченной.

Ключевое слово `checked`

Начнем с выяснения роли ключевого слова `checked`. Давайте добавим в `Program` новый метод, суммирующий две переменных типа `byte`, каждой из которых присвоено значение, не превышающее максимум (255). По идее, после сложения значений этих двух переменных (с приведением результата `int` к типу `byte`) должна быть получена точная сумма.

```
static void ProcessBytes()
{
    byte b1 = 100;
    byte b2 = 250;
    byte sum = (byte)Add(b1, b2);

    // В sum должно содержаться значение 350. Однако там оказывается значение 94!
    Console.WriteLine("sum = {0}", sum);
}
```

Удивительно, но при просмотре вывода этого приложения обнаруживается, что в `sum` содержится значение 94 (а не 350, как ожидалось). Причина проста. Учитывая, что `System.Byte` может хранить только значение из диапазона от 0 до 255 включительно, в `sum` будет помещено значение переполнения ($350 - 256 = 94$). По умолчанию, если не предпринимаются никакие корректирующие действия, то условия переполнения и потери значимости происходят без выдачи сообщений об ошибках.

Для обработки условий переполнения и потери значимости в приложении доступны два способа. Это можно делать вручную, полагаясь на свои знания и навыки в области программирования. Недостаток такого подхода произрастает из того факта, что мы всего лишь люди, и даже приложив максимум усилий, мы можем попросту не заметить какие-то из ошибок.

К счастью, в C# предусмотрено ключевое слово `checked`. Когда оператор (или блок операторов) помещен в контекст `checked`, компилятор C# выдает дополнительные CIL-инструкции, обеспечивающие проверку условий переполнения, которые могут возникать при сложении, умножении, вычитании или делении двух числовых типов данных.

Если возникает переполнение, во время выполнения генерируется исключение `System.OverflowException`. В главе 7 будут представлены подробные сведения о структурированной обработке исключений и применении ключевых слов `try` и `catch`. Пока, не вдаваясь особо в детали, взгляните на следующий модифицированный код:

```
static void ProcessBytes()
{
    byte b1 = 100;
    byte b2 = 250;
```

```
// На этот раз сообщить компилятору о необходимости добавления
// CIL-кода, необходимого для генерации исключения, если возникает
// переполнение или потеря значимости.
try
{
    byte sum = checked((byte)Add(b1, b2));
    Console.WriteLine("sum = {0}", sum);
}
catch (OverflowException ex)
{
    Console.WriteLine(ex.Message);
}
}
```

Обратите внимание, что возвращаемое значение метода `Add()` помещено в контекст ключевого слова `checked`. Поскольку значение `sum` выходит за пределы допустимого диапазона для типа `byte`, генерируется исключение времени выполнения. Сообщение об ошибке выводится с использованием свойства `Message`:

Arithmetic operation resulted in an overflow.
 Арифметическая операция привела к переполнению.

Чтобы применить проверку переполнения к блоку операторов, контекст `checked` можно определить следующим образом:

```
try
{
    checked
    {
        byte sum = (byte)Add(b1, b2);
        Console.WriteLine("sum = {0}", sum);
    }
}
catch (OverflowException ex)
{
    Console.WriteLine(ex.Message);
}
```

В любом случае код будет автоматически проверяться на предмет возможных условий переполнения, и если они обнаружатся, будет сгенерировано исключение, связанное с переполнением.

Настройка проверки переполнения на уровне всего проекта

Если создается приложение, в котором переполнение никогда не должно оставаться незамеченным, может оказаться, что в контекст ключевого слова `checked` приходится помещать слишком много строк кода. В качестве альтернативы, компилятор C# поддерживает флаг `/checked`. Когда этот флаг указан, все присутствующие в коде арифметические операции будут проверяться на предмет переполнения, не требуя использования ключевого слова `checked`. Если переполнение обнаруживается, генерируется исключение времени выполнения.

Для активизации этого флага в Visual Studio откройте страницу свойств проекта, перейдите на вкладку **Build** (Сборка) и щелкните на кнопке **Advanced** (Дополнительно).

В открывшемся диалоговом окне отметьте флажок **Check for arithmetic overflow/underflow** (Проверять арифметическое переполнение и потерю значимости), как показано на рис. 3.3.

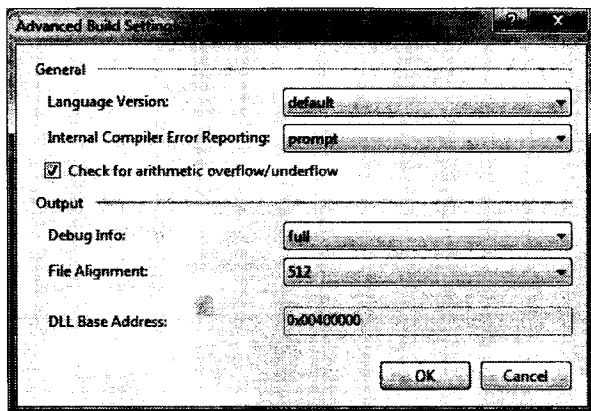


Рис. 3.3. Включение проверки переполнения и потери значимости в масштабах всего проекта

Включение этой настройки может быть очень удобным при создании отладочной версии сборки. После устранения всех условий переполнения из кодовой базы флаг `/checked` можно отключить для последующих сборок (это приведет к увеличению производительности приложения).

Ключевое слово `unchecked`

А теперь предположим, что проверка переполнения и потери значимости включена в масштабах всего проекта, но имеется блок кода, в котором потеря данных приемлема. Учитывая, что действие флага `/checked` распространяется на всю арифметическую логику, в C# предусмотрено ключевое слово `unchecked`, которое предназначено для отмены генерации исключений, связанных с переполнением, в отдельных случаях. Это ключевое слово используется аналогично ключевому слову `checked`, т.е. его можно применять как к единственному оператору, так и к блоку операторов.

```
// Предполагая, что флаг /checked активизирован,
// этот блок не будет генерировать исключение
// времени выполнения.
unchecked
{
    byte sum = (byte) (b1 + b2);
    Console.WriteLine("sum = {0} ", sum);
}
```

Итак, подводя итоги по ключевым словам `checked` и `unchecked` в C#, отметим, что стандартное поведение исполняющей среды .NET предусматривает игнорирование арифметического переполнения и потери значимости. Когда необходимо обработать отдельные операторы, должно использоваться ключевое слово `checked`. Если нужно перехватывать ошибки переполнения по всему приложению, понадобится активизировать флаг `/checked`. Наконец, ключевое слово `unchecked` может применяться, когда есть блок кода, в котором переполнение является приемлемым (и, таким образом, не должно приводить к генерации исключения времени выполнения).

Понятие неявно типизированных локальных переменных

До этого места в главе при объявлении каждой локальной переменной явно указывался ее тип данных:

```
static void DeclareExplicitVars()
{
    // Явно типизированные локальные переменные
    // объявляются следующим образом:
    // типДанных имяПеременной = начальноеЗначение;
    int myInt = 0;
    bool myBool = true;
    string myString = "Time, marches on...";
}
```

Хотя многие (в том числе и автор) согласятся с тем, что явное указание типа данных для каждой переменной является рекомендуемой практикой, язык C# поддерживает возможность *неявной типизации* локальных переменных с использованием ключевого слова `var`. Ключевое слово `var` может применяться вместо указания конкретного типа данных (такого как `int`, `bool` или `string`). В этом случае компилятор автоматически выведет лежащий в основе тип данных на основе начального значения, используемого для инициализации локального элемента данных.

Чтобы продемонстрировать роль неявной типизации, создадим новый проект консольного приложения по имени `ImplicitlyTypedLocalVars`. Обратите внимание, что локальные переменные, которые присутствовали в предыдущем методе, теперь объявлены следующим образом:

```
static void DeclareImplicitVars()
{
    // Неявно типизированные локальные переменные
    // объявляются следующим образом:
    // var имяПеременной = начальноеЗначение;
    var myInt = 0;
    var myBool = true;
    var myString = "Time, marches on...";
}
```

На заметку! Строго говоря, `var` не является ключевым словом C#. Вполне допустимо объявлять переменные, параметры и поля с именами `var`, не получая ошибок на этапе компиляции. Однако когда лексема `var` используется в качестве типа данных, то в таком контексте она трактуется компилятором как ключевое слово.

В этом случае компилятор, располагая первоначально присвоенными значениями, может вывести для переменной `myInt` тип `System.Int32`, для переменной `myBool` — тип `System.Boolean`, а для переменной `myString` — тип `System.String`. В этом легко удостовериться с помощью *рефлексии*. Как будет показано в главе 15, рефлексия — это процесс определения состава типа во время выполнения. Например, используя рефлексия, можно определить тип данных неявно типизированной локальной переменной. Модифицируем метод `DeclareImplicitVars()` следующим образом:

```
static void DeclareImplicitVars()
{
    // Неявно типизированные локальные переменные.
    var myInt = 0;
    var myBool = true;
    var myString = "Time, marches on...";
}
```

```
// Вывести имена лежащих в основе типов.
Console.WriteLine("myInt is a: {0}", myInt.GetType().Name);
Console.WriteLine("myBool is a: {0}", myBool.GetType().Name);
Console.WriteLine("myString is a: {0}", myString.GetType().Name);
}
```

На заметку! Имейте в виду, что такую неявную типизацию можно использовать для любых типов, включая массивы, обобщенные типы (см. главу 9) и собственные нестандартные типы. Далее в книге будут встречаться другие примеры неявной типизации.

Если вызвать метод `DeclareImplicitVars()` внутри `Main()`, будет получен показанный ниже вывод:

```
***** Fun with Implicit Typing *****

myInt is a: Int32
myBool is a: Boolean
myString is a: String
```

Ограничения неявно типизированных переменных

С использованием ключевого слова `var` связаны разнообразные ограничения. Прежде всего, неявная типизация применима *только* к локальным переменным внутри контекста метода или свойства. Применять ключевое слово `var` для определения возвращаемых значений, параметров или данных полей в специальном типе не допускается. Например, следующее определение класса приведет к выдаче различных сообщений об ошибках на этапе компиляции:

```
class ThisWillNeverCompile
{
    // Ошибка! var не может применяться к полям!
    private var myInt = 10;
    // Ошибка! var не может применяться к возвращаемому значению
    // или типу параметра!
    public var MyMethod(var x, var y){}
}
```

Кроме того, локальным переменным, объявленным с помощью ключевого слова `var`, должно быть присвоено начальное значение в самом объявлении, при этом присваивать `null` в качестве начального значения не допускается. Последнее ограничение должно быть понятным, т.к. на основании только `null` компилятор не сможет вывести тип, на значение которого указывает переменная.

```
// Ошибка! Должно быть присвоено значение!
var myData;

// Ошибка! Значение должно присваиваться в самом объявлении!
var myInt;
myInt = 0;

// Ошибка! Нельзя присваивать null в качестве начального значения!
var myObj = null;
```

Однако присваивать `null` разрешается локальной переменной с выведенным после начального присваивания типом (при условии, что она относится к ссылочному типу):

```
// Допустимо, если SportsCar является переменной ссылочного типа!
var myCar = new SportsCar();
myCar = null;
```

Вдобавок, значение неявно типизированной локальной переменной можно присваивать другим переменным, как неявно, так и явно типизированным:


```
// Также допустимо!
var myInt = 0;
var anotherInt = myInt;
string myString = "Wake up!";
var myData = myString;
```

Кроме того, неявно типизированную локальную переменную можно возвращать вызывающему коду, при условии, что возвращаемый тип метода совпадает с выведенным типом переменной, определенной с помощью `var`:

```
static int GetAnInt()
{
    var retVal = 9;
    return retVal;
}
```

Неявно типизированные данные являются строго типизированными

Следует иметь в виду, что неявная типизация локальных переменных дает в результате *строго типизированные данные*. Таким образом, применение ключевого слова `var` в языке C# отличается от техники, используемой в языках сценариев (таких как JavaScript или Perl), а также от типа данных `Variant` в COM, где переменная на протяжении своего времени жизни может хранить значения разных типов (это часто называют *динамической типизацией*).

На заметку! В C# существует возможность динамической типизации с использованием ключевого слова `dynamic`. Более подробно об этом аспекте языка будет рассказываться в главе 16.

Вместо этого выводение типа сохраняет аспект строгой типизации языка C# и влияет только на объявление переменных на этапе компиляции. После этого данные трактуются как объявленные с выведенным типом; присваивание такой переменной значения другого типа будет приводить к ошибке при компиляции.

```
static void ImplicitTypingIsStrongTyping()
{
    // Компилятор знает, что s имеет тип System.String.
    var s = "This variable can only hold string data!";
    s = "This is fine...";

    // Можно вызывать любой член лежащего в основе типа.
    string upper = s.ToUpper();

    // Ошибка! Присваивание числовых данных строке не допускается!
    s = 44;
}
```

Польза от неявно типизированных локальных переменных

Теперь, когда вы видели синтаксис, используемый для объявления неявно типизируемых локальных переменных, вас наверняка интересует, в каких ситуациях его следует применять? Прежде всего, использование `var` для объявления локальных переменных просто ради интереса особой пользы не принесет. Такой подход может вызвать путаницу у тех, кто будет изучать код, поскольку лишает возможности быстро определить тип данных и, следовательно, затрудняет выяснение предназначения переменной. Поэтому если точно известно, что переменная должна относиться к типу `int`, то лучше сразу и объявить ее как `int`.

Однако, как будет показано в начале главы 12, в наборе технологий LINQ используются выражения запросов, которые могут выдавать динамически создаваемые резуль-

тирующие наборы, основанные на формате самого запроса. В таких случаях неявная типизация исключительно удобна, поскольку не требуется явно определять тип, который запрос может вернуть, что в ряде ситуаций вообще невозможно. Взгляните, сможете ли вы определить базовый тип данных `subset` в следующем коде:

```
static void LinqQueryOverInts()
{
    int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };

    // Запрос LINQ!
    var subset = from i in numbers where i < 10 select i;

    Console.WriteLine("Values in subset: ");
    foreach (var i in subset)
    {
        Console.WriteLine("{0} ", i);
    }
    Console.WriteLine();

    // К какому типу относится subset?
    Console.WriteLine("subset is a: {0}", subset.GetType().Name);
    Console.WriteLine("subset is defined in: {0}", subset.GetType().Namespace);
}
```

Можно предположить, что типом данных `subset` является массив целочисленных значений. Именно так он выглядит внешне, но на самом деле он представляет собой низкоуровневый тип данных LINQ, о котором вы вряд ли что-то знаете, если только не работаете с LINQ очень долгое время или не откроете скомпилированный образ в утилите `ildasm.exe`. Хорошая новость состоит в том, что при использовании LINQ редко приходится (если вообще приходится) заботиться о типе возвращаемого значения запроса — достаточно просто присвоить значение неявно типизированной локальной переменной.

Фактически можно даже утверждать, что *единственным случаем*, когда применение ключевого слова `var` полностью оправдано, является определение данных, возвращаемых из запроса LINQ. Запомните: если вы знаете, что нужна переменная `int`, то просто объявляйте ее как `int`. Злоупотребление неявной типизацией (через ключевое слово `var`) считается плохим стилем для использования в производственном коде.

Исходный код. Проект `ImplicitlyTypedLocalVars` доступен в подкаталоге `Chapter 03`.

Итерационные конструкции C#

Все языки программирования предлагают способы для повторения блоков кода до тех пор, пока не будет удовлетворено условие завершения. Каким бы языком вы не пользовались в прошлом, итерационные операторы C# не должны вызывать недоумения или требовать особо подробных объяснений. В C# предоставляются четыре итерационных конструкции:

- цикл `for`;
- цикл `foreach/in`;
- цикл `while`;
- цикл `do/while`.

Давайте кратко рассмотрим каждую из этих конструкций по очереди, используя новый проект консольного приложения по имени `IterationsAndDecisions`.

На заметку! В этом заключительном разделе предполагается наличие у вас опыта работы с аналогичными ключевыми словами (if, for, switch и т.д.) в других языках программирования. Если требуется дополнительная информация, просмотрите темы "Iteration Statements (C# Reference)" ("Операторы итерации (Справочник по C#)"), "Jump Statements (C# Reference)" ("Операторы перехода (Справочник по C#)") и "Selection Statements (C# Reference)" ("Операторы выбора (Справочник по C#)") в документации .NET Framework 4.5 SDK.

Цикл for

Если требуется повторять блок кода фиксированное количество раз, хороший уровень гибкости предлагает оператор for. Он позволяет указать, сколько раз должен повторяться блок кода, а также задать условие завершения. Ниже приведен пример применения этого оператора:

```
// Простой цикл for.
static void ForLoopExample()
{
    // Обратите внимание, что переменная i является видимой только
    // в контексте этого цикла for.
    for(int i = 0; i < 4; i++)
    {
        Console.WriteLine("Number is: {0} ", i);
    }
    // Здесь переменная i уже не видима.
}
```

Все приемы, освоенные в языках C, C++ и Java, применимы также при создании операторов for в C#. Можно создавать сложные условия завершения, строить бесконечные циклы, циклы в обратном направлении (с помощью операции --) и использовать ключевые слова для переходов goto, continue и break.

Цикл foreach

Ключевое слово foreach в C# позволяет проходить в цикле по всем элементам контейнера, не требуя проверки его верхнего предела. Однако в отличие от цикла for, цикл foreach будет выполнять проход по контейнеру только линейным (n+1) образом (т.е. нельзя выполнять проход по контейнеру в обратном направлении, пропускать каждый третий элемент и т.п.).

Тем не менее, если требуется просто выполнить проход по коллекции элемент за элементом, цикл foreach — именно то, что нужно. Ниже приведены два примера использования цикла foreach — один для обхода массива строк и один для обхода массива целых чисел. Обратите внимание, что тип данных, указанный перед ключевым словом in, представляет тип данных контейнера.

```
// Проход по элементам массива с помощью foreach.
static void ForEachLoopExample()
{
    string[] carTypes = { "Ford", "BMW", "Yugo", "Honda" };
    foreach (string c in carTypes)
        Console.WriteLine(c);
    int[] myInts = { 10, 20, 30, 40 };
    foreach (int i in myInts)
        Console.WriteLine(i);
}
```

Элементом, который следует за ключевым словом in, может быть простой массив (как в приведенном примере) или, точнее говоря, любой класс, который реализует ин-

терфейс `IEnumerable`. Как будет показано в главе 9, библиотеки базовых классов .NET поставляются с рядом коллекций, содержащих реализации общих абстрактных типов данных. Любой из этих элементов (такой как обобщенный `List<T>`) может использоваться внутри цикла `foreach`.

Использование неявной типизации в конструкциях `foreach`

В итерационных конструкциях `foreach` также допускается применять неявную типизацию. Как и можно было догадаться, компилятор будет корректно выводить соответствующий "тип типа". Вспомните пример метода `Linq`, приведенный ранее в этой главе. Даже не зная точного типа данных переменной `subset`, все же можно выполнить итерацию по результирующему набору, используя неявную типизацию:

```
static void LinqQueryOverInts()
{
    int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };
    // Запрос LINQ!
    var subset = from i in numbers where i < 10 select i;
    Console.WriteLine("Values in subset: ");
    foreach (var i in subset)
    {
        Console.WriteLine("{0} ", i);
    }
}
```

Циклы `while` и `do/while`

Итерационную конструкцию `while` удобно применять, когда требуется, чтобы блок операторов выполнялся до тех пор, пока не будет удовлетворено условие завершения. Внутри контекста цикла `while` необходимо позаботиться о том, чтобы это условие действительно удовлетворялось, иначе получится бесконечный цикл. Ниже приведен пример, в котором на консоль выводится сообщение "In while loop" до тех пор, пока пользователь не завершит цикл вводом `yes` в командной строке:

```
static void WhileLoopExample()
{
    string userIsDone = "";
    // Проверить строку в нижнем регистре.
    while(userIsDone.ToLower() != "yes")
    {
        Console.WriteLine("In while loop");
        Console.WriteLine("Are you done? [yes] [no]: ");
        userIsDone = Console.ReadLine();
    }
}
```

Тесно связанным с `while` является цикл `do/while`. Подобно простому `while`, цикл `do/while` применяется тогда, когда какое-то действие должно выполняться неопределенное количество раз. Разница между этими двумя циклами состоит в том, что цикл `do/while` гарантирует выполнение соответствующего блока кода хотя бы один раз, в то время как цикл `while` может его вообще не выполнить, если условие с самого начала оказывается ложным.

```
static void DoWhileLoopExample()
{
    string userIsDone = "";
    do
    {
```

```
    Console.WriteLine("In do/while loop");
    Console.Write("Are you done? [yes] [no]: ");
    userIsDone = Console.ReadLine();
} while(userIsDone.ToLower() != "yes"); // Обратите внимание на точку с запятой!
}
```

Конструкции принятия решений и операции равенства/сравнения

Теперь, когда вы знаете, как многократно выполнять блок операторов, давайте рассмотрим следующую связанную концепцию — управление потоком выполнения программы. Для изменения хода выполнения программы в C# предлагаются две следующих конструкции:

- оператор if/else;
- оператор switch.

Оператор if/else

Сначала рассмотрим оператор if/else. В отличие от C и C++, в языке C# этот оператор может работать только с булевыми выражениями, но не с произвольными значениями наподобие -1 и 0.

Операции равенства и сравнения

Как правило, для получения литеральных булевских значений в операторах if/else обычно применяются операции, перечисленные в табл. 3.7.

Таблица 3.7. Операции равенства и сравнения в C#

Операция равенства/сравнения	Пример использования	Описание
==	if (age == 30)	Возвращает true, только если выражения одинаковы
!=	if ("Foo" != myStr)	Возвращает true, только если выражения разные
<	if (bonus < 2000)	Возвращает true, если выражение слева (bonus) меньше, больше, меньше или равно либо больше или равно выражению справа (2000)
>	if (bonus > 2000)	
<=	if (bonus <= 2000)	
>=	if (bonus >= 2000)	

Программисты, ранее работавшие с C и C++, должны иметь в виду, что старые приемы проверки неравенства значения нулю в C# работать не будут. Например, предположим, что требуется проверить, состоит ли текущая строка из более чем нуля символов. У программистов на C и C++ может возникнуть соблазн написать код следующего вида:

```
static void IfElseExample()
{
    // Такой код не допускается, т.к. Length возвращает int, а не bool.
    string stringData = "My textual data";
    if (stringData.Length)
    {
        Console.WriteLine("string is greater than 0 characters");
    }
}
```

Если необходимо использовать свойство `String.Length` для определения истинности или ложности, вычисляемое в условии выражение потребуется изменить так, чтобы оно давало в результате булево значение:

```
// Такой код допустим, поскольку условие возвращает true или false.
if(stringData.Length > 0)
{
    Console.WriteLine("string is greater than 0 characters");
}
```

Условные операции

В операторе `if` могут также применяться сложные выражения, и он может содержать операторы `else`, обеспечивая выполнение более сложных проверок. Его синтаксис идентичен аналогам в C (C++) и Java. Для построения сложных выражений в C# используется вполне ожидаемый набор условных операций, описанный в табл. 3.8.

Таблица 3.8. Условные операции C#

Операция	Пример	Описание
<code>&&</code>	<code>if (age == 30 && name == "Fred")</code>	Операция "И". Возвращает <code>true</code> , если все выражения дают <code>true</code>
<code> </code>	<code>if (age == 30 name == "Fred")</code>	Операция "ИЛИ". Возвращает <code>true</code> , если хотя бы одно из выражений дает <code>true</code>
<code>!</code>	<code>if (!myBool)</code>	Операция "НЕ". Возвращает <code>true</code> , если выражение дает <code>false</code> , или <code>false</code> , если выражение дает <code>true</code>

На заметку! Операции `&&` и `||` при необходимости поддерживают сокращенный путь выполнения. Это значит, что после определения, что сложное выражение должно быть `false`, оставшиеся подвыражения вычисляться не будут. Если требуется, чтобы все выражения проверялись в любом случае, можно использовать операции `&` и `|`.

Оператор switch

Еще одной простой конструкцией в C#, предназначенной для реализации выбора, является оператор `switch`. Как и в остальных основанных на C языках, оператор `switch` позволяет организовать выполнение программы на основе заранее определенного набора вариантов. Например, в приведенном ниже методе `Main()` для каждого из двух возможных вариантов выводится свое сообщение (блок `default` обрабатывает недопустимый выбор).

```
// Переход на основе числового значения.
static void SwitchExample()
{
    Console.WriteLine("1 [C#], 2 [VB]");
    Console.Write("Please pick your language preference: ");
    // Выберите предпочитаемый язык:
    string langChoice = Console.ReadLine();
    int n = int.Parse(langChoice);
    switch (n)
    {
        case 1:
            Console.WriteLine("Good choice, C# is a fine language.");
            // Хороший выбор. C# – замечательный язык.
```

```

break;
case 2:
    Console.WriteLine("VB: OOP, multithreading, and more!");
    // VB: ООП, многопоточность и многое другое!

break;
default:
    Console.WriteLine("Well...good luck with that!");
    // Хорошо, удачи!

break;
}
}

```

На заметку! Язык C# требует, чтобы каждый блок case (включая default), который содержит выполняемые операторы, завершался оператором break или goto во избежание сквозного прохода по блокам.

Одна из замечательных особенностей оператора switch в C# заключается в том, что помимо числовых значений он также позволяет производить вычисления со строковыми данными. Ниже для примера приведена модифицированная версия предыдущего оператора switch (обратите внимание, что при этом синтаксический разбор пользовательских данных в числовые значения не требуется).

```

static void SwitchOnStringExample()
{
    Console.WriteLine("C# or VB");
    Console.Write("Please pick your language preference: ");
    string langChoice = Console.ReadLine();
    switch (langChoice)
    {
        case "C#":
            Console.WriteLine("Good choice, C# is a fine language.");
            break;
        case "VB":
            Console.WriteLine("VB: OOP, multithreading and more!");
            break;
        default:
            Console.WriteLine("Well...good luck with that!");
            break;
    }
}

```

Оператор switch можно также применять к перечислимым типам данных. Как будет показано в главе 4, ключевое слово enum языка C# позволяет определять специальный набор пар "имя/значение". В качестве иллюстрации рассмотрим следующую вспомогательную функцию, которая выполняет проверку switch для перечисления System.DayOfWeek. Этот пример содержит ряд синтаксических конструкций, которые мы еще не рассматривали, но пока обращайтесь внимание только на enum; недостающие сведения вы получите в последующих главах.

```

static void SwitchOnEnumExample()
{
    Console.Write("Enter your favorite day of the week: ");
    DayOfWeek favDay;
    try
    {
        favDay = (DayOfWeek)Enum.Parse(typeof(DayOfWeek), Console.ReadLine());
    }
}

```

```

catch (Exception)
{
    Console.WriteLine("Bad input!");
    return;
}
switch (favDay)
{
    case DayOfWeek.Friday:
        Console.WriteLine("Yes, Friday rules!");
        break;
    case DayOfWeek.Monday:
        Console.WriteLine("Another day, another dollar");
        break;
    case DayOfWeek.Saturday:
        Console.WriteLine("Great day indeed.");
        break;
    case DayOfWeek.Sunday:
        Console.WriteLine("Football!!");
        break;
    case DayOfWeek.Thursday:
        Console.WriteLine("Almost Friday...");
        break;
    case DayOfWeek.Tuesday:
        Console.WriteLine("At least it is not Monday");
        break;
    case DayOfWeek.Wednesday:
        Console.WriteLine("A fine day.");
        break;
}
}

```

Исходный код. Проект `IterationsAndDecisions` доступен в подкаталоге `Chapter 03`.

Резюме

Цель этой главы заключалась в представлении многочисленных ключевых аспектов языка программирования C#. Сначала рассматривались типичные конструкции, используемые в любом приложении. Затем была описана роль объекта приложения и рассказано о том, что каждая исполняемая программа на C# должна иметь тип, определяющий метод `Main()`, который служит точкой входа. Внутри метода `Main()` обычно создается любое количество объектов, которые, работая вместе, приводят приложение в действие.

Далее были детально описаны базовые типы данных в C# и разъяснено, что используемые для их представления ключевые слова (например, `int`) на самом деле являются сокращенными обозначениями полноценных типов из пространства имен `System` (в этом случае `System.Int32`). Благодаря этому, каждый тип данных в C# имеет набор встроенных членов. Кроме того, была описана роль расширения и сужения, а также ключевых слов `checked` и `unchecked`.

В завершение главы были продемонстрированы роль неявной типизации с применением ключевого слова `var`. Как было отмечено, неявная типизация наиболее полезна в модели программирования `LINQ`. И, наконец, в главе кратко рассматривались различные конструкции C#, предназначенные для организации циклов и принятия решений.

Теперь, когда известны базовые характеристики языка C#, можно переходить к изучению ключевых функциональных средств языка (глава 4), а также его объектно-ориентированных возможностей (глава 5).

Главные конструкции программирования на C#: часть II

В этой главе завершается начатый в главе 3 обзор ключевых аспектов языка программирования C#. Здесь будут рассматриваться различные детали, касающиеся построения методов, в частности, ключевые слова `out`, `ref` и `params`. Вдобавок вы узнаете о роли необязательных и именованных параметров.

После ознакомления с концепцией *перегрузки методов* вы научитесь манипулировать массивами с использованием синтаксиса C# и исследуете функциональность связанного с массивами класса `System.Array`.

В главе также будет показано, как создавать типы перечислений и структур, и подробно описаны отличия между *типами значений* и *ссылочными типами*. И, наконец, будет описана роль типов данных, допускающих `null`, и операций `?` и `??`. После освоения материалов настоящей главы можно смело переходить к изучению объектно-ориентированных возможностей языка C#, которое начинается в главе 5.

Методы и модификаторы параметров

Для начала давайте исследуем детали, связанные с определением методов. Как и метод `Main()` (см. главу 3), специальные методы могут принимать или не принимать параметры, а также возвращать или не возвращать значения вызывающему коду. В следующих главах будет показано, что методы могут быть реализованы внутри контекста классов или структур (а также прототипированы внутри интерфейсных типов) и декорироваться различными ключевыми словами (например, `static`, `virtual`, `public`, `new`) для уточнения их поведения. До настоящего момента в этой книге каждый из рассматриваемых методов следовал такому базовому формату:

```
// Вспомните, что статические методы могут вызываться
// напрямую без создания экземпляра класса.
class Program
{
    // static возвращаемыйТип ИмяМетода (список параметров) { /* Реализация */ }
    static int Add(int x, int y){ return x + y; }
}
```

Хотя определение метода в C# выглядит довольно понятно, существуют несколько ключевых слов, с помощью которых можно управлять способом передачи аргументов интересующему методу. Все эти ключевые слова описаны в табл. 4.1.

Таблица 4.1. Модификаторы параметров в C#

Модификатор параметра	Описание
(отсутствует)	Если параметр не помечен модификатором, предполагается, что он должен передаваться по значению, т.е. вызываемый метод получает копию исходных данных
out	Значения выходных параметров должны присваиваться вызываемым методом и, следовательно, они передаются по ссылке. Если выходным параметрам в вызываемом методе значения не присвоены, компилятор сообщит об ошибке
ref	Значение первоначально присваивается вызывающим кодом и при желании может быть изменено в вызываемом методе (поскольку данные также передаются по ссылке). Если параметру ref в вызываемом методе значение не присвоено, никакой ошибки компилятор не генерирует
params	Этот модификатор позволяет передавать переменное количество аргументов как единый логический параметр. Метод может иметь только один модификатор params, которым должен быть помечен последний параметр метода. В реальности необходимость в использовании модификатора params возникает не особо часто, однако он применяется во многих методах внутри библиотек базовых классов

Для иллюстрации использования этих ключевых слов мы создадим новый проект консольного приложения по имени FunWithMethods. А теперь давайте рассмотрим роль каждого ключевого слова.

Стандартное поведение передачи параметров по значению

По умолчанию параметр передается *по значению*. Другими словами, если аргумент не помечен модификатором параметра, в функцию передается копия данных. Как будет объясняться в конце этой главы, точное *содержимое* копии зависит от того, к какому типу относится параметр — типу значения или ссылочному типу. Пока что предположим, что внутри класса Program имеется следующий метод, который оперирует на двух числовых типах данных, передаваемых по значению:

```
// По умолчанию аргументы передаются по значению.
static int Add(int x, int y)
{
    int ans = x + y;

    // Вызывающий код не увидит эти изменения,
    // т.к. изменяется копия исходных данных.
    x = 10000;
    y = 88888;
    return ans;
}
```

Числовые данные относятся к категории *типов значений*. Следовательно, если вы измените значения параметров внутри контекста члена, вызывающий код будет оставаться в полном неведении об этом, поскольку изменения происходят только в *копии* исходных данных:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Methods *****\n");

    // Передать две переменных по значению.
    int x = 9, y = 10;
    Console.WriteLine("Before call: X: {0}, Y: {1}", x, y);
    Console.WriteLine("Answer is: {0}", Add(x, y));
}
```

```

    Console.WriteLine("After call: X: {0}, Y: {1}", x, y);
    Console.ReadLine();
}

```

Как и следовало ожидать, значения *x* и *y* остаются теми же самыми до и после вызова `Add()`, что видно в приведенном ниже выводе; это объясняется передачей элементов данных по значению. Таким образом, любые изменения этих параметров, производимые внутри метода `Add()`, не видны вызывающему коду, потому что метод `Add()` оперирует на копии данных.

```

***** Fun with Methods *****

Before call: X: 9, Y: 10
Answer is: 19
After call: X: 9, Y: 10

```

Модификатор `out`

Теперь посмотрим, как используются *выходные параметры*. Методы, которым при определении указано принимать выходные параметры (с помощью ключевого слова `out`), должны перед выходом обязательно присваивать им соответствующие значения (в противном случае компилятор сообщит об ошибке).

В целях иллюстрации ниже приведена альтернативная версия метода `Add()`, возвращающая сумму двух целых чисел с применением модификатора `out` (обратите внимание, что возвращаемым значением метода теперь является `void`):

```

// Значения выходных параметров должны быть установлены вызываемым методом.
static void Add(int x, int y, out int ans)
{
    ans = x + y;
}

```

Вызов метода с выходными параметрами также требует использования модификатора `out`. Однако локальные переменные, которые передаются в качестве выходных параметров, не требуют предварительной установки значений (после вызова эти значения все равно будут утеряны). Причина, по которой компилятор позволяет передавать на первый взгляд неинициализированные данные, связана с тем, что вызываемый метод должен выполнить присваивание. Рассмотрим пример:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Methods *****");
    ...
    // Присваивать начальные значения локальным переменным,
    // используемым как выходные параметры, не обязательно,
    // при условии, что в таком качестве они используются первый раз.
    int ans;
    Add(90, 90, out ans);
    Console.WriteLine("90 + 90 = {0}", ans);
    Console.ReadLine();
}

```

Этот пример представлен исключительно для иллюстративных целей; на самом деле нет никаких причин возвращать значение суммы через выходной параметр. Тем не менее, модификатор `out` в C# действительно очень полезен: он позволяет вызывающему коду получать из единственного вызова метода множество значений.

```

// Возврат множества выходных параметров.
static void FillTheseValues(out int a, out string b, out bool c)
{

```

```

a = 9;
b = "Enjoy your string.";
c = true;
}

```

Теперь вызывающий код может обратиться к методу `FillTheseValues()`, как показано ниже. Вспомните, что модификатор `out` должен указываться как при вызове, так и при реализации данного метода:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Methods *****");
    ...
    int i; string str; bool b;
    FillTheseValues(out i, out str, out b);

    Console.WriteLine("Int is: {0}", i);
    Console.WriteLine("String is: {0}", str);
    Console.WriteLine("Boolean is: {0}", b);
    Console.ReadLine();
}

```

И, наконец, не забывайте, что перед выходом из метода, определяющего выходные параметры, этим параметрам *должны* быть присвоены допустимые значения. Таким образом, следующий код вызовет ошибку на этапе компиляции, потому что внутри контекста метода отсутствует присваивание значения выходному параметру:

```

static void ThisWontCompile(out int a)
{
    Console.WriteLine("Error! Forgot to assign output arg!");
}

```

Модификатор `ref`

А теперь посмотрим, как в C# используется модификатор `ref`. Ссылочные параметры необходимы, когда нужно позволить методу оперировать (и обычно изменять значения) различными элементами данных, объявленных в вызывающем коде (таким как процедура сортировки или обмена). Обратите внимание на следующие отличия между ссылочными и выходными параметрами.

- Выходные параметры не нуждаются в инициализации перед передачей методу. Причина в том, что метод сам должен присваивать значения выходным параметрам перед выходом.
- Ссылочные параметры должны быть инициализированы перед передачей методу. Причина в том, что передается ссылка на существующую переменную. Если начальное значение ей не присвоено, это будет равнозначно оперированию над неинициализированной локальной переменной.

Давайте рассмотрим применение ключевого слова `ref` на примере метода, меняющего местами две строковых переменных (естественно, здесь могли бы использоваться два любых типа данных, включая `int`, `bool`, `float` и т.д.):

```

// Ссылочные параметры.
public static void SwapStrings(ref string s1, ref string s2)
{
    string tempStr = s1;
    s1 = s2;
    s2 = tempStr;
}

```

Этот метод может быть вызван следующим образом:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Methods *****");
    ...
    string str1 = "Flip";
    string str2 = "Flop";
    Console.WriteLine("Before: {0}, {1} ", str1, str2);
    SwapStrings(ref str1, ref str2);
    Console.WriteLine("After: {0}, {1} ", str1, str2);
    Console.ReadLine();
}
```

Здесь вызывающий код присваивает начальные значения локальным строкам (str1 и str2). После вызова SwapStrings() строка str1 будет содержать значение "Flop", а строка str2 — значение "Flip":

```
Before: Flip, Flop
After: Flop, Flip
```

На заметку! Ключевое слово `ref` в C# рассматривается в разделе “Типы значений и ссылочные типы” далее в главе. Как будет показано, поведение этого ключевого слова немного меняется в зависимости от того, является аргумент типом значения или ссылочный типом.

Модификатор `params`

Язык C# поддерживает применение массивов параметров с использованием ключевого слова `params`. Чтобы понять это языковое средство, необходимо уметь манипулировать массивами C#, основные сведения о которых приведены в разделе “Массивы в C#” далее в главе.

Ключевое слово `params` позволяет передавать методу переменное количество параметров одного и того же типа (или типов классов, связанных наследованием) в виде единого логического параметра. Кроме того, аргументы, помеченные ключевым словом `params`, могут обрабатываться, если вызывающий код передает строго типизированный массив или список элементов, разделенных запятыми. Конечно, это может вызывать путаницу. Чтобы стало понятнее, предположим, что требуется создать функцию, которая бы позволила вызывающему коду передавать любое количество аргументов и возвращала бы их среднее значение.

Если прототипировать этот метод так, чтобы он принимал массив значений `double`, вызывающий код должен будет сначала определить массив, затем заполнить его значениями и, наконец, передать его методу. Однако если определить метод `CalculateAverage()` так, чтобы он принимал параметр `params` типа `double[]`, тогда вызывающий код может просто передать разделяемый запятыми список значений `double`. “За кулисами” исполняющая среда .NET автоматически упакует этот набор значений `double` в массив типа `double`:

```
// Возвращение среднего из некоторого количества значений double.
static double CalculateAverage(params double[] values)
{
    Console.WriteLine("You sent me {0} doubles.", values.Length);
    double sum = 0;
    if(values.Length == 0)
        return sum;
    for (int i = 0; i < values.Length; i++)
        sum += values[i];
    return (sum / values.Length);
}
```

Этот метод определен для приема массива параметров типа `double`. По сути, этот метод ожидает любое количество (включая ноль) значений `double` и вычисляет их среднее значение. С учетом этого, метод может вызываться любым из показанных ниже способов:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Methods *****");
    ...
    // Передать разделяемый запятыми список значений double...
    double average;
    average = CalculateAverage(4.0, 3.2, 5.7, 64.22, 87.2);
    Console.WriteLine("Average of data is: {0}", average);

    // ...или передать массив значений double.
    double[] data = { 4.0, 3.2, 5.7 };
    average = CalculateAverage(data);
    Console.WriteLine("Average of data is: {0}", average);

    // Среднее из 0 равно 0!
    Console.WriteLine("Average of data is: {0}", CalculateAverage());
    Console.ReadLine();
}
```

Если бы в определении `CalculateAverage()` не было модификатора `params`, первый вызов этого метода приводил бы к ошибке на этапе компиляции, поскольку компилятор искал бы версию `CalculateAverage()`, принимающую пять аргументов `double`.

На заметку! Во избежание любой неоднозначности, язык С# требует, чтобы метод поддерживал только один параметр `params`, который должен быть последним в списке параметров.

Как не трудно догадаться, такой подход просто более удобен для вызывающего кода, т.к. в случае его применения необходимый массив создается самой средой CLR. До момента, когда этот массив попадет в контекст вызываемого метода, его можно трактовать как полноценный массив `.NET`, обладающий всей функциональностью базового библиотечного класса `System.Array`. Ниже показан вывод приведенного выше метода:

```
You sent me 5 doubles.
Average of data is: 32.864
You sent me 3 doubles.
Average of data is: 4.3
You sent me 0 doubles.
Average of data is: 0
```

Определение необязательных параметров

Язык С# позволяет создавать методы, которые могут принимать *необязательные аргументы*. Этот прием дает возможность вызывать метод, опуская ненужные аргументы, при условии, что подходят их стандартные значения.

На заметку! Как будет показано в главе 16, основной мотивацией добавления необязательных аргументов в язык С# послужила необходимость в упрощении взаимодействия с объектами COM. Ряд объектных моделей Microsoft (например, Microsoft Office) открывают свою функциональность через объекты COM, многие из которых были написаны давно и рассчитаны на использование необязательных параметров, не поддерживаемых в ранних версиях С#.

Для иллюстрации работы с необязательными аргументами предположим, что имеется метод по имени `EnterLogData()` с одним необязательным параметром:

```
static void EnterLogData(string message, string owner = "Programmer")
{
    Console.Beep();
    Console.WriteLine("Error: {0}", message);
    Console.WriteLine("Owner of Error: {0}", owner);
}
```

Последнему аргументу `string` был присвоено стандартное значение `"Programmer"` с применением операции присваивания внутри определения параметров. В результате метод `EnterLogData()` можно вызывать из `Main()` двумя способами:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Methods *****");
    ...
    EnterLogData("Oh no! Grid can't find data");
    EnterLogData("Oh no! I can't find the payroll data", "CFO");
    Console.ReadLine();
}
```

Поскольку в первом вызове `EnterLogData()` не был указан второй аргумент `string`, будет использоваться его стандартное значение — `"Programmer"`. Во втором вызове `EnterLogData()` для второго аргумента задано значение `"CFO"`.

Еще один важный момент, о котором следует помнить, связан с тем, что значение, присваиваемое необязательному параметру, должно быть известно во время компиляции, и не может вычисляться во время выполнения (иначе на этапе компиляции будет сообщено об ошибке). В целях иллюстрации предположим, что понадобилось модифицировать метод `EnterLogData()`, добавив в него еще один необязательный параметр:

```
// Ошибка! Стандартное значение для необязательного аргумента
// должно быть известно во время компиляции!
static void EnterLogData(string message,
    string owner = "Programmer", DateTime timeStamp = DateTime.Now)
{
    Console.Beep();
    Console.WriteLine("Error: {0}", message);
    Console.WriteLine("Owner of Error: {0}", owner);
    Console.WriteLine("Time of Error: {0}", timeStamp);
}
```

Этот код не скомпилируется, потому что значение свойства `Now` класса `DateTime` вычисляется во время выполнения, а не во время компиляции.

На заметку! Во избежание неоднозначности, необязательные параметры должны всегда размещаться в конце сигнатуры метода. Если необязательный параметр окажется перед обязательными, компилятор сообщит об ошибке.

Вызов методов с использованием именованных параметров

Еще одним полезным языковым средством в C# является поддержка *именованных аргументов*. По правде говоря, на первый взгляд может показаться, что эта языковая конструкция способна лишь запутать код. И это действительно может быть так! Подобно необязательным аргументам, основным стимулом для включения поддержки именованных параметров послужило желание упростить работу с уровнем взаимодействия с COM (см. главу 16).

Именованные аргументы позволяют вызывать метод с указанием значений параметров в любом желаемом порядке. Таким образом, вместо передачи параметров исключительно в соответствии с позициями, в которых они определены (как это делается в большинстве случаев), можно указывать имя каждого аргумента, двоеточие и конкретное значение. Чтобы продемонстрировать применение именованных аргументов, добавим в класс Program следующий метод:

```
static void DisplayFancyMessage(ConsoleColor textColor,
    ConsoleColor backgroundColor, string message)
{
    // Сохранить старые цвета с целью их восстановления после вывода сообщения.
    ConsoleColor oldTextColor = Console.ForegroundColor;
    ConsoleColor oldbackgroundColor = Console.BackgroundColor;

    // Установить новые цвета и вывести сообщение.
    Console.ForegroundColor = textColor;
    Console.BackgroundColor = backgroundColor;

    Console.WriteLine(message);

    // Восстановить предыдущие цвета.
    Console.ForegroundColor = oldTextColor;
    Console.BackgroundColor = oldbackgroundColor;
}
```

Теперь, когда метод `DisplayFancyMessage()` написан, можно было бы ожидать, что при его вызове будут передаваться две переменных типа `ConsoleColor` со следующим за ним значением типа `string`. Однако за счет применения именованных аргументов `DisplayFancyMessage()` допустимо вызвать и так, как показано ниже:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Methods *****");
    ...
    DisplayFancyMessage(message: "Wow! Very Fancy indeed!",
        textColor: ConsoleColor.DarkRed,
        backgroundColor: ConsoleColor.White);

    DisplayFancyMessage(backgroundColor: ConsoleColor.Green,
        message: "Testing...",
        textColor: ConsoleColor.DarkBlue);
    Console.ReadLine();
}
```

С именованными аргументами связанная одна особенность — при вызове метода позиционные параметры должны находиться перед любыми именованными параметрами. Другими словами, именованные аргументы должны всегда размещаться в конце вызова метода. Ниже показан пример:

```
// Здесь все в порядке, т.к. позиционные аргументы идут перед именованными.
DisplayFancyMessage(ConsoleColor.Blue,
    message: "Testing...",
    backgroundColor: ConsoleColor.White);
// Это ОШИБКА, поскольку позиционные аргументы идут после именованных.
DisplayFancyMessage(message: "Testing...",
    backgroundColor: ConsoleColor.White,
    ConsoleColor.Blue);
```

Даже не учитывая это ограничение, может возникать вопрос: когда вообще понадобится эта языковая конструкция? Зачем нужно менять позиции аргументов метода?

Как выясняется, если есть метод, который определяет необязательные аргументы, то это средство может оказаться действительно полезным. Предположим, что метод `DisplayFancyMessage()` переписан с целью поддержки необязательных аргументов, для которых указаны подходящие стандартные значения:

```
static void DisplayFancyMessage(ConsoleColor textColor = ConsoleColor.Blue,
    ConsoleColor backgroundColor = ConsoleColor.White,
    string message = "Test Message")
{
    ...
}
```

Учитывая, что каждый аргумент имеет стандартное значение, именованные аргументы позволяют вызывающему коду указывать только тот параметр или параметры, которые не должны принимать стандартные значения. То есть, если нужно, чтобы значение "Hello!" появлялось в виде текста синего цвета на белом фоне, в вызывающем коде можно использовать следующую строку:

```
DisplayFancyMessage(message: "Hello!")
```

Если же необходимо, чтобы строка "Test Message" выводилась синим цветом на зеленом фоне, то должен применяться такой код:

```
DisplayFancyMessage(backgroundColor: ConsoleColor.Green);
```

Как видите, необязательные аргументы и именованные параметры действительно часто работают бок о бок. И в завершение темы построения методов C# необходимо ознакомиться с концепцией *перегрузки методов*.

Исходный код. Проект `FunWithEnums` доступен в подкаталоге `Chapter 04`.

Понятие перегрузки методов

Как и другие современные языки объектно-ориентированного программирования, C# допускает *перегрузку* методов. Выражаясь просто, когда определяется набор одинаково именованных методов, которые отличаются друг от друга количеством (или типом) параметров, то говорят, что такой метод был перегружен.

Чтобы оценить удобство перегрузки методов, давайте представим себя на месте разработчика, использующего Visual Basic 6.0. Предположим, что требуется создать набор методов, возвращающих сумму значений различных типов (`Integer`, `Double` и т.д.). Из-за того, что VB6 не поддерживает перегрузку методов, придется определить уникальный набор методов, каждый из которых, в сущности, будет делать одно и то же (возвращать сумму аргументов):

```
' Примеры кода VB6.
Public Function AddInts(ByVal x As Integer, ByVal y As Integer) As Integer
    AddInts = x + y
End Function

Public Function AddDoubles(ByVal x As Double, ByVal y As Double) As Double
    AddDoubles = x + y
End Function

Public Function AddLongs(ByVal x As Long, ByVal y As Long) As Long
    AddLongs = x + y
End Function
```

Такой код не только труден в сопровождении, но и заставляет помнить имя каждого метода. Используя перегрузку, можно предоставить вызывающему коду возможность

вызова единственного метода по имени `Add()`. Ключевой момент заключается в обеспечении отличающегося набора аргументов для каждой версии метода (различий только в возвращаемом типе не достаточно).

На заметку! Как объясняется в главе 9, существует возможность построения обобщенных методов, которые переводят концепцию перегрузки на новый уровень. Применяя обобщения, можно определять *заполнители типов* для реализации метода, которые указываются во время вызова этого метода.

Чтобы попрактиковаться с перегруженными методами, создадим новый проект консольного приложения по имени `Methodoverloading` и добавим в него следующее определение класса:

```
// Код C#.
class Program
{
    static void Main(string[] args)
    {
    }
    // Перегруженный метод Add().
    static int Add(int x, int y)
    { return x + y; }
    static double Add(double x, double y)
    { return x + y; }
    static long Add(long x, long y)
    { return x + y; }
}
```

Теперь вызывающий код может просто обращаться к методу `Add()` с требуемыми аргументами, а компилятор будет самостоятельно находить подходящую для вызова реализацию на основе предоставленных аргументов:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Method Overloading *****\n");
    // Вызов int-версии Add().
    Console.WriteLine(Add(10, 10));
    // Вызов long-версии Add().
    Console.WriteLine(Add(9000000000000, 9000000000000));
    // Вызов double-версии Add().
    Console.WriteLine(Add(4.3, 4.4));
    Console.ReadLine();
}
```

Среда `Visual Studio` оказывает помощь при вызове перегруженных методов. При вводе имени перегруженного метода (как, например, хорошо знакомого `Console.WriteLine()`) средство `IntelliSense` выводит список всех его доступных версий. Обратите внимание, что по списку можно легко перемещаться, щелкая на кнопках со стрелками вниз и вверх (рис. 4.1).

Исходный код. Проект `MethodOverloading` доступен в подкаталоге `Chapter 04`.

На этом начальный обзор деталей построения методов с использованием синтаксиса C# завершен. Теперь давайте посмотрим, как в C# создавать и манипулировать массивами, перечислениями и структурами.

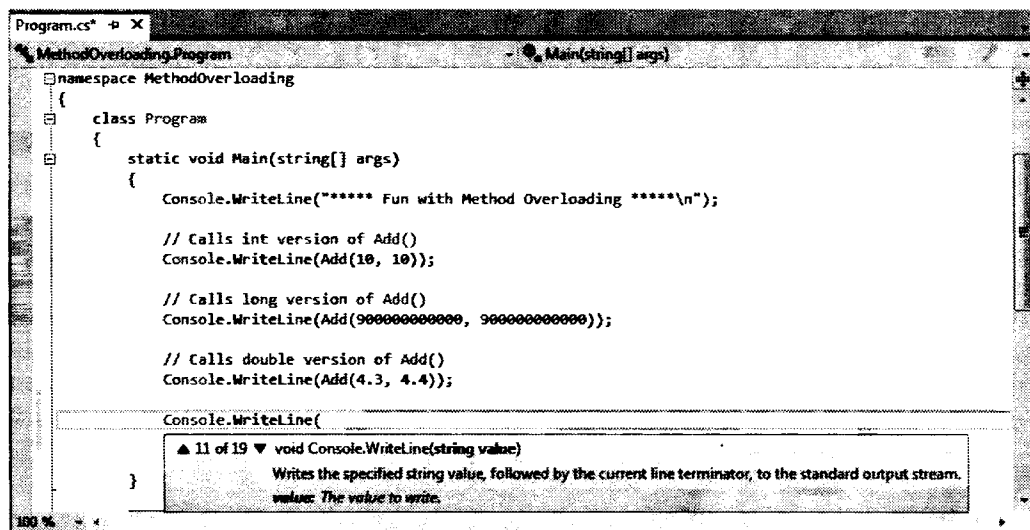


Рис. 4.1. Средство IntelliSense в Visual Studio для перегруженных методов

Массивы в C#

Как, скорее всего, вам уже известно, *массив* — это набор элементов данных, для доступа к которым используется числовой индекс. Выражаясь более конкретно, любой массив, по сути, представляет собой набор расположенных рядом элементов данных одного и того же типа (массив `int`, массив `string`, массив `SportCar` и т.п.). Объявление, заполнение и доступ к массиву в C# довольно прямолинейны. В целях иллюстрации создадим новый проект консольного приложения (по имени `FunWithArrays`), который содержит вспомогательный метод под названием `SimpleArrays()`, вызываемый из `Main()`:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with Arrays *****");
        SimpleArrays();
        Console.ReadLine();
    }

    static void SimpleArrays()
    {
        Console.WriteLine("> Simple Array Creation.");
        // Создать массив int, содержащий 3 элемента с индексами 0, 1, 2.
        int[] myInts = new int[3];

        // Создать массив string, содержащий 100 элементов с индексами 0 – 99.
        string[] booksOnDotNet = new string[100];
        Console.WriteLine();
    }
}
```

Внимательно читайте комментарии в коде. При объявлении массива C# с помощью такого синтаксиса указывается в объявлении число обозначает общее количество элементов, а не верхнюю границу. Кроме того, нижняя граница в массиве всегда начинается с 0. Следовательно, в результате объявления `int[] myInts = new int[3]` получается массив, содержащий три элемента, проиндексированные по позициям 0, 1, 2.

После определения переменной массива можно переходить к заполнению элементов от индекса к индексу, как показано ниже в модифицированном методе `SimpleArrays()`:

```
static void SimpleArrays()
{
    Console.WriteLine("> Simple Array Creation.");
    // Создать и заполнить массив из трех целочисленных значений.
    int[] myInts = new int[3];
    myInts[0] = 100;
    myInts[1] = 200;
    myInts[2] = 300;

    // Вывести все значения.
    foreach(int i in myInts)
        Console.WriteLine(i);
    Console.WriteLine();
}
```

На заметку! Имейте в виду, что если массив объявлен, но явно не заполнен по каждому индексу, его элементы получают стандартное значение для соответствующего типа данных (например, элементы массива `bool` будут установлены в `false`, а элементы массива `int` — в `0`).

Синтаксис инициализации массивов C#

В дополнение к заполнению массива элемент за элементом это можно также делать с использованием *синтаксиса инициализации массивов*. Для этого понадобится указать все элементы массива в фигурных скобках `{ }`. Такой синтаксис удобен при создании массива известного размера, когда нужно быстро задать его начальные значения. Ниже показаны альтернативные версии объявления массива:

```
static void ArrayInitialization()
{
    Console.WriteLine("> Array Initialization.");

    // Синтаксис инициализации массива с использованием ключевого слова new.
    string[] stringArray = new string[]
    { "one", "two", "three" };
    Console.WriteLine("stringArray has {0} elements", stringArray.Length);

    // Синтаксис инициализации массива без использования ключевого слова new.
    bool[] boolArray = { false, false, true };
    Console.WriteLine("boolArray has {0} elements", boolArray.Length);

    // Синтаксис инициализации массива с использованием ключевого слова new и размера.
    int[] intArray = new int[4] { 20, 22, 23, 0 };
    Console.WriteLine("intArray has {0} elements", intArray.Length);
    Console.WriteLine();
}
```

Обратите внимание, что в случае применения синтаксиса с фигурными скобками размер массива указывать не требуется (как видно на примере создания переменной `stringArray`), поскольку этот размер автоматически вычисляется на основе количества элементов внутри фигурных скобок. Кроме того, использовать ключевое слово `new` не обязательно (как при создании массива `boolArray`).

В случае объявления `intArray` снова вспомните, что указанное числовое значение представляет количество элементов в массиве, а не верхнюю границу. Если объявленный размер и количество инициализаторов не совпадают (т.е. инициализаторов слишком много либо же их не хватает), то на этапе компиляции возникнет ошибка.

Пример представлен ниже:

```
// Несоответствие размера и количества элементов!
int[] intArray = new int[2] { 20, 22, 23, 0 };
int[] intArray = new int[2] { 20, 22, 23, 0 };
```

Неявно типизированные локальные массивы

В главе 3 рассматривалась тема неявно типизированных локальных переменных. Вспомните, что ключевое слово `var` позволяет определить переменную так, тип которой выводится компилятором. Аналогичным образом можно также определять *неявно типизированные локальные массивы*. Используя такой подход, можно выделить память под новую переменную массива без указания типа элементов, содержащихся в массиве:

```
static void DeclareImplicitArrays()
{
    Console.WriteLine("=> Implicit Array Initialization.");
    // a – на самом деле int[].
    var a = new[] { 1, 10, 100, 1000 };
    Console.WriteLine("a is a: {0}", a.ToString());
    // b – на самом деле double[].
    var b = new[] { 1, 1.5, 2, 2.5 };
    Console.WriteLine("b is a: {0}", b.ToString());
    // c – на самом деле string[].
    var c = new[] { "hello", null, "world" };
    Console.WriteLine("c is a: {0}", c.ToString());
    Console.WriteLine();
}
```

Разумеется, как и при создании массива с использованием явного синтаксиса C#, элементы, указываемые в списке инициализации массива, должны иметь один и тот же тип (например, должны все быть `int`, `string` или `SportsCar`). В отличие от возможных ожиданий, неявно типизированный локальный массив не получает по умолчанию тип `System.Object`, поэтому приведенный ниже код приводит к ошибке на этапе компиляции:

```
// Ошибка! Используются смешанные типы!
var d = new[] { 1, "one", 2, "two", false };
```

Определение массива объектов

В большинстве случаев массив определяется за счет указания явного типа элементов, содержащихся в массиве. Хотя это выглядит довольно прямолинейным, существует одна важная особенность. Как будет показано в главе 6, изначальным базовым классом для каждого типа (включая фундаментальные типы данных) в системе типов .NET является `System.Object`. С учетом этого факта, если определить массив типа данных `System.Object`, то его элементы могут представлять все что угодно. Взгляните на следующий метод `ArrayOfObjects()` (который может быть вызван в `Main()` для тестирования):

```
static void ArrayOfObjects()
{
    Console.WriteLine("=> Array of Objects.");
    // Массив объектов может содержать все что угодно.
    object[] myObjects = new object[4];
    myObjects[0] = 10;
    myObjects[1] = false;
    myObjects[2] = new DateTime(1969, 3, 24);
```

```

myObjects[3] = "Form & Void";
foreach (object obj in myObjects)
{
    // Вывести тип и значение каждого элемента в массиве.
    Console.WriteLine("Type: {0}, Value: {1}", obj.GetType(), obj);
}
Console.WriteLine();
}

```

Во время итерации по содержимому массива `myObjects` производится вывод лежащего в основе типа для каждого элемента (с помощью метода `GetType()` из `System.Object`), а также его значения. Не вдаваясь пока в детали метода `System.Object.GetType()`, отметим, что этот метод может использоваться для получения полностью заданного имени элемента (информация о типах и службы рефлексии подробно рассматриваются в главе 15). В результате вызова `ArrayOfObjects()` получается следующий вывод:

```

=> Array of Objects.
Type: System.Int32, Value: 10
Type: System.Boolean, Value: False
Type: System.DateTime, Value: 3/24/1969 12:00:00 AM
Type: System.String, Value: Form & Void

```

Работа с многомерными массивами

В дополнение к одномерным массивам, которые демонстрировались до сих пор, в языке С# также поддерживаются две разновидности многомерных массивов. Первая разновидность — это *прямоугольный массив*, который просто содержит несколько измерений и все его строки имеют одинаковую длину. Прямоугольный многомерный массив объявляется и заполняется следующим образом:

```

static void RectMultidimensionalArray()
{
    Console.WriteLine("> Rectangular multidimensional array.");
    // Прямоугольный многомерный массив.
    int[,] myMatrix;
    myMatrix = new int[6,6];

    // Заполнить массив (6 * 6).
    for(int i = 0; i < 6; i++)
        for(int j = 0; j < 6; j++)
            myMatrix[i, j] = i * j;

    // Вывести массив (6 * 6).
    for(int i = 0; i < 6; i++)
    {
        for(int j = 0; j < 6; j++)
            Console.Write(myMatrix[i, j] + "\t");
        Console.WriteLine();
    }
    Console.WriteLine();
}

```

Вторая разновидность многомерных массивов — это *зубчатый (ступенчатый) массив*, содержащий некоторое количество внутренних массивов, каждый из которых может иметь отличающийся верхний предел. Вот пример:

```

static void JaggedMultidimensionalArray()
{
    Console.WriteLine("> Jagged multidimensional array.");
    // Зубчатый многомерный массив (т.е. массив массивов).

```

```
// Здесь мы имеем массив из 5 разных массивов.
int[][] myJagArray = new int[5][];

// Создать зубчатый массив.
for (int i = 0; i < myJagArray.Length; i++)
    myJagArray[i] = new int[i + 7];

// Вывести каждую строку (не забывайте, что каждый элемент имеет
// стандартное значение 0).
for(int i = 0; i < 5; i++)
{
    for(int j = 0; j < myJagArray[i].Length; j++)
        Console.Write(myJagArray[i][j] + " ");
    Console.WriteLine();
}
Console.WriteLine();
}
```

На рис. 4.2 показан вывод, полученный в результате вызова в Main() методов RectMultidimensionalArray() и JaggedMultidimensionalArray().

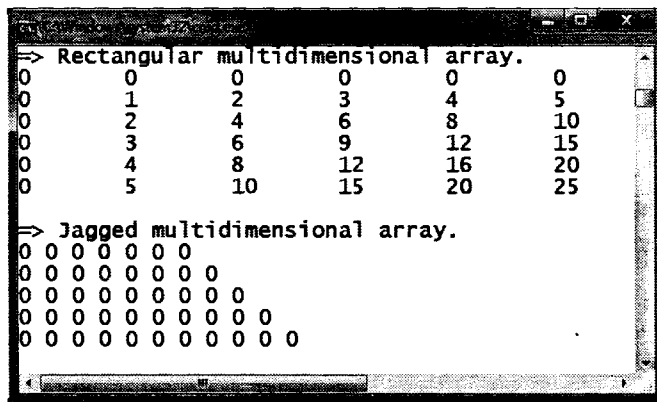


Рис. 4.2. Прямоугольные и зубчатые многомерные массивы

Использование массивов в качестве аргументов и возвращаемых значений

Сразу после создания массив можно передавать как аргумент или получать в виде возвращаемого значения. Например, приведенный ниже метод PrintArray() принимает входящий массив значений int и выводит все его элементы на консоль, а метод GetString() заполняет массив значений string и возвращает его вызывающему коду:

```
static void PrintArray(int[] myInts)
{
    for(int i = 0; i < myInts.Length; i++)
        Console.WriteLine("Item {0} is {1}", i, myInts[i]);
}

static string[] GetStringArray()
{
    string[] theStrings = {"Hello", "from", "GetStringArray"};
    return theStrings;
}
```

Эти методы могут быть вызваны вполне ожидаемым образом:

```
static void PassAndReceiveArrays()
{
    Console.WriteLine("=> Arrays as params and return values.");
    // Передать массив в качестве параметра.
    int[] ages = {20, 22, 23, 0};
    PrintArray(ages);

    // Получить массив в качестве возвращаемого значения.
    string[] strs = GetStringArray();
    foreach (string s in strs)
        Console.WriteLine(s);

    Console.WriteLine();
}
```

К этому моменту вы должны уметь определять, заполнять и просматривать содержимое массивов C#. В заключение давайте рассмотрим роль класса `System.Array`.

Базовый класс `System.Array`

Каждый создаваемый массив получает большую часть своей функциональности от класса `System.Array`. Общие члены этого класса позволяют работать с массивом с использованием полноценной объектной модели. В табл. 4.2 приведено краткое описание наиболее интересных членов класса `System.Array` (за полным описанием этих и других членов данного класса обращайтесь в документацию .NET Framework 4.5 SDK).

Таблица 4.2. Избранные члены класса `System.Array`

Член класса <code>System.Array</code>	Описание
<code>Clear()</code>	Этот статический метод устанавливает пустые значения для заданного диапазона элементов в массиве (0 — для чисел, <code>null</code> — для объектных ссылок и <code>false</code> — для булевских выражений)
<code>CopyTo()</code>	Этот метод используется для копирования элементов из исходного массива в целевой массив
<code>Length</code>	Это свойство возвращает количество элементов в массиве
<code>Rank</code>	Это свойство возвращает количество измерений в массиве
<code>Reverse()</code>	Этот статический метод обращает содержимое одномерного массива
<code>Sort()</code>	Этот статический метод сортирует одномерный массив внутренних типов. Если элементы в массиве реализуют интерфейс <code>IComparer</code> , можно также сортировать специальные типы (глава 9)

Давайте посмотрим на некоторые из этих членов в действии. Приведенный ниже вспомогательный метод использует статические методы `Reverse()` и `Clear()` для вывода на консоль информации о массиве строковых типов:

```
static void SystemArrayFunctionality()
{
    Console.WriteLine("=> Working with System.Array.");
    // Инициализировать элементы при запуске.
    string[] gothicBands = {"Tones on Tail", "Bauhaus", "Sisters of Mercy"};

    // Вывести имена в порядке их объявления.
    Console.WriteLine("-> Here is the array:");
    for (int i = 0; i < gothicBands.Length; i++)
    {
```



```

    // Вывести имя.
    Console.Write(gothicBands[i] + ", ");
}
Console.WriteLine("\n");
// Обратить порядок следования элементов...
Array.Reverse(gothicBands);
Console.WriteLine("-> The reversed array");
// ... и вывести их.
for (int i = 0; i < gothicBands.Length; i++)
{
    // Вывести имя.
    Console.Write(gothicBands[i] + ", ");
}
Console.WriteLine("\n");
// Удалить все элементы кроме последнего.
Console.WriteLine("-> Cleared out all but one...");
Array.Clear(gothicBands, 1, 2);
for (int i = 0; i < gothicBands.Length; i++)
{
    // Вывести имя.
    Console.Write(gothicBands[i] + ", ");
}
Console.WriteLine();
}

```

Вызов этого метода в Main() дает следующий вывод:

```

=> Working with System.Array.
-> Here is the array:
Tones on Tail, Bauhaus, Sisters of Mercy,
-> The reversed array
Sisters of Mercy, Bauhaus, Tones on Tail,
-> Cleared out all but one...
Sisters of Mercy, , ,

```

Обратите внимание, что многие члены `System.Array` определены как статические и потому вызываются на уровне класса (примерами могут служить методы `Array.Sort()` и `Array.Reverse()`). Таким методам передается массив, подлежащий обработке. Другие члены `System.Array` (такие как свойство `Length`) действуют на уровне объекта и потому могут вызываться прямо на массиве.

Исходный код. Проект `FunWithArrays` доступен в подкаталоге `Chapter 04`.

Тип `enum`

Вспомните из главы 1, что система типов .NET состоит из классов, структур, перечислений, интерфейсов и делегатов. Приступая к исследованию этих типов, начнем с рассмотрения роли *перечисления* (`enum`), создав новый проект консольного приложения по имени `FunWithEnums`.

На заметку! Не путайте термины “перечисление” и “перечислитель”; они обозначают совершенно разные концепции. Перечисление — это специальный тип данных, состоящих из пар “имя/значение”. Перечислитель — это класс или структура, которая реализует интерфейс `.NET` по имени `IEnumerable`. Обычно этот интерфейс реализуется классами коллекций, а также классом `System.Array`. Как будет показано в главе 8, объекты, поддерживающие `IEnumerable`, могут работать с циклами `foreach`.

При построении какой-либо системы часто удобно создавать набор символических имен, которые отображаются на известные числовые значения. Например, в случае создания системы начисления заработной платы необходимо сослаться на типы сотрудников с помощью таких констант, как `VicePresident` (вице-президент), `Manager` (менеджер), `Contractor` (подрядчик) и `Grunt` (рядовой сотрудник). Для этой цели в С# поддерживается нотация специальных перечислений. Например, ниже показано специальное перечисление по имени `EmpType`:

```
// Специальное перечисление.
enum EmpType
{
    Manager,           // = 0
    Grunt,             // = 1
    Contractor,        // = 2
    VicePresident       // = 3
}
```

В перечислении `EmpType` определены четыре именованных константы, соответствующие дискретным числовым значениям. По умолчанию первому элементу присваивается значение 0, а всем остальным элементам значения присваиваются согласно схеме $n + 1$. При желании исходное значение можно изменять как угодно. Например, если имеет смысл нумеровать члены `EmpType` с 102 до 105, можно поступить следующим образом:

```
// Начать нумерацию со значения 102.
enum EmpType
{
    Manager = 102,
    Grunt,           // = 103
    Contractor,      // = 104
    VicePresident     // = 105
}
```

Нумерация в перечислениях не обязательно должна быть последовательной и содержать только уникальные значения. Если (по той или иной причине) необходимо сконфигурировать перечисление `EmpType` показанным ниже образом, компиляция пройдет гладко и без ошибок:

```
// Значения элементов в перечислении не обязательно должны быть последовательными!
enum EmpType
{
    Manager = 10,
    Grunt = 1,
    Contractor = 100,
    VicePresident = 9
}
```

Управление хранилищем, лежащим в основе перечисления

По умолчанию для хранения значений перечисления используется тип `System.Int32` (`int` в С#); однако при желании его легко заменить. Перечисления в С# можно определять похожим образом для любых ключевых системных типов (`byte`, `short`, `int` или `long`). Например, чтобы в основе перечисления `EmpType` лежал тип `byte`, а не `int`, можно записать так:

```
// На этот раз EmpType отображается на тип byte.
enum EmpType : byte
{
    Manager = 10,
    Grunt = 1,
}
```

```

    Contractor = 100,
    VicePresident = 9
}

```

Изменение типа, лежащего в основе перечисления, полезно при построении приложения .NET, которое планируется развертывать на устройствах с небольшим объемом памяти (таких как устройства Windows Phone 7), чтобы сэкономить память везде, где возможно. Естественно, когда в качестве типа хранилища для перечисления указан `byte`, каждое значение должно входить в диапазон его допустимых значений. Например, следующая версия `EmpType` вызовет ошибку на этапе компиляции, поскольку значение 999 не вписывается в диапазон допустимых значений типа `byte`:

```

// Ошибка на этапе компиляции! Значение 999 слишком велико для типа byte!
enum EmpType : byte
{
    Manager = 10,
    Grunt = 1,
    Contractor = 100,
    VicePresident = 999
}

```

Объявление переменных типа перечисления

После установки диапазона и типа хранилища перечисление можно использовать вместо так называемых “магических чисел”. Поскольку перечисления — это просто определяемый пользователем тип данных, их можно применять как возвращаемые значения функций, параметры методов, локальные переменные и т.д. Предположим, что имеется метод по имени `AskForBonus()`, принимающий в качестве единственного параметра переменную `EmpType`. На основе значения этого входного параметра в окно консоли будет выводиться соответствующий ответ на запрос о надбавке к зарплате.

```

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("**** Fun with Enums ****");
        // Создать тип подрядчика.
        EmpType emp = EmpType.Contractor;
        AskForBonus(emp);
        Console.ReadLine();
    }

    // Использовать перечисления в качестве параметров.
    static void AskForBonus(EmpType e)
    {
        switch (e)
        {
            case EmpType.Manager:
                Console.WriteLine("How about stock options instead?");
                // Не желаете ли взамен фондовые опционы?
                break;
            case EmpType.Grunt:
                Console.WriteLine("You have got to be kidding...");
                // Вы, наверное, шутите...
                break;
            case EmpType.Contractor:
                Console.WriteLine("You already get enough cash...");
                // Вы уже получаете вполне достаточно...
                break;
        }
    }
}

```

```

        case EmpType.VicePresident:
            Console.WriteLine("VERY GOOD, Sir!");
            // Очень хорошо, сэр!
        break;
    }
}

```

Обратите внимание, что в операторе присваивания значения переменной типа `enum` перед самым значением (`Grunt`) должно быть указано имя перечисления (`EmpType`). Поскольку перечисления представляют собой фиксированные наборы пар “имя/значение”, установка переменной типа `enum` в значение, которое не определено напрямую в перечислимом типе, не допускается:

```

static void ThisMethodWillNotCompile()
{
    // Ошибка! SalesManager отсутствует в перечислении EmpType!
    EmpType emp = EmpType.SalesManager;

    // Ошибка! Не указано имя EmpType перед значением Grunt!
    emp = Grunt;
}

```

Тип System.Enum

С перечислениями .NET связан один интересный аспект — они получают свою функциональность от класса `System.Enum`. В этом классе определено множество методов, которые позволяют исследовать и трансформировать заданное перечисление. Одним из них является метод `Enum.GetUnderlyingType()`, который возвращает тип данных, используемый для хранения значений перечислимого типа (в текущем объявлении `EmpType` это `System.Byte`):

```

static void Main(string[] args)
{
    Console.WriteLine("**** Fun with Enums ****");
    // Создать тип подрядчика.
    EmpType emp = EmpType.Contractor;
    AskForBonus(emp);

    // Вывести тип хранилища, используемый в перечислении.
    Console.WriteLine("EmpType uses a {0} for storage",
        Enum.GetUnderlyingType(emp.GetType()));
    Console.ReadLine();
}

```

Заглянув в браузер объектов Visual Studio, можно удостовериться, что метод `Enum.GetUnderlyingType()` требует передачи `System.Type` в качестве первого параметра. В главе 15 будет показано, что `Type` представляет описание метаданных для конкретной сущности .NET.

Один из возможных способов получения метаданных (как показывалось ранее) предусматривает применение метода `GetType()`, который является общим для всех типов в библиотеках базовых классов .NET. Другой подход состоит в использовании операции `typeof` из C#. Преимущество этого способа связано с тем, что он не требует объявления переменной сущности, описание метаданных которой требуется получить:

```

// На этот раз для получения информации о типе применяется операция typeof.
Console.WriteLine("EmpType uses a {0} for storage",
    Enum.GetUnderlyingType(typeof(EmpType)));

```

Динамическое извлечение пар “имя/значение” перечисления

Помимо метода `Enum.GetUnderlyingType()`, все перечисления C# также поддерживают метод по имени `ToString()`, который возвращает строковое имя текущего значения перечисления. Ниже приведен соответствующий пример:

```
static void Main(string[] args)
{
    Console.WriteLine("**** Fun with Enums ****");
    EmpType emp = EmpType.Contractor;
    AskForBonus(emp);

    // Выводит строку "emp is a Contractor".
    Console.WriteLine("emp is a {0}.", emp.ToString());
    Console.ReadLine();
}
```

Чтобы выяснить не имя, а значение определенной переменной перечисления, можно просто привести ее к лежащему в основе типу хранения. Вот пример:

```
static void Main(string[] args)
{
    Console.WriteLine("**** Fun with Enums ****");
    EmpType emp = EmpType.Contractor;
    ...

    // Выводит строку "Contractor = 100".
    Console.WriteLine("{0} = {1}", emp.ToString(), (byte)emp);
    Console.ReadLine();
}
```

На заметку! Статический метод `Enum.Format()` предоставляет более высокий уровень форматирования за счет указания флага желаемого формата. Более подробную информацию об этом методе ищите в документации .NET Framework 4.5 SDK.

В `System.Enum` также определен еще один статический метод по имени `GetValues()`. Этот метод возвращает экземпляр `System.Array`. Каждый элемент в массиве соответствует члену в указанном перечислении. Рассмотрим показанный ниже метод, который выводит на консоль пары “имя/значение” из перечисления, переданного в качестве параметра:

```
// Этот метод выводит детали любого перечисления.
static void EvaluateEnum(System.Enum e)
{
    Console.WriteLine("=> Information about {0}", e.GetType().Name);
    Console.WriteLine("Underlying storage type: {0}",
        Enum.GetUnderlyingType(e.GetType()));
    // Получить все пары "имя/значение" для входного параметра.
    Array enumData = Enum.GetValues(e.GetType());
    Console.WriteLine("This enum has {0} members.", enumData.Length);
    // Вывести строковое имя и ассоциированное значение,
    // используя флаг формата D (см. главу 3).
    for(int i = 0; i < enumData.Length; i++)
    {
        Console.WriteLine("Name: {0}, Value: {0:D}",
            enumData.GetValue(i));
    }
    Console.WriteLine();
}
```

Чтобы протестировать этот новый метод, модифицируем `Main()` для создания переменных нескольких типов перечислений, объявленных в пространстве имен `System` (вместе с перечислением `EmpType`):

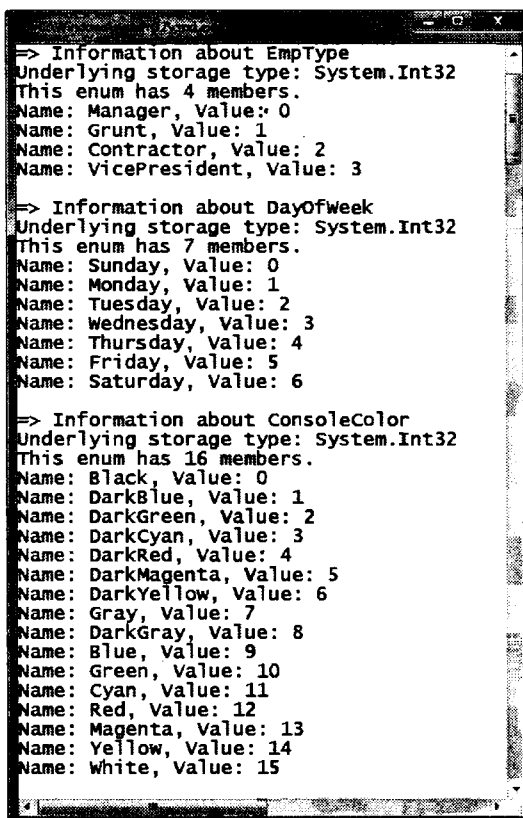
```
static void Main(string[] args)
{
    Console.WriteLine("**** Fun with Enums ****");
    ...
    EmpType e2 = EmpType.Contractor;

    // Эти типы являются перечислениями из пространства имен System.
    DayOfWeek day = DayOfWeek.Monday;
    ConsoleColor cc = ConsoleColor.Gray;

    EvaluateEnum(e2);
    EvaluateEnum(day);
    EvaluateEnum(cc);
    Console.ReadLine();
}
```

Результирующий вывод представлен на рис. 4.3.

Как можно будет убедиться по ходу чтения настоящей книги, перечисления широко используются во всех библиотеках базовых классов .NET. Например, в ADO.NET множество перечислений применяется для представления состояния подключения к базе данных (например, открыто или закрыто) либо состояния строки в `DataTable` (например, изменена, новая или отсоединена). Таким образом, когда вы пользуетесь любым перечислением, всегда помните о возможности взаимодействия с парами “имя/значение” с помощью членов класса `System.Enum`.



```
=> Information about EmpType
Underlying storage type: System.Int32
This enum has 4 members.
Name: Manager, Value: 0
Name: Grunt, Value: 1
Name: Contractor, Value: 2
Name: VicePresident, Value: 3

=> Information about DayOfWeek
Underlying storage type: System.Int32
This enum has 7 members.
Name: Sunday, Value: 0
Name: Monday, Value: 1
Name: Tuesday, Value: 2
Name: Wednesday, Value: 3
Name: Thursday, Value: 4
Name: Friday, Value: 5
Name: Saturday, Value: 6

=> Information about ConsoleColor
Underlying storage type: System.Int32
This enum has 16 members.
Name: Black, Value: 0
Name: DarkBlue, Value: 1
Name: DarkGreen, Value: 2
Name: DarkCyan, Value: 3
Name: DarkRed, Value: 4
Name: DarkMagenta, Value: 5
Name: DarkYellow, Value: 6
Name: Gray, Value: 7
Name: DarkGray, Value: 8
Name: Blue, Value: 9
Name: Green, Value: 10
Name: Cyan, Value: 11
Name: Red, Value: 12
Name: Magenta, Value: 13
Name: Yellow, Value: 14
Name: White, Value: 15
```

Рис. 4.3. Динамическое извлечение пар “имя/значение” для типов перечислений

Исходный код. Проект FunWithEnums доступен в подкаталоге Chapter 04.

Типы структур

Теперь, когда роль типов перечислений должна быть ясна, давайте посмотрим, как используются *структуры* (struct) в .NET. Типы структур хорошо подходят для моделирования математических, геометрических и других “атомарных” сущностей в приложении. Структура (как и перечисление) — это определяемый пользователем тип; тем не менее, структура не является просто коллекцией пар “имя/значение”. Вместо этого структуры представляют собой типы, которые могут содержать любое количество полей данных и членов, оперирующих над этими полями.

На заметку! Если вы имеете опыт объектно-ориентированного программирования, можете считать структуры “облегченными классами”, т.к. они предоставляют способ определения типа, поддерживающего инкапсуляцию, но не могут применяться для построения семейства взаимосвязанных типов. Когда возникает потребность в создании семейства типов, связанных через наследование, необходимо использовать классы.

На первый взгляд процесс определения и использования структур выглядит очень просто, но, как известно, сложности обычно скрываются в деталях. Чтобы приступить к изучению основных аспектов структур, создадим новый проект по имени FunWithStructures. В C# структуры определяются с применением ключевого слова struct. Определим новую структуру под названием Point, содержащую две переменных-члена типа int и набор методов для взаимодействия с ними:

```
struct Point
{
    // Поля структуры.
    public int X;
    public int Y;

    // Добавить 1 к позиции (X, Y).
    public void Increment()
    {
        X++; Y++;
    }

    // Вычесть 1 к позиции (X, Y).
    public void Decrement()
    {
        X--; Y--;
    }

    // Отобразить текущую позицию.
    public void Display()
    {
        Console.WriteLine("X = {0}, Y = {1}", X, Y);
    }
}
```

Здесь определены два целочисленных поля (X и Y) с использованием ключевого слова public, которое представляет собой один из модификаторов управления доступом (более подробно эта тема раскрывается в главе 5). Объявление данных с ключевым словом public обеспечивает вызывающему коду возможность напрямую получать доступ к этим данным через переменную Point (посредством операции точки).

На заметку! Определение открытых данных внутри класса или структуры обычно считается плохим стилем. Вместо этого рекомендуется определять *закрытые* данные и получать к ним доступ и изменять их с помощью *открытых* свойств. Более подробно об этом речь пойдет в главе 5.

Вот метод `Main()`, который позволяет протестировать тип `Point`:

```
static void Main(string[] args)
{
    Console.WriteLine("***** A First Look at Structures *****\n");
    // Создать начальный экземпляр Point.
    Point myPoint;
    myPoint.X = 349;
    myPoint.Y = 76;
    myPoint.Display();

    // Скорректировать значения X и Y.
    myPoint.Increment();
    myPoint.Display();
    Console.ReadLine();
}
```

Как и следовало ожидать, вывод выглядит так:

```
***** A First Look at Structures *****
X = 349, Y = 76
X = 350, Y = 77
```

Создание переменных типа структур

Для создания переменной типа структуры на выбор доступно несколько вариантов. В следующем коде просто создается переменная `Point` с присваиванием значения каждому ее открытому полю данных перед обращением к ее членам. Если не присвоить значения открытым полям данных структуры (`X` и `Y` в этом случае) перед ее использованием, компилятор сообщит об ошибке:

```
// Ошибка! Полю Y не присвоено значение.
Point p1;
p1.X = 10;
p1.Display();

// Все в порядке! Обоим полям присвоены значения перед использованием.
Point p2;
p2.X = 10;
p2.Y = 10;
p2.Display();
```

В качестве альтернативы переменные типа структур можно создавать с применением ключевого слова `new` языка С#, что приводит к вызову *стандартного конструктора* структуры. По определению стандартный конструктор не принимает аргументов. Преимущество вызова стандартного конструктора структуры связано с тем, что каждое поле данных автоматически получает свое стандартное значение:

```
// Установить все поля в стандартные значения, используя стандартный конструктор.
Point p1 = new Point();

// Выводит X=0, Y=0
p1.Display();
```

В принципе также возможно спроектировать структуру со *специальным конструктором*. Это позволит указывать значения для полей данных при создании переменной,

а не устанавливать их по отдельности. Конструкторы подробно рассматриваются в главе 5; тем не менее, в целях иллюстрации изменим структуру Point согласно следующему коду:

```
struct Point
{
    // Поля структуры.
    public int X;
    public int Y;

    // Специальный конструктор.
    public Point(int XPos, int YPos)
    {
        X = XPos;
        Y = YPos;
    }
    ...
}
```

В таком случае переменные типа Point можно создавать так, как показано ниже:

```
// Вызвать специальный конструктор.
Point p2 = new Point(50, 60);

// Выводит X=50, Y=60
p2.Display();
```

Как упоминалось ранее, на первый взгляд работа со структурами довольно проста. Однако чтобы лучше в этом разобраться, необходимо ознакомиться с отличиями между типами значений и ссылочными типами .NET.

Исходный код. Проект FunWithStructures доступен в подкаталоге Chapter 04.

Типы значений и ссылочные типы

На заметку! В представленном далее обсуждении типов значений и ссылочных типов предполагается наличие базовых знаний по объектно-ориентированному программированию. Дополнительные сведения по этой теме можно найти в главах 5 и 6.

В отличие от массивов, строк и перечислений, структуры C# не имеют идентично названного представления в библиотеке .NET (т.е. не существует класса наподобие System.Structure), однако они неявно порождены от System.ValueType. Выразаясь просто, роль класса System.ValueType заключается в обеспечении размещения экземпляра производного типа (например, любой структуры) в *стеке*, а не в *куче* с автоматической сборкой мусора. Данные, размещаемые в стеке, могут создаваться и уничтожаться очень быстро, поскольку время их существования зависит от контекста, в котором они определены. С другой стороны, сборщик мусора .NET осуществляет мониторинг данных, размещаемых в куче, и время их жизни зависит от множества факторов, которые подробно рассматриваются в главе 13.

Функционально единственным назначением System.ValueType является переопределение виртуальных методов, объявленных в System.Object, для использования семантики на основе значений, а не ссылок. Скорее всего, вы уже знаете, что переопределение — это процесс изменения реализации виртуального (или, возможно, абстрактного) метода, определенного внутри базового класса. Базовым классом для ValueType является System.Object. В действительности методы экземпляра, определенные в System.ValueType, идентичны тем, что определены в System.Object:

```
// Структуры и перечисления неявно расширяют System.ValueType.
public abstract class ValueType : object
{
    public virtual bool Equals(object obj);
    public virtual int GetHashCode();
    public Type GetType();
    public virtual string ToString();
}
```

С учетом того факта, что типы значений используют семантику на основе значений, время жизни структуры (что относится ко всем числовым типам данных (int, float), а также к любому перечислению или структуре) является весьма предсказуемым. Когда переменная типа структуры покидает пределы контекста, в котором она была определена, она немедленно удаляется из памяти:

```
// Локальные структуры извлекаются из стека при возврате управления методом.
static void LocalValueTypes()
{
    // Вспомните, что int в действительности является структурой System.Int32.
    int i = 0;

    // Вспомните, что Point в действительности является типом структуры.
    Point p = new Point();
} // Здесь i и p извлекаются из стека.
```

Типы значений, ссылочные типы и операция присваивания

Когда один тип значения присваивается другому, получается почленная копия полей данных. В случае простого типа данных, такого как System.Int32, единственным копируемым членом является числовое значение. Однако в ситуации с типом Point копироваться в новую переменную структуры будут значения X и Y. В целях иллюстрации создадим новый проект консольного приложения по имени ValueAndReferenceTypes и скопируем предыдущее определение Point в новое пространство имен. После этого добавим к типу Program следующий метод:

```
// Присваивание двух внутренних типов значений в результате
// дает две независимых переменных в стеке.
static void ValueTypeAssignment()
{
    Console.WriteLine("Assigning value types\n");

    Point p1 = new Point(10, 10);
    Point p2 = p1;

    // Вывести обе переменных Point.
    p1.Display();
    p2.Display();

    // Изменить p1.X и вывести снова. Значение p2.X не изменилось.
    p1.X = 100;
    Console.WriteLine("\n=> Changed p1.X\n");
    p1.Display();
    p2.Display();
}
```

Здесь сначала создается переменная типа Point (p1), которая затем присваивается другой переменной типа Point (p2). Поскольку Point относится к типу значения, в стеке размещаются две копии MyPoint, каждой из которых можно манипулировать независимо. Поэтому при изменении значения p1.X значение p2.X остается незатронутым:

Assigning value types

```
X = 10, Y = 10
X = 10, Y = 10
=> Changed p1.X
X = 100, Y = 10
X = 10, Y = 10
```

В отличие от типов значений, когда операция присваивания применяется к ссылочным типам (т.е. экземплярам всех классов), происходит переадресация на то, на что ссылочная переменная указывает в памяти. В целях иллюстрации создадим новый класс по имени `PointRef` с теми же членами, что и у структуры `Point`, но только переименуем конструктор в соответствие с именем этого класса:

```
// Классы всегда являются ссылочными типами.
class PointRef
{
    // Те же члены, что и в структуре Point...

    // Не забудьте изменить имя конструктора на PointRef!
    public PointRef(int XPos, int YPos)
    {
        X = XPos;
        Y = YPos;
    }
}
```

Теперь воспользуемся этим типом `PointRef` в показанном ниже новом методе. Обратите внимание, что кроме работы с классом `PointRef`, а не структурой `Point`, код идентичен методу `ValueTypeAssignment()`:

```
static void ReferenceTypeAssignment()
{
    Console.WriteLine("Assigning reference types\n");
    PointRef p1 = new PointRef(10, 10);
    PointRef p2 = p1;

    // Вывести обе переменных PointRef.
    p1.Display();
    p2.Display();

    // Изменить p1.X и вывести снова.
    p1.X = 100;
    Console.WriteLine("\n=> Changed p1.X\n");
    p1.Display();
    p2.Display();
}
```

В этом случае получаются две ссылки, указывающие на один и тот же объект в управляемой куче. Таким образом, при изменении значения `X` с использованием ссылки `p1` изменится также и значение `p2.X`. Вызов этого нового метода в `Main()` дает следующий вывод:

```
Assigning reference types
X = 10, Y = 10
X = 10, Y = 10
=> Changed p1.X
X = 100, Y = 10
X = 100, Y = 10
```

Типы значений, содержащие ссылочные типы

Теперь, когда вы лучше понимаете базовые отличия между типами значений и ссылочными типами, давайте рассмотрим более сложный пример. Предположим, что имеется следующий ссылочный тип (класс), который поддерживает информационную строку (`infoString`), устанавливаемую с помощью специального конструктора:

```
class ShapeInfo
{
    public string infoString;
    public ShapeInfo(string info)
    {
        infoString = info;
    }
}
```

Представим, что переменная этого класса должна содержаться внутри типа значения по имени `Rectangle`. Также предусмотрен специальный конструктор, который позволяет вызывающему коду устанавливать значение этой внутренней переменной-члена типа `ShapeInfo`. Вот полное определение типа `Rectangle`:

```
struct Rectangle
{
    // Структура Rectangle содержит член ссылочного типа.
    public ShapeInfo rectInfo;
    public int rectTop, rectLeft, rectBottom, rectRight;
    public Rectangle(string info, int top, int left, int bottom, int right)
    {
        rectInfo = new ShapeInfo(info);
        rectTop = top; rectBottom = bottom;
        rectLeft = left; rectRight = right;
    }
    public void Display()
    {
        Console.WriteLine("String = {0}, Top = {1}, Bottom = {2}, " +
            "Left = {3}, Right = {4}",
            rectInfo.infoString, rectTop, rectBottom, rectLeft, rectRight);
    }
}
```

Теперь ссылочный тип содержится внутри типа значения. Возникает важный вопрос: что произойдет в результате присваивания одной переменной типа `Rectangle` другой? С учетом того, что уже известно о типах значений, можно корректно предположить, что целочисленные данные (которые на самом деле являются структурой — `System.Int32`) должны быть независимой сущностью для каждой переменной `Rectangle`. Но что насчет внутреннего ссылочного типа? Будет ли полностью скопировано *состояние* этого объекта или же только ссылка на него? Чтобы получить ответ на этот вопрос, определим показанный ниже метод и вызовем его в `Main()`:

```
static void ValueTypeContainingRefType()
{
    // Создать первую переменную Rectangle.
    Console.WriteLine("-> Creating r1");
    Rectangle r1 = new Rectangle("First Rect", 10, 10, 50, 50);

    // Присвоить новой переменной Rectangle переменную r1.
    Console.WriteLine("-> Assigning r2 to r1");
    Rectangle r2 = r1;
}
```

```
// Изменить некоторые значения в r2.
Console.WriteLine("-> Changing values of r2");
r2.rectInfo.infoString = "This is new info!";
r2.rectBottom = 4444;

// Вывести значения из обеих переменных Rectangle.
r1.Display();
r2.Display();
}
```

Вывод будет следующим:

```
-> Creating r1
-> Assigning r2 to r1
-> Changing values of r2
String = This is new info!, Top = 10, Bottom = 50, Left = 10, Right = 50
String = This is new info!, Top = 10, Bottom = 4444, Left = 10, Right = 50
```

Как видите, после изменения значения информационной строки с использованием ссылки `r2` для ссылки `r1` отображается то же самое значение. По умолчанию, когда тип значения содержит другие ссылочные типы, присваивание приводит к копированию ссылок. В результате получаются две независимых структуры, каждая из которых содержит ссылку, указывающую на один и тот же объект в памяти (т.е. создается поверхностная копия). Для выполнения глубокого копирования, когда в новый объект полностью копируется состояние внутренних ссылок, можно реализовать интерфейс `ICloneable` (как будет показано в главе 8).

Исходный код. Проект `ValueAndReferenceTypes` доступен в подкаталоге `Chapter 04`.

Передача ссылочных типов по значению

Очевидно, что ссылочные типы и типы значений могут передаваться методам в виде параметров. Тем не менее, передача ссылочного типа (например, класса) по ссылке совершенно отличается от передачи его по значению. Чтобы понять разницу, предположим, что имеется простой класс `Person`, определенный в новом проекте консольного приложения по имени `RefTypeValTypeParams`:

```
class Person
{
    public string personName;
    public int personAge;

    // Конструкторы.
    public Person(string name, int age)
    {
        personName = name;
        personAge = age;
    }
    public Person() {}
    public void Display()
    {
        Console.WriteLine("Name: {0}, Age: {1}", personName, personAge);
    }
}
```

А теперь создадим метод, который позволяет вызывающему коду передавать объект `Person` по значению (отметьте отсутствие модификаторов параметров, таких как `out` или `ref`):

```
static void SendAPersonByValue(Person p)
{
    // Изменить значение возраста в p?
    p.personAge = 99;

    // Увидит ли вызывающий код это изменение?
    p = new Person("Nikki", 99);
}
```

Обратите внимание на то, как метод `SendAPersonByValue()` пытается присвоить входной ссылке `Person` новый объект `Person`, а также изменить некоторые данные состояния. Протестируем этот метод, вызвав его в `Main()` следующим образом:

```
static void Main(string[] args)
{
    // Передача ссылочных типов по значению.
    Console.WriteLine("***** Passing Person object by value *****");
    Person fred = new Person("Fred", 12);
    Console.WriteLine("\nBefore by value call, Person is:"); // перед вызовом
    fred.Display();

    SendAPersonByValue(fred);
    Console.WriteLine("\nAfter by value call, Person is:"); // после вызова
    fred.Display();
    Console.ReadLine();
}
```

Ниже показан результирующий вывод:

```
***** Passing Person object by value *****

Before by value call, Person is:
Name: Fred, Age: 12

After by value call, Person is:
Name: Fred, Age: 99
```

Как видите, значение `PersonAge` изменилось. Кажется, что такое поведение противоречит смыслу передачи параметра “по значению”. Учитывая успешную попытку изменения состояния входного объекта `Person`, возникает вопрос: а что же тогда было скопировано? Ответ: была получена копия ссылки на объект из вызывающего кода. Таким образом, поскольку метод `SendAPersonByValue()` указывает на тот же самый объект, что и вызывающий код, становится возможным изменение данных состояния этого объекта. Невозможно лишь изменять то, на что ссылка указывает.

Передача ссылочных типов по ссылке

Теперь предположим, что имеется метод `SendAPersonByReference()`, в котором ссылочный тип передается по ссылке (обратите внимание на наличие модификатора параметра `ref`):

```
static void SendAPersonByReference(ref Person p)
{
    // Изменить некоторые данные в p.
    p.personAge = 555;

    // p теперь указывает на новый объект в куче!
    p = new Person("Nikki", 999);
}
```

Нетрудно догадаться, что такой подход предоставляет вызываемому коду полную свободу в плане манипулирования входным параметром.

Вызываемый код может не только изменять состояние объекта, но и переопределять ссылку так, чтобы она указывала на новый объект `Person`. Давайте испробуем новый метод `SendAPersonByReference()`, вызвав его в методе `Main()`:

```
static void Main(string[] args)
{
    // Передача ссылочных типов по ссылке.
    Console.WriteLine("***** Passing Person object by reference *****");
    ...

    Person mel = new Person("Mel", 23);
    Console.WriteLine("Before by ref call, Person is:"); // перед вызовом
    mel.Display();

    SendAPersonByReference(ref mel);
    Console.WriteLine("After by ref call, Person is:"); // после вызова
    mel.Display();
    Console.ReadLine();
}
```

Ниже показан вывод:

```
***** Passing Person object by reference *****
Before by ref call, Person is:
Name: Mel, Age: 23
After by ref call, Person is:
Name: Nikki, Age: 999
```

Как видите, после вызова объект по имени `Mel` возвращается как объект по имени `Nikki`, т.к. метод имел возможность изменить то, на что указывала в памяти входная ссылка. Ниже представлены основные правила, которые необходимо соблюдать при передаче ссылочных типов.

- Если ссылочный тип передается по ссылке, вызываемый код может изменять значения в данных состояния объекта, а также объект, на который указывает ссылка.
- Если ссылочный тип передается по значению, вызываемый код может изменять значения в данных состояния объекта, но не объект, на который указывает ссылка.

Исходный код. Проект `RefTypeValTypeParams` доступен в подкаталоге `Chapter 04`.

Заключительные детали относительно типов значений и ссылочных типов

В завершение этой темы взгляните на табл. 4.3, в которой приведена сводка основных отличий между типами значений и ссылочными типами.

Несмотря на все эти отличия, типы значений и ссылочные типы имеют возможность реализовывать интерфейсы и могут поддерживать любое количество полей, методов, перегруженных операций, констант, свойств и событий.

Таблица 4.3. Отличия между типами значений и ссылочными типами

Интересующий вопрос	Тип значения	Ссылочный тип
Где размещаются объекты?	Размещаются в стеке	Размещаются в управляемой куче
Как представляется переменная?	Переменные типов значений являются локальными копиями	Переменные ссылочных типов указывают на память, занимаемую размещенным экземпляром
Какой тип является базовым?	Неявно расширяет <code>System.ValueType</code>	Может быть производным от любого другого типа (кроме <code>System.ValueType</code>), если только этот тип не запечатан (об этом рассказывается в главе 6)
Может ли этот тип выступать в качестве базового для других типов?	Нет. Типы значений всегда являются запечатанными, и наследовать от них нельзя	Да. Если тип не запечатан, он может выступать в качестве базового для других типов
Каково стандартное поведение передачи параметров?	Переменные передаются по значению (т.е. вызываемой функции передается копия переменной)	Для типов значений объект копируется по значению. Для ссылочных типов ссылка копируется по значению
Может ли в этом типе переопределяться метод <code>System.Object.Finalize()</code> ?	Нет. Типы значений, никогда не размещаются в куче и потому не нуждаются в финализации	Да, косвенно (как показано в главе 13)
Можно ли определять конструкторы для этого типа?	Да, но стандартный конструктор является зарезервированным (т.е. все специальные конструкторы должны иметь аргументы)	Безусловно!
Когда переменные этого типа прекращают свое существование?	Когда покидают контекст, в котором они были определены	Когда объект подвергается сборке мусора

Понятие типов, допускающих `null`, в C#

В заключение этой главы давайте исследуем роль типов данных, допускающих `null`, используя консольное приложение по имени `NullableTypes`. Как уже известно, типы данных C# обладают фиксированным диапазоном значений и представлены в виде типов пространства имен `System`. Например, типу данных `System.Boolean` могут присваиваться только значения из набора `{true, false}`. Вспомните, что все числовые типы данных (а также `Boolean`) являются типами значений. Типам значений никогда не может быть присвоено значение `null`, т.к. оно служит для представления пустой ссылки на объект:

```
static void Main(string[] args)
{
    // Ошибка на этапе компиляции!
    // Типы значений не могут быть установлены в null!
    bool myBool = null;
    int myInt = null;

    // Все в порядке! Строки являются ссылочными типами.
    string myString = null;
}
```


Язык C# поддерживает концепцию *типов данных, допускающих null*. Тип, допускающий null, может представлять все значения типа плюс null. Таким образом, если объявить допускающим null тип bool, то можно будет присваивать значение из набора {true, false, null}. Это может быть очень удобно при работе с реляционными базами данных, поскольку в таблицах баз данных довольно часто встречаются столбцы, для которых значения не определены. Без концепции типов данных, допускающих null, в C# не было бы удобного способа для представления числовых элементов данных, не имеющих значения.

Чтобы определить переменную типа, допускающего null, необходимо предварить имя необходимого типа данных знаком вопроса (?). Обратите внимание, что такой синтаксис разрешено применять только к типам значений. При попытке создать ссылочный тип, допускающий null (включая string), компилятор сообщит об ошибке. Как и переменным, не допускающим null, локальным переменным, допускающим null, должно быть присвоено начальное значение, прежде чем их можно будет использовать:

```
static void LocalNullableVariables()
{
    // Определить несколько локальных переменных, допускающих null.
    int? nullableInt = 10;
    double? nullableDouble = 3.14;
    bool? nullableBool = null;
    char? nullableChar = 'a';
    int?[] arrayOfNullableInts = new int?[10];

    // Ошибка! Строки являются ссылочными типами!
    // string? s = "oops";
}
```

В C# нотация суффикса ? — это сокращенный вариант создания экземпляра обобщенной структуры System.Nullable<T>. Хотя подробное исследование обобщений откладывается до главы 9, сейчас важно знать, что тип System.Nullable<T> предоставляет набор членов, которые могут использоваться всеми типами, допускающими null.

Например, выяснить программно, было ли переменной, допускающей null, действительно присвоено значение null, можно с помощью свойства HasValue или операции !=. Значение, которое присвоено типу, допускающему null, можно получить напрямую либо через свойство Value. Учитывая, что суффикс ? является всего лишь сокращением для использования Nullable<T>, метод LocalNullableVariables() можно было бы реализовать следующим образом:

```
static void LocalNullableVariablesUsingNullable()
{
    // Определить несколько типов, допускающих null, с применением Nullable<T>.
    Nullable<int> nullableInt = 10;
    Nullable<double> nullableDouble = 3.14;
    Nullable<bool> nullableBool = null;
    Nullable<char> nullableChar = 'a';
    Nullable<int>[] arrayOfNullableInts = new int?[10];
}
```

Работа с типами, допускающими null

Как упоминалось ранее, типы данных, допускающие null, особенно полезны при взаимодействии с базами данных, поскольку столбцы в таблицах данных могут быть преднамеренно оставлены пустыми (т.е. быть неопределенными). В целях иллюстрации рассмотрим приведенный ниже класс, эмулирующий процесс доступа к базе данных с таблицей, два столбца в которой могут принимать значения null.

Обратите внимание, что в методе `GetIntFromDatabase()` значение целочисленной переменной-члена, допускающей `null`, не присваивается, тогда как в методе `GetBoolFromDatabase()` значение члену `bool?` присваивается:

```
class DatabaseReader
{
    // Поле данных, допускающее null.
    public int? numericValue = null;
    public bool? boolValue = true;

    // Обратите внимание на возвращаемый тип, допускающий null.
    public int? GetIntFromDatabase()
    { return numericValue; }

    // Обратите внимание на возвращаемый тип, допускающий null.
    public bool? GetBoolFromDatabase()
    { return boolValue; }
}
```

Теперь предположим, что в следующем методе `Main()` вызывается каждый член класса `DatabaseReader` и выясняются присвоенные значения с помощью членов `HasValue` и `Value`, а также операции равенства C# (точнее — операции “не равно”):

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Nullable Data *****\n");
    DatabaseReader dr = new DatabaseReader();

    // Получить значение int из "базы данных".
    int? i = dr.GetIntFromDatabase();
    if (i.HasValue)
        Console.WriteLine("Value of 'i' is: {0}", i.Value);
    else
        Console.WriteLine("Value of 'i' is undefined."); // не определено

    // Получить значение bool из "базы данных".
    bool? b = dr.GetBoolFromDatabase();
    if (b != null)
        Console.WriteLine("Value of 'b' is: {0}", b.Value);
    else
        Console.WriteLine("Value of 'b' is undefined."); // не определено
    Console.ReadLine();
}
```

Операция ??

Последним аспектом типов, допускающих `null`, о котором следует знать, является возможность использования с ними операции `??` языка C#. Эта операция позволяет присваивать значение типу, допускающему `null`, если извлеченное значение на самом деле равно `null`. Для примера предположим, что в случае возврата методом `GetIntFromDatabase()` значения `null` (да, этот метод запрограммирован так, что он всегда возвращает `null`, однако общая идея должна быть ясна) локальной переменной целочисленного типа, допускающего `null`, необходимо присвоить значение 100:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Nullable Data *****\n");
    DatabaseReader dr = new DatabaseReader();
    ...
```

```
// Если значение, возвращаемое GetIntFromDatabase(),
// равно null, присвоить локальной переменной значение 100.
int myData = dr.GetIntFromDatabase() ?? 100;
Console.WriteLine("Value of myData: {0}", myData);
Console.ReadLine();
}
```

Преимущество использования операции ?? заключается в том, что она обеспечивает более компактную версию кода, чем традиционная условная конструкция if/else. Однако при желании можно написать следующий функционально эквивалентный код, который в случае null устанавливает значение переменной в 100:

```
// Более длинная нотация, не использующая синтаксис ??.
int? moreData = dr.GetIntFromDatabase();
if (!moreData.HasValue)
    moreData = 100;
Console.WriteLine("Value of moreData: {0}", moreData);
```

На этом первоначальное знакомство с языком программирования C# завершено. В главе 5 вы начнете погружаться в детали объектно-ориентированной разработки.

Исходный код. Проект NullableTypes доступен в подкаталоге Chapter 04.

Резюме

Эта глава начиналась с рассмотрения нескольких ключевых слов C#, которые позволяют строить специальные методы. Вспомните, что по умолчанию параметры передаются по значению, однако их можно передавать и по ссылке, если добавить к ним модификатор `ref` или `out`. Вы также узнали о роли необязательных и именованных параметров и о том, как определять и вызывать методы, принимающие массивы параметров.

После исследования темы перегрузки методов в главе приводились различные детали, касающиеся определения массивов, перечислений и структур в C# и их представления в библиотеках базовых классов .NET. Вдобавок были описаны основные характеристики типов значений и ссылочных типов, включая их поведение при передаче в качестве параметров методам, и способы взаимодействия с типами данных, допускающими `null`, с использованием операций `?` и `??`.

часть III

Объектно-ориентированное программирование на C#

В этой части

Глава 5. Инкапсуляция

Глава 6. Понятие наследования и полиморфизма

Глава 7. Структурированная обработка исключений

Глава 8. Работа с интерфейсами

Инкапсуляция

В главах 3 и 4 мы исследовали ряд основных синтаксических конструкций, присущих любому приложению .NET, которое вам придется разрабатывать. Здесь мы приступим к изучению объектно-ориентированных возможностей C#. Первое, что предстоит узнать — это процесс построения четко определенных типов классов, которые поддерживают любое количество *конструкторов*. После описания основ определения классов и размещения объектов в остальной части главы рассматривается тема *инкапсуляции*. По ходу изложения вы узнаете, как определяются свойства класса, а также поймете роль статических членов, синтаксиса инициализации объектов, полей, доступных только для чтения, константных данных и частичных классов.

Знакомство с типом класса C#

Что касается платформы .NET, то наиболее фундаментальной программной конструкцией является *тип класса*. Формально класс — это определяемый пользователем тип, который состоит из полей данных (часто именуемых *переменными-членами*) и членов, оперирующих этими данными (конструкторов, свойств, методов, событий и т.п.). Все вместе поля данных класса представляют “состояние” экземпляра класса (иначе называемого *объектом*). Мощь объектно-ориентированных языков, подобных C#, состоит в их способности группировать данные и связанную с ними функциональность в определении класса, что позволяет моделировать программное обеспечение на основе сущностей реального мира.

Для начала создадим новое консольное приложение C# по имени `SimpleClassExample`. Затем добавим в проект новый файл класса (по имени `Car.cs`), используя пункт меню `Project⇒Add Class` (Проект⇒Добавить класс). Далее в результирующем диалоговом окне необходимо выбрать значок `Class` (Класс), как показано на рис. 5.1, и щелкнуть на кнопке `Add` (Добавить).

Класс определяется в C# с помощью ключевого слова `class`. Вот как выглядит простейшее из возможных объявление класса:

```
class Car
{
}
```

После определения типа класса нужно определить набор переменных-членов, которые будут использоваться для представления его состояния. Например, вы можете решить, что объекты `Car` (автомобили) должны иметь поле данных типа `int`, представляющее текущую скорость, и поле данных типа `string` для представления дружественного названия автомобиля.

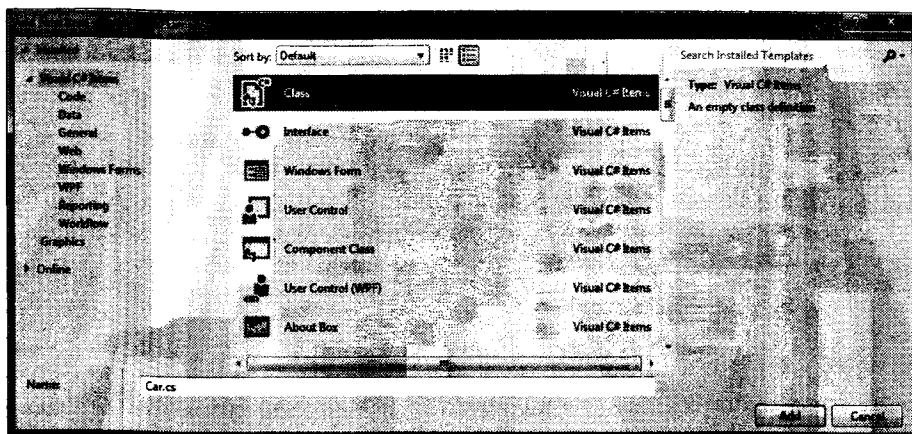


Рис. 5.1. Добавление нового типа класса C#

С учетом этих начальных проектных положений класс `Car` будет выглядеть следующим образом:

```
class Car
{
    // 'Состояние' объекта Car.
    public string petName;
    public int currSpeed;
}
```

Обратите внимание, что эти переменные-члены объявлены с использованием модификатора доступа `public`. Открытые (`public`) члены класса доступны непосредственно, как только создается объект данного типа. Как вам должно быть уже известно, термин “объект” служит для представления экземпляра данного типа класса, созданного с помощью ключевого слова `new`.

На заметку! Поля данных класса редко (если вообще когда-нибудь) должны определяться с модификатором `public`. Чтобы обеспечить целостность данных состояния, намного лучше объявлять данные закрытыми (`private`) или, возможно, защищенными (`protected`) и разрешать контролируемый доступ к данным через свойства (как будет показано далее в этой главе). Однако чтобы сделать первый пример насколько возможно простым, мы оставляем поля данных открытыми.

После определения набора переменных-членов, представляющих состояние класса, следующим шагом в проектировании будет создание членов, которые моделируют его поведение. Для этого примера класс `Car` определяет один метод по имени `SpeedUp()` и еще один — по имени `PrintState()`. Модифицируйте код класса следующим образом:

```
class Car
{
    // 'Состояние' объекта Car.
    public string petName;
    public int currSpeed;

    // Функциональность Car.
    public void PrintState()
    {
        Console.WriteLine("{0} is going {1} MPH.", petName, currSpeed);
    }
}
```

```

public void SpeedUp(int delta)
{
    currSpeed += delta;
}
}

```

Метод `PrintState()` — это в какой-то мере диагностическая функция, которая просто выводит текущее состояние объекта `Car` в окно командной строки. Метод `SpeedUp()` повышает скорость `Car`, увеличивая ее на величину, переданную во входящем параметре типа `int`. Теперь обновите код метода `Main()`, как показано ниже:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Class Types *****\n");
    // Разместить в памяти и сконфигурировать объект Car.
    Car myCar = new Car();
    myCar.petName = "Henry";
    myCar.currSpeed = 10;

    // Увеличить скорость автомобиля в несколько раз и вывести новое состояние.
    for (int i = 0; i <= 10; i++)
    {
        myCar.SpeedUp(5);
        myCar.PrintState();
    }
    Console.ReadLine();
}

```

Запустив программу, вы увидите, что переменная `Car` (`myCar`) поддерживает свое текущее состояние на протяжении жизни всего приложения, как показано в следующем выводе:

```

***** Fun with Class Types *****
Henry is going 15 MPH.
Henry is going 20 MPH.
Henry is going 25 MPH.
Henry is going 30 MPH.
Henry is going 35 MPH.
Henry is going 40 MPH.
Henry is going 45 MPH.
Henry is going 50 MPH.
Henry is going 55 MPH.
Henry is going 60 MPH.
Henry is going 65 MPH.

```

Размещение объектов с помощью ключевого слова `new`

Как было показано в предыдущем примере кода, объекты должны быть размещены в памяти с использованием ключевого слова `new`. Если ключевое слово `new` не указать и попытаться воспользоваться переменной класса в следующем операторе кода, будет получена ошибка компиляции. Например, следующий метод `Main()` компилироваться не будет:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Class Types *****\n");
    // Ошибка! Забыли использовать new для создания объекта!
    Car myCar;
    myCar.petName = "Fred";
}

```

Чтобы корректно создать объект с применением ключевого слова `new`, можно определить и разместить в памяти объект `Car` в одной строке кода:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Class Types *****\n");
    Car myCar = new Car();
    myCar.petName = "Fred";
}
```

В качестве альтернативы, определение и размещение в памяти экземпляра класса может осуществляться в разных строках кода:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Class Types *****\n");
    Car myCar;
    myCar = new Car();
    myCar.petName = "Fred";
}
```

Здесь первый оператор кода просто объявляет ссылку на еще не созданный объект типа `Car`. Ссылка будет указывать на действительный объект в памяти только после явного присваивания.

В любом случае, к этому моменту мы получили простейший тип класса, который определяет несколько элементов данных и некоторые базовые операции. Чтобы расширить функциональность текущего класса `Car`, необходимо разобраться с ролью *конструкторов*.

Понятие конструкторов

Учитывая, что объект имеет состояние (представленное значениями его переменных-членов), обычно желательно присвоить осмысленные значения полям объекта перед тем, как работать с ним. В настоящий момент тип `Car` требует присваивания значений полям `petName` и `currSpeed`. Для текущего примера это не слишком проблематично, поскольку открытых элементов данных всего два. Однако нередко классы состоят из нескольких десятков полей. Ясно, что было бы нежелательно писать 20 операторов инициализации для всех 20 элементов данных такого класса.

К счастью, в C# поддерживается механизм *конструкторов*, которые позволяют устанавливать состояние объекта в момент его создания. Конструктор — это специальный метод класса, который вызывается неявно при создании объекта с использованием ключевого слова `new`. Однако в отличие от “нормального” метода, конструктор никогда не имеет возвращаемого значения (даже `void`) и всегда именуется идентично имени класса, который он конструирует.

Роль стандартного конструктора

Каждый класс C# снабжается *стандартным конструктором*, который при необходимости может быть переопределен. По определению стандартный конструктор никогда не принимает аргументов. После размещения нового объекта в памяти стандартный конструктор гарантирует установку всех полей данных в соответствующие стандартные значения (стандартные значения для типов данных C# описаны в главе 3).

Если вы не удовлетворены такими стандартными присваиваниями, можете переопределить стандартный конструктор в соответствии со своими нуждами. В целях иллюстрации модифицируем класс C#, как показано ниже:


```

class Car
{
    // 'Состояние' объекта Car.
    public string petName;
    public int currSpeed;

    // Специальный стандартный конструктор.
    public Car()
    {
        petName = "Chuck";
        currSpeed = 10;
    }
    ...
}

```

В данном случае мы заставляем объекты Car начинать свою жизнь под именем Chuck и скоростью 10 миль в час. При этом создавать объекты Car со стандартными значениями можно следующим образом:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Class Types *****\n");
    // Вызов стандартного конструктора.
    Car chuck = new Car();
    // Выводит "Chuck is going 10 MPH."
    chuck.PrintState();
    ...
}

```

Определение специальных конструкторов

Обычно помимо стандартного конструктора в классах определяются дополнительные конструкторы. При этом пользователь объекта обеспечивается простым и согласованным способом инициализации состояния объекта непосредственно в момент его создания. Взгляните на следующее изменение класса Car, который теперь поддерживает целых три конструктора:

```

class Car
{
    // 'Состояние' объекта Car.
    public string petName;
    public int currSpeed;
    // Специальный стандартный конструктор.
    public Car()
    {
        petName = "Chuck";
        currSpeed = 10;
    }
    // Здесь currSpeed получает стандартное значение для типа int (0).
    public Car(string pn)
    {
        petName = pn;
    }
    // Позволяет вызывающему коду установить полное состояние Car.
    public Car(string pn, int cs)
    {
        petName = pn;
        currSpeed = cs;
    }
    ...
}

```

Имейте в виду, что один конструктор отличается от другого (с точки зрения компилятора C#) количеством и типом аргументов. В главе 4 было показано, что определение методов с одним и тем же именем, но разным количеством и типами аргументов, называется *перегрузкой*. Таким образом, класс `Car` имеет перегруженный конструктор, чтобы предоставить несколько способов создания объекта во время объявления. В любом случае, теперь можно создавать объекты `Car`, используя любой из его открытых конструкторов. Например:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Class Types *****\n");
    // Создать объект Car по имени Chuck со скоростью 10 миль в час.
    Car chuck = new Car();
    chuck.PrintState();

    // Создать объект Car по имени Mary со скоростью 0 миль в час.
    Car mary = new Car("Mary");
    mary.PrintState();

    // Создать объект Car по имени Daisy со скоростью 75 миль в час.
    Car daisy = new Car("Daisy", 75);
    daisy.PrintState();
}
```

Еще раз о стандартном конструкторе

Как вы только что узнали, все классы бесплатно снабжаются стандартным конструктором. Таким образом, если добавить в текущий проект новый класс по имени `Motorcycle`, определенный следующим образом:

```
class Motorcycle
{
    public void PopAWheely()
    {
        Console.WriteLine("Yeeeeeeee Haaaaaeewww!");
    }
}
```

то сразу можно будет создавать экземпляры `Motorcycle` с помощью стандартного конструктора:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Class Types *****\n");
    Motorcycle mc = new Motorcycle();
    mc.PopAWheely();
}
```

Однако, как только определен специальный конструктор с любым количеством параметров, стандартный конструктор молча удаляется из класса и становится недоступным! Воспринимайте это так: если вы не определили специальный конструктор, то компилятор C# снабжает класс стандартным конструктором, чтобы позволить пользователю объекта размещать его в памяти с набором полей данных, которые имеют корректные стандартные значения. В случае же, когда вы определяете уникальный конструктор, компилятор предполагает, что вы решили взять власть в свои руки.

Таким образом, чтобы позволить пользователю объекта создавать экземпляр типа посредством стандартного конструктора, а также специального конструктора, понадобится явно переопределить стандартный конструктор. И, наконец, в подавляющем большинстве случаев реализация стандартного конструктора класса намеренно остав-

ляется пустой, поскольку все, что требуется — это создание объекта со стандартными значениями всех полей. Внесем в класс `Motorcycle` следующие изменения:

```
class Motorcycle
{
    public int driverIntensity;
    public void PopAWheely()
    {
        for (int i = 0; i <= driverIntensity; i++)
        {
            Console.WriteLine("Yeeeeeeee Haaaaaaewww!");
        }
    }

    // Вернуть стандартный конструктор, который будет устанавливать
    // для всех членов данных стандартные значения.
    public Motorcycle() {}

    // Специальный конструктор.
    public Motorcycle(int intensity)
    {
        driverIntensity = intensity;
    }
}
```

На заметку! Теперь, когда вы лучше понимаете роль конструкторов класса, взгляните на одно удобное сокращение. IDE-среда Visual Studio предоставляет фрагмент кода `ctor`. Когда вы набираете слово `ctor` и два раза нажимаете клавишу <Tab>, IDE-среда автоматически определит специальный стандартный конструктор! Затем можно добавить нужные параметры и логику реализации. Попробуйте это.

Роль ключевого слова `this`

В языке C# имеется ключевое слово `this`, которое обеспечивает доступ к текущему экземпляру класса. Одно из возможных применений ключевого слова `this` состоит в том, чтобы разрешать неоднозначность контекста, которая может возникнуть, когда входящий параметр назван так же, как поле данных конкретного класса. Разумеется, в идеале необходимо просто придерживаться соглашения об именовании, которое не может привести к такой неоднозначности; тем не менее, чтобы проиллюстрировать такое использование ключевого слова `this`, добавим в класс `Motorcycle` новое поле типа `string` (по имени `name`), представляющее имя водителя. После этого добавим метод по имени `SetDriverName()`, реализованный следующим образом:

```
class Motorcycle
{
    public int driverIntensity;

    // Новые члены для представления имени водителя.
    public string name;
    public void SetDriverName(string name)
    { name = name; }
    ...
}
```

Хотя этот код нормально компилируется, Visual Studio отобразит предупреждающее сообщение о том, что переменная присваивается сама себе! Чтобы проиллюстрировать это, добавим в `Main()` вызов `SetDriverName()` и выведем значение поля `name`. Обнаружится, что значением поля `name` осталась пустая строка!

```
// Усадим на Motorcycle байкера по имени Tiny?
Motorcycle c = new Motorcycle(5);
c.SetDriverName("Tiny");
c.PopAWheely();
Console.WriteLine("Rider name is {0}", c.name); // Выводит пустое значение name!
```

Проблема в том, что реализация `SetDriverName()` выполняет присваивание входящему параметру значения *его* самого, поскольку компилятор предполагает, что `name` здесь ссылается на переменную, существующую в контексте метода, а не на поле `name` в контексте класса. Для информирования компилятора о том, что необходимо установить значение поля данных текущего объекта, просто примените `this` для разрешения этой неоднозначности:

```
public void SetDriverName(string name)
{
    this.name = name;
}
```

Имейте в виду, что если неоднозначности нет, то вы не обязаны использовать ключевое слово `this`, когда классу нужно обращаться к собственным данным или членам. Например, если переименовать член данных `name` типа `string` в `driverName` (что также потребует обновления метода `Main()`), то применение `this` станет не обязательным, поскольку исчезает неоднозначность контекста:

```
class Motorcycle
{
    public int driverIntensity;
    public string driverName;

    public void SetDriverName(string name)
    {
        // Эти два оператора функционально эквивалентны.
        driverName = name;
        this.driverName = name;
    }
    ...
}
```

Помимо небольшого выигрыша от использования `this` в неоднозначных ситуациях, это ключевое слово может быть полезно при реализации членов, поскольку такие IDE-среды, как `SharpDevelop` и `Visual Studio`, включают средство `IntelliSense`, когда вводится `this`. Это может здорово помочь, когда вы забыли название члена класса и хотите быстро вспомнить его определение. Взгляните на рис. 5.2.

Построение цепочки вызовов конструкторов с использованием `this`

Другое применение ключевого слова `this` состоит в проектировании класса, использующего прием под названием *сцепление конструкторов* или *построение цепочки конструкторов*. Этот шаблон проектирования полезен, когда имеется класс, определяющий несколько конструкторов. Учитывая тот факт, что конструкторы часто проверяют входящие аргументы на соблюдение различных бизнес-правил, возникает необходимость в избыточной логике проверки достоверности внутри множества конструкторов. Рассмотрим следующее измененное объявление класса `Motorcycle`:

```
class Motorcycle
{
    public int driverIntensity;
    public string driverName;
```

```

public Motorcycle() { }
// Избыточная логика конструктора!
public Motorcycle(int intensity)
{
    if (intensity > 10)
    {
        intensity = 10;
    }
    driverIntensity = intensity;
}

public Motorcycle(int intensity, string name)
{
    if (intensity > 10)
    {
        intensity = 10;
    }
    driverIntensity = intensity;
    driverName = name;
}
...
}

```

Здесь (возможно, стараясь обеспечить безопасность гонщика) в каждом конструкторе предпринимается проверка, что уровень мощности не превышает 10. Хотя все это правильно и хорошо, в двух конструкторах появляется избыточный код. Это далеко от идеала, поскольку придется менять код в нескольких местах в случае изменения правил (например, если предельное значение мощности будет установлено равным 5).

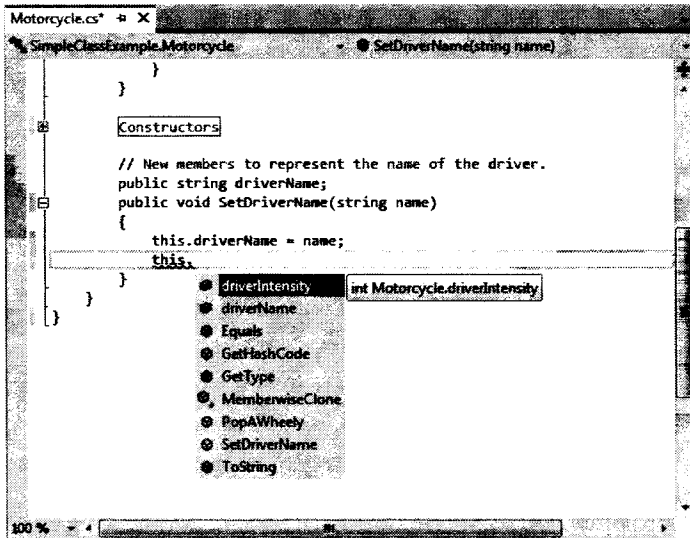


Рис. 5.2. Активизация средства IntelliSense для this

Один из способов исправить создавшуюся ситуацию состоит в определении в классе Motorcycle метода, который выполнит проверку входных аргументов. Если поступить так, то каждый конструктор должен будет вызывать этот метод перед присваиванием значений полям. Хотя такой подход позволяет изолировать код, который придется обновлять при изменении бизнес-правил, теперь появилась другая избыточность:

```

class Motorcycle
{
    public int driverIntensity;
    public string driverName;

    // Конструкторы.
    public Motorcycle() { }
    public Motorcycle(int intensity)
    {
        SetIntensity(intensity);
    }
    public Motorcycle(int intensity, string name)
    {
        SetIntensity(intensity);
        driverName = name;
    }
    public void SetIntensity(int intensity)
    {
        if (intensity > 10)
        {
            intensity = 10;
        }
        driverIntensity = intensity;
    }
    ...
}

```

Более ясный подход предусматривает назначение конструктора, который принимает *максимальное количество аргументов*, в качестве “главного конструктора”, с реализацией внутри него необходимой логики проверки достоверности. Остальные конструкторы смогут использовать ключевое слово `this`, чтобы передать входные аргументы главному конструктору и при необходимости предоставить любые дополнительные параметры. В результате беспокоиться придется только о поддержке единственного конструктора для всего класса, в то время как остальные конструкторы останутся в основном пустыми.

Ниже приведена финальная реализация класса `Motorcycle` (с дополнительным конструктором в целях иллюстрации). При связывании конструкторов в цепочку обратите внимание, что ключевое слово `this` располагается вне тела конструктора (и отделяется от его имени двоеточием):

```

class Motorcycle
{
    public int driverIntensity;
    public string driverName;

    // Связывание конструкторов в цепочку.
    public Motorcycle() {}
    public Motorcycle(int intensity)
        : this(intensity, "") {}
    public Motorcycle(string name)
        : this(0, name) {}

    // Это 'главный конструктор', выполняющий всю реальную работу.
    public Motorcycle(int intensity, string name)
    {
        if (intensity > 10)
        {
            intensity = 10;
        }
    }
}

```

```

        driverIntensity = intensity;
        driverName = name;
    }
    ...
}

```

Имейте в виду, что применение ключевого слова `this` для связывания вызовов конструкторов в цепочку вовсе не обязательно. Однако использование такого приема позволяет получить лучше сопровождаемое и более краткое определение кода. С помощью этой техники также можно упростить решение программистских задач, поскольку реальная работа делегируется единственному конструктору (обычно имеющему максимальное количество параметров), в то время как остальные просто передают ему ответственность.

На заметку! Вспомните из главы 4, что в C# поддерживаются необязательные параметры. С использованием необязательных параметров в конструкторах классов можно добиться тех же преимуществ, что и при связывании конструкторов в цепочку, но со значительно меньшим объемом кода. Очень скоро будет показано, как это сделать.

Обзор потока конструктора

Напоследок отметим, что как только конструктор передал аргументы выделенному главному конструктору (и этот конструктор обработал данные), вызывающий конструктор продолжает выполнение всех остальных операторов. Чтобы прояснить мысль, модифицируем конструкторы класса `Motorcycle`, добавив вызов `Console.WriteLine()`:

```

class Motorcycle
{
    public int driverIntensity;
    public string driverName;

    // Связывание конструкторов в цепочку.
    public Motorcycle()
    {
        Console.WriteLine("In default ctor");
    }
    public Motorcycle(int intensity)
        : this(intensity, "")
    {
        Console.WriteLine("In ctor taking an int");
    }
    public Motorcycle(string name)
        : this(0, name)
    {
        Console.WriteLine("In ctor taking a string");
    }

    // Это 'главный конструктор', выполняющий всю реальную работу.
    public Motorcycle(int intensity, string name)
    {
        Console.WriteLine("In master ctor ");
        if (intensity > 10)
        {
            intensity = 10;
        }
        driverIntensity = intensity;
        driverName = name;
    }
    ...
}

```

Теперь изменим метод `Main()`, чтобы он обрабатывал объект `Motorcycle`, как показано ниже:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Class Types *****\n");
    // Создание Motorcycle.
    Motorcycle c = new Motorcycle(5);
    c.SetDriverName("Tiny");
    c.PopAWheely();
    Console.WriteLine("Rider name is {0}", c.driverName); // вывод имени гонщика
    Console.ReadLine();
}
```

Вывод, полученный в результате выполнения предыдущего метода `Main()`, выглядит следующим образом:

```
***** Fun with Class Types *****
In master ctor
In ctor taking an int
Yeeeeeeee Haaaaaeewww!
Yeeeeeeee Haaaaaeewww!
Yeeeeeeee Haaaaaeewww!
Yeeeeeeee Haaaaaeewww!
Yeeeeeeee Haaaaaeewww!
Yeeeeeeee Haaaaaeewww!
Yeeeeeeee Haaaaaeewww!
Rider name is Tiny
```

Поток логики конструкторов описан ниже.

- Сначала создается объект за счет вызова конструктора, который принимает один аргумент типа `int`.
- Конструктор передает полученные данные главному конструктору и предоставляет необходимые дополнительные начальные аргументы, не указанные вызывающим кодом.
- Главный конструктор присваивает входные данные полям данных объекта.
- Управление возвращается первоначально вызванному конструктору, который выполняет остальные операторы кода.

В построении цепочек конструкторов замечательно то, что этот шаблон программирования работает с любой версией языка C# и платформой .NET. Однако в случае если целевой платформой является .NET 4.0 или последующая версия, можно еще более упростить задачу программирования за счет использования необязательных аргументов в качестве альтернативы традиционным цепочкам конструкторов.

Еще раз о необязательных аргументах

Вы в главе 4 узнали о необязательных и именованных аргументах. Вспомните, что необязательные аргументы позволяют определять стандартные значения для входных аргументов. Если вызывающий код удовлетворяет эти стандартные значения, указывать уникальные значения не обязательно, но это можно делать, когда объект требуется снабдить специальными данными. Рассмотрим следующую версию класса `Motorcycle`, который теперь предоставляет несколько возможностей конструирования объектов, используя *единственное* определение конструктора.


```

class Motorcycle
{
    // Единственный конструктор, использующий необязательные аргументы.
    public Motorcycle(int intensity = 0, string name = "")
    {
        if (intensity > 10)
        {
            intensity = 10;
        }
        driverIntensity = intensity;
        driverName = name;
    }
    ...
}

```

С помощью этого единственного конструктора можно создать объект `Motorcycle`, указывая ноль, один или два аргумента. Вспомните, что синтаксис именованных аргументов позволяет, в сущности, пропускать приемлемые стандартные установки (см. главу 4).

```

static void MakeSomeBikes()
{
    // driverName = "", driverIntensity = 0
    Motorcycle m1 = new Motorcycle();
    Console.WriteLine("Name= {0}, Intensity= {1}",
        m1.driverName, m1.driverIntensity);

    // driverName = "Tiny", driverIntensity = 0
    Motorcycle m2 = new Motorcycle(name: "Tiny");
    Console.WriteLine("Name= {0}, Intensity= {1}",
        m2.driverName, m2.driverIntensity);

    // driverName = "", driverIntensity = 7
    Motorcycle m3 = new Motorcycle(7);
    Console.WriteLine("Name= {0}, Intensity= {1}",
        m3.driverName, m3.driverIntensity);
}

```

Хотя применение необязательных/именованных аргументов — очень удобный путь упрощения определения набора конструкторов, используемых заданным классом, следует всегда помнить, что этот синтаксис является допустимым только в .NET 4.0 или последующих версиях. Если требуется строить классы, которые должны выполняться на платформе .NET любой версии, лучше придерживаться классической техники цепочек конструкторов.

В любом случае теперь можно определить класс с полями данными (т.е. переменными-членами) и различными операциями, такими как методы и конструкторы. Теперь давайте формально рассмотрим роль ключевого слова `static`.

Исходный код. Проект `SimpleClassExample` доступен в подкаталоге `Chapter 05`.

Понятие ключевого слова `static`

Класс C# может определять любое количество *статических членов*, которые объявляются с использованием ключевого слова `static`. При этом соответствующий член должен вызываться непосредственно на уровне класса, а не на переменной, хранящей ссылку на объект. Чтобы проиллюстрировать разницу, обратимся к `System.Console`. Как уже было показано, метод `WriteLine()` не вызывается на уровне объекта:

```
// Ошибка! WriteLine() не является методом уровня объекта!
Console c = new Console();
c.WriteLine("I can't be printed...");
```

Вместо этого статический член `WriteLine()` предворяется именем класса:

```
// Правильно! WriteLine() – статический метод.
Console.WriteLine("Thanks...");
```

Проще говоря, статические члены — это элементы, задуманные (проектировщиком класса) как общие, так что нет нужды создавать экземпляр класса перед их вызовом. Хотя в любом классе можно определять статические члены, чаще всего их можно обнаружить внутри “обслуживающих классов”. По определению обслуживающий класс — это такой класс, который поддерживает состояние на уровне объектов и не создается посредством ключевого слова `new`. Вместо этого обслуживающий класс открывает всю функциональность в виде членов уровня класса (т.е. статических).

Например, если воспользоваться браузером объектов Visual Studio (выбрав пункт меню `View⇒Object Browser` (Вид⇒Браузер объектов)) для просмотра пространства имен `System` из сборки `mscorlib.dll`, можно увидеть, что все члены классов `Console`, `Math`, `Environment` и `GC` (и ряд других) открывают свою функциональность через статические члены. Это лишь несколько обслуживающих классов, определенных в библиотеках базовых классов .NET.

Опять-таки, следует помнить, что статические члены могут находиться не только в обслуживающих классах; они могут быть частью любого определения класса. Просто не забывайте, что статические члены перемещают заданный элемент на уровень класса, а не объекта. Как будет показано в нескольких последующих разделах, ключевое слово `static` может быть применено к следующим конструкциям:

- данные класса;
- методы класса;
- свойства класса;
- конструктор;
- целое определение класса.

Давайте рассмотрим все варианты, начав с концепции статических данных.

На заметку! Роль статических свойств будет объясняться позже в этой главе, во время исследования самих свойств.

Определение статических полей данных

Большую часть времени при проектировании класса данные определяются на уровне экземпляра; говоря иначе, это нестатические данные. Когда определяются данные уровня экземпляра, известно, что при каждом создании нового объекта этот объект поддерживает собственную независимую копию таких данных. В противоположность этому, если определены *статические* данные класса, эта память разделяется всеми объектами соответствующей категории.

Чтобы увидеть разницу, создадим новый проект консольного приложения по имени `StaticDataAndMembers` и вставим в него новый класс под названием `SavingsAccount`. Начнем с определения элемента данных уровня экземпляра (для моделирования текущего баланса) и специального конструктора для установки начального баланса:

```
// Простой класс депозитного счета.
class SavingsAccount
{
    // Данные уровня экземпляра.
    public double currBalance;
    public SavingsAccount(double balance)
    {
        currBalance = balance;
    }
}
```

При создании объектов `SavingsAccount` память под поле `currBalance` выделяется для каждого объекта. Таким образом, можно было бы создать пять разных объектов `SavingsAccount`, каждый с собственным уникальным балансом. Более того, если вы измените баланс на каком-нибудь одном счету, другие объекты не будут затронуты.

С другой стороны, статические данные распределяются однажды и разделяются всеми объектами того же самого класса. Добавьте в класс `SavingsAccount` статический элемент данных по имени `currInterestRate`, принимающий стандартное значение 0.04:

```
// Простой класс депозитного счета.
class SavingsAccount
{
    // Данные уровня экземпляра.
    public double currBalance;
    // Статический элемент данных.
    public static double currInterestRate = 0.04;
    public SavingsAccount(double balance)
    {
        currBalance = balance;
    }
}
```

Если создать три экземпляра `SavingsAccount`, как показано ниже:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Static Data *****\n");
    SavingsAccount s1 = new SavingsAccount(50);
    SavingsAccount s2 = new SavingsAccount(100);
    SavingsAccount s3 = new SavingsAccount(10000.75);
    Console.ReadLine();
}
```

то размещение данных в памяти будет выглядеть примерно так, как показано на рис. 5.3.

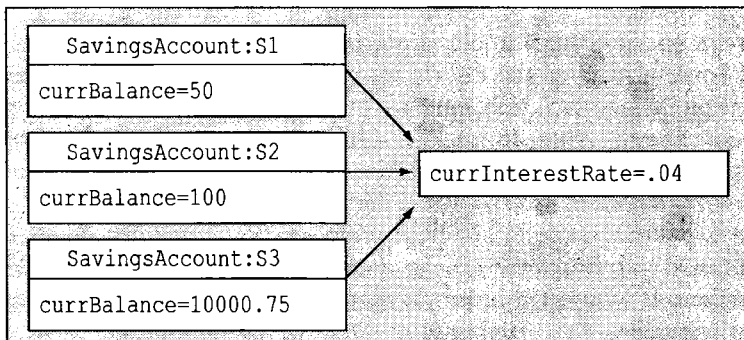


Рис. 5.3. Статические данные размещаются один раз и разделяются между всеми экземплярами класса

Здесь мы предполагаем, что все депозитные счета должны иметь одну и ту же процентную ставку. Поскольку статические данные разделяются всеми объектами той же самой категории, если вы измените процентную ставку каким-либо образом, то все объекты будут “видеть” новое значение при следующем доступе к статическим данным, поскольку все они, в сущности, просматривают одно и то же местоположение в памяти. Чтобы понять, как изменять (или получать) статические данные, понадобится рассмотреть роль статических методов.

Определение статических методов

Давайте изменим класс `SavingsAccount`, добавив к нему два статических метода. Первый статический метод (`GetInterestRate()`) будет возвращать текущую процентную ставку, а второй (`SetInterestRate()`) позволит изменять процентную ставку:

```
// Простой класс депозитного счета.
class SavingsAccount
{
    // Данные уровня экземпляра.
    public double currBalance;

    // Статический элемент данных.
    public static double currInterestRate = 0.04;

    public SavingsAccount(double balance)
    {
        currBalance = balance;
    }

    // Статические члены для установки/получения процентной ставки.
    public static void SetInterestRate(double newRate)
    {
        currInterestRate = newRate;
    }

    public static double GetInterestRate()
    {
        return currInterestRate;
    }
}
```

Рассмотрим следующий сценарий использования класса:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Static Data *****\n");
    SavingsAccount s1 = new SavingsAccount(50);
    SavingsAccount s2 = new SavingsAccount(100);

    // Вывести текущую процентную ставку.
    Console.WriteLine("Interest Rate is: {0}", SavingsAccount.GetInterestRate());

    // Создать новый объект; это не 'сбросит' процентную ставку.
    SavingsAccount s3 = new SavingsAccount(10000.75);
    Console.WriteLine("Interest Rate is: {0}", SavingsAccount.GetInterestRate());
    Console.ReadLine();
}
```

Вывод предыдущего метода `Main()` показан ниже:

```
***** Fun with Static Data *****
Interest Rate is: 0.04
Interest Rate is: 0.04
```

Как видите, при создании новых экземпляров класса `SavingsAccount` значение статических данных не сбрасывается, поскольку CLR выделяет для них место в памяти только один раз. После этого все объекты типа `SavingsAccount` оперируют одним и тем же значением в статическом поле `currInterestRate`.

При проектировании любого класса C# одна из задач связана с выяснением того, какие части данных должны быть определены как статические члены, а какие — нет. Хотя на этот счет не существует строгих правил, помните, что поле статических данных разделяется между всеми объектами конкретного класса. Поэтому, если необходимо, чтобы часть данных совместно использовалась всеми объектами, то статические члены будут самым подходящим вариантом.

Предположим, что переменная `currInterestRate` не определена с ключевым словом `static`. Это означает, что каждый объект `SavingsAccount` имеет собственную копию `currInterestRate`. Пусть создано 100 объектов `SavingsAccount`, и нужно изменить значение процентной ставки. Это потребует стократного вызова метода `SetInterestRate()`! Ясно, что такой способ моделирования общих для объектов класса данных нельзя считать удобным. Еще раз: статические данные идеальны, когда имеет-ся значение, которое должно быть общим для всех объектов отдельной категории.

На заметку! Ссылка на нестатические члены внутри реализации статического члена приводит к ошибке компиляции. Вдобавок следует отметить, что ошибкой будет и применение операции `this` в статическом члене, поскольку она подразумевает объект!

Определение статических конструкторов

Типичный конструктор используется для установки значений данных уровня экземпляра в объекте во время его создания. Однако что случится, если вы попытаетесь присвоить значение статическому элементу данных в типичном конструкторе? Вас может удивить, когда обнаружится, что это значение сбрасывается каждый раз, когда создается новый объект!

В целях иллюстрации предположим, что конструктор класса `SavingsAccount` изменен, как показано ниже (также обратите внимание, что поле `currInterestRate` больше не устанавливается при объявлении):

```
class SavingsAccount
{
    public double currBalance;
    public static double currInterestRate;

    public SavingsAccount(double balance)
    {
        currInterestRate = 0.04; // Это статические данные!
        currBalance = balance;
    }
    ...
}
```

Теперь рассмотрим следующий код в методе `Main()`:

```
static void Main( string[] args )
{
    Console.WriteLine("***** Fun with Static Data *****\n");
    // Создать счет.
    SavingsAccount s1 = new SavingsAccount(50);
    // Вывести текущую процентную ставку.
    Console.WriteLine("Interest Rate is: {0}", SavingsAccount.GetInterestRate());
}
```

```
// Попытаться изменить процентную ставку через свойство.
SavingsAccount.SetInterestRate(0.08);

// Создать второй счет.
SavingsAccount s2 = new SavingsAccount(100);

// Должно вывестись 0.08... не так ли??
Console.WriteLine("Interest Rate is: {0}", SavingsAccount.GetInterestRate());
Console.ReadLine();
}
```

При выполнении предыдущего метода `Main()` обнаруживается, что переменная `currInterestRate` будет сбрасываться при каждом создании нового объекта `SavingsAccount`, всегда возвращаясь к значению `0.04`. Ясно, что установка значений статических данных в нормальном конструкторе уровня экземпляра сводит на нет весь их смысл. Всякий раз, когда создается новый объект, данные уровня класса сбрасываются! Один из способов правильной установки статического поля состоит в использовании синтаксиса инициализации члена, как это делалось изначально:

```
class SavingsAccount
{
    public double currBalance;

    // Статические данные.
    public static double currInterestRate = 0.04;
    ...
}
```

Этот подход гарантирует, что статическое поле будет установлено только однажды, независимо от того, сколько объектов будет создано. Однако что, если значение статических данных нужно получить во время выполнения? Например, в типичном банковском приложении значение переменной, представляющей процентную ставку, должно быть прочитано из базы данных или внешнего файла. Решение подобных задач требует контекста метода (такого как конструктор), чтобы можно было выполнить операторы кода.

Именно по этой причине в С# предусмотрена возможность определения *статического конструктора*, который позволяет безопасно устанавливать значения статических данных. Взгляните на следующее изменение в классе:

```
class SavingsAccount
{
    public double currBalance;
    public static double currInterestRate;

    public SavingsAccount(double balance)
    {
        currBalance = balance;
    }

    // Статический конструктор.
    static SavingsAccount()
    {
        Console.WriteLine("In static ctor!");
        currInterestRate = 0.04;
    }
    ...
}
```

Выражаясь упрощенно, статический конструктор — это специальный конструктор, который является идеальным местом для инициализации значений статических данных, когда их значение не известно на момент компиляции (например, когда его нужно прочитать из внешнего файла, базы данных, сгенерировать случайное число или еще

каким-то образом получить значение). Если запустить заново предыдущий метод `Main()`, вы увидите ожидаемый вывод. Обратите внимание, что сообщение "In static ctor!" выводится только один раз, поскольку среда CLR вызывает все статические конструкторы перед первым использованием (и никогда не вызывает их повторно для этого экземпляра приложения):

```
***** Fun with Static Data *****
```

```
In static ctor!
```

```
Interest Rate is: 0.04
```

```
Interest Rate is: 0.08
```

Ниже приведено несколько интересных моментов, касающихся статических конструкторов.

- В отдельном классе может быть определен только один статический конструктор. Другими словами, статический конструктор нельзя перегружать.
- Статический конструктор не имеет модификатора доступа и не может принимать параметров.
- Статический конструктор выполняется только один раз, независимо от того, сколько объектов отдельного класса создается.
- Исполняющая система вызывает статический конструктор, когда создает экземпляр класса или перед первым обращением к статическому члену этого класса.
- Статический конструктор выполняется перед любым конструктором уровня экземпляра.

Учитывая сказанное, при создании новых объектов `SavingsAccount` значения статических данных сохраняются, поскольку статический член устанавливается только один раз внутри статического конструктора, независимо от количества созданных объектов.

Исходный код. Проект `StaticDataAndMembers` доступен в подкаталоге `Chapter 05`.

Определение статических классов

Ключевое слово `static` допускается также применять прямо на уровне класса. Когда класс определен как статический, его экземпляры нельзя создавать с использованием ключевого слова `new`, и он может включать в себя только члены или поля данных, помеченные ключевым словом `static`. Если это правило нарушить, возникнет ошибка компиляции.

На заметку! Помните, что класс (или структуру), который открывает только статическую функциональность, часто называют обслуживающим. При проектировании обслуживающего класса рекомендуется применять ключевое слово `static` к определению класса.

На первый взгляд это может показаться довольно странным средством, учитывая невозможность создания экземпляров класса. Однако следует учесть, что класс, не содержащий ничего кроме статических членов и/или константных данных, и не нуждается в выделении памяти. В целях иллюстрации создадим новое консольное приложение по имени `SimpleUtilityClass`. Далее определим следующий класс:

```
// Статические классы могут содержать только статические члены!
static class TimeUtilClass
{
```

```

public static void PrintTime()
{ Console.WriteLine(DateTime.Now.ToShortTimeString()); }

public static void PrintDate()
{ Console.WriteLine(DateTime.Today.ToShortDateString()); }
}

```

Учитывая, что этот класс определен с ключевым словом `static`, создавать экземпляры `TimeUtilClass` с помощью ключевого слова `new` нельзя. Вместо этого вся функциональность доступна на уровне класса:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Static Classes *****\n");

    // Это работает нормально.
    TimeUtilClass.PrintDate();
    TimeUtilClass.PrintTime();

    // Ошибка компиляции! Создавать экземпляр статического класса нельзя!
    TimeUtilClass u = new TimeUtilClass ();

    Console.ReadLine();
}

```

К этому месту главы уже должно быть ясно, как определять простые классы, включающие конструкторы, поля и различные статические (и нестатические) члены. Обладая такими базовыми знаниями о конструкции классов, можно приступить к ознакомлению с тремя основными принципами объектно-ориентированного программирования.

Исходный код. Проект `SimpleUtilityClass` доступен в подкаталоге `Chapter 05`.

Основные принципы объектно-ориентированного программирования

Все объектно-ориентированные языки (C#, Java, C++, Smalltalk, Visual Basic и т.п.) должны отвечать трем основным принципам объектно-ориентированного программирования (ООП), которые перечислены ниже.

1. *Инкапсуляция.* Как данный язык скрывает детали внутренней реализации объектов и предохраняет целостность данных?
2. *Наследование.* Как данный язык стимулирует многократное использование кода?
3. *Полиморфизм.* Как данный язык позволяет трактовать связанные объекты сходным образом?

Прежде чем погрузиться в синтаксические детали реализации указанных принципов, важно понять базовую роль каждого из них. Ниже предлагается обзор всех принципов, а в остальной части этой и последующих главах рассматриваются необходимые подробности.

Роль инкапсуляции

Первый основной принцип ООП называется *инкапсуляцией*. Этот принцип касается способности языка скрывать излишние детали реализации от пользователя объекта. Например, предположим, что используется класс по имени `DaabaseReader`, который имеет два главных метода: `Open()` и `Close()`.


```
// Этот класс инкапсулирует детали открытия и закрытия базы данных.
DatabaseReader dbReader = new DatabaseReader();
dbReader.Open(@"C:\AutoLot.mdf");

// Сделать что-то с файлом данных и закрыть файл.
dbReader.Close();
```

Фиктивный класс `DatabaseReader` инкапсулирует внутренние детали нахождения, загрузки, манипуляций и закрытия файла данных. Программистам нравится инкапсуляция, поскольку этот принцип ООП упрощает кодирование. Нет необходимости беспокоиться о многочисленных строках кода, которые работают “за кулисами”, чтобы реализовать функционирование класса `DatabaseReader`. Все, что потребуется — это создать экземпляр и отправить ему соответствующие сообщения (например, “открыть файл по имени `AutoLot.mdf`, расположенный на диске `C:`”).

С идеей инкапсуляции программной логики тесно связана идея защиты данных. В идеале данные состояния объекта должны быть определены с использованием ключевого слова `private` (или, возможно, `protected`). Таким образом, внешний мир должен вежливо попросить о возможности изменения или получения лежащего в основе значения. Это хороший принцип, поскольку открыто объявленные элементы данных можно легко повредить (даже нечаянно, а не преднамеренно). Чуть позже будет дано формальное определение этого аспекта инкапсуляции.

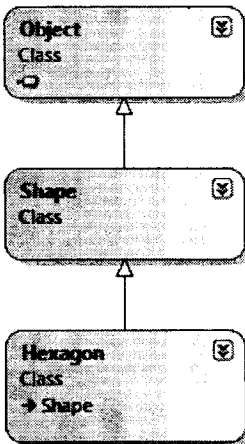


Рис. 5.4. Отношение “является” (“is-a”)

Роль наследования

Следующий принцип ООП — *наследование* — касается способности языка позволять строить новые определения классов на основе определений существующих классов. По сути, наследование позволяет расширять поведение базового (или *родительского*) класса, наследуя его основную функциональность в производном подклассе (также именуемом *дочерним классом*). На рис. 5.4 показан простой пример.

Прочсть диаграмму на рис. 5.4 можно так: “шестиугольник (Hexagon) является фигурой (Shape), которая является объектом (Object)”. При наличии классов, связанных этой формой наследования, между типами устанавливается *отношение “является” (“is-a”)*. Такое отношение и называют *наследованием*.

Здесь можно предположить, что Shape определяет некоторое количество членов, общих для всех наследников (скажем, значение для представления цвета фигуры и другие значения, задающие высоту и ширину). Учитывая, что класс Hexagon расширяет Shape, он наследует основную функциональность, определенную классами Shape и Object, а также определяет дополнительные собственные детали, касающиеся шестиугольников (какими бы они ни были).

На заметку! На платформе .NET класс `System.Object` всегда находится на вершине любой иерархии классов, являясь главным родительским классом, и определяет общую функциональность для всех типов (как показано в главе 6).

В мире ООП существует и другая форма повторного использования кода: модель включения/делегации, также известная под названием *отношение “имеет” (“has-a”)* или *агрегация*. Эта форма повторного использования не применяется для установки отношений “родительский–дочерний”. Вместо этого такое отношение позволяет одному классу определять переменную-член другого класса и опосредованно представлять его функциональность (когда необходимо) пользователю объекта.

Например, предположим, что снова моделируется автомобиль. Может понадобиться выразить идею, что автомобиль “имеет” радиоприемник. Было бы нелогично пытаться наследовать класс Car от Radio или наоборот (ведь Car не “является” Radio). Взамен имеются два независимых класса, работающих совместно, причем класс Car создает и представляет функциональность Radio:

```
class Radio
{
    public void Power(bool turnOn)
    {
        Console.WriteLine("Radio on: {0}", turnOn);
    }
}

class Car
{
    // Car 'имеет' Radio.
    private Radio myRadio = new Radio();
    public void TurnOnRadio(bool onOff)
    {
        // Делегировать вызов внутреннему объекту.
        myRadio.Power(onOff);
    }
}
```

Обратите внимание, что пользователь объекта не имеет понятия, что класс Car использует внутренний объект Radio.

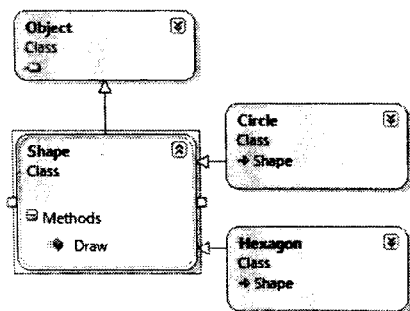
```
static void Main(string[] args)
{
    // Вызов передается Radio внутренне.
    Car viper = new Car();
    viper.TurnOnRadio(false);
}
```

Роль полиморфизма

Последний основной принцип ООП — *полиморфизм*. Он обозначает способность языка трактовать связанные объекты в сходной манере. В частности, этот принцип ООП позволяет базовому классу определять набор членов (формально называемый *полиморфным интерфейсом*), которые доступны всем наследникам. Полиморфный интерфейс класса конструируется с использованием любого количества *виртуальных* или *абстрактных* членов (подробности ищите в главе 6).

В сущности, *виртуальный член* — это член базового класса, определяющий стандартную реализацию, которая может быть изменена (или, говоря более формально, *переопределена*) в производном классе. В отличие от него, *абстрактный метод* — это член базового класса, который *не предусматривает* стандартной реализации, а предлагает только сигнатуру. Когда класс наследуется от базового класса, определяющего абстрактный метод, этот метод *обязательно* должен быть переопределен в производном классе. В любом случае, когда производные классы переопределяют члены, определенные в базовом классе, они по существу переопределяют свою реакцию на один и тот же запрос.

Чтобы увидеть полиморфизм в действии, давайте представим некоторые детали иерархии фигур, показанной на рис. 5.4. Предположим, что в классе Shape определен виртуальный метод Draw(), не принимающий параметров. Учитывая тот факт, что каждая фигура должна визуализировать себя уникальным образом, подклассы (такие как Hexagon и Circle) вольны переопределить этот метод по своему усмотрению (рис. 5.5).



Вызов Draw () на объекте Circle приводит к рисованию двумерного круга.

Вызов Draw () на объекте Hexagon приводит к рисованию двумерного шестиугольника.

Рис. 5.5. Классический полиморфизм

Когда полиморфный интерфейс спроектирован, можно делать ряд предположений в коде. Например, учитывая, что классы Hexagon и Circle унаследованы от общего родителя (Shape), массив типов Shape может содержать всех наследников базового класса. Более того, учитывая, что в Shape определен полиморфный интерфейс для всех производных типов (в данном примере — метод Draw()), можно предположить, что каждый член массива обладает этой функциональностью.

Рассмотрим следующий метод Main(), который заставляет массив типов-наследников Shape визуализировать себя с использованием метода Draw():

```

class Program
{
    static void Main(string[] args)
    {
        Shape[] myShapes = new Shape[3];
        myShapes[0] = new Hexagon();
        myShapes[1] = new Circle();
        myShapes[2] = new Hexagon();

        foreach (Shape s in myShapes)
        {
            // Использовать полиморфный интерфейс!
            s.Draw();
        }
        Console.ReadLine();
    }
}
  
```

На этом краткий обзор основных принципов ООП завершен. Оставшаяся часть главы посвящена дальнейшим подробностям инкапсуляции в C#. Детали наследования и полиморфизма представлены в главе 6.

Модификаторы доступа C#

При работе с инкапсуляцией всегда следует принимать во внимание то, какие аспекты типа видимы различным частям приложения. В частности, типы (классы, интерфейсы, структуры, перечисления и делегаты), а также их члены (свойства, методы, конструкторы и поля) определяются с использованием специфического ключевого слова, которое управляет “видимостью” элемента другим частям приложения. Хотя в C# определены многочисленные ключевые слова для управления доступом, их действие может отличаться в зависимости от места применения (к типу или члену). В табл. 5.1 описана роль и применение модификаторов доступа.

Таблица 5.1. Модификаторы доступа C#

Модификатор доступа	К чему может быть применен	Назначение
<code>public</code>	Типы или члены типов	Открытые (<code>public</code>) элементы не имеют ограничений доступа. Открытый член может быть доступен как из объекта, так и из любого производного класса. Открытый тип может быть доступен из других внешних сборок
<code>private</code>	Члены типов или вложенные типы	Закрытые (<code>private</code>) элементы могут быть доступны только в классе (или структуре), в котором они определены
<code>protected</code>	Члены типов или вложенные типы	Защищенные (<code>protected</code>) элементы могут использоваться классом, который определил их, и любым дочерним классом. Однако защищенные элементы не доступны внешнему миру через операцию точки (.)
<code>internal</code>	Типы или члены типов	Внутренние (<code>internal</code>) элементы доступны только в пределах текущей сборки. Таким образом, если в библиотеке классов .NET определен набор внутренних типов, то другие сборки не смогут ими пользоваться
<code>protected internal</code>	Члены типов или вложенные типы	Когда ключевые слова <code>protected</code> и <code>internal</code> комбинируются в объявлении элемента, такой элемент доступен внутри определяющей его сборки, определяющего класса и всех его наследников

В этой главе рассматриваются только ключевые слова `public` и `private`. В последующих главах будет рассказываться о роли модификаторов `internal` и `protected internal` (удобных при построении библиотек кода .NET) и модификатора `protected` (удобного при создании иерархий классов).

Стандартные модификаторы доступа

По умолчанию члены типов являются *неявно закрытыми* (`private`) и *неявно внутренними* (`internal`). Таким образом, следующее определение класса автоматически установлено как `internal`, в то время как стандартный конструктор этого типа автоматически является `private`:

```
// Внутренний класс с закрытым стандартным конструктором.
class Radio
{
    Radio() {}
}
```

Чтобы позволить другим частям программы обращаться к членам объекта, эти члены потребуются пометить как открытые (`public`). К тому же, если необходимо открыть `Radio` внешним сборкам (опять-таки, это удобно при построении библиотек кода .NET; см. главу 14), следует добавить к нему модификатор `public`.

```
// Открытый класс с закрытым стандартным конструктором.
public class Radio
{
    Radio() {}
}
```

Модификаторы доступа и вложенные типы

Как было показано в табл. 5.1, модификаторы доступа `private`, `protected` и `protected internal` могут применяться к *вложенному типу*. Вложение типов детально рассматривается в главе 6. Пока же достаточно знать, что вложенный тип — это тип, объявленный непосредственно внутри объявления класса или структуры. Для примера ниже приведено закрытое перечисление (по имени `CarColor`), вложенное в открытый класс (по имени `SportsCar`):

```
public class SportsCar
{
    // Нормально! Вложенные типы могут быть помечены как private.
    private enum CarColor
    {
        Red, Green, Blue
    }
}
```

Здесь допускается применять модификатор доступа `private` к вложенному типу. Однако не вложенные типы (вроде `SportsCar`) могут определяться только с модификаторами `public` или `internal`. Поэтому следующее определение класса неверно:

```
// Ошибка! Не вложенный тип не может быть помечен как private!
private class SportsCar
{ }
```

Первый принцип: службы инкапсуляции C#

Концепция инкапсуляции вращается вокруг принципа, гласящего, что внутренние данные объекта не должны быть напрямую доступны через экземпляр объекта. Вместо этого данные класса определяются как закрытые. Если вызывающий код желает изменить состояние объекта, то должен делать непрямо через открытые методы. Чтобы проиллюстрировать необходимость в службах инкапсуляции, предположим, что создано следующее определение класса:

```
// Класс с единственным открытым полем.
class Book
{
    public int numberOfPages;
}
```

Проблема с открытыми данными состоит в том, что сами по себе эти данные не имеют возможности “понять”, является ли присваиваемое значение допустимым в рамках существующих бизнес-правил системы. Как известно, верхний предел значений для типа `int` в C# довольно велик (2 147 483 647), поэтому компилятор разрешит следующее присваивание:

```
// Хм... Ничего себе — мини-новелла!
static void Main(string[] args)
{
    Book miniNovel = new Book();
    miniNovel.numberOfPages = 30000000;
}
```

Хотя границы типа данных `int` не превышены, ясно, что мини-новелла на 30 миллионов страниц выглядит несколько неправдоподобно. Как видите, открытые поля не дают возможности перехватывать ошибки, связанные с преодолением верхних (или нижних) логических границ. Если в текущей системе установлено бизнес-правило, гла-

сящее, что книга должна иметь от 1 до 1000 страниц, его придется обеспечить программно. По этой причине открытым полям обычно нет места в определении класса производственного уровня.

На заметку! Говоря точнее, члены класса, представляющие состояние объекта, не должны помечаться как `public`. В то же время, как будет показано далее в главе, вполне допускается иметь открытые константы и открытые поля, предназначенные только для чтения.

Инкапсуляция предоставляет способ предохранения целостности данных о состоянии объекта. Вместо определения открытых полей (которые легко приводят к повреждению данных), необходимо выработать привычку определения *закрытых данных*, управление которыми осуществляется опосредованно, с применением одного из двух главных приемов:

- определение пары открытых методов доступа и изменения;
- определение открытого свойства `.NET`.

Какой бы прием не был выбран, идея состоит в том, что хорошо инкапсулированный класс должен защищать свои данные и скрывать подробности своего устройства от любопытных глаз из внешнего мира. Это часто называют *программированием черного ящика*. Преимущество такого подхода состоит в том, что объект может свободно изменять внутреннюю реализацию любого метода. За счет обеспечения неизменности сигнатуры метода, работа существующего кода, который использует этот метод, не нарушается.

Инкапсуляция с использованием традиционных методов доступа и изменения

В оставшейся части этой главы будет построен довольно полный класс, моделирующий обычного сотрудника. Для начала создадим новое консольное приложение по имени `EmployeeApp` и добавим в него новый файл класса (под названием `Employee.cs`), используя пункт меню `Project⇒Add class` (Проект⇒Добавить класс). Дополним класс `Employee` следующими полями, методами и конструкторами:

```
class Employee
{
    // Поля данных.
    private string empName;
    private int empID;
    private float currPay;

    // Конструкторы.
    public Employee() {}
    public Employee(string name, int id, float pay)
    {
        empName = name;
        empID = id;
        currPay = pay;
    }

    // Методы.
    public void GiveBonus(float amount)
    {
        currPay += amount;
    }

    public void DisplayStats()
    {

```

```

    Console.WriteLine("Name: {0}", empName);
    Console.WriteLine("ID: {0}", empID);
    Console.WriteLine("Pay: {0}", currPay);
}
}

```

Обратите внимание, что поля класса `Employee` определены с применением ключевого слова `private`. С учетом этого, поля `empName`, `empID` и `currPay` напрямую через объектную переменную не доступны. Следовательно, показанная ниже логика в `Main()` приведет к ошибкам на этапе компиляции:

```

static void Main(string[] args)
{
    // Ошибка! Невозможно напрямую обращаться
    // к закрытым полям объекта!
    Employee emp = new Employee();
    emp.empName = "Marv";
}

```

Если необходимо, чтобы внешний мир мог взаимодействовать с полным именем сотрудника, традиционный подход (который очень распространен в Java) предусматривает определение методов доступа (метод `get`) и изменения (метод `set`). Роль метода `get` состоит в возврате вызывающему коду значения лежащих в основе статических данных. Метод `set` позволяет вызывающему коду изменять текущее значение лежащих в основе статических данных при условии соблюдения бизнес-правил.

В целях иллюстрации давайте инкапсулируем поле `empName`. Для этого к существующему классу `Employee` следует добавить показанные ниже открытые члены. Обратите внимание, что метод `SetName()` выполняет проверку входящих данных, чтобы удостовериться, что строка имеет длину не более 15 символов. Если это не так, на консоль выводится сообщение об ошибке и происходит возврат без изменения значения поля `empName`.

На заметку! В классе производственного уровня внутри логики конструктора следовало бы предусмотреть проверку длины строки с именем сотрудника. Пока опустим эту деталь, но улучшим код позже, при рассмотрении синтаксиса свойств .NET.

```

class Employee
{
    // Поля данных.
    private string empName;
    ...

    // Метод доступа (метод get) .
    public string GetName()
    {
        return empName;
    }

    // Метод изменения (метод set) .
    public void SetName(string name)
    {
        // Перед присваиванием проверить входное значение.
        if (name.Length > 15)
            // Ошибка! Имя должно иметь меньше 16 символов!
            Console.WriteLine("Error! Name must be less than 16 characters!");
        else
            empName = name;
    }
}

```

Эта техника требует наличия двух уникально именованных методов для управления единственным элементом данных. Для тестирования новых методов модифицируем метод `Main()` следующим образом:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Encapsulation *****\n");
    Employee emp = new Employee("Marvin", 456, 30000);
    emp.GiveBonus(1000);
    emp.DisplayStats();

    // Использовать методы get/set для взаимодействия с именем объекта.
    emp.SetName("Marv");
    Console.WriteLine("Employee is named: {0}", emp.GetName());
    Console.ReadLine();
}
```

Благодаря коду в методе `SetName()`, попытка присвоить строку длиннее 16 символов (как показано ниже) приводит к выводу на консоль жестко закодированного сообщения об ошибке:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Encapsulation *****\n");
    ...
    // Длиннее 16 символов! На консоль выводится сообщение об ошибке.
    Employee emp2 = new Employee();
    emp2.SetName("Xena the warrior princess");
    Console.ReadLine();
}
```

Пока все хорошо. Закрытое поле `empName` инкапсулировано с использованием двух открытых методов `GetName()` и `SetName()`. Для дальнейшей инкапсуляции данных в классе `Employee` понадобится добавить ряд дополнительных методов (например, `GetID()`, `SetID()`, `GetCurrentPay()`, `SetCurrentPay()`). Каждый метод, изменяющий данные, может иметь в себе несколько строк кода для проверки дополнительных бизнес-правил. Хотя можно поступить именно так, для инкапсуляции данных класса в C# предлагается удобная альтернативная нотация.

Инкапсуляция с использованием свойств .NET

Вдобавок к возможности инкапсуляции полей данных с использованием традиционной пары методов `get/set`, в языках .NET имеется более предпочтительный способ инкапсуляции данных с помощью *свойств*. Прежде всего, имейте в виду, что свойства — это всего лишь упрощенное представление “реальных” методов доступа и изменения. Это значит, что разработчик класса по-прежнему может реализовать любую внутреннюю логику, которую нужно выполнить перед присваиванием значения (например, преобразовать в верхний регистр, очистить от недопустимых символов, проверить границы числовых значений и т.д.).

Ниже приведен измененный класс `Employee`, который теперь обеспечивает инкапсуляцию каждого поля с применением синтаксиса свойств вместо традиционных методов `get/set`.

```
class Employee
{
    // Поля данных.
    private string empName;
    private int empID;
    private float currPay;
```



```
// Свойства.
public string Name
{
    get { return empName; }
    set
    {
        if (value.Length > 15)
            Console.WriteLine("Error! Name must be less than 16 characters!");
        else
            empName = value;
    }
}
// Можно было бы добавить дополнительные бизнес-правила для установки
// этих свойств, но в данном примере в этом нет необходимости.
public int ID
{
    get { return empID; }
    set { empID = value; }
}
public float Pay
{
    get { return currPay; }
    set { currPay = value; }
}
...
}
```

Свойство C# состоит из определений контекстов `get` (метод доступа) и `set` (метод изменения), вложенных непосредственно в контекст самого свойства. Обратите внимание, что свойство указывает тип данных, которые оно инкапсулирует, как тип возвращаемого значения. Кроме того, в отличие от метода, в определении свойства не используются скобки (даже пустые). Обратите внимание на комментарий к текущему свойству `ID`:

```
// int представляет тип инкапсулируемых свойством данных.
// Тип данных должен быть идентичен связанному полю (empID).
public int ID // Обратите внимание на отсутствие скобок.
{
    get { return empID; }
    set { empID = value; }
}
```

В контексте `set` свойства используется лексема `value`, которая представляет входное значение, присваиваемое свойству вызывающим кодом. Эта лексема не является настоящим ключевым словом C#, а представляет собой то, что называется *контекстным ключевым словом*. Когда лексема `value` находится внутри контекста `set`, она всегда обозначает значение, присваиваемое вызывающим кодом, и всегда имеет тип, совпадающий с типом самого свойства. Поэтому свойство `Name` может проверить допустимую длину `string` следующим образом:

```
public string Name
{
    get { return empName; }
    set
    {
        // Здесь value имеет тип string.
        if (value.Length > 15)
            Console.WriteLine("Error! Name must be less than 16 characters!");
        else
            empName = value;
    }
}
```

При наличии этих свойств вызывающему коду кажется, что он имеет дело с открытым элементом данных; однако “за кулисами” при каждом обращении вызывается корректный блок `get` или `set`, предохраня инкапсуляцию:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Encapsulation *****\n");
    Employee emp = new Employee("Marvin", 456, 30000);
    emp.GiveBonus(1000);
    emp.DisplayStats();

    // Установка и получение свойства Name.
    emp.Name = "Marv";
    Console.WriteLine("Employee is named: {0}", emp.Name);
    Console.ReadLine();
}
```

Свойства (в противоположность методам доступа и изменения) также облегчают манипулирование типами, поскольку способны реагировать на внутренние операции C#. В целях иллюстрации предположим, что тип класса `Employee` имеет внутреннюю закрытую переменную-член, хранящую возраст сотрудника. Ниже показаны необходимые изменения (обратите внимание на использование цепочки вызовов конструкторов):

```
class Employee
{
    ...
    // Новое поле и свойство.
    private int empAge;
    public int Age
    {
        get { return empAge; }
        set { empAge = value; }
    }

    // Обновленные конструкторы.
    public Employee() {}
    public Employee(string name, int id, float pay)
        :this(name, 0, id, pay){}
    public Employee(string name, int age, int id, float pay)
    {
        empName = name;
        empID = id;
        empAge = age;
        currPay = pay;
    }

    // Обновленный метод DisplayStats() теперь учитывает возраст.
    public void DisplayStats()
    {
        Console.WriteLine("Name: {0}", empName);
        Console.WriteLine("ID: {0}", empID);
        Console.WriteLine("Age: {0}", empAge);
        Console.WriteLine("Pay: {0}", currPay);
    }
}
```

Теперь предположим, что создан объект `Employee` по имени `joe`. Необходимо, чтобы в день рождения сотрудника возраст увеличивался на 1 год. Используя традиционные методы `set/get`, пришлось бы написать код вроде следующего:

```
Employee joe = new Employee();
joe.SetAge(joe.GetAge() + 1);
```

Однако если `empAge` инкапсулируется через свойство по имени `Age`, можно записать проще:

```
Employee joe = new Employee();
joe.Age++;
```

Использование свойств внутри определения класса

Свойства, а в особенности их часть `set` — это общепринятое место для размещения бизнес-правил класса. В настоящее время класс `Employee` имеет свойство `Name`, которое гарантирует длину имени не более 15 символов. Остальные свойства (`ID`, `Pay` и `Age`) также могут быть обновлены с добавлением соответствующей логики.

Хотя все это хорошо, но следует принимать во внимание также и то, что обычно происходит внутри конструктора класса. Конструктор получает входные параметры, проверяет корректность данных и затем выполняет присваивания внутренним закрытым полям. В настоящее время главный конструктор не проверяет входные строковые данные на допустимый диапазон, поэтому можно было бы изменить его следующим образом:

```
public Employee(string name, int age, int id, float pay)
{
    // Это может оказаться проблемой...
    if (name.Length > 15)
        Console.WriteLine("Error! Name must be less than 16 characters!");
    else
        empName = name;
    empID = id;
    empAge = age;
    currPay = pay;
}
```

Наверняка вы заметили проблему, связанную с этим подходом. Свойство `Name` и главный конструктор выполняют одну и ту же проверку ошибок! В результате получается дублирование кода. Чтобы упростить код и разместить всю проверку ошибок в центральном месте, для установки и получения данных внутри класса разумно всегда использовать свойства. Ниже показан соответствующим образом обновленный конструктор:

```
public Employee(string name, int age, int id, float pay)
{
    // Уже лучше! Используйте свойства для установки данных класса.
    // Это сократит количество дублированных проверок ошибок.
    Name = name;
    Age = age;
    ID = id;
    Pay = pay;
}
```

Помимо обновления конструкторов с целью использования свойств для присваивания значений, имеет смысл сделать это повсюду в реализации класса, чтобы гарантировать неукоснительное соблюдение бизнес-правил. Во многих случаях единственное место, где можно напрямую обращаться к закрытым данным — это внутри самого свойства. С учетом сказанного модифицируем класс `Employee`, как показано ниже:

```
class Employee
{
    // Поля данных.
    private string empName;
```

```

private int empID;
private float currPay;
private int empAge;

// Конструкторы.
public Employee() { }
public Employee(string name, int id, float pay)
    :this(name, 0, id, pay){}
public Employee(string name, int age, int id, float pay)
{
    Name = name;
    Age = age;
    ID = id;
    Pay = pay;
}

// Методы.
public void GiveBonus(float amount)
{ Pay += amount; }

public void DisplayStats()
{
    Console.WriteLine("Name: {0}", Name);
    Console.WriteLine("ID: {0}", ID);
    Console.WriteLine("Age: {0}", Age);
    Console.WriteLine("Pay: {0}", Pay);
}

// Свойства остаются прежними...
...
}

```

Свойства, доступные только для чтения и только для записи

При инкапсуляции данных может понадобиться сконфигурировать *свойство, доступное только для чтения*. Для этого нужно просто опустить блок `set`. Аналогично, если требуется создать *свойство, доступное только для записи*, следует опустить блок `get`. Например, предположим, что необходимо новое свойство по имени `SocialSecurityNumber`, которое инкапсулирует закрытую строковую переменную `empSSN`. Если нужно получить свойство, доступное только для чтения, можно поступить так:

```

public string SocialSecurityNumber
{
    get { return empSSN; }
}

```

Теперь представим, что конструктор класса имеет новый параметр, чтобы позволить вызывающему коду устанавливать номер карточки социального страхования для объекта. Поскольку свойство `SocialSecurityNumber` допускает только чтение, устанавливать значение, как показано ниже, нельзя:

```

public Employee(string name, int age, int id, float pay, string ssn)
{
    Name = name;
    Age = age;
    ID = id;
    Pay = pay;
    // Теперь это невозможно, поскольку свойство
    // предназначено только для чтения!
    SocialSecurityNumber = ssn;
}

```

Если не планируется делать это свойство доступным как для чтения, так и для записи, единственный выбор предусматривает использование внутри логики конструктора лежащей в основе переменной-члена `empSSN`:

```
public Employee(string name, int age, int id, float pay, string ssn)
{
    ...
    empSSN = ssn;
}
```

В завершение вспомните, что для инкапсуляции данных в C# предпочтение отдается свойствам. Эти синтаксические сущности служат для той же цели, что и традиционные методы для доступа (`get`) и изменения (`set`). Преимущество свойств в том, что пользователи объектов имеют возможность манипулировать внутренним элементом данных с помощью единственной именованной конструкции.

Исходный код. Проект `EmployeeApp` доступен в подкаталоге `Chapter 05`.

Еще раз о ключевом слове `static`: определение статических свойств

Ранее в этой главе рассказывалось о роли ключевого слова `static`. Теперь, когда вы ознакомились с использованием синтаксиса свойств C#, мы можем формализовать статические свойства. В проекте `StaticDataAndMembers` класс `SavingsAccount` имел два открытых статических метода для получения и установки процентной ставки. Однако более стандартный подход предусматривает помещение этого элемента данных в свойство. Таким образом, вместо двух методов для получения и установки процентной ставки можно было определить следующее свойство класса (обратите внимание на использование ключевого слова `static`):

```
// Простой класс депозитного счета.
class SavingsAccount
{
    // Данные уровня экземпляра.
    public double currBalance;

    // Статический элемент данных.
    private static double currInterestRate = 0.04;

    // Статическое свойство.
    public static double InterestRate
    {
        get { return currInterestRate; }
        set { currInterestRate = value; }
    }
    ...
}
```

Чтобы работать с этим свойством вместо предыдущих статических методов, следует модифицировать метод `Main()`, как показано ниже:

```
// Вывести текущую процентную ставку через свойство.
Console.WriteLine("Interest Rate is: {0}", SavingsAccount.InterestRate);
```

Понятие автоматических свойств

При создании свойств для инкапсуляции данных часто обнаруживается, что контексты `set` содержат код для применения бизнес-правил программы. Тем не менее, в некоторых случаях требуется реализовать только простое извлечение или установку значения и никакой другой логики. В конечном итоге получается большой объем кода следующего вида:

```
// Тип Car, использующий стандартный синтаксис свойств.
class Car
{
    private string carName = "";
    public string PetName
    {
        get { return carName; }
        set { carName = value; }
    }
}
```

В таких случаях было бы слишком громоздко многократно определять закрытые поддерживающие поля и простые определения свойств. Например, при построении класса, которому нужно 15 закрытых элементов данных, в конечном итоге получаются 15 связанных с ними свойств, которые представляют собой не более чем тонкие оболочки для служб инкапсуляции.

Чтобы упростить процесс простой инкапсуляции данных полей, можно применять *синтаксис автоматических свойств*. Как следует из названия, это средство перекладывает работу по определению закрытого поддерживающего поля и связанного свойства C# на компилятор, используя небольшое усовершенствование синтаксиса. В целях иллюстрации создадим новое консольное приложение по имени `AutoProps` и поместим в него переделанный класс `Car`, в котором этот синтаксис применяется для быстрого создания трех свойств:

```
class Car
{
    // Автоматические свойства!
    public string PetName { get; set; }
    public int Speed { get; set; }
    public string Color { get; set; }
}
```

На заметку! В Visual Studio предоставляется фрагмент кода `prop`. Когда вы набираете слово `prop` и два раза нажимаете клавишу `<Tab>`, IDE-среда сгенерирует начальный код для нового автоматического свойства! Затем можно с помощью клавиши `<Tab>` циклически пройти по всем частям определения и заполнить необходимые детали. Попробуйте это.

При определении автоматических свойств указывается модификатор доступа, лежащий в основе тип данных, имя свойства и пустые контексты `get/set`. Во время компиляции тип будет оснащен автоматически сгенерированным полем и соответствующей реализацией логики `get/set`.

На заметку! Имя автоматически сгенерированного закрытого поддерживающего поля в кодовой базе C# не является видимым. Единственный способ взглянуть на него — воспользоваться таким инструментом, как `ildasm.exe`.

Однако в отличие от традиционных свойств C#, создавать автоматические свойства, предназначенные только для чтения или только для записи, нельзя. Хотя может показаться, что для этого достаточно опустить `get` или `set` в объявлении свойства, как показано ниже:

```
// Свойство только для чтения? Ошибка!
public int MyReadOnlyProp { get; }

// Свойство только для записи? Ошибка!
public int MyWriteOnlyProp { set; }
```

на самом деле это приведет к ошибке компиляции. Определяемое автоматическое свойство должно поддерживать функциональность и чтения, и записи. Вспомните следующий код:

```
// Автоматические свойства должны быть доступны для чтения и для записи.
public string PetName { get; set; }
```

Взаимодействие с автоматическими свойствами

Поскольку компилятор будет определять закрытые поддерживающие поля во время компиляции, класс с автоматическими свойствами всегда должен использовать синтаксис свойств для установки и чтения лежащих в основе значений. Это важно отметить, потому что многие программисты напрямую применяют закрытые поля *внутри* определения класса, что в данном случае невозможно. Например, если бы класс `Car` включал метод `DisplayStats()`, он должен был бы реализовать этот метод, используя имя свойства:

```
class Car
{
    // Автоматические свойства!
    public string PetName { get; set; }
    public int Speed { get; set; }
    public string Color { get; set; }

    public void DisplayStats()
    {
        Console.WriteLine("Car Name: {0}", PetName);
        Console.WriteLine("Speed: {0}", Speed);
        Console.WriteLine("Color: {0}", Color);
    }
}
```

При использовании объекта, определенного с автоматическими свойствами, можно присваивать и получать значения, используя ожидаемый синтаксис свойств:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Automatic Properties *****\n");

    Car c = new Car();
    c.PetName = "Frank";
    c.Speed = 55;
    c.Color = "Red";

    Console.WriteLine("Your car is named {0}? That's odd...",
        c.PetName);

    c.DisplayStats();
    Console.ReadLine();
}
```

Замечания относительно автоматических свойств и стандартных значений

При использовании автоматических свойств для инкапсуляции числовых и булевских данных можно сразу применять автоматически сгенерированные свойства типа прямо в своей кодовой базе, поскольку их скрытым поддерживающим полям будут присвоены безопасные стандартные значения, которые могут быть использованы непосредственно. Однако будьте осторожны, если синтаксис автоматического свойства применяется для упаковки переменной другого класса, потому что скрытое поле ссылочного типа также будет установлено в стандартное значение, т.е. `null`.

Рассмотрим следующий новый класс по имени `Garage` (гараж), в котором используются два автоматических свойства (разумеется, реальный класс гаража должен поддерживать коллекцию объектов `Car`; однако пока что проигнорируем эту деталь):

```
class Garage
{
    // Скрытое поддерживающее поле int установлено в 0!
    public int NumberOfCars { get; set; }
    // Скрытое поддерживающее поле Car установлено в null!
    public Car MyAuto { get; set; }
}
```

Имея установленные стандартные значения C# для полей данных, значение `NumberOfCars` можно вывести в том виде, как оно есть (поскольку ему автоматически присвоено значение 0). Однако если напрямую обратиться к `MyAuto`, то во время выполнения сгенерируется исключение ссылки на `null`, потому что лежащей в основе переменной-члену типа `Car` не был присвоен новый объект:

```
static void Main(string[] args)
{
    ...
    Garage g = new Garage();

    // Нормально, выводится стандартное значение, равное 0.
    Console.WriteLine("Number of Cars: {0}", g.NumberOfCars);
    // Ошибка времени выполнения! Поддерживающее поле в данный момент равно null!
    Console.WriteLine(g.MyAuto.PetName);
    Console.ReadLine();
}
```

Учитывая, что закрытые поддерживающие поля создаются во время компиляции, в синтаксисе инициализации полей C# нельзя непосредственно размещать экземпляр ссылочного типа посредством `new`. Это должно делаться конструкторами класса, что обеспечит создание объекта безопасным образом. Например:

```
class Garage
{
    // Скрытое поддерживающее поле установлено в 0!
    public int NumberOfCars { get; set; }

    // Скрытое поддерживающее поле установлено в null!
    public Car MyAuto { get; set; }

    // Для переопределения стандартных значений, присвоенных скрытым
    // поддерживающим полям, должны использоваться конструкторы.
    public Garage()
    {
        MyAuto = new Car();
        NumberOfCars = 1;
    }
}
```



```
public Garage(Car car, int number)
{
    MyAuto = car;
    NumberOfCars = number;
}
}
```

После этой модификации объект Car можно поместить в объект Garage, как показано ниже:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Automatic Properties *****\n");

    // Создать автомобиль.
    Car c = new Car();
    c.PetName = "Frank";
    c.Speed = 55;
    c.Color = "Red";
    c.DisplayStats();

    // Поместить автомобиль в гараж.
    Garage g = new Garage();
    g.MyAuto = c;

    // Вывод количества автомобилей в гараже.
    Console.WriteLine("Number of Cars in garage: {0}", g.NumberOfCars);

    // Вывод названия автомобиля.
    Console.WriteLine("Your car is named: {0}", g.MyAuto.PetName);

    Console.ReadLine();
}
```

Нельзя не согласиться с тем, что это очень полезное средство языка программирования C#, поскольку свойства для класса можно определять с использованием простого синтаксиса. Естественно, если свойство помимо получения и установки закрытого поддерживающего поля требует дополнительного кода (такого как логика проверки достоверности, запись в журнал событий, взаимодействие с базой данных), придется определить его как “нормальное” свойство .NET вручную. Автоматические свойства C# никогда не делают ничего кроме обеспечения простой инкапсуляции для лежащих в основе закрытых данных (сгенерированных компилятором).

Исходный код. Проект AutoProps доступен в подкаталоге Chapter 05.

Понятие синтаксиса инициализации объектов

Как можно было видеть на протяжении этой главы, при создании нового объекта конструктор позволяет указывать начальные значения. Также было показано, что свойства позволяют получать и устанавливать лежащие в основе данные в безопасной манере. При работе с классами, которые написаны другими, включая классы из библиотеки базовых классов .NET, нередко можно заметить, что в них есть более одного конструктора, позволяющего устанавливать каждую порцию данных внутреннего состояния. Учитывая это, программист обычно старается выбрать наиболее подходящий конструктор, после чего присваивает недостающие значения, используя доступные в классе свойства.

Чтобы облегчить процесс создания и запуска объекта, в С# предлагается *синтаксис инициализатора объекта*. С помощью этого механизма можно создать новую объектную переменную и присвоить значения множеству свойств и/или открытых полей в нескольких строках кода. Синтаксически инициализатор объекта выглядит как список значений, разделенных запятыми, помещенный в фигурные скобки ({ }). Каждый элемент в списке инициализации отображается на имя открытого поля или свойства инициализируемого объекта.

Рассмотрим пример применения этого синтаксиса. Создадим новое консольное приложение по имени `ObjectInitializers`. Ниже показан класс `Point`, в котором используются автоматические свойства (что вообще-то не обязательно в данном примере, но помогает сократить код):

```
class Point
{
    public int X { get; set; }
    public int Y { get; set; }

    public Point(int xVal, int yVal)
    {
        X = xVal;
        Y = yVal;
    }
    public Point() { }

    public void DisplayStats()
    {
        Console.WriteLine("[{0}, {1}]", X, Y);
    }
}
```

А теперь посмотрим, как создавать объекты `Point`:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Object Init Syntax *****\n");

    // Создать объект Point с установкой каждого свойства вручную.
    Point firstPoint = new Point();
    firstPoint.X = 10;
    firstPoint.Y = 10;
    firstPoint.DisplayStats();

    // Создать объект Point с использованием специального конструктора.
    Point anotherPoint = new Point(20, 20);
    anotherPoint.DisplayStats();

    // Создать объект Point с использованием синтаксиса инициализатора объекта.
    Point finalPoint = new Point { X = 30, Y = 30 };
    finalPoint.DisplayStats();
    Console.ReadLine();
}
```

При создании последней переменной `Point` не используется специальный конструктор (как это принято делать традиционно), а вместо этого устанавливаются значения открытых свойств `X` и `Y`. “За кулисами” вызывается стандартный конструктор типа, за которым следует установка значений указанных свойств. В конечном счете, синтаксис инициализации объектов — это просто сокращенная нотация синтаксиса создания переменной класса с помощью стандартного конструктора, с последующей установкой свойств данных состояния.

Вызов специальных конструкторов с помощью синтаксиса инициализации

В предыдущих примерах типы `Point` инициализировались неявным вызовом стандартного конструктора этого типа:

```
// Здесь стандартный конструктор вызывается неявно.
Point finalPoint = new Point { X = 30, Y = 30 };
```

При желании стандартный конструктор можно вызывать и явно:

```
// Здесь стандартный конструктор вызывается явно
Point finalPoint = new Point() { X = 30, Y = 30 };
```

Имейте в виду, что при конструировании типа с использованием нового синтаксиса инициализации можно вызывать *любой* конструктор, определенный в классе. В настоящий момент в типе `Point` определен конструктор с двумя аргументами для установки позиции (*x*, *y*). Таким образом, следующее объявление `Point` в результате приведет к установке *X* в 100 и *Y* в 100, независимо от того факта, что в аргументах конструктора указаны значения 10 и 16:

```
// Вызов специального конструктора.
Point pt = new Point(10, 16) { X = 100, Y = 100 };
```

Имея текущее определение типа `Point`, вызов специального конструктора с применением синтаксиса инициализации не особенно полезен (и чересчур многословен). Однако если тип `Point` предоставляет новый конструктор, позволяющий вызывающему коду установить цвет (через специальное перечисление `PointColor`), комбинация специальных конструкторов и синтаксиса инициализации объекта становится ясной. Изменим `Point` следующим образом:

```
public enum PointColor
{ LightBlue, BloodRed, Gold }

class Point
{
    public int X { get; set; }
    public int Y { get; set; }
    public PointColor Color { get; set; }
    public Point(int xVal, int yVal)
    {
        X = xVal;
        Y = yVal;
        Color = PointColor.Gold;
    }
    public Point(PointColor ptColor)
    {
        Color = ptColor;
    }
    public Point()
        : this(PointColor.BloodRed) { }
    public void DisplayStats()
    {
        Console.WriteLine("[{0}, {1}]", X, Y);
        Console.WriteLine("Point is {0}", Color);
    }
}
```

С помощью этого нового конструктора можно создать золотую точку (в позиции (90, 20)), как показано ниже:

```
// Вызов более интересного специального конструктора с синтаксисом инициализации.
Point goldPoint = new Point(PointColor.Gold){ X = 90, Y = 20 };
goldPoint.DisplayStats();
```

Инициализация вложенных типов

Как было ранее кратко упомянуто в этой главе (и будет подробно рассматриваться в главе 6), отношение “имеет” (“has-a”) позволяет составлять новые классы, определяя переменные-члены существующих классов. Например, предположим, что существует класс `Rectangle`, который использует тип `Point` для представления координат верхнего левого и нижнего правого углов. Поскольку автоматические свойства устанавливают все внутренние переменные классов в `null`, новый класс будет реализован с использованием “традиционного” синтаксиса свойств.

```
class Rectangle
{
    private Point topLeft = new Point();
    private Point bottomRight = new Point();
    public Point TopLeft
    {
        get { return topLeft; }
        set { topLeft = value; }
    }
    public Point BottomRight
    {
        get { return bottomRight; }
        set { bottomRight = value; }
    }
    public void DisplayStats()
    {
        Console.WriteLine("[TopLeft: {0}, {1}, {2} BottomRight: {3}, {4}, {5}]",
            topLeft.X, topLeft.Y, topLeft.Color,
            bottomRight.X, bottomRight.Y, bottomRight.Color);
    }
}
```

С помощью синтаксиса инициализации объекта можно было бы создать новую переменную `Rectangle` и установить внутренние экземпляры `Point` следующим образом:

```
// Создать и инициализировать Rectangle.
Rectangle myRect = new Rectangle
{
    TopLeft = new Point { X = 10, Y = 10 },
    BottomRight = new Point { X = 200, Y = 200 }
};
```

Преимущество синтаксиса инициализации объектов в том, что он в основном сокращает объем кода (предполагая отсутствие подходящего конструктора). Вот как выглядит традиционный подход для установки того же экземпляра `Rectangle`:

```
// Традиционный подход.
Rectangle r = new Rectangle();
Point p1 = new Point();
p1.X = 10;
p1.Y = 10;
r.TopLeft = p1;
Point p2 = new Point();
p2.X = 200;
p2.Y = 200;
r.BottomRight = p2;
```

Хотя поначалу синтаксис инициализации объекта может показаться не слишком привычным, как только вы освоитесь с кодом, то будете удивлены, насколько быстро и с минимальными усилиями можно устанавливать состояние нового объекта.

В завершение главы рассмотрим три небольших темы, которые способствуют лучшему пониманию построения хорошо инкапсулированных классов: константные данные, поля, доступные только для чтения, и определения частичных классов.

Исходный код. Проект `ObjectInitializers` доступен в подкаталоге `Chapter 05`.

Работа с данными константных полей

В C# имеется ключевое слово `const` для определения константных данных, которые никогда не могут изменяться после начальной установки. Как и можно было предположить, это полезно при определении наборов известных значений, логически привязанных к конкретному классу или структуре, для использования в приложениях.

Предположим, что создается обслуживающий класс по имени `MyMathClass`, в котором нужно определить значение `PI` (будем считать его равным 3.14). Начнем с создания нового проекта консольного приложения по имени `ConstData`. Учитывая, что другие разработчики не должны иметь возможность изменять значение `PI` в коде, его можно смоделировать с помощью следующей константы:

```
namespace ConstData
{
    class MyMathClass
    {
        public const double PI = 3.14;
    }

    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** Fun with Const *****\n");
            Console.WriteLine("The value of PI is: {0}", MyMathClass.PI);

            // Ошибка! Нельзя изменять константу!
            MyMathClass.PI = 3.1444;

            Console.ReadLine();
        }
    }
}
```

Обратите внимание, что обращение к константным данным, определенным в классе `MyMathClass`, осуществляется с использованием префикса в виде имени класса (т.е. `MyMathClass.PI`). Это связано с тем, что константные поля класса являются неявно *статическими*. Однако допустимо определять и обращаться к локальным константным переменным внутри члена типа, например:

```
static void LocalConstStringVariable()
{
    // Локальные константные данные доступны непосредственно.
    const string fixedStr = "Fixed string Data";
    Console.WriteLine(fixedStr);

    // Ошибка!
    fixedStr = "This will not work!";
}
```

Независимо от того, где определяется константная порция данных, следует всегда помнить, что начальное значение константы всегда должно быть указано в момент ее определения. Таким образом, если модифицировать класс `MyMathClass`, чтобы значение `PI` присваивалось в конструкторе класса, то возникнет ошибка на этапе компиляции:

```
class MyMathClass
{
    // Попытка установить PI в конструкторе?
    public const double PI;

    public MyMathClass()
    {
        // Ошибка!
        PI = 3.14;
    }
}
```

Причина этого ограничения в том, что значение константных данных должно быть известно *во время компиляции*. Конструкторы же, как известно, вызываются *во время выполнения*.

Понятие полей, допускающих только чтение

Близко к понятию константных данных лежит понятие данных полей, *доступных только для чтения* (которые не следует путать со свойствами только для чтения). Подобно константам, поля только для чтения не могут быть изменены после начального присваивания. Однако, в отличие от констант, значение, присваиваемое такому полю, может быть определено во время выполнения, и потому может быть на законном основании присвоено в контексте конструктора, но нигде более.

Это может быть очень полезно в ситуациях, когда значение поля неизвестно вплоть до момента выполнения (возможно, потому, что для получения значения необходимо прочесть внешний файл), но нужно гарантировать, что оно не будет изменяться после первоначального присваивания. В целях иллюстрации рассмотрим следующее изменение в классе `MyMathClass`:

```
class MyMathClass
{
    // Поля только для чтения могут присваиваться
    // в конструкторах, но нигде более.
    public readonly double PI;
    public MyMathClass ()
    {
        PI = 3.14;
    }
}
```

Любая попытка выполнить присваивание полю, помеченному как `readonly`, вне контекста конструктора приведет к ошибке компиляции:

```
class MyMathClass
{
    public readonly double PI;
    public MyMathClass ()
    {
        PI = 3.14;
    }

    // Ошибка!
    public void ChangePI()
    { PI = 3.14444; }
}
```

Статические поля, допускающие только чтение

В отличие от константных полей, поля, допускающие только чтение, не являются явно статическими. Поэтому если необходимо представить PI на уровне класса, то для этого понадобится явно применить ключевое слово `static`. Если значение статического поля только для чтения известно во время компиляции, то начальное присваивание выглядит очень похожим на константу (однако в этом случае проще воспользоваться ключевым словом `const` в первом месте, поскольку поле данных присваивается во время его объявления):

```
class MyMathClass
{
    public static readonly double PI = 3.14;
}

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with Const *****");
        Console.WriteLine("The value of PI is: {0}", MyMathClass.PI);
        Console.ReadLine();
    }
}
```

Однако если значение статического поля только для чтения не известно до момента выполнения, можно прибегнуть к использованию статического конструктора, как было описано ранее в этой главе:

```
class MyMathClass
{
    public static readonly double PI;

    static MyMathClass()
    { PI = 3.14; }
}
```

Исходный код. Проект `ConstData` доступен в подкаталоге `Chapter 05`.

Понятие частичных типов

В этой главе осталось еще разобраться с ролью ключевого слова `partial`. Класс производственного уровня легко может содержать многие сотни строк кода. К тому же, учитывая, что типичный класс определен внутри одного файла `*.cs`, может получиться очень длинный файл. В процессе создания классов нередко большая часть кода, будучи однажды написанной, может в основном игнорироваться. Например, поля данных, свойства и конструкторы, как правило, остаются неизменными во время эксплуатации, в то время как методы имеют тенденцию довольно часто модифицироваться.

При желании можно разнести единственный класс по нескольким файлам C#, чтобы изолировать рутинный код от более ценных полезных членов. Для примера загрузим ранее созданный проект `EmployeeApp` в Visual Studio и откроем файл `Employee.cs` для редактирования. Сейчас этот единственный файл содержит код для всех аспектов класса:

```
class Employee
{
    // Поля данных
    // Конструкторы
    // Методы
    // Свойства
}
```

Механизм частичных классов позволяет вынести конструкторы и поля данных в совершенно новый файл по имени `Employee.Internal.cs` (обратите внимание, что имя файла не имеет значения; здесь оно выбрано в соответствии с назначением класса). Первый шаг состоит в добавлении ключевого слова `partial` к текущему определению класса и вырезании кода, который должен быть помещен в новый файл:

```
// Employee.cs
partial class Employee
{
    // Методы
    // Свойства
}
```

Предполагая, что к проекту добавлен новый класс, можно переместить поля данных и конструкторы в новый файл посредством простой операции вырезания и вставки. Кроме того, необходимо добавить ключевое слово `partial` к этому аспекту определения класса.

```
// Employee.Internal.cs
partial class Employee
{
    // Поля данных
    // Конструкторы
}
```

На заметку! Помните, что каждый аспект определения частичного класса должен быть помечен ключевым словом `partial`!

После компиляции модифицированного проекта вы не должны заметить никакой разницы. Вся идея, положенная в основу частичного класса, касается только этапа проектирования. Как только приложение скомпилировано, в сборке оказывается один цельный класс. Единственное требование при определении частичных типов связано с тем, что разные части должны иметь одно и то же имя и находиться в пределах одного и того же пространства имен `.NET`.

Откровенно говоря, определения частичных классов применяются нечасто. Тем не менее, среда `Visual Studio` постоянно использует их в фоновом режиме. Позже в этой книге, когда речь пойдет о разработке приложений с графическим пользовательским интерфейсом посредством `Windows Presentation Foundation` или `ASP.NET`, будет показано, что `Visual Studio` изолирует сгенерированный визуальным конструктором код в частичном классе, позволяя сосредоточиться на специфичной для приложения программной логике.

Исходный код. Проект `EmployeeAppPartial` доступен в подкаталоге `Chapter 05`.

Резюме

Цель этой главы заключалась в ознакомлении с ролью типов классов C#. Вы видели, что классы могут иметь любое количество *конструкторов*, которые позволяют пользователю объекта устанавливать состояние объекта при его создании. В главе также было проиллюстрировано несколько приемов проектирования классов (и связанных с ними ключевых слов). Ключевое слово `this` используется для получения доступа к текущему объекту, ключевое слово `static` позволяет определять поля и члены, привязанные к классу (а не объекту), а ключевое слово `const` (и модификатор `readonly`) дает возможность определять элементы данных, которые никогда не изменяются после первоначальной установки.

Большая часть главы была посвящена деталям первого принципа ООП: инкапсуляции. Здесь вы узнали о модификаторах доступа C# и роли свойств типа, о синтаксисе инициализации объектов и о частичных классах. Обладая всеми этими знаниями, теперь вы готовы к тому, чтобы перейти к следующей главе, в которой рассказывается о построении семейства взаимосвязанных классов с использованием наследования и полиморфизма.

Понятие наследования и полиморфизма

В главе 5 рассматривался первый основной принцип объектно-ориентированного программирования (ООП) — инкапсуляция. Вы узнали, как построить отдельный правильно спроектированный тип класса с конструкторами и различными членами (конструкторами, полями, свойствами, методами, константами и полями, доступными только для чтения). В настоящей главе мы сосредоточимся на остальных двух принципах ООП: наследовании и полиморфизме.

Прежде всего, вы научитесь строить семейства связанных классов с применением наследования. Как будет показано, эта форма повторного использования кода позволяет определять в родительском классе общую функциональность, которая может применяться и, возможно, изменяться в дочерних классах. По ходу изложения вы узнаете, как устанавливать полиморфный интерфейс в иерархиях классов, используя виртуальные и абстрактные члены, а также о роли явного приведения. Завершается глава рассмотрением роли изначального родительского класса в библиотеках базовых классов .NET, которым является `System.Object`.

Базовый механизм наследования

Вспомните из главы 5, что *наследование* — это аспект ООП, облегчающий повторное использование кода. Строго говоря, повторное использование кода существует в двух видах: наследование (отношение “является”) и модель включения/делегации (отношение “имеет”). Начнем главу с рассмотрения классической модели наследования — отношения “является”.

При установке между классами отношения “является” строится зависимость между двумя или более типами классов. Базовая идея, лежащая в основе классического наследования, заключается в том, что новые классы могут создаваться с применением существующих классов в качестве отправной точки. Давайте начнем с очень простого примера, создав новый проект консольного приложения по имени `BasicInheritance`. Предположим, что спроектирован класс по имени `Car`, моделирующий некоторые базовые детали автомобиля:

```
// Простой базовый класс.  
class Car  
{  
    public readonly int maxSpeed;  
    private int currSpeed;
```

```

public Car(int max)
{
    maxSpeed = max;
}

public Car()
{
    maxSpeed = 55;
}

public int Speed
{
    get { return currSpeed; }
    set
    {
        currSpeed = value;
        if (currSpeed > maxSpeed)
        {
            currSpeed = maxSpeed;
        }
    }
}
}

```

Обратите внимание на использование в Car инкапсуляции для управления доступом к закрытому полю `currSpeed` с помощью открытого свойства по имени `Speed`. Имея такое определение, с типом `Car` можно работать следующим образом:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Basic Inheritance *****\n");
    // Создать экземпляр типа Car и установить максимальную скорость.
    Car myCar = new Car(80);

    // Установить текущую скорость и вывести ее на консоль.
    myCar.Speed = 50;
    Console.WriteLine("My car is going {0} MPH", myCar.Speed);
    Console.ReadLine();
}

```

Указание родительского класса для существующего класса

Теперь предположим, что планируется построить новый класс по имени `MiniVan`. Подобно базовому классу `Car`, необходимо, чтобы `MiniVan` поддерживал максимальную скорость, текущую скорость и свойство по имени `Speed`, позволяющее пользователю модифицировать состояние объекта. Ясно, что классы `Car` и `MiniVan` взаимосвязаны; фактически можно сказать, что `MiniVan` “является” `Car`. Отношение “является” (формально называемое *классическим наследованием*) позволяет строить новые определения классов, расширяющие функциональность существующих классов.

Существующий класс, который будет служить основой для нового класса, называется *базовым* или *родительским* классом. Назначение базового класса состоит в определении всех общих данных и членов для классов, которые расширяют его. Расширяющие классы формально называются *производными* или *дочерними* классами. В C# для установки между классами отношения “является” используется операция двоеточия в определении класса. Предположим, что написан следующий новый класс `MiniVan`:

```

// MiniVan "является" Car.
class MiniVan : Car
{
}

```

В настоящее время этот новый класс не определяет никаких своих членов. Так в чем же тогда заключается выигрыш от наследования `MiniVan` от базового класса `Car`? Выражаясь просто, объекты `MiniVan` теперь имеют доступ ко всем открытым членам, определенным в базовом классе.

На заметку! Хотя конструкторы обычно определяются как открытые, производный класс никогда не наследует конструкторы своего родительского класса. Конструкторы применяются только для создания экземпляра класса, внутри которого они определены.

Учитывая отношение между этими двумя типами классов, класс `MiniVan` можно использовать следующим образом:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Basic Inheritance *****\n");
    ...
    // Создать объект MiniVan.
    MiniVan myVan = new MiniVan();
    myVan.Speed = 10;
    Console.WriteLine("My van is going {0} MPH",
        myVan.Speed);
    Console.ReadLine();
}
```

Обратите внимание, что хотя к классу `MiniVan` не добавлены никакие члены, имеет-ся прямой доступ к открытому свойству `Speed` родительского класса и, таким образом, его код используется повторно. Это намного лучше, чем создавать класс `MiniVan`, име-ющий в точности такие же члены, что и `Car`, вроде свойства `Speed`. В случае дублирова-ния кода в этих двух классах придется сопровождать два фрагмента одинакового кода, что очевидно является непроизводительным расходом времени.

Всегда помните, что наследование предохраняет инкапсуляцию, а потому следую-щий код вызовет ошибку компиляции, поскольку закрытые члены никогда не могут быть доступны через ссылку на объект:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Basic Inheritance *****\n");
    ...
    // Создать объект MiniVan.
    MiniVan myVan = new MiniVan();
    myVan.Speed = 10;
    Console.WriteLine("My van is going {0} MPH",
        myVan.Speed);
    // Ошибка! Доступ к закрытым членам невозможен!
    myVan.currSpeed = 55;
    Console.ReadLine();
}
```

Кстати говоря, если в `MiniVan` будет определен собственный набор членов, он по-прежнему не будет иметь возможность доступа к закрытым членам базового класса `Car`. Вспомните, что закрытые члены могут быть доступны *только* в классе, в котором они определены. Например, следующий метод в `MiniVan` вызовет ошибку компиляции:

```
// MiniVan унаследован от Car.
class MiniVan : Car
{
    public void TestMethod()
    {
```

```
// Нормально! Доступ к открытым членам родительского
// типа в производном типе возможен.
Speed = 10;

// Ошибка! Нельзя осуществлять доступ к закрытым
// членам родительского типа из производного типа!
currSpeed = 10;
}
}
```

Замечание относительно множества базовых классов

Говоря о базовых классах, важно иметь в виду, что язык C# требует, чтобы любой конкретный класс имел в точности один непосредственный базовый класс. Невозможно создать тип класса, который был бы напрямую унаследованным от двух и более базовых классов (эта техника, поддерживаемая в неуправляемом языке C++, называется *множественным наследованием*). Попытка создать класс, в котором заданы два непосредственных родительских класса, как показано в следующем коде, приводит к ошибке компиляции:

```
// Не разрешено! Язык C# не допускает
// множественного наследования классов!
class WontWork
: BaseClassOne, BaseClassTwo
{ }
```

Как будет объясняться в главе 8, платформа .NET позволяет конкретному классу или структуре реализовывать любое количество дискретных интерфейсов. Благодаря этому, тип C# может представлять набор поведений, избегая сложностей, присущих множественному наследованию. К слову, в то время как класс может иметь только один непосредственный базовый класс, интерфейс разрешено наследовать от множества других интерфейсов. Используя эту технику, можно строить изощренные иерархии интерфейсов, моделирующих сложные поведения (см. главу 8).

Ключевое слово `sealed`

В C# поддерживается еще одно ключевое слово — `sealed`, которое предотвращает наследование. Если класс помечен как `sealed` (запечатанный), компилятор не позволяет наследовать от него. Например, предположим, решено, что нет смысла в дальнейшем наследовании от класса `MiniVan`:

```
// Класс Minivan не может быть расширен!
sealed class MiniVan : Car
{
}
```

Если вы (или коллега по команде) попытаетесь унаследовать от этого класса, то получите ошибку на этапе компиляции:

```
// Ошибка! Нельзя расширять класс, помеченный ключевым словом sealed!
class DeluxeMiniVan
: MiniVan
{ }
```

Чаще всего запечатывание имеет смысл при проектировании обслуживающего класса. Например, в пространстве имен `System` определено множество запечатанных классов. В этом легко убедиться, открыв браузер объектов в Visual Studio (через меню View (Вид)) и выбрав класс `String`, определенный в пространстве имен `System` внутри сборки `mscorlib.dll`. На рис. 6.1 обратите внимание на использование ключевого слова `sealed`, выделенного в окне Summary (Сводка).

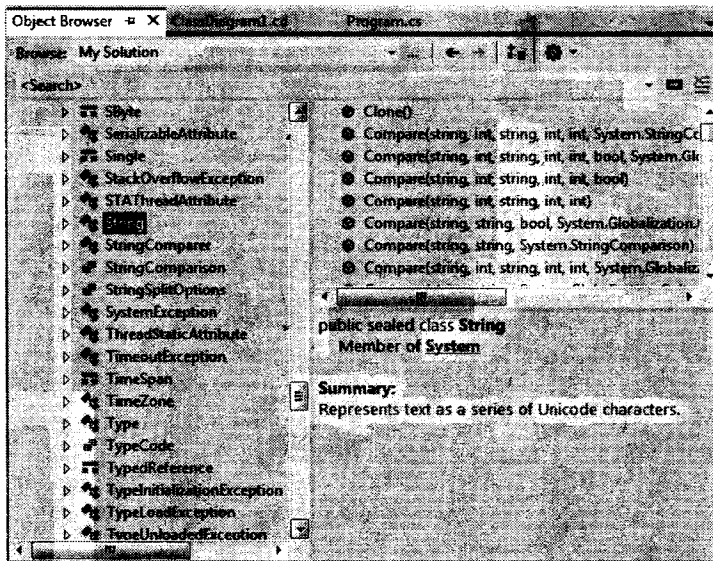


Рис. 6.1. В библиотеках базовых классов определено множество запечатанных типов, таких как `System.String`

Таким образом, как и с `MiniVan`, если попытаться построить новый класс, расширяющий `System.String`, возникнет ошибка компиляции:

```
// Ошибка! Нельзя расширять класс, помеченный как sealed!
class MyString
    : String
{ }
```

На заметку! В главе 4 было показано, что структуры C# всегда неявно запечатаны (см. табл. 4.3). Поэтому ни унаследовать одну структуру от другой, ни класс от структуры, ни структуру от класса не получится. Структуры могут использоваться только для моделирования отдельных атомарных, определенных пользователем типов. Для реализации отношения “является” должны применяться классы.

Как и можно было ожидать, существует множество других деталей наследования, о которых вы узнаете в оставшейся части главы. Пока просто имейте в виду, что операция двоеточия позволяет устанавливать между классами отношения “базовый–производный”, а ключевое слово `sealed` предотвращает наследование.

Изменение диаграмм классов Visual Studio

В главе 2 кратко упоминалось, что среда Visual Studio позволяет устанавливать между классами отношения “базовый–производный” визуально во время проектирования. Для использования этого аспекта IDE-среды первый шаг состоит во включении нового файла диаграммы классов в текущий проект. Для этого выберите в меню пункт `Project→Add New Item` (Проект→Добавить новый элемент) и затем значок `Class Diagram` (Диаграмма классов); на рис. 6.2 имя файла `ClassDiagram1.cd` было изменено на `Cars.cd`.

После щелчка на кнопке `Add` (Добавить) появится пустая поверхность проектирования. Для добавления классов к диаграмме просто перетаскивайте каждый файл из окна `Solution Explorer` (Проводник решения) на эту поверхность.

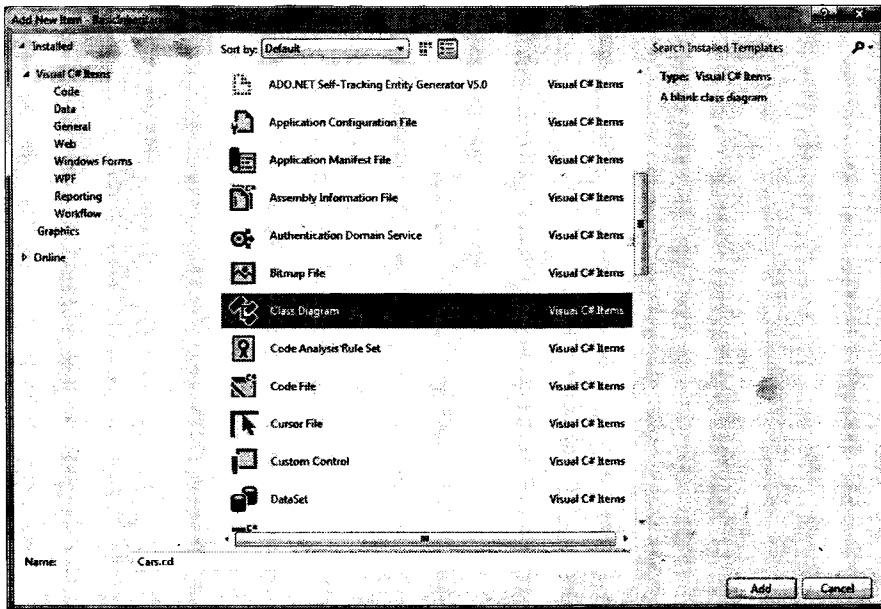


Рис. 6.2. Вставка в проект новой диаграммы классов

Также помните, что удаление элемента в визуальном конструкторе (за счет его выбора и нажатия клавиши <Delete>) не приводит к удалению ассоциированного исходного кода, а просто убирает элемент из поверхности проектирования. Текущая иерархия классов показана на рис. 6.3.

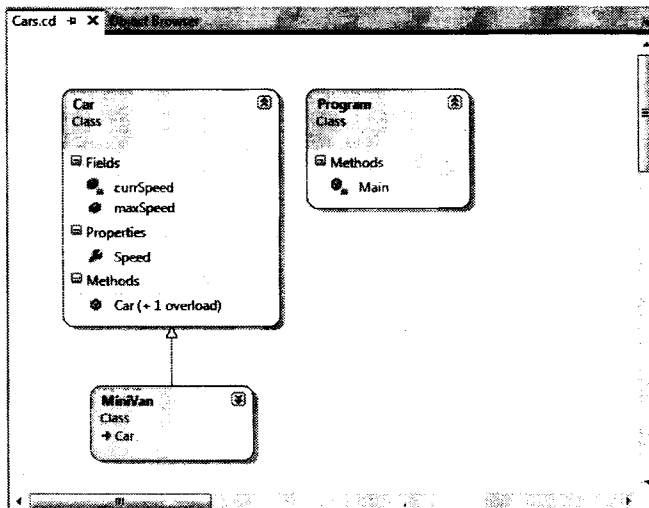


Рис. 6.3. Визуальный конструктор Visual Studio

На заметку! Итак, если необходимо автоматически добавить все текущие типы проекта на поверхность проектирования, выберите узел Project (Проект) в Solution Explorer и щелкните на кнопке View Class Diagram (Просмотреть диаграмму классов) в правом верхнем углу окна Solution Explorer.

Помимо простого отображения отношений между типами внутри текущего приложения, вспомните из главы 2, что можно также создавать совершенно новые типы и наполнять их членами, используя панель инструментов Class Designer (Конструктор классов) и окно Class Details (Детали класса).

Если хотите использовать эти визуальные инструменты в процессе дальнейшего чтения книги — пожалуйста. Однако всегда анализируйте сгенерированный код, чтобы четко понимать, что эти инструменты делают за вашей спиной.

Исходный код. Проект BasicInheritance доступен в подкаталоге Chapter 06.

Второй принцип ООП: подробности о наследовании

Ознакомившись с базовым синтаксисом наследования, давайте рассмотрим более сложный пример и узнаем о многочисленных деталях построения иерархий классов. Для этого воспользуемся классом Employee, спроектированным в главе 5. Для начала создадим новое консольное приложение C# по имени Employees.

Выберите пункт меню Project⇒Add Existing Item (Проект⇒Добавить существующий элемент) и перейдите к месту нахождения файлов Employee.cs и Employee.Internals.cs, которые были созданы в примере EmployeeApp из главы 5. Выберите оба файла (щелкая на них при нажатой клавише <Ctrl>) и щелкните на кнопке OK. Среда Visual Studio отреагирует копированием каждого файла в текущий проект.

Прежде чем начать построение производных классов, следует уделить внимание одной детали. Поскольку первоначальный класс Employee был создан в проекте по имени EmployeeApp, этот класс находится в идентично названном пространстве имен .NET. Пространства имен подробно рассматриваются в главе 14, а пока для простоты просто переименуйте текущее пространство имен (в обоих файлах) на Employees, чтобы оно соответствовало имени нового проекта:

```
// Не забудьте изменить название пространства имен в обоих файлах!
namespace Employees
{
    partial class Employee
    { ... }
}
```

На заметку! Чтобы подстраховаться, скомпилируйте и запустите новый проект, нажав <Ctrl+F5>.

Пока программа ничего не делает, однако это позволит удостовериться в отсутствии ошибок компиляции.

Наша цель заключается в создании семейства классов, моделирующих различные типы сотрудников компании. Предположим, что необходимо воспользоваться функциональностью класса Employee при создании двух новых классов (SalesPerson и Manager). Новый класс SalesPerson “является” Employee (как и Manager). Вспомните, что в модели классического наследования базовые классы (вроде Employee) используются для определения характеристик, общих для всех наследников. Подклассы (такие как SalesPerson и Manager) расширяют общую функциональность, добавляя дополнительную специфическую функциональность.

Для нашего примера предположим, что класс Manager расширяет Employee, храня количество опционов на акции, в то время как класс SalesPerson поддерживает количество продаж. Добавьте новый файл класса (Manager.cc), определяющий тип Manager следующим образом:


```
// Менеджерам нужно знать количество их опционов на акции.
class Manager : Employee
{
    public int StockOptions { get; set; }
}
```

Затем добавьте новый файл класса (SalesPerson.cs), в котором определен класс SalesPerson с соответствующим автоматическим свойством:

```
// Продавцам нужно знать количество продаж.
class SalesPerson : Employee
{
    public int SalesNumber { get; set; }
}
```

Теперь, после установки отношения “является”, SalesPerson и Manager автоматически наследуют все открытые члены базового класса Employee. В целях иллюстрации обновите метод Main() следующим образом:

```
// Создание объекта подкласса и доступ к функциональности базового класса.
static void Main(string[] args)
{
    Console.WriteLine("***** The Employee Class Hierarchy *****\n");
    SalesPerson fred = new SalesPerson();
    fred.Age = 31;
    fred.Name = "Fred";
    fred.SalesNumber = 50;
    Console.ReadLine();
}
```

Управление созданием базового класса с помощью ключевого слова base

Сейчас объекты SalesPerson и Manager могут быть созданы только с использованием “бесплатного” стандартного конструктора (см. главу 5). Памятуя об этом, предположим, что к типу Manager добавлен новый конструктор, который принимает шесть аргументов и вызывается следующим образом:

```
static void Main(string[] args)
{
    ...
    // Предположим, что у Manager есть конструктор со следующей сигнатурой:
    // (string fullName, int age, int empID,
    // float currPay, string ssn, int numbofOpts)
    Manager chucky = new Manager("Chucky", 50, 92, 100000, "333-23-2322", 9000);
    Console.ReadLine();
}
```

Если взглянуть на список параметров, то ясно видно, что большинство из них должно быть сохранено в переменных-членах, определенных в базовом классе Employee. В этом случае для класса Manager можно реализовать специальный конструктор следующего вида:

```
public Manager(string fullName, int age, int empID,
               float currPay, string ssn, int numbofOpts)
{
    // Это свойство определено в классе Manager.
    StockOptions = numbofOpts;

    // Присвоить входные параметры, используя
    // унаследованные свойства родительского класса.
```

```

ID = empID;
Age = age;
Name = fullName;
Pay = currPay;
// Здесь возникнет ошибка компиляции, поскольку
// свойство SSN доступно только для чтения!
SocialSecurityNumber = ssn;
}

```

Первая проблема такого подхода состоит в том, что если определить какое-то свойство как доступное только для чтения (например, `SocialSecurityNumber`), то присвоить значение входного параметра `string` соответствующему полю не удастся, как можно видеть в финальном операторе специального конструктора.

Вторая проблема состоит в том, что был неявно создан довольно неэффективный конструктор, учитывая тот факт, что в C#, если не указать иного, стандартный конструктор базового класса вызывается автоматически перед выполнением логики производного конструктора. После этого момента текущая реализация имеет доступ к многочисленным открытым свойствам базового класса `Employee` для установки его состояния. Таким образом, в действительности во время создания объекта `Manager` выполняется семь действий (обращения к пяти унаследованным свойствам и двум конструкторам).

Для оптимизации создания производного класса необходимо хорошо реализовать конструкторы подкласса, чтобы они явно вызывали специальный конструктор базового класса вместо стандартного конструктора. Поступая подобным образом, можно сократить количество вызовов инициализаций унаследованных членов (что уменьшит время обработки). Давайте модифицируем специальный конструктор класса `Manager`, применив ключевое слово `base`:

```

public Manager(string fullName, int age, int empID,
               float currPay, string ssn, int numbOfOpts)
    : base(fullName, age, empID, currPay, ssn)
{
    // Это свойство определено в классе Manager.
    StockOptions = numbOfOpts;
}

```

Здесь ключевое слово `base` ссылается на сигнатуру конструктора (подобно синтаксису, используемому для сцепления конструкторов в единственном классе с использованием ключевого слова `this`, как было показано в главе 5), а это всегда указывает на то, что производный конструктор передает данные конструктору непосредственного родителя. В данной ситуации явно вызывается конструктор с пятью параметрами, определенный в `Employee`, что избавляет от излишних вызовов во время создания экземпляра базового класса. Специальный конструктор `SalesPerson` выглядит в основном идентично:

```

// В качестве общего правила, все подклассы должны явно вызывать
// соответствующий конструктор базового класса.
public SalesPerson(string fullName, int age, int empID,
                  float currPay, string ssn, int numbOfSales)
    : base(fullName, age, empID, currPay, ssn)
{
    // Это касается нас!
    SalesNumber = numbOfSales;
}

```

На заметку! Ключевое слово `base` можно использовать везде, где подкласс желает обратиться к открытому или защищенному члену, определенному в родительском классе. Применение этого ключевого слова не ограничивается логикой конструктора. Вы увидите примеры использования `base` в такой манере далее в главе, во время рассмотрения полиморфизма.

И, наконец, вспомните, что как только в определении класса появляется специальный конструктор, стандартный конструктор из класса молча удаляется. Следовательно, не забудьте переопределить стандартный конструктор для типов `SalesPerson` и `Manager`. Например:

```
// Вернуть классу Manager стандартный конструктор.
public SalesPerson() {}
```

Хранение секретов семейства: ключевое слово `protected`

Как уже должно быть известно, открытые элементы непосредственно доступны отовсюду, в то время как закрытые могут быть доступны только в классе, где они определены. Вспомните из главы 5, что C# следует примеру многих других современных объектно-ориентированных языков и предлагает дополнительное ключевое слово для определения доступности членов, а именно — `protected` (защищенный).

Когда базовый класс определяет защищенные данные или защищенные члены, он устанавливает набор элементов, которые могут быть доступны непосредственно любому наследнику. Например, чтобы позволить дочерним классам `SalesPerson` и `Manager` непосредственно обращаться к разделу данных, определенному в `Employee`, можете изменить исходный класс `Employee` следующим образом:

```
// Защищенные данные состояния.
partial class Employee
{
    // Теперь производные классы могут напрямую обращаться к этой информации.
    protected string empName;
    protected int empID;
    protected float currPay;
    protected int empAge;
    protected string empSSN;
    ...
}
```

Преимущество определения защищенных членов в базовом классе состоит в том, что производным типам больше не нужно обращаться к данным опосредованно, используя открытые методы и свойства. Возможным недостатком такого подхода, конечно же, является то, что когда производный тип имеет прямой доступ к внутренним данным своего родителя, возникает вероятность непреднамеренного нарушения существующих бизнес-правил, которые реализованы в открытых свойствах. При определении защищенных членов создается уровень доверия между родительским и дочерним классами, поскольку компилятор не перехватывает никаких нарушений существующих бизнес-правил.

И, наконец, имейте в виду, что с точки зрения пользователя объекта защищенные данные трактуются как закрытые (поскольку пользователь находится “за пределами” семейства). Потому следующий код некорректен:

```
static void Main(string[] args)
{
    // Ошибка! Доступ к защищенным данным через экземпляр объекта невозможен!
    Employee emp = new Employee();
    emp.empName = "Fred";
}
```

На заметку! Хотя защищенные поля данных могут нарушить инкапсуляцию, объявлять защищенные методы вполне безопасно (и полезно). При построении иерархий классов очень часто приходится определять набор методов, которые используются только производными типами и не предназначены для применения внешним миром.

Добавление запечатанного класса

Вспомните, что *запечатанный* (sealed) класс не может быть расширен другими классами. Как уже упоминалось, эта техника чаще всего применяется при проектировании обслуживающих классов. Тем не менее, при построении иерархий классов можно обнаружить, что некоторая ветвь в цепочке наследования нуждается в “отсечении”, поскольку дальнейшее ее расширение не имеет смысла. Например, предположим, что в приложение добавлен еще один класс (PTSalesPerson), который расширяет существующий тип SalesPerson. На рис. 6.4 показано текущее обновление.

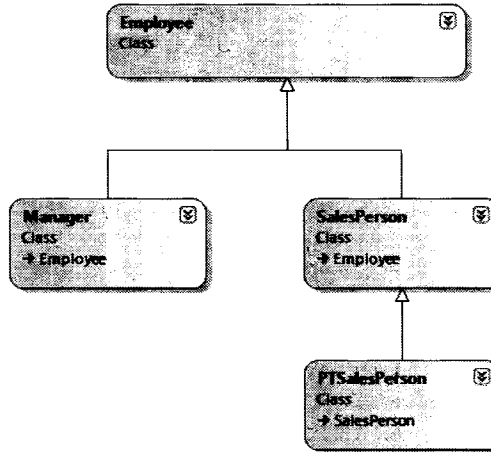


Рис. 6.4. Класс PTSalesPerson

Класс PTSalesPerson представляет продавца, который работает на условиях частичной занятости. Предположим, что необходимо гарантировать отсутствие возможности наследования от класса PTSalesPerson. (В конце концов, какой смысл в “частичной занятости от частичной занятости”?) Для предотвращения наследования от класса используется ключевое слово sealed:

```
sealed class PTSalesPerson : SalesPerson
{
    public PTSalesPerson(string fullName, int age, int empID,
                        float currPay, string ssn, int numbofSales)
        : base (fullName, age, empID, currPay, ssn, numbofSales)
    {
    }
    // Остальные члены класса...
}
```

Реализация модели включения/делегации

Вспомните, что повторное использование кода возможно в двух вариантах. Только что было рассмотрено классическое отношение “является”. Перед тем, как мы обратимся к третьему принципу ООП (полиморфизму), давайте поговорим об отношении “имеет” (еще известном под названием модели *включения/делегации* или *агрегации*). Предположим, что создан новый класс, который моделирует пакет льгот для сотрудников:

```
// Этот новый тип будет работать как включаемый класс.
class BenefitPackage
{

```

```
// Предположим, что есть другие члены, представляющие
// медицинские/стоматологические программы и т.д.
public double ComputePayDeduction()
{
    return 125.0;
}
}
```

Очевидно, что было бы довольно нелепо устанавливать отношение “является” между классом `BenefitPackage` и типами сотрудников. (Разве `Employee` “является” `BenefitPackage`? Вряд ли). Однако должно быть ясно, что какие-то отношения между ними должны быть установлены. Короче говоря, понадобится выразить идею о том, что каждый сотрудник “имеет” `BenefitPackage`. Для этого можно модифицировать определение класса `Employee` следующим образом:

```
// Сотрудники имеют льготы.
partial class Employee
{
    // Содержит объект BenefitPackage.
    protected BenefitPackage empBenefits = new BenefitPackage();
    ...
}
```

Таким образом, один объект успешно содержит в себе другой объект. Однако чтобы представить функциональность включенного объекта внешнему миру, потребуется делегация. Делегация — это просто акт добавления открытых членов к включающему классу, которые используют функциональность включенного объекта. Например, можно было бы обновить класс `Employee`, чтобы он открывал включенный объект `empBenefits` с помощью специального свойства, а также пользоваться его функциональностью внутренне, через новый метод по имени `GetBenefitCost()`:

```
public partial class Employee
{
    // Содержит объект BenefitPackage.
    protected BenefitPackage empBenefits = new BenefitPackage();

    // Открывает некоторое поведение, связанное с включенным объектом.
    public double GetBenefitCost()
    { return empBenefits.ComputePayDeduction(); }

    // Открывает объект через специальное свойство.
    public BenefitPackage Benefits
    {
        get { return empBenefits; }
        set { empBenefits = value; }
    }
    ...
}
```

В следующем обновленном методе `Main()` обратите внимание на взаимодействие с внутренним типом `BenefitsPackage`, который определен в типе `Employee`:

```
static void Main(string[] args)
{
    Console.WriteLine("***** The Employee Class Hierarchy *****\n");
    ...
    Manager chucky = new Manager("Chucky", 50, 92, 100000, "333-23-2322", 9000);
    double cost = chucky.GetBenefitCost();
    Console.ReadLine();
}
```

Определения вложенных типов

В главе 5 была кратко упомянута концепция вложенных типов, которая является разновидностью только что рассмотренного отношения “имеет”. В С# (как и в других языках .NET) допускается определять тип (перечисление, класс, интерфейс, структуру или делегат) непосредственно внутри контекста класса или структуры. При этом вложенный (или “внутренний”) тип считается членом охватывающего (или “внешнего”) класса, и в глазах исполняющей системы им можно манипулировать как любым другим членом (полем, свойством, методом и событием). Синтаксис, используемый для вложения типа, достаточно прост:

```
public class OuterClass
{
    // Открытый вложенный тип может использоваться повсюду.
    public class PublicInnerClass {}

    // Закрытый вложенный тип может использоваться только членами включающего класса.
    private class PrivateInnerClass {}
}
```

Хотя синтаксис довольно очевиден, понять, для чего это может потребоваться, не так-то просто. Чтобы разобраться с этой техникой, рассмотрим характерные особенности вложенных типов.

- Вложенные типы позволяют получить полный контроль над уровнем доступа внутреннего типа, поскольку они могут быть объявлены как закрытые (вспомните, что не вложенные классы не могут быть объявлены с использованием ключевого слова `private`).
- Поскольку вложенный тип является членом включающего класса, он может иметь доступ к закрытым членам включающего класса.
- Часто вложенные типы удобны в качестве вспомогательных для внешнего класса и не предназначены для использования внешним миром.

Когда тип включает в себя другой тип класса, он может создавать переменные-члены этого типа, как любой другой элемент данных. Однако если вложенный тип нужно применять вне включающего типа, его понадобится квалифицировать именем включающего типа. Взгляните на следующий код:

```
static void Main(string[] args)
{
    // Создать и использовать открытый вложенный класс. Правильно!
    OuterClass.PublicInnerClass inner;
    inner = new OuterClass.PublicInnerClass();

    // Ошибка компиляции! Доступ к закрытому классу невозможен!
    OuterClass.PrivateInnerClass inner2;
    inner2 = new OuterClass.PrivateInnerClass();
}
```

Чтобы использовать эту концепцию в рассматриваемом примере с сотрудниками, предположим, что теперь определение `BenefitPackage` вложено непосредственно в класс `Employee`:

```
partial class Employee
{
    public class BenefitPackage
    {
        // Предположим, что есть другие члены, представляющие
        // медицинские/стоматологические программы и т.д.
    }
}
```

```

    public double ComputePayDeduction()
    {
        return 125.0;
    }
    ...
}

```

Вложение может иметь произвольную глубину. Например, пусть требуется создать перечисление по имени `BenefitPackageLevel`, документирующее различные уровни льгот, которые могут быть предоставлены сотруднику. Чтобы программно установить тесную связь между `Employee`, `BenefitPackage` и `BenefitPackageLevel`, можно вложить перечисление следующим образом:

```

// В Employee вложен класс BenefitPackage.
public partial class Employee
{
    // В BenefitPackage вложено перечисление BenefitPackageLevel.
    public class BenefitPackage
    {
        public enum BenefitPackageLevel
        {
            Standard, Gold, Platinum
        }
        public double ComputePayDeduction()
        {
            return 125.0;
        }
    }
    ...
}

```

Из-за отношений вложения обратите внимание на то, как приходится использовать это перечисление:

```

static void Main(string[] args)
{
    ...
    // Определить уровень льгот.
    Employee.BenefitPackage.BenefitPackageLevel myBenefitLevel =
        Employee.BenefitPackage.BenefitPackageLevel.Platinum;
    Console.ReadLine()
}

```

Итак, к этому моменту вы ознакомились с множеством ключевых слов (и концепций), которые позволяют строить иерархии взаимосвязанных типов через классическое наследование, включение и вложенные типы. Если пока не все детали ясны, не переживайте. На протяжении оставшейся части книги вы построите еще много дополнительных иерархий. А теперь давайте перейдем к рассмотрению последнего принципа ООП — полиморфизма.

Третий принцип ООП: поддержка полиморфизма в C#

Вспомните, что в базовом классе `Employee` был определен метод по имени `GiveBonus()` со следующей первоначальной реализацией:

```
public partial class Employee
{
    public void GiveBonus(float amount)
    {
        Pay += amount;
    }
    ...
}
```

Поскольку этот метод был определен с ключевым словом `public`, теперь можно раздавать бонусы продавцам и менеджерам (а также продавцам с частичной занятостью):

```
static void Main(string[] args)
{
    Console.WriteLine("***** The Employee Class Hierarchy *****\n");
    // Предоставить каждому сотруднику бонус?
    Manager chucky = new Manager("Chuck", 50, 92, 100000, "333-23-2322", 9000);
    chucky.GiveBonus(300);
    chucky.DisplayStats();
    Console.WriteLine();

    SalesPerson fran = new SalesPerson("Fran", 43, 93, 3000, "932-32-3232", 31);
    fran.GiveBonus(200);
    fran.DisplayStats();
    Console.ReadLine();
}
```

Проблема текущего кода состоит в том, что открыто унаследованный метод `GiveBonus()` работает идентично для всех подклассов. В идеале при подсчете бонуса для штатного продавца и частично занятого продавца должно приниматься во внимание количество продаж. Возможно, менеджеры должны получать дополнительные опционы на акции вместе с денежным вознаграждением. Учитывая это, вы однажды столкнетесь с интересным вопросом: "Как сделать так, чтобы связанные типы по-разному реагировали на один и тот же запрос?". Попробуем отыскать на него ответ.

Ключевые слова `virtual` и `override`

Полиморфизм предоставляет подклассу способ определения собственной версии метода, определенного в его базовом классе, с использованием процесса, который называется *переопределением метода*. Чтобы пересмотреть текущее проектное решение, нужно понять значение ключевых слов `virtual` и `override`. Если базовый класс желает определить метод, который *может быть* (но не обязательно) переопределен в подклассе, он должен пометить его ключевым словом `virtual`:

```
partial class Employee
{
    // Этот метод теперь может быть переопределен производным классом.
    public virtual void GiveBonus(float amount)
    {
        Pay += amount;
    }
    ...
}
```

На заметку! Методы, помеченные ключевым словом `virtual`, называются *виртуальными методами*.

Когда класс желает изменить реализацию деталей виртуального метода, он делает это с помощью ключевого слова `override`. Например, `SalesPerson` и `Manager` могли бы переопределить `GiveBonus()`, как показано ниже (предполагая, что `PTSalesPerson` не будет переопределять `GiveBonus()`, а потому просто наследует версию, определенную `SalesPerson`):

```
class SalesPerson : Employee
{
    ...
    // Бонус продавца зависит от количества продаж.
    public override void GiveBonus(float amount)
    {
        int salesBonus = 0;
        if (SalesNumber >= 0 && SalesNumber <= 100)
            salesBonus = 10;
        else
        {
            if (SalesNumber >= 101 && SalesNumber <= 200)
                salesBonus = 15;
            else
                salesBonus = 20;
        }
        base.GiveBonus(amount * salesBonus);
    }
}

class Manager : Employee
{
    ...
    public override void GiveBonus(float amount)
    {
        base.GiveBonus(amount);
        Random r = new Random();
        StockOptions += r.Next(500);
    }
}
```

Обратите внимание на применение каждым переопределенным методом стандартного поведения через ключевое слово `base`. Таким образом, полностью повторять реализацию логики `GiveBonus()` вовсе не обязательно, а вместо этого можно повторно использовать (и, возможно, расширять) стандартное поведение родительского класса.

Также предположим, что текущий метод `DisplayStatus()` класса `Employee` объявлен виртуальным. При этом каждый подкласс может переопределять этот метод в расчете на отображение количества продаж (для продавцов) и текущих опционов на акции (для менеджеров). Например, рассмотрим версию метода `DisplayStatus()` в классе `Manager` (класс `SalesPerson` должен реализовать `DisplayStatus()` аналогичным образом, чтобы вывести на консоль количество продаж):

```
public override void DisplayStats()
{
    base.DisplayStats();
    Console.WriteLine("Number of Stock Options: {0}", numberOfOptions);
}
```

Теперь, когда каждый подкласс может интерпретировать, что именно эти виртуальные методы означают для него, каждый экземпляр объекта ведет себя как более независимая сущность:

```

static void Main(string[] args)
{
    Console.WriteLine("***** The Employee Class Hierarchy *****\n");
    // Лучшая система бонусов!
    Manager chucky = new Manager("Chucky", 50, 92, 100000, "333-23-2322", 9000);
    chucky.GiveBonus(300);
    chucky.DisplayStats();
    Console.WriteLine();
    SalesPerson fran = new SalesPerson("Fran", 43, 93, 3000, "932-32-3232", 31);
    fran.GiveBonus(200);
    fran.DisplayStats();
    Console.ReadLine();
}

```

Ниже показан результат тестового запуска приложения в нынешнем виде:

***** The Employee Class Hierarchy *****

Name: Chucky
 ID: 92
 Age: 50
 Pay: 100300
 SSN: 333-23-2322
 Number of Stock Options: 9337

Name: Fran
 ID: 93
 Age: 43
 Pay: 5000
 SSN: 932-32-3232
 Number of Sales: 31

Переопределение виртуальных членов в IDE-среде Visual Studio

Как вы уже, возможно, заметили, при переопределении члена класса необходимо помнить типы всех параметров, а также соглашения о передаче параметров (ref, out и params). В Visual Studio доступна очень полезная возможность, которой можно пользоваться при переопределении виртуального члена. Если набрать слово `override` внутри контекста типа класса (и нажать клавишу пробела), то IntelliSense автоматически отобразит список всех переопределяемых членов родительского класса, как показано на рис. 6.5.

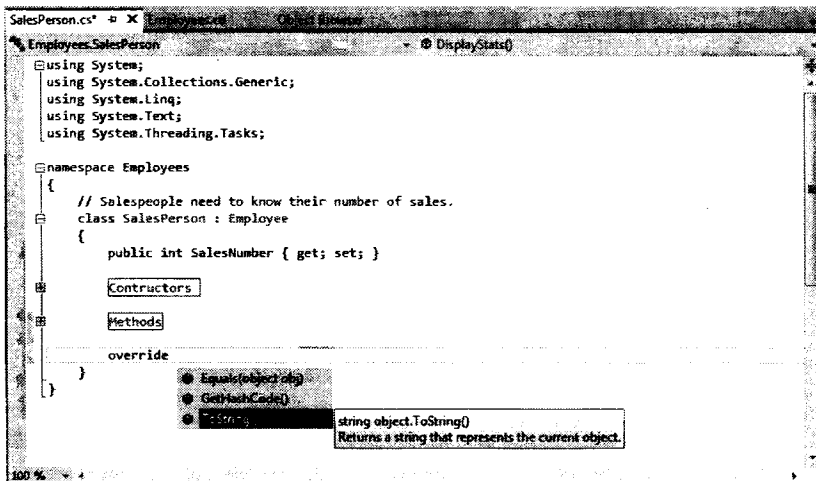


Рис. 6.5. Быстрый просмотр переопределяемых методов в Visual Studio

После выбора члена и нажатия клавиши <Enter> среда IDE реагирует автоматическим заполнением шаблона метода вместо вас. Обратите внимание, что также добавляется оператор кода, который вызывает родительскую версию виртуального члена (эту строку можно удалить, если она не нужна). Например, в случае применения этой техники во время переопределения метода `DisplayStatus()` добавится следующий автоматически сгенерированный код:

```
public override void DisplayStats()
{
    base.DisplayStats();
}
```

Запечатывание виртуальных членов

Вспомните, что ключевое слово `sealed` применяется к типу класса для предотвращения расширения другими типами его поведения через наследование. Ранее класс `PTSalesPerson` был запечатан на основе предположения, что разработчикам не имеет смысла дальше расширять эту линейку наследования.

Иногда требуется не запечатывать класс целиком, а просто предотвратить переопределение некоторых виртуальных методов в производных типах. Например, предположим, что продавцы с частичной занятостью не должны получать определенные бонусы. Чтобы предотвратить переопределение виртуального метода `GiveBonus()` в классе `PTSalesPerson`, можно запечатать этот метод в классе `SalesPerson` следующим образом:

```
// Класс SalesPerson запечатал метод GiveBonus()!
class SalesPerson : Employee
{
    ...
    public override sealed void GiveBonus(float amount)
    {
        ...
    }
}
```

Здесь `SalesPerson` действительно переопределяет виртуальный метод `GiveBonus()`, определенный в классе `Employee`, однако он явно помечен как `sealed`. Поэтому попытка переопределения этого метода в классе `PTSalesPerson` приведет к ошибке на этапе компиляции:

```
sealed class PTSalesPerson : SalesPerson
{
    public PTSalesPerson(string fullName, int age, int empID,
        float currPay, string ssn, int numbofSales)
        : base (fullName, age, empID, currPay, ssn, numbofSales)
    {
    }
    // Ошибка! Этот метод переопределять нельзя!
    public override void GiveBonus(float amount)
    {
    }
}
```

Абстрактные классы

В настоящее время базовый класс `Employee` спроектирован так, что предоставляет различные данные-члены своим наследникам, а также предлагает два виртуальных метода (`GiveBonus()` и `DisplayStatus()`), которые могут быть переопределены наследниками.

Хотя все это замечательно, у такого проектного решения имеется один неприятный побочный эффект: можно прямо создавать экземпляры базового класса Employee:

```
// Что это будет означать?
Employee X = new Employee();
```

В нашем примере базовый класс Employee имеет единственное назначение — определить общие члены для всех подклассов. По всем признакам вы не намерены позволять кому-либо создавать непосредственные экземпляры этого класса, поскольку тип Employee является слишком общим по своей природе. Например, если кто-то скажет: “Я сотрудник!”, то тут же возникнет вопрос: “Какой конкретный сотрудник?” (консультант, инструктор, административный работник, редактор, советник в правительстве и т.п.).

Учитывая, что многие базовые классы склонны быть довольно неопределенными сущностями, более удачное проектное решение для рассматриваемого примера не должно разрешать непосредственное создание в коде нового объекта Employee.

В С# можно добиться этого с использованием ключевого слова `abstract` в определении класса, таким образом, создавая *абстрактный базовый класс*:

```
// Превращение класса Employee в абстрактный
// для предотвращения прямого создания экземпляров.
abstract partial class Employee
{
    ...
}
```

После этого попытка создать экземпляр класса Employee приведет к ошибке на этапе компиляции:

```
// Ошибка! Нельзя создавать экземпляр
// абстрактного класса!
Employee X = new Employee();
```

На первый взгляд может показаться очень странным, зачем определять класс, экземпляр которого нельзя создать непосредственно. Однако вспомните, что базовые классы (абстрактные или нет) очень полезны тем, что содержат общие данные и общую функциональность для унаследованных типов. Используя эту форму абстракции, можно также моделировать общую “идею” сотрудника, а не обязательно конкретную сущность. Также следует понимать, что хотя *непосредственно* создать экземпляр абстрактного класса нельзя, он все же присутствует в памяти, когда создан экземпляр его производного класса. Таким образом, совершенно нормально (и принято) для абстрактных классов определять любое количество конструкторов, вызываемых *опосредованно* при размещении в памяти экземпляров производных классов.

Теперь получилась довольно интересная иерархия сотрудников. Позднее в этой главе, при рассмотрении правил приведения типов С#, мы добавим немного больше функциональности к этому приложению. А пока на рис. 6.6 показано текущее проектное решение.

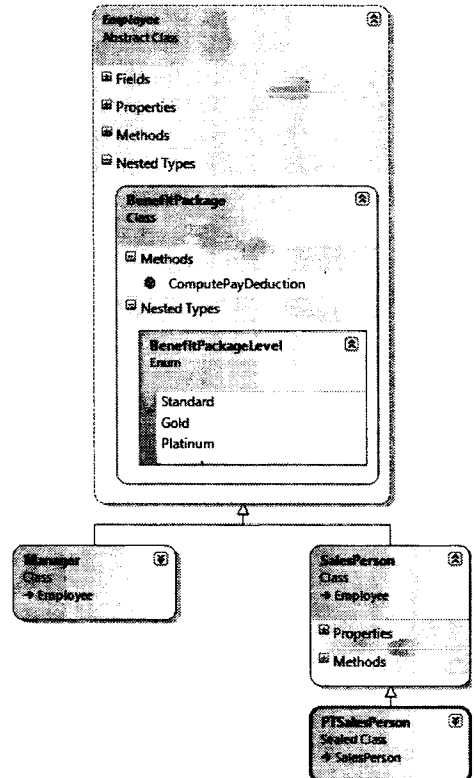


Рис. 6.6. Иерархия классов Employee

Понятие полиморфного интерфейса

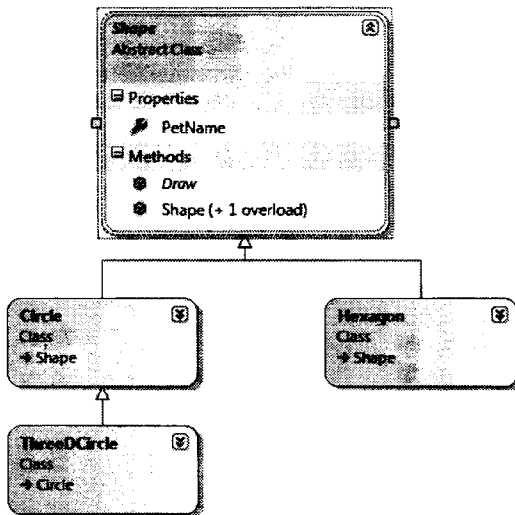


Рис. 6.7. Иерархия классов фигур

Когда класс определен как абстрактный базовый (с помощью ключевого слова `abstract`), в нем может определяться любое количество абстрактных членов. Абстрактные члены могут использоваться везде, где необходимо определить член, которые не предлагает стандартной реализации. За счет этого вы навязываете полиморфный интерфейс каждому наследнику, возлагая на них задачу реализации конкретных деталей абстрактных методов.

Полиморфный интерфейс абстрактного базового класса просто ссылается на его набор виртуальных и абстрактных методов. На самом деле это интереснее, чем может показаться на первый взгляд, поскольку данная особенность ООП позволяет строить легко расширяемое и гибкое программное обеспечение. Для иллюстрации реализуем (и слегка модифицируем) иерархию фигур, кратко описанную в главе 5 при обзоре принципов ООП.

Для начала создадим новый проект консольного приложения C# по имени `Shapes`.

Обратите внимание на рис. 6.7, что типы `Hexagon` и `Circle` расширяют базовый класс `Shape`. Подобно любому базовому классу, в `Shape` определен набор членов (в данном случае свойство `PetName` и метод `Draw()`), общих для всех наследников.

Подобно иерархии классов сотрудников, нужно запретить непосредственное создание экземпляров `Shape`, поскольку этот тип представляет слишком абстрактную концепцию. Чтобы предотвратить прямое создание экземпляров `Shape`, можно определить его как абстрактный класс. Также, учитывая, что производные типы должны уникальным образом реагировать на вызов метода `Draw()`, давайте пометим его как `virtual` и определим стандартную реализацию.

```
// Абстрактный базовый класс иерархии.
abstract class Shape
{
    public Shape(string name = "NoName")
    { PetName = name; }

    public string PetName { get; set; }

    // Единственный виртуальный метод.
    public virtual void Draw()
    {
        Console.WriteLine("Inside Shape.Draw()");
    }
}
```

Обратите внимание, что виртуальный метод `Draw()` предоставляет стандартную реализацию, которая просто выводит на консоль сообщение, информирующее о том, что вызван метод `Draw()` базового класса `Shape`. Теперь вспомните, что когда метод помечен ключевым словом `virtual`, он предоставляет стандартную реализацию, которую авто-

матически наследуют все производные типы. Если дочерний класс решит, он *может* переопределить такой метод, но он не *обязан* это делать. Учитывая это, рассмотрим следующую реализацию типов Circle и Hexagon:

```
// Circle не переопределяет Draw().
class Circle : Shape
{
    public Circle() {}
    public Circle(string name) : base(name) {}
}

// Hexagon переопределяет Draw().
class Hexagon : Shape
{
    public Hexagon() {}
    public Hexagon(string name) : base(name) {}
    public override void Draw()
    {
        Console.WriteLine("Drawing {0} the Hexagon", PetName);
    }
}
```

Польза от абстрактных методов становится совершенно ясной, как только вы запомните, что подклассы *никогда не обязаны* переопределять виртуальные методы (как в случае Circle). Поэтому если создать экземпляр типа Hexagon и Circle, то обнаружится, что Hexagon знает, как правильно “рисовать” себя (или, по крайней мере, выводит на консоль соответствующее сообщение). Однако реакция Circle слегка приведет в замешательство:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Polymorphism *****\n");
    Hexagon hex = new Hexagon("Beth");
    hex.Draw();
    Circle cir = new Circle("Cindy");
    // Вызывает реализацию базового класса!
    cir.Draw();
    Console.ReadLine();
}
```

Вывод этого метода Main() выглядит следующим образом:

```
***** Fun with Polymorphism *****

Drawing Beth the Hexagon
Inside Shape.Draw()
```

Ясно, что это не особо интеллектуальное проектное решение для текущей иерархии. Чтобы заставить каждый класс переопределять метод Draw(), можно определить Draw() как абстрактный метод класса Shape, а это означает отсутствие какой-либо стандартной реализации. Для пометки метода как абстрактного в C# служит ключевое слово **abstract**. Не забывайте, что абстрактные методы не предоставляют вообще никакой реализации:

```
abstract class Shape
{
    // Вынудить все дочерние классы определить свою визуализацию.
    public abstract void Draw();
    ...
}
```

На заметку! Абстрактные методы могут определяться только в абстрактных классах. Попытка поступить иначе приводит к ошибке на этапе компиляции.

Методы, помеченные как `abstract`, являются чистым протоколом. Они просто определяют имя, возвращаемый тип (если есть) и набор параметров (при необходимости). Здесь абстрактный класс `Shape` информирует типы-наследники о том, что у него имеется метод по имени `Draw()`, который не принимает аргументов и ничего не возвращает. О необходимых деталях должен позаботиться наследник.

С учетом этого метод `Draw()` в классе `Circle` теперь должен быть обязательно переопределен. В противном случае `Circle` также должен быть абстрактным типом и оснащен ключевым словом `abstract` (что очевидно не подходит в данном примере). Ниже показаны необходимые изменения в коде:

```
// Если не реализовать здесь абстрактный метод Draw(), то Circle также
// должен считаться абстрактным, и тогда должен быть помечен как abstract!
class Circle : Shape
{
    public Circle() {}
    public Circle(string name) : base(name) {}
    public override void Draw()
    {
        Console.WriteLine("Drawing {0} the Circle", PetName);
    }
}
```

Выражаясь кратко, теперь делается предположение о том, что любой унаследованный от `Shape` класс должен иметь уникальную версию метода `Draw()`. Для демонстрации полной картины полиморфизма рассмотрим следующий код:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Polymorphism *****\n");
    // Создать массив совместимых с Shape объектов.
    Shape[] myShapes = {new Hexagon(), new Circle(), new Hexagon("Mick"),
        new Circle("Beth"), new Hexagon("Linda")};
    // Пройти в цикле по всем элементам и взаимодействовать
    // с полиморфным интерфейсом.
    foreach (Shape s in myShapes)
    {
        s.Draw();
    }
    Console.ReadLine();
}
```

Ниже показан вывод этого метода `Main()`:

```
***** Fun with Polymorphism *****

Drawing NoName the Hexagon
Drawing NoName the Circle
Drawing Mick the Hexagon
Drawing Beth the Circle
Drawing Linda the Hexagon
```

Этот метод `Main()` иллюстрирует полиморфизм в чистом виде. Хотя невозможно напрямую создавать экземпляры абстрактного базового класса (`Shape`), можно свободно сохранять ссылки на объекты любого подкласса в абстрактной базовой переменной. Таким образом, созданный массив объектов `Shape` может хранить объекты, унаследо-

ванные от базового класса Shape (попытка поместить в массив объекты, несовместимые с Shape, приводит к ошибке на этапе компиляции).

Учитывая, что все элементы в массиве `myShapes` действительно порождены от Shape, известно, что все они поддерживают один и тот же полиморфный интерфейс (или, говоря конкретно — все они имеют метод `Draw()`). Осуществляя итерацию по массиву ссылок Shape, исполняющая система самостоятельно определяет, какой конкретный тип имеет каждый его элемент. И в этот момент вызывается корректная версия метода `Draw()`.

Эта техника также делает очень простой и безопасной задачу расширения текущей иерархии. Например, предположим, что от абстрактного базового класса Shape унаследовано еще пять классов (Triangle, Square и т.д.). Благодаря полиморфному интерфейсу, код внутри цикла `foreach` не потребует никаких изменений, если компилятор увидит, что в массив `myShapes` помещены только Shape-совместимые типы.

Соккрытие членов

Язык C# предоставляет средство, логически противоположное переопределению методов, которое называется *сокрытием*. Выражаясь формально, если производный класс определяет член, который идентичен члену, определенному в базовом классе, то производный класс *скрывает* родительскую версию. В реальном мире такая ситуация чаще всего возникает при наследовании от класса, который создавали не вы (и не ваша команда), например, в случае приобретения пакета программного обеспечения .NET у независимого поставщика.

В целях иллюстрации предположим, что вы получили от коллеги класс по имени `ThreeDCircle`, в котором определен метод `Draw()`, не принимающий аргументов:

```
class ThreeDCircle
{
    public void Draw()
    {
        Console.WriteLine("Drawing a 3D Circle");
    }
}
```

Вы обнаруживаете, что `ThreeDCircle` «является» `Circle`, поэтому наследуете его от существующего типа `Circle`:

```
class ThreeDCircle : Circle
{
    public void Draw()
    {
        Console.WriteLine("Drawing a 3D Circle");
    }
}
```

После компиляции вы получаете следующее предупреждение:

'Shapes.ThreeDCircle.Draw()' hides inherited member 'Shapes.Circle.Draw()'. To make the current member override that implementation, add the override keyword. Otherwise add the new keyword.

'Shapes.ThreeDCircle.Draw()' скрывает унаследованный член 'Shapes.Circle.Draw()'. Чтобы заставить текущий член переопределить эту реализацию, добавьте ключевое слово *override*. В противном случае добавьте ключевое слово *new*.

Проблема в том, что у вас есть производный класс (`ThreeDCircle`), содержащий метод, который является идентичным унаследованному методу. Существуют два способа решения этой проблемы. Можно просто обновить родительскую версию `Draw()`,

используя ключевое слово `override` (как рекомендует компилятор). При таком подходе тип `ThreeDCircle` может расширять стандартное поведение родительского типа, что и требовалось. Однако если доступ к коду, определяющему базовый класс, отсутствует (как обычно случается с библиотеками от независимых поставщиков), то нет возможности модифицировать метод `Draw()`, сделав его виртуальным.

В качестве альтернативы можно добавить ключевое слово `new` в определение члена `Draw()` производного типа (`ThreeDCircle` в данном случае). Делая это явно, вы устанавливаете, что реализация производного типа преднамеренно спроектирована так, чтобы игнорировать родительскую версию (в реальном проекте это может помочь, если внешнее программное обеспечение .NET каким-то образом конфликтует с вашим программным обеспечением).

```
// Этот класс расширяет Circle и скрывает унаследованный метод Draw().
class ThreeDCircle : Circle
{
    // Скрыть любую реализацию Draw(), находящуюся выше в иерархии.
    public new void Draw()
    {
        Console.WriteLine("Drawing a 3D Circle");
    }
}
```

Можно также применить ключевое слово `new` к любому члену типа, унаследованному от базового класса (полю, константе, статическому члену или свойству). В качестве еще одного примера предположим, что `ThreeDCircle()` желает скрыть унаследованное поле `shapeName`:

```
class ThreeDCircle : Circle
{
    // Скрыть поле shapeName, определенное выше в иерархии.
    protected new string shapeName;

    // Скрыть любую реализацию Draw(), находящуюся выше в иерархии.
    public new void Draw()
    {
        Console.WriteLine("Drawing a 3D Circle");
    }
}
```

И, наконец, имейте в виду, что всегда можно обратиться к реализации базового класса скрытого члена, используя явное приведение (описанное в следующем разделе). Например, это демонстрируется в следующем коде:

```
static void Main(string[] args)
{
    ...
    // Здесь вызывается метод Draw() из класса ThreeDCircle.
    ThreeDCircle o = new ThreeDCircle();
    o.Draw();

    // Здесь вызывается метод Draw() родительского класса!
    ((Circle)o).Draw();
    Console.ReadLine();
}
```

Правила приведения к базовому и производному классу

Теперь, когда вы научились строить семейства взаимосвязанных типов классов, следует ознакомиться с правилами, которым подчиняются *операции приведения классов*. Для этого вернемся к иерархии классов `Employee`, созданной ранее в главе. На платформе .NET изначальным базовым классом служит `System.Object`. Поэтому все, что создается, “является” `Object` и может трактоваться как таковой. Учитывая этот факт, в объектной переменной можно хранить ссылку на экземпляр любого типа:

```
static void CastingExamples()
{
    // Manager "является" System.Object, поэтому можно сохранять
    // ссылку на Manager в переменной типа object.
    object frank = new Manager("Frank Zappa", 9, 3000, 40000, "111-11-1111", 5);
}
```

В примере `Employees` типы `Manager`, `SalesPerson` и `PTSalesPerson` расширяют класс `Employee`, поэтому можно хранить любой из этих объектов в допустимой ссылке на базовый класс. Это значит, что следующий код также корректен:

```
static void CastingExamples()
{
    // Manager "является" System.Object, поэтому можно сохранять
    // ссылку на Manager в переменной типа object.
    object frank = new Manager("Frank Zappa", 9, 3000, 40000, "111-11-1111", 5);

    // Manager также "является" Employee.
    Employee moonUnit = new Manager("MoonUnit Zappa", 2, 3001, 20000, "101-11-1321", 1);

    // PTSalesPerson "является" SalesPerson.
    SalesPerson jill = new PTSalesPerson("Jill", 834, 3002, 100000, "111-12-1119", 90);
}
```

Первое правило приведения между типами классов гласит, что когда два класса связаны отношением “является”, всегда можно безопасно сохранить производный тип в ссылке базового класса. Формально это называется *неявным приведением*, поскольку оно “просто работает” в соответствии с законами наследования. Это делает возможным построение некоторых мощных программных конструкций. Например, предположим, что в текущем классе `Program` определен новый метод:

```
static void GivePromotion(Employee emp)
{
    // Повысить зарплату...
    // Предоставить место на парковке компании...
    Console.WriteLine("{0} was promoted!", emp.Name);
}
```

Поскольку этот метод принимает единственный параметр типа `Employee`, можно эффективно передавать этому методу любого наследника от класса `Employee`, учитывая отношение “является”:

```
static void CastingExamples()
{
    // Manager "является" System.Object, поэтому можно сохранять
    // ссылку на Manager в переменной типа object.
    object frank = new Manager("Frank Zappa", 9, 3000, 40000, "111-11-1111", 5);

    // Manager также "является" Employee.
    Employee moonUnit = new Manager("MoonUnit Zappa", 2, 3001, 20000, "101-11-1321", 1);
}
```

```

GivePromotion(moonUnit);

// PTSalesPerson "является" SalesPerson.
SalesPerson jill = new PTSalesPerson("Jill", 834, 3002, 100000, "111-12-1119", 90);
GivePromotion(jill);
}

```

Предыдущий код компилируется благодаря неявному приведению от типа базового класса (Employee) к производному классу. Однако что, если нужно также вызвать метод GivePromotion() для объекта frank (хранямого в данный момент в обобщенной ссылке System.Object)? Если вы передадите объект frank непосредственно в GivePromotion(), как показано ниже, то получите ошибку на этапе компиляции:

```

// Ошибка!
object frank = new Manager("Frank Zappa", 9, 3000, 40000, "111-11-1111", 5);
GivePromotion(frank);

```

Проблема в том, что предпринимается попытка передать переменную, которая является не Employee, а более общим объектом System.Object. Поскольку в цепочке наследования он находится выше, чем Employee, компилятор не допустит неявного приведения, стараясь обеспечить максимально возможную безопасность типов.

Несмотря на то что вы можете определить, что объектная ссылка указывает на совместимый с Employee класс в памяти, компилятор этого сделать не может, поскольку это не будет известно вплоть до времени выполнения. Чтобы удовлетворить компилятор, понадобится выполнить *явное приведение*. Второе правило приведения гласит: необходимо явно выполнять приведение “вниз”, используя операцию приведения C#. Базовый шаблон, которому нужно следовать при выполнении явного приведения, выглядит примерно так:

```
(класс_к_которому_нужно_привести) существующая_ссылка
```

Таким образом, чтобы передать переменную object методу GivePromotion(), потребуется написать следующий код:

```

// Правильно!
GivePromotion((Manager) frank);

```

Ключевое слово as

Помните, что явное приведение происходит *во время выполнения*, а не на этапе компиляции. Поэтому показанный ниже код:

```

// Нет! Приводить frank к типу Hexagon нельзя, хотя код скомпилируется!
Hexagon hex = (Hexagon) frank;

```

компилируется нормально, но вызывает ошибку времени выполнения, или, более формально — *исключение времени выполнения*. В главе 7 будут рассматриваться подробности структурированной обработки исключений, а пока следует лишь отметить, что при выполнении явного приведения можно перехватывать возможные ошибки приведения, применяя ключевые слова try и catch (см. главу 7):

```

// Перехват возможной ошибки приведения.
try
{
    Hexagon hex = (Hexagon) frank;
}
catch (InvalidCastException ex)
{
    Console.WriteLine(ex.Message);
}

```

Хотя это хороший пример защитного программирования, C# предоставляет ключевое слово `as` для быстрого определения совместимости одного типа с другим во время выполнения. С помощью ключевого слова `as` можно определить совместимость, проверив возвращенное значение на равенство `null`. Взгляните на следующий код:

```
// Использование as для проверки совместимости.
Hexagon hex2 = frank as Hexagon;
if (hex2 == null)
    Console.WriteLine("Sorry, frank is not a Hexagon...");
```

Ключевое слово `is`

Учитывая, что метод `GivePromotion()` был спроектирован для приема любого возможного типа, производного от `Employee`, может возникнуть вопрос: как этот метод может определить, какой именно производный тип был ему передан? И, кстати, если входной параметр имеет тип `Employee`, как получить доступ к специализированным членам типов `SalesPerson` и `Manager`?

В дополнение к ключевому слову `as`, в C# предлагается ключевое слово `is`, которое позволяет определить совместимость двух типов. В отличие от ключевого слова `as`, если типы не совместимы, ключевое слово `is` возвращает `false`, а не ссылку `null`. Рассмотрим следующую реализацию метода `GivePromotion()`:

```
static void GivePromotion(Employee emp)
{
    Console.WriteLine("{0} was promoted!", emp.Name);
    if (emp is SalesPerson)
    {
        Console.WriteLine("{0} made {1} sale(s)!", emp.Name,
            ((SalesPerson)emp).SalesNumber);
        Console.WriteLine();
    }
    if (emp is Manager)
    {
        Console.WriteLine("{0} had {1} stock options...", emp.Name,
            ((Manager)emp).StockOptions);
        Console.WriteLine();
    }
}
```

Здесь во время выполнения производится проверка, на что именно в памяти указывает ссылка типа базового класса. Определив, что принят объект `SalesPerson` или `Manager`, можно применить явное приведение и получить доступ к специализированным членам класса. Также обратите внимание, что окружать операции приведения конструкцией `try/catch` не обязательно, поскольку внутри контекста `if`, выполнившего проверку условия, уже известно, что приведение безопасно.

Главный родительский класс `System.Object`

В завершение этой главы исследуем детали устройства родительского главного класса всей платформы .NET — `Object`. Возможно, вы уже заметили в предыдущих разделах, что базовые классы всех иерархий (`Car`, `Shape`, `Employee`) никогда явно не указывали свои родительские классы:

```
// Кто является родительским классом Car?
class Car
{...}
```

В мире .NET каждый тип в конечном итоге наследуется от базового класса по имени `System.Object` (который в C# может быть представлен ключевым словом `object`). Класс `Object` определяет набор общих членов для каждого типа внутри инфраструктуры. Фактически при построении класса, который явно не указывает своего родителя, компилятор автоматически наследует его от `Object`. Если нужно очень четко прояснить свои намерения, можно определить класс, производный от `Object`, следующим образом:

```
// Явное наследование класса от System.Object.
class Car : object
{...}
```

Как и в любом другом классе, в `System.Object` определен набор членов. В следующем формальном определении C# обратите внимание, что некоторые из этих членов определены как `virtual`, а это говорит о том, что данный член может быть переопределен в подклассе, в то время как другие помечены как `static` (и потому вызываются только на уровне класса):

```
public class Object
{
    // Виртуальные члены.
    public virtual bool Equals(object obj);
    protected virtual void Finalize();
    public virtual int GetHashCode();
    public virtual string ToString();

    // Невиртуальные члены уровня экземпляра.
    public Type GetType();
    protected object MemberwiseClone();

    // Статические члены.
    public static bool Equals(object objA, object objB);
    public static bool ReferenceEquals(object objA, object objB);
}
```

В табл. 6.1 приведен перечень функциональности, предоставляемой некоторыми часто используемыми методами `System.Object`.

Таблица 6.1. Основные методы `System.Object`

Метод экземпляра	Назначение
<code>Equals()</code>	По умолчанию этот метод возвращает <code>true</code>, только если сравниваемые элементы ссылаются в точности на один и тот же объект в памяти. Таким образом, <code>Equals()</code> используется для сравнения объектных ссылок, а не состояния объекта. Обычно этот метод переопределяется, чтобы возвращать <code>true</code>, только если сравниваемые объекты имеют одинаковые значения внутреннего состояния. Следует отметить, что в случае переопределения <code>Equals()</code> потребуется также переопределить метод <code>GetHashCode()</code>, потому что эти методы применяются внутренне типами <code>Hashtable</code> для извлечения подобъектов из контейнера. Также вспомните из главы 4, что в классе <code>ValueType</code> этот метод переопределен для всех структур, чтобы он работал для сравнения на базе значений

Метод экземпляра	Назначение
<code>Finalize()</code>	На данный момент можно считать, что этот метод (будучи переопределенным) вызывается для освобождения любых размещенных ресурсов перед удалением объекта. Сборка мусора CLR более подробно рассматривается в главе 9
<code>GetHashCode()</code>	Этот метод возвращает значение <code>int</code> , идентифицирующее конкретный экземпляр объекта
<code>ToString()</code>	Этот метод возвращает строковое представление объекта, используя формат <code><пространство_имен>.<имя_типа></code> (т.н. <i>полностью заданное имя</i>). Этот метод часто переопределяется в подклассе для возврата строки, состоящей из пар "имя/значение", которая представляет внутреннее состояние объекта, вместо полностью заданного имени
<code>GetType()</code>	Этот метод возвращает объект <code>Type</code> , полностью описывающий объект, на который в данный момент производится ссылка. Коротко говоря, это метод идентификации типа во время выполнения (<i>Runtime Type Identification</i> — RTTI), доступный всем объектам (подробно обсуждается в главе 15)
<code>MemberwiseClone()</code>	Этот метод возвращает полную (почленную) копию текущего объекта и часто используется для клонирования объектов (см. главу 8)

Чтобы проиллюстрировать стандартное поведение, обеспечиваемое базовым классом `Object`, создадим новое консольное приложение C# по имени `ObjectOverrides`. Добавим в проект новый тип класса C#, содержащий следующее пустое определение типа по имени `Person`:

```
// Помните: Person расширяет Object.
class Person {
```

Теперь дополним метод `Main()` взаимодействием с унаследованными членами `System.Object`, как показано ниже:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with System.Object *****\n");
        Person p1 = new Person();

        // Использовать унаследованные члены System.Object.
        Console.WriteLine("ToString: {0}", p1.ToString());
        Console.WriteLine("Hash code: {0}", p1.GetHashCode());
        Console.WriteLine("Type: {0}", p1.GetType());

        // Создать другую ссылку на p1.
        Person p2 = p1;
        object o = p2;

        // Указывают ли ссылки на один и тот же объект в памяти?
        if (o.Equals(p1) && p2.Equals(o))
        {
            Console.WriteLine("Same instance!"); // Один и тот же экземпляр
        }
        Console.ReadLine();
    }
}
```

Выход этого метода `Main()` выглядит следующим образом:

```
***** Fun with System.Object *****
ToString: ObjectOverrides.Person
Hash code: 46104728
Type: ObjectOverrides.Person
Same instance!
```

Первым делом, обратите внимание, что стандартная реализация `ToString()` возвращает полностью заданное имя текущего типа (`ObjectOverrides.Person`). Как будет показано позже, при рассмотрении построения специальных пространств имен в главе 14, каждый проект C# определяет “корневое пространство имен”, название которого совпадает с именем проекта. Здесь мы создали проект под названием `ObjectOverrides`, поэтому тип `Person` (как и класс `Program`) помещен в пространство имен `ObjectOverrides`.

Стандартное поведение `Equals()` заключается в проверке того, указывают ли две переменных на один и тот же объект в памяти. Здесь создается новая переменная `Person` по имени `p1`. В этот момент новый объект `Person` помещается в управляемую кучу. Переменная `p2` также относится к типу `Person`. Однако вы не создаете новый экземпляр, а вместо этого присваиваете этой переменной ссылку `p1`. Таким образом, `p1` и `p2` указывают на один и тот же объект в памяти, как и переменная `o` (типа `object`). Учитывая, что `p1`, `p2` и `o` указывают на одно и то же местоположение в памяти, проверка эквивалентности дает положительный результат.

Хотя готовое поведение `System.Object` во многих случаях может удовлетворять всем потребностям, довольно часто специальные типы переопределяют некоторые из этих унаследованных методов. В целях иллюстрации модифицируем класс `Person`, добавив некоторые свойства, которые представляют имя, фамилию и возраст лица; все они могут быть установлены с помощью специального конструктора:

```
// Помните: Person расширяет Object.
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Age { get; set; }

    public Person(string fName, string lName, int personAge)
    {
        FirstName = fName;
        LastName = lName;
        Age = personAge;
    }
    public Person() {}
}
```

Переопределение `System.Object.ToString()`

Многие создаваемые классы (и структуры) выигрывают от переопределения `ToString()` с целью возврата строки с текстовым представлением текущего состояния экземпляра типа. Помимо прочего, это довольно полезно при отладке. То, как вы решите конструировать эту строку — дело персонального вкуса; однако рекомендованный подход состоит в разделении двоеточиями пар “имя/значение” и взятии всей строки в квадратные скобки (многие типы из библиотек базовых классов .NET следуют этому принципу). Рассмотрим следующую переопределенную версию `ToString()` для нашего класса `Person`:

```
public override string ToString()
{
    string myState;
    myState = string.Format("[First Name: {0}; Last Name: {1}; Age: {2}]",
        FirstName, LastName, Age);
    return myState;
}
```

Эта реализация `ToString()` довольно прямолинейна, учитывая, что класс `Person` состоит всего из трех фрагментов данных состояния. Однако всегда нужно помнить, что правильное переопределение `ToString()` должно также учитывать все данные, определенные выше в цепочке наследования.

Когда вы переопределяете `ToString()` для класса, расширяющего специальный базовый класс, первое, что следует сделать — получить возвращаемое значение `ToString()` от родительского класса, используя ключевое слово `base`. Получив строковые данные родителя, можно добавить к ним специальную информацию производного класса.

Переопределение `System.Object.Equals()`

Давайте также переопределим поведение `Object.Equals()` для работы с семантикой на основе значений. Вспомните, что по умолчанию `Equals()` возвращает `true`, только если два сравниваемых объекта ссылаются на один и тот же экземпляр объекта в памяти. В классе `Person` может быть полезно реализовать `Equals()` для возврата `true`, когда две сравниваемых переменных содержат одинаковые значения (т.е. фамилию, имя и возраст).

Прежде всего, обратите внимание, что входной аргумент метода `Equals()` — это общий `System.Object`. С учетом этого первое, что нужно сделать — удостовериться, что вызывающий код действительно передал тип `Person`, и для дополнительной подстраховки проверить, что входной параметр не является ссылкой `null`.

Установив, что вызывающий код передал размещенный `Person`, один из подходов состоит в реализации `Equals()` для выполнения сравнения поле за полем данных входного объекта с соответствующими данными текущего объекта:

```
public override bool Equals(object obj)
{
    if (obj is Person && obj != null)
    {
        Person temp;
        temp = (Person)obj;
        if (temp.FirstName == this.FirstName
            && temp.LastName == this.LastName
            && temp.Age == this.Age)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
    return false;
}
```

Здесь производится сравнение значения входного объекта с внутренними значениями текущего объекта (обратите внимание на применение ключевого слова `this`). Если имя, фамилия и возраст, записанные в двух объектах, идентичны, значит, эти два объ-

екта содержат одинаковые данные, потому возвращается true. Любые другие возможные результаты приводят к возвращению false.

Хотя этот подход действительно работает, представьте, насколько трудоемкой была бы реализация специального метода Equals() для нетривиальных типов, которые могут содержать десятки полей данных. Распространенным сокращением является использование собственной реализации ToString(). Если у класса имеется правильная реализация ToString(), которая учитывает все поля данных вверх по цепочке наследования, можно просто сравнить строковые данные объектов:

```
public override bool Equals(object obj)
{
    // Больше нет необходимости приводить obj к типу Person,
    // поскольку у всех имеется метод ToString().
    return obj.ToString() == this.ToString();
}
```

Обратите внимание, что в этом случае нет необходимости проверять входной аргумент на принадлежность к корректному типу (в нашем примере — Person), поскольку все классы в .NET поддерживают метод ToString(). Еще лучше то, что больше не требуется выполнять проверку равенства свойства за свойством, поскольку теперь просто проверяются значения, возвращенные методом ToString().

Переопределение System.Object.GetHashCode()

Когда класс переопределяет метод Equals(), вы также обязаны переопределить стандартную реализацию GetHashCode(). Говоря упрощенно, хеш-код — это числовое значение, представляющее объект как определенное состояние. Например, если созданы две переменных string, хранящие значение Hello, они должны давать один и тот же хеш-код. Однако если одна переменная string хранит строку в нижнем регистре (hello), должны быть получены разные хеш-коды.

По умолчанию метод System.Object.GetHashCode() для порождения хеш-значения использует текущее местоположение объекта в памяти. Тем не менее, при построении специального типа, который нужно хранить в коллекции Hashtable (из пространства имен System.Collections), этот член должен быть всегда переопределен, поскольку Hashtable внутри вызывает Equals() и GetHashCode(), чтобы извлечь правильный объект.

На заметку! Точнее говоря, класс System.Collections.Hashtable внутренне вызывает метод GetHashCode() для получения общего представления о местоположении объекта, но последующий (внутренний) вызов Equals() определяет точное соответствие.

Хотя мы не собираемся помещать Person в System.Collections.Hashtable, для полноты давайте переопределим GetHashCode(). Существует немало алгоритмов, которые могут применяться для создания хеш-кода, как весьма изощренные, так и не очень. В большинстве случаев можно сгенерировать значение хеш-кода, полагаясь на реализацию System.String.GetHashCode().

Исходя из того, что класс String уже имеет хороший алгоритм хеширования, использующий символьные данные String для сравнения хеш-значений: если вы можете идентифицировать часть данных полей класса, которая должна быть уникальной для всех экземпляров (вроде номера карточки социального страхования), просто вызовите GetHashCode() на этой части полей данных. Поскольку в классе Person определено свойство SSN, можно написать следующий код:

```
// Предположим, что имеется свойство SSN.
class Person
{
    public string SSN {get; set;}

    // Вернуть хеш-код на основе уникальных строковых данных.
    public override int GetHashCode()
    {
        return this.ToString().GetHashCode();
    }
}
```

Если же выбрать уникальный строковый элемент данных затруднительно, но есть переопределенный метод `ToString()`, вызовите `GetHashCode()` на собственном строковом представлении:

```
// Возвратить хеш-код на основе значения ToString() объекта Person.
public override int GetHashCode()
{
    return this.ToString().GetHashCode();
}
```

Тестирование модифицированного класса `Person`

Теперь, когда виртуальные члены `Object` переопределены, давайте обновим `Main()` для добавления проверки внесенных изменений.

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with System.Object *****\n");

    // ПРИМЕЧАНИЕ: эти объекты идентичны для проверки
    // методов Equals() и GetHashCode().
    Person p1 = new Person("Homer", "Simpson", 50);
    Person p2 = new Person("Homer", "Simpson", 50);

    // Получить строковые версии объектов.
    Console.WriteLine("p1.ToString() = {0}", p1.ToString());
    Console.WriteLine("p2.ToString() = {0}", p2.ToString());

    // Проверить переопределенный метод Equals().
    Console.WriteLine("p1 = p2?: {0}", p1.Equals(p2));

    // Проверить хеш-коды.
    Console.WriteLine("Same hash codes?: {0}", p1.GetHashCode() == p2.GetHashCode());
    Console.WriteLine();

    // Изменить возраст p2 и проверить снова.
    p2.Age = 45;
    Console.WriteLine("p1.ToString() = {0}", p1.ToString());
    Console.WriteLine("p2.ToString() = {0}", p2.ToString());
    Console.WriteLine("p1 = p2?: {0}", p1.Equals(p2));
    Console.WriteLine("Same hash codes?: {0}", p1.GetHashCode() == p2.GetHashCode());
    Console.ReadLine();
}
```

Ниже показан вывод:

```
***** Fun with System.Object *****
p1.ToString() = [First Name: Homer; Last Name: Simpson; Age: 50]
p2.ToString() = [First Name: Homer; Last Name: Simpson; Age: 50]
p1 = p2?: True
Same hash codes?: True
```

```
p1.ToString() = [First Name: Homer; Last Name: Simpson; Age: 50]
p2.ToString() = [First Name: Homer; Last Name: Simpson; Age: 45]
p1 = p2?: False
Same hash codes?: False
```

Статические члены System.Object

В дополнение к только что рассмотренным членам уровня экземпляра, в System.Object определены два очень полезных статических члена, которые также проверяют эквивалентность на основе значений или на основе ссылок. Рассмотрим следующий код:

```
static void StaticMembersOfObject()
{
    // Статические члены System.Object.
    Person p3 = new Person("Sally", "Jones", 4);
    Person p4 = new Person("Sally", "Jones", 4);
    Console.WriteLine("P3 and P4 have same state: {0}", object.Equals(p3, p4));
    Console.WriteLine("P3 and P4 are pointing to same object: {0}",
        object.ReferenceEquals(p3, p4));
}
```

Здесь можно просто передать два объекта (любого типа) и позволить классу System.Object автоматически определить детали.

Исходный код. Проект ObjectOverrides доступен в подкаталоге Chapter 06.

Резюме

В этой главе рассматривалась роль и подробности наследования и полиморфизма. Были представлены многочисленные новые ключевые слова и лексемы для поддержки каждой этой техники. Например, вспомните, что двоеточие применяется для установки родительского класса для заданного типа. Родительские типы могут определять любое количество виртуальных и/или абстрактных членов для установки полиморфного интерфейса. Производные типы переопределяют эти члены с использованием ключевого слова `override`.

В дополнение к построению многочисленных иерархий классов, в главе также рассказывалось о явном приведении между базовым и производным типами. Кроме того, было дано описание главного класса среди всех родительских типов библиотеки базовых классов .NET — System.Object.

Структурированная обработка исключений

В этой главе вы узнаете о том, как обрабатывать аномалии, возникающие во время выполнения кода C#, с использованием *структурированной обработки исключений* (structured exception handling — SEH). Будут описаны не только ключевые слова C#, предназначенные для этих целей (try, catch, throw, finally), но также и отличия исключений уровня приложения и системы, а также роль базового класса System.Exception. Кроме того, будет показано, как создавать специальные исключения, и рассмотрены инструменты отладки в Visual Studio, ориентированные на исключения.

Ода ошибкам и исключениям

Что бы ни нашептывало наше (порой раздутое) самолюбие, идеальных программистов не существует. Разработка программного обеспечения является сложным делом, и из-за этой сложности довольно часто даже самые лучшие программы поставляются с различными “проблемами”. В одних случаях причиной этих проблем служит “плохо написанный” код (например, допускающий выход за границы массива), а в других — некорректный пользовательский ввод, который не был учтен в кодовой базе приложения (скажем, поле для ввода телефонного номера, установленное в значение “Chucky”). Что бы ни служило причиной проблемы, в конечном итоге приложение не работает так, как ожидалось. Прежде чем переходить к рассмотрению структурированной обработки исключений, давайте сначала ознакомимся с тремя распространенными терминами, предназначенными для описания аномалий.

- **Программные ошибки.** Так обычно называются ошибки, которые допускает программист. Например, предположим, что приложение создается на неуправляемом языке C++. Если динамически выделенная память не была освобождена, приведя к утечке памяти, появляется программная ошибка.
- **Пользовательские ошибки.** С другой стороны, пользовательские ошибки обычно возникают из-за тех, кто запускает приложение, а не тех, кто его создает. Например, ввод конечным пользователем в текстовое поле неправильно оформленной строки вполне может привести к генерации такой ошибки, если в коде не была предусмотрена возможность обработки некорректного ввода.
- **Исключения.** Исключениями обычно называются аномалии, возникающие во время выполнения, которые трудно, а порой и невозможно, учесть во время программирования приложения. Примерами исключений могут быть попытка подключения к базе данных, которая больше не существует, открытие поврежденного

файла или попытка установления связи с машиной, которая в текущий момент находится в автономном режиме. В каждом из этих случаев программист (или конечный пользователь) мало что может сделать с подобными “исключительными” обстоятельствами.

С учетом этих описаний должно стать понятно, что структурированная обработка исключений в .NET — это прием, предназначенный для работы с *исключительными ситуациями*, возникающими во время выполнения. Тем не менее, даже для программных и пользовательских ошибок, которые ускользнули от глаз программиста, среда CLR будет часто генерировать соответствующее исключение, идентифицирующее возникшую проблему. В библиотеках базовых классов .NET определено множество различных исключений, таких как `FormatException`, `IndexOutOfRangeException`, `FileNotFoundException`, `ArgumentOutOfRangeException` и т.д.

В терминологии .NET под “исключением” подразумеваются программные ошибки, некорректный пользовательский ввод и ошибки времени выполнения, даже несмотря на то, что мы, программисты, можем рассматривать каждый вид ошибки как отдельную проблему. Прежде чем погружаться в детали, давайте формализуем роль структурированной обработки исключений и посмотрим, чем она отличается от традиционных методов обработки ошибок.

На заметку! Чтобы максимально упростить примеры кода, мы не будем перехватывать абсолютно все исключения, которые может выдавать конкретный метод из библиотеки базовых классов. С другой стороны, в проектах производственного уровня следует поступать согласно существующим требованиям.

Роль обработки исключений .NET

До появления платформы .NET обработка ошибок в среде операционной системы Windows представляла собой довольно запутанную смесь технологий. Многие программисты внедряли собственную логику обработки ошибок в контекст интересующего приложения. Например, команда разработчиков могла определять набор числовых констант для представления известных сбойных ситуаций и затем использовать эти константы как возвращаемые значения методов. Для примера рассмотрим следующий фрагмент кода на языке C:

```
/* Типичный механизм отлавливания ошибок в стиле C. */
#define E_FILENOTFOUND 1000

int UseFileSystem()
{
    // Предполагается, что в этой функции происходит нечто
    // такое, что приводит к возврату следующего значения.
    return E_FILENOTFOUND;
}

void main()
{
    int retVal = UseFileSystem();
    if (retVal == E_FILENOTFOUND)
        printf("Cannot find file..."); // Не удастся найти файл
}
```

Такой подход далек от идеального с учетом того факта, что константа `E_FILENOTFOUND` — это всего лишь числовое значение, которое мало что скажет о том, как решить возникшую проблему. В идеале желательно, чтобы название ошибки, описательное сообщение

и другая полезная информация об условиях возникновения этой ошибки были помещены в единственный хорошо определенный пакет (что как раз и происходит при структурированной обработке исключений).

В дополнение к приемам, изобретаемым самими разработчиками, в API-интерфейсе Windows определены сотни кодов ошибок, которые поступают в виде `#define` и `HRESULT`, а также множества вариаций простых булевских значений (`bool`, `BOOL`, `VARIANT_BOOL` и т.д.). Более того, многие разработчики COM-приложений на C++ используют небольшой набор стандартных COM-интерфейсов (например, `ISupportErrorInfo`, `IErrorInfo`, `ICreateErrorInfo`) для возврата значащей информации об ошибке клиенту COM.

Очевидная проблема со всеми этими более старыми методиками — отсутствие симметрии. Каждая из них более-менее вписывается в рамки какой-то одной технологии, одного языка и возможно даже одного проекта. Чтобы положить конец всему этому безумству, в .NET была предложена стандартная методика для генерации и выявления ошибок в исполняющей среде — структурированная обработка исключений.

Преимущество этой методики заключается в том, что разработчики теперь имеют унифицированный подход к обработке ошибок, который является общим для всех языков, ориентированных на платформу .NET. Следовательно, способ, с помощью которого программист на C# обрабатывает ошибки, синтаксически подобен способу, применяемому программистом на VB или программистом на C++, использующим C++/CLI.

Дополнительное преимущество связано с тем, что синтаксис, используемый для генерации и перехвата исключений за пределами границ сборок и машин, является идентичным. Например, при построении службы Windows Communication Foundation (WCF) на C# можно сгенерировать исключение SOAP для удаленного вызывающего кода с применением тех же ключевых слов, что и для генерации исключения внутри методов в одном приложении.

Еще одно преимущество исключений .NET состоит в том, что в отличие от запутаных числовых значений, просто обозначающих текущую проблему, они представляют собой объекты, в которых содержится читабельное описание проблемы, а также детальный снимок стека вызовов на момент первоначального возникновения исключения. Более того, конечному пользователю можно предоставить справочную ссылку, которая указывает на URL-адрес, сообщающий подробности об ошибке, а также специальные данные, определенные программистом.

Строительные блоки обработки исключений в .NET

Программирование со структурированной обработкой исключений предусматривает использование четырех связанных между собой сущностей:

- тип класса, который представляет детали исключения;
- член, способный *генерировать* экземпляр класса исключения в вызывающем коде при соответствующих обстоятельствах;
- блок кода на вызывающей стороне, обращающийся к члену, в котором может возникнуть исключение;
- блок кода на вызывающей стороне, который будет обрабатывать (или *перехватывать*) исключение в случае его возникновения.

Язык программирования C# предлагает четыре ключевых слова (`try`, `catch`, `throw` и `finally`), которые позволяют генерировать и обрабатывать исключения. Объект, который представляет текущую проблему, относится к классу, расширяющему `System.Exception` (или производный от него класс). С учетом этого давайте сначала рассмотрим роль этого базового класса, связанного с исключениями.

Базовый класс System.Exception

Все исключения в конечном итоге порождены от базового класса System.Exception, который, в свою очередь, является производным от System.Object. Ниже показана основная часть этого класса (обратите внимание, что некоторые его члены являются виртуальными и, следовательно, могут быть переопределены в производных классах):

```
public class Exception : ISerializable, _Exception
{
    // Открытые конструкторы.
    public Exception(string message, Exception innerException);
    public Exception(string message);
    public Exception();
    ...

    // Методы.
    public virtual Exception GetBaseException();
    public virtual void GetObjectData(SerializationInfo info,
        StreamingContext context);

    // Свойства.
    public virtual IDictionary Data { get; }
    public virtual string HelpLink { get; set; }
    public virtual Exception InnerException { get; }
    public virtual string Message { get; }
    public virtual string Source { get; set; }
    public virtual string StackTrace { get; }
    public MethodBase TargetSite { get; }
    ...
}
```

Как видите, многие свойства System.Exception по своей природе являются доступными только для чтения. Это объясняется тем, что стандартные значения для каждого из них обычно поставляются производными типами. Например, стандартное сообщение типа IndexOutOfRangeException выглядит так: "Index was outside the bounds of the array" ("Индекс вышел за границы массива").

На заметку! Класс Exception реализует два интерфейса .NET. Хотя интерфейсы пока еще подробно не рассматривались (см. главу 8), сейчас важно понять, что интерфейс _Exception позволяет исключению .NET обрабатываться неуправляемой кодовой базой (такой как приложение COM), а интерфейс ISerializable дает возможность объекту исключения пересекать границы (например, границы машины).

В табл. 7.1 приведены описания наиболее важных членов класса System.Exception.

Таблица 7.1. Основные члены типа System.Exception

Свойство System.Exception	Описание
Data	Это свойство, доступное только для чтения, позволяет извлекать коллекцию пар "ключ/значение" (представленную объектом, реализующим IDictionary), которая предоставляет дополнительную определяемую программистом информацию об исключении. По умолчанию эта коллекция пуста
HelpLink	Это свойство позволяет получать или устанавливать URL для доступа к справочному файлу или веб-сайту с подробным описанием ошибки

Свойство <code>System.Exception</code>	Описание
<code>InnerException</code>	Это свойство, доступное только для чтения, может использоваться для получения информации о предыдущем исключении или исключениях, которые послужили причиной возникновения текущего исключения. Запись предыдущих исключений осуществляется путем их передачи конструктору самого последнего исключения
<code>Message</code>	Это свойство, доступное только для чтения, возвращает текстовое описание заданной ошибки. Само сообщение об ошибке устанавливается как параметр конструктора
<code>Source</code>	Это свойство позволяет получать или устанавливать имя сборки или объекта, который привел к генерации исключения
<code>StackTrace</code>	Это свойство, доступное только для чтения, содержит строку, идентифицирующую последовательность вызовов, которая привела к возникновению исключения. Как нетрудно догадаться, данное свойство очень полезно во время отладки или для сохранения информации об ошибке во внешнем журнале ошибок
<code>TargetSite</code>	Это свойство, доступное только для чтения, возвращает объект <code>MethodBase</code> с описанием многочисленных деталей о методе, который привел к генерации исключения (вызов <code>ToString()</code> будет идентифицировать этот метод по имени)

Простейший пример

Для демонстрации пользы от структурированной обработки исключений мы должны создать класс, который будет генерировать исключение в надлежащих (или, можно сказать, *исключительных*) обстоятельствах. Создадим новый проект консольного приложения на C# по имени `SimpleException` и определим в нем два класса (`Car` (автомобиль) и `Radio` (радиоприемник)), связав их между собой отношением “имеет”. В классе `Radio` определен единственный метод, отвечающий за включение и выключение радиоприемника:

```
class Radio
{
    public void TurnOn(bool on)
    {
        if (on)
            Console.WriteLine("Jamming..."); // включен
        else
            Console.WriteLine("Quiet time..."); // выключен
    }
}
```

В дополнение к использованию класса `Radio` через включение/делегацию, класс `Car` (код которого показан ниже) определен так, что когда пользователь превышает предопределенную максимальную скорость (заданную с помощью константного члена `MaxSpeed`), двигатель выходит из строя, приводя объект `Car` в нерабочее состояние (что отражается закрытой переменной-членом типа `bool` по имени `carIsDead`).

Кроме того, класс `Car` имеет несколько свойств для представления текущей скорости и указанного пользователем “дружественного названия” автомобиля, а также различные конструкторы для установки состояния нового объекта `Car`.

Ниже приведено полное определение Car вместе с поясняющими комментариями.

```
class Car
{
    // Константа для представления максимальной скорости.
    public const int MaxSpeed = 100;

    // Свойства автомобиля.
    public int CurrentSpeed {get; set;}
    public string PetName {get; set;}

    // Не вышел ли автомобиль из строя?
    private bool carIsDead;

    // Автомобиль имеет радиоприемник.
    private Radio theMusicBox = new Radio();

    // Конструкторы.
    public Car() {}
    public Car(string name, int speed)
    {
        CurrentSpeed = speed;
        PetName = name;
    }

    public void CrankTunes(bool state)
    {
        // Делегировать запрос внутреннему объекту.
        theMusicBox.TurnOn(state);
    }

    // Проверить, не перегрелся ли автомобиль.
    public void Accelerate(int delta)
    {
        if (carIsDead)
            Console.WriteLine("{0} is out of order...", PetName);
        else
        {
            CurrentSpeed += delta;
            if (CurrentSpeed > MaxSpeed)
            {
                Console.WriteLine("{0} has overheated!", PetName);
                CurrentSpeed = 0;
                carIsDead = true;
            }
            else
                Console.WriteLine("=> CurrentSpeed = {0}", CurrentSpeed);
        }
    }
}
```

Теперь реализуем метод Main(), в котором объект Car будет превышать заданную максимальную скорость (установленную в 100 в классе Car):

```
static void Main(string[] args)
{
    Console.WriteLine("***** Simple Exception Example *****");
    Console.WriteLine("=> Creating a car and stepping on it!");
    Car myCar = new Car("Zippy", 20);
    myCar.CrankTunes(true);
    for (int i = 0; i < 10; i++)
        myCar.Accelerate(10);
    Console.ReadLine();
}
```

Вывод будет выглядеть следующим образом:

```
***** Simple Exception Example *****
=> Creating a car and stepping on it!
Jamming...
=> CurrentSpeed = 30
=> CurrentSpeed = 40
=> CurrentSpeed = 50
=> CurrentSpeed = 60
=> CurrentSpeed = 70
=> CurrentSpeed = 80
=> CurrentSpeed = 90
=> CurrentSpeed = 100
Zippy has overheated!
Zippy is out of order...
```

Генерация общего исключения

Теперь, когда имеется функциональный класс `Car`, рассмотрим простейший способ генерации исключения. Текущая реализация `Accelerate()` просто отображает сообщение об ошибке, если вызывающий код пытается разогнать автомобиль до скорости, превышающей максимальный предел.

Для модернизации этого метода так, чтобы когда пользователь попытался разогнать автомобиль до скорости, превышающий установленный предел, генерировалось исключение, потребуется создать и сконфигурировать новый экземпляр класса `System.Exception`, установив значение доступного только для чтения свойства `Message` через конструктор класса. Для отправки объекта ошибки обратно вызывающему коду применяется ключевое слово `throw` языка `C#`. Ниже приведен код модифицированного метода `Accelerate()`.

```
// На этот раз в случае превышения пользователем указанного
// в MaxSpeed предела должно генерироваться исключение.
public void Accelerate(int delta)
{
    if (carIsDead)
        Console.WriteLine("{0} is out of order...", PetName);
    else
    {
        CurrentSpeed += delta;
        if (CurrentSpeed >= MaxSpeed)
        {
            carIsDead = true;
            CurrentSpeed = 0;

            // Использовать ключевое слово throw для генерации исключения.
            throw new Exception(string.Format("{0} has overheated!", PetName));
        }
        else
            Console.WriteLine("=> CurrentSpeed = {0}", CurrentSpeed);
    }
}
```

Прежде чем переходить к рассмотрению перехвата данного исключения в вызывающем коде, необходимо отметить несколько интересных моментов. Прежде всего, если вы генерируете исключение, то всегда самостоятельно решаете, как в точности будет выглядеть ошибка и когда она должна выдаваться. В рассматриваемом примере предполагается, что при попытке увеличить скорость автомобиля (объекта `Car`), который уже вышел из строя, должен быть сгенерирован объект `System.Exception` для уведомления

о том, что выполнение метода `Accelerate()` не может быть продолжено (в зависимости от создаваемого приложения такое предположение может оказаться как подходящим, так и нет).

В качестве альтернативы метод `Accelerate()` можно было бы реализовать и так, чтобы он производил автоматическое восстановление, не выдавая перед этим никакого исключения. По большому счету, исключения должны генерироваться только в случае возникновения более критичных условий (например, отсутствие нужного файла, невозможность подключения к базе данных и т.п.). Принятие решения о том, что должно служить причиной генерации исключения, требует серьезного продумывания и поиска веских оснований на стадии проектирования. Для преследуемых сейчас целей будем считать, что попытка увеличить скорость неисправного автомобиля является вполне оправданной причиной для выдачи исключения.

Перехват исключений

Поскольку теперь метод `Accelerate()` способен генерировать исключение, вызывающий код должен быть готов обработать его, если оно вдруг возникнет. При вызове метода, который может генерировать исключение, должен использоваться блок `try/catch`. После перехвата объекта исключения можно обращаться к различным его членам и извлекать детальную информацию о проблеме.

Что делать с этими деталями дальше в основном зависит от вас. Может быть решено зафиксировать их в специальном файле отчета, записать в журнал событий Windows, отправить по электронной почте системному администратору или отобразить конечно-му пользователю. Для простоты мы выведем детали исключения в окне консоли.

```
// Обработка сгенерированного исключения.
static void Main(string[] args)
{
    Console.WriteLine("***** Simple Exception Example *****");
    Console.WriteLine("=> Creating a car and stepping on it!");
    Car myCar = new Car("Zippy", 20);
    myCar.CrankTunes(true);

    // Разогнаться до скорости, превышающей максимальный
    // предел автомобиля, для выдачи исключения.
    try
    {
        for(int i = 0; i < 10; i++)
            myCar.Accelerate(10);
    }
    catch(Exception e)
    {
        Console.WriteLine("\n*** Error! ***");           // ошибка
        Console.WriteLine("Method: {0}", e.TargetSite); // метод
        Console.WriteLine("Message: {0}", e.Message);   // сообщение
        Console.WriteLine("Source: {0}", e.Source);     // источник
    }

    // Ошибка была обработана, продолжается выполнение
    // следующего оператора.
    Console.WriteLine("\n***** Out of exception logic *****");
    Console.ReadLine();
}
```

По сути, блок `try` представляет собой раздел операторов, которые в ходе выполнения могут генерировать исключение. Если исключение обнаруживается, управление переходит к соответствующему блоку `catch`. С другой стороны, в случае, если код внутри

блока `try` не приводит к генерации исключения, блок `catch` полностью пропускается, и все проходит гладко. Ниже представлен вывод в результате тестового запуска данной программы.

```
***** Simple Exception Example *****
=> Creating a car and stepping on it!
Jamming...
=> CurrentSpeed = 30
=> CurrentSpeed = 40
=> CurrentSpeed = 50
=> CurrentSpeed = 60
=> CurrentSpeed = 70
=> CurrentSpeed = 80
=> CurrentSpeed = 90

*** Error! ***
Method: Void Accelerate(Int32)
Message: Zippy has overheated!
Source: SimpleException

***** Out of exception logic *****
```

Как здесь видно, после обработки исключения приложение может продолжать свою работу с оператора, который находится сразу после блока `catch`. В некоторых случаях исключение может оказаться достаточно серьезным и стать причиной для завершения работы приложения. Однако часто логика внутри обработчика исключений позволяет приложению спокойно продолжить работу (хотя, возможно, с меньшим объемом функциональности, например, без возможности подключения к удаленному источнику данных).

Конфигурирование состояния исключения

В настоящий момент объект `System.Exception`, сконфигурированный в методе `Accelerate()`, просто устанавливает значение свойства `Message` (переданное в параметре конструктора). Как было показано ранее в табл. 7.1, класс `Exception` имеет множество дополнительных членов (`TargetSite`, `StackTrace`, `HelpLink` и `Data`), которые полезны для дополнительного уточнения природы проблемы. Чтобы усовершенствовать текущий пример, давайте рассмотрим возможности каждого из этих членов более подробно.

Свойство `TargetSite`

Свойство `System.Exception.TargetSite` позволяет выяснить различные детали о методе, в котором было сгенерировано данное исключение. Как было показано в предыдущем методе `Main()`, вывод значения свойства `TargetSite` приводит к отображению возвращаемого значения, имени и типов параметров метода, который сгенерировал исключение. Тем не менее, свойство `TargetSite` возвращает не простую строку, а строго типизированный объект `System.Reflection.MethodBase`. Этот тип позволяет собрать многочисленные детали, связанные с проблемным методом, а также классом, в котором он определен. В целях иллюстрации изменим предыдущую логику в блоке `catch` следующим образом:

```
static void Main(string[] args)
{
    ...
    // Свойство TargetSite в действительности возвращает объект MethodBase.
```

```

catch(Exception e)
{
    Console.WriteLine("\n*** Error! ***");
    Console.WriteLine("Member name: {0}", e.TargetSite); // имя члена
    Console.WriteLine("Class defining member: {0}",
        e.TargetSite.DeclaringType); // класс, определяющий член
    Console.WriteLine("Member type: {0}",
        e.TargetSite.MemberType); // тип члена
    Console.WriteLine("Message: {0}", e.Message); // сообщение
    Console.WriteLine("Source: {0}", e.Source); // источник
}
Console.WriteLine("\n***** Out of exception logic *****");
Console.ReadLine();
}

```

На этот раз в коде с помощью свойства `MethodBase.DeclaringType` выясняется полностью заданное имя сгенерировавшего ошибку класса (в данном случае `SimpleException.Car`), а с помощью свойства `MemberType` объекта `MethodBase` идентифицируется тип члена (например, свойство или метод), в котором возникло исключение. Ниже показано, как теперь будет выглядеть вывод в результате выполнения логики в блоке `catch`:

```

*** Error! ***
Member name: Void Accelerate(Int32)
Class defining member: SimpleException.Car
Member type: Method
Message: Zippy has overheated!
Source: SimpleException

```

Свойство `StackTrace`

Свойство `System.Exception.StackTrace` позволяет определить последовательность вызовов, которая в результате привела к генерации исключения. Значение этого свойства никогда не устанавливается вручную — это делается автоматически во время создания объекта исключения. Чтобы проиллюстрировать это, модифицируем логику в блоке `catch` следующим образом:

```

catch(Exception e)
{
    ...
    Console.WriteLine("Stack: {0}", e.StackTrace); // стек
}

```

Если теперь снова запустить программу, можно будет увидеть в окне консоли следующие данные трассировки стека (естественно, номера строк и пути к файлам на разных машинах могут выглядеть по-разному):

```

Stack: at SimpleException.Car.Accelerate(Int32 delta)
in c:\MyApps\SimpleException\car.cs:line 65 at SimpleException.Program.Main()
in c:\MyApps\SimpleException\Program.cs:line 21

```

Строка, возвращаемая из `StackTrace`, отражает последовательность вызовов, которая привела к выдаче этого исключения. Обратите внимание, что самый нижний номер в этой строке указывает на место возникновения первого вызова в последовательности, а самый верхний — на место, где точно находится породивший проблему член. Очевидно, что такая информация очень полезна при отладке или ведении журнала для конкретного приложения, поскольку позволяет проследить весь путь к источнику ошибки.

Свойство HelpLink

Хотя свойства `TargetSite` и `StackTrace` позволяют программистам понять, почему возникло то или иное исключение, пользователям выдаваемая ими информация мало что дает. Как уже было показано ранее, для отображения конечному пользователю читабельной информации может применяться свойство `System.Exception.Message`. Кроме того, можно установить свойство `HelpLink` для указания на конкретный URL или стандартный справочный файл Windows, где содержатся более детальные сведения о проблеме.

По умолчанию значением свойства `HelpLink` является пустая строка. Присваивание этому свойству какого-то более интересного значения должно делаться перед генерацией исключения `System.Exception`. Чтобы посмотреть, как это делается, изменим метод `Car.Accelerate()` следующим образом:

```
public void Accelerate(int delta)
{
    if (carIsDead)
        Console.WriteLine("{0} is out of order...", PetName);
    else
    {
        CurrentSpeed += delta;
        if (CurrentSpeed >= MaxSpeed)
        {
            carIsDead = true;
            CurrentSpeed = 0;

            // Создать локальную переменную перед генерацией объекта Exception,
            // чтобы можно было обращаться к свойству HelpLink.
            Exception ex =
                new Exception(string.Format("{0} has overheated!", PetName));
            ex.HelpLink = "http://www.CarsRUs.com";
            throw ex;
        }
        else
            Console.WriteLine("=> CurrentSpeed = {0}", CurrentSpeed); // Вывод текущей
                                                                            // скорости
    }
}
```

Теперь можно обновить логику в блоке `catch` для вывода на консоль информации из свойства `HelpLink`:

```
catch(Exception e)
{
    ...
    Console.WriteLine("Help Link: {0}", e.HelpLink); // Ссылка для справки
}
```

Свойство Data

Свойство `Data` класса `System.Exception` позволяет заполнить объект исключения релевантной вспомогательной информацией (такой как метка времени). Свойство `Data` возвращает объект, реализующий интерфейс по имени `IDictionary`, который определен в пространстве имен `System.Collections`. В главе 8 исследуется роль программирования на основе интерфейсов, а также рассматривается пространство имен `System.Collections`. На данный момент важно понять лишь то, что словарные коллекции позволяют создавать наборы значений, извлекаемых по ключу. Взгляните на очередное изменение метода `Car.Accelerate()`:

```

public void Accelerate(int delta)
{
    if (carIsDead)
        Console.WriteLine("{0} is out of order...", PetName);
    else
    {
        CurrentSpeed += delta;
        if (CurrentSpeed >= MaxSpeed)
        {
            carIsDead = true;
            CurrentSpeed = 0;

            // Создать локальную переменную перед генерацией объекта Exception,
            // чтобы можно было обращаться к свойству HelpLink.
            Exception ex =
                new Exception(string.Format("{0} has overheated!", PetName));
            ex.HelpLink = "http://www.CarsRUs.com";

            // Указать специальные данные, касающиеся ошибки.
            ex.Data.Add("TimeStamp",
                string.Format("The car exploded at {0}", DateTime.Now)); // метка времени
            ex.Data.Add("Cause", "You have a lead foot."); // причина
            throw ex;
        }
        else
            Console.WriteLine("=> CurrentSpeed = {0}", CurrentSpeed);
    }
}

```

Для успешного перечисления пар “ключ/значение” нужно сначала указать директиву `using` для пространства имен `System.Collections`, поскольку в файле с классом, реализующим метод `Main()`, будет использоваться тип `DictionaryEntry`:

```
using System.Collections;
```

Затем потребуется обновить логику `catch`, чтобы обеспечить проверку на предмет равенства `null` значения свойства `Data` (`null` является стандартным значением). После этого можно пользоваться свойствами `Key` и `Value` типа `DictionaryEntry` для вывода специальных данных на консоль:

```

catch (Exception e)
{
    ...
    // По умолчанию поле данных является пустым, поэтому проверить его на null.
    Console.WriteLine("\n-> Custom Data:");
    if (e.Data != null)
    {
        foreach (DictionaryEntry de in e.Data)
            Console.WriteLine("-> {0}: {1}", de.Key, de.Value);
    }
}

```

Вот как теперь выглядит финальный вывод программы:

```

***** Simple Exception Example *****
=> Creating a car and stepping on it!
Jamming...
=> CurrentSpeed = 30
=> CurrentSpeed = 40
=> CurrentSpeed = 50
=> CurrentSpeed = 60
=> CurrentSpeed = 70

```

```
=> CurrentSpeed = 80
=> CurrentSpeed = 90

*** Error! ***
Member name: Void Accelerate(Int32)
Class defining member: SimpleException.Car
Member type: Method
Message: Zippy has overheated!
Source: SimpleException
Stack: at SimpleException.Car.Accelerate(Int32 delta)
      at SimpleException.Program.Main(String[] args)
Help Link: http://www.CarsRU.com

-> Custom Data:
-> Timestamp: The car exploded at 5/12/2012 9:02:12 PM
-> Cause: You have a lead foot.

***** Out of exception logic *****
```

Свойство `Data` очень полезно, т.к. позволяет упаковывать специальную информацию об ошибке, не требуя построения нового класса, расширяющего базовый класс `Exception`. Однако каким бы полезным ни было свойство `Data`, разработчики .NET-приложений по-прежнему часто предпочитают создавать строго типизированные классы исключений, которые поддерживают специальные данные с использованием строго типизированных свойств.

Этот подход позволяет вызывающему коду перехватывать конкретный производный от `Exception` тип, а не углубляться в коллекцию данных для получения дополнительных деталей. Чтобы понять, как это работает, необходимо разобраться с отличием между исключениями уровня системы и уровня приложения.

Исходный код. Проект `SimpleException` доступен в подкаталоге `Chapter 07`.

Исключения уровня системы (`System.SystemException`)

В библиотеках базовых классов .NET определено много классов, которые в конечном итоге порождены от `System.Exception`. Например, в пространстве имен `System` определены основные объекты исключений, такие как `ArgumentOutOfRangeException`, `IndexOutOfRangeException`, `StackOverflowException` и т.д. В других пространствах имен имеются исключения, отражающие их поведение. Например, в пространстве имен `System.Drawing.Printing` определены исключения, связанные с печатью, в `System.IO` — исключения, возникающие во время ввода-вывода, в `System.Data` — исключения, специфичные для баз данных, и т.д.

Исключения, которые генерируются самой платформой .NET, называются (соответственно) *системными исключениями*. Эти исключения в общем случае рассматриваются как неисправимые фатальные ошибки. Системные исключения порождены прямо от базового класса `System.SystemException`, который, в свою очередь, порожден от `System.Exception` (а тот — от класса `System.Object`):

```
public class SystemException : Exception
{
    // Разнообразные конструкторы.
}
```

Из-за того, что в `System.SystemException` никакой дополнительной функциональности помимо набора специальных конструкторов не добавлено, может возникнуть воп-

рос о том, зачем он вообще существует. Говоря просто, когда тип исключения порожден от `System.SystemException`, есть возможность определить, что сущностью, сгенерировавшей исключение, является исполняющая среда .NET, а не кодовая база функционирующего приложения. Это можно довольно легко проверить с помощью ключевого слова `is`:

```
// Действительно так! NullReferenceException является SystemException.
NullReferenceException nullRefEx = new NullReferenceException();
Console.WriteLine("NullReferenceException is-a SystemException? : {0}",
    nullRefEx is SystemException);
```

Исключения уровня приложения (`System.ApplicationException`)

Поскольку все исключения .NET — это типы классов, вполне допускается создавать собственные исключения, специфичные для приложения. Однако из-за того, что базовый класс `System.SystemException` представляет исключения, генерируемые средой CLR, может сложиться впечатление о том, что специальные исключения тоже должны быть порождены от типа `System.Exception`. Хотя можно поступать и так, но вместо этого лучше порождать их от `System.ApplicationException`:

```
public class ApplicationException : Exception
{
    // Разнообразные конструкторы.
}
```

Как и `SystemException`, класс `ApplicationException` не определяет никаких дополнительных членов кроме набора конструкторов. С точки зрения функциональности единственной целью `System.ApplicationException` является идентификация источника ошибки. При обработке исключения, производного от `System.ApplicationException`, можно смело предполагать, что исключение было инициировано кодом работающего приложения, а не библиотеками базовых классов .NET либо исполняющей средой .NET.

На заметку! На практике лишь немногие разработчики .NET строят специальные исключения, которые расширяют `ApplicationException`. Вместо этого они чаще создают подкласс `System.Exception`, хотя оба подхода формально допустимы.

Построение специальных исключений, способ первый

В то время как для сигнализации об ошибке времени выполнения можно всегда генерировать экземпляры `System.Exception` (как было показано в первом примере), иногда предпочтительнее построить *строго типизированное исключение*, которое представляет уникальные детали, связанные с текущей проблемой. Например, предположим, что необходимо создать специальное исключение (по имени `CarIsDeadException`) для предоставления деталей об ошибке, возникающей из-за увеличения скорости неисправного автомобиля. Первый шаг состоит в порождении нового класса от `System.Exception`/`System.ApplicationException` (по соглашению имена всех классов исключений заканчиваются суффиксом `Exception`; на самом деле это является рекомендуемой практикой в .NET).

На заметку! Как правило, все специальные классы исключений должны быть определены как открытые (вспомните, что стандартным модификатором доступа для не вложенных типов является `internal`). Причина в том, что исключения часто передаются за границы сборок, следовательно, они должны быть доступны вызывающей кодовой базе.

Создадим новый проект консольного приложения по имени `CustomException` и скопируем в него приведенные ранее файлы `Car.cs` и `Radio.cs`, выбрав пункт меню `Project⇒Add Existing Item` (Проект⇒Добавить существующий элемент) и изменив для ясности название пространства имен, в котором определены типы `Car` и `Radio`, с `SimpleException` на `CustomException`. После этого добавим в него следующее определение класса:

```
// Это специальное исключение описывает детали условия
// выхода автомобиля из строя.
// (Не забывайте, что можно также просто расширить Exception.)
public class CarIsDeadException : ApplicationException
{
}
```

Как и с любым классом, допускается включать произвольное количество специальных членов, которые могут быть вызваны внутри блока `catch` в вызывающем коде. Кроме того, можно также переопределять любые виртуальные члены, которые определены родительскими классами. Например, реализовать `CarIsDeadException` можно было бы за счет переопределения виртуального свойства `Message`.

Вместо заполнения словаря данных (через свойство `Data`), когда генерируется исключение, конструктор позволяет отправителю передавать метку времени и причину возникновения ошибки. Наконец, метка времени и причина возникновения ошибки могут быть получены с помощью строго типизированных свойств:

```
public class CarIsDeadException : ApplicationException
{
    private string messageDetails = String.Empty;
    public DateTime ErrorTimeStamp {get; set;}
    public string CauseOfError {get; set;}

    public CarIsDeadException() {}
    public CarIsDeadException(string message,
        string cause, DateTime time)
    {
        messageDetails = message;
        CauseOfError = cause;
        ErrorTimeStamp = time;
    }
    // Переопределение свойства Exception.Message.
    public override string Message
    {
        get
        {
            return string.Format("Car Error Message: {0}", messageDetails);
        }
    }
}
```

Здесь класс `CarIsDeadException` поддерживает закрытое поле (`messageDetails`), которое представляет данные, касающиеся текущего исключения; его можно устанавливать с использованием специального конструктора. Генерация этого исключения из метода `Accelerate()` осуществляется довольно легко. Нужно просто выделить, настроить и сгенерировать объект типа `CarIsDeadException`, а не `System.Exception` (обратите внимание, что в этом случае заполнять коллекцию данных вручную больше не требуется):

```
// Сгенерировать специальное исключение CarIsDeadException.
public void Accelerate(int delta)
{
}
```

```

CarIsDeadException ex =
    new CarIsDeadException (string.Format("{0} has overheated!", PetName),
                           "You have a lead foot", DateTime.Now);
ex.HelpLink = "http://www.CarsRUs.com";
throw ex;
...
}

```

Для перехвата такого входящего исключения теперь можно модифицировать блок `catch`, чтобы в нем перехватывался специфичный тип `CarIsDeadException` (хотя из-за того, что `System.CarIsDeadException` “является” `System.Exception`, также разрешено перехватывать `System.Exception`):

```

static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Custom Exceptions *****\n");
    Car myCar = new Car("Rusty", 90);
    try
    {
        // Отслеживание исключения.
        myCar.Accelerate(50);
    }
    catch (CarIsDeadException e)
    {
        Console.WriteLine(e.Message);
        Console.WriteLine(e.ErrorTimeStamp);
        Console.WriteLine(e.CauseOfError);
    }
    Console.ReadLine();
}

```

Теперь, понимая общий процесс построения специальных исключений, может возникнуть вопрос о том, когда это может потребоваться. Обычно необходимость в создании специальных исключений возникает, только если ошибка тесно связана с выдающим ее классом (например, специальный класс для работы с файлами может выдавать набор файловых ошибок, класс `Car` — ошибки, связанные с автомобилем, объект доступа к данным — ошибки, связанные с отдельной таблицей базы данных, и т.д.). Создание специальных исключений позволяет обеспечить вызывающий код возможностью обрабатывать многочисленные исключения на дескриптивной основе.

Построение специальных исключений, способ второй

Текущий тип `CarIsDeadException` переопределяет виртуальное свойство `System.Exception.Message` для конфигурирования специального сообщения об ошибке и предлагает два специальных свойства для учета дополнительных порций данных. Тем не менее, в реальности переопределять виртуальное свойство `Message` не требуется, т.к. можно просто передать входящее сообщение конструктору родительского класса:

```

public class CarIsDeadException : ApplicationException
{
    public DateTime ErrorTimeStamp { get; set; }
    public string CauseOfError { get; set; }
    public CarIsDeadException() { }
    // Передача сообщения конструктору родительского класса.
    public CarIsDeadException(string message, string cause, DateTime time)
        :base(message)
    {
        CauseOfError = cause;
        ErrorTimeStamp = time;
    }
}

```

Обратите внимание, что на этот раз *не* объявляется строковая переменная для представления сообщения и *не* переопределяется свойство `Message`. Вместо этого соответствующий параметр просто передается конструктору базового класса. При таком решении специальный класс исключения является не более чем уникально именованным классом, производным от `System.ApplicationException` (при необходимости с дополнительными свойствами), который лишен каких-либо переопределений базовых классов.

Не удивляйтесь, если многие (а то и все) специальные классы исключений будут следовать такому простому шаблону. Во многих случаях роль специального исключения не обязательно заключается в предоставлении дополнительной функциональности помимо той, что унаследована от базовых классов, а в обеспечении *строго именованного типа*, который четко идентифицирует природу ошибки; это позволяет клиенту предусмотреть разную логику обработки для разных типов исключений.

Построение специальных исключений, способ третий

Если необходимо создать действительно заслуживающий внимания специальный класс исключения, необходимо позаботиться о том, чтобы он соответствовал наилучшим рекомендациям .NET. В частности, это означает, что он должен соответствовать следующим характеристикам:

- наследоваться от `Exception/ApplicationException`;
- быть помеченным атрибутом `[System.Serializable]`;
- определять стандартный конструктор;
- определять конструктор, который устанавливает значение унаследованного свойства `Message`;
- определять конструктор для обработки “внутренних исключений”;
- определять конструктор для поддержки сериализации типа.

Если вы располагаете только представленным на текущий момент базовым материалом по .NET, то роль атрибутов и сериализации объектов может быть не ясна, но это нормально. Данные темы будут подробно раскрыты далее в книге (в главе 15 объясняются атрибуты, а в главе 20 — службы сериализации). Тем не менее, в завершение изучения специальных исключений ниже приведена последняя версия класса `CarIsDeadException`, в которой поддерживается каждый из упомянутых выше специальных конструкторов (остальные специальные свойства и конструкторы были показаны в примере внутри раздела “Построение специальных исключений, способ второй”):

```
[Serializable]
public class CarIsDeadException : ApplicationException
{
    public CarIsDeadException() { }
    public CarIsDeadException(string message) : base( message ) { }
    public CarIsDeadException(string message,
                               System.Exception inner)
        : base( message, inner ) { }
    protected CarIsDeadException(
        System.Runtime.Serialization.SerializationInfo info,
        System.Runtime.Serialization.StreamingContext context)
        : base( info, context ) { }

    // Любые дополнительные специальные свойства, конструкторы и члены данных...
}
```

С учетом того, что специальные исключения, создаваемые в соответствии с наилучшими практическими рекомендациям .NET, отличаются только именами, не может не радовать тот факт, что в Visual Studio предоставляется специальный фрагмент кода под названием Exception (рис. 7.1), который позволяет автоматически генерировать новый класс исключения, отвечающий требованиям рекомендаций .NET. (Вспомните из главы 2, что для активизации фрагмента кода необходимо ввести его имя, которым в данном случае является exception, и два раза нажать клавишу <Tab>.)

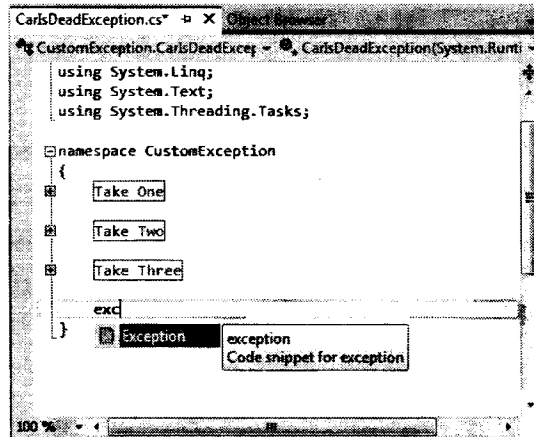


Рис. 7.1. Фрагмент кода Exception

Исходный код. Проект CustomException находится в подкаталоге Chapter 07.

Обработка нескольких исключений

В своей простейшей форме блок try сопровождается только одним блоком catch. Однако в реальности часто приходится сталкиваться с ситуациями, когда операторы внутри блока try могут генерировать несколько исключений. Создадим новый проект консольного приложения на C# по имени ProcessMultipleExceptions, добавим в него файлы Car.cs, Radio.cs и CarIsDeadException.cs из предыдущего примера CustomException (через пункт меню Project → Add Existing Item) и соответствующим образом модифицируем названия пространств имен.

Далее изменим метод Accelerate() класса Car так, чтобы он генерировал еще и предопределенное в библиотеках базовых классов исключение ArgumentOutOfRangeException, если передается недопустимый параметр (которым будет считаться любое значение меньше нуля). Обратите внимание, что конструктор этого класса исключения принимает имя проблемного аргумента в первом параметре string, за которым следует сообщение с описанием ошибки.

```
// Проверить аргумент на предмет допустимости перед продолжением.
public void Accelerate(int delta)
{
    if(delta < 0)
        throw new
            // Скорость должна быть больше нуля!
            ArgumentOutOfRangeException("delta", "Speed must be greater than zero!");
    ...
}
```

Теперь логика в блоке `catch` может специфически реагировать на каждый тип исключения:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Handling Multiple Exceptions *****\n");
    Car myCar = new Car("Rusty", 90);
    try
    {
        // Arg вызовет исключение выхода за пределы диапазона.
        myCar.Accelerate(-10);
    }
    catch (CarIsDeadException e)
    {
        Console.WriteLine(e.Message);
    }
    catch (ArgumentOutOfRangeException e)
    {
        Console.WriteLine(e.Message);
    }
    Console.ReadLine();
}
```

При написании множества блоков `catch` следует иметь в виду, что когда исключение сгенерировано, оно будет обрабатываться “первым доступным” блоком `catch`. В целях иллюстрации, что конкретно означает “первый доступный” блок `catch`, модифицируем предыдущий код, добавив еще один блок `catch`, который пытается обработать все исключения кроме `CarIsDeadException` и `ArgumentOutOfRangeException`, перехватывая общий тип `System.Exception`:

```
// Этот код не скомпилируется!
static void Main(string[] args)
{
    Console.WriteLine("***** Handling Multiple Exceptions *****\n");
    Car myCar = new Car("Rusty", 90);
    try
    {
        // Вызвать исключение выхода за пределы диапазона аргумента.
        myCar.Accelerate(-10);
    }
    catch (Exception e)
    {
        // Обработать все остальные исключения?
        Console.WriteLine(e.Message);
    }
    catch (CarIsDeadException e)
    {
        Console.WriteLine(e.Message);
    }
    catch (ArgumentOutOfRangeException e)
    {
        Console.WriteLine(e.Message);
    }
    Console.ReadLine();
}
```

Эта логика обработки исключений приводит к ошибкам на этапе компиляции. Проблема в том, что первый блок `catch` может обрабатывать любые исключения, производные от `System.Exception` (с учетом отношения “является”), в том числе

`CarIsDeadException` и `ArgumentOutOfRangeException`. Следовательно, два последних блока `catch` являются недостижимыми!

Запомните одно эмпирическое правило: необходимо обеспечить такое структурирование блоков `catch`, при котором самый первый `catch` должен перехватывать наиболее специфическое исключение (т.е. производный тип, расположенный позже всех в цепочке наследования типов исключений), а последний `catch` — наиболее общее исключение (т.е. базовый класс всей цепочки наследования, в данном случае — `System.Exception`).

Таким образом, если необходимо определить блок `catch`, который будет обрабатывать любые ошибки кроме `CarIsDeadException` и `ArgumentOutOfRangeException`, можно написать следующий код:

```
// Этот код скомпилируется без проблем.
static void Main(string[] args)
{
    Console.WriteLine("***** Handling Multiple Exceptions *****\n");
    Car myCar = new Car("Rusty", 90);
    try
    {
        // Вызвать исключение выхода за пределы диапазона аргумента.
        myCar.Accelerate(-10);
    }
    catch (CarIsDeadException e)
    {
        Console.WriteLine(e.Message);
    }
    catch (ArgumentOutOfRangeException e)
    {
        Console.WriteLine(e.Message);
    }
    // Это будет перехватывать любые исключения кроме
    // CarIsDeadException и ArgumentOutOfRangeException.
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
    Console.ReadLine();
}
```

На заметку! Везде, где только возможно, отдавайте предпочтение перехвату специфичных классов исключений, а не общих исключений типа `System.Exception`. Хотя может показаться, что это упрощает жизнь в краткосрочной перспективе (поскольку охватывает все исключения, которые пока что не волнуют), в долгосрочной перспективе во время выполнения могут возникать странные отказы, поскольку обработка более серьезной ошибки не была прямо предусмотрена в коде. Не забывайте, что финальный блок `catch`, который отвечает за обработку исключений `System.Exception`, на самом деле имеет тенденцию становиться чрезвычайно общим.

Общие операторы `catch`

В C# также поддерживается “общий” контекст `catch`, который не получает явно объект исключения, сгенерированный заданным членом:

```
// Общий catch.
static void Main(string[] args)
{
    Console.WriteLine("***** Handling Multiple Exceptions *****\n");
    Car myCar = new Car("Rusty", 90);
```

```

try
{
    myCar.Accelerate(90);
}
catch
{
    Console.WriteLine("Something bad happened...");
}
Console.ReadLine();
}

```

Очевидно, что это не самый информативный способ обработки исключений, т.к. нет никакой возможности для получения значащих данных о возникшей ошибке (вроде имени метода, стека вызовов или специального сообщения). Тем не менее, C# разрешает такую конструкцию, поскольку она может оказаться полезной, когда требуется обработать все ошибки в чрезвычайно общей манере.

Повторная генерация исключений

Когда исключение перехватывается, внутри блока `try` разрешено *повторно сгенерировать* его для передачи вверх по стеку вызовов предшествующему вызывающему коду. Для этого нужно просто воспользоваться ключевым словом `throw` в блоке `catch`. Исключение будет передано вверх по цепочке логики вызовов, что может оказаться полезным в ситуации, если блок `catch` способен обработать текущую ошибку лишь частично:

```

// Передача ответственности.
static void Main(string[] args)
{
    ...
    try
    {
        // Логика увеличения скорости автомобиля...
    }
    catch(CarIsDeadException e)
    {
        // Выполнение любую частичную обработку этой ошибки
        // и передать ответственность далее.
        throw;
    }
    ...
}

```

Следует иметь в виду, что в приведенном выше примере кода конечным получателем исключения `CarIsDeadException` будет среда CLR, т.к. в методе `Main()` оно было сгенерировано повторно. По этой причине конечному пользователю будет отображаться системное диалоговое окно с информацией об ошибке. Обычно повторная генерация частично обработанного исключения для передачи вызывающему коду производится только в случае, если он имеет возможность обработать входящее исключение более элегантным образом.

Обратите также внимание на неявную повторную генерацию объекта `CarIsDeadException` с помощью ключевого слова `throw` без аргументов. Мы не создаем здесь новый объект исключения, а просто передаем исходный объект исключения (со всей его исходной информацией). Это позволяет сохранить контекст первоначального целевого объекта.

Внутренние исключения

Нетрудно догадаться, что исключение может быть сгенерировано во время обработки другого исключения. Например, предположим, что осуществляется обработка исключения `CarIsDeadException` внутри определенного контекста `catch`, и в ходе этого процесса предпринимается попытка записать данные трассировки стека в файл `carErrors.txt` на диске `C:` (для доступа к таким ориентированным на ввод-вывод типам в директиве `using` должно быть указано пространство имен `System.IO`):

```
catch(CarIsDeadException e)
{
    // Попытаться открыть файл carErrors.txt на диске C:.
    FileStream fs = File.Open(@"C:\carErrors.txt", FileMode.Open);
    ...
}
```

Теперь, если указанный файл на диске `C:` отсутствует, то вызов `File.Open()` приведет к генерации исключения `FileNotFoundException`. Позже в этой книге вы найдете исчерпывающие сведения о пространстве имен `System.IO`, где будет показано, как программно определить, существует ли файл на жестком диске, перед попыткой его открытия (таким образом, вообще избегая исключения). Тем не менее, чтобы не отходить от темы исключений, предположим, что такое исключение было сгенерировано.

Если во время обработки исключения возникает еще одно исключение, то согласно наилучшим практическим рекомендациям, необходимо сохранить новый объект исключения как “внутреннее исключение” в новом объекте того же типа, что и у исходного исключения. Причина, по которой необходимо выделять новый объект для обрабатываемого исключения, связана с тем, что единственным способом документирования внутреннего исключения является параметр конструктора. Взгляните на следующий код:

```
catch (CarIsDeadException e)
{
    try
    {
        FileStream fs = File.Open(@"C:\carErrors.txt", FileMode.Open);
        ...
    }
    catch (Exception e2)
    {
        // Сгенерировать исключение, которое записывает новое
        // исключение, а также сообщение первого исключения.
        throw new CarIsDeadException(e.Message, e2);
    }
}
```

Обратите внимание, что в этом случае конструктору `CarIsDeadException` во втором параметре передается объект `FileNotFoundException`. После конфигурирования этот объект передается вверх по стеку вызовов следующему вызывающему коду, которым будет метод `Main()`.

С учетом того, что после `Main()` нет “следующего вызывающего кода”, который мог бы перехватить исключение, пользователю будет отображаться системное диалоговое окно с сообщением об ошибке. Во многом подобно повторной генерации исключения, запись внутренних исключений обычно полезна только тогда, когда вызывающий код способен обрабатывать данное исключение более элегантно. В этом случае внутри логики `catch` вызывающего кода с помощью свойства `InnerException` можно извлечь детали объекта внутреннего исключения.

Блок `finally`

В контексте `try/catch` можно также определять необязательный блок `finally`. Этот блок гарантирует, что заданный внутри него набор операторов будет выполняться *всегда*, независимо от того, возникло исключение (любого типа) или нет. В целях иллюстрации предположим, что перед выходом из метода `Main()` радиоприемник в автомобиле должен всегда выключаться, независимо от обрабатываемого исключения:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Handling Multiple Exceptions *****\n");
    Car myCar = new Car("Rusty", 90);
    myCar.CrankTunes(true);

    try
    {
        // Логика, связанная с увеличением скорости автомобиля.
    }
    catch (CarIsDeadException e)
    {
        // Обработать CarIsDeadException.
    }
    catch (ArgumentOutOfRangeException e)
    {
        // Обработать ArgumentOutOfRangeException.
    }
    catch (Exception e)
    {
        // Обработать любой другой объект Exception.
    }
    finally
    {
        // Это код будет выполняться всегда, возникало исключение или нет.
        myCar.CrankTunes(false);
    }
    Console.ReadLine();
}
```

При отсутствии блока `finally` в случае возникновения исключения радиоприемник не выключался бы (что может как являться, так и не являться проблемой). В более реалистичном сценарии, когда необходимо освободить объекты, закрыть файл либо отключиться от базы данных (или чего-то подобного), блок `finally` представляет собой идеальное место для выполнения надлежащей очистки.

Какие исключения могут генерировать методы?

Учитывая, что метод в .NET Framework может генерировать любое количество исключений в разных обстоятельствах, возникает вполне логичный вопрос: как узнать, какие исключения может генерировать определенный метод из библиотеки базовых классов? Окончательный ответ прост: нужно заглянуть в документацию .NET Framework 4.5 SDK. В этой документации для каждого метода перечислены все исключения, которые он может генерировать. В среде Visual Studio можно просмотреть список всех исключений, генерируемых членом библиотеки базовых классов, наведя курсор мыши на имя члена в окне кода (рис. 7.2).

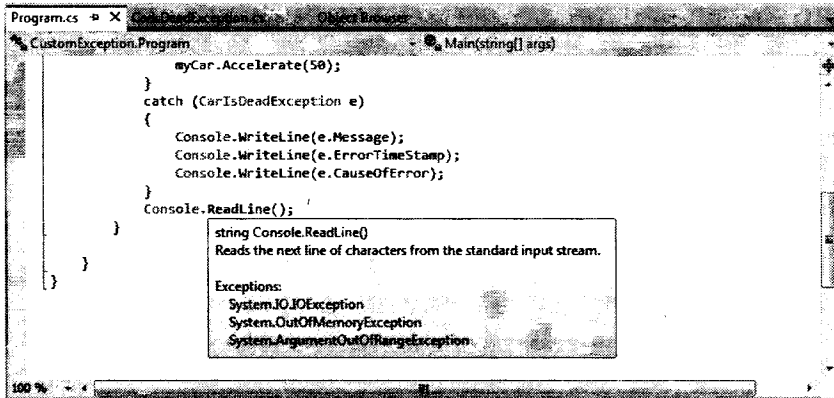


Рис. 7.2. Идентификация исключений, генерируемых определенным методом

На заметку! Программисты, пришедшие в .NET из мира Java, должны понять, что члены типов не прототипируются набором исключений, которые они могут генерировать (другими словами, платформа .NET не поддерживает проверяемые исключения). Хорошо это или плохо, но вы не обязаны обрабатывать все и каждое исключение, генерируемое определенным членом.

Результат наличия необработанных исключений

Сейчас вам наверняка интересно узнать, что произойдет, если сгенерированное исключение не будет обработано. Предположим, что логика внутри Main() увеличивает скорость объекта Car до значения, превышающего допустимый максимум, а блок try/catch отсутствует:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Handling Multiple Exceptions *****\n");
    Car myCar = new Car("Rusty", 90);
    myCar.Accelerate(500);
    Console.ReadLine();
}
```

В результате игнорирования исключения конечный пользователь приложения получит диалоговое окно с сообщением о необработанном исключении, которое показано на рис. 7.3.

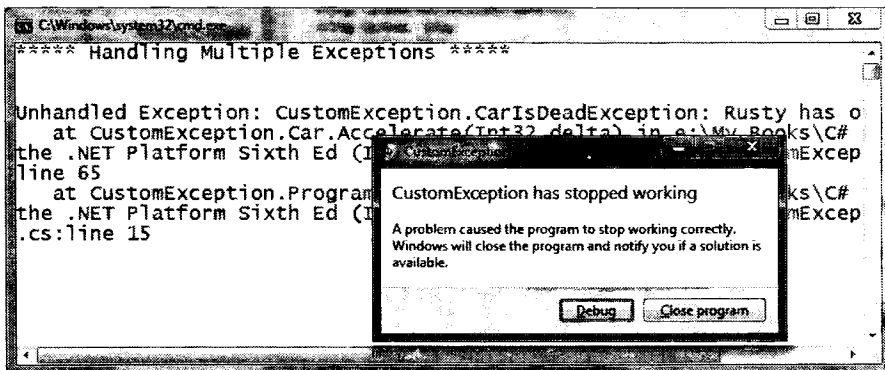


Рис. 7.3. Результат отсутствия обработки исключения

Отладка необработанных исключений с использованием Visual Studio

Среда Visual Studio предлагает набор инструментов, которые помогают отлаживать необработанные специальные исключения. В целях иллюстрации предположим, что скорость объекта Car была увеличена до значения, превышающего допустимый максимум. Если запустить сеанс отладки в Visual Studio (выбрав пункт меню Debug⇒Start (Отладка⇒Начать)), во время генерации необработанного исключения произойдет останов. Вдобавок откроется окно (рис. 7.4), отображающее значение свойства Message.

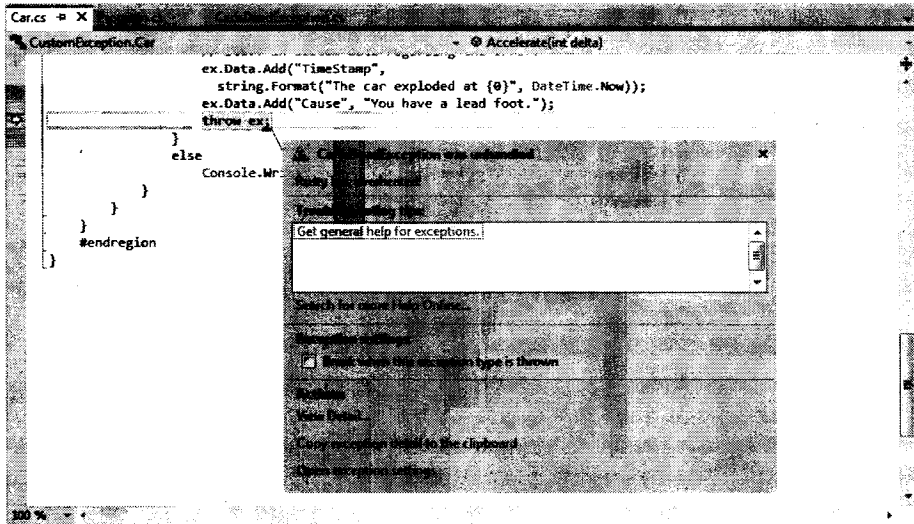


Рис. 7.4. Отладка необработанных специальных исключений в Visual Studio

На заметку! Если не обработать исключение, сгенерированное методом из библиотек базовых классов .NET, отладчик Visual Studio остановит выполнение на операторе, который вызвал этот проблемный метод.

Щелчок на ссылке View Detail (Показать подробности) в этом окне приводит к отображению подробной информации о состоянии объекта (рис. 7.5).

Исходный код. Проект ProcessMultipleExceptions доступен в подкаталоге Chapter 07.

Резюме

В этой главе была раскрыта роль структурированной обработки исключений. Когда методу необходимо отправить объект ошибки вызывающему коду, он должен создать, сконфигурировать и сгенерировать специфичный объект производного от System.Exception типа с помощью ключевого слова throw языка C#. Вызывающий код имеет возможность обрабатывать любые входящие исключения, используя ключевое слово catch и необязательный блок finally.

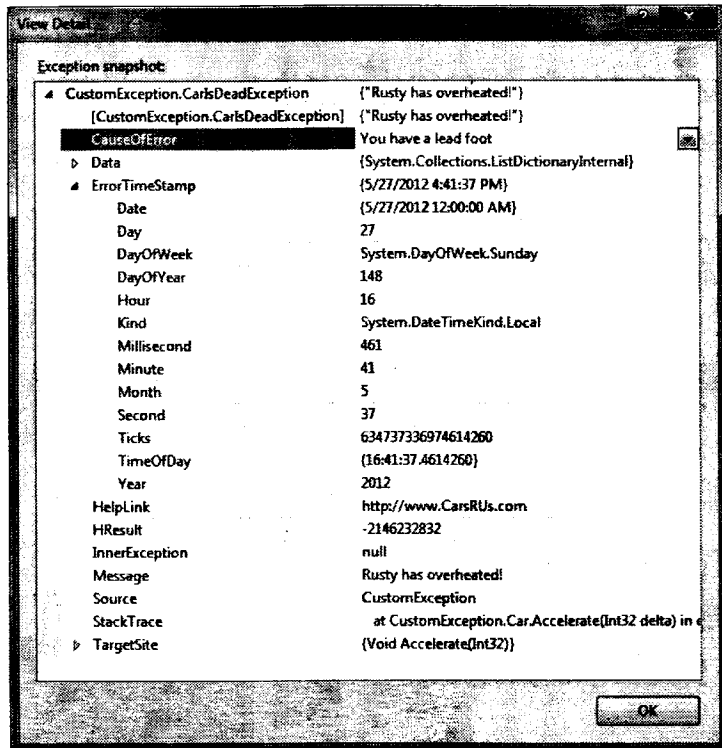


Рис. 7.5. Просмотр деталей исключения

При построении собственных специальных исключений в конечном итоге создается класс, производный от `System.ApplicationException`, который обозначает исключение, генерируемое текущим выполняющимся приложением. В противоположность этому, объекты ошибок, производные от `System.SystemException`, представляют критические (и фатальные) ошибки, генерируемые средой CLR. Наконец, в главе были продемонстрированы различные инструменты внутри Visual Studio, которые могут использоваться для создания специальных исключений (в соответствии с наилучшими практическими рекомендациями .NET), а также для отладки исключений.

Работа с интерфейсами

Материал этой главы основан на начальных знаниях объектно-ориентированной разработки и посвящен теме программирования на основе интерфейсов. Вы узнаете, как определять и реализовывать интерфейсы, а также ознакомитесь с преимуществами построения типов, поддерживающих несколько видов поведения. В ходе изложения будут рассмотрены и другие связанные с этим темы, такие как получение ссылок на интерфейсы, явная реализация интерфейсов и построение иерархий интерфейсов. Будет также описано несколько стандартных интерфейсов, определенных внутри библиотек базовых классов .NET. Вы увидите, что специальные классы и структуры могут реализовать эти предопределенные интерфейсы для поддержки нескольких полезных поведений, таких как клонирование, перечисление и сортировка объектов.

Понятие интерфейсных типов

Для начала давайте ознакомимся с формальным определением *интерфейсного типа*. *Интерфейс* представляет собой просто именованный набор *абстрактных членов*. Как упоминалось в главе 6, абстрактные методы являются чистым протоколом, поскольку они не предоставляют стандартной реализации. Специфичные члены, определяемые интерфейсом, зависят от того, какое точно поведение он моделирует. Другими словами, интерфейс выражает *поведение*, которое заданный класс или структура может избрать для поддержки. Более того, как будет показано далее в этой главе, класс или структура может поддерживать столько интерфейсов, сколько необходимо, и, следовательно, тем самым поддерживать множество поведений.

Нетрудно догадаться, что в библиотеках базовых классов .NET поставляются сотни предопределенных интерфейсных типов, которые реализованы различными классами и структурами. Например, как будет показано в главе 21, инфраструктура ADO.NET содержит множество поставщиков данных, которые позволяют взаимодействовать с определенной системой управления базами данных. Это означает, что в ADO.NET на выбор доступно несколько объектов подключения (SqlConnection, OleDbConnection, OdbcConnection и т.д.).

Несмотря на то что каждый из этих объектов подключения имеет уникальное имя, определен в отдельном пространстве имен и (в некоторых случаях) упакован в отдельную сборку, все они реализуют общий интерфейс под названием IDbConnection:

```
// Интерфейс IDbConnection определяет общий набор членов,
// поддерживаемых всеми объектами подключения.
public interface IDbConnection : IDisposable
{
    // Методы.
    IDbTransaction BeginTransaction();
    IDbTransaction BeginTransaction(IsolationLevel il);
```

```

void ChangeDatabase(string databaseName);
void Close();
IDbCommand CreateCommand();
void Open();

// Свойства.
string ConnectionString { get; set; }
int ConnectionTimeout { get; }
string Database { get; }
ConnectionState State { get; }
}

```

На заметку! По соглашению имена всех интерфейсов .NET снабжаются префиксом в виде заглавной буквы "I". При создании собственных специальных интерфейсов рекомендуется тоже следовать этому соглашению.

В данный момент детали того, что делают все эти члены, не важны. Просто запомните, что интерфейс `IDbConnection` определяет набор членов, которые являются общими для всех объектов подключения ADO.NET. Учитывая это, имеется гарантия, что каждый объект подключения поддерживает такие члены, как `Open()`, `Close()`, `CreateCommand()` и т.д. Более того, поскольку методы этого интерфейса всегда являются абстрактными, в каждом объекте подключения они могут быть реализованы собственным уникальным образом.

В оставшейся части книги вы столкнетесь с десятками интерфейсов, поставляемых в библиотеках базовых классов .NET. Вы увидите, что эти интерфейсы могут быть реализованы в собственных специальных классах и структурах для определения типов, которые тесно интегрированы с платформой .NET.

Сравнение интерфейсных типов и абстрактных базовых классов

После изучения главы 6 интерфейсный тип может показаться очень похожим на абстрактный базовый класс. Вспомните, что когда класс помечается как абстрактный, он *может* определять любое количество абстрактных членов для предоставления полиморфного интерфейса всем производным типам. Однако даже если класс действительно определяет набор абстрактных членов, он также может определять любое количество конструкторов, полей данных, неабстрактных членов (с реализацией) и т.п. С другой стороны, интерфейсы могут содержать *только* определения абстрактных членов.

Полиморфный интерфейс, устанавливаемый абстрактным родительским классом, обладает одним серьезным ограничением: члены, определенные абстрактным родительским классом, поддерживаются *только производными типами*. Тем не менее, в крупных программных системах очень часто разрабатываются многочисленные иерархии классов, не имеющие общего родителя за исключением `System.Object`. Учитывая, что абстрактные члены в абстрактном базовом классе применимы *только* к производным типам, не существует никакого способа настройки типов в разных иерархиях на поддержку одного и того же полиморфного интерфейса. Для примера предположим, что определен следующий абстрактный класс:

```

public abstract class CloneableType
{
    // Только производные типы могут поддерживать этот
    // "полиморфный интерфейс". Классы в других иерархиях
    // не имеют доступа к этому абстрактному члену.
    public abstract object Clone();
}

```

При таком определении поддерживать метод `Clone()` могут только члены, расширяющие класс `CloneableType`. Если создается новый набор классов, которые не расширяют этот базовый класс, воспользоваться этим полиморфным интерфейсом не удастся. Кроме того, как уже упоминалось, в С# не поддерживается множественное наследование для классов. Следовательно, когда нужно создать класс `MiniVan`, являющийся и `Car`, и `CloneableType`, поступить так, как показано ниже, не получится:

```
// Нельзя! Множественное наследование для классов в С# не разрешено.
public class MiniVan : Car, CloneableType
{
}
```

Нетрудно догадаться, что здесь на помощь приходят интерфейсные типы. После того как интерфейс определен, он может быть реализован любым классом или структурой, в любой иерархии и внутри любого пространства имен или сборки (написанной на любом языке программирования .NET). Вы увидите, что интерфейсы являются чрезвычайно полиморфными. Возьмем стандартный интерфейс .NET под названием `ICloneable`, определенный в пространстве имен `System`. Этот интерфейс определяет единственный метод по имени `Clone()`:

```
public interface ICloneable
{
    object Clone();
}
```

Заглянув в документацию .NET Framework 4.5 SDK, можно обнаружить, что этот интерфейс реализован очень многими по виду несвязанными типами (`System.Array`, `System.Data.SqlClient.SqlConnection`, `System.OperatingSystem`, `System.String` и т.д.). Хотя эти типы не имеют общего родителя (кроме `System.Object`), их можно трактовать полиморфным образом через интерфейсный тип `ICloneable`.

Например, если есть метод по имени `CloneMe()`, принимающий параметр интерфейсного типа `ICloneable`, этому методу можно передавать любой объект, который реализует указанный интерфейс. Рассмотрим следующий простой класс `Program`, определенный в консольном приложении по имени `ICloneableExample`:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** A First Look at Interfaces *****\n");
        // Все эти классы поддерживают интерфейс ICloneable.
        string myStr = "Hello";
        OperatingSystem unixOS = new OperatingSystem(PlatformID.Unix, new Version());
        System.Data.SqlClient.SqlConnection sqlCnn =
            new System.Data.SqlClient.SqlConnection();
        // Следовательно, все они могут быть переданы методу, принимающему ICloneable.
        CloneMe(myStr);
        CloneMe(unixOS);
        CloneMe(sqlCnn);
        Console.ReadLine();
    }
    private static void CloneMe(ICloneable c)
    {
        // Клонировать то, что получено, и вывести его имя.
        object theClone = c.Clone();
        Console.WriteLine("Your clone is a: {0}",
            theClone.GetType().Name);
    }
}
```


После запуска этого приложения в окне консоли выводится имя каждого класса, полученное с помощью метода `GetType()`, унаследованного от `System.Object`. Как объясняется в главе 15, этот метод (и службы рефлексии .NET) позволяют разобрать строение любого типа во время выполнения. В любом случае ниже показан вывод предыдущей программы:

```
***** A First Look at Interfaces *****
Your clone is a: String
Your clone is a: OperatingSystem
Your clone is a: SqlConnection
```

Другое ограничение абстрактных базовых классов связано с тем, что *каждый производный тип* должен предоставить реализации для всего набора абстрактных членов. Чтобы увидеть, в чем заключается проблема, давайте вспомним иерархию фигур, которая была определена в главе 6. Предположим, что в базовом классе `Shape` определен новый абстрактный метод по имени `GetNumberOfPoints()`, который позволяет производным типам возвращать количество вершин, требуемых для визуализации фигуры:

```
abstract class Shape
{
    ...
    // Каждый производный класс должен теперь поддерживать этот метод!
    public abstract byte GetNumberOfPoints();
}
```

Очевидно, что изначально единственным классом, который в принципе имеет вершины, является `Hexagon`. Но теперь из-за внесенного обновления *каждый* производный класс (`Circle`, `Hexagon` и `ThreeDCircle`) должен предоставить конкретную реализацию `GetNumberOfPoints()`, даже если в этом нет никакого смысла. В таком случае снова на помощь приходит интерфейсный тип. Если определить интерфейс, который представляет поведение “наличия вершин”, можно будет просто подключить его к классу `Hexagon`, не затрагивая `Circle` и `ThreeDCircle`.

Исходный код. Проект `ICloneableExample` доступен в подкаталоге `Chapter 08`.

Определение специальных интерфейсов

Теперь, когда вы лучше понимаете общую роль интерфейсов, давайте рассмотрим пример определения и реализации специальных интерфейсов. Для начала создадим новое консольное приложение по имени `CustomInterface`. Далее с помощью пункта меню `Project⇒Add Existing Item` (Проект⇒Добавить существующий элемент) вставим в него файлы, содержащие определения типов фигур (в примерах кода это `Shape.cs` и `DerivedShapes.cs`), которые были созданы в главе 6. После этого переименуем пространство имен, в котором содержатся определения отвечающих за фигуры типов, на `CustomInterface` (просто чтобы избежать импортирования определений пространства имен в новом проекте):

```
namespace CustomInterface
{
    // Здесь определяются типы фигур...
}
```

Теперь, выбрав пункт меню `Project⇒Add Existing Item`, вставим в проект новый интерфейс по имени `IPointy`, как показано на рис. 8.1.

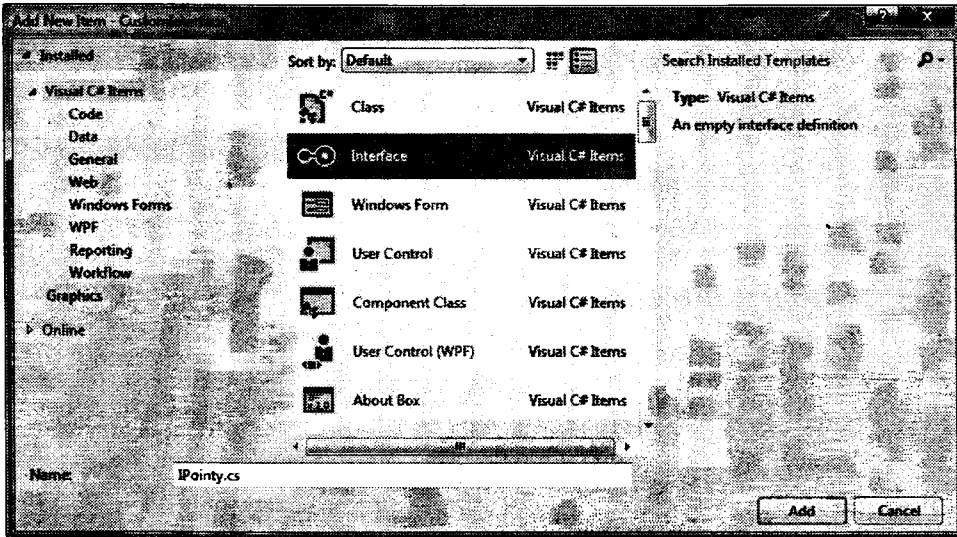


Рис. 8.1. Интерфейсы, как и классы, могут определяться в файлах *.cs

На синтаксическом уровне любой интерфейс определяется с использованием ключевого слова `interface` языка C#. В отличие от классов, для интерфейсов никогда не указывается базовый класс (даже `System.Object`; хотя, как будет показано позже в главе, могут задаваться базовые интерфейсы). Кроме того, для членов интерфейса никогда не указываются модификаторы доступа (т.к. все члены интерфейса являются неявно открытыми и абстрактными). Ниже приведен пример определения специального интерфейса на C#:

```
// Этот интерфейс определяет поведение "наличия вершин".
public interface IPointy
{
    // Член является неявно открытым и абстрактным.
    byte GetNumberOfPoints();
}
```

Вспомните, что при определении членов интерфейса контекст их реализации не определяется. Интерфейсы — это чистый протокол, поэтому реализация для них никогда не предоставляется (за это отвечает поддерживающий класс или структура). Таким образом, следующая версия `IPointy` приведет к выдаче различных ошибок на этапе компиляции:

```
// Внимание! В этом коде полно ошибок!
public interface IPointy
{
    // Ошибка! Интерфейсы не могут иметь поля данных!
    public int numbOfPoints;

    // Ошибка! Интерфейсы не могут иметь конструкторы!
    public IPointy() { numbOfPoints = 0; };

    // Ошибка! Интерфейсы не могут предоставлять реализацию членов!
    byte GetNumberOfPoints() { return numbOfPoints; }
}
```

В любом случае, этот начальный интерфейс `IPointy` определяет единственный метод. Однако интерфейсные типы .NET могут также определять любое количество прототипов свойств. Например, давайте модифицируем интерфейс `IPointy` так, чтобы в нем использовалось доступное только для чтения свойство, а не традиционный метода доступа:

```
// Определение свойства, доступного только для чтения.
public interface IPointy
{
    // Свойство, доступное для чтения и для записи,
    // в этом интерфейсе может выглядеть так:
    // retVal PropName { get; set; }
    //
    // а свойство, доступное только для записи - так:
    // retVal PropName { set; }

    byte Points { get; }
}
```

На заметку! Интерфейсные типы также могут содержать определения событий (глава 10) и индексаторов (глава 11).

Сами по себе типы интерфейсов довольно бесполезны, поскольку представляют собой не более чем просто именованную коллекцию абстрактных членов. Например, размещать типы интерфейсов таким же образом, как классы или структуры, невозможно:

```
// Внимание! Размещать типы интерфейсов не допускается!
static void Main(string[] args)
{
    IPointy p = new IPointy(); // Ошибка на этапе компиляции!
}
```

Интерфейсы ничего особого не дают до тех пор, пока не будут реализованы классом или структурой. Здесь `IPointy` представляет собой интерфейс, который выражает поведение “наличия вершин”. Идея проста: некоторые классы в иерархии фигур (например, `Hexagon`) имеют вершины, а некоторые (вроде `Circle`) — нет.

Реализация интерфейса

Чтобы расширить функциональность класса (или структуры) за счет поддержки интерфейсов, необходимо добавить в его определение список нужных интерфейсов, разделенных запятыми. Имейте в виду, что прямой базовый класс должен быть первым в этом списке, т.е. сразу же после двоеточия. Когда тип класса порожден непосредственно от `System.Object`, допускается перечислять только поддерживаемые интерфейсы, т.к. компилятор C# автоматически расширяет типы от `System.Object`, если не указано иначе. К слову, поскольку структуры всегда являются производными от `System.ValueType` (см. главу 4), интерфейсы просто должны перечисляться после определения структуры. Ниже приведены примеры.

```
// Этот класс порожден от System.Object
// и реализует единственный интерфейс.
public class Pencil : IPointy
{...}

// Этот класс тоже порожден от System.Object
// и реализует единственный интерфейс.
public class SwitchBlade : object, IPointy
{...}

// Этот класс порожден от специального базового
// класса и реализует единственный интерфейс.
public class Fork : Utensil, IPointy
{...}
```

```
// Эта структура неявно порождена от System.ValueType
// и реализует два интерфейса.
public struct PitchFork : IClonable, IPointy
{...}
```

Важно понимать, что реализация интерфейса работает по принципу “все или ничего”. Поддерживающий тип не имеет возможности выбирать, какие члены должны быть реализованы, а какие — нет. Учитывая, что в интерфейсе `IPointy` определено единственное доступное только для чтения свойство, накладные расходы невелики. Однако в случае реализации интерфейса, который определяет десять членов (вроде показанного ранее `IDbConnection`), тип будет отвечать за предоставление деталей для всех десяти абстрактных членов.

Вернемся к рассматриваемому примеру и добавим в него новый тип класса по имени `Triangle`, который является `Shape` и поддерживает `IPointy`. Обратите внимание, что реализация доступного только для чтения свойства `Points` просто возвращает соответствующее количество вершин (3).

```
// Новый производный от Shape класс по имени Triangle.
class Triangle : Shape, IPointy
{
    public Triangle() { }
    public Triangle(string name) : base(name) { }
    public override void Draw()
    { Console.WriteLine("Drawing {0} the Triangle", PetName); }

    // Реализация IPointy.
    public byte Points
    {
        get { return 3; }
    }
}
```

Теперь изменим существующий тип `Hexagon` так, чтобы он тоже поддерживал интерфейс `IPointy`:

```
// Hexagon now implements IPointy.
class Hexagon : Shape, IPointy
{
    public Hexagon() { }
    public Hexagon(string name) : base(name) { }
    public override void Draw()
    { Console.WriteLine("Drawing {0} the Hexagon", PetName); }

    // Реализация IPointy.
    public byte Points
    {
        get { return 6; }
    }
}
```

Чтобы подытожить все изученное к этому моменту, на рис. 8.2 приведена диаграмма классов Visual Studio, на которой все совместимые с `IPointy` классы представлены с помощью популярной нотации “леденца на палочке”. Обратите внимание, что `Circle` и `ThreeDCircle` не реализуют `IPointy`, т.к. это поведение не имеет смысла в данных классах.

На заметку! Чтобы скрыть или отобразить имена интерфейсов в визуальном конструкторе классов, щелкните правой кнопкой мыши на значке, представляющем интерфейс, и выберите в контекстном меню пункт `Collapse` (Свернуть) или `Expand` (Развернуть).

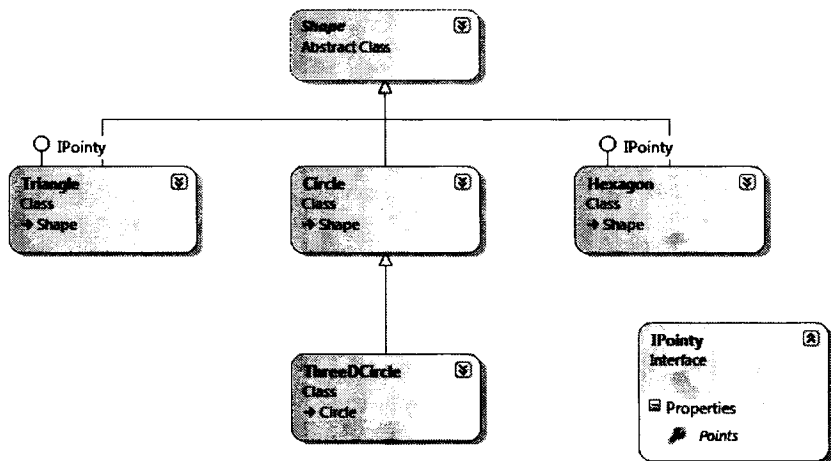


Рис. 8.2. Иерархия фигур, теперь с интерфейсами

Вызов членов интерфейса на уровне объектов

Теперь, при наличии нескольких классов, поддерживающих интерфейс `IPointy`, давайте посмотрим, как взаимодействовать с этой новой функциональностью. Самый простой способ взаимодействия с функциональностью, предлагаемой указанным интерфейсом, предусматривает вызов его членов прямо на уровне объектов (при условии, что члены этого интерфейса не реализованы явно, о чем более подробно рассказывается в разделе “Явная реализация интерфейсов” далее в главе). Например, рассмотрим следующий метод `Main()`:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Interfaces *****\n");
    // Вызвать свойство Points, определенное интерфейсом IPointy.
    Hexagon hex = new Hexagon();
    Console.WriteLine("Points: {0}", hex.Points);
    Console.ReadLine();
}

```

Такой подход нормально работает в этом конкретном случае, поскольку здесь точно известно, что тип `Hexagon` реализует упомянутый интерфейс и, следовательно, поддерживает свойство `Points`. Однако в других случаях определить, какие интерфейсы поддерживает данный тип, может быть невозможно. Например, предположим, что имеется массив, содержащий 50 совместимых с `Shape` типов, причем только некоторые из них поддерживают `IPointy`. Очевидно, что при попытке обратиться к свойству `Points` для типа, который не реализует `IPointy`, возникнет ошибка. Как же динамически определить, поддерживает ли класс или структура нужный интерфейс?

Одним из способов для определения во время выполнения того, поддерживает ли тип конкретный интерфейс, является применение явного приведения. Если тип не поддерживает запрашиваемый интерфейс, сгенерируется исключение `InvalidCastException`. Чтобы аккуратно учесть эту возможность, необходимо воспользоваться структурированной обработкой исключений, как в следующем примере:

```
static void Main(string[] args)
{
    ...
    // Перехватить возможное исключение InvalidCastException.
    Circle c = new Circle("Lisa");
    IPointy itfPt = null;
    try
    {
        itfPt = (IPointy)c;
        Console.WriteLine(itfPt.Points);
    }
    catch (InvalidCastException e)
    {
        Console.WriteLine(e.Message);
    }
    Console.ReadLine();
}
```

Хотя можно было бы применить логику try/catch и надеяться на лучшее, в идеале хотелось бы выяснять, какие интерфейсы поддерживаются, перед обращением к их членам. Давайте рассмотрим два способа, которыми это можно делать.

Получение ссылок на интерфейсы: ключевое слово as

Для определения, поддерживает ли данный тип тот или иной интерфейс, можно воспользоваться ключевым словом **as**, которое было представлено в главе 6. Если объект может быть интерпретирован как указанный интерфейс, то возвращается ссылка на интересующий интерфейс, а если нет, то ссылка **null**. Таким образом, перед продолжением в коде необходимо предусмотреть проверку на предмет **null**:

```
static void Main(string[] args)
{
    ...
    // Может ли hex2 интерпретироваться как IPointy?
    Hexagon hex2 = new Hexagon("Peter");
    IPointy itfPt2 = hex2 as IPointy;
    if(itfPt2 != null)
        Console.WriteLine("Points: {0}", itfPt2.Points);
    else
        Console.WriteLine("OOPS! Not pointy...");
    Console.ReadLine();
}
```

Обратите внимание, что в случае применения ключевого слова **as** отпадает необходимость в логике try/catch, т.к. возврат ссылки, отличной от **null**, означает, что вызов осуществляется с использованием действительной ссылки на интерфейс.

Получение ссылок на интерфейсы: ключевое слово is

Проверить, реализован ли нужный интерфейс, можно также с помощью ключевого слова **is** (которое тоже впервые упоминалось в главе 6). Если объект не совместим с указанным интерфейсом, возвращается значение **false**. С другой стороны, если тип совместим с интерфейсом, то можно безопасно обращаться к его членам без применения логики try/catch.

В целях иллюстрации предположим, что имеется массив типов **Shape**, часть элементов которого реализуют интерфейс **IPointy**. Ниже показано, как определить, какие из

элементов в этом массиве поддерживают данный интерфейс, с помощью ключевого слова `is` внутри модифицированной версии метода `Main()`:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Interfaces *****\n");
    // Создать массив элементов Shape.
    Shape[] myShapes = { new Hexagon(), new Circle(),
                        new Triangle("Joe"), new Circle("JoJo") } ;
    for(int i = 0; i < myShapes.Length; i++)
    {
        // Вспомните, что базовый класс Shape определяет абстрактный
        // член Draw(), поэтому все фигуры знают, как себя рисовать.

        myShapes[i].Draw();

        // У каких фигур есть вершины?
        if(myShapes[i] is IPointy)
            Console.WriteLine("-> Points: {0}", ((IPointy) myShapes[i]).Points);
        else
            Console.WriteLine("-> {0}\n's not pointy!", myShapes[i].PetName);
        Console.WriteLine();
    }
    Console.ReadLine();
}
```

Вывод выглядит следующим образом:

```
***** Fun with Interfaces *****

Drawing NoName the Hexagon
-> Points: 6

Drawing NoName the Circle
-> NoName's not pointy!

Drawing Joe the Triangle
-> Points: 3

Drawing JoJo the Circle
-> JoJo's not pointy!
```

Использование интерфейсов в качестве параметров

Благодаря тому, что интерфейсы — это допустимые типы .NET, можно строить методы, принимающие интерфейсы в качестве параметров, как это было с методом `CloneMe()`, показанным ранее в главе. Для целей текущего примера предположим, что определен еще один интерфейс по имени `IDraw3D`:

```
// Моделирует способность визуализировать тип в трехмерном виде.
public interface IDraw3D
{
    void Draw3D();
}
```

Далее сконфигурируем две из трех наших фигур (`Circle` и `Hexagon`) для поддержки этого нового поведения:

```
// Circle поддерживает IDraw3D.
class ThreeDCircle : Circle, IDraw3D
{

```

```

...
public void Draw3D()
{ Console.WriteLine("Drawing Circle in 3D!"); }
}

// Hexagon поддерживает IPointy и IDraw3D.
class Hexagon : Shape, IPointy, IDraw3D
{
    ...
    public void Draw3D()
    { Console.WriteLine("Drawing Hexagon in 3D!"); }
}

```

На рис. 8.3 показана обновленная диаграмма классов Visual Studio.

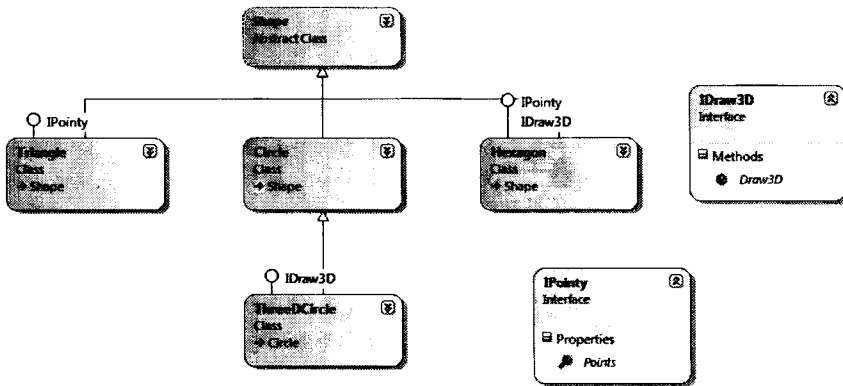


Рис. 8.3. Обновленная иерархия фигур

Если теперь определить метод, принимающий интерфейс IDraw3D в качестве параметра, то ему можно будет передавать, в сущности, любой объект, реализующий IDraw3D. (При попытке передать тип, не поддерживающий необходимый интерфейс, компилятор сообщит об ошибке.) Взгляните на следующий метод, определенный в классе Program:

```

// Будет рисовать любую фигуру, поддерживающую IDraw3D.
static void DrawIn3D(IDraw3D itf3d)
{
    Console.WriteLine("-> Drawing IDraw3D compatible type");
    itf3d.Draw3D();
}

```

Теперь можно проверить, поддерживает ли элемент в массиве Shape новый интерфейс, и если да, то передать его методу DrawIn3D() на обработку:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Interfaces *****\n");
    Shape[] myShapes = { new Hexagon(), new Circle(),
                        new Triangle(), new Circle("JoJo") };
    for(int i = 0; i < myShapes.Length; i++)
    {
        ...
        // Можно ли нарисовать эту фигуру в трехмерном виде?
        if(myShapes[i] is IDraw3D)
            DrawIn3D((IDraw3D)myShapes[i]);
    }
}

```


Ниже представлен вывод из модифицированной версии приложения. Обратите внимание, что в трехмерном виде отображается только объект `Hexagon`, поскольку все остальные члены массива `Shape` не реализуют интерфейса `IDraw3D`.

```
***** Fun with Interfaces *****
Drawing NoName the Hexagon
-> Points: 6
-> Drawing IDraw3D compatible type
Drawing Hexagon in 3D!
Drawing NoName the Circle
-> NoName's not pointy!
Drawing Joe the Triangle
-> Points: 3
Drawing JoJo the Circle
-> JoJo's not pointy!
```

Использование интерфейсов в качестве возвращаемых значений

Интерфейсы могут также использоваться как возвращаемые значения методов. Для примера напишем метод, который получает массив объектов `Shape` и возвращает ссылку на первый элемент, поддерживающий `IPointy`:

```
// Этот метод возвращает из массива первый объект,
// который реализует интерфейс IPointy.
static IPointy FindFirstPointyShape(Shape[] shapes)
{
    foreach (Shape s in shapes)
    {
        if (s is IPointy)
            return s as IPointy;
    }
    return null;
}
```

Взаимодействовать с этим методом можно следующим образом:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Interfaces *****\n");
    // Создать массив элементов Shape.
    Shape[] myShapes = { new Hexagon(), new Circle(),
                        new Triangle("Joe"), new Circle("JoJo") };
    // Получить первый элемент, имеющий вершины.
    // Для безопасности не помешает проверить firstPointyItem на предмет null.
    IPointy firstPointyItem = FindFirstPointyShape(myShapes);
    Console.WriteLine("The item has {0} points", firstPointyItem.Points);
    ...
}
```

Массивы интерфейсных типов

Вспомните, что один и тот же интерфейс может быть реализован множеством типов, даже если они не находятся внутри одной и той же иерархии классов и не имеют общего родительского класса помимо `System.Object`. Это позволяет формировать очень

мощные программные конструкции. Например, предположим, что в текущем проекте созданы три новых класса, два из которых (Knife (нож) и Fork (вилка)) моделируют кухонные приборы, а третий (PitchFork (вилы)) — садовый инструмент (рис. 8.4).

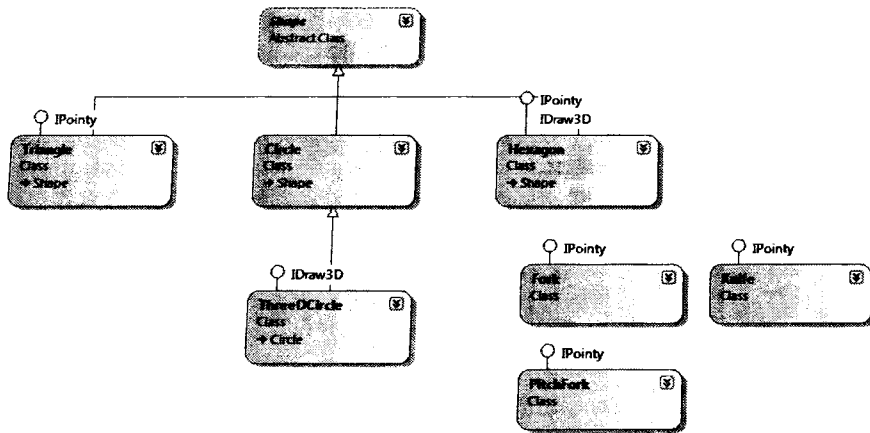


Рис. 8.4. Вспомните, что интерфейсы могут “подключаться” к любому типу внутри любой части иерархии классов

Имея определения типов PitchFork, Fork и Knife, можно определить массив объектов, совместимых с IPointy. Поскольку все эти члены поддерживают один и тот же интерфейс, можно выполнять проход по массиву и интерпретировать каждый его элемент как совместимый с IPointy объект, несмотря на разницу между иерархиями классов:

```

static void Main(string[] args)
{
    ...
    // Этот массив может содержать только типы,
    // которые реализуют интерфейс IPointy.
    IPointy[] myPointyObjects = {new Hexagon(), new Knife(),
        new Triangle(), new Fork(), new PitchFork()};
    foreach(IPointy i in myPointyObjects)
        Console.WriteLine("Object has {0} points.", i.Points);
    Console.ReadLine();
}

```

Просто чтобы подчеркнуть важность этого примера, запомните следующее: массив заданного интерфейса может содержать любой класс или структуру, которая реализует этот интерфейс.

Исходный код. Проект CustomInterface доступен в подкаталоге Chapter 08.

Реализация интерфейсов с использованием Visual Studio

Хотя программирование на основе интерфейсов является очень мощной технологией, реализация интерфейсов может сопровождаться довольно большим объемом кодирования. Так как интерфейсы представляют собой именованные наборы абстрактных членов, для каждого метода интерфейса в каждом типе, который поддерживает это поведение, потребуется вводить определение и реализацию. Следовательно, для под-

держки интерфейса, который определяет пять методов и три свойства, необходимо уделять внимание всем восьми членам (в противном случае возникнут ошибки на этапе компиляции).

К счастью, в Visual Studio поддерживаются различные инструменты, которые существенно упрощают задачу реализации интерфейсов. Для примера вставим в текущий проект еще один класс по имени `PointyTestClass`. При реализации для него интерфейса `IPointy` (или любого другого подходящего интерфейса) можно будет заметить, как по окончании ввода имени интерфейса (или при размещении на нем курсора мыши в окне кода) первая буква будет выделена подчеркиванием (или, согласно формальной терминологии, снабжена контекстной меткой — *смарт-тегом*). Щелчок на этом смарте приводит к появлению раскрывающегося списка, который позволяет реализовать этот интерфейс (рис. 8.5).

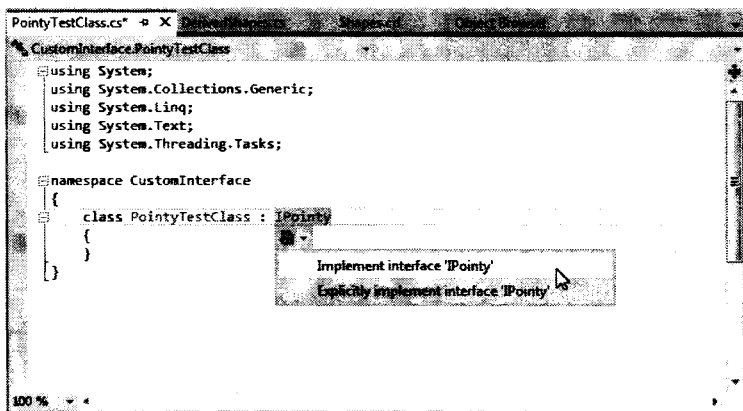


Рис. 8.5. Реализация интерфейсов в Visual Studio

Обратите внимание, что в этом списке предлагаются два варианта, из которых второй (явная реализация интерфейса) подробно рассматривается в следующем разделе. Пока что выберем первый вариант. В этом случае Visual Studio сгенерирует код заглушки, предназначенный для дальнейшего обновления (обратите внимание, что стандартная реализация генерирует исключение `System.NotImplementedException`, что очевидно подлежит удалению).

```
namespace CustomInterface
{
    class PointyTestClass : IPointy
    {
        public byte Points
        {
            get { throw new NotImplementedException(); }
        }
    }
}
```

На заметку! Среда Visual Studio также поддерживает рефакторинг типа выделения интерфейса (Extract Interface), доступный через меню Refactoring (Рефакторинг). Это позволяет извлекать определение нового интерфейса из существующего определения класса. Например, вы можете находиться на полпути к завершению написания класса, но вдруг вас осеняет, что данное поведение может быть обобщено в виде интерфейса (что, таким образом, откроет возможность для альтернативных реализаций).

Явная реализация интерфейсов

Как было показано ранее в главе, класс или структура может реализовать любое количество интерфейсов. С учетом этого, всегда существует возможность реализации интерфейсов с членами, имеющими идентичные имена, и, следовательно, возникает необходимость в устранении конфликтов имен. Чтобы ознакомиться с различными способами решения этой проблемы, создадим новое консольное приложение по имени `InterfaceNameClash` и добавим в него три специальных интерфейса, представляющих различные места, в которых реализующий их тип может визуализировать свой вывод:

```
// Вывести изображение на форме.
public interface IDrawToForm
{
    void Draw();
}

// Вывести изображение в буфер памяти.
public interface IDrawToMemory
{
    void Draw();
}

// Вывести изображение на принтер.
public interface IDrawToPrinter
{
    void Draw();
}
```

Обратите внимание, что каждый из этих интерфейсов определяет метод по имени `Draw()` с идентичной сигнатурой (без аргументов). Если теперь необходимо поддерживать все эти интерфейсы в одном классе `Octagon`, компилятор позволит использовать следующее определение:

```
class Octagon : IDrawToForm, IDrawToMemory, IDrawToPrinter
{
    public void Draw()
    {
        // Разделяемая логика вывода.
        Console.WriteLine("Drawing the Octagon...");
    }
}
```

Хотя компиляция этого кода пройдет гладко, одна возможная проблема все же присутствует. Выражаясь просто, предоставление единственной реализации метода `Draw()` не позволяет предпринимать уникальные действия на основе того, какой интерфейс получен от объекта `Octagon`. Например, следующий код будет приводить к вызову одного и того же метода `Draw()`, какой бы интерфейс не был получен:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Interface Name Clashes *****\n");
    // Все эти обращения приводят к вызову
    // одного и того же метода Draw()!
    Octagon oct = new Octagon();

    IDrawToForm itfForm = (IDrawToForm)oct;
    itfForm.Draw();

    IDrawToPrinter itfPriner = (IDrawToPrinter)oct;
    itfPriner.Draw();
}
```

```

IDrawToMemory itfMemory = (IDrawToMemory)oct;
itfMemory.Draw();

Console.ReadLine();
}

```

Очевидно, что код, требуемый для визуализации изображения в окне, довольно сильно отличается от того, который необходим для вывода изображения на сетевой принтер или в область памяти. При реализации нескольких интерфейсов, имеющих идентичные члены, разрешить такой конфликт имен можно с использованием синтаксиса *явной реализации интерфейсов*. Взгляните на следующее изменение типа Octagon:

```

class Octagon : IDrawToForm, IDrawToMemory, IDrawToPrinter
{
    // Явно привязать реализации Draw()
    // к конкретным интерфейсам.
    void IDrawToForm.Draw()
    {
        Console.WriteLine("Drawing to form...");
    }
    void IDrawToMemory.Draw()
    {
        Console.WriteLine("Drawing to memory...");
    }
    void IDrawToPrinter.Draw()
    {
        Console.WriteLine("Drawing to a printer...");
    }
}

```

Как видите, при явной реализации члена интерфейса основной шаблон сводится к следующему:

```
возвращаемыйТип ИмяИнтерфейса.ИмяМетода(параметры) {}
```

Обратите внимание, что при использовании этого синтаксиса не указывается модификатор доступа; явно реализованные члены автоматически являются закрытыми. Например, следующий синтаксис недопустим:

```

// Ошибка! Модификатор доступа не может быть указан!
public void IDrawToForm.Draw()
{
    Console.WriteLine("Drawing to form...");
}

```

Поскольку явно реализованные члены всегда неявно закрыты, они перестают быть доступными на уровне объектов. И действительно, если вы примените к типу Octagon операцию точки, то обнаружите, что средство IntelliSense не отображает никаких членов Draw(). Как и следовало ожидать, для доступа к требуемой функциональности должно использоваться явное приведение. Ниже представлен пример:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Interface Name Clashes *****\n");
    Octagon oct = new Octagon();

    // Теперь для доступа к членам Draw()
    // должно использоваться приведение.
    IDrawToForm itfForm = (IDrawToForm)oct;
    itfForm.Draw();

    // Сокращенная нотация, если переменная интерфейса не нужна.
    ((IDrawToPrinter)oct).Draw();
}

```

```
// Можно было бы также использовать ключевое слово as.
if (oct is IDrawToMemory)
    ((IDrawToMemory)oct).Draw();
Console.ReadLine();
}
```

Наряду с тем, что этот синтаксис полезен, когда необходимо устранить конфликты имен, явную реализацию интерфейсов можно применять и просто для сокрытия более “сложных” членов на уровне объектов. В таком случае при использовании операции точки пользователь объекта будет видеть только подмножество всей функциональности типа. Тем не менее, те, кому требуется более сложное поведение, могут извлекать желаемый интерфейс через явное приведение.

Исходный код. Проект `InterfaceNameClash` доступен в подкаталоге `Chapter 08`.

Проектирование иерархий интерфейсов

Интерфейсы могут быть организованы в иерархии. Как и в иерархии классов, когда интерфейс расширяет существующий интерфейс, он наследует все абстрактные члены своего родителя (или родителей). Конечно, в отличие от наследования на основе классов, производный интерфейс никогда не наследует настоящую реализацию. Вместо этого он просто расширяет собственное определение дополнительными абстрактными членами.

Иерархия интерфейсов может быть удобна, когда нужно расширить функциональность определенного интерфейса без нарушения работы существующих кодовых баз. В целях иллюстрации создадим новое консольное приложение по имени `InterfaceHierarchy`. Теперь спроектируем новый набор интерфейсов, связанных с визуализацией, так, чтобы `IDrawable` был корневым интерфейсом в дереве этого семейства:

```
public interface IDrawable
{
    void Draw();
}
```

Учитывая, что `IDrawable` определяет базовое поведение рисования, можно создать производный интерфейс, который расширяет `IDrawable` возможностью визуализации в других форматах, например:

```
public interface IAdvancedDraw : IDrawable
{
    void DrawInBoundingBox(int top, int left, int bottom, int right);
    void DrawUpsideDown();
}
```

При таком проектном решении для реализации интерфейса `IAdvancedDraw` в классе потребуется реализовать все члены, определенные в цепочке наследования (т.е. методы `Draw()`, `DrawInBoundingBox()` и `DrawUpsideDown()`):

```
public class BitmapImage : IAdvancedDraw
{
    public void Draw()
    {
        Console.WriteLine("Drawing...");
    }
}
```

```

public void DrawInBoundingBox(int top, int left, int bottom, int right)
{
    Console.WriteLine("Drawing in a box...");
}

public void DrawUpsideDown()
{
    Console.WriteLine("Drawing upside down!");
}
}

```

Теперь в случае использования `BitmapImage` можно вызывать каждый метод на уровне объекта (т.к. все они являются открытыми), а также извлекать ссылку на каждый поддерживаемый интерфейс явным образом с помощью приведения:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Simple Interface Hierarchy *****");

    // Вызвать на уровне объекта.
    BitmapImage myBitmap = new BitmapImage();
    myBitmap.Draw();
    myBitmap.DrawInBoundingBox(10, 10, 100, 150);
    myBitmap.DrawUpsideDown();

    // Получить IAdvancedDraw явным образом.
    IAdvancedDraw iAdvDraw = myBitmap as IAdvancedDraw;
    if (iAdvDraw != null)
        iAdvDraw.DrawUpsideDown();

    Console.ReadLine();
}

```

Исходный код. Проект `InterfaceHierarchy` доступен в подкаталоге `Chapter 08`.

Множественное наследование посредством интерфейсных типов

В отличие от классов, один интерфейс может расширять сразу несколько базовых интерфейсов, что позволяет проектировать очень мощные и гибкие абстракции. Создадим новый проект консольного приложения по имени `MIInterfaceHierarchy`. Затем построим еще одну коллекцию интерфейсов, которые моделируют различные абстракции, связанные с визуализацией и фигурами. Обратите внимание, что интерфейс `IShape` в этой коллекции расширяет и `IDrawable`, и `IPrintable`.

```

// Множественное наследование для интерфейсных типов разрешено.
interface IDrawable
{
    void Draw();
}

interface IPrintable
{
    void Print();
    void Draw(); // <-- Здесь возможен конфликт имен!
}

// Множественное наследование интерфейсов. Нормально!
interface IShape : IDrawable, IPrintable
{
    int GetNumberOfSides();
}

```

На рис. 8.6 показана текущая иерархия интерфейсов.

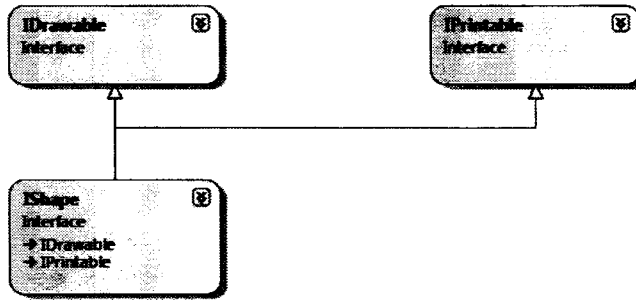


Рис. 8.6. В отличие от классов, интерфейсы могут расширять сразу несколько базовых интерфейсов

Теперь главный вопрос заключается в том, сколько методов должен реализовать класс, поддерживающий IShape? Ответ таков: в зависимости от обстоятельств. Если нужно предоставить простую реализацию метода Draw(), понадобится только реализовать три его члена, как показано в следующем классе Rectangle:

```

class Rectangle : IShape
{
    public int GetNumberOfSides()
    { return 4; }

    public void Draw()
    { Console.WriteLine("Drawing..."); }

    public void Print()
    { Console.WriteLine("Prining..."); }
}
  
```

Если вы предпочитаете иметь специфические реализации для каждого метода Draw() (что в данном случае имеет смысл), конфликт имен можно разрешить с применением явной реализации интерфейсов, как это сделано в представленном ниже классе Square:

```

class Square : IShape
{
    // Использование явной реализации для устранения конфликта имен членов.
    void IPrintable.Draw()
    {
        // Вывести на принтер...
    }
    void IDrawable.Draw()
    {
        // Вывести на экран...
    }
    public void Print()
    {
        // Печатать...
    }
    public int GetNumberOfSides()
    { return 4; }
}
  
```

К этому моменту процесс определения и реализации специальных интерфейсов на C# должен стать более понятным. По правде говоря, на привыкание к программированию на основе интерфейсов может уйти некоторое время.

Однако уже сейчас важно уяснить, что интерфейсы являются фундаментальным аспектом .NET Framework. Независимо от типа разрабатываемого приложения (веб-приложение, настольное приложение с графическим пользовательским интерфейсом, библиотека доступа к данным и т.п.), работа с интерфейсами будет составной частью этого процесса. Подводя итог всему изложенному, следует отметить, что интерфейсы могут быть исключительно полезны в таких ситуациях:

- имеется единственная иерархия, в которой только подмножество производных типов поддерживает общее поведение;
- необходимо моделировать общее поведение, которое должно встречаться в нескольких иерархиях, не имеющих общего родительского класса помимо System.Object.

Итак, вы ознакомились со спецификой построения и реализации специальных интерфейсов. Остаток этой главы посвящен исследованию ряда предопределенных интерфейсов, содержащихся в библиотеках базовых классов .NET.

Исходный код. Проект MIInterfaceHierarchy доступен в подкаталоге Chapter 08.

Интерфейсы IEnumerable и IEnumerator

Прежде чем приступить к исследованию процесса реализации существующих интерфейсов .NET, давайте сначала рассмотрим роль интерфейсов IEnumerable и IEnumerator. Вспомните, что в C# поддерживается ключевое слово foreach, которое позволяет осуществлять проход по содержимому массива любого типа:

```
// Итерация по массиву элементов.
int[] myArrayOfInts = {10, 20, 30, 40};

foreach(int i in myArrayOfInts)
{
    Console.WriteLine(i);
}
```

Хотя может показаться, что данная конструкция подходит только для массивов, на самом деле ее можно применять к любому типу, который поддерживает метод GetEnumerator(). В целях иллюстрации создадим новый проект консольного приложения по имени CustomEnumerator и добавим в него файлы Car.cs и Radio.cs, которые были определены в примере SimpleException из главы 7 (с помощью пункта меню Project⇒Add Existing Item).

На заметку! Во избежание импорта в новый проект пространства имен CustomException, имеет смысл переименовать пространство имен, содержащее типы Car и Radio, в CustomEnumerator.

Теперь вставим в проект новый класс Garage (гараж), который хранит набор объектов Car (автомобиль) внутри System.Array:

```
// Garage содержит набор объектов Car.
public class Garage
{
    private Car[] carArray = new Car[4];

    // Заполнить первоначально несколькими объектами Car.
    public Garage()
    {
        carArray[0] = new Car("Rusty", 30);
```

```

        carArray[1] = new Car("Clunker", 55);
        carArray[2] = new Car("Zippy", 30);
        carArray[3] = new Car("Fred", 30);
    }
}

```

В идеале было бы удобно проходить по элементам объекта `Garage`, используя конструкцию `foreach`, как в случае массива значений данных:

```

// Это выглядит корректным...
public class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with IEnumerable / IEnumerator *****\n");
        Garage carLot = new Garage();
        // Проход по всем объектам Car в коллекции?
        foreach (Car c in carLot)
        {
            Console.WriteLine("{0} is going {1} MPH",
                               c.PetName, c.CurrentSpeed);
        }
        Console.ReadLine();
    }
}

```

К сожалению, компилятор сообщит, что в классе `Garage` не реализован метод по имени `GetEnumerator()`. Этот метод формально определен в интерфейсе `IEnumerable`, который находится в пространстве имен `System.Collections`.

На заметку! В главе 9 вы узнаете о роли обобщений и о пространстве имен `System.Collections.Generic`. Как будет показано, это пространство имен содержит обобщенные версии `IEnumerable/IEnumerator`, которые предоставляют более безопасный к типам способ итерации по подобъектам.

Классы или структуры, которые поддерживают такое поведение, позиционируются как способные предоставлять содержащиеся внутри них элементы вызывающему коду (в рассматриваемом примере — самому ключевому слову `foreach`). Ниже приведено определение этого стандартного интерфейса `.NET`:

```

// Этот интерфейс информирует вызывающий код о том,
// что подэлементы объекта могут перечисляться.
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}

```

Как видите, метод `GetEnumerator()` возвращает ссылку на еще один интерфейс по имени `System.Collections.IEnumerator`. Этот интерфейс предоставляет инфраструктуру, которая позволяет вызывающему коду обходить внутренние объекты, содержащиеся в совместимом с `IEnumerable` контейнере:

```

// Этот интерфейс позволяет вызывающему коду получать подэлементы контейнера.
public interface IEnumerator
{
    bool MoveNext();           // Переместить вперед внутреннюю позицию курсора.
    object Current { get; }    // Получить текущий элемент (свойство,
                               // доступное только для чтения).
    void Reset();              // Поместить курсор перед первым элементом.
}

```

При модификации типа `Garage` для поддержки этих интерфейсов можно пойти длинным путем и реализовать каждый метод вручную. Хотя, конечно же, вы вольны предоставить специализированные версии методов `GetEnumerator()`, `MoveNext()`, `Current` и `Reset()`, существует более простой путь. Поскольку тип `System.Array` (а также многие другие классы коллекций) уже реализует интерфейсы `IEnumerable` и `IEnumerator`, можно просто делегировать запрос к `System.Array` следующим образом:

```
using System.Collections;
...
public class Garage : IEnumerable
{
    // System.Array уже реализует IEnumerator!
    private Car[] carArray = new Car[4];

    public Garage()
    {
        carArray[0] = new Car("FeeFee", 200);
        carArray[1] = new Car("Clunker", 90);
        carArray[2] = new Car("Zippy", 30);
        carArray[3] = new Car("Fred", 30);
    }

    public IEnumerator GetEnumerator()
    {
        // Возвратить IEnumerator объекта массива.
        return carArray.GetEnumerator();
    }
}
```

Изменив тип `Garage` подобным образом, можно безопасно использовать его внутри конструкции `foreach`. Более того, учитывая, что метод `GetEnumerator()` был определен как открытый, пользователь объекта может также взаимодействовать с `IEnumerator`:

```
// Работать напрямую с IEnumerator.
IEnumerator i = carLot.GetEnumerator();
i.MoveNext();
Car myCar = (Car)i.Current;
Console.WriteLine("{0} is going {1} MPH", myCar.PetName, myCar.CurrentSpeed);
```

Если нужно скрыть функциональность `IEnumerable` на уровне объекта, достаточно воспользоваться явной реализацией интерфейса:

```
IEnumerator IEnumerable.GetEnumerator()
{
    // Возвратить IEnumerator объекта массива.
    return carArray.GetEnumerator();
}
```

После этого обычный пользователь объекта не будет видеть метода `GetEnumerator()` в `Garage`, в то время как конструкция `foreach` будет получать интерфейс незаметным образом, когда это необходимо.

Исходный код. Проект `CustomEnumerator` доступен в подкаталоге `Chapter 08`.

Построение методов итератора с применением ключевого слова `yield`

В ранних версиях платформы .NET, когда требовалось построить специальную коллекцию (такую как `Garage`), поддерживающую перечисление `foreach`, реализация

интерфейса `IEnumerable` (и возможно `IEnumerator`) была единственным доступным вариантом. Однако теперь имеется альтернативный способ построения типов, работающих с циклом `foreach`, который предусматривает использование *итераторов*.

Выражаясь кратко, *итератор* — это член, который указывает, как должны возвращаться внутренние элементы контейнера при обработке в цикле `foreach`. Хотя метод итератора по-прежнему должен именоваться `GetEnumerator()`, а его возвращаемое значение иметь тип `IEnumerator`, создаваемый специальный класс не нуждается в реализации каких-либо ожидаемых интерфейсов.

В целях иллюстрации создадим новый проект консольного приложения по имени `CustomEnumeratorWithYield` и вставим в него типы `Car`, `Radio` и `Garage` из предыдущего примера (снова при желании переименовав пространство имен для текущего проекта). Затем модифицируем тип `Garage` следующим образом:

```
public class Garage : IEnumerator
{
    private Car[] carArray = new Car[4];
    ...
    // Метод итератора.
    public IEnumerator GetEnumerator()
    {
        foreach (Car c in carArray)
        {
            yield return c;
        }
    }
}
```

Обратите внимание, что в этой реализации метода `GetEnumerator()` проход по подэлементам осуществляется с использованием внутренней логики `foreach`, а каждый объект `Car` возвращается вызывающему коду с применением синтаксиса `yield return`. Ключевое слово `yield` служит для указания значения (или значений), которое должно возвращаться конструкции `foreach` в вызывающем коде. При достижении оператора `yield return` производится сохранение текущего местоположения в контейнере, а при следующем вызове итератора выполнение начинается с этого местоположения.

Методы итераторов не обязаны использовать ключевое слово `foreach` для возврата своего содержимого. Допускается также определить метод итератора следующим образом:

```
public IEnumerator GetEnumerator()
{
    yield return carArray[0];
    yield return carArray[1];
    yield return carArray[2];
    yield return carArray[3];
}
```

В этой реализации обратите внимание на то, что метод `GetEnumerator()` явно возвращает вызывающему коду новое значение после каждого прохода. В данном примере применение такого подхода не имеет особого смысла, поскольку при добавлении дополнительных объектов к переменной-члену `carArray` метод `GetEnumerator()` окажется нескоординированным. Тем не менее, такой синтаксис может быть полезен, когда необходимо возвращать из метода локальные данные для последующей обработки с помощью `foreach`.

Построение именованного итератора

Также интересно отметить, что ключевое слово `yield` формально может использоваться внутри любого метода, независимо от его имени. Такие методы (которые называются *именованными итераторами*) уникальны в том, что могут принимать любое количество аргументов. При построении именованного итератора очень важно понимать, что метод будет возвращать интерфейс `IEnumerable`, а не ожидаемый совместимый с `IEnumerator` тип. В целях иллюстрации добавим к типу `Garage` следующий метод:

```
public IEnumerable GetTheCars(bool ReturnReversed)
{
    // Возвратить элементы в обратном порядке.
    if (ReturnReversed)
    {
        for (int i = carArray.Length; i != 0; i--)
        {
            yield return carArray[i-1];
        }
    }
    else
    {
        // Возвратить элементы в том порядке, в каком они размещены в массиве.
        foreach (Car c in carArray)
        {
            yield return c;
        }
    }
}
```

Обратите внимание, что этот новый метод позволяет вызывающему коду получать подэлементы как в прямом, так и в обратном порядке, если во входном параметре передается значение `true`. Теперь с ним можно взаимодействовать следующим образом:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with the Yield Keyword *****\n");
    Garage carLot = new Garage();

    // Получить элементы, используя GetEnumerator().
    foreach (Car c in carLot)
    {
        Console.WriteLine("{0} is going {1} MPH",
            c.PetName, c.CurrentSpeed);
    }

    Console.WriteLine();

    // Получить элементы (в обратном порядке!), используя именованный итератор.
    foreach (Car c in carLot.GetTheCars(true))
    {
        Console.WriteLine("{0} is going {1} MPH",
            c.PetName, c.CurrentSpeed);
    }

    Console.ReadLine();
}
```

Как видите, именованные итераторы представляют собой полезные конструкции, т.к. позволяют определять в единственном специальном контейнере несколько способов для запрашивания возвращаемого набора.

Итак, в завершение темы построения перечислимых объектов запомните: для того, чтобы специальные типы могли работать с ключевым словом `foreach` языка C#, контейнер должен определять метод по имени `GetEnumerator()`, который формально определен интерфейсным типом `IEnumerable`. Реализация этого метода обычно осуществляется путем делегирования внутреннему члену, который хранит подобъекты; однако можно также использовать синтаксис `yield return`, чтобы предоставить множество методов “именованных итераторов”.

Исходный код. Проект `CustomEnumeratorWithYield` доступен в подкаталоге `Chapter 08`.

Интерфейс `ICloneable`

Как уже рассказывалось в главе 6, в `System.Object` определен метод по имени `MemberwiseClone()`. Этот метод используется для получения *поверхностной (неглубокой) копии* текущего объекта. Пользователи объекта не могут вызывать этот метод напрямую, т.к. он является защищенным. Однако сам объект может это делать во время процесса *клонирования*. Для примера создадим новое консольное приложение по имени `CloneablePoint`, в котором определен класс `Point`, представляющий точку:

```
// Класс по имени Point.
public class Point
{
    public int X {get; set;}
    public int Y {get; set;}

    public Point(int xPos, int yPos) { X = xPos; Y = yPos;}
    public Point(){}

    // Переопределить Object.ToString().
    public override string ToString()
    { return string.Format("X = {0}; Y = {1}", X, Y ); }
}
```

Как уже должно быть известно из материала о ссылочных типах и типах значений (см. главу 4), в случае присваивания одной переменной ссылочного типа другой получаются две ссылки, указывающие на один и тот же объект в памяти. Следовательно, показанное ниже присваивание даст в результате две ссылки на один и тот же объект `Point` в куче; модификация с использованием любой из этих ссылок оказывает воздействие на тот же самый объект в куче:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Object Cloning *****\n");

    // Две ссылки на один и тот же объект!
    Point p1 = new Point(50, 50);
    Point p2 = p1;
    p2.X = 0;
    Console.WriteLine(p1);
    Console.WriteLine(p2);
    Console.ReadLine();
}
```

Чтобы снабдить специальный тип способностью возвращать вызывающему коду идентичную копию самого себя, можно реализовать стандартный интерфейс `ICloneable`. Как было показано в начале этой главы, `ICloneable` определяет единственный метод по имени `Clone()`:

```
public interface ICloneable
{
    object Clone();
}
```

Очевидно, что реализация метода `Clone()` в разных классах варьируется. Тем не менее, базовая функциональность обычно остается неизменной: копирование значений переменных-членов в новый объект того же типа и возврат его пользователю. В целях иллюстрации внесем следующие изменения в класс `Point`:

```
// Теперь Point поддерживает возможность клонирования.
public class Point : ICloneable
{
    public int X { get; set; }
    public int Y { get; set; }

    public Point(int xPos, int yPos) { X = xPos; Y = yPos; }
    public Point() { }

    // Переопределить Object.ToString().
    public override string ToString()
    { return string.Format("X = {0}; Y = {1}", X, Y); }

    // Возвратить копию текущего объекта.
    public object Clone()
    { return new Point(this.X, this.Y); }
}
```

Теперь можно создавать точные автономные копии типа `Point`, как показано ниже:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Object Cloning *****\n");
    // Обратите внимание, что Clone() возвращает простой тип object.
    // Для получения нужно производного типа требуется явное приведение.
    Point p3 = new Point(100, 100);
    Point p4 = (Point)p3.Clone();

    // Изменить p4.X (что не приводит к изменению p3.x).
    p4.X = 0;

    // Вывести все объекты.
    Console.WriteLine(p3);
    Console.WriteLine(p4);
    Console.ReadLine();
}
```

Хотя текущая реализация `Point` удовлетворяет всем требованиям, ее все равно можно немного улучшить. Поскольку `Point` не содержит никаких внутренних переменных ссылочного типа, реализацию метода `Clone()` можно упростить:

```
public object Clone()
{
    // Копировать каждое поле Point почленно.
    return this.MemberwiseClone();
}
```

Однако следует иметь в виду, что если бы в `Point` содержались внутренние переменные ссылочного типа, метод `MemberwiseClone()` копировал бы ссылки на эти объекты (т.е. создавал *поверхностную копию*). Для поддержки *глубокой (детальной) копии* во время процесса клонирования потребуется создать новый экземпляр каждой переменной ссылочного типа. Давайте рассмотрим пример.

Более сложный пример клонирования

Теперь предположим, что класс `Point` содержит переменную-член ссылочного типа `PointDescription`. Этот класс поддерживает дружественное имя точки, а также ее идентификационный номер, выраженный как `System.Guid` (глобально уникальный идентификатор (globally unique identifier — GUID) — это статистически уникальное 128-битное число). Соответствующая реализация представлена ниже:

```
// Этот класс описывает точку.
public class PointDescription
{
    public string PetName {get; set;}
    public Guid PointID {get; set;}

    public PointDescription()
    {
        PetName = "No-name";
        PointID = Guid.NewGuid();
    }
}
```

Начальные изменения самого класса `Point` включают модификацию метода `ToString()` для учета новых данных состояния, а также определение и создание ссылочного типа `PointDescription`. Чтобы позволить внешнему миру устанавливать дружественное имя для `Point`, необходимо также изменить аргументы, передаваемые перегруженному конструктору:

```
public class Point : ICloneable
{
    public int X { get; set; }
    public int Y { get; set; }
    public PointDescription desc = new PointDescription();

    public Point(int xPos, int yPos, string petName)
    {
        X = xPos; Y = yPos;
        desc.PetName = petName;
    }
    public Point(int xPos, int yPos)
    {
        X = xPos; Y = yPos;
    }
    public Point() { }

    // Переопределить Object.ToString().
    public override string ToString()
    {
        return string.Format("X = {0}; Y = {1}; Name = {2};\nID = {3}\n",
            X, Y, desc.PetName, desc.PointID);
    }

    // Возвратить копию текущего объекта.
    public object Clone()
    { return this.MemberwiseClone(); }
}
```

Обратите внимание, что метод `Clone()` пока еще не был модифицирован. Следовательно, когда пользователь объекта запросит клонирование с применением текущей реализации, будет создана поверхностная (почленная) копия. В целях иллюстрации изменим метод `Main()`, как показано ниже:


```

static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Object Cloning *****\n");
    Console.WriteLine("Cloned p3 and stored new Point in p4");
    Point p3 = new Point(100, 100, "Jane");
    Point p4 = (Point)p3.Clone();

    Console.WriteLine("Before modification:");
    Console.WriteLine("p3: {0}", p3);
    Console.WriteLine("p4: {0}", p4);
    p4.desc.PetName = "My new Point";
    p4.X = 9;

    Console.WriteLine("\nChanged p4.desc.petName and p4.X");
    Console.WriteLine("After modification:");
    Console.WriteLine("p3: {0}", p3);
    Console.WriteLine("p4: {0}", p4);
    Console.ReadLine();
}

```

В приведенном ниже выводе следует отметить, что хотя типы значений на самом деле изменились, внутренние ссылочные типы поддерживают те же самые значения, т.к. они “указывают” на одинаковые объекты в памяти (в частности, дружественным именем у обоих объектов является `My new Point`).

```

***** Fun with Object Cloning *****

Cloned p3 and stored new Point in p4
Before modification:
p3: X = 100; Y = 100; Name = Jane;
ID = 133d66a7-0837-4bd7-95c6-b22ab0434509

p4: X = 100; Y = 100; Name = Jane;
ID = 133d66a7-0837-4bd7-95c6-b22ab0434509

Changed p4.desc.petName and p4.X
After modification:
p3: X = 100; Y = 100; Name = My new Point;
ID = 133d66a7-0837-4bd7-95c6-b22ab0434509

p4: X = 9; Y = 100; Name = My new Point;
ID = 133d66a7-0837-4bd7-95c6-b22ab0434509

```

Чтобы заставить метод `Clone()` создавать полную детальную копию внутренних ссылочных типов, необходимо сконфигурировать объект, возвращаемый методом `MemberwiseClone()` для учета имени текущего объекта `Point` (тип `System.Guid` на самом деле представляет собой структуру, поэтому числовые данные будут действительно копироваться). Ниже показана одна из возможных реализаций:

```

// Теперь необходимо настроить код для учета члена PointDescription.
public object Clone()
{
    // Сначала получить поверхностную копию.
    Point newPoint = (Point)this.MemberwiseClone();

    // Теперь заполнить пробелы.
    PointDescription currentDesc = new PointDescription();
    currentDesc.PetName = this.desc.PetName;
    newPoint.desc = currentDesc;
    return newPoint;
}

```

Если теперь запустить приложение и посмотреть на его вывод (показанный ниже), то будет видно, что возвращаемый методом `Clone()` объект `Point` действительно копирует свои внутренние переменные-члены ссылочного типа (обратите внимание, что дружественные имена у `p3` и `p4` теперь уникальны):

```
***** Fun with Object Cloning *****

Cloned p3 and stored new Point in p4
Before modification:
p3: X = 100; Y = 100; Name = Jane;
ID = 51f64f25-4b0e-47ac-ba35-37d263496406

p4: X = 100; Y = 100; Name = Jane;
ID = 0d3776b3-b159-490d-b022-7f3f60788e8a

Changed p4.desc.petName and p4.X
After modification:
p3: X = 100; Y = 100; Name = Jane;
ID = 51f64f25-4b0e-47ac-ba35-37d263496406

p4: X = 9; Y = 100; Name = My new Point;
ID = 0d3776b3-b159-490d-b022-7f3f60788e8a
```

Итак, подведем итоги по процессу клонирования. При наличии класса или структуры, содержащей только типы значений, необходимо реализовать метод `Clone()` с использованием `MemberwiseClone()`. Однако если имеется специальный тип, поддерживающий ссылочные типы, для построения “глубокой копии” может потребоваться создание нового объекта, который учитывает каждую переменную-член ссылочного типа.

Исходный код. Проект `CloneablePoint` доступен в подкаталоге `Chapter 08`.

Интерфейс `Comparable`

Интерфейс `System.Comparable` описывает поведение, которое позволяет сортировать объект на основе указанного ключа. Вот его формальное определение:

```
// Этот интерфейс позволяет объекту указать
// его отношение с другими подобными объектами.
public interface Comparable
{
    int CompareTo(object o);
}
```

На заметку! Обобщенная версия этого интерфейса (`Comparable<T>`) предоставляет более безопасный к типам способ обработки сравнений объектов. Обобщения будут более подробно рассматриваться в главе 9.

Для примера создадим новое консольное приложение по имени `ComparableCar`, в котором определен следующий обновленный класс `Car` (обратите внимание, что здесь просто добавлено новое свойство для представления уникального идентификатора каждого автомобиля и модифицированный конструктор):

```
public class Car
{
    ...
    public int CarID {get; set;}
    public Car(string name, int currSp, int id)
    {
```

```

    CurrentSpeed = currSp;
    PetName = name;
    CarID = id;
}
...
}

```

Теперь предположим, что существует следующий массив объектов Car:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Object Sorting *****\n");
    // Создать массив объектов Car.
    Car[] myAutos = new Car[5];
    myAutos[0] = new Car("Rusty", 80, 1);
    myAutos[1] = new Car("Mary", 40, 234);
    myAutos[2] = new Car("Viper", 40, 34);
    myAutos[3] = new Car("Mel", 40, 4);
    myAutos[4] = new Car("Chucky", 40, 5);
    Console.ReadLine();
}

```

Класс `System.Array` определяет статический метод по имени `Sort()`. При вызове этого метода на массиве внутренних типов (`int`, `short`, `string` и т.д.) имеется возможность сортировать элементы массива в числовом или алфавитном порядке, поскольку эти внутренние типы данных реализуют `Comparable`. Однако что произойдет в случае передачи методу `Sort()` массива объектов `Car`, как показано ниже?

```

// Отсортируются ли объекты Car? Пока еще нет!
Array.Sort(myAutos);

```

Запуск этого тестового кода приведет к исключению времени выполнения, потому что класс `Car` не поддерживает необходимый интерфейс. При построении специальных типов для обеспечения возможности сортировки массивов, которые содержат элементы этих типов, можно реализовать интерфейс `Comparable`. Реализуя детали `CompareTo()`, вы должны самостоятельно принять решение о том, что должно браться за основу в операции упорядочивания. Для типа `Car` вполне логичным кандидатом является внутреннее свойство `CarID`:

```

// Итерация по объектам Car может быть упорядочена на основе CarID.
public class Car : Comparable
{
    ...
    // Реализация Comparable.
    int Comparable.CompareTo(object obj)
    {
        Car temp = obj as Car;
        if (temp != null)
        {
            if (this.CarID > temp.CarID)
                return 1;
            if (this.CarID < temp.CarID)
                return -1;
            else
                return 0;
        }
        else
            // Параметр не является объектом типа Car!
            throw new ArgumentException("Parameter is not a Car!");
    }
}

```

Как видите, логика `CompareTo()` предусматривает сравнение входного объекта с текущим экземпляром на основе конкретного элемента данных. Возвращаемое значение `CompareTo()` используется для выяснения того, является текущий объект меньше, больше или равным объекту, с которым он сравнивается (табл. 8.1).

Таблица 8.1. Возвращаемые значения `CompareTo()`

Возвращаемое значение <code>CompareTo()</code>	Описание
Любое число меньше нуля	Этот экземпляр находится перед указанным объектом в порядке сортировки
Ноль	Этот экземпляр равен указанному объекту
Любое число больше нуля	Этот экземпляр находится после указанного объекта в порядке сортировки

Предыдущую реализацию `CompareTo()` можно усовершенствовать с учетом того факта, что в C# тип данных `int` (который представляет собой сокращенное обозначение типа `System.Int32` в CLR) реализует интерфейс `IComparable`. Реализовать `CompareTo()` в `Car` можно следующим образом:

```
int IComparable.CompareTo(object obj)
{
    Car temp = obj as Car;
    if (temp != null)
        return this.CarID.CompareTo(temp.CarID);
    else
        // Параметр не является объектом типа Car!
        throw new ArgumentException("Parameter is not a Car!");
}
```

В любом случае, чтобы тип `Car` понимал, каким образом сравнивать себя с подобными объектами, можно написать следующий код:

```
// Использование интерфейса IComparable.
static void Main(string[] args)
{
    // Создать массив объектов Car.
    ...
    // Отобразить содержимое текущего массива.
    Console.WriteLine("Here is the unordered set of cars:");
    foreach(Car c in myAutos)
        Console.WriteLine("{0} {1}", c.CarID, c.PetName);

    // Теперь отсортировать массив, используя IComparable!
    Array.Sort(myAutos);
    Console.WriteLine();

    // Отобразить содержимое отсортированного массива.
    Console.WriteLine("Here is the ordered set of cars:");
    foreach(Car c in myAutos)
        Console.WriteLine("{0} {1}", c.CarID, c.PetName);
    Console.ReadLine();
}
```

Ниже показан вывод, полученный в результате выполнения приведенного выше метода `Main()`:

```
***** Fun with Object Sorting *****
```

```
Here is the unordered set of cars:
```

```
1 Rusty
234 Mary
34 Viper
4 Mel
5 Chucky
```

```
Here is the ordered set of cars:
```

```
1 Rusty
4 Mel
5 Chucky
34 Viper
234 Mary
```

Указание нескольких порядков сортировки посредством IComparer

В данной версии класса Car в качестве основы для порядка сортировки используется идентификатор автомобиля (carID). В другом проектном решении для этого могло бы применяться дружественное имя автомобиля (для перечисления автомобилей в алфавитном порядке). А что если требуется построить класс Car, который поддерживал бы сортировку по идентификатору *и также* по дружественному имени? Для этого должен использоваться другой стандартный интерфейс по имени IComparer, который определен в пространстве имен System.Collections следующим образом:

```
// Общий способ для сравнения двух объектов.
interface IComparer
{
    int Compare(object o1, object o2);
}
```

На заметку! Обобщенная версия этого интерфейса (IComparable<T>) предоставляет более безопасный к типам способ обработки сравнений объектов. Обобщения будут более подробно рассматриваться в главе 9.

В отличие от IComparable, интерфейс IComparer обычно не реализуется в типе, подлежащем сортировке (т.е. Car). Вместо этого данный интерфейс реализуется в любом количестве вспомогательных классов, по одному для каждого порядка сортировки (по дружественному имени, идентификатору автомобиля и т.д.). В настоящий момент тип Car уже знает, как сравнивать автомобили друг с другом по внутреннему идентификатору. Следовательно, чтобы позволить пользователю объекта сортировать массив объектов Car по дружественному имени, потребуется создать дополнительный вспомогательный класс, реализующий IComparer. Ниже приведен весь необходимый код (не забудьте импортировать пространство имен System.Collections в файле кода):

```
// Этот вспомогательный класс используется для сортировки
// массива объектов Car по дружественному имени.
public class PetNameComparer : IComparer
{
    // Проверить дружественное имя каждого объекта.
    int IComparer.Compare(object o1, object o2)
    {
        Car t1 = o1 as Car;
        Car t2 = o2 as Car;
        if (t1 != null && t2 != null)
            return String.Compare(t1.PetName, t2.PetName);
    }
}
```

```

    else
        throw new ArgumentException("Parameter is not a Car!");
    }
}

```

Теперь этот вспомогательный класс можно использовать в коде. В `System.Array` имеется несколько перегруженных версий метода `Sort()`, одна из которых принимает объект, реализующий интерфейс `IComparer`.

```

static void Main(string[] args)
{
    ...
    // Теперь сортировать по дружественному имени.
    Array.Sort(myAutos, new PetNameComparer());

    // Вывести отсортированный массив.
    Console.WriteLine("Ordering by pet name:");
    foreach(Car c in myAutos)
        Console.WriteLine("{0} {1}", c.CarID, c.PetName);
    ...
}

```

Специальные свойства и специальные типы сортировки

Важно отметить, что можно использовать специальное статическое свойство для оказания пользователю объекта помощи с сортировкой типов `Car` по специфичному элементу данных. Предположим, что в класс `Car` добавлено статическое свойство, доступное только для чтения, по имени `SortByPetName`, которое возвращает экземпляр объекта, реализующего интерфейс `IComparer` (в данном случае — `PetNameComparer`):

```

// Теперь поддерживается специальное свойство
// для возврата корректного интерфейса IComparer.

public class Car : IComparable
{
    ...
    // Свойство, возвращающее PetNameComparer.
    public static IComparer SortByPetName
    { get { return (IComparer)new PetNameComparer(); } }
}

```

В коде пользователя объекта теперь можно сортировать по дружественному имени с использованием жестко ассоциированного свойства, а не автономного класса `PetNameComparer`:

```

// Сортировка по дружественному имени теперь немного проще.
Array.Sort(myAutos, Car.SortByPetName);

```

Исходный код. Проект `ComparableCar` доступен в подкаталоге `Chapter 08`.

К этому моменту вы должны понимать не только способы определения и реализации собственных интерфейсов, но и то, какую пользу они могут приносить. Следует отметить, что интерфейсы встречаются в каждом ключевом пространстве имен `.NET`, и в оставшейся части книги неоднократно придется иметь дело с разнообразными стандартными интерфейсами.

Резюме

Интерфейс может быть определен как именованная коллекция абстрактных членов. Поскольку интерфейс не предоставляет никаких деталей реализации, он часто рассматривается как поведение, которое может поддерживаться тем или иным типом. Когда несколько классов реализуют тот же самый интерфейс, каждый из них может трактоваться одинаковым образом (полиморфизм на основе интерфейсов), даже если эти классы находятся в разных иерархиях.

Для определения новых интерфейсов в C# предусмотрено ключевое слово `interface`. Как было показано в главе, тип может поддерживать столько интерфейсов, сколько необходимо, используя разделенный запятыми список. Более того, допускается создавать интерфейсы, порожденные от нескольких базовых интерфейсов.

В дополнение к возможности построения специальных интерфейсов, в библиотеках .NET определен набор стандартных (поставляемых вместе с платформой) интерфейсов. Вы узнали, что можно создавать специальные типы, которые реализуют эти предопределенные интерфейсы для обеспечения таких возможностей, как клонирование, сортировка и перечисление.

часть IV

Дополнительные конструкции программирования на C#

В этой части

Глава 9. Коллекции и обобщения

Глава 10. Делегаты, события и лямбда-выражения

Глава 11. Расширенные средства языка C#

Глава 12. LINQ to Objects

Глава 13. Время жизни объектов

Коллекции и обобщения

Любому приложению, создаваемому с помощью платформы .NET, потребуется решать проблемы поддержки и манипулирования набором значений данных в памяти. Эти значения данных могут поступать из множества местоположений, включая реляционную базу данных, локальный текстовый файл, XML-документ, вызов веб-службы и даже через предоставляемый пользователем источник ввода.

В первом выпуске платформы .NET программисты часто применяли классы из пространства имен `System.Collections` для хранения и взаимодействия с элементами данных, используемыми внутри приложения. В версии .NET 2.0 язык программирования C# был расширен для поддержки средства под названием *обобщения*; и вместе с этим изменением в библиотеках базовых классов появилось совершенно новое пространство имен: `System.Collections.Generic`.

В этой главе представлен обзор различных пространств имен и типов коллекций (обобщенных и необобщенных), находящихся в библиотеках базовых классов .NET. Как вы увидите, обобщенные контейнеры часто превосходят свои необобщенные аналоги, поскольку они обычно предоставляют лучшую безопасность к типам и преимущества в плане производительности. После объяснения того, как создавать и манипулировать обобщенными элементами внутри платформы, в оставшейся части главы будет показано, как создавать собственные методы и обобщенные типы. Вы узнаете о роли *ограничений* (и *соответствующего ключевого слова* `where` в C#), которые позволяют строить исключительно безопасные к типам классы.

Побудительные причины создания классов коллекций

Самым элементарным контейнером, который можно использовать для хранения данных приложения, является, несомненно, массив. Как было показано в главе 4, массивы C# позволяют определять наборы типизированных элементов (включая массив объектов типа `System.Object`, по сути представляющий собой массив любых типов) с фиксированным верхним пределом. Кроме того, вспомните из главы 4, что все переменные массивов C# имеют дело с функциональностью из класса `System.Array`. В качестве краткого напоминания, взгляните на следующий метод `Main()`, который создает массив текстовых данных и манипулирует его содержимым несколькими способами:

```
static void Main(string[] args)
{
    // Создать массив строковых данных.
    string[] strArray = {"First", "Second", "Third" };

    // Отобразить количество элементов в массиве с помощью свойства Length.
    Console.WriteLine("This array has {0} items.", strArray.Length);
}
```

```

Console.WriteLine();

// Отобразить содержимое массива с использованием перечислителя.
foreach (string s in strArray)
{
    Console.WriteLine("Array Entry: {0}", s);
}
Console.WriteLine();

// Обратить массив и снова вывести его содержимое.
Array.Reverse(strArray);
foreach (string s in strArray)
{
    Console.WriteLine("Array Entry: {0}", s);
}

Console.ReadLine();
}

```

Хотя базовые массивы могут быть удобны для управления небольшими объемами данных фиксированного размера, бывает также немало случаев, когда требуются более гибкие структуры данных, такие как динамически растущие и сокращающиеся контейнеры или контейнеры, которые хранят объекты, отвечающие только определенному критерию (например, объекты, унаследованные от заданного базового класса, или объекты, реализующие определенный интерфейс). При использовании простого массива всегда помните о том, что он имеет "фиксированный размер". Если вы создали массив из трех элементов, то вы и получите только три элемента; следовательно, приведенный ниже код даст в результате исключение времени выполнения (конкретнее — `IndexOutOfRangeException`):

```

static void Main(string[] args)
{
    // Создать массив строковых данных.
    string[] strArray = { "First", "Second", "Third" };

    // Попытаться добавить новый элемент после конца массива. Ошибка времени выполнения!
    strArray[3] = "new item?";

    ...
}

```

Чтобы помочь в преодолении ограничений простого массива, библиотеки базовых классов .NET поставляются с несколькими пространствами имен, содержащими *классы коллекций*. В отличие от простого массива C#, классы коллекций построены с возможностью динамического изменения своих размеров на лету при вставке либо удалении из них элементов. Более того, многие классы коллекций предлагают улучшенную безопасность к типам и оптимизированы для обработки содержащихся внутри данных эффективно с точки зрения расхода памяти. По мере чтения этой главы, вы быстро заметите, что класс коллекции может принадлежать к одной из двух обширных категорий:

- **необобщенные коллекции** (в основном находящиеся в пространстве имен `System.Collections`);
- **обобщенные коллекции** (в основном находящиеся в пространстве имен `System.Collections.Generic`).

Необобщенные коллекции обычно предназначены для оперирования над типами `System.Object` и, таким образом, являются слабо типизированными контейнерами (тем не менее, некоторые необобщенные коллекции работают только со специфическим типом данных, таким как объекты `string`). В противоположность этому, обобщенные коллекции являются намного более безопасными к типам, учитывая, что вы должны

указать “тип типа”, который они будут содержать после создания. Как вы увидите, признаком любого обобщенного элемента является наличие “параметра типа”, обозначаемого с помощью угловых скобок (например, `List<T>`). Детали обобщений (в том числе связанные с ними преимущества) будут рассматриваться позже в этой главе. А сейчас давайте ознакомимся с некоторыми ключевыми типами необобщенных коллекций из пространств имен `System.Collections` и `System.Collections.Specialized`.

Пространство имен `System.Collections`

С момента появления платформы .NET программисты часто использовали классы необобщенных коллекций из пространства имен `System.Collecitons`, которое содержит набор классов, предназначенных для управления и организации больших объемов данных в памяти. В табл. 9.1 документированы некоторые наиболее часто используемые классы коллекций, определенные в этом пространстве имен, а также основные интерфейсы, которые они реализуют.

На заметку! Любое приложение .NET, построенное с помощью .NET 2.0 или последующих версий, должно игнорировать классы в `System.Collections` и отдавать предпочтение соответствующим классам из `System.Collections.Generic`. Тем не менее, важно знать основы необобщенных классов коллекций, поскольку может возникнуть необходимость в сопровождении унаследованного программного обеспечения.

Таблица 9.1. Полезные типы из `System.Collections`

Класс <code>System.Collections</code>	Назначение	Основные реализуемые интерфейсы
<code>ArrayList</code>	Представляет коллекцию динамически изменяемого размера, содержащую объекты в определенном порядке	<code>ICollection</code> , <code>IEnumerable</code> и <code>ICloneable</code>
<code>BitArray</code>	Управляет компактным массивом битовых значений, которые представляются как булевские, где <code>true</code> обозначает установленный (1) бит, а <code>false</code> — неустановленный (0) бит	<code>ICollection</code> , <code>IEnumerable</code> и <code>ICloneable</code>
<code>Hashtable</code>	Представляет коллекцию пар “ключ/значение”, организованных на основе хеш-кода ключа	<code>IDictionary</code> , <code>ICollection</code> , <code>IEnumerable</code> и <code>ICloneable</code>
<code>Queue</code>	Представляет стандартную очередь объектов, работающую по алгоритму FIFO (“первый вошел — первый вышел”)	<code>ICollection</code> , <code>IEnumerable</code> и <code>ICloneable</code>
<code>SortedList</code>	Представляет коллекцию пар “ключ/значение”, отсортированных по ключу и доступных по ключу и по индексу	<code>IDictionary</code> , <code>ICollection</code> , <code>IEnumerable</code> и <code>ICloneable</code>
<code>Stack</code>	Представляет стек LIFO (“последний вошел — первый вышел”), поддерживающий функциональность заталкивания и выталкивания, а также считывания	<code>ICollection</code> , <code>IEnumerable</code> и <code>ICloneable</code>

Интерфейсы, реализованные этими классами коллекций, представляют огромное "окно" в их общую функциональность. В табл. 9.2 представлено описание общей природы этих основных интерфейсов, часть из которых поверхностно рассматривалась в главе 8.

Таблица 9.2. Основные интерфейсы, поддерживаемые классами System.Collections

Интерфейс System.Collections	Назначение
ICollection	Определяет общие характеристики (т.е. размер, перечисление и безопасность к потокам) всех необобщенных типов коллекций
ICloneable	Позволяет реализующему объекту возвращать копию самого себя вызывающему коду
IDictionary	Позволяет объекту необобщенной коллекции представлять свое содержимое в виде пар "имя/значение"
IEnumerable	Возвращает объект, реализующий интерфейс IEnumerator (см. следующую строку в этой таблице)
IEnumerator	Делает возможной итерацию в стиле foreach по элементам коллекции
ICollection	Обеспечивает поведение добавления, удаления и индексирования элементов в списке объектов

Иллюстративный пример: работа с ArrayList

Возможно, вы уже имеете первоначальный опыт использования (или реализации) некоторых из указанных выше классических структур данных, например, стеков, очередей или списков. Если это не так, то позже в главе, при рассмотрении обобщенных аналогов таких структур, будут предоставлены дополнительные сведения об отличиях между ними. А пока что взгляните на метод Main(), в котором используется объект ArrayList. Обратите внимание, что мы можем добавлять (и удалять) элементы на лету, а контейнер автоматически соответствующим образом изменяет свой размер:

```
// Для доступа к ArrayList потребуется импортировать System.Collections.
static void Main(string[] args)
{
    ArrayList strArray = new ArrayList();
    strArray.AddRange(new string[] { "First", "Second", "Third" });

    // Отобразить количество элементов в ArrayList.
    Console.WriteLine("This collection has {0} items.", strArray.Count);
    Console.WriteLine();

    // Добавить новый элемент и отобразить текущее их количество.
    strArray.Add("Fourth!");
    Console.WriteLine("This collection has {0} items.", strArray.Count);

    // Отобразить содержимое.
    foreach (string s in strArray)
    {
        Console.WriteLine("Entry: {0}", s);
    }
    Console.WriteLine();
}
```

Несложно догадаться, что класс ArrayList имеет множество полезных членов помимо свойства Count и методов AddRange() и Add(), которые подробно описаны в до-

кументации по .NET Framework. К слову, другие классы `System.Collections` (`Stack`, `Queue` и т.д.) также подробно документированы в справочной системе .NET.

Тем не менее, очень важно отметить, что в большинстве ваших проектов .NET, скорее всего, классы коллекций из пространства имен `System.Collections` использоваться не будут! В наши дни намного чаще применяются их обобщенные аналоги, расположенные в пространстве имен `System.Collections.Generic`. Учитывая это, остальные необобщенные классы из `System.Collections` здесь не обсуждаются (и примеры их использования не приводятся).

Обзор пространства имен `System.Collections.Specialized`

`System.Collections` — не единственное пространство имен .NET, которое содержит необобщенные классы коллекций. Например, в пространстве имен `System.Collections.Specialized` определено несколько специализированных типов коллекций. В табл. 9.3 описаны некоторые наиболее полезные типы в этом конкретном пространстве имен, причем все они необобщенные.

Таблица 9.3. Полезные классы `System.Collections.Specialized`

Тип <code>System.Collections.Specialized</code>	Назначение
<code>HybridDictionary</code>	Этот класс реализует интерфейс <code>IDictionary</code> за счет использования <code>ListDictionary</code> , когда коллекция маленькая, и затем переключается на <code>Hashtable</code> , когда коллекция становится большой
<code>ListDictionary</code>	Этот класс удобен, когда необходимо управлять небольшим количеством элементов (10 или около того), которые могут изменяться со временем. Для управления своими данными класс использует односвязный список
<code>StringCollection</code>	Этот класс обеспечивает оптимальный способ для управления крупными коллекциями строковых данных
<code>BitVector32</code>	Этот класс предоставляет простую структуру, которая хранит булевские значения и небольшие целые числа в 32 битах памяти

Помимо указанных конкретных типов это пространство имен также содержит множество дополнительных интерфейсов и абстрактных базовых классов, которые можно применять в качестве стартовых точек для создания специальных классов коллекций. Хотя эти “специализированные” типы и могут оказаться тем, что требуется для ваших проектов в ряде ситуаций, здесь они рассматриваться не будут. Опять-таки, во многих случаях вы, скорее всего, обнаружите, что пространство имен `System.Collections.Generic` предлагает классы с похожей функциональностью, но с набором преимуществ.

На заметку! В библиотеках базовых классов .NET доступны два дополнительных пространства имен, связанных с коллекциями (`System.Collections.ObjectModel` и `System.Collections.Concurrent`). Первое из них будет описано позже в этой главе, когда вы освоите тему обобщений. Пространство имен `System.Collections.Concurrent` предоставляет класс коллекций, безопасные к потокам (многопоточность рассматривается в главе 19).

Проблемы, связанные с необобщенными коллекциями

Хотя на протяжении многих лет с применением этих необобщенных классов коллекций (и интерфейсов) было построено немало успешных приложений .NET, опыт показал, что применение этих типов может быть сопряжено с множеством проблем.

Первая проблема состоит в том, что использование классов коллекций `System.Collections` и `System.Collections.Specialized` приводит к созданию низкопроизводительного кода, особенно в случае манипуляций с числовыми данными (т.е. типами значений). Как вскоре будет показано, при хранении таких данных в любом необобщенном классе коллекции, прототипированном для работы с `System.Object`, среде CLR приходится выполнять массу операций перемещения данных в памяти, что может значительно снизить скорость выполнения.

Вторая проблема связана с тем, что большинство необобщенных классов коллекций не являются безопасными к типам, т.к. они были созданы для оперирования на `System.Object` и потому могут содержать в себе все что угодно. Если разработчику .NET требовалось создать безопасную в отношении типов коллекцию (т.е. контейнер, который может содержать объекты, реализующие только определенный интерфейс), то единственным реальным вариантом было создание совершенно нового класса коллекции собственноручно. Это не слишком трудоемкая задача, но довольно утомительная.

Прежде чем будет показано, как использовать обобщения в своих программах, стоит глубже рассмотреть недостатки необобщенных классов коллекций; это поможет лучше понять проблемы, которые был призван решить механизм обобщений. Давайте создадим новое консольное приложение по имени `IssuesWithNongenericCollections` и затем импортируем пространство имен `System.Collections` в начале кода C#:

```
using System.Collections;
```

Проблема производительности

Как уже должно быть известно из главы 4, платформа .NET поддерживает две обширных категории данных: типы значений и ссылочные типы. Поскольку в .NET определены две основных категории типов, однажды может возникнуть необходимость представить переменную одной категории в виде переменной другой категории. Для этого в C# предлагается простой механизм, называемый *упаковкой* (boxing), который служит для сохранения данных типа значения в ссылочной переменной. Предположим, что в методе по имени `SimpleBoxUnboxOperation()` создана локальная переменная типа `int`. Если далее в приложении понадобится представить этот тип значения в виде ссылочно-го типа, значение следует *упаковать*, как показано ниже:

```
private static void SimpleBoxUnboxOperation()
{
    // Создать переменную ValueType (int).
    int myInt = 25;

    // Упаковать int в ссылку на object.
    object boxedInt = myInt;
}
```

Упаковку можно формально определить как процесс явного присваивания типа значения переменной `System.Object`. При упаковке значения среда CLR размещает в куче новый объект и копирует значение типа значения (в данном случае 25) в этот экземпляр. В качестве результата возвращается ссылка на вновь размещенный в куче объект.

Противоположная операция также разрешена, и она называется *распаковкой* (unboxing). Распаковка — это процесс преобразования значения, хранящегося в объектной ссылке, обратно в соответствующий тип значения в стеке. Синтаксически операция распаковки выглядит как нормальная операция приведения, однако ее семантика несколько отличается. Среда CLR начинает с проверки того, что полученный тип данных эквивалентен упакованному типу, и если это так, то копирует значение обратно в находящуюся в стеке переменную. Например, следующие операции распаковки работают успешно при условии, что типом `boxedInt` в действительности является `int`:

```
private static void SimpleBoxUnboxOperation()
{
    // Создать переменную ValueType (int).
    int myInt = 25;

    // Упаковать int в ссылку на object.
    object boxedInt = myInt;

    // Распаковать ссылку обратно в int.
    int unboxedInt = (int)boxedInt;
}
```

Когда компилятор C# встречает синтаксис упаковки/распаковки, он генерирует CIL-код, содержащий коды операций `box/unbox`. Заглянув в сборку с помощью утилиты `ildasm.exe`, можно найти там следующий CIL-код:

```
.method private hidebysig static void SimpleBoxUnboxOperation() cil managed
{
    // Code size 19 (0x13)
    .maxstack 1
    .locals init ([0] int32 myInt, [1] object boxedInt, [2] int32 unboxedInt)
    IL_0000: nop
    IL_0001: ldc.i4.s 25
    IL_0003: stloc.0
    IL_0004: ldloc.0
    IL_0005: box [mscorlib]System.Int32
    IL_000a: stloc.1
    IL_000b: ldloc.1
    IL_000c: unbox.any [mscorlib]System.Int32
    IL_0011: stloc.2
    IL_0012: ret
} // end of method Program::SimpleBoxUnboxOperation
```

Помните, что в отличие от обычного приведения распаковка *должна* производиться только в соответствующий тип данных. Попытка распаковать порцию данных в некорректный тип данных приводит к генерации исключения `InvalidCastException`. Для полной безопасности следовало бы поместить каждую операцию распаковки в конструкцию `try/catch`, однако делать это для абсолютно каждой операции распаковки в приложении может оказаться довольно трудоемкой задачей. Взгляните на следующий измененный код, который выдаст ошибку, поскольку предпринята попытка распаковать упакованный `int` в `long`:

```
static void SimpleBoxUnboxOperation()
{
    // Создать переменную ValueType (int).
    int myInt = 25;

    // Упаковать int в ссылку на object.
    object boxedInt = myInt;

    // Распаковать в неверный тип данных, чтобы
    // инициализировать исключение времени выполнения.
}
```

```

try
{
    long unboxedInt = (long)boxedInt;
}
catch (InvalidCastException ex)
{
    Console.WriteLine(ex.Message);
}
}

```

На первый взгляд упаковка/распаковка может показаться довольно несущественным средством языка, представляющим скорее академический интерес, нежели практическую ценность. На самом деле процесс упаковки/распаковки очень полезен, поскольку позволяет предположить, что все можно трактовать как `System.Object`, причем CLR берет на себя все заботы о деталях, связанных с памятью.

Давайте посмотрим на практическое применение этих приемов. Предположим, что создан необобщенный класс `System.Collections.ArrayList` для хранения множества числовых (расположенных в стеке) данных. Члены `ArrayList` прототипированы для работы с данными `System.Object`. Теперь рассмотрим методы `Add()`, `Insert()`, `Remove()`, а также индексатор класса:

```

public class ArrayList : object,
    IList, ICollection, IEnumerable, ICloneable
{
    ...
    public virtual int Add(object value);
    public virtual void Insert(int index, object value);
    public virtual void Remove(object obj);
    public virtual object this[int index] { get; set; }
}

```

Класс `ArrayList` ориентирован на работу с экземплярами `object`, которые представляют данные, расположенные в куче, поэтому может показаться странным, что следующий код компилируется и выполняется без ошибок:

```

static void WorkWithArrayList()
{
    // Типы значений упаковываются автоматически
    // при передаче методу, запросившему объект.
    ArrayList myInts = new ArrayList();
    myInts.Add(10);
    myInts.Add(20);
    myInts.Add(35);
}

```

Несмотря на непосредственную передачу числовых данных в методы, требующие тип `object`, исполняющая среда автоматически упаковывает их в данные, расположенные в стеке. При последующем извлечении элемента из `ArrayList` с использованием индексатора типа потребуются распаковать посредством операции приведения объект, находящийся в куче, в целочисленное значение, расположенное в стеке. Помните, что индексатор `ArrayList` возвращает `System.Object`, а не `System.Int32`:

```

static void WorkWithArrayList()
{
    // Типы значений автоматически упаковываются, когда
    // передаются члену, принимающему объект.
    ArrayList myInts = new ArrayList();
    myInts.Add(10);
    myInts.Add(20);
    myInts.Add(35);
}

```



```
// Распаковка происходит, когда объект преобразуется
// обратно в расположенные в стеке данные.
int i = (int)myInts[0];

// Теперь значение вновь упаковывается, т.к. WriteLine() требует объектные типы!
Console.WriteLine("Value of your int: {0}", i);
}
```

Обратите внимание, что расположенные в стеке значения `System.Int32` упаковываются перед вызовом `ArrayList.Add()`, чтобы их можно было передать в требуемом виде `System.Object`. Также отметьте, что объекты `System.Object` распаковываются обратно в `System.Int32` после их извлечения из `ArrayList` через операцию приведения только для того, чтобы вновь быть упакованными для передачи в метод `Console.WriteLine()`, поскольку этот метод оперирует переменными `System.Object`.

Хотя упаковка и распаковка очень удобны с точки зрения программиста, этот упрощенный подход к передаче данных между стеком и кучей влечет за собой проблемы, связанные с производительностью (это касается как скорости выполнения, так и размера кода), а также недостаток безопасности к типам. Чтобы понять, в чем состоят проблемы с производительностью, взгляните на перечень действий, которые должны быть выполнены при упаковке и распаковке простого целого числа.

1. Новый объект должен быть размещен в управляемой куче.
2. Значение данных, находящихся в стеке, должно быть передано в выделенное место в памяти.
3. При распаковке значение, которое хранится в объекте, находящемся в куче, должно быть передано обратно в стек.
4. Неиспользуемый больше объект в куче будет (в конечном итоге) удален сборщиком мусора.

Хотя существующий метод `Main()` не является основным узким местом в смысле производительности, вы определенно это почувствуете, если `ArrayList` будет содержать тысячи целочисленных значений, к которым программа обращается на регулярной основе. В идеальном случае хотелось бы манипулировать расположенными в стеке данными внутри контейнера, не имея проблем с производительностью. Было бы хорошо иметь возможность извлекать данные из контейнера, обходясь без конструкций `try/catch` (именно это обеспечивают обобщения).

Проблемы с безопасностью типов

Проблема безопасности типов уже затрагивалась, когда речь шла об операциях упаковки. Вспомните, что данные должны быть распакованы в тот же тип, который был для них объявлен перед упаковкой. Однако существует и другой аспект безопасности типов, который следует иметь в виду в мире без обобщений: тот факт, что классы из `System.Collections` могут хранить все что угодно, поскольку их члены прототипированы для работы с `System.Object`. Например, в следующем методе контейнер `ArrayList` хранит произвольные фрагменты несвязанных данных:

```
static void ArrayListOfRandomObjects()
{
    // ArrayList может хранить все что угодно.
    ArrayList allMyObject = new ArrayList();
    allMyObjects.Add(true);
    allMyObjects.Add(new OperatingSystem(PlatformID.MacOSX, new Version(10, 0)));
    allMyObjects.Add(66);
    allMyObjects.Add(3.14);
}
```

В некоторых случаях действительно необходим исключительно гибкий контейнер, который может хранить буквально все. Однако в большинстве ситуаций понадобится *безопасный в отношении типов* контейнер, который может оперировать только определенным типом данных, например, контейнер, который хранит только подключения к базе данных, битовыми образами или объекты, совместимые с `IPointy`.

До появления обобщений единственным способом решения этой проблемы было создание вручную класса строго типизированной коллекции. Предположим, что создана специальная коллекция, которая может содержать только объекты типа `Person`:

```
public class Person
{
    public int Age {get; set;}
    public string FirstName {get; set;}
    public string LastName {get; set;}

    public Person(){}
    public Person(string firstName, string lastName, int age)
    {
        Age = age;
        FirstName = firstName;
        LastName = lastName;
    }

    public override string ToString()
    {
        return string.Format("Name: {0} {1}, Age: {2}",
            FirstName, LastName, Age);
    }
}
```

Чтобы построить коллекцию, позволяющую хранить только объекты `Person`, можно определить переменную-член `System.Collection.ArrayList` внутри класса по имени `PeopleCollection` и сконфигурировать все члены для работы со строго типизированными объектами `Person` вместо объектов типа `System.Object`. Ниже приведен простой пример (реальная коллекция производственного уровня должна включать множество дополнительных членов и расширять абстрактный базовый класс из пространства имен `System.Collections` или `System.Collections.Specialized`):

```
public class PeopleCollection : IEnumerable
{
    private ArrayList arPeople = new ArrayList();

    // Приведение для вызывающего кода.
    public Person GetPerson(int pos)
    { return (Person)arPeople[pos]; }

    // Вставка только объектов Person.
    public void AddPerson(Person p)
    { arPeople.Add(p); }

    public void ClearPeople()
    { arPeople.Clear(); }

    public int Count
    { get { return arPeople.Count; } }

    // Поддержка перечисления с помощью foreach.
    IEnumerator IEnumerable.GetEnumerator()
    { return arPeople.GetEnumerator(); }
}
```

Обратите внимание, что класс `PeopleCollection` реализует интерфейс `IEnumerable`, который делает возможной итерацию в стиле `foreach` по всем содержащимся в коллекции элементам. Кроме того, методы `GetPerson()` и `AddPerson()` прототипированы на работу только с объектами `Person`, а не битовыми образами, строками, подключениями к базе данных или другими элементами. За счет создания таких классов обеспечивается безопасность типов, учитывая, что компилятор C# будет иметь возможность выявить любую попытку вставки элемента несовместимого типа:

```
static void UsePersonCollection()
{
    Console.WriteLine("***** Custom Person Collection *****\n");
    PersonCollection myPeople = new PersonCollection();
    myPeople.AddPerson(new Person("Homer", "Simpson", 40));
    myPeople.AddPerson(new Person("Marge", "Simpson", 38));
    myPeople.AddPerson(new Person("Lisa", "Simpson", 9));
    myPeople.AddPerson(new Person("Bart", "Simpson", 7));
    myPeople.AddPerson(new Person("Maggie", "Simpson", 2));

    // Это вызовет ошибку при компиляции!
    // myPeople.AddPerson(new Car());

    foreach (Person p in myPeople)
        Console.WriteLine(p);
}
```

Хотя подобные специальные коллекции гарантируют безопасность типов, такой подход все же обязывает создавать (в основном идентичные) специальные коллекции для каждого уникального типа данных, который планируется хранить. Таким образом, если нужна специальная коллекция, которая будет способна оперировать только классами, унаследованными от базового класса `Car`, понадобится построить очень похожий класс коллекции:

```
public class CarCollection : IEnumerable
{
    private ArrayList arCars = new ArrayList();

    // Приведение для вызывающего кода.
    public Car GetCar(int pos)
    { return (Car) arCars[pos]; }

    // Вставка только объектов Car.
    public void AddCar(Car c)
    { arCars.Add(c); }

    public void ClearCars()
    { arCars.Clear(); }

    public int Count
    { get { return arCars.Count; } }

    // Поддержка перечисления с помощью foreach.
    IEnumerator IEnumerable.GetEnumerator()
    { return arCars.GetEnumerator(); }
}
```

Однако эти специальные контейнеры мало помогают в решении проблем упаковки/распаковки. Даже если создать специальную коллекцию по имени `IntCollection`, предназначенную для работы только с элементами `System.Int32`, все равно придется выделить некоторый тип объекта для хранения данных (например, `System.Array` и `ArrayList`):

```

public class IntCollection : IEnumerable
{
    private ArrayList arInts = new ArrayList();

    // Получить int (выполнить распаковку).
    public int GetInt(int pos)
    { return (int)arInts[pos]; }

    // Вставить int (выполнить упаковку).
    public void AddInt(int i)
    { arInts.Add(i); }

    public void ClearInts()
    { arInts.Clear(); }

    public int Count
    { get { return arInts.Count; } }

    IEnumerator IEnumerable.GetEnumerator()
    { return arInts.GetEnumerator(); }
}

```

Независимо от того, какой тип выбран для хранения целых чисел, дилеммы упаковки нельзя избежать, применяя необобщенные контейнеры.

Первый взгляд на обобщенные коллекции

В случае использования классов обобщенных коллекций исчезают все описанные выше проблемы, включая затраты на упаковку/распаковку и недостаток безопасности типов. Кроме того, потребность в создании специального класса (обобщенной) коллекции становится довольно редкой. Вместо построения специальных коллекций, которые могут хранить людей, автомобили и целые числа, можно обратиться к обобщенному классу коллекции и указать тип хранимых элементов.

В показанном ниже методе класс `List<T>` (из пространства имен `System.Collection.Generic`) используется для хранения различных типов данных в строго типизированной манере (пока не обращайте внимания на детали синтаксиса обобщений):

```

static void UseGenericList()
{
    Console.WriteLine("***** Fun with Generics *****\n");

    // Этот List<> может хранить только объекты Person.
    List<Person> morePeople = new List<Person>();
    morePeople.Add(new Person ("Frank", "Black", 50));
    Console.WriteLine(morePeople[0]);

    // Этот List<> может хранить только целые числа.
    List<int> moreInts = new List<int>();
    moreInts.Add(10);
    moreInts.Add(2);
    int sum = moreInts[0] + moreInts[1];

    // Ошибка компиляции! Объект Person не может быть добавлен в список int!
    // moreInts.Add(new Person());
}

```

Первая коллекция `List<T>` может содержать только объекты `Person`. Поэтому выполнять приведение при извлечении элементов из контейнера не требуется, что делает этот подход более безопасным в отношении типов. Вторая коллекция `List<T>` может хранить только целые числа, и все они размещены в стеке; другими словами, здесь не происходит никакой скрытой упаковки/распаковки, как это имеет место в необобщенном `ArrayList`.

Ниже приведен краткий перечень преимуществ обобщенных контейнеров по сравнению с их необобщенными аналогами.

- Обобщения обеспечивают более высокую производительность, поскольку не страдают от проблем упаковки/распаковки при хранении типов значений.
- Обобщения являются безопасными в отношении типов, т.к. могут содержать только объекты указанного типа.
- Обобщения значительно сокращают потребность в специальных типах коллекций, потому что вы указываете “тип типа” при создании обобщенного контейнера.

Исходный код. Проект `IssuesWithNonGenericCollections` доступен в подкаталоге `Chapter 09`.

Роль параметров обобщенных типов

Обобщенные классы, интерфейсы, структуры и делегаты буквально разбросаны по всей базовой библиотеке классов .NET, и они могут быть частью любого пространства имен .NET. Кроме того, учтите, что использование обобщений далеко не ограничивается одним лишь определением класса коллекции. Разумеется, в оставшейся части книги вы увидите много других обобщений, применяемых для разных целей.

На заметку! Обобщенными могут быть только классы, структуры, интерфейсы и делегаты, но не перечисления.

Отличить обобщенный элемент в документации .NET Framework или браузере объектов Visual Studio от других элементов очень легко по наличию пары угловых скобок с буквой или другой лексемой. На рис. 9.1 показан браузер объектов Visual Studio, который отображает множество обобщенных элементов из пространства имен `System.Collections.Generic`, включая выделенный класс `List<T>`.

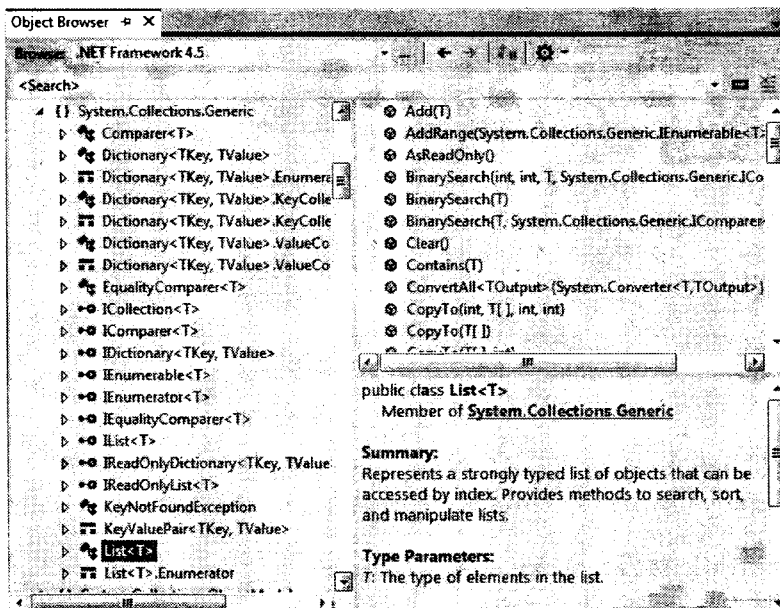


Рис. 9.1. Обобщенные элементы, поддерживающие параметры типа

Формально эти лексемы можно называть *параметрами типа*, однако в более дружественных к пользователю терминах их можно считать просто *заполнителями*. Конструкцию `<T>` можно воспринимать как *типа T*. Таким образом, `IEnumerable<T>` можно читать как *IEnumerable типа T*, или, говоря иначе, *перечисление типа T*.

На заметку! Имя параметра типа (заполнитель) не важно, и это — дело вкуса разработчика, создавшего обобщенный элемент. Тем не менее, обычно для представления типов используется `T`, для представления ключей — `TKey` или `K`, а для представления значений — `TValue` или `V`.

При создании обобщенного объекта, реализации обобщенного интерфейса или вызове обобщенного члена должно быть указано значение для параметра типа. Как в этой главе, так и в остальной части книги будет продемонстрировано немало примеров. Однако для начала следует ознакомиться с основами взаимодействия с обобщенными типами и членами.

Указание параметров типа для обобщенных классов и структур

При создании экземпляра обобщенного класса или структуры параметр типа указывается, когда объявляется переменная и когда вызывается конструктор. В предыдущем фрагменте кода было показано, что `UseGenericList()` определяет два объекта `List<T>`:

```
// Этот List<> может хранить только объекты Person.
List<Person> morePeople = new List<Person>();
```

Этот фрагмент можно трактовать как *List<> объектов T, где T — тип Person*, или более просто — *список объектов персон*. После указания параметра типа обобщенного элемента его нельзя изменить (помните: обобщения предназначены для поддержки безопасности типов). Когда параметр типа задается для обобщенного класса или структуры, все вхождения заполнителей заменяются указанным значением.

Просмотрев полное объявление обобщенного класса `List<T>` в браузере объектов Visual Studio, можно заметить, что заполнитель `T` используется в определении повсеместно. Ниже приведен частичный листинг (обратите внимание на элементы, выделенные полужирным):

```
// Частичный листинг класса List<T>.
namespace System.Collections.Generic
{
    public class List<T> :
        IList<T>, ICollection<T>, IEnumerable<T>,
        IList, ICollection, IEnumerable
    {
        ...
        public void Add(T item);
        public ReadOnlyCollection<T> AsReadOnly();
        public int BinarySearch(T item);
        public bool Contains(T item);
        public void CopyTo(T[] array);
        public int FindIndex(System.Predicate<T> match);
        public T FindLast(System.Predicate<T> match);
        public bool Remove(T item);
        public int RemoveAll(System.Predicate<T> match);
        public T[] ToArray();
        public bool TrueForAll(System.Predicate<T> match);
        public T this[int index] { get; set; }
    }
}
```

Когда создается `List<T>` с указанием объектов `Person`, это все равно, как если бы тип `List<T>` был определен следующим образом:

```
namespace System.Collections.Generic
{
    public class List<Person> :
        IList<Person>, ICollection<Person>, IEnumerable<Person>, IReadOnlyList<Person>
        IList, ICollection, IEnumerable
    {
        ...
        public void Add(Person item);
        public ReadOnlyCollection<Person> AsReadOnly();
        public int BinarySearch(Person item);
        public bool Contains(Person item);
        public void CopyTo(Person[] array);
        public int FindIndex(System.Predicate<Person> match);
        public Person FindLast(System.Predicate<Person> match);
        public bool Remove(Person item);
        public int RemoveAll(System.Predicate<Person> match);
        public Person[] ToArray();
        public bool TrueForAll(System.Predicate<Person> match);
        public Person this[int index] { get; set; }
    }
}
```

Разумеется, при создании в коде обобщенной переменной `List<T>` компилятор на самом деле не создает совершенно новую реализацию класса `List<T>`. Вместо этого он обрабатывает только члены обобщенного типа, к которым действительно производится обращение.

Указание параметров типа для обобщенных членов

Для необобщенного класса или структуры вполне допустимо поддерживать несколько обобщенных членов (например, методов и свойств). В таких случаях указывать значение заполнителя нужно также и во время вызова метода. Например, `System.Array` поддерживает несколько обобщенных методов. В частности, статический метод `Sort()` имеет обобщенный конструктор по имени `Sort<T>()`. Рассмотрим следующий фрагмент кода, в котором `T` — это тип `int`:

```
int[] myInts = { 10, 4, 2, 33, 93 };

// Указание заполнителя для обобщенного метода Sort<>().
Array.Sort<int>(myInts);
foreach (int i in myInts)
{
    Console.WriteLine(i);
}
```

Указание параметров типов для обобщенных интерфейсов

Обобщенные интерфейсы обычно реализуются при построении классов или структур, которые должны поддерживать различные поведения платформы (например, клонирование, сортировку и перечисление). В главе 8 рассматривалось множество необобщенных интерфейсов, таких как `IComparable`, `IEnumerable`, `IEnumerator` и `IComparer`. Вспомните, как определен необобщенный интерфейс `IComparable`:

```
public interface IComparable
{
    int CompareTo(object obj);
}
```

В той же главе 8 этот интерфейс был реализован в классе `Car` для обеспечения сортировки в стандартном массиве. Однако код требовал нескольких проверок времени выполнения и операций приведения, потому что параметром был общий тип `System.Object`:

```
public class Car : IComparable
{
    ...
    // Реализация IComparable.
    int IComparable.CompareTo(object obj)
    {
        Car temp = obj as Car;
        if (temp != null)
        {
            if (this.CarID > temp.CarID)
                return 1;
            if (this.CarID < temp.CarID)
                return -1;
            else
                return 0;
        }
        else
            throw new ArgumentException("Parameter is not a Car!");
    }
}
```

Теперь воспользуемся обобщенным аналогом этого интерфейса:

```
public interface IComparable<T>
{
    int CompareTo(T obj);
}
```

В таком случае код реализации будет значительно яснее:

```
public class Car : IComparable<Car>
{
    ...
    // Реализация IComparable<T>.
    int IComparable<Car>.CompareTo(Car obj)
    {
        if (this.CarID > obj.CarID)
            return 1;
        if (this.CarID < obj.CarID)
            return -1;
        else
            return 0;
    }
}
```

Здесь уже не нужно проверять, относится ли входной параметр к типу `Car`, потому что он может быть *только* `Car`! В случае передачи несовместимого типа данных возникает ошибка на этапе компиляции.

Итак, вы получили начальные сведения о том, как взаимодействовать с обобщенными элементами, а также ознакомились с ролью параметров типа (т.е. заполнителей), и теперь можно приступить к изучению классов и интерфейсов из пространства имен `System.Collections.Generic`.

Пространство имен `System.Collections.Generic`

Когда выполняется построение приложения .NET и необходим способ управления данным и в памяти, классы из пространства имен `System.Collections.Generic`, скорее всего, удовлетворят всем требованиям. В начале этой главы кратко упоминались некоторые из необобщенных интерфейсов, реализованных необобщенными классами коллекций. Не должно вызывать удивления, что в пространстве имен `System.Collections.Generic` определены обобщенные замены для многих из них.

В действительность есть много обобщенных интерфейсов, которые расширяют свои необобщенные аналоги. Это может показаться странным; однако благодаря этому, реализации новых классов также поддерживают унаследованную функциональность, имеющуюся у их необобщенных аналогов. Например, `IEnumerable<T>` расширяет `IEnumerable`. В табл. 9.4 документированы основные обобщенные интерфейсы, с которыми придется иметь дело при работе с обобщенными классами коллекций.

**Таблица 9.4. Основные интерфейсы, поддерживаемые классами
из пространства имен `System.Collections.Generic`**

Интерфейс <code>System.Collections.Generic</code>	Назначение
<code>ICollection<T></code>	Определяет общие характеристики (например, размер, перечисление и безопасность к потокам) для всех типов обобщенных коллекций
<code>IComparer<T></code>	Определяет способ сравнения объектов
<code>IDictionary<TKey, TValue></code>	Позволяет объекту обобщенной коллекции представлять свое содержимое посредством пар "ключ/значение"
<code>IEnumerable<T></code>	Возвращает интерфейс <code>IEnumerator<T></code> для заданного объекта
<code>IEnumerator<T></code>	Позволяет выполнять итерацию в стиле <code>foreach</code> по элементам коллекции
<code>ICollection<T></code>	Обеспечивает поведение добавления, удаления и индексации элементов в последовательном списке объектов
<code>ISet<T></code>	Предоставляет базовый интерфейс для абстракции множеств

В пространстве имен `System.Collections.Generic` также определен набор классов, реализующих многие из этих основных интерфейсов. В табл. 9.5 описаны часто используемые классы из этого пространства имен, реализуемые ими интерфейсы и их базовая функциональность.

В пространстве имен `System.Collections.Generic` также определен ряд вспомогательных классов и структур, которые работают в сочетании со специфическим контейнером. Например, тип `LinkedListNode<T>` представляет узел внутри обобщенного контейнера `LinkedList<T>`, исключение `KeyNotFoundException` генерируется при попытке получить элемент из коллекции с указанием несуществующего ключа, и т.д.

Важно отметить, что `mscorlib.dll` и `System.dll` — не единственные сборки, которые добавляют новые типы в пространство имен `System.Collections.Generic`. Например, `System.Core.dll` добавляет класс `HashSet<T>`. Детальные сведения о пространстве имен `System.Collections.Generic` доступны в документации .NET Framework.

Таблица 9.5. Классы из пространства имен `System.Collections.Generic`

Обобщенный класс	Поддерживаемые основные интерфейсы	Назначение
<code>Dictionary<TKey, TValue></code>	<code>ICollection<T></code> , <code>IDictionary<TKey, TValue></code> , <code>IEnumerable<T></code>	Представляет обобщенную коллекцию ключей и значений
<code>LinkedList<T></code>	<code>ICollection<T></code> , <code>IEnumerable<T></code>	Представляет двухсвязный список
<code>List<T></code>	<code>ICollection<T></code> , <code>IEnumerable<T></code> , <code>IList<T></code>	Последовательный список элементов с динамически изменяемым размером
<code>Queue<T></code>	<code>ICollection</code> (Это не опечатка! Именно так называется необобщенный интерфейс коллекции), <code>IEnumerable<T></code>	Обобщенная реализация очереди — списка, работающего по алгоритму “первый вошел — первый вышел” (FIFO)
<code>SortedDictionary<TKey, TValue></code>	<code>ICollection<T></code> , <code>IDictionary<TKey, TValue></code> , <code>IEnumerable<T></code>	Обобщенная реализация словаря — отсортированного множества пар “ключ/значение”
<code>SortedSet<T></code>	<code>ICollection<T></code> , <code>IEnumerable<T></code> , <code>ISet<T></code>	Представляет коллекцию объектов, поддерживаемых в сортированном порядке без дублирования
<code>Stack<T></code>	<code>ICollection</code> (Это не опечатка! Это интерфейс необобщенной коллекции!), <code>IEnumerable<T></code>	Обобщенная реализация стека — списка, работающего по алгоритму “последний вошел — первый вышел” (LIFO)

В любом случае следующая задача заключается в том, чтобы научиться использовать некоторые из этих обобщенных классов коллекций. Но прежде давайте рассмотрим языковые средства C# (впервые появившиеся в .NET 3.5), которые упрощают наполнение данными обобщенных (и необобщенных) коллекций.

Синтаксис инициализации коллекций

В главе 4 был представлен синтаксис инициализации объектов, который позволяет устанавливать свойства для новой переменной во время ее конструирования. С ним тесно связан синтаксис инициализации коллекций. Это средство языка C# позволяет наполнять множество контейнеров (таких как `ArrayList` или `List<T>`) элементами с использованием синтаксиса, похожего на тот, что применяется для наполнения базового массива.

На заметку! Синтаксис инициализации коллекций может применяться только к классам, которые поддерживают метод `Add()`, формализованный интерфейсами `ICollection<T>/ICollection`.

Рассмотрим следующие примеры:

```
// Инициализация стандартного массива.
int[] myArrayOfInts = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

```
// Инициализация обобщенного списка List<> элементов int.
List<int> myGenericList = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// Инициализация ArrayList числовыми данными.
ArrayList myList = new ArrayList { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

Если контейнер управляет коллекцией классов или структур, можно смешивать синтаксис инициализации объектов с синтаксисом инициализации коллекций, создавая некоторый функциональный код. Возможно, вы помните класс `Point` из главы 5, в котором были определены два свойства `X` и `Y`. Чтобы построить обобщенный список `List<T>` объектов `P`, можно написать такой код:

```
List<Point> myListOfPoints = new List<Point>
{
    new Point { X = 2, Y = 2 },
    new Point { X = 3, Y = 3 },
    new Point(PointColor.BloodRed) { X = 4, Y = 4 }
};

foreach (var pt in myListOfPoints)
{
    Console.WriteLine(pt);
}
```

Преимущество этого синтаксиса в экономии большого объема клавиатурного ввода. Хотя вложенные фигурные скобки затрудняют чтение, если не позаботиться о форматировании, только представьте себе объем кода, который потребовалось бы написать для наполнения следующего списка `List<T>` объектов `Rectangle`, если бы не было синтаксиса инициализации коллекций (вспомните, как в главе 4 создавался класс `Rectangle`, который содержал два свойства, инкапсулирующих объекты `Point`):

```
List<Rectangle> myListOfRects = new List<Rectangle>
{
    new Rectangle { TopLeft = new Point { X = 10, Y = 10 },
                  BottomRight = new Point { X = 200, Y = 200 } },
    new Rectangle { TopLeft = new Point { X = 2, Y = 2 },
                  BottomRight = new Point { X = 100, Y = 100 } },
    new Rectangle { TopLeft = new Point { X = 5, Y = 5 },
                  BottomRight = new Point { X = 90, Y = 75 } }
};

foreach (var r in myListOfRects)
{
    Console.WriteLine(r);
}
```

Работа с классом `List<T>`

Для начала создадим новый проект консольного приложения по имени `FunWithGenericCollections`. Обратите внимание, что в первоначальном файле кода C# пространство имен `System.Collections.Generic` уже импортировано.

Первый обобщенный класс, который мы рассмотрим — это `List<T>`, который уже применялся ранее в этой главе. Из всех классов пространства имен `System.Collections.Generic` класс `List<T>` будет использоваться наиболее часто, потому что он позволяет динамически изменять размер контейнера. Чтобы проиллюстрировать основы этого типа, добавьте в класс `Program` метод `UseGenericList()`, в котором `List<T>` применяется для манипуляций множеством объектов `Person`; вы должны помнить, что в классе `Person` определены три свойства (`Age`, `FirstName` и `LastName`) и специальная реализация метода `ToString()`.

```

static void UseGenericList()
{
    // Создать список объектов Person и заполнить его с помощью
    // синтаксиса инициализации объектов/коллекций.
    List<Person> people = new List<Person>()
    {
        new Person {FirstName= "Homer", LastName="Simpson", Age=47},
        new Person {FirstName= "Marge", LastName="Simpson", Age=45},
        new Person {FirstName= "Lisa", LastName="Simpson", Age=9},
        new Person {FirstName= "Bart", LastName="Simpson", Age=8}
    };
    // Вывести на консоль количество элементов в списке.
    Console.WriteLine("Items in list: {0}", people.Count);
    // Выполнить перечисление по списку.
    foreach (Person p in people)
        Console.WriteLine(p);
    // Вставить новую персону.
    Console.WriteLine("\n->Inserting new person.");
    people.Insert(2, new Person { FirstName = "Maggie", LastName = "Simpson", Age = 2 });
    Console.WriteLine("Items in list: {0}", people.Count);
    // Скопировать данные в новый массив.
    Person[] arrayOfPeople = people.ToArray();
    for (int i = 0; i < arrayOfPeople.Length; i++)
    {
        Console.WriteLine("First Names: {0}", arrayOfPeople[i].FirstName);
    }
}

```

Здесь вы используете синтаксис инициализации для наполнения вашего `List<T>` объектами как сокращенную нотацию вызовов `Add()` множество раз. После вывода количества элементов в коллекции (а также перечисления по всем элементам) произойдет вызов `Insert()`. Как можно видеть, `Insert()` позволяет вставить новый элемент в `List<T>` по указанному индексу.

И, наконец, обратите внимание на вызов метода `ToArray()`, который возвращает массив объектов `Person`, основанный на содержимом исходного `List<T>`. Затем осуществляется проход по всем элементам этого массива с использованием синтаксиса индексируемого массива. Если вы вызовете этот метод из `Main()`, то получите следующий вывод:

```

***** Fun with Generic Collections *****

Items in list: 4
Name: Homer Simpson, Age: 47
Name: Marge Simpson, Age: 45
Name: Lisa Simpson, Age: 9
Name: Bart Simpson, Age: 8

->Inserting new person.
Items in list: 5
First Names: Homer
First Names: Marge
First Names: Maggie
First Names: Lisa
First Names: Bart

```

В классе `List<T>` определено множество дополнительных членов, представляющих интерес, поэтому за дополнительной информацией обращайтесь в документацию .NET Framework. Теперь рассмотрим еще несколько обобщенных коллекций: `Stack<T>`, `Queue<T>` и `SortedSet<T>`. Это должно дать более полное понимание базовых вариантов хранения данных в приложении.

Работа с классом Stack<T>

Класс Stack<T> представляет коллекцию элементов, работающую по алгоритму “последний вошел — первый вышел” (LIFO). Как и можно было ожидать, в Stack<T> определены члены Push() и Pop(), предназначенные для вставки и удаления элементов в стеке. Приведенный ниже метод создает стек объектов Person:

```
static void UseGenericStack()
{
    Stack<Person> stackOfPeople = new Stack<Person>();
    stackOfPeople.Push(new Person
        { FirstName = "Homer", LastName = "Simpson", Age = 47 });
    stackOfPeople.Push(new Person
        { FirstName = "Marge", LastName = "Simpson", Age = 45 });
    stackOfPeople.Push(new Person
        { FirstName = "Lisa", LastName = "Simpson", Age = 9 });

    // Просмотреть верхний элемент, вытолкнуть его и посмотреть снова.
    Console.WriteLine("First person is: {0}", stackOfPeople.Peek());
    Console.WriteLine("Popped off {0}", stackOfPeople.Pop());
    Console.WriteLine("\nFirst person is: {0}", stackOfPeople.Peek());
    Console.WriteLine("Popped off {0}", stackOfPeople.Pop());
    Console.WriteLine("\nFirst person item is: {0}", stackOfPeople.Peek());
    Console.WriteLine("Popped off {0}", stackOfPeople.Pop());

    try
    {
        Console.WriteLine("\nFirst person is: {0}", stackOfPeople.Peek());
        Console.WriteLine("Popped off {0}", stackOfPeople.Pop());
    }
    catch (InvalidOperationException ex)
    {
        Console.WriteLine("\nError! {0}", ex.Message); // Ошибка! Стек пуст.
    }
}
```

В коде строится стек, содержащий информацию о трех людях, добавленных в порядке их имен: Homer, Marge и Lisa. Заглядывая (посредством Peek()) в стек, вы всегда видите объект, находящийся на его вершине; поэтому первый вызов Peek() вернет третий объект Person. После серии вызовов Pop() и Peek() стек, наконец, опустошается, после чего вызовы Peek() и Pop() приводят к генерации системного исключения. Вывод этого примера показан ниже:

```
***** Fun with Generic Collections *****

First person is: Name: Lisa Simpson, Age: 9
Popped off Name: Lisa Simpson, Age: 9

First person is: Name: Marge Simpson, Age: 45
Popped off Name: Marge Simpson, Age: 45

First person item is: Name: Homer Simpson, Age: 47
Popped off Name: Homer Simpson, Age: 47

Error! Stack empty.
```

Работа с классом Queue<T>

Очереди — это контейнеры, гарантирующие доступ к элементам в стиле “первый вошел — первый вышел” (FIFO). К сожалению, людям приходится сталкиваться с очередями каждый день: очереди в банк, очереди в кинотеатр, очереди в кафе. Когда нужно смоделировать сценарий, в котором элементы обрабатываются в режиме FIFO, класс

`Queue<T>` подходит наилучшим образом. В дополнение к функциональности, предоставляемой поддерживаемыми интерфейсами, `Queue` определяет основные члены, которые перечислены в табл. 9.6.

Таблица 9.6. Члены типа `Queue<T>`

Член <code>Queue<T></code>	Назначение
<code>Dequeue()</code>	Удаляет и возвращает объект из начала <code>Queue<T></code>
<code>Enqueue()</code>	Добавляет объект в конец <code>Queue<T></code>
<code>Peek()</code>	Возвращает объект из начала <code>Queue<T></code> , не удаляя его

Теперь давайте посмотрим на эти методы в работе. Можно снова вернуться к классу `Person` и построить объект `Queue<T>`, эмулирующий очередь людей, которые ожидают заказа кофе. Для начала представим, что имеется следующий статический метод:

```
static void GetCoffee(Person p)
{
    Console.WriteLine("{0} got coffee!", p.FirstName);
}
```

Кроме того, есть также дополнительный вспомогательный метод, который вызывает `GetCoffee()` **внутренне**:

```
static void UseGenericQueue()
{
    // Создать очередь из трех человек.
    Queue<Person> peopleQ = new Queue<Person>();
    peopleQ.Enqueue(new Person {FirstName= "Homer",
        LastName="Simpson", Age=47});
    peopleQ.Enqueue(new Person {FirstName= "Marge",
        LastName="Simpson", Age=45});
    peopleQ.Enqueue(new Person {FirstName= "Lisa",
        LastName="Simpson", Age=9});

    // Кто первый в очереди?
    Console.WriteLine("{0} is first in line!", peopleQ.Peek().FirstName);

    // Удалить всех из очереди.
    GetCoffee(peopleQ.Dequeue());
    GetCoffee(peopleQ.Dequeue());
    GetCoffee(peopleQ.Dequeue());

    // Попробовать извлечь кого-то из очереди снова.
    try
    {
        GetCoffee(peopleQ.Dequeue());
    }
    catch(InvalidOperationException e)
    {
        Console.WriteLine("Error! {0}", e.Message); // Ошибка! Очередь пуста.
    }
}
```

Здесь вы вставляете три элемента в класс `Queue<T>`, используя метод `Enqueue()`. Вызов `Peek()` позволяет просматривать (но не удалять) первый элемент, находящийся в данный момент в `Queue`. Наконец, вызов `Dequeue()` удаляет элемент из очереди и посылает его вспомогательной функции `GetCoffee()` для обработки. Обратите внимание, что если вы пытаетесь удалять элементы из пустой очереди, генерируется исключение

времени выполнения. Ниже приведен вывод, который будет получен при вызове этого метода:

```
***** Fun with Generic Collections *****
Homer is first in line!
Homer got coffee!
Marge got coffee!
Lisa got coffee!
Error! Queue empty.
```

Работа с классом SortedSet<T>

Класс SortedSet<T> удобен тем, что при вставке или удалении элементов он автоматически обеспечивает сортировку элементов в наборе. Класс SortedSet<T> понадобится информировать о том, как должны сортироваться объекты, за счет передачи его конструктору аргумента — объекта, реализующего обобщенный интерфейс IComparer<T>.

Начнем с создания нового класса по имени SortPeopleByAge, реализующего IComparer<T>, где T — тип Person. Вспомните, что этот интерфейс определяет единственный метод по имени Compare(), в котором можно запрограммировать логику сравнения элементов. Ниже приведена простая реализация этого класса:

```
class SortPeopleByAge : IComparer<Person>
{
    public int Compare(Person firstPerson, Person secondPerson)
    {
        if (firstPerson.Age > secondPerson.Age)
            return 1;
        if (firstPerson.Age < secondPerson.Age)
            return -1;
        else
            return 0;
    }
}
```

Теперь добавим в класс Program следующий новый метод, который должен будет вызван в Main():

```
static void UseSortedSet()
{
    // Создать несколько людей разного возраста.
    SortedSet<Person> setOfPeople = new SortedSet<Person>(new SortPeopleByAge())
    {
        new Person {FirstName= "Homer", LastName="Simpson", Age=47},
        new Person {FirstName= "Marge", LastName="Simpson", Age=45},
        new Person {FirstName= "Lisa", LastName="Simpson", Age=9},
        new Person {FirstName= "Bart", LastName="Simpson", Age=8}
    };

    // Обратите внимание, что элементы отсортированы по возрасту.
    foreach (Person p in setOfPeople)
    {
        Console.WriteLine(p);
    }
    Console.WriteLine();

    // Добавить еще несколько людей разного возраста.
    setOfPeople.Add(new Person { FirstName = "Saku", LastName = "Jones", Age = 1 });
    setOfPeople.Add(new Person { FirstName = "Mikko", LastName = "Jones", Age = 32 });
}
```

```
// Элементы по-прежнему отсортированы по возрасту.
foreach (Person p in setOfPeople)
{
    Console.WriteLine(p);
}
```

После запуска приложения видно, что список объектов будет всегда упорядочен по значению свойства Age, независимо от порядка вставки и удаления объектов в коллекцию:

```
***** Fun with Generic Collections *****

Name: Bart Simpson, Age: 8
Name: Lisa Simpson, Age: 9
Name: Marge Simpson, Age: 45
Name: Homer Simpson, Age: 47

Name: Saku Jones, Age: 1
Name: Bart Simpson, Age: 8
Name: Lisa Simpson, Age: 9
Name: Mikko Jones, Age: 32
Name: Marge Simpson, Age: 45
Name: Homer Simpson, Age: 47
```

Исходный код. Проект FunWithGenericCollections доступен в подкаталоге Chapter 09.

Пространство имен System.Collections.ObjectModel

Теперь, когда вы понимаете основы работы с обобщенными классами, мы можем кратко взглянуть на дополнительное пространство имен, связанное с коллекциями — System.Collections.ObjectModel. Это относительно небольшое пространство имен, содержащее лишь горстку классов. В табл. 9.7 документированы два класса, которые вы должны знать обязательно.

Таблица 9.7. Полезные типы в System.Collections.ObjectModel

Тип System.Collections.ObjectModel	Назначение
ObservableCollection<T>	Представляет динамическую коллекцию данных, которая обеспечивает уведомления при добавлении элементов, их удалении и обновлении всего списка
ReadOnlyObservableCollection<T>	Представляет версию ObservableCollection<T>, предназначенную только для чтения

Класс ObservableCollection<T> очень удобен в том, что он обладает возможностью информировать внешние объекты, когда его содержимое каким-нибудь образом изменяется (как вы могли догадаться, работа с ReadOnlyObservableCollection<T> очень похожа, но имеет природу только для чтения).

Работа с ObservableCollection<T>

Создадим новое консольное приложение по имени FunWithObservableCollection и импортируем в первоначальный файл кода C# пространство имен System.Collections.ObjectModel. Во многих отношениях работа с ObservableCollection<T> идентичная

работе с `List<T>`, учитывая то, что оба класса реализуют одни и те же основные интерфейсы. Уникальность класса `ObservableCollection<T>` состоит в том, что он поддерживает событие по имени `CollectionChanged`. Это событие будет инициироваться каждый раз, когда вставляется новый элемент, удаляется (или перемещается) текущий элемент либо модифицируется вся коллекция целиком.

Подобно любому событию, `CollectionChanged` определено в терминах делегата, которым в данном случае является `NotifyCollectionChangedEventHandler`. Этот делегат может вызывать любой метод, принимающий объект в первом параметре и `NotifyCollectionChangedEventArgs` — во втором. Рассмотрим следующий метод `Main()`, который заполняет наблюдаемую коллекцию, содержащую объекты `Person`, и привязывается к событию `CollectionChanged`:

```
class Program
{
    static void Main(string[] args)
    {
        // Сделать коллекцию наблюдаемой и добавить в нее несколько объектов Person.
        ObservableCollection<Person> people = new ObservableCollection<Person>()
        {
            new Person{ FirstName = "Peter", LastName = "Murphy", Age = 52 },
            new Person{ FirstName = "Kevin", LastName = "Key", Age = 48 },
        };

        // Привязаться к событию CollectionChanged.
        people.CollectionChanged += people_CollectionChanged;
    }

    static void people_CollectionChanged(object sender,
        System.Collections.Specialized.NotifyCollectionChangedEventArgs e)
    {
        throw new NotImplementedException();
    }
}
```

Входной параметр `NotifyCollectionChangedEventArgs` определяет два важных свойства, `OldItems` и `NewItems`, предоставляющие список элементов, которые имелись в коллекции перед генерацией события, и новых элементов, которые участвовали в изменении. Тем не менее, эти списки будут исследоваться только при подходящих обстоятельствах. Вспомните, что событие `CollectionChanged` может инициироваться, когда элементы добавляются, удаляются, перемещаются или сбрасываются. Чтобы выяснить, какое из этих действий запустило событие, можно воспользоваться свойством `Action` объекта `NotifyCollectionChangedEventArgs`. Свойство `Action` может проверяться на предмет равенства с любым из следующих членов перечисления `NotifyCollectionChangedAction`:

```
public enum NotifyCollectionChangedAction
{
    Add = 0,
    Remove = 1,
    Replace = 2,
    Move = 3,
    Reset = 4,
}
```

Ниже приведена реализация обработчика событий `CollectionChanged`, который будет обходить старый и новый наборы, когда элемент вставляется или удаляется из рабочей коллекции:

```

static void people_CollectionChanged(object sender,
    System.Collections.Specialized.NotifyCollectionChangedEventArgs e)
{
    // Выяснить действие, которое привело к генерации события.
    Console.WriteLine("Action for this event: {0}", e.Action);

    // Было что-то удалено.
    if (e.Action == System.Collections.Specialized.NotifyCollectionChangedAction.Remove)
    {
        Console.WriteLine("Here are the OLD items:");
        foreach (Person p in e.OldItems)
        {
            Console.WriteLine(p.ToString());
        }
        Console.WriteLine();
    }

    // Было что-то добавлено.
    if (e.Action == System.Collections.Specialized.NotifyCollectionChangedAction.Add)
    {
        // Теперь вывести новые элементы, которые были вставлены.
        Console.WriteLine("Here are the NEW items:");
        foreach (Person p in e.NewItems)
        {
            Console.WriteLine(p.ToString());
        }
    }
}

```

Теперь, предполагая, что вы обновили метод `Main()` для добавления и удаления элемента, вы увидите следующий вывод:

```

Action for this event: Add
Here are the NEW items:
Name: Fred Smith, Age: 32

Action for this event: Remove
Here are the OLD items:
Name: Peter Murphy, Age: 52

```

На этом исследование различных пространств имен, связанных с коллекциями, в библиотеках базовых классов .NET завершено. В конце этой главы будет также показано, как и для чего строить собственные обобщенные методы и обобщенные типы.

Исходный код. Проект `FunWithObservableCollection` доступен в подкаталоге `Chapter 09`.

Создание специальных обобщенных методов

Хотя большинство разработчиков обычно используют существующие обобщенные типы из библиотек базовых классов, можно также строить собственные обобщенные методы и специальные обобщенные типы. Чтобы понять, как включать обобщения в собственные проекты, начнем с построения обобщенного метода обмена, предварительно создав новое консольное приложение по имени `CustomGenericMethods`.

Построение специальных обобщенных методов представляет собой более развитую версию традиционной перегрузки методов. В главе 2 было показано, что перегрузка — это определение нескольких версий одного метода, отличающихся друг от друга количеством или типами параметров.

Хотя перегрузка — полезное средство объектно-ориентированного языка, при этом возникает проблема, вызванная появлением огромного количества методов, которые в конечном итоге делают одно и то же. Например, предположим, что требуется создать методы, которые позволяют менять местами два фрагмента данных. Можно начать с написания простого метода для обмена двух целочисленных значений:

```
// Обмен двух значений int.
static void Swap(ref int a, ref int b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

Пока что все хорошо. А теперь представим, что нужно поменять местами два объекта Person; для этого понадобится новая версия метода Swap():

```
// Обмен двух объектов Person.
static void Swap(ref Person a, ref Person b)
{
    Person temp;
    temp = a;
    a = b;
    b = temp;
}
```

Уже должно стать ясно, куда это приведет. Если также потребуются поменять местами два значения с плавающей точкой, две битовые карты, два объекта автомобилей или еще что-нибудь, придется писать дополнительные методы, что в конечном итоге превратится в кошмар при сопровождении. Правда, можно было бы построить один (необобщенный) метод, оперирующий параметрами типа object, но тогда возникнут проблемы, которые были описаны ранее в этой главе, т.е. упаковка, распаковка, недостаток безопасности типов, явное приведение и т.п.

Всякий раз, когда имеется группа перегруженных методов, отличающихся только входными аргументами — это явный признак того, что за счет применения обобщений удастся облегчить себе жизнь. Рассмотрим следующий обобщенный метод Swap<T>, который может менять местами два значения T:

```
// Этот метод обменивает между собой значения двух
// элементов типа, переданного в параметре <T>.
static void Swap<T>(ref T a, ref T b)
{
    Console.WriteLine("You sent the Swap() method a {0}",
        typeof(T));
    T temp;
    temp = a;
    a = b;
    b = temp;
}
```

Обратите внимание, что обобщенный метод определен за счет спецификации параметра типа после имени метода и перед списком параметров. Здесь устанавливается, что метод Swap() может оперировать любыми двумя параметрами типа <T>. Чтобы немного прояснить картину, имя подставляемого типа выводится на консоль с использованием операции typeof(). Теперь рассмотрим следующий метод Main(), обменивающий значениями целочисленные и строковые переменные:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Custom Generic Methods *****\n");

    // Обмен двух значений int.
    int a = 10, b = 90;
    Console.WriteLine("Before swap: {0}, {1}", a, b);
    Swap<int>(ref a, ref b);
    Console.WriteLine("After swap: {0}, {1}", a, b);
    Console.WriteLine();

    // Обмен двух строк.
    string s1 = "Hello", s2 = "There";
    Console.WriteLine("Before swap: {0} {1}!", s1, s2);
    Swap<string>(ref s1, ref s2);
    Console.WriteLine("After swap: {0} {1}!", s1, s2);

    Console.ReadLine();
}
```

Ниже показан вывод этого примера:

```
***** Fun with Custom Generic Methods *****

Before swap: 10, 90
You sent the Swap() method a System.Int32
After swap: 90, 10

Before swap: Hello There!
You sent the Swap() method a System.String
After swap: There Hello!
```

Основное преимущество этого подхода в том, что нужно будет сопровождать только одну версию `Swap<T>()`, хотя она может оперировать любыми двумя элементами определенного типа, причем в безопасной к типам манере. Еще лучше то, что находящиеся в стеке элементы остаются в стеке, а расположенные в куче — соответственно, в куче.

Выведение параметров типа

При вызове таких обобщенных методов, как `Swap<T>`, можно опускать параметр типа, если (и только если) обобщенный метод требует аргументов, поскольку компилятор может вывести параметр типа из параметров членов. Например, добавив к `Main()` следующий код, можно обменивать значения `System.Boolean`:

```
// Компилятор самостоятельно выведет тип System.Boolean.
bool b1 = true, b2 = false;
Console.WriteLine("Before swap: {0}, {1}", b1, b2);
Swap(ref b1, ref b2);
Console.WriteLine("After swap: {0}, {1}", b1, b2);
```

Несмотря на то что компилятор может определить параметр типа на основе типа данных, использованного в объявлении `b1` и `b2`, стоит выработать привычку всегда указывать параметр типа явно:

```
Swap<string>(ref b1, ref b2);
```

Это позволит понять неопытным программистам, что данный метод на самом деле является обобщенным. Более того, выводение типов параметров работает только в том случае, если обобщенный метод принимает, по крайней мере, один параметр.

Например, предположим, что в классе `Program` определен следующий обобщенный метод:

```
static void DisplayBaseClass<T>()
{
    // BaseType – это метод, используемый в рефлексии;
    // он будет рассматриваться в главе 15.
    Console.WriteLine("Base class of {0} is: {1}.",
        typeof(T), typeof(T).BaseType);
}
```

При его вызове потребуется указать параметр типа:

```
static void Main(string[] args)
{
    ...
    // Необходимо указать параметр типа,
    // если метод не принимает параметров.
    DisplayBaseClass<int>();
    DisplayBaseClass<string>();

    // Ошибка на этапе компиляции! Нет параметров?
    // Значит, необходимо указать тип для подстановки!
    // DisplayBaseClass();
    Console.ReadLine();
}
```

В настоящее время обобщенные методы `Swap<T>` и `DisplayBaseClass<T>` определены в классе `Program` приложения. Конечно, как и любой другой метод, если вы захотите определить эти члены в отдельном классе (`MyGenericMethods`), то можно поступить так:

```
public static class MyGenericMethods
{
    public static void Swap<T>(ref T a, ref T b)
    {
        Console.WriteLine("You sent the Swap() method a {0}",
            typeof(T));
        T temp;
        temp = a;
        a = b;
        b = temp;
    }

    public static void DisplayBaseClass<T>()
    {
        Console.WriteLine("Base class of {0} is: {1}.",
            typeof(T), typeof(T).BaseType);
    }
}
```

Статические методы `Swap<T>` и `DisplayBaseClass<T>` находятся в контексте нового типа статического класса, поэтому потребуется указать имя типа при вызове каждого члена, например:

```
MyGenericMethods.Swap<int>(ref a, ref b);
```

Разумеется, методы не обязательно должны быть статическими. Если бы `Swap<T>` и `DisplayBaseClass<T>` были методами уровня экземпляра (и определенными в нестатическом классе), понадобилось бы просто создать экземпляр `MyGenericMethods` и вызывать их с использованием объектной переменной:

```
MyGenericMethods c = new MyGenericMethods();
c.Swap<int>(ref a, ref b);
```

Создание специальных обобщенных структур и классов

Теперь, когда известно, как определяются и вызываются обобщенные методы, давайте посмотрим, каким образом сконструировать обобщенную структуру (процесс построения обобщенного класса идентичен) в новом проекте консольного приложения по имени `GenericPoint`. Предположим, что строится обобщенная структура `Point`, которая поддерживает единственный параметр типа, определяющий внутреннее представление координат (x, y). Вызывающий код должен иметь возможность создавать типы `Point<T>` следующим образом:

```
// Точка с координатами int.
Point<int> p = new Point<int>(10, 10);

// Точка с координатами double.
Point<double> p2 = new Point<double>(5.4, 3.3);
```

Вот полное определение `Point<T>` с последующим анализом:

```
// Обобщенная структура Point.
public struct Point<T>
{
    // Обобщенные данные состояния.
    private T xPos;
    private T yPos;

    // Обобщенный конструктор.
    public Point(T xVal, T yVal)
    {
        xPos = xVal;
        yPos = yVal;
    }

    // Обобщенные свойства.
    public T X
    {
        get { return xPos; }
        set { xPos = value; }
    }

    public T Y
    {
        get { return yPos; }
        set { yPos = value; }
    }

    public override string ToString()
    {
        return string.Format("[{0}, {1}]", xPos, yPos);
    }

    // Сбросить поля в стандартные значения
    // для заданного параметра типа.
    public void ResetPoint()
    {
        xPos = default(T);
        yPos = default(T);
    }
}
```

Ключевое слово `default` в обобщенном коде

Как видите, структура `Point<T>` использует параметр типа в определении данных полей, аргументов конструктора и определении свойств. Обратите внимание, что в дополнение к переопределению `ToString()`, в `Point<T>` определен метод по имени `ResetPoint()`, в котором применяется не встречавшийся ранее новый синтаксис:

```
// Ключевое слово default в языке C# перегружено.
// При использовании с обобщениями оно представляет
// стандартное значение для параметра типа.
public void ResetPoint()
{
    X = default(T);
    Y = default(T);
}
```

С появлением обобщений ключевое слово `default` обрело второй смысл. В дополнение к использованию с конструкцией `switch`, оно теперь может применяться для установки стандартного значения для параметра типа. Это очень удобно, учитывая, что обобщенный тип не знает заранее, что будет подставлено вместо заполнителя в угловых скобках, и потому не может безопасно строить предположения о стандартных значениях. Умолчания для параметров типа следующие:

1. стандартное значение числовых величин равно 0;
2. ссылочные типы имеют стандартное значение `null`;
3. поля структур устанавливаются в 0 (для типов значений) или в `null` (для ссылочных типов).

Для `Point<T>` можно было установить значение `X` и `Y` в 0 напрямую, исходя из предположения, что вызывающий код будет применять только числовые значения. Однако за счет использования синтаксиса `default(T)` повышается общая гибкость обобщенного типа. В любом случае теперь можно применять методы `Point<T>` следующим образом:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Generic Structures *****\n");
    // Объект Point, в котором используются int.
    Point<int> p = new Point<int>(10, 10);
    Console.WriteLine("p.ToString()={0}", p.ToString());
    p.ResetPoint();
    Console.WriteLine("p.ToString()={0}", p.ToString());
    Console.WriteLine();
    // Объект Point, в котором используются double.
    Point<double> p2 = new Point<double>(5.4, 3.3);
    Console.WriteLine("p2.ToString()={0}", p2.ToString());
    p2.ResetPoint();
    Console.WriteLine("p2.ToString()={0}", p2.ToString());
    Console.ReadLine();
}
```

Ниже показан вывод этого примера:

```
***** Fun with Generic Structures *****
p.ToString()=[10, 10]
p.ToString()=[0, 0]
p2.ToString()=[5.4, 3.3]
p2.ToString()=[0, 0]
```

Исходный код. Проект GenericPoint доступен в подкаталоге Chapter 09.

Ограничение параметров типа

Как показано в этой главе, любой обобщенный элемент имеет, по крайней мере, один параметр типа, который должен быть указан при взаимодействии с обобщенным типом или членом. Одно это позволит строить безопасный в отношении типов код; однако платформа .NET позволяет использовать ключевое слово `where` для указания особых требований к определенному параметру типа.

С помощью ключевого слова `this` можно добавлять набор ограничений к конкретному параметру типа, которые компилятор C# проверит во время компиляции. В частности, параметр типа можно ограничить, как описано в табл. 9.8.

Таблица 9.8. Возможные ограничения параметров типа для обобщений

Ограничение обобщения	Назначение
<code>where T : struct</code>	Параметр типа <code><T></code> должен иметь в своей цепочке наследования <code>System.ValueType</code> (т.е. <code><T></code> должен быть структурой)
<code>where T : class</code>	Параметр типа <code><T></code> не должен иметь <code>System.ValueType</code> в своей цепочке наследования (т.е. <code><T></code> должен быть ссылочным типом)
<code>where T : new()</code>	Параметр типа <code><T></code> должен иметь стандартный конструктор. Это полезно, если обобщенный тип должен создавать экземпляры параметра типа, поскольку не удастся предположить формат специальных конструкторов. Обратите внимание, что в типе с несколькими ограничениями это ограничение должно указываться последним
<code>where T : <i>ИмяБазовогоКласса</i></code>	Параметр типа <code><T></code> должен быть наследником класса, указанного в <i>ИмяБазовогоКласса</i>
<code>where T : <i>ИмяИнтерфейса</i></code>	Параметр типа <code><T></code> должен реализовать интерфейс, указанный в <i>ИмяИнтерфейса</i> . Можно задавать несколько интерфейсов, разделяя их запятыми

Если только не требуется строить какие-то исключительно безопасные к типам специальные коллекции, возможно, никогда не придется использовать ключевое слово `where` в проектах C#. Так или иначе, но в следующих нескольких примерах (частичного) кода демонстрируется работа с ключевым словом `where`.

Примеры использования ключевого слова `where`

Будем исходить из того, что создан специальный обобщенный класс, и необходимо гарантировать наличие в параметре типа стандартного конструктора. Это может быть полезно, когда специальный обобщенный класс должен создавать экземпляры `T`, потому что стандартный конструктор — это единственный конструктор, потенциально общий для всех типов. Также подобного рода ограничение `T` позволит производить проверку во время компиляции; если `T` — ссылочный тип, то компилятор напомним программисту о необходимости переопределения стандартного конструктора в объявлении класса (если помните, стандартные конструкторы удаляются из классов, в которых определены собственные конструкторы).


```
// Класс MyGenericClass унаследован от object, причем содержащиеся
// в нем элементы должны иметь стандартный конструктор.
public class MyGenericClass<T> where T : new()
{
    ...
}
```

Обратите внимание, что конструкция `where` указывает параметр типа, на который накладывается ограничение, а за ним следует операция двоеточия. После этой операции перечисляются все возможные ограничения (в данном случае — стандартный конструктор). Ниже показан еще один пример:

```
// MyGenericClass унаследован от Object, причем содержащиеся
// в нем элементы должны относиться к классу, реализующему IDrawable,
// и поддерживать стандартный конструктор.
public class MyGenericClass<T> where T : class, IDrawable, new()
{...}
```

В данном случае к `T` предъявляются три требования. Во-первых, это должен быть ссылочный тип (не структура), что помечено лексемой `class`. Во-вторых, `T` должен реализовывать интерфейс `IDrawable`. В-третьих, он также должен иметь стандартный конструктор. Множество ограничений перечисляются в списке, разделенном запятыми; однако имейте в виду, что ограничение `new()` всегда должно идти последним! По этой причине следующий код не скомпилируется:

```
// Ошибка! Ограничение new() должно быть последним в списке!
public class MyGenericClass<T> where T : new(), class, IDrawable
{
    ...
}
```

В случае создания обобщенного класса коллекции с несколькими параметрами типа можно указывать уникальный набор ограничений для каждого параметра с помощью отдельной конструкции `where`:

```
// <K> должен расширять SomeBaseClass и иметь стандартный конструктор, в то время
// как <T> должен быть структурой и реализовывать обобщенный интерфейс IComparable.
public class MyGenericClass<K, T> where K : SomeBaseClass, new()
    where T : IComparable<T>
{
    ...
}
```

Необходимость построения полностью нового обобщенного класса коллекции возникает редко; однако ключевое слово `where` также допускается применять и в обобщенных методах. Например, если необходимо гарантировать, чтобы метод `Swap<T>()` работал только со структурами, измените код следующим образом:

```
// Этот метод обменяет местами любые структуры, но не классы.
static void Swap<T>(ref T a, ref T b) where T : struct
{
    ...
}
```

Обратите внимание, что если ограничить метод `Swap()` подобным образом, обменивать местами объекты `string` (как это делалось в коде примера) уже не получится, поскольку `string` является ссылочным типом.

Недостаток ограничений операций

В конце этой главы следует упомянуть об одном моменте относительно обобщенных методов и ограничений. При создании обобщенных методов может оказаться сюрпризом появление ошибок компиляции во время применения любых операций C# (+, -, *, == и т.д.) к параметрам типа. Например, подумайте, насколько полезным был бы класс, который может выполнять операции `Add()`, `Subtract()`, `Multiply()` и `Divide()` над обобщенными типами:

```
// Ошибка на этапе компиляции! Нельзя
// применять операции к параметрам типа!
public class BasicMath<T>
{
    public T Add(T arg1, T arg2)
    { return arg1 + arg2; }

    public T Subtract(T arg1, T arg2)
    { return arg1 - arg2; }

    public T Multiply(T arg1, T arg2)
    { return arg1 * arg2; }

    public T Divide(T arg1, T arg2)
    { return arg1 / arg2; }
}
```

К сожалению, приведенный выше класс `BasicMath<T>` не скомпилируется. Хотя это может показаться серьезным недостатком, следует снова вспомнить, что обобщения являются общими. Естественно, числовые данные работают достаточно хорошо с бинарными операциями C#. С другой стороны, если аргумент `<T>` является специальным классом или структурой, то компилятор мог бы предположить, что этот класс или структура поддерживает операции +, -, * и /. В идеале язык C# должен был бы позволять ограничивать обобщенные типы поддерживаемыми операциями, например:

```
// Код только для иллюстрации!
public class BasicMath<T> where T : operator +, operator -,
    operator *, operator /
{
    public T Add(T arg1, T arg2)
    { return arg1 + arg2; }

    public T Subtract(T arg1, T arg2)
    { return arg1 - arg2; }

    public T Multiply(T arg1, T arg2)
    { return arg1 * arg2; }

    public T Divide(T arg1, T arg2)
    { return arg1 / arg2; }
}
```

К сожалению, ограничения операций в текущей версии C# не поддерживаются. Тем не менее, достичь желаемого эффекта можно (хотя это и потребует дополнительной работы) за счет определения интерфейса, который поддерживает эти операции (интерфейсы C# могут определять операции!), и последующего указания ограничения интерфейса для обобщенного класса. На этом первоначальный обзор построения специальных обобщенных типов завершен. В главе 10 мы вновь обратимся к теме обобщений, когда будем рассматривать тип делегата `.NET`.

Резюме

Эта глава начиналась с исследования необобщенных типов коллекций в пространствах имен `System.Collections` и `System.Collections.Specialized`, включая различные проблемы, которые связаны со многими необобщенными контейнерами, в том числе недостаток безопасности к типам и накладные расходы времени выполнения в форме операций упаковки и распаковки. Как упоминалось в главе, именно по этим причинам в современных приложениях .NET будут использоваться обобщенные классы коллекций из пространств имен `System.Collections.Generic` и `System.Collections.ObjectModel`.

Вы видели, что обобщенный элемент позволяет указывать заполнители (параметры типа), которые задаются во время создания (или вызова — в случае обобщенных методов). Хотя чаще всего будут просто применяться обобщенные типы, предоставляемые библиотеками базовых классов .NET, можно также создавать собственные обобщенные типы (и обобщенные методы). При этом имеется возможность указания любого количества ограничений (с помощью ключевого слова `where`) для повышения уровня безопасности к типам и обеспечения гарантии выполнения операций над типами в известном объеме, что позволяет предоставить определенные базовые возможности.

В качестве финального замечания: не забывайте, что обобщения можно обнаружить во многих местах библиотек базовых классов .NET. В настоящей главе мы сосредоточили внимание конкретно на обобщенных коллекциях. Тем не менее, по мере изучения остальных материалов книги (и погружении в платформу с учетом своей специфики), вы наверняка найдете обобщенные классы, структуры и делегаты, расположенные в том или ином пространстве имен. Кроме того, будьте настороже относительно обобщенных членов необобщенного класса!

Делегаты, события и лямбда-выражения

Вплоть до этого момента большинство разработанных приложений добавляли различные порции кода к методу `Main()`, тем или иным способом отправляющие запросы заданному объекту. Однако многие приложения требуют, чтобы объект мог обращаться *обратно* к сущности, которая создала его — посредством механизма обратного вызова. Хотя механизмы обратного вызова могут применяться в любом приложении, они особенно важны в графических пользовательских интерфейсах, где элементы управления (такие как кнопки) нуждаются в вызове внешних методов при надлежащих условиях (выполнен щелчок на кнопке, курсор мыши находится на поверхности кнопки и т.п.).

В рамках платформы .NET тип *делегата* является предпочтительным средством определения и реагирования на обратные вызовы в приложении. По сути, тип делегата .NET — это безопасный к типам объект, который “указывает” на метод или список методов, которые могут быть вызваны позднее. Однако в отличие от традиционного указателя на функцию C++, делегаты .NET представляют собой классы, обладающие встроенной поддержкой группового выполнения и асинхронного вызова методов.

В этой главе будет показано, как создавать и управлять типами делегатов, а также использовать ключевое слово `event` в C#, которое облегчает работу с типами делегатов. По ходу дела вы также изучите несколько языковых средств C#, ориентированных на делегаты и события, включая анонимные методы и групповые преобразования методов.

Завершается глава исследованием *лямбда-выражений*. Используя лямбда-операцию C# (`=>`), теперь можно указывать блок операторов кода (и параметры для передачи этим операторам) везде, где требуется строго типизированный делегат. Как будет показано, лямбда-выражение является не более чем маскировкой анонимного метода и представляет собой упрощенный подход к работе с делегатами.

Понятие типа делегата .NET

Прежде чем приступить к формальному определению делегатов .NET, давайте оглянемся немного назад. Исторически сложилось так, что в API-интерфейсе Windows часто использовались указатели на функции в стиле C для создания сущностей, именуемых *функциями обратного вызова* или просто *обратными вызовами*. С помощью обратных вызовов программисты могли конфигурировать одну функцию таким образом, чтобы она осуществляла обратный вызов другой функции в приложении. Применяя такой подход, разработчики Windows смогли обрабатывать щелчки на кнопках, перемещения курсора мыши, выбор пунктов меню и общие двусторонние коммуникации между программными сущностями в памяти.

Проблема со стандартными функциями обратного вызова в стиле C заключается в том, что они представляют собой не более чем простой адрес в памяти. В идеальном случае обратные вызовы могли бы конфигурироваться для включения дополнительной безопасной к типам информации, такой как количество (и типы) параметров и возвращаемого значения (если оно есть) метода, на который они указывают. К сожалению, подобное невозможно делать с традиционными функциями обратного вызова и это, как следовало ожидать, является постоянным источником ошибок, аварийных завершений и прочих неприятностей во время выполнения. Тем не менее, обратные вызовы — это полезная вещь.

В .NET Framework обратные вызовы по-прежнему возможны, и их функциональность обеспечивается в гораздо более безопасной и объектно-ориентированной манере с использованием *делегатов*. По сути, делегат — это безопасный в отношении типов объект, указывающий на другой метод (или, возможно, список методов) приложения, который может быть вызван позднее. В частности, объект делегата поддерживает три важных фрагмента информации:

- *адрес* метода, на котором он вызывается;
- *аргументы* (если есть) этого метода;
- *возвращаемое значение* (если есть) этого метода.

На заметку! Делегаты .NET могут указывать как на статические методы, так и на методы экземпляра.

После того как делегат создан и снабжен необходимой информацией, он может динамически вызывать методы, на которые указывает, во время выполнения. Каждый делегат в .NET Framework (включая специальные делегаты) автоматически оснащается способностью вызывать свои методы *синхронно* или *асинхронно*. Этот факт значительно упрощает задачи программирования, поскольку позволяет вызывать метод во вторичном потоке выполнения без ручного создания и управления объектом Thread.

На заметку! Асинхронное поведение типов делегатов будет рассматриваться во время исследования многопоточности и асинхронных вызовов в главе 19. В настоящей главе мы будем касаться только синхронных аспектов типа делегата.

Определение типа делегата в C#

Для определения делегата в C# используется ключевое слово `delegate`. Имя делегата может быть любым. Однако сигнатура определяемого делегата должна соответствовать сигнатуре метода (или методов), на который он будет указывать. Например, приведенный ниже тип делегата (по имени `BinaryOp`) может указывать на любой метод, который возвращает целое число и принимает два целых числа в качестве входных параметров (вы постройте и будете пользоваться таким делегатом несколько позже в этой главе, а пока что он показан кратко):

```
// Этот делегат может указывать на любой метод, который
// принимает два целых и возвращает целое значение.
public delegate int BinaryOp(int x, int y);
```

Когда компилятор C# обрабатывает тип делегата, он автоматически генерирует запечатанный (*sealed*) класс, унаследованный от `System.MulticastDelegate`. Этот класс (в сочетании с его базовым классом `System.Delegate`) предоставляет необходимую инфраструктуру для делегата, чтобы хранить список методов, подлежащих вызову в бо-

лее позднее время. Например, если просмотреть делегат `BinaryOp` с помощью утилиты `ildasm.exe`, обнаружится класс, показанный на рис. 10.1.

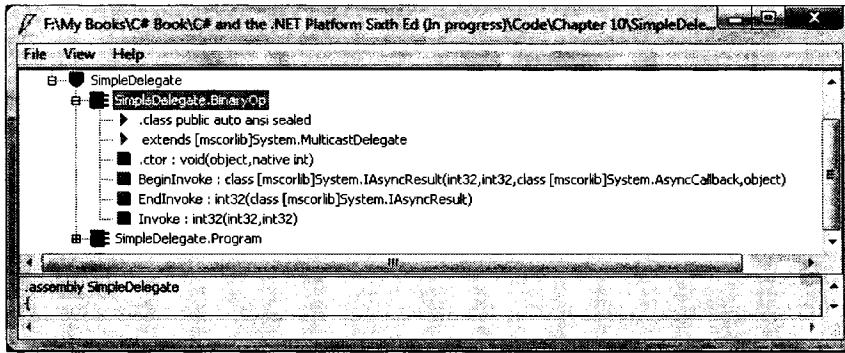


Рис. 10.1. Ключевое слово `delegate` языка C# представляет запечатанный класс, унаследованный от `System.MulticastDelegate`

Как видите, сгенерированный компилятором класс `BinaryOp` определяет три открытых метода. `Invoke()` — пожалуй, главный из них, поскольку он используется для синхронного вызова каждого из методов, поддерживаемых объектом делегата; это означает, что вызывающий код должен ожидать завершения вызова, прежде чем продолжить свою работу. Может показаться странным, что синхронный метод `Invoke()` не должен вызываться явно в коде C#. Как вскоре будет показано, `Invoke()` вызывается “за кулисами”, когда применяется соответствующий синтаксис C#.

Методы `BeginInvoke()` и `EndInvoke()` предлагают возможность вызова текущего метода асинхронным образом, в отдельном потоке выполнения. Имеющим опыт в многопоточной разработке должно быть известно, что одной из основных причин, вынуждающих разработчиков создавать вторичные потоки выполнения, является необходимость вызова методов, которые требуют определенного времени для завершения. Хотя в библиотеках базовых классов .NET предусмотрено несколько пространств имен, связанных с многопоточным и параллельным программированием, делегаты предлагают эту функциональность в готовом виде.

Итак, каким же образом компилятор знает, как следует определить методы `Invoke()`, `BeginInvoke()` и `EndInvoke()`? Чтобы разобраться в процессе, ниже приведен код сгенерированного компилятором типа класса `BinaryOp` (полуужирным курсивом выделены элементы, указанные определенным типом делегата):

```
sealed class BinaryOp : System.MulticastDelegate
{
    public int Invoke(int x, int y);
    public IAsyncResult BeginInvoke(int x, int y,
        AsyncCallback cb, object state);
    public int EndInvoke(IAsyncResult result);
}
```

Первым делом, обратите внимание, что параметры и возвращаемый тип для метода `Invoke()` в точности соответствуют определению делегата `BinaryOp`. Начальные параметры для членов `BeginInvoke()` (в данном случае — два целых числа) также основаны на делегате `BinaryOp`; однако `BeginInvoke()` всегда будет предоставлять два финальных параметра (типа `AsyncCallback` и `object`), используемых для облегчения асинхронного вызова методов. И, наконец, возвращаемый тип `EndInvoke()` идентичен исходному объявлению делегата и всегда принимает единственный параметр — объект, реализующий интерфейс `IAsyncResult`.

Давайте рассмотрим еще один пример. Предположим, что определен тип делегата, который может указывать на любой метод, возвращающий `string` и принимающий три входных параметра `System.Boolean`:

```
public delegate string MyDelegate(bool a, bool b, bool c);
```

На этот раз сгенерированный компилятором класс будет выглядеть так:

```
sealed class MyDelegate : System.MulticastDelegate
{
    public string Invoke(bool a, bool b, bool c);
    public IAsyncResult BeginInvoke(bool a, bool b, bool c,
        AsyncCallback cb, object state);
    public string EndInvoke(IAsyncResult result);
}
```

Делегаты также могут “указывать” на методы, содержащие любое количество параметров `out` и `ref` (равно как и параметров типа массивов, помеченных ключевым словом `params`). Например, предположим, что имеется следующий тип делегата:

```
public delegate string MyOtherDelegate(out bool a, ref bool b, int c);
```

Сигнатуры методов `Invoke()` и `BeginInvoke()` выглядят так, как и следовало ожидать; однако взгляните на метод `EndInvoke()`, который теперь включает набор аргументов `out/ref`, определенных типом делегата:

```
public sealed class MyOtherDelegate : System.MulticastDelegate
{
    public string Invoke(out bool a, ref bool b, int c);
    public IAsyncResult BeginInvoke(out bool a, ref bool b, int c,
        AsyncCallback cb, object state);
    public string EndInvoke(out bool a, ref bool b, IAsyncResult result);
}
```

Чтобы подытожить: определение типа делегата C# дает в результате запечатанный класс с тремя сгенерированными компилятором методами, типы параметров и возвращаемых значений которых основаны на объявлении делегата. Базовый шаблон может быть описан с помощью следующего псевдокода:

```
// Это только псевдокод!
public sealed class ИмяДелегата : System.MulticastDelegate
{
    public возвращаемоеЗначениеДелегата Invoke(всеВходныеRefиOutПараметрыДелегата);
    public IAsyncResult BeginInvoke(всеВходныеRefиOutПараметрыДелегата,
        AsyncCallback cb, object state);
    public возвращаемоеЗначениеДелегата EndInvoke(всеВходныеRefиOutПараметрыДелегата,
        IAsyncResult result);
}
```

Базовые классы `System.MulticastDelegate` и `System.Delegate`

При построении типа, использующего ключевое слово `delegate`, неявно объявляется тип класса, унаследованного от `System.MulticastDelegate`. Этот класс обеспечивает своим наследникам доступ к списку, который содержит адреса методов, поддерживаемых типом делегата, а также несколько дополнительных методов (и несколько перегруженных операций), чтобы взаимодействовать со списком вызовов.

Ниже приведены некоторые избранные методы `System.MulticastDelegate`:

```
public abstract class MulticastDelegate : Delegate
{
    // Возвращает список методов, на которые "указывает" делегат.
    public sealed override Delegate[] GetInvocationList();
    // Перегруженные операции.
    public static bool operator ==(MulticastDelegate d1, MulticastDelegate d2);
    public static bool operator !=(MulticastDelegate d1, MulticastDelegate d2);
    // Используются внутренне для управления списком методов, поддерживаемых делегатом.
    private IntPtr _invocationCount;
    private object _invocationList;
}
```

Класс `System.MulticastDelegate` получает дополнительную функциональность от своего родительского класса `System.Delegate`. Далее показан фрагмент определения класса:

```
public abstract class Delegate : ICloneable, ISerializable
{
    // Методы для взаимодействия со списком функций.
    public static Delegate Combine(params Delegate[] delegates);
    public static Delegate Combine(Delegate a, Delegate b);
    public static Delegate Remove(Delegate source, Delegate value);
    public static Delegate RemoveAll(Delegate source, Delegate value);
    // Перегруженные операции.
    public static bool operator ==(Delegate d1, Delegate d2);
    public static bool operator !=(Delegate d1, Delegate d2);
    // Свойства, показывающие цель делегата.
    public MethodInfo Method { get; }
    public object Target { get; }
}
```

Запомните, что вы никогда не сможете напрямую наследовать от этих базовых классов в коде (при попытке сделать это выдается ошибка компиляции). Тем не менее, при использовании ключевого слова `delegate` неявно создается класс, который "является" `MulticastDelegate`.

В табл. 10.1 описаны основные члены, общие для всех типов делегатов.

Таблица 10.1. Основные члены `System.MulticastDelegate/System.Delegate`

Член	Назначение
<code>Method</code>	Это свойство возвращает объект <code>System.Reflection.Method</code> , который представляет детали статического метода, поддерживаемого делегатом
<code>Target</code>	Если метод, подлежащий вызову, определен на уровне объекта (т.е. не является статическим), то <code>Target</code> возвращает объект, который представляет метод, поддерживаемый делегатом. Если возвращенное <code>Target</code> значение равно <code>null</code> , значит, подлежащий вызову метод является статическим
<code>Combine()</code>	Этот статический метод добавляет метод в список, поддерживаемый делегатом. В C# этот метод вызывается за счет использования перегруженной операции <code>+=</code> в качестве сокращенной нотации
<code>GetInvocationList()</code>	Этот метод возвращает массив типов <code>System.Delegate</code> , каждый из которых представляет определенный метод, доступный для вызова
<code>Remove()</code>	Эти статические методы удаляют метод (или все методы) из списка вызовов делегата. В C# метод <code>Remove()</code> может быть вызван неявно, посредством перегруженной операции <code>--</code>
<code>RemoveAll()</code>	

Пример простейшего делегата

При первоначальном знакомстве делегаты могут показаться несколько запутанными. Рассмотрим для начала очень простое консольное приложение под названием SimpleDelegate, в котором используется ранее показанный тип делегата BinaryOp. Ниже приведен полный код с последующим анализом.

```
namespace SimpleDelegate
{
    // Этот делегат может указывать на любой метод,
    // принимающий два целых и возвращающий целое.
    public delegate int BinaryOp(int x, int y);
    // Этот класс содержит методы, на которые будет указывать BinaryOp.
    public class SimpleMath
    {
        public static int Add(int x, int y)
        { return x + y; }
        public static int Subtract(int x, int y)
        { return x - y; }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** Simple Delegate Example *****\n");
            // Создать объект делегата BinaryOp, "указывающий" на SimpleMath.Add().
            BinaryOp b = new BinaryOp(SimpleMath.Add);
            // Вызвать метод Add() непрямо с использованием объекта делегата.
            Console.WriteLine("10 + 10 is {0}", b(10, 10));
            Console.ReadLine();
        }
    }
}
```

Обратите внимание на формат объявления типа делегата BinaryOp: он определяет, что объекты делегата BinaryOp могут указывать на любой метод, принимающий два целых и возвращающий целое (действительное имя метода, на который он указывает, не существенно). Здесь создан класс по имени SimpleMath, определяющий два статических метода, которые соответствуют шаблону, определенному делегатом BinaryOp.

Когда нужно вставить целевой метод в заданный объект делегата, просто передайте имя этого метода конструктору делегата:

```
// Создать делегат BinaryOp, который "указывает" на SimpleMath.Add().
BinaryOp b = new BinaryOp(SimpleMath.Add);
```

С этого момента указанный метод можно вызывать с использованием синтаксиса, который выглядит как прямой вызов функции:

```
// Invoke() действительно вызывается здесь!
Console.WriteLine("10 + 10 is {0}", b(10, 10));
```

“За кулисами” исполняющая среда на самом деле вызывает сгенерированный компилятором метод Invoke() на производном классе MulticastDelegate. В этом можно убедиться, открыв сборку в утилите ildasm.exe и просмотрев код CIL в методе Main():

```
.method private hidebysig static void Main(string[] args) cil managed
{
    ...
    callvirt instance int32 SimpleDelegate.BinaryOp::Invoke(int32, int32)
}
```

C# не требует явного вызова `Invoke()` в кодовой базе. Поскольку `BinaryOp` может указывать на методы, которые принимают два аргумента, следующий оператор кода является допустимым:

```
Console.WriteLine("10 + 10 is {0}", b.Invoke(10, 10));
```

Вспомните, что делегаты .NET безопасны в отношении типов. Поэтому попытка передать делегату метод, не соответствующий шаблону, приводит к ошибке на этапе компиляции. Чтобы проиллюстрировать это, предположим, что в классе `SimpleMath` теперь определен дополнительный метод по имени `SquareNumber()`, принимающий единственный целочисленный аргумент:

```
public class SimpleMath
{
    ...
    public static int SquareNumber(int a)
    { return a * a; }
}
```

Учитывая, что делегат `BinaryOp` может указывать только на методы, принимающие два целых и возвращающие целое, следующий код неверен и компилироваться не будет:

```
// Ошибка компиляции! Метод не соответствует шаблону делегата!
BinaryOp b2 = new BinaryOp(SimpleMath.SquareNumber);
```

Исследование объекта делегата

Давайте усложним текущий пример, создав статический метод (по имени `DisplayDelegateInfo()`) в классе `Program`. Этот метод будет выводить на консоль имена методов, поддерживаемых объектом делегата, а также имя класса, определяющего метод. Для этого будет реализована итерация по массиву `System.Delegate`, возвращенному `GetInvocationList()`, с обращением к свойствам `Target` и `Method` каждого объекта.

```
static void DisplayDelegateInfo(Delgate delObj)
{
    // Вывести на консоль имена каждого члена в списке вызовов делегата.
    foreach (Delegate d in delObj.GetInvocationList())
    {
        Console.WriteLine("Method Name: {0}", d.Method);    // Имя метода
        Console.WriteLine("Type Name: {0}", d.Target);      // Имя типа
    }
}
```

Исходя из предположения, что в метод `Main()` добавлен вызов этого вспомогательного метода:

```
BinaryOp b = new BinaryOp(SimpleMath.Add);
DisplayDelegateInfo(b);
```

вывод приложения будет выглядеть следующим образом:

```
***** Simple Delegate Example *****
Method Name: Int32 Add(Int32, Int32)
Type Name:
10 + 10 is 20
```

Обратите внимание, что имя целевого класса (`SimpleMath`) в настоящий момент не отображается при обращении к свойству `Target`. Причина в том, что делегат `BinaryOp` указывает на статический метод, и, таким образом, просто нет объекта, на который

нужно ссылаться! Однако если сделать методы `Add()` и `Substract()` нестатическими (удалив в их объявлениях ключевое слово `static`), можно будет создавать экземпляр типа `SimpleMath` и указывать методы для вызова с использованием ссылки на объект:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Simple Delegate Example *****\n");
    // Делегаты .NET могут указывать на методы экземпляра.
    SimpleMath m = new SimpleMath();
    BinaryOp b = new BinaryOp(m.Add);

    // Вывести сведения об объекте.
    DisplayDelegateInfo(b);
    Console.WriteLine("10 + 10 is {0}", b(10, 10));
    Console.ReadLine();
}
```

В этом случае вывод будет выглядеть, как показано ниже:

```
***** Simple Delegate Example *****
Method Name: Int32 Add(Int32, Int32)
Type Name: SimpleDelegate.SimpleMath
10 + 10 is 20
```

Исходный код. Проект `SimpleDelegate` доступен в подкаталоге `Chapter 10`.

Отправка уведомлений о состоянии объекта с использованием делегатов

Ясно, что предыдущий пример `SimpleDelegate` был предназначен только для целей иллюстрации, поскольку нет особого смысла создавать делегат только для того, чтобы просуммировать два числа. Рассмотрим более реалистичный пример, в котором делегаты используются для определения класса `Car`, обладающего способностью информировать внешние сущности о текущем состоянии двигателя. Для этого понадобится выполнить перечисленные ниже шаги.

1. Определить новый тип делегата, который будет отправлять уведомления вызывающему коду.
2. Объявить переменную-член этого типа делегата в классе `Car`.
3. Создать в классе `Car` вспомогательную функцию, которая позволяет вызывающему коду указывать метод для обратного вызова.
4. Реализовать метод `Accelerate()` для обращения к списку вызовов делегата при нужных условиях.

Для начала создадим проект консольного приложения по имени `CarDelegate`. Затем определим новый класс `Car`, который изначально выглядит следующим образом:

```
public class Car
{
    // Данные состояния.
    public int CurrentSpeed { get; set; }
    public int MaxSpeed { get; set; }
    public string PetName { get; set; }

    // Исправен ли автомобиль?
    private bool carIsDead;
```

```
// Конструкторы класса.
public Car() { MaxSpeed = 100; }
public Car(string name, int maxSp, int currSp)
{
    CurrentSpeed = currSp;
    MaxSpeed = maxSp;
    PetName = name;
}
}
```

Ниже показаны обновления, связанные с реализацией трех первых пунктов:

```
public class Car
{
    ...
    // 1. Определить тип делегата.
    public delegate void CarEngineHandler(string msgForCaller);
    // 2. Определить переменную-член типа этого делегата.
    private CarEngineHandler listOfHandlers;
    // 3. Добавить регистрационную функцию для вызывающего кода.
    public void RegisterWithCarEngine(CarEngineHandler methodToCall)
    {
        listOfHandlers = methodToCall;
    }
}
```

Обратите внимание, что в данном примере типы делегатов определяются непосредственно в контексте класса `Car`, что совершенно не обязательно, но помогает укрепить идею о том, что делегат работает естественным образом с этим отдельным классом. Наш тип делегата — `CarEngineHandler` — может указывать на любой метод, принимающий значение `string` в качестве параметра и имеющий `void` в качестве типа возврата.

Кроме того, была объявлена закрытая переменная-член делегата (по имени `listOfHandlers`) и вспомогательная функция (по имени `RegisterWithCarEngine()`), которая позволяет вызывающему коду добавлять метод к списку вызовов делегата.

На заметку! Строго говоря, можно было бы определить переменную-член делегата как `public`, избежав необходимости в добавлении дополнительных методов регистрации. Однако за счет определения этой переменной-члена делегата как `private` усиливается инкапсуляция и обеспечивается более безопасное к типам решение. Опасности объявления переменных-членов делегатов как `public` еще будут рассматриваться в этой главе, когда речь пойдет о ключевом слове `event` языка `C#`.

Теперь необходимо создать метод `Accelerate()`. Вспомните, что здесь стоит задача позволить объекту `Car` отправлять связанные с двигателем сообщения любому подписавшемуся слушателю. Ниже показаны необходимые изменения в коде.

```
// 4. Реализовать метод Accelerate() для обращения
// к списку вызовов делегата при нужных условиях.
public void Accelerate(int delta)
{
    // Если этот автомобиль сломан, отправить сообщение об этом.
    if (carIsDead)
    {
        if (listOfHandlers != null)
            listOfHandlers("Sorry, this car is dead...");
    }
    else
```

```

{
    CurrentSpeed += delta;
    // Автомобиль почти сломан?
    if (10 == (MaxSpeed - CurrentSpeed)
        && listOfHandlers != null)
    {
        listOfHandlers("Careful buddy! Gonna blow!");
    }
    if (CurrentSpeed >= MaxSpeed)
        carIsDead = true;
    else
        Console.WriteLine("CurrentSpeed = {0}", CurrentSpeed);
}
}

```

Обратите внимание, что прежде чем вызывать методы, поддерживаемые переменной-членом `listOfHandlers`, она проверяется на равенство `null`. Причина в том, что размещение этих объектов вызовом вспомогательного метода `RegisterWithCarEngine()` является задачей вызывающего кода. Если вызывающий код не вызовет этот метод, а мы попытаемся обратиться к списку вызовов делегата, то получим исключение `NullReferenceException` во время выполнения. Теперь, имея всю инфраструктуру делегатов, рассмотрим обновления класса `Program`:

```

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Delegates as event enablers *****\n");

        // Сначала создать объект Car.
        Car c1 = new Car("SlugBug", 100, 10);

        // Теперь сообщить ему, какой метод вызывать,
        // когда он захочет отправить сообщение.
        c1.RegisterWithCarEngine(new Car.CarEngineHandler(OnCarEngineEvent));

        // Ускорить (это иницирует события).
        Console.WriteLine("***** Speeding up *****");
        for (int i = 0; i < 6; i++)
            c1.Accelerate(20);
        Console.ReadLine();
    }

    // Это цель для входящих сообщений.
    public static void OnCarEngineEvent(string msg)
    {
        Console.WriteLine("\n***** Message From Car Object *****");
        Console.WriteLine("=> {0}", msg);
        Console.WriteLine("*****\n");
    }
}

```

Метод `Main()` начинается с создания нового объекта `Car`. Поскольку мы заинтересованы в событиях, связанных с двигателем, следующий шаг заключается в вызове специальной регистрационной функции `RegisterWithCarEngine()`. Вспомните, что этот метод ожидает получения экземпляра вложенного делегата `CarEngineHandler`, и как с любым делегатом, метод, на который он должен указывать, задается в параметре конструктора. Трюк в этом примере заключается в том, что интересующий метод находится в классе `Program`! Обратите внимание, что метод `OnCarEngineEvent()` полностью соответствует связанному делегату в том, что принимает `string` и возвращает `void`.

Ниже показан вывод этого примера:

```
***** Delegates as event enablers *****

***** Speeding up *****
CurrentSpeed = 30
CurrentSpeed = 50
CurrentSpeed = 70

***** Message From Car Object *****
=> Careful buddy! Gonna blow!
*****

CurrentSpeed = 90

***** Message From Car Object *****
=> Sorry, this car is dead...
*****
```

Включение группового вызова

Вспомните, что делегаты .NET обладают встроенной возможностью *группового вызова*. Другими словами, объект делегата может поддерживать целый список методов для вызова, а не просто единственный метод. Для добавления нескольких методов к объекту делегата используется перегруженная операция `+=`, а не прямое присваивание. Чтобы включить групповой вызов в типе `Car`, можно модифицировать метод `RegisterWithCarEngine()` следующим образом:

```
public class Car
{
    // Добавление поддержки группового вызова.
    // Обратите внимание на использование операции +=, а не операции присваивания (=).
    public void RegisterWithCarEngine(CarEngineHandler methodToCall)
    {
        listOfHandlers += methodToCall;
    }
    ...
}
```

Когда операция `+=` используется с объектом делегата, компилятор преобразует это в вызов статического метода `Delegate.Combine()`. В действительности можно было бы вызвать `Delegate.Combine()` напрямую; однако операция `+=` предлагает более простую альтернативу. Нет никакой необходимости в модификации текущего метода `RegisterWithCarEngine()`, но ниже представлен пример применения `Delegate.Combine()` вместо операции `+=`:

```
public void RegisterWithCarEngine( CarEngineHandler methodToCall )
{
    if (listOfHandlers == null)
        listOfHandlers = methodToCall;
    else
        Delegate.Combine(listOfHandlers, methodToCall);
}
```

В любом случае вызывающий код может теперь регистрировать множественные цели для одного и того же обратного вызова. Здесь второй обработчик выводит входное сообщение в верхнем регистре, просто в целях отображения:

```
class Program
{
    static void Main(string[] args)
    {
```

```

Console.WriteLine("***** Delegates as event enablers *****\n");
Car c1 = new Car("SlugBug", 100, 10);

// Зарегистрировать несколько обработчиков событий.
c1.RegisterWithCarEngine(new Car.CarEngineHandler(OnCarEngineEvent));
c1.RegisterWithCarEngine(new Car.CarEngineHandler(OnCarEngineEvent2));

// Ускорить (это инициирует события).
Console.WriteLine("***** Speeding up *****");
for (int i = 0; i < 6; i++)
    c1.Accelerate(20);
Console.ReadLine();
}

// Теперь есть ДВА метода, которые будут вызваны Car
// при отправке уведомлений.
public static void OnCarEngineEvent(string msg)
{
    Console.WriteLine("\n***** Message From Car Object *****");
    Console.WriteLine("=> {0}", msg);
    Console.WriteLine("*****\n");
}

public static void OnCarEngineEvent2(string msg)
{
    Console.WriteLine("=> {0}", msg.ToUpper());
}
}

```

Удаление целей из списка вызовов делегата

В классе `Delegate` также определен метод `Remove()`, позволяющий вызывающему коду динамически удалять отдельные члены из списка вызовов объекта делегата. Это позволяет вызывающему коду легко "отписываться" от определенного уведомления во время выполнения. Хотя в коде можно непосредственно вызывать `Delegate.Remove()`, разработчики C# могут использовать также перегруженную операцию `--` в качестве сокращения. Давайте добавим в класс `Car` новый метод, который позволяет вызывающему коду исключать метод из списка вызовов:

```

public class Car
{
    ...
    public void UnRegisterWithCarEngine(CarEngineHandler methodToCall)
    {
        listOfHandlers -= methodToCall;
    }
}

```

В текущей версии класса `Car` прекратить получение уведомлений от второго обработчика можно за счет изменения метода `Main()` следующим образом:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Delegates as event enablers *****\n");

    // Сначала создать объект Car.
    Car c1 = new Car("SlugBug", 100, 10);
    c1.RegisterWithCarEngine(new Car.CarEngineHandler(OnCarEngineEvent));

    // На этот раз сохранить объект делегата для последующей отмены регистрации.
    Car.CarEngineHandler handler2 = new Car.CarEngineHandler(OnCarEngineEvent2);
    c1.RegisterWithCarEngine(handler2);

    ...
}

```

```
// Ускорить (это инициирует события).
Console.WriteLine("***** Speeding up *****");
for (int i = 0; i < 6; i++)
    cl.Accelerate(20);

// Отменить регистрацию второго обработчика.
cl.UnRegisterWithCarEngine(handler2);

// Сообщения в верхнем регистре больше не выводятся.
Console.WriteLine("***** Speeding up *****");
for (int i = 0; i < 6; i++)
    cl.Accelerate(20);

Console.ReadLine();
}
```

Одно отличие `Main()` заключается в том, что на этот раз создается объект `Car.CarEngineHandler`, который сохраняется в локальной переменной, чтобы иметь возможность позднее отменить подписку на получение уведомлений. Тогда при следующем ускорении `Car` уже больше не будет выводиться версия входящего сообщения в верхнем регистре, поскольку эта цель исключена из списка вызовов делегата.

Исходный код. Проект `CarDelegate` доступен в подкаталоге `Chapter 10`.

Синтаксис групповых преобразований методов

В предыдущем примере `CarDelegate` явно создавались экземпляры объекта делегата `Car.CarEngineHandler`, чтобы регистрировать и отменять регистрацию на получение уведомлений:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Delegates as event enablers *****\n");
    Car c1 = new Car("SlugBug", 100, 10);
    c1.RegisterWithCarEngine(new Car.CarEngineHandler(OnCarEngineEvent));

    Car.CarEngineHandler handler2 =
        new Car.CarEngineHandler(OnCarEngineEvent2);
    c1.RegisterWithCarEngine(handler2);
    ...
}
```

Конечно, если нужно вызывать любые унаследованные члены `MulticastDelegate` или `Delegate`, то наиболее простым способом сделать это будет ручное создание переменной делегата. Однако в большинстве случаев обращаться к внутренностям объекта делегата не понадобится. Объект делегата будет нужен только для того, чтобы передать имя метода как параметр конструктора.

Для простоты в C# предлагается сокращение, которое называется *групповым преобразованием методов*. Это средство позволяет указывать прямое имя метода, а не объект делегата, когда вызываются методы, принимающие делегаты в качестве аргументов.

На заметку! Как будет показано далее в этой главе, синтаксис группового преобразования методов можно также использовать для упрощения регистрации событий C#.

В целях иллюстрации создадим новое консольное приложение по имени `CarDelegate` `MethodGroupConversion` и добавим в него файл, содержащий класс `Car`, который был определен в проекте `CarDelegate` (и обновим название пространства имен в файле

Car.cs для соответствия новому названию). В показанном ниже коде класса Program используется групповое преобразование методов для регистрации и отмены регистрации подписки на уведомления.

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Method Group Conversion *****\n");
        Car c1 = new Car();

        // Зарегистрировать простое имя метода.
        c1.RegisterWithCarEngine(CallMeHere);

        Console.WriteLine("***** Speeding up *****");
        for (int i = 0; i < 6; i++)
            c1.Accelerate(20);

        // Отменить регистрацию простого имени метода.
        c1.UnRegisterWithCarEngine(CallMeHere);

        // Уведомления больше не поступают!
        for (int i = 0; i < 6; i++)
            c1.Accelerate(20);
        Console.ReadLine();
    }

    static void CallMeHere(string msg)
    {
        Console.WriteLine("=> Message from Car: {0}", msg);
    }
}
```

Обратите внимание, что мы не создаем непосредственно объект делегата, а просто указываем метод, который соответствует ожидаемой сигнатуре делегата (в данном случае — метод, возвращающий void и принимающий единственный аргумент string). Имейте в виду, что компилятор C# по-прежнему обеспечивает безопасность типов. Поэтому, если метод CallMeHere() не принимает string и не возвращает void, компилятор сообщит об ошибке.

Исходный код. Проект CarDelegateMethodGroupConversion доступен в подкаталоге Chapter 10.

Понятие обобщенных делегатов

В предыдущей главе упоминалось, что язык C# позволяет определять обобщенные типы делегатов. Например, предположим, что необходимо определить делегат, который может вызывать любой метод, возвращающий void и принимающий единственный параметр. Если передаваемый аргумент может изменяться, это моделируется через параметр типа. Для иллюстрации рассмотрим следующий код нового консольного приложения по имени GenericDelegate:

```
namespace GenericDelegate
{
    // Этот обобщенный делегат может вызывать любой метод, который
    // возвращает void и принимает единственный параметр типа.
    public delegate void MyGenericDelegate<T>(T arg);

    class Program
    {
```

```

static void Main(string[] args)
{
    Console.WriteLine("***** Generic Delegates *****\n");
    // Зарегистрировать цели.
    MyGenericDelegate<string> strTarget =
        new MyGenericDelegate<string>(StringTarget);
    strTarget("Some string data");

    MyGenericDelegate<int> intTarget =
        new MyGenericDelegate<int>(IntTarget);
    intTarget(9);
    Console.ReadLine();
}

static void StringTarget(string arg)
{
    Console.WriteLine("arg in uppercase is: {0}", arg.ToUpper());
}

static void IntTarget(int arg)
{
    Console.WriteLine("++arg is: {0}", ++arg);
}
}

```

Обратите внимание, что в `MyGenericDelegate<T>` определен единственный параметр, представляющий аргумент для передачи цели делегата. При создании экземпляра этого типа необходимо указать значение параметра типа вместе с именем метода, который может вызывать делегат. Таким образом, если указать тип `string`, целевому методу будет отправлено строковое значение:

```

// Создать экземпляр MyGenericDelegate<T>
// со string в качестве параметра типа.
MyGenericDelegate<string> strTarget =
    new MyGenericDelegate<string>(StringTarget);
strTarget("Some string data");

```

Имея формат объекта `strTarget`, метод `StringTarget` теперь должен принимать в качестве параметра единственную строку:

```

static void StringTarget(string arg)
{
    Console.WriteLine("arg in uppercase is: {0}", arg.ToUpper());
}

```

Исходный код. Проект `GenericDelegate` доступен в подкаталоге `Chapter 10`.

Обобщенные делегаты `Action<>` и `Func<>`

На протяжении этой главы вы видели, что когда необходимо использовать делегаты для включения обратных вызовов в приложениях, обычно выполнялись следующие шаги:

- определение специального делегата, соответствующего формату метода, на который он указывает;
- создание экземпляра специального делегата с передачей имени метода в качестве аргумента конструктора;
- косвенное обращение к методу через вызов `Invoke()` на объекте делегата.

В случае принятия такого подхода, как правило, в конечном итоге получается несколько специальных делегатов, которые никогда не могут применяться за пределами текущей задачи (например, `MyGenericDelegate<T>`, `CarEngineHandler` и т.д.). Хотя может случаться так, что в проекте требуется специальный, уникально именованный делегат, в других ситуациях точное имя делегата является несущественным. Во многих случаях необходим просто “некоторый делегат”, принимающий набор аргументов и возможно возвращающий значение, отличное от `void`. В таких ситуациях можно воспользоваться встроенными в платформу делегатами `Action<>` и `Func<>`. Для иллюстрации их удобства создадим проект консольного приложения по имени `ActionAndFuncDelegates`.

Обобщенный делегат `Action<>` определен в пространствах имен `System` внутри сборки `mscorlib.dll` и `System.Core.dll`. Этот обобщенный делегат можно применять для “указания на” метод, который принимает вплоть до 16 аргументов (чего должно быть достаточно!) и возвращает `void`. Вспомните, что поскольку `Action<>` является обобщенным делегатом, понадобится также указывать типы каждого параметра.

Модифицируйте код класса `Program`, определив новый статический метод, который принимает три (или около того) уникальных параметра, например:

```
// Это цель для делегата Action<>.
static void DisplayMessage(string msg, ConsoleColor txtColor, int printCount)
{
    // Установить цвет текста консоли.
    ConsoleColor previous = Console.ForegroundColor;
    Console.ForegroundColor = txtColor;

    for (int i = 0; i < printCount; i++)
    {
        Console.WriteLine(msg);
    }

    // Восстановить цвет.
    Console.ForegroundColor = previous;
}
```

Теперь вместо построения специального делегата вручную для передачи потока программы методу `DisplayMessage()` мы можем использовать готовый делегат `Action<>`, как показано ниже:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Action and Func *****");

    // Использовать делегат Action<> для указания на DisplayMessage.
    Action<string, ConsoleColor, int> actionTarget =
        new Action<string, ConsoleColor, int>(DisplayMessage);
    actionTarget("Action Message!", ConsoleColor.Yellow, 5);

    Console.ReadLine();
}
```

Как видите, применение делегата `Action<>` не заставляет беспокоиться об определении специального делегата. Однако вспомните, что делегат `Action<>` может указывать только на методы, которые имеют возвращаемое значение `void`. Если нужно указывать на метод с другим возвращаемым значением (и нет желания заниматься написанием собственного делегата), можно прибегнуть к делегату `Func<>`.

Обобщенный делегат `Func<>` может указывать на методы, которые (подобно `Action<>`) принимают вплоть до 16 параметров и имеют специальное возвращаемое значение. В целях иллюстрации добавим следующий новый метод в класс `Program`:

```
// Цель для делегата Func<>.
static int Add(int x, int y)
```

```
{
    return x + y;
}
```

Ранее в этой главе был построен специальный делегат `BinaryOp` для “указания на” методы сложения и вычитания. Теперь можно упростить задачу, воспользовавшись версией `Func<>`, которая принимает всего три параметра типа. Учтите, что *финальный* параметр типа `Func<>` — это *всегда* возвращаемое значение метода. Чтобы закрепить этот момент, предположим, что в классе `Program` также определен следующий метод:

```
static string SumToString(int x, int y)
{
    return (x + y).ToString();
}
```

Теперь метод `Main()` может вызывать каждый из этих методов, как показано ниже:

```
Func<int, int, int> funcTarget = new Func<int, int, int>(Add);
int result = funcTarget.Invoke(40, 40);
Console.WriteLine("40 + 40 = {0}", result);

Func<int, int, string> funcTarget2 = new Func<int, int, string>(SumToString);
string sum = funcTarget2(90, 300);
Console.WriteLine(sum);
```

Таким образом, учитывая, что делегаты `Action<>` и `Func<>` могут устранить шаг ручному определению специального делегата, вас может интересовать, следует ли ими пользоваться всегда. Ответом будет, как и в случае многих аспектов программирования — в зависимости от ситуации. Во многих случаях `Action<>` и `Func<>` будут предпочтительным вариантом. Тем не менее, если нужен делегат со специфическим именем, которое, как вы чувствуете, помогает лучше отразить предметную область, то построение специального делегата сведется к одиночному оператору кода. В оставшейся части главы будут продемонстрированы оба подхода.

На заметку! Делегаты `Action<>` и `Func<>` интенсивно используются во многих важных API-интерфейсах .NET, включая инфраструктуру параллельного программирования и LINQ (помимо прочих).

На этом первоначальный экскурс в тип делегата .NET завершен. Мы еще вернемся к некоторым дополнительным деталям работы с делегатами в конце этой главы и еще раз — в главе 19, когда будем рассматривать многопоточность и асинхронные вызовы. А теперь переходим к связанной теме — ключевому слову `event` языка C#.

Исходный код. Проект `ActionAndFuncDelegates` доступен в подкаталоге `Chapter 10`.

Понятие событий C#

Делегаты — весьма интересные конструкции в том смысле, что позволяют объектам, находящимся в памяти, участвовать в двустороннем общении. Однако работа с делегатами напрямую может порождать довольно однообразный код (определение делегата, определение необходимых переменных-членов и создание специальных методов регистрации и отмены регистрации для предохранения инкапсуляции).

Более того, если делегаты используются в качестве механизма обратного вызова в приложениях напрямую, существует еще одна проблема: если не определить переменную-член типа делегата в классе как закрытую, то вызывающий код получит прямой доступ к объектам делегатов. В этом случае вызывающий код может присвоить перемен-

ной новый объект-делегат (фактически удалив текущий список функций, подлежащих вызову), и что еще хуже — вызывающий код сможет напрямую обращаться к списку вызовов делегата. Чтобы проиллюстрировать проблему, рассмотрим следующую переделанную (и упрощенную) версию класса Car из предыдущего примера CarDelegate:

```
public class Car
{
    public delegate void CarEngineHandler(string msgForCaller);

    // Теперь это член public!
    public CarEngineHandler listOfHandlers;

    // Просто вызвать уведомление Exploded.
    public void Accelerate(int delta)
    {
        if (listOfHandlers != null)
            listOfHandlers("Sorry, this car is dead...");
    }
}
```

Обратите внимание, что теперь больше нет закрытых переменных-членов с типами делегатов, инкапсулированных с помощью специальных методов регистрации. Поскольку эти члены сделаны открытыми, вызывающий код может непосредственно обращаться к переменной-члену listOfHandlers и присвоить ей новые объекты CarEngineHandler, после чего вызывать делегат, когда вздумается:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Agh! No Encapsulation! *****\n");

        // Создать Car.
        Car myCar = new Car();

        // Есть прямой доступ к делегату!
        myCar.listOfHandlers = new Car.CarEngineHandler(CallWhenExploded);
        myCar.Accelerate(10);

        // Присвоить ему совершенно новый объект...
        // В лучшем случае получается путаница.
        myCar.listOfHandlers = new Car.CarEngineHandler(CallHereToo);
        myCar.Accelerate(10);

        // Вызывающий код может также напрямую вызывать делегат!
        myCar.listOfHandlers.Invoke("hee, hee, hee...");
        Console.ReadLine();
    }

    static void CallWhenExploded(string msg)
    { Console.WriteLine(msg); }

    static void CallHereToo(string msg)
    { Console.WriteLine(msg); }
}
```

Открытие членов-делегатов нарушает инкапсуляцию, что не только затруднит сопровождение кода (и отладку), но также сделает приложение уязвимым в смысле безопасности! Ниже показан вывод текущего примера:

```
***** Agh! No Encapsulation! *****
Sorry, this car is dead...
Sorry, this car is dead...
hee, hee, hee...
```

Очевидно, что не следует предоставлять другим приложениям право изменять то, на что указывает делегат, или вызывать его члены напрямую. С учетом этого, общепринятой практикой является объявление переменных-членов с типами делегатов закрытыми.

Исходный код. Проект `PublicDelegateProblem` доступен в подкаталоге `Chapter 10`.

Ключевое слово `event`

В качестве сокращения, избавляющего от необходимости строить специальные методы для добавления и удаления методов из списка вызовов делегата, в C# предусмотрено ключевое слово `event`. Обработка компилятором ключевого слова `event` приводит к автоматическому получению методов регистрации и отмены регистрации наряду со всеми необходимыми переменными-членами для типов делегатов. Эти переменные-члены типов делегатов *всегда* объявляются закрытыми и, следовательно, они не доступны напрямую из объекта, инициировавшего событие. Таким образом, ключевое слово `event` может применяться для упрощения отправки уведомлений из специального класса внешним объектам.

Определение события представляет собой двухэтапный процесс. Во-первых, нужно определить делегат, который будет хранить список методов, подлежащих вызову при возникновении события. Во-вторых, необходимо объявить событие (используя ключевое слово `event`) в терминах связанного типа делегата.

Чтобы проиллюстрировать использование ключевого слова `event`, создадим новое консольное приложение по имени `CarEvents`. В классе `Car` будут определены два события под названиями `AboutToBlow` и `Exploded`. Эти события ассоциированы с единственным типом делегата по имени `CarEngineHandler`. Ниже показаны начальные изменения в классе `Car`:

```
public class Car
{
    // Этот делегат работает в сочетании с событиями Car.
    public delegate void CarEngineHandler(string msg);

    // Car может посылать следующие события:
    public event CarEngineHandler Exploded;
    public event CarEngineHandler AboutToBlow;
    ...
}
```

Отправка события вызываемому коду состоит просто в указании имени события вместе со всеми необходимыми параметрами, определенными в ассоциированном делегате. Чтобы удостовериться, что вызывающий код действительно зарегистрировал событие, его следует проверить на равенство `null` перед вызовом набора методов делегата. Ниже приведена новая версия метода `Accelerate()` класса `Car`:

```
public void Accelerate(int delta)
{
    // Если автомобиль сломан, инициировать событие Exploded.
    if (carIsDead)
    {
        if (Exploded != null)
            Exploded("Sorry, this car is dead...");
    }
    else
    {
        CurrentSpeed += delta;
    }
}
```

```

// Почти сломан?
if (10 == MaxSpeed - CurrentSpeed
    && AboutToBlow != null)
{
    AboutToBlow("Careful buddy! Gonna blow!");
}
// Все в порядке!
if (CurrentSpeed >= maxSpeed)
    carIsDead = true;
else
    Console.WriteLine("CurrentSpeed = {0}", CurrentSpeed);
}
}

```

Итак, объект `Car` сконфигурирован для отправки двух специальных событий без необходимости в определении специальных функций регистрации или объявлении переменных-членов с типами делегатов. Чуть ниже будет продемонстрировано применение этого нового объекта, но сначала давайте рассмотрим архитектуру событий немного подробнее.

“За кулисами” событий

Когда компилятор обрабатывает ключевое слово `event` языка C#, он генерирует два скрытых метода, один из которых имеет префикс `add_`, а другой — `remove_`. За префиксом следует имя события C#. Например, событие `Exploded` превращается в два скрытых метода с именами `add_Exploded()` и `remove_Exploded()`. Если заглянуть в CIL-код метода `add_AboutToBlow()`, можно обнаружить там вызов метода `Delegate.Combine()`. Ниже показан частичный код CIL:

```

.method public hidebysig specialname instance void
add_AboutToBlow(class CarEvents.Car/CarEngineHandler 'value') cil managed
{
    ...
    call class [mscorlib]System.Delegate
    [mscorlib]System.Delegate::Combine(
        class [mscorlib]System.Delegate, class [mscorlib]System.Delegate)
    ...
}

```

Как и можно было ожидать, `remove_AboutToBlow()` будет вызывать `Delegate.Remove()`:

```

.method public hidebysig specialname instance void
remove_AboutToBlow(class CarEvents.Car/CarEngineHandler 'value')
    cil managed
{
    ...
    call class [mscorlib]System.Delegate
    [mscorlib]System.Delegate::Remove(
        class [mscorlib]System.Delegate, class [mscorlib]System.Delegate)
    ...
}

```

И, наконец, в CIL-коде, представляющем само событие, используются директивы `.addon` и `.removeon` с целью отображения на имена корректных методов `add_XXX()` и `remove_XXX()` для вызова:

```

.event CarEvents.Car/EngineHandler AboutToBlow
{
    .addon instance void CarEvents.Car::add_AboutToBlow
        (class CarEvents.Car/CarEngineHandler)

```

```
.removeon instance void CarEvents.Car::remove_AboutToBlow
(class CarEvents.Car/CarEngineHandler)
}
```

Теперь, когда вы разобрались, как строить класс, способный отправлять события C# (и уже знаете, что события — это лишь способ сэкономить время на наборе кода), следующий большой вопрос связан с организацией прослушивания входящих событий на стороне вызывающего кода.

Прослушивание входящих событий

События C# также упрощают акт регистрации обработчиков событий на стороне вызывающего кода. Вместо того чтобы указывать специальные вспомогательные методы, вызывающий код просто использует операции `+=` и `-=` непосредственно (что приводит к внутренним вызовам методов `add_XXX()` или `remove_XXX()`). Для регистрации события руководствуйтесь показанным ниже шаблоном:

```
// ИмяОбъекта.ИмяСобытия += new СвязанныйДелегат(функцияДляВызова);
//
Car.CarEngineHandler d = new Car.CarEngineHandler(CarExplodedEventHandler);
myCar.Exploded += d;
```

Для отключения от источника событий служит операция `-=` в соответствии со следующим шаблоном:

```
// ИмяОбъекта.ИмяСобытия -= new СвязанныйДелегат(функцияДляВызова);
//
myCar.Exploded -= d;
```

Следуя этому очень простому шаблону, переделаем метод `Main()`, применив на этот раз синтаксис регистрации методов C#:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with Events *****\n");
        Car c1 = new Car("SlugBug", 100, 10);

        // Зарегистрировать обработчики событий.
        c1.AboutToBlow += new Car.CarEngineHandler(CarIsAlmostDoomed);
        c1.AboutToBlow += new Car.CarEngineHandler(CarAboutToBlow);

        Car.CarEngineHandler d = new Car.CarEngineHandler(CarExploded);
        c1.Exploded += d;

        Console.WriteLine("***** Speeding up *****");
        for (int i = 0; i < 6; i++)
            c1.Accelerate(20);

        // Удалить метод CarExploded из списка вызовов.
        c1.Exploded -= d;

        Console.WriteLine("\n***** Speeding up *****");
        for (int i = 0; i < 6; i++)
            c1.Accelerate(20);
        Console.ReadLine();
    }

    public static void CarAboutToBlow(string msg)
    { Console.WriteLine(msg); }

    public static void CarIsAlmostDoomed(string msg)
    { Console.WriteLine("=> Critical Message from Car: {0}", msg); }
```



```
public static void CarExploded(string msg)
{ Console.WriteLine(msg); }
}
```

Чтобы еще более упростить регистрацию событий, можно воспользоваться групповым преобразованием методов. Ниже показана очередная модификация Main().

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Events *****\n");
    Car cl = new Car("SlugBug", 100, 10);
    // Регистрация обработчиков событий.
    cl.AboutToBlow += CarIsAlmostDoomed;
    cl.AboutToBlow += CarAboutToBlow;
    cl.Exploded += CarExploded;
    Console.WriteLine("***** Speeding up *****");
    for (int i = 0; i < 6; i++)
        cl.Accelerate(20);
    cl.Exploded -= CarExploded;
    Console.WriteLine("\n***** Speeding up *****");
    for (int i = 0; i < 6; i++)
        cl.Accelerate(20);
    Console.ReadLine();
}
```

Упрощенная регистрация событий с использованием Visual Studio

Среда Visual Studio предоставляет помощь в процессе регистрации обработчиков событий. В случае применения синтаксиса += во время регистрации событий открывается окно IntelliSense, приглашающее нажать клавишу <Tab> для автоматического заполнения экземпляра делегата (рис. 10.2), которое получено с использованием синтаксиса групповых преобразований методов.

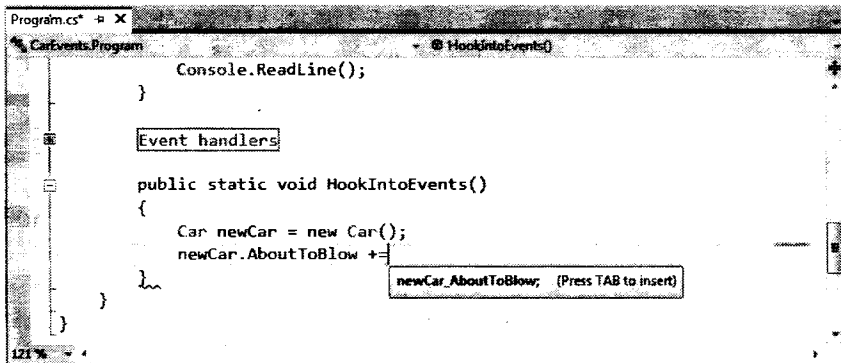


Рис. 10.2. Выбор делегата с помощью IntelliSense

После нажатия клавиши <Tab> появляется возможность ввести имя обработчика событий, который нужно сгенерировать (или просто принять стандартное имя), как показано на рис. 10.3.

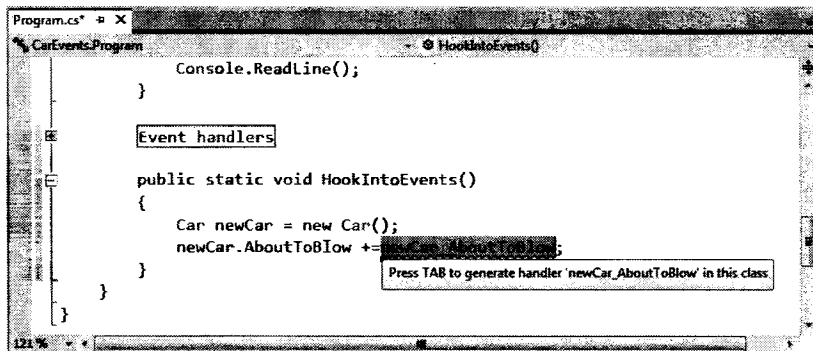


Рис. 10.3. Формат цели делегата IntelliSense

Снова нажав <Tab>, вы получите заготовку кода цели делегата в корректном формате (обратите внимание, что этот метод объявлен статическим, потому что событие было зарегистрировано внутри статического метода):

```
static void newCar_AboutToBlow(string msg)
{
    // Удалите следующую строку и добавьте свой код!
    throw new NotImplementedException();
}
```

Средство IntelliSense доступно для всех событий .NET из библиотек базовых классов. Это средство интегрированной среды разработки замечательно экономит время, но не избавляет от необходимости поиска в справочной системе .NET правильного делегата для использования с определенным событием, а также формата целевого метода делегата.

Исходный код. Проект CarEvents доступен в подкаталоге Chapter 10.

Создание специальных аргументов событий

По правде говоря, существует еще одно последнее усовершенствование, которое можно внести в текущую итерацию класса Car и которое отражает рекомендованный Microsoft шаблон событий. Если вы начнете исследовать события, отправляемые определенным типом из библиотек базовых классов, то обнаружите, что первым параметром лежащего в основе делегата будет System.Object, в то время как вторым параметром — тип, являющийся потомком System.EventArgs.

Аргумент System.Object представляет ссылку на объект, который отправляет событие (такой как Car), а второй параметр — информацию, относящуюся к обрабатываемому событию. Базовый класс System.EventArgs представляет событие, которое не посылает никакой специальной информации:

```
public class EventArgs
{
    public static readonly EventArgs Empty;
    public EventArgs();
}
```

Для простых событий можно передать экземпляр EventArgs непосредственно. Однако чтобы передавать какие-то специальные данные, потребуется построить подходящий класс, унаследованный от EventArgs. Для примера предположим, что есть

класс по имени CarEventArgs, поддерживающий строковое представление сообщения, отправленного получателю:

```
public class CarEventArgs : EventArgs
{
    public readonly string msg;
    public CarEventArgs(string message)
    {
        msg = message;
    }
}
```

Теперь понадобится модифицировать делегат CarEngineHandler, как показано ниже (само событие не изменяется):

```
public class Car
{
    public delegate void CarEngineHandler(object sender, CarEventArgs e);
    ...
}
```

Здесь при инициировании события из метода Accelerate() нужно использовать ссылку на текущий Car (через ключевое слово this) и экземпляр типа CarEventArgs. Например, рассмотрим следующее обновление:

```
public void Accelerate(int delta)
{
    // Если этот автомобиль сломан, инициировать событие Exploded.
    if (carIsDead)
    {
        if (Exploded != null)
            Exploded(this, new CarEventArgs("Sorry, this car is dead..."));
    }
    ...
}
```

Все, что потребуется сделать на вызывающей стороне — это обновить обработчики событий для получения входных параметров и получения сообщения через поле, доступное только для чтения. Например:

```
public static void CarAboutToBlow(object sender, CarEventArgs e)
{
    Console.WriteLine("{0} says: {1}", sender, e.msg);
}
```

Если получатель желает взаимодействовать с объектом, отправившим событие, можно выполнить явное приведение System.Object. С помощью такой ссылки можно вызвать любой открытый метод объекта, который отправил событие:

```
public static void CarAboutToBlow(object sender, CarEventArgs e)
{
    // Чтобы подстраховаться, произведем проверку
    // во время выполнения перед приведением.
    if (sender is Car)
    {
        Car c = (Car)sender;
        Console.WriteLine("Critical Message from {0}: {1}", c.PetName, e.msg);
    }
}
```

Обобщенный делегат EventHandler<T>

Учитывая, что очень многие специальные делегаты принимают object в первом параметре и наследника EventArgs — во втором, можно еще более упростить предыдущий пример, используя обобщенный тип EventHandler<T>, где T — специальный тип-наследник EventArgs. Рассмотрим следующую модификацию типа Car (обратите внимание, что строить специальный делегат больше не нужно):

```
public class Car
{
    public event EventHandler<CarEventArgs> Exploded;
    public event EventHandler<CarEventArgs> AboutToBlow;
    ...
}
```

Метод Main() может затем применять EventHandler<CarEventArgs> везде, где ранее указывался CarEventHandler:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Prim and Proper Events *****\n");
    // Создать Car обычным образом.
    Car c1 = new Car("SlugBug", 100, 10);
    // Зарегистрировать обработчики событий.
    c1.AboutToBlow += CarIsAlmostDoomed;
    c1.AboutToBlow += CarAboutToBlow;

    EventHandler<CarEventArgs> d = new EventHandler<CarEventArgs>(CarExploded);
    c1.Exploded += d;
    ...
}
```

Итак, вы ознакомились с основными аспектами работы с делегатами и событиями на языке C#. Хотя этого вполне достаточно для решения практически любых задач, связанных с обратными вызовами, в завершение главы рассмотрим ряд финальных упрощений, а именно — анонимные методы и лямбда-выражения.

Исходный код. Проект GenericPrimAndProperCarEvents доступен в подкаталоге Chapter 10.

Понятие анонимных методов C#

Как было показано выше, когда вызывающий код желает прослушивать входящие события, он должен определить специальный метод в классе (или структуре), соответствующий сигнатуре ассоциированного делегата. Ниже приведен пример:

```
class Program
{
    static void Main(string[] args)
    {
        SomeType t = new SomeType();
        // Предположим, что SomeDelegate может указывать на методы,
        // которые не принимают аргументов и возвращают void.
        t.SomeEvent += new SomeDelegate(MyEventHandler);
    }
}
```

```
// Обычно вызывается только объектом SomeDelegate.
public static void MyEventHandler()
{
    // Что-то делать по возникновении события.
}
}
```

Однако если подумать, то такие методы, как `MyEventHandler()`, редко предназначены для обращения из любой другой части программы помимо делегата. Учитывая продуктивность, несложно вручную определить отдельный метод для вызова объектом делегата. Для решения этой проблемы можно ассоциировать событие непосредственно с блоком операторов кода во время регистрации события. Формально такой код называется *анонимным методом*. Для иллюстрации синтаксиса напомним метод `Main()`, который обрабатывает события, посланные из типа `Car`, с использованием анонимных методов вместо специальных именованных обработчиков событий:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Anonymous Methods *****\n");
        Car c1 = new Car("SlugBug", 100, 10);

        // Зарегистрировать обработчики событий в виде анонимных методов.
        c1.AboutToBlow += delegate {
            Console.WriteLine("Eek! Going too fast!");
        };

        c1.AboutToBlow += delegate(object sender, CarEventArgs e) {
            Console.WriteLine("Message from Car: {0}", e.msg);
        };

        c1.Exploded += delegate(object sender, CarEventArgs e) {
            Console.WriteLine("Fatal Message from Car: {0}", e.msg);
        };

        // Это в конечном итоге инициирует события.
        for (int i = 0; i < 6; i++)
            c1.Accelerate(20);
        Console.ReadLine();
    }
}
```

На заметку! Последняя фигурная скобка анонимного метода должна завершаться точкой с запятой. Если забыть об этом, на этапе компиляции возникнет ошибка.

Обратите внимание, что в классе `Program` теперь не определяются специальные статические обработчики событий вроде `CarAboutToBlow()` или `CarExploded()`. Вместо этого с помощью синтаксиса `+=` определяются встроенные неименованные (анонимные) методы, к которым вызывающий код будет обращаться во время обработки события. Базовый синтаксис анонимного метода соответствует следующему псевдокоду:

```
class Program
{
    static void Main(string[] args)
    {
        НекоторыйТип t = new НекоторыйТип();
        t.НекотороеСобытие += delegate (дополнительноУказанныеАргументыДелегата)
        { /* операторы */ };
    }
}
```

Обратите внимание, что при обработке первого события `AboutToBlow` внутри предыдущего метода `Main()` аргументы, передаваемые из делегата, не указываются:

```
cl.AboutToBlow += delegate {
    Console.WriteLine("Eek! Going too fast!");
};
```

Строго говоря, вы не обязаны принимать входные аргументы, отправленные определенным событием. Однако если планируется использовать эти входные аргументы, нужно указать параметры, прототипированные типом делегата (как показано во второй обработке событий `AboutToBlow` и `Exploded`). Например:

```
cl.AboutToBlow += delegate(object sender, CarEventArgs e) {
    Console.WriteLine("Critical Message from Car: {0}", e.msg);
};
```

Доступ к локальным переменным

Анонимные методы интересны тем, что могут обращаться к локальным переменным метода, в котором они определены. Формально такие переменные называются *внешними переменными* анонимного метода. Ниже отмечены некоторые важные моменты, касающиеся взаимодействия между контекстом анонимного метода и контекстом определяющего их метода.

1. Анонимный метод не имеет доступа к параметрам `ref` и `out` определяющего их метода.
2. Анонимный метод не может иметь локальные переменные, имена которых совпадают с именами локальных переменных объемлющего метода.
3. Анонимный метод может обращаться к переменным экземпляра (или статическим переменным) из контекста объемлющего класса.
4. Анонимный метод может объявлять локальные переменные с теми же именами, что и у членов объемлющего класса (локальные переменные имеют отдельный контекст и скрывают внешние переменные-члены).

Предположим, что метод `Main()` определяет локальную переменную по имени `aboutToBlowCounter` типа `int`. Внутри анонимных методов, обрабатывающих событие `AboutToBlow`, мы увеличим значение этого счетчика на 1 и выведем результат на консоль перед завершением `Main()`:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Anonymous Methods *****\n");
    int aboutToBlowCounter = 0;

    // Создать Car обычным образом.
    Car cl = new Car("SlugBug", 100, 10);

    // Зарегистрировать обработчики событий в виде анонимных методов.
    cl.AboutToBlow += delegate
    {
        aboutToBlowCounter++;
        Console.WriteLine("Eek! Going too fast!");
    };

    cl.AboutToBlow += delegate(object sender, CarEventArgs e)
    {
        aboutToBlowCounter++;
        Console.WriteLine("Critical Message from Car: {0}", msg);
    };
    ...
}
```

```

Console.WriteLine("AboutToBlow event was fired {0} times.",
    aboutToBlowCounter);
Console.ReadLine();
}

```

После запуска этого модифицированного метода `Main()` вы обнаружите, что финальный вывод `Console.WriteLine()` сообщит о двукратном вызове `AboutToBlow`.

Исходный код. Проект `AnonymousMethods` доступен в подкаталоге `Chapter 10`.

Понятие лямбда-выражений

Чтобы завершить знакомство с архитектурой событий .NET, рассмотрим лямбда-выражения. Как объяснялось ранее в этой главе, C# поддерживает способность обрабатывать события “встроенным образом”, назначая блок операторов кода непосредственно событию с использованием анонимных методов вместо построения отдельного метода, подлежащего вызову делегатом. Лямбда-выражения — это всего лишь лаконичный способ записи анонимных методов, который в конечном итоге упрощает работу с типами делегатов .NET.

Чтобы подготовить фундамент для изучения лямбда-выражений, создадим новое консольное приложение по имени `SimpleLambdaExpressions`. Теперь займемся методом `FindAll()` обобщенного типа `List<T>`. Этот метод может быть вызван, когда нужно извлечь подмножество элементов из коллекции, и он имеет следующий прототип:

```

// Метод класса System.Collections.Generic.List<T>.
public List<T> FindAll(Predicate<T> match)

```

Как видите, этот метод возвращает объект `List<T>`, представляющий подмножество данных. Также обратите внимание, что единственный параметр `FindAll()` — обобщенный делегат типа `System.Predicate<T>`. Этот делегат может указывать на любой метод, возвращающий `bool` и принимающий единственный параметр:

```

// Этот делегат используется методом FindAll()
// для извлечения подмножества.
public delegate bool Predicate<T>(T obj);

```

Когда вызывается `FindAll()`, каждый элемент в `List<T>` передается методу, указанному объектом `Predicate<T>`. Реализация этого метода будет производить некоторые вычисления для проверки соответствия элемента данным указанному критерию, возвращая в результате `true` или `false`. Если метод вернет `true`, то текущий элемент будет добавлен в `List<T>`, представляющий искомое подмножество.

Прежде чем посмотреть, как лямбда-выражения упрощают работу с `FindAll()`, давайте решим эту задачу в длинной нотации, используя объекты делегатов непосредственно. Добавим в класс `Program` метод (по имени `TraditionalDelegateSyntax()`), который взаимодействует с `System.Predicate<T>` для обнаружения четных чисел в списке `List<T>` целочисленных значений:

```

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with Lambdas *****\n");
        TraditionalDelegateSyntax();
        Console.ReadLine();
    }
}

```

```

static void TraditionalDelegateSyntax()
{
    // Создать список целых чисел.
    List<int> list = new List<int>();
    list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });

    // Вызов FindAll() с использованием традиционного синтаксиса делегатов.
    Predicate<int> callback = new Predicate<int>(IsEvenNumber);
    List<int> evenNumbers = list.FindAll(callback);
    Console.WriteLine("Here are your even numbers:");
    foreach (int evenNumber in evenNumbers)
    {
        Console.Write("{0}\t", evenNumber);
    }
    Console.WriteLine();
}

// Цель для делегата Predicate<>.
static bool IsEvenNumber(int i)
{
    // Это четное число?
    return (i % 2) == 0;
}
}

```

Здесь имеется метод (`IsEvenNumber()`), отвечающий за проверку входного целочисленного параметра на предмет четности или нечетности с применением операции C# взятия модуля `%` (получения остатка от деления). В результате запуска приложения на консоль выводятся числа 20, 4, 8 и 44.

Хотя этот традиционный подход к работе с делегатами функционирует ожидаемым образом, метод `IsEvenNumber()` вызывается только при очень ограниченных условиях; в частности, когда вызывается `FindAll()`, который взваливает на нас все заботы относительно определения метода. Если бы вместо этого использовался анонимный метод, код стал бы существенно яснее. Рассмотрим следующий новый метод в классе `Program`:

```

static void AnonymousMethodSyntax()
{
    // Создать список целых.
    List<int> list = new List<int>();
    list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });

    // Теперь использовать анонимный метод.
    List<int> evenNumbers = list.FindAll(delegate(int i)
        { return (i % 2) == 0; });

    // Вывод четных чисел.
    Console.WriteLine("Here are your even numbers:");
    foreach (int evenNumber in evenNumbers)
    {
        Console.Write("{0}\t", evenNumber);
    }
    Console.WriteLine();
}

```

В этом случае вместо прямого создания типа делегата `Predicate<T>` с последующим написанием отдельного метода метод встраивается как анонимный. Хотя это шаг в правильном направлении, мы все еще обязаны применять ключевое слово `delegate` (или строго типизированный `Predicate<T>`) и должны удостовериться в точном соответствии списка параметров:


```
List<int> evenNumbers = list.FindAll(
    delegate(int i)
    {
        return (i % 2) == 0;
    }
);
```

Для дальнейшего упрощения вызова `FindAll()` можно применять *лямбда-выражения*. Используя этот новый синтаксис, вообще не приходится иметь дело с лежащим в основе объектом делегата. Рассмотрим следующий новый метод в классе `Program`:

```
static void LambdaExpressionSyntax()
{
    // Создать список целых.
    List<int> list = new List<int>();
    list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });

    // Теперь использовать лямбда-выражение C#.
    List<int> evenNumbers = list.FindAll(i => (i % 2) == 0);

    // Вывод четных чисел.
    Console.WriteLine("Here are your even numbers:");
    foreach (int evenNumber in evenNumbers)
    {
        Console.Write("{0}\t", evenNumber);
    }
    Console.WriteLine();
}
```

Здесь обратите внимание на довольно странный оператор кода, передаваемый методу `FindAll()`, который в действительности и является лямбда-выражением. В этой модификации примера вообще нет никаких следов делегата `Predicate<T>` (как и ключевого слова `delegate`). Все, что указано вместо них — это лямбда-выражение:

```
i => (i % 2) == 0
```

Прежде чем разбирать синтаксис дальше, пока просто запомните, что лямбда-выражения могут применяться везде, где используется анонимный метод или строго типизированный делегат (обычно в более лаконичном виде). “За кулисами” компилятор C# транслирует лямбда-выражение в стандартный анонимный метод, использующий тип делегата `Predicate<T>` (в чем можно убедиться с помощью утилиты `ildasm.exe` или `reflector.exe`). В частности, следующий оператор кода:

```
// Это лямбда-выражение...
List<int> evenNumbers = list.FindAll(i => (i % 2) == 0);
```

компилируется в примерно такой код C#:

```
// ...превращается в следующий анонимный метод.
List<int> evenNumbers = list.FindAll(delegate (int i)
{
    return (i % 2) == 0;
});
```

Анализ лямбда-выражения

Лямбда-выражение начинается со списка параметров, за которым следует лексема `=>` (лексема C# для лямбда-выражения позаимствована из лямбда-вычислений), а за ней — набор операторов (или единственный оператор), который будет обрабатывать параметры. На самом высоком уровне лямбда-выражение можно представить следующим образом:

```
АргументыДляОбработки => ОбрабатывающиеОператоры
```

То, что находится внутри метода `LambdaExpressionSyntax()`, следует понимать так:

```
// i – список параметров.
// (i % 2) == 0 – набор операторов для обработки i.
List<int> evenNumbers = list.FindAll(i => (i % 2) == 0);
```

Параметры лямбда-выражения могут быть типизированы явно или неявно. В настоящий момент тип данных, представляющий параметр `i` (целое), определяется неявно. Компилятор способен понять, что `i` — целое, на основе контекста всего лямбда-выражения, поместив тип данных и имя переменной в пару скобок, как показано ниже:

```
// Теперь установим тип параметров явно.
List<int> evenNumbers = list.FindAll((int i) => (i % 2) == 0);
```

Как было показано, если лямбда-выражение имеет одиночный неявно типизированный параметр, то скобки в списке параметров могут быть опущены. При желании быть последовательным в использовании параметров лямбда-выражений, можно всегда заключать список параметров в скобки, чтобы выражение выглядело так:

```
List<int> evenNumbers = list.FindAll((i) => (i % 2) == 0);
```

И, наконец, обратите внимание, что сейчас выражение не заключено в скобки (разумеется, вычисление остатка от деления помещается в скобки, чтобы гарантировать его выполнение перед проверкой равенства). Лямбда-выражение с оператором, заключенным в скобки, выглядит следующим образом:

```
// Теперь заключим в скобки и выражение.
List<int> evenNumbers = list.FindAll((i) => ((i % 2) == 0));
```

Теперь, когда известны разные способы построения лямбда-выражения, как его представить в понятных человеку терминах? Оставив чистую математику в стороне, можно привести следующее объяснение:

```
// Список параметров (в данном случае – единственное целое по имени i)
// будет обработан выражением (i % 2) == 0.
List<int> evenNumbers = list.FindAll((i) => ((i % 2) == 0));
```

Обработка аргументов внутри множества операторов

Наше первое лямбда-выражение состояло из единственного оператора, который в результате вычисления дает булевское значение. Однако, как должно быть хорошо известно, многие цели делегатов должны выполнять множество операторов кода. По этой причине C# позволяет строить лямбда-выражения, состоящие из нескольких блоков операторов. Когда выражение должно обрабатывать параметры в нескольких строках кода, понадобится выделить контекст этих операторов с помощью фигурных скобок. Рассмотрим следующую модификацию метода `LambdaExpressionSyntax()`:

```
static void LambdaExpressionSyntax()
{
    // Создать список целых.
    List<int> list = new List<int>();
    list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });

    // Обработать каждый аргумент в группе операторов кода.
    List<int> evenNumbers = list.FindAll((i) =>
    {
        Console.WriteLine("value of i is currently: {0}", i);
        bool isEven = ((i % 2) == 0);
        return isEven;
    });
}
```

```
// Вывод четных чисел.
Console.WriteLine("Here are your even numbers:");
foreach (int evenNumber in evenNumbers)
{
    Console.Write("{0}\t", evenNumber);
}
Console.WriteLine();
}
```

В этом случае список параметров (опять состоящий из единственного целого `i`) обрабатывается набором операторов кода. Помимо вызова `Console.WriteLine()`, оператор вычисления остатка от деления разбит на два оператора для повышения читабельности. Предположим, что каждый из рассмотренных выше методов вызывается в `Main()`:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Lambdas *****\n");
    TraditionalDelegateSyntax();
    AnonymousMethodSyntax();
    Console.WriteLine();
    LambdaExpressionSyntax();
    Console.ReadLine();
}
```

Запуск этого приложения даст следующий вывод:

```
***** Fun with Lambdas *****
Here are your even numbers:
20    4    8    44
Here are your even numbers:
20    4    8    44
value of i is currently: 20
value of i is currently: 1
value of i is currently: 4
value of i is currently: 8
value of i is currently: 9
value of i is currently: 44
Here are your even numbers:
20    4    8    44
```

Исходный код. Проект `SimpleLambdaExpressions` доступен в подкаталоге `Chapter 10`.

Лямбда-выражения с несколькими параметрами и без параметров

Показанные выше лямбда-выражения обрабатывали единственный параметр. Однако это вовсе не обязательно, поскольку лямбда-выражения могут обрабатывать множество аргументов или вообще не иметь аргументов. Для иллюстрации первого сценария создадим консольное приложение по имени `LambdaExpressionsMultipleParams` со следующей версией класса `SimpleMath`:

```
public class SimpleMath
{
    public delegate void MathMessage(string msg, int result);
    private MathMessage mmDelegate;

    public void SetMathHandler(MathMessage target)
    {mmDelegate = target; }

    public void Add(int x, int y)
    {
```

```

    if (mmDelegate != null)
        mmDelegate.Invoke("Adding has completed!", x + y);
}

```

Обратите внимание, что делегат `MathMessage` принимает два параметра. Чтобы представить их в виде лямбда-выражения, метод `Main()` может быть реализован так:

```

static void Main(string[] args)
{
    // Зарегистрировать делегат как лямбда-выражение.
    SimpleMath m = new SimpleMath();
    m.SetMathHandler((msg, result) =>
        { Console.WriteLine("Message: {0}, Result: {1}", msg, result); });
    // Это приведет к выполнению лямбда-выражения.
    m.Add(10, 10);
    Console.ReadLine();
}

```

Здесь используется выводение типа компилятором, поскольку для простоты два параметра не типизированы строго. Однако можно было бы вызвать `SetMathHandler()` следующим образом:

```

m.SetMathHandler((string msg, int result) =>
    { Console.WriteLine("Message: {0}, Result: {1}", msg, result); });

```

И, наконец, если лямбда-выражение применяется для взаимодействия с делегатом, вообще не принимающим параметров, то это можно сделать, указав в качестве параметра пару пустых скобок. Таким образом, предполагая, что определен следующий тип делегата:

```
public delegate string VerySimpleDelegate();
```

вот как можно обработать результат вызова:

```

// Вывод на консоль строки "Enjoy your string!".
VerySimpleDelegate d = new VerySimpleDelegate( () => { return "Enjoy your string!"; } );
Console.WriteLine(d.Invoke());

```

Исходный код. Проект `LambdaExpressionsMultipleParams` доступен в подкаталоге `Chapter 10`.

Усовершенствование примера `PrimAndProperCarEvents` за счет использования лямбда-выражений

Учитывая то, что главное предназначение лямбда-выражений состоит в обеспечении возможности в чистой, сжатой манере определить анонимный метод (и тем самым упростить работу с делегатами), давайте переделаем проект `PrimAndProperCarEvents`, созданный ранее в этой главе. Ниже приведена упрощенная версия класса `Program` этого проекта, в которой используется синтаксис лямбда-выражений (вместо простых делегатов) для перехвата всех событий, поступающих от объекта `Car`.

```

static void Main(string[] args)
{
    Console.WriteLine("***** More Fun with Lambdas *****\n");
    // Создать объект Car обычным образом.
    Car c1 = new Car("SlugBug", 100, 10);
    // Использовать лямбда-выражения.
    c1.AboutToBlow += (sender, e) => { Console.WriteLine(e.msg); };
    c1.Exploded += (sender, e) => { Console.WriteLine(e.msg); };
}

```

```
// Ускорить (это инициирует события).
Console.WriteLine("\n***** Speeding up *****");
for (int i = 0; i < 6; i++)
    cl.Accelerate(20);

Console.ReadLine();
}
```

Теперь общая роль лямбда-выражений должна проясниться, и становится понятно, что они обеспечивают “функциональную манеру” работы с анонимными методами и типами делегатов. Несмотря на то что привыкание к лямбда-операции ($\Rightarrow$) может занять некоторое время, помните, что любые лямбда-выражения сводятся к следующему простому уравнению:

```
( АргументыДляОбработки ) => { ОбработывающиеИхОператоры }
```

Стоит отметить, что лямбда-выражения широко используются также и в модели программирования LINQ, помогая упрощать кодирование. Знакомство с LINQ начинается в главе 12.

Исходный код. Проект CarEventsWithLambdas доступен в подкаталоге Chapter 10.

Резюме

В этой главе вы ознакомились с несколькими способами двустороннего взаимодействия множества объектов. Во-первых, было рассмотрено ключевое слово `delegate`, используемое для непрямого конструирования класса, производного от `System.MulticastDelegate`. Как было показано, объект делегата поддерживает список методов для вызова тогда, когда ему об этом укажут. Такие вызовы могут выполняться синхронно (с использованием метода `Invoke()`) или асинхронно (через методы `BeginInvoke()` и `EndInvoke()`). Асинхронная природа типов делегатов .NET будет описана в главе 19.

Во-вторых, вы ознакомились с ключевым словом `event`, которое в сочетании с типом делегата может упростить процесс отправки уведомлений о событиях ожидающим объектам. Как видно в результирующем коде CIL, модель событий .NET отображается на скрытые обращения к типам `System.Delegate/System.MulticastDelegate`. В этом свете ключевое слово `event` является необязательным и просто позволяет сэкономить на наборе кода.

В-третьих, в этой главе также рассматривалось средство языка C#, которое называется *анонимными методами*. С помощью такой синтаксической конструкции можно явно ассоциировать блок операторов кода с заданным событием. Как было показано, анонимные методы могут игнорировать параметры, переданные событием, и имеют доступ к “внешним переменным” определяющего их метода. Вы также ознакомились с упрощенным способом регистрации событий с применением *групповых преобразований методов*.

И, наконец, в завершение главы было дано описание лямбда-операции $\Rightarrow$ в C#. Как было показано, этот синтаксис значительно сокращает нотацию написания анонимных методов, когда набор аргументов может быть передан на обработку группе операторов. Любой метод внутри платформы .NET, который принимает объект делегата в качестве аргумента, может быть заменен соответствующим лямбда-выражением, что обычно существенно упрощает кодовую базу.

Расширенные средства языка C#

В этой главе рассматриваются некоторые более сложные синтаксические конструкции языка программирования C#. Сначала будет показано, как реализуется и используется *метод-индексатор*. Этот механизм C# позволяет строить специальные типы, которые обеспечивают доступ к внутренним подтипам с применением синтаксиса, похожего на синтаксис массивов. Затем вы узнаете о том, как перегружать различные операции (+, -, <, > и т.д.) и как создавать специальные процедуры явного и неявного преобразования типов (а также причины, по которым это может требоваться).

Далее рассматриваются темы, которые особенно полезны при работе с API-интерфейсами LINQ (хотя это применимо и вне контекста LINQ), а именно: расширяющие методы и анонимные типы.

И в завершение вы узнаете, как создавать контекст “небезопасного” кода, чтобы напрямую манипулировать неуправляемыми указателями. Хотя использовать указатели в приложениях C# приходится исключительно редко, понимание того, как это делается, может пригодиться в определенных ситуациях со сложными сценариями взаимодействия.

Понятие методов-индексаторов

Программисты хорошо знакомы с процессом доступа к индивидуальным элементам, содержащимся в стандартных массивах, через операцию индекса ([]). Например:

```
static void Main(string[] args)
{
    // Цикл по аргументам командной строки с использованием операции индекса.
    for(int i = 0; i < args.Length; i++)
        Console.WriteLine("Args: {0}", args[i]);

    // Объявление локального массива целочисленных значений.
    int[] myInts = { 10, 9, 100, 432, 9874 };

    // Использование операции индекса для доступа к элементам.
    for(int j = 0; j < myInts.Length; j++)
        Console.WriteLine("Index {0} = {1} ", j, myInts[j]);
    Console.ReadLine();
}
```

Приведенный код не должен быть для вас чем-то новым. В C# имеется возможность проектировать специальные классы и структуры, которые могут быть индексированы подобно стандартному массиву, за счет определения *метода-индексатора*. Это конк-

ретное языковое средство наиболее полезно при создании специальных типов коллекций (обобщенных и необобщенных).

Прежде чем ознакомиться с реализацией специального индексатора, начнем с рассмотрения его в действии. Предположим, что вы добавили поддержку метода-индексатора к специальному типу `PersonCollection`, разработанному в главе 9 (в проекте `IssuesWithNonGenericCollections`). Хотя индексатор еще не добавлен, рассмотрим следующее его применение в новом консольном приложении по имени `SimpleIndexer`:

```
// Индексаторы позволяют обращаться к элементам в стиле массива.
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with Indexers *****\n");

        PersonCollection myPeople = new PersonCollection();

        // Добавить объекты с применением синтаксиса индексатора.
        myPeople[0] = new Person("Homer", "Simpson", 40);
        myPeople[1] = new Person("Marge", "Simpson", 38);
        myPeople[2] = new Person("Lisa", "Simpson", 9);
        myPeople[3] = new Person("Bart", "Simpson", 7);
        myPeople[4] = new Person("Maggie", "Simpson", 2);

        // Получить и отобразить элементы с использованием индексатора.
        for (int i = 0; i < myPeople.Count; i++)
        {
            Console.WriteLine("Person number: {0}", i);
            Console.WriteLine("Name: {0} {1}",
                myPeople[i].FirstName, myPeople[i].LastName);
            Console.WriteLine("Age: {0}", myPeople[i].Age);
            Console.WriteLine();
        }
    }
}
```

Как видите, индексаторы позволяют манипулировать внутренней коллекцией подобно стандартному массиву. Но тут возникает серьезный вопрос: как сконфигурировать класс `PersonCollection` (или любой другой класс либо структуру) для поддержки этой функциональности? Индексатор представлен как несколько видоизмененное определение свойства C#. В его простейшей форме индексатор создается с использованием синтаксиса `this[]`. Ниже показано необходимое изменение класса `PersonCollection`:

```
// Добавим индексатор к существующему определению класса.
public class PersonCollection : IEnumerable
{
    private ArrayList arPeople = new ArrayList();

    // Специальный индексатор для этого класса.
    public Person this[int index]
    {
        get { return (Person)arPeople[index]; }
        set { arPeople.Insert(index, value); }
    }

    ...
}
```

Помимо использования ключевого слова `this`, индексатор выглядит как объявление любого другого свойства C#. Например, роль конструкции `get` состоит в возврате корректного объекта вызывающему коду. Здесь мы фактически и делаем это, делегируя

запрос к индексатору объекта `ArrayList`, поскольку этот класс также поддерживает индексатор. В противоположность этому, конструкция `set` отвечает за добавление новых объектов `Person`; в данном примере это достигается вызовом метода `Insert()` объекта `ArrayList`.

Индексаторы — это просто еще одна форма синтаксиса, учитывая, что та же функциональность может быть обеспечена с использованием “нормальных” открытых методов вроде `AddPerson()` или `GetPerson()`. Тем не менее, поддержка методов-индексаторов в специальных типах коллекций позволяет их легко интегрировать с библиотеками базовых классов .NET.

Хотя создание методов-индексаторов является обычным делом при построении специальных коллекций, следует помнить, что обобщенные типы предлагают эту функциональность в готовом виде. В следующем методе используется обобщенный список `List<T>` объектов `Person`. Обратите внимание, что индексатор `List<T>` можно просто применять непосредственно. Например:

```
static void UseGenericListOfPeople()
{
    List<Person> myPeople = new List<Person>();
    myPeople.Add(new Person("Lisa", "Simpson", 9));
    myPeople.Add(new Person("Bart", "Simpson", 7));

    // Изменить первую персону с помощью индексатора.
    myPeople[0] = new Person("Maggie", "Simpson", 2);

    // Теперь получить и отобразить каждый элемент с использованием индексатора.
    for (int i = 0; i < myPeople.Count; i++)
    {
        Console.WriteLine("Person number: {0}", i);
        Console.WriteLine("Name: {0} {1}", myPeople[i].FirstName,
            myPeople[i].LastName);
        Console.WriteLine("Age: {0}", myPeople[i].Age);
        Console.WriteLine();
    }
}
```

Исходный код. Проект `SimpleIndexer` доступен в подкаталоге `Chapter 11`.

Индексация данных с использованием строковых значений

В текущем классе `PersonCollection` определен индексатор, позволяющий вызывающему коду идентифицировать подэлементы с применением числовых значений. Однако надо понимать, что это не обязательное требование метода-индексатора. Предположим, что решено хранить объекты `Person`, используя `System.Collections.Generic.Dictionary<TKey, TValue>` вместо `ArrayList`. Учитывая, что типы `Dictionary` позволяют производить доступ к содержащимся в них типам с использованием строкового маркера (такого как фамилия персоны), индексатор можно было бы определить следующим образом:

```
public class PersonCollection : IEnumerable
{
    private Dictionary<string, Person> listPeople =
        new Dictionary<string, Person>();

    // Этот индексатор возвращает персону по строковому индексу.
    public Person this[string name]
    {
        get { return (Person)listPeople[name]; }
```



```

    set { listPeople[name] = value; }
}

public void ClearPeople()
{ listPeople.Clear(); }

public int Count
{ get { return listPeople.Count; } }

IEnumerator IEnumerable.GetEnumerator()
{ return listPeople.GetEnumerator(); }
}

```

Теперь вызывающий код может взаимодействовать с содержащимися внутри объектами `Person`, как показано ниже:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Indexers *****\n");
    PersonCollection myPeople = new PersonCollection();
    myPeople["Homer"] = new Person("Homer", "Simpson", 40);
    myPeople["Marge"] = new Person("Marge", "Simpson", 38);

    // Получить "Homer" и вывести данные.
    Person homer = myPeople["Homer"];
    Console.WriteLine(homer.ToString());
    Console.ReadLine();
}

```

Опять-таки, если использовать обобщенный тип `Dictionary<TKey, TValue>` напрямую, получится функциональность метода-индексатора в готовом виде, без построения специального необобщенного класса, поддерживающего строковый индексатор. Тем не менее, необходимо понимать, что тип данных любого индексатора будет основан на том, как поддерживающий тип коллекции позволяет вызывающему коду извлекать подэлементы.

Исходный код. Проект `StringIndexer` доступен в подкаталоге `Chapter 11`.

Перегрузка методов-индексаторов

Имейте в виду, что методы-индексаторы могут быть перегружены в отдельном классе или структуре. То есть если имеет смысл позволить вызывающему коду обращаться к подэлементам с использованием числового индекса или строкового значения, в одном и том же типе можно определить несколько индексаторов. Например, в ADO.NET (встроенный API-интерфейс .NET для доступа к базам данных) класс `DataSet` поддерживает свойство по имени `Tables`, которое возвращает строго типизированную коллекцию `DataTableCollection`. В свою очередь, в `DataTableCollection` определены три индексатора для получения объектов `DataTable` — по порядковому номеру, по дружественным строковым именам и необязательному пространству имен:

```

public sealed class DataTableCollection : InternalDataCollectionBase
{
    ...
    // Перегруженные индексаторы.
    public DataTable this[string name] { get; }
    public DataTable this[string name, string tableNamespace] { get; }
    public DataTable this[int index] { get; }
}

```

Поддержка методов-индексаторов очень характерна для типов из библиотек базовых классов. Поэтому даже если текущий проект не требует построения специальных индексаторов для классов и структур, помните, что многие типы уже поддерживают этот синтаксис.

Многомерные индексаторы

Можно также создавать метод-индексатор, принимающий несколько параметров. Предположим, что имеется специальная коллекция, хранящая подэлементы двумерного массива. В этом случае метод-индексатор можно сконфигурировать следующим образом:

```
public class SomeContainer
{
    private int[,] my2DintArray = new int[10, 10];
    public int this[int row, int column]
    { /* установить или получить значение из двумерного массива */ }
}
```

Если только не строится очень специализированный класс коллекций, то вряд ли понадобится создавать многомерные индексаторы. Здесь снова пример ADO.NET показывает, насколько полезной может быть эта конструкция. Класс `DataTable` в ADO.NET — это, по сути, коллекция строк и столбцов, похожая на миллиметровку или электронную таблицу Microsoft Excel.

Хотя объекты `DataTable` обычно наполняются без вашего участия, посредством связанных с ними “адаптеров данных”, в приведенном ниже коде показано, как вручную создать находящийся в памяти объект `DataTable`, содержащий три столбца (для имени, фамилии и возраста). Обратите внимание, что после добавления одной строки в `DataTable` с помощью многомерного индексатора производится обращение ко всем столбцам первой (и единственной) строки. (Чтобы реализовать это, в файле кода понадобится импортировать пространство имен `System.Data`.)

```
static void MultiIndexerWithDataTable()
{
    // Создать простой объект DataTable с тремя столбцами.
    DataTable myTable = new DataTable();
    myTable.Columns.Add(new DataColumn("FirstName"));
    myTable.Columns.Add(new DataColumn("LastName"));
    myTable.Columns.Add(new DataColumn("Age"));

    // Добавить строку к таблице.
    myTable.Rows.Add("Mel", "Appleby", 60);

    // Использовать многомерный индексатор для вывода деталей первой строки.
    Console.WriteLine("First Name: {0}", myTable.Rows[0][0]);
    Console.WriteLine("Last Name: {0}", myTable.Rows[0][1]);
    Console.WriteLine("Age : {0}", myTable.Rows[0][2]);
}
```

Начиная с главы 21, мы продолжим рассмотрение ADO.NET, так что не пугайтесь, если что что-то в приведенном выше коде покажется незнакомым. Этот пример просто иллюстрирует, что методы-индексаторы могут поддерживать множество измерений, а при правильном использовании могут упростить взаимодействие с подобъектами, содержащимися в специальных коллекциях.

Определения индексаторов в интерфейсных типах

Индексаторы могут определяться в типе интерфейса, позволяя поддерживающим типам предоставлять их специальные реализации. Ниже показан пример такого интерфейса, который определяет протокол для получения строковых объектов с использованием числового индексатора:

```
public interface IStringContainer
{
    string this[int index] { get; set; }
}
```

При таком определении интерфейса любой класс или структура, реализующие его, должны поддерживать индексатор чтения/записи, манипулирующий подэлементами через числовое значение. Вот частичная реализация такого класса:

```
class SomeClass : IStringContainer
{
    private List<string> myStrings = new List<string>();
    public string this[int index]
    {
        get { return myStrings[index]; }
        set { myStrings.Insert(index, value); }
    }
}
```

На этом первая главная тема настоящей главы завершена. А теперь давайте рассмотрим языковое средство, позволяющее строить специальные классы и структуры, которые уникальным образом реагируют на встроенные операции C# — *перегрузку операций*.

Понятие перегрузки операций

Подобно любому языку программирования, в C# имеется готовый набор лексем, используемых для выполнения базовых операций над встроенными типами. Например, известно, что операция `+` может применяться к двум целым, чтобы дать их сумму:

```
// Операция + с целыми.
int a = 100;
int b = 240;
int c = a + b; // c теперь равно 340
```

Здесь нет ничего нового, но задумывались ли вы когда-нибудь о том, что одна и та же операция `+` может применяться к большинству встроенных типов данных C#? К примеру, рассмотрим такой код:

```
// Операция + со строками.
string s1 = "Hello";
string s2 = " world!";
string s3 = s1 + s2; // s3 теперь содержит "Hello world!"
```

По сути, функциональность операции `+` уникальным образом базируются на представленных типах данных (в этом случае строках или целых числах). Когда операция `+` применяется к числовым типам, мы получаем арифметическую сумму операндов. Однако когда та же операция применяется к строковым типам, получается конкатенация строк.

Язык C# предоставляет возможность строить специальные классы и структуры, которые также уникально реагируют на один и тот же набор базовых лексем (вроде операции `+`). Хотя не каждая встроенная операция C# может быть перегружена, многие операции перегрузку допускают, как показано в табл. 11.1.

Таблица 11.1. Возможность перегрузки операций C#

Операция C#	Возможность перегрузки
<code>+, -, !, ~, ++, --, true, false</code>	Эти унарные операции могут быть перегружены
<code>+, -, *, /, %, &, , ^, <<, >></code>	Эти бинарные операции могут быть перегружены
<code>==, !=, <, >, <=, >=</code>	Эти операции сравнения могут быть перегружены. C# требует совместной перегрузки "подобных" операций (т.е. <code><</code> и <code>></code> , <code><=</code> и <code>>=</code> , <code>==</code> и <code>!=</code>)
<code>[]</code>	Операция <code>[]</code> не может быть перегружена. Однако, как было показано ранее в этой главе, аналогичную функциональность предлагают индексаторы
<code>()</code>	Операция <code>()</code> не может быть перегружена. Однако, как будет показано далее в этой главе, ту же функциональность предоставляют специальные методы преобразования
<code>+=, -=, *=, /=, %=, &=, =, ^=, <<=, >>=</code>	Сокращенные операции присваивания не могут перегружаться; однако вы получаете их автоматически, перегружая соответствующую бинарную операцию

Перегрузка бинарных операций

Чтобы проиллюстрировать процесс перегрузки бинарных операций, представим следующий простой класс `Point`, определенный в новом консольном приложении по имени `OverloadedOps`:

```
// Простой класс C# для повседневного пользования.
public class Point
{
    public int X {get; set;}
    public int Y {get; set;}

    public Point(int xPos, int yPos)
    {
        X = xPos;
        Y = yPos;
    }

    public override string ToString()
    {
        return string.Format("[{0}, {1}]", this.X, this.Y);
    }
}
```

Теперь, рассуждая логически, суммирование экземпляров `Point` имеет смысл. Например, если сложить вместе две переменных `Point`, получится новая `Point` с суммарными значениями `x` и `y`. Кстати, также может оказаться полезной возможность вычитания одной `Point` из другой. В идеале хотелось бы иметь возможность написания такого кода:

```
// Сложение и вычитание двух точек?
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Overloaded Operators *****\n");

    // Создать две точки.
    Point ptOne = new Point(100, 100);
    Point ptTwo = new Point(40, 40);
```

```

Console.WriteLine("ptOne = {0}", ptOne);
Console.WriteLine("ptTwo = {0}", ptTwo);

// Сложить две точки, чтобы получить большую?
Console.WriteLine("ptOne + ptTwo: {0} ", ptOne + ptTwo);

// Вычесть одну точку из другой, чтобы получить меньшую?
Console.WriteLine("ptOne - ptTwo: {0} ", ptOne - ptTwo);
Console.ReadLine();
}

```

Однако в том виде, как он есть, класс `Point` приведет к ошибкам на этапе компиляции, поскольку типу `Point` не известно, как реагировать на операции `+` и `-`. Чтобы оснастить специальный тип возможностью уникально реагировать на встроенные операции, в C# служит ключевое слово `operator`, которое может использоваться только в сочетании с ключевым словом `static`. При перегрузке бинарной операции (вроде `+` и `-`) чаще всего передаются два аргумента того же типа, что и определяющий их класс (в данном примере — `Point`); это иллюстрируется в следующей модификации кода:

```

// Более интеллектуальный тип Point.
public class Point
{
    ...
    // Перегруженная операция +.
    public static Point operator + (Point p1, Point p2)
    { return new Point(p1.X + p2.X, p1.Y + p2.Y); }

    // Перегруженная операция -.
    public static Point operator - (Point p1, Point p2)
    { return new Point(p1.X - p2.X, p1.Y - p2.Y); }
}

```

Логика, положенная в основу операции `+`, состоит просто в возврате нового экземпляра `Point` на основе сложения соответствующих полей входных параметров `Point`. Поэтому, когда вы напишете `pt1 + pt2`, “за кулисами” произойдет следующий скрытый вызов статического метода `operator+`:

```

// Псевдокод: Point p3 = Point.operator+ (p1, p2)
Point p3 = p1 + p2;

```

Аналогично, `p1 - p2` отображается на следующее:

```

// Псевдокод: Point p4 = Point.operator- (p1, p2)
Point p4 = p1 - p2;

```

После этого дополнения программа скомпилируется, и мы получим возможность складывать и вычитать объекты `Point`, как показано в следующем выводе:

```

ptOne = [100, 100]
ptTwo = [40, 40]
ptOne + ptTwo: [140, 140]
ptOne - ptTwo: [60, 60]

```

При перегрузке бинарной операции вы не обязаны передавать ей два параметра одинакового типа. Если это имеет смысл, один из аргументов может отличаться. Например, ниже показана перегруженная операция `+`, которая позволяет вызывающему коду получить новый объект `Point` на основе числового смещения:

```

public class Point
{
    ...
    public static Point operator + (Point p1, int change)
    {

```

```

    return new Point(p1.X + change, p1.Y + change);
}
public static Point operator + (int change, Point p1)
{
    return new Point(p1.X + change, p1.Y + change);
}
}

```

Обратите внимание, что если нужно передавать аргументы в любом порядке, потребуются обе версии метода (т.е. нельзя просто определить один из методов и рассчитывать, что компилятор автоматически будет поддерживать другой). Теперь можно использовать эти новые версии операции + следующим образом:

```

// Выводит [110, 110].
Point biggerPoint = ptOne + 10;
Console.WriteLine("ptOne + 10 = {0}", biggerPoint);

// Выводит [120, 120].
Console.WriteLine("10 + biggerPoint = {0}", 10 + biggerPoint);
Console.WriteLine();

```

А как насчет операций += и -=?

Перешедших на C# с языка C++ может удивить отсутствие возможности перегрузки операций сокращенного присваивания (+=, -= и т.д.). Не беспокойтесь. В C# операции сокращенного присваивания автоматически эмулируются при перегрузке соответствующих бинарных операций. Таким образом, если в классе Point уже перегружены операции + и -, можно написать следующий код:

```

// Перегрузка бинарных операций автоматически обеспечивает
// перегрузку сокращенных операций.
static void Main(string[] args)
{
    ...
    // Операция += автоматически перегружена.
    Point ptThree = new Point(90, 5);
    Console.WriteLine("ptThree = {0}", ptThree);
    Console.WriteLine("ptThree += ptTwo: {0}", ptThree += ptTwo);

    // Операция -= автоматически перегружена.
    Point ptFour = new Point(0, 500);
    Console.WriteLine("ptFour = {0}", ptFour);
    Console.WriteLine("ptFour -= ptThree: {0}", ptFour -= ptThree);
    Console.ReadLine();
}

```

Перегрузка унарных операций

В C# также допускается перегружать и унарные операции, такие как ++ и --. При перегрузке унарной операции также должно использоваться ключевое слово `static` с ключевым словом `operator`, однако в этом случае просто передается единственный параметр того же типа, что и определяющий его класс/структура. Например, дополните Point следующими перегруженными операциями:

```

public struct Point
{
    ...
    // Прибавить 1 к значениям X/Y входного объекта Point.
    public static Point operator ++(Point p1)
    { return new Point(p1.X+1, p1.Y+1); }
}

```

```
// Вычесть 1 из значений X/Y входного объекта Point.
public static Point operator --(Point p1)
{ return new Point(p1.X-1, p1.Y-1); }
}
```

В результате появляется возможность выполнять инкремент и декремент значений X и Y класса Point, как показано ниже:

```
static void Main(string[] args)
{
    ...
    // Применение унарных операций ++ и -- к Point.
    Point ptFive = new Point(1, 1);
    Console.WriteLine("++ptFive = {0}", ++ptFive); // [2, 2]
    Console.WriteLine("--ptFive = {0}", --ptFive); // [1, 1]
    // Применение тех же операций для постфиксного инкремента/декремента.
    Point ptSix = new Point(20, 20);
    Console.WriteLine("ptSix++ = {0}", ptSix++); // [20, 20]
    Console.WriteLine("ptSix-- = {0}", ptSix--); // [21, 21]
    Console.ReadLine();
}
```

В предыдущем примере кода обратите внимание, что специальные операции ++ и -- применяются двумя разными способами. В C++ допускается перегружать операции префиксного и постфиксного инкремента/декремента по отдельности. В C# это невозможно; тем не менее, возвращаемое значение инкремента/декремента автоматически обрабатывается правильно (т.е. для перегруженной операции ++ выражение pt++ имеет значение немодифицированного объекта, в то время как ++pt имеет новое значение, примененное перед использованием выражения).

Перегрузка операций эквивалентности

Как вы должны помнить из главы 6, метод System.Object.Equals() может быть перегружен для выполнения сравнений объектов на основе значений (а не ссылок). Если вы решите переопределить Equals() (часто вместе со связанным методом System.Object.GetHashCode()), это позволит легко переопределить и операции проверки эквивалентности (== и !=). Для иллюстрации рассмотрим модифицированное определение типа Point:

```
// Этот вариант Point также перегружает операции == и !=.
public class Point
{
    ...
    public override bool Equals(object o)
    {
        return o.ToString() == this.ToString();
    }

    public override int GetHashCode()
    { return this.ToString().GetHashCode(); }

    // Теперь перегрузим операции == и !=.
    public static bool operator ==(Point p1, Point p2)
    {
        return p1.Equals(p2);
    }

    public static bool operator !=(Point p1, Point p2)
    {
        return !p1.Equals(p2);
    }
}
```

Обратите внимание, что для выполнения нужной работы реализации операций `==` и `!=` просто вызывают перегруженный метод `Equals()`. Теперь класс `Point` можно использовать следующим образом:

```
// Использование перегруженных операций эквивалентности.
static void Main(string[] args)
{
    ...
    Console.WriteLine("ptOne == ptTwo : {0}", ptOne == ptTwo);
    Console.WriteLine("ptOne != ptTwo : {0}", ptOne != ptTwo);
    Console.ReadLine();
}
```

Как видите, сравнение двух объектов с применением хорошо знакомых операций `==` и `!=` выглядит намного интуитивнее, чем вызов `Object.Equals()`. При перегрузке операций эквивалентности для определенного класса помните, что C# требует, чтобы в случае перегрузки операции `==` обязательно перегружалась также и операция `!=` (компилятор напомним, если вы забудете это сделать).

Перегрузка операций сравнения

В главе 8 было показано, как реализовать интерфейс `Comparable` для выполнения сравнений двух сходных объектов. Вдобавок для того же класса можно перегрузить операции сравнения (`<`, `>`, `<=` и `>=`). Подобно операциям эквивалентности, C# и здесь требует, чтобы в случае перегрузки операции `<` обязательно перегружалась также и операция `>`. После перегрузки в классе `Point` этих операций сравнения пользователь объекта сможет сравнивать объекты `Point` следующим образом:

```
// Использование перегруженных операций < и >.
static void Main(string[] args)
{
    ...
    Console.WriteLine("ptOne < ptTwo : {0}", ptOne < ptTwo);
    Console.WriteLine("ptOne > ptTwo : {0}", ptOne > ptTwo);
    Console.ReadLine();
}
```

Предполагая, что интерфейс `Comparable` (или, даже лучше, его обобщенный эквивалент) уже реализован, перегрузка операций сравнения становится тривиальной. Ниже показано модифицированное определение класса.

```
// Объекты Point также можно сравнивать с помощью операций сравнения.
public class Point : Comparable<Point>
{
    ...
    public int CompareTo(Point other)
    {
        if (this.X > other.X && this.Y > other.Y)
            return 1;
        if (this.X < other.X && this.Y < other.Y)
            return -1;
        else
            return 0;
    }

    public static bool operator <(Point p1, Point p2)
    { return (p1.CompareTo(p2) < 0); }

    public static bool operator >(Point p1, Point p2)
    { return (p1.CompareTo(p2) > 0); }
```



```

public static bool operator <=(Point p1, Point p2)
{ return (p1.CompareTo(p2) <= 0); }

public static bool operator >=(Point p1, Point p2)
{ return (p1.CompareTo(p2) >= 0); }
}

```

Финальные соображения относительно перегрузки операций

Как уже было показано, C# предлагает возможность строить типы, которые могут уникальным образом реагировать на различные встроенные, хорошо известные операции. Теперь перед добавлением поддержки этого поведения в классы необходимо удостовериться в том, что операции, которые вы собираетесь перегружать, имеют хоть какой-то смысл в реальном мире.

Например, предположим, что перегружена операция умножения для класса `MiniVan` (минивэн). Что вообще должно означать перемножение двух объектов `MiniVan`? Не слишком много. Фактически, если коллеги по команде увидят следующее использование объектов `MiniVan`, то будут весьма озадачены:

```

// Это не слишком понятно...
MiniVan newVan = myVan * yourVan;

```

Перегрузка операций обычно полезна только при построении атомарных типов данных. Текст, точки, прямоугольники, функции и шестиугольники — подходящие кандидаты на перегрузку операций. Люди, менеджеры, автомобили, подключения к базе данных и веб-страницы — нет. Можно сформулировать такое эмпирическое правило: если перегруженная операция *затрудняет* пользователю понимание функциональности типа, не делайте этого. В общем, используйте средство перегрузки операций с умом.

Исходный код. Проект `OverloadedOps` доступен в подкаталоге `Chapter 11`.

Понятие специальных преобразований типов

Теперь обратимся к теме, близкой к перегрузке операций — специальным преобразованиям типов. Чтобы заложить фундамент для последующей дискуссии, давайте кратко вспомним нотацию явного и неявного преобразования между числовыми данными и связанными с ними типами классов.

Числовые преобразования

В терминах встроенных числовых типов (`sbyte`, `int`, `float` и т.п.) *явное преобразование* требуется при попытке сохранить большее значение в меньшем контейнере, поскольку это может привести к потере данных. По сути, это означает, что вы говорите компилятору: «я знаю, что делаю». В противоположность этому *неявное преобразование* происходит автоматически, когда вы пытаетесь поместить меньший тип в целевой тип, и в результате этой операции не происходит потеря данных:

```

static void Main()
{
    int a = 123;
    long b = a;           // Неявное преобразование int в long.
    int c = (int) b;      // Явное преобразование long в int.
}

```

Преобразования между связанными типами классов

Как было показано в главе 6, типы классов могут быть связаны классическим наследованием (отношение “является” (“is a”)). В этом случае процесс преобразования C# позволяет выполнять приведение вверх и вниз по иерархии классов. Например, класс-наследник всегда может быть неявно приведен к базовому типу. Однако если необходимо хранить тип базового класса в переменной типа класса-наследника, понадобится явное приведение:

```
// Два связанных типа классов.
class Base{}
class Derived : Base{}
class Program
{
    static void Main(string[] args)
    {
        // Неявное приведение наследника к предку.
        Base myBaseType;
        myBaseType = new Derived();
        // Для хранения базовой ссылки в ссылке
        // на наследника нужно явное преобразование.
        Derived myDerivedType = (Derived)myBaseType;
    }
}
```

Это явное приведение работает благодаря тому факту, что классы `Base` и `Derived` связаны отношением классического наследования. Однако что если есть два типа классов из *разных иерархий* без общего предка (кроме `System.Object`), которые требуют преобразования друг в друга? Если они не связаны классическим наследованием, типичные операции приведения здесь не помогут.

В качестве примера рассмотрим типы значений (структуры). Предположим, что определены две структуры .NET с именами `Square` и `Rectangle`. Учитывая, что они не могут полагаться на классическое наследование (поскольку всегда запечатаны), нет естественного способа выполнить приведение между этими, на первый взгляд, связанными типами.

Наряду с возможностью создания в структурах вспомогательных методов (вроде `Rectangle.ToSquare()`), язык C# позволяет строить специальные процедуры преобразования, которые позволяют типам реагировать на операцию приведения (). Таким образом, если корректно сконфигурировать эти структуры, можно будет использовать следующий синтаксис для явного преобразования между ними:

```
// Преобразовать Rectangle в Square!
Rectangle rect;
rect.Width = 3;
rect.Height = 10;
Square sq = (Square)rect;
```

Создание специальных процедур преобразования

Начнем с создания нового консольного приложения по имени `CustomConversions`. В C# предусмотрены два ключевых слова, `explicit` и `implicit`, которые можно применять для управления реакцией на попытки выполнить преобразования. Предположим, что имеются следующие определения структур:

```
public struct Rectangle
{
    public int Width {get; set;}
    public int Height {get; set;}
```

```

public Rectangle(int w, int h) : this()
{
    Width = w; Height = h;
}

public void Draw()
{
    for (int i = 0; i < Height; i++)
    {
        for (int j = 0; j < Width; j++)
        {
            Console.Write("*");
        }
        Console.WriteLine();
    }
}

public override string ToString()
{
    return string.Format("[Width = {0}; Height = {1}]",
        Width, Height);
}
}

public struct Square
{
    public int Length {get; set;}
    public Square(int l) : this()
    {
        Length = l;
    }

    public void Draw()
    {
        for (int i = 0; i < Length; i++)
        {
            for (int j = 0; j < Length; j++)
            {
                Console.Write("*");
            }
            Console.WriteLine();
        }
    }

    public override string ToString()
    { return string.Format("[Length = {0}]", Length); }

    // Rectangle можно явно преобразовать в Square.
    public static explicit operator Square(Rectangle r)
    {
        Square s = new Square();
        s.Length = r.Height;
        return s;
    }
}

```

На заметку! Вы наверняка заметили в конструкторах `Square` и `Rectangle` связывание в цепочку со стандартным конструктором. Причина в том, что при наличии структуры, которая использует синтаксис автоматических свойств (как в данном случае), стандартный конструктор должен быть явно вызван (из всех специальных конструкторов) для инициализации закрытых поддерживающих полей. Да, это весьма своеобразное правило C#, но, в конце концов, ведь данная глава посвящена сложным темам.

Обратите внимание, что эта итерация типа `Square` определяет явную операцию преобразования. Подобно процессу перегрузки операций, процедуры преобразования используют ключевое слово `operator` в сочетании с ключевым словом `explicit` или `implicit` и должны быть определены как `static`. Входным параметром является сущность, из которой выполняется преобразование, в то время как тип операции — сущность, в которую оно производится.

В этом случае предполагается, что квадрат (геометрическая фигура с четырьмя равными сторонами) может быть получен из высоты прямоугольника. Таким образом, преобразовать `Rectangle` в `Square` можно следующим образом:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Conversions *****\n");
    // Создать Rectangle.
    Rectangle r = new Rectangle(15, 4);
    Console.WriteLine(r.ToString());
    r.Draw();
    Console.WriteLine();
    // Преобразовать r в Square на основе высоты Rectangle.
    Square s = (Square)r;
    Console.WriteLine(s.ToString());
    s.Draw();
    Console.ReadLine();
}
```

Ниже представлен вывод этой программы:

```
***** Fun with Conversions *****
[Width = 15; Height = 4]
*****
*****
*****
*****
[Length = 4]
****
****
****
****
```

Хотя, может быть, не слишком полезно преобразовывать `Rectangle` в `Square` в пределах одного контекста, предположим, что есть функция, спроектированная так, чтобы принимать параметров `Square`:

```
// Этот метод требует параметр типа Square.
static void DrawSquare(Square sq)
{
    Console.WriteLine(sq.ToString());
    sq.Draw();
}
```

Имея операцию явного преобразования в тип `Square`, теперь можно передавать типы `Rectangle` для обработки этому методу с использованием явного приведения:

```
static void Main(string[] args)
{
    ...
    // Преобразовать Rectangle в Square для вызова метода.
    Rectangle rect = new Rectangle(10, 5);
    DrawSquare((Square)rect);
    Console.ReadLine();
}
```

Дополнительные явные преобразования для типа Square

Теперь, когда можно явно преобразовывать объекты Rectangle в объекты Square, давайте рассмотрим несколько дополнительных явных преобразований. Учитывая, что квадрат симметричен по всем сторонам, может быть полезно предусмотреть процедуру преобразования, которая позволит вызывающему коду привести целочисленный тип к типу Square (который, разумеется, будет иметь длину стороны, равную переданному целому). Аналогично, что если вы захотите модифицировать Square так, чтобы вызывающий код мог выполнять приведение из Square в int? Логика вызова выглядит следующим образом:

```
static void Main(string[] args)
{
    ...
    // Преобразование int в Square.
    Square sq2 = (Square)90;
    Console.WriteLine("sq2 = {0}", sq2);

    // Преобразование Square в int.
    int side = (int)sq2;
    Console.WriteLine("Side length of sq2 = {0}", side);
    Console.ReadLine();
}
```

Ниже показаны необходимые изменения в структуре Square:

```
public struct Square
{
    ...

    public static explicit operator Square(int sideLength)
    {
        Square newSq = new Square();
        newSq.Length = sideLength;
        return newSq;
    }

    public static explicit operator int (Square s)
    {return s.Length;}
}
```

По правде говоря, преобразование Square в int может показаться не слишком интуитивно понятной (или полезной) операцией. Однако это указывает на один очень важный факт, касающийся процедур специальных преобразований: компилятор не волнует, что и куда преобразуется, до тех пор, пока пишется синтаксически корректный код.

Таким образом, как и с перегруженными операциями, возможность создания операций явного приведения еще не означает, что вы обязаны их создавать. Обычно этот прием наиболее полезен при создании типов структур .NET, учитывая, что они не могут участвовать в отношениях классического наследования (где приведение обеспечивается автоматически).

Определение процедур неявного преобразования

До сих пор мы создавали различные специальные операции *явного* преобразования. Однако что, если понадобится *неявное* преобразование?

```
static void Main(string[] args)
{
    ...
```

```

Square s3 = new Square();
s3.Length = 83;

// Попытка выполнить неявное приведение?
Rectangle rect2 = s3;

Console.ReadLine();
}

```

Этот код не скомпилируется, если для типа `Rectangle` не будет предусмотрена процедура неявного преобразования. Ловушка здесь вот в чем: не допускается иметь одновременно функции явного и неявного преобразования, если они не отличаются по типу возвращаемого значения или по списку параметров. Это может показаться ограничением, однако вторая ловушка состоит в том, что когда тип определяет процедуру неявного преобразования, никто не запретит вызывающему коду использовать синтаксис явного приведения!

Запутались? Для того чтобы прояснить ситуацию, давайте добавим к структуре `Rectangle` процедуру неявного преобразования, используя для этого ключевое слово `implicit` (обратите внимание, что в следующем коде предполагается, что ширина результирующего `Rectangle` вычисляется умножением стороны `Square` на 2):

```

public struct Rectangle
{
    ...
    public static implicit operator Rectangle(Square s)
    {
        Rectangle r = new Rectangle();
        r.Height = s.Length;

        // Предположим, что длина нового Rectangle
        // будет равна (Length x 2).
        r.Width = s.Length * 2;
        return r;
    }
}

```

После такой модификации можно будет выполнять преобразование между типами:

```

static void Main(string[] args)
{
    ...
    // Неявное преобразование работает!
    Square s3 = new Square();
    s3.Length = 7;
    Rectangle rect2 = s3;
    Console.WriteLine("rect2 = {0}", rect2);

    // Синтаксис явного преобразования также работает!
    Square s4 = new Square();
    s4.Length = 3;
    Rectangle rect3 = (Rectangle)s4;
    Console.WriteLine("rect3 = {0}", rect3);
    Console.ReadLine();
}

```

На этом рассмотрение определения операций специального преобразования завершено. Как и с перегруженными операциями, здесь следует помнить, что данный фрагмент синтаксиса представляет собой просто сокращенное обозначение “нормальных” функций-членов, и в этом смысле является необязательным. Однако в случае правильного применения специальные структуры могут использоваться более естественно, поскольку трактуются как настоящие типы классов, связанные наследованием.

Исходный код. Проект CustomConversions доступен в подкаталоге Chapter 11.

Понятие расширяющих методов

В .NET 3.5 появилась концепция *расширяющих методов*, которая позволила добавлять новые методы или свойства к классу либо структуре, не модифицируя напрямую исходный тип. Когда это может оказаться полезным? Рассмотрим следующие ситуации.

Первым делом, предположим, что имеется класс, находящийся в производстве. Со временем становится ясно, что этот класс должен поддерживать несколько новых членов. При модификации текущего определения класса напрямую возникает риск нарушения обратной совместимости со старыми кодовыми базами, которые использовали этот класс, поскольку они могут не скомпилироваться с последним улучшенным определением класса. Один из способов обеспечения обратной совместимости предусматривает создание нового класса, производного от существующего; однако тогда придется сопровождать два класса. А как все мы знаем, сопровождение кода является наиболее скучной частью деятельности разработчика программного обеспечения.

Теперь представим следующую ситуацию. Пусть имеется структура (или, скажем, запечатанный класс), к которой нужно добавить новые члены, чтобы она вела себя полиморфно в рамках системы. Поскольку структуры и запечатанные классы не могут быть расширены, единственный выбор заключается в том, чтобы добавить необходимые члены к типу, снова приводя к риску нарушения обратной совместимости!

За счет использования расширяющих методов появляется возможность модифицировать типы, не создавая подклассов и не изменяя код типа напрямую. По правде говоря, этот прием, в сущности, является иллюзией. Новая функциональность доступна типу только при условии, что в проекте будет присутствовать ссылка на эти расширяющие методы.

Определение расширяющих методов

При определении расширяющих методов первое ограничение состоит в том, что они должны быть определены внутри статического класса (см. главу 5), и потому каждый расширяющий метод должен быть объявлен с ключевым словом `static`. Второй момент заключается в том, что все расширяющие методы помечаются как таковые посредством ключевого слова `this` в виде модификатора первого (и только первого) параметра данного метода. Параметр, помеченный `this`, представляет расширяемый элемент.

В целях иллюстрации создадим новое консольное приложение по имени `ExtensionMethods`. Далее предположим, что строится класс по имени `MyExtensions`, в котором определены два расширяющих метода. Первый позволяет любому объекту пользоваться новым методом под названием `DisplayDefiningAssembly()`, который применяет типы из пространства имен `System.Reflection` для отображения имени сборки, содержащей данный тип.

На заметку! API-интерфейс рефлексии формально рассматривается в главе 15. Если эта тема является для вас новой, просто знайте, что рефлексия позволяет исследовать структуру сборок, типов и членов типов во время выполнения.

Второй расширяющий метод по имени `ReverseDigits()` позволяет любому `int` получить новую версию самого себя, при этом значение имеет обратный порядок следования цифр. Например, если вызвать `ReverseDigits()` на целом значении 1234, возвращенное целое значение будет равно 4321. Взгляните на следующую реализацию класса (не забудьте импортировать пространство имен `System.Reflection`):

```

static class MyExtensions
{
    // Этот метод позволяет любому объекту отобразить
    // сборку, в которой он определен.
    public static void DisplayDefiningAssembly(this object obj)
    {
        Console.WriteLine("{0} lives here: => {1}\n", obj.GetType().Name,
            Assembly.GetAssembly(obj.GetType()).GetName().Name);
    }

    // Этот метод позволяет любому целому изменить порядок следования
    // десятичных цифр на обратный. Например, 56 превратится в 65.
    public static int ReverseDigits(this int i)
    {
        // Транслировать int в string и затем получить все его символы.
        char[] digits = i.ToString().ToCharArray();

        // Изменить порядок элементов массива.
        Array.Reverse(digits);

        // Вставить обратно в строку.
        string newDigits = new string(digits);

        // Вернуть модифицированную строку как int.
        return int.Parse(newDigits);
    }
}

```

Обратите внимание, что первый параметр каждого расширяющего метода помечен ключевым словом `this`, указанным перед определением типа параметра. Первый параметр расширяющего метода всегда представляет расширяемый тип. Учитывая, что `DisplayDefiningAssembly()` прототипирован для расширения `System.Object`, этот новый член теперь имеется в каждом типе, поскольку `Object` является родителем всех типов в .NET. Однако `ReverseDigits()` прототипирован только для расширения целочисленных типов, и потому если что-то другое попытается вызвать этот метод, возникнет ошибка на этапе компиляции.

На заметку! Знайте, что каждый расширяющий метод может иметь множество параметров, но только первый параметр может быть помечен как `this`. Дополнительные параметры будут трактоваться как нормальные входные параметры, используемые методом.

Вызов расширяющих методов

Теперь, имея эти расширяющие методы, рассмотрим следующий метод `Main()`:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Extension Methods *****\n");

    // В int появилась новая идентичность!
    int myInt = 12345678;
    myInt.DisplayDefiningAssembly();

    // То же и у DataSet!
    System.Data.DataSet d = new System.Data.DataSet();
    d.DisplayDefiningAssembly();

    // И у SoundPlayer!
    System.Media.SoundPlayer sp = new System.Media.SoundPlayer();
    sp.DisplayDefiningAssembly();
}

```



```
// Использовать новую функциональность int.
Console.WriteLine("Value of myInt: {0}", myInt);
Console.WriteLine("Reversed digits of myInt: {0}", myInt.ReverseDigits());
Console.ReadLine();
}
```

Ниже показан вывод:

```
***** Fun with Extension Methods *****
Int32 lives here: => mscorlib
DataSet lives here: => System.Data
SoundPlayer lives here: => System
Value of myInt: 12345678
```

Импорт расширяющих методов

Когда определяется класс, содержащий расширяющие методы, он, несомненно, будет принадлежать какому-то пространству имен .NET. Если это пространство имен отличается от пространства имен, использующего расширяющие методы, необходимо применять ключевое слово `using` языка C#. Тогда файл кода будет иметь доступ ко всем расширяющим методам типа. Об этом следует помнить, поскольку если не импортировать явно корректное пространство имен, то расширяющие методы не будут доступны в таком файле кода C#.

Хотя на первый взгляд может показаться, что расширяющие методы глобальны по своей природе, на самом деле они ограничены пространствами имен, в которых они определены, или пространствами имен, которые их импортируют. Таким образом, если поместить класс `MyExtensions` в пространство имен `MyExtensionMethods`, как показано ниже:

```
namespace MyExtensionMethods
{
    static class MyExtensions
    {
        ...
    }
}
```

то другие пространства имен в проекте должны явно импортировать пространство `MyExtensionMethods` для получения расширяющих методов, определенных этим классом.

На заметку! Общепринятая практика предполагает изоляцию расширяющих методов не только в отдельном пространстве имен .NET, но также и в отдельной библиотеке классов. В таком случае новые приложения могут пользоваться расширениями путем явной ссылки на подходящую библиотеку. В главе 14 будут представлены подробности построения и использования специальных библиотек классов .NET.

Поддержка расширяющих методов средством IntelliSense

Учитывая тот факт, что расширяющие методы не определены буквально на расширяемом типе, при чтении кода есть шансы запутаться. Например, предположим, что имеется импортированное пространство имен, в котором определено несколько расширяющих методов, написанных кем-то из команды разработчиков. При написании своего кода вы создаете переменную расширенного типа, применяете операцию точки и обнаруживаете десятки новых методов, которые не являются членами исходного определения класса!

К счастью, средство IntelliSense в Visual Studio маркирует все расширяющие методы уникальным значком с изображением стрелки вниз (рис. 11.1).

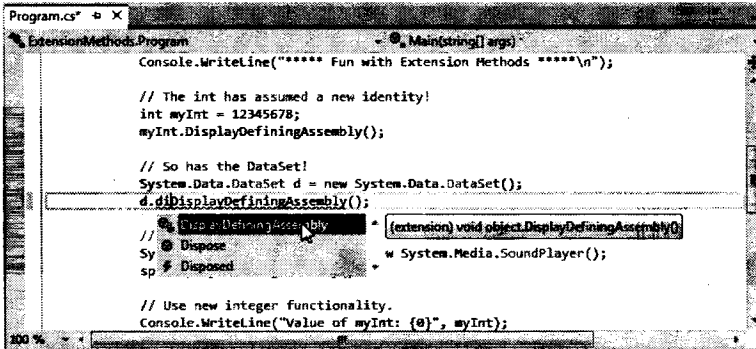


Рис. 11.1. Отображение расширяющих методов в IntelliSense

Если метод помечен этим значком, это означает, что он определен вне исходного определения класса, через механизм расширяющих методов.

Исходный код. Проект `ExtensionMethods` доступен в подкаталоге `Chapter 11`.

Расширение типов, реализующих специфичные интерфейсы

Итак, было показано, каким образом расширять классы (а также структуры, которые следуют тому же синтаксису) новой функциональностью через расширяющие методы. Также возможно определить расширяющий метод, который может расширять только класс или структуру, реализующую корректный интерфейс. Например, можно было бы заявить следующее: если класс или структура реализует `IEnumerable<T>`, то этот тип получит новые члены. Конечно, вполне допустимо требовать, чтобы тип поддерживал любой интерфейс вообще, включая ваши специальные интерфейсы.

В качестве примера создадим новое консольное приложение по имени `InterfaceExtensions`. Цель заключается в добавлении нового метода к любому типу, реализующему интерфейс `IEnumerable`, что будет включать все массивы и многие необобщенные классы коллекций (вспомните из главы 8, что обобщенный интерфейс `IEnumerable<T>` расширяет необобщенный интерфейс `IEnumerable`). Добавьте в новый проект приведенный ниже расширяющий класс:

```
static class AnnoyingExtensions
{
    public static void PrintDataAndBeep(this System.Collections.IEnumerable
iterator)
    {
        foreach (var item in iterator)
        {
            Console.WriteLine(item);
            Console.Beep();
        }
    }
}
```

Учитывая, что метод `PrintDataAndBeep()` может применяться любым классом или структурой, которая реализует интерфейс `IEnumerable`, мы могли бы проверить это с помощью следующего метода `Main()`:

```
static void Main( string[] args )
{
    Console.WriteLine("***** Extending Interface Compatible Types *****\n");
    // System.Array реализует IEnumerable!
    string[] data = { "Wow", "this", "is", "sort", "of", "annoying",
                     "but", "in", "a", "weird", "way", "fun!" };
    data.PrintDataAndBeep();
    Console.WriteLine();
    // List<T> реализует IEnumerable!
    List<int> myInts = new List<int>() {10, 15, 20};
    myInts.PrintDataAndBeep();
    Console.ReadLine();
}
```

На этом исследование расширяющих методов C# завершено. Помните, что это конкретное языковое средство может оказаться очень полезным, когда нужно расширить функциональность типа, не создавая подклассы (или если тип запечатан), в целях поддержания полиморфизма. Как будет показано позже, расширяющие методы играют ключевую роль в API-интерфейсах LINQ. На самом деле вы увидите, что в API-интерфейсах LINQ одним из наиболее часто расширяемых элементов является класс или структура, реализующая обобщенную версию интерфейса `IEnumerable`.

Исходный код. Проект `InterfaceExtension` доступен в подкаталоге `Chapter 11`.

Понятие анонимных типов

Как объектно-ориентированный программист, вы знаете преимущества определения классов для представления состояния и функциональности заданной сущности, которую вы пытаетесь смоделировать. То есть, когда нужно определить класс, который предполагает многократное использование и предоставляет обширную функциональность через набор методов, событий, свойств и специальных конструкторов, то разработка нового класса C# является общепринятой практикой.

Однако есть и другие случаи, когда может понадобиться определить класс просто для моделирования набора инкапсулированных (и каким-то образом связанных) элементов данных, без ассоциированных с ними методов, событий или другой специализированной функциональности. Более того, что если этот тип будет использоваться только небольшим набором методов внутри вашей программы? Было бы довольно утомительно приводить полное определение класса, как показано ниже, если хорошо известно, что этот класс будет применяться только в нескольких местах. Чтобы подчеркнуть этот момент, вот примерный план того, что придется делать, когда необходимо создать "простой" тип данных, который следует обычной семантике на основе значений:

```
class SomeClass
{
    // Определить набор закрытых переменных-членов...
    // Создать свойство для каждой закрытой переменной...
    // Переопределить ToString() для учета ключевых переменных-членов...
    // Переопределить GetHashCode() и Equals() для работы с эквивалентностью
    // на основе значений...
}
```

Как видите, это не обязательно будет настолько просто. Вам потребуется не только написать большой объем кода, но еще и сопровождать дополнительный класс в системе. Для временных данных подобного рода было бы полезно создавать специальный

тип данных на лету. Например, предположим, что необходимо построить специальный метод, который принимает набор входных параметров. Эти параметры нужно использовать для создания нового типа данных, который будет применяться в рамках метода. Кроме того, требуется иметь возможность быстрого вывода этих данных с помощью метода ToString() и работы с другими членами System.Object. Все это можно сделать с использованием синтаксиса анонимных типов.

Определение анонимного типа

Анонимный тип определяется с применением ключевого слова var (глава 3) в сочетании с синтаксисом инициализации объектов (глава 5). Ключевое слово var должно использоваться потому, что компилятор будет автоматически генерировать новое определение класса на этапе компиляции (и мы никогда не увидим имя этого класса в коде C#). Синтаксис инициализации применяется для сообщения компилятору о необходимости создания в новом построенном типе закрытых поддерживающих полей и (предназначенных только для чтения) свойств.

В целях иллюстрации создадим новое консольное приложение по имени AnonymousTypes. Затем добавим в класс Program показанный ниже новый метод, который формирует новый тип на лету, используя данные входного параметра:

```
static void BuildAnonType( string make, string color, int currSp )
{
    // Построить анонимный тип, используя входные аргументы.
    var car = new { Make = make, Color = color, Speed = currSp };

    // Обратите внимание, что теперь этот тип можно
    // использовать для получения данных свойств!
    Console.WriteLine("You have a {0} {1} going {2} MPH",
        car.Color, car.Make, car.Speed);

    // Анонимные типы имеют специальные реализации каждого
    // виртуального метода System.Object. Например:
    Console.WriteLine("ToString() == {0}", car.ToString());
}
```

Как и ожидалось, этот метод можно вызвать в Main(). Тем не менее, обратите внимание, что анонимный тип также может быть создан с применением жестко закодированных значений, как показано ниже:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Anonymous Types *****\n");

    // Создать анонимный тип, представляющий автомобиль.
    var myCar = new { Color = "Bright Pink", Make = "Saab", CurrentSpeed = 55 };

    // Вывести на консоль цвет и производителя.
    Console.WriteLine("My car is a {0} {1}.", myCar.Color, myCar.Make);

    // А теперь вызвать вспомогательный метод для построения
    // анонимного типа посредством аргументов.
    BuildAnonType("BMW", "Black", 90);

    Console.ReadLine();
}
```

Итак, к этому моменту достаточно понимать, что анонимные типы позволяют быстро моделировать “форму” данных с очень небольшими накладными расходами. Это не более чем способ построения нового типа данных на лету, который поддерживает базовую инкапсуляцию через свойства и действует в соответствии с семантикой, основанной на значениях. Чтобы понять последнее утверждение, давайте посмотрим, как

компилятор C# строит анонимные типы на этапе компиляции, и в особенности — как он переопределяет члены `System.Object`.

Внутреннее представление анонимных типов

Все анонимные типы автоматически наследуются от `System.Object` и потому поддерживают все члены, предоставленные этим базовым классом. Учитывая это, можно вызывать `ToString()`, `GetHashCode()`, `Equals()` или `GetType()` на неявно типизированном объекте `myCar`. Предположим, что в классе `Program` определена следующая статическая вспомогательная функция:

```
static void ReflectOverAnonymousType(object obj)
{
    Console.WriteLine("obj is an instance of: {0}", obj.GetType().Name);
    Console.WriteLine("Base class of {0} is {1}",
        obj.GetType().Name,
        obj.GetType().BaseType);
    Console.WriteLine("obj.ToString() == {0}", obj.ToString());
    Console.WriteLine("obj.GetHashCode() == {0}", obj.GetHashCode());
    Console.WriteLine();
}
```

Теперь вызовем этот метод в `Main()`, передав ему объект `myCar` в качестве параметра:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Anonymous Types *****\n");
    // Создать анонимный тип, представляющий автомобиль.
    var myCar = new {Color = "Bright Pink", Make = "Saab", CurrentSpeed = 55};
    // Отобразить то, что сгенерировал компилятор.
    ReflectOverAnonymousType(myCar);
    ...
    Console.ReadLine();
}
```

Вывод будет выглядеть примерно так:

```
***** Fun with Anonymous Types *****
obj is an instance of: <>f__AnonymousType0`3
Base class of <>f__AnonymousType0`3 is System.Object
obj.ToString() = { Color = Bright Pink, Make = Saab, CurrentSpeed = 55 }
obj.GetHashCode() = -439083487
```

Прежде всего, обратите внимание в этом примере, что объект `myCar` имеет тип `<>f__AnonymousType0`3` (конкретное имя типа может отличаться). Помните, что назначаемое типу имя полностью определяется компилятором и напрямую в коде C# недоступно.

Возможно, наиболее важно здесь то, что каждая пара “имя/значение”, определенная с использованием синтаксиса инициализации объектов, отображается на идентично именованное свойство, доступное только для чтения, и соответствующее закрытое поддерживающее поле, также предназначенное только для чтения. Следующий код C# примерно отражает сгенерированный компилятором класс, используемый для представления объекта `myCar` (который можно увидеть с помощью утилиты `ildasm.exe`):

```
internal sealed class <>f__AnonymousType0<<Color>j__TPar,
    <Make>j__TPar, <CurrentSpeed>j__TPar>
{
```

```
// Поля только для чтения.
private readonly <Color>j__TPar <Color>i__Field;
private readonly <CurrentSpeed>j__TPar <CurrentSpeed>i__Field;
private readonly <Make>j__TPar <Make>i__Field;

// Стандартный конструктор.
public <>f__AnonymousType0(<Color>j__TPar Color,
    <Make>j__TPar Make, <CurrentSpeed>j__TPar CurrentSpeed);

// Переопределенные методы.
public override bool Equals(object value);
public override int GetHashCode();
public override string ToString();

// Свойства только для чтения.
public <Color>j__TPar Color { get; }
public <CurrentSpeed>j__TPar CurrentSpeed { get; }
public <Make>j__TPar Make { get; }
}
```

Реализация методов ToString() и GetHashCode()

Все анонимные типы автоматически наследуются от System.Object и предоставляют переопределенные версии методов Equals(), GetHashCode() и ToString(). Реализация ToString() просто строит строку из каждой пары "имя/значение". Например:

```
public override string ToString()
{
    StringBuilder builder = new StringBuilder();
    builder.Append("{ Color = ");
    builder.Append(this.<Color>i__Field);
    builder.Append(", Make = ");
    builder.Append(this.<Make>i__Field);
    builder.Append(", CurrentSpeed = ");
    builder.Append(this.<CurrentSpeed>i__Field);
    builder.Append(" }");
    return builder.ToString();
}
```

Реализация GetHashCode() вычисляет хеш-значение, используя каждую переменную-член анонимного типа в качестве входной для типа System.Collections.Generic.EqualityComparer<T>. С применением этой реализации GetHashCode() два анонимных типа породят одинаковое хеш-значение тогда (и только тогда), когда они имеют одинаковый набор свойств, которым присвоены одинаковые значения. При такой реализации анонимные типы хорошо подходят для помещения в контейнер Hashtable.

Семантика эквивалентности анонимных типов

Хотя реализация переопределенных методов ToString() и GetHashCode() достаточно проста, реализация метода Equals() может вызвать вопросы. Например, если определено две переменных "анонимных автомобилей" с одинаковым набором пар "имя/значение", должны ли эти переменные трактоваться как эквивалентные? Чтобы увидеть результат такого сравнения, дополним класс Program следующим новым методом:

```
static void EqualityTest()
{
    // Создать два анонимных класса с идентичным набором пар "имя/значение".
    var firstCar = new { Color = "Bright Pink", Make = "Saab", CurrentSpeed = 55 };
    var secondCar = new { Color = "Bright Pink", Make = "Saab", CurrentSpeed = 55 };
};

// Считать ли их эквивалентными на основе использования Equals()?
```

```

if (firstCar.Equals(secondCar))
    Console.WriteLine("Same anonymous object!");           // один и тот же объект
else
    Console.WriteLine("Not the same anonymous object!");    // разные объекты
// Можно ли проверить их эквивалентность с помощью ==?
if (firstCar == secondCar)
    Console.WriteLine("Same anonymous object!");           // один и тот же объект
else
    Console.WriteLine("Not the same anonymous object!");    // разные объекты
// Имеют ли эти объекты в основе одинаковый тип?
if (firstCar.GetType().Name == secondCar.GetType().Name)
    Console.WriteLine("We are both the same type!");       // один и тот же тип
else
    Console.WriteLine("We are different types!");          // разные типы
// Отобразить все детали.
Console.WriteLine();
ReflectOverAnonymousType(firstCar);
ReflectOverAnonymousType(secondCar);
}

```

Ниже показан (несколько неожиданный) вывод, полученный в результате вызова этого метода в Main():

```

My car is a Bright Pink Saab.
You have a Black BMW going 90 MPH
ToString() == { Make = BMW, Color = Black, Speed = 90 }
Same anonymous object!
Not the same anonymous object!
We are both the same type!
obj is an instance of: <>f__AnonymousType0`3
Base class of <>f__AnonymousType0`3 is System.Object
obj.ToString() == { Color = Bright Pink, Make = Saab, CurrentSpeed = 55 }
obj.GetHashCode() == -439083487
obj is an instance of: <>f__AnonymousType0`3
Base class of <>f__AnonymousType0`3 is System.Object
obj.ToString() == { Color = Bright Pink, Make = Saab, CurrentSpeed = 55 }
obj.GetHashCode() == -439083487

```

После запуска этого тестового кода вы увидите, что первая проверка, при которой вызывается Equals(), возвращает true, и потому на консоль выводится сообщение "Same anonymous object!" (один и тот же анонимный объект). Причина в том, что сгенерированный компилятором метод Equals() при проверке эквивалентности использует семантику на основе значений (т.е. проверяет значения каждого поля сравниваемого объекта).

Однако вторая проверка, в которой используется операция равенства (==), приводит к выводу на консоль строки "Not the same anonymous object!" (разные анонимные объекты), что на первый взгляд выглядит несколько нелогично. Такой результат объясняется тем, что анонимные типы не получают перегруженной версии операций проверки равенства (== и !=). Поэтому при проверке эквивалентности объектов анонимных типов с использованием операций равенства C# (вместо метода Equals()) проверяются ссылки, а не значения, поддерживаемые объектами.

И последнее (по порядку, но не по важности): финальная проверка (где проверяется имя лежащего в основе типа) показывает, что экземпляры анонимных типов относятся к одному и тому же сгенерированному компилятором типу класса (в рассматриваемом примере это <>f__AnonymousType0`3), потому что firstCar и secondCar имеют одинаковый набор свойств (Color, Make и CurrentSpeed).

Это иллюстрирует важный, но тонкий момент: компилятор генерирует определение нового класса тогда, когда анонимный тип имеет *уникальные* имена свойств. Поэтому в случае объявления идентичных анонимных типов (в смысле — с одинаковыми именами) в одной сборке компилятор сгенерирует только одно определение анонимного типа.

Анонимные типы, содержащие другие анонимные типы

Можно создавать анонимные типы, состоящие из других анонимных типов. Например, предположим, что необходимо смоделировать заказ на покупку, который состоит из временной метки, цены и приобретаемого автомобиля. Ниже показан новый (несколько более сложный) анонимный тип, представляющий эту сущность:

```
// Создать анонимный тип, состоящий из другого анонимного типа.
var purchaseItem = new {
    TimeBought = DateTime.Now,
    ItemBought = new {Color = "Red", Make = "Saab", CurrentSpeed = 55},
    Price = 34.000};

ReflectOverAnonymousType(purchaseItem);
```

К этому моменту синтаксис, используемый для определения анонимных типов, должен быть понятным, но, скорее всего, остался главный вопрос: где и когда может понадобиться это новое языковое средство? Если кратко, то объявления анонимных типов следует применять сдержанно, обычно только в сочетании с набором технологий LINQ (глава 12). Никогда не отказывайтесь от использования строго типизированных классов и структур просто потому, что это возможно, учитывая многочисленные ограничения анонимных типов, которые перечислены ниже.

1. Контроль над именами анонимных типов отсутствует.
2. Анонимные типы всегда расширяют `System.Object`.
3. Поля и свойства анонимного типа всегда доступны только для чтения.
4. Анонимные типы не могут поддерживать события, специальные методы, специальные операции или специальные переопределения.
5. Анонимные типы всегда неявно запечатаны.
6. Экземпляры анонимных типов всегда создаются с применением стандартных конструкторов.

Тем не менее, при программировании с использованием набора технологий LINQ вы обнаружите, что во многих случаях этот синтаксис оказывается очень полезным, когда нужно быстро смоделировать общую *форму* сущности, а не ее функциональность.

Исходный код. Проект `AnonymousTypes` доступен в подкаталоге `Chapter 11`.

Работа с типами указателей

Последняя тема настоящей главы касается средства C#, которое наименее часто используется в подавляющем большинстве проектов.

На заметку! В последующих примерах предполагается наличие у вас определенных знаний о работе с указателями в C (C++). Если это не так, можете спокойно пропустить данную тему. В подавляющем большинстве приложений C# указатели не используются.

В главе 4 вы узнали, что в .NET определены две основных категории типов данных: типы значений и ссылочные типы. На самом деле существует еще и третья категория: *типы указателей*. Для работы с типами указателей доступны специфические ключевые слова и операции, описанные в табл. 11.2, которые позволяют обойти схему управления памятью CLR и взять бразды правления в собственные руки.

Таблица 11.2. Операции и ключевые слова C#, связанные с указателями

Операция/ ключевое слово	Назначение
*	Эта операция используется для создания переменной-указателя (т.е. переменной, представляющей непосредственное местоположение в памяти). Как и в C (C++), та же самая операция служит для разыменования указателя
&	Эта операция используется для получения адреса переменной в памяти
->	Эта операция используется для доступа к полям типа, представленного указателем (небезопасной версией операции точки C#)
[]	Операция [] (в небезопасном контексте) позволяет индексировать участок, заданный переменной-указателем (вспомните взаимосвязь между переменной-указателем и операцией [] в C (C++))
++, --	В небезопасном контексте операции инкремента и декремента могут применяться к типам указателей
+, -	В небезопасном контексте операции сложения и вычитания могут применяться к типам указателей
==, !=, <, >, <=, >=	В небезопасном контексте операции сравнения и эквивалентности могут применяться к типам указателей
stackalloc	В небезопасном контексте ключевое слово <code>stackalloc</code> может быть использовано для размещения массивов C# непосредственно в стеке
fixed	В небезопасном контексте ключевое слово <code>fixed</code> может быть использовано для временного закрепления переменной, чтобы можно было впоследствии найти ее адрес

Перед погружением в детали следует учесть, что вам очень редко, если вообще когда-нибудь, понадобится использовать типы указателей. Хотя C# позволяет перейти на уровень прямых манипуляций указателями, знайте, что исполняющая среда .NET не имеет абсолютно никакого представления о ваших намерениях. Поэтому, если вы произведете неверное действие с указателем, то сами будете отвечать за последствия. С учетом этого предупреждения возникает вопрос: когда вообще может понадобиться работа с типами указателей? Существуют только две таких ситуации.

- Необходимо оптимизировать некоторые части приложения, напрямую обращаясь к его членам за рамками управления CLR.
- Требуется вызывать методы из динамической библиотеки (* .dll), написанной на C, или же из сервера COM, который требует в качестве параметров типы указателей. Но даже в этом случае часто можно обойтись без применения типов указателей, отдав предпочтение типу `System.IntPtr` и членам типа `System.Runtime.InteropServices.Marshal`.

В случае если все же решено пользоваться этим средством языка C#, понадобится информировать компилятор C# (`csc.exe`) о своих намерениях, позволив проекту поддерживать “небезопасный” код.

Чтобы сделать это в командной строке, добавьте при вызове компилятора флаг /unsafe:

```
csc /unsafe *.cs
```

В среде Visual Studio нужно будет перейти на страницу Properties (Свойства) проекта и на вкладке Build (Сборка) отметить флажок Allow Unsafe Code (Разрешить небезопасный код), как показано на рис. 11.2. Чтобы поэкспериментировать с типами указателей, создадим новое консольное приложение по имени UnsafeCode и разрешим небезопасный код.

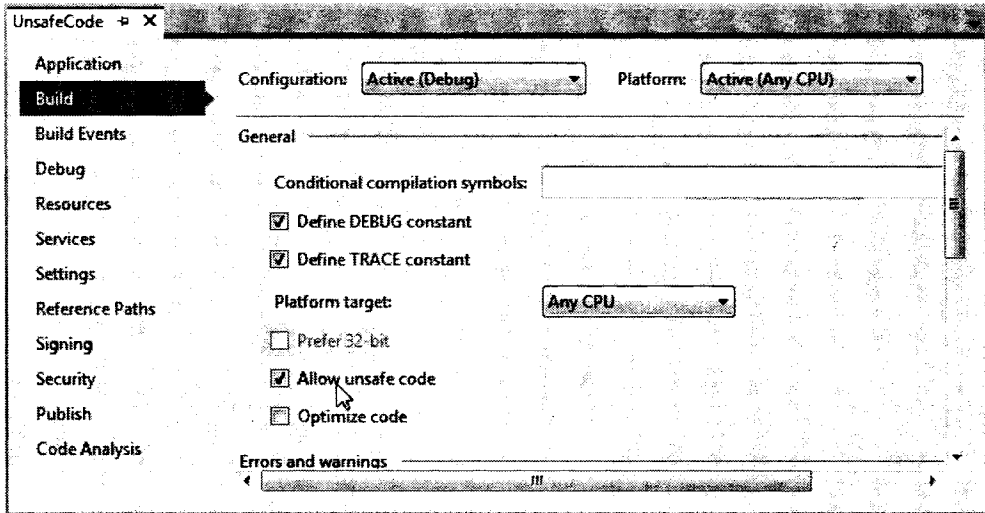


Рис. 11.2. Включение поддержки небезопасного кода в Visual Studio

Ключевое слово unsafe

Если необходимо работать с указателями в C#, следует специально объявить блок “небезопасного” кода с помощью ключевого слова `unsafe` (любой код, который не помечен ключевым словом `unsafe`, автоматически считается “безопасным”). Например, в следующем классе `Program` объявляется область небезопасного кода внутри метода `Main()`:

```
class Program
{
    static void Main(string[] args)
    {
        unsafe
        {
            // Здесь работаем с указателями!
        }
        // Здесь нельзя работать с указателями!
    }
}
```

В дополнение к объявлению области небезопасного кода внутри метода, можно строить “небезопасные” структуры, классы, члены типов и параметры. Ниже приведено несколько примеров (тип `Node` или `Node2` в текущем проекте определять не нужно):

```
// Вся эта структура небезопасна и может
// использоваться только в небезопасном контексте.
```

```

unsafe struct Node
{
    public int Value;
    public Node* Left;
    public Node* Right;
}

// Структура безопасна, но члены Node2* - нет.
// Формально извне небезопасного контекста можно
// обращаться к Value, но не к Left и Right.
public struct Node2
{
    public int Value;

    // Это доступно только из небезопасного контекста!
    public unsafe Node2* Left;
    public unsafe Node2* Right;
}

```

Методы (статические и уровня экземпляра) также могут быть помечены как небезопасные. Предположим, известно, что определенный статический метод будет использовать логику указателей. Чтобы обеспечить возможность вызова этого метода только из небезопасного контекста, данный метод можно определить следующим образом:

```

unsafe static void SquareIntPtr(int* myIntPtr)
{
    // Возвести значение в квадрат - просто в целях тестирования.
    *myIntPtr *= *myIntPtr;
}

```

Конфигурация метода требует, чтобы вызывающий код обращался к методу `SquareIntPtr()`, как показано ниже:

```

static void Main(string[] args)
{
    unsafe
    {
        int myInt = 10;
        // Нормально, мы в небезопасном контексте.
        SquareIntPtr(&myInt);
        Console.WriteLine("myInt: {0}", myInt);
    }

    int myInt2 = 5;
    // Ошибка компиляции! Должен быть небезопасный контекст!
    SquareIntPtr(&myInt2);
    Console.WriteLine("myInt: {0}", myInt2);
}

```

Если вы не хотите заставлять вызывающий код помещать этот вызов в оболочку небезопасного контекста, можете пометить весь метод `Main()` ключевым словом `unsafe`. В таком случае приведенный ниже код скомпилируется:

```

unsafe static void Main(string[] args)
{
    int myInt2 = 5;
    SquareIntPtr(&myInt2);
    Console.WriteLine("myInt: {0}", myInt2);
}

```

Запустив на выполнение этот метод `Main()`, вы увидите следующий вывод:

```
myInt: 25
```

Работа с операциями * и &

Установив небезопасный контекст, можно строить указатели и типы данных, использующие операцию *, а также получать адрес указателя с помощью операции &. В отличие от C или C++, в языке C# операция * применяется только к лежащему в основе типу, а не является префиксом имени каждой переменной указателя. Например, рассмотрим следующий код, который иллюстрирует правильный и неправильный способы объявления указателей на целочисленные переменные:

```
// Нет! В C# это неправильно!
int *pi, *pj;

// Да! Так правильно в C#.
int* pi, pj;
```

Рассмотрим следующий небезопасный метод:

```
unsafe static void PrintValueAndAddress()
{
    int myInt;
    // Определить указатель на int и присвоить ему адрес myInt.
    int* ptrToMyInt = &myInt;

    // Присвоить значение myInt, используя обращение через указатель.
    *ptrToMyInt = 123;

    // Вывести на консоль некоторые значения.
    Console.WriteLine("Value of myInt {0}", myInt); // значение myInt
    Console.WriteLine("Address of myInt {0:X}", (int)&ptrToMyInt); // адрес myInt
}
```

Небезопасная (и безопасная) функция обмена

Разумеется, объявление указателей на локальные переменные только для того, чтобы присвоить им значения (как в предыдущем примере), никогда не понадобится и вообще неудобно. Чтобы проиллюстрировать более практичный пример небезопасного кода, предположим, что нужно построить функцию обмена с использованием арифметики указателей:

```
unsafe public static void UnsafeSwap(int* i, int* j)
{
    int temp = *i;
    *i = *j;
    *j = temp;
}
```

Очень похоже на язык C, не правда ли? Однако на основе уже имеющихся знаний вы в состоянии написать следующую безопасную версию алгоритма обмена с применением ключевого слова `ref`:

```
public static void SafeSwap(ref int i, ref int j)
{
    int temp = i;
    i = j;
    j = temp;
}
```

Функциональность обеих версий метода идентична, а это доказывает, что прямые манипуляции указателями в C# вовсе не обязательны. Ниже показана логика вызова с использованием безопасного метода `Main()`, но с небезопасным контекстом:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Calling method with unsafe code *****");
    // Значения, подлежащие обмену.
    int i = 10, j = 20;

    // "Безопасный" обмен значений.
    Console.WriteLine("\n***** Safe swap *****");
    Console.WriteLine("Values before safe swap: i = {0}, j = {1}", i, j);
    SafeSwap(ref i, ref j);
    Console.WriteLine("Values after safe swap: i = {0}, j = {1}", i, j);

    // "Небезопасный" обмен значений.
    Console.WriteLine("\n***** Unsafe swap *****");
    Console.WriteLine("Values before unsafe swap: i = {0}, j = {1}", i, j);
    unsafe { UnsafeSwap(&i, &j); }
    Console.WriteLine("Values after unsafe swap: i = {0}, j = {1}", i, j);
    Console.ReadLine();
}
```

Доступ к полям через указатели (операция ->)

Теперь предположим, что определена простая безопасная структура Point:

```
struct Point
{
    public int x;
    public int y;
    public override string ToString()
    {
        return string.Format("{0}, {1}", x, y);
    }
}
```

При объявлении указателя на тип Point для доступа к открытым членам структуры понадобится применять операцию доступа к полю (в виде ->). Как показано в табл. 11.3, это — небезопасная версия стандартной (безопасной) операции точки (.). Фактически, используя операцию обращения к указателю (*), можно разыменовать указатель и вновь применить операцию точки. Рассмотрим следующий небезопасный метод:

```
unsafe static void UsePointerToPoint()
{
    // Доступ к членам через указатель.
    Point point;
    Point* p = &point;
    p->x = 100;
    p->y = 200;
    Console.WriteLine(p->ToString());

    // Доступ к членам через разыменованный указатель.
    Point point2;
    Point* p2 = &point2;
    (*p2).x = 100;
    (*p2).y = 200;
    Console.WriteLine((*p2).ToString());
}
```

Ключевое слово `stackalloc`

В небезопасном контексте может понадобиться объявить локальную переменную, память для которой выделяется непосредственно в стеке вызовов (и потому она не доступна для системы сборки мусора .NET). Для этого в C# предусмотрено ключевое слово `stackalloc`, которое является эквивалентом C# функции `_alloca` библиотеки времени выполнения C. Ниже приведен простой пример:

```
unsafe static void UnsafeStackAlloc()
{
    char* p = stackalloc char[256];
    for (int k = 0; k < 256; k++)
        p[k] = (char)k;
}
```

Закрепление типа с помощью ключевого слова `fixed`

Как было показано в предыдущем примере, выделение фрагмента памяти в пределах небезопасного контекста может быть осуществлено с помощью ключевого слова `stackalloc`. Ввиду природы этой операции, выделенная память очищается, как только метод, который ее выделил, возвращает управление (поскольку память распределена в стеке). Однако рассмотрим более сложный пример. Во время экспериментов с операцией `->` был создан тип значения по имени `Point`. Подобно всем типам значений, выделенная его экземплярам память выталкивается из стека, как только завершается контекст выполнения. Предположим, что вместо этого структура `Point` определена как **ссылочный тип**:

```
class PointRef // Переименовано и переписано.
{
    public int x;
    public int y;
    public override string ToString()
    {
        return string.Format("{0}, {1}", x, y);
    }
}
```

Как вам хорошо известно, если теперь в вызывающем коде объявить переменную типа `Point`, то память будет выделена в куче, подверженной сборке мусора. И тут возникает животрепещущий вопрос: а что если небезопасный контекст пожелает взаимодействовать с этим объектом (или любым другим объектом из кучи)? Учитывая, что сборка мусора может произойти в любой момент, представьте проблему, с которой вы столкнетесь при обращении к членам `Point` в тот момент, когда происходит реорганизация кучи. Теоретически может случиться так, что небезопасный контекст попытается взаимодействовать с членом, который уже недоступен или был перемещен в куче после ее реорганизации (что является очевидной проблемой).

Для фиксации переменной ссылочного типа в памяти из небезопасного контекста в C# предусмотрено ключевое слово `fixed`. Оператор `fixed` устанавливает указатель на управляемый тип и закрепляет эту переменную на время выполнения оператора. Без `fixed` от указателей на управляемые переменные было бы мало пользы, поскольку сборка мусора может перемещать переменные в памяти непредсказуемым образом. (На самом деле компилятор C# не позволит установить указатель на управляемую переменную без оператора `fixed`.)

Таким образом, если вы создадите экземпляр тип `Point` (теперь в виде класса) и захотите взаимодействовать с его членами, то должны будете написать следующий код (или, в противном случае, получить ошибку на этапе компиляции):

```
unsafe public static void UseAndPinPoint()
{
    PointRef pt = new PointRef ();
    pt.x = 5;
    pt.y = 6;

    // Закрепить указатель pt на месте, чтобы он не мог быть
    // перемещен или собран сборщиком мусора.
    fixed (int* p = &pt.x)
    {
        // Использовать здесь переменную int*!
    }

    // Указатель pt теперь не закреплен и готов к сборке мусора.
    Console.WriteLine ("Point is: {0}", pt);
}
```

По сути, ключевое слово `fixed` позволяет строить оператор, блокирующий ссылочную переменную в памяти, чтобы ее адрес оставался постоянным на протяжении работы оператора. Поэтому, взаимодействуя со ссылочным типом из контекста небезопасного кода, нужно обязательно закреплять ссылку.

Ключевое слово `sizeof`

Последнее ключевое слово C#, связанное с небезопасным кодом — это `sizeof`. Как и в языке C (C++), ключевое слово `sizeof` в C# служит для получения размера в байтах типа значения (но никогда — ссылочного типа), и может применяться только внутри небезопасного контекста. Как и несложно было представить, эта возможность может пригодиться при взаимодействии с неуправляемыми API-интерфейсами на базе C. Его применение очевидно:

```
unsafe static void UseSizeOfOperator()
{
    Console.WriteLine("The size of short is {0}.", sizeof(short)); // размер short
    Console.WriteLine("The size of int is {0}.", sizeof(int));      // размер int
    Console.WriteLine("The size of long is {0}.", sizeof(long));   // размер long
}
```

Поскольку `sizeof` вычисляет количество байт для любой сущности, унаследованной от `System.ValueType`, также можно получить размер специальных структур. Например, передать структуру `Point` в `sizeof` можно следующим образом:

```
unsafe static void UseSizeOfOperator()
{
    ...
    Console.WriteLine("The size of Point is {0}.", sizeof(Point)); // размер Point
}
```

Исходный код. Проект `UnsafeCode` доступен в подкаталоге `Chapter 11`.

На этом обзор более сложных средств языка программирования C# завершен. Напоследок важно отметить, что в большей части проектов .NET эти средства могут вообще не понадобиться (особенно указатели). Тем не менее, как будет показано в последующих главах, некоторые средства исключительно важны при работе с API-интерфейсами `LANG`, особенно это касается расширяющих методов и анонимных типов.

Резюме

Целью этой главы было углубление знаний языка программирования C#. Мы начали с исследования применений различных сложных конструкций (методов-индексаторов, перегруженных операций и процедур специального преобразования типов).

Затем была рассмотрена роль расширяющих методов и анонимных типов. Как будет показано в следующей главе, эти средства очень полезны при работе с API-интерфейсами LINQ (хотя их можно использовать повсюду в коде, если это покажется удобным). Вспомните, что анонимные методы позволяют быстро моделировать “форму” типа, в то время как расширяющие методы позволяют добавлять новую функциональность к типам, без необходимости в определении подклассов.

Финальная часть главы была посвящена рассмотрению небольшого набора малоизвестных ключевых слов (`sizeof`, `checked`, `unsafe` и т.п.), а попутно была продемонстрирована работа с низкоуровневыми типами указателей. Как было установлено в процессе рассмотрения этих типов, в большинстве приложений C# *никогда* не понадобится их использовать.

LINQ to Objects

Независимо от типа приложения, которое вы создаете с использованием платформы .NET, ваша программа определенно нуждается в доступе к некоторой форме данных в процессе выполнения. Данные могут находиться в самых разных местах, включая файлы XML, реляционные базы данных, коллекции в памяти и элементарные массивы. Исторически сложилось так, что в зависимости от места хранения данных программистам приходилось использовать очень разные и никак не связанные API-интерфейсы. Набор технологий LINQ (Language Integrated Query — язык интегрированных запросов), появившийся в .NET 3.5, предоставил краткий, симметричный и строго типизированный способ доступа к широкому разнообразию хранилищ данных. Изучение LINQ начинается в этой главе с рассмотрения LINQ to Objects.

Прежде чем погрузиться в LINQ to Objects, в первой части этой главы мы кратко рассмотрим ключевые программные конструкции C#, которые обеспечили возможность существования LINQ. По мере чтения главы вы убедитесь, насколько полезны (а иногда и обязательны) такие средства, как неявно типизированные переменные, синтаксис инициализации объектов, лямбда-выражения, расширяющие методы и анонимные типы.

После ознакомления с поддерживающей инфраструктурой в оставшемся материале главы будет представлена модель программирования LINQ и объяснена роль, которую она играет в рамках платформы .NET. Вы узнаете назначение операций и выражений запросов, которые позволяют определять операторы, опрашивающие источник данных для выдачи запрошенного результирующего набора. Попутно будут рассмотрены многочисленные примеры применения LINQ, которые иллюстрируют взаимодействие с данными в массивах и коллекциях различного типа (как обобщенных, так и необобщенных), а также сборки, пространства имен и типы, представляющие API-интерфейс LINQ to Objects.

На заметку! Предлагаемые в этой главе сведения послужат фундаментом для освоения последующих глав книги, в которых описаны дополнительные технологии LINQ, включая LINQ to XML (глава 24), Parallel LINQ (глава 19) и LINQ to Entities (глава 23).

Программные конструкции, специфичные для LINQ

На самом высоком уровне LINQ можно воспринимать как строго типизированный язык запросов, встроенный непосредственно в грамматику самого языка C#. Используя LINQ, можно строить любое количество выражений, которые выглядят и ведут себя подобно SQL-запросам к базе данных. Однако запрос LINQ может применяться к любому числу хранилищ данных, включая хранилища, которые не имеют ничего общего с истинными реляционными базами данных.

На заметку! Хотя запросы LINQ внешне похожи на запросы SQL, их синтаксис *не* идентичен. На самом деле многие запросы LINQ выглядят прямой противоположностью формата запроса к базе данных! Если вы попытаетесь отобразить LINQ непосредственно на SQL, то определенно запутаетесь. Чтобы этого не случилось, рекомендуется воспринимать запросы LINQ как уникальные операторы, которые лишь случайно похожи на SQL.

Когда LINQ впервые был представлен в составе платформы .NET 3.5, языки C# и VB уже были снабжены огромным количеством программных конструкций для поддержки набора технологий LINQ. В частности, язык C# использует следующие связанные с LINQ средства:

- неявно типизированные локальные переменные;
- синтаксис инициализации объектов и коллекций;
- лямбда-выражения;
- расширяющие методы;
- анонимные типы.

Эти средства уже детально рассматривались в различных главах настоящей книги. Однако чтобы освежить все в памяти, давайте быстро вспомним о каждом из средств по очереди, просто чтобы удостовериться в правильном их понимании.

Неявная типизация локальных переменных

В главе 3 вы узнали о ключевом слове `var` в языке C#. Это ключевое слово позволяет определять локальную переменную без явного указания для нее типа данных. Тем не менее, такая переменная будет строго типизированной, поскольку компилятор определит ее корректный тип данных на основе начального присваивания. Вспомните следующий код примера из главы 3:

```
static void DeclareImplicitVars()
{
    // Неявно типизированные локальные переменные.
    var myInt = 0;
    var myBool = true;
    var myString = "Time, marches on...";

    // Вывод имен типов, лежащих в основе этих переменных.
    Console.WriteLine("myInt is a: {0}", myInt.GetType().Name);
    Console.WriteLine("myBool is a: {0}", myBool.GetType().Name);
    Console.WriteLine("myString is a: {0}", myString.GetType().Name);
}
```

Это средство языка очень удобно и зачастую обязательно, когда используется LINQ. Как будет показано на протяжении главы, многие запросы LINQ будут возвращать последовательности типов данных, которые неизвестны к моменту компиляции. Учитывая, что лежащий в основе тип данных до компиляции приложения не известен, очевидно, что объявить такую переменную явно не удастся!

Синтаксис инициализации объектов и коллекций

В главе 5 объяснялась роль синтаксиса инициализации объектов, который позволяет создать переменную типа класса или структуры и установить любое количество ее открытых свойств за один прием. В результате получается очень компактный (и легко читаемый) синтаксис, который может применяться для подготовки объектов к использованию. Также вспомните из главы 9, что язык C# поддерживает очень похожий син-

таксис инициализации коллекций объектов. Взгляните на следующий фрагмент кода, где синтаксис инициализации коллекций используется для наполнения `List<T>` объектами `Rectangle`, каждый из которых состоит из пары объектов `Point`, представляющих точку с координатами (x, y):

```
List<Rectangle> myListOfRects = new List<Rectangle>
{
    new Rectangle {TopLeft = new Point { X = 10, Y = 10 },
                  BottomRight = new Point { X = 200, Y = 200}},
    new Rectangle {TopLeft = new Point { X = 2, Y = 2 },
                  BottomRight = new Point { X = 100, Y = 100}},
    new Rectangle {TopLeft = new Point { X = 5, Y = 5 },
                  BottomRight = new Point { X = 90, Y = 75}}
};
```

Хотя ничто не заставляет применять синтаксис инициализации коллекции или объекта, с его помощью можно получить более компактную кодовую базу. Более того, этот синтаксис в сочетании с неявной типизацией локальных переменных позволяет объявлять анонимный тип, что очень удобно при создании проекций LINQ. О проекциях LINQ речь пойдет далее в этой главе.

Лямбда-выражения

Лямбда-операция C# (`=>`) была уже полностью описана в главе 10. Вспомните, что эта операция позволяет строить лямбда-выражение, которое может быть использовано в любой момент, когда вызывается метод, требующий строго типизированного делегата в качестве аргумента. Лямбда-выражения значительно упрощают работу с делегатами .NET, поскольку сокращают объем кода, который должен быть написан вручную. Лямбда-выражения могут быть описаны следующим образом:

```
( АргументыДляОбработки ) => { ОбработывающиеИхОператоры }
```

В главе 10 рассматривались способы взаимодействия с методом `FindAll()` обобщенного класса `List<T>` с применением различных подходов. После работы с простым делегатом `Predicate<T>` и анонимным методом C# мы пришли к следующей (исключительно краткой) итерации, в которой использовалось следующее лямбда-выражение:

```
static void LambdaExpressionSyntax()
{
    // Создать список целых.
    List<int> list = new List<int>();
    list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });

    // Лямбда-выражение C#.
    List<int> evenNumbers = list.FindAll(i => (i % 2) == 0);

    // Вывод на консоль четных чисел.
    Console.WriteLine("Here are your even numbers:");
    foreach (int evenNumber in evenNumbers)
    {
        Console.Write("{0}\t", evenNumber);
    }
    Console.WriteLine();
}
```

Лямбда-выражения очень полезны при работе с объектной моделью, лежащей в основе LINQ. Как вы вскоре убедитесь, операции запросов C# LINQ — это просто сокращенная нотация вызова методов класса по имени `System.Linq.Enumerable`. Эти методы обычно всегда требуют передачи в качестве параметров делегатов (в частности, делегата `Func<>`), которые применяются для обработки данных с целью получения кор-

ректного результирующего набора. За счет использования лямбда-выражений можно упростить код и позволить компилятору вывести необходимый делегат.

Расширяющие методы

Расширяющие методы C# позволяют добавлять новую функциональность к существующим классам без необходимости применять наследование. Кроме того, расширяющие методы позволяют добавлять новую функциональность к запечатанным классам и структурам, от которых просто невозможно наследовать производные классы. Вспомните из главы 11, в которой создавался расширяющий метод, что первый параметр такого метода снабжается ключевым словом `this` и отмечает расширяемый тип. Кроме того, расширяющие методы должны всегда определяться внутри статического класса, а потому объявляться с применением ключевого слова `static`. Вот пример:

```
namespace MyExtensions
{
    static class ObjectExtensions
    {
        // Определение расширяющего метода для System.Object.
        public static void DisplayDefiningAssembly(this object obj)
        {
            Console.WriteLine("{0} lives here:\n\t->{1}\n", obj.GetType().Name,
                Assembly.GetAssembly(obj.GetType()));
        }
    }
}
```

Чтобы использовать это расширение, приложение должно импортировать пространство имен, в котором определено расширение (и возможно установить ссылку на внешнюю сборку). После этого можно просто импортировать определенное пространство имен и написать следующий код:

```
static void Main(string[] args)
{
    // Поскольку все расширяет System.Object, все классы и структуры
    // могут использовать это расширение.
    int myInt = 12345678;
    myInt.DisplayDefiningAssembly();

    System.Data.DataSet d = new System.Data.DataSet();
    d.DisplayDefiningAssembly();
    Console.ReadLine();
}
```

При работе с LINQ вам редко придется строить собственные расширяющие методы, если вообще придется. Однако при создании выражений запросов LINQ на самом деле применяются многочисленные расширяющие методы, уже определенные Microsoft. Фактически каждая операция запроса C# LINQ — это сокращенная нотация ручного вызова лежащего в основе расширяющего метода, обычно определенного служебным классом `System.Linq.Enumerable`.

Анонимные типы

И последнее средство языка C#, которое мы здесь кратко пересмотрим — анонимные типы, которые были описаны в главе 11. Это средство может использоваться для быстрого моделирования “формы” данных, позволяя компилятору генерировать новое определение класса во время компиляции на основе указанного набора пар “имя/значение”. Вспомните, что этот тип будет составлен с применением семантики, основанной

на значениях, и каждый виртуальный метод `System.Object` будет соответствующим образом переопределен. Чтобы определить анонимный тип, необходимо объявить неявно типизированную переменную и указать форму данных с применением синтаксиса инициализации объекта:

```
// Создать анонимный тип, состоящий из другого анонимного типа.
var purchaseItem = new {
    TimeBought = DateTime.Now,
    ItemBought = new {Color = "Red", Make = "Saab", CurrentSpeed = 55},
    Price = 34.000};
```

В LINQ анонимные типы часто используются при проецировании новых форм данных “на лету”. Например, может существовать коллекция объектов `Person`, и нужно с помощью LINQ получить информацию о возрасте и номере карточки социального страхования для каждого объекта. Используя проекцию LINQ, можно позволить компилятору сгенерировать новый анонимный тип, содержащий необходимую информацию.

Роль LINQ

На этом краткий обзор средств языка C#, позволяющих LINQ выполнять свою работу, завершен. Однако важно понять, зачем вообще нужен язык LINQ. Любой разработчик программного обеспечения согласится с утверждением, что значительная часть времени при программировании тратится на получение и манипулирование данными. Когда говорят о “данных”, немедленно приходит на ум информация, хранящаяся внутри реляционных баз данных. Тем не менее, другими популярными местами нахождения данных являются документы XML (файлы *.config, локально сохраненные наборы DataSet или данные в памяти, возвращенные службами WCF).

Данные могут быть найдены в различных местах и помимо этих двух общепринятых хранилищ информации. Например, предположим, что имеется массив или обобщенный тип `List<T>`, содержащий 300 целых чисел, и нужно получить подмножество, которое отвечает заданному критерию (например, только четные или только нечетные числа, только простые числа, только неповторяющиеся числа больше 50). Или, возможно, при использовании API-интерфейсов рефлексии необходимо получить в массиве элементов `Type` только метаданные для каждого класса, унаследованного от определенного родительского класса. В действительности данные находятся повсюду.

До появления .NET 3.5 взаимодействие с определенной разновидностью данных требовало от программистов применения очень разных API-интерфейсов. Например, в табл. 12.1 описаны некоторые распространенные API-интерфейсы, используемые для доступа к различным типам данных.

Таблица 12.1. Способы манипулирования различными типами данных

Необходимые данные	Как получить
Реляционные данные	<code>System.Data.dll</code> , <code>System.Data.SqlClient.dll</code> и т.д.
Данные документов XML	<code>System.Xml.dll</code>
Таблицы метаданных	Пространство имен <code>System.Reflection</code>
Коллекции объектов	Пространства имен <code>System.Array</code> и <code>System.Collections/System.Collections.Generic</code>

Разумеется, это вполне нормальные подходы к манипулированию данными. В действительности вы можете (и будете) пользоваться напрямую ADO.NET, пространствами имен XML, службами рефлексии и различными типами коллекций. Однако основная

проблема состоит в том, что каждый из этих API-интерфейсов представляет собой “изолированный островок”, слабо интегрируемый с другими. Правда, можно (к примеру) сохранить DataSet из ADO.NET в документ XML и затем манипулировать им через пространства имен System.Xml, но все равно манипуляции данными остаются довольно асимметричными.

В рамках API-интерфейса LINQ была предпринята попытка предложить программистам согласованный, симметричный способ получения и манипулирования “данными” в самом широком смысле этого понятия. Используя LINQ, можно создавать непосредственно внутри синтаксиса языка C# конструкции, называемые *выражениями запросов*. Эти выражения запросов основаны на множестве операций запросов, которые намеренно сделаны похожими, внешне и по поведению (но не идентичными), на выражения SQL.

Фокус, однако, в том, что выражение запроса может применяться для взаимодействия с многочисленными типами данных, даже данными, которые никак не связаны с реляционными базами. Строго говоря, “LINQ” — это термин, описывающий общий подход к доступу к данным. В зависимости от места использования запросов LINQ, существуют разновидности LINQ, которые перечислены ниже.

- *LINQ to Objects*. Эта разновидность позволяет применять запросы LINQ к массивам и коллекциям.
- *LINQ to XML*. Эта разновидность позволяет применять LINQ для манипулирования и опроса документов XML.
- *LINQ to DataSet*. Эта разновидность позволяет применять запросы LINQ к объектам DataSet из ADO.NET.
- *LINQ to Entities*. Эта разновидность позволяет применять запросы LINQ внутри API-интерфейса ADO.NET Entity Framework (EF).
- *Parallel LINQ* (он же *PLINQ*). Эта разновидность позволяет выполнять параллельную обработку данных, возвращенных запросом LINQ.

Похоже, в Microsoft намерены глубоко интегрировать поддержку LINQ в среду программирования .NET. В настоящее время LINQ является неотъемлемой частью библиотек базовых классов .NET, управляемых языков и самой среды разработки Visual Studio.

Выражения LINQ строго типизированы

Очень важно отметить, что выражение запроса LINQ (в отличие от традиционных операторов SQL) является *строго типизированным*. Поэтому компилятор C# следит за этим и гарантирует, что выражения синтаксически оформлены корректно. Попутно следует отметить, что выражения запросов имеют представление метаданных внутри использующей их сборки, поскольку операции запросов LINQ всегда обладают развитой объектной моделью. Такие инструменты, как Visual Studio, могут пользоваться этими метаданными для обеспечения работы полезных средств вроде IntelliSense, автозавершения и т.п.

Основные сборки LINQ

Как упоминалось в главе 2, в диалоговом окне New Project (Новый проект) в Visual Studio доступна возможность выбора версии платформы .NET, для которой нужно производить компиляцию. При компиляции для .NET 3.5 и последующих версий каждый из шаблонов проектов автоматически ссылается на ключевые сборки LINQ, что легко заметить в Solution Explorer. В табл. 12.2 описана роль ключевых сборок LINQ. В остальной части книги вы встретите и другие дополнительные библиотеки LINQ.

Таблица 12.2. Основные сборки, связанные с LINQ

Сборка	Назначение
System.Core.dll	Определяет типы, представляющие основной API-интерфейс LINQ. Это единственная сборка, к которой нужно иметь доступ, чтобы пользоваться любым API-интерфейсом LINQ, включая LINQ to Objects
System.Data.DataSetExtensions.dll	Определяет набор типов для интеграции типов ADO.NET в программную парадигму LINQ (LINQ to DataSet)
System.Xml.Linq.dll	Предоставляет функциональность для использования LINQ с данными документов XML (LINQ to XML)

Для того чтобы работать с LINQ to Objects, потребуется обеспечить, чтобы в каждом файле кода C#, содержащем запросы LINQ, импортировалось пространство имен System.Linq (определенное внутри сборки System.Core.dll). В противном случае возникнет множество проблем. Если вы столкнулись с примерно таким сообщением об ошибке во время компиляции:

```
Error 1 Could not find an implementation of the query pattern for source type 'int[]'.
'Where' not found. Are you missing a reference to 'System.Core.dll' or a using
directive for 'System.Linq'?
```

```
Ошибка 1 Не удается обнаружить реализацию шаблона запроса для исходного типа 'int[]'.
'Where' не найдено. Возможно, пропущена ссылка на 'System.Core.dll' или
директива using для 'System.Linq'?
```

то очень высока вероятность, что в файле C# отсутствует следующая директива:

```
using System.Linq;
```

Применение запросов LINQ к элементарным массивам

Чтобы приступить к изучению LINQ to Objects, давайте построим приложение, которое будет применять запросы LINQ к различным объектам типа массива. Создадим консольное приложение по имени LinqOverArray и определим внутри класса Program статический вспомогательный метод под названием QueryOverStrings(). В этом методе создадим массив строк, содержащий несколько элементов по своему усмотрению (например, названия видеоигр). Пусть хотя бы два элемента содержат числовые значения, и еще несколько — внутренние пробелы.

```
static void QueryOverStrings()
{
    // Предположим, что имеется массив строк.
    string[] currentVideoGames = {"Morrowind", "Uncharted 2",
                                   "Fallout 3", "Daxter", "System Shock 2"};
}
```

Теперь модифицируем Main() для вызова QueryOverStrings():

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with LINQ to Objects *****\n");
    QueryOverStrings();
    Console.ReadLine();
}
```

Имея дело с любым массивом данных, очень часто приходится извлекать из него подмножество элементов на основе определенного критерия. Скажем, требуется получить только элементы, содержащие какое-нибудь число (т.е. “System Shock 2”, “Uncharted 2” и “Fallout 3”), либо имеющие больше или меньше определенного количества символов, либо не содержащие внутренних пробелов (т.е. “Morrowind” или “Daxter”) и т.п. Хотя такие задачи определенно можно решать с использованием членов типа `System.Array` и прикладыванием приличных усилий, выражения запросов LINQ значительно упрощают ситуацию.

Исходя из предположения, что требуется получить из массива только элементы, содержащие внутри себя пробел, причем в алфавитном порядке, можно построить следующее выражение запроса LINQ:

```
static void QueryOverStrings()
{
    // Предположим, что имеется массив строк.
    string[] currentVideoGames = {"Morrowind", "Uncharted 2",
        "Fallout 3", "Daxter", "System Shock 2"};

    // Выражение запроса для нахождения элементов массива, включающих пробелы.
    IEnumerable<string> subset = from g in currentVideoGames
        where g.Contains(" ") orderby g select g;

    // Вывести на консоль результаты.
    foreach (string s in subset)
        Console.WriteLine("Item: {0}", s);
}
```

Обратите внимание, что созданное здесь выражение запроса использует LINQ-операции `from`, `in`, `where`, `orderby` и `select`. Формальности синтаксиса выражений запросов будут изложены далее в этой главе. Однако даже сейчас можно прочесть этот оператор примерно как “предоставить элементы из `currentVideoGames`, содержащие пробелы, в алфавитном порядке”.

Здесь каждый элемент, соответствующий критерию поиска, получает имя `g` (от “game”); однако можно было бы использовать любое допустимое имя переменной C#:

```
IEnumerable<string> subset = from game in currentVideoGames
    where game.Contains(" ") orderby
    game select game;
```

Возвращенная последовательность хранится в переменной по имени `subset`, тип которой реализует обобщенную версию интерфейса `IEnumerable<T>`, где `T` — тип `System.String` (в конце концов, опрашивается массив элементов `string`). После получения результирующего набора затем можно просто вывести на консоль его элементы с помощью стандартной конструкции `foreach`. В результате запуска приложения на выполнение будет получен следующий вывод:

```
***** Fun with LINQ to Objects *****
Item: Fallout 3
Item: System Shock 2
Item: Uncharted 2
```

Решение без использования LINQ

Строго говоря, применение LINQ никогда не бывает обязательным. При желании тот же результирующий набор можно получить, отказавшись от LINQ и воспользовавшись такими программными конструкциями, как операторы `if` и циклы `for`. Ниже приведен метод, который выдает тот же результат, что и `QueryOverStrings()`, но в намного более многословной манере:


```
static void QueryOverStringsLongHand()
{
    // Предположим, что имеется массив строк.
    string[] currentVideoGames = {"Morrowind", "Uncharted 2",
        "Fallout 3", "Daxter", "System Shock 2"};
    string[] gamesWithSpaces = new string[5];
    for (int i = 0; i < currentVideoGames.Length; i++)
    {
        if (currentVideoGames[i].Contains(" "))
            gamesWithSpaces[i] = currentVideoGames[i];
    }

    // Отсортировать набор.
    Array.Sort(gamesWithSpaces);

    // Вывести на консоль результат.
    foreach (string s in gamesWithSpaces)
    {
        if (s != null)
            Console.WriteLine("Item: {0}", s);
    }
    Console.WriteLine();
}
```

Несмотря на возможные пути улучшения этого метода, факт остается фактом — запросы LINQ могут радикально упростить процесс извлечения новых подмножеств данных из источника. Вместо построения вложенных циклов, сложной логики if/else, временных типов данных и тому подобного, компилятор C# выполнит всю черновую работу за вас, как только будет создан подходящий запрос LINQ.

Рефлексия результирующего набора LINQ

Теперь предположим, что в классе Program определена дополнительная вспомогательная функция по имени `ReflectOverQueryResults()`, которая выводит на консоль разнообразные детали о результирующем наборе LINQ (обратите внимание на тип параметра — `System.Object`, что позволяет принимать различные типы результирующих наборов):

```
static void ReflectOverQueryResults(object resultSet)
{
    Console.WriteLine("***** Info about your query *****");
    Console.WriteLine("resultSet is of type: {0}", resultSet.GetType().Name); // тип
    Console.WriteLine("resultSet location: {0}",
        resultSet.GetType().Assembly.GetName().Name); // расположение
}
```

Поместив вызов этого метода в `QueryOverStrings()` непосредственно после вывода полученного подмножества и запустив приложение, можно увидеть, что это подмножество на самом деле представляет собой экземпляр обобщенного типа `OrderedEnumerable<TElement, TKey>` (представленного в CIL-коде как `OrderedEnumerable`2`), который является абстрактным внутренним типом, находящимся в сборке `System.Core.dll`:

```
***** Info about your query *****
resultSet is of type: OrderedEnumerable`2
resultSet location: System.Core
```

На заметку! Многие из типов, представляющих результат LINQ, скрыты в браузерке объектов Visual Studio. Это низкоуровневые типы, которые не предназначены для прямого использования в приложениях.

LINQ и неявно типизированные локальные переменные

Хотя в приведенном примере программы относительно легко догадаться, что результирующий набор может быть интерпретирован как перечисление объектов `string` (т.е. `IEnumerable<string>`), менее очевидно, что типом подмножества на самом деле является `OrderedEnumerable<TElement, TKey>`.

Поскольку результирующие наборы LINQ могут быть представлены с использованием изрядного числа типов из различных пространств имен LINQ, было бы утомительно определять подходящий тип для хранения результирующего набора, потому что во многих случаях лежащий в основе тип не очевиден и даже напрямую не доступен в коде (и как будет показано, иногда тип генерируется во время компиляции).

Чтобы еще более подчеркнуть это обстоятельство, ниже представлен дополнительный вспомогательный метод, определенный внутри класса `Program` (который должен быть вызван из метода `Main()`):

```
static void QueryOverInts()
{
    int[] numbers = {10, 20, 30, 40, 1, 2, 3, 8};
    // Вывести только элементы меньше 10.
    IEnumerable<int> subset = from i in numbers where i < 10 select i;
    foreach (int i in subset)
        Console.WriteLine("Item: {0}", i);
    ReflectOverQueryResults(subset);
}
```

В рассматриваемом случае переменная `subset` будет иметь совершенно другой внутренний тип. На этот раз тип, реализующий интерфейс `IEnumerable<int>` — это низкоуровневый класс по имени `WhereArrayIterator<T>`:

```
Item: 1
Item: 2
Item: 3
Item: 8

***** Info about your query *****
resultSet is of type: WhereArrayIterator`1
resultSet location: System.Core
```

Поскольку точный тип запроса LINQ определенно неизвестен, в первом примере результат запроса был представлен как `IEnumerable<T>`, где `T` — тип данных возвращенной последовательности (`string`, `int` и т.д.). Это по-прежнему довольно запутано. Еще более усложняет картину то, что поскольку `IEnumerable<T>` расширяет необобщенный интерфейс `IEnumerable`, получить результат запроса LINQ допускается следующим образом:

```
System.Collections.IEnumerable subset =
    from i in numbers where i < 10 select i;
```

К счастью, неявная типизация существенно проясняет картину при работе с запросами LINQ:

```
static void QueryOverInts()
{
    int[] numbers = {10, 20, 30, 40, 1, 2, 3, 8};
    // Здесь используется неявная типизация...
    var subset = from i in numbers where i < 10 select i;
    // ...и здесь тоже.
    foreach (var i in subset)
        Console.WriteLine("Item: {0} ", i);
    ReflectOverQueryResults(subset);
}
```

Запомните эмпирическое правило: при захвате результатов запроса LINQ всегда следует применять неявную типизацию. Однако помните, что (в большинстве случаев) реальное возвращенное значение имеет тип, реализующий интерфейс `IEnumerable<T>`.

Какой именно тип кроется за этим (`OrderedEnumerable<TElement, TKey>`, `WhereArrayIterator<T>` и т.п.) — не важно, и определять его не обязательно. Как было показано в предыдущем примере кода, можно просто воспользоваться ключевым словом `var` с конструкцией `foreach` для проведения итерации по полученным данным.

LINQ и расширяющие методы

Хотя в рассматриваемом примере не требуется явно писать какие-то расширяющие методы, на самом деле они используются на заднем плане. Выражения запросов LINQ могут применяться для итерации по содержимому контейнеров данных, которые реализуют обобщенный интерфейс `IEnumerable<T>`. Однако класс .NET `System.Array` (используемый для представления массива строк и массива целых чисел) не реализует этот контракт:

```
// Тип System.Array, похоже, не реализует корректную
// инфраструктуру для выражений запросов!
public abstract class Array : ICloneable, IList, ICollection,
    IEnumerable, IStructuralComparable, IStructuralEquatable
{
    ...
}
```

Несмотря на то что `System.Array` не реализует напрямую интерфейс `IEnumerable<T>`, он опосредованно получает необходимую функциональность этого типа (а также многие другие члены, связанные с LINQ) через статический тип класса `System.Linq.Enumerable`.

В этом служебном классе определено множество обобщенных расширяющих методов (таких как `Aggregate<T>()`, `First<T>`, `Max<T>` и т.д.), которые `System.Array` (и другие типы) получают в свое распоряжение на заднем плане. Таким образом, после ввода операции точки за текущей локальной переменной `currentVideoGames` обнаружится значительное количество членов, которые *отсутствуют* в формальном определении `System.Array` (рис. 12.1).

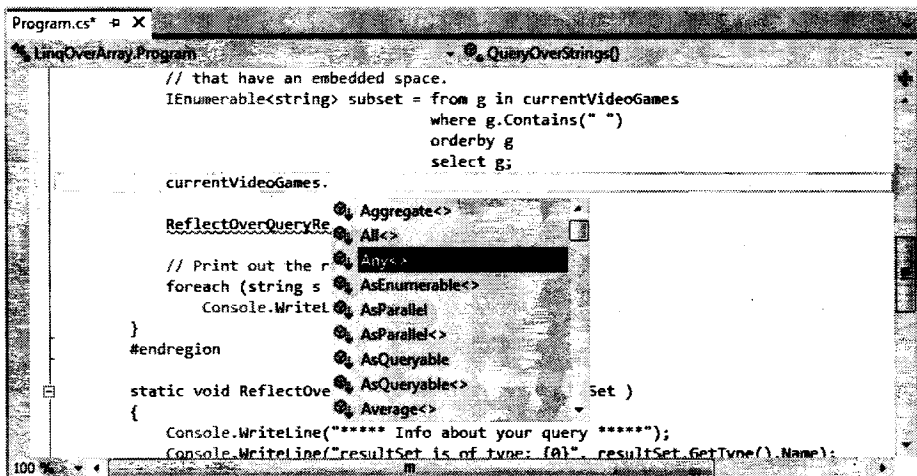


Рис. 12.1. Тип `System.Array` был расширен членами `System.Linq.Enumerable`

Роль отложенного выполнения

Другой важный момент, касающийся выражений запросов LINQ, состоит в том, что на самом деле они не выполняются до тех пор, пока не будет начата итерация по последовательности. Формально это называется *отложенным выполнением*. Преимущество такого подхода заключается в возможности применения одного и того же запроса LINQ многократно к одному и тому же контейнеру, с полной гарантией получения самых актуальных результатов. Рассмотрим следующую модификацию метода `QueryOverInts()`:

```
static void QueryOverInts()
{
    int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };

    // Получить числа меньше 10.
    var subset = from i in numbers where i < 10 select i;

    // Оператор LINQ выполняется здесь!
    foreach (var i in subset)
        Console.WriteLine("{0} < 10", i);
    Console.WriteLine();

    // Изменить некоторые данные в массиве.
    numbers[0] = 4;

    // Оператор LINQ снова выполняется!
    foreach (var j in subset)
        Console.WriteLine("{0} < 10", j);

    Console.WriteLine();
    ReflectOverQueryResults(subset);
}
```

Ниже показан вывод после запуска этой программы. Обратите внимание, что во второй итерации по запрошенной последовательности появился дополнительный член, поскольку для первого элемента массива было установлено значение меньше 10.

```
1 < 10
2 < 10
3 < 10
8 < 10

4 < 10
1 < 10
2 < 10
3 < 10
8 < 10
```

Среда Visual Studio обладает одним очень полезным аспектом: поместив точку останова перед выполнением запроса LINQ, можно просматривать содержимое в сеансе отладки. Переместите курсор мыши на переменную результирующего набора LINQ (`subset` на рис. 12.2). После этого появляется возможность выполнить запрос в этот момент, раскрыв узел Results View (Представление результатов).

Роль немедленного выполнения

Чтобы выполнить выражение LINQ за пределами логики итерации `foreach`, можно вызвать любое количество расширяющих методов, определенных типом `Enumerable`, таких как `ToArray<T>`, `ToDictionary<TSource, TKey>()` и `ToList<T>()`. Все эти методы заставляют запрос LINQ выполняться в момент их вызова, чтобы получить снимок данных.

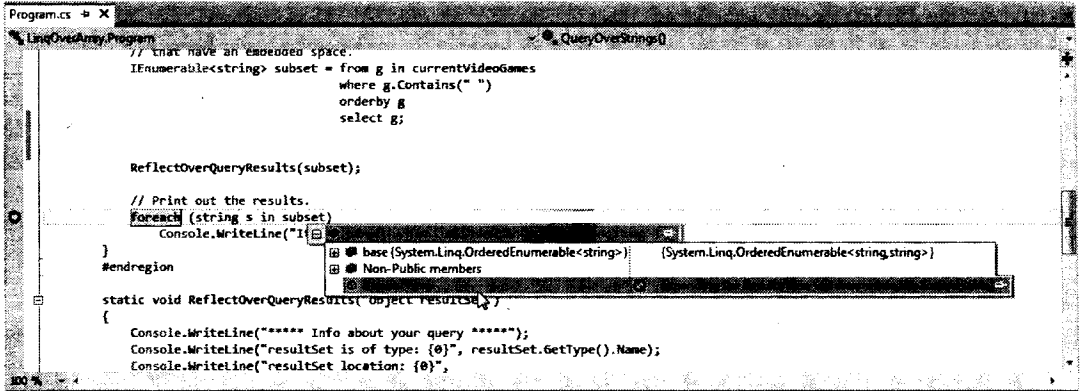


Рис. 12.2. Отладка выражений LINQ

После этого полученным снимком можно манипулировать независимо.

```
static void ImmediateExecution()
{
    int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };

    // Получить данные НЕМЕДЛЕННО как int[].
    int[] subsetAsIntArray =
        (from i in numbers where i < 10 select i).ToArray<int>();

    // Получить данные НЕМЕДЛЕННО как List<int>.
    List<int> subsetAsListOfInts =
        (from i in numbers where i < 10 select i).ToList<int>();
}
```

Обратите внимание, что все выражение LINQ заключено в скобки для приведения к корректному лежащему в основе типу (каким бы он ни был); это позволяет вызывать методы Enumerable.

Также вспомните из главы 9, что когда компилятор C# может однозначно определить параметр типа обобщенного элемента, то вы не обязаны указывать этот параметр. Таким образом, `ToArray<T>` (или `ToList<T>`) можно было бы вызвать следующим образом:

```
int[] subsetAsIntArray =
    (from i in numbers where i < 10 select i).ToArray();
```

Польза от немедленного выполнения очевидна, когда нужно вернуть результат запроса LINQ внешнему вызывающему коду. Это будет темой следующего раздела главы.

Исходный код. Проект `LinqOverArray` доступен в подкаталоге `Chapter 12`.

Возврат результата запроса LINQ

В классе (или структуре) можно определить поле, значением которого будет результат запроса LINQ. Однако для этого нельзя использовать неявную типизацию (т.е. ключевое слово `var` для таких полей применяться не может), и целью запроса LINQ не могут быть данные уровня экземпляра, а потому он должен быть статическим. Учитывая эти ограничения, писать код вроде показанного ниже придется редко:

```

class LINQBasedFieldsAreClunky
{
    private static string[] currentVideoGames = {"Morrowind", "Uncharted 2",
        "Fallout 3", "Daxter", "System Shock 2"};
    // Здесь нельзя использовать неявную типизацию! Нужно знать тип subset!
    private IEnumerable<string> subset = from g in currentVideoGames
        where g.Contains(" ") orderby g select g;
    public void PrintGames()
    {
        foreach (var item in subset)
        {
            Console.WriteLine(item);
        }
    }
}

```

Довольно часто запросы LINQ определяются в контексте метода или свойства. Более того, с целью упрощения переменная, предназначенная для хранения результирующего набора, будет храниться в неявно типизированной локальной переменной с использованием ключевого слова `var`. Вспомните из главы 3, что неявно типизированные переменные не могут применяться для определения параметров, возвращаемых значений или полей класса либо структуры.

С учетом всего этого может возникнуть вопрос: как вернуть результат запроса внешнему коду? Все зависит от обстоятельств. Если есть результирующий набор, состоящий из строго типизированных данных, такой как массив строк или список `List<T>` объектов `Car`, можно отказаться от ключевого слова `var` и использовать правильный тип `IEnumerable<T>` либо `IEnumerable` (поскольку `IEnumerable<T>` расширяет `IEnumerable`). Ниже показан пример нового консольного приложения под названием `LinqRetValValues`.

```

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** LINQ Transformations *****\n");
        IEnumerable<string> subset = GetStringSubset();
        foreach (string item in subset)
        {
            Console.WriteLine(item);
        }
        Console.ReadLine();
    }
    static IEnumerable<string> GetStringSubset()
    {
        string[] colors = {"Light Red", "Green",
            "Yellow", "Dark Red", "Red", "Purple"};
        // Обратите внимание, что subset представляет собой
        // объект, совместимый с IEnumerable<string>.
        IEnumerable<string> theRedColors = from c in colors
            where c.Contains("Red") select c;
        return theRedColors;
    }
}

```

Результат выглядит следующим образом:

```

LINQ Transformations
Light Red
Dark Red
Red

```

Возврат результатов LINQ через немедленное выполнение

Этот пример работает ожидаемым образом только потому, что возвращаемое значение `GetStringSubset()` и запрос LINQ внутри этого метода строго типизированы. Если воспользоваться ключевым словом `var` для определения переменной `subset`, то вернуть значение можно будет, только если метод по-прежнему возвращает `IEnumerable<string>` (и если неявно типизированная локальная переменная на самом деле совместима с указанным типом возврата).

Поскольку оперировать `IEnumerable<T>` несколько неудобно, можно воспользоваться немедленным выполнением. Например, вместо возврата `IEnumerable<string>` можно просто вернуть `string[]`, при условии трансформирования последовательности в строго типизированный массив. Ниже показан новый метод класса `Program`, который именно это и делает:

```
static string[] GetStringSubsetAsArray()
{
    string[] colors = {"Light Red", "Green",
        "Yellow", "Dark Red", "Red", "Purple"};
    var theRedColors = from c in colors
        where c.Contains("Red") select c;

    // Отобразить результаты в массив.
    return theRedColors.ToArray();
}
```

При этом вызывающему коду не известно, что полученный им результат поступил от запроса LINQ, и он просто работает с массивом строк, как ожидалось. Например:

```
foreach (string item in GetStringSubsetAsArray())
{
    Console.WriteLine(item);
}
```

Немедленное выполнение также важно при попытке вернуть вызывающему коду результат проекции LINQ. Чуть позже в этой главе мы еще вернемся к этой теме. Однако сначала давайте посмотрим, как применять запросы LINQ к обобщенным и необобщенным объектам коллекций.

Исходный код. Проект `LinqRetValues` доступен в подкаталоге `Chapter 12`.

Применение запросов LINQ к объектам коллекций

Помимо извлечения результатов из простого массива данных, выражения запросов LINQ также манипулируют данными внутри членов коллекций из пространства имен `System.Collections.Generic`, таких как `List<T>`. Создадим новое консольное приложение по имени `ListOverCollections` и определим базовый класс `Car`, поддерживающий текущую скорость, цвет, производителя и дружественное имя, как показано ниже:

```
class Car
{
    public string PetName {get; set;}
    public string Color {get; set;}
    public int Speed {get; set;}
    public string Make {get; set;}
}
```

Теперь определим в методе `Main()` переменную `List<T>` для хранения элементов типа `Car` и воспользуемся синтаксисом инициализации объектов для заполнения списка несколькими новыми объектами `Car`:

```
static void Main(string[] args)
{
    Console.WriteLine("***** LINQ over Generic Collections *****\n");
    // Создать список List<> объектов Car.
    List<Car> myCars = new List<Car>() {
        new Car{ PetName = "Henry", Color = "Silver", Speed = 100, Make = "BMW"},
        new Car{ PetName = "Daisy", Color = "Tan", Speed = 90, Make = "BMW"},
        new Car{ PetName = "Mary", Color = "Black", Speed = 55, Make = "VW"},
        new Car{ PetName = "Clunker", Color = "Rust", Speed = 5, Make = "Yugo"},
        new Car{ PetName = "Melvin", Color = "White", Speed = 43, Make = "Ford"}
    };
    Console.ReadLine();
}
```

Доступ к содержащимся в контейнере подобъектам

Применение запроса LINQ к обобщенному контейнеру ничем не отличается от применения к простому запросу, поскольку LINQ to Objects может использоваться с любым типом, реализующим интерфейс `IEnumerable<T>`. На этот раз цель заключается в построении выражения запроса для получения объектов `Car` из списка `myCars`, у которых значение скорости больше 55.

После получения подмножества на консоль будет выводиться имя каждого объекта `Car` за счет обращения к его свойству `PetName`. Предположим, что определен следующий вспомогательный метод (принимаящий параметр `List<Car>`), который вызывается в `Main()`:

```
static void GetFastCars(List<Car> myCars)
{
    // Найти в контейнере List<> все объекты Car, у которых Speed больше 55.
    var fastCars = from c in myCars where c.Speed > 55 select c;
    foreach (var car in fastCars)
    {
        Console.WriteLine("{0} is going too fast!", car.PetName);
    }
}
```

Обратите внимание, что выражение запроса выбирает только те элементы из `List<T>`, у которых `Speed` больше 55. Запустив приложение, можно увидеть, что критерии поиска отвечают только два элемента — “Henry” и “Daisy”.

Чтобы построить более сложный критерий, можно запросить только автомобили марки BMW со значением `Speed` больше 90. Для этого понадобится создать составной булевский оператор, в котором используется операция `&&` языка C#:

```
static void GetFastBMWs(List<Car> myCars)
{
    // Найти быстрые автомобили BMW!
    var fastCars = from c in myCars where c.Speed > 90 && c.Make == "BMW" select c;
    foreach (var car in fastCars)
    {
        Console.WriteLine("{0} is going too fast!", car.PetName);
    }
}
```

В этом случае на консоль выводится только одно имя — “Henry”.

Применение запросов LINQ к необобщенным коллекциям

Вспомните, что операции запросов LINQ предназначены для работы с любым типом, реализующим `IEnumerable<T>` (как непосредственно, так и через расширяющие методы). Учитывая то, что класс `System.Array` оснащен всей необходимой инфраструктурой, может оказаться сюрпризом, что унаследованные (необобщенные) контейнеры из `System.Collections` такой поддержки не имеют. К счастью, итерацию по данным, содержащимся внутри необобщенных коллекций, все равно можно выполнять с использованием обобщенного расширяющего метода `Enumerable.OfTye<T>()`.

Метод `OfTye<T>()` — один из нескольких членов `Enumerable`, которые не расширяют обобщенные типы. При вызове этого члена на необобщенном контейнере, реализующем интерфейс `IEnumerable` (вроде `ArrayList`), просто укажите тип элемента в контейнере для извлечения совместимого с `IEnumerable<T>` объекта. Сохранить эти данные в коде можно в неявно типизированной переменной.

Ниже показан новый метод, который заполняет `ArrayList` множеством объектов `Car` (не забудьте импортировать пространство имен `System.Collections` в файл `Program.cs`).

```
static void LINQOverArrayList()
{
    Console.WriteLine("***** LINQ по ArrayList *****");
    // Необобщенная коллекция автомобилей.
    ArrayList myCars = new ArrayList() {
        new Car{ PetName = "Henry", Color = "Silver", Speed = 100, Make = "BMW"},
        new Car{ PetName = "Daisy", Color = "Tan", Speed = 90, Make = "BMW"},
        new Car{ PetName = "Mary", Color = "Black", Speed = 55, Make = "VW"},
        new Car{ PetName = "Clunker", Color = "Rust", Speed = 5, Make = "Yugo"},
        new Car{ PetName = "Melvin", Color = "White", Speed = 43, Make = "Ford"}
    };
    // Трансформировать ArrayList в тип, совместимый с IEnumerable<T>.
    var myCarsEnum = myCars.OfTye<Car>();
    // Создать выражение запроса, нацеленное на совместимый с IEnumerable<T> тип.
    var fastCars = from c in myCarsEnum where c.Speed > 55 select c;
    foreach (var car in fastCars)
    {
        Console.WriteLine("{0} is going too fast!", car.PetName);
    }
}
```

Аналогично предыдущим примерам, этот метод, вызванный в `Main()`, отобразит только имена "Henry" и "Daisy" на основе формата нашего запроса LINQ.

Фильтрация данных с использованием `OfTye<T>()`

Как уже известно, необобщенные типы способны содержать комбинации любых элементов в качестве членов контейнеров (таких как `ArrayList`), поскольку они прототипированы для приема экземпляров `System.Object`. Например, предположим, что `ArrayList` содержит различные элементы, часть из которых являются числами. Получить подмножество, состоящее только из числовых данных, можно с помощью метода `OfTye<T>()`, поскольку во время итераций он отфильтрует все элементы, тип которых отличается от заданного.

```
static void OfTyeAsFilter()
{
    // Извлечение целых из ArrayList.
    ArrayList myStuff = new ArrayList();
```

```

myStuff.AddRange(new object[] { 10, 400, 8, false, new Car(), "string data" });
var myInts = myStuff.OfType<int>();

// Выведет на консоль 10, 400 и 8.
foreach (int i in myInts)
{
    Console.WriteLine("Int value: {0}", i);
}
}

```

Теперь есть возможность применять запросы LINQ к массивам, а также обобщенным и необобщенным коллекциям. Эти контейнеры содержат как элементарные типы C# (целочисленные, строковые данные), так и пользовательские специальные классы. Следующая задача — изучить множество дополнительных операций LINQ, которые можно использовать для построения сложных и полезных запросов.

Исходный код. Проект LinqOverCollectons доступен в подкаталоге Chapter 12.

Исследование операций запросов LINQ

В языке C# определено множество операций запросов в готовом виде. Некоторые наиболее часто используемые из них перечислены в табл. 12.3.

На заметку! Документация .NET Framework 4.5 SDK содержит подробные сведения по каждой операции C# LINQ (в разделе "LINQ General Programming Guide" ("Общее руководство по программированию на LINQ")).

Таблица 12.3. Часто используемые операции запросов LINQ

Операции запросов	Назначение
from, in	Используются для определения основы любого выражения LINQ, позволяющей извлечь подмножество данных из нужного контейнера
where	Используется для определения ограничений о том, т.е. какие элементы должны извлекаться из контейнера
select	Используется для выбора последовательности из контейнера
join, on, equals, into	Выполняют соединения на основе указанного ключа. Помните, что эти "соединения" ничего не делают с данными в реляционных базах
orderby, ascending, descending	Позволяют упорядочить результирующий набор в порядке возрастания или убывания
group, by	Выдают подмножество с данными, сгруппированными по указанному значению

В дополнение к частичному списку операций, приведенному в табл. 12.3, класс `System.Linq.Enumerable` предлагает набор методов, которые не имеют прямой сокращенной нотации в форме операций запроса C#, а представлены в виде расширяющих методов. Эти обобщенные методы могут вызываться для трансформации результирующего набора в различной манере (`Reverse<>()`, `ToArray<>()`, `ToList<>()` и т.п.). Некоторые из них используются для извлечения одиночных элементов из результирующего набора, другие выполняют различные операции над множествами (`Distinct<>()`, `Union<>()`, `Intersect<>()` и т.д.), третьи агрегируют результаты (`Count<>()`, `Sum<>()`, `Min<>()`, `Max<>()` и т.п.).

Чтобы приступить к исследованию более сложных запросов LINQ, создадим новое консольное приложение под названием FunWithLinqExpressions. Затем определим массив или коллекцию некоторых простых данных. Для данного проекта создается массив объектов ProductInfo, определенный в следующем коде:

```
class ProductInfo
{
    public string Name {get; set;}
    public string Description {get; set;}
    public int NumberInStock {get; set;}
    public override string ToString()
    {
        return string.Format("Name={0}, Description={1}, Number in Stock={2}",
            Name, Description, NumberInStock);
    }
}
```

Теперь внутри метода Main() заполним массив объектами ProductInfo:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Query Expressions *****\n");
    // Этот массив будет основой для тестирования...
    ProductInfo[] itemsInStock = new[] {
        new ProductInfo{ Name = "Mac's Coffee",
            Description = "Coffee with TEETH",
            NumberInStock = 24},
        new ProductInfo{ Name = "Milk Maid Milk",
            Description = "Milk cow's love",
            NumberInStock = 100},
        new ProductInfo{ Name = "Pure Silk Tofu",
            Description = "Bland as Possible",
            NumberInStock = 120},
        new ProductInfo{ Name = "Cruchy Pops",
            Description = "Cheezy, peppery goodness",
            NumberInStock = 2},
        new ProductInfo{ Name = "RipOff Water",
            Description = "From the tap to your wallet",
            NumberInStock = 100},
        new ProductInfo{ Name = "Classic Valpo Pizza",
            Description = "Everyone loves pizza!",
            NumberInStock = 73}
    };
    // Здесь мы будем вызывать разнообразные методы!
    Console.ReadLine();
}
```

Базовый синтаксис выборки

Поскольку синтаксическая корректность выражения запроса LINQ проверяется во время компиляции, следует помнить, что порядок этих операций важен. В простейшем виде каждый запрос LINQ строится из операций from, in и select. Ниже показан базовый шаблон, которому нужно следовать:

```
var результат = from сопоставляемыйЭлемент in контейнер
                select сопоставляемыйЭлемент;
```

Элемент, следующий за операцией from, представляет элемент, соответствующий критерию запроса LINQ, и назвать его можно по своему усмотрению. Элемент после операции in представляет контейнер данных для поиска (массив, коллекция, документ XML и т.д.).

Рассмотрим очень простой запрос, который не делает ничего помимо извлечения каждого элемента контейнера (это похоже на поведение SQL-оператора `SELECT *` в базе данных):

```
static void SelectEverything(ProductInfo[] products)
{
    // Получить все!
    Console.WriteLine("All product details:");
    var allProducts = from p in products select p;

    foreach (var prod in allProducts)
    {
        Console.WriteLine(prod.ToString());
    }
}
```

По правде говоря, это выражение запроса не слишком полезно, поскольку выдаст подмножество, идентичное содержимому входного параметра. При желании из входящего параметра можно извлечь только значения `Name` каждого товара, используя следующий синтаксис выборки:

```
static void ListProductNames(ProductInfo[] products)
{
    // Теперь получить только наименования товаров.
    Console.WriteLine("Only product names:");
    var names = from p in products select p.Name;

    foreach (var n in names)
    {
        Console.WriteLine("Name: {0}", n);
    }
}
```

Получение подмножества данных

Чтобы получить определенное подмножество из контейнера, можно воспользоваться операцией `where`. При этом общий шаблон запроса становится таким:

```
var результат = from элемент in контейнер where булевскоеВыражение select элемент;
```

Обратите внимание, что операция `where` ожидает выражения, которое возвращает булевское значение. Например, чтобы получить из массива-аргумента `ProductInfo[]` только те позиции, которых на складе имеется более 25 единиц, можно написать следующий код:

```
static void GetOverstock(ProductInfo[] products)
{
    Console.WriteLine("The overstock items!");

    // Получить только те товары, которых на складе более 25.
    var overstock = from p in products where p.NumberInStock > 25 select p;

    foreach (ProductInfo c in overstock)
    {
        Console.WriteLine(c.ToString());
    }
}
```

Как уже было показано ранее в этой главе, при построении конструкции `where` допускается применять любые операции C# для получения сложных выражений. Например, вспомним запрос, который извлекал только автомобили BMW, движущихся со скоростью выше 100 миль в час:

```
// Получить машины BMW, движущиеся со скоростью свыше 100 миль в час.
var onlyFastBMWs = from c in myCars
                    where c.Make == "BMW" && c.Speed >= 100 select c;

foreach (Car c in onlyFastBMWs)
{
    Console.WriteLine("{0} is going {1} MPH", c.PetName, c.Speed);
}
```

Проецирование новых типов данных

Новые формы данных можно также проецировать на основе существующих источников. Предположим, что для входящего параметра `ProductInfo[]` необходимо получить результирующий набор, который содержит только имя и описание каждого элемента. Для этого понадобится определить оператор `select`, который динамически выдаст новый анонимный тип:

```
static void GetNamesAndDescriptions(ProductInfo[] products)
{
    Console.WriteLine("Names and Descriptions:");
    var nameDesc = from p in products select new { p.Name, p.Description };
    foreach (var item in nameDesc)
    {
        // Можно также использовать свойства Name и Description напрямую.
        Console.WriteLine(item.ToString());
    }
}
```

Всегда помните, что когда имеете дело с запросом `LINQ`, выполняющим проекцию, нет никакой возможности узнать лежащий в ее основе тип данных, поскольку он определяется во время компиляции. В этих случаях обязательным является ключевое слово `var`. Кроме того, нельзя создавать методы с неявно типизированными возвращаемыми значениями. Поэтому следующий код не скомпилируется:

```
static var GetProjectedSubset(ProductInfo[] products)
{
    var nameDesc = from p in products select new { p.Name, p.Description };
    return nameDesc; // Ошибка!
}
```

Когда необходимо вернуть проекцию данных вызывающему коду, один из подходов предусматривает трансформацию результата запроса в `.NET`-объект `System.Array` за счет применения расширяющего метода `ToArray()`. Таким образом, если модифицировать выражение запроса следующим образом:

```
// Теперь возвращаемым значением является Array.
static Array GetProjectedSubset(ProductInfo[] products)
{
    var nameDesc = from p in products select new { p.Name, p.Description };
    // Отобразить набор анонимных объектов на объект Array.
    return nameDesc.ToArray();
}
```

то его можно вызвать и обработать данные в методе `Main()`, как показано ниже:

```
Array objs = GetProjectedSubset(itemsInStock);
foreach (object o in objs)
{
    Console.WriteLine(o); // Вызывает ToString() на каждом анонимном объекте.
}
```

Обратите внимание, что должен использоваться объект `System.Array`, при этом нельзя применять синтаксис объявления массива `C#`. Это связано с тем, что лежащий в основе проекции тип не известен, поскольку речь идет об анонимном классе, который сгенерирован компилятором. Кроме того, не указывается параметр типа для обобщенного метода `ToArray<T>()`, поскольку он тоже не известен до этапа компиляции.

Очевидная проблема здесь состоит в утере строгой типизации, поскольку каждый элемент в объекте `Array` считается относящимся к типу `Object`. Тем не менее, когда нужно вернуть результирующий набор `LINQ`, полученный в результате операции проекции, трансформация данных в тип `Array` (или другой подходящий контейнер через другие члены типа `Enumerable`) обязательна.

Получение счетчиков посредством `Enumerable`

При проецировании новых пакетов данных может понадобиться знать количество элементов перед тем, как вернуть их в последовательности. Для определения числа элементов, возвращаемых выражением запроса `LINQ`, используется расширяющий метод `Count()` класса `Enumerable`. Например, следующий метод пересчитает все объекты `string` в локальном массиве, имеющие длину более шести символов:

```
static void GetCountFromQuery()
{
    string[] currentVideoGames = {"Morrowind", "Uncharted 2",
                                   "Fallout 3", "Daxter", "System Shock 2"};
    // Получить количество из запроса.
    int numb =
        (from g in currentVideoGames where g.Length > 6 select g).Count();
    // Вывести на консоль количество элементов.
    Console.WriteLine("{0} items honor the LINQ query.", numb);
}
```

Обращение результирующих наборов

Поменять порядок элементов в результирующем наборе на противоположный довольно легко с помощью расширяющего метода `Reverse<T>()` класса `Enumerable`. Например, в следующем методе выбираются все элементы из входного массива `ProductInfo[]` в обратном порядке:

```
static void ReverseEverything(ProductInfo[] products)
{
    Console.WriteLine("Product in reverse:");
    var allProducts = from p in products select p;
    foreach (var prod in allProducts.Reverse())
    {
        Console.WriteLine(prod.ToString());
    }
}
```

Выражения сортировки

В начальных примерах настоящей главы было показано, что выражение запроса может принимать операцию `orderby` для сортировки элементов в подмножестве по заданному значению. По умолчанию принят порядок по возрастанию, поэтому упорядочение строк производится в алфавитном порядке, числовых значений — от меньшего к большему, и т.д. Чтобы отсортировать в обратном порядке, укажите операцию `descending`. Рассмотрим следующий метод:

```
static void AlphabetizeProductNames(ProductInfo[] products)
{
    // Получить названия товаров в алфавитном порядке.
    var subset = from p in products orderby p.Name select p;

    Console.WriteLine("Ordered by Name:");
    foreach (var p in subset)
    {
        Console.WriteLine(p.ToString());
    }
}
```

Хотя порядок по возрастанию принят по умолчанию, можно прояснить намерения, явно указав операцию ascending:

```
var subset = from p in products orderby p.Name ascending select p;
```

Для получения элементов в порядке убывания служит операция descending:

```
var subset = from p in products orderby p.Name descending select p;
```

LINQ как лучшее средство построения диаграмм

Класс `Enumerable` поддерживает набор расширяющих методов, которые позволяют использовать два (или более) запроса LINQ в качестве основы для нахождения объединений, разностей, конкатенаций и пересечений данных. Прежде всего, рассмотрим расширяющий метод `Except()`. Этот метод возвращает результирующий набор LINQ, содержащий разность между двумя контейнерами, которым в данном случае является значение "Yugo":

```
static void DisplayDiff()
{
    List<string> myCars = new List<String> { "Yugo", "Aztec", "BMW" };
    List<string> yourCars = new List<String> { "BMW", "Saab", "Aztec" };
    var carDiff = (from c in myCars select c)
        .Except(from c2 in yourCars select c2);

    Console.WriteLine("Here is what you don't have, but I do:");
    foreach (string s in carDiff)
        Console.WriteLine(s); // Выводит Yugo.
}
```

Метод `Intersect()` возвращает результирующий набор, содержащий общие элементы данных для множества контейнеров. Например, следующий метод вернет последовательность из "Aztec" и "BMW":

```
static void DisplayIntersection()
{
    List<string> myCars = new List<String> { "Yugo", "Aztec", "BMW" };
    List<string> yourCars = new List<String> { "BMW", "Saab", "Aztec" };

    // Получить общие члены.
    var carIntersect = (from c in myCars select c)
        .Intersect(from c2 in yourCars select c2);

    Console.WriteLine("Here is what we have in common:");
    foreach (string s in carIntersect)
        Console.WriteLine(s); // Выводит Aztec и BMW.
}
```

Метод `Union()` возвращает результирующий набор, который включает все члены множества запросов LINQ. Подобно любому объединению, повторяющиеся члены в нем

встречаются только однажды. Поэтому следующий метод выводит на консоль значения “Yugo”, “Aztec”, “BMW” и “Saab”:

```
static void DisplayUnion()
{
    List<string> myCars = new List<String> { "Yugo", "Aztec", "BMW" };
    List<string> yourCars = new List<String> { "BMW", "Saab", "Aztec" };

    // Получить объединение двух контейнеров.
    var carUnion = (from c in myCars select c)
        .Union(from c2 in yourCars select c2);
    Console.WriteLine("Here is everything:");

    foreach (string s in carUnion)
        Console.WriteLine(s); // Выводит все общие члены.
}
```

Наконец, расширяющий метод `Concat()` возвращает результирующий набор, который представляет собой прямую конкатенацию результирующих наборов LINQ. Например, следующий метод выводит на консоль в качестве результата “Yugo”, “Aztec”, “BMW”, “BMW”, “Saab” и “Aztec”:

```
static void DisplayConcat()
{
    List<string> myCars = new List<String> { "Yugo", "Aztec", "BMW" };
    List<string> yourCars = new List<String> { "BMW", "Saab", "Aztec" };

    var carConcat = (from c in myCars select c)
        .Concat(from c2 in yourCars select c2);

    // Выводит Yugo Aztec BMW BMW Saab Aztec.
    foreach (string s in carConcat)
        Console.WriteLine(s);
}
```

Исключение дубликатов

После вызова расширяющего метода `Concat()` очень легко можно получить в результате излишние элементы; иногда это является именно тем, что и требовалось. В других случаях может понадобиться удалить дублированные элементы данных. Для этого необходимо вызвать расширяющий метод `Distinct()`, как показано ниже:

```
static void DisplayConcatNoDups()
{
    List<string> myCars = new List<String> { "Yugo", "Aztec", "BMW" };
    List<string> yourCars = new List<String> { "BMW", "Saab", "Aztec" };
    var carConcat = (from c in myCars select c)
        .Concat(from c2 in yourCars select c2);

    // Выводит Yugo Aztec BMW Saab.
    foreach (string s in carConcat.Distinct())
        Console.WriteLine(s);
}
```

Агрегатные операции LINQ

Запросы LINQ могут также выполнять различные агрегатные операции над результирующим набором. Одним из примеров агрегации может служить расширяющий метод `Count()`. Другие возможности включают получение среднего, максимума, минимума или суммы значений за счет использования методов `Max()`, `Min()`, `Average()`, `Sum()`, которые являются членами класса `Enumerable`. Ниже приведен простой пример:


```
static void AggregateOps()
{
    double[] winterTemps = { 2.0, -21.3, 8, -4, 0, 8.2 };
    // Различные примеры агрегации.
    Console.WriteLine("Max temp: {0}",
        (from t in winterTemps select t).Max());
    Console.WriteLine("Min temp: {0}",
        (from t in winterTemps select t).Min());
    Console.WriteLine("Average temp: {0}",
        (from t in winterTemps select t).Average());
    Console.WriteLine("Sum of all temps: {0}",
        (from t in winterTemps select t).Sum());
}
```

Эти примеры должны предоставить достаточно сведений, чтобы вы могли уверенно приступить к построению собственных выражений запросов LINQ. Существуют и дополнительные операции, которые пока еще не рассматривались; они будут описаны позже, когда речь пойдет о связанных технологиях LINQ. В завершение вводного экскурса в LINQ, оставшаяся часть главы посвящена деталям отношений между операциями запросов LINQ и лежащей в основе объектной моделью.

Исходный код. Проект `FunWithLinqExpressions` доступен в подкаталоге `Chapter 12`.

Внутреннее представление операторов запросов LINQ

К настоящему моменту вы уже знакомы с процессом построения выражений запросов с использованием различных операций запросов C# (таких как `from`, `in`, `where`, `orderby` и `select`). Также вам известно, что некоторая функциональность API-интерфейса `LINQ to Objects` может быть доступна только при вызове расширяющих методов класса `Enumerable`. Реальность, однако, заключается в том, что компилятор C# на этапе компиляции транслирует все операции C# LINQ в вызовы методов класса `Enumerable`. Фактически это означает, что при желании можно строить все операторы LINQ, не используя ничего помимо лежащей в основе объектной модели.

Подавляющее большинство методов `Enumerable` прототипированы так, чтобы принимать в качестве аргументов делегаты. В частности, многие методы требуют обобщенного делегата по имени `Func<>`, который был представлен во время рассмотрения обобщенных делегатов в главе 9. Взгляните на метод `Where()` класса `Enumerable`, который вызывается автоматически, когда применяется LINQ-операция `where`:

```
// Перегруженные версии метода Enumerable.Where<T>().
// Обратите внимание, что второй параметр имеет тип System.Func<>.
public static IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource> source,
    System.Func<TSource,int,bool> predicate)

public static IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource> source,
    System.Func<TSource,bool> predicate)
```

Делегат `Func<>` (как следует из его имени) представляет шаблон функции с набором до 16 аргументов и возвращаемым значением. Просмотрев этот тип в браузере объектов Visual Studio, можно обратить внимание на многообразие форм делегата `Func<>`. Например:

```
// Различные форматы делегата Func<>.
public delegate TResult Func<T1,T2,T3,T4,TResult>(T1 arg1, T2 arg2, T3 arg3, T4 arg4)
public delegate TResult Func<T1,T2,T3,TResult>(T1 arg1, T2 arg2, T3 arg3)
public delegate TResult Func<T1,T2,TResult>(T1 arg1, T2 arg2)
public delegate TResult Func<T1,TResult>(T1 arg1)
public delegate TResult Func<TResult>()
```

Поскольку множество членов `System.Linq.Enumerable` требуют при вызове на входе делегат, можно либо вручную создать новый тип делегата и разработать для него необходимые целевые методы, воспользовавшись анонимным методом C#, либо определить подходящее лямбда-выражение. Независимо от выбранного подхода, конечный результат будет одним и тем же.

Хотя простейший способ построения запросов LINQ предусматривает применение операций запросов C# LINQ, давайте все-таки кратко рассмотрим все возможные подходы, просто чтобы увидеть связь между операциями запросов C# и лежащим в основе типом `Enumerable`.

Построение выражений запросов с использованием операций запросов

Для начала создадим новое консольное приложение по имени `LinqUsingEnumerable`. В классе `Program` будет определен набор статических вспомогательных методов (каждый из которых вызывается из метода `Main()`) для иллюстрации различных способов построения выражений запросов LINQ.

Первый метод, `QueryStringsWithOperators()`, обеспечивает наиболее прямолинейный способ построения выражений запросов и идентичен коду примера `LinqOverArray`, приведенному ранее в этой главе:

```
static void QueryStringWithOperators()
{
    Console.WriteLine("***** Using Query Operators *****");
    string[] currentVideoGames = {"Morrowind", "Uncharted 2",
        "Fallout 3", "Daxter", "System Shock 2"};
    var subset = from game in currentVideoGames
        where game.Contains(" ") orderby game select game;
    foreach (string s in subset)
        Console.WriteLine("Item: {0}", s);
}
```

Очевидное преимущество применения операций запросов C# при построении выражений запросов связано с тем, что делегаты `Func<>` и вызовы типа `Enumerable` остаются вне поля зрения и внимания, поскольку работа компилятора C# состоит в выполнении необходимой трансляции. Несомненно, создание выражений LINQ с использованием различных операций запросов (`from`, `in`, `where` или `orderby`) является наиболее распространенным и простым подходом.

Построение выражений запросов с использованием типа `Enumerable` и лямбда-выражений

Имейте в виду, что используемые здесь операции запросов LINQ — это на самом деле сокращенные версии вызова различных расширяющих методов, определенных в типе `Enumerable`. Рассмотрим следующий метод `QueryStringsWithEnumerableAndLambdas()`, который обрабатывает локальный массив строк, но на этот раз в нем напрямую применяются расширяющие методы `Enumerable`:

```
static void QueryStringsWithEnumerableAndLambdas()
{
    Console.WriteLine("***** Using Enumerable / Lambda Expressions *****");
    string[] currentVideoGames = {"Morrowind", "Uncharted 2",
        "Fallout 3", "Daxter", "System Shock 2"};

    // Построить выражение запроса с использованием расширяющих методов,
    // предоставленных типу Array через тип Enumerable.
    var subset = currentVideoGames.Where(game => game.Contains(" "))
        .OrderBy(game => game).Select(game => game);

    // Вывести на консоль результаты.
    foreach (var game in subset)
        Console.WriteLine("Item: {0}", game);

    Console.WriteLine();
}
```

Код начинается с вызова расширяющего метода `Where()` на строковом массиве `currentVideoGames`. Вспомните, что класс `Array` получает этот расширяющий метод от класса `Enumerable`. Метод `Enumerable.Where()` требует параметра-делегата `System.Func<T1, TResult>`. Первый параметр типа этого делегата представляет совместимые с `IEnumerable<T>` данные для обработки (в рассматриваемом случае это массив строк), в то время как второй — результирующие данные метода, которые получаются от единственного оператора, вставленного в лямбда-выражение.

Возвращаемое значение метода `Where()` в этом примере кода скрыто от глаз, но “за кулисами” работа происходит с типом `OrderedEnumerable`. На этом объекте вызывается обобщенный метод `OrderBy()`, который также принимает параметр — делегат `Func<>`. На этот раз производится передача всех элементов по очереди через соответствующее лямбда-выражение. Конечным результатом вызова `OrderBy()` будет новая упорядоченная последовательность начальных данных.

И, наконец, производится вызов метода `Select()` на последовательности, возвращенной `OrderBy()`, который в конечном итоге вернет результирующий набор данных, сохраняемый в неявно типизированной переменной по имени `subset`.

Приведенный запрос LINQ несколько сложнее понять, чем предыдущий пример с операциями запросов C# LINQ. Часть этой сложности, конечно, связана с вызовами через операцию точки. Ниже показан в точности тот же запрос, но с выделением каждого шага в отдельный фрагмент:

```
static void QueryStringsWithEnumerableAndLambdas2()
{
    Console.WriteLine("***** Using Enumerable / Lambda Expressions *****");
    string[] currentVideoGames = {"Morrowind", "Uncharted 2",
        "Fallout 3", "Daxter", "System Shock 2"};

    // Разделить на фрагменты!
    var gamesWithSpaces = currentVideoGames.Where(game => game.Contains(" "));
    var orderedGames = gamesWithSpaces.OrderBy(game => game);
    var subset = orderedGames.Select(game => game);

    foreach (var game in subset)
        Console.WriteLine("Item: {0}", game);

    Console.WriteLine();
}
```

Как видите, построение выражения запроса LINQ с прямым использованием методов класса `Enumerable` намного более многословно, чем с применением операций запросов C#. Кроме того, поскольку методы `Enumerable` требуют параметров-делегатов, обычно необходимо писать лямбда-выражения, чтобы обеспечить обработку входных данных лежащей в основе целью делегата.

Построение выражений запросов с использованием типа `Enumerable` и анонимных методов

Учитывая, что лямбда-выражения C# — это просто сокращенная нотация вызова анонимных методов, рассмотрим третье выражение запроса во вспомогательном методе `QueryStringsWithAnonymousMethods()`:

```
static void QueryStringsWithAnonymousMethods()
{
    Console.WriteLine("***** Using Anonymous Methods *****");
    string[] currentVideoGames = {"Morrowind", "Uncharted 2",
        "Fallout 3", "Daxter", "System Shock 2"};
    // Построить необходимые делегаты Func<> с использованием анонимных методов.
    Func<string, bool> searchFilter =
        delegate(string game) { return game.Contains(" "); };
    Func<string, string> itemToProcess = delegate(string s) { return s; };
    // Передать делегаты в методы Enumerable.
    var subset = currentVideoGames.Where(searchFilter)
        .OrderBy(itemToProcess).Select(itemToProcess);
    // Вывести на консоль результаты.
    foreach (var game in subset)
        Console.WriteLine("Item: {0}", game);
    Console.WriteLine();
}
```

Этот вариант выражения запроса еще более многословен, потому что делегаты `Func<>` создаются вручную с использованием методов `Where()`, `OrderBy()` и `Select()` класса `Enumerable`. Положительная сторона этого подхода связана с тем, что синтаксис анонимных методов позволяет заключить всю обработку, выполняемую делегатами, в одном определении метода. Тем не менее, этот метод функционально эквивалентен методам `QueryStringsWithEnumerableAndLambdas()` и `QueryStringsWithOperators()`, которые рассматривались в предыдущих разделах.

Построение выражений запросов с использованием типа `Enumerable` и низкоуровневых делегатов

Наконец, чтобы построить выражение запроса, используя *по-настоящему* многословный подход, можно отказаться от применения синтаксиса лямбда-выражений и анонимных методов и непосредственно создавать цели делегатов для каждого типа `Func<>`. Ниже показана финальная итерация выражения запроса, смоделированная внутри нового типа класса по имени `VeryComplexQueryExpression`:

```
class VeryComplexQueryExpression
{
    public static void QueryStringsWithRawDelegates()
    {
        Console.WriteLine("***** Using Raw Delegates *****");
        string[] currentVideoGames = {"Morrowind", "Uncharted 2",
            "Fallout 3", "Daxter", "System Shock 2"};
        // Построить необходимые делегаты Func<>.
        Func<string, bool> searchFilter = new Func<string, bool>(Filter);
        Func<string, string> itemToProcess = new Func<string, string>(ProcessItem);
        // Передать делегаты в методы Enumerable.
        var subset = currentVideoGames
            .Where(searchFilter).OrderBy(itemToProcess).Select(itemToProcess);
        // Вывести на консоль результаты.
        foreach (var game in subset)
            Console.WriteLine("Item: {0}", game);
    }
}
```

```

    Console.WriteLine();
}

// Цели делегатов.
public static bool Filter(string game) {return game.Contains(" ");}
public static string ProcessItem(string game) { return game; }
}

```

Чтобы протестировать эту итерацию логики обработки строк, нужно вызвать этот метод в `Main()` класса `Program` следующим образом:

```
VeryComplexQueryExpression.QueryStringsWithRawDelegates();
```

Если теперь запустить приложение, чтобы опробовать все возможные подходы, вывод окажется идентичным, независимо от выбранного пути. Относительно выражений запросов и их скрытого представления следует запомнить перечисленные ниже моменты.

1. Выражения запросов создаются с использованием различных операций запросов C#.
2. Операции запросов — это просто сокращенная нотация вызова расширяющих методов, определенных в типе `System.Linq.Enumerable`.
3. Многие методы `Enumerable` принимают в качестве параметров делегаты (в частности, `Func<>`).
4. Любой метод, ожидающий параметр-делегат, может принимать вместо него лямбда-выражение.
5. Лямбда-выражения — это просто замаскированные анонимные методы (которые значительно улучшают читаемость).
6. Анонимные методы являются сокращенной нотацией для размещения низкоуровневого делегата и ручного построения целевого метода делегата.

Приведенная выше информация поможет понять, что на самом деле происходит “за кулисами” дружественных к пользователю операций запросов C#.

Исходный код. Проект `LinqUsingEnumerable` доступен в подкаталоге `Chapter 12`.

Резюме

LINQ — это набор взаимосвязанных технологий, которые были разработаны для обеспечения единой, симметричной манеры взаимодействия с различными формами данных. Как объяснялось на протяжении этой главы, **LINQ** может взаимодействовать с любым типом, реализующим интерфейс `IEnumerable<T>`, включая простые массивы, а также обобщенные и необобщенные коллекции данных.

Было показано, что работа с технологиями **LINQ** обеспечивается несколькими средствами языка C#. Например, учитывая тот факт, что выражения запросов **LINQ** могут возвращать любое количество результирующих наборов, для представления лежащего в основе типа данных принято использовать ключевое слово `var`. Кроме того, лямбда-выражения, синтаксис инициализации объектов и анонимные типы позволяют строить очень функциональные и компактные запросы **LINQ**.

Что более важно, вы увидели, что операции запросов C# **LINQ** на самом деле являются просто сокращенными нотациями вызовов статических членов типа `System.Linq.Enumerable`. Как было показано, большинство членов `Enumerable` оперируют типами делегатов `Func<T>`, которые могут принимать на входе адреса литеральных методов, анонимные методы или лямбда-выражения для выполнения запроса.

Время жизни объектов

К этому моменту было предоставлено немало сведений о создании специальных типов классов в C#. Теперь речь пойдет о том, как среда CLR управляет размещенными экземплярами классов (т.е. объектами) с помощью процесса *сборки мусора*. Программистам на C# никогда не приходится непосредственно удалять управляемый объект из памяти (в языке C# нет даже ключевого слова, подобного `delete`). Вместо этого объекты .NET размещаются в области памяти, которая называется *управляемой кучей*, откуда они автоматически удаляются сборщиком мусора в какой-то момент в будущем.

После рассмотрения ключевых деталей процесса сборки мусора в настоящей главе будет показано, как программно взаимодействовать со сборщиком мусора с помощью класса `System.GC` (хотя в большинстве проектов .NET это обычно не требуется). Кроме того, мы рассмотрим, как с применением виртуального метода `System.Object.Finalize()` и интерфейса `IDisposable` создавать классы, способные освобождать внутренние *неуправляемые ресурсы* в определенное время и в предсказуемой манере.

Кроме того, будут описаны некоторые новые функциональные возможности сборки мусора, появившиеся в версии .NET 4.0, включая фоновую сборку мусора и ленивое (отложенное) создание объектов с использованием обобщенного класса `System.Lazy<>`. После изучения материалов настоящей главы должно появиться четкое представление об управлении объектами .NET в среде CLR.

Классы, объекты и ссылки

Перед изучением тем, рассматриваемых в настоящей главе, сначала необходимо немного больше прояснить различия между классами, объектами и ссылочными переменными. Вспомните, что класс представляет собой не что иное, как схему, которая описывает то, каким образом экземпляр данного типа должен выглядеть и вести себя в памяти. Классы, естественно, определяются в файлах кода (которым по соглашению назначается расширение `*.cs`). Взгляните на следующий простой класс `Car`, определенный в новом проекте консольного приложения C# по имени `SimpleGC`:

```
// Car.cs
public class Car
{
    public int CurrentSpeed {get; set;}
    public string PetName {get; set;}

    public Car(){}
    public Car(string name, int speed)
    {
        PetName = name;
        CurrentSpeed = speed;
    }
}
```

```

public override string ToString()
{
    return string.Format("{0} is going {1} MPH",
        PetName, CurrentSpeed);
}
}

```

После того как класс определен, с применением ключевого слова `new` языка C# можно размещать в памяти любое количество его объектов. Однако при этом следует помнить, что ключевое слово `new` возвращает ссылку на объект в куче, а не фактический объект. Если ссылочная переменная объявляется как локальная переменная в контексте метода, она сохраняется в стеке для дальнейшего использования в приложении. Для вызова членов объекта к сохраненной ссылке применяется операция точки C#:

```

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** GC Basics *****");
        // Создать новый объект Car в управляемой куче.
        // Возвращается ссылка на этот объект (refToMyCar).
        Car refToMyCar = new Car("Zippy", 50);
        // Операция точки (.) используется для вызова членов
        // данного объекта с применением ссылочной переменной.
        Console.WriteLine(refToMyCar.ToString());
        Console.ReadLine();
    }
}

```

На рис. 13.1 показаны отношения между классами, объектами и ссылками.

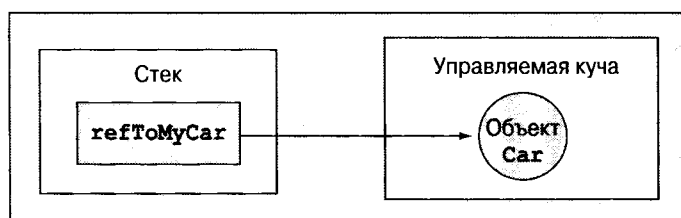


Рис. 13.1. Ссылки на объекты в управляемой куче

На заметку! Вспомните из главы 4, что структуры представляют собой *типы значений*, которые всегда размещаются прямо в стеке и никогда не попадают в управляемую кучу .NET. Размещение в куче происходит только при создании экземпляров классов.

Базовые сведения о времени жизни объектов

При создании приложений на C# можно смело полагать, что исполняющая среда .NET самостоятельно позаботится об управляемой куче без непосредственного вмешательства со стороны программиста. На самом деле “золотое правило” по управлению памятью в .NET звучит просто.

На заметку! Размещайте объект в управляемой куче с использованием ключевого слова `new` и просто забывайте о нем.

После создания объект будет автоматически удален сборщиком мусора тогда, когда в нем отпадет необходимость. Разумеется, возникает вопрос о том, каким образом сборщик мусора определяет момент, когда в объекте отпадает необходимость? В двух словах на этот вопрос можно ответить так: сборщик мусора удаляет объект из кучи только тогда, когда тот становится *недостижимым* ни в одной части кодовой базы. Например, добавим в класс Program метод, который размещает в памяти объект Car:

```
public static void MakeACar()
{
    // Если myCar является единственной ссылкой на объект
    // Car, тогда при возврате результата данным
    // методом объект Car *может* быть уничтожен.
    Car myCar = new Car();
    ...
}
```

Обратите внимание, что ссылка на объект Car (myCar) была создана непосредственно внутри метода MakeACar() и не передавалась за пределы определяющей области действия (ни в виде возвращаемого значения, ни в виде параметров ref/out). Поэтому после завершения вызова данного метода ссылка myCar окажется недостижимой, а объект Car, соответственно — кандидатом на удаление сборщиком мусора. Следует, однако, понять, что нельзя гарантировать немедленное удаление этого объекта из памяти сразу же после выполнения метода MakeACar(). Все, что в данный момент можно предполагать, так это то, что когда в CLR-среде будет в следующий раз производиться сборка мусора, объект myCar может быть безопасно уничтожен.

Как станет ясно со временем, программирование в среде с автоматической сборкой мусора значительно облегчает разработку приложений. Программистам на языке C++ хорошо известно, что если они специально не позаботятся об удалении размещаемых в куче объектов, утечки памяти не заставят себя долго ждать. На самом деле отслеживание проблем, связанных с утечками памяти, является одним из самых длительных (и утомительных) аспектов программирования в неуправляемых средах. Благодаря назначению сборщика мусора ответственным за уничтожение объектов, обязанности по управлению памятью, по сути, сняты с программиста и возложены на среду CLR.

Код CIL для ключевого слова new

При обнаружении ключевого слова new компилятор C# вставляет в реализацию метода CIL-инструкцию newobj. Если скомпилировать текущий пример кода и заглянуть в полученную сборку с помощью утилиты ildasm.exe, то можно обнаружить внутри метода MakeACar() следующие операторы CIL:

```
.method private hidebysig static void MakeACar() cil managed
{
    // Code size 8 (0x8)
    .maxstack 1
    .locals init ([0] class SimpleGC.Car)
    IL_0000: nop
    IL_0001: newobj instance void SimpleGC.Car::.ctor()
    IL_0006: stloc.0
    IL_0007: ret
} // end of method Program::MakeACar
```

Прежде чем ознакомиться с точными правилами, которые определяют момент, когда объект должен удаляться из управляемой кучи, давайте более подробно рассмотрим роль CIL-инструкции newobj. Для начала важно понять, что управляемая куча представляет собой нечто большее, чем просто произвольный участок памяти, к которому CLR

имеет доступ. Сборщик мусора .NET “убирает” кучу довольно тщательно, причем (при необходимости) даже сжимает пустые блоки памяти с целью оптимизации. Чтобы ему было легче это делать, в управляемой куче поддерживается указатель (обычно называемый *указателем на следующий объект* или *указателем на новый объект*), который показывает, где точно будет размещаться следующий объект. Таким образом, инструкция `newobj` заставляет среду CLR выполнить перечисленные ниже ключевые операции.

- Вычисление общего объема памяти, требуемого для размещения объекта (включая память, необходимую для членов данных и базовых классов).
- Проверка, действительно ли в управляемой куче хватает места для хранения размещаемого объекта. Если места достаточно, производится вызов указанного конструктора с возвратом вызывающему коду ссылки на новый объект в памяти, адрес которого совпадает с последней позицией указателя на следующий объект.
- И, наконец, перед возвратом ссылки вызывающему коду указатель на следующий объект продвигается, чтобы указывать на следующую доступную ячейку в управляемой куче.

Весь описанный процесс схематично представлен на рис. 13.2.

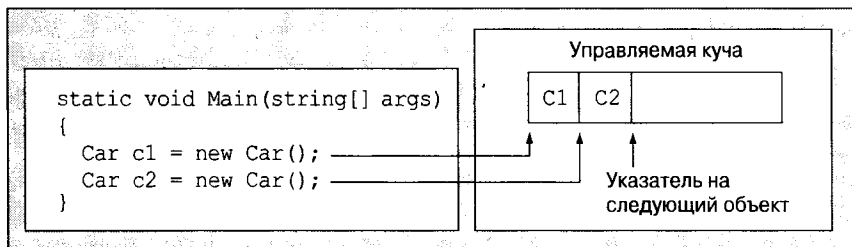


Рис. 13.2. Детали размещения объектов в управляемой куче

Из-за постоянного размещения объектов приложением пространство в управляемой куче может со временем заполниться. В случае если при обработке следующей инструкции `newobj` среда CLR обнаруживает, что в управляемой куче не хватает пространства для размещения запрашиваемого типа, она приступает к сборке мусора, пытаясь освободить память. Таким образом, следующее правило сборки мусора также звучит довольно просто.

На заметку!. Если в управляемой куче не хватает пространства для размещения запрашиваемого объекта, происходит сборка мусора.

Однако то, как происходит сборка мусора, зависит от версии платформы .NET, под управлением которой выполняется приложение. Различия будут описаны далее в этой главе.

Установка объектных ссылок в null

Программисты на C/C++ часто устанавливают переменные указателей в `null`, чтобы гарантировать, что они больше не ссылаются на неуправляемую память. С учетом этого может возникнуть вопрос о том, что в конечном итоге происходит после установки в `null` ссылок на объекты в C#? Для примера изменим метод `MakeACar()` следующим образом:

```
static void MakeACar()
{
    Car myCar = new Car();
    myCar = null;
}
```

Когда ссылки на объекты устанавливаются в `null`, компилятор C# генерирует CIL-код, который заботится о том, чтобы ссылка (в этом примере `myCar`) больше не ссылалась ни на какой объект. Если теперь снова воспользоваться утилитой `ildasm.exe` для просмотра CIL-кода модифицированного метода `MakeACar()`, можно обнаружить в нем код операции `ldnull` (заталкивает значение `null` в виртуальный стек выполнения), за которым следует код операции `stloc.0` (устанавливает для переменной ссылку `null`):

```
.method private hidebysig static void MakeACar() cil managed
{
    // Code size 10 (0xa)
    .maxstack 1
    .locals init ([0] class SimpleGC.Car myCar)
    IL_0000: nop
    IL_0001: newobj instance void SimpleGC.Car::.ctor()
    IL_0006: stloc.0
    IL_0007: ldnull
    IL_0008: stloc.0
    IL_0009: ret
} // end of method Program::MakeACar
```

Однако обязательно следует понять, что установка ссылки в `null` никоим образом не вынуждает сборщик мусора немедленно запуститься и удалить объект из кучи. Единственное, что достигается такой установкой — явный разрыв связи между ссылкой и объектом, на который она ранее указывала. С учетом этого, установка ссылок в `null` в C# имеет гораздо меньше последствий, чем в других языках, основанных на C; тем не менее, никакого вреда она не причиняет.

Роль корневых элементов приложения

Теперь вернемся к вопросу о том, каким образом сборщик мусора определяет момент, когда объект уже более не нужен. Чтобы разобраться в стоящих за этим деталях, необходимо знать, что собой представляют *корневые элементы приложения*. Выразаясь просто, *корневой элемент* — это местоположение в памяти, в котором содержится ссылка на объект в управляемой куче. Строго говоря, корневой элемент может относиться к любой из следующих категорий:

- ссылки на глобальные объекты (хотя в C# они не разрешены, код CIL позволяет размещать глобальные объекты);
- ссылки на любые статические объекты или статические поля;
- ссылки на локальные объекты в пределах кодовой базы приложения;
- ссылки на объектные параметры, передаваемые методу;
- ссылки на объекты, ожидающие *финализации* (об этом подробно рассказывается далее в главе);
- любой регистр центрального процессора, который ссылается на объект.

Во время процесса сборки мусора исполняющая среда будет исследовать объекты в управляемой куче, чтобы определить, являются ли они по-прежнему достижимыми (т.е. корневыми) для приложения. Для этого среда CLR будет строить *граф объектов*, который представляет каждый достижимый объект в куче. Более подробно графы объектов

будут описаны при рассмотрении сериализации объектов в главе 20. Пока главное усвоить то, что такие графы применяются для документирования всех достижимых объектов. Кроме того, следует иметь в виду, что сборщик мусора никогда не будет создавать граф для одного и того же объекта дважды, избегая необходимости в выполнении подсчета циклических ссылок, который характерен для программирования COM.

Предположим, что в управляемой куче содержится набор объектов с именами A, B, C, D, E, F и G. Во время сборки мусора эти объекты (а также любые внутренние объектные ссылки, которые они могут содержать) будут исследованы на предмет активных корневых элементов. После построения графа все недостижимые объекты (которыми в примере являются объекты C и F) помечаются как являющиеся мусором.

На рис. 13.3 показано, как может выглядеть граф объектов в только что описанном сценарии (линии со стрелками следует воспринимать как "зависит от" или "требуется"; например, "E зависит от G и B", "A не зависит ни от чего" и т.д.).

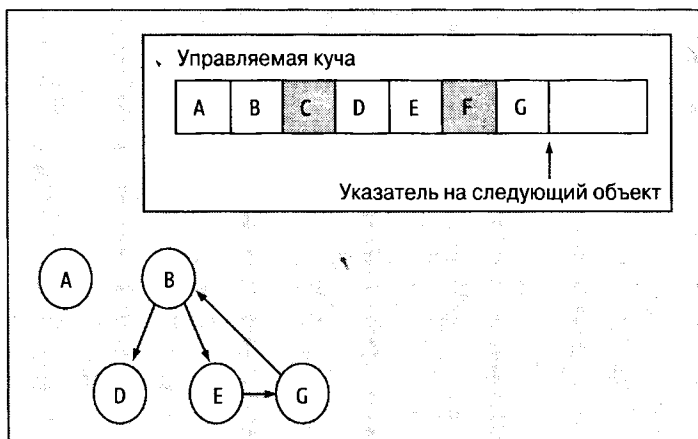


Рис. 13.3. Графы объектов строятся с целью определения объектов, достижимых для корневых элементов приложения

После того как объект помечен для уничтожения (в данном случае это объекты C и F, поскольку в графе объектов они во внимание не принимаются), они будут удалены из памяти. Оставшееся пространство в куче сжимается, что, в свою очередь, вынуждает CLR модифицировать набор активных корневых элементов приложения (и лежащих в основе указателей), чтобы они ссылались на правильные местоположения в памяти (это делается автоматически и прозрачно). И, наконец, указатель на следующий объект настраивается так, чтобы указывать на следующий доступный участок памяти. Конечный результат этих изменений можно видеть на рис. 13.4.

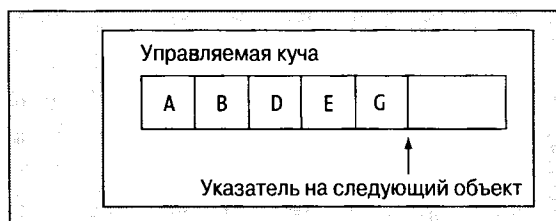


Рис. 13.4. Очищенная и сжатая куча

На заметку! Строго говоря, сборщик мусора использует две отдельных кучи, одна из которых предназначена специально для хранения очень больших объектов. Доступ к ней во время сборки мусора производится менее часто из-за возможного снижения производительности, связанного с перемещением больших объектов. Невзирая на этот факт, управляемую кучу можно спокойно воспринимать как единую область памяти.

Понятие поколений объектов

Когда среда CLR пытается найти недостижимые объекты, она не проверяет буквально каждый находящийся в куче объект. Очевидно, это потребовало бы массы времени, особенно в крупных (реальных) приложениях.

Для оптимизации процесса каждому объекту в куче назначается определенное "поколение". Идея, лежащая в основе поколений, довольно проста: чем дольше объект находится в куче, тем выше вероятность того, что он там и будет оставаться. Например, класс, который определяет главное окно настольного приложения, будет оставаться в памяти вплоть до завершения приложения. В противоположность этому, объекты, которые были размещены в куче лишь недавно (такие как объект, размещенный в области действия метода), скорее всего, станут недостижимым довольно быстро. Исходя из этих предположений, каждый объект в куче относится к одному из перечисленных ниже поколений.

- **Поколение 0.** Идентифицирует новый размещенный объект, который еще никогда не помечался как подлежащий сборке мусора.
- **Поколение 1.** Идентифицирует объект, который уже пережил один процесс сборки мусора (т.е. был помечен как подлежащий сборке мусора, но не был удален из-за наличия достаточного пространства в куче).
- **Поколение 2.** Идентифицирует объект, которому удалось пережить более одного прохода сборщика мусора.

На заметку! Поколения 0 и 1 называются *эфемерными* (недолговечными). В следующем разделе будет показано, что процесс сборки мусора трактует эфемерные поколения по-разному.

Сборщик мусора сначала анализирует все объекты, которые относятся к поколению 0. Если после их удаления остается достаточное количество свободной памяти, все остальные (уцелевшие) объекты повышаются до поколения 1. Чтобы увидеть, как поколение, к которому относится объект, влияет на процесс сборки мусора, обратите внимание на рис. 13.5, где схематически показано, как набор уцелевших объектов поколения 0 (A, B и E) повышается до следующего поколения после освобождения требуемого объема памяти.

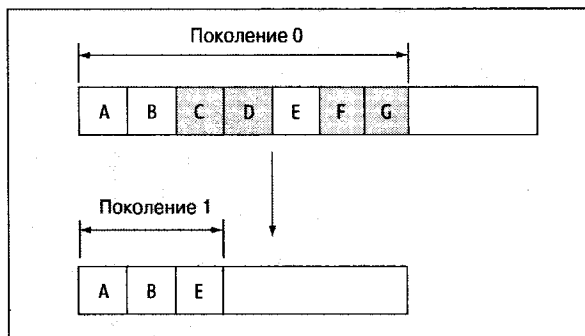


Рис. 13.5. Объекты поколения 0, которые уцелили после сборки мусора, повышаются до поколения 1

Если все объекты поколения 0 проверены, но по-прежнему требуется дополнительная память, начинают проверяться на предмет достижимости и подвергаться сборке мусора объекты поколения 1. Уцелевшие объекты поколения 1 повышаются до поколения 2. Если же сборщику мусора *все еще* требуется дополнительная память, начинают проверяться на предмет достижимости объекты поколения 2. Объекты поколения 2, которым удастся пережить сборку мусора на этом этапе, остаются объектами того же поколения 2, поскольку достигнут заранее определенный верхний предел поколений объектов.

В заключение следует отметить, что за счет назначения объектам в куче определенного поколения более новые объекты (такие как локальные переменные) будут удаляться быстрее, тогда как более старые (наподобие главного окна приложения) будут существовать дольше.

Параллельная сборка мусора в .NET 1.0 — .NET 3.5

До выхода версии .NET 4.0 очистка неиспользуемых объектов в исполняющей среде производилась с применением техники *параллельной сборки мусора*. В этой модели при выполнении сборки мусора для любых объектов поколения 0 или 1 (т.е. *эфемерных поколений*) сборщик мусора временно приостанавливал все активные *потоки* внутри текущего процесса, чтобы приложение не могло получать доступ к управляемой куче вплоть до завершения процесса сборки.

Потоки более подробно рассматриваются в главе 19, а пока что поток можно считать просто путем выполнения внутри функционирующей программы. По завершении цикла сборки мусора приостановленным потокам разрешалось продолжить свою работу. К счастью, сборщик мусора в .NET 3.5 (и предшествующих версиях) был хорошо оптимизирован и потому связанные с этим короткие перерывы в работе приложения редко становились заметными (а то и вообще никогда).

В качестве оптимизации, параллельная сборка мусора позволяла производить очистку объектов, которые не были обнаружены ни в одном из эфемерных поколений, в отдельном потоке. Это сокращало (но не устраняло) необходимость в приостановке активных потоков исполняющей средой .NET. Более того, параллельная сборка мусора позволяла программам продолжать размещать объекты в куче во время сборки объектов не эфемерных поколений.

Фоновая сборка мусора в .NET 4.0 и последующих версиях

Начиная с .NET 4.0, сборщик мусора способен решать вопрос с приостановкой потоков при очистке объектов в управляемой куче, используя технику *фоновой сборки мусора*. Несмотря на название, это вовсе не означает, что вся сборка мусора теперь происходит в дополнительных фоновых потоках выполнения. На самом деле, если фоновая сборка мусора осуществляется для объектов, относящихся к не эфемерному поколению, то исполняющая среда .NET теперь может производить сборку объектов эфемерных поколений в отдельном фоновом потоке.

Механизм сборки мусора в .NET 4.0 и последующих версиях был улучшен так, чтобы еще больше сократить время приостановки потока, связанной со сборкой мусора. Благодаря этим изменениям, процесс очистки неиспользуемых объектов поколения 0 или 1 был оптимизирован и позволяет получать более высокие показатели производительности приложений (что действительно важно для систем, работающих в реальном времени и нуждающихся в небольших и предсказуемых перерывах на сборку мусора).

Однако следует понимать, что ввод такой новой модели сборки мусора никоим образом не отражается на способе построения приложений .NET. Теперь практически всегда

можно просто позволить сборщику мусора .NET выполнять свою работу без непосредственного вмешательства (и радоваться тому, что разработчики в Microsoft продолжают улучшать процесс сборки мусора прозрачным образом).

Тип System.GC

Сборка `mscorlib.dll` предоставляет класс по имени `System.GC`, который позволяет программно взаимодействовать со сборщиком мусора за счет обращения к его статическим членам. Необходимость в непосредственном использовании этого класса в разрабатываемом коде возникает редко (если вообще возникает). Обычно единственным случаем, когда нужно иметь дело с членами `System.GC`, является создание классов, которые внутренне работают с *неуправляемыми ресурсами*. Это может быть, например, класс, который делает вызовы API-интерфейса Windows, основанного на C, используя протокол вызовов платформы .NET, или какая-то низкоуровневая и сложная логика взаимодействия с COM. В табл. 13.1 приведено краткое описание некоторых наиболее интересных членов класса `System.GC` (полные сведения можно найти в документации .NET Framework 4.5 SDK).

Таблица 13.1. Избранные члены класса System.GC

Член	Описание
<code>AddMemoryPressure()</code> , <code>RemoveMemoryPressure()</code>	Позволяют указывать числовое значение, отражающее "уровень срочности", который вызывающий объект применяет в отношении к сборке мусора. Следует иметь в виду, что эти методы должны изменять уровень давления <i>в тандеме</i> и, следовательно, никогда не удалять больше давления, чем было добавлено
<code>Collect()</code>	Заставляет сборщик мусора провести сборку мусора. Этот метод перегружен для указания собираемого поколения, а также режима сборки мусора (с помощью перечисления <code>GC.CollectionMode</code>)
<code>CollectionCount()</code>	Возвращает числовое значение, показывающее, сколько раз производилась сборка мусора для заданного поколения
<code>GetGeneration()</code>	Возвращает поколение, к которому в настоящий момент относится объект
<code>GetTotalMemory()</code>	Возвращает приблизительный объем памяти (в байтах), занятой в настоящий момент в управляемой куче. Булевский параметр указывает, должен ли вызов перед возвратом сначала дожидаться выполнения сборки мусора
<code>MaxGeneration</code>	Возвращает максимальное количество поколений, поддерживаемое целевой системой. Например, в .NET 4.0 существуют три возможных поколения: 0, 1 и 2
<code>SuppressFinalize()</code>	Устанавливает флаг, указывающий, что заданный объект не должен вызывать свой метод <code>Finalize()</code>
<code>WaitForPendingFinalizers()</code>	Приостанавливает текущий поток до тех пор, пока не будут финализированы все финализируемые объекты. Обычно вызывается сразу же после вызова метода <code>GC.Collect()</code>

Рассмотрим применение `System.GC` для получения разнообразных деталей, связанных со сборкой мусора, на примере следующего метода `Main()`, в котором используются несколько членов `System.GC`:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with System.GC *****");
    // Вывести приблизительное количество байтов в куче.
    Console.WriteLine("Estimates bytes on heap: {0}",
        GC.GetTotalMemory(false));
    // MaxGeneration основано на 0, поэтому при выводе добавить 1.
    Console.WriteLine("This OS has {0} object generations.\n",
        (GC.MaxGeneration + 1));
    Car refToMyCar = new Car("Zippy", 100);
    Console.WriteLine(refToMyCar.ToString());
    // Вывести поколение объекта refToMyCar.
    Console.WriteLine("Generation of refToMyCar is: {0}",
        GC.GetGeneration(refToMyCar));
    Console.ReadLine();
}
```

Принудительный запуск сборщика мусора

Основное предназначение сборщика мусора .NET заключается в управлении памятью вместо разработчиков. Тем не менее, в очень редких случаях может оказаться полезным принудительный запуск сборщика мусора с помощью `GC.Collect()`. Взаимодействие с процессом сборки мусора может требоваться в двух следующих ситуациях.

- Приложение входит в блок кода, который не должен прерываться возможной сборкой мусора.
- Приложение только что закончило размещать исключительно большое количество объектов и нуждается в скорейшем освобождении большого объема памяти.

Если выясняется, что выполнение сборщиком мусора проверки на предмет наличия недостижимых объектов обеспечит выгоду, можно инициировать процесс сборки мусора явным образом, как показано ниже:

```
static void Main(string[] args)
{
    ...
    // Принудительно запустить сборку мусора и
    // ожидать финализации каждого из объектов.
    GC.Collect();
    GC.WaitForPendingFinalizers();
    ...
}
```

Когда сборка мусора запускается вручную, всегда нужно вызывать метод `GC.WaitForPendingFinalizers()`. Это даст возможность всем *финализируемым объектам* (которые рассматриваются в следующем разделе) выполнить любую необходимую очистку перед продолжением работы программы. “За кулисами” метод `GC.WaitForPendingFinalizers()` приостанавливает вызывающий поток на время сборки мусора. Это очень хорошо, т.к. гарантирует невозможность обращения в коде к методам объекта, который в текущий момент уничтожается.

Методу `GC.Collect()` можно передать числовое значение, идентифицирующее самое старое поколение, для которого должна выполняться сборка мусора. Например, чтобы указать среде CLR на необходимость исследования только объектов поколения 0, можно написать следующий код:

```
static void Main(string[] args)
{
    ...
    // Исследовать только объекты поколения 0.
    GC.Collect(0);
    GC.WaitForPendingFinalizers();
    ...
}
```

Вдобавок методу `Collect()` можно передать во втором параметре значение перечисления `GC.CollectionMode`, что позволяет точно настроить способ принудительного запуска сборщика мусора исполняющей средой. Ниже показаны значения, определенные этим перечислением:

```
public enum GC.CollectionMode
{
    Default, // Forced является текущим стандартным значением.
    Forced,  // Указывает исполняющей среде начать сборку мусора немедленно.
    Optimized // Позволяет исполняющей среде выяснить, оптимален
              // ли текущий момент для удаления объектов.
}
```

Как и при любой сборке мусора, в результате вызова `GC.Collect()` уцелевшие объекты переводятся в более высокие поколения. В целях иллюстрации предположим, что метод `Main()` модифицирован следующим образом:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with System.GC *****");
    // Вывести приблизительное количество байтов в куче.
    Console.WriteLine("Estimated bytes on heap: {0}",
        GC.GetTotalMemory(false));
    // MaxGeneration основано на 0.
    Console.WriteLine("This OS has {0} object generations.\n",
        (GC.MaxGeneration + 1));
    Car refToMyCar = new Car("Zippy", 100);
    Console.WriteLine(refToMyCar.ToString());
    // Вывести поколение refToMyCar.
    Console.WriteLine("\nGeneration of refToMyCar is: {0}",
        GC.GetGeneration(refToMyCar));
    // Создать большое количество объектов в целях тестирования.
    object[] tonsOfObjects = new object[50000];
    for (int i = 0; i < 50000; i++)
        tonsOfObjects[i] = new object();
    // Выполнить сборку мусора только для объектов поколения 0.
    GC.Collect(0, GC.CollectionMode.Forced);
    GC.WaitForPendingFinalizers();
    // Вывести поколение refToMyCar.
    Console.WriteLine("Generation of refToMyCar is: {0}",
        GC.GetGeneration(refToMyCar));
    // Посмотреть, существует ли еще tonsOfObjects[9000].
    if (tonsOfObjects[9000] != null)
    {
        Console.WriteLine("Generation of tonsOfObjects[9000] is: {0}",
            GC.GetGeneration(tonsOfObjects[9000]));
    }
}
```



```

else
    Console.WriteLine("tonsOfObjects[9000] is no longer alive.");
// Вывести количество проведенных сборок мусора для разных поколений.
Console.WriteLine("\nGen 0 has been swept {0} times",
    GC.CollectionCount(0));
Console.WriteLine("Gen 1 has been swept {0} times",
    GC.CollectionCount(1));
Console.WriteLine("Gen 2 has been swept {0} times",
    GC.CollectionCount(2));
Console.ReadLine();
}

```

В тестовых целях преднамеренно был создан очень большой массив типа `object` (состоящий из 50 000 элементов). В показанном ниже выводе видно, что хотя в методе `Main()` был сделан только один явный запрос на выполнение сборки мусора (с помощью метода `GC.Collect()`), среда CLR в фоновом режиме провела несколько таких сборок.

```

***** Fun with System.GC *****
Estimated bytes on heap: 70240
This OS has 3 object generations.

Zippy is going 100 MPH

Generation of refToMyCar is: 0
Generation of refToMyCar is: 1
Generation of tonsOfObjects[9000] is: 1

Gen 0 has been swept 1 times
Gen 1 has been swept 0 times
Gen 2 has been swept 0 times

```

К этому моменту детали жизненного цикла объектов должны стать более понятными. В следующем разделе исследование процесса сборки мусора продолжается; будет показано, как создавать *финализируемые объекты* и *освобождаемые объекты*. Очень важно отметить, что описанные далее приемы обычно необходимы только при построении классов C#, которые поддерживают внутренние неуправляемые ресурсы.

Исходный код. Проект SimpleGC доступен в подкаталоге Chapter 13.

Построение финализируемых объектов

В главе 6 уже рассказывалось о том, что в самом главном базовом классе .NET — `System.Object` — имеется виртуальный метод по имени `Finalize()`. В своей стандартной реализации он ничего особенного не делает:

```

// System.Object
public class Object
{
    ...
    protected virtual void Finalize() {}
}

```

За счет его переопределения в специальных классах устанавливается специфическое место для выполнения любой логики очистки, необходимой данному типу. Из-за того, что метод `Finalize()` определен как защищенный (`protected`), вызывать его напрямую из экземпляра класса с помощью операции точки невозможно. Вместо этого метод `Finalize()` (если он поддерживается) будет вызываться *сборщиком мусора* перед удалением объекта из памяти.

На заметку! Переопределять метод `Finalize()` в типах структур не допускается. Это вполне логичное ограничение, поскольку структуры являются типами значений, которые изначально никогда не размещаются в куче и, следовательно, никогда не подвергаются сборке мусора. Тем не менее, при создании структуры, которая содержит ресурсы, нуждающиеся в очистке, можно реализовать интерфейс `IDisposable` (рассматривается далее в главе).

Разумеется, вызов метода `Finalize()` будет происходить (в конечном итоге) либо во время “естественной” сборки мусора, либо при ее принудительной активизации программным образом с помощью `GC.Collect()`. Кроме того, финализатор типа будет автоматически вызываться и при выгрузке из памяти домена, который отвечает за обслуживание приложения. В зависимости от опыта работы с .NET, уже может быть известно, что домены приложений применяются для обслуживания исполняемой сборки и любых необходимых внешних библиотек кода. Если вы еще не знакомы с этой концепцией .NET, то необходимые сведения будут даны в главе 17. Пока необходимо лишь запомнить, что при выгрузке домена приложения из памяти среда CLR автоматически вызывает финализаторы для каждого финализируемого объекта, созданного во время существования домена приложения.

О чем бы ни говорили инстинкты разработчика, подавляющее большинство классов C# не требует написания явной логики очистки или специального финализатора. Причина проста: если в классах используются лишь другие управляемые объекты, все они рано или поздно будут подвергаться сборке мусора. Единственным случаем, когда может возникать потребность в создании класса, способного выполнять после себя процедуру очистки, является работа с *неуправляемыми* ресурсами (такими как низкоуровневые файловые дескрипторы, низкоуровневые неуправляемые подключения к базам данных, фрагменты неуправляемой памяти и т.п.). В рамках платформы .NET неуправляемые ресурсы получаются в результате непосредственного обращения к API-интерфейсу операционной системы с помощью служб `PInvoke` (`Platform Invocation Services` — службы вызова платформы) или в очень сложных сценариях взаимодействия с COM. С учетом этого можно сформулировать еще одно правило сборки мусора.

На заметку! Единственная серьезная причина переопределения `Finalize()` связана с использованием в классе C# неуправляемых ресурсов через `PInvoke` или сложные задачи взаимодействия с COM (обычно посредством членов, определенных в типе `System.Runtime.InteropServices.Marshal`). Это объясняется тем, что в таких сценариях производится манипулирование памятью, которой среда CLR управлять не может.

Переопределение `System.Object.Finalize()`

В редком случае, когда строится класс C#, в котором применяются неуправляемые ресурсы, очевидно, понадобится обеспечить предсказуемое освобождение соответствующей памяти. Для примера создадим новое консольное приложение C# по имени `SimpleFinalize` и вставим в него класс `MyResourceWrapper`, в котором используется неуправляемый ресурс. Теперь необходимо переопределить метод `Finalize()`. Как ни странно, применять для этого ключевое слово `override` в C# нельзя:

```
class MyResourceWrapper
{
    // Ошибка на этапе компиляции!
    protected override void Finalize() { }
}
```

Вместо этого для достижения того же эффекта должен использоваться синтаксис деструктора (почти как в C++). Причина такой альтернативной формы переопределения виртуального метода состоит в том, что при обработке синтаксиса финализатора компилятор автоматически добавляет в неявно переопределяемый метод `Finalize()` приличное количество требуемых элементов инфраструктуры (как будет показано ниже).

Финализаторы в C# выглядят очень похожими на конструкторы тем, что именуются идентично классу, внутри которого определены. Вдобавок они сопровождаются префиксом в виде тильды (~). Однако в отличие от конструкторов, они никогда не снабжаются модификатором доступа (поскольку всегда являются неявно защищенными), не принимают параметров и не могут быть перегружены (в каждом классе может присутствовать только один финализатор).

Ниже приведен специальный финализатор для класса `MyResourceWrapper`, который при вызове выдает звуковой сигнал. Очевидно, этот пример предназначен только для демонстрационных целей. В реальном приложении финализатор будет только освобождать любые неуправляемые ресурсы, и *не будет* взаимодействовать ни с какими другими управляемыми объектами, даже теми, на которые ссылается текущий объект, поскольку нельзя предполагать, что они все еще существуют на момент вызова этого метода `Finalize()` сборщиком мусора.

```
// Переопределить System.Object.Finalize()
// с использованием синтаксиса финализатора.
class MyResourceWrapper
{
    ~MyResourceWrapper()
    {
        // Очистить неуправляемые ресурсы.

        // Выдать звуковой сигнал при уничтожении (только в целях тестирования).
        Console.Beep();
    }
}
```

Если теперь просмотреть CIL-код данного деструктора с помощью утилиты `ildasm.exe`, обнаружится, что компилятор добавил необходимый код для проверки ошибок. Первым делом, он поместил операторы внутри области действия метода `Finalize()` в блок `try` (см. главу 7). Кроме того, он добавил соответствующий блок `finally`, чтобы гарантировать выполнение методов `Finalize()` базовых классов независимо от возникших в контексте `try` исключений.

```
.method family hidebysig virtual instance void
    Finalize() cil managed
{
    // Code size      13 (0xd)
    .maxstack 1
    .try
    {
        IL_0000: ldc.i4      0x4e20
        IL_0005: ldc.i4      0x3e8
        IL_000a: call
            void [mscorlib]System.Console::Beep(int32, int32)
        IL_000f: nop
        IL_0010: nop
        IL_0011: leave.s      IL_001b
    } // end .try
    finally
    {
        IL_0013: ldarg.0
```

```

IL_0014:
    call instance void [mscorlib]System.Object::Finalize()
IL_0019: nop
IL_001a: endfinally
} // end handler
IL_001b: nop
IL_001c: ret
} // end of method MyResourceWrapper::Finalize
// конец метода MyResourceWrapper::Finalize

```

Тестирование класса `MyResourceWrapper` показывает, что звуковой сигнал выдается во время завершения приложения, т.к. среда CLR автоматически вызывает финализаторы при выгрузке домена приложения.

```

static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Finalizers *****\n");
    Console.WriteLine("Hit the return key to shut down this app");
    Console.WriteLine("and force the GC to invoke Finalize()");
    Console.WriteLine("for finalizable objects created in this AppDomain.");
    // Нажмите клавишу <Enter>, чтобы завершить приложение
    // и заставить сборщик мусора вызвать метод Finalize()
    // для всех финализируемых объектов, которые
    // были созданы в домене этого приложения.
    Console.ReadLine();
    MyResourceWrapper rw = new MyResourceWrapper();
}

```

Исходный код. Проект `SimpleFinalize` доступен в подкаталоге `Chapter 13`.

Описание процесса финализации

Чтобы не делать лишнюю работу, следует всегда помнить, что задачей метода `Finalize()` является забота о том, чтобы объект .NET мог освобождать неуправляемые ресурсы во время сборки мусора. Следовательно, при построении класса, в котором неуправляемая память не используется (а так бывает чаще всего), от финализации мало толку. На самом деле, всегда по возможности следует проектировать классы так, чтобы в них не поддерживался метод `Finalize()` по той простой причине, что финализация требует времени.

При размещении объекта в управляемой куче исполняющая среда автоматически определяет, поддерживается ли в нем специальный метод `Finalize()`. Если да, тогда она помечает его как *финализируемый* и сохраняет указатель на него во внутренней очереди, называемой *очередью финализации*. Очередь финализации представляет собой поддерживаемую сборщиком мусора таблицу, указывающую на все объекты, которые должны быть финализированы перед удалением из кучи.

Когда сборщик мусора определяет, что наступило время удалить объект из памяти, он проверяет каждую запись в очереди финализации и копирует объект из кучи в еще одну управляемую структуру, называемую *таблицей объектов, доступных для финализации*. Затем он создает отдельный поток для вызова метода `Finalize()` на каждом объекте из этой таблицы *при следующей сборке мусора*. В результате получается, что для окончательной финализации объекта требуется как минимум две сборки мусора.

Из всего сказанного следует, что хотя финализация объекта действительно гарантирует для объекта возможность освобождения неуправляемых ресурсов, она все равно остается недетерминированной по своей природе, и по причине дополнительной незаметной обработки протекает гораздо медленнее.

Создание освобождаемых объектов

Как было показано, финализаторы могут применяться для освобождения неуправляемых ресурсов при запуске сборщика мусора. Однако многие неуправляемые объекты являются дорогостоящими ресурсами (например, низкоуровневые подключения к базам данных или файловые дескрипторы) и часто выгоднее освобождать их как можно раньше, еще до наступления момента сборки мусора. В качестве альтернативы переопределению `Finalize()` класс мог бы реализовать интерфейс `IDisposable`, который определяет единственный метод по имени `Dispose()`:

```
public interface IDisposable
{
    void Dispose();
}
```

При реализации интерфейса `IDisposable` предполагается, что после завершения работы с объектом метод `Dispose()` должен вручную вызываться *пользователем этого объекта*, прежде чем объектной ссылке будет разрешено покинуть область действия. Благодаря этому объект может производить любую необходимую очистку неуправляемых ресурсов без попадания в очередь финализации и без ожидания того, когда сборщик мусора запустит содержащуюся в классе логику финализации.

На заметку! Интерфейс `IDisposable` может быть реализован как в классах, так и в структурах (в отличие от метода `Finalize()`, который допускается переопределять только в классах), потому что метод `Dispose()` вызывается пользователем объекта (а не сборщиком мусора).

Рассмотрим пример использования этого интерфейса. Создадим новое консольное приложение C# по имени `SimpleDispose` и добавим в него следующую модифицированную версию класса `MyResourceWrapper`, которая вместо переопределения метода `System.Object.Finalize()` теперь реализует интерфейс `IDisposable`:

```
// Реализация интерфейса IDisposable.
public class MyResourceWrapper : IDisposable
{
    // После окончания работы с объектом пользователь
    // объекта должен вызывать этот метод.
    public void Dispose()
    {
        // Очистить неуправляемые ресурсы...

        // Освободить другие содержащиеся внутри освобождаемые объекты.

        // Только для целей тестирования.
        Console.WriteLine("***** In Dispose! *****");
    }
}
```

Обратите внимание, что метод `Dispose()` отвечает не только за освобождение неуправляемых ресурсов типа, но может также и вызывать `Dispose()` на любых других содержащихся в нем освобождаемых объектах. В отличие от `Finalize()`, в нем вполне безопасно взаимодействовать с другими управляемыми объектами. Причина проста: сборщик мусора не имеет понятия об интерфейсе `IDisposable`, и потому никогда не будет вызывать метод `Dispose()`. Следовательно, когда пользователь объекта вызывает данный метод, объект все еще существует в управляемой куче и имеет доступ ко всем остальным находящимся там объектам. Логика вызова этого метода выглядит довольно просто:

```

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with Dispose *****\n");
        // Создать освобождаемый объект и вызвать метод Dispose()
        // для освобождения любых внутренних ресурсов.
        MyResourceWrapper rw = new MyResourceWrapper();
        rw.Dispose();
        Console.ReadLine();
    }
}

```

Конечно, перед попыткой вызова метода `Dispose()` на объекте нужно проверить, поддерживает ли соответствующий тип интерфейс `IDisposable`. И хотя в документации .NET Framework 4.5 SDK всегда доступна информация о том, какие типы в библиотеке базовых классов реализуют `IDisposable`, такую проверку удобнее выполнять программно с применением ключевого слова `is` или `as` (см. главу 6):

```

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with Dispose *****\n");
        MyResourceWrapper rw = new MyResourceWrapper();
        if (rw is IDisposable)
            rw.Dispose();
        Console.ReadLine();
    }
}

```

Этот пример раскрывает еще одно правило, касающееся управления памятью.

На заметку! Имеет смысл вызывать метод `Dispose()` на любом создаваемом вами напрямую объекте, если он поддерживает интерфейс `IDisposable`. Следует исходить из того, что в случае, если разработчик класса решил реализовать метод `Dispose()`, значит, класс должен выполнять какую-то очистку. Если вы забудете это сделать, то память в конечном итоге будет очищена (так что можно не переживать), но это займет больше времени, чем необходимо.

С приведенным выше правилом связано одно предостережение. Ряд типов в библиотеках базовых классов, которые реализуют интерфейс `IDisposable`, предоставляют (несколько сбивающий с толку) псевдоним для метода `Dispose()`, чтобы отвечающий за очистку метод выглядел более естественным для определяющего его типа. В качестве примера можно взять класс `System.IO.FileStream`, в котором реализуется интерфейс `IDisposable` (и, следовательно, поддерживается метод `Dispose()`), но при этом также определяется и метод `Close()`, предназначенный для той же цели:

```

// Предполагается, что было импортировано пространство имен System.IO.
static void DisposeFileStream()
{
    FileStream fs = new FileStream("myFile.txt", FileMode.OpenOrCreate);
    // Как минимум, сбивает с толку!
    // Вызовы этих методов делают одно и то же!
    fs.Close();
    fs.Dispose();
}

```

И хотя “закрытие” (close) файла действительно звучит более естественно, чем его “освобождение” (dispose), нельзя не согласиться с тем, что подобное дублирование методов очистки вносит путаницу. Поэтому при использовании этих нескольких типов, в которых предоставляются псевдонимы, просто помните о том, что если тип реализует интерфейс `IDisposable`, то вызов метода `Dispose()` всегда является безопасным способом действия.

Повторное использование ключевого слова `using` в C#

При работе с управляемым объектом, который реализует интерфейс `IDisposable`, довольно часто требуется применять структурированную обработку исключений, гарантируя, что метод `Dispose()` типа будет вызываться даже в случае возникновения какого-то исключения:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Dispose *****\n");
    MyResourceWrapper rw = new MyResourceWrapper();
    try
    {
        // Использовать члены rw.
    }
    finally
    {
        // Всегда вызывать Dispose(), возникла ошибка или нет.
        rw.Dispose();
    }
}
```

Хотя это является замечательным примером безопасного программирования, реальность заключается в том, что лишь немногих разработчиков прельщает перспектива заключать каждый освобождаемый тип в блок `try/finally` только для того, чтобы обеспечить вызов `Dispose()`. Для достижения аналогичного результата, но гораздо менее громоздким образом, в C# поддерживается специальный фрагмент синтаксиса, который выглядит так:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Dispose *****\n");
    // Метод Dispose() вызывается автоматически
    // при выходе за пределы области действия using.
    using(MyResourceWrapper rw = new MyResourceWrapper())
    {
        // Использовать объект rw.
    }
}
```

Если теперь просмотреть CIL-код этого метода `Main()` с помощью `ildasm.exe`, то обнаружится, что синтаксис `using` в таких случаях расширяется до логики `try/finally`, которая включает в себя и вполне ожидаемый вызов `Dispose()`:

```
.method private hidebysig static void Main(string[] args) cil managed
{
    ...
    .try
    {
        ...
    } // end .try
```

```

finally
{
...
IL_0012: callvirt instance void
    SimpleFinalize.MyResourceWrapper::Dispose()
} // end handler
...
} // end of method Program::Main

```

На заметку! При попытке применить `using` к объекту, который не реализует интерфейс `IDisposable`, возникнет ошибка на этапе компиляции.

Хотя этот синтаксис избавляет от необходимости вручную помещать освобождаемые объекты внутрь блоков `try/finally`, к сожалению, теперь ключевое слово `using` в C# имеет двойное предназначение (импорт пространств имен и вызов метода `Dispose()`). Тем не менее, при работе с типами .NET, которые поддерживают интерфейс `IDisposable`, данная синтаксическая конструкция будет гарантировать автоматический вызов метода `Dispose()` на соответствующем объекте при выходе из блока `using`.

Кроме того, в контексте `using` допускается объявлять несколько объектов *одного и того же* типа. Как не трудно догадаться, в таком случае компилятор будет вставлять код с вызовом `Dispose()` для каждого объявляемого объекта.

```

static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Dispose *****\n");
    // Использование разделенного запятыми списка для
    // объявления нескольких подлежащих освобождению объектов.
    using(MyResourceWrapper rw = new MyResourceWrapper(),
        rw2 = new MyResourceWrapper())
    {
        // Использовать объекты rw и rw2.
    }
}

```

Исходный код. Проект `SimpleDispose` доступен в подкаталоге `Chapter 13`.

Создание финализируемых и освобождаемых типов

К этому моменту были рассмотрены два разных подхода к конструированию класса, который очищает внутренние неуправляемые ресурсы. С одной стороны, можно использовать финализатор. Такой подход позволяет гарантировать то, что объект будет очищать себя сам во время сборки мусора (когда бы это ни случилось) без вмешательства со стороны пользователя. С другой стороны, можно реализовать интерфейс `IDisposable`, чтобы предоставить пользователю объекта способ очистки объекта по окончании работы с ним. Однако если пользователь забудет вызвать метод `Dispose()`, неуправляемые ресурсы могут остаться в памяти на неопределенный срок.

Как не трудно догадаться, оба подхода можно комбинировать и применять вместе в определении одного класса, получая лучшее от обеих моделей. Если пользователь объекта не забыл вызвать метод `Dispose()`, можно проинформировать сборщик мусора о пропуске финализации, вызвав метод `GC.SuppressFinalize()`. Если же пользователь объекта забыл вызвать `Dispose()`, объект в конечном итоге будет финализирован и

получит шанс освободить внутренние ресурсы. Преимущество заключается в том, что внутренние неуправляемые ресурсы будут освобождены одним или другим способом.

Ниже приведена очередная версия класса `MyResourceWrapper`, которая теперь является финализируемой и освобождаемой; она определена в консольном приложении C# по имени `FinalizableDisposableClass`:

```
// Сложная оболочка для ресурсов.
public class MyResourceWrapper : IDisposable
{
    // Сборщик мусора будет вызывать этот метод, если
    // пользователь объекта забыл вызвать Dispose().
    ~MyResourceWrapper()
    {
        // Очистить любые внутренние неуправляемые ресурсы.
        // **Не** вызывать Dispose() на управляемых объектах.
    }

    // Пользователь объекта будет вызывать этот метод
    // для очистки ресурсов как можно быстрее.
    public void Dispose()
    {
        // Очистить неуправляемые ресурсы.
        // Вызвать Dispose() для других содержащихся внутри освобождаемых объектов.

        // Если пользователь вызвал Dispose(), то финализация
        // не нужна, поэтому подавить ее.
        GC.SuppressFinalize(this);
    }
}
```

Обратите внимание, что этот метод `Dispose()` был модифицирован для вызова метода `GC.SuppressFinalize()`, который информирует среду CLR о том, что вызывать деструктор при обработке данного объекта сборщиком мусора больше не обязательно, т.к. неуправляемые ресурсы уже освобождены посредством логики `Dispose()`.

Формализованный шаблон освобождения

Текущая реализация `MyResourceWrapper` работает довольно хорошо, но все равно еще осталось несколько небольших недостатков. Во-первых, методы `Finalize()` и `Dispose()` должны освобождать одни и те же неуправляемые ресурсы, а это чревато дублированием кода, что может существенно усложнить сопровождение. В идеале следовало бы определить закрытую вспомогательную функцию и вызывать ее в этих методах.

Во-вторых, желательно удостовериться в том, что метод `Finalize()` не пытается освободить любые управляемые объекты, в то время как метод `Dispose()` должен делать это. Наконец, в-третьих, имеет смысл также позаботиться о том, чтобы пользователь объекта мог спокойно вызывать метод `Dispose()` много раз без получения ошибки. В настоящий момент в методе `Dispose()` никакой защиты подобного рода не предусмотрено.

Для решения таких проектных задач в Microsoft создали формальный шаблон освобождения, который позволяет достичь баланса между надежностью, удобством в сопровождении и производительностью. Ниже представлена окончательная версия `MyResourceWrapper`, в которой применяется этот официальный шаблон.

```
class MyResourceWrapper : IDisposable
{
    // Используется для выяснения того,
    // вызывался ли уже метод Dispose().
```

```

private bool disposed = false;

public void Dispose()
{
    // Вызвать вспомогательный метод.
    // Указание true означает, что очистку
    // запустил пользователь объекта.
    Cleanup(true);

    // Подавить финализацию.
    GC.SuppressFinalize(this);
}

private void Cleanup(bool disposing)
{
    // Удостовериться, не выполнялось ли уже освобождение.
    if (!this.disposed)
    {
        // Если disposing равно true, освободить
        // все управляемые ресурсы.
        if (disposing)
        {
            // Освободить управляемые ресурсы.
        }
        // Очистить неуправляемые ресурсы.
    }
    disposed = true;
}

~MyResourceWrapper()
{
    // Вызвать вспомогательный метод.
    // Указание false означает, что
    // очистку запустил сборщик мусора.
    Cleanup(false);
}
}

```

Обратите внимание, что в `MyResourceWrapper` теперь определяется закрытый вспомогательный метод по имени `Cleanup()`. Передавая ему в качестве аргумента значение `true`, мы указываем, что очистку инициировал пользователь объекта и, следовательно, требуется освободить все управляемые и неуправляемые ресурсы. Однако когда очистка инициируется сборщиком мусора, при вызове метода `Cleanup()` передается значение `false`, чтобы внутренние освобождаемые объекты не освобождались (поскольку нельзя рассчитывать на то, что они по-прежнему находятся в памяти). И, наконец, перед выходом из `Cleanup()` переменная `disposed` типа `bool` устанавливается в `true`, что дает возможность вызывать метод `Dispose()` много раз без возникновения ошибки.

На заметку! После того как объект “освобожден”, клиент по-прежнему может обращаться к его членам, т.к. объект пока еще находится в памяти. Следовательно, в надежном классе-оболочке для ресурсов также необходимо снабдить каждый член дополнительной логикой, которая бы сообщала: “если объект освобожден, ничего не делать, а просто вернуть управление”.

Чтобы протестировать последнюю версию класса `MyResourceWrapper`, добавим в финализатор вызов `Console.Beep()`:

```

~MyResourceWrapper()
{
    Console.Beep();
    // Вызвать вспомогательный метод.
}

```

```
// Указание false означает, что очистку запустил сборщик мусора.
CleanUp(false);
}
```

Далее обновим метод Main() следующим образом:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Dispose() / Destructor Combo Platter *****");
    // Вызвать метод Dispose() вручную. Это не приведет к вызову финализатора.
    MyResourceWrapper rw = new MyResourceWrapper();
    rw.Dispose();

    // Не вызывать метод Dispose(). Это приведет к вызову финализатора
    // и выдаче звукового сигнала.
    MyResourceWrapper rw2 = new MyResourceWrapper();
}
```

Обратите внимание, что мы явно вызываем метод Dispose() на объекте rw, поэтому вызов деструктора подавляется. Однако мы “забываем” вызвать метод Dispose() на объекте rw2 и потому по окончании выполнения приложения услышим звуковой сигнал. Если закомментировать вызов Dispose() на объекте rw, звуковых сигналов будет два.

Исходный код. Проект FinalizableDisposableClass доступен в подкаталоге Chapter 13.

На этом исследование особенностей управления объектами со стороны среды CLR через сборку мусора завершено. Хотя здесь не рассматривались дополнительные (довольно экзотические) детали, касающиеся процесса сборки мусора (такие как слабые ссылки и восстановление объектов), полученных сведений должно быть достаточно, чтобы продолжить изучение самостоятельно. В завершение главы мы исследуем программное средство под названием *ленивое (отложенное) создание объектов*.

Ленивое создание объектов

При создании классов иногда требуется предусмотреть в коде определенную переменную-член, которая на самом деле может никогда не понадобиться из-за того, что пользователь объекта не будет обращаться к методу (или свойству), в котором она используется. Это довольно распространенная ситуация, хотя на практике ее реализация может оказаться весьма проблематичной в случае, если создание такой переменной-члена требует большого объема памяти.

Для примера предположим, что строится класс, который инкапсулирует операции цифрового музыкального проигрывателя. Помимо ожидаемых методов вроде Play(), Pause() и Stop() нужно также обеспечить возможность возврата коллекции объектов Song (через класс по имени AllTracks), которая представляет все имеющиеся в устройстве цифровые музыкальные файлы.

Чтобы получить такой класс, создадим новое консольное приложение по имени LazyObjectInstantiation и определим в нем следующие типы классов:

```
// Представляет одну композицию.
class Song
{
    public string Artist { get; set; }
    public string TrackName { get; set; }
    public double TrackLength { get; set; }
}
```

```
// Представляет все композиции в проигрывателе.
class AllTracks
{
    // Наш проигрыватель может содержать
    // максимум 10 000 композиций.
    private Song[] allSongs = new Song[10000];

    public AllTracks()
    {
        // Предположим, что здесь производится заполнение
        // массива объектов Song.
        Console.WriteLine("Filling up the songs!");
    }
}

// Класс MediaPlayer включает объект AllTracks.
class MediaPlayer
{
    // Предположим, что эти методы делают что-то полезное.
    public void Play() { /* Воспроизведение композиции */ }
    public void Pause() { /* Пауза в воспроизведении */ }
    public void Stop() { /* Останов воспроизведения */ }

    private AllTracks allSongs = new AllTracks();

    public AllTracks GetAllTracks()
    {
        // Вернуть все композиции.
        return allSongs;
    }
}
```

В текущей реализации MediaPlayer делается предположение о том, что пользователю объекта понадобится получать список объектов с помощью метода GetAllTracks(). А что если пользователю объекта не нужен этот список? В приведенной реализации память под переменную-член AllTracks все равно будет выделена, таким образом, приводя к созданию 10 000 объектов Song в памяти:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Lazy Instantiation *****\n");

    // В этом вызывающем коде получение всех композиций
    // не производится, но косвенно все равно создаются
    // 10 000 объектов!
    MediaPlayer myPlayer = new MediaPlayer();
    myPlayer.Play();

    Console.ReadLine();
}
```

Очевидно, что лучше избежать создания 10 000 объектов, которыми никто не будет пользоваться, т.к. это изрядно прибавит работы сборщику мусора .NET. Хотя можно вручную добавить код, который обеспечит создание объекта allSongs только в случае, если он используется (например, за счет применения шаблона фабричного метода), существует и более простой путь.

В библиотеках базовых классов предоставляется очень полезный обобщенный класс по имени Lazy<>, который определен в пространстве имен System внутри сборки mscorlib.dll. Этот класс позволяет определять данные, которые не будут создаваться до тех пор, пока они в действительности не начнут использоваться в коде. Поскольку класс является обобщенным, при первом его применении должен быть явно указан тип

элемента, который будет создаваться. Этим типом может быть любой из типов, определенных в библиотеках базовых классов .NET, либо специальный тип, построенный разработчиком самостоятельно. Для включения отложенной инициализации переменной-члена AllTracks можно просто заменить следующий фрагмент кода:

```
// Класс MediaPlayer включает объект AllTracks.
class MediaPlayer
{
    ...
    private AllTracks allSongs = new AllTracks();
    public AllTracks GetAllTracks()
    {
        // Возвратить все композиции.
        return allSongs;
    }
}
```

таким кодом:

```
// Класс MediaPlayer включает объект Lazy<AllTracks>.
class MediaPlayer
{
    ...
    private Lazy<AllTracks> allSongs = new Lazy<AllTracks>();
    public AllTracks GetAllTracks()
    {
        // Возвратить все композиции.
        return allSongs.Value;
    }
}
```

Помимо того факта, что переменная-член AllTrack теперь имеет тип Lazy<>, важно отметить, что реализация предыдущего метода GetAllTracks() тоже изменилась. В частности, для получения фактических хранимых данных (в этом случае — объекта AllTracks, поддерживающего 10 000 объектов Song) мы должны использовать доступное только для чтения свойство Value класса Lazy<>.

Взгляните, как благодаря этому простому изменению показанный ниже модифицированный метод Main() будет косвенно размещать объекты Song в памяти только в случае, если метод GetAllTracks() действительно вызывался:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Lazy Instantiation *****\n");
    // Никакого размещения объекта AllTracks здесь не происходит!
    MediaPlayer myPlayer = new MediaPlayer();
    myPlayer.Play();

    // Размещение объекта AllTracks происходит
    // только в случае вызова метода GetAllTracks().
    MediaPlayer yourPlayer = new MediaPlayer();
    AllTracks yourMusic = yourPlayer.GetAllTracks();

    Console.ReadLine();
}
```

На заметку! Ленивое создание объектов полезно не только для уменьшения количества случаев выделения памяти под ненужные объекты. Этот прием можно также использовать в ситуации, когда для создания члена применяется дорогостоящий в смысле ресурсов код, такой как вызов удаленного метода, взаимодействие с реляционной базой данных и т.п.

Настройка процесса создания данных Lazy<>

При объявлении переменной Lazy<> фактический внутренний тип данных создается с помощью стандартного конструктора:

```
// При использовании переменной Lazy<> вызывается
// стандартный конструктор для AllTracks.
private Lazy<AllTracks> allSongs = new Lazy<AllTracks>();
```

В некоторых случаях подобное поведение может быть вполне подходящим, но что если класс AllTracks имеет дополнительные конструкторы, и нужно обеспечить вызов подходящего из них? Более того, что если при создании переменной Lazy() необходимо выполнить какую-то дополнительную работу (помимо просто создания объекта AllTracks)? К счастью, класс Lazy() позволяет указывать в качестве необязательного параметра обобщенный делегат, который задает метод для вызова во время создания находящегося внутри него типа.

Этим обобщенным делегатом является System.Func<>, который может указывать на метод, возвращающий тот же тип данных, что создается соответствующей переменной Lazy<>, и способный принимать вплоть до 16 аргументов (которые типизируются с помощью обобщенных параметров типа). В большинстве случаев никаких параметров для передачи методу, на который указывает Func<>, задавать не понадобится. Вдобавок, чтобы значительно упростить применение Func<>, лучше использовать лямбда-выражения (отношения между делегатами и лямбда-выражениями подробно рассматривались в главе 10).

С учетом всего сказанного, ниже приведена окончательная версия MediaPlayer, в которой теперь при создании внутреннего объекта AllTracks добавляется небольшой специальный код. Не забывайте, что перед выходом этот метод должен возвращать новый экземпляр указанного в Lazy<> типа, и применять можно любой конструктор (в коде для AllTracks по-прежнему вызывается стандартный конструктор).

```
class MediaPlayer
{
    ...
    // Использовать лямбда-выражение для добавления
    // дополнительного кода при создании объекта AllTracks.
    private Lazy<AllTracks> allSongs = new Lazy<AllTracks>(() =>
    {
        Console.WriteLine("Creating AllTracks object!");
        return new AllTracks();
    });
    public AllTracks GetAllTracks()
    {
        // Возвратить все композиции.
        return allSongs.Value;
    }
}
```

Надо надеяться, что вы смогли оценить удобство применения класса Lazy<>. В сущности, этот обобщенный класс позволяет делать так, чтобы дорогостоящие в плане ресурсов объекты размещались в памяти только тогда, когда они будут действительно нужны пользователю объекта. Если эта тема вас заинтересовала, загляните в раздел документации .NET Framework 4.5 SDK, посвященный классу System.Lazy<>, и найдите там дополнительные примеры реализации ленивого создания.

Резюме

Цель настоящей главы заключалась в снятии завесы с процесса сборки мусора. Было показано, что сборщик мусора запускается только тогда, когда не удастся получить необходимый объем памяти из управляемой кучи (или когда происходит выгрузка из памяти соответствующего домена приложения). После сборки мусора поводов для волнения возникать не должно, поскольку разработанный в Microsoft алгоритм сборки мусора хорошо оптимизирован и предусматривает использование поколений объектов, дополнительных потоков для финализации объектов и управляемой кучи для обслуживания больших объектов.

В главе также было показано, как программно взаимодействовать со сборщиком мусора с применением класса `System.GC`. Как отмечалось, единственным случаем, когда в этом может возникнуть необходимость, является построение финализируемых или освобождаемых классов, которые оперируют неуправляемыми ресурсами.

Вспомните, что финализируемые типы — это классы, которые предоставляют деструктор (переопределяя метод `Finalize()`) для очистки неуправляемых ресурсов во время сборки мусора. С другой стороны, освобождаемые объекты являются классами (или структурами), которые реализуют интерфейс `IDisposable`, вызываемый пользователем объекта по окончании работы с ним. Кроме того, в главе был продемонстрирован официальный шаблон освобождения, позволяющий совмещать оба эти подхода.

И, наконец, в главе был описан обобщенный класс по имени `Lazy<>`. Как было показано, данный класс позволяет отложить создание дорогостоящих (в плане потребления памяти) объектов до тех пор, пока вызывающая сторона действительно не затребует их. Это помогает сократить количество объектов, сохраняемых в управляемой куче, и обеспечивает создание дорогостоящих объектов только тогда, когда они на самом деле требуются в вызывающем коде.

Программирование с использованием сборок .NET

В этой части

Глава 14. Построение и конфигурирование библиотек классов

Глава 15. Рефлексия типов, позднее связывание и программирование
с использованием атрибутов

Глава 16. Динамические типы и среда DLR

Глава 17. Процессы, домены приложений и объектные контексты

Глава 18. Язык CIL и роль динамических сборок

Построение и конфигурирование библиотек классов

На протяжении первых четырех частей этой книги было создано несколько “автономных” исполняемых приложений, вся программная логика которых упаковывалась в единственный исполняемый файл (*.exe). Эти исполняемые сборки пользовались в основном главной библиотекой классов .NET под названием mscorlib.dll. Хотя некоторые простые программы .NET могут быть сконструированы с применением только библиотек базовых классов .NET, часто многократно используемая программная логика изолируется в *специальные библиотеки классов* (файлы *.dll), разделяемые между множеством приложений.

В этой главе вы ознакомитесь с различными способами упаковки типов в специальные библиотеки кода. Вначале вы узнаете о разделении типов по пространствам имен .NET. После этого вы исследуете шаблоны проектов библиотеки классов Visual Studio и разберетесь в отличиях между закрытыми и разделяемыми сборками.

Затем будет показано, как исполняющая среда определяет местонахождение сборки, и рассказано о том, что собой представляют глобальный кеш сборок (Global Assembly Cache — GAC), XML-файлы конфигурации приложений (файлы *.config), сборки политик издателя и пространство имен System.Configuration.

Определение специальных пространств имен

Прежде чем углубляться в детали развертывания и конфигурирования библиотек, сначала необходимо узнать о том, как упаковывать свои специальные типы в пространства имен .NET. Вплоть до этого места в книге разрабатывались лишь небольшие тестовые программы, в которых использовались существующие пространства имен .NET (в частности, System). Однако когда строится крупное приложение со многими типами, может быть удобно сгруппировать взаимосвязанные типы в специальные пространства имен. В C# это делается с помощью ключевого слова namespace. Явное определение специальных пространств имен становится еще более важным при построении .NET-сборок *.dll, т.к. другие разработчики получают возможность ссылаться на эти библиотеки и импортировать специальные пространства имен для работы с содержащимися в них типами.

Чтобы увидеть все это в действии, создадим новый проект консольного приложения по имени CustomNamespaces. Предположим, что требуется разработать коллекцию

геометрических классов Square (квадрат), Circle (круг) и Hexagon (шестиугольник). Учитывая схожие между ними черты, было бы желательно сгруппировать их вместе в уникальном пространстве имен MyShapes внутри сборки CustomNamespaces.exe. На выбор доступны два основных подхода. Первый подход предусматривает определение всех классов в единственном файле C# (ShapesLib.cs), как показано ниже:

```
// Файл Shapeslib.cs.
using System;

namespace MyShapes
{
    // Класс Circle.
    public class Circle{ /* Интересные члены... */ }

    // Класс Hexagon.
    public class Hexagon{ /* Более интересные члены... */ }

    // Класс Square.
    public class Square{ /* Даже еще более интересные члены... */ }
}
```

Хотя компилятор C# без проблем воспримет единственный файл кода C#, содержащий множество типов, могут возникнуть сложности, когда понадобится повторно использовать определения классов в новых проектах. Например, пусть разрабатывается проект, в котором необходимо взаимодействовать только с классом Circle. Если все типы определены в единственном файле кода, вы так или иначе привязаны к целому набору типов. Следовательно, в качестве альтернативы вы можете разделить одно пространство имен на несколько файлов кода C#. Чтобы обеспечить размещение типов в одной и той же логической группе, нужно просто упаковать определения всех данных классов в контекст одного и того же пространства имен, как показано ниже:

```
// Файл Circle.cs.
using System;

namespace MyShapes
{
    // Класс Circle.
    public class Circle { /* Интересные методы... */ }
}

// Файл Hexagon.cs.
using System;

namespace MyShapes
{
    // Класс Hexagon.
    public class Hexagon { /* Более интересные методы... */ }
}

// Файл Square.cs.
using System;

namespace MyShapes
{
    // Класс Square.
    public class Square { /* Даже еще более интересные методы... */ }
}
```

В обоих случаях обратите внимание, что пространство MyShapes действует в качестве концептуального “контейнера” для этих классов. Если другое пространство имен (такое как CustomNamespaces) желает пользоваться типами из отдельного пространства имен, нужно применить ключевое слово using, как это делается при работе с пространствами имен библиотек базовых классов .NET:

```
// Использовать пространство имен из библиотек базовых классов.
using System;

// Использовать типы, определенные в пространстве имен MyShapes.
using MyShapes;

namespace CustomNamespaces
{
    public class Program
    {
        static void Main(string[] args)
        {
            Hexagon h = new Hexagon();
            Circle c = new Circle();
            Square s = new Square();
        }
    }
}
```

В этом примере предполагается, что файл (или файлы) C#, где определено пространство имен `MyShapes`, является частью того же проекта консольного приложения, что и файл с определением пространства имен `CustomNamespaces`. Другими словами, все эти файлы используются для компиляции единственной исполняемой сборки .NET. Если пространство имен `MyShapes` определено внутри внешней сборки, для успешной компиляции потребуется также добавить ссылку на эту библиотеку. Вы изучите все детали построения приложений, взаимодействующих с внешними библиотеками, на протяжении настоящей главы.

Устранение конфликтов между именами посредством полностью заданных имен

Формально применение ключевого слова `using` языка C# при ссылках на типы, определенные во внешних пространствах имен, не является обязательным. Вместо этого можно использовать *полностью заданные имена* типов, которые, как упоминалось в главе 1, представляют собой имена типов, предваренные названиями пространств имен, где эти типы определены. Например:

```
// Обратите внимание, что MyShapes больше не импортируется!
using System;
namespace CustomNamespaces
{
    public class Program
    {
        static void Main(string[] args)
        {
            MyShapes.Hexagon h = new MyShapes.Hexagon();
            MyShapes.Circle c = new MyShapes.Circle();
            MyShapes.Square s = new MyShapes.Square();
        }
    }
}
```

Обычно необходимость в указании полностью заданного имени не возникает. Это требует большего объема клавиатурного ввода, никак не влияя на размер и скорость выполнения кода. С другой стороны, в CIL-коде типы *всегда* определяются с использованием полностью заданных имен. Таким образом, ключевое слово `using` в C# является просто средством экономии времени.

Тем не менее, полностью заданные имена могут быть очень полезными (а иногда необходимыми) для устранения потенциальных конфликтов имен, которые могут возникнуть при использовании множества пространств имен, содержащих одинаково названные типы. Предположим, что имеется новое пространство имен `My3DShapes`, в котором определены следующие три класса, способные визуализировать фигуры в трехмерном формате:

```
// Еще одно пространство имен для работы с фигурами.
using System;

namespace My3DShapes
{
    // Класс трехмерного круга.
    public class Circle { }

    // Класс трехмерного многоугольника.
    public class Hexagon { }

    // Класс трехмерного квадрата.
    public class Square { }
}
```

Если теперь модифицировать класс `Program`, как показано ниже, компилятор выдаст множество сообщений об ошибках, поскольку оба пространства имен определяют идентично именованные классы:

```
// Обилие неоднозначностей!
using System;
using MyShapes;
using My3DShapes;

namespace CustomNamespaces
{
    public class Program
    {
        static void Main(string[] args)
        {
            // На какое пространство имен производится ссылка?
            Hexagon h = new Hexagon();           // Ошибка на этапе компиляции!
            Circle c = new Circle();              // Ошибка на этапе компиляции!
            Square s = new Square();              // Ошибка на этапе компиляции!
        }
    }
}
```

Использование полностью заданных имен позволяет устранить все эти неоднозначности:

```
// Теперь неоднозначности устранены.
static void Main(string[] args)
{
    My3DShapes.Hexagon h = new My3DShapes.Hexagon();
    My3DShapes.Circle c = new My3DShapes.Circle();
    MyShapes.Square s = new MyShapes.Square();
}
```

Устранение конфликтов между именами посредством псевдонимов

Ключевое слово `using` языка C# также позволяет создавать псевдоним для полностью уточненного имени типа. В этом случае определяется лексема, которая во время компиляции заменяется полностью заданным именем типа. Определение псевдонимов является вторым способом разрешения конфликтов имен.

Например:

```
using System;
using MyShapes;
using My3DShapes;

// Устранить неоднозначность, используя специальный псевдоним.
using The3DHexagon = My3DShapes.Hexagon;

namespace CustomNamespaces
{
    class Program
    {
        static void Main(string[] args)
        {
            // Это на самом деле создает экземпляр класса My3DShapes.Hexagon.
            The3DHexagon h2 = new The3DHexagon();
            ...
        }
    }
}
```

Этот альтернативный синтаксис `using` позволяет также создавать псевдонимы для пространств имен, имеющих длинные названия. Примером такого пространства имен в библиотеке базовых классов является пространство `System.Runtime.Serialization.Formatters.Binary`, содержащее член по имени `BinaryFormatter`. При желании экземпляр `BinaryFormatter` можно создать следующим образом:

```
using bfHome = System.Runtime.Serialization.Formatters.Binary;

namespace MyApp
{
    class ShapeTester
    {
        static void Main(string[] args)
        {
            bfHome.BinaryFormatter b = new bfHome.BinaryFormatter();
            ...
        }
    }
}
```

А вот как это сделать с применением обычной директивы `using`:

```
using System.Runtime.Serialization.Formatters.Binary;

namespace MyApp
{
    class ShapeTester
    {
        static void Main(string[] args)
        {
            BinaryFormatter b = new BinaryFormatter();
            ...
        }
    }
}
```

На этом этапе беспокоиться о том, для чего служит класс `BinaryFormatter`, не нужно (он подробно рассматривается в главе 20). Сейчас важно уяснить, что ключевое слово `using` в C# позволяет создавать псевдонимы для длинных полностью заданных имен и потому может применяться для разрешения конфликтов имен, которые могут возникать в результате импорта пространств имен, определяющих идентично названные типы.

На заметку! Имейте в виду, что чрезмерное использование псевдонимов C# может привести к получению запутанной кодовой базы. Если другие программисты в команде не знают об этих специальных псевдонимах, они могут предположить, что они ссылаются на типы в библиотеках базовых классов .NET, и прийти в замешательство, не найдя их описаний в документации .NET Framework 4.5 SDK.

Создание вложенных пространств имен

При организации типов допускается определять пространства имен внутри других пространств имен. В библиотеках базовых классов .NET подобное встречается во многих местах и обеспечивает размещение типов на более глубоких уровнях. Например, пространство имен `IO` находится внутри пространства имен `System` и дает в результате `System.IO`. Чтобы создать корневое пространство имен, содержащее внутри себя существующее пространство имен `My3DShapes`, необходимо модифицировать код следующим образом:

```
// Создание вложенного пространства имен.
namespace Chapter14
{
    namespace My3DShapes
    {
        // Класс трехмерного круга.
        public class Circle{ }

        // Класс трехмерного шестиугольника.
        public class Hexagon{ }

        // Класс трехмерного квадрата.
        public class Square{ }
    }
}
```

Во многих случаях роль корневого пространства имен заключается просто в предоставлении дополнительного уровня контекста, поэтому в контексте его самого могут отсутствовать определения каких-либо типов (как в случае пространства имен `Chapter14`). Когда дело обстоит именно так, вложенное пространство имен может быть определено с применением следующей компактной формы:

```
// Создание вложенного пространства имен (другой способ).
namespace Chapter14.My3DShapes
{
    // Класс трехмерного круга.
    public class Circle{ }

    // Класс трехмерного шестиугольника.
    public class Hexagon{ }

    // Класс трехмерного квадрата.
    public class Square{ }
}
```

Из-за того, что теперь пространство `My3DShapes` вложено внутрь корневого пространства имен `Chapter14`, необходимо соответствующим образом обновить все существующие директивы `using` и псевдонимы типов:

```
using Chapter14.My3DShapes;
using The3DHexagon = Chapter14.My3DShapes.Hexagon;
```

Стандартное пространство имен Visual Studio

В завершение темы пространств имен полезно отметить, что по умолчанию при создании нового проекта на C# в Visual Studio стандартное пространство имен приложения будет иметь название, идентичное имени проекта. После этого при вставке в проект новых файлов кода с помощью пункта меню Project⇒Add New Item (Проект⇒Добавить новый элемент) типы будут автоматически помещаться внутрь этого стандартного пространства имен. Чтобы изменить название стандартного пространства имен, откройте окно свойств проекта, перейдите в нем на вкладку Application (Приложение) и введите желаемое имя в поле Default namespace (Стандартное пространство имен), как показано на рис. 14.1.

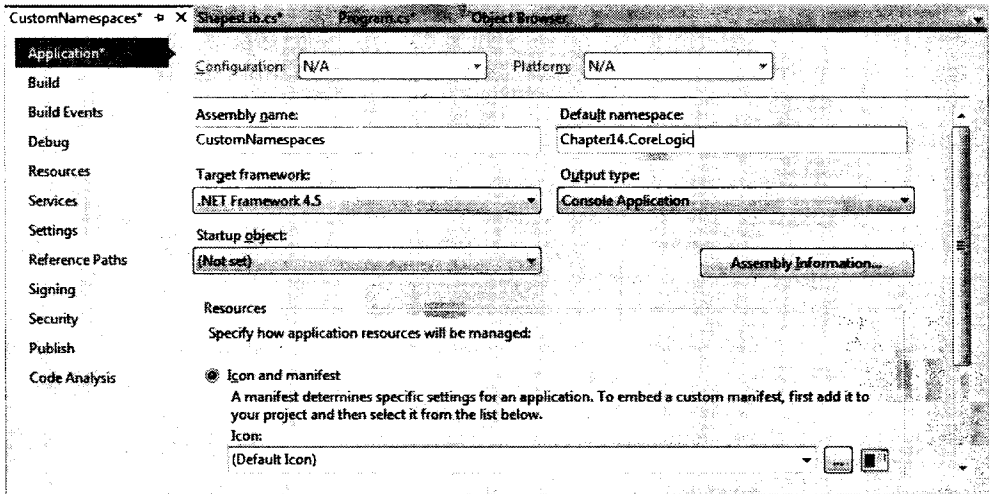


Рис. 14.1. Изменение названия стандартного пространства имен

После такого изменения любой новый элемент, вставляемый в проект, будет размещаться внутри пространства имен Chapter14.CoreLogic (разумеется, для использования его типов в другом пространстве имен понадобится применить соответствующую директиву using).

Теперь, когда вы ознакомились с некоторыми деталями упаковки специальных типов в четко организованные пространства имен, давайте кратко рассмотрим преимущества и формат сборок .NET. После этого мы углубимся в подробности создания, развертывания и конфигурирования специальных библиотек классов.

Исходный код. Проект CustomNamespaces доступен в подкаталоге Chapter 14.

Роль сборок .NET

Приложения .NET конструируются за счет соединения вместе любого количества сборок. Выражаясь просто, сборка представляет собой поддерживающий версии само-описательный двоичный файл, обслуживаемый средой CLR. Несмотря на то что сборки .NET имеют точно такие же файловые расширения (*.exe или *.dll), как и старые двоичные файлы Windows, внутри они устроены совсем по-другому. Поэтому прежде чем углубляться в изучение дальнейшего материала, давайте сначала посмотрим, какие преимущества предлагает формат сборок.

Сборки увеличивают возможности для повторного использования кода

При построении консольных приложений в предыдущих главах могло показаться, что вся функциональность этих приложений содержалась внутри создававшейся исполняемой сборки. На самом деле во всех этих приложениях использовались многочисленные типы из всегда доступной библиотеки кода .NET по имени `mscorlib.dll` (вспомните, что компилятор C# ссылается на `mscorlib.dll` автоматически), а в некоторых случаях также из библиотеки по имени `System.Core.dll`.

Как уже может быть известно, *библиотека кода* (также называемая *библиотекой классов*) представляет собой файл `*.dll`, который содержит типы, предназначенные для использования во внешних приложениях. При создании исполняемых сборок, несомненно, придется использовать много системных и специальных библиотек кода по мере разработки текущего приложения. Однако следует учесть, что библиотека кода не обязательно должна иметь файловое расширение `*.dll`. Вполне допускается (хотя и не часто), чтобы исполняемая сборка работала с типами, определенными внутри внешнего исполняемого файла. В этом случае ссылаемый файл `*.exe` также может считаться библиотекой кода.

Независимо от того, как упакована библиотека кода, платформа .NET позволяет многократно использовать типы в независимой от языка манере. Например, можно было бы создать библиотеку кода на C# и работать с этой библиотекой на другом языке программирования .NET. Допускается не только размещать экземпляры типов из разных языков, но и наследовать от них. Базовый класс, определенный в C#, может быть расширен классом, написанным на Visual Basic. Интерфейсы, определенные в F#, могут быть реализованы структурами, определенными в C#, и т.д. Важно понимать, что разбиение единого монолитного исполняемого файла на несколько сборок .NET открывает возможности для многократного использования кода в *нейтральной к языку* форме.

Сборки определяют границы типов

Вспомните, что *полностью заданное имя* типа получается за счет предварения имени этого типа (например, `Console`) названием пространства имен, где он определен (например, `System`). Строго говоря, сборка, в которой находится тип, в дальнейшем определяет его идентичность. Например, если есть две уникально именованных сборки (скажем, `MyCars.dll` и `YourCars.dll`), которые обе определяют пространство имен (`CarLibrary`), содержащее класс по имени `SportsCar`, то в мире .NET эти типы будут считаться уникальными.

Сборки являются единицами, поддерживающими версии

Сборкам .NET назначается состоящий из четырех частей числовой номер версии в форме *<старший номер>.<младший номер>.<номер сборки>.<номер редакции>*. (Если номер версии явно не указан, сборке автоматически назначается версия 1.0.0.0 из стандартных настроек проекта в Visual Studio.) Этот номер вместе с необязательным значением *открытого ключа* позволяет множеству версий той же самой сборки нормально сосуществовать на одной машине. Формально сборки, которые предоставляют информацию открытого ключа, называются *строго именованными*. Как будет показано далее, при наличии строгого имени среда CLR гарантирует загрузку корректной версии в интересах вызывающего клиента.

Сборки являются самоописательными

Сборки считаются *самоописательными* частично потому, что содержат информацию обо всех внешних сборках, к которым они должны иметь доступ для корректного функционирования. Таким образом, если сборке требуются `System.Windows.Forms.dll` и `System.Core.dll`, это будет документировано в *манифесте* сборки. Вспомните из главы 1, что манифестом называется блок метаданных, описывающий саму сборку (имя, версию, обязательные внешние сборки и т.д.).

В дополнение к данным манифеста сборка содержит метаданные, которые описывают структуру каждого из имеющихся в ней типов (имена членов, реализуемые интерфейсы, базовые классы, конструкторы и т.п.). Благодаря наличию в сборке такого детального описания, среде CLR не требуется обращение к системному реестру Windows для выяснения ее местонахождения (что радикально отличается от унаследованной модели программирования COM, предлагаемой Microsoft). Как вы узнаете в этой главе, для получения информации о местонахождении внешних библиотек кода среда CLR применяет совершенно новую схему.

Сборки являются конфигурируемыми

Сборки могут развертываться как “закрытые” или как “разделяемые”. Закрытые сборки размещаются в том же каталоге (или подкаталоге), что и клиентское приложение, в котором они используются. С другой стороны, разделяемые сборки — это библиотеки, предназначенные для применения во многих приложениях на одной и той же машине, и они развертываются в специальном каталоге, который называется *глобальным кешем сборок* (Global Assembly Cache — GAC).

Независимо от способа развертывания сборок, для них могут быть созданы конфигурационные XML-файлы. С их помощью можно указать CLR-среде, где конкретно искать сборки, какую версию той или иной сборки загружать для определенного клиента или в какой каталог на локальной машине, место в сети либо URL-адрес заглядывать. Конфигурационные XML-файлы рассматриваться далее в настоящей главе.

Формат сборки .NET

Теперь, когда вы ознакомились с некоторыми преимуществами сборок .NET, давайте более детально рассмотрим, как эти сборки устроены внутри. С точки зрения структуры сборки .NET (*.dll или *.exe) состоит из следующих элементов:

- заголовок файла Windows;
- заголовок файла CLR;
- CIL-код;
- метаданные типов;
- манифест сборки;
- дополнительные встроенные ресурсы.

Хотя первые два элемента (заголовки Windows и CLR) являются блоками данных, которые обычно можно игнорировать, краткого рассмотрения они все-таки заслуживают. Ниже представлен обзор всех перечисленных элементов.

Заголовок файла Windows

Заголовок файла Windows указывает на тот факт, что сборка может загружаться и использоваться в операционных системах семейства Windows. Кроме того, этот заголовок

идентифицирует тип приложения (консольное, с графическим пользовательским интерфейсом или библиотека кода *.dll). Чтобы просмотреть информацию, содержащуюся в заголовке Windows, необходимо открыть сборку .NET с помощью утилиты dumpbin.exe (через окно командной строки разработчика), указав ей флаг /headers:

```
dumpbin /headers CarLibrary.dll
```

Ниже показана (не полностью) информация заголовка Windows для сборки CarLibrary.dll, которая будет построена далее в этой главе (можете запустить утилиту dumpbin.exe, указав вместо CarLibrary имя любого из ранее созданных файлов *.dll или *.exe):

```
Dump of file CarLibrary.dll
```

```
PE signature found
```

```
File Type: DLL
```

FILE HEADER VALUES

```

    14C machine (x86)
      3 number of sections
4B37DCD8 time date stamp Sun Dec 27 16:16:56 2011
      0 file pointer to symbol table
      0 number of symbols
      E0 size of optional header
    2102 characteristics
          Executable
          32 bit word machine
          DLL

```

OPTIONAL HEADER VALUES

```

    10B magic # (PE32)
    8.00 linker version
    E00 size of code
    600 size of initialized data
      0 size of uninitialized data
    2CDE entry point (00402CDE)
    2000 base of code
    4000 base of data
    400000 image base (00400000 to 00407FFF)
    2000 section alignment
    200 file alignment
    4.00 operating system version
    0.00 image version
    4.00 subsystem version
      0 Win32 version
    8000 size of image
    200 size of headers
      0 checksum
      3 subsystem (Windows CUI)

```

```
...
```

```
Дамп файла CarLibrary.dll
```

```
Обнаружена подпись PE
```

```
Тип файла: DLL
```

```
Значения заголовка файла
```

```

    14C машина (x86)
      3 количество разделов
4B37DCD8 дата и время Sun Dec 27 16:16:56 2011
      0 файловый указатель на таблицу символов
      0 количество символов
      E0 размер необязательного заголовка

```

```

2102 характеристики
      Исполняемый файл
      Машина с 32-битным словом
      DLL

Необязательные значения заголовка
10В магический номер (PE32)
8.00 версия редактора связей
E00 размер кода
600 размер инициализированных данных
0 размер неинициализированных данных
2CDE точка входа (00402CDE)
2000 база кода
4000 база данных
400000 база образа (от 00400000 до 00407FFF)
2000 выравнивание раздела
200 выравнивание файла
4.00 версия операционной системы
0.00 версия образа
4.00 версия подсистемы
0 версия Win32
8000 размер образа
200 размер заголовков
0 контрольная сумма
3 подсистема (Windows CUI)

...

```

Запомните, что подавляющему большинству программистов .NET никогда не придется беспокоиться о формате данных заголовков, встроенных в сборку .NET. Если вы не занимаетесь разработкой нового компилятора языка .NET (в таком случае об этой информации придется позаботиться), то можете не вникать в тонкие детали данных заголовков. Однако помните, что эта информация используется “за кулисами”, когда Windows загружает двоичный образ в память.

Заголовок файла CLR

Заголовок CLR — это блок данных, который должны поддерживать все сборки .NET (это они и делают, благодаря компилятору C#) для того, чтобы они могли обслуживаться CLR-средой. Вкратце, в этом заголовке определены многочисленные флаги, которые позволяют исполняющей среде воспринимать компоновку управляемого файла. Например, существуют флаги, которые идентифицируют местоположение метаданных и ресурсов внутри файла, версию исполняющей среды, для которой построена сборка, значение (необязательного) открытого ключа и т.д. Чтобы просмотреть данные заголовка CLR для сборки .NET, необходимо открыть сборку в утилите `dumpbin.exe` с указанием флага `/clrheader`:

```
dumpbin /clrheader CarLibrary.dll
```

Вот как будут выглядеть данные заголовка CLR в этой сборке .NET:

```
Dump of file CarLibrary.dll
File Type: DLL
```

```

clr Header:
  48 cb
  2.05 runtime version
  2164 [  A74] RVA [size] of MetaData Directory
  1 flags
  IL Only

```

```

0 entry point token
0 [      0] RVA [size] of Resources Directory
0 [      0] RVA [size] of StrongNameSignature Directory
0 [      0] RVA [size] of CodeManagerTable Directory
0 [      0] RVA [size] of VTableFixups Directory
0 [      0] RVA [size] of ExportAddressTableJumps Directory
0 [      0] RVA [size] of ManagedNativeHeader Directory

```

Summary

```

2000 .reloc
2000 .rsrc
2000 .text

```

Данные дампа файла CarLibrary.dll

Тип файла: DLL

Заголовок clr:

```

48 cb
2.05 версия исполняющей среды
2164 [      A74] RVA [размер] каталога MetaData
1 flags
    IL Only
0 entry point token
0 [      0] RVA [размер] каталога Resources
0 [      0] RVA [размер] каталога StrongNameSignature
0 [      0] RVA [размер] каталога CodeManagerTable
0 [      0] RVA [размер] каталога VTableFixups
0 [      0] RVA [размер] каталога ExportAddressTableJumps
0 [      0] RVA [размер] каталога ManagedNativeHeader

```

Сводка

```

2000 .reloc
2000 .rsrc
2000 .text

```

Опять-таки, разработчикам приложений .NET не придется беспокоиться о тонких деталях информации заголовка CLR. Нужно лишь помнить, что каждая сборка .NET содержит эти данные, которые используются “за кулисами” исполняющей средой .NET при загрузке образа в память. Теперь рассмотрим информацию, которая является гораздо более полезной при решении повседневных программистских задач.

CIL-код, метаданные типов и манифест сборки

Ядром любой сборки является код на языке CIL, который, как уже упоминалось ранее, представляет собой промежуточный язык, не зависящий ни от платформы, ни от процессора. Во время выполнения внутренних CIL-код “на лету” (посредством JIT-компилятора) компилируется в инструкции, соответствующие конкретной платформе и процессору. Благодаря этому, сборки .NET в действительности могут выполняться в средах разнообразных архитектур, устройств и операционных систем. (Хотя разработчики .NET-приложений могут спокойно жить и работать, не разбираясь в деталях языка программирования CIL, в главе 18 предлагается введение в синтаксис и семантику CIL.)

Сборка также содержит метаданные, полностью описывающие формат находящихся внутри нее типов, а также внешних типов, на которые она ссылается. Исполняющая среда .NET использует эти метаданные для выяснения местоположения типов (и их членов) внутри двоичного файла, для размещения типов в памяти и для упрощения удаленного вызова методов. Более подробно о формате этих метаданных будет рассказываться в главе 15 при рассмотрении служб рефлексии.

Сборка должна также содержать ассоциируемый с ней *манифест* (по-другому называемый *метаданными сборки*). Манифест документирует каждый модуль внутри

сборки, версию сборки, а также любые *внешние* сборки, на которые ссылается текущая сборка. Как будет показано далее в главе, манифест сборки интенсивно применяется CLR-средой в течение процесса определения местоположений, указываемых ссылками на внешние сборки.

Необязательные ресурсы сборки

И, наконец, сборка .NET может содержать любое количество встроенных ресурсов, таких как значки приложения, файлы изображений, звуковые клипы или таблицы строк. В действительности платформа .NET поддерживает *подчиненные сборки*, которые содержат только локализованные ресурсы и ничего другого. Это может быть очень удобно, когда необходимо разделять ресурсы на основе культуры (русской, английской, немецкой и т.п.) при построении интернационального программного обеспечения. Тема создания подчиненных сборок выходит за рамки настоящей книги; при желании получить дополнительные сведения о подчиненных сборках обращайтесь в документацию по .NET Framework 4.5.

Построение и использование специальной библиотеки классов

Чтобы приступить к исследованию мира библиотек классов .NET, давайте вначале создадим сборку *.dll (по имени CarLibrary), которая содержит небольшой набор открытых типов. Для построения библиотеки классов в Visual Studio выберите рабочее пространство проекта Class Library (Библиотека классов) через пункт меню File⇒New Project (Файл⇒Новый проект), как показано на рис. 14.2.

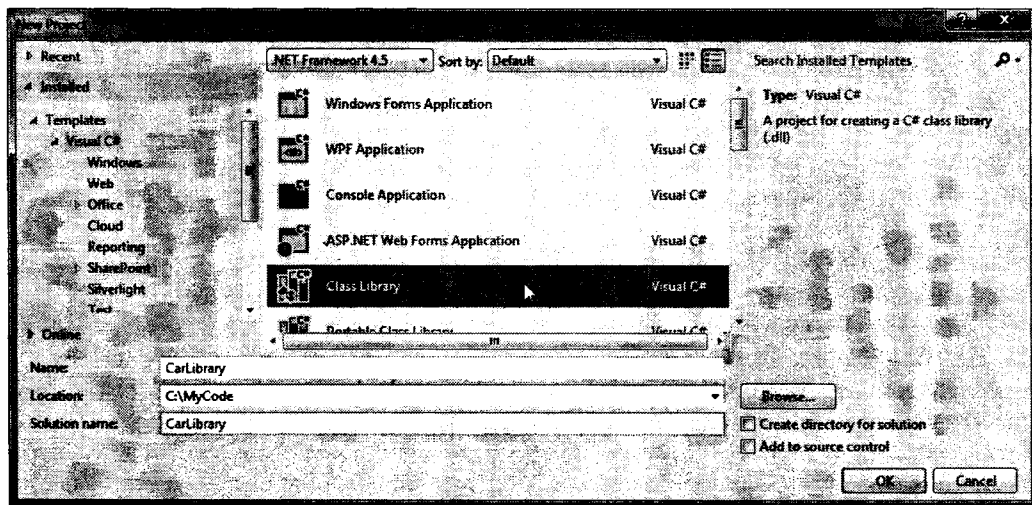


Рис. 14.2. Создание библиотеки классов C#

Первым делом, добавим в библиотеку абстрактный базовый класс по имени Car, который определяет разнообразные данные состояния через синтаксис автоматических свойств. Этот класс также имеет единственный абстрактный метод TurboBoost(), который использует специальное перечисление (EngineState), представляющее текущее состояние двигателя автомобиля, как показано ниже:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CarLibrary
{
    // Представляет состояние двигателя.
    public enum EngineState
    { engineAlive, engineDead }

    // Абстрактный базовый класс в иерархии.
    public abstract class Car
    {
        public string PetName {get; set;}
        public int CurrentSpeed {get; set;}
        public int MaxSpeed {get; set;}

        protected EngineState egnState = EngineState.engineAlive;
        public EngineState EngineState
        {
            get { return egnState; }
        }

        public abstract void TurboBoost();

        public Car(){}
        public Car(string name, int maxSp, int currSp)
        {
            PetName = name; MaxSpeed = maxSp; CurrentSpeed = currSp;
        }
    }
}

```

Теперь предположим, что есть два непосредственных потомка `Car` с именами `MiniVan` (минивэн) и `SportsCar` (спортивный автомобиль). Каждый из них переопределяет абстрактный метод `TurboBoost()` для отображения соответствующего сообщения в окне сообщений `Windows Forms`. Вставим в проект новый файл классов `C#` по имени `DerivedCars.cs` со следующим кодом:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

// Продолжайте чтение! Этот код не скомпилируется до тех пор,
// пока не будет добавлена ссылка на библиотеку .NET.
using System.Windows.Forms;

namespace CarLibrary
{
    public class SportsCar : Car
    {
        public SportsCar(){}
        public SportsCar(string name, int maxSp, int currSp)
            : base (name, maxSp, currSp){ }

        public override void TurboBoost()
        {
            MessageBox.Show("Ramming speed!", "Faster is better...");
        }
    }
}

```

```

public class MiniVan : Car
{
    public MiniVan() { }
    public MiniVan(string name, int maxSp, int currSp)
        : base (name, maxSp, currSp) { }
    public override void TurboBoost()
    {
        // Минивэны имеют плохие возможности ускорения!
        engnState = EngineState.engineDead;
        MessageBox.Show("Eek!", "Your engine block exploded!");
    }
}

```

Обратите внимание, что каждый подкласс реализует метод `TurboBoost()` с использованием класса `MessageBox` из `Windows Forms`, который определен в сборке `System.Windows.Forms.dll`. Для использования в своей сборке типов, определенных внутри указанной внешней сборки, проект `CarLibrary` должен установить ссылку на соответствующий двоичный файл с помощью диалогового окна `Add Reference` (Добавление ссылки), которое показано на рис. 14.3 (чтобы открыть его, выберите пункт меню `Project` ⇒ `Add Reference` (Проект ⇒ Добавить ссылку)).

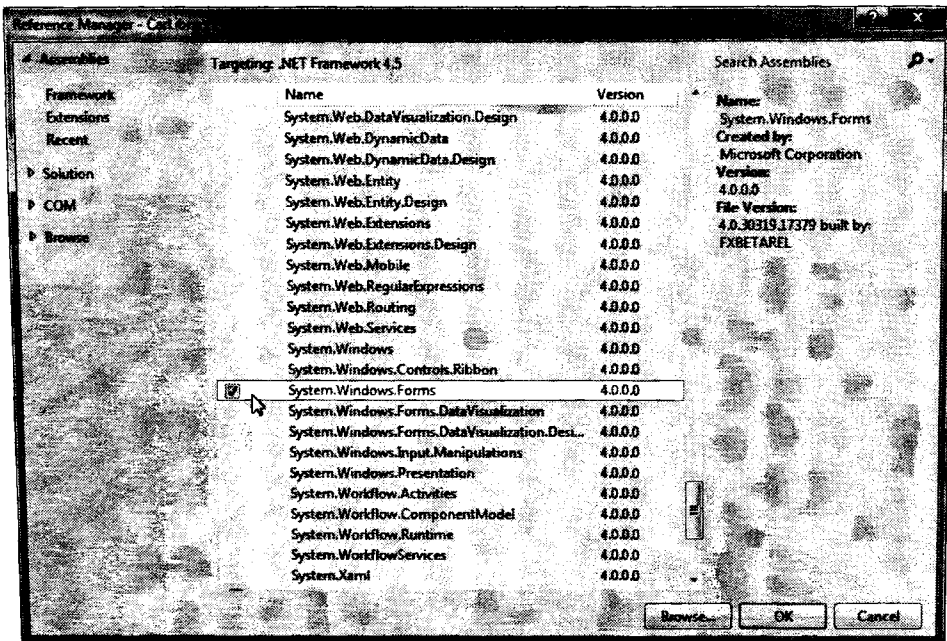


Рис. 14.3. Добавление ссылок на внешние сборки .NET с помощью диалогового окна `Add Reference`

Очень важно знать, что в области `Framework` диалогового окна `Add Reference` отображаются далеко не все присутствующие на машине сборки. Диалоговое окно `Add Reference` не будет показывать специальные библиотеки, равно как и не будет отображать все библиотеки, расположенные в ГАС (соответствующие объяснения приводятся далее в главе). Вместо этого данное диалоговое окно просто представляет список общих сборок, которые среда `Visual Studio` была изначально запрограммирована отображать. При построении приложений, нуждающихся в использовании сборки, не перечисленной в диалоговом окне `Add Reference`, понадобится щелкнуть на узле `Browse` (Обзор) и вручную перейти к интересующему файлу `*.dll` или `*.exe`.

На заметку! Также следует знать о том, что в области Recent (Недавние ссылки) диалогового окна Add Reference поддерживается постоянно обновляемый список сборок, на которые производились ссылки ранее. Он может быть очень удобен, т.к. во многих проектах .NET часто приходится применять один и тот же ключевой набор внешних библиотек.

Исследование манифеста

Прежде чем переходить к использованию библиотеки CarLibrary.dll в клиентском приложении, давайте сначала посмотрим, как она устроена внутри. Предполагая, что проект скомпилирован, загрузим CarLibrary.dll в утилиту ildasm.exe, выбрав пункт меню File⇒Open (Файл⇒Открыть) и перейдя в подкаталог \bin\Debug проекта CarLibrary. После этого вы должны увидеть свою библиотеку, отображаемую в дизассемблере IL (рис. 14.4).

Теперь откроем манифест сборки CarLibrary.dll, дважды щелкнув на значке MANIFEST (Манифест). В первом блоке кода любого манифеста указываются все внешние сборки, требуемые текущей сборкой для нормального функционирования. Вспомните, что в CarLibrary.dll используются типы из внешних сборок mscorlib.dll и System.Windows.Forms.dll, поэтому они обе отображаются в манифесте с применением лексемы .assembly extern:

```
.assembly extern mscorlib
{
    .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
    .ver 4:0:0:0
}
.assembly extern System.Windows.Forms
{
    .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
    .ver 4:0:0:0
}
```

Здесь каждый блок .assembly extern сопровождается уточняющими директивами .publickeytoken и .ver. Инструкция .publickeytoken присутствует только в случае, если сборка была сконфигурирована со *строгим именем* (более подробно о строгих именах речь пойдет в разделе “Понятие строгих имен” этой главы). Лексема .ver определяет числовой идентификатор версии ссылаемой сборки.

После ссылок на внешние сборки находится набор лексем .custom, которые идентифицируют атрибуты уровня сборки (информация об авторском праве, название компании, версия сборки и т.д.). Ниже приведена небольшая часть этой порции данных манифеста:

```
.assembly CarLibrary
{
    .custom instance void ...AssemblyDescriptionAttribute...
    .custom instance void ...AssemblyConfigurationAttribute...
    .custom instance void ...RuntimeCompatibilityAttribute...
    .custom instance void ...TargetFrameworkAttribute...
    .custom instance void ...AssemblyTitleAttribute...
    .custom instance void ...AssemblyTrademarkAttribute...
```

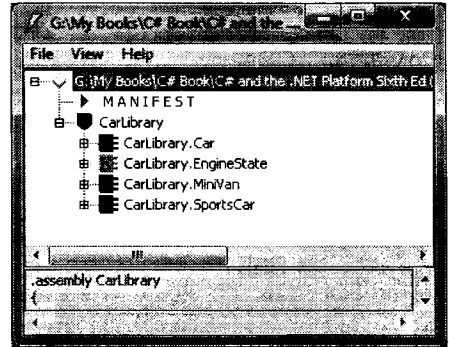


Рис. 14.4. Библиотека CarLibrary.dll, загруженная в ildasm.exe


```
.custom instance void ...AssemblyCompanyAttribute...
.custom instance void ...AssemblyProductAttribute...
.custom instance void ...AssemblyCopyrightAttribute...
...
.ver 1:0:0:0
}
.module CarLibrary.dll
```

Обычно эти параметры устанавливаются визуально с применением редактора свойств текущего проекта. Вернитесь в среду Visual Studio, щелкните на значке Properties (Свойства) в окне Solution Explorer, а затем щелкните на кнопке Assembly Information (Информация о сборке) на вкладке Application (Приложение), выбранной автоматически. Откроется диалоговое окно Assembly Information (Информация о сборке), показанное на рис. 14.5.

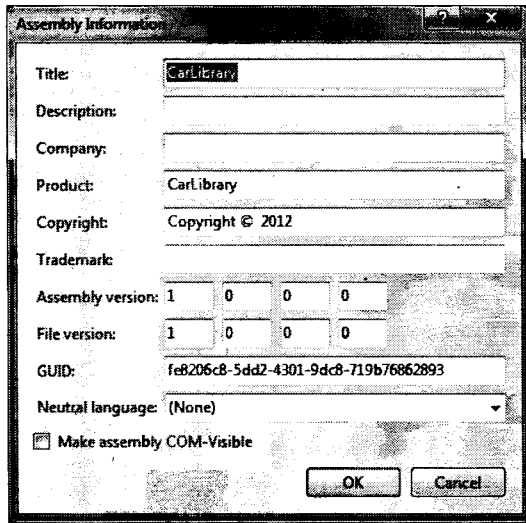


Рис. 14.5. Редактирование информации о сборке в диалоговом окне Assembly Information

После сохранения изменений обновляется файл AssemblyInfo.cs проекта, который поддерживается Visual Studio и может быть просмотрен за счет раскрытия в окне Solution Explorer узла Properties (Свойства), как показано на рис. 14.6.

В этом файле C# можно обнаружить набор заключенных в квадратные скобки *атрибутов* .NET, например:

```
[assembly: AssemblyTitle("CarLibrary")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("CarLibrary")]
[assembly: AssemblyCopyright("Copyright © 2012")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
```

О роли атрибутов будет более подробно рассказываться в главе 15, поэтому на данном этапе беспокоиться о деталях не нужно. Пока достаточно знать, что большинство из атрибутов в файле AssemblyInfo.cs будет использоваться для обновления значений в блоках .custom внутри манифеста сборки.

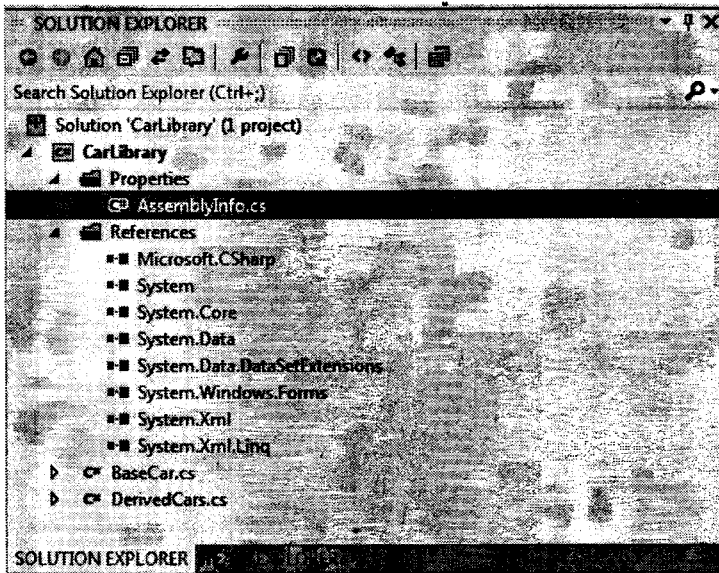


Рис. 14.6. Использование редактора свойств приводит к обновлению файла AssemblyInfo.cs

Исследование CIL-кода

Вспомните, что сборка не содержит специфичных для платформы инструкций; вместо этого в ней хранятся инструкции на независимом от платформы общем промежуточном языке (Common Intermediate Language — CIL). Когда исполняющая среда .NET загружает сборку в память, лежащий в ее основе CIL-код (с помощью JIT-компилятора) компилируется и преобразуется в инструкции, воспринимаемые целевой платформой. Например, если вернуться в `ildasm.exe` и дважды щелкнуть на методе `TurboBoost()` класса `SportsCar`, откроется новое окно, в котором будут отображаться CIL-инструкции, реализующие данный метод:

```
.method public hidebysig virtual instance void
  TurboBoost() cil managed
{
  // Code size      18 (0x12)
  .maxstack 8
  IL_0000: nop
  IL_0001: ldstr "Ramming speed!"
  IL_0006: ldstr "Faster is better..."
  IL_000b: call valuetype [System.Windows.Forms]System.Windows.Forms.DialogResult
    [System.Windows.Forms]System.Windows.Forms.MessageBox::Show(string, string)
  IL_0010: pop
  IL_0011: ret
} // end of method SportsCar::TurboBoost
```

Хотя большинству разработчиков приложений .NET не нужно глубоко разбираться в деталях CIL-кода при повседневной работе, в главе 18 будут приведены более подробные сведения о синтаксисе и семантике CIL. Понимание грамматики языка CIL может оказаться полезным при создании более сложных приложений, которые требуют более сложных служб, таких как конструирование сборок во время выполнения (см. главу 18).

Исследование метаданных типов

Если перед построением приложений, использующих вашу специальную библиотеку .NET, нажать комбинацию клавиш <Ctrl+M>, в окне `ildasm.exe` отобразятся метаданные для всех типов внутри сборки `CarLibrary.dll` (рис. 14.7).

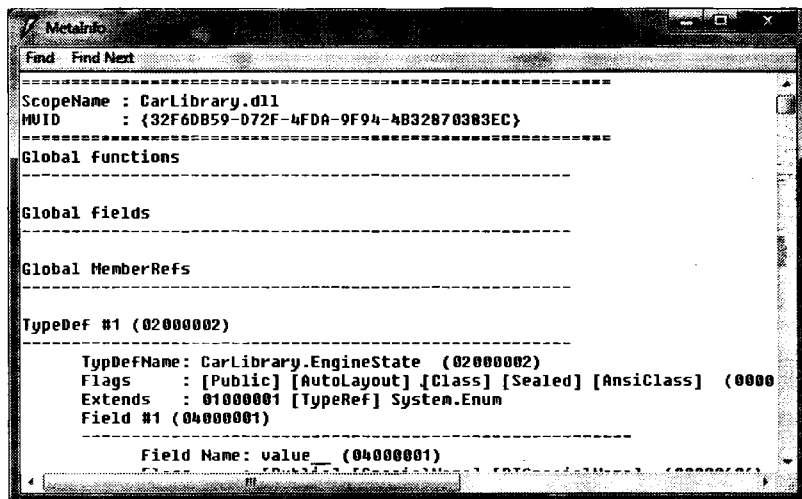


Рис. 14.7. Метаданные для типов, имеющих в сборке `CarLibrary.dll`

Как объясняется в следующей главе, метаданные сборки являются очень важным элементом платформы .NET и служат основой для многочисленных технологий (сериализации объектов, позднего связывания, расширяемых приложений и т.д.). В любом случае, теперь, когда мы заглянули внутрь сборки `CarLibrary.dll`, можно приступать к созданию клиентских приложений, в которых будут использоваться типы из сборки.

Исходный код. Проект `CarLibrary` доступен в подкаталоге Chapter 14.

Построение клиентского приложения на C#

Поскольку все типы в `CarLibrary` были объявлены с ключевым словом `public`, они могут применяться в других приложениях .NET. Вспомните, что типы могут также объявляться с ключевым словом `internal` (на самом деле это стандартный режим доступа в C#). Внутренние (`internal`) типы могут использоваться только сборкой, в которой они определены. Внешние клиенты не могут ни видеть, ни создавать типы, помеченные ключевым словом `internal`.

Чтобы воспользоваться функциональностью построенной библиотеки, создадим новый проект консольного приложения на C# по имени `CSharpCarClient`. После этого установим ссылку на `CarLibrary.dll` с помощью узла `Browse` диалогового окна `Add Reference` (если сборка `CarLibrary.dll` компилировалась в Visual Studio, она будет находиться в подкаталоге `\Bin\Debug` внутри каталога проекта `CarLibrary`). Теперь можно приступать к построению клиентского приложения для использования внешних типов. Модифицируем начальный файл кода C#, как показано ниже:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
// Не забудьте импортировать пространство имен CarLibrary!
using CarLibrary;

namespace CSharpCarClient
{
    public class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** C# CarLibrary Client App *****");
            // Создать объект спортивного автомобиля.
            SportsCar viper = new SportsCar("Viper", 240, 40);
            viper.TurboBoost();

            // Создать объект минивэна.
            MiniVan mv = new MiniVan();
            mv.TurboBoost();

            Console.WriteLine("Done. Press any key to terminate");
            Console.ReadLine();
        }
    }
}
```

Этот код выглядит подобным коду в других ранее разработанных приложениях. Единственный интересный момент в нем касается того, что в клиентском приложении C# теперь используются типы, определенные в отдельной специальной библиотеке. Давайте теперь запустим это приложение. Как и следовало ожидать, будут отображаться различные окна с сообщениями.

У вас может возникнуть вопрос: что в точности происходит при добавлении ссылки на CarLibrary.dll в диалоговом окне Add Reference? Если вы щелкнете на кнопке Show All Files (Показать все файлы) в окне Solution Explorer, то увидите, что среда Visual Studio добавила копию исходной библиотеки CarLibrary.dll в подкаталог \bin\Debug каталога проекта CSharpCarClient (рис. 14.8).

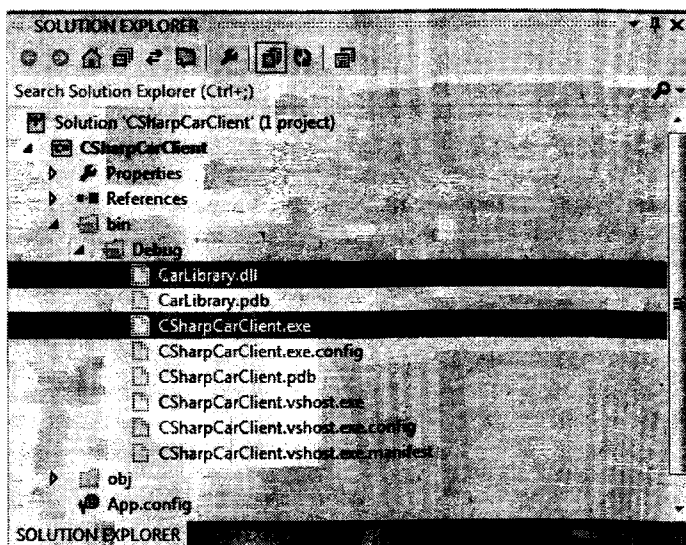


Рис. 14.8. Среда Visual Studio копирует закрытые сборки в каталог клиентского приложения

Как скоро будет объяснено, CarLibrary.dll была сконфигурирована как закрытая сборка (поскольку именно такое поведение применяется автоматически для всех проектов библиотек классов Visual Studio). При ссылке на закрытые сборки в новых приложениях (вроде CSharpCarClient.exe), IDE-среда всегда реагирует помещением копии соответствующей библиотеки в выходной каталог клиентского приложения.

Исходный код. Проект CSharpCarClient доступен в подкаталоге Chapter 14.

Построение клиентского приложения на Visual Basic

Вспомните, что платформа .NET позволяет разработчикам разделять скомпилированный код между языками программирования. В целях иллюстрации языковой независимости платформы .NET создадим еще одно консольное приложение (по имени VisualBasicCarClient), на этот раз используя язык Visual Basic (рис. 14.9). После создания проекта добавим ссылку на CarLibrary.dll с помощью диалогового окна Add Reference (через пункт меню Project⇒Add Reference).

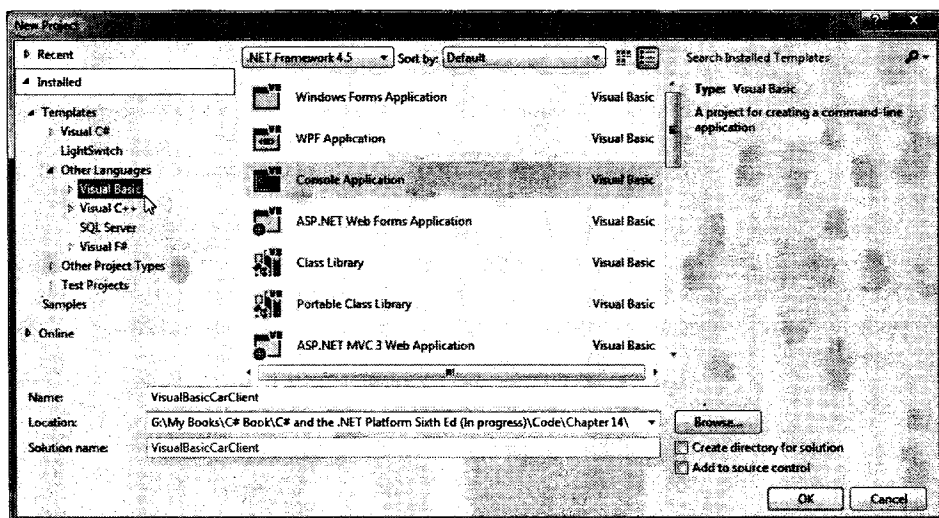


Рис. 14.9. Создание консольного приложения Visual Basic

Подобно C#, Visual Basic позволяет перечислять все пространства имен, используемые в текущем файле. Вместо ключевого слова using, применяемого в C#, в Visual Basic для этого предусмотрено ключевое слово Imports. Добавим в файл кода Module1.vb следующий оператор Imports:

```
Imports CarLibrary
Module Module1
    Sub Main()
    End Sub
End Module
```

Обратите внимание, что метод Main() определен внутри типа модуля Visual Basic. Говоря кратко, модули — это нотация Visual Basic для определения класса, который может содержать только статические методы (что очень похоже на статический класс C#). В любом случае, чтобы испробовать типы MiniVan и SportsCar, используя синтаксис Visual Basic, модифицируем метод Main() следующим образом:

```

Sub Main()
    Console.WriteLine("***** VB CarLibrary Client App *****")

    ' Локальные переменные объявлены с применением ключевого слова Dim.
    Dim myMiniVan As New MiniVan()
    myMiniVan.TurboBoost()

    Dim mySportsCar As New SportsCar()
    mySportsCar.TurboBoost()
    Console.ReadLine()
End Sub

```

Если теперь скомпилировать и запустить приложение, на экране снова будет отображаться последовательность окон с сообщениями. Вдобавок это новое клиентское приложение имеет собственную локальную копию CarLibrary.dll в своем подкаталоге bin\Debug.

Межязыковое наследование в действии

Весьма привлекательным аспектом разработки для .NET является понятие *межязыкового наследования*. В целях иллюстрации создадим новый класс Visual Basic, порожденный от SportsCar (который был написан на C#). Для начала добавим в текущее приложение Visual Basic новый файл класса по имени PerformanceCar.vb (выбрав пункт меню Project⇒Add Class (Проект⇒Добавить класс)). Модифицируем начальное определение этого класса, породив его от типа SportsCar посредством ключевого слова Inherits. Затем переопределим абстрактный метод TurboBoost() с помощью ключевого слова Overrides, как показано ниже:

```

Imports CarLibrary

' Этот класс VB порожден от класса SportsCar, написанного на C#.
Public Class PerformanceCar
    Inherits SportsCar

    Public Overrides Sub TurboBoost()
        Console.WriteLine("Zero to 60 in a cool 4.8 seconds...")
    End Sub
End Class

```

Чтобы протестировать этот новый класс, изменим код метода Main() в модуле следующим образом:

```

Sub Main()
    ...
    Dim dreamCar As New PerformanceCar()

    ' Использовать унаследованное свойство.
    dreamCar.PetName = "Hank"
    dreamCar.TurboBoost()
    Console.ReadLine()
End Sub

```

Обратите внимание, что объект dreamCar способен обращаться к любому открытому члену (такому как свойство PetName), находящемуся выше в цепочке наследования, невзирая на тот факт, что базовый класс был определен на совершенно другом языке и в совершенно другой сборке! Возможность расширения классов за пределы границ сборок в независимой от языка манере является очень полезным аспектом цикла разработки для .NET. Это существенно упрощает использование скомпилированного кода, написанного программистами, которые не захотели создавать свой разделяемый код на языке C#.

Понятие закрытых сборок

С технической точки зрения библиотеки классов, которые создавались до сих пор в главе, разворачивались как *закрытые сборки*. Такие сборки должны всегда размещаться в том же самом каталоге, что и клиентское приложение, которое ими пользуется (в *каталоге приложения*), или в каком-то из его подкаталогов. Вспомните, что при установке ссылки на CarLibrary.dll при построении приложений CSharpCarClient.exe и VbNetCarClient.exe среда Visual Studio помещает копию CarLibrary.dll в каталоги этих клиентских приложений (во всяком случае, после первой компиляции).

Когда клиентская программа использует типы, определенные во внешней сборке CarLibrary.dll, среда CLR просто загружает локальную копию этой сборки. Поскольку исполняющая среда .NET не должна заглядывать в системный реестр при поиске внешних сборок, можно переместить сборки CSharpCarClient.exe (или VisualBasicCarClient.exe) и CarLibrary.dll в любое другое место на машине и успешно запускать приложение (это часто называется *развертыванием с помощью Хсору*).

Удаление (или копирование) приложения, в котором используются закрытые сборки, не требует особых усилий: достаточно просто удалить (или скопировать) каталог приложения. Более важно то, что удаление закрытых сборок не может нарушить работу других приложений на машине.

Идентичность закрытой сборки

Полная идентичность закрытой сборки включает дружественное имя и номер версии, которые записываются в манифесте сборки. *Дружественное имя* — это просто название модуля, который содержит манифест сборки, без файлового расширения. Например, если просмотреть манифест сборки CarLibrary.dll, в нем можно обнаружить следующее:

```
.assembly CarLibrary
{
    ...
    .ver 1:0:0:0
}
```

Учитывая изолированную природу закрытой сборки, должно быть понятно, что CLR-среда не использует номер версии при выяснении места ее размещения. Она предполагает, что закрытые сборки не нуждаются в выполнении сложной проверки версий, поскольку клиентское приложение является единственной сущностью, которая “знает” об их существовании. По этой причине на одной машине (с высокой вероятностью) может размещаться множество копий одной и той же сборки в различных каталогах приложений.

Понятие процесса зондирования

Исполняющая среда .NET определяет местонахождение закрытой сборки с применением приема под названием *зондирование*, который в действительности не такой надоедливый, как может показаться поначалу. Зондирование — это процесс отображения запроса внешней сборки на местоположение соответствующего двоичного файла. Строго говоря, запрос внешней сборки может быть либо *явным*, либо *неявным*. Неявный запрос загрузки происходит, когда CLR-среда заглядывает в манифест для определения, где находится требуемая сборка, по лексемам `.assembly extern`.

Например:

```
// Неявный запрос загрузки.
.assembly extern CarLibrary
{ ... }
```

Явный запрос загрузки осуществляется программно с использованием метода `Load()` или `LoadFrom()` класса `System.Reflection.Assembly`, обычно в целях позднего связывания или динамического вызова членов типа. Более подробно эти темы рассматриваются в главе 15, а пока ниже приведен пример явного запроса загрузки:

```
// Явный запрос загрузки, основанный на дружественном имени.
Assembly asm = Assembly.Load("CarLibrary");
```

В любом случае CLR-среда извлекает дружественное имя сборки и начинает зондировать каталог клиентского приложения в поисках файла по имени `CarLibrary.dll`. Если этот файл обнаружить не удалось, предпринимается попытка найти исполняемую сборку с таким же дружественным именем (например, `CarLibrary.exe`). Если ни один из упомянутых файлов в каталоге приложения не найден, исполняющая среда генерирует исключение `FileNotFoundException` во время выполнения.

На заметку! Формально, когда копия запрашиваемой сборки не найдена в каталоге клиентского приложения, CLR-среда также будет искать клиентский подкаталог, имеющий дружественное имя сборки (например, `C:\MyClient\CarLibrary`). Если запрашиваемая сборка хранится в этом подкаталоге, CLR-среда загрузит ее в память.

Конфигурирование закрытых сборок

Хотя вполне вероятно, что какое-нибудь .NET-приложение может быть развернуто простым копированием всех необходимых сборок в единственную папку на жестком диске пользователя, скорее всего, понадобится определить несколько подкаталогов для группирования связанного содержимого. Например, предположим, что имеется каталог приложения под названием `C:\MyApp`, в котором содержится файл `CSharpCarClient.exe`. В этом каталоге может быть предусмотрен подкаталог `MyLibraries`, хранящий сборку `CarLibrary.dll`.

Несмотря на подразумеваемую взаимосвязь между двумя этими каталогами, CLR-среда *не будет* зондировать подкаталог `MyLibraries`, если только не предоставить ей конфигурационный файл. Конфигурационные файлы содержат различные XML-элементы, которые позволяют влиять на процесс зондирования. Конфигурационные файлы должны иметь такое же имя, как у запускаемого приложения, с расширением `*.config` и быть развернутыми в каталоге клиентского приложения. Таким образом, если нужно создать конфигурационный файл для приложения `CSharpCarClient.exe`, он должен иметь имя `CSharpCarClient.exe.config` и располагаться (в этом примере) в каталоге `C:\MyApp`.

В целях иллюстрации процесса создадим на диске `C:` новый каталог по имени `MyApp`, используя проводник `Windows`. Далее скопируем в него файлы `CSharpCarClient.exe` и `CarLibrary.dll` и запустим программу, дважды щелкнув на исполняемом файле. В этот момент программа должна запуститься успешно.

Теперь создадим в `C:\MyApp` новый подкаталог по имени `MyLibraries` (рис. 14.10) и переместим в него сборку `CarLibrary.dll`.

Попробуем запустить программу снова, дважды щелкнув на исполняемом файле. Поскольку CLR-среда не удастся обнаружить сборку по имени `CarLibrary` непосредственно в каталоге приложения, возникает необработанное исключение `FileNotFoundException`.

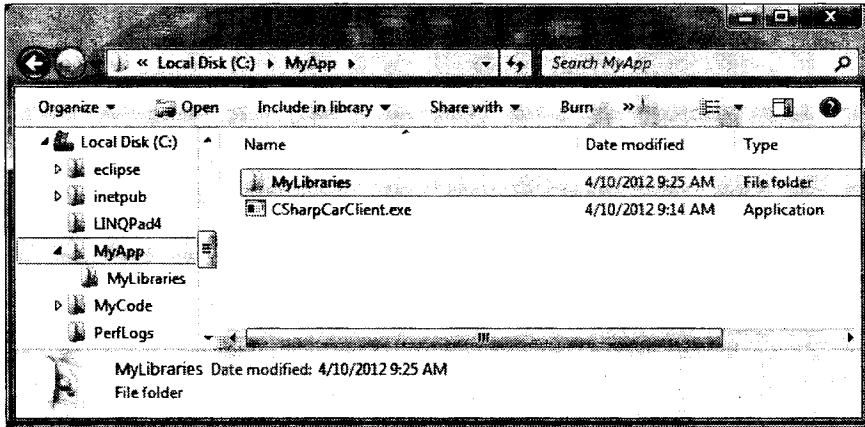


Рис. 14.10. Теперь файл CarLibrary.dll находится в подкаталоге MyLibraries

Чтобы указать CLR-среде на необходимость зондирования подкаталога MyLibraries, создадим с помощью любого текстового редактора новый конфигурационный файл по имени CSharpCarClient.exe.config и сохраним его в каталоге, содержащем приложение CSharpCarClient.exe (C:\MyApp в рассматриваемом примере). Откроем этот файл и введем в него следующее содержимое (обратите внимание, что язык XML чувствителен к регистру символов):

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <probing privatePath="MyLibraries"/>
    </assemblyBinding>
  </runtime>
</configuration>
```

Файлы *.config в .NET всегда начинаются с корневого элемента <configuration>. Вложенный в него элемент <runtime> может содержать элемент <assemblyBinding>, а тот, в свою очередь — вложенный элемент <probing>. В этом примере главный интерес представляет атрибут privatePath, поскольку именно он служит для указания подкаталогов внутри каталога приложения, которые должна зондировать CLR-среда.

Завершив создание конфигурационного файла CSharpCarClient.exe.config, снова запустим клиентское приложение. На этот раз выполнение CSharpCarClient.exe должно пройти без проблем (если это не так, проверьте конфигурационный файл на предмет опечаток).

Обратите внимание на то, что в элементе <probing> не указывается, какая сборка размещена в заданном подкаталоге. Другими словами, нельзя сказать, что сборка CarLibrary находится в подкаталоге MyLibraries, а сборка MathLibrary — в подкаталоге OtherStuff. Элемент <probing> просто инструктирует CLR-среду о необходимости просмотра всех перечисленных подкаталогов на предмет наличия запрашиваемой сборки до тех пор, пока не встретится первое совпадение.

На заметку! Очень важно запомнить, что атрибут privatePath не может использоваться для указания абсолютного (C:\Папка\Подпапка) или относительного (..\Папка\ДругаяПапка) пути! Если нужно указать каталог, находящийся за пределами каталога клиентского приложения, понадобится применить совершенно другой XML-элемент под названием <codeBase> (который более подробно рассматривается позже в главе).

Атрибут `privatePath` допускает указание множества подкаталогов в виде списка значений, разделенных точками с запятой. В данном случае в этом нет необходимости, однако ниже приведен пример, в котором CLR-среде указывается просматривать клиентские подкаталоги `MyLibraries` и `MyLibraries\Tests`:

```
<probing privatePath="MyLibraries;MyLibraries\Tests"/>
```

Давайте в целях тестирования изменим (произвольным образом) имя конфигурационного файла и попробуем запустить приложение снова. На этот раз клиентское приложение должно дать сбой. Вспомните, что имя файла `*.config` должно предваряться именем связанного с ним клиентского приложения. В качестве последнего теста откроем конфигурационный файл для редактирования и изменим регистр символов любого XML-элемента. И снова клиентское приложение должно дать сбой (т.к. язык XML чувствителен к регистру символов).

На заметку! Важно понимать, что CLR-среда будет загружать сборку, которая во время процесса зондирования обнаруживается первой. Например, если в каталоге `C:\MyApp` имеется копия `CarLibrary.dll`, то она и будет загружена в память, а копия, содержащаяся в подкаталоге `MyLibraries`, полностью игнорируется.

Роль файла `App.Config`

Хотя конфигурационные XML-файлы всегда можно создавать вручную с использованием любого текстового редактора, среда Visual Studio позволяет создавать конфигурационный файл во время разработки клиентской программы. Первый шаг заключается в добавлении к проекту нового элемента Application Configuration File (Конфигурационный файл приложения) через пункт меню `Project` → `Add New Item`. Обратите внимание на рис. 14.11, что для этого файла оставлено предложенное стандартное имя `App.config`.

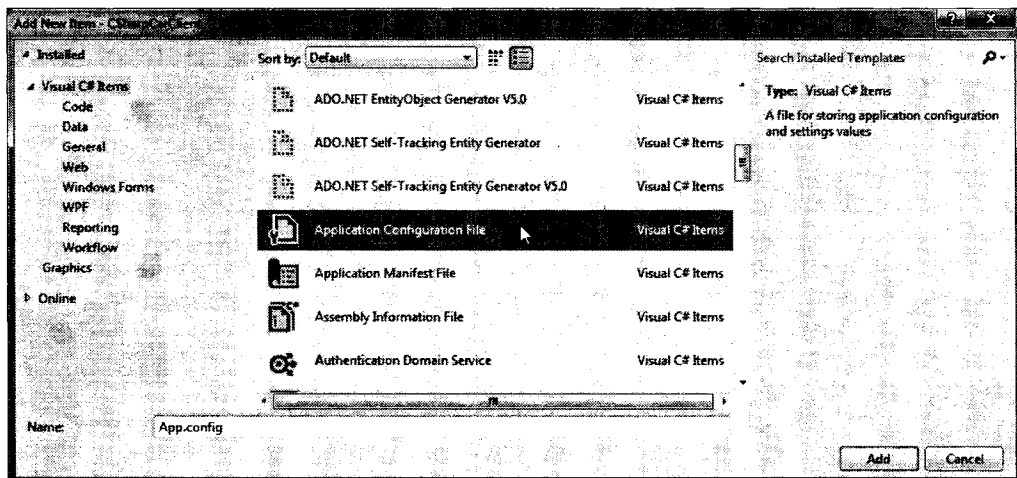


Рис. 14.11. Вставка нового конфигурационного XML-файла

Открыв этот файл для просмотра, можно увидеть минимальный набор инструкций, к которым будут добавляться дополнительные элементы:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
</configuration>
```

Отметим один замечательный момент. Каждый раз, когда проект компилируется, Visual Studio будет автоматически копировать данные из App.config в новый файл, расположенный в каталоге \bin\Debug, используя подходящие соглашения об именовании (вроде CSharpCarClient.exe.config). Однако это будет происходить только в случае, если конфигурационный файл действительно имеет имя App.config (рис. 14.12).

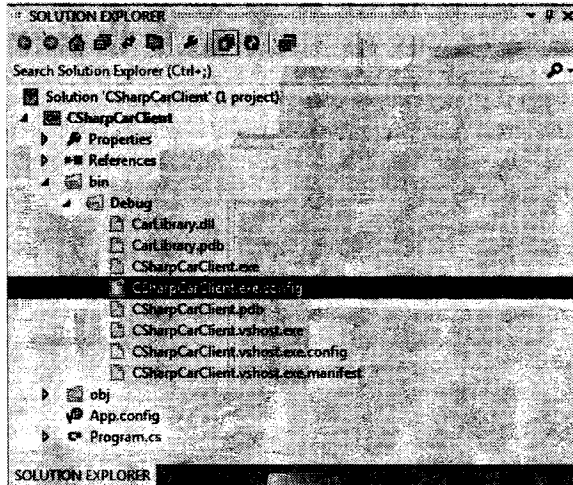


Рис. 14.12. Содержимое App.config будет скопировано в корректно именованный файл *.config, находящийся в выходном каталоге проекта

При таком подходе все, что требуется — это поддерживать файл App.config, а среда Visual Studio позаботится о том, чтобы в каталоге приложения содержались самые последние и актуальные конфигурационные данные (даже если проект будет переименован).

Понятие разделяемых сборок

Теперь, когда вы знаете, как развертывать и конфигурировать закрытые сборки, можно переходить к исследованию роли *разделяемых сборок*. Подобно закрытой сборке, любая разделяемая сборка — это коллекция типов, предназначенных для многократного использования во множестве проектов. Самое очевидное отличие между разделяемыми и закрытыми сборками состоит в том, что единственная копия разделяемой сборки может применяться несколькими приложениями на одной и той же машине.

Вспомните о том, что все приложения, создаваемые в этой книге, требуют доступа к mscorlib.dll. Если вы заглянете в каталог любого из этих клиентских приложений, то никакой закрытой копии указанной сборки .NET там не найдете. Причина в том, что сборка mscorlib.dll развернута как разделяемая. Очевидно, что если необходимо создать библиотеку классов для использования в масштабах всей машины, именно так и следует поступать.

На заметку! Принятие решения о том, развертывать библиотеку кода как закрытую или как разделяемую сборку, является еще одним вопросом, который должен быть продуман на этапе проектирования, и ответ на него зависит от множества специфических деталей проекта. Существует эмпирическое правило: при построении библиотек, которые необходимо использовать во множестве различных приложений, разделяемые сборки могут быть намного удобнее, поскольку их очень легко обновлять до новых версий (как вскоре будет показано).

Глобальный кеш сборок

Как вытекает из сказанного ранее, разделяемая сборка не развертывается внутри того же самого каталога, что и приложение, в котором она используется. Вместо этого разделяемые сборки устанавливаются в глобальный кеш сборок (Global Assembly Cache — GAC). Тем не менее, точное местоположение GAC будет зависеть от версии платформы .NET, установленной на целевом компьютере.

На машинах с версиями, предшествующими .NET 4.0, глобальный кеш сборок размещен в подкаталоге Assembly внутри каталога Windows (например, C:\Windows\assembly). В наши дни мы можем считать это “историческим GAC”, потому что он содержит только библиотеки .NET, скомпилированные для версий 1.0, 2.0, 3.0 или 3.5 (рис. 14.13).

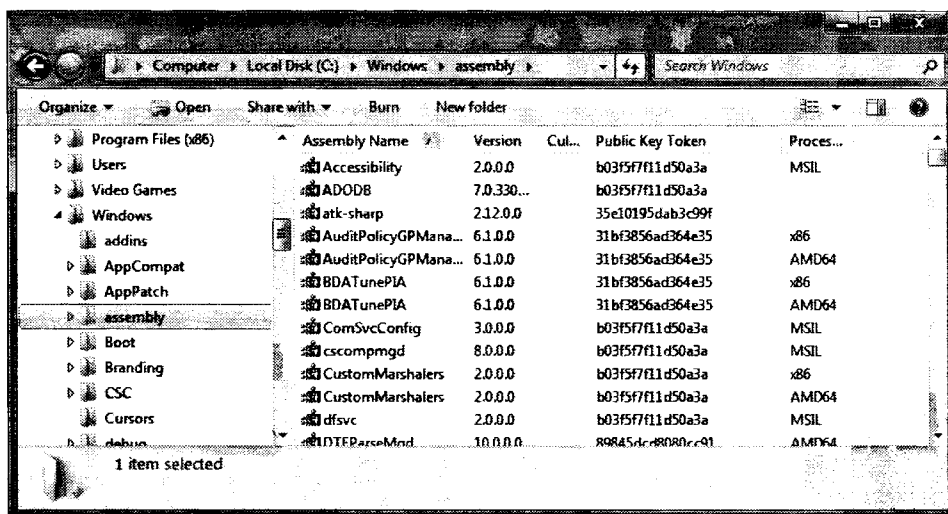


Рис. 14.13. “Исторический” глобальный кеш сборок

На заметку! Устанавливать в GAC исполняемые сборки (*.exe) нельзя. В качестве разделяемых можно развертывать только сборки с расширением *.dll.

С выходом версии .NET 4.0 в Microsoft решили изолировать библиотеки для .NET 4.0 и последующих версий в отдельном местоположении, которое выглядит как C:\Windows\Microsoft.NET\assembly\GAC_MSIL (рис. 14.14).

В этом новом каталоге находится множество подкаталогов, каждый из которых назван идентично дружественному имени отдельной библиотеки кода (например, \System.Windows.Forms, \System.Core и т.д.). Внутри конкретного подкаталога с дружественным именем имеется еще один подкаталог, который всегда именуется в соответствии со следующим соглашением:

v4.0_старший.младший.сборка.редакция_значениеМаркераОткрытогоКлюча

Префикс v4.0 обозначает, что библиотека скомпилирована под .NET 4.0 или последующей версии. За этим префиксом следует одиночный знак подчеркивания (_) и версия рассматриваемой библиотеки (например, 1.0.0.0). После пары знаков подчеркивания находится то, что называется значением маркера открытого ключа. Как будет показано в следующем разделе, значение открытого ключа является частью “строного имени” сборки. Наконец, внутри этого подкаталога находится копия интересующей сборки *.dll.

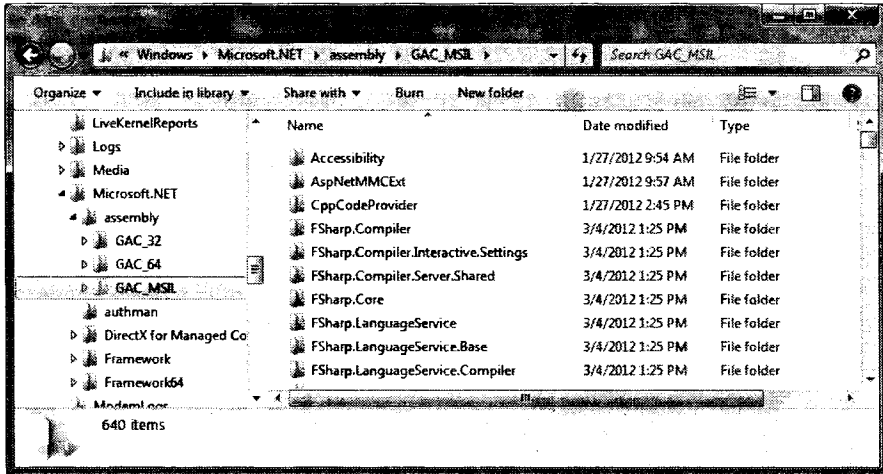


Рис. 14.14. Глобальный кеш сборок в .NET 4.0 и последующих версиях

В настоящей книге предполагается построение приложений с использованием .NET 4.5; таким образом, если вы устанавливаете библиотеку в GAC, она попадет в каталог `C:\Windows\Microsoft.NET\assembly\GAC_MSIL`. Тем не менее, не забывайте, что если проект библиотеки классов сконфигурирован для компиляции с применением .NET 3.5 или более ранней версии, разделяемые библиотеки следует искать в каталоге `C:\Windows\assembly`.

Понятие строгих имен

Перед развертыванием сборки в GAC ей должно быть назначено *строгое имя*, которое используется для уникальной идентификации издателя данного двоичного файла .NET. Следует иметь в виду, что в роли "издателя" может выступать отдельный программист, подразделение внутри компании или компания в целом.

В некоторых отношениях строгое имя является современным .NET-эквивалентом глобально уникальных идентификаторов (GUID), которые применялись в COM. Если вы ранее имели дело с COM, то можете вспомнить, что идентификаторы приложений — это глобально уникальные идентификаторы, указывающие на конкретные COM-приложения. В отличие от значений GUID в COM (которые представляют собой 128-битные числа), строгие имена основаны (частично) на двух криптографически связанных ключах (*открытых и секретных*), которые являются гораздо более уникальными и устойчивыми к подделке, чем простые GUID.

Формально строгое имя состоит из набора связанных данных, большинство которых указывается с использованием перечисленных ниже атрибутов уровня сборки.

- Дружественное имя сборки (которое, как вы помните, представляет собой имя сборки без файлового расширения).
- Номер версии сборки (назначается в атрибуте `[AssemblyVersion]`).
- Значение открытого ключа (назначается в атрибуте `[AssemblyKeyFile]`).
- Необязательное значение, обозначающее культуру, служащее для целей локализации (назначается в атрибуте `[AssemblyCulture]`).
- Встроенная *цифровая подпись*, созданная с использованием хеш-значения содержимого сборки и значения секретного ключа.

Для создания строгого имени сборки сначала генерируются данные открытого и секретного ключей с помощью утилиты `sn.exe` из .NET Framework 4.5. Эта утилита генерирует файл, который обычно оканчивается расширением `*.snk` (Strong Name Key — ключ строгого имени) и содержит данные для двух разных, но математически связанных ключей — открытого и секретного. После того, как компилятору C# указано местонахождение этого файла `*.snk`, он запишет полное значение открытого ключа в манифест сборки с использованием лексемы `.publickey`.

Компилятор C# также сгенерирует хеш-код на основе содержимого всей сборки (CIL-кода, метаданных и т.д.). Как упоминалось в главе 6, хеш-код — это числовое значение, которое является статистически уникальным для фиксированных входных данных. Следовательно, в случае изменения любого аспекта сборки .NET (даже одного символа в каком-нибудь строковом литерале), компилятор выдает другой хеш-код. Затем этот хеш-код комбинируется с содержащимися внутри файла `*.snk` данными секретного ключа для получения цифровой подписи, встраиваемой в данные заголовка CLR сборки. Процесс создания строгого имени проиллюстрирован на рис. 14.15.

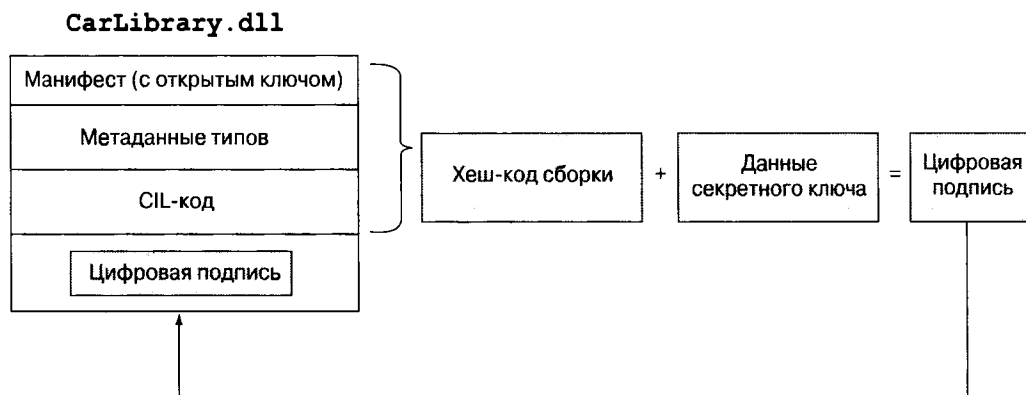


Рис. 14.15. Во время компиляции на основе данных открытого и секретного ключей генерируется цифровая подпись, которая затем вставляется в сборку

Важно понимать, что действительные данные секретного ключа нигде в манифесте не встречаются, а служат только для цифрового подписания содержимого сборки (в сочетании со сгенерированным хеш-кодом). Общая идея использования открытого и секретного ключей связана с необходимостью обеспечения того, что никакие две компании, подразделения или отдельных программиста не будут иметь одну и ту же идентичность в мире .NET. В любом случае по завершении процесса назначения строгого имени сборка может быть установлена в GAC.

На заметку! Строгие имена также предоставляют уровень защиты от потенциальной подделки содержимого сборок. С учетом этого, наилучшим практическим приемом .NET считается назначение строгих имен всем сборкам (включая сборки `.exe`), независимо от того, развертываются они в GAC или нет.

Генерация строгих имен в командной строке

Давайте рассмотрим процесс назначения строгого имени сборке `CarLibrary`, созданной ранее в этой главе. В наши дни необходимый файл `*.snk`, скорее всего, будет генерироваться с применением Visual Studio. Однако в прежние времена (пример-

но до 2003 г.) назначать сборке строгое имя можно было только в командной строке. Посмотрим, как это делается.

В первую очередь должны быть сгенерированы требуемые данные ключей с помощью утилиты `sn.exe`. Хотя эта утилита поддерживает множество опций командной строки, в настоящий момент интерес представляет только флаг `-k`, который заставляет генерировать новый файл с информацией об открытом и секретном ключах. Создадим на диске C: новую папку по имени `MyTestKeyPair`, перейдем в нее в окне командной строки разработчика и введем следующую команду для генерации файла `MyTestKeyPair.snk`:

```
sn -k MyTestKeyPair.snk
```

Теперь, располагая данными о ключах, необходимо проинформировать компилятор C# о том, где расположен файл `MyTestKeyPair.snk`. Как уже рассказывалось ранее в настоящей главе, когда создается любое новое рабочее пространство проекта C# в Visual Studio один из начальных файлов проекта (отображаемых в узле Properties окна Solution Explorer) имеет имя `AssemblyInfo.cs`. Этот файл содержит несколько атрибутов, которые описывают саму сборку. В `AssemblyInfo.cs` можно добавить атрибут `[AssemblyKeyFile]` уровня сборки для сообщения компилятору местоположения действительного файла `*.snk`. Путь должен быть указан в виде строкового параметра, например:

```
[assembly: AssemblyKeyFile(@"C:\MyTestKeyPair\MyTestKeyPair.snk")]
```

На заметку! Когда значение атрибута `[AssemblyKeyFile]` устанавливается вручную, Visual Studio выдаст предупреждающее сообщение о том, что нужно либо указать `csc.exe` опцию `/keyfile`, либо установить файл ключей в окне Properties. Мы сделаем это в IDE-среде немного позже, поэтому просто проигнорируйте полученное предупреждение.

Поскольку версия разделяемой сборки является одним из аспектов строгого имени, выбор номера версии для `CarLibrary.dll` является необходимой деталью. В файле `AssemblyInfo.cs` вы найдете еще один атрибут под названием `AssemblyVersion`. Первоначально его значение установлено в `1.0.0.0`:

```
[assembly: AssemblyVersion("1.0.0.0")]
```

Номер версии в .NET состоит из четырех частей (старшего номера, младшего номера, номера сборки и номера редакции). Хотя указание номера версии возлагается полностью на вас, можно заставить Visual Studio автоматически инкрементировать номера сборки и редакции во время каждой компиляции, используя групповой символ. В рассматриваемом примере необходимость в этом отсутствует; тем не менее, взгляните на следующий код:

```
// Формат: <старший номер>.<младший номер>.<номер сборки>.<номер редакции>
// Допустимые значения для каждой части номера версии лежат в диапазоне от 0 до 65535.
[assembly: AssemblyVersion("1.0.*")]
```

Теперь у компилятора C# есть вся информация, необходимая для генерации строгого имени (т.к. значение культуры в атрибуте `[AssemblyCulture]` не указано, "наследуется" культура, установленная на текущей машине).

Скомпилируем библиотеку кода `CarLibrary`, откроем ее в `ildasm.exe` и заглянем в манифест. Теперь можно видеть, что новый дескриптор `.publickey` содержит полную информацию об открытом ключе, а дескриптор `.ver` хранит номер версии, указанный в атрибуте `[AssemblyVersion]` (рис. 14.16).

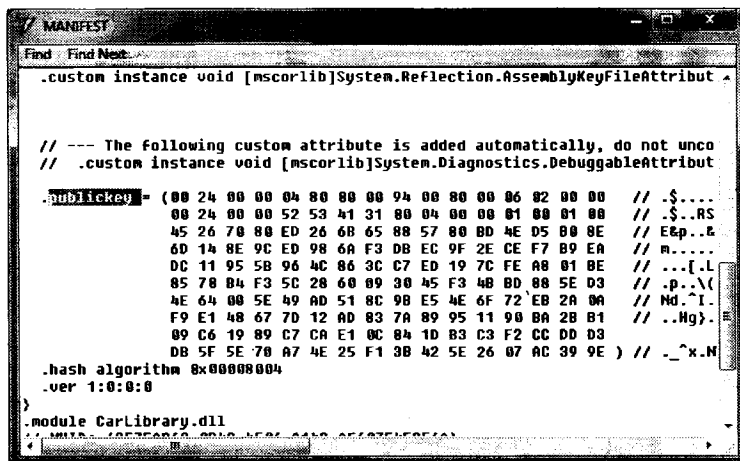


Рис. 14.16. В манифесте строго именованной сборки записана информация об открытом ключе

В этот момент можно было бы развернуть разделяемую сборку CarLibrary.dll в ГАС. Однако вспомните, что теперь для создания сборок со строгими именами разработчики приложений .NET могут применять Visual Studio вместо утилиты командной строки sn.exe. Прежде чем опробовать этот способ создания строгих имен, удалите (или закомментируйте) следующую строку кода в файле AssemblyInfo.cs (при условии, что она была ранее добавлена):

```
// [assembly: AssemblyKeyFile(@"C:\MyTestKeyPair\MyTestKeyPair.snk")]
```

Генерация строгих имен в Visual Studio

Среда Visual Studio позволяет указывать местоположение существующего файла *.snk на странице свойств проекта, а также генерировать новый файл *.snk. Чтобы создать новый файл *.snk для проекта CarLibrary, дважды щелкните на значке Properties (Свойства) в окне Solution Explorer, перейдите в область Signing (Подпись), отметьте флажок Sign the assembly (Подписать сборку) и выберите в раскрывающемся списке вариант <New...> (Новый), как показано на рис. 14.17.

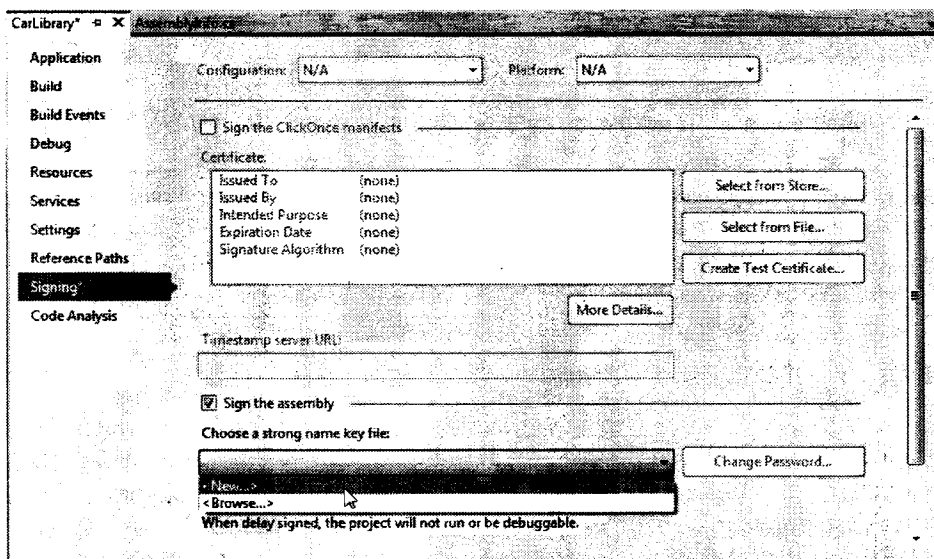


Рис. 14.17. Создание нового файла *.snk в Visual Studio

После этого откроется окно с приглашением указать имя для нового файла *.snk (например, myKeyFile.snk) и флажком Protect my key file with a password (Защитить файл ключей с помощью пароля), отмечать который в рассматриваемом примере не требуется (рис. 14.18).

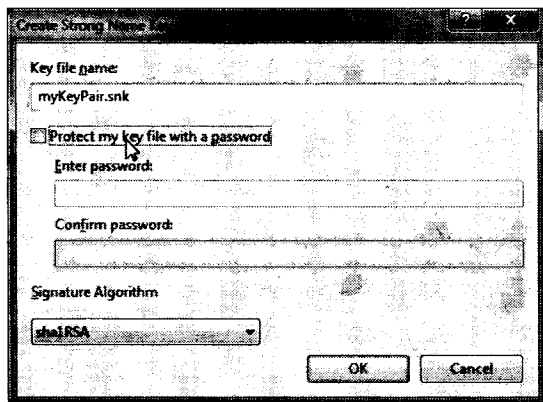


Рис. 14.18. Именование нового файла *.snk в Visual Studio

После этого новый файл *.snk появится в окне Solution Explorer (рис. 14.19). Каждый раз, когда будет производиться компиляция приложения, эти данные будут использоваться для назначения сборке надлежащего строгого имени.

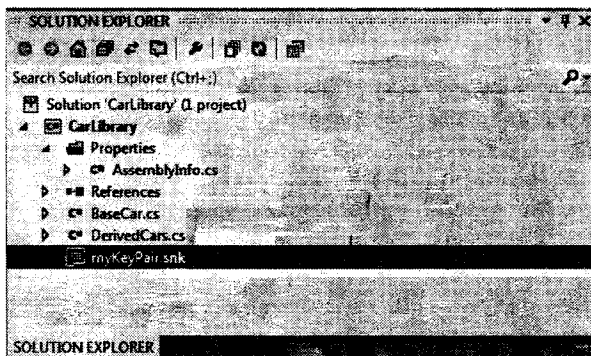


Рис. 14.19. Теперь при каждой компиляции Visual Studio будет назначать сборке строгое имя

На заметку! Вспомните, что на вкладке Application (Приложение) окна редактора свойств имеет-ся кнопка Assembly Information (Информация о сборке). Щелчок на ней приводит к открытию диалогового окна, которое позволяет устанавливать различные атрибуты уровня сборки, включая номер версии, информацию об авторских правах и т.д.

Установка строго именованных сборок в GAC

Последний шаг заключается в установке (теперь строго именованной) сборки CarLibrary.dll в GAC. Хотя предпочитаемым способом для развертывания сборок в GAC в производственной среде является создание установочного пакета Windows MSI (или применение коммерческой программы установки, такой как InstallShield), в составе .NET Framework 4.5 SDK поставляется работающая утилита командной строки gacutil.exe, которая удобна для проведения быстрых тестов.

На заметку! Для взаимодействия с GAC на своей машине необходимо иметь права администратора, что может требовать настройки параметров контроля учетных записей пользователей (UAC).

В табл. 14.1 перечислены некоторые наиболее важные опции этой утилиты (для вывода полного списка опций служит флаг `/?`).

Таблица 14.1. Опции, которые принимает утилита `gacutil.exe`

Опция	Описание
<code>-i</code>	Устанавливает сборку со строгим именем в GAC
<code>-u</code>	Удаляет сборку из GAC
<code>-l</code>	Отображает список сборок (или конкретную сборку) в GAC

Чтобы установить строго именованную сборку с помощью `gacutil.exe`, нужно открыть в окно командной строки разработчика и перейти в каталог, содержащий файл `CarLibrary.dll`, например:

```
cd C:\MyCode\CarLibrary\bin\Debug
```

Затем можно установить библиотеку, используя опцию `-i`:

```
gacutil -i CarLibrary.dll
```

После этого можно проверить, действительно ли библиотека была развернута, выполнив следующую команду (обратите внимание, что расширение файла в случае применения опции `/l` не указывается):

```
gacutil -l CarLibrary
```

Если все в порядке, в окне консоли должен появиться примерно такой вывод (разумеется, значение `PublicKeyToken` будет уникальным):

```
The Global Assembly Cache contains the following assemblies:
CarLibrary, Version=1.0.0.0, Culture=neutral, PublicKeyToken=33a2bc294331e8b9,
processorArchitecture=MSIL
```

Глобальный кеш сборок содержит следующие сборки:

```
CarLibrary, Version=1.0.0.0, Culture=neutral, PublicKeyToken=33a2bc294331e8b9,
processorArchitecture=MSIL
```

Кроме того, если перейти в каталог `C:\Windows\Microsoft.NET\assembly\GAC_MSIL`, в нем обнаружится новая папка `CarLibrary` с корректной структурой подкаталогов (рис. 14.20).

Использование разделяемой сборки

При построении приложений, использующих разделяемую сборку, единственным отличием от закрытой сборки является способ добавления ссылки на библиотеку в Visual Studio. Фактически для этого применяется тот же самый инструмент — диалоговое окно `Add Reference`.

Когда необходимо сослаться на закрытую сборку, можно было бы воспользоваться кнопкой `Browse` (Обзор) для перехода в необходимый подкаталог GAC. Однако можно также просто перейти к местоположению строго именованной сборки (такой как папка `\bin\debug` проекта библиотеки классов) и сослаться на копию. Когда среда Visual Studio находит строго именованную сборку, она не будет копировать библиотеку в выходной каталог клиентского приложения. В любом случае на рис. 14.21 показана ссылаемая библиотека.

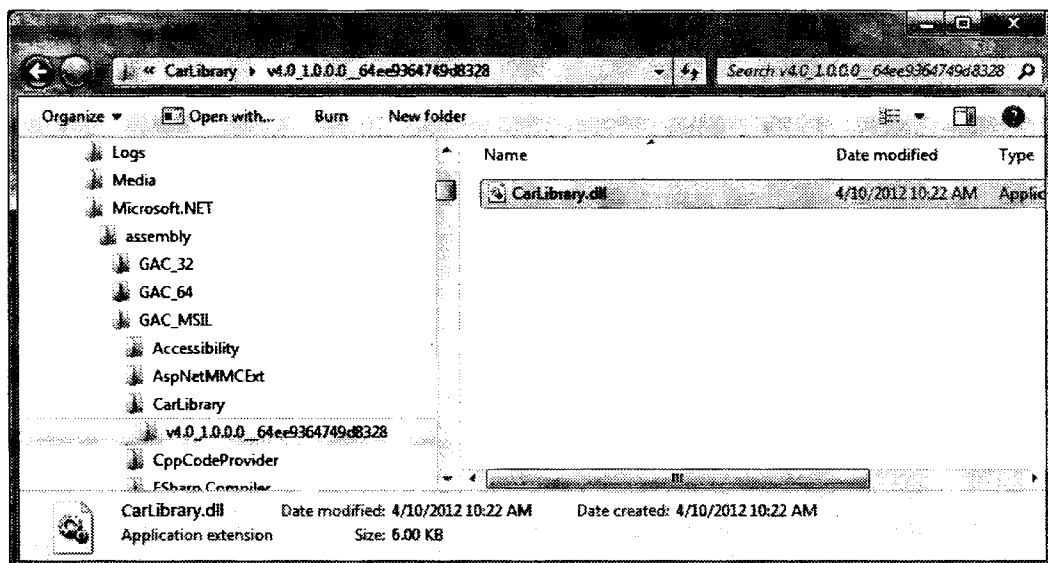


Рис. 14.20. Разделяемая сборка CarLibrary в GAC

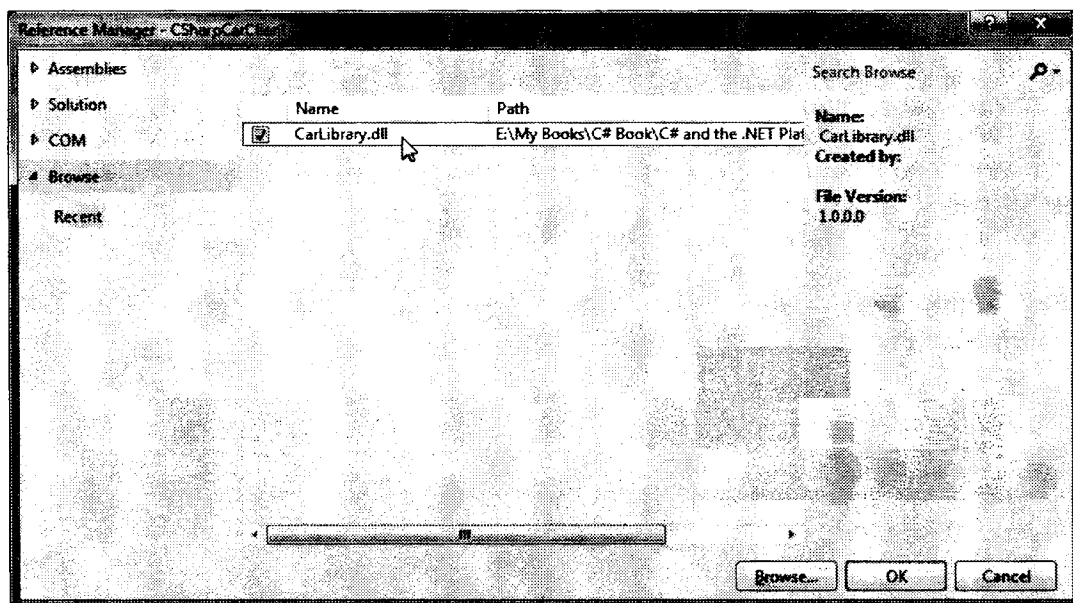


Рис. 14.21. Добавление ссылки на разделяемую сборку CarLibrary (версии 1.0.0.0) в Visual Studio

В целях иллюстрации создадим новое консольное приложение по имени SharedCarLibClient и добавим в него ссылку на сборку CarLibrary, как было только что описано. Как и следовало ожидать, после этого в папке References окна Solution Explorer появится соответствующий значок. Выбрав этот значок и открыв окно свойств (через меню View (Вид)), можно увидеть, что свойство Copy Local (Локальная копия) теперь установлено в False. Добавим в новое клиентское приложение следующий тестовый код:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using CarLibrary;
```

```
namespace SharedCarLibClient
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** Shared Assembly Client *****");
            SportsCar c = new SportsCar();
            c.TurboBoost();
            Console.ReadLine();
        }
    }
}
```

Если после компиляции этого клиентского приложения перейти с помощью проводника Windows в каталог, где находится файл SharedCarLibClient.exe, то можно будет заметить, что среда Visual Studio не скопировала CarLibrary.dll в каталог клиентского приложения. При добавлении ссылки на сборку, манифест которой содержит значение .publickey, Visual Studio считает, что эта строго именованная сборка, скорее всего, будет развертываться в GAC, и потому не заботится о копировании ее двоичного файла.

Исследование манифеста SharedCarLibClient

Вспомните, что при генерации строгого имени для сборки в ее манифест записывается полный открытый ключ. В связи с этим, когда клиент ссылается на строго именованную сборку, в ее манифест записывается компактное хеш-значение полного открытого ключа в дескрипторе .publickeytoken. Если вы откроете манифест SharedCarLibClient.exe в ildasm.exe, то увидите следующий код (разумеется, значение открытого ключа в маркере .publickeytoken может выглядеть по-другому):

```
.assembly extern CarLibrary
{
    .publickeytoken = (33 A2 BC 29 43 31 E8 B9 )
    .ver 1:0:0:0
}
```

Если сравнить значения маркера открытого ключа, записанного в манифесте клиента и отображаемого в GAC, то легко обнаружить, что они совпадают. Как упоминалось ранее, открытый ключ представляет один из аспектов идентичности строго именованной сборки. С учетом этого CLR-среда будет загружать только версию 1.0.0.0 сборки по имени CarLibrary, из открытого ключа которой может быть получено хеш-значение 33A2BC294331E8B9. Если CLR не удастся найти сборку, удовлетворяющую этому описанию, в GAC (и закрытую сборку по имени CarLibrary в каталоге клиента), генерируется исключение FileNotFoundException.

Исходный код. Проект SharedCarLibClient доступен в подкаталоге Chapter 14.

Конфигурирование разделяемых сборок

Подобно закрытым сборкам, разделяемые сборки можно конфигурировать с использованием клиентского файла *.config. Естественно, поскольку разделяемые сборки развертываются в хорошо известном месте (GAC), элемент <privatePath> для них не используется, как это делалось для закрытых сборок (хотя, если клиент работает как с разделяемыми, так и с закрытыми сборками, то элемент <privatePath> может присутствовать в файле *.config).

Конфигурационные файлы приложения вместе с разделяемыми сборками могут применяться, когда необходимо заставить CLR-среду привязаться к другой версии заданной сборки, пропуская значение, которое записано в манифесте клиента. Это может быть удобно по нескольким причинам. Например, предположим, что была выпущена версия 1.0.0.0 сборки, в которой через какое-то время обнаружился серьезный дефект. Одна из возможных мер предусматривает перекомпиляцию клиентского приложения для ссылки на корректную версию сборки, не содержащей дефекта (скажем, 1.1.0.0), и передачу приложения и новой библиотеки на все целевые машины.

Другой вариант состоит в поставке новой библиотеки кода и файла *.config, который автоматически указывает исполняющей среде привязываться к новой (свободной от дефектов) версии. После установки этой новой версии в GAC исходное клиентское приложение будет функционировать без перекомпиляции или повторного распространения.

Рассмотрим еще один пример. Предположим, что была поставлена первая версия свободной от дефектов сборки (1.0.0.0), а через пару месяцев к ней была добавлена новая функциональность, в результате чего получилась версия 2.0.0.0. Очевидно, что существующие клиентские приложения, которые компилировались с версией 1.0.0.0, не имеют никакого понятия о появившихся новых типах, т.к. их кодовая база не содержит ссылок на них.

Однако новым клиентским приложениям требуется ссылка на новую функциональность в версии 2.0.0.0. Платформа .NET позволяет поставить на целевые машины сборку версии 2.0.0.0 и обеспечить ее работу бок о бок со сборкой версии 1.0.0.0. При необходимости существующие клиенты могут динамически перенаправляться для загрузки версии 2.0.0.0 (чтобы получить доступ к реализованным в этой версии улучшениям) с использованием конфигурационного файла, не требуя повторной компиляции и развертывания клиентского приложения.

Фиксация текущей версии разделяемой сборки

В целях иллюстрации динамической привязки к конкретной версии разделяемой сборки откроем окно проводника Windows и скопируем текущую версию скомпилированной сборки CarLibrary.dll (1.0.0.0) в другой подкаталог (например, CarLibrary Version 1.0.0.0), чтобы симитировать фиксацию данной версии (рис. 14.22).

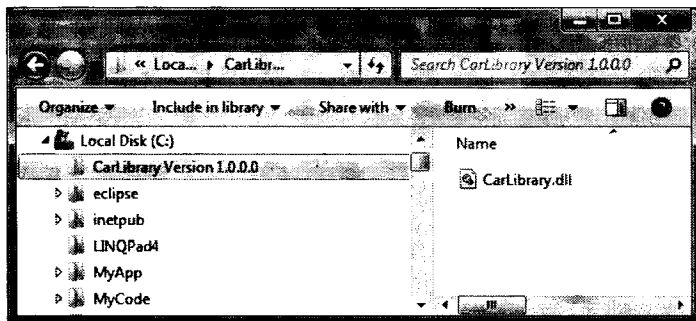


Рис. 14.22. Фиксация текущей версии CarLibrary.dll

Построение разделяемой сборки версии 2.0.0.0

Теперь откроем существующий проект CarLibrary и обновим кодовую базу, добавив новый тип enum по имени MusicMedia, который определяет четыре возможных музыкальных проигрывателя:

```
// Тип музыкального проигрывателя, установленный в автомобиле.
public enum MusicMedia
{
    musicCd,
    musicTape,
    musicRadio,
    musicMp3
}
```

Добавим в тип `Car` новый открытый метод, который позволяет вызывающему коду включать один из определенных музыкальных проигрывателей (не забыв при необходимости импортировать пространство имен `System.Windows.Forms`):

```
public abstract class Car
{
    ...
    public void TurnOnRadio(bool musicOn, MusicMedia mm)
    {
        if(musicOn)
            MessageBox.Show(string.Format("Jamming {0}", mm));
        else
            MessageBox.Show("Quiet time...");
    }
}
```

Теперь модифицируем конструкторы класса `Car` так, чтобы они отображали окно `MessageBox` с сообщением, подтверждающим использование версии 2.0.0.0 сборки `CarLibrary`:

```
public abstract class Car
{
    ...
    public Car()
    {
        MessageBox.Show("CarLibrary Version 2.0!");
    }
    public Car(string name, int maxSp, int currSp)
    {
        MessageBox.Show("CarLibrary Version 2.0!");
        PetName = name; MaxSpeed = maxSp; CurrentSpeed = currSp;
    }
    ...
}
```

И, наконец, перед перекомпиляцией новой библиотеки обновим номер версии с 1.0.0.0 на 2.0.0.0. Вспомните, что это можно сделать визуально, дважды щелкнув на значке **Properties** в окне **Solution Explorer**, а затем щелкнув на кнопке **Assembly Information** в области **Application** (Приложение). В отрывшемся диалоговом окне необходимо просто изменить значения в полях **Assembly Version** (Версия сборки), как показано на рис. 14.23.

Если теперь заглянуть в каталог `\bin\Debug` проекта, можно заметить, что в нем появилась новая версия этой сборки (2.0.0.0), а версия 1.0.0.0 безопасно сохранена в подкаталоге `CarLibrary Version 1.0.0.0`. Установим новую версию сборки в GAC для .NET 4.0 с помощью утилиты `gacutil.exe`, как было описано ранее в главе. Обратите внимание, что после этого на машине будут присутствовать две версии одной и той же сборки (рис. 14.24).

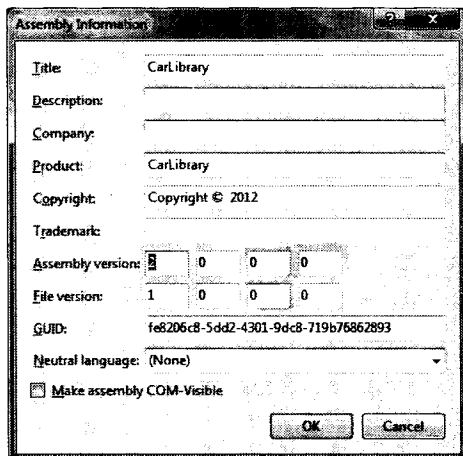


Рис. 14.23. Установка номера версии сборки CarLibrary.dll в 2.0.0.0

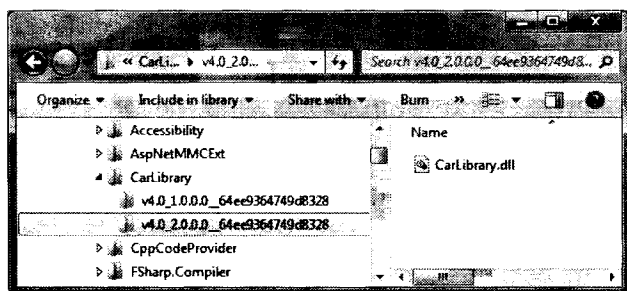


Рис. 14.24. Несколько версий одной и той же разделяемой сборки

После запуска приложения SharedCarLibClient.exe окно с сообщением CarLibrary Version 2.0! не отображается, поскольку в манифесте явным образом запрашивается версия 1.0.0.0. Как же тогда указать CLR-среде о необходимости привязки к версии 2.0.0.0? Ответ на этот вопрос ищите ниже.

На заметку! Среда Visual Studio автоматически сбрасывает ссылки при компиляции приложений!

Таким образом, когда вы запускаете приложение SharedCarLibClient.exe внутри IDE-среды Visual Studio, она автоматически захватывает версию 2.0.0.0. Если вы случайно запустили приложение именно таким образом, просто удалите текущую ссылку на CarLibrary.dll и выберите версию 1.0.0.0 (которая должна была быть размещена в папке CarLibrary Version 1.0.0.0).

Динамическое перенаправление на специфичные версии разделяемой сборки

Когда нужно сообщить CLR-среде о загрузке версии разделяемой сборки, отличной от указанной в манифесте, можно создать файл *.config, который содержит элемент <dependentAssembly>. Внутри этого элемента понадобится создать подэлемент <assemblyIdentity> с дружественным именем сборки, которое указано в манифесте клиента (CarLibrary в этом примере), и необязательным атрибутом культуры (которому можно присвоить пустую строку или вообще опустить, если должна использоваться

ся стандартная культура машины). Кроме того, элемент `<dependentAssembly>` будет содержать подэлемент `<bindingRedirect>` для определения версии, в текущий момент заданной в манифесте (в атрибуте `oldVersion`), и версии, которая должна загружаться вместо нее из GAC (в атрибуте `newVersion`). Модифицируем текущий конфигурационный файл в каталоге приложения `SharedCarLibClient` под названием `SharedCarLibClient.exe.config`, как показано ниже.

На заметку! Значение вашего маркера открытого ключа будет отличаться от того, который вы видите в приведенной далее разметке. Чтобы найти маркер открытого ключа, откройте клиентскую сборку в `ildasm.exe`, дважды щелкните на значке **MANIFEST** и скопируйте нужное значение в буфер (только не забудьте удалить пробелы).

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <!-- Информация о привязке во время выполнения -->
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="CarLibrary"
          publicKeyToken="64ee9364749d8328"
          culture="neutral"/>
        <bindingRedirect oldVersion="1.0.0.0"
          newVersion="2.0.0.0"/>
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

Теперь запустим программу `SharedCarLibClient.exe`. На этот раз должно появиться окно с сообщением о загрузке версии 2.0.0.0 сборки.

В конфигурационном файле клиента может присутствовать множество элементов `<dependentAssembly>`. Хотя в текущем примере в этом нет никакой необходимости, предположим, что манифест `SharedCarLibClient.exe` также ссылается на сборку `MathLibrary` версии 2.5.0.0. Если нужно перенаправить на версию 3.0.0.0 сборки `MathLibrary` (в дополнение к версии 2.0.0.0 сборки `CarLibrary`), содержимое конфигурационного файла `SharedCarLibClient.exe.config` могло бы выглядеть следующим образом:

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <!-- Управляет привязкой к CarLibrary -->
      <dependentAssembly>
        <assemblyIdentity name="CarLibrary"
          publicKeyToken="64ee9364749d8328"
          culture=""/>
        <bindingRedirect oldVersion="1.0.0.0" newVersion="2.0.0.0"/>
      </dependentAssembly>
      <!-- Управляет привязкой к MathLibrary -->
      <dependentAssembly>
        <assemblyIdentity name="MathLibrary"
          publicKeyToken="64ee9364749d8328"
          culture=""/>
        <bindingRedirect oldVersion="2.5.0.0" newVersion="3.0.0.0"/>
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```


На заметку! В атрибуте `oldVersion` допускается указывать диапазон номеров старых версий; например, `<bindingRedirect oldVersion="1.0.0.0-1.2.0.0" newVersion="2.0.0.0"/>` информирует CLR-среду о необходимости использования версии 2.0.0.0 вместо любой старой версии из диапазона от 1.0.0.0 до 1.2.0.0.

Понятие сборки политик издателя

Следующим моментом, связанным с конфигурацией, который мы рассмотрим, является роль *сборки политик издателя*. Как было только что показано, с помощью файлов `*.config` можно привязываться к специфической версии разделяемой сборки, таким образом, обходя версию, записанную в манифесте клиента. Хотя все это замечательно, представим, что вам, как администратору, требуется переконфигурировать все клиентские приложения на данной машине для привязки к версии 2.0.0.0 сборки `CarLibrary.dll`. Учитывая строгое соглашение об именовании конфигурационных файлов, придется дублировать одно и то же XML-содержимое во множестве мест (при условии, что вы действительно знаете местоположения исполняемых файлов, использующих `CarLibrary`). Понятно, что это будет настоящим кошмаром.

Политики издателя позволяют издателю конкретной сборки (в роли которого может выступать программист, подразделение или целая компания) поставлять двоичную версию файла `*.config`, предназначенную для установки в GAC вместе с более новой версией связанной с ним сборки. Преимущество такого подхода состоит в том, что каталоги клиентских приложений не должны содержать специфические файлы `*.config`. Вместо этого CLR-среда будет считывать текущий манифест и пытаться найти запрашиваемую версию сборки в GAC. Однако если CLR-среда обнаружит сборку политик издателя, она прочитает содержащиеся в ней XML-данные и выполнит запрашиваемое перенаправление на уровне GAC.

Сборки политик издателя создаются в командной строке с применением утилиты `.NET` под названием `al.exe` (редактор связей сборки). Эта утилита поддерживает множество опций, но построение сборки политик издателя требует передачи только следующих входных параметров:

- местоположение файла `*.config` или `*.xml`, содержащего инструкции перенаправления;
- имя результирующей сборки политик издателя;
- местоположение файла `*.snk`, используемого для подписания сборки политик издателя;
- номера версии для назначения создаваемой сборке политик издателя.

Чтобы построить сборку политик издателя, которая управляет библиотекой `CarLibrary.dll`, выполните следующую команду (должна вводиться в одной строке):

```
al /link: CarLibraryPolicy.xml /out:policy.1.0.CarLibrary.dll
/keyf:C:\MyKey\myKey.snk /v:1.0.0.0
```

Здесь указано, что необходимое XML-содержимое находится в файле по имени `CarLibraryPolicy.xml`. Имя выходного файла (которое должно соответствовать формату `policy.<старший номер>.<младший номер>.<имя КонфигурируемойСборки>`) задано с помощью флага `/out`. Кроме того, обратите внимание, что имя файла, содержащего пару открытого и секретного ключей, также должно быть указано посредством флага `/keyf`. Не забывайте, что файлы политик издателя являются разделяемыми и потому должны иметь строгие имена!

В результате запуска `al.exe` создается новая сборка, которая может быть помещена в GAC, чтобы вынудить всех клиентов привязаться к версии 2.0.0.0 сборки `CarLibrary.dll`, без использования специфичного конфигурационного файла для каждого клиентского приложения. При таком подходе можно спроектировать перенаправление в масштабах всей машины для всех приложений, работающих с конкретной версией (или диапазоном версий) существующей сборки.

Отключение политики издателя

Теперь предположим, что вы (как системный администратор) развернули сборку политик издателя (и последнюю версию связанной сборки) в GAC на клиентской машине. При этом, как обычно бывает, девять из десяти задействованных приложений переключились на версию 2.0.0.0 без проблем, а одно (по ряду причин) при получении доступа к `CarLibrary.dll` версии 2.0.0.0 постоянно выдает ошибку. (Как известно, создание программного обеспечения с полной обратной совместимостью, которое бы функционировало корректно в 100% случаев, практически невозможно.) В подобных случаях для проблемного клиента можно создать конфигурационный файл, указывающий CLR-среде *игнорировать* наличие любых файлов политик издателя, установленных в GAC. Остальные клиентские приложения, способные работать с новейшей версией сборки .NET, будут просто перенаправляться через установленную сборку политик издателя. Чтобы отключить политику издателя на уровне отдельного клиента, необходимо создать файл `*.config` (с соответствующим именем) и добавить в него элемент `<publisherPolicy>` с атрибутом `apply`, установленным в `no`. После этого CLR-среда будет загружать ту версию сборки, которая была изначально указана в манифесте клиента.

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <publisherPolicy apply="no" />
    </assemblyBinding>
  </runtime>
</configuration>
```

Элемент `<codeBase>`

В конфигурационных файлах приложений можно также указывать *кодовые базы* с помощью элемента `<codeBase>`. Этот элемент позволяет инструктировать CLR-среду о необходимости выполнять поиск зависимых сборок в произвольных местоположениях (таких как сетевые конечные точки или пути на машине за пределами каталога клиентского приложения).

Если в `<codeBase>` указано местоположение на удаленной машине, сборка будет загружаться по требованию в определенный каталог внутри GAC, который называется *кешем загрузки*. С учетом того, что уже известно о разворачивании сборок в GAC, должно быть ясно, что сборки, загружаемые из указанного в `<codeBase>` места, должны иметь строгие имена (иначе CLR-среда не смогла бы их установить в GAC). Просмотреть содержимое кеша загрузки на своей машине можно, запустив утилиту `gacutil.exe` с опцией `/ldl`:

```
gacutil /ldl
```

На заметку! Формально элемент `<codeBase>` может использоваться для поиска сборок, не обладающих строгим именем. В таком случае местоположение сборки должно быть относительным к каталогу клиентского приложения (и, следовательно, мы получаем всего лишь альтернативу элементу `<privatePath>`).

Чтобы увидеть элемент `<codeBase>` в действии, создадим новое консольное приложение по имени `CodeBaseClient`, добавим в него ссылку на `CarLibrary.dll` версии 2.0.0.0 и модифицируем начальный файл кода следующим образом:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using CarLibrary;

namespace CodeBaseClient
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** Fun with CodeBases *****");
            SportsCar c = new SportsCar();
            Console.WriteLine("Sports car has been allocated.");
            Console.ReadLine();
        }
    }
}
```

Из-за того, что сборка `CarLibrary.dll` была развернута в GAC, в принципе можно уже запускать приложение в таком, как оно есть виде. Чтобы поработать с элементом `<codeBase>`, создадим на диске C: новую папку (например, `C:\MyAsms`) и поместим в нее копию сборки `CarLibrary.dll` версии 2.0.0.0.

Теперь добавим в проект `CodeBaseClient` файл `App.config` (или отредактируем существующий), как объяснялось ранее в этой главе, поместив в него следующее XML-содержимое (не забывайте, что ваше значение `.publickeytoken` будет отличаться; при необходимости его можно просмотреть в GAC):

```
<configuration>
...
<runtime>
  <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
    <dependentAssembly>
      <assemblyIdentity name="CarLibrary" publicKeyToken="33A2BC294331E8B9" />
      <codeBase version="2.0.0.0" href="file:///C:/MyAsms/CarLibrary.dll" />
    </dependentAssembly>
  </assemblyBinding>
</runtime>
</configuration>
```

Как видите, элемент `<codeBase>` размещен внутри элемента `<assemblyIdentity>`, в атрибутах `name` и `publicKeyToken` которого указано ассоциируемое со сборкой дружественное имя и значение открытого ключа. В самом элементе `<codeBase>` сообщается версия и (в свойстве `href`) местоположение сборки, которая должна загружаться. Теперь даже если удалить версию 2.0.0.0 сборки `CarLibrary.dll` из GAC, это клиентское приложение все равно будет успешно выполняться, потому что CLR-среда способна найти необходимую внешнюю сборку в каталоге `C:\MyAsms`.

На заметку! Размещая сборки в произвольных местах на машине разработки, в сущности, вы воспроизводите системный реестр (и связанный с ним “ад DLL”), если учесть, что перемещение или переименование папки, содержащей двоичные файлы, приведет к разрушению текущей привязки. Поэтому элементом `<codeBase>` следует пользоваться с осторожностью.

Элемент `<codeBase>` может также быть полезным при добавлении ссылок на сборки, находящиеся на удаленной машине в сети. Например, предположим, что у вас есть права доступа к папке, расположенной на `http://www.MySite.com`. Для загрузки удаленного файла `*.dll` в кеш загрузки GAC на локальной машине элемент `<codeBase>` можно обновить следующим образом:

```
<codeBase version="2.0.0.0"
  href="http://www.MySite.com/Assemblies/CarLibrary.dll" />
```

Исходный код. Проект `CodeBaseClient` доступен в подкаталоге `Chapter 14`.

Пространство имен `System.Configuration`

До сих пор во всех показанных в этой главе файлах `*.config` применялись хорошо известные XML-элементы для указания CLR-среде, где следует искать внешние сборки. В дополнение к этим распознаваемым элементам конфигурационный файл клиента может также содержать специфичные для приложения данные, которые не имеют ничего общего с механизмом привязки. Учитывая это, не должно вызывать удивление наличие в составе .NET Framework пространства имен, позволяющего программно читать данные из конфигурационного файла клиента.

Пространство имен `System.Configuration` предоставляет небольшой набор типов, которые можно использовать для чтения специальных данных из файла `*.config` клиента. Эти специальные параметры должны содержаться внутри элемента `<appSettings>`. В свою очередь, элемент `<appSettings>` может содержать любое количество элементов `<add>`, которые определяют пары “ключ/значение” для получения программным образом.

В качестве примера предположим, что имеется файл `*.config`, предназначенный для консольного приложения `AppConfigReaderApp`, в котором определены два значения, специфичных для приложения:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
  </startup>

  <!-- Специальные параметры приложения -->
  <appSettings>
    <add key="TextColor" value="Green" />
    <add key="RepeatCount" value="8" />
  </appSettings>
</configuration>
```

Чтение этих значений с целью использования в клиентском приложении сводится просто к вызову метода `GetValue()` типа `System.Configuration.AppSettingsReader` на уровне экземпляра. Как показано в следующем коде, первый параметр `GetValue()` — это имя ключа в файле `*.config`, а второй — тип этого ключа (получаемый посредством операции `typeof`):

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using System.Configuration;
```

```

namespace AppConfigReaderApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** Reading <appSettings> Data *****\n");
            // Извлечь специальные данные из файла *.config.
            AppSettingsReader ar = new AppSettingsReader();
            int numbOfTimes = (int)ar.GetValue("RepeatCount", typeof(int));
            string textColor = (string)ar.GetValue("TextColor", typeof(string));

            Console.ForegroundColor =
                (ConsoleColor)Enum.Parse(typeof(ConsoleColor), textColor);

            // Вывести сообщение соответствующее количеству раз.
            for (int i = 0; i < numbOfTimes; i++)
                Console.WriteLine("Howdy!");
            Console.ReadLine();
        }
    }
}

```

Исходный код. Проект AppConfigReaderApp доступен в подкаталоге Chapter 14.

Документация по схеме конфигурационного файла

В этой главе вы узнали о роли XML-файлов конфигурации. Здесь основное внимание было сконцентрировано на нескольких параметрах, которые можно добавить к элементу `<runtime>` для управления тем, как CLR-среда будет искать необходимые внешние библиотеки. По мере дальнейшего чтения книги (и переходу к построению реального программного обеспечения) вы очень быстро заметите, что XML-файлы конфигурации применяются повсеместно.

И действительно, в платформе .NET файлы `*.config` используются в многочисленных API-интерфейсах. Например, в главе 25 вы увидите, что в инфраструктуре Windows Communication Foundation (WCF) конфигурационные файлы применяются для установки сложных настроек сети. Позже в этой книге, когда будет демонстрироваться разработка веб-приложений с помощью ASP.NET, вы заметите, что файл `web.config` содержит инструкции того же типа, что и файл `App.config` для настольных приложений.

Поскольку конфигурационный файл .NET может содержать большое количество инструкций, вы должны знать, что полная схема этого XML-файла документирована в справочной системе .NET. В частности, если поискать в справочной системе тему “Configuration File Schema for the .NET Framework” (“Схема конфигурационного файла для .NET Framework”), вы получите детальные объяснения каждого элемента (рис. 14.25).

Резюме

В этой главе была исследована роль библиотек классов .NET (файлов `*.dll` для .NET). Как было показано, библиотеки классов — это двоичные файлы .NET, которые содержат логику, предназначенную для многократного использования в разнообразных проектах.

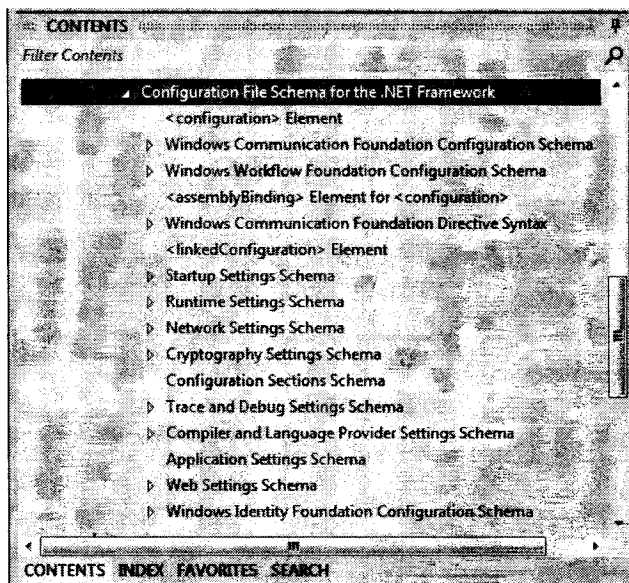


Рис. 14.25. Конфигурационные XML-файлы полностью документированы в справочной системе .NET

Вспомните, что библиотеки могут быть развернуты двумя основными путями — как закрытые или как разделяемые. Закрытые сборки развертываются в каталоге клиента или в его подкаталоге при условии, что имеется подходящий XML-файл конфигурации. Разделяемые сборки представляют собой библиотеки, которые могут использоваться любым приложением на машине, и на них также можно оказывать влияние через параметры в конфигурационном файле клиентской стороны.

Вы узнали, как назначать разделяемым сборкам строгие имена, с помощью которых устанавливается уникальная идентичность библиотек с точки зрения CLR. Кроме того, вы ознакомились с различными инструментами командной строки (`sn.exe` и `gacutil.exe`), применяемыми во время разработки и развертывания разделяемых библиотек.

Глава была завершена исследованием роли политик издателя и процесса сохранения и извлечения специальных параметров с использованием пространства имен `System.Configuration`.

Рефлексия типов, позднее связывание и программирование с использованием атрибутов

Как было показано в главе 14, сборки являются базовой единицей развертывания в мире .NET. Используя интегрированные браузеры объектов Visual Studio (и многих других IDE-сред), можно просматривать типы внутри набора сборок, на которые ссылается проект. Кроме того, внешние инструменты, такие как утилита ildasm.exe, позволяют анализировать лежащий в основе CIL-код, метаданные типов и манифест сборки для любого двоичного файла .NET. В дополнение к такому исследованию сборок .NET на этапе проектирования, ту же самую информацию можно получить *программно* с применением пространства имен System.Reflection. Таким образом, первой задачей этой главы является определение роли рефлексии и потребности в метаданных .NET.

В оставшейся части главы рассматривается несколько тесно связанных тем, так или иначе касающихся служб рефлексии. Например, вы узнаете, как клиент .NET может задействовать динамическую загрузку и позднее связывание для активизации типов, о которых на этапе компиляции ничего не известно. Кроме того, будет показано, как вставлять в сборки .NET специальные метаданные за счет использования системных и специальных атрибутов. Для практической демонстрации всех этих аспектов в конце главы приводится пример построения нескольких “объектов-оснасток”, которые можно подключать к расширяемому настольному приложению с графическим пользовательским интерфейсом.

Потребность в метаданных типов

Возможность полного описания типов (классов, интерфейсов, структур, перечислений и делегатов) с помощью метаданных является ключевым элементом платформы .NET. Для многочисленных технологий .NET, таких как Windows Communication Foundation (WCF) и сериализация объектов, требуется возможность выяснения формата типов во время выполнения. Более того, межъязыковое взаимодействие, многие службы компилятора и средства IntelliSense в IDE-среде опираются на конкретное описание *типа*.

Вспомните из главы 1, что утилита `ildasm.exe` позволяет просматривать метаданные типов сборки по нажатию комбинации клавиш `<Ctrl+M>`. Таким образом, если открыть в `ildasm.exe` любую из сборок `*.dll` или `*.exe`, которые создавались ранее в книге (например, `CarLibrary.dll` из главы 14), и нажать `<Ctrl+M>`, то можно увидеть метаданные типов (рис. 15.1).

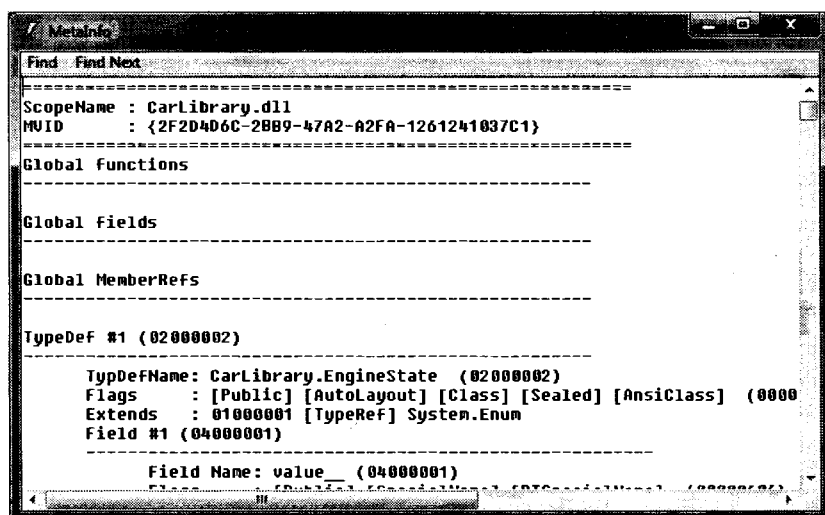


Рис. 15.1. Просмотр метаданных сборки с помощью утилиты `ildasm.exe`

Как видите, утилита `ildasm.exe` отображает метаданные типов .NET очень подробно (действительный двоичный формат гораздо компактнее). В действительности описание всех метаданных сборки `CarLibrary.dll` заняло бы несколько страниц. Однако вполне достаточно будет кратко взглянуть на некоторые ключевые описания метаданных сборки `CarLibrary.dll`.

На заметку! Не стоит слишком глубоко вникать в синтаксис каждого фрагмента метаданных .NET, приводимого в нескольких следующих разделах. Главное — понять, что метаданные .NET являются исключительно дескриптивными и учитывают каждый внутренне определенный (или внешне ссылаемый) тип, который найден в данной кодовой базе.

Просмотр (частичных) метаданных для перечисления `EngineState`

Каждый тип, определенный внутри текущей сборки, документируется с применением маркера `TypeDef #n` (где `TypeDef` — сокращение от *type definition* (определение типа)). Если описываемый тип использует какой-нибудь тип, определенный в отдельной сборке .NET, ссылаемый тип документируется с помощью маркера `TypeRef #n` (где `TypeRef` — сокращение от *type reference* (ссылка на тип)). Можно считать, что маркер `TypeRef` является указателем на полное определение метаданных ссылаемого типа во внешней сборке. В сущности, метаданные .NET — это набор таблиц, которые явно помечают все определения типов (`TypeDef`) и ссылаемые типы (`TypeRef`), причем все они могут быть просмотрены с использованием окна метаданных утилиты `ildasm.exe`.

В случае сборки `CarLibrary.dll` один из набора `TypeDef` представляет описание метаданных перечисления `CarLibrary.EngineState` (номер `TypeDef` может отличаться; нумерация `TypeDef` основана на порядке, в котором компилятор C# обрабатывает файл).


```
TypeDef #2 (02000003)
```

```
-----
TypDefName: CarLibrary.EngineState (02000003)
Flags      : [Public] [AutoLayout] [Class] [Sealed] [AnsiClass] (00000101)
Extends    : 01000001 [TypeRef] System.Enum
Field #1 (04000006)
```

```
-----
Field Name: value__ (04000006)
Flags      : [Public] [SpecialName] [RTSpecialName] (00000606)
CallCnvtn : [FIELD]
Field type: I4
```

```
Field #2 (04000007)
```

```
-----
Field Name: engineAlive (04000007)
Flags      : [Public] [Static] [Literal] [HasDefault] (00008056)
DefltValue: (I4) 0
CallCnvtn : [FIELD]
Field type: ValueClass CarLibrary.EngineState
```

```
...
```

Здесь маркер `TypDefName` служит для описания имени данного типа, которым в рассматриваемом случае является специальное перечисление `CarLibrary.EngineState`. Маркер метаданных `Extends` документирует базовый тип для данного типа (в этом случае ссылаемого типа `System.Enum`). Каждое поле перечисления помечается с применением маркера `Field #n`. Ради краткости выше были приведены только метаданные для поля `CarLibrary.EngineState.engineAlive`.

Просмотр (частичных) метаданных для типа `Car`

Ниже приведена часть метаданных класса `Car`, которая иллюстрирует следующие аспекты:

- как поля определены в терминах метаданных .NET;
- как методы документированы через метаданные .NET;
- как автоматическое свойство представлено в метаданных .NET.

```
TypeDef #3 (02000004)
```

```
-----
TypDefName: CarLibrary.Car (02000004)
Flags : [Public] [AutoLayout] [Class] [Abstract]
       [AnsiClass] [BeforeFieldInit] (00100081)
Extends : 01000002 [TypeRef] System.Object
```

```
...
```

```
Field #2 (0400000a)
```

```
-----
Field Name: <PetName>k__BackingField (0400000a)
Flags      : [Private] (00000001)
CallCnvtn : [FIELD]
Field type: String
```

```
...
```

```
Method #1 (06000001)
```

```
-----
MethodName: get_PetName (06000001)
Flags      : [Public] [HideBySig] [ReuseSlot] [SpecialName] (00000886)
RVA        : 0x000020d0
ImplFlags : [IL] [Managed] (00000000)
```

```

CallCnvntn: [DEFAULT]
hasThis
ReturnType: String
No arguments.
...
Method #2 (06000002)
-----
MethodName: set_PetName (06000002)
Flags      : [Public] [HideBySig] [ReuseSlot] [SpecialName] (00000886)
RVA        : 0x000020e7
ImplFlags  : [IL] [Managed] (00000000)
CallCnvntn: [DEFAULT]
hasThis
ReturnType: Void
1 Arguments
  Argument #1: String
1 Parameters
  (1) ParamToken : (08000001) Name : value flags: [none] (00000000)
...
Property #1 (17000001)
-----
Prop.Name : PetName (17000001)
Flags      : [none] (00000000)
CallCnvntn: [PROPERTY]
hasThis
ReturnType: String
No arguments.
DefltValue:
Setter      : (06000002) set_PetName
Getter      : (06000001) get_PetName
0 Others
...

```

Прежде всего, важно отметить, что метаданные класса `Car` указывают базовый класс этого типа (`System.Object`) и включают различные флаги, которые описывают то, как тип был сконструирован (`[Public]`, `[Abstract]` и т.д.). Описания методов (таких как конструктор `Car`) содержат имя, принимаемые параметры и возвращаемое значение.

Обратите внимание, что автоматическое свойство дает в результате сгенерированное компилятором закрытое поддерживающее поле (по имени `<PetName>k__BackingField`) и два сгенерированных компилятором метода (в случае свойства, доступного для чтения и записи), которые в рассматриваемом примере имеют имена `get_PetName()` и `set_PetName()`. И, наконец, само свойство отображается на внутренние методы `get/set` с использованием маркеров `Setter` и `Getter` метаданных .NET.

Исследование блока `TypeRef`

Вспомните, что метаданные сборки будут описывать не только набор внутренних типов (`Car`, `EngineState` и т.д.), но также любые внешние типы, на которые ссылаются внутренние типы. Например, с учетом того, что в сборке `CarLibrary.dll` определены два перечисления, метаданные типа `System.Enum` будут содержать следующий блок `TypeRef`:

```

TypeRef #1 (01000001)
-----
Token:          0x01000001
ResolutionScope: 0x23000001
TypeRefName:    System.Enum

```

Документирование определяемой сборки

Окно метаданных утилиты ildasm.exe также позволяет просматривать метаданные .NET, которые описывают саму сборку с использованием маркера Assembly. Как показано в приведенном ниже (неполном) листинге, информация, документируемая внутри таблицы Assembly, совпадает с той, которую можно просматривать через значок MANIFEST (Манифест). Ниже представлена часть метаданных манифеста сборки CarLibrary.dll (версии 2.0.0.0):

```
Assembly
-----
Token: 0x20000001
Name : CarLibrary
Public Key   : 00 24 00 00 04 80 00 00 // И т.д...

Hash Algorithm : 0x00008004
Major Version: 0x00000002
Minor Version: 0x00000000
Build Number: 0x00000000
Revision Number: 0x00000000
Locale: <null>
Flags : [PublicKey] ...
```

Документирование ссылаемых сборок

В дополнение к маркеру Assembly и набору блоков TypeDef и TypeRef, в метаданных .NET могут также применяться маркеры AssemblyRef #n для документирования каждой внешней сборки. Поскольку в сборке CarLibrary.dll используется класс System.Windows.Forms.MessageBox, в метаданных для сборки System.Windows.Forms будет присутствовать следующий блок AssemblyRef:

```
AssemblyRef #2 (23000002)
-----
Token: 0x23000002
Public Key or Token: b7 7a 5c 56 19 34 e0 89
Name: System.Windows.Forms
Version: 4.0.0.0
Major Version: 0x00000004
Minor Version: 0x00000000
Build Number: 0x00000000
Revision Number: 0x00000000
Locale: <null>
HashValue Blob:
Flags: [none] (00000000)
```

Документирование строковых литералов

Последний интересный аспект, касающийся метаданных .NET, заключается в том, что все строковые литералы в кодовой базе документируются внутри маркера User Strings:

```
User Strings
-----
70000001 : (11) L"Jamming {0}"
70000019 : (13) L"Quiet time..."
70000035 : (23) L"CarLibrary Version 2.0!"
70000065 : (14) L"Ramming speed!"
70000083 : (19) L"Faster is better..."
700000ab : (16) L"Time to call AAA"
700000cd : (16) L"Your car is dead"
```

На заметку! Как показано в последнем листинге метаданных, все строки четко документируются в метаданных сборки. Это может привести к серьезным последствиям в плане безопасности, если строковые литералы используются для хранения паролей, номеров кредитных карт и другой секретной информации.

Далее может возникнуть следующий вопрос: “Как использовать эту информацию в разрабатываемых приложениях?” (при лучшем сценарии) или “Зачем вообще нужны метаданные?” (при худшем сценарии). Чтобы получить ответ, необходимо ознакомиться со службами рефлексии .NET. Имейте в виду, что польза от рассматриваемых ниже средств может проявиться только к концу главы, поэтому наберитесь терпения.

На заметку! В окне MetalInfo утилиты ildasm.exe вы также найдете несколько маркеров CustomAttribute, которые документируют атрибуты, применяемые внутри кодовой базы. Роль этих атрибутов описана далее в главе.

Понятие рефлексии

В мире .NET *рефлексией* называется процесс обнаружения типов во время выполнения. С помощью служб рефлексии появляется возможность получать программно ту же самую информацию о метаданных, которую отображает утилита ildasm.exe, используя удобную объектную модель. Например, рефлексия позволяет извлечь список всех типов, которые содержатся внутри заданной сборки *.dll или *.exe, включая методы, поля, свойства и события, определенные конкретным типом. Можно также динамически выяснять набор интерфейсов, поддерживаемых заданным типом, параметры метода и другие детали подобного рода (базовые классы, информация о пространствах имен, данные манифеста и т.д.).

Подобно любому пространству имен, System.Reflection (которое определено в сборке mscorlib.dll) содержит набор связанных типов. В табл. 15.1 описан ряд ключевых элементов этого набора, которые вы должны знать.

Таблица 15.1. Некоторые члены пространства имен System.Reflection

Тип	Описание
Assembly	Этот абстрактный класс содержит статические методы, которые позволяют загружать, исследовать и манипулировать сборкой
AssemblyName	Этот класс позволяет выяснить многочисленные детали, связанные с идентичностью сборки (номер версии, информация о культуре и т.д.)
EventInfo	Этот абстрактный класс хранит информацию о заданном событии
FieldInfo	Этот абстрактный класс хранит информацию о заданном поле
MemberInfo	Этот абстрактный базовый класс определяет общее поведение для типов EventInfo, FieldInfo, MethodInfo и PropertyInfo
MethodInfo	Этот абстрактный класс содержит информацию о заданном методе
Module	Этот абстрактный класс позволяет получить доступ к определенному модулю внутри многофайловой сборки
ParameterInfo	Этот класс хранит информацию о заданном параметре
PropertyInfo	Этот абстрактный класс хранит информацию о заданном свойстве

Чтобы понять, как пользоваться пространством имен `System.Reflection` для программного чтения метаданных .NET, необходимо сначала ознакомиться с классом `System.Type`.

Класс `System.Type`

Класс `System.Type` определяет набор членов, которые могут применяться для исследования метаданных типа, при этом многие члены возвращают типы из пространства имен `System.Reflection`. Например, `Type.GetMethods()` возвращает массив объектов `MethodInfo`, член `Type.GetFields()` — массив объектов `FieldInfo` и т.д.

В табл. 15.2 приведен неполный список членов, поддерживаемых `System.Type` (полные сведения можно найти в документации .NET Framework 4.5 SDK).

Таблица 15.2. Некоторые члены `System.Type`

Член	Описание
<code>IsAbstract</code> <code>IsArray</code> <code>IsClass</code> <code>IsCOMObject</code> <code>IsEnum</code> <code>IsGenericTypeDefinition</code> <code>IsGenericParameter</code> <code>IsInterface</code> <code>IsPrimitive</code> <code>IsNestedPrivate</code> <code>IsNestedPublic</code> <code>IsSealed</code> <code>IsValueType</code>	Эти свойства позволяют выяснять основные детали об интересующем типе (например, является ли он абстрактной сущностью, массивом, вложенным классом и т.д.)
<code>GetConstructors()</code> <code>GetEvents()</code> <code>GetFields()</code> <code>GetInterfaces()</code> <code>GetMembers()</code> <code>GetMethods()</code> <code>GetNestedTypes()</code> <code>GetProperties()</code>	Эти методы позволяют получать массив представляющих интерес элементов (интерфейсов, методов, свойств и т.д.). Каждый метод возвращает соответствующий массив (например, <code>GetFields()</code> возвращает массив <code>FieldInfo</code> , метод <code>GetMethods()</code> — массив <code>MethodInfo</code> и т.д.). Обратите внимание, что каждый метод имеет также форму единственного числа (например, <code>GetMethod()</code> , <code>GetProperty()</code> и т.п.), которая позволяет извлекать специфический элемент по имени вместо целого массива связанных элементов
<code>FindMembers()</code>	Этот метод возвращает массив объектов <code>MemberInfo</code> на основе заданного критерия поиска
<code>GetType()</code>	Этот статический метод возвращает экземпляр <code>Type</code> с заданным строковым именем
<code>InvokeMember()</code>	Этот метод позволяет выполнять “позднее связывание” для указанного элемента. Вы узнаете о позднем связывании далее в этой главе

Получение информации о типе с помощью `System.Object.GetType()`

Экземпляр класса `Type` можно получить несколькими способами. Единственное, что нельзя делать — это напрямую создавать объект `Type`, используя ключевое слово `new`, т.к. `Type` является абстрактным классом. Что касается первого способа, то вспомните,

что в `System.Object` определен метод `GetType()`, который возвращает экземпляр класса `Type`, представляющий метаданные текущего объекта:

```
// Получить информацию о типе с использованием экземпляра SportsCar.
SportsCar sc = new SportsCar();
Type t = sc.GetType();
```

Очевидно, такой подход будет работать только при условии, что подвергаемый рефлексии тип (`SportsCar` в данном случае) известен на этапе компиляции и в памяти имеется его экземпляр. С учетом этого ограничения должно быть понятно, почему инструменты вроде `ildasm.exe` не получают информацию о типах прямым вызовом `System.Object.GetType()` для каждого типа; ведь утилита `ildasm.exe` не компилировалась вместе с вашими специальными сборками.

Получение информации о типе с помощью `typeof()`

Следующий способ для получения информации о типе заключается в применении операции `typeof`:

```
// Получить информацию о типе с использованием операции typeof.
Type t = typeof(SportsCar);
```

В отличие от метода `System.Object.GetType()`, операция `typeof` удобна тем, что она не требует предварительного создания экземпляра объекта перед получением информации о типе. Тем не менее, кодовой базе по-прежнему должно быть известно о представляющем интерес типе на этапе компиляции, поскольку `typeof` ожидает получения строго типизированного имени типа.

Получение информации о типе с помощью `System.Type.GetType()`

Для получения информации о типе более гибким образом можно вызвать статический метод `GetType()` класса `System.Type` и указать полностью заданное строковое имя типа, который требуется изучить. При таком подходе знать тип, из которого будут извлекаться метаданные, на этапе компиляции не требуется, поскольку `Type.GetType()` принимает в качестве параметра экземпляр вездесущего класса `System.String`.

На заметку! Когда речь идет о том, что при вызове метода `Type.GetType()` знать тип на этапе компиляции не нужно, имеется в виду тот факт, что этот метод может принимать любое строковое значение (а не строго типизированную переменную). Естественно, знать имя типа в строковом формате по-прежнему необходимо!

Метод `Type.GetType()` перегружен для приема двух булевских параметров, один из которых управляет тем, должно ли генерироваться исключение, когда тип не удается найти, а второй отвечает за то, должен ли учитываться регистр символов в строке.

В целях иллюстрации рассмотрим следующий код:

```
// Получить информацию о типе с использованием статического метода Type.GetType()
// (не генерировать исключение, если SportsCar не может быть найден
// и игнорировать регистр).
Type t = Type.GetType("CarLibrary.SportsCar", false, true);
```

Обратите внимание в приведенном примере на то, что в строке, передаваемой методу `GetType()`, никак не упоминается сборка, внутри которой содержится интересующий тип. В этом случае подразумевается, что тип определен внутри сборки, выполняемой в текущий момент. Однако когда необходимо получить метаданные для типа, находящегося во внешней закрытой сборке, строковый параметр должен быть сформатирован с

использованием полностью заданного имени типа, за которым следует символ запятой и дружественное имя сборки, содержащей этот тип:

```
// Получить информацию о типе внутри внешней сборки.
Type t = Type.GetType("CarLibrary.SportsCar, CarLibrary");
```

Кроме того, в строке, передаваемой `GetType()`, может быть указан знак плюс (+) для обозначения *вложенного типа*. Например, предположим, что необходимо получить информацию о типе перечисления (`SpyOptions`), который вложен в класс по имени `JamesBondCar`. Для этого можно воспользоваться следующим кодом:

```
// Получить информацию о типе перечисления, вложенного в текущую сборку.
Type t = Type.GetType("CarLibrary.JamesBondCar+SpyOptions");
```

Построение специального средства для просмотра метаданных

Для демонстрации базового процесса рефлексии (и пользы от `System.Type`) давайте создадим консольное приложение по имени `MyTypeViewer`. Это приложение будет отображать детали методов, свойств, полей и поддерживаемых интерфейсов (а также другие интересные данные) для любого типа внутри сборки `mscorlib.dll` (вспомните, что все приложения .NET получают автоматически доступ к этой ключевой библиотеке классов платформы) или типа внутри самого приложения `MyTypeViewer`. После создания приложения не забудьте импортировать пространство имен `System.Reflection`:

```
// Это пространство имен должно импортироваться для выполнения любой рефлексии!
using System.Reflection;
```

Рефлексия методов

Далее потребуется модифицировать класс `Program` для определения в нем нескольких статических методов, каждый из которых принимает единственный параметр `System.Type` и возвращает `void`. Сначала определим метод `ListMethods()`, который выводит имена методов, определенных во входном типе. Обратите внимание, что `Type.GetMethods()` возвращает массив объектов `System.Reflection.MethodInfo`, по которому можно осуществлять проход с помощью цикла `foreach`:

```
// Отобразить имена методов типа.
static void ListMethods(Type t)
{
    Console.WriteLine("***** Methods *****");
    MethodInfo[] mi = t.GetMethods();
    foreach(MethodInfo m in mi)
        Console.WriteLine("->{0}", m.Name);
    Console.WriteLine();
}
```

Здесь просто выводится имя метода с использованием свойства `MethodInfo.Name`. Как не трудно догадаться, класс `MethodInfo` имеет множество дополнительных членов, которые позволяют выяснить, является метод статическим, виртуальным, обобщенным или абстрактным. Кроме того, тип `MethodInfo` позволяет получить информацию о возвращаемом значении и наборе параметров метода. Чуть позже реализация `ListMethods()` будет несколько улучшена.

Для перечисления имен методов при желании можно было бы также построить LINQ-запрос. Вспомните из главы 12, что технология LINQ to Object позволяет создавать строго типизированные запросы и применять их к коллекциям объектов в памяти.

При обнаружении блоков с циклами или программной логикой принятия решения хорошим правилом считается использование соответствующего LINQ-запроса. Например, предыдущий метод можно было бы переписать следующим образом:

```
static void ListMethods(Type t)
{
    Console.WriteLine("***** Methods *****");
    var methodNames = from n in t.GetMethods() select n.Name;
    foreach (var name in methodNames)
        Console.WriteLine("->{0}", name);
    Console.WriteLine();
}
```

Рефлексия полей и свойств

Реализация метода `ListFields()` будет похожей. Единственным заметным отличием является вызов `Type.GetFields()` и результирующий массив `FieldInfo`. Здесь также для простоты выводятся только имена каждого из полей с применением LINQ-запроса:

```
// Отобразить имена полей типа.
static void ListFields(Type t)
{
    Console.WriteLine("***** Fields *****");
    var fieldNames = from f in t.GetFields() select f.Name;
    foreach (var name in fieldNames)
        Console.WriteLine("->{0}", name);
    Console.WriteLine();
}
```

Логика для отображения имен свойств типа выглядит аналогично:

```
// Отобразить имена свойств типа.
static void ListProps(Type t)
{
    Console.WriteLine("***** Properties *****");
    var propNames = from p in t.GetProperties() select p.Name;
    foreach (var name in propNames)
        Console.WriteLine("->{0}", name);
    Console.WriteLine();
}
```

Рефлексия реализованных интерфейсов

Следующим создается метод по имени `ListInterfaces()`, который будет выводить имена любых интерфейсов, поддерживаемых входным типом. Единственный интересный момент здесь в том, что вызов `GetInterfaces()` возвращает массив объектов `System.Type!` Это вполне логично, поскольку интерфейсы в конечном итоге являются типами.

```
// Отобразить имена реализованных интерфейсов.
static void ListInterfaces(Type t)
{
    Console.WriteLine("***** Interfaces *****");
    var ifaces = from i in t.GetInterfaces() select i;
    foreach (Type i in ifaces)
        Console.WriteLine("->{0}", i.Name);
}
```


На заметку! Следует иметь в виду, что большинство методов "get" в `System.Type` (`GetMethods()`, `GetInterfaces()` и т.д.) перегружены для приема значений из перечисления `BindingFlags`. Это предоставляет высокий уровень контроля над тем, что необходимо искать (например, только статические члены, только открытые члены, включать закрытые члены и т.д.). За более подробной информацией по этому поводу обращайтесь к документации .NET Framework 4.5 SDK.

Отображение различных дополнительных деталей

И, наконец, реализуем еще последний вспомогательный метод, который будет просто отображать различные статистические данные о входном типе (является ли тип обобщенным, какой его базовый класс, является ли тип запечатанным, и т.д.).

```
// Просто для полноты картины.
static void ListVariousStats(Type t)
{
    Console.WriteLine("***** Various Statistics *****");
    Console.WriteLine("Base class is: {0}", t.BaseType);           // Базовый класс
    Console.WriteLine("Is type abstract? {0}", t.IsAbstract);     // Абстрактный?
    Console.WriteLine("Is type sealed? {0}", t.IsSealed);         // Запечатанный?
    Console.WriteLine("Is type generic? {0}", t.IsGenericTypeDefinition); // Обобщенный?
    Console.WriteLine("Is type a class type? {0}", t.IsClass);     // Класс?
    Console.WriteLine();
}
```

Реализация метода Main()

Метод `Main()` класса `Program` запрашивает у пользователя полностью заданное имя типа. После получения этих строковых данных они передаются методу `Type.GetType()`, а результирующий объект `System.Type` отправляется каждому из вспомогательных методов. Этот процесс повторяется до тех пор, пока пользователь не введет `Q` для завершения работы приложения.

```
static void Main(string[] args)
{
    Console.WriteLine("***** Welcome to MyTypeViewer *****");
    string typeName = "";

    do
    {
        Console.WriteLine("\nEnter a type name to evaluate");
        // Запросить ввод имени типа.
        Console.Write("or enter Q to quit: ");

        // Получить имя типа.
        typeName = Console.ReadLine();

        // Пользователь желает завершить программу?
        if (typeName.ToUpper() == "Q")
        {
            break;
        }

        // Попробовать отобразить информацию о типе.
        try
        {
            Type t = Type.GetType(typeName);
            Console.WriteLine("");
            ListVariousStats(t);
        }
    }
}
```

```
ListFields(t);
ListProps(t);
ListMethods(t);
ListInterfaces(t);
}
catch
{
    Console.WriteLine("Sorry, can't find type"); // Тип не найден.
}
} while (true);
}
```

В этот момент приложение `MyTypeViewer.exe` готово к первому тестовому запуску. Давайте запустим его и введем следующие полностью заданные имена (не забывая, что выбранный способ вызова `Type.GetType()` требует ввода строковых имен с учетом регистра):

- `System.Int32`
- `System.Collections.ArrayList`
- `System.Threading.Thread`
- `System.Void`
- `System.IO.BinaryWriter`
- `System.Math`
- `System.Console`
- `MyTypeViewer.Program`

Ниже показан частичный вывод при указании типа `System.Math`:

```
***** Welcome to MyTypeViewer *****
Enter a type name to evaluate
or enter Q to quit: System.Math
***** Various Statistics *****
Base class is: System.Object
Is type abstract? True
Is type sealed? True
Is type generic? False
Is type a class type? True
***** Fields *****
->PI
->E
***** Properties *****
***** Methods *****
->Acos
->Asin
->Atan
->Atan2
->Ceiling
->Ceiling
->Cos
...
```

Рефлексия обобщенных типов

При вызове `Type.GetType()` для получения описаний метаданных обобщенных типов должен применяться специальный синтаксис в виде символа обратной одинарной кавычки (```) со следующим за ним числовым значением, которое представляет количество параметров типов, поддерживаемое данным типом. Например, чтобы вывести описание метаданных обобщенного типа `System.Collections.Generic.List<T>`, приложению потребуется передать следующую строку:

```
System.Collections.Generic.List`1
```

Здесь используется числовое значение 1, поскольку `List<T>` имеет только один параметр типа. Однако для применения рефлексии к типу `Dictionary<TKey, TValue>` пришлось бы указать значение 2:

```
System.Collections.Generic.Dictionary`2
```

Рефлексия параметров и возвращаемых значений методов

Давайте сделаем небольшое улучшение приложения, а именно — модифицируем вспомогательную функцию `ListMethods()` так, чтобы она выводила не только имя заданного метода, но также возвращаемый тип и типы входных параметров. Для решения этих задач в типе `MethodInfo` предусмотрено свойство `ReturnType` и метод `GetParameters()`. Обратите внимание в следующем измененном коде, что строка с информацией о типе и имени каждого параметра строится с использованием вложенного цикла `foreach` (а не LINQ-запроса):

```
static void ListMethods(Type t)
{
    Console.WriteLine("***** Methods *****");
    MethodInfo[] mi = t.GetMethods();
    foreach (MethodInfo m in mi)
    {
        // Получить информацию о возвращаемом типе.
        string retVal = m.ReturnType.FullName;
        string paramInfo = " ( ";

        // Получить информацию о параметрах.
        foreach (ParameterInfo pi in m.GetParameters())
        {
            paramInfo += string.Format("{0} {1} ", pi.ParameterType, pi.Name);
        }
        paramInfo += " )";

        // Отобразить базовую сигнатуру метода.
        Console.WriteLine("->{0} {1} {2}", retVal, m.Name, paramInfo);
    }
    Console.WriteLine();
}
```

Если теперь запустить обновленное приложение, то обнаружится, что методы заданного типа будут описаны более подробно. Например, в случае ввода типа `System.Object` отобразятся следующие описания методов:

```
***** Methods *****
->System.String ToString ( )
->System.Boolean Equals ( System.Object obj )
->System.Boolean Equals ( System.Object objA System.Object objB )
->System.Boolean ReferenceEquals ( System.Object objA System.Object objB )
->System.Int32 GetHashCode ( )
->System.Type GetType ( )
```

Текущая реализация `ListMethods()` удобна тем, что позволяет исследовать непосредственно каждый параметр и возвращаемый тип методов с применением объектной модели `System.Reflection`. Следует иметь в виду, что каждый из типов `XXXInfo` (`MethodInfo`, `PropertyInfo`, `EventInfo` и т.д.) имеет переопределенную версию метода `ToString()`, которая отображает сигнатуру запрашиваемого элемента. Следовательно, метод `ListMethods()` можно было бы реализовать и так, как показано ниже (с использованием LINQ-запроса, выбирающего все объекты `MethodInfo`, а не только значения `Name`):

```
static void ListMethods(Type t)
{
    Console.WriteLine("***** Methods *****");
    var methodNames = from n in t.GetMethods() select n;
    foreach (var name in methodNames)
        Console.WriteLine("->{0}", name);
    Console.WriteLine();
}
```

Довольно интересно, не так ли? Очевидно, что пространство имен `System.Reflection` и класс `System.Type` позволяют выполнять рефлексия многих других аспектов типа помимо тех, которые в настоящий момент отображает приложение `MyTypeViewer`. Как и можно было ожидать, можно также получать информацию о событиях, поддерживаемых типом, любых обобщенных параметрах для заданных членов и десятки других деталей.

Итак, к этому моменту построен довольно мощный браузер объектов. Его основное ограничение состоит в том, что он не позволяет подвергать рефлексии ничего кроме текущей сборки (`MyTypeViewer`) и всегда доступной сборки `mscorlib.dll`. Возникает вопрос: как создать приложение, способное загружать (и выполнять рефлексия) сборки, о которых на этапе компиляции ничего не известно? Об этом пойдет речь в следующем разделе.

Исходный код. Проект `MyTypeViewer` доступен в подкаталоге Chapter 15.

Динамически загружаемые сборки

В главе 14 вы узнали все о том, что CLR-среда заглядывает в манифест сборки, когда ищет внешние сборки, на которые ссылается текущая сборка. Однако во многих случаях необходимо загружать сборки на лету программным образом, даже если в манифесте о них не упоминается. Формально процесс загрузки внешних сборок по требованию называется *динамической загрузкой*.

В пространстве имен `System.Reflection` определен класс `Assembly`, с применением которого можно динамически загружать сборку, а также исследовать связанные с ней свойства. С помощью `Assembly` можно динамически загружать закрытые или разделяемые сборки, расположенные в произвольных местах. В сущности, класс `Assembly` предоставляет методы (например, `Load()` и `LoadFrom()`), которые позволяют программно поставлять ту же информацию, которая встречается в клиентских файлах `*.config`.

Для демонстрации динамической загрузки создадим новое консольное приложение по имени `ExternalAssemblyReflector`. Задача заключается в построении метода `Main()`, который запрашивает у пользователя дружественное имя сборки для ее динамической загрузки. Ссылка на `Assembly` будет передана вспомогательному методу под названием `DisplayTypes()`, который просто выведет имена всех содержащихся в сборке классов, интерфейсов, структур, перечислений и делегатов.

Необходимый код на удивление прост:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using System.Reflection;
using System.IO; // Для определения FileNotFoundException.

namespace ExternalAssemblyReflector
{
    class Program
    {
        static void DisplayTypesInAsm(Assembly asm)
        {
            Console.WriteLine("\n***** Types in Assembly *****");
            Console.WriteLine("->{0}", asm.FullName);
            Type[] types = asm.GetTypes();
            foreach (Type t in types)
                Console.WriteLine("Type: {0}", t);
            Console.WriteLine("");
        }

        static void Main(string[] args)
        {
            Console.WriteLine("***** External Assembly Viewer *****");
            string asmName = "";
            Assembly asm = null;

            do
            {
                Console.WriteLine("\nEnter an assembly to evaluate");
                // Запросить имя сборки.
                Console.Write("or enter Q to quit: ");

                // Получить имя сборки.
                asmName = Console.ReadLine();

                // Пользователь желает завершить программу?
                if (asmName.ToUpper() == "Q")
                {
                    break;
                }

                // Попробовать загрузить сборку.
                try
                {
                    asm = Assembly.Load(asmName);
                    DisplayTypesInAsm(asm);
                }
                catch
                {
                    Console.WriteLine("Sorry, can't find assembly."); // Сборка не найдена.
                }
            } while (true);
        }
    }
}
```

Обратите внимание, что статическому методу `Assembly.Load()` передается только дружественное имя сборки, которую требуется загрузить в память.

Таким образом, чтобы подвергнуть рефлексии сборку `CarLibrary.dll`, понадобится скопировать двоичный файл `CarLibrary.dll` в подкаталог `bin\Debug` внутри каталога приложения `ExternalAssemblyReflector`. После этого можно будет получить примерно такой вывод:

```
***** External Assembly Viewer *****
Enter an assembly to evaluate
or enter Q to quit: CarLibrary

***** Types in Assembly *****
->CarLibrary, Version=2.0.0.0, Culture=neutral, PublicKeyToken=33a2bc294331e8b9
Type: CarLibrary.MusicMedia
Type: CarLibrary.EngineState
Type: CarLibrary.Car
Type: CarLibrary.SportsCar
Type: CarLibrary.MiniVan
```

Если необходимо сделать приложение `ExternalAssemblyReflector` более гибким, можно модифицировать код так, чтобы загрузка внешней сборки производилась с помощью метода `Assembly.LoadFrom()`, а не `Assembly.Load()`:

```
try
{
    asm = Assembly.LoadFrom(asmName);
    DisplayTypesInAsm(asm);
}
```

После этого пользователь сможет вводить абсолютный путь к интересующей сборке (например, `C:\MyApp\MyAsm.dll`). По сути, метод `Assembly.LoadFrom()` позволяет программно предоставлять значение `<codeBase>`. Теперь консольному приложению можно передавать полный путь. Таким образом, если сборка `CarLibrary.dll` находится в папке `C:\MyCode`, будет получен следующий вывод:

```
***** External Assembly Viewer *****
Enter an assembly to evaluate
or enter Q to quit: C:\MyCode\CarLibrary.dll

***** Types in Assembly *****
->CarLibrary, Version=2.0.0.0, Culture=neutral, PublicKeyToken=33a2bc294331e8b9
Type: CarLibrary.EngineState
Type: CarLibrary.Car
Type: CarLibrary.SportsCar
Type: CarLibrary.MiniVan
```

Исходный код. Проект `ExternalAssemblyReflector` доступен в подкаталоге `Chapter 15`.

Рефлексия разделяемых сборок

Метод `Assembly.Load()` имеет несколько перегруженных версий. Одна из них позволяет указать значение культуры (для локализованных сборок), а также номер версии и значение маркера открытого ключа (для разделяемых сборок). Все вместе эти элементы, которые идентифицируют сборку, называются *отображаемым именем*. Формат отображаемого имени — это строка с разделителями-запятыми пар “имя/значение”, начинающаяся с дружественного имени сборки, за которой следуют необязательные квалификаторы (в любом порядке). Ниже приведен шаблон, которым следует пользоваться (необязательные элементы указаны в круглых скобках):

Имя (,Version = <старший номер>.<младший номер>.<номер сборки>.<номер редакции>)
(,Culture = <маркер культуры>) (,PublicKeyToken = <маркер открытого ключа>)

При создании отображаемого имени соглашение PublicKeyToken=null указывает, что требуется связывание и сопоставление со сборкой, не имеющей строгого имени. Кроме того, Culture="" определяет, что сопоставление должно выполняться с использованием стандартной культуры целевой машины, например:

```
// Загрузить версию 1.0.0.0 сборки CarLibrary, используя стандартную культуру.
Assembly a =
    Assembly.Load(@"CarLibrary, Version=1.0.0.0, PublicKeyToken=null, Culture="");
```

Следует также иметь в виду, что в пространстве имен System.Reflection имеется тип AssemblyName, который позволяет представлять строковую информацию наподобие той, что была показана выше, в виде удобной объектной переменной. Обычно этот класс применяется вместе с классом System.Version, который является объектно-ориентированной оболочкой для номера версии сборки. После создания отображаемое имя может передаваться перегруженной версии метода Assembly.Load():

```
// Использовать AssemblyName для определения отображаемого имени.
AssemblyName asmName;
asmName = new AssemblyName();
asmName.Name = "CarLibrary";
Version v = new Version("1.0.0.0");
asmName.Version = v;
Assembly a = Assembly.Load(asmName);
```

Для загрузки разделяемой сборки из GAC в параметре Assembly.Load() должно быть указано значение PublicKeyToken. Например, предположим, что создано новое консольное приложение по имени SharedAsmReflector, и нужно загрузить версию 4.0.0.0 сборки System.Windows.Forms.dll, предоставляемую библиотеками базовых классов .NET. Поскольку количество типов в данной сборке довольно велико, приложение будет выводить только имена открытых перечислений, используя простой запрос LINQ:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using System.Reflection;
using System.IO;

namespace SharedAsmReflector
{
    public class SharedAsmReflector
    {
        private static void DisplayInfo(Assembly a)
        {
            Console.WriteLine("***** Info about Assembly *****");
            Console.WriteLine("Loaded from GAC? {0}", a.GlobalAssemblyCache);
            // Загружена из GAC?
            Console.WriteLine("Asm Name: {0}", a.GetName().Name);
            // Имя сборки
            Console.WriteLine("Asm Version: {0}", a.GetName().Version);
            // Версия сборки
            Console.WriteLine("Asm Culture: {0}",
                a.GetName().CultureInfo.DisplayName);
            Console.WriteLine("\nHere are the public enums:");
            // Список открытых перечислений
```

```
// Использовать запрос LINQ для поиска открытых перечислений.
Type[] types = a.GetTypes();
var publicEnums = from pe in types where pe.IsEnum &&
                  pe.IsPublic select pe;

foreach (var pe in publicEnums)
{
    Console.WriteLine(pe);
}

static void Main(string[] args)
{
    Console.WriteLine("***** The Shared Asm Reflector App *****\n");
    // Загрузить System.Windows.Forms.dll из GAC.
    string displayName = null;
    displayName = "System.Windows.Forms," +
        "Version=4.0.0.0," +
        "PublicKeyToken=b77a5c561934e089," +
        @"Culture=""";
    Assembly asm = Assembly.Load(displayName);
    DisplayInfo(asm);
    Console.WriteLine("Done!");
    Console.ReadLine();
}
}
```

Исходный код. Проект SharedAsmReflector доступен в подкаталоге Chapter 15.

К этому моменту вы должны понимать, как использовать некоторые ключевые члены пространства имен System.Reflection для получения метаданных во время выполнения. Разумеется, необходимость в самостоятельном построении специальных браузеров объектов в повседневной работе, скорее всего, будет возникать нечасто. Тем не менее, не забывайте, что службы рефлексии являются основой для нескольких общих действий программирования, включая *позднее связывание*.

Позднее связывание

Поздним связыванием называется технология, которая позволяет создавать экземпляры заданного типа и обращаться к его членам во время выполнения без необходимости в жестком кодировании факта его существования на этапе компиляции. При построении приложения, в котором предусмотрено позднее связывание с типом из внешней сборки, устанавливать ссылку на эту сборку нет никаких причин; следовательно, в манифесте вызывающего кода она непосредственно не указывается.

На первый взгляд оценить пользу от позднего связывания не так-то легко. Действительно, если есть возможность выполнить раннее связывание с объектом (например, добавить ссылку на сборку и разместить экземпляр типа с помощью ключевого слова new), то так следует и поступать. Одна из наиболее веских причин состоит в том, что раннее связывание позволяет выявлять ошибки на этапе компиляции, а не во время выполнения. Тем не менее, позднее связывание играет важную роль в любом расширяемом приложении. Пример построения такого «расширяемого» приложения будет приведен в конце настоящей главы, в разделе «Построение расширяемого приложения», а пока рассмотрим роль класса Activator.

Класс System.Activator

Класс System.Activator (определенный в mscorlib.dll) играет ключевую роль в процессе позднего связывания .NET. Для текущего примера интересен пока только метод Activator.CreateInstance(), который позволяет создавать экземпляр типа в стиле позднего связывания. Этот метод имеет несколько перегруженных версий и потому обеспечивает довольно высокую гибкость. В самой простой версии CreateInstance() принимает действительный объект Type, описывающий сущность, которая должна размещаться в памяти на лету.

Создадим новое консольное приложение по имени LateBindingApp, импортируем в него пространства имен System.IO и System.Reflection с помощью ключевого слова using и модифицируем класс Program следующим образом:

```
// Это приложение будет загружать внешнюю сборку
// и создавать объект, используя позднее связывание.
public class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with Late Binding *****");
        // Попробовать загрузить локальную копию CarLibrary.
        Assembly a = null;
        try
        {
            a = Assembly.Load("CarLibrary");
        }
        catch(FileNotFoundException ex)
        {
            Console.WriteLine(ex.Message);
            return;
        }
        if(a != null)
            CreateUsingLateBinding(a);
        Console.ReadLine();
    }
    static void CreateUsingLateBinding(Assembly asm)
    {
        try
        {
            // Получить метаданные для типа Minivan.
            Type miniVan = asm.GetType("CarLibrary.Minivan");
            // Создать объект Minivan на лету.
            object obj = Activator.CreateInstance(miniVan);
            Console.WriteLine("Created a {0} using late binding!", obj);
        }
        catch(Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
    }
}
```

Перед запуском этого приложения необходимо вручную скопировать файл CarLibrary.dll в подкаталог bin\Debug внутри каталога нового приложения. Причина в том, что в коде вызывается метод Assembly.Load(), а это значит, что CLR-среда будет зондировать только папку клиента (при желании можно было бы воспользоваться методом Assembly.LoadFrom() и указывать полный путь к сборке, но в данном случае в этом нет необходимости).

На заметку! В этом примере не нужно устанавливать ссылку на `CarLibrary.dll` в Visual Studio! Это действие приведет к добавлению записи о данной библиотеке в манифест клиента. Вся суть позднего связывания состоит в том, чтобы попытаться создать объект, о котором ничего не известно на этапе компиляции.

Обратите внимание, что метод `Activator.CreateInstance()` возвращает экземпляр `System.Object`, а не строго типизированный объект `MiniVan`. Следовательно, если применить к переменной `obj` операцию точки, члены класса `MiniVan` не будут видны. На первый взгляд может показаться, что эту проблему удастся решить с помощью явного приведения:

```
// Выполнить приведение для получения доступа к членам MiniVan?  
// Нет! Компилятор сообщит об ошибке!  
object obj = (MiniVan)Activator.CreateInstance(minivan);
```

Однако из-за того, что для приложения не была установлена ссылка на сборку `CarLibrary.dll`, применять ключевое слово `using` для импортирования пространства имен `CarLibrary` нельзя, а значит и нельзя использовать `MiniVan` в операции приведения! Не забывайте, что весь смысл позднего связывания заключается в создании экземпляров объектов, о которых на этапе компиляции ничего не известно. Учитывая это, возникает вопрос: как вызывать методы объекта `MiniVan`, сохраненного в ссылке на `System.Object`? Ответ: конечно же, с помощью рефлексии.

Вызов методов без параметров

Предположим, что требуется вызвать метод `TurboBoost()` объекта `MiniVan`. Вспомните, что этот метод переводит двигатель в нерабочее состояние и затем отображает окно с соответствующим сообщением. Первый шаг заключается в получении объекта `MethodInfo` для метода `TurboBoost()`, используя `Type.GetMethod()`. Имея результирующий объект `MethodInfo`, можно вызвать `MiniVan.TurboBoost()` с помощью метода `Invoke()`. Метод `MethodInfo.Invoke()` требует указания всех параметров, которые подлежат передаче методу, представленному `MethodInfo`, в виде массива объектов `System.Object` (т.к. параметры могут быть самыми разнообразными сущностями).

Поскольку метод `TurboBoost()` не принимает параметров, можно просто передать `null` (это говорит о том, что вызываемый метод не имеет параметров). Модифицируем метод `CreateUsingLateBinding()` следующим образом:

```
static void CreateUsingLateBinding(Assembly asm)  
{  
    try  
    {  
        // Получить метаданные для типа Minivan.  
        Type miniVan = asm.GetType("CarLibrary.Minivan");  
        // Создать объект MiniVan на лету.  
        object obj = Activator.CreateInstance(miniVan);  
        Console.WriteLine("Created a {0} using late binding!", obj);  
        // Получить информацию для TurboBoost.  
        MethodInfo mi = miniVan.GetMethod("TurboBoost");  
        // Вызвать метод (null означает отсутствие параметров).  
        mi.Invoke(obj, null);  
    }  
    catch(Exception ex)  
    {  
        Console.WriteLine(ex.Message);  
    }  
}
```

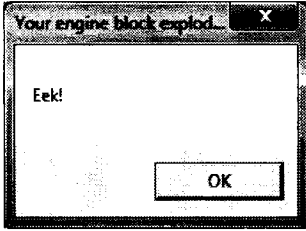


Рис. 15.2. Вызов метода через позднее связывание

Теперь после запуска приложения в результате вызова метода `TurboBoost()` отображается окно с сообщением, показанное на рис. 15.2.

Вызов методов с параметрами

Когда нужно использовать позднее связывание для вызова метода, ожидающего параметры, их потребуется предоставить в виде слабо типизированного массива `object`. Вспомните, что в версии 2.0.0.0 библиотеки `CarLibrary.dll` был определен следующий метод в классе `Car`:

```
public void TurnOnRadio(bool musicOn, MusicMedia mm)
{
    if (musicOn)
        MessageBox.Show(string.Format("Jamming {0}", mm));
    else
        MessageBox.Show("Quiet time...");
}
```

Этот метод принимает два параметра: булевское значение, которое указывает, включена ли музыкальная система в автомобиле, и перечисление, представляющее тип музыкального проигрывателя. Вспомните, что это перечисление выглядит следующим образом:

```
public enum MusicMedia
{
    musicCd,           // 0
    musicTape,         // 1
    musicRadio,        // 2
    musicMp3           // 3
}
```

Ниже приведен код нового метода класса `Program`, в котором вызывается `TurnOnRadio()`. Обратите внимание, что для перечисления `MusicMedia` используются лежащие в основе числовые значения.

```
static void InvokeMethodWithArgsUsingLateBinding(Assembly asm)
{
    try
    {
        // Получить описание метаданных типа SportsCar.
        Type sport = asm.GetType("CarLibrary.SportsCar");

        // Создать объект типа SportsCar.
        object obj = Activator.CreateInstance(sport);

        // Вызвать метод TurnOnRadio() с аргументами.
        MethodInfo mi = sport.GetMethod("TurnOnRadio");
        mi.Invoke(obj, new object[] { true, 2 });
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}
```

К этому моменту уже должны быть ясны отношения между рефлексией, динамической загрузкой и поздним связыванием. Естественно, кроме описанных выше, API-интерфейс рефлексии предлагает множество других средств, тем не менее, в главе была заложена хорошая основа для дальнейшего изучения.

По-прежнему может интересовать вопрос: *когда необходимо применять все эти приемы в своих приложениях?* Это должно проясниться в конце главы, а в следующем разделе пойдет речь о роли атрибутов в .NET.

Исходный код. Проект LateBindingApp доступен в подкаталоге Chapter 15.

Роль атрибутов .NET

Как было показано в начале настоящей главы, одной из задач компилятора .NET является генерация описаний метаданных для всех типов, которые были определены и на которые имеются ссылки в текущем проекте. В дополнение к этим стандартным метаданным, содержащимся в любой сборке, платформа .NET позволяет программистам встраивать в сборку дополнительные метаданные, используя *атрибуты*. В сущности, атрибуты представляют собой не более чем просто аннотации кода, которые могут применяться к заданному типу (классу, интерфейсу, структуре и т.д.), члену (свойству, методу и т.д.), сборке или модулю.

Атрибуты .NET — это типы классов, которые расширяют абстрактный базовый класс `System.Attribute`. В пространствах имен .NET имеется множество predefined атрибутов, которые можно использовать в своих приложениях. Более того, можно также строить собственные атрибуты для дополнительного уточнения поведения своих типов за счет создания нового типа, порожденного от `Attribute`.

Библиотека базовых классов .NET предлагает множество атрибутов в различных пространствах имен. Некоторые из predefined атрибутов перечислены в табл. 15.3.

Таблица 15.3. Некоторые из predefined атрибутов в .NET

Атрибут	Описание
[CLSCompliant]	Заставляет аннотированный элемент соответствовать правилам CLS (Common Language Specification — общезыко́вая спецификация). Вспомните, что типы, совместимые с CLS, могут гарантированно использоваться всеми языками программирования .NET
[DllImport]	Позволяет коду .NET отправлять вызовы в любую неуправляемую библиотеку кода на C или C++, включая API-интерфейс операционной системы. Обратите внимание, что при взаимодействии с программным обеспечением на основе COM этот атрибут не применяется
[Obsolete]	Помечает тип или член как устаревший. Когда другие программисты попытаются использовать элемент с таким атрибутом, они получат соответствующее предупреждение от компилятора
[Serializable]	Помечает класс или структуру как сериализуемую, что означает способность сохранения своего текущего состояния в потоке
[NonSerialized]	Указывает, что данное поле в классе или структуре не должно сохраняться в процессе сериализации
[ServiceContract]	Помечает метод как контракт, реализованный службой WCF

Важно уяснить, что при применении атрибутов в коде встроенные метаданные, в сущности, бесполезны до тех пор, пока другая часть программного обеспечения явно не запросит эту информацию через рефлексия. Если этого не происходит, метаданные, встроенные в сборку, игнорируются и не причиняют никакого вреда.

Потребители атрибутов

Как нетрудно догадаться, в составе .NET Framework 4.5 SDK поставляется множество утилит, которые позволяют производить поиск разнообразных атрибутов. Даже сам компилятор C# (csc.exe) запрограммирован на обнаружение различных атрибутов во время компиляции. Например, встретив атрибут [CLSCompliant], компилятор автоматически проверяет помеченный им элемент и удостоверяется в том, что в нем открыты внешнему миру только совместимые с CLS конструкции. Если же компилятор обнаруживает элемент с атрибутом [Obsolete], он отображает в окне Error List (Список ошибок) среды Visual Studio соответствующее предупреждение.

В дополнение к инструментам для разработки, многочисленные методы в библиотеках базовых классов .NET изначально запрограммированы на распознавание определенных атрибутов посредством рефлексии. Например, если нужно сохранить информацию о состоянии объекта в файле, то все, что потребуется сделать — аннотировать класс или структуру атрибутом [Serializable]. Встретив этот атрибут, метод Serialize() класса BinaryFormatter автоматически сохраняет соответствующий объект в файле в компактном двоичном формате.

И, наконец, можно строить приложения, способные распознавать специальные атрибуты, а также любые атрибуты из библиотек базовых классов .NET. Подобным образом можно, в сущности, создавать набор “ключевых слов”, воспринимаемых специфичным набором сборок.

Применение атрибутов в C#

Для демонстрации процесса применения атрибутов в C# создадим новое консольное приложение по имени ApplyingAttributes. Предположим, что требуется построить класс под названием Motorcycle (мотоцикл), который бы мог сохраняться в двоичном формате. Для этого достаточно применить к определению класса атрибут [Serializable]. Если в классе есть какое-то поле, которое не должно сохраняться, к нему должен быть применен атрибут [NonSerialized].

```
// Этот класс может быть сохранен на диске.
[Serializable]
public class Motorcycle
{
    // Однако это поле сохраняться не будет.
    [NonSerialized]
    float weightOfCurrentPassengers;

    // Эти поля остаются сериализируемыми.
    bool hasRadioSystem;
    bool hasHeadSet;
    bool hasSissyBar;
}
```

На заметку! Атрибут применяется только к элементу, который следует сразу же после него. Например, в классе Motorcycle сериализации не будет подвергнуто только поле weightOfCurrentPassengers. Остальные поля будут сериализируемыми, потому что сам класс аннотирован атрибутом [Serializable].

В данный момент беспокоиться о процессе сериализации объектов не следует (т.к. все необходимые детали будут представлены в главе 20). Достаточно отметить, что для применения атрибута его имя должно быть заключено в квадратные скобки.

После компиляции этого класса можно просмотреть дополнительные метаданные с помощью утилиты ildasm.exe. Обратите внимание, что эти атрибуты сопро-

обладают маркером `serializable` (в строке с треугольником красного цвета внутри класса `Motorcycle`) и маркером `notserialized` (в строке, представляющей поле `weightOfCurrentPassengers`), как показано на рис. 15.3.

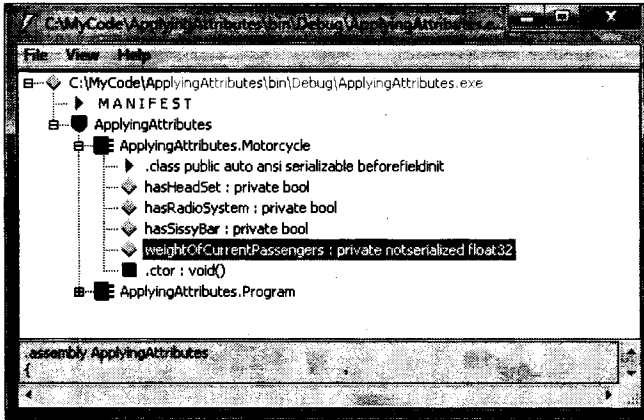


Рис. 15.3. Просмотр атрибутов в `ildasm.exe`

Нетрудно догадаться, что к единственному элементу можно применять множество атрибутов. Предположим, что имеется унаследованный тип класса `C#` (`HorseAndBuggy`), который был помечен как сериализуемый, но для текущей разработки он считается устаревшим. Вместо того чтобы удалять определение этого класса из кодовой базы (с риском нарушения работы существующего программного обеспечения), можно поместить этот класс атрибутом `[Obsolete]`. Для применения множества атрибутов к одному элементу просто используйте список, разделенный запятыми:

```
[Serializable, Obsolete("Use another vehicle!")]
public class HorseAndBuggy
{
    // ...
}
```

В качестве альтернативы применить множество атрибутов к единственному элементу можно следующим образом (конечный результат будет идентичным):

```
[Serializable]
[Obsolete("Use another vehicle!")]
public class HorseAndBuggy
{
    // ...
}
```

Сокращенная нотация атрибутов `C#`

В документации .NET Framework 4.5 SDK вы могли заметить, что действительным именем класса, представляющего атрибут `[Obsolete]`, является `ObsoleteAttribute`, а не `Obsolete`. По соглашению имена всех атрибутов .NET (включая специальные, создаваемые вами) сопровождаются суффиксом `Attribute`. Однако для упрощения процесса применения атрибутов язык `C#` не требует обязательного ввода этого суффикса. Учитывая это, следующая версия класса `HorseAndBuggy` является идентичной предыдущей (но влечет за собой более объемный ввод с клавиатуры):

```
[SerializableAttribute]
[ObsoleteAttribute("Use another vehicle!")]
public class HorseAndBuggy
{
    // ...
}
```

Имейте в виду, что такая сокращенная нотация для атрибутов разрешена только в C#. Не все языки .NET ее поддерживают.

Указание параметров конструктора для атрибутов

Обратите внимание, что атрибут `[Obsolete]` может принимать нечто похожее на параметр конструктора. Если взглянуть на формальное определение этого атрибута в окне кода (открываемое через меню View (Вид) в Visual Studio), обнаружится, что этот класс действительно предоставляет конструктор, принимающий `System.String`:

```
public sealed class ObsoleteAttribute : Attribute
{
    public ObsoleteAttribute(string message, bool error);
    public ObsoleteAttribute(string message);
    public ObsoleteAttribute();
    public bool IsError { get; }
    public string Message { get; }
}
```

Очень важно понимать, что при предоставлении атрибуту параметров конструктора, этот атрибут не размещается в памяти до тех пор, пока к параметрам не будет применена рефлексия другим типом или внешним инструментом. Строковые данные, определенные на уровне атрибутов, просто сохраняются внутри сборки в виде блока метаданных.

Атрибут `[Obsolete]` в действии

Теперь, когда класс `HorseAndBuggy` помечен как устаревший, попытка размещения его экземпляра, как показано ниже:

```
static void Main(string[] args)
{
    HorseAndBuggy mule = new HorseAndBuggy();
}
```

приводит к тому, что указанные строковые данные извлекаются и отображаются в окне Error List среды Visual Studio, а при наведении курсора мыши на этот устаревший тип появляется проблемная строка кода (рис. 15.4).

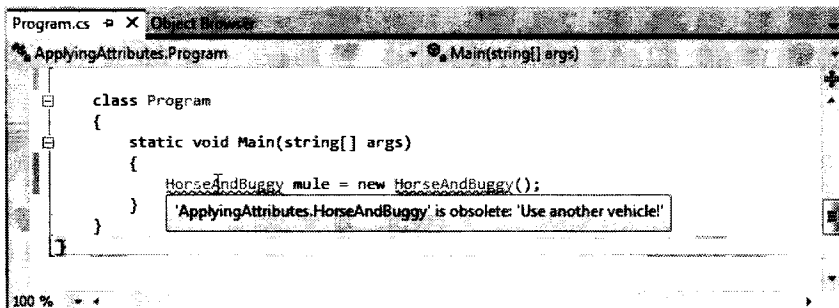


Рис. 15.4. Атрибуты в действии

В этом случае “другой частью программного обеспечения кода”, применяющей рефлексии к атрибуту [Obsolete], является компилятор C#. К настоящему моменту вы должны понимать следующие ключевые моменты, касающиеся атрибутов .NET.

- Атрибуты — это классы, производные от System.Attribute.
- Атрибуты приводят к включению в сборку дополнительных метаданных.
- Атрибуты в основном бесполезны до тех пор, пока другой агент не применит к ним рефлексии.
- Атрибуты применяются в C# с использованием квадратных скобок.

А теперь давайте посмотрим, как создавать собственные специальные атрибуты и писать код, применяющий рефлексию к встроенным метаданным.

Исходный код. Проект ApplyingAttributes доступен в подкаталоге Chapter 15.

Построение специальных атрибутов

Первый шаг при построении специального атрибута заключается в создании нового класса, производного от System.Attribute. Чтобы не отклоняться от автомобильной темы, используемой повсюду в этой книге, создадим новый проект типа Class Library (Библиотека классов) на C# под названием AttributedCarLibrary. В этой сборке будет определено несколько транспортных средств с использованием специального атрибута по имени VehicleDescriptionAttribute:

```
// Специальный атрибут.
public sealed class VehicleDescriptionAttribute : System.Attribute
{
    public string Description { get; set; }
    public VehicleDescriptionAttribute(string vehicalDescription)
    {
        Description = vehicalDescription;
    }
    public VehicleDescriptionAttribute() { }
}
```

Как видно в коде, VehicleDescriptionAttribute поддерживает фрагмент строковых данных, которым можно манипулировать с помощью автоматического свойства (Description). Помимо того факта, что этот класс является производным от System.Attribute, ничего особенного в нем больше нет.

На заметку! По причинам безопасности в .NET наилучшим приемом считается запечатывание всех специальных атрибутов. В действительности Visual Studio предлагает фрагмент кода под названием Attribute, который позволяет сгенерировать в окне кода новый производный от Attribute класс. Об использовании фрагментов кода подробно рассказывалось в главе 2; вспомните, что для раскрытия любого фрагмента кода необходимо два раза нажать клавишу <Tab>.

Применение специальных атрибутов

С учетом того, что класс VehicleDescriptionAttribute является производным от System.Attribute, теперь можно аннотировать транспортные средства. В целях тестирования добавим в новую библиотеку классов следующие определения:


```
// Назначить описание с помощью "именованного свойства".
[Serializable]
[VehicleDescription(Description = "My rocking Harley")]
public class Motorcycle
{
}

[SerializableAttribute]
[ObsoleteAttribute("Use another vehicle!")]
[VehicleDescription("The old gray mare, she ain't what she used to be...")]
public class HorseAndBuggy
{
}

[VehicleDescription("A very long, slow, but feature-rich auto")]
public class Winnebago
{
}
```

Синтаксис именованных свойств

Обратите внимание, что для назначения описания классу `Motorcycle` применяется новый фрагмент связанного с атрибутами синтаксиса, который называется *именованным свойством*. В конструкторе первого атрибута `[VehicleDescription]` установка лежащих в основе строковых данных производится с использованием свойства `Description`. Если внешний агент применит рефлексию к данному атрибуту, свойству `Description` будет передано соответствующее значение (синтаксис именованных свойств разрешен только в случае предоставления в атрибуте доступного для записи свойства .NET).

В противоположность этому, в типах `HorseAndBuggy` и `Winnebago` синтаксис именованных свойств не используется, а строковые данные просто передаются через специальный конструктор. В любом случае после компиляции сборки `AttributedCarLibrary` можно воспользоваться утилитой `ildasm.exe` и посмотреть, какие описания метаданных будут вставлены для типа. Например, на рис. 15.5 показано, как в `ildasm.exe` будет выглядеть описание класса `Winnebago`, в частности — данные внутри элемента `beforefieldinit`.

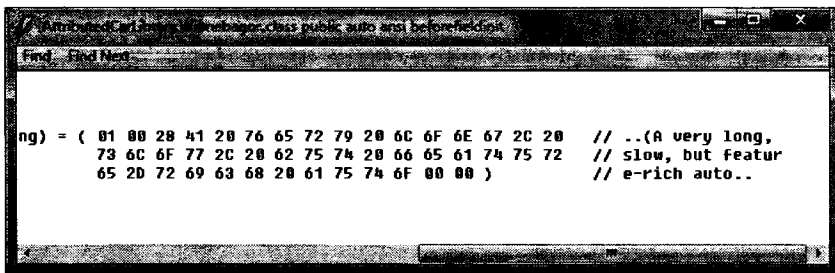


Рис. 15.5. Встроенные данные описания транспортного средства

Ограничение использования атрибутов

По умолчанию специальные атрибуты могут быть применены к практически любому аспекту кода (методам, классам, свойствам и т.д.). Следовательно, если бы это имело смысл, можно было бы использовать `VehicleDescription` для уточнения методов, свойств или полей (помимо прочего):

```
[VehicleDescription("A very long, slow, but feature-rich auto")]
public class Winnebago
{
    [VehicleDescription("My rocking CD player")]
    public void PlayMusic(bool On)
    {
        ...
    }
}
```

В одних случаях такое поведение оказывается именно тем, что нужно, однако в других может требоваться создание специального атрибута, действие которого распространяется только на избранные элементы кода. Чтобы ограничить область действия специального атрибута, необходимо применить к его определению атрибут `[AttributeUsage]`. Атрибут `[AttributeUsage]` позволяет предоставлять (посредством операции "ИЛИ") любую комбинацию значений из перечисления `AttributeTargets`:

```
// Это перечисление определяет возможные цели атрибута.
public enum AttributeTargets
{
    All, Assembly, Class, Constructor,
    Delegate, Enum, Event, Field, GenericParameter,
    Interface, Method, Module, Parameter,
    Property, ReturnValue, Struct
}
```

Кроме того, атрибут `[AttributeUsage]` также позволяет дополнительно устанавливать свойство `AllowMultiple`, которое указывает, может ли атрибут применяться к одному и тому же элементу более одного раза (стандартным значением этого свойства является `false`). Помимо этого он позволяет указывать, должен ли атрибут наследоваться производными классами, за счет применения именованного свойства `Inherited` (со стандартным значением `true`).

Для указания, что атрибут `[VehicleDescription]` может применяться к классу или структуре только один раз, необходимо обновить определение `VehicleDescriptionAttribute` следующим образом:

```
// На этот раз мы используем атрибут AttributeUsage
// для аннотирования специального атрибута.
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct,
    Inherited = false)]
public sealed class VehicleDescriptionAttribute : System.Attribute
{
    ...
}
```

Если разработчик попытается применить атрибут `[VehicleDescription]` не к классу или структуре, на этапе компиляции возникнет ошибка.

Атрибуты уровня сборки

Атрибуты можно также применять ко всем типам внутри отдельной сборки, используя дескриптор `[assembly:]`. Например, предположим, что необходимо обеспечить совместимость с CLS для всех открытых членов во всех открытых типах, определенных внутри сборки.

На заметку! В главе 1 рассказывалось о роли сборок, совместимых с CLS. Вспомните, что совместимая с CLS сборка может использоваться всеми поставляемыми языками программирования .NET. Если в открытых типах создаются открытые члены, которые позволяют видеть несовместимые с CLS конструкции программирования (такие как данные без знака или параметры типа указателей), то другие языки .NET могут оказаться не в состоянии ими пользоваться. Таким образом, когда вы строите библиотеки кода C#, которые должны применяться множеством языков .NET, проверка на совместимость с CLS является обязательной.

Для этого необходимо просто добавить в самое начало файла исходного кода C# показанный ниже атрибут уровня сборки. Имейте в виду, что все атрибуты уровня сборки или модуля должны быть перечислены за пределами контекста любого пространства имен! При добавлении в проект атрибутов уровня сборки или модуля рекомендуется придерживаться следующей схемы для файла кода:

```
// Перечислить первыми операторы using.
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

// Теперь перечислить атрибуты уровня сборки или модуля.
// Обеспечить совместимость с CLS для всех открытых типов в этой сборке.
[assembly: CLSCompliant(true)]

// Далее может следовать ваше пространство имен и типы.
namespace AttributedCarLibrary
{
    // Типы...
}
```

Если теперь добавить фрагмент кода, выходящий за рамки спецификации CLS (такой как открытый элемент данных без знака):

```
// Типы ulong не отвечают спецификации CLS.
public class Winnebago
{
    public ulong notCompliant;
}
```

компилятор выдаст соответствующее предупреждение.

Файл AssemblyInfo.cs, генерируемый Visual Studio

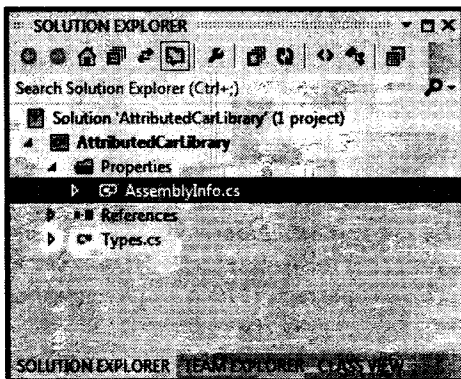


Рис. 15.6. Файл AssemblyInfo.cs

По умолчанию все проекты Visual Studio получают файл по имени AssemblyInfo.cs, который можно просмотреть, раскрыв в окне Solution Explorer узел Properties (Свойства), как показано на рис. 15.6.

Этот файл является очень удобным местом для помещения атрибутов, которые должны применяться на уровне сборки. В главе 14 во время исследования сборок .NET уже рассказывалось о том, что в манифесте содержатся метаданные уровня сборки. Большая часть из них берется из атрибутов уровня сборки, которые описаны в табл. 15.4.

Таблица 15.4. Избранные атрибуты уровня сборки

Атрибут	Описание
[AssemblyCompany]	Хранит общую информацию о компании
[AssemblyCopyright]	Хранит любую информацию, касающуюся авторских прав на данный продукт или сборку
[AssemblyCulture]	Предоставляет информацию о том, какие культуры или языки поддерживает сборка
[AssemblyDescription]	Хранит дружественное описание продукта или модулей, из которых состоит сборка
[AssemblyKeyFile]	Указывает имя файла, в котором содержится пара ключей, используемых для подписания сборки (т.е. создания строгого имени)
[AssemblyProduct]	Предоставляет информацию о продукте
[AssemblyTrademark]	Предоставляет информацию о торговой марке
[AssemblyVersion]	Указывает информацию о версии сборки в формате <старший_номер.младший_номер.номер_сборки.номер_редакции>

Исходный код. Проект `AttributedCarLibrary` доступен в подкаталоге `Chapter 15`.

Рефлексия атрибутов с использованием раннего связывания

Вспомните, что атрибуты остаются бесполезными до тех пор, пока к их значениям не применяется рефлексия в другой части программного обеспечения. После обнаружения атрибута может предприниматься любое необходимое действие. Как и в любом приложении, специальный атрибут может быть обнаружен с использованием либо раннего, либо позднего связывания. Для применения раннего связывания определение интересующего атрибута (в данном случае `VehicleDescriptionAttribute`) должно присутствовать в клиентском приложении на этапе компиляции. Учитывая то, что специальный атрибут определен в сборке `AttributedCarLibrary` как открытый класс, раннее связывание будет наилучшим вариантом.

Чтобы проиллюстрировать процесс рефлексии специальных атрибутов, создадим новое консольное приложение по имени `VehicleDescriptionAttributeReader`, добавим в него ссылку на сборку `AttributedCarLibrary` и поместим в первоначальный файл `*.cs` следующий код:

```
// Рефлексия атрибутов с использованием раннего связывания.
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using AttributedCarLibrary;

namespace VehicleDescriptionAttributeReader
{
    class Program
    {
        static void Main(string[] args)
        {

```

```

        Console.WriteLine("***** Value of VehicleDescriptionAttribute *****\n");
        ReflectOnAttributesUsingEarlyBinding();
        Console.ReadLine();
    }

    private static void ReflectOnAttributesUsingEarlyBinding()
    {
        // Получить объект Type, представляющий Winnebago.
        Type t = typeof(Winnebago);

        // Получить все атрибуты Winnebago.
        object[] customAtts = t.GetCustomAttributes(false);

        // Вывести описание.
        foreach (VehicleDescriptionAttribute v in customAtts)
            Console.WriteLine("-> {0}\n", v.Description);
    }
}

```

Метод `Type.GetCustomAttributes()` возвращает массив объектов со всеми атрибутами, которые применяются к члену, представленному `Type` (булевский параметр указывает, должен ли поиск продолжаться вверх по цепочке наследования). После получения списка атрибутов осуществляется проход по всем классам `VehicleDescriptionAttribute` с отображением значения свойства `Description`.

Исходный код. Проект `VehicleDescriptionAttributeReader` доступен в подкаталоге `Chapter 15`.

Рефлексия атрибутов с использованием позднего связывания

В предыдущем примере для вывода описания транспортного средства типа `Winnebago` применялось ранее связывание. Это было возможно благодаря тому, что тип класса `VehicleDescriptionAttribute` определен в сборке `AttributedCarLibrary` как открытый член. Для рефлексии атрибутов также допускается использование динамической загрузки и позднего связывания.

Создадим новый проект по имени `VehicleDescriptionAttributeReaderLateBinding` и скопируем сборку `AttributedCarLibrary.dll` в его каталог `bin\Debug`. Теперь модифицируем класс `Program` следующим образом:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using System.Reflection;

namespace VehicleDescriptionAttributeReaderLateBinding
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** Value of VehicleDescriptionAttribute *****\n");
            ReflectAttributesUsingLateBinding();
            Console.ReadLine();
        }
    }
}

```

```
private static void ReflectAttributesUsingLateBinding()
{
    try
    {
        // Загрузить локальную копию AttributedCarLibrary.
        Assembly asm = Assembly.Load("AttributedCarLibrary");

        // Получить информацию о типе для VehicleDescriptionAttribute.
        Type vehicleDesc =
            asm.GetType("AttributedCarLibrary.VehicleDescriptionAttribute");

        // Получить информацию о типе для свойства Description.
        PropertyInfo propDesc = vehicleDesc.GetProperty("Description");

        // Получить все типы в сборке.
        Type[] types = asm.GetTypes();

        // Пройти по всем типам и получить любые VehicleDescriptionAttribute.
        foreach (Type t in types)
        {
            object[] objs = t.GetCustomAttributes(vehicleDesc, false);

            // Пройти по каждому VehicleDescriptionAttribute и вывести
            // описание с использованием позднего связывания.
            foreach (object o in objs)
            {
                Console.WriteLine("-> {0}: {1}\n",
                    t.Name, propDesc.GetValue(o, null));
            }
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}
```

Если вы внимательно прорабатывали приводимые ранее примеры, то этот код должен быть более или менее понятен. Единственным интересным моментом здесь является использование метода `PropertyInfo.GetValue()`, который служит для активизации средства доступа к свойству. Ниже показан вывод текущего примера:

```
***** Value of VehicleDescriptionAttribute *****
-> Motorcycle: My rocking Harley
-> HorseAndBuggy: The old gray mare, she ain't what she used to be...
-> Winnebago: A very long, slow, but feature-rich auto
```

Исходный код. Проект `VehicleDescriptionAttributeReaderLateBinding` доступен в подкаталоге `Chapter 15`.

Возможное применение на практике рефлексии, позднего связывания и специальных атрибутов

Хотя в главе приводилось множество примеров применения этих приемов, все равно может остаться вопрос о том, когда следует использовать рефлексия, динамическое связывание и специальные атрибуты в своих приложениях. Эта тема может показаться

более академической, нежели практической. Чтобы оценить возможность их применения в реальном мире, необходим более серьезный пример. Предположим, что поставлена задача разработать расширяемое приложение, к которому можно было бы подключать сторонние инструменты.

Что означает *расширяемость*? Рассмотрим IDE-среду Visual Studio. При разработке в этом приложении были предусмотрены специальные привязки, которые предоставляют другим производителям программного обеспечения возможность подключения своих специальных модулей. Очевидно, что разработчики Visual Studio не могли установить ссылки на пока еще не построенные внешние сборки .NET (т.е. воспользоваться ранним связыванием), тогда как им удалось предоставить в приложении необходимые привязки? Ниже описан один из возможных способов решения этой задачи.

- Во-первых, расширяемое приложение должно предоставлять какой-нибудь механизм ввода, позволяющий пользователю указать модуль для подключения (такой как диалоговое окно или флаг командной строки). Это требует применения *динамической загрузки*.
- Во-вторых, расширяемое приложение должно иметь возможность выяснения, поддерживает ли модуль корректную функциональность (такую как набор обязательных интерфейсов), необходимую для его подключения к среде. Это требует применения *рефлексии*.
- В-третьих, расширяемое приложение должно получать ссылку на требуемую инфраструктуру (такую как набор интерфейсных типов) и вызывать члены для запуска лежащей в основе функциональности. Это может требовать применения *позднего связывания*.

Если расширяемое приложение изначально программируется так, чтобы запрашивать определенные интерфейсы, оно получает возможность выяснять во время выполнения, может ли активизироваться интересующий тип. После успешного прохождения такой проверки тип может поддерживать дополнительные интерфейсы, которые формируют полиморфную фабрику его функциональности. Именно такой подход был принят командой разработчиков Visual Studio, и в нем нет ничего особо сложного.

Построение расширяемого приложения

В следующих разделах будет рассмотрен полноценный пример создания расширяемого приложения Windows Forms, которое может быть дополнено функциональностью внешних сборок. Если вы не имеете опыта построения графических пользовательских интерфейсов с помощью Windows Forms, можете загрузить предоставленный код решения и работать с ним.

На заметку! Windows Forms был первым API-интерфейсом для разработки настольных приложений в .NET. Однако после выхода версии .NET 3.0 предпочтительной инфраструктурой для построения графических пользовательских интерфейсов быстро стал API-интерфейс Windows Presentation Foundation (WPF). Несмотря на это, Windows Forms будет применяться в нескольких примерах с графическими пользовательскими интерфейсами, т.к. связанный с ним код более понятен, чем соответствующий код WPF.

Если вы не знакомы с процессом построения приложений Windows Forms, просто откройте загруженный код примера и следуйте дальнейшим указаниям в главе. Для удобства ниже перечислены все сборки, которые потребуется создать для разрабатываемого расширяемого приложения.

- `CommonSnappableTypes.dll`. Эта сборка содержит определения типов, которые будут использоваться каждым объектом-оснасткой. На данную сборку будет напрямую ссылаться приложение `Windows Forms`.
- `CSharpSnapIn.dll`. Оснастка, написанная на C#, которая использует типы из сборки `CommonSnappableTypes.dll`.
- `VbSnapIn.dll`. Оснастка, написанная на Visual Basic, которая использует типы из сборки `CommonSnappableTypes.dll`.
- `MyExtendableApp.exe`. Приложение `Windows Forms`, которое может быть расширено функциональностью каждой оснастки.

В этом приложении будут применяться динамическая загрузка, рефлексия и позднее связывание для динамического получения функциональности сборок, о которых заранее ничего не известно.

Построение сборки `CommonSnappableTypes.dll`

В первую очередь необходимо создать сборку, содержащую типы, которые должна использовать оснастка, чтобы быть подключенной к расширяемому приложению `Windows Forms`. Для этого создадим проект типа `Class Library` (Библиотека классов) по имени `CommonSnappableTypes` и определим в нем два следующих типа:

```
namespace CommonSnappableTypes
{
    public interface IAppFunctionality
    {
        void DoIt();
    }

    [AttributeUsage(AttributeTargets.Class)]
    public sealed class CompanyInfoAttribute : System.Attribute
    {
        public string CompanyName { get; set; }
        public string CompanyUrl { get; set; }
    }
}
```

Интерфейс `IAppFunctionality` предоставляет полиморфный интерфейс для всех оснасток, которые могут подключаться к расширяемому приложению `Windows Forms`. Поскольку рассматриваемый пример является демонстрационным, в этом интерфейсе определен единственный метод по имени `DoIt()`. Более реалистичный интерфейс (или набор интерфейсов) мог бы генерировать код сценария, визуализировать изображение в панели инструментов приложения или интегрироваться в главное меню основного приложения.

Тип `CompanyInfoAttribute` — это специальный атрибут, который может применяться к любому классу, желающему подключиться к контейнеру. Как не трудно догадаться по определению класса, `[CompanyInfo]` позволяет разработчику оснастки предоставлять общие сведения о месте происхождения компонента.

Построение оснастки на C#

Далее потребуется создать тип, реализующий интерфейс `IAppFunctionality`. Чтобы не усложнять пример создания расширяемого приложения, сделаем этот тип максимально простым. Создадим новый проект типа `Class Library` на C# по имени `CSharpSnapIn` и определим в нем класс `CSharpModule`. Учитывая, что этот класс должен использовать типы, определенные в сборке `CommonSnappableTypes`, не забудьте установить ссылку на

нее (а также на сборку System.Windows.Forms.dll для отображения сообщений). Ниже показан необходимый код.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using CommonSnappableTypes;
using System.Windows.Forms;

namespace CSharpSnapIn
{
    [CompanyInfo(CompanyName = "My Company",
        CompanyUrl = "www.MyCompany.com")]
    public class CSharpModule : IAppFunctionality
    {
        void IAppFunctionality.DoIt()
        {
            MessageBox.Show("You have just used the C# snap-in!");
        }
    }
}
```

Обратите внимание на явную реализацию интерфейса IAppFunctionality. Это не является обязательным; главное понимать, что единственной частью системы, которая нуждается в непосредственном взаимодействии с данным интерфейсным типом, является размещающее Windows-приложение. Благодаря явной реализации этого интерфейса, метод DoIt() не будет доступен непосредственно из типа CSharpModule.

Построение оснастки на Visual Basic

Чтобы симитировать стороннего производителя, предпочитающего использовать Visual Basic, а не C#, создадим новый проект типа Class Library (Библиотека классов) на Visual Basic по имени VbSnapIn и добавим в него ссылки на те же самые внешние сборки, что и в предыдущем проекте CSharpSnapIn.

На заметку! По умолчанию папка References (Ссылки) проекта Visual Basic в окне Solution Explorer отображаться не будет, поэтому для добавления ссылок в этот проект необходимо пользоваться пунктом меню Add Reference⇒Project (Добавить ссылку⇒Проект) в Visual Studio.

Необходимый код Visual Basic также прост:

```
Imports System.Windows.Forms
Imports CommonSnappableTypes

<CompanyInfo(CompanyName:="Chucky's Software", CompanyUrl:="www.ChuckySoft.com")>
Public Class VbSnapIn
    Implements IAppFunctionality

    Public Sub DoIt() Implements CommonSnappableTypes.IAppFunctionality.DoIt
        MessageBox.Show("You have just used the VB snap in!")
    End Sub
End Class
```

Обратите внимание, что в синтаксисе Visual Basic для применения атрибутов используются не квадратные ([]), а угловые (< >) скобки. Кроме того, для реализации интерфейсных типов в заданном классе или структуре служит ключевое слово Implements.

Построение расширяемого приложения Windows Forms

И, наконец, последний шаг заключается в создании нового приложения Windows Forms (MyExtendableApp) на C#, которое позволяет пользователю выбирать оснастку с помощью стандартного диалогового окна открытия файлов Windows. Если ранее вам не приходилось создавать приложения Windows Forms, начните с выбора в диалоговом окне New Project (Новый проект) в Visual Studio шаблона проекта Windows Forms Application (Приложение Windows Forms), как показано на рис. 15.7.

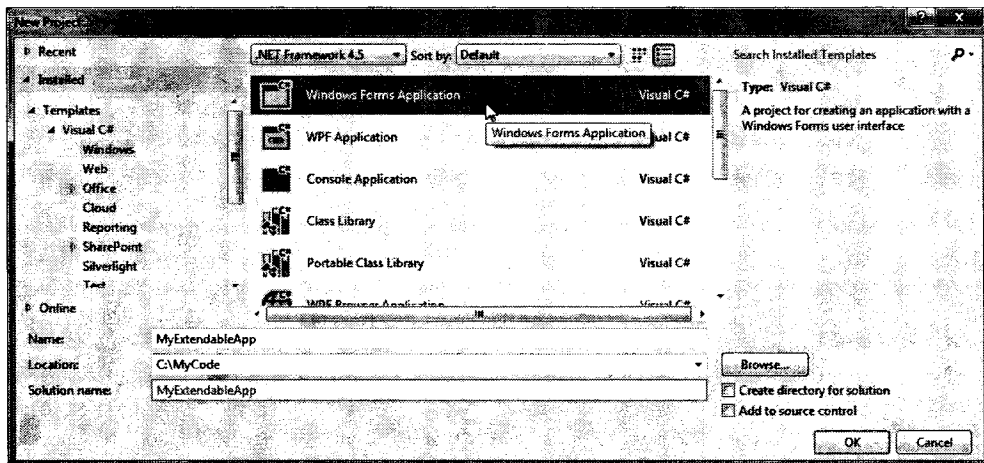


Рис. 15.7. Создание нового проекта Windows Forms в Visual Studio

Теперь установите ссылку на сборку CommonSnappableTypes.dll, но не на библиотеки кода CSharpSnapIn.dll и VbSnapIn.dll. Кроме того, импортируйте в главный файл кода формы (который можно открыть щелчком правой кнопкой мыши на визуальном конструкторе форм и выбором в контекстном меню пункта View Code (Просмотреть код)) пространства имен System.Reflection и CommonSnappableTypes. Вспомните, что основная цель этого приложения заключается в использовании позднего связывания и рефлексии для выяснения “подключаемости” независимых двоичных файлов, созданных другими производителями.

Вдаваться в детали разработки приложения Windows Forms пока не будем (это описано в приложении A), а просто поместим в визуальный конструктор форм компонент MenuStrip и определим для него один элемент меню File (Файл) верхнего уровня с единственным подменю Snap In Module (Подключить модуль). Также добавим в главное окно элемент ListBox (переименовав его в lstLoadedSnapIns), чтобы отображать в нем имена загруженных пользователем оснасток. На рис. 15.8 показан финальный графический пользовательский интерфейс.

Код обработки события Click, генерируемого при выборе пункта меню File⇒Snap In Module (который может быть создан двойным щелчком на этом пункте в визуальном конструкторе форм), должен отображать диалоговое окно File Open (Открытие файла) и извлекать путь к выбранному файлу. Предполагая, что пользователь не выбрал сборку CommonSnappableTypes.dll (поскольку она обеспечивает просто инфраструктуру), этот путь затем передается на обработку вспомогательному методу LoadExternalModule(), которая будет реализована следующей. Этот метод будет возвращать false, если не удастся найти класс, реализующий IAppFunctionality.

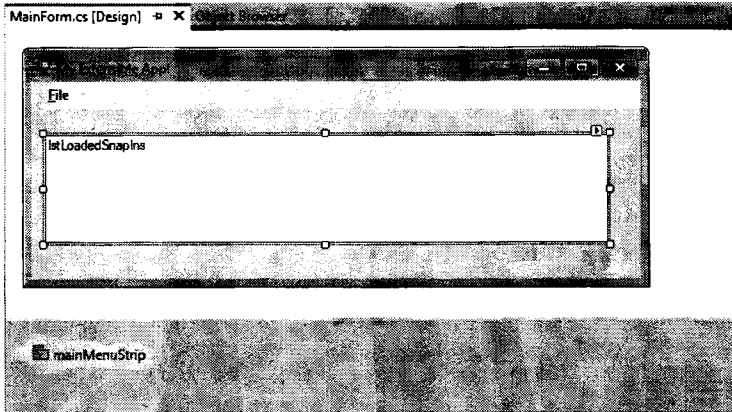


Рис. 15.8. Графический пользовательский интерфейс для приложения MyExtendableApp

```
private void snapInModuleToolStripMenuItem_Click(object sender, EventArgs e)
{
    // Позволить пользователю выбрать сборку для загрузки.
    OpenFileDialog dlg = new OpenFileDialog();
    if (dlg.ShowDialog() == DialogResult.OK)
    {
        if (dlg.FileName.Contains("CommonSnappableTypes"))
            // CommonSnappableTypes не содержит оснасток.
            MessageBox.Show("CommonSnappableTypes has no snap-ins!");
        else if (!LoadExternalModule(dlg.FileName))
            // Не удается обнаружить класс, реализующий IAppFunctionality.
            MessageBox.Show("Nothing implements IAppFunctionality!");
    }
}
```

Метод LoadExternalModule() выполняет следующие действия:

- динамически загружает выбранную сборку в память;
- определяет, содержит ли сборка типы, реализующие IAppFunctionality;
- создает экземпляр типа, используя позднее связывание.

Если тип, реализующий IAppFunctionality, найден, вызывается метод DoIt() и полностью заданное имя типа добавляется в ListBox (обратите внимание, что цикл foreach будет проходить по всем типам в сборке, учитывая возможность наличия в одной сборке нескольких оснасток).

```
private bool LoadExternalModule(string path)
{
    bool foundSnapIn = false;
    Assembly theSnapInAsm = null;
    try
    {
        // Динамически загрузить выбранную сборку.
        theSnapInAsm = Assembly.LoadFrom(path);
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
        return foundSnapIn;
    }
}
```

```
// Получить все совместимые с IAppFunctionality классы в сборке.
var theClassTypes = from t in theSnapInAsm.GetTypes()
                    where t.IsClass &&
                        (t.GetInterface("IAppFunctionality") != null)
                    select t;

// Создать объект и вызвать метод DoIt().
foreach (Type t in theClassTypes)
{
    foundSnapIn = true;

    // Использовать позднее связывание для создания экземпляра типа.
    IAppFunctionality itfApp =
        (IAppFunctionality)theSnapInAsm.CreateInstance(t.FullName, true);
    itfApp.DoIt();
    IstLoadedSnapIns.Items.Add(t.FullName);
}
return foundSnapIn;
}
```

В этом момент можно запустить приложение. В случае выбора сборки CSharpSnapIn.dll или VbSnapIn.dll должно отображаться корректное сообщение. Последней задачей является отображение метаданных, предоставляемых атрибутом [CompanyInfo]. Для этого модифицируем метод LoadExternalModule() так, чтобы в конце тела цикла foreach вызывался новый вспомогательный метод по имени DisplayCompanyData(), принимающий параметр System.Type:

```
private bool LoadExternalModule(string path)
{
    ...
    foreach (Type t in theClassTypes)
    {
        ...
        // Отобразить информацию о компании.
        DisplayCompanyData(t);
    }
    return foundSnapIn;
}
```

Используя этот входной тип, применим рефлексия к атрибуту [CompanyInfo]:

```
private void DisplayCompanyData(Type t)
{
    // Получить данные [CompanyInfo].
    var compInfo = from ci in t.GetCustomAttributes(false) where
                    (ci.GetType() == typeof(CompanyInfoAttribute))
                    select ci;

    // Отобразить данные.
    foreach (CompanyInfoAttribute c in compInfo)
    {
        MessageBox.Show(c.CompanyUrl,
            string.Format("More info about {0} can be found at", c.CompanyName));
    }
}
```

На рис. 15.9 показан результат выполнения приложения.

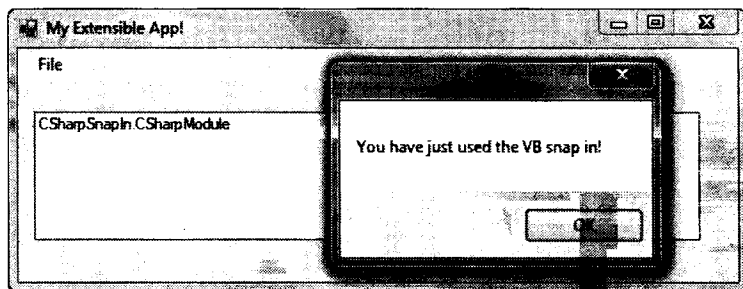


Рис. 15.9. Подключение внешних сборок

На этом создание примера расширяемого приложения завершено. Благодаря этому примеру, должно стать понятно, что представленные в главе технологии могут оказаться весьма полезными в реальности, причем не только для разработчиков специальных инструментов.

Исходный код. Проекты `CommonSnappableTypes`, `CSharpSnapIn`, `VbSnapIn` и `MyExtendableApp` доступны в подкаталоге `ExtendableApp` внутри Chapter 15.

Резюме

Рефлексия является очень интересным аспектом надежной среды объектно-ориентированного программирования. В мире .NET все, что касается служб рефлексии, главным образом связано с классом `System.Type` и пространством имен `System.Reflection`. Как было показано в настоящей главе, рефлексия — это процесс помещения типа во время выполнения под “увеличительное стекло” для получения детальной информации о его характеристиках и возможностях.

Позднее связывание представляет собой процесс создания типа и вызова его членов без предварительного знания имен этих членов. Позднее связывание часто является непосредственным результатом *динамической загрузки*, которая позволяет программно загружать сборку .NET в память. В продемонстрированном в главе примере создания расширяемого приложения было показано, что подобная методика является очень мощной и используется не только разработчиками инструментов, но и их потребителями.

Кроме того, в главе рассказывалось о роли программирования с применением атрибутов. Снабжение типов атрибутами позволяет включать в исходную сборку дополнительные метаданные.

Динамические типы и среда DLR

В версии .NET 4.0 язык C# получил новое ключевое слово `dynamic`. Это ключевое слово позволяет внедрять в строго типизированный мир безопасности к типам, точек с запятой и фигурных скобок поведение, характерное для сценариев. Используя эту слабую типизацию, можно значительно упростить некоторые сложные задачи кодирования и также получить возможность взаимодействия с множеством динамических языков (таких как IronRuby и IronPython), которые поддерживают .NET. В этой главе будет описано ключевое слово `dynamic`, а также показано, как слабо типизированные вызовы отображаются на корректные объекты в памяти, благодаря *исполняющей среде динамического языка* (Dynamic Language Runtime — DLR). После рассмотрения служб, предоставляемых средой DLR, будут приведены примеры использования динамических типов для облегчения вызовов методов с поздним связыванием (через службы рефлексии) и для упрощения взаимодействия с унаследованными библиотеками COM.

На заметку! Не путайте ключевое слово `dynamic` языка C# с концепцией *динамической сборки* (объясняемой в главе 18). Хотя ключевое слово `dynamic` может использоваться при построении динамической сборки, все же это две совершенно независимые концепции.

Роль ключевого слова `dynamic` языка C#

В главе 3 вы узнали о ключевом слове `var`, которое позволяет объявлять локальные переменные таким образом, что их действительные типы данных определяются на основе начального присваивания во время компиляции (это называется *неявной типизацией*). Но как только начальное присваивание выполнено, вы получаете строго типизированную переменную, и любая попытка присвоить ей несовместимое значение приведет к ошибке компиляции.

Чтобы приступить к исследованию ключевого слова `dynamic` языка C#, создадим консольное приложение по имени `DynamicKeyword`. После этого добавим в класс `Program` показанный ниже метод и удостоверимся, что финальный оператор кода действительно инициирует ошибку во время компиляции, если убрать с него символы комментария.

```
static void ImplicitlyTypedVariable()
{
    // Переменная a имеет тип List<int>.
    var a = new List<int>();
    a.Add(90);
    // Это вызовет ошибку во время компиляции!
    // a = "Hello";
}
```

Использование неявной типизации просто потому, что она возможна, многие считают плохим стилем (если известно, что нужен тип `List<int>`, то его и следует объявлять). Однако, как было показано в главе 12, неявная типизация очень полезна в сочетании с LINQ, поскольку многие запросы LINQ возвращают перечисления анонимных классов (через проекции), которые объявить прямо в коде C# не получится. Тем не менее, даже в таких случаях неявно типизированная переменная на самом деле является строго типизированной.

Как уже известно из главы 6, `System.Object` — это самый начальный родительский класс в иерархии классов .NET Framework, который может представлять все, что угодно. После объявления переменной типа `object` получается строго типизированный элемент данных, однако то, на что эта переменная указывает в памяти, может отличаться в зависимости от присваивания ссылки. Для того чтобы получить доступ к членам объекта, на который установлена ссылка в памяти, необходимо выполнять явное приведение.

Предположим, что есть простой класс по имени `Person`, в котором определены два автоматических свойства (`FirstName` и `LastName`), инкапсулирующие `string`. Теперь взгляните на следующий код:

```
static void UseObjectVariable()
{
    // Предположим, что есть класс по имени Person.
    object o = new Person() { FirstName = "Mike", LastName = "Larson" };

    // Для получения доступа к свойствам Person потребуется привести object к Person.
    Console.WriteLine("Person's first name is {0}", ((Person)o).FirstName);
}
```

С выходом версии .NET 4.0 в язык C# было введено ключевое слово `dynamic`. На самом высоком уровне ключевое слово `dynamic` можно рассматривать как специализированную форму `System.Object`, в том смысле, что типу данных `dynamic` может быть присвоено любое значение. На первый взгляд, это может привести к серьезной путанице, поскольку теперь получается, что доступны три способа определения данных, внутренний тип которых явно не указан в кодовой базе. Например, следующий метод:

```
static void PrintThreeStrings()
{
    var s1 = "Greetings";
    object s2 = "From";
    dynamic s3 = "Minneapolis";

    Console.WriteLine("s1 is of type: {0}", s1.GetType());
    Console.WriteLine("s2 is of type: {0}", s2.GetType());
    Console.WriteLine("s3 is of type: {0}", s3.GetType());
}
```

будучи вызванным в `Main()`, выведет на консоль следующее:

```
s1 is of type: System.String
s2 is of type: System.String
s3 is of type: System.String
```

Динамическую переменную от переменной, объявленной неявно или через ссылку на `System.Object`, значительно отличает то, что она *не является строго типизированной*. Другими словами, динамические данные *не типизированы статически*. Для компилятора C# это выглядит так, что элементу данных, объявленному как `dynamic`, может быть присвоено любое начальное значение, а на протяжении времени его существования это значение может быть заменено новым (возможно, не связанным с первоначальным). Рассмотрим показанный ниже метод и его результирующий вывод:

```
static void ChangeDynamicDataType()
{
    // Объявить одиночный элемент данных dynamic по имени t.
    dynamic t = "Hello!";
    Console.WriteLine("t is of type: {0}", t.GetType());

    t = false;
    Console.WriteLine("t is of type: {0}", t.GetType());

    t = new List<int>();
    Console.WriteLine("t is of type: {0}", t.GetType());
}
```

Вот как выглядит вывод:

```
t is of type: System.String
t is of type: System.Boolean
t is of type: System.Collections.Generic.List`1[System.Int32]
```

Имейте в виду, что приведенный выше код успешно бы скомпилировался и дал идентичные результаты, если бы переменная `t` была объявлена с типом `System.Object`. Тем не менее, как вскоре будет показано, ключевое слово `dynamic` предлагает много дополнительных возможностей.

Вызов членов на динамически объявленных данных

Теперь, учитывая, что тип данных `dynamic` может на лету принимать идентичность любого типа (подобно переменной типа `System.Object`), наверняка возник следующий вопрос: а как вызывать члены на динамической переменной (свойства, методы, индексы, регистрации событий и т.п.)? В отношении синтаксиса здесь никаких отличий нет. Нужно просто применить операцию точки к динамической переменной, указать открытый член и передать ему при необходимости аргументы.

Однако (и это очень важно) допустимость указываемых членов компилятором не проверяется! Помните, что в отличие от переменной, объявленной как `System.Object`, динамические данные не являются статически типизированными. Вплоть до времени выполнения не будет известно, поддерживают ли вызываемые динамические данные указанный член, переданы ли корректные параметры, правильно ли указано имя члена, и т.д. Поэтому, как бы странно это не выглядело, следующий метод скомпилируется без ошибок:

```
static void InvokeMembersOnDynamicData()
{
    dynamic textData1 = "Hello";
    Console.WriteLine(textData1.ToUpper());

    // Здесь следовало ожидать ошибку компиляции!
    // Но все компилируется нормально.
    Console.WriteLine(textData1.toupper());
    Console.WriteLine(textData1.Foo(10, "ee", DateTime.Now));
}
```

Обратите внимание, что во втором вызове `WriteLine()` производится попытка обращения к методу по имени `toupper()` на динамическом элементе данных. Как видите, `textData1` имеет тип `string`, и потому известно, что у этого типа отсутствует метод с таким именем в нижнем регистре. Более того, тип `string` определенно не имеет метода по имени `Foo()`, который принимает `int`, `string` и объект `DateTime`!

Тем не менее, компилятор C# ни о каких ошибках не сообщает. Однако если вызвать этот метод в `Main()`, возникнет ошибка времени выполнения с примерно таким сообщением:

Unhandled Exception: Microsoft.CSharp.RuntimeBinder.RuntimeBinderException: 'string' does not contain a definition for 'toupper'

Необработанное исключение: Microsoft.CSharp.RuntimeBinder.RuntimeBinderException: string не содержит определения для toupper

Другое значительное отличие между вызовом членов на динамических и строго типизированных данных состоит в том, что после применения операции точки к элементу динамических данных средство IntelliSense в Visual Studio не активизируется. Вместо этого отображается общее сообщение, показанное на рис. 16.1.

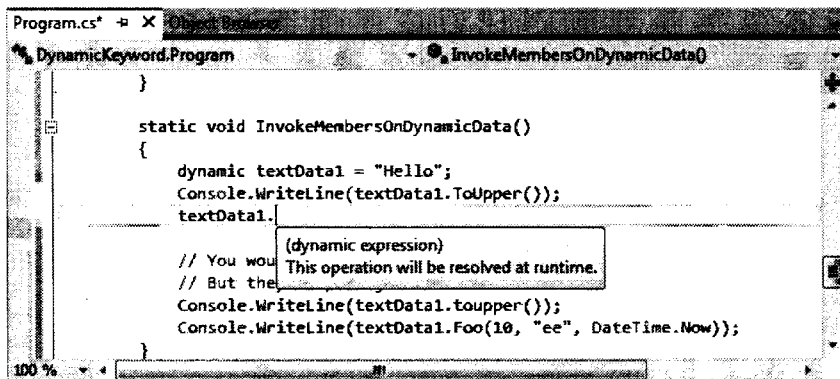


Рис. 16.1. Динамические данные не приводят к активизации средства IntelliSense

То, что средство IntelliSense не доступно для динамических данных, имеет смысл. Однако это означает, что при наборе кода C# с такими переменными следует соблюдать предельную осторожность. Любая опечатка или некорректный регистр символов в имени члена приведет к ошибке времени выполнения, а именно — к генерации исключения типа RuntimeBinderException.

Роль сборки Microsoft.CSharp.dll

Сразу же после создания нового проекта C# в Visual Studio автоматически устанавливается ссылка на сборку по имени Microsoft.CSharp.dll (в этом легко удостовериться, заглянув в папку References (Ссылки) в проводнике решений). В этой очень маленькой библиотеке определено единственное пространство имен (Microsoft.CSharp.RuntimeBinder) с двумя классами (рис. 16.2).

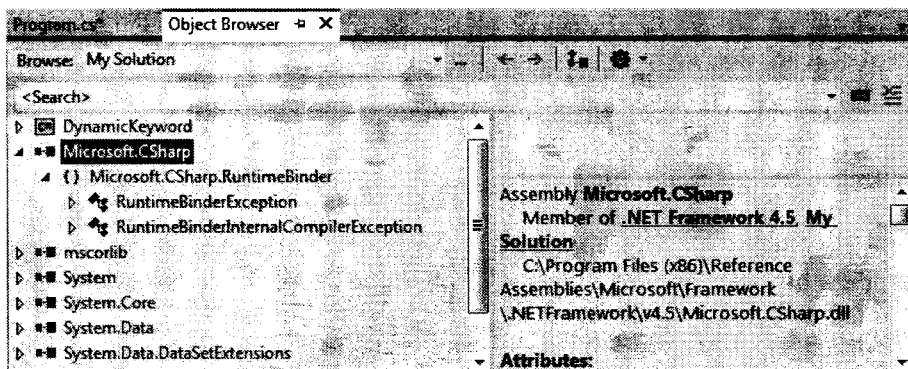


Рис. 16.2. Сборка Microsoft.CSharp.dll

Как можно догадаться по их именам, оба класса — это строго типизированные исключения. Более общий класс, `RuntimeBinderException`, представляет ошибку, которая будет сгенерирована при попытке обращения к несуществующему члену динамического типа данных (как это было в случае методов `toupper()` и `Foo()`). Та же самая ошибка будет инициирована в случае указания некорректных данных параметров для члена, который существует.

Поскольку динамические данные настолько изменчивы, любой вызов члена переменной, объявленной с ключевым словом `dynamic`, может быть помещен в подходящий блок `try/catch` с соответствующей обработкой ошибок, как показано ниже:

```
static void InvokeMembersOnDynamicData()
{
    dynamic textData1 = "Hello";

    try
    {
        Console.WriteLine(textData1.ToUpper());
        Console.WriteLine(textData1.toupper());
        Console.WriteLine(textData1.Foo(10, "ee", DateTime.Now));
    }
    catch (Microsoft.CSharp.RuntimeBinder.RuntimeBinderException ex)
    {
        Console.WriteLine(ex.Message);
    }
}
```

Если вызвать этот метод снова, можно увидеть, что обращение к `ToUpper()` (обратите внимание на заглавные “T” и “U”) работает корректно, однако затем на консоль выводится следующее сообщение об ошибке:

```
HELLO
'string' does not contain a definition for 'toupper'
HELLO
string не содержит определения для toupper
```

Конечно, процесс помещения всех динамических вызовов методов в блоки `try/catch` довольно утомителен. Если вы тщательно следите за написанием кода и передачей параметров, то это делать не обязательно. Тем не менее, перехват исключений удобен, когда заранее не известно, существует ли интересующий член в целевом типе.

Область применения ключевого слова `dynamic`

Вспомните, что неявно типизированные данные (объявленные с ключевым словом `var`) возможны только для локальных переменных в области определения члена. Ключевое слово `var` никогда не может использоваться для возвращаемого значения, параметра или члена класса/структуры. Однако это не касается ключевого слова `dynamic`. Взгляните на следующее определение класса:

```
class VeryDynamicClass
{
    // Поле dynamic.
    private static dynamic myDynamicField;
    // Свойство dynamic.
    public dynamic DynamicProperty { get; set; }
    // Тип возврата dynamic и тип параметра dynamic.
    public dynamic DynamicMethod(dynamic dynamicParam)
    {
        // Локальная переменная dynamic.
        dynamic dynamicLocalVar = "Local variable";
```

```

int myInt = 10;
if (dynamicParam is int)
{
    return dynamicLocalVar;
}
else
{
    return myInt;
}
}

```

Теперь открытые члены можно вызывать ожидаемым образом, однако при оперировании с динамическими методами и свойствами нет полной уверенности в том, каким именно будет тип данных! По правде говоря, определение `VeryDynamicClass` может оказаться не особенно полезным в реальном приложении, но оно иллюстрирует область применения ключевого слова `dynamic`.

Ограничения ключевого слова `dynamic`

Хотя с использованием ключевого слова `dynamic` можно определить очень много вещей, с ним связаны свои ограничения. Хотя они не так уж существенны, имейте в виду, что элементы динамических данных не могут использовать лямбда-выражения или анонимные методы C# при вызове метода. Например, следующий код всегда приводит к ошибкам, даже если целевой метод на самом деле принимает параметр-делегат, который, в свою очередь, принимает значение `string` и возвращает `void`:

```

dynamic a = GetDynamicObject();
// Ошибка! Методы на динамических данных не могут использовать лямбда-выражения!
a.Method(arg => Console.WriteLine(arg));

```

Чтобы обойти это ограничение, придется работать с лежащим в основе делегатом напрямую, используя приемы, описанные в главе 10. Другое ограничение состоит в том, что динамический элемент данных не может воспринимать расширяющие методы (см. главу 11). К сожалению, это касается также всех расширяющих методов из API-интерфейсов LINQ. Таким образом, переменная, объявленная с ключевым словом `dynamic`, имеет очень ограниченное применение в рамках LINQ to Objects и других технологий LINQ:

```

dynamic a = GetDynamicObject();
// Ошибка! Динамические данные не могут найти расширяющий метод Select()!
var data = from d in a select d;

```

Практическое применение ключевого слова `dynamic`

Учитывая тот факт, что динамические данные не являются строго типизированными, не проверяются во время компиляции, не имеют возможности инициировать средство IntelliSense и не могут быть целью запроса LINQ, совершенно корректно предположить, что использование ключевого слова `dynamic` только потому, что оно существует — это очень плохая практика программирования.

Тем не менее, в редких обстоятельствах ключевое слово `dynamic` может радикально сократить объем кода, который придется вводить вручную. В частности, при построении приложения .NET, в котором интенсивно применяется позднее связывание (через рефлексии), ключевое слово `dynamic` может сэкономить время на наборе кода. Аналогично, при разработке приложения .NET, которое должно взаимодействовать с

унаследованными библиотеками COM (такими как продукты Microsoft Office), можно значительно упростить кодовую базу за счет использования ключевого слова `dynamic`.

Как с любым “сокращением”, прежде чем его применять, необходимо взвесить все “за” и “против”. Использование ключевого слова `dynamic` — это компромисс между краткостью кода и безопасностью к типам. Хотя C# в своей основе является строго типизированным языком, можно применять (или не применять) динамическое поведение от вызова к вызову. Помните, что пользоваться ключевым словом `dynamic` вы не обязаны. Тот же самый конечный результат всегда можно получить, написав альтернативный код вручную (правда, обычно существенно большего объема).

Исходный код. Проект `DynamicKeyword` доступен в подкаталоге `Chapter 16`.

Роль исполняющей среды динамического языка (DLR)

Теперь, когда прояснилась суть “динамических данных”, давайте посмотрим, как их обрабатывать. В версии .NET 4.0 общезыковая исполняющая среда (Common Language Runtime — CLR) получила дополнительную среду времени выполнения, которая называется *исполняющей средой динамического языка* (Dynamic Language Runtime — DLR). Концепция “динамической исполняющей среды” определенно не нова. На самом деле ее много лет используют такие языки программирования, как Smalltalk, LISP, Ruby и Python. В своей основе динамическая исполняющая среда предоставляет динамическим языкам возможность обнаруживать типы целиком во время выполнения, без каких-либо проверок на этапе компиляции.

При наличии опыта работы только со строго типизированными языками (включая C# без динамических типов), может показаться неподходящим даже само понятие такой исполняющей среды. В конце концов, обычно лучше получать ошибки во время компиляции, а не во время выполнения, когда это только возможно. Тем не менее, динамические языки и исполняющие среды предлагают ряд интересных возможностей, включая перечисленные ниже.

- Исключительно гибкая кодовая база. Можно проводить рефакторинг кода, не внося многочисленных изменений в типы данных.
- Очень простой способ взаимодействия с разнообразными типами объектов, построенными на разных платформах и языках программирования.
- Способ добавления или удаления членов типа в памяти во время выполнения.

Одна из ролей среды DLR состоит в том, чтобы позволить различным динамическим языкам работать с исполняющей средой .NET и предоставлять им возможность взаимодействия с другим кодом .NET. Два популярных динамических языка, которые используют DLR — это IronPython и IronRuby. Эти языки находятся в “динамической вселенной”, где типы определяются исключительно во время выполнения. К тому же эти языки имеют доступ ко всему богатству библиотек базовых классов .NET. Еще лучше то, что благодаря наличию ключевого слова `dynamic`, их кодовые базы могут взаимодействовать с языком C# (и наоборот).

На заметку! Вопросы применения среды DLR для интеграции с динамическими языками в этой главе не рассматриваются. За необходимыми деталями по этому поводу обращайтесь на веб-сайты IronPython (<http://ironpython.codeplex.com>) и IronRuby (<http://rubyforge.org/projects/ironruby>).

Роль деревьев выражений

Для описания динамического вызова в нейтральных терминах среда DLR использует *деревья выражений*. Например, когда среда DLR встречает код C# вроде следующего:

```
dynamic d = GetSomeData();
d.SuperMethod(12);
```

она автоматически строит дерево выражения, которое, в сущности, гласит: “Вызвать метод по имени SuperMethod на объекте d, передав число 12 в качестве аргумента”. Эта информация (формально называемая *рабочей нагрузкой*) затем передается корректному связывателю времени выполнения, который, опять-таки, может быть динамическим связывателем C#, динамическим связывателем IronPython или даже (как вскоре будет показано) унаследованными объектами COM.

После этого запрос отображается на необходимую структуру вызовов для целевого объекта. Замечательная характеристика деревьев выражений (помимо того факта, что вам не нужно создавать их вручную) состоит в том, что они позволяют написать фиксированный оператор кода C#, не заботясь о его действительной цели (объект COM, кодовая база IronPython или IronRuby и т.п.). На рис. 16.3 представлена высокоуровневая иллюстрация концепции деревьев выражений.

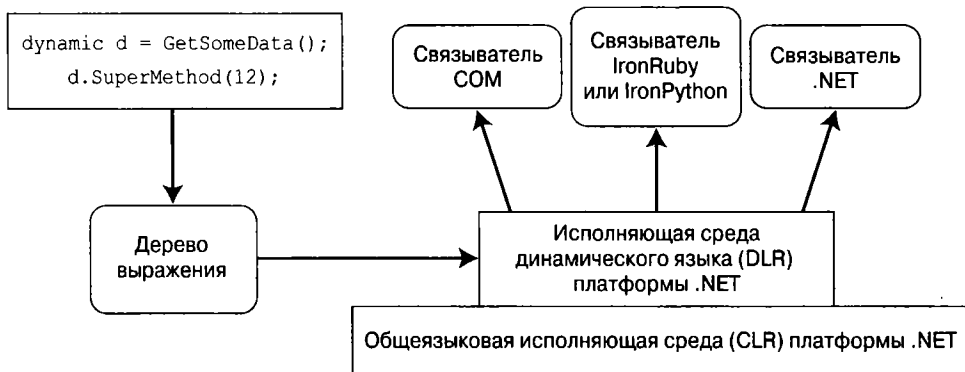


Рис. 16.3. Деревья выражений фиксируют динамические вызовы в нейтральных терминах и обрабатываются связывателями

Роль пространства имен System.Dynamic

Сборка System.Core.dll включает пространство имен под названием System.Dynamic. По правде говоря, шансы, что вам когда-либо придется использовать типы из этого пространства имен, весьма невелики. Однако если вы являетесь разработчиком языка и желаете обеспечить своему динамическому языку возможность взаимодействия со средой DLR, то пространство имен System.Dynamic пригодится для построения специального связывателя времени выполнения.

Подробные сведения о типах из пространства имен System.Dynamic можно найти в документации .NET Framework 4.5 SDK. Для практических нужд просто знайте, что это пространство имен предоставляет необходимую инфраструктуру, которая позволяет обеспечить динамическим языкам взаимодействие с .NET.

Динамический поиск в деревьях выражений во время выполнения

Как уже объяснялось, среда DLR передает деревья выражений целевому объекту, однако на этот процесс оказывает влияние несколько факторов. Если динамический тип данных указывает в памяти на объект COM, то дерево выражения отправляется низкоуровневому интерфейсу COM по имени `IDispatch`. Как вам может быть известно, этот интерфейс представляет собой способ, которым COM внедряет собственный набор динамических служб. Тем не менее, объекты COM могут использоваться в приложении .NET без применения DLR или ключевого слова `dynamic` языка C#. Однако это (как вы удостоверитесь) ведет к более сложному кодированию C#.

Если динамические данные не указывают на объект COM, то дерево выражения может быть передано объекту, реализующему интерфейс `IDynamicObject`. Этот интерфейс используется “за кулисами”, чтобы позволить такому языку, как IronRuby, принять дерево выражения DLR и отобразить его на специфику языка Ruby.

Наконец, если динамические данные указывают на объект, который *не* является объектом COM и *не* реализует интерфейс `IDynamicObject`, то это — нормальный, повседневный объект .NET. В этом случае дерево выражения передается на обработку связывателю исполняющей среды C#. Процесс отображения дерева выражений на специфику .NET включает участие служб рефлексии. После того как дерево выражения обработано определенным связывателем, динамические данные преобразуются в реальный тип данных в памяти, после чего вызывается корректный метод со всеми необходимыми параметрами. Теперь давайте рассмотрим несколько практических применений DLR, начав с упрощения вызовов .NET с поздним связыванием.

Упрощение вызовов с поздним связыванием посредством динамических типов

Одним из случаев, когда имеет смысл использовать ключевое слово `dynamic`, может быть работа со службами рефлексии, а именно — выполнение вызовов методов с поздним связыванием. В главе 15 приводилось несколько примеров, когда такого рода вызовы методов могут быть очень полезны — чаще всего при построении расширяемого приложения. Там было показано, как использовать метод `Activator.CreateInstance()` для создания экземпляра `object`, о котором ничего не известно на этапе компиляции (помимо его отображаемого имени). Затем с помощью типов из пространства имен `System.Reflection` можно обращаться к членам через механизм позднего связывания. Вспомните следующий пример из главы 15:

```
static void CreateUsingLateBinding(Assembly asm)
{
    try
    {
        // Получить метаданные для типа Minivan.
        Type miniVan = asm.GetType("CarLibrary.Minivan");
        // Создать объект Minivan на лету.
        object obj = Activator.CreateInstance(miniVan);
        // Получить информацию для TurboBoost.
        MethodInfo mi = miniVan.GetMethod("TurboBoost");
        // Вызвать метод (null означает отсутствие параметров).
        mi.Invoke(obj, null);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}
```

Хотя этот код работает ожидаемым образом, нельзя не отметить его громоздкость. Здесь приходится вручную использовать класс `MethodInfo`, вручную запрашивать метаданные и т.д. Ниже приведена версия этого метода, в которой применяется ключевое слово `dynamic` и DLR:

```
static void InvokeMethodWithDynamicKeyword(Assembly asm)
{
    try
    {
        // Получить метаданные для типа Minivan.
        Type miniVan = asm.GetType("CarLibrary.Minivan");

        // Создать объект Minivan на лету и вызвать метод.
        dynamic obj = Activator.CreateInstance(miniVan);
        obj.TurboBoost();
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}
```

За счет объявления переменной `obj` с ключевым словом `dynamic` вся рутинная работа, связанная с рефлексией, возлагается на среду DLR.

Использование ключевого слова `dynamic` для передачи аргументов

Польза от DLR становится еще более очевидной, когда нужно выполнять вызовы методов с поздним связыванием, которые принимают параметры. В случае использования обычных вызовов рефлексии аргументы приходится упаковывать в массив экземпляров `object`, который передается методу `Invoke()` класса `MethodInfo`.

Чтобы проиллюстрировать это на примере, создадим новое консольное приложение C# по имени `LateBindingWithDynamic`. Добавим к текущему решению проект типа `Class Library` (Библиотека классов), используя пункт меню `File⇒Add⇒New Project` (Файл⇒Добавить⇒Новый проект) и назовем его `MathLibrary`. Переименуем первоначальный класс `Class1.cs` в проекте `MathLibrary` на `SimplaMath.cs` и реализуем класс, как показано ниже:

```
public class SimpleMath
{
    public int Add(int x, int y)
    {
        return x + y;
    }
}
```

После компиляции сборки `MathLibrary.dll` поместим ее копию в подкаталог `bin\Debug` проекта `LateBindingWithDynamic` (если щелкнуть на кнопке `Show All Files` (Показать все файлы) для каждого проекта в `Solution Explorer`, можно просто перетащить файл из одного проекта в другой). В этот момент окно `Solution Explorer` должно выглядеть примерно так, как показано на рис. 16.4.

На заметку! Помните, что главная цель позднего связывания — позволить приложению создать объект, для которого не предусмотрено записи в манифесте. Именно поэтому нужно вручную скопировать сборку `MathLibrary.dll` в выходную папку консольного проекта, а не устанавливать ссылку на сборку через `Visual Studio`.

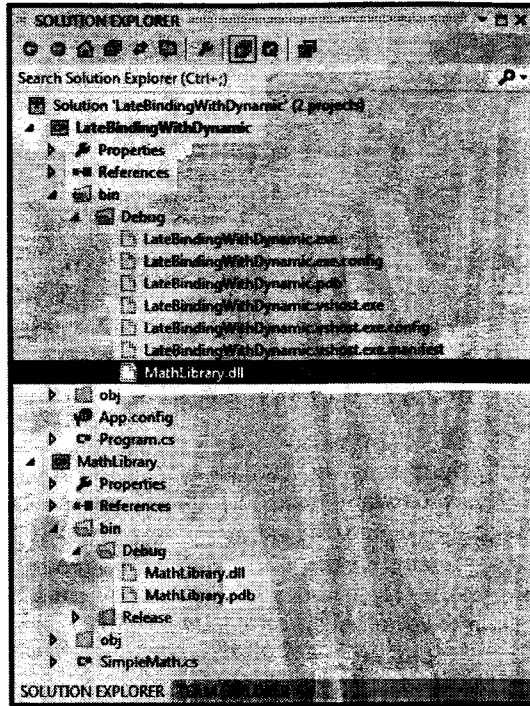


Рис. 16.4. Проект LateBindingWithDynamic имеет закрытую копию сборки MathLibrary.dll

Теперь импортируем пространство имен `System.Reflection` в файл `Program.cs` проекта консольного приложения. Добавим в класс `Program` следующий метод, который вызывает метод `Add()`, используя типичные вызовы API-интерфейса рефлексии:

```
private static void AddWithReflection()
{
    Assembly asm = Assembly.Load("MathLibrary");
    try
    {
        // Получить метаданные для типа SimpleMath.
        Type math = asm.GetType("MathLibrary.SimpleMath");
        // Создать объект SimpleMath на лету.
        object obj = Activator.CreateInstance(math);
        // Получить информацию по методу Add.
        MethodInfo mi = math.GetMethod("Add");
        // Вызвать метод (с параметрами).
        object[] args = { 10, 70 };
        Console.WriteLine("Result is: {0}", mi.Invoke(obj, args));
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}
```

Ниже показано, как можно упростить предыдущую логику метода за счет использования ключевого слова `dynamic`:


```
private static void AddWithDynamic()
{
    Assembly asm = Assembly.Load("MathLibrary");
    try
    {
        // Получить метаданные для типа SimpleMath.
        Type math = asm.GetType("MathLibrary.SimpleMath");
        // Создать объект SimpleMath на лету и вызвать метод.
        dynamic obj = Activator.CreateInstance(math);
        Console.WriteLine("Result is: {0}", obj.Add(10, 70));
    }
    catch (Microsoft.CSharp.RuntimeBinder.RuntimeBinderException ex)
    {
        Console.WriteLine(ex.Message);
    }
}
```

Не так уж плохо! Вызов обоих этих методов в `Main()` даст идентичный вывод. Однако при использовании ключевого слова `dynamic` сокращается объем кодирования. Для динамически определяемых данных больше не нужно вручную упаковывать аргументы в массив экземпляров `object`, запрашивать метаданные сборки либо иметь дело с другими деталями подобного рода. При построении приложения, в котором интенсивно применяется динамическая загрузка и позднее связывание, экономия на кодировании станет со временем еще более ощутимой.

Исходный код. Проект `LastBindingWithDynamic` доступен в подкаталоге `Chapter 16`.

Упрощение взаимодействия с COM посредством динамических данных

Давайте рассмотрим другое полезное использование ключевого слова `dynamic` — в контексте проекта взаимодействия с COM. Если нет опыта разработки для COM, то имейте в виду, что следующий пример, в котором компилируется библиотека COM, содержит метаданные подобно библиотеке .NET. Тем не менее, ее формат совершенно отличается. По этой причине если программа .NET нуждается во взаимодействии с объектом COM, то первое, что потребуется сделать — сгенерировать так называемую *сборку взаимодействия* (которая описана ниже). Делается это довольно легко. Просто откройте диалоговое окно `Add Reference` (Добавить ссылку), перейдите на вкладку `COM` и найдите библиотеку COM, которую необходимо использовать (рис. 16.5).

На заметку! Имейте в виду, что некоторые важные объектные модели Microsoft (включая продукты Office) в настоящее время доступны только через взаимодействие с COM. Таким образом, даже если вы не имеете непосредственного опыта построения приложений COM, может потребоваться потреблять их внутри разрабатываемой программы .NET.

После выбора COM-библиотеки IDE-среда отреагирует генерацией совершенно новой сборки, которая включает описания .NET метаданных COM. Формально она называется *сборкой взаимодействия* и не содержит никакого кода реализации, кроме самого минимума, который помогает транслировать события COM в события .NET. Однако эти сборки взаимодействия очень полезны в том, что защищают кодовую базу .NET от сложностей внутреннего механизма COM.

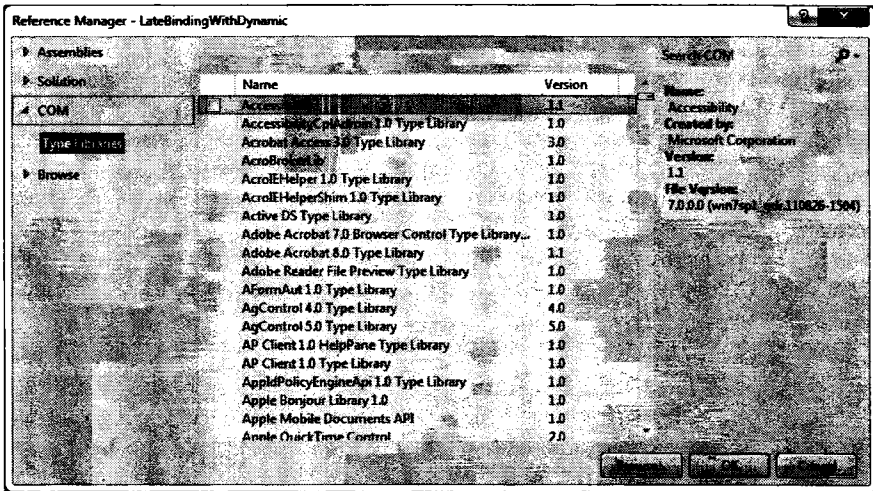


Рис. 16.5. На вкладке COM диалогового окна Add Reference отображаются все зарегистрированные на машине библиотеки COM

В коде C# можно напрямую работать со сборкой взаимодействия, позволяя среде CLR (и среде DLR, если используется ключевое слово `dynamic`) автоматически отображать типы данных .NET на типы COM и наоборот. “За кулисами” данные маршализируются между приложениями .NET и COM с применением вызываемой оболочки времени выполнения (Runtime Callable Wrapper — RCW), которая на самом деле является динамически сгенерированным прокси. Этот прокси RCW будет маршализировать и трансформировать типы данных .NET в типы COM и отображать любые возвращаемые значения COM на их эквиваленты .NET. На рис. 16.6 показана общая картина взаимодействия .NET с COM.

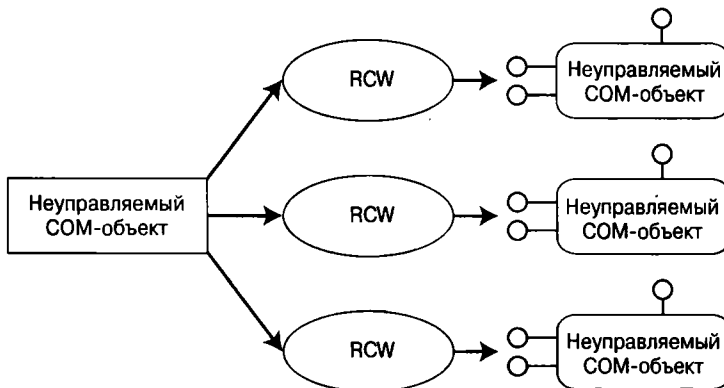


Рис. 16.6. Программы .NET взаимодействуют с объектами COM, используя прокси под названием RCW

Роль основных сборок взаимодействия

Многие библиотеки COM, созданные поставщиками библиотек COM (такие как библиотеки Microsoft COM, обеспечивающие доступ к объектной модели продуктов Microsoft Office), предоставляют “официальную” сборку взаимодействия, которая называется

основной сборкой взаимодействия (primary interop assembly — PIA). Сборки PIA — это оптимизированные сборки взаимодействия, которые делают яснее (и возможно, расширяют) код, обычно генерируемый при добавлении ссылки на библиотеку COM с помощью диалогового окна Add Reference.

Сборки PIA обычно перечислены в разделе Assemblies (Сборки) диалогового окна Add Reference (в области Extensions (Расширения)). В действительности, если вы ссылаетесь на библиотеку COM из вкладки COM диалогового окна Add Reference, то Visual Studio не генерирует новой библиотеки взаимодействия, как делает это обычно, а вместо этого использует предоставленную сборку PIA. На рис. 16.7 показана сборка PIA объектной модели Microsoft Office Excel, которая будет применяться в следующем примере.

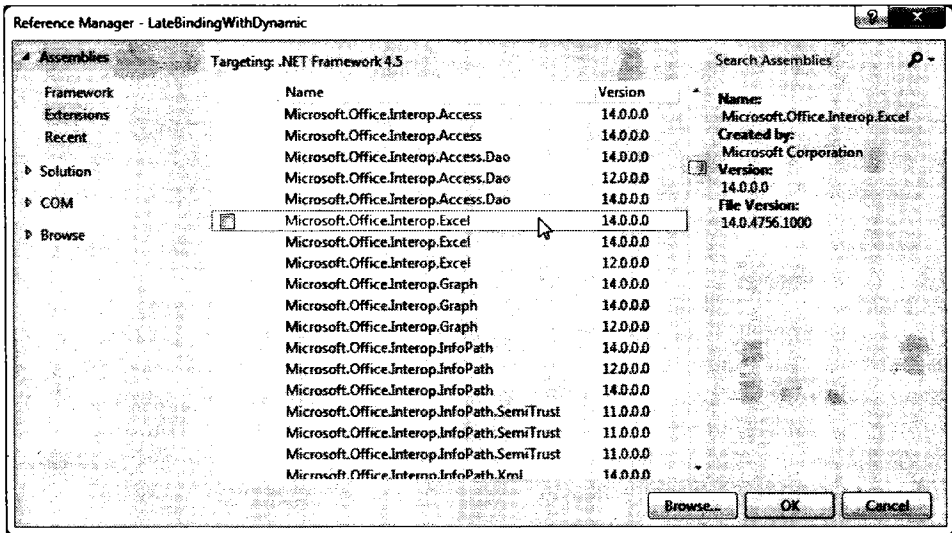


Рис. 16.7. Сборки PIA перечислены в области Extensions раздела Assemblies внутри диалогового окна Add Reference

Встраивание метаданных взаимодействия

До появления .NET 4.0, когда приложение C# использовало библиотеку COM (через сборку PIA или нет), нужно было обеспечить наличие на клиентской машине копии сборки взаимодействия. Помимо того, что увеличивался размер установочного пакета приложения, сценарий установки должен был также проверять действительное наличие сборок PIA, и если они отсутствовали, то устанавливать их копии в GAC.

Однако в .NET 4.0 и последующих версиях теперь можно встраивать данные взаимодействия непосредственно в скомпилированное приложение .NET. В этом случае поставлять копию сборки взаимодействия вместе с приложением .NET больше не требуется, поскольку все необходимые метаданные взаимодействия жестко закодированы в приложении .NET.

По умолчанию после выбора библиотеки COM (PIA или нет) в диалоговом окне Add Reference интегрированная среда разработки автоматически устанавливает свойство Embed Interop Types (Встраивать типы взаимодействия) библиотеки в True. Чтобы увидеть эту настройку, необходимо выбрать ссылаемую сборку взаимодействия в папке References (Ссылки) внутри окна Solution Explorer и открыть ее окно свойств (рис. 16.8).

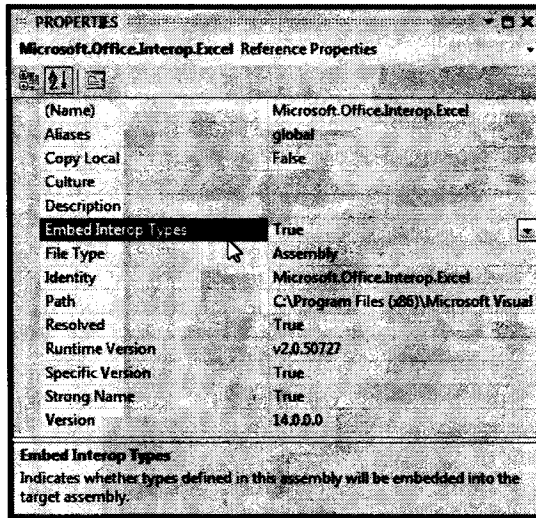


Рис. 16.8. Логика сборки взаимодействия может быть встроена прямо в приложение .NET

Компилятор C# включит только те части библиотеки взаимодействия, которые действительно используются. Таким образом, даже если реальная библиотека взаимодействия содержит .NET-описания сотен объектов COM, будут получены определения только подмножества, которое действительно применяется в написанном коде C#. Помимо сокращения размеров приложения, поставляемого клиенту, также упрощается процесс установки, поскольку не понадобится устанавливать недостающие сборки PIA на целевую машину.

Общие сложности взаимодействия с COM

Давайте перед следующим примером рассмотрим еще одну предварительную тему. До выхода среды DLR при написании кода C#, работающего с библиотекой COM (через сборку взаимодействия), неизбежно возникал ряд сложностей. Например, многие COM-библиотеки определяют методы, принимающие необязательные аргументы, которые вплоть до выхода версии .NET 3.5 в языке C# не поддерживались. Это требовало указания значения `Type.Missing` для каждого вхождения необязательного аргумента. Например, если метод COM принимал пять аргументов, и все они были необязательными, приходилось писать следующий код C#, чтобы принимать стандартные значения:

```
myComObj.SomeMethod(Type.Missing, Type.Missing, Type.Missing,
                    Type.Missing, Type.Missing);
```

К счастью, теперь можно записывать следующий упрощенный код, т.к. значения `Type.Missing` будут вставлены на этапе компиляции, если не указано специфичное значение:

```
myComObj.SomeMethod();
```

В связи с этим стоит отметить, что многие методы COM предоставляют поддержку именованных аргументов, которые, как было показано в главе 4, позволяют передавать значения параметрам в любом порядке. Поскольку язык C# поддерживает это средство, можно очень просто "пропустить" множество необязательных аргументов и устанавливать только те, которые важны в данном случае.

Еще одна общая сложность взаимодействия с COM была связана с тем фактом, что многие методы COM спроектированы так, чтобы принимать и возвращать очень специфический тип данных по имени `Variant`. Во многом похоже на ключевое слово `dynamic` в C#, типу данных `Variant` может быть присвоен любой тип данных COM на лету (строка, ссылка на интерфейс, числовое значение и т.п.). До появления ключевого слова `dynamic` передача или прием элементов данных типа `Variant` требовал значительных ухищрений, обычно связанных с многочисленными операциями приведения.

Когда свойство `Embed Interop Types` установлено в `True`, все типы `Variant` из COM автоматически отображаются на динамические данные. Это не только сокращает потребность в лишних операциях приведения при работе с типами данных `Variant`, но также еще более скрывает некоторые сложности, присущие COM, вроде работы с индексами COM.

Для того чтобы продемонстрировать упрощение взаимодействия с COM за счет совместной работы необязательных аргументов, именованных аргументов и ключевого слова `dynamic` в C#, построим приложение, в котором используется объектная модель `Microsoft Office`. При работе с этим примером вы получите шанс применить новые средства, а также обойтись без них, и затем сравнить объем работ в обоих случаях.

На заметку! Если вы не имеете опыта построения графических пользовательских интерфейсов с помощью `Windows Forms`, можете загрузить предоставленный код решения и работать с ним, а не создавать приложение вручную.

Взаимодействие с COM с использованием динамических данных C#

Предположим, что имеется приложение `Windows Forms` (под названием `ExportDataToOfficeApp`) с графическим пользовательским интерфейсом, главное окно которого содержит элемент управления по имени `dataGridCars`. Это же окно имеет два элемента управления `Button`, один из которых обеспечивает открытие специального диалогового окна для вставки новой строки данных в сетку, а другой отвечает за экспорт данных сетки в электронную таблицу `Excel`. На рис. 16.9 показан завершенный графический пользовательский интерфейс.

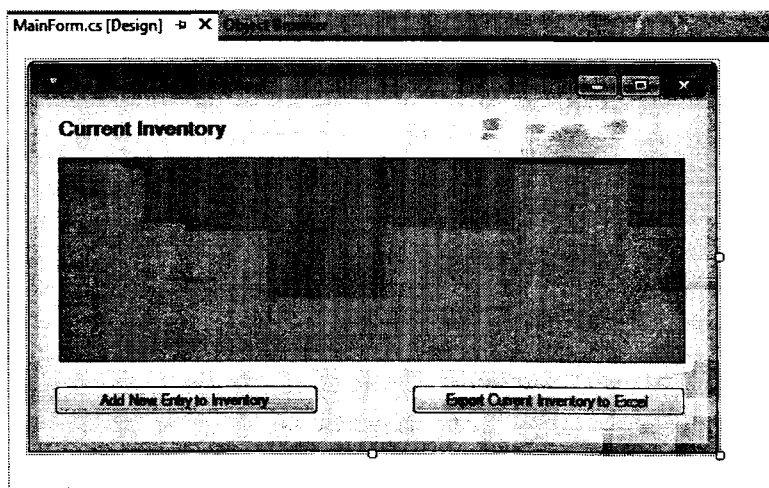


Рис. 16.9. Графический пользовательский интерфейс для примера взаимодействия с COM

Элемент управления DataGridView заполняется некоторыми начальными данными в обработчике события Load формы (класс Car, используемый в качестве параметра типа для обобщенного List<T> — это простой класс в проекте, имеющий свойства Color, Make и PetName):

```
public partial class MainForm : Form
{
    List<Car> carsInStock = null;
    public MainForm()
    {
        InitializeComponent();
    }
    private void MainForm_Load(object sender, EventArgs e)
    {
        carsInStock = new List<Car>
        {
            new Car {Color="Green", Make="VW", PetName="Mary"},
            new Car {Color="Red", Make="Saab", PetName="Mel"},
            new Car {Color="Black", Make="Ford", PetName="Hank"},
            new Car {Color="Yellow", Make="BMW", PetName="Davie"}
        };
        UpdateGrid();
    }
    private void UpdateGrid()
    {
        // Сбросить источник данных.
        dataGridCars.DataSource = null;
        dataGridCars.DataSource = carsInStock;
    }
}
```

В обработчике события Click кнопки Add New Entry to Inventory (Добавить новую запись в инвентарную ведомость) открывается специальное диалоговое окно, которое позволяет пользователю ввести новые данные для объекта Car; после щелчка на кнопке ОК данные добавляются в сетку. (Код этого диалогового окна здесь не показан, поэтому за подробностями обращайтесь к доступному для загрузки решению.) Если хотите повторить пример самостоятельно, включите файлы NewCarDialog.cs, NewCarDialog.designer.cs и NewCarDialog.resx в проект (их можно найти в загружаемом коде примеров). Затем реализуйте обработчик щелчка на кнопке Add New Entry to Inventory, как показано ниже:

```
private void btnAddNewCar_Click(object sender, EventArgs e)
{
    NewCarDialog d = new NewCarDialog();
    if (d.ShowDialog() == DialogResult.OK)
    {
        // Добавить новый автомобиль в список.
        carsInStock.Add(d.theCar);
        UpdateGrid();
    }
}
```

Ядром этого примера является обработчик события Click для кнопки Export Current Inventory to Excel (Экспортировать текущую инвентарную ведомость в Excel). Используя диалоговое окно Add Reference, добавьте ссылку на основную сборку взаимодействия Microsoft.Office.Interop.Excel.dll (как было показано ранее на рис. 16.7). Добавьте приведенный ниже псевдоним пространства имен в главный файл кода формы. Имейте в виду, что при взаимодействии с библиотеками COM псевдоним определять не обяза-

тельно. Однако, поступая так, вы получаете удобный квалификатор для всех импортированных объектов COM, что очень пригодится, если некоторые из этих COM-объектов будут иметь имена, конфликтующие с вашими типами .NET.

```
// Создать псевдоним для объектной модели Excel.
using Excel = Microsoft.Office.Interop.Excel;
```

Реализуйте следующий обработчик события Click, чтобы он вызывал закрытую вспомогательную функцию по имени ExportToExcel():

```
private void btnExportToExcel_Click(object sender, EventArgs e)
{
    ExportToExcel(carsInStock);
}
```

Поскольку библиотека COM была импортирована с помощью Visual Studio, сборка PIA автоматически сконфигурирована так, что используемые метаданные будут встроены в приложение .NET (вспомните о роли свойства Embed Interop Types). Таким образом, все COM-типы Variant будут реализованы как типы данных dynamic. Более того, можно использовать необязательные и именованные аргументы C#. С учетом всего сказанного вот как будет выглядеть реализация ExportToExcel():

```
static void ExportToExcel(List<Car> carsInStock)
{
    // Загрузить Excel, затем создать новую пустую рабочую книгу.
    Excel.Application excelApp = new Excel.Application();
    excelApp.Workbooks.Add();

    // В этом примере используется единственный рабочий лист.
    Excel._Worksheet workSheet = excelApp.ActiveSheet;

    // Установить заголовки столбцов в ячейках.
    workSheet.Cells[1, "A"] = "Make";
    workSheet.Cells[1, "B"] = "Color";
    workSheet.Cells[1, "C"] = "Pet Name";

    // Отобразить все данные в List<Car> на ячейки электронной таблицы.
    int row = 1;
    foreach (Car c in carsInStock)
    {
        row++;
        workSheet.Cells[row, "A"] = c.Make;
        workSheet.Cells[row, "B"] = c.Color;
        workSheet.Cells[row, "C"] = c.PetName;
    }

    // Придать симпатичный вид табличным данным.
    workSheet.Range["A1"].AutoFormat(
        Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);

    // Сохранить файл, завершить Excel и отобразить сообщение пользователю.
    workSheet.SaveAs(string.Format(@"{0}\Inventory.xlsx", Environment.CurrentDirectory));
    excelApp.Quit();
    MessageBox.Show("The Inventory.xlsx file has been saved to your app folder",
        "Export complete!"); // Файл Inventory.xlsx сохранен в папке приложения.
}
```

Метод начинается с загрузки приложения Excel в память, однако на рабочем столе компьютера оно не покажется. В данном приложении интересует только использование объектной модели Excel. Если же необходимо отобразить пользовательский интерфейс Excel, метод понадобится дополнить следующей строкой кода:

```
static void ExportToExcel(List<Car> carsInStock)
{
    // Загрузить Excel, затем создать новую пустую рабочую книгу.
    Excel.Application excelApp = new Excel.Application();

    // Сделать приложение Excel видимым.
    excelApp.Visible = true;

    ...
}
```

После создания пустого рабочего листа добавляются три столбца, названные по именам свойств класса Car. После этого ячейки заполняются данными List<Car> и файл сохраняется под жестко закодированным именем Inventory.xlsx.

Если теперь запустить приложение, добавить несколько записей и экспортировать их в Excel, в подкаталоге bin\Debug приложения Windows Forms появится файл Inventory.xlsx, который можно открыть в приложении Excel (рис. 16.10).

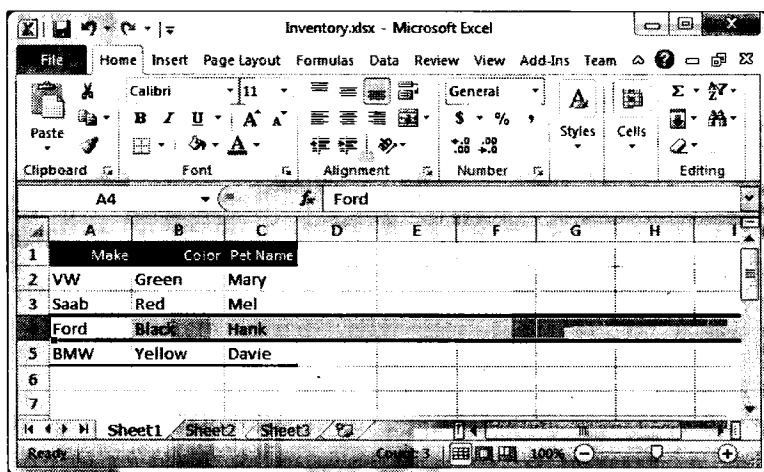


Рис. 16.10. Экспорт данных в файл Excel

Взаимодействие с COM без использования динамических данных C#

Теперь если вы выберете сборку Microsoft.Office.Interop.Excel.dll (в Solution Explorer) и установите ее свойство Embed Interop Type в False, появятся новые сообщения об ошибках компиляции, поскольку COM-данные Variant отныне будут трактоваться не как динамические данные, а как переменные System.Object. Это потребует добавления в ExportToExcel() нескольких явных операций приведения.

Кроме того, если проект скомпилировать для платформы .NET 3.5 или предшествующих версий, утратятся преимущества необязательных/именованных параметров, и в этом случае понадобится явно пометить все пропущенные аргументы. Ниже показана версия метода ExportToExcel(), предназначенная для ранних версий C# (обратите внимание на возросшую сложность кода):

```
static void ExportToExcel2008(List<Car> carsInStock)
{
    Excel.Application excelApp = new Excel.Application();

    // Потребуется пометить пропущенные параметры!
    excelApp.Workbooks.Add(Type.Missing);
```



```

// Потребуется привести Object к _Worksheet!
Excel._Worksheet workSheet = (Excel._Worksheet)excelApp.ActiveSheet;

// Потребуется привести каждый Object к объекту Range
// и затем обратиться к низкоуровневому свойству Value2!
((Excel.Range)excelApp.Cells[1, "A"]).Value2 = "Make";
((Excel.Range)excelApp.Cells[1, "B"]).Value2 = "Color";
((Excel.Range)excelApp.Cells[1, "C"]).Value2 = "Pet Name";
int row = 1;
foreach (Car c in carsInStock)
{
    row++;
    // Потребуется привести каждый Object к объекту Range
    // и обратиться к низкоуровневому свойству Value2!
    ((Excel.Range)workSheet.Cells[row, "A"]).Value2 = c.Make;
    ((Excel.Range)workSheet.Cells[row, "B"]).Value2 = c.Color;
    ((Excel.Range)workSheet.Cells[row, "C"]).Value2 = c.PetName;
}

// Потребуется вызвать метод get_Range и указать все пропущенные аргументы!
excelApp.get_Range("A1", Type.Missing).AutoFormat(
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2,
    Type.Missing, Type.Missing, Type.Missing,
    Type.Missing, Type.Missing, Type.Missing);

// Потребуется указать все пропущенные аргументы!
workSheet.SaveAs(string.Format(@"{0}\Inventory.xlsx", Environment.CurrentDirectory),
    Type.Missing, Type.Missing, Type.Missing, Type.Missing, Type.Missing,
    Type.Missing, Type.Missing, Type.Missing, Type.Missing);

excelApp.Quit();
MessageBox.Show("The Inventory.xlsx file has been saved to your app folder",
    "Export complete!"); // Файл Inventory.xlsx сохранен в папке приложения.
}

```

Хотя конечный результат выполнения этой программы идентичен, очевидно, что данная версия метода намного более многословна. На этом рассмотрение ключевого слова `dynamic` языка C# и среды DLR завершено. Наверняка вы смогли оценить, насколько эти средства могут упростить решение сложных задач программирования, а также (что, пожалуй, даже более важно) понять сопутствующие компромиссы. Делая выбор в пользу динамических данных, вы теряете значительную часть безопасности типов, и код становится уязвимым для гораздо большего числа ошибок времени выполнения.

Несмотря на то что о среде DLR можно еще рассказать многое, основное внимание в этой главе было сосредоточено на темах, которые являются полезными при повседневном программировании. За дополнительной информацией по более сложным вопросам, связанным со средой DLR, обращайтесь в документацию .NET Framework 4.5 SDK.

Исходный код. Проект `ExportDataToOfficeApp` доступен в подкаталоге Chapter 16.

Резюме

Ключевое слово `dynamic`, появившееся в C# 4.0, позволяет определять данные, истинная идентичность которых не известна вплоть до времени выполнения. При обработке новой исполняющей средой динамического языка (DLR) автоматически создаваемое “дерево выражения” будет передаваться соответствующему связывателю динамического языка, причем рабочая нагрузка будет распакована и отправлена правильному члену объекта.

За счет использования динамических данных и DLR многие сложные задачи программирования C# могут быть радикально упрощены; особенно это касается включения библиотек COM в приложения .NET. Кроме того, как было показано в этой главе, платформа .NET 4.0 и последующих версий предлагает ряд дальнейших упрощений взаимодействия с COM (которые не имеют отношения к динамическим данным), в том числе встраивание данных взаимодействия COM в разрабатываемые приложения, необязательные аргументы и именованные аргументы.

Хотя все эти средства, несомненно, могут упростить код, не забывайте, что динамические данные существенно снижают безопасность кода C# в отношении типов и создают возможности для возникновения ошибок времени выполнения. Поэтому тщательно взвешивайте все “за” и “против” использования динамических данных в проектах C# и соответствующим образом проверяйте их!

Процессы, домены приложений и объектные контексты

В главах 14 и 15 были представлены шаги, предпринимаемые средой CLR для определения местонахождения ссылаемых внешних сборок, а также описана роль метаданных .NET. В этой главе вы узнаете, как среда CLR размещает сборки. Вдобавок будут описаны отношения между процессами, доменами приложений и объектными контекстами.

В сущности, *домены приложений* (Application Domain — AppDomain) — это логические подразделы внутри отдельного процесса, в котором размещается набор связанных сборок .NET. Как здесь будет показано, каждый домен приложения, в свою очередь, делится на *контекстные границы*, которые используются для группирования вместе подобных объектов .NET. Понятие контекста позволяет среде CLR обеспечивать надлежащую обработку объектов с особыми потребностями времени выполнения.

Несмотря на то что многие повседневные задачи программирования могут не требовать непосредственного взаимодействия с процессами, доменами приложений или объектными контекстами, понимание этих аспектов очень важно при работе с многочисленными API-интерфейсами .NET, включая Windows Communication Foundation (WCF), многопоточную и параллельную обработку и сериализацию объектов.

Роль процесса Windows

Понятие “процесса” существовало в операционных системах Windows задолго до появления платформы .NET. Выражаясь простыми терминами, *процесс* — это выполняющаяся программа. Тем не менее, формально процесс определяется как концепция уровня операционной системы, которая используется для описания набора ресурсов (таких как внешние библиотеки кода и главный поток) и необходимой памяти, выделяемой выполняющимся приложением. Для каждого загруженного в память файла *.exe операционная система создает отдельный изолированный процесс, который применяется на протяжении всего времени жизни.

Результатом такой изоляции приложений является гораздо более надежная и стабильная исполняющая среда, поскольку сбой одного процесса не влияет на функционирование других процессов. Более того, данные одного процесса не доступны напрямую другим процессам, если только не используется API-интерфейс для программирования распределенных вычислений, подобный Windows Communication Foundation. С учетом

этих моментов, процесс можно рассматривать как фиксированную и безопасную границу для выполняющегося приложения.

Теперь каждому процессу Windows назначается уникальный идентификатор процесса (process identifier — PID) и он может независимо загружаться и выгружаться операционной системой при необходимости (а также и программно). Как вам уже наверняка известно, в окне диспетчера задач Windows (открываемом нажатием комбинации клавиш <Ctrl+Shift+Esc>) имеется вкладка Processes (Процессы), на которой можно просматривать различные статические данные о выполняющихся на машине процессах, в том числе их PID и имена образов (рис. 17.1).

На заметку! По умолчанию столбец PID (Идентификатор процесса) на вкладке Processes не отображается. Для его включения необходимо выбрать пункт меню View⇒Select Columns (Вид⇒Выбрать столбцы) и отметить флажок PID (Process Identifier) (ИД процесса (PID)).

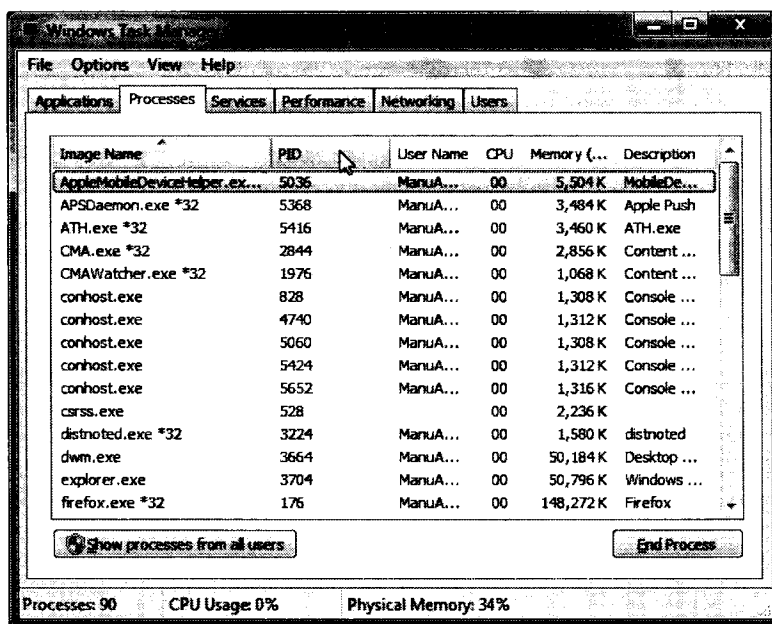


Рис. 17.1. Диспетчер задач Windows

Роль потоков

В каждом процессе Windows содержится начальный “поток”, который функционирует в качестве входной точки для приложения. Детали построения многопоточных приложений на платформе .NET будут рассматриваться в главе 19; тем не менее, для понимания излагаемого здесь материала необходимо ознакомиться с несколькими рабочими определениями. Прежде всего, *поток* — это путь выполнения внутри процесса. Формально выражаясь, первый поток, созданный точкой входа процесса, называется *главным потоком*. В любой исполняемой программе .NET (консольном приложении, приложении Windows Forms, приложении WPF и т.д.) точка входа помечается с помощью метода `Main()`. При вызове этого метода главный поток создается автоматически.

Процессы, которые содержат единственный главный поток выполнения, изначально являются безопасными к потокам, поскольку в каждый момент времени доступ к

данным приложения может получать только один поток. Однако однопоточный процесс (особенно с графическим пользовательским интерфейсом) часто замедленно реагирует на действия пользователя, когда его единственный поток выполняет сложную операцию (такую как печать длинного текстового файла, сложные математические вычисления или подключение к удаленному серверу).

С учетом такого потенциального недостатка однопоточных приложений, API-интерфейс Windows (а также платформа .NET) предоставляет главному потоку возможность порождения дополнительных вторичных потоков (называемых *рабочими потоками*) с применением функций из API-интерфейса Windows, таких как `CreateThread()`. Каждый поток (первичный или вторичный) становится уникальным путем выполнения в процессе и может параллельно получать доступ ко всем разделяемым элементам данных внутри этого процесса.

Как не трудно догадаться, разработчики обычно создают дополнительные потоки для улучшения общей степени отзывчивости программы. Многопоточные процессы обеспечивают иллюзию того, что выполнение многочисленных действий происходит более или менее одновременно. Например, приложение может порождать дополнительный рабочий поток для выполнения трудоемкой порции работы (вроде вывода на печать крупного текстового файла). После запуска вторичного потока главный поток по-прежнему реагирует на пользовательский ввод, что дает всему процессу возможность обеспечивать более высокую производительность. Тем не менее, в действительности так происходит не всегда: использование слишком большого количества потоков в одном процессе может привести к *ухудшению* производительности из-за того, что центральный процессор должен переключаться между активными потоками внутри процесса (что требует времени).

На некоторых машинах многопоточность по большей части представляет собой иллюзию, обеспечиваемую операционной системой. Машины с единственным (не поддерживающим гиперпотоки) центральным процессором не имеют возможности обрабатывать множество потоков в точности в одно и то же время. Вместо этого один центральный процессор выполняет по одному потоку за единицу времени (называемую *квантом времени*), основываясь отчасти на приоритете потока. По истечении выделенного кванта времени выполнение существующего потока приостанавливается, давая другому потоку возможность выполнить свою работу. Чтобы поток не забывал, что происходило до того, как его выполнение было приостановлено, ему предоставляется возможность записывать данные в локальное хранилище потоков (Thread Local Storage — TLS) и выделяется отдельный стек вызовов, как показано на рис. 17.2.

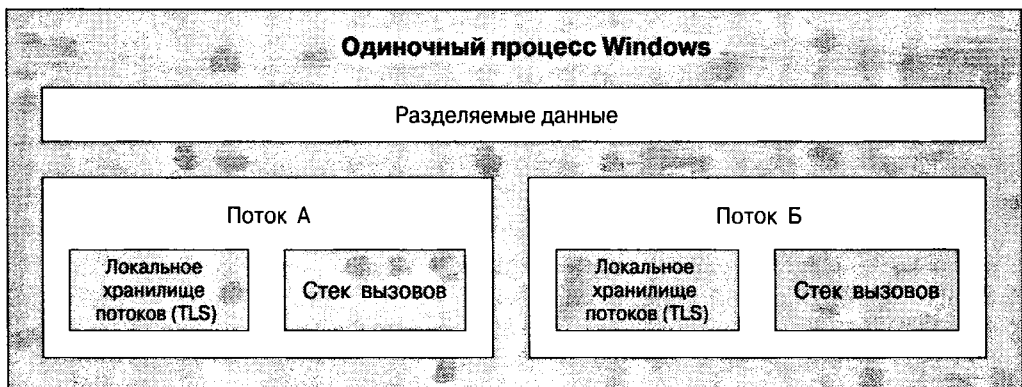


Рис. 17.2. Отношения между потоками и процессами в Windows

Если тема потоков является для вас новой, не стоит беспокоиться по поводу деталей. На данном этапе главное запомнить, что любой поток представляет собой просто уникальный путь выполнения внутри процесса Windows. Каждый процесс имеет главный поток (созданный в точке входа приложения) и может содержать дополнительные потоки, создаваемые программным образом.

Взаимодействие с процессами на платформе .NET

Хотя в самих процессах и потоках нет ничего нового, способ, которым с ними можно взаимодействовать в рамках платформы .NET, довольно сильно изменился (в лучшую сторону). Чтобы проложить оптимальный путь к пониманию приемов построения многопоточных сборок (о которых речь пойдет в главе 19), начнем с того, что посмотрим, каким образом можно взаимодействовать с процессами, используя библиотеки базовых классов .NET.

В пространстве имен `System.Diagnostics` определено несколько типов, которые позволяют программно взаимодействовать с процессами и различными связанными с диагностикой средствами, такими как журнал событий системы и счетчики производительности. В настоящей главе нас интересуют только типы, связанные с процессами, которые перечислены в табл. 17.1.

Таблица 17.1. Избранные типы пространства имен `System.Diagnostics`

Тип	Описание
<code>Process</code>	Этот класс предоставляет доступ к локальным и удаленным процессам, а также позволяет программно запускать и останавливать процессы
<code>ProcessModule</code>	Этот тип представляет модуль (*.dll или *.exe), загруженный в определенный процесс. Важно понимать, что тип <code>ProcessModule</code> может представлять любой модуль, т.е. двоичные сборки на основе COM, .NET или традиционного C
<code>ProcessModuleCollection</code>	Этот тип предоставляет строго типизированную коллекцию объектов <code>ProcessModule</code>
<code>ProcessStartInfo</code>	Этот тип позволяет указывать набор значений, используемых при запуске процесса с помощью метода <code>Process.Start()</code>
<code>ProcessThread</code>	Этот тип представляет поток в заданном процессе. Следует иметь в виду, что тип <code>ProcessThread</code> применяется для диагностики набора потоков процесса, но не для порождения новых потоков выполнения внутри процесса
<code>ProcessThreadCollection</code>	Этот тип предоставляет строго типизованную коллекцию объектов <code>ProcessThread</code>

Класс `System.Diagnostics.Process` позволяет анализировать процессы, выполняемые на заданной машине (локальной или удаленной). Класс `Process` также предоставляет члены, которые позволяют программно запускать и завершать процессы, просматривать (или модифицировать) уровень приоритета процесса и получать список активных потоков и/или загруженных модулей внутри конкретного процесса. В табл. 17.2 перечислены некоторые основные свойства класса `System.Diagnostics.Process`.

Таблица 17.2. Избранные свойства класса `Process`

Свойство	Описание
<code>ExitTime</code>	Это свойство позволяет извлекать метку времени, ассоциированную с процессом, который был завершен (представляется с помощью типа <code>DateTime</code>)
<code>Handle</code>	Это свойство возвращает дескриптор (представляемый с помощью <code>IntPtr</code>), который был назначен процессу операционной системой. Может быть полезным при построении приложений .NET, нуждающихся во взаимодействии с неуправляемым кодом
<code>Id</code>	Это свойство позволяет получать идентификатор (PID) соответствующего процесса
<code>MachineName</code>	Это свойство позволяет получать имя компьютера, на котором выполняется соответствующий процесс
<code>MainWindowTitle</code>	Это свойство позволяет получать заголовок главного окна процесса (если у процесса нет главного окна, возвращается пустая строка)
<code>Modules</code>	Это свойство позволяет получать доступ к строго типизированной коллекции <code>ProcessModuleCollection</code> , представляющей набор модулей (*.dll или *.exe), которые были загружены внутри текущего процесса
<code>ProcessName</code>	Это свойство позволяет получать имя процесса (которое, как и можно было предполагать, совпадает с именем самого приложения)
<code>Responding</code>	Это свойство позволяет получать значение, показывающее, реагирует ли пользовательский интерфейс процесса на пользовательский ввод (или в текущий момент находится в “зависшем” состоянии)
<code>StartTime</code>	Это свойство позволяет получать значение времени, когда был запущен процесс (представленное типом <code>DateTime</code>)
<code>Threads</code>	Это свойство позволяет получать набор потоков, выполняемых в заданном процессе (представленный с помощью коллекции объектов <code>ProcessThread</code>)

Помимо перечисленных выше свойств, в классе `System.Diagnostics.Process` определено несколько полезных методов (табл. 17.3).

Таблица 17.3. Избранные методы класса `Process`

Метод	Описание
<code>CloseMainWindow()</code>	Этот метод закрывает процесс, который имеет пользовательский интерфейс, отправляя его главному окну сообщение о закрытии
<code>GetCurrentProcess()</code>	Этот статический метод возвращает новый объект <code>Process</code> , представляющий процесс, который является активным в текущий момент
<code>GetProcesses()</code>	Этот статический метод возвращает массив новых объектов <code>Process</code> , которые выполняются на заданной машине
<code>Kill()</code>	Этот метод немедленно останавливает заданный процесс
<code>Start()</code>	Этот метод запускает процесс

Перечисление выполняющихся процессов

В целях иллюстрации манипулирования объектами `Process` создадим новое консольное приложение C# по имени `ProcessManipulator` и определим следующий вспомогательный статический метод в классе `Program` (не забудьте импортировать в файл кода пространство имен `System.Diagnostics`):

```
static void ListAllRunningProcesses()
{
    // Получить все процессы на локальной машине, упорядоченные по PID.
    var runningProcs =
        from proc in Process.GetProcesses(".") orderby proc.Id select proc;
    // Вывести для каждого процесса PID и имя.
    foreach (var p in runningProcs)
    {
        string info = string.Format("-> PID: {0}\tName: {1}",
            p.Id, p.ProcessName);
        Console.WriteLine(info);
    }
    Console.WriteLine("*****\n");
}
```

Статический метод `Process.GetProcesses()` возвращает массив объектов `Process`, которые представляют выполняющиеся процессы на целевой машине (используемая здесь нотация в виде точки обозначает локальный компьютер). После получения массива объектов `Process` можно обращаться к любому из членов, описанных в табл. 17.2 и 17.3. В примере просто выводятся идентификаторы PID и имена всех процессов, упорядоченные по PID. Добавив в `Main()` вызов метода `ListAllRunningProcesses()`:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Processes *****\n");
    ListAllRunningProcesses();
    Console.ReadLine();
}
```

можно будет увидеть список имен и идентификаторов PID всех процессов, выполняющихся на локальном компьютере. Ниже приведен пример части вывода:

```
***** Fun with Processes *****
-> PID: 0      Name: Idle
-> PID: 4      Name: System
-> PID: 108     Name: iexplore
-> PID: 268     Name: smss
-> PID: 432     Name: csrss
-> PID: 448     Name: svchost
-> PID: 472     Name: wininit
-> PID: 504     Name: csrss
-> PID: 536     Name: winlogon
-> PID: 560     Name: services
-> PID: 584     Name: lsass
-> PID: 592     Name: lsm
-> PID: 660     Name: devenv
-> PID: 684     Name: svchost
-> PID: 760     Name: svchost
-> PID: 832     Name: svchost
-> PID: 844     Name: svchost
-> PID: 856     Name: svchost
-> PID: 900     Name: svchost
```



```
-> PID: 924      Name: svchost
-> PID: 956      Name: VMWareService
-> PID: 1116     Name: spoolsv
-> PID: 1136     Name: ProcessManipulator.vshost
*****
```

Исследование конкретного процесса

В дополнение к полному списку всех выполняющихся процессов на конкретной машине, статический метод `Process.GetProcessById()` позволяет получать одиночный объект `Process` по ассоциированному PID. В случае запроса несуществующего PID генерируется исключение `ArgumentException`. Например, чтобы получить объект `Process`, представляющий процесс с PID, равным 987, можно написать следующий код:

```
// Если процесс с PID, равным 987, не существует, сгенерируется исключение.
static void GetSpecificProcess()
{
    Process theProc = null;
    try
    {
        theProc = Process.GetProcessById(987);
    }
    catch(ArgumentException ex)
    {
        Console.WriteLine(ex.Message);
    }
}
```

К этому моменту вы уже знаете, как получить список всех процессов или конкретный процесс на машине, выполняя поиск по PID. Класс `Process` также позволяет просмотреть набор текущих потоков и библиотек, используемых внутри заданного процесса. Давайте посмотрим, как это делается.

Исследование набора потоков процесса

Набор потоков представлен с помощью строго типизованной коллекции `Process.ThreadCollection`, в которой содержится определенное количество отдельных объектов `ProcessThread`. Для примера предположим, что в текущее приложение добавлен следующий вспомогательный статический метод:

```
static void EnumThreadsForPid(int pID)
{
    Process theProc = null;
    try
    {
        theProc = Process.GetProcessById(pID);
    }
    catch(ArgumentException ex)
    {
        Console.WriteLine(ex.Message);
        return;
    }

    // Вывести статистические данные по каждому потоку в указанном процессе.
    Console.WriteLine("Here are the threads used by: {0}",
        theProc.ProcessName);
    ProcessThreadCollection theThreads = theProc.Threads;
    foreach(ProcessThread pt in theThreads)
    {
```

```

string info =
    string.Format("-> Thread ID: {0}\tStart Time: {1}\tPriority: {2}",
        pt.Id, pt.StartTime.ToShortTimeString(), pt.PriorityLevel);
Console.WriteLine(info);
}
Console.WriteLine("*****\n");
}

```

Как видите, свойство `Threads` в типе `System.Diagnostics.Process` предоставляет доступ к классу `ProcessThreadCollection`. Здесь для каждого потока в указанном клиентом процессе выводится назначенный идентификатор потока, время запуска и уровень приоритета. Давайте теперь обновим метод `Main()` в классе `Program` так, чтобы он запрашивал у пользователя PID интересующего процесса:

```

static void Main(string[] args)
{
    ...
    // Запросить у пользователя PID и вывести набор активных потоков.
    Console.WriteLine("***** Enter PID of process to investigate *****");
    Console.Write("PID: ");
    string pID = Console.ReadLine();
    int theProcID = int.Parse(pID);

    EnumThreadsForPid(theProcID);
    Console.ReadLine();
}

```

Запустив приложение, можно ввести PID любого процесса на машине и просмотреть используемые внутри него потоки. Ниже приведен пример вывода потоков, используемых в процессе с PID, равным 108, который, так случилось, отвечает за обслуживание Microsoft Internet Explorer:

```

***** Enter PID of process to investigate *****
PID: 108
Here are the threads used by: iexplore
-> Thread ID: 680      Start Time: 9:05 AM      Priority: Normal
-> Thread ID: 2040     Start Time: 9:05 AM      Priority: Normal
-> Thread ID: 880      Start Time: 9:05 AM      Priority: Normal
-> Thread ID: 3380     Start Time: 9:05 AM      Priority: Normal
-> Thread ID: 3376     Start Time: 9:05 AM      Priority: Normal
-> Thread ID: 3448     Start Time: 9:05 AM      Priority: Normal
-> Thread ID: 3476     Start Time: 9:05 AM      Priority: Normal
-> Thread ID: 2264     Start Time: 9:05 AM      Priority: Normal
-> Thread ID: 2380     Start Time: 9:05 AM      Priority: Normal
-> Thread ID: 2384     Start Time: 9:05 AM      Priority: Normal
-> Thread ID: 2308     Start Time: 9:05 AM      Priority: Normal
-> Thread ID: 3096     Start Time: 9:07 AM      Priority: Highest
-> Thread ID: 3600     Start Time: 9:45 AM      Priority: Normal
-> Thread ID: 1412     Start Time: 10:02 AM     Priority: Normal

```

Помимо `Id`, `StartTime` и `PriorityLevel`, класс `ProcessThread` имеет дополнительные члены, наиболее интересные из которых перечислены в табл. 17.4.

Прежде чем двигаться дальше, необходимо четко уяснить, что тип `ProcessThread` не является сущностью, применяемой для создания, приостановки или уничтожения потоков на платформе .NET. Этот тип скорее представляет собой средство, позволяющее получать диагностическую информацию по активным потокам Windows внутри выполняющегося процесса. Более подробные сведения о том, как создавать многопоточные приложения с применением пространства имен `System.Threading`, будут приведены в главе 19.

Таблица 17.4. Избранные члены класса `ProcessThread`

Член	Описание
<code>CurrentPriority</code>	Получает информацию о текущем приоритете потока
<code>Id</code>	Получает уникальный идентификатор потока
<code>IdealProcessor</code>	Устанавливает предпочитаемый процессор для выполнения заданного потока
<code>PriorityLevel</code>	Получает или устанавливает уровень приоритета потока
<code>ProcessorAffinity</code>	Устанавливает процессоры, на которых может выполняться соответствующий поток
<code>StartAddress</code>	Получает адрес памяти функции, вызванной операционной системой, которая запустила данный поток
<code>StartTime</code>	Получает время, когда операционная система запустила поток
<code>ThreadState</code>	Получает текущее состояние данного потока
<code>TotalProcessorTime</code>	Получает общее время, потраченное данным потоком на использование процессора
<code>WaitReason</code>	Получает причину, по которой поток находится в состоянии ожидания

Исследование набора модулей процесса

Теперь давайте посмотрим, как реализовать проход по загруженным модулям, которые размещены внутри конкретного процесса. Когда речь идет о процессах, *модуль* — это общий термин, используемый для описания заданной сборки *.dll (или *.exe), которая обслуживается определенным процессом. При доступе к коллекции `ProcessModuleCollection` через свойство `Process.Modules` можно пройти по *всем модулям*, размещенным внутри процесса: библиотекам на основе .NET, COM и традиционному C. Для примера создадим показанную ниже дополнительную вспомогательную функцию, которая будет перечислять модули в конкретном процессе, заданном PID:

```
static void EnumModsForPid(int pID)
{
    Process theProc = null;
    try
    {
        theProc = Process.GetProcessById(pID);
    }
    catch (ArgumentException ex)
    {
        Console.WriteLine(ex.Message);
        return;
    }

    Console.WriteLine(«Here are the loaded modules for: {0}»,
        theProc.ProcessName);
    ProcessModuleCollection theMods = theProc.Modules;
    foreach (ProcessModule pm in theMods)
    {
        string info = string.Format(«-> Mod Name: {0}», pm.ModuleName);
        Console.WriteLine(info);
    }
    Console.WriteLine("*****\n");
}
```

Чтобы получить некоторый вывод, давайте посмотрим загружаемые модули для процесса, размещающего программу текущего примера (ProcessManipulator). Для этого необходимо запустить приложение, выяснить PID, назначенный ProcessManipulator.exe (с помощью диспетчера задач), и передать это значение методу EnumModsForPid() (соответствующим образом обновив метод Main()). Как оказалось, с этим довольно простым консольным приложением связан внушительный список библиотек *.dll (GDI32.dll, USER32.dll, ole32.dll и т.д.):

```
Here are the loaded modules for: ProcessManipulator
-> Mod Name: ProcessManipulator.exe
-> Mod Name: ntdll.dll
-> Mod Name: MSCOREEE.DLL
-> Mod Name: KERNEL32.dll
-> Mod Name: KERNELBASE.dll
-> Mod Name: ADVAPI32.dll
-> Mod Name: msvcrt.dll
-> Mod Name: sechost.dll
-> Mod Name: RPCRT4.dll
-> Mod Name: SspiCli.dll
-> Mod Name: CRYPTBASE.dll
-> Mod Name: mscoreei.dll
-> Mod Name: SHLWAPI.dll
-> Mod Name: GDI32.dll
-> Mod Name: USER32.dll
-> Mod Name: LPK.dll
-> Mod Name: USP10.dll
-> Mod Name: IMM32.DLL
-> Mod Name: MSCTF.dll
-> Mod Name: clr.dll
-> Mod Name: MSVCR100_CLR0400.dll
-> Mod Name: mscorlib.ni.dll
-> Mod Name: nlssorting.dll
-> Mod Name: ole32.dll
-> Mod Name: clrjit.dll
-> Mod Name: System.ni.dll
-> Mod Name: System.Core.ni.dll
-> Mod Name: psapi.dll
-> Mod Name: shfolder.dll
-> Mod Name: SHELL32.dll
*****
```

Запуск и останов процессов программным образом

Финальными аспектами класса System.Diagnostics.Process, которые мы здесь рассмотрим, являются методы Start() и Kill(). Эти методы позволяют, соответственно, программно запускать и завершать процесс. В качестве примера создадим вспомогательный статический метод StartAndKillProcess(), код которого показан ниже.

На заметку! Для того чтобы запускать новые процессы, среда Visual Studio должна выполняться с правами администратора. В противном случае во время выполнения возникает ошибка.

```
static void StartAndKillProcess()
{
    Process ieProc = null;

    // Запустить Internet Explorer и перейти на сайт facebook.com.
    try
    {
```

```

    ieProc = Process.Start("IExplore.exe", "www.facebook.com");
}
catch (InvalidOperationException ex)
{
    Console.WriteLine(ex.Message);
}

Console.Write("--> Hit enter to kill {0}...", ieProc.ProcessName);
Console.ReadLine();

// Уничтожить процесс iexplore.exe.
try
{
    ieProc.Kill();
}
catch (InvalidOperationException ex)
{
    Console.WriteLine(ex.Message);
}
}

```

Статический метод `Process.Start()` имеет несколько перегруженных версий. Как минимум, необходимо указывать дружественное имя запускаемого процесса (такое как `iexplore.exe` в случае `Microsoft Internet Explorer`). В приведенном примере используется версия метода `Start()`, которая позволяет задавать любое количество дополнительных аргументов, подлежащих передаче в точку входа программы (т.е. в метод `Main()`).

После вызова метода `Start()` возвращается ссылка на новый запущенный процесс. Чтобы завершить этот процесс, потребуется вызывать метод `Kill()` уровня экземпляра. Вызовы `Start()` и `Kill()` помещены внутрь блока `try/catch` с обработкой любых исключений `InvalidOperationException`. Это особенно важно при вызове метода `Kill()`, поскольку такое исключение будет генерироваться, если процесс был завершен до вызова `Kill()`.

Управление запуском процесса с использованием класса `ProcessStartInfo`

Метод `Start()` позволяет также передавать тип `System.Diagnostics.ProcessStartInfo` для указания дополнительной информации относительно запуска определенного процесса. Ниже приведено частичное определение `ProcessStartInfo` (полное определение можно найти в документации .NET Framework 4.5 SDK):

```

public sealed class ProcessStartInfo : object
{
    public ProcessStartInfo();
    public ProcessStartInfo(string fileName);
    public ProcessStartInfo(string fileName, string arguments);
    public string Arguments { get; set; }
    public bool CreateNoWindow { get; set; }
    public StringDictionary EnvironmentVariables { get; }
    public bool ErrorDialog { get; set; }
    public IntPtr ErrorDialogParentHandle { get; set; }
    public string FileName { get; set; }
    public bool LoadUserProfile { get; set; }
    public SecureString Password { get; set; }
    public bool RedirectStandardError { get; set; }
    public bool RedirectStandardInput { get; set; }
    public bool RedirectStandardOutput { get; set; }
    public Encoding StandardErrorEncoding { get; set; }
}

```

```
public Encoding StandardOutputEncoding { get; set; }
public bool UseShellExecute { get; set; }
public string Verb { get; set; }
public string[] Verbs { get; }
public ProcessWindowStyle WindowStyle { get; set; }
public string WorkingDirectory { get; set; }
}
```

Для иллюстрации настройки запуска процесса изменим метод `StartAndKillProcess()` так, чтобы в нем загружался браузер Microsoft Internet Explorer, производился переход на сайт `www.facebook.com`, а окно браузера отображалось в развернутом виде:

```
static void StartAndKillProcess()
{
    Process ieProc = null;
    // Запустить Internet Explorer и перейти на сайт facebook.com
    // с развернутым на весь экран окном.
    try
    {
        ProcessStartInfo startInfo = new
            ProcessStartInfo("IExplore.exe", "www.facebook.com");
        startInfo.WindowStyle = ProcessWindowStyle.Maximized;
        ieProc = Process.Start(startInfo);
    }
    catch (InvalidOperationException ex)
    {
        Console.WriteLine(ex.Message);
    }
    ...
}
```

Теперь, когда известна роль процессов Windows и способы взаимодействия с ними из кода C#, можно переходить к изучению концепции доменов приложений .NET.

Исходный код. Проект `ProcessManipulator` доступен в подкаталоге `Chapter 17`.

Домены приложений .NET

На платформе .NET исполняемые файлы не размещаются прямо внутри процесса Windows, как это происходит в случае традиционных неуправляемых приложений. Вместо этого они попадают в отдельный логический раздел внутри процесса, который называется *доменом приложения*. Как будет показано, один процесс может содержать несколько доменов приложений, каждый из которых обслуживает свой исполняемый файл .NET. Такое дополнительное разделение традиционного процесса Windows представляет ряд преимуществ, главные из которых описаны ниже.

- Домены приложений являются ключевым аспектом нейтральной по отношению к операционным системам природы платформы .NET, поскольку такое логическое разделение абстрагирует отличия в том, как лежащая в основе операционная система представляет загруженный исполняемый файл.
- Домены приложений гораздо менее дорогостоящие в плане потребления вычислительных ресурсов и памяти по сравнению с полноценными процессами. Благодаря этому среда CLR способна загружать и выгружать домены приложений намного быстрее, чем формальные процессы, тем самым значительно улучшая масштабируемость серверных приложений.

- Домены приложений обеспечивают более глубокий уровень изоляции при размещении загруженных приложений. В случае сбоя одного домена приложения внутри процесса остальные домены приложений остаются работоспособными.

Как упоминалось выше, в одном процессе может обслуживаться любое количество доменов приложений, каждый из которых полностью изолирован от остальных доменов внутри того же самого (или любого другого) процесса. Учитывая этот факт, очень важно понимать, что приложение, выполняющееся в одном домене приложения, не может получать данные любого рода (глобальные переменные или статические поля) из другого домена приложения, если только не будет использоваться протокол распределенного программирования (такой как Windows Communication Foundation).

Хотя в одном процессе *может* находиться множество доменов приложений, обычно такого не происходит. Как минимум, процесс операционной системы будет обслуживать так называемый *стандартный домен приложения*. Этот специфичный домен приложения создается автоматически средой CLR во время запуска процесса. После этого среда CLR создает дополнительные домены приложений по мере необходимости.

Класс System.AppDomain

Платформа .NET позволяет программно осуществлять мониторинг доменов приложений, создавать новые домены приложений (или выгружать их) во время выполнения, загружать сборки в домены приложений и решать целый ряд других задач с применением класса AppDomain из пространства имен System, которое находится в сборке mscorlib.dll. В табл. 17.5 перечислены наиболее полезные методы этого класса (полную информацию об этом классе и его членах можно найти в документации .NET Framework 4.5 SDK).

Таблица 17.5. Избранные методы класса AppDomain

Метод	Описание
CreateDomain()	Этот статический метод позволяет создавать новый домен приложения в текущем процессе
CreateInstance()	Этот метод позволяет создавать экземпляр типа из внешней сборки после загрузки данной сборки в вызывающий домен приложения
ExecuteAssembly()	Этот метод запускает сборку *.exe внутри домена приложения, получив ее имя файла
GetAssemblies()	Этот метод получает набор сборок .NET, которые были загружены в данный домен приложения (двоичные сборки на основе COM и C игнорируются)
GetCurrentThreadId()	Этот статический метод возвращает идентификатор активного потока в текущем домене приложения
Load()	Этот метод применяется для динамической загрузки сборки в текущий домен приложения
Unload()	Этот статический метод позволяет выгрузить указанный домен приложения из заданного процесса

На заметку! Платформа .NET не позволяет выгружать конкретную сборку из памяти. Единственным способом для программной выгрузки библиотек является уничтожение размещающего домена приложения с помощью метода Unload().

Кроме того, класс `AppDomain` определяет набор свойств, которые могут быть удобны при мониторинге действий указанного домена приложения. Наиболее интересные свойства кратко описаны в табл. 17.6.

Таблица 17.6. Избранные свойства класса `AppDomain`

Событие	Описание
<code>BaseDirectory</code>	Это свойство позволяет получить путь к каталогу, который распознаватель сборок использует для зондирования сборок
<code>CurrentDomain</code>	Это статическое свойство позволяет получить домен приложения, используемый для текущего выполняющегося потока
<code>FriendlyName</code>	Это свойство позволяет получить дружественное имя текущего домена приложения
<code>MonitoringIsEnabled</code>	Это свойство позволяет получить или установить значение, которое указывает, включен ли мониторинг ресурсов центрального процессора и памяти для текущего процесса. После того, как мониторинг включен для процесса, отключить его невозможно
<code>SetupInformation</code>	Это свойство позволяет извлечь детали конфигурации для указанного домена приложения, которые представлены в виде объекта <code>AppDomainSetup</code>

И, наконец, класс `AppDomain` поддерживает набор событий, которые соответствуют различным аспектам жизненного цикла домена приложения. Наиболее полезные события, к которым можно привязаться, перечислены в табл. 17.7.

Таблица 17.7. Избранные события класса `AppDomain`

Событие	Описание
<code>AssemblyLoad</code>	Это событие возникает, когда сборка загружается в память
<code>AssemblyResolve</code>	Это событие возникает, когда распознаватель сборок не может найти местоположение обязательной сборки
<code>DomainUnload</code>	Это событие возникает перед началом выгрузки домена приложения из размещающего процесса
<code>FirstChanceException</code>	Это событие позволяет получать уведомление о том, что в домене приложения было сгенерировано исключение, перед тем как среда CLR начнет поиск подходящего оператора <code>catch</code>
<code>ProcessExit</code>	Это событие возникает в стандартном домене приложения тогда, когда его родительский процесс завершается
<code>UnhandledException</code>	Это событие возникает, когда исключение не было перехвачено обработчиком исключений

Взаимодействие со стандартным доменом приложения

Вспомните, что при запуске исполняемого файла .NET среда CLR автоматически помещает его в стандартный домен приложения размещающего процесса. Это делается автоматически и прозрачно, и писать какой-то специальный код не понадобится. Тем не менее, к стандартному домену приложения можно получать доступ в своем прило-

жении с помощью статического свойства `AppDomain.CurrentDomain`. После этого появляется возможность привязки к любым интересующим событиям либо обращения к методам и свойствам `AppDomain` для проведения диагностики во время выполнения.

Чтобы научиться взаимодействовать со стандартным доменом приложения, начнем с создания нового консольного приложения по имени `DefaultAppDomainApp`. Модифицируем класс `Program`, включив в него приведенный ниже код, который просто выводит детальные сведения о стандартном домене приложения с использованием ряда членов класса `AppDomain`.

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with the default AppDomain *****\n");
        DisplayDADStats();
        Console.ReadLine();
    }
    private static void DisplayDADStats()
    {
        // Получить доступ к домену приложения для текущего потока.
        AppDomain defaultAD = AppDomain.CurrentDomain;
        // Вывести различные статистические данные об этом домене.
        Console.WriteLine("Name of this domain: {0}",
            defaultAD.FriendlyName);           // Дружественное имя
        Console.WriteLine("ID of domain in this process: {0}",
            defaultAD.Id);                     // Идентификатор
        Console.WriteLine("Is this the default domain?: {0}",
            defaultAD.IsDefaultAppDomain()); // Является ли стандартным
        Console.WriteLine("Base directory of this domain: {0}",
            defaultAD.BaseDirectory);          // Базовый каталог
    }
}
```

Ниже приведен вывод этого примера:

```
***** Fun with the default AppDomain *****
Name of this domain: DefaultAppDomainApp.exe
ID of domain in this process: 1
Is this the default domain?: True
Base directory of this domain: E:\MyCode\DefaultAppDomainApp\bin\Debug\
```

Обратите внимание, что имя стандартного домена приложения будет идентичным имени содержащегося внутри него исполняемого файла (`DefaultAppDomainApp.exe` в этом примере). Кроме того, значение базового каталога, которое будет использоваться для зондирования обязательных внешних закрытых сборок, отображается на текущее местоположение развернутой исполняемой сборки.

Перечисление загруженных сборок

Используя метод `GetAssemblies()` уровня экземпляра, можно просмотреть все сборки .NET, загруженные в указанный домен приложения. Этот метод будет возвращать массив объектов типа `Assembly`, который, как было показано в главе 15, является членом пространства имен `System.Reflection` (не забудьте импортировать это пространство имен в файл кода C#).

В целях иллюстрации определим в классе `Program` новый метод по имени `ListAllAssembliesInAppDomain()`. Этот вспомогательный метод будет получать список всех загруженных сборок и выводить для каждой из них дружественное имя и номер версии.

```
static void ListAllAssembliesInAppDomain()
{
    // Получить доступ к домену приложения для текущего потока.
    AppDomain defaultAD = AppDomain.CurrentDomain;
    // Извлечь все сборки, загруженные в стандартный домен приложения.
    Assembly[] loadedAssemblies = defaultAD.GetAssemblies();
    Console.WriteLine("***** Here are the assemblies loaded in {0} *****\n",
        defaultAD.FriendlyName);
    foreach (Assembly a in loadedAssemblies)
    {
        Console.WriteLine("-> Name: {0}", a.GetName().Name);           // Имя
        Console.WriteLine("-> Version: {0}\n", a.GetName().Version);    // Версия
    }
}
```

Добавив в метод Main() вызов этого нового метода, можно просмотреть все библиотеки .NET, которые используются в домене приложения, размещающем исполняемую сборку:

```
***** Here are the assemblies loaded in DefaultAppDomainApp.exe *****
-> Name: mscorlib
-> Version: 4.0.0.0

-> Name: DefaultAppDomainApp
-> Version: 1.0.0.0
```

Важно понимать, что список загруженных сборок может изменяться в любой момент при написании нового кода C#.

Например, предположим, что метод ListAllAssembliesInAppDomain() модифицирован так, чтобы в нем использовался LINQ-запрос, упорядочивающий загруженные сборки по имени:

```
static void ListAllAssembliesInAppDomain()
{
    // Получить доступ к домену приложения для текущего потока.
    AppDomain defaultAD = AppDomain.CurrentDomain;
    // Извлечь все сборки, загруженные в стандартный домен приложения.
    var loadedAssemblies = from a in defaultAD.GetAssemblies()
        orderby a.GetName().Name select a;
    Console.WriteLine("***** Here are the assemblies loaded in {0} *****\n",
        defaultAD.FriendlyName);
    foreach (var a in loadedAssemblies)
    {
        Console.WriteLine("-> Name: {0}", a.GetName().Name);           // Имя
        Console.WriteLine("-> Version: {0}\n", a.GetName().Version);    // Версия
    }
}
```

Запустив приложение еще раз, можно увидеть, что в память также были загружены сборки System.Core.dll и System.dll, поскольку они требуются для API-интерфейса LINQ to Objects:

```
***** Here are the assemblies loaded in DefaultAppDomainApp.exe *****
-> Name: DefaultAppDomainApp
-> Version: 1.0.0.0

-> Name: mscorlib
-> Version: 4.0.0.0

-> Name: System
-> Version: 4.0.0.0

-> Name: System.Core
-> Version: 4.0.0.0
```

Получение уведомлений о загрузке сборок

Для получения уведомлений от среды CLR при загрузке новой сборки в определенный домен приложения необходимо обработать событие `AssemblyLoad`. Это событие имеет тип делегата `AssemblyLoadEventHandler`, который может указывать на любой метод, принимающий `System.Object` в первом параметре и `AssemblyLoadEventArgs` — во втором.

Давайте добавим в текущий класс `Program` последний метод по имени `InitDAD()`, который будет инициализировать стандартный домен приложения, путем обработки события `AssemblyLoad` через подходящее лямбда-выражение.

```
private static void InitDAD()
{
    // Эта логика будет выводить имя любой сборки,
    // загруженной в домен приложения после его создания.
    AppDomain defaultAD = AppDomain.CurrentDomain;
    defaultAD.AssemblyLoad += (o, s) =>
    {
        Console.WriteLine("{0} has been loaded!", s.LoadedAssembly.GetName().Name);
    };
}
```

Как и можно было ожидать, запустив модифицированное приложение, при загрузке новой сборки будет отображаться соответствующее уведомление. В этом примере просто выводится дружественное имя сборки за счет использования свойства `LoadedAssembly` входного параметра `AssemblyLoadedEventArgs`.

Исходный код. Проект `DefaultAppDomainApp` доступен в подкаталоге `Chapter 17`.

Создание новых доменов приложений

Вспомните, что единственный процесс способен размещать множество доменов приложений, создаваемых с помощью статического метода `AppDomain.CreateDomain()`. Несмотря на то что потребность в создании новых доменов приложений на лету в большинстве приложений .NET возникает довольно редко, важно понимать основы того, как это делается. Например, в главе 18 будет показано, что создаваемые динамические сборки должны устанавливаться в специальный домен приложения. Вдобавок многие API-интерфейсы, связанные с безопасностью .NET, требуют понимания того, как конструировать новые домены приложений для изоляции сборок на основе предоставляемых учетных данных безопасности.

Чтобы посмотреть, как конструировать новые домены приложений на лету (и загружать в них новые сборки), создадим новое консольное приложение по имени `CustomAppDomains`. Метод `AppDomain.CreateDomain()` имеет несколько перегруженных версий. Как минимум, понадобится указать дружественное имя создаваемого домена приложения. Ниже приведен код, который должен быть помещен в класс `Program`. Здесь используется метод `ListAllAssembliesInAppDomain()` из предыдущего примера, но на этот раз ему в качестве входного аргумента передается объект `AppDomain`, предназначенный для анализа.

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with Custom AppDomains *****\n");
```

```
// Вывести все сборки, загруженные в стандартный домен приложения.
AppDomain defaultAD = AppDomain.CurrentDomain;
ListAllAssembliesInAppDomain(defaultAD);

// Создать новый домен приложения.
MakeNewAppDomain();
Console.ReadLine();
}

private static void MakeNewAppDomain()
{
    // Создать новый домен приложения в текущем процессе
    // и вывести список загруженных сборок.
    AppDomain newAD = AppDomain.CreateDomain("SecondAppDomain");
    ListAllAssembliesInAppDomain(newAD);
}

static void ListAllAssembliesInAppDomain(AppDomain ad)
{
    // Получить все сборки, загруженные в стандартный домен приложения.
    var loadedAssemblies = from a in ad.GetAssemblies()
                           orderby a.GetName().Name select a;

    Console.WriteLine("***** Here are the assemblies loaded in {0} *****\n",
        ad.FriendlyName);
    foreach (var a in loadedAssemblies)
    {
        Console.WriteLine("-> Name: {0}", a.GetName().Name);           // Имя
        Console.WriteLine("-> Version: {0}\n", a.GetName().Version); // Версия
    }
}
```

Запустив приложение, вы увидите, что в стандартный домен приложения (CustomAppDomains.exe) были загружены сборки mscorlib.dll, System.dll, System.Core.dll и CustomAppDomains.exe, учитывая кодовую базу C# текущего проекта. Однако новый домен приложения содержит только сборку mscorlib.dll, которая, как вы помните, является одной из сборок .NET, всегда загружаемых средой CLR в каждый домен приложения.

```
***** Fun with Custom AppDomains *****
***** Here are the assemblies loaded in CustomAppDomains.exe *****
-> Name: CustomAppDomains
-> Version: 1.0.0.0
-> Name: mscorlib
-> Version: 4.0.0.0
-> Name: System
-> Version: 4.0.0.0
-> Name: System.Core
-> Version: 4.0.0.0
***** Here are the assemblies loaded in SecondAppDomain *****
-> Name: mscorlib
-> Version: 4.0.0.0
```

На заметку! Запустив отладку этого проекта (нажатием <F5>), вы обнаружите, что в каждый домен приложения загружается множество дополнительных сборок, которые используются процессом отладки Visual Studio. Если просто запустить проект на выполнение (нажатием <Ctrl+F5>), будут отображаться только сборки, напрямую загруженные в каждый домен приложения.

Если вы имеете опыт работы с традиционными приложениями Windows, подобное поведение может показаться нелогичным (т.к. предполагается, что оба домена приложений имеют доступ к одному и тому же набору сборок). Тем не менее, вспомните, что сборка загружается в домен приложения, а не прямо в сам процесс.

Загрузка сборок в специальные домены приложений

Среда CLR будет всегда загружать сборки в стандартный домен приложения по мере необходимости. Однако в случае создания вручную новых доменов приложений сборки можно загружать в указанный домен с помощью метода `AppDomain.Load()`. Кроме того, не следует забывать о возможности вызова метода `AppDomain.ExecuteAssembly()`, который позволяет загрузить сборку *.exe и выполнить метод `Main()`.

Предположим, что необходимо загрузить сборку `CarLibrary.dll` в новый вторичный домен приложения. При условии, что эта библиотека скопирована в папку `bin\Debug` текущего приложения, можно изменить метод `MakeNewAppDomain()`, как показано ниже (не забыв импортировать пространство имен `System.IO` для получения доступа к классу `FileNotFoundException`):

```
private static void MakeNewAppDomain()
{
    // Создать новый домен приложения в текущем процессе.
    AppDomain newAD = AppDomain.CreateDomain("SecondAppDomain");

    try
    {
        // Загрузить CarLibrary.dll в этот новый домен.
        newAD.Load("CarLibrary");
    }
    catch (FileNotFoundException ex)
    {
        Console.WriteLine(ex.Message);
    }

    // Вывести список всех сборок.
    ListAllAssembliesInAppDomain(newAD);
}
```

На этот раз вывод приложения выглядит следующим образом (обратите внимание на присутствие сборки `CarLibrary.dll`):

```
***** Fun with Custom AppDomains *****

***** Here are the assemblies loaded in CustomAppDomains.exe *****

-> Name: CustomAppDomains
-> Version: 1.0.0.0

-> Name: mscorlib
-> Version: 4.0.0.0

-> Name: System
-> Version: 4.0.0.0

-> Name: System.Core
-> Version: 4.0.0.0

***** Here are the assemblies loaded in SecondAppDomain *****

-> Name: CarLibrary
-> Version: 2.0.0.0

-> Name: mscorlib
-> Version: 4.0.0.0
```

На заметку! Не забывайте, что во время отладки приложения в каждый домен приложения будут загружаться многие дополнительные библиотеки.

Выгрузка доменов приложений программным образом

Важно обратить внимание, что среда CLR не разрешает выгружать индивидуальные сборки .NET. Тем не менее, с помощью метода `AppDomain.Unload()` можно избирательно выгружать домены приложений из размещающего процесса. В этом случае вместе с доменом приложения будут выгружаться и все содержащиеся в нем сборки.

Вспомните, что в типе `AppDomain` определено событие `DomainUnload`, которое инициируется при выгрузке специального домена приложения из содержащего его процесса. Еще одним интересным событием является `ProcessExit`, которое генерируется при выгрузке из процесса стандартного домена приложения (что очевидно влечет за собой завершение самого процесса).

Чтобы реализовать программную выгрузку домена `newAD` из размещающего процесса с получением уведомления об его уничтожении, модифицируем метод `MakeNewAppDomain()`, добавив в него следующую логику:

```
private static void MakeNewAppDomain()
{
    // Создать новый домен приложения в текущем процессе.
    AppDomain newAD = AppDomain.CreateDomain("SecondAppDomain");
    newAD.DomainUnload += (o, s) =>
    {
        Console.WriteLine("The second AppDomain has been unloaded!");
    };
    try
    {
        // Загрузить CarLibrary.dll в этот новый домен.
        newAD.Load("CarLibrary");
    }
    catch (FileNotFoundException ex)
    {
        Console.WriteLine(ex.Message);
    }
    // Вывести список всех сборок.
    ListAllAssembliesInAppDomain(newAD);
    // Уничтожить этот домен приложения.
    AppDomain.Unload(newAD);
}
```

Чтобы обеспечить уведомление при выгрузке стандартного домена приложения, в методе `Main()` понадобится обработать событие `ProcessEvent` этого домена, как показано ниже:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Custom AppDomains *****\n");
    // Вывести все сборки, загруженные в стандартный домен приложения.
    AppDomain defaultAD = AppDomain.CurrentDomain;
    defaultAD.ProcessExit += (o, s) =>
    {
        Console.WriteLine("Default AD unloaded!");
    };
    ListAllAssembliesInAppDomain(defaultAD);
    MakeNewAppDomain();
    Console.ReadLine();
}
```

На этом рассмотрение доменов приложений .NET завершено. Напоследок мы рассмотрим в этой главе еще один уровень разделения, применяемый для группирования объектов в контекстные границы.

Исходный код. Проект CustomAppDomains доступен в подкаталоге Chapter 17.

Контекстные границы объектов

Как было только что показано, домены приложений — это логические разделы внутри процесса, используемого для размещения сборок .NET. Однако каждый домен приложения может быть дополнительно разделен на многочисленные контекстные границы. В сущности, контекст .NET предоставляет одиночному домену приложения возможность установки “специфического местоположения” для заданного объекта.

На заметку! В то время как понимание концепции процессов и доменов приложений является довольно важным, необходимость работы с объектными контекстами в большинстве приложений .NET никогда не возникает. Материал по объектным контекстам включен в настоящую главу лишь для предоставления более полной картины.

Используя контекст, среда CLR способна обеспечить надлежащую и согласованную обработку объектов, которые имеют специальные требования времени выполнения, за счет перехвата вызовов методов, производимых внутри и за пределами заданного контекста. Этот уровень перехвата позволяет среде CLR настраивать текущий вызов метода так, чтобы он соответствовал контекстным настройкам конкретного объекта. Например, если определен класс C#, который требует автоматической безопасности в отношении потоков (с применением атрибута [Synchronization]), среда CLR будет создавать во время его размещения “синхронизированный контекст”.

Точно так же, как процесс определяет стандартный домен приложения, каждый домен приложения имеет стандартный контекст. Этот стандартный контекст (иногда называемый *контекстом 0*, т.к. он всегда создается в домене приложения первым) применяется для группирования вместе объектов .NET, которые не имеют никаких специфических или уникальных контекстных потребностей. Как и можно было ожидать, подавляющее большинство объектов .NET загружается именно в контекст 0. Если среда CLR определяет, что вновь созданный объект предъявляет специальные требования, она создает внутри размещающего домена приложения новую контекстную границу. На рис. 17.3 показаны отношения между процессом, доменом приложения и контекстом.

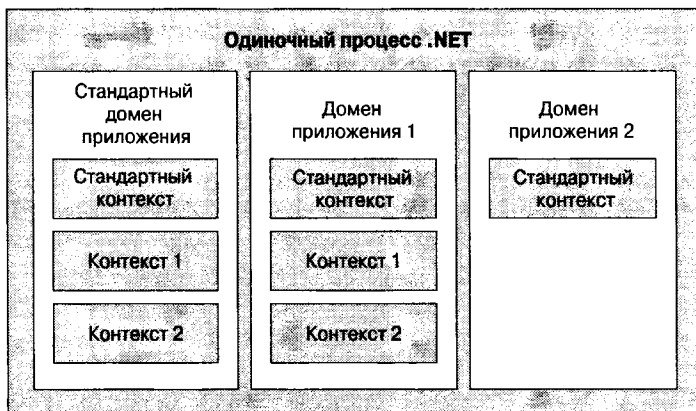


Рис. 17.3. Процессы, домены приложений и контекстные границы

Контекстно-свободные и контекстно-связанные типы

Объекты .NET, которые не требуют какого-то особого контекстного сопровождения, называются *контекстно-свободными*. Доступ к таким объектам может быть получен из любого места внутри размещающего домена приложения без взаимодействия с их требованиями времени выполнения. Построение контекстно-свободных объектов осуществляется очень просто, поскольку при этом вообще ничего не нужно делать (в частности, их не требуется снабжать любыми контекстными атрибутами, равно как и порождать от базового класса `System.ContextBoundObject`), например:

```
// Контекстно-свободный объект загружается в контекст 0.
class SportsCar{}
```

С другой стороны, объекты, которые действительно требуют выделения отдельного контекста, называются *контекстно-связанными* и должны быть порождены от базового класса `System.ContextBoundObject`. Этот базовый класс отражает тот факт, что интересующий объект может функционировать надлежащим образом только в рамках контекста, в котором он был создан. Учитывая роль контекста .NET, должно быть ясно, что в случае попадания контекстно-связанного объекта по какой-то причине в несовместимый контекст, обязательно произойдет что-то неприемлемое, причем в самый неподходящий момент. Помимо порождения от `System.ContextBoundObject`, контекстно-связанный тип будет также оснащен специальной категорией атрибутов .NET, которая называется *контекстными атрибутами*. Все контекстные атрибуты являются производными от базового класса `ContextAttribute`. Давайте рассмотрим пример.

Определение контекстно-связанного объекта

Предположим, что необходимо определить класс (по имени `SportsCarTS`), который автоматически обеспечивает безопасность в отношении потоков, даже если внутри реализации его членов отсутствует жестко закодированная логика синхронизации потоков. Чтобы создать такой класс, унаследуйте его от `ContextBoundObject` и примените к нему атрибут `[Synchronization]`, как показано ниже:

```
using System.Runtime.Remoting.Contexts;
// Этот контекстно-связанный тип будет загружаться только в синхронизированный
// (и, следовательно, безопасный к потокам) контекст.
[Synchronization]
class SportsCarTS : ContextBoundObject
{}
```

Типы, снабженные атрибутом `[Synchronization]`, загружаются в безопасный к потокам контекст. Учитывая специальные контекстные потребности класса `MyThreadSafeObject`, только представьте, какие проблемы возникли бы в случае перемещения созданного объекта из синхронизированного контекста в несинхронизированный. Объект сразу же перестает быть безопасным к потокам и, следовательно, становится кандидатом на массовое повреждение данных, т.к. доступ к нему попытаются получить одновременно многие потоки. Для обеспечения, что среда CLR никогда не переместит объекты `SportsCarTS` за пределы синхронизированного контекста, просто унаследуйте `SportsCarTS` от `ContextBoundObject`.

Исследование контекста объекта

Хотя необходимость в программном взаимодействии с контекстом возникает в приложениях очень редко, давайте все-таки рассмотрим один показательный пример. Создадим новое консольное приложение по имени `ObjectContextApp` и определим в нем контекстно-свободный класс `SportsCar` и контекстно-связанный класс `SportsCarTS`:


```

using System;
using System.Runtime.Remoting.Contexts; // Для типа Context.
using System.Threading;                 // Для типа Thread.
// SportsCar не имеет никаких специальных контекстных
// потребностей и будет загружаться в стандартный
// контекст домена приложения.
class SportsCar
{
    public SportsCar()
    {
        // Получить информацию о контексте и вывести идентификатор контекста.
        Context ctx = Thread.CurrentContext;
        Console.WriteLine("{0} object in context {1}",
            this.ToString(), ctx.ContextID);
        foreach(IContextProperty itfCtxProp in ctx.ContextProperties)
            Console.WriteLine("-> Ctx Prop: {0}", itfCtxProp.Name);
    }
}
// SportsCarTS требует загрузки в синхронизированный контекст.
[Synchronization]
class SportsCarTS : ContextBoundObject
{
    public SportsCarTS()
    {
        // Получить информацию о контексте и вывести идентификатор контекста.
        Context ctx = Thread.CurrentContext;
        Console.WriteLine("{0} object in context {1}",
            this.ToString(), ctx.ContextID);
        foreach(IContextProperty itfCtxProp in ctx.ContextProperties)
            Console.WriteLine("-> Ctx Prop: {0}", itfCtxProp.Name);
    }
}

```

Обратите внимание, что каждый конструктор получает объект Context из текущего потока выполнения посредством статического свойства Thread.CurrentContext. Имея объект Context, можно вывести статистические данные о контекстной границе, такие как назначенный идентификатор, а также набор дескрипторов, полученных через свойство Context.ContextProperties. Это свойство возвращает массив объектов, реализующих интерфейс IContextProperty, который открывает доступ к каждому дескриптору через свойство Name. Добавим в метод Main() код для размещения экземпляра каждого из этих классов в памяти, как показано ниже:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Object Context *****\n");
    // Объекты при создании будут отображать контекстную информацию.
    SportsCar sport = new SportsCar();
    Console.WriteLine();

    SportsCar sport2 = new SportsCar();
    Console.WriteLine();
    SportsCarTS synchroSport = new SportsCarTS();
    Console.ReadLine();
}

```

По мере создания объектов конструкторы классов будут выводить различные фрагменты контекстной информации (выводимое свойство LeaseLifeTimeServiceProperty представляет собой низкоуровневый аспект уровня удаленной обработки .NET, поэтому его можно проигнорировать):

```
***** Fun with Object Context *****
```

```
ObjectContextApp.SportsCar object in context 0
```

```
-> Ctx Prop: LeaseLifeTimeServiceProperty
```

```
ObjectContextApp.SportsCar object in context 0
```

```
-> Ctx Prop: LeaseLifeTimeServiceProperty
```

```
ObjectContextApp.SportsCarTS object in context 1
```

```
-> Ctx Prop: LeaseLifeTimeServiceProperty
```

```
-> Ctx Prop: Synchronization
```

Поскольку класс `SportsCar` не был снабжен атрибутом контекста, среда CLR разместила объекты `sport` и `sport2` в контексте 0 (т.е. в стандартном контексте). Однако объект `SportsCarTS` загрузился в уникальную контекстную границу (которой был назначен идентификатор контекста, равный 1), учитывая тот факт, что к этому контекстно-связанному классу был применен атрибут `[Synchronization]`.

Исходный код. Проект `ObjectContextApp` доступен в подкаталоге `Chapter 17`.

Итоговые сведения о процессах, доменах приложений и контекстах

К этому моменту вы должны намного лучше понимать идею, положенную в основу размещения сборок .NET средой CLR. Ниже для удобства перечислены основные моменты.

- Процесс .NET размещает один и более доменов приложений. Каждый домен приложения способен размещать любое количество связанных сборок .NET. Домены приложений могут независимо загружаться и выгружаться средой CLR (или программно с помощью класса `System.AppDomain`).
- Любой заданный домен приложения может состоять из одного и более контекстов. Используя контекст, среда CLR имеет возможность поместить объект с "особыми потребностями" в логический контейнер, чтобы гарантировать учет этих требований времени выполнения.

Если изложенный выше материал показался слишком низкоуровневым, не переживайте. По большей части среда CLR самостоятельно разбирается со всеми деталями процессов, доменов приложений и контекстов. Тем не менее, приведенные сведения являются хорошей основой для понимания многопоточного программирования на платформе .NET.

Резюме

Главная задача настоящей главы заключалась в том, чтобы продемонстрировать, каким образом платформа .NET размещает образы исполняемых сборок .NET. Давно существующее понятие процесса Windows было внутренне изменено и адаптировано под потребности CLR. Любой отдельно взятый процесс (которым на программном уровне можно управлять с помощью типа `System.Diagnostics.Process`) теперь включает в себя один или более доменов приложений, представляющих изолированные и независимые границы внутри процесса.

Как было показано, одиночный процесс может иметь несколько доменов приложений, каждый из которых способен размещать и выполнять любое количество связанных сборок. Кроме того, один домен приложения может содержать любое количество контекстных границ. Используя такой дополнительный уровень изоляции типов, среда CLR обеспечивает надлежащую обработку объектов с особыми потребностями во время выполнения.

Язык CIL и роль динамических сборок

При построении полномасштабного приложения .NET, скорее всего, будет применяться язык C# (или другой управляемый язык, такой как Visual Basic), из-за присущей ему продуктивности и удобства в использовании. Однако, как было показано в самой первой главе, роль управляемого компилятора состоит в преобразовании файлов кода *.cs в инструкции на языке CIL, метаданные типов и манифест сборки. CIL является полноценным языком программирования .NET, имеющим собственный синтаксис, семантику и компилятор (ilasm.exe).

В этой главе предлагается краткий экскурс по этому внутреннему языку .NET. Здесь будет показано, в чем состоят различия между *директивой*, *атрибутом* и *кодом операции* в CIL, а также описана роль возвратного проектирования сборки .NET и различных инструментов программирования на CIL. Также будут представлены основы определения пространств имен, типов и членов с использованием грамматики CIL. В завершение главы рассматривается роль пространства имен System.Reflection.Emit и реализация (с помощью инструкций CIL) динамического создания сборки во время выполнения.

Естественно, необходимость иметь дело непосредственно с кодом на CIL в повседневной работе будет возникать только у очень немногих программистов. Глава начинается с описания причин, по которым изучение синтаксиса и семантики этого низкоуровневого языка .NET может оказаться полезным.

Причины для изучения грамматики языка CIL

Язык CIL является самым настоящим собственным языком платформы .NET. При создании сборки .NET с помощью выбранного управляемого языка (C#, VB, F#, COBOL, NET и т.д.) соответствующий компилятор всегда транслирует исходный код в инструкции CIL. Подобно любому языку программирования, CIL предоставляет множество лексем, связанных со структурированием и реализацией. Поскольку CIL представляет собой просто еще один язык программирования .NET, не должен удивлять тот факт, что сборки .NET можно создавать непосредственно на CIL и компилировать их с помощью CIL-компилятора ilasm.exe, который входит в состав .NET Framework 4.5 SDK.

Хотя строить все приложение .NET непосредственно на CIL решаются очень немногие программисты, этот язык по-прежнему является чрезвычайно интересным объектом для изучения. Хорошие знания грамматики CIL позволяют совершенствовать приемы разработки .NET-приложений.

Разработчики, разбирающиеся в CIL, обладают следующими навыками.

- Точно знают, на какие лексемы CIL отображаются ключевые слова из различных языков программирования .NET.
- Умеют дизассемблировать существующие сборки .NET, редактировать лежащий в их основе код CIL и заново компилировать обновленную кодовую базу в модифицированный двоичный файл .NET. Например, некоторые сценарии могут требовать внесения изменений в код CIL для взаимодействия с расширенными средствами COM.
- Умеют строить динамические сборки с использованием пространства имен `System.Reflection.Emit`. Этот API-интерфейс позволяет генерировать в памяти сборку .NET, которая впоследствии может быть сохранена на диске.
- Умеют использовать такие возможности CTS, которые не поддерживаются управляемыми языками более высокого уровня, но существуют на уровне CIL. На самом деле CIL является единственным языком .NET, который позволяет получать доступ к абсолютно всем возможностям CTS. Например, с применением низкоуровневого кода CIL можно определять члены и поля глобального уровня (то, что в C# делать не разрешается).

Следует еще раз подчеркнуть, что овладеть всеми навыками работы с языком C# и библиотеками базовых классов .NET можно и без изучения деталей кода CIL. Во многих отношениях знание языка CIL аналогично знанию языка ассемблера для программиста, работающего на C (C++). Те, кто разбирается в низкоуровневых деталях, способны создавать более совершенные решения существующих задач и лучше понимают, как работает базовая среда программирования (и выполнения). Поэтому, если вам интересно, давайте приступим к изучению основных аспектов CIL.

На заметку! Следует отметить, что эта глава не задумывалась как исчерпывающее описание синтаксиса и семантики CIL. Полная информация по данной теме приведена в официальной спецификации ECMA (документ `ecma-335.pdf`), доступная для загрузки на веб-сайте ECMA International по адресу www.ecma-international.org.

Директивы, атрибуты и коды операций CIL

В начале изучения любого низкоуровневого языка вроде CIL обязательно встречаются новые (и часто пугающие) названия для хорошо знакомых понятий. Например, к этому моменту в книге приведенный ниже набор элементов почти наверняка будет воспринят как ключевые слова языка C# (и это правильно):

```
{new, public, this, base, get, set, explicit, unsafe, enum, operator, partial}
```

Однако внимательней присмотревшись к элементам в этом наборе, можно заметить, что хотя каждый из них действительно является ключевым словом C#, он обладает радикально отличающейся семантикой. Например, ключевое слово `enum` определяет производный от `System.Enum` тип, а ключевые слова `this` и `base` позволяют ссылаться соответственно, на текущий объект и его родительский класс. Ключевое слово `unsafe` используется для создания блока кода, который не может напрямую отслеживаться средой CLR, а ключевое слово `operator` предназначено для создания скрытого (со специальным именем) метода, который будет вызываться в случае применения специфической операции C# (например, знака "плюс").

В отличие от высокоуровневого языка, такого как C#, в CIL отсутствует определение общего набора ключевых слов само по себе. Вместо этого набор лексем, распознавае-

мых компилятором CIL, разделен на три следующих обширных категории на основе их семантики:

- директивы CIL;
- атрибуты CIL;
- коды операций CIL.

Лексемы CIL каждой из этих категорий выражаются с помощью определенного синтаксиса и комбинируются для построения допустимой сборки .NET.

Роль директив CIL

Прежде всего, в CIL имеется набор хорошо известных лексем, которые применяются для описания общей структуры сборки .NET. Эти лексемы называются *директивами*. Директивы CIL позволяют информировать компилятор CIL о том, как должны определяться пространства имен, типы и члены, входящие в состав сборки.

Синтаксически директивы представляются с использованием префикса в виде точки (.), например, `.namespace`, `.class`, `.publickeytoken`, `.override`, `.method`, `.assembly` и т.д. Следовательно, если в файле с расширением `*.il` (принятое по соглашению расширение для файлов CIL-кода) указана одна директива `.namespace` и три директивы `.class`, то компилятор CIL сгенерирует сборку, в которой определено единственное пространство имен, содержащее три класса .NET.

Роль атрибутов CIL

Во многих случаях директивы CIL сами по себе оказываются недостаточно описательными для того, чтобы полностью выразить определение заданного типа или члена типа .NET. С учетом этого многие директивы CIL могут сопровождаться разнообразными *атрибутами* CIL, которые уточняют способ обработки той или иной директивы. Например, директива `.class` может быть снабжена атрибутом `public` (задающим видимость типа), атрибутом `extends` (явно указывающим базовый класс типа) и атрибутом `implements` (позволяющим перечислить интерфейсы, поддерживаемые данным типом).

На заметку! Не путайте следующие два совершенно разных понятия: "атрибут .NET" (глава 15) и "атрибут CIL".

Роль кодов операций CIL

После того как сборка .NET, пространство имен и набор типов определены в терминах языка CIL с использованием различных директив и связанных атрибутов, останется только предоставить для каждого типа логику реализации. Это работа *кодов операций*. Подобно другим низкоуровневым языкам программирования, коды операций CIL, как правило, имеют непонятный и нечитабельный вид. Например, для загрузки в память переменной `string` в CIL применяется код операции, который имеет не дружественное имя наподобие `LoadString`, а записывается как `ldstr`.

Некоторые коды операций CIL вполне естественно отображаются на свои аналоги в C# (например, `box`, `unbox`, `throw` и `sizeof`). Как будет показано далее в главе, коды операций CIL всегда используются только в контексте реализации членов и, в отличие от директив, никогда не сопровождаются префиксом в виде точки.

Разница между кодами операций и их мнемоническими эквивалентами в CIL

Как только что объяснялось, для реализации членов конкретного типа применяются коды операций вроде `ldstr`. Однако в действительности лексемы, подобные `ldstr`, являются *мнемоническими эквивалентами CIL* действительных двоичных кодов операций CIL. Чтобы прояснить разницу, предположим, что написан следующий метод на C#:

```
static int Add(int x, int y)
{
    return x + y;
}
```

Действие сложения двух чисел в CIL выражается кодом операции `0X58`. В том же стиле, вычитание двух чисел выражается с помощью кода операции `0X59`, а действие по размещению нового объекта в управляемой куче записывается с использованием кода операции `0X73`. Таким образом, “код CIL”, обрабатываемый JIT-компилятором, в действительности является всего лишь порциями двоичных данных.

К счастью, для каждого двоичного кода операции CIL существует соответствующий мнемонический эквивалент. Например, вместо кода `0X58` может использоваться мнемонический эквивалент `add`, вместо `0X59` — эквивалент `sub`, а вместо `0X73` — `newobj`. Это значит, что декомпиляторы CIL, такие как `ildasm.exe`, транслируют двоичные коды операций сборки в соответствующие им мнемонические эквиваленты. Например, ниже показано, как `ildasm.exe` представит в CIL предыдущий метод `Add()`, написанный на языке C#:

```
.method private hidebysig static int32 Add(int32 x, int32 y) cil managed
{
    // Code size 9 (0x9)
    .maxstack 2
    .locals init ([0] int32 CS$1$0000)
    IL_0000: nop
    IL_0001: ldarg.0
    IL_0002: ldarg.1
    IL_0003: add
    IL_0004: stloc.0
    IL_0005: br.s IL_0007
    IL_0007: ldloc.0
    IL_0008: ret
}
```

Если только вы не занимаетесь разработкой исключительно низкоуровневого программного обеспечения .NET (например, специального управляемого компилятора), то необходимость иметь дело с числовыми двоичными кодами операций CIL никогда не возникнет. Обычно, если программисты .NET говорят о “кодах операций CIL”, то имеют в виду удобные для восприятия строковые мнемонические эквиваленты, а не лежащие в основе числовые значения.

Основанная на стеке природа CIL

В языках .NET высокого уровня (таких как C#) низкоуровневые детали CIL обычно максимально возможно скрываются из виду. Одним из особенно хорошо скрываемых аспектов является тот факт, что CIL является языком программирования, основанным на использовании стека. Вспомните из описания пространств имен коллекций (приведенного в главе 9), что класс `Stack<T>` может применяться для помещения значения в стек, а также для извлечения самого верхнего значения из стека. Разумеется, разработ-

чики на CIL не используют в буквальном смысле объект типа `Stack<T>` для загрузки и выгрузки вычисляемых значений, но применяют похожий стиль помещения и извлечения из стека.

Формально сущность, используемая для хранения набора вычисляемых значений, называется *виртуальным стеком выполнения*. Вы увидите, что CIL предоставляет несколько кодов операций, которые служат для помещения значения в стек; этот процесс называется *загрузкой*. Кроме того, в CIL определен набор дополнительных кодов операций, которые перемещают самое верхнее значение из стека в память (например, в локальную переменную), применяя процесс, называемый *сохранением*.

В мире CIL невозможно получать доступ к элементам данных, включая локально определенные переменные, входные аргументы методов и данные полей типа. Вместо этого элемент данных должен быть явно загружен в стек и затем извлекаться оттуда для последующего использования (помните о таком требовании, т.к. это поможет понять, почему блок кода CIL может выглядеть несколько избыточным).

На заметку! Вспомните, что CIL-код не выполняется напрямую, а компилируется по требованию. Во время компиляции CIL-кода многие избыточные аспекты реализации оптимизируются. Более того, если для текущего проекта включена опция оптимизации кода (на вкладке Build (Сборка) окна свойств проекта в Visual Studio), компилятор будет также удалять разнообразные избыточные детали CIL.

Чтобы понять, каким образом CIL использует модель обработки, основанную на стеке, создадим простой метод C# по имени `PrintMessage()`, не принимающий аргументов и возвращающий `void`. Внутри его реализации будет просто выводиться в стандартный выходной поток значение локальной переменной:

```
public void PrintMessage()
{
    string myMessage = "Hello.";
    Console.WriteLine(myMessage);
}
```

Если просмотреть CIL-код, который получился в результате трансляции этого метода компилятором C#, выяснится, что метод `PrintMessage()` определяет ячейку для хранения локальной переменной с помощью директивы `.locals`. Локальная строка затем загружается и сохраняется в этой локальной переменной с использованием кодов операций `ldstr` (загрузка строки) и `stloc.0` (сохранение текущего значения в локальной переменной, находящейся в ячейке 0).

После этого значение (по индексу 0) загружается в память с помощью кода операции `ldloc.0` (загрузка локального аргумента по индексу 0) для применения в вызове метода `System.Console.WriteLine()` (представленного кодом операции `call`). И, наконец, возврат из функции обеспечивается посредством кода операции `ret`. Ниже показан (прокомментированный) CIL-код для метода `PrintMessage()` (для краткости из листинга были удалены коды операций `nop`).

```
.method public hidebysig instance void PrintMessage() cil managed
{
    .maxstack 1
    // Определить локальную переменную типа string (по индексу 0).
    .locals init ([0] string myMessage)

    // Загрузить в стек строку со значением "Hello.".
    ldstr " Hello."

    // Сохранить строковое значение из стека в локальной переменной.
    stloc.0
```

```
// Загрузить значение по индексу 0.
ldloc.0
// Вызвать метод с текущим значением.
call void [mscorlib]System.Console::WriteLine(string)
ret
}
```

На заметку! Как видите, CIL поддерживает синтаксис комментариев в виде двойной косой черты (а также синтаксис `/* ... */`). Как и в случае C#, компилятор CIL полностью игнорирует комментарии в коде.

Теперь, когда вы знаете основы директив, атрибутов и кодов операций CIL, давайте приступим непосредственно к программированию на CIL, начав с рассмотрения темы возвратного проектирования.

Возвратное проектирование

В главе 1 было показано, как использовать утилиту `ildasm.exe` для просмотра кода CIL, сгенерированного компилятором C#. Кроме того, эта утилита позволяет сбрасывать код CIL, содержащийся внутри загруженной в `ildasm.exe` сборки, во внешний файл. Полученный подобным образом CIL-код можно легко редактировать и затем компилировать с помощью компилятора CIL (`ilasm.exe`).

На заметку! Вспомните, что для просмотра CIL-кода любой заданной сборки, а также его трансляции в приблизительную кодовую базу C# можно применять утилиту `reflector.exe`.

Формально такой прием называется *возвратным проектированием* и может быть полезным в перечисленных ниже обстоятельствах.

- Вам необходимо модифицировать сборку, исходный код которой больше не доступен.
- Вы работаете с далеким от идеала компилятором языка .NET, который генерирует неэффективный (или совершенно некорректный) код CIL, поэтому нужно изменить кодовую базу.
- Вы конструируете библиотеку взаимодействия с COM и желаете учесть ряд атрибутов COM IDL, которые были утеряны во время процесса преобразования (такие как COM-атрибут `[helpstring]`).

Для иллюстрации применения возвратного проектирования создадим в простом текстовом редакторе новый файл кода C# (`HelloProgram.cs`) и определим в нем показанный ниже класс (при желании можно создать новое консольное приложение в Visual Studio и удалить из него файл `AssemblyInfo.cs`, чтобы уменьшить объем генерируемого CIL-кода).

```
// Простое консольное приложение C#.
using System;
// Обратите внимание, что для упрощения генерируемого CIL-кода
// класс не помещается в пространство имен.
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello CIL code!");
        Console.ReadLine();
    }
}
```


Сохраним этот файл в удобном месте (например, в C:\RoundTrip) и скомпилируем программу с помощью csc.exe:

```
csc HelloProgram.cs
```

Теперь можно открыть файл HelloProgram.exe в ildasm.exe и, выбрав пункт меню File⇒Dump (Файл⇒Сбросить), сохранить низкоуровневый CIL-код в новом файле *.il (HelloProgram.il), расположенном в той же папке, где находится скомпилированная сборка (оставив нетронутыми стандартные настройки в диалоговом окне сохранения).

На заметку! При сбросе содержимого сборки в файл утилита ildasm.exe также сгенерирует файл *.res. Ресурсные файлы подобного рода можно игнорировать (и удалять), поскольку в этой главе они использоваться не будут.

Теперь можно просмотреть файл HelloProgram.il в любом текстовом редакторе. Ниже показано его содержимое (для удобства несколько переформатированное и прокомментированное).

```
// Ссылаемые сборки.
.assembly extern mscorlib
{
    .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
    .ver 4:0:0:0
}

// Наша сборка.
.assembly HelloProgram
{
    /**** Данные TargetFrameworkAttribute удалены для ясности! ****/

    .hash algorithm 0x00008004
    .ver 0:0:0:0
}

.module HelloProgram.exe
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003
.corflags 0x00000003

// Определение класса Program.
.class private auto ansi beforefieldinit Program
    extends [mscorlib]System.Object
{
    .method private hidebysig static void Main(string[] args) cil managed
    {
        // Помечает этот метод как точку входа исполняемой сборки.
        .entrypoint
        .maxstack 8
        IL_0000: nop
        IL_0001: ldstr "Hello CIL code!"
        IL_0006: call void [mscorlib]System.Console::WriteLine(string)
        IL_000b: nop
        IL_000c: call string [mscorlib]System.Console::ReadLine()
        IL_0011: pop
        IL_0012: ret
    }

    // Стандартный конструктор.
    .method public hidebysig specialname rtspecialname
        instance void .ctor() cil managed
```

```

{
    .maxstack 8
    IL_0000: ldarg.0
    IL_0001: call instance void [mscorlib]System.Object::.ctor()
    IL_0006: ret
}
}

```

Прежде всего, обратите внимание, что файл *.il начинается с объявления всех внешних сборок, на которые ссылается текущая скомпилированная сборка. В данном случае присутствует только одна лексема .assembly extern, которая указывает на постоянно присутствующую сборку mscorlib.dll. Конечно, если бы в библиотеке классов использовались типы из других сборок, здесь бы были определены и другие директивы .assembly extern.

Далее идет формальное определение сборки HelloProgram.exe, который был назначен стандартный номер версии 0.0.0.0 (поскольку никакой номер версии не был указан с помощью атрибута [AssemblyVersion]). Сборка дополнительно описывается с применением различных директив CIL (таких как .module, .imagebase и т.п.).

После списка внешних сборок и определения текущей сборки находится определение типа Program. Обратите внимание, что директива .class имеет разнообразные атрибуты (многие из которых в действительности необязательны), например, показанный ниже атрибут extends, который задает базовый класс для типа:

```

.class private auto ansi beforefieldinit Program
    extends [mscorlib]System.Object
{ ... }

```

Большая часть CIL-кода представляет реализацию стандартного конструктора класса и метода Main(); оба они определяются (частично) с помощью директивы .method. После определения этих членов с помощью соответствующих директив и атрибутов, они реализуются с использованием различных кодов операций.

Очень важно запомнить, что при взаимодействии с типами .NET (такими как System.Console) в CIL всегда должно применяться полностью заданное имя нужного типа. Кроме того, полностью заданное имя типа всегда должно снабжаться префиксом в форме дружественного имени сборки, в которой тип определен (в квадратных скобках). Взгляните на следующую реализацию метода Main() в CIL:

```

.method private hidebysig static void Main(string[] args) cil managed
{
    .entrypoint
    .maxstack 8
    IL_0000: nop
    IL_0001: ldstr "Hello CIL code!"
    IL_0006: call void [mscorlib]System.Console::WriteLine(string)
    IL_000b: nop
    IL_000c: call string [mscorlib]System.Console::ReadLine()
    IL_0011: pop
    IL_0012: ret
}

```

В реализации стандартного конструктора на языке CIL используется еще одна инструкция, связанная с загрузкой — ldarg.0. В этом случае значение, загружаемое в стек, представляет собой не специальную переменную, указанную вами, а ссылку на текущий объект (подробнее об этом — чуть позже). Также обратите внимание, что в стандартном конструкторе явно производится вызов конструктора базового класса, которым в данном случае является хорошо знакомый класс System.Object:

```
.method public hidebysig specialname rtspecialname
instance void .ctor() cil managed
{
    .maxstack 8
    IL_0000: ldarg.0
    IL_0001: call instance void [mscorlib]System.Object::.ctor()
    IL_0006: ret
}
```

Роль меток в коде CIL

Вы наверняка заметили, что каждая строка в коде реализации предваряется лексемой в форме `IL_XXX:` (например, `IL_0000:`, `IL_0001:` и т.д.). Такие лексемы называются *метками кода* и могут быть именованы в произвольной манере (при условии, что они не дублируются в рамках одного и того же члена). При сбросе содержимого сборки в файл утилиты `ildasm.exe` автоматически генерирует метки кода, которые следуют соглашению об именовании `IL_XXX:`. Тем не менее, это можно изменить, сделав их более описательными, например:

```
.method private hidebysig static void Main(string[] args) cil managed
{
    .entrypoint
    .maxstack 8
    Nothing_1: nop
    Load_String: ldstr "Hello CIL code!"
    PrintToConsole: call void [mscorlib]System.Console::WriteLine(string)
    Nothing_2: nop
    WaitFor_KeyPress: call string [mscorlib]System.Console::ReadLine()
    RemoveValueFromStack: pop
    Leave_Function: ret
}
```

В действительности метки кода по большей части являются не обязательными. Единственный случай, когда метки кода совершенно необходимы, связан с написанием CIL-кода, в котором используются различные конструкции ветвления или заикливания, поскольку они позволяют указать, куда должен быть направлен поток логики. В текущем примере все автоматически сгенерированные метки кода можно удалить безо всяких последствий:

```
.method private hidebysig static void Main(string[] args) cil managed
{
    .entrypoint
    .maxstack 8
    nop
    ldstr "Hello CIL code!"
    call void [mscorlib]System.Console::WriteLine(string)
    nop
    call string [mscorlib]System.Console::ReadLine()
    pop
    ret
}
```

Взаимодействие с CIL: модификация файла *.il

Теперь, когда вы лучше понимаете, из чего состоит базовый файл CIL, давайте завершим эксперимент с возвратным проектированием.

Наша цель здесь — модифицировать CIL-код в существующем файле *.il, как описано ниже:

- добавить ссылку на сборку System.Windows.Forms.dll;
- загрузить локальную строку внутри метода Main();
- вызвать метод System.Windows.Forms.MessageBox.Show(), используя локальную строковую переменную в качестве аргумента.

Первый шаг заключается в добавлении новой директивы .assembly (снабженной атрибутом extern), которая указывает, что нашей сборке требуется сборка System.Windows.Forms.dll. Для этого необходимо вставить в файл *.il сразу же после ссылки на внешнюю сборку mscorlib следующий код:

```
.assembly extern System.Windows.Forms
{
    .publickeytoken = (B7 7A 5C 56 19 34 E0 89)
    .ver 4:0:0:0
}
```

Значение, присваиваемое директиве .ver, может отличаться в зависимости от версии платформы .NET, установленной на машине разработки. Здесь видно, что используется сборка System.Windows.Forms.dll версии 4.0.0.0, которая имеет значение открытого ключа B77A5C561934E089. Открыв GAC (см. главу 14) и отыскав там версию сборки System.Windows.Forms.dll, можно просто скопировать корректный номер версии и значение открытого ключа.

Теперь понадобится изменить текущую реализацию метода Main(). Для этого необходимо найти этот метод внутри файла *.il и удалить текущий код реализации (оставив только директивы .maxstack и .entrypoint):

```
.method private hidebysig static void Main(string[] args) cil managed
{
    .entrypoint
    .maxstack 8
    // Написать новый код CIL!
}
```

Цель здесь состоит в том, чтобы поместить новую строку в стек и вызвать метод MessageBox.Show() (вместо Console.WriteLine()). Вспомните, что при указании имени внешнего типа должно использоваться его полностью заданное имя (в сочетании с дружественным именем сборки). Также обратите внимание, что в терминах CIL каждый вызов метода документируется полностью заданным возвращаемым типом. В результате метод Main() должен приобрести следующий вид:

```
.method private hidebysig static void Main(string[] args) cil managed
{
    .entrypoint
    .maxstack 8

    ldstr "CIL is way cool"
    call valuetype [System.Windows.Forms]
        System.Windows.Forms.DialogResult
        [System.Windows.Forms]
        System.Windows.Forms.MessageBox::Show(string)
    pop
    ret
}
```

В действительности CIL-код был модифицирован для соответствия показанному ниже определению класса C#:

```
class Program
{
    static void Main(string[] args)
    {
        System.Windows.Forms.MessageBox.Show("CIL is way cool");
    }
}
```

Компиляция CIL-кода с-помощью `ilasm.exe`

После сохранения этого измененного файла `*.il` можно скомпилировать новую сборку .NET с применением утилиты `ilasm.exe` (компилятора CIL). Компилятор CIL поддерживает многочисленные опции командной строки (для их просмотра укажите при запуске опцию `-?`), наиболее интересные из которых описаны в табл. 18.1.

Таблица 18.1. Опции командной строки, наиболее часто применяемые при работе с утилитой `ilasm.exe`

Опция	Описание
<code>/debug</code>	Позволяет включить отладочную информацию (такую как имена локальных переменных и аргументов, а также номера строк)
<code>/dll</code>	Позволяет генерировать в качестве вывода файл <code>*.dll</code>
<code>/exe</code>	Позволяет генерировать в качестве вывода файл <code>*.exe</code> . Это стандартная опция, которая может быть опущена
<code>/key</code>	Позволяет компилировать сборку со строгим именем, используя заданный файл <code>*.snk</code>
<code>/output</code>	Позволяет указать имя и расширение выходного файла. Если флаг <code>/output</code> отсутствует, результирующее имя файла (без расширения) совпадает с именем первого исходного файла

Чтобы скомпилировать измененный файл `HelloProgram.il` в новую .NET-сборку `*.exe`, необходимо ввести в командной строке разработчика следующую команду:

```
ilasm /exe HelloProgram.il /output=NewAssembly.exe
```

Если все прошло успешно, на экране должен появиться следующий отчет:

```
Microsoft (R) .NET Framework IL Assembler. Version 4.0.21006.1
Copyright (c) Microsoft Corporation. All rights reserved.
Assembling 'HelloProgram.il' to EXE --> 'NewAssembly.exe'
Source file is UTF-8

Assembled method Program::Main
Assembled method Program::.ctor
Creating PE file

Emitting classes:
Class 1:      Program

Emitting fields and methods:
Global
Class 1 Methods: 2;

Emitting events and properties:
Global
Class 1
Writing PE file
Operation completed successfully
```

В этот момент новое приложение можно запустить. Конечно же, вместо окна консоли теперь сообщение отображается в отдельном окне сообщения. И хотя этот простой пример не является столь уж впечатляющим, он иллюстрирует одно из практических применений возвратного проектирования CIL.

Роль инструмента `peverify.exe`

При построении либо изменении сборок с использованием кода CIL рекомендуется всегда проверять, является ли скомпилированный двоичный образ правильно оформленным образом .NET, с помощью утилиты командной строки `peverify.exe`:

```
peverify NewAssembly.exe
```

Эта утилита проверит достоверность всех кодов операций CIL внутри указанной сборки. Например, в терминах CIL-кода стек вычислений должен всегда быть пустым перед выходом из функции. Если вы забудете извлечь любые оставшиеся значения, компилятор `ilasm.exe` все равно сгенерирует сборку (поскольку компиляторы заботит только синтаксис). С другой стороны, утилита `peverify.exe` контролирует правильность семантики, поэтому если стек не был очищен перед выходом из функции, она уведомит об этом еще до запуска кодовой базы.

Исходный код. Пример `RoundTrip` доступен в подкаталоге `Chapter 18`.

Директивы и атрибуты CIL

Теперь, когда было показано, как применять утилиты `ildasm.exe` и `ilasm.exe` для возвратного проектирования, можно переходить к более детальному изучению синтаксиса и семантики CIL. В следующих разделах будет поэтапно рассмотрен весь процесс создания специального пространства имен, содержащего набор типов. Однако для простоты пока эти типы не содержат логику реализации своих членов. Разобравшись с созданием простых типов, внимание можно будет переключить на процесс определения “реальных” членов с использованием кодов операций CIL.

Указание ссылок на внешние сборки в CIL

Давайте создадим с помощью текстового редактора новый файл по имени `CILTypes.il`. Первой задачей в проекте CIL является перечисление внешних сборок, которые будут использоваться текущей сборкой. В рассматриваемом примере будут применяться только типы из сборки `mscorlib.dll`. Для этого необходимо указать в новом файле директиву `.assembly` с уточняющим атрибутом `external`. При добавлении ссылки на сборку со строгим именем, такую как `mscorlib.dll`, должны также указываться директивы `.publickeytoken` и `.ver`:

```
.assembly extern mscorlib
{
    .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
    .ver 4:0:0:0
}
```

На заметку! Строго говоря, явно добавлять ссылку на сборку `mscorlib.dll` не обязательно, поскольку компилятор `ilasm.exe` добавит ее автоматически. Для всех остальных внешних библиотек .NET, требуемых в проекте CIL, обязательно должны быть предусмотрены соответствующие директивы `.assembly extern`.

Определение текущей сборки в CIL

Следующий шаг заключается в определении интересующей сборки с использованием директивы `.assembly`. В самом простом варианте сборка может быть определена за счет указания дружественного имени двоичного файла:

```
// Наша сборка.  
.assembly CILTypes { }
```

Хотя это вполне допустимое определение новой сборки .NET, обычно внутри такого объявления размещаются дополнительные директивы. В рассматриваемом примере определение сборки необходимо снабдить номером версии 1.0.0.0 с помощью директивы `.ver` (обратите внимание, что числа в номере версии разделяются двоеточиями, а не точками, как в C#):

```
// Наша сборка.  
.assembly CILTypes  
{  
    .ver 1:0:0:0  
}
```

Из-за того, что `CILTypes` представляет собой однофайловую сборку, в конце определения понадобится поместить одну директиву `.module` и указать в ней официальное имя двоичного файла .NET — `CILTypes.dll`:

```
.assembly CILTypes  
{  
    .ver 1:0:0:0  
}  
// Модуль нашей однофайловой сборки.  
.module CILTypes.dll
```

Помимо `.assembly` и `.module`, существуют и другие директивы CIL, которые позволяют дополнительно уточнять общую структуру создаваемого двоичного файла .NET. В табл. 18.2 перечислены некоторые наиболее часто используемые директивы.

Таблица 18.2. Дополнительные директивы, связанные со сборками

Директива	Описание
<code>.resources</code>	Если сборка использует внутренние ресурсы (такие как растровые изображения или таблицы строк), эта директива позволяет указать имя файла, в котором содержатся ресурсы, подлежащие включению в сборку
<code>.subsystem</code>	Эта директива CIL служит для указания предпочитаемого пользовательского интерфейса, внутри которого должна выполняться сборка. Например, значение 2 указывает, что сборка должна выполняться в приложении с графическим пользовательским интерфейсом, а значение 3 — в консольном приложении

Определение пространств имен в CIL

После определения внешнего вида и поведения сборки (и обязательных внешних ссылок) можно переходить к созданию пространства имен .NET (`MyNamespace`), используя директиву `.namespace`:

```
// Наша сборка имеет единственное пространство имен.  
.namespace MyNamespace { }
```

Подобно C#, определения пространств имен CIL могут быть вложены в другие пространства имен. Хотя в рассматриваемом примере корневое пространство имен не тре-

буется, ради интереса посмотрим, как создать корневое пространство имен под названием MyCompany:

```
.namespace MyCompany
{
    .namespace MyNamespace {}
}
```

Как и C#, язык CIL позволяет определить вложенное пространство имен следующим образом:

```
// Определение вложенного пространства имен.
.namespace MyCompany.MyNamespace {}
```

Определение типов классов в CIL

Пустые пространства имен не особенно интересны, поэтому давайте посмотрим, как в CIL определяются типы классов. Для определения нового типа класса применяется директива `.class`. Тем не менее, эта простая директива может снабжаться многочисленными дополнительными атрибутами, позволяющими уточнить природу типа. В целях иллюстрации добавим в наше пространство имен открытый класс по имени `MyBaseClass`. Как и в C#, если базовый класс явно не указан, тип будет автоматически унаследован от `System.Object`:

```
.namespace MyNamespace
{
    // Предполагается базовый класс System.Object.
    .class public MyBaseClass {}
}
```

При построении типа класса, производного от любого класса, отличного от `System.Object`, используется атрибут `extends`. Для ссылки на тип, определенный внутри той же самой сборки, CIL требует применения полностью заданного имени (хотя если базовый тип находится внутри той же самой сборки, можно не указывать префикс в виде дружественного имени сборки). Таким образом, следующая попытка расширения `MyBaseClass` приводит к ошибке на этапе компиляции:

```
// Этот код не компилируется!
.namespace MyNamespace
{
    .class public MyBaseClass {}
    .class public MyDerivedClass
        extends MyBaseClass {}
}
```

Чтобы правильно определить родительский класс для `MyDerivedClass`, необходимо указать полностью заданное имя `MyBaseClass`:

```
// Правильный код.
.namespace MyNamespace
{
    .class public MyBaseClass {}
    .class public MyDerivedClass
        extends MyNamespace.MyBaseClass {}
}
```

В дополнение к атрибутам `public` и `extends`, определение класса CIL может иметь множество дополнительных спецификаторов, уточняющих видимость типа, компоновку полей и т.п. В табл. 18.3 описаны некоторые атрибуты, которые могут использоваться в сочетании с директивой `.class`.

Таблица 18.3. Разнообразные атрибуты, используемые вместе с директивой `.class`

Атрибут	Описание
<code>public</code> , <code>private</code> , <code>nested assembly</code> , <code>nested famandassem</code> , <code>nested family</code> , <code>nested famorassem</code> , <code>nested public</code> , <code>nested private</code>	Эти атрибуты применяются для указания степени видимости типа. Как не трудно заметить, в CIL предлагается много других возможностей помимо тех, что доступны в C#. За дополнительными сведениями обращайтесь к документу ECMA 335
<code>abstract</code> , <code>sealed</code>	Эти два атрибута могут быть присоединены к директиве <code>.class</code> для определения, соответственно, абстрактного или запечатанного класса
<code>auto</code> , <code>sequential</code> , <code>explicit</code>	Эти атрибуты позволяют указать CLR-среде, как данные полей должны размещаться в памяти. Для типов классов подходящим является стандартный флаг <code>auto</code> . Изменение этой стандартной установки может быть полезно в случае использования <code>P/Invoke</code> для обращения к неуправляемому коду C
<code>extends</code> , <code>implements</code>	Эти атрибуты позволяют определять базовый класс для типа (<code>extends</code>) и реализовать интерфейс в типе (<code>implements</code>)

Определение и реализация интерфейсов в CIL

Как ни странно, но типы интерфейсов в CIL тоже определяются с использованием директивы `.class`. Однако когда эта директива сопровождается атрибутом `interface`, тип трактуется как интерфейсный тип CTS. После определения интерфейс может привязываться к типу класса или структуры с применением атрибута `implements`, как показано ниже:

```
.namespace MyNamespace
{
    // Определение интерфейса.
    .class public interface IMyInterface {}

    // Простой базовый класс.
    .class public MyBaseClass {}

    // MyDerivedClass теперь реализует IMyInterface
    // и расширяет MyBaseClass.
    .class public MyDerivedClass
        extends MyNamespace.MyBaseClass
        implements MyNamespace.IMyInterface {}
}
```

На заметку! Конструкция `extends` должна предшествовать конструкции `implements`. Кроме того, в конструкции `implements` может содержаться разделенный запятыми список интерфейсов.

Как рассказывалось в главе 8, интерфейсы могут выступать в роли базовых для других типов интерфейсов, позволяя строить иерархии интерфейсов. Однако, вопреки возможным ожиданиям, использовать атрибут `extends` для наследования интерфейса А от интерфейса Б в CIL нельзя. Этот атрибут разрешено применять только для указания базового класса типа. Когда требуется расширить какой-нибудь интерфейс, должен использоваться атрибут `implements`:

```
// Расширение интерфейсов в CIL.
.class public interface IMyInterface {}

.class public interface IMyOtherInterface
    implements MyNamespace.IMyInterface {}
```

Определение структур в CIL

Директива `.class` может использоваться для определения любой структуры CTS при условии, что тип расширяет `System.ValueType`. Кроме того, директива `.class` должна сопровождаться атрибутом `sealed` (учитывая, что структуры никогда не могут выступать в роли базовых для других типов значений). В случае несоблюдения этого требования `ilasm.exe` будет сообщать об ошибке на этапе компиляции.

```
// Определение структуры всегда является запечатанным.
.class public sealed MyStruct
    extends [mscorlib]System.ValueType{}
```

В CIL также предусмотрен сокращенный вариант синтаксиса для определения типа структуры. В случае использования атрибута `value` новый тип будет автоматически наследоваться от типа `[mscorlib]System.ValueType`. Следовательно, тип `MyStruct` можно было бы определить и так:

```
// Сокращенный вариант объявления структуры.
.class public sealed value MyStruct{}
```

Определение перечислений в CIL

Перечисления .NET являются производными от класса `System.Enum`, который, в свою очередь, унаследован от `System.ValueType` (и потому также должен быть запечатанным). Чтобы определить перечисление в CIL, необходимо расширить `[mscorlib]System.Enum`, как показано ниже:

```
// Перечисление.
.class public sealed MyEnum
    extends [mscorlib]System.Enum{}
```

Как и структуры, перечисления могут быть определены с помощью сокращенной нотации, использующей атрибут `enum`:

```
// Сокращенный вариант определения перечисления.
.class public sealed enum MyEnum{}
```

Указание пар “имя/значение” перечисления рассматривается далее в главе.

На заметку! Делегаты, которые являются еще одним фундаментальным типом .NET, также имеют специфическое представление CIL. За дополнительными сведениями обращайтесь в главу 10.

Определение обобщений в CIL

Обобщенные типы также имеют собственное представление в синтаксисе CIL. Вспомните из главы 9, что обобщенный тип или член может иметь один и более параметров типа. Например, в типе `List<T>` определен один параметр типа, а в `Dictionary<TKey, TValue>` — два. В CIL количество параметров типа указывается с использованием символа обратной одиночной кавычки (```), за которым следует число. Как и в C#, действительные значения параметров типа заключаются в квадратные скобки.

На заметку! На большинстве клавиатур символ ` находится на клавише, расположенной над клавишей <Tab> (слева от клавиши <1>).

Например, предположим, что необходимо создать переменную `List<T>`, где `T` является типом `System.Int32`. В CIL это делается следующим образом (данный код может находиться внутри контекста любого метода CIL):

```
// В C#: List<int> myInts = new List<int>();
newobj instance void class [mscorlib]
    System.Collections.Generic.List`1<int32>::.ctor()
```

Обратите внимание, что этот обобщенный класс определен как `List`1<int32>`, поскольку `List<T>` имеет единственный параметр типа. А вот как можно определить тип `Dictionary<string, int>`:

```
// В C#: Dictionary<string, int> d = new Dictionary<string, int>();
newobj instance void class [mscorlib]
    System.Collections.Generic.Dictionary`2<string, int32>::.ctor()
```

В качестве еще одного примера предположим, что имеется обобщенный тип, использующий в качестве параметра типа другой обобщенный тип. Соответствующий CIL-код выглядит следующим образом:

```
// В C#: List<List<int>> myInts = new List<List<int>>();
newobj instance void class [mscorlib]
    System.Collections.Generic.List`1<class
    [mscorlib]System.Collections.Generic.List`1<int32>>::.ctor()
```

Компиляция файла `CilTypes.il`

Несмотря на то что к определенным ранее типам пока еще не были добавлены ни члены, ни код реализации, этот файл `*.il` уже можно компилировать в .NET-сборку `*.dll` (что и должно делаться ввиду отсутствия метода `Main()`). Для этого необходимо открыть окно командной строки и ввести следующую команду для запуска `ilasm.exe`:

```
ilasm /dll CilTypes.il
```

Сделав это, можно открыть скомпилированную сборку в `ildasm.exe`, чтобы удостовериться в создании каждого типа. После проверки содержимого сборки следует запустить в ее отношении утилиту `peverify.exe`:

```
peverify CilTypes.dll
```

Обратите внимание, что утилита сообщит о наличии ошибок, поскольку все типы совершенно пусты. Ниже показан частичный вывод:

```
Microsoft (R) .NET Framework PE Verifier. Version 4.0.21006.1
Copyright (c) Microsoft Corporation. All rights reserved.
```

```
[MD]: Error: Value class has neither fields nor size parameter.
[token:0x02000005]
```

Ошибка: Класс значений не имеет ни полей, ни параметра размера.

```
[MD]: Error: Enum has no instance field. [token:0x02000006]
```

Ошибка: Перечисление не имеет полей экземпляра.

```
...
```

Перед тем как приступить к заполнению типа содержимым, необходимо сначала ознакомиться с фундаментальными типами данных CIL.

Соответствия между базовыми классами .NET, C# и CIL

В табл. 18.4 приведены соответствия между базовыми классами .NET, ключевыми словами C# и их представлениями в CIL. Кроме того, для каждого типа CIL приведена сокращенная константная нотация. Как вскоре будет показано, на эти константы часто ссылаются многие коды операций CIL.

Таблица 18.4. Соответствия между базовыми классами .NET, ключевыми словами C# и их представлениями в CIL

Базовый класс .NET	Ключевое слово в C#	Представление CIL	Константная нотация CIL
System.SByte	sbyte	int8	I1
System.Byte	byte	unsigned int8	U1
System.Int16	short	int16	I2
System.UInt16	ushort	unsigned int16	U2
System.Int32	int	int32	I4
System.UInt32	uint	unsigned int32	U4
System.Int64	long	int64	I8
System.UInt64	ulong	unsigned int64	U8
System.Char	char	char	CHAR
System.Single	float	float32	R4
System.Double	double	float64	R8
System.Boolean	bool	bool	BOOLEAN
System.String	string	string	-
System.Object	object	object	-
System.Void	void	void	VOID

На заметку! Типы `System.IntPtr` и `System.UIntPtr` отображаются на представления `int` и `unsigned int` в CIL (это полезно знать, т.к. они интенсивно используются во многих сценариях взаимодействия с COM и P/Invoke).

Определение членов типов в CIL

Как уже известно, типы .NET могут поддерживать разнообразные члены. Перечисления содержат набор пар “имя/значение”. Структуры и классы могут иметь конструкторы, поля, методы, свойства, статические члены и т.д. В предшествующих семнадцати главах вы уже видели частичные определения в CIL упомянутых элементов, но давайте еще раз кратко рассмотрим, как различные члены отображаются на примитивы CIL.

Определение полей данных в CIL

Перечисления, структуры и классы могут поддерживать поля данных. В каждом случае для их определения будет использоваться директива `.field`. Например, давайте добавим в перечисление `MyEnum` следующие три пары “имя/значение” (обратите внимание, что значения указаны в круглых скобках):

```
.class public sealed enum MyEnum
{
    .field public static literal valuetype
    MyNamespace.MyEnum A = int32(0)
    .field public static literal valuetype
    MyNamespace.MyEnum B = int32(1)
    .field public static literal valuetype
    MyNamespace.MyEnum C = int32(2)
}
```

Поля, находящиеся в контексте производного от `System.Enum` типа .NET, уточняются с применением атрибутов `static` и `literal`. Как не трудно догадаться, эти атрибуты указывают, что данные этих полей должны представлять собой фиксированное значение, доступное только из самого типа (например, `MyEnum.A`).

На заметку! Значения, присваиваемые полям в перечислении, могут быть представлены в шестнадцатеричном формате с использованием префикса `0x`.

Конечно, когда нужно определить элемент поля данных внутри класса или структуры, вы не ограничены только открытыми статическими литеральными данными. Например, класс `MyBaseClass` можно было бы модифицировать так, чтобы он поддерживал два закрытых поля данных уровня экземпляра со стандартными значениями:

```
.class public MyBaseClass
{
    .field private string stringField = "hello!"
    .field private int32 intField = int32(42)
}
```

Как и в C#, поля данных класса будут автоматически инициализироваться подходящими стандартными значениями. Чтобы предоставить пользователю объекта возможность задавать собственные значения во время создания закрытых полей данных, потребуется создать специальные конструкторы.

Определение конструкторов типа в CIL

В CTS поддерживается создание конструкторов как уровня экземпляра, так и уровня класса (статических). В CIL конструкторы уровня экземпляра представляются с помощью лексемы `.ctor`, а конструкторы уровня класса — посредством лексемы `.cctor` (class constructor — конструктор класса). Обе эти лексемы CIL должны сопровождаться атрибутами `rtspecialname` (return type special name — специальное имя возвращаемого типа) и `specialname`. Эти атрибуты применяются для обозначения специфической лексемы CIL, которая может интерпретироваться уникальным образом в любом отдельно взятом языке .NET. Например, в языке C# конструкторы не определяют возвращаемый тип, но в CIL возвращаемым значением конструктора на самом деле является `void`:

```
.class public MyBaseClass
{
    .field private string stringField
    .field private int32 intField
```

```
.method public hidebysig specialname rtspecialname
instance void .ctor(string s, int32 i) cil managed
{
    // Добавить код реализации...
}
}
```

Обратите внимание, что директива `.ctor` снабжена атрибутом `instance` (поскольку это не статический конструктор). Атрибуты `cil managed` указывают на то, что внутри этого метода содержится код CIL, а не неуправляемый код, который может использоваться при выполнении запросов `P/Invoke`.

Определение свойств в CIL

Свойства и методы также имеют специфические представления в CIL. В качестве примера модифицируем класс `MyBaseClass` для поддержки открытого свойства по имени `TheString`, написав следующий CIL-код (обратите внимание на использование атрибута `specialname`):

```
.class public MyBaseClass
{
    ...
    .method public hidebysig specialname
instance string get_TheString() cil managed
    {
        // Добавить код реализации...
    }
    .method public hidebysig specialname
instance void set_TheString(string 'value') cil managed
    {
        // Добавить код реализации...
    }
    .property instance string TheString()
    {
        .get instance string
        MyNamespace.MyBaseClass::get_TheString()
        .set instance void
        MyNamespace.MyBaseClass::set_TheString(string)
    }
}
```

В рамках CIL свойство отображается на пару методов, снабженных префиксами `get_` и `set_`. В директиве `.property` применяются связанные с этим директивы `.get` и `.set` для отображения синтаксиса свойств на корректные “специально именованные” методы.

На заметку! Обратите внимание, что входной параметр метода `set` в свойстве заключен в одинарные кавычки и представляет имя лексемы, которая должна использоваться в правой части операции присваивания внутри контекста метода.

Определение параметров членов

В сущности, аргументы в CIL задаются (более или менее) идентично C#. Например, каждый аргумент в CIL определяется за счет указания его типа данных и затем самого его имени. Более того, подобно C#, язык CIL позволяет определять входные, выходные и передаваемые по ссылке параметры. Вдобавок в CIL допускается определять аргументы,

представляющие массивы параметров (равнозначно ключевому слову `params` в C#), и необязательные параметры.

Для целей иллюстрации определения параметров в низкоуровневом CIL предположим, что требуется построить метод, принимающий параметр `int32` (по значению), параметр `int32` (по ссылке), параметр `[mscorlib]System.Collection.ArrayList` и один выходной параметр (типа `int32`). В C# этот метод выглядел бы примерно так:

```
public static void MyMethod(int inputInt,
    ref int refInt, ArrayList ar, out int outputInt)
{
    outputInt = 0; // Просто чтобы удовлетворить требованиям компилятора C#...
}
```

После отображения этого метода на конструкции CIL вы обнаружите, что ссылочные параметры C# помечаются символом амперсанда (&), который присоединяется в виде суффикса к типу параметра (`int32&`).

Выходные параметры также снабжаются суффиксом &, но при этом дополнительно уточняются лексемой `[out]`. Обратите внимание, что если параметр относится к ссылочному типу (в этом случае `[mscorlib]System.Collections.ArrayList`), перед типом данных указывается лексема `class` (не путайте ее с директивой `.class`).

```
.method public hidebysig static void MyMethod(int32 inputInt,
    int32& refInt,
    class [mscorlib]System.Collections.ArrayList ar,
    [out] int32& outputInt) cil managed
{
    ...
}
```

Исследование кодов операций CIL

Последний аспект CIL-кода, который будет рассмотрен в этой главе, связан с ролью различных кодов операций. Вспомните, что код операции — это просто лексема CIL, используемая для построения логики реализации заданного члена. Все коды операций CIL (которых довольно много) могут быть разделены на три обширных категории:

- коды операций, которые управляют потоком выполнения;
- коды операций, которые вычисляют выражения;
- коды операций, которые обращаются к значениям в памяти (через параметры, локальные переменные и т.д.).

В табл. 18.5 приведены описания некоторых наиболее полезных кодов операций, имеющие непосредственное отношение к логике реализации членов (для удобства они сгруппированы по функциональности).

Коды операций из следующей обширной категории (часть которых описана в табл. 18.6) используются для загрузки (заталкивания) аргументов в виртуальный стек выполнения. Обратите внимание, что все эти ориентированные на загрузку коды операций сопровождаются префиксом `ld` (`load` — загрузить).

В дополнение к набору кодов операций, связанных с загрузкой, CIL предоставляет коды операций, которые позволяют явно извлекать из стека самое верхнее значение. Как было показано в нескольких примерах в начале главы, извлечение значения из стека обычно предусматривает его сохранение во временном локальном хранилище для дальнейшего использования (например, это может быть параметр для последующего вызова метода). С учетом этого, многие коды операций, извлекающие текущее значение из виртуального стека выполнения, снабжены префиксом `st` (`store` — сохранить). Некоторые наиболее часто используемые коды операций описаны в табл. 18.7.

Таблица 18.5. Различные коды операций CIL, имеющих отношение к реализации членов

Коды операций	Описание
add, sub, mul, div, rem	Позволяют выполнять сложение, вычитание, умножение и деление двух значений (rem возвращает остаток от деления)
and, or, not, xor	Позволяют выполнять побитовые операции над двумя значениями
ceq, cgt, clt	Позволяют сравнивать два значения в стеке различными способами: • ceq — сравнение на равенство • cgt — сравнение “больше чем” • clt — сравнение “меньше чем”
box, unbox	Применяются для преобразования ссылочных типов в типы значений и наоборот
ret	Применяется для выхода из метода и возврата значения вызывающему коду (при необходимости)
beq, bgt, ble, blt, switch	Применяются (вместе с другими похожими кодами операций) для управления логикой ветвления внутри метода: • beq — позволяет переходить к определенной метке в коде, если при проверке значения оказываются равными • bgt — позволяет переходить к определенной метке в коде, если при проверке одно из значений оказывается больше другого • ble — позволяет переходить к определенной метке в коде, если при проверке одно из значений оказывается меньше или равным другому • blt — позволяет переходить к определенной метке в коде, если при проверке одно из значений оказывается меньше другого Все связанные с ветвлением коды операций требуют указания в CIL-коде метки для перехода в случае, если результат проверки оказывается истинным
call	Применяется для вызова члена заданного типа
newarr, newobj	Позволяют размещать в памяти новый массив или новый объект (соответственно)

Таблица 18.6. Основные коды операций CIL, связанные с загрузкой в стек

Код операции	Описание
ldarg (и многочисленные вариации)	Загружает в стек аргумент метода. Помимо общей формы ldarg (которая работает с индексом, идентифицирующим аргумент), существует множество других вариаций. Например, коды операций ldarg, которые имеют числовой суффикс (ldarg_0), жестко кодируют загружаемый аргумент. Кроме того, вариации ldarg позволяют жестко кодировать тип данных, используя константную нотацию CIL из табл. 18.4 (например, ldarg_I4 для int32), а также тип данных и значение (например, ldarg_I4_5 для загрузки int32 со значением 5)
ldc (и многочисленные вариации)	Загружает в стек константное значение
ldfld (и многочисленные вариации)	Загружает в стек значение поля уровня экземпляра
ldloc (и многочисленные вариации)	Загружает в стек значение локальной переменной
ldobj	Получает все значения, собранные размещенным в куче объектом, и помещает их в стек
ldstr	Загружает в стек строковое значение

Таблица 18.7. Различные коды операций CIL, связанные с извлечением из стека

Код операции	Описание
pop	Удаляет значение, которое в текущий момент находится на вершущке стека вычислений, но не заботится о его сохранении
starg	Сохраняет значение из вершущки стека в аргументе метода с определенным индексом
stloc (и многочисленные вариации)	Извлекает текущее значение из вершущки стека вычислений и сохраняет его в списке локальных переменных с определенным индексом
stobj	Копирует значение указанного типа из стека вычислений по заданному адресу в памяти
stsfld	Заменяет значение статического поля значением из стека вычислений

Следует учитывать, что различные коды операций CIL будут неявно извлекать значения из стека во время решения поставленной задачи. Например, при вычитании одного числа из другого с использованием кода операции `sub` должно быть очевидным, что перед самым вычислением операция `sub` должна извлечь из стека два следующих доступных значения. Результат вычисления будет снова помещен в стек.

Директива `.maxstack`

При написании кода реализации методов на низкоуровневом CIL необходимо помнить о специальной директиве, называемой `.maxstack`. Эта директива позволяет указать максимальное количество переменных, которые могут помещаться в стек в любой момент во время выполнения метода. Директива имеет стандартное значение (8), которое должно подходить для подавляющего большинства создаваемых методов. Однако при желании можно вручную подсчитать количество локальных переменных в стеке и определить значение `.maxstack` явно:

```
.method public hidebysig instance void
  Speak() cil managed
{
  // Во время выполнения этого метода в стеке находится
  // в точности одно значение (строковый литерал).
  .maxstack 1
  ldstr "Hello there..."
  call void [mscorlib]System.Console::WriteLine(string)
  ret
}
```

Объявление локальных переменных в CIL

Теперь давайте посмотрим, как объявить в CIL локальную переменную. Предположим, что необходимо построить в CIL метод по имени `MyLocalVariables()`, не принимающий аргументов и возвращающий `void`, и определить внутри него три локальных переменных типа `System.String`, `System.Int32` и `System.Object`. В C# этот метод выглядел бы следующим образом (вспомните, что локальные переменные не получают стандартных значений и потому перед использованием должны инициализироваться):

```
public static void MyLocalVariables()
{
  string myStr = "CIL code is fun!";
  int myInt = 33;
  object myObj = new object();
}
```

Сконструировать метод `MyLocalVariables()` на CIL можно так, как показано ниже:

```
.method public hidebysig static void MyLocalVariables() cil managed
{
    .maxstack 8
    // Определить три локальных переменных.
    .locals init ([0] string myStr, [1] int32 myInt, [2] object myObj)
    // Загрузить строку в виртуальный стек выполнения.
    ldstr "CIL code is fun!"
    // Извлечь текущее значение и сохранить его в локальной переменной [0].
    stloc.0
    // Загрузить константу типа i4 (сокращенное обозначение для int32)
    // со значением 33.
    ldc.i4 33
    // Извлечь текущее значение и сохранить его в локальной переменной [1].
    stloc.1
    // Создать новый объект и поместить его в стек.
    newobj instance void [mscorlib]System.Object::.ctor()
    // Извлечь текущее значение и сохранить его в локальной переменной [2].
    stloc.2
    ret
}
```

Как видите, первый шаг при размещении локальных переменных в CIL является использование директивы `.locals` в паре с атрибутом `init`. Внутри соответствующих скобок с каждой переменной необходимо ассоциировать определенный числовой индекс (в примере это `[0]`, `[1]` и `[2]`). Каждый индекс идентифицируется типом данных и необязательным именем переменной. После определения локальных переменных значения загружаются в стек (с помощью различных кодов операций, предназначенных для загрузки) и сохраняются в этих локальных переменных (посредством кодов операций, связанных с сохранением).

Отображение параметров на локальные переменные в CIL

Вы уже видели, как объявлять локальные переменные в низкоуровневом CIL с использованием директивы `.local init`; однако осталось еще взглянуть на то, как входные параметры отображаются на локальные методы. Рассмотрим следующий статический метод C#:

```
public static int Add(int a, int b)
{
    return a + b;
}
```

Этот просто выглядящий метод требует немалых комментариев в терминах CIL. Во-первых, входные аргументы (`a` и `b`) должны быть помещены в виртуальный стек выполнения с применением кода операции `ldarg`. Во-вторых, с помощью кода операции `add` будут извлечены следующие два значения из стека, вычислена их сумма, и ее значение опять сохранено в стек. В-третьих, эта сумма извлекается из стека и возвращается вызывающему коду с использованием кода операции `ret`. Дизассемблировав этот метод C# посредством `ildasm.exe`, вы обнаружите множество дополнительных лексем, вставленных `csc.exe`, но основная часть CIL-кода довольно проста:

```
.method public hidebysig static int32 Add(int32 a, int32 b) cil managed
{
    .maxstack 2
    ldarg.0          // Загрузить a в стек.
    ldarg.1          // Загрузить b в стек.
    add              // Сложить оба значения.
    ret
}
```

Скрытая ссылка `this`

Обратите внимание, что для ссылки в CIL-коде на два входных аргумента (a и b) используются их индексные позиции (0 и 1), учитывая, что нумерация этих позиций в виртуальном стеке выполнения начинается с нуля.

При изучении и создании CIL-кода нужно помнить о том, что каждый нестатический метод, который принимает входные аргументы, получает неявный дополнительный входной параметр, представляющий собой ссылку на текущий объект (аналогичный ключевому слову `this` в C#). Если, например, метод `Add()` определен как *нестатический*:

```
// Больше не является статическим!
public int Add(int a, int b)
{
    return a + b;
}
```

то входные аргументы a и b будут загружаться с помощью `ldarg.1` и `ldarg.2` (а не `ldarg.0` и `ldarg.1`). Причина в том, что ячейка с номером 0 будет содержать неявную ссылку `this`. Взгляните на следующий псевдокод:

```
// Это ТОЛЬКО псевдокод!
.method public hidebysig static int32 AddTwoIntParams(
    MyClass_HiddenThisPointer this, int32 a, int32 b) cil managed
{
    ldarg.0    // Загрузить MyClass_HiddenThisPointer в стек.
    ldarg.1    // Загрузить a в стек.
    ldarg.2    // Загрузить b в стек.
    ...
}
```

Представление итерационных конструкций в CIL

Итерационные конструкции в языке программирования C# представляются с помощью таких ключевых слов, как `for`, `foreach`, `while` и `do`, каждое из которых имеет специальное представление в CIL. Для примера рассмотрим следующий классический цикл `for`:

```
public static void CountToTen()
{
    for(int i = 0; i < 10; i++)
    ;
}
```

Вспомните, что для управления ходом выполнения программы на основе условий в CIL применяются коды операций `br` (`br`, `blt` и т.д.). В этом примере предусмотрено условие, согласно которому выполнение цикла `for` должно прекращаться, когда значение локальной переменной `i` становится больше или равно 10. С каждым проходом к значению `i` добавляется 1, после чего проверяемое условие вычисляется заново.

Кроме того, при использовании любого из связанных с ветвлением кодов операций CIL должна быть определена специальная метка (или две) для обозначения места, куда будет произведен переход в случае удовлетворения условия. С учетом всего вышесказанного взглянем на следующий (расширенный) код CIL, сгенерированный утилитой `ildasm.exe` (вместе с автоматически созданными метками):

```
.method public hidebysig static void CountToTen() cil managed
{
    .maxstack 2
```

```

.locals init ([0] int32 i) // Инициализировать локальную целочисленную переменную i.
IL_0000: ldc.i4.0           // Загрузить это значение в стек.
IL_0001: stloc.0           // Сохранить это значение по индексу 0.
IL_0002: br.s IL_0008      // Перейти на метку IL_0008.
IL_0004: ldloc.0           // Загрузить значение переменной по индексу 0.
IL_0005: ldc.i4.1         // Загрузить значение 1 в стек.
IL_0006: add              // Добавить текущее значение в стеке по индексу 0.
IL_0007: stloc.0           // Сохранить это значение по индексу 0.
IL_0008: ldloc.0           // Загрузить значение по индексу 0.
IL_0009: ldc.i4.s 10      // Загрузить значение 10 в стек.
IL_000b: blt.s IL_0004     // Меньше чем? Если да, перейти на метку IL_0004.
IL_000d: ret
}

```

Этот CIL-код начинается с определения локальной переменной типа `int32` и ее загрузки в стек. После этого осуществляются переходы туда и обратно между метками `IL_0008` и `IL_0004`, во время каждого из которых значение `i` увеличивается на 1 и проверяется на предмет того, что оно все еще меньше 10. Если условие нарушается, осуществляется выход из метода.

Исходный код. Пример `CilTypes` доступен в подкаталоге Chapter 18.

Создание сборки .NET на CIL

После ознакомления с синтаксисом и семантикой языка CIL наступило время закрепить изученный материал путем построения приложения .NET с использованием одной только утилиты `ilasm.exe` и текстового редактора по выбору. Это приложение будет состоять из закрытой однофайловой сборки `*.dll`, которая содержит определения двух типов классов, и консольной сборки `*.exe`, взаимодействующей с этими типами.

Построение сборки `CILCars.dll`

В первую очередь необходимо создать сборку `*.dll`, которая будет использоваться клиентом. Откройте текстовый редактор и создайте новый файл `*.il` по имени `CILCars.il`. В этой однофайловой сборке будут использоваться две внешних сборки .NET. Поэтому для начала понадобится модифицировать файл кода следующим образом:

```

// Добавить ссылки на mscorlib.dll и System.Windows.Forms.dll.
.assembly extern mscorlib
{
    .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
    .ver 4:0:0:0
}

.assembly extern System.Windows.Forms
{
    .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
    .ver 4:0:0:0
}

// Определить однофайловую сборку.
.assembly CILCars
{
    .hash algorithm 0x00008004
    .ver 1:0:0:0
}

.module CILCars.dll

```

В этой сборке будут содержаться два класса. Первый из них, `CILCar`, определяет два поля данных (открытых для упрощения примера) и специальный конструктор. Второй класс, `CarInfoHelper`, определяет единственный статический метод по имени `DisplayCarInfo()`, принимающий `CILCar` в качестве параметра и возвращающий `void`. Оба типа находятся в пространстве имен `CILCars`. Класс `CILCar` может быть реализован в CIL следующим образом:

```
// Реализация типа CILCars.CILCar.
.namespace CILCars
{
    .class public auto ansi beforefieldinit CILCar
    extends [mscorlib]System.Object
    {
        // Поля данных CILCar.
        .field public string petName
        .field public int32 currSpeed

        // Специальный конструктор просто позволяет
        // вызываемому коду присваивать поля данных.
        .method public hidebysig specialname rtspecialname
            instance void .ctor(int32 c, string p) cil managed
        {
            .maxstack 8

            // Загрузить первый аргумент в стек и вызвать конструктор базового класса.
            ldarg.0    // объект this, а не int32!
            call instance void [mscorlib]System.Object::.ctor()

            // Загрузить первый и второй аргументы в стек.
            ldarg.0    // объект this
            ldarg.1    // аргумент int32

            // Сохранить самый верхний элемент стека (int 32) в поле currSpeed.
            stfld int32 CILCars.CILCar::currSpeed

            // Загрузить строковый аргумент и сохранить в поле petName.
            ldarg.0    // объект this
            ldarg.2    // аргумент string
            stfld string CILCars.CILCar::petName
            ret
        }
    }
}
```

Помня о том, что действительным первым аргументом для любого нестатического члена является ссылка на текущий объект, в первом блоке CIL просто загружается ссылка на текущий объект и вызывается конструктор базового класса. Затем входные аргументы конструктора помещаются в стек и сохраняются в соответствующих полях данных с помощью кода операции `stfld`.

Теперь давайте реализуем второй класс в этом пространстве имен — `CILCarInfo`. Главным в этом типе является статический метод `Display()`. Роль этого метода заключается в том, чтобы принимать входной параметр `CILCar`, извлекать значения из его полей данных и отображать их в окне сообщений Windows Forms. Ниже приведена полная реализация `CILCarInfo`, а после нее — необходимые пояснения:

```
.class public auto ansi beforefieldinit CILCarInfo
extends [mscorlib]System.Object
{
    .method public hidebysig static void
        Display(class CILCars.CILCar c) cil managed
    {
```

```

.maxstack 8

// Нам нужна локальная строковая переменная.
.locals init ([0] string caption)

// Загрузка строки и входного параметра CILCar в стек.
ldstr "{0}'s speed is:"
ldarg.0

// Поместить в стек значение поля petName из CILCar
// и вызвать статический метод String.Format().
ldfld string CILCars.CILCar::petName
call string [mscorlib]System.String::Format(string, object)
stloc.0

// Загрузить значение поля currSpeed и получить его строковое
// представление (обратите внимание на вызов ToString()).
ldarg.0
ldflda int32 CILCars.CILCar::currSpeed
call instance string [mscorlib]System.Int32::ToString()
ldloc.0

// Вызвать метод MessageBox.Show() с загруженными значениями.
call valuetype [System.Windows.Forms]
    System.Windows.Forms.DialogResult
    [System.Windows.Forms]
    System.Windows.Forms.MessageBox::Show(string, string)

pop
ret
}
}

```

Хотя объем этого CIL-кода немного больше, чем код реализации CILCar, все равно он довольно прямолинеен. Поскольку определяется статический метод, беспокоиться о скрытой ссылке на текущий объект больше не требуется (таким образом, код операции `ldarg.0` действительно загружает входной аргумент CILCar).

Метод начинается с загрузки в стек строки "{0}'s speed is" и следом за ней аргумента CILCar. После этого производится загрузка значения `petName` и вызов статического метода `System.String.Format()` для подстановки на месте метки-заполнителя внутри фигурных скобок дружественного имени CILCar.

Обработка поля `currSpeed` производится согласно той же самой общей процедуре, но на этот раз используется код операции `ldflda`, который загружает в стек адрес аргумента. После этого вызывается метод `System.Int32.ToString()` для преобразования находящегося по указанному адресу значения в строковый тип.

И, наконец, после того, как обе строки сформатированы необходимым образом, вызывается метод `MessageBox.Show()`. Теперь можно скомпилировать новую сборку *.dll с помощью `ilasm.exe`:

```
ilasm /dll CILCars.il
```

и проверить содержащийся внутри нее CIL-код посредством утилиты `peverify.exe`:

```
peverify CILCars.dll
```

Построение сборки CILCarClient.exe

Далее можно создать простую сборку *.exe с методом `Main()`, который будет создавать объект CILCar и передавать его статическому методу `CILCarInfo.Display()`.

Создадим новый файл `CarClient.il`, добавим в него ссылки на внешние сборки `mscorlib.dll` и `CILCars.dll` (не забыв поместить копию последней в каталог клиентского приложения) и определим в нем единственный тип (`Program`), в котором будут

производиться все необходимые манипуляции со сборкой CILCars.dll. Ниже приведен полный код:

```
// Ссылки на внешние сборки.
.assembly extern mscorlib
{
    .publickeytoken = (B7 7A 5C 56 19 34 E0 89)
    .ver 4:0:0:0
}
.assembly extern CILCars
{
    .ver 1:0:0:0
}

// Наша исполняемая сборка.
.assembly CarClient
{
    .hash algorithm 0x00008004
    .ver 1:0:0:0
}
.module CarClient.exe

// Реализация типа Program.
.namespace CarClient
{
    .class private auto ansi beforefieldinit Program
    extends [mscorlib]System.Object
    {
        .method private hidebysig static void
        Main(string[] args) cil managed
        {
            // Пометить точку входа в *.exe.
            .entrypoint
            .maxstack 8

            // Объявить локальную переменную CILCar и поместить
            // значения в стек для вызова конструктора.
            .locals init ([0] class
            [CILCars]CILCars.CILCar myCilCar)
            ldc.i4 55
            ldstr "Junior"

            // Создать новый объект CilCar; сохранить и загрузить ссылку на него.
            newobj instance void
            [CILCars]CILCars.CILCar::.ctor(int32, string)
            stloc.0
            ldloc.0

            // Вызвать метод Display() с передачей ему самого верхнего значения из стека.
            call void [CILCars]
            CILCars.CILCarInfo::Display(
            class [CILCars]CILCars.CILCar)
            ret
        }
    }
}
```

Единственным кодом операции, на который здесь важно обратить внимание, является `.entrypoint`. Как упоминалось ранее в главе, этот код применяется для пометки метода, который должен служить точкой входа в сборке `*.exe`. В действительности, поскольку с помощью `.entrypoint` среда CLR определяет начальный метод для выполнения, этот метод может иметь какое угодно имя, хотя для него было использовано

стандартное имя `Main()`. В остальной части CIL-кода метода `Main()` производятся типичные операции по помещению и извлечению значений из стека.

Однако для создания `CILCar` применяется код операции `.newobj`. В связи с этим вспомните, что для вызова члена типа непосредственно в CIL используется синтаксис в виде двойного двоеточия и, как обычно, указывается полностью заданное имя типа. Теперь можно скомпилировать новый файл с помощью `ilasm.exe`, проверить сборку посредством `peverify.exe` и затем запустить программу. Введите следующие команды:

```
ilasm CarClient.il
peverify CarClient.exe
CarClient.exe
```

Исходный код. Пример `CilCars` доступен в подкаталоге `Chapter 18`.

Динамические сборки

Естественно, процесс построения сложных .NET-приложений на CIL будет довольно неблагодарным трудом. С одной стороны, CIL является чрезвычайно выразительным языком программирования, который позволяет взаимодействовать со всеми программными конструкциями, разрешенными CTS. С другой стороны, написание низкоуровневого кода CIL отнимает много времени и сил и чревато допущением ошибок. Хотя знание — это всегда сила, все же интересно, насколько в действительности важно держать правила синтаксиса CIL в голове. Ответ на этот вопрос зависит от ситуации. Конечно, в большинстве случаев при программировании .NET-приложений просматривать, редактировать или создавать CIL-код совершенно необязательно. Тем не менее, знание основ CIL позволяет перейти к исследованию мира динамических сборок (как противоположности статическим сборкам) и роли пространства имен `System.Reflection.Emit`.

В первую очередь может возникнуть вопрос: чем отличаются статические и динамические сборки? По определению, *статической сборкой* называется двоичная сборка .NET, которая загружается прямо из хранилища на диске, т.е. на момент запроса средой CLR она находится в физическом файле (или в наборе файлов, если сборка многофайловая) где-то на жестком диске. Как не трудно догадаться, при каждой компиляции исходного кода C# в результате получается статическая сборка.

С другой стороны, *динамическая сборка* создается в памяти на лету с использованием типов из пространства имен `System.Reflection.Emit`. Это пространство имен делает возможным построение сборки и ее модулей, определений типов и логики реализации на CIL *во время выполнения*. После этого сборку в памяти можно сохранять в дисковый файл, превращая ее в статическую. Несомненно, для создания динамических сборок с помощью пространства имен `System.Reflection.Emit` необходимо разбираться в природе кодов операций CIL.

Хотя создание динамических сборок является довольно сложной (и нечасто применяемой) задачей программирования, оно полезно в ряде обстоятельств, часть из которых перечислена ниже.

- Вы строите инструмент программирования .NET, который должен быть способен генерировать сборки по требованию на основе пользовательского ввода.
- Вы создаете приложение, которое нуждается в генерации прокси для удаленных типов на лету, основываясь на полученных метаданных.
- Вам необходима возможность загрузки статической сборки и динамической вставки в ее двоичный образ новых типов.

Некоторые аспекты исполняющей среды .NET связаны с генерацией динамических сборок в фоновом режиме. Например, в ASP.NET эта технология применяется для отображения разметки и кода сценариев стороны сервера на объектную модель времени выполнения. В LINQ код тоже может генерироваться на лету на основе различных выражений запросов. Давайте посмотрим, какие типы доступны в пространстве имен `System.Reflection.Emit`.

Исследование пространства имен `System.Reflection.Emit`

Создание динамических сборок требует некоторых знаний кодов операций CIL, однако типы из пространства имен `System.Reflection.Emit` максимально возможно скрывают сложность CIL. Например, вместо того, чтобы напрямую указывать необходимые директивы и атрибуты CIL для определения типа класса, можно просто воспользоваться классом `TypeBuilder`. Еще один класс, `ConstructorBuilder`, позволяет определить новый конструктор уровня экземпляра, без необходимости в выдаче лексем `specialname`, `rtspecialname` или `.ctor`.

Основные члены пространства имен `System.Reflection.Emit` перечислены в табл. 18.8.

Таблица 18.8. Избранные члены пространства имен `System.Reflection.Emit`

Член	Описание
<code>AssemblyBuilder</code>	Используется для создания сборки (*.dll или *.exe) во время выполнения. В сборках *.exe должен обязательно вызываться метод <code>ModuleBuilder.SetEntryPoint()</code> для установки метода, который является точкой входа в модуль. Если точка входа не указана, генерируется файл *.dll
<code>ModuleBuilder</code>	Используется для определения набора модулей внутри текущей сборки
<code>EnumBuilder</code>	Используется для создания типа перечисления .NET
<code>TypeBuilder</code>	Может применяться для создания в модуле классов, интерфейсов, структур и делегатов во время выполнения
<code>MethodBuilder</code> <code>LocalBuilder</code> <code>PropertyBuilder</code> <code>FieldBuilder</code> <code>ConstructorBuilder</code> <code>CustomAttributeBuilder</code> <code>ParameterBuilder</code> <code>EventBuilder</code>	Используются для создания членов типов (таких как методы, локальные переменные, свойства, конструкторы и атрибуты) во время выполнения
<code>ILGenerator</code>	Выдает коды операций CIL внутри указанного члена типа
<code>OpCodes</code>	Предоставляет множество полей, которые отображаются на коды операций CIL. Используется вместе с различными членами <code>System.Reflection.Emit.ILGenerator</code>

В целом типы из пространства имен `System.Reflection.Emit` позволяют представлять низкоуровневые лексемы CIL программным образом во время построения динамической сборки. Многие из них будут использоваться в приведенном ниже примере; однако тип `ILGenerator` заслуживает более детального рассмотрения.

Роль типа `System.Reflection.Emit.ILGenerator`

Роль типа `ILGenerator` заключается во вставке соответствующих кодов операций CIL в заданный член типа. Напрямую создавать объекты `ILGenerator` нельзя, потому что этот тип не имеет открытых конструкторов. Вместо этого объекты `ILGenerator` должны получаться за счет вызова специфических методов типов, связанных с построением (таких как `MethodBuilder` и `ConstructorBuilder`), например:

```
// Получить экземпляр ILGenerator из объекта ConstructorBuilder
// по имени myCtorBuilder.
ConstructorBuilder myCtorBuilder =
    new ConstructorBuilder(/* ...различные аргументы... */);
ILGenerator myCILGen = myCtorBuilder.GetILGenerator();
```

При наличии объекта `ILGenerator` появляется возможность выдавать низкоуровневые коды операций CIL с помощью любых его методов. Некоторые (но не все) методы `ILGenerator` кратко описаны в табл. 18.9.

Таблица 18.9. Избранные методы типа `ILGenerator`

Метод	Описание
<code>BeginCatchBlock()</code>	Начинает блок <code>catch</code>
<code>BeginExceptionBlock()</code>	Начинает блок нефилтρουемого исключения
<code>BeginFinallyBlock()</code>	Начинает блок <code>finally</code>
<code>BeginScope()</code>	Начинает лексический контекст
<code>DeclareLocal()</code>	Объявляет локальную переменную
<code>DefineLabel()</code>	Объявляет новую метку
<code>Emit()</code>	Имеет множество перегруженных версий и позволяет выдавать коды операций CIL
<code>EmitCall()</code>	Помещает в поток CIL код операции <code>call</code> или <code>callvirt</code>
<code>EmitWriteLine()</code>	Выдает вызов <code>Console.WriteLine()</code> со значениями разных типов
<code>EndExceptionBlock()</code>	Завершает блок исключения
<code>EndScope()</code>	Завершает лексический контекст
<code>ThrowException()</code>	Выдает инструкцию для генерации исключения
<code>UsingNamespace()</code>	Указывает пространство имен, которое должно использоваться при вычислении локальных и наблюдаемых значений в текущем активном лексическом контексте

Основным методом `ILGenerator` является `Emit()`, который работает в сочетании с классом `System.Reflection.Emit.OpCodes`. Как упоминалось ранее в этой главе, данный тип открывает множество доступных только для чтения полей, которые отображаются на низкоуровневые коды операций CIL. Исчерпывающее описание полного набора его членов можно найти в онлайн-овой справочной системе, а примеры применения некоторых из них — далее в настоящей главе.

Выдача динамической сборки

Для иллюстрации процесса определения сборки .NET во время выполнения давайте создадим однофайловую динамическую сборку по имени `MyAssembly.dll`. Внутри этого модуля находится класс `HelloWorld`. Класс `HelloWorld` поддерживает стандартный конструктор и специальный конструктор, который используется для установки значения закрытой переменной-члена (`theMessage`) типа `string`. Вдобавок в классе `HelloWorld` определен открытый метод экземпляра по имени `SayHello()`, который выводит приветственное сообщение в стандартный поток ввода-вывода, и еще один метод экземпляра по имени `GetMsg()`, возвращающий внутреннюю закрытую строку. В сущности, мы собираемся программно сгенерировать следующий тип класса:

```
// Этот класс будет создаваться во время выполнения
// с использованием System.Reflection.Emit.
public class HelloWorld
{
    private string theMessage;
    HelloWorld() {}
    HelloWorld(string s) {theMessage = s;}
    public string GetMsg() {return theMessage;}
    public void SayHello()
    {
        System.Console.WriteLine("Hello from the HelloWorld class!");
    }
}
```

Создадим в Visual Studio новое консольное приложение по имени `MyAsmBuilder`, импортируем в него пространства имен `System.Reflection`, `System.Reflection.Emit` и `System.Threading` и определим статический метод по имени `CreateMyAsm()`. Этот метод будет отвечать за решение следующих задач:

- определение характеристик динамической сборки (имя, версия и т.д.);
- реализация типа `HelloClass`;
- сохранение сгенерированной в памяти сборки в физическом файле.

Метод `CreateMyAsm()` принимает в качестве единственного параметра тип `System.AppDomain` и использует его для получения доступа к типу `AssemblyBuilder`, ассоциированному с текущим доменом приложения (домены приложений рассматривались в главе 17). Код метода `CreateMyAsm()` показан ниже.

```
// Тип AppDomain отправляется вызывающим кодом.
public static void CreateMyAsm(AppDomain curAppDomain)
{
    // Установить общие характеристики сборки.
    AssemblyName assemblyName = new AssemblyName();
    assemblyName.Name = "MyAssembly";
    assemblyName.Version = new Version("1.0.0.0");

    // Создать новую сборку в текущем домене приложения.
    AssemblyBuilder assembly =
        curAppDomain.DefineDynamicAssembly(assemblyName,
            AssemblyBuilderAccess.Save);

    // Поскольку строится однофайловая сборка, имя модуля
    // должно совпадать с именем самой сборки.
    ModuleBuilder module =
        assembly.DefineDynamicModule("MyAssembly", "MyAssembly.dll");

    // Определить открытый класс по имени HelloWorld.
    TypeBuilder helloWorldClass = module.DefineType("MyAssembly.HelloWorld",
        TypeAttributes.Public);
```

```

// Определить закрытую переменную-член типа String по имени theMessage.
FieldBuilder msgField =
    helloWorldClass.DefineField("theMessage", Type.GetType("System.String"),
        FieldAttributes.Private);

// Создать специальный конструктор.
Type[] constructorArgs = new Type[1];
constructorArgs[0] = typeof(string);
ConstructorBuilder constructor =
    helloWorldClass.DefineConstructor(MethodAttributes.Public,
        CallingConventions.Standard, constructorArgs);
ILGenerator constructorIL = constructor.GetILGenerator();
constructorIL.Emit(OpCodes.Ldarg_0);
Type objectClass = typeof(object);
ConstructorInfo superConstructor =
    objectClass.GetConstructor(new Type[0]);
constructorIL.Emit(OpCodes.Call, superConstructor);
constructorIL.Emit(OpCodes.Ldarg_0);
constructorIL.Emit(OpCodes.Ldarg_1);
constructorIL.Emit(OpCodes.Stfld, msgField);
constructorIL.Emit(OpCodes.Ret);

// Создать стандартный конструктор.
helloWorldClass.DefineDefaultConstructor(MethodAttributes.Public);
// Создать метод GetMsg().
MethodBuilder getMsgMethod =
    helloWorldClass.DefineMethod("GetMsg", MethodAttributes.Public,
        typeof(string), null);
ILGenerator methodIL = getMsgMethod.GetILGenerator();
methodIL.Emit(OpCodes.Ldarg_0);
methodIL.Emit(OpCodes.Ldfld, msgField);
methodIL.Emit(OpCodes.Ret);

// Создать метод SayHello.
MethodBuilder sayHiMethod = helloWorldClass.DefineMethod("SayHello",
    MethodAttributes.Public, null, null);
methodIL = sayHiMethod.GetILGenerator();
methodIL.EmitWriteLine("Hello from the HelloWorld class!");
methodIL.Emit(OpCodes.Ret);

// Построить класс HelloWorld.
helloWorldClass.CreateType();

// (Дополнительно) сохранить сборку в файле.
assembly.Save("MyAssembly.dll");
}

```

Выдача сборки и набора модулей

Тело метода начинается с установки минимального набора характеристик сборки с применением типов `AssemblyName` и `Version` (из пространства имен `System.Reflection`). После этого производится получение типа `AssemblyBuilder` через метод `AppDomain.DefineDynamicAssembly()` уровня экземпляра (вспомните, что ссылка на `AppDomain` передается методу `CreateMyAsm()` из вызывающего кода):

```

// Установить общие характеристики сборки
// и получить доступ к типу AssemblyBuilder.
public static void CreateMyAsm(AppDomain curAppDomain)
{
    AssemblyName assemblyName = new AssemblyName();
    assemblyName.Name = "MyAssembly";
    assemblyName.Version = new Version("1.0.0.0");
}

```

```
// Создать новую сборку в текущем домене приложения.  
AssemblyBuilder assembly =  
    curAppDomain.DefineDynamicAssembly(assemblyName,  
        AssemblyBuilderAccess.Save);  
...  
}
```

При вызове `AppDomain.DefineDynamicAssembly()` должен быть указан режим доступа к создаваемой сборке, часто используемые значения которого приведены в табл. 18.10.

Таблица 18.10. Часто используемые значения перечисления `AssemblyBuilderAccess`

Значение	Описание
ReflectionOnly	Указывает, что доступ к динамической сборке возможен только посредством рефлексии
Run	Указывает, что динамическая сборка может выполняться в памяти, но не сохраняться на диске
RunAndSave	Указывает, что динамическая сборка может выполняться в памяти и сохраняться на диске
Save	Указывает, что динамическая сборка может сохраняться на диске, но не выполняться в памяти

Следующая задача заключается в определении набора модулей для новой сборки. С учетом того, что в данном случае сборка является однофайловой, необходимо определить только один модуль. При построении многофайловой сборки с использованием метода `DefineDynamicModule()` понадобилось бы указать необязательный второй параметр, который представляет имя заданного модуля (например, `myMod.dotnetmodule`). В случае однофайловой сборки имя модуля совпадает с именем самой сборки. В результате вызова метода `DefineDynamicModule()` возвращается ссылка на действительный тип `ModuleBuilder`:

```
// Однофайловая сборка.  
ModuleBuilder module =  
    assembly.DefineDynamicModule("MyAssembly", "MyAssembly.dll");
```

Роль типа `ModuleBuilder`

Тип `ModuleBuilder` играет ключевую роль во время разработки динамических сборок. Как и можно было ожидать, `ModuleBuilder` поддерживает набор членов, которые позволяют определять типы, содержащиеся внутри модуля (классы, интерфейсы, структуры и т.д.), а также встроенные ресурсы (таблицы строк, изображения и т.п.).

В табл. 18.11 кратко описаны некоторые методы, связанные с созданием. (Обратите внимание, что каждый метод возвращает соответствующий тип, который представляет создаваемый тип.)

Таблица 18.11. Избранные методы типа `ModuleBuilder`

Метод	Описание
<code>DefineEnum()</code>	Используется для выдачи определения перечисления .NET
<code>DefineResource()</code>	Определяет управляемый встроенный ресурс, который должен храниться в данном модуле
<code>DefineType()</code>	Конструирует объект <code>TypeBuilder</code> , который позволяет определять типы значений, интерфейсы и типы классов (включая делегаты)

Основным членом класса `ModuleBuilder` является метод `DefineType()`. Помимо указания имени типа (в виде простой строки), в нем может использоваться перечисление `System.Reflection.TypeAttributes` для описания формата этого типа. В табл. 18.12 перечислены наиболее важные члены перечисления `TypeAttributes`.

Таблица 18.12. Избранные члены перечисления `TypeAttributes`

Член	Описание
<code>Abstract</code>	Указывает, что тип является абстрактным
<code>Class</code>	Указывает, что тип является классом
<code>Interface</code>	Указывает, что тип является интерфейсом
<code>NestedAssembly</code>	Указывает, что класс находится в области видимости сборки, а потому доступен только методам внутри этой сборки
<code>NestedFamAndAssem</code>	Указывает, что класс находится в области видимости сборки и семейства, а потому доступен только методам, которые относятся к пересечению этого семейства и сборки
<code>NestedFamily</code>	Указывает, что класс находится в области видимости семейства, а потому доступен только методам, которые содержатся внутри собственного типа и любых подтипов
<code>NestedFamORAssem</code>	Указывает, что класс находится в области видимости семейства или сборки, а потому доступен только методам, которые относятся к объединению этого семейства и сборки
<code>NestedPrivate</code>	Указывает, что класс является вложенным и закрытым
<code>NestedPublic</code>	Указывает, класс является вложенным и открытым
<code>NotPublic</code>	Указывает класс, что класс не является открытым
<code>Public</code>	Указывает, что класс является открытым
<code>Sealed</code>	Указывает, что класс является конкретным и не может быть расширен
<code>Serializable</code>	Указывает, что класс может быть сериализован

Выдача типа `HelloClass` и строковой переменной-члена

Теперь, когда вы понимаете роль метода `ModuleBuilder.CreateType()`, давайте посмотрим, как можно выдать открытый тип класса `HelloWorld` и закрытую строковую переменную:

```
// Определить открытый класс по имени MyAssembly.HelloWorld.
TypeBuilder helloWorldClass = module.DefineType("MyAssembly.HelloWorld",
    TypeAttributes.Public);

// Определить закрытую переменную-член типа String по имени theMessage.
FieldBuilder msgField =
    helloWorldClass.DefineField("theMessage",
        typeof(string),
        FieldAttributes.Private);
```

Обратите внимание, что метод `TypeBuilder.DefineField()` предоставляет доступ к типу `FieldBuilder`. В классе `TypeBuilder` определены и дополнительные методы, которые обеспечивают доступ к другим типам-построителям. Например, метод `DefineConstructor()` возвращает тип `ConstructorBuilder`, метод `DefineProperty()` — тип `PropertyBuilder` и т.д.

Выдача конструкторов

Как упоминалось ранее, для определения конструктора текущего типа можно использовать метод `TypeBuilder.DefineConstructor()`. Однако когда дело доходит до реализации конструктора `HelloClass`, в тело конструктора необходимо вставить низкоуровневый CIL-код, который будет отвечать за присваивание входного параметра внутренней закрытой строке. Для получения типа `ILGenerator` понадобится вызвать метод `GetILGenerator()` из соответствующего типа-построителя (`ConstructorBuilder` в этом случае).

Метод `Emit()` класса `ILGenerator` — это сущность, которая отвечает за помещение CIL-кода в реализацию членов. В методе `Emit()` часто используется тип класса `OpCodes`, который предоставляет доступ к набору кодов операций CIL посредством предназначенных только для чтения свойств. Например, свойство `OpCodes.Ret` сигнализирует о возврате из вызова метода; `OpCodes.Stfld` позволяет выполнять присваивание значения переменной экземпляра; `OpCodes.Call` применяется для вызова заданного метода (в этом случае — конструктора базового класса). С учетом всего сказанного, логика для реализации конструктора будет выглядеть следующим образом:

```
// Создать специальный конструктор, принимающий
// единственный аргумент типа System.String.
Type[] constructorArgs = new Type[1];
constructorArgs[0] = typeof(string);
ConstructorBuilder constructor =
    helloWorldClass.DefineConstructor(MethodAttributes.Public,
        CallingConventions.Standard, constructorArgs);

// Выдать необходимый код CIL для помещения в конструктор.
ILGenerator constructorIL = constructor.GetILGenerator();
constructorIL.Emit(OpCodes.Ldarg_0);
Type objectClass = typeof(object);
ConstructorInfo superConstructor = objectClass.GetConstructor(new Type[0]);
constructorIL.Emit(OpCodes.Call, superConstructor);    // Вызвать конструктор
                                                         // базового класса.

// Загрузить в стек указатель this объекта.
constructorIL.Emit(OpCodes.Ldarg_0);

// Загрузить входной аргумент в стек и сохранить его в msgField.
constructorIL.Emit(OpCodes.Ldarg_1);
constructorIL.Emit(OpCodes.Stfld, msgField);           // Присвоить msgField.
constructorIL.Emit(OpCodes.Ret);                       // Произвести возврат.
```

Как теперь уже хорошо известно, в результате определения специального конструктора для типа стандартный конструктор незаметно удаляется. Чтобы снова определить конструктор без аргументов, достаточно просто вызвать метод `DefineDefaultConstructor()` на типе `TypeBuilder`:

```
// Вставить заново стандартный конструктор.
helloWorldClass.DefineDefaultConstructor(MethodAttributes.Public);
```

Этот единственный вызов выдает стандартный CIL-код, используемый для определения стандартного конструктора:

```
.method public hidebysig specialname rtspecialname
    instance void .ctor() cil managed
{
    .maxstack 1
    ldarg.0
    call instance void [mscorlib]System.Object::.ctor()
    ret
}
```

Выдача метода SayHello()

И, наконец, давайте рассмотрим процесс выдачи метода SayHello(). Первая задача заключается в получении типа MethodBuilder из переменной helloWorldClass. После этого можно определить сам метод и получить лежащий в основе тип ILGenerator для вставки необходимых CIL-инструкций:

```
// Создать метод SayHello.
MethodBuilder sayHiMethod =
    helloWorldClass.DefineMethod("SayHello",
        MethodAttributes.Public, null, null);
methodIL = sayHiMethod.GetILGenerator();

// Вывести строку на консоль.
methodIL.EmitWriteLine("Hello there!");
methodIL.Emit(OpCodes.Ret);
```

Здесь был определен открытый метод (MethodAttributes.Public), не принимающий параметров и ничего не возвращающий (на что указывают значения null в вызове DefineMethod()). Также обратите внимание на вызов EmitWriteLine(). Этот вспомогательный член класса ILGenerator автоматически записывает строку в стандартный поток вывода с минимальными усилиями.

Использование динамически сгенерированной сборки

При наличии готовой логики для создания и сохранения сборки, осталось только построить класс, который будет ее запускать. Определим в текущем проекте еще один класс по имени AsmReader. В коде Main() с помощью метода Thread.GetDomain() необходимо получить ссылку на текущий домен приложения, который будет применяться для размещения динамически создаваемой сборки. После получения ссылки можно будет вызывать метод CreateMyAsm().

Чтобы сделать пример немного интереснее, после вызова CreateMyAsm() воспользуемся поздним связыванием (см. главу 15) для загрузки вновь созданной сборки в память и взаимодействия с членами класса HelloWorld. Модифицируем метод Main(), как показано ниже.

```
static void Main(string[] args)
{
    Console.WriteLine("***** The Amazing Dynamic Assembly Builder App *****");

    // Получить домен приложения для текущего потока.
    AppDomain curAppDomain = Thread.GetDomain();

    // Создать динамическую сборку с помощью нашего вспомогательного метода.
    CreateMyAsm(curAppDomain);
    Console.WriteLine("-> Finished creating MyAssembly.dll.");

    // Загрузить новую сборку из файла.
    Console.WriteLine("-> Loading MyAssembly.dll from file.");
    Assembly a = Assembly.Load("MyAssembly");

    // Получить тип HelloWorld.
    Type hello = a.GetType("MyAssembly.HelloWorld");

    // Создать объект HelloWorld и вызвать корректный конструктор.
    Console.WriteLine("-> Enter message to pass HelloWorld class: ");
    string msg = Console.ReadLine();
    object[] ctorArgs = new object[1];
    ctorArgs[0] = msg;
    object obj = Activator.CreateInstance(hello, ctorArgs);
```



```
// Вызвать SayHello и отобразить возвращенную строку.
Console.WriteLine("-> Calling SayHello() via late binding.");
MethodInfo mi = hello.GetMethod("SayHello");
mi.Invoke(obj, null);

// Вызвать метод.
mi = hello.GetMethod("GetMsg");
Console.WriteLine(mi.Invoke(obj, null));
}
```

На самом деле только что была построена сборка .NET, которая способна создавать и запускать другие сборки .NET во время выполнения. На этом исследование языка CIL и роли динамическихборок завершено. Эта глава должна была помочь углубить знания системы типов .NET, а также синтаксиса и семантики языка CIL.

Исходный код. Проект `DynamicAsmBuilder` доступен в подкаталоге `Chapter 18`.

Резюме

В этой главе был представлен краткий обзор синтаксиса и семантики языка CIL. В отличие от управляемых языков более высокого уровня, таких как C#, в CIL не просто определяется набор ключевых слов, а предоставляются директивы (используемые для определения структуры сборки и ее типов), атрибуты (дополнительно уточняющие нужные директивы) и коды операций (которые применяются для реализации членов типов).

Было продемонстрировано несколько инструментов, связанных с программированием на CIL, а также показано, как изменять содержимое сборки .NET за счет добавления новых CIL-инструкций с использованием возвратного проектирования. Кроме того, рассматривались способы определения в CIL текущей (и ссылаемой сборки), пространств имен, типов и членов. Был предложен простой пример создания библиотеки кода .NET и исполняемого файла с применением только CIL и соответствующих инструментов командной строки.

В конце главы приводился краткий обзор процесса создания динамической сборки. Используя пространство имен `System.Reflection.Emit`, сборку .NET можно определить в памяти во время выполнения. Однако, как было указано, работа с этим пространством имен требует знания семантики CIL. Хотя необходимость в построении динамическихборок при разработке большинства приложений .NET возникает довольно редко, такие сборки могут быть очень полезны при создании инструментальных средств поддержки компиляции и других утилит для программирования.

часть VI

Введение в библиотеки базовых классов .NET

В этой части

Глава 19. Многопоточное, параллельное и асинхронное программирование

Глава 20. Файловый ввод-вывод и сериализация объектов

Глава 21. ADO.NET, часть I: подключенный уровень

Глава 22. ADO.NET, часть II: автономный уровень

Глава 23. ADO.NET, часть III: Entity Framework

Глава 24. Введение в LINQ to XML

Глава 25. Введение в Windows Communication Foundation

Глава 26. Введение в Windows Workflow Foundation

Многопоточное, параллельное и асинхронное программирование

Вряд ли многим нравится работать с приложением, которое притормаживает во время выполнения. Аналогично, никто не будет в восторге от того, что запуск некоторой задачи в приложении (возможно, щелчком на элементе в панели инструментов) снижает отзывчивость других частей приложения. До выпуска платформы .NET построение приложений, способных выполнять сразу несколько задач, требовало написания очень сложного кода на языке C++, в котором использовались API-интерфейсы многопоточности Windows. К счастью, платформа .NET предложила множество способов построения программного обеспечения, которое может выполнять сложные операции по уникальным путям выполнения, с намного меньшими сложностями.

Эта глава начинается с определения общей природы “многопоточного приложения”. Затем мы обратимся к уже известному типу делегата .NET, чтобы исследовать его внутреннюю поддержку *асинхронных вызовов методов*. Как вы увидите, данный прием позволяет вызвать метод во вторичном потоке выполнения без необходимости в ручном создании и конфигурировании самого потока.

После этого будет представлено первоначальное пространство имен для многопоточности, которое поставляется, начиная с версии .NET 1.0, и называется `System.Threading`. Вы ознакомитесь с многочисленными типами (`Thread`, `ThreadStart` и т.д.), которые позволяют явно создавать дополнительные потоки выполнения и синхронизировать разделяемые ресурсы, обеспечивая совместное использование данных несколькими потоками в неизменчивой манере.

В оставшихся разделах этой главы будут представлены три последних технологии, которые разработчики .NET-приложений могут применять для построения многопоточного программного обеспечения, в частности — библиотека `Task Parallel Library (TPL)`, технология `PLINQ (Parallel LINQ)` и новые ключевые слова языка C#, связанные с асинхронной обработкой (`async` и `await`). Вы увидите, что эти средства помогают существенно упростить процесс создания отзывчивых многопоточных прикладных приложений.

Отношения между процессом, доменом приложения, контекстом и потоком

В главе 17 *поток* был определен как путь выполнения внутри исполняемого приложения. Хотя многие .NET-приложения могут успешно и продуктивно работать, будучи однопоточными, первичный поток сборки (создаваемый CLR-средой при выполнении `Main()`) в любое время может создавать вторичные потоки для выполнения дополнительных единиц работы. За счет создания дополнительных потоков можно строить более отзывчивые (но не обязательно быстрее выполняемые на одноядерных машинах) приложения.

Пространство имен `System.Threading` появилось в версии .NET 1.0 и предлагает один из подходов к построению многопоточных приложений. Главным в этом пространстве имен, пожалуй, можно назвать класс `Thread`, поскольку он представляет отдельный поток. Чтобы программно получить ссылку на поток, который в текущий момент выполняет заданный член, просто обратитесь к статическому свойству `Thread.CurrentThread`:

```
static void ExtractExecutingThread()
{
    // Получить поток, выполняющий данный метод.
    Thread currThread = Thread.CurrentThread;
}
```

В рамках платформы .NET *не существует* прямого соответствия “один к одному” между доменами приложений (`AppDomain`) и потоками. Фактически определенный домен приложения может иметь несколько потоков, выполняющихся в каждый конкретный момент времени. Более того, конкретный поток не привязан к одному домену приложения на протяжении своего времени жизни. Потоки могут пересекать границы доменов приложений, когда это посчитают целесообразным планировщик потоков Windows и CLR-среда .NET.

Несмотря на то что активные потоки могут перемещаться между границами доменов приложений, каждый поток в любой конкретный момент времени может выполняться только внутри одного домена приложения (другими словами, одиночный поток не может выполнять работу в более чем одном домене приложения одновременно). Чтобы программно получить доступ к домену приложения, в котором размещен текущий поток, вызовите статический метод `Thread.GetDomain()`:

```
static void ExtractAppDomainHostingThread()
{
    // Получить домен приложения, размещающий текущий поток.
    AppDomain ad = Thread.GetDomain();
}
```

Одиночный поток также в любой момент может быть перемещен в определенный контекст, и он может перемещаться в пределах нового контекста по прихоти CLR-среды. Для получения текущего контекста, в котором выполняется поток, используйте статическое свойство `Thread.CurrentContext` (которое возвращает объект `System.Runtime.Remoting.Contexts.Context`):

```
static void ExtractCurrentThreadContext()
{
    // Получить контекст, в котором работает текущий поток.
    Context ctx = Thread.CurrentContext;
}
```

Еще раз: за перемещение потоков между доменами приложений и контекстами отвечает CLR-среда. Как разработчик .NET, вы всегда остаетесь в счастливом неведении

относительно того, где завершается каждый конкретный поток (или, в точности, когда он помещается в новые границы). Тем не менее, вы должны быть осведомлены о различных способах получения лежащих в основе примитивов.

Проблема параллелизма

Один из многих болезненных аспектов многопоточного программирования связан с ограниченным контролем над использованием потоков операционной системой или CLR-средой. Например, написав блок кода, который создает новый поток выполнения, нельзя гарантировать, что этот поток запустится немедленно. Вместо этого такой код лишь просит операционную систему запустить поток, как только это будет возможно (обычно, когда планировщик потоков доберется до него).

Более того, учитывая, что потоки могут перемещаться между границами приложений и контекстов, когда это нужно CLR, вы должны представлять, какие аспекты приложения являются *изменчивыми в потоках* (например, подвергаются многопоточному доступу), а какие операции — *атомарными* (изменчивые в потоках операции опасны).

Чтобы проиллюстрировать проблему, предположим, что поток вызывает метод специфичного объекта. Теперь представьте, что этот поток приостановлен планировщиком потока, чтобы позволить другому потоку обратиться к тому же методу того же самого объекта.

Если исходный поток еще не полностью завершил свою операцию, второй входящий поток может увидеть объект в частично модифицированном состоянии. В этот момент второй поток, по сути, читает фиктивные данные, что определенно может привести к очень странным (и трудно обнаруживаемым) ошибкам, воспроизвести и отладить которые даже еще труднее.

С другой стороны, атомарные операции всегда безопасны в многопоточной среде. К сожалению, очень мало операций в библиотеках базовых классов .NET являются гарантированно атомарными. Даже операция присваивания переменной-члену не является атомарной! Если в документации .NET Framework 4.5 SDK специально не сказано, что операция атомарна, ее следует считать изменчивой в потоках и предпринимать соответствующие меры предосторожности.

Роль синхронизации потоков

Сейчас уже должно быть ясно, что многопоточные программы сами по себе довольно изменчивы, поскольку множество потоков могут оперировать общими разделяемыми ресурсами (более или менее) одновременно. Чтобы защитить ресурсы приложений от возможного повреждения, разработчики .NET должны использовать любое количество потоковых примитивов (таких как блокировки, мониторы, атрибут [Synchronization] или поддержка языковых ключевых слов) для контроля доступа среди выполняющихся потоков.

Хотя платформа .NET не может полностью скрыть сложности, связанные с построением надежных многопоточных приложений, этот процесс все же значительно упрощен. Используя типы, определенные внутри пространства имен `System.Threading`, библиотеку `Task Parallel Library (TPL)` в .NET 4.0 и выше, а также ключевые слова `async` и `await` языка C# в .NET 4.5, можно работать с множеством потоков, прикладывая минимальные усилия.

Прежде чем углубляться в пространство имен `System.Threading`, библиотеку TPL и ключевые слова `async` и `await` языка C#, мы начнем с выяснения того, как можно использовать тип делегата .NET для вызова метода в асинхронной манере. Хотя, безусловно, верно то, что в .NET 4.5 ключевые слова `async` и `await` предлагают более простую альтернативу асинхронным делегатам, по-прежнему важно знать, каким образом

взаимодействовать с кодом, в котором применяется этот подход (поверьте, сейчас в производстве находится масса кода, в котором используются асинхронные делегаты).

Краткий обзор делегатов .NET

Вспомните, что делегат .NET — это по существу безопасный в отношении типов, объектно-ориентированный указатель на функцию. На объявление типа делегата .NET компилятор C# отвечает построением запечатанного класса, производного от `System.MulticastDelegate` (который, в свою очередь, унаследован от `System.Delegate`). Эти базовые классы предоставляют каждому делегату возможность поддерживать список адресов методов, которые могут быть вызваны в более позднее время. Рассмотрим следующий делегат `BinaryOp`, впервые определенный в главе 10:

```
// Тип делегата C#.
public delegate int BinaryOp(int x, int y);
```

Исходя из определения, `BinaryOp` может указывать на любой метод, принимающий два целых числа (по значению) в качестве аргументов и возвращающий целое число. После компиляции сборка с определением делегата будет содержать полноценное определение класса, сгенерированного динамически на основе объявления делегата при компиляции проекта. В случае `BinaryOp` этот класс более или менее похож на следующий (записан в псевдокоде):

```
public sealed class BinaryOp : System.MulticastDelegate
{
    public BinaryOp(object target, uint functionAddress);
    public int Invoke(int x, int y);
    public IAsyncResult BeginInvoke(int x, int y,
        AsyncCallback cb, object state);
    public int EndInvoke(IAsyncResult result);
}
```

Как вы должны помнить, сгенерированный метод `Invoke()` используется для вызова методов, поддерживаемых объектом делегата, в *синхронной манере*. В этом случае вызывающий поток (такой как первичный поток приложения) должен будет ждать, пока не завершится вызов делегата. Также вспомните, что в языке C# метод `Invoke()` не нужно вызывать в коде напрямую — он может быть инициирован неявно, “за кулисами”, при применении “нормального” синтаксиса вызова методов.

Рассмотрим следующий пример консольного приложения (`SyncDelegateReview`), в котором статический метод `Add()` вызывается в синхронном (т.е. блокирующем) режиме (не забудьте импортировать в файл кода C# пространство имен `System.Threading`, т.к. будет вызываться метод `Thread.Sleep()`):

```
namespace SyncDelegateReview
{
    public delegate int BinaryOp(int x, int y);
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** Synch Delegate Review *****");

            // Вывести идентификатор выполняющегося потока.
            Console.WriteLine("Main() invoked on thread {0}.",
                Thread.CurrentThread.ManagedThreadId);

            // Вызвать Add() в синхронном режиме.
            BinaryOp b = new BinaryOp(Add);
```

```

    // Можно было бы также написать b.Invoke(10, 10);
    int answer = b(10, 10);

    // Эти строки не будут выполняться до тех пор,
    // пока не завершится метод Add().
    Console.WriteLine("Doing more work in Main()!");
    Console.WriteLine("10 + 10 is {0}.", answer);
    Console.ReadLine();
}

static int Add(int x, int y)
{
    // Вывести идентификатор выполняющегося потока.
    Console.WriteLine("Add() invoked on thread {0}.",
        Thread.CurrentThread.ManagedThreadId);

    // Организовать паузу для моделирования длительной операции.
    Thread.Sleep(5000);
    return x + y;
}
}
}

```

Внутри метода `Add()` вызывается статический метод `Thread.Sleep()` для приостановки вызывающего потока примерно на 5 секунд, чтобы имитировать длительную задачу. Учтявая, что метод `Add()` вызывается в *синхронной* манере, метод `Main()` не будет выводить результат операции до тех пор, пока не завершится метод `Add()`.

Обратите внимание, что метод `Man()` получает доступ к текущему потоку (через `Thread.CurrentThread`) и выводит идентификатор потока, взяв его из свойства `ManagedThreadId`. Та же самая логика повторяется в статическом методе `Add()`. Как и можно было ожидать, учтявая, что вся работа в этом приложении выполняется исключительно в первичном потоке, на консоль будет выведено одно и то же значение идентификатора:

```

***** Synch Delegate Review *****
Main() invoked on thread 1.
Add() invoked on thread 1.
Doing more work in Main()!
10 + 10 is 20.

Press any key to continue . . .

```

Запустив эту программу, вы должны заметить пятисекундную задержку перед тем, как выполнится последний вызов `Console.WriteLine()` в `Main()`. Хотя многие (если не большинство) методов могут вызываться синхронно без болезненных последствий, при необходимости делегаты .NET позволяют вызывать назначенные им методы асинхронным образом.

Исходный код. Проект `SyncDelegateReview` доступен в подкаталоге `Chapter 19`.

Асинхронная природа делегатов

Если для вас тема многопоточности в новинку, может возникнуть вопрос: что собой представляет *асинхронный* вызов метода? Как известно, некоторые программные операции требуют значительного времени для своего завершения. Приведенный выше метод `Add()` предназначен только для целей иллюстрации, но предположим, что строится однопоточное приложение, вызывающее метод на удаленном объекте, который выполняет длительный запрос к базе данных, загружает большой документ либо выводит

500 строк текста во внешний файл. На протяжении выполнения этих операций приложение может “замереть” на некоторый период времени. И пока эта задача не будет завершена, все другие поведенческие аспекты программы (вроде активизации пунктов меню, щелчков в панели инструментов или вывода на консоль) приостанавливаются (что может раздражать пользователей).

Отсюда возникает вопрос: как заставить делегат запустить назначенный ему метод в отдельном потоке выполнения, чтобы имитировать “одновременное” выполнение многочисленных задач? К счастью, каждый тип делегата .NET автоматически оснащен такой возможностью. А еще лучше то, вы не обязаны углубляться в детали типов пространства имен `System.Threading`, чтобы делать это (хотя эти сущности могут успешно работать рука об руку).

Методы `BeginInvoke()` и `EndInvoke()`

Когда компилятор C# обрабатывает ключевое слово `delegate`, он динамически генерирует класс, определяющий два метода с именами `BeginInvoke()` и `EndInvoke()`. Учитывая определение делегата `BinaryOp`, эти методы будут прототипированы следующим образом:

```
public sealed class BinaryOp : System.MulticastDelegate
{
    ...
    // Используется для асинхронного вызова метода.
    public IAsyncResult BeginInvoke(int x, int y,
        AsyncCallback cb, object state);

    // Используется для получения возвращаемого значения
    // вызванного метода.
    public int EndInvoke(IAsyncResult result);
}
```

Первый набор параметров, переданный `BeginInvoke()`, будет основан на формате делегата C# (два целых числа в случае `BinaryOp`). Последними двумя аргументами всегда будут `System.AsyncCallback` и `System.Object`. Чуть позже вы узнаете о роли этих параметров, а пока просто передадим `null` в каждом из них. Также обратите внимание, что возвращаемое значение `EndInvoke()` является целочисленным, как и тип возврата `BinaryOp`, хотя единственный параметр этого метода всегда имеет тип `IAsyncResult`.

Интерфейс `System.IAsyncResult`

Метод `BeginInvoke()` всегда возвращает объект, реализующий интерфейс `IAsyncResult`, в то время как `EndInvoke()` требует единственный параметр совместимого с `IAsyncResult` типа. Совместимый с `IAsyncResult` объект, возвращаемый из `BeginInvoke()` — это в основном связывающий механизм, который позволяет вызывающему потоку получить результат вызова асинхронного метода через `EndInvoke()` в более позднее время. Интерфейс `IAsyncResult` (находящийся в пространстве имен `System`) определен следующим образом:

```
public interface IAsyncResult
{
    object AsyncState { get; }
    WaitHandle AsyncWaitHandle { get; }
    bool CompletedSynchronously { get; }
    bool IsCompleted { get; }
}
```


В простейшем случае можно избежать прямого обращения к этим членам. Все, что потребуется сделать — это кешировать совместимый с `IAsyncResult` объект, возвращенный методом `BeginInvoke()`, и передать его методу `EndInvoke()`, когда имеется готовность получить результат вызова метода. Как будет показано, члены совместимого с `IAsyncResult` объекта можно вызывать, когда требуется “большая вовлеченность” в процесс получения возвращаемого методом значения.

На заметку! Метод, возвращающий `void`, можно просто вызвать асинхронно и забыть. В таких случаях нет необходимости кешировать совместимый с `IAsyncResult` объект или вызывать `EndInvoke()`, поскольку нет возвращаемого значения, которое требуется получить.

Асинхронный вызов метода

Чтобы указать делегату `BinaryOp` на необходимость вызова `Add()` асинхронным образом, модифицируем логику предыдущего проекта (можно добавить код к имеющемуся проекту, однако в коде примеров для настоящей главы доступно новое консольное приложение по имени `AsyncDelegate`). Обновите предыдущий метод `Main()` следующим образом:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Async Delegate Invocation *****");
    // Вывести идентификатор выполняющегося потока.
    Console.WriteLine("Main() invoked on thread {0}.",
        Thread.CurrentThread.ManagedThreadId);

    // Вызвать Add() во вторичном потоке.
    BinaryOp b = new BinaryOp(Add);
    IAsyncResult iftAR = b.BeginInvoke(10, 10, null, null);

    // Выполнить другую работу в первичном потоке...
    Console.WriteLine("Doing more work in Main()!");

    // Получить результат метода Add() по готовности.
    int answer = b.EndInvoke(iftAR);
    Console.WriteLine("10 + 10 is {0}.", answer);
    Console.ReadLine();
}
```

После запуска этого приложения на консоль выводятся два уникальных идентификатора потоков, поскольку в текущем домене приложения функционируют несколько потоков:

```
***** Async Delegate Invocation *****
Main() invoked on thread 1.
Doing more work in Main()!
Add() invoked on thread 3.
10 + 10 is 20.
```

В дополнение к уникальным идентификаторам, при выполнении этого приложения вы заметите, что сообщение `Doing more work in Main()!` выводится практически мгновенно, в то время как вторичный поток продолжает свою работу.

Синхронизация вызывающего потока

Если внимательно проанализировать текущую реализацию `Main()`, то можно понять, что промежуток времени между вызовами `BeginInvoke()` и `EndInvoke()` однозначно

меньше 5 секунд. Следовательно, как только сообщение `Doing more work in Main()` ! выводится на консоль, вызывающий поток блокируется и ожидает завершения вторичного потока, чтобы получить результат вызова метода `Add()`. Таким образом, на самом деле производится еще один *синхронный* вызов.

```
static void Main(string[] args)
{
    ...
    BinaryOp b = new BinaryOp(Add);
    // После следующего оператора вызывающий поток блокируется,
    // пока не будет завершен BeginInvoke().
    IAsyncResult iftAR = b.BeginInvoke(10, 10, null, null);
    // Этот вызов займет намного меньше 5 секунд!
    Console.WriteLine("Doing more work in Main()!");
    // Теперь снова происходит ожидание завершения другого потока!
    int answer = b.EndInvoke(iftAR);
    ...
}
```

Очевидно, что асинхронные делегаты утратили бы свою привлекательность, если бы вызывающий поток при определенных обстоятельствах мог блокироваться. Чтобы позволить вызывающему потоку выяснять, завершил ли свою работу асинхронно вызванный метод, в интерфейсе `IAsyncResult` предусмотрено свойство `IsCompleted`. С его помощью вызывающий поток может определять, действительно ли асинхронный вызов был завершен, прежде чем обращаться к `EndInvoke()`.

Если метод еще не завершился, свойство `IsCompleted` возвращает `false`, и вызывающий поток может продолжить заниматься своей работой. Когда `IsCompleted` возвращает `true`, вызывающий поток может получить результат в "наименее блокирующей манере". Рассмотрим следующую модификацию метода `Main()`:

```
static void Main(string[] args)
{
    ...
    BinaryOp b = new BinaryOp(Add);
    IAsyncResult iftAR = b.BeginInvoke(10, 10, null, null);
    //Это сообщение продолжит выводиться до тех пор, пока не будет завершен метод Add().
    while(!iftAR.IsCompleted)
    {
        Console.WriteLine("Doing more work in Main()!");
        Thread.Sleep(1000);
    }
    // Теперь известно, что метод Add() завершен.
    int answer = b.EndInvoke(iftAR);
    ...
}
```

Здесь запускается цикл, который продолжает выполнять оператор `Console.WriteLine()` до тех пор, пока не завершится вторичный поток. Когда это произойдет, можно получить результат выполнения метода `Add()`, уже зная точно, что он закончил работу. Вызов `Thread.Sleep(1000)` не обязателен для корректного функционирования этого конкретного приложения; однако, заставляя первичный поток ожидать приблизительно секунду на каждой итерации, мы предотвращаем вывод на консоль слишком большого количества одного и того же сообщения. Ниже показан вывод (он может слегка отличаться, в зависимости от скорости машины и времени запуска потока):

```

***** Async Delegate Invocation *****
Main() invoked on thread 1.
Doing more work in Main()!
Add() invoked on thread 3.
Doing more work in Main()!
Doing more work in Main()!
Doing more work in Main()!
Doing more work in Main()!
Doing more work in Main()!
Doing more work in Main()!
10 + 10 is 20.

```

В дополнение к свойству `IsCompleted`, интерфейс `IAsyncResult` предлагает свойство `AsyncWaitHandle`, предназначенное для реализации более гибкой логики ожидания. Это свойство возвращает экземпляр типа `WaitHandle`, который открывает метод по имени `WaitOne()`. Преимущество `WaitHandle.WaitOne()` в том, что можно задавать максимальное время ожидания. Если это время истекает, метод `WaitOne()` возвращает `false`. Взгляните на следующий измененный цикл `while`, в котором больше не используется вызов `Thread.Sleep()`:

```

while (!itfAR.AsyncWaitHandle.WaitOne(1000, true))
{
    Console.WriteLine("Doing more work in Main()!");
}

```

Хотя эти свойства интерфейса `IAsyncResult` и предоставляют способ синхронизации вызывающего потока, все же это не самый эффективный подход. Во многих отношениях свойство `IsCompleted` подобно надоедливому менеджеру, который постоянно спрашивает: “Вы уже сделали это?”. К счастью, делегаты предлагают несколько дополнительных (и более элегантных) приемов получения результата из метода, который был вызван асинхронным образом.

Исходный код. Проект `AsyncDelegate` доступен в подкаталоге `Chapter 19`.

Роль делегата `AsyncCallback`

Вместо опроса делегата с целью определения, завершился ли асинхронно вызванный метод, было бы более эффективно заставить вторичный поток информировать вызывающий поток о завершении выполнения задачи. Чтобы включить такое поведение, понадобится передать методу `BeginInvoke()` в качестве параметра экземпляр делегата `System.AsyncCallback`; до сих пор этот параметр был равен `null`. Когда передается объект `AsyncCallback`, делегат будет автоматически вызывать указанный метод по завершении асинхронного вызова.

На заметку! Метод обратного вызова будет вызван во вторичном потоке, а не в первичном. Это имеет важные последствия для потоков с графическим пользовательским интерфейсом (WPF или Windows Forms), поскольку элементы управления привязаны к потоку, который их создал, и могут обрабатываться только им. При рассмотрении библиотеки TPL и новых ключевых слов `async` и `await` языка C# в .NET 4.5 далее в этой главе будут представлены некоторые примеры работы потоков из графического пользовательского интерфейса.

Как и любой делегат, `AsyncCallback` может вызывать только методы, соответствующие определенному шаблону, которым в данном случае является метод, принимающий единственный параметр `IAAsyncResult` и ничего не возвращающий:

```

// Целевые методы AsyncCallback должны соответствовать следующему шаблону.
void MyAsyncCallbackMethod(IAAsyncResult itfAR)

```

Предположим, что имеется другое консольное приложение (`AsyncCallbackDelegate`), использующее делегат `BinaryOp`. Однако на этот раз мы не будем опрашивать делегата, чтобы выяснить, завершился ли метод `Add()`. Вместо этого мы определим статический метод по имени `AddComplete()` для получения уведомления о том, что асинхронный вызов завершен. Также в этом примере используется булевское статическое поле уровня класса, которое служит для удержания в активном состоянии первичного потока `Main()`, пока не завершится вторичный поток.

На заметку! Использование булевской переменной в данном примере, строго говоря, не является безопасным в отношении потоков, поскольку к его значению имеют доступ два разных потока. В рассматриваемом примере это допустимо; тем не менее, запомните в качестве хорошего эмпирического правила: вы должны обеспечивать блокировку данных, разделяемых между несколькими потоками. Далее в главе будет показано, как это делается.

```
namespace AsyncCallbackDelegate
{
    public delegate int BinaryOp(int x, int y);
    class Program
    {
        private static bool isDone = false;
        static void Main(string[] args)
        {
            Console.WriteLine("***** AsyncCallbackDelegate Example *****");
            Console.WriteLine("Main() invoked on thread {0}.",
                Thread.CurrentThread.ManagedThreadId);

            BinaryOp b = new BinaryOp(Add);
            IAsyncResult iftAR = b.BeginInvoke(10, 10,
                new AsyncCallback(AddComplete), null);

            // Предположим, что здесь выполняется какая-то другая работа...
            while (!isDone)
            {
                Thread.Sleep(1000);
                Console.WriteLine("Working....");
            }
            Console.ReadLine();
        }

        static int Add(int x, int y)
        {
            Console.WriteLine("Add() invoked on thread {0}.",
                Thread.CurrentThread.ManagedThreadId);
            Thread.Sleep(5000);
            return x + y;
        }

        static void AddComplete(IAsyncResult iftAR)
        {
            Console.WriteLine("AddComplete() invoked on thread {0}.",
                Thread.CurrentThread.ManagedThreadId);
            Console.WriteLine("Your addition is complete");
            isDone = true;
        }
    }
}
```

И снова статический метод `AddComplete()` будет вызван делегатом `AsyncCallback` по завершении метода `Add()`. Запустив эту программу, можно убедиться, что именно вторичный поток вызывает `AddComplete()`:

```
***** AsyncCallbackDelegate Example *****
Main() invoked on thread 1.
Add() invoked on thread 3.
Working....
Working....
Working....
Working....
Working....
AddComplete() invoked on thread 3.
Your addition is complete
```

Как и в других примерах настоящей главы, вывод может несколько отличаться. В действительности может появиться только одно финальное сообщение `Working...` после завершения `AddComplete()`. Это просто побочный эффект односекундной задержки в `Main()`.

Роль класса `AsyncResult`

В настоящий момент метод `AddComplete()` не выводит на консоль действительный результат операции (сложения двух чисел). Причина в том, что целевой метод делегата `AsyncCallback` (`AsyncResult()` в этом примере) не имеет доступа к исходному делегату `BinaryOp`, созданному в контексте `Main()`, и потому вызывать `EndInvoke()` из `AddComplete()` нельзя!

Хотя можно было бы просто объявить переменную `BinaryOp` как статический член класса, чтобы позволить обоим методам обращаться к одному и тому же объекту, более элегантное решение предусматривает применение входного параметра `IAAsyncResult`.

Входной параметр `IAAsyncResult`, передаваемый целевому методу делегата `AsyncCallback` — это на самом деле экземпляр класса `AsyncResult` (обратите внимание на отсутствие префикса `I`), определенного в пространстве имен `System.Runtime.Remoting.Messaging`. Свойство `AsyncDelegate` возвращает ссылку на исходный асинхронный делегат, который был создан где-то в другом месте.

Таким образом, если нужно получить ссылку на объект делегата `BinaryOp`, размещенный внутри `Main()`, просто приведите экземпляр `System.Object`, возвращенный свойством `AsyncDelegate`, к типу `BinaryOp`. В этот момент можно запустить `EndInvoke()`, как и ожидалось.

```
// Не забудьте импортировать пространство имен System.Runtime.Remoting.Messaging!
static void AddComplete(IAAsyncResult itfAR)
{
    Console.WriteLine("AddComplete() invoked on thread {0}.",
        Thread.CurrentThread.ManagedThreadId);
    Console.WriteLine("Your addition is complete");

    // Теперь получить результат.
    AsyncResult ar = (AsyncResult)itfAR;
    BinaryOp b = (BinaryOp)ar.AsyncDelegate;
    Console.WriteLine("10 + 10 is {0}.", b.EndInvoke(itfAR));
    isDone = true;
}
```

Передача и получение специальных данных состояния

Финальным аспектом асинхронных делегатов, который должен быть учтен, является последний аргумент метода `BeginInvoke()` (который до сих пор был `null`). Этот параметр позволяет передавать дополнительную информацию о состоянии методу обратного вызова из первичного потока. Поскольку упомянутый аргумент прототипирован как `System.Object`, в нем можно передавать данные любого типа при условии, что методу обратного вызова известно, чего ожидать. В целях демонстрации предположим, что первичный поток желает передать специальное текстовое сообщение методу `AddComplete()`, примерно так:

```
static void Main(string[] args)
{
    ...
    IAsyncResult iftAR = b.BeginInvoke(10, 10,
        new AsyncCallback(AddComplete),
        "Main() thanks you for adding these numbers.");
    ...
}
```

Для получения этих данных в контексте метода `AddComplete()` используется свойство `AsyncState` входного параметра `IAsyncResult`. Обратите внимание, что здесь требуется явное приведение, поэтому первичный и вторичный потоки должны согласоваться между собой тип, возвращаемый `AsyncState`.

```
static void AddComplete(IAsyncResult iftAR)
{
    ...
    // Получить информационный объект и привести его к string.
    string msg = (string)iftAR.AsyncState;
    Console.WriteLine(msg);
    isDone = true;
}
```

Ниже показан вывод последней модификации примера:

```
***** AsyncCallbackDelegate Example *****
Main() invoked on thread 1.
Add() invoked on thread 3.
Working....
Working....
Working....
Working....
Working....
AddComplete() invoked on thread 3.
Your addition is complete
10 + 10 is 20.
Main() thanks you for adding these numbers.
```

Теперь, когда известно, как применять делегат `.NET` для автоматического запуска вторичного потока выполнения для обработки асинхронного вызова метода, давайте обратимся непосредственно к взаимодействию с потоками, используя пространство имен `System.Threading`. Вспомните, что это пространство имен было первоначальным API-интерфейсом для реализации многопоточности `.NET`, поставляемым еще в версии 1.0.

Пространство имен System.Threading

Пространство имен System.Threading в .NET предлагает несколько типов, которые предоставляют возможность непосредственного конструирования многопоточных приложений. В дополнение к типам, позволяющим взаимодействовать с определенным потоком CLR, в этом пространстве имен определены типы, которые открывают доступ к пулу потоков, обслуживаемому CLR, простому (не связанному с графическим пользовательским интерфейсом) классу Timer и многочисленным типам, используемым для предоставления синхронизированного доступа к разделяемым ресурсам. В табл. 19.1 перечислены некоторые важные члены этого пространства имен (за полной информацией обращайтесь в документацию .NET Framework 4.5 SDK).

Таблица 19.1. Ключевые типы пространства имен System.Threading

Тип	Назначение
Interlocked	Этот тип предоставляет атомарные операции для переменных, разделяемых между несколькими потоками
Monitor	Этот тип обеспечивает синхронизацию потоковых объектов, используя блокировки и ожидания/сигналы. Ключевое слово lock языка C# применяет “за кулисами” объект Monitor
Mutex	Этот примитив синхронизации может использоваться для синхронизации между границами доменов приложений
ParameterizedThreadStart	Этот делегат позволяет потоку вызывать методы, принимающие произвольное количество аргументов
Semaphore	Этот тип позволяет ограничить количество потоков, которые могут иметь доступ к ресурсу или к определенному типу ресурсов одновременно
Thread	Этот тип представляет поток, выполняемый в CLR-среде. Используя этот тип, можно порождать дополнительные потоки в исходном домене приложения
ThreadPool	Этот тип позволяет взаимодействовать с поддерживаемым CLR пулом потоков внутри заданного процесса
ThreadPriority	Это перечисление представляет уровень приоритета потока (Highest, Normal и т.д.)
ThreadStart	Этот делегат позволяет указать метод для вызова в заданном потоке. В отличие от делегата ParametrizedThreadStart, целевые методы ThreadStart всегда должны иметь один и тот же прототип
ThreadState	Это перечисление задает допустимые состояния потока (Running, Aborted и т.д.)
Timer	Этот тип предоставляет механизм выполнения метода через указанные интервалы
TimerCallback	Этот тип делегата используется в сочетании с типами Timer

Класс System.Threading.Thread

Класс Thread является самым элементарным из всех типов в пространстве имен System.Threading. Этот класс представляет объектно-ориентированную оболочку вокруг заданного пути выполнения внутри определенного домена приложения. Этот тип

также определяет набор методов (как статических, так и уровня экземпляра), которые позволяют создавать новые потоки внутри текущего домена приложения, а также приостанавливать, останавливать и уничтожать определенный поток. Список основных статических членов приведен в табл. 19.2.

Таблица 19.2. Основные статические члены типа Thread

Статический член	Назначение
CurrentContext	Это свойство, предназначенное только для чтения, возвращает контекст, в котором в данный момент выполняется поток
CurrentThread	Это свойство, предназначенное только для чтения, возвращает ссылку на текущий выполняемый поток
GetDomain() GetDomainID()	Этот метод возвращает ссылку на текущий домен приложения или идентификатор домена, в котором выполняется текущий поток
Sleep()	Этот метод приостанавливает текущий поток на указанное время

Класс Thread также поддерживает ряд методов уровня экземпляра, часть из которых описана в табл. 19.3.

Таблица 19.3. Некоторые члены уровня экземпляра типа Thread

Член уровня экземпляра	Назначение
IsAlive	Возвращает булевское значение, указывающее на то, запущен ли поток (и пока еще не прерван и не отменен)
IsBackground	Получает или устанавливает значение, указывающее, является ли данный поток фоновым (более подробно это объясняется далее)
Name	Позволяет установить дружественное текстовое имя потока
Priority	Получает или устанавливает приоритет потока, который может принимать значение из перечисления ThreadPriority
ThreadState	Получает состояние данного потока, которое может принимать значение из перечисления ThreadState
Abort()	Указывает CLR-среде на необходимость прекращения потока, как только это будет возможно
Interrupt()	Прерывает (например, приостанавливает) текущий поток на подходящий период ожидания
Join()	Блокирует вызывающий поток до тех пор, пока указанный поток (тот, на котором вызван метод Join()) не завершится
Resume()	Возобновляет ранее приостановленный поток
Start()	Указывает CLR-среде на необходимость запуска потока, как только это будет возможно
Suspend()	Приостанавливает поток. Если поток уже приостановлен, вызов Suspend() не дает никакого эффекта

На заметку! Прерывание или приостановка активного потока обычно считается плохой идеей. В этом случае есть шанс (хотя и небольшой), что поток может допустить "утечку" своей рабочей нагрузки.

Получение статистики о текущем потоке выполнения

Вспомните, что точка входа исполняемой сборки (т.е. метод `Main()`) запускается в первичном потоке выполнения. Чтобы проиллюстрировать базовое применение типа `Thread`, предположим, что имеется новое консольное приложение по имени `ThreadStats`. Как известно, статическое свойство `Thread.CurrentThread` извлекает объект `Thread`, представляющий текущий выполняющийся поток. После получения текущего потока можно вывести разнообразную статистику о нем, как показано ниже:

```
// Не забудьте импортировать пространство имен System.Threading.
static void Main(string[] args)
{
    Console.WriteLine("***** Primary Thread stats *****\n");

    // Получить имя текущего потока.
    Thread primaryThread = Thread.CurrentThread;
    primaryThread.Name = "ThePrimaryThread";

    // Показать детали включающего домена приложений и контекста.
    Console.WriteLine("Name of current AppDomain: {0}",
        Thread.GetDomain().FriendlyName);
    Console.WriteLine("ID of current Context: {0}",
        Thread.CurrentContext.ContextID);

    // Вывести некоторую статистику о текущем потоке.
    Console.WriteLine("Thread Name: {0}",
        primaryThread.Name); // имя потока
    Console.WriteLine("Has thread started?: {0}",
        primaryThread.IsAlive); // запущен ли поток?
    Console.WriteLine("Priority Level: {0}",
        primaryThread.Priority); // приоритет потока
    Console.WriteLine("Thread State: {0}",
        primaryThread.ThreadState); // состояние потока
    Console.ReadLine();
}
```

Ниже показан вывод:

```
***** Primary Thread stats *****
Name of current AppDomain: ThreadStats.exe
ID of current Context: 0
Thread Name: ThePrimaryThread
Has thread started?: True
Priority Level: Normal
Thread State: Running
```

Свойство Name

Хотя этот код более или менее очевиден, обратите внимание, что класс `Thread` поддерживает свойство по имени `Name`. Если не установить его значение явно, то `Name` будет возвращать пустую строку. Присваивание дружественного имени конкретному объекту `Thread` может значительно упростить отладку. Во время сеанса отладки в Visual Studio можно открыть окно `Threads` (Потоки), выбрав пункт меню `Debug⇒Windows⇒Threads` (Отладка⇒Окна⇒Потоки). Как показано на рис. 19.1, это окно позволяет быстро идентифицировать поток, который нужно диагностировать.

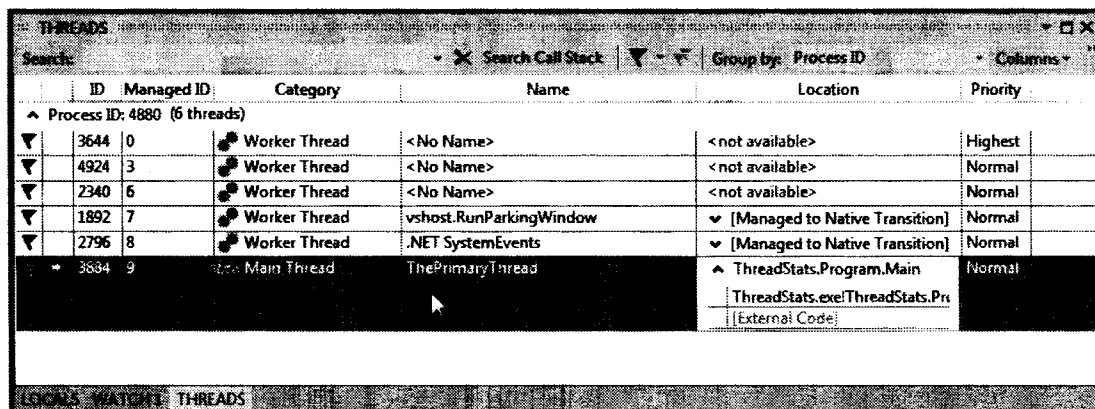


Рис. 19.1. Отладка потока в Visual Studio

Свойство Priority

Далее обратите внимание, что в типе Thread определено свойство по имени Priority. По умолчанию все потоки имеют уровень приоритета Normal. Тем не менее, в любой момент жизненного цикла потока это можно изменить, используя свойство Thread.Priority и связанное с ним перечисление System.Threading.ThreadPriority:

```
public enum ThreadPriority
{
    Lowest,
    BelowNormal,
    Normal, // Стандартное значение.
    AboveNormal,
    Highest
}
```

При установке для уровня приоритета потока значения, отличного от стандартного (ThreadPriority.Normal), помните об отсутствии прямого контроля над тем, когда планировщик потоков будет переключать потоки между собой. В действительности уровень приоритета потока предоставляет CLR-среде подсказку относительно важности действия потока. Таким образом, поток с уровнем приоритета ThreadPriority.Highest не обязательно гарантированно получит наивысший приоритет.

Опять-таки, если планировщик потоков занят решением определенной задачи (например, синхронизацией объекта, переключением потоков либо их перемещением), то уровень приоритета, скорее всего, будет соответствующим образом изменен. Однако, как бы то ни было, среда CLR прочитает эти значения и проинструктирует планировщик потоков о том, как наилучшим образом выделять кванты времени. Потоки с одинаковыми уровнями приоритета должны получать один и тот же объем времени на выполнение своей работы.

В большинстве случаев редко требуется (если вообще требуется) напрямую изменять уровень приоритета потока. Теоретически можно так повысить уровень приоритета набора потоков, что тем самым вообще предотвратить выполнение низкоприоритетных потоков с их запрошенными уровнями (поэтому соблюдайте осторожность).

Ручное создание вторичных потоков

Когда нужно программно создать дополнительные потоки для выполнения некоторой единицы работы, при использовании типов из пространства имен `System.Threading` следуйте описанному ниже строго регламентированному процессу.

1. Создайте метод, который будет служить точкой входа для нового потока.
2. Создайте новый делегат `ParametrizedThreadStart` (или `ThreadStart`), передав конструктору адрес метода, который был определен на шаге 1.
3. Создайте объект `Thread`, передав конструктору в качестве аргумента делегат `ParametrizedThreadStart/ThreadStart`.
4. Установите начальные характеристики потока (имя, приоритет и т.п.).
5. Вызовите метод `Thread.Start()`. Это приведет к запуску потока для метода, на который ссылается делегат, созданный на шаге 2, при первой же возможности.

Согласно шагу 2, для указания на метод, который будет выполняться во вторичном потоке, можно использовать два разных типа делегата. Делегат `ThreadStart` может указывать на любой метод, который не принимает аргументов и ничего не возвращает. Этот делегат может быть полезен, когда метод предназначен просто для запуска в фоновом режиме без какого-либо дальнейшего взаимодействия с ним.

Очевидное ограничение `ThreadStart` связано с невозможностью передавать ему параметры для обработки. Тем не менее, тип делегата `ParametrizedThreadStart` позволяет передать единственный параметр типа `System.Object`. Учитывая, что с помощью `System.Object` представляется все, что угодно, можно передавать любое количество параметров через специальный класс или структуру. Однако имейте в виду, что делегат `ParametrizedThreadStart` может указывать только на методы, возвращающие `void`.

Работа с делегатом `ThreadStart`

Чтобы проиллюстрировать процесс построения многопоточного приложения (а также пользу от него), предположим, что имеется консольное приложение (`SimpleMultiThreadApp`), которое позволяет конечному пользователю выбирать, будет приложение выполнять свою работу в единственном первичном потоке либо же распределит рабочую нагрузку на два отдельных потока выполнения.

После импортирования пространства имен `System.Threading` следующий шаг заключается в определении метода для выполнения работы (возможного) вторичного потока. Чтобы сосредоточиться на механизме построения многопоточных программ, этот метод будет просто выводить на консоль последовательность чисел, делая паузу примерно в 2 секунды на каждом шаге. Ниже показано полное определение класса `Printer`:

```
public class Printer
{
    public void PrintNumbers()
    {
        // Вывести информацию о потоке.
        Console.WriteLine("-> {0} is executing PrintNumbers()",
            Thread.CurrentThread.Name);
        // Вывести числа.
        Console.Write("Your numbers: ");
        for(int i = 0; i < 10; i++)
        {
            Console.Write("{0}, ", i);
            Thread.Sleep(2000);
        }
    }
}
```

```

        Console.WriteLine();
    }
}

```

Внутри `Main()` сначала пользователю предлагается решить, сколько потоков применять для выполнения работы приложения: один или два. Если пользователь запрашивает один поток, нужно просто вызвать метод `PrintNumbers()` внутри первичного потока. Если же пользователь отдает предпочтение двум потокам, необходимо создать делегата `ThreadStart`, указывающий на `PrintNumbers()`, передать объект делегата конструктору нового объекта `Thread` и вызвать метод `Start()`, информируя CLR-среду о том, что этот поток готов к обработке.

Первым делом, установим ссылку на сборку `System.Windows.Forms.dll` (и импортируем пространство имен `System.Windows.Forms`) и отобразим сообщение в `Main()`, используя `MessageBox.Show()` (причина такого решения станет понятной после запуска программы). Вот полная реализация `Main()`:

```

static void Main(string[] args)
{
    Console.WriteLine("***** The Amazing Thread App *****\n");
    Console.Write("Do you want [1] or [2] threads? ");
    string threadCount = Console.ReadLine();

    // Назначить имя текущему потоку.
    Thread primaryThread = Thread.CurrentThread;
    primaryThread.Name = "Primary";

    // Вывести информацию о потоке.
    Console.WriteLine("-> {0} is executing Main()",
        Thread.CurrentThread.Name);

    // Создать рабочий класс.
    Printer p = new Printer();
    switch(threadCount)
    {
        case "2":
            // Создать поток.
            Thread backgroundThread =
                new Thread(new ThreadStart(p.PrintNumbers));
            backgroundThread.Name = "Secondary";
            backgroundThread.Start();
            break;
        case "1":
            p.PrintNumbers();
            break;
        default:
            Console.WriteLine("I don't know what you want...you get 1 thread.");
            goto case "1"; // Для всех остальных вариантов ввода принимается один поток.
    }

    // Выполнить некоторую дополнительную работу.
    MessageBox.Show("I'm busy!", "Work on main thread...");
    Console.ReadLine();
}

```

Если теперь запустить эту программу с одним потоком, обнаружится, что финальное окно сообщения не отображает сообщения, пока вся последовательность чисел не будет выведена на консоль. Поскольку после вывода каждого числа установлена пауза примерно в 2 секунды, это создаст не слишком приятное впечатление у пользователя. Однако в случае выбора двух потоков окно сообщения отображается немедленно, поскольку для вывода чисел на консоль выделен отдельный объект `Thread` (рис. 19.2).

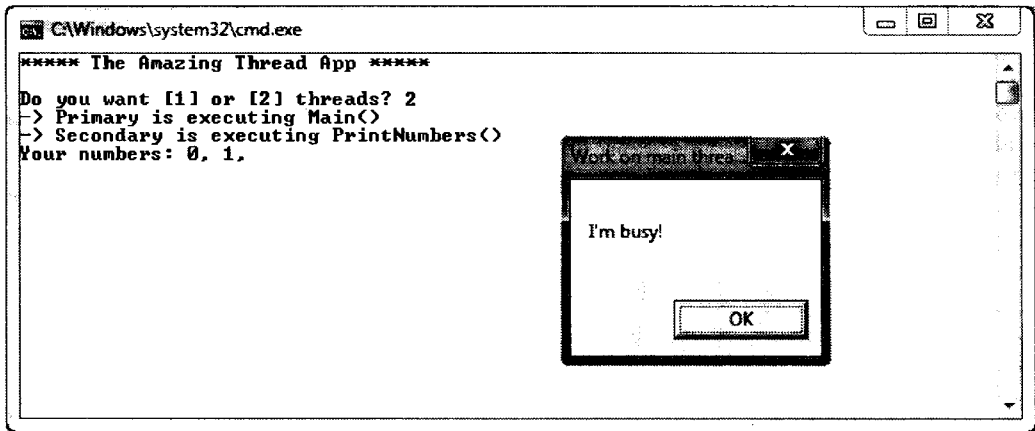


Рис. 19.2. Многопоточные приложения позволяют создавать более отзывчивые пользовательские интерфейсы

Исходный код. Проект SimpleMultiThreadApp доступен в подкаталоге Chapter 19.

Работа с делегатом ParametrizedThreadStart

Вспомните, что делегат `ThreadStart` может указывать только на методы, возвращающие `void` и не принимающие аргументов. В некоторых случаях это подходит, но если нужно передать данные методу, выполняющемуся во вторичном потоке, то придется использовать тип делегата `ParametrizedThreadStart`. В целях иллюстрации давайте воспроизведем логику проекта `AsyncCallbackDelegate`, разработанного ранее в настоящей главе, но на этот раз применим тип делегата `ParameterizedThreadStart`.

Для начала создадим новое консольное приложение по имени `AddWithThreads` и импортируем пространство имен `System.Threading`. Теперь, учитывая, что `ParametrizedThreadStart` может указывать на любой метод, принимающий параметр `System.Object`, построим специальный тип, который содержит числа, предназначенные для сложения:

```
class AddParams
{
    public int a, b;

    public AddParams(int numb1, int numb2)
    {
        a = numb1;
        b = numb2;
    }
}
```

Затем создадим в классе `Program` статический метод, который принимает параметр `AddParams` и выводит на консоль сумму двух чисел:

```
static void Add(object data)
{
    if (data is AddParams)
    {
        Console.WriteLine("ID of thread in Add(): {0}",
            Thread.CurrentThread.ManagedThreadId);

        AddParams ap = (AddParams) data;
```

```

        Console.WriteLine("{0} + {1} is {2}",
            ap.a, ap.b, ap.a + ap.b);
    }
}

```

Код внутри `Main()` достаточно очевиден. Просто вместо `ThreadStart` используется `ParameterizedThreadStart`:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Adding with Thread objects *****");
    Console.WriteLine("ID of thread in Main(): {0}",
        Thread.CurrentThread.ManagedThreadId);
    // Создать объект AddParams для передачи вторичному потоку.
    AddParams ap = new AddParams(10, 10);
    Thread t = new Thread(new ParameterizedThreadStart(Add));
    t.Start(ap);
    // Подождать, пока другой поток завершится.
    Thread.Sleep(5);
    Console.ReadLine();
}

```

Класс `AutoResetEvent`

В этих первых примерах для информирования первичного потока о необходимости подождать, пока вторичный поток не завершится, применялось несколько прямолинейных способов. Во время рассмотрения асинхронных делегатов в качестве переключателя использовалась простая булевская переменная. Однако это решение не является рекомендуемым, т.к. оба потока обращаются к одному и тому же элементу данных, что может привести к его повреждению. Более безопасной, хотя все еще нежелательной альтернативой может быть вызов `Thread.Sleep()` на фиксированный период времени. Проблема в том, что нет желания ожидать больше, чем необходимо.

Простой и безопасный к потокам способ заставить один поток ожидать, пока не завершится другой поток, предусматривает применение класса `AutoResetEvent`. В потоке, который должен ожидать (таком как метод `Main()`), создадим экземпляр `AutoResetEvent` и передадим его конструктору `false`, указав, что уведомления пока не было. В точке, где требуется ожидать, вызовем метод `WaitOne()`. Ниже приведен модифицированный класс `Program`, который делает все это, используя статическую переменную-член `AutoResetEvent`:

```

class Program
{
    private static AutoResetEvent waitHandle = new AutoResetEvent(false);
    static void Main(string[] args)
    {
        Console.WriteLine("***** Adding with Thread objects *****");
        Console.WriteLine("ID of thread in Main(): {0}",
            Thread.CurrentThread.ManagedThreadId);
        AddParams ap = new AddParams(10, 10);
        Thread t = new Thread(new ParameterizedThreadStart(Add));
        t.Start(ap);

        // Подождать, пока не поступит уведомление!
        waitHandle.WaitOne();
        Console.WriteLine("Other thread is done!");
        Console.ReadLine();
    }
    ...
}

```

Когда другой поток завершит свою работу, он вызовет метод `Set()` на том же экземпляре типа `AutoResetEvent`:

```
static void Add(object data)
{
    if (data is AddParams)
    {
        Console.WriteLine("ID of thread in Add(): {0}",
            Thread.CurrentThread.ManagedThreadId);

        AddParams ap = (AddParams)data;
        Console.WriteLine("{0} + {1} is {2}",
            ap.a, ap.b, ap.a + ap.b);

        // Сообщить другому потоку о завершении работы.
        waitHandle.Set();
    }
}
```

Исходный код. Проект `AddWithThreads` доступен в подкаталоге `Chapter 19`.

Потоки переднего плана и фоновые потоки

Теперь, когда вы знаете, как программно создавать новые потоки.выполнения с применением типов из пространства имен `System.Threading`, давайте формально определим разницу между потоками *переднего плана* и *фоновыми потоками*.

- *Потоки переднего плана* имеют возможность предохранять текущее приложение от завершения. Среда CLR не будет прекращать приложение (что означает, скажем, выгрузку текущего домена приложения) до тех пор, пока не будут завершены все фоновые потоки.
- *Фоновые потоки* (иногда называемые *потоками-демонами*) воспринимаются средой CLR как расширяемые пути выполнения, которые в любой момент времени могут быть проигнорированы (даже если они заняты выполнением некоторой части работы). Таким образом, если все потоки переднего плана завершены, то все фоновые потоки автоматически уничтожаются при выгрузке домена приложения.

Важно отметить, что потоки переднего плана и фоновые потоки — это *не* синонимы первичных и рабочих потоков. По умолчанию каждый поток, создаваемый через метод `Thread.Start()`, автоматически становится потоком переднего плана. Это означает, что домен приложения не выгрузится до тех пор, пока все потоки выполнения не завершат свои единицы работы. В большинстве случаев именно такое поведение и требуется.

Чтобы подтвердить сказанное, предположим, что необходимо вызвать `Printer.PrintNumbers()` во вторичном потоке, который должен вести себя как фоновый. Это означает, что метод, указываемый типом `Thread` (посредством делегата `ThreadStart` или `ParametrizedThreadStart`), должен обладать возможностью безопасного останова, как только все потоки переднего плана закончат свою работу. Конфигурирование такого потока сводится просто к установке свойства `IsBackground` в `true`, как показано ниже:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Background Threads *****\n");

    Printer p = new Printer();
    Thread bgroundThread =
        new Thread(new ThreadStart(p.PrintNumbers));
```

```
// Теперь это фоновый поток.
bgroundThread.IsBackground = true;
bgroundThread.Start();
}
```

Обратите внимание, что в этом методе `Main()` не производится вызов `Console.ReadLine()`, чтобы оставить окно консоли видимым, пока не будет нажата клавиша `<Enter>`. Таким образом, после запуска приложение прекращается немедленно, потому что объект `Thread` сконфигурирован как фоновый поток. Учитывая, что метод `Main()` инициирует создание первичного потока *переднего плана*, как только логика метода `Main()` завершится, домен приложения будет выгружен, прежде чем вторичный поток сможет закончить свою работу.

Однако если закомментировать строку, которая устанавливает в `true` свойство `IsBackground`, обнаружится, что все числа выводятся на консоль, т.к. все потоки переднего плана должны завершить свою работу перед тем, как домен приложения будет выгружен из размещающего процесса.

По большей части конфигурирование потока для выполнения в фоновом режиме может пригодиться, когда интересующий рабочий поток выполняет не критичную задачу, потребность в которой пропадает после завершения основной задачи программы. Например, можно было бы построить приложение, которое проверяет сервер электронной почты каждые несколько минут на предмет поступления новых писем, обновляет текущий прогноз погоды или решает какие-то другие не критичные задачи.

Проблемы параллелизма

При построении многопоточных приложений необходимо гарантировать, что любая часть разделяемых данных защищена от возможности изменения их значений множеством потоков. Учитывая, что все потоки в домене приложения имеют параллельный доступ к разделяемым данным приложения, представьте, что может произойти, если несколько потоков одновременно обратятся к одному и тому же элементу данных. Поскольку планировщик потоков случайным образом будет приостанавливать их работу, что если первый поток будет прерван до того, как завершит свою работу? А вот что: второй поток после этого прочитает нестабильные данные.

Чтобы проиллюстрировать проблему, связанную с параллелизмом, давайте построим еще один проект консольного приложения под названием `MultiThreadedPrinting`. В этом приложении опять будет использоваться созданный ранее класс `Printer`, но на этот раз метод `PrintNumbers()` приостановит текущий поток на случайно сгенерированный период времени.

```
public class Printer
{
    public void PrintNumbers()
    {
        ...
        for (int i = 0; i < 10; i++)
        {
            // Приостановить поток на случайный период времени.
            Random r = new Random();
            Thread.Sleep(1000 * r.Next(5));
            Console.Write("{0}, ", i);
        }
        Console.WriteLine();
    }
}
```


Метод `Main()` отвечает за создание массива из десяти (уникально именованных) объектов `Thread`, каждый из которых производит вызов одного и того же экземпляра объекта `Printer`:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("*****Synchronizing Threads *****\n");

        Printer p = new Printer();

        // Создать 10 потоков, которые указывают на один
        // и тот же метод того же самого объекта.
        Thread[] threads = new Thread[10];
        for (int i = 0; i < 10; i++)
        {
            threads[i] =
                new Thread(new ThreadStart(p.PrintNumbers));
            threads[i].Name = string.Format("Worker thread #{0}", i);
        }

        // Теперь запустить их все.
        foreach (Thread t in threads)
            t.Start();
        Console.ReadLine();
    }
}
```

Прежде чем посмотреть на тестовые запуски, давайте еще раз проясним проблему. Первичный поток внутри этого домена приложения начинает свое существование, порождая десять вторичных рабочих потоков. Каждый рабочий поток должен вызвать метод `PrintNumbers()` на том же самом экземпляре `Printer`. Учитывая, что никаких мер для блокировки разделяемых ресурсов этого объекта (консоли) не предпринималось, есть неплохие шансы, что текущий поток будет отключен, прежде чем метод `PrintNumbers()` сможет вывести полные результаты. Поскольку в точности не известно, когда это может произойти (и может ли вообще), будут получены непредсказуемые результаты. Например, вывод может выглядеть так:

```
*****Synchronizing Threads *****
-> Worker thread #1 is executing PrintNumbers()
Your numbers: -> Worker thread #0 is executing PrintNumbers()
-> Worker thread #2 is executing PrintNumbers()
Your numbers: -> Worker thread #3 is executing PrintNumbers()
Your numbers: -> Worker thread #4 is executing PrintNumbers()
Your numbers: -> Worker thread #6 is executing PrintNumbers()
Your numbers: -> Worker thread #7 is executing PrintNumbers()
Your numbers: -> Worker thread #8 is executing PrintNumbers()
Your numbers: -> Worker thread #9 is executing PrintNumbers()
Your numbers: Your numbers: -> Worker thread #5 is executing PrintNumbers()
Your numbers: 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3, 2, 1, 0, 0, 4, 3,
4, 1, 2, 4, 5, 5, 5, 6, 6, 6, 2, 7, 7, 7, 3, 4, 0, 8, 4, 5, 1, 5, 8, 8, 9,
2, 6, 1, 0, 9, 1,
6, 2, 7, 9,
2, 1, 7, 8, 3, 2, 3, 3, 9,
8, 4, 4, 5, 9,
4, 3, 5, 5, 6, 3, 6, 7, 4, 7, 6, 8, 7, 4, 8, 5, 5, 6, 6, 8, 7, 7, 9,
8, 9,
8, 9,
9,
9,
```

Запустите приложение еще несколько раз. Вот еще один вариант вывода:

```
*****Synchronizing Threads *****
-> Worker thread #0 is executing PrintNumbers()
-> Worker thread #1 is executing PrintNumbers()
-> Worker thread #2 is executing PrintNumbers()
Your numbers: -> Worker thread #4 is executing PrintNumbers()
Your numbers: -> Worker thread #5 is executing PrintNumbers()
Your numbers: Your numbers: -> Worker thread #6 is executing PrintNumbers()
Your numbers: -> Worker thread #7 is executing PrintNumbers()
Your numbers: Your numbers: -> Worker thread #8 is executing PrintNumbers()
Your numbers: -> Worker thread #9 is executing PrintNumbers()
Your numbers: -> Worker thread #3 is executing PrintNumbers()
Your numbers: 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4,
4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 7, 7, 7, 7, 7, 7,
7, 7, 7, 7, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 9,
9,
9,
9,
9,
9,
9,
9,
9,
9,
9,
```

На заметку! Если не удастся получить непредсказуемый вывод, увеличьте количество потоков с 10 до 100 (к примеру) или добавьте в код еще один вызов `Thread.Sleep()`. В конце концов, вы получите проблему, связанную с параллелизмом.

Должно быть совершенно ясно, что здесь присутствуют проблемы. Как только каждый поток требует от `Printer` вывода числовых данных, планировщик потоков благополучно меняет их местами в фоновом режиме. В результате получается несогласованный вывод. Необходим способ обеспечения в коде синхронизированного доступа к разделяемым ресурсам. Как и можно было предположить, пространство имен `System.Threading` предлагает множество типов, ориентированных на синхронизацию. В языке C# также предусмотрено специальное ключевое слово для синхронизации разделяемых данных в многопоточном приложении.

Синхронизация с использованием ключевого слова `lock` языка C#

Первый прием, который можно использовать для синхронизации доступа к разделяемым ресурсам, заключается в применении ключевого слова `lock` языка C#. Это ключевое слово позволяет определять контекст операторов, которые должны быть синхронизованными между потоками. В результате входящие потоки не могут прервать текущий поток, мешая ему завершить свою работу. Ключевое слово `lock` требует указания маркера (ссылки на объект), который должен быть получен потоком для входа в контекст блокировки. Чтобы попытаться заблокировать закрытый метод уровня экземпляра, нужно просто передать ссылку на текущий тип:

```
private void SomePrivateMethod()
{
    // Использовать текущий объект как маркер потока.
    lock(this)
    {
        // Весь код внутри этого контекста является безопасным к потокам.
    }
}
```

Однако если вы блокируете область кода внутри *открытого* члена, безопаснее (и вообще лучше) объявить закрытую переменную-член `object` для использования в качестве маркера блокировки:

```
public class Printer
{
    // Маркер блокировки.
    private object threadLock = new object();
    public void PrintNumbers()
    {
        // Использование маркера блокировки.
        lock (threadLock)
        {
            ...
        }
    }
}
```

В любом случае, если взглянуть на метод `PrintNumbers()`, то можно заметить, что разделяемый ресурс, за доступ к которому соперничают потоки — это окно консоли. Таким образом, если поместить все взаимодействие с типом `Console` в контекст `lock`, как показано ниже, будет получен метод, который позволит текущему потоку завершить свою задачу:

```
public void PrintNumbers()
{
    // Использовать в качестве маркера блокировки закрытый объект.
    lock (threadLock)
    {
        // Вывести информацию о потоке.
        Console.WriteLine("-> {0} is executing PrintNumbers()",
            Thread.CurrentThread.Name);

        // Вывести числа.
        Console.Write("Your numbers: ");
        for (int i = 0; i < 10; i++)
        {
            Random r = new Random();
            Thread.Sleep(1000 * r.Next(5));
            Console.Write("{0}, ", i);
        }
        Console.WriteLine();
    }
}
```

Как только поток входит в контекст `lock`, маркер блокировки (в данном случае — ссылка на текущий объект) станет недоступным другим потокам до тех пор, пока блокировка не будет освобождена после выхода из контекста `lock`. Таким образом, если поток А получил маркер блокировки, другие потоки не смогут войти ни в один из контекстов, использующих тот же самый маркер, до тех пор, пока поток А не освободит его.

На заметку! Если вы пытаетесь заблокировать код в статическом методе, просто объявите закрытую статическую переменную-член типа `object`, которая будет служить маркером блокировки.

Если теперь запустить приложение, можно увидеть, что каждый поток получил возможность выполнить свою работу до конца:

```

*****Synchronizing Threads *****
-> Worker thread #0 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #1 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #3 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #2 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #4 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #5 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, .
-> Worker thread #7 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #6 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #8 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #9 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,

```

Исходный код. Проект `MultiThreadedPrinting` доступен в подкаталоге `Chapter 19`.

Синхронизация с использованием типа `System.Threading.Monitor`

Оператор `lock` языка C# — это на самом деле сокращенная нотация для работы с классом `System.Threading.Monitor`. При обработке компилятором C# контекст `lock` преобразуется в следующую конструкцию (в чем легко убедиться с помощью утилиты `ldasm.exe`):

```

public void PrintNumbers()
{
    Monitor.Enter(threadLock);
    try
    {
        // Вывести информацию о потоке.
        Console.WriteLine("-> {0} is executing PrintNumbers()",
            Thread.CurrentThread.Name);

        // Вывести числа.
        Console.Write("Your numbers: ");
        for (int i = 0; i < 10; i++)
        {
            Random r = new Random();
            Thread.Sleep(1000 * r.Next(5));
            Console.Write("{0}, ", i);
        }
        Console.WriteLine();
    }
    finally
    {
        Monitor.Exit(threadLock);
    }
}

```

Для начала обратите внимание, что метод `Monitor.Enter()` является конечным получателем маркера потока, который указывается как аргумент ключевого слова `lock`. Весь код внутри контекста `lock` помещен в блок `try`. Соответствующая конструкция `finally` гарантирует освобождение маркера блокировки (через метод `Monitor.Exit()`) независимо от каких бы то ни было исключений времени выполнения. Модифицировав программу `MultiThreadShareData` для прямого использования типа `Monitor` (как только что было показано), вы обнаружите, что вывод идентичен.

С учетом того, что ключевое слово `lock`, похоже, требует меньше кода, чем явная работа с типом `System.Threading.Monitor`, может возникнуть вопрос о преимуществах прямого применения этого типа. Краткий ответ формулируется так: больший контроль. Используя тип `Monitor`, можно заставить активный поток ожидать в течение некоторого периода времени (с помощью статического метода `Monitor.Wait()`), информировать ожидающие потоки, когда текущий поток завершен (через статические методы `Monitor.Pulse()` и `Monitor.PulseAll()`), и т.д.

Как и можно было ожидать, в значительном числе случаев ключевого слова `lock` вполне достаточно. Если вас интересуют дополнительные члены класса `Monitor`, обращайтесь в документацию `.NET Framework 4.5 SDK`.

Синхронизация с использованием типа `System.Threading.Interlocked`

Не заглядывая в CIL-код, очень трудно поверить, что присваивание и простые арифметические операции не являются атомарными. По этой причине в пространстве имен `System.Threading` предоставляется тип, позволяющий оперировать одиночным элементом данных атомарно и с меньшими накладными расходами, чем тип `Monitor`. В классе `Interlocked` определены статические члены, ключевые из которых перечислены в табл. 19.4.

Таблица 19.4. Избранные статические члены типа `System.Threading.Interlocked`

Член	Назначение
<code>CompareExchange()</code>	Безопасно проверяет два значения на равенство и, если они равны, заменяет одно из значений
<code>Decrement()</code>	Безопасно уменьшает значение на 1
<code>Exchange()</code>	Безопасно меняет два значения местами
<code>Increment()</code>	Безопасно уменьшает значение на 1

Хотя это не видно сразу, процесс атомарного изменения одиночного значения довольно часто применяется в многопоточной среде. Предположим, что имеется метод по имени `AddOne()`, который инкрементирует целочисленную переменную-член по имени `intVal`. Вместо написания кода синхронизации наподобие следующего:

```
public void AddOne()
{
    lock(myLockToken)
    {
        intVal++;
    }
}
```

можно воспользоваться статическим методом `Interlocked.Increment()` и в результате упростить код. Этому методу нужно передать по ссылке переменную для увеличения.

Обратите внимание, что метод `Increment()` не только изменяет значение входного параметра, но также возвращает полученное новое значение:

```
public void AddOne()
{
    int newVal = Interlocked.Increment(ref intVal);
}
```

В дополнение к `Increment()` и `Decrement()` тип `Interlocked` позволяет атомарным образом присваивать числовые и объектные данные. Например, чтобы присвоить значение 83 переменной-члену, можно обойтись без явного оператора `lock` (или явной логики `Monitor`) и применить вместо этого метод `Interlock.Exchange()`:

```
public void SafeAssignment()
{
    Interlocked.Exchange(ref myInt, 83);
}
```

Наконец, если необходимо проверить два значения на равенство и заменить элемент сравнения в безопасной к потокам манере, можно воспользоваться методом `Interlocked.CompareExchange()`, как показано ниже:

```
public void CompareAndExchange()
{
    // Если значение i равно 83, заменить его 99.
    Interlocked.CompareExchange(ref i, 99, 83);
}
```

Синхронизация с использованием атрибута [Synchronization]

Последний из примитивов синхронизации, который мы здесь рассмотрим — это атрибут `[Synchronization]`, который является членом пространства имен `System.Runtime.Remoting.Contexts`. Этот атрибут уровня класса эффективно блокирует весь код членов экземпляра объекта, обеспечивая безопасность в отношении потоков. Когда среда CLR размещает объекты, снабженные атрибутами `[Synchronization]`, она помещает объект в контекст синхронизации. Как было показано в главе 17, объекты, которые не должны выходить за границы контекста, должны наследоваться от `ContextBoundObject`. Поэтому, чтобы сделать класс `Printer` безопасным к потокам (без явного написания кода внутри членов класса), необходимо модифицировать его следующим образом:

```
using System.Runtime.Remoting.Contexts;
...
// Все методы Printer теперь безопасны к потокам!
[Synchronization]
public class Printer : ContextBoundObject
{
    public void PrintNumbers()
    {
        ...
    }
}
```

В некоторых отношениях этот подход выглядит как “ленивый” способ написания безопасного к потокам кода, учитывая, что не приходится углубляться в детали относительно того, какие именно аспекты типа действительно манипулируют чувствительными к потокам данными. Однако главный недостаток этого подхода состоит в том, что даже если определенный метод не использует чувствительные к потокам данные, CLR-

среда будет *по-прежнему* блокировать вызовы этого метода. Очевидно, что это приведет к деградации общей функциональности типа, поэтому используйте такой прием с осторожностью.

Программирование с использованием обратных вызовов Timer

Многие приложения нуждаются в вызове специфического метода через регулярные интервалы времени. Например, в приложении может понадобиться отображать текущее время в панели состояния с помощью определенной вспомогательной функции. Или еще один пример: необходимо, чтобы приложение вызывало вспомогательную функцию периодически, выполняя некоторые не критичные фоновые задачи, такие как проверка поступления новых сообщений электронной почты. Для ситуаций вроде этой можно использовать тип `System.Threading.Timer` в сочетании с делегатом по имени `TimerCallback`.

В целях иллюстрации предположим, что имеется консольное приложение (`TimerApp`), которое выводит текущее время каждую секунду до тех пор, пока пользователь не нажмет клавишу `<Enter>` для завершения приложения. Первый очевидный шаг — написать метод, который будет вызываться типом `Timer` (не забудьте импортировать `System.Threading` в файл кода):

```
class Program
{
    static void PrintTime(object state)
    {
        Console.WriteLine("Time is: {0}",
            DateTime.Now.ToLongTimeString());
    }
    static void Main(string[] args)
    {
    }
}
```

Обратите внимание, что метод `PrintTime()` принимает единственный параметр типа `System.Object` и возвращает `void`. Это обязательно, поскольку делегат `TimerCallback` может вызывать только методы, соответствующие такой сигнатуре. Значение, переданное целевому методу делегата `TimerCallback`, может быть объектом любого типа (в случае примера с электронной почтой этот параметр может представлять имя сервера Microsoft Exchange для взаимодействия в течение процесса). Также обратите внимание, что поскольку этот параметр на самом деле является `System.Object`, в нем можно передавать несколько аргументов, используя `System.Array` или специальный класс/структуру.

Следующий шаг связан с конфигурированием экземпляра делегата `TimerCallback` и передачей его объекту `Timer`. В дополнение к настройке делегата `TimerCallback`, конструктор `Timer` позволяет указать необязательный информационный параметр для передачи целевому методу делегата (определенному как `System.Object`), интервал вызова метода и период времени ожидания (в миллисекундах), которое должно истечь перед первым вызовом. Ниже показан пример.

```
static void Main(string[] args)
{
    Console.WriteLine("***** Working with Timer type *****\n");
    // Создать делегат для типа Timer.
    TimerCallback timeCB = new TimerCallback(PrintTime);
```

```
// Установить параметры таймера.
Timer t = new Timer(
    timeCB,          // Объект делегата TimerCallback.
    null,            // Информация для передачи в вызванный метод
                    // (null – информация отсутствует).
    0,               // Период времени ожидания перед запуском (в миллисекундах).
    1000);           // Интервал времени между вызовами (в миллисекундах).
Console.WriteLine("Hit key to terminate...");
Console.ReadLine();
}
```

В этом случае метод `PrintTime()` вызывается приблизительно каждую секунду и не получает никакой дополнительной информации. Вот вывод примера:

```
***** Working with Timer type *****
```

```
Hit key to terminate...
Time is: 6:51:48 PM
Time is: 6:51:49 PM
Time is: 6:51:50 PM
Time is: 6:51:51 PM
Time is: 6:51:52 PM
Press any key to continue . . .
```

Чтобы передать целевому методу делегата какую-то информацию, замените значение `null` для второго параметра конструктора соответствующей информацией, например:

```
// Установить параметры таймера.
Timer t = new Timer(timeCB, "Hello From Main", 0, 1000);
```

Получить входные данные можно следующим образом:

```
static void PrintTime(object state)
{
    Console.WriteLine("Time is: {0}, Param is: {1}",
        DateTime.Now.ToLongTimeString(), state.ToString());
}
```

Исходный код. Проект `TimerApp` доступен в подкаталоге `Chapter 19`.

Пул потоков CLR

Следующей темой, связанной с потоками, которую мы рассмотрим в этой главе, является роль пула потоков CLR. При асинхронном вызове метода с использованием типов делегатов (через метод `BeginInvoke()`) среда CLR на самом деле не создает новый поток. В целях эффективности метод `BeginInvoke()` делегата задействует пул рабочих потоков, который поддерживается исполняющей средой. Для взаимодействия с этим пулом ожидающих потоков в пространстве имен `System.Threading` предусмотрен класс `ThreadPool`.

Чтобы запросить поток из пула для обработки вызова метода, можно воспользоваться методом `ThreadPool.QueueUserWorkItem()`. Этот метод перегружен, чтобы в дополнение к экземпляру делегата `WaitCallback` позволять указывать необязательный параметр `System.Object` для специальных данных состояния:

```
public static class ThreadPool
{
    ...
    public static bool QueueUserWorkItem(WaitCallback callBack);
    public static bool QueueUserWorkItem(WaitCallback callBack, object state);
}
```


Делегат `WaitCallback` может указывать на любой метод, который принимает в качестве единственного параметра экземпляр `System.Object` (представляющий необязательные данные состояния) и ничего не возвращает. Обратите внимание, что если при вызове `QueueUserWorkItem()` не указывается `System.Object`, среда CLR автоматически передает значение `null`. Чтобы проиллюстрировать работу методов очередей, работающих с пулом потоков CLR, рассмотрим еще раз программу, использующую тип `Printer`. Однако в этом случае массив объектов `Thread` вручную создаваться не будет; взамен методу `PrintNumbers()` будут присваиваться члены пула:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with the CLR Thread Pool *****\n");
        Console.WriteLine("Main thread started. ThreadID = {0}",
            Thread.CurrentThread.ManagedThreadId);

        Printer p = new Printer();
        WaitCallback workItem = new WaitCallback(PrintTheNumbers);

        // Поставить в очередь метод десять раз.
        for (int i = 0; i < 10; i++)
        {
            ThreadPool.QueueUserWorkItem(workItem, p);
        }
        Console.WriteLine("All tasks queued");
        Console.ReadLine();
    }
    static void PrintTheNumbers(object state)
    {
        Printer task = (Printer)state;
        task.PrintNumbers();
    }
}
```

Здесь может возникнуть вопрос: в чем же заключаются преимущества взаимодействия с поддерживаемым средой CLR пулом потоков по сравнению с явным созданием объектов `Thread`? Эти преимущества указаны ниже.

1. Пул потоков управляет потоками эффективным образом, сводя к минимуму количество создаваемых, запускаемых и останавливаемых потоков.
2. За счет использования пула потоков можно сосредоточиться на решении задачи, а не на потоковой инфраструктуре приложения.

Тем не менее, в некоторых случаях ручное управление потоками оказывается более предпочтительным.

- Когда нужны потоки переднего плана или должен быть установлен приоритет потока. Потоки из пула всегда являются фоновыми и имеют стандартный приоритет (`ThreadPriority.Normal`).
- Когда требуется поток с фиксированной идентичностью, чтобы его можно было прерывать, приостанавливать или находить по имени.

Исходный код. Проект `TimerPoolApp` доступен в подкаталоге `Chapter 19`.

На этом исследование пространства имен `System.Threading` завершено. Несомненно, понимание вопросов, рассмотренных в настоящей главе до сих пор (особенно в разделе, посвященном проблемам параллелизма), будет чрезвычайно ценным

при создании многопоточного приложения. А теперь, опираясь на этот фундамент, мы обратим внимание на несколько новых аспектов, связанных с потоками, которые доступны только в .NET 4.0 и последующих версиях. Для начала мы исследуем роль альтернативной потоковой модели, которая называется TPL.

Параллельное программирование с использованием TPL

К этому моменту в главе мы ознакомились с двумя технологиями программирования (применение асинхронных делегатов и взаимодействие с членами `System.Threading`), которые позволяют строить многопоточное программное обеспечение. Вспомните, что оба эти подхода будут работать в любой версии платформы .NET.

Начиная с версии .NET 4.0, в Microsoft ввели новый подход к разработке многопоточных приложений, предусматривающий использование библиотеки параллельного программирования, которая называется *TPL* (Task Parallel Library — библиотека параллельных задач). С помощью типов из `System.Threading.Tasks` можно строить мелкокомодульный масштабируемый параллельный код без необходимости напрямую иметь дело с потоками или пулом потоков.

Однако это не означает, что вы не будете применять типы из `System.Threading` при работе с TPL. В реальности эти два инструментальных набора могут вполне естественно работать вместе. Это особенно верно потому, что пространство имен `System.Threading` по-прежнему предоставляет большинство примитивов синхронизации, которые рассматривались ранее (`Monitor`, `Interlocked` и т.д.). В итоге вы, скорее всего, обнаружите, что работать с TPL предпочтительнее, чем с исходным пространством имен `System.Threading`, учитывая то, что те же самые задачи могут быть решены гораздо проще.

На заметку! Следует также отметить, что новые ключевые слова `async` and `await` языка C#, появившиеся в .NET 4.5, используют разнообразные члены пространства имен `System.Threading.Tasks`.

Пространство имен `System.Threading.Tasks`

Все вместе типы из пространства `System.Threading.Tasks` объединены общим названием *библиотека параллельных задач* (Task Parallel Library — TPL). Библиотека TPL позволяет автоматически распределять нагрузку приложений по доступным процессорам в динамическом режиме, используя пул потоков CLR. Библиотека TPL поддерживает разбиение работы на части, планирование потоков, управление состоянием и выполнение других низкоуровневых деталей. В конечном итоге появляется возможность максимизировать производительность приложений .NET, не имея дела со сложностями прямой работы с потоками (рис. 19.3).

Роль класса `Parallel`

Ключевым классом в TPL является `System.Threading.Tasks.Parallel`. Этот класс поддерживает набор методов, которые позволяют осуществлять итерацию по коллекции данных (точнее, по объекту, реализующему интерфейс `IEnumerable<T>`) в параллельном режиме. В документации .NET Framework 4.5 SDK можно заметить, что упомянутый класс поддерживает два основных статических метода, `Parallel.For()` и `Parallel.ForEach()`, каждый из которых имеет множество перегруженных версий.

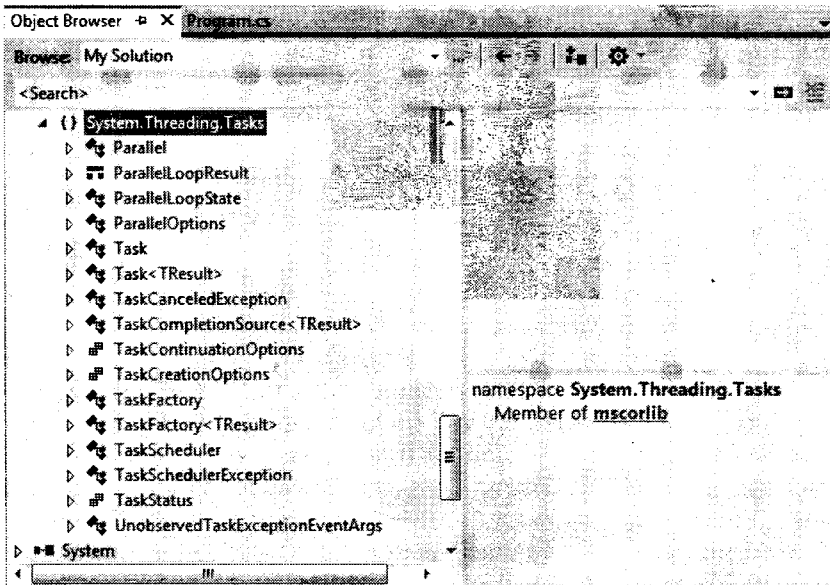


Рис. 19.3. Члены пространства имен System.Threading.Tasks

Эти методы позволяют создавать тело операторов кода, которое будет выполняться в параллельном режиме. Концептуально эти операторы представляют собой логику того же рода, которая была бы написана в нормальной циклической конструкции (с применением ключевых слов `for` и `foreach` языка C#). Преимущество заключается в том, что класс `Parallel` самостоятельно будет брать потоки из пула потоков (и управлять параллелизмом).

Оба метода требуют указания совместимого с `IEnumerable` или `IEnumerable<T>` контейнера, хранящего данные, которые нужно обработать в параллельном режиме. Контейнер может быть простым массивом, необобщенной коллекцией (такой как `ArrayList`), обобщенной коллекцией (наподобие `List<T>`) или результатом запроса LINQ.

Вдобавок с помощью делегатов `System.Func<T>` и `System.Action<T>` понадобится задать целевой метод, который будет вызываться для обработки данных. Делегат `Func<T>` уже встречался в главе 12, когда рассматривалась технология LINQ to Objects. Вспомните, что `Func<T>` представляет метод, который возвращает значение и принимает различное количество аргументов. Делегат `Action<T>` очень похож на `Func<T>` в том, что позволяет задавать метод, принимающий несколько параметров. Однако `Action<T>` указывает метод, который может возвращать только `void`.

Хотя можно было бы вызывать методы `Parallel.For()` и `Parallel.ForEach()` и передавать им строго типизированный объект делегата `Func<T>` или `Action<T>`, задача программирования упрощается за счет использования подходящих анонимных методов и лямбда-выражений языка C#.

Обеспечение параллелизма данных с помощью класса `Parallel`

Первое применение TPL связано с обеспечением параллелизма данных. Этим термином обозначается задача прохода по массиву или коллекции в параллельном режиме с использованием методов `Parallel.For()` и `Parallel.ForEach()`. Предположим, что нужно выполнить некоторые трудоемкие операции, связанные с файловым вводом-выводом. Например, требуется загрузить в память большое количество файлов `*.jpg`, повернуть содержащиеся в них изображения и сохранить модифицированные данные изображений в новом местоположении.

В документации .NET Framework 4.5 SDK предоставлен пример консольного приложения, решающего эту задачу; однако мы решим ее с применением графического пользовательского интерфейса, чтобы взглянуть на использование “анонимных делегатов”, позволяющих вторичным потокам обновлять первичный поток пользовательского интерфейса.

На заметку! При построении многопоточного приложения с графическим пользовательским интерфейсом вторичные потоки никогда не могут напрямую обращаться к элементам управления пользовательского интерфейса. Причина в том, что элементы управления (кнопки, текстовые поля, метки, индикаторы хода работ и т.п.) привязаны к потоку, который их создал. В следующем примере демонстрируется один из способов обеспечения для вторичных потоков возможности получить доступ к элементам пользовательского интерфейса в безопасной к потокам манере. При рассмотрении ключевых слов `async` и `await` языка C#, появившихся в .NET 4.5, будет представлен более простой подход.

В целях иллюстрации создадим приложение Windows Forms по имени `DataParallelism` `WithForEach` и с помощью `Solution Explorer` переименуем `Form1.cs` на `MainForm.cs`. После этого импортируем в главный файл кода следующие пространства имен:

```
// Удостоверьтесь в наличии этих пространств имен!
using System.Threading.Tasks;
using System.Threading;
using System.IO;
```

Графический пользовательский интерфейс этого приложения содержит многострочное текстовое поле `TextBox` и одну кнопку `Button` (по имени `btnProcessImages`). Текстовое поле предназначено для ввода данных во время выполнения работы в фоновом режиме, иллюстрируя неблокирующую природу параллельной задачи. В обработчике события `Click` элемента `Button` в конечном итоге будет использоваться библиотека TPL, а пока напомним блокирующий код, который приведен ниже.

На заметку! Вы должны обновить строку, передаваемую методу `Directory.GetFiles()`, чтобы она указывала конкретный путь на вашей машине к каталогу, содержащему какие-то файлы изображений. Для примера в коде используется каталог `C:\Users\Public\Pictures\Sample Pictures`.

```
public partial class MainForm : Form
{
    public MainForm()
    {
        InitializeComponent();
    }

    private void btnProcessImages_Click(object sender, EventArgs e)
    {
        ProcessFiles();
    }

    private void ProcessFiles()
    {
        // Загрузить все файлы *.jpg и создать новую папку для модифицированных данных.
        string[] files = Directory.GetFiles
            (@":C:\Users\Public\Pictures\Sample Pictures", "*.jpg",
            SearchOption.AllDirectories);
        string newDir = @":C:\ModifiedPictures";
        Directory.CreateDirectory(newDir);
```

```
// Обработать данные изображений в блокирующей манере.
foreach (string currentFile in files)
{
    string filename = Path.GetFileName(currentFile);
    using (Bitmap bitmap = new Bitmap(currentFile))
    {
        bitmap.RotateFlip(RotateFlipType.Rotate180FlipNone);
        bitmap.Save(Path.Combine(newDir, filename));

        // Вывести идентификатор потока, обрабатывающего текущее изображение.
        this.Text = string.Format("Processing {0} on thread {1}", filename,
            Thread.CurrentThread.ManagedThreadId);
    }
}
}
```

Обратите внимание, что метод `ProcessFiles()` выполнит поворот изображения в каждом файле `*.jpg` из указанного каталога, который в данный момент содержит всего 37 файлов (при необходимости укажите другой путь в вызове `Directory.GetFiles()`). В настоящее время вся работа происходит в первичном потоке исполняемой программы. Поэтому после щелчка на кнопке программа выглядит зависшей. Более того, заголовок окна также сообщит о том, что тот же самый первичный поток обрабатывает файл, поскольку в наличии имеется единственный поток выполнения.

Чтобы обеспечить обработку файлов на как можно большем числе процессоров, можно переписать текущий цикл `foreach`, воспользовавшись `Parallel.ForEach()`. Вспомните, что этот метод имеет множество перегрузок. Простейшая форма метода принимает совместимый с `IEnumerable<T>` объект, который содержит элементы, подлежащие обработке (например, строковый массив `files`), и делегат `Action<T>`, указывающий на метод, который будет выполнять необходимую работу.

Ниже показана требуемая модификация, в которой вместо литерального объекта делегата `Action<T>` используется лямбда-операция C#. Обратите внимание, что в коде закомментированы строки, отображающие идентификатор потока, который обрабатывает текущий файл изображения. Причина объясняется в следующем разделе.

```
// Обработать данные изображений в параллельном режиме!
Parallel.ForEach(files, currentFile =>
{
    string filename = Path.GetFileName(currentFile);
    using (Bitmap bitmap = new Bitmap(currentFile))
    {
        bitmap.RotateFlip(RotateFlipType.Rotate180FlipNone);
        bitmap.Save(Path.Combine(newDir, filename));

        // Этот оператор кода теперь вызывает проблемы! См. следующий раздел.
        // this.Text = string.Format("Processing {0} on thread {1}", filename,
        // Thread.CurrentThread.ManagedThreadId);
    }
});
```

Доступ к элементам пользовательского интерфейса во вторичных потоках

В показанном выше коде строки, которые обновляют заголовок главного окна значением идентификатора текущего выполняющегося потока, закомментированы. Как отмечалось ранее, элементы управления графического пользовательского интерфейса

привязаны к создавшему их потоку. Если вторичные потоки пытаются получить доступ к элементу управления, которые они напрямую не создавали, при отладке программного обеспечения возникают ошибки времени выполнения. С другой стороны, если *запустить* приложение (нажатием <Ctrl+F5>), первоначальный код может и не вызвать каких-либо проблем.

На заметку! Не лишним будет повториться: при отладке (по нажатию <F5>) многопоточного приложения IDE-среда Visual Studio часто способна перехватывать ошибки, когда вторичный поток обращается к элементу управления, созданному в первичном потоке. Тем не менее, нередко после запуска (с помощью <Ctrl+F5>) приложение может выглядеть выполняющимся корректно (или же ошибка может возникнуть сразу). Если только не будут предприняты меры предосторожности (описанные далее), приложение в подобных обстоятельствах может потенциально сгенерировать ошибку во время выполнения.

Один из подходов, которым можно воспользоваться для предоставления вторичным потокам доступа к элементам управления в безопасной к потокам манере, предусматривает применение другого приема — *анонимного делегата*. Родительский класс Control в API-интерфейсе Windows Forms определяет метод по имени `Invoke()`, который принимает на входе `System.Delegate`. Этот метод можно вызывать внутри кода, выполняющегося во вторичных потоках, обеспечивая безопасное к потокам обновление пользовательского интерфейса для заданного элемента управления. Хотя можно было бы написать весь требуемый код делегата напрямую, большинство разработчиков используют в качестве простой альтернативы анонимные делегаты. Ниже показана соответствующая модификация кода:

```
using (Bitmap bitmap = new Bitmap(currentFile))
{
    bitmap.RotateFlip(RotateFlipType.Rotate180FlipNone);
    bitmap.Save(Path.Combine(newDir, filename));

    // Это больше не работает!
    //this.Text = string.Format("Processing {0} on thread {1}", filename,
    // Thread.CurrentThread.ManagedThreadId);

    // Вызвать Invoke на объекте Form, чтобы позволить вторичным потокам
    // получать доступ к элементам управления в безопасной к потокам манере.
    this.Invoke((Action)delegate
    {
        this.Text = string.Format("Processing {0} on thread {1}", filename,
            Thread.CurrentThread.ManagedThreadId);
    });
}
```

На заметку! Метод `this.Invoke()` уникален для API-интерфейса Windows Forms. При построении приложения WPF для той же цели должен использоваться вызов `this.Dispatcher.Invoke()`.

Если теперь запустить программу, библиотека TPL распределит рабочую нагрузку по множеству потоков, взятых из пула, используя столько процессоров, сколько возможно. Однако в заголовке окна не будут отображаться имена уникальных потоков, а при вводе в текстовой области ничего не будет видно, пока не обработаются все файлы изображений! Причина в том, что первичный поток пользовательского интерфейса все равно остается заблокированным, ожидая, пока все прочие потоки завершат свою работу.

Класс Task

Класс Task позволяет легко вызывать метод во вторичном потоке и может применяться в качестве простой альтернативы работе с асинхронными делегатами. Изменим обработчик события Click элемента Button следующим образом:

```
private void btnProcessImages_Click(object sender, EventArgs e)
{
    // Запустить новую "задачу" для обработки файлов.
    Task.Factory.StartNew(() =>
    {
        ProcessFiles();
    });
}
```

Свойство Factory класса Task возвращает объект TaskFactory. При вызове метода StartNow() ему передается делегат Action<T> (здесь это скрыто подходящим лямбда-выражением), который указывает на метод, подлежащий вызову в асинхронной манере. После этой небольшой модификации вы обнаружите, что заголовок окна отображает информацию о потоке из пула, обрабатывающем конкретный файл, а текстовое поле может принимать ввод, поскольку пользовательский интерфейс больше не блокируется.

Обработка запроса на отмену

В текущий пример можно внести еще одно усовершенствование — предоставить пользователю способ для останова обработки данных изображения путем щелчка на второй кнопке Cancel (Отмена). К счастью, оба метода, Parallel.For() и Parallel.ForEach(), поддерживают отмену через использование *признаков отмены*. При вызове методов на Parallel можно передавать объект ParallelOptions, который, в свою очередь, содержит объект CancellationTokensource.

Прежде всего, определим в производном от Form классе следующую закрытую переменную-член cancelToken типа CancellationTokensource:

```
public partial class MainForm : Form
{
    // Новая переменная уровня Form.
    private CancellationTokensource cancelToken =
        new CancellationTokensource();
    ...
}
```

Предполагая, что был добавлен новый элемент Button (по имени btnCancel), реализуем его обработчик события Click следующим образом:

```
private void btnCancel_Click(object sender, EventArgs e)
{
    // Это будет использоваться для сообщения всем рабочим потокам
    // о необходимости останова!
    cancelToken.Cancel();
}
```

Теперь можно внести необходимые модификации в метод ProcessFiles(). Ниже показана окончательная реализация этого метода:

```
private void ProcessFiles()
{
    // Использовать экземпляр ParallelOptions для хранения CancellationToken.
    ParallelOptions parOpts = new ParallelOptions();
    parOpts.CancellationToken = cancelToken.Token;
    parOpts.MaxDegreeOfParallelism = System.Environment.ProcessorCount;
```

```
// Загрузить все файлы *.jpg и создать новую папку для модифицированных данных.
string[] files = Directory.GetFiles
(@"C:\Users\Public\Pictures\Sample Pictures", "*.jpg",
    SearchOption.AllDirectories);
string newDir = @"C:\ModifiedPictures";
Directory.CreateDirectory(newDir);

try
{
    // Обработать данные изображения в параллельном режиме!
    Parallel.ForEach(files, parOpts, currentFile =>
    {
        parOpts.CancellationToken.ThrowIfCancellationRequested();

        string filename = Path.GetFileName(currentFile);
        using (Bitmap bitmap = new Bitmap(currentFile))
        {
            bitmap.RotateFlip(RotateFlipType.Rotate180FlipNone);
            bitmap.Save(Path.Combine(newDir, filename));

            this.Invoke((Action)delegate
            {
                this.Text = string.Format("Processing {0} on thread {1}", filename,
                    Thread.CurrentThread.ManagedThreadId);
            });
        }
    });
}
catch (OperationCanceledException ex)
{
    this.Invoke((Action)delegate
    {
        this.Text = ex.Message;
    });
}
}
```

Обратите внимание, что метод начинается с конфигурирования объекта `ParallelOptions` путем установки его свойства `CancellationToken` в `cancelToken`. Кроме того, при вызове метода `Parallel.ForEach()` во втором параметре передается объект `ParallelOptions`.

Внутри логики цикла осуществляется вызов `ThrowIfCancellationRequested()` на признаке отмены. Это гарантирует, что если пользователь щелкнет на кнопке `Cancel`, то все потоки будут остановлены и сгенерируется исключение времени выполнения. Перехватив исключение `OperationCanceledException`, можно включить сообщение об ошибке в текст главного окна.

Исходный код. Проект `DataParallelismWithForEach` доступен в подкаталоге `Chapter 19`.

Обеспечение параллелизма задач с помощью класса `Parallel`

В дополнение к обеспечению параллелизма данных, библиотека TPL также может применяться для запуска любого количества асинхронных задач с помощью метода `Parallel.Invoke()`. Этот подход немного проще, чем использование делегатов или типов из пространства имен `System.Threading`.

Тем не менее, если нужна более высокая степень контроля над выполняемыми задачами, следует отказаться от `Parallel.Invoke()` и напрямую работать с классом `Task`, как это делалось в предыдущем примере.

Чтобы проиллюстрировать параллелизм задач в действии, создадим новое приложение Windows Forms по имени `MyEBookReader` и импортируем в нем пространства имен `System.Threading.Tasks` и `System.Net`. Этот пример представляет собой модификацию примера из документации .NET Framework 4.5 SDK. Здесь мы будем извлекать публично доступную электронную книгу из сайта проекта Гутенберга (www.gutenberg.org) и затем параллельно выполнять набор длительных задач.

Графический пользовательский интерфейс состоит из многострочного текстового поля `TextBox` (по имени `txtBook`) и двух кнопок `Button` (`btnDownload` и `btnGetStats`). Для каждой кнопки понадобится обработать событие `Click`, а в файле кода формы объявить на уровне класса переменную `string` по имени `theEBook`. Ниже показана реализация обработчика события `Click` для кнопки `btnDownload`:

```
private void btnDownload_Click(object sender, EventArgs e)
{
    WebClient wc = new WebClient();
    wc.DownloadStringCompleted += (s, eArgs) =>
    {
        theEBook = eArgs.Result;
        txtBook.Text = theEBook;
    };
    // Загрузить электронную книгу "A Tale of Two Cities" Чарльза Диккенса.
    wc.DownloadStringAsync(new Uri("http://www.gutenberg.org/files/98/98-8.txt"));
}
```

Класс `WebClient` находится в пространстве имен `System.Net`. Он предоставляет несколько методов для отправки и получения данных от ресурса, идентифицируемого посредством URL. В свою очередь, многие из этих методов имеют асинхронные версии, такие как `DownloadStringAsync()`. Этот метод автоматически запускает новый поток, взятый из пула потоков CLR. Когда `WebClient` завершает получение данных, он инициирует событие `DownloadStringCompleted`, которое обрабатывается с использованием лямбда-выражения C#. Если вызвать синхронную версию этого метода (`DownloadString()`), то форма на некоторое время перестанет реагировать на действия пользователя.

Обработчик события `Click` кнопки `btnGetStats` реализован так, чтобы извлекать индивидуальные слова, содержащиеся в переменной `theEBook`, и передавать строковый массив на обработку нескольким вспомогательным функциям, как показано ниже:

```
private void btnGetStats_Click(object sender, EventArgs e)
{
    // Получить слова из электронной книги.
    string[] words = theEBook.Split(new char[]
    { ' ', '\u000A', ',', '.', ';', ':', '-', '?', '/' },
        StringSplitOptions.RemoveEmptyEntries);

    // Найти 10 наиболее часто встречающихся слов.
    string[] tenMostCommon = FindTenMostCommon(words);

    // Получить самое длинное слово.
    string longestWord = FindLongestWord(words);

    // Когда все задачи завершены, построить строку,
    // показывающую всю статистику в окне сообщений.
    StringBuilder bookStats = new StringBuilder("Ten Most Common Words are:\n");
    foreach (string s in tenMostCommon)
    {
        bookStats.AppendLine(s);
    }
}
```

```

bookStats.AppendFormat("Longest word is: {0}", longestWord);
bookStats.AppendLine();
MessageBox.Show(bookStats.ToString(), "Book info");
}

```

Метод `FindTenMostCommon()` использует запрос LINQ для получения списка объектов `string`, которые наиболее часто встречаются в массиве `string`, а метод `FindLongestWord()` находит самое длинное слово:

```

private string[] FindTenMostCommon(string[] words)
{
    var frequencyOrder = from word in words
                          where word.Length > 6
                          group word by word into g
                          orderby g.Count() descending
                          select g.Key;

    string[] commonWords = (frequencyOrder.Take(10)).ToArray();
    return commonWords;
}

private string FindLongestWord(string[] words)
{
    return (from w in words orderby w.Length descending select w).FirstOrDefault();
}

```

После запуска этого проекта на выполнение всех задач может потребоваться некоторое время, в зависимости от количества процессоров машины и их тактовой частоты. В конце концов, должен появиться вывод, показанный на рис. 19.4.

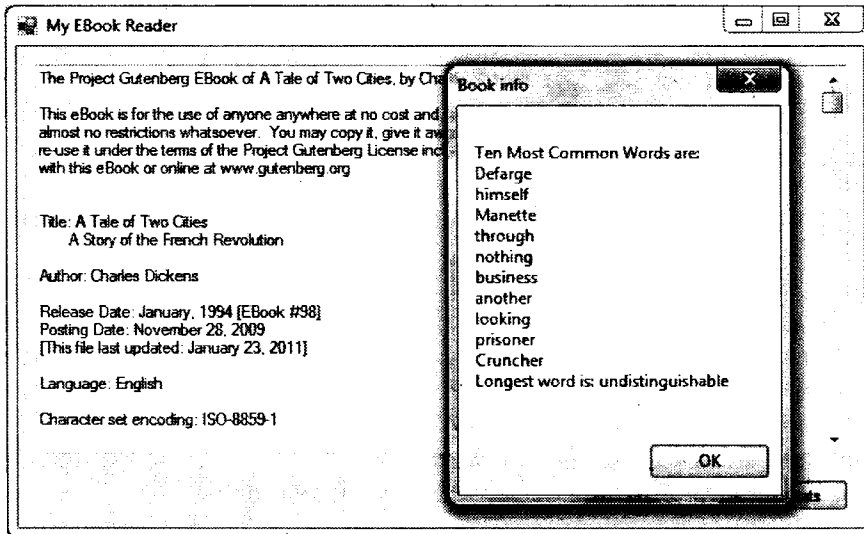


Рис. 19.4. Статистика загруженной электронной книги

Помочь удостовериться, что приложение использует все доступные процессоры машины, может параллельный вызов методов `FindTenMostCommon()` и `FindLongestWord()`. Для этого необходимо модифицировать метод `btnGetStats_Click()` следующим образом:

```

private void btnGetStats_Click(object sender, EventArgs e)
{
    // Получить слова из электронной книги.
    string[] words = theEBook.Split(
        new char[] { ' ', '\u000A', ',', '.', ';', ':', '-', '?', '/' },
        StringSplitOptions.RemoveEmptyEntries);
}

```

```

string[] tenMostCommon = null;
string longestWord = string.Empty;

Parallel.Invoke(
    () =>
    {
        // Найти 10 наиболее часто встречающихся слов.
        tenMostCommon = FindTenMostCommon(words);
    },
    () =>
    {
        // Найти самое длинное слово.
        longestWord = FindLongestWord(words);
    });

// Когда все задачи завершены, построить строку,
// показывающую всю статистику в окне сообщений.
...
}

```

Метод `Parallel.Invoke()` ожидает в качестве параметра массива делегатов `Action<>`, который передается неявно с помощью лямбда-выражения. Хотя вывод идентичен, преимущество библиотеки TPL состоит в использовании всех доступных процессоров машины для вызова каждого метода параллельно, если это возможно.

Исходный код. Проект `MyEBookReader` доступен в подкаталоге `Chapter 19`.

Запросы Parallel LINQ (PLINQ)

В завершение знакомства с библиотекой TPL следует отметить, что существует и другой способ встраивания параллельных задач в приложения .NET. При желании можно использовать набор расширяющих методов, которые позволяют конструировать запрос `LINQ`, распределяющий свою нагрузку по параллельным потокам (если это возможно). Неудивительно, что запросы `LINQ`, которые спроектированы для параллельного выполнения, называются *запросами Parallel LINQ (PLINQ)*.

Подобно параллельному коду, написанному с использованием класса `Parallel`, в `PLINQ` имеется опция игнорирования запроса на обработку коллекции в параллельном режиме. Библиотека `PLINQ` оптимизирована во многих отношениях, включая определение того, не будет ли запрос на самом деле эффективнее выполняться в синхронном режиме.

Во время выполнения `PLINQ` анализирует общую структуру запроса, и если есть вероятность, что запрос выиграет от параллелизма, он будет выполнен параллельно. Однако если это ухудшит производительность, `PLINQ` выполнит запрос последовательно. Если возникает выбор между потенциально дорогостоящим (в смысле ресурсов) параллельным алгоритмом и недорогим последовательным, предпочтение по умолчанию отдается последовательному алгоритму.

Необходимые расширяющие методы находятся в классе `ParallelEnumerable` пространства им

Чтобы посмотреть на `PLINQ` в действии, создадим приложение `Windows Forms` по имени `PLINQDataProcessingWithCancellation` и импортируем в него пространство имен `System.Threading`. Эта простая форма потребует всего двух кнопок с именами `btnExecute` и `btnCancel`. `ен System.Linq`. В табл. 19.5 описаны некоторые полезные расширения `PLINQ`.

Таблица 19.5. Избранные члены класса `ParallelEnumerable`

Член	Назначение
<code>AsParallel()</code>	Указывает, что остаток запроса должен быть выполнен параллельно, если это возможно
<code>WithCancellation()</code>	Указывает, что PLINQ должен периодически следить за состоянием предоставленного признака отмены и, если понадобится, отменять выполнение
<code>WithDegreeOfParallelism()</code>	Указывает максимальное количество процессоров, которое PLINQ должен использовать для распараллеливания запроса
<code>ForAll()</code>	Позволяет обрабатывать результаты параллельно, без предварительного слияния с потоком потребителя, как это происходит при перечислении результата LINQ посредством ключевого слова <code>foreach</code>

Щелчок на кнопке `Execute` (Выполнить) приводит к запуску новой задачи (`Task`), которая выполнит запрос LINQ, просматривающий очень большой массив целых чисел в поисках элементов, для которых остаток от деления на 3 равен 0. Ниже показана *непараллельная* версия этого запроса:

```
public partial class MainForm : Form
{
    ...
    private void btnExecute_Click(object sender, EventArgs e)
    {
        // Запустить новую "задачу" для обработки целых чисел.
        Task.Factory.StartNew(() =>
        {
            ProcessIntData();
        });
    }

    private void ProcessIntData()
    {
        // Получить очень большой массив целых чисел.
        int[] source = Enumerable.Range(1, 10000000).ToArray();
        // Найти числа, для которых истинно условие num % 3 == 0,
        // и вернуть их в порядке убывания.
        int[] modThreeIsZero = (from num in source where num % 3 == 0
                                orderby num descending select num).ToArray();

        MessageBox.Show(string.Format("Found {0} numbers that match query!",
            modThreeIsZero.Count()));
    }
}
```

Выполнение запроса PLINQ

Чтобы заставить TPL выполнить этот запрос в параллельном режиме (по возможности), для этого придется воспользоваться расширяющим методом `AsParallel()`:

```
int[] modThreeIsZero = (from num in source.AsParallel() where num % 3 == 0
                        orderby num descending select num).ToArray();
```

Обратите внимание, что общий формат запроса LINQ идентичен тому, что вы видели в предыдущих главах. Однако при включенном вызове `AsParallel()` библиотека TPL попытается распределить рабочую нагрузку по всем доступным процессорам.

Отмена запроса PLINQ

С помощью объекта `CancellationTokenSource` можно заставить запрос PLINQ прекращать обработку при определенных условиях (обычно из-за вмешательства пользователя). Для этого потребуется объявить на уровне формы объект `CancellationTokenSource` по имени `cancelToken` и реализовать обработчик события `Click` кнопки `btnCancel`, внутри которого вызвать метод `Cancel()` на этом объекте.

Ниже показаны соответствующие изменения в коде:

```
public partial class MainForm : Form
{
    private CancellationTokenSource cancelToken = new CancellationTokenSource();
    private void btnCancel_Click(object sender, EventArgs e)
    {
        cancelToken.Cancel();
    }
    ...
}
```

Теперь необходимо информировать запрос PLINQ о том, что он должен ожидать входящего запроса на отмену выполнения, добавив в цепочку расширяющий метод `WithCancellation()` и передав признак отмены. Кроме того, этот запрос PLINQ нужно поместить в контекст `try/catch` и обработать возможные исключения. Финальная версия метода `ProcessIntData()` выглядит следующим образом:

```
private void ProcessIntData()
{
    // Получить очень большой массив целых чисел.
    int[] source = Enumerable.Range(1, 10000000).ToArray();
    // Найти числа, для которых истинно условие num % 3 == 0,
    // и вернуть их в порядке убывания.
    int[] modThreeIsZero = null;
    try
    {
        modThreeIsZero = (from num in
            source.AsParallel().WithCancellation(cancelToken.Token)
            where num % 3 == 0 orderby num descending
            select num).ToArray();
        MessageBox.Show(string.Format("Found {0} numbers that match query!",
            modThreeIsZero.Count()));
    }
    catch (OperationCanceledException ex)
    {
        this.Invoke((Action)delegate
        {
            this.Text = ex.Message;
        });
    }
}
```

Асинхронные вызовы в версии .NET 4.5

В этой (довольно длинной) главе мы раскрыли много насыщенного материала. Конечно, построение, отладка и понимание сложных многопоточных приложений является проблемой в любой инфраструктуре. Хотя TPL, PLINQ и тип делегата могут до некоторой степени упростить решение (особенно по сравнению с другими платформами и языками), разработчики по-прежнему должны быть знакомы со всеми подробностями разнообразных расширенных технологий.

С выходом версии .NET 4.5 в языке программирования C# (а также и VB) появились два новых ключевых слова, которые дополнительно упрощают процесс написания асинхронного кода. В отличие от всех примеров, показанных ранее в этой главе, когда применяются новые ключевые слова `async` и `await`, компилятор будет самостоятельно генерировать большой объем кода, связанного с потоками, используя многочисленные члены пространств имен `System.Threading` и `System.Threading.Tasks`.

Знакомство с ключевыми словами `async` и `await` языка C#

Ключевое слово `async` языка C# применяется для указания на то, что метод, лямбда-выражение или анонимный метод должны вызываться в асинхронной манере *автоматически*. Да, это правда. Благодаря простой пометке метода с помощью модификатора `async`, среда CLR будет создавать новый поток выполнения для обработки текущей задачи. Более того, при вызове метода `async` ключевое слово `await` будет *автоматически* приостанавливать текущий поток до тех пор, пока задача не завершится, давая возможность вызывающему потоку продолжать свою работу.

В целях иллюстрации создадим новое приложение Windows Forms по имени `FunWithCSharpAsync` и импортируем пространство имен `System.Threading` в первоначальный файл кода формы (с переименованием исходной формы в `MainForm`). После этого поместим на поверхность визуального конструктора один элемент управления `Button` (по имени `btnCallMethod`) и один элемент управления `TextBox` (по имени `txtInput`), а затем сконфигурируем базовые свойства пользовательского интерфейса (цвета, шрифты, текст) желаемым образом. В обработчике события `Click` кнопки `btnCallMethod` вызовем закрытый вспомогательный метод `DoWork()`, который заставляет вызывающий поток ожидать 10 секунд. Ниже показан код:

```
public partial class MainForm : Form
{
    public MainForm()
    {
        InitializeComponent();
    }

    private void btnCallMethod_Click(object sender, EventArgs e)
    {
        this.Text = DoWork();
    }

    private string DoWork()
    {
        Thread.Sleep(10000);
        return "Done with work!";
    }
}
```

Теперь вам известно, что после запуска программы и щелчка на кнопке придется ждать 10 секунд, пока текстовое поле сможет получить ввод с клавиатуры. Кроме того,

в течение 10 секунд не будет также обновлен сообщением Done with work! и заголовок главного окна.

Если бы мы решили воспользоваться любым ранее описанным в этой главе приемом для того, чтобы сделать приложение более отзывчивым, пришлось бы немало потрудиться. Тем не менее, в .NET 4.5 мы можем написать следующий код C#:

```
public partial class MainForm : Form
{
    public MainForm()
    {
        InitializeComponent();
    }

    private async void btnCallMethod_Click(object sender, EventArgs e)
    {
        this.Text = await DoWork();
    }

    // Ниже приведены пояснения по этому коду...
    private Task<string> DoWork()
    {
        return Task.Run(() =>
        {
            Thread.Sleep(10000);
            return "Done with work!";
        });
    }
}
```

Первым делом, обратите внимание, что обработчик события Click кнопки помечен ключевым словом **async**. Это значит, что данный метод должен вызываться в неблокирующей манере. Кроме того, в реализации обработчика события *перед* именем вызываемого метода используется ключевое слово **await**. Это важный момент: если метод декорируется ключевым словом **async**, но не имеет хотя бы одного внутреннего вызова метода с применением **await**, получается блокирующий, синхронный вызов (на самом деле в таком случае компилятор выдаст соответствующее предупреждение).

Мы должны были воспользоваться классом Task из пространства имен System.Threading.Tasks для проведения рефакторинга метода DoWork() с целью обеспечения его корректного функционирования. В сущности, вместо возврата специфического значения напрямую (объекта string в текущем примере) мы возвращаем объект Task<T>, где обобщенный параметр типа T — это лежащее в основе возвращаемое значение.

Реализация DoWork() теперь непосредственно возвращает объект Task<T>, который является возвращаемым значением Task.Run(). Метод Run() принимает делегат Func<> или Action<> и, как уже известно к этому моменту, реализацию можно упростить за счет применения лямбда-выражения. В целом новая версия DoWork() обладает следующими характеристиками.

При вызове запускается новая задача. Эта задача заставляет вызывающий поток уснуть на 10 секунд. По завершении вызывающий поток предоставляет строковое возвращаемое значение. Эта строка помещается в новый объект Task<string> и возвращается вызывающему коду.

Благодаря этой новой реализации метода DoWork(), мы можем получить некоторое представление о действительной роли ключевого слова **await**. Оно всегда будет модифицировать метод, который возвращает объект Task.

Когда поток выполнения достигает `await`, вызывающий поток приостанавливается до тех пор, пока вызов не будет завершен. Запустив эту версию приложения, вы обнаружите, что можно щелкнуть на кнопке и сразу же вводить в текстовом поле. Спустя 10 секунд заголовок окна будет обновлен сообщением о завершении работы.

Соглашения об именовании асинхронных методов

А теперь предположим, что новая версия метода `DoWork()` осталась точно такой же, как показанная ранее, однако обработчик события `Click` кнопки реализован следующим образом:

```
private async void btnCallMethod_Click(object sender, EventArgs e)
{
    // Ключевое слово await отсутствует!
    this.Text = DoWork();
}
```

Обратите внимание, что мы действительно поместили метод `DoWork()` ключевым словом `async`, но не указали ключевое слово `await` при его вызове. В этой точке мы получим ошибки на этапе компиляции, поскольку возвращаемым значением `DoWork()` является объект `Task`, который мы пытаемся присвоить непосредственно свойству `Text` (имеющему тип `string`). Вспомните, что ключевое слово `await` отвечает за извлечение внутреннего возвращаемого значения, содержащегося в объекте `Task`. Из-за отсутствия этого ключевого слова возникает несовпадение типов. Как показано на рис. 19.5, компилятор информирует о том, что метод, который мы пытаемся вызвать, поддерживает `await` (“awaitable”), и предлагает корректную форму его вызова.

На заметку! Метод, поддерживающий `await` (“awaitable”) — это просто метод, который возвращает `Task<T>`.

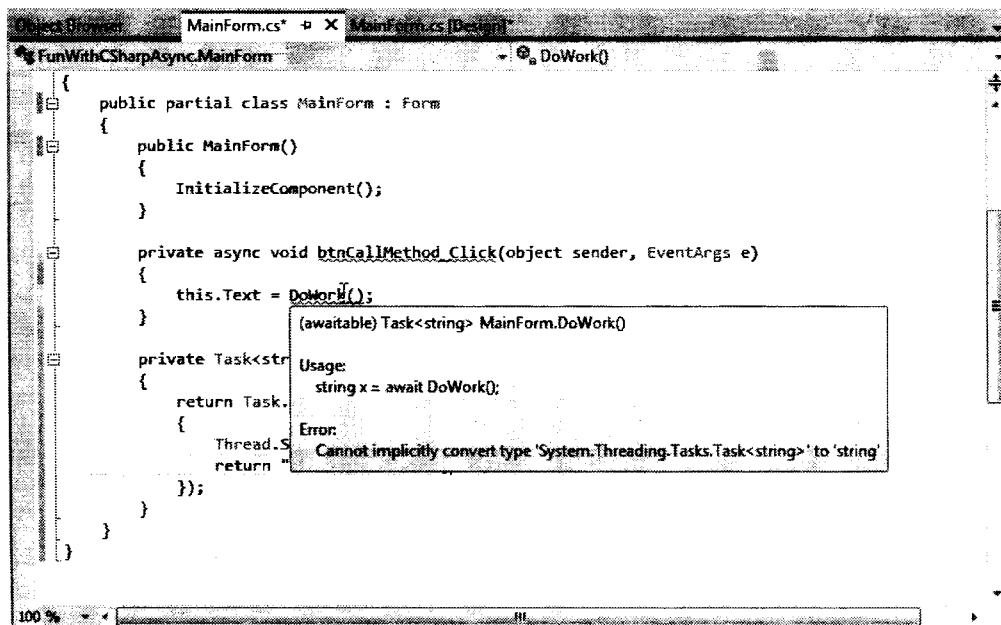


Рис. 19.5. Методы, которые возвращают объект `Task`, могут быть вызваны через `await`

С учетом того факта, что методы, которые возвращают объекты `Task`, могут теперь вызываться в неблокирующей манере через конструкции `async` и `await`, в Microsoft рекомендуют помечать любой метод, возвращающий `Task`, посредством суффикса `Async`. В этом случае разработчики, которым известно данное соглашение об именовании, получают визуальное напоминание о том, что ключевое слово `await` является обязательным, если такой метод планируется вызывать в асинхронном контексте.

На заметку! Обработчики событий для элементов управления графического пользовательского интерфейса (вроде обработчика события `Click` кнопки в примере), которые используют ключевые слова `async/await`, не следуют этому соглашению об именовании.

Кроме того, метод `DoWork()` может также быть декорирован с помощью ключевых слов `async` и `await` (хотя это не является обязательным в текущем примере). С учетом всего этого ниже приведена окончательная модификация текущего примера, удовлетворяющая рекомендуемым соглашениям об именовании:

```
public partial class MainForm : Form
{
    public MainForm()
    {
        InitializeComponent();
    }
    private async void btnCallMethod_Click(object sender, EventArgs e)
    {
        this.Text = await DoWorkAsync();
    }
    private async Task<string> DoWorkAsync()
    {
        return await Task.Run(() =>
        {
            Thread.Sleep(10000);
            return "Done with work!";
        });
    }
}
```

Асинхронные методы, возвращающие `void`

В настоящий момент наш метод `DoWork()` возвращает объект `Task`, содержащий “действительные данные” для вызывающего кода, которые будут получены прозрачным образом через ключевое слово `await`. Однако как быть, если требуется построить асинхронный метод, возвращающий `void`? В этом случае мы используем необобщенный класс `Task` и опускаем любые операторы `return`, например:

```
private async Task MethodReturningVoidAsync()
{
    await Task.Run(() => { /* Выполнение каких-то действий... */
        Thread.Sleep(4000);
    });
}
```

Затем в коде, вызывающем этот метод, таком как обработчик события `Click` второй кнопки, можно применить ключевые слова `await` и `async`, как показано ниже:

```
private async void btnVoidMethodCall_Click(object sender, EventArgs e)
{
    await MethodReturningVoidAsync();
    MessageBox.Show("Done!");
}
```

Асинхронные методы с множеством контекстов `await`

Внутри реализации асинхронного метода вполне допустимо иметь множество контекстов `await`. Предположим, что в приложение добавлен обработчик события `Click` третьей кнопки, помеченный ключевым словом `async`. В предыдущих частях этого примера обработчики события `Click` специально вызывали некоторый внешний метод, запускающий задачу; тем не менее, эту логику можно было бы встроить через набор лямбда-выражений:

```
private async void btnMutliAwaits_Click(object sender, EventArgs e)
{
    await Task.Run(() => { Thread.Sleep(2000); });
    MessageBox.Show("Done with first task!"); // завершена первая задача
    await Task.Run(() => { Thread.Sleep(2000); });
    MessageBox.Show("Done with second task!"); // завершена вторая задача
    await Task.Run(() => { Thread.Sleep(2000); });
    MessageBox.Show("Done with third task!"); // завершена третья задача
}
```

Здесь каждая задача всего лишь приостанавливает текущий поток на некоторый период времени; тем не менее, с помощью этих задач может быть представлена любая единица работы (обращение к веб-службе, чтение базы данных или что-нибудь еще). Ниже перечислены ключевые моменты, связанные с этим примером.

- Методы (а также лямбда-выражения или анонимные методы) могут быть помечены с помощью ключевого слова `async`, что позволяет методу работать в неблокирующей манере.
- Методы (а также лямбда-выражения или анонимные методы), помеченные ключевым словом `async`, будут выполняться в *блокирующей* манере до тех пор, пока не встретится ключевое слово `await`.
- Один метод `async` может иметь множество контекстов `await`.
- Как только встретилось выражение `await`, вызывающий поток приостанавливается вплоть до завершения ожидаемой задачи. Тем временем управление возвращается коду, вызвавшему метод.
- Ключевое слово `await` будет скрывать возвращаемый объект `Task`, выглядя как прямой возврат лежащего в основе возвращаемого значения. Методы, не имеющие возвращаемого значения, просто возвращают `void`.
- По соглашению об именовании методы, которые могут быть вызваны асинхронно, должны быть помечены с помощью суффикса `Async`.

Исходный код. Проект `FunWithCSharpAsync` доступен в подкаталоге `Chapter 19`.

Модернизация примера `AddWithThreads` с использованием `async/await`

Ранее в этой главе был разработан пример под названием `AddWithThreads` с применением первоначального API-интерфейса для многопоточности в .NET — пространства имен `System.Threading`. Теперь давайте модернизируем этот пример с использованием новых ключевых слов `async` и `await` языка C#, чтобы продемонстрировать, насколько яснее может стать логика приложения. Для начала вспомним, как изначально строился проект `AddWithThreads`.

- Мы создали специальный класс по имени `AddParams`, который представлял данные, предназначенные для суммирования.
- Мы применяли класс `Thread` и делегат `ParameterizedThreadStart` для указания на метод `Add()`, принимающий объект `AddParams`.
- Мы использовали класс `AutoResetEvent` для обеспечения ожидания вызывающим потоком завершения работы вторичного потока.

В общем, нам пришлось приложить немало усилий, чтобы всего лишь подсчитать сумму двух чисел во вторичном потоке выполнения! Ниже приведен код того же самого проекта, переделанный с применением новых технологий .NET 4.5 (код класса `AddParams` здесь не показан, но, как вы должны помнить, он просто имеет два поля, `a` и `b`, для представления суммируемых данных):

```
class Program
{
    static void Main(string[] args)
    {
        AddAsync();
        Console.ReadLine();
    }

    private static async Task AddAsync()
    {
        Console.WriteLine("***** Adding with Thread objects *****");
        Console.WriteLine("ID of thread in Main(): {0}",
            Thread.CurrentThread.ManagedThreadId);

        AddParams ap = new AddParams(10, 10);
        await Sum(ap);

        Console.WriteLine("Other thread is done!");
    }

    static async Task Sum(object data)
    {
        await Task.Run(() =>
        {
            if (data is AddParams)
            {
                Console.WriteLine("ID of thread in Add(): {0}",
                    Thread.CurrentThread.ManagedThreadId);

                AddParams ap = (AddParams)data;
                Console.WriteLine("{0} + {1} is {2}",
                    ap.a, ap.b, ap.a + ap.b);
            }
        });
    }
}
```

Первое, что следует отметить — код, который изначально находился в `Main()`, был перемещен в новый метод по имени `AddAsync()`. Причина связана не только с соблюдением ожидаемого соглашения об именовании, но также и со следующим важным моментом.

На заметку! Метод `Main()` исполняемой сборки не может быть помечен с помощью ключевого слова `async`.

Обратите внимание, что метод `AddAsync()` помечен ключевым словом `async` и в нем определен контекст `await`. Кроме того, метод `Sum()` запускает новую задачу для выполнения единицы работы. В любом случае после запуска этой программы обнаружится, что числа 10 и 10 по-прежнему дают в сумме 20. Однако в данной ситуации мы имеем два уникальных идентификатора потоков (рис. 19.6).

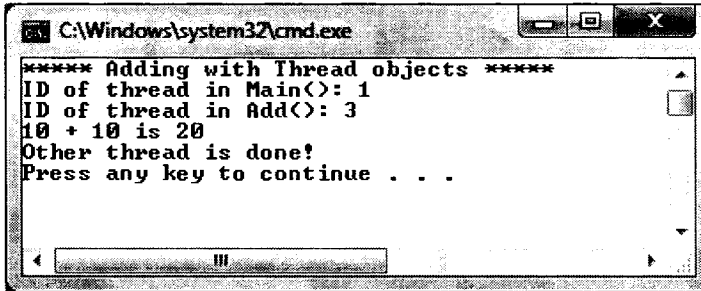


Рис. 19.6. Модернизированный пример `System.Threading`, в котором теперь используются асинхронные возможности .NET 4.5

Исходный код. Проект `AddWithThreadsAsync` доступен в подкаталоге `Chapter 19`.

Итак, как видите, ключевые слова `async` и `await`, введенные в .NET 4.5, радикально упрощают процесс вызова методов во вторичном потоке выполнения. Хотя было рассмотрено лишь несколько примеров применения новой функциональности языка C#, у вас появилась прочная основа для дальнейших исследований.

Резюме

Эта глава начиналась с описания того, каким образом конфигурировать типы делегатов .NET для выполнения метода в асинхронном режиме. Как вы видели, методы `BeginInvoke()` и `EndInvoke()` позволяют косвенно манипулировать вторичным потоком с минимальными усилиями со стороны разработчика. Далее были представлены интерфейс `IAsyncResult` и класс `AsyncResult`. Было показано, что эти типы предлагают различные способы синхронизации вызывающего потока и получения возможных возвращаемых значений методов.

Следующая часть главы была посвящена рассмотрению роли пространства имен `System.Threading`. Как было сказано, когда приложение создает дополнительные потоки выполнения, в результате оно может выполнять несколько задач (как кажется) одновременно. Также были продемонстрированы различные способы защиты чувствительных к потокам блоков кода для предотвращения повреждения разделяемых ресурсов.

Затем в главе исследовались новые модели для разработки многопоточных приложений, введенной в .NET 4.0, в частности, `Task Parallel Library` и `PLINQ`. В завершение главы была проанализирована роль новых ключевых слов `async` и `await` языка C#, которые появились в .NET 4.5. Вы видели, что эти ключевые слова используются многими типами в библиотеке TPL; тем не менее, большую часть работы по созданию кода для многопоточной обработки и синхронизации компилятор выполняет самостоятельно.

Файловый ввод-вывод и сериализация объектов

При создании настольных приложений способность сохранения информации между пользовательскими сеансами является обязательной. В этой главе рассматривается множество тем, касающихся ввода-вывода, с точки зрения платформы .NET Framework. Первая задача связана с исследованием основных типов, определенных в пространстве имен `System.IO`, которые позволяют программно модифицировать структуру каталогов и файлов. Вторая задача состоит в изучении различных способов чтения и записи символьных, двоичных и располагаемых в памяти структур данных.

Изучив способы манипулирования файлами и каталогами с использованием базовых типов ввода-вывода, вы ознакомитесь с родственной темой — *сериализацией объектов*. Сериализация объектов служит для сохранения и восстановления состояния объекта в любом типе, производном от `System.IO.Stream`. Возможность сериализации объектов критична, когда необходимо копировать объект на удаленную машину с помощью технологий удаленного взаимодействия, таких как Windows Communication Foundation. Однако сериализация удобна и сама по себе, и она наверняка пригодится во многих разрабатываемых приложениях .NET (как распределенных, так и обычных).

Исследование пространства имен `System.IO`

Пространство имен `System.IO` в .NET — это область библиотек базовых классов, посвященная службам файлового ввода-вывода, а также ввода-вывода из памяти. Подобно любому пространству имен, в `System.IO` определен набор классов, интерфейсов, перечислений, структур и делегатов, большинство из которых находятся в `mscorlib.dll`. В дополнение к типам, содержащимся внутри `mscorlib.dll`, в сборке `System.dll` определены дополнительные члены пространства имен `System.IO`. Обратите внимание, что во всех проектах Visual Studio автоматически устанавливаются ссылки на обе сборки.

Многие типы из пространства имен `System.IO` сосредоточены на программной манипуляции физическими каталогами и файлами. Дополнительные типы предоставляют поддержку чтения и записи данных в строковые буферы, а также области памяти.

В табл. 20.1 кратко описаны основные (неабстрактные) классы, которые дают понятие о функциональности `System.IO`.

Таблица 20.1. Ключевые члены пространства имен `System.IO`

Неабстрактные классы ввода-вывода	Описание
BinaryReader BinaryWriter	Эти классы позволяют сохранять и извлекать элементарные типы данных (целочисленные, булевские, строковые и т.п.) в двоичном виде
BufferedStream	Этот класс предоставляет временное хранилище для потока байтов, который может быть зафиксирован в постоянном хранилище в более позднее время
Directory DirectoryInfo	Эти классы используются для манипуляций структурой каталогов машины. Тип <code>Directory</code> открывает функциональность с использованием <i>статических членов</i> . Тип <code>DirectoryInfo</code> обеспечивает аналогичную функциональность через действительную <i>объектную ссылку</i>
DriveInfo	Этот класс предоставляет детальную информацию относительно дисковых устройств, используемых на заданной машине
File FileInfo	Эти классы служат для манипулирования множеством файлов на машине. Тип <code>File</code> открывает функциональность через <i>статические члены</i> . Тип <code>FileInfo</code> обеспечивает аналогичную функциональность через действительную <i>объектную ссылку</i>
FileStream	Этот класс предоставляет произвольный доступ к файлу (например, возможности поиска) с данными, представленными в виде потока байт
FileSystemWatcher	Этот класс позволяет отслеживать модификации внешних файлов в определенном каталоге
MemoryStream	Этот класс обеспечивает произвольный доступ к данным, хранящимся в памяти, а не в физическом файле
Path	Этот класс выполняет операции над типами <code>System.String</code> , содержащими информацию о пути к файлу или каталогу в независимой от платформы манере
StreamWriter StreamReader	Эти классы используются для хранения (и извлечения) текстовой информации из файла. Они не поддерживают произвольный доступ к файлу
StringWriter StringReader	Подобно <code>StreamWriter/StreamReader</code> , эти классы также работают с текстовой информацией. Однако лежащим в основе хранилищем является строковый буфер, а не физический файл

В дополнение к этим конкретным типам классов в `System.IO` определено несколько перечислений, а также набор абстрактных классов (например, `Stream`, `TextReader` и `TextWriter`), которые определяют разделяемый полиморфный интерфейс для всех наследников. В этой главе вы узнаете о множестве этих типов.

Классы `Directory` (`DirectoryInfo`) и `File` (`FileInfo`)

В `System.IO` предоставляются четыре класса, которые позволяют манипулировать индивидуальными файлами, а также взаимодействовать со структурой каталогов машины. Первые два класса, `Directory` и `File`, предлагают операции создания, удаления, копирования и перемещения с использованием различных статических членов. Родственные им классы `FileInfo` и `DirectoryInfo` обеспечивают похожую функциональность в виде методов уровня экземпляра (и потому должны размещаться в памяти с помощью ключевого слова `new`). Как показано на рис. 20.1, классы `Directory` и `File` напрямую расширяют `System.Object`, в то время как `DirectoryInfo` и `FileInfo` являются производными от абстрактного класса `FileSystemInfo`.

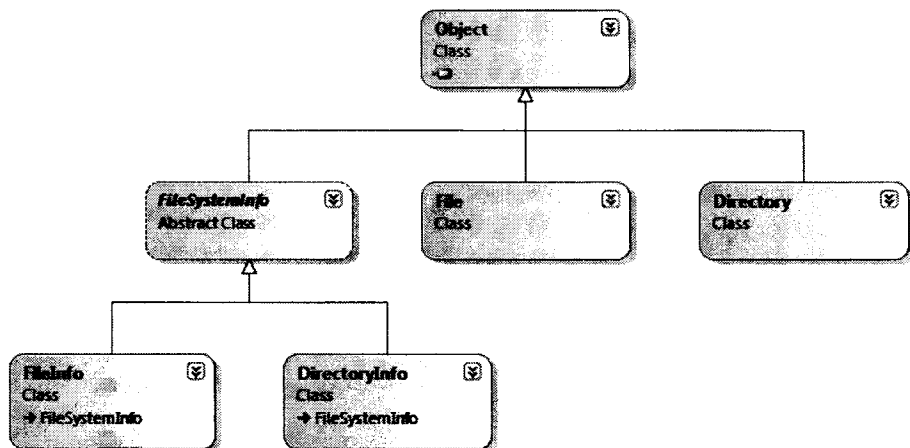


Рис. 20.1. Классы для работы с файлами и каталогами

Обычно `FileInfo` и `DirectoryInfo` представляют собой лучший выбор для получения полных деталей о файле или каталоге (например, время создания, возможности чтения/записи и т.п.), поскольку их члены возвращают строго типизированные объекты. В отличие от этого, члены классов `Directory` и `File`, как правило, возвращают простые строковые значения, а не строго типизированные объекты. Тем не менее, это лишь руководящий принцип; во многих случаях одну и ту же работу можно делать, используя `File/FileInfo` или `Directory/DirectoryInfo`.

Абстрактный базовый класс `FileSystemInfo`

Классы `DirectoryInfo` и `FileInfo` наследуют значительную часть своего поведения от абстрактного базового класса `FileSystemInfo`. По большей части члены класса `FileSystemInfo` используются для выяснения общих характеристик (таких как время создания, различные атрибуты и т.д.) определенного файла или каталога. В табл. 20.2 перечислены некоторые основные свойства, представляющие интерес.

Таблица 20.2. Избранные свойства класса `FileSystemInfo`

Свойство	Описание
<code>Attributes</code>	Получает или устанавливает ассоциированные с текущим файлом атрибуты, которые представлены перечислением <code>FileAttributes</code> (например, доступный только для чтения, зашифрованный, скрытый или сжатый)
<code>CreationTime</code>	Получает или устанавливает время создания текущего файла или каталога
<code>Exists</code>	Может использоваться для определения, существует ли данный файл или каталог
<code>Extension</code>	Извлекает расширение файла
<code>FullName</code>	Получает полный путь к файлу или каталогу
<code>LastAccessTime</code>	Получает или устанавливает время последнего доступа к текущему файлу или каталогу
<code>LastWriteTime</code>	Получает или устанавливает время последней записи в текущий файл или каталог
<code>Name</code>	Получает имя текущего файла или каталога

В классе `FileSystemInfo` также определен метод `Delete()`. Этот метод реализуется производными типами для удаления файла или каталога с жесткого диска. Кроме того, перед получением информации об атрибутах можно вызвать метод `Refresh()`, чтобы обеспечить актуальность статистики о текущем файле или каталоге.

Работа с типом `DirectoryInfo`

Первый неабстрактный тип, связанный с вводом-выводом, который мы рассмотрим здесь — `DirectoryInfo`. Этот класс содержит набор членов, используемых для создания, перемещения, удаления и перечисления каталогов и подкаталогов. В дополнение к функциональности, предоставленной его базовым классом (`FileSystemInfo`), класс `DirectoryInfo` предлагает ключевые члены, описанные в табл. 20.3.

Таблица 20.3. Ключевые члены типа `DirectoryInfo`

Член	Описание
<code>Create()</code> <code>CreateSubdirectory()</code>	Создает каталог (или набор подкаталогов) по заданному путевому имени
<code>Delete()</code>	Удаляет каталог и все его содержимое
<code>GetDirectories()</code>	Возвращает массив объектов <code>DirectoryInfo</code> , представляющих все подкаталоги в текущем каталоге
<code>GetFiles()</code>	Извлекает массив объектов <code>FileInfo</code> , представляющий набор файлов в заданном каталоге
<code>MoveTo()</code>	Перемещает каталог со всем содержимым по новому пути
<code>Parent</code>	Извлекает родительский каталог данного каталога
<code>Root</code>	Получает корневую часть пути

Работа с типом `DirectoryInfo` начинается с указания определенного пути в качестве параметра конструктора. Если требуется получить доступ к текущему рабочему каталогу (т.е. каталогу выполняющегося приложения), применяйте обозначение в виде точки (`.`). Вот некоторые примеры:

```
// Привязаться к текущему рабочему каталогу.
DirectoryInfo dir1 = new DirectoryInfo(".");

// Привязаться к C:\Windows, используя дословную строку.
DirectoryInfo dir2 = new DirectoryInfo(@"C:\Windows");
```

Во втором примере предполагается, что переданный конструктору путь (`C:\Windows`) уже существует на физической машине. Однако при попытке взаимодействия с несуществующим каталогом генерируется исключение `System.IO.DirectoryNotFoundException`. Таким образом, чтобы указать каталог, который пока еще не создан, сначала придется вызвать метод `Create()`:

```
// Привязаться к несуществующему каталогу, затем создать его.
DirectoryInfo dir3 = new DirectoryInfo(@"C:\MyCode\Testing");
dir3.Create();
```

После создания объекта `DirectoryInfo` можно исследовать содержимое лежащего в основе каталога с помощью любого свойства, унаследованного от `FileSystemInfo`. В целях иллюстрации создадим новое консольное приложение по имени `DirectoryApp` и импортируем в файл кода C# пространство имен `System.IO`.

Далее модифицируем класс Program, добавив показанный ниже новый статический метод, который создает объект DirectoryInfo, отображенный на C:\Windows (при необходимости подкорректируйте путь), и выводит ряд интересных статистических данных:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with Directory(Info) *****\n");
        ShowWindowsDirectoryInfo();
        Console.ReadLine();
    }

    static void ShowWindowsDirectoryInfo()
    {
        // Вывести информацию о каталоге.
        DirectoryInfo dir = new DirectoryInfo(@"C:\Windows");
        Console.WriteLine("***** Directory Info *****");
        Console.WriteLine("FullName: {0}", dir.FullName);    // полное имя
        Console.WriteLine("Name: {0}", dir.Name);            // имя каталога
        Console.WriteLine("Parent: {0}", dir.Parent);        // родительский каталог
        Console.WriteLine("Creation: {0}", dir.CreationTime); // время создания
        Console.WriteLine("Attributes: {0}", dir.Attributes); // атрибуты
        Console.WriteLine("Root: {0}", dir.Root);            // корневой каталог
        Console.WriteLine("*****\n");
    }
}
```

Вывод будет выглядеть примерно так, как показано ниже:

```
***** Fun with Directory(Info) *****
***** Directory Info *****
FullName: C:\Windows
Name: Windows
Parent:
Creation: 7/13/2012 10:22:32 PM
Attributes: Directory
Root: C:\
*****
```

Перечисление файлов с помощью типа DirectoryInfo

В дополнение к получению базовых деталей о существующем каталоге, текущий пример можно расширить использованием некоторых методов типа DirectoryInfo. Для начала применим метод GetFiles() для получения информации обо всех файлах *.jpg, расположенных в каталоге C:\Windows\Web\Wallpaper.

На заметку! Если на вашей машине нет каталога C:\Windows\Web\Wallpaper, измените код для чтения файлов из какого-то существующего каталога (например, прочитайте все файлы *.bmp из каталога C:\Windows).

Метод GetFiles() возвращает массив объектов типа FileInfo, каждый из которых отражает детальную информацию о конкретном файле (все подробности о типе FileInfo будут описаны далее в этой главе). Предположим, что в методе Main() вызывается следующий статический метод класса Program:

```

static void DisplayImageFiles()
{
    DirectoryInfo dir = new DirectoryInfo(@"C:\Windows\Web\Wallpaper");

    // Получить все файлы с расширением *.jpg.
    FileInfo[] imageFiles = dir.GetFiles("*.jpg", SearchOption.AllDirectories);

    // Сколько файлов найдено?
    Console.WriteLine("Found {0} *.jpg files\n", imageFiles.Length);

    // Вывести информацию о каждом файле.
    foreach (FileInfo f in imageFiles)
    {
        Console.WriteLine("*****");
        Console.WriteLine("File name: {0}", f.Name);           // имя файла
        Console.WriteLine("File size: {0}", f.Length);         // размер
        Console.WriteLine("Creation: {0}", f.CreationTime);    // время создания
        Console.WriteLine("Attributes: {0}", f.Attributes);    // атрибуты
        Console.WriteLine("*****\n");
    }
}

```

Обратите внимание на указание в вызове `GetFiles()` опции поиска; `SearchOption.AllDirectories` обеспечивает просмотр всех подкаталогов корня. После запуска этого приложения получается список файлов, отвечающих шаблону поиска.

Создание подкаталогов с помощью типа `DirectoryInfo`

Для программного расширения структуры каталогов служит метод `DirectoryInfo.CreateSubdirectory()`. С его помощью можно создавать как одиночный подкаталог, так и множество вложенных подкаталогов за один вызов. В целях иллюстрации ниже приведен метод, который расширяет структуру диска C: дополнительными подкаталогами:

```

static void ModifyAppDirectory()
{
    DirectoryInfo dir = new DirectoryInfo(@"C:\");

    // Создать \MyFolder в каталоге приложения.
    dir.CreateSubdirectory("MyFolder");

    // Создать \MyFolder2\Data в каталоге приложения.
    dir.CreateSubdirectory(@"MyFolder2\Data");
}

```

Вызвав этот метод в `Main()` и запустив программу, в проводнике Windows можно будет увидеть новые подкаталоги (рис. 20.2).

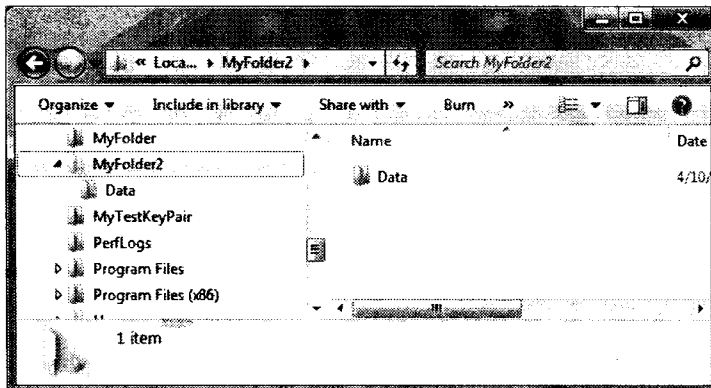


Рис. 20.2. Результат создания подкаталогов

Хотя получать возвращаемое значение метода `CreateSubdirectory()` не обязательно, имейте в виду, что в случае его успешного выполнения возвращается объект `DirectoryInfo`, представляющий вновь созданный элемент. Ниже показана модификация предыдущего метода. Обратите внимание на применение "." в конструкторе `DirectoryInfo`; это обеспечивает доступ к месту установки приложения.

```
static void ModifyAppDirectory()
{
    DirectoryInfo dir = new DirectoryInfo(".");
    // Создать \MyFolder в начальном каталоге.
    dir.CreateSubdirectory("MyFolder");
    // Получить возвращенный объект DirectoryInfo.
    DirectoryInfo myDataFolder = dir.CreateSubdirectory(@"MyFolder2\Data");
    // Выводит путь к ..\MyFolder2\Data.
    Console.WriteLine("New Folder is: {0}", myDataFolder);
}
```

Работа с типом Directory

После опробования типа `DirectoryInfo` в действии можно приступить к изучению типа `Directory`. По большей части статические члены `Directory` повторяют функциональность, предоставляемую членами уровня экземпляра, которые определены в `DirectoryInfo`. Однако вспомните, что члены `Directory` обычно возвращают строковые данные вместо строго типизированных объектов `FileInfo/DirectoryInfo`.

Теперь рассмотрим некоторую функциональность типа `Directory`. Вспомогательная функция, код которой приведен ниже, отображает имена всех устройств текущего компьютера (с помощью метода `Directory.GetLogicalDrives()`) и использует статический метод `Directory.Delete()` для удаления созданных ранее подкаталогов `\MyFolder` и `\MyFolder2\Data`.

```
static void FunWithDirectoryType()
{
    // Вывести список всех дисковых устройств текущего компьютера.
    string[] drives = Directory.GetLogicalDrives();
    Console.WriteLine("Here are your drives:");
    foreach (string s in drives)
        Console.WriteLine("--> {0} ", s);

    // Удалить то, что было ранее создано.
    Console.WriteLine("Press Enter to delete directories");
    Console.ReadLine();
    try
    {
        Directory.Delete(@"C:\MyFolder");
        // Второй параметр указывает, нужно ли удалять подкаталоги.
        Directory.Delete(@"C:\MyFolder2", true);
    }
    catch (IOException e)
    {
        Console.WriteLine(e.Message);
    }
}
```

Работа с типом DriveInfo

Пространство имен System.IO включает класс DriveInfo. Подобно Directory.GetLogicalDrives(), статический метод DriveInfo.GetDrives() позволяет получить имена устройств на машине. Однако, в отличие от Directory.GetLogicalDrives(), класс DriveInfo предоставляет множество дополнительных деталей (например, тип устройства, доступное свободное пространство и метка тома). Рассмотрим следующий класс Program, определенный в новом консольном приложении по имени DriveInfoApp (не забудьте импортировать пространство имен System.IO):

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with DriveInfo *****\n");
        // Получить информацию обо всех устройствах.
        DriveInfo[] myDrives = DriveInfo.GetDrives();
        // Вывести сведения об устройствах.
        foreach(DriveInfo d in myDrives)
        {
            Console.WriteLine("Name: {0}", d.Name);           // имя
            Console.WriteLine("Type: {0}", d.DriveType);       // тип
            // Проверить, смонтировано ли устройство.
            if (d.IsReady)
            {
                Console.WriteLine("Free space: {0}", d.TotalFreeSpace);
                // Свободное пространство
                Console.WriteLine("Format: {0}", d.DriveFormat); // формат
                Console.WriteLine("Label: {0}", d.VolumeLabel);  // метка
            }
            Console.WriteLine();
        }
        Console.ReadLine();
    }
}
```

Ниже показан возможный вывод:

```
***** Fun with DriveInfo *****
```

```
Name: C:\
Type: Fixed
Free space: 587376394240
Format: NTFS
Label: Mongo Drive
```

```
Name: D:\
Type: CDROM
```

```
Name: E:\
Type: CDROM
```

```
Name: F:\
Type: CDROM
```

```
Name: H:\
Type: Fixed
Free space: 477467508736
Format: FAT32
Label: My Passport
```

К этому моменту вы ознакомились с некоторым базовым поведением классов `Directory`, `DirectoryInfo` и `DriveInfo`. Далее вы научитесь создавать, открывать, закрывать и удалять файлы, находящиеся в заданном каталоге.

Исходный код. Проект `DriveInfoApp` доступен в подкаталоге `Chapter 20`.

Работа с классом `FileInfo`

Как было показано в предыдущем примере `DirectoryApp`, класс `FileInfo` позволяет получать подробные сведения о существующих файлах на жестком диске (например, время создания, размер и атрибуты) и помогает создавать, копировать, перемещать и удалять файлы. В дополнение к набору функциональности, унаследованной от `FileSystemInfo`, класс `FileInfo` имеет ряд уникальных членов, которые описаны в табл. 20.4.

Таблица 20.4. Основные члены `FileInfo`

Член	Описание
<code>AppendText()</code>	Создает объект <code>StreamWriter</code> (описанный ниже) и добавляет текст в файл
<code>CopyTo()</code>	Копирует существующий файл в новый файл
<code>Create()</code>	Создает новый файл и возвращает объект <code>FileStream</code> (описанный ниже) для взаимодействия с вновь созданным файлом
<code>CreateText()</code>	Создает объект <code>StreamWriter</code> , записывающий новый текстовый файл
<code>Delete()</code>	Удаляет файл, к которому привязан экземпляр <code>FileInfo</code>
<code>Directory</code>	Получает экземпляр родительского каталога
<code>DirectoryName</code>	Получает полный путь к родительскому каталогу
<code>Length</code>	Получает размер текущего файла или каталога
<code>MoveTo()</code>	Перемещает указанный файл в новое местоположение, предоставляя возможность указания нового имени для файла
<code>Name</code>	Получает имя файла
<code>Open()</code>	Открывает файл с различными привилегиями чтения/записи и совместного доступа
<code>OpenRead()</code>	Создает доступный только для чтения объект <code>FileStream</code>
<code>OpenText()</code>	Создает объект <code>StreamReader</code> (описанный ниже) и читает из существующего текстового файла
<code>OpenWrite()</code>	Создает доступный только для записи объект <code>FileStream</code>

Обратите внимание, что большинство методов класса `FileInfo` возвращают специфический объект ввода-вывода (такой как `FileStream` и `StreamWriter`), который позволяет начать чтение и запись данных в ассоциированный файл в разнообразных форматах. Скоро мы рассмотрим эти типы; однако прежде чем увидеть работающий пример, давайте изучим различные способы получения дескриптора файла с использованием класса `FileInfo`.

Метод `FileInfo.Create()`

Один из способов создания дескриптора файла предусматривает применение метода `FileInfo.Create()`:

```
static void Main(string[] args)
{
    // Создать новый файл на диске C:.
    FileInfo f = new FileInfo(@"C:\Test.dat");
    FileStream fs = f.Create();

    // Использовать объект FileStream...

    // Закрыть файловый поток.
    fs.Close();
}
```

Метод `FileInfo.Create()` возвращает тип `FileStream`, который предоставляет синхронную и асинхронную операции записи/чтения лежащего в его основе файла. Имейте в виду, что объект `FileStream`, возвращаемый `FileInfo.Create()`, открывает полный доступ по чтению и записи всем пользователям.

Также обратите внимание, что после окончания работы с текущим объектом `FileStream` необходимо обеспечить закрытие его дескриптора для освобождения внутренних неуправляемых ресурсов потока. Учитывая, что `FileStream` реализует интерфейс `IDisposable`, можно применить контекст `using` и позволить компилятору сгенерировать логику завершения (подробности ищите в главе 8):

```
static void Main(string[] args)
{
    // Определение контекста using для файлового ввода-вывода.
    FileInfo f = new FileInfo(@"C:\Test.dat");
    using (FileStream fs = f.Create())
    {
        // Использовать объект FileStream...
    }
}
```

Метод `FileInfo.Open()`

С помощью метода `FileInfo.Open()` можно открывать существующие файлы, а также создавать новые файлы с гораздо более высокой точностью, чем `FileInfo.Create()`, учитывая, что `Open()` обычно принимает несколько параметров для описания общей структуры файла, с которым будет производиться работа. В результате вызова `Open()` получается возвращенный им объект `FileStream`. Взгляните на следующий код:

```
static void Main(string[] args)
{
    // Создать новый файл через FileInfo.Open().
    FileInfo f2 = new FileInfo(@"C:\Test2.dat");
    using (FileStream fs2 = f2.Open(FileMode.OpenOrCreate,
        FileAccess.ReadWrite, FileShare.None))
    {
        // Использовать объект FileStream...
    }
}
```

Эта версия перегруженного метода `Open()` требует трех параметров. Первый параметр указывает общий тип запроса ввода-вывода (например, создать новый файл, открыть существующий файл и дописать в файл), указываемый в виде перечисления `FileMode` (описание членов приведено в табл. 20.5):

```
public enum FileMode
{
    CreateNew,
    Create,
    Open,
    OpenOrCreate,
    Truncate,
    Append
}
```

Таблица 20.5. Члены перечисления FileMode

Член	Описание
CreateNew	Информирует операционную систему о создании нового файла. Если файл уже существует, генерируется исключение <code>IOException</code>
Create	Информирует операционную систему о создании нового файла. Если файл уже существует, он будет перезаписан
Open	Открывает существующий файл. Если файл не существует, генерируется исключение <code>FileNotFoundException</code>
OpenOrCreate	Открывает файл, если он существует; в противном случае создает новый
Truncate	Открывает файл и усекает его до нулевой длины
Append	Открывает файл, переходит в его конец и начинает операции записи (этот флаг может использоваться только с потоками, предназначенными лишь для записи). Если файл не существует, то создается новый

Второй параметр метода `Open()` — значение перечисления `FileAccess` — используется для определения поведения чтения/записи лежащего в основе потока:

```
public enum FileAccess
{
    Read,
    Write,
    ReadWrite
}
```

И, наконец, третий параметр метода `Open()` — `FileShare` — указывает, как файл может совместно использоваться другими файловыми дескрипторами. Ниже перечислены его возможные значения:

```
public enum FileShare
{
    Delete,
    Inheritable,
    None,
    Read,
    ReadWrite,
    Write
}
```

Методы `FileOpen.OpenRead()` и `FileInfo.OpenWrite()`

Хотя метод `FileOpen.Open()` позволяет получить дескриптор файла довольно гибким способом, в классе `FileInfo` также предусмотрены для этого члены `OpenRead()` и `OpenWrite()`. Как и можно было ожидать, эти методы возвращают объект `FileStream`, соответствующим образом сконфигурированный только для чтения или только для записи, без необходимости в указании различных значений перечисления.

Подобно `FileInfo.Create()` и `FileInfo.Open()`, методы `OpenRead()` и `OpenWrite()` возвращают объект `FileStream` (обратите внимание, что в следующем коде предполагается наличие файлов по имени `Test3.dat` и `Test4.dat` на диске C:).

```
static void Main(string[] args)
{
    // Получить объект FileStream с правами только для чтения.
    FileInfo f3 = new FileInfo(@"C:\Test3.dat");
    using(FileStream readOnlyStream = f3.OpenRead())
    {
        // Использовать объект FileStream...
    }
    // Теперь получить объект FileStream с правами только для записи.
    FileInfo f4 = new FileInfo(@"C:\Test4.dat");
    using(FileStream writeOnlyStream = f4.OpenWrite())
    {
        // Использовать объект FileStream...
    }
}
```

Метод `FileInfo.OpenText()`

Еще один член типа `FileInfo`, связанный с открытием файлов — `OpenText()`. В отличие от `Create()`, `Open()`, `OpenRead()` и `OpenWrite()`, метод `OpenText()` возвращает экземпляр типа `StreamReader`, а не `FileStream`. Исходя из того, что на диске C: уже есть файл по имени `boot.ini`, получить доступ к его содержимому можно следующим образом:

```
static void Main(string[] args)
{
    // Получить объект StreamReader.
    FileInfo f5 = new FileInfo(@"C:\boot.ini");
    using(StreamReader sreader = f5.OpenText())
    {
        // Использовать объект StreamReader...
    }
}
```

Как вскоре будет показано, тип `StreamReader` предоставляет способ чтения символьных данных из лежащего в основе файла.

Методы `FileInfo.CreateText()` и `FileInfo.AppendText()`

Последними двумя методами, представляющими интерес, являются `CreateText()` и `AppendText()`. Оба они возвращают объект `StreamWriter`, как показано ниже:

```
static void Main(string[] args)
{
    FileInfo f6 = new FileInfo(@"C:\Test6.txt");
    using(StreamWriter swriter = f6.CreateText())
    {
        // Использовать объект StreamWriter...
    }
    FileInfo f7 = new FileInfo(@"C:\FinalTest.txt");
    using(StreamWriter swriterAppend = f7.AppendText())
    {
        // Использовать объект StreamWriter...
    }
}
```

Как и можно было ожидать, тип `StreamWriter` предлагает способ записи данных в лежащий в основе файл.

Работа с типом File

Тип File предоставляет функциональность, почти идентичную типу FileInfo, с помощью нескольких статических методов. Подобно FileInfo, тип File поддерживает методы AppendText(), Create(), CreateText(), Open(), OpenRead(), OpenWrite() и OpenText(). Фактически во многих случаях типы File и FileInfo могут использоваться взаимозаменяемым образом. В целях иллюстрации каждый из предшествующих примеров применения FileStream можно упростить, используя вместо него тип File:

```
static void Main(string[] args)
{
    // Получить объект FileStream через File.Create().
    using(FileStream fs = File.Create(@"C:\Test.dat"))
    {}
    // Получить объект FileStream через File.Open().
    using(FileStream fs2 = File.Open(@"C:\Test2.dat",
        FileMode.OpenOrCreate,
        FileAccess.ReadWrite, FileShare.None))
    {}
    // Получить объект FileStream с правами только для чтения.
    using(FileStream readOnlyStream = File.OpenRead(@"Test3.dat"))
    {}
    // Получить объект FileStream с правами только для записи.
    using(FileStream writeOnlyStream = File.OpenWrite(@"Test4.dat"))
    {}
    // Получить объект StreamReader.
    using(StreamReader sreader = File.OpenText(@"C:\boot.ini"))
    {}
    // Получить несколько объектов StreamWriter.
    using(StreamWriter swriter = File.CreateText(@"C:\Test6.txt"))
    {}
    using(StreamWriter swriterAppend = File.AppendText(@"C:\FinalTest.txt"))
    {}
}
```

Дополнительные члены File

Тип File также поддерживает несколько членов, перечисленных в табл. 20.6, которые могут значительно упростить процесс чтения и записи текстовых данных.

Таблица 20.6. Методы типа File

Метод	Описание
ReadAllBytes()	Открывает указанный файл, возвращает двоичные данные в виде массива байт и затем закрывает файл
ReadAllLines()	Открывает указанный файл, возвращает символьные данные в виде массива строк, затем закрывает файл
ReadAllText()	Открывает указанный файл, возвращает символьные данные в виде System.String(), затем закрывает файл
WriteAllBytes()	Открывает указанный файл, записывает в него массив байтов и закрывает файл
WriteAllLines()	Открывает указанный файл, записывает в него массив строк и закрывает файл
WriteAllText()	Открывает указанный файл, записывает в него данные из указанной строки и закрывает файл

Используя эти методы типа `File`, можно осуществлять чтение и запись пакетов данных с помощью всего нескольких строк кода. Еще лучше то, что эти методы автоматически закрывают лежащий в основе файловый дескриптор. Например, следующая консольная программа (по имени `SimpleFileIO`) сохраняет строковые данные в новом файле на диске `C:` (и читает их в память) с минимальными усилиями (предполагается, что было импортировано пространство имен `System.IO`):

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Simple I/O with the File Type *****\n");
        string[] myTasks = {
            "Fix bathroom sink", "Call Dave",
            "Call Mom and Dad", "Play Xbox 360"};

        // Записать все данные в файл на диске C:.
        File.WriteAllLines(@"C:\tasks.txt", myTasks);

        // Прочитать все данные и вывести на консоль.
        foreach (string task in File.ReadAllLines(@"C:\tasks.txt"))
        {
            Console.WriteLine("TODO: {0}", task);
        }
        Console.ReadLine();
    }
}
```

Отсюда вывод: когда необходимо быстро получить файловый дескриптор, тип `File` позволит сэкономить на объеме кодирования. Однако преимущество предварительного создания объекта `FileInfo` связано с возможностью исследования файла с использованием членов абстрактного базового класса `FileSystemInfo`.

Исходный код. Проект `SimpleFileIO` доступен в подкаталоге `Chapter 20`.

Абстрактный класс `Stream`

К этому моменту было показано несколько способов получения объектов `FileStream`, `StreamReader` и `StreamWriter`, но еще нужно читать данные и записывать их в файл, использующий эти типы. Чтобы понять, как это делается, необходимо изучить концепцию потока. В мире манипуляций вводом-выводом *поток* (stream) представляет порцию данных, протекающую от источника к цели. Потоки предоставляют общий способ взаимодействия с *последовательностью байтов*, независимо от того, какого рода устройство (файл, сеть, соединение, принтер и т.п.) хранит или отображает эти байты.

В абстрактном классе `System.IO.Stream` определен набор членов, которые обеспечивают поддержку синхронного и асинхронного взаимодействия с хранилищем (например, файлом или областью памяти).

На заметку! Концепция потока не ограничена файловым вводом-выводом. Разумеется, библиотеки `.NET` предоставляют потоковый доступ к сетям, областям памяти и прочим абстракциям, связанным с потоками.

Потомки класса `Stream` представляют данные как низкоуровневые потоки байт, а непосредственная работа с низкоуровневыми потоками может оказаться довольно загадочной. Некоторые типы, унаследованные от `Stream`, поддерживают поиск, что озна-

чает возможность получения и изменения текущей позиции в потоке. Чтобы приблизиться к пониманию функциональности класса `Stream`, рассмотрим список основных его членов, приведенный в табл. 20.7.

Таблица 20.7. Члены абстрактного класса `Stream`

Член	Описание
<code>CanRead</code> <code>CanWrite</code> <code>CanSeek</code>	Определяют, поддерживает ли текущий поток чтение, поиск и/или запись
<code>Close()</code>	Закрывает текущий поток и освобождает все ресурсы (такие как сокеты и файловые дескрипторы), ассоциированные с текущим потоком. Внутренне этот метод является псевдонимом <code>Dispose()</code> . Поэтому закрытие потока функционально эквивалентно освобождению потока
<code>Flush()</code>	Обновляет лежащий в основе источник данных или репозиторий текущим состоянием буфера с последующей очисткой буфера. Если поток не реализует буфер, метод ничего не делает
<code>Length</code>	Возвращает длину потока в байтах
<code>Position</code>	Определяет текущую позицию в потоке
<code>Read()</code> <code>ReadByte()</code>	Читает последовательность байт (или одиночный байт) из текущего потока и перемещает текущую позицию потока вперед на количество прочитанных байтов
<code>Seek()</code>	Устанавливает позицию в текущем потоке
<code>SetLength()</code>	Устанавливает длину текущего потока
<code>Write()</code> <code>WriteByte()</code>	Записывает последовательность байтов (или одиночный байт) в текущий поток и перемещает текущую позицию вперед на количество записанных байтов

Работа с классом `FileStream`

Класс `FileStream` предоставляет реализацию абстрактного члена `Stream` в манере, подходящей для потоковой работы с файлами. Это элементарный поток, и он может записывать или читать только один байт или массив байтов. Тем не менее, взаимодействовать с членами типа `FileStream` придется нечасто. Вместо этого, скорее всего, будут использоваться разнообразные оболочки потоков, которые облегчают работу с текстовыми данными или типами .NET. Однако в целях иллюстрации полезно поэкспериментировать с возможностями синхронного чтения/записи типа `FileStream`.

Предположим, что имеется консольное приложение под названием `FileStreamApp` (и в файле кода C# выполнен импорт пространств имен `System.IO` и `System.Text`). Цель заключается в записи простого текстового сообщения в новый файл по имени `myMessage.dat`. Однако, учитывая, что `FileStream` может работать только с низкоуровневыми байтами, придется закодировать тип `System.String` в соответствующий байтовый массив. К счастью, в пространстве имен `System.Text` определен тип по имени `Encoding`, который содержит члены, способные кодировать и декодировать строки в массивы байтов (подробное описание типа `Encoding` ищите в документации .NET Framework 4.5 SDK).

После кодирования байтовый массив сохраняется в файле с помощью метода `FileStream.Write()`. Чтобы прочитать байты обратно в память, необходимо сбросить внутреннюю позицию потока (через свойство `Position`) и вызвать метод `ReadByte()`. И, наконец, на консоль выводится содержимое низкоуровневого байтового массива и декодированная строка. Ниже приведен полный код метода `Main()`.

```
// Не забудьте импортировать пространства имен System.Text и System.IO.
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with FileStreams *****\n");

    // Получить объект FileStream.
    using(FileStream fStream = File.Open(@"C:\myMessage.dat",
        FileMode.Create))
    {
        // Закодировать строку в виде массива байт.
        string msg = "Hello!";
        byte[] msgAsByteArray = Encoding.Default.GetBytes(msg);

        // Записать byte[] в файл.
        fStream.Write(msgAsByteArray, 0, msgAsByteArray.Length);

        // Сбросить внутреннюю позицию потока.
        fStream.Position = 0;

        // Прочитать типы из файла и вывести на консоль.
        Console.Write("Your message as an array of bytes: ");
        byte[] bytesFromFile = new byte[msgAsByteArray.Length];
        for (int i = 0; i < msgAsByteArray.Length; i++)
        {
            bytesFromFile[i] = (byte)fStream.ReadByte();
            Console.Write(bytesFromFile[i]);
        }

        // Вывести декодированные сообщения.
        Console.Write("\nDecoded Message: ");
        Console.WriteLine(Encoding.Default.GetString(bytesFromFile));
    }
    Console.ReadLine();
}
```

В этом примере производится не только наполнение файла данными, но также демонстрируется основной недостаток прямой работы с типом `FileStream`: приходится оперировать низкоуровневыми байтами. Другие унаследованные от `Stream` типы работают аналогично. Например, чтобы записать последовательность байт в область памяти, необходимо создать объект `MemoryStream`. Аналогично, для передачи массива байтов по сетевому соединению используется класс `NetworkStream` (из пространства имен `System.Net.Sockets`).

Как уже упоминалось, в пространстве имен `System.IO` доступно множество типов для чтения и записи, которые инкапсулируют детали работы с типами, производными от `Stream`.

Исходный код. Проект `FileStreamApp` доступен в подкаталоге `Chapter 20`.

Работа с классами `StreamWriter` и `StreamReader`

Классы `StreamWriter` и `StreamReader` удобны во всех случаях, когда нужно читать или записывать символьные данные (например, строки). Оба типа работают по умолчанию с символами `Unicode`; однако это можно изменить предоставлением правильно сконфигурированной ссылки на объект `System.Text.Encoding`. Чтобы не усложнять пример, предположим, что стандартная кодировка `Unicode` вполне устраивает.

Класс `StreamReader`, как и `StringReader` (о котором речь пойдет далее в этой главе), унаследован от абстрактного класса по имени `TextReader`. Базовый класс предлагает очень ограниченный набор функциональности каждому из своих наследников, в частности — возможность читать и “заглядывать” (`peek`) в символьный поток.

Класс `StreamWriter` (а также и `StringWriter`, который рассматривается далее в главе) порожден от абстрактного базового класса по имени `TextWriter`. В этом классе определены члены, позволяющие производным типам записывать текстовые данные в заданный символьный поток.

Чтобы приблизить вас к пониманию основных возможностей записи классов `StreamWriter` и `StringWriter`, в табл. 20.8 представлены описания основных членов абстрактного базового класса `TextWriter`.

Таблица 20.8. Основные члены `TextWriter`

Член	Описание
<code>Close()</code>	Этот метод закрывает средство записи и освобождает все связанные с ним ресурсы. В процессе автоматически сбрасывается буфер (опять-таки, этот член функционально эквивалентен методу <code>Dispose()</code>)
<code>Flush()</code>	Этот метод очищает все буферы текущего средства записи и записывает все буферизованные данные на лежащее в основе устройство, но не закрывает его
<code>NewLine</code>	Это свойство задает константу новой строки для унаследованного класса средства записи. По умолчанию ограничителем строки в Windows является возврат каретки, за которым следует перевод строки (<code>\r\n</code>)
<code>Write()</code>	Этот перегруженный метод записывает данные в текстовый поток без добавления константы новой строки
<code>WriteLine()</code>	Этот перегруженный метод записывает данные в текстовый поток с добавлением константы новой строки

На заметку! Последние два члена класса `TextWriter`, скорее всего, покажутся знакомыми. Если помните, тип `System.Console` имеет члены `Write()` и `WriteLine()`, которые выталкивают текстовые данные на стандартное устройство вывода. Фактически свойство `Console.In` хранит `TextWriter`, а `Console.Out` — `TextWriter`.

Унаследованный класс `StreamWriter` предоставляет соответствующую реализацию методов `Write()`, `Close()` и `Flush()`, а также определяет дополнительное свойство `AutoFlush`. Когда это свойство установлено в `true`, оно заставляет `StreamWriter` выталкивать данные при каждой операции записи. Следует отметить, что можно обеспечить более высокую производительность, установив `AutoFlush` в `false`, но при этом всегда вызывать `Close()` по завершении работы с `StreamWriter`.

Запись в текстовый файл

Чтобы увидеть класс `StreamWriter` в действии, создадим новое консольное приложение по имени `StreamWriterReaderApp` и импортируем пространство имен `System.IO`. В показанном ниже методе `Main()` создается новый файл по имени `reminders.txt` с помощью метода `File.CreateText()`. Используя полученный объект `StreamWriter`, в новый файл будут добавлены некоторые текстовые данные.

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with StreamWriter / StreamReader *****\n");
```

```
// Получить StreamWriter и записать строковые данные.
using(StreamWriter writer = File.CreateText("reminders.txt"))
{
    writer.WriteLine("Don't forget Mother's Day this year...");
    writer.WriteLine("Don't forget Father's Day this year...");
    writer.WriteLine("Don't forget these numbers:");
    for(int i = 0; i < 10; i++)
        writer.Write(i + " ");

    // Вставить новую строку.
    writer.Write(writer.NewLine);
}
Console.WriteLine("Created file and wrote some thoughts...");
Console.ReadLine();
}
```

После выполнения этой программы можно просмотреть содержимое созданного файла (рис. 20.3). Этот файл находится в папке bin\Debug текущего приложения, поскольку при вызове `CreateText()` абсолютный путь не указывался.

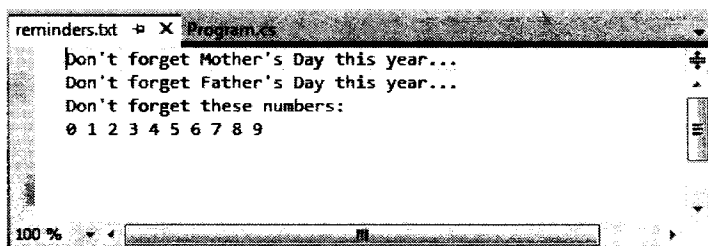


Рис. 20.3. Содержимое созданного текстового файла

Чтение из текстового файла

Теперь давайте посмотрим, как программно прочитать данные из файла, используя соответствующий тип `StreamReader`. Как вы помните, этот класс унаследован от абстрактного класса `TextReader`, который обеспечивает функциональность, описанную в табл. 20.9.

Таблица 20.9. Основные члены `TextReader`

Член	Описание
<code>Peek()</code>	Возвращает следующий доступный символ, не изменяя текущей позиции средства чтения. Значение <code>-1</code> указывает на достижение конца потока
<code>Read()</code>	Читает данные из входного потока
<code>ReadBlock()</code>	Читает указанное максимальное количество символов из текущего потока и записывает данные в буфер, начиная с заданного индекса
<code>ReadLine()</code>	Читает строку символов из текущего потока и возвращает данные в виде строки (строка <code>null</code> указывает на признак конца файла)
<code>ReadToEnd()</code>	Читает все символы от текущей позиции до конца потока и возвращает их в виде одной строки

Если теперь расширить текущий пример приложения, чтобы в нем применялся класс `StreamReader`, то можно будет прочитать текстовые данные из файла `reminders.txt`:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with StreamWriter / StreamReader *****\n");
    ...

    // Прочитать данные из файла.
    Console.WriteLine("Here are your thoughts:\n");
    using(StreamReader sr = File.OpenText("reminders.txt"))
    {
        string input = null;
        while ((input = sr.ReadLine()) != null)
        {
            Console.WriteLine (input);
        }
    }
    Console.ReadLine();
}
```

После запуска этой программы на консоли отображаются символьные данные из файла reminders.txt.

Прямое создание экземпляров классов StreamWriter/StreamReader

Одним из сбивающих с толку аспектов работы с типами, которые входят в пространство имен System.IO, является то, что одного и того же результата часто можно достичь с использованием разных подходов. Например, ранее уже было показано, что с помощью метода CreateText() можно получить объект StreamWriter с типом File или FileInfo. В действительности есть и еще один способ работы с объектами StreamWriter и StreamReader: создавать их напрямую. Например, текущее приложение можно было бы переделать следующим образом:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with StreamWriter / StreamReader *****\n");

    // Получить StreamWriter и записать строковые данные.
    using(StreamWriter writer = new StreamWriter("reminders.txt"))
    {
        ...
    }

    // Прочитать данные из файла.
    using(StreamReader sr = new StreamReader("reminders.txt"))
    {
        ...
    }
}
```

Несмотря на то что существование такого количества на первый взгляд идентичных подходов к файловому вводу-выводу может обескуражить, следует иметь в виду, что в результате достигается более высокая гибкость. Теперь, когда известно, как перемещать символьные данные в файл и из файла с использованием классов StreamWriter и StreamReader, рассмотрим роль классов StringWriter и StringReader.

Работа с классами `StringWriter` и `StringReader`

Классы `StringWriter` и `StringReader` можно использовать для трактовки текстовой информации как потока символов, находящихся в памяти. Это полезно, когда требуется добавить символьную информацию к лежащему в основе буферу. Для иллюстрации в следующем консольном приложении (по имени `StringReaderWriterApp`) блок строковых данных записывается в объект `StringWriter` вместо файла на локальном жестком диске:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with StringWriter / StringReader *****\n");
    // Создать StringWriter и записать символьные данные в память.
    using (StringWriter strWriter = new StringWriter())
    {
        strWriter.WriteLine("Don't forget Mother's Day this year...");
        // Получить копию содержимого (хранящегося в строке)
        // и вывести на консоль.
        Console.WriteLine("Contents of StringWriter:\n{0}", strWriter);
    }
    Console.ReadLine();
}
```

Поскольку и `StringWriter`, и `StreamWriter` порождены от одного и того же базового класса (`TextWriter`), логика записи в какой-то мере похожа. Однако, учитывая природу `StringWriter`, имейте в виду, что этот класс позволяет использовать метод `GetStringBuilder()` для извлечения объекта `System.Text.StringBuilder`:

```
using (StringWriter strWriter = new StringWriter())
{
    strWriter.WriteLine("Don't forget Mother's Day this year...");
    Console.WriteLine("Contents of StringWriter:\n{0}", strWriter);

    // Получить внутренний StringBuilder.
    StringBuilder sb = strWriter.GetStringBuilder();
    sb.Insert(0, "Hey!! ");
    Console.WriteLine("-> {0}", sb.ToString());
    sb.Remove(0, "Hey!! ".Length);
    Console.WriteLine("-> {0}", sb.ToString());
}
```

Когда необходимо выполнить чтение из потока строковых данных, используйте соответствующий тип `StringReader`, который (как и можно было ожидать) функционирует идентично классу `StreamReader`. На самом деле в классе `StringReader` всего лишь переопределяются унаследованные члены для чтения из блока символьных данных, а не из файла:

```
using (StringWriter strWriter = new StringWriter())
{
    strWriter.WriteLine("Don't forget Mother's Day this year...");
    Console.WriteLine("Contents of StringWriter:\n{0}", strWriter);
    // Читать данные из StringWriter.
    using (StringReader strReader = new StringReader(strWriter.ToString()))
    {
        string input = null;
        while ((input = strReader.ReadLine()) != null)
        {
            Console.WriteLine(input);
        }
    }
}
```


Исходный код. Проект StringReaderWriterApp доступен в подкаталоге Chapter 20.

Работа с классами BinaryWriter и BinaryReader

И последний набор классов средств чтения и записи, которые рассматриваются в настоящем разделе — это BinaryWriter и BinaryReader; оба они являются прямыми наследниками System.Object. Эти типы позволяют читать и записывать дискретные типы данных в потоки в компактном двоичном формате. В классе BinaryWriter определен многократно перегруженный метод Write() для помещения типов данных в лежащий в основе поток. В дополнение к Write(), класс BinaryWriter предоставляет дополнительные члены, позволяющие получать или устанавливать объекты унаследованных от Stream типов, а также поддерживает произвольный доступ к данным (табл. 20.10).

Таблица 20.10. Основные члены BinaryWriter

Член	Описание
BaseStream	Это свойство, предназначенное только для чтения, обеспечивает доступ к лежащему в основе потоку, используемому объектом BinaryWriter
Close()	Этот метод закрывает двоичный поток
Flush()	Этот метод выталкивает буфер двоичного потока
Seek()	Этот метод устанавливает позицию в текущем потоке
Write()	Этот метод записывает значение в текущий поток

Основные члены класса BinaryReader перечислены в табл. 20.11.

Таблица 20.11. Основные члены BinaryReader

Член	Описание
BaseStream	Это свойство, предназначенное только для чтения, обеспечивает доступ к лежащему в основе потоку, используемому объектом BinaryReader
Close()	Этот метод закрывает двоичный поток
PeekChar()	Этот метод возвращает следующий доступный символ без перемещения текущей позиции потока
Read()	Этот метод читает заданный набор байт или символов и сохраняет их в переданном ему массиве
ReadXXXX()	В классе BinaryReader определено множество методов чтения, которые извлекают из потока объекты различных типов (ReadBoolean(), ReadByte(), ReadInt32() и т.д.)

В следующем примере (консольное приложение по имени BinaryWriterReader) объекты данных разных типов записываются в файл *.dat:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Binary Writers / Readers *****\n");
    // Открыть средство двоичной записи в файл.
    FileInfo f = new FileInfo("BinFile.dat");
```

```

using(BinaryWriter bw = new BinaryWriter(f.OpenWrite()))
{
    // Вывести на консоль тип BaseStream
    // (System.IO.FileStream в данном случае).
    Console.WriteLine("Base stream is: {0}", bw.BaseStream);

    // Создать некоторые данные для сохранения в файле.
    double aDouble = 1234.67;
    int anInt = 34567;
    string aString = "A, B, C";

    // Записать данные.
    bw.Write(aDouble);
    bw.Write(anInt);
    bw.Write(aString);
}
Console.WriteLine("Done!");
Console.ReadLine();
}

```

Обратите внимание, что объект `FileStream`, возвращенный методом `FileInfo.OpenWrite()`, передается конструктору типа `BinaryWriter`. Используя эту технику, очень просто организовать по уровням поток перед записью данных. Имейте в виду, что конструктор `BinaryWriter` принимает любой тип, унаследованный от `Stream` (например, `FileStream`, `MemoryStream` или `BufferedStream`). Таким образом, если необходимо записать двоичные данные в память, просто используйте объект `MemoryStream`.

Для чтения данных из файла `BinFile.dat` в классе `BinaryReader` предлагается ряд опций. Например, ниже будут вызываться различные члены, выполняющие чтение, для извлечения каждого фрагмента данных из файлового потока:

```

static void Main(string[] args)
{
    ...
    FileInfo f = new FileInfo("BinFile.dat");
    ...

    // Читать двоичные данные из потока.
    using(BinaryReader br = new BinaryReader(f.OpenRead()))
    {
        Console.WriteLine(br.ReadDouble());
        Console.WriteLine(br.ReadInt32());
        Console.WriteLine(br.ReadString());
    }
    Console.ReadLine();
}

```

Исходный код. Проект `BinaryWriterReader` доступен в подкаталоге `Chapter 20`.

Программное отслеживание файлов

Теперь, когда вы научились использовать различные классы для чтения и записи, следующая задача заключается в исследовании роли класса `FileSystemWatcher`. Этот тип полезен, когда требуется программно отслеживать состояние файлов в системе. В частности, с помощью `FileSystemWatcher` можно организовать мониторинг файлов на предмет любых действий, указанных в перечислении `System.IO.NotifyFilters` (за более подробной информацией об этом перечислении обращайтесь в документацию .NET Framework 4.5 SDK):

```
public enum NotifyFilters
{
    Attributes, CreationTime,
    DirectoryName, FileName,
    LastAccess, LastWrite,
    Security, Size,
}
```

Первый шаг, который необходимо предпринять при работе с типом `FileSystemWatcher` — это установить свойство `Path`, чтобы оно указывало имя (и местоположение) каталога, содержащего файлы, которые нужно отслеживать, а также свойство `Filter`, определяющее расширения отслеживаемых файлов.

В настоящий момент можно выбрать обработку событий `Changed`, `Created` и `Deleted` — все они работают в сочетании с делегатом `FileSystemEventHandler`. Этот делегат может вызывать любой метод, соответствующий следующей сигнатуре:

```
// Делегат FileSystemEventHandler должен указывать
// на метод, соответствующий следующей сигнатуре.
void MyNotificationHandler(object source, FileSystemEventArgs e)
```

Событие `Renamed` также может быть обработано делегатом типа `RenamedEventHandler`, который может вызывать методы с такой сигнатурой:

```
// Делегат RenamedEventHandler должен указывать
// на метод, соответствующий следующей сигнатуре.
void MyNotificationHandler(object source, RenamedEventArgs e)
```

Хотя для обработки каждого события можно было бы применить традиционный синтаксис делегат/событие, вы, несомненно, воспользуетесь синтаксисом лямбда-выражений (как это сделано в загружаемом коде этого проекта).

Чтобы проиллюстрировать процесс мониторинга файлов, предположим, что на диске `C:` создан новый каталог по имени `MyFolder`, содержащий различные файлы `*.txt` (с произвольными именами). Следующее консольное приложение (под названием `MyDirectoryWatcher`) будет выполнять мониторинг файлов `*.txt` внутри каталога `MyFolder` и выводить на консоль сообщения при создании, удалении, модификации или переименовании файлов:

```
static void Main(string[] args)
{
    Console.WriteLine("***** The Amazing File Watcher App *****\n");

    // Установить путь к каталогу, за которым нужно наблюдать.
    FileSystemWatcher watcher = new FileSystemWatcher();
    try
    {
        watcher.Path = @"C:\MyFolder";
    }
    catch (ArgumentException ex)
    {
        Console.WriteLine(ex.Message);
        return;
    }

    // Указать цели наблюдения.
    watcher.NotifyFilter = NotifyFilters.LastAccess
        | NotifyFilters.LastWrite
        | NotifyFilters.FileName
        | NotifyFilters.DirectoryName;

    // Следить только за текстовыми файлами.
    watcher.Filter = "*.txt";
```

```
// Добавить обработчики событий.
watcher.Changed += new FileSystemEventHandler(OnChanged);
watcher.Created += new FileSystemEventHandler(OnChanged);
watcher.Deleted += new FileSystemEventHandler(OnChanged);
watcher.Renamed += new RenamedEventHandler(OnRenamed);

// Начать наблюдение за каталогом.
watcher.EnableRaisingEvents = true;

// Ожидать команды пользователя на завершение программы.
Console.WriteLine(@"Press 'q' to quit app.");
while(Console.Read() != 'q');
}
```

Следующие два обработчика событий просто сообщают о модификациях файлов:

```
static void OnChanged(object source, FileSystemEventArgs e)
{
    // Показать, что сделано, если файл изменен, создан или удален.
    Console.WriteLine("File: {0} {1}!", e.FullPath, e.ChangeType);
}

static void OnRenamed(object source, RenamedEventArgs e)
{
    // Показать, что файл был переименован.
    Console.WriteLine("File: {0} renamed to {1}", e.OldFullPath, e.FullPath);
}
```

Чтобы протестировать эту программу, запустите ее и откройте проводник Windows. Попробуйте переименовать файлы, создать файл *.txt, удалить файл *.txt и т.д. При этом вы увидите информацию относительно состояния текстовых файлов внутри MyFolder, как показано ниже:

```
***** The Amazing File Watcher App *****

Press 'q' to quit app.
File: C:\MyFolder\New Text Document.txt Created!
File: C:\MyFolder\New Text Document.txt renamed to C:\MyFolder\Hello.txt
File: C:\MyFolder\Hello.txt Changed!
File: C:\MyFolder\Hello.txt Changed!
File: C:\MyFolder\Hello.txt Deleted!
```

Исходный код. Проект MyDirectoryWatcher доступен в подкаталоге Chapter 20.

На этом знакомство с фундаментальными операциями ввода-вывода, предлагаемыми платформой .NET, завершено. Эти приемы определенно будут использоваться во многих приложениях. Кроме того, значительно упростить задачу сохранения больших объемов данных могут службы *сериализации объектов*.

Понятие сериализации объектов

Термин *сериализация* описывает процесс сохранения (и, возможно, передачи) состояния объекта в потоке (например, файловом потоке и потоке в памяти). Последовательность сохраняемых данных содержит всю информацию, необходимую для реконструкции (или *десериализации*) состояния объекта с целью последующего использования. Применяя эту технологию, очень просто сохранять большие объемы данных (в различных форматах) с минимальными усилиями. Во многих случаях сохранение данных приложения с использованием служб сериализации выливается в код меньшего объема, чем применение классов для чтения/записи из пространства имен System.IO.

Например, предположим, что требуется создать настольное приложение с графическим пользовательским интерфейсом, которое должно предоставлять конечным пользователям возможность сохранения их предпочтений (цвета окон, размер шрифта и т.п.). Для этого можно определить класс по имени `UserPrefs` и инкапсулировать в нем около двадцати полей данных. В случае применения типа `System.IO.BinaryWriter` придется *вручную* сохранять каждое поле объекта `UserPrefs`. Аналогично, когда вы понадобится загрузить данные из файла обратно в память, нужно будет использовать `System.IO.BinaryReader` и, опять-таки, *вручную* читать каждое значение, чтобы реконструировать новый объект `UserPrefs`.

Сэкономить значительное время можно, снабдив класс `UserPrefs` атрибутом `[Serializable]`, как показано ниже:

```
[Serializable]
public class UserPrefs
{
    public string WindowColor;
    public int FontSize;
}
```

После этого полное состояние объекта может быть сохранено с помощью всего нескольких строк кода. Не вдаваясь пока в детали, взгляните на следующий метод `Main()`:

```
static void Main(string[] args)
{
    UserPrefs userData = new UserPrefs();
    userData.WindowColor = "Yellow";
    userData.FontSize = 50;

    // BinaryFormatter сохраняет данные в двоичном формате.
    // Чтобы получить доступ к BinaryFormatter, понадобится
    // импортировать System.Runtime.Serialization.Formatters.Binary.
    BinaryFormatter binFormat = new BinaryFormatter();

    // Сохранить объект в локальном файле.
    using(Stream fStream = new FileStream("user.dat",
        FileMode.Create, FileAccess.Write, FileShare.None))
    {
        binFormat.Serialize(fStream, userData);
    }
    Console.ReadLine();
}
```

Хотя сохранять объекты с помощью механизма сериализации объектов .NET довольно просто, процесс, происходящий при этом “за кулисами”, достаточно сложен. Например, когда объект сохраняется в потоке, все ассоциированные с ним данные (т.е. данные базового класса и содержащиеся в нем объекты) также автоматически сериализуются. Поэтому при попытке сериализовать производный класс в игру вступают также все данные по цепочке наследования. И как будет показано, набор взаимосвязанных объектов, участвующих в этом, представляется с помощью графа объектов.

Службы сериализации .NET также позволяют сохранять граф объектов в различных форматах. В предыдущем примере кода применялся тип `BinaryFormatter`, поэтому состояние объекта `UserPrefs` сохраняется в компактном двоичном формате. Граф объектов можно также сохранить в формате SOAP или XML, применяя другие типы форматеров. Эти форматы полезны, когда необходимо гарантировать возможность передачи хранимых объектов между разными операционными системами, языками и архитектурами.

На заметку! В WCF предлагается слегка отличающийся механизм для сериализации объектов в и из операций служб WCF, в нем используются атрибуты `[DataContract]` и `[DataMember]`. Подробнее об этом речь пойдет в главе 25.

И, наконец, имейте в виду, что граф объектов может быть сохранен в любом типе, производном от `System.IO.Stream`. В предыдущем примере объект `UserPrefs` был сохранен в локальном файле через тип `FileStream`. Однако если вместо этого понадобится сохранить объект в определенной области памяти, можно применить тип `MemoryStream`. Главное, чтобы последовательность данных корректно представляла состояние объектов в графе.

Роль графов объектов

Как упоминалось ранее, когда объект сериализуется, среда CLR учитывает все связанные объекты, чтобы гарантировать корректное сохранение данных. Этот набор связанных объектов называется *графом объектов*. Графы объектов предоставляют простой способ документирования того, как между собой связаны элементы из набора. Следует отметить, что графы объектов *не* обозначают отношения “является” и “имеет” объектно-ориентированного программирования. Вместо этого стрелки в графе объектов можно читать как “требует” или “зависит от”.

Каждый объект в графе получает уникальное числовое значение. Имейте в виду, что числа, назначенные объектам в графе, являются произвольными и не имеют никакого значения для внешнего мира. После того как всем объектам назначены числовые значения, граф объектов может записывать набор зависимостей для каждого объекта.

В качестве простого примера предположим, что создан набор классов, моделирующих автомобили. Существует базовый класс по имени `Car`, который “имеет” класс `Radio`. Другой класс по имени `JamesBondCar` расширяет базовый тип `Car`. На рис. 20.4 показан возможный граф объектов, который моделирует эти отношения.

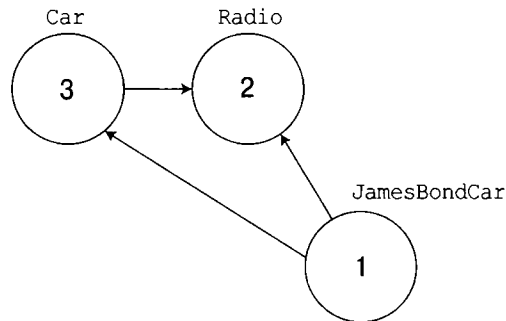


Рис. 20.4. Простой граф объектов

При чтении графов объектов для описания соединяющих стрелок можно использовать выражение “зависит от” или “ссылается на”. Таким образом, на рис. 20.1 видно, что класс `Car` ссылается на класс `Radio` (учитывая отношение “имеет”), `JamesBondCar` ссылается на `Car` (учитывая отношение “является”), а также на `Radio` (поскольку наследует эту защищенную переменную-член).

Конечно, среда CLR не рисует картинки в памяти для представления графа связанных объектов. Вместо этого отношение, документированное в предыдущей диаграмме, представляется формулой, которая выглядит примерно так:

```
[Car 3, ref 2], [Radio 2], [JamesBondCar 1, ref 3, ref 2]
```

Если вы проанализируете эту формулу, то увидите, что объект 3 (Car) имеет зависимость от объекта 2 (Radio). Объект 2 (Radio) — это “одинокий волк”, которому не нужен никто. И, наконец, объект 1 (JamesBondCar) имеет зависимость как от объекта 3, так и от объекта 2. В любом случае, при сериализации или десериализации экземпляра JamesBondCar граф объектов гарантирует, что типы Radio и Car также будут принимать участие в процессе.

Удобство процесса сериализации состоит в том, что граф, представляющий отношения между объектами, устанавливается автоматически, “за кулисами”. Как будет показано далее в этой главе, если необходимо вмешаться в конструирование графа объектов, это можно сделать посредством настройки процесса сериализации через атрибуты и интерфейсы.

На заметку! Строго говоря, тип XmlSerializer (описанный далее в главе) не сохраняет состояние с использованием графа объектов; тем не менее, он сериализует и десериализует связанные объекты в предсказуемой манере.

Конфигурирование объектов для сериализации

Чтобы сделать объект доступным для служб сериализации .NET, понадобится только декорировать каждый связанный класс (или структуру) атрибутом [Serializable]. Если выясняется, что какой-то тип имеет члены-данные, которые не должны (или не могут) участвовать в сериализации, можно пометить такие поля атрибутом [NonSerialized]. Это помогает сократить размер хранимых данных, при условии, что в сериализуемом классе есть переменные-члены, которые не следует “запоминать” (например, фиксированные значения, случайные значения или кратковременные данные).

Определение сериализуемых типов

Для начала создадим новое консольное приложение по имени SimpleSerialize. Добавим в него новый класс по имени Radio, помеченный атрибутом [Serializable], у которого исключается одна переменная-член (radioID), помеченная атрибутом [NonSerialized] и потому не сохраняемая в указанном потоке данных.

```
[Serializable]
public class Radio
{
    public bool hasTweeters;
    public bool hasSubWoofers;
    public double[] stationPresets;

    [NonSerialized]
    public string radioID = "XF-552RR6";
}
```

Затем добавим два дополнительных типа, представляющих классы JamesBondCar и Car (оба они также помечены атрибутом [Serializable]), и определим в них следующие поля данных:

```
[Serializable]
public class Car
{
    public Radio theRadio = new Radio();
    public bool isHatchBack;
}

[Serializable]
```

```
public class JamesBondCar : Car
{
    public bool canFly;
    public bool canSubmerge;
}
```

Имейте в виду, что атрибут `[Serializable]` не может наследоваться от родительского класса. Поэтому при наследовании типа, помеченного как `[Serializable]`, дочерний класс также должен быть помечен атрибутом `[Serializable]` или же его нельзя будет сохранить в потоке. В действительности все объекты в графе объектов должны быть помечены атрибутом `[Serializable]`. Попытка сериализовать несериализуемый объект с использованием `BinaryFormatter` или `SoapFormatter` приводит к генерации исключения `SerializationException` во время выполнения.

Открытые поля, закрытые поля и открытые свойства

Обратите внимание, что в каждом из этих классов поля данных определены как `public`, это сделано для упрощения примера. Конечно, закрытые данные, представленные открытыми свойствами, были бы более предпочтительными с точки зрения объектно-ориентированного программирования. Также для простоты в этих типах не определились никакие специальные конструкторы, и потому все неинициализированные поля данных получают ожидаемые стандартные значения.

Оставив в стороне принципы объектно-ориентированного программирования, может возникнуть вопрос: какого определения полей данных типа требуют различные форматы, чтобы сериализовать их в поток? Ответ такой: в зависимости от обстоятельств. Если вы сохраняете состояние объекта с применением `BinaryFormatter` или `SoapFormatter`, то разницы никакой. Эти типы запрограммированы для сериализации всех сериализуемых полей типа независимо от того, представлены они открытыми полями, закрытыми полями или закрытыми полями с соответствующими открытыми свойствами. Однако вспомните, что если есть элементы данных, которые не должны сохраняться в графе объектов, можно выборочно пометить открытые или закрытые поля атрибутом `[NonSerialized]`, как сделано со строковым полем в типе `Radio`.

Тем не менее, ситуация существенно меняется, если вы собираетесь использовать тип `XmlSerializer`. Этот тип будет сериализовать *только* открытые поля данных или закрытые поля, представленные открытыми свойствами. Закрытые данные, не представленные свойствами, будут игнорироваться. Например, рассмотрим следующий сериализуемый тип `Person`:

```
[Serializable]
public class Person
{
    // Открытое поле.
    public bool isAlive = true;

    // Закрытое поле.
    private int personAge = 21;

    // Открытое свойство/закрытые данные.
    private string fName = string.Empty;
    public string FirstName
    {
        get { return fName; }
        set { fName = value; }
    }
}
```


При обработке этого типа с помощью `BinaryFormatter` или `SoapFormatter` обнаружится, что поля `isAlive`, `personAge` и `fName` сохраняются в выбранном потоке. Однако `XmlSerializer` не сохранит значение `personAge`, поскольку эта часть закрытых данных не инкапсулирована в открытом свойстве. Чтобы сохранять `personAge` с помощью `XmlSerializer`, это поле понадобится определить как `public` или же инкапсулировать его в открытом свойстве.

Выбор формatera сериализации

После конфигурирования типов для участия в схеме сериализации .NET за счет применения необходимых атрибутов следующий шаг состоит в выборе формата (двоичного, SOAP или XML) для сохранения состояния объектов. Перечисленные возможности представлены следующими классами:

- `BinaryFormatter`
- `SoapFormatter`
- `XmlSerializer`

Тип `BinaryFormatter` сериализует состояние объекта в поток, используя компактный двоичный формат. Этот тип определен в пространстве имен `System.Runtime.Serialization.Formatters.Binary`, которое входит в сборку `mscorlib.dll`. Таким образом, чтобы получить доступ к этому типу, необходимо указать следующую директиву `using`:

```
// Получить доступ к BinaryFormatter в mscorlib.dll.
using System.Runtime.Serialization.Formatters.Binary;
```

Тип `SoapFormatter` сохраняет состояние объекта в виде сообщения SOAP (стандартный XML-формат для передачи и приема сообщений от веб-служб). Этот тип определен в пространстве имен `System.Runtime.Serialization.Formatters.Soap`, находящемся в отдельной сборке. Поэтому для форматирования графа объектов в сообщение SOAP необходимо сначала установить ссылку на `System.Runtime.Serialization.Formatters.Soap.dll`, используя диалоговое окно `Add Reference` (Добавить ссылку) в `Visual Studio` и затем указать следующую директиву `using`:

```
// Необходима ссылка на System.Runtime.Serialization.Formatters.Soap.dll!
using System.Runtime.Serialization.Formatters.Soap;
```

И, наконец, для сохранения дерева объектов в документе XML предусмотрен тип `XmlSerializer`. Чтобы использовать этот тип, нужно указать директиву `using` для пространства имен `System.Xml.Serialization` и установить ссылку на сборку `System.Xml.dll`. К счастью, шаблоны проектов `Visual Studio` автоматически ссылаются на `System.Xml.dll`, так что достаточно просто указать соответствующее пространство имен:

```
// Определено внутри System.Xml.dll.
using System.Xml.Serialization;
```

Интерфейсы `IFormatter` и `IRemotingFormatter`

Независимо от того, какой форматар выбран, имейте в виду, все они унаследованы непосредственно от `System.Object`, так что они не разделяют общий набор членов от какого-то базового класса сериализации. Однако типы `BinaryFormatter` и `SoapFormatter` поддерживают общие члены через реализацию интерфейсов `IFormatter` и `IRemotingFormatter` (как ни странно, `XmlSerializer` не реализует ни одного из них).

В интерфейсе `System.Runtime.Serialization.IFormatter` определены основные методы `Serialize()` и `Deserialize()`, которые выполняют черновую работу по перемещению графов объектов в определенный поток и обратно. Помимо этих членов в `IFormatter` определено несколько свойств, используемых “за кулисами” реализующим типом:

```
public interface IFormatter
{
    SerializationBinder Binder { get; set; }
    StreamingContext Context { get; set; }
    ISurrogateSelector SurrogateSelector { get; set; }
    object Deserialize(Stream serializationStream);
    void Serialize(Stream serializationStream, object graph);
}
```

Интерфейс `System.Runtime.Remoting.Messaging.IRemotingFormatter` (который внутренне используется уровнем удаленного взаимодействия .NET Remoting) перегружает члены `Serialize()` и `Deserialize()` в манере, более подходящей для распределенного сохранения. Обратите внимание, что интерфейс `IRemotingFormatter` унаследован от более общего интерфейса `IFormatter`:

```
public interface IRemotingFormatter : IFormatter
{
    object Deserialize(Stream serializationStream, HeaderHandler handler);
    void Serialize(Stream serializationStream, object graph, Header[] headers);
}
```

Хотя взаимодействовать с этими интерфейсами в большинстве сценариев сериализации не понадобится, вспомните, что полиморфизм на основе интерфейсов позволяет подставлять экземпляры `BinaryFormatter` или `SoapFormatter` там, где ожидается `IFormatter`. Таким образом, если необходимо построить метод, который может сериализовать граф объектов с применением любого из этих классов, можно записать так:

```
static void SerializeObjectGraph(IFormatter itfFormat,
                                Stream destStream, object graph)
{
    itfFormat.Serialize(destStream, graph);
}
```

Точность типов среди форматов

Наиболее очевидное отличие между тремя формateraми связано с тем, как граф объектов сохраняется в потоке (двоичном, SOAP или XML). Следует также знать о некоторых более тонких отличиях, в частности — каким образом формтеры добиваются *точности типов*. Когда используется тип `BinaryFormatter`, он сохраняет не только данные полей объектов из графа, но также полностью заданное имя каждого типа и полное имя определяющей его сборки (имя, версия, маркер открытого ключа и культура). Эти дополнительные элементы данных делают `BinaryFormatter` идеальным выбором, когда необходимо передавать объекты по значению (т.е. полные копии) между границами машин для использования в .NET-приложениях.

Форматер `SoapFormatter` сохраняет трассировки сборок-источников за счет применения пространства имен XML. Например, вспомните тип `Person`, определенный ранее в этой главе. Если понадобится сохранить этот тип в сообщении SOAP, вы обнаружите, что открывающий элемент `Person` снабжен сгенерированным параметром `xmlns`. Взгляните на следующее частичное определение, обратив особое внимание на пространство имен XML под названием `al`:

```
<a1:Person id="ref-1" xmlns:a1=
  "http://schemas.microsoft.com/clr/nsassem/SimpleSerialize/MyApp%2C%20
  Version%3D1.0.0.0%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull">
  <isAlive>true</isAlive>
  <personAge>21</personAge>
  <fName id="ref-3">Mel</fName>
</a1:Person>
```

Однако XmlSerializer не старается предохранить точную информацию о типе, поэтому не записывает его полностью заданное имя или сборку, в которой он определен. Хотя на первый взгляд это может показаться ограничением, причина состоит в открытой природе представления данных XML. Ниже показано возможное XML-представление типа Person:

```
<?xml version="1.0"?>
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <isAlive>true</isAlive>
  <PersonAge>21</PersonAge>
  <FirstName>Frank</FirstName>
</Person>
```

Если необходимо сохранить состояние объекта так, чтобы его можно было использовать в среде любой операционной системы (Windows, Mac OS X и различных дистрибутивах Linux), на любой платформе приложений (.NET, Java Enterprise Edition, COM и т.п.) или в любом языке программирования, придерживаться полной точности типов не следует, поскольку нельзя рассчитывать, что все возможные адресаты смогут понять специфичные для .NET типы данных. Учитывая это, SoapFormatter и XmlSerializer являются идеальным выбором, когда требуется гарантировать как можно более широкое распространение дерева объектов.

Сериализация объектов с использованием BinaryFormatter

Чтобы проиллюстрировать, насколько просто сохранять экземпляр JamesBondCar в физическом файле, воспользуемся типом BinaryFormatter. Двумя ключевыми методами типа BinaryFormatter, о которых следует знать, являются Serialize() и Deserialize():

1. Serialize() сохраняет граф объектов в указанный поток в виде последовательности байтов;
2. Deserialize() преобразует сохраненную последовательность байтов в граф объектов.

Предположим, что после создания экземпляра JamesBondCar и модификации некоторых данных состояния требуется сохранить этот экземпляр в файле *.dat. Начать следует с создания самого файла *.dat. Для этого можно создать экземпляр типа System.IO.FileStream. Затем понадобится создать экземпляр BinaryFormatter и передать ему FileStream и граф объектов для сохранения. Взгляните на следующий метод Main():

```
// Не забудьте импортировать пространства имен
// System.Runtime.Serialization.Formatters.Binary и System.IO!
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Object Serialization *****\n");
  // Создать JamesBondCar и установить состояние.
  JamesBondCar jbc = new JamesBondCar();
```

```

jbc.canFly = true;
jbc.canSubmerge = false;
jbc.theRadio.stationPresets = new double[]{89.3, 105.1, 97.1};
jbc.theRadio.hasTweeters = true;

// Сохранить объект в указанном файле в двоичном формате.
SaveAsBinaryFormat(jbc, "CarData.dat");
Console.ReadLine();
}

```

Метод `SaveAsBinaryFormat()` реализован следующим образом:

```

static void SaveAsBinaryFormat(object objGraph, string fileName)
{
    // Сохранить объект в файл CarData.dat в двоичном виде.
    BinaryFormatter binFormat = new BinaryFormatter();

    using(Stream fStream = new FileStream(fileName,
        FileMode.Create, FileAccess.Write, FileShare.None))
    {
        binFormat.Serialize(fStream, objGraph);
    }
    Console.WriteLine("=> Saved car in binary format!");
}

```

Как видите, метод `BinaryFormatter.Serialize()` — это член, отвечающий за построение графа объектов и передачу последовательности байтов в некоторый объект производного от `Stream` типа. В данном случае поток представляет физический файл. Однако сериализовать объекты можно также в любой тип, производный от `Stream`, такой как область памяти или сетевой поток.

После выполнения программы можно просмотреть содержимое файла `CarData.dat`, представляющее этот экземпляр `JamesBondCar`, в папке `bin\Debug` текущего проекта. На рис. 20.5 показано содержимое этого файла, открытого в Visual Studio.

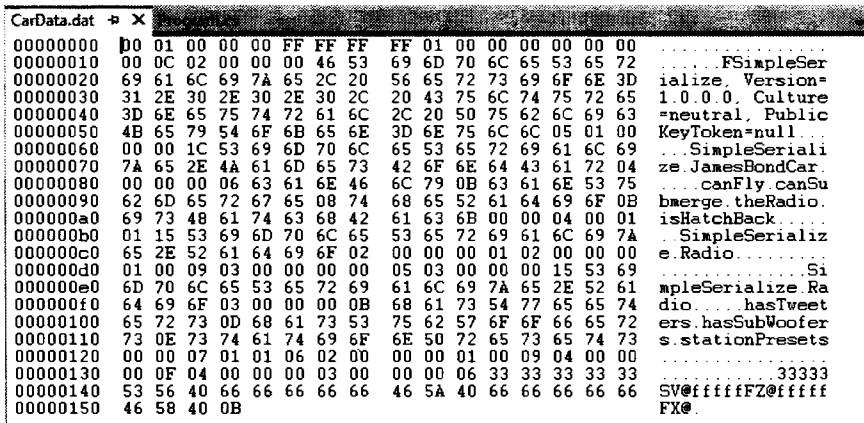


Рис. 20.5. Объект `JamesBondCar`, сериализованный с использованием `BinaryFormatter`

Десериализация объектов с использованием `BinaryFormatter`

Теперь предположим, что необходимо прочитать сохраненный объект `JamesBondCar` из двоичного файла обратно в объектную переменную. После открытия файла `CataData.dat` (посредством метода `File.OpenRead()`) можно вызвать метод `Deserialize()` класса `BinaryFormatter`. Имейте в виду, что `Deserialize()` возвращает объект общего типа `System.Object`, так что понадобится применить явное приведение, как показано ниже:

```
static void LoadFromBinaryFile(string fileName)
{
    BinaryFormatter binFormat = new BinaryFormatter();
    // Прочитать JamesBondCar из двоичного файла.
    using(Stream fStream = File.OpenRead(fileName))
    {
        JamesBondCar carFromDisk =
            (JamesBondCar)binFormat.Deserialize(fStream);
        Console.WriteLine("Can this car fly? : {0}", carFromDisk.canFly);
    }
}
```

Обратите внимание, что при вызове `Deserialize()` ему передается производный от `Stream` тип, представляющий местоположение сохраненного графа объектов. Приведя возвращенный объект к правильному типу, вы получаете объект в том состоянии, в каком он был на момент сохранения.

Сериализация объектов с использованием SoapFormatter

Следующий формater, которым мы воспользуемся, будет `SoapFormatter`, сериализующий данные в подходящем конверте SOAP. Протокол SOAP (Simple Object Access Protocol — простой протокол доступа к объектам) определяет стандартный процесс вызова методов в независимой от платформы и операционной системы манере.

Предполагая, что ссылка на сборку `System.Runtime.Serialization.Formatters.Soap.dll` установлена, а пространство имен `System.Runtime.Serialization.Formatters.Soap` импортировано, для сохранения и извлечения `JamesBondCar` в виде сообщения SOAP можно просто заменить в предыдущем примере все вхождения `BinaryFormatter` на `SoapFormatter`. Ниже показан новый метод класса `Program`, который сериализует объект в локальный файл в формате SOAP:

```
// Не забудьте импортировать пространства имен
// System.Runtime.Serialization.Formatters.Soap
// и установить ссылку на System.Runtime.Serialization.Formatters.Soap.dll!
static void SaveAsSoapFormat(object objGraph, string fileName)
{
    // Сохранить объект в файле CarData.soap в формате SOAP.
    SoapFormatter soapFormat = new SoapFormatter();

    using(Stream fStream = new FileStream(fileName,
        FileMode.Create, FileAccess.Write, FileShare.None))
    {
        soapFormat.Serialize(fStream, objGraph);
    }
    Console.WriteLine("=> Saved car in SOAP format!");
}
```

Как и ранее, для перемещения графа объектов в поток и обратно применяются методы `Serialize()` и `Deserialize()`. После вызова метода `SaveAsSoapFormat()` в `Main()` и запуска приложения можно открыть файл `*.soap`. В нем находятся XML-элементы, которые описывают значения состояния текущего объекта `JamesBondCar`, а также отношения между объектами в графе, представленные с помощью лексем `#ref` (рис. 20.6).

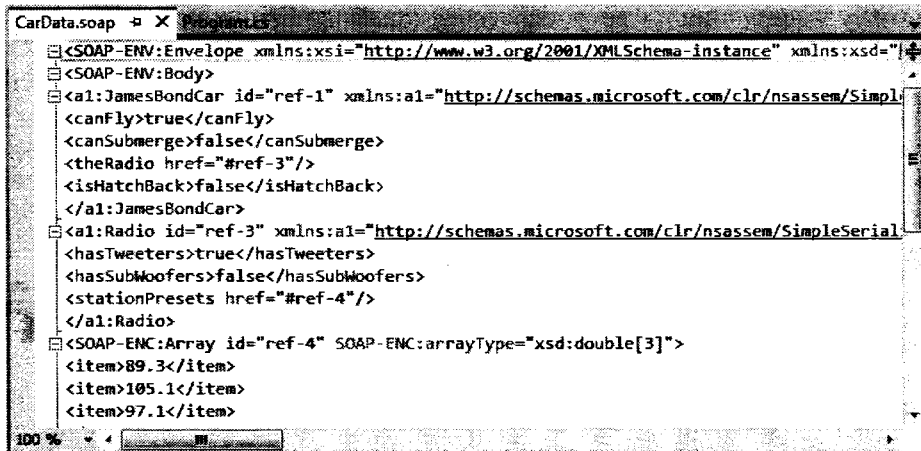


Рис. 20.6. Объект JamesBondCar, сериализованный с использованием SoapFormatter

Сериализация объектов с использованием XmlSerializer

В дополнение к двоичному формату и формату SOAP, сборка System.Xml.dll предлагает третий класс форматора — System.Xml.Serialization.XmlSerializer. Этот форматор может использоваться для сохранения *открытого* состояния заданного объекта в виде чистой XML-разметки, в противоположность данным XML внутри сообщения SOAP. Работа с этим типом несколько отличается от работы с типами SoapFormatter или BinaryFormatter. Рассмотрим следующий код (в нем предполагается, что было импортировано пространство имен System.Xml.Serialization):

```
static void SaveAsXmlFormat(object objGraph, string fileName)
{
    // Сохранить объект в файле CarData.xml в формате XML.
    XmlSerializer xmlFormat = new XmlSerializer(typeof(JamesBondCar),
        new Type[] { typeof(Radio), typeof(Car) });
    using(Stream fStream = new FileStream(fileName,
        FileMode.Create, FileAccess.Write, FileShare.None))
    {
        xmlFormat.Serialize(fStream, objGraph);
    }
    Console.WriteLine("=> Saved car in XML format!");
}
```

Ключевое отличие состоит в том, что тип XmlSerializer требует указания информации о типе, представляющей класс, который необходимо сериализовать. В сгенерированном файле XML находятся показанные ниже данные XML:

```
<?xml version="1.0"?>
<JamesBondCar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <theRadio>
    <hasTweeters>true</hasTweeters>
    <hasSubWoofers>false</hasSubWoofers>
    <stationPresets>
      <double>89.3</double>
      <double>105.1</double>
      <double>97.1</double>
```

```
</stationPresets>
<radioID>XF-552RR6</radioID>
</theRadio>
<isHatchBack>false</isHatchBack>
<canFly>true</canFly>
<canSubmerge>false</canSubmerge>
</JamesBondCar>
```

На заметку! Класс `XmlSerializer` требует, чтобы все сериализованные типы в графе объектов поддерживали стандартный конструктор (поэтому не забудьте его добавить, если определяли специальные конструкторы). Если этого не сделать, во время выполнения сгенерируется исключение `InvalidOperationException`.

Управление генерацией данных XML

Если у вас есть опыт работы с технологиями XML, то вы хорошо знаете, насколько важно удостоверяться, что данные внутри документа XML отвечают набору правил, которые устанавливают *действительность* данных. Понятие *действительного* документа XML не имеет отношения к синтаксической правильности элементов XML (наподобие того, что все открывающие элементы должны иметь соответствующие закрывающие элементы). Действительные документы — это те, что отвечают согласованным правилам форматирования (например, поле `X` должно быть выражено как атрибут, но не как подэлемент), которые обычно определены в схеме XML или файле определения типа документа (`Document-Type Definition — DTD`).

По умолчанию класс `XmlSerializer` сериализует все открытые поля/свойства как элементы XML, а не как атрибуты XML. Чтобы управлять генерацией результирующего документа XML с помощью класса `XmlSerializer`, необходимо декорировать типы любым количеством дополнительных атрибутов из пространства имен `System.Xml.Serialization`. В табл. 20.12 документированы некоторые (но не все) атрибуты, которые влияют на кодирование данных XML в потоке.

Таблица 20.12. Избранные атрибуты из пространства имен `System.Xml.Serialization`

Атрибут .NET	Описание
[XmlAttribute]	Этот атрибут .NET можно применять к полю или свойству для того, чтобы сообщить <code>XmlSerializer</code> о необходимости сериализовать данные как атрибут XML (а не как подэлемент)
[XmlElement]	Поле или свойство будет сериализовано как элемент XML с указанным именем
[XmlAttribute]	Этот атрибут предоставляет имя элемента, являющееся членом перечисления
[XmlRoot]	Этот атрибут управляет тем, как будет сконструирован корневой элемент (пространство имен и имя элемента)
[XmlText]	Свойство или поле будет сериализовано как текст XML (т.е. содержимое, находящееся между начальным и конечным дескрипторами корневого элемента)
[XmlType]	Этот атрибут предоставляет имя и пространство имен типа XML

В следующем простом примере показано текущее представление данных полей `JamesBondCar` в XML:

```
<?xml version="1.0" encoding="utf-8"?>
<JamesBondCar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  ...
  <canFly>true</canFly>
  <canSubmerge>false</canSubmerge>
</JamesBondCar>
```

Если необходимо указать специальное пространство имен XML, которое уточняет `JamesBondCar` и кодирует значения `canFly` и `canSubmerge` в виде атрибутов XML, модифицируем определение класса `JamesBondCar` следующим образом:

```
[Serializable, XmlRoot(Namespace = "http://www.MyCompany.com")]
public class JamesBondCar : Car
{
  [XmlAttribute]
  public bool canFly;
  [XmlAttribute]
  public bool canSubmerge;
}
```

Это выдает показанный ниже документ XML (обратите внимание на открывающий элемент `<JamesBondCar>`):

```
<?xml version="1.0" ""?>
<JamesBondCar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               canFly="true" canSubmerge="false"
               xmlns="http://www.MyCompany.com">
  ...
</JamesBondCar>
```

Естественно, для управления генерацией результирующего XML-документа с помощью `XmlSerializer` могут применяться многие другие атрибуты. За подробной информацией обращайтесь к описанию пространства имен `System.Xml.Serialization` в документации .NET Framework 4.5 SDK.

Сериализация коллекций объектов

Теперь, когда вы знаете, как сохранять единственный объект в потоке, давайте посмотрим, каким образом можно сохранить множество объектов. Как вы, возможно, заметили, метод `Serialize()` интерфейса `IFormatter` не предоставляет способа указать произвольное количество объектов в качестве ввода (допускается только единственный объект `System.Object`). Вдобавок возвращаемое значение `Deserialize()` также представляет собой одиночный объект `System.Object` (то же базовое ограничение касается и `XmlSerializer`):

```
public interface IFormatter
{
  ...
  object Deserialize(Stream serializationStream);
  void Serialize(Stream serializationStream, object graph);
}
```

Вспомните, что `System.Object` представляет целое дерево объектов. С учетом этого, если передать объект, который помечен атрибутом `[Serializable]` и содержит в себе другие объекты `[Serializable]`, то с помощью единственного вызова данного метода будет сохраняться весь набор объектов. К счастью, большинство типов из пространств имен `System.Collections` и `System.Collections.Generic` уже помечены атрибутом `[Serializable]`. Таким образом, чтобы сохранить множество объектов, просто добавь-

те это множество в контейнер (такой как обычный массив, `ArrayList` или `List<T>`) и сериализуйте данный объект в выбранный поток.

Предположим, что класс `JamesBondCar` дополнен конструктором, принимающим два аргумента, для установки нескольких фрагментов данных состояния (обратите внимание, что должен быть также добавлен стандартный конструктор, как того требует `XmlSerializer`):

```
[Serializable, XmlRoot(Namespace = "http://www.MyCompany.com")]
public class JamesBondCar : Car
{
    public JamesBondCar(bool skyWorthy, bool seaWorthy)
    {
        canFly = skyWorthy;
        canSubmerge = seaWorthy;
    }
    // XmlSerializer требует стандартного конструктора!
    public JamesBondCar(){}
    ...
}
```

Теперь можно сохранять любое количество объектов `JamesBondCar`, как показано ниже:

```
static void SaveListOfCars()
{
    // Сохранить список List<T> объектов JamesBondCar.
    List<JamesBondCar> myCars = new List<JamesBondCar>();
    myCars.Add(new JamesBondCar(true, true));
    myCars.Add(new JamesBondCar(true, false));
    myCars.Add(new JamesBondCar(false, true));
    myCars.Add(new JamesBondCar(false, false));
    using(Stream fStream = new FileStream("CarCollection.xml",
        FileMode.Create, FileAccess.Write, FileShare.None))
    {
        XmlSerializer xmlFormat = new XmlSerializer(typeof(List<JamesBondCar>));
        xmlFormat.Serialize(fStream, myCars);
    }
    Console.WriteLine("=> Saved list of cars!");
}
```

Поскольку здесь используется `XmlSerializer`, необходимо указать информацию о типе для каждого из подобъектов внутри корневого объекта (которым в данном случае является `List<JamesBondCar>`). Если бы применялись типы `BinaryFormatter` или `SoapFormatter`, то логика была бы еще проще. Например:

```
static void SaveListOfCarsAsBinary()
{
    // Сохранить объект ArrayList (myCars) в двоичном виде.
    List<JamesBondCar> myCars = new List<JamesBondCar>();

    BinaryFormatter binFormat = new BinaryFormatter();
    using(Stream fStream = new FileStream("AllMyCars.dat",
        FileMode.Create, FileAccess.Write, FileShare.None))
    {
        binFormat.Serialize(fStream, myCars);
    }
    Console.WriteLine("=> Saved list of cars in binary!");
}
```

Настройка процессов сериализации SOAP и двоичной сериализации

В большинстве случаев стандартная схема сериализации, предоставляемая платформой .NET, вполне подходит. Нужно лишь применить атрибут `[Serializable]` к связанным типам и передать дерево объектов выбранному формату для обработки. Однако в некоторых случаях может понадобиться вмешательство в процесс конструирования дерева и процесс сериализации. Например, может существовать бизнес-правило, которое гласит, что все поля данных должны сохраняться в определенном формате, или же необходимо добавить дополнительные данные в поток, которые напрямую не отображаются на поля сохраняемого объекта (скажем, временные метки или уникальные идентификаторы).

В пространстве имен `System.Runtime.Serialization` предусмотрено несколько типов, которые позволяют вмешаться в процесс сериализации объектов. В табл. 20.13 описаны основные типы, о которых следует знать.

Таблица 20.13. Основные типы пространства имен `System.Runtime.Serialization`

Тип	Описание
<code>ISerializable</code>	Этот интерфейс может быть реализован типом <code>[Serializable]</code> для управления его сериализацией и десериализацией
<code>ObjectIDGenerator</code>	Этот тип генерирует идентификаторы для членов в графе объектов
<code>[OnDeserialized]</code>	Этот атрибут позволяет указать метод, который будет вызван немедленно после десериализации объекта
<code>[OnDeserializing]</code>	Этот атрибут позволяет указать метод, который будет вызван перед началом процесса десериализации
<code>[OnSerialized]</code>	Этот атрибут позволяет указать метод, который будет вызван немедленно после того, как объект сериализован
<code>[OnSerializing]</code>	Этот атрибут позволяет указать метод, который будет вызван перед началом процесса сериализации
<code>[OptionalField]</code>	Этот атрибут позволяет определить поле типа, которое может быть пропущено в указанном потоке
<code>SerializationInfo</code>	В сущности, этот класс является <i>пакетом свойств</i> , который поддерживает пары “имя/значение”, представляющие состояние объекта во время процесса сериализации

Углубленный взгляд на сериализацию объектов

Прежде чем приступить к изучению различных способов настройки процесса сериализации, полезно внимательнее присмотреться к тому, что происходит “за кулисами”. Когда `BinaryFormatter` сериализует граф объектов, он отвечает за передачу следующей информации в указанный поток:

- полностью заданное имя объекта в графе (например, `MyApp.JamesBondCar`);
- имя сборки, определяющей граф объектов (например, `MyApp.exe`);
- экземпляр класса `SerializationInfo`, содержащего все данные состояния, которые поддерживаются членами графа объектов.

Во время процесса десериализации `BinaryFormatter` использует ту же самую информацию для построения идентичной копии объекта с применением данных, извлеченных из потока-источника. Процесс, выполняемый `SoapFormatter`, очень похож.

На заметку! Вспомните, что для обеспечения максимальной мобильности объекта форматор `XmlSerializer` не сохраняет полностью заданное имя типа или имя сборки, в которой он содержится. Этот тип может сохранять только открытые данные.

Помимо перемещения необходимых данных в поток и обратно, форматоры также анализируют члены графа объектов на предмет перечисленных ниже частей инфраструктуры.

- Проверка пометки объекта атрибутом `[Serializable]`. Если объект не помечен, генерируется исключение `SerializationException`.
- Если объект помечен атрибутом `[Serializable]`, производится проверка, реализует ли объект интерфейс `ISerializable`. Если да, на этом объекте вызывается метод `GetObjectData()`.
- Если объект не реализует интерфейс `ISerializable`, используется стандартный процесс сериализации, который обрабатывает все поля, не помеченные как `[NonSerialized]`.

В дополнение к определению того, поддерживает ли тип интерфейс `ISerializable`, форматоры также отвечают за исследование типов на предмет поддержки членов, которые оснащены атрибутами `[OnSerializing]`, `[OnSerialized]`, `[OnDeserializing]` или `[OnDeserialized]`. Мы рассмотрим назначение этих атрибутов чуть позже, а сначала давайте посмотрим на предназначение `ISerializable`.

Настройка сериализации с использованием `ISerializable`

Для объектов, которые помечены атрибутом `[Serializable]`, имеется возможность реализации интерфейса `ISerializable`. Реализация этого интерфейса позволяет вмешаться в процесс сериализации и выполнить необходимое форматирование данных до и после сериализации.

На заметку! После выхода версии .NET 2.0 предпочтительный способ настройки процесса сериализации стал предусматривать применение атрибутов сериализации. Тем не менее, знание интерфейса `ISerializable` важно для сопровождения систем, которые уже существуют.

Интерфейс `ISerializable` довольно прост, учитывая, что в нем определен единственный метод `GetObjectData()`:

```
// Чтобы получить возможность настройки процесса сериализации,
// необходимо реализовать ISerializable.
public interface ISerializable
{
    void GetObjectData(SerializationInfo info,
        StreamingContext context);
}
```

Метод `GetObjectData()` вызывается автоматически заданным форматором во время процесса сериализации. Реализация этого метода заполняет входной параметр `SerializationInfo` последовательностью пар "имя/значение", которые (обычно) отображают данные полей сохраняемого объекта. В `SerializationInfo` определены многочисленные вариации перегруженного метода `AddValue()`, а также небольшой набор

свойств, которые позволяют устанавливать и получать имя типа, определять сборку и счетчик членов. Ниже показано частичное определение:

```
public sealed class SerializationInfo
{
    public SerializationInfo(Type type, IFormatterConverter converter);
    public string AssemblyName { get; set; }
    public string FullTypeName { get; set; }
    public int MemberCount { get; }
    public void AddValue(string name, short value);
    public void AddValue(string name, ushort value);
    public void AddValue(string name, int value);
    ...
}
```

Типы, реализующие интерфейс `ISerializable`, также должны определять специальный конструктор со следующей сигнатурой:

```
// Необходимо предоставить специальный конструктор с такой сигнатурой,
// чтобы позволить исполняющей среде устанавливать состояние объекта.
[Serializable]
class SomeClass : ISerializable
{
    protected SomeClass (SerializationInfo si, StreamingContext ctx) {...}
    ...
}
```

Обратите внимание, что видимость конструктора указана как `protected`. Это решено, поскольку форматер будет иметь доступ к этому члену независимо от его видимости. Такие специальные конструкторы обычно определяются как `protected` (или `private`), гарантируя тем самым, что небрежный пользователь объекта никогда не создаст объект с их помощью. Первым параметром конструктора является экземпляр типа `SerializationInfo` (который был показан ранее).

Второй параметр этого специального конструктора имеет тип `StreamingContext` и содержит информацию относительно источника или места назначения передаваемых данных. Наиболее информативным членом `StreamingContext` является свойство `State`, которое хранит значение из перечисления `StreamingContextStates`. Значения этого перечисления представляют базовую композицию текущего потока.

Если только не планируется реализовать какие-то низкоуровневые службы удаленного взаимодействия, то иметь дело с этим перечислением напрямую придется редко. Тем не менее, ниже приведены члены перечисления `StreamingContextStates` (за более подробной информацией обращайтесь в документацию .NET Framework 4.5 SDK):

```
public enum StreamingContextStates
{
    CrossProcess,
    CrossMachine,
    File,
    Persistence,
    Remoting,
    Other,
    Clone,
    CrossAppDomain,
    All
}
```

Давайте рассмотрим пример настройки процесса сериализации с использованием `ISerializable`. Предположим, что имеется новый проект консольного приложения (по имени `CustomSerialization`), в котором определен тип класса, содержащий два эле-

мента данных string. Также представим, что требуется обеспечить сериализацию этих объектов string в верхнем регистре, а десериализацию — в нижнем. Для удовлетворения этих правил можно реализовать интерфейс ISerializable так, как показано ниже (не забудьте импортировать пространство имен System.Runtime.Serialization):

```
[Serializable]
class StringData : ISerializable
{
    private string dataItemOne = "First data block";
    private string dataItemTwo = "More data";

    public StringData() {}
    protected StringData(SerializationInfo si, StreamingContext ctx)
    {
        // Восстановить переменные-члены из потока.
        dataItemOne = si.GetString("First_Item").ToLower();
        dataItemTwo = si.GetString("dataItemTwo").ToLower();
    }

    void ISerializable.GetObjectData(SerializationInfo info, StreamingContext ctx)
    {
        // Наполнить объект SerializationInfo форматированными данными.
        info.AddValue("First_Item", dataItemOne.ToUpper());
        info.AddValue("dataItemTwo", dataItemTwo.ToUpper());
    }
}
```

Обратите внимание, что при наполнении объекта типа SerializationInfo внутри метода GetObjectData() именовать элементы данных идентично именам внутренних переменных-членов типа не обязательно. Это очевидно полезно, если требуется отвязать данные типа от формата хранения. Однако имейте в виду, что получать значения в специальном защищенном конструкторе необходимо с указанием тех же самых имен, что были назначены внутри GetObjectData().

Чтобы опробовать специализированную сериализацию, предположим, что экземпляр MyStringData сохраняется с применением SoapFormatter (обновите соответствующим образом ссылки на сборки и директивы using):

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Custom Serialization *****");

    // Вспомните, что этот тип реализует ISerializable.
    StringData myData = new StringData();

    // Сохранить в локальный файл в формате SOAP.
    SoapFormatter soapFormat = new SoapFormatter();
    using(Stream fStream = new FileStream("MyData.soap",
        FileMode.Create, FileAccess.Write, FileShare.None))
    {
        soapFormat.Serialize(fStream, myData);
    }
    Console.ReadLine();
}
```

Просматривая полученный файл *.soap, вы заметите, что строковые поля действительно сохранены в верхнем регистре, как показано ниже:

```
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
```

```

xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>

  <al:StringData id="ref-1" ...>
    <First_Item id="ref-3">FIRST DATA BLOCK</First_Item>
    <dataItemTwo id="ref-4">MORE DATA</dataItemTwo>
  </al:StringData>
</SOAP-ENV:Body>

</SOAP-ENV:Envelope>

```

Настройка сериализации с использованием атрибутов

Хотя реализация интерфейса `ISerializable` является одним из возможных способов настройки процесса сериализации, с момента выхода версии .NET 2.0 предпочтительным способом такой настройки стало определение методов, оснащенных атрибутами из следующего перечня: `[OnSerializing]`, `[OnSerialized]`, `[OnDeserializing]` и `[OnDeserialized]`. Использование этих атрибутов дает менее громоздкий код, чем реализация интерфейса `ISerializable`, учитывая, что не приходится вручную взаимодействовать с параметром `SerializationInfo`. Вместо этого можно напрямую модифицировать данные состояния, когда формater работает с типом.

На заметку! Эти атрибуты сериализации определены в пространстве имен `System.Runtime.Serialization`.

В случае применения этих атрибутов метод должен быть определен так, чтобы принимать параметр `StreamingContext` и не возвращать ничего (иначе будет сгенерировано исключение времени выполнения). Обратите внимание, что применять каждый из атрибутов сериализации не обязательно, а можно просто вмешиваться в те стадии процесса сериализации, которые интересуют. В целях иллюстрации ниже приведен новый тип `[Serializable]`, который имеет те же самые требования, что и `StringData`, но на этот раз он полагается на использование атрибутов `[OnSerializing]` и `[OnDeserialized]`:

```

[Serializable]
class MoreData
{
    private string dataItemOne = "First data block";
    private string dataItemTwo = "More data";

    [OnSerializing]
    private void OnSerializing(StreamingContext context)
    {
        // Вызывается во время процесса сериализации.
        dataItemOne = dataItemOne.ToUpper();
        dataItemTwo = dataItemTwo.ToUpper();
    }

    [OnDeserialized]
    private void OnDeserialized(StreamingContext context)
    {
        // Вызывается по завершении процесса десериализации.
        dataItemOne = dataItemOne.ToLower();
        dataItemTwo = dataItemTwo.ToLower();
    }
}

```

Выполнив сериализацию этого нового типа, вы снова обнаружите, что данные сохраняются в верхнем регистре, а десериализируются — в нижнем.

Исходный код. Проект CustomSerialization доступен в подкаталоге Chapter 20.

Приведенный пример наглядно продемонстрировал основные детали служб сериализации объектов, включая различные способы настройки этого процесса. Как было показано, процессы сериализации и десериализации существенно упрощают задачу сохранения больших объемов данных, причем с меньшими затратами, чем при работе с различными классами чтения/записи данных из пространства имен System.IO.

Резюме

Эта глава начиналась с демонстрации использования типов Directory (DirectoryInfo) и File (FileInfo). Вы узнали, что упомянутые классы позволяют манипулировать физическими файлами и каталогами на жестком диске. Затем вы ознакомились с несколькими типами, унаследованными от абстрактного класса Stream. Учитывая, что типы, производные от Stream, оперируют низкоуровневым потоком байтов, пространство имен System.IO предоставляет многочисленные типы для чтения/записи (StreamWriter, StringWriter, BinaryWriter и т.п.), которые упрощают этот процесс. Попутно вы узнали о функциональности, предлагаемой типом DriveType, и научились наблюдать за файлами с применением типа FileSystemWatcher, а также взаимодействовать с потоками в асинхронном режиме.

В главе также рассматривались службы сериализации объектов. Было показано, что платформа .NET использует граф объектов, чтобы учесть полный набор объектов, которые должны сохраниться в потоке. До тех пор, пока каждый объект в графе помечен атрибутом [Serializable], данные сохраняются в выбранном формате (двоичном или SOAP).

Вы также ознакомились с возможностями настройки готового процесса сериализации посредством двух возможных подходов. Во-первых, вы узнали, как реализовать интерфейс ISerializable (и поддерживать специальный закрытый конструктор), что позволяет вмешиваться в процесс сохранения форматором данных объекта. Во-вторых, вы ознакомились с набором атрибутов .NET, которые упрощают процесс специальной сериализации. Все, что нужно для этого — применить атрибут [OnSerializing], [OnSerialized], [OnDeserializing] или [OnDeserialized] к членам, принимающим параметр StreamingContext, и форматоры будут вызывать их на соответствующих фазах сериализации или десериализации.

ADO.NET, часть I: подключенный уровень

Платформа .NET определяет ряд пространств имен, которые позволяют взаимодействовать с реляционными базами данных. Все вместе эти пространства имен известны как *ADO.NET*. В настоящей главе вначале будет описана в общих чертах роль самой инфраструктуры ADO.NET, а затем мы перейдем к теме поставщиков данных ADO.NET. Платформа .NET поддерживает множество различных поставщиков данных, каждый из которых оптимизирован на взаимодействие с конкретной системой управления базами данных (СУБД), такой как Microsoft SQL Server, Oracle, MySQL и т.д.

После того как мы разберемся с общими возможностями различных поставщиков данных, мы рассмотрим образец фабрики поставщиков данных. Вы увидите, что типы из пространства имен *System.Data.Common* (и связанного с ним файла *App.config*) позволяют создать единую кодовую базу, которая может динамически выбрать соответствующий поставщик данных без необходимости перекомпиляции или нового разрывывания кодовой базы этого приложения.

Пожалуй, наиболее важно то, что в данной главе вы сможете создать собственную сборку библиотеки доступа к данным (*AutoLotDAL.dll*), в которой будут инкапсулированы различные операции, выполняемые в пользовательской базе данных *AutoLot*. Эта библиотека, которая будет расширена в главах 23 и 24, неоднократно пригодится в последующих главах. И в завершение мы познакомимся с транзакциями баз данных.

Высокоуровневое определение ADO.NET

Если вы уже знакомы с предыдущей моделью Microsoft доступа к данным на основе COM (*Active Data Objects* — ADO), то сразу же учтите, что ADO.NET имеет очень мало общего с ADO, за исключением букв “A”, “D” и “O”. Хотя некоторая взаимосвязь между этими двумя системами и существует (например, в обеих присутствует концепция объектов подключения и объектов команд), некоторые знакомые по ADO типы (например, *Recordset*) больше не доступны. Кроме того, в ADO.NET появился ряд новых типов, не имеющих прямых эквивалентов в классической технологии ADO (например, адаптер данных).

В отличие от классической ADO, которая была в основном предназначена для тесно связанных клиент-серверных систем, технология ADO.NET больше нацелена на автономную работу с помощью объектов *DataSet*. Эти типы представляют локальные копии любого количества взаимосвязанных таблиц данных, каждая из которых содержит набор строк и столбцов. Объекты *DataSet* позволяют вызывающей сборке (наподобие веб-страницы или программы, выполняющейся на настольном компьютере) работать с содержимым *DataSet*, изменять его, не требуя подключения к источнику данных, и

отправлять обратно блоки измененных данных для обработки с помощью соответствующего *адаптера данных*.

С точки зрения программиста, основу ADO.NET составляет базовая сборка по имени `System.Data.dll`. В этом двоичном файле находится значительное количество пространств имен (рис. 21.1), многие из которых представляют типы конкретного поставщика данных ADO.NET (о которых речь пойдет ниже).

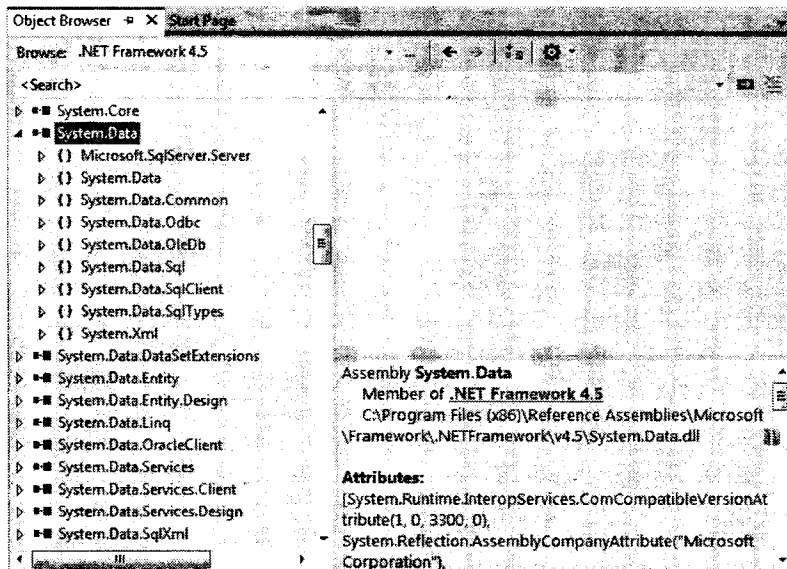


Рис. 21.1. Базовая сборка ADO.NET называется `System.Data.dll`

Большинство шаблонов проектов Visual Studio автоматически ссылаются на эту ключевую библиотеку доступа к данным. Учтите также, что кроме `System.Data.dll` существуют и другие ориентированные на ADO.NET сборки, которые необходимо вручную указывать в текущем проекте с помощью диалогового окна Add Reference (Добавление ссылки).

Три грани ADO.NET

Библиотеки ADO.NET можно применять тремя концептуально различными способами: в подключенном режиме, в автономном режиме и с помощью технологии Entity Framework. При использовании *подключенного уровня*, рассматриваемого в этой главе, кодовая база явно подключается к соответствующему хранилищу данных и отключается от него. При таком способе использования ADO.NET обычно происходит взаимодействие с хранилищем данных с помощью объектов подключения, объектов команд и объектов чтения данных.

Автономный уровень, который будет рассмотрен в главе 22, позволяет работать с набором объектов `DataTable` (содержащихся в `DataSet`), который представляет на стороне клиента копию внешних данных. При получении `DataSet` с помощью соответствующего объекта адаптера данных подключение открывается и закрывается автоматически. Понятно, что этот подход помогает быстро освобождать подключения для других вызовов и повышает масштабируемость систем.

Получив объект `DataSet`, вызывающий код может просматривать и обрабатывать данные без затрат на сетевой трафик. А если нужно занести изменения в хранилище данных, то адаптер данных (вместе с набором операторов SQL) задействуется для об-

новления данных — при этом подключение открывается заново для проведения обновлений в базе, а затем сразу же закрывается.

И, наконец, в главе 23 вы ознакомитесь с API-интерфейсом доступа к данным, который называется *Entity Framework* (EF). Инфраструктура EF позволяет взаимодействовать с реляционной базой данных с помощью объектов на стороне клиента, которые скрывают конкретные низкоуровневые особенности СУБД. Кроме того, модель программирования EF позволяет взаимодействовать с реляционными СУБД с помощью строго типизированных запросов LINQ, использующих грамматику LINQ to Entities.

Поставщики данных ADO.NET

В ADO.NET нет единого набора типов, которые взаимодействуют с различными СУБД. Вместо этого в ADO.NET имеются различные *поставщики данных*, каждый из которых оптимизирован для взаимодействия с конкретной СУБД. Первая выгода этого подхода состоит в том, что можно запрограммировать специализированный поставщик данных для доступа к любым уникальным средствам конкретной СУБД. Еще одна выгода — конкретный поставщик данных может напрямую подключиться к механизму соответствующей СУБД, не пользуясь промежуточным уровнем отображения.

В первом приближении поставщик данных можно рассматривать как набор типов, определенных в заданном пространстве имен, который предназначен для взаимодействия с конкретным источником данных. Однако независимо от используемого поставщика данных, каждый из них определяет набор классов, обеспечивающих основную функциональность. В табл. 21.1 приведены некоторые общие основные объекты, их базовые классы (определенные в пространстве имен `System.Data.Common`) и основные интерфейсы (определенные в пространстве имен `System.Data`), которые они реализуют.

Таблица 21.1. Основные объекты поставщиков данных ADO.NET

Тип объекта	Базовый класс	Соответствующие интерфейсы	Описание
Connection	DbConnection	IDbConnection	Позволяет подключаться к хранилищу данных и отключаться от него. Кроме того, объекты подключения обеспечивают доступ к соответствующим объектам транзакций
Command	DbCommand	IDbCommand	Представляет SQL-запрос или хранимую процедуру. Кроме того, объекты команд предоставляют доступ к объекту чтения данных конкретного поставщика данных
DataReader	DbDataReader	IDataReader, IDataRecord	Предоставляет доступ к данным, предназначенным только для чтения, в прямом направлении с помощью курсора на стороне сервера
DataAdapter	DbDataAdapter	IDataAdapter, IDbDataAdapter	Передаёт наборы данных между вызывающим процессом и хранилищем данных. Адаптеры данных содержат подключение и набор из четырех внутренних объектов команд для выборки, вставки, изменения и удаления информации в хранилище данных

Тип объекта	Базовый класс	Соответствующие интерфейсы	Описание
Parameter	DbParameter	IDataParameter, IDbDataParameter	Представляет именованный параметр в параметризованном запросе
Transaction	DbTransaction	IDbTransaction	Инкапсулирует транзакцию базы данных

Конкретные имена этих основных классов различаются у различных поставщиков (например, SqlConnection, OracleConnection, OdbcConnection и MySqlConnection), но все эти объекты порождены от одного и того же базового класса (в случае объектов подключения это DbConnection), который реализует идентичные интерфейсы (вроде IDbConnection). Поэтому если вы научитесь работать с одним поставщиком данных, то легко справитесь и с остальными.

На заметку! В ADO.NET термин *объект подключения* на самом деле относится к конкретному типу, производному от DbConnection; не существует класса, который бы назывался Connection. То же можно сказать и об объекте команды, объекте адаптера данных и т.д. По соглашению имена объектов в конкретном поставщике данных имеют префиксы соответствующей СУБД (например, SqlConnection, OracleConnection, SqlDataReader и т.д.).

На рис. 21.2 подробно показано, что означают поставщики данных в ADO.NET. На этой диаграмме Клиентская сборка может быть практически любым типом приложения .NET: консольная программа, приложение Windows Forms, приложение WPF, веб-страница ASP.NET, служба WCF, библиотека кода .NET и т.д.

Кроме типов, показанных на рис. 21.2, поставщики данных содержат и другие типы. Однако эти основные объекты определяют общие свойства всех поставщиков данных.

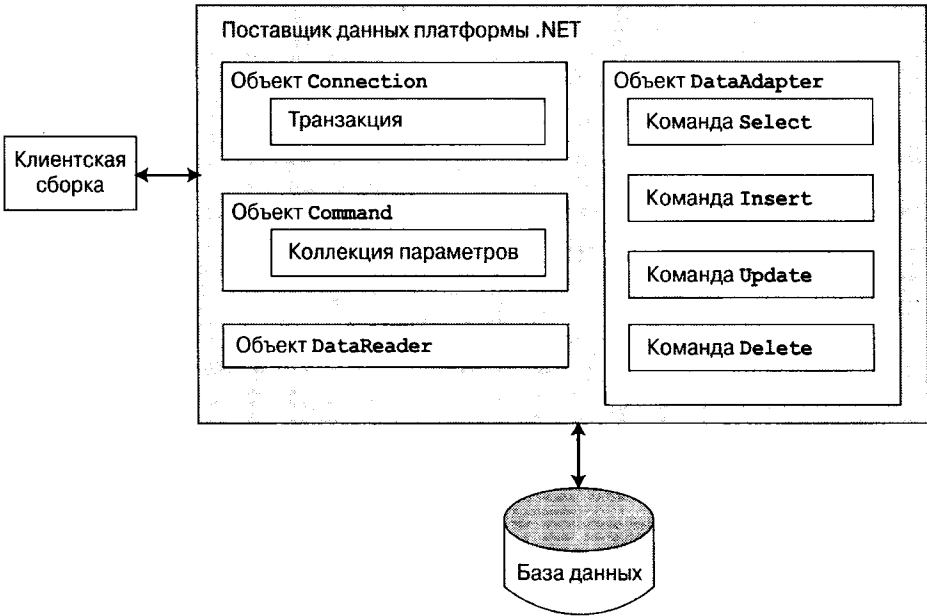


Рис. 21.2. Поставщики данных ADO.NET предоставляют доступ к конкретным СУБД

Поставщики данных ADO.NET от Microsoft

Дистрибутив .NET, поставляемый Microsoft, содержит множество различных поставщиков данных, например, поставщик в стиле Oracle, SQL Server и OLE DB/ODBC. В табл. 21.2 перечислены пространства имен и содержащие их сборки для всех поставщиков данных Microsoft ADO.NET.

Таблица 21.2. Поставщики данных Microsoft ADO.NET

Поставщик данных	Пространство имен	Сборка
OLE DB	System.Data.OleDb	System.Data.dll
Microsoft SQL Server	System.Data.SqlClient	System.Data.dll
Microsoft SQL Server Mobile	System.Data.SqlServerCe	System.Data.SqlServerCe.dll
ODBC	System.Data.Odbc	System.Data.dll

На заметку! Поставщик данных, выполняющий непосредственное отображение на механизм Jet (и, следовательно, на Microsoft Access), отсутствует. Если вам нужно взаимодействие с файлом данных Access, это можно сделать с помощью поставщика данных OLE DB или ODBC.

Поставщик данных OLE DB, состоящий из типов, которые определены в пространстве имен System.Data.OleDb, обеспечивает доступ к данным, находящимся в любом хранилище данных, если оно поддерживает классический протокол OLE DB на основе COM. Этот поставщик позволяет взаимодействовать с любой базой данных, совместимой с OLE DB — для этого потребуется лишь настроить сегмент Provider в строке соединения.

Однако поставщик OLE DB неявно взаимодействует с различными COM-объектами, что может снизить производительность приложения. В общем-то, поставщик данных OLE DB нужен только для взаимодействия с какой-нибудь СУБД, для которой нет специального поставщика данных .NET. Но, учитывая тот факт, что сейчас у любой мало-мальски известной СУБД доступен для загрузки и специальный поставщик данных ADO.NET, следует рассматривать System.Data.OleDb как устаревшее пространство имен, которое вряд ли понадобится в мире .NET 4.5 (если к тому же учесть модель фабрики поставщиков данных, введенную в .NET 2.0, о которой будет рассказано немного позже).

На заметку! В одном случае использование типов из System.Data.OleDb все-таки необходимо: если понадобится взаимодействие с Microsoft SQL Server версии 6.5 или более ранней. Пространство имен System.Data.SqlClient может взаимодействовать только с Microsoft SQL Server 7.0 и последующих версий.

Поставщик данных Microsoft SQL Server предоставляет прямой доступ к хранилищам данных Microsoft SQL Server — и только к хранилищам данных SQL Server (версии 7.0 или больше). Пространство имен System.Data.SqlClient содержит типы, необходимые для поставщика SQL Server, и предлагает ту же базовую функциональность, что и поставщик OLE DB. Основное различие между ними состоит в том, что поставщик SQL Server не задействует уровень OLE DB и дает существенный выигрыш в производительности. А поставщик данных Microsoft SQL Server позволяет получить доступ к уникальным возможностям этой конкретной СУБД.

Остальные поставщики, предлагаемые Microsoft (System.Data.Odbc и System.Data.SqlClientCe), обеспечивают взаимодействие с ODBC-подключениями и доступ

к SQL Server версии Mobile (которая обычно применяется на портативных устройствах, подобных Windows Mobile). Типы ODBC, определенные в пространстве имен `System.Data.Odbc`, обычно полезны, только если требуется взаимодействие с СУБД, для которой нет специального поставщика данных .NET. Так бывает нередко, т.к. ODBC является широко распространенной моделью, которая предоставляет доступ к целому ряду хранилищ данных.

0 сборке `System.Data.OracleClient.dll`

В предшествующих версиях платформы .NET имела сборка `System.Data.OracleClient.dll`, которая, как понятно из названия, предоставляла поставщик данных для взаимодействия с базами данных Oracle. Однако в версии .NET 4.0 эта сборка была помечена как *устаревшая*, и в дальнейшем применять ее *не рекомендуется*.

На первый взгляд может показаться, что ADO.NET постепенно концентрируется только на хранилищах данных от Microsoft, но это не так. Просто Oracle поставляет собственную сборку .NET, которая разработана на тех же общих принципах, что и поставщики данных, предоставляемые Microsoft. Если вам понадобится эта сборка, зайдите на страницу загрузки веб-сайта Oracle по следующему адресу:

www.oracle.com/technetwork/indexes/downloads/index.html

Получение сторонних поставщиков данных ADO.NET

Кроме поставщиков данных, поставляемых Microsoft (а также собственной библиотеки .NET для Oracle), существует и множество сторонних поставщиков данных для различных СУБД, как коммерческих, так и с открытым исходным кодом. Скорее всего, у вас не возникнет трудностей при получении поставщика данных ADO.NET непосредственно от разработчика СУБД, но на всякий случай возьмите на заметку следующий сайт:

<http://www.sqlsummit.com/DataProv.htm>

Это один из множества сайтов, где собрана документация на все известные поставщики данных ADO.NET и ссылки на дополнительную информацию и сайты загрузки. Здесь перечислены различные поставщики ADO.NET: SQLite, IBM DB2, MySQL, PostgreSQL, TurboDB, Sybase и многие другие.

Однако, несмотря на наличие множества поставщиков данных ADO.NET, в примере этой главы будет применяться поставщик данных Microsoft SQL Server (`System.Data.SqlClient.dll`). Этот поставщик позволяет взаимодействовать с Microsoft SQL Server 7.0 и последующих версий, в том числе и с SQL Server Express Edition. Если вы хотите использовать ADO.NET для работы с другой СУБД, проблем возникать не должно, если вы уясните материал, изложенный ниже.

Дополнительные пространства имен ADO.NET

Кроме пространств имен .NET, которые определяют типы конкретных поставщиков данных, в библиотеках базовых классов .NET содержатся дополнительные пространства имен, ориентированные на ADO.NET. Некоторые из них перечислены в табл. 21.3 (сборки и пространства имен, относящиеся к Entity Framework, будут описаны в главе 23).

Мы не собираемся изучать каждый тип в каждом пространстве имен ADO.NET (эта задача потребовала бы отдельной книги), однако важно ознакомиться с типами из пространства имен `System.Data`.

Таблица 21.3. Дополнительные пространства имен для работы с ADO.NET

Пространство имен	Описание
Microsoft.SqlServer.Server	Содержит типы для работы службы интеграции CLR и SQL Server 2005 и последующих версий
System.Data	Определяет основные типы ADO.NET, используемые всеми поставщиками данных, в том числе общие интерфейсы и разнообразные типы, представляющие автономный уровень (например, DataSet и DataTable)
System.Data.Common	Содержит типы для общего использования всеми поставщиками ADO.NET, в том числе и общие абстрактные базовые классы
System.Data.Sql	Содержит типы, позволяющие обнаружить экземпляры Microsoft SQL Server, которые установлены в текущей локальной сети
System.Data.SqlTypes	Содержит типы данных, используемые Microsoft SQL Server. Можно применять и соответствующие типы CLR, но пространство SqlTypes оптимизировано для работы с SQL Server (например, если база данных SQL Server содержит целочисленное значение, его можно представить либо как int, либо как SqlTypes.SqlInt32)

Типы из пространства имен System.Data

System.Data является “наименьшим общим знаменателем” для всех пространств имен ADO.NET. Если нужен доступ к данным, то приложения ADO.NET невозможно создать без указания этого пространства имен. Оно содержит общие типы, используемые всеми поставщиками данных ADO.NET, независимо от непосредственного хранилища данных. Кроме ряда исключений, специфичных для баз данных (NotNullAllowedException, RowNotInTableException и MissingPrimaryKeyException), System.Data содержит типы, представляющие различные примитивы баз данных (например, таблицы, строки, столбцы и ограничения), а также общие интерфейсы, реализованные объектами поставщиков данных. Некоторые из основных типов перечислены в табл. 21.4.

Таблица 21.4. Основные члены пространства имен System.Data

Тип	Описание
Constraint	Ограничение для заданного объекта DataColumn
DataColumn	Один столбец в объекте DataTable
DataRelation	Отношение “родительский–дочерний” между двумя объектами DataTable
DataRow	Одна строка в объекте DataTable
DataSet	Находящийся в памяти кеш данных, который состоит из любого количества взаимосвязанных объектов DataTable
DataTable	Табличный блок данных, находящихся в памяти
DataTableReader	Позволяет обращаться с DataTable как с примитивным курсором (доступ к данным только для чтения и только в прямом направлении)
DataRowView	Специализированное представление DataTable для сортировки, фильтрации, поиска, редактирования и навигации
IDataAdapter	Определяет основное поведение объекта адаптера данных

Тип	Описание
<code>IDataParameter</code>	Определяет основное поведение объекта параметра
<code>IDataReader</code>	Определяет основное поведение объекта чтения данных
<code>IDbCommand</code>	Определяет основное поведение объекта команды
<code>IDbDataAdapter</code>	Расширяет <code>IDataAdapter</code> для получения дополнительных возможностей объекта адаптера данных
<code>IDbTransaction</code>	Определяет основное поведение объекта транзакции

Подавляющее большинство классов из `System.Data` используется при программировании для автономного уровня ADO.NET. В следующей главе вы более подробно ознакомитесь с `DataSet` и связанными с ним объектами (например, `DataTable`, `DataRelation` и `DataRow`), а также научитесь применять их (и соответствующие адаптеры данных) для представления копий удаленных данных на стороне клиента и работы с ними.

Однако следующей задачей будет знакомство с основными интерфейсами `System.Data` на высоком уровне: это поможет лучше разобраться в общей функциональности любого поставщика данных. Конкретные детали будут предоставлены далее в этой главе, а пока просто рассмотрим общее поведение произвольного типа интерфейса.

Роль интерфейса `IDbConnection`

Тип `IDbConnection` реализован объектом подключения поставщика данных. Этот интерфейс определяет набор членов, применяемых для настройки подключения к конкретному хранилищу данных, а, кроме того, позволяет получить объект транзакции поставщика данных. Вот формальное определение `IDbConnection`:

```
public interface IDbConnection : IDisposable
{
    string ConnectionString { get; set; }
    int ConnectionTimeout { get; }
    string Database { get; }
    ConnectionState State { get; }

    IDbTransaction BeginTransaction();
    IDbTransaction BeginTransaction(IsolationLevel il);
    void ChangeDatabase(string databaseName);
    void Close();
    IDbCommand CreateCommand();
    void Open();
}
```

На заметку! Как и многие другие типы в библиотеках базовых классов .NET, метод `Close()` функционально эквивалентен непосредственному или косвенному вызову метода `Dispose()` с помощью области видимости `C#` (см. главу 13).

Роль интерфейса `IDbTransaction`

Перегруженный метод `BeginTransaction()`, определенный в интерфейсе `IDbConnection`, предоставляет доступ к объекту транзакции поставщика. Члены, определенные в `IDbTransaction`, позволяют программным образом взаимодействовать с сеансом транзакций и соответствующим хранилищем данных:

```
public interface IDbTransaction : IDisposable
{
    IDbConnection Connection { get; }
    IsolationLevel IsolationLevel { get; }
    void Commit();
    void Rollback();
}
```

Роль интерфейса IDbCommand

Интерфейс `IDbCommand` реализуется объектом команды поставщика данных. Как и другие объектные модели доступа к данным, объекты команды позволяют программно работать с операторами SQL, хранимыми процедурами и параметризованными запросами. Кроме того, объекты команды обеспечивают доступ к типу чтения данных поставщика данных с помощью перегруженного метода `ExecuteReader()`:

```
public interface IDbCommand : IDisposable
{
    string CommandText { get; set; }
    int CommandTimeout { get; set; }
    CommandType CommandType { get; set; }
    IDbConnection Connection { get; set; }
    IDataParameterCollection Parameters { get; }
    IDbTransaction Transaction { get; set; }
    UpdateRowSource UpdatedRowSource { get; set; }
    void Cancel();
    IDbDataParameter CreateParameter();
    int ExecuteNonQuery();
    IDataReader ExecuteReader();
    IDataReader ExecuteReader(CommandBehavior behavior);
    object ExecuteScalar();
    void Prepare();
}
```

Роль интерфейсов IDbDataParameter и IDataParameter

Свойство `Parameters` интерфейса `IDbCommand` возвращает строго типизированную коллекцию, которая реализует `IDataParameterCollection`. Этот интерфейс предоставляет доступ к набору классов, совместимых с `IDbDataParameter` (объекты параметров):

```
public interface IDbDataParameter : IDataParameter
{
    byte Precision { get; set; }
    byte Scale { get; set; }
    int Size { get; set; }
}
```

Интерфейс `IDbDataParameter` расширяет интерфейс `IDataParameter` с целью получения дополнительных поведений:

```
public interface IDataParameter
{
    DbType DbType { get; set; }
    ParameterDirection Direction { get; set; }
    bool IsNullable { get; }
    string ParameterName { get; set; }
    string SourceColumn { get; set; }
    DataRowVersion SourceVersion { get; set; }
    object Value { get; set; }
}
```


Как видите, функциональность интерфейсов `IDbDataParameter` и `IDataParameter` позволяет использовать параметры в командах SQL (в том числе и в хранимых процедурах) с помощью специфических объектов параметров ADO.NET вместо жестко закодированных строковых литералов.

Роль интерфейсов `IDbDataAdapter` и `IDataAdapter`

Адаптеры данных используются для выборки и занесения объектов `DataSet` в конкретное хранилище данных. Поэтому интерфейс `IDbDataAdapter` определяет приведенный ниже набор свойств, предназначенных для поддержки операторов SQL для соответствующих операций выборки, вставки, изменения и удаления:

```
public interface IDbDataAdapter : IDataAdapter
{
    IDbCommand DeleteCommand { get; set; }
    IDbCommand InsertCommand { get; set; }
    IDbCommand SelectCommand { get; set; }
    IDbCommand UpdateCommand { get; set; }
}
```

Кроме этих четырех свойств, адаптер данных ADO.NET заодно получает поведение, определенное в базовом интерфейсе `IDataAdapter`. Этот интерфейс определяет основную функцию типа адаптера данных: возможность передачи объектов `DataSet` между вызывающим процессом и непосредственным хранилищем данных с помощью методов `Fill()` и `Update()`. Кроме того, посредством свойства `TableMappings` интерфейс `IDataAdapter` позволяет отобразить имена столбцов из базы данных на более понятные отображаемые имена:

```
public interface IDataAdapter
{
    MissingMappingAction MissingMappingAction { get; set; }
    MissingSchemaAction MissingSchemaAction { get; set; }
    ITableMappingCollection TableMappings { get; }

    int Fill(System.Data.DataSet dataSet);
    DataTable[] FillSchema(DataSet dataSet, SchemaType schemaType);
    IDataParameter[] GetFillParameters();
    int Update(DataSet dataSet);
}
```

Роль интерфейсов `IDataReader` и `IDataRecord`

Следующим интерфейсом, о котором важно знать, является `IDataReader`, который представляет общее поведение, поддерживаемое конкретным объектом чтения данных. Получив от поставщика данных ADO.NET тип, совместимый с `IDataReader`, вы сможете выполнять проход по результирующему набору только в прямом направлении в стиле только для чтения.

```
public interface IDataReader : IDisposable, IDataRecord
{
    int Depth { get; }
    bool IsClosed { get; }
    int RecordsAffected { get; }

    void Close();
    DataTable GetSchemaTable();
    bool NextResult();
    bool Read();
}
```

И, наконец, интерфейс `IDataReader` расширяет `IDataRecord`, в котором определено значительное количество членов, позволяющих извлекать из потока строго типизированные значения, а не приводить к нужному типу обобщенный тип `System.Object`, который получен от перегруженного метода индексатора из объекта чтения данных. Ниже приведена часть листинга различных методов `GetXXX()`, определенных в `IDataRecord` (полный листинг ищите в документации .NET Framework 4.5 SDK):

```
public interface IDataRecord
{
    int FieldCount { get; }
    object this[ string name ] { get; }
    object this[ int i ] { get; }
    bool GetBoolean(int i);
    byte GetByte(int i);
    char GetChar(int i);
    DateTime GetDateTime(int i);
    Decimal GetDecimal(int i);
    float GetFloat(int i);
    short GetInt16(int i);
    int GetInt32(int i);
    long GetInt64(int i);
    ...
    bool IsDBNull(int i);
}
```

На заметку! Метод `IDataReader.IsDBNull()` позволяет программным способом выяснить, установлено ли конкретное поле в `null`, прежде чем получать значение из объекта чтения данных (во избежание исключения времени выполнения). Также вспомните, что в языке C# поддерживаются типы данных, допускающие `null` (см. главу 4), идеально подходящие для взаимодействия со столбцами таблицы базы данных, которые могут принимать значения `null`.

Абстрагирование поставщиков данных с помощью интерфейсов

К этому моменту вы уже должны лучше разбираться в общей функциональности всех поставщиков данных .NET. И хотя точные имена реализованных типов отличаются в различных поставщиках данных, в коде они используются в схожей манере — и в этом состоит прелесть полиморфизма на основе интерфейсов. К примеру, если определить метод, принимающий параметр `IDbConnection`, то в него можно передать любой объект подключения ADO.NET:

```
public static void OpenConnection(IDbConnection cn)
{
    // Открыть входящее подключение для вызывающего процесса.
    cn.Open();
}
```

На заметку! Использование интерфейсов не обязательно. Аналогичного уровня абстракции можно достигнуть с применением абстрактных базовых классов (таких как `DbConnection`) в качестве параметров или возвращаемых значений.

То же самое относится и к возвращаемым значениям. Рассмотрим, например, следующий простой проект консольного приложения C# (под названием `MyConnectionFactory`), в котором конкретный объект подключения выбирается на основе значения из специального перечисления. В целях диагностики мы просто выводим соответствующий объект подключения с помощью служб рефлексии:

```

using System;
...
// Необходимо для того, чтобы иметь определения общих интерфейсов
// и различные объекты подключения для тестирования.
using System.Data;
using System.Data.SqlClient;
using System.Data.Odbc;
using System.Data.OleDb;

namespace MyConnectionFactory
{
    // Список возможных поставщиков.
    enum DataProvider
    { SqlServer, OleDb, Odbc, None }

    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("**** Very Simple Connection Factory ****\n");

            // Получить конкретное подключение.
            IDbConnection myCn = GetConnection(DataProvider.SqlServer);
            Console.WriteLine("Your connection is a {0}", myCn.GetType().Name);

            // Открыть, использовать и закрыть подключение...
            Console.ReadLine();
        }

        // Этот метод возвращает конкретный объект подключения
        // на основе значения перечисления DataProvider.
        static IDbConnection GetConnection(DataProvider dp)
        {
            IDbConnection conn = null;
            switch (dp)
            {
                case DataProvider.SqlServer:
                    conn = new SqlConnection();
                    break;
                case DataProvider.OleDb:
                    conn = new OleDbConnection();
                    break;
                case DataProvider.Odbc:
                    conn = new OdbcConnection();
                    break;
            }
            return conn;
        }
    }
}

```

Преимущество работы с обобщенными интерфейсами из `System.Data` (или с абстрактными базовыми классами из `System.Data.Common`) состоит в том, что при этом имеется гораздо больше возможностей для создания гибкой кодовой базы, которую со временем можно развивать. Допустим, что сегодня вы создаете приложение для Microsoft SQL Server, но что, если спустя несколько месяцев ваша компания перейдет на другую СУБД? Если в своем решении вы жестко закодировали типы `System.Data.SqlClient`, ориентированные конкретно на Microsoft SQL Server, то понятно, что при изменении СУБД на сервере придется снова выполнять редактирование, компиляцию и развертывание сборки.

Повышение гибкости с помощью конфигурационных файлов приложения

Для повышения гибкости приложений ADO.NET можно использовать клиентский файл *.config, элемент <appSettings> которого может содержать произвольные пары “ключ/значение”. Вспомните из главы 14, что пользовательские данные, хранящиеся в файле *.config, можно получить программным образом с помощью типов из пространства имен System.Configuration. Предположим, например, что в каком-нибудь конфигурационном файле задано следующее значение поставщика данных:

```
<configuration>
  <appSettings>
    <!-- Это значение ключа отображается на одно из значений перечисления -->
    <add key="provider" value="SqlServer"/>
  </appSettings>
</configuration>
```

Теперь можно изменить метод Main(), чтобы программно получить соответствующий поставщик данных. По сути, при этом создается *фабрика объектов подключений*, которая позволяет изменить поставщик без необходимости в повторной компиляции кодовой базы (понадобится лишь модифицировать файл *.config). Ниже приведены необходимые изменения в Main():

```
static void Main(string[] args)
{
    Console.WriteLine("**** Very Simple Connection Factory ****\n");
    // Прочитать ключ provider.
    string dataProvString = ConfigurationManager.AppSettings["provider"];
    // Преобразовать строку в перечисление.
    DataProvider dp = DataProvider.None;
    if (Enum.IsDefined(typeof(DataProvider), dataProvString))
        dp = (DataProvider)Enum.Parse(typeof(DataProvider), dataProvString);
    else
        Console.WriteLine("Sorry, no provider exists!"); // Поставщик не существует.
    // Получить конкретное подключение.
    IDbConnection myCn = GetConnection(dp);
    if (myCn != null)
        Console.WriteLine("Your connection is a {0}", myCn.GetType().Name);
    // Открыть, использовать и закрыть подключение...
    Console.ReadLine();
}
```

На заметку! Для использования типа ConfigurationManager необходимо установить ссылку на сборку System.Configuration.dll и импортировать пространство имен System.Configuration.

К этому моменту построен код ADO.NET, позволяющий динамически указывать нужное подключение. Очевидно, здесь имеется одна проблема: эта абстракция используется только в приложении MyConnectionFactory.exe. Но если этот код оформить в библиотеку кода .NET (например, MyConnectionFactory.dll), то можно будет создавать любое количество клиентов, которые смогут получать различные объекты подключения с помощью уровней абстракции.

Однако получение объекта подключения — лишь один аспект работы с ADO.NET. Чтобы разработать полезную библиотеку фабрик поставщиков данных, необходимо учитывать объекты команд, объекты чтения данных, адаптеры данных, объекты транзакций и другие типы, ориентированные на работу с данными.

Создание такой библиотеки кода не обязательно будет трудным, но все же потребует существенного объема кодирования и времени.

Начиная с выпуска .NET 2.0, эта возможность встроена непосредственно в библиотеки базовых классов .NET. Ниже мы рассмотрим этот формальный API-интерфейс, но вначале понадобится создать собственную базу данных для работы как в этой, так и во многих последующих главах.

Исходный код. Проект MyConnectionFactory доступен в подкаталоге Chapter 21.

Создание базы данных AutoLot

Далее в этой главе мы будем запускать запросы к простой тестовой базе данных SQL Server по имени AutoLot. В продолжение все той же автомобильной темы, эта база данных будет содержать три взаимосвязанных таблицы (Inventory, Orders и Customers), хранящих различные данные о заказах гипотетической компании по продаже автомобилей.

В книге предполагается наличие копии Microsoft SQL Server (7.0 или последующей версии) или Microsoft SQL Server Express Edition. Если это не так, загрузите копию со следующей страницы:

www.microsoft.com/sqlserver/en/us/editions/2012-editions/express.aspx

Этот облегченный сервер баз данных отлично подходит для наших потребностей: он бесплатен, предоставляет графический пользовательский интерфейс (SQL Server Management Tool) для создания и администрирования баз данных и интегрирован с Visual Studio/Visual C# Express Edition.

В целях иллюстрации последнего утверждения остаток этого раздела будет посвящен созданию базы данных AutoLot с применением Visual Studio. Если вы пользуетесь Visual C# Express Edition, то сможете выполнить аналогичные действия в окне проводника баз данных (который открывается через пункт меню View⇒Other Windows (Вид⇒Другие окна)).

На заметку! База данных AutoLot будет использоваться в оставшейся части книги.

Создание таблицы Inventory

Чтобы приступить к созданию тестовой базы данных, запустите Visual Studio и откройте окно Server Explorer через меню View (Просмотр). Затем щелкните правой кнопкой мыши на узле Data Connections (Подключения к данным) и выберите в контекстном меню пункт Create New SQL Server Database (Создать новую базу данных SQL Server), как показано на рис. 21.3.

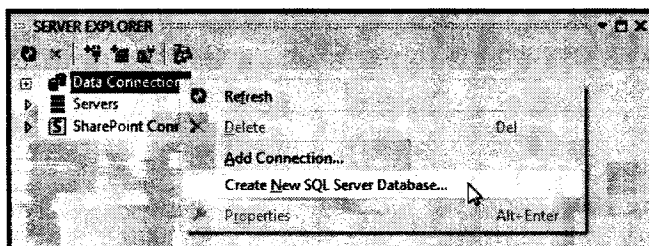


Рис. 21.3. Создание новой базы данных SQL Server внутри Visual Studio

В открывшемся диалоговом окне необходимо ввести значение в текстовом поле **Server name** (Имя сервера), которое представляет компьютер, где будет создана база данных. Если на вашем компьютере установлена полная версия Microsoft SQL Server, введите `(local)`, чтобы создать базу данных на локальной машине. Однако если у вас установлена версия Microsoft SQL Express, введите `(local)\SQLEXPRESS`.

Назовите созданную базу данных **AutoLot** и оставьте выбранным переключатель **Use Windows Authentication** (Использовать аутентификацию Windows), как показано на рис. 21.4.

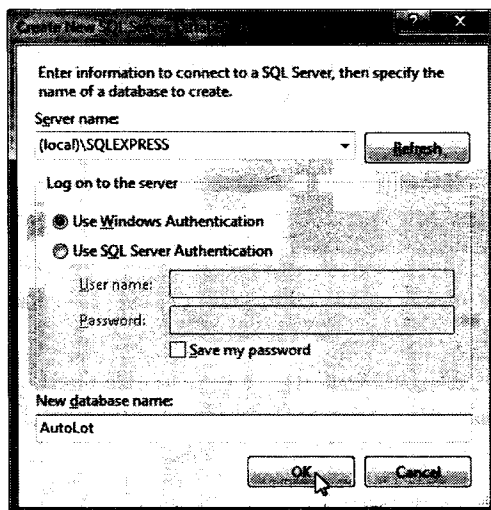


Рис. 21.4. Создание новой базы данных SQL Server Express с помощью Visual Studio

В этот момент база данных **AutoLot** не содержит никаких объектов (таблиц, хранимых процедур и т.п.). Для добавления новой таблицы щелкните правой кнопкой мыши на узле **Tables** (Таблицы) и выберите в контекстном меню пункт **Add New Table** (рис. 21.5).

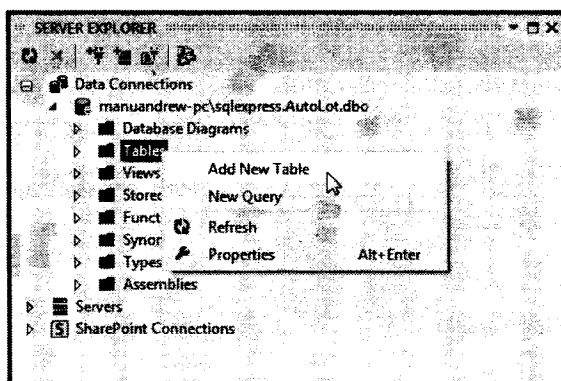


Рис. 21.5. Добавление таблицы **Inventory**

С помощью редактора таблиц добавьте в таблицу четыре столбца данных — **CarID** (Идентификатор автомобиля), **Make** (Марка), **Color** (Цвет) и **PetName** (Дружественное имя) — все типа `varchar(50)`. Удостоверьтесь, что столбец **CarID** установлен в качестве первичного ключа, для чего щелкните правой кнопкой мыши на строке **CarID** и выберите в контекстном меню пункт **Set Primary Key** (Установить первичный ключ).

Также обратите внимание, что всем столбцам, кроме CarID, могут быть присвоены значения null. Окончательные параметры таблицы показаны на рис. 21.6. В редакторе Column Properties (Свойства столбца) ничего делать не надо, просто запомните типы данных для каждого столбца.

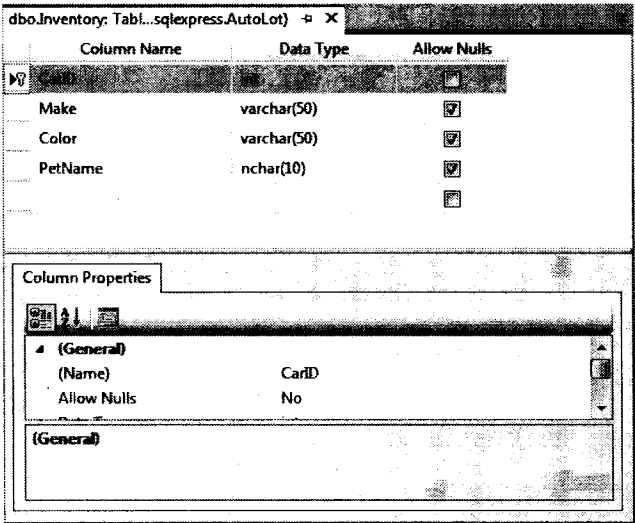


Рис. 21.6. Структура таблицы Inventory

После создания схемы таблицы сохраните ее с помощью комбинации клавиш <Ctrl+S> или пункта меню File⇒Save (Файл⇒Сохранить). При этом появится запрос имени новой таблицы — назовите ее Inventory. Теперь таблица Inventory должна быть видна под узлом Tables (Таблицы) в окне Server Explorer.

Занесение тестовых записей в таблицу Inventory

Для добавления записей в эту первую таблицу щелкните правой кнопкой мыши на ее значке и выберите в контекстном меню пункт Show Table Data (Показать данные таблицы). Введите информацию о нескольких новых автомобилях по своему усмотрению (чтобы было интереснее, пусть у некоторых автомобилей совпадают цвета и марки). Один из возможных вариантов списка товаров приведен на рис. 21.7.

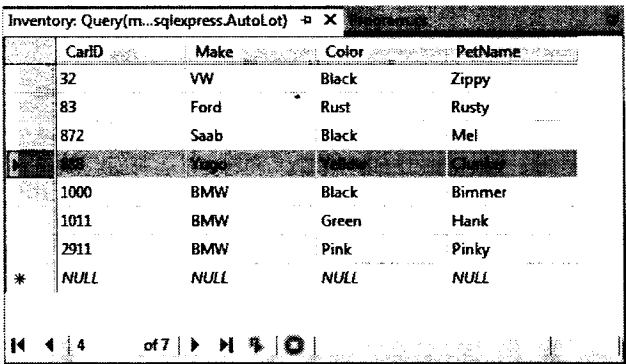


Рис. 21.7. Заполнение таблицы Inventory

Создание хранимой процедуры GetPetName ()

Позже в этой главе будет показано, как вызывать хранимые процедуры в ADO.NET. Возможно, вы уже знаете, что хранимые процедуры — это подпрограммы, хранящиеся непосредственно в базе данных; обычно они работают с данными таблиц и возвращают какое-то значение. Мы добавим в базу данных одну хранимую процедуру, которая по идентификатору автомобиля будет возвращать его дружественное имя. Для этого щелкните правой кнопкой мыши на узле Stored Procedures (Хранимые процедуры) базы данных AutoLot в Server Explorer и выберите в контекстном меню пункт Add New Stored Procedure (Добавить новую хранимую процедуру). В появившемся окне редактора введите следующий текст:

```
CREATE PROCEDURE GetPetName
@carID int,
@petName char(10) output
AS
SELECT @petName = PetName from Inventory where CarID = @carID
```

На заметку! Хранимые процедуры не обязательно должны возвращать данные через выходные параметры, как сделано здесь; однако это пригодится, когда речь пойдет о свойстве Direction объектов SqlParameter далее в главе.

При сохранении эта процедура автоматически получит имя GetPetName, взятое из оператора CREATE PROCEDURE (учтите, что при первом сохранении Visual Studio автоматически изменяет имя SQL-сценария на ALTER PROCEDURE...). После этого новая хранимая процедура будет видна в Server Explorer (рис. 21.8).

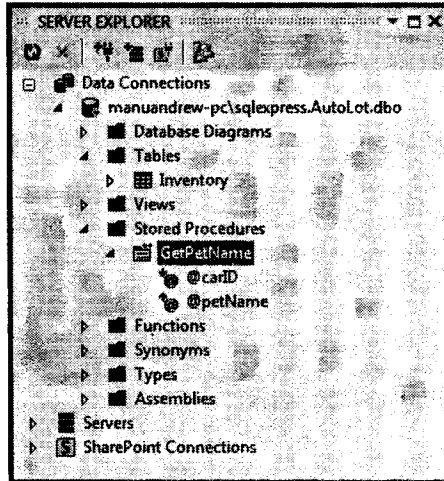


Рис. 21.8. Хранимая процедура GetPetName

Создание таблиц Customers и Orders

В базе данных AutoLot будут находиться еще две таблицы: Customers (Клиенты) и Orders (Заказы). Таблица Customers будет содержать список клиентов и состоять из трех столбцов: CustID (Идентификатор клиента; должен быть первичным ключом), FirstName (Имя) и LastName (Фамилия). Повторите шаги, которые были выполнены для создания таблицы Inventory, и создайте таблицу Customers, пользуясь схемой, приведенной на рис. 21.9.

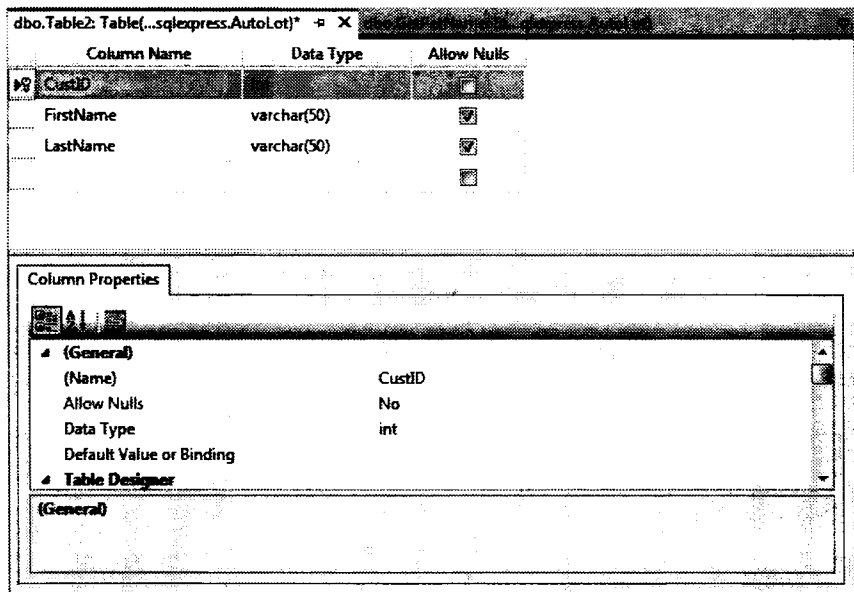


Рис. 21.9. Структура таблицы Customers

После сохранения и именования этой таблицы добавьте в нее несколько записей (рис. 21.10).

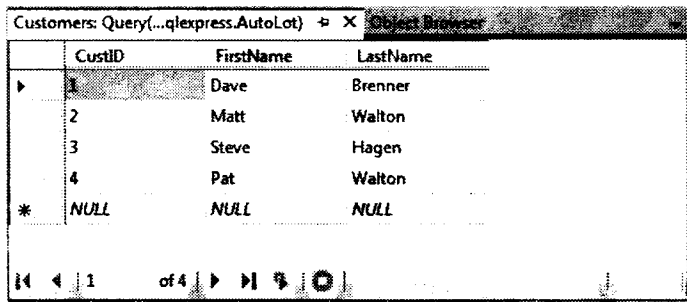


Рис. 21.10. Заполнение таблицы Customers

Последняя таблица — Orders — предназначена для связи клиентов и интересующих их автомобилей. Для этого осуществляется отображение значений OrderID на значения CarID/CustID. Структура таблицы Orders показана на рис. 21.11 (обратите внимание, что OrderID является первичным ключом).

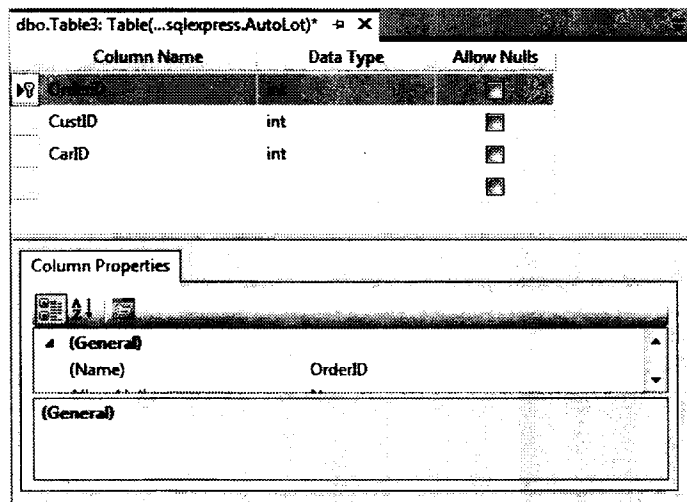


Рис. 21.11. Структура таблицы Orders

Теперь добавьте в таблицу `Orders` данные. Предполагая, что значения `OrderID` начинаются с 1000, выберите для каждого значения `CustID` уникальное значение `CarID` (рис. 21.12).

OrderID	CustID	CarID
1000	1	1000
1001	2	32
1002	3	888
1003	4	2911
***	NULL	NULL

Рис. 21.12. Заполнение таблицы `Orders`

Например, в соответствии с информацией, приведенной на рисунках, видно, что Дэйв Бреннер (Dave Brenner, `CustID` = 1) мечтает о черном BMW (`CarID` = 1000), а Пэт Уолтон (Pat Walton, `CustID` = 4) приглянулся розовый Saab (`CarID` = 2911).

Визуальное создание отношений между таблицами

И, наконец, между таблицами `Customers`, `Orders` и `Inventory` нужно установить отношения “родительский–дочерний”. В Visual Studio это делается очень просто, т.к. на этапе проектирования можно вставить новую диаграмму базы данных. Для этого откройте `Server Explorer`, щелкните правой кнопкой мыши на узле `Database Diagrams` (Диаграммы базы данных) базы `AutoLot` и выберите пункт контекстного меню `Add New Diagram` (Добавить новую диаграмму). Откроется диалоговое окно, в котором можно выбирать таблицы и добавлять их в диаграмму. Выберите все таблицы из базы данных `AutoLot`, как показано на рис. 21.13.

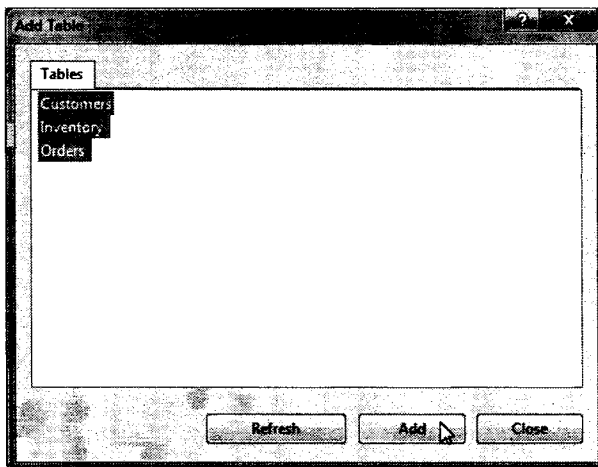


Рис. 21.13. Выбор таблиц для диаграммы

Чтобы приступить к установке отношений между таблицами, щелкните на ключе `CardID` таблицы `Inventory`, а затем (не отпуская кнопку мыши) перетащите его на поле `CardID` таблицы `Orders`. Когда вы отпустите кнопку, появится диалоговое окно; согласитесь со всеми предложенными в нем значениями по умолчанию.

Теперь повторите те же действия для отображения ключа CustID таблицы Customers на поле CustID таблицы Orders. После этого вы увидите диалоговое окно классов, показанное на рис. 21.14.

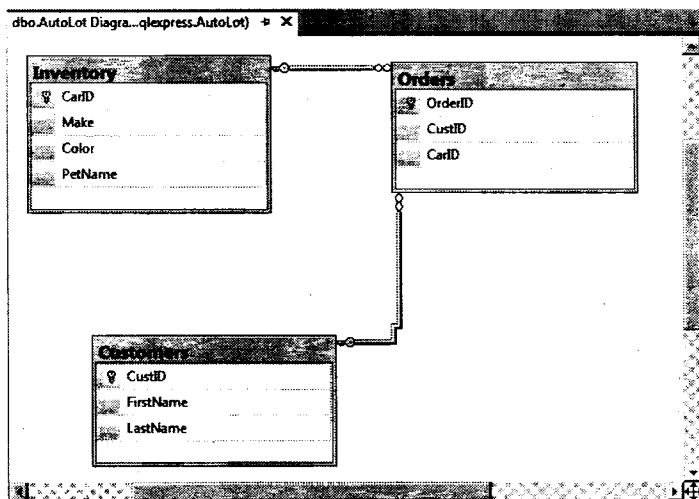


Рис. 21.14. Отношения между таблицами Orders, Inventory и Customers

На этом построение базы данных AutoLot завершено. Конечно, это лишь бледное подобие реальных корпоративных баз данных, но она отлично послужит нам до конца книги. Теперь, имея тестовую базу, можно начать подробно разбираться в модели фабрики поставщиков данных ADO.NET.

Модель фабрик поставщиков данных ADO.NET

Фабрика поставщиков данных .NET позволяет создать единую кодовую базу с помощью обобщенных типов доступа к данным. Более того, посредством конфигурационных файлов приложения (и подэлемента <connectionStrings>) можно получить поставщики и строки соединения декларативным образом, без необходимости в повторной компиляции или развертывания сборки, в которой используются API-интерфейсы ADO.NET.

Чтобы разобраться в реализации фабрики поставщиков данных, вспомните из табл. 21.1, что все классы в поставщике данных являются производными от одних и тех же базовых классов, определенных в пространстве имен System.Data.Common:

- DbCommand — абстрактный базовый класс для всех классов команд;
- DbConnection — абстрактный базовый класс для всех классов подключений;
- DbDataAdapter — абстрактный базовый класс для всех классов адаптеров данных;
- DbDataReader — абстрактный базовый класс для всех классов чтения данных;
- DbParameter — абстрактный базовый класс для всех классов параметров;
- DbTransaction — абстрактный базовый класс для всех классов транзакций.

Все поставщики данных, разработанные Microsoft, содержат класс, производный от System.Data.Common.DbProviderFactory. В этом базовом классе определен ряд методов, которые выбирают объекты данных, характерные для конкретных поставщиков. Вот фрагмент с членами DbProviderFactory:

```
public abstract class DbProviderFactory
{
    ...
    public virtual DbCommand CreateCommand();
    public virtual DbCommandBuilder CreateCommandBuilder();
    public virtual DbConnection CreateConnection();
    public virtual DbConnectionStringBuilder CreateConnectionStringBuilder();
    public virtual DbDataAdapter CreateDataAdapter();
    public virtual DbDataSourceEnumerator CreateDataSourceEnumerator();
    public virtual DbParameter CreateParameter();
}
```

Для получения типа, производного от `DbProviderFactory`, непосредственно для вашего поставщика данных в пространстве имен `System.Data.Common` имеется класс `DbProviderFactories`. С помощью метода `GetFactory()` можно получить конкретный объект `DbProviderFactory` для указанного поставщика данных. Для этого нужно указать строковое имя, которое представляет пространство имен .NET, содержащее функциональность поставщика:

```
static void Main(string[] args)
{
    // Получить фабрику для поставщика данных SQL.
    DbProviderFactory sqlFactory =
        DbProviderFactories.GetFactory("System.Data.SqlClient");
    ...
}
```

Разумеется, фабрику можно получить не с помощью жестко закодированного строкового литерала, а, например, прочесть эту информацию из клиентского файла `*.config` (приблизительно так же, как в предыдущем примере с `MyConnectionFactory`). Вскоре вы увидите, как это сделать, а пока после получения фабрики для поставщика данных можно получить связанные с ним объекты данных (например, подключения, команды и объекты чтения данных).

На заметку! Для всех практических целей аргумент, передаваемый в `DbProviderFactories.GetFactory()`, можно рассматривать как название пространства имен .NET для поставщика данных. В реальности это строковое значение используется в `machine.config` для динамической загрузки корректной библиотеки из глобального кеша сборок.

Полный пример фабрики поставщиков данных

Для примера мы создадим новое консольное приложение C# по имени `DataProviderFactory`, которое выводит список всех автомобилей из базы данных `AutoLot`. Поскольку это первый пример, мы жестко закодируем логику доступа к данным непосредственно в сборке `DataProviderFactory.exe` (чтобы пока не усложнять программирование ADO.NET, логику данных будет изолирована в специальной библиотеке кода .NET, которая будет использоваться на протяжении всего оставшегося текста).

Вначале добавьте ссылку на сборку `System.Configuration.dll` и импортируйте пространство имен `System.Configuration`. Затем добавьте файл `App.config` в текущий проект и определите пустой элемент `<appSettings>`. Добавьте новый ключ `provider`, который отображается на пространство имен поставщика данных, который требуется получить (`System.Data.SqlClient`). Кроме того, определите строку соединения, которая описывает подключение к базе данных `AutoLot` (на локальном экземпляре `SQL Server Express`):

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <!-- Поставщик -->
    <add key="provider" value="System.Data.SqlClient" />
    <!-- Строка соединения -->
    <add key="cnStr" value="Data Source=(local)\SQLEXPRESS;
      Initial Catalog=AutoLot;Integrated Security=True"/>
  </appSettings>
</configuration>
```

На заметку! Чуть ниже мы рассмотрим строки соединения подробнее. А пока учтите, что если вы брать в окне Server Explorer значок базы данных AutoLot, то можно скопировать и вставить правильную строку соединения из свойства Connection String (Строка соединения) в окне Properties (Свойства) среды Visual Studio.

Теперь у вас есть корректный файл *.config, и вы можете прочитать значения provider и cnStr с помощью индексатора ConfigurationManager.AppSettings. Значение provider будет передано методу DbProviderFactories.GetFactory(), чтобы получить тип фабрики для необходимого поставщика данных. Значение cnStr будет использовано для установки свойства ConnectionString в типе, производном от DbConnection. Предполагая, что были импортированы пространства имен System.Data и System.Data.Common, метод Main() можно модифицировать следующим образом:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Data Provider Factories *****\n");

    // Получить строку соединения и поставщика из файла *.config.
    string dp = ConfigurationManager.AppSettings["provider"];
    string cnStr = ConfigurationManager.AppSettings["cnStr"];

    // Получить фабрику поставщиков.
    DbProviderFactory df = DbProviderFactories.GetFactory(dp);

    // Получить объект подключения.
    using (DbConnection cn = df.CreateConnection())
    {
        Console.WriteLine("Your connection object is a: {0}", cn.GetType().Name);
        cn.ConnectionString = cnStr;
        cn.Open();

        // Создать объект команды.
        DbCommand cmd = df.CreateCommand();
        Console.WriteLine("Your command object is a: {0}", cmd.GetType().Name);
        cmd.Connection = cn;
        cmd.CommandText = "Select * From Inventory";

        // Вывести данные с помощью объекта чтения данных.
        using (DbDataReader dr = cmd.ExecuteReader())
        {
            Console.WriteLine("Your data reader object is a: {0}", dr.GetType().Name);
            Console.WriteLine("\n***** Current Inventory *****");
            while (dr.Read())
            {
                Console.WriteLine("-> Car #{0} is a {1}.",
                    dr["CarID"], dr["Make"].ToString());
            }
        }
    }
    Console.ReadLine();
}
```

Здесь, в целях диагностики с помощью службы рефлексии выводятся полностью определенные имена соответствующих объектов подключения, команды и чтения данных. Если запустить это приложение, то на консоль будут выведены текущие данные из таблицы Inventory базы данных AutoLot:

```
***** Fun with Data Provider Factories *****
```

```
Your connection object is a: SqlConnection
```

```
Your command object is a: SqlCommand
```

```
Your data reader object is a: SqlDataReader
```

```
***** Current Inventory *****
```

```
-> Car #32 is a VW.
```

```
-> Car #83 is a Ford.
```

```
-> Car #872 is a Saab.
```

```
-> Car #888 is a Yugo.
```

```
-> Car #1000 is a BMW.
```

```
-> Car #1011 is a BMW.
```

```
-> Car #2911 is a BMW.
```

Теперь измените содержимое файла *.config, указав в качестве поставщика данных System.Data.OleDb (и модифицируйте строку соединения и сегмент Provider):

```
<configuration>
  <appSettings>
    <!-- Поставщик -->
    <add key="provider" value="System.Data.OleDb" />

    <!-- Строка соединения -->
    <add key="cnStr" value=
      "Provider=SQLOLEDB;Data Source=(local)\SQLEXPRESS;
       Integrated Security=SSPI;Initial Catalog=AutoLot"/>
  </appSettings>
</configuration>
```

Вы обнаружите, что “за кулисами” использовались типы System.Data.OleDb; вывод в этом случае будет таким:

```
***** Fun with Data Provider Factories *****
```

```
Your connection object is a: OleDbConnection
```

```
Your command object is a: OleDbCommand
```

```
Your data reader object is a: OleDbDataReader
```

```
***** Current Inventory *****
```

```
-> Car #32 is a VW.
```

```
-> Car #83 is a Ford.
```

```
-> Car #872 is a Saab.
```

```
-> Car #888 is a Yugo.
```

```
-> Car #1000 is a BMW.
```

```
-> Car #1011 is a BMW.
```

```
-> Car #2911 is a BMW.
```

Конечно, при недостатке опыта работы с ADO.NET вы можете не быть полностью осведомлены о том, что именно *делают* объекты подключения, команды и чтения данных. Не вдаваясь в детали, пока просто уясните, что модель фабрик поставщиков данных ADO.NET позволяет создать единую кодовую базу, которая может использовать разнообразные поставщики данных в декларативной манере.

Потенциальный недостаток модели фабрик поставщиков данных

Хотя это действительно мощная модель, все же нужно удостовериться, что в кодовой базе используются только типы и методы, которые являются общими для всех постав-

щиков благодаря членам абстрактных базовых классов. Следовательно, при разработке кодовой базы вы ограничены членами `DbConnection`, `DbCommand` и других типов из пространства имен `System.Data.Common`.

С учетом этого, вы можете обнаружить, что такой обобщенный подход предотвращает прямой доступ к дополнительным возможностям конкретной СУБД. Если все же потребуются вызовы специфических членов конкретного поставщика (например, `SqlConnection`), то это можно сделать с помощью явного приведения, как показано ниже:

```
using (DbConnection cn = df.CreateConnection())
{
    Console.WriteLine("Your connection object is a: {0}", cn.GetType().Name);
    cn.ConnectionString = cnStr;
    cn.Open();
    if (cn is SqlConnection)
    {
        // Вывести информацию об используемой версии SQL Server.
        Console.WriteLine(((SqlConnection)cn).ServerVersion);
    }
    ...
}
```

Однако при этом сопровождение кодовой базы несколько затруднится (а гибкость снизится), поскольку придется еще добавить ряд проверок времени выполнения. Тем не менее, если необходимо построить библиотеки доступа к данным наиболее гибким способом, то модель фабрик поставщиков данных предоставляет для этого замечательный механизм.

Элемент <connectionStrings>

В настоящий момент наша строка соединения находится в элементе <appSettings> файла `*.config`. В конфигурационных файлах приложения может быть определен элемент <connectionStrings>. В этом элементе можно задать любое количество пар "имя/значение", которые будут программно прочитаны в память с помощью индексатора `ConfigurationManager.ConnectionStrings`. Одно из преимуществ такого подхода (по сравнению с использованием элемента <appSettings> и индексатора `ConfigurationManager.AppSettings`) — возможность определения нескольких строк соединения для одного приложения в согласованной манере.

Чтобы увидеть все это в действии, модифицируйте текущий файл `App.config` следующим образом (обратите внимание, что каждая строка соединения описана с помощью атрибутов `name` и `connectionString`, а не `key` и `value`, как в <appSettings>):

```
<configuration>
  <appSettings>
    <!-- Поставщик -->
    <add key="provider" value="System.Data.SqlClient" />
  </appSettings>
  <!-- Строки соединения -->
  <connectionStrings>
    <add name="AutoLotSqlProvider" connectionString =
      "Data Source=(local)\SQLEXPRESS;
      Integrated Security=SSPI;Initial Catalog=AutoLot"/>
    <add name="AutoLotOleDbProvider" connectionString =
      "Provider=SQLOLEDB;Data Source=(local)\SQLEXPRESS;
      Integrated Security=SSPI;Initial Catalog=AutoLot"/>
  </connectionStrings>
</configuration>
```

Теперь можно модифицировать метод `Main()`:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Data Provider Factories *****\n");
    string dp =
        ConfigurationManager.AppSettings["provider"];
    string cnStr =
        ConfigurationManager.ConnectionStrings["AutoLotSqlProvider"].ConnectionString;
    ...
}
```

К этому моменту построено приложение, которое может отображать содержимое таблицы `Inventory` базы данных `AutoLot`, используя нейтральную кодовую базу. Вывос имени поставщика и строки соединения во внешний файл `*.config` позволяет модели фабрик поставщиков данных самостоятельно динамически загружать нужный поставщик. На этом первый пример завершен, и теперь можно углубиться в детали работы с подключенным уровнем ADO.NET.

На заметку! Теперь, когда вы понимаете роль фабрик поставщиков данных ADO.NET, в оставшихся примерах этой книги внимание будет сосредоточено на текущих задачах за счет явного использования типов из пространства имен `System.Data.SqlClient`. Если вы работаете с другой СУБД (например, `Oracle`), то понадобится соответствующим образом изменить кодовую базу.

Исходный код. Проект `DataProviderFactory` доступен в подкаталоге `Chapter 21`.

Понятие подключенного уровня ADO.NET

Вспомните, что *подключенный уровень* ADO.NET позволяет взаимодействовать с базой данных с помощью объектов подключения, чтения данных и команд конкретного поставщика данных. Вы уже использовали эти объекты в предыдущем приложении `DataProviderFactory`, но все же мы рассмотрим весь процесс еще раз на расширенном примере. Чтобы подключиться к базе данных и прочитать записи с помощью объекта чтения данных, необходимо выполнить следующие шаги.

1. Создать, сконфигурировать и открыть объект подключения.
2. Создать и сконфигурировать объект команды, указав объект подключения в качестве аргумента конструктора или с помощью свойства `Connection`.
3. Вызвать метод `ExecuteReader()` сконфигурированного объекта команды.
4. Обработать каждую запись с помощью метода `Read()` объекта чтения данных.

Итак, для начала создайте новое консольное приложение по имени `AutoLotDataReader` и импортируйте пространства имен `System.Data` и `System.Data.SqlClient`. Ниже приведен полный код метода `Main()` с последующим анализом:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with Data Readers *****\n");

        // Создать и открыть подключение.
        using(SqlConnection cn = new SqlConnection())
        {
```



```

cn.ConnectionString =
    @"Data Source=(local)\SQLEXPRESS;Integrated Security=SSPI;" +
    "Initial Catalog=AutoLot";
cn.Open();

// Создать объект команды SQL.
string strSQL = "Select * From Inventory";
SqlCommand myCommand = new SqlCommand(strSQL, cn);

// Получить объект чтения данных с помощью ExecuteReader().
using(SqlDataReader myDataReader = myCommand.ExecuteReader())
{
    // Организовать цикл по результатам.
    while (myDataReader.Read())
    {
        Console.WriteLine("-> Make: {0}, PetName: {1}, Color: {2}.",
            myDataReader["Make"].ToString(),
            myDataReader["PetName"].ToString(),
            myDataReader["Color"].ToString());
    }
}
Console.ReadLine();
}
}

```

Работа с объектами подключения

Первое, что нужно сделать при работе с поставщиком данных — это установить сеанс с источником данных с помощью объекта подключения (производного, как вы помните, от `DbConnection`). Объекты подключения .NET снабжаются форматированной строкой соединения, которая содержит ряд пар “имя/значение”, разделенных точками с запятой. Эта информация содержит имя машины, к которой нужно подключиться, необходимые параметры безопасности, имя базы данных на этой машине и другую информацию, специфичную для поставщика.

Из приведенного выше кода можно понять, что имя `Initial Catalog` относится к базе данных, с которой нужно установить сеанс. Имя `Data Source` определяет имя машины, на которой расположена база данных. Элемент `(local)` позволяет указать текущую локальную машину (независимо от конкретного имени этой машины), а элемент `\SQLEXPRESS` сообщает поставщику SQL Server, что вы подключаетесь к стандартной установке SQL Server Express (если база данных `AutoLot` создавалась с помощью полной версии Microsoft SQL Server на локальном компьютере, необходимо указать `Data Source=(local)`).

Кроме того, можно задать любое количество элементов, которые представляют полномочия безопасности. Здесь `Integrated Security` устанавливается в `SSPI` (что эквивалентно `true`), что означает использование для аутентификации пользователей текущих полномочий учетной записи Windows.

На заметку! Смысл каждой пары “имя/значение” для специфичной СУБД можно узнать в описании свойства `ConnectionString` объекта подключения для вашего поставщика данных внутри документации .NET Framework 4.5 SDK.

При наличии строки соединения вызов метода `Open()` устанавливает соединение с СУБД. В дополнение к членам `ConnectionString`, `Open()` и `Close()` объект подключения содержит несколько членов, которые позволяют настроить дополнительные параметры подключения, например, время таймаута и информацию, относящуюся к транзакциям. В табл. 21.5 кратко описаны избранные члены базового класса `DbConnection`.

Таблица 21.5. Члены типа `DbConnection`

Член	Описание
<code>BeginTransaction()</code>	Этот метод позволяет начать транзакцию базы данных
<code>ChangeDatabase()</code>	Этот метод изменяет базу данных, связанную с открытым подключением
<code>ConnectionTimeout</code>	Это свойство только для чтения возвращает время ожидания при установке подключения, после которого ожидание прекращается и выдается сообщение об ошибке (стандартное значение составляет 15 секунд). Для изменения этого времени нужно указать в строке соединения сегмент <code>Connect Timeout</code> (например, <code>Connect Timeout=30</code>)
<code>Database</code>	Это свойство только для чтения возвращает имя базы данных, обслуживаемой объектом подключения
<code>DataSource</code>	Это свойство только для чтения возвращает местоположение базы данных, обслуживаемой объектом подключения
<code>GetSchema()</code>	Этот метод возвращает объект <code>DataTable</code> , содержащий информацию схемы из источника данных
<code>State</code>	Это свойство только для чтения возвращает текущее состояние подключения, представленное перечислением <code>ConnectionState</code>

Свойства типа `DbConnection` обычно предназначены только для чтения и полезны, только когда требуется получить характеристики подключения во время выполнения. Для переопределения стандартных значений понадобится изменить саму строку соединения. Например, вот как поменять время таймаута с 15 на 30 секунд:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Data Readers *****\n");
    using(SqlConnection cn = new SqlConnection())
    {
        cn.ConnectionString =
            @"Data Source=(local)\SQLEXPRESS;" +
            "Integrated Security=SSPI;Initial Catalog=AutoLot;Connect Timeout=30";
        cn.Open();

        // Новая вспомогательная функция (см. ниже).
        ShowConnectionStatus(cn);
        ...
    }
}
```

В показанном выше коде объект подключения передается в качестве параметра новому статическому вспомогательному методу `ShowConnectionStatus()` класса `Program`, который реализован следующим образом:

```
static void ShowConnectionStatus(SqlConnection cn)
{
    // Вывести различные сведения о текущем объекте подключения.
    Console.WriteLine("***** Info about your connection *****");
    Console.WriteLine("Database location: {0}", cn.DataSource);
        // Местоположение базы данных
    Console.WriteLine("Database name: {0}", cn.Database);           // Имя базы данных
    Console.WriteLine("Timeout: {0}", cn.ConnectionTimeout);      // Таймаут
    Console.WriteLine("Connection state: {0}\n", cn.State.ToString()); // Состояние
}
```

Хотя большинство этих свойств должно быть понятно без объяснений, свойство `State` требует дополнительного внимания. Этому свойству можно присвоить любое значение из показанного ниже перечисления `ConnectionState`:

```
public enum ConnectionState
{
    Broken, Closed,
    Connecting, Executing,
    Fetching, Open
}
```

Тем не менее, допустимыми значениями `ConnectionState` являются только `ConnectionState.Open` и `ConnectionState.Closed` (остальные члены этого перечисления зарезервированы для будущего использования). Кроме того, всегда безопасно закрывать подключение, когда его состояние соответствует `ConnectionState.Closed`.

Работа с объектами `ConnectionStringBuilder`

Программная работа со строками соединения может оказаться несколько затруднительной, поскольку они часто представлены в виде строковых литералов, которые трудно обрабатывать и контролировать на наличие ошибок. Поставщики данных ADO.NET, разработанные Microsoft, поддерживают *объекты построителей строк соединения*, которые позволяют устанавливать пары "имя/значение" с помощью строго типизированных свойств. Рассмотрим следующую модификацию текущего метода `Main()`:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Data Readers *****\n");

    // Создать строку соединения с помощью объекта построителя.
    SqlConnectionStringBuilder cnStrBuilder =
        new SqlConnectionStringBuilder();

    cnStrBuilder.InitialCatalog = "AutoLot";
    cnStrBuilder.DataSource = @"(local)\SQLEXPRESS";
    cnStrBuilder.ConnectTimeout = 30;
    cnStrBuilder.IntegratedSecurity = true;

    using(SqlConnection cn = new SqlConnection())
    {
        cn.ConnectionString = cnStrBuilder.ConnectionString;
        cn.Open();

        ShowConnectionStatus(cn);
        ...
    }
    Console.ReadLine();
}
```

В этой версии метода `Main()` создается экземпляр `SqlConnectionStringBuilder`, устанавливаются его свойства, и с помощью свойства `ConnectionString` извлекается внутренняя строка. Обратите внимание, что здесь используется стандартный конструктор типа. При желании можно также создать экземпляр объекта построителя для строки соединения поставщика данных, передав в качестве отправной точки существующую строку соединения (это может оказаться удобным при динамическом чтении значений из файла `App.config`). После наполнения объекта начальными строковыми данными можно изменить отдельные пары "имя/значение" с помощью соответствующих свойств, как показано в следующем примере:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Data Readers *****\n");
    // Предположим, что значение cnStr действительно получено из файла *.config.
    string cnStr = @"Data Source=(local)\SQLEXPRESS;" +
        "Integrated Security=SSPI;Initial Catalog=AutoLot";
    SqlConnectionStringBuilder cnStrBuilder =
        new SqlConnectionStringBuilder(cnStr);
    // Изменить значение таймаута.
    cnStrBuilder.ConnectTimeout = 5;
    ...
}
```

Работа с объектами команд

Теперь, когда прояснилась роль объекта подключения, можно посмотреть, каким образом отправлять SQL-запросы в базу данных. Тип `SqlCommand` (производный от `DbCommand`) — это объектно-ориентированное представление SQL-запроса, имени таблицы или хранимой процедуры. Тип команды указывается с помощью свойства `CommandType`, которое принимает любое значение из следующего перечисления `CommandType`:

```
public enum CommandType
{
    StoredProcedure,
    TableDirect,
    Text // Стандартное значение.
}
```

При создании объекта команды можно установить SQL-запрос, передав его в качестве параметра конструктора или напрямую через свойство `CommandText`. Кроме того, при создании объекта команды необходимо указать используемое подключение. Это также можно сделать в виде параметра конструктора либо с помощью свойства `Connection`. Взгляните на следующий фрагмент кода:

```
// Создать объект команды с помощью аргументов конструктора.
string strSQL = "Select * From Inventory";
SqlCommand myCommand = new SqlCommand(strSQL, cn);

// Создать еще один объект команды с помощью свойств.
SqlCommand testCommand = new SqlCommand();
testCommand.Connection = cn;
testCommand.CommandText = strSQL;
```

Учтите, что в этот момент SQL-запрос еще не отправляется в базу данных `AutoLot`, а вместо этого лишь подготавливается состояние объекта команды для дальнейшего использования. В табл. 21.6 описаны некоторые дополнительные члены типа `DbCommand`.

Таблица 21.6. Члены типа `DbCommand`

Член	Описание
<code>CommandTimeout</code>	Возвращает или устанавливает время ожидания при выполнении команды до прекращения попытки и генерации ошибки. Стандартное значение составляет 30 секунд
<code>Connection</code>	Возвращает или устанавливает объект <code>DbConnection</code> , используемый текущим объектом <code>DbCommand</code>
<code>Parameters</code>	Возвращает коллекцию типов <code>DbParameter</code> , используемых для параметризованного запроса

Член	Описание
Cancel ()	Отменяет выполнение команды
ExecuteReader ()	Выполняет SQL-запрос и возвращает объект DbDataReader поставщика данных, которые предоставляет доступ к результату запроса только для чтения в прямом направлении
ExecuteNonQuery ()	Выполняет SQL-оператор, отличный от запроса (например, вставка, обновление, удаление или создание таблицы)
ExecuteScalar ()	Облегченная версия метода ExecuteReader (), созданная специально для одноэлементных запросов (наподобие получения количества записей)
Prepare ()	Создает подготовленную (или скомпилированную) версию команды в источнике данных. Как известно, <i>подготовленный запрос</i> выполняется несколько быстрее и удобен, когда требуется многократное выполнение одного и того же запроса (обычно каждый раз с различными параметрами)

Работа с объектами чтения данных

После установки подключения и создания объекта команды можно отправлять запросы к источнику данных. Это делается несколькими способами. Самым простым и быстрым способом получения информации из хранилища данных является тип `DbDataReader` (реализующий интерфейс `IDataReader`). Вспомните, что объекты чтения данных представляют поток данных, допускающий только чтение в прямом направлении, и возвращают каждый раз по одной записи. Поэтому объекты чтения данных применяются только для отправки SQL-операторов выборки информации из хранилища данных.

Объекты чтения данных удобны, когда нужно быстро просмотреть большой объем данных без необходимости их представления в памяти. Например, если запросить из таблицы 20 000 записей для сохранения их в текстовом файле, то помещение этой информации в `DataSet` будет излишней затратой памяти (поскольку `DataSet` хранит полный результат запроса в памяти).

Гораздо лучше создать объект чтения данных, который будет максимально быстро выдавать по одной записи. Учтите, что объекты чтения данных (в отличие от объектов адаптеров данных, которые рассматриваются ниже) поддерживают открытое подключение к источнику данных, пока сеанс не будет явно закрыт.

Объект чтения данных можно получить из объекта команды с помощью вызова `ExecuteReader ()`. Объект чтения данных представляет текущую запись, прочитанную из базы данных. В нем имеется метод индексатора (например, синтаксис `[]` в C#), который обеспечивает доступ к столбцам текущей записи. Доступ к конкретному столбцу возможен либо по имени, либо по целочисленному индексу, начинающемуся с нуля.

В приведенном ниже примере использования объекта чтения данных применяется метод `Read ()`, позволяющий определить, когда записи закончились (в этом случае он возвращает значение `false`). Для каждой прочитанной из базы данных записи с помощью индексатора типа выводится марка, дружественное имя и цвет каждого автомобиля. Обратите внимание, что сразу после завершения обработки записей вызывается метод `Close ()`, чтобы освободить объект подключения.

```
static void Main(string[] args)
{
    ...
    // Получить объект чтения данных с помощью ExecuteReader().
    using(SqlDataReader myDataReader = myCommand.ExecuteReader())
    {
        // Организовать цикл по результатам.
        while (myDataReader.Read())
        {
            Console.WriteLine("-> Make: {0}, PetName: {1}, Color: {2}.",
                myDataReader["Make"].ToString(),
                myDataReader["PetName"].ToString(),
                myDataReader["Color"].ToString());
        }
    }
    Console.ReadLine();
}
```

В приведенном выше фрагменте кода индексатор объекта чтения данных перегружен, чтобы принимать либо объект `string` (имя столбца), либо значение `int` (порядковый номер столбца). Это позволяет прояснить логику объекта чтения (и избежать применения жестко закодированных строковых имен) с помощью следующей модификации (обратите внимание на использование свойства `FieldCount`):

```
while (myDataReader.Read())
{
    Console.WriteLine("***** Record *****");
    for (int i = 0; i < myDataReader.FieldCount; i++)
    {
        Console.WriteLine("{0} = {1} ",
            myDataReader.GetName(i),
            myDataReader.GetValue(i).ToString());
    }
    Console.WriteLine();
}
```

Если в этот момент скомпилировать и запустить проект, то будет выдан список всех автомобилей из таблицы `Inventory` базы данных `AutoLot`. В следующем выводе показано несколько первых записей из текущей базы данных `AutoLot`:

```
***** Fun with Data Readers *****
***** Info about your connection *****
Database location: (local)\SQLEXPRESS
Database name: AutoLot
Timeout: 30
Connection state: Open
***** Record *****
CarID = 83
Make = Ford
Color = Rust
PetName = Rusty
***** Record *****
CarID = 107
Make = Ford
Color = Red
PetName = Snake
```

Получение нескольких результирующих наборов с использованием объекта чтения данных

Объекты чтения данных могут получать несколько результирующих наборов с применением одного объекта команды. Например, если нужно получить все строки из таблицы Inventory, а также все строки из таблицы Customers, то можно указать сразу оба SQL-оператора, разделив их точкой с запятой:

```
string strSQL = "Select * From Inventory;Select * from Customers";
```

После получения объекта чтения данных можно просмотреть все записи результирующего набора с помощью метода `NextResult()`. Учтите, что автоматически всегда возвращается первый результирующий набор. Таким образом, если требуется просмотреть все строки каждой таблицы, нужно будет создать следующую конструкцию итерации:

```
do
{
    while (myDataReader.Read())
    {
        Console.WriteLine("***** Record *****");
        for (int i = 0; i < myDataReader.FieldCount; i++)
        {
            Console.WriteLine("{0} = {1}",
                myDataReader.GetName(i),
                myDataReader.GetValue(i).ToString());
        }
        Console.WriteLine();
    }
} while (myDataReader.NextResult());
```

К этому моменту вы должны хорошо знать функциональность, которую приносят к таблицам объекты чтения данных. Также не забывайте, что объект чтения данных может обрабатывать только SQL-операторы `Select`; его нельзя использовать для изменения существующей таблицы базы данных посредством запросов `Insert`, `Update` или `Delete`. Модификация существующей базы данных требует дальнейшего изучения объектов команд.

Исходный код. Проект `AutoLotDataReader` доступен в подкаталоге `Chapter 21`.

Создание многократно используемой библиотеки доступа к данным

Метод `ExecuteReader()` извлекает объект чтения данных, который позволяет просматривать результаты SQL-оператора `Select` с помощью потока информации, доступного только для чтения в прямом направлении. Однако если требуется выполнить операторы SQL, модифицирующие таблицу данных, то нужен вызов метода `ExecuteNonQuery()` данного объекта команды. Этот единый метод предназначен для выполнения вставок, изменений и удалений на основе формата текста команды.

На заметку! Формально понятие *не запросный* (`nonquery`) означает оператор SQL, который не возвращает результирующий набор. Таким образом, операторы `Select` являются запросам, а операторы `Insert`, `Update` и `Delete` — нет. Соответственно, метод `ExecuteNonQuery()` возвращает значение `int`, которое представляет количество строк, затронутых этими операторами, а не новое множество записей.

Далее вы научитесь модифицировать содержимое существующей базы данных с помощью только вызова метода `ExecuteNonQuery()`. Следующей целью будет построение специальной библиотеки доступа к данным, в которой инкапсулируется процесс работы с базой данных `AutoLot`. В реальной производственной среде логика ADO.NET почти всегда будет изолирована в `.dll`-сборке .NET по одной простой причине — многократное использование кода. В начальных примерах этой главы это не делалось, чтобы не отвлекаться от решаемых задач. Тем не менее, совершенно непроизводительно тратить время на разработку *той же самой* логики подключения, *той же самой* логики чтения данных и *той же самой* логики выполнения команд для каждого приложения, которому понадобится работать с базой данных `AutoLot`.

Изолирование логики доступа к данным в библиотеке кода .NET означает, что различные приложения с любым пользовательским интерфейсом (например, консольный, в стиле рабочего стола или в веб-стиле) могут ссылаться на существующую библиотеку в независимой от языка манере. Таким образом, если написать библиотеку доступа к данным на C#, то другие разработчики смогут строить пользовательские интерфейсы на любом удобном для них языке .NET.

В настоящей главе библиотека доступа к данным (`AutoLotDAL.dll`) будет содержать единственное пространство имен (`AutoLotConnectedLayer`), которое взаимодействует с базой данных `AutoLot` с применением подключенных типов ADO.NET. В следующей главе в эту же библиотеку будет добавлено новое пространство имен (`AutoLotDisconnectionLayer`), содержащее типы для взаимодействия с базой данных `AutoLot` с помощью автономного уровня. После этого библиотека будет неоднократно применяться в различных приложениях, разрабатываемых в остальных главах книги.

Начните с создания нового проекта C# Class Library (Библиотека классов C#) по имени `AutoLotDAL` (сокращение от *AutoLot Data Access Layer* (уровень доступа к данным `AutoLot`)) и переименуйте первоначальный файл кода C# в `AutoLotConnDAL.cs`. Затем переименуйте пространство имен в `AutoLotConnectedLayer` и замените имя первоначального класса вариантом `InventoryDAL`; этот класс будет определять разнообразные члены, предназначенные для взаимодействия с таблицей `Inventory` базы данных `AutoLot`. И, наконец, импортируйте следующие пространства имен .NET:

```
using System;
...
// Будет использоваться поставщик SQL Server;
// тем не менее, для обеспечения более высокой гибкости
// разрешено также применять шаблон фабрики поставщиков ADO.NET.
using System.Data;
using System.Data.SqlClient;

namespace AutoLotConnectedLayer
{
    public class InventoryDAL
    {
    }
}
```

На заметку! Вспомните из главы 13, что когда объекты используют типы, управляющие низкоуровневыми ресурсами (например, подключением к базе данных), рекомендуется реализовать интерфейс `IDisposable` и написать подходящий финализатор. В производственной среде классы, подобные `InventoryDAL`, делают то же самое, но здесь это предприниматься не будет, чтобы не отвлекаться от особенностей ADO.NET.

Добавление логики подключения

Первая задача, которую понадобится решить, связана с определением методов, позволяющих вызывающему процессу подключаться к источнику данных с помощью допустимой строки соединения и отключаться от него. Поскольку в сборке AutoLotDAL.dll будет жестко закодировано использование типов из пространства имен System.Data.SqlClient, определите закрытую переменную типа SqlConnection, которая будет создаваться во время создания объекта InventoryDAL. Кроме того, определите методы OpenConnection() и CloseConnection(), которые будут взаимодействовать с этой переменной.

```
public class InventoryDAL
{
    // Этот член будет использоваться всеми методами.
    private SqlConnection sqlCn = null;

    public void OpenConnection(string connectionString)
    {
        sqlCn = new SqlConnection();
        sqlCn.ConnectionString = connectionString;
        sqlCn.Open();
    }

    public void CloseConnection()
    {
        sqlCn.Close();
    }
}
```

Для краткости в типе InventoryDAL не будут проверяться все возможные исключения, и не будут генерироваться специальные исключения в различных ситуациях (например, когда строка соединения неверно сформирована). Однако при создании производственной библиотеки доступа к данным определенно пришлось бы воспользоваться приемами структурированной обработки исключений для учета всех аномалий во время выполнения.

Добавление логики вставки

Вставка новой записи в таблицу Inventory сводится к формированию SQL-оператора Insert (на основе введенных пользователем данных) и вызову метода ExecuteNonQuery() с использованием объекта команды. Чтобы увидеть это в действии, добавьте в класс InventoryDAL открытый метод InsertAuto(), принимающий четыре параметра, которые соответствуют четырем столбцам таблицы Inventory (CarID, Color, Make и PetName). На основании этих аргументов сформируйте строку для вставки новой записи. И, наконец, воспользуйтесь объектом SqlConnection, чтобы выполнить построенный SQL-оператор.

```
public void InsertAuto(int id, string color, string make, string petName)
{
    // Сформировать SQL-оператор.
    string sql = string.Format("Insert Into Inventory" +
        "(CarID, Make, Color, PetName) Values" +
        "('{0}', '{1}', '{2}', '{3}')" , id, make, color, petName);

    // Выполнить SQL-оператор с применением нашего подключения.
    using(SqlCommand cmd = new SqlCommand(sql, this.sqlCn))
    {
        cmd.ExecuteNonQuery();
    }
}
```

Синтаксически этот метод вполне корректен, но можно было бы предусмотреть его перегруженную версию, которая позволила бы вызывающему коду передать объект строго типизированного класса, представляющий данные для новой строки. Определите новый класс `NewCar`, который представляет новую строку в таблице `Inventory`:

```
public class NewCar
{
    public int CarID { get; set; }
    public string Color { get; set; }
    public string Make { get; set; }
    public string PetName { get; set; }
}
```

Теперь добавьте в класс `InventoryDAL` следующую версию метода `InsertAuto()`:

```
public void InsertAuto(NewCar car)
{
    // Сформировать SQL-оператор.
    string sql = string.Format("Insert Into Inventory" +
        "(CarID, Make, Color, PetName) Values" +
        "('{0}', '{1}', '{2}', '{3}')" , car.CarID, car.Make, car.Color, car.PetName);

    // Выполнить SQL-оператор с применением нашего подключения.
    using (SqlCommand cmd = new SqlCommand(sql, this.sqlCn))
    {
        cmd.ExecuteNonQuery();
    }
}
```

Определение классов, представляющих записи в реляционной базе данных — это распространенный способ создания библиотеки доступа к данным. В действительности, как будет показано в главе 23, инфраструктура ADO.NET Entity Framework автоматически генерирует строго типизированные классы, которые позволяют взаимодействовать с данными базы. Кстати, автономный уровень ADO.NET (рассматриваемый в главе 22) генерирует строго типизированные объекты `DataSet` для представления данных из заданной таблицы в реляционной базе данных.

На заметку! Как вам, возможно, известно, построение оператора SQL с применением конкатенации строк может оказаться небезопасным (вспомните атаки внедрением в SQL). Текст команды лучше формировать с помощью параметризованного запроса, который будет описан далее в главе.

Добавление логики удаления

Удаление существующей записи не сложнее вставки новой записи. В отличие от кода для метода `InsertAuto()`, здесь вы узнаете о важном контексте `try/catch`, который обрабатывает возможную ситуацию, когда предпринимается попытка удаления автомобиля, уже заказанного кем-то из таблицы `Customers`. Добавьте в класс `InventoryDAL` следующий метод:

```
public void DeleteCar(int id)
{
    // Получить идентификатор удаляемого автомобиля, затем выполнить удаление.
    string sql = string.Format("Delete from Inventory where CarID = '{0}'", id);
    using (SqlCommand cmd = new SqlCommand(sql, this.sqlCn))
    {
        try
        {
            cmd.ExecuteNonQuery();
        }
    }
}
```

```

        catch (SqlException ex)
        {
            Exception error = new Exception("Sorry! That car is on order!", ex);
            throw error;
        }
    }
}

```

Добавление логики обновления

Когда дело доходит до обновления существующей записи в таблице Inventory, то сразу же возникает очевидный вопрос: что именно можно позволить изменять вызывающему процессу: цвет автомобиля, дружественное имя, марку или все сразу? Один из способов достижения максимальной гибкости предусматривает определение метода, принимающего параметр типа string, который может содержать любой оператор SQL, но это в лучшем случае рискованно.

В идеале лучше иметь набор методов, которые позволяют вызывающему процессу обновлять запись различными способами. Однако для нашей простой библиотеки доступа к данным мы определим единственный метод, который позволяет вызывающему процессу обновить дружественное имя указанного автомобиля:

```

public void UpdateCarPetName(int id, string newPetName)
{
    // Получить идентификатор модифицируемого автомобиля и новое дружественное имя.
    string sql =
        string.Format("Update Inventory Set PetName = '{0}' Where CarID = '{1}'",
            newPetName, id);
    using (SqlCommand cmd = new SqlCommand(sql, this.sqlCn))
    {
        cmd.ExecuteNonQuery();
    }
}

```

Добавление логики выборки

Теперь необходимо добавить метод для выборки записей. Как было показано ранее в этой главе, объект чтения данных конкретного поставщика данных позволяет выбирать записи с помощью курсора, допускающего только чтение в прямом направлении. Посредством вызова метода Read() можно обработать каждую запись поочередно. Все это замечательно, но теперь необходимо разобраться, как вернуть эти записи вызывающему уровню приложения.

Один из подходов заключается в наполнении и возврате многомерного массива (или другого возвращаемого значения, такого как обобщенный объект List<NewCar>) с применением данных, полученных методом Read(). Ниже представлен этот подход получения данных из таблицы Inventory:

```

public List<NewCar> GetAllInventoryAsList()
{
    // Здесь будут храниться записи.
    List<NewCar> inv = new List<NewCar>();
    // Подготовить объект команды.
    string sql = "Select * From Inventory";
    using (SqlCommand cmd = new SqlCommand(sql, this.sqlCn))
    {
        SqlDataReader dr = cmd.ExecuteReader();
        while (dr.Read())
        {
            inv.Add(new NewCar

```

```

    {
        CarID = (int)dr["CarID"],
        Color = (string)dr["Color"],
        Make = (string)dr["Make"],
        PetName = (string)dr["PetName"]
    });
}
dr.Close();
}
return inv;
}

```

Еще один способ предусматривает возврат объекта `System.Data.DataTable`, который в действительности является частью автономного уровня ADO.NET. Полное описание автономного уровня будет приведено в следующей главе, а пока достаточно уяснить, что `DataTable` — это класс, представляющий табличный блок данных (например, сетку электронной таблицы).

Внутренне класс `DataTable` представляет данные в виде коллекции строк и столбцов. Эти коллекции можно заполнять программным образом, но в типе `DataTable` имеется метод `Load()`, который может автоматически наполнять их с помощью объекта чтения данных. Взгляните на следующие методы, которые возвращают данные из таблицы `Inventory` в виде `DataTable`:

```

public DataTable GetAllInventoryAsDataTable()
{
    // Здесь будут храниться записи.
    DataTable inv = new DataTable();
    // Подготовить объект команды.
    string sql = "Select * From Inventory";
    using(SqlCommand cmd = new SqlCommand(sql, this.sqlCn))
    {
        SqlDataReader dr = cmd.ExecuteReader();
        // Заполнить DataTable данными из объекта чтения и выполнить очистку.
        inv.Load(dr);
        dr.Close();
    }
    return inv;
}

```

Работа с параметризованными объектами команд

Пока в логике вставки, обновления и удаления для типа `InventoryDAL` использовались жестко закодированные строковые литералы для каждого SQL-запроса. Возможно, вы знаете о существовании *параметризованных запросов*, которые позволяют трактовать параметры SQL как объекты, а не простые текстовые фрагменты. Работа с SQL-запросами в более объектно-ориентированной манере помогает сократить количество опечаток (при наличии строго типизированных свойств). Вдобавок параметризованные запросы обычно выполняются значительно быстрее запросов в виде строковых литералов, поскольку они анализируются только один раз (а не каждый раз, как это происходит, когда свойству `CommandText` присваивается SQL-строка). Кроме того, параметризованные запросы защищают от атак внедрением в SQL (широко известная проблема безопасности доступа к данным).

Для поддержки параметризованных запросов объекты команд ADO.NET поддерживают коллекцию индивидуальных объектов параметров. По умолчанию эта коллекция пуста, но в нее можно занести любое количество объектов параметров, которые соответствуют *параметрам-заполнителям* в SQL-запросе. Если нужно связать параметр SQL-запроса с членом коллекции параметров некоторого объекта команды, поставьте

перед параметром SQL символ @ (по крайней мере, при работе с Microsoft SQL Server, хотя не все СУБД поддерживают эту нотацию).

Указание параметров с использованием типа DbParameter

Прежде чем приступить к построению параметризованного запроса, необходимо ознакомиться с типом DbParameter (который является базовым классом для объектов параметров поставщиков). У этого класса имеется ряд свойств, которые позволяют задать имя, размер и тип параметра, а также другие характеристики, например, направление просмотра параметра. Некоторые важные свойства типа DbParameter приведены в табл. 21.7.

Таблица 21.7. Основные члены типа DbParameter

Свойство	Описание
DbType	Возвращает или устанавливает собственный тип данных параметра, представленный как тип CLR
Direction	Возвращает или устанавливает вид параметра: только для ввода, только для вывода, для ввода и для вывода или параметр для возвращаемого значения
IsNullable	Возвращает или устанавливает признак, может ли параметр принимать значения null
ParameterName	Возвращает или устанавливает имя DbParameter
Size	Возвращает или устанавливает максимальный размер данных для параметра в байтах (полезно только для текстовых данных)
Value	Возвращает или устанавливает значение параметра

Давайте теперь посмотрим, как заполнить коллекцию объекта команды совместимыми с DbParameter объектами, переделав метод InsertAuto() так, чтобы в нем использовались объекты параметров (аналогично можно переделать и все остальные методы, однако в настоящем примере это не обязательно):

```
public void InsertAuto(int id, string color, string make, string petName)
{
    // Обратите внимание на "заполнители" в SQL-запросе.
    string sql = string.Format("Insert Into Inventory" +
        "(CarID, Make, Color, PetName) Values" +
        "(@CarID, @Make, @Color, @PetName)");

    // Эта команда будет иметь внутренние параметры.
    using(SqlCommand cmd = new SqlCommand(sql, this.sqlCn))
    {
        // Заполнить коллекцию параметров.
        SqlParameter param = new SqlParameter();
        param.ParameterName = "@CarID";
        param.Value = id;
        param.SqlDbType = SqlDbType.Int;
        cmd.Parameters.Add(param);

        param = new SqlParameter();
        param.ParameterName = "@Make";
        param.Value = make;
        param.SqlDbType = SqlDbType.Char;
        param.Size = 10;
        cmd.Parameters.Add(param);
    }
}
```

```

param = new SqlParameter();
param.ParameterName = "@Color";
param.Value = color;
param.SqlDbType = SqlDbType.Char;
param.Size = 10;
cmd.Parameters.Add(param);

param = new SqlParameter();
param.ParameterName = "@PetName";
param.Value = petName;
param.SqlDbType = SqlDbType.Char;
param.Size = 10;
cmd.Parameters.Add(param);

cmd.ExecuteNonQuery();
}
}

```

Обратите внимание, что здесь SQL-запрос также содержит четыре символа-заполнителя, перед каждым из которых находится символ @. С помощью свойства `ParameterName` в типе `SqlParameter` можно описать каждый из этих заполнителей и задать различную информацию (значение, тип данных, размер и т.д.), причем строго типизированным образом. После подготовки всех объектов параметров они добавляются в коллекцию объекта команды вызовом метода `Add()`.

На заметку! Для установки объектов параметров в приведенном примере используются различные свойства. Однако обратите внимание, что объекты параметров поддерживают несколько перегруженных конструкторов, которые позволяют устанавливать значения различных свойств (что в результате дает более компактную кодовую базу). Учтите также, что в Visual Studio имеются различные графические конструкторы, которые автоматически сгенерируют большой объем этого утомительного кода работы с параметрами (см. главы 22 и 23).

Хотя построение параметризованного запроса часто требует большего объема кода, в результате получается более удобный способ для программной настройки SQL-операторов, а также более высокая производительность. Эту технику можно применять для любых SQL-запросов, хотя параметризованные запросы наиболее удобны, если нужно запускать хранимые процедуры.

Выполнение хранимой процедуры

Вспомните, что *хранимая процедура* — это именованный блок SQL-кода, хранимый в базе данных. Хранимые процедуры можно создавать для возврата набора строк или скалярных типов данных, а также выполнения других нужных действий (например, вставки, обновления или удаления); они могут принимать любое количество необязательных параметров. В конечном итоге получается единица работы, которая ведет себя подобно типичной функции, но только находится в хранилище данных, а не в двоичном бизнес-объекте. На этом этапе в базе данных `AutoLot` определена одна хранимая процедура по имени `GetPetName`, имеющая следующий формат:

```

GetPetName
@carID int,
@petName char(10) output
AS
SELECT @petName = PetName from Inventory where CarID = @carID

```

Теперь рассмотрим следующий финальный метод типа `InventoryDAL`, который вызывает эту хранимую процедуру:

```

public string LookUpPetName(int carID)
{
    string carPetName = string.Empty;
    // Установить имя хранимой процедуры.
    using (SqlCommand cmd = new SqlCommand("GetPetName", this.sqlCn))
    {
        cmd.CommandType = CommandType.StoredProcedure;
        // Входной параметр.
        SqlParameter param = new SqlParameter();
        param.ParameterName = "@carID";
        param.SqlDbType = SqlDbType.Int;
        param.Value = carID;
        // По умолчанию параметры считаются входными (Input), но все же для ясности:
        param.Direction = ParameterDirection.Input;
        cmd.Parameters.Add(param);
        // Выходной параметр.
        param = new SqlParameter();
        param.ParameterName = "@petName";
        param.SqlDbType = SqlDbType.Char;
        param.Size = 10;
        param.Direction = ParameterDirection.Output;
        cmd.Parameters.Add(param);
        // Выполнить хранимую процедуру.
        cmd.ExecuteNonQuery();
        // Возвратить выходной параметр.
        carPetName = (string)cmd.Parameters["@petName"].Value;
    }
    return carPetName;
}

```

Относительно вызова хранимых процедур существует один важный аспект: имейте в виду, что объект команды может представлять оператор SQL (по умолчанию) или имя хранимой процедуры. Если необходимо сообщить объекту команды, что он должен вызывать хранимую процедуру, то нужно передать имя этой процедуры (через аргумент конструктора или с помощью свойства `CommandText`) и установить в свойстве `CommandType` значение `CommandType.StoredProcedure` (иначе возникнет исключение времени выполнения, т.к. по умолчанию объект команды ожидает оператор SQL):

```

SqlCommand cmd = new SqlCommand("GetPetName", this.sqlCn);
cmd.CommandType = CommandType.StoredProcedure;

```

Далее обратите внимание, что свойство `Direction` объекта параметра позволяет указать направление действия каждого параметра, передаваемого хранимой процедуре (например, входной параметр, выходной параметр, входной/выходной параметр или возвращаемое значение). Как и ранее, все объекты параметров добавляются в коллекцию параметров для объекта команды:

```

// Входной параметр.
SqlParameter param = new SqlParameter();
param.ParameterName = "@carID";
param.SqlDbType = SqlDbType.Int;
param.Value = carID;
param.Direction = ParameterDirection.Input;
cmd.Parameters.Add(param);

```

После завершения работы хранимой процедуры с помощью вызова `ExecuteNonQuery()` значение выходного параметра можно получить за счет просмотра коллекции параметров для объекта команды и соответствующего приведения типа:

```

// Возвратить выходной параметр.
carPetName = (string)cmd.Parameters["@petName"].Value;

```

К этому моменту первый вариант библиотеки доступа к данным AutoLotDAL.dll готов. Эту сборку можно использовать при построении произвольных интерфейсных приложений (консольного, настольного с графическим пользовательским интерфейсом или веб-приложения на основе HTML) для отображения и редактирования данных. Создание графических пользовательских интерфейсов пока еще не рассматривалось, поэтому протестируем полученную библиотеку доступа к данным с помощью нового консольного приложения.

Исходный код. Проект AutoLotDAL доступен в подкаталоге Chapter 21.

Создание приложения с консольным пользовательским интерфейсом

Создайте новое консольное приложение по имени AutoLotCUIClient. После создания нового проекта не забудьте добавить в него ссылку на сборку AutoLotDAL.dll, а также на System.Configuration.dll. Затем добавьте в файл кода C# следующие операторы using:

```
using AutoLotConnectedLayer;
using System.Configuration;
using System.Data;
```

Теперь вставьте в проект новый файл App.config, содержащий элемент <connectionString>, который нужен для подключения к вашему экземпляру базы данных AutoLot, например:

```
<configuration>
  <connectionStrings>
    <add name="AutoLotSqlProvider" connectionString =
      "Data Source=(local)\SQLEXPRESS;
      Integrated Security=SSPI;Initial Catalog=AutoLot"/>
  </connectionStrings>
</configuration>
```

Реализация метода Main()

Метод Main() отвечает за выяснение у пользователя необходимого действия и выполнение соответствующего запроса с применением оператора switch.

Эта программа позволяет пользователю вводить следующие команды:

- I — вставка новой записи в таблицу Inventory;
- U — обновление существующей записи в таблице Inventory;
- D — удаление существующей записи в таблице Inventory;
- L — отображение имеющихся в наличии автомобилей с помощью объекта чтения данных;
- S — вывод списка возможных команд;
- P — вывод дружественного имени автомобиля по его идентификатору;
- Q — завершение работы программы.

Каждая из этих команд поддерживается специальным статическим методом внутри класса Program. Ниже приведена полная реализация метода Main(). Обратите внимание, что все методы, вызываемые в цикле do/while (за исключением ShowInstructions()), принимают в качестве единственного параметра объект InventoryDAL.


```

static void Main(string[] args)
{
    Console.WriteLine("***** The AutoLot Console UI *****\n");

    // Получить строку соединения из App.config.
    string cnStr =
        ConfigurationManager.ConnectionStrings["AutoLotSqlProvider"].ConnectionString;
    bool userDone = false;
    string userCommand = "";

    // Создать объект InventoryDAL.
    InventoryDAL invDAL = new InventoryDAL();
    invDAL.OpenConnection(cnStr);

    // Продолжать запрашивать у пользователя ввод вплоть до получения команды Q.
    try
    {
        ShowInstructions();
        do
        {
            Console.Write("\nPlease enter your command: ");
            userCommand = Console.ReadLine();
            Console.WriteLine();
            switch (userCommand.ToUpper())
            {
                case "I":
                    InsertNewCar(invDAL);
                    break;
                case "U":
                    UpdateCarPetName(invDAL);
                    break;
                case "D":
                    DeleteCar(invDAL);
                    break;
                case "L":
                    ListInventory(invDAL);
                    break;
                case "S":
                    ShowInstructions();
                    break;
                case "P":
                    LookUpPetName(invDAL);
                    break;
                case "Q":
                    userDone = true;
                    break;
                default:
                    Console.WriteLine("Bad data! Try again");
                    break;
            }
        } while (!userDone);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    finally
    {
        invDAL.CloseConnection();
    }
}

```

Реализация метода ShowInstructions()

Метод ShowInstructions() делает то, что и можно было ожидать — выводит инструкции:

```
private static void ShowInstructions()
{
    Console.WriteLine("I: Inserts a new car."); // Вставить новый автомобиль.
    Console.WriteLine("U: Updates an existing car."); // Обновить существующий автомобиль.
    Console.WriteLine("D: Deletes an existing car."); // Удалить существующий автомобиль.
    Console.WriteLine("L: Lists current inventory."); // Вывести текущие запасы.
    Console.WriteLine("S: Shows these instructions."); // Вывести эти инструкции.
    Console.WriteLine("P: Looks up pet name."); // Найти дружественное имя автомобиля.
    Console.WriteLine("Q: Quits program."); // Завершить программу.
}
```

Реализация метода ListInventory()

Метод ListInventory() можно реализовать двумя способами, в зависимости от того, как была создана библиотека доступа к данным. Вспомните, что метод GetAllInventoryAsDataTable() класса InventoryDAL возвращает объект DataTable. Этот подход можно реализовать так:

```
private static void ListInventory(InventoryDAL invDAL)
{
    // Получить список автомобилей на складе.
    DataTable dt = invDAL.GetAllInventoryAsDataTable();
    // Передать DataTable вспомогательной функции для отображения.
    DisplayTable(dt);
}
```

Вспомогательный метод DisplayTable() отображает данные таблицы, используя свойства Rows и Columns входного объекта DataTable (более подробно объект DataTable рассматривается в следующей главе, поэтому пока не обращайте внимания на детали):

```
private static void DisplayTable(DataTable dt)
{
    // Вывести имена столбцов.
    for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
    {
        Console.Write(dt.Columns[curCol].ColumnName + "\t");
    }
    Console.WriteLine("\n-----");
    // Вывести содержимое DataTable.
    for (int curRow = 0; curRow < dt.Rows.Count; curRow++)
    {
        for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
        {
            Console.Write(dt.Rows[curRow][curCol].ToString() + "\t");
        }
        Console.WriteLine();
    }
}
```

Если же вы предпочитаете пользоваться методом GetAllInventoryAsList() класса InventoryDAL, то можно реализовать метод ListInventoryViaList() следующим образом:

```
private static void ListInventoryViaList(InventoryDAL invDAL)
{
    // Получить список автомобилей на складе.
    List<NewCar> record = invDAL.GetAllInventoryAsList();
}
```

```
foreach (NewCar c in record)
{
    Console.WriteLine("CarID: {0}, Make: {1}, Color: {2}, PetName: {3}",
        c.CarID, c.Make, c.Color, c.PetName);
}
}
```

Реализация метода DeleteCar ()

Удаление существующего автомобиля сводится к запросу у пользователя идентификатора этого автомобиля и его передаче методу DeleteCar() типа InventoryDAL, как показано ниже:

```
private static void DeleteCar(InventoryDAL invDAL)
{
    // Получить идентификатор удаляемого автомобиля.
    Console.Write("Enter ID of Car to delete: ");
    int id = int.Parse(Console.ReadLine());

    // На случай нарушения ссылочной целостности.
    try
    {
        invDAL.DeleteCar(id);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}
```

Реализация метода InsertNewCar ()

Для вставки новой записи в таблицу Inventory необходимо запросить у пользователя информацию о новом автомобиле (с помощью вызовов Console.ReadLine()) и передать эти данные в метод InsertAuto() класса InventoryDAL:

```
private static void InsertNewCar(InventoryDAL invDAL)
{
    // Сначала получить пользовательские данные.
    int newCarID;
    string newCarColor, newCarMake, newCarPetName;
    Console.Write("Enter Car ID: "); // Запрос идентификатора автомобиля.
    newCarID = int.Parse(Console.ReadLine());

    Console.Write("Enter Car Color: "); // Запрос цвета автомобиля.
    newCarColor = Console.ReadLine();

    Console.Write("Enter Car Make: "); // Запрос марки автомобиля.
    newCarMake = Console.ReadLine();

    Console.Write("Enter Pet Name: "); // Запрос дружественного имени автомобиля.
    newCarPetName = Console.ReadLine();

    // Теперь передать информацию библиотеке доступа к данным.
    invDAL.InsertAuto(newCarID, newCarColor, newCarMake, newCarPetName);
}
```

Вспомните, что перегруженная версия метода InsertAuto() принимает объект NewCar, а не набор независимых аргументов. Таким образом, метод InsertNewCar() можно реализовать примерно так:

```
private static void InsertNewCar(InventoryDAL invDAL)
{
    // Сначала получить пользовательские данные.
    ...
    // Теперь передать информацию библиотеке доступа к данным.
    NewCar c = new NewCar { CarID = newCarID, Color = newCarColor,
                           Make = newCarMake, PetName = newCarPetName };
    invDAL.InsertAuto(c);
}
```

Реализация метода UpdateCarPetName ()

Реализация метода UpdateCarPetName () выглядит аналогично:

```
private static void UpdateCarPetName(InventoryDAL invDAL)
{
    // Сначала получить пользовательские данные.
    int carID;
    string newCarPetName;

    Console.Write("Enter Car ID: ");
    carID = int.Parse(Console.ReadLine());
    Console.Write("Enter New Pet Name: ");
    newCarPetName = Console.ReadLine();

    // Теперь передать информацию библиотеке доступа к данным.
    invDAL.UpdateCarPetName(carID, newCarPetName);
}
```

Реализация метода LookUpPetName ()

Получение дружественного имени указанного автомобиля реализуется аналогично предыдущим методам; это объясняется тем, что все низкоуровневые вызовы ADO.NET инкапсулированы в библиотеке доступа к данным:

```
private static void LookUpPetName(InventoryDAL invDAL)
{
    // Получить идентификатор автомобиля для поиска дружественного имени.
    Console.Write("Enter ID of Car to look up: ");
    int id = int.Parse(Console.ReadLine());
    Console.WriteLine("Petname of {0} is {1}.",
        id, invDAL.LookUpPetName(id).TrimEnd());
}
```

На этом консольное интерфейсное приложение завершено. Самое время запустить полученную программу и проверить работу каждого метода. Ниже приведен частичный вывод с тестированием команд L, P и Q:

***** The AutoLot Console UI *****

I: Inserts a new car.
 U: Updates an existing car.
 D: Deletes an existing car.
 L: Lists current inventory.
 S: Shows these instructions.
 P: Looks up pet name.
 Q: Quits program.

Please enter your command: L

CarID	Make	Color	PetName
-------	------	-------	---------

83	Ford	Rust	Rusty
107	Ford	Red	Snake

```

678      Yugo      Green   Clunker
904      VW        Black   Hank
1000     BMW       Black   Bimmer
1001     BMW       Tan      Daisy
1992     Saab      Pink     Pinkey

```

Please enter your command: P

Enter ID of Car to look up: 904

Petname of 904 is Hank.

Please enter your command: Q

Press any key to continue . . .

Исходный код. Проект AutoLotCUIClient доступен в подкаталоге Chapter 21.

Понятие транзакций базы данных

В завершение исследований подключенного уровня ADO.NET рассмотрим концепцию транзакций базы данных. Выражаясь просто, *транзакция* — это набор операций в базе данных, которые должны быть либо *все* выполнены, либо *все* не выполнены. Транзакции применяются для обеспечения безопасности, достоверности и согласованности данных в таблице.

Транзакции важны в ситуациях, когда операция базы данных предусматривает взаимодействие с несколькими таблицами или хранимыми процедурами (либо с комбинацией атомарных объектов базы данных). Классическим примером транзакции может служить процесс перевода денежных средств с одного банковского счета на другой. Например, если вам понадобилось перевести \$500 с депозитного счета на текущий счет, то нужно выполнить в режиме транзакции следующие шаги:

- банк должен снять \$500 с вашего депозитного счета;
- банк должен добавить \$500 на ваш текущий счет.

Вряд ли вам бы понравилось, если бы деньги были сняты с депозитного счета, но не переведены (из-за какой-то ошибки со стороны банка) на текущий счет. Однако если эти шаги упаковать в транзакцию базы данных, то СУБД гарантирует, что все взаимосвязанные шаги будут выполнены как единое целое. Если любая часть транзакции даст сбой, будет произведен *откат* всей транзакции в исходное состояние. А если все шаги будут выполнены успешно, то транзакция будет *зафиксирована*.

На заметку! В литературе, посвященной транзакциям, вам могло встречаться сокращение ACID. С его помощью обозначаются четыре ключевых свойства классической транзакции: *атомарность* (Atomic; все или ничего), *согласованность* (Consistent; на протяжении транзакции данные остаются в согласованном состоянии), *изоляция* (Isolated; транзакции не мешают друг другу) и *устойчивость* (Durable; транзакции сохраняются и протоколируются).

Оказывается, в платформе .NET есть несколько способов поддержки транзакций. В этой главе мы рассмотрим объект транзакции для поставщика данных ADO.NET (SqlTransaction в случае System.Data.SqlClient). Библиотеки базовых классов ADO.NET также обеспечивают поддержку транзакций в многочисленных API-интерфейсах, включая указанные ниже.

- System.EnterpriseServices. Это пространство имен (находящееся в сборке System.EnterpriseServices.dll) содержит типы, которые позволяют интеграцию с уровнем исполняющей среды COM+, в том числе поддержку распределенных транзакций.

- **System.Transactions.** Это пространство имен (находящееся в сборке `System.Transactions.dll`) содержит классы, позволяющие писать собственные транзакционные приложения и диспетчеры ресурсов для различных служб (MSMQ, ADO.NET, COM+ и т.д.).
- **Windows Communication Foundation.** API-интерфейс WCF предоставляет службы для упрощения организации транзакций с различными классами распределенной привязки.
- **Windows Workflow Foundations.** API-интерфейс WF предоставляет транзакционную поддержку для действий рабочих потоков.

В дополнение к встроенной поддержке транзакций внутри библиотек базовых классов .NET, также можно пользоваться языком SQL используемой СУБД. Например, можно было бы написать хранимую процедуру, в которой задействованы операторы `BEGIN TRANSACTION`, `ROLLBACK` и `COMMIT`.

Основные члены объекта транзакции ADO.NET

Типы для работы с транзакциями существуют во всех библиотеках базовых классов, но мы будем рассматривать объекты транзакции, которые имеются в поставщиках данных ADO.NET; все они являются производными от `DBTransaction` и реализуют интерфейс `IDbTransaction`. Как было указано в начале этой главы, интерфейс `IDbTransaction` определяет несколько следующих членов:

```
public interface IDbTransaction : IDisposable
{
    IDbConnection Connection { get; }
    IsolationLevel IsolationLevel { get; }
    void Commit();
    void Rollback();
}
```

Обратите внимание на свойство `Connection`, которое возвращает ссылку на объект подключения, инициировавший текущую транзакцию (как будет показано, объект транзакции можно получить из заданного объекта подключения). Метод `Commit()` вызывается, если все операции в базе данных завершились успешно. При этом все ожидающие изменения фиксируются в хранилище данных. В противоположность этому, метод `Rollback()` можно вызвать при возникновении исключения времени выполнения, тем самым сообщив СУБД, что все ожидающие изменения следует отменить и оставить первоначальные данные без изменений.

На заметку! Свойство `IsolationLevel` объекта транзакции позволяет указать степень защиты текущей транзакции от действий со стороны параллельных транзакций. По умолчанию транзакции полностью изолируются вплоть до их фиксации. Полную информацию о значениях перечисления `IsolationLevel` можно найти в документации .NET Framework 4.5 SDK.

Кроме членов, определенных интерфейсом `IDbTransaction`, в типе `SqlTransaction` определен дополнительный член по имени `Save()`, который предназначен для определения *точек сохранения*. Эта концепция позволяет откатить неудачную транзакцию до указанной точки, не выполняя откат всей транзакции. При вызове метода `Save()` с использованием объекта `SqlTransaction` можно задать удобный строковый псевдоним. А при вызове `Rollback()` этот псевдоним можно указать в качестве аргумента, чтобы выполнить *частичный откат*. Вызов `Rollback()` без аргументов приводит к отмене всех ожидающих изменений.

Добавление таблицы CreditRisks в базу данных AutoLot

CreditRisks	
	CustID
	FirstName
	LastName

Рис. 21.15. Взаимосвязанные таблицы Orders, Inventory и Customers

Давайте теперь посмотрим, как применять транзакции в ADO.NET. Начните с использования окна Server Explorer в Visual Studio для добавления в базу данных AutoLot новой таблицы по имени CreditRisks, которая содержит точно такие же столбцы, как и таблица Customers, созданная ранее в этой главе: CustID (первичный ключ), FirstName и LastName. Таблица CreditRisks предназначена для отсеивания нежелательных клиентов с плохой кредитной историей (рис. 21.15).

Как и предыдущий пример с переводом денег с депозита на текущий счет, этот пример, в котором подозрительный клиент перемещается из таблицы Customers в таблицу CreditRisks, должен работать под недремлющим оком транзакционного контекста (в конце концов, вы хотите запомнить идентификаторы и имена некредитоспособных клиентов). В частности, необходимо гарантировать, что либо текущие кредитные риски будут успешно удалены из таблицы Customers и добавлены в таблицу CreditRisks, либо ни одна из этих операций базы данных не произойдет.

На заметку! В производственной среде вам не понадобится строить совершенно новую таблицу базы данных для подозрительных клиентов. Вместо этого в существующую таблицу Customers можно было бы добавить булевский столбец IsCreditRisk. Тем не менее, эта новая таблица позволяет поработать с простой транзакцией.

Добавление метода транзакции в InventoryDAL

А теперь посмотрим, как работать с транзакциями ADO.NET программным образом. Откройте созданный ранее проект библиотеки кода AutoLotDAL и добавьте в класс InventoryDAL новый открытый метод по имени processCreditRisk(), предназначенный для работы с кредитными рисками (обратите внимание, что в этом примере для простоты не используется параметризованный запрос, однако в методе производственного уровня он должен быть задействован).

```
// Новый член класса InventoryDAL.
public void ProcessCreditRisk(bool throwEx, int custID)
{
    // Первым делом, найти текущее имя по идентификатору клиента.
    string fName = string.Empty;
    string lName = string.Empty;
    SqlCommand cmdSelect = new SqlCommand(
        string.Format("Select * from Customers where CustID = {0}", custID), sqlCn);
    using (SqlDataReader dr = cmdSelect.ExecuteReader())
    {
        if(dr.HasRows)
        {
            dr.Read();
            fName = (string)dr["FirstName"];
            lName = (string)dr["LastName"];
        }
        else
            return;
    }
    // Создать объекты команд, которые представляют каждый шаг операции.
    SqlCommand cmdRemove = new SqlCommand(
```

```

    string.Format("Delete from Customers where CustID = {0}", custID), sqlCn);
SqlCommand cmdInsert = new SqlCommand(string.Format("Insert Into CreditRisks" +
    "(CustID, FirstName, LastName) Values" +
    "({0}, '{1}', '{2}')" , custID, fName, lName), sqlCn);
// Это будет получено из объекта подключения.
SqlTransaction tx = null;
try
{
    tx = sqlCn.BeginTransaction();
    // Включить команды в транзакцию.
    cmdInsert.Transaction = tx;
    cmdRemove.Transaction = tx;

    // Выполнить команды.
    cmdInsert.ExecuteNonQuery();
    cmdRemove.ExecuteNonQuery();

    // Имитировать ошибку.
    if (throwEx)
    {
        throw new Exception("Sorry! Database error! Tx failed...");
    }
    // Зафиксировать транзакцию.
    tx.Commit();
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
    // Любая ошибка приведет к откату транзакции.
    tx.Rollback();
}
}

```

Здесь используется входной параметр типа `bool`, который указывает, нужно ли генерировать произвольное исключение при попытке обработки нежелательного клиента. Это позволит имитировать непредвиденные ситуации, которые могут привести к неудачному завершению транзакции. Понятно, что здесь это сделано лишь в демонстрационных целях; в реальности метод транзакции не должен позволять вызывающему процессу нарушать работу логики по своему усмотрению!

Обратите внимание на использование двух объектов `SqlCommand` для представления каждого шага предстоящей транзакции. После того, как на основе входного параметра `custID` получены имя и фамилия клиента, с помощью метода `BeginTransaction()` объекта подключения получается допустимый объект `SqlTransaction`. Затем (и это очень важно) потребуется *включить каждый объект команды*, присвоив его свойству `Transaction` только что полученного объекта транзакции. Если этого не сделать, логика вставки и удаления не будет выполняться в транзакционном контексте.

После вызова `ExecuteNonQuery()` на каждой команде генерируется исключение, если (и только если) значение параметра `bool` равно `true`. В этом случае производится откат всех ожидающих операций базы данных. Если исключения не произошло, оба шага будут зафиксированы в таблицах базы данных после вызова `Commit()`. Скомпилируйте измененный проект `AutoLotDAL` и удостоверьтесь в отсутствии ошибок.

Тестирование транзакции базы данных

Можно было модифицировать существующее приложение `AutoLotCUIClient`, добавив в него вызов метода `ProcessCreditRisk()`, однако вместо этого будет создано новое консольное приложение по имени `AdoNetTransaction`. Добавьте ссылку на сборку `AutoLotDAL.dll` и импортируйте пространство имен `AutoLotConnectedLayer`.

После этого откройте таблицу Customers, щелкнув правой кнопкой мыши на значке таблицы в окне Server Explorer и выбрав в контекстном меню пункт Show Table Data (Показать данные таблицы). Создайте нового клиента с плохой кредитной историей, например:

- CustID: 333
- FirstName: Homer
- LastName: Simpson

И, наконец, измените метод Main() следующим образом:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Simple Transaction Example *****\n");
    // Простой способ разрешить или запретить успешное выполнение транзакции.
    bool throwEx = true;
    string userAnswer = string.Empty;
    Console.Write("Do you want to throw an exception (Y or N): ");
    userAnswer = Console.ReadLine();
    if (userAnswer.ToLower() == "n")
    {
        throwEx = false;
    }
    InventoryDAL dal = new InventoryDAL();
    dal.OpenConnection(@"Data Source=(local)\SQLEXPRESS;Integrated Security=SSPI;" +
        "Initial Catalog=AutoLot");
    // Обработать клиента с идентификатором 333.
    dal.ProcessCreditRisk(throwEx, 333);
    Console.WriteLine("Check CreditRisk table for results");
    Console.ReadLine();
}
```

Если запустить программу и выбрать генерацию исключения, то клиент с именем Homer не будет удален из таблицы Customers, т.к. выполнится откат всей транзакции. Но при отсутствии исключения окажется, что клиент с идентификатором 333 находится уже не в таблице Customers, а в таблице CreditRisks.

Исходный код. Проект AdoNetTransaction доступен в подкаталоге Chapter 21.

Резюме

ADO.NET — это собственная технология доступа к данным платформы .NET, которую можно использовать тремя различными способами: подключенным, автономным и с помощью Entity Framework. В настоящей главе был рассмотрен подключенный уровень, а также продемонстрирована роль поставщиков данных, которые представляют собой конкретные реализации нескольких абстрактных базовых классов (в пространстве имен System.Data.Common) и типов интерфейсов (из пространства имен System.Data). Вы увидели, как создавать кодовую базу, не зависящую от поставщика, с помощью модели фабрик поставщиков данных ADO.NET.

Вы также узнали, как с применением объектов подключений, объектов транзакций, объектов команд и объектов чтения данных из подключенного уровня выбирать, обновлять, вставлять и удалять записи. Кроме того, как было показано в главе, объекты команд поддерживают внутреннюю коллекцию параметров, которые можно использовать для обеспечения безопасности к типам в SQL-запросах; она также весьма полезна при запуске хранимых процедур.

ADO.NET, часть II: автономный уровень

В предыдущей главе рассматривался *подключенный уровень* ADO.NET, который позволяет отправлять в базу данных SQL-операторы с использованием объектов подключения, объектов команд и объектов чтения данных соответствующего поставщика данных. В этой главе вы ознакомитесь с *автономным уровнем* ADO.NET. Этот аспект ADO.NET позволяет смоделировать в памяти данные из базы, внутри вызывающего уровня, за счет применения многочисленных членов из пространства имен System.Data (в особенности DataSet, DataTable, DataRow, DataColumn, DataView и DataRelation). При этом возникает иллюзия, что вызывающий уровень постоянно подключен к внешнему источнику данных, хотя на самом деле все операции выполняются с локальной копией реляционных данных.

Хотя этот автономный аспект ADO.NET можно использовать даже без подключения к реляционной базе данных, чаще всего вы будете получать заполненные объекты DataSet с помощью объекта адаптера данных конкретного поставщика данных. Как будет показано, объекты адаптеров данных выполняют связующую роль между клиентским уровнем и реляционной базой данных. С их помощью можно получить объекты DataSet, поработать с их содержимым и отправить измененные строки обратно для дальнейшей обработки. В результате получается высоко масштабируемое .NET-приложение обработки данных.

В настоящей главе также будут продемонстрированы некоторые приемы привязки данных с применением контекста приложений с графическим пользовательским интерфейсом Windows Forms и рассмотрена роль *строго типизированного* объекта DataSet. Мы добавим в библиотеку доступа к данным AutoLotDAL.dll, созданную в главе 21, новое пространство имен, в котором используется автономный уровень ADO.NET. В завершение главы мы рассмотрим технологию LINQ to DataSet, которая позволяет применять запросы LINQ к находящемуся в памяти кешу данных.

На заметку! Разнообразные технологии привязки данных для приложений Windows Presentation Foundation и ASP.NET будут описаны позже в этой книге.

Понятие автономного уровня ADO.NET

Как было показано в предыдущей главе, работа с подключенным уровнем позволяет взаимодействовать с базой данных с помощью первичных объектов подключения, команд и чтения данных. Этот небольшой набор типов позволяет выбирать, вставлять, обновлять и удалять записи (а также вызывать хранимые процедуры или выполнять

другие операции над данными — например, операторы DDL для создания таблицы и DCL для назначения полномочий). Тем не менее, вы увидели лишь половину ADO.NET, поскольку объектную модель ADO.NET можно применять и в автономной манере.

За счет использования автономных типы становится возможным моделирование реляционных данных с помощью модели объектов, находящихся в памяти. Кроме простого моделирования табличных данных, состоящих из строк и столбцов, типы в пространстве имен `System.Data` позволяют представлять отношения между таблицами, ограничения столбцов, первичные ключи, представления и другие примитивы баз данных. К смоделированным данным можно применять фильтры, отправлять запросы в памяти и сохранять (или загружать) данные в формате XML и в двоичном формате. И все это можно делать, даже не подключаясь к СУБД (откуда и термин *автономный уровень*) — достаточно загрузить данные из локального XML-файла или программным образом построить объект `DataSet`.

На заметку! В главе 23 будет рассмотрена инфраструктура ADO.NET Entity Framework, построенная на описанных здесь концепциях автономного уровня.

Автономные типы действительно можно было бы использовать без подключения к базе данных, но обычно все-таки применяются подключения и объекты команд. Кроме того, применяется и специфичный объект — *адаптер данных* (расширяющий абстрактный тип `DbDataAdapter`), который как раз выбирает и обновляет данные. В отличие от подключенного уровня, данные, полученные через адаптер данных, не обрабатываются с помощью объектов чтения данных. Вместо этого объекты адаптеров пересылают данные между вызывающим процессом и источником данных с помощью объектов `DataSet` (точнее, объектов `DataTable` в `DataSet`). Тип `DataSet` представляет собой контейнер для любого количества объектов `DataTable`, каждый из которых содержит коллекцию объектов `DataRow` и `DataColumn`.

Объект адаптера данных конкретного поставщика данных автоматически обслуживает подключение к базе данных. Для повышения масштабируемости адаптеры данных удерживают подключение открытым в течение минимально возможного времени. Как только вызывающий процесс получит объект `DataSet`, вызывающий уровень полностью отключается от базы данных и остается с локальной копией удаленных данных. Теперь в нем можно вставлять, удалять или обновлять строки в указанном объекте `DataTable`, но физическая база данных не будет обновлена до тех пор, пока вызывающий процесс явно не передаст `DataTable` из `DataSet` адаптеру данных для обновления. По сути, объекты `DataSet` имитируют постоянное подключение клиентов, хотя на самом деле они работают с находящейся в памяти базой данных (рис. 22.1).

Поскольку основой автономного уровня является класс `DataSet`, то первоочередная задача этой главы — научиться манипулировать им вручную. После этого у вас не будет проблем с обработкой содержимого `DataSet`, извлеченного из объекта адаптера данных.



Рис. 22.1. Объекты адаптеров данных пересылают информацию на клиентский уровень и обратно

Роль объектов DataSet

Как отмечалось ранее, объект DataSet является представлением реляционных данных в памяти. Более конкретно, DataSet — это класс, который внутренне поддерживает три строго типизированных коллекции (рис. 22.2).

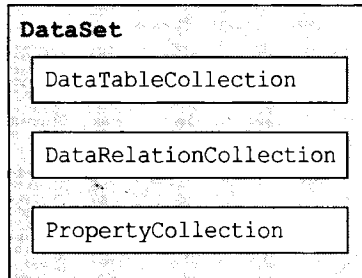


Рис. 22.2. Внутреннее устройство класса DataSet

Свойство Tables класса DataSet предоставляет доступ к коллекции DataTable Collection, которая содержит отдельные объекты DataTable. В DataSet используется еще одна важная коллекция — DataRelationCollection. Учитывая, что DataSet является автономной версией схемы базы данных, его можно использовать для программного представления отношений “родительский–дочерний” между таблицами. Например, с применением типа DataRelation можно создать отношение между двумя таблицами для моделирования ограничения внешнего ключа. Затем с помощью свойства Relations этот объект можно добавить в коллекцию DataRelationCollection. С этого момента при поиске данных можно перемещаться по связанным таблицам. Далее в главе будет показано, как это сделать.

Свойство ExtendedProperties предоставляет доступ к объекту PropertyCollection, который позволяет связать с DataSet любую дополнительную информацию в виде пар “имя/значение”. Эта информация может быть совершенно произвольной, даже не имеющей отношения к самим данным. К примеру, с объектом DataSet можно связать название компании, которое затем будет играть роль метаданных, находящихся в памяти. Другими примерами расширенных свойств могут служить временные метки, зашифрованный пароль, который необходим для доступа к содержимому DataSet, число, представляющее частоту обновления данных, и многое другое.

На заметку! Классы DataTable и DataColumn также поддерживают свойство ExtendedProperties.

Основные свойства класса DataSet

Прежде чем погрузиться в разнообразные мелочи программирования, рассмотрим некоторые основные члены класса DataSet. Кроме свойств Tables, Relations и ExtendedProperties, в табл. 22.1 кратко описано несколько дополнительных полезных свойств.

Основные методы класса DataSet

Методы класса DataSet работают в сочетании с некоторой функциональностью, предоставляемой описанными выше свойствами. В дополнение к взаимодействию с потоками XML класс DataSet предлагает методы, позволяющие копировать содержимое DataSet, перемещаться между внутренними таблицами и устанавливать начальные и конечные точки пакетных обновлений. Некоторые основные методы описаны в табл. 22.2.

Таблица 22.1. Свойства класса DataSet

Свойство	Описание
CaseSensitive	Указывает, чувствительно ли к регистру сравнение строк в объектах DataTable. По умолчанию равно false (сравнение строк выполняется без учета регистра)
DataSetName	Представляет понятное имя для объекта DataSet. Обычно это значение передается через параметр конструктора
EnforceConstraints	Возвращает или получает значение, определяющее, применяются ли правила ограничений при выполнении любых обновлений (по умолчанию равно true)
HasErrors	Получает значение, определяющее, имеются ли ошибки в любой строке любого из объектов DataTable в DataSet
RemotingFormat	Позволяет определить, как объект DataSet должен сериализовать свое содержимое (в двоичном виде или стандартно в XML)

Таблица 22.2. Методы класса DataSet

Метод	Описание
AcceptChanges ()	Отправляет все изменения, сделанные в этом объекте DataSet после его загрузки или последнего вызова AcceptChanges ()
Clear ()	Полностью очищает DataSet, удаляя все строки в каждом объекте DataTable
Clone ()	Клонирует структуру DataSet, в том числе и всех объектов DataTable, а также все отношения и ограничения
Copy ()	Копирует структуру и данные текущего объекта DataSet
GetChanges ()	Возвращает копию объекта DataSet, содержащую все изменения, которые были сделаны в этом DataSet после его загрузки или последнего вызова AcceptChanges (). У этого метода есть перегруженные варианты, которые позволяют получить только новые строки, только измененные строки или только удаленные строки
HasChanges ()	Возвращает признак, содержит ли DataSet изменения, т.е. новые, удаленные или измененные строки
Merge ()	Объединяет текущий DataSet с указанным DataSet
ReadXml ()	Позволяет определить структуру объекта DataSet и заполнить его данными на основе XML-схемы и данных из потока
RejectChanges ()	Отменяет все изменения, которые были сделаны в этом DataSet после его загрузки или последнего вызова AcceptChanges ()
WriteXml ()	Позволяет записать содержимое DataSet в поток

Создание DataSet

Теперь, когда вы лучше понимаете роль DataSet (и имеете некоторое представление о его возможностях), создайте новое консольное приложение по имени SimpleDataSet и импортируйте пространство имен System.Data. В методе Main() определите новый объект DataSet с тремя расширенными свойствами, представляющими временную метку, уникальный идентификатор (типа System.Guid) и название компании:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with DataSets *****\n");
    // Создать объект DataSet и добавить несколько свойств.
    DataSet carsInventoryDS = new DataSet("Car Inventory");
    carsInventoryDS.ExtendedProperties["TimeStamp"] = DateTime.Now;
    carsInventoryDS.ExtendedProperties["DataSetID"] = Guid.NewGuid();
    carsInventoryDS.ExtendedProperties["Company"] =
        "Mikko's Hot Tub Super Store";
    Console.ReadLine();
}
```

На заметку! Глобально уникальный идентификатор (globally unique identifier — GUID) представляет собой статически уникальное 128-битовое число.

Объект `DataSet` не особенно интересен, если он не содержит хоть немного объектов `DataTable`. Значит, теперь нужно изучить внутреннее устройство `DataTable`, начиная с типа `DataColumn`.

Работа с объектами `DataColumn`

Тип `DataColumn` представляет один столбец внутри объекта `DataTable`. Вообще говоря, набор всех объектов `DataColumn`, содержащихся в заданном объекте `DataTable`, представляет информацию схемы таблицы. Например, если понадобится создать копию таблицы `Inventory` из базы данных `AutoLot` (см. главу 21), нужно будет создать четыре объекта `DataColumn`, по одному для каждого столбца (`CarID`, `Make`, `Color` и `PetName`). После создания объектов `DataColumn` они обычно добавляются в коллекцию столбцов типа `DataTable` (с помощью свойства `Columns`).

Возможно, вы уже знаете, что любому столбцу в таблице базы данных можно назначить набор ограничений (skonфигурировать как первичный ключ, присвоить стандартное значение, разрешить только чтение и т.п.). Кроме того, каждый столбец таблицы должен отображаться на лежащий в основе тип данных. К примеру, схема таблицы `Inventory` требует, чтобы столбец `CarID` отображался на целое число, а столбцы `Make`, `Color` и `PetName` — на массив символов. Класс `DataColumn` имеет ряд свойств для точного конфигурирования таких аспектов. Описания некоторых основных свойств приведены в табл. 22.3.

Таблица 22.3. Свойства класса `DataColumn`

Свойство	Описание
<code>AllowDBNull</code>	Это свойство указывает, может ли данный столбец содержать пустые значения. Стандартным значением является <code>true</code>
<code>AutoIncrement</code>	Эти свойства применяются для настройки поведения автоинкремента в заданном столбце. Это может оказаться удобным, если нужно гарантировать уникальность значений в <code>DataColumn</code> (например, когда он является первичным ключом). По умолчанию <code>DataColumn</code> не поддерживает автоинкрементное поведение
<code>AutoIncrementSeed</code>	
<code>AutoIncrementStep</code>	
<code>Caption</code>	Это свойство возвращает или устанавливает заголовок, который должен отображаться для столбца. Это позволяет определить дружественную к пользователю версию для имени столбца в базе данных

Свойство	Описание
ColumnMapping	Это свойство определяет представление DataColumn при сохранении DataSet в виде XML-документа с помощью метода DataSet.WriteXml(). Можно указать, что столбец данных должен быть записан как XML-элемент, XML-атрибут, простое текстовое содержимое либо вообще проигнорирован
ColumnName	Это свойство возвращает или устанавливает имя столбца из коллекции Columns (т.е. его внутреннее представление в DataTable). Если не установить ColumnName явно, то стандартным значением будет Column с числовым суффиксом по формуле $n+1$ (т.е. Column1, Column2, Column3 и т.д.)
DataType	Это свойство определяет тип данных (булевский, строковый, с плавающей точкой и т.д.), хранящихся в столбце
DefaultValue	Это свойство возвращает или устанавливает стандартное значение, присваиваемое столбцу при вставке новых строк
Expression	Это свойство возвращает или устанавливает выражение для фильтрации строк, вычисления значения столбца или создания агрегированного столбца
Ordinal	Это свойство возвращает или устанавливает числовое положение столбца в коллекции Columns, содержащейся в DataTable
ReadOnly	Это свойство определяет, предназначен ли данный столбец только для чтения после добавления строки в таблицу. Стандартным значением является false
Table	Это свойство возвращает объект DataTable, содержащий DataColumn
Unique	Это свойство возвращает или устанавливает значение, указывающее, должны ли быть уникальными значения во всех строках столбца, или же повторяющиеся значения разрешены. Когда столбцу назначается ограничение первичного ключа, свойство Unique должно быть установлено в true

Построение объекта DataColumn

Продолжая создание проекта SimpleDataSet (и демонстрацию применения типа DataColumn), предположим, что вам необходимо смоделировать столбцы таблицы Inventory. Поскольку столбец CarID будет первичным ключом таблицы, он должен быть сконфигурирован как предназначенный только для чтения, содержащий уникальные значения и не допускающим null (с помощью свойств ReadOnly, Unique и AllowDBNull). Добавьте в класс Program новый метод по имени FillDataSet(), предназначенный для построения четырех объектов DataColumn. Обратите внимание, что этот метод принимает в качестве единственного параметра объект DataSet.

```
static void FillDataSet(DataSet ds)
{
    // Создать столбцы данных, которые отображаются на "реальные"
    // столбцы в таблице Inventory из базы данных AutoLot.
    DataColumn carIDColumn = new DataColumn("CarID", typeof(int));
    carIDColumn.Caption = "Car ID";
    carIDColumn.ReadOnly = true;
    carIDColumn.AllowDBNull = false;
    carIDColumn.Unique = true;
    DataColumn carMakeColumn = new DataColumn("Make", typeof(string));
```

```

DataColumn carColorColumn = new DataColumn("Color", typeof(string));
DataColumn carPetNameColumn = new DataColumn("PetName", typeof(string));
carPetNameColumn.Caption = "Pet Name";
}

```

Обратите внимание, что при конфигурировании объекта `carIDColumn` было присвоено значение свойству `Caption`. Это позволяет определить строковое значение для отображения при выводе данных, которое может отличаться от имени столбца (имена столбцов в таблицах баз данных обычно более удобны для программирования (например, `au_fname`), чем для отображения (например, `Author First Name` (Имя автора))). По той же причине был установлен заголовок для столбца `PetName`, т.к. *Pet Name* понятнее конечному пользователю, чем *PetName*.

Включение автоинкрементных полей

Одним из аспектов `DataColumn`, допускающим конфигурирование, является возможность *автоинкремента*. Автоинкрементное поле используется для обеспечения того, что при добавлении в таблицу новой строки значение этого поля устанавливается автоматически на основе предыдущего значения и шага увеличения. Это удобно, когда нужно гарантировать, что в столбце отсутствуют повторяющиеся значения (например, в первичном ключе).

Это поведение управляется свойствами `AutoIncrement`, `AutoIncrementSeed` и `AutoIncrementStep`. Значение `AutoIncrementSeed` используется для определения начального значения в столбце, а `AutoIncrementStep` — для указания числа, которое прибавляется для каждого последующего значения. Рассмотрим следующую модификацию создания `carIDColumn`:

```

static void FillDataSet(DataSet ds)
{
    DataColumn carIDColumn = new DataColumn("CarID", typeof(int));
    carIDColumn.ReadOnly = true;
    carIDColumn.Caption = "Car ID";
    carIDColumn.AllowDBNull = false;
    carIDColumn.Unique = true;
    carIDColumn.AutoIncrement = true;
    carIDColumn.AutoIncrementSeed = 0;
    carIDColumn.AutoIncrementStep = 1;
    ...
}

```

Здесь объект `carIDColumn` сконфигурирован так, что при добавлении новых строк в соответствующую таблицу значение этого столбца увеличиваются на 1. Поскольку начальное значение установлено в 0, в столбце будут содержаться числа 0, 1, 2, 3 и т.д.

Добавление объектов DataColumn в DataTable

Обычно тип `DataColumn` не применяется как обособленная сущность, а вставляется в соответствующий объект `DataTable`. Для примера создайте новый объект `DataTable` (как будет показано ниже) и вставьте все объекты `DataColumn` в коллекцию столбцов с применением свойства `Columns`:

```

static void FillDataSet(DataSet ds):
{
    ...
    // Добавить объекты DataColumn в DataTable.
    DataTable inventoryTable = new DataTable("Inventory");
    inventoryTable.Columns.AddRange(new DataColumn[]
    { carIDColumn, carMakeColumn, carColorColumn, carPetNameColumn });
}

```


Теперь объект `DataTable` содержит четыре объекта `DataColumn` , которые представляют схему находящейся в памяти таблицы `Inventory`. Но пока эта таблица не содержит данных и не входит в коллекцию таблиц, обслуживаемых `DataSet`. Мы восполним эти пробелы, начав с наполнения таблицы данными с использованием объектов `DataRow`.

Работа с объектами DataRow

Как вы уже видели, коллекция объектов `DataColumn` представляет схему `DataTable`. В отличие от этого, коллекция объектов `DataRow` представляет действительные данные в таблице. Таким образом, если таблица `Inventory` базы данных `AutoLot` содержит 20 строк, то представить эти записи можно с помощью 20 объектов `DataRow`. В табл. 22.4 кратко описаны некоторые (но не все) члены типа `DataRow`.

Таблица 22.4. Основные члены типа DataRow

Член	Описание
<code>HasErrors</code> <code>GetColumnsInError()</code> <code>GetColumnError()</code> <code>ClearErrors()</code> <code>RowError</code>	Свойство <code>HasErrors</code> возвращает булевское значение, указывающее на наличие ошибок в <code>DataRow</code> . Если ошибки есть, можно воспользоваться методом <code>GetColumnsInError()</code> для получения проблемных столбцов и методом <code>GetColumnError()</code> для получения описания ошибки. С помощью метода <code>ClearErrors()</code> можно очистить список ошибок для строки. Свойство <code>RowError</code> позволяет создать текстовое описание ошибки для заданной строки
<code>ItemArray</code>	Это свойство возвращает или устанавливает все значения столбцов для строки, используя массив объектов
<code>RowState</code>	Это свойство позволяет определить текущее состояние объекта <code>DataRow</code> в содержащем его объекте <code>DataTable</code> с помощью значений перечисления <code>RowState</code> (например, строка может быть помечена как новая, измененная, не измененная или удаленная)
<code>Table</code>	Это свойство используется для получения ссылки на объект <code>DataTable</code> , содержащий текущий объект <code>DataRow</code>
<code>AcceptChanges()</code> <code>RejectChanges()</code>	Эти методы предназначены для фиксации или отклонения всех изменений, сделанных в текущей строке с момента последнего вызова <code>AcceptChanges()</code>
<code>BeginEdit()</code> <code>EndEdit()</code> <code>CancelEdit()</code>	Эти методы начинают, заканчивают или отменяют операцию редактирования для объекта <code>DataRow</code>
<code>Delete()</code>	Этот метод помечает строку, подлежащую удалению при вызове метода <code>AcceptChanges()</code>
<code>IsNull()</code>	Этот метод возвращает значение, которое указывает, содержит ли заданный столбец <code>null</code>

Работа с объектами `DataRow` несколько отличается от работы с `DataColumn` ; создавать экземпляр этого типа напрямую невозможно, т.к. в нем нет открытых конструкторов:

```
// Ошибка! Отсутствуют открытые конструкторы!  
DataRow r = new DataRow();
```

Вместо этого новый объект `DataRow` получается из указанного объекта `DataTable`. Например, предположим, что в таблицу `Inventory` требуется вставить две строки. Метод `DataTable.NewRow()` позволяет получить очередную область в таблице, после чего можно заполнить каждый столбец новыми данными с помощью индексатора типа.

При этом можно указать либо строковое имя, назначенное объекту DataColumn, либо номер его позиции (начинающийся с нуля):

```
static void FillDataSet(DataSet ds)
{
    ...
    // Добавить несколько строк в таблицу Inventory.
    DataRow carRow = inventoryTable.NewRow();
    carRow["Make"] = "BMW";
    carRow["Color"] = "Black";
    carRow["PetName"] = "Hamlet";
    inventoryTable.Rows.Add(carRow);

    carRow = inventoryTable.NewRow();
    // Столбец 0 - это автоинкрементное поле идентификатора,
    // поэтому начать заполнение со столбца 1.
    carRow[1] = "Saab";
    carRow[2] = "Red";
    carRow[3] = "Sea Breeze";
    inventoryTable.Rows.Add(carRow);
}
```

На заметку! В случае передачи методу индекса типа DataRow недопустимого имени столбца или позиции во время выполнения сгенерируется исключение.

К этому моменту имеется один объект DataTable, содержащий две строки. Разумеется, этот общий процесс можно повторить, чтобы создать нужное количество объектов DataTable, определить схему и заполнить данными. Но перед вставкой объекта inventoryTable в DataSet необходимо разобраться с очень важным свойством RowState.

Свойство RowState

Свойство RowState применяется для программной идентификации набора строк таблицы, которые изменили свое исходное значение, были вставлены и т.п. Это свойство может принимать любое значение из перечисления DataRowState, которое кратко описано в табл. 22.5.

Таблица 22.5. Значения перечисления DataRowState

Значение	Описание
Added	Строка была добавлена в DataRowCollection, а метод AcceptChanges () еще не был вызван
Deleted	Строка была помечена для удаления с помощью метода Delete () класса DataRow, а метод AcceptChanges () еще не был вызван
Detached	Строка была создана, но не является частью какого-либо объекта DataRowCollection. Объект DataRow находится в этом состоянии непосредственно после создания, но до добавления его в коллекцию либо после удаления из коллекции
Modified	Строка была изменена, а AcceptChanges () еще не был вызван
Unchanged	Строка не была изменена с момента последнего вызова AcceptChanges ()

При программном манипулировании строками заданного объекта DataTable свойство RowState устанавливается автоматически. Для примера добавьте в свой класс Program новый метод, который оперирует на локальном объекте DataRow, попутно выводя на консоль состояние его строк:

```
private static void ManipulateDataRowState()
{
    // Создать объект temp типа DataTable для тестирования.
    DataTable temp = new DataTable("Temp");
    temp.Columns.Add(new DataColumn("TempColumn", typeof(int)));
    // RowState = Detached (т.е. пока еще не является частью DataTable).
    DataRow row = temp.NewRow();
    Console.WriteLine("After calling NewRow(): {0}", row.RowState);
    // RowState = Added.
    temp.Rows.Add(row);
    Console.WriteLine("After calling Rows.Add(): {0}", row.RowState);
    // RowState = Added.
    row["TempColumn"] = 10;
    Console.WriteLine("After first assignment: {0}", row.RowState);
    // RowState = Unchanged.
    temp.AcceptChanges();
    Console.WriteLine("After calling AcceptChanges: {0}", row.RowState);
    // RowState = Modified.
    row["TempColumn"] = 11;
    Console.WriteLine("After first assignment: {0}", row.RowState);
    // RowState = Deleted.
    temp.Rows[0].Delete();
    Console.WriteLine("After calling Delete: {0}", row.RowState);
}
```

Объект DataRow в ADO.NET достаточно интеллектуален, чтобы отслеживать свое текущее состояние. С учетом этого, владеющий им объект DataTable способен идентифицировать, какие строки были добавлены, обновлены или удалены. Это очень важная функциональная возможность DataSet, потому что когда наступит время передачи обновленной информации в хранилище данных, будут отправлены только измененные данные.

Свойство DataRowVersion

Помимо отслеживания текущего состояния строк с помощью свойства RowState, объект DataRow поддерживает три возможных версии содержащихся в нем данных посредством свойства DataRowVersion. При первоначальном конструировании объект DataRow содержит лишь одну копию данных, которая считается текущей версией. Однако по мере программного манипулирования объектом DataRow (с помощью вызовов различных методов) появляются дополнительные версии данных. Свойство DataRowVersion может быть установлено в любое значение перечисления DataRowVersion (табл. 22.6).

Таблица 22.6. Значения перечисления DataRowVersion

Значение	Описание
Current	Представляет текущее значение строки, даже после внесения изменений
Default	Стандартная версия DataRowState. Если значение DataRowState равно Added, Modified или Deleted, то стандартной версией является Current. Для значения DataRowState, равного Detached, стандартной версией является Proposed
Original	Представляет значение, первоначально вставленное в DataRow, или значение при последнем вызове AcceptChanges()
Proposed	Значение строки, редактируемой в настоящий момент в результате вызова BeginEdit()

Как показано в табл. 22.6, значение свойства `DataRowVersion` во многих случаях зависит от значения свойства `DataRowState`. Как упоминалось ранее, свойство `DataRowVersion` изменяется автоматически при вызовах различных методов объекта `DataRow` (или в некоторых случаях `DataTable`). Ниже представлена схема влияния этих методов на значение свойства `DataRowVersion` произвольной строки.

- Если вызывается метод `DataRow.BeginEdit()` и изменяется значение строки, становятся доступными значения `Current` и `Proposed`.
- Если вызывается метод `DataRow.CancelEdit()`, значение `Proposed` удаляется.
- После вызова `DataRow.EndEdit()` значение `Proposed` становится значением `Current`.
- После вызова метода `DataRow.AcceptChanges()` значение `Original` становится идентичным значению `Current`. То же самое происходит и при вызове `DataTable.AcceptChanges()`.
- После вызова `DataRow.RejectChanges()` значение `Proposed` отбрасывается, и версия становится `Current`.

Действительно, все это несколько запутанно, и не в последнюю очередь потому, что в любой момент времени `DataRow` может иметь, а может и не иметь все версии (при попытке получить версию строки, которая в текущий момент не отслеживается, генерируются исключения времени выполнения). Несмотря на эту сложность, с учетом того, что `DataRow` поддерживает три копии данных, становится более простым построение клиентского приложения, которое позволяет конечному пользователю изменить значения, а затем отказаться от изменений или зафиксировать новые значения для постоянного хранения. В оставшейся части этой главы будут приведены разнообразные примеры использования этих методов.

Работа с объектами `DataTable`

Тип `DataTable` определяет значительное количество членов, многие из которых совпадают по именам и функциональности с аналогичными членами `DataSet`. В табл. 22.7 приведены краткие описания некоторых основных членов типа `DataTable`, кроме `Rows` и `Columns`.

Таблица 22.7. Основные члены типа `DataTable`

Член	Описание
<code>CaseSensitive</code>	Указывает, являются ли сравнения строк чувствительными к регистру символов. Стандартное значение равно <code>false</code>
<code>ChildRelations</code>	Возвращает коллекцию дочерних отношений для этого объекта <code>DataTable</code> (если они есть)
<code>Constraints</code>	Возвращает коллекцию ограничений, поддерживаемых таблицей
<code>Copy()</code>	Метод, который копирует схему и данные из объекта <code>DataTable</code> в новый такой объект
<code>DataSet</code>	Возвращает <code>DataSet</code> , содержащий эту таблицу (если он есть)
<code>DefaultView</code>	Возвращает настроенное представление таблицы, которое может содержать отфильтрованное представление или позицию курсора
<code>ParentRelations</code>	Возвращает коллекцию родительских отношений для этого объекта <code>DataTable</code>

Член	Описание
PrimaryKey	Возвращает или устанавливает массив столбцов, которые выступают в качестве первичных ключей для таблицы данных
TableName	Возвращает или устанавливает имя таблицы. Значение для этого свойства можно также указать через параметр конструктора

В продолжение текущего примера установим для свойства `PrimaryKey` в `DataTable` объект `DataColumn` по имени `carIDColumn`. Обратите внимание, что свойство `PrimaryKey` является коллекцией объектов `DataColumn`, чтобы учесть ключи, состоящие из нескольких столбцов. Однако в рассматриваемом случае необходимо указать только столбец `CarID` (который находится в самой первой позиции таблицы):

```
static void FillDataSet(DataSet ds)
{
    ...
    // Задать первичный ключ этой таблицы.
    inventoryTable.PrimaryKey = new DataColumn[] { inventoryTable.Columns[0] };
}
```

Вставка объектов `DataTable` в `DataSet`

Итак, объект `DataTable` завершен. Осталось вставить его в объект `carsInventoryDS` типа `DataSet`, используя коллекцию `Tables`:

```
static void FillDataSet(DataSet ds)
{
    ...
    // Наконец, добавить таблицу в DataSet.
    ds.Tables.Add(inventoryTable);
}
```

Теперь модифицируйте метод `Main()`, добавив вызов `FillDataSet()` с передачей в качестве аргумента локального объекта `DataSet`. Затем передайте этот объект новому (пока еще не написанному) вспомогательному методу по имени `PrintDataSet()`:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with DataSets *****\n");
    ...
    FillDataSet(carsInventoryDS);
    PrintDataSet(carsInventoryDS);
    Console.ReadLine();
}
```

Получение данных из объекта `DataSet`

Метод `PrintDataSet()` просто проходит по всем метаданным `DataSet` (используя коллекцию `ExtendedProperties`) и по всем `DataTable` в этом `DataSet`, выводя на консоль имена столбцов и значения строк с помощью индексов типов:

```
static void PrintDataSet(DataSet ds)
{
    // Вывести имя DataSet и любые расширенные свойства.
    Console.WriteLine("DataSet is named: {0}", ds.DataSetName);
    foreach (System.Collections.DictionaryEntry de in ds.ExtendedProperties)
    {
        Console.WriteLine("Key = {0}, Value = {1}", de.Key, de.Value);
    }
}
```

```

Console.WriteLine();
// Вывести каждую таблицу.
foreach (DataTable dt in ds.Tables)
{
    Console.WriteLine("=> {0} Table:", dt.TableName);
    // Вывести имена столбцов.
    for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
    {
        Console.Write(dt.Columns[curCol].ColumnName + "\t");
    }
    Console.WriteLine("\n-----");
    // Вывести DataTable.
    for (int curRow = 0; curRow < dt.Rows.Count; curRow++)
    {
        for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
        {
            Console.Write(dt.Rows[curRow][curCol].ToString() + "\t");
        }
        Console.WriteLine();
    }
}
}

```

Если теперь запустить программу, будет получен следующий вывод (разумеется, метка времени и значение GUID будут отличаться):

```

***** Fun with DataSets *****

DataSet is named: Car Inventory
Key = Timestamp, Value = 1/22/2012 6:41:09 AM
Key = DataSetID, Value = 11c533ed-d1aa-4c82-96d4-b0f88893ab21
Key = Company, Value = Mikko's Hot Tub Super Store

=> Inventory Table:
CarID Make   Color  PetName
-----
0      BMW    Black  Hamlet
1      Saab    Red    Sea Breeze

```

Обработка данных из DataTable с использованием объектов DataTableReader

Учитывая проделанную в главе 21 работу, вы должны заметить, что способы обработки данных с применением подключенного уровня (объектов чтения данных) и автономного уровня (объектов DataSet) существенно отличаются. При работе с объектом чтения данных обычно организуется цикл while, вызывается метод Read() и с использованием индекса выбираются пары "имя/значение". С другой стороны, обработка DataSet, как правило, предусматривает наличие последовательности итерационных конструкций, позволяющей добраться до данных внутри таблиц, строк и столбцов (не забывайте, что объект DataReader требует открытого подключения к базе данных, чтобы он мог прочитать данные из действительной базы).

Объекты DataTable поддерживают метод по имени CreateDataReader(). Указанный метод позволяет получать данные из DataTable с применением схемы навигации, подобной объекту чтения данных (в этом случае данные будут читаться из находящегося в памяти объекта DataTable, а не из реальной базы данных, поэтому подключение к базе данных здесь не требуется). Основное преимущество такого подхода состоит в том, что при обработке данных используется единая модель, независимо от уровня ADO.NET,

применяемого для получения этих данных. Предположим, что в класс Program добавлен следующий вспомогательный метод по имени PrintTable():

```
static void PrintTable(DataTable dt)
{
    // Получить объект DataTableReader.
    DataTableReader dtReader = dt.CreateDataReader();
    // DataTableReader работает в точности как DataReader.
    while (dtReader.Read())
    {
        for (int i = 0; i < dtReader.FieldCount; i++)
        {
            Console.Write("{0}\t", dtReader.GetValue(i).ToString().Trim());
        }
        Console.WriteLine();
    }
    dtReader.Close();
}
```

Обратите внимание, что DataTableReader работает идентично объекту чтения данных используемого поставщика данных. Тип DataTableReader может быть идеальным вариантом, когда необходимо быстро извлечь данные из DataTable, не занимаясь обходом внутренних коллекций строк и столбцов. Теперь предположим, что предыдущий метод PrintDataSet() модифицирован для обращения к PrintTable() вместо работы с коллекциями Rows и Columns:

```
static void PrintDataSet(DataSet ds)
{
    // Вывести имя DataSet и любые расширенные свойства.
    Console.WriteLine("DataSet is named: {0}", ds.DataSetName);
    foreach (System.Collections.DictionaryEntry de in ds.ExtendedProperties)
    {
        Console.WriteLine("Key = {0}, Value = {1}", de.Key, de.Value);
    }
    Console.WriteLine();
    foreach (DataTable dt in ds.Tables)
    {
        Console.WriteLine("=> {0} Table:", dt.TableName);
        // Вывести имена столбцов.
        for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
        {
            Console.Write(dt.Columns[curCol].ColumnName.Trim() + "\t");
        }
        Console.WriteLine("\n-----");
        // Вызвать новый вспомогательный метод.
        PrintTable(dt);
    }
}
```

В результате запуска приложения получается вывод, идентичный приведенному выше. Единственная разница связана с тем, как осуществляется внутренний доступ к содержимому DataTable.

Сериализация объектов DataTable и DataSet в формате XML

Типы DataSet и DataTable поддерживают методы WriteXml() и ReadXml(). Метод WriteXml() позволяет сохранить содержимое объекта в локальном файле (а также в лю-

бом типе, производном от `System.IO.Stream`) как XML-документ. Метод `ReadXml()` позволяет восстановить состояние `DataSet` (или `DataTable`) из указанного XML-документа. Кроме этого, типы `DataSet` и `DataTable` поддерживают методы `WriteXmlSchema()` и `ReadXmlSchema()`, предназначенные для сохранения или загрузки файла *.xsd.

Для проверки их работы измените код `Main()`, чтобы в нем вызывался следующий вспомогательный метод (которому передается единственный параметр типа `DataSet`):

```
static void SaveAndLoadAsXml(DataSet carsInventoryDS)
{
    // Сохранить этот DataSet в виде XML.
    carsInventoryDS.WriteXml("carsDataSet.xml");
    carsInventoryDS.WriteXmlSchema("carsDataSet.xsd");

    // Очистить DataSet.
    carsInventoryDS.Clear();

    // Загрузить DataSet из файла XML.
    carsInventoryDS.ReadXml("carsDataSet.xml");
}
```

Если открыть файл `carsDataSet.xml` (который находится в папке `\bin\Debug` проекта), то можно увидеть, что каждый столбец таблицы закодирован в виде XML-элемента:

```
<?xml version="1.0" standalone="yes"?>
<Car_x0020_Inventory>
  <Inventory>
    <CarID>0</CarID>
    <Make>BMW</Make>
    <Color>Black</Color>
    <PetName>Hamlet</PetName>
  </Inventory>
  <Inventory>
    <CarID>1</CarID>
    <Make>Saab</Make>
    <Color>Red</Color>
    <PetName>Sea Breeze</PetName>
  </Inventory>
</Car_x0020_Inventory>
```

Двойной щелчок на сгенерированном файле .xsd (который также находится в папке `\bin\Debug`) в Visual Studio приводит к открытию встроенного редактора схемы XML (рис. 22.3).

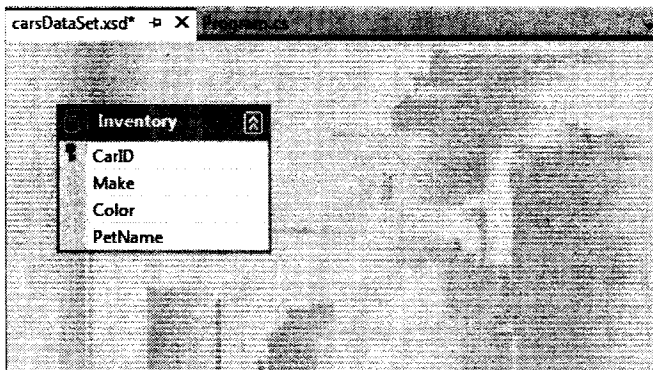


Рис. 22.3. Редактор XSD в Visual Studio

На заметку! В главе 24 будет представлен API-интерфейс LINQ to XML, который является рекомендуемым способом манипулирования XML-данными в рамках платформы .NET.

Сериализация объектов DataTable и DataSet в двоичном формате

Содержимое объекта DataSet (или отдельного DataTable) можно также сохранить в компактном двоичном формате. Это особенно полезно, когда необходимо передать объект DataSet за границы компьютера (в случае распределенного приложения). Один из недостатков представления данных в виде XML заключается в том, что его дескриптивная природа может привести к высоким накладным расходам при передаче.

Чтобы сохранить объект DataTable или DataSet в двоичном формате, установите свойство RemotingFormat в SerializationFormat.Binary. После этого, как нетрудно догадаться, можно использовать тип BinaryFormatter (см. главу 20). Рассмотрим следующий финальный метод проекта SimpleDataSet (не забудьте импортировать пространства имен System.IO и System.Runtime.Serialization.Formatters.Binary):

```
static void SaveAndLoadAsBinary(DataSet carsInventoryDS)
{
    // Установить флаг двоичной сериализации.
    carsInventoryDS.RemotingFormat = SerializationFormat.Binary;

    // Сохранить этот DataSet в двоичном виде.
    FileStream fs = new FileStream("BinaryCars.bin", FileMode.Create);
    BinaryFormatter bFormat = new BinaryFormatter();
    bFormat.Serialize(fs, carsInventoryDS);
    fs.Close();

    // Очистить DataSet.
    carsInventoryDS.Clear();

    // Загрузить DataSet из двоичного файла.
    fs = new FileStream("BinaryCars.bin", FileMode.Open);
    DataSet data = (DataSet)bFormat.Deserialize(fs);
}
```

После вызова этого метода в Main() в папке bin\Debug можно будет найти файл *.bin. Содержимое файла BinaryCars.bin показано на рис. 22.4.

Исходный код. Проект SimpleDataSet доступен в подкаталоге Chapter 22.

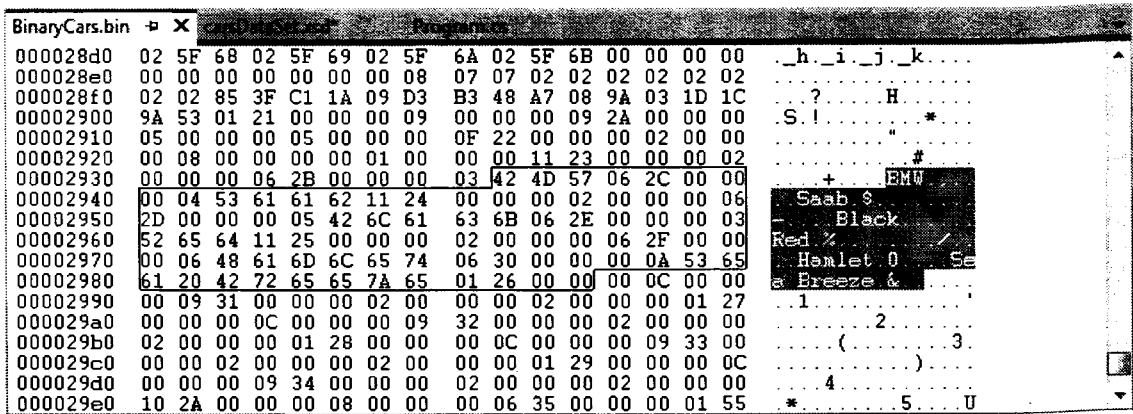


Рис. 22.4. Объект DataSet сохранен в двоичном формате

Привязка объектов DataTable к графическим пользовательским интерфейсам Windows Forms

До сих пор вы видели, как создавать, заполнять и проходить по содержимому объекта DataSet вручную с применением внутренней объектной модели ADO.NET. Хотя знание всего этого довольно важно, в рамках платформы .NET поставляются многочисленные API-интерфейсы, которые обладают возможностью автоматической привязки данных к элементам пользовательского интерфейса.

Например, в первоначальном инструментальном наборе для построения графических пользовательских интерфейсов .NET — Windows Forms — имеется элемент управления по имени DataGridView, который позволяет отображать содержимое объекта DataSet или DataTable посредством написания всего нескольких строк кода.

ASP.NET (API-интерфейс для разработки веб-приложений в .NET) и Windows Presentation Foundation (новый и более мощный API-интерфейс для построения графических пользовательских интерфейсов, появившийся в .NET 3.0) также поддерживают понятие привязки данных.

Вы научитесь привязывать данные к графическим элементам WPF и ASP.NET позже в этой книге; однако в настоящей главе будет применяться Windows Forms, т.к. это довольно простая и понятная модель программирования.

На заметку! В следующем примере предполагается, что у вас есть некоторый опыт использования Windows Forms для создания графических пользовательских интерфейсов. Если это не так, можете просто открыть готовое решение (доступное в загружаемом коде для книги) и проследить за изложением.

Задача заключается в построении приложения Windows Forms, которое будет отображать внутри своего пользовательского интерфейса содержимое объекта DataTable. Попутно будет показано, как фильтровать и изменять данные в таблице, а также рассмотрена роль объекта DataView.

Начните с создания нового рабочего пространства проекта Windows Forms по имени WindowsFormsDataBinding. Измените в Solution Explorer имя первоначального файла Form1.cs на более понятное MainForm.cs. Затем перетащите из панели инструментов Visual Studio элемент DataGridView на поверхность визуального конструктора (и переименуйте его в carInventoryGridView с использованием свойства (Name) в окне Properties (Свойства)).

Вы можете заметить, что при первом добавлении элемента DataGridView в визуальный конструктор активизируется контекстное меню, которое позволяет подключиться к физическому источнику данных. Пока не обращайтесь на это внимание, поскольку привязка объекта DataTable будет осуществляться программно. И, наконец, добавьте в визуальный конструктор элемент Label с понятным текстом.

Один из возможных вариантов внешнего вида пользовательского интерфейса показан на рис. 22.5.

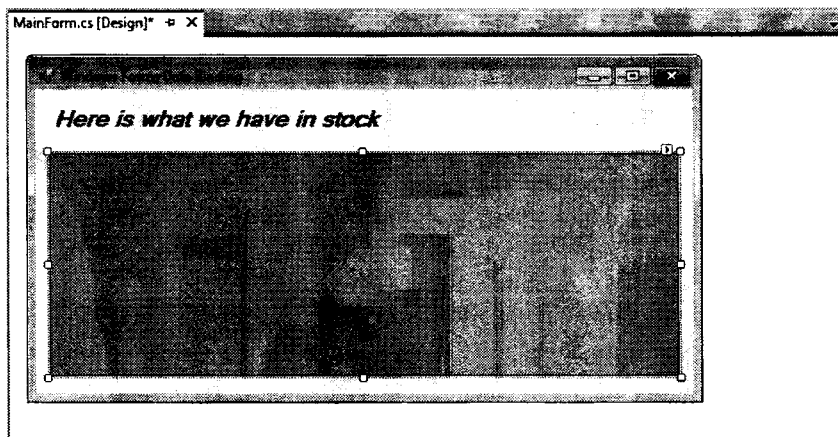


Рис. 22.5. Первоначальный графический пользовательский интерфейс приложения Windows Forms

Заполнение DataTable из обобщенного List<T>

Аналогично предыдущему примеру SimpleDataSet, в приложении WindowsForms DataBinding будет конструироваться объект DataTable, содержащий несколько DataColumn, которые будут представлять различные столбцы и строки данных. Но на этот раз строки будут заполняться с помощью переменной-члена обобщенного типа List<T>. Для начала вставьте в проект новый класс C# по имени Car, определенный следующим образом:

```
public class Car
{
    public int ID { get; set; }
    public string PetName { get; set; }
    public string Make { get; set; }
    public string Color { get; set; }
}
```

Внутри стандартного конструктора главной формы заполните переменную-член типа List<T> (с именем listCars) набором новых объектов Car:

```
public partial class MainForm : Form
{
    // Коллекция объектов Car.
    List<Car> listCars = null;
    public MainForm()
    {
        InitializeComponent();
        // Заполнить список несколькими автомобилями.
        listCars = new List<Car>
        {
            new Car { ID = 100, PetName = "Chucky", Make = "BMW", Color = "Green" },
            new Car { ID = 101, PetName = "Tiny", Make = "Yugo", Color = "White" },
            new Car { ID = 102, PetName = "Ami", Make = "Jeep", Color = "Tan" },
            new Car { ID = 103, PetName = "Pain Inducer", Make = "Caravan", Color = "Pink" },
            new Car { ID = 104, PetName = "Fred", Make = "BMW", Color = "Green" },
            new Car { ID = 105, PetName = "Sidd", Make = "BMW", Color = "Black" },
            new Car { ID = 106, PetName = "Mel", Make = "Firebird", Color = "Red" },
            new Car { ID = 107, PetName = "Sarah", Make = "Colt", Color = "Black" },
        };
    }
}
```

Добавьте в класс MainForm новую переменную-член типа DataTable по имени inventoryTable:

```
public partial class MainForm : Form
{
    // Коллекция объектов Car.
    List<Car> listCars = null;

    // Складская информация.
    DataTable inventoryTable = new DataTable();
    ...
}
```

Теперь добавьте в этот же класс новый вспомогательный метод CreateDataTable() и вызовите его в стандартном конструкторе класса MainForm:

```
private void CreateDataTable()
{
    // Создать схему таблицы.
    DataColumn carIDColumn = new DataColumn("ID", typeof(int));
    DataColumn carMakeColumn = new DataColumn("Make", typeof(string));
    DataColumn carColorColumn = new DataColumn("Color", typeof(string));
    DataColumn carPetNameColumn = new DataColumn("PetName", typeof(string));
    carPetNameColumn.Caption = "Pet Name";
    inventoryTable.Columns.AddRange(new DataColumn[] { carIDColumn,
        carMakeColumn, carColorColumn, carPetNameColumn });

    // Пройти по List<T> для создания строк.
    foreach (Car c in listCars)
    {
        DataRow newRow = inventoryTable.NewRow();
        newRow["ID"] = c.ID;
        newRow["Make"] = c.Make;
        newRow["Color"] = c.Color;
        newRow["PetName"] = c.PetName;
        inventoryTable.Rows.Add(newRow);
    }

    // Привязать DataTable к carInventoryGridView.
    carInventoryGridView.DataSource = inventoryTable;
}
```

Реализация метода начинается с создания схемы DataTable путем конструирования четырех объектов DataColumn (для простоты включать автоинкрементирование для поля CarID или устанавливать его в качестве первичного ключа не обязательно). После этого их можно добавить в коллекцию столбцов, доступную через переменную-член DataTable. Данные строк из коллекции List<T> отображаются на DataTable с помощью итерационной конструкции foreach и собственной объектной модели ADO.NET.

Однако обратите внимание, что в последнем операторе кода внутри метода CreateDataTable() таблица inventoryTable присваивается свойству DataSource объекта DataGridView. Это свойство является единственным, которое понадобится установить для привязки DataTable к объекту DataGridView из Windows Forms. “За кулисами” этот элемент управления графического пользовательского интерфейса выполняет внутреннее чтение коллекций строк и столбцов, почти так же, как это происходило в методе PrintDataSet() примера SimpleDataSet. В этот момент должна уже быть возможность запуска приложения и просмотра содержимого DataTable внутри элемента управления DataGridView (рис. 22.6).

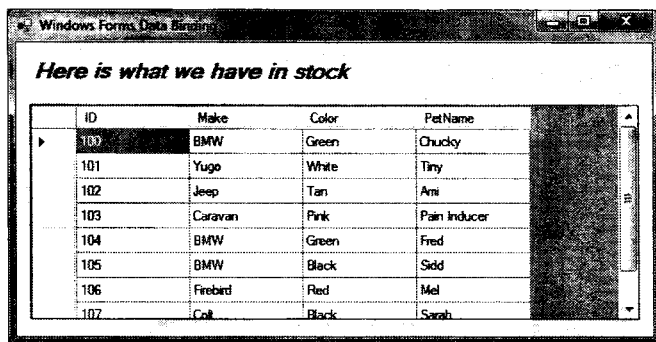


Рис. 22.6. Привязка объекта DataTable к элементу DataGridView из Windows Forms

Удаление строк из DataTable

Теперь предположим, что необходимо добавить к графическому интерфейсу возможность удаления строки из находящегося в памяти объекта DataTable, который привязан к элементу DataGridView. Один из возможных подходов предусматривает вызов метода `Delete()` объекта `DataRow`, который представляет удаляемую строку. В этом случае указывается индекс (или объект `DataRow`), соответствующий этой строке. Чтобы пользователь мог выбрать строку, подлежащую удалению, добавьте на поверхность визуального конструктора элементы управления `TextBox` (по имени `txtRowToRemove`) и `Button` (по имени `btnRemoveRow`). На рис. 22.7 показан возможный внешний вид интерфейса после изменения (обратите внимание на группирование этих двух элементов управления посредством `GroupBox`, что подчеркивает их взаимосвязь).

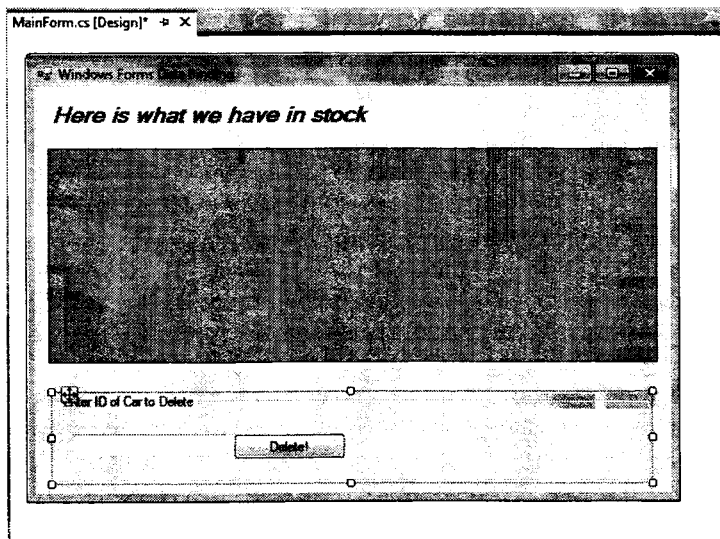


Рис. 22.7. Измененный пользовательский интерфейс для удаления строк из DataTable

Ниже приведен код обработчика события `Click` новой кнопки, который удаляет указанную пользователем строку на основе идентификатора автомобиля из находящегося в памяти объекта `DataTable`. Метод `Select()` класса `DataTable` позволяет задавать критерий поиска, который похож на обычный синтаксис `SQL`. Возвращаемым значением является массив объектов `DataRow`, удовлетворяющих критерию поиска.

```
// Удалить эту строку из DataRowCollection.
private void btnRemoveCar_Click (object sender, EventArgs e)
{
    try
    {
        // Найти правильную строку для удаления.
        DataRow[] rowToDelete = inventoryTable.Select (
            string.Format ("ID={0}", int.Parse (txtCarToRemove.Text)));
        // Удалить ее.
        rowToDelete[0].Delete();
        inventoryTable.AcceptChanges();
    }
    catch (Exception ex)
    {
        MessageBox.Show (ex.Message);
    }
}
```

Теперь можно запустить приложение и указать идентификатор удаляемого автомобиля. При удалении объектов `DataRow` из `DataTable` пользовательский интерфейс `DataGridView` обновляется немедленно, поскольку он привязан к состоянию объекта `DataTable`.

Выборка строк на основе критерия фильтрации

Во многих приложениях обработки данных бывает необходимо просматривать небольшое подмножество данных из `DataTable` на основе какого-то критерия фильтрации. Например, предположим, что требуется видеть только автомобили марки BMW, которые хранятся внутри `DataTable` в памяти. Ранее уже было показано, что метод `Select()` класса `DataTable` позволяет найти строку, подлежащую удалению, но его можно применять и для выборки подмножества записей с целью их отображения.

Чтобы увидеть это в действии, снова измените пользовательский интерфейс, на этот раз предоставив пользователям возможность указывать строку, которая представляет интересующую модель автомобиля (рис. 22.8), с применением новых элементов `TextBox` (по имени `txtMakeToView`) и `Button` (по имени `btnDisplayMakes`).



Рис. 22.8. Добавление к пользовательскому интерфейсу возможности фильтрации строк

Метод `Select()` имеет несколько перегруженных версий, предлагающих различную семантику выборки. На самом простом уровне передаваемый в `Select()` параметр является строкой, которая содержит какое-то условное выражение. Для начала рассмотрим следующую логику обработки события `Click` только что добавленной кнопки:

```
private void btnDisplayMakes_Click(object sender, EventArgs e)
{
    // Построить фильтр на основе пользовательского ввода.
    string filterStr = string.Format("Make= '{0}'", txtMakeToView.Text);

    // Найти все строки, удовлетворяющие фильтру.
    DataRow[] makes = inventoryTable.Select(filterStr);

    if (makes.Length == 0)
        MessageBox.Show("Sorry, no cars...", "Selection error!");
    else
    {
        string strMake = "";
        for (int i = 0; i < makes.Length; i++)
        {
            // Получить значение PetName из текущей строки.
            strMake += makes[i]["PetName"] + "\n";
        }

        // Вывести имена всех найденных автомобилей указанной марки.
        MessageBox.Show(strMake,
            string.Format("We have {0}s named:", txtMakeToView.Text));
    }
}
```

Здесь сначала создается простой фильтр на основе значения из соответствующего элемента `TextBox`. Если в этом поле ввести `BMW`, то получится следующий фильтр:

```
Make = 'BMW'
```

Передача этого фильтра методу `Select()` приводит к возврату массива объектов `DataRow`, представляющих строки, которые удовлетворяют заданному критерию фильтрации (рис. 22.9).

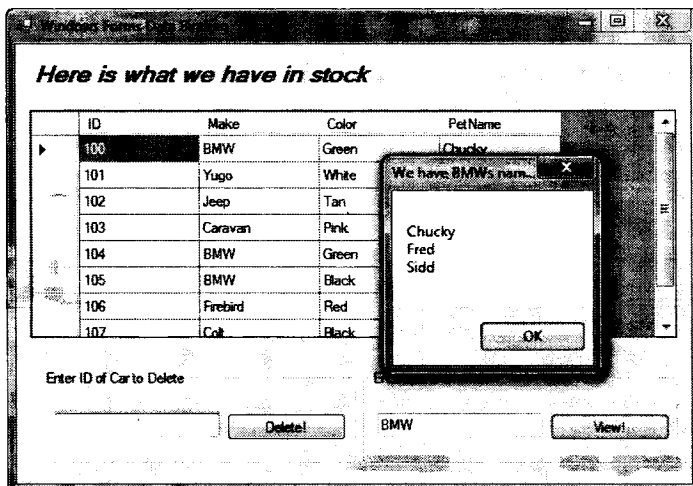


Рис. 22.9. Отображение фильтрованных данных

Логика фильтрации основана на стандартом синтаксисе SQL. Для примера предположим что результаты, полученные от предыдущего вызова `Select()`, нужно выдать упорядоченными по столбцу `PetName`. К счастью, имеется перегруженный вариант метода `Select()`, позволяющий указать критерий сортировки:

```
// Сортировать по PetName.
makes = inventoryTable.Select(filterStr, "PetName");
```

Если необходимо вывести результаты в убывающем порядке, вызовите метод `Select()`, как показано ниже:

```
// Возвратить результаты в убывающем порядке.
makes = inventoryTable.Select(filterStr, "PetName DESC");
```

В общем случае строка с критерием сортировки должна содержать имя столбца, за которым идет слово `ASC` (по возрастанию; принято по умолчанию) или `DESC` (по убыванию). При необходимости можно указать несколько столбцов, разделенных запятыми. И, наконец, строка с критерием фильтрации может содержать любое количество операций отношения. Например, вот вспомогательный метод, который выполняет поиск всех автомобилей со значением идентификатора больше 5:

```
private void ShowCarsWithIdGreaterThanOrEqualToFive()
{
    // Вывести дружественные имена всех автомобилей со значением ID больше 5.
    DataRow[] properIDs;
    string newFilterStr = "ID > 5";
    properIDs = inventoryTable.Select(newFilterStr);
    string strIDs = null;
    for(int i = 0; i < properIDs.Length; i++)
    {
        DataRow temp = properIDs[i];
        strIDs += temp["PetName"]
            + " is ID " + temp["ID"] + "\n";
    }
    MessageBox.Show(strIDs, "Pet names of cars where ID > 5");
}
```

Обновление строк в DataTable

Последний аспект `DataTable`, с которым следует ознакомиться — это процесс обновления существующих строк новыми значениями. Один из подходов предусматривает сначала получение строки (или строк), удовлетворяющей заданному критерию фильтрации, с помощью метода `Select()`. После получения объекта (объектов) `DataRow` необходимо соответствующим образом модифицировать содержимое. Предположим, например, что на форме имеется новый элемент `Button` по имени `btnChangeMakes`, при щелчке на котором выполняется поиск в `DataTable` всех строк, где столбец `Make` содержит значение `BMW`. После получения этих элементов значение `Make` изменяется на `Yugo`:

```
// Найти с помощью фильтра все строки, которые нужно отредактировать.
private void btnChangeMakes_Click(object sender, EventArgs e)
{
    // Подтвердить выбор.
    if (DialogResult.Yes ==
        MessageBox.Show("Are you sure?? BMWs are much nicer than Yugos!",
            "Please Confirm!", MessageBoxButtons.YesNo))
    {
        // Построить фильтр.
        string filterStr = "Make='BMW'";
        string strMake = string.Empty;
```



```
// Найти все строки, удовлетворяющие фильтру.
DataRow[] makes = inventoryTable.Select(filterStr);
// Заменить все BMW на Yugo.
for (int i = 0; i < makes.Length; i++)
{
    makes[i]["Make"] = "Yugo";
}
}
```

Работа с типом DataView

Объект представления — это альтернативный вид таблицы (или набора таблиц). Например, с помощью Microsoft SQL Server можно создать представление для таблицы Inventory, которое возвращает новую таблицу, содержащую автомобили только определенного цвета. В ADO.NET тип DataView позволяет программным образом извлекать подмножество данных из DataTable в отдельный объект.

Серьезное преимущество наличия нескольких представлений одной и той же таблицы состоит в том, что эти представления можно привязать к различным виджетам графического пользовательского интерфейса (таким как DataGridView). Например, один DataGridView может быть привязан к DataView, показывающему все автомобили из таблицы Inventory, а другой можно сконфигурировать на вывод только автомобилей зеленого цвета.

Для примера добавьте в текущий пользовательский интерфейс еще один элемент DataGridView по имени dataGridColtsView и элемент Label с пояснением. Затем опделите переменную-член типа DataView по имени coltsOnlyView:

```
public partial class MainForm : Form
{
    // Представление DataTable.
    DataView yugosOnlyView;
    ...
}
```

Теперь создайте новый вспомогательный метод CreateDataView() и вызовите его в стандартном конструкторе главной формы непосредственно после завершения создания DataTable, как показано ниже:

```
public MainForm()
{
    ...
    // Создать таблицу данных.
    CreateDataTable();
    // Создать представление.
    CreateDataView();
}
```

Ниже показана реализация этого нового вспомогательного метода. Обратите внимание, что конструктору каждого DataView передается объект DataTable, который будет использован для построения специального набора строк данных.

```
private void CreateDataView()
{
    // Установить таблицу, которая используется для создания этого представления.
    yugosOnlyView = new DataView(inventoryTable);
    // Сконфигурировать представление с помощью фильтра.
    yugosOnlyView.RowFilter = "Make = 'Yugo'";
    // Привязать к новому элементу DataGridView.
    dataGridYugosView.DataSource = yugosOnlyView;
}
```

Как видите, класс `DataView` содержит свойство по имени `RowFilter`, содержащее строку с критерием фильтрации, который используется для извлечения искомых строк. После создания представления измените соответствующим образом свойство `DataSource` в `DataGridView`. Завершенное приложение в действии показано на рис. 22.10.

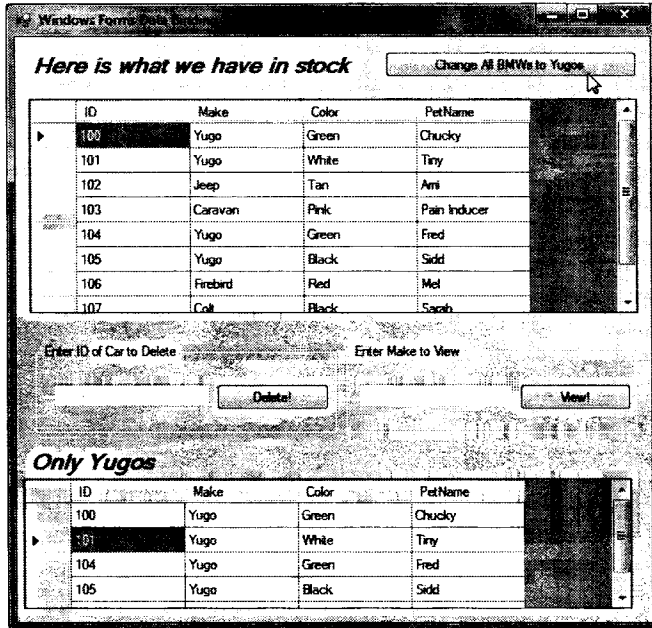


Рис. 22.10. Отображение уникального представления данных

Исходный код. Проект `WindowsFormsDataBinding` доступен в подкаталоге `Chapter 22`.

Работа с адаптерами данных

Теперь, когда вы понимаете нюансы ручного манипулирования объектами `DataSet` в ADO.NET, самое время взглянуть, что собой представляют *объекты адаптеров данных*. Адаптер данных — это класс, применяемый для заполнения `DataSet` объектами `DataTable`; этот класс также может отправлять измененные объекты `DataTable` обратно базе данных для обработки. В табл. 22.8 перечислены основные члены базового класса `DbDataAdapter`, который является общим родительским классом для всех объектов адаптеров данных (например, `SqlDataAdapter` и `OdbcDataAdapter`).

Таблица 22.8. Основные члены класса `DbDataAdapter`

Член	Описание
<code>Fill()</code>	Выполняет SQL-команду <code>SELECT</code> (указанную в свойстве <code>SelectCommand</code>) для запроса у базы данных и загрузки полученных данных в объект <code>DataTable</code>
<code>SelectCommand</code> <code>InsertCommand</code> <code>UpdateCommand</code> <code>DeleteCommand</code>	Устанавливают SQL-команды, отправляемые в хранилище данных, когда вызываются методы <code>Fill()</code> и <code>Update()</code>
<code>Update()</code>	Выполняет SQL-команды <code>INSERT</code> , <code>UPDATE</code> и <code>DELETE</code> (указанные в свойствах <code>InsertCommand</code> , <code>UpdateCommand</code> и <code>DeleteCommand</code>) для сохранения в базе данных изменений, произведенных в <code>DataTable</code>

Обратите внимание, что в адаптере данных определены четыре свойства: `SelectCommand`, `InsertCommand`, `UpdateCommand` и `DeleteCommand`. При создании объекта адаптера данных для конкретного поставщика данных (скажем, `SqlDataAdapter`) можно передать строку с текстом команды, который будет использоваться объектом команды `SelectCommand`.

Предполагая, что каждый из четырех объектов команд должным образом сконфигурирован, можно затем вызвать метод `Fill()` для получения объекта `DataSet` (или, при желании, отдельного объекта `DataTable`). Чтобы сделать это, адаптер данных должен выполнить SQL-оператор `SELECT`, указанный в свойстве `SelectCommand`.

Аналогично, если необходимо сохранить модифицированный объект `DataSet` (или `DataTable`) в базе данных, можно вызвать метод `Update()`, который будет использовать какой-то из оставшихся объектов команд в зависимости от состояния каждой строки в `DataTable` (более подробно это будет рассматриваться ниже).

Один из самых странных аспектов работы с объектом адаптера данных состоит в том, что при этом не нужно открывать или закрывать подключение к базе данных. Вместо этого управление подключением к базе данных осуществляется автоматически. Тем не менее, адаптеру данных все равно необходимо передать объект подключения или строку соединения (на основе которой внутренне создается объект подключения), чтобы сообщить, с какой базой данных взаимодействовать.

На заметку! Адаптер данных независим по своей природе. К нему можно присоединять на лету разные объекты подключения и объекты команд и выбирать данные из широкого разнообразия баз данных. Например, одиночный объект `DataSet` может содержать табличные данные, полученные от поставщиков данных `SQL Server`, `Oracle` и `MySQL`.

Простой пример адаптера данных

Следующий шаг заключается в добавлении новой функциональности в сборку библиотеки доступа к данным (`AutoLotDAL.dll`), которая была создана в главе 21. Начнем с рассмотрения простого примера, в котором объект `DataSet` заполняется одной таблицей с помощью объекта адаптера данных `ADO.NET`.

Создайте новое консольное приложение по имени `FillDataSetUsingSqlDataAdapter` и импортируйте в первоначальный файл кода C# пространства имен `System.Data` и `System.Data.SqlClient`. Теперь модифицируйте метод `Main()` следующим образом (в зависимости от того, как создавалась база данных `AutoLot` в главе 21, может понадобиться изменить строку соединения):

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Data Adapters *****\n");
    // Жестко закодированная строка соединения.
    string cnStr = "Integrated Security = SSPI;Initial Catalog=AutoLot;" +
        @"Data Source=(local)\SQLEXPRESS";
    // Объект DataSet создается вызывающим процессом.
    DataSet ds = new DataSet("AutoLot");
    // Указать адаптеру текст команды Select и строку соединения.
    SqlDataAdapter dAdapt =
        new SqlDataAdapter("Select * From Inventory", cnStr);
    // Заполнить DataSet новой таблицей по имени Inventory.
    dAdapt.Fill(ds, "Inventory");
    // Отобразить содержимое DataSet с использованием вспомогательного
    // метода, созданного ранее в данной главе.
    PrintDataSet(ds);
    Console.ReadLine();
}
```

Обратите внимание, что адаптер данных конструируется с указанием строкового литерала, который будет отображен на SQL-оператор SELECT. Это значение будет использоваться для внутреннего построения объекта команды, который можно получить позже с помощью свойства `SelectCommand`.

Кроме того, следует отметить, что создание экземпляра класса `DataSet`, который затем передается методу `Fill()`, является заботой вызывающего процесса. Дополнительно этому методу можно передать в качестве второго аргумента строковое имя, которое будет применяться для установки свойства `TableName` нового объекта `DataTable` (если не указывать имя таблицы, то адаптер данных назовет таблицу просто `Table`). Обычно имя, назначенное `DataTable`, будет идентичным имени физической таблицы в реляционной базе данных; тем не менее, это не обязательно.

На заметку! Метод `Fill()` возвращает целое число, которое представляет количество строк, возвращенных SQL-запросом.

И, наконец, обратите внимание, что в методе `Main()` нигде нет явного открытия или закрытия подключения к базе данных. В методе `Fill()` любого адаптера данных с самого начала заложена возможность открытия и закрытия подключения перед вызовом метода `Fill()`. Следовательно, при передаче объекта `DataSet` методу `PrintDataSet()`, реализованному ранее в этой главе, вы оперируете с локальной копией автономных данных, которой не нужны обращения к СУБД для выборки данных.

Отображение имен из базы данных на дружелюбные к пользователю имена

Как упоминалось ранее, администраторы баз данных обычно создают такие имена таблиц и столбцов, которые редко бывают удобными для конечных пользователей (например, `au_id`, `au_fname`, `au_lname` и т.п.). Тем не менее, чтобы решить эту проблему, объекты адаптеров данных поддерживают внутреннюю строго типизированную коллекцию `DataTableMappingCollection` объектов `System.Data.Common.DataTableMapping`, доступ к которой осуществляется через свойство `TableMappings` объекта адаптера данных.

При желании с помощью этой коллекции можно информировать объект `DataTable` о том, какие *отображаемые имена* должны использоваться при выводе его содержимого. Предположим, например, что во время вывода вместо имени таблицы `Inventory` необходимо отображать `Current Inventory`, вместо имени столбца `CarID` — заголовок `Car ID` (т.е. с пробелом), а вместо имени столбца `PetName` — строку `Name of Car`. Для этого добавьте перед вызовом метода `Fill()` объекта адаптера данных показанный ниже код (не забудьте импортировать пространство имен `System.Data.Common`, чтобы было доступным определение типа `DataTableMapping`):

```
static void Main(string[] args)
{
    ...
    // Отобразить имена столбцов базы данных на дружелюбные к пользователю имена.
    DataTableMapping custMap =
        dAdapt.TableMappings.Add("Inventory", "Current Inventory");
    custMap.ColumnMappings.Add("CarID", "Car ID");
    custMap.ColumnMappings.Add("PetName", "Name of Car");
    dAdapt.Fill(ds, "Inventory");
    ...
}
```

Еще раз запустив эту программу, вы увидите, что метод `PrintDataSet()` теперь отображает дружелюбные имена объектов `DataTable` и `DataRow`, а не имена из схемы базы данных:

```
***** Fun with Data Adapters *****
DataSet is named: AutoLot
=> Current Inventory Table:
Car ID Make    Color    Name of Car
-----
83      Ford     Rust     Rusty
107     Ford     Red      Snake
678     Yugo     Green    Clunker
904     VW       Black    Hank
1000    BMW      Black    Bimmer
1001    BMW      Tan      Daisy
1992    Saab     Pink     Pinkey
2003    Yugo     Rust     Mel
```

Исходный код. Проект `FillDataSetUsingSqlDataAdapter` доступен в подкаталоге `Chapter 22`.

Добавление в `AutoLotDAL.dll` функциональности автономного уровня

Чтобы продемонстрировать применение адаптера данных для передачи изменений из `DataTable` в базу данных, мы модифицируем созданную в главе 21 сборку `AutoLotDAL.dll` для включения нового пространства имен (под названием `AutoLotDisconnectedLayer`). Это пространство имен будет содержать новый класс `InventoryDALDisLayer`, который использует адаптер данных для взаимодействия с объектом `DataTable`.

Для начала имеет смысл скопировать всю папку проекта `AutoLotDAL`, который был создан в главе 21, в новое место на жестком диске и переименовать ее в `AutoLotDAL (Version 2)`. Теперь запустите Visual Studio, выберите пункт меню `File⇒Open Project/Solution...` (Файл⇒Открыть проект/решение...) и откройте файл `AutoLotDAL.sln` в папке `AutoLotDAL (Version 2)`.

Определение начального класса

С помощью пункта меню `Project⇒Add Class` (Проект⇒Добавить класс) вставьте новый класс по имени `InventoryDALDisLayer`; он должен быть открытым (`public`). Измените название пространства имен, в котором упакован этот класс, на `AutoLotDisconnectedLayer` и импортируйте пространства имен `System.Data` и `System.Data.SqlClient`.

В отличие от типа `InventoryDAL`, ориентированного на работу с подключением, новому классу не нужны специальные методы открытия и закрытия, т.к. адаптер данных обрабатывает все детали автоматически.

Начните с добавления специального конструктора, который устанавливает закрытую переменную `string`, представляющую строку соединения. Кроме того, определите закрытую переменную-член `SqlDataAdapter`, которая будет конфигурироваться вызовом (пока еще не созданного) вспомогательного метода `ConfigureAdapter()`, принимающего выходной параметр `SqlDataAdapter`:

```

namespace AutoLotDisconnectedLayer
{
    public class InventoryDALDisLayer
    {
        // Поля данных.
        private string cnString = string.Empty;
        private SqlDataAdapter dAdapt = null;

        public InventoryDALDisLayer(string connectionString)
        {
            cnString = connectionString;
            // Конфигурировать SqlDataAdapter.
            ConfigureAdapter(out dAdapt);
        }
    }
}

```

Конфигурирование адаптера данных с использованием SqlCommandBuilder

При использовании адаптера данных для модификации таблиц в DataSet сначала необходимо присвоить свойствам UpdateCommand, DeleteCommand и InsertCommand допустимые объекты команд (до тех пор эти свойства будут возвращать ссылки null).

Ручное конфигурирование объектов команд для свойств InsertCommand, UpdateCommand и DeleteCommand может потребовать значительный объем кода, особенно если применяются параметризованные запросы. Вспомните из главы 21, что параметризованные запросы позволяют строить SQL-операторы с использованием набора объектов параметров. Таким образом, если избран длинный путь, то можно было бы реализовать метод ConfigureAdapter() для создания вручную трех новых объектов SqlCommand, каждый из которых содержал бы набор объектов SqlParameter. После этого полученные объекты можно было бы присвоить свойствам UpdateCommand, DeleteCommand и InsertCommand адаптера.

В Visual Studio имеется несколько визуальных конструкторов, которые берут на себя хлопоты по созданию этого утомительного кода. Визуальные конструкторы немного отличаются в зависимости от используемого API-интерфейса (например, Windows Forms, WPF или ASP.NET), но их общая функциональность аналогична. Примеры применения этих визуальных конструкторов будут неоднократно встречаться в настоящей книге, включая использование ряда визуальных конструкторов Windows Forms далее в главе.

В этот момент не понадобится писать многочисленные операторы кода для полного конфигурирования адаптера данных; вместо этого можно реализовать метод ConfigureAdapter(), как показано ниже:

```

private void ConfigureAdapter(out SqlDataAdapter dAdapt)
{
    // Создать адаптер и настроить SelectCommand.
    dAdapt = new SqlDataAdapter("Select * From Inventory", cnString);
    // Динамически получить остальные объекты команд
    // во время выполнения, используя SqlCommandBuilder.
    SqlCommandBuilder builder = new SqlCommandBuilder(dAdapt);
}

```

Для упрощения создания объектов адаптеров данных каждый поставщик данных ADO.NET, разработанный Microsoft, предоставляет тип *построителя команд* — SqlCommandBuilder. Этот тип автоматически генерирует значения для свойств InsertCommand, UpdateCommand и DeleteCommand объекта SqlDataAdapter на основе

начального объекта `SelectCommand`. Преимущество заключается в том, что не требуется вручную создавать все типы `SqlCommand` и `SqlParameter`.

В этот момент возникает очевидный вопрос: почему построитель команд имеет возможность создания этих объектов команд на лету? Краткий ответ на этот вопрос: благодаря метаданным. Когда во время выполнения вызывается метод `Update()` адаптера данных, соответствующий построитель команд читает информацию схемы базы данных и автоматически генерирует необходимые объекты команд вставки, удаления и обновления.

Очевидно, что это требует дополнительных обращений к удаленной базе данных, а при неоднократном использовании `SqlCommandBuilder` в одном приложении производительность ухудшится. Здесь мы минимизируем негативный эффект с помощью вызова метода `ConfigureAdapter()` во время конструирования объекта `InventoryDALDisLayer`, сохранив сконфигурированный объект `SqlDataAdapter` для использования на протяжении всего времени жизни `InventoryDALDisLayer`.

В приведенном ранее коде объект построителя команд (`SqlCommandBuilder`) не использовался за исключением того, что его конструктору был передан в качестве параметра объект адаптера данных. Как бы странно это не выглядело, это все, что должно быть сделано (в минимальном варианте). “За кулисами” этот тип конфигурирует адаптер данных с применением остальных объектов команд.

Тем не менее, следует учесть, что построители команд обладают рядом серьезных ограничений. В частности, построитель команд может автоматически генерировать команды SQL для использования их адаптером данных, если выполнены все следующие условия:

- SQL-команда `SELECT` работает только с одной таблицей (т.е. никаких соединений);
- эта единственная таблица имеет первичный ключ;
- таблица должна иметь столбец (или столбцы), представляющий первичный ключ, который включается в SQL-оператор `SELECT`.

Учитывая способ построения базы данных `AutoLot`, эти ограничения не доставляют никаких проблем. Однако в более реальных базах данных следует подумать, полезен ли этот тип вообще (если нет, то примите во внимание, что `Visual Studio` автоматически генерирует большую часть необходимого кода — это будет показано далее в этой главе).

Реализация метода `GetAllInventory()`

Теперь наш адаптер данных готов к применению. Первый метод нового класса будет просто вызывать метод `Fill()` объекта `SqlDataAdapter` для получения объекта `DataTable`, представляющего все записи в таблице `Inventory` базы данных `AutoLot`:

```
public DataTable GetAllInventory()
{
    DataTable inv = new DataTable("Inventory");
    dAdapt.Fill(inv);
    return inv;
}
```

Реализация метода `UpdateInventory()`

Как показано ниже, метод `UpdateInventory()` очень прост:

```
public void UpdateInventory(DataTable modifiedTable)
{
    dAdapt.Update(modifiedTable);
}
```

Здесь объект адаптера данных проверяет значение `RowState` у каждой строки входной таблицы `DataTable`. В зависимости от его значения (т.е. `RowState.Added`, `RowState.Deleted` или `RowState.Modified`) автоматически используется нужный объект команды.

Установка номера версии

К этому моменту построение второй версии библиотеки доступа к данным завершено. Хотя это и необязательно, все же с целью учета установите для этой библиотеки номер версии 2.0.0.0. Как было описано в главе 14, чтобы изменить версию сборки .NET, щелкните на значке **Properties** (Свойства) в окне **Solution Explorer**, а затем на кнопке **Assembly Information** (Информация о сборке), которая находится на вкладке **Application** (Приложение). В открывшемся диалоговом окне укажите 2 для старшего номера версии сборки (см. главу 14). После этого перекомпилируйте приложение, чтобы обновить манифест сборки.

Исходный код. Проект `AutoLotDAL` (Version 2) доступен в подкаталоге `Chapter 22`.

Тестирование функциональности автономного уровня

Теперь есть все для создания клиентского приложения, которое позволит проверить работу нового класса `InventoryDALDisLayer`. Здесь мы опять воспользуемся API-интерфейсом **Windows Forms** для отображения данных в графическом пользовательском интерфейсе. Создайте новое приложение **Windows Forms** по имени `InventoryDALDisconnectedGUI` и измените в **Solution Explorer** первоначальное имя файла `Form1.cs` на `MainForm.cs`. После создания проекта установите ссылку на обновленную сборку `AutoLotDAL.dll` (удостоверьтесь, что выбрана версия 2.0.0.0 сборки) и импортируйте следующее пространство имен:

```
using AutoLotDisconnectedLayer;
```

Главная форма приложения содержит элементы управления `Label`, `DataGridView` (по имени `inventoryGrid`) и `Button` (по имени `btnUpdateInventory`), при этом для кнопки должен быть создан обработчик события `Click`. Ниже приведено определение формы:

```
public partial class MainForm : Form
{
    InventoryDALDisLayer dal = null;

    public MainForm()
    {
        InitializeComponent();

        string cnStr =
            @"Data Source=(local)\SQLEXPRESS;Initial Catalog=AutoLot;" +
            "Integrated Security=True;Pooling=False";

        // Создать объект доступа к данным.
        dal = new InventoryDALDisLayer(cnStr);

        // Заполнить сетку.
        inventoryGrid.DataSource = dal.GetAllInventory();
    }

    private void btnUpdateInventory_Click(object sender, EventArgs e)
    {
        // Получить модифицированные данные из сетки.
        DataTable changedDT = (DataTable)inventoryGrid.DataSource;
```



```

try
{
    // Зафиксировать изменения.
    dal.UpdateInventory(changedDT);
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
}
}
}

```

После создания объекта `InventoryDALDisLayer` можно привязать объект `DataTable`, возвращенный вызовом `GetAllInventory()`, к объекту `DataGridView`. Когда пользователь щелкнет на кнопке `Update Database` (Обновить базу данных), из сетки извлекается модифицированный объект `DataTable` (с помощью свойства `DataSource`) и передается в метод `UpdateInventory()`.

Вот и все! Запустите приложение, добавьте на сетке несколько новых строк и измените и/или удалите ряд других. После щелчка на кнопке `Update Database` все изменения будут сохранены в базе данных `AutoLot`.

Исходный код. Проект `InventoryDALDisconnectedGUI` доступен в подкаталоге `Chapter 22`.

Объекты `DataSet` с несколькими таблицами и отношения между данными

До сих пор все рассмотренные в главе примеры оперировали с единственным объектом `DataTable`. Тем не менее, вся мощь автономного уровня проявляется тогда, когда объект `DataSet` содержит несколько взаимосвязанных объектов `DataTable`. В этом случае в коллекции `DataRelation` объекта `DataSet` может быть определено любое количество объектов `DataRelation`, которые описывают все взаимозависимости между таблицами. На клиентском уровне с помощью этих объектов можно осуществлять навигацию между данными таблиц, не обращаясь к сети или СУБД.

На заметку! Вместо изменения сборки `AutoLotDAL.dll` для учета таблиц `Customers` и `Orders` в этом примере вся логика доступа к данным изолируется в новом проекте `Windows Forms`. Разумеется, в производственном приложении смешивание пользовательского интерфейса и логики обработки данных не рекомендуется. В последних примерах этой главы применяются различные инструменты проектирования баз данных, которые позволяют отделить код пользовательского интерфейса от логики обработки данных.

Начните этот пример с создания нового приложения `Windows Forms` по имени `MultitabledDataSetApp`. Графический пользовательский интерфейс приложения максимально прост (обратите внимание, что первоначальное имя файла `Form1.cs` изменено на `MainForm.cs`). На рис. 22.11 показаны три элемента `DataGridView` (`dataGridViewInventory`, `dataGridViewCustomers` и `dataGridViewOrders`), которые содержат данные, извлеченные из таблиц `Inventory`, `Orders` и `Customers` базы данных `AutoLot`. Кроме того, в интерфейсе предусмотрен элемент `Button` (по имени `btnUpdateDatabase`), позволяющий отправить все изменения, которые были внесены в сетках, в базу данных для обработки с применением объектов адаптеров данных.

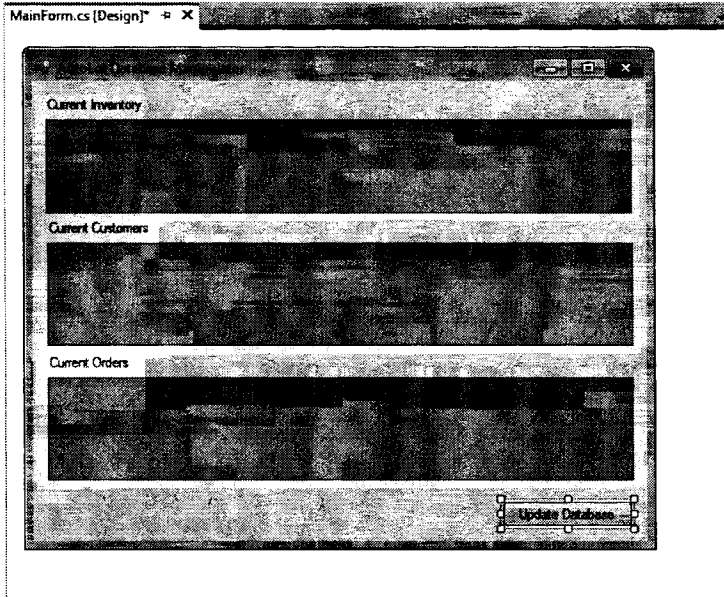


Рис. 22.11. Первоначальный пользовательский интерфейс будет отображать содержимое всех таблиц базы данных AutoLot

Подготовка адаптеров данных

Для максимально возможного упрощения кода доступа к данным в классе MainForm будут использоваться объекты построителей команд, которые автоматически сгенерируют команды SQL для всех трех объектов SqlDataAdapter (по одному на каждую таблицу). Ниже представлена первая версия типа, производного от Form (не забудьте импортировать пространство имен System.Data.SqlClient):

```
public partial class MainForm : Form
{
    // Объект DataSet уровня формы.
    private DataSet autoLotDS = new DataSet("AutoLot");

    // Использовать построители команд для упрощения конфигурирования адаптеров данных
    private SqlCommandBuilder sqlCBIInventory;
    private SqlCommandBuilder sqlCBCustomers;
    private SqlCommandBuilder sqlCBOOrders;

    // Адаптеры данных (для каждой таблицы).
    private SqlDataAdapter invTableAdapter;
    private SqlDataAdapter custTableAdapter;
    private SqlDataAdapter ordersTableAdapter;

    // Строка соединения уровня формы.
    private string cnStr = string.Empty;
    ...
}
```

Конструктор выполняет всю черновую работу по созданию переменных-членов, относящихся к данным, и заполнению DataSet. В этом примере предполагается, что вы создали файл App.config, содержащий подходящую строку соединения (а также добавили ссылку на сборку System.Configuration.dll и импортировали пространство имен System.Configuration), как показано ниже:

```

<configuration>
  <connectionStrings>
    <add name="AutoLotSqlProvider" connectionString =
      "Data Source=(local)\SQLEXPRESS;
      Integrated Security=SSPI;Initial Catalog=AutoLot" />
  </connectionStrings>
</configuration>

```

Кроме того, обратите внимание на добавление вызова закрытого вспомогательного метода `BuildTableRelationship()`:

```

public MainForm()
{
  InitializeComponent();

  // Получить строку соединения из файла *.config.
  cnStr = ConfigurationManager.ConnectionStrings[
    "AutoLotSqlProvider"].ConnectionString;

  // Создать адаптеры.
  invTableAdapter = new SqlDataAdapter("Select * from Inventory", cnStr);
  custTableAdapter = new SqlDataAdapter("Select * from Customers", cnStr);
  ordersTableAdapter = new SqlDataAdapter("Select * from Orders", cnStr);

  // Автоматически сгенерировать команды.
  sqlCBInventory = new SqlCommandBuilder(invTableAdapter);
  sqlCBOrders = new SqlCommandBuilder(ordersTableAdapter);
  sqlCBCustomers = new SqlCommandBuilder(custTableAdapter);

  // Заполнить таблицы в DataSet.
  invTableAdapter.Fill(autoLotDS, "Inventory");
  custTableAdapter.Fill(autoLotDS, "Customers");
  ordersTableAdapter.Fill(autoLotDS, "Orders");

  // Построить отношения между таблицами.
  BuildTableRelationship();

  // Привязать к сеткам.
  dataGridViewInventory.DataSource = autoLotDS.Tables["Inventory"];
  dataGridViewCustomers.DataSource = autoLotDS.Tables["Customers"];
  dataGridViewOrders.DataSource = autoLotDS.Tables["Orders"];
}

```

Построение отношений между таблицами

Вспомогательный метод `BuildTableRelationship()` выполняет всю рутинную работу по добавлению в `autoLotDS` двух объектов `DataRelation`. Вспомните из главы 21, что в базе данных `AutoLot` имеется несколько отношений "родительский-дочерний", и они учтены в следующем коде:

```

private void BuildTableRelationship()
{
  // Создать объект отношения между данными CustomerOrder.
  DataRelation dr = new DataRelation("CustomerOrder",
    autoLotDS.Tables["Customers"].Columns["CustID"],
    autoLotDS.Tables["Orders"].Columns["CustID"]);
  autoLotDS.Relations.Add(dr);

  // Создание объекта отношения между данными InventoryOrder.
  dr = new DataRelation("InventoryOrder",
    autoLotDS.Tables["Inventory"].Columns["CarID"],
    autoLotDS.Tables["Orders"].Columns["CarID"]);
  autoLotDS.Relations.Add(dr);
}

```

Обратите внимание, что при создании объекта `DataRelation` в первом параметре указывается более понятный строковый псевдоним (вы вскоре увидите, почему это удобно). Кроме того, устанавливаются ключи, используемые для построения самого отношения. В коде видно, что сначала указывается родительская таблица (второй параметр конструктора), а затем дочерняя (третий параметр конструктора).

Обновление таблиц базы данных

Теперь, когда объект `DataSet` заполнен информацией из источника данных, каждым объектом `DataTable` можно манипулировать локально. Для этого запустите приложение и вставляйте, обновляйте или удаляйте значения в любом из трех элементов `DataGridView`. По готовности отправки данных в базу на обработку, щелкните на кнопке **Update Database** (Обновить базу данных). Теперь уже должен быть понятен код обработки события `Click`:

```
private void btnUpdateDatabase_Click(object sender, EventArgs e)
{
    try
    {
        invTableAdapter.Update(autoLotDS, "Inventory");
        custTableAdapter.Update(autoLotDS, "Customers");
        ordersTableAdapter.Update(autoLotDS, "Orders");
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

Запустите приложение и выполните различные обновления данных. При следующем запуске приложения вы увидите, что сетки содержат все последние изменения.

Навигация между связанными таблицами

Теперь давайте посмотрим, как `DataRelation` позволяет программно перемещаться между связанными таблицами. Добавьте в графический интерфейс новый элемент `Button` (по имени `btnGetOrderInfo`), связанный с ним `TextBox` (`txtCustID`) и `Label` с подходящим текстом (для улучшения внешнего вида эти элементы можно сгруппировать внутри `GroupBox`). На рис. 22.12 показан один из возможных вариантов графического интерфейса разрабатываемого приложения.

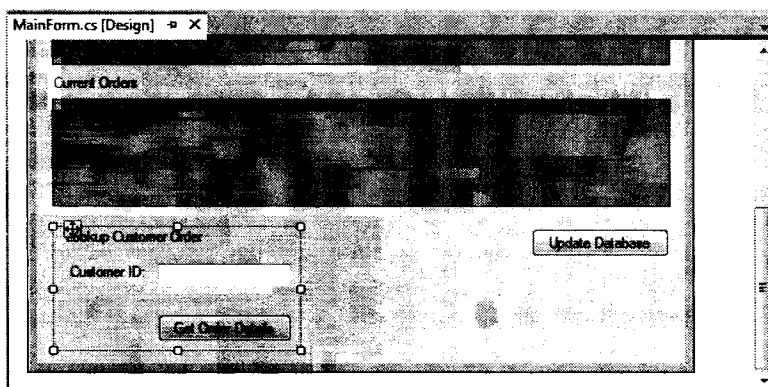


Рис. 22.12. Измененный интерфейс позволяет пользователю выполнять поиск информации о заказах клиента

Этот измененный интерфейс позволяет пользователю ввести идентификатор клиента и получить всю информацию о заказе этого клиента (имя, номер заказа, автомобиль). Эта информация форматируется в виде строки string, а затем выводится в окне сообщения. Ниже показан код обработчика события Click только что добавленной кнопки:

```
private void btnGetOrderInfo_Click(object sender, System.EventArgs e)
{
    string strOrderInfo = string.Empty;
    DataRow[] drsCust = null;
    DataRow[] drsOrder = null;

    // Получить идентификатор клиента из текстового поля.
    int custID = int.Parse(this.txtCustID.Text);

    // На основе custID получить подходящую строку из таблицы Customers.
    drsCust = autoLotDS.Tables["Customers"].Select(
        string.Format("CustID = {0}", custID));
    strOrderInfo += string.Format("Customer {0}: {1} {2}\n",
        drsCust[0]["CustID"].ToString(),
        drsCust[0]["FirstName"].ToString(),
        drsCust[0]["LastName"].ToString());

    // Перейти из таблицы Customers в таблицу Orders.
    drsOrder = drsCust[0].GetChildRows(autoLotDS.Relations["CustomerOrder"]);

    // Проход в цикле по всем заказам этого клиента.
    foreach (DataRow order in drsOrder)
    {
        strOrderInfo += string.Format("----\nOrder Number:
            {0}\n", order["OrderID"]);

        // Получить автомобиль, на который ссылается этот заказ.
        DataRow[] drsInv = order.GetParentRows(autoLotDS.Relations[
            "InventoryOrder"]);

        // Получить информацию для (ОДНОГО) автомобиля из этого заказа.
        DataRow car = drsInv[0];
        strOrderInfo += string.Format("Make: {0}\n", car["Make"]);
        // Марка
        strOrderInfo += string.Format("Color: {0}\n", car["Color"]);
        // Цвет
        strOrderInfo += string.Format("Pet Name: {0}\n", car["PetName"]);
        // Дружественное имя
    }
    MessageBox.Show(strOrderInfo, "Order Details");
}
```

На рис. 22.13 показан один из возможных результатов работы, когда задан идентификатор клиента 3 (ваш результат может отличаться в зависимости от содержимого таблиц базы данных AutoLot).

Этот последний пример должен был убедить вас в полезности класса DataSet. Учитывая, что объект DataSet полностью отключен от лежащего в основе источника данных, можно работать с находящейся в памяти копией данных и переходить от таблицы к таблице для внесения необходимых обновлений, удалений или вставок, не обращаясь к базе данных. После этого можно отправить измененные данные обратно в хранилище. В результате получается масштабируемое и надежное приложение.

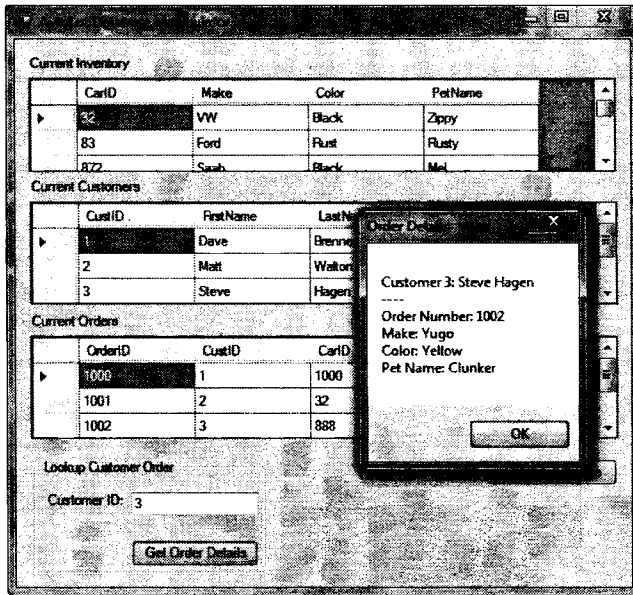


Рис. 22.13. Навигация с помощью отношений между данными

Инструменты визуального конструирования баз данных Windows Forms

Во всех приведенных до сих пор примеров делался ощутимый объем черновой работы в том смысле, что вся логика доступа к данным была написана вручную. Хотя значительная часть этого кода была вынесена в библиотеку .NET (AutoLotDAL.dll) для повторного использования в последующих главах книги, все равно приходится вручную создавать различные объекты поставщика данных перед тем, как можно будет взаимодействовать с реляционной базой данных. Следующая задача заключается в исследовании инструментов визуального конструирования баз данных Windows Forms, которые могут создать за вас значительный объем кода доступа к данным.

На заметку! При создании проектов Windows Presentation Foundation и ASP.NET доступны похожие инструменты визуального конструирования баз данных; с некоторыми из них вы ознакомитесь далее в главе.

Одним из способов применения этих интегрированных инструментов является использование визуальных конструкторов, поддерживаемых элементом управления DataGridView из Windows Forms. Проблема, связанная с этим подходом, состоит в том, что инструменты визуального конструирования баз данных вставляют весь код доступа к данным непосредственно в кодовую базу графического пользовательского интерфейса! В идеальном случае весь код, сгенерированный визуальным конструктором, лучше изолировать в выделенную библиотеку кода .NET, чтобы можно было многократно использовать логику доступа к данным в различных проектах.

Тем не менее, полезно начать с рассмотрения того, как с помощью элемента DataGridView генерировать код доступа к данным, поскольку такой подход может быть полезен в небольших проектах и прототипах приложений. После этого будет показано, каким образом изолировать этот сгенерированный визуальным конструктором код в третьей версии библиотеки AutoLotDAL.dll.

Визуальное проектирование элемента управления DataGridView

Элемент управления DataGridView имеет связанный с ним мастер, который может генерировать код доступа к данным. Для начала создайте новый проект приложения Windows Forms по имени DataGridViewDataDesigner. Переименуйте с помощью Solution Explorer первоначальную форму в MainForm.cs и добавьте на нее экземпляр элемента управления DataGridView (с именем inventoryDataGridView). Справа от этого элемента управления должен открыться встроенный редактор (если он не открылся, щелкните на кнопке с изображением треугольника в правом верхнем углу элемента). В раскрывающемся списке Choose Data Source (Выберите источник данных) выберите вариант (none) (отсутствует) и щелкните на ссылке Add Project Data Source... (Добавить источник данных для проекта...), как показано на рис. 22.14.

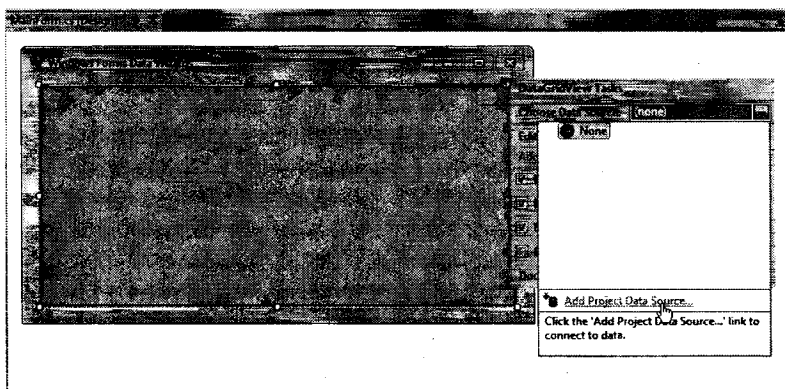


Рис. 22.14. Редактор элемента управления DataGridView

После этого запустится мастер конфигурирования источников данных (Data Source Configuration Wizard). Он проведет вас через последовательность шагов, позволяющих выбрать и настроить источник данных, который затем будет привязан к DataGridView. На первом шаге мастер запрашивает тип источника данных, с которым необходимо взаимодействовать. Выберите вариант Database (База данных), как показано на рис. 22.15, и щелкните на кнопке Next (Далее).

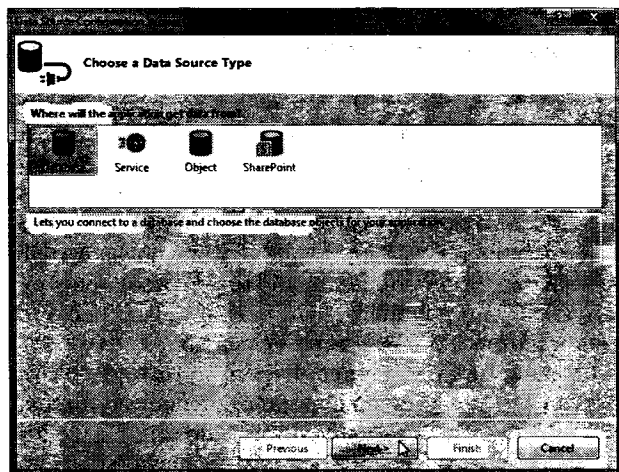


Рис. 22.15. Выбор типа источника данных

На следующем шаге (который будет слегка отличаться в зависимости от выбора, произведенного на первом шаге) мастер запрашивает, нужно ли использовать модель базы данных DataSet или модель сущностных данных (Entity Data Model). Выберите вариант DataSet (рис. 22.16), т.к. инфраструктура Entity Framework пока еще не рассматривалась (о ней речь пойдет в следующей главе).

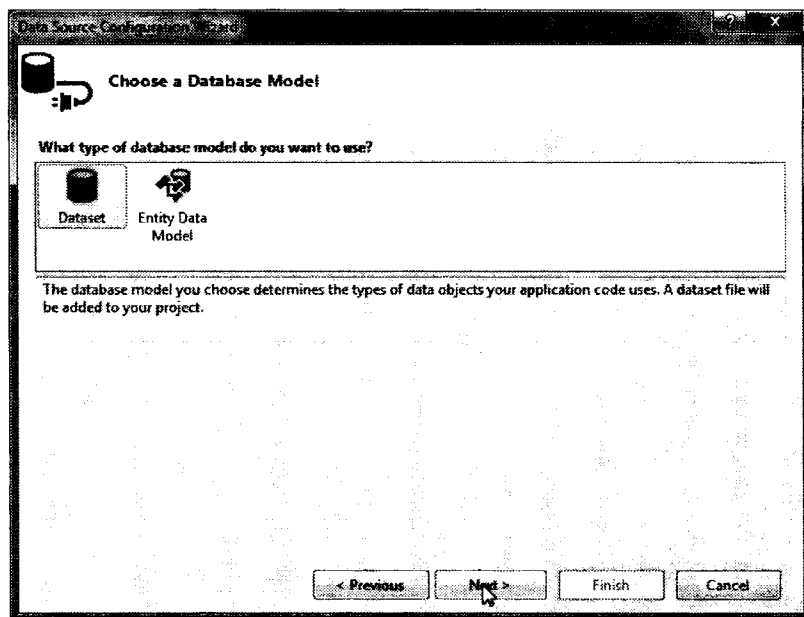


Рис. 22.16. Выбор модели базы данных

Следующий шаг мастера позволяет настроить подключение к базе данных. Если эта база данных уже добавлена в Server Explorer, то она будет присутствовать в раскрывающемся списке. Если же это так (или нужно подключиться к базе данных, ранее не добавленной в Server Explorer), щелкните на кнопке New Connection... (Новое подключение...). Результат выбора локального экземпляра базы данных AutoLot показан на рис. 22.17.

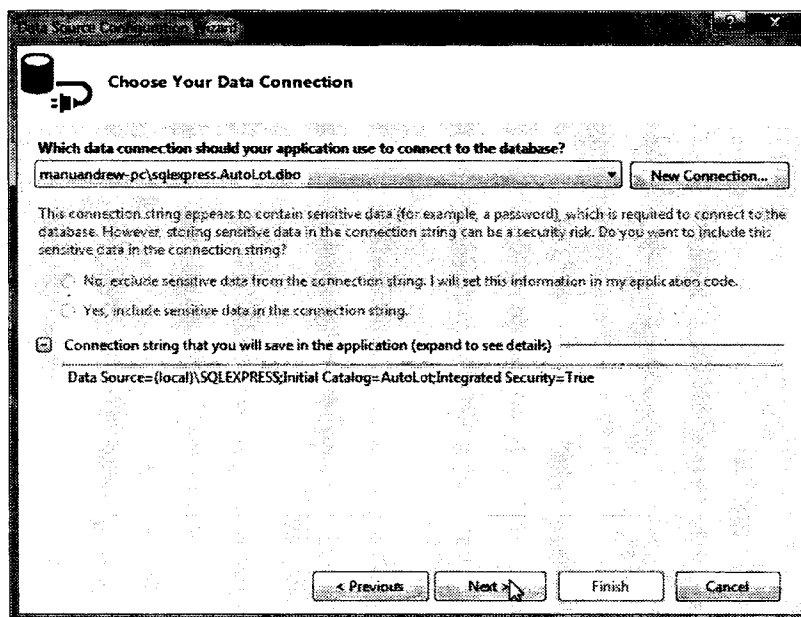


Рис. 22.17. Выбор базы данных

На следующем шаге мастера выдается запрос о том, хотите ли вы сохранить строку соединения в конфигурационном файле приложения (соответствующий снимок экрана здесь не приводится). Укажите, что нужно сохранить строку соединения, и щелкните на кнопке Next. Последний шаг мастера позволяет выбрать объекты баз данных, которые будут учтены в автоматически сгенерированном классе DataSet и связанных адаптерах данных. Хотя можно было бы выбрать все объекты данных из AutoLot, понадобится только таблица Inventory. С учетом этого, измените предлагаемое имя DataSet на InventoryDataSet (рис. 22.18), отметьте флажок возле таблицы Inventory и щелкните на кнопке Finish (Готово).

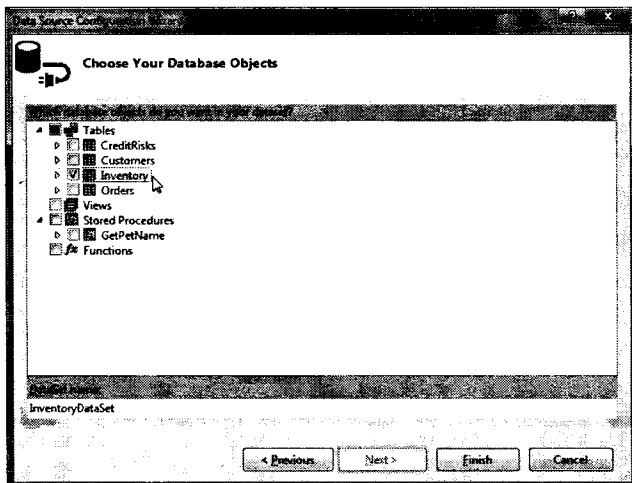


Рис. 22.18. Выбор таблицы Inventory

После этого визуальный конструктор обновится сразу во многих отношениях. Наиболее заметное изменение касается того, что в элементе DataGridView отображается схема таблицы Inventory, иллюстрируемая заголовками столбцов. Кроме того, в нижней части конструктора формы (в области, называемой *лотком с компонентами*) присутствуют три компонента: DataSet, BindingSource и TableAdapter (рис. 22.19).

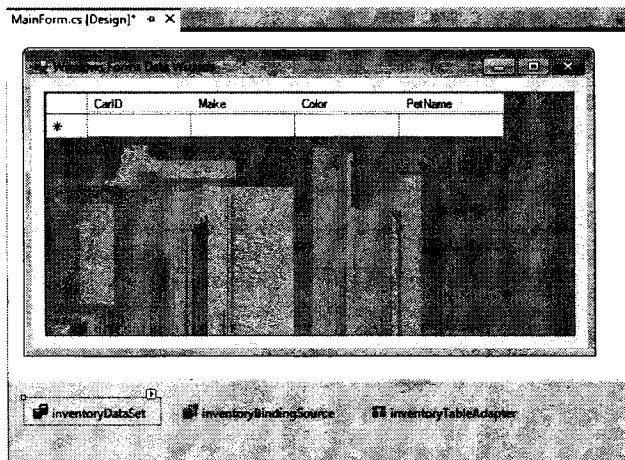


Рис. 22.19. Проект Windows Forms после выполнения мастера конфигурирования источников данных

В этот момент можно запустить приложение — и вы увидите, что сетка заполнена записями из таблицы `Inventory`. Конечно, никакой магии тут нет. IDE-среда создала за вас значительный объем кода и соответствующим образом настроила элемент управления типа сетки. Давайте рассмотрим этот автоматически сгенерированный код.

Сгенерированный файл `App.config`

Если посмотреть на проект в `Solution Explorer`, легко заметить, что он уже содержит файл `App.config`, в котором имеется элемент `<connectionStrings>` с несколько необычным именем, как показано ниже:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
  </configSections>
  <connectionStrings>
    <add name="DataGridViewDataDesigner.Properties.Settings.AutoLotConnectionString"
      connectionString=
        "Data Source=(local)\SQLEXPRESS;
        Initial Catalog=AutoLot;Integrated Security=True"
        providerName="System.Data.SqlClient" />
  </connectionStrings>
</configuration>
```

Автоматически сгенерированный объект адаптера данных (о котором будет рассказано ниже) использует длинное значение `"DataGridViewDataDesigner.Properties.Settings.AutoLotConnectionString"`.

Исследование строго типизированного класса `DataSet`

В дополнение к конфигурационному файлу мастер генерирует так называемый *строго типизированный DataSet*. Этим термином обозначается специальный класс, который расширяет `DataSet` и открывает несколько членов, позволяющих взаимодействовать с базой данных с применением более интуитивно понятной объектной модели. Например, строго типизированные объекты `DataSet` содержат свойства, которые отображаются непосредственно на имена таблиц из базы данных. Таким образом, можно использовать свойство `Inventory` для прямого обращения к строкам и столбцам базы, не углубляясь в коллекцию таблиц через свойство `Tables`.

Если вы вставите в проект новый файл диаграммы классов (выбрав в `Solution Explorer` значок проекта и щелкнув на кнопке `View Class Diagram` (Просмотреть диаграмму классов)), то заметите, что мастер создал класс по имени `InventoryDataSet`. В этом классе определен ряд членов, наиболее важным из которых является свойство `Inventory` (рис. 22.20).

Двойной щелчок на файле `InventoryDataSet.xsd` в `Solution Explorer` приводит к загрузке визуального конструктора наборов данных `Visual Studio` (который более подробно рассматривается далее в главе). Если щелкнуть правой кнопкой мыши в любом месте этого конструктора и выбрать в контекстном меню пункт `View Code` (Просмотреть код), вы увидите следующее почти пустое определение частичного класса:

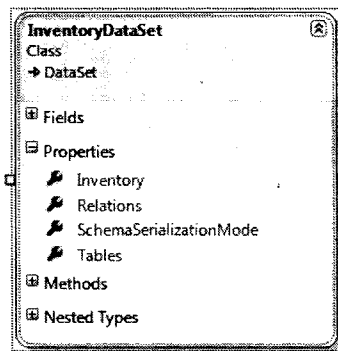


Рис. 22.20. Мастер конфигурирования источников данных создал строго типизированный `DataSet`

```
public partial class InventoryDataSet {
}
```

При необходимости в это определение частичного класса можно добавить специальные члены, однако реальное действие происходит в сопровождаемом конструктором файле `InventoryDataSet.Designer.cs`. Если открыть этот файл в `Solution Explorer`, то можно увидеть, что `InventoryDataSet` расширяет родительский класс `DataSet`. Взгляните на показанный ниже фрагмент кода с комментариями, добавленными для ясности:

```
// Весь этот код сгенерирован конструктором!
public partial class InventoryDataSet : global::System.Data.DataSet
{
    // Переменная-член типа InventoryDataTable.
    private InventoryDataTable tableInventory;

    // Каждый конструктор вызывает вспомогательный метод по имени InitClass().
    public InventoryDataSet()
    {
        ...
        this.InitClass();
        ...
    }

    // InitClass() подготавливает DataSet и добавляет
    // InventoryDataTable в коллекцию Tables.
    private void InitClass()
    {
        this.DataSetName = "InventoryDataSet";
        this.Prefix = "";
        this.Namespace = "http://tempuri.org/InventoryDataSet.xsd";
        this.EnforceConstraints = true;
        this.SchemaSerializationMode =
            global::System.Data.SchemaSerializationMode.IncludeSchema;
        this.tableInventory = new InventoryDataTable();
        base.Tables.Add(this.tableInventory);
    }

    // Свойство Inventory, предназначенное только для чтения,
    // возвращает переменную-член InventoryDataTable.
    public InventoryDataTable Inventory
    {
        get { return this.tableInventory; }
    }
}
```

Обратите внимание, что в показанном строго типизированном `DataSet` имеется переменная-член, которая является *строго типизированным* `DataTable` — в этом случае классом по имени `InventoryDataTable`. Конструктор строго типизированного класса `DataSet` вызывает закрытый метод инициализации `InitClass()`, который добавляет экземпляр этого строго типизированного `DataTable` в коллекцию `Tables` объекта `DataSet`. И еще один важный момент: реализация свойства `Inventory` возвращает переменную-член `InventoryDataTable`.

Исследование строго типизированного класса `DataTable`

Теперь вернитесь к файлу диаграммы классов и откройте узел `Nested Types` (Вложенные типы) у значка `InventoryDataSet`. Вы увидите здесь строго типизированный класс `DataTable` по имени `InventoryDataTable` и строго типизированный класс `DataRow` по имени `InventoryRow`.

В классе `InventoryDataTable` (который имеет тот же тип, что и только что рассмотренная переменная-член строго типизированного `DataSet`) определен набор свойств, основанных на именах столбцов физической таблицы `Inventory` (`CarIDColumn`, `ColorColumn`, `MakeColumn` и `PetNameColumn`), а также специальный индексатор и свойство `Count` для получения текущего количества записей.

Но более интересно то, что в этом строго типизированном классе `DataTable` определен набор методов, которые позволяют вставлять, находить и удалять строки в таблице с помощью строго типизированных членов (удобная альтернатива ручной навигации по индексаторам `Rows` и `Columns`). К примеру, метод `AddInventoryRow()` предназначен для добавления новой строки в находящуюся в памяти таблицу. `FindByCarID()` — для поиска в таблице по первичному ключу, а `RemoveInventoryRow()` позволяет удалить строку из строго типизированной таблицы (рис. 22.21).

Исследование строго типизированного класса `DataRow`

Строго типизированный класс `DataRow`, также вложенный в строго типизированный класс `DataSet`, расширяет класс `DataRow` и открывает свойства, которые отображаются напрямую на схему таблицы `Inventory`. Кроме того, мастер конфигурирования источников данных создал метод `IsPetNameNull()`, который проверяет, содержит ли этот столбец значение (рис. 22.22).

Исследование строго типизированного адаптера данных

Строгая типизация для автономных типов является серьезным преимуществом использования мастера конфигурирования источников данных, поскольку создание этих классов вручную может оказаться утомительным (хотя и вполне посильным) занятием.

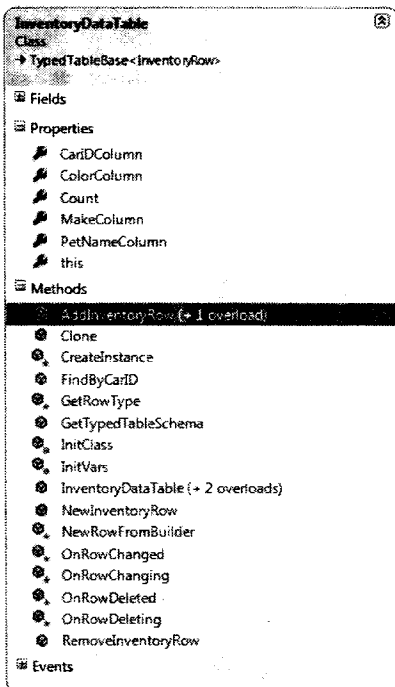


Рис. 22.21. Строго типизированный класс `DataTable`, вложенный в строго типизированный класс `DataSet`

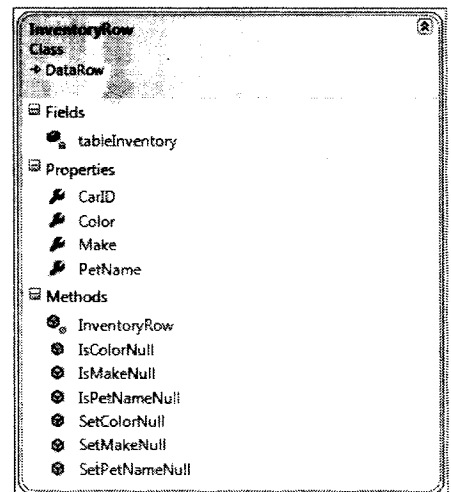


Рис. 22.22. Строго типизированный класс `DataRow`

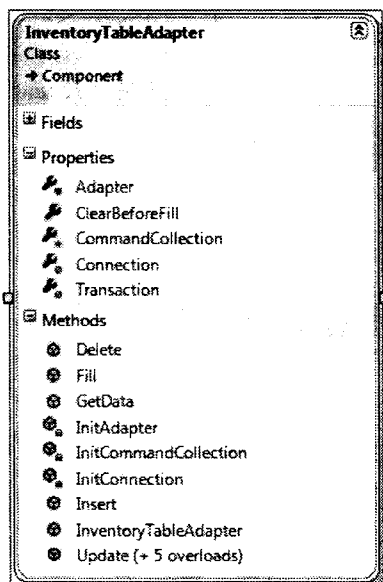


Рис. 22.23. Специализированный адаптер данных, который оперирует со строго типизированными DataSet и DataTable

Тот же самый мастер способен даже сгенерировать объект специального адаптера данных, который может строго типизированным образом заполнять и обновлять объекты InventoryDataSet и InventoryDataTable. Найдите в окне визуального конструктора классов класс InventoryTableAdapter и просмотрите сгенерированные для него члены (рис. 22.23).

Автоматически сгенерированный тип InventoryTableAdapter поддерживает коллекцию объектов SqlCommand (доступ к ним возможен с помощью свойства CommandCollection), у каждого из которых имеется полностью заполненный набор объектов SqlParameter. Кроме того, этот специальный адаптер данных предоставляет набор свойств для извлечения лежащих в основе объектов подключения, транзакции и адаптера данных, а также свойство для получения массива, представляющего все типы команд.

Завершение приложения Windows Forms

Если внимательно изучить обработчик события Load в типе, производном от формы (другими словами, если отобразить код MainForm.cs и найти метод MainForm_Load()), то можно заметить, что в самом начале вызывается метод Fill() специального адаптера данных с передачей ему специального объекта DataTable, поддерживаемого специальным DataSet:

```
private void MainForm_Load(object sender, EventArgs e)
{
    this.inventoryTableAdapter.Fill(this.inventoryDataSet.Inventory);
}
```

Этот же специальный объект адаптера данных можно применять для обновления изменений, которые были внесены в сетку. Добавьте в пользовательский интерфейс формы еще один элемент Button (по имени btnUpdateInventory). Затем создайте для него обработчик события Click со следующим кодом:

```
private void btnUpdateInventory_Click(object sender, EventArgs e)
{
    try
    {
        // Сохранить изменения, внесенные в таблицу Inventory, в базе данных.
        this.inventoryTableAdapter.Update(this.inventoryDataSet.Inventory);
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }

    // Получить актуальную копию для сетки.
    this.inventoryTableAdapter.Fill(this.inventoryDataSet.Inventory);
}
```

Снова запустите приложение; выполните добавления, удаления или обновления записей, отображаемых в сетке, а затем щелкните на кнопке Update Database (Обновить базу данных). При следующем запуске программы вы увидите, что все изменения были учтены.

Итак, рассмотренный пример продемонстрировал, насколько полезным может быть визуальный конструктор элемента управления `DataGridView`. Он позволяет работать со строго типизированными данными и генерирует за вас большую часть необходимой логики, связанной с базой данных. Очевидной проблемой является то, что полученный код тесно связан с окном, в котором он используется. В идеальном случае такая разновидность кода должна находиться в сборке `AutoLotDAL.dll` (или в другой библиотеке доступа к данным). Однако может также интересовать использование кода, сгенерированного мастером элемента `DataGridView`, в каком-нибудь проекте библиотеки классов, т.к. по умолчанию конструктор форм в нем отсутствует.

Исходный код. Проект `DataGridViewDataDesigner` доступен в подкаталоге `Chapter 22`.

Изоляция строго типизированного кода работы с базой данных в библиотеке классов

К счастью, активизировать инструменты проектирования данных Visual Studio можно в любой разновидности проекта (как с пользовательским интерфейсом, так и без), без необходимости копирования и вставки крупных фрагментов кода между проектами. Чтобы увидеть это в действии, добавим в `AutoLotDAL.dll` дополнительную функциональность.

Скопируйте всю папку `AutoLotDAL (Version 2)`, созданную ранее в этой главе, в новое место на жестком диске и переименуйте папку в `AutoLotDAL (Version 3)`. Затем выберите в Visual Studio пункт меню `File⇒Open Project/Solution...` (Файл⇒Открыть проект/решение...) и откройте файл `AutoLotDAL.sln` из новой папки `AutoLotDAL (Version 3)`.

Теперь вставьте в проект новый строго типизированный класс `DataSet` (по имени `AutoLotDataSet.xsd`) с помощью пункта меню `Project⇒Add New Item` (Проект⇒Добавить новый элемент). Чтобы быстро найти тип проекта `DataSet`, выберите в диалоговом окне `New Item` (Новый элемент) раздел `Data` (Данные), как показано на рис. 22.24.

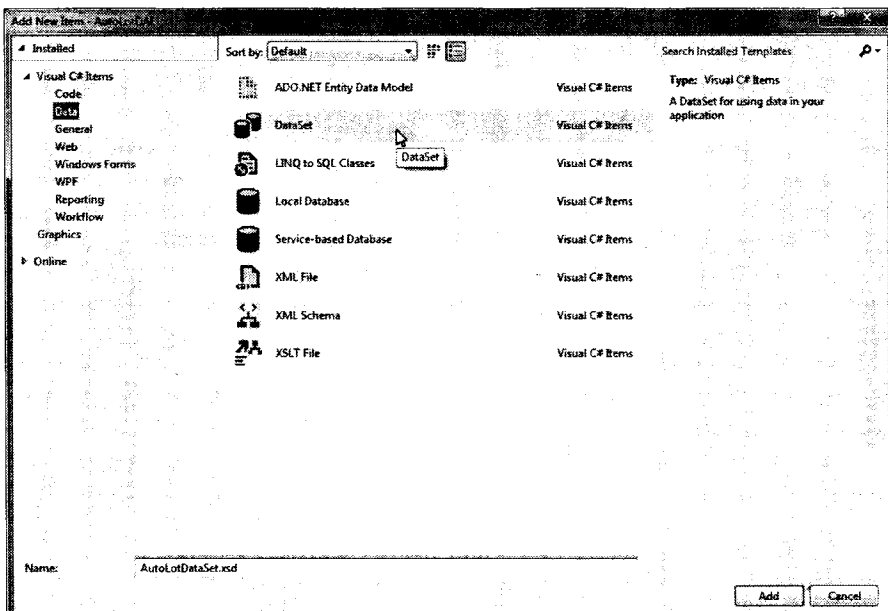


Рис. 22.24. Вставка нового строго типизированного `DataSet`

Откроется пустая поверхность визуального конструктора наборов данных. В этот момент можно воспользоваться окном Server Explorer для подключения к необходимой базе данных (подключение к AutoLot уже должно существовать) и перетащить на поверхность все таблицы и хранимые процедуры, которые должны быть сгенерированы. На рис. 22.25 видно, что все специфические аспекты базы AutoLot учтены, а отношения между ними реализованы автоматически (в этом примере таблица CreditRisk не перетаскивалась).

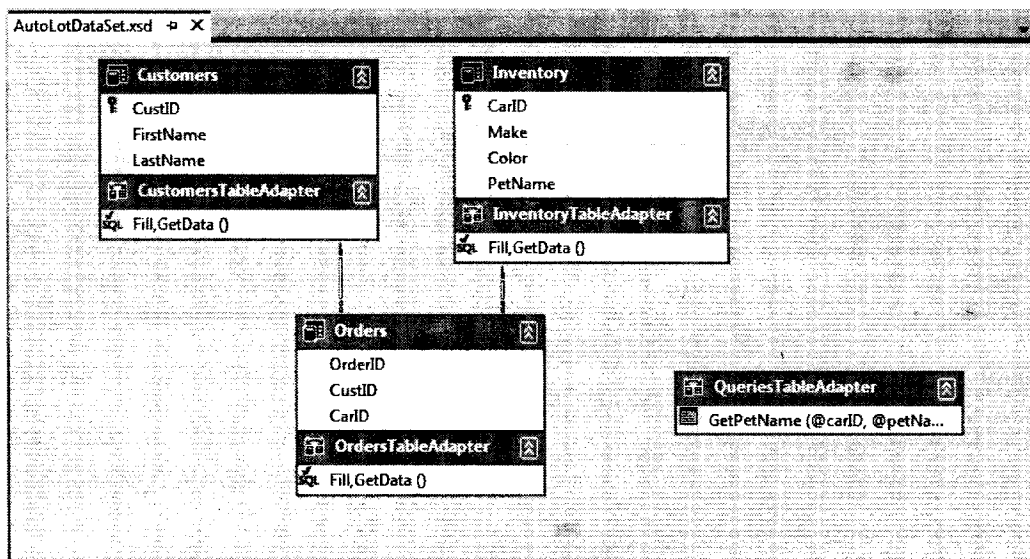


Рис. 22.25. Специальные строго типизированные классы, на этот раз находящиеся внутри проекта библиотеки классов

Просмотр сгенерированного кода

Визуальный конструктор DataSet создал в точности такой же код, как и мастер DataGridView в предыдущем примере приложения Windows Forms. Однако на этот раз задействованы таблицы Inventory, Customers и Orders, а также хранимая процедура GetPetName, поэтому сгенерированных классов получилось намного больше. По существу каждая таблица базы данных, которую вы перетаскили на поверхность визуального конструктора, дала в результате классы DataTable, DataRow и адаптера данных, содержащиеся в строго типизированном DataSet.

Строго типизированные классы DataSet, DataTable и DataRow будут помещены в корневое пространство имен проекта (AutoLotDAL). Специальные адаптеры таблиц будут находиться во вложенном пространстве имен. Просмотреть все сгенерированные типы проще всего с использованием окна Class View (Представление классов), которое открывается через меню View (Вид) среды Visual Studio (рис. 22.26).

Ради завершенности можно открыть окно Properties (Свойства) в Visual Studio (детали ищите в главе 14) и изменить версию этой последней модификации AutoLotDAL.dll на 3.0.0.0.

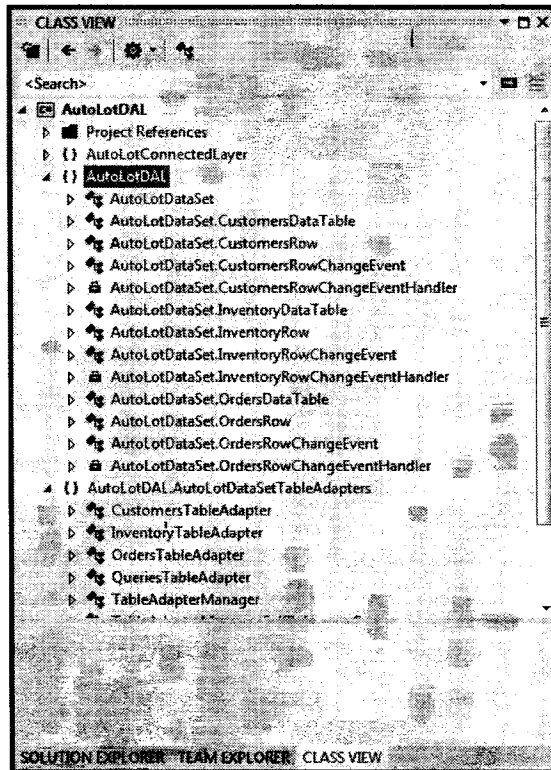


Рис. 22.26. Автоматически сгенерированные строго типизированные классы для базы данных AutoLot

Выборка данных с помощью сгенерированного кода

Теперь полученные строго типизированные классы можно использовать в любом приложении .NET, которому необходимо взаимодействовать с базой данных AutoLot. Чтобы удостовериться в понимании всех основных механизмов, создайте консольное приложение по имени `StronglyTypedDataSetConsoleClient`. Добавьте в него ссылку на последнюю версию `AutoLotDAL.dll` и импортируйте в первоначальный файл кода C# пространства имен `AutoLotDAL` и `AutoLotDAL.AutoLotDataSetTableAdapters`.

Ниже приведен код метода `Main()`, в котором объект `InventoryTableAdapter` применяется для выборки всех данных из таблицы `Inventory`. Обратите внимание, что в нем нет необходимости указывать строку соединения, т.к. эта информация теперь является частью строго типизированной объектной модели. После заполнения таблицы можно вывести результаты с помощью вспомогательного метода по имени `PrintInventory()`. При этом манипулировать строго типизированным `DataTable` можно точно так же, как “обычным” `DataTable` — используя коллекции `Rows` и `Columns`.

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with Strongly Typed DataSets *****\n");

        AutoLotDataSet.InventoryDataTable table =
            new AutoLotDataSet.InventoryDataTable();
        InventoryTableAdapter dAdapt = new InventoryTableAdapter();
```



```

dAdapt.Fill(table);
PrintInventory(table);
Console.ReadLine();
}
static void PrintInventory(AutoLotDataSet.InventoryDataTable dt)
{
    // Вывести имена столбцов.
    for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
    {
        Console.Write(dt.Columns[curCol].ColumnName + "\t");
    }
    Console.WriteLine("\n-----");
    // Вывести данные.
    for (int curRow = 0; curRow < dt.Rows.Count; curRow++)
    {
        for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
        {
            Console.Write(dt.Rows[curRow][curCol].ToString() + "\t");
        }
        Console.WriteLine();
    }
}
}

```

Вставка данных с помощью сгенерированного кода

Предположим, что теперь нужно вставить новые записи с применением этой строго типизированной объектной модели. Показанный ниже код вспомогательного метода добавляет две новых строки в текущую таблицу InventoryDataTable, после чего обновляет содержимое базы данных с помощью адаптера данных. Первая строка добавляется вручную за счет конфигурирования строго типизированного DataRow, а вторая — путем передачи необходимых данных столбцов, что позволяет создать DataRow автоматически “за кулисами”.

```

public static void AddRecords(AutoLotDataSet.InventoryDataTable tb,
    InventoryTableAdapter dAdapt)
{
    try
    {
        // Получить из таблицы новую строго типизированную строку.
        AutoLotDataSet.InventoryRow newRow = tb.NewInventoryRow();
        // Заполнить строку данными.
        newRow.CarID = 999;
        newRow.Color = "Purple";
        newRow.Make = "BMW";
        newRow.PetName = "Saku";

        // Вставить новую строку.
        tb.AddInventoryRow(newRow);

        // Добавить еще одну строку, используя перегруженный метод добавления.
        tb.AddInventoryRow(888, "Yugo", "Green", "Zippy");

        // Обновить базу данных.
        dAdapt.Update(tb);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}

```

Этот метод можно вызвать в `Main()`; это приведет к обновлению таблицы базы данных новыми записями:

```
static void Main(string[] args)
{
    ...
    // Добавить строки, обновить и вывести повторно.
    AddRecords(table, dAdapt);
    table.Clear();
    dAdapt.Fill(table);
    PrintInventory(table);
    Console.ReadLine();
}
```

Удаление данных с помощью сгенерированного кода

Удаление записей с помощью строго типизированной объектной модели также не представляет трудностей. Автоматически сгенерированный метод `FindByXXXX()` (где **XXXX** — имя столбца первичного ключа) строго типизированного класса `DataTable` возвращает корректный (строго типизированный) `DataRow`, используя первичный ключ. Ниже представлен код еще одного вспомогательного метода, который удаляет две только что созданные записи:

```
private static void RemoveRecords(AutoLotDataSet.InventoryDataTable tb,
    InventoryTableAdapter dAdapt)
{
    try
    {
        AutoLotDataSet.InventoryRow rowToDelete = tb.FindByCarID(999);
        dAdapt.Delete(rowToDelete.CarID, rowToDelete.Make,
            rowToDelete.Color, rowToDelete.PetName);
        rowToDelete = tb.FindByCarID(888);
        dAdapt.Delete(rowToDelete.CarID, rowToDelete.Make,
            rowToDelete.Color, rowToDelete.PetName);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}
```

Если вызвать этот метод в `Main()` и повторно вывести содержимое таблицы, вы должны увидеть, что две ранее вставленных тестовых записи больше не отображаются.

На заметку! Если запустить это приложение второй раз с вызовом методом `AddRecord()` каждый раз, то возникнет ошибка `VIOLATION CONSTRAINT ERROR`, потому что метод `AddRecord()` оба раза пытается вставить одно и то же значение первичного ключа `CardID` (поэтому логика доступа к данным помещена в блок `try/catch`). Чтобы обеспечить большую гибкость в этом примере, можно запрашивать данные у пользователя с применением класса `Console`.

Вызов хранимой процедуры с помощью сгенерированного кода

Давайте рассмотрим еще один пример использования строго типизированной объектной модели. Создадим последний метод, который вызывает хранимую процедуру `GetPetName`. Когда создавались адаптеры данных для базы `AutoLot`, был сгенерирован специальный класс по имени `QueriesTableAdapter`, который инкапсулирует процесс вызова хранимых процедур реляционной базы данных. Ниже приведен код финального

вспомогательного метода, который в случае вызова в `Main()` отображает название указанного автомобиля:

```
public static void CallStoredProc()
{
    try
    {
        QueriesTableAdapter q = new QueriesTableAdapter();
        Console.Write("Enter ID of car to look up: ");
        string carID = Console.ReadLine();
        string carName = "";
        q.GetPetName(int.Parse(carID), ref carName);
        Console.WriteLine("CarID {0} has the name of {1}", carID, carName);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}
```

К этому моменту вы знаете, как работать со строго типизированными классами базы данных и упаковывать их в отдельную библиотеку классов. В этой объектной модели есть и другие аспекты, с которыми можно поэкспериментировать, но вы уже знаете достаточно, чтобы самостоятельно разобраться в них. В завершение этой главы будет показано, как применять LINQ-запросы к объекту `DataSet` из ADO.NET.

Исходный код. Проект `StronglyTypedDataSetConsoleClient` доступен в подкаталоге Chapter 22.

Программирование с помощью LINQ to DataSet

В этой главе вы узнали, что данными внутри `DataSet` можно манипулировать тремя различными способами:

- с использованием коллекций `Tables`, `Rows` и `Columns`;
- с использованием объектов чтения таблиц данных;
- с использованием строго типизированных классов данных.

Различные индексаторы типов `DataSet` и `DataTable` позволяют взаимодействовать с содержащимися данными в прямолинейной, но слабо типизированной манере. Вспомните, что этот запрос требует трактовки данных как табличного блока ячеек, как показано в следующем примере:

```
static void PrintDataWithIndexers(DataTable dt)
{
    // Вывести содержимое DataTable.
    for (int curRow = 0; curRow < dt.Rows.Count; curRow++)
    {
        for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
        {
            Console.Write(dt.Rows[curRow][curCol].ToString() + "\t");
        }
        Console.WriteLine();
    }
}
```

Метод `CreateDataReader()` класса `DataTable` предлагает другой подход, при котором данные, содержащиеся в `DataSet`, трактуются как линейный набор строк, пред-

назначенный для обработки последовательным образом. Это позволяет применять модель программирования с подключенными объектами чтения данных к автономному DataSet.

```
static void PrintDataWithDataTableReader(DataTable dt)
{
    // Получить объект DataTableReader.
    DataTableReader dtReader = dt.CreateDataReader();
    while (dtReader.Read())
    {
        for (int i = 0; i < dtReader.FieldCount; i++)
        {
            Console.Write("{0}\t", dtReader.GetValue(i));
        }
        Console.WriteLine();
    }
    dtReader.Close();
}
```

И, наконец, можно использовать строго типизированный DataSet для получения кодовой базы, которая позволяет взаимодействовать с данными, содержащимися в объекте, с помощью свойств, которые отображаются на имена столбцов в реляционной базе данных. Строго типизированные объекты позволяют написать, к примеру, следующий код:

```
static void AddRowWithTypedDataSet()
{
    InventoryTableAdapter invDA = new InventoryTableAdapter();
    AutoLotDataSet.InventoryDataTable inv = invDA.GetData();
    inv.AddInventoryRow(999, "Ford", "Yellow", "Sal");
    invDA.Update(inv);
}
```

Хотя все эти подходы вполне работоспособны, существует еще один вариант — API-интерфейс LINQ to DataSet, который предназначен для манипулирования данными DataSet с помощью выражений запросов LINQ.

На заметку! API-интерфейс LINQ to DataSet используется для применения запросов LINQ только к объектам DataSet, которые возвращаются адаптерами данных, и это никак не связано с применением запросов LINQ напрямую к механизму базы данных. В главе 23 вы ознакомитесь с технологиями LINQ to Entities и ADO.NET Entity Framework, которые предоставляют способ представления запросов SQL в виде запросов LINQ.

В исходном состоянии объект DataSet из ADO.NET (и связанные с ним типы, такие как DataTable и DataView) не имеет необходимой инфраструктуры, чтобы служить непосредственной целью для запроса LINQ. Например, следующий метод (в котором применяются типы из пространства имен AutoLotDisconnectedLayer) дает в результате ошибку на этапе компиляции:

```
static void LinqOverDataTable()
{
    // Получить объект DataTable с данными.
    InventoryDALDisLayer dal = new InventoryDALDisLayer(
        @"Data Source=(local)\SQLEXPRESS;" +
        "Initial Catalog=AutoLot;Integrated Security=True");
    DataTable data = dal.GetAllInventory();

    // Применить запрос LINQ к DataSet?
    var moreData = from c in data where (int)c["CarID"] > 5 select c;
}
```

При компиляции метода `LinqOverDataTable()` компилятор сообщит, что тип `DataTable` предоставляет *реализацию шаблонов запросов*. Аналогично применению запросов LINQ к объектам, которые не реализуют интерфейс `IEnumerable<T>`, объекты ADO.NET должны быть трансформированы в совместимые типы. Чтобы понять, как это сделать, потребуется исследовать типы из `System.Data.DataSetExtensions.dll`.

Роль библиотеки расширений DataSet

Сборка `System.Data.DataSetExtensions.dll`, ссылка на которую по умолчанию присутствует во всех проектах Visual Studio, дополняет пространство имен `System.Data` набором новых типов (рис. 22.27).

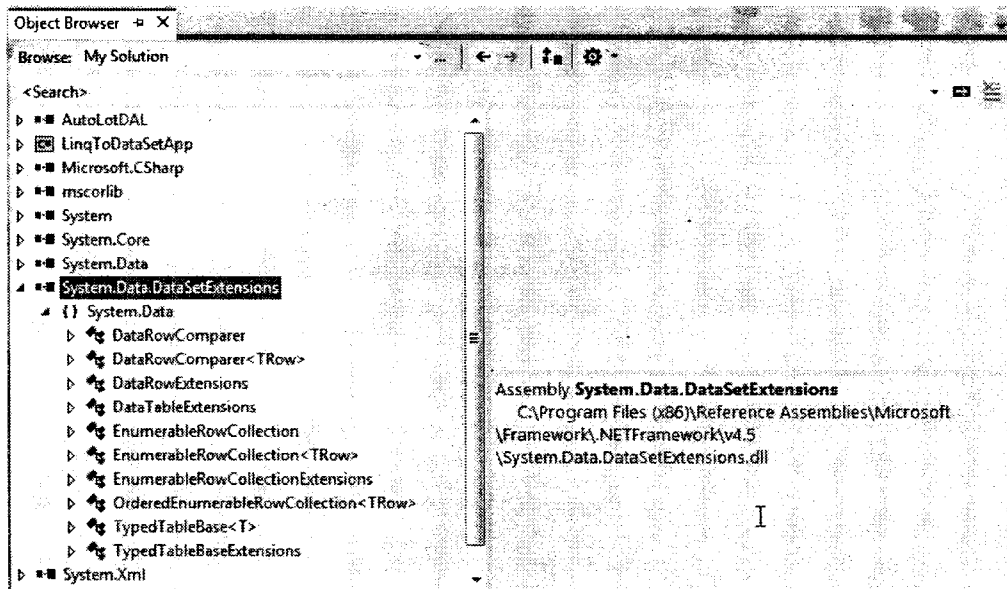


Рис. 22.27. Сборка `System.Data.DataSetExtensions.dll`

Двумя наиболее полезными типами являются `DataTableExtensions` и `DataRowExtensions`. Эти классы расширяют функциональность `DataTable` и `DataRow` за счет использования набора расширяющих методов (см. главу 12).

Еще один важный класс — `TypedTableBaseExtensions` — определяет расширяющие методы, которые можно применять к строго типизированным объектам `DataSet`, чтобы обеспечить поддержку LINQ для внутренних объектов `DataTable`. Все остальные члены сборки `System.Data.DataSetExtensions.dll` относятся к чистой инфраструктуре и не предназначены для непосредственного использования в кодовой базе.

Получение объекта DataTable, совместимого с LINQ

А теперь давайте посмотрим, как работать с расширениями `DataSet`. Предположим, что имеется новое консольное приложение C# по имени `LinqToDataSetApp`. Добавьте в него ссылку на последнюю версию (3.0.0.0) сборки `AutoLotDAL.dll` и модифицируйте первоначальный файл кода согласно следующей логике:

```
using System;
...
// Местоположение строго типизированных контейнеров данных.
using AutoLotDAL;

// Местоположение строго типизированных адаптеров данных.
using AutoLotDAL.AutoLotDataSetTableAdapters;
```

```

namespace LinqToDataSetApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** LINQ over DataSet *****\n");
            // Получить строго типизированный объект DataTable, содержащий
            // текущие данные таблицы Inventory из базы данных AutoLot.
            AutoLotDataSet dal = new AutoLotDataSet();
            InventoryTableAdapter da = new InventoryTableAdapter();
            AutoLotDataSet.InventoryDataTable data = da.GetData();

            // Вызвать описанные ниже методы.
            Console.ReadLine();
        }
    }
}

```

Чтобы преобразовать объект `DataTable` (в том числе и строго типизированный `DataTable`) из ADO.NET в объект, совместимый с LINQ, потребуется вызвать расширяющий метод `AsEnumerable()`, определенный в классе `DataTableExtensions`. Этот метод возвращает объект `EnumerableRowCollection`, который содержит коллекцию объектов `DataRow`. После этого тип `EnumerableRowCollection` можно использовать для обработки каждой строки с помощью основного синтаксиса `DataRow` (т.е. синтаксиса индексатора). Рассмотрим следующий новый метод класса `Program`, который принимает строго типизированный `DataTable`, получает перечислимую копию данных и выводит все значения `CarID`:

```

static void PrintAllCarIDs(DataTable data)
{
    // Получить перечислимую версию DataTable.
    EnumerableRowCollection enumData = data.AsEnumerable();

    // Вывести значения идентификаторов автомобилей.
    foreach (DataRow r in enumData)
        Console.WriteLine("Car ID = {0}", r["CarID"]);
}

```

Запросы LINQ пока еще не применялись, но здесь важно отметить, что объект `enumData` теперь может служить целью выражения запроса LINQ. Обратите внимание, что объект `EnumerableRowCollection` содержит коллекцию объектов `DataRow`, т.к. для вывода значений `CarID` к каждому подобъекту применяется индексатор типа.

В большинстве случаев нет необходимости объявлять переменную типа `EnumerableRowCollection` для хранения значения, возвращаемого методом `AsEnumerable()`. Этот метод можно вызывать внутри самого выражения запроса.

Ниже приведен код более интересного метода класса `Program`, который получает проекцию `CarID + Makes` из всех записей в `DataTable`, соответствующих автомобилям красного цвета (если в вашей таблице `Inventory` нет красных автомобилей, измените цвет в LINQ-запросе):

```

static void ShowRedCars(DataTable data)
{
    // Проецировать новый результирующий набор, содержащий
    // идентификатор/цвет для строк, в которых Color = Red.
    var cars = from car in data.AsEnumerable()
               where
                   (string)car["Color"] == "Red"

```

```

        select new
        {
            ID = (int)car["CarID"],
            Make = (string)car["Make"]
        };
Console.WriteLine("Here are the red cars we have in stock:");
foreach (var item in cars)
{
    Console.WriteLine("-> CarID = {0} is {1}", item.ID, item.Make);
}
}

```

Роль расширяющего метода `DataRowExtensions.Field<T>()`

Одним из нежелательных аспектов текущего выражения запроса LINQ является то, что для получения результирующего набора приходится применять множество операций приведения и индексаторов `DataRow`. Это может привести к исключениям времени выполнения, если будет предпринята попытка приведения несовместимого типа данных. Для внесения в запрос строгой типизации можно использовать расширяющий метод `Field<T>()` типа `DataRow`. Он позволяет увеличить безопасность к типам запроса, поскольку совместимость типов данных проверяется на этапе компиляции. Взгляните на следующее изменение:

```

var cars = from car in data.AsEnumerable()
where
    car.Field<string>("Color") == "Red"
select new
{
    ID = car.Field<int>("CarID"),
    Make = car.Field<string>("Make")
};

```

В этом случае можно вызвать метод `Field<T>()` и указать параметр типа для представления типа данных столбца. В качестве аргумента этому методу передается само имя столбца. Учитывая эту дополнительную проверку на этапе компиляции, при обработке элементов `EnumerableRowCollection` рекомендуется использовать метод `Field<T>()` (а не индексатор `DataRow`).

Кроме вызова метода `AsEnumerable()` общий формат запроса LINQ остается в точности таким же, как было показано в главе 13, поэтому не имеет смысла здесь повторять детали различных операций LINQ. Дополнительные примеры можно найти в разделе "LINQ to DataSet Examples" ("Примеры LINQ to DataSet") документации .NET Framework 4.5 SDK.

Заполнение новых объектов `DataTable` из запросов LINQ

Заполнить данными новый объект `DataTable` можно также на основе результатов запроса LINQ, при условии, что в нем не используются проекции. При наличии результирующего набора, тип которого может быть представлен как `IEnumerable<T>`, на нем можно вызвать расширяющий метод `CopyToDataTable<T>()`, как показано в следующем примере:

```

static void BuildDataTableFromQuery(DataTable data)
{
    var cars = from car in data.AsEnumerable()
where
    car.Field<int>("CarID") > 5
select car;
}

```

```
// Использовать этот результирующий набор для построения нового объекта DataTable.
DataTable newTable = cars.CopyToDataTable();

// Вывести содержимое DataTable.
for (int curRow = 0; curRow < newTable.Rows.Count; curRow++)
{
    for (int curCol = 0; curCol < newTable.Columns.Count; curCol++)
    {
        Console.Write(newTable.Rows[curRow][curCol].ToString().Trim() + "\t");
    }
    Console.WriteLine();
}
}
```

На заметку! Допускается также трансформировать запрос LINQ в тип DataView за счет применения расширяющего метода AsDataView<T>().

Этот прием может оказаться удобным, когда результат запроса LINQ нужно использовать в качестве источника для операции привязки к данным. Вспомните, что элемент управления DataGridView в Windows Forms (а также элемент управления типа сетки в ASP.NET) поддерживает свойство по имени DataSource. Привязать результат запроса LINQ к сетке можно было бы следующим образом:

```
// Предположим, что myDataGrid — это объект сетки графического
// пользовательского интерфейса.
myDataGrid.DataSource = (from car in data.AsEnumerable()
    where
        car.Field<int>("CarID") > 5
    select car).CopyToDataTable();
```

На этом исследование автономного уровня ADO.NET завершено. С помощью этого аспекта API-интерфейса можно извлекать данные из реляционной базы, обрабатывать их и возвращать обратно в базу, открывая подключение к базе данных лишь на минимальный промежуток времени.

Исходный код. Проект LinqToDataSetApp доступен в подкаталоге Chapter 22.

Резюме

В этой главе подробно рассматривался автономный уровень ADO.NET. Как вы видели, центральной частью автономного уровня является тип DataSet — размещаемое в памяти представление любого числа таблиц и дополнительно любого количества отношений, ограничений и выражений. Установка отношений между локальными таблицами удобна тем, что появляется возможность программной навигации между ними без подключения к удаленному хранилищу данных.

В главе также анализировалась роль типа адаптера данных. С помощью этого типа (и связанных свойств SelectCommand, InsertCommand, UpdateCommand и DeleteCommand) адаптер может переносить изменения из DataSet в исходное хранилище данных. Кроме того, вы научились осуществлять навигацию по объектной модели DataSet прямолинейным ручным способом, а также с помощью строго типизированных объектов, которые обычно генерируют инструменты визуального конструктора наборов данных в Visual Studio.

В конце главы был рассмотрен один из аспектов набора технологий LINQ под названием LINQ to DataSet. Он позволяет получать поддерживающую запросы копию DataSet, которую могут принимать правильно построенные запросы LINQ.

ADO.NET, часть III: Entity Framework

В предыдущих двух главах рассматривались фундаментальные программные модели ADO.NET, а именно — подключенный и автономный уровни. Эти подходы позволяли программистам .NET работать с реляционными данными (в относительно прямой манере) с самого первого выпуска платформы. Однако в версии .NET 3.5 Service Pack 1 был предложен совершенно новый компонент API-интерфейса ADO.NET под названием Entity Framework (EF).

Главная цель EF заключалась в том, чтобы предоставить возможность взаимодействия с реляционными базами данных через объектную модель, которая отображается непосредственно на бизнес-объекты в приложении. Например, вместо трактовки пакета данных как коллекции строк и столбцов можно оперировать коллекцией строго типизированных объектов, именуемых *сущностями* (entity). Эти сущности также естественным образом поддерживают LINQ, и к ним можно выполнять запросы с использованием той же грамматики LINQ, которая была описана в главе 12. Исполняющая среда EF транслирует запросы LINQ в подходящие запросы SQL.

В этой главе вы ознакомитесь с программной моделью EF. Будут подробно рассматриваться различные части инфраструктуры, включая службы объектов, клиент сущностей, LINQ to Entities и Entity SQL. Также будет описан формат наиболее важного файла *.edmx и его роль в API-интерфейсе Entity Framework. Вы научитесь генерировать файлы *.edmx с помощью Visual Studio и в командной строке, используя утилиту генератора EDM (EdmGen.exe).

К концу главы вы построите финальную версию сборки AutoLotDAL.dll и узнаете, как привязать сущностные объекты к настольному приложению Windows Forms.

Роль Entity Framework

Подключенный и автономный уровни ADO.NET снабжают фабрикой, которая позволяет выбирать, вставлять, обновлять и удалять данные с помощью объектов соединений, команд, чтения данных, адаптеров данных и DataSet. Хотя все это замечательно, такие аспекты ADO.NET заставляют трактовать полученные данные в манере, которая тесно связана с физической схемой данных. Вспомните, например, что при использовании подключенного уровня обычно производится итерация по каждой записи за счет указания имен столбцов объекту чтения данных. С другой стороны, в случае работы с автономным уровнем придется иметь дело с коллекциями строк и столбцов объекта DataTable внутри контейнера DataSet.

Если используется автономный уровень в сочетании со строго типизированными классами `DataSet` или адаптерами данных, то получается программная абстракция, которая обеспечивает ряд важных преимуществ. Во-первых, строго типизированный класс `DataSet` открывает данные таблицы через свойства класса. Во-вторых, строго типизированный адаптер таблицы поддерживает методы, которые инкапсулируют конструирование лежащих в основе операторов SQL. Вспомните следующий метод `AddRecords()` из главы 22:

```
public static void AddRecords(AutoLotDataSet.InventoryDataTable tb,
                             InventoryTableAdapter dAdapt)
{
    // Получить из таблицы новую строго типизированную строку.
    AutoLotDataSet.InventoryRow newRow = tb.NewInventoryRow();

    // Заполнить строку данными.
    newRow.CarID = 999;
    newRow.Color = Purple;
    newRow.Make = BMW;
    newRow.PetName = Saku;

    // Вставить новую строку.
    tb.AddInventoryRow(newRow);

    // Добавить еще одну строку с помощью перегруженного метода Add.
    tb.AddInventoryRow(888, Yugo, Green, Zippy);

    // Обновить базу данных.
    dAdapt.Update(tb);
}
```

Все становится еще лучше, если скомбинировать автономный уровень с `LINQ to DataSet`. В этом случае можно применить запросы `LINQ` к находящимся в памяти данным для получения нового результирующего набора и затем дополнительно отобразить его на автономный объект, такой как `DataTable`, `List<T>`, `Dictionary<K, V>` или массив данных:

```
static void BuildDataTableFromQuery(DataTable data)
{
    var cars = from car in data.AsEnumerable()
               where car.Field<int>(CarID) > 5 select car;

    // Использовать этот результирующий набор для построения нового объекта DataTable.
    DataTable newTable = cars.CopyToDataTable();

    // Работать с объектом DataTable...
}
```

Несмотря на удобство интерфейса `LINQ to DataSet`, следует помнить, что целью запроса `LINQ` являются данные, возвращаемые из базы данных, а не сам механизм базы данных. В идеале хотелось бы строить запрос `LINQ`, который отправлялся бы на обработку непосредственно базе данных и возвращал строго типизированные данные (именно это и позволяет достичь ADO.NET Entity Framework).

При использовании подключенного и автономного уровней ADO.NET всегда придется помнить о физической структуре лежащей в основе базы данных. Необходимо знать схему каждой таблицы данных, писать сложные SQL-запросы для взаимодействия с данными таблиц и т.д. Это вынуждает писать довольно громоздкий код C#, поскольку C# существенно отличается от языка самой базы данных.

Хуже того, способ построения физической базы данных (администратором баз данных) полностью сосредоточен на таких конструкциях, как внешние ключи, представления и хранимые процедуры. Сложность баз данных, спроектированных адми-

нистратором, может еще более возрастая, если администратор при этом заботится о безопасности и масштабируемости. Это также усложняет код C#, который приходится писать для взаимодействия с хранилищем данных.

Платформа ADO.NET Entity Framework (EF) — это программная модель, которая пытается заполнить пробел между конструкциями базы данных и объектно-ориентированными конструкциями. Используя EF, можно взаимодействовать с реляционными базами данных, не имея дела с кодом SQL (при желании). Исполняющая среда EF генерирует подходящие операторы SQL, когда вы применяете запросы LINQ к строго типизированным классам.

На заметку! *LINQ to Entities* — это термин, описывающий применение запросов LINQ к сущностным объектам ADO.NET.

Другой возможный подход состоит в том, чтобы вместо обновления базы данных посредством нахождения строки, обновления строки и отправки строки обратно на обработку в пакете запросов SQL, просто изменять свойства объекта и сохранять его состояние. И в этом случае исполняющая среда EF обновляет базу данных автоматически.

В Microsoft считают ADO.NET Entity Framework просто еще одним подходом к организации API-интерфейса доступа к данным и не намерены заменять им подключенный и автономный уровни. Однако после относительно недолгого использования EF этой развитой объектной модели часто отдается предпочтение перед относительно примитивным миром SQL-запросов и коллекций строк/столбцов.

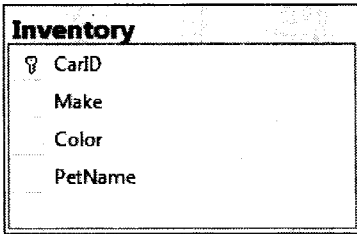
Тем не менее, иногда в проектах .NET используются все три подхода, поскольку одна только модель EF может чрезмерно усложнить код. Например, при построении внутреннего приложения, которому нужно взаимодействовать с единственной таблицей базы данных, подключенный уровень может применяться для запуска пакета хранимых процедур. Существенно выиграть от использования EF могут более крупные приложения, особенно если команда разработчиков уверено работает с LINQ. Как с любой новой технологией, следует знать, как (и когда) имеет смысл применять ADO.NET EF.

На заметку! Вспомните, что в .NET 3.5 появился API-интерфейс программирования для баз данных под названием LINQ to SQL. Он был построен на основе концепции, близкой (и даже очень близкой в смысле конструкций программирования) к ADO.NET EF. Хотя LINQ to SQL формально еще существует, официальное мнение в Microsoft состоит в том, что теперь следует обращать внимание на EF, а не на LINQ to SQL.

Роль сущностей

Строго типизированные классы, упомянутые ранее, называются *сущностями*. Сущности — это концептуальная модель физической базы данных, которая отображается на предметную область. Формально говоря, эта модель называется *моделью сущностных данных* (Entity Data Model — EDM). Модель EDM представляет собой набор классов клиентской стороны, которые отображаются на физическую базу данных. Тем не менее, необходимо понимать, что сущности вовсе не обязаны напрямую отображаться на схему базы данных, как может показаться, исходя из названия. Сущностные классы можно реструктурировать для соответствия существующим потребностям, и исполняющая среда EF отобразит эти уникальные имена на корректную схему базы данных.

Например, вспомним простую таблицу Inventory из базы данных AutoLot; схема которой показана на рис. 23.1.



Inventory	
CarID	
Make	
Color	
PetName	

Рис. 23.1. Структура таблицы Inventory базы данных AutoLot

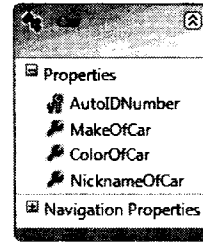


Рис. 23.2. Сущностный класс Car — это клиентское представление схемы Inventory

Если сгенерировать EDM для таблицы Inventory базы данных AutoLot (ниже будет показано, как это делается), то по умолчанию сущность будет называться Inventory. Тем не менее, сущностный класс можно переименовать в Car и определить для него уникально именованные свойства по своему выбору, которые будут отображены на столбцы таблицы Inventory. Такая слабая привязка означает возможность формирования сущностей так, чтобы они наиболее точно соответствовали предметной области. На рис. 23.2 показан пример сущностного класса.

На заметку! Во многих случаях сущностный класс клиентской стороны называется по имени связанной с ним таблицы базы данных. Однако помните, что вы всегда можете изменить сущность для более точного соответствия конкретной ситуации.

Вскоре мы построим полноценный пример с применением EF. Однако сейчас давайте рассмотрим следующий класс Program, в котором используется сущностный класс Car (и связанный с ним класс по имени AutoLotEntities) для добавления новой строки к таблице Inventory базы данных AutoLot. Этот класс называется *контекстом объектов*, и его назначение — обеспечивать взаимодействие с физической базой данных (о деталях речь пойдет ниже).

```

class Program
{
    static void Main(string[] args)
    {
        // Строка соединения автоматически читается
        // из сгенерированного конфигурационного файла.
        using (AutoLotEntities context = new AutoLotEntities())
        {
            // Добавить новую строку в таблицу Inventory, используя сущность Car.
            context.Cars.AddObject(new Car() { AutoIDNumber = 987, CarColor = Black,
                                                MakeOfCar = Pinto,
                                                NicknameOfCar = Pete });

            context.SaveChanges();
        }
    }
}
  
```

Обязанность исполняющей среды EF — позаботиться о клиентском представлении таблицы Inventory (класс по имени Car в рассматриваемом случае) и выполнить обратное отображение на корректные столбцы таблицы Inventory. Обратите внимание, что здесь нет никаких следов SQL-оператора INSERT; производится просто добавление нового объекта Car в коллекцию, поддерживаемую соответственно названным свойством Cars в контексте объектов, после чего изменения сохраняются. Если затем заглянуть в таблицу данных с помощью проводника сервера в Visual Studio, можно увидеть там добавленную новую строку (рис. 23.3).

CarID	Make	Color	PetName
16	Ford	White	FooFoo
32	VW	Black	Zippy
83	Ford	Rust	Rusty
872	Saab	Black	Mel
888	Yugo	Yellow	Clunker
987	Pinto	Black	Pete
1000	BMW	Black	Bimmer
1011	BMW	Green	Hank
2911	BMW	Pink	Pinky
12345	Yugo	Green	Zippy
* NULL	NULL	NULL	NULL

Рис. 23.3. Результат сохранения контекста

В приведенном примере нет никакой магии. “За кулисами” процесса открывается соединение с базой данных, генерируется подходящий оператор SQL и т.д. Преимущество EF заключается в том, что эти детали обрабатываются без вашего участия. Теперь давайте взглянем на базовые службы EF, которые все это делают.

Строительные блоки Entity Framework

API-интерфейс EF находится на вершине существующей инфраструктуры ADO.NET, которая рассматривалась в предыдущих двух главах. Подобно любому взаимодействию ADO.NET, платформа EF использует поставщик данных ADO.NET для взаимодействия с хранилищем данных. Однако поставщик данных должен быть модернизирован, чтобы поддерживать новый набор служб, прежде чем он сможет взаимодействовать с API-интерфейсом EF. И как можно было ожидать, поставщик данных Microsoft SQL Server уже обновлен соответствующей инфраструктурой, которая полагается на использование сборки `System.Data.Entity.dll`.

На заметку! Многие сторонние системы управления базами данных (например, Oracle и MySQL) предоставляют EF-совместимые поставщики данных. Детальную информацию можно узнать у поставщика системы управления базами данных или просмотреть список известных поставщиков данных ADO.NET по адресу www.sqlsummit.com/dataprov.htm.

В дополнение к добавлению необходимых компонентов к поставщику данных Microsoft SQL Server, сборка `System.Data.Entity.dll` содержит различные пространства имен, которые сами полагаются на службы EF. Две ключевых части API-интерфейса EF, на которые следует обратить внимание сейчас — это *службы объектов* и *клиент сущности*.

Роль служб объектов

Под *службами объектов* подразумевается часть EF, которая управляет сущностями клиентской стороны при работе с ними в коде. Службы объектов отслеживают изменения, внесенные в сущность (например, смена цвета автомобиля с зеленого на синий), управляют отношениями между сущностями (скажем, просмотр всех заказов для клиента с заданным именем), а также обеспечивают возможности сохранения изменений в базе данных и сохранение состояния сущности с помощью сериализации (XML и двоичной).

С точки зрения программирования уровень служб объектов управляет любым классом, расширяющим базовый класс `EntityObject`. Как и ожидалось, `EntityObject` находится в цепочке наследования любого сущностного класса в рамках программной модели ЕF. Например, взглянув на цепочку наследования сущностного класса `Car` из предыдущего примера, можно увидеть, что `Car` связан с `EntityObject` отношением “является” (рис. 23.4).

Роль клиента сущности

Еще одним важным аспектом API-интерфейса ЕF является уровень клиента сущности. Эта часть API-интерфейса ЕF отвечает за работу с поставщиком данных ADO.NET для установки соединений с базой данных, генерации необходимых SQL-операторов на основе состояния сущностей и запросов LINQ, отображения извлеченных данных на корректные формы сущностей, а также управления прочими деталями, которые обычно приходится делать вручную, если не используется Entity Framework.

Функциональность уровня клиента сущности определена в пространстве имен `System.Data.EntityClient`. Указанное пространство имен включает набор классов, которые отображают концепции ЕF (такие как запросы LINQ to Entity) на лежащего в основе поставщика данных ADO.NET. Эти классы (т.е. `EntityCommand` и `EntityConnection`) очень похожи на классы, которые можно найти в составе поставщика данных ADO.NET; например, на рис. 23.5 показано, что классы уровня клиента сущности расширяют те же абстрактные базовые классы любого другого поставщика (например, `DbCommand` и `DbConnection`; дополнительные сведения по этой теме приведены в главе 21).

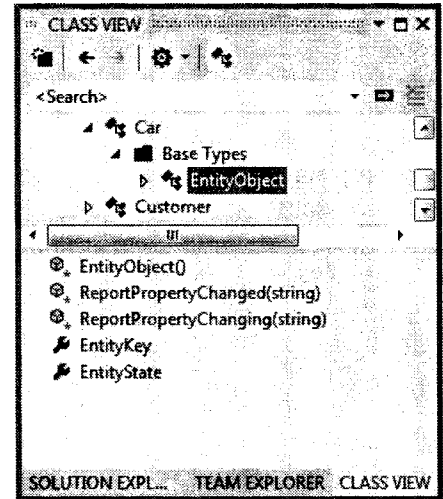


Рис. 23.4. Уровень служб объектов ЕF может управлять любым классом, расширяющим `EntityObject`

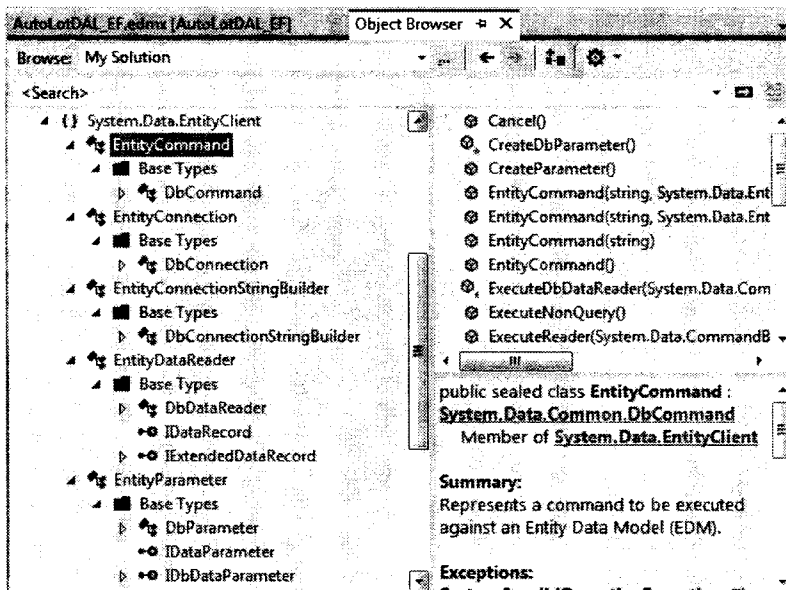


Рис. 23.5. Уровень клиента сущности отображает команды сущности на лежащего в основе поставщика данных ADO.NET

Уровень клиента сущности обычно работает “за кулисами”, но вполне допустимо взаимодействовать с клиентом сущности напрямую, если нужен полный контроль над его действиями (прежде всего, над генерацией запросов SQL и обработкой возвращаемых данных из базы).

Если требуется более точный контроль над тем, как сущностный клиент строит SQL-оператор на основе входящего запроса LINQ, можно использовать *Entity SQL*. Это независимый от базы данных диалект SQL, который работает непосредственно с сущностями. Построенный запрос Entity SQL может быть отправлен непосредственно службам клиента сущности (или, при желании, службам объектов), где он будет сформатирован в подходящий SQL-оператор для лежащего в основе поставщика данных. Далее в этой главе будет приведено несколько примеров применения Entity SQL.

Если требуется более высокая степень контроля над манипуляциями с извлеченными результатами, можно отказаться от автоматического отображения результатов базы данных на сущностные объекты и вручную обрабатывать записи с помощью класса *EntityDataReader*. Как и можно было ожидать, *EntityDataReader* позволяет обрабатывать извлеченные данные с применением однонаправленного, доступного только для чтения потока данных, как это делает *SqlDataReader*. Ниже в главе рассматривается работающий пример этого подхода.

Роль файла *.edmx

Подведем итог сказанному: сущности — это классы клиентской стороны, которые функционируют как модель сущностных данных (Entity Data Model). Хотя сущности клиентской стороны в конечном итоге отображаются на таблицу базы данных, жесткая связь между именами свойств сущностных классов и именами столбцов таблиц с данными отсутствует.

В контексте API-интерфейса Entity Framework для корректного отображения данных сущностных классов на данные таблиц требуется правильное определение логики отображения. В любой системе, управляемой моделью данных, уровни сущностей, реальной базы данных и отображения разделены на три связанных части: *концептуальная модель*, *логическая модель* и *физическая модель*.

- Концептуальная модель определяет сущности и отношения между ними (при их наличии).
- Логическая модель отображает сущности и отношения на таблицы с любыми необходимыми ограничениями внешних ключей.
- Физическая модель представляет возможности конкретного механизма данных, указывая детали хранилища, такие как табличная схема, разбиение на разделы и индексация.

В мире EF каждый из этих трех уровней фиксируется в файле XML-формата. В результате использования интегрированных визуальных конструкторов Entity Framework из Visual Studio получается файл с расширением *.edmx. Этот файл содержит XML-описания для сущностей, физической базы данных и инструкций по отображению этой информации между концептуальной и физической моделью. Формат файла *.edmx рассматривается в первом примере этой главы.

При компиляции основанных на EF проектов в Visual Studio файл *.edmx применяется для генерации трех отдельных XML-файлов: одного для концептуальной модели данных (*.csdl), одного для физической модели (*.ssdl) и одного для уровня отображения (*.msl). Данные из этих трех XML-файлов затем объединяются с приложением в виде двоичных ресурсов. После компиляции сборка .NET имеет все необходимые данные для вызовов API-интерфейса EF, имеющихся в коде.

Роль классов `ObjectContext` и `ObjectSet<T>`

Последним фрагментом мозаики ЕФ является класс `ObjectContext`, определенный в пространстве имен `System.Data.Objects`. Генерация файла `*.edmx` дает в результате сущностные классы, которые отображаются на таблицы базы данных, и класс, расширяющий `ObjectContext`. Обычно этот класс используется для непрямого взаимодействия со службами объектов и функциональностью клиента сущности.

Класс `ObjectContext` предлагает набор основных служб для дочерних классов, включая возможность сохранения всех изменений (которые в конечном итоге сводятся к обновлению базы данных), настройку строки соединения, удаление объектов, вызов хранимых процедур, а также обработку других фундаментальных деталей. В табл. 23.1 описаны некоторые ключевые члены класса `ObjectContext` (не забывайте, что большинство этих членов остаются в памяти, пока не будет произведен вызов `SaveChanges()`).

Таблица 23.1. Общие члены `ObjectContext`

Член	Описание
<code>AcceptAllChanges()</code>	Принимает все изменения, проведенные в сущностных объектах внутри контекста объектов
<code>AddObject()</code>	Добавляет объект к контексту объектов
<code>DeleteObject()</code>	Помечает объект для удаления
<code>ExecuteFunction<T>()</code>	Выполняет хранимую процедуру в базе данных
<code>ExecuteStoreCommand()</code>	Позволяет отправлять команду SQL прямо в хранилище данных
<code>GetObjectByKey()</code>	Находит объект внутри контекста объектов по его ключу
<code>SaveChanges()</code>	Отправляет все обновления в хранилище данных
<code>CommandTimeout</code>	Это свойство получает или устанавливает значение таймута в секундах для всех операций контекста объектов
<code>Connection</code>	Это свойство возвращает строку соединения, используемую текущим контекстом объектов
<code>SavingChanges</code>	Это событие инициируется, когда контекст объектов сохраняет изменения в хранилище данных

Производный от `ObjectContext` класс служит контейнером, управляющим сущностными объектами, которые сохраняются в коллекции типа `ObjectSet<T>`. Например, в результате генерации файла `*.edmx` для таблицы `Inventory` базы данных `AutoLot` получается класс с именем (по умолчанию) `AutoLotEntities`. Этот класс поддерживает свойство по имени `Inventories` (обратите внимание на форму множественного числа), которое инкапсулирует член данных `ObjectSet<Inventory>`. В случае создания EDM для таблицы `Orders` базы данных `AutoLot` в классе `AutoLotEntities` определено второе свойство по имени `Orders`, которое инкапсулирует переменную-член `ObjectSet<Order>`. В табл. 23.2 описаны некоторые общие члены `System.Data.Objects.ObjectSet<T>`.

Таблица 23.2. Общие члены `ObjectSet<T>`

Член	Описание
<code>AddObject()</code>	Позволяет вставить новый сущностный объект в коллекцию
<code>CreateObject<T>()</code>	Создает новый экземпляр указанного сущностного типа
<code>DeleteObject</code>	Помечает объект для удаления

Добравшись до нужного свойства контекста объектов, можно вызывать любой член `ObjectSet<T>`. Еще раз взглянем на пример кода, приведенный в начале этой главы:

```
using (AutoLotEntities context = new AutoLotEntities())
{
    // Добавить новую строку в таблицу Inventory, используя сущность Car.
    context.Cars.AddObject(new Car() { AutoIDNumber = 987, CarColor = Black,
                                      MakeOfCar = Pinto,
                                      NicknameOfCar = Pete });

    context.SaveChanges();
}
```

Здесь `AutoLotEntities` «является» `ObjectContext`. Свойство `Cars` предоставляет доступ к переменной `ObjectSet<Car>`. Эта ссылка используется для вставки нового сущностного объекта `Car` и сообщения экземпляру `ObjectContext` о необходимости сохранения всех изменений в базе данных.

Обычной целью запросов LINQ to Entities является экземпляр класса `ObjectSet<T>`; этот класс поддерживает те же расширяющие методы, о которых говорилось в главе 12. Более того, `ObjectSet<T>` получает значительную часть своей функциональности от своего непосредственного родительского класса `ObjectQuery<T>`, который представляет строго типизированный запрос LINQ (или Entity SQL).

Собираем все вместе

Прежде чем построить первое приложение, в котором используется Entity Framework, взгляните на рис. 23.6, на котором показана организация API-интерфейса EF.

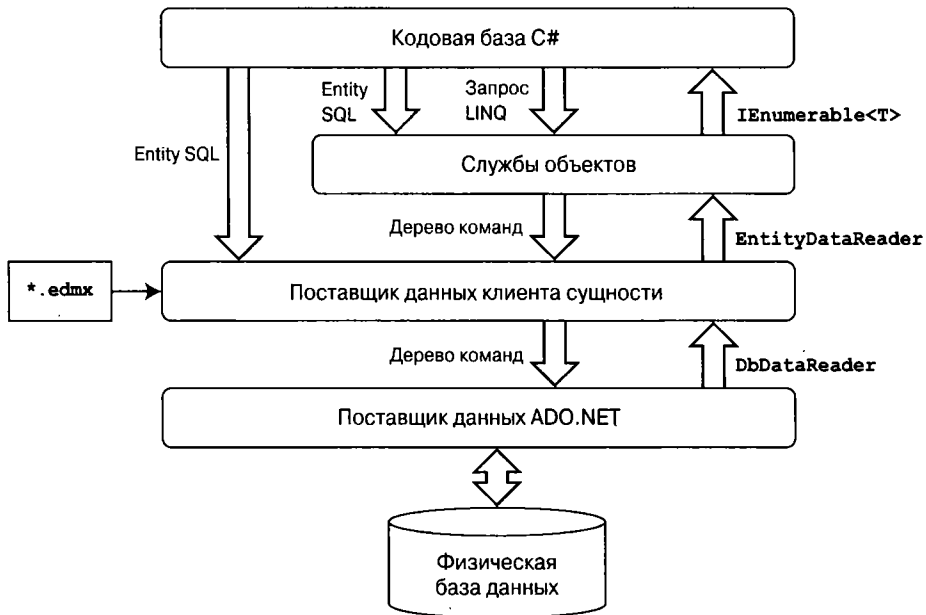


Рис. 23.6. Основные компоненты ADO.NET Entity Framework

Составные части на рис. 23.6 не столь сложны, как могут показаться на первый взгляд. Например, рассмотрим следующий распространенный сценарий. Вы пишете код C#, в котором применяется запрос LINQ к сущности, полученной от контекста. Этот запрос проходит через службы объектов, где преобразует команду LINQ в дерево, которое может понять клиент сущности. В свою очередь, клиент сущности форматирует это

дерево в подходящий оператор SQL для лежащего в основе поставщика ADO.NET. Этот поставщик вернет объект чтения данных (т.е. производный от `DbDataReader` объект), который клиентские службы используют для направления данных службам объектов с помощью `EntityDataReader`. Кодовая база получает обратно перечисление данных сущностей (`IEnumerable<T>`).

Теперь рассмотрим другой сценарий. Кодовая база C# желает получить больший контроль над тем, как клиентские службы конструируют конечный оператор SQL для отправки в базу данных. Поэтому код C# пишется с применением Entity SQL, который может быть передан непосредственно клиенту сущности или объектным службам. Конечный результат возвращается в виде `IEnumerable<T>`.

В любом из этих сценариев XML-данные из файла `*.edmx` должны быть сделаны известными клиентским службам; это позволит им понять, как следует отображать элементы базы данных на сущности. Наконец, помните, что клиент (т.е. кодовая база C#) также может получить результаты, отправленные клиентом сущности, используя `EntityDataReader` непосредственно.

Построение и анализ первой модели EDM

Понимая предназначение инфраструктуры ADO.NET Entity Framework, а также имея общее представление о ее работе, можно приступить к рассмотрению первого примера. Чтобы пока не усложнять картину, построим модель EDM, которая обеспечит доступ только к таблице `Inventory` базы данных `AutoLot`. Разобравшись с основами, мы затем построим новую модель EDM, которая будет рассчитана на всю базу данных `AutoLot`, и отобразим данные в графическом пользовательском интерфейсе.

Генерация файла `*.edmx`

Начнем с создания нового консольного приложения по имени `InventoryEDM ConsoleApp`. Когда планируется использование Entity Framework, первый шаг состоит в генерации необходимой концептуальной, логической и физической модели данных, определенной в файле `*.edmx`. Один из способов предусматривает применение для этого утилиты командной строки `EdmGen.exe` из .NET Framework 4.5 SDK. Откройте окно командной строки разработчика и введите следующую команду:

```
EdmGen.exe -?
```

На консоль должен быть выведен список опций, которые можно указывать утилите для генерации необходимых файлов на основе существующей базы данных; кроме того, доступны опции для генерации совершенно новой базы данных на основе имеющихся сущностных файлов. В табл. 23.3 описаны некоторые общие опции `EdmGen.exe`.

Как и платформа .NET 4.0 в целом, программная модель EF поддерживает программирование в стиле сначала предметная область, что позволяет создавать свойства (с применением типичных объектно-ориентированных приемов) и использовать их для генерации новой базы данных. В этом вводном обзоре ADO.NET EF ни подход “сначала модель”, ни генерация сущностной модели клиентской стороны с помощью утилиты `EdmGen.exe` применяться не будет. Вместо этого будут использоваться визуальные конструкторы EDM из среды Visual Studio.

Выберите пункт меню `Project⇒Add New Item` (Проект⇒Добавить новый элемент) и вставьте новый элемент `ADO.NET Entity Data Model` (Модель сущностных данных ADO.NET) по имени `InventoryEDM.edmx`, как показано на рис. 23.7.

Таблица 23.3. Часто используемые флаги командной строки утилиты EdmGen . exe

Опция	Описание
/mode:FullGeneration	Сгенерировать файлы *.ssdl, *.msl, *.csdl и клиентские сущности из указанной базы данных
/project:	Базовое имя, которое должно использоваться для сгенерированного кода и файлов. Обычно это имя базы данных, из которой извлекается информация (допускается сокращенная форма — /p:)
/connectionstring:	Строка соединения, используемая для взаимодействия с базой данных (допускается сокращенная форма — /c:)
/language:	Позволяет указать, какой синтаксис должен использоваться для сгенерированного кода — C# или VB
/pluralize	Позволяет автоматически выбирать форму множественного или единственного числа для имени набора сущностей, имени типа сущности и имени навигационного свойства, согласно правилам английского языка

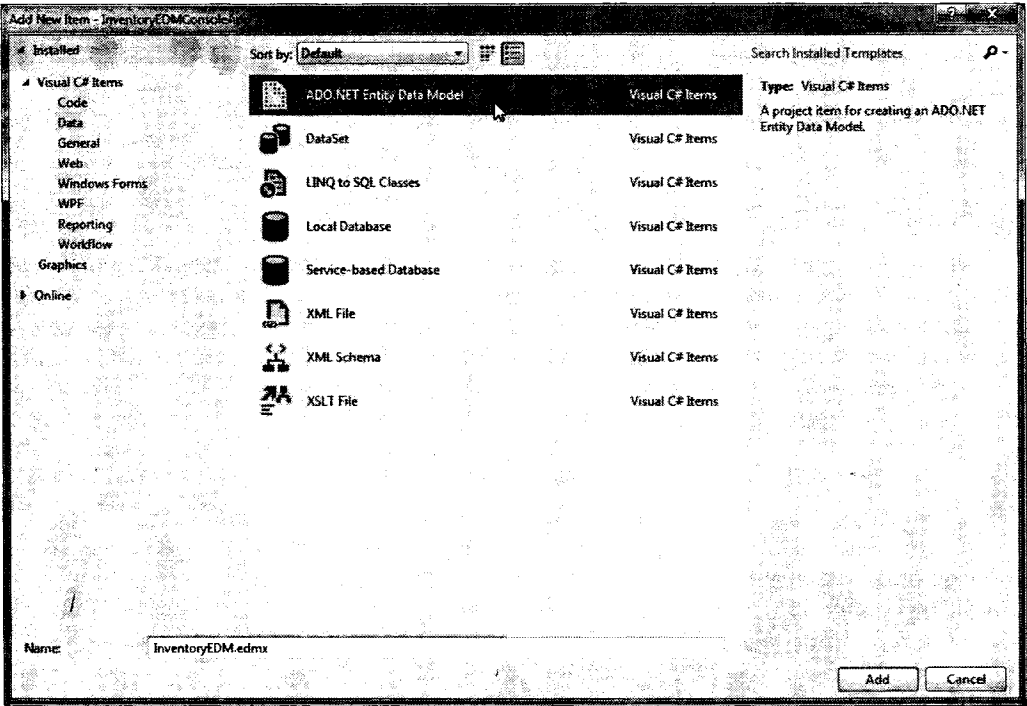


Рис. 23.7. Вставка нового элемента проекта ADO.NET Entity Data Model

Щелчок на кнопке Add (Добавить) приводит к запуску мастера создания модели сущностных данных (Entity Data Model Wizard). На первом шаге мастер позволяет выбрать, нужно ли генерировать EDM из существующей базы данных либо определить пустую модель (для разработки в стиле “сначала модель”). Выберите опцию Generate from database (Генерировать из базы данных) и щелкните на кнопке Next (Далее), как показано на рис. 23.8.

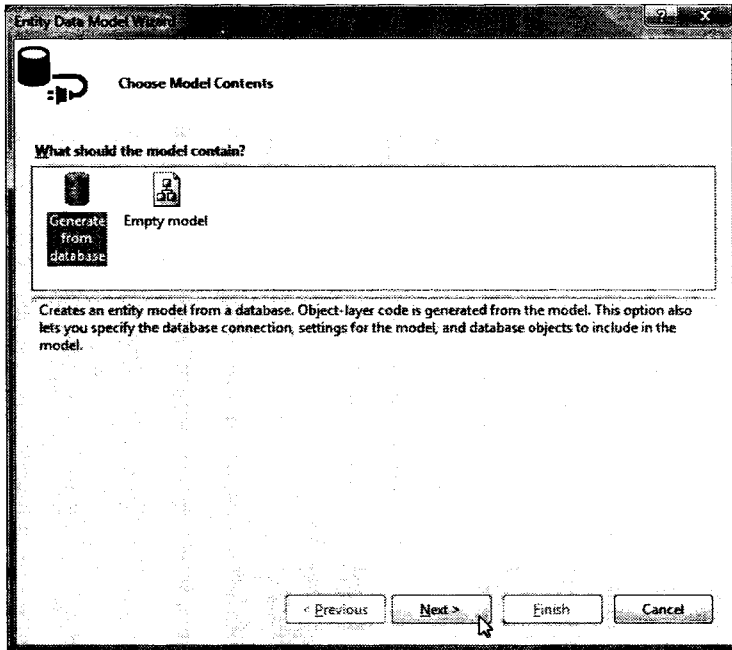


Рис. 23.8. Генерация модели EDM из существующей базы данных

На втором шаге мастера выбирается база данных. Если соединение с базой данных внутри проводника сервера Visual Studio уже существует, оно будет присутствовать в раскрывающемся списке. Если же нет, щелкните на кнопке **New Connection** (Создать соединение). В любом случае выберите базу данных **AutoLot** и отметьте флажок **Save entity connection settings in App.config as** (Сохранить настройки соединения в файле **App.config** как), как показано на рис. 23.9.

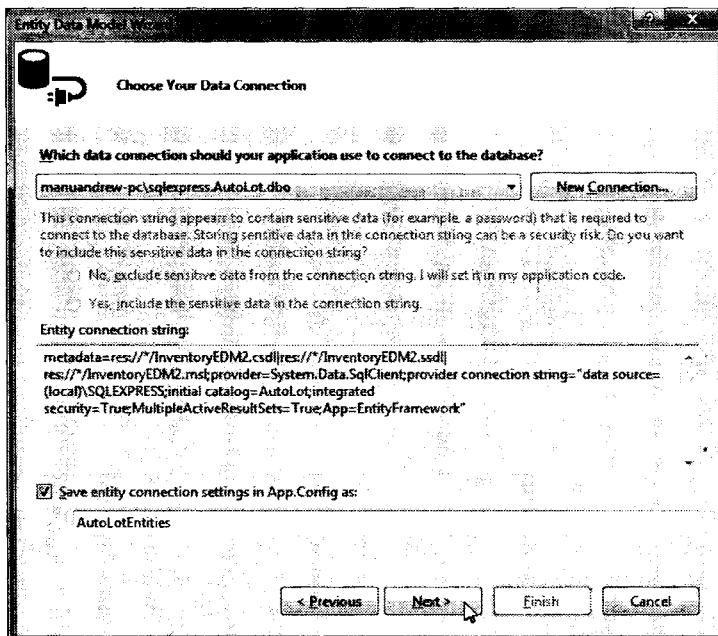


Рис. 23.9. Выбор базы данных для генерации EDM

Прежде чем щелкнуть на кнопке Next, взгляните на формат строки соединения:

```
metadata=res://*/InventoryEDM.csdl|res://*/InventoryEDM.ssdl|res://*/InventoryEDM.msl;
provider=System.Data.SqlClient;provider connection string=
Data Source=(local)\SQLEXPRESS;
Initial Catalog=AutoLot;Integrated Security=True;Pooling=False
```

Хотя ваша строка соединения может отличаться в зависимости от конфигурации машины, основной интерес здесь представляет флаг metadata, который используется для указания имен встроенных данных XML-ресурсов концептуального, физического и файла отображений (вспомните, что во время компиляции файл *.edmx будет разделен на три отдельных файла, и данные в этих файлах примут форму двоичных ресурсов, встраиваемых в сборку).

На последнем шаге мастера можно выбрать элементы из базы данных, для которой необходимо сгенерировать модель EDM. В рассматриваемом примере ограничимся только таблицей Inventory (рис. 23.10).

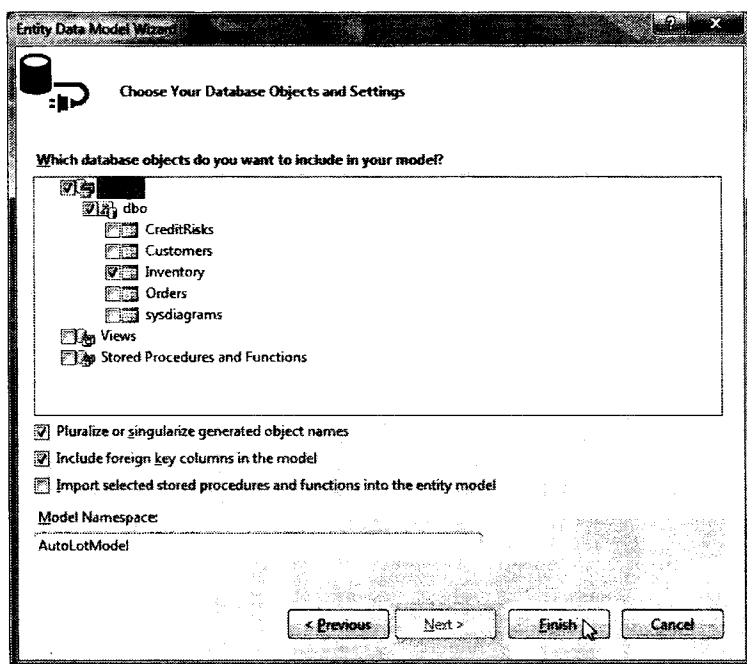


Рис. 23.10. Выбор элементов базы данных

Теперь щелкните на кнопке Finish (Готово) для генерации модели EDM.

Изменение формы сущностных данных

После завершения работы с мастером откроется визуальный конструктор EDM в IDE-среде с одной сущностью по имени Inventory. Просмотреть композицию любой сущности в визуальном конструкторе можно с помощью окна Model Browser (Браузер моделей), которое открывается через пункт меню View ⇒ Other Windows (Вид ⇒ Другие окна). Теперь взгляните на формат концептуальной модели для таблицы базы данных Inventory, представленный в папке Entity Types (Типы сущности), как показано на рис. 23.11. В узле хранилища, имя которого совпадает с именем базы данных (AutoModel.Store), находится физическая модель базы данных.

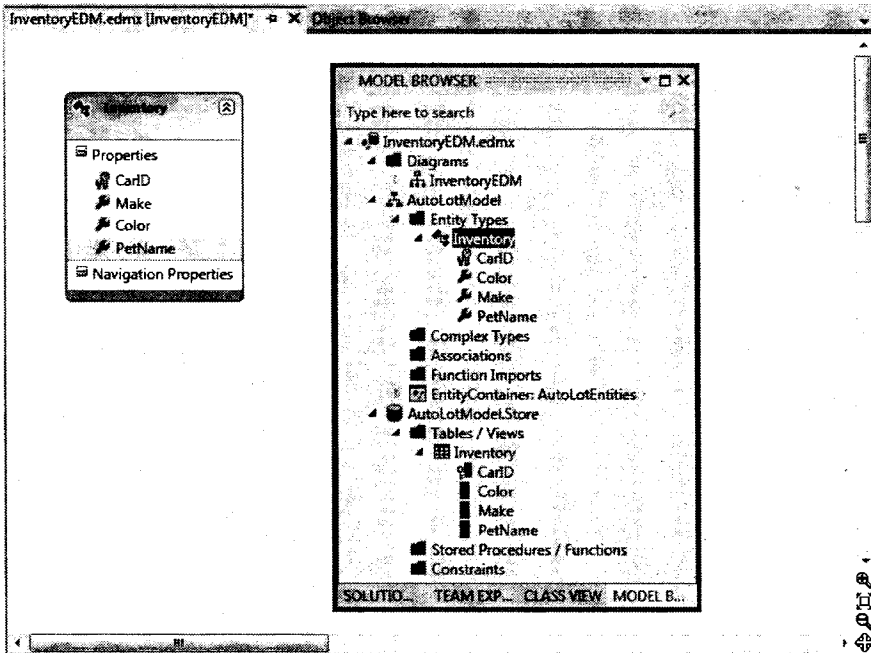


Рис. 23.11. Визуальный конструктор EDM и окно браузера моделей

По умолчанию имена сущностей будут основаны на именах исходных объектов баз данных; однако вспомните, что имена сущностей в концептуальной модели могут быть любыми. Чтобы изменить имя сущности либо имена свойств сущности, необходимо выбрать нужный элемент в визуальном конструкторе и установить соответствующим образом свойство Name в окне свойств (Properties). Переименуйте сущность Inventory в Car и свойство PetName в CarNickname (рис. 23.12).

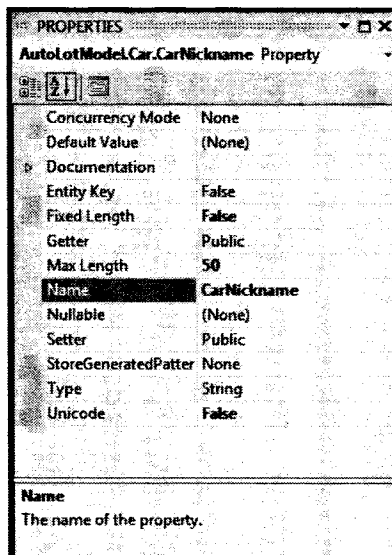


Рис. 23.12. Изменение формы сущностей с помощью окна свойств

К этому моменту концептуальная модель должна выглядеть примерно так, как показано на рис. 23.13.

Теперь выберите в визуальном конструкторе сущность `Car` и снова загляните в окно `Properties`. Вы должны увидеть поле `Entity Set Name` (Имя набора сущностей), также переименованное из `Inventories` в `Cars` (рис. 23.14). Значение `Entity Set Name` важно, потому что оно соответствует имени свойства в классе контекста данных, который используется для модификации базы данных. Вспомните, что это свойство инкапсулирует переменную-член `ObjectSet<T>` класса-наследника `ObjectContext`.

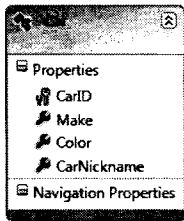


Рис. 23.13. Модель клиентской стороны, измененная в соответствии с предметной областью

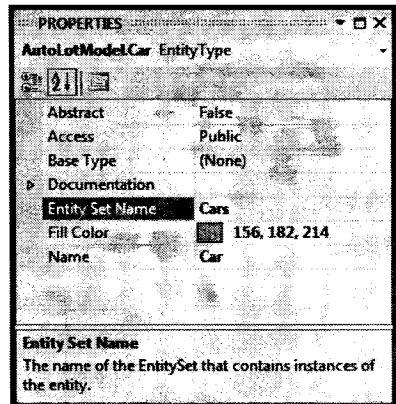


Рис. 23.14. Имя оболочки свойства `ObjectSet<T>`

Прежде чем двигаться дальше, скомпилируйте приложение; это приведет к обновлению кодовой базы и генерации файлов `*.csdl`, `*.msl` и `*.ssdl` на основе данных файла `*.edmx`.

Просмотр отображений

Теперь, изменив форму данных, можно просматривать отображения между концептуальным уровнем и физическим уровнем в окне `Mapping Details` (Подробности отображения), которое открывается через пункт меню `View` → `Other Windows` → `Mapping Details` (Вид → Другие окна → Подробности отображения). Взгляните на рис. 23.15 и обратите внимание, что узлы в левой части дерева представляют имена данных из физического уровня, в то время как узлы справа представляют имена концептуальной модели.

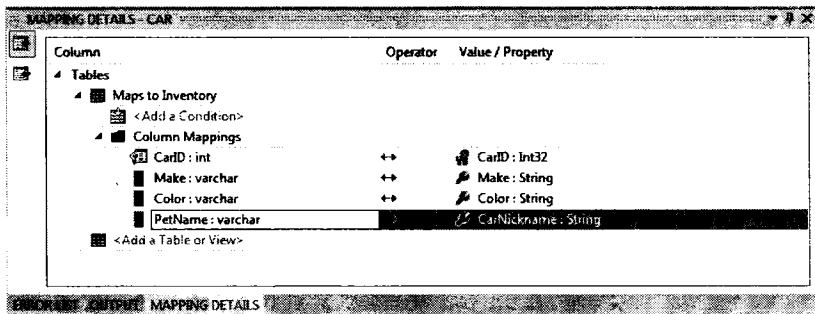


Рис. 23.15. В окне `Mapping Details` можно просматривать отображения концептуальной и физической моделей

Просмотр сгенерированного файла *.edmx

Теперь давайте посмотрим, что именно мастер EDM Wizard сгенерировал. Щелкните правой кнопкой мыши на файле InventoryEDM.edmx в проводнике решения и выберите в контекстном меню пункт Open With... (Открыть с помощью...). В открывшемся диалоговом окне выберите опцию XML Editor (Редактор XML). Это позволит просмотреть XML-данные, лежащие в основе представления, которое отображается в визуальном конструкторе EDM. Структура этого XML-документа содержится внутри корневого элемента <edmx:Edmx>.

Развернув корневой элемент, можно увидеть два дочерних элемента. Первый из них, <edmx:Runtime>, содержит метаданные, используемые приложением во время выполнения, а второй элемент, <Designer>, хранит метаданные, которые применяются Visual Studio на этапе разработки.

Внутри элемента <edmx:Runtime> находятся три дочерних элемента, которые описывают физическую модель хранения, логическую объектную модель C# и отображение между этими моделями. Давайте сначала кратко рассмотрим метаданные модели хранения.

```
<!-- Содержимое SSDL. -->
<edmx:StorageModels>
  <Schema Namespace=AutoLotModel.Store Alias=Self
    Provider=System.Data.SqlClient
    ProviderManifestToken=2008
    xmlns:store=
      http://schemas.microsoft.com/ado/2007/12/edm/EntityStoreSchemaGenerator
    xmlns=http://schemas.microsoft.com/ado/2009/02/edm/ssdl>
    <EntityContainer Name=AutoLotModelStoreContainer>
      <EntitySet Name=Inventory EntityType=AutoLotModel.Store.Inventory
        store:Type=Tables Schema=dbo />
    </EntityContainer>
    <EntityType Name=Inventory>
      <Key>
        <PropertyRef Name=CarID />
      </Key>
      <Property Name=CarID Type=int Nullable=false />
      <Property Name=Make Type=varchar Nullable=false MaxLength=50 />
      <Property Name=Color Type=varchar Nullable=false MaxLength=50 />
      <Property Name=PetName Type=varchar MaxLength=50 />
    </EntityType>
  </Schema>
</edmx:StorageModels>
```

Обратите внимание, что узел <Schema> определяет имя поставщика данных ADO.NET, который использует эту информацию при взаимодействии с базой данных (System.Data.SqlClient). Узлами <EntityType> помечается имя физической таблицы базы данных, а также каждый столбец в таблице.

Следующая важная часть файла *.edmx — элемент <edmx:ConceptualModels>, который определяет сущности клиентской стороны с измененной формой. Как показано ниже, сущность Cars определяет свойство CarNickname, которое было изменено в визуальном конструкторе.

```
<!-- Содержимое CSDL. -->
<edmx:ConceptualModels>
  <Schema Namespace=AutoLotModel Alias=Self
    xmlns:annotation=http://schemas.microsoft.com/ado/2009/02/edm/annotation
    xmlns=http://schemas.microsoft.com/ado/2008/09/edm>
```



```

<EntityContainer
  Name=AutoLotEntities annotation:LazyLoadingEnabled=true>
  <EntitySet Name=Cars EntityType=AutoLotModel.Car />
</EntityContainer>

<EntityType Name=Car>
  <Key>
    <PropertyRef Name=CarID />
  </Key>
  <Property Name=CarID Type=Int32 Nullable=false />
  <Property Name=Make Type=String Nullable=false MaxLength=50
    Unicode=false FixedLength=false />
  <Property Name=Color Type=String Nullable=false MaxLength=50
    Unicode=false FixedLength=false />
  <Property Name=CarNickname Type=String MaxLength=50
    Unicode=false FixedLength=false />
</EntityType>
</Schema>
</edmx:ConceptualModels>

```

Это перемещает на уровень отображения, который окно Mapping Details (а также исполняющая среда EF) применяют для подключения имен в концептуальной модели к физической модели:

```

<!-- Содержимое отображения C-S. -->
<edmx:Mappings>

  <Mapping Space=C-S
    xmlns=http://schemas.microsoft.com/ado/2008/09/mapping/cs>
    <EntityContainerMapping StorageEntityContainer=AutoLotModelStoreContainer
      CdmEntityContainer=AutoLotEntities>

      <EntitySetMapping Name=Cars>
        <EntityTypeMapping TypeName=AutoLotModel.Car>
          <MappingFragment StoreEntitySet=Inventory>
            <ScalarProperty Name=CarID ColumnName=CarID />
            <ScalarProperty Name=Make ColumnName=Make />
            <ScalarProperty Name=Color ColumnName=Color />
            <ScalarProperty Name=CarNickname ColumnName=PetName />
          </MappingFragment>
        </EntityTypeMapping>
      </EntitySetMapping>
    </EntityContainerMapping>
  </Mapping>
</edmx:Mappings>

```

Последней частью файла *.edmx является элемент <Designer>, который исполняющей средой EF не используется. Он содержит инструкции, применяемые Visual Studio для отображения сущностей на поверхности визуального конструктора.

Удостоверьтесь, что проект скомпилирован, по крайней мере, один раз, и щелкните на кнопке Show All Files (Показать все файлы) в проводнике решения. Затем зайдите в папку obj\Debug, а после этого — в edmxResourcesToEmbed. Здесь находятся три XML-файла, основанные на содержимом файла *.edmx (рис. 23.16).

Данные в этих файлах будут встраиваться в сборку как двоичные ресурсы. Таким образом, приложение .NET обладает всей информацией, необходимой для восприятия концептуального, физического и уровня отображения модели EDM.

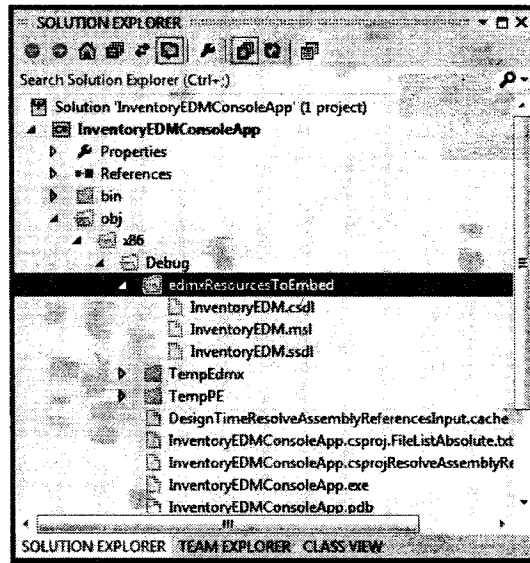


Рис. 23.16. Файл *.edmx используется для генерации трех отдельных XML-файлов

Просмотр сгенерированного исходного кода

Теперь вы почти готовы к тому, чтобы написать некоторый код, использующий построенную модель EDM. Однако прежде чем делать это, стоит заглянуть в сгенерированный код C#. Откройте окно Class View (Представление классов) и разверните стандартное пространство имен. В дополнение к классу Program там будет присутствовать сгенерированный мастером EDM Wizard сущностный класс (который вы переименовали в Car) и другой класс по имени AutoLotEntities.

Зайдя в проводник решения и раскрыв узел InventoryEDM.edmx, вы увидите поддерживаемый IDE-средой файл по имени InventoryEDM.Designer.cs. Как и любой другой файл подобного рода, его не следует редактировать напрямую, потому что IDE-среда пересоздает его при каждой компиляции. Тем не менее, этот файл можно открыть для просмотра двойным щелчком кнопкой мыши.

Класс AutoLotEntities расширяет классObjectContext, который (как вы, возможно, помните), представляет собой входную точку в программную модель EF. Конструктор предоставляет различные способы заполнения данных строки соединения. Стандартный конструктор сконфигурирован на автоматическое чтение данных строки соединения из файла App.config:

```
public partial class AutoLotEntities : ObjectContext
{
    public AutoLotEntities() : base(name=AutoLotEntities, AutoLotEntities)
    {
        this.ContextOptions.LazyLoadingEnabled = true;
        OnContextCreated();
    }
    ...
}
```

Далее обратите внимание, что свойство Cars класса AutoLotEntities инкапсулирует член данных ObjectSet<Car>. Это свойство можно использовать для работы с моделью EDM с целью не прямой модификации физической базы данных:

```

public partial class AutoLotEntities :ObjectContext
{
    ...
    public ObjectSet<Car> Cars
    {
        get
        {
            if ((_Cars == null))
            {
                _Cars = base.CreateObjectSet<Car>(Cars);
            }
            return _Cars;
        }
    }
    private ObjectSet<Car> _Cars;
}

```

На заметку! В классах, производных от `ObjectContext`, доступно множество методов, имена которых начинаются с `AddTo`. Хотя их можно применять для добавления новых сущностей к переменным-членам `ObjectSet<T>`, предпочтительнее делать это за счет обращения к члену `ObjectSet<T>`, полученному от строго типизированных свойств.

И последним интересным аспектом в файле кода визуального конструктора является сущностный класс `Car`. Значительная часть кода каждого сущностного класса представляет собой коллекцию, моделирующую форму концептуальной модели. Каждое из этих свойств реализует свою логику `set` за счет вызова статического метода `StructuralObject.SetValue()` из API-интерфейса EF.

Кроме того, логика `set` включает код, который информирует исполняющую среду EF о том, что состояние сущности изменилось; это важно, поскольку объект `ObjectContext` должен знать обо всех этих изменениях, чтобы вытолкнуть изменения в физическую базу данных.

Вдобавок внутри логики `set` производятся вызовы двух частичных методов. Вспомните, что частичный метод C# предоставляет простой способ обращения с уведомлениями об изменениях внутри приложений. Если частичный метод не реализован, компилятор его игнорирует и полностью отбрасывает. Ниже приведена реализация свойства `CarNickname` сущностного класса `Car`:

```

public partial class Car : EntityObject
{
    ...
    public global::System.String CarNickname
    {
        get
        {
            return _CarNickname;
        }
        set
        {
            OnCarNicknameChanging(value);
            ReportPropertyChanging(CarNickname);
            _CarNickname = StructuralObject.SetValidValue(value, true);
            ReportPropertyChanged(CarNickname);
            OnCarNicknameChanged();
        }
    }
    private global::System.String _CarNickname;
    partial void OnCarNicknameChanging(global::System.String value);
    partial void OnCarNicknameChanged();
}

```

Улучшение сгенерированного исходного кода

Все классы, сгенерированные визуальным конструктором, объявлены с ключевым словом `partial`, которое позволяет разнести реализацию класса по нескольким файлам кода C#. Это особенно полезно при работе с программной моделью EF, поскольку означает возможность добавлять “реальные” методы к сущностным классам, что помогает лучше моделировать предметную область.

В этом примере будет переопределен метод `ToString()` сущностного класса `Car` для возврата состояния сущности в виде хорошо форматированной строки. Также будут завершены определения частичных методов `OnCarNicknameChanging()` и `OnCarNicknameChanged()`, для обслуживания простых диагностических уведомлений. Определим следующий частичный класс в новом файле `Car.cs`:

```
public partial class Car
{
    public override string ToString()
    {
        // Поскольку столбец PetName может быть пустым,
        // указать в качестве стандартного имени **No Name**.
        return string.Format("{0} is a {1} {2} with ID {3}.,
            this.CarNickname ?? **No Name**,
            this.Color, this.Make, this.CarID);
    }
    partial void OnCarNicknameChanging(global::System.String value)
    {
        Console.WriteLine("\t-> Changing name to: {0}, value);
    }
    partial void OnCarNicknameChanged()
    {
        Console.WriteLine("\t-> Name of car has been changed!");
    }
}
```

Помните, что после предоставления реализаций этих частичных методов можно получать уведомления, когда свойства *сущностных классов* изменены или находятся на стадии изменения, но не при изменении физической базы данных. Если требуется знать, когда изменяется физическая база данных, можно обработать событие `SavingChanges` класса, производного от `ObjectContext`.

Программирование с использованием концептуальной модели

Теперь можно написать некоторый код, взаимодействующий с моделью EDM. Начнем с добавления в метод `Main()` класса `Program` вызова вспомогательного метода, который выводит каждый элемент из базы данных `Inventory` с применением концептуальной модели, и вызова вспомогательного метода, вставляющего новую запись в таблицу `Inventory`:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with ADO.NET EF *****\n");
        AddNewRecord();
        PrintAllInventory();
        Console.ReadLine();
    }
}
```

```

private static void AddNewRecord()
{
    // Добавить запись в таблицу Inventory базы данных AutoLot.
    using (AutoLotEntities context = new AutoLotEntities())
    {
        try
        {
            // Жестко закодировать данные новой записи (для целей тестирования) .
            context.Cars.AddObject(new Car() { CarID = 2222,
                Make = Yugo, Color = Brown });
            context.SaveChanges();
        }
        catch(Exception ex)
        {
            Console.WriteLine(ex.InnerException.Message);
        }
    }
}

private static void PrintAllInventory()
{
    // Выбрать все элементы из таблицы Inventory базы AutoLot
    // и вывести данные, используя специальный метод ToString()
    // сущностного класса Car.
    using (AutoLotEntities context = new AutoLotEntities())
    {
        foreach (Car c in context.Cars)
            Console.WriteLine(c);
    }
}
}

```

Код, похожий на показанный выше, уже встречался ранее в этой главе, но сейчас вы должны лучше представлять, как он работает. Каждый вспомогательный метод создает экземпляр производного от `ObjectContext` класса (`AutoLotEntities`) и использует строго типизированное свойство `Cars` для взаимодействия с полем `ObjectSet<Car>`. Перечисление каждого элемента, открытого свойством `Cars`, позволяет неявно передать SQL-оператор `SELECT` лежащему в основе поставщику данных ADO.NET. SQL-оператор `INSERT` выполняется путем вставки нового объекта `Car` с помощью метода `AddObject()` класса `ObjectSet<Car>` с последующим вызовом метода `SaveChanges()` на объекте контекста.

Удаление записи

Для того чтобы удалить запись из базы данных, сначала необходимо найти корректный элемент в `ObjectSet<T>`. Это можно сделать, передав объект `EntityKey` (член пространства имен `System.Data`) методу `GetObjectByKey()`. Предполагая, что это пространство имен импортировано в файл кода C#, теперь можно написать следующий вспомогательный метод:

```

private static void RemoveRecord()
{
    // Найти автомобиль для удаления по первичному ключу.
    using (AutoLotEntities context = new AutoLotEntities())
    {
        // Определить ключ для искомой сущности.
        EntityKey key = new EntityKey(AutoLotEntities.Cars, CarID, 2222);

        // Проверить ее существование, и если она существует, то удалить.
        Car carToDelete = (Car)context.GetObjectByKey(key);
    }
}

```

```

    if (carToDelete != null)
    {
        context.DeleteObject(carToDelete);
        context.SaveChanges();
    }
}

```

На заметку! Хорошо это или плохо, но вызов `GetObjectByKey()` требует обращения к базе данных, прежде чем можно будет удалить объект.

Обратите внимание, что при создании объекта `EntityKey` в первом аргументе передается объект `string`, информирующий о том, какой объект `ObjectSet<T>` должен обрабатываться в заданном классе, производном от `ObjectContext`. Вторым аргументом (еще один объект `string`) представляется имя свойства сущностного класса, которое служит ключом, а последний аргумент — это значение первичного ключа. Как только нужный объект найден, можно вызвать `DeleteObject()` на объекте контекста и сохранить изменения.

Обновление записи

Обновление записи также делается просто: найдите объект, который хотите изменить, установите новые значения свойств возвращенной сущности и сохраните изменения:

```

private static void UpdateRecord()
{
    // Найти автомобиль для обновления по первичному ключу.
    using (AutoLotEntities context = new AutoLotEntities())
    {
        // Определить ключ для сущности, которую мы ищем.
        EntityKey key = new EntityKey(AutoLotEntities.Cars, CarID, 2222);

        // Извлечь объект автомобиля, изменить его и сохранить.
        Car carToUpdate = (Car)context.GetObjectByKey(key);
        if (carToUpdate != null)
        {
            carToUpdate.Color = Blue;
            context.SaveChanges();
        }
    }
}

```

Приведенный выше метод может показаться немного странным, по крайней мере, до тех пор, пока вы не вспомните, что сущностный объект, возвращенный из `GetObjectByKey()` — это ссылка на существующий объект в поле `ObjectSet<T>`. Таким образом, установка свойств для изменения состояния приводит к изменению того же самого объекта в памяти.

На заметку! Во многом подобно объекту `DataRow` из ADO.NET (см. главу 22), любой потомок `EntityObject` (т.е. все сущностные классы) имеет свойство по имени `EntityState`, используемое контекстом объектов для определения того, был ли элемент модифицирован, удален, отсоединен и т.д. Оно устанавливается без вашего участия при работе с программной моделью; тем не менее, при необходимости его можно изменять вручную.

Запросы с помощью LINQ to Entities

До сих пор вы работали с несколькими простыми методами на контексте объектов и сущностными объектами для выполнения выборки, вставки, обновления и удаления. Это удобно и само по себе; однако гораздо больший эффект от EF можно получить, добавив запросы LINQ. Чтобы использовать LINQ для обновления или удаления записей, создавать объект EntityKey вручную не понадобится. Взгляните на следующую модификацию метода RemoveRecord(), который пока еще *не работает* должным образом:

```
private static void RemoveRecord()
{
    // Найти автомобиль для удаления по первичному ключу.
    using (AutoLotEntities context = new AutoLotEntities())
    {
        // Проверить его наличие.
        var carToDelete = from c in context.Cars where c.CarID == 2222 select c;
        if (carToDelete != null)
        {
            context.DeleteObject(carToDelete);
            context.SaveChanges();
        }
    }
}
```

Этот код скомпилируется, но при попытке вызова метода DeleteObject() возникнет исключение времени выполнения. Причина в том, что этот конкретный запрос LINQ возвращает объект ObjectQuery<T>, а не объект Car. Помните, что запрос LINQ для поиска единственной сущности дает в результате объект ObjectQuery<T>, который представляет запрос, способный доставить необходимые данные. Чтобы выполнить запрос (и вернуть сущность Car), потребуется выполнить такой метод, как FirstOrDefault(), на объекте запроса, как показано в следующем примере:

```
var carToDelete =
    (from c in context.Cars where c.CarID == 2222 select c).FirstOrDefault();
```

Вызов FirstOrDefault() на ObjectQuery<T> позволяет искать нужный элемент; если не окажется экземпляра Car с идентификатором 2222, будет получено стандартное значение — null.

Учитывая, что вы уже имели дело с многими выражениями LINQ в главе 13, уместно привести несколько дополнительных примеров запросов LINQ:

```
private static void FunWithLINQQueries()
{
    using (AutoLotEntities context = new AutoLotEntities())
    {
        // Получить проекцию новых данных.
        var colorsMakes = from item in context.Cars select
            new { item.Color, item.Make };
        foreach (var item in colorsMakes)
        {
            Console.WriteLine(item);
        }

        // Получить только элементы с CarID < 1000
        var idsLessThan1000 = from item in context.Cars
            where item.CarID < 1000 select item;
        foreach (var item in idsLessThan1000)
        {
            Console.WriteLine(item);
        }
    }
}
```

Хотя синтаксис этих запросов достаточно прост, помните, что при каждом применении запроса LINQ к контексту объектов происходит обращение к базе данных! Когда нужно получить независимую копию данных, которая может быть целью новых запросов LINQ, пригодятся (среди прочих) расширяющие методы `ToList<T>()`, `ToArray<T>()` или `ToDictionary<K, V>()`. Ниже показан измененный предыдущий метод, который выполняет эквивалент оператора `SELECT *`, кеширует сущности в виде массива и манипулирует данными массива с помощью LINQ to Objects:

```
using (AutoLotEntities context = new AutoLotEntities())
{
    // Получить все данные из таблицы Inventory.
    // Можно было бы также записать следующий код:
    // var allData = (from item in context.Cars select item).ToArray();
    var allData = context.Cars.ToArray();

    // Получить проекцию новых данных.
    var colorsMakes = from item in allData select new { item.Color, item.Make };

    // Получить только элементы с CarID < 1000.
    var idsLessThan1000 = from item in allData where
                           item.CarID < 1000 select item;
}
```

Работать с LINQ to Entities намного интереснее, когда модель EDM содержит несколько взаимосвязанных таблиц. Ниже будут продемонстрированы некоторые примеры, иллюстрирующие сказанное; а сейчас давайте завершим текущий пример, взглянув на два других способа взаимодействия с контекстом объектов.

Запросы с помощью Entity SQL

В большинстве случаев запросы к `ObjectSet<T>` производятся с помощью LINQ. Клиент сущности преобразует запрос LINQ в соответствующий оператор SQL и передает его на обработку базе данных. В случаях, когда необходим больший контроль над формированием запроса, можно воспользоваться Entity SQL.

Entity SQL — это SQL-подобный язык, который может применяться к сущностям. Хотя формат операторов Entity SQL подобен традиционному SQL, все же они не идентичны. Entity SQL обладает уникальным синтаксисом, т.к. имеет дело с запросом, а не с физической базой данных. Подобно запросу LINQ to Entities, запрос Entity SQL используется для передачи “реального” SQL-запроса базе данных.

Подробные сведения о командах Entity SQL можно найти в документации .NET Framework 4.5 SDK, а ниже представлен один полезный пример. Взгляните на следующий метод, где строится запрос Entity SQL, который находит все автомобили черного цвета в коллекции `ObjectSet<Car>`:

```
private static void FunWithEntitySQL()
{
    using (AutoLotEntities context = new AutoLotEntities())
    {
        // Построить строку, содержащую синтаксис Entity SQL.
        string query = SELECT VALUE car FROM AutoLotEntities.Cars +
                       AS car WHERE car.Color='black';

        // Теперь построить ObjectQuery<T> на основе строки.
        var blackCars = context.CreateQuery<Car>(query);
        foreach (var item in blackCars)
        {
            Console.WriteLine(item);
        }
    }
}
```


Сформатированный оператор Entity SQL передается в качестве аргумента методу `CreateQuery<T>` контекста объектов.

Работа с объектом `EntityDataReader`

При использовании LINQ to Entities или Entity SQL извлеченные данные отображаются на сущностные классы автоматически, благодаря службе клиента сущности. Обычно именно это и нужно; но при желании можно перехватить результирующий набор, прежде чем он превратится в сущностные объекты, и вручную обработать с использованием `EntityDataReader`.

Ниже приведен последний вспомогательный метод для этого примера, в котором применяется несколько членов пространства имен `System.Data.EntityClient` для построения объекта соединения вручную, через объект команды и объект чтения данных. Этот код должен показаться знакомым по главе 21; основное отличие состоит в том, что здесь используется Entity SQL, а не “нормальный” SQL.

```
private static void FunWithEntityDataReader()
{
    // Создать объект соединения на основе файла *.config.
    using (EntityConnection cn = new EntityConnection(name=AutoLotEntities))
    {
        cn.Open();

        // Построить запрос Entity SQL.
        string query = SELECT VALUE car FROM AutoLotEntities.Cars AS car;

        // Создать объект команды.
        using (EntityCommand cmd = cn.CreateCommand())
        {
            cmd.CommandText = query;

            // Получить объект чтения данных и обработать записи.
            using (EntityDataReader dr =
                cmd.ExecuteReader(CommandBehavior.SequentialAccess))
            {
                while (dr.Read())
                {
                    Console.WriteLine(***** RECORD *****);
                    Console.WriteLine(ID: {0}, dr[CarID]);           // идентификатор
                    Console.WriteLine(Make: {0}, dr[Make]);           // производитель
                    Console.WriteLine(Color: {0}, dr[Color]);         // цвет
                    Console.WriteLine(Pet Name: {0}, dr[CarNickname]); // дружественное имя
                    Console.WriteLine();
                }
            }
        }
    }
}
```

Этот начальный пример должен открыть вам длинный путь к пониманию деталей работы с Entity Framework. Как упоминалось ранее, все становится намного интереснее, когда модель EDM содержит взаимосвязанные таблицы, о чем речь пойдет ниже.

Исходный код. Проект `InventoryEDMConsoleApp` доступен в подкаталоге Chapter 23.

Проект AutoLotDAL версии 4, теперь с сущностями

Далее будет показано, как строить модель EDM, которая охватывает значительную часть базы данных AutoLot, включая хранимую процедуру GetPetName. Рекомендуется скопировать проект AutoLotDAL (Version 3), созданный в главе 22, и переименовать копию в AutoLotDAL (Version 4).

Откройте проект AutoLotDAL (Version 4) в Visual Studio и добавьте новый элемент ADO.NET Entity Data Model (Модель сущностных данных ADO.NET) по имени AutoLotDAL_EF.edmx. На третьем шаге мастера понадобится выбрать таблицы Inventory, Orders и Customers (таблица CreditRisks пока не нужна), а также специальную хранимую процедуру (рис. 23.17).

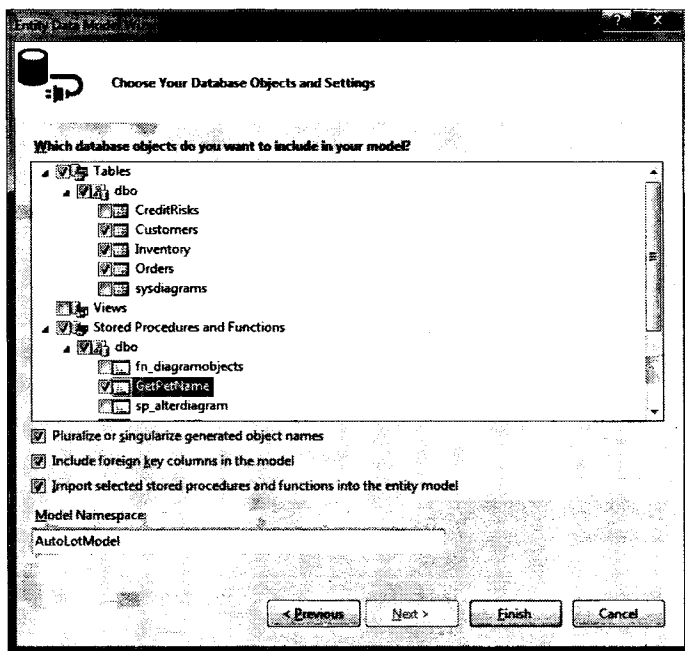


Рис. 23.17. Построение файла *.edmx для большей части базы данных AutoLot

В отличие от первого примера с EDM, на этот раз переименование сущностных классов и их свойств не требуется. В любом случае в окне Model Browser (Браузер модели) вы увидите, что в дополнение к специальной хранимой процедуре, присутствует и учтена каждая сущность (рис. 23.18).

Роль навигационных свойств

Если теперь посмотреть на визуальный конструктор EDM, можно увидеть, что все таблицы учтены, включая новые записи в разделе Navigation Properties (Навигационные свойства) заданного сущностного класса (рис. 23.19).

Как следует из названия, *навигационные свойства* позволяют выражать операции JOIN в программной модели Entity Framework (без необходимости в написании сложных SQL-операторов).

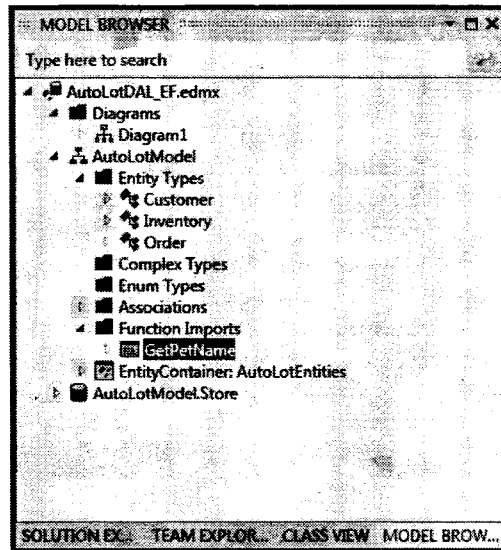


Рис. 23.18. Импортированные данные модели

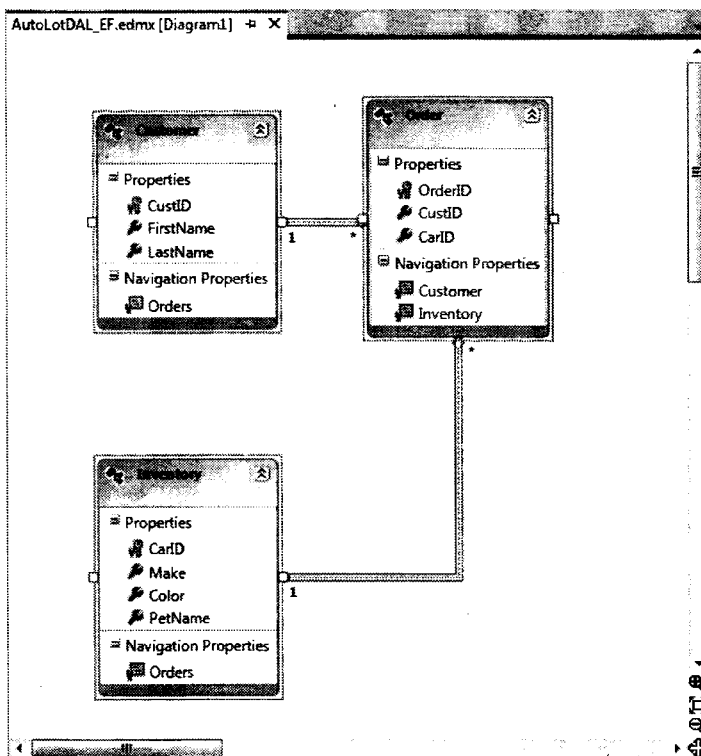


Рис. 23.19. Навигационные свойства

Чтобы учесть эти отношения внешних ключей, каждая сущность в файле *.edmx теперь содержит новые XML-данные, которые показывают, как сущности соединены между собой через данные ключей. Чтобы просмотреть разметку, откройте файл *.edmx в редакторе XML; тем не менее, ту же самую информацию можно видеть и в папке Associations (Ассоциации) окне Model Browser (рис. 23.20).

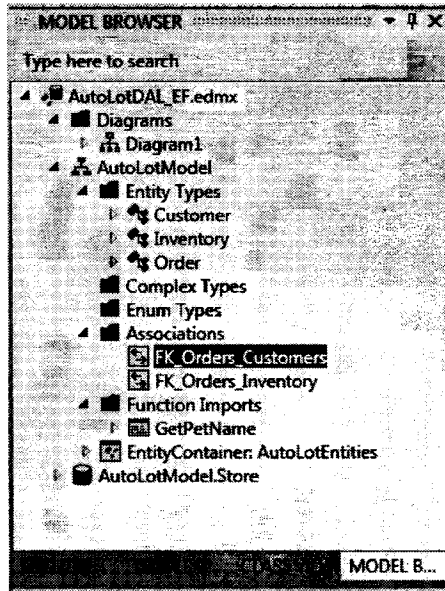


Рис. 23.20. Просмотр отношений между сущностями

При желании номер версии этой новой библиотеки можно изменить на 4.0.0.0 (используя кнопку Assembly Information (Информация о сборке) на вкладке Applications (Приложения) окна Properties (Свойства)). Прежде чем перейти к созданию первого клиентского приложения, скомпилируйте модифицированную сборку AutoLotDAL.dll.

Исходный код. Проект AutoLotDAL (Version 4) доступен в подкаталоге Chapter 23.

Использование навигационных свойств внутри запросов LINQ to Entity

Теперь давайте посмотрим, как использовать навигационные свойства внутри контекста запросов LINQ to Entities (их можно также применять и с Entity SQL, однако эта тема здесь не рассматривается). Перед выполнением привязки данных к графическому пользовательскому интерфейсу Windows Forms мы создадим еще одно консольное приложение по имени AutoLotEDMClient. После создания проекта понадобится установить ссылку на System.Data.Entity.dll и на последнюю версию AutoLotDAL.dll.

Далее откройте файл App.config из проекта AutoLotDAL (Version Four) для просмотра (с помощью пункта меню File⇒Open...⇒File (Файл⇒Открыть...⇒Файл)) и скопируйте строку соединения в текущий конфигурационный файл (в случае отсутствия файла App.config в проекте можно просто включить целый файл через пункт меню Project⇒Add Existing Item (Проект⇒Добавить существующий элемент)). Кроме того, импортируйте пространство имен AutoLotDAL в первоначальный файл кода C#.

Теперь обновим физическую таблицу Orders несколькими новыми записями. В частности, необходимо обеспечить, чтобы один заказчик имел несколько заказов. Используя проводник сервера Visual Studio, добавьте в таблицу Orders одну или две новых записи, чтобы гарантировать наличие у одного заказчика двух или более заказов. Например, на рис. 23.21 заказчик с CustID, равным 4, имеет два ожидающих заказа на автомобили с CarID, равными 1992 и 83.

Inventory: Query(m...qlfexpress.AutoLot)		Orders: Query(manu...qlfexpress.AutoLot)	
	OrderID	CustID	CardID
	1000	1	1000
	1001	2	32
	1002	3	888
	1003	4	1992
	1005	4	83
*	NULL	NULL	NULL

Рис. 23.21. Один заказчик с несколькими заказами

После этого модифицируем класс Program, добавив новый вспомогательный метод (вызываемый в Main()). Этот метод использует навигационные свойства для выбора каждого объекта Inventory по заказу для заданного заказчика:

```
private static void PrintCustomerOrders(string custID)
{
    int id = int.Parse(custID);
    using (AutoLotEntities context = new AutoLotEntities())
    {
        var carsOnOrder = from o in context.Orders
                          where o.CustID == id select o.Inventory;
        Console.WriteLine("\nCustomer has {0} orders pending:", carsOnOrder.Count());
        foreach (var item in carsOnOrder)
        {
            Console.WriteLine("-> {0} {1} named {2}..",
                               item.Color, item.Make, item.PetName);
        }
    }
}
```

Ниже показан вывод, полученный в результате запуска этого приложения (при вызове PrintCustomerOrders() в Main()) указан идентификатор заказчика, равный 4):

```
***** Navigation Properties *****
Please enter customer ID: 4

Customer has 2 orders pending:
-> Pink Saab named Pinky.
-> Rust Ford named Rusty.
```

Здесь в контексте обнаруживается единственная сущность Customer с указанным значением CustID. Найдя заказчика, можно перейти к таблице Inventory для выбора каждого заказанного им автомобиля. Возвращаемым значением запроса LINQ будет перечисление объектов Inventory, которые выводятся на консоль с помощью стандартного цикла foreach.

Вызов хранимой процедуры

Теперь в модели EMD под названием AutoLotDAL присутствует вся необходимая информация для вызова хранимой процедуры GetPetName. Это можно сделать с использованием одного из двух подходов. Ниже показан полный код:

```
private static void CallStoredProc()
{
    using (AutoLotEntities context = new AutoLotEntities())
    {
```

```
// Подход #1.
ObjectParameter input = new ObjectParameter(carID, 83);
ObjectParameter output = new ObjectParameter(petName, typeof(string));

// Вызвать ExecuteFunction на объекте контекста...
context.ExecuteFunction(GetPetName, input, output);

// Подход #2.
// ...либо воспользоваться строго типизированным методом контекста.
context.GetPetName(83, output);
Console.WriteLine(Car #83 is named {0}, output.Value);
}
}
```

Первый подход предусматривает вызов метода `ExecuteFunction()` на контексте объектов. В этом случае хранимая процедура идентифицируется строковым именем, а каждый параметр представлен объектом типа `ObjectParameter`, который находится в пространстве имен `System.Data.Objects` (не забудьте импортировать его в файл кода C#).

Второй подход состоит в использовании строго типизированного имени в контексте объектов. Такой подход значительно проще, потому что входные параметры (наподобие `carID`) можно передавать как типизированные данные, а не как объект `ObjectParameter`.

Исходный код. Проект `AutoLotEDMClient` доступен в подкаталоге `Chapter 23`.

Привязка данных сущностей к графическим пользовательским интерфейсам Windows Forms

В завершения знакомства с ADO.NET Entity Framework давайте рассмотрим простой пример, в котором производится привязка сущностных объектов к графическому пользовательскому интерфейсу Windows Forms. Как упоминалось ранее в этой главе, операции привязки данных будут демонстрироваться в проектах WPF и ASP.NET.

Создайте новое приложение Windows Forms по имени `AutoLotEDM_GUI` и переименуйте его первоначальную форму в `MainForm.cs`. Далее установите ссылки на сборку `System.Data.Entity.dll` и на последнюю версию `AutoLotDAL.dll`. Модифицируйте файл `App.config` этого нового проекта, включив строку соединения из проекта `AutoLotDAL (Version 4)`, и импортируйте пространство имен `AutoLotDAL` в файл кода главной формы.

В визуальном конструкторе форм добавьте элемент управления `DataGridView` и переименуйте его в `gridInventory`. После переименования этого элемента управления выберите встроенный редактор сетки (щелкнув на маленькой стрелке в верхнем правом углу элемента `DataGridView`). В раскрывающемся списке `Choose Data Source` (Выберите источник данных) укажите источник данных проекта (рис. 23.22).

В этом случае необходима привязка не напрямую к базе данных, а к сущностному классу, поэтому выберите в качестве типа источника данных `Object` (рис. 23.23).

На завершающем шаге мастера отметьте таблицу `Inventory` из `AutoLotDAL.dll`, как показано на рис. 23.24 (если она не видна, значит, не была установлена ссылка на эту библиотеку).

После щелчка на кнопке `Finish` (Готово) сетка `DataGridView` отобразит все свойства сущностного класса `Inventory`, включая и навигационные.

В завершение пользовательского интерфейса добавьте на форму элемент управления `Button` и переименуйте его на `btnUpdate`. В этот момент поверхность визуального конструктора должна выглядеть примерно так, как показано на рис. 23.25.

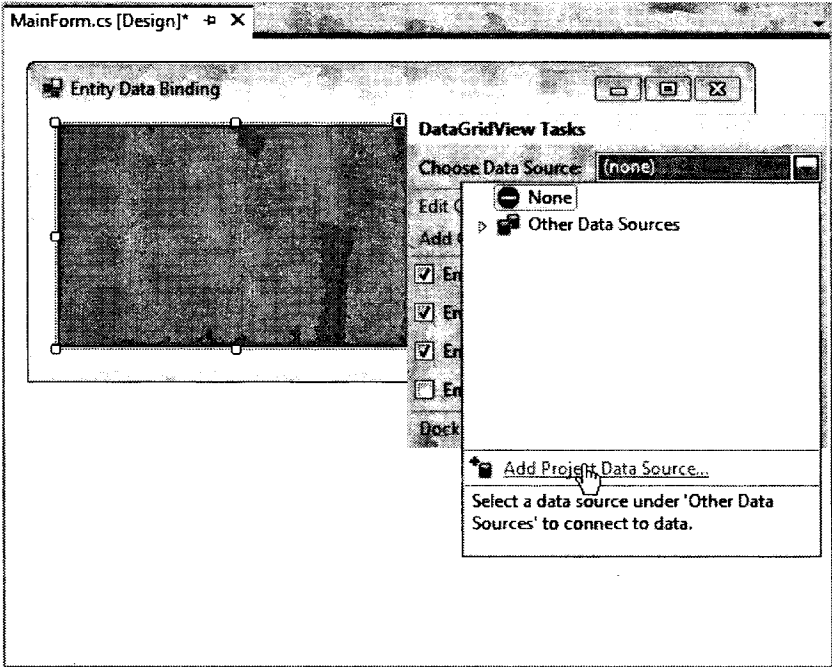


Рис. 23.22. Проектирование элемента управления DataGridView из Windows Forms

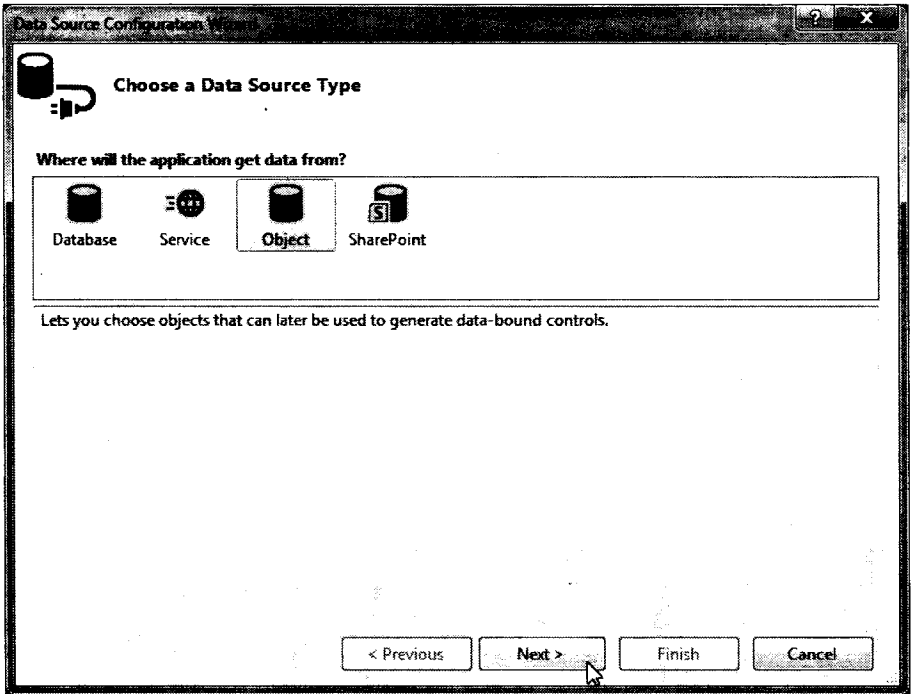


Рис. 23.23. Привязка к строго типизированному объекту

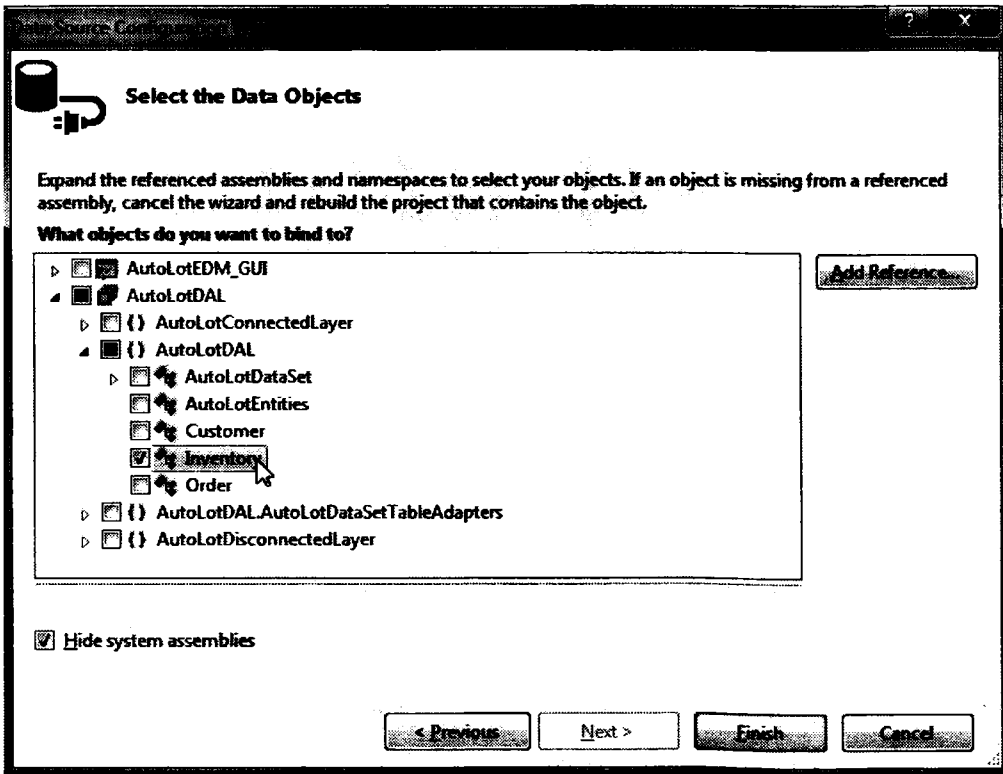


Рис. 23.24. Выбор таблицы Inventory

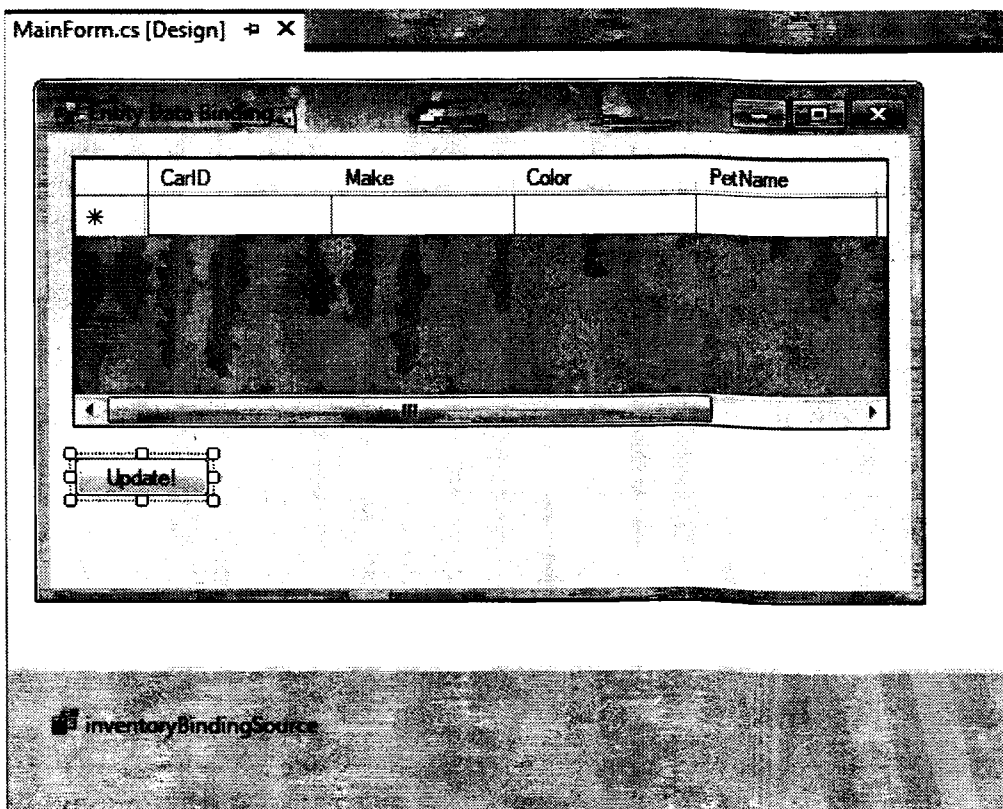


Рис. 23.25. Окончательный пользовательский интерфейс

Добавление кода привязки данных

К этому моменту имеется экранная сетка, которая может отображать любое количество объектов Inventory; однако понадобится еще написать соответствующий код. Благодаря исполняющей среде EF, это очень просто. Начнем с обработки событий FormClosed и Load класса MainForm (используя окно Properties (Свойства)) и события Click элемента управления Button. После этого модифицируем файл кода следующим образом:

```
public partial class MainForm : Form
{
    AutoLotEntities context = new AutoLotEntities();
    public MainForm()
    {
        InitializeComponent();
    }

    private void MainForm_Load(object sender, EventArgs e)
    {
        // Привязать коллекцию ObjectSet<Inventory> к сетке.
        gridInventory.DataSource = context.Inventories;
    }

    private void btnUpdate_Click(object sender, EventArgs e)
    {
        context.SaveChanges();
        MessageBox.Show(Data saved!);
    }

    private void MainForm_FormClosed(object sender, FormClosedEventArgs e)
    {
        context.Dispose();
    }
}
```

Вот и все, что потребуется сделать. Запустив приложение, можно добавлять новые записи в сетку, выбирать строки и удалять их, а также модифицировать существующие строки. Щелчок на кнопке Update (Обновить) приводит к автоматическому обновлению таблицы Inventory, т.к. контекст объектов достаточно интеллектואлен, чтобы сгенерировать все необходимые SQL-операторы для автоматической выборки, обновления, удаления и вставки. Ниже перечислены важные моменты, связанные с предыдущим примером.

- Контекст остается в памяти на протяжении работы приложения.
- Вызов context.Inventories выполняет SQL-код для извлечения в память всех строк из таблицы Inventory.
- Контекст отслеживает все *грязные* сущности, так что ему известно, какой SQL-код следует выполнить при вызове SaveChanges().
- После вызова SaveChanges() сущности снова считаются *чистыми*.

Продолжение изучения API-интерфейсов доступа к данным в .NET

В последних трех главах был представлен обзор трех подходов к манипуляции данными с использованием ADO.NET, в частности — подключенный и автономный уровни, а также инфраструктура Entity Framework. Каждый подход обладает своими достоинс-

твами и многие разрабатываемые вами приложения, скорее всего, будут задействовать разнообразные аспекты каждого из них. Не забывайте, что здесь был представлен только краткий экскурс в темы, которые можно найти в рамках набора технологий ADO.NET. Для получения исчерпывающих сведений рекомендуется обратиться в документацию .NET Framework 4.5 SDK, особенно в ее раздел Data and Modeling (Данные и моделирование), где можно найти многочисленные примеры кода (рис. 23.26).

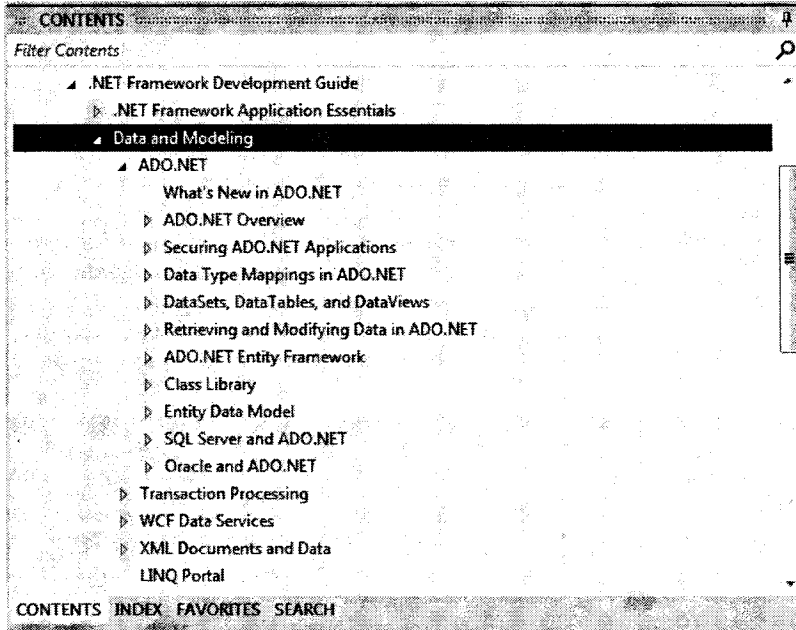


Рис. 23.26. Раздел Data and Modeling документации .NET Framework 4.5 SDK

Резюме

В этой главе рассмотрением роли инфраструктуры Entity Framework завершается формальное исследование программирования для баз данных с использованием ADO.NET. Инфраструктура EF позволяет программировать на уровне концептуальной модели, которая близко отражает предметную область. Хотя можно изменять форму сущностей как угодно, исполняющая среда EF гарантирует, что измененные данные модели будут отображены на корректные данные физической таблицы.

В главе рассказывалось о роли (и составе) файлов *.edmx, а также о том, как их генерировать с помощью IDE-среды Visual Studio. Кроме того, было показано, как отображать хранимые процедуры на функции концептуального уровня, как применять запросы LINQ к объектной модели и как потреблять извлеченные данные на самом низком уровне, используя EntityDataReader. Рассматривалась также и роль Entity SQL.

Завершил главу простой пример привязки сущностных классов к графическому пользовательскому интерфейсу Windows Forms. Когда речь пойдет о Windows Presentation Foundation и ASP.NET, будут продемонстрированы и другие примеры привязки сущностей к приложениям с графическим пользовательским интерфейсом.

Введение в LINQ to XML

Разработчики, пользующиеся платформой .NET, сталкиваются с данными в формате XML повсеместно. Конфигурационные файлы для обычных и веб-приложений хранят информацию в формате XML. Технологии Windows Presentation Foundation, Silverlight и Windows Workflow Foundation используют основанную на XML грамматику (XAML) для представления настольных пользовательских интерфейсов, браузерных пользовательских интерфейсов и рабочих потоков, соответственно. Объекты DataSet из ADO.NET могут легко сохранять (или загружать) данные в формате XML. Даже инфраструктура Windows Communication Foundation хранит многочисленные параметры в формате строк XML.

Хотя XML распространен повсюду, исторически сложилось так, что программирование с использованием XML было утомительным, громоздким и очень сложным, если не знать множество XML-технологий (XPath, XQuery, XSLT, DOM, SAX и т.п.). Еще в самом первом выпуске .NET появилась специальная сборка по имени System.Xml.dll, ориентированная на программирование для XML-документов. В ней находятся множество пространств имен и типов для различных технологий программирования XML, а также несколько специфичных для .NET API-интерфейсов XML, таких как классы XmlReader/XmlWriter.

В наши дни большинство программистов предпочитают взаимодействовать с XML-данными посредством API-интерфейса LINQ to XML. Как будет показано в этой главе, программная модель LINQ to XML позволяет выражать структуру XML-данных в коде, предлагая намного более простой способ создания, манипулирования, загрузки и сохранения XML-данных. Помимо применения LINQ to XML для простого создания XML-документов, с помощью выражений запросов LINQ можно также быстро извлекать информацию из этих документов.

История о двух API-интерфейсах XML

Когда появилась первая версия платформы .NET, программисты могли манипулировать XML-документами, используя типы из сборки System.Xml.dll. Содержащиеся в ней пространства имен и типы позволяли генерировать XML-данные в памяти и сохранять их на диске. Кроме того, сборка System.Xml.dll предоставляла типы, с помощью которых осуществлялась загрузка XML-документа в память, поиск в нем специфических узлов, проверка документа на соответствие заданной схеме и решение других распространенных задач.

Хотя эта исходная библиотека успешно применялась во многих проектах .NET, работа с ее типами была несколько запутанной, поскольку программная модель не имела отношения к структуре самого XML-документа. Например, предположим, что нужно создать файл XML в памяти и сохранить его в файловой системе. В случае использования типов из System.Xml.dll можно написать код, подобный показанному ниже (предварительно понадобится создать новое консольное приложение по имени LinqToXmlFirstLook и импортировать пространство имен System.Xml):

```

private static void BuildXmlDocWithDOM()
{
    // Создать новый документ XML в памяти.
    XmlDocument doc = new XmlDocument();

    // Заполнить документ корневым элементом
    // по имени <Inventory>.
    XmlElement inventory = doc.CreateElement("Inventory");

    // Теперь создать подэлемент по имени <Car>
    // с атрибутом ID.
    XmlElement car = doc.CreateElement("Car");
    car.SetAttribute("ID", "1000");

    // Построить данные внутри элемента <Car>.
    XmlElement name = doc.CreateElement("PetName");
    name.InnerText = "Jimbo";
    XmlElement color = doc.CreateElement("Color");
    color.InnerText = "Red";
    XmlElement make = doc.CreateElement("Make");
    make.InnerText = "Ford";

    // Добавить <PetName>, <Color> и <Make> в элемент <Car>.
    car.AppendChild(name);
    car.AppendChild(color);
    car.AppendChild(make);

    // Добавить элемент <Car> к элементу <Inventory>.
    inventory.AppendChild(car);

    // Вставить завершенный XML в объект XmlDocument
    // и сохранить в файле.
    doc.AppendChild(inventory);
    doc.Save("Inventory.xml");
}

```

После вызова этого метода созданный им файл `Inventory.xml` (находящийся в папке `bin\Debug`) будет содержать следующие данные:

```

<Inventory>
  <Car ID="1000">
    <PetName>Jimbo</PetName>
    <Color>Red</Color>
    <Make>Ford</Make>
  </Car>
</Inventory>

```

Хотя этот метод работает ожидаемым образом, нужно сделать несколько замечаний. Программная модель `System.Xml.dll` — это реализация Microsoft спецификации W3C DOM (Document Object Model — объектная модель документов). Согласно этой модели, документ XML строится снизу вверх. Сначала создается документ, затем элементы и в конце элементы добавляются в документ. Чтобы выразить это в коде, потребуется написать довольно много вызовов функций из классов `XmlDocument` и `XmlElement` (помимо прочих).

В приведенном примере для построения очень простого XML-документа понадобилось 16 строк кода (без учета комментариев). Создание более сложных документов с помощью сборки `System.Xml.dll` требует написания гораздо большего объема кода. Хотя этот код определенно можно упростить, выполняя построение узлов посредством циклических и условных конструкций, факт остается фактом — тело кода имеет минимум визуального отражения финального дерева XML.

Интерфейс LINQ to XML как лучшая модель DOM

API-интерфейс LINQ to XML предлагает альтернативный способ построения, манипулирования и опроса XML-документов, который использует намного более функциональный подход, чем модель DOM из System.Xml. Вместо построения XML-документа за счет индивидуального создания элементов и обновления дерева XML через набор вызовов функций, код пишется сверху вниз:

```
private static void BuildXmlDocWithLINQToXml()
{
    // Создать XML-документ в более "функциональной" манере.
    XElement doc =
        new XElement("Inventory",
            new XElement("Car", new XAttribute("ID", "1000"),
                new XElement("PetName", "Jimbo"),
                new XElement("Color", "Red"),
                new XElement("Make", "Ford")
            )
        );
    // Сохранить документ в файл.
    doc.Save("InventoryWithLINQ.xml");
}
```

Здесь используется новый набор типов из пространства имен System.Xml.Linq, а имен — XElement и XAttribute. В результате вызова метода BuildXmlDocWithLINQToXml() получаются те же данные XML, но на этот раз с гораздо меньшими усилиями. Обратите внимание, что благодаря тщательно выстроенным отступам, исходный код теперь имеет ту же структуру, что и результирующий XML-документ. Это очень удобно и само по себе, но к тому же оцените, насколько компактнее этот код по сравнению с предыдущим примером (экономлено около 10 строк).

Здесь не применяются выражения запросов LINQ, а просто с помощью типов из пространства имен System.Xml.Linq генерируется находящийся в памяти XML-документ, который затем сохраняется в файле. Фактически API-интерфейс LINQ to XML использовался как лучшая модель DOM. Далее в этой главе будет показано, что классы из System.Xml.Linq поддерживают LINQ и могут быть целью для той же разновидности запросов LINQ, которые рассматривались в главе 12.

По мере изучения LINQ to XML, скорее всего, вы начнете отдавать предпочтение этому API-интерфейсу перед первоначальными библиотеками XML платформы .NET. Это не означает, что вы вообще перестанете пользоваться пространством имен из System.Xml.dll, однако случаев выбора System.Xml.dll в новых проектах будет значительно меньше.

Синтаксис литералов Visual Basic как наилучший интерфейс LINQ to XML

Прежде чем приступить к формальному ознакомлению с LINQ to XML с точки зрения языка C#, имеет смысл кратко упомянуть о том, что язык Visual Basic переносит функциональный подход этого API-интерфейса на следующий уровень. В Visual Basic можно определять *литералы XML*, которые позволяют присваивать объекту XElement поток встроенной XML-разметки прямо в коде. Предполагая наличие проекта VB, можно создать следующий метод:

```
Public Class XmlLiteralExample
    Public Sub MakeXmlFileUsingLiterals()
        ' Обратите внимание на возможность встраивания данных XML в XElement.
```

```

Dim doc As XElement =
    <Inventory>
        <Car ID="1000">
            <PetName>Jimbo</PetName>
            <Color>Red</Color>
            <Make>Ford</Make>
        </Car>
    </Inventory>
' Сохранить в файл.
doc.Save("InventoryVBStyle.xml")
End Sub
End Class

```

Когда компилятор VB обрабатывает литерал XML, он отображает данные XML на лежащую в основе корректную объектную модель LINQ to XML. Фактически, при работе с LINQ to XML в проекте VB среда IDE уже понимает, что литеральный синтаксис XML — это сокращенная нотация соответствующего кода. Обратите внимание, что на рис. 24.1 применяется операция точки к конечному дескриптору `</Inventory>` и видны те же самые члены, которые отображаются после применения операции точки к строго типизированному `XElement`.

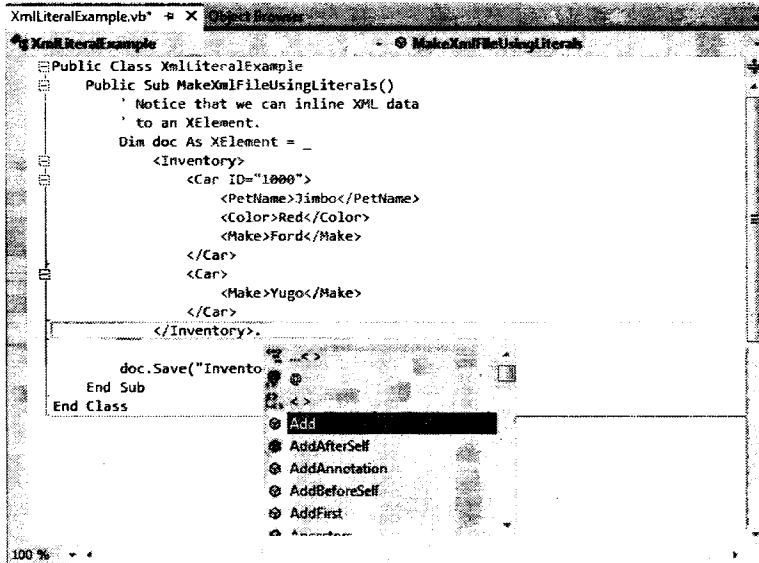


Рис. 24.1. Литеральный синтаксис XML в VB — это сокращенная нотация работы с объектной моделью LINQ to XML

Хотя книга посвящена языку программирования C#, некоторые разработчики считают, что поддержка XML в VB является непревзойденной. Даже если вы из тех, кто не может представить себе работу с языком из семейства BASIC, все равно рекомендуется изучить литеральный синтаксис VB в документации .NET Framework 4.5 SDK. Все операции манипуляций с данными XML можно вынести в отдельную сборку *.dll, так что вполне допускается применять для этого VB!

Члены пространства имен `System.Xml.Linq`

Несколько неожиданно, но в основной сборке LINQ to XML (`System.Xml.Linq.dll`) определено совсем небольшое количество типов в трех разных пространствах имен: `System.Xml.Linq`, `System.Xml.Schema` и `System.Xml.XPath` (рис. 24.2).

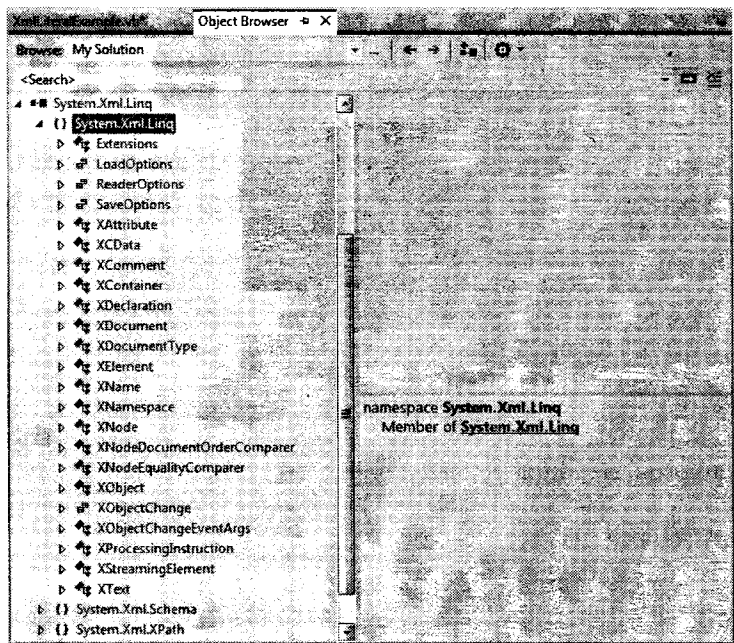


Рис. 24.2. Пространства имен System.Xml.Linq.dll

Основное пространство имен System.Xml.Linq содержит легко управляемый набор классов, представляющих различные аспекты документа XML (элементы и атрибуты, пространства имен XML, комментарии XML, инструкции обработки и т.п.). В табл. 24.1 описаны избранные члены System.Xml.Linq.

Таблица 24.1. Избранные члены пространства имен System.Xml.Linq

Член	Описание
XAttribute	Представляет XML-атрибут заданного элемента XML
XCData	Представляет раздел CDATA в документе XML. Информация в разделе CDATA представляет данные документа XML, которые должны быть включены, но не отвечают правилам грамматики XML (например, код сценария)
XComment	Представляет комментарий XML
XDeclaration	Представляет открывающее объявление документа XML
XDocument	Представляет целиком весь документ XML
XElement	Представляет заданный элемент внутри документа XML, включая корневой
XName	Представляет имя XML-элемента или XML-атрибута
XNamespace	Представляет пространство имен XML
XNode	Представляет абстрактную концепцию узла (элемент, комментарий, тип документа, инструкция обработки или текстовый узел) в дереве XML
XProcessingInstruction	Представляет инструкцию обработки XML
XStreamingElement	Представляет элементы в дереве XML, которые поддерживают отложенный потоковый вывод

На рис. 24.3 показана цепочка наследования ключевых классов.

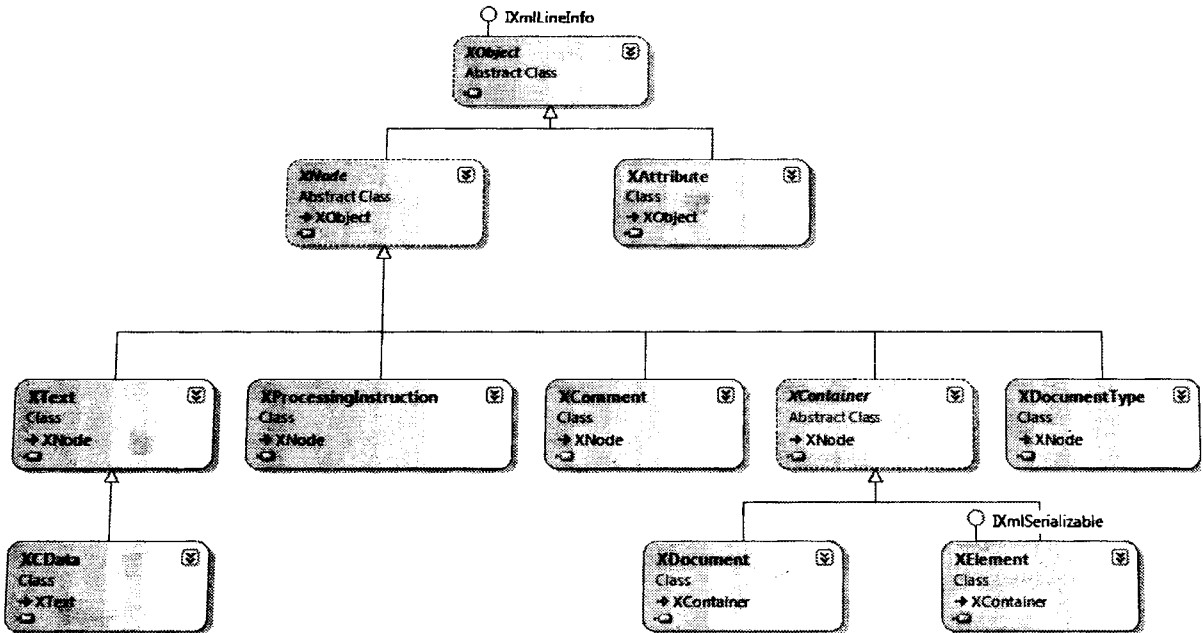


Рис. 24.3. Иерархия основных классов LINQ to XML

Осевые методы LINQ to XML

В дополнение к классам X^* в пространстве имен `System.Xml.Linq` определен класс по имени `Extensions`. В этом классе определен набор расширяющих методов, которые обычно расширяют `IEnumerable<T>`, где T — некоторый потомок `XNode` или `XContainer`. В табл. 24.2 описаны важные расширяющие методы, о которых следует знать (они очень удобны при работе с запросами LINQ).

Таблица 24.2. Избранные члены класса LINQ to XML

Член	Описание
<code>Ancestor<T>()</code>	Возвращает отфильтрованную коллекцию элементов, которая содержит предков каждого узла в исходной коллекции
<code>Attributes()</code>	Возвращает отфильтрованную коллекцию атрибутов каждого элемента в исходной коллекции
<code>DescendantNodes<T>()</code>	Возвращает коллекцию узлов-потомков каждого документа и элемента в исходной коллекции
<code>Descendants<T>()</code>	Возвращает отфильтрованную коллекцию элементов, которая содержит элементы-потомки каждого элемента и документа в исходной коллекции
<code>Elements<T>()</code>	Возвращает коллекцию дочерних элементов каждого элемента и документа в исходной коллекции
<code>Nodes<T>()</code>	Возвращает коллекцию дочерних узлов каждого документа и элемента в исходной коллекции
<code>Remove()</code>	Удаляет каждый атрибут исходной коллекции из родительского элемента
<code>Remove<T>()</code>	Удаляет все вхождения заданного узла в исходной коллекции

Как можно понять из названий, эти методы позволяют опрашивать загруженное дерево XML в поисках элементов, атрибутов и их значений. Все вместе такие методы называются *осевыми методами* или просто *осями*. Эти методы можно применять непосредственно к частям дерева или узлам либо использовать их для построения более сложных запросов LINQ.

На заметку! Абстрактный класс `XContainer` поддерживает множество методов, которые имеют имена, идентичные членам класса `Extensions`. Класс `XContainer` является родительским как для `XElement`, так и для `XDocument`, и потому оба класса поддерживают общую функциональность.

Примеры использования этих осевых методов будут встречаться на протяжении оставшейся части главы. А сейчас рассмотрим краткий пример:

```
private static void DeleteNodeFromDoc()
{
    XElement doc =
        new XElement("Inventory",
            new XElement("Car", new XAttribute("ID", "1000"),
                new XElement("PetName", "Jimbo"),
                new XElement("Color", "Red"),
                new XElement("Make", "Ford")
            )
        );
    // Удалить элемент PetName из дерева.
    doc.Descendants("PetName").Remove();
    Console.WriteLine(doc);
}
```

В результате вызова этого метода получается следующее "усеченное" дерево XML:

```
<Inventory>
  <Car ID="1000">
    <Color>Red</Color>
    <Make>Ford</Make>
  </Car>
</Inventory>
```

Избыточность `XName` (и `XNamespace`)

Если посмотреть на сигнатуру осевых методов LINQ to XML (или идентично именованных членов `XContainer`), можно заметить, что обычно они требуют указания того, что определяется как объект `XName`. Взгляните на сигнатуру метода `Descendants()`, определенного в `XContainer`:

```
public IEnumerable<XElement> Descendants(XName name)
```

Класс `XName` является "избыточным" в том смысле, что он никогда не будет напрямую использоваться в коде. Фактически, поскольку этот класс не имеет открытого конструктора, объект типа `XName` создавать нельзя:

```
// Ошибка! Объекты XName создавать невозможно!
doc.Descendants(new Xname("PetName")).Remove();
```

Просмотрев формальное определение `XName`, вы обнаружите, что в этом классе определена специальная операция неявного преобразования (специальные операции преобразования рассматривались в главе 11), которая отобразит простой тип `System.String` на правильный объект `XName`:

```
// На самом деле "за кулисами" создается XName!
doc.Descendants("PetName").Remove();
```

На заметку! Класс `XNamespace` также поддерживает аналогичное неявное преобразование строк.

Важно отметить, что при работе с осевыми методами можно применять текстовые значения для представления имен элементов или атрибутов и позволять API-интерфейсу LINQ to XML отображать строковые данные на необходимые типы объектов.

Исходный код. Проект `LinqToXmlFirstLook` доступен в подкаталоге Chapter 24.

Работа с XElement и XDocument

Продолжим исследование LINQ to XML в новом консольном приложении по имени `ConstructingXmlDocs`. После создания проекта импортируйте пространство имен `System.Xml.Linq` в начальный файл кода. Как уже было показано, `XDocument` представляет полный XML-документ в программной модели LINQ to XML, поскольку он может использоваться для определения корневого элемента, всех содержащихся в нем элементов, инструкций обработки и объявлений XML. Ниже показан еще один пример построения данных XML с применением `XDocument`:

```
static void CreateFullXDocument()
{
    XDocument inventoryDoc =
        new XDocument(
            new XDeclaration("1.0", "utf-8", "yes"),
            new XComment("Current Inventory of cars!"),
            new XProcessingInstruction("xml-stylesheet",
                "href='MyStyles.css' title='Compact' type='text/css'"),
            new XElement("Inventory",
                new XElement("Car", new XAttribute("ID", "1"),
                    new XElement("Color", "Green"),
                    new XElement("Make", "BMW"),
                    new XElement("PetName", "Stan")
                ),
                new XElement("Car", new XAttribute("ID", "2"),
                    new XElement("Color", "Pink"),
                    new XElement("Make", "Yugo"),
                    new XElement("PetName", "Melvin")
                )
            )
        );
    // Сохранить на диск.
    inventoryDoc.Save("SimpleInventory.xml");
}
```

Обратите внимание, что конструктор объекта `XDocument` на самом деле представляет собой дерево дополнительных объектов LINQ to XML. Вызываемый здесь конструктор принимает в качестве первого параметра `XDeclaration`, за которым следует параметр-массив объектов (вспомните, что параметры-массивы позволяют передавать разделенные запятыми списки аргументов, которые “за кулисами” упаковываются в массив):

```
public XDocument(System.Xml.Linq.XDeclaration declaration,
    params object[] content)
```

После вызова этого метода в `Main()` в файл `SimpleInventory.xml` будут записаны следующие данные:

```

<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<!--Current Inventory of cars!-->
<?xml-stylesheet href='MyStyles.css' title='Compact' type='text/css'?>
<Inventory>
  <Car ID="1">
    <Color>Green</Color>
    <Make>BMW</Make>
    <PetName>Stan</PetName>
  </Car>
  <Car ID="2">
    <Color>Pink</Color>
    <Make>Yugo</Make>
    <PetName>Melvin</PetName>
  </Car>
</Inventory>

```

Как выясняется, стандартное XML-объявление для любого XDocument предусматривает использование кодировки utf-8, версии XML 1.0 и автономного документа. Таким образом, можно полностью удалить создание объекта XDeclaration и получить те же самые данные; учитывая, что почти любой документ требует одного и того же объявления, применять XDeclaration приходится нечасто.

Если определять инструкции обработки или специальное объявление XML не нужно, можно вообще избежать использования XDocument и просто применять XElement. Помните, что XElement может использоваться для представления корневого элемента документа XML и всех подобъектов. Итак, сгенерировать прокомментированный список складских запасов можно следующим образом:

```

static void CreateRootAndChildren()
{
    XElement inventoryDoc =
        new XElement("Inventory",
            new XComment("Current Inventory of cars!"),
            new XElement("Car", new XAttribute("ID", "1"),
                new XElement("Color", "Green"),
                new XElement("Make", "BMW"),
                new XElement("PetName", "Stan")
            ),
            new XElement("Car", new XAttribute("ID", "2"),
                new XElement("Color", "Pink"),
                new XElement("Make", "Yugo"),
                new XElement("PetName", "Melvin")
            )
        );
    // Сохранить на диск.
    inventoryDoc.Save("SimpleInventory.xml");
}

```

Вывод будет более или менее идентичным, исключая инструкцию обработки для гипотетической таблицы стиля:

```

<?xml version="1.0" encoding="utf-8"?>
<Inventory>
  <!--Current Inventory of cars!-->
  <Car ID="1">
    <Color>Green</Color>
    <Make>BMW</Make>
    <PetName>Stan</PetName>
  </Car>

```

```

<Car ID="2">
  <Color>Pink</Color>
  <Make>Yugo</Make>
  <PetName>Melvin</PetName>
</Car>
</Inventory>

```

Генерация документов из массивов и контейнеров

До сих пор XML-документы строились с использованием жестко закодированных значений для конструктора. Но гораздо чаще требуется генерировать XElement (или XDocument), читая данные из массивов, объектов ADO.NET, файлов данных и т.п. Один из способов отображения данных из памяти на новый XElement предусматривает применение стандартных циклов for для перемещения данных в объектную модель LINQ to XML. Хотя это определенно возможно, проще будет встроить запрос LINQ в конструкцию XElement непосредственно.

Предположим, что имеется анонимный массив анонимных классов (просто чтобы сократить объем кода в этом примере; подойдет также любой массив, List<T> или другой контейнер). Отобразить эти данные на XElement можно было бы следующим образом:

```

static void MakeXElementFromArray()
{
    // Создать анонимный массив анонимных типов.
    var people = new[] {
        new { FirstName = "Mandy", Age = 32 },
        new { FirstName = "Andrew", Age = 40 },
        new { FirstName = "Dave", Age = 41 },
        new { FirstName = "Sara", Age = 31 }
    };
    XElement peopleDoc =
        new XElement("People",
            from c in people select new XElement("Person", new XAttribute("Age", c.Age),
                new XElement("FirstName", c.FirstName))
        );
    Console.WriteLine(peopleDoc);
}

```

Здесь объект peopleDoc определяет корневой элемент <People> с результатами запроса LINQ. Этот запрос LINQ создает новые объекты XElement на основе каждого элемента в массиве people. Если встроенный запрос покажется трудно воспринимаемым, можете разделить этот процесс на отдельные шаги следующим образом:

```

static void MakeXElementFromArray()
{
    // Создать анонимный массив анонимных типов.
    var people = new[] {
        new { FirstName = "Mandy", Age = 32 },
        new { FirstName = "Andrew", Age = 40 },
        new { FirstName = "Dave", Age = 41 },
        new { FirstName = "Sara", Age = 31 }
    };
    var arrayDataAsXElements = from c in people
        select
            new XElement("Person",
                new XAttribute("Age", c.Age),
                new XElement("FirstName", c.FirstName));
    XElement peopleDoc = new XElement("People", arrayDataAsXElements);
    Console.WriteLine(peopleDoc);
}

```

В любом случае вывод будет выглядеть следующим образом:

```
<People>
  <Person Age="32">
    <FirstName>Mandy</FirstName>
  </Person>
  <Person Age="40">
    <FirstName>Andrew</FirstName>
  </Person>
  <Person Age="41">
    <FirstName>Dave</FirstName>
  </Person>
  <Person Age="31">
    <FirstName>Sara</FirstName>
  </Person>
</People>
```

Загрузка и разбор XML-содержимого

Типы `XElement` и `XDocument` поддерживают методы `Load()` и `Parse()`, которые позволяют наполнить объектную модель XML из объектов `string`, содержащих XML-данные, либо из внешних XML-файлов. Рассмотрим следующий метод, в котором иллюстрируются оба подхода:

```
static void ParseAndLoadExistingXml()
{
    // Построить XElement из строки.
    string myElement =
        @"<Car ID ='3'>
          <Color>Yellow</Color>
          <Make>Yugo</Make>
        </Car>";
    XElement newElement = XElement.Parse(myElement);
    Console.WriteLine(newElement);
    Console.WriteLine();

    // Загрузить файл SimpleInventory.xml.
    XDocument myDoc = XDocument.Load("SimpleInventory.xml");
    Console.WriteLine(myDoc);
}
```

Исходный код. Проект `ConstructingXmlDocs` доступен в подкаталоге `Chapter 24`.

Манипулирование XML-документом в памяти

К настоящему моменту были продемонстрированы разные способы использования `Linq to XML` для создания, сохранения, разбора и загрузки данных XML. Следующий аспект `Linq to XML`, с которым нужно ознакомиться — это навигация по документу для нахождения и изменения определенных элементов дерева с применением запросов `Linq` и осевых методов `Linq to XML`.

Для этого построим приложение `Windows Forms`, которое будет отображать данные из XML-документа, сохраненного на жестком диске. Графический пользовательский интерфейс позволит вводить данные для нового узла, который будет добавлен к тому же XML-документу. Наконец, пользователю будет предоставлено несколько способов выполнения поиска в документе с помощью ряда запросов `Linq`.

На заметку! Учитывая, что некоторые запросы LINQ уже строились в главе 12, здесь они повторяться не будут. В разделе “Запрос к XML-деревьям” документации .NET Framework 4.5 SDK можно посмотреть дополнительные специфические примеры LINQ to XML.

Построение пользовательского интерфейса приложения LINQ to XML

Создадим приложение Windows Forms под названием `LinqToXmlWinApp` и изменим имя первоначального файла `Form1.cs` на `MainForm.cs` (в проводнике решения). Графический пользовательский интерфейс этого приложения довольно прост. В левой части окна находится элемент управления `TextBox` (по имени `txtInventory`), свойство `Multiline` которого установлено в `true`, а свойство `ScrollBars` — в `Both`.

Кроме того, имеется группа простых элементов управления `TextBox` (`txtMake`, `txtColor` и `txtPetName`) и элемент `Button` (`btnAddNewItem`), щелчок на котором приводит к добавлению нового элемента к XML-документу. Наконец, есть еще одна группа элементов управления (`TextBox` по имени `txtMakeToLookUp` и еще один элемент `Button` с именем `btnLookUpColors`), которая позволяет запросить из XML-документа набор указанных узлов. Возможная компоновка представлена на рис. 24.4.

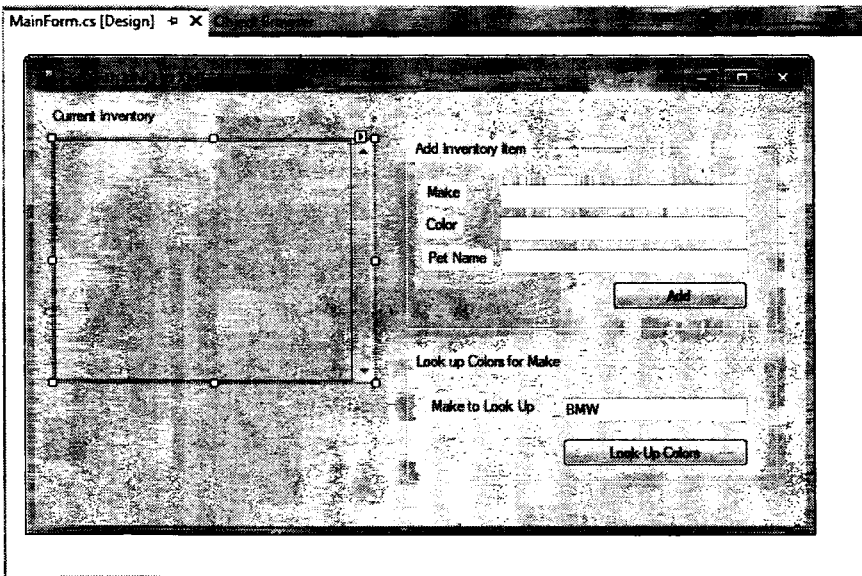


Рис. 24.4. Графический пользовательский интерфейс приложения LINQ to XML

Потребуется обработать событие `Click` каждой кнопки для генерации методов обработки событий, а также обработать событие `Load` самой формы. Чуть ниже мы займемся этим.

Импорт файла `Inventory.xml`

В состав кода примеров для этой главы включен файл `Inventory.xml`, в котором имеется набор сущностей внутри корневого элемента `<Inventory>`. Импортируйте этот файл в проект, выбрав пункт меню `Project`⇒`Add Existing Item` (Проект⇒Добавить существующий элемент). Если вы посмотрите на данные, то увидите, что корневым элементом определяет набор элементов `<Car>`, каждый из которых определен примерно так:

```
<Car carID ="0">
  <Make>Ford</Make>
  <Color>Blue</Color>
  <PetName>Chuck</PetName>
</Car>
```

Прежде чем продолжить, выберите этот файл в Solution Explorer и затем в окне Properties (Свойства) установите свойство Copy to Output Directory (Копировать в выходной каталог) в Copy Always (Копировать всегда). Это гарантирует, что после компиляции приложения данные будут размещены в папке bin\Debug.

Определение вспомогательного класса LINQ to XML

Чтобы изолировать данные LINQ to XML, добавьте в проект новый класс по имени `LinqToXmlObjectModel`. В этом классе будет определен набор статических методов, инкапсулирующих некоторую логику LINQ to XML. Для начала определите метод, возвращающий заполненный `XDocument` на основе содержимого файла `Inventory.xml` (не забудьте импортировать в этот новый файл пространства имен `System.Xml.Linq` и `System.Windows.Forms`):

```
public static XDocument GetXmlInventory()
{
    try
    {
        XDocument inventoryDoc = XDocument.Load("Inventory.xml");
        return inventoryDoc;
    }
    catch (System.IO.FileNotFoundException ex)
    {
        MessageBox.Show(ex.Message);
        return null;
    }
}
```

Метод `InsertNewElement()`, код которого показан ниже, принимает значения элементов управления `TextBox` из раздела `Add Inventory Item` (Добавить элемент на склад), чтобы поместить новый узел внутрь элемента `<Inventory>`, используя осевой метод `Descendants()`. После этого документ будет сохранен.

```
public static void InsertNewElement(string make, string color, string petName)
{
    // Загрузить текущий документ.
    XDocument inventoryDoc = XDocument.Load("Inventory.xml");

    // Сгенерировать случайное число для идентификатора.
    Random r = new Random();

    // Создать новый объект XElement на основе входных параметров.
    XElement newElement = new XElement("Car", new XAttribute("ID", r.Next(50000)),
        new XElement("Color", color),
        new XElement("Make", make),
        new XElement("PetName", petName));

    // Добавить к объекту XDocument в памяти.
    inventoryDoc.Descendants("Inventory").First().Add(newElement);

    // Сохранить изменения на диске.
    inventoryDoc.Save("Inventory.xml");
}
```

Последний метод, `LookUpColorsForMake()`, принимает данные из последнего элемента `TextBox` для построения строки, которая содержит цвета, используемые определенным изготовителем, с помощью запроса LINQ. Взгляните на следующую реализацию:

```
public static void LookUpColorsForMake(string make)
{
    // Загрузить текущий документ.
    XDocument inventoryDoc = XDocument.Load("Inventory.xml");

    // Найти цвета заданного изготовителя.
    var makeInfo = from car in inventoryDoc.Descendants("Car")
        where (string)car.Element("Make") == make
        select car.Element("Color").Value;

    // Построить строку, представляющую каждый цвет.
    string data = string.Empty;
    foreach (var item in makeInfo.Distinct())
    {
        data += string.Format("- {0}\n", item);
    }

    // Показать цвета.
    MessageBox.Show(data, string.Format("{0} colors:", make));
}
```

Связывание пользовательского интерфейса и вспомогательного класса

Теперь все, что осталось сделать — наполнить кодом обработчики событий. Задача сводится к простым вызовам статических вспомогательных методов, как показано ниже:

```
public partial class MainForm : Form
{
    public MainForm()
    {
        InitializeComponent();
    }

    private void MainForm_Load(object sender, EventArgs e)
    {
        // Отобразить текущий XML-документ склада в элементе управления TextBox.
        txtInventory.Text = LinqToXmlObjectModel.GetXmlInventory().ToString();
    }

    private void btnAddNewItem_Click(object sender, EventArgs e)
    {
        // Добавить новый элемент к документу.
        LinqToXmlObjectModel.InsertNewElement(txtMake.Text,
                                                txtColor.Text,
                                                txtPetName.Text);

        // Отобразить текущий XML-документ склада в элементе управления TextBox.
        txtInventory.Text = LinqToXmlObjectModel.GetXmlInventory().ToString();
    }

    private void btnLookUpColors_Click(object sender, EventArgs e)
    {
        LinqToXmlObjectModel.LookUpColorsForMake(txtMakeToLookUp.Text);
    }
}
```

Конечный результат показан на рис. 24.5.

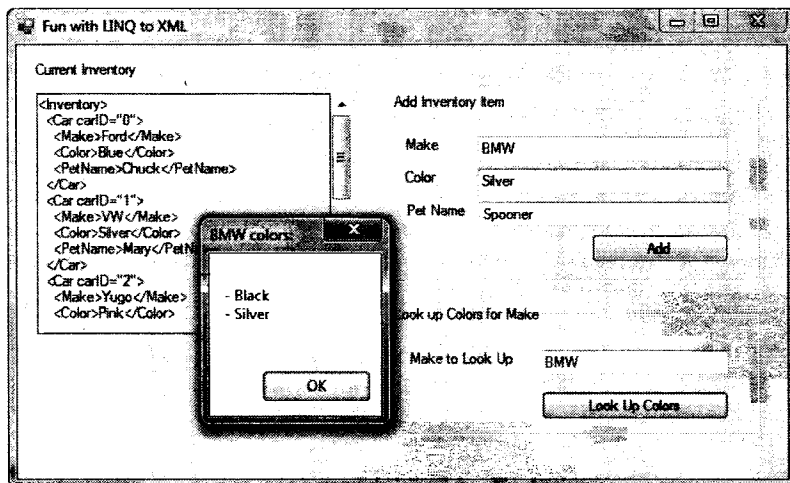


Рис. 24.5. Готовое приложение LINQ to XML

На этом начальное знакомство с LINQ to XML и изучение LINQ завершено. Впервые вы встретились с технологией LINQ в главе 12, где шла речь о LINQ to Objects. В главе 19 приводились различные примеры использования PLINQ, а в главе 23 рассматривалось применение LINQ к объектам сущностей ADO.NET. Теперь вы готовы и должны двигаться дальше. В Microsoft весьма недвусмысленно дали понять, что LINQ будет развиваться вместе с развитием платформы .NET.

Исходный код. Проект `LinqToXmlWinApp` доступен в подкаталоге `Chapter 24`.

Резюме

В этой главе рассматривалась роль LINQ to XML. Как было показано, этот API-интерфейс является альтернативой первоначальной библиотеке для манипуляций XML под названием `System.Xml.dll`, которая поставлялась в составе платформы .NET. С помощью библиотеки `System.Xml.Linq.dll` можно генерировать новые XML-документы, используя подход “сверху вниз”, при котором структура кода очень напоминает конечные данные XML. В этом свете LINQ to XML — лучшая модель DOM. Также было показано, как строить объекты `XDocument` и `XElement` различными путями (разбором, загрузкой из файла, отображением объектов в памяти) и как выполнять навигацию и манипуляции данными с применением запросов LINQ.

Введение в Windows Communication Foundation

В версии .NET 3.0 был представлен API-интерфейс, специально предназначенный для построения распределенных систем — Windows Communication Foundation (WCF). В отличие от других распределенных API-интерфейсов, которые, возможно, приходилось применять в прошлом (например, DCOM, .NET Remoting, веб-службы XML, очереди сообщений), WCF предлагает единую, унифицированную и расширяемую объектную модель для программирования, которая может использоваться для взаимодействия с множеством ранее разрозненных распределенных технологий.

Эта глава начинается с определения потребностей, которые призвана удовлетворить инфраструктура WCF, и рассмотрения задач, которые данный API-интерфейс должен решать, с приведением краткого обзора предшествующих API-интерфейсов распределенных вычислений. После этого рассматриваются службы, предоставляемые WCF, а также основные сборки, пространства имен и типы, которые представляет эта программная модель. На протяжении главы будет построено несколько служб, хостов и клиентов WCF с использованием различных инструментов разработки WCF.

На заметку! В этой главе будет создаваться код, который требует запуска Visual Studio с административными привилегиями (вдобавок вы и сами должны иметь административные привилегии). Чтобы запустить Visual Studio с правами администратора, щелкните правой кнопкой мыши на значке Visual Studio и выберите в контекстном меню пункт **Запуск от имени администратора**.

Собрание API-интерфейсов распределенных вычислений

Исторически сложилось так, что операционная система Windows предоставляла множество API-интерфейсов для построения распределенных систем. Хотя и верно, что под *распределенной системой* большинство понимает систему, состоящую минимум из двух сетевых компьютеров, этот термин в широком смысле может охватывать два исполняемых модуля, которые нуждаются в обмене данными, даже если они функционируют на одной и той же физической машине. Используя такое определение, выбор распределенных API-интерфейсов для решения текущей задачи программирования обычно подразумевает ответ на следующий основополагающий вопрос.

Будет ли эта система использоваться исключительно внутренне, или же доступ к функциональности приложения потребуется и внешним пользователям?

Если вы строите распределенную систему для внутреннего применения, то имеются немалые шансы гарантировать, что каждый подключенный компьютер будет работать под управлением одной и той же операционной системы и использовать одну и ту же программную платформу (например, .NET, COM или Java). Выполнение внутренних систем также означает возможность применения для аутентификации, авторизации и тому подобного существующей системы безопасности. В такой ситуации можно выбрать конкретный распределенный API-интерфейс, который, исходя из соображений производительности, наилучшим образом подходит для существующей операционной системы и программной платформы.

В отличие от этого, при построении системы, которая должна быть доступна другим за пределами ваших стен, вы сталкиваетесь с целым букетом проблем, подлежащих решению. Прежде всего, вряд ли удастся диктовать внешним пользователям, какую операционную систему (или системы) они должны применять, какую программную платформу (или платформы) использовать для построения программного обеспечения и как настраивать параметры безопасности.

Если вы работаете в крупной компании или в университете, где используется множество операционных систем и технологий программирования, то в этом случае даже внутренние приложения неожиданно сталкиваются с теми же сложностями, что и приложения, ориентированные на внешний мир. В любом из этих случаев следует ограничиться наиболее гибким API-интерфейсом для распределенных вычислений, чтобы обеспечить максимальную доступность результирующего приложения.

В зависимости от ответа на этот ключевой вопрос распределенных вычислений, следующей задачей будет выбор конкретного API-интерфейса (либо их набора). В последующих разделах представлен краткий перечень некоторых основных распределенных API-интерфейсов, исторически используемых разработчиками программного обеспечения для Windows. После этого краткого экскурса вы легко сможете оценить удобство Windows Communication Foundation.

На заметку! Чтобы исключить возможные недоразумения, следует указать, что WCF (и включаемые технологии) не имеет ничего общего с построением веб-сайтов на основе HTML. Хотя и верно считать веб-приложения распределенными, в том смысле, что в обмене обычно участвуют две машины, API-интерфейс WCF ориентирован на установку соединений с машинами для разделения функциональности удаленных компонентов, а не для отображения HTML-разметки в веб-браузере. Построение веб-сайтов с помощью платформы .NET будет рассматриваться, начиная с главы 32.

Роль DCOM

До появления платформы .NET основным API-интерфейсом удаленных вычислений в Microsoft была распределенная модель компонентных объектов (Distributed Component Object Model — DCOM). С помощью DCOM можно было строить распределенные системы, используя при этом объекты COM, системный реестр и определенную долю старания. Одно из преимуществ модели DCOM заключалось в том, что она обеспечивала прозрачность расположения компонентов. Говоря просто, это позволяло разрабатывать клиентское программное обеспечение так, что физическое расположение удаленных объектов не должно было жестко кодироваться в приложении. Независимо от того, располагался удаленный объект на той же самой или на другой сетевой машине, кодовая база могла оставаться нейтральной, поскольку действительное местоположение записывалось внешне в системный реестр.

Хотя модель DCOM имела определенный успех, для всех практических нужд это был API-интерфейс, ориентированный на Windows. Сама модель DCOM не представляла собой основы для построения полноценных решений, включающих различные операционные системы (Windows, Unix, Mac) или разделяющих данные между разными архитектурами (COM, Java или CORBA).

На заметку! Предпринимались некоторые попытки переноса DCOM для запуска на различных версиях Unix/Linux, но результат не был блестящим и в конечном итоге превратился в технологический тупик.

В общем и целом, модель DCOM была наилучшим образом приспособлена для разработки внутренних приложений, поскольку передача типов COM за пределы компании влекла за собой дополнительные сложности (брандмауэры и т.п.). С выходом платформы .NET модель DCOM быстро стала устаревшей, и если вам не приходится сопровождать унаследованные системы DCOM, то можете смело поставить крест на этой технологии.

Роль служб COM+/Enterprise Services

Модель DCOM представляла собой лишь немногим более чем способ установки канала взаимодействия между двумя частями программного обеспечения на основе COM. Чтобы заполнить брешь недостающих компонентов для построения полноценных распределенных вычислительных решений, в Microsoft в конечном итоге выпустили сервер транзакций (Microsoft Transaction Server — MTS), который позже был переименован в COM+.

Несмотря на название, COM+ использовали не только программисты COM; эта технология полностью доступна и профессиональным разработчикам приложений для .NET. Со времен выхода первого выпуска платформы .NET библиотеки базовых классов предоставляют пространство имен `System.EnterpriseServices`. С его помощью программисты .NET могут строить управляемые библиотеки, которые устанавливаются в исполняющую среду COM+, получая доступ к тому же набору служб, что и традиционный COM-сервер, поддерживающий COM+. В любом случае, как только поддерживающая COM+ библиотека устанавливается в исполняющей среде COM+, она становится служебным компонентом.

Технология COM+ предлагает множество средств, которые могут использовать служебные компоненты, включая управление транзакциями, управление временем жизни объектов, службы поддержки пулов, систему безопасности на основе ролей, модель слабо связанных событий и т.д. Это было главным преимуществом на то время, учитывая, что большинство распределенных систем требовали одинакового набора служб. Вместо того чтобы заставлять разработчиков кодировать их вручную, технология COM+ представила готовое решение.

Одним из самых неотразимых аспектов COM+ был тот факт, что все эти настройки можно было конфигурировать в декларативной манере, используя административные инструменты. Таким образом, если вы хотели обеспечить мониторинг объекта в транзакционном контексте или отнести его к определенной роли безопасности, то просто должны были правильно отметить нужные флажки.

Хотя службы COM+/Enterprise Services все еще используются и сегодня, данная технология предлагает решение только для Windows. Это решение больше подходит для разработки внутренних приложений либо в качестве серверной службы, опосредованно управляемой более совершенными интерфейсными средствами (например, это может быть публичный веб-сайт, вызывающий служебные компоненты (объекты COM+) в фоновом режиме).

На заметку! Инфраструктура WCF не предоставляет способа для построения служебных компонентов. Однако службы WCF имеют возможность взаимодействовать с существующими объектами COM+. Если необходимо строить служебные компоненты на C#, придется непосредственно работать с пространством имен `System.EnterpriseServices`. Подробные сведения ищите в документации .NET Framework 4.5 SDK.

Роль MSMQ

API-интерфейс MSMQ (Microsoft Message Queuing — очередь сообщений Microsoft) позволяет разработчикам строить распределенные системы, которые нуждаются в надежной доставке сообщений по сети. Хорошо известно, что в любой распределенной системе существует риск отказа сетевого сервера, отключения базы данных или потери соединений по каким-то необъяснимым причинам. Более того, множество приложений должны быть сконструированы так, чтобы данные сообщений, которые невозможно доставить немедленно, сохранялись для последующей доставки (это называется *очередью задержки данных*).

Изначально в Microsoft представили MSMQ как упакованный набор низкоуровневых API-интерфейсов и COM-объектов на основе C. После выхода платформы .NET программисты на C# могут использовать пространство имен `System.Messaging` для привязки к MSMQ и построения программного обеспечения, взаимодействующего с периодически подключающимися приложениями в зависимом режиме.

К слову, уровень COM+, включающий функциональность MSMQ в исполняющую среду (в упрощенном формате), использует технологию, которая называется *Queued Components (QC)*. Этот способ взаимодействия с MSMQ был оформлен в пространство имен `System.EnterpriseServices`, которое упоминалось в предыдущем разделе.

Независимо от того, какая программная модель используется для взаимодействия с исполняющей средой MSMQ, в конечном итоге гарантируется надежная и своевременная доставка сообщений. Подобно COM+, API-интерфейс MSMQ определенно можно считать частью фабрики по построению распределенного программного обеспечения на платформе операционной системы Windows.

Роль .NET Remoting

Как ранее упоминалось, с выходом платформы .NET технология DCOM быстро стала устаревшим API-интерфейсом распределенных систем. Библиотеки базовых классов, обслуживающие уровень .NET Remoting, представлены пространством имен `System.Runtime.Remoting`. Этот API-интерфейс (теперь устаревший) позволяет множеству компьютеров распределять объекты при условии, что они работают в составе приложений на платформе .NET.

API-интерфейсы .NET Remoting предоставили множество очень полезных средств. Наиболее важным было применение конфигурационных файлов на основе XML для декларативного определения внутренних механизмов, используемых клиентским и серверным программным обеспечением. Посредством файлов `*.config` очень просто радикально изменить функциональность распределенной системы, модифицировав содержимое конфигурационных файлов и перезапустив приложение.

К тому же, учитывая факт, что этот API-интерфейс используется только приложениями .NET, можно получить значительный выигрыш в производительности, если данные будут закодированы в компактном двоичном формате, и при определении параметров и возвращаемых значений можно будет применять систему общих типов (*Common Type System — CTS*). Хотя .NET Remoting можно было использовать для построения распределенных систем, охватывающих несколько операционных систем (через платформу

Моно, кратко упомянутую в главе 1), взаимодействие с другими программными архитектурами (такими как Java) еще не было возможным.

Роль веб-служб XML

Каждый из предшествующих API-интерфейсов распределенных вычислений обеспечивал минимальную поддержку (или вообще не обеспечивал таковой) доступа удаленных пользователей к функциональности в *независимой манере*. Когда нужно было предоставить услуги удаленных объектов *любой* операционной системе и *любой* программной модели, веб-службы XML предлагали наиболее простой способ сделать это.

В отличие от традиционных браузерных веб-приложений, веб-служба обеспечивает способ представления функциональности удаленных компонентов через стандартные веб-протоколы. С момента появления начального выпуска .NET программистам была предложена превосходная поддержка построения и использования веб-служб XML через пространство имен `System.Web.Services`. Фактически во многих случаях построение полноценной веб-службы сводится просто к применению атрибута `[WebMethod]` к каждому открытому методу, к которому планируется обеспечить доступ. Более того, Visual Studio позволяет подключаться к удаленным веб-службам с помощью пары щелчков на кнопках.

Веб-службы позволяют разработчикам строить сборки .NET, включающие типы, которые могут быть доступны по простому протоколу HTTP. Кроме того, веб-служба кодирует свои данные в виде простого XML. Учитывая тот факт, что веб-службы базируются на открытых отраслевых стандартах (например, HTTP, XML и SOAP), а не на патентованной системе типов и сетевых форматов (как в случае DCOM или .NET Remoting), они допускают высокую степень взаимодействия и возможности обмена данными. Независимая природа веб-служб XML проиллюстрирована на рис. 25.1.

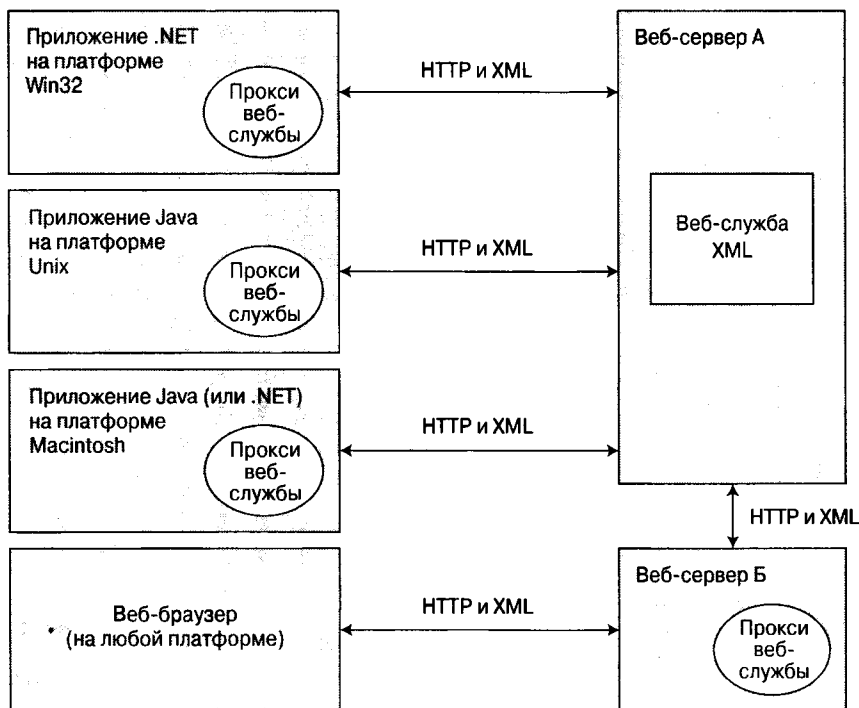


Рис. 25.1. Веб-службы XML обеспечивают высокую степень взаимодействия

Разумеется, ни один распределенный API-интерфейс не является безупречным. Один из потенциальных недостатков веб-служб состоит в том, что они могут страдать от некоторых проблем с производительностью (учитывая использование HTTP и XML для представления данных). Другой недостаток связан с тем, что они могут оказаться не идеальным решением для внутренних приложений, где беспрепятственно можно применять протоколы на основе TCP и двоичное форматирование данных.

Стандарты веб-служб

Еще одна проблема заключается в том, что реализации веб-служб, с которыми мы имели дело ранее, созданные крупными компаниями (Microsoft, IBM и Sun Microsystems), были не на 100% совместимыми между собой. Вполне очевидно, что это было проблемой, учитывая, что основной целью веб-служб является достижение высокой степени взаимодействия между платформами и операционными системами!

Чтобы гарантировать возможность взаимодействия веб-служб, группа под названием World Wide Web Consortium (W3C; www.w3.org) и организация Web Services Interoperability Organization (WS-I; www.ws-i.org) приступили к разработке спецификаций. Эти спецификации были призваны определить, каким образом поставщики программного обеспечения (т.е. IBM, Microsoft и Sun Microsystems) должны строить библиотеки программного обеспечения для разработки веб-служб, чтобы обеспечивать совместимость.

Все эти спецификации получили общее имя WS-* и охватили вопросы безопасности, вложений, описание веб-служб (через язык описания веб-служб WSDL (Web Service Description Language)), политики, форматы SOAP и массу прочих деталей. Вы увидите, что WCF поддерживает многие спецификации WS-*. Как правило, ваши службы WCF будут пользоваться различными спецификациями WS-* на основе выбранных привязок.

На заметку! В дополнение к кратко рассмотренным выше распределенным API-интерфейсам, разработчики могут также применять разнообразные протоколы межпроцессного взаимодействия, такие как именованные каналы и сокеты.

Роль WCF

Широкий массив распределенных технологий затрудняет выбор правильного инструмента для работы. Ситуация еще более усложняется из-за того факта, что функциональность некоторых из этих технологий перекрывается (наиболее заметно в областях транзакций и безопасности).

Даже когда разработчик .NET выбрал технологию, которая кажется правильной для текущей задачи, построение, сопровождение и конфигурирование такого приложения будет в лучшем случае сложным. Каждый API-интерфейс имеет собственную программную модель, собственный уникальный набор инструментов конфигурирования и т.д. Это означает, что до появления WCF было трудно подключать распределенные API-интерфейсы, не написав существенного объема кода для специальной инфраструктуры. Например, если вы строите систему с использованием API-интерфейсов .NET Remoting, а позднее решите, что веб-службы XML являются более подходящим решением, придется полностью перепроектировать кодовую базу.

Инфраструктура WCF — это инструментальный набор распределенных вычислений, который интегрирует все эти ранее независимые технологии распределенной обработки в один согласованный API-интерфейс, представленный в первую очередь пространством имен `System.ServiceModel`. С помощью WCF можно открывать эти службы вызывающим компонентам, применяя для этого широкое разнообразие приемов.

Например, при создании внутреннего приложения, где все подключенные машины работают под управлением Windows, можно использовать различные протоколы TCP для достижения максимально возможной производительности. Те же самые службы также могут быть представлены с применением протоколов HTTP и SOAP, чтобы позволить внешним клиентам пользоваться их функциональностью, независимо от языка программирования или операционной системы.

Учитывая тот факт, что WCF позволяет выбрать правильный протокол для выполнения работы (используя общую программную модель), вы обнаружите, что подключить и запустить низкоуровневые механизмы распределенного приложения достаточно просто. В большинстве случаев это можно делать без перекомпиляции или повторного развертывания клиентского программного обеспечения и службы, поскольку низкоуровневые детали часто задаются в конфигурационных файлах приложения.

Обзор функциональных возможностей WCF

Возможность взаимодействия и интеграция различных API-интерфейсов — это только два важных аспекта WCF. Вдобавок WCF предлагает развитую фабрику программного обеспечения, которая дополняет технологии удаленной разработки, предлагаемые WCF. Ниже приведен список главных средств WCF.

- Поддержка как строго типизированных, так и не типизированных сообщений. Этот подход позволяет приложениям .NET эффективно совместно использовать типы, в то время как программное обеспечение, созданное с помощью других платформ (таких как Java), может потреблять потоки слабо типизированного XML.
- Поддержка нескольких привязок (например, низкоуровневый HTTP, TCP, MSMQ и именованные каналы), позволяющая выбирать наиболее подходящий механизм для транспортировки данных сообщений.
- Поддержка последних спецификаций веб-служб (WS-*).
- Полностью интегрированная модель безопасности, охватывающая как встроенные протоколы безопасности Windows/.NET, так и многочисленные нейтральные технологии защиты, построенные на стандартах веб-служб.
- Поддержка технологий сеансового управления состоянием, а также поддержка односторонних сообщений без состояния.

Каким бы впечатляющим ни был этот список, на самом деле он лишь поверхностно касается функциональности, предоставляемой WCF. Технология WCF также предлагает средства трассировки и протоколирования, счетчики производительности, модель публикации и подписки на события, поддержку транзакций и многое другое.

Обзор архитектуры, ориентированной на службы

Еще одно преимущество инфраструктуры WCF связано с тем, что она базируется на принципах проектирования, установленных *архитектурой, ориентированной на службы* (service-oriented architecture — SOA). Конечно, SOA является модным словечком в отрасли, и подобно многим модным словечкам, может быть определено различными способами. Попросту говоря, SOA — это способ проектирования распределенных систем, где несколько автономных *служб* работают совместно, передавая сообщения через границы (либо сетевых машин, либо двух процессов на одной и той же машине) с использованием четко определенных *интерфейсов*.

В мире WCF эти четко определенные интерфейсы обычно создаются с применением интерфейсных типов CLR (см. главу 9). Однако в более общем смысле интерфейс службы просто описывает набор членов, которые могут быть вызваны внешними компонентами.

Команда разработчиков WCF пользовалась четырьмя принципами проектирования SOA. Хотя данные принципы реализуются автоматически, просто при построении приложения WCF, понимание этих четырех кардинальных правил проектирования SOA может помочь в дальнейшем применении WCF. В последующих разделах приведен краткий обзор этих принципов.

Принцип 1: границы установлены явно

Этот принцип подчеркивает тот факт, что функциональность службы WCF выражается через четко определенные интерфейсы (т.е. описания каждого члена, его параметров и возвращаемых значений). Единственный способ, которым внешний клиент может связаться со службой WCF — через интерфейс, при этом оставаясь в блаженном неведении о деталях ее внутренней реализации.

Принцип 2: службы являются автономными

Когда о службах говорят как об *автономных сущностях*, имеют в виду тот факт, что каждая служба WCF является (насколько возможно) отдельным "островом". Автономная служба должна быть независимой от аспектов, касающихся версии, развертывания и установки. Чтобы помочь в продвижении этого принципа, мы опять возвращаемся к ключевому аспекту программирования на основе интерфейсов. Как только интерфейс внедрен в производство, он никогда не должен изменяться (или вы рискуете разрушить существующие клиенты). Когда требуется расширить функциональность службы WCF, просто напишите новый интерфейс, который моделирует желаемую функциональность.

Принцип 3: службы взаимодействуют через контракт, а не реализацию

Третий принцип — еще один побочный продукт программирования на основе интерфейсов — состоит в том, что реализация деталей службы WCF (на каком языке она написана, как именно выполняет свою работу, и т.п.) не касается вызывающего ее внешнего компонента. Клиенты WCF взаимодействуют со службами исключительно через их открытые интерфейсы.

Принцип 4: совместимость служб основана на политике

Поскольку интерфейсы CLR предоставляют строго типизированные контракты всем клиентам WCF (и также могут быть использованы для генерации соответствующего документа WSDL на основе выбранной привязки), важно понимать, что интерфейсы и WSDL сами по себе недостаточно выразительны, чтобы детализировать аспекты того, что способна делать служба. Учитывая это, SOA позволяет определять *политики*, которые дополнительно проясняют семантику службы (например, ожидаемые требования безопасности, применяемые для общения со службой). Используя эти политики, можно отделять низкоуровневые синтаксические описания службы (открываемые интерфейсы) от семантических деталей их работы и способов их вызова.

WCF: заключение

Этот небольшой исторический экскурс прояснил, почему WCF является предпочтительным подходом для построения распределенных приложений. Строите ли вы внутреннее приложение с использованием протоколов TCP, перемещаете данные между программами на одной и той же машине с применением именованных каналов, или же в общем открываете данные внешнему миру — для всего этого рекомендуется пользоваться API-интерфейсом WCF.

Это не означает невозможность применения в новых разработках исходных пространств имен, связанных с распределенными вычислениями (например,

`System.Runtime.Remoting`, `System.Messaging`, `System.EnterpriseServices` и `System.Web.Services`). В некоторых случаях (в частности, когда нужно строить объекты COM+) это делать придется. Но, так или иначе, если вы использовали эти API-интерфейсы в прошлых проектах, изучение WCF не представит особой сложности. Подобно предшествующим технологиям, в WCF интенсивно применяются конфигурационные XML-файлы, атрибуты .NET и утилиты генерации прокси. Вооружившись всеми этими знаниями, теперь можно сконцентрировать внимание на построении приложения WCF. Имейте в виду, что полное описание WCF потребовало бы целой книги, поскольку описание каждой из поддерживающих служб (скажем, MSMQ, COM+, P2P и именованных каналов) могло бы занять отдельную главу. Здесь вы изучите общий процесс построения программ WCF с использованием протоколов TCP и HTTP (т.е. веб-служб). Это должно подготовить почву для дальнейшего углубления знаний в данной области.

Исследование основных сборок WCF

Как и можно было ожидать, фабрика программного обеспечения WCF представлена набором сборок .NET, установленных в GAC. В табл. 25.1 описаны основные сборки WCF, которые понадобятся применять почти в любом приложении WCF.

Таблица 25.1. Основные сборки WCF

Сборка	Описание
<code>System.Runtime.Serialization.dll</code>	Эта основная сборка определяет пространства имен и типы, используемые для сериализации и десериализации объектов в инфраструктуре WCF
<code>System.ServiceModel.dll</code>	Эта основная сборка содержит типы, применяемые для построения приложений WCF любого рода

В двух сборках, представленных в табл. 25.1, определено много новых пространств имен и типов. Детальные сведения о них доступны в документации .NET Framework 4.5 SDK, а в табл. 25.2 описаны важные пространства имен, которые следует знать.

Таблица 25.2. Основные пространства имен WCF

Пространство имен	Описание
<code>System.Runtime.Serialization</code>	Это пространство имен определяет многочисленные типы, которые используются для управления сериализацией и десериализацией данных в WCF
<code>System.ServiceModel</code>	Это первичное пространство имен WCF определяет типы привязки и хостинга, а также базовые типы безопасности и транзакций
<code>System.ServiceModel.Configuration</code>	Это пространство имен определяет многочисленные типы, предоставляющие программный доступ к конфигурационным файлам WCF
<code>System.ServiceModel.Description</code>	Это пространство имен определяет типы, которые предоставляют объектную модель для адресов, привязок и контрактов, определенных внутри конфигурационных файлов WCF
<code>System.ServiceModel.MsmqIntegration</code>	Это пространство имен определяет типы для интеграции со службой MSMQ
<code>System.ServiceModel.Security</code>	Это пространство имен определяет многочисленные типы для управления аспектами уровней безопасности WCF

Шаблоны проектов WCF в Visual Studio

Как будет более подробно объясняться позже в этой главе, приложение WCF обычно представлено тремя сборками; одной из них является библиотека *.dll, содержащая типы, с которыми могут взаимодействовать внешние вызываемые компоненты (другими словами, сама служба WCF). Когда вы хотите построить службу WCF, совершенно допустимо выбрать в качестве начальной точки стандартный шаблон проекта Class Library (Библиотека классов), как было показано в главе 14, и вручную добавить ссылки на сборки WCF.

В качестве альтернативы новую службу WCF можно создать, выбрав в Visual Studio шаблон проекта WCF Service Library (Библиотека служб WCF), как показано на рис. 25.2. Этот тип проекта автоматически устанавливает ссылки на необходимые сборки WCF, однако он также генерирует и приличный объем начального кода, который довольно часто просто удаляется.

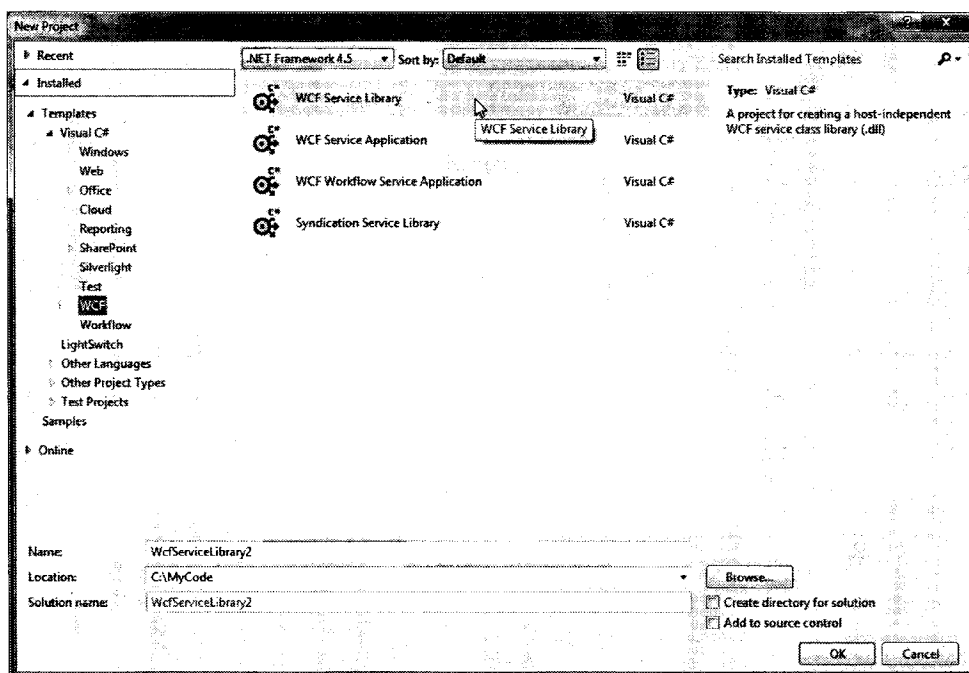


Рис. 25.2. Шаблон проекта WCF Service Library в Visual Studio

Одно из преимуществ выбора этого шаблона проекта связано с тем, что он также снабжает файлом App.config, что поначалу может показаться странным, т.к. строится .NET-сборка *.dll, а не *.exe. Однако этот файл очень полезен тем, что при отладке или запуске проекта WCF Service Library интегрированная среда разработки Visual Studio автоматически запустит приложение WCF Test Client (Тестовый клиент WCF). Программа WcfTestClient.exe читает настройки из файла App.config, поэтому может использоваться для тестирования службы. Более подробно эта программа рассматривается далее в главе.

На заметку! Файл App.config из проекта WCF Service Library также полезен тем, что демонстрирует начальные установки для конфигурации хост-приложения WCF. Фактически большую часть этого кода можно копировать и вставлять в конфигурационный файл реальных служб.

В дополнение к базовому шаблону WCF Service Library, в категории проектов WCF диалогового окна New Project (Новый проект) определены еще два библиотечных проекта WCF, которые интегрируют функциональность Windows Workflow Foundation (WF) в службу WCF, а также шаблон для построения библиотеки RSS (см. рис. 25.2). Технология Windows Workflow Foundation рассматривается в следующей главе, поэтому упомянутые шаблоны проектов WCF можно пока проигнорировать.

Шаблон проекта для построения веб-сайта со службами WCF

В Visual Studio доступен еще один шаблон проекта, связанный с WCF, который находится в диалоговом окне New Web Site (Новый веб-сайт), открываемом через пункт меню File⇒New⇒Web Site (Файл⇒Создать⇒Веб-сайт) и показанном на рис. 25.3.

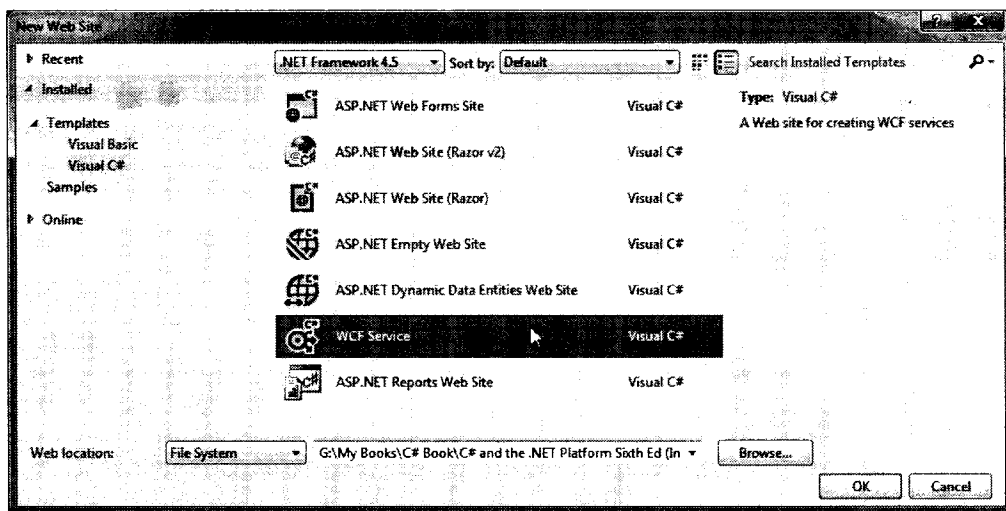


Рис. 25.3. Шаблон проекта WCF Service в Visual Studio

Шаблон проекта WCF Service (Служба WCF) удобен, когда заранее известно, что служба WCF будет использовать протоколы на основе HTTP, а не, к примеру, TCP или именованные каналы. Эта опция может автоматически создать новый виртуальный каталог Internet Information Services (IIS) для хранения программных файлов WCF, сформировать подходящий файл Web.config для открытия службы через HTTP и сгенерировать необходимый файл *.svc (подробнее о файлах *.svc рассказывается далее в этой главе). Таким образом, веб-ориентированный проект WCF Service просто экономит время, поскольку IDE-среда автоматически настроит всю необходимую инфраструктуру IIS.

В противоположность этому, если новая служба WCF создается с применением шаблона WCF Service Library, появляется возможность размещения службы несколькими способами (например, специальный хост Windows-служба или вручную созданный виртуальный каталог IIS). Эта опция больше подходит, когда требуется построение специального хоста для службы WCF, которая может работать с любым количеством привязок WCF.

Базовая структура приложения WCF

При построении распределенной системы WCF обычно создаются три следующих взаимосвязанных сборки.

- *Сборка службы WCF.* Эта библиотека *.dll содержит классы и интерфейсы, представляющие общую функциональность, которая предлагается внешним клиентам.
- *Хост службы WCF.* Этот программный модуль представляет собой сущность, которая размещает в себе сборку службы WCF.
- *Клиент WCF.* Это приложение, которое обращается к функциональности службы через промежуточный прокси.

Как упоминалось ранее, сборка службы WCF — это библиотека классов .NET, содержащая в себе множество контрактов WCF и их реализаций. Ключевое отличие состоит в том, что интерфейсные контракты оснащены разнообразными атрибутами, которые управляют представлением типа данных, взаимодействием исполняющей среды WCF с открытыми типами и т.д.

Вторая сборка — хост WCF — может быть буквально любой исполняемой программой .NET. Как будет показано в этой главе, WCF настраивается таким образом, что службы могут быть легко открыты из приложения любого типа (например, приложения Windows Forms, Windows-службы, приложения WPF). При построении специального хоста применяется тип `ServiceHost` и возможно связанный с ним файл *.config. Последний содержит детали, касающиеся механизмов серверной стороны, которые вы хотите использовать. Однако если в качестве хоста для службы WCF применяется IIS, то нет необходимости в программном построении специального хоста, поскольку IIS использует “за кулисами” тип `ServiceHost`.

На заметку! Развернуть службу WCF также можно с применением службы Windows Activation Service (WAS). За подробными сведениями обращайтесь в документацию .NET Framework 4.5 SDK.

Последняя сборка представляет клиента, который осуществляет вызовы службы WCF. Как и можно было ожидать, этим клиентом может быть приложение .NET любого типа. Подобно хосту, клиентское приложение также обычно использует файл *.config клиентской стороны, определяющий все механизмы на стороне клиента. Следует также помнить, что если служба WCF строится с применением привязок, основанных на HTTP, то клиентское приложение может быть реализовано на другой платформе (например, Java).

На рис. 25.4 показаны высокоуровневые отношения между этими тремя взаимосвязанными сборками WCF. “За кулисами” используется множество низкоуровневых деталей, реализующих все необходимые внутренние механизмы (фабрики, каналы, слушатели и т.п.). Эти низкоуровневые детали чаще всего скрыты от глаз; однако при необходимости они могут быть расширены или настроены. В большинстве случаев вполне подходят стандартные настройки.

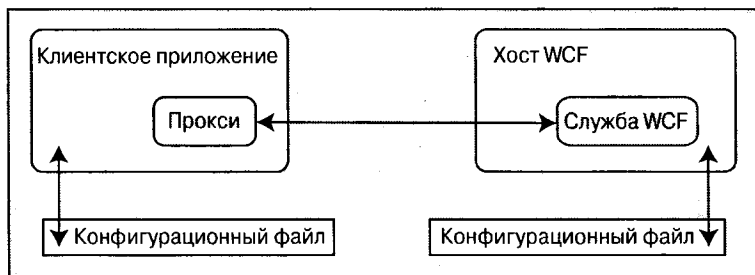


Рис. 25.4. Высокоуровневое представление типичного приложения WCF

Также полезно отметить, что применение файла *.config серверной или клиентской стороны не является обязательным. При желании можно жестко закодировать хост (а также клиент), указав необходимые детали (т.е. конечные точки, привязку, адреса). Очевидная проблема такого подхода состоит в том, что если нужно изменить детали настройки, понадобится вносить изменения в код, перекомпилировать и заново разворачивать множество сборок. Использование файла *.config делает кодовую базу намного более гибкой, поскольку все изменения в настройках сводятся к редактированию конфигурационных файлов и последующему перезапуску. С другой стороны, программные конфигурации обеспечивают приложению более динамичную гибкость — оно может выбирать конфигурацию внутренних механизмов, например, в зависимости от результатов проверки условий.

Адрес, привязка и контракт в WCF

Хосты и клиенты взаимодействуют друг с другом, согласовывая так называемые ABC — условное наименование для запоминания основных строительных блоков приложения WCF, таких как *адрес*, *привязка* и *контракт* (address, binding, contract — ABC).

- *Адрес*. Описывает местоположение службы. В коде представляется типом `System.Uri`, однако его значение обычно хранится в файлах *.config.
- *Привязка*. Инфраструктура WCF поставляется с множеством различных привязок, которые указывают сетевые протоколы, механизмы кодирования и транспортный уровень.
- *Контракт*. Предоставляет описание каждого метода, открытого из службы WCF.

Имейте в виду, что аббревиатура ABC не подразумевает, что разработчик обязан определить сначала адрес, за ним привязку и только потом — контракт. Во многих случаях разработчик WCF начинает с определения контракта для службы, а за ним адреса и привязок (допускается любой порядок при условии, что учтены все аспекты). Прежде чем перейти к построению первого приложения WCF, давайте более детально рассмотрим ABC.

Понятие контрактов WCF

Понятие *контракта* является ключевым при построении службы WCF. Хотя это и не обязательно, но подавляющее большинство приложений WCF будут начинаться с определения набора интерфейсных типов .NET, используемых для представления набора членов, которые будет поддерживать заданная служба WCF. В частности, интерфейсы, которые представляют контракт WCF, называются *контрактами служб*. Классы (или структуры), которые реализуют их, носят название *типы служб*.

Контракты служб WCF оснащаются разнообразными атрибутами, причем самые часто используемые из них определены в пространстве имен `System.ServiceModel`. Когда члены контракта службы (методы в интерфейсе) содержат только простые типы данных (такие как числовые, булевские и строковые), завершённую службу WCF можно построить, используя одни только атрибуты `[ServiceContract]` и `[OperationContract]`.

Однако если члены открывают специальные типы, скорее всего, будут применяться различные типы из пространства имен `System.Runtime.Serialization` (рис. 25.5), находящегося в сборке `System.Runtime.Serialization.dll`. Здесь доступны дополнительные атрибуты (наподобие `[DataMember]` и `[DataContract]`) для тонкой настройки процесса определения того, как составные типы будут сериализоваться и десериализоваться из XML при передаче в и из операций службы.

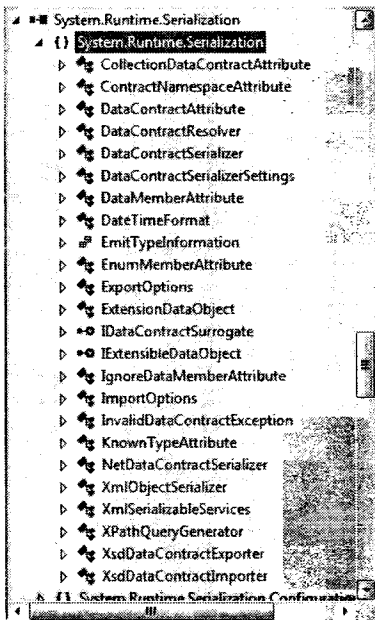


Рис. 25.5. В сборке `System.Runtime.Serialization` определен ряд атрибутов, используемых при построении контрактов данных WCF

Строго говоря, использовать интерфейсы CLR для определения контракта WCF не обязательно. Многие из этих атрибутов могут применяться к открытым членам открытого класса (или структуры). Однако, учитывая множество преимуществ программирования на основе интерфейсов (например, полиморфизм и элегантная поддержка множества версий), использование интерфейсов CLR для описания контракта WCF имеет смысл считать рекомендуемым приемом.

Понятие привязок WCF

Как только контракт (или набор контрактов) определен и реализован внутри библиотеки службы, следующий логический шаг заключается в построении агента хостинга для самой службы WCF. Как упоминалось ранее, на выбор доступно множество возможных хостов, и все они должны указывать *привязки*, применяемые удаленными клиентами для получения доступа к функциональности типа службы.

Инфраструктура WCF поставляется со многими вариантами привязок, каждая из которых ориентирована на определенные нужды. Если ни одна из готовых привязок не удовлетворяет существующим требованиям, можно создать собственную привязку, расширив тип `CustomBinding` (в настоящей главе это делаться не будет).

Привязка WCF может описывать следующие характеристики:

- транспортный уровень, используемый для перемещения данных (HTTP, MSMQ, именованные каналы и TCP);
- каналы, используемые транспортом (однаправленные, запрос-ответ и дуплексные);
- механизм кодирования, используемый для работы с данными (например, XML и двоичный);
- любые поддерживаемые протоколы веб-служб (если разрешены привязкой), такие как WS-Security, WS-Transactions, WS-Reliability и т.д.

Давайте рассмотрим возможные варианты.

Привязки на основе HTTP

Классы `BasicHttpBinding`, `WSHttpBinding`, `WSDualHttpBinding` и `WSFederationHttpBinding` предназначены для открытия контрактных типов через протоколы HTTP/SOAP. Если для разрабатываемой службы потребуются дополнительные возможности (например, множество операционных систем и множество программных архитектур), то эти привязки подойдут наилучшим образом, потому что все они кодируют данные на основе представления XML и используют в сети протокол HTTP.

В табл. 25.3 показано, как можно представлять привязки WCF в коде (с помощью классов из пространства имен `System.ServiceModel`) или в виде атрибутов XML, определенных внутри файлов `*.config`.

Таблица 25.3. Привязки WCF на основе HTTP

Класс привязки	Элемент привязки	Описание
BasicHttpBinding	<basicHttpBinding>	Используется для построения службы WCF, совместимой с профилем WS-Basic Profile (WS-I Basic Profile 1.1). Эта привязка использует HTTP в качестве транспорта и Text/XML в качестве стандартного метода кодирования сообщений
WSHttpBinding	<wsHttpBinding>	Подобен классу BasicHttpBinding, но предоставляет больше средств веб-служб. Эта привязка добавляет поддержку транзакций, надежной доставки сообщений и протокола WS-Addressing
WSDualHttpBinding	<wsDualHttpBinding>	Подобен классу WSHttpBinding, но предназначен для применения с дуплексными контрактами (например, когда служба и клиент могут посылать сообщения туда и обратно). Эта привязка поддерживает только безопасность SOAP и требует надежного обмена сообщениями
WSFederationHttpBinding	<wsFederationHttpBinding>	Безопасная привязка с возможностью взаимодействия, которая поддерживает протокол WS-Federation, позволяя организациям, объединенным в федерацию, эффективно проводить аутентификацию и авторизацию пользователей

Как и можно было догадаться, BasicHttpBinding является простейшим из всех протоколов на основе веб-служб. В частности, эта привязка гарантирует соответствие службы WCF спецификации по имени WS-I Basic Profile 1.1 (определенной WS-I). Основная причина применения этой привязки состоит в поддержке обратной совместимости с приложениями, ранее построенными для взаимодействия с веб-службами ASP.NET (которые были частью библиотек .NET, начиная с версии 1.0).

Протокол WSHttpBinding не только включает поддержку подмножества спецификаций WS-* (транзакции, безопасность и надежные сеансы), но также обеспечивает возможность обработки двоичного кодирования данных с использованием механизма MTOM (Message Transmission Optimization Mechanism — механизм оптимизации передачи сообщений).

Основное преимущество привязки WSDualHttpBinding в том, что она добавляет возможность *двустороннего (дуплексного) обмена сообщениями* между отправителем и получателем. При выборе WSDualHttpBinding можно применять модель публикации/подписки на события WCF.

И, наконец, WSFederationHttpBinding — это протокол на основе веб-служб, который стоит применять, когда наиболее важна безопасность в рамках группы организаций. Эта

привязка поддерживает спецификации WS-Trust, WS-Security и WS-SecureConversation, которые представлены API-интерфейсами CardSpace в WCF.

Привязки на основе TCP

При построении распределенного приложения, функционирующего на машинах, которые сконфигурированы с библиотеками .NET 4.5 (другими словами, на всех машинах установлена операционная система Windows), можно получить выигрыш в производительности, минуя привязки веб-служб и работая непосредственно с привязкой TCP, которая обеспечивает кодирование данных в компактном двоичном формате вместо XML. При использовании привязок, перечисленных в табл. 25.4, клиент и хост должны быть приложениями .NET.

Таблица 25.4. Привязки WCF на основе TCP

Класс привязки	Элемент привязки	Описание
NetNamedPipeBinding	<netNamedPipeBinding>	Безопасная, надежная, оптимизированная привязка для коммуникаций между приложениями .NET на одной и той же машине
NetPeerTcpBinding	<netPeerTcpBinding>	Безопасная привязка для сетевых приложений P2P
NetTcpBinding	<netTcpBinding>	Безопасная и оптимизированная привязка, подходящая для межмашинных коммуникаций между приложениями .NET

Класс NetTcpBinding использует протокол TCP для перемещения двоичных данных между клиентом и службой WCF. Как упоминалось ранее, это дает выигрыш в производительности по сравнению с протоколами веб-служб, но при этом решения ограничены средой Windows. Положительной стороной является поддержка в NetTcpBinding транзакций, надежных сеансов и безопасных коммуникаций.

Подобно NetTcpBinding, класс NetNamedPipeBinding поддерживает транзакции, надежные сеансы и безопасные коммуникации, но при этом обладает способностью межмашинных вызовов. Если вы ищете самый быстрый способ передачи данных между приложениями WCF на одной машине, то привязке NetNamedPipeBinding нет равных. Более подробную информацию о NetPeerTcpBinding можно найти в документации .NET Framework 4.5 SDK.

Привязки на основе MSMQ

И, наконец, если цель заключается в интеграции с сервером MSMQ, то непосредственный интерес представляют привязки NetMsmqBinding и MsmqIntegrationBinding. Детали применения привязок MSMQ в этой главе не рассматриваются, но в табл. 25.5 описано основное назначение каждой из них.

Понятие адресов WCF

После того как контракты и привязки установлены, остается указать *адрес* для службы WCF. Это важно, поскольку удаленные клиенты не смогут взаимодействовать с удаленными типами, если им не удастся найти их. Подобно большинству аспектов WCF, адрес может быть жестко закодирован в сборке (с использованием типа System.Uri) или вынесен в файл *.config.

Таблица 25.5. Привязки WCF на основе MSMQ

Класс привязки	Элемент привязки	Описание
MsmqIntegrationBinding	<msmqIntegrationBinding>	Эта привязка может применяться для того, чтобы позволить приложениям WCF отправлять и принимать сообщения от существующих приложений MSMQ, которые используют COM, собственный C++ или типы, определенные в пространстве имен System.Messaging
NetMsmqBinding	<netMsmqBinding>	Эта привязка на основе очередей может применяться для межмашинных коммуникаций между приложениями .NET. Это предпочтительный подход среди привязок, основанных на MSMQ

В любом случае точный формат адреса WCF отличается в зависимости от выбранной привязки (на основе HTTP, именованных каналов, TCP или MSMQ). На самом высоком уровне адреса WCF могут указывать перечисленные ниже единицы информации.

- Scheme. Транспортный протокол (HTTP и т.п.).
- MachineName. Полное доменное имя машины.
- Port. Во многих случаях не обязательный параметр. Например, привязка HTTP по умолчанию использует порт 80.
- Path. Путь к службе WCF.

Эта информация может быть представлена с помощью следующего обобщенного шаблона (значение Port необязательно, т.к. некоторые привязки его не используют):

```
Scheme://<MachineName>[:Port]/Path
```

При использовании привязки на основе HTTP (basicHttpBinding, wsHttpBinding, wsDualHttpBinding или wsFederationHttpBinding) адрес разбивается так, как показано ниже (вспомните, что если номер порта не указан, протоколы на основе HTTP по умолчанию выбирают порт 80):

```
http://localhost:8080/MyWCFService
```

Если применяется привязка на основе TCP (такая как NetTcpBinding или NetPeerTcpBinding), то URI принимает следующий формат:

```
net.tcp://localhost:8080/MyWCFService
```

Привязки на основе MSMQ (NetMsmqBinding и MsmqIntegrationBinding) уникальны в части их формата URI, учитывая, что MSMQ может использовать открытые или закрытые очереди (доступные только на локальной машине), а номера портов не имеют смысла в URI, связанных с MSMQ. Взгляните на следующий URI, который описывает закрытую очередь по имени MyPrivateQ:

```
net.msmq://localhost/private$/MyPrivateQ
```

И последнее: формат адреса, используемый привязкой для именованных каналов NetNamedPipeBinding, выглядит так, как показано ниже (вспомните, что именованные каналы делают возможными межпроцессные взаимодействия приложений на одной и той же физической машине):

```
net.pipe://localhost/MyWCFService
```

Хотя одиночная служба WCF может открывать только один адрес (основанный на единственной привязке), допускается конфигурировать коллекцию уникальных адресов (с разными привязками). Это делается внутри файла *.config за счет определения множества элементов <endpoint>. Для одной и той же службы можно указывать любое количество ABC. Такой подход полезен, когда необходимо позволить клиентам выбирать протокол, которые они хотят использовать для взаимодействия со службой.

Построение службы WCF

Теперь, когда вы получили представление о строительных блоках приложения WCF, давайте создадим первое простое приложение, чтобы посмотреть, как ABC можно учитывать в коде и конфигурации. В первом примере шаблоны проектов WCF из Visual Studio использоваться не будут, что позволит сосредоточиться на специфических шагах по созданию службы WCF. Начните с создания нового проекта библиотеки классов C# по имени MagicEightBallServiceLib.

Переименуйте начальный файл Class1.cs на MagicEightBallService.cs и добавьте ссылку на сборку System.ServiceModel.dll. В начальный файл кода добавьте оператор using для пространства имен System.ServiceModel.

К этому моменту файл C# должен выглядеть следующим образом (обратите внимание, что в этот момент мы имеем открытый класс):

```
// Основное пространство имен WCF.
using System.ServiceModel;

namespace MagicEightBallServiceLib
{
    public class MagicEightBallService
    {
    }
}
```

В этом классе реализован единственный контракт службы WCF, представленный строго типизированным интерфейсом CLR по имени IEightBall. Как известно, магический шар Magic 8-Ball — это игрушка, позволяющая получить один из набора ответов на задаваемый вопрос. В интерфейсе определен единственный метод, который позволит клиенту задать вопрос магическому шару, чтобы получить случайный ответ.

Интерфейсы службы WCF помечены атрибутом [ServiceContract], в то время как каждый член интерфейса снабжен атрибутом [OperationContract] (подробнее об этих двух атрибутах будет рассказываться позже). Ниже показано определение интерфейса IEightBall:

```
[ServiceContract]
public interface IEightBall
{
    // Задайте вопрос, получите ответ!
    [OperationContract]
    string ObtainAnswerToQuestion(string userQuestion);
}
```

На заметку! Допускается определять интерфейс контракта службы, который содержит методы, не оснащенные атрибутом [OperationContract]. Однако такие члены не будут открываться через исполняющую среду WCF.

Как известно из главы 8, интерфейс — довольно бесполезная вещь, пока он не реализован классом или структурой, которая наполнить его функциональностью.

Подобно реальному магическому шару, реализация типа `MagicEightBallService` будет возвращать случайно выбранный ответ из массива строк. Стандартный конструктор будет отображать информационное сообщение, которое будет (в конечном итоге) выведено в окне консоли хоста (для диагностических целей):

```
public class MagicEightBallService : IEightBall
{
    // Для отображения на хосте.
    public MagicEightBallService()
    {
        Console.WriteLine("The 8-Ball awaits your question...");
    }
    public string ObtainAnswerToQuestion(string userQuestion)
    {
        string[] answers = { "Future Uncertain", "Yes", "No",
            "Hazy", "Ask again later", "Definitely" };

        // Вернуть случайный ответ.
        Random r = new Random();
        return answers[r.Next(answers.Length)];
    }
}
```

На этом библиотека службы WCF завершена. Тем не менее, перед конструированием хоста для этой службы давайте ознакомимся с некоторыми подробностями атрибутов `[ServiceContract]` и `[OperationContract]`.

Атрибут `[ServiceContract]`

Чтобы интерфейс CLR принимал участие в службах, предоставляемых WCF, он должен быть снабжен атрибутом `[ServiceContract]`. Подобно многим другим атрибутам .NET, тип `ServiceContractAttribute` поддерживает набор свойств для дальнейшего прояснения его назначения. Два свойства — `Name` и `Namespace` — могут быть установлены для управления именем типа службы и именем пространства имен XML, определяющим тип службы. Если используется привязка HTTP, эти значения применяются для определения элементов `<portType>` связанного документа WSDL.

Здесь мы не заботимся о присваивании значения свойству `Name`, учитывая, что стандартное имя типа службы основано на имени класса C#. Однако стандартным именем для лежащего в основе пространства имен XML будет просто `http://tempuri.org` (оно должно быть изменено для всех создаваемых служб WCF).

При построении службы WCF, которая будет отправлять и получать специальные типы данных (чего мы пока не делаем), важно установить осмысленное значение для лежащего в основе пространства имен XML, чтобы гарантировать уникальность специального типа. Как вам должно быть известно из опыта построения веб-служб XML, пространства имен XML предоставляют способ помещения типов в уникальный контейнер, гарантируя отсутствие конфликтов с типами из других организаций.

По этой причине можно обновить определение интерфейса, сделав его более подходящим — почти так же, как это делается при определении пространства имен XML в проекте .NET Web Service, когда в качестве пространства имен указывается URI издателя службы. Например:

```
[ServiceContract(Namespace = "http://MyCompany.com")]
public interface IEightBall
{
    ...
}
```

Помимо `Namespace` и `Name`, атрибут `[ServiceContract]` может быть сконфигурирован с помощью дополнительных свойств, которые перечислены в табл. 25.6. Имейте в виду, что в зависимости от выбранной привязки, некоторые из этих настроек будут игнорироваться.

Таблица 25.6. Различные именованные свойства атрибута `[ServiceContract]`

Свойство	Описание
<code>CallbackContract</code>	Устанавливает функциональность обратного вызова для двустороннего обмена сообщениями
<code>ConfigurationName</code>	Это имя используется для нахождения элемента службы в конфигурационном файле приложения. По умолчанию представляет собой имя класса, реализующего службу
<code>ProtectionLevel</code>	Позволяет указать степень, до которой привязка контракта требует шифрования, цифровых подписей или того и другого для конечных точек, открываемых контрактом
<code>SessionMode</code>	Используется для установки разрешения сеанса, запрета сеанса или обязательности сеанса для данного контракта службы

Атрибут `[OperationContract]`

Методы, которые планируется использовать внутри WCF, должны быть оснащены атрибутом `[OperationContract]`, который также может быть сконфигурирован с помощью различных именованных свойств. Используя свойства, перечисленные в табл. 25.7, можно указать, что данный метод предназначен для однонаправленной работы, поддерживает асинхронные вызовы, требует шифрования данных сообщений и т.д. (в зависимости от выбранной привязки, многие из этих значений могут быть проигнорированы).

Таблица 25.7. Различные именованные свойства атрибута `[OperationContract]`

Свойство	Описание
<code>AsyncPattern</code>	Указывает, реализована ли операция асинхронно с использованием пары методов <code>Begin/End</code> службы. Это позволяет службе передавать обработку другому потоку серверной стороны; это не имеет никакого отношения к асинхронному вызову метода клиентом!
<code>IsInitiating</code>	Указывает, может ли эта операция быть начальной операцией сеанса
<code>IsOneWay</code>	Указывает, состоит ли операция только из одного входного сообщения (и никакого ассоциированного вывода)
<code>IsTerminating</code>	Указывает, должна ли исполняющая среда WCF пытаться завершить текущий сеанс после выполнения операции

В этом начальном примере дополнительное конфигурирование метода `ObtainAnswerToQuestion()` не требуется, поэтому атрибут `[OperationContract]` оставлен в том виде, как он определен сейчас.

Служебные типы как контракты операций

И, наконец, вспомните, что при построении служебных типов WCF использование интерфейсов не обязательно.

Фактически атрибуты [ServiceContract] и [OperationContract] можно применять только непосредственно к самому служебному типу, примерно так:

```
// Только для целей иллюстрации;
// в текущем примере не используется.
[ServiceContract(Namespace = "http://MyCompany.com")]
public class ServiceTypeAsContract
{
    [OperationContract]
    void SomeMethod() { }

    [OperationContract]
    void AnotherMethod() { }
}
```

Хотя такой подход возможен, явное определение интерфейсного типа для представления контракта службы дает массу преимуществ. Наиболее очевидный выигрыш состоит в том, что один интерфейс может быть применен к нескольким типам служб (написанных на разных языках и в разных архитектурах), чтобы достичь высокой степени полиморфизма. Другое преимущество связано с тем, что интерфейс контракта службы может использоваться в качестве основы для новых контрактов (через наследование интерфейсов), без необходимости заботиться о реализации.

В любом случае к данному моменту ваша первая библиотека службы WCF готова. Скомпилируйте проект, чтобы удостовериться в отсутствии опечаток.

Исходный код. Проект MagicEightBallServiceLib доступен в подкаталоге MagicEightBallServiceHTTP каталога Chapter 25.

Хостинг службы WCF

Теперь все готово для определения хоста. Хотя служба производственного уровня должна развертываться в службе Windows или виртуальном каталоге IIS, наш первый хост будет просто консольным приложением по имени MagicEightBallServiceHost.

После создания нового проекта консольного приложения добавьте ссылку на сборки System.ServiceModel.dll и MagicEightBallServiceLib.dll и обновите начальный файл кода, добавив импорт пространств имен System.ServiceModel и MagicEightBallServiceLib:

```
using System;
...

using System.ServiceModel;
using MagicEightBallServiceLib;

namespace MagicEightBallServiceHost
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** Console Based WCF Host *****");
            Console.ReadLine();
        }
    }
}
```

Первый шаг, который потребуется предпринять при построении хоста для служебного типа WCF, связан с решением, будет ли необходимая логика хостинга определена

полностью в коде либо несколько низкоуровневых деталей будут перемещены в конфигурационный файл приложения. Как упоминалось ранее, преимущество файлов *.config состоит в том, что хост может изменять низкоуровневые механизмы без перекомпиляции и повторного развертывания исполняемой программы. Однако всегда помните, что это совершенно не обязательно, и можно жестко закодировать логику хостинга с использованием типов из сборки System.ServiceModel.dll.

Этот консольный хост будет использовать конфигурационный файл приложения, поэтому добавьте новый файл (если его еще нет в проекте) в текущий проект, воспользовавшись пунктом меню Project⇒Add New Item (Проект⇒Добавить новый элемент) и затем выбрав элемент Application Configuration File (Конфигурационный файл приложения).

Установка ABC внутри файла App.config

При построении хоста для служебного типа WCF необходимо следовать заранее предсказуемому набору шагов, часть из которых полагается на конфигурацию, а часть — на код.

- Определить конечную точку для службы WCF в конфигурационном файле хоста.
- Программно использовать тип ServiceHost для открытия служебных типов, доступных из этой конечной точки.
- Обеспечить постоянную работу хоста для обслуживания входящих клиентских запросов. Очевидно, что этот шаг не обязателен, если для хостинга применяется служба Windows или IIS.

В мире WCF термин *конечная точка* представляет адрес, привязку и контракт, объединенные вместе в один пакет. В XML конечная точка выражается элементом <endpoint> и его атрибутами address, binding и contract. Модифицируйте файл *.config, указав в нем единственную конечную точку (доступную через порт 8080), открытую данным хостом:

```
<?xml version = "1.0" encoding = "utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name = "MagicEightBallServiceLib.MagicEightBallService">
        <endpoint address = "http://localhost:8080/MagicEightBallService"
          binding = "basicHttpBinding"
          contract = "MagicEightBallServiceLib.IEightBall"/>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

Обратите внимание, что элемент <system.serviceModel> находится в корне всех настроек WCF хоста. Каждая служба, развернутая на хосте, представлена элементом <service>, который помещен в базовый элемент <services>. Здесь единственный элемент <service> использует (необязательный) атрибут name для указания дружественного имени служебного типа.

С помощью вложенного элемента <endpoint> задается адрес, модель привязки (basicHttpBinding в этом примере) и полностью заданное имя интерфейсного типа, определяющего контракт службы WCF (IEightBall). Поскольку применяется привязка на основе HTTP, указывается схема http:// с произвольным идентификатором порта.

Кодирование с использованием типа `ServiceHost`

При текущем конфигурационном файле действительная логика программирования, необходимая для завершения хоста, чрезвычайно проста. Когда исполняемая программа стартует, создается экземпляр типа `ServiceHost`, которому сообщается служба WCF, отвечающая за хостинг. Во время выполнения этот объект автоматически читает данные из контекста элемента `<system.serviceModel>` файла `*.config` хоста для определения правильного адреса, привязки и контракта, и создает все необходимые механизмы:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Console Based WCF Host *****");
    using (ServiceHost serviceHost = new ServiceHost(typeof(MagicEightBallService)))
    {
        // Открыть хост и начать прослушивание входных сообщений.
        serviceHost.Open();
        // Оставить службу функционирующей до тех пор, пока не будет нажата клавиша <Enter>
        Console.WriteLine("The service is ready.");
        Console.WriteLine("Press the Enter key to terminate service.");
        Console.ReadLine();
    }
}
```

После запуска этого приложения вы обнаружите, что хост находится в памяти и готов к приему входящих запросов от удаленных клиентов.

На заметку! Вспомните, что для выполнения многих типов проектов WCF среда Visual Studio должна быть запущена с административными привилегиями!

Указание базовых адресов

В настоящее время объект `ServiceHost` создается с использованием конструктора, который требует только информацию о типе службы. Однако в качестве аргумента конструктора можно также передать массив элементов типа `System.Uri`, чтобы представить коллекцию адресов, для которых доступна данная служба. В настоящий момент адрес обнаруживается через файл `*.config`; однако если обновить контекст `using` следующим образом:

```
using (ServiceHost serviceHost = new
    ServiceHost (typeof(MagicEightBallService),
        new Uri[] {new Uri("http://localhost:8080/MagicEightBallService")}))
{
    ...
}
```

то конечную точку можно определить так:

```
<endpoint address = ""
    binding = "basicHttpBinding"
    contract = "MagicEightBallServiceLib.IEightBall"/>
```

Разумеется, слишком большой объем жесткого кодирования внутри кодовой базы хоста снижает гибкость, поэтому в текущем примере предполагается, что хост службы создается просто за счет передачи информации о типе, как делалось раньше:

```
using (ServiceHost serviceHost = new ServiceHost (typeof(MagicEightBallService)))
{
    ...
}
```


Один из несколько обескураживающих аспектов написания файлов *.config связан с тем, что существует несколько способов конструирования дескрипторов XML, в зависимости от объема жесткого кодирования (как это было в случае с необязательным массивом Uri). Чтобы продемонстрировать другой способ написания файла *.config, рассмотрим следующее изменение:

```
<?xml version = "1.0" encoding = "utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name = "MagicEightBallServiceLib.MagicEightBallService">
        <!-- Адрес получен из <baseAddresses> -->
        <endpoint address = ""
          binding = "basicHttpBinding"
          contract = "MagicEightBallServiceLib.IEightBall"/>

        <!-- Перечислить все базовые адреса в выделенном разделе -->
        <host>
          <baseAddresses>
            <add baseAddress = "http://localhost:8080/MagicEightBallService"/>
          </baseAddresses>
        </host>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

В данном случае атрибут address элемента <endpoint> по-прежнему пуст и, невзирая на то, что массив Uri при создании ServiceHost в коде не указывался, приложение работает, как и раньше, поскольку значение извлекается из контекста baseAddresses. Преимущество хранения базового адреса в подразделе <baseAddresses> раздела <host> состоит в том, что другие части файла *.config также должны знать адрес конечной точки службы. Поэтому вместо копирования и вставки значений адресов внутри файла *.config можно изолировать единственное значение, как было показано выше.

На заметку! В последующем примере будет представлен графический инструмент конфигурирования, позволяющий создавать конфигурационные файлы менее утомительным образом.

В любом случае, перед тем, как построить клиентское приложение для взаимодействия со службой, давайте немного углубимся в исследование роли класса ServiceHost и элемента <service.serviceModel>, а также роли служб обмена метаданными (metadata exchange — MEX).

Подробный анализ типа ServiceHost

Класс ServiceHost применяется для конфигурирования и представления службы WCF из приложения-хоста. Однако имейте в виду, что этот тип будет использоваться напрямую только при построении специальных сборок *.exe, предназначенных для хостинга служб. Если же для открытия службы применяется IIS, то объект ServiceHost создается автоматически.

Как уже было показано, этот тип требует полного описания службы, которое получается динамически через конфигурационные настройки файла *.config хоста. Хотя это происходит автоматически при создании объекта, можно вручную сконфигурировать состояние объекта ServiceHost с помощью ряда его членов. Кроме Open() и Close() (которые взаимодействуют со службой в синхронной манере), есть и другие члены этого класса, перечисленные в табл. 25.8.

Таблица 25.8. Избранные члены типа `ServiceHost`

Член	Описание
<code>Authorization</code>	Это свойство получает уровень авторизации для размещенной службы
<code>AddDefaultEndpoints()</code>	Этот метод применяется для программного конфигурирования хоста службы WCF, чтобы он использовал любое количество готовых конечных точек, предоставленных платформой
<code>AddServiceEndpoint()</code>	Этот метод позволяет программно регистрировать конечную точку для хоста
<code>BaseAddresses</code>	Это свойство получает список зарегистрированных базовых адресов для текущей службы
<code>BeginOpen()</code> <code>BeginClose()</code>	Эти методы позволяют асинхронно открывать и закрывать объект <code>ServiceHost</code> , используя стандартный асинхронный синтаксис делегата .NET
<code>CloseTimeout</code>	Это свойство позволяет устанавливать и получать время, отведенное службе на закрытие
<code>Credentials</code>	Это свойство получает удостоверения безопасности, используемые текущей службой
<code>EndOpen()</code> <code>EndClose()</code>	Эти методы представляют собой асинхронные аналоги <code>BeginOpen()</code> и <code>BeginClose()</code>
<code>OpenTimeout</code>	Это свойство позволяет устанавливать и получать время, отведенное службе на запуск
<code>State</code>	Это свойство получает значение, которое указывает текущее состояние объекта коммуникации, представленное перечислением <code>CommunicationState</code> (например, <code>Opened</code> , <code>Closed</code> , <code>Created</code>)

Чтобы проиллюстрировать в действии некоторые дополнительные аспекты `ServiceHost`, модифицируем класс `Program`, добавив новый статический метод, который выводит на консоль различные аспекты текущего хоста:

```
static void DisplayHostInfo(ServiceHost host)
{
    Console.WriteLine();
    Console.WriteLine("***** Host Info *****");
    foreach (System.ServiceModel.Description.ServiceEndpoint se
        in host.Description.Endpoints)
    {
        Console.WriteLine("Address: {0}", se.Address);           // адрес
        Console.WriteLine("Binding: {0}", se.Binding.Name);     // привязка
        Console.WriteLine("Contract: {0}", se.Contract.Name);   // контракт
        Console.WriteLine();
    }
    Console.WriteLine("*****");
}
```

Предположим, что этот новый метод вызывается в `Main()` после открытия хоста:

```
using (ServiceHost serviceHost = new ServiceHost(typeof(MagicEightBallService)))
{
    // Открыть хост и начать прослушивание входящих сообщений.
    serviceHost.Open();
    DisplayHostInfo(serviceHost);
    ...
}
```

В результате выводится следующая статистика:

```
***** Console Based WCF Host *****
***** Host Info *****
Address: http://localhost:8080/MagicEightBallService
Binding: BasicHttpBinding
Contract: IEightBall
*****
The service is ready.
Press the Enter key to terminate service.
```

Подробный анализ элемента <system.serviceModel>

Подобно любому XML-элементу, в <system.serviceModel> может определяться набор подэлементов, каждый из которых может быть снабжен многочисленными атрибутами. Хотя все детали набора возможных атрибутов описаны в документации .NET Framework 4.5 SDK, ниже приведен скелет, в котором перечислены полезные подэлементы (далеко не все):

```
<system.serviceModel>
  <behaviors>
</behaviors>
  <client>
</client>
  <commonBehaviors>
</commonBehaviors>
  <diagnostics>
</diagnostics>
  <comContracts>
</comContracts>
  <services>
</services>
  <bindings>
</bindings>
</system.serviceModel>
```

На протяжении этой главы вы увидите и более экзотические конфигурационные файлы; в табл. 25.9 приведены краткие описания подэлементов.

Включение обмена метаданными

Вспомните, что клиентское приложение WCF взаимодействует со службой WCF через промежуточный прокси. Хотя вполне можно написать код прокси вручную, это было бы довольно утомительно и чревато ошибками. В идеале должен использоваться инструмент для генерации необходимого рутинного кода (включая файлы *.config клиентской стороны). К счастью, в рамках .NET Framework 4.5 SDK доступен инструмент командной строки (svcutil.exe), предназначенный именно для этих целей. Кроме того, Visual Studio предлагает аналогичную функциональность через пункт меню Project⇒Add Service Reference (Проект⇒Добавить ссылку на службу).

Чтобы эти инструменты генерировали необходимый код прокси и файл *.config, они должны иметь возможность исследовать формат интерфейсов службы WCF и любых определенных контрактов данных (т.е. имена методов и типы параметров).

Обмен метаданными (metadata exchange — MEX) — это поведение службы WCF, которое может применяться для тонкой настройки способа обработки службы исполняющей средой WCF. Выражаясь просто, каждый элемент <behavior> может определять набор действий, на которые данная служба может подписываться.

Таблица 25.9. Избранные подэлементы <service.serviceModel>

Подэлемент	Описание
behaviors	Инфраструктура WCF поддерживает различные поведения конечных точек и служб. По сути, <i>поведение</i> позволяет точнее задавать функциональность хоста или клиента
bindings	Этот элемент позволяет тонко настраивать каждую привязку WCF (basicHttpBinding, netMsmqBinding и т.д.), а также указывать любые специальные привязки, используемые хостом
client	Этот элемент содержит список конечных точек, используемых клиентом для подключения к службе. Очевидно, что это не особенно полезно в файле *.config хоста
comContracts	Этот элемент определяет контракты COM, обеспечивающие возможность взаимодействия WCF и COM
commonBehaviors	Этот элемент может устанавливаться только внутри файла machine.config. Он применяется для определения всех поведений, используемых каждой службой WCF на заданной машине
diagnostics	Этот элемент содержит настройки для средств диагностики WCF. Пользователь может включать или отключать трассировку, счетчики производительности и поставщика WMI, а также добавлять специальные фильтры сообщений
services	Этот элемент содержит коллекцию служб WCF, открываемых хостом

WCF предоставляет многочисленные поведения в готовом виде, к тому же можно строить собственные поведения.

Поведение MEX (которое по умолчанию отключено) перехватит любые запросы метаданных, отправленные через HTTP-запрос GET. Чтобы позволить svcutil.exe или Visual Studio автоматизировать создание необходимого прокси клиентской стороны и файла *.config, понадобится включить MEX.

Включение MEX осуществляется в файле *.config хоста с помощью соответствующих настроек (или написанием подходящего кода C#). Во-первых, необходимо добавить новый элемент <endpoint> конкретно для MEX. Во-вторых, потребуется определить поведение WCF для разрешения доступа HTTP GET. В-третьих, нужно ассоциировать это поведение по имени со службой с помощью атрибута behaviorConfiguration в открывающем элементе <service>. И, наконец, в-четвертых, понадобится добавить элемент <host> для определения базового класса этой службы (MEX будет искать здесь местоположение описываемых типов).

На заметку! Финальный шаг можно опустить, если для представления базового адреса конструктору ServiceHost передается в качестве параметра объект System.Uri.

Взгляните на следующий измененный файл *.config хоста, который создает специальный элемент <behavior> (по имени EightBallServiceMEXBehavior), ассоциированный со службой через атрибут behaviorConfiguration внутри определения <service>:

```
<?xml version = "1.0" encoding = "utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name = "MagicEightBallServiceLib.MagicEightBallService"
        behaviorConfiguration="EightBallServiceMEXBehavior">
```

```

<endpoint address = ""
    binding = "basicHttpBinding"
    contract = "MagicEightBallServiceLib.IEightBall"/>

<!-- Включить конечную точку MEX -->
<endpoint address = "mex"
    binding = "mexHttpBinding"
    contract = "IMetadataExchange" />

<!-- Это необходимо добавить, чтобы MEX был известен адрес нашей службы -->
<host>
    <baseAddresses>
        <add baseAddress = "http://localhost:8080/MagicEightBallService"/>
    </baseAddresses>
</host>
</service>
</services>

<!-- Определение поведения для MEX -->
<behaviors>
    <serviceBehaviors>
        <behavior name = "EightBallServiceMEXBehavior" >
            <serviceMetadata httpGetEnabled = "true" />
        </behavior>
    </serviceBehaviors>
</behaviors>
</system.serviceModel>
</configuration>

```

Теперь можно перезапустить приложение-хост службы и просмотреть описание метаданных в веб-браузере. Для этого при функционирующем хосте введите следующий URL в строке адреса:

`http://localhost:8080/MagicEightBallService`

На домашней странице службы WCF (рис. 25.6) можно получить базовую информацию о том, как программно взаимодействовать с этой службой.

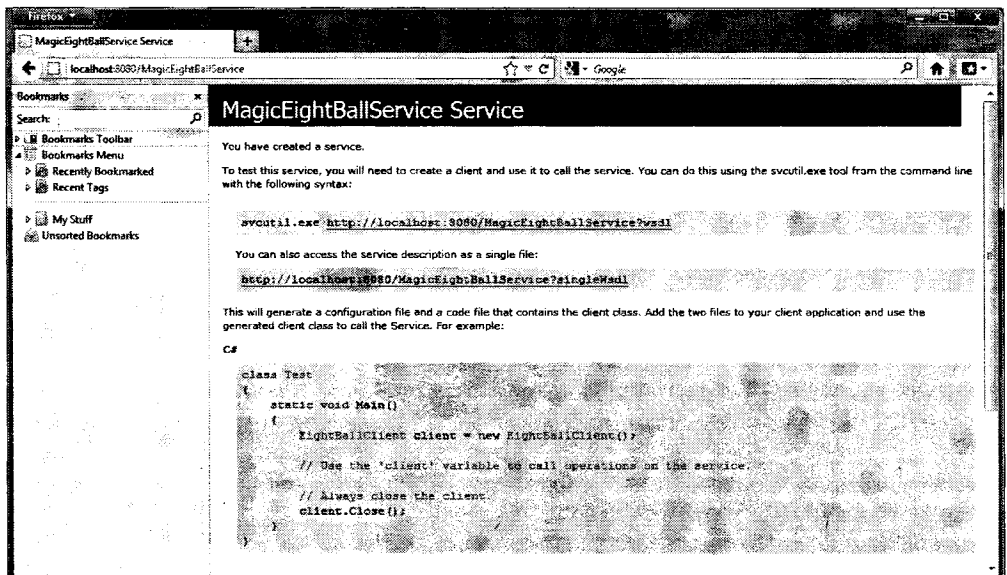


Рис. 25.6. Просмотр метаданных с использованием MEX

Щелчок на гиперссылке в верхней части страницы позволяет просмотреть контракт WSDL. Вспомните, что язык описания веб-служб (Web Service Description Language — WSDL) — это грамматика, описывающая структуру веб-служб в заданной конечной точке.

Хост теперь открывает две различных конечных точки (одна для службы и одна для MEX), поэтому консольный вывод хоста будет выглядеть следующим образом:

```
***** Console Based WCF Host *****
***** Host Info *****
Address: http://localhost:8080/MagicEightBallService
Binding: BasicHttpBinding
Contract: IEightBall
Address: http://localhost:8080/MagicEightBallService/mex
Binding: MetadataExchangeHttpBinding
Contract: IMetadataExchange
*****
The service is ready.
```

Исходный код. Проект MagicEightBallServiceHost доступен в подкаталоге MagicEightBallServiceHTTP каталога Chapter 25.

Построение клиентского приложения WCF

Теперь, имея готовый хост, последняя задача заключается в построении фрагмента программного обеспечения для взаимодействия с этим служебным типом WCF. Хотя можно избрать длинный путь и построить всю необходимую инфраструктуру вручную (осуществимая, но трудоемкая задача), в .NET Framework 4.5 SDK предлагается несколько подходов для быстрой генерации прокси клиентской стороны. Начните с создания нового консольного приложения по имени MagicEightBallServiceClient.

Генерация кода прокси с использованием svcutil.exe

Первый способ построения прокси клиентской стороны предусматривает использование инструмента командной строки svcutil.exe. С его помощью можно генерировать новый файл на языке C#, представляющий код прокси, а также конфигурационный файл клиентской стороны. Для этого укажите в первом параметре конечную точку службы. Флаг /out: применяется для определения имени файла *.cs, содержащего код прокси, а флаг /config: позволяет указать имя генерируемого файла *.config клиентской стороны.

Предполагая, что служба запущена, следующий набор параметров, переданный svcutil.exe, приведет к генерации двух новых файлов в рабочем каталоге (вся команда с параметрами должна вводиться в одной строке внутри окна командной строки разработчика):

```
svcutil http://localhost:8080/MagicEightBallService
/out:myProxy.cs /config:app.config
```

Открыв файл myProxy.cs, вы найдете там представление интерфейса IEightBall клиентской стороны, а также новый класс по имени EightBallClient, который и является классом прокси.

Этот класс унаследован от обобщенного класса System.ServiceModel.ClientBase<T>, где T — зарегистрированный интерфейс службы.

В дополнение к ряду специальных конструкторов, каждый метод прокси (который основан на исходных методах интерфейса) будет реализован для использования унаследованного свойства `Channels` с целью вызова корректного метода службы. Ниже показан частичный код типа прокси:

```
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.CodeDom.Compiler.GeneratedCodeAttribute("System.ServiceModel",
    "4.5.0.0")]
public partial class EightBallClient :
    System.ServiceModel.ClientBase<IEightBall>, IEightBall
{
    ...
    public string ObtainAnswerToQuestion(string userQuestion)
    {
        return base.Channel.ObtainAnswerToQuestion(userQuestion);
    }
}
```

При создании экземпляра типа прокси в клиентском приложении базовый класс установит соединение с конечной точкой, используя настройки, указанные в конфигурационном файле приложения клиентской стороны. Во многом подобно конфигурационному файлу серверной стороны, сгенерированный файл `App.config` стороны клиента содержит элемент `<endpoint>` и детали, которые касаются привязки `basicHttpBinding`, используемой для взаимодействия со службой.

Кроме того, там имеется следующий элемент `<client>`, который устанавливает АВС с точки зрения клиента:

```
<client>
  <endpoint
    address = "http://localhost:8080/MagicEightBallService"
    binding = "basicHttpBinding" bindingConfiguration = "BasicHttpBinding_IEightBall"
    contract = "IEightBall" name = "BasicHttpBinding_IEightBall" />
</client>
```

В этот момент можно было бы включить эти два файла в проект клиента (вместе со ссылкой на сборку `System.ServiceModel.dll`) и применять тип прокси для коммуникаций с удаленной службой WCF. Однако воспользуемся другим подходом и посмотрим, как Visual Studio может помочь в дальнейшей автоматизации создания файлов прокси клиентской стороны.

Генерация кода прокси с использованием Visual Studio

Подобно любому хорошему инструменту командной строки, в `svcutil.exe` предусмотрено огромное количество опций, которые можно использовать для управления процессом генерации прокси. Если же расширенные опции не нужны, те же два файла можно сгенерировать в IDE-среде Visual Studio. Просто выберите пункт `Add Service Reference` (Добавить ссылку на службу) в меню `Project` (Проект).

После выбора этого пункта меню будет предложено ввести URI службы. Щелкните на кнопке `Go` (Перейти), чтобы увидеть описание службы (рис. 25.7).

Помимо создания и вставки файлов прокси в текущий проект, этот инструмент автоматически установит ссылки на сборки WCF. В соответствии с соглашением об именовании, класс прокси определен в пространстве имен `ServiceReference1`, которое вложено в пространство имен клиента (во избежание возможных конфликтов имен).

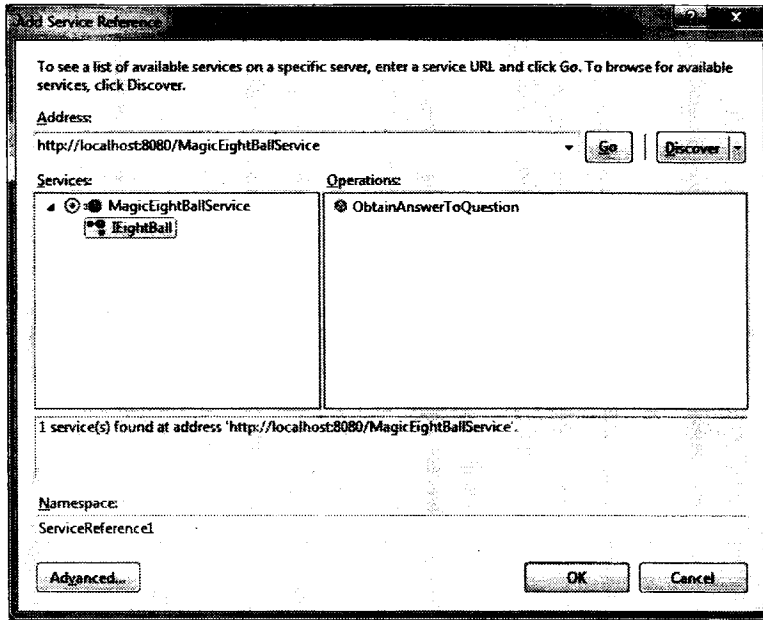


Рис. 25.7. Генерация файлов прокси с использованием Visual Studio

Ниже приведен полный код клиента:

// Местоположение прокси.

```
using MagicEightBallServiceClient.ServiceReference1;
namespace MagicEightBallServiceClient
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** Ask the Magic 8 Ball *****\n");
            using (EightBallClient ball = new EightBallClient())
            {
                Console.Write("Your question: ");
                string question = Console.ReadLine();
                string answer =
                    ball.ObtainAnswerToQuestion(question);
                Console.WriteLine("8-Ball says: {0}", answer);
            }
            Console.ReadLine();
        }
    }
}
```

Предполагая, что хост WCF запущен, можно выполнить программу клиента. Вывод может выглядеть следующим образом:

```
***** Ask the Magic 8 Ball *****
Your question: Will I ever finish Skyrim?
8-Ball says: No
Press any key to continue . . .
```

Исходный код. Проект MagicEightBallServiceClient доступен в подкаталоге MagicEightBallServiceHTTP каталога Chapter 25.

Конфигурирование привязки на основе TCP

К этому моменту приложения хоста и клиента сконфигурированы на применение простейшей из привязок, основанной на HTTP — `basicHttpBinding`. Вспомните, что преимущество переноса настроек в конфигурационные файлы связано с возможностью менять внутренние механизмы в декларативной манере и предоставлять множество привязок для одной и той же службы.

В целях демонстрации проведем небольшой эксперимент. Создайте новую папку на диске C: (или где был сохранен код) по имени `EightBallTCP` и внутри него два подкаталога с именами `Host` и `Client`.

Затем в проводнике Windows перейдите в папку `bin\Debug` проекта хоста (рассмотренного ранее в главе) и скопируйте `MagicEightBallServiceHost.exe`, `MagicEightBallServiceHost.exe.config` и `MagicEightBallServiceLib.dll` в папку `C:\EightBallTCP\Host`. Откройте файл `*.config` в простом текстовом редакторе и модифицируйте его содержимое следующим образом:

```
<?xml version = "1.0" encoding = "utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name = "MagicEightBallServiceLib.MagicEightBallService">
        <endpoint address = ""
                  binding = "netTcpBinding"
                  contract = "MagicEightBallServiceLib.IEightBall"/>
      </service>
    </services>
    <host>
      <baseAddresses>
        <add baseAddress = "net.tcp://localhost:8090/MagicEightBallService"/>
      </baseAddresses>
    </host>
  </system.serviceModel>
</configuration>
```

По сути, из файла `*.config` удалены все настройки MEX (поскольку уже создан прокси) и установлено использование типа `NetTcpBinding` через уникальный порт. Теперь запустите приложение двойным щелчком на его файле `*.exe`. Если все сделано правильно, должен появиться вывод, показанный ниже:

```
***** Console Based WCF Host *****
***** Host Info *****
Address: net.tcp://localhost:8090/MagicEightBallService
Binding: NetTcpBinding
Contract: IEightBall
*****
The service is ready.
Press the Enter key to terminate service.
```

Для завершения этого теста скопируйте файлы `MagicEightBallServiceClient.exe` и `MagicEightBallServiceClient.exe.config` из папки `bin\Debug` клиентского приложения (рассматривалось ранее в главе) в папку `C:\EightBallTCP\Client`. Модифицируйте конфигурационный файл следующим образом:

```
<?xml version = "1.0" encoding = "utf-8" ?>
<configuration>
  <system.serviceModel>
    <client>
```

```

    <endpoint address = "net.tcp://localhost:8090/MagicEightBallService"
      binding = "netTcpBinding"
      contract = "ServiceReference1.IEightBall"
      name = "netTcpBinding_IEightBall" />
  </client>
</system.serviceModel>
</configuration>

```

Этот конфигурационный файл клиентской стороны значительно проще файла, создаваемого генератором прокси Visual Studio. Обратите внимание, что существующий элемент `<bindings>` полностью удален. Изначально файл `*.config` содержал элемент `<bindings>` с подэлементом `<basicHttpBinding>`, который определял множество деталей, касающихся настроек привязки клиента (таймауты и т.п.).

В действительности для рассматриваемого примера эти детали никогда не понадобятся, поскольку мы автоматически получим стандартные значения лежащего в основе объекта `BasicHttpBinding`. При необходимости можно было бы модифицировать существующий элемент `<bindings>`, определив детали подэлемента `<netTcpBinding>`, но это совершенно не обязательно, если устраивают стандартные значения объекта `NetTcpBinding`.

В любом случае теперь вы должны быть готовы запустить клиентское приложение, и если хост все еще работает в фоновом режиме, вы сможете перемещать данные между сборками по протоколу TCP.

Исходный код. Проект `MagicEightBallTCP` доступен в подкаталоге `Chapter 25`.

Упрощение конфигурационных настроек

Работая над первым примером этой главы, вы могли заметить, что логика конфигурации хостинга довольно громоздка. Например, файл `*.config` хоста (для начальной базовой привязки HTTP) должен определять элемент `<endpoint>` службы, второй элемент `<endpoint>` для MEX, элемент `<baseAddresses>` (необязательный) для сокращения избыточных URI, а затем еще и раздел `<behaviors>` для определения характеристик обмена метаданными во время выполнения.

По правде говоря, изучение правил написания файлов `*.config` при построении служб WCF может оказаться довольно трудным. Чтобы еще более усложнить картину, значительное количество служб WCF склонно требовать одинаковых базовых настроек в конфигурационном файле хоста. Например, если создается совершенно новая служба WCF и совершенно новый хост, и нужно открыть эту службу, используя элемент `<basicHttpBinding>` с поддержкой MEX, необходимый файл `*.config` будет выглядеть практически идентичным созданному ранее.

К счастью, начиная с .NET 4.0, API-интерфейс Windows Communication Foundation включает ряд упрощений, в числе которых стандартные установки (и прочие сокращения), облегчающие процесс построения конфигурации хоста.

Использование стандартных конечных точек

До появления поддержки стандартных конечных точек, когда вызывался метод `Open()` на объекте `ServiceHost`, а в конфигурационном файле еще не было определено ни одного элемента `<endpoint>`, исполняющая среда генерировала исключение. Аналогичный результат получался при вызове метода `AddServiceEndpoint()` в коде для указания конечной точки. Однако в версии .NET 4.5 каждая служба WCF автоматически получает *стандартные конечные точки*, которые фиксируют общепринятые детали конфигурации для каждого поддерживаемого протокола.

Открыв файл `machine.config` для .NET 4.5, вы обнаружите в нем новый элемент имени `<protocolMapping>`. Этот элемент документирует привязки WCF, которые будут применяться по умолчанию, если никаких привязок явно не указано:

```
<system.serviceModel>
...
<protocolMapping>
  <add scheme = "http" binding="basicHttpBinding"/>
  <add scheme = "net.tcp" binding="netTcpBinding"/>
  <add scheme = "net.pipe" binding="netNamedPipeBinding"/>
  <add scheme = "net.msmq" binding="netMsmqBinding"/>
</protocolMapping>
...
</system.serviceModel>
```

Все, что нужно для использования этих стандартных привязок — указание базовых адресов в конфигурационном файле хоста. Чтобы увидеть это в действии, откройте проект `MagicEightBallServiceHost` в Visual Studio. После этого модифицируйте файл `*.config` хостинга, полностью удалив элемент `<endpoint>` для службы WCF и данные, связанные с MEX. В результате конфигурационный файл должен выглядеть примерно так:

```
<configuration>
  <system.serviceModel>
    <services>
      <service name = "MagicEightBallServiceLib.MagicEightBallService" >
        <host>
          <baseAddresses>
            <add baseAddress = "http://localhost:8080/MagicEightBallService"/>
          </baseAddresses>
        </host>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

Поскольку в `<baseAddress>` указан допустимый HTTP-адрес, хост автоматически использует привязку `basicHttpBinding`. Запустив хост снова, можно увидеть тот же самый вывод:

```
***** Console Based WCF Host *****

***** Host Info *****
Address: http://localhost:8080/MagicEightBallService
Binding: BasicHttpBinding
Contract: IEightBall
*****
The service is ready.
Press the Enter key to terminate service.
```

Здесь пока еще не включены данные MEX; это будет сделано чуть ниже с использованием другого упрощения, которое называется *конфигурациями стандартного поведения*. Однако давайте сначала разберемся, как открывать одиночную службу WCF с множеством привязок.

Открытие одной службы WCF, использующей множество привязок

Со времен своего первого выпуска WCF позволяет одному хосту открывать службу WCF с несколькими конечными точками.

Например, чтобы открыть службу `MagicEightBallService`, использующую привязки HTTP, TCP и именованного канала, необходимо просто добавить новые конечные точки в конфигурационный файл. После перезапуска хоста вся необходимая оснастка будет создана автоматически.

Это является огромным преимуществом по многим причинам. До появления WCF было трудно открыть одну службу с множеством привязок, т.к. каждый тип привязки (например, HTTP и TCP) имел собственную модель программирования. Тем не менее, возможность разрешить вызывающему коду выбирать наиболее подходящую привязку чрезвычайно удобна. Внутренние клиенты могут отдать предпочтение привязкам TCP, внешние клиенты (находящиеся за брандмауэром компании) — использовать для доступа HTTP, в то время как клиенты, находящиеся на той же машине, выберут именованный канал.

Чтобы сделать это в версиях, предшествующих .NET 4.5, в конфигурационном файле хоста требовалось определять несколько элементов `<endpoint>` вручную. Также для каждого протокола необходимо было определять множество элементов `<baseAddress>`. Однако теперь можно просто подготовить следующий конфигурационный файл:

```
<configuration> .
  <system.serviceModel>
    <services>
      <service name = "MagicEightBallServiceLib.MagicEightBallService" >
        <host>
          <baseAddresses>
            <add baseAddress = "http://localhost:8080/MagicEightBallService"/>
            <add baseAddress =
              "net.tcp://localhost:8099/MagicEightBallService"/>
          </baseAddresses>
        </host>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

Скомпилировав проект (для обновления развернутого файла `*.config`) и перезапустив хост, можно будет увидеть следующие данные конечной точки:

```
***** Console Based WCF Host *****

***** Host Info *****
Address: http://localhost:8080/MagicEightBallService
Binding: BasicHttpBinding
Contract: IEightBall

Address: net.tcp://localhost:8099/MagicEightBallService
Binding: NetTcpBinding
Contract: IEightBall
*****
The service is ready.
Press the Enter key to terminate service.
```

Теперь, когда служба WCF достижима из двух конечных точек, возникает вопрос: как клиент может производить выбор между ними? При генерации прокси клиентской стороны инструмент `Add Service Reference` назначит каждой представленной конечной точке строковое имя в файле `*.config` клиентской стороны. В коде можно передать корректное строковое имя конструктору прокси и иметь уверенность в том, что будет выбрана правильная привязка. Однако прежде чем сделать это, потребуются переустановка MEX для модифицированного конфигурационного файла хоста и научиться настраивать параметры стандартной привязки.

Изменение параметров привязки WCF

В случае указания ABC службы в коде C# (это рассматривается далее в главе) для изменения стандартных параметров привязки WCF необходимо просто модифицировать значения свойств объекта. Например, если нужно использовать BasicHttpBinding, но также изменить параметры таймаута, можно написать следующий код:

```
void ConfigureBindingInCode()
{
    BasicHttpBinding binding = new BasicHttpBinding();
    binding.OpenTimeout = TimeSpan.FromSeconds(30);
    ...
}
```

Параметры привязки всегда можно сконфигурировать декларативно. Например, .NET 3.5 позволяет построить конфигурационный файл хоста, в котором изменяется свойство OpenTimeout класса BasicHttpBinding, как показано ниже:

```
<configuration>
  <system.serviceModel>

    <bindings>
      <basicHttpBinding>
        <binding name = "myCustomHttpBinding"
                  openTimeout = "00:00:30" />
      </basicHttpBinding>
    </bindings>

    <services>
      <service name = "WcfMathService.MyCalc">
        <endpoint address = "http://localhost:8080/MyCalc"
                  binding = "basicHttpBinding"
                  bindingConfiguration = "myCustomHttpBinding"
                  contract = "WcfMathService.IBasicMath" />
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

Здесь мы имеем конфигурационный файл для службы по имени WcfMathService.MyCalc, которая поддерживает единственный интерфейс IBasicMath.

Обратите внимание, что раздел <bindings> позволяет определить именованный элемент <binding>, который изменяет параметры для заданной привязки. Внутри <endpoint> службы специфические параметры можно подключать с использованием атрибута bindingConfiguration.

Такая конфигурация хостинга по-прежнему работает и в последующих версиях .NET; однако в случае применения стандартной конечной точки подключить <binding> к <endpoint> не удастся. К счастью, параметрами стандартной конечной точки можно управлять, просто опустив атрибут name элемента <binding>. Например, в следующем фрагменте кода изменяются некоторые свойства объектов BasicHttpBinding и NetTcpBinding, используемые внутренне:

```
<configuration>
  <system.serviceModel>
    <services>
      <service name = "MagicEightBallServiceLib.MagicEightBallService" >
        <host>
          <baseAddresses>
            <add baseAddress = "http://localhost:8080/MagicEightBallService"/>
```

```

    <add baseAddress =
      "net.tcp://localhost:8099/MagicEightBallService"/>
    </baseAddresses>
  </host>
</service>
</services>

<bindings>
  <basicHttpBinding>
    <binding openTimeout = "00:00:30" />
  </basicHttpBinding>
  <netTcpBinding>
    <binding closeTimeout = "00:00:15" />
  </netTcpBinding>
</bindings>

</system.serviceModel>
</configuration>

```

Использование конфигурации стандартного поведения MEX

Инструмент генерации прокси должен сначала обнаружить композицию службы во время выполнения. В WCF такое обнаружение во время выполнения разрешается за счет включения MEX. В большинстве конфигурационных файлов хоста MEX должно быть включено (по крайней мере, во время разработки); к счастью, способ конфигурирования MEX редко изменяется, поэтому .NET 4.5 предлагается несколько удобных сокращений.

Наиболее полезным сокращением является готовая поддержка MEX. Не понадобится добавлять конечную точку MEX, определять именованное поведение службы MEX и затем подключать именованную привязку к службе (как это делалось в HTTP-версии MagicEightBallServiceHost); вместо этого теперь можно просто добавить следующий код:

```

<configuration>
  <system.serviceModel>
    <services>
      <service name = "MagicEightBallServiceLib.MagicEightBallService" >
        <host>
          <baseAddresses>
            <add baseAddress = "http://localhost:8080/MagicEightBallService"/>
            <add baseAddress =
              "net.tcp://localhost:8099/MagicEightBallService"/>
          </baseAddresses>
        </host>
      </service>
    </services>

    <bindings>
      <basicHttpBinding>
        <binding openTimeout = "00:00:30" />
      </basicHttpBinding>
      <netTcpBinding>
        <binding closeTimeout = "00:00:15" />
      </netTcpBinding>
    </bindings>

    <behaviors>
      <serviceBehaviors>
        <behavior>
          <!-- Для получения стандартного MEX
               не именуйте элемент <serviceMetadata> -->

```

```

    <serviceMetadata httpGetEnabled = "true"/>
  </behavior>
</serviceBehaviors>
</behaviors>

</system.serviceModel>
</configuration>

```

Трюк заключается в том, что элемент `<serviceMetadata>` больше не имеет атрибута `name` (также обратите внимание, что элементу `<service>` больше не нужен атрибут `behaviorConfiguration`). После этих корректировок появляется поддержка MEX во время выполнения. Чтобы удостовериться в этом, запустите хост (после компиляции и обновления конфигурационного файла) и введите следующий URL в браузере:

```
http://localhost:8080/MagicEightBallService
```

После этого можно щелкнуть на ссылке `wsdl` в верхней части веб-страницы и просмотреть WSDL-описание службы (см. рис. 25.6). В консольном окне хоста в выводе будут отсутствовать данные о конечной точке MEX, потому что она не определялась явно для `IMetadataExchange` в конфигурационном файле. Тем не менее, MEX включен, и можно приступить к построению клиентских прокси.

Обновление клиентского прокси и выбор привязки

Предполагая, что модифицированный хост скомпилирован и выполняется в фоновом режиме, теперь необходимо открыть клиентское приложение и обновить текущую ссылку на службу. Начните с открытия папки `Service Refreshes` (Ссылки на службу) в `Solution Explorer`. Затем щелкните правой кнопкой мыши на элементе `ServiceReference` и выберите в контекстном меню пункт `Update Service Reference` (Обновить ссылку на службу), как показано на рис. 25.8.

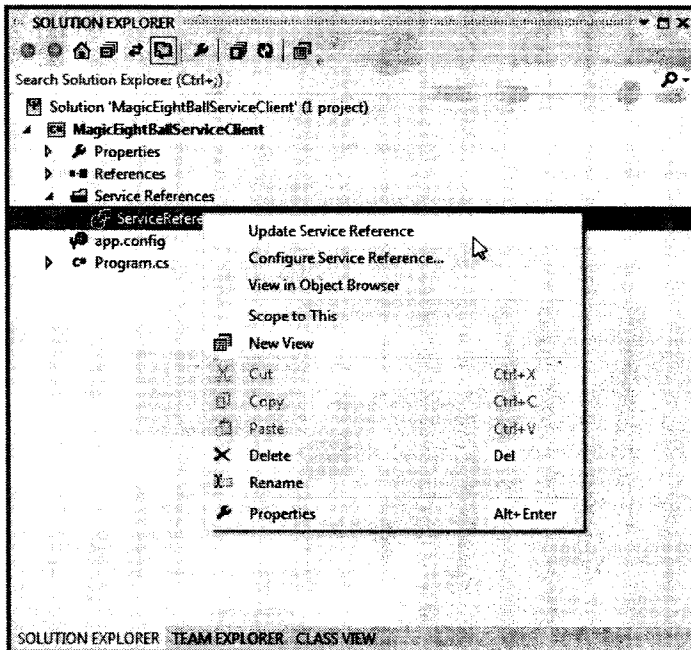


Рис. 25.8. Обновления прокси и файла *.config клиентской стороны

После этого в файле *.config клиентской стороны появятся на выбор две привязки: одна для HTTP и другая — для TCP. Каждой привязке назначено подходящее имя. Ниже приведен частичный листинг обновленного конфигурационного файла:

```
<configuration>
  <system.serviceModel>
    <bindings>
      <basicHttpBinding>
        <binding name = "BasicHttpBinding_IEightBall" ... />
      </basicHttpBinding>
      <netTcpBinding>
        <binding name = "NetTcpBinding_IEightBall" ... />
      </netTcpBinding>
    </bindings>
    ...
  </system.serviceModel>
</configuration>
```

Клиент может использовать эти имена при создании прокси-объекта для выбора желаемой привязки. Таким образом, если клиент предпочитает применять TCP, можно изменить код C# клиентской стороны следующим образом:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Ask the Magic 8 Ball *****\n");
    using (EightBallClient ball = new EightBallClient("NetTcpBinding_IEightBall"))
    {
        ...
    }
    Console.ReadLine();
}
```

Если же клиент вместо этого будет использовать привязку HTTP, можно написать так:

```
using (EightBallClient ball = new
    EightBallClient("BasicHttpBinding_IEightBall"))
{
    ...
}
```

На этом текущий пример, в котором был продемонстрирован ряд полезных сокращений, завершен. Указанные средства упрощают написание конфигурационных файлов хостинга. Далее будет показано, как использовать шаблон проекта WCF Service Library (Библиотека служб WCF).

Исходный код. Проект MagicEightBallServiceHTTPDefaultBindings доступен в подкаталоге Chapter 25.

Использование шаблона проекта WCF Service Library

Прежде чем строить более экзотическую службу WCF, которая будет взаимодействовать с базой данных AutoLot, созданной в главе 21, в следующем примере иллюстрируется ряд важных тем, включая преимущества шаблона проекта WCF Service Library, приложения WCF Test Client, редактора конфигурации WCF, хостинга служб WCF внутри

Windows-службы и асинхронных клиентских вызовов. Чтобы сосредоточить все внимание на новых концепциях, эта служба WCF также будет умышленно простой.

Построение простой математической службы

Для начала создайте новый проект WCF Service Library под названием MathService Library, выбрав соответствующую опцию в узле WCF диалогового окна New Project (см. рис. 25.2). Затем измените имя начального файла IService1.cs на IBasicMath.cs. После этого *удалите* весь код внутри пространства имен MathServiceLibrary и замените его следующим кодом:

```
[ServiceContract (Namespace="http://MyCompany.com")]
public interface IBasicMath
{
    [OperationContract]
    int Add(int x, int y);
}
```

Переименуйте файл Service1.cs в MathService.cs, удалите весь код внутри пространства имен MathServiceLibrary и реализуйте контракт службы, как показано ниже:

```
public class MathService : IBasicMath
{
    public int Add(int x, int y)
    {
        // Эмулировать длительный запрос.
        System.Threading.Thread.Sleep(5000);
        return x + y;
    }
}
```

Наконец, откройте файл App.config и замените все вхождения IService1 на IBasicMath, а также все вхождения Service1 на MathService. Обратите внимание, что этот файл *.config уже включает поддержку MEX; по умолчанию конечная точка службы использует протокол WsHttpBinding.

Тестирование службы WCF с помощью WcfTestClient.exe

Одно из преимуществ применения проекта WCF Service Library состоит в том, что при отладке или запуске библиотеки он читает настройки из файла *.config и применяет их для загрузки приложения WCF Test Client (WcfTestClient.exe). Это приложение с графическим пользовательским интерфейсом позволяет протестировать каждый член интерфейса службы по мере ее построения, вместо того, чтобы вручную строить хост/клиент, как это делалось ранее, просто для целей тестирования.

На рис. 25.9 показана тестовая среда для MathService. Обратите внимание, что двойной щелчок на методе интерфейса позволяет указать входные параметры и вызвать метод.

Эта утилита работает в готовом виде после создания проекта WCF Service Library, однако имейте в виду, что данный инструмент можно применять для тестирования любой службы WCF, запустив его в командной строке и указав конечную точку MEX. Например, если запустить приложение MagicEightBallServiceHost.exe, можно ввести следующую команду в окне командной строки разработчика:

```
wcftestclient http://localhost:8080/MagicEightBallService
```

После этого можно вызвать ObtainAnswerToQuestion() аналогичным образом.

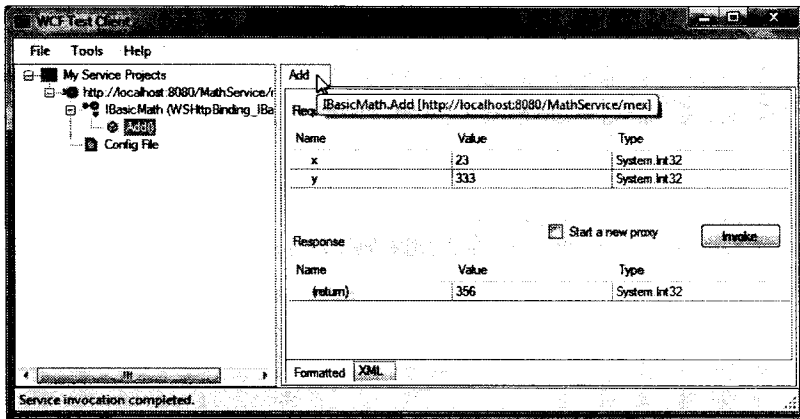


Рис. 25.9. Тестирование службы WCF с использованием WcfTestClient.exe

Изменение конфигурационных файлов с помощью SvcConfigEditor.exe

Другое преимущество применения проекта WCF Service Library связано с тем, что щелчком правой кнопкой мыши на файле App.config внутри Solution Explorer можно активизировать графический редактор конфигурирования службы (Service Configuration Editor), SvcConfigEditor.exe (рис. 25.10). Та же самая техника может применяться из клиентского приложения, которое ссылается на службу WCF.

Запустив этот инструмент, можно изменять данные в формате XML, используя дружественный пользовательский интерфейс. Существует много очевидных преимуществ от применения подобных инструментов для сопровождения файлов *.config. Первое (и главное): появляется уверенность в том, что сгенерированная разметка соответствует ожидаемому формату и свободна от опечаток. Второе: это замечательный способ увидеть правильные значения, которые могут быть присвоены каждому атрибуту. И третье: больше не понадобится вручную вводить данные XML.

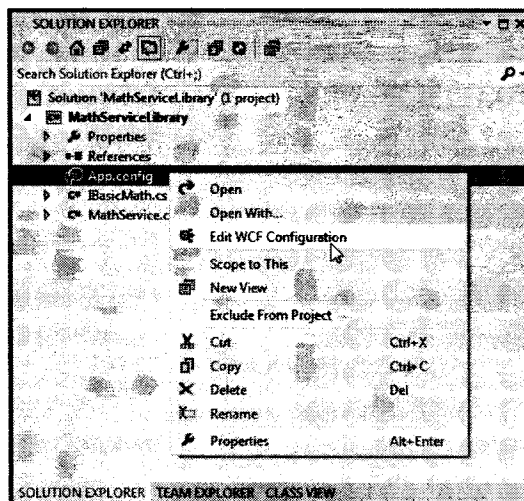


Рис. 25.10. Запуск графического редактора файлов *.config

На рис. 25.11 показан внешний вид редактора Service Configuration Editor. По правде говоря, описание всех интересных опций SvcConfigEditor.exe (интеграция с COM+, создание файлов *.config и т.п.) могло бы занять целую главу. Найдите время, чтобы изучить этот инструмент, пользуясь детальной справочной системой, которая вызывается нажатием клавиши <F1>.

На заметку! Утилита SvcConfigEditor.exe позволяет редактировать (или создавать) конфигурационные файлы, даже если не был выбран начальный проект WCF Service Library. Запустите этот инструмент в окне командной строки разработчика и воспользуйтесь пунктом меню File⇒Open (Файл⇒Открыть) для загрузки существующего файла *.config с целью редактирования.

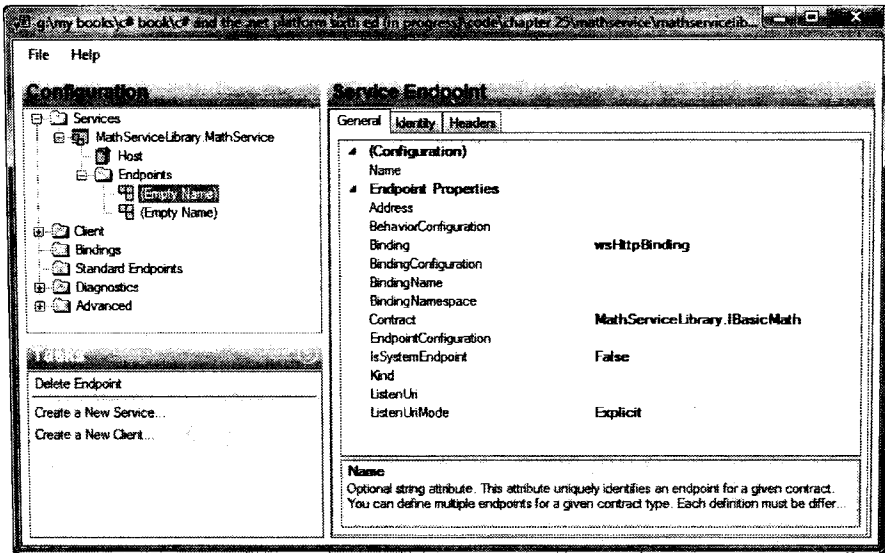


Рис. 25.11. Работа с редактором WCF Service Configuration Editor

Конфигурировать MathService больше не понадобится, поэтому можно перейти к задаче построения специального хоста.

Хостинг службы WCF внутри Windows-службы

Хостинг службы WCF в консольном приложении (или внутри настольного приложения с графическим пользовательским интерфейсом) — не идеальный выбор для сервера производственного уровня, учитывая, что хост всегда должен оставаться запущенным в фоновом режиме и готовым к обслуживанию клиентов. Даже если свернуть приложение-хост в панели задач Windows, все равно останется вероятность его случайного закрытия и, таким образом, разрыва всех соединений с клиентскими приложениями.

На заметку! Хотя и верно то, что настольное приложение Windows не обязано отображать главное окно, типичная программа *.exe требует взаимодействия с пользователем для загрузки и запуска. Служба Windows (описанная ниже) может быть сконфигурирована для запуска, даже если ни один пользователь не вошел в систему рабочей станции.

Если вы строите внутреннее приложение WCF, то еще одной альтернативой для хостинга библиотеки службы WCF является ее размещение в Windows-службе.

Преимущество такого решения состоит в том, что Windows-служба можно сконфигурировать для автоматического запуска при загрузке системы на целевой машине. Другое преимущество связано с тем, что Windows-служба работает невидимо, в фоновом режиме (в отличие от консольного приложения), и не требует участия пользователя (к тому же на машине хостинга не требуется установленный сервер IIS).

Давайте рассмотрим построение такого хоста. Начните с создания проекта Windows Service (Служба Windows) по имени MathWindowsServiceHost (рис. 25.12). Переименуйте начальный файл Service1.cs на MathService.cs, используя Solution Explorer.

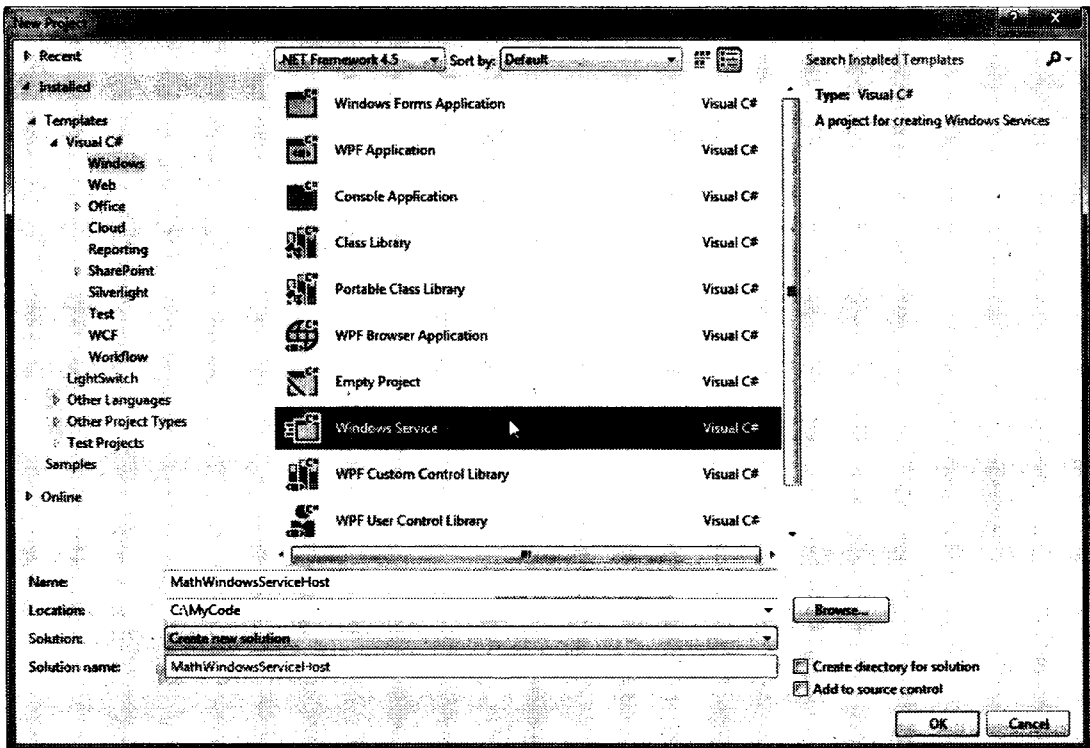


Рис. 25.12. Создание Windows-службы для хостинга службы WCF

Указание ABC в коде

Теперь, предполагая, что ссылки на сборки MathServiceLibrary.dll и System.ServiceModel.dll установлены, все, что осталось сделать — это использовать тип ServiceHost внутри методов OnStart() и OnStop() типа Windows-службы. Откройте файл кода для класса хоста службы (щелчком правой кнопкой мыши в визуальном конструкторе и выбором в контекстном меню пункта View Code (Просмотреть код)) и добавьте следующий код:

```
// Не забудьте импортировать следующие пространства имен:
using MathServiceLibrary;
using System.ServiceModel;

namespace MathWindowsServiceHost
{
    public partial class MathWinService: ServiceBase
    {
        // Переменная-член типа ServiceHost.
        private ServiceHost myHost;
```

```

public MathWinService()
{
    InitializeComponent();
}
protected override void OnStart(string[] args)
{
    // Проверить для подстраховки.
    if (myHost != null)
    {
        myHost.Close();
        myHost = null;
    }
    // Создать хост.
    myHost = new ServiceHost(typeof(MathService));
    // Указать ABC в коде.
    Uri address = new Uri("http://localhost:8080/MathServiceLibrary");
    WSHttpBinding binding = new WSHttpBinding();
    Type contract = typeof(IBasicMath);
    // Добавить эту конечную точку.
    myHost.AddServiceEndpoint(contract, binding, address);
    // Открыть хост.
    myHost.Open();
}
protected override void OnStop()
{
    // Остановить хост.
    if (myHost != null)
        myHost.Close();
}
}
}

```

Хотя ничто не мешает применять конфигурационный файл при построении хоста для службы WCF на основе Windows-службы, здесь (для разнообразия) вместо использования файла *.config конечная точка устанавливается программно с помощью классов Uri, WSHttpBinding и Type. После создания всех аспектов ABC хост программно информируется о них вызовом AddServiceEndpoint().

Если нужно информировать исполняющую среду о том, что необходимо получить доступ к каждой из привязок стандартных конечных точек, описанных в конфигурационном файле machine.config платформы .NET 4.5, можно упростить программную логику, указав базовый адрес при вызове конструктора ServiceHost. В этом случае не потребуется задавать ABC в коде вручную или вызывать AddServiceEndPoint(), а нужно просто вызвать AddDefaultEndpoints(). Взгляните на следующее изменение кода:

```

protected override void OnStart(string[] args)
{
    if (myHost != null)
    {
        myHost.Close();
    }
    // Создать хост и указать URL для привязки HTTP.
    myHost = new ServiceHost(typeof(MathService),
        new Uri("http://localhost:8080/MathServiceLibrary"));
    // Выбрать стандартные конечные точки.
    myHost.AddDefaultEndpoints();
    // Открыть хост.
    myHost.Open();
}

```

Включение MEX

Несмотря на то что включить MEX можно программно, сделаем это в конфигурационном файле. Добавьте новый файл `App.config` к проекту Windows-службы, внося в него следующие стандартные настройки MEX:

```
<?xml version = "1.0" encoding = "utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name = "MathServiceLibrary.MathService">
        </service>
      </services>
    <behaviors>
      <serviceBehaviors>
        <behavior>
          <serviceMetadata httpGetEnabled = "true"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

Создание программы установки для Windows-службы

Чтобы зарегистрировать Windows-службу в операционной системе, к проекту понадобится добавить программу установки, которая будет содержать необходимый код для регистрации службы. Для этого щелкните правой кнопкой мыши на поверхности визуального конструктора Windows-службы и выберите в контекстном меню пункт **Add Installer** (Добавить программу установки), как показано на рис. 25.13.

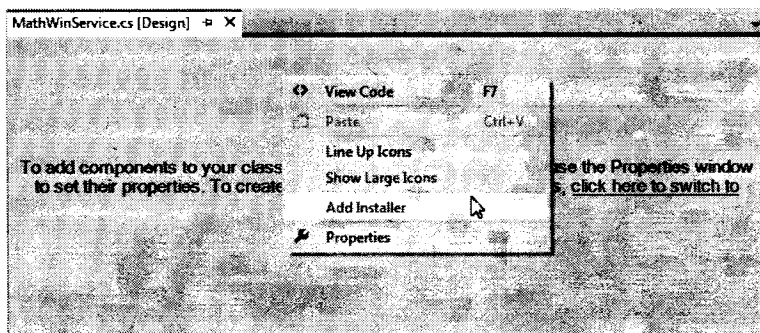


Рис. 25.13. Добавление программы установки для Windows-службы

В результате на поверхность визуального конструктора добавятся два новых компонента. Первый компонент (именуемый по умолчанию `serviceProcessInstaller1`) представляет элемент, который может установить новую Windows-службу на целевой машине. Выберите этот элемент в визуальном конструкторе и воспользуйтесь окном **Properties** (Свойства) для установки свойства `Account` в `LocalSystem` (рис. 25.14).

Второй компонент (под названием `serviceInstaller`) представляет тип, который будет устанавливать конкретную Windows-службу. В окне **Properties** измените значение свойства `ServiceName` на `MathService`, установите свойство `StartType` в `Automatic` и укажите дружественное описание зарегистрированной Windows-службы в свойстве `Description` (рис. 25.15). После этого можно компилировать полученное приложение.

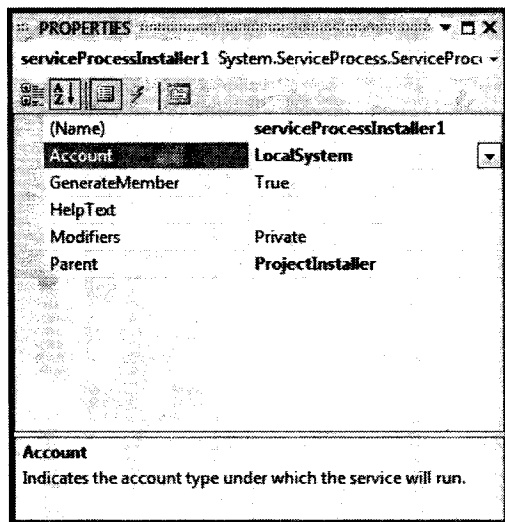


Рис. 25.14. Служба Windows должна запускаться от имени учетной записи LocalSystem

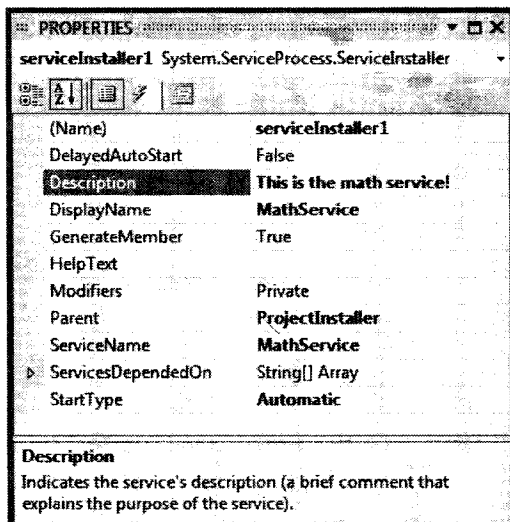


Рис. 25.15. Конфигурирование деталей, связанных с программой установки

Установка Windows-службы

Служба Windows может быть установлена на машине-хосте с помощью традиционной программы установки (такой как *.msi) или инструмента командной строки installutil.exe.

На заметку! Чтобы установить Windows-службу с использованием installutil.exe, окно командной строки разработчика должно быть запущено с административными привилегиями. Для этого щелкните правой кнопкой мыши на значке Developer Command Prompt (Командная строка разработчика) и выберите в контекстном меню пункт Запуск от имени администратора.

В окне командной строки разработчика перейдите в папку bin\Debug проекта MathWindowsServiceHost и введите следующую команду:

```
installutil MathWindowsServiceHost.exe
```

Предполагая, что установка прошла успешно, можно щелкнуть на значке Services (Службы) в папке Administrative Tools (Администрирование) панели управления Windows. В списке служб, упорядоченном по алфавиту, должно присутствовать дружественное имя службы MathService. Запустите эту службу на локальной машине, щелкнув на ссылке Start (Запустить), как показано на рис. 25.16.

Теперь, когда служба установлена и функционирует, остается последний шаг — создание клиентского приложения для потребления этой службы.

Исходный код. Проект MathWindowsServiceHost доступен в подкаталоге Chapter 25.

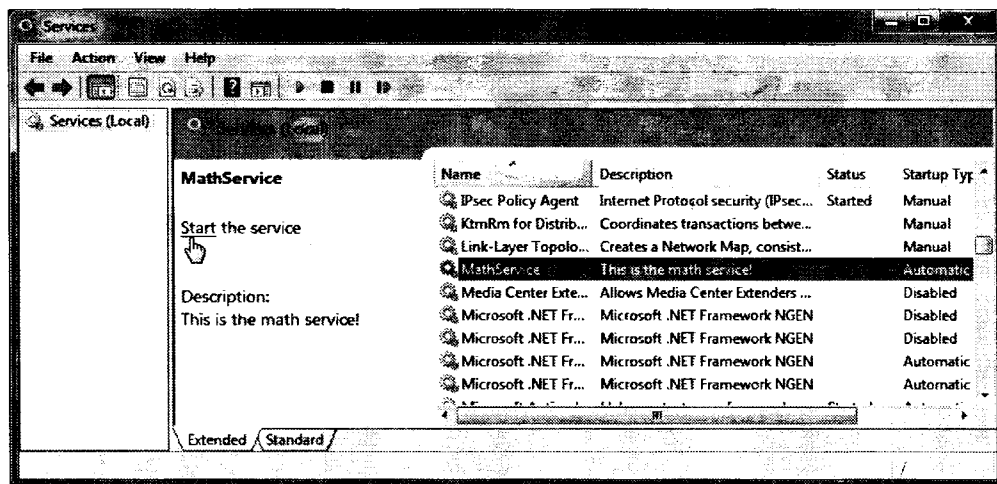
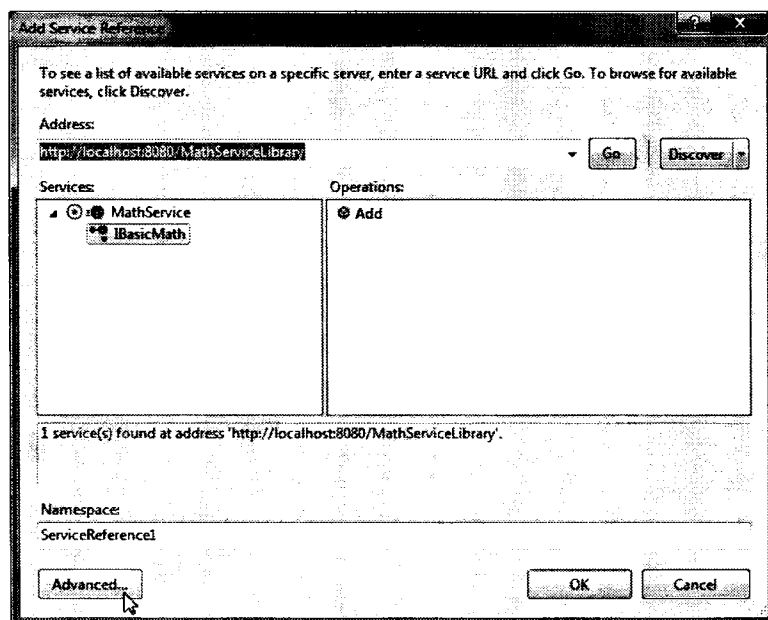


Рис. 25.16. Служба Windows, исполняющая роль хоста для службы WCF

Асинхронный вызов службы из клиента

Создайте новый проект консольного приложения по имени `MathClient` и укажите в качестве ссылки на службу работающую службу WCF (в настоящий момент размещенную в Windows-службе, функционирующей в фоновом режиме), выбрав для этого пункт меню `Project→Add Service Reference` (Проект→Добавить ссылку на службу) в Visual Studio (понадобится ввести URL в поле адреса, который должен выглядеть как `http://localhost:8080/MathServiceLibrary`). Однако не щелкайте пока на кнопке `OK`! Обратите внимание на кнопку `Advanced` (Дополнительно) в нижнем левом углу диалогового окна `Add Service Reference` (Добавление ссылки на службу), показанном на рис. 25.17.

Рис. 25.17. Ссылка на службу `MathService` и возможность конфигурирования расширенных установок

Щелкните на этой кнопке, чтобы увидеть дополнительные установки конфигурации прокси (рис. 25.18). Используя это диалоговое окно, можно генерировать код, который позволит вызывать удаленные методы в асинхронном режиме, если выбран переключатель *Generate asynchronous operations* (Генерировать асинхронные операции). Попробуйте сделать это.

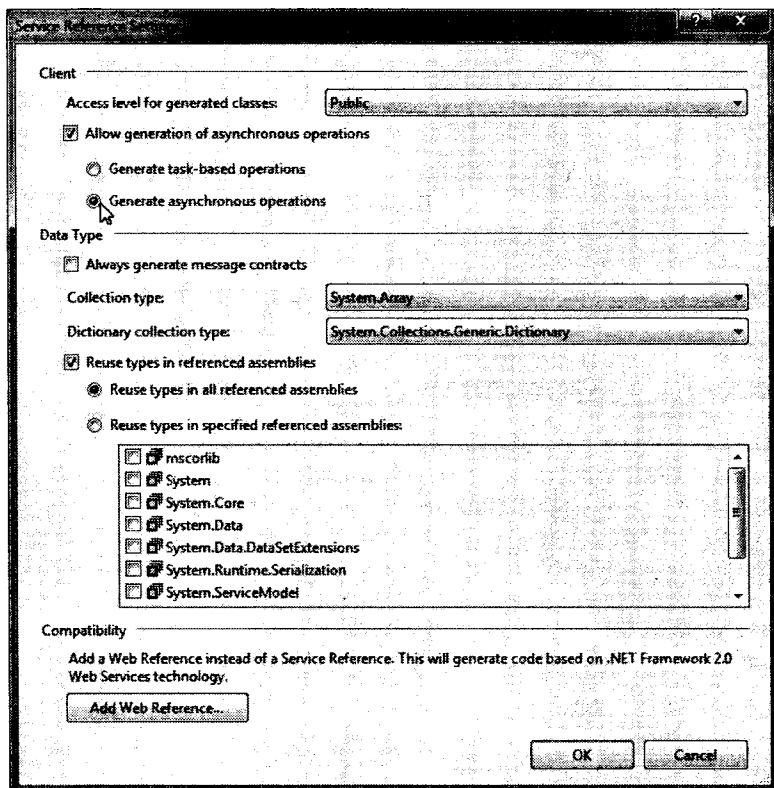


Рис. 25.18. Дополнительные опции конфигурации прокси клиентской стороны

К этому моменту код прокси содержит дополнительные методы, которые позволяют вызывать каждый член контракта службы с использованием ожидаемого шаблона асинхронного вызова *Begin/End*, описанного в главе 19. Ниже показана простая реализация, в которой вместо строго типизированного делегата *AsyncCallback* применяется лямбда-выражение:

```
using System;
using MathClient.ServiceReferencel;
...

namespace MathClient
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** The Async Math Client *****\n");
            using (BasicMathClient proxy = new BasicMathClient())
            {
                proxy.Open();
            }
        }
    }
}
```

```
// Суммировать числа в асинхронном режиме, используя лямбда-выражение.
IAsyncResult result = proxy.BeginAdd(2, 3,
    ar =>
    {
        Console.WriteLine("2 + 3 = {0}", proxy.EndAdd(ar));
    },
    null);

while (!result.IsCompleted)
{
    Thread.Sleep(200);
    Console.WriteLine("Client working...");
}

Console.ReadLine();
}
```

Исходный код. Проект MathClient доступен в подкаталоге Chapter 25.

Проектирование контрактов данных WCF

В последнем примере этой главы рассматривается конструирование *контрактов данных WCF*. В ранее созданных службах WCF были определены очень простые методы, оперирующие примитивными типами данных CLR. При использовании любого типа привязки HTTP (например, `basicHttpBinding` или `wsHttpBinding`) входящие и исходящие простые типы данных автоматически форматируются в XML-элементы. Кстати говоря, если применяется привязка на основе TCP (такая как `netTcpBinding`), то параметры и возвращаемые значения простых типов данных передаются в компактном двоичном формате.

На заметку! Исполняющая среда WCF также автоматически кодирует любой тип, помеченный атрибутом `[Serializable]`. Однако это не является предпочтительным способом определения контрактов WCF и предназначено только для обеспечения обратной совместимости.

Однако при определении контрактов служб, которые используют специальные классы в качестве параметров или возвращаемых значений, рекомендуется моделировать эти данные с применением контрактов данных WCF. Выражаясь просто, контракт данных — это тип, снабженный атрибутом `[DataContract]`. Аналогично каждое поле, которое планируется использовать как часть предполагаемого контракта, должно помечаться атрибутом `[DataMember]`.

На заметку! В ранних версиях платформы .NET использование атрибутов `[DataContract]` и `[DataMember]` было обязательным для обеспечения корректного представления специальных типов данных. Это требование в Microsoft было ослаблено; формально выражаясь, вы не обязаны применять указанные атрибуты в специальных типах данных; тем не менее, в .NET это считается рекомендуемым подходом.

Использование веб-ориентированного шаблона проекта WCF Service

Наша следующая служба WCF позволит внешним вызывающим клиентам взаимодействовать с базой данных AutoLot, созданной в главе 21. Более того, финальная служба WCF будет построена с использованием веб-ориентированного шаблона проекта WCF Service и размещаться в IIS.

Для начала запустите Visual Studio (с правами администратора) и выберите пункт меню File⇒New⇒Web Site (Файл⇒Создать⇒Веб-сайт). Укажите тип проекта WCF Service и удостоверьтесь, что в списке Web Location (Веб-местоположение) выбран вариант HTTP (что приведет к установке службы в IIS). Откройте службу их следующего URI:

`http://localhost/AutoLotWCFService`

На рис. 25.19 показан сконфигурированный проект.

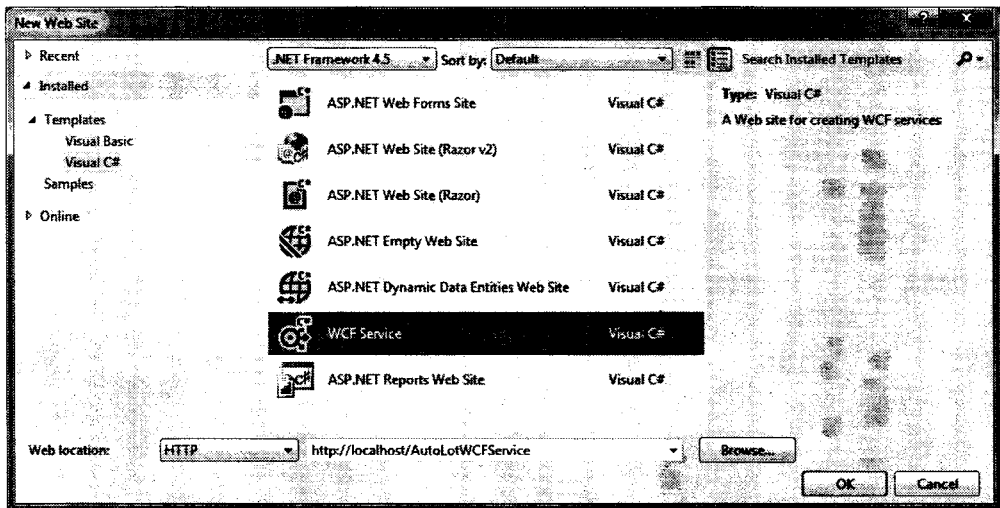


Рис. 25.19. Создание веб-ориентированной службы WCF

После этого установите ссылку на сборку AutoLotDAL.dll, созданную в главе 21 (через пункт меню Website⇒Add Reference (Веб-сайт⇒Добавить ссылку)). Будет предоставлен некоторый начальный код (расположенный в папке Add_Code), который необходимо удалить. Первым делом, переименуйте исходный файл IService.cs в IAutoLotService.cs и внутри переименованного файла определите начальный контракт службы:

```
[ServiceContract]
public interface IAutoLotService
{
    [OperationContract]
    void InsertCar(int id, string make, string color, string petname);

    [OperationContract]
    void InsertCar(InventoryRecord car);

    [OperationContract]
    InventoryRecord[] GetInventory();
}
```

В этом интерфейсе определены три метода, один из которых возвращает массив объектов типа InventoryRecord (пока еще не созданного). Вы можете вспомнить, что

метод `GetInventory()` класса `InventoryDAL` просто возвращает объект `DataTable`, из-за чего возникает вопрос: почему бы методу `GetInventory()` создаваемой службы не делать то же самое?

Хотя и можно было бы вернуть `DataTable` из метода службы WCF, вспомните, что технология WCF обязана следовать принципам SOA, одним из которых является программирование на основе контрактов, а не реализаций. Поэтому вместо возврата внешнему клиенту специфичного для .NET типа `DataTable` мы вернем специальный контракт данных (`InventoryRecord`), который будет корректно выражен в документе WSDL в независимой манере.

Также обратите внимание, что в показанном ранее интерфейсе определен перегруженный метод по имени `InsertCar()`. Первая версия принимает четыре входных параметра, а вторая — один параметр типа `InventoryRecord`. Контракт данных `InventoryRecord` может быть определен следующим образом:

```
[DataContract]
public class InventoryRecord
{
    [DataMember]
    public int ID;
    [DataMember]
    public string Make;
    [DataMember]
    public string Color;
    [DataMember]
    public string PetName;
}
```

Если реализовать интерфейс `IAutoLotService` в таком виде, построить хост и попытаться вызвать эти методы на стороне клиента, возникнет исключение времени выполнения. Причина в том, что одно из требований описания WSDL связано с тем, что каждый метод, открытый заданной конечной точкой, должен быть *уникально именован*. Таким образом, хотя перегрузка методов замечательно работает в контексте языка C#, современные спецификации веб-служб не допускают существования двух методов с одним и тем же именем `InsertCar()`.

К счастью, атрибут `[OperationContract]` поддерживает свойство `Name`, которое позволяет указать, как метод C# будет представлен в описании WSDL. С учетом этого, модифицируйте вторую версию `InsertCar()` следующим образом:

```
public interface IAutoLotService
{
    ...
    [OperationContract(Name = "InsertCarWithDetails")]
    void InsertCar(InventoryRecord car);
}
```

Реализация контракта службы

Теперь переименуйте `Service.cs` в `AutoLotService.cs`. Тип `AutoLotService` реализует интерфейс `IAutoLotService`, как показано ниже (не забудьте импортировать пространства имен `AutoLotConnectedLayer` и `System.Data` в этот файл кода и должным образом скорректировать строку соединения):

```
using AutoLotConnectedLayer;
using System.Data;

public class AutoLotService : IAutoLotService
{
```

```

private const string ConnString =
    @"Data Source=(local)\SQLEXPRESS;Initial Catalog=AutoLot"+
    ";Integrated Security=True";

public void InsertCar(int id, string make, string color, string petname)
{
    InventoryDAL d = new InventoryDAL();
    d.OpenConnection(ConnString);
    d.InsertAuto(id, color, make, petname);
    d.CloseConnection();
}

public void InsertCar(InventoryRecord car)
{
    InventoryDAL d = new InventoryDAL();
    d.OpenConnection(ConnString);
    d.InsertAuto(car.ID, car.Color, car.Make, car.PetName);
    d.CloseConnection();
}

public InventoryRecord[] GetInventory()
{
    // Сначала получить DataTable из базы данных.
    InventoryDAL d = new InventoryDAL();
    d.OpenConnection(ConnString);
    DataTable dt = d.GetAllInventoryAsDataTable();
    d.CloseConnection();

    // Теперь создать List<T> для хранения полученных данных.
    List<InventoryRecord> records = new List<InventoryRecord>();

    // Скопировать таблицу данных в список List<> специальных контрактов.
    DataTableReader reader = dt.CreateDataReader();
    while (reader.Read())
    {
        InventoryRecord r = new InventoryRecord();
        r.ID = (int)reader["CarID"];
        r.Color = ((string)reader["Color"]);
        r.Make = ((string)reader["Make"]);
        r.PetName = ((string)reader["PetName"]);
        records.Add(r);
    }

    // Трансформировать List<T> в массив элементов типа InventoryRecord.
    return (InventoryRecord[])records.ToArray();
}

```

В приведенном выше коде нет ничего особенного. Для простоты мы жестко закодировали значение строки соединения (которую может понадобиться скорректировать согласно текущим настройкам машины) вместо сохранения ее в файле Web.config. Учитывая, что библиотека доступа к данным выполняет всю реальную работу по взаимодействию с базой данных AutoLot, все, что потребуется сделать — это передать входные параметры методу `InsertAuto()` класса `InventoryDAL`. Единственное, что здесь представляет интерес — это отображение значений объекта `DataTable` на обобщенный список элементов типа `InventoryRecord` (посредством `DataTableReader`) с последующей трансформацией `List<T>` в массив объектов типа `InventoryRecord`.

Роль файла *.svc

При создании веб-ориентированной службы WCF обнаруживается, что проект содержит специфичный файл с расширением *.svc. Этот конкретный файл необходим любой службе WCF, развернутой в IIS; в нем описано имя и местоположение реализации службы внутри точки установки. Поскольку имена начального файла и типов WCF были изменены, потребуется модифицировать содержимое файла Service.svc:

```
<%@ ServiceHost Language="C#" Debug="true"
    Service="AutoLotService" CodeBehind="~/App_Code/AutoLotService.cs" %>
```

Содержимое файла Web.config

Файл Web.config службы WCF, созданной для HTTP, использует ряд упрощений WCF, рассмотренных ранее в этой главе. Как будет подробно описано далее в этой книге при рассмотрении ASP.NET, файл Web.config служит той же цели, что и файл *.config исполняемой программы; тем не менее, он также управляет рядом специфических веб-настроек. Например, обратите внимание, что механизм MEX включен, и не нужно указывать специальный элемент <endpoint> вручную:

```
<configuration>
...
<system.serviceModel>
  <behaviors>
    <serviceBehaviors>
      <behavior>
        <!-- Чтобы избежать раскрытия информации метаданных,
              установите следующее значение в false и удалите
              конечную точку метаданных перед развертыванием -->
        <serviceMetadata httpGetEnabled="true" httpsGetEnabled="true" />
        <!-- Для получения деталей исключений при сбоях в целях отладки
              установите следующее значение в true.
              Перед развертыванием установите его снова в false
              во избежание раскрытия информации об исключении -->
        <serviceDebug includeExceptionDetailInFaults="false"/>
      </behavior>
    </serviceBehaviors>
  </behaviors>
  <serviceHostingEnvironment aspNetCompatibilityEnabled="true"
                             multipleSiteBindingsEnabled="true" />
</system.serviceModel>
...
</configuration>
```

Тестирование службы

Теперь для тестирования службы можно построить клиент любого рода, в том числе и передать конечную точку файла *.svc в приложение WcfTestClient.exe:

```
WcfTestClient http://localhost/AutoLotWCFService/Service.svc
```

Чтобы создать специальное клиентское приложение, воспользуйтесь диалоговым окном Add Service Reference (Добавить ссылку на службу), как это делалось ранее в главе, в примерах проектов MagicEightBallServiceClient и MathClient.

На этом рассмотрение API-интерфейса Windows Communication Foundation завершено. Разумеется, эта тема слишком обширна, чтобы раскрыть ее полностью в одной ознакомительной главе, однако, разобравшись с изложенным здесь материалом, можно продолжить изучение WCF самостоятельно. Исчерпывающие сведения по WCF приведены в документации .NET Framework 4.5 SDK.

Резюме

В этой главе вы ознакомились с API-интерфейсом Windows Communication Foundation (WCF), который является частью библиотеки базовых классов, начиная с версии .NET 3.0. Как здесь объяснялось, основная мотивация, лежащая в основе WCF, состоит в предоставлении унифицированной объектной модели, которая открывает ряд (ранее несвязанных) API-интерфейсов распределенных вычислений, собранных воедино. В начале главы было показано, что служба WCF представлена за счет указания адресов, привязок и контрактов (обозначаемых аббревиатурой ABC).

Как вы видели, типичное приложение WCF предусматривает использование трёх взаимосвязанных сборок. В первой из них определены контракты и типы, которые представляют функциональность службы. Эта сборка затем размещается в специальной исполняемой программе, виртуальном каталоге IIS либо Windows-службе. И, наконец, клиентская сборка использует генерируемый файл кода, определяющий тип прокси (и настройки в конфигурационном файле приложения) для взаимодействия с удаленным типом.

В главе также рассматривалось применение ряда инструментов программирования WCF, таких как `SvcConfigEditor.exe` (который позволяет модифицировать содержимое файлов `*.config`), приложение `WcfTestClient.exe` (для быстрого тестирования служб WCF) и различные шаблоны проектов Visual Studio. Кроме того, было описано множество упрощений конфигурации, включая стандартные конечные точки и поведение.

Введение в Windows Workflow Foundation

Платформа .NET поддерживает программную модель под названием Windows Workflow Foundation (WF). Этот API-интерфейс позволяет моделировать, конфигурировать, проводить мониторинг и выполнять рабочие потоки (которые служат для моделирования бизнес-процессов), используемые внутри конкретного приложения .NET. Рабочие потоки моделируются (по умолчанию) с применением декларативной, основанной на XML грамматики XAML, где данные, используемые рабочим потоком, являются “почетными гражданами”.

Для новичков в мире WF глава начинается с определения роли бизнес-процессов и описания их связи с API-интерфейсом WF. Кроме того, будет представлена концепция действия WF, общие типы рабочих потоков, а также различные шаблоны проектов и инструменты программирования. После ознакомления с основами будет построено несколько примеров программ, которые проиллюстрируют применение программной модели WF для установки бизнес-процессов, которые выполняются под неусыпным надзором исполняющей среды WF.

На заметку! Полное описание API-интерфейса WF невозможно уместить в единственную ознакомительную главу. Более глубокое изложение дается в книге *Pro WF 4.5* (Apress, 2012 г.).

Определение бизнес-процесса

Любое реальное приложение должно иметь возможность моделировать различные бизнес-процессы. Бизнес-процесс — это концептуально объединенная группа задач, которая логически работает как единое целое. Например, предположим, что строится приложение, позволяющее приобретать автомобили через Интернет. Когда пользователь отправляет заказ, приводится в действие большое количество действий. Начать можно с проверки платежеспособности. Если пользователь прошел такую проверку, можно запустить транзакцию базы данных, чтобы исключить элемент из таблицы Inventory (склад), добавить новый элемент в таблицу Orders (заказы) и обновить информацию счета заказчика. После завершения транзакции базы данных может понадобиться отправить покупателю сообщение электронной почты с подтверждением, а затем вызвать удаленную службу для передачи заказа дилеру. Все вместе эти задачи представляют единый бизнес-процесс.

Исторически сложилось, что моделирование бизнес-процессов являлось еще одной деталью, которую должны были принимать во внимание программисты, часто за счет написания специального кода, который гарантирует не только правильное моделиро-

вание бизнес-процесса, но также его корректное выполнение внутри самого приложения. Например, может потребоваться написать код, учитывающий возможные сбои, выполняющий трассировку и протоколирование (чтобы видеть текущее состояние бизнес-процесса), поддерживающий постоянство (для сохранения состояния длительно выполняющегося процесса) и т.д. Понятно, что создание такой инфраструктуры с нуля требует массы времени и ручной работы.

Даже если команда разработчиков на самом деле построит специальную инфраструктуру бизнес-процессов для своих приложений, на этом их работа не завершится. Низкоуровневую кодовую базу C# не получится легко объяснить членам команды, не являющимся программистами, которые также нуждаются в понимании бизнес-процесса. Суровая правда заключается в том, что эксперты предметной области, менеджеры, продавцы и члены команды графического дизайна часто не говорят на языке кода. Учитывая это, программистам понадобятся инструменты моделирования (вроде Microsoft Visio, лекционной доски и т.п.) для графического представления необходимых процессов, используя общепонятную терминологию. Очевидная проблема здесь в том, что придется постоянно поддерживать в согласованном состоянии две сущности: если изменяется код, то нужно обновить диаграмму, а если меняется диаграмма, то придется обновить код.

Более того, при построении изолированных приложений, использующих стопроцентный кодовый подход, кодовая база очень слабо представляет внутренний “поток” приложения. Например, типичная программа .NET может состоять из сотен специальных типов (не говоря уже о многочисленных типах из библиотек базовых классов). Хотя программисты могут представлять себе, какие объекты обращаются к другим объектам, сам код далеко не в состоянии заменить живую документацию, которая объяснит последовательность действий простым человеческим языком. Хотя команда разработчиков может подготовить внешнюю документацию и графики рабочих потоков, опять возникает упомянутая проблема с несколькими представлениями одного и того же процесса.

Роль WF

В сущности API-интерфейс Windows Workflow Foundation позволяет программистам декларативно проектировать бизнес-процессы, используя предварительно разработанный набор действий. Поэтому вместо применения только набора специальных сборок для представления заданного бизнес-действия и всей необходимой инфраструктуры можно пользоваться визуальными конструкторами WF среды Visual Studio для создания бизнес-процессов на этапе проектирования. Благодаря этому, WF позволяет построить скелет бизнес-процесса, который затем может быть наполнен конкретным кодом, где это необходимо.

При программировании с применением API-интерфейса WF для представления всего бизнес-процесса и определяющего его кода может использоваться единственная сущность. В дополнение к дружественному визуальному представлению этого процесса применяется единственный документ WF, поэтому больше не придется беспокоиться о возможном рассогласовании нескольких документов. Еще лучше то, что документ WF наглядно иллюстрирует сам процесс. С минимальной помощью специалиста даже представители нетехнического персонала быстро понимают то, что моделирует визуальный конструктор WF.

Построение простого рабочего потока

При построении приложений с рабочими потоками вы, несомненно, отметите, что процесс разработки выглядит совершенно иначе, чем для типичного приложения .NET.

Например, вплоть до этого момента каждый пример кода начинался с создания нового рабочего пространства проекта (чаще всего проекта консольного приложения) и предусматривал написание кода для представления программы в целом. Приложение WF также содержит специальный код; однако, вдобавок к этому, производится встраивание непосредственно в сборку модели самого бизнес-процесса.

Еще один аспект WF, который существенно отличается от других видов приложений .NET, заключается в том, что подавляющее большинство рабочих потоков будут моделироваться в декларативной манере, с использованием грамматики на основе XML, называемой XAML. По большей части не придется непосредственно писать код разметки, поскольку IDE-среда Visual Studio сделает это автоматически, в процессе работы с инструментами проектирования WF. В этом состоит значительное отличие от предыдущей версии API-интерфейса WF, где в качестве стандартного средства моделирования рабочего потока применялся код C#.

На заметку! Имейте в виду, что диалект XAML, используемый в WF, не идентичен диалекту XAML, применяемому в WPF. Вы ознакомитесь с синтаксисом и семантикой XAML для WPF в главе 27, поскольку в отличие от WF для XAML, там довольно часто приходится непосредственно редактировать сгенерированный визуальным конструктором XAML-код.

Чтобы приступить к работе с рабочими потоками, откройте Visual Studio. В диалоговом окне New Project (Новый проект) выберите проект Workflow Console Application (Консольное приложение рабочего потока) и назначьте ему имя FirstWorkflowExampleApp (рис. 26.1).

Теперь взгляните на рис. 26.2, на котором показана начальная диаграмма рабочего потока, сгенерированная Visual Studio. Как видите, здесь мало что происходит, помимо появления в визуальном конструкторе сообщения, предлагающего поместить действия на поверхность проектирования.

Для этого простого тестового рабочего потока откройте панель инструментов Visual Studio и найдите действие WriteLine в разделе Primitives (Примитивы), как показано на рис. 26.3. Найдя это действие, перетащите его на область с надписью Drop activity here (Поместите сюда действие) и в поле редактирования Text введите сообщение в двойных кавычках. На рис. 26.4 показан один из возможных рабочих потоков.

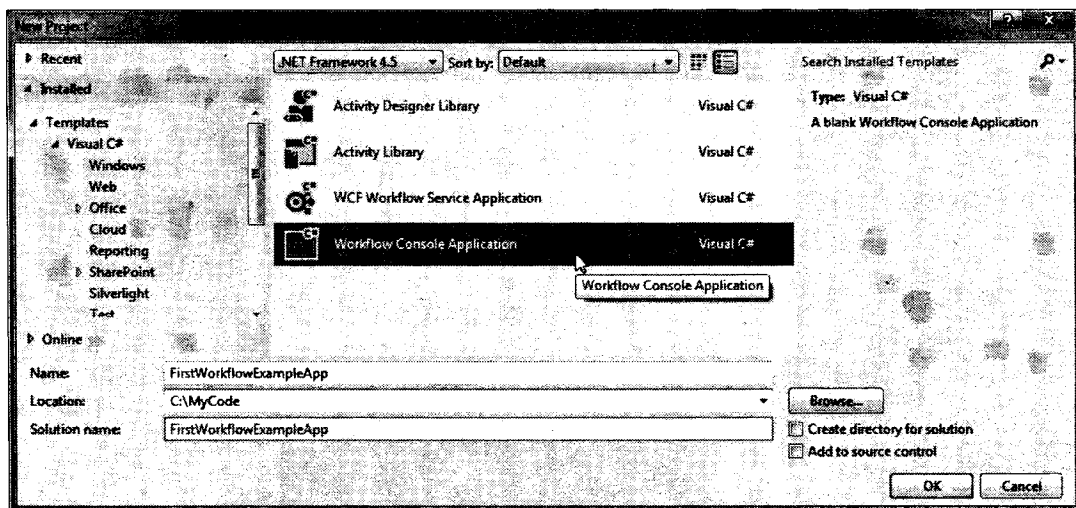


Рис. 26.1. Создание нового консольного приложения рабочего потока

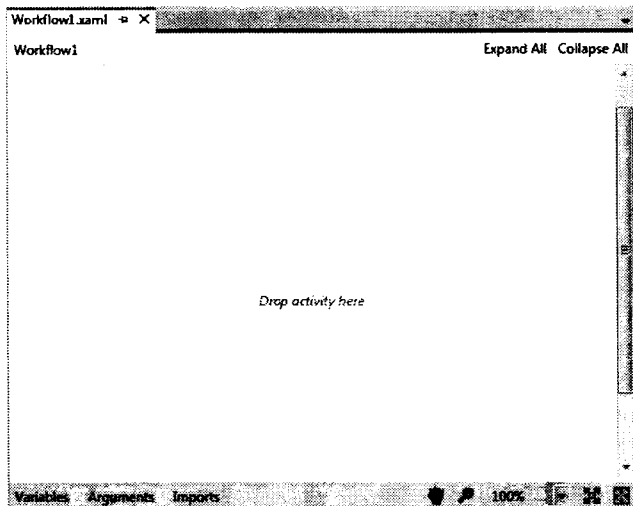


Рис. 26.2. Визуальный конструктор рабочего потока — это контейнер для действий, моделирующих бизнес-процесс

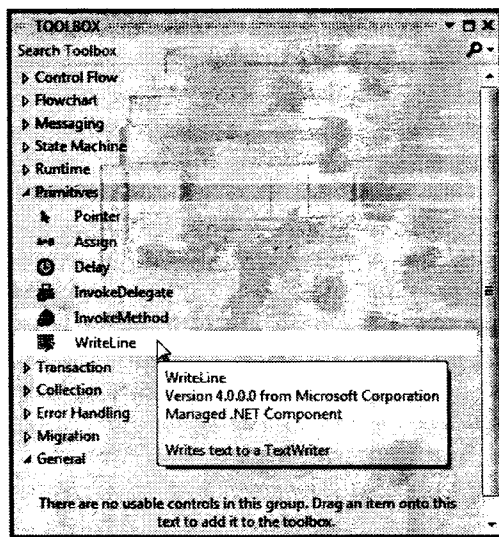


Рис. 26.3. Панель инструментов отображает все стандартные действия WF

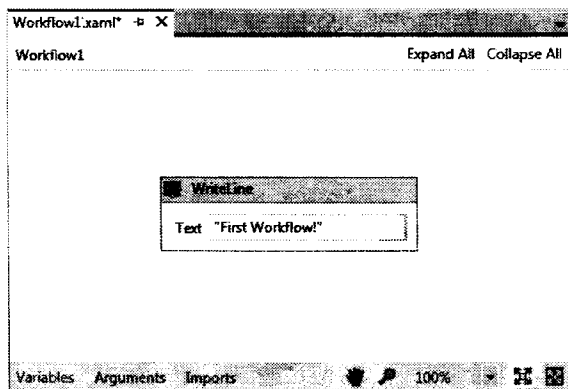


Рис. 26.4. Действие WriteLine отображает текст в TextWriter, в данном случае — на консоли

Важно понимать, что инфраструктура WF — это намного больше, чем просто симпатичный визуальный конструктор, который позволяет моделировать действия бизнес-процесса. При построении диаграммы WF разметка может быть всегда расширена для представления поведения процесса во время выполнения. Фактически, при желании можно вообще избежать использования XAML и писать рабочие потоки исключительно на C#. Однако в этом случае мы вернемся к той же основополагающей проблеме: наличию кода, который не понятен нетехническому персоналу. В любом случае запуск приложения на этом этапе приводит к отображению в окне консоли указанного ранее сообщения:

```
First Workflow!
Press any key to continue . . .
```

Выглядит неплохо, но пока непонятно, что же запустило этот рабочий поток? И как можно обеспечить, чтобы консольное приложение оставалось в состоянии выполнения достаточно долго для завершения рабочего потока? Ответы на эти вопросы требуют понимания исполняющего механизма рабочего потока.

Исполняющая среда рабочих потоков

Следующее, что нужно понять — API-интерфейс WF также включает исполняющий механизм, который загружает, выполняет, выгружает и иными способами манипулирует рабочим потоком. Исполняющий механизм WF может быть развернут внутри любого домена приложения .NET; однако имейте в виду, что отдельный домен приложения может иметь только один запущенный экземпляр механизма WF.

Вспомните из главы 17, что домен приложения — это раздел внутри процесса Windows, который играет роль хоста для приложения .NET и любых внешних библиотек кода. Как таковой, механизм WF может быть встроен в простую консольную программу, настольное приложение с графическим пользовательским интерфейсом (Windows Forms или WPF) или же открываться через службу WCF.

На заметку! Шаблон проекта WCF Workflow Service Application (Приложение службы WCF рабочих потоков) — замечательная стартовая точка, если необходимо построить службу WCF (см. главу 25), которая внутренне использует рабочие потоки.

При моделировании бизнес-процесса, который должен использоваться широким разнообразием систем, также есть возможность написания рабочего потока внутри проекта библиотеки классов C# в библиотеке действий рабочих потоков. В этом случае новые приложения смогут просто ссылаться на библиотеку *.dll, чтобы повторно использовать заранее предопределенную коллекцию бизнес-процессов. Это очевидно полезно, когда не хочется многократно пересоздавать одни и те же рабочие потоки.

Хостинг рабочего потока с использованием класса WorkflowInvoker

Хост-процесс исполняющей среды WF может взаимодействовать с этой исполняющей средой посредством различных приемов. Простейший способ предусматривает применение класса WorkflowInvoker из пространства имен System.Activities. Этот класс позволяет запустить рабочий поток с помощью единственной строки кода. Открыв файл Program.cs текущего проекта Workflow Console Application, вы увидите следующий метод Main():

```
static void Main(string[] args)
{
    // Создать и кешировать определение рабочего потока.
    Activity workflow1 = new Workflow1();
    WorkflowInvoker.Invoke(workflow1);
}
```

Использовать `WorkflowInvoker` очень удобно, когда нужно просто запустить рабочий поток и не следить за ним далее. Метод `Invoke()` будет выполнять рабочий поток в синхронном блокирующем режиме. Вызывающий поток блокируется до тех пор, пока весь рабочий поток не завершится, либо не будет прерван принудительно. Поскольку метод `Invoke()` является синхронным вызовом, гарантируется, что весь рабочий поток на самом деле будет завершен до окончания `Main()`. В действительности, если добавить какой-нибудь код после вызова `WorkflowInvoker.Invoke()`, он будет выполнен, только когда рабочий поток завершится (или в худшем случае будет принудительно прерван):

```
static void Main(string[] args)
{
    // Создать и кешировать определение рабочего потока.
    Activity workflow1 = new Workflow1();
    WorkflowInvoker.Invoke(workflow1);

    Console.WriteLine("Thanks for playing");
}
```

Передача аргументов рабочему потоку с использованием класса *WorkflowInvoker*

Когда хост-процесс запускает рабочий поток, ему очень часто нужно передавать специальные стартовые аргументы. Например, предположим, что пользователю программы необходимо дать возможность указывать сообщение, которое должно отображаться в действии `WriteLine` вместо жестко закодированного текстового сообщения. В нормальном коде C# можно было бы создать специальный конструктор класса для приема таких аргументов. Однако рабочий поток всегда создается с применением стандартного конструктора! Более того, большинство рабочих потоков определено с помощью только XAML, а не процедурного кода.

Оказывается, метод `Invoke()` имеет несколько перегрузок, одна из которых позволяет передавать аргументы рабочему потоку при запуске. Эти аргументы представлены с использованием переменной `Dictionary<string, object>`, содержащей набор пар "имя/значение", которые будут применяться для установки идентично именованных (и типизированных) переменных-аргументов в самом рабочем потоке.

Определение аргументов с использованием визуального конструктора рабочих потоков

Для определения аргументов, которые получают входные данные словаря, воспользуемся визуальным конструктором рабочих потоков. В окне `Solution Explorer` щелкните правой кнопкой мыши на `Workflow1.xaml` и выберите в контекстном меню пункт `View Designer` (Просмотреть визуальный конструктор). Обратите внимание на кнопку `Arguments` (Аргументы) в нижней части визуального конструктора. Щелкните на ней и в открывшемся диалоговом окне добавьте входной аргумент типа `string` по имени `MessageToShow` (присваивать стандартное значение этому новому аргументу не нужно). Удалите начальное сообщение из действия `WriteLine`, сбросив свойство `Text` действия `WriteLine` в окне `Properties` (Свойства) среды `Visual Studio`. Конечный результат показан на рис. 26.5.

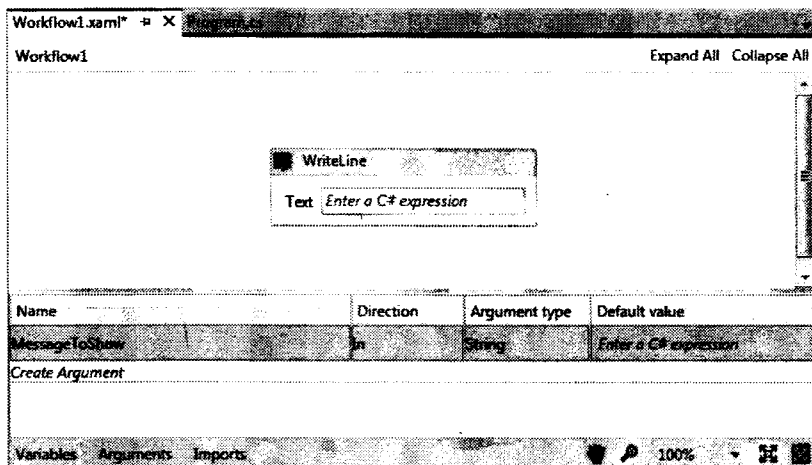


Рис. 26.5. Аргументы рабочего потока могут использоваться для приема аргументов от хоста

Теперь в свойстве Text действия WriteLine можно просто ввести MessageToShow в качестве вычисляемого выражения. Во время ввода вы заметите помощь со стороны средства IntelliSense (рис. 26.6).

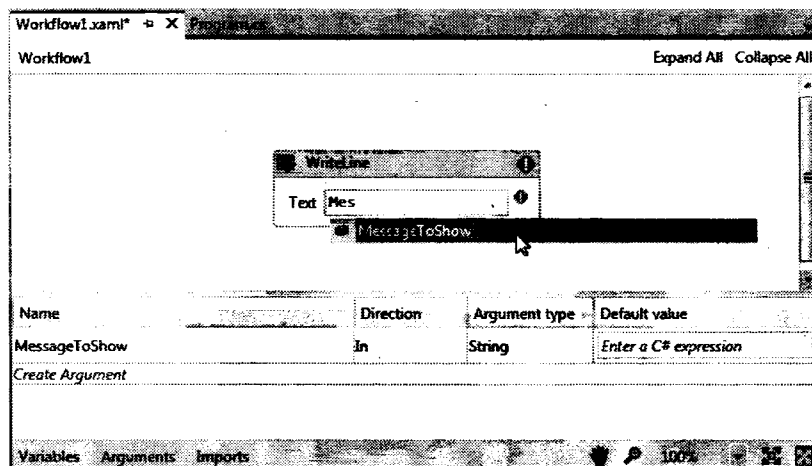


Рис. 26.6. Использование специального аргумента в качестве входных данных для действия

Теперь, когда имеется вся необходимая инфраструктура, в метод Main() класса Program понадобится внести показанные ниже изменения. Обратите внимание, что для объявления переменной Dictionary<> в файл Program.cs должно быть импортировано пространство имен System.Collections.Generic.

```
static void Main(string[] args)
{
    Console.WriteLine("***** Welcome to this amazing WF application *****");
    // Получить от пользователя данные для передачи рабочему потоку.
    Console.Write("Please enter the data to pass the workflow: ");
    string wfData = Console.ReadLine();
    // Упаковать данные в словарь.
    Dictionary<string, object> wfArgs = new Dictionary<string, object>();
    wfArgs.Add("MessageToShow", wfData);
    // Передать словарь рабочему потоку.
    Activity workflow1 = new Workflow1();
    WorkflowInvoker.Invoke(workflow1, wfArgs);
    Console.WriteLine("Thanks for playing");
}
```

Важно отметить, что строковые значения каждого члена переменной Dictionary<> должны быть именованы в соответствие с переменными-аргументами рабочего потока. Запуск модифицированной программы дает вывод, подобный следующему:

```
***** Welcome to this amazing WF application *****
Please enter the data to pass the workflow: Hello Mr. Workflow!
Hello Mr. Workflow!
Thanks for playing
Press any key to continue . . .
```

Помимо метода Invoke() другими действительно интересными членами WorkflowInvoker являются методы BeginInvoke() и EndInvoke(), которые позволяют запускать рабочий поток во вторичном потоке выполнения, используя шаблон асинхронного делегата .NET (см. главу 19). Если требуется большая степень контроля над тем, как исполняющая среда WF манипулирует рабочим потоком, можете вместо него применить класс WorkflowApplication.

Хостинг рабочего потока с использованием класса WorkflowApplication

Класс WorkflowApplication (как противоположность WorkflowInvoker) необходимо использовать, когда требуется сохранять или загружать длительно выполняющийся рабочий поток с применением служб постоянного хранения WF и получать уведомления о различных событиях, которые происходят на протяжении жизненного цикла экземпляра рабочего потока, работать с "закладками" WF и прочими расширенными средствами. В этом случае понадобится вызывать метод Run() класса WorkflowApplication.

При вызове метода Run() новый фоновый поток выполнения извлекается из пула потоков CLR. Таким образом, если не добавить дополнительную поддержку для обеспечения ожидания основным потоком завершения вторичного потока, то экземпляр рабочего потока может не получить шанс завершить свою работу.

Один из способов обеспечения времени ожидания, достаточного для завершения работы фонового потока, предусматривает использование объекта AutoResetEvent из пространства имен System.Threading. Ниже показаны изменения текущего примера, в котором теперь вместо WorkflowInvoker применяется WorkflowApplication:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Welcome to this amazing WF application *****");
    // Получить данные от пользователя для передачи рабочему потоку.
    Console.Write("Please enter the data to pass the workflow: ");
    string wfData = Console.ReadLine();

    // Упаковать данные в словарь.
    Dictionary<string, object> wfArgs = new Dictionary<string, object>();
    wfArgs.Add("MessageToShow", wfData);

    // Использовать для информирования первичного потока о необходимости ожидания.
    AutoResetEvent waitHandle = new AutoResetEvent(false);

    // Передать словарь рабочему потоку.
    WorkflowApplication app = new WorkflowApplication(new Workflow1(), wfArgs);

    // Связать событие с данным объектом app.
    // О факте завершения уведомить другой поток
    // и вывести на консоль соответствующее сообщение.
    app.Completed = (completedArgs) => {
        waitHandle.Set();
        Console.WriteLine("The workflow is done!");
    };
}
```

```
// Запустить рабочий поток.
app.Run();

// Подождать, пока не поступит уведомление о завершении рабочего потока.
waitHandle.WaitOne();

Console.WriteLine("Thanks for playing");
}
```

Вывод будет похожим на вывод предыдущей версии этого проекта:

```
***** Welcome to this amazing WF application *****
Please enter the data to pass the workflow: Hey again!
Hey again!
The workflow is done!
Thanks for playing
Press any key to continue . . .
```

Преимущество использования класса `WorkflowApplication` связано с возможностью вмешательства в события (как это было сделано непрямо с применением свойства `Completed`) и подключения более сложных служб (постоянного хранения, закладок и т.п.).

На заметку! В настоящем ознакомительном вступлении в WF детали этих служб времени выполнения не рассматриваются. Поведения времени выполнения и службы исполняющей среды Windows Workflow Foundation подробно описаны в документации .NET Framework 4.5 SDK.

Итоги по первому рабочему потоку

Несмотря на то что этот пример был тривиальным, вы научились решать несколько очень интересных (и полезных) задач. Было показано, как передавать объект `Dictionary`, содержащий пары “имя/значение”, которые затем попадают в виде идентично именованных аргументов в рабочий поток. Это действительно удобно, когда необходимо получить пользовательский ввод (такой как идентификационный номер клиента, номер карточки социального страхования и т.д.), который будет использоваться рабочим потоком для выполнения действий.

Вы также узнали, что рабочий поток .NET определяется в декларативной манере (по умолчанию) с применением основанной на XML грамматики под названием XAML. С помощью XAML можно указывать действия, которые содержит рабочий поток. Во время выполнения эти данные используются для создания корректной объектной модели в памяти. И последнее: были продемонстрированы два разных подхода к запуску рабочего потока с применением классов `WorkflowInvoker` и `WorkflowApplication`.

Исходный код. Проект `FirstWorkflowExampleApp` доступен в подкаталоге `Chapter 26`.

Знакомство с действиями рабочих потоков

Вспомните, что цель WF — позволить моделировать бизнес-процесс в декларативной манере, с последующим его выполнением исполняющей средой WF. В контексте WF бизнес-процесс состоит из любого количества действий. Выражаясь просто, действие WF — это атомарный “шаг” в общем процессе. При создании нового приложения рабочего потока вы обнаружите, что панель инструментов содержит значки для встроенных действий, сгруппированные по категориям.

Эти готовые действия используются для моделирования бизнес-процесса. Каждое действие в панели инструментов отображается на реальный класс внутри сборки `System.Activities.dll` (чаще всего содержащийся в пространстве имен `System.Activities.Statements`). Несколько таких готовых действий будут применяться на протяжении этой главы; ниже представлен краткий обзор многих стандартных действий. Как обычно, полная информация доступна в документации `.NET Framework 4.5 SDK`.

Действия потока управления

Первая категория действий в панели инструментов позволяет представлять задачи организации циклов и принятия решений в более крупном рабочем потоке. Их польза должна быть очевидной, учитывая, что похожие задачи часто приходится решать в коде `C#`. Действия потока управления перечислены в табл. 26.1; обратите внимание, что некоторые из них допускают параллельную обработку действий с использованием `Task Parallel Library` (см. главу 19).

Таблица 26.1. Действия потока управления в WF

Действие	Описание
<code>DoWhile</code>	Действие цикла, которое выполняет содержащиеся внутри действия как минимум один раз и повторяет это, пока логическое условие истинно
<code>ForEach<T></code>	Выполняет действие по одному разу для каждого значения, представленного в коллекции <code>ForEach<T>.Values</code>
<code>If</code>	Моделирует условие <code>If-Then-Else</code>
<code>Parallel</code>	Действие, которое выполняет все дочерние действия одновременно и асинхронно
<code>ParallelForEach<T></code>	Перечисляет элементы коллекции и выполняет каждый элемент коллекции параллельно
<code>Pick</code>	Обеспечивает моделирование потока управления на основе событий
<code>PickBranch</code>	Потенциальный путь выполнения внутри родительского действия <code>Pick</code>
<code>Sequence</code>	Выполняет набор дочерних действий последовательно
<code>Switch<T></code>	Выбирает одно из возможных действий для выполнения, в зависимости от значения заданного выражения типа, указанного в параметре типа объекта
<code>While</code>	Выполняет содержащийся элемент рабочего потока, пока условие истинно

Действия блок-схемы

Рассмотрим действия блок-схемы, которые довольно важны, учитывая, что действие `Flowchart` часто является первым элементом, который помещается на поверхность визуального конструктора рабочих потоков. Этот тип рабочего потока позволяет строить рабочий поток, используя известную модель блок-схемы, где выполнение рабочего потока основано на множестве ветвящихся путей, выбор каждого из которых зависит от истинности или ложности определенного внутреннего условия. В табл. 26.6 описаны члены этого набора действий.

Таблица 26.2. Действия блок-схемы в WF

Действие	Описание
Flowchart	Моделирует рабочие потоки, используя известную парадигму блок-схемы. Это — самое первое действие, которое помещается на поверхность визуального конструктора
FlowDecision	Действие, предоставляющее возможность моделировать условный узел с двумя возможными исходами
FlowSwitch<T>	Действие, позволяющее моделировать конструкцию переключения, с одним выражением и одним исходом для каждого соответствия

Действия обмена сообщениями

Рабочий поток может легко вызывать члены внешней веб-службы XML или службы WCF, а также получать уведомления от внешних служб, используя действия обмена сообщениями. Поскольку эти действия очень тесно связаны с разработкой WCF, они собраны в отдельную сборку .NET под названием `System.ServiceModel.Activities.dll`. Внутри этой библиотеки находится пространство имен `Activities`, в котором определены основные действия, перечисленные в табл. 26.3.

Таблица 26.3. Действия обмена сообщениями в WF

Действие	Описание
CorrelationScope	Используется для управления дочерними действиями обмена сообщениями
InitializeCorrelation	Инициализирует корреляцию без отправки или приема сообщения
Receive	Принимает сообщение от службы WCF
Send	Отправляет сообщение службе WCF
SendAndReceiveReply	Отправляет сообщение службе WCF и захватывает возвращенное значение
TransactedReceiveScope	Действие, которое позволяет направить транзакцию в рабочий поток или созданные диспетчером транзакции стороны сервера

Наиболее часто используемыми действиями обмена сообщениями являются `Send` и `Receive`, которые позволяют взаимодействовать с внешними веб-службами XML или службами WCF.

Действия конечного автомата

В версии .NET 4.5 в API-интерфейс WF был включен новый набор действий, который позволяет моделировать рабочие потоки, основанные на *конечных автоматах*. В сущности, конечные автоматы дают возможность определять рабочий поток, который в каждый момент времени может находиться в одном из нескольких заданных состояний, и допустимые переходы между этими состояниями.

Известным примером конечного автомата является торговый автомат по продаже газированной воды с сиропом. В любой момент времени “автомат” может находиться в одном состоянии, таком как (к примеру) “Ожидание ввода”, “Разлив газированной воды”, “Возврат платежа”, “Возврат сдачи”, “Отображение пустого выбора” и т.п. Между этими состояниями существует набор допустимых переходов.

Например, если автомат пребывает в состоянии “Отображение пустого выбора”, допустимые переходы могут включать “Возврат платежа” или “Разлив газированной воды” (при условии, что пользователь выбрал другой вариант). Для построения такого рабочего потока в .NET 4.5 предусмотрены действия `StateMachine`, `State` и `FinalState`.

Действия исполняющей среды и действия-примитивы

Следующие две категории действий в панели инструментов — `Runtime` (Исполняющая среда) и `Primitives` (Примитивы) — позволяют строить рабочий поток, который делает обращения к исполняющей среде `WF` (в случае `Persists` и `TerminateWorkflow`) и выполняет общие операции, такие как помещение текста в выходной поток или вызов метода на объекте .NET. Некоторые часто используемые действия этих двух категорий описаны в табл. 26.4.

Таблица 26.4. Действия исполняющей среды и действия-примитивы в WF

Действие	Описание
<code>Persist</code>	Заставляет рабочий поток сохранить свое состояние в базе данных, используя службу постоянства WF
<code>TerminateWorkflow</code>	Прерывает выполняющийся экземпляр рабочего потока, инициирует на хосте событие <code>WorkflowApplication.Completed</code> и выдает сообщение об ошибке. После такого прерывания рабочий поток не может быть возобновлен
<code>Assign</code>	Позволяет устанавливать свойства действия с использованием присваиваемых значений, определенных в визуальном конструкторе рабочего потока
<code>Delay</code>	Заставляет рабочий поток приостановиться на заданный период времени
<code>InvokeMethod</code>	Вызывает метод указанного объекта или типа
<code>WriteLine</code>	Записывает заданную строку в указанный экземпляр типа, производного от <code>TextWriter</code> . По умолчанию это будет стандартный выходной поток (т.е. консоль); однако можно сконфигурировать и другие потоки, такие как <code>FileStream</code>

Пожалуй, наиболее интересным и полезным действием в этом наборе является `InvokeMethod`, потому что оно позволяет вызывать методы классов .NET в декларативной манере. Действие `InvokeMethod` можно также сконфигурировать для сохранения любого возвращаемого значения, которое присылает вызванный метод. Действие `TerminateWorkflow` может пригодиться, когда нужно рассчитывать на точку невозврата. Если экземпляр рабочего потока добирается до этого действия, он инициирует событие `Completed`, которое может быть перехвачено на стороне хоста, как это делалось в первом примере.

Действия транзакций

При построении рабочего потока может требоваться обеспечение выполнения группы действий в атомарной манере — в том смысле, что они либо все должны выполняться успешно, либо все вместе быть отменены. Даже если соответствующие действия не работают непосредственно с реляционной базой данных, основные действия, представленные в табл. 26.5, позволяют добавлять транзакционный контекст в рабочий поток.

Таблица 26.5. Действия транзакций в WF

Действие	Описание
CancellationScope	Ассоциирует логику отмены внутри главного потока выполнения
CompensableActivity	Любое действие, поддерживающее компенсацию своих дочерних действий
TransactionScope	Действие, обозначающее границы транзакции

Действия над коллекциями и действия обработки ошибок

Последние две категории, которые осталось рассмотреть в этой вводной главе, позволяют декларативно манипулировать обобщенными коллекциями и реагировать на исключения времени выполнения. Действия коллекций незаменимы, когда требуется манипулировать объектами, представляющими бизнес-данные (такими как заказы на покупку, объекты медицинской информации или отслеживание заказов) на лету в XAML. Действия обработки ошибок, с другой стороны, позволяют реализовать логику try/catch/throw внутри рабочего потока. Этот последний набор действий WF описан в табл. 26.6.

Таблица 26.6. Действия над коллекциями и действия обработки ошибок в WF

Действие	Описание
AddToCollection<T>	Добавляет элемент к указанной коллекции
ClearCollection<T>	Очищает указанную коллекцию от всех элементов
ExistInCollection<T>	Определяет, представлен ли указанный элемент в данной коллекции
RemoveFromCollection<T>	Удаляет элемент из указанной коллекции
Rethrow	Иницирует ранее перехваченное исключение из действия Catch
Throw	Иницирует исключение
TryCatch	Содержит элементы рабочего потока, которые должны быть выполнены исполняющей средой рабочего потока внутри блока обработки исключений

Ознакомившись со многими из стандартных действий, можно приступить к построению ряда интересных рабочих потоков, которые их используют. По ходу дела вы узнаете о двух ключевых действиях, которые обычно функционируют в качестве корня рабочего потока — Flowchart и Sequence.

Построение рабочего потока в виде блок-схемы

В первом примере было показано, как перетащить простое действие WriteLine непосредственно на поверхность визуального конструктора рабочих потоков. Хотя и верно, что любое действие, находящееся в панели инструментов Visual Studio, может быть первым элементом, помещенным на поверхность визуального конструктора, только несколько из них способны содержать в себе дочерние поддействия (которые представляют коллекцию сгруппированных вместе связанных действий). При построении нового рабочего потока велика вероятность, что первым элементом, который будет помещен на поверхность визуального конструктора, будет действие Flowchart или Sequence.

Оба эти встроенных действия обладают способностью содержать любое количество внутренних дочерних действий (включая дополнительные действия Flowchart или Sequence), представляющих сущность бизнес-процесса. Для начала создайте новое консольное приложение рабочего потока по имени EnumerateMachineDataWF. Затем переименуйте начальный файл *.xaml в MachineInfoWF.xaml.

Из раздела Flowchart (Блок-схема) панели инструментов перетащите на поверхность визуального конструктора действие Flowchart. В окне Properties (Свойства) измените значение свойства DisplayName на что-то более осмысленное, такое как Show Machine Data Flowchart (свойство DisplayName определяет то, как элемент называется в визуальном конструкторе). Теперь поверхность визуального конструктора рабочего потока должна выглядеть примерно так, как показано на рис. 26.7.

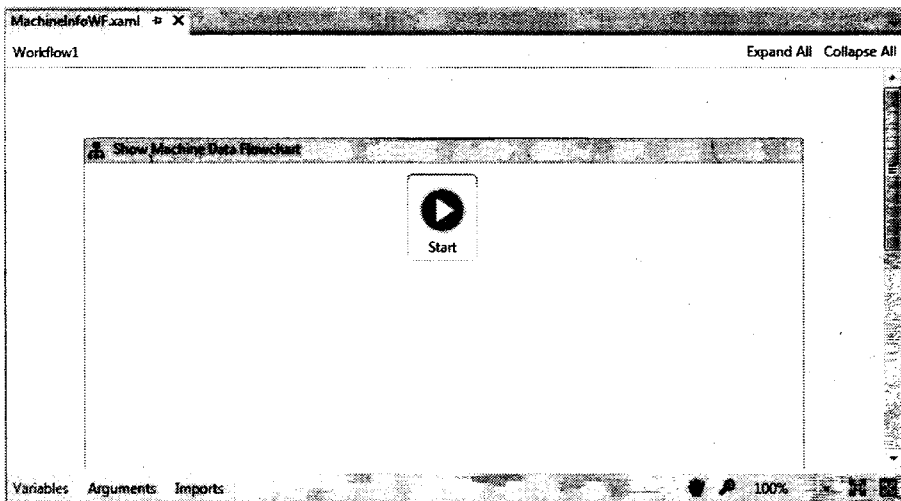


Рис. 26.7. Начальное действие Flowchart

Скоба захвата, которая появляется в нижнем правом углу действия Flowchart (при наведении курсора мыши), позволяет увеличивать и уменьшать размер пространства блок-схемы. Размер понадобится увеличить, чтобы добавить внутри новые действия.

Подключение действий к блок-схеме

Большой значок Start (Пуск) представляет точку входа в действие Flowchart, которое в этом примере является первым действием всего рабочего потока и будет инициировано при запуске рабочего потока с помощью классов WorkflowInvoker или WorkflowApplication. Этот значок может располагаться в любом месте визуального конструктора, но рекомендуется его переместить в левый верхний угол, чтобы освободить больше места.

Цель состоит в том, чтобы собрать блок-схему, соединяя любое количество дополнительных действий вместе, обычно с использованием в процессе действия FlowDecision. Для начала перетащите действие WriteLine на поверхность визуального конструктора, изменив значение его свойства DisplayName на Greet User. Наведя курсор мыши на значок Start, вы увидите петельки для стыковки с каждой стороны. Щелкните на ближайшей к действию WriteLine петельке для стыковки и перетащите ее к петельке для стыковки действия WriteLine. Между этими двумя элементами появится соединение, которое означает, что первым выполняемым действием рабочего потока будет Greet User.

Теперь, подобно первому примеру этой главы, добавьте аргумент рабочего потока (через кнопку Arguments (Аргументы)) по имени `UserName` типа `string` без какого-либо стандартного значения. Чуть позже оно будет передано динамически через специальный объект `Dictionary<>`. Наконец, установите в качестве значения свойства `Text` действия `WriteLine` следующий оператор кода:

```
"Hello" + UserName
```

Добавьте на поверхность визуального конструктора второе действие `WriteLine` и соедините его с предыдущим. На этот раз для свойства `Text` определите жестко закодированное строковое значение `"Do you want me to list all machine drives?"` и измените значение свойства `DisplayName` на `Ask User`. На рис. 26.8 показано соединение между текущими действиями рабочего потока.

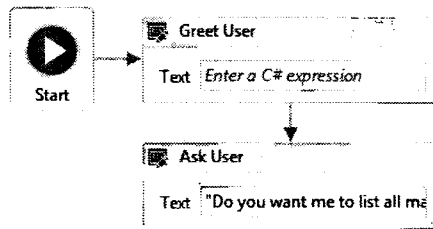


Рис. 26.8. Рабочие потоки в виде блок-схемы соединяют действия вместе

Работа с действием `InvokeMethod`

Поскольку большая часть рабочего потока определяется в декларативной манере с помощью XAML, наверняка будет часто использоваться действие `InvokeMethod`, которое позволяет вызывать методы реальных объектов в различных точках рабочего потока. Перетащите один из таких элементов на поверхность визуального конструктора, измените значение свойства `DisplayName` на `Get Y or N`, и установите соединение между ним и действием `Ask User` типа `WriteLine`.

Первым свойством для конфигурирования действия `InvokeMethod` является `TargetType`. Оно представляет имя класса, определяющего статический член, который требуется вызвать. В раскрывающемся списке свойства `TargetType` действия `InvokeMethod` выберите пункт `Browse for Types...` (Обзор типов...), как показано на рис. 26.9.

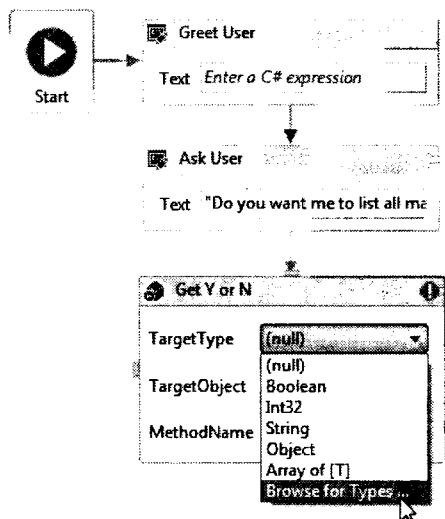


Рис. 26.9. Указание целевого типа для `InvokeMethod`

В открывшемся диалоговом окне выберите класс `System.Console` из сборки `microsoft.dll` (если ввести имя типа в поле `Type Name` (Имя типа), он будет автоматически найден). После нахождения класса `System.Console` щелкните на кнопке `OK`.

В визуальном конструкторе введите `ReadLine` в качестве значения свойства `MethodName` действия `InvokeMethod`. Это сконфигурирует действие `InvokeMethod` на вызов метода `Console.ReadLine()` по достижении данного шага рабочего потока.

Как известно, `Console.ReadLine()` возвращает строковое значение, которое содержит символы, введенные с клавиатуры до нажатия клавиши `<Enter>`; однако нужен какой-то способ получить его. Этим сейчас и займемся.

Определение переменных уровня рабочего потока

Определение переменной рабочего потока в XAML в основном идентично определению аргумента, в том плане, что это можно делать непосредственно в визуальном конструкторе (на этот раз через кнопку `Variables` (Переменные)). Отличие в том, что аргументы применяются для захвата данных, переданных хостом, в то время как переменные — это просто элементы данных в рабочем потоке, которые будут использованы для влияния на его поведение времени выполнения.

Щелкнув на кнопке `Variables` визуального конструктора, добавьте новую переменную типа `string` по имени `YesOrNo`. Обратите внимание, что при наличии в рабочем потоке нескольких родительских контейнеров (например, `Flowchart`, содержащего внутри `Sequence`) можно выбрать область видимости переменной. В рассматриваемом примере доступен единственный выбор — корневое действие `Flowchart` (рис. 26.10).

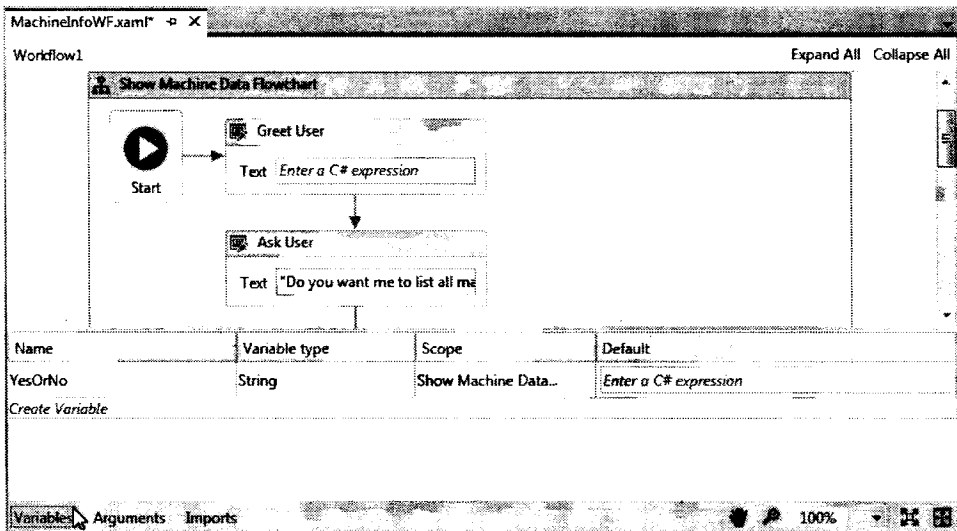


Рис. 26.10. Определение переменной рабочего потока

Затем выберите в визуальном конструкторе рабочих потоков действие `InvokeMethod` и в окне `Property` (Свойства) укажите для свойства `Result` новую переменную (рис. 26.11).

Теперь, получив фрагмент данных от вызова внешнего метода, его можно применять для принятия решений во время выполнения внутри блок-схемы, используя действие `FlowDecision`.

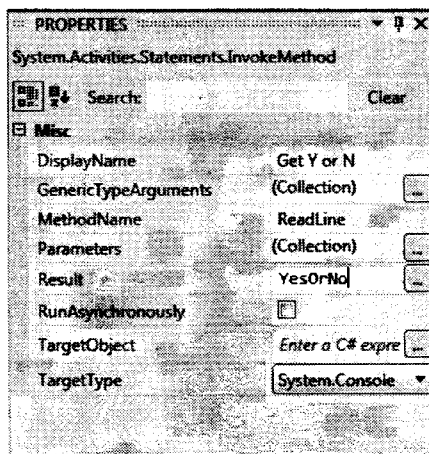


Рис. 26.11. Полностью сконфигурированное действие InvokeMethod

Работа с действием FlowDecision

Действие FlowDecision служит для выбора одного из двух возможных действий, в зависимости от истинности или ложности булевской переменной либо оператора, возвращающего булевское значение. Перетащите одно из этих действий на поверхность визуального конструктора и соедините с действием InvokeMethod (рис. 26.12).

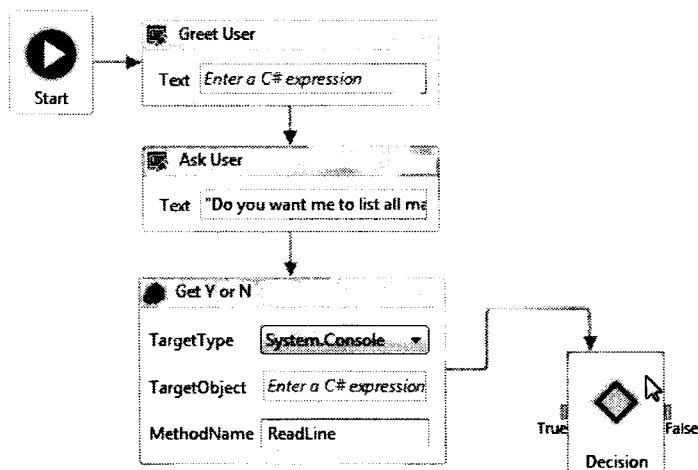


Рис. 26.12. Действие FlowDecision позволяет организовать ветвление по двум направлениям

На заметку! Если необходимо оформить условие множественного ветвления внутри блок-схемы, используйте действие FlowSwitch<T>. Оно позволяет определить любое количество путей, из которых выбирается один в зависимости от значения определенной переменной рабочего потока.

Установите для свойства Condition (в окне Properties) действия FlowDecision следующий оператор кода, который можно ввести непосредственно в редакторе (здесь производится проверка значения переменной YesOrNo, представленное в верхнем регистре, на предмет равенства "Y"):

```
YesOrNo.ToUpper() == "Y"
```


Работа с действием `TerminateWorkflow`

Теперь необходимо построить действия, которые происходят с обеих сторон от действия `FlowDecision`. На стороне `False` подключитесь к действию `WriteLine`, выводящему жестко закодированное сообщение, за которым следует действие `TerminateWorkflow` (рис. 26.13).

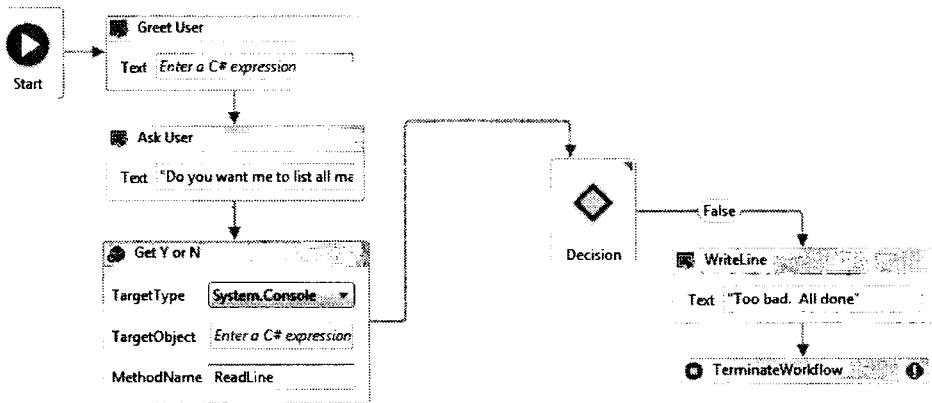


Рис. 26.13. Ветвь `False`

Строго говоря, использовать действие `TerminateWorkflow` не обязательно, поскольку рабочий поток и так завершается по завершении ветви `False`. Однако с помощью данного действия можно сгенерировать исключение для хоста рабочего потока, информируя его о причине останова. Это исключение может быть сконфигурировано в окне `Properties`.

Выберите действие `TerminateWorkflow` в визуальном конструкторе и в окне `Properties` щелкните на кнопке с многоточием для свойства `Exception`. Откроется редактор, который позволит определить исключение, как это бы делалось в коде (рис. 26.14).

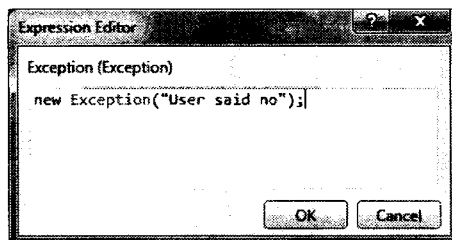


Рис. 26.14. Конфигурирование исключения для генерации по достижении действия `TerminateWorkflow`

Завершите конфигурирование этого действия, установив для свойства `Reason` значение `"YesOrNo was false"`.

Построение условия `True`

Чтобы начать построение условия `True` для действия `FlowDecision`, подключите действие `WriteLine`, которое просто отображает жестко закодированную строку, сообщающую о согласии пользователя продолжить работу. Соедините его с новым действием `InvokeMethod`, которое вызовет метод `GetLogicalDrives()` класса `System.Environment`. Для этого установите свойство `TargetType` в `System.Environment`, а свойство `MethodName` — в `GetLogicalDrives` (рис. 26.15).

Добавьте новую переменную уровня рабочего потока (щелкнув на кнопке **Variables** в визуальном конструкторе рабочих потоков) по имени `DriveNames` типа `string[]`. Для указания массива строк выберите пункт **Array of [T]** в раскрывающемся списке **Variable Type** (Тип переменной) и укажите `String` в открывшемся диалоговом окне. Наконец, установите свойство **Result** этого нового действия `InvokeMethod` в переменную `DriveNames`, выбрав действие `InvokeMethod` на поверхности визуального конструктора и открыв окно **Properties**.

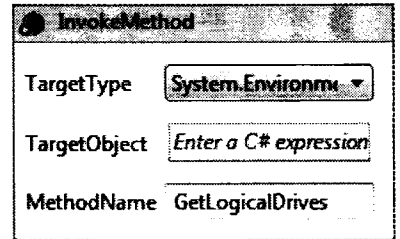


Рис. 26.15. Сконфигурированное действие `InvokeMethod`

Работа с действием `ForEach<T>`

Следующая часть ветви `True` будет выводить имена всех дисковых устройств в окне консоли, что подразумевает необходимость в циклическом проходе по данным, представленным в переменной `DriveNames`, которая была сконфигурирована как массив объектов `string`. Действие `ForEach<T>` — это WF-эквивалент ключевого слова `foreach` языка C#, и настраивается это действие похожим образом (во всяком случае, концептуально).

Перетащите действие `ForEach<T>` на поверхность визуального конструктора и соедините с предыдущим действием `InvokeMethod`. Чуть позже вы сконфигурируете действие `ForEach<T>`, а пока, чтобы завершить ветвь `True`, поместите еще одно действие `WriteLine` на поверхность визуального конструктора. На рис. 26.16 показано, как теперь должен выглядеть рабочий поток.

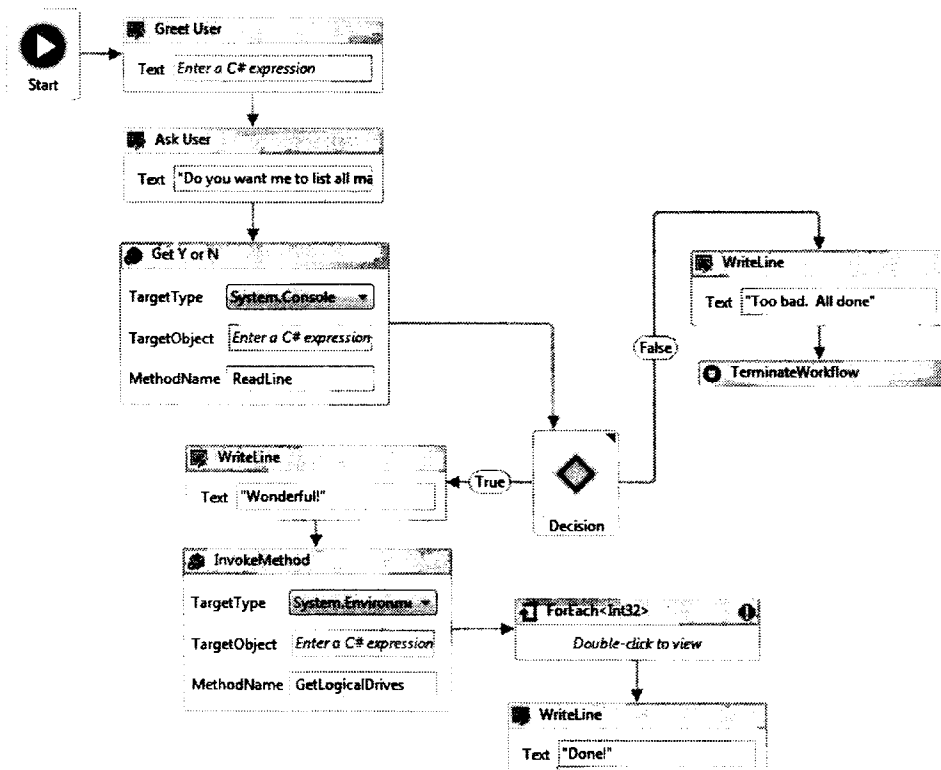


Рис. 26.16. Завершенный рабочий поток высшего уровня

Чтобы избавиться от присутствующей в визуальном конструкторе ошибки, понадобится завершить конфигурирование действия `ForEach<T>`. Сначала в окне `Properties` укажите аргумент типа для обобщения, которым в данном примере будет `String`. Свойство `Values` представляет собой место, откуда берутся данные, и в рассматриваемом случае это переменная `DriveNames` (рис. 26.17).

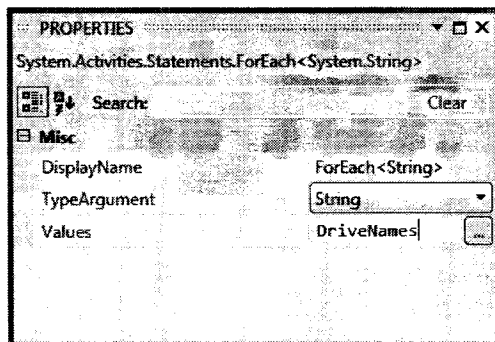


Рис. 26.17. Установка типа для перечисления `ForEach`

Это конкретное действие нуждается в дополнительном редактировании, для чего нужно дважды щелкнуть в визуальном конструкторе, чтобы открыть визуальный мини-конструктор, предназначенный только для данного действия. Не все действия WF поддерживают двойной щелчок с открытием визуального мини-конструктора, но это несложно определить по самому действию (оно сообщает “Double-click to view” (“Дважды щелкните для просмотра”)). Выполните двойной щелчок на действии `ForEach<T>` и добавьте одиночное свойство `WriteLine`, которое выведет все значения `string` в возвращаемом значении `DriveNames` (рис. 26.18).

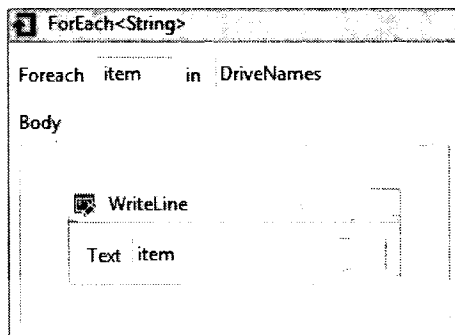


Рис. 26.18. Завершающий шаг конфигурирования действия `ForEach<String>`

На заметку! В визуальном мини-конструкторе `ForEach<T>` можно добавлять любое необходимое количество действий. Все они будут выполняться на каждой итерации цикла.

Завершив конфигурирование “поддействий” `ForEach<T>`, можно воспользоваться ссылками в верхнем левом углу визуального конструктора, чтобы вернуться на высший уровень рабочего потока (при работе с наборами действий вы довольно часто будете использовать эти навигационные цепочки; на рис. 26.19 они отмечены курсором мыши).

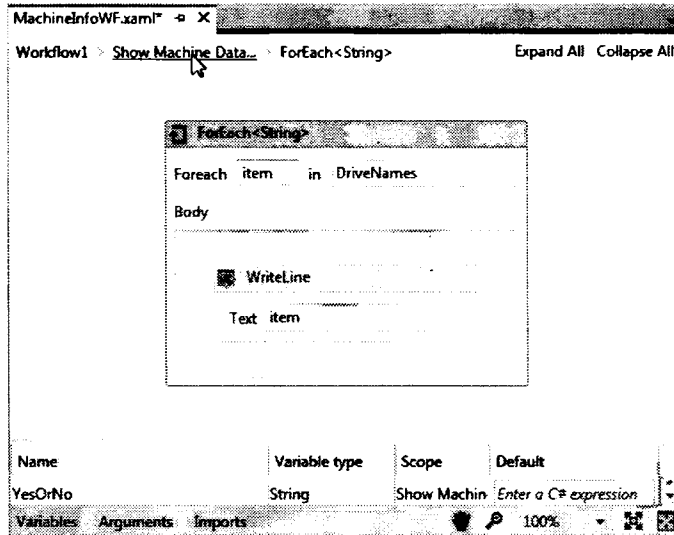


Рис. 26.19. Навигационные цепочки визуального конструктора рабочих потоков позволяют возвратиться к действию высшего уровня

Завершение приложения

Пример почти готов. Все, что осталось сделать — это модифицировать метод `Main()` класса `Program`, добавив перехват исключения, которое будет инициировано, если пользователь ответит “N” и тем самым вызовет генерацию `Exception`. Измените код, как показано ниже (не забыв импортировать в файл кода пространство имен `System.Collections.Generic`):

```
static void Main(string[] args)
{
    try
    {
        Dictionary<string, object> wfArgs = new Dictionary<string, object>();
        wfArgs.Add("UserName", "Mel");
        Activity workflow1 = new Workflow1();
        WorkflowInvoker.Invoke(workflow1, wfArgs);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
        Console.WriteLine(ex.Data[«Reason»]);
    }
}
```

Обратите внимание, что причина ("Reason") исключения может быть получена из свойства `Data` класса `System.Exception`. Если запустить программу и ввести “Y” в ответ на вопрос о перечислении дисков, то появится следующий вывод:

```
Hello Andrew
Do you want me to list all machine drives?
Y
Wonderful!
C:\
D:\
E:\
F:\
G:\
H:\
I:\
Thanks for using this workflow
```

Однако если ввести “N” (или любое другое значение, отличное от “Y” или “y”), вывод будет таким:

```
Hello Andrew
Do you want me to list all machine drives?
n
Too bad. All done
YesOrNo was false
```

Промежуточные итоги

У новичков в работе со средой рабочих потоков может возникнуть вопрос: в чем состоит преимущество кодирования столь простого процесса с использованием WF XAML по сравнению с чистым кодом C#? В конце концов, можно было бы вообще обойтись без Windows Workflow Foundation и просто написать следующий класс C#:

```
class Program
{
    static void Main(string[] args)
    {
        try
        {
            ExecuteBusinessProcess();
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
            Console.WriteLine(ex.Data["Reason"]);
        }
    }
    private static void ExecuteBusinessProcess()
    {
        string UserName = "Andrew";
        Console.WriteLine("Hello {0}", UserName);
        Console.WriteLine("Do you want me to list all machine drives?");
        string YesOrNo = Console.ReadLine();
        if (YesOrNo.ToUpper() == "Y")
        {
            Console.WriteLine("Wonderful!");
            string[] DriveNames = Environment.GetLogicalDrives();
            foreach (string item in DriveNames)
            {
                Console.WriteLine(item);
            }
            Console.WriteLine("Thanks for using this workflow");
        }
        else
        {
            Console.WriteLine("K, Bye...");
            Exception ex = new Exception("User Said No!");
            ex.Data["Reason"] = "YesOrNo was false";
        }
    }
}
```

Вывод этой программы был бы совершенно идентичным предыдущему рабочему потоку на базе XAML. Так зачем вообще связываться со всеми этими действиями? Прежде всего, учтите, что не всем комфортно читать код C#. Если придется объяснять этот бизнес-процесс, скажем, продавцам и нетехническим менеджерам, что вы выберете —

код C# или блок-схему? Ответ вполне предсказуем. Важнее всего то, что API-интерфейс WF имеет множество дополнительных служб времени выполнения, включая сохранение длительно выполняющихся рабочих потоков в базе данных, автоматическое отслеживание событий рабочего потока и т.п. Если подумать об объеме работы, которую пришлось бы сделать, чтобы воспроизвести всю эту функциональность в новом проекте, польза от WF станет еще более очевидной.

С учетом сказанного, API-интерфейс WF — не обязательно правильный выбор для всех программ .NET. Тем не менее, для наиболее традиционных бизнес-приложений возможности определения, размещения, выполнения и мониторинга рабочих потоков действительно очень полезны. Как с любой новой технологией, понадобится определить, что будет полезно для текущего проекта. Давайте рассмотрим другой пример работы с API-интерфейсом WF, на этот раз упаковав рабочий поток в отдельную сборку *.dll.

Исходный код. Проект EnumerateMachineDataWF доступен в подкаталоге Chapter 26.

Построение последовательного рабочего потока (в выделенной библиотеке)

Хотя создание консольного приложения рабочего потока — замечательный способ поэкспериментировать с API-интерфейсом WF, готовый для реальной производственной эксплуатации рабочий поток должен быть упакован в специальную сборку .NET *.dll. После этого рабочие потоки можно многократно использовать на двоичном уровне в различных проектах.

Хотя можно было бы начать с использования проекта библиотеки классов C# в качестве стартовой точки, более простой способ построения библиотеки рабочего потока предусматривает выбор шаблона проекта Activity Library (Библиотека действий), который находится в узле Workflow (Рабочий поток) диалогового окна New Project. Преимущество этого типа проекта заключается в том, что он автоматически устанавливает все необходимые ссылки на сборки WF и предоставляет файл *.xaml для создания начального рабочего потока.

Создаваемый рабочий поток будет моделировать процесс запроса к базе данных AutoLot с целью проверки, имеется ли в таблице Inventory указанный автомобиль соответствующего цвета и изготовителя. Если запрошенный автомобиль есть на складе, для хоста будет сформирован ответ в виде выходного параметра. Если затребованного элемента на складе нет, будет сгенерировано сообщение руководителю отдела продаж о том, чтобы он нашел автомобиль нужного цвета.

Определение начального проекта

Создайте новый проект Activity Library по имени CheckInventoryWorkflowLib (рис. 26.20). После создания проекта переименуйте начальный файл Activity1.xaml в CheckInventory.xaml. К сожалению, когда производится переименование XAML-файла рабочего потока, лежащий в основе класс не переименовывается, как можно было ожидать. Чтобы устранить эту проблему, щелкните правой кнопкой мыши на файле CheckInventory.xaml в Solution Explorer и выберите в контекстном меню представление кода. Модифицируйте открывающий корневой элемент <Activity> для отражения корректного имени, как показано ниже:

```
<Activity mc:Ignorable="sap sap2010 sads"
  x:Class="CheckInventoryWorkflowLib.CheckInventory"
  ...
>
```

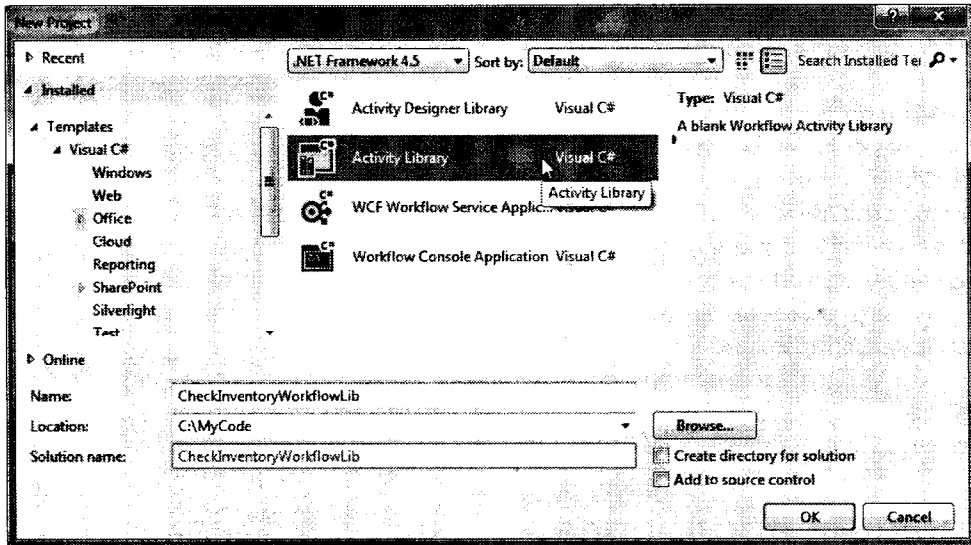


Рис. 26.20. Построение библиотеки действий

Этот рабочий поток в качестве первичного действия будет использовать Sequence, а не Flowchart. Перетащите новое действие Sequence на поверхность визуального конструктора (оно находится в области Control Flow (Поток управления) панели инструментов) и измените значение свойства DisplayName на Look Up Product. Поверхность визуального конструктора к этому моменту выглядит так, как показано на рис. 26.21.

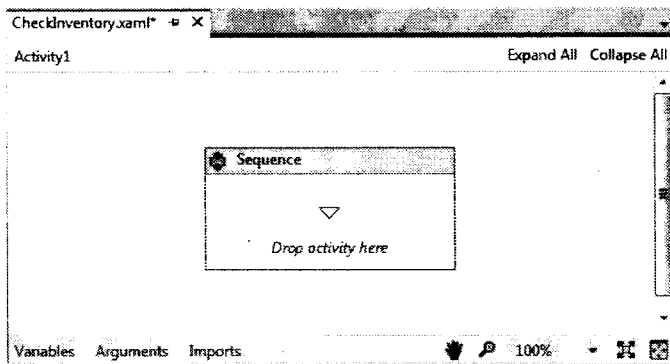


Рис. 26.21. Действие Sequence верхнего уровня

Как следует из названия, действие Sequence позволяет легко создавать последовательные задачи, которые выполняются друг за другом. Тем не менее, это не обязательно означает, что дочерние действия должны следовать строго линейному порядку. Последовательность может содержать блок-схемы, другие последовательности, параллельную обработку данных, ветвление If/Else и все, что имеет смысл для проектируемого бизнес-процесса.

Импорт сборок и пространств имен

Поскольку рабочий поток будет взаимодействовать с базой данных AutoLot, следующий шаг состоит в добавлении ссылки на сборку AutoLot.dll с помощью диалогового окна Add Reference (Добавить ссылку) в Visual Studio. В этом примере используется автономный уровень, поэтому рекомендуется взять финальную версию этой сборки, созданную в главе 22 (AutoLotDAL (Version 3)).

В этом рабочем потоке для опроса возвращаемого объекта `DataTable` с целью выяснения наличия требуемого товара на складе используется API-интерфейс `Linq to DataSet`. Следовательно, понадобится дополнительно установить ссылку на `System.Data.DataSetExtensions.dll`, т.к. это не делается автоматически для новых проектов `Activity Library`.

После добавления ссылок на эти сборки щелкните на кнопке `Imports` (Импорты) в нижней части визуального конструктора рабочих потоков. Вверху редактора `Imports` (Импорты) имеется текстовое поле, где можно вводить названия пространств имен `.NET`, которые необходимо применять в области видимости рабочего потока (считайте эту область декларативной версией ключевого слова `using` языка `C#`).

Для добавления пространств имен из любой ссылаемой сборки необходимо ввести их в текстовом поле, расположенном в верхней части редактора `Imports`. Импортируйте с его помощью пространство имен `AutoLotDisconnectedLayer`. После этого можно будет ссылаться на содержащиеся внутри него типы без необходимости использования полностью заданных имен. На рис. 26.22 показано содержимое области `Imports` после завершения импорта.

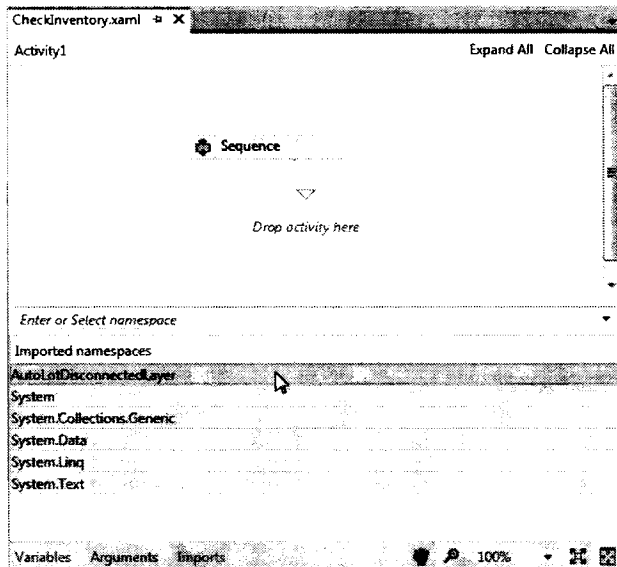


Рис. 26.22. Область `Imports` позволяет включать пространства имен `.NET` в рабочий поток

Определение аргументов рабочего потока

Далее понадобится определить два новых входных аргумента уровня рабочего потока с именами `RequestedMake` и `RequestedColor`, которые оба будут относиться к типу `String`. Подобно предыдущим примерам, хост рабочего потока будет создавать объект `Dictionary`, который содержит данные, отображаемые на эти аргументы, так что нет необходимости присваивать этим элементам стандартные значения в редакторе `Arguments` (Аргументы). Как и можно было предположить, рабочий поток будет применять эти входные значения для выполнения запроса к базе данных.

Тот же самый редактор `Arguments` можно использовать и для определения выходного аргумента по имени `FormattedResponse` типа `String`. Когда необходимо вернуть данные из рабочего потока обратно хосту, разрешается создавать любое количество выходных аргументов, которые могут быть перечислены хостом по завершении рабочего потока. На рис. 26.23 показано текущее состояние визуального конструктора рабочих потоков.

Name	Direction	Argument type	Default value
RequestedMake	In	String	Enter a C# expression
RequestedColor	In	String	Enter a C# expression
FormattedResponse	Out	String	Default value not supported

Create Argument
Variables Arguments Imports

Рис. 26.23. Входные и выходные аргументы

Определение переменных рабочего потока

Теперь в рабочем потоке необходимо объявить переменную-член, которая соответствует классу InventoryDALDisLayer из AutoLotDAL.dll. Вспомните из главы 22, что этот класс позволяет получать данные из базы Inventory, возвращенные в виде объекта DataTable. Выберите созданное ранее действие Sequence в визуальном конструкторе и, щелкнув на кнопке Variables, создайте переменную по имени AutoLotInventory. В раскрывающемся списке Variable Type (Тип переменной) выберите пункт Browse for Types... (Обзор типов...) и введите тип InventoryDALDisLayer (рис. 26.24).

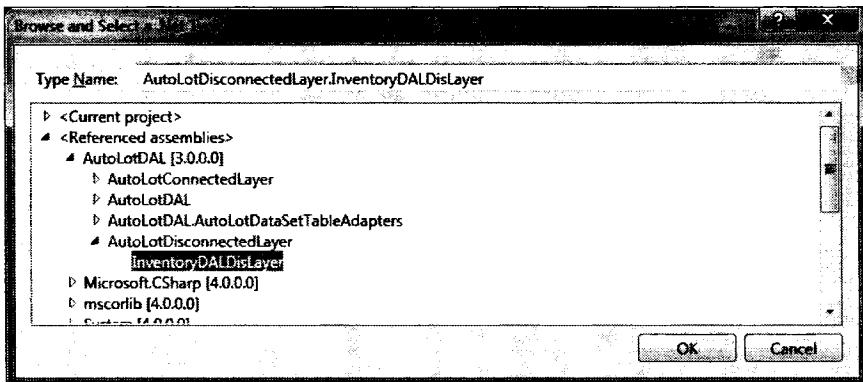


Рис. 26.24. Переменные рабочего потока предоставляют способ их декларативного определения внутри области видимости

Удостоверившись, что новая переменная выбрана, щелкните в окне Properties на кнопке с многоточием для свойства Default. Откроется окно редактора кода, размеры которого можно изменять произвольным образом (что удобно при вводе объемного кода). Этот редактор значительно упрощает ввод сложного кода для присваивания переменной. Введите следующий код (в одной строке), который создаст новую переменную InventoryDALDisLayer:

```
new InventoryDALDisLayer(@"Data Source=(local)\SQLEXPRESS;  
Initial Catalog=AutoLot;Integrated Security=True")
```

Объявите вторую переменную рабочего потока типа System.Data.DataTable по имени Inventory, опять используя пункт меню Browse for Types... (установите ее стандартное значение в null, как показано на рис. 26.25).

Позднее переменной Inventory будет присвоен результат вызова метода GetAllInventory() на переменной InventoryDALDisLayer.

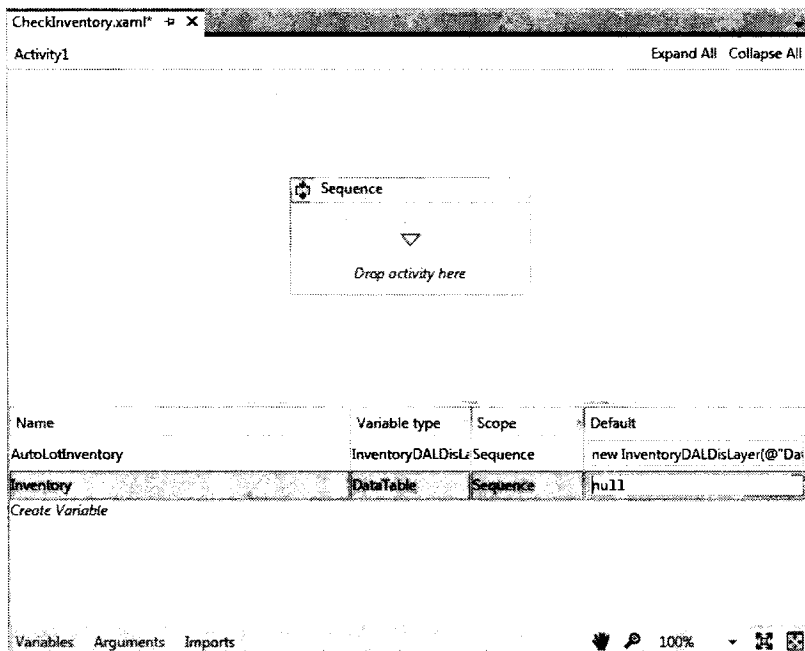


Рис. 26.25. Объявление переменной DataTable в рабочем потоке

Работа с действием Assign

Действие Assign позволяет устанавливать значение переменной, которое может быть результатом выполнения любых допустимых операторов кода. Перетащите действие Assign (находящееся в области Primitives (Примитивы) панели инструментов) на действие Sequence (рис. 26.26).

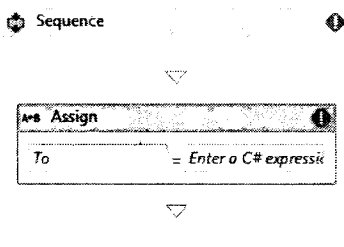


Рис. 26.26. Действие Assign

В поле редактирования слева укажите переменную Inventory. В поле редактирования справа введите следующий код:

```
AutoLotInventory.GetAllInventory()
```

После выполнения действия Assign в рабочем потоке будет получен объект DataTable, содержащий все записи из таблицы Inventory. Однако необходимо проверить наличие корректного товара на складе, используя значения аргументов RequestedMake и RequestedColor, присланных хостом. Для определения существования товара применяется API-интерфейс LINQ to DataSet и действие If рабочего потока.

Работа с действиями If и Switch

Перетащите действие If на узел Sequence непосредственно под действием Assign (рис. 26.27).

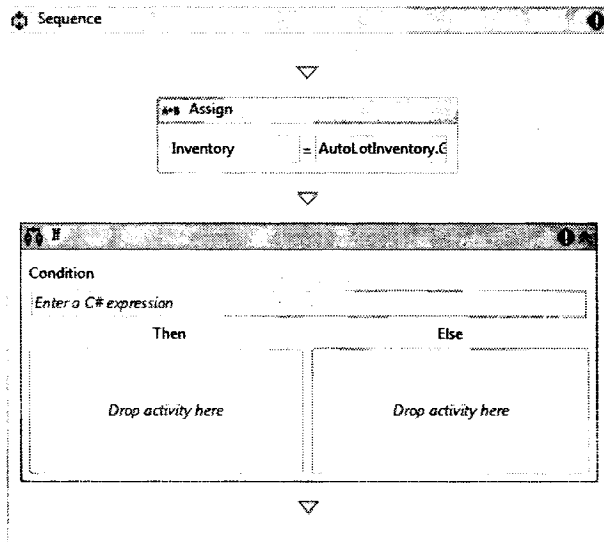


Рис. 26.27. Действие If

Поскольку это действие If позволяет принимать решения во время выполнения, прежде всего оно должно быть сконфигурировано для проверки булевского выражения. В редакторе Condition (Условие) введите следующий запрос LINQ to DataSet:

```
(from car in Inventory.AsEnumerable()
 where (string)car["Color"] == RequestedColor &&
 (string)car["Make"] == RequestedMake select car).Any()
```

Этот запрос LINQ использует аргументы RequestedColor и RequestedMake, предоставляемые хостом рабочего потока, для извлечения из DataTable всех записей об автомобилях нужного изготовителя и цвета. Вызов расширяющего метода Any() возвратит значение true или false в зависимости от того, содержит ли результат запроса какие-то данные.

Следующая задача заключается в конфигурировании набора действий, которые будут выполнены, когда указанное условие истинно или ложно. Вспомните, что конечной целью является отправка пользователю сформатированного сообщения, если запрошенный им автомобиль действительно имеется на складе. Тем не менее, чтобы усложнить задачу, мы будем возвращать уникальное сообщение, зависящее от того, автомобиль какого производителя был запрошен (BMW, Yugo или что-то еще).

Перетащите действие Switch<T> (находящееся в области Control Flow панели инструментов) на область Then действия If. Откроется диалоговое окно с запросом параметра обобщенного типа; укажите String. После этого воспользуйтесь визуальным конструктором рабочих потоков для установки поля Expression действия Switch в RequestedMake (рис. 26.28).

Вы увидите, что опция Default (Стандартная) действия Switch уже на месте, но ее нужно развернуть, чтобы добавить поддействия (щелкнув на ссылке Add an activity (Добавить действие)). Добавьте одиночное действие Assign.

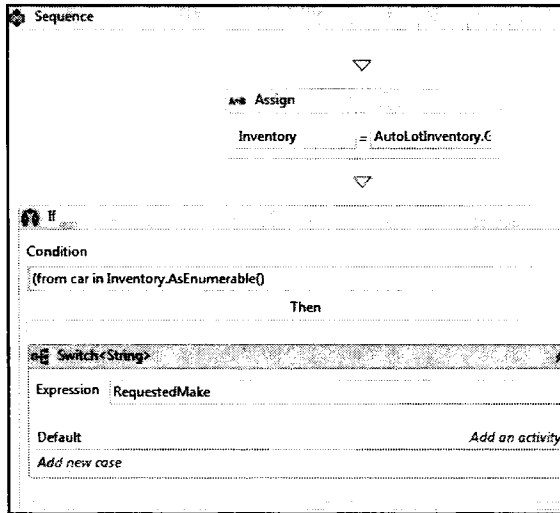


Рис. 26.28. Действие Switch

После добавления действие Assign в область редактирования Default присвойте аргументу FormattedResponse следующий оператор кода:

```
String.Format("Yes, we have a {0} {1} you can purchase",
    RequestedColor, RequestedMake)
```

К этому моменту редактор Switch будет выглядеть примерно так, как показано на рис. 26.29.

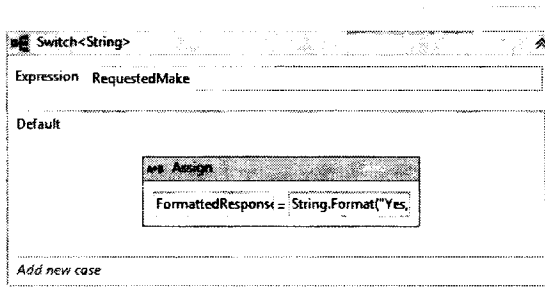


Рис. 26.29. Определение стандартной задачи для действия Switch

Теперь щелкните на ссылке Add New Case (Добавить новый вариант) и введите BMW (без двойных кавычек) для первой ветви Case, после чего еще раз щелкните на указанной ссылке для ввода финальной ветви со значением Yugo (снова без двойных кавычек). В каждую из этих областей Case перетащите действие Assign и в обоих случаях присвойте значение переменной FormattedResponse. В случае BMW укажите такое значение:

```
String.Format("Yes sir! We can send you {0} {1} as soon as {2}!",
    RequestedColor, RequestedMake, DateTime.Now)
```

В случае Yugo используйте следующее выражение:

```
String.Format("Please, we will pay you to get this {0} off our lot!",
    RequestedMake)
```

После этого действие Switch будет выглядеть, как показано на рис. 26.30.

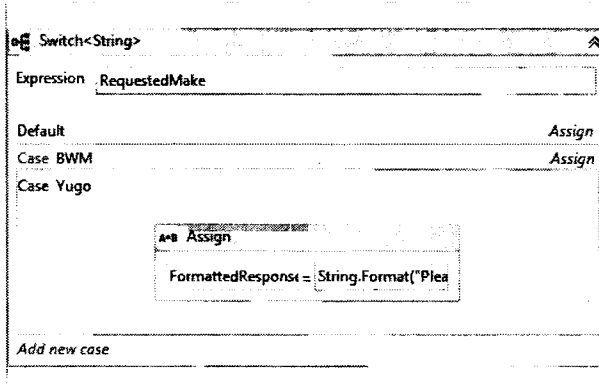


Рис. 26.30. Окончательный вид действия Switch

Построение специального действия кода

Впечатляющим средством визуального конструктора рабочего потока является его возможность встраивания сложных операторов кода (и запросов LINQ) в файл XAML, поскольку наверняка будет возникать необходимость написания кода в выделенном классе. С помощью API-интерфейса WF сделать это можно несколькими способами, самый простой из которых предусматривает создание класса, расширяющего `CodeActivity`, или же, если действие должно возвращать значение, то расширяющего `CodeActivity<T>` (где `T` — тип возвращаемого значения).

Рассмотрим пример создания специального действия, которое будет выводить данные в текстовый файл, информируя персонал отдела продаж о поступлении запроса на автомобиль, которого нет на складе. Выберите пункт меню `Project⇒Add New Item` (Проект⇒Добавить новый элемент), укажите в качестве шаблона `Code Activity` (Действие кода) и назначьте ему имя `CreateSalesMemoActivity.cs` (рис. 26.31).

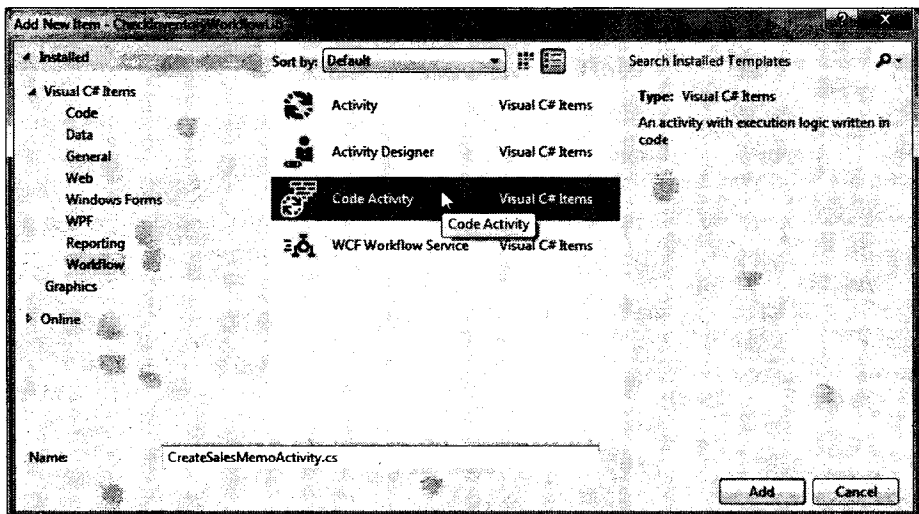


Рис. 26.31. Добавление нового действия кода

Если специальное действие требует какого-то ввода для последующей обработки, он может быть представлен свойством, инкапсулирующим объект `InArgument<T>`. Тип класса `InArgument<T>` — это специфическая сущность API-интерфейса WF, которая обеспечивает возможность передачи данных, предоставленных рабочим потоком, самому специальному классу действия. Создаваемому действию понадобится два таких свойства, которые будут представлять производителя и цвет искомого товара на складе.

Кроме того, специальное действие кода должно переопределить виртуальный метод `Execute()`, который будет вызван исполняющей средой WF, когда до него дойдет очередь выполнения рабочего потока. Обычно в этом методе будут использоваться свойства `InArgument<T>` для получения переданной рабочей нагрузки. Реальное переданное значение должно получаться косвенно с помощью метода `GetValue()` входного объекта `CodeActivityContext`.

Ниже приведен код специального действия, которое будет генерировать новый файл *.txt, описывающий ситуацию для отдела продаж:

```
public sealed class CreateSalesMemoActivity : CodeActivity
{
    // Два свойства для специального действия.
    public InArgument<string> Make { get; set; }
    public InArgument<string> Color { get; set; }

    // Если действие возвращает значение, унаследовать его от
    // CodeActivity<TResult> и вернуть значение из метода Execute().
    protected override void Execute(CodeActivityContext context)
    {
        // Вывести сообщение в локальный текстовый файл.
        StringBuilder salesMessage = new StringBuilder();
        salesMessage.AppendLine("***** Attention sales team! *****");
        salesMessage.AppendLine("Please order the following ASAP!");
        salesMessage.AppendFormat("1 {0} {1}\n",
            context.GetValue(Color), context.GetValue(Make));
        salesMessage.AppendLine("*****");

        System.IO.File.WriteAllText("SalesMemo.txt", salesMessage.ToString());
    }
}
```

Скомпилируйте сборку рабочего потока. Открыв окно визуального конструктора рабочих потоков в Visual Studio, загляните в верхнюю часть панели инструментов. Там должно появиться специальное действие (рис. 26.32).

Перетащите действие `Sequence` на ветвь `Else` действия `If`. Затем перетащите специальное действие на действие `Sequence`. В этот момент можно присвоить значения каждому из представленных свойств с помощью окна `Properties`. Используя переменные `RequestedMake` и `RequestedColor`, установите свойства `Make` и `Color` действия, как показано на рис. 26.33.

Для завершения рабочего потока перетащите финальное действие `Assign` на действие `Sequence` ветви `Else` и установите в качестве значения `FormattedResponse` строку "Sorry, out of stock" (Сожалею, нет на складе). На рис. 26.34 показан окончательный вид действия `If`.

Скомпилируйте проект и переходите к последней части главы, где будет построен клиентский хост, в котором будет применяться этот рабочий поток.

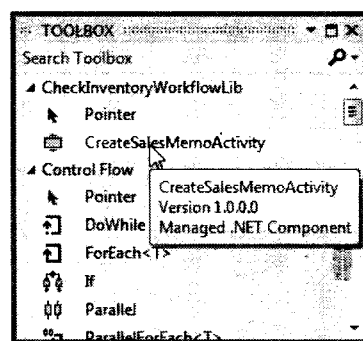


Рис. 26.32. Специальные действия кода появляются в панели инструментов Visual Studio

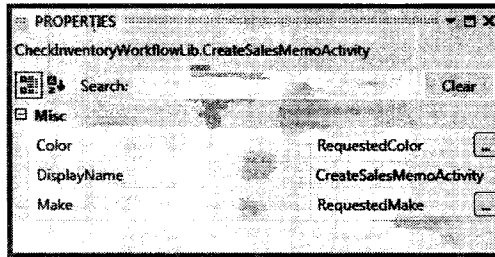


Рис. 26.33. Установка свойств специального действия кода

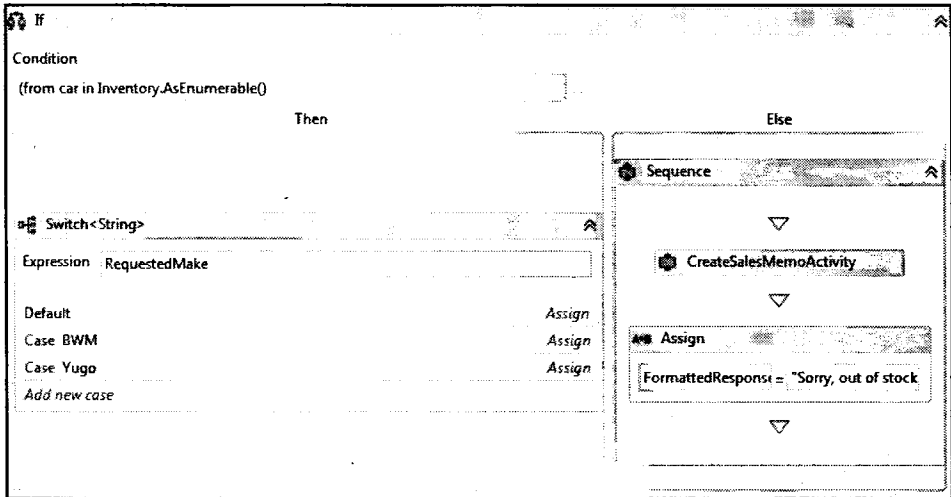


Рис. 26.34. Завершенное действие If

Исходный код. Проект CheckInventoryWorkflowLib доступен в подкаталоге Chapter 26.

Использование библиотеки рабочего потока

Библиотеку рабочего потока может использовать приложение любого рода; однако здесь мы отдадим предпочтение простоте и построим простое консольное приложение под названием WorkflowLibraryClient. После создания проекта понадобится установить ссылки не только на сборки CheckInventoryWorkflowLib.dll и AutoLotDAL.dll, но также и на ключевую библиотеку WF по имени System.Activities.dll. Добавьте указанные ссылки на библиотеки в проект.

Теперь поместите в файл Program.cs следующий код:

```
using System;
...
using CheckInventoryWorkflowLib;
namespace WorkflowLibraryClient
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("**** Inventory Look up ****");
        }
    }
}
```

```

// Получить предпочтения пользователя.
Console.Write("Enter Color: ");
string color = Console.ReadLine();
Console.Write("Enter Make: ");
string make = Console.ReadLine();

// Упаковать данные для рабочего потока.
Dictionary<string, object> wfArgs = new Dictionary<string, object>()
{
    {"RequestedColor", color},
    {"RequestedMake", make}
};
try
{
    // Отправить данные рабочему потоку.
    WorkflowInvoker.Invoke(new CheckInventory(), wfArgs);
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
}
}

```

Как это уже делалось в других примерах, в коде применяется класс `WorkflowInvoker` для запуска рабочего потока в синхронной манере. Теперь давайте посмотрим, каким образом получить возвращаемое значение из рабочего потока. Вспомните, что после завершения рабочего потока, вы должны получить обратно сформатированный ответ.

Извлечение выходного аргумента рабочего потока

Метод `WorkflowInvoker.Invoke()` возвратит объект, реализующий интерфейс `IDictionary<string, object>`. Поскольку рабочий поток может возвращать любое количество выходных аргументов, понадобится указать имя нужного выходного аргумента как строковое значение для индексатора типа. Модифицируйте логику `try/catch`, как показано ниже:

```

try
{
    // Отправить данные рабочему потоку.
    IDictionary<string, object> outputArgs =
        WorkflowInvoker.Invoke(new CheckInventory(), wfArgs);

    // Вывести выходное сообщение на консоль.
    Console.WriteLine(outputArgs["FormattedResponse"]);
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}

```

Теперь можно запустить приложение и ввести изготовителя и цвет автомобиля, имеющегося в вашей копии таблицы `Inventory` базы данных `AutoLot`. Ниже показан результирующий вывод:

```

**** Inventory Look up ****
Enter Color: Black
Enter Make: BMW
Yes sir! We can send you Black BMW as soon as 2/17/2012 9:23:01 PM!
Press any key to continue . . .

```


Если ввести информацию об элементе, которого на складе нет, вывод будет выглядеть следующим образом:

```
**** Inventory Look up ****
Enter Color: Pea Soup Green
Enter Make: Viper
Sorry, out of stock
Press any key to continue . . .
```

Кроме того, вы обнаружите в папке bin\Debug клиентского приложения новый файл *.txt, в котором сохранено напоминание для продавцов:

```
***** Attention sales team! *****
Please order the following ASAP!
1 Pea Soup Green Viper
*****
```

На этом знакомство с API-интерфейсом WF завершено. Хотя в этой главе кратко рассматривались лишь некоторые ключевые аспекты этого API-интерфейса .NET 4.0, был построен достаточный фундамент для дальнейшего самостоятельного изучения этой инфраструктуры.

Исходный код. Проект WorkflowLibraryClient доступен в подкаталоге Chapter 26.

Резюме

По существу инфраструктура WF позволяет моделировать внутренние бизнес-процессы приложения непосредственно в самом приложении. Однако помимо простого моделирования общего рабочего потока в WF предлагается завершенная исполняющая среда и несколько служб, которые дополняют общую функциональность этого API-интерфейса (службы постоянного хранения и отслеживания и т.п.). Несмотря на то что в настоящей главе указанные службы не рассматривались непосредственно, помните, что приложение WF производственного уровня почти наверняка будет использовать их.

В этой вводной главе вы ознакомились с двумя ключевыми действиями верхнего уровня — Flowchart и Sequence. Хотя каждое из них управляет потоком логики уникальным способом, оба они могут содержать одинаковые дочерние действия и выполняться хостом в одинаковой манере (через WorkflowInvoker или WorkflowApplication). Кроме того, вы узнали, как передавать аргументы хоста в рабочий поток с применением обобщенного объекта Dictionary и как получать выходные аргументы из рабочего потока с помощью обобщенного объекта, совместимого с IDictionary.

ЧАСТЬ VII

Windows Presentation Foundation

В этой части

Глава 27. Введение в Windows Presentation Foundation и XAML

Глава 28. Программирование с использованием элементов управления WPF

Глава 29. Службы визуализации графики WPF

Глава 30. Ресурсы, анимация и стили WPF

Глава 31. Свойства зависимости, маршрутизируемые события и шаблоны

Введение в Windows Presentation Foundation и XAML

Когда вышла версия 1.0 платформы .NET, программисты, которым нужно было строить графические настольные приложения, использовали два API-интерфейса под названиями Windows Forms и GDI+, упакованные в основном в сборки System.Windows.Forms.dll и System.Drawing.dll. Хотя Windows Forms и GDI+ — великолепные API-интерфейсы для построения традиционных настольных графических пользовательских интерфейсов, начиная с версии .NET 3.0, также предлагается альтернативный API-интерфейс, который называется Windows Presentation Foundation (WPF).

Эта вводная глава о WPF начинается с рассмотрения мотивации, лежащей в основе этой новой инфраструктуры для построения графических пользовательских интерфейсов, что поможет увидеть разницу между моделями программирования Windows Forms/GDI+ и WPF. Затем мы рассмотрим различные типы приложений WPF, поддерживаемых этим API-интерфейсом, и исследуем роль нескольких важных классов, включая Application, Window, ContentControl, Control, UIElement и FrameworkElement. Попутно вы научитесь перехватывать действия мыши и клавиатуры, определять данные на уровне приложения и решать другие распространенные задачи WPF, не используя ничего, кроме кода C#.

В этой главе будет представлена грамматика на основе XML, которая называется *расширяемым языком разметки приложений* (Extensible Application Markup Language — XAML). Вы изучите синтаксис и семантику XAML (включая синтаксис присоединяемых свойств, роль преобразователей типов и расширений разметки), а также узнаете, как генерировать, загружать и синтаксически анализировать XAML во время выполнения. Кроме того, будет показано, как интегрировать данные XAML в кодовую базу WPF на C# (и связанные с этим преимущества).

Глава завершается рассмотрением визуальных конструкторов WPF, встроенных в Visual Studio. Будет построен собственный специальный редактор/анализатор XAML, который продемонстрирует манипулирование XAML во время выполнения для создания динамического пользовательского интерфейса.

Мотивация, лежащая в основе WPF

На протяжении многих лет в Microsoft разработали многочисленные инструменты для создания графических пользовательских интерфейсов (для низкоуровневой раз-

работки на C/C++/Windows API, VB6, MFC и т.д.), предназначенные для построения настольных приложений. Каждый из этих инструментов предлагает кодовую базу для представления основных аспектов приложения с графическим пользовательским интерфейсом, включая главные окна, диалоговые окна, элементы управления, системы меню и т.п. В начальном выпуске платформы .NET API-интерфейс Windows Forms быстро стал предпочтительной моделью разработки пользовательских интерфейсов, благодаря его простой, но очень мощной объектной модели.

Хотя с помощью Windows Forms было успешно разработано множество полноценных настольных приложений, следует признать, что эта программная модель довольно *асимметрична*. Другими словами, сборки `System.Windows.Forms.dll` и `System.Drawing.dll` не обеспечивают прямой поддержки многих дополнительных технологий, требуемых для построения полнофункционального настольного приложения. Чтобы проиллюстрировать это утверждение, рассмотрим природу разработки графического пользовательского интерфейса, предшествующую WPF (табл. 27.1).

Таблица 27.1. Решения, предшествующие WPF, для обеспечения желаемой функциональности

Желаемая функциональность	Технология
Построение окон с элементами управления	Windows Forms
Поддержка двумерной графики	GDI+ (<code>System.Drawing.dll</code>)
Поддержка трехмерной графики	API-интерфейсы DirectX
Поддержка потокового видео	API-интерфейсы Windows Media Player
Поддержка документов нефиксированного формата	Программное манипулирование PDF-файлами

Как видите, разработчик, применяющий Windows Forms, вынужден заимствовать типы из множества несвязанных API-интерфейсов и объектных моделей. Хотя и верно то, что использование всех этих разнообразных API-интерфейсов синтаксически похоже (в конце концов, это просто код C#), каждая технология требует радикально иного мышления. Например, навыки, необходимые для создания трехмерной анимации с использованием DirectX, совершенно отличаются от тех, что нужны для привязки данных к экранной сетке. По правде говоря, программисту Windows Forms чрезвычайно трудно в равной мере овладеть природой каждого из этих API-интерфейсов.

Унификация различных API-интерфейсов

Инфраструктура WPF (появившаяся в .NET 3.0) специально создавалась для того, чтобы объединить все эти ранее несвязанные программистские задачи в единую объектную модель. Таким образом, при разработке трехмерной анимации больше не возникает необходимости в ручном кодировании с использованием API-интерфейсов DirectX (хотя это можно делать), поскольку нужная функциональность уже встроена в WPF. Чтобы продемонстрировать, насколько все стало яснее, в табл. 27.2 представлена модель разработки настольных приложений, введенная в .NET 3.0.

Очевидное преимущество здесь в том, что программисты .NET теперь имеют единый *симметричный* API-интерфейс для всех распространенных потребностей, связанных с построением графических пользовательских интерфейсов. Освоив функциональность ключевых сборок WPF и грамматику XAML, вы будете приятно удивлены, насколько быстро с их помощью можно создавать очень сложные пользовательские интерфейсы.

Таблица 27.2. Решения .NET 3.0 для обеспечения желаемой функциональности

Желаемая функциональность	Технология
Построение форм с элементами управления	WPF
Поддержка двумерной графики	WPF
Поддержка трехмерной графики	WPF
Поддержка потокового видео	WPF
Поддержка документов нефиксированного формата	WPF

Обеспечение разделения ответственности через XAML

Возможно, одним из наиболее значительных преимуществ WPF стал способ четкого отделения внешнего вида и поведения приложения с графическим пользовательским интерфейсом от программной логики, которая им управляет. Используя XAML, можно определить пользовательский интерфейс приложения через *разметку XML*. Эта разметка (в идеале генерируемая с помощью таких инструментов, как Microsoft Visual Studio или Microsoft Expression Blend) затем может быть подключена к связанному файлу кода для обеспечения деталей функциональности программы.

На заметку! Применение XAML не ограничивается приложениями WPF. Любое приложение может использовать XAML для описания дерева объектов .NET, даже если они не имеют никакого отношения к визуальному пользовательскому интерфейсу. Например, в API-интерфейсе Windows Workflow Foundation грамматика, основанная на XAML, применяется для определения бизнес-процессов и специальных действий. Кроме того, XAML используют все прочие платформы для построения графических пользовательских интерфейсов в .NET, такие как приложения Silverlight, Windows Phone 7 и Windows 8.

По мере погружения в WPF вы можете удивиться, насколько высокую гибкость обеспечивает эта “разметка рабочего стола”. XAML позволяет определять не только простые элементы пользовательского интерфейса (кнопки, таблицы, окна списков и т.п.) в разметке, но также интерактивную двух- и трехмерную графику, анимацию, логику привязки данных и функциональность мультимедиа (вроде воспроизведения видео).

XAML также делает очень простой настройку визуализации элемента управления. Например, определение круглой кнопки, на которой выполняется анимация логотипа компании, требует всего нескольких строк разметки. Как будет показано в главе 31, элементы управления WPF могут быть модифицированы через стили и шаблоны, что позволяет изменять целиком весь внешний вид приложения с минимальными усилиями. В отличие от разработки с помощью Windows Forms, единственной веской причиной построения специального элемента управления WPF с нуля является необходимость в изменении *поведения* элемента управления (например, добавление специальных методов, свойств или событий, создание подкласса существующего элемента управления с целью переопределения виртуальных членов). Если вы просто хотите изменить *внешний вид* элемента управления (как в случае с круглой анимированной кнопкой), это можно сделать полностью через разметку.

Обеспечение оптимизированной модели визуализации

Наборы инструментов для построения графических пользовательских интерфейсов, такие как Windows Forms, MFC или VB6, предварительно формируют все запросы на

графическую визуализацию (включая визуализацию элементов управления наподобие кнопок и окон списков) с использованием низкоуровневого API-интерфейса, основанного на языке C (GDI), который является составной частью операционной системы Windows на протяжении многих лет. GDI обеспечивает адекватную производительность для типовых бизнес-приложения или простых графических программ; однако если пользовательскому интерфейсу приложения нужна была высокопроизводительная графика, приходилось обращаться к DirectX.

Программная модель WPF существенно отличается в том, что при визуализации графических данных GDI *не используется*. Все операции визуализации (например, двух- и трехмерная графика, анимация, визуализация элементов управления и т.д.) теперь используют API-интерфейс DirectX. Очевидная выгода такого подхода заключается в том, что приложение WPF будет автоматически получать преимущества аппаратной и программной оптимизации. К тому же приложения WPF могут задействовать очень развитые графические службы (эффекты размытия, сглаживания, прозрачности и т.п.) без сложностей, присущих программированию с прямым применением API-интерфейса DirectX.

На заметку! Хотя WPF переносит все запросы визуализации на уровень DirectX, нельзя утверждать, что приложение WPF будет работать столь же быстро, как приложение, построенное на основе неуправляемого C++ и DirectX. Если вы намереваетесь построить настольное приложение, которое требует максимально возможной скорости выполнения (вроде трехмерной игры), то неуправляемый C++ и DirectX по-прежнему остаются наилучшим подходом.

Упрощение программирования сложных пользовательских интерфейсов

Чтобы закрепить тему, повторимся еще раз: Windows Presentation Foundation (WPF) — это API-интерфейс, предназначенный для построения настольных приложений, который интегрирует различные настольные API-интерфейсы в единую объектную модель и обеспечивает четкое разделение ответственностей через XAML. В дополнение к этим важнейшим моментам, приложения WPF также выигрывают от очень простого способа интеграции со службами, которые исторически были достаточно сложными. Ниже кратко перечислены основные функциональные возможности WPF.

- Множество диспетчеров компоновки (намного больше, чем в Windows Forms) для обеспечения исключительно гибкого контроля над размещением и изменением позиций содержимого.
- Использование расширенного механизма привязки данных для связи содержимого с элементами пользовательского интерфейса разнообразными способами.
- Встроенный механизм стилей, позволяющий определять “темы” для приложения WPF.
- Применение векторной графики, которая позволяет автоматически изменять размеры содержимого для соответствия размеру и разрешению экрана, размещающего приложение.
- Поддержка двух- и трехмерной графики, анимации и воспроизведения видео и аудио.
- Развитый типографский API-интерфейс, поддерживающий документы XML Paper Specification (XPS), фиксированные документы (WYSIWYG), документы нефиксированного формата и аннотации в документах (например, API-интерфейс Sticky Notes).

- Поддержка взаимодействия с унаследованными моделями графических пользовательских интерфейсов (т.е. Windows Forms, ActiveX и Hwnd-дескрипторы Win32). Например, в приложение WPF можно встраивать специальные элементы управления Windows Forms и наоборот.

Теперь, имея представление о вкладе WPF в платформу, давайте рассмотрим различные типы приложений, которые могут быть созданы с использованием этого API-интерфейса. Многие из перечисленных выше возможностей будут детально исследоваться в последующих главах.

Различные варианты приложений WPF

API-интерфейс WPF может использоваться для построения широкого разнообразия приложений с графическим пользовательским интерфейсом, которые в основном отличаются структурой навигации и моделями развертывания. Ниже будут представлены их краткие описания.

Традиционные настольные приложения

Первый (и наиболее популярный) вариант предусматривает применение WPF для построения традиционной исполняемой сборки, которая запускается на локальной машине. Например, WPF можно было бы использовать для построения текстового редактора, программы рисования или мультимедийной программы, такой как цифровой музыкальный проигрыватель, средство просмотра фотографий и т.д. Подобно любому другому настольному приложению, эти файлы *.exe могут устанавливаться традиционными средствами (программами установки, пакетами Windows Installer и т.п.) или же посредством технологии ClickOnce, позволяющей распространять и устанавливать настольные приложения через удаленный веб-сервер.

Говоря языком программирования, этот тип приложений WPF, в дополнение к ожидаемому набору диалоговых окон, панелей инструментов, панелей состояния, систем меню и прочих элементов пользовательского интерфейса, будет работать (как минимум) с классами Window и Application.

WPF позволяет создавать как простые бизнес-приложения без каких-либо излишеств, так и встраивать средства подобного рода. На рис. 27.1 показан пример настольного приложения WPF для просмотра медицинских карточек пациентов в учреждении здравоохранения.

К сожалению, на печатной странице невозможно отразить весь набор средств этой программы. Например, запустив это приложение, вы увидите, что в правом верхнем углу главного окна отображается график реального времени, показывающий ритм синуса пациента. Если щелкнуть на кнопке Patient Details (Информация о пациенте) в нижнем правом углу, произойдут несколько анимаций, которые трансформируют пользовательский интерфейс к виду, показанному на рис. 27.2.

Можно ли построить подобное приложение без WPF? Безусловно. Однако объем и сложность кода будут намного выше.

На заметку! Этот пример приложения, а также многие другие, доступен для загрузки (вместе с исходным кодом) на официальном веб-сайте WPF по адресу <http://windowsclient.net>. На этом сайте вы найдете множество документов по WPF (и Windows Forms), примеров проектов, демонстраций технологии и форумов.

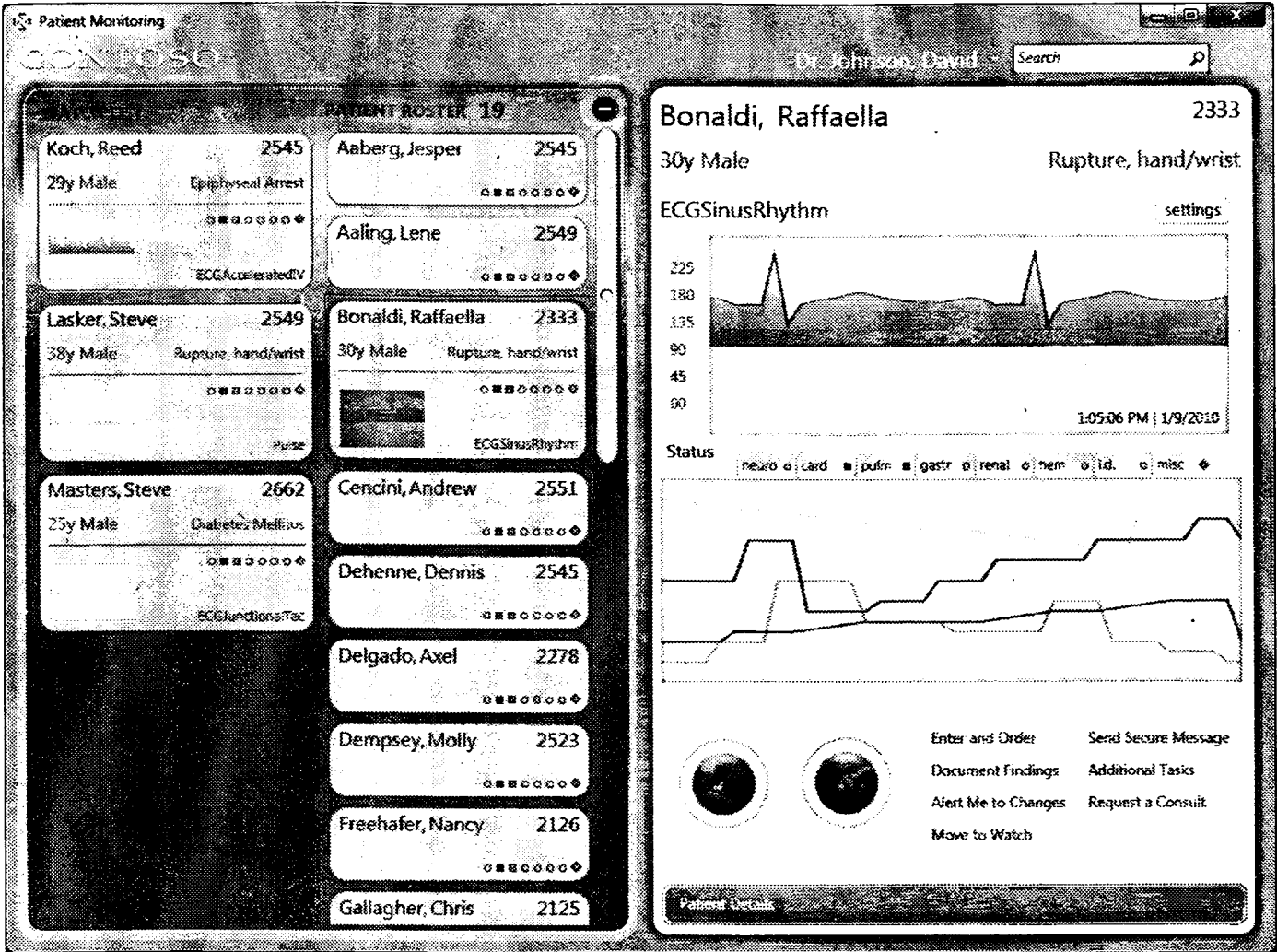


Рис. 27.1. Настольное приложение WPF с развитым интерфейсом



Рис. 27.2. В WPF очень легко реализуются трансформации и анимации

WPF-приложения на основе навигации

Приложения WPF могут также использовать структуру на основе навигации, которая позволяет традиционному настольному приложению вести себя подобно приложению веб-браузера. Применяя эту модель, можно построить настольную программу *.exe, включающую в себя кнопки “вперед” и “назад”, которые позволяют конечному пользователю перемещаться вперед и назад по различным экранам пользовательского интерфейса, именуемым *страницами*.

Приложение такого типа поддерживает список всех страниц и обеспечивает необходимую инфраструктуру для навигации по ним, передачи данных между страницами (подобно переменным в веб-приложении) и поддержки списка хронологии. Для примера взгляните на проводник Windows (рис. 27.3), в котором используется функциональность подобного рода. Обратите внимание на кнопки навигации (и список хронологии), находящийся в верхнем левом углу окна.

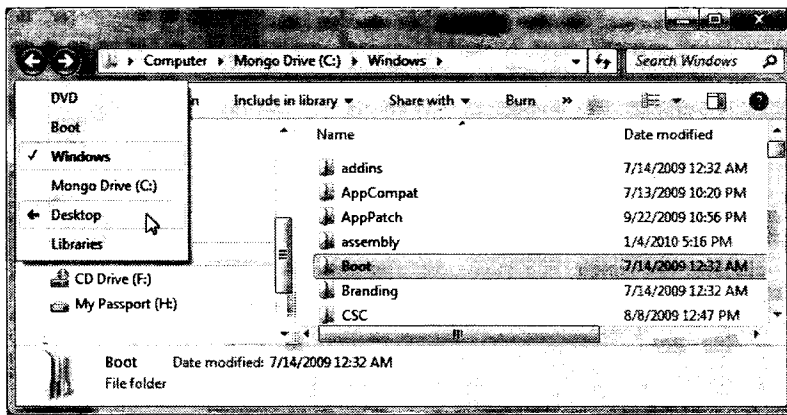


Рис. 27.3. Настольная программа на основе навигации

Несмотря на то что настольное приложение WPF может принимать веб-подобную схему навигации, помните, что это всего лишь вопрос дизайна пользовательского интерфейса. Само приложение остается в виде той же исполняемой сборки, запускаемой на настольной машине, и помимо внешнего сходства не имеет никакого отношения к веб-приложениям. Говоря языком программирования, эта разновидность WPF-приложений построена с применением таких классов, как *Application*, *Page*, *NavigationWindow* и *Frame*.

Приложения XBAP

WPF также позволяет строить приложения, которые могут размещаться *внутри* веб-браузера. Такая разновидность приложений WPF называется *браузерным приложением* XAML, или XBAP. В рамках этой модели конечный пользователь переходит по заданному URL-адресу, указывающему на приложение XBAP (которое, в сущности, является коллекцией объектов *Page*), затем прозрачно загружает и устанавливает его на локальной машине. В отличие от традиционной установки исполняемого приложения с помощью ClickOnce, программа XBAP размещается непосредственно в браузере и принимает встроенную систему навигации браузера. На рис. 27.4 показана XBAP-программа в действии (а именно — пример программы WPF ExpensesIt, который можно найти на сайте <http://windowsclient.net>).

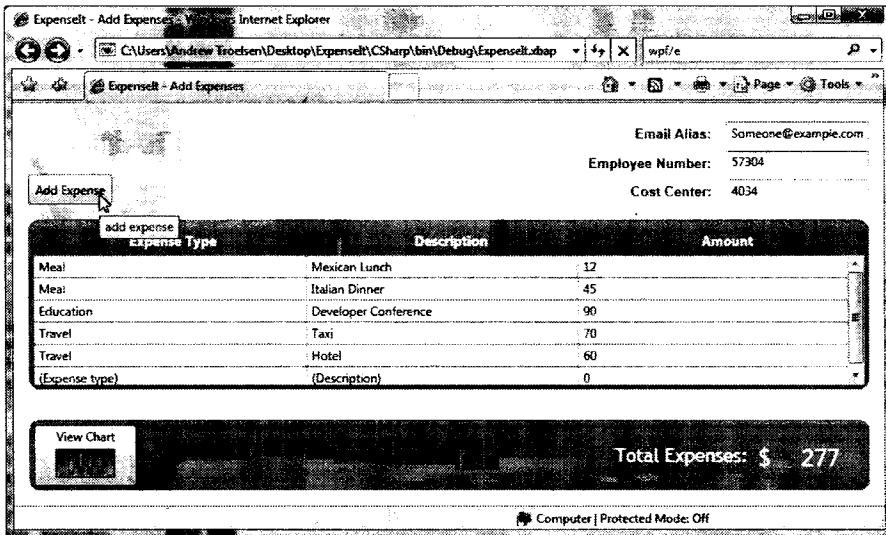


Рис. 27.4. Программы XBAP загружаются на локальную машину и размещаются внутри веб-браузера

Однако из преимуществ технологии XBAP состоит в том, она позволяет создавать сложные пользовательские интерфейсы, которые являются намного более выразительными, чем типичная веб-страница, построенная с помощью HTML и JavaScript (хотя HTML5, конечно, улучшает текущую ситуацию). Объект `Page` в WPF может пользоваться теми же службами, что и настольное приложение WPF, включая анимацию, двух- и трехмерную графику, темы и т.д. По сути, веб-браузер в данном случае — это просто контейнер для объектов `Page`, а не средство отображения веб-страниц ASP.NET.

Однако, учитывая, что объекты `Page` развертываются на удаленном веб-сервере, приложения XBAP можно легко сопровождать в разных версиях и обновлять без необходимости повторного развертывания исполняемых сборок на пользовательских настольных машинах. Подобно традиционному веб-приложению, объекты `Page` можно легко обновлять на веб-сервере, и пользователь всегда будет получать самую актуальную версию, обращаясь по заданному URL-адресу.

Возможным недостатком этой разновидности программ WPF является то, что XBAP могут работать только внутри веб-браузеров Microsoft Internet Explorer или Firefox. При развертывании приложений XBAP в корпоративной сети компании совместимость браузеров не должна быть проблемой, т.к. системные администраторы могут просто диктовать выбор браузера, обязательного для установки на машинах пользователей. Тем не менее, открывая доступ к приложению XBAP из внешнего мира, невозможно гарантировать, что каждый пользователь будет работать с браузером Internet Explorer или Firefox, а потому некоторые внешние пользователи просто не смогут его просмотреть.

Другая проблема состоит в том, что машина, которая выполняет XBAP-приложение, должна иметь установленную локальную копию платформы .NET, поскольку объекты `Page` пользуются теми же сборками .NET, что и традиционные приложения. Учитывая это, XBAP-приложения ограничены только средами Windows и не могут просматриваться на системах, работающих под управлением Mac OS X или Linux.

Отношения между WPF и Silverlight

WPF и XAML также предоставляют фундамент для межплатформенной, межбраузерной, основанной на WPF технологии, которая называется *Silverlight*. На самом высоком

уровне Silverlight можно считать скорее конкурентом Adobe Flash, но с преимуществами использования C# и XAML, а не новым набором инструментов и языков. Silverlight является подмножеством функциональности WPF, применяемым для построения интерактивных подключаемых модулей для более крупных страниц на основе HTML. Однако в действительности Silverlight — это совершенно уникальный дистрибутив платформы .NET, включающий в себя уменьшенные версии среды CLR и библиотек базовых классов .NET.

В отличие от XBAP, устанавливать .NET Framework на машине пользователя не требуется. До тех пор, пока целевая машина имеет установленную исполняющую среду Silverlight, браузер будет загружать ее и отображать приложения Silverlight автоматически. А лучше всего то, что подключаемые модули Silverlight не ограничены операционными системами Windows. В Microsoft также разработали исполняющую среду Silverlight для Mac OS.

С помощью Silverlight можно строить исключительно многофункциональные (и интерактивные) веб-приложения. Например, подобно WPF, Silverlight обладает системой векторной графики, поддержкой анимации и поддержкой мультимедиа. Более того, в приложения можно внедрять подмножество библиотеки базовых классов .NET. Это подмножество включает API-интерфейсы LINQ, обобщенные коллекции, поддержку WCF и полезное подмножество mscorlib.dll (файловый ввод-вывод, манипулирование XML и т.д.).

На заметку! В этой книге технология Silverlight подробно не рассматривается, однако большая часть знаний WPF напрямую отображается на конструкцию подключаемых модулей Silverlight. Дополнительные сведения об этом API-интерфейсе доступны по адресу www.silverlight.net.

Исследование сборок WPF

Независимо от того, какого типа приложение WPF вы собираетесь строить, в конечном итоге WPF — это лишь немногим более чем коллекция типов, встраиваемых в сборки WPF. В табл. 27.3 описаны ключевые сборки, используемые для построения приложений WPF, ссылка на каждую из которых должна включаться при создании нового проекта. Как и следовало ожидать, WPF-проекты в Visual Studio и Expression Blend автоматически получают ссылки на эти обязательные сборки.

Таблица 27.3. Основные сборки WPF

Сборка	Описание
PresentationCore.dll	В этой сборке определены многочисленные пространства имен, образующие фундамент уровня графического пользовательского интерфейса в WPF. Например, она включает поддержку API-интерфейса WPF Ink (для программирования перьевого ввода для Pocket PC и Tablet PC), примитивы анимации и множество типов графической визуализации
PresentationFramework.dll	Эта сборка содержит большинство элементов управления WPF, классы Application и Window, поддержку интерактивной двухмерной графики и многочисленные типы, применяемые для привязки данных
System.Xaml.dll	Эта сборка предоставляет пространства имен, которые позволяют программно взаимодействовать с документами XAML во время выполнения. В основном эта сборка полезна только при разработке инструментов поддержки WPF или когда нужен абсолютный контроль над XAML во время выполнения

Сборка	Описание
WindowsBase.dll	В этой сборке определены типы, образующие инфраструктуру API-интерфейса WPF. Сюда входят типы потоков WPF, типы безопасности, различные преобразователи типов и поддержка свойств зависимости и маршрутизируемых событий (которые рассматриваются в главе 31)

Все вместе эти четыре сборки определяют ряд новых пространств имен и сотни новых классов, интерфейсов, структур, перечислений и делегатов .NET. Хотя за полной информацией следует обращаться в документацию .NET Framework 4.5 SDK, в табл. 27.4 описаны некоторые основные пространства имен.

Таблица 27.4. Основные пространства имен WPF

Пространство имен	Описание
System.Windows	Это корневое пространство имен WPF. Здесь вы найдете основные классы (такие как <code>Application</code> и <code>Window</code>), которые требуются в любом настольном проекте WPF
System.Windows.Controls	Содержит все ожидаемые графические элементы (виджеты) WPF, включая типы для построения систем меню, всплывающих подсказок и многочисленные диспетчеры компоновки
System.Windows.Data	Содержит типы для работы с механизмом привязки данных WPF, а также для поддержки шаблонов привязки данных
System.Windows.Documents	Содержит типы для работы с API-интерфейсом документов, который позволяет интегрировать в WPF-приложения функциональность в стиле PDF через протокол XML Paper Specification (XPS)
System.Windows.Ink	Предоставляет поддержку Ink API — интерфейса, который позволяет получать ввод от пера или мыши, реагировать на входные жесты и т.д. Этот API-интерфейс очень полезен при программировании для Tablet PC, однако может использоваться и в любых WPF-приложениях
System.Windows.Markup	В этом пространстве имен определено множество типов, обеспечивающих программный анализ и обработку разметки XAML (и эквивалентного двоичного формата BAML)
System.Windows.Media	Это корневое пространство имен для нескольких пространств имен, связанных с мультимедиа. Внутри таких пространств имен определены типы для работы с анимацией, визуализацией трехмерной графики, визуализацией текста и прочие мультимедийные примитивы
System.Windows.Navigation	Это пространство имен предоставляет типы для обеспечения логики навигации, используемой браузерными приложениями XAML (XBAP), а также настольными приложениями, требующими страничной навигационной модели
System.Windows.Shapes	В этом пространстве имен определены классы, которые позволяют визуализировать двухмерную графику, автоматически реагирующую на ввод с помощью мыши

Чтобы начать путешествие по программной модели WPF, давайте рассмотрим два члена пространства имен `System.Windows`, которые являются общими при традиционной разработке любого настольного приложения: `Application` и `Window`.

На заметку! Если вы создавали пользовательские интерфейсы настольных приложений с помощью API-интерфейса `Windows Forms`, имейте в виду, что сборки `System.Windows.Forms.*` и `System.Drawing.*` не имеют отношения к WPF. Эти библиотеки представляют первоначальный инструментальный набор .NET для построения графических пользовательских интерфейсов, т.е. `Windows Forms/GDI+`.

Роль класса `Application`

Класс `System.Windows.Application` представляет глобальный экземпляр работающего приложения WPF. В этом классе предусмотрен метод `Run()` (для запуска приложения), комплект событий, которые можно обрабатывать для взаимодействия с приложением на протяжении его времени жизни (вроде `Startup` и `Exit`), и ряд членов, специфичных для браузерных приложений XAML (таких как события, инициируемые при перемещении пользователя по страницам). В табл. 27.5 описаны важные ключевые свойства, о которых нужно знать.

Таблица 27.5. Ключевые свойства типа `Application`

Свойство	Описание
<code>Current</code>	Это статическое свойство позволяет получить доступ к работающему объекту <code>Application</code> из любого места кода. Это может быть очень полезно, когда обычному или диалоговому окну необходим доступ к объекту <code>Application</code> , который его создал, обычно для взаимодействия с переменными или функциональностью уровня приложения
<code>MainWindow</code>	Это свойство позволяет программно получать и устанавливать главное окно приложения
<code>Properties</code>	Это свойство позволяет устанавливать и получать данные, доступные через все аспекты приложения WPF (окна, диалоговые окна и т.п.)
<code>StartupUri</code>	Это свойство получает или устанавливает URI, который указывает окно или страницу для автоматического открытия при запуске приложения
<code>Windows</code>	Это свойство возвращает тип <code>WindowCollection</code> , который предоставляет доступ ко всем окнам, которые созданы в потоке, создавшем объект <code>Application</code> . Это может весьма пригодиться, когда необходимо выполнить итерацию по всем открытым окнам приложения и изменить их состояние (например, свернуть все окна)

Конструирование класса `Application`

Любое WPF-приложение нуждается в определении класса, который расширяет класс `Application`. Внутри этого класса определяется точка входа программы (метод `Main()`), которая создает экземпляр данного подкласса и обычно обрабатывает события `Startup` и `Exit`. Чуть ниже мы рассмотрим полноценный проект, а пока ниже представлен краткий пример:

```
// Определяет глобальный объект приложения для данной программы WPF.
class MyApp : Application
{
    [STAThread]
```

```
static void Main(string[] args)
{
    // Создать объект приложения.
    MyApp app = new MyApp();

    // Зарегистрировать события Startup/Exit.
    app.Startup += (s, e) => { /* Запуск приложения */ };
    app.Exit += (s, e) => { /* Завершение приложения */ };
}
}
```

Чаще всего в обработчике события `Startup` будут обрабатываться входные аргументы командной строки и запускаться главное окно программы. Обработчик `Exit`, как и следовало ожидать — это место, где размещается необходимая логика завершения программы (сохранение пользовательских предпочтений, запись в реестр Windows и т.п.).

Перечисление элементов коллекции *Windows*

Еще одним интересным свойством класса `Application` является `Windows`, предоставляющее доступ к коллекции, в которой представлены все окна, загруженные в память для текущего WPF-приложения. Вспомните, что создаваемые новые объекты `Window` автоматически добавляются в коллекцию `Application.Windows`. Ниже приведен пример метода, который сворачивает все окна приложения (возможно, в ответ на нажатие определенного сочетания клавиш или выбор пункта меню конечным пользователем):

```
static void MinimizeAllWindows()
{
    foreach (Window wnd in Application.Current.Windows)
    {
        wnd.WindowState = WindowState.Minimized;
    }
}
```

В примере, рассматриваемом далее в главе, будет построен завершенный тип, производный от `Application`. А пока что давайте рассмотрим основную функциональность типа `Window` и изучим в процессе ряд важных базовых классов WPF.

Роль класса *Window*

Класс `System.Windows.Window` (расположенный в сборке `PresentationFramework.dll`) представляет одиночное окно, которым владеет производный от `Application` класс, включая все диалоговые окна, отображаемые главным окном. Как и можно было ожидать, тип `Window` имеет ряд родительских классов, каждый из которых привносит свою функциональность. На рис. 27.5 показана цепочка наследования (и реализованные интерфейсы) для `System.Windows.Window`, как она выглядит в браузере объектов Visual Studio.

По мере чтения этой и последующих глав, вы начнете понимать функциональность, предлагаемую многими базовыми классами WPF. В следующем разделе будет предоставлен краткий перечень функциональности, предлагаемой каждым базовым классом (полные сведения ищите в документации .NET Framework 4.5 SDK).

Роль класса *System.Windows.Controls.ContentControl*

Непосредственным предком `Window` является класс `ContentControl`, который вполне можно считать наиболее впечатляющим из всех классов WPF. Этот базовый класс обеспечивает производные типы способностью размещать в себе одиночный фрагмент *содержимого*, который, попросту говоря, относится к визуальным данным, помещенным внутри области элемента управления через свойство `Content`.

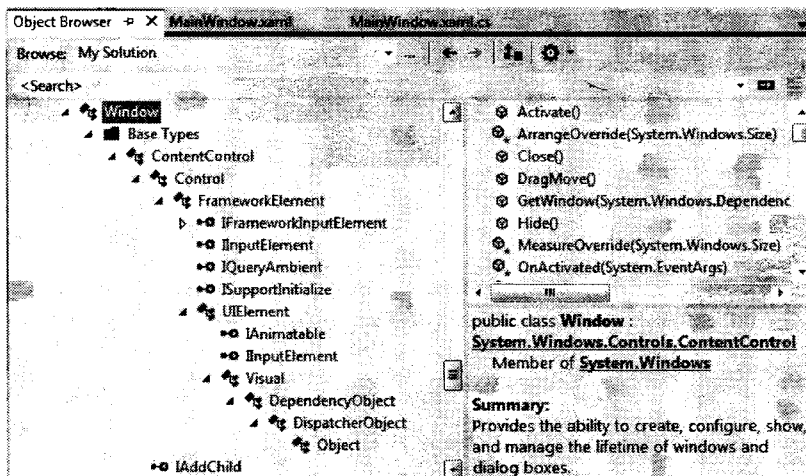


Рис. 27.5. Иерархия наследования класса Window

Модель содержимого WPF позволяет очень легко настраивать базовый вид и поведение элемента управления ContentControl.

Например, когда речь идет о типичном “кнопочном” элементе управления, то обычно предполагается, что его содержимым будет простой строковый литерал (OK, Cancel, Abort и т.п.). Если для описания элемента управления WPF используется XAML, а значение, которое необходимо присвоить свойству Content, может быть выражено в виде простой строки, то установить свойство Content внутри открывающего определения элемента допускается так, как показано ниже:

```
<!-- Установка значения Content в открывающем элементе -->
<Button Height="80" Width="100" Content="OK"/>
```

На заметку! Свойство Content может также устанавливаться в коде C#, что позволяет изменять внутренности элемента управления во время выполнения.

Тем не менее, содержимое может быть практически любым. Например, предположим, что нужна “кнопка”, которая содержит в себе нечто более интересное, чем простую строку — скажем, специальную графику или текстовый фрагмент. На других платформах построения пользовательских интерфейсов, таких как Windows Forms, потребовалось бы создавать специальный элемент управления, что могло повлечь за собой написание значительного объема кода и сопровождения полностью нового класса. Благодаря модели содержимого WPF, это не требуется.

Когда для свойства Content должно быть установлено значение, которое не может быть выражено простым массивом символов, его нельзя присвоить с использованием атрибута в открывающем определении элемента управления. Вместо этого понадобится определить данные содержимого неявно, внутри области действия элемента. Например, показанный ниже элемент <Button> включает в качестве содержимого элемент <StackPanel>, который сам содержит некоторые уникальные данные (а именно — <Ellipse> и <Label>):

```
<!-- Неявная установка в свойстве Content сложных данных -->
<Button Height="80" Width="100">
  <StackPanel>
    <Ellipse Fill="Red" Width="25" Height="25"/>
    <Label Content="OK!"/>
  </StackPanel>
</Button>
```

Для установки сложного содержимого можно также использовать синтаксис “свойство-элемент” языка XAML. Рассмотрим следующее функционально эквивалентное определение `<Button>`, которое устанавливает свойство `Content` явно с помощью синтаксиса “свойство-элемент” (дополнительная информация о XAML будет представлена далее в главе, так что пока не обращайтесь внимания на детали):

```
<!-- Установка свойства Content с использованием синтаксиса 'свойство-элемент'
-->
<Button Height="80" Width="100">
  <Button.Content>
    <StackPanel>
      <Ellipse Fill="Red" Width="25" Height="25"/>
      <Label Content="OK!" />
    </StackPanel>
  </Button.Content>
</Button>
```

Имейте в виду, что не каждый элемент WPF порожден от `ContentControl` и потому не все элементы поддерживают эту уникальную модель содержимого (хотя большинство элементов делают это). К тому же некоторые элементы управления WPF вносят несколько усовершенствований в только что рассмотренную модель. В главе 28 роль содержимого WPF рассматривается более подробно.

Роль класса `System.Windows.Controls.Control`

В отличие от `ContentControl`, все элементы управления WPF разделяют в качестве общего предка базовый класс `Control`. Этот базовый класс предоставляет множество членов, незаменимых для обеспечения функциональности пользовательского интерфейса. Например, `Control` определяет свойства для установки размеров элемента управления, прозрачности, порядка обхода по нажатию клавиши `<Tab>`, дисплейного курсора, цвета фона и т.д. Более того, этот родительский класс обеспечивает поддержку *шаблонных служб*. Как объясняется в главе 30, элементы управления WPF могут полностью изменять свой внешний вид при визуализации, используя шаблоны и стили. В табл. 27.6 описаны некоторые ключевые члены типа `Control`, сгруппированные по функциональности.

Таблица 27.6. Ключевые члены типа `Control`

Член	Описание
<code>Background</code> , <code>Foreground</code> , <code>BorderBrush</code> , <code>BorderThickness</code> , <code>Padding</code> , <code>HorizontalContentAlignment</code> , <code>VerticalContentAlignment</code>	Эти свойства позволяют устанавливать базовые настройки, касающиеся того, как элемент управления будет визуализироваться и позиционироваться
<code>FontFamily</code> , <code>FontSize</code> , <code>FontStretch</code> , <code>FontWeight</code>	Эти свойства управляют настройками шрифтов
<code>IsTabStop</code> , <code>TabIndex</code>	Эти свойства используются для установки порядка обхода элементов управления в окне по нажатию <code><Tab></code>
<code>MouseDoubleClick</code> , <code>PreviewMouseDoubleClick</code>	Эти события обрабатывают двойной щелчок на виджете
<code>Template</code>	Это свойство позволяет получать и устанавливать шаблон элемента, который может быть использован для изменения вывода визуализации виджета

Роль класса `System.Windows.FrameworkElement`

Этот базовый класс предоставляет множество низкоуровневых членов, которые используются повсюду в WPF, например, для поддержки раскадровки (для анимации), привязки данных, а также возможности именования членов (через свойство `Name`), получения ресурсов, определенных производным типом, и установки общих измерений производного типа. Ключевые члены класса `FrameworkElement` кратко описаны в табл. 27.7.

Таблица 27.7. Ключевые члены типа `FrameworkElement`

Член	Описание
<code>ActualHeight</code> , <code>ActualWidth</code> , <code>MaxHeight</code> , <code>MaxWidth</code> , <code>MinHeight</code> , <code>MinWidth</code> , <code>Height</code> , <code>Width</code>	Управляют размерами производного типа
<code>ContextMenu</code>	Получает или устанавливает всплывающее меню, ассоциированное с производным типом
<code>Cursor</code>	Получает или устанавливает курсор мыши, ассоциированный с производным типом
<code>HorizontalAlignment</code> , <code>VerticalAlignment</code>	Управляет позиционированием типа внутри контейнера (такого как панель или окно списка)
<code>Name</code>	Позволяет назначать имя типу, чтобы обращаться к его функциональности в файле кода
<code>Resources</code>	Предоставляет доступ к любому ресурсу, определенному типом (система ресурсов WPF объясняется в главе 30)
<code>ToolTip</code>	Получает или устанавливает всплывающую подсказку, ассоциированную с производным типом

Роль класса `System.Windows.UIElement`

Из всех типов в цепочке наследования `Window` максимальный объем функциональности обеспечивает базовый класс `UIElement`. Его основная задача — предоставить производному типу многочисленные события, чтобы он мог получать фокус и обрабатывать входные запросы. Например, в этом классе предусмотрены многочисленные события для обслуживания операций перетаскивания, перемещений курсора мыши, клавиатурного ввода и ввода с помощью пера (для `Pocket PC` и `Tablet PC`).

Модель событий WPF будет подробно описана в главе 29; однако многие из основных событий покажутся знакомыми (`MouseMove`, `KeyUp`, `MouseDown`, `MouseEnter`, `MouseLeave` и т.п.). В дополнение к определению десятков событий, этот родительский класс предоставляет множество свойств, которые предназначены для управления фокусом, состоянием доступности, видимостью и логикой проверки попаданий (табл. 27.8).

Таблица 27.8. Ключевые члены типа `UIElement`

Член	Описание
<code>Focusable</code> , <code>IsFocused</code>	Эти свойства позволяют устанавливать фокус на заданный производный тип
<code>IsEnabled</code>	Это свойство позволяет управлять доступностью заданного производного типа

Член	Описание
IsMouseDown, IsMouseOver	Эти свойства предлагают простой способ выполнения логики проверки попадания
IsVisible, Visibility	Эти свойства позволяют работать с установкой видимости производного типа
RenderTransform	Это свойство позволяет устанавливать трансформацию, которая будет использована для визуализации производного типа

Роль класса `System.Windows.Media.Visual`

Класс `Visual` предлагает основную поддержку визуализации в WPF, включая проверку попадания на графические данные, трансформацию координат и вычисление ограничивающего прямоугольника. Фактически для рисования данных на экране класс `Visual` взаимодействует с подсистемой `DirectX`. Как будет показано в главе 29, WPF поддерживает три возможных способа визуализации графических данных, каждый из которых отличается от других в отношении функциональности и производительности. Применение типа `Visual` (и его потомков вроде `DrawingVisual`) является наиболее легковесным способом визуализации графических данных, но также подразумевает участие большого объема управляемого кода для обеспечения работы всех необходимых служб. Более подробно об этом речь пойдет в главе 29.

Роль класса `System.Windows.DependencyObject`

WPF поддерживает специальную разновидность свойств .NET, именуемую *свойствами зависимости*. Выражаясь упрощенно, этот стиль свойств предоставляет дополнительный код, чтобы дать возможность свойству реагировать на отдельные технологии WPF, такие как стили, привязка данных, анимация и т.д. Чтобы тип поддерживал такую схему свойств, он должен быть порожден от базового класса `DependencyObject`. Хотя свойства зависимости являются ключевым аспектом разработки WPF, большую часть времени их детали скрыты. В главе 28 содержатся дополнительные сведения о свойствах зависимости.

Роль класса `System.Windows.Threading.DispatcherObject`

Последним базовым классом для типа `Window` (помимо `System.Object`, который здесь не требует дополнительных пояснений) является `DispatcherObject`. Этот класс определяет одно свойство, представляющее интерес — `Dispatcher`, которое возвращает ассоциированный объект `System.Windows.Threading.Dispatcher`. Класс `Dispatcher` — это точка входа в очередь событий приложения WPF, и он предоставляет базовые конструкции для работы с параллелизмом и многопоточностью.

Построение приложения WPF без XAML

Учитывая всю функциональность, предоставляемую родительскими классами типа `Window`, представить окно в приложении можно, либо напрямую создав объект `Window`, либо указав этот класс в качестве родительского для строго типизированного наследника. В следующем примере демонстрируются оба подхода. Хотя в большинстве приложений WPF используется XAML, формально так поступать не обязательно. Все, что может быть выражено в XAML, также можно выразить в коде и (по большей части) наоборот.

При желании можно построить полный проект WPF, используя лежащую в основе объектную модель и процедурный код C#.

Чтобы проиллюстрировать сказанное, давайте построим минимальное, однако полное приложение без применения XAML, работая с классами `Application` и `Window` напрямую. Создайте новое консольное приложение (шаблон проекта WPF в Visual Studio будет использоваться позже) по имени `WpfAppAllCode`. Откройте диалоговое окно `Add Reference` (Добавление ссылки) и добавьте ссылки на сборки `WindowBase.dll`, `PresentationCore.dll`, `System.Xaml.dll` и `PresentationFramework.dll`.

Поместите в начальный файл C# приведенный ниже код, который создает окно с минимальной функциональностью (здесь указаны только те пространства имен, которые должны быть импортированы, чтобы код скомпилировался; любые автоматически включенные операторы `using` можно оставить на месте):

```
// Простое приложение WPF, написанное без XAML.
using System;
using System.Windows;
using System.Windows.Controls;

namespace WpfAppAllCode
{
    // В этом первом примере определяется один класс для
    // представления самого приложения и главного окна.
    class Program : Application
    {
        [STAThread]
        static void Main(string[] args)
        {
            // Обработать события Startup и Exit и затем запустить приложение.
            Program app = new Program();
            app.Startup += AppStartUp;
            app.Exit += AppExit;
            app.Run(); // Инициализирует событие Startup.
        }

        static void AppExit(object sender, ExitEventArgs e)
        {
            MessageBox.Show("App has exited");
        }

        static void AppStartUp(object sender, StartupEventArgs e)
        {
            // Создать объект Window и установить некоторые базовые свойства.
            Window mainWindow = new Window();
            mainWindow.Title = "My First WPF App!";
            mainWindow.Height = 200;
            mainWindow.Width = 300;
            mainWindow.WindowStartupLocation = WindowStartupLocation.CenterScreen;
            mainWindow.Show();
        }
    }
}
```

На заметку! Метод `Main()` в приложении WPF должен быть снабжен атрибутом `[STAThread]`, который обеспечивает безопасность к потокам любых унаследованных COM-объектов, используемых приложением. Если не аннотировать `Main()` подобным образом, возникнет исключение времени выполнения.

Обратите внимание, что класс `Program` расширяет класс `System.Windows.Application`. Внутри метода `Main()` создается экземпляр объекта приложения, затем обрабатываются события `Startup` и `Exit` с помощью синтаксиса группового преобразования методов. Вспомните из главы 10, что эта сокращенная нотация устраняет необходимость ручного указания делегатов, используемых определенным событием. Разумеется, при желании можно указывать лежащие в основе делегаты прямо по имени.

В следующем модифицированном методе `Main()` обратите внимание, что событие `Startup` работает в сочетании с делегатом `StartupEventHandler`, который может указывать только на методы, принимающие `Object` в качестве первого параметра и `StartupEventArgs` — в качестве второго. Событие `Exit`, с другой стороны, работает с делегатом `ExitEventHandler`, который требует, чтобы указанный им метод принимал тип `ExitEventArgs` во втором параметре.

```
[STAThread]
static void Main(string[] args)
{
    // На этот раз указать лежащие в основе делегаты.
    Program app = new Program();
    app.Startup += new StartupEventHandler(AppStartup);
    app.Exit += new ExitEventHandler(AppExit);
    app.Run(); // Инициализирует событие Startup.
}
```

В любом случае метод `AppStartup()` сконфигурирован для создания объекта `Window`, выполнения некоторых установок базовых свойств и вызова метода `Show()` для отображения окна на экране в немодальном режиме (с помощью метода `ShowDialog()` можно открыть модальное диалоговое окно). Метод `AppExit()` просто использует класс `MessageBox` из WPF для отображения диагностического сообщения при завершении приложения.

После компиляции и запуска проекта вы обнаружите очень простое главное окно, которое можно свернуть, развернуть и закрыть. Чтобы немного украсить его, понадобится добавить некоторые элементы пользовательского интерфейса. Но прежде чем сделать это, давайте переделаем код с использованием строго типизированного и хорошо инкапсулированного класса, производного от `Window`.

Создание строго типизированного окна

Сейчас производный от `Application` класс при запуске приложения напрямую создает экземпляр типа `Window`. В идеале нужно было бы создать класс, унаследованный от `Window`, чтобы инкапсулировать его внешний вид и функциональность. Предположим, что создано следующее определение класса в текущем пространстве имен `WpfAppAllCode` (если этот класс помещен в новый файл кода C#, в нем необходимо импортировать пространство имен `System.Windows`):

```
class MainWindow : Window
{
    public MainWindow(string windowTitle, int height, int width)
    {
        this.Title = windowTitle;
        this.WindowStartupLocation = WindowStartupLocation.CenterScreen;
        this.Height = height;
        this.Width = width;
    }
}
```

Теперь можно модифицировать обработчик событий `Startup` для прямого создания экземпляра `MainWindow`:

```
static void AppStartup(object sender, StartupEventArgs e)
{
    // Создать объект MainWindow.
    MainWindow wnd = new MainWindow("My better WPF App!", 200, 300);
    wnd.Show();
}
```

После компиляции и запуска программы получается вывод, идентичный предыдущей версии. Очевидное преимущество состоит в том, что теперь имеется строго типизированный класс, представляющий главное окно, с применением которого можно строить пользовательский интерфейс.

На заметку! Когда создается объект `Window` (или производный от `Window`), он автоматически добавляется к внутренней коллекции окон класса `Application` (посредством некоторой логики конструктора, реализованной в самом классе `Window`). С помощью свойства `Application.Windows` можно пройти по списку объектов `Window`, находящихся в данный момент в памяти.

Создание простого пользовательского интерфейса

Добавление элемента пользовательского интерфейса (наподобие `Button`) к `Window` в коде C# включает выполнение перечисленных ниже базовых шагов.

1. Определить переменную-член для представления нужного элемента управления.
2. Настроить внешний вид и поведение элемента управления при конструировании объекта `Window`.
3. Присвоить элемент управления унаследованному свойству `Content` или в качестве альтернативы передать его как параметр унаследованному методу `AddChild()`.

Вспомните, что модель содержимого элемента управления WPF требует, чтобы в свойстве `Content` указывался одиночный элемент. Естественно, окно, которое содержит только один элемент управления, не особенно полезно. Поэтому почти в каждом случае "единственной порцией содержимого", которая присваивается свойству `Content`, на самом деле является диспетчер компоновки, такой как `DockPanel`, `Grid`, `Canvas` или `StackPanel`. Внутри диспетчера компоновки можно иметь любую комбинацию внутренних элементов, в том числе другие вложенные диспетчеры компоновки (более подробно этот аспект разработки WPF описан в главе 28).

Пока что добавим в производный от `Window` класс один объект типа `Button`. В результате щелчка на этой кнопке текущее окно будет закрываться, что косвенно завершит приложение, поскольку других окон в памяти не имеется. Взгляните на следующую модификацию класса `MainWindow` (не забудьте импортировать `System.Windows.Controls` для получения доступа к классу `Button`):

```
class MainWindow : Window
{
    // Наш элемент пользовательского интерфейса.
    private Button btnExitApp = new Button();
    public MainWindow(string windowTitle, int height, int width)
    {
        // Сконфигурировать кнопку и установить дочерний элемент управления.
        btnExitApp.Click += new RoutedEventHandler(btnExitApp_Clicked);
        btnExitApp.Content = "Exit Application";
        btnExitApp.Height = 25;
        btnExitApp.Width = 100;
    }
}
```

```

// Установить в качестве содержимого окна единственную кнопку.
this.AddChild(btnExitApp);

// Сконфигурировать окно.
this.Title = windowTitle;
this.WindowStartupLocation = WindowStartupLocation.CenterScreen;
this.Height = height;
this.Width = width;
this.Show();
}

private void btnExitApp_Clicked(object sender, RoutedEventArgs e)
{
    // Закрыть окно.
    this.Close();
}
}

```

Обратите внимание, что событие Click кнопки WPF работает в сочетании с делегатом по имени `RoutedEventHandler`. Это вызывает очевидный вопрос: что собой представляет маршрутизируемое событие? Модели событий WPF подробно рассматриваются в следующей главе, а пока просто учтите, что цели делегата `RoutedEventHandler` должны принимать `Object` в качестве первого параметра и `RoutedEventArgs` — в качестве второго.

После компиляции и запуска приложения отображается измененное окно, показанное на рис. 27.6. Кнопка автоматически позиционирована по центру клиентской области окна, что является стандартным поведением, когда содержимое не помещено в тип панели WPF.

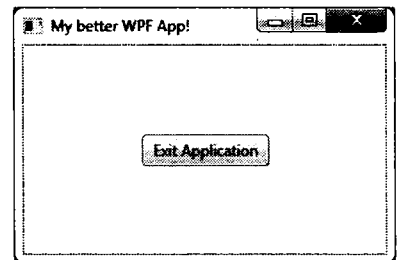


Рис. 27.6. Простое WPF-приложение, реализованное полностью в коде C#

Взаимодействие с данными уровня приложения

Вспомните, что в классе `Application` имеется свойство по имени `Properties`, которое позволяет определить коллекцию пар “имя/значение” через индексатор типа. Поскольку этот индексатор определен для оперирования над типом `System.Object`, в коллекции можно сохранять элементы любого рода (включая экземпляры пользовательских классов) с целью последующего извлечения по дружественному имени. Используя этот подход, можно очень просто разделить данные среди всех окон в приложении WPF.

В целях иллюстрации модифицируем текущий обработчик событий, чтобы он проверял входящие параметры командной строки на присутствие значения `/GODMODE` (распространенный мошеннический код во многих играх). Если эта лексема найдена, внутри коллекции свойств значение `bool` под именем `GodMode` устанавливается в `true` (а иначе в `false`).

Звучит достаточно просто, тем не менее, может возникнуть один вопрос: как передать обработчику события `Startup` входные аргументы командной строки (обычно получаемые методом `Main()`)? Один из подходов предусматривает вызов статического метода `Environment.GetCommandLineArgs()`. Однако те же самые аргументы автоматически добавляются во входной параметр `StartupEventArgs` и доступны через свойство `Args`. С учетом этого, ниже показано первое обновление текущей кодовой базы:

```
static void AppStartup(object sender, StartupEventArgs e)
{
    // Проверить входные аргументы командной строки
    // на предмет наличия флага /GODMODE.
    Application.Current.Properties["GodMode"] = false;
    foreach(string arg in e.Args)
    {
        if (arg.ToLower() == "/godmode")
        {
            Application.Current.Properties["GodMode"] = true;
            break;
        }
    }
    // Создать объект MainWindow.
    MainWindow wnd = new MainWindow("My better WPF App!", 200, 300);
}
```

Данные уровня приложения доступны повсеместно внутри WPF-приложения. Все, что нужно для этого сделать — получить точку доступа к глобальному объекту приложения (через `Application.Current`) и исследовать коллекцию. Например, обработчик событий `Click` для `Button` можно было бы изменить следующим образом:

```
private void btnExitApp_Clicked(object sender, RoutedEventArgs e)
{
    // Включил ли пользователь /godmode?
    if((bool)Application.Current.Properties["GodMode"])
        MessageBox.Show("Cheater!");

    this.Close();
}
```

Если теперь конечный пользователь запустит программу так, как показано ниже:

```
WpfAppAllCode.exe /godmode
```

то получит окно сообщения, отображаемое по завершению приложения.

На заметку! Вспомните, что аргументы командной строки можно указывать и в Visual Studio. Для этого дважды щелкните на значке `Properties` (Свойства) в окне `Solution Explorer`, перейдите на вкладку `Debug` (Отладка) и введите `/godmode` в поле `Command line arguments` (Аргументы командной строки).

Обработка закрытия объекта Window

Конечные пользователи могут завершить работу окна, применяя для этого многочисленные встроенные средства уровня системы (например, щелкнув на кнопке закрытия `X` на рамке окна) или непосредственно вызвав метод `Close()` в ответ на некоторое действие пользователя с интерактивным элементом (например, выбор пункта меню `File⇒pExit` (Файл⇒pВыход)). В любом случае WPF предлагает два события, которые можно перехватить для определения того, действительно ли пользователь намерен остановить работу окна и удалить его из памяти. Первое такое событие — это `Closing`, которое работает в сочетании с делегатом `CancelEventHandler`.

Этот делегат ожидает целевой метод, принимающий `System.ComponentModel.CancelEventArgs` во втором параметре. Класс `CancelEventArgs` предоставляет свойство `Cancel`, которое, будучи установленным в `true`, предотвратит действительное закрытие окна (это удобно, когда необходимо спросить пользователя, действительно ли он хочет закрыть окно или, возможно, сначала сохранить свою работу).

Если пользователь действительно желает закрыть окно, то `CancelEventArgs.Cancel` можно установить в `false` (это стандартное значение). Это затем приведет к выдаче события `Closed` (работающего с делегатом `System.EventHandler`), представляющего собой точку, в которой окно полностью и безвозвратно готово к закрытию.

Давайте модифицируем класс `MainWindow` для обработки этих двух событий, добавив следующие операторы кода к текущему конструктору:

```
public MainWindow(string windowTitle, int height, int width)
{
    ...
    this.Closing += MainWindow_Closing;
    this.Closed += MainWindow_Closed;
}
```

Теперь реализуем обработчики событий, как показано ниже:

```
private void MainWindow_Closing(object sender,
    System.ComponentModel.CancelEventArgs e)
{
    // Проверить, действительно ли пользователь желает закрыть окно.
    string msg = "Do you want to close without saving?";
    MessageBoxResult result = MessageBox.Show(msg,
        "My App", MessageBoxButton.YesNo, MessageBoxImage.Warning);
    if (result == MessageBoxResult.No)
    {
        // Если пользователю не желает, отменить закрытие.
        e.Cancel = true;
    }
}

private void MainWindow_Closed(object sender, EventArgs e)
{
    MessageBox.Show("See ya!");
}
```

Запустите программу и попробуйте закрыть окно, щелкнув либо на значке X в правом верхнем углу окна, либо на кнопке. Должно открыться показанное на рис. 27.7 диалоговое окно с запросом подтверждения.

Щелчок на кнопке Yes (Да) приведет к завершению приложения, а щелчок на кнопке No (Нет) оставит окно в памяти.

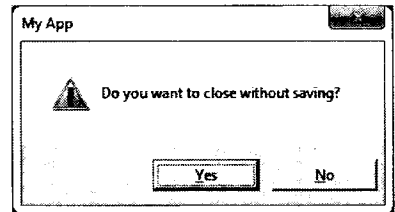


Рис. 27.7. Перехват события `Closing` объекта `Window`

Перехват событий мыши

API-интерфейс WPF предоставляет множество событий, которые можно перехватывать для организации взаимодействия с мышью. В частности, в базовом классе `UIElement` определены такие события мыши, как `MouseMove`, `MouseUp`, `MouseDown`, `MouseEnter`, `MouseLeave` и т.д.

Для примера обработаем событие `MouseMove`. Это событие работает в сочетании с делегатом `System.Windows.Input.MouseEventHandler`, который ожидает, что его целевая функция будет принимать тип `System.Windows.Input.MouseEventArgs` во втором параметре. Используя `MouseEventArgs`, можно извлечь координаты позиции (*x*, *y*) курсора мыши и прочие связанные с ним детали. Рассмотрим следующее частичное определение:


```

public class MouseEventArgs : InputEventArgs
{
    ...
    public Point GetPosition(IInputElement relativeTo);
    public MouseButtonState LeftButton { get; }
    public MouseButtonState MiddleButton { get; }
    public MouseDevice MouseDevice { get; }
    public MouseButtonState RightButton { get; }
    public StylusDevice StylusDevice { get; }
    public MouseButtonState XButton1 { get; }
    public MouseButtonState XButton2 { get; }
}

```

На заметку! Свойства XButton1 и XButton2 позволяют взаимодействовать с “расширенными кнопками мыши” (такими как кнопки “вперед” и “назад”, которые имеются в некоторых устройствах). Они часто используются для взаимодействия с хронологией навигации в браузере для переходов между посещенными страницами.

Метод `GetPosition()` позволяет получать значение (x, y) относительно элемента пользовательского интерфейса в окне. Если вы заинтересованы в захвате позиции относительно активного окна, просто передайте `this`. В конструкторе класса `MainWindow` для события `MouseMove` можно добавить обработчик:

```

public MainWindow(string windowTitle, int height, int width)
{
    ...
    this.MouseMove += MainWindow_MouseMove;
}

```

Ниже приведен обработчик события `MouseMove`, который отобразит местоположение мыши в области заголовка окна (обратите внимание, что возвращенный тип `Point` транслируется в строковое значение с помощью `ToString()`):

```

protected void MainWindow_MouseMove(object sender,
    System.Windows.Input.MouseEventArgs e)
{
    // Установить в заголовке окна текущие координаты (x, y) мыши.
    this.Title = e.GetPosition(this).ToString();
}

```

Перехват клавиатурных событий

Обработка клавиатурного ввода для окна, на котором находится фокус, также очень проста. `UIElement` определяет ряд событий, которые можно перехватывать для отслеживания нажатий клавиш клавиатуры на активном элементе (например, `KeyUp`, `KeyDown`). События `KeyUp` и `KeyDown` работают с делегатом `System.Windows.Input.KeyEventHandler`, который ожидает второго параметра типа `KeyEventArgs`, определяющего несколько важных открытых свойств:

```

public class KeyEventArgs : KeyboardEventArgs
{
    ...
    public bool IsDown { get; }
    public bool IsRepeat { get; }
    public bool IsToggled { get; }
    public bool IsUp { get; }
    public Key Key { get; }
    public KeyStates KeyStates { get; }
    public Key SystemKey { get; }
}

```

Чтобы проиллюстрировать обработку события `KeyDown` в конструкторе `MainWindow` (как это сделано для предыдущих событий), реализуем обработчик события, который изменяет содержимое кнопки на информацию о текущей нажатой клавише:

```
private void MainWindow_KeyDown(object sender, System.Windows.Input.KeyEventArgs e)
{
    // Отобразить на кнопке нажатую клавишу.
    btnExitApp.Content = e.Key.ToString();
}
```

В качестве последнего штриха дважды щелкните на значке `Properties` (Свойства) в окне `Solution Explorer` и на вкладке `Application` (Приложение) установите для `Output Type` (Тип вывода) вариант `Windows Application` (Windows-приложение). На рис. 27.8 показан конечный результат работы первой WPF-программы.

К этому моменту WPF может показаться просто еще одной платформой для построения графических пользовательских интерфейсов, которая обеспечивает (более или менее) те же самые службы, что и `Windows Forms`, `MFC` и `VB6`. Заодно возникает вопрос: для чего нужен еще один инструментальный набор для создания пользовательских интерфейсов? Чтобы оценить уникальность WPF, потребуется освоить основанную на XML грамматику — XAML.

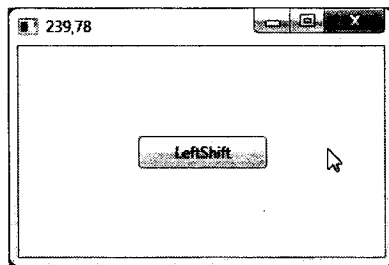


Рис. 27.8. Первая WPF-программа, написанная без использования XAML

Исходный код. Проект `WpfAppAllCode` доступен в подкаталоге `Chapter 27`.

Построение приложения WPF с использованием только XAML

Типичное WPF-приложение не состоит исключительно из кода, как это было в первом примере. Вместо этого файлы кода `C#` дополняются связанным исходным файлом `XAML`, и вместе они представляют сущность конкретного объекта `Window` или `Application`, а также других классов, которые пока не рассматривались, наподобие `UserControl` и `Page`.

Это называется *подходом файла кода* к построению WPF-приложения, и именно он будет интенсивно использоваться при рассмотрении WPF в остальной части книги. Однако прежде чем двигаться дальше, в следующем примере мы рассмотрим построение WPF-приложения с применением только файлов `XAML`. Хотя этот подход использовать не рекомендуется, он поможет лучше понять, каким образом блоки разметки `XAML` трансформируются в кодовую базу `C#` и в конечном итоге — в сборку `.NET`.

На заметку! В следующем примере используются приемы `XAML`, которые пока еще не рассматривались, однако не переживайте по поводу того, что какой-нибудь синтаксис выглядит незнакомым. Можете просто загрузить файлы решения в текстовый редактор и просмотреть код строку за строкой; однако не используйте для этого среду `Visual Studio`! Некоторая часть представленной здесь разметки `XAML` не может быть отображена в визуальных конструкторах `Visual Studio`.

В общем случае файлы `XAML` будут содержать разметку, описывающую внешний вид и поведение окна, а связанные с ними файлы кода `C#` — логику реализации. Например, файл `XAML` для объекта `Window` может описывать общую систему разметки, элементы управления внутри системы разметки, а также имена различных обработчиков собы-

тий. Связанный файл C# должен содержать логику реализации этих обработчиков событий и любой специальный код, необходимый приложению.

Расширяемый язык разметки приложений (Extensible Application Markup Language — XAML) — это основанная на XML грамматика, позволяющая определять состояние (и до некоторой степени функциональность) дерева объектов .NET через разметку. Хотя XAML часто применяется при построении пользовательских интерфейсов с помощью WPF, на самом деле его можно использовать для описания любого дерева *неабстрактных* типов .NET (включая разработанные вами специальные типы, определенные в отдельной сборке .NET) при условии, что каждый из них имеет стандартный конструктор. Как вы вскоре убедитесь, разметка, находящаяся в файле *.xaml, трансформируется в полноценную объектную модель.

Поскольку грамматика XAML основана на XML, мы получаем вместе с ним все преимущества и недостатки XML. Положительная сторона XAML заключается в том, что этому языку присущ самоописательный характер (как любому документу XML). По большому счету, каждый элемент в файле XAML представляет имя типа (такое как Button, Window или Application) в рамках заданного пространства имен .NET. Атрибуты в пределах контекста открывающего элемента отображаются на свойства (Height, Width, и т.п.) и события (Startup, Click и т.д.) указанного типа.

Учитывая тот факт, что XAML является просто декларативным способом определения состояния объекта, виджет WPF можно определить через разметку либо в процедурном коде. Например, следующий код XAML:

```
<!-- Определение WPF-элемента Button в XAML -->
<Button Name = "btnClickMe" Height = "40" Width = "100" Content = "Click Me" />
```

может быть представлен программно так:

```
// Определение того же WPF-элемента Button в коде C#.
Button btnClickMe = new Button();
btnClickMe.Height = 40;
btnClickMe.Width = 100;
btnClickMe.Content = "Click Me";
```

Отрицательным моментом является то, что XAML может быть довольно многословным и (как любой документ XML) зависимым от регистра, а потому сложные определения XAML влекут собой немалую работу с разметкой. Большинству разработчиков не приходится вручную создавать полные XAML-описания WPF-приложений. Большая часть задачи возлагается на инструменты разработки, такие как Visual Studio, Microsoft Expression Blend или другие продукты от независимых поставщиков. Как только инструмент сгенерировал базовую разметку XAML, при необходимости можно провести ее тонкую настройку вручную.

Определение объекта Window в XAML

Хотя инструменты могут генерировать вполне приемлемый код XAML, все же важно понимать основы синтаксиса XAML и то, как разметка в конечном итоге трансформируется в корректную сборку .NET. Чтобы проиллюстрировать XAML в действии, в следующем примере мы построим полноценное приложение WPF всего лишь из пары файлов *.xaml.

Первый производный от Window класс (MainWindow) был определен в C# как тип класса, расширяющего базовый класс System.Windows.Window. Этот класс содержит единственный объект Button, который вызывает зарегистрированный обработчик события по щелчку на нем. Определение того же типа Window с применением грамматики XAML может выглядеть следующим образом. Воспользуйтесь простым текстовым редактором (вроде Блокнота) для создания нового файла по имени MainWindow.xaml, сохранив его в

легко доступном подкаталоге на диске C:, поскольку этот файл будет обрабатываться в командной строке. Поместите в этот файл приведенную ниже разметку XAML:

```
<!-- Определение класса Window -->
<Window x:Class="WpfAppAllXaml.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="A Window built using 100% XAML"
  Height="200" Width="300"
  WindowStartupLocation="CenterScreen">
  <!-- Это окно имеет в качестве содержимого единственную кнопку -->
  <Button x:Name="btnExitApp" Width="133" Height="24"
    Content = "Close Window" Click ="btnExitApp_Clicked"/>
  <!-- Реализация обработчика события Click кнопки -->
  <x:Code>
    <![CDATA[
      private void btnExitApp_Clicked(object sender, RoutedEventArgs e)
      {
        this.Close();
      }
    ]]>
  </x:Code>
</Window>
```

Первым делом, обратите внимание, что корневой элемент `<Window>` использует атрибут `Class` для указания имени класса, который будет сгенерирован при обработке этого файла XAML. Кроме того, атрибут `Class` снабжен префиксом `x:`. Заглянув в открывающий элемент `<Window>`, вы увидите, что этому префиксу дескриптора XML присваивается строка `"http://schemas.microsoft.com/winfx/2006/xaml"` для построения объявления пространства имен XML. Детали этого определения пространства имен XML станут ясными чуть позже в этой главе, а пока просто имейте в виду, что всякий раз, когда хотите сослаться на элемент, определенный в пространстве имен XAML под названием `http://schemas.microsoft.com/winfx/2006/xaml`, вы должны указывать префикс — лексему `x:`.

В контексте открывающего дескриптора `<Window>` заданы значения для атрибутов `Title`, `Height`, `Width` и `WindowStartupLocation`, которые напрямую отображаются на одноименные свойства, поддерживаемые типом `System.Windows.Window` из сборки `PresentationFramework.dll`.

Далее обратите внимание, что в контексте определения окна находится разметка, описывающая внешний вид и поведение экземпляра `Button`, который будет использован для неявной установки свойства `Content` окна. Помимо установки имени переменной (с применением `x:Name`) и его общих размеров, мы также обрабатываем событие `Click` типа `Button`, присвоив метод делегату, вызываемому при возникновении события `Click`.

Финальным аспектом файла XAML является элемент `<x:Code>`, который позволяет определять обработчики событий и прочие методы этого класса непосредственно внутри файла *.xaml. В качестве меры безопасности сам код помещен в контекст `CDATA`, чтобы предотвратить попытки анализатора XML напрямую интерпретировать данные (хотя в текущем примере это не обязательно).

Важно отметить, что использовать специальную функциональность внутри элемента `<x:Code>` не рекомендуется. Хотя подход на основе единственного файла изолирует все действия в одном месте, все же встроенный код не обеспечивает четкого разделения ответственности между разметкой пользовательского интерфейса и программной логикой. В большинстве приложений WPF код реализации находится в связанном файле C# (как и мы поступим в конечном итоге).

Определение объекта Application в XAML

Вспомните, что XAML может применяться для определения разметки любого неабстрактного класса .NET, поддерживающего стандартный конструктор. Исходя из этого, в разметке можно также определить объект приложения. Рассмотрим следующее содержимое нового файла MyApp.xaml:

```
<!-- Похоже, отсутствует метод Main()!
      Однако атрибут StartupUri является
      его функциональным эквивалентом -->
<Application x:Class="WpfAppAllXaml.MyApp"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  StartupUri="MainWindow.xaml">
</Application>
```

Как видите, здесь отображение между классом-наследником Application и его описанием XAML не столь очевидно, как в случае с XAML-определением для MainWindow. В частности, не видно никаких следов метода Main(). Учитывая, что каждая исполняемая программа .NET должна иметь точку входа, вы не ошибетесь, если предположите, что она будет сгенерирована во время компиляции на основе части свойства StartupUri. Значение, присвоенное StartupUri, представляет ресурс XAML, предназначенный для загрузки при запуске приложения. В этом примере мы указали для свойства StartupUri имя ресурса XAML, определяющего наш начальный объект Window, т.е. MainWindow.xaml.

Хотя метод Main() автоматически создается во время компиляции, при желании можно воспользоваться элементом <x:Code> и определить другие блоки кода C#. Например, чтобы отобразить сообщение при завершении программы, необходимо реализовать обработчик события Exit, как показано ниже (обратите внимание, что теперь в открывающем элементе <Application> устанавливается атрибут Exit для захвата события Exit класса Application):

```
<Application x:Class="SimpleXamlApp.MyApp"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  StartupUri="MainWindow.xaml" Exit="AppExit">
<x:Code>
  <![CDATA[
    private void AppExit(object sender, ExitEventArgs e)
    {
      MessageBox.Show("App has exited");
    }
  ]]>
</x:Code>
</Application>
```

Обработка файлов XAML с помощью msbuild.exe

К этому моменту все готово к трансформации разметки в корректную сборку .NET. Однако использовать для этого напрямую компилятор C# не получится. На данный момент компилятор C# не имеет встроенной возможности распознавания разметки XAML. Тем не менее, утилита командной строки msbuild.exe знает, как трансформировать XAML в код C#, и компилирует этот код на лету, когда она информирована о правильных целевых файлах *.targets.

Утилита msbuild.exe — это инструмент, который скомпилирует код .NET на основе инструкций, содержащихся в основанном на XML сценарии сборки. В свою очередь,

этот сценарий сборки содержит в точности те же самые данные, что находятся в файле *.csproj, сгенерированном Visual Studio. Следовательно, можно компилировать программу .NET в командной строке с помощью msbuild.exe или же с применением самой среды Visual Studio.

На заметку! Полное описание утилиты msbuild.exe выходит за рамки настоящей главы. Искерпывающие сведения о ней можно найти в разделе, посвященном MSBuild, документации .NET Framework 4.5 SDK.

Ниже приведен очень простой сценарий WpfAppAllXaml.csproj, содержащий достаточно информации для объяснения msbuild.exe, каким образом трансформировать файлы XAML в соответствующую кодовую базу C#:

```
<Project DefaultTargets="Build"
  xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <RootNamespace>WpfAppAllXaml</RootNamespace>
    <AssemblyName>WpfAppAllXaml</AssemblyName>
    <OutputType>winexe</OutputType>
  </PropertyGroup>
  <ItemGroup>
    <Reference Include="System" />
    <Reference Include="WindowsBase" />
    <Reference Include="PresentationCore" />
    <Reference Include="PresentationFramework" />
  </ItemGroup>
  <ItemGroup>
    <ApplicationDefinition Include="MyApp.xaml" />
    <Page Include="MainWindow.xaml" />
  </ItemGroup>
  <Import Project="$(MSBuildBinPath)\Microsoft.CSharp.targets" />
  <Import Project="$(MSBuildBinPath)\Microsoft.WinFX.targets" />
</Project>
```

На заметку! Этот файл *.csproj не может быть загружен непосредственно в Visual Studio, поскольку он содержит только минимальный набор инструкций, необходимых для построения приложения в командной строке.

Элемент <PropertyGroup> используется для указания некоторых базовых аспектов сборки, таких как корневое пространство имен, имя результирующей сборки и тип вывода (эквивалент опции /target:winexe компилятора csc.exe).

Первый элемент <ItemGroup> задает набор внешних сборок для ссылки из текущей сборки, которыми, как видно, являются основные сборки WPF, упомянутые ранее в этой главе.

Второй элемент <ItemGroup> более интересен. Обратите внимание, что атрибут Include элемента <ApplicationDefinition> присвоен файл *.xaml, определяющий объект приложения. Атрибут Include элемента <Page> может применяться для перечисления всех остальных файлов *.xaml, в которых определены окна (и страницы, что часто происходит при построении браузерных приложений XAML), обрабатываемые объектом Application.

Однако “магия” этого сценария сборки кроется в финальных элементах <Import>. Здесь производится ссылка на два файла *.targets, каждый из которых содержит множество других инструкций, используемых во время процесса сборки.

В файле `Microsoft.WinFX.targets` определены необходимые настройки для трансформации определений XAML в эквивалентные файлы кода C#, а в файле `Microsoft.CSharp.targets` содержатся данные для взаимодействия с самим компилятором C#.

Теперь можно открыть окно командной строки разработчика и обработать данные XAML с помощью `msbuild.exe`. Для этого перейдите в каталог, в котором находятся файлы `MainWindow.xaml`, `MyApp.xaml` и `WpfAppAllXaml.csproj`, и введите следующую команду:

```
msbuild WpfAppAllXaml.csproj
```

После завершения процесса сборки в рабочем каталоге обнаружится подкаталог `\bin\obj` (точно как в проекте Visual Studio). В папке `\bin\Debug` находится новая сборка .NET по имени `WpfAppAllXaml.exe`. Открыв эту сборку в `ildasm.exe`, вы увидите, что разметка XAML была трансформирована в исполняемое приложение (рис. 27.9).

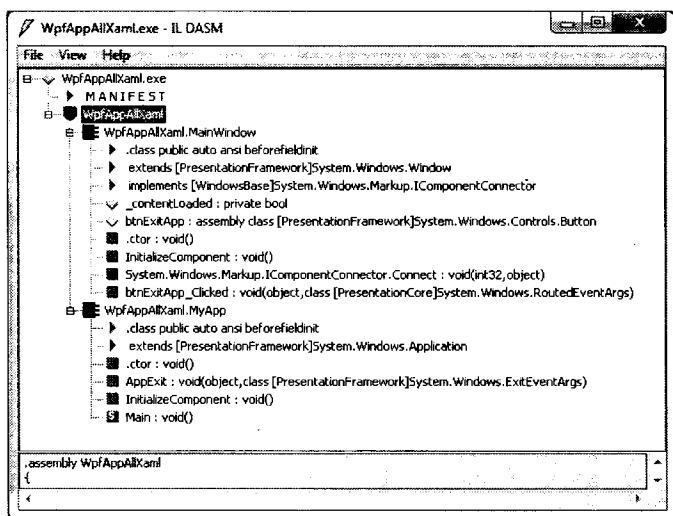


Рис. 27.9. Трансформация разметки XAML в исполняемую сборку .NET

Запустив программу двойным щелчком на исполняемом файле, вы увидите на экране ее главное окно.

Трансформация разметки в сборку .NET

Чтобы полностью понять, каким образом разметка была трансформирована в сборку .NET, нужно немного углубиться в процесс `msbuild.exe` и изучить ряд сгенерированных компилятором файлов, включая отдельный двоичный ресурс, встроенный в сборку во время компиляции. Первая задача заключается в изучении того, как файлы `*.xaml` трансформируются в соответствующую кодовую базу C#.

Отображение XAML-разметки окна на код C#

Файлы `*.targets`, указанные в сценарии для `msbuild.exe`, содержат множество инструкций трансляции элементов XAML в код C#. Когда `msbuild.exe` обрабатывает файл `*.csproj`, создаются два файла `*.g.cs` (где `g` означает *generated* (автоматически сгенерированные)), которые сохраняются в папке `\obj\Debug`. На основе имен файлов `*.xaml` полученные файлы C# получают названия `MainWindow.g.cs` и `MyApp.g.cs`.

Открыв файл `MainWindow.g.cs` в текстовом редакторе, вы найдете там класс по имени `MainWindow`, расширяющий базовый класс `Window`. Имя этого класса — прямой результат действия атрибута `x:Class` открывающего дескриптора `<Window>`. Кроме того, в этом классе определена переменная-член типа `System.Windows.Controls.Button` с именем `btnExitApp`. В данном случае имя элемента управления основано на значении атрибута `x:Name` открывающего объявления `<Button>`. Этот класс также содержит обработчик события `Click` кнопки — `btnExitApp_Clicked()`. Ниже приведена часть листинга этого сгенерированного компилятором файла `MainWindow.g.cs`:

```
public partial class MainWindow :
    System.Windows.Window, System.Windows.Markup.IComponentConnector
{
    internal System.Windows.Controls.Button btnExitApp;
    private void btnExitApp_Clicked(object sender, RoutedEventArgs e)
    {
        this.Close();
    }
    ...
}
```

В классе определена закрытая переменная-член типа `bool` (по имени `named_contentLoaded`), которая не была напрямую представлена в разметке XAML. Этот член данных используется для того, чтобы гарантировать присваивание содержимого окна только один раз:

```
public partial class MainWindow :
    System.Windows.Window, System.Windows.Markup.IComponentConnector
{
    // Назначение этой переменной-члена поясняется ниже.
    private bool _contentLoaded;
    ...
}
```

Обратите внимание, что сгенерированный компилятором класс также явно реализует WPF-интерфейс `IComponentConnector`, определенный в пространстве имен `System.Windows.Markup`. В этом интерфейсе имеется единственный метод `Connect()`, который реализован для подготовки каждого элемента управления, определенного в разметке, и обеспечения логики событий, как указано в исходном файле `MainWindow.xaml`. Перед завершением этого метода переменная-член `_contentLoaded` устанавливается в `true`. Вот как выглядит этот метод:

```
void System.Windows.Markup.IComponentConnector.Connect(int connectionId,
    object target)
{
    switch (connectionId)
    {
        case 1:
            this.btnExitApp = ((System.Windows.Controls.Button) (target));
            this.btnExitApp.Click += new
                System.Windows.RoutedEventHandler(this.btnExitApp_Clicked);
            return;
    }
    this._contentLoaded = true;
}
```

И, наконец, в классе `MainWindow` также реализован метод `InitializeComponent()`. Можно было бы ожидать, что этот метод содержит код, устанавливающий внешний вид и поведение каждого элемента управления за счет присваивания различных свойств

(Height, Width, Content и т.п.). Однако это не так! Как же тогда эти элементы управления получают корректный пользовательский интерфейс? Логика `InitializeComponent()` определяет местоположение встроенного в сборку ресурса, имя которого совпадает с именем исходного файла *.xaml:

```
public void InitializeComponent()
{
    if (_contentLoaded)
    {
        return;
    }
    _contentLoaded = true;
    System.Uri resourceLocater = new
        System.Uri("/WpfAppAllXaml;component/mainwindow.xaml",
            System.UriKind.Relative);
    System.Windows.Application.LoadComponent(this, resourceLocater);
}
```

И здесь возникает вопрос: что такое встроенный ресурс?

Роль BAML

Когда утилита `msbuild.exe` обрабатывает файл *.csproj, она генерирует файл с расширением *.baml, именованный согласно начальному файлу `MainWindow.xaml`. Следовательно, в папке `\obj\Debug` должен появиться файл под названием `MainWindow.baml` (рис. 27.10).

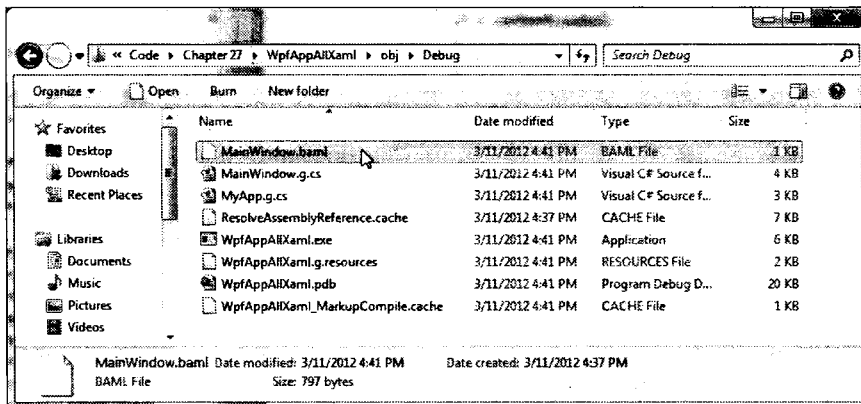


Рис. 27.10. Формат BAML — это просто компактная двоичная версия XAML

Как и можно было предположить, формат BAML (Binary Application Markup Language — двоичный язык разметки приложений) — это компактное двоичное представление исходных данных XAML. Этот файл *.baml встраивается в виде ресурса (через сгенерированный файл *.g.resources) в скомпилированную сборку.

Ресурс BAML содержит все необходимые данные для настройки внешнего вида виджетов пользовательского интерфейса (свойства, подобные Height и Width). Открыв файл *.baml в Visual Studio, можно увидеть следы начальных атрибутов XAML (рис. 27.11).

Здесь важно понять, что WPF-приложение содержит внутри себя двоичное представление (BAML) разметки. Во время выполнения ресурс BAML будет извлечен из контейнера ресурсов и использован для настройки внешнего вида всех окон и элементов управления.

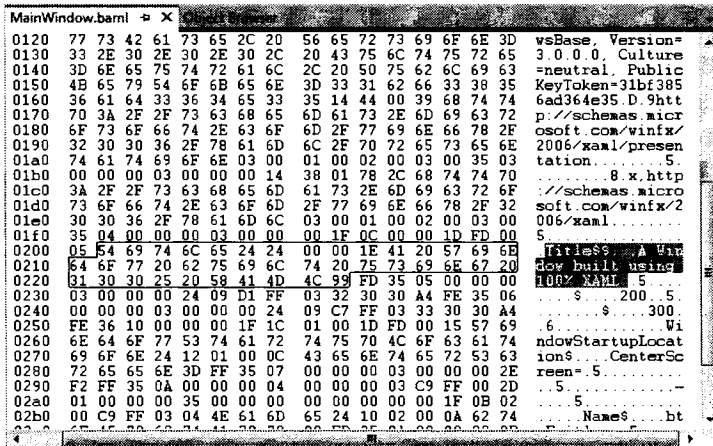


Рис. 27.11. Файл BAML содержит значения свойств, используемые для конструирования объектов во время выполнения

Кроме того, запомните, что имена этих двоичных ресурсов *идентичны* именам написанных автономных файлов *.xaml. Однако это вовсе не означает необходимость поставки файлов *.xaml вместе со скомпилированной программой WPF. Если только не строится WPF-приложение, которое должно динамически загружать и разбирать файлы *.xaml во время выполнения, поставлять исходную разметку никогда не придется.

Отображение XAML-разметки приложения на код C#

Последняя часть автоматически сгенерированного кода, которую мы рассмотрим, находится в файле MyApp.g.cs. Здесь имеется производный от Application класс с соответствующей точкой входа — методом Main(). Реализация Main() вызывает метод InitializeComponent() на типе-наследнике Application, который, в свою очередь, устанавливает свойство StartupUri, позволяя каждому объекту делать корректные установки свойств на основе двоичного представления XAML.

```
namespace WpfAppAllXaml
{
    public partial class MyApp : System.Windows.Application
    {
        void AppExit(object sender, ExitEventArgs e)
        {
            MessageBox.Show("App has exited");
        }
        [System.Diagnostics.DebuggerNonUserCodeAttribute()]
        public void InitializeComponent()
        {
            this.Exit += new System.Windows.ExitEventHandler(this.AppExit);
            this.StartupUri = new System.Uri("MainWindow.xaml", System.UriKind.Relative);
        }
        [System.STAThreadAttribute()]
        [System.Diagnostics.DebuggerNonUserCodeAttribute()]
        public static void Main() {
            SimpleXamlApp.MyApp app = new SimpleXamlApp.MyApp();
            app.InitializeComponent();
            app.Run();
        }
    }
}
```

Итоговые замечания о процессе трансформирования XAML в сборку

Итак, к этому моменту получена полноценная программа WPF с использованием всего двух файлов XAML и связанного сценария сборки для `msbuild.exe`. Как было показано, для обработки файлов XAML (и генерации `*.baml`) утилита `msbuild.exe` в процессе сборки полагается на вспомогательные настройки, определенные внутри файла `*.targets`. На рис. 27.12 показана общая картина, касающаяся обработки файлов `*.xaml` во время компиляции.

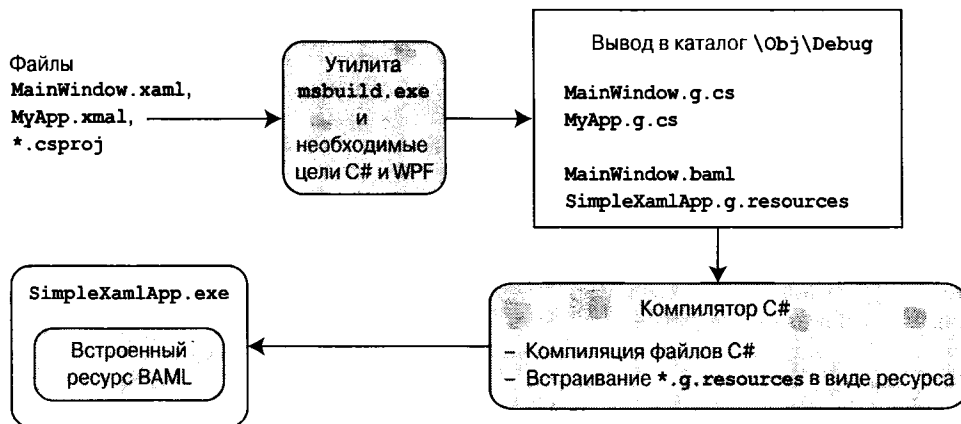


Рис. 27.12. Процесс трансформации XAML в сборку во время компиляции

Теперь вы лучше представляете, как используются данные XAML для построения приложения .NET. Далее можно переходить к рассмотрению синтаксиса и семантики самого языка XAML.

Исходный код. Проект `WpfAppAllXaml` доступен в подкаталоге Chapter 27.

Синтаксис XAML для WPF

Приложения WPF производственного уровня обычно будут использовать специальные инструменты для генерации необходимой XAML-разметки. Как бы ни были хороши эти инструменты, все же понимание общей структуры XAML не помешает. Для оказания помощи в процессе изучения доступен очень популярный (и бесплатный) инструмент, который позволяет легко экспериментировать с XAML.

Введение в Kaхaml

Если вы впервые приступаете к изучению грамматики XAML, то очень удобно пользоваться бесплатным инструментом под названием *Kaхaml*. Этот популярный редактор/анализатор XAML доступен для загрузки на веб-сайте <http://www.kaxaml.com>.

Редактор *Kaхaml* хорош тем, что он не имеет никакого понятия об исходном коде C#, обработчиках ошибок или логике реализации, и предлагает намного более простой способ тестирования фрагментов XAML, чем применение полноценного шаблона проекта WPF в Visual Studio. К тому же *Kaхaml* обладает набором интегрированных инструментов, таких как средство выбора цвета и диспетчер фрагментов XAML, и даже опцию

“очистки XAML”, которая форматирует XAML-разметку на основе заданных настроек. Открыв Kaхaml в первый раз, вы найдете там простую разметку для элемента управления <Page>:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>

  </Grid>
</Page>
```

Подобно Window, элемент Page содержит различные диспетчеры компоновки и элементы управления. Однако, в отличие от Window, объекты Page не могут выполняться как отдельные сущности. Вместо этого они должны помещаться внутри подходящего хоста, такого как NavigationWindow, Frame или веб-браузер (и в этом случае просто делается приложение ХВАР). Очень удобно то, что можно вводить идентичную разметку в области <Page> или <Window>.

На заметку! Если в окне разметки Kaхaml заменить элементы <Page> и </Page> на <Window> и </Window>, то можно нажать клавишу <F5> для отображения нового окна на экране.

Для выполнения простого теста введите следующую разметку в панели XAML, расположенной внизу окна Kaхaml:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>
    <!-- Кнопка со специальным содержимым -->
    <Button Height="100" Width="100">
      <Ellipse Fill="Green" Height="50" Width="50"/>
    </Button>
  </Grid>
</Page>
```

В верхней части окна Kaхaml появится визуализированная страница (рис. 27.13).

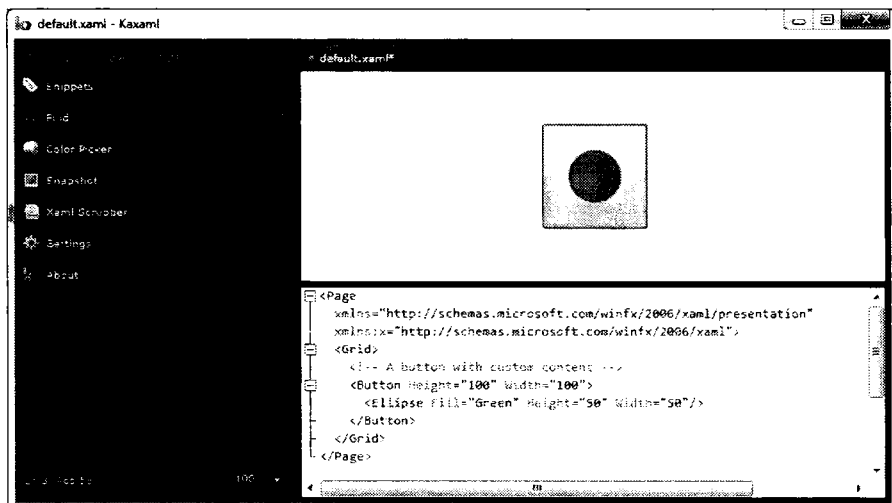


Рис. 27.13. Редактор Kaхaml — очень удобный (и бесплатный) инструмент, применяемый для изучения грамматики XAML

При работе в `Кахамл` помните, что этот инструмент не позволяет писать разметку, которая влечет за собой какую-либо компиляцию кода (однако разрешено использовать `x:Name`). Сюда входит определение атрибута `x:Class` (для указания файла кода), ввод имен обработчиков событий в разметке или применение любых ключевых слов XAML, которые также вызывают компиляцию кода (вроде `FieldModifier` или `ClassModifier`). Попытка сделать это приведет к ошибке разметки.

Пространства имен XAML XML и “ключевые слова” XAML

Корневой элемент XAML-документа WPF (такой как `<Window>`, `<Page>`, `<UserControl>` или `<Application>`) почти всегда ссылается на следующие два заранее определенных пространства имен XML:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>

  </Grid>
</Page>
```

Первое пространство имен XML, `http://schemas.microsoft.com/winfx/2006/xaml/presentation`, отображает множество связанных с WPF пространств имен .NET для использования в текущем файле *.xaml (`System.Windows`, `System.Windows.Controls`, `System.Windows.Data`, `System.Windows.Ink`, `System.Windows.Media`, `System.Windows.Navigation` и т.д.).

Это отображение “один ко многим” на самом деле жестко закодировано внутри сборки WPF (`WindowsBase.dll`, `PresentationCore.dll` и `PresentationFramework.dll`) с применением атрибута `[XmlnsDefinition]` уровня сборки. Например, если открыть браузер объектов Visual Studio и выбрать сборку `PresentationCore.dll`, можно увидеть списки, подобные показанному ниже, который импортирует пространство имен `System.Windows`:

```
[assembly: XmlnsDefinition(
  "http://schemas.microsoft.com/winfx/2006/xaml/presentation",
  "System.Windows")]
```

Второе пространство имен XML, `http://schemas.microsoft.com/winfx/2006/xaml`, используется для включения специфичных для XAML “ключевых слов” (этот термин выбран за неимением лучшего) вместе с пространством имен `System.Windows.Markup`:

```
[assembly: XmlnsDefinition("http://schemas.microsoft.com/winfx/2006/xaml",
  "System.Windows.Markup")]
```

Одно из правил любого корректно сформированного XML-документа (вспомните, что XAML — это грамматика, основанная на XML) состоит в том, что он должен иметь открывающий корневой элемент, назначающий одно пространство имен XML в качестве *первичного пространства имен*, которым обычно является пространство, содержащее наиболее часто используемые элементы. Если корневой элемент требует включения дополнительных вторичных пространств имен (как здесь показано), они должны быть определены с уникальным префиксом (чтобы разрешить возможные конфликты имен). В качестве префикса принято применять просто `x`, тем не менее, это может быть любой уникальный маркер по вашему выбору, например, `XamlSpecificStuff`:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:XamlSpecificStuff="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>
```

```

<!-- Кнопка со специальным содержанием -->
<Button XamlSpecificStuff:Name="button1" Height="100" Width="100">
  <Ellipse Fill="Green" Height="50" Width="50"/>
</Button>
</Grid>
</Page>

```

Очевидный недостаток определения длинных префиксов для пространств имен XML связан с тем, что `XamlSpecificStuff` придется набирать всякий раз, когда в файле XAML нужно сослаться на один из типов, определенных в этом XML-пространстве XAML. Поскольку набирать каждый раз префикс `XamlSpecificStuff` слишком утомительно, давайте ограничимся `x`.

В любом случае, помимо ключевых слов `x:Name`, `x:Class` и `x:Code`, пространство имен XML под названием `http://schemas.microsoft.com/winfx/2006/xaml` также предоставляет доступ к дополнительным ключевым словам XAML, наиболее часто используемые из которых перечислены в табл. 27.9.

Таблица 27.9. Ключевые слова XAML

Ключевое слово XAML	Описание
<code>x:Array</code>	Представляет в XAML тип массива .NET
<code>x:ClassModifier</code>	Позволяет определять видимость типа класса (<code>internal</code> или <code>public</code>), обозначенного ключевым словом <code>Class</code>
<code>x:FieldModifier</code>	Позволяет определять видимость члена типа (<code>internal</code> , <code>public</code> , <code>private</code> или <code>protected</code>) для любого именованного элемента корня (например, <code><Button></code> внутри элемента <code><Window></code>). Именованный элемент определяется с использованием ключевого слова <code>Name</code> в XAML
<code>x:Key</code>	Позволяет устанавливать значение ключа для элемента XAML, которое должно быть помещено в элемент словаря
<code>x:Name</code>	Позволяет указывать сгенерированное C# имя заданного элемента XAML
<code>x:Null</code>	Представляет ссылку <code>null</code>
<code>x:Static</code>	Позволяет ссылаться на статический член типа
<code>x:Type</code>	XAML-эквивалент операции <code>typeof</code> языка C# (она будет выдавать <code>System.Type</code> на основе указанного имени)
<code>x:TypeArguments</code>	Позволяет устанавливать элемент как обобщенный тип с определенным параметром типа (например, <code>List<int></code> или <code>List<bool></code>)

В дополнение к этим двум необходимым объявлениям пространств имен XML можно, а иногда и нужно, определить дополнительные префиксы дескрипторов в открывающем элементе XAML-документа. Обычно это делается, когда требуется описать в XAML класс .NET, определенный во внешней сборке.

Например, предположим, что вы построили несколько специальных элементов управления WPF и упаковали их в библиотеку под названием `MyControls.dll`. Теперь, если необходимо создать новый объект `Window`, который использует эти элементы, можно установить специальное пространство имен XML, отображаемое на библиотеку `MyControls.dll`, с применением лексем `clr-namespace` и `assembly`. Ниже приведен пример разметки, создающей префикс дескриптора по имени `myCtrls`, который может использоваться для доступа к элементам управления в этой библиотеке:

```

<Window x:Class="WpfApplication1.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

```

```

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:myCtrls="clr-namespace:MyControls;assembly=MyControls"
Title="MainWindow" Height="350" Width="525">
<Grid>
    <myCtrls:MyCustomControl />
</Grid>
</Window>

```

Лексеме `clr-namespace` назначается название пространства имен .NET в сборке, в то время как для лексемы `assembly` указывается дружественное имя внешней сборки *.dll. Такой синтаксис можно использовать для любой внешней библиотеки .NET, которой необходимо манипулировать внутри разметки. В настоящее время в этом нет необходимости, но в последующих главах потребуется определять специальные объявления пространства имен XML для описания типов в разметке.

На заметку! Если нужно определить в разметке класс, который является частью текущей сборки, но находится в другом пространстве имен .NET, префикс дескриптора `xmlns` должен быть определен без атрибута `assembly`. Например:

```
xmlns:myCtrls="clr-namespace:SomeNamespaceInMyApp"
```

Управление видимостью классов и переменных-членов

Многие из этих ключевых полей вы увидите в действии там, где они понадобятся в последующих главах. Давайте в качестве простого примера рассмотрим следующее XAML-определение `<Window>`, в котором используются ключевые слова `ClassModifier` и `FieldModifier`, а также `x:Name` и `x:Class` (вспомните, что редактор `Кахамл` не позволяет применять ключевые слова XAML, вовлекающие компиляцию, такие как `x:Code`, `x:FieldModifier` или `x:ClassModifier`):

```

<!-- Этот класс теперь будет объявлен как internal в файле *.g.cs -->
<Window x:Class="MyWPFApp.MainWindow" x:ClassModifier="internal"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

    <!-- Эта кнопка будет объявлена как public в файле *.g.cs -->
    <Button x:Name="myButton" x:FieldModifier="public" Content="OK"/>
</Window>

```

По умолчанию все определения типов C#/XAML являются открытыми (`public`), а члены — внутренними (`internal`). Однако на основе показанного определения XAML результирующий автоматически сгенерированный файл содержит внутренний тип класса с открытой переменной-членом `Button`:

```

internal partial class MainWindow : System.Windows.Window,
    System.Windows.Markup.IComponentConnector
{
    public System.Windows.Controls.Button myButton;
    ...
}

```

Элементы XAML, атрибуты XAML и преобразователи типов

После установки корневого элемента и необходимых пространств имен XML следующей задачей будет наполнение корня дочерним элементом. Как уже упоминалось, в реальных WPF-приложениях дочерним элементом будет диспетчер компоновки (наподобие `Grid` или `StackPanel`), который, в свою очередь, содержит любое количество элементов, определяющих пользовательский интерфейс. Эти диспетчеры компоновки под-

робно рассматриваются в следующей главе, а пока предположим, что элемент `<Window>` будет содержать единственный элемент `Button`.

Как было показано ранее в главе, *элементы XAML* отображаются на типы классов или структур внутри заданного пространства имен .NET, в то время как *атрибуты* в открывающем дескрипторе элемента отображаются на свойства и события конкретного типа. В целях иллюстрации введите следующее определение `<Button>` в редакторе *Кахамл*:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>
    <!-- Настроить внешний вид элемента Button -->
    <Button Height=»50» Width=»100» Content=»OK!»
      FontSize=»20» Background=»Green» Foreground=»Yellow»/>
  </Grid>
</Page>
```

Обратите внимание, что значения, присвоенные свойствам, являются строковыми. Это может показаться полным несоответствием типам данных, поскольку после создания такого элемента `Button` в коде C# этим свойствам будут присваиваться не строковые объекты, а значения специфических типов данных. Например, ниже показано, как та же самая кнопка описана в коде:

```
public void MakeAButton()
{
    Button myBtn = new Button();
    myBtn.Height = 50;
    myBtn.Width = 100;
    myBtn.FontSize = 20;
    myBtn.Content = "OK!";
    myBtn.Background = new SolidColorBrush(Colors.Green);
    myBtn.Foreground = new SolidColorBrush(Colors.Yellow);
}
```

Оказывается, в состав инфраструктуры WPF входит множество классов *преобразователей типов*, которые применяются для трансформации простых текстовых значений в корректные типы данных. Этот процесс происходит прозрачно (и автоматически).

Тем не менее, нередко возникает потребность в присваивании атрибуту XAML намного более сложного значения, которое не может быть выражено простой строкой. Например, предположим, что необходимо построить специальную кисть, которая будет устанавливаться в свойстве `Background` элемента `Button`. При создании этой кисти в коде все достаточно просто:

```
public void MakeAButton()
{
    ...
    // Забавная кисть для фона.
    LinearGradientBrush fancyBrush =
        new LinearGradientBrush(Colors.DarkGreen, Colors.LightGreen, 45);
    myBtn.Background = fancyBrush;
    myBtn.Foreground = new SolidColorBrush(Colors.Yellow);
}
```

Но можно ли представить эту сложную кисть в виде строки? Нет, нельзя! К счастью, в XAML предусмотрен специальный синтаксис, который можно применять всякий раз, когда нужно присвоить сложный объект в качестве значения свойства, и это *синтаксис "свойство-элемент"*.

Понятие синтаксиса “свойство-элемент” в XAML

Синтаксис свойство-элемент позволяет присваивать свойству сложные объекты. Вот как выглядит XAML-описание элемента `Button`, свойство `Background` которого установлено в `LinearGradientBrush`:

```
<Button Height="50" Width="100" Content="OK!"
        FontSize="20" Foreground="Yellow">
  <Button.Background>
    <LinearGradientBrush>
      <GradientStop Color="DarkGreen" Offset="0"/>
      <GradientStop Color="LightGreen" Offset="1"/>
    </LinearGradientBrush>
  </Button.Background>
</Button>
```

Обратите внимание, что внутри контекста дескрипторов `<Button>` и `</Button>` определен вложенный контекст по имени `<Button.Background>`, а в его рамках — специальный элемент `<LinearGradientBrush>`. (Пока не беспокойтесь о коде кисти; графика WPF будет рассматриваться в главе 29.)

В общем случае любое свойство может быть установлено с использованием синтаксиса “свойство-элемент”, который всегда сводится к следующему шаблону:

```
<ОпределяющийКласс>
  <ОпределяющийКласс.СвойствоОпределяющегоКласса>
    <!-- Значение для СвойствоОпределяющегоКласса -->
  </ОпределяющийКласс.СвойствоОпределяющегоКласса>
</ОпределяющийКласс >
```

Хотя любое свойство *может* быть установлено с использованием этого синтаксиса, указание значения в виде простой строки экономит время на ввод. Например, ниже показано, как мог бы выглядеть более громоздкий способ установки свойства `Width` элемента `Button`:

```
<Button Height="50" Content="OK!"
        FontSize="20" Foreground="Yellow">
  ...
  <Button.Width>
    100
  </Button.Width>
</Button>
```

Понятие присоединяемых свойств XAML

В дополнение к синтаксису “свойство-элемент” в XAML поддерживается специальный синтаксис, используемый для установки значения *присоединяемого свойства*. В сущности, присоединяемое свойство позволяет дочернему элементу устанавливать значение свойства, которое в действительности определено в родительском элементе. Общий шаблон, которому нужно следовать, выглядит так:

```
<РодительскийЭлемент>
  <ДочернийЭлемент РодительскийЭлемент.СвойствоРодительскогоЭлемента = "Значение">
</РодительскийЭлемент>
```

Наиболее распространенное применение синтаксиса присоединяемых свойств связано с позиционированием элементов пользовательского интерфейса внутри одного из классов диспетчеров компоновки (`Grid`, `DockPanel` и т.п.). Эти диспетчеры компоновки более подробно рассматриваются в следующей главе, а пока введите в редакторе `KaXaml` следующую разметку:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Canvas Height="200" Width="200" Background="LightBlue">
    <Ellipse Canvas.Top="40" Canvas.Left="40" Height="20" Width="20" Fill="DarkBlue"/>
  </Canvas>
</Page>
```

Здесь определен диспетчер компоновки Canvas, содержащий в себе элемент Ellipse. Обратите внимание, что с помощью синтаксиса присоединяемого свойства элемент Ellipse может информировать свой родительский элемент (Canvas) о том, как внутри него позиционировать левый верхний угол Ellipse.

В отношении присоединяемых свойств следует запомнить несколько моментов. Прежде всего, это не универсальный синтаксис, который может применяться к *любому* свойству *любого* родительского элемента. Например, следующая разметка XAML содержит ошибку:

```
<!-- Ошибка! Нельзя устанавливать свойство Background в Canvas через
присоединяемое свойство -->
<Canvas Height="200" Width="200">
  <Ellipse Canvas.Background="LightBlue"
    Canvas.Top="40" Canvas.Left="90"
    Height="20" Width="20" Fill="DarkBlue"/>
</Canvas>
```

В действительности присоединяемые свойства представляют собой специализированную форму связанной с WPF концепции, которая называется *свойством зависимости*. Если свойство не было реализовано в очень специфической манере, его значение не может быть установлено с использованием синтаксиса присоединяемых свойств. Более подробно свойства зависимости рассматриваются в главе 31.

На заметку! Инструменты Kaхaml, Visual Studio и Expression Blend оснащены средством IntelliSense, отображающим действительные присоединяемые свойства, которые могут быть установлены для каждого элемента.

Понятие расширений разметки XAML

Как уже объяснялось, значения свойств чаще всего представляются в виде простой строки или через синтаксис “свойство-элемент”. Однако существует и другой способ указать значение атрибута XAML — с использованием *расширений разметки*. Расширения разметки позволяют анализатору XAML получать значение свойства из выделенного внешнего класса. Это может обеспечить большие преимущества, учитывая, что некоторые свойства требуют выполнения множества операторов кода для получения значений.

Расширения разметки предлагают способ ясного расширения грамматики XAML новой функциональностью. Расширение разметки внутренне представлено как класс, производный от MarkupExtension. Следует подчеркнуть: шансы, что когда-либо придется строить специальное расширение разметки, невелики. Тем не менее, подмножество ключевых слов XAML (таких как x:Array, x:Null, x:Static и x:Type) представляют собой расширения разметки.

Расширение разметки заключается в фигурные скобки, как показано ниже:

```
<Элемент УстанавливаемоеСвойство = "{РасширениеРазметки}"/>
```

Чтобы увидеть расширение разметки в действии, введите следующий код в редакторе Kaхaml:

```

<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:CorLib="clr-namespace:System;assembly=mscorlib">
  <StackPanel>
    <!-- Расширение разметки Static позволяет получать значение
         статического члена класса -->
    <Label Content="{x:Static CorLib:Environment.OSVersion}"/>
    <Label Content="{x:Static CorLib:Environment.ProcessorCount}"/>

    <!-- Расширение разметки Type — это XAML-версия операции typeof языка C# -->
    <Label Content="{x:Type Button}" />
    <Label Content="{x:Type CorLib:Boolean}" />

    <!-- Наполнение элемента ListBox массивом строк -->
    <ListBox Width="200" Height="50">
      <ListBox.ItemsSource>
        <x:Array Type="CorLib:String">
          <CorLib:String>Sun Kil Moon</CorLib:String>
          <CorLib:String>Red House Painters</CorLib:String>
          <CorLib:String>Besnard Lakes</CorLib:String>
        </x:Array>
      </ListBox.ItemsSource>
    </ListBox>
  </StackPanel>
</Page>

```

Прежде всего, обратите внимание, что определение `<Page>` имеет новое объявление пространства имен XML, которое позволяет получать доступ к пространству имен сборки `mscorlib.dll`. Имея это пространство имен, сначала с помощью расширения разметки `x:Static` извлекаются значения `OSVersion` и `ProcessorCount` класса `System.Environment`.

Расширение разметки `x:Type` позволяет получить доступ к описанию метаданных указанного элемента. Здесь просто назначаются полностью заданные имена типов `Button` и `System.Boolean` из WPF.

Наиболее интересная часть показанной выше разметки связана с элементом `ListBox`. Его свойство `ItemSource` устанавливается в массив строк, полностью объявленный в разметке. Обратите внимание, что расширение разметки `x:Array` позволяет указывать набор подэлементов внутри своего контекста:

```

<x:Array Type="CorLib:String">
  <CorLib:String>Sun Kil Moon</CorLib:String>
  <CorLib:String>Red House Painters</CorLib:String>
  <CorLib:String>Besnard Lakes</CorLib:String>
</x:Array>

```

На заметку! Приведенный выше пример XAML служит только для демонстрации расширения разметки в действии. Как будет показано в главе 28, существуют намного более простые способы наполнения элементов управления `ListBox`.

На рис. 27.14 показана разметка этого элемента `<Page>` в редакторе `KaXaml`.

Итак, вы ознакомились с многочисленными примерами, демонстрирующими основные аспекты синтаксиса XAML. Наверняка вы согласитесь, что XAML — очень интересный язык в том плане, что позволяет описывать дерево объектов .NET в декларативной манере. Хотя это исключительно полезно для конфигурирования графических пользовательских интерфейсов, следует помнить, что XAML может описать *любой* тип из *любой* сборки при условии, что это неабстрактный тип, имеющий стандартный конструктор.

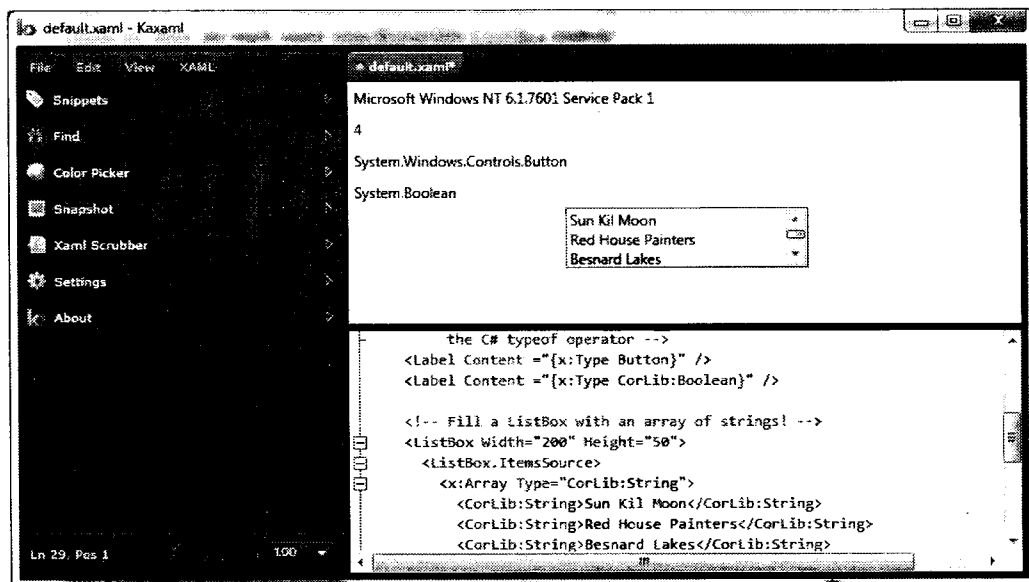


Рис. 27.14. Расширения разметки позволяют устанавливать значения через функциональность выделенного класса

Построение приложений WPF с использованием файлов отделенного кода

В первых двух примерах этой главы демонстрировались крайние случаи построения приложения WPF с использованием только кода C# или только разметки XAML. Рекомендованный способ построения любого приложения WPF предусматривает применение подхода на основе *файлов кода*. Согласно этой модели, файлы XAML проекта не содержат ничего кроме разметки, которая описывает общее состояние классов, в то время как файлы кода содержат детали реализации.

Добавление файла кода для класса MainWindow

Для иллюстрации сказанного модифицируем пример `WpfAppAllXaml`, добавив к нему файлы кода. Скопируйте всю папку предыдущего примера и назовите ее `WpfAppCodeFiles`.

Создайте в этой папке новый файл кода C# по имени `MainWindow.xaml.cs` (по существующему соглашению имя файла отделенного кода C# имеет форму `*.xaml.cs`). Поместите в этот новый файл показанный ниже код:

```
// MainWindow.xaml.cs
using System;
using System.Windows;
using System.Windows.Controls;

namespace WpfAppAllXaml
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            // Обратите внимание, что этот метод определен внутри
            // сгенерированного файла MainWindow.g.cs.
        }
    }
}
```

```

        InitializeComponent();
    }

    private void btnExitApp_Clicked(object sender, RoutedEventArgs e)
    {
        this.Close();
    }
}

```

Здесь определен частичный класс, содержащий логику обработки событий, которая может быть объединена с определением частичного класса того же самого типа в файле *.g.cs. Учитывая, что `InitializeComponent()` определен внутри файла `MainWindow.g.cs`, конструктор окна выполняет вызов для загрузки и обработки встроенного ресурса BAML.

Файл `MainWindow.xaml` также должен быть изменен; это означает просто удаление в нем всех следов прежнего кода C#:

```

<Window x:Class="WpfAppAllXaml.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="A Window built using Code Files!"
        Height="200" Width="300"
        WindowStartupLocation="CenterScreen">

    <!-- Обработчик событий теперь находится в файле кода -->
    <Button x:Name="btnExitApp" Width="133" Height="24"
            Content="Close Window" Click="btnExitApp_Clicked"/>
</Window>

```

Добавление файла кода для класса `MyApp`

При необходимости можно было бы также построить файл отделенного кода для типа, производного от `Application`. Поскольку большая часть действий происходит в файле `MyApp.g.cs`, код внутри файла `MyApp.xaml.cs` выглядит следующим образом:

```

// MyApp.xaml.cs
using System;
using System.Windows;
using System.Windows.Controls;

namespace WpfAppAllXaml
{
    public partial class MyApp : Application
    {
        private void AppExit(object sender, ExitEventArgs e)
        {
            MessageBox.Show("App has exited");
        }
    }
}

```

В файле `MyApp.xaml` теперь содержится такая разметка:

```

<Application x:Class="WpfAppAllXaml.MyApp"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        StartupUri="MainWindow.xaml"
        Exit="AppExit">
</Application>

```

Обработка файлов кода с помощью msbuild.exe

Прежде чем перекомпилировать файлы с использованием msbuild.exe, понадобится модифицировать файл *.csproj для учета новых файлов C#, подлежащих включению в процесс компиляции, с помощью элементов <Compile> (выделены полужирным):

```
<Project DefaultTargets="Build" xmlns=
"http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <RootNamespace>WpfAppAllXaml</RootNamespace>
    <AssemblyName>WpfAppAllXaml</AssemblyName>
    <OutputType>winexe</OutputType>
  </PropertyGroup>
  <ItemGroup>
    <Reference Include="System" />
    <Reference Include="WindowsBase" />
    <Reference Include="PresentationCore" />
    <Reference Include="PresentationFramework" />
  </ItemGroup>
  <ItemGroup>
    <ApplicationDefinition Include="MyApp.xaml" />
    <Compile Include = "MainWindow.xaml.cs" />
    <Compile Include = "MyApp.xaml.cs" />
    <Page Include="MainWindow.xaml" />
  </ItemGroup>
  <Import Project="$(MSBuildBinPath)\Microsoft.CSharp.targets" />
  <Import Project="$(MSBuildBinPath)\Microsoft.WinFX.targets" />
</Project>
```

После передачи сценария сборки утилите msbuild.exe:

```
msbuild WpfAppAllXaml.csproj
```

вы получите ту же исполняемую сборку, что и в приложении WpfAppAllXaml (расположенную в папке bin\Debug). Однако в том, что касается разработки, теперь имеется четкое отделение представления (XAML) от программной логики (C#).

Поскольку это — предпочтительный метод разработки WPF, в приложениях WPF, созданных с использованием Visual Studio (или Expression Blend), всегда применяется модель отделенного кода.

Исходный код. Проект WpfAppCodeFiles доступен в подкаталоге Chapter 27.

Построение приложений WPF с использованием Visual Studio

На протяжении этой главы примеры создавались с использованием простых текстовых редакторов, компилятора командной строки и редактора Kaхaml. Тогда важно было сосредоточить внимание на основном синтаксисе приложений WPF, не отвлекаясь на дополнительные украшения из графического конструктора. Однако теперь, зная, как строятся WPF-приложения с помощью простейших средств, давайте посмотрим, каким образом Visual Studio может упростить процесс разработки.

На заметку! Ниже представлены некоторые ключевые особенности использования Visual Studio для построения WPF-приложений. В последующих главах при необходимости будут иллюстрироваться дополнительные аспекты этой IDE-среды.

Шаблоны проектов WPF

В диалоговом окне New Project (Новый проект) среды Visual Studio определен набор рабочих пространств проектов WPF, и все они расположены в узле Windows корня Visual C#. На выбор доступны следующие варианты: WPF Application (Приложение WPF), WPF User Control Library (Библиотека пользовательских элементов управления WPF), WPF Custom Control Library (Библиотека специальных элементов управления WPF) и WPF Browser Application (Браузерное приложение WPF, т.е. ХВАР). Для начала создадим новое приложение WPF по имени WpfTesterApp (рис. 27.15).

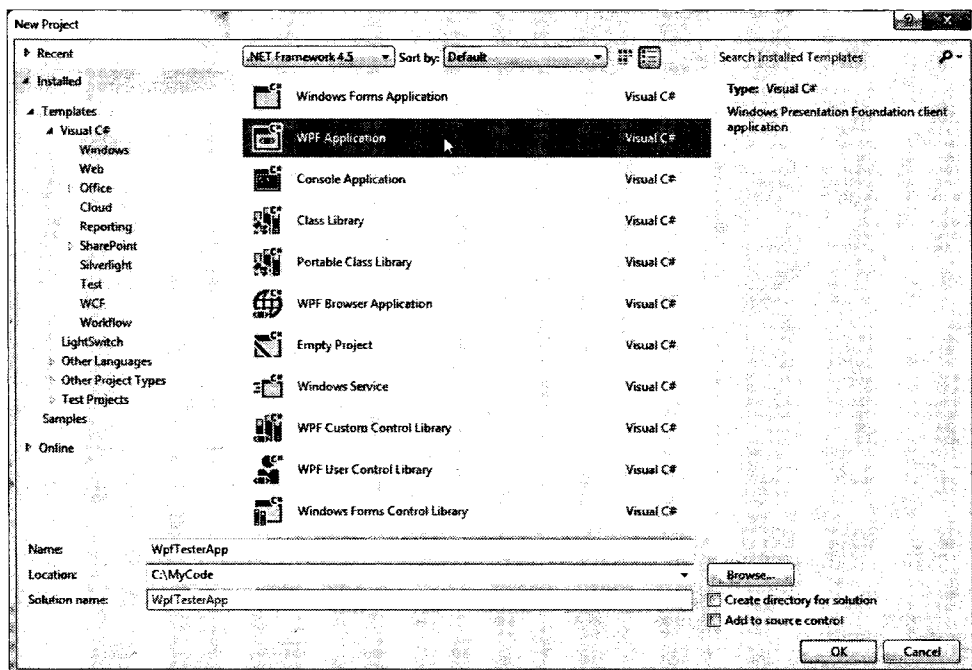


Рис. 27.15. Шаблоны проектов WPF в Visual Studio находятся в узле Windows

Помимо установки ссылок на все сборки WPF (PresentationCore.dll, PresentationFramework.dll, System.Xaml.dll и WindowsBase.dll), вы также получаете начальные классы, производные от Window и Application, каждый из которых представлен с использованием XAML и файла кода C#. На рис. 27.16 показано окно Solution Explorer для этого нового проекта WPF.

Панель инструментов и визуальный конструктор/редактор XAML

В Visual Studio предусмотрена панель инструментов (Toolbox), доступная через меню View (Вид) и содержащая множество элементов управления WPF (рис. 27.17).

Используя стандартные операции перетаскивания с помощью мыши, можно поместить любой из этих элементов управления на поверхность визуального конструктора элемента Window или перетащить элемент управления на редактор разметки XAML в нижней части визуального конструктора. При этом начальная разметка XAML будет сгенерирована автоматически. Перетащите посредством мыши элементы управления Button и Calendar на поверхность визуального конструктора. Сделав это, обратите внимание на возможность изменения позиций и размеров элементов управления (и просмотрите результирующую разметку XAML, генерируемую на основе изменений).

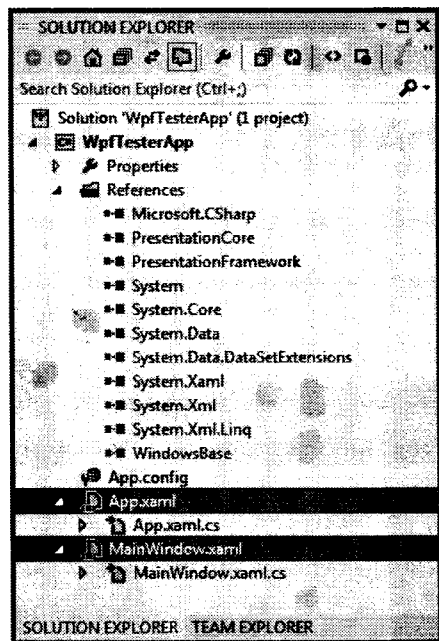


Рис. 27.16. Начальные файлы проекта типа WPF Application

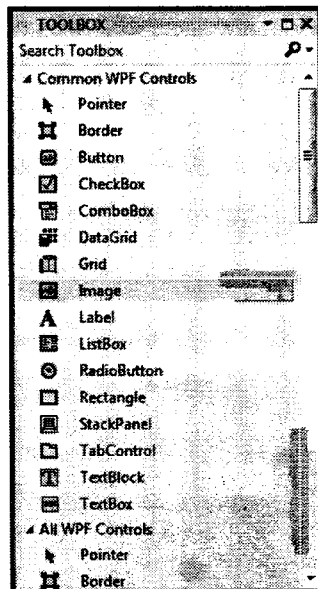


Рис. 27.17. Панель инструментов содержит элементы управления WPF, которые могут быть помещены на поверхность визуального конструктора

В дополнение к построению пользовательского интерфейса с помощью мыши и панели инструментов, разметку можно также вводить вручную, используя интегрированный редактор XAML. Как показано на рис. 27.18, при этом обеспечивается поддержка средства IntelliSense, что помогает упростить написание разметки. Для примера попробуйте добавить свойство `Background` к открывающему элементу `<Window>`.

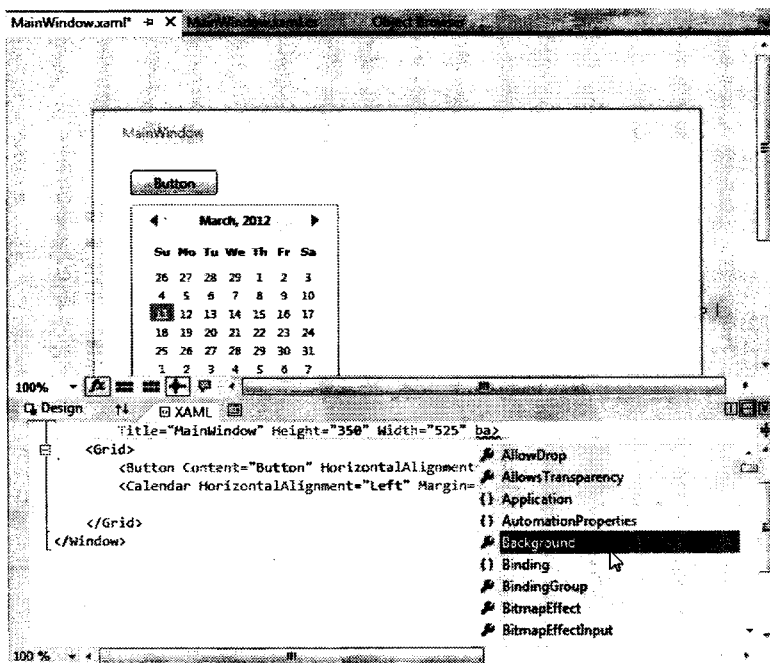


Рис. 27.18. Визуальный конструктор элемента Window

Посвятите некоторое время добавлению ряда значений свойств непосредственно в редакторе XAML. Обязательно должным образом освоите этот аспект визуального конструктора WPF.

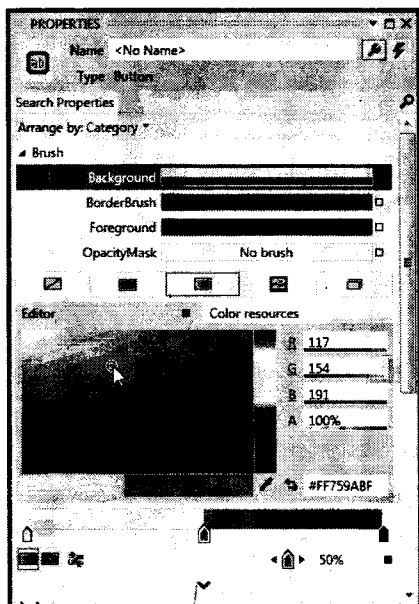


Рис. 27.19. Окно Properties позволяет конфигурировать пользовательский интерфейс элемента управления WPF

Установка свойств с использованием окна Properties

После помещения нескольких элементов управления на поверхность визуального конструктора (или определения их в редакторе вручную) можно воспользоваться окном Properties (Свойства) для установки значений свойств выбранного элемента управления, а также для создания обработчиков событий. В качестве простого примера выберите в визуальном конструкторе ранее добавленный элемент управления Button. С помощью окна Properties измените цвет в свойстве Background элемента Button, применяя встроенный редактор кистей, как показано на рис. 27.19 (редактор кистей более подробно рассматривается в главе 29, когда будет исследоваться графика WPF).

На заметку! В верхней части окна Properties имеется текстовая область, предназначенная для поиска. Введите имя свойства, которое требуется установить, чтобы быстро найти нужный элемент.

Завершив упражнение с редактором кистей, взгляните на сгенерированную разметку. Она может выглядеть примерно так:

```
<Button Content="Button" Height="23" HorizontalAlignment="Left"
Margin="12,12,0,0"
    Name="button1" VerticalAlignment="Top" Width="75">
  <Button.Background>
    <LinearGradientBrush EndPoint="1,0.5" StartPoint="0,0.5">
      <GradientStop Color="#FF7488CE" Offset="0" />
      <GradientStop Color="#FFC11E1E" Offset="0.837" />
    </LinearGradientBrush>
  </Button.Background>
</Button>
```

Обработка событий с использованием окна Properties

Для организации обработки событий, связанных с определенным элементом управления, также может использоваться окно Properties, но на этот раз понадобится щелкнуть на кнопке Events (События), расположенной в верхней правой части окна (кнопка с изображением молнии). Удостоверьтесь, что на поверхности визуального конструктора выбрана кнопка, щелкните на кнопке Events в окне Properties и найдите событие Click. Среда Visual Studio автоматически построит обработчик событий, имеющий следующую общую форму:

ИмяЭлементаУправления_ИмяСобытия

Поскольку кнопка не была переименована, в окне Properties отображается сгенерированный обработчик событий по имени `Button_Click_1` (рис. 27.20).

Кроме того, Visual Studio сгенерирует соответствующий обработчик событий C# в файле кода для окна. Здесь можно добавить любой код, который должен выполняться по щелчку на кнопке. В качестве примера добавьте следующий оператор кода:

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }

    private void Button_Click_1(
        object sender, RoutedEventArgs e)
    {
        MessageBox.Show("You clicked the button!");
    }
}
```

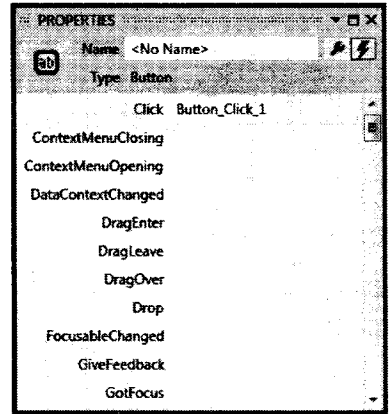


Рис. 27.20. Обработка событий с использованием окна Properties

Обработка событий в редакторе XAML

Обрабатывать события можно также непосредственно в редакторе XAML. Для примера поместите курсор мыши внутрь элемента `<Window>` и введите имя события `MouseMove`, а за ним — знак равенства. Visual Studio отобразит все совместимые обработчики из файла кода (если они существуют), а также опцию `<New Event Handler>` (Новый обработчик события), как показано на рис. 27.21.

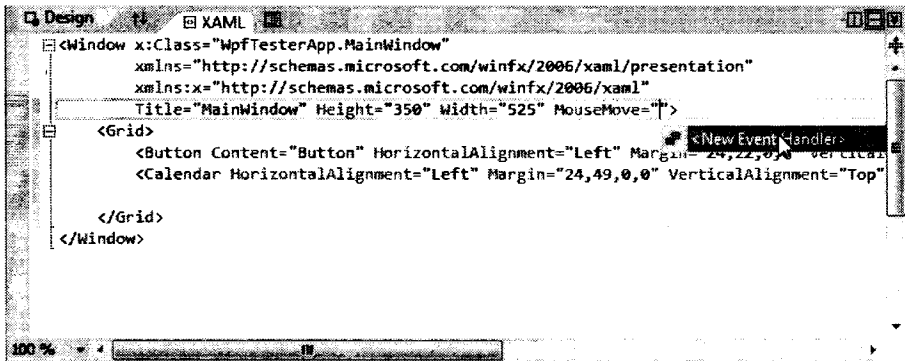


Рис. 27.21. Обработка событий с применением редактора XAML

Двойной щелчок на пункте `<New Event Handler>` приводит к тому, что IDE-среда сгенерирует соответствующий обработчик в файле кода C#. Поместите следующий код в обработчик события `MouseMove`, после чего запустите приложение, чтобы увидеть конечный результат:

```
private void Window_MouseMove (object sender, MouseEventArgs e)
{
    this.Title = e.GetPosition(this).ToString();
}
```

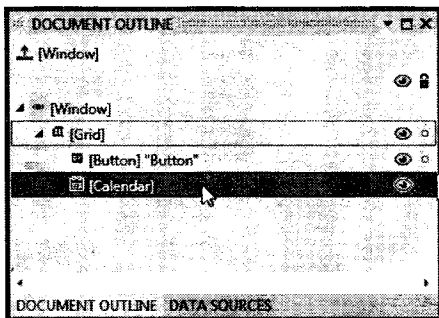


Рис. 27.22. Визуализация разметки XAML с помощью окна Document Outline

Этот инструмент также предоставляет способ временного сокрытия заданного элемента (или набора элементов) на поверхности визуального конструктора, а также блокировку элементов с целью предотвращения их редактирования. В следующей главе будет показано, что окно Document Outline предоставляет и многие другие возможности, в числе которых группирование выбранных элементов в новые диспетчеры компоновки.

Окно Document Outline

Во время работы с любым основанным на XAML проектом (приложение WPF, Silverlight, Windows Phone 7 или Windows 8) вы, безусловно, будете применять значительное количество разметки для представления пользовательских интерфейсов. Когда вы начнете сталкиваться с более сложной разметкой XAML, может оказаться удобной визуализация разметки для быстрого выбора элементов с целью редактирования в визуальном конструкторе Visual Studio.

В настоящее время наша разметка довольно проста, т.к. было определено лишь несколько эле-

Просмотр автоматически сгенерированных файлов кода

Перед построением последнего примера этой главы найдите окно Solution Explorer и щелкните на кнопке Show All Files (Показать все файлы), как показано на рис. 27.23. Обратите внимание на присутствие файлов *.baml и *.g.cs (в папке obj\Debug). Добавлять свой код в эти автоматически сгенерированные файлы никогда не придется, а предыдущие примеры главы должны были прояснить, как обрабатывается разметка XAML.

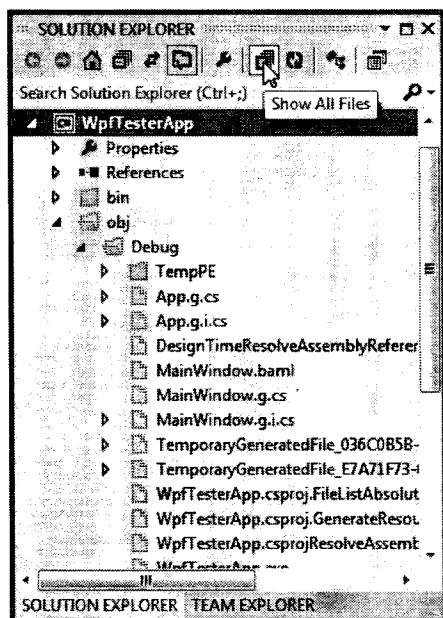


Рис. 27.23. Просмотр выходных файлов проекта WPF с использованием окна Solution Explorer

Построение специального редактора XAML с помощью Visual Studio

Теперь, когда вы ознакомились с базовыми инструментами, используемыми внутри Visual Studio для проектирования окна WPF, в последнем примере этой главы будет показано, как построить приложение, которое позволяет манипулировать

разметкой XAML во время выполнения. Закройте текущий проект и создайте новый проект типа WPF Application (WPF-приложение) по имени MyXamlPad. Этот проект по завершении будет функционировать аналогично KaXaml, но без разнообразных украшений. В частности, это приложение позволит вводить правильно сформированную разметку и щелкать на кнопке для динамического преобразования XAML в новый объект Window.

Проектирование графического пользовательского интерфейса окна

API-интерфейс WPF поддерживает возможность загрузки, разбора и сохранения XAML-описаний программным образом. Это может быть полезно во многих ситуациях. Например, предположим, что есть пять разных файлов XAML, описывающих внешний вид и поведение типа Window. До тех пор, пока имена каждого элемента (и всех необходимых обработчиков событий) идентичны внутри каждого файла, можно динамически менять "обложки" окна (возможно, на основе аргумента, передаваемого приложению при запуске).

Взаимодействие с XAML во время выполнения вращается вокруг типов XamlReader и XamlWriter — оба они определены в пространстве имен System.Windows.Markup. Чтобы проиллюстрировать, как программно наполнить объект Window из внешнего файла *.xaml, создадим приложение, которое имитирует базовую функциональность редактора KaXaml.

На заметку! Классы XamlReader и XamlWriter предоставляют базовую функциональность для манипулирования разметкой XAML во время выполнения. Если когда-нибудь понадобится получить полный контроль над объектной моделью XAML, придется изучить сборку System.Xaml.dll.

Хотя наше приложение определенно не будет столь же полнофункциональным, как KaXaml, все же оно предоставит возможность вводить допустимую разметку XAML, просматривать результаты и сохранять разметку XAML во внешнем файле. Для начала измените первоначальное XAML-определение элемента <Window>, как показано ниже (в этой точке можно набрать разметку XAML вручную; однако для генерации обработчиков событий следует воспользоваться IDE-средой, как было показано ранее).

На заметку! В следующей главе мы погрузимся в детали работы с элементами управления и панелями, так что пока не беспокойтесь об аспектах, связанных с объявлениями элементов управления.

```
<Window x:Class="MyXamlPad.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="My Custom XAML Editor" Height="338" Width="1041"
  Loaded="Window_Loaded" Closed="Window_Closed"
  WindowStartupLocation="CenterScreen">

  <!-- Используйте DockPanel, а не Grid -->
  <DockPanel LastChildFill="True" >
    <!-- Эта кнопка запустит окно с определенным XAML -->
    <Button DockPanel.Dock="Top" Name = "btnViewXaml" Width="100" Height="40"
      Content = "View Xaml" Click="btnViewXaml_Click" />
    <!-- Это будет область для ввода -->
    <TextBox AcceptsReturn = "True" Name = "txtXamlData"
      FontSize = "14" Background="Black" Foreground="Yellow"
      BorderBrush = "Blue" VerticalScrollBarVisibility="Auto"
      AcceptsTab="True"/>
  </DockPanel>
</Window>
```

Прежде всего, обратите внимание на то, что первоначальный элемент `<Grid>` заменен диспетчером компоновки `<DockPanel>`, содержащим элементы `Button` (по имени `btnViewXaml`) и `TextBox` (по имени `txtXamlData`), а также на то, как обрабатывается событие `Click` элемента `Button`. Кроме того, события `Loaded` и `Closed` самого типа `Window` были обработаны внутри открывающего элемента `<Window>` (опять-таки, для генерации обработчиков событий необходимо воспользоваться IDE-средой, как описывалось ранее). Если вы применяли визуальный конструктор для обработки событий, то должны найти следующий код в файле `MainWindow.Xaml.cs`:

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }
    private void btnViewXaml_Click(object sender, RoutedEventArgs e)
    {
    }
    private void Window_Closed(object sender, EventArgs e)
    {
    }
    private void Window_Loaded(object sender, RoutedEventArgs e)
    {
    }
}
```

Прежде чем продолжить, не забудьте импортировать следующие пространства имен в файл `MainWindow.xaml.cs`:

```
using System.IO;
using System.Windows.Markup;
```

Реализация события `Loaded`

Событие `Loaded` главного окна отвечает за определение наличия файла `YourXaml.xaml` в папке, содержащей приложение. Если этот файл существует, данные из него будут прочитаны и помещены в элемент `TextBox` главного окна. В противном случае элемент `TextBox` заполняется начальным стандартным XAML-описанием пустого окна (это описание в точности совпадает с разметкой, полученной при начальном определении окна, за исключением того, что вместо `<Grid>` используется `<StackPanel>`).

На заметку! Строку, которая была построена для представления начальной разметки, отображаемой в разрабатываемом редакторе, набирать довольно утомительно, учитывая потребность в символах отмены, которые необходимы для внутренних кавычек, поэтому будьте внимательны.

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    // При загрузке главного окна приложения поместить
    // некоторый базовый текст XAML в текстовый блок.
    if (File.Exists("YourXaml.xaml"))
    {
        txtXamlData.Text = File.ReadAllText("YourXaml.xaml");
    }
    else
    {
        txtXamlData.Text =
            "<Window xmlns=\"http://schemas.microsoft.com/winfx/2006/xaml/presentation\"\\n"
            +"xmlns:x=\"http://schemas.microsoft.com/winfx/2006/xaml\"\\n"
```

```

+ "Height = \"400\" Width = \"500\" WindowStartupLocation = \"CenterScreen\">\n"
+ "<StackPanel>\n"
+ "</StackPanel>\n"
+ "</Window>";
}
}

```

Используя этот подход, приложение сможет загружать XAML, введенный в предыдущем сеансе, или при необходимости предлагать стандартный блок разметки. В данный момент можно запустить программу и увидеть внутри объекта TextBox разметку, показанную на рис. 27.24.

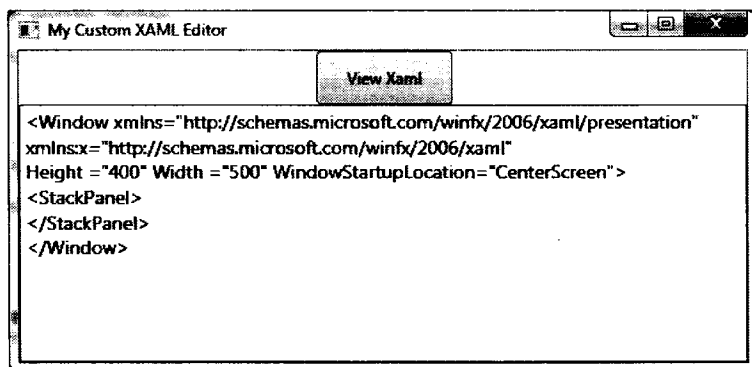


Рис. 27.24. Первый запуск MyXamlPad.exe

Реализация события Click объекта Button

При щелчке на объекте Button сначала сохраняются текущие данные из TextBox в файле YourXaml.xaml. Сохраненные данные читаются через File.Open() для получения объекта FileStream. Это необходимо, поскольку для представления XAML-разметки, подлежащей анализу, метод XamlReader.Load() требует типа, производного от Stream (вместо простого System.String).

После загрузки XAML-описания элемента <Window>, который будет конструироваться, создайте экземпляр System.Windows.Window на основе находящейся в памяти разметки XAML и отобразите Window в виде модального диалогового окна:

```

private void btnViewXaml_Click(object sender, RoutedEventArgs e)
{
    // Записать данные из текстового блока в локальный файл *.xaml.
    File.WriteAllText("YourXaml.xaml", txtXamlData.Text);
    // Это окно, к которому будет динамически применяться XAML-разметка.
    Window myWindow = null;
    // Открыть локальный файл *.xaml.
    try
    {
        using (Stream sr = File.Open("YourXaml.xaml", FileMode.Open))
        {
            // Подключить XAML-разметку к объекту Window.
            myWindow = (Window)XamlReader.Load(sr);
            // Отобразить диалоговое окно и выполнить очистку.
            myWindow.ShowDialog();
            myWindow.Close();
            myWindow = null;
        }
    }
}

```

```

    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}

```

Обратите внимание, что большая часть логики помещена в блок try/catch. Таким образом, если файл YourXaml.xaml содержит неверно сформированную разметку, в результирующем окне сообщения отобразится сообщение об ошибке. Например, запустите программу и намеренно внесите ошибку в написание элемента <StackPanel>, скажем, добавив лишнюю букву р в открывающий элемент. После щелчка на кнопке отобразится сообщение об ошибке, подобное показанному на рис. 27.25.

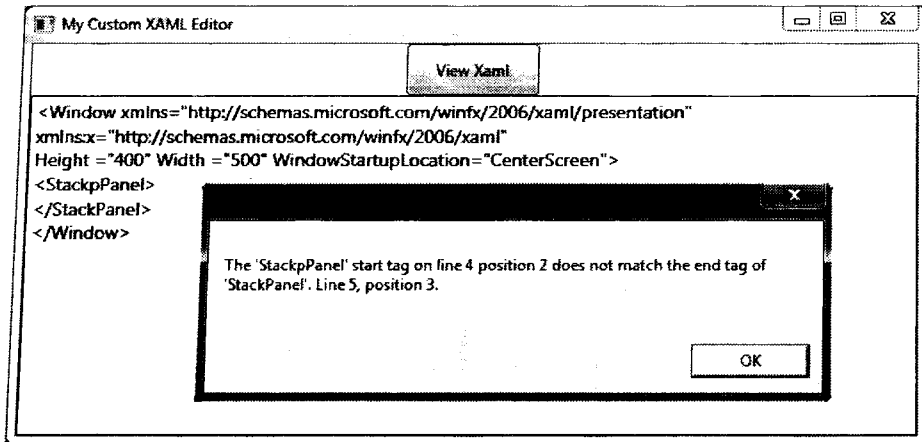


Рис. 27.25. Перехват ошибок разметки

Реализация события Closed

И, наконец, событие Closed типа Window гарантирует, что данные, введенные в TextBox, сохраняются в файле YourXaml.xaml:

```

private void Window_Closed(object sender, EventArgs e)
{
    // Записать данные из текстового поля в локальный файл *.xaml.
    File.WriteAllText("YourXaml.xaml", txtXamlData.Text);
    Application.Current.Shutdown();
}

```

Тестирование приложения

Запустите приложение и ведите некоторую XAML-разметку в текстовой области. Имейте в виду, что (подобно редактору KaXaml) это приложение не позволяет указывать атрибуты XAML, связанные с генерацией кода (такие как Class или обработчики событий). В качестве первого теста введите следующий код XAML внутри контекста <StackPanel>:

```

<Button Height = "100" Width = "100" Content = "Click Me!">
    <Button.Background>
        <LinearGradientBrush StartPoint = "0,0" EndPoint = "1,1">
            <GradientStop Color = "Blue" Offset = "0" />
            <GradientStop Color = "Yellow" Offset = "0.25" />
            <GradientStop Color = "Green" Offset = "0.75" />
            <GradientStop Color = "Pink" Offset = "0.50" />
        </LinearGradientBrush>
    </Button.Background>
</Button>

```

После щелчка на кнопке появится окно, визуализирующее XAML-определения (или, возможно, окно с сообщением об ошибке анализа — будьте внимательны при вводе). На рис. 27.26 показан возможный вывод.

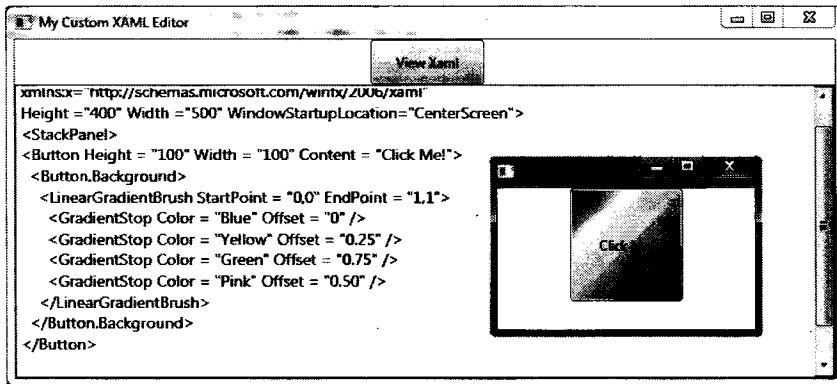


Рис. 27.26. Приложение MyXamlPad.exe в действии

Теперь поместите следующую разметку XAML непосредственно после текущего определения `<Button>`:

```

<Label Content="Interesting...">
  <Label.Triggers>
    <EventTrigger RoutedEvent="Label.Loaded">
      <EventTrigger.Actions>
        <BeginStoryboard>
          <Storyboard TargetProperty="FontSize">
            <DoubleAnimation From="12" To="100" Duration="0:0:4"
              RepeatBehavior="Forever"/>
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger.Actions>
    </EventTrigger>
  </Label.Triggers>
</Label>
  
```

Эта разметка является великолепной демонстрацией реальной мощи XAML. При тестировании данной разметки вы обнаружите, что создали простую анимационную последовательность. Службы анимации (а также графической визуализации) детально рассматриваются в последующих главах; тем не менее, можете корректировать разметку XAML и смотреть на полученные результаты.

Изучение документации WPF

В заключение этой главы следует отметить, что в документации .NET 4.5 Framework SDK тематике WPF посвящен целый раздел. По мере исследования этого API-интерфейса и чтения остальных глав, посвященных WPF, вы обнаружите настоятельную потребность в обращении к справочной системе. Там вы найдете множество примеров разметки XAML и детальные обучающие руководства по многочисленным темам, начиная с программирования трехмерной графики и заканчивая примерами сложной привязки к данным.

Документация WPF доступна через путь .NET Framework 4.5 ⇒ .NET Framework Development Guide ⇒ Developing Client Applications (.NET Framework 4.5 ⇒ Руководство по .NET Framework ⇒ Разработка клиентских приложений), как показано на рис. 27.27.

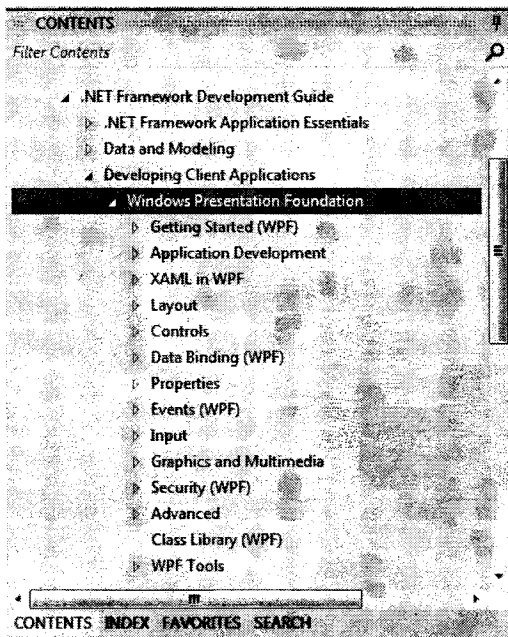


Рис. 27.27. Документация .NET 4.5 Framework SDK предлагает развернутую справку по WPF

появившийся в версии .NET 3.0. Основная цель WPF состоит в интеграции и унификации множества ранее разрозненных настольных технологий (двух- и трехмерная графика, разработка окон и элементов управления и т.п.) в единую унифицированную программную модель. Помимо этого, в WPF-приложениях обычно используется расширяемый язык разметки приложений (Extendable Application Markup Language — XAML), который позволяет определять внешний вид и поведение элементов WPF через разметку.

Вспомните, что XAML позволяет описывать деревья объектов .NET с применением декларативного синтаксиса. Во время исследования XAML в настоящей главе вы узнали несколько новых частей синтаксиса, включая синтаксис “свойство-элемент” и присоединяемые свойства, а также роль преобразователей типов и расширений разметки XAML.

Хотя XAML — ключевой аспект любого профессионального WPF-приложения, в первом примере этой главы было показано, как построить программу WPF исключительно с помощью кода C#. Затем вы узнали, как строить программу WPF, используя только XAML (что, однако, не рекомендуется; это был всего лишь учебный пример). Наконец, было продемонстрировано применение файлов отделенного кода, которые позволяют отделять поведение от функциональности.

В последнем примере этой главы строилось WPF-приложение, позволяющее программно взаимодействовать с определениями XAML с использованием классов `XamlReader` и `XamlWriter`. Кроме того, вы ознакомились с визуальными конструкторами WPF среды Visual Studio. Дополнительные сведения по этому поводу будут представлены в последующих главах.

По мере изучения этой части справочной системы вы столкнетесь с многочисленными примерами разметки XAML, которые можно копировать в буфер обмена и вставлять в построенный специальный редактор XAML. Тем не менее, перед тестированием понадобится поменять корневой элемент `<Page>` на `<Window>` (приложение было запрограммировано на отображение полноценных объектов `Window`, а не `Page`). Перед тем, как переходить к следующей главе, уделите время интересующим вас темам и протестируйте дополнительную разметку в созданном ранее специальном инструменте XAML.

Исходный код. Проект `MyXamlPad` доступен в подкаталоге `Chapter 27`.

Резюме

Windows Presentation Foundation (WPF) — это набор инструментов для построения пользовательских интерфейсов,

Программирование с использованием элементов управления WPF

В главе 27 был заложен фундамент модели программирования WPF, включая рассмотрение классов `Window` и `Application`, грамматики XAML и использования файлов кода. Вы также ознакомились с процессом построения WPF-приложений посредством визуальных конструкторов среды Visual Studio. В настоящей главе мы углубимся в конструирование более сложных графических пользовательских интерфейсов с применением нескольких новых элементов управления и диспетчеров компоновки, к тому же попутно исследуем дополнительные возможности визуальных конструкторов WPF, предлагаемых Visual Studio.

В этой главе также будут рассмотрены некоторые важные связанные с элементами управления WPF темы, такие как программная модель привязки данных и использование команд управления. Вы узнаете, как работать с интерфейсами Ink API и Documents API, которые позволяют получать ввод от пера (или мыши) и создавать RTF-документы с помощью XML Paper Specification.

На заметку! В предыдущих изданиях этой книги использовался продукт под названием Microsoft Expression Blend, который позволял упрощать построение графических пользовательских интерфейсов WPF-приложений. Тем не менее, функциональности, касающейся создания пользовательских интерфейсов WPF, которая предлагается в последней версии Visual Studio, вполне достаточно для реализации рассматриваемых в этой книге примеров. Если вы хотите изучить детали работы с Expression Blend, обратитесь к книге Эндрю Троелсена *Expression Blend 4 с примерами на C# для профессионалов* (ИД "Вильямс", 2011 г.)

Обзор основных элементов управления WPF

Если вы не являетесь новичком в области построения графических пользовательских интерфейсов, то общее назначение большинства элементов управления WPF не должно вызывать много вопросов. Независимо от того, какой набор инструментов для построения графических пользовательских интерфейсов вы применяли в прошлом (напри-

мер, VB 6.0, MFC, Java AWT/Swing, Windows Forms, Mac OS X (Cocoa) или GTK+/GTK#), основные элементы управления WPF, перечисленные в табл. 28.1, скорее всего, покажутся знакомыми.

Таблица 28.1. Основные элементы управления WPF

Категория элементов управления WPF	Примеры членов	Описание
Основные элементы управления для пользовательского ввода	Button, RadioButton, ComboBox, CheckBox, Calendar, DatePicker, Expander, DataGrid, ListBox, ListView, ToggleButton, TreeView, ContextMenu, ScrollBar, Slider, TabControl, TextBlock, TextBox, RepeatButton, RichTextBox, Label	WPF предлагает полное семейство элементов управления, которые можно применять для построения пользовательских интерфейсов
Элементы украшения окон и элементов управления	Menu, ToolBar, StatusBar, ToolTip, ProgressBar	Эти элементы пользовательского интерфейса служат для декорирования рамки объекта Window компонентами для ввода (наподобие Menu) и элементами информирования пользователя (например, StatusBar и ToolTip)
Элементы управления мультимедиа	Image, MediaElement, SoundPlayerAction	Эти элементы управления предоставляют поддержку воспроизведения аудио/видео и визуализации изображений
Элементы управления компоновкой	Border, Canvas, DockPanel, Grid, GridView, GridSplitter, GroupBox, Panel, TabControl, StackPanel, Viewbox, WrapPanel	WPF предлагает множество элементов управления, которые позволяют группировать и организовывать другие элементы для управления компоновкой

Элементы управления Ink API

В дополнение к общим элементам управления WPF, перечисленным в табл. 28.1, в WPF определены элементы управления для работы с интерфейсом Ink API. Этот аспект разработки WPF полезен при построении приложений для Tablet PC, поскольку он позволяет захватывать ввод от пера. Тем не менее, это вовсе не означает, что стандартное настольное приложение не может использовать Ink API, т.к. те же самые элементы управления могут работать с вводом от мыши.

Пространство имен System.Windows.Ink сборки PresentationCore.dll содержит разнообразные поддерживающие типы Ink API (например, Stroke и StrokeCollection); однако большинство элементов управления Ink API (наподобие InkCanvas и InkPresenter) упакованы вместе с общими элементами управления WPF в пространстве имен System.Windows.Controls внутри сборки PresentationFramework.dll. Работа с Ink API будет производиться позже в этой главе.

Элементы управления документов WPF

В WPF также предлагаются элементы управления для расширенной обработки документов, позволяющие строить приложения, которые включают функциональность в стиле Adobe PDF. Применяя типы из пространства имен `System.Windows.Documents` (также находящегося в сборке `PresentationFramework.dll`), можно создавать готовые к печати документы, которые поддерживают масштабирование, поиск, пользовательские аннотации (“клеякие” заметки) и другие развитые средства работы с текстом.

Однако “за кулисами” элементы управления документов не используют API-интерфейсы Adobe PDF, а вместо этого работают с API-интерфейсом XML Paper Specification. Конечные пользователи никакой разницы не заметят, поскольку документы PDF и XPS имеют практически идентичный вид и поведение. В действительности доступно множество бесплатных утилит, которые позволяют преобразовывать эти форматы друг в друга на лету. В последующих примерах мы будем иметь дело с некоторыми аспектами элементов управления документами.

Общие диалоговые окна WPF

В WPF также предоставляются несколько общих диалоговых окон, таких как `OpenFileDialog` и `SaveFileDialog`. Эти диалоговые окна определены внутри пространства имен `Microsoft.Win32` из сборки `PresentationFramework.dll`. Работа с любым из этих диалоговых окон сводится к созданию объекта и вызову метода `ShowDialog()`:

```
using Microsoft.Win32;
namespace WpfControls
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private void btnShowDlg_Click(object sender, RoutedEventArgs e)
        {
            // Отобразить диалоговое окно сохранения файла.
            SaveFileDialog saveDlg = new SaveFileDialog();
            saveDlg.ShowDialog();
        }
    }
}
```

Как и следовало ожидать, в этих классах определены различные члены, позволяющие устанавливать фильтры файлов и пути к каталогам и получать доступ к выбранным пользователем файлам. Некоторые диалоговые окна будут использоваться в последующих примерах; кроме того, будет показано, как строить специальные диалоговые окна для получения пользовательского ввода.

Подробные сведения находятся в документации

Целью этой главы не является обзор всех членов каждого элемента управления WPF. Вместо этого будет дан обзор различных элементов управления с упором на лежащую в основе программную модель и ключевые службы, общие для большинства элементов управления WPF.

Чтобы получить полное представление о конкретной функциональности определенного элемента управления, обращайтесь в документацию .NET Framework 4.5 SDK.

В частности, элементы управления описаны в разделе Control Library (Библиотека элементов управления) справочной системы, ссылка на который находится ниже ссылки Windows Presentation Foundation⇒Controls (Windows Presentation Foundation⇒Элементы управления), как показано на рис. 28.1.

Здесь вы найдете исчерпывающие описания каждого элемента управления, разнообразие примеры кода (в XAML и C#), а также информацию о цепочке наследования, реализованных интерфейсах и примененных атрибутах для любого элемента управления. Обязательно уделите время на исследование элементов управления, рассматриваемых в настоящей главе.

Краткий обзор визуального конструктора WPF в Visual Studio

Большинство этих стандартных элементов управления WPF упаковано в пространство имен `System.Windows.Controls` внутри сборки `PresentationFramework.dll`. При построении WPF-приложения в Visual Studio большинство этих элементов находится в панели инструментов, когда в активном окне открыт визуальный конструктор WPF (рис. 28.2).

Как и в случае других инфраструктур для построения пользовательских интерфейсов в Visual Studio, эти элементы управления можно перетаскивать на поверхность визуального конструктора WPF и конфигурировать их в окне Properties (Свойства), как было показано в главе 27. Хотя Visual Studio сгенерирует значительный объем XAML автоматически, нет ничего необычного в ручном редактировании разметки. Давайте рассмотрим основы.

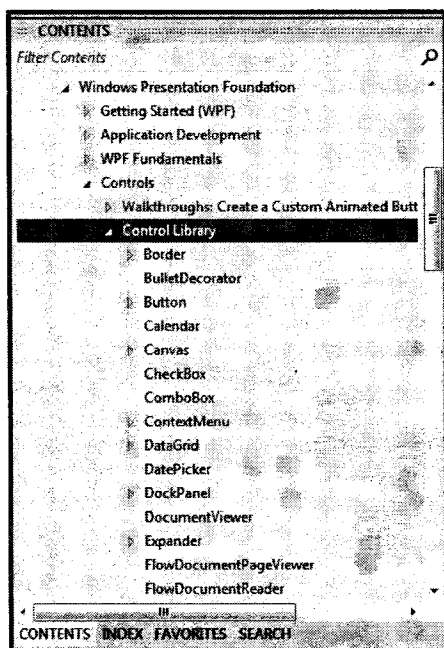


Рис. 28.1. Детальная информация по каждому элементу управления WPF доступна по нажатию клавиши <F1>

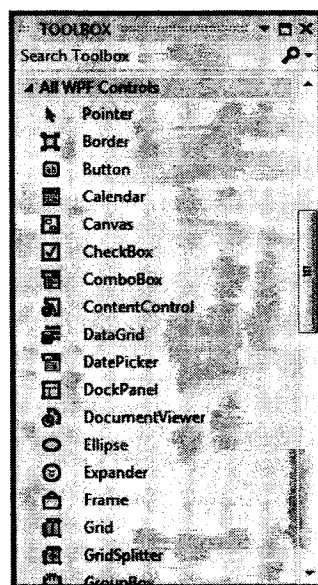


Рис. 28.2. Панель инструментов Visual Studio представляет многие часто используемые элементы управления WPF

Работа с элементами управления WPF в Visual Studio

Вы можете вспомнить из главы 27, что после помещения элемента управления WPF на поверхность визуального конструктора Visual Studio необходимо установить свойство `x:Name` в окне Properties, поскольку оно позволяет обращаться к объекту в связанном файле кода C#. Кроме того, для генерации обработчиков событий в выбранном элементе управления можно использовать вкладку Events (События) окна Properties. Таким образом, с помощью Visual Studio можно сгенерировать следующую разметку для простого элемента управления Button:

```
Button x:Name="btnMyButton" Content="Click Me!" Height="23" Width="140"
Click="btnMyButton_Click" />
```

Здесь свойство Content элемента Button устанавливается в простую строку "Click Me!". Тем не менее, благодаря модели содержимого элементов управления WPF, можно оформить элемент Button, имеющий следующее сложное содержимое:

```
<Button x:Name="btnMyButton" Height="121" Width="156"
Click="btnMyButton_Click">
  <Button.Content>
    <StackPanel Height="95" Width="128" Orientation="Vertical">
      <Ellipse Fill="Red" Width="52" Height="45" Margin="5"/>
      <Label Width="59" FontSize="20" Content="Click!" Height="36" />
    </StackPanel>
  </Button.Content>
</Button>
```

Вы можете также вспомнить, что непосредственный дочерний элемент унаследованного от ContentControl класса — это неявное содержимое, поэтому при указании сложного содержимого определять контекст `<Button.Content>` явно не требуется. В такой ситуации достаточно написать следующую разметку:

```
<Button x:Name="btnMyButton" Height="121" Width="156"
Click="btnMyButton_Click">
  <StackPanel Height="95" Width="128" Orientation="Vertical">
    <Ellipse Fill="Red" Width="52" Height="45" Margin="5"/>
    <Label Width="59" FontSize="20" Content="Click!" Height="36" />
  </StackPanel>
</Button>
```

В любом случае вы устанавливаете свойство Content кнопки в `<StackPanel>` со связанными элементами. Создавать такого рода сложное содержимое можно также с использованием визуального конструктора Visual Studio. После определения диспетчера компоновки можно выбрать в визуальном конструкторе в качестве целевого компонента, куда будут перетаскиваться внутренние элементы управления. Каждый из них можно конфигурировать в окне Properties. Если вы применяли окно Properties для обработки события Click элемента управления Button (как было показано в предшествующих объявлениях XAML), то IDE-среда сгенерирует пустой обработчик события, в который можно будет добавить собственный код. Например:

```
private void btnMyButton_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show("You clicked the button!");
}
```

Работа с редактором Document Outline

Вы должны также помнить, что окно Document Outline (Схема документа) в Visual Studio (которое можно открыть через меню View⇒Other Windows (Вид⇒Другие окна))

удобно при проектировании элемента управления WPF со сложным содержимым. На рис. 28.3 показано логическое дерево XAML для создаваемого элемента Window. Щелчок на любом из этих узлов приводит к его автоматическому выбору в визуальном конструкторе для редактирования.

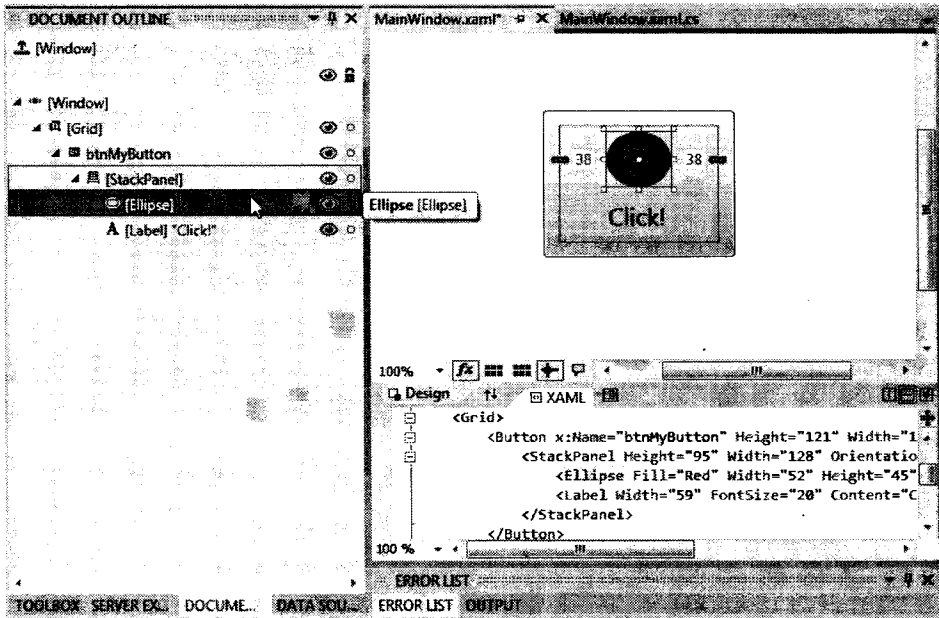


Рис. 28.3. Окно Document Outline в Visual Studio помогает в навигации по сложному содержимому

В текущей версии Visual Studio редактор Document Outline имеет несколько дополнительных функциональных возможностей, которые вы можете считать удобными. Обратите внимание, что справа от любого узла находится значок, напоминающий глазное яблоко. Щелчок на этом значке позволяет скрывать или отображать элемент в визуальном конструкторе, что может оказаться полезным, когда необходимо сосредоточиться на отдельном сегменте при редактировании (следует отметить, что это не приводит к исчезновению элемента во время выполнения, а просто к его сокрытию на поверхности визуального конструктора).

Справа от значка с глазным яблоком находится еще один значок, который позволяет «блокировать» элемент в визуальном конструкторе. Как и можно было догадаться, это очень удобно, когда нужно воспрепятствовать случайному изменению разметки XAML для конкретного элемента. В действительности блокировка элемента делает его предназначенным только для чтения во время проектирования (что, очевидно, не мешает изменять его состояние во время выполнения).

Управление компоновкой содержимого с использованием панелей

Реальное приложение WPF неизбежно содержит большое количество элементов пользовательского интерфейса (элементов ввода, графического содержимого, систем меню, строк состояния и т.п.), которые должны быть хорошо организованы внутри содержащих их окон. Кроме того, виджеты пользовательского интерфейса должны вести себя

адекватно в случае изменения конечным пользователем размеров всего окна или его части (как в случае окна с разделителем). Для обеспечения того, что элементы управления WPF сохранят свое положение в принимающем окне, предусмотрено множество *типов панелей* (которые также называются *диспетчерами компоновки*).

По умолчанию новый WPF-элемент Window, созданный с помощью Visual Studio, будет использовать диспетчер компоновки типа <Grid> (более подробно о нем пойдет речь ниже). Тем не менее, сейчас предположим, что в Window отсутствуют какие-либо объявленные диспетчеры компоновки. Вот пример:

```
<Window x:Class="MyWPFApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  ...
  Title="Fun with Panels!" Height="285" Width="325">
</Window>
```

При объявлении элемента управления непосредственно внутри окна, не имеющего панелей, он размещается в центре контейнера. Рассмотрим следующее простое объявление окна, которое содержит единственный элемент управления Button. Независимо от того, как будут изменяться размеры окна, этот виджет пользовательского интерфейса всегда окажется на равном удалении от всех четырех границ клиентской области. Размер Button определяется установленными значениями свойств Height и Width элемента Button.

```
<!-- Эта кнопка всегда находится в центре окна -->
<Window x:Class="MyWPFApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  ...
  Title="Fun with Panels!" Height="285" Width="325">
  <Button x:Name="btnOK" Height = "100" Width="80" Content="OK"/>
</Window>
```

Также вспомните, что попытка разместить несколько элементов прямо в контексте <Window> приводит к ошибкам разметки и компиляции. Причина в том, что свойству Content окна (или любого потомка ContentControl) может быть присвоен только один объект. Следовательно, показанная ниже разметка XAML приведет к ошибкам разметки и компиляции:

```
<!-- Ошибка! Свойство Content неявно устанавливается более одного раза! -->
<Window x:Class="MyWPFApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  ...
  Title="Fun with Panels!" Height="285" Width="325">
  <!-- Ошибка! Два непосредственных дочерних элемента в <Window>! -->
  <Label x:Name="lblInstructions"
    Width="328" Height="27" FontSize="15" Content="Enter Information"/>
  <Button x:Name="btnOK" Height = "100" Width="80" Content="OK"/>
</Window>
```

Понятно, что от окна, которое может содержать единственный элемент, толку мало. Поэтому, когда в окно требуется поместить множество элементов, это делается с помощью панелей. Панель будет содержать все элементы пользовательского интерфейса, представляющие окно, а сама панель используется как единственный объект, присваиваемый свойству Content окна.

Пространство имен `System.Windows.Controls` предлагает многочисленные панели, каждая из которых по-своему обслуживает внутренние элементы. С помощью панелей вы сможете устанавливать поведение элементов управления при изменении размеров окна пользователем — будут они оставаться в том же месте, куда были помещены во время проектирования, будут располагаться свободным потоком слева направо или сверху вниз, и т.д.

В одних панелях допускается размещать другие панели (например, `DockPanel` может содержать `StackPanel` с другими элементами), чтобы обеспечить еще более высокую гибкость и степень управления. В табл. 28.2 кратко описаны некоторые из наиболее часто используемых панелей WPF.

Таблица 28.2. Основные панели WPF

Панель	Описание
Canvas	Предлагает классический режим размещения содержимого. Элементы остаются в точности там, где были размещены во время проектирования
DockPanel	Привязывает содержимое к определенной стороне панели (Top (верхняя), Bottom (нижняя), Left (левая) или Right (правая))
Grid	Располагает содержимое внутри серии ячеек, расположенных в табличной сетке
StackPanel	Укладывает содержимое по вертикали или горизонтали, в зависимости от значения свойства Orientation
WrapPanel	Позиционирует содержимое слева направо, перенося на следующую строку по достижении границы панели. Последовательность размещения происходит сначала сверху вниз или сначала слева направо, в зависимости от значения свойства Orientation

В следующих нескольких разделах будет показано, как применять эти типы панелей; для этого предопределенные данные XAML будут копироваться в приложение `MyXamlPad.exe`, созданное в главе 27 (при желании данные можно также загружать и в редактор `Кахамл`). Необходимые файлы XAML находятся в подкаталоге `PanelMarkup` каталога `Chapter 28` (рис. 28.4).

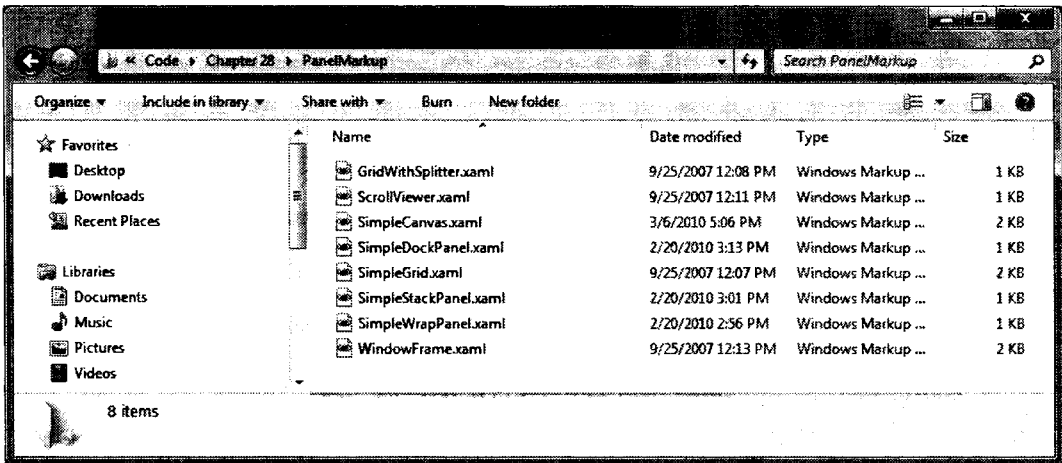


Рис. 28.4. Для тестирования различных компоновок эти XAML-данные будут загружаться в приложение `MyXamlPad.exe`

Позиционирование содержимого внутри панелей Canvas

Панель Canvas поддерживает абсолютное позиционирование содержимого пользовательского интерфейса. Если конечный пользователь изменяет размер окна, делая его меньше, чем компоновка, обслуживаемая панелью Canvas, ее внутреннее содержимое становится невидимым до тех пор, пока контейнер вновь не увеличится до размера, равного или превышающего начальный размер области Canvas.

Для добавления содержимого к Canvas определите необходимые элементы управления внутри области между открывающим (`<Canvas>`) и закрывающим (`</Canvas>`) дескрипторами. Затем для каждого элемента управления укажите левый верхний угол с использованием свойств `Canvas.Top` и `Canvas.Left`; здесь должна начинаться визуализация. Нижняя правая область для каждом элементе управления может быть задана неявно, за счет установки свойств `Canvas.Height` и `Canvas.Width`, либо явно — через свойства `Canvas.Right` и `Canvas.Bottom`.

Чтобы увидеть Canvas в действии, откройте файл `SimpleCanvas.xaml` (входящий в состав примеров кода для этой главы) в текстовом редакторе и скопируйте его содержимое в приложение `MyXamlPad.exe` (или `KaXaml`). Вы должны увидеть следующее определение Canvas:

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  ...
  Title="Fun with Panels!" Height="285" Width="325">
  <Canvas Background="LightSteelBlue">
    <Button x:Name="btnOK" Canvas.Left="212" Canvas.Top="203"
      Width="80" Content="OK"/>
    <Label x:Name="lblInstructions" Canvas.Left="17" Canvas.Top="14"
      Width="328" Height="27" FontSize="15"
      Content="Enter Car Information"/>
    <Label x:Name="lblMake" Canvas.Left="17" Canvas.Top="60"
      Content="Make"/>
    <TextBox x:Name="txtMake" Canvas.Left="94" Canvas.Top="60"
      Width="193" Height="25"/>
    <Label x:Name="lblColor" Canvas.Left="17" Canvas.Top="109"
      Content="Color"/>
    <TextBox x:Name="txtColor" Canvas.Left="94" Canvas.Top="107"
      Width="193" Height="25"/>
    <Label x:Name="lblPetName" Canvas.Left="17" Canvas.Top="155"
      Content="Pet Name"/>
    <TextBox x:Name="txtPetName"
      Canvas.Left="94" Canvas.Top="153"
      Width="193" Height="25"/>
  </Canvas>
</Window>
```

Щелчок на кнопке View Xaml (Показать XAML) приведет к отображению окна, показанного на рис. 28.5.

Обратите внимание, что порядок, в котором являются элементы содержимого внутри Canvas, для вычисления местоположения не используется; вместо этого оно базируется на размерах элемента и свойствах `Canvas.Top`, `Canvas.Bottom`, `Canvas.Left` и `Canvas.Right`.

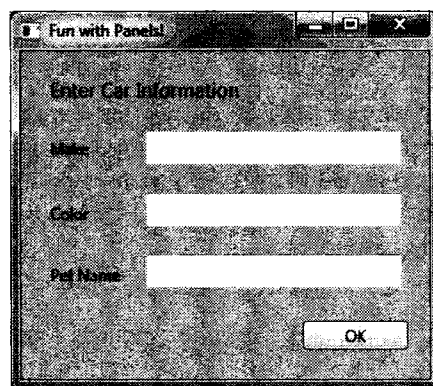


Рис. 28.5. Диспетчер компоновки Canvas обеспечивает абсолютное позиционирование содержимого

На заметку! Если подэлементы внутри Canvas не определяют специфическое местоположение с использованием синтаксиса присоединяемых свойств (например, `Canvas.Left` и `Canvas.Top`), они автоматически прикрепляются к верхнему левому углу Canvas.

Хотя применение типа Canvas может показаться предпочтительным способом организации содержимого (поскольку выглядит столь знакомым), этому подходу присущи некоторые ограничения. Во-первых, элементы внутри Canvas не изменяют себя динамически при применении стилей или шаблонов (например, их шрифты не изменяются). Во-вторых, панель Canvas не пытается сохранять элементы видимыми, когда пользователь изменяет размер окна в меньшую сторону.

Возможно, наилучшим применением типа Canvas является позиционирование *графического содержимого*. Например, при построении изображения с использованием XAML определенно понадобится, чтобы все линии, фигуры и текст оставались именно там, где они были размещены, а не позволять им динамически перемещаться при изменении размеров окна. Мы еще вернемся к Canvas в главе 29, когда речь пойдет о службах визуализации графики WPF.

Позиционирование содержимого внутри панелей WrapPanel

WrapPanel позволяет определить содержимое, которое обтекает панель при изменении размеров окна. При позиционировании элементов внутри WrapPanel вы не указываете положение относительно верхней, левой, нижней и правой границ, как это обычно делается с Canvas. Однако для каждого подэлемента могут быть определены значения Height и Width (наряду с другими значениями свойств) для управления общим размером контейнера.

Поскольку содержимое внутри WrapPanel не стыкуется к определенной стороне панели, порядок определения элементов важен (содержимое визуализируется от первого элемента к последнему). В файле SimpleWrapPanel.xaml имеется следующая разметка (заключенная в определении <Window>):

```
<WrapPanel Background="LightSteelBlue">
  <Label x:Name="lblInstruction" Width="328"
    Height="27" FontSize="15" Content="Enter Car Information"/>
  <Label x:Name="lblMake" Content="Make"/>
  <TextBox Name="txtMake" Width="193" Height="25"/>
  <Label x:Name="lblColor" Content="Color"/>
  <TextBox Name="txtColor" Width="193" Height="25"/>
  <Label x:Name="lblPetName" Content="Pet Name"/>
  <TextBox x:Name="txtPetName" Width="193" Height="25"/>
  <Button x:Name="btnOK" Width="80" Content="OK"/>
</WrapPanel>
```

Загрузив эту разметку и попробовав изменить ширину окна, содержимое будет перетекать внутри окна слева направо (рис. 28.6).

По умолчанию содержимое WrapPanel перетекает слева направо. Однако если изменить значение свойства Orientation на Vertical, то содержимое будет располагаться сверху вниз:

```
<WrapPanel Background="LightSteelBlue" Orientation="Vertical">
```

Панель WrapPanel (как и ряд других типов панелей) может быть объявлена с указанием значений ItemWidth и ItemHeight, которые управляют стандартным размером каждого элемента. Если подэлемент предоставляет собственные значения Height и/или Width, он позиционируется относительно размера, установленного для него панелью.

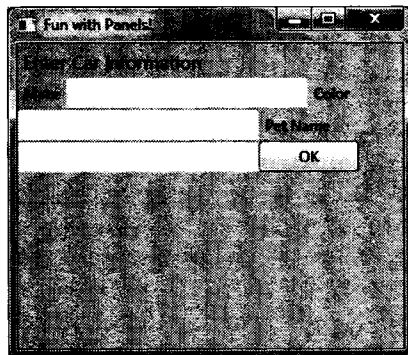


Рис. 28.6. Содержимое WrapPanel ведет себя подобно традиционной HTML-странице

Рассмотрим следующую разметку:

```
<WrapPanel Background="LightSteelBlue" ItemWidth="200" ItemHeight="30">
  <Label x:Name="lblInstruction"
    FontSize="15" Content="Enter Car Information"/>
  <Label x:Name="lblMake" Content="Make"/>
  <TextBox x:Name="txtMake"/>
  <Label x:Name="lblColor" Content="Color"/>
  <TextBox x:Name="txtColor"/>
  <Label x:Name="lblPetName" Content="Pet Name"/>
  <TextBox x:Name="txtPetName"/>
  <Button x:Name="btnOK" Width="80" Content="OK"/>
</WrapPanel>
```

После визуализации получается окно, показанное на рис. 28.7 (обратите внимание на размер и положение элемента управления Button, для которого было задано уникальное значение Width).

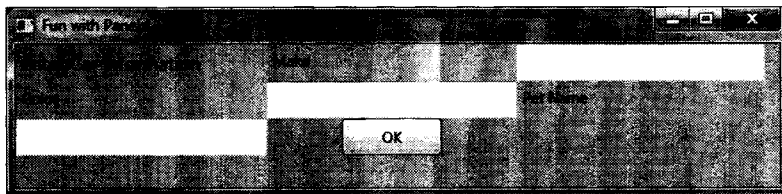


Рис. 28.7. Панель WrapPanel может устанавливать ширину и высоту каждого заданного элемента

Взглянув на рис. 28.7, трудно не согласиться с тем, что панель WrapPanel — обычно не лучший выбор для организации содержимого непосредственно в окне, поскольку ее элементы могут перемешиваться, когда пользователь изменяет размеры окна. В большинстве случаев WrapPanel будет подэлементом панели другого типа, позволяя небольшой области окна переносить свое содержимое при изменении размера (как, например, элемент управления Toolbar).

Позиционирование содержимого внутри панелей StackPanel

Подобно WrapPanel, элемент управления StackPanel организует содержимое в одну строку, которая может быть ориентирована горизонтально или вертикально (по умолчанию), в зависимости от значения, присвоенного свойству Orientation. Однако от-

личие между ними состоит в том, что `StackPanel` не пытается переносить содержимое при изменении размера окна пользователем. Вместо этого элементы внутри `StackPanel` просто растягиваются (в зависимости от ориентации), приспособливаясь к размеру самой панели `StackPanel`. Например, в файле `SimpleStackPanel.xaml` содержится следующая разметка, которая дает вывод, показанный на рис. 28.8:

```
<StackPanel Background="LightSteelBlue">
  <Label x:Name="lblInstruction"
    FontSize="15" Content="Enter Car Information"/>
  <Label x:Name="lblMake" Content="Make"/>
  <TextBox x:Name="txtMake"/>
  <Label x:Name="lblColor" Content="Color"/>
  <TextBox Name="txtColor"/>
  <Label x:Name="lblPetName" Content="Pet Name"/>
  <TextBox x:Name="txtPetName"/>
  <Button x:Name="btnOK" Content="OK"/>
</StackPanel>
```

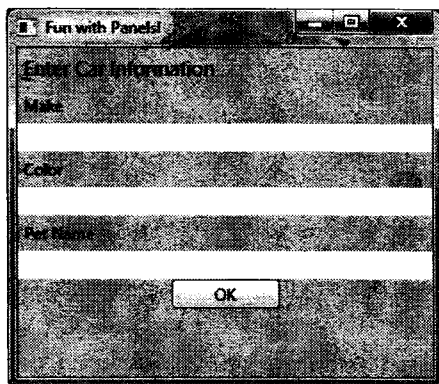


Рис. 28.8. Вертикальное расположение содержимого

Если присвоить свойству `Orientation` значение `Horizontal`, то визуализированный вывод станет таким, как показано на рис. 28.9:

```
<StackPanel Background="LightSteelBlue" Orientation="Horizontal">
```

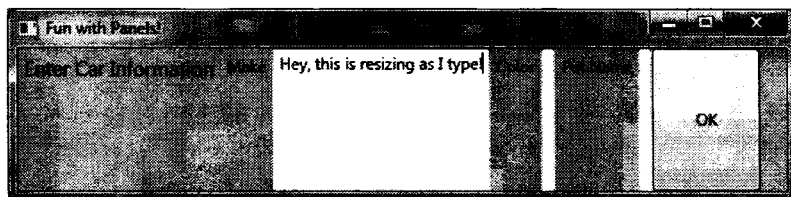


Рис. 28.9. Горизонтальное расположение содержимого

Опять-таки, как и `WrapPanel`, использовать `StackPanel` для организации содержимого прямо в окне редко когда придется. Вместо этого `StackPanel` больше подходит в качестве вложенной панели какой-нибудь главной панели.

Позиционирование содержимого внутри панелей `Grid`

Из всех панелей, предлагаемых API-интерфейсами WPF, наиболее гибкой является `Grid`. Подобно HTML-таблице, `Grid` может состоять из набора ячеек, каждая из которых имеет свое содержимое.

При определении Grid выполняются следующие шаги.

1. Определение и конфигурирование каждого столбца.
2. Определение и конфигурирование каждой строки.
3. Назначение содержимого каждой ячейке сетки с использованием синтаксиса присоединяемых свойств.

На заметку! Если не определить никаких строк и столбцов, то по умолчанию <Grid> будет состоять из одной ячейки, занимающей всю поверхность окна. Кроме того, если не указать ячейку для подэлемента <Grid>, он автоматически разместится в столбце 0 и строке 0.

Первые два шага (определение столбцов и строк) выполняются за счет использования элементов <Grid.ColumnDefinitions> и <Grid.RowDefinitions>, которые содержат коллекции элементов <ColumnDefinition> и <RowDefinition>, соответственно. Поскольку каждая ячейка внутри сетки является настоящим объектом .NET, можно должным образом конфигурировать внешний вид и поведение каждого элемента.

Ниже приведено простое определение <Grid> (из файла SimpleGrid.xaml), которое организует содержимое пользовательского интерфейса, как показано на рис. 28.10:

```
<Grid ShowGridLines="True" Background="LightSteelBlue">
  <!-- Определить строки/столбцы -->
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition/>
    <RowDefinition/>
  </Grid.RowDefinitions>

  <!-- Добавить элементы в ячейки сетки -->
  <Label x:Name="lblInstruction" Grid.Column="0" Grid.Row="0"
    FontSize="15" Content="Enter Car Information"/>
  <Button x:Name="btnOK" Height="30" Grid.Column="0"
    Grid.Row="0" Content="OK"/>
  <Label x:Name="lblMake" Grid.Column="1"
    Grid.Row="0" Content="Make"/>
  <TextBox x:Name="txtMake" Grid.Column="1"
    Grid.Row="0" Width="193" Height="25"/>
  <Label x:Name="lblColor" Grid.Column="0"
    Grid.Row="1" Content="Color"/>
  <TextBox x:Name="txtColor" Width="193" Height="25"
    Grid.Column="0" Grid.Row="1" />

  <!-- Чтобы сделать картину интереснее, добавить цвет ячейке с именем -->
  <Rectangle Fill="LightGreen" Grid.Column="1" Grid.Row="1" />
  <Label x:Name="lblPetName" Grid.Column="1" Grid.Row="1" Content="Pet Name"/>
  <TextBox x:Name="txtPetName" Grid.Column="1" Grid.Row="1"
    Width="193" Height="25"/>
</Grid>
```

Обратите внимание, что каждый элемент (включая элемент Rectangle светло-зеленого цвета) прикрепляется к ячейке сетки, используя присоединяемые свойства Grid.Row и Grid.Column. Стандартный порядок ячеек начинается с левой верхней, которая указана с помощью Grid.Column="0" и Grid.Row="0". Учитывая, что сетка состоит всего из четырех ячеек, нижняя правая ячейка может быть идентифицирована как Grid.Column="1" и Grid.Row="1".

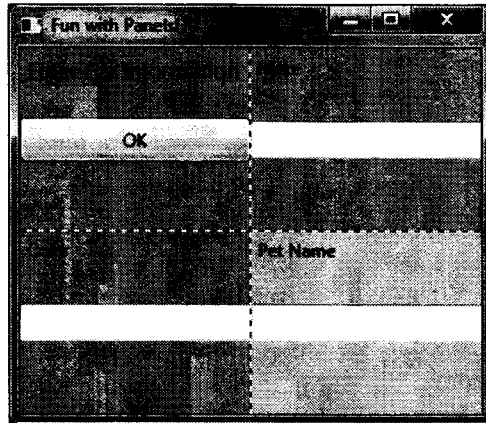


Рис. 28.10. Панель Grid в действии

Панели Grid с типами GridSplitter

Панели Grid также могут поддерживать разделители. Как известно, разделители позволяют конечному пользователю изменять размеры столбцов и строк сетки. При этом содержимое каждой ячейки с изменяемым размером реорганизуется на основе содержащихся в нем элементов. Добавить разделители к Grid довольно просто: для этого необходимо определить элемент управления `<GridSplitter>` и с применением синтаксиса присоединяемых свойств указать строку или столбец, к которому он относится.

Имейте в виду, что для обеспечения видимости на экране разделителя потребуется указать значение `Width` или `Height` (в зависимости от вертикального или горизонтального разделения). Ниже показана простая панель Grid с разделителем на первом столбце (`Grid.Column="0"`) из файла `GridWithSplitter.xaml`:

```
<Grid Background="LightSteelBlue">
  <!-- Определить столбцы -->
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto"/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>

  <!-- Добавить метку в ячейку 0 -->
  <Label x:Name="lblLeft" Background="GreenYellow"
        Grid.Column="0" Content="Left!"/>

  <!-- Определить разделитель -->
  <GridSplitter Grid.Column="0" Width="5"/>

  <!-- Добавить метку в ячейку 1 -->
  <Label x:Name="lblRight" Grid.Column="1" Content="Right!"/>
</Grid>
```

Прежде всего, обратите внимание, что столбец с разделителем имеет свойство `Width`, значение которого установлено в `Auto`. Кроме того, элемент `<GridSplitter>` использует синтаксис присоединяемых свойств для установки столбца, с которым он работает. Если вы взглянете на вывод, то увидите там 5-пиксельный разделитель, который позволяет изменять размер каждого элемента `Label`. Поскольку для каждого элемента `Label` не было задано свойство `Height` или `Width`, они заполняют всю ячейку (рис. 28.11).

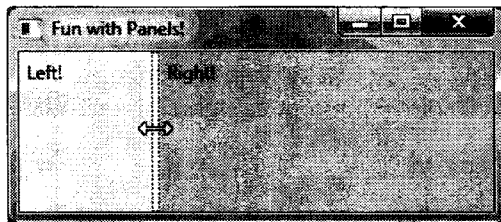


Рис. 28.11. Панель Grid с разделителями

Позиционирование содержимого внутри панелей DockPanel

Панель DockPanel обычно применяется в качестве контейнера, который включает любое количество дополнительных панелей для группирования связанного содержимого. Панели DockPanel используют синтаксис присоединяемых свойств (как было показано в типах Canvas и Grid) для управления тем, куда будет стыкован каждый элемент внутри DockPanel.

В файле SimpleDockPanel.xaml определена следующая простая панель DockPanel, которая дает результат, показанный на рис. 28.12:

```
<DockPanel LastChildFill="True">
  <!-- Стыковать элементы к панели -->
  <Label x:Name="lblInstruction" DockPanel.Dock="Top"
    FontSize="15" Content="Enter Car Information"/>
  <Label x:Name="lblMake" DockPanel.Dock="Left" Content="Make"/>
  <Label x:Name="lblColor" DockPanel.Dock="Right" Content="Color"/>
  <Label x:Name="lblPetName" DockPanel.Dock="Bottom" Content="Pet Name"/>
  <Button x:Name="btnOK" Content="OK"/>
</DockPanel>
```

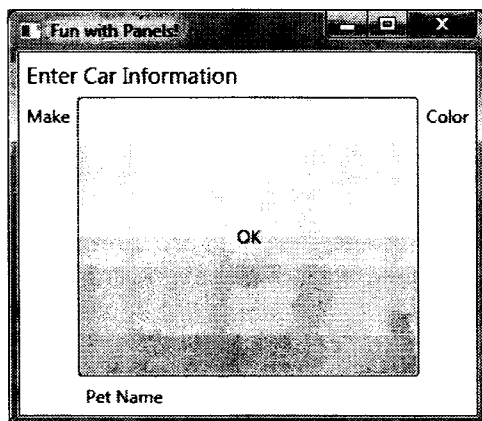


Рис. 28.12. Простая панель DockPanel

На заметку! Если добавить множество элементов к одной стороне DockPanel, они выстроятся вдоль указанной грани в порядке их объявления.

Преимущество применения типов DockPanel состоит в том, что при изменении размеров окна пользователем каждый элемент остается прикрепленным к указанной (в DockPanel.Dock) стороне панели. Также обратите внимание, что в открывающем дескрипторе <DockPanel> приведенного примера атрибут LastChildFill устанавливается в true. Учитывая, что элемент Button на самом деле является “последним дочерним” элементом в контейнере, он будет растянут, чтобы занять все оставшееся место.

Включение прокрутки в типах панелей

В WPF предусмотрен класс `<ScrollView>`, который обеспечивает возможность прокрутки данных внутри объектов панелей. Ниже показан фрагмент разметки из файла `ScrollView.xaml`:

```
<ScrollView>
  <StackPanel>
    <Button Content="First" Background="Green" Height="40"/>
    <Button Content="Second" Background="Red" Height="40"/>
    <Button Content="Third" Background="Pink" Height="40"/>
    <Button Content="Fourth" Background="Yellow" Height="40"/>
    <Button Content="Fifth" Background="Blue" Height="40"/>
  </StackPanel>
</ScrollView>
```

Результат приведенного определения XAML можно видеть на рис. 28.13.

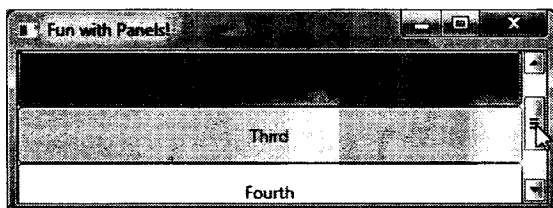


Рис. 28.13. Работа с классом `ScrollView`

Как и следовало ожидать, каждая панель предоставляет множество членов, позволяющих настраивать расположение содержимого. В частности, элементы управления WPF поддерживают два таких свойства (`Padding` и `Margin`), которые позволяют самому элементу информировать панель о том, как с ним нужно обращаться. В частности, свойство `Padding` контролирует свободное пространство, которое должно окружать внутренний элемент управления, а свойство `Margin` — пространство, окружающее элемент управления извне.

На этом краткий экскурс в основные типы панелей WPF и различные способы позиционирования их содержимого завершен. Далее мы перейдем к рассмотрению примера, в котором используются вложенные панели для создания системы компоновки главного окна. А теперь давайте посмотрим, как создавать компоновки с помощью визуальных конструкторов Visual Studio.

Конфигурирование панелей с использованием визуальных конструкторов Visual Studio

Теперь, когда вы бегло ознакомились с разметкой XAML, применяемой для определения ряда общих диспетчеров компоновки, знайте, что в Visual Studio имеется довольно хорошая поддержка конструирования компоновок. Ключевым компонентом является окно Document Outline, описанное ранее в этой главе. Для иллюстрации некоторых основ создадим новый проект WPF Application (WPF-приложение) по имени `VisualLayoutTesterApp` (данный пример предназначен только для демонстрации работы с визуальными конструкторами, поэтому он не включен в состав загружаемого исходного кода для книги).

В первоначальной разметке для Window по умолчанию используется диспетчер компоновки `Grid`, как показано ниже:

```
<Window x:Class="VisualLayoutTesterApp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
    </Grid>
</Window>
```

Если система компоновки Grid устраивает, обратите внимание на рис. 28.14, что в визуальном конструкторе можно легко разделять и менять размеры ячеек сетки. Для этого сначала выберите компонент Grid в окне Document Outline, а затем щелкните на границе сетки, чтобы создать новые строки и колонки.

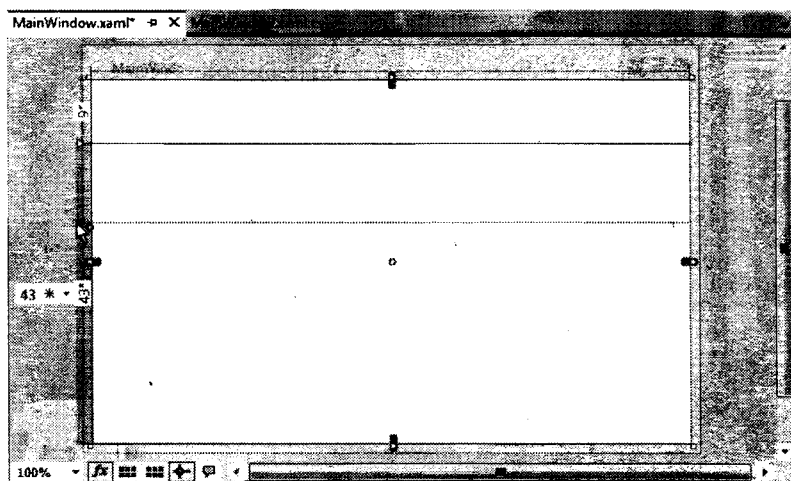


Рис. 28.14. Элемент управления Grid может быть визуально разделен на ячейки с применением конструктора IDE-среды

Теперь предположим, что определена сетка с некоторым количеством ячеек. Затем можно перетаскивать элементы управления на заданную ячейку сетки и IDE-среда автоматически установит должным образом свойства Grid.Row и Grid.Column элемента управления. Вот как выглядит возможная разметка, сгенерированная IDE-средой после перетаскивания элемента Button в предопределенную ячейку:

```
<Button Content="Button" HorizontalAlignment="Left" Margin="10,10,0,0"
        Grid.Row="1" VerticalAlignment="Top" Width="75"
        Grid.Column="1"/>
```

Пусть, например, вы решили не использовать элемент Grid вообще. Щелкнув правой кнопкой мыши на любом узле разметки в окне Document Outline, вы получите пункт меню, который позволит заменить текущий контейнер другим (рис. 28.15). Это следует делать с особой осторожностью, т.к. при этом, скорее всего, радикально поменяются позиции элементов управления, чтобы удовлетворять правилам панели нового типа.

Другой удобный трюк связан с возможностью выбора в визуальном конструкторе набора элементов управления и группирования их в новый вложенный диспетчер компоновки. Предположим, что имеется панель Canvas, в которой определен набор произвольных объектов (если хотите попробовать, преобразуйте первоначальную панель Grid в Canvas, используя прием, который был показан на рис. 28.15). Теперь выберите набор элементов на поверхности визуального конструктора, щелкая на каждом элементе левой кнопкой мыши при нажатой клавише <Ctrl>. Если затем щелкнуть правой кнопкой мыши на выделенном множестве элементов, можно сгруппировать их в новую вложенную панель (рис. 28.16).

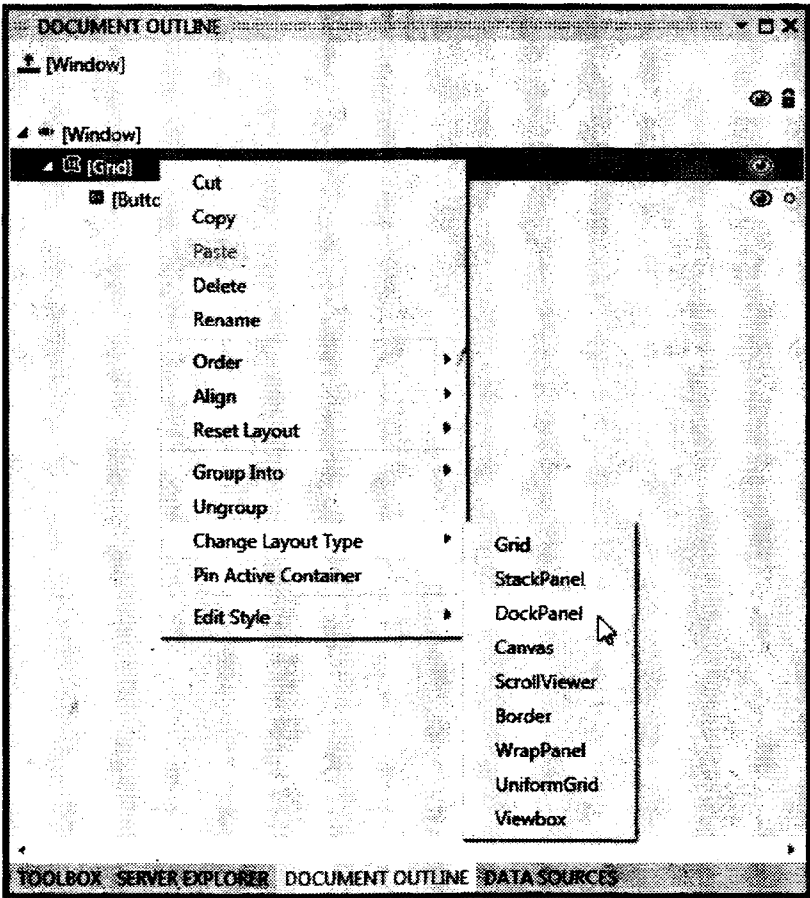


Рис. 28.15. Окно Document Outline позволяет выполнять преобразования в другие типы панелей

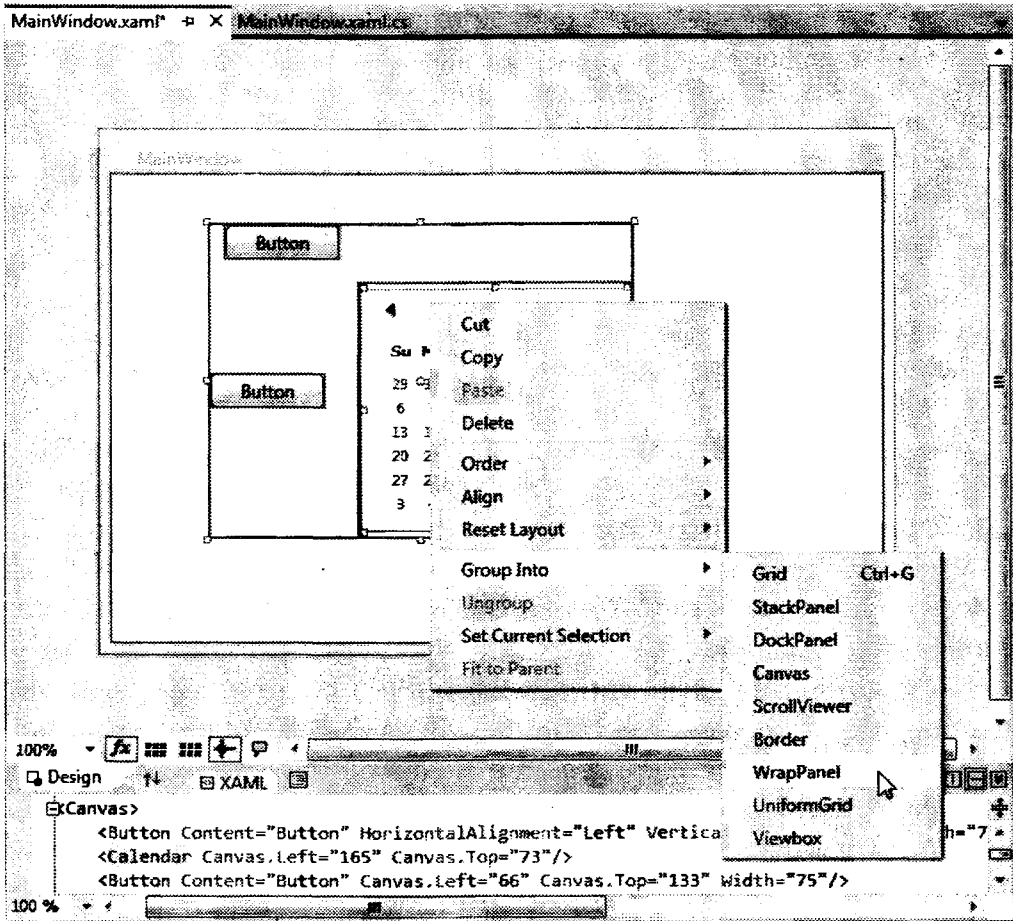


Рис. 28.16. Группирование элементов в новую вложенную панель

После этого загляните в окно Document Outline еще раз, чтобы проверить вложенную систему компоновки. Поскольку вы строите полнофункциональные окна WPF, скорее всего, всегда нужно будет применять вложенную систему компоновки, а не просто выбирать единственную панель для отображения всего пользовательского интерфейса (фактически в оставшихся примерах WPF обычно так и будет делаться). В качестве финального замечания следует указать, что все узлы в окне Document Outline поддерживают перетаскивание. Например, если требуется переместить элемент управления, находящийся внутри Canvas, в родительскую панель, это можно сделать так, как предлагается на рис. 28.17.

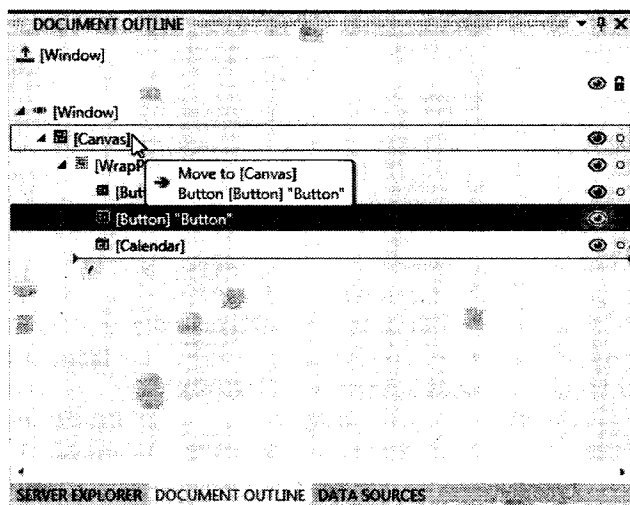


Рис. 28.17. Перемещение элементов с помощью окна Document Outline

В последующих главах, посвященных WPF, будут представлены дополнительные сокращения разметки, где они возможны. Тем не менее, определенно полезно уделите время самостоятельному экспериментированию и проверке разнообразных функциональных средств. В следующем примере этой главы будет показано, как построить вложенный диспетчер компоновки для специального приложения текстового процессора (с поддержкой проверки правописания).

Построение окна с использованием вложенных панелей

Как упоминалось ранее, в типичном окне WPF для получения нужной системы компоновки применяется не единственный элемент управления типа панели, а одни панели вкладываются в другие. Начать следует с создания нового проекта WPF Application по имени MyWordPad.

Наша цель состоит в том, чтобы сконструировать компоновку, в которой главное окно имеет расположенную в верхней части систему меню, под ней — панель инструментов и в нижней части окна — строку состояния. Строка состояния будет содержать область для хранения текстовых сообщений, отображаемых при выборе пункта меню (или кнопки в панели инструментов), а система меню и панель инструментов предоставят триггеры пользовательского интерфейса для закрытия приложения и отображения вариантов правописания в элементе Expander. На рис. 28.18 показана начальная компоновка, в которой отображаются подсказки по правописанию для “WPF”.

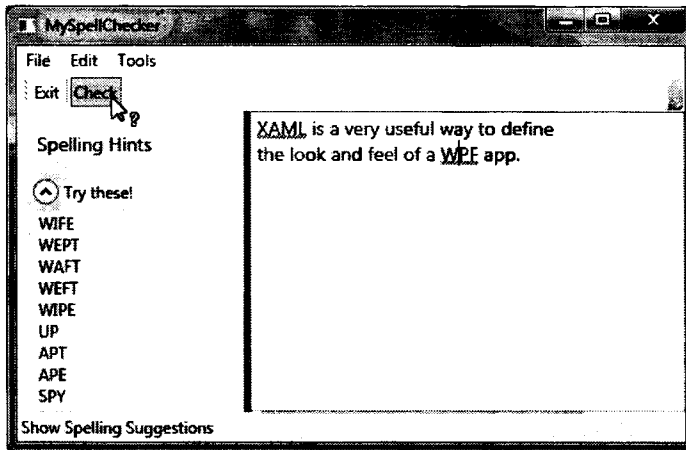


Рис. 28.18. Использование вложенных панелей для построения пользовательского интерфейса окна

Обратите внимание, что две кнопки панели инструментов не содержат в себе ожидаемых изображений, а только текстовые значения. Хотя этого явно не достаточно для профессионального приложения, установка изображений для кнопок панели инструментов обычно предполагает применение встроенных ресурсов, и эта тема рассматривается в главе 29 (а пока обойдемся текстом). Кроме того, при наведении на кнопку Check (Проверить) курсор мыши изменяется, и в единственной панели строки состояния отображается полезное сообщение пользовательского интерфейса.

Чтобы приступить к построению описанного пользовательского интерфейса, модифицируйте начальное определение XAML типа Window для использования в качестве дочернего элемента `<DockPanel>` вместо стандартного `<Grid>`:

```
<Window x:Class="MySpellChecker.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="MySpellChecker" Height="331" Width="508"
  WindowStartupLocation="CenterScreen">

  <!-- Эта панель устанавливает содержимое окна -->
  <DockPanel>
  </DockPanel>

</Window>
```

Построение системы меню

Системы меню в WPF представляются классом `Menu`, который поддерживает коллекцию объектов `MenuItem`. При построении системы меню в XAML каждый `MenuItem` может обрабатывать различные события, из которых в первую очередь следует упомянуть `Click`, возникающее, когда пользователь выбирает подэлемент. В рассматриваемом примере будут созданы два пункта меню верхнего уровня (File (Файл) и Tools (Сервис); меню Edit (Правка) строится позже), которые, соответственно, будут содержать в себе подэлементы Exit (Выход) и Spelling Hints (Подсказки по правописанию).

В дополнение к обработке события `Click` для каждого подэлемента будет также организована обработка событий `MouseEnter` и `MouseExit`, которые используются для установки текста в строке состояния. Добавьте следующую разметку в контекст элемента `<DockPanel>` (для обработки каждого события применяйте окно Properties (Свойства) среды Visual Studio, как было описано в главе 27):

```

<!-- Закрепить систему меню сверху -->
<Menu DockPanel.Dock="Top"
      HorizontalAlignment="Left" Background="White" BorderBrush="Black">
  <MenuItem Header="_File">
    <Separator/>
    <MenuItem Header="_Exit" MouseEnter="MouseEnterExitArea"
              MouseLeave="MouseLeaveArea" Click="FileExit_Click"/>
  </MenuItem>
  <MenuItem Header="_Tools">
    <MenuItem Header="_Spelling Hints" MouseEnter="MouseEnterToolsHintsArea"
              MouseLeave="MouseLeaveArea" Click="ToolsSpellingHints_Click"/>
  </MenuItem>
</Menu>

```

Обратите внимание, что система меню стыкована к верхней части DockPanel. Кроме того, элемент <Separator> используется для добавления в систему меню тонкой горизонтальной линии непосредственно перед пунктом Exit. Значения Header для каждого MenuItem содержат символ подчеркивания (например, _Exit). Это указывает символ, который станет подчеркнутым, когда пользователь нажмет клавишу <Alt> (для ввода клавиатурного сокращения).

После построения системы меню необходимо реализовать различные обработчики событий. Прежде всего, понадобится обработчик пункта меню File⇒Exit по имени FileExit_Click(), который просто закрывает окно, что, в свою очередь, приводит к завершению приложения, поскольку это окно самого высшего уровня. Обработчики событий MouseEnter и MouseExit для каждого из подэлементов должны обновлять строку состояния; однако пока мы просто оставим их пустыми. И, наконец, обработчик ToolsSpellingHints_Click() для пункта меню Tools⇒Spelling Hints также оставим пока пустым. Ниже показаны текущие обновления файла отделенного кода:

```

public partial class MainWindow : System.Windows.Window
{
    public MainWindow()
    {
        InitializeComponent();
    }

    protected void FileExit_Click(object sender, RoutedEventArgs args)
    {
        // Закрыть это окно.
        this.Close();
    }

    protected void ToolsSpellingHints_Click(object sender, RoutedEventArgs args)
    {
    }

    protected void MouseEnterExitArea(object sender, RoutedEventArgs args)
    {
    }

    protected void MouseEnterToolsHintsArea(object sender, RoutedEventArgs args)
    {
    }

    protected void MouseLeaveArea(object sender, RoutedEventArgs args)
    {
    }
}

```

Визуальное построение меню

Хотя всегда полезно знать, как вручную определять элементы в XAML, часто это оказывается довольно утомительным. В Visual Studio поддерживается возможность визуального конструирования систем меню, панелей инструментов, строк состояния и многих других элементов управления пользовательского интерфейса. В качестве краткого примера предположим, что создан новый элемент управления Menu внутри нового Window (можете вставить тестовое окно через пункт меню Project⇒Add Window (Проект⇒Добавить окно) и двигаться дальше). Если теперь щелкнуть правой кнопкой мыши на элементе управления Menu, в контекстном меню появится пункт Add MenuItem (Добавить пункт меню), как показано на рис. 28.19.

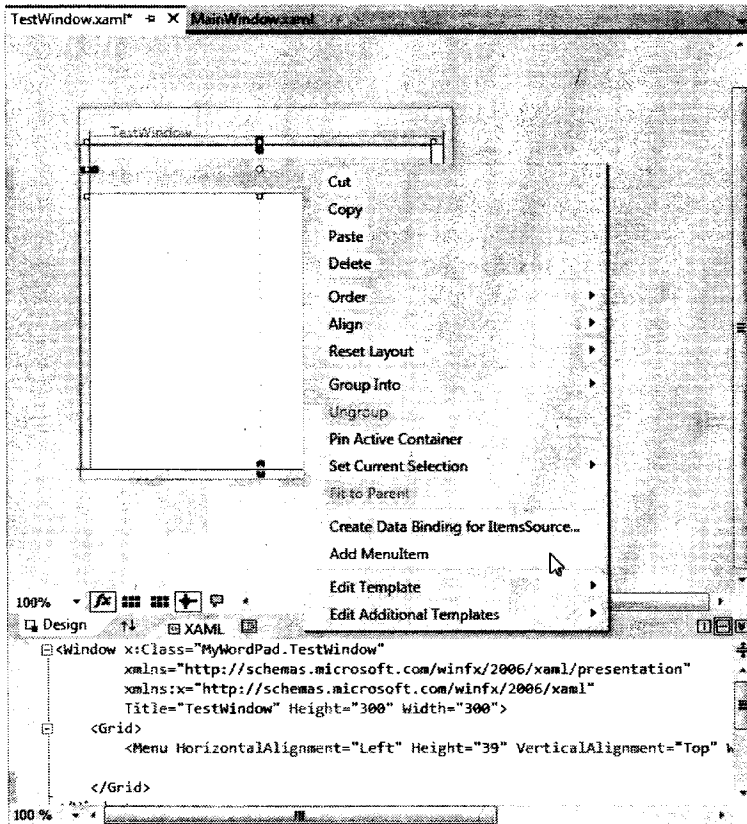


Рис. 28.19. Визуальное добавление элементов в объект Menu

После добавления набора элементов верхнего уровня можно переходить к добавлению пунктов подменю, разделителей, развертыванию и свертыванию самого меню и выполнению других связанных с меню операций, снова производя щелчок правой кнопкой мыши. На рис. 28.20 показан один из возможных способов визуального проектирования простой системы меню (обязательно просмотрите сгенерированную разметку XAML).

По мере рассмотрения оставшейся части примера MyWordPad обычно будет показана финальная сгенерированная разметка XAML; тем не менее, уделите некоторое время на экспериментирование с визуальными конструкторами.

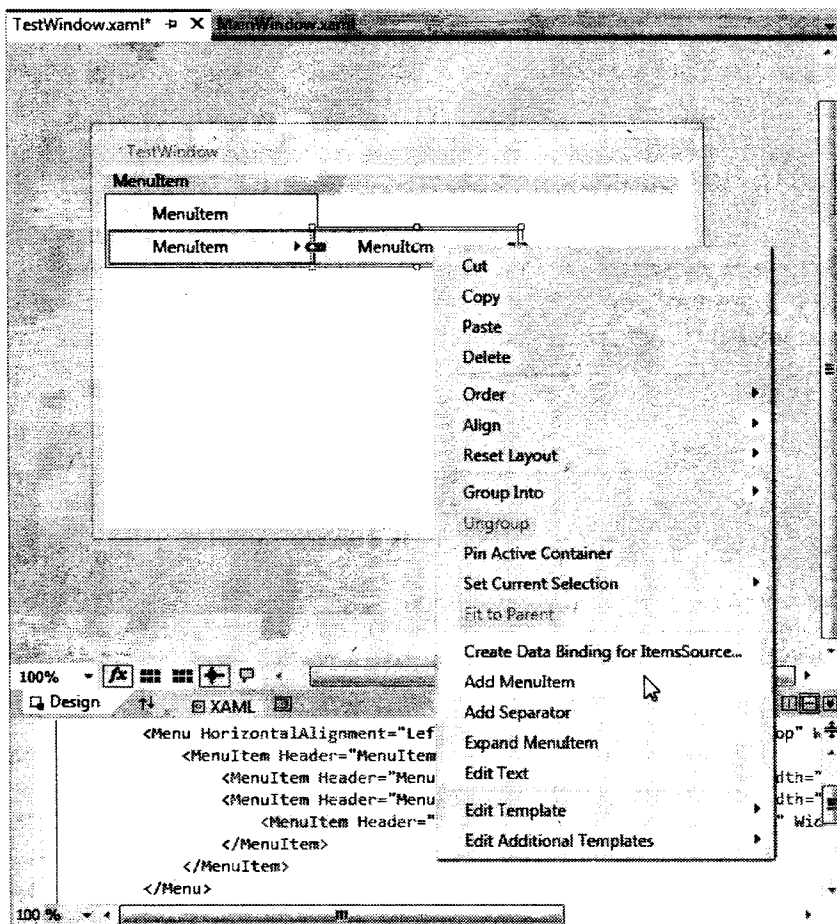


Рис. 28.20. Визуальное добавление элементов в объект MenuItem

Построение панели инструментов

Панели инструментов (представляемые в WPF классом `ToolBar`) обычно предлагают альтернативный способ активизации пунктов меню. Поместите следующую разметку непосредственно после закрывающего дескриптора определения `<Menu>`:

```
<!-- Разместить панель инструментов ниже области меню -->
<ToolBar DockPanel.Dock = "Top" >
  <Button Content = "Exit" MouseEnter = "MouseEnterExitArea"
    MouseLeave = "MouseLeaveArea" Click = "FileExit_Click"/>
  <Separator/>
  <Button Content = "Check" MouseEnter = "MouseEnterToolsHintsArea"
    MouseLeave = "MouseLeaveArea" Click = "ToolsSpellingHints_Click"
    Cursor = "Help" />
</ToolBar>
```

Элемент управления `ToolBar` состоит из двух элементов `Button`, которые предназначены для обработки тех же самых событий теми же методами из файла кода. С помощью этого приема можно дублировать обработчики для обслуживания как пунктов меню, так и кнопок панели управления. Хотя в этой панели используются типичные нажимаемые кнопки, имейте в виду, что `ToolBar` “является” `ContentControl`, и потому на его поверхность можно помещать любые типы (раскрывающиеся списки, изображения, графику и т.п.). Единственное, что еще может интересовать здесь — это то, что кнопка `Check` (Проверить) поддерживает специальный курсор мыши через свойство `Cursor`.

На заметку! Элемент `ToolBar` может быть дополнительно помещен в элемент `<ToolBarTray>`, который управляет компоновкой, стыковкой и перетаскиванием для набора объектов `ToolBar`. За подробной информацией обращайтесь в документацию .NET Framework 4.5 SDK.

Построение строки состояния

Элемент управления `StatusBar` прикрепляется к нижней части `<DockPanel>` и содержит единственный элемент управления `<TextBlock>`, который пока еще в этой главе не использовался. Элемент `TextBlock` может применяться для хранения текста с добавлением форматирования, такого как полужирный текст, подчеркнутый текст, разрывы строк и т.д. Поместите следующую разметку непосредственно после предшествующего определения `ToolBar`:

```
<!-- Разместить строку состояния внизу -->
<StatusBar DockPanel.Dock="Bottom" Background="Beige" >
  <StatusBarItem>
    <TextBlock Name="statBarText" Text="Ready"/>
  </StatusBarItem>
</StatusBar>
```

Завершение проектирования пользовательского интерфейса

Финальный аспект проектирования нашего пользовательского интерфейса заключается в определении элемента `Grid` с разделителями и двумя столбцами. Слева будет помещен элемент управления `Expander`, который отобразит список подсказок по правописанию, помещенный в `<StackPanel>`, а справа — элемент `TextBox` с поддержкой многострочного текста, линейек прокрутки и включенной проверкой орфографии. Элемент `<Grid>` целиком может быть размещен в левой части родительской панели `<DockPanel>`. Чтобы завершить определение пользовательского интерфейса окна, поместите следующую XAML-разметку непосредственно под разметкой, описывающей `StatusBar`:

```
<Grid DockPanel.Dock="Left" Background="AliceBlue">
  <!-- Определить строки и столбцы -->
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>
  <GridSplitter Grid.Column="0" Width="5" Background="Gray" />
  <StackPanel Grid.Column="0" VerticalAlignment="Stretch" >
    <Label Name="lblSpellingInstructions" FontSize="14" Margin="10,10,0,0">
      Spelling Hints
    </Label>
    <Expander Name="expanderSpelling" Header="Try these!" Margin="10,10,10,10">
      <!-- Будет заполняться программно -->
      <Label Name="lblSpellingHints" FontSize="12"/>
    </Expander>
  </StackPanel>
  <!-- Это будет область для ввода -->
  <TextBox Grid.Column="1"
    SpellCheck.IsEnabled="True"
    AcceptsReturn="True"
    Name="txtData" FontSize="14"
    BorderBrush="Blue"
    VerticalScrollBarVisibility="Auto"
    HorizontalScrollBarVisibility="Auto">
  </TextBox>
</Grid>
```

Реализация обработчиков событий `MouseEnter`/`MouseLeave`

К этому моменту пользовательский интерфейс окна готов. Осталось предоставить реализации остальных обработчиков событий. Начните с обновления файла кода C# так, чтобы каждый из обработчиков `MouseEnter` и `MouseLeave` устанавливал в текстовой панели строки состояния соответствующий текст сообщения для конечного пользователя:

```
public partial class MainWindow : System.Windows.Window
{
    ...
    protected void MouseEnterExitArea(object sender, RoutedEventArgs args)
    {
        statBarText.Text = "Exit the Application";
    }
    protected void MouseEnterToolsHintsArea(object sender, RoutedEventArgs args)
    {
        statBarText.Text = "Show Spelling Suggestions";
    }
    protected void MouseLeaveArea(object sender, RoutedEventArgs args)
    {
        statBarText.Text = "Ready";
    }
}
```

В этот момент приложение можно запустить. Обратите внимание, что текст в строке состояния меняется в зависимости от того, над каким пунктом меню или кнопкой панели инструментов расположен курсор.

Реализация логики проверки правописания

API-интерфейс WPF имеет встроенную поддержку проверки правописания, независимую от продуктов Microsoft Office. Это значит, что использовать уровень взаимодействия с COM для обращения к функции проверки правописания Microsoft Word не придется, а вместо этого данная функциональность добавляется с помощью всего нескольких строк кода.

Вспомните, что при определении элемента управления `<TextBox>` свойство `SpellCheck.IsEnabled` было установлено в `true`. После этого неправильно написанные слова будут подчеркиваться красной волнистой линией, как это делается в Microsoft Office. Более того, лежащая в основе программная модель предоставляет доступ к механизму проверки правописания, который позволяет получить список предполагаемых слов, написанных с ошибкой. Добавьте в метод `ToolsSpellingHints_Click()` следующий код:

```
protected void ToolsSpellingHints_Click(object sender, RoutedEventArgs args)
{
    string spellingHints = string.Empty;
    // Попробовать получить ошибку правописания в текущем положении курсора ввода.
    SpellingError error = txtData.GetSpellingError(txtData.CaretIndex);
    if (error != null)
    {
        // Построить строку с подсказками по правописанию.
        foreach (string s in error.Suggestions)
        {
            spellingHints += string.Format("{0}\n", s);
        }
    }
}
```

```
// Отобразить подсказки и раскрыть элемент Expander.
lblSpellingHints.Content = spellingHints;
expanderSpelling.IsExpanded = true;
}
}
```

Приведенный выше код довольно прост. С помощью свойства `CaretIndex` определяется текущее положение курсора ввода в текстовом поле и извлекается объект `SpellingError`. Если в указанном месте имеется ошибка (т.е. значение не равно `null`), производится проход в цикле по списку подсказок с использованием свойства `Suggestions`. После получения все подсказки по правописанию помещаются в метку `Label` внутри элемента `Expander`.

Вот и все! С помощью всего нескольких строк процедурного кода (и приличной порции XAML-разметки) заложен фундамент для будущего текстового процессора. После освоения *управляющих команд* можно будет добавить несколько новых возможностей.

Понятие команд WPF

В Windows Presentation Foundation предлагается поддержка *независимых от элементов управления событий* через *архитектуру команд*. Обычное событие .NET определяется внутри некоторого базового класса и может использоваться только этим классом или его потомками. Таким образом, нормальные события .NET очень тесно связаны с классом, в котором они определены.

В отличие от этого, команды WPF представляют собой подобные событиям сущности, которые независимы от конкретного элемента управления и во многих случаях могут успешно применяться к многочисленным (и на вид несвязанным) типам элементов управления. Приведем несколько примеров: WPF поддерживает команды копирования, вырезания и вставки, которые могут применяться к широкому разнообразию элементов пользовательского интерфейса (пунктам меню, кнопкам панели инструментов, специальным кнопкам), а также клавиатурные комбинации (`<Ctrl+C>`, `<Ctrl+V>` и т.д.).

Хотя другие наборы инструментов для построения пользовательских интерфейсов (вроде Windows Forms) предлагают для этих целей стандартные события, в результате получается избыточный и трудный в сопровождении код. В рамках модели WPF команды могут использоваться в качестве альтернативы. В результате получается более компактная и гибкая кодовая база.

Объекты внутренних команд

WPF поставляется с множеством встроенных команд, каждая из которых может быть ассоциирована в соответствующей горячей клавишей (или другим источником ввода). С точки зрения программирования команда WPF — это любой объект, поддерживающий свойство (часто называемое `Command`), которое возвращает объект, реализующий интерфейс `ICommand`:

```
public interface ICommand
{
    // Возникает, когда происходит изменение режима,
    // должна или нет выполняться данная команда.
    event EventHandler CanExecuteChanged;
    // Определяет метод, выясняющий, может ли команда
    // выполняться в ее текущем состоянии.
    bool CanExecute(object parameter);
    // Определяет метод, предназначенный для вызова,
    // когда команда инициализирована.
    void Execute(object parameter);
}
```

WPF предлагает многочисленные классы команд, которые открывают около сотни готовых объектов команд. В этих классах определено множество свойств, представляющих специфические объекты команд, каждый из которых реализует интерфейс `ICommand`. В табл. 28.3 кратко описаны некоторые стандартные объекты команд (более подробную информацию ищите в документации .NET Framework 4.5 SDK).

Таблица 28.3. Стандартные объекты команд WPF

Класс WPF	Объекты команд	Описание
<code>ApplicationCommands</code>	<code>Close</code> , <code>Copy</code> , <code>Cut</code> , <code>Delete</code> , <code>Find</code> , <code>Open</code> , <code>Paste</code> , <code>Save</code> , <code>SaveAs</code> , <code>Redo</code> , <code>Undo</code>	Разнообразные команды уровня приложения
<code>ComponentCommands</code>	<code>MoveDown</code> , <code>MoveFocusBack</code> , <code>MoveLeft</code> , <code>MoveRight</code> , <code>ScrollToEnd</code> , <code>ScrollToHome</code>	Разнообразные команды, которые являются общими для компонентов пользовательского интерфейса
<code>MediaCommands</code>	<code>BoostBase</code> , <code>ChannelUp</code> , <code>ChannelDown</code> , <code>FastForward</code> , <code>NextTrack</code> , <code>Play</code> , <code>Rewind</code> , <code>Select</code> , <code>Stop</code>	Разнообразные команды, связанные с мультимедиа
<code>NavigationCommands</code>	<code>BrowseBack</code> , <code>BrowseForward</code> , <code>Favorites</code> , <code>LastPage</code> , <code>NextPage</code> , <code>Zoom</code>	Разнообразные команды, связанные с навигационной моделью WPF
<code>EditingCommands</code>	<code>AlignCenter</code> , <code>CorrectSpellingError</code> , <code>DecreaseFontSize</code> , <code>EnterLineBreak</code> , <code>EnterParagraphBreak</code> , <code>MoveDownByLine</code> , <code>MoveRightByWord</code>	Разнообразные команды, связанные с API-интерфейсом документов WPF

Подключение команд к свойству `Command`

Для подключения любого из свойств команд WPF к элементу пользовательского интерфейса, поддерживающему свойство `Command` (такому как `Button` или `MenuItem`), понадобится сделать совсем немного. Для примера модифицируем текущую систему меню, добавив новый пункт верхнего уровня по имени `Edit` (Правка) с тремя подэлементами, позволяющими копировать, вставлять и вырезать текстовые данные:

```
<Menu DockPanel.Dock ="Top"
    HorizontalAlignment="Left"
    Background="White" BorderBrush ="Black">
  <MenuItem Header="_File" Click ="FileExit_Click" >
    <MenuItem Header ="_Exit" MouseEnter ="MouseEnterExitArea"
      MouseLeave ="MouseLeaveArea" Click ="FileExit_Click"/>
  </MenuItem>

  <!-- Новые пункты меню с командами -->
  <MenuItem Header="_Edit">
    <MenuItem Command ="ApplicationCommands.Copy"/>
    <MenuItem Command ="ApplicationCommands.Cut"/>
    <MenuItem Command ="ApplicationCommands.Paste"/>
  </MenuItem>
```

```

<MenuItem Header="_ Tools">
  <MenuItem Header = "_ Spelling Hints"
    MouseEnter = "MouseEnterToolsHintsArea"
    MouseLeave = "MouseLeaveArea"
    Click = "ToolsSpellingHints_Click"/>
</MenuItem>
</Menu>

```

Обратите внимание, что свойству `Command` каждого подэлемента в меню `Edit` присвоено некоторое значение. За счет этого пункты меню автоматически получают корректные имена и горячие клавиши (например, `<Ctrl+C>` для операции вырезания) в интерфейсе меню, и приложение теперь знает, как *копировать*, *вырезать* и *вставлять*, без необходимости в написании процедурного кода.

Запустив приложение и выделив некоторую часть текста, сразу же можно пользоваться новыми пунктами меню. Вдобавок приложение также сможет реагировать на стандартную операцию щелчка правой кнопкой мыши, предлагая пользователю те же самые опции (рис. 28.21).

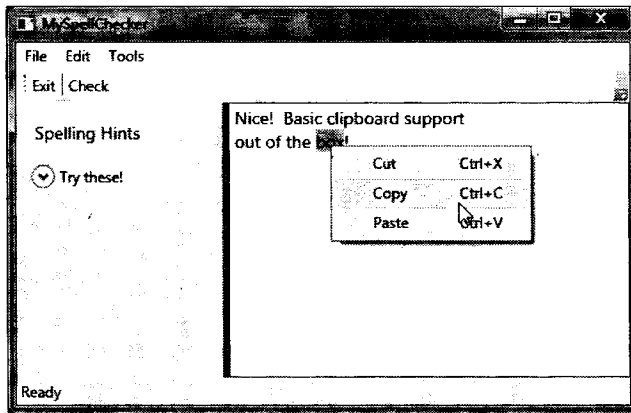


Рис. 28.21. Объекты команд предлагают полезный набор встроенной функциональности

Подключение команд к произвольным действиям

Если необходимо подключить объект команды к произвольному событию (специфичному для приложения), то для этого понадобится написать процедурный код. Это несложно, но требует немного больше логики, чем можно видеть в XAML. Например, предположим, что все окно должно реагировать на нажатие клавиши `<F1>`, активизируя связанную справочную систему. Кроме того, пусть в файле кода для главного окна определен новый метод по имени `SetF1CommandBinding()`, который вызывается в конструкторе после вызова `InitializeComponent()`:

```

public MainWindow()
{
    InitializeComponent();
    SetF1CommandBinding();
}

```

Этот новый метод программно создает новый объект `CommandBinding`, который можно использовать всякий раз, когда нужно привязать объект команды к заданному обработчику событий приложения. Сконфигурируем объект `CommandBinding` для работы с командой `ApplicationCommands.Help`, которая автоматически доступна по нажатию `<F1>`:

```
private void SetF1CommandBinding()
{
    CommandBinding helpBinding = new CommandBinding(ApplicationCommands.Help);
    helpBinding.CanExecute += CanHelpExecute;
    helpBinding.Executed += HelpExecuted;
    CommandBindings.Add(helpBinding);
}
```

Большинство объектов `CommandBinding` будет обрабатывать событие `CanExecute` (которое позволяет указать команду на основе операций программы) и событие `Executed` (в котором определяется то, что должно произойти в ответ на команду). Добавьте показанные ниже обработчики событий к производному от `Window` типу (обратите внимание на формат каждого метода, которого требуют ассоциированные делегаты):

```
private void CanHelpExecute(object sender, CanExecuteRoutedEventArgs e)
{
    // Если нужно предотвратить выполнение команды,
    // следует установить CanExecute в false.
    e.CanExecute = true;
}

private void HelpExecuted(object sender, ExecutedRoutedEventArgs e)
{
    MessageBox.Show("Look, it is not that difficult. Just type something!",
        "Help!");
}
```

В предыдущем фрагменте кода метод `CanHelpExecute()` реализован так, чтобы справка по нажатию `<F1>` всегда работала, для чего просто возвращается `true`. Если в каких-то ситуациях справочная система отображаться не должна, тогда нужно вернуть `false`. Наша "справочная система", отображаемая внутри `HelpExecuted()` — это всего лишь простое окно сообщения. Теперь можно запустить приложение. После нажатия `<F1>` отображается наша не слишком полезная справочная система (рис. 28.22).

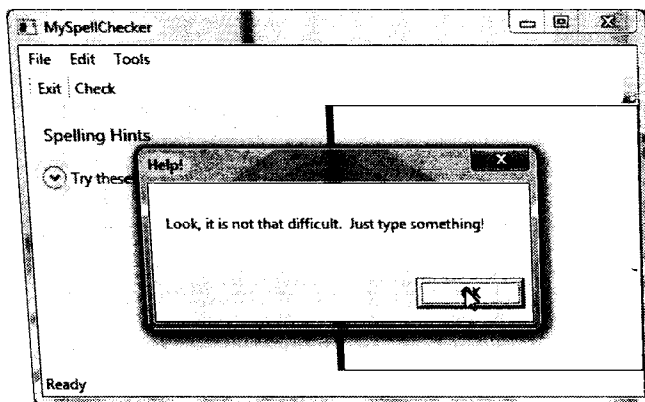


Рис. 28.22. Пример специальной справочной системы

Работа с командами `Open` и `Save`

Чтобы завершить текущий пример, добавим функциональность сохранения текстовых данных во внешнем файле и открытия файлов `*.txt` для редактирования. Можно пойти длинным путем, вручную добавив программную логику, которая включает и отключает пункты меню в зависимости от того, имеются ли данные внутри `TextBox`. Однако для сокращения затрат можно воспользоваться командами.

Начнем с обновления элемента `<MenuItem>`, который представляет меню `File` высшего уровня, добавив следующих два новых подменю, использующих объекты `Save` и `Open` класса `ApplicationCommands`:

```
<MenuItem Header="_File">
  <MenuItem Command="ApplicationCommands.Open"/>
  <MenuItem Command="ApplicationCommands.Save"/>
  <Separator/>
  <MenuItem Header="_Exit"
    MouseEnter="MouseEnterExitArea"
    MouseLeave="MouseLeaveArea" Click="FileExit_Click"/>
</MenuItem>
```

Как уже упоминалось, все объекты команд реализуют интерфейс `ICommand`, определяющий два события (`CanExecute` и `Executed`). Теперь нужно позволить окну выполнять эти команды.

Это делается наполнением коллекции `CommandBindings`, поддерживаемой окном. Чтобы сделать это в XAML, необходимо воспользоваться синтаксисом “свойство-элемент” для определения контекста `<Window.CommandBindings>`, куда будут помещены определения `<CommandBinding>`. Модифицируйте определение `<Window>`, как показано ниже:

```
<Window x:Class="MyWordPad.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="MySpellChecker" Height="331" Width="508"
  WindowStartupLocation="CenterScreen" >

  <!-- Это информирует Window, какие обработчики вызывать
        при проверке команд Open и Save -->
  <Window.CommandBindings>
    <CommandBinding Command="ApplicationCommands.Open"
      Executed="OpenCmdExecuted"
      CanExecute="OpenCmdCanExecute"/>
    <CommandBinding Command="ApplicationCommands.Save"
      Executed="SaveCmdExecuted"
      CanExecute="SaveCmdCanExecute"/>
  </Window.CommandBindings>

  <!-- Эта панель устанавливает содержимое окна -->
  <DockPanel>
    ...
  </DockPanel>
</Window>
```

Теперь щелкните правой кнопкой мыши на каждом из атрибутов `Executed` и `CanExecute` в редакторе XAML и выберите в контекстном меню пункт `Navigate to Event Handler` (Перейти к обработчику события). Как было описано в главе 27, это автоматически сгенерирует заготовку кода для обработчика события. Теперь в файле кода C# для окна имеются пустые обработчики событий.

Реализация обработчиков событий `CanExecute` должна сообщить окну, что можно инициировать соответствующие события `Executed` в любой момент, для чего понадобится установить свойство `CanExecute` входящего объекта `CanExecuteRoutedEventArgs`:

```
private void OpenCmdCanExecute(object sender, CanExecuteRoutedEventArgs e)
{
  e.CanExecute = true;
}
```

```
private void SaveCmdCanExecute(object sender, CanExecuteRoutedEventArgs e)
{
    e.CanExecute = true;
}
```

Соответствующие обработчики Executed выполняют действительную работу по отображению диалоговых окон открытия и сохранения файла; кроме того, они передают данные из TextBox в файл. Начните с импорта пространства имен System.IO и Microsoft.Win32 в файл кода. Готовый код прост и приведен ниже:

```
private void OpenCmdExecuted(object sender, ExecutedRoutedEventArgs e)
{
    // Создать диалоговое окно открытия файла, отображающее только текстовые файлы.
    OpenFileDialog openDlg = new OpenFileDialog();
    openDlg.Filter = "Text Files |*.txt";

    // Был ли совершен щелчок на кнопке OK?
    if (true == openDlg.ShowDialog())
    {
        // Загрузить содержимое выбранного файла.
        string dataFromFile = File.ReadAllText(openDlg.FileName);

        // Отобразить строку в TextBox.
        txtData.Text = dataFromFile;
    }
}

private void SaveCmdExecuted(object sender, ExecutedRoutedEventArgs e)
{
    SaveFileDialog saveDlg = new SaveFileDialog();
    saveDlg.Filter = "Text Files |*.txt";

    // Был ли совершен щелчок на кнопке OK?
    if (true == saveDlg.ShowDialog())
    {
        // Сохранить данные из TextBox в указанном файле.
        File.WriteAllText(saveDlg.FileName, txtData.Text);
    }
}
```

На этом пример и начальное знакомство с работой с элементами управления WPF завершены. Вы узнали, как работать с основными командами, системами меню, строками состояния, панелями инструментов, вложенными панелями и несколькими базовыми элементами пользовательского интерфейса, такими как TextBox и Expander. В следующем примере будут задействованы более экзотические элементы управления, и попутно вы ознакомитесь с несколькими важными службами WPF.

Исходный код. Проект MyWordPad доступен в подкаталоге Chapter 28.

Более глубокий взгляд на API-интерфейсы и элементы управления WPF

В оставшейся части этой главы будет строиться новое приложение WPF с использованием Visual Studio. Цель заключается в создании пользовательского интерфейса, который состоит их виджета TabControl, содержащего набор вкладок. Каждая вкладка будет иллюстрировать некоторые новые элементы управления WPF и интересные API-интерфейсы, которые могут применяться в разрабатываемых проектах. Попутно вы также изучите дополнительные функциональные возможности визуальных конструкторов WPF в Visual Studio.

Работа с элементом управления TabControl

Для начала создайте новый проект WPF Application по имени `WpfControlsAndAPIs`. Как уже упоминалось, начальное окно будет содержать виджет `TabControl` с четырьмя различными вкладками, каждая из которых отображает набор связанных элементов управления и/или API-интерфейсов WPF. Найдите элемент управления `TabControl` в панели инструментов Visual Studio, перетащите его на поверхность визуального конструктора, измените размеры компонента так, чтобы он занимал большую часть области отображения, и переименуйте этот элемент пользовательского интерфейса в `myTabSystem`.

Вы заметите, что два элемента типа вкладок предоставляются автоматически. Для добавления дополнительных вкладок нужно просто щелкнуть правой кнопкой мыши на узле `TabControl` в окне Document Outline и выбрать в контекстном меню пункт `Add TabItem` (Добавить `TabItem`); можно также щелкнуть правой кнопкой мыши на самом элементе `TabControl` в визуальном конструкторе и выбрать аналогичный пункт меню. Добавьте две дополнительные вкладки, используя любой из подходов (на рис. 28.23 показан подход с окном Document Outline).

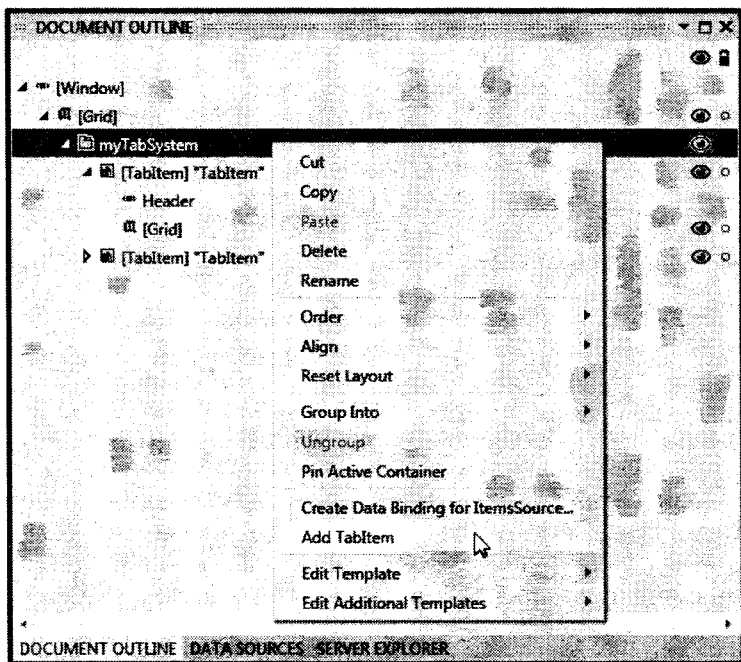


Рис. 28.23. Визуальное добавление элементов `TabItem`

Выберите каждый элемент управления `TabItem` (на поверхности визуального конструктора или в окне Document Outline) и измените его свойство `Header`, последовательно установив `Ink API`, `Documents`, `Data Binding` и `DataGrid`. После этого окно визуального конструктора должно выглядеть примерно так, как показано на рис. 28.24.

Теперь снова по очереди щелкните на каждой вкладке и с помощью окна `Properties` назначьте ей подходящее уникальное имя. Помните, что при выборе вкладки для редактирования эта вкладка становится активной, и ее содержимое можно компоновать, перетаскивая элементы управления из панели инструментов. Перед тем, как приступить к проектированию вкладок, давайте взглянем на разметку XAML, которую сгенерировала IDE-среда.

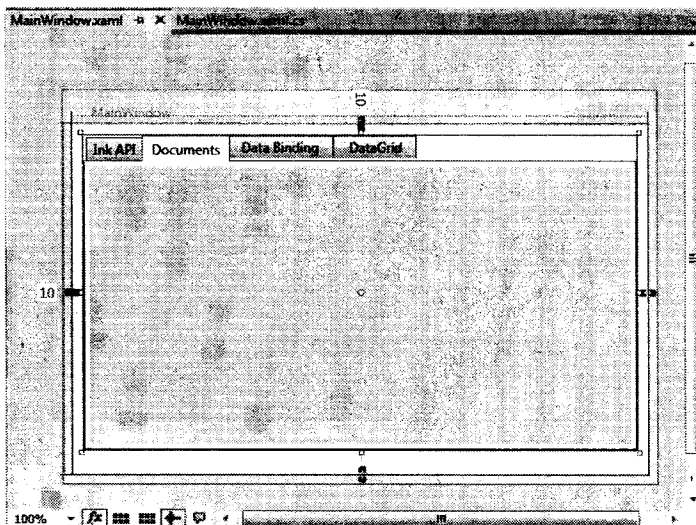


Рис. 28.24. Начальная компоновка системы вкладок

Разметка должна выглядеть похожей на приведенную ниже (отличаясь только установленными значениями свойств):

```
<TabControl x:Name="myTabSystem" HorizontalAlignment="Left" Height="280"
    Margin="10,10,0,0" VerticalAlignment="Top" Width="489">
    <TabItem Header="Ink API">
        <Grid Background="#FFE5E5E5"/>
    </TabItem>
    <TabItem Header="Documents">
        <Grid Background="#FFE5E5E5"/>
    </TabItem>
    <TabItem Header="Data Binding" HorizontalAlignment="Left" Height="20"
        VerticalAlignment="Top" Width="95" Margin="-2,-2,-36,0">
        <Grid Background="#FFE5E5E5"/>
    </TabItem>
    <TabItem Header="DataGrid" HorizontalAlignment="Left" Height="20"
        VerticalAlignment="Top" Width="74" Margin="-2,-2,-15,0">
        <Grid Background="#FFE5E5E5"/>
    </TabItem>
</TabControl>
```

Теперь, когда имеется определение основного элемента управления `TabControl`, можно работать с деталями каждой вкладки, по пути изучая дополнительные средства API-интерфейса WPF.

Построение вкладки Ink API

Первая вкладка раскрывает общее назначение интерфейса Ink API, который позволяет легко встраивать в программу функциональность рисования. Конечно, его применение не обязательно ограничивается приложениями для рисования; этот API-интерфейс можно использовать для решения широкого круга задач, включая фиксацию рукописного ввода посредством пера на Tablet PC.

Начните с нахождения в окне Document Outline узла, представляющего вкладку Ink API, и раскройте его. Вы должны увидеть, что стандартным диспетчером компоновки для этого элемента управления `TabItem` является `<Grid>`. Щелкните правой кнопкой мыши на нем и поменяйте диспетчер компоновки на `StackPanel` (рис. 28.25).

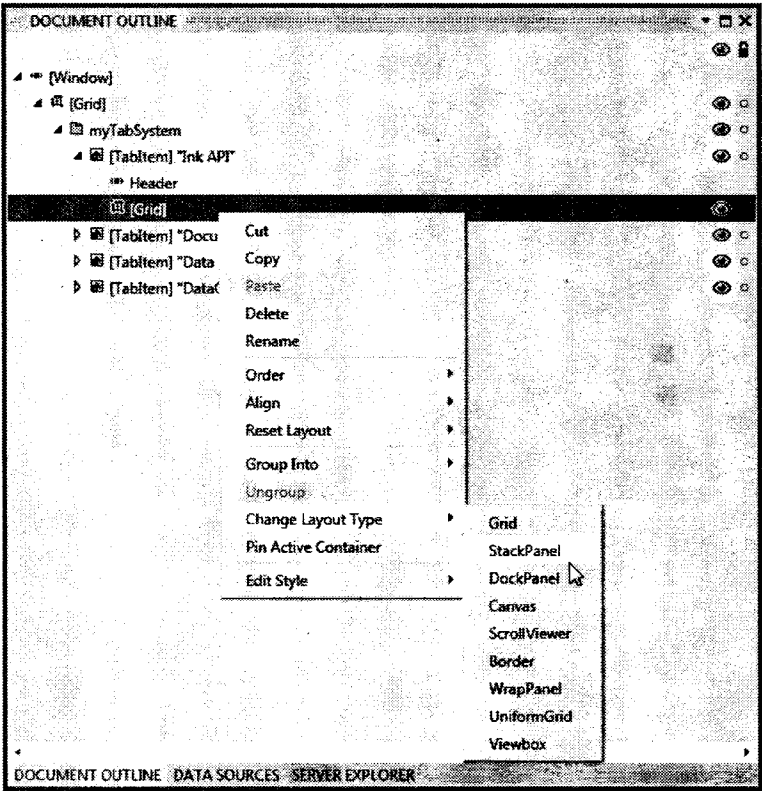


Рис. 28.25. Изменение диспетчера компоновки для первой вкладки

Проектирование панели инструментов

Удостоверьтесь, что узел StackPanel в данный момент выбран в редакторе Document Outline, и вставьте новый элемент управления ToolBar по имени inkToolbar. Выберите

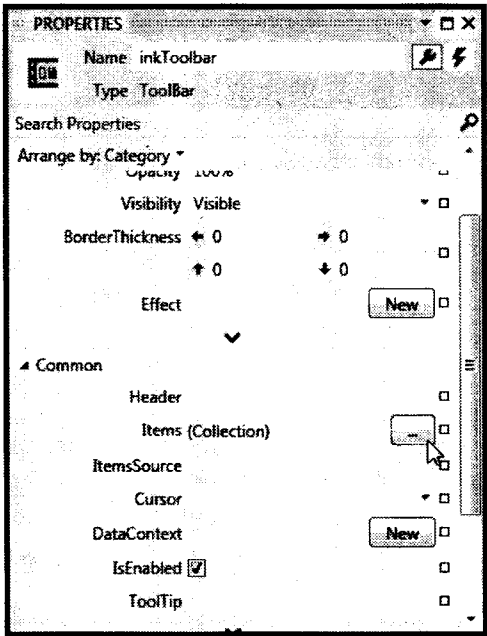


Рис. 28.26. Здесь начинается заполнение ToolBar элементами

узел inkToolbar для редактирования и установите свойство Height элемента Toolbar равным 60 (текущее значение свойства Width оставьте без изменений). Отыщите область Common (Общие) в окне Properties и щелкните на кнопке с многоточием возле свойства Items (Collection), как показано на рис. 28.26.

После щелчка на указанной кнопке откроется диалоговое окно, которое позволяет выбрать элементы управления для добавления в ToolBar. Щелкните на раскрывающемся списке в нижней центральной части диалогового окна и добавьте три элемента управления RadioButton. Воспользуйтесь встроенным редактором свойств в этом диалоговом окне, чтобы установить для каждого элемента RadioButton значение свойства Height в 50 и Width — в 100 (эти свойства находятся в области Layout (Компоновка)). Также последовательно установите свойство Content (в области Common) каждого элемента RadioButton в Ink Mode!, Erase Mode! и Select Mode!, соответственно (рис. 28.27).

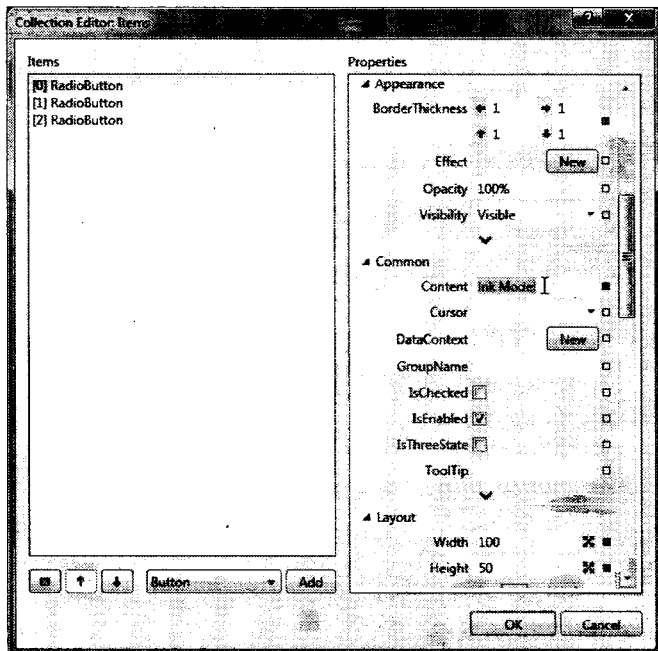


Рис. 28.27. Конфигурирование каждого элемента управления RadioButton

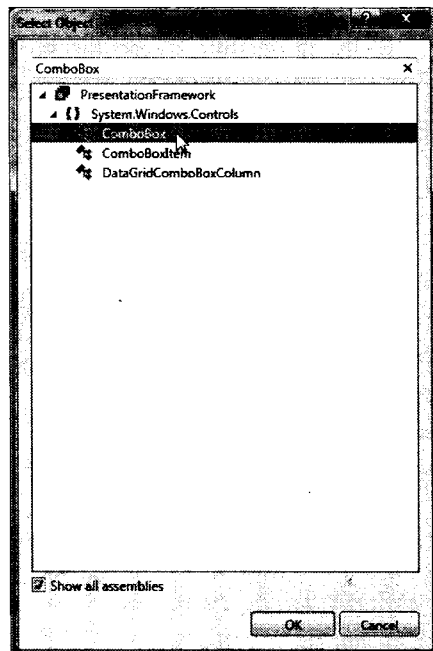


Рис. 28.28. Использование редактора Select Object для добавления уникальных элементов в панель инструментов

После добавления трех элементов управления RadioButton добавьте элемент Separator, используя раскрывающийся список в редакторе Items (Элементы). Теперь осталось добавить финальный элемент управления ComboBox (не ComboBoxItem). Когда необходимо вставить нестандартные элементы управления в редакторе Items, просто выберите в раскрывающемся списке вариант <Other Type...> (<Другой тип...>). Это приведет к открытию редактора Select Object (Выберите объект), в котором можно ввести имя нужного элемента управления. Удостоверьтесь, что флажок Show all assemblies (Показать все сборки) отмечен, и найдите интересующий элемент управления (рис. 28.28).

Установите свойство Width элемента ComboBox в 100 и добавьте в ComboBox три объекта ComboBoxItem, используя свойство Items (Collection) в области Common окна Properties. Установите свойства Content элементов ComboBoxItem в строки Red, Green и Blue.

После этого закройте редактор для возврата в визуальный конструктор окна. Последней задачей в настоящем разделе будет использование свойства Name для назначения имен переменных для новых элементов. Назовите три элемента RadioButton следующим образом: inkRadio, selectRadio и eraseRadio. Элементу ComboBox назначьте имя comboColors. Когда все указанное выше сделано, разметка XAML для первого элемента управления TabItem должна выглядеть следующим образом:

```
<TabItem Header="Ink API">
  <StackPanel Background="#FFE5E5">
    <ToolBar x:Name="inkToolBar" HorizontalAlignment="Left" Width="479" Height="60">
      <RadioButton x:Name="inkRadio" Content="Ink Mode!" Height="50" Width="100"/>
      <RadioButton x:Name="selectRadio" Content="Erase Mode!" Height="50" Width="100"/>
      <RadioButton x:Name="eraseRadio" Content="Select Mode!" Height="50" Width="100"/>
      <Separator/>
    </ToolBar>
  </StackPanel>
</TabItem>
```

```

<ComboBox x:Name="comboColors" Width="100">
  <ComboBoxItem Content="Red"/>
  <ComboBoxItem Content="Green"/>
  <ComboBoxItem Content="Blue"/>
</ComboBox>
</ToolBar>
</StackPanel>
</TabItem>

```

На заметку! При построении панели инструментов с помощью IDE-среды вы поймете, насколько быстрее все можно было бы сделать, если иметь возможность просто вручную редактировать XAML. Если вам удобно вводить код разметки непосредственно, можете так и поступать. Тем не менее, настоятельно рекомендуется уделить время на освоение окна **Properties** в Visual Studio. Вы увидите, что в нем предлагается несколько удобных расширенных возможностей.

Элемент управления RadioButton

В данном примере требуется, чтобы эти три элемента управления **RadioButton** были взаимно исключающими. В случае других платформ для построения графических пользовательских интерфейсов такие связанные элементы должны были бы помещаться в групповую рамку. В WPF так поступать не обязательно. Вместо этого элементам просто назначается одинаковое *групповое имя*. Это удобно, т.к. связанные элементы не обязательно должны быть физически собраны в одной области, а могут располагаться где угодно в окне.

Сделайте это, выбрав каждый элемент **RadioButton** в визуальном конструкторе (чтобы выбрать все три элемента, щелкайте на них при нажатой клавише <Shift>) и затем установив для свойства **GroupName** (находящегося в области **Common** окна **Properties**) значение **InkMode**.

Когда элемент управления **RadioButton** не размещается внутри родительского элемента-панели, он принимает вид, идентичный элементу управления **Button**! Однако в отличие от **Button**, класс **RadioButton** имеет свойство **IsChecked**, которое переключается между **true** и **false**, когда конечный пользователь щелкает на элементе. Вдобавок элемент управления **RadioButton** поддерживает два события (**Checked** и **Unchecked**), которые можно использовать для перехвата изменения этого состояния.

Для настройки элементов управления **RadioButton**, чтобы они выглядели как типичные переключатели, выберите их в визуальном конструкторе, последовательно щелкая при нажатой клавише <Shift>, а затем щелкните правой кнопкой мыши и выберите в контекстном меню пункт **Group Into⇒Border** (Сгруппировать в⇒Border), как показано на рис. 28.29.

Теперь все готово к тестированию программы, для чего понадобится нажать клавишу <F5>. Вы должны увидеть три взаимно исключающих переключателя и раскрывающийся список с тремя элементами (рис. 28.30).

Обработка событий для вкладки Ink API

Следующая задача для вкладки **Ink API** заключается в обработке события **Click** для каждого элемента управления **RadioButton**. Как вы делали это в других проектах WPF, просто щелкните на значке с изображением молнии в окне **Properties** среды Visual Studio и введите имена обработчиков событий. С помощью этого подхода свяжите событие **Click** всех элементов управления **RadioButton** с одним и тем же обработчиком по имени **RadioButtonClicked**.

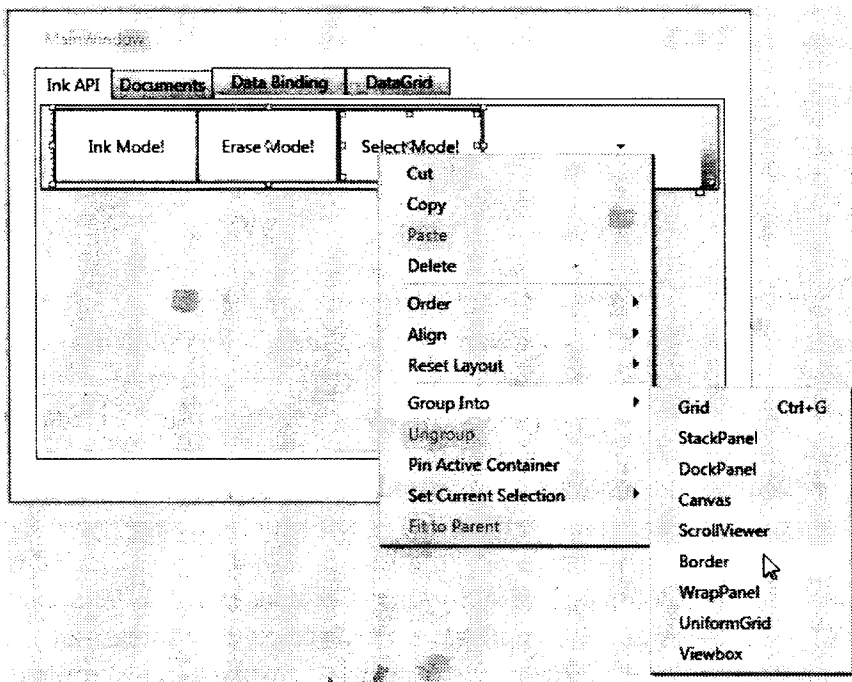


Рис. 28.29. Группирование элементов в элемент управления Border

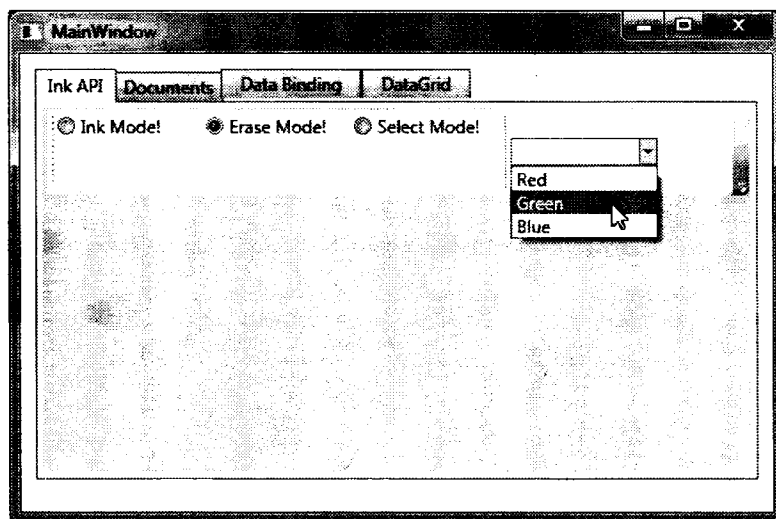


Рис. 28.30. Готовая система панели инструментов

После обработки событий Click сделайте это для событий SelectionChanged элемента управления ComboBox, используя обработчик по имени ColorChanged. В результате должен получиться следующий код C#:

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        this.InitializeComponent();
        // Добавьте код, необходимый при создании объекта.
    }
    private void RadioButtonClicked(object sender, System.Windows.RoutedEventArgs e)
    {
        // TODO: Добавьте сюда реализацию обработчика событий.
    }
}
```

```
private void ColorChanged(object sender,
    System.Windows.Controls.SelectionChangedEventArgs e)
{
    // TODO: Добавьте сюда реализацию обработчика событий.
}
}
```

Эти обработчики будут реализованы позже, так что пока оставьте их пустыми.

Элемент управления InkCanvas

Для завершения пользовательского интерфейса этой вкладки необходимо поместить элемент управления InkCanvas в StackPanel, чтобы он появился под только что созданным элементом ToolBar. К сожалению, по умолчанию панель инструментов Visual Studio не отображает все возможные компоненты WPF. Хотя можно было бы просто ввести необходимую разметку XAML, следует знать, что в действительности имеется возможность обновления элементов, отображаемых в панели инструментов.

Для этого щелкните правой кнопкой мыши где-нибудь в области панели инструментов и выберите из контекстного меню пункт Choose Items... (Выбрать элементы...). После этого появится список возможных компонентов для добавления в панель инструментов. Нас интересует добавление элемента управления InkCanvas (рис. 28.31).

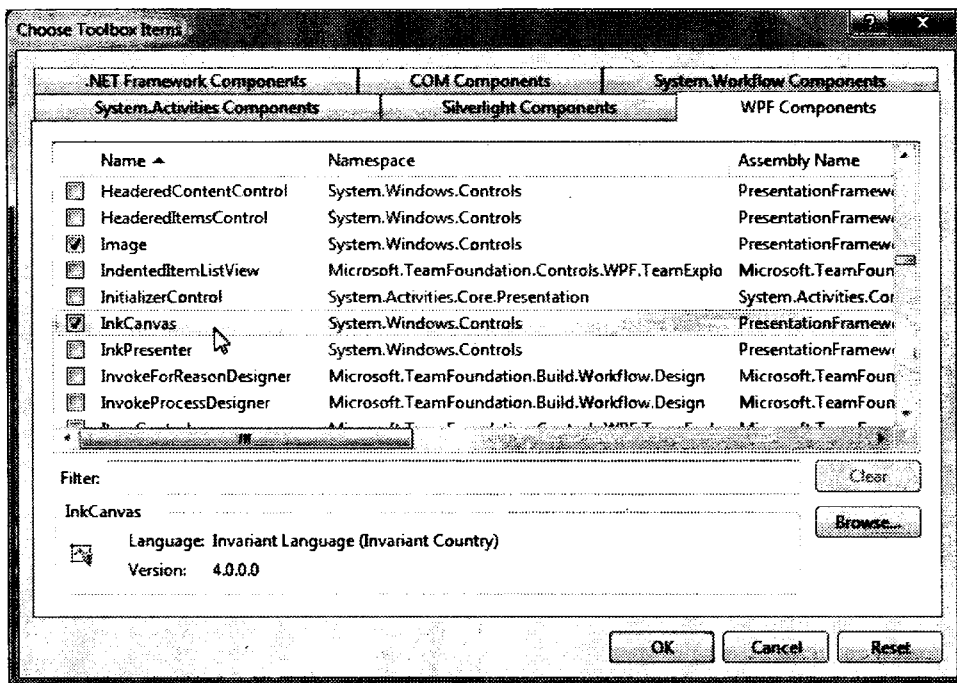


Рис. 28.31. Добавление новых компонентов в панель инструментов Visual Studio

Выберите StackPanel для объекта tabInk в окне Document Outline и добавьте элемент управления InkCanvas по имени myInkCanvas. Растяните этот новый элемент управления, чтобы он занял большую часть области вкладки. Кроме того, с помощью редактора Brushes (Кисти) можно задать для InkCanvas уникальный цвет фона (более подробно редактор Brushes рассматривается в следующей главе). После этого запустите программу, нажав <F5>. Вы увидите, что поверхность холста теперь позволяет рисовать графические данные за счет нажатия левой кнопки и перемещения мыши (рис. 28.32).

Элемент управления InkCanvas позволяет делать нечто большее, чем просто рисование линий мышью (или пером); он также поддерживает множество уникальных режимов редактирования, управляемых свойством EditingMode. Этому свойству можно присвоить любое значение из связанного перечисления InkCanvasEditingMode.

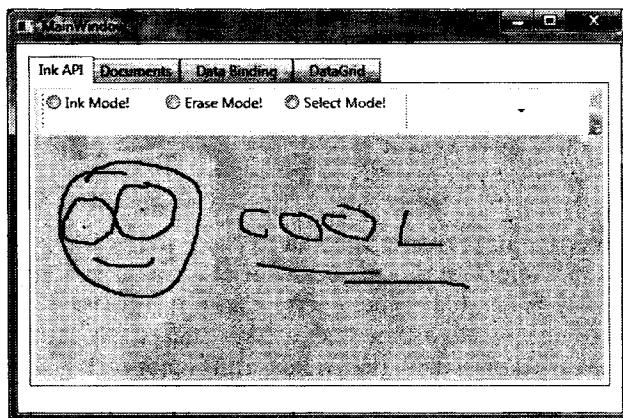


Рис. 28.32. Элемент управления InkCanvas в действии

В данном примере нас интересует режим Ink, принятый по умолчанию, который только что использовался; режим Select, позволяющий пользователю выбирать с помощью мыши область для последующего перемещения или изменения размера; и режим EraseByStroke, который удаляет предыдущий след, нарисованный мышью.

На заметку! Штрих — это визуализация, которая происходит при одиночной операции нажатия/отпускания кнопки мыши. InkCanvas сохраняет все штрихи в объекте StrokeCollection, который доступен через свойство Strokes.

Обновите обработчик RadioButtonClicked() следующей логикой, которая переводит InkCanvas в правильный режим в зависимости от выбранного переключателя RadioButton:

```
private void RadioButtonClicked(object sender,
    System.Windows.RoutedEventArgs e)
{
    // В зависимости от того, какая кнопка отправила событие,
    // переключить InkCanvas в нужный режим работы.
    switch((sender as RadioButton).Content.ToString())
    {
        // Эти строки должны совпадать со значениями Content
        // каждого элемента RadioButton.
        case "Ink Mode!":
            this.myInkCanvas.EditingMode = InkCanvasEditingMode.Ink;
            break;

        case "Erase Mode!":
            this.myInkCanvas.EditingMode = InkCanvasEditingMode.EraseByStroke;
            break;

        case "Select Mode!":
            this.myInkCanvas.EditingMode = InkCanvasEditingMode.Select;
            break;
    }
}
```

Также установите Ink в качестве стандартного режима в конструкторе окна. Там же установите стандартный выбор для ComboBox (более подробно этот элемент рассматривается в следующем разделе):


```

public MainWindow()
{
    this.InitializeComponent();
    // Установить режим Ink в качестве стандартного.
    this.myInkCanvas.EditingMode = InkCanvasEditingMode.Ink;
    this.inkRadio.IsChecked = true;
    this.comboColors.SelectedIndex=0;
}

```

Теперь снова запустите программу, нажав <F5>. Войдите в режим Ink и нарисуйте что-нибудь. Затем перейдите в режим Erase и сотрите ранее нарисованное (курсор мыши автоматически примет вид стирающей резинки). Наконец, переключитесь в режим Select, выберите несколько линий, используя мышь как лассо. Охватив элемент, вы сможете перемещать его по поверхности холста и изменять его размеры.

На рис. 28.33 показан результат работы в разных режимах.

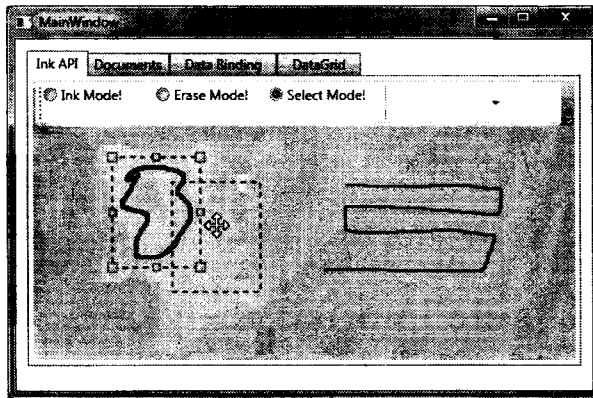


Рис. 28.33. Элемент InkCanvas в действии, с разными режимами редактирования

Элемент управления ComboBox

После заполнения элемента управления ComboBox (или ListBox) есть три способа определения выбранного в них элемента. Во-первых, когда необходимо найти числовой индекс выбранного элемента, должно использоваться свойство `SelectedIndex` (отсчет начинается с 0; значение -1 означает отсутствие выбора). Во-вторых, если требуется получить объект, выбранный внутри списка, подойдет свойство `SelectedItem`. В-третьих, `SelectedValue` позволяет получить значение выбранного объекта (обычно через вызов его метода `ToString()`).

И последний фрагмент кода, который понадобится добавить для данной вкладки, отвечает за изменение цвета линий, нарисованных в InkCanvas. Свойство `DefaultDrawingAttributes` элемента InkCanvas возвращает объект `DrawingAttributes`, который позволяет конфигурировать различные аспекты пера, включая его размер и цвет (помимо прочего). Модифицируйте код C# следующей реализацией метода `ColorChanged()`:

```

private void ColorChanged(object sender,
    System.Windows.Controls.SelectionChangedEventArgs e)
{
    // Получить выбранный элемент в раскрывающемся списке.
    string colorToUse =
        (this.comboColors.SelectedItem as ComboBoxItem).Content.ToString();
}

```

```
// Изменить цвет, используемый для визуализации линий.
this.myInkCanvas.DefaultDrawingAttributes.Color =
    (Color)ColorConverter.ConvertFromString(colorToUse);
}
```

Вспомните, что `ComboBox` содержит коллекцию `ComboBoxItems`. В сгенерированной разметке XAML имеется следующее определение:

```
<ComboBox x:Name="comboColors" Width="100" SelectionChanged="ColorChanged">
    <ComboBoxItem Content="Red"/>
    <ComboBoxItem Content="Green"/>
    <ComboBoxItem Content="Blue"/>
</ComboBox>
```

В результате вызова `SelectedItem` получается выбранный `ComboBoxItem`, который хранится как экземпляр общего типа `Object`. После приведения `Object` к `ComboBoxItem` получается значение `Content`, представленное строкой `Red`, `Green` или `Blue`. Эта строка затем преобразуется в объект `Color` с помощью удобного служебного класса `ColorConverter`. Снова запустите программу. Теперь появилась возможность переключать цвета при визуализации изображения.

Обратите внимание, что элементы управления `ComboBox` и `ListBox` могут также включать сложное содержимое, а не только списки текстовых данных. Чтобы получить представление о некоторых возможностях, откройте редактор XAML для окна и измените определение элемента управления `ComboBox`, чтобы он имел набор элементов `<StackPanel>`, каждый из которых содержит `<Ellipse>` и `<Label>` (значение свойства `Width` элемента `ComboBox` равно 200):

```
<ComboBox x:Name="comboColors" Width="200" SelectionChanged="ColorChanged">
    <StackPanel Orientation="Horizontal" Tag="Red">
        <Ellipse Fill="Red" Height="50" Width="50"/>
        <Label FontSize="20" HorizontalAlignment="Center"
            VerticalAlignment="Center" Content="Red"/>
    </StackPanel>
    <StackPanel Orientation="Horizontal" Tag="Green">
        <Ellipse Fill="Green" Height="50" Width="50"/>
        <Label FontSize="20" HorizontalAlignment="Center"
            VerticalAlignment="Center" Content="Green"/>
    </StackPanel>
    <StackPanel Orientation="Horizontal" Tag="Blue">
        <Ellipse Fill="Blue" Height="50" Width="50"/>
        <Label FontSize="20" HorizontalAlignment="Center"
            VerticalAlignment="Center" Content="Blue"/>
    </StackPanel>
</ComboBox>
```

Теперь свойству `Tag` каждого элемента `StackPanel` присвоено какое-то значение; это представляет собой быстрый и удобный способ определения стека элементов, выбранных пользователем (существуют и лучшие способы делать это, но пока такого достаточно). С этой поправкой потребуется изменить реализацию метода `ColorChanged()`, как показано ниже:

```
private void ColorChanged(object sender,
    System.Windows.Controls.SelectionChangedEventArgs e)
{
    // Получить свойство Tag выбранного элемента StackPanel.
    string colorToUse = (this.comboColors.SelectedItem
        as StackPanel).Tag.ToString();
    ...
}
```

Запустите программу и обратите внимание на уникальный элемент управления ComboBox (рис. 28.34).

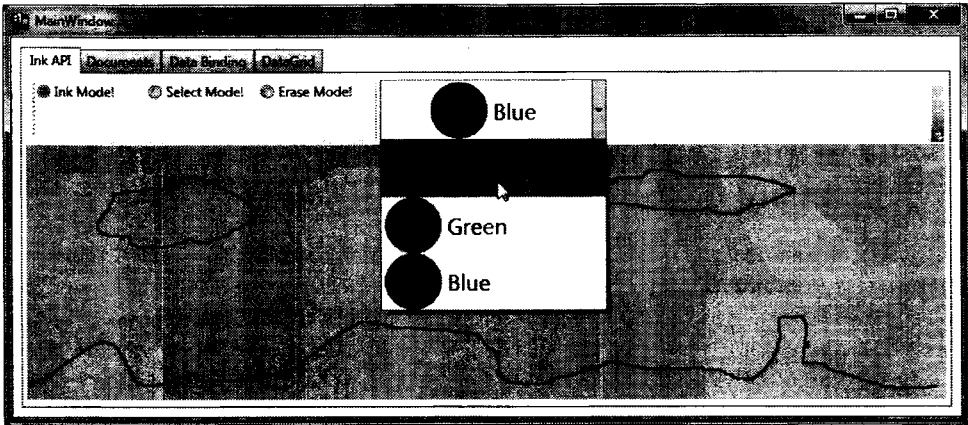


Рис. 28.34. Специальный элемент ComboBox, реализованный благодаря модели содержимого WPF

Сохранение, загрузка и очистка данных InkCanvas

Последняя часть этой вкладки позволит сохранять и загружать данные полотна, а также очищать его содержимое. К этому моменту вы уже должны довольно уверенно чувствовать себя в проектировании пользовательских интерфейсов, так что инструкции будут краткими и по существу.

Начните с импортирования пространств имен `System.IO` и `System.Windows.Ink` в файл кода. Затем добавьте в `ToolBar` три новых элемента управления `Button` с именами `btnSave`, `btnLoad` и `btnClear`. После этого обработайте событие `Click` для каждого элемента управления и реализуйте обработчики следующим образом:

```
private void SaveData(object sender, System.Windows.RoutedEventArgs e)
{
    // Сохранить все данные InkCanvas в локальном файле.
    using (FileStream fs = new FileStream("StrokeData.bin", FileMode.Create))
    {
        this.myInkCanvas.Strokes.Save(fs);
        fs.Close();
    }
}

private void LoadData(object sender, System.Windows.RoutedEventArgs e)
{
    // Наполнить StrokeCollection из файла.
    using (FileStream fs = new FileStream("StrokeData.bin",
        FileMode.Open, FileAccess.Read))
    {
        StrokeCollection strokes = new StrokeCollection(fs);
        this.myInkCanvas.Strokes = strokes;
    }
}

private void Clear(object sender, System.Windows.RoutedEventArgs e)
{
    // Очистить все штрихи.
    this.myInkCanvas.Strokes.Clear();
}
```

Теперь должна появиться возможность сохранения данных в файле, загрузки из файла и очистки InkCanvas от всех данных. На этом работа с первой вкладкой элемента управления TabControl завершена, равно как и исследования интерфейса Ink API. По правде говоря, об этой технологии можно рассказать еще много чего, однако теперь вы должны обладать достаточными знаниями, чтобы продолжить изучение этой темы самостоятельно. Далее мы приступаем к рассмотрению интерфейса Documents API в WPF.

Введение в интерфейс Documents API

Инфраструктура WPF поставляется с множеством элементов управления, которые позволяют вводить или отображать простые блоки текста, включая Label, TextBox, TextBlock и PasswordBox. Эти элементы управления удобны, но некоторые WPF-приложения требуют использования тщательно сформатированных текстовых данных, подобные тем, что можно найти в файле Adobe PDF. Интерфейс Documents API в WPF поддерживает такую функциональность, однако вместо формата файла PDF в нем используется формат XML Paper Specification (XPS).

Интерфейс Documents API можно применять для конструирования готового к печати документа, используя несколько классов из пространства имен System.Windows.Documents. В этом пространстве имен доступно множество типов, представляющих части документа XPS: List, Paragraph, Section, Table, LineBreak, Figure, Floater и Span.

Блочные элементы и встроенные элементы

Формально элементы, которые добавляются в документ XPS, относятся к одной из двух категорий: *блочные элементы* и *встроенные элементы*. Первая категория, блочные элементы, состоит из классов, которые расширяют базовый класс System.Windows.Documents.Block. Примерами блочных элементов могут служить List, Paragraph, BlockUIContainer, Section и Table. Классы из этой категории применяются для группирования вместе другого содержимого (например, список, содержащий данные абзаца, и абзац, содержащий подабзацы для разного форматирования текста).

Вторая категория, встроенные элементы, состоит из классов, расширяющих базовый класс System.Windows.Documents.Inline. Встроенные элементы вкладываются внутрь блочного элемента (или, возможно, внутрь другого встроенного элемента внутри блочного элемента). К общим встроенным элементам относятся Run, Span, LineBreak, Figure и Floater.

Эти классы имеют имена, которые можно встретить при построении форматированного документа в профессиональном редакторе. Как и любой другой элемент управления WPF, эти классы можно конфигурировать в XAML-разметке или в коде. Таким образом, можно либо объявить пустой элемент <Paragraph>, который наполняется во время выполнения (в следующем примере будет показано, как это делается), либо определить заполненный элемент <Paragraph> со статическим текстом.

Диспетчеры компоновки документа

Может показаться, что встроенные и блочные элементы следует размещать непосредственно в контейнере типа панели, таком как Grid, но на самом деле они должны быть упакованы в элементы <FlowDocument> или <FixedDocument>.

Помещать элементы в FlowDocument идеально, когда нужно позволить конечному пользователю изменять способ представления данных. Пользователь может изменять масштаб текста или способ представления данных (например, в виде одной длинной страницы или в виде пары столбцов). FixedDocument лучше применять для представления готовых к печати (WYSIWYG), неизменяемых документов.

В рассматриваемом примере мы будем иметь дело только с контейнером FlowDocument. После вставки встроенных и блочных элементов в FlowDocument этот объект будет помещен в один из поддерживающих XPS диспетчеров компоновки, которые перечислены в табл. 28.4.

Таблица 28.4. Диспетчеры компоновки XPS

Элемент управления типа панели	Описание
FlowDocumentReader	Отображает данные в FlowDocument и добавляет поддержку масштабирования, поиска и компоновки содержимого в различных формах
FlowDocumentScrollViewer	Отображает данные в FlowDocument; однако данные представлены как единый документ, просматриваемый с использованием линейек прокрутки. Этот контейнер не поддерживает масштабирование, поиск или альтернативные режимы компоновки
RichTextBox	Отображает данные в FlowDocument и добавляет поддержку редактирования со стороны пользователей
FlowDocumentPageViewer	Отображает документ постранично, т.е. одну страницу за раз. Данные можно масштабировать, но поиск не предусмотрен

Наиболее полнофункциональный способ отображения FlowDocument состоит в том, чтобы поместить его внутрь диспетчера FlowDocumentReader. В результате пользователь получает возможность изменять компоновку, искать слова в документе и масштабировать отображение данных. Одно из ограничений этого контейнера (а также FlowDocumentScrollViewer и FlowDocumentPageViewer) связано с тем, что отображаемое содержимое доступно только для чтения. Если же необходимо позволить конечному пользователю вводить новую информацию в FlowDocument, его можно упаковать в элемент управления RichTextBox.

Построение вкладки Documents

Щелкните на вкладке Documents элемента TabItem и откройте ее в визуальном конструкторе для редактирования. Здесь уже имеется стандартный элемент управления <Grid>, который является прямым дочерним элементом TabItem; с помощью окна Document Outline замените его панелью StackPanel. На вкладке Documents будет отображаться элемент FlowDocument, который позволит выделять текст и добавлять аннотации, используя API-интерфейс Sticky Notes (“клеякие” заметки).

Начните с определения следующего элемента управления ToolBar, который включает в себя три простых (неименованных) элемента управления Button. Позднее к этим элементам управления будут привязаны команды, поэтому ссылаться на них в коде не понадобится (можете либо ввести код разметки XAML вручную, либо воспользоваться IDE-средой).

```
<TabItem x:Name="tabDocuments" Header="Documents"
        VerticalAlignment="Bottom" Height="20">
  <StackPanel>
    <ToolBar>
      <Button BorderBrush="Green" Content="Add Sticky Note"/>
      <Button BorderBrush="Green" Content="Delete Sticky Notes"/>
      <Button BorderBrush="Green" Content="Highlight Text"/>
    </ToolBar>
  </StackPanel>
</TabItem>
```

При желании можно модифицировать панель инструментов Visual Studio, включив в нее элемент управления `FlowDocumentReader` (с применением того же самого приема, что и в случае `InkCanvas`), или обновить текущую разметку `TabItem` вручную с помощью редактора XAML.

В любом случае добавьте элемент `FlowDocumentReader` в `StackPanel`, переименуйте его в `myDocumentReader` и растяните на всю поверхность `StackPanel`. Добавьте в этот компонент пустой элемент `<FlowDocument>`.

```
<FlowDocumentReader x:Name="myDocumentReader" Height="269.4">
  <FlowDocument/>
</FlowDocumentReader>
```

Теперь можно добавлять к элементу `<FlowDocument>` классы документа (например, `List`, `Paragraph`, `Section`, `Table`, `LineBreak`, `Figure`, `Floater` и `Span`). Вот один из возможных способов конфигурирования `FlowDocument`:

```
<FlowDocumentReader x:Name="myDocumentReader" Height="269.4">
  <FlowDocument>
    <Section Foreground = "Yellow" Background = "Black">
      <Paragraph FontSize = "20">
        Here are some fun facts about the WPF Documents API!
      </Paragraph>
    </Section>
    <List/>
    <Paragraph/>
  </FlowDocument>
</FlowDocumentReader>
```

Если теперь запустить программу (нажав `<F5>`), можно масштабировать отображение документа (используя ползунок в нижнем правом углу), искать ключевые слова (с помощью редактора поиска, расположенного внизу слева) и отображать данные в одном из трех режимов (применяя кнопки компоновки).

Прежде чем приступить к следующему шагу, отредактируйте XAML-разметку, чтобы вместо `FlowDocumentReader` использовался другой контейнер `FlowDocument`, например, такой как `FlowDocumentScrollViewer` или `RichTextBox`. После этого снова запустите приложение и обратите внимание на отличия в обработке данных документа. По завершении этого эксперимента восстановите применение типа `FlowDocumentReader`.

Наполнение `FlowDocument` с помощью кода

Теперь давайте построим блок `List` и оставшийся блок `Paragraph` в коде. В этом важно разобраться, поскольку часто требуется наполнять документ `FlowDocument` на основе пользовательского ввода, внешних файлов, информации из базы данных или любого другого источника. Для начала воспользуйтесь редактором XAML, чтобы назначить элементам `List` и `Paragraph` подходящие имена и тем самым получить возможность обращаться к ним в коде:

```
<List x:Name="listOfFunFacts"/>
<Paragraph x:Name="paraBodyText"/>
```

В файле кода определите новый закрытый метод по имени `PopulateDocument()`. Этот метод сначала добавляет набор элементов `ListItem` к `List`, каждый из которых имеет элемент `Paragraph` с единственным элементом `Run`. Кроме того, этот вспомогательный метод динамически строит сформатированный абзац, используя три отдельных объекта `Run`, как показано ниже:

```
private void PopulateDocument()
{
```

```

// Добавить некоторые данные в элемент List.
this.listOfFunFacts.FontSize = 14;
this.listOfFunFacts.MarkerStyle = TextMarkerStyle.Circle;
this.listOfFunFacts.ListItems.Add(new ListItem( new
    Paragraph(new Run("Fixed documents are for WYSIWYG print ready docs!"))));
this.listOfFunFacts.ListItems.Add(new ListItem(
    new Paragraph(new Run("The API supports tables and embedded figures!"))));
this.listOfFunFacts.ListItems.Add(new ListItem(
    new Paragraph(new Run("Flow documents are read only!"))));
this.listOfFunFacts.ListItems.Add(new ListItem(new Paragraph(new Run
    ("BlockUIContainer allows you to embed WPF controls in the document!"))));

// Добавить некоторые данные в элемент Paragraph.
// Первая часть абзаца.
Run prefix = new Run("This paragraph was generated ");

// Середина абзаца.
Bold b = new Bold();
Run infix = new Run("dynamically");
infix.Foreground = Brushes.Red;
infix.FontSize = 30;
b.Inlines.Add(infix);

// Последняя часть абзаца.
Run suffix = new Run(" at runtime!");

// Добавить все части в коллекцию встроенных элементов Paragraph.
this.paraBodyText.Inlines.Add(prefix);
this.paraBodyText.Inlines.Add(infix);
this.paraBodyText.Inlines.Add(suffix);
}

```

Обеспечьте вызов этого метода в конструкторе окна. После этого запустите приложение и обратите внимание на новое, динамически сгенерированное содержимое документа.

Включение аннотаций и “клеяких” заметок

Теперь можно строить документ с интересными данными, используя XAML-разметку и код C#, однако нужно еще что-то сделать с тремя кнопками в панели инструментов на вкладке Documents. В составе WPF доступен набор команд, предназначенных для применения специально с интерфейсом Documents API. Эти команды позволяют пользователю выделять часть документа, а также добавлять “клеякие” заметки — аннотации. Все это делается с помощью всего нескольких строк кода (и небольшого объема разметки).

Объекты команд для Documents API находятся в пространстве имен System.Windows.Annotations сборки PresentationFramework.dll. Таким образом, понадобится определить специальное пространство имен XML в открывающем элементе <Window> для использования таких объектов в XAML (обратите внимание, что префиксом дескриптора является a):

```

<Window
...
xmlns:a=
    "clr-namespace:System.Windows.Annotations;assembly=PresentationFramework"
x:Class="WpfControlsAndAPIs.MainWindow"
x:Name="Window"
Title="MainWindow"
Width="856" Height="383" mc:Ignorable="d" WindowStartupLocation="CenterScreen">
...
</Window>

```

Модифицируйте три определения `<Button>`, установив свойства `Command` в соответствующие команды аннотаций:

```
<ToolBar>
  <Button BorderBrush="Green" Content="Add Sticky Note"
    Command="a:AnnotationService.CreateTextStickyNoteCommand"/>
  <Button BorderBrush="Green" Content="Delete Sticky Notes"
    Command="a:AnnotationService.DeleteStickyNotesCommand"/>
  <Button BorderBrush="Green" Content="Highlight Text"
    Command="a:AnnotationService.CreateHighlightCommand"/>
</ToolBar>
```

Последнее, что потребуется делать — это включить службы аннотации для объекта `FlowDocumentReader`, который был назван `myDocumentReader`. Добавьте в класс еще один закрытый метод по имени `EnableAnnotations()`, который будет вызываться в конструкторе окна. Затем импортируйте следующие пространства имен:

```
using System.Windows.Annotations;
using System.Windows.Annotations.Storage;
```

Теперь реализуйте этот метод:

```
private void EnableAnnotations()
{
    // Создать объект AnnotationService, работающий с FlowDocumentReader.
    AnnotationService anoService = new AnnotationService(myDocumentReader);

    // Создать объект MemoryStream, который будет содержать аннотации.
    MemoryStream anoStream = new MemoryStream();

    // Создать основанное на XML хранилище на основе MemoryStream.
    // Этот объект можно использовать для программного добавления,
    // удаления или поиска аннотаций.
    AnnotationStore store = new XmlStreamStore(anoStream);

    // Включить службы аннотаций.
    anoService.Enable(store);
}
```

Класс `AnnotationService` позволяет заданному диспетчеру компоновки документа получить поддержку аннотаций. Прежде чем вызывать метод `Enable()` этого объекта, необходимо предоставить местоположение объекта для хранения аннотированных данных, которые в данном примере являются областью памяти, представленной объектом `MemoryStream`. Обратите внимание, что объект `AnnotationService` соединяется с объектом `Stream`, используя `AnnotationStore`.

Запустите приложение. Выделите некоторый текст, щелкните на кнопке `Add Sticky Note` (Добавить “клеякую” заметку) и введите некоторую информацию. Выделенный текст также можно подсвечивать (по умолчанию желтым цветом). Наконец, можно удалять заметки, выбирая их и щелкая на кнопке `Delete Stricky Note` (Удалить “клеякую” заметку). На рис. 28.35 показан результат тестового запуска.

Сохранение и загрузка потокового документа

Экскурс в интерфейс `Documents API` завершается рассмотрением простого процесса сохранения документа в файл, а также чтения документа из файла. Вспомните, что если объект `FlowDocument` не упакован в `RichTextBox`, конечный пользователь не сможет редактировать документ; кроме того, часть документа создавалась динамически во время выполнения, так что может понадобиться сохранить его для последующего использования. Возможность сохранения документа в стиле XPS также может быть полезна во многих приложениях WPF, т.к. она позволяет определить пустой документ и загружать его на лету.

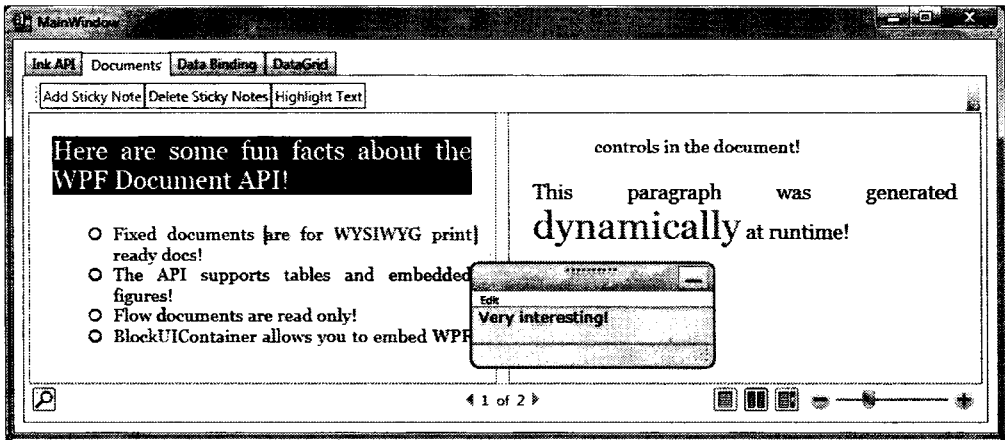


Рис. 28.35. Использование "клеяких" заметок

В следующем фрагменте разметки предполагается, что в панель инструментов вкладки Documents были добавлены два новых элемента Button, объявленные следующим образом (обратите внимание, что в разметке никакие события не обрабатываются):

```
<Button x:Name="btnSaveDoc" HorizontalAlignment="Stretch"
        VerticalAlignment="Stretch" Width="75" Content="Save Doc"/>
<Button x:Name="btnLoadDoc" HorizontalAlignment="Stretch"
        VerticalAlignment="Stretch" Width="75" Content="Load Doc"/>
```

В конструкторе окна напишите следующие лямбда-выражения для сохранения и загрузки данных FlowDocument (для получения доступа к классам XamlReader и XamlWriter понадобится импортировать пространство имен System.Windows.Markup):

```
public MainWindow()
{
    ...
    // Построить обработчики событий Click для сохранения и загрузки
    // документа нефиксированного формата.
    btnSaveDoc.Click += (o, s) =>
    {
        using(FileStream fStream = File.Open(
            "documentData.xaml", FileMode.Create))
        {
            XamlWriter.Save(this.myDocumentReader.Document, fStream);
        }
    };
    btnLoadDoc.Click += (o, s) =>
    {
        using(FileStream fStream = File.Open("documentData.xaml", FileMode.Open))
        {
            try
            {
                FlowDocument doc = XamlReader.Load(fStream) as FlowDocument;
                this.myDocumentReader.Document = doc;
            }
            catch(Exception ex) {MessageBox.Show(ex.Message, "Error Loading Doc!");}
        }
    };
}
```

Вот и все, что требуется сделать для сохранения документа (следует отметить, что при этом не сохраняются аннотации; тем не менее, это можно сделать с помощью служб аннотаций). После щелчка на кнопке Save Doc (Сохранить документ) в папке `\bin\Debug` появится новый файл `*.xaml`, который содержит данные текущего документа.

На этом рассмотрение интерфейса Documents API завершено. В данном API-интерфейсе есть еще много такого, что здесь не было показано, но вы получили достаточное представление об основах. В конце главы мы затронем нескольких тем, касающихся привязки данных, и завершим текущее приложение.

Введение в модель привязки данных WPF

Элементы управления часто служат целью для различных операций привязки данных. Выражаясь просто, *привязка данных* — это акт подключения свойств элемента управления к значениям данных, которые могут изменяться на протяжении жизненного цикла приложения. Это позволяет элементу пользовательского интерфейса отображать состояние переменной в коде. Например, привязку данных можно использовать для выполнения следующих действий:

- отмечать флажок элемента управления `CheckBox` на основе булевского свойства заданного объекта;
- отображать в элементах `TextBox` информацию, извлеченную из реляционной базы данных;
- подключать элемент `Label` к целому числу, представляющему количество файлов в папке.

При использовании встроенного в WPF механизма привязки данных следует помнить о разнице между *источником* и *местом назначения* операции привязки. Как и можно было ожидать, источником операции привязки данных являются сами данные (булевское свойство, реляционные данные и т.п.), в то время как местом назначения (или целью) является свойство элемента управления пользовательского интерфейса, которое использует содержимое данных (например, свойство элемента управления `CheckBox` или `TextBox`).

По правде говоря, применение инфраструктуры привязки данных WPF никогда не является обязательным. Если разработчик пожелает самостоятельно реализовать собственную логику привязки данных, то подключение между источником и местом назначения обычно потребует обработки различных событий и написания процедурного кода для соединения с источником и целью. Например, если в окне имеется элемент `ScrollBar`, который должен отобразить свое значение внутри `Label`, можно обработать событие `ValueChanged` элемента `ScrollBar` и соответствующим образом обновить содержимое `Label`.

Однако привязку данных WPF можно применять для соединения источника и цели непосредственно в разметке XAML (или в файле кода C#) без необходимости в обработке различных событий или жесткого кодирования соединений между источником и целью. К тому же, на основе настройки логики привязки данных можно обеспечить постоянную синхронизацию целевого объекта с изменяющимися значениями данных.

Построение вкладки Data Binding

С помощью окна Document Outline замените `Grid` в третьей вкладке контейнером `StackPanel`. Затем воспользуйтесь панелью инструментов и окном Properties среды Visual Studio для построения следующей начальной компоновки:

```

<TabItem x:Name="tabDataBinding" Header="Data Binding">
  <StackPanel Width="250">
    <Label Content="Move the scroll bar to see the current value"/>
    <!-- Значение линейки прокрутки является источником привязки данных -->
    <ScrollBar x:Name="mySB" Orientation="Horizontal" Height="30"
      Minimum = "1" Maximum = "100" LargeChange="1" SmallChange="1"/>
    <!-- Содержимое Label будет привязано к линейке прокрутки -->
    <Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue"
      BorderThickness="2" Content = "0"/>
  </StackPanel>
</TabItem>

```

Обратите внимание, что объект `<ScrollBar>` (названный здесь `mySB`) сконфигурирован с диапазоном от 1 до 100. Цель заключается в том, чтобы при изменении положения ползунка линейки прокрутки (или по щелчку на стрелке влево или вправо) элемент `Label` автоматически обновлялся текущим значением. В настоящий момент значение свойства `Content` элемента управления `Label` установлено в "0"; тем не менее, мы изменим это через операцию привязки данных.

Установка привязки данных с использованием Visual Studio

Механизм, обеспечивающий определение привязки в XAML — это расширение разметки (`Binding`). Установка привязки между элементами управления в Visual Studio осуществляется легко. В рассматриваемом примере найдите свойство `Content` объекта `Label` (в области `Common` окна `Properties`) и щелкните на маленьком квадрате рядом со свойством для открытия контекстного меню. Выберите в нем пункт `Create Data Binding...` (Создать привязку данных...), как показано на рис. 28.36.

Далее выберите вариант `ElementName` в раскрывающемся списке `Binding type` (Тип привязки), что приведет к выдаче списка всех элементов в вашем файле XAML, которые могут быть выбраны в качестве источника операции привязки данных. В дереве `Element name` (Имя элемента) найдите объект `ScrollBar` (по имени `mySB`). В дереве `Path` (Путь) отыщите свойство `Value` (рис. 28.37). Щелкните на кнопке `OK`.

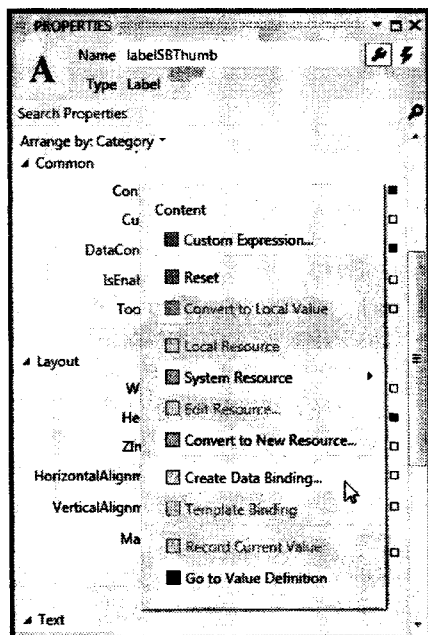


Рис. 28.36. Конфигурирование операции привязки данных

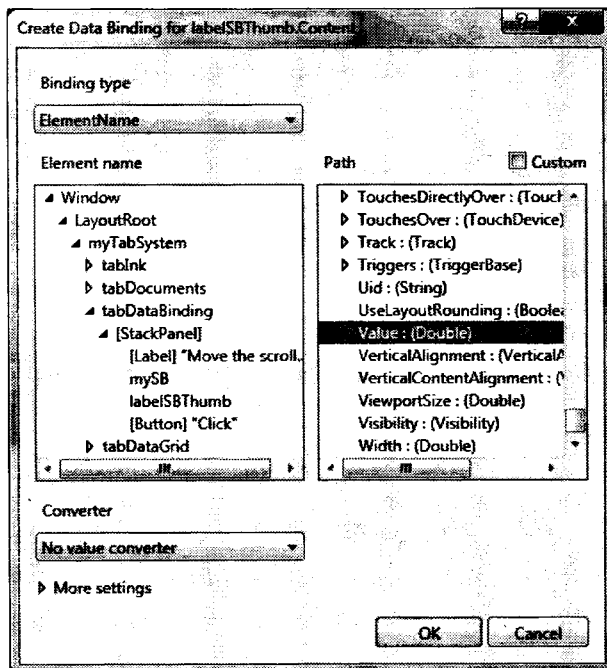


Рис. 28.37. Выбор объекта-источника и его свойства

Запустив программу, вы обнаружите, что содержимое метки обновляется при перемещении ползунка. Теперь взглянем на разметку XAML, сгенерированную инструментом привязки данных:

```
<Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue" BorderThickness="2"
      Content = "{Binding Value, ElementName=mySB}"/>
```

Обратите внимание на значение, присвоенное свойству Content элемента Label. Значение ElementName здесь представляет источник операции привязки данных (объект ScrollBar), а первый элемент после ключевого слова Binding (т.е. Value) представляет (в этом случае) свойство элемента, который необходимо получить.

Если вам приходилось ранее работать с привязкой данных WPF, то вы можете ожидать увидеть использование лексемы Path для установки наблюдаемого свойства объекта. Например, следующая разметка будет также корректно обновлять Label:

```
<Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue"
      BorderThickness="2" Content = "{Binding Path=Value, ElementName=mySB }"/>
```

По умолчанию аспект Path= операции привязки данных опускается, если только свойство не является подсвойством другого объекта (например, myObject.MyProperty.Object2.Property2).

Свойство DataContext

Для определения операции привязки данных в XAML может применяться альтернативный формат, при котором допускается разбивать значения, указанные расширением разметки {Binding}, за счет явной установки свойства DataContext в источник операции привязки:

```
<!-- Разбиение пары "объект/значение" через DataContext -->
<Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue"
      BorderThickness="2"
      DataContext = "{Binding ElementName=mySB}"
      Content = "{Binding Path=Value}" />
```

В этом примере вывод будет идентичным. С учетом этого может возникнуть вопрос: когда необходимо устанавливать свойство DataContext явно? Это полезно в ситуации, когда подэлементы наследуют свое значение в дереве разметки.

Подобным образом можно легко устанавливать один и тот же источник данных для семейства элементов управления, вместо того, чтобы повторять избыточные XAML-фрагменты "{Binding ElementName=X, Path=Y}" во множестве элементов управления. Например, предположим, что в контейнер <StackPanel> этой вкладки добавлен новый элемент Button:

```
<Button Content="Click" Height="140"/>
```

Для генерации привязок данных к множеству элементов управления можно было бы воспользоваться Visual Studio, но вместо этого давайте попробуем ввести модифицированную разметку в редакторе XAML:

```
<!-- Обратите внимание, что StackPanel устанавливает свойство DataContext -->
<StackPanel Width="250" DataContext = "{Binding ElementName=mySB}">
  <Label Content="Move the scroll bar to see the current value"/>
  <ScrollBar Orientation="Horizontal" Height="30" Name="mySB"
    Maximum = "100" LargeChange="1" SmallChange="1"/>

  <!-- Теперь оба элемента пользовательского интерфейса работают
    со значением линейки прокрутки уникальным образом -->
  <Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue" BorderThickness="2"
    Content = "{Binding Path=Value}"/>
```

```

<Button Content="Click" Height="200"
        FontSize = "{Binding Path=Value}"/>
</StackPanel>

```

Здесь свойство `DataContext` в `<StackPanel>` устанавливается напрямую. В результате при перемещении ползунка не только отображается текущее значение в элементе `Label`, но также увеличивается размер шрифта элемента `Button` в соответствии с тем же значением. На рис. 28.38 показан возможный вывод.

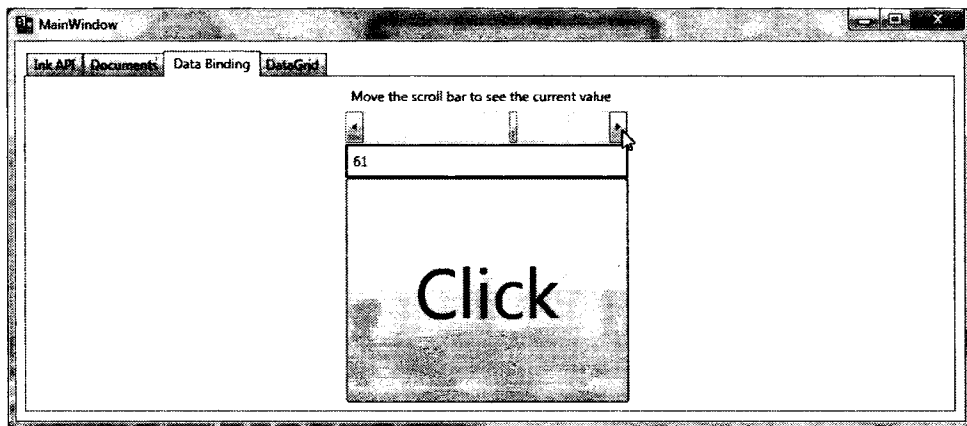


Рис. 28.38. Привязка значения `ScrollBar` к `Label` и `Button`

Преобразование данных с использованием `IValueConverter`

Вместо ожидаемого целого числа для представления положения ползунка тип `ScrollBar` использует значение `double`. Поэтому при перемещении ползунка в элементе `Label` будут отображаться различные значения с плавающей точкой (вроде 61.0576923076923), которые выглядят не слишком интуитивно понятными для конечного пользователя, который ожидает целые числа (такие как 61, 62, 63 и т.д.).

Одним из возможных способов преобразования значения из операции привязки данных в альтернативный формат является создание специального класса, реализующего интерфейс `IValueConverter`, доступный в пространстве имен `System.Windows.Data`. В этом интерфейсе определены два члена, которые позволяют осуществлять преобразование между источником и целью (в случае двунаправленной привязки). После определения такой класс можно применять для последующего уточнения процесса привязки данных.

Исходя из того, что в элементе управления `Label` необходимо отображать целые числа, можно построить показанный ниже специальный класс преобразования. Выберите пункт меню `Project⇒Add Class` (Проект⇒Добавить класс) и добавьте класс по имени `MyDoubleConverter`. Затем поместите в него следующий код:

```

class MyDoubleConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter,
        System.Globalization.CultureInfo culture)
    {
        // Преобразовать значение double в int.
        double v = (double)value;
        return (int)v;
    }
}

```

```

public object ConvertBack(object value, Type targetType, object parameter,
                          System.Globalization.CultureInfo culture)
{
    // Поскольку заботиться о "двунаправленной" привязке
    // не нужно, просто вернуть value.
    return value;
}
}

```

Метод `Convert()` вызывается при передаче значения от источника (`ScrollBar`) к цели (свойство `Content` элемента `Label`). Хотя принимается множество входных аргументов, для этого преобразования понадобится манипулировать только входным `object`, который представляет текущее значение `double`. Этот тип можно использовать для приведения к целому и возврата нового числа.

Метод `ConvertBack()` будет вызван, когда значение передается от цели к источнику (если включен двунаправленный режим). Здесь мы просто возвращаем значение. Это позволяет вводить в `TextBox` значение с плавающей точкой (например, 99.9) и автоматически преобразовывать его в целочисленное значение (99), когда производится выход из элемента управления. Такое "свободное" преобразование происходит потому, что метод `Convert()` будет вызван еще раз после вызова `ConvertBack()`. Если просто вернуть `null` из `ConvertBack()`, то синхронизация привязки будет нарушена, потому что `TextBox` все еще будет отображать число с плавающей точкой.

Установка привязок данных в коде

Имея этот класс, можно зарегистрировать специальный преобразователь с любым элементом управления, который желает использовать его. Это допускается делать исключительно в XAML-разметке, но тогда потребуются определить ряд специальных объектных ресурсов, которые рассматриваются в следующей главе. А пока класс преобразователя данных можно зарегистрировать в коде. Начните с очистки текущего определения элемента управления `<Label>` на вкладке `Data Binding`, чтобы больше не применялось расширение разметки `{Binding}`:

```

<Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue"
        BorderThickness="2" Content = "0"/>

```

В конструкторе окна вызовите новую закрытую вспомогательную функцию по имени `SetBindings()`, код которой показан ниже:

```

private void SetBindings()
{
    // Создать объект Binding.
    Binding b = new Binding();

    // Зарегистрировать преобразователь, источник и путь.
    b.Converter = new MyDoubleConverter();
    b.Source = this.mySB;
    b.Path = new PropertyPath("Value");

    // Вызвать метод SetBinding на Label.
    this.labelSBThumb.SetBinding(Label.ContentProperty, b);
}

```

Единственный фрагмент этой функции, который выглядит несколько необычно — это вызов `SetBinding()`. Обратите внимание, что первый параметр обращается к статическому, доступному только для чтения полю класса `Label` по имени `ContentProperty`. Как будет показано в главе 31, эта конструкция называется *свойством зависимости*. Пока просто знайте, что при установке привязки в коде в первом аргументе почти всегда тре-

буется указывать имя класса, который нуждается в привязке (в данном случае Label), за которым следует имя свойства с суффиксом Property (см. главу 31). Запустив приложение, можно удостовериться, что элемент Label отображает только целые числа.

Построение вкладки DataGrid

В предыдущем примере привязки данных было показано, как сконфигурировать два (или более) элемента управления для участия в операции привязки данных. Допускается также привязывать данные из файлов XML, базы данных и объектов в памяти. Для завершения рассматриваемого примера спроектируем финальную вкладку DataGrid, которая будет отображать информацию, извлеченную из таблицы Inventory базы данных AutoLot.

Как и с другими вкладками, начнем с замены текущего контейнера Grid контейнером StackPanel. Сделайте это путем непосредственного изменения XAML-разметки в Visual Studio. После этого в новом элементе StackPanel определите элемент управления DataGrid по имени gridInventory:

```
<TabItem x:Name="tabDataGrid" Header="DataGrid">
  <StackPanel>
    <DataGrid x:Name="gridInventory" Height="288"/>
  </StackPanel>
</TabItem>
```

Затем добавьте ссылку на сборку AutoLotDAL.dll, созданную в главе 23 (с применением Entity Framework). Когда вы сделаете это, Visual Studio автоматически сошлется на необходимую сборку System.Data.Entity.dll, а также на связанный файл App.config, который содержит требуемые данные строки соединения (проверьте в Solution Explorer, так ли это; в противном случае добавьте обязательные элементы вручную).

Откройте файл кода окна и добавьте финальную вспомогательную функцию по имени ConfigureGrid(); вызовите ее в конструкторе. Предполагая, что пространство имен AutoLotDAL импортировано, все, что остается сделать — это добавить несколько строк кода:

```
private void ConfigureGrid()
{
    using (AutoLotEntities context = new AutoLotEntities())
    {
        // Построить запрос LINQ, который извлекает некоторые данные из таблицы Inventory.
        var dataToShow = from c in context.Inventories
            select new { c.CarID, c.Make, c.Color, c.PetName };
        this.gridInventory.ItemsSource = dataToShow;
    }
}
```

Обратите внимание, что непосредственная привязка context.Inventories к коллекции ItemsSource сетки не производится; вместо этого строится запрос LINQ, который запрашивает те же данные в сущностях. Причина такого подхода в том, что объект Inventory также содержит дополнительные свойства EF (Entity Framework), которые будут появляться в сетке, но не отображаются на физическую базу данных.

Если запустить этот проект в том виде, как он есть, можно увидеть исключительно простую плоскую сетку. Чтобы немного улучшить ее, воспользуйтесь окном Properties в Visual Studio для редактирования категории Rows элемента DataGrid. Как минимум, понадобится установить свойство AlternationCount в 2 и задать специальные цвета в свойствах AlternatingRowBackground и RowBackground с помощью интегрированного редактора. Внешний вид вкладки для данного примера показан на рис. 28.39.

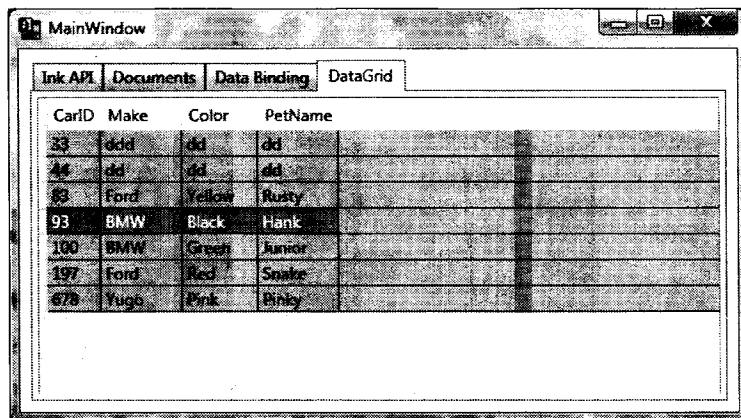


Рис. 28.39. Финальная вкладка проекта

На этом рассматриваемый в главе пример завершен. В последующих главах будут применяться и другие элементы управления, а к настоящему моменту вы должны почувствовать себя увереннее при построении пользовательских интерфейсов в Visual Studio и непосредственной работе с XAML-разметкой и кодом C#.

Исходный код. Проект WpfControlsAndAPIs доступен в подкаталоге Chapter 28.

Резюме

В этой главе рассматривались некоторые аспекты, связанные с элементами управления WPF, начиная с обзора инструментария для элементов управления и роли диспетчеров компоновки (панелей). Первый пример был посвящен построению простого приложения текстового процессора. В нем демонстрировалось использование интегрированной в WPF функциональности проверки правописания, а также создание главного окна с системой меню, строкой состояния и панелью инструментов.

Что более важно, вы узнали, как строить команды WPF. Эти независимые от элемента управления события можно присоединять к элементу пользовательского интерфейса или входному жесту для автоматического наследования готовой функциональности (например, операций с буфером обмена).

Кроме того, вы получили дополнительные сведения о применении встроенных визуальных конструкторов Visual Studio для построения пользовательских интерфейсов. В частности, с их помощью был построен сложный пользовательский интерфейс. Попутно вы узнали об интерфейсах Ink API и Documents API, доступных в WPF. Вы также получили представление об операциях привязки данных WPF, включая использование класса DataGrid из WPF для отображения информации из специальной базы данных AutoLot.

Службы визуализации графики WPF

В этой главе мы рассмотрим возможности графической визуализации WPF. Вы увидите, что WPF предлагает три отдельных способа визуализации графических данных — фигуры, рисунки и визуальные объекты. Разобравшись в преимуществах и недостатках каждого подхода, мы приступим к изучению мира интерактивной двухмерной графики с использованием классов из пространства имен `System.Windows.Shapes`. После этого будет показано, как с помощью рисунков и геометрии визуализировать двухмерные данные в облегченной манере. И, наконец, вы узнаете, как обеспечить наивысший уровень функциональности и производительности визуального уровня.

Попутно рассматривается множество связанных тем, таких как создание специальных кистей и перьев, применение графических трансформаций к визуализациям и выполнение операций проверки попадания. Будет показано, как упростить решение задач кодирования графики с помощью интегрированных инструментов Visual Studio и дополнительного средства, которое называется Expression Design.

На заметку! Графика является ключевым аспектом разработки WPF. Даже если задача не связана с построением приложения с интенсивной графикой (вроде видеоигры или мультимедийного приложения), темы, представленные в этой главе, критичны для работы с такими службами, как шаблоны элементов управления, анимация и настройка привязки данных.

Службы графической визуализации WPF

В WPF используется особая разновидность графической визуализации, которая называется *графикой режима сохранения* (retained mode). Это означает, что в случае использования XAML-разметки или процедурного кода для генерации графической визуализации за сохранение этих визуальных элементов и обеспечение их корректной перерисовки и обновления в оптимальной манере отвечает WPF. Поэтому визуализируемые графические данные присутствуют постоянно, даже когда конечный пользователь скрывает изображение, перемещая или сворачивая окно, перекрывая одно окно другим и т.д.

В отличие от этого, прежние версии API-интерфейсов графической визуализации (включая GDI+ в Windows Forms) были графическими системами *непосредственного режима* (immediate mode). В рамках этой модели ответственность за то, чтобы визуализируемые элементы корректно запомнились и обновлялись на протяжении времени жизни приложения, возлагается на программиста. Например, в приложении Windows Forms визуализация фигуры, такой как прямоугольник, предполагала обработку собы-

тия `Paint` (или переопределение виртуального метода `OnPaint()`), получение объекта `Graphics` для рисования прямоугольника и, что важнее всего, добавление инфраструктуры для обеспечения сохранности изображения в ситуации, когда пользователь изменяет размеры окна (например, за счет создания переменных-членов для представления позиции прямоугольника и вызова метода `Invalidate()` повсеместно в программе).

Переход от графики непосредственного режима к графике режима сохранения — однозначно хорошее решение, поскольку программисты в этом случае должны писать и сопровождать меньший объем кода. Однако нельзя утверждать, что графический API-интерфейс WPF полностью отличается от более ранних инструментариев визуализации. Например, подобно GDI+, в WPF поддерживаются разнообразные типы объектов кистей и перьев, приемы проверки попадания, области отсечения, графические трансформации и т.д. Таким образом, если у вас есть опыт работы в GDI+ (или GDI), это значит, что уже имеется хорошее представление о том, как выполнять базовую визуализацию под WPF.

Варианты графической визуализации WPF

Как и с другими аспектами WPF-разработки, существует несколько вариантов относительно того, как выполнять графическую визуализацию, после принятия решения о том, делать это в XAML-разметке или же в процедурном коде C# (либо, возможно, за счет их комбинации). В частности, в WPF предлагаются следующие три отличающихся подхода к визуализации графических данных.

- **Фигуры.** В пространстве имен `System.Windows.Shapes` определено небольшое количество классов для визуализации двумерных геометрических объектов (прямоугольников, эллипсов, многоугольников и т.п.). Хотя эти типы очень просты в применении и достаточно мощные, в случае непродуманного использования они могут привести к значительным накладным расходам памяти.
- **Рисунки и геометрии.** Второй способ визуализации графических данных в WPF предусматривает работу с наследниками абстрактного класса `System.Windows.Media.Drawing`. Применяя такие классы, как `GeometryDrawing` или `ImageDrawing` (в дополнение к различным *геометрическим объектам*), можно осуществлять визуализацию графических данных в более легковесной (но с ограниченными возможностями) манере.
- **Визуальные объекты.** Самый быстрый и наиболее легковесный способ визуализации графических данных в WPF предполагает использование визуального уровня, который доступен только через код C#. С помощью наследников класса `System.Windows.Media.Visual` можно взаимодействовать непосредственно с графической подсистемой WPF.

Причина существования разнообразных способов решения одной и той же задачи (т.е. визуализации графических данных) связана с необходимостью оптимального расхода памяти и обеспечения приемлемой производительности приложения. Поскольку WPF — система, интенсивно применяющая графику, нет ничего необычного в том, что приложению приходится визуализировать сотни или даже тысячи различных изображений на поверхности окна, и возможность выбора реализации (фигуры, рисунки или визуальные объекты) может иметь огромное значение.

Следует понимать, что при построении приложения WPF велика вероятность, что придется использовать все три варианта выбора. В качестве эмпирического правила отметим, что если нужен небольшой объем *интерактивных* графических данных, которыми должен манипулировать пользователь (принимать ввод от мыши, отображать всплывающие подсказки и т.п.), то стоит отдать предпочтение классам из пространства имен `System.Windows.Shapes`.

В отличие от этого, рисунки и геометрии лучше подходят, когда необходимо моделировать сложные, в основном не интерактивные векторные графические данные с применением XAML или C#. Хотя рисунки и геометрии могут реагировать на события мыши, поддерживают проверку попадания и выполняют операции перетаскивания, обычно это требует написания большего объема кода.

И последнее: если требуется самый быстрый способ визуализации значительных объемов графических данных, то для этого наиболее подходит визуальный уровень. Например, предположим, что WPF применяется для построения научного приложения, которое должно отображать тысячи показателей данных. За счет использования визуального уровня эти показатели можно визуализировать наиболее оптимальным способом. Как будет показано далее в главе, визуальный уровень доступен только из кода C#, но не из разметки XAML.

Независимо от выбранного подхода (фигуры, рисунки и геометрии либо визуальные объекты), всегда будут применяться общие графические примитивы, такие как кисти (для заполнения ограниченных областей), перья (для рисования контуров) и объекты трансформации (которые, как должно быть понятно, трансформируют данные). Начнем изучение с классов из пространства `System.Windows.Shapes`.

На заметку! WPF поставляется также с полноценным API-интерфейсом для визуализации и манипулирования трехмерной графикой, который в этой книге не рассматривается. Подробную информацию по этому поводу можно найти в документации .NET Framework 4.5 SDK.

Визуализация графических данных с использованием фигур

Члены пространства имен `System.Windows.Shapes` предлагают самый прямой и интерактивный, но и наиболее ресурсоемкий по занимаемой памяти способ визуализации двумерных изображений. Это пространство имен (определенное в сборке `PresentationFramework.dll`), достаточно мало, и состоит всего из шести запечатанных классов, расширяющих абстрактный базовый класс `Shape`: `Ellipse`, `Rectangle`, `Line`, `Polygon`, `Polyline` и `Path`.

Создайте новое приложение WPF под названием `RenderingWithShapes`. Найдите в браузере объектов Visual Studio абстрактный класс `Shape` (рис. 29.1) и раскройте все его узлы. Вы увидите, что каждый потомок `Shape` получает значительную часть функциональности по цепочке наследования.

Некоторые из этих родительских классов должны быть знакомы из материала предыдущих двух глав. Вспомните, например, что в классе `UIElement` определены многочисленные методы для получения ввода от мыши и обработки событий перетаскивания, а в классе `FrameworkElement` доступны члены для работы с изменением размеров, всплывающими подсказками, курсорами мыши и тому подобным. С учетом этой цепочки наследования, имейте в виду, что при визуализации графических данных с применением классов, производных от `Shape`, объекты получаются почти столь же функциональными (в смысле взаимодействия с пользователем), как и элементы управления WPF!

Например, определение факта щелчка на визуализированном изображении реализуется не сложнее, чем обработка события `MouseDown`. Вот простой пример: написав следующий код XAML объекта `Rectangle` в элементе `Grid` начального окна `Window`:

```
<Rectangle x:Name="myRect" Height="30" Width="30"
  Fill="Green" MouseDown="myRect_MouseDown"/>
```

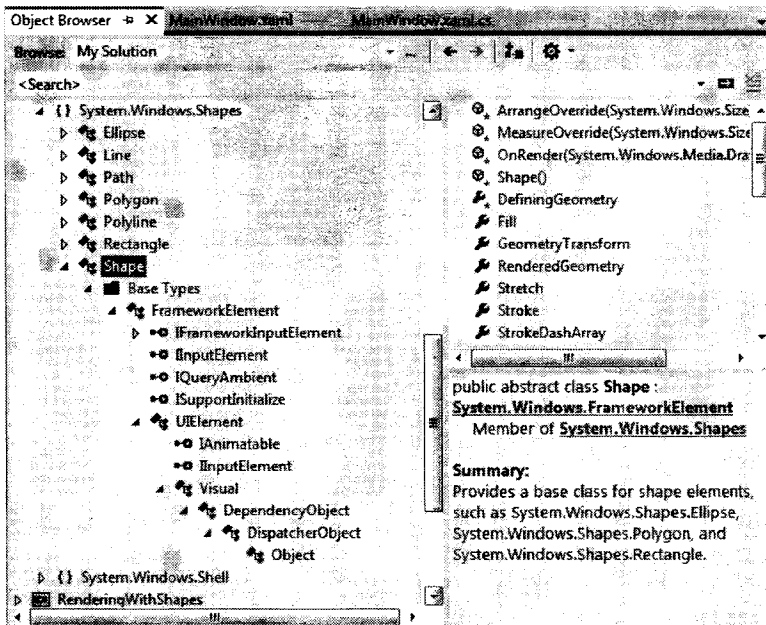


Рис. 29.1. Базовый класс Shape получает значительную часть функциональности от своих родительских классов

можно реализовать обработчик события `MouseDown`, который по щелчку на `Rectangle` изменяет его цвет фона:

```
private void myRect_MouseDown(object sender, MouseButtonEventArgs e)
{
    // Изменить цвет Rectangle при щелчке на нем.
    myRect.Fill = Brushes.Pink;
}
```

В отличие от других графических инструментальных наборов, здесь не придется писать громоздкий код инфраструктуры, который вручную сопоставляет координаты мыши с геометрией, вручную вычисляет попадание в границы, визуализирует в неотображаемый буфер и т.д. Члены `System.Windows.Shapes` просто реагируют на события, на которые вы регистрируетесь, подобно типичному элементу управления WPF (`Button` и т.д.).

Отрицательная сторона такой готовой функциональности состоит в том, что фигуры потребляют довольно много памяти. При построении научного приложения, которое рисует тысячи точек на экране, применение фигур будет неподходящим выбором (по сути, это будет столь же расточительным по памяти, как визуализация тысяч объектов `Button`). Однако когда нужно сгенерировать интерактивное двумерное векторное изображение, фигуры оказываются прекрасным выбором.

Помимо функциональности, унаследованной от родительских классов `UIElement` и `FrameworkElement`, в `Shape` определено множество собственных членов, наиболее полезные из которых перечислены в табл. 29.1.

На заметку! Если вы забудете установить свойства `Fill` и `Stroke`, WPF предоставит “невидимые” кисти, в результате чего фигура не будет видна на экране!

Таблица 29.1. Ключевые свойства базового класса Shape

Свойства	Описание
DefiningGeometry	Возвращает объект Geometry, представляющий общие размеры текущей фигуры. Этот объект содержит <i>только</i> точки, используемые для визуализации данных, и не имеет никаких следов функциональности UIElement или FrameworkElement
Fill	Позволяет указать "объект кисти" для визуализации внутренней области фигуры
GeometryTransform	Позволяет применять трансформацию к фигуре до ее визуализации на экране. Унаследованное свойство RenderTransform (из UIElement) применяет трансформацию после визуализации фигуры на экране
Stretch	Описывает, как фигура располагается внутри выделенного ей пространства, например, внутри диспетчера компоновки. Это управляется соответствующим перечислением System.Windows.Media.Stretch
Stroke	Определяет объект кисти или в некоторых случаях объект пера (который на самом деле также является кистью), используемый для рисования границы фигуры
StrokeDashArray, StrokeEndLineCap, StrokeStartLineCap, StrokeThickness	Эти (и прочие) свойства, связанные со штрихами, управляют тем, как сконфигурированы линии при рисовании границ фигуры. В большинстве случаев с помощью этих свойств будет настраиваться кисть, применяемая для рисования границы или линии

Добавление прямоугольников, эллипсов и линий на поверхность Canvas

Позже в этой главе будет показано, как использовать инструмент Expression Design для генерации XAML-описаний графических данных. А пока давайте построим WPF-приложение, которое может визуализировать фигуры с применением XAML и C#, и параллельно посмотрим, что собой представляет процесс проверки попадания. Прежде всего, удалите текущее описание Rectangle и логику обработчика событий C#. Затем добавьте в первоначальную XAML-разметку элемента <Window> определение контейнера <DockPanel>, содержащего (пока пустые) элементы <ToolBar> и <Canvas> (холст). Обратите внимание, что каждому содержащемуся элементу назначается подходящее имя через свойство Name.

```
<DockPanel LastChildFill="True">
  <ToolBar DockPanel.Dock="Top" Name="mainToolBar" Height="50">
  </ToolBar>
  <Canvas Background="LightBlue" Name="canvasDrawingArea"/>
</DockPanel>
```

Теперь наполним <ToolBar> набором объектов <RadioButton>, каждый из которых содержит специфический класс, производный от Shape. Обратите внимание, что каждому элементу <RadioButton> назначается одно и то же групповое имя GroupName (чтобы обеспечить взаимное исключение) и также подходящее имя.

```
<ToolBar DockPanel.Dock="Top" Name="mainToolBar" Height="50">
  <RadioButton Name="circleOption" GroupName="shapeSelection">
    <Ellipse Fill="Green" Height="35" Width="35" />
  </RadioButton>
  <RadioButton Name="rectOption" GroupName="shapeSelection">
    <Rectangle Fill="Red" Height="35"
      Width="35" RadiusY="10" RadiusX="10" />
  </RadioButton>
```

```

<RadioButton Name="lineOption" GroupName="shapeSelection">
  <Line Height="35" Width="35"
    StrokeThickness="10" Stroke="Blue"
    X1="10" Y1="10" Y2="25" X2="25"
    StrokeStartLineCap="Triangle" StrokeEndLineCap="Round" />
</RadioButton>
</ToolBar>

```

Как видите, объявления объектов `Rectangle`, `Ellipse` и `Line` в XAML довольно прямолинейны и требуют лишь минимальных комментариев. Вспомните, что свойство `Fill` позволяет указать кисть для рисования внутренностей фигуры. Когда нужна кисть сплошного цвета, можно просто задать жестко закодированную строку известных значений, а соответствующий преобразователь типов сгенерирует правильный объект. Одна интересная характеристика типа `Rectangle` связана с тем, что в нем определены свойства `RadiusX` и `RadiusY`, позволяющие при желании визуализировать скругленные углы.

Линия `Line` представляется начальной и конечной точками с помощью свойств `X1`, `X2`, `Y1` и `Y2` (учитывая, что *высота* и *ширина* применительно к линиям не имеют смысла). В разметке устанавливаются несколько дополнительных свойств, управляющих визуализацией начальной и конечной точек `Line`, а также конфигурирующих параметры штриха. На рис. 29.2 показана визуализированная панель инструментов в визуальном конструкторе WPF среды Visual Studio.

Теперь с помощью окна `Properties` (Свойства) среды Visual Studio создайте обработчик события `MouseLeftButtonDown` для `Canvas` и обработчик события `Click` для каждой `RadioButton`. В файле C# цель заключается в том, чтобы визуализировать выбранную фигуру (круг, квадрат или линию), когда пользователь щелкнет на `Canvas`.

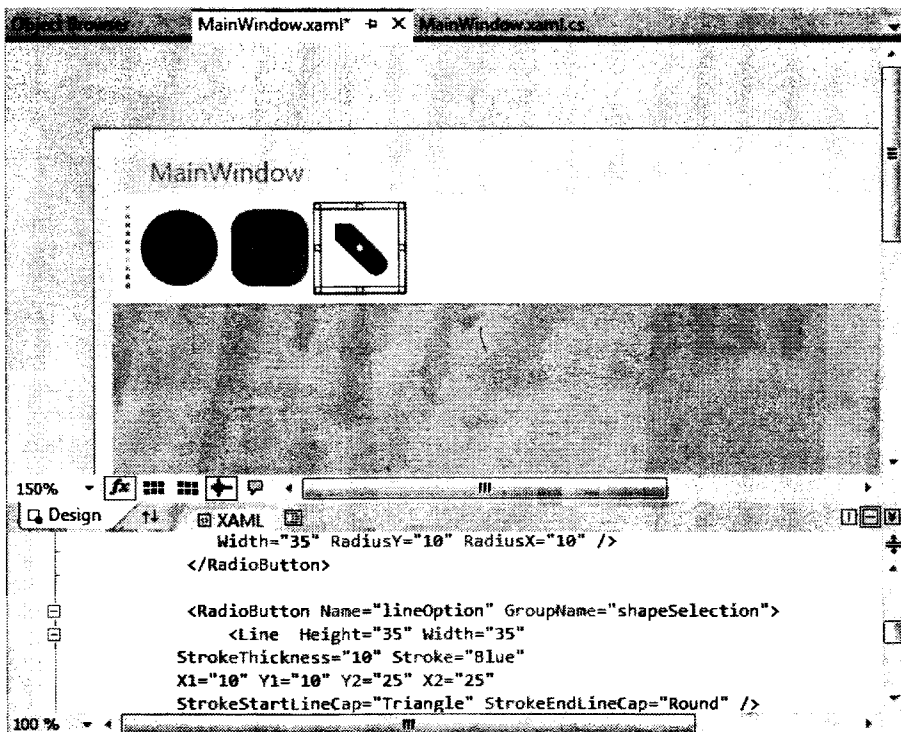


Рис. 29.2. Использование объектов `Shape` в качестве содержимого для набора элементов `RadioButton`

Для начала определите следующее встроенное перечисление (и соответствующую переменную-член) внутри класса, производного от Window:

```
public partial class MainWindow : Window
{
    private enum SelectedShape
    { Circle, Rectangle, Line }

    private SelectedShape currentShape;
    ...
}
```

Внутри каждого обработчика Click установите для переменной-члена currentShape необходимое значение SelectedShape. Например, ниже показан код обработчика события Click элемента RadioButton по имени circleOption. Реализуйте остальные два обработчика Click аналогичным образом.

```
private void circleOption_Click(object sender, RoutedEventArgs e)
{
    currentShape = SelectedShape.Circle;
}
```

С помощью обработчика события MouseLeftButtonDown элемента Canvas будет визуализироваться корректная фигура (предопределенного размера) в начальной точке, соответствующей позиции X, Y курсора мыши. Ниже показана полная реализация с последующим анализом:

```
private void canvasDrawingArea_MouseLeftButtonDown(object sender,
MouseButtonEventArgs e)
{
    Shape shapeToRender = null;

    // Сконфигурировать корректную фигуру для рисования.
    switch (currentShape)
    {
        case SelectedShape.Circle:
            shapeToRender = new Ellipse() { Fill = Brushes.Green, Height = 35, Width = 35 };
            break;
        case SelectedShape.Rectangle:
            shapeToRender = new Rectangle()
            { Fill = Brushes.Red, Height = 35, Width = 35, RadiusX = 10, RadiusY = 10 };
            break;
        case SelectedShape.Line:
            shapeToRender = new Line()
            {
                Stroke = Brushes.Blue,
                StrokeThickness = 10,
                X1 = 0, X2 = 50, Y1 = 0, Y2 = 50,
                StrokeStartLineCap = PenLineCap.Triangle,
                StrokeEndLineCap = PenLineCap.Round
            };
            break;
        default:
            return;
    }

    // Установить верхний левый угол для рисования на холсте.
    Canvas.SetLeft(shapeToRender, e.GetPosition(canvasDrawingArea).X);
    Canvas.SetTop(shapeToRender, e.GetPosition(canvasDrawingArea).Y);

    // Нарисовать фигуру.
    canvasDrawingArea.Children.Add(shapeToRender);
}
```

На заметку! Вы могли заметить, что объекты `Ellipse`, `Rectangle` и `Line`, создаваемые в этом методе, имеют те же настройки свойств, что и соответствующие определения XAML. Как и можно было ожидать, данный код вполне реально упростить, однако это требует понимания объектов ресурсов WPF, которые рассматриваются в главе 30.

Как видите, в коде осуществляется проверка переменной-члена `currentShape` для создания корректного объекта, производного от `Shape`. После этого устанавливаются значения координат левой верхней вершины внутри `Canvas` с использованием входного объекта `MouseButtonEventArgs`. И, наконец, в коллекцию объектов `UIElement`, поддерживаемую `Canvas`, добавляется новый производный от `Shape` объект. Если теперь запустить программу, то можно щелкать левой кнопки мыши где угодно на холсте и при этом в позиции щелчка будут появляться выбранные фигуры.

Удаление прямоугольников, эллипсов и линий с поверхности Canvas

Имея элемент `Canvas` с коллекцией объектов, может возникнуть вопрос: как теперь динамически удалить элемент, возможно, в ответ на щелчок пользователя правой кнопкой мыши на фигуре? Это делается с помощью класса `VisualTreeHelper` из пространства имен `System.Windows.Media`. В главе 31 вы узнаете о роли “визуальных деревьев” и “логических деревьев” более подробно. А пока обработаем событие `MouseRightButtonDown` класса `Canvas` и реализуем соответствующий обработчик, как показано ниже:

```
private void canvasDrawingArea_MouseRightButtonDown(
    object sender, MouseButtonEventArgs e)
{
    // Сначала получить координаты X,Y позиции, где пользователь выполнил щелчок.
    Point pt = e.GetPosition((Canvas)sender);

    // Воспользоваться методом HitTest() класса VisualTreeHelper,
    // чтобы проверить попадание на элемент внутри Canvas.
    HitTestResult result = VisualTreeHelper.HitTest(canvasDrawingArea, pt);

    // Если результат не равен null, щелчок произведен на фигуре.
    if (result != null)
    {
        // Получить фигуру, на которой совершен щелчок, и удалить ее из Canvas.
        canvasDrawingArea.Children.Remove(result.VisualHit as Shape);
    }
}
```

Этот метод начинается с получения точных координат `X`, `Y` позиции, где пользователь щелкнул в `Canvas`, и проверяет попадание через статический метод `VisualTreeHelper.HitTest()`. Возвращаемым значением является объект `HitTestResult`, который будет установлен в `null`, если пользователь не щелкнул на `UIElement` внутри `Canvas`. Если `HitTestResult` не равен `null`, с помощью свойства `VisualHit` можно получить элемент `UIElement`, на котором совершен щелчок, и привести его к объекту, производному от `Shape` (вспомните, что `Canvas` может содержать любой `UIElement`, а не только фигуры). Подробности устройства “визуального дерева” будут изложены в следующей главе.

На заметку! По умолчанию `VisualTreeHelper.HitTest()` возвращает `UIElement` самого верхнего уровня, на котором совершен щелчок, и не предоставляет информации о других объектах, расположенных под ним (т.е. перекрытых в Z-порядке).

После этих модификаций должна появиться возможность добавления фигуры на Canvas щелчком левой кнопкой мыши и ее удаления щелчком правой кнопкой мыши. На рис. 29.3 продемонстрирована функциональность текущего примера.

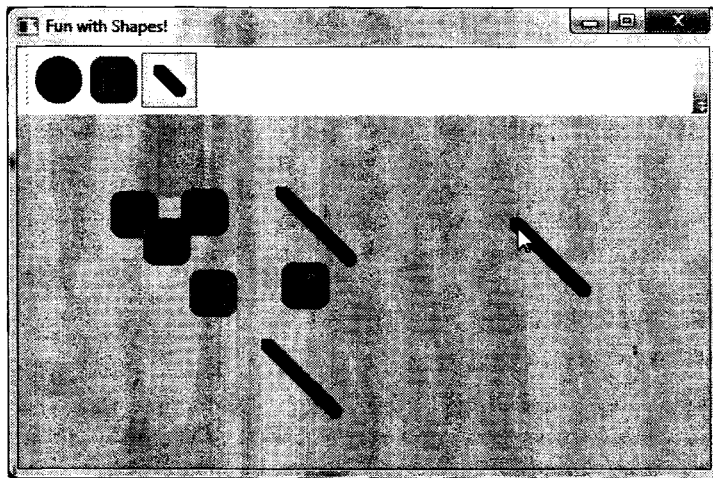


Рис. 29.3. Работа с фигурами

К этому моменту мы использовали объекты, производные от Shape, для визуализации содержимого элементов RadioButton с применением XAML-разметки и заполнили Canvas в коде C#. При рассмотрении роли кистей и графических трансформаций к этому примеру будет добавлена еще некоторая функциональность. Кстати говоря, в другом примере этой главы иллюстрируется реализация технологии перетаскивания на объектах UIElement. А пока уделим внимание членам пространства имен System.Windows.Shapes.

Работа с элементами Polyline и Polygon

В текущем примере используются только три класса, производные от Shape. Остальные дочерние классы (Polyline, Polygon и Path) чрезвычайно утомительно корректно визуализировать без инструментальной поддержки (такой как Expression Blend или Expression Design) — просто потому, что они требуют определения большого количества точек для своего выходного представления. Роль инструмента Expression Design будет продемонстрирована немного позже, а пока представим краткий обзор остальных типов Shapes.

Тип Polyline позволяет определить коллекцию координат (x, y) (через свойство Points) для рисования последовательности линейных сегментов, не требующих замыкания. Тип Polygon похож, однако он запрограммирован так, что всегда замыкает контур, соединяя начальную точку с конечной, и заполняет внутреннюю область указанной кистью. Предположим, что показанный ниже элемент <StackPanel> описан в редакторе Kaхaml или в специальном редакторе XAML, который был создан в качестве примера в главе 27:

```
<!-- Элемент Polyline не замыкает автоматически конечные точки -->
<Polyline Stroke="Red" StrokeThickness="20" StrokeLineJoin="Round"
  Points="10,10 40,40 10,90 300,50"/>

<!-- Элемент Polygon всегда замыкает конечные точки -->
<Polygon Fill="AliceBlue" StrokeThickness="5" Stroke="Green"
  Points="40,10 70,80 10,50" />
```

На рис. 29.4 показывает визуализированный вывод.

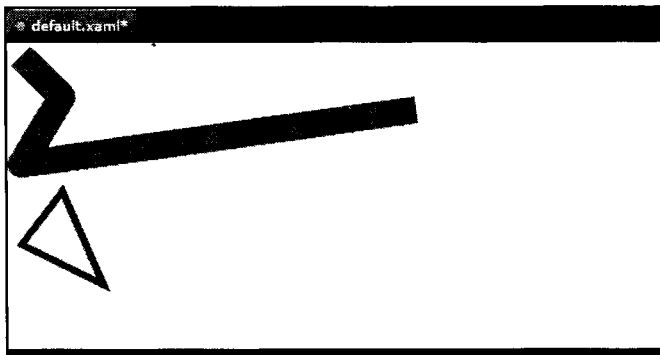


Рис. 29.4. Элементы Polyline и Polygon

Работа с элементом Path

Используя только типы `Rectangle`, `Ellipse`, `Polygon`, `Polyline` и `Line`, нарисовать детализированное двухмерное векторное изображение было бы чрезвычайно сложно, поскольку эти примитивы не позволяют легко фиксировать графические данные, подобные кривым, объединениям перекрывающихся данных и т.д. Последний производный от `Shape` класс `Path` предоставляет возможность определения сложных двухмерных графических данных в виде коллекции независимых геометрий. После определения коллекции таких геометрий ее можно присвоить свойству `Data` класса `Path`, где эта информация будет применяться для визуализации сложного двухмерного изображения.

Свойство `Data` получает экземпляр класса, производного от `System.Windows.Media.Geometry`, который содержит ключевые члены, перечисленные в табл. 29.2.

Таблица 29.2. Избранные члены класса `System.Windows.Media.Geometry`

Член	Описание
<code>Bounds</code>	Устанавливает текущий ограничивающий прямоугольник, содержащий в себе геометрию
<code>FillContains()</code>	Определяет, находится ли данный объект <code>Point</code> (или другой объект <code>Geometry</code>) в границах определенного класса, производного от <code>Geometry</code> . Это полезно при вычислениях с целью проверки попадания
<code>GetArea()</code>	Возвращает общую область, занятую объектом, производным от <code>Geometry</code>
<code>GetRenderBounds()</code>	Возвращает объект <code>Rect</code> , содержащий наименьший возможный прямоугольник, который может быть использован для визуализации объекта класса, производного от <code>Geometry</code>
<code>Transform</code>	Назначает геометрии объект <code>Transform</code> для изменения визуализации

Классы, которые расширяют класс `Geometry` (табл. 29.3), выглядят очень похоже на свои аналоги, производные от `Shape`. Например, `EllipseGeometry` имеет члены, похожие на члены класса `Ellipse`. Значительное отличие состоит в том, что классы, производные от `Geometry`, не знают, как визуализировать себя непосредственно, поскольку они не являются `UIElement`. Вместо этого производные от `Geometry` классы представляют всего лишь коллекцию данных о точках, которая указывает `Path`, как визуализировать себя.

На заметку! Path — не единственный класс в WPF, который может работать с коллекциями геометрий. Например, DoubleAnimationUsingPath, DrawingGroup, GeometryDrawing и даже UIElement могут использовать геометрии для визуализации через свойства PathGeometry, ClipGeometry, Geometry и Clip соответственно.

Таблица 29.3. Классы, производные от Geometry

Класс	Описание
LineGeometry	Представляет прямую линию
RectangleGeometry	Представляет прямоугольник
EllipseGeometry	Представляет эллипс
GeometryGroup	Позволяет сгруппировать вместе несколько объектов Geometry
CombinedGeometry	Позволяет объединить два разных объекта Geometry в единую фигуру
PathGeometry	Представляет фигуру, состоящую из линий и кривых

Ниже приведена определенная в Xaml разметка для элемента Path, который использует несколько производных от Geometry типов. Обратите внимание, что свойство Data объекта Path устанавливается в объект GeometryGroup, содержащий другие производные от Geometry объекты, такие как EllipseGeometry, RectangleGeometry и LineGeometry. Результат можно видеть на рис. 29.5.

```
<!-- Элемент Path содержит набор объектов
      Geometry, установленный в свойстве Data -->
<Path Fill = "Orange" Stroke = "Blue" StrokeThickness = "3">
  <Path.Data>
    <GeometryGroup>
      <EllipseGeometry Center = "75,70"
        RadiusX = "30" RadiusY = "30" />
      <RectangleGeometry Rect = "25,55 100 30" />
      <LineGeometry StartPoint="0,0" EndPoint="70,30" />
      <LineGeometry StartPoint="70,30" EndPoint="0,30" />
    </GeometryGroup>
  </Path.Data>
</Path>
```

Изображение на рис. 29.5 может быть визуализировано с помощью показанных ранее классов Line, Ellipse и Rectangle. Однако для этого потребуется поместить в память несколько объектов UIElement. Когда для моделирования того, что нужно нарисовать, используются геометрии, а затем коллекция геометрий помещается в контейнер, который может визуализировать данные (в данном случае — Path), тем самым сокращается расход памяти.



Рис. 29.5. Элемент Path, содержащий различные объекты Geometry

Теперь вспомните, что Path имеет ту же цепочку наследования, что и любой член System.Windows.Shapes, поэтому он в состоянии отправлять те же уведомления о событиях, что и другие элементы UIElement. Следовательно, если определить тот же самый элемент <Path> в проекте Visual Studio, то можно будет выяснить, что пользователь щелкнул в любом месте линии, просто обрабатывая событие мыши (редактор Xaml не позволяет обрабатывать события для написанной разметки).

Мини-язык моделирования путей

Из всех классов, перечисленных в табл. 29.2, PathGeometry является наиболее сложным для конфигурирования в терминах XAML и кода. Это связано с тем, что каждый сегмент PathGeometry состоит из объектов, содержащих различные сегменты и фигуры (например, ArcSegment, BezierSegment, LineSegment, PolyBezierSegment, PolyLineSegment, PolyQuadraticBezierSegment и т.д.). Ниже приведен пример объекта Path, свойство Data которого установлено в <PathGeometry>, состоящий из различных фигур и сегментов.

```
<Path Stroke="Black" StrokeThickness="1" >
  <Path.Data>
    <PathGeometry>
      <PathGeometry.Figures>
        <PathFigure StartPoint="10,50">
          <PathFigure.Segments>
            <BezierSegment
              Point1="100,0"
              Point2="200,200"
              Point3="300,100"/>
            <LineSegment Point="400,100" />
            <ArcSegment
              Size="50,50" RotationAngle="45"
              IsLargeArc="True" SweepDirection="Clockwise"
              Point="200,100"/>
          </PathFigure.Segments>
        </PathFigure>
      </PathGeometry.Figures>
    </PathGeometry>
  </Path.Data>
</Path>
```

Следует отметить, что очень немногим программистам понадобится когда-либо вручную строить сложные двумерные изображения, напрямую описывая объекты классы, производные от Geometry или PathSegment. В действительности сложные пути будут формироваться автоматически при работе с Expression Design.

Тем не менее, даже учитывая помощь упомянутых ранее инструментов, объем XAML-разметки, требуемый для определения сложных объектов Path, может быть устрашающе большим, поскольку такие данные должны включать полные описания разнообразных классов, производных от Geometry или PathSegment. Для того чтобы создавать более краткую и компактную разметку, в классе Path поддерживается специализированный "мини-язык".

Например, вместо установки свойства Data объекта Path в коллекцию объектов, производных от Geometry и PathSegment, его можно установить в единственный строковый литерал, содержащий набор известных символов и различных значений, которые определяют фигуру, подлежащую визуализации. На самом деле инструменты Expression для построения объекта Path внутри автоматически используют мини-язык. Рассмотрим простой пример и его результирующий вывод (рис. 29.6):

```
<Path Stroke="Black" StrokeThickness="3"
  Data="M 10,75 C 70,15 250,270 300,175 H 240" />
```

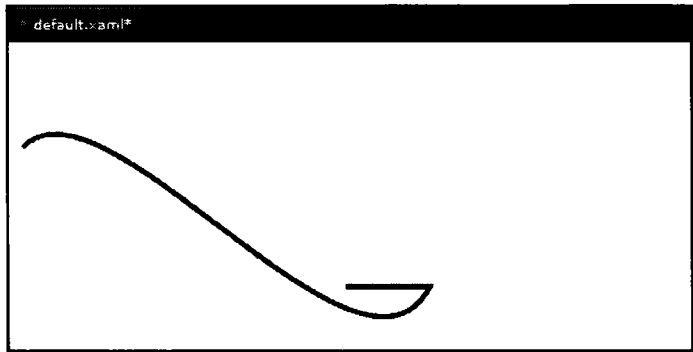


Рис. 29.6. Мини-язык моделирования путей позволяет компактно описывать объектную модель Geometry/PathSegment

Команда *M* (от *move* — переместить) принимает координаты *X*, *Y* позиции, представляющей начальную точку рисования. Команда *C* (от *curve* — кривая) принимает последовательность точек для визуализации кривой (точнее — кубической кривой Безье), а команда *H* (от *horizontal* — горизонтальная) рисует горизонтальную линию.

И снова следует отметить, что вручную строить или анализировать строковый литерал, содержащий инструкции мини-языка моделирования путей, придется очень редко. Однако, во всяком случае, XAML-разметка, сгенерированная специализированными инструментами, не будет казаться чем-то непонятным. Если интересуют детали этой конкретной грамматики, обращайтесь в раздел “Path Markup Syntax” (“Синтаксис разметки путей”) документации .NET Framework 4.5 SDK.

Кисти и перья WPF

Каждый из способов графической визуализации (фигура, рисование и геометрии, а также визуальные объекты) интенсивно использует *кисти*, которые позволяют управлять заполнением внутренней области двумерной фигуры. В WPF предлагаются шесть разных типов кистей, и все они расширяют класс `System.Windows.Media.Brush`. Хотя `Brush` является абстрактным классом, его потомки, перечисленные в табл. 29.4, могут применяться для заполнения области содержимым почти любого рода.

Таблица 29.4. Классы, производные от `Brush`

Класс	Описание
<code>DrawingBrush</code>	Заполняет область с помощью объекта, производного от <code>Drawing</code> (<code>GeometryDrawing</code> , <code>ImageDrawing</code> или <code>VideoDrawing</code>)
<code>OmageBrush</code>	Заполняет область изображением (представленным объектом <code>ImageSource</code>)
<code>LinearGradientBrush</code>	Заполняет область линейным градиентом
<code>RadialGradientBrush</code>	Заполняет область радиальным градиентом
<code>SolidColorBrush</code>	Заполняет область сплошным цветом, указанным в свойстве <code>Color</code>
<code>VisualBrush</code>	Заполняет область с помощью объекта, производного от <code>Visual</code> (<code>DrawingVisual</code> , <code>Viewport3DVisual</code> и <code>ContentVisual</code>)

Классы `DrawingBrush` и `VisualBrush` позволяют строить кисть на основе существующего класса, производного от `Drawing` или `Visual`. Эти классы кистей используются при работе с двумя другими вариантами графики WPF (рисунками или визуальными объектами) и будут объясняться далее в этой главе.

Класс `ImageBrush` позволяет строить кисть, отображающую графические данные из внешнего файла или встроенного ресурса приложения, который указан в его свойстве `ImageSource`. Остальные типы кистей (`LinearGradientBrush` и `RadialGradientBrush`) применять достаточно легко, хотя ввод необходимого кода XAML может быть утомительным. К счастью, в среде `Visual Studio` поддерживаются интегрированные редакторы кистей, которые упрощают задачу генерации стилизованных кистей.

Конфигурирование кистей с использованием Visual Studio

Давайте обновим WPF-приложение рисования `RenderingShapes` для использования некоторых интересных кистей. Три фигуры, с которыми мы имели дело до сих пор при визуализации данных в панели инструментов, применяют простые сплошные цвета, так что их значения можно зафиксировать с помощью простых строковых литералов. Чтобы сделать задачу немного более интересной, применим теперь интегрированный редактор кистей. Убедитесь, что редактор XAML начального окна открыт в IDE-среде, и выберите элемент `Ellipse`. В окне `Properties` найдите категорию `Brush` (Кисть) и щелкните на свойстве `Fill`, расположенном сверху (рис. 29.7).

В верхней части редактора кистей находится набор свойств выбранного элемента, которые являются «совместимым с кистью» (т.е. `Fill`, `Stroke` и `OpacityMask`). Под ними расположен набор вкладок, которые позволяют конфигурировать различные типы кистей, включая текущую кисть со сплошным цветом. Для управления цветом текущей кисти можно использовать инструмент выбора цвета, а также ползунки `ARGB` (alpha, red, green and blue (прозрачность, красный, зеленый и синий)). С помощью этих ползунков и связанной с ними области выбора цвета можно создать любой сплошной цвет. Воспользуйтесь этими инструментами для изменения цвета в свойстве `Fill` элемента `Ellipse` и просмотрите результирующую XAML-разметку. Вы заметите, что цвет сохраняется в виде шестнадцатеричного значения, например:

```
<Ellipse Fill="#FF47CE47" Height="35" Width="35" />
```

Что более интересно, тот же самый редактор позволяет конфигурировать и градиентные кисти, которые используются для определения последовательностей цветов и точек перехода. Вспомните, что редактор кистей предоставляет набор вкладок, первая из которых позволяет установить `null`-кисть для визуализированного вывода. Остальные четыре предназначены для установки кисти сплошного цвета (это было только что сделано), градиентной кисти, мозаичной кисти и кисти с изображением.

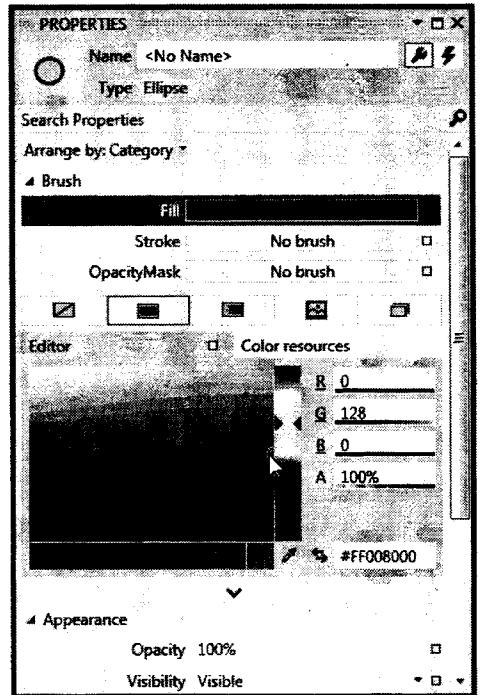


Рис. 29.7. Любое свойство, требующее кисти, может быть сконфигурировано с помощью интегрированного редактора кистей

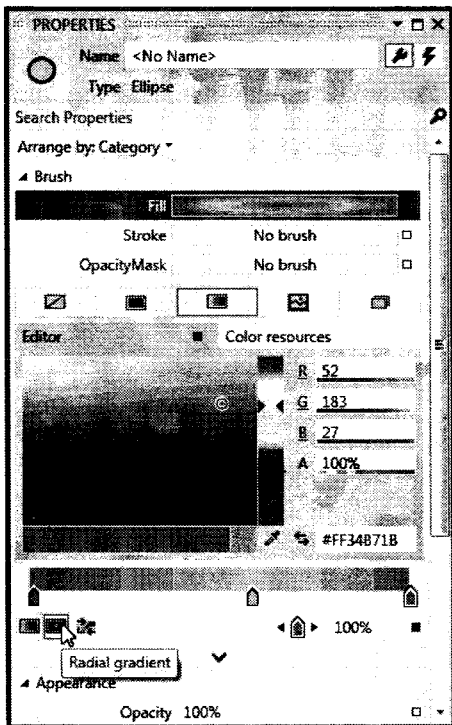


Рис. 29.8. Редактор кистей Visual Studio позволяет строить базовые градиентные кисти

Щелкните на вкладке градиентной кисти; редактор отобразит несколько новых опций (рис. 29.8). Три кнопки в нижнем левом углу позволяют выбрать линейный градиент, радиальный градиент или обратить градиентные переходы. Полоса внизу покажет текущий цвет каждого градиентного перехода, и каждый из них будет представлен специальным ползунком. По мере перетаскивания ползунка по полосе градиента можно управлять смещением градиента. Более того, щелкнув на конкретном ползунке, можно изменить цвет определенного градиентного перехода с помощью селектора цвета. Наконец, щелчок прямо на полосе градиента позволяет добавлять дополнительные градиентные переходы.

Потратьте некоторое время на освоение этого редактора, чтобы построить радиальную градиентную кисть, содержащую три градиентных перехода, и установите их цвета по своему выбору. На рис. 29.8 показан пример кисти, использующей три разных оттенка зеленого цвета.

В результате IDE-среда обновит XAML-разметку набором специальных кистей, установив их в совместимых с кистями свойствах (в рассматриваемом примере это свойство `Fill` элемента `Ellipse`) с применением синтаксиса "свойство-элемент".

Например:

```
<Ellipse Height="35" Width="35">
  <Ellipse.Fill>
    <RadialGradientBrush>
      <GradientStop Color="#FF87E71B" Offset="0.589" />
      <GradientStop Color="#FF2BA92B" Offset="0.013" />
      <GradientStop Color="#FF34B71B" Offset="1" />
    </RadialGradientBrush>
  </Ellipse.Fill>
</Ellipse>
```

Конфигурирование кистей в коде

Итак, мы построили специальную кисть для XAML-определения элемента `Ellipse`, но поскольку соответствующий код C# устарел, в результате все равно получается крут со сплошным зеленым цветом. Чтобы восстановить синхронизацию, модифицируем соответствующий оператор `case` для использования только что созданной кисти. Ниже приведено необходимое обновление, которое выглядит несколько сложнее, чем можно было ожидать, потому что шестнадцатеричное значение преобразуется в подходящий объект `Color` через класс `System.Windows.Media.ColorConverter` (измененный вывод показан на рис. 29.9).

```
case SelectedShape.Circle:
    shapeToRender = new Ellipse() { Height = 35, Width = 35 };

    // Создать кисть RadialGradientBrush в коде.
    RadialGradientBrush brush = new RadialGradientBrush();
```

```
brush.GradientStops.Add(new GradientStop(
    (Color)ColorConverter.ConvertFromString("#FF87E71B"), 0.589));
brush.GradientStops.Add(new GradientStop(
    (Color)ColorConverter.ConvertFromString("#FF2BA92B"), 0.013));
brush.GradientStops.Add(new GradientStop(
    (Color)ColorConverter.ConvertFromString("#FF34B71B"), 1));
shapeToRender.Fill = brush;
break;
```

Кстати, объекты `GradientStop` можно строить, указывая простой цвет в качестве первого параметра конструктора с использованием перечисления `Colors`, возвращающего сконфигурированный объект `Color`:

```
GradientStop g = new GradientStop(Colors.Aquamarine, 1);
```

Если же нужен более тонкий контроль, можно сразу передать сконфигурированный объект `Color`, например:

```
Color myColor = new Color() { R = 200, G = 100, B = 20, A = 40 };
GradientStop g = new GradientStop(myColor, 34);
```

Разумеется, применение перечисления `Colors` и класса `Color` не ограничено градиентными кистями. Их можно использовать где угодно — там, где нужно представлять значение цвета в коде.

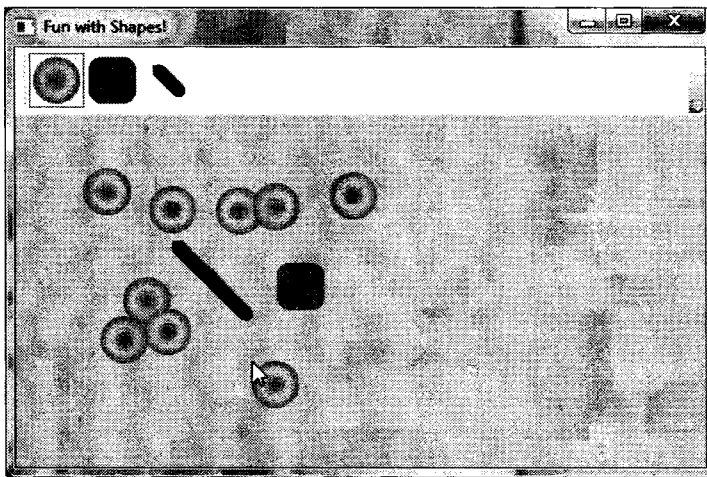


Рис. 29.9. Рисование более изящных кругов

Конфигурирование перьев

В отличие от кисти, *перо* — это объект, предназначенный для рисования границ геометрий, либо в случае класса `Line` или `PolyLine` — самих геометрий. В частности, класс `Pen` позволяет рисовать линию заданной толщины, представленную значением типа `double`. Вдобавок `Pen` может быть сконфигурирован с помощью тех же свойств, что были представлены в классе `Shape`, таких как начальный и конечный концы пера, шаблоны точек-тире и т.д. Например:

```
<Pen Thickness="10" LineJoin="Round" EndLineCap="Triangle" StartLineCap="Round" />
```

Во многих случаях непосредственно создавать объект `Pen` не придется, поскольку это делается неявно, когда устанавливаются значения для таких свойств, как `StrokeThickness` в классе, производном от `Shape` (а также других типов `UIElement`).

Однако построение специального объекта Pen может пригодиться при работе с типами, производными от Drawing (которые рассматриваются далее в главе). В Visual Studio редактор перьев, как таковой, отсутствует, но все свойства выбранного элемента, связанные со штрихами, можно конфигурировать в окне Properties.

Применение графических трансформаций

Чтобы завершить обсуждение использования фигур, рассмотрим тему *трансформаций*. WPF поставляется с многочисленными классами, расширяющими абстрактный базовый класс System.Windows.Media.Transform. В табл. 29.5 перечислены основные классы, производные от Transform.

Таблица 29.5. Основные потомки типа System.Windows.Media.Transform

Класс	Описание
MatrixTransform	Создает произвольную матричную трансформацию, используемую для манипулирования объектами или координатными системами на двухмерной поверхности
RotateTransform	Поворачивает объект по часовой стрелке вокруг указанной точки в двухмерной системе координат (x, y)
ScaleTransform	Масштабирует объект в двухмерной системе координат (x, y)
SkewTransform	Перекашивает объект в двухмерной системе координат (x, y)
TranslateTransform	Транслирует (перемещает) объект в двухмерной системе координат (x, y)
TransformGroup	Представляет комбинированный объект Transform, состоящий из других объектов Transform

Трансформации могут применяться к любому классу UIElement (например, к потомкам Shape, а также к таким элементам управления, как Button, TextBox и им подобным). Используя эти классы трансформаций, можно визуализировать графические данные под заданным углом, перекашивать изображение на поверхности, а также растягивать, сжимать или поворачивать целевой элемент самыми разными способами.

На заметку! Хотя объекты трансформаций могут применяться повсеместно, вы найдете их наиболее удобными при работе с анимацией WPF и специальными шаблонами элементов управления. Как будет показано далее, анимацию WPF можно использовать для включения в специальный элемент управления визуальных сигналов, предназначенных конечному пользователю.

Объекты трансформаций (или целое их множество) могут назначаться целевому объекту (Button, Path и т.п.) с помощью двух общих свойств. Свойство LayoutTransform удобно тем, что трансформация происходит *перед* визуализацией элементов в диспетчере компоновки, и потому она не влияет на z-упорядочивание (другими словами, трансформируемые данные изображений не перекрываются).

Свойство RenderTransform, с другой стороны, работает после того, как элементы оказались в своих контейнерах, поэтому вполне возможно, что элементы будут трансформированы таким образом, что могут перекрывать друг друга, в зависимости от того, как они организованы в контейнере.

Первый взгляд на трансформации

Скоро мы добавим к проекту `RenderingWithShapes` некоторую трансформирующую логику. Чтобы увидеть объект трансформации в действии, откройте редактор `Кахам1` (или собственный специальный редактор XML) и определите простой элемент `<StackPanel>` в корневом элементе `<Page>` или `<Window>`, установив свойство `Orientation` в `Horizontal`! Теперь добавьте следующий элемент `<Rectangle>`, который будет нарисован под углом в 45 градусов с использованием объекта `RotateTransform`:

```
<!-- Элемент Rectangle с трансформацией поворотом -->
<Rectangle Height="100" Width="40" Fill="Red">
  <Rectangle.LayoutTransform>
    <RotateTransform Angle="45"/>
  </Rectangle.LayoutTransform>
</Rectangle>
```

Далее элемент `<Button>` перекашивается на поверхности на 20% с помощью трансформации `<SkewTransform>`:

```
<!-- Элемент Button с трансформацией перекосом -->
<Button Content="Click Me!" Width="95" Height="40">
  <Button.LayoutTransform>
    <SkewTransform AngleX="20" AngleY="20"/>
  </Button.LayoutTransform>
</Button>
```

И для полноты картины ниже приведен элемент `<Ellipse>`, масштабированный на 20% посредством трансформации `ScaleTransform` (обратите внимание на значения, установленные в свойствах `Height` и `Width`), а также элемент `<TextBox>`, к которому применена группа объектов трансформации:

```
<!-- Элемент Ellipse, масштабированный на 20% -->
<Ellipse Fill="Blue" Width="5" Height="5">
  <Ellipse.LayoutTransform>
    <ScaleTransform ScaleX="20" ScaleY="20"/>
  </Ellipse.LayoutTransform>
</Ellipse>

<!-- Элемент TextBox, повернутый и скошенный -->
<TextBox Text="Me Too!" Width="50" Height="40">
  <TextBox.LayoutTransform>
    <TransformGroup>
      <RotateTransform Angle="45"/>
      <SkewTransform AngleX="5" AngleY="20"/>
    </TransformGroup>
  </TextBox.LayoutTransform>
</TextBox>
```

Следует отметить, что при применении трансформации не требуется выполнять какие-либо ручные вычисления для корректной проверки попадания, принятия фокуса ввода и т.п. Графический механизм WPF решает такие задачи самостоятельно. Например, на рис. 29.10 можно видеть, что элемент `TextBox` по-прежнему реагирует на клавиатурный ввод.

Трансформация данных Canvas

Теперь давайте посмотрим, как включить некоторую логику трансформации в пример `RenderingWithShapes`. В дополнение к применению объекта трансформации к одиночному элементу (`Rectangle`, `TextBox` и т.д.), трансформации можно также применять на уровне диспетчера компоновки, чтобы трансформировать все его внутренние данные.

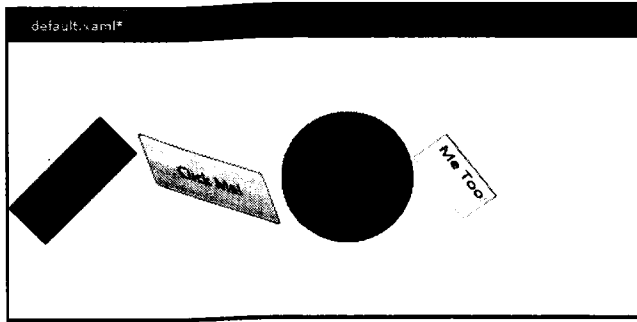


Рис. 29.10. Результат применения объектов графических трансформаций

Например, можно визуализировать всю панель `<DockPanel>` главного окна, повернув ее на заданный угол:

```
<DockPanel LastChildFill="True">
  <DockPanel.LayoutTransform>
    <RotateTransform Angle="45"/>
  </DockPanel.LayoutTransform>
  ...
</DockPanel>
```

Это несколько чрезмерно для рассматриваемого примера, поэтому давайте добавим последнее (менее энергичное) средство, которое позволяет пользователю повернуть весь контейнер `Canvas` с содержащейся в нем графикой. Начните с добавления финального элемента `<ToggleButton>` к `<ToolBar>`, определив его следующим образом:

```
<ToggleButton Name="flipCanvas" Click="flipCanvas_Click" Content="Flip Canvas!"/>
```

Внутри обработчика события `Click`, который инициируется щелчком на новом элементе `ToggleButton`, создадим объект `RotateTransform` и подключим его к объекту `Canvas` через свойство `LayoutTransform`. Если щелчок на `ToggleButton` не производился, трансформация удаляется за счет установки этого же свойства в `null`.

```
private void flipCanvas_Click(object sender, RoutedEventArgs e)
{
    if (flipCanvas.IsChecked == true)
    {
        RotateTransform rotate = new RotateTransform(-180);
        canvasDrawingArea.LayoutTransform = rotate;
    }
    else
    {
        canvasDrawingArea.LayoutTransform = null;
    }
}
```

Запустите приложение и добавьте какие-то графические фигуры в область `Canvas`. После щелчка на новой кнопке обнаружится, что фигуры выходят за границы `Canvas` (рис. 29.11). Причина в том, что не был определен прямоугольник отсечения.

Исправить это просто. Вместо того чтобы вручную писать сложную логику отсечения, просто установите свойство `ClipToBounds` элемента `<Canvas>` в `true`, что предотвратит визуализацию дочерних элементов вне границ родителя. Запустив программу вновь, вы обнаружите, что теперь графические данные не покидают границ отведенной области.

```
<Canvas ClipToBounds = "True" ... >
```

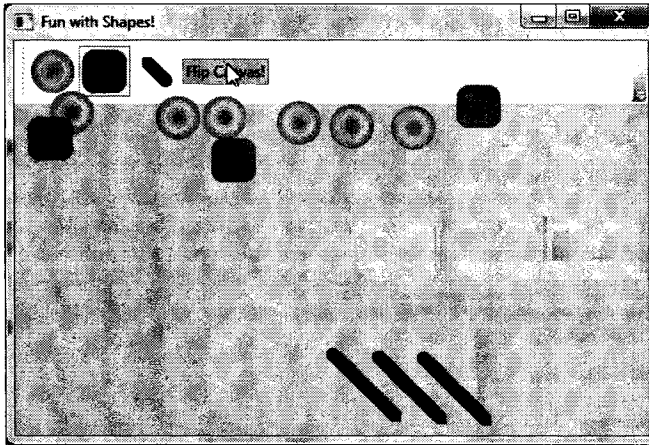


Рис. 29.11. После трансформации фигуры выходят за границы Canvas

Последняя небольшая модификация, которую понадобится провести, связана с тем фактом, что когда вы поворачиваете холст нажатием кнопки переключения, а затем щелкаете на нем для рисования новой фигуры, точка, где был произведен щелчок, не является той позицией, куда попадут графические данные. Вместо этого они появляются под курсором мыши.

Чтобы исправить данную проблему, давайте еще раз взглянем на код решения для этого примера, к которому добавляется еще одна переменная-член булевского типа (`isFlipped`). Она обеспечивает применение той же трансформации к ранее нарисованной фигуре перед выполнением визуализации (через `RenderTransform`). Ниже показан соответствующий фрагмент кода:

```
private void canvasDrawingArea_MouseLeftButtonDown(
    object sender, MouseButtonEventArgs e)
{
    Shape shapeToRender = null;
    ...

    // isFlipped – закрытое булевское поле. Его значение
    // изменяется при щелчке на кнопке переключения.
    if (isFlipped)
    {
        RotateTransform rotate = new RotateTransform(-180);
        shapeToRender.RenderTransform = rotate;
    }

    // Установить верхнюю левую точку для рисования на холсте.
    Canvas.SetLeft(shapeToRender, e.GetPosition(canvasDrawingArea).X);
    Canvas.SetTop(shapeToRender, e.GetPosition(canvasDrawingArea).Y);

    // Нарисовать фигуру.
    canvasDrawingArea.Children.Add(shapeToRender);
}
```

На этом рассмотрение пространства имен `System.Windows.Shapes`, кистей и трансформаций завершено. Прежде чем перейти к теме визуализации графики с помощью рисунков и геометрий, давайте посмотрим, как Visual Studio может упростить работу с примитивными графическими элементами.

Работа с редактором трансформаций Visual Studio

В предыдущем примере разнообразные трансформации применялись за счет ручного ввода разметки и написания некоторого кода C#. Хотя это вполне удобно, в последней версии Visual Studio имеется встроенный редактор трансформаций, который позволяет легко генерировать необходимую разметку трансформаций с использованием интегрированных инструментов. Теперь вспомните, что любой элемент пользовательского интерфейса может быть получателем служб трансформаций, в том числе и система компоновки, содержащая разнообразные элементы управления. Для иллюстрации применения редактора трансформаций Visual Studio создайте новое приложение WPF по имени FunWithTransforms.

Построение начальной компоновки

Первым делом, разделите первоначальный элемент Grid на две колонки с использованием встроенного редактора сетки (точные размеры роли не играют). Теперь найдите в панели инструментов элемент управления StackPanel и добавьте его так, чтобы он занимал все пространство первой колонки Grid. Ниже показан пример:

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="221*" />
    <ColumnDefinition Width="288*" />
  </Grid.ColumnDefinitions>
  <StackPanel HorizontalAlignment="Left" Height="292" Margin="10,10,0,0"
    VerticalAlignment="Top" Width="201" />
</Grid>
```

Затем выберите этот контейнер StackPanel в окне Document Outline (Схема документа) и добавьте в него три элемента управления Button (рис. 29.12).

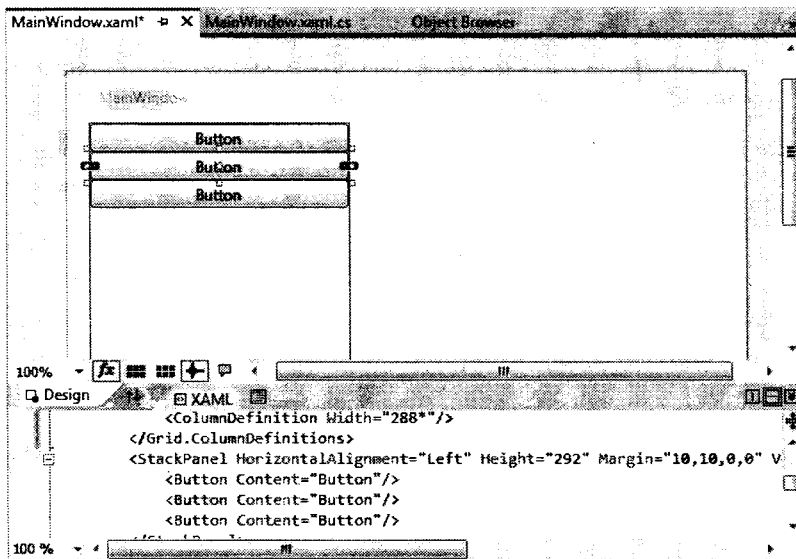


Рис. 29.12. Контейнер StackPanel с элементами управления Button

Теперь по очереди выберите элементы Button и укажите для свойства Content (находящегося в области Common (Общие) окна Properties (Свойства)) значения Skew, Rotate и Flip.

Кроме того, в области Name (Имя) окна Properties назначьте каждой кнопке подходящее имя, такое как `btnSkew`, `btnRotate` и `btnFile`, а на вкладке Events (События) обработайте событие `Click` для каждого элемента управления `Button`. Вскоре мы реализуем эти обработчики.

Для завершения пользовательского интерфейса создайте графику по своему выбору (применив любой прием, показанный в этой главе) во второй колонке `Grid`. Финальная компоновка показана на рис. 29.13. Здесь имеются два элемента управления `Ellipse`, сгруппированные в элемент управления `Canvas`, который имеет имя `myCanvas`.

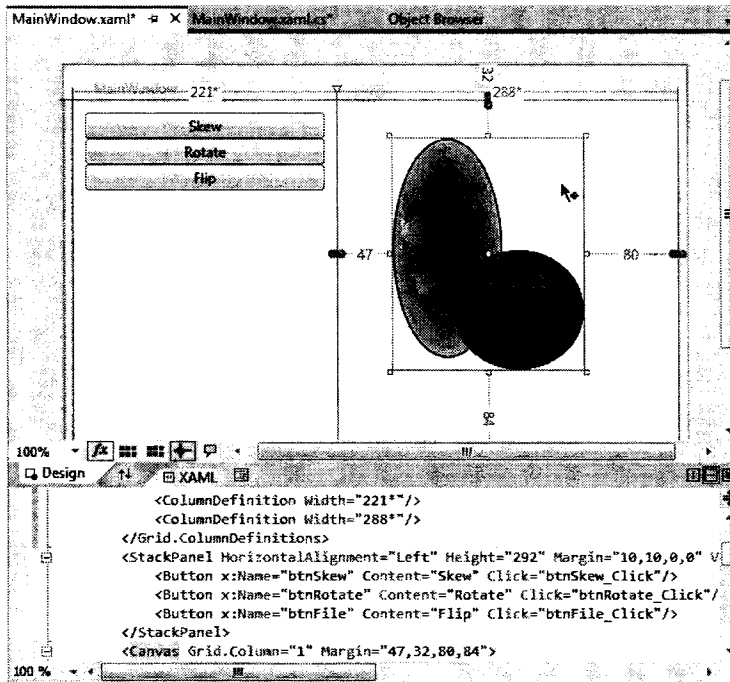


Рис. 29.13. Компоновка для примера с трансформациями

Применение трансформаций на этапе проектирования

Как упоминалось ранее, Visual Studio предоставляет встроенный редактор трансформаций, который можно найти в окне Properties. Раскройте раздел Transform (Трансформация) и найдите области `RenderTransform` и `LayoutTransform` редактора трансформаций (рис. 29.14).

Подобно разделу Brush (Кисть), раздел Transform (Трансформация) предоставляет несколько вкладок, предназначенных для конфигурирования различных типов графической трансформации текущего выбранного элемента. Предлагаемые этими вкладками (указанными в порядке слева направо) варианты трансформации описаны в табл. 29.6.

Опробуйте каждую из описанных трансформаций, используя специальную фигуру в качестве цели (для отмены выполненной операции просто нажимайте `<Ctrl+Z>`). Как и многие другие аспекты раздела Transform окна Properties, каждая трансформация имеет уникальный набор опций конфигурации, которые должны стать вполне понятными, как только вы просмотрите их. Например, редактор трансформации Skew позволяет устанавливать значения перекоса X и Y, а редактор трансформации Flip дает возможность зеркально отображать относительно оси X или Y и т.д.

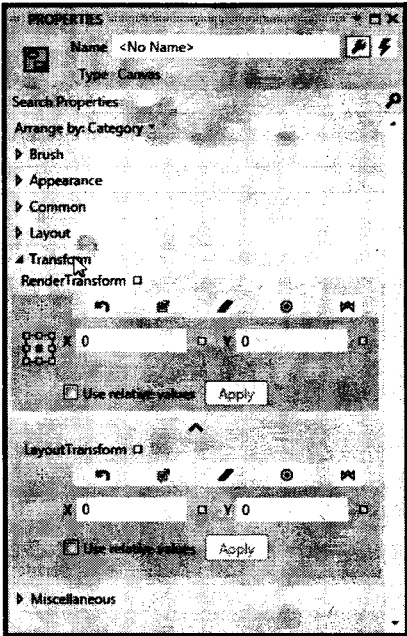


Рис. 29.14. Редактор трансформаций

Таблица 29.6. Варианты трансформации

Трансформация	Описание
Translate (Трансляция)	Позволяет переместить местоположение элемента в позицию X, Y
Rotate (Поворот)	Позволяет повернуть элемент на угол 360 градусов
Scale (Масштабирование)	Позволяет увеличить или уменьшить элемент на масштабный коэффициент в направлениях X и Y
Skew (Перекашивание)	Позволяет перекосить ограничивающий прямоугольник, содержащий выбранный элемент, на масштабный коэффициент в направлениях X и Y
Center Point (Центральная точка)	При повороте или зеркальном отображении объекта элемент перемещается относительно фиксированной точки, которая называется центральной точкой объекта. По умолчанию центральная точка объекта расположена в центре объекта; тем не менее, эта трансформация позволяет изменять центральную точку объекта для выполнения поворота или зеркального отображения относительно другой точки
Flip (Зеркальное отображение)	Позволяет зеркально отобразить выбранный элемент на основе координаты X или Y центральной точки

Трансформация холста в коде

Реализации всех обработчиков событий Click будут более или менее похожими. Мы сконфигурируем объект трансформации и присвоим его объекту myCanvas. Таким образом, после запуска приложения можно щелкать на кнопке и наблюдать результат примененной трансформации. Ниже приведен полный код всех обработчиков событий (обратите внимание на установку свойства LayoutTransform; это позволяет данным фигуры позиционироваться относительно родительского контейнера):

```
private void btnFlip_Click(object sender, System.Windows.RoutedEventArgs e)
{
    myCanvas.LayoutTransform = new ScaleTransform(-1, 1);
}

private void btnRotate_Click(object sender, System.Windows.RoutedEventArgs e)
{
    myCanvas.LayoutTransform = new RotateTransform(180);
}

private void btnSkew_Click(object sender, System.Windows.RoutedEventArgs e)
{
    myCanvas.LayoutTransform = new SkewTransform(40, -20);
}
```

Исходный код. Проект FunWithTransformations доступен в подкаталоге Chapter 29.

Визуализация графических данных с использованием рисунков и геометрий

Хотя типы Shape позволяют генерировать любые интерактивные двухмерные поверхности, из-за насыщенной цепочки наследования они потребляют довольно много памяти. И хотя класс Path может помочь снизить накладные расходы за счет использования включенной геометрии (вместо огромной коллекции других фигур), в WPF предлагается развитый API-интерфейс рисования и геометрии, который позволяет визуализировать даже более легковесные двухмерные векторные изображения.

Входной точкой в этот API-интерфейс является абстрактный класс System.Windows.Media.Drawing (из сборки PresentationCore.dll), который сам по себе всего лишь определяет ограничивающий прямоугольник для хранения визуализаций. Обратите внимание, что на рис. 29.15 цепочка наследования класса Drawing существенно меньше, чем у Shape, учитывая, что в ней отсутствуют как UIElement, так и FrameworkElement.

WPF предлагает различные классы, которые расширяют Drawing; каждый из них представляет определенный способ рисования содержимого, как описано в табл. 29.7.

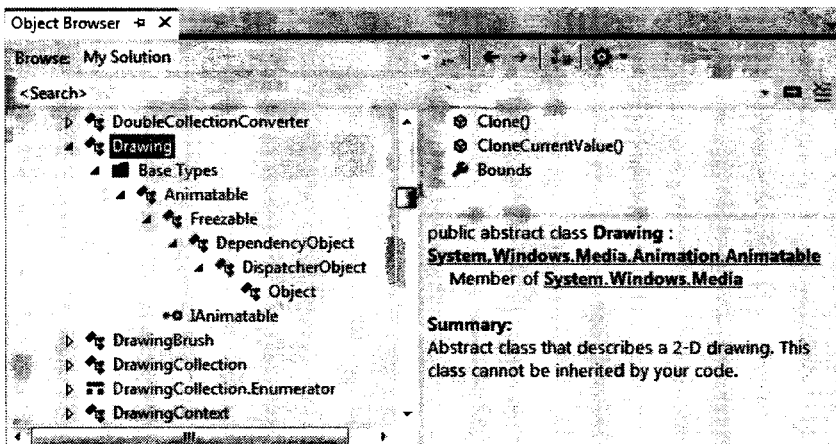


Рис. 29.15. Класс Drawing является более легковесным, чем Shape

Таблица 29.7. Классы, производные от Drawing

Класс	Описание
DrawingGrdoup	Используется для комбинирования коллекции отдельных объектов, производных от Drawing, в единую составную визуализацию
GeometryDrawing	Используется для визуализации двумерных фигур в очень легковесной манере
GlyphRunDrawing	Используется для визуализации текстовых данных с применением служб графической визуализации WPF
ImageDrawing	Используется для визуализации файла изображения, или набора геометрий, внутри ограничивающего прямоугольника
VideoDrawing	Используется для воспроизведения аудио- или видеофайла. Этот тип может полноценно применяться только в процедурном коде. Для воспроизведения видео в XAML-разметке лучше подойдет класс MediaPlayer

Будучи более легковесными, производные от Drawing типы не обладают встроенной возможностью обработки событий, т.к. они не являются UIElement или FrameworkElement (хотя допускают программную реализацию логики проверки попадания).

Другое ключевое отличие между типами, производными от Drawing, и типами, производными от Shape, состоит в том, что производные от Drawing типы не умеют визуализировать себя, поскольку не наследуются от UIElement! Для отображения содержимого производные типы должны помещаться в какой-нибудь контейнерный объект (в частности — DrawingImage, DrawingBrush или DrawingVisual).

Объект DrawingImage позволяет помещать рисунки и геометрии внутри элемента управления Image из WPF, который обычно используется для отображения данных из внешнего файла. DrawingBrush позволяет строить кисть на основе рисунков и их геометрий, чтобы установить свойство, требующее кисти. Наконец, DrawingVisual применяется только на “визуальном” уровне графической визуализации, полностью управляемом из кода C#.

Хотя использовать рисунки немного сложнее, чем простые фигуры, отделение графической композиции от графической визуализации делает производные от Drawing типы намного более облегченными, чем типы, производные от Shape, сохраняя их ключевые службы.

Построение кисти DrawingBrush с использованием геометрий

Ранее в этой главе элемент Path заполнялся группой геометрий следующим образом:

```
<Path Fill = "Orange" Stroke = "Blue" StrokeThickness = "3">
  <Path.Data>
    <GeometryGroup>
      <EllipseGeometry Center = "75,70"
        RadiusX = "30" RadiusY = "30" />
      <RectangleGeometry Rect = "25,55 100 30" />
      <LineGeometry StartPoint="0,0" EndPoint="70,30" />
      <LineGeometry StartPoint="70,30" EndPoint="0,30" />
    </GeometryGroup>
  </Path.Data>
</Path>
```

В этом случае достигается интерактивность Path при чрезвычайной легковесности, присущей геометриям. Однако если необходимо визуализировать аналогичный вывод, но никакая (готовая) интерактивность не нужна, тот же элемент <GeometryGroup> можно поместить внутри DrawingBrush, как показано ниже:

```
<DrawingBrush>
  <DrawingBrush.Drawing>
    <GeometryDrawing>
      <GeometryDrawing.Geometry>
        <GeometryGroup>
          <EllipseGeometry Center = "75,70"
                                RadiusX = "30" RadiusY = "30" />
          <RectangleGeometry Rect = "25,55 100 30" />
          <LineGeometry StartPoint="0,0" EndPoint="70,30" />
          <LineGeometry StartPoint="70,30" EndPoint="0,30" />
        </GeometryGroup>
      </GeometryDrawing.Geometry>
    </GeometryDrawing>
    <!-- Специальное перо для рисования границ -->
    <GeometryDrawing.Pen>
      <Pen Brush="Blue" Thickness="3"/>
    </GeometryDrawing.Pen>
    <!-- Специальная кисть для заполнения внутренней области -->
    <GeometryDrawing.Brush>
      <SolidColorBrush Color="Orange"/>
    </GeometryDrawing.Brush>
  </GeometryDrawing>
</DrawingBrush.Drawing>
</DrawingBrush>
```

При помещении группы геометрий в DrawingBrush также нужно установить объект Pen, используемый для рисования границ, поскольку свойство Stroke больше не наследуется от базового класса Shape. Здесь был создан элемент <Pen> с теми же настройками, которые использовались в значениях Stroke и StrokeThickness из предыдущего примера Path.

Более того, поскольку свойство Fill больше не наследуется из класса Shape, нужно также применять синтаксис “элемент-свойство” для определения объекта кисти, предназначенного элементу <DrawingGeometry>; в данном случае это кисть сплошного оранжевого цвета, как в предыдущих установках Path.

Рисование с помощью DrawingBrush

Теперь кисть DrawingBrush можно использовать для установки значения любого свойства, требующего объекта кисти. Например, подготовив следующую разметку в редакторе Кахамл, с помощью синтаксиса “элемент-свойство” можно рисовать изображение по всей поверхности Page:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Page.Background>
    <!-- Та же кисть DrawingBrush, что и раньше -->
    <DrawingBrush>
      ...
    </DrawingBrush>
  </Page.Background>
</Page>
```

Или же кисть <DrawingBrush> можно применять для установки другого совместимого с кистью свойства, такого как свойство Background элемента Button:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Button Height="100" Width="100">
    <Button.Background>
      <!-- Та же кисть DrawingBrush, что и раньше -->
      <DrawingBrush>
        ...
      </DrawingBrush>
    </Button.Background>
  </Button>
</Page>
```

Независимо от того, для какого совместимого с кистью свойства будет указан специальный объект <DrawingBrush>, визуализировать двухмерное графическое изображение получится с намного меньшими накладными расходами, чем в случае фигур.

Включение типов Drawing в DrawingImage

Тип DrawingImage позволяет включать рисованную геометрию в элемент управления <Image> из WPF. Взгляните на следующую разметку:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Image Height="100" Width="100">
    <Image.Source>
      <DrawingImage>
        <DrawingImage.Drawing>
          <GeometryDrawing>
            <GeometryDrawing.Geometry>
              <GeometryGroup>
                <EllipseGeometry Center = "75,70"
                  RadiusX = "30" RadiusY = "30" />
                <RectangleGeometry Rect = "25,55 100 30" />
                <LineGeometry StartPoint="0,0" EndPoint="70,30" />
                <LineGeometry StartPoint="70,30" EndPoint="0,30" />
              </GeometryGroup>
            </GeometryDrawing.Geometry>
          <!-- Специальное перо для рисования границ -->
          <GeometryDrawing.Pen>
            <Pen Brush="Blue" Thickness="3"/>
          </GeometryDrawing.Pen>
          <!-- Специальная кисть для заполнения внутренней области -->
          <GeometryDrawing.Brush>
            <SolidColorBrush Color="Orange"/>
          </GeometryDrawing.Brush>
        </GeometryDrawing>
      </DrawingImage.Drawing>
    </DrawingImage>
  </Image.Source>
</Image>
</Page>
```

В этом случае элемент <GeometryDrawing> был помещен в <DrawingImage>, а не в <DrawingBrush>. Используя этот элемент <DrawingImage>, можно установить свойство Source элемента управления Image.

Роль инструмента Expression Design

Вы наверняка согласитесь с утверждением, что художнику будет довольно трудно построить сложное векторное изображение с помощью инструментов и приемов, предлагаемых средой Visual Studio. К счастью, инструмент Microsoft Expression Design позволяет создавать более сложные графические данные и затем экспортировать их в разнообразные файловые форматы, включая XAML. А данные, экспортированные в XAML, очень легко импортировать в WPF. После этого можно программно работать с полученной объектной моделью, используя Visual Studio.

На заметку! В следующем примере приложения будет проиллюстрирована возможность экспорта графических данных в виде XAML и импорта этой разметки в новое WPF-приложение Visual Studio. Не стоит беспокоиться, если вы не располагаете копией Expression Design. Необходимый файл `bear_paper.xaml` находится в подкаталоге Chapter 29 в загружаемом коде примеров для этой книги.

Экспорт файла с примером графики в виде XAML

Перед тем, как сложные графические данные можно будет импортировать в новое WPF-приложение, их потребуется сгенерировать. Запустите Expression Design. При наличии навыков создания графики можете нарисовать для этого примера что-нибудь свое. Иначе просто выберите пункт меню `Help` → `Samples` (Справка → Образцы). Здесь вы найдете несколько разных файлов `*.design`, доступных для загрузки, таких как `bear_paper.design` (рис. 29.16).

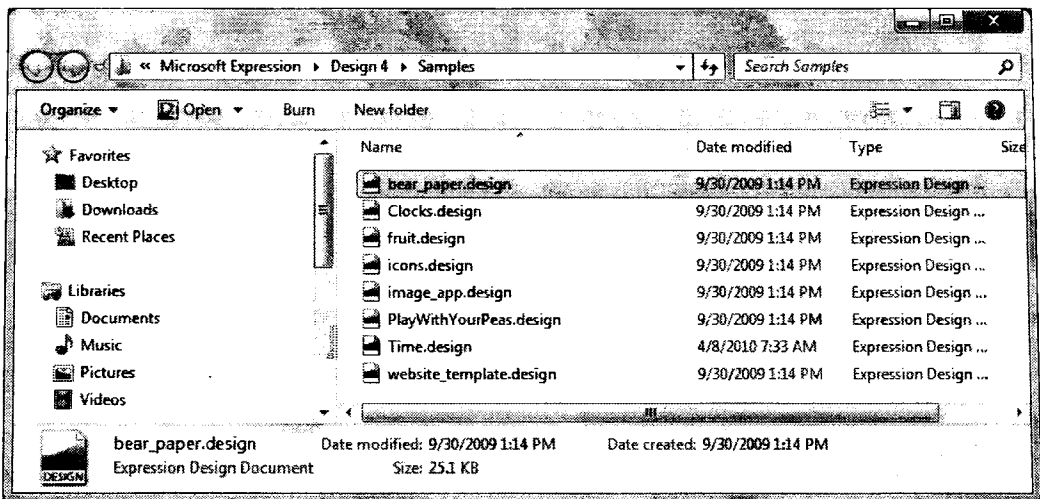


Рис. 29.16. Образец графики с плюшевым медвежонком

На заметку! Полное описание инструмента Expression Design выходит за рамки настоящей книги. Если вы заинтересованы в изучении Expression Design, откройте руководство пользователя через меню `Help` (Справка).

Теперь вы должны видеть часть изображения плюшевого медвежонка в панели рисования Expression Design. Нажмите комбинацию клавиш `<Ctrl+A>` для выбора всего изображения и скорректируйте его размеры, чтобы оно умещалось в окне просмотра, как показано на рис. 29.17.

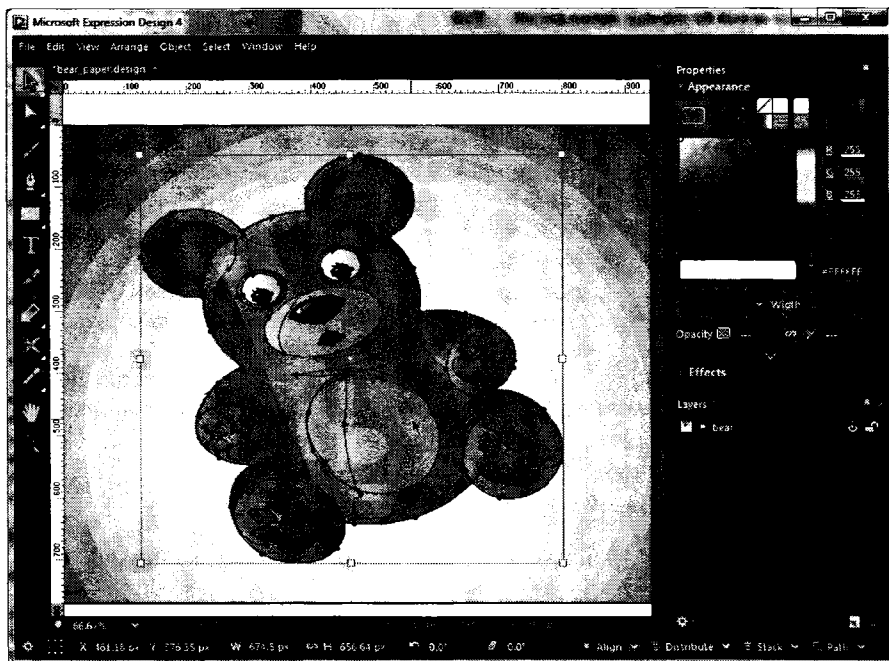


Рис. 29.17. Изменение размеров графики с плюшевым медвежонком

Завершив изменение размера, данные (или часть данных) можно экспортировать с помощью пункта меню **File**⇒**Export...** (**Файл**⇒**Экспорт...**). В окне списка **Format** (**Формат**) вы увидите множество популярных файловых форматов, но для рассматриваемого примера необходим вариант **XAML Silverlight 4/WPF Canvas** (рис. 29.18).

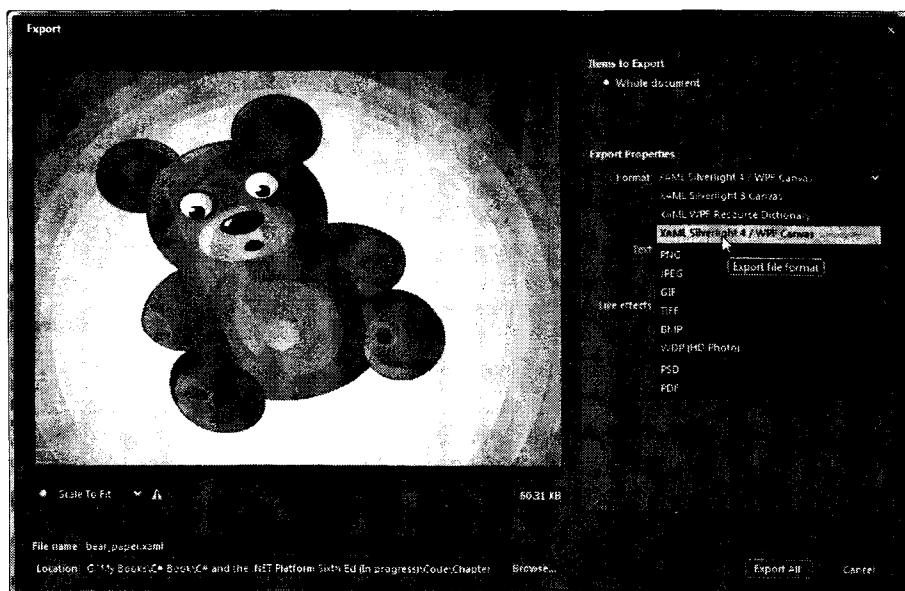


Рис. 29.18. Экспортирование образца графики с плюшевым медвежонком в виде XAML

После выбора подходящего файлового формата XAML снимите отметку с флажка **Always name objects** (Всегда именовать объекты). Вспомните, что когда элемент имеет атрибут `x:Name`, с ним можно взаимодействовать в коде. Однако эта графика с плюшевым медвежонком будет описана с применением большого числа элементов XAML.

Если указать инструменту на необходимость именования каждого возможного объекта, в результате может получиться огромное количество переменных-членов, добавленных в кодовую базу C#, которые в действительности использоваться не будут (и это также приведет к увеличению размера конечного скомпилированного приложения).

Кроме того, удостоверьтесь в выборе такого местоположения для экспортируемых данных (`bear_paper.xaml`), которое впоследствии будет легко найти (скажем, рабочий стол Windows). Для всех остальных параметров можно оставить стандартные установки. Как только все будет готово, щелкните на кнопке **Export All** (Экспортировать все), после чего изображение плюшевого медвежонка экспортируется в виде XAML. Теперь инструмент Expression Design можно закрыть.

Импорт графических данных в проект WPF

В этот момент можно создать в Visual Studio новое WPF-приложение (по имени `InteractiveTeddyBear`). Сделав это, измените значения `Height` и `Width` в первоначальном элементе `Window`, как показано ниже, и удалите исходный элемент управления `Grid`:

```
<Window x:Class="InteractiveTeddyBear.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="824" Width="1056">
</Window>
```

Теперь выберите пункт меню **Project**⇒**Add Existing Item** (Проект⇒Добавить существующий элемент) и в открывшемся диалоговом окне перейдите к местоположению, в котором находится экспортированный файл `bear_paper.xaml`. Щелкните на кнопке **Add** (Добавить), после чего этот XAML-файл будет добавлен в проект (в этом можно убедиться с помощью окна `Solution Explorer`). Дважды щелкните на этом файле, чтобы загрузить хранящиеся в нем графические данные в визуальный конструктор Visual Studio.

Если заглянуть в окно `Document Outline` для файла `bear_paper.xaml`, вы увидите, что учтены абсолютно все элементы XAML. Цель заключается в том, чтобы найти у медвежонка *левое глазное яблоко* и *правое ухо* и назначить этим элементам имена. Хотя можно было бы поискать эти объекты в коде (что будет весьма утомительным), лучше щелкнуть на них в визуальном конструкторе. Это приведет к автоматическому выделению нужного узла в окне `Document Outline`.

Выберите левое глазное яблоко и в окне `Properties` назначьте этому объекту имя `leftEye`. Выберите часть правого уха (любую, какую хотите) и назначьте ей имя `rightEar`. Базовый процесс показан на рис. 29.19.

Далее скопируйте весь XAML-элемент `Canvas` в буфер обмена. В этот момент можно закрыть файл `bear_paper.xaml`. Переключитесь обратно на визуальный конструктор для первоначального элемента `Window` и вставьте данные `Canvas` между открывающим и закрывающим дескрипторами `<Window>`. Вы должны увидеть изображение плюшевого медвежонка внутри главного окна приложения. При желании можете подкорректировать позицию изображения в диспетчере компоновки. Тем не менее, поскольку левый глаз и правое ухо видны, все готово к взаимодействию с этими объектами в коде.



Рис. 29.19. Нахождение и именование объектов

Взаимодействие с объектами изображения

Теперь можно обрабатывать события для объектов `leftEye` и `rightEar`. Для этого выберите объект в визуальном конструкторе, перейдите на вкладку **Events** (События) в окне **Properties** и введите имена требуемых обработчиков событий. В текущем примере обработайте событие `MouseLeftButtonDown` для каждого объекта, указывая каждый раз уникальное имя метода.

Ниже приведен простой код C#, который будет изменять внешний вид каждого объекта после щелчка на нем (если не хотите вводить весь показанный здесь код, можете лишь поместить в обработчики вызовы `MessageBox.Show()` с отображением подходящих сообщений).

```
private void leftEye_MouseLeftButtonDown(object sender,
    System.Windows.Input.MouseButtonEventArgs e)
{
    // Изменить цвет глаза при щелчке на нем.
    leftEye.Fill = new SolidColorBrush(Colors.Red);
}

private void rightEar_MouseLeftButtonDown(object sender,
    System.Windows.Input.MouseButtonEventArgs e)
{
    // Размыть ухо при щелчке на нем.
    System.Windows.Media.Effects.BlurEffect blur =
        new System.Windows.Media.Effects.BlurEffect();
    blur.Radius = 10;
    rightEar.Effect = blur;
}
```

Запустите приложение. Щелкните на левом глазе и правом ухе плюшевого медвежонка и наблюдайте за эффектами.

Теперь вы должны понимать процесс импортирования сложных графических данных, созданных с помощью Expression Design, в проект Visual Studio и, что более важно, уметь взаимодействовать с этими графическими данными в коде. Вы наверняка согласитесь с исключительной важностью того факта, что профессиональные художники имеют возможность генерации сложных графических данных и экспортирования их в виде XAML. После того как графические данные получены, разработчики могут импортировать разметку и взаимодействовать с объектной моделью в коде.

Исходный код. Проект InteractiveTeddyBear доступен в подкаталоге Chapter 29.

Визуализация графических данных с использованием визуального уровня

Последний вариант визуализации графических данных с помощью WPF называется *визуальным уровнем*. Как уже упоминалось, доступ к этому уровню возможен только из кода (он не поддерживает XAML). Хотя для подавляющего большинства WPF-приложений вполне достаточно фигур, рисунков и геометрий, визуальный уровень обеспечивает самый быстрый способ визуализации крупного объема графических данных. Как ни странно, он также может быть полезен, когда необходимо визуализировать единственное изображение на очень большой площади. Например, если задача состоит в заполнении фона окна простым статическим изображением, то визуальный уровень оказывается наиболее быстрым способом сделать это. Кроме того, он удобен, когда требуется очень быстро менять фон окна в зависимости от ввода пользователя или еще чего-нибудь.

Давайте построим небольшой пример программы, иллюстрирующей основы использования визуального уровня.

Базовый класс Visual и производные дочерние классы

Абстрактный класс `System.Windows.Media.Visual` предлагает минимальный набор служб (визуализацию, проверку попадания, трансформации) для визуализации графики, но не предусматривает поддержки дополнительных не визуальных служб, которые могут приводить к разбуханию кода (события ввода, службы компоновки, стили и привязка данных). Обратите внимание на простую цепочку наследования для типа `Visual`, показанную на рис. 29.20.

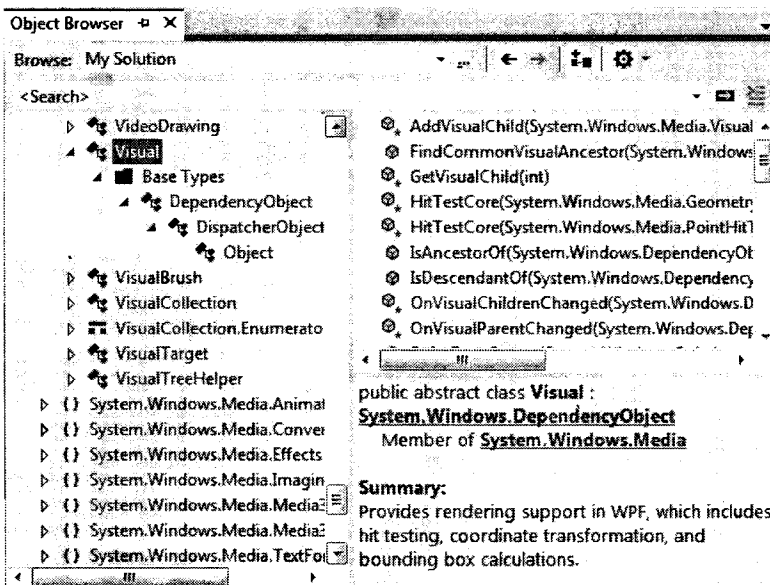


Рис. 29.20. Класс `Visual` предоставляет базовую проверку попадания, трансформацию координат и вычисления ограничивающих прямоугольников

Учитывая, что `Visual` — это абстрактный базовый класс, для выполнения действительных операций визуализации должен использоваться один из его производных классов. В WPF определено несколько подклассов `Visual`, включая `DrawingVisual`, `Viewport3DVisual` и `ContainerVisual`.

В рассматриваемом ниже примере внимание сосредоточено только на `DrawingVisual` — легковесном классе рисования, который применяется для визуализации фигур, изображений или текста.

Первый взгляд на класс `DrawingVisual`

Чтобы визуализировать данные на поверхности с помощью класса `DrawingVisual`, потребуется выполнить следующие базовые шаги:

- получить объект `DrawingContext` от `DrawingVisual`;
- использовать объект `DrawingContext` для визуализации графических данных.

Эти два шага представляют собой абсолютный минимум того, что нужно предпринять для визуализации некоторых данных на поверхности. Однако если необходимо, чтобы визуализируемые графические данные сами отвечали за определение проверки попадания (что может быть важно для взаимодействия с пользователем), тогда понадобится выполнить следующие дополнительные шаги:

- обновить логическое и визуальное деревья, поддерживаемые контейнером, на поверхности которого производится рисование;
- переопределить два виртуальных метода из класса `FrameworkElement`, позволив контейнеру получать созданные визуальные данные.

Давайте немного задержимся на последних двух шагах. Чтобы посмотреть, как использовать класс `DrawingVisual` для визуализации двумерных данных, создайте в Visual Studio новое приложение WPF по имени `RenderingWithVisuals`. Наша первая цель — применение `DrawingVisual` для динамического присваивания данных элементу управления `Image` из WPF. Начните со следующего обновления XAML-разметки окна:

```
<Window x:Class="RenderingWithVisuals.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Fun with the Visual Layer" Height="350" Width="525"
        . Loaded="Window_Loaded" WindowStartupLocation="CenterScreen">
    <StackPanel Background="AliceBlue" Name="myStackPanel">
        <Image Name="myImage" Height="80"/>
    </StackPanel>
</Window>
```

Обратите внимание, что элемент управления `<Image>` пока не имеет значения для `Source`, поскольку оно будет установлено во время выполнения. Кроме того, предусмотрен обработчик события `Loaded` окна, который выполняет работу по построению графических данных в памяти с использованием объекта `DrawingBrush`. Ниже показана реализация обработчика события `Loaded`:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    const int TextFontSize = 30;
    // Создать объект System.Windows.Media.FormattedText.
    FormattedText text = new FormattedText("Hello Visual Layer!",
        new System.Globalization.CultureInfo("en-us"),
        FlowDirection.LeftToRight,
        new Typeface(this.FontFamily, FontStyles.Italic,
            FontWeights.DemiBold, FontStretches.UltraExpanded),
```

```

    TextFontSize,
    Brushes.Green);

// Создать DrawingVisual и получить DrawingContext.
DrawingVisual drawingVisual = new DrawingVisual();
using(DrawingContext drawingContext = drawingVisual.RenderOpen())
{
    // Вызвать любой из методов DrawingContext для визуализации данных.
    drawingContext.DrawRoundedRectangle(Brushes.Yellow, new Pen(Brushes.Black, 5),
        new Rect(5, 5, 450, 100), 20, 20);
    drawingContext.DrawText(text, new Point(20, 20));
}

// Динамически создать битовое изображение, используя данные из DrawingVisual.
RenderTargetBitmap bmp = new RenderTargetBitmap(500, 100, 100, 90,
    PixelFormats.Pbgra32);
bmp.Render(drawingVisual);

// Установить источник для элемента управления Image.
myImage.Source = bmp;
}

```

В этом коде представлен ряд новых классов WPF, которые кратко описываются ниже (более подробные сведения по ним можно получить в документации .NET Framework 4.5 SDK). Метод начинается с создания нового объекта `FormattedText`, представляющего текстовую часть изображения в памяти, которое мы конструируем. Как видите, конструктор позволяет указывать многочисленные атрибуты, вроде размера шрифта, семейства шрифтов, цвета переднего плана и самого текста.

Затем получается необходимый объект `DrawingContext` через вызов `RenderOpen()` на экземпляре `DrawingVisual`. Здесь в `DrawingVisual` визуализирован цветной, со скругленными углами прямоугольник, за которым следует форматированный текст. В обоих случаях графические данные помещаются в `DrawingVisual` с использованием жестко закодированных значений, что не слишком хорошая идея для реального приложения, но вполне сойдет для простого теста.

На заметку! Обязательно ознакомьтесь с описанием класса `DrawingContext` в документации .NET Framework 4.5 SDK, чтобы просмотреть все члены, связанные с визуализацией. Если вы работали в прошлом с объектом `Graphics` из `Windows Forms`, то `DrawingContext` должен выглядеть очень похожим.

Последние несколько операторов отображают `DrawingVisual` на объект `RenderTargetBitmap`, который является членом пространства имен `System.Windows.Media.Imaging`. Этот класс принимает визуальный объект и трансформирует его в находящееся в памяти битовое изображение. После этого устанавливается свойство `Source` элемента управления `Image` и получается вывод, показанный на рис. 29.21.

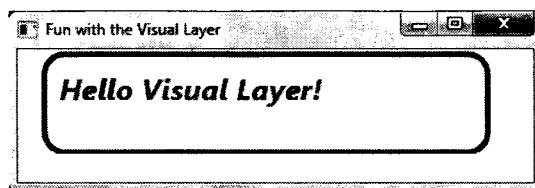


Рис. 29.21. Использование визуального уровня для отображения находящегося в памяти битового изображения

На заметку! Пространство имен `System.Windows.Media.Imaging` содержит множество дополнительных классов кодирования, которые позволяют сохранять находящийся в памяти объект `RenderTargetBitmap` в физический файл в различных форматах. Дополнительную информацию ищите в `JpegBitmapEncoder` и связанных с ним классах.

Визуализация графических данных в специальном диспетчере компоновки

Хотя использование `DrawingVisual` для рисования на фоне элемента управления WPF представляет интерес, скорее всего, чаще придется строить специальный диспетчер компоновки (`Grid`, `StackPanel`, `Canvas` и т.п.), внутри которого для визуализации содержимого применяется визуальный уровень. После создания такой специальный диспетчер компоновки можно включать в нормальный элемент `Window` (или `Page`, или `UserControl`). В таком случае часть пользовательского интерфейса будет обрабатываться высоко оптимизированным агентом визуализации, а для визуализации некритичных аспектов включающего элемента `Window` будут использоваться фигуры и рисунки.

Если дополнительная функциональность, предлагаемая выделенным диспетчером компоновки, не требуется, можно просто расширить `FrameworkElement`, который уже имеет необходимую инфраструктуру, позволяющую хранить внутри также и визуальные элементы. Давайте рассмотрим пример. Вставьте в проект новый класс по имени `CustomVisualFrameworkElement`. Унаследуйте его от `FrameworkElement` и импортируйте пространства имен `System.Windows`, `System.Windows.Input` и `System.Windows.Media`. Этот класс будет поддерживать переменную-член типа `VisualCollection`, содержащую два фиксированных объекта `DrawingVisual` (конечно, можно было бы добавить члены в эту коллекцию, но ради простоты это не делается). Модифицируйте класс `CustomVisualFrameworkElement` следующим образом:

```
class CustomVisualFrameworkElement : FrameworkElement
{
    // Коллекция всех визуальных объектов.
    VisualCollection theVisuals;
    public CustomVisualFrameworkElement()
    {
        // Заполнить коллекцию VisualCollection несколькими объектами DrawingVisual.
        // Аргумент конструктора представляет владельца визуальных объектов.
        theVisuals = new VisualCollection(this);
        theVisuals.Add(AddRect());
        theVisuals.Add(AddCircle());
    }
    private Visual AddCircle()
    {
        DrawingVisual drawingVisual = new DrawingVisual();
        // Получить DrawingContext для создания нового содержимого.
        using (DrawingContext drawingContext = drawingVisual.RenderOpen())
        {
            // Создать круг и нарисовать его в DrawingContext.
            Rect rect = new Rect(new Point(160, 100), new Size(320, 80));
            drawingContext.DrawEllipse(Brushes.DarkBlue, null, new Point(70, 90), 40, 50);
        }
        return drawingVisual;
    }
    private Visual AddRect()
    {
        DrawingVisual drawingVisual = new DrawingVisual();
```

```

using (DrawingContext drawingContext = drawingVisual.RenderOpen())
{
    Rect rect = new Rect(new Point(160, 100), new Size(320, 80));
    drawingContext.DrawRectangle(Brushes.Tomato, null, rect);
}
return drawingVisual;
}
}

```

Перед тем как можно будет использовать специальный элемент `FrameworkElement` в `Window`, потребуется переопределить два виртуальных метода, упомянутых ранее, которые оба вызываются внутренне WPF во время процесса визуализации. Метод `GetVisualChild()` возвращает дочерний элемент по указанному индексу из коллекции дочерних элементов. Свойство `VisualChildrenCount`, предназначенное только для чтения, возвращает количество визуальных дочерних элементов внутри этой визуальной коллекции. Оба метода легко реализовать, поскольку реальную работу можно делегировать переменной-члену `VisualCollection`:

```

protected override int VisualChildrenCount
{
    get { return theVisuals.Count; }
}

protected override Visual GetVisualChild(int index)
{
    // Значение должно быть больше нуля, поэтому на всякий случай проверить.
    if (index < 0 || index >= theVisuals.Count)
    {
        throw new ArgumentOutOfRangeException();
    }
    return theVisuals[index];
}

```

Теперь имеется достаточно функциональности, чтобы протестировать специальный класс. Обновите XAML-описание элемента `Window`, добавив в существующий контейнер `StackPanel` один объект `CustomVisualFrameworkElement`. Это потребует создания специального пространства имен XML, которое отображается на пространство имен .NET (см. главу 28).

```

<Window x:Class="RenderingWithVisuals.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:custom="clr-namespace:RenderingWithVisuals"
    Title="Fun with the Visual Layer" Height="350" Width="525"
    Loaded="Window_Loaded" WindowStartupLocation="CenterScreen">
    <StackPanel Background="AliceBlue" Name="myStackPanel">
        <Image Name="myImage" Height="80"/>
        <custom:CustomVisualFrameworkElement/>
    </StackPanel>
</Window>

```

Результат выполнения программы показан на рис. 29.22.

Реагирование на операции проверки попадания

Поскольку `DrawingVisual` не поддерживает инфраструктуру `UIElement` или `FrameworkElement`, необходимо программно добавить возможность вычисления операций проверки попадания. К счастью, на визуальном уровне сделать это очень просто, благодаря концепции логического и визуального деревьев.

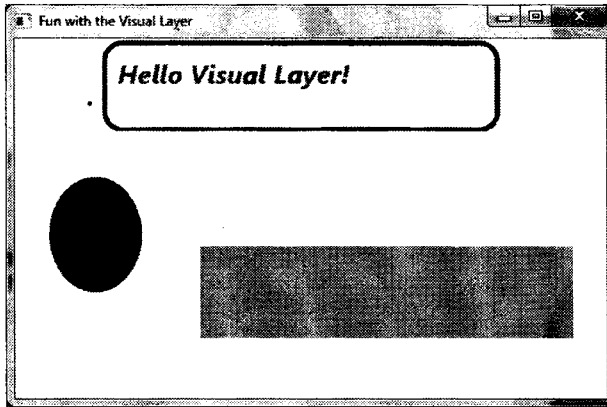


Рис. 29.22. Использование визуального уровня для визуализации данных в специальном элементе `FrameworkElement`

Оказывается, что в результате написания блока XAML, по сути, строится логическое дерево элементов. Однако за каждым логическим деревом стоит намного более развитое описание, известное как визуальное дерево, которое содержит низкоуровневые инструкции визуализации.

Упомянутые деревья подробно рассматриваются в главе 31, а сейчас достаточно знать, что до тех пор, пока специальные визуальные объекты не будут зарегистрированы в этих структурах данных, операции проверки попадания выполнять невозможно. К счастью, контейнер `VisualCollection` делает это автоматически (что объясняет, почему необходимо передавать ссылку на специальный элемент `FrameworkElement` в качестве аргумента конструктора).

Модифицируйте класс `CustomVisualFrameworkElement` для обработки события `MouseDown` в конструкторе класса, используя стандартный синтаксис C#:

```
this.MouseDown += MyVisualHost_MouseDown;
```

Реализация этого обработчика вызовет метод `VisualTreeHelper.HitTest()`, чтобы проверить нахождение курсора мыши в границах одного из отображенных визуальных объектов. Для этого в качестве параметра метода `HitTest()` указывается делегат `HitTestResultCallback`, который выполнит вычисления. Добавьте в класс `CustomVisualFrameworkElement` следующие методы:

```
void MyVisualHost_MouseDown(object sender, MouseButtonEventArgs e)
{
    // Найти место, где пользователь выполнил щелчок.
    Point pt = e.GetPosition((UIElement)sender);

    // Вызвать вспомогательную функцию через делегат.
    VisualTreeHelper.HitTest(this, null,
        new HitTestResultCallback(myCallback), new PointHitTestParameters(pt));
}

public HitTestResultBehavior myCallback(HitTestResult result)
{
    // Если щелчок был совершен на визуальном объекте,
    // переключаться между перекошенной и нормальной визуализацией.
    if (result.VisualHit.GetType() == typeof(DrawingVisual))
    {
        if (((DrawingVisual)result.VisualHit).Transform == null)
        {
            ((DrawingVisual)result.VisualHit).Transform = new SkewTransform(7, 7);
        }
    }
}
```

```

else
{
    ((DrawingVisual)result.VisualHit).Transform = null;
}
}

// Сообщить методу HitTest() об останове углубления в визуальное дерево.
return HitTestResultBehavior.Stop;
}

```

Запустите программу. Теперь должна появиться возможность выполнения щелчка на любом из отображенных визуальных объектов и наблюдения за трансформацией в действии. Хотя это очень простой пример работы с визуальным уровнем WPF, помните, что можно использовать те же самые кисти, трансформации, перья и диспетчеры компоновки, что и при работе с XAML. Это значит, что вы уже знаете достаточно много о работе классов, производных от `Visual`.

Исходный код. Проект `RenderingWithVisuals` доступен в подкаталоге `Chapter 29`.

На этом исследование служб графической визуализации `Windows Presentation Foundation` завершено. В действительности мы лишь слегка коснулись обширной области графических средств WPF. Дальнейшее изучение фигур, рисунков, кистей, трансформаций и визуальных объектов продолжайте самостоятельно (некоторые дополнительные детали на эту тему будут приводиться в оставшихся главах, посвященных WPF).

Резюме

Поскольку `Windows Presentation Foundation` является чрезвычайно насыщенным графикой API-интерфейсом для построения графических пользовательских интерфейсов, не удивительно, что существует несколько способов визуализации графического вывода. В начале главы были рассмотрены все три подхода к визуализации (фигуры, рисунки и визуальные объекты) вместе с разнообразными примитивами визуализации, такими как кисти, перья и трансформации.

Вспомните, что когда нужно построить интерактивную двухмерную визуализацию, фигуры делают этот процесс очень простым. Однако статические, не интерактивные изображения могут визуализироваться более оптимально с применением рисунков и геометрии. Визуальный уровень (доступный только из кода) обеспечивает максимальную степень контроля и высокую производительность.

Ресурсы, анимация и стили WPF

В этой главе будут представлены три важных и взаимосвязанных темы, которые позволят углубить понимание API-интерфейса Windows Presentation Foundation (WPF). Первая тема касается изучения роли *логических ресурсов*. Как вы увидите, система логических ресурсов (также называемых *объектными ресурсами*) — это способ ссылаться на часто используемые объекты внутри WPF-приложения. Хотя логические ресурсы часто пишутся на XAML, они могут быть определены и в процедурном коде.

Затем будет показано, как определять, выполнять и управлять последовательностью анимации. Вопреки тому, что можно было подумать, применение анимации WPF не ограничено видеоиграми или мультимедиа-приложениями. В API-интерфейсе WPF анимация может использоваться, например, для подсветки кнопки, когда она получает фокус, или увеличения размера выбранной строки в DataGrid. Освоение анимации является ключевым аспектом построения специальных шаблонов элементов управления (о чем пойдет речь в главе 31).

Завершается глава рассмотрением роли стилей WPF. Подобно веб-странице, которая использует стили CSS или механизм тем ASP.NET, приложение WPF может определять общий вид и поведение набора элементов управления. Эти стили определяются в разметке и сохраняются в виде объектных ресурсов для последующего использования, а затем динамически применяются во время выполнения.

Система ресурсов WPF

Нашей первой целью является исследование способов встраивания и доступа к ресурсам приложения. В WPF поддерживаются два вида ресурсов. Первый из них — это *двоичный ресурс*, который обычно включает элементы, обычно рассматриваемые большинством программистов как ресурс в его традиционном смысле (встроенные файлы изображений или звуковых клипов, значки, используемые приложением, и т.д.).

Второй вид, называемый *объектным* или *логическим ресурсом*, представляет именованный объект .NET, который может быть упакован и многократно использован по всему приложению. Хотя любой объект .NET может быть упакован в виде объектного ресурса, логические ресурсы особенно полезны при работе с графическими данными любого рода, учитывая, что можно определять часто применяемые графические примитивы (кисти, перья, анимации и т.п.) и ссылаться на них по мере необходимости.

Работа с двоичными ресурсами

Прежде чем приступить к теме объектных ресурсов, давайте быстро посмотрим, как упаковывать такие двоичные ресурсы, как значки и файлы изображений (например, логотипы компаний или изображения для анимации), в приложения. Создайте в Visual Studio новое WPF-приложение по имени BinaryResourcesApp. Модифицируйте разметку начального окна для использования DockPanel в качестве корня компоновки:

```
<Window x:Class="BinaryResourcesApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Fun with Binary Resources" Height="500" Width="649">

  <DockPanel LastChildFillI="True">
  </DockPanel>
</Window>
```

Теперь предположим, что приложение должно отображать один из трех файлов изображений внутри части окна, в зависимости от пользовательского ввода. Элемент управления Image из WPF может применяться не только для отображения типичных файлов изображений (*.bmp, *.gif, *.ico, *.jpg, *.png, *.wdp или *.tiff), но также данных из DrawingImage (как было показано в главе 29). Постройте пользовательский интерфейс окна на основе диспетчера компоновки DockPanel с простой панелью инструментов, включающей кнопки Next (Вперед) и Previous (Назад). Ниже панели инструментов можно расположить элемент управления Image, который пока не имеет значения, установленного в свойстве Source, т.к. это будет делаться в коде:

```
<Window x:Class="BinaryResourcesApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Fun with Binary Resources" Height="500" Width="649">
  <DockPanel LastChildFill="True">
    <ToolBar Height="60" Name="picturePickerToolBar" DockPanel.Dock="Top">
      <Button x:Name="btnPreviousImage" Height="40" Width="100" BorderBrush="Black"
        Margin="5" Content="Previous" Click="btnPreviousImage_Click"/>
      <Button x:Name="btnNextImage" Height="40" Width="100" BorderBrush="Black"
        Margin="5" Content="Next" Click="btnNextImage_Click"/>
    </ToolBar>

    <!-- Этот элемент Image будет заполняться в коде -->
    <Border BorderThickness="2" BorderBrush="Green">
      <Image x:Name="imageHolder" Stretch="Fill" />
    </Border>
  </DockPanel>
</Window>
```

Обратите внимание, что событие Click обрабатывается для каждого объекта Button. Предполагая, что для обработки этих событий использовалась IDE-среда, в файле кода C# будут определены два пустых метода. Каким образом реализовать обработчики событий Click для организации цикла по графическим данным? И что более важно: графические данные будут храниться на пользовательском жестком диске или же будут встроены в скомпилированную сборку? Давайте рассмотрим оба варианта.

Включение в проект файлов свободных ресурсов

Предположим, что изображения должны поставляться в виде набора файлов внутри подкаталога пути установки приложения. В окне Solution Explorer среды Visual Studio щелкните правой кнопкой мыши на узле проекта и выберите в контекстном меню пункт Add⇒New Folder (Добавить⇒Новая папка) для создания папки по имени Images.

Теперь щелкните правой кнопкой мыши на папке Images и выберите в контекстном меню пункт Add⇒Existing Item (Добавить⇒Существующий элемент), чтобы скопировать в нее файлы изображений. В исходном коде для этого проекта доступны три файла изображений с именами Welcome.jpg, Dogs.jpg и Deer.jpg, которые можно включить в проект; или же можно просто добавить три файла изображений по своему выбору. На рис. 30.1 показан текущий вид окна Solution Explorer.

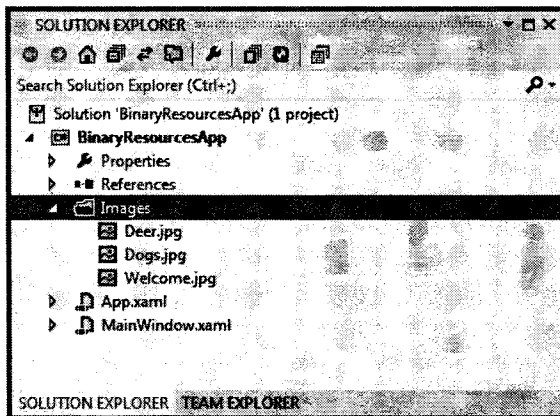


Рис. 30.1. Новый подкаталог в проекте WPF, содержащий данные изображений

Конфигурирование свободных ресурсов

Если необходимо, чтобы IDE-среда Visual Studio копировала содержимое проекта в выходной каталог, потребуется скорректировать несколько настроек, используя окно Properties (Свойства). Чтобы обеспечить копирование содержимого папки Images в папку \bin\Debug, начните с выбора всех изображений в Solution Explorer. Когда все изображения выбраны, в окне Properties установите свойство Build Action (Действие сборки) в Content (Содержимое), а свойство Copy Output Directory (Копировать в выходной каталог) в Copy always (Копировать всегда), как показано на рис. 30.2.

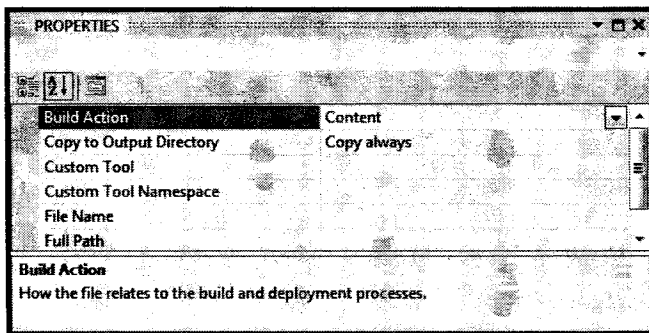


Рис. 30.2. Конфигурирование изображений для копирования в выходной каталог

Перекомпилировав программу, щелкните на кнопке Show all Files (Показать все файлы) в Solution Explorer и просмотрите скопированную папку Images в каталоге \bin\Debug (может потребоваться также щелкнуть на кнопке обновления). Результат показан на рис. 30.3.

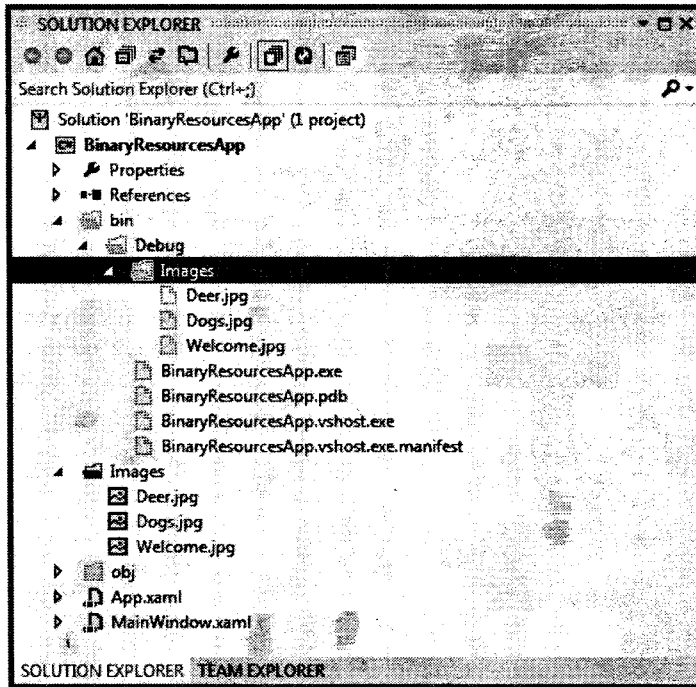


Рис. 30.3. Скопированные данные изображений

Программная загрузка изображения

В WPF доступен класс по имени `BitmapImage`, входящий в пространство имен `System.Windows.Media.Imaging`. Этот класс позволяет загружать данные из файла изображения, местоположение которого представлено объектом `System.Uri`. Обработав событие `Loaded` окна, можно заполнить список `List<T>` элементов `BitmapImages` следующим образом:

```
public partial class MainWindow : Window
{
    // Список файлов BitmapImage.
    List<BitmapImage> images = new List<BitmapImage>();

    // Текущая позиция в списке.
    private int currImage = 0;
    private const int MAX_IMAGES = 2;
    private void Window_Loaded(object sender, RoutedEventArgs e)
    {
        try
        {
            string path = Environment.CurrentDirectory;

            // Загрузить эти изображения при загрузке окна.
            images.Add(new BitmapImage(new Uri(string.Format(@"{0}\Images\Deer.jpg", path))));
            images.Add(new BitmapImage(new Uri(string.Format(@"{0}\Images\Dogs.jpg", path))));
            images.Add(new BitmapImage(new Uri(string.Format(@"{0}\Images\Welcome.jpg",
                path))));

            // Показать первое изображение в List<>.
            imageHolder.Source = images[currImage];
        }
    }
}
```

```

        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
        }
    }
    ...
}

```

Обратите внимание, что в этом классе также определена переменная-член типа `int` (`currImage`), которая позволяет обработчикам событий `Click` проходить циклом по каждому элементу в `List<T>` и отображать его в элементе `Image`, устанавливая свойство `Source`. (Здесь обработчик события `Loaded` устанавливает свойство `Source` в первое изображение из `List<T>`.) Вдобавок константа `MAX_IMAGES` дает возможность проверить верхнюю и нижнюю границы по мере итерации по списку. Ниже показаны обработчики `Click`, которые делают это:

```

private void btnPreviousImage_Click(object sender, RoutedEventArgs e)
{
    if (--currImage < 0)
        currImage = MAX_IMAGES;
    imageHolder.Source = images[currImage];
}
private void btnNextImage_Click(object sender, RoutedEventArgs e)
{
    if (++currImage > MAX_IMAGES)
        currImage = 0;
    imageHolder.Source = images[currImage];
}

```

Теперь можно запустить программу и переключаться между всеми изображениями.

Встраивание ресурсов приложения

Если вы предпочитаете встроить файлы изображений непосредственно в сборку .NET в виде двоичных ресурсов, выберите файлы изображений в `Solution Explorer` (в папке `\Images`, а не `\bin\Debug\Images`). Затем установите свойство `Build Action` в `Resource` (Ресурс), а свойство `Copy to Output Directory` — в `Do not copy` (Не копировать), как показано на рис. 30.4.

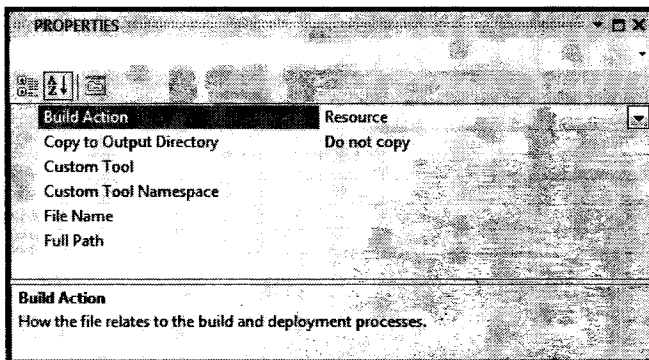


Рис. 30.4. Конфигурирование изображений как встроенных ресурсов

В меню `Build` (Сборка) среды `Visual Studio` выберите пункт `Clean Solution` (Очистить решение) для очистки текущего содержимого `\bin\Debug\Images` и повторно скомпилируйте проект. Обновите окно `Solution Explorer` и обратите внимание на отсутствие

данных в каталоге `\bin\Debug\Images`. При текущих параметрах сборки графические данные больше не копируются в выходную папку, а встраиваются в саму сборку.

Теперь необходимо модифицировать код для загрузки изображений, чтобы извлекать их из скомпилированной сборки:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    try
    {
        images.Add(new BitmapImage(new Uri(@"Images/Deer.jpg", UriKind.Relative)));
        images.Add(new BitmapImage(new Uri(@"Images/Dogs.jpg", UriKind.Relative)));
        images.Add(new BitmapImage(new Uri(@"Images/Welcome.jpg", UriKind.Relative)));
        imageHolder.Source = images[currImage];
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

Как видите, в этом случае больше не нужно определять путь установки, а можно просто просматривать ресурсы по имени, которое учитывает имя исходного подкаталога. Также обратите внимание, что при создании объектов `Uri` указывается значение `UriKind.Relative`. В любом случае в нынешнем виде исполняемая программа представляет собой автономную сущность, которая может быть запущена из любого местоположения на машине, поскольку все скомпилированные данные находятся внутри сборки. На рис. 30.5 показано готовое приложение.

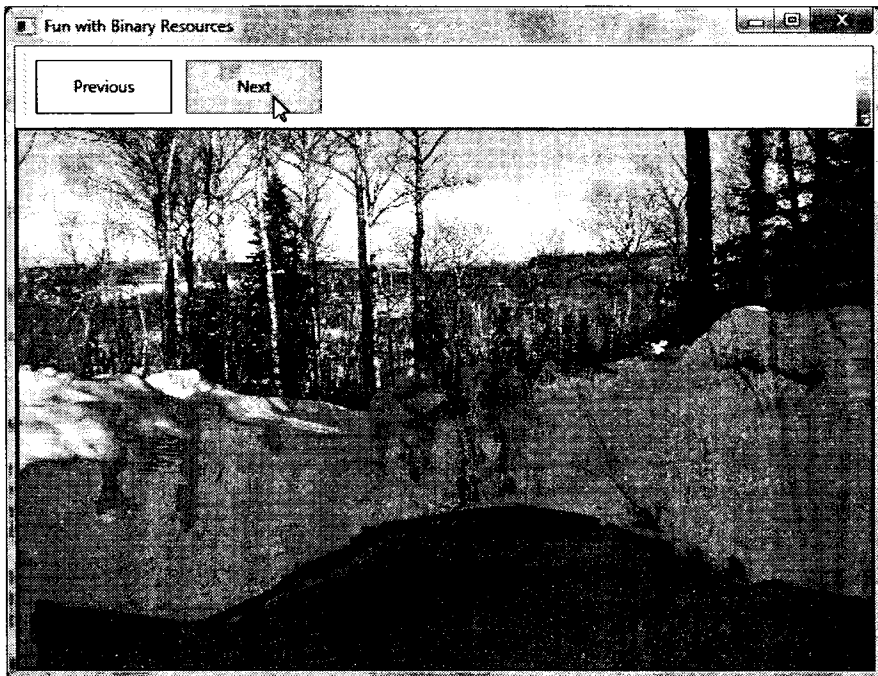


Рис. 30.5. Простое приложение для просмотра изображений

Работа с объектными (логическими) ресурсами

При построении WPF-приложения очень часто приходится определять большой объем XAML-разметки для использования во множестве мест окна или, возможно, в нескольких окнах или проектах. Например, предположим, что создана идеальная кисть линейного градиента, описываемая 10 строками кода разметки. Теперь эту кисть необходимо использовать в качестве фонового цвета для каждого элемента Button в проекте, состоящем из 8 окон, т.е. всего получается 16 элементов Button.

Худшее, что можно предпринять в такой ситуации — это копировать и вставлять одну и ту же XAML-разметку для каждого элемента управления Button. Ясно, что это может стать кошмаром при сопровождении, поскольку придется вносить изменения во множестве мест всякий раз, когда нужно скорректировать внешний вид и поведение кисти.

К счастью, *объектные ресурсы* позволяют определить фрагмент XAML-разметки, назначить ему имя и сохранить в подходящем словаре для последующего использования. Подобно бинарному ресурсу, объектные ресурсы часто компилируются в сборку, которая нуждается в них. Однако при этом не требуется оперировать свойством Build Action. Достаточно поместить XAML-разметку в правильное место, а об остальном позаботится компилятор.

Манипуляции объектными ресурсами составляют значительную часть разработки WPF. Как будет показано, объектные ресурсы могут быть намного сложнее, чем специальная кисть. Можно определять анимацию на основе XAML, трехмерную визуализацию, специальный стиль элемента управления, шаблон данных и многое другое — и все это в одном многократно используемом ресурсе.

Роль свойства Resources

Как упоминалось ранее, объектные ресурсы должны помещаться в подходящий объект словаря, чтобы их можно было использовать во всем приложении. Как известно, каждый потомок FrameworkElement поддерживает свойство Resources. Это свойство инкапсулирует объект ResourceDictionary, содержащий определенные объектные ресурсы. ResourceDictionary может хранить элементы любого типа, поскольку оперирует экземплярами System.Object, и допускает манипуляции из XAML или процедурного кода.

В WPF все элементы управления, окна (Window), страницы (Page), используемые при построении навигационных приложений или программ ХВАР, и элементы UserControl расширяют класс FrameworkElement, поэтому почти все виджеты предоставляют доступ к объекту ResourceDictionary. Более того, класс Application, хотя и не расширяет FrameworkElement, но поддерживает свойство с идентичным именем Resources, которое предназначено для тех же целей.

Определение ресурсов уровня окна

Чтобы приступить к исследованию роли объектных ресурсов, создайте приложение WPF по имени ObjectResourcesApp, используя для этого Visual Studio, и замените начальный контейнер Grid горизонтальным выровненным диспетчером компоновки StackPanel. В этом StackPanel определите два элемента управления Button:

```
<Window x:Class="ObjectResourcesApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Fun with Object Resources" Height="350" Width="525">
```

```

<StackPanel Orientation="Horizontal">
  <Button Margin="25" Height="200" Width="200" Content="OK" FontSize="20"/>
  <Button Margin="25" Height="200" Width="200" Content="Cancel" FontSize="20"/>
</StackPanel>
</Window>

```

Теперь выберите кнопку ОК и укажите для свойства цвета Background специальный тип кисти с помощью интегрированного редактора кистей (который обсуждался в главе 29).

После этого обратите внимание, что кисть встраивается в контекст дескрипторов `<Button>` и `</Button>`:

```

<Button Margin="25" Height="200" Width="200" Content="OK" FontSize="20">
  <Button.Background>
    <RadialGradientBrush>
      <GradientStop Color="#FFC44EC4" Offset="0" />
      <GradientStop Color="#FF829CEB" Offset="1" />
      <GradientStop Color="#FF793879" Offset="0.669" />
    </RadialGradientBrush>
  </Button.Background>
</Button>

```

Чтобы позволить кнопке Cancel (Отмена) также пользоваться этой кистью, необходимо распространить область `<RadialGradientBrush>` на ресурсный словарь родительского элемента. Например, если переместить его в `<StackPanel>`, обе кнопки смогут использовать одну и ту же кисть, поскольку они являются дочерними элементами диспетчера компоновки. Более того, можно было бы поместить кисть в ресурсный словарь самого окна, так что все аспекты содержимого окна (вложенные панели, и т.п.) могли бы свободно пользоваться ею.

Когда требуется определить ресурс, для установки свойства Resources владельца применяется синтаксис “свойство-элемент”. Кроме того, элементу ресурса задается значение `x:Key`, которое будет использовано другими частями окна, когда им нужно будет обратиться к объектному ресурсу. Имейте в виду, что `x:Key` и `x:Name` — не одно и то же! Атрибут `x:Name` позволяет получить доступ к объекту как к переменной-члену в файле кода, в то время как атрибут `x:Key` позволяет сослаться на элемент в словаре ресурсов.

Среда Visual Studio позволяет продвинуть ресурс на более высокий уровень с применением соответствующих окон Properties. Чтобы сделать это, сначала необходимо идентифицировать свойство, имеющее сложный объект, который необходимо упаковать в виде ресурса (в рассматриваемом примере это свойство Background). Рядом со свойством находится небольшой квадрат белого цвета, щелчок на котором приводит к открытию всплывающего меню. Выберите в этом меню пункт Convert to New Resource... (Преобразовать в новый ресурс...), как показано на рис. 30.6.

Теперь будет запрошено имя ресурса (`myBrush`) и предложено указать, куда он должен быть помещен. В данном примере оставьте выбор по умолчанию — This document (Этот документ), как показано на рис. 30.7.

Покончив с этим, вы увидите, что разметка будет реструктурирована следующим образом:

```

<Window x:Class="ObjectResourcesApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Fun with Object Resources" Height="350" Width="525">
  <Window.Resources>
    <RadialGradientBrush x:Key="myBrush">
      <GradientStop Color="#FFC44EC4" Offset="0" />
      <GradientStop Color="#FF829CEB" Offset="1" />
    </RadialGradientBrush>
  </Window.Resources>

```

```
<GradientStop Color="#FF793879" Offset="0.669" />
</RadialGradientBrush>
</Window.Resources>

<StackPanel Orientation="Horizontal">
  <Button Margin="25" Height="200" Width="200" Content="OK"
    FontSize="20" Background="{StaticResource myBrush}"></Button>
  <Button Margin="25" Height="200" Width="200" Content="Cancel" FontSize="20"/>
</StackPanel>
</Window>
```

Обратите внимание на новый контекст `<Window.Resources>`, который теперь содержит объект `RadialGradientBrush`, имеющий значение ключа `myBrush`.

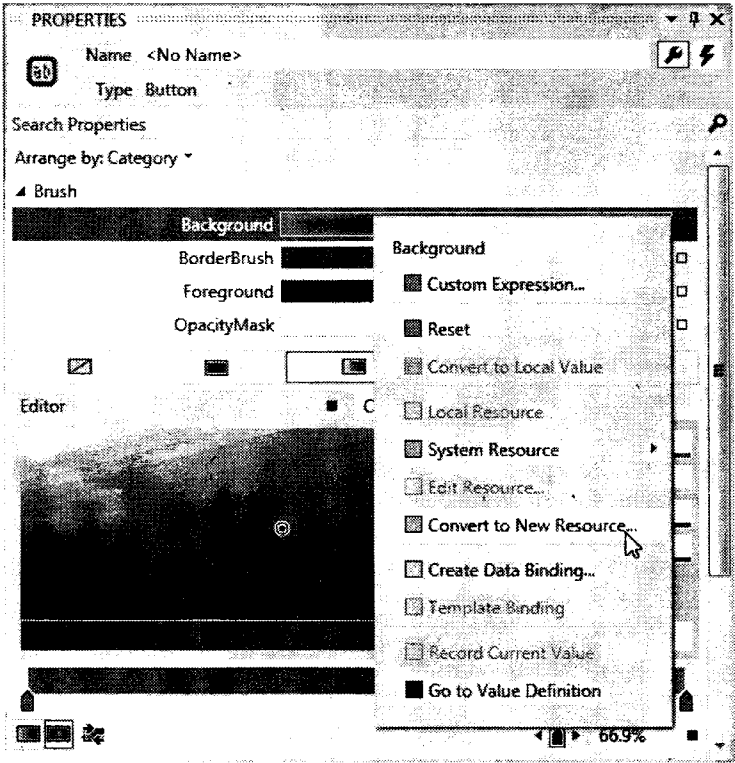


Рис. 30.6. Перемещение сложного объекта в контейнер ресурсов

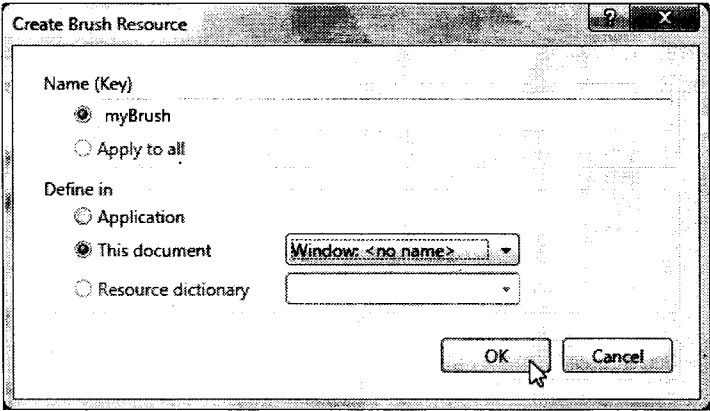


Рис. 30.7. Назначение имени объектному ресурсу

Расширение разметки {StaticResource}

Другое изменение, имеющее место при извлечении объектного ресурса, связано с тем, что свойство, которое было целью извлечения (опять-таки, Background), теперь использует расширение разметки {StaticResource}. Как видите, имя ключа указано в виде аргумента. Если теперь кнопке Cancel нужно будет применять ту же кисть для рисования фона, она сможет это сделать. Или же, если кнопка Cancel имеет некоторое сложное содержимое, то любой подэлемент этой кнопки может также использовать ресурс уровня окна, например, свойство Fill элемента Ellipse:

```
<StackPanel Orientation="Horizontal">
  <Button Margin="25" Height="200" Width="200" Content="OK" FontSize="20"
    Background="{StaticResource myBrush}">
  </Button>
  <Button Margin="25" Height="200" Width="200" FontSize="20">
    <StackPanel>
      <Label HorizontalAlignment="Center" Content="No Way!" />
      <Ellipse Height="100" Width="100" Fill="{StaticResource myBrush}" />
    </StackPanel>
  </Button>
</StackPanel>
```

Расширение разметки {DynamicResource}

При подключении к ключевому ресурсу свойство также может использовать расширение разметки {DynamicResource}. Чтобы понять разницу, назначьте кнопке OK имя btnOK и обработайте ее событие Click. В этом обработчике события воспользуйтесь свойством Resources для получения специальной кисти с последующим изменением некоторых ее аспектов:

```
private void btnOK_Click(object sender, RoutedEventArgs e)
{
    // Получить кисть и внести изменение.
    RadialGradientBrush b = (RadialGradientBrush)Resources["myBrush"];
    b.GradientStops[1] = new GradientStop(Colors.Black, 0.0);
}
```

На заметку! Здесь для поиска ресурса по имени применяется индексатор Resources. Однако имейте в виду, что если ресурс не обнаруживается, это приводит к генерации исключения времени выполнения. Можно также использовать метод TryFindResource(), который не генерирует исключение времени выполнения, а просто возвращает null, если указанный ресурс не найден.

Запустив это приложение и щелкнув на кнопке ОК, вы увидите, что изменение кисти учтено, и каждая кнопка обновляется для визуализации модифицированной кисти. А что если полностью изменить тип кисти, указанной ключом myBrush? Например:

```
private void btnOK_Click(object sender, RoutedEventArgs e)
{
    // Поместить совершенно новую кисть в ячейку myBrush.
    Resources["myBrush"] = new SolidColorBrush(Colors.Red);
}
```

На этот раз после щелчка на кнопке никаких обновлений не ожидается. Причина в том, что расширение разметки {StaticResource} применяет ресурс только однажды и остается "подключенным" к исходному объекту на протяжении времени жизни приложения. Однако если изменить в разметке все вхождения {StaticResource} на {DynamicResource}, обнаружится, что специальная кисть будет заменена кистью со сплошным красным цветом.

По существу расширение разметки {DynamicResource} может обнаруживать замену лежащего в основе ключевого объекта новым. Как и следовало ожидать, это требует некоторой дополнительной инфраструктуры времени выполнения, так что обычно стоит придерживаться использования {StaticResource}, если только не планируется заменять объектный ресурс другим объектом во время выполнения с уведомлением всех элементов, использующих этот ресурс.

Ресурсы уровня приложения

Когда в ресурсном словаре окна имеются объектные ресурсы, то все элементы этого окна могут пользоваться ими, но другие окна приложения — нет. Назначьте кнопке Cancel имя btnCancel и обработайте событие Click. Добавьте в текущий проект новое окно (по имени TestWindow.xaml), содержащее единственную кнопку Button, по щелчку на которой окно закрывается:

```
public partial class TestWindow : Window
{
    public TestWindow()
    {
        InitializeComponent();
    }

    private void btnClose_Click(object sender, RoutedEventArgs e)
    {
        this.Close();
    }
}
```

В обработчике Click кнопки Cancel первого окна загрузите и отобразите это новое окно, как показано ниже:

```
private void btnCancel_Click(object sender, RoutedEventArgs e)
{
    TestWindow w = new TestWindow();
    w.Owner = this;
    w.WindowStartupLocation = WindowStartupLocation.CenterOwner;
    w.ShowDialog();
}
```

Если новое окно желает использовать myBrush, в данный момент оно не может этого сделать, поскольку myBrush не находится внутри корректного “контекста”. Решение состоит в определении объектного ресурса на уровне приложения, а не на уровне конкретного окна. В Visual Studio не предусмотрено какого-либо способа автоматизировать такое действие, поэтому просто вырежьте текущий объект кисти из области <Windows.Resource> и поместите его в область <Application.Resources> файла App.xaml:

```
<Application x:Class="ObjectResourcesApp.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="MainWindow.xaml">

    <Application.Resources>
        <RadialGradientBrush x:Key="myBrush">
            <GradientStop Color="#FFC44EC4" Offset="0" />
            <GradientStop Color="#FF829CEB" Offset="1" />
            <GradientStop Color="#FF793879" Offset="0.669" />
        </RadialGradientBrush>
    </Application.Resources>
</Application>
```

Теперь объект `TestWindow` может применять эту же кисть для рисования своего фона. Найдите свойство `Background` для этого нового объекта `Window` и щелкните на вкладке `Brush resources` (Ресурсы кисти) для просмотра ресурсов уровня приложения (рис. 30.8).

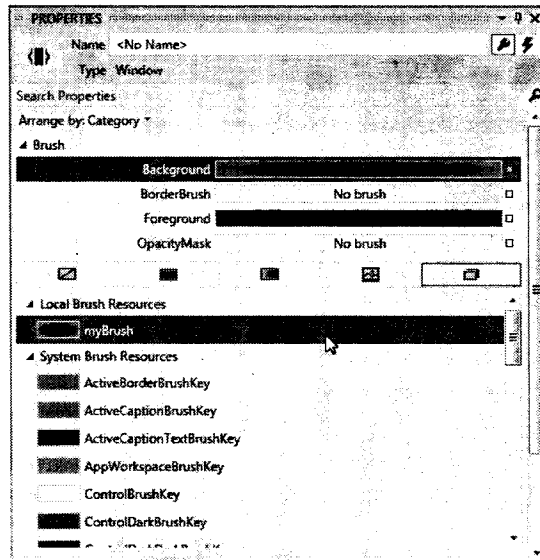


Рис. 30.8. Применение ресурсов уровня приложения

Определение объединенных словарей ресурсов

Ресурсы уровня приложения являются хорошей отправной точкой, но что, если необходимо определить набор сложных (или не слишком сложных) ресурсов, которые должны многократно использоваться во множестве проектов WPF? В этом случае требуется определить то, что известно как *объединенный словарь ресурсов*. Объединенный словарь ресурсов представляет собой всего лишь файл `.xaml`, который не содержит ничего помимо коллекции объектных ресурсов. Один проект может иметь любое необходимое количество таких файлов (один для кистей, один для анимации, и т.д.), каждый из которых может добавляться в диалоговом окне `Add New Item` (Добавить новый элемент), доступном через меню `Project` (рис. 30.9).

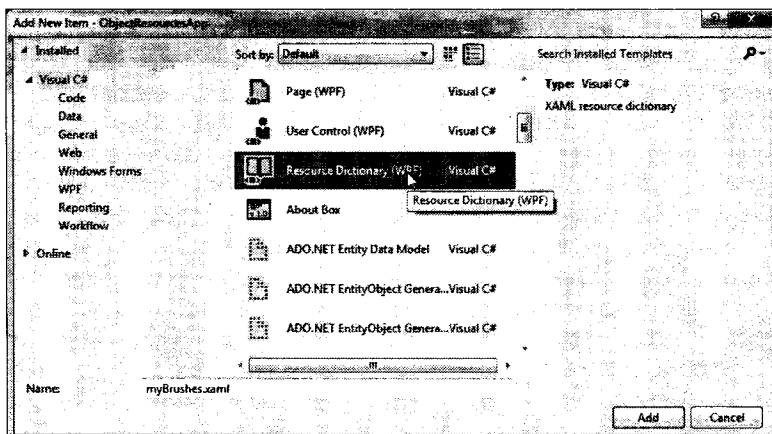


Рис. 30.9. Вставка нового объединенного словаря ресурсов

В новом файле `MyBrushes.xaml` понадобится вырезать текущие ресурсы из контекста `Application.Resources` и перенести их в словарь, как показано ниже:

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

    <RadialGradientBrush x:Key="myBrush">
        <GradientStop Color="#FFC44EC4" Offset="0" />
        <GradientStop Color="#FF829CEB" Offset="1" />
        <GradientStop Color="#FF793879" Offset="0.669" />
    </RadialGradientBrush>

</ResourceDictionary>
```

Несмотря на то что этот словарь ресурсов является частью проекта, по-прежнему возникают ошибки времени выполнения. Причина в том, что все словари ресурсов должны быть объединены (обычно на уровне приложения) в существующий словарь ресурсов. Для этого воспользуйтесь следующим форматом (обратите внимание, что множество словарей ресурсов может быть объединено добавлением нескольких элементов `<ResourceDictionary>` в область `<ResourceDictionary.MergedDictionaries>`):

```
<Application x:Class="ObjectResourcesApp.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="MainWindow.xaml">

    <!-- Взять логические ресурсы из файла MyBrushes.xaml -->

    <Application.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary Source = "MyBrushes.xaml"/>
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Application.Resources>

</Application>
```

Определение сборки, включающей только ресурсы

И последнее (по порядку, но не по важности): существует возможность создавать библиотеки классов, которые не содержат ничего, кроме словарей объектных ресурсов. Это может быть полезно, если определен набор тем, которые должны применяться на уровне машины. Объектные ресурсы можно упаковать в выделенную сборку, и тогда приложения, которые хотят использовать их, смогут загружать их в память.

Самый легкий способ построения сборки из одних ресурсов состоит в том, чтобы начать с проекта WPF User Control Library (Библиотека пользовательских элементов управления WPF). Добавьте такой проект (под названием `MyBrushesLibrary`) к текущему решению, используя пункт меню `Add⇒New Project` (Добавить⇒Новый проект) среды Visual Studio (рис. 30.10).

Теперь полностью удалите файл `UserControl1.xaml` из проекта (единственное, что понадобится — это ссылаемые сборки WPF). Перетащите файл `MyBrushes.xaml` в проект `MyBrushesLibrary` и удалите его из проекта `ObjectResourcesApp`. Окно `Solution Explorer` должно теперь выглядеть, как показано на рис. 30.11.

Скомпилируйте проект WPF User Control Library. Затем установите ссылку на эту библиотеку из проекта `ObjectResourcesApp`, используя диалоговое окно `Add Reference` (Добавить ссылку). Теперь необходимо объединить эти двоичные ресурсы со словарем ресурсов уровня приложения из проекта `ObjectResourcesApp`.

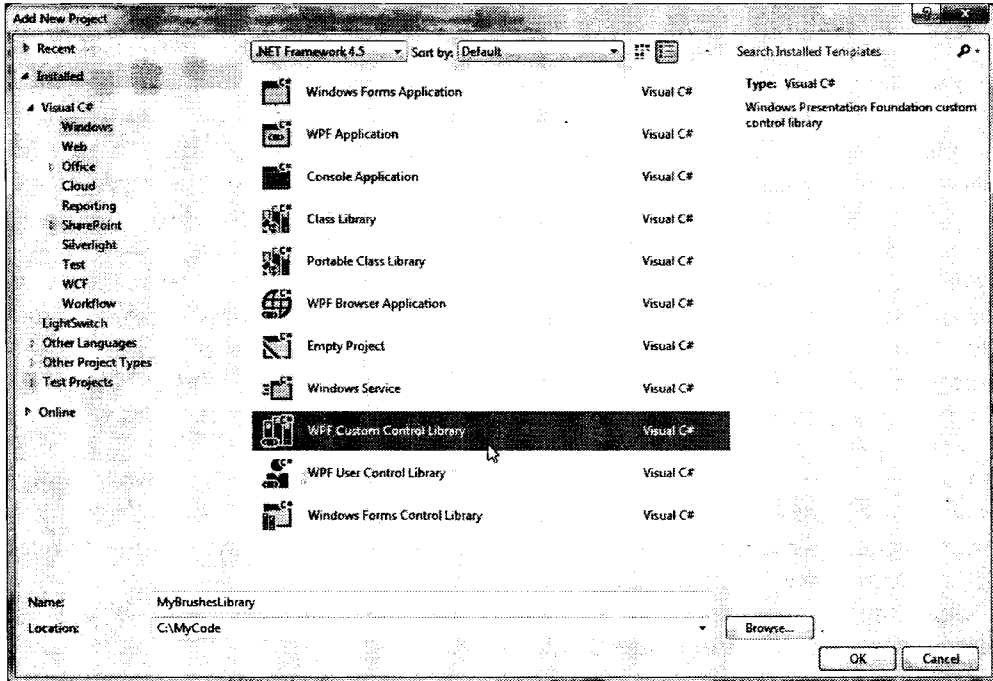


Рис. 30.10. Добавление проекта WPF User Control Library в качестве начальной точки для построения библиотеки из одних только ресурсов

Это потребует некоторого довольно забавного синтаксиса, как показано ниже:

```
<Application.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <!-- Синтаксис: /ИмяСборки;Component/
ИмяфайлаXamlвСборке.xaml -->
      <ResourceDictionary Source =
"/MyBrushesLibrary;Component/MyBrushes.xaml"/>
    </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
</Application.Resources>
```

Имейте в виду, что эта строка чувствительна к пробелам. Если возле двоеточия или косой черты будут лишние пробелы, возникнут ошибки времени выполнения. Первая часть строки — это дружественное имя внешней библиотеки (без расширения файла). После двоеточия идет слово *Component*, а за ним — имя скомпилированного двоичного ресурса, совпадающее с именем исходного словаря ресурсов XAML.

На этом знакомство с системой управления ресурсами WPF завершено. В большинстве приложений довольно часто придется использовать описанные приемы, и мы еще не раз обратимся к ним в оставшихся главах, посвященных WPF. Теперь давайте приступим к исследованию API-интерфейса встроенной анимации Windows Presentation Foundation.

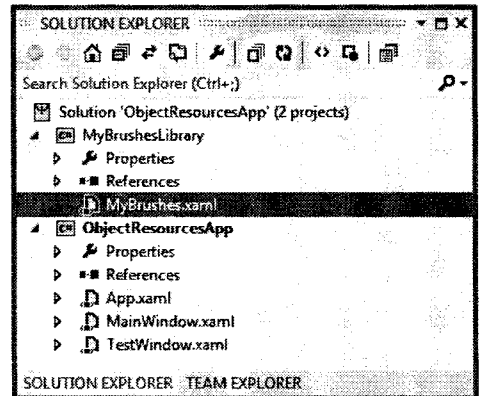


Рис. 30.11. Перемещение файла *MyBrushes.xaml* в новый проект библиотеки

Службы анимации WPF

В дополнение к службам графической визуализации, которые рассматривались в главе 29, в WPF предлагается API-интерфейс для поддержки служб анимации. С термином *анимация* многие связывают вращающийся логотип компании, последовательность сменяемых друг друга изображений (создающих иллюзию движения), бегущую по экрану строку текста либо программу специфического типа, подобную видеоигре или мультимедиа-приложению.

Хотя API-интерфейсы анимации WPF определенно могли бы использоваться в упомянутых выше целях, анимация может применяться всегда, когда приложению необходимо придать дополнительный лоск. Например, можно построить анимацию для кнопки на экране, которая слегка увеличивается, когда курсор мыши попадает в ее границы (и уменьшается обратно, когда курсор покидает ее поверхность). Или же можно анимировать окно, чтобы оно закрывалось, используя определенное визуальное представление, например, постепенно исчезая до полной прозрачности. Фактически поддержка анимации WPF может применяться в приложениях любого рода (бизнес-приложениях, мультимедиа-программах, видеоиграх и т.д.), когда нужно создать более выгодное впечатление у пользователя.

Как и многие другие аспекты WPF, понятие анимации не представляет собой ничего нового. Новостью является только то, что в отличие от других API-интерфейсов, которые вы могли использовать в прошлом (включая Windows Forms), разработчики не обязаны создавать необходимую инфраструктуру вручную. В среде WPF нет необходимости создавать заранее фоновые потоки или таймеры, чтобы выполнять анимированные последовательности, определять специальные типы для представления анимации, очищать и перерисовывать изображения, либо заниматься утомительными математическими вычислениями. Как и другие аспекты WPF, анимацию можно построить целиком в XAML-разметке, целиком в коде C#, либо за счет комбинации того и другого.

На заметку! В среде Visual Studio отсутствует поддержка построения анимации с помощью графических инструментов. В Visual Studio для создания анимации придется набирать XAML-разметку вручную. Тем не менее, инструмент Expression Blend обладает встроенным редактором анимации, который может существенно упростить решение таких задач.

Роль классов анимации

Чтобы разобраться в поддержке анимации WPF, необходимо начать с рассмотрения классов анимации из пространства имен `System.Windows.Media.Animation` сборки `PresentationCore.dll`. Здесь можно найти свыше 100 разных типов классов, которые содержат в своем имени слово `Animation`.

Все эти классы могут быть отнесены к одной из трех обширных категорий. Во-первых, любой класс, который следует соглашению об именовании вида *ТипДанныхAnimation* (`ByteAnimation`, `ColorAnimation`, `DoubleAnimation`, `Int32Animation` и т.д.), позволяет работать с анимацией линейной интерполяцией. Она позволяет гладко изменять значение во времени, от начального к конечному.

Во-вторых, классы, которые следуют соглашению об именовании вида *ТипДанныхAnimationUsingKeyFrames* (`StringAnimationUsingKeyFrames`, `DoubleAnimationUsingKeyFrames`, `PointAnimationUsingKeyFrames` и т.д.), представляют анимацию ключевыми кадрами, которая позволяет проходить в цикле по набору определенных значений за указанный период времени. Например, ключевые кадры можно использовать для замены надписи на кнопке, проходя в цикле по последовательности индивидуальных символов.

Наконец, в-третьих, классы, которые следуют соглашению об именовании вида *ТипДанныхAnimationUsingPath* (*DoubleAnimationUsingPath*, *PointAnimationUsingPath* и т.п.) представляют анимацию на основе пути, которая позволяет анимировать объекты, перемещая их по определенному пути. Например, при построении приложения глобального позиционирования (GPS) можно использовать анимацию на основе пути, чтобы перемещать элемент по кратчайшему пути к указанному пользователем месту.

Теперь очевидно, что эти классы не позволяют каким-то образом предоставить последовательность анимации непосредственно переменной определенного типа (в конце концов, анимировать значение 9 с помощью *Int32Animation* невозможно).

Например, рассмотрим свойства *Height* и *Width* типа *Label*. Оба они являются свойствами зависимости, упаковывающими значение *double*. Чтобы определить анимацию, которая будет увеличивать ширину метки с течением времени, можно подключить объект *DoubleAnimation* к свойству *Height* и позволить WPF позаботиться о деталях выполнения анимации. А вот другой пример: если требуется изменить цвет кисти с зеленого на желтый в течение 5 секунд, это можно сделать с использованием анимации *ColorAnimation*.

Следует прояснить, что классы *Animation* могут подключаться к любому свойству зависимости определенного объекта, который соответствует лежащему в основе типу. Как будет показано в главе 31, свойства зависимости — это специальная разновидность свойств, применяемых многими службами WPF, включая анимацию, привязку данных и стили.

По соглашению свойство зависимости определено как статическое, доступное только для чтения поле класса, чье имя формируется добавлением слова *Property* к нормальному имени свойства. Например, свойство зависимости для свойства *Height* класса *Button* будет доступно в коде как *Button.HeightProperty*.

Свойства *To*, *From* и *By*

Во всех классах *Animation* определено несколько ключевых свойств, которые управляют начальным и конечным значениями, используемыми для выполнения анимации:

- *To* — представляет конечное значение анимации;
- *From* — представляет начальное значение анимации;
- *By* — представляет общую величину, на которую анимация изменяет начальное значение.

Несмотря на тот факт, что все классы поддерживают свойства *To*, *From* и *By*, они не получают их через виртуальные члены базового класса. Причина в том, что лежащие в основе типы, упакованные в эти свойства, варьируются в широких пределах (целочисленные, цвета, объекты *Thickness* и т.п.), и представление всех возможностей через единственный базовый класс привело бы к очень сложным кодовым конструкциям.

В связи с этим может также возникнуть вопрос: почему бы ни воспользоваться обобщениями .NET для определения единственного обобщенного класса анимации с единственным параметром типа (например, *Animate<T>*)? Опять-таки, учитывая, что существует огромное количество типов данных (цвета, векторы, целые числа, строки и т.д.), применяемых для анимации свойств зависимости, это решение не будет столь ясным, как можно было бы ожидать (не говоря уже о том, что XAML предлагает лишь ограниченную поддержку обобщенных типов).

Роль базового класса Timeline

Хотя для определения виртуальных свойств To, From и By не использовался единственный базовый класс, классы Animation все же разделяют общий базовый класс: System.Windows.Media.Animation.Timeline. Этот тип предоставляет множество дополнительных свойств, которые управляют темпом анимации (табл. 30.1).

Таблица 30.1. Основные свойства базового класса Timeline

Свойство	Описание
AccelerationRatio, DecelerationRatio, SpeedRatio	Эти свойства могут использоваться для управления общим темпом анимационной последовательности
AutoReverse	Это свойство получает и устанавливает значение, которое указывает, должна ли временная шкала воспроизводиться в обратном направлении после завершения прямой итерации (стандартным значением является false)
BeginTime	Это свойство получает и устанавливает время запуска временной шкалы. По умолчанию принято значение 0, что запускает анимацию немедленно
Duration	Это свойство позволяет устанавливать продолжительность воспроизведения временной шкалы
FileBehavior, RepeatBehavior	Эти свойства используются для управления тем, что случится по завершении временной шкалы (повторение анимации, ничего и т.д.)

Реализация анимации в коде C#

Мы построим окно, которое содержит элемент Button, обладающий довольно странным поведением: когда на него наводится курсор мыши, он вращается вокруг своего левого верхнего угла. Начните с создания нового WPF-приложения по имени SpinningButtonAnimationApp в Visual Studio. Измените начальную разметку, как показано ниже (обратите внимание на обработку события MouseEnter кнопки):

```
<Window x:Class="SpinningButtonAnimationApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Animations in C# code" Height="350"
  Width="525" WindowStartupLocation="CenterScreen">
  <Grid>
    <Button x:Name="btnSpinner" Height="50" Width="100" Content="I Spin!"
      MouseEnter="btnSpinner_MouseEnter"/>
  </Grid>
</Window>
```

Теперь импортируйте пространство имен System.Windows.Animation и добавьте следующий код в файл C#:

```
public partial class MainWindow : Window
{
    private bool isSpinning = false;

    private void btnSpinner_MouseEnter(object sender, MouseEventArgs e)
    {
        if (!isSpinning)
        {
            isSpinning = true;
        }
    }
}
```

```

// Создать объект анимации double и зарегистрировать
// с событием Completed.
DoubleAnimation dblAnim = new DoubleAnimation();
dblAnim.Completed += (o, s) => { isSpinning = false; };

// Установить начальное и конечное значения.
dblAnim.From = 0;
dblAnim.To = 360;

// Создать объект RotateTransform и присвоить
// его свойству RenderTransform кнопки.
RotateTransform rt = new RotateTransform();
btnSpinner.RenderTransform = rt;

// Выполнить анимацию объекта RotateTransform.
rt.BeginAnimation(RotateTransform.AngleProperty, dblAnim);
}
}
}

```

Первая главная задача этого метода — сконфигурировать объект `DoubleAnimation`, который начнется со значения 0 и закончится значением 360. Обратите внимание, что для этого объекта также обрабатывается событие `Completed`, где переключается булевская переменная уровня класса, которая используется для того, чтобы выполняющаяся анимация не была сброшена в начало.

Затем создается объект `RotateTransform`, который подключается к свойству `RenderTransform` элемента управления `Button` (по имени `btnSpinner`). И, наконец, объект `RenderTransform` информируется о начале анимации его свойства `Angle` с использованием объекта `DoubleAnimation`. Описание анимации в коде обычно осуществляется вызовом метода `BeginAnimation()` и передачей ему лежащего в основе свойства зависимости, которое требуется анимировать (вспомните, что по существующему соглашению это статическое поле класса), и связанного объекта анимации.

Добавим в программу еще одну анимацию, которая заставит кнопку плавно становиться невидимой при щелчке. Для начала создадим обработчик события `Click` объекта `btnSpinner` и поместим в него следующий код:

```

private void btnSpinner_Click(object sender, RoutedEventArgs e)
{
    DoubleAnimation dblAnim = new DoubleAnimation();
    dblAnim.From = 1.0;
    dblAnim.To = 0.0;
    btnSpinner.BeginAnimation(Button.OpacityProperty, dblAnim);
}

```

Здесь изменяется свойство `Opacity`, чтобы постепенно скрыть кнопку из виду. В настоящее время, однако, это сделать трудно, поскольку кнопка вращается слишком быстро. Каким образом управлять ходом анимации? Ответ на этот вопрос ищите ниже.

Управление темпом анимации

По умолчанию анимация занимает примерно одну секунду на переход между значениями, присвоенными свойствам `From` и `To`. Поэтому кнопке требуется одна секунда, чтобы повернуться на 360 градусов, и еще за одну секунду она постепенно скрывается из виду (после щелчка на ней).

Определить другой период времени на выполнение анимации можно с помощью свойства `Duration` объекта анимации, которому присваивается экземпляр объекта `Duration`. Обычно период времени устанавливается за счет передачи объекта `TimeSpan` конструктору `Duration`.

Рассмотрим следующее изменение, при котором кнопке выделяется на вращение 4 секунды:

```
private void btnSpinner_MouseEnter(object sender, MouseEventArgs e)
{
    if (!isSpinning)
    {
        isSpinning = true;
        // Создать объект анимации double и зарегистрировать
        // событие Completed.
        DoubleAnimation dblAnim = new DoubleAnimation();
        dblAnim.Completed += (o, s) => { isSpinning = false; };

        // На завершение поворота кнопке отводится 4 секунды.
        dblAnim.Duration = new Duration(TimeSpan.FromSeconds(4));
        ...
    }
}
```

Благодаря этой модификации, появляется возможность щелкнуть на кнопке во время ее вращения, после чего она плавно исчезает.

На заметку! Свойство `BeginTime` класса `Animation` также принимает объект `TimeSpan`. Помните, что это свойство устанавливается для указания времени до начала процесса анимации.

Запуск в обратном порядке и циклическое выполнение анимации

Объекты `Animation` можно также заставить запускать анимацию в обратном порядке по ее завершении, устанавливая для этого свойство `AutoReverse` в `true`. Например, если необходимо, чтобы кнопка снова стала видимой после исчезновения, можно написать следующий код:

```
private void btnSpinner_Click(object sender, RoutedEventArgs e)
{
    DoubleAnimation dblAnim = new DoubleAnimation();
    dblAnim.From = 1.0;
    dblAnim.To = 0.0;

    // По завершении запустить в обратном порядке.
    dblAnim.AutoReverse = true;
    btnSpinner.BeginAnimation(Button.OpacityProperty, dblAnim);
}
```

Если хотите, чтобы анимация повторялась некоторое количество раз (или никогда не прекращалась), то для этого можно применить свойство `RepeatBehavior`, общее для всех классов `Animation`. Передача конструктору простого числового значения позволяет задать жестко закодированное количество повторений. С другой стороны, если передать конструктору объект `TimeSpan`, то можно указать длительность времени повторения анимации. Наконец, чтобы выполнять анимацию бесконечно, можно просто указать `RepeatBehavior.Forever`. Рассмотрим следующие способы изменения поведения повторения любого из двух объектов `DoubleAnimation`, использованных в этом примере:

```
// Повторять бесконечно.
dblAnim.RepeatBehavior = RepeatBehavior.Forever;

// Повторять три раза.
dblAnim.RepeatBehavior = new RepeatBehavior(3);

// Повторять в течение 30 секунд.
dblAnim.RepeatBehavior = new RepeatBehavior(TimeSpan.FromSeconds(30));
```

На этом исследование анимационных аспектов объекта в коде C# и API-интерфейса анимации WPF завершено. Теперь посмотрим, как сделать то же самое в XAML-разметке.

Исходный код. Проект `SpinningButtonAnimationApp` доступен в подкаталоге `Chapter 30`.

Реализация анимации в разметке XAML

Реализация анимации в разметке подобна ее реализации в коде, во всяком случае, если речь идет о простых анимационных последовательностях. Когда нужно создавать более сложные анимации, которые могут включать одновременные изменения значений сразу множества свойств, объем разметки может значительно увеличиться. Даже если применяется какой-то инструмент для генерирования анимации, основанной на XAML, важно знать основы представления анимации в XAML, поскольку это облегчит решение задачи модификации и подгонки сгенерированного инструментом содержимого.

На заметку! В папке `XamlAnimations` внутри загружаемых примеров кода для настоящей главы вы найдете несколько файлов XAML. Скопируйте эти файлы разметки в специальный редактор XAML или в редактор `Kaхамl`, чтобы просмотреть результаты.

По большей части создание анимации подобно всему тому, что вы уже видели: сначала конфигурируется объект `Animation`, который затем ассоциируется со свойством объекта. Однако одно большое отличие состоит в том, что API-интерфейс WPF не является дружелюбным к вызовам функций. В связи с этим вместо вызова `BeginAnimation()` применяется *раскадровка* в качестве непрямого уровня.

Давайте рассмотрим полный пример анимации, определенный в терминах XAML, с последующим подробным разбором. Следующее определение XAML отобразит окно, содержащее единственную метку. Как только объект `Label` загружается в память, он начинает анимационную последовательность, при которой размер шрифта увеличивается от 12 до 100 точек за период в четыре секунды. Анимация будет повторяться столько времени, сколько объект остается загруженным в память. Эта разметка находится в файле `GrowLabelFont.xaml`, поэтому скопируйте его в приложение `MyXamlPad.exe` (или в редактор `Kaхамl`) и понаблюдайте за поведением.

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Height="200" Width="600" WindowStartupLocation="CenterScreen"
  Title="Growing Label Font!">
  <StackPanel>
    <Label Content = "Interesting...">
      <Label.Triggers>
        <EventTrigger RoutedEvent = "Label.Loaded">
          <EventTrigger.Actions>
            <BeginStoryboard>
              <Storyboard TargetProperty = "FontSize">
                <DoubleAnimation From = "12" To = "100" Duration = "0:0:4"
                  RepeatBehavior = "Forever"/>
              </Storyboard>
            </BeginStoryboard>
          </EventTrigger.Actions>
        </EventTrigger>
      </Label.Triggers>
    </Label>
  </StackPanel>
</Window>
```

А теперь подробно разберем этот пример.

Роль раскадровок

Двигаясь от наиболее глубоко вложенного элемента к внешнему, сначала мы встречаем элемент `<DoubleAnimation>`, который использует те же свойства, что были установлены в процедурном коде (`From`, `To`, `Duration` и `RepeatBehavior`):

```
<DoubleAnimation From = "12" To = "100" Duration = "0:0:4"
                RepeatBehavior = "Forever"/>
```

Как упоминалось ранее, элементы `Animation` помещаются внутрь элемента `<Storyboard>`, предназначенного для отображения объекта анимации на родительский тип через свойство `TargetProperty` — которым в данном случае является `FontSize`. Элемент `<Storyboard>` всегда помещен в родительский элемент по имени `<BeginStoryboard>`:

```
<BeginStoryboard>
  <Storyboard TargetProperty = "FontSize">
    <DoubleAnimation From = "12" To = "100" Duration = "0:0:4"
                      RepeatBehavior = "Forever"/>
  </Storyboard>
</BeginStoryboard>
```

Роль триггеров событий

После определения элемента `<BeginStoryboard>` должна быть указана какая-то разновидность действия, которая вызовет запуск анимации. В WPF предусмотрены различные способы реагирования на условия времени выполнения в разметке, один из которых называется *триггером*. В самом общем виде триггер можно рассматривать как способ реагирования на условия событий в XAML-разметке, без необходимости в процедурном коде.

Обычно, когда определяется реакция на событие в C#, пишется специальный код, который выполняется при наступлении события. Однако триггер — это всего лишь способ получить уведомление о том, что некоторое событие произошло (загрузка элемента в память, наведение курсора мыши на элемент, получение элементом фокуса и т.п.).

Получив уведомление о наступлении события, можно запускать раскадровку. В рассматриваемом примере мы реагируем на факт загрузки элемента `Label` в память. Поскольку нас интересует событие `Loaded` элемента `Label`, элемент `<EventTrigger>` помещается в коллекцию триггеров элемента `Label`:

```
<Label Content = "Interesting...">
  <Label.Triggers>
    <EventTrigger RoutedEvent = "Label.Loaded">
      <EventTrigger.Actions>
        <BeginStoryboard>
          <Storyboard TargetProperty = "FontSize">
            <DoubleAnimation From = "12" To = "100" Duration = "0:0:4"
                              RepeatBehavior = "Forever"/>
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger.Actions>
    </EventTrigger>
  </Label.Triggers>
</Label>
```

Теперь рассмотрим другой пример определения анимации в XAML, на этот раз, для анимации *ключевыми кадрами*.

Анимация с использованием дискретных ключевых кадров

В отличие от анимации с линейной интерполяцией, которая обеспечивает только перемещение между начальной и конечной точками, анимация *ключевыми кадрами* позволяет создавать коллекции заданных значений, которые должны встречаться в определенные моменты времени.

Чтобы проиллюстрировать использование типа дискретного ключевого кадра, предположим, что необходимо построить элемент управления Button, который анимирует свое содержимое таким образом, чтобы на протяжении трех секунд появлялось значение OK!, по одному символу за раз. Показанная ниже разметка находится в файле StringAnimation.xaml. Скопируйте ее в программу MyXamlPad.exe (или в редактор Кахамл) и просмотрите результат.

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Height="100" Width="300"
  WindowStartupLocation="CenterScreen" Title="Animate String Data!">
  <StackPanel>
    <Button Name="myButton" Height="40"
      FontSize="16pt" FontFamily="Verdana" Width="100">
      <Button.Triggers>
        <EventTrigger RoutedEvent="Button.Loaded">
          <BeginStoryboard>
            <Storyboard>
              <StringAnimationUsingKeyFrames RepeatBehavior="Forever"
                Storyboard.TargetName="myButton"
                Storyboard.TargetProperty="Content"
                Duration="0:0:3">
                <DiscreteStringKeyFrame Value="" KeyTime="0:0:0" />
                <DiscreteStringKeyFrame Value="O" KeyTime="0:0:1" />
                <DiscreteStringKeyFrame Value="OK" KeyTime="0:0:1.5" />
                <DiscreteStringKeyFrame Value="OK!" KeyTime="0:0:2" />
              </StringAnimationUsingKeyFrames>
            </Storyboard>
          </BeginStoryboard>
        </EventTrigger>
      </Button.Triggers>
    </Button>
  </StackPanel>
</Window>
```

Прежде всего, обратите внимание, что для кнопки определен триггер события, который обеспечивает запуск раскадровки при загрузке кнопки в память.

Класс StringAnimationUsingKeyFrames отвечает за изменение содержимого кнопки, через значения Storyboard.TargetName и Storyboard.TargetProperty.

Внутри контекста элемента <StringAnimationUsingKeyFrames> определены четыре элемента DiscreteStringKeyFrame, которые изменяют свойство Content на протяжении двух секунд (длительность, установленная StringAnimationUsingKeyFrames, составляет в сумме три секунды, поэтому между финальным ! и следующим появлением O заметна небольшая пауза).

Теперь, когда вы получили некоторое представление о том, как строятся анимации в коде C# и XAML, давайте уделим внимание стилям WPF, которые интенсивно используют графику, объектные ресурсы и анимацию.

Исходный код. Эти свободные файлы XAML доступны в подкаталоге XamlAnimations каталога Chapter 30.

Роль стилей WPF

При построении пользовательского интерфейса WPF-приложения нередко требуется обеспечить общий вид и поведение для какого-то семейства элементов управления. Например, необходимо, чтобы все типы кнопок имели общую высоту, ширину, цвет и размер шрифта своего строкового содержимого. Хотя это можно обеспечить за счет установки значений индивидуальных свойств, такой подход затрудняет внесение изменений, поскольку при каждом изменении придется переустанавливать один и тот же набор свойств во множестве объектов.

К счастью, в WPF предлагается простой способ ограничения внешнего вида и поведения взаимосвязанных элементов управления с использованием стилей. Попросту говоря, стиль WPF — это объект, который поддерживает коллекцию пар “свойство-значение”. С точки зрения программирования индивидуальный стиль представлен классом `System.Windows.Style`. Этот класс имеет свойство по имени `Setters`, которое является строго типизированной коллекцией объектов `Setter`. Именно объект `Setter` позволяет определять пары “свойство-значение”.

В дополнение к коллекции `Setters`, класс `Style` также определяет ряд других важных членов, которые позволяют включать триггеры, ограничивать место применения стиля и даже создавать новый стиль на основе существующего (воспринимайте это как “наследование стиля”). Ниже перечислены важные члены класса `Style`:

- `Triggers` — представляет коллекцию объектов триггеров, которая позволяет фиксировать различные условия событий в стиле;
- `BasedOn` — позволяет строить новый стиль на основе существующего;
- `TargetType` — позволяет ограничивать места применения стиля.

Определение и применение стиля

Почти в любом случае объект `Style` упаковывается в виде объектного ресурса. Подобно любому объектному ресурсу, его можно упаковывать на уровне окна или уровня приложения, а также внутри выделенного словаря ресурсов (это замечательно, потому что делает объект `Style` легко доступным по всему приложению). Теперь вспомните, что целью является определение объекта `Style`, который наполняет (как минимум) коллекцию `Setters` набором пар “свойство-значение”.

Создайте новое приложение WPF по имени `WpfStyles`, используя `Visual Studio`. Давайте построим стиль, фиксирующий базовые характеристики элемента управления в нашем приложении.

Откройте файл `App.xaml` и определите следующий именованный стиль:

```
<Application x:Class="WpfStyles.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  StartupUri="MainWindow.xaml">
  <Application.Resources>
    <Style x:Key="BasicControlStyle">
      <Setter Property="Control.FontSize" Value="14"/>
      <Setter Property="Control.Height" Value="40"/>
      <Setter Property="Control.Cursor" Value="Hand"/>
    </Style>
  </Application.Resources>
</Application>
```

Обратите внимание, что `BasicControlStyle` добавляет во внутреннюю коллекцию три объекта `Setter`. Теперь применим этот стиль к нескольким элементам управ-

ления в главном окне. Поскольку этот стиль является объектным ресурсом, элементы управления, которые хотят использовать его, нуждаются в расширении разметки {StackResource} или {DynamicResource} для нахождения стиля. Когда они находят стиль, то устанавливают элемента ресурса в идентично именованное свойство Style. Рассмотрим следующее определение <Window>:

```
<Window x:Class="WpfStyles.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="A Window with Style!" Height="229"
  Width="525" WindowStartupLocation="CenterScreen">
  <StackPanel>
    <Label x:Name="lblInfo" Content="This style is boring..."
      Style="{StaticResource BasicControlStyle}" Width="150"/>
    <Button x:Name="btnTestButton" Content="Yes, but we are reusing settings!"
      Style="{StaticResource BasicControlStyle}" Width="250"/>
  </StackPanel>
</Window>
```

Запустив это приложение, вы обнаружите, что оба элемента управления поддерживают один и тот же курсор, высоту и размер шрифта.

Переопределение настроек стиля

В примере определены элементы Button и Label, которые подчиняются ограничениям, накладываемым нашим стилем. Разумеется, если элемент управления желает применить стиль и затем изменить некоторые из определенных установок, то это вполне нормально. Например, Button теперь использует курсор Help (вместо курсора Hand, определенного в стиле):

```
<Button x:Name="btnTestButton" Content="Yes, but we are reusing settings!"
  Cursor="Help" Style="{StaticResource BasicControlStyle}" Width="250" />
```

Стили обрабатываются перед индивидуальными установками свойств элемента управления, использующего стиль, поэтому элементы управления могут “переопределять” настройки от случая к случаю.

Автоматическое применение стиля с помощью TargetType

В настоящий момент стиль определен таким образом, что его может принимать любой элемент управления (и должен делать это явно, устанавливая свое свойство Style), исходя из того, что каждое свойство поддерживается классом Control. Для программы, определяющей десятки настроек, это означало бы значительный объем повторяющегося кода. Один из способов в некоторой степени улучшить ситуацию предусматривает использование атрибута TargetType. Добавление этого атрибута к открываемому элементу Style позволяет точно указать, когда он может быть применен:

```
<Style x:Key="BasicControlStyle" TargetType="Control">
  <Setter Property="FontSize" Value="14"/>
  <Setter Property="Height" Value="40"/>
  <Setter Property="Cursor" Value="Hand"/>
</Style>
```

На заметку! При построении стиля, использующего тип базового класса, не нужно беспокоиться о том, что присваиваемое значение свойству зависимости не поддерживается производными типами. Если производный тип не поддерживает заданное свойство зависимости, оно игнорируется.



Рис. 30.12.
Элементы управления с разными стилями

Это до некоторой степени помогает, но все равно мы имеем стиль, который может применяться к любому элементу управления. Атрибут `TargetType` более удобен, когда необходимо определить стиль, который может быть применен только к определенному типу элементов управления. Добавьте следующий стиль в словарь ресурсов приложения:

```
<Style x:Key = "BigGreenButton" TargetType="Button">
  <Setter Property = "FontSize" Value = "20"/>
  <Setter Property = "Height" Value = "100"/>
  <Setter Property = "Width" Value = "100"/>
  <Setter Property = "Background" Value = "DarkGreen"/>
  <Setter Property = "Foreground" Value = "Yellow"/>
</Style>
```

Этот стиль будет работать только с элементами управления `Button` (или подклассами `Button`), и если применить его к несовместимому элементу, то возникнут ошибки разметки и компиляции. Если `Button` использует новый стиль так, как продемонстрировано ниже, то вывод будет выглядеть подобно показанному на рис. 30.12:

```
<Button x:Name="btnTestButton" Content="This Style ROCKS!"
  Cursor="Help" Style="{StaticResource BigGreenButton}" Width="250" />
```

Создание подклассов существующих стилей

Новые стили можно также строить на основе какого-то существующего стиля с помощью свойства `BasedOn`. Расширяемый стиль должен иметь соответствующий атрибут `x:Key` в словаре, поскольку производный стиль будет ссылаться на него по имени, используя расширение разметки `{StaticResource}`. Ниже показан новый стиль, основанный на стиле `BigGreenButton`, который поворачивает элемент-кнопку на 20 градусов:



Рис. 30.13.
Использование производного стиля

```
<!-- Стиль основан на BigGreenButton -->
<Style x:Key = "TiltButton" TargetType="Button"
  BasedOn = "{StaticResource BigGreenButton}">
  <Setter Property = "Foreground" Value = "White"/>
  <Setter Property = "RenderTransform">
    <Setter.Value>
      <RotateTransform Angle = "20"/>
    </Setter.Value>
  </Setter>
</Style>
```

На этот раз вывод будет выглядеть так, как показано на рис. 30.13.

Роль неименованных стилей

Предположим, что необходимо гарантировать, чтобы все элементы управления `TextBox` имели одинаковый внешний вид и поведение. Пусть определен стиль в форме ресурса уровня приложения, доступ которому имеют все окна программы. Хотя это шаг в правильном направлении, но если есть множество окон с множеством элементов управления `TextBox`, то свойство `Style` придется устанавливать много раз!

Стили WPF могут быть неявно применены ко всем элементам управления внутри заданного контекста XAML. Чтобы создать такой стиль, необходимо воспользоваться свойством `TargetType`, но не присваивать ресурсу `Style` значение `x:Key`. Такой "неименованный стиль" теперь применяется ко всем элементам управления корректного типа. Ниже показан другой стиль уровня приложения, который будет применен автоматически ко всем элементам управления `TextBox` текущего приложения:

```
<!-- Стандартный стиль для всех текстовых полей -->
<Style TargetType="TextBox">
  <Setter Property = "FontSize" Value = "14"/>
  <Setter Property = "Width" Value = "100"/>
  <Setter Property = "Height" Value = "30"/>
  <Setter Property = "BorderThickness" Value = "5"/>
  <Setter Property = "BorderBrush" Value = "Red"/>
  <Setter Property = "FontStyle" Value = "Italic"/>
</Style>
```

Можно определить любое количество элементов управления `TextBox`, и все они автоматически получат этот определенный внешний вид. Если конкретному элементу `TextBox` не нужен такой стандартный внешний вид, он может отказаться от него, установив свойство `Style` в `{x:Null}`. Например, элемент `txtTest` будет иметь неименованный стандартный стиль, а элемент `txtTest2` сделает все самостоятельно:

```
<TextBox x:Name="txtTest"/>
<TextBox x:Name="txtTest2" Style="{x:Null}" BorderBrush="Black"
  BorderThickness="5" Height="60" Width="100" Text="Ha!"/>
```

Определение стилей с триггерами

Стили WPF могут также содержать триггеры, упаковывая объекты `Trigger` в коллекцию `Triggers` объекта `Style`. Использование триггеров в стиле позволяет определить некоторые элементы `<Setter>` таким образом, что они будут применены только в случае истинности определенного условия триггера. Например, возможно, требуется увеличить размер шрифта, когда курсор мыши наведен на кнопку. Или, может быть, необходимо обеспечить, чтобы текстовое поле с текущим фокусом было подсвечено фоном заданного цвета. Триггеры очень удобны в ситуациях подобного рода, т.к. они позволяют предпринимать специфические действия при изменении свойства, не требуя реализации явной логики C# в файле отделенного кода.

Ниже приведена модифицированная разметка для стиля `TextBox`, которая обеспечивает установку фона желтого цвета этого элемента, когда он получает фокус:

```
<!-- Стандартный стиль для всех текстовых полей -->
<Style TargetType="TextBox">
  <Setter Property = "FontSize" Value = "14"/>
  <Setter Property = "Width" Value = "100"/>
  <Setter Property = "Height" Value = "30"/>
  <Setter Property = "BorderThickness" Value = "5"/>
  <Setter Property = "BorderBrush" Value = "Red"/>
  <Setter Property = "FontStyle" Value = "Italic"/>
  <!-- Следующая установка будет применена, только когда
    текстовое поле находится в фокусе -->
  <Style.Triggers>
    <Trigger Property = "IsFocused" Value = "True">
      <Setter Property = "Background" Value = "Yellow"/>
    </Trigger>
  </Style.Triggers>
</Style>
```

Опробовав этот стиль, вы обнаружите, что по мере перехода с помощью клавиши `<Tab>` между объектами `TextBox` текущий выбранный `TextBox` получает фон желтого цвета (если только это не отключено за счет присваивания `{x:Null}` свойству `Style`).

Триггеры свойств также весьма интеллектуальны в том смысле, что когда условие триггера не истинно, свойство автоматически получает значение, присваиваемое по умолчанию. Поэтому, как только `TextBox` теряет фокус, он также автоматически принимает стандартный цвет, без какого-либо вашего участия. В отличие от этого, тригге-

ры событий (которые рассматривались при описании анимации WPF) не возвращаются автоматически в предыдущее состояние.

Определение стилей с несколькими триггерами

Триггеры также могут быть спроектированы так, что определенные элементы `<Setter>` будут применены, когда истинно *несколько условий* (это похоже на построение оператора `if` для множества условий). Предположим, что необходимо установить фон Yellow для элемента `TextBox` только в том случае, если он имеет активный фокус и курсор мыши наведен на него. В этом случае можно воспользоваться элементом `<MultiTrigger>` для определения каждого условия:

```
<!-- Стандартный стиль для всех текстовых полей -->
<Style TargetType="TextBox">
    <Setter Property = "FontSize" Value = "14"/>
    <Setter Property = "Width" Value = "100"/>
    <Setter Property = "Height" Value = "30"/>
    <Setter Property = "BorderThickness" Value = "5"/>
    <Setter Property = "BorderBrush" Value = "Red"/>
    <Setter Property = "FontStyle" Value = "Italic"/>
<!-- Следующая установка будет применена, только когда
    текстовое поле в фокусе И на него наведен курсор мыши -->
    <Style.Triggers>
        <MultiTrigger>
            <MultiTrigger.Conditions>
                <Condition Property = "IsFocused" Value = "True"/>
                <Condition Property = "IsMouseOver" Value = "True"/>
            </MultiTrigger.Conditions>
            <Setter Property = "Background" Value = "Yellow"/>
        </MultiTrigger>
    </Style.Triggers>
</Style>
```

Анимированные стили

Стили также могут включать триггеры, которые запускают анимационную последовательность.

Ниже приведен последний стиль, который, будучи примененным к элементам управления `Button`, заставит их увеличиваться и уменьшаться в размерах, когда курсор мыши находится внутри области поверхности кнопки:

```
<!-- Стиль увеличивающейся кнопки -->
<Style x:Key = "GrowingButtonStyle" TargetType="Button">
    <Setter Property = "Height" Value = "40"/>
    <Setter Property = "Width" Value = "100"/>
    <Style.Triggers>
        <Trigger Property = "IsMouseOver" Value = "True">
            <Trigger.EnterActions>
                <BeginStoryboard>
                    <Storyboard TargetProperty = "Height">
                        <DoubleAnimation From = "40" To = "200"
                            Duration = "0:0:2" AutoReverse="True"/>
                    </Storyboard>
                </BeginStoryboard>
            </Trigger.EnterActions>
        </Trigger>
    </Style.Triggers>
</Style>
```

Здесь коллекция триггеров проверяет истинность свойства `IsMouseOver`. Если оно истинно, определяется элемент `<Trigger.EnterActions>` для запуска простой раскадровки, которая заставляет кнопку за две секунды увеличиться до значения `Height`, равного 200 (и затем вернуться к значению `Height`, равному 40). Чтобы добавить другие изменения свойств, можно также определить контекст `<Trigger.ExitActions>` для определения любых специальных действий, которые должны быть выполнены, когда `IsMouseOver` равно `false`.

Применение стилей в коде

Вспомните, что стиль может быть применен также во время выполнения. Это полезно, когда нужно позволить конечным пользователем выбирать, как должен выглядеть их пользовательский интерфейс, либо если требуется принудительно применить внешний вид и поведение на основе настроек безопасности (например, стиль `DisableAllButton`) или каких-то других условий.

В этом проекте уже определено множество стилей, и многие из них могут быть применены к элементам управления `Button`. Теперь давайте переделаем пользовательский интерфейс главного окна, чтобы позволить пользователю выбирать из некоторых этих стилей, указывая имя в списке `ListBox`. На основе такого выбора будет применяться соответствующий стиль. Ниже показана финальная разметка элемента `<Window>`:

```
<Window x:Class="WpfStyles.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Height="350" Title="A Window with Style!"
        Width="525" WindowStartupLocation="CenterScreen">
    <DockPanel>
        <StackPanel Orientation="Horizontal" DockPanel.Dock="Top">
            <Label Content="Please Pick a Style for this Button" Height="50"/>
            <ListBox x:Name="lstStyles" Height="80" Width="150"
                    Background="LightBlue"
                    SelectionChanged="comboStyles_Changed" />
        </StackPanel>
        <Button x:Name="btnStyle" Height="40" Width="100" Content="OK!" />
    </DockPanel>
</Window>
```

Элемент управления `ListBox` (по имени `lstStyles`) будет заполняться динамически внутри конструктора окна:

```
public MainWindow()
{
    InitializeComponent();

    // Заполнить окно списка всеми стилями для элементов Button.
    lstStyles.Items.Add("GrowingButtonStyle");
    lstStyles.Items.Add("TiltButton");
    lstStyles.Items.Add("BigGreenButton");
    lstStyles.Items.Add("BasicControlStyle");
}
```

Последняя задача связана с обработкой события `SelectionChanged` в связанном файле кода. Обратите внимание, что в следующем коде имеется возможность извлечения текущего ресурса по имени с использованием унаследованного метода `TryFindResource()`:

```
private void comboStyles_Changed(object sender, SelectionChangedEventArgs e)
{
    // Получить стиль, выбранный из окна списка.
    Style currStyle = (Style)
        TryFindResource(lstStyles.SelectedValue);
    if (currStyle != null)
    {
        // Установить стиль для типа кнопки.
        this.btnStyle.Style = currStyle;
    }
}
```

Запустив это приложение, можно выбирать один из четырех стилей кнопок на лету. На рис. 30.14 показано готовое приложение в действии.

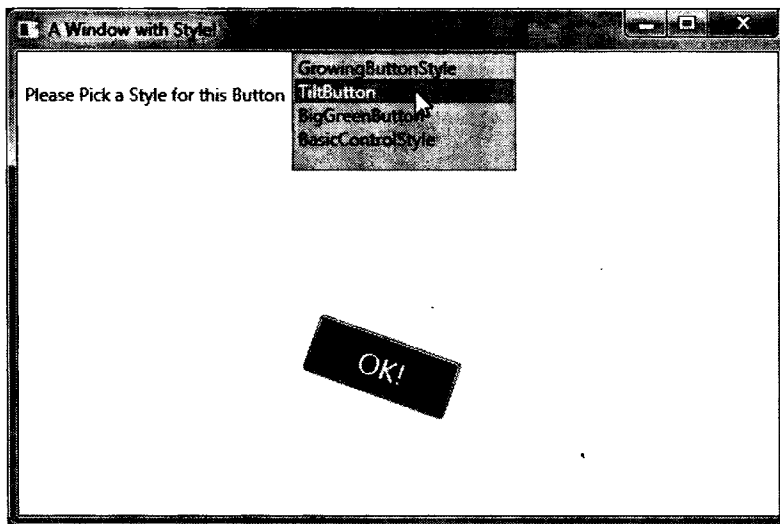


Рис. 30.14. Элементы управления с разными стилями

Исходный код. Проект WpfStyles доступен в подкаталоге Chapter 30.

Резюме

В первой части этой главы рассматривалась система управления ресурсами WPF. Мы начали с изучения работы с двоичными ресурсами, а затем ознакомились с ролью объектных ресурсов. Вы узнали, что объектные ресурсы — это именованные фрагменты XAML-разметки, которые могут быть сохранены в различных местах, позволяя многократно использовать это содержимое.

После этого был описан API-интерфейс анимации WPF. Приводились некоторые примеры создания анимации в коде C#, а также в XAML-разметке. Для управления выполнением анимации, определенной в разметке, служат элементы <Storyboard> и триггеры. В завершение главы был продемонстрирован механизм стилей WPF, который интенсивно использует графику, объектные ресурсы и анимацию.

Свойства зависимости, маршрутизируемые события и шаблоны

В этой главе изучение модели программирования WPF завершается рассмотрением вопросов, связанных с построением настраиваемых шаблонов элементов управления. Хотя модель содержимого и механизм стилей позволяют добавлять уникальные части к стандартным элементам управления WPF, процесс построения специальных шаблонов позволяет полностью определить, как элемент управления должен визуализировать свой вывод, реагировать на переходы между состояниями и интегрироваться в API-интерфейсы WPF.

Мы начнем рассмотрение с двух тем, которые важны для создания специального элемента управления — *свойств зависимости* и *маршрутизируемых событий*. После освоения этих тем мы перейдем к изучению роли *стандартного шаблона* и программному взаимодействию с ним во время выполнения. Оставшаяся часть главы посвящена построению собственных шаблонов элементов управления и добавлению визуальных подсказок с помощью инфраструктуры триггеров WPF.

На заметку! Построение шаблонов производственного уровня определенно потребует использования средства Expression Blend, т.к. оно содержит многочисленные встроенные инструменты для работы с шаблонами. В этой главе будет создано несколько *простых* шаблонов, которые легко построить в Visual Studio. Если вы хотите изучить детали работы с Expression Blend, обратитесь к книге Эндрю Троелсена *Expression Blend 4 с примерами на C# для профессионалов* (ИД “Вильямс”, 2011 г.).

Роль свойств зависимости

Как и в любом API-интерфейсе .NET, в WPF используются все члены системы типов .NET (классы, структуры, интерфейсы, делегаты, перечисления) и все члены каждого типа (свойства, методы, события, константные данные, поля только для чтения и т.п.). Однако WPF также поддерживает уникальную концепцию программирования — *свойства зависимости*.

Подобно “нормальному” свойству .NET (которое в литературе, посвященной WPF, часто называют *свойством CLR*), свойства зависимости можно устанавливать декларативно с помощью XAML или программно в файле кода. Кроме того, свойства зависимости (как и свойства CLR) в конечном итоге предназначены для инкапсуляции полей данных

класса и могут быть сконфигурированы как доступные только для чтения, только для записи или для чтения и записи.

Но более интересно то, что почти всегда вам и не надо будет знать, к чему на самом деле вы обращаетесь — к свойству зависимости или к свойству CLR! Например, свойства `Height` и `Width`, которые элементы управления WPF наследуют от класса `FrameworkElement`, а также член `Content`, унаследованный от `ControlContent` — все это фактически свойства зависимости:

```
<!-- Установить три свойства зависимости -->
<Button x:Name = "btnMyButton" Height = "50" Width = "100" Content = "OK"/>
```

С учетом всех этих сходств возникает вопрос: зачем нужно было определять в WPF новый термин для такой знакомой концепции? Причина кроется в способе реализации свойства зависимости внутри класса. Пример кодирования будет показан ниже, а на высоком уровне все свойства зависимости создаются следующим образом.

- Прежде всего, класс, определяющий свойство зависимости, должен иметь в своей цепочке наследования класс `DependencyObject`.
- В классе `DependencyProperty` только одно свойство зависимости представлено как открытое статическое поле, предназначенное только для чтения. По соглашению имя этого поля формируется добавлением слова `Property` к имени оболочки CLR (см. последний пункт этого списка).
- Переменная `DependencyProperty` регистрируется с помощью статического вызова `DependencyProperty.Register()`, обычно в статическом конструкторе или прямо при объявлении переменной.
- Наконец, класс определяет дружественное к XAML свойство CLR, которое осуществляет вызовы методов, предоставленных `DependencyObject`, для получения и установки значения.

После реализации свойства зависимости предоставляют множество мощных средств, используемых различными технологиями WPF: привязка данных, службы анимации, стили, шаблоны и т.д. В основном, цель создания свойств зависимостей заключается в том, чтобы предложить способ для вычисления значения свойства на основе значений из других источников. Ниже приведен список некоторых основных преимуществ, которые далеко выходят за рамки простой инкапсуляции данных, поддерживаемой свойствами CLR.

- Свойства зависимости могут наследовать свои значения от XAML-определения родительского элемента. Например, если в открывающем дескрипторе `<Window>` определено значение атрибута `FontSize`, то все элементы управления в этом `Window` будут по умолчанию иметь этот размер шрифта.
- Свойства зависимости поддерживают возможность установки значений с помощью элементов внутри их контекста XAML, например, установка в `Button` свойства `Dock` из родительского контейнера `DockPanel`. (Вспомните из главы 28, что *присоединяемые свойства* делают именно это, поскольку присоединяемые свойства — это разновидность свойств зависимости.)
- Свойства зависимости позволяют WPF вычислять значение на основе нескольких внешних значений, что может быть очень важно для служб анимации и привязки данных.
- Свойства зависимости предоставляют поддержку инфраструктуры для триггеров WPF (также довольно часто используемых при работе с анимацией и привязкой данных).

Запомните, что во многих случаях вы будете взаимодействовать с существующим свойством зависимости так же, как и с обычным свойством CLR (благодаря оболочке XAML). Однако когда в главе 28 шла речь о привязке данных, вы узнали, что при необходимости установки привязки данных в коде следует вызывать метод `SetBinding()` на целевом объекте операции и указывать *свойство зависимости*, которым привязка будет оперировать:

```
private void SetBindings()
{
    Binding b = new Binding();
    b.Converter = new MyDoubleConverter();
    b.Source = this.mySB;
    b.Path = new PropertyPath("Value");

    // Указать свойство зависимости.
    this.labelSBThumb.SetBinding(Label.ContentProperty, b);
}
```

Похожий код встречался в предыдущей главе, когда шла речь о запуске анимации в коде:

```
// Указать свойство зависимости.
rt.BeginAnimation(RotateTransform.AngleProperty, dblAnim);
```

Единственный случай, когда может понадобиться создавать специальное свойство зависимости — во время разработки собственного элемента управления WPF. Например, при построении класса `UserControl` с четырьмя специальными свойствами, которые должны тесно интегрироваться с API-интерфейсом WPF, они должны быть написаны с использованием логики свойств зависимости.

В частности, если свойства должны быть целью операции привязки данных либо анимации, если свойство должно уведомлять всех о своем изменении, если оно должно уметь работать, как `Setter` в стиле WPF, или если оно должно уметь получать свои значения от родительского элемента, то обычного свойства CLR для этого не достаточно. В случае применения обычного свойства другие программисты смогут получать и устанавливать значение, однако если они попытаются использовать такие свойства в контексте службы WPF, все будет работать не так, как ожидалось. Поскольку никогда заранее не известно, как другие пожелают взаимодействовать со свойствами специальных классов `UserControl`, нужно приучить себя *всегда* определять свойства зависимости при построении специальных элементов управления.

Знакомство с существующим свойством зависимости

Прежде чем переходить к объяснению того, как создавать специальное свойство зависимости, давайте взглянем на внутреннюю реализацию свойства `Height` класса `FrameworkElement`. Соответствующий код показан ниже (с комментариями).

```
// FrameworkElement является DependencyObject.
public class FrameworkElement : UIElement, IFrameworkInputElement,
    IInputElement, ISupportInitialize, IHaveResources, IQueryAmbient
{
    ...
    // Статическое поле только для чтения типа DependencyProperty.
    public static readonly DependencyProperty HeightProperty;

    // Поле DependencyProperty часто регистрируется
    // в статическом конструкторе класса.
    static FrameworkElement()
    {
        ...
    }
}
```

```

HeightProperty = DependencyProperty.Register(
    "Height",
    typeof(double),
    typeof/FrameworkElement),
    new FrameworkPropertyMetadata((double) 1.0 / (double) 0.0,
        FrameworkPropertyMetadataOptions.AffectsMeasure,
        new PropertyChangedCallback(FrameworkElement.OnTransformDirty)),
    new ValidateValueCallback(FrameworkElement.IsWidthHeightValid));
}

// Оболочка CLR, реализованная унаследованными
// методами GetValue()/SetValue().
public double Height
{
    get { return (double) base.GetValue(HeightProperty); }
    set { base.SetValue(HeightProperty, value); }
}
}

```

Как видите, для свойств зависимости нужен немалый объем дополнительного кода по сравнению с обычным свойством CLR. На самом деле зависимость может оказаться еще более сложной, чем здесь показано (к счастью, многие реализации проще, чем Height).

Прежде всего, вспомните, что если в классе нужно определить свойство зависимости, он должен иметь в своей цепочке наследования `DependencyObject`, т.к. в этом классе определены методы `GetValue()` и `SetValue()`, используемые оболочкой CLR. Поскольку `FrameworkElement` “является” классом `DependencyObject`, это требование удовлетворено.

Далее, вспомните, что место хранения действительного значения свойства (в случае `Height` — значения `double`) представлено как открытое, статическое, предназначенное только для чтения поле типа `DependencyProperty`. По соглашению имя этого свойства образуется из имени соответствующей оболочки CLR с добавлением суффикса `Property`:

```
public static readonly DependencyProperty HeightProperty;
```

Поскольку свойства зависимости объявляются как статические поля, они обычно создаются (и регистрируются) в статическом конструкторе класса.

Объект `DependencyProperty` создается вызовом статического метода `DependencyProperty.Register()`. Этот метод имеет множество перегруженных версий; однако в случае `Height` метод `DependencyProperty.Register()` вызывается следующим образом:

```

HeightProperty = DependencyProperty.Register(
    "Height",
    typeof(double),
    typeof/FrameworkElement),
    new FrameworkPropertyMetadata((double)0.0,
        FrameworkPropertyMetadataOptions.AffectsMeasure,
        new PropertyChangedCallback(FrameworkElement.OnTransformDirty)),
    new ValidateValueCallback(FrameworkElement.IsWidthHeightValid));

```

Первый аргумент метода `DependencyProperty.Register()` — это имя обычного CLR-свойства класса (в данном случае `Height`), а второй аргумент содержит информацию о типе инкапсулируемых данных (`double`). Третий аргумент задает информации о типе класса, к которому относится это свойство (в данном случае — `FrameworkElement`). Третий аргумент может показаться излишним (ведь поле `HeightProperty` уже определено внутри класса `FrameworkElement`), но это очень разумный аспект WPF, который

позволяет одному классу регистрировать свойства в другом классе (даже если определение класса было запечатано).

Четвертый аргумент, передаваемый методу `DependencyProperty.Register()` в приведенном примере — это то, что и придает свойствам зависимости их уникальные характеристики. Здесь передается объект `FrameworkPropertyMetadata`, описывающий, как инфраструктура WPF должна обрабатывать это свойство в отношении уведомлений с помощью обратных вызовов (если свойство должно извещать других об изменениях своего значения), и различные параметры (представленные перечислением `FrameworkPropertyMetadataOptions`), которые управляют тем, что именно затрагивается данным свойством (работает ли оно с привязкой данных, может ли наследоваться, и т.п.). В этом случае аргументы конструктора `FrameworkPropertyMetadata` описаны так, как показано ниже:

```
new FrameworkPropertyMetadata(
    // Стандартное значение свойства.
    (double)0.0,
    // Параметры метаданных.
    FrameworkPropertyMetadataOptions.AffectsMeasure,
    // Делегат, указывающий на метод, вызываемый при изменении свойства.
    new PropertyChangedCallback(FrameworkElement.OnTransformDirty)
)
```

Так как последний аргумент конструктора `FrameworkPropertyMetadata` является делегатом, обратите внимание, что этот параметр конструктора указывает на статический метод `OnTransformDirty()` класса `FrameworkElement`. Хотя код этого метода подробно рассматриваться не будет, имейте в виду, что при создании специального свойства зависимости всегда можно задать делегат `PropertyChangeCallback`, чтобы указать на метод, который вызывается при изменении значения свойства.

Возвратимся к последнему параметру, переданному методу `DependencyProperty.Register()` — второму делегату типа `ValidateValueCallback`, который указывает на метод класса `FrameworkElement`, вызываемый для проверки достоверности значения, которое присваивается свойству:

```
new ValidateValueCallback(FrameworkElement.IsWidthHeightValid)
```

Этот метод содержит логику, которая обычно содержится в блоке установки значения свойства (подробнее об этом рассказывается в следующем разделе):

```
private static bool IsWidthHeightValid(object value)
{
    double num = (double) value;
    return ((!DoubleUtil.IsNaN(num) && (num >= 0.0))
        && !double.IsPositiveInfinity(num));
}
```

После того, как объект `DependencyProperty` зарегистрирован, остается лишь упаковать поле в обычное свойство CLR (в рассматриваемом случае — `Height`). Однако обратите внимание, что блоки `get` и `set` не просто возвращают или устанавливают значение `double` переменной-члена класса, но делают это непрямо, используя методы `GetValue()` и `SetValue()` из базового класса `System.Windows.DependencyObject`:

```
public double Height
{
    get { return (double) base.GetValue(HeightProperty); }
    set { base.SetValue(HeightProperty, value); }
}
```


Важные замечания относительно оболочек свойств CLR

Итак, свойства зависимости выглядят как обычные свойства, когда вы извлекаете или устанавливаете их значения в XAML либо в коде, но их внутренняя реализация основана на намного более изощренных приемах кодирования. Вспомните, что основное назначение этого процесса — создание специального элемента управления, имеющего специальные свойства, которые должны быть интегрированы со службами WPF, требующими взаимодействия с помощью свойства зависимости (например, анимация, привязка данных и стили).

Хотя часть реализации свойства зависимости включает определение оболочки CLR, *никогда не следует помещать логику проверки достоверности в блок set*. И потому CLR-оболочка свойства зависимости *не должна делать ничего помимо вызовов GetValue() или SetValue()*.

Дело в том, что исполняющая среда WPF сконструирована таким образом, что если написать разметку XAML, которая выглядит как установка свойства, например:

```
<Button x:Name="myButton" Height="100" .../>
```

то исполняющая среда полностью минует блок set свойства Height и *непосредственно* вызовет SetValue()! Причина столь странного поведения кроется в оптимизации. Если бы исполняющая среда WPF непосредственно вызывала блок установки свойства Height, то ей пришлось бы во время выполнения посредством рефлексии находить поле DependencyProperty (указанное в первом аргументе SetValue()), ссылаться на него в памяти и т.д. То же самое верно и в случае написания разметки XAML, которая извлекает значение свойства Height — метод GetValue() будет вызван напрямую.

Но раз так, то зачем вообще создавать CLR-оболочку? Дело в том, что XAML в WPF не позволяет вызывать функции в разметке, так что следующий фрагмент был бы ошибочным:

```
<!-- Ошибка! Вызывать методы в XAML-разметке WPF нельзя! -->
<Button x:Name="myButton" this.SetValue("100") .../>
```

Установку или получение значения в разметке с использованием CLR-оболочки следует воспринимать как способ указания исполняющей среде WPF о необходимости вызвать GetValue()/SetValue(), поскольку напрямую сделать это в разметке невозможно. А что, если вызвать CLR-оболочку в коде следующим образом?

```
Button b = new Button();
b.Height = 10;
```

В этом случае, если бы блок set свойства Height содержал какой-то код помимо вызова SetValue(), он *должен* был бы выполняться, т.к. оптимизация синтаксического анализатора XAML в WPF не применяется.

Основное правило таково: при регистрации свойства зависимости используйте делегат ValidateValueCallback для указания на метод, выполняющий проверку достоверности данных. Это гарантирует корректное поведение независимо от того, что применяется для получения/установки свойства зависимости — XAML или код.

Построение специального свойства зависимости

Если вы уже слегка запутались, то это совершенно нормально. Создание свойств зависимости может требовать некоторого времени на привыкание. Однако, хорошо это или плохо, но так выглядит часть процесса построения многих специальных элементов управления WPF, поэтому давайте посмотрим, как создается свойство зависимости.

Начните с создания приложения WPF по имени CustomDepPropApp. В меню Project (Проект) выберите пункт Add User Control (Добавить пользовательский элемент управления) и создайте элемент с именем ShowNumberControl.xaml (рис. 31.1).

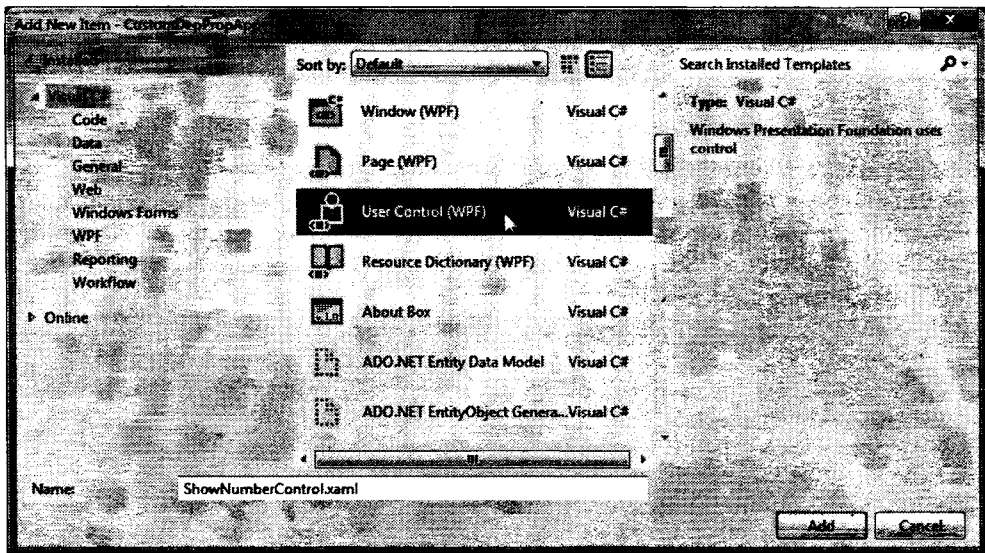


Рис. 31.1. Вставка нового специального элемента управления UserControl

На заметку! Описание элемента управления UserControl из WPF будет представлено позже в этой главе.

Подобно окну, типы UserControl в WPF имеют файл XAML и связанный файл кода. Измените XAML-разметку пользовательского элемента, добавив в Grid элемент Label:

```
<UserControl x:Class="CustomDepPropApp.ShowNumberControl"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  mc:Ignorable="d"
  d:DesignHeight="300" d:DesignWidth="300">
  <Grid>
    <Label x:Name="numberDisplay" Height="50" Width="200" Background="LightBlue"/>
  </Grid>
</UserControl>
```

В файле кода для этого элемента создайте обычное свойство .NET, которое упаковывает значение int и устанавливает новое значение для свойства Content элемента Label:

```
public partial class ShowNumberControl : UserControl
{
    public ShowNumberControl()
    {
        InitializeComponent();
    }

    // Обычное свойство .NET.
    private int currNumber = 0;
```

```

public int CurrentNumber
{
    get { return currNumber; }
    set
    {
        currNumber = value;
        numberDisplay.Content = CurrentNumber.ToString();
    }
}
}

```

Теперь добавьте в XAML-определение окна объявление экземпляра специального элемента управления внутри диспетчера компоновки `StackPanel`. Поскольку этот специальный элемент управления не входит в состав основных сборок WPF, понадобится определить специальное пространство имен XML, отображаемое на этот элемент (см. главу 27). Ниже показана необходимая разметка:

```

<Window x:Class="CustomDepPropApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:myCtrls="clr-namespace:CustomDepPropApp"
  Title="Simple Dependency Property App" Height="150" Width="250"
  WindowStartupLocation="CenterScreen">
  <StackPanel>
    <myCtrls:ShowNumberControl x:Name="myShowNumberCtrl" CurrentNumber="100"/>
  </StackPanel>
</Window>

```

Как видите, визуальный конструктор Visual Studio вроде бы корректно отображает значение, установленное в свойстве `CurrentNumber` (рис. 31.2).

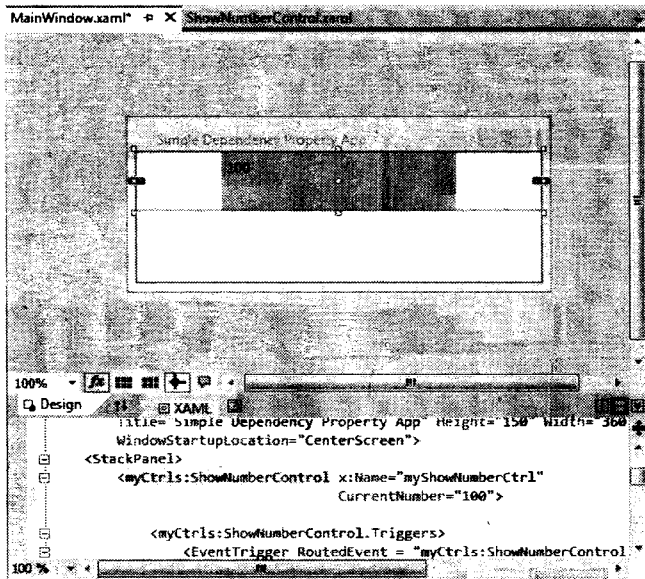


Рис. 31.2. Свойство выглядит вполне работоспособным

А что, если к свойству `CurrentNumber` нужно применить объект анимации, чтобы значение свойства изменялось от 100 до 200 в течение периода в 10 секунд? Если это требуется сделать в разметке, то раздел `<myCtrls:ShowNumberControl>` можно изменить следующим образом:

```

<myCtrls:ShowNumberControl x:Name="myShowNumberCtrl" CurrentNumber="100">
  <myCtrls:ShowNumberControl.Triggers>
    <EventTrigger RoutedEvent = "myCtrls:ShowNumberControl.Loaded">
      <EventTrigger.Actions>
        <BeginStoryboard>
          <Storyboard TargetProperty = "CurrentNumber">
            <Int32Animation From = "100" To = "200" Duration = "0:0:10"/>
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger.Actions>
    </EventTrigger>
  </myCtrls:ShowNumberControl.Triggers>
</myCtrls:ShowNumberControl>

```

Если попробовать запустить это приложение, то объект анимации не сможет найти подходящую цель и потому будет проигнорирован. Причина в том, что свойство `CurrentNumber` не было зарегистрировано как свойство зависимости! Чтобы исправить это, вернитесь к файлу кода для нашего специального элемента управления и полностью закомментируйте текущую логику свойства (включая закрытое поддерживающее поле). Теперь поместите курсор внутрь контекста класса и введите фрагмент кода `propdp` (рис. 31.3).

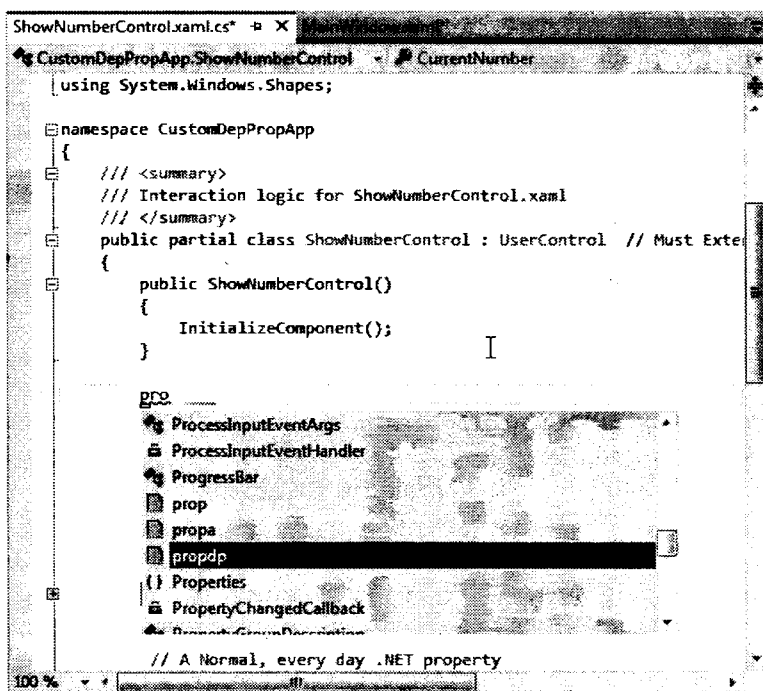


Рис. 31.3. Фрагмент кода `propdp` предоставляет начальную точку для построения свойства зависимости

После ввода `propdp` два раза нажмите клавишу `<Tab>`. Фрагмент кода развернется в базовый каркас свойства зависимости (рис. 31.4).

Простейшая версия свойства `CurrentNumber` будет выглядеть так:

```

public partial class ShowNumberControl : UserControl
{
    public int CurrentNumber
    {
        get { return (int)GetValue(CurrentNumberProperty); }
        set { SetValue(CurrentNumberProperty, value); }
    }
}

```

```

public static readonly DependencyProperty CurrentNumberProperty =
    DependencyProperty.Register("CurrentNumber",
        typeof(int),
        typeof>ShowNumberControl),
        new UIPropertyMetadata(0));
...
}

```

Это похоже на то, что делалось в реализации свойства Height; однако фрагмент кода `propdp` регистрирует свойство непосредственно в теле, а не в статическом конструкторе (что лучше). Также обратите внимание, что объект `UIPropertyMetadata` используется для определения стандартного целого значения (0) вместо более сложного объекта `FrameworkPropertyMetadata`.

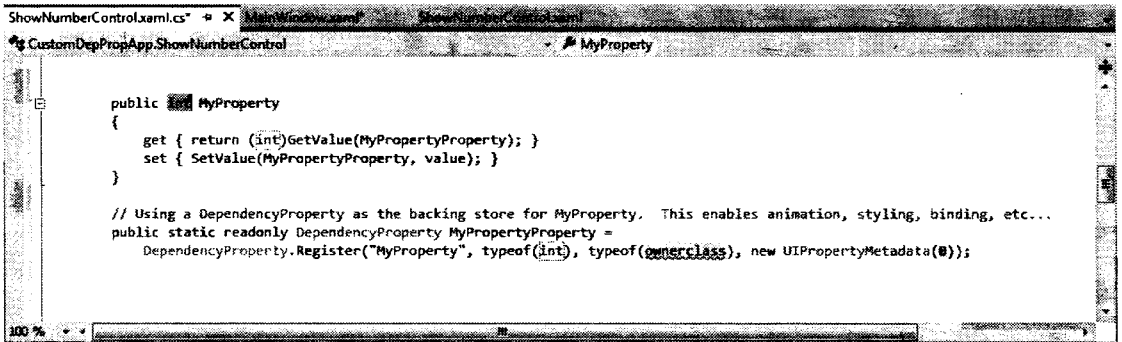


Рис. 31.4. Развернутый фрагмент кода

Добавление процедуры проверки достоверности данных

Теперь у нас есть свойство зависимости по имени `CurrentNumber`, но анимация все равно не наблюдается. Теперь необходимо указать функцию, которая вызывается для выполнения проверки достоверности. В этом примере предположим, что нужно гарантировать нахождение значения свойства `CurrentNumber` в диапазоне от 0 до 500.

Для этого добавьте в метод `DependencyProperty.Register()` финальный аргумент типа `ValidateValueCallback`, указывающий на метод по имени `ValidateCurrentNumber`.

Здесь `ValidateValueCallback` — это делегат, который может указывать только на методы, возвращающие `bool` и принимающие единственный аргумент типа `object`. Этот `object` представляет присваиваемое новое значение. Реализация `ValidateCurrentNumber` должна возвращать `true`, если входное значение находится в заданном диапазоне, и `false` — в противном случае:

```

public static readonly DependencyProperty CurrentNumberProperty =
    DependencyProperty.Register("CurrentNumber", typeof(int),
        typeof>ShowNumberControl),
        new UIPropertyMetadata(100),
        new ValidateValueCallback(ValidateCurrentNumber));

public static bool ValidateCurrentNumber(object value)
{
    // Очень простое бизнес-правило: значение должно находиться в диапазоне между 0 и 500.
    if (Convert.ToInt32(value) >= 0 && Convert.ToInt32(value) <= 500)
        return true;
    else
        return false;
}

```

Реагирование на изменение свойства

Итак, допустимое число уже имеется, но анимации все еще нет. Последнее изменение, которое потребуется внести — передать во втором аргументе конструктора `UIPropertyMetadata` объект `PropertyChangedCallback`. Этот делегат может указывать на любой метод, принимающий `DependencyObject` в качестве первого параметра и `DependencyPropertyChangedEventArgs` — в качестве второго. Сначала модифицируйте код следующим образом:

```
// Обратите внимание на второй параметр конструктора UIPropertyMetadata.
public static readonly DependencyProperty CurrentNumberProperty =
    DependencyProperty.Register("CurrentNumber", typeof(int),
        typeof>ShowNumberControl',
        new UIPropertyMetadata(100,
            new PropertyChangedCallback(CurrentNumberChanged)),
            new ValidateValueCallback(ValidateCurrentNumber));
```

Конечной целью внутри метода `CurrentNumberChanged()` является изменение значения свойства `Content` объекта `Label` новым, присвоенным свойством `CurrentNumber`. Но тут мы сталкиваемся с серьезной проблемой: метод `CurrentNumberChanged()` является статическим, т.к. он должен работать со статическим объектом `DependencyProperty`.

Каким же образом тогда получить доступ к `Label` для текущего экземпляра `ShowNumberControl`? Эта ссылка находится в первом параметре `DependencyObject`. Новое значение можно найти, используя входные аргументы события. Ниже показан код, необходимый для изменения свойства `Content` объекта `Label`:

```
private static void CurrentNumberChanged(DependencyObject depObj,
    DependencyPropertyChangedEventArgs args)
{
    // Привести DependencyObject к ShowNumberControl.
    ShowNumberControl c = (ShowNumberControl)depObj;

    // Получить элемент Label в ShowNumberControl.
    Label theLabel = c.numberDisplay;

    // Установить для Label новое значение.
    theLabel.Content = args.NewValue.ToString();
}
```

Видите, насколько долгий путь пришлось пройти, чтобы всего лишь изменить содержимое метки! Преимущество заключается в том, что теперь свойство зависимости `CurrentNumber` может быть целью для стиля WPF, объекта анимации, операций привязки данных и т.д. Если сейчас снова запустить приложение, вы увидите, что значение изменяется во время выполнения.

На этом обзор свойств зависимости WPF завершен. Хотя теперь вы должны гораздо лучше понимать, что эти конструкции позволяют делать, и как создавать собственные такие свойства, имейте в виду, что о многие детали здесь не раскрывались.

Если вам однажды понадобится создать множество собственных элементов управления, поддерживающих специальные свойства, загляните в подраздел “Properties” (“Свойства”) узла “WPF Fundamentals” (“Основы WPF”) в документации .NET Framework 4.5 SDK. Там вы найдете намного больше примеров построения свойств зависимости, присоединяемых свойств, различных способов конфигурирования метаданных и массу других деталей.

Маршрутизируемые события

Свойства — не единственная программная конструкция .NET, которая потребовала небольшой модернизации для успешной работы с API-интерфейсом WPF. Стандартная модель событий CLR также претерпела некоторые усовершенствования, чтобы обеспечить обработку событий в соответствии с XAML-описанием дерева объектов. Предположим, что имеется новый проект WPF-приложения по имени `WPF RoutedEvents`. Добавьте в XAML-описание первоначального окна следующий элемент управления `<Button>` с определением некоторого сложного содержимого:

```
<Button Name="btnClickMe" Height="75" Width = "250"
    Click ="btnClickMe_Clicked">
  <StackPanel Orientation ="Horizontal">
    <Label Height="50" FontSize ="20">Fancy Button!</Label>
    <Canvas Height ="50" Width ="100" >
      <Ellipse Name = "outerEllipse" Fill ="Green" Height ="25"
        Width ="50" Cursor="Hand" Canvas.Left="25" Canvas.Top="12"/>
      <Ellipse Name = "innerEllipse" Fill ="Yellow" Height = "15" Width ="36"
        Canvas.Top="17" Canvas.Left="32"/>
    </Canvas>
  </StackPanel>
</Button>
```

Обратите внимание, что в открывающем определении `<Button>` было обработано событие `Click` путем указания имени метода, который должен вызываться при возникновении события. Событие `Click` работает с делегатом `RoutedEventHandler`, который ожидает обработчика события, принимающего `object` в первом параметре и `System.Windows.RoutedEventArgs` — во втором. Реализуем этот обработчик, как показано ниже:

```
public void btnClickMe_Clicked(object sender, RoutedEventArgs e)
{
    // Делать что-нибудь, когда на кнопке произведен щелчок.
    MessageBox.Show("Clicked the button");
}
```

Если вы запустите приложение, то увидите это окно сообщения, независимо от того, на какой части кнопки был выполнен щелчок (зеленый `Ellipse`, желтый `Ellipse`, `Label` или поверхность `Button`). И это хорошо. Представьте, насколько громоздким оказалась бы обработка событий WPF, если бы пришлось обрабатывать событие `Click` для каждого подэлемента. И дело не только в том, что создание отдельных обработчиков событий для каждого аспекта `Button` было бы трудоемкой задачей, а еще и в том, что в результате получился бы сложный и громоздкий код, требующий весьма трудоемкого сопровождения.

К счастью, *маршрутизируемые события* WPF позволяют устроить все так, чтобы автоматически вызывался единый обработчик события `Click`, независимо от того, на какую часть кнопки пришелся щелчок. То есть модель маршрутизируемых событий автоматически распространяет события вверх (или вниз) по дереву объектов в поисках подходящего обработчика.

Точнее говоря, маршрутизируемое событие может использовать три *стратегии маршрутизации*. Если событие распространяется от исходной точки в другие области внутри дерева объектов, говорят, что это *пузырьковое событие*. А если событие распространяется от внешнего элемента (например, `Window`) вниз к исходной точке, такое событие называется *туннельным*. Наконец, если событие инициируется и обрабатывается только исходным элементом (его можно было бы назвать нормальным событием CLR), говорят, что это *прямое событие*.

Роль маршрутизируемых пузырьковых событий

Если в текущем примере пользователь щелкнет на внутреннем овале желтого цвета, событие Click поднимется на следующий уровень (Canvas), затем на StackPanel и, наконец, на уровень Button, где и будет обработано. Аналогичным образом, если пользователь щелкнет на Label, событие всплывет на уровень StackPanel и, наконец, попадет в элемент Button.

Благодаря этому шаблону пузырьковых маршрутизируемых событий, не нужно беспокоиться о регистрации специфичных обработчиков событий Click для всех членов составного элемента управления. Тем не менее, если требуется выполнить специальную логику обработки щелчков для нескольких элементов внутри одного и того же дерева объектов, то это можно сделать.

В целях иллюстрации предположим, что щелчок на элементе управления outerEllipse должен быть обработан уникальным образом. Сначала обработайте событие MouseDown для этого подэлемента (графически визуализируемые типы вроде Ellipse не поддерживают событие Click, но могут отслеживать действия кнопки мыши через события MouseDown, MouseUp и т.п.):

```
<Button Name="btnClickMe" Height="75" Width = "250"
    Click ="btnClickMe_Clicked">
<StackPanel Orientation ="Horizontal">
    <Label Height="50" FontSize ="20">Fancy Button!</Label>
    <Canvas Height ="50" Width ="100" >
        <Ellipse Name = "outerEllipse" Fill ="Green"
            Height ="25" MouseDown ="outerEllipse_MouseDown"
            Width ="50" Cursor ="Hand" Canvas.Left="25" Canvas.Top="12"/>
        <Ellipse Name = "innerEllipse" Fill ="Yellow" Height = "15" Width ="36"
            Canvas.Top="17" Canvas.Left="32"/>
    </Canvas>
</StackPanel>
</Button>
```

Затем реализуйте соответствующий обработчик событий, который для демонстрационных целей просто изменяет свойство Title главного окна, как показано ниже:

```
public void outerEllipse_MouseDown(object sender, MouseButtonEventArgs e)
{
    // Изменить заголовок окна.
    this.Title = "You clicked the outer ellipse!";
}
```

Теперь можно выполнять различные действия в зависимости от того, на чем конкретно щелкнул конечный пользователь (на внешнем эллипсе или в любом месте внутри области кнопки).

На заметку! Маршрутизируемые пузырьковые события всегда перемещаются от исходной точки до следующего определяющего контекста. Поэтому в рассмотренном примере щелчок на элементе innerEllipse привел бы к попаданию события в контейнер Canvas, а не в элемент outerEllipse, т.к. оба элемента являются типами Ellipse внутри области Canvas.

Продолжение или прекращение пузырькового распространения

В текущей ситуации, если пользователь щелкнет на объекте outerEllipse, запустится зарегистрированный обработчик событий MouseDown для данного объекта Ellipse, и в этот момент событие всплывет до события Click кнопки. Чтобы прекра-

тить пузырьковое распространение по дереву объектов, необходимо установить для свойства `Handled` параметра `MouseButtonEventArgs` значение `true`:

```
public void outerEllipse_MouseDown(object sender, MouseButtonEventArgs e)
{
    // Изменить заголовок окна.
    this.Title = "You clicked the outer ellipse!";

    // Остановить пузырьковое распространение!
    e.Handled = true;
}
```

В этом случае заголовок окна изменится, но окно `MessageBox` не отобразится обработчиком события `Click` элемента `Button`. По сути, маршрутизируемые пузырьковые события позволяют сложной группе содержимого действовать либо как единый логический элемент (например, `Button`), либо как отдельные элементы (например, `Ellipse` внутри `Button`).

Роль маршрутизируемых туннельных событий

Строго говоря, маршрутизируемые события могут быть по своей природе *пузырьковыми* (как только что было описано) или *туннельными*. Туннельные события (которые все начинаются с префикса `Preview`, например, `PreviewMouseDown`) спускаются от элемента верхнего уровня во вложенные контексты дерева объектов. В общем случае, каждому пузырьковому событию в библиотеках базовых классов WPF соответствует туннельное событие, которое возникает *перед* его пузырьковым двойником. Например, перед возникновением пузырькового события `MouseDown` сначала инициируется туннельное событие `PreviewMouseDown`.

Обработка туннельных событий выглядит как обработка любых других событий: нужно просто назначить имя обработчика события в разметке XAML (или, если необходимо, использовать соответствующий синтаксис обработки событий на C# в файле кода) и реализовать этот обработчик в файле кода. Для демонстрации взаимодействия туннельных и пузырьковых событий начните с обработки события `PreviewMouseDown` для объекта `outerEllipse`, как показано ниже:

```
<Ellipse Name = "outerEllipse" Fill ="Green" Height ="25"
    MouseDown ="outerEllipse_MouseDown"
    PreviewMouseDown ="outerEllipse_PreviewMouseDown"
    Width ="50" Cursor="Hand" Canvas.Left="25" Canvas.Top="12"/>
```

Затем измените все обработчики событий в существующем определении класса C#, добавив данные о текущем событии в переменную-член `mouseActivity` типа `string` с использованием входного объекта аргументов события. Это позволит наблюдать за потоком возникновения событий в фоновом режиме.

```
public partial class MainWindow : Window
{
    string mouseActivity = string.Empty;
    public MainWindow()
    {
        InitializeComponent();
    }
    public void btnClickMe_Clicked(object sender, RoutedEventArgs e)
    {
        AddEventInfo(sender, e);
        MessageBox.Show(mouseActivity, "Your Event Info");
        // Очистить строку для следующего использования.
        mouseActivity = "";
    }
}
```

```

private void AddEventInfo(object sender, RoutedEventArgs e)
{
    mouseActivity += string.Format(
        "{0} sent a {1} event named {2}.\n", sender,
        e.RoutedEvent.RoutingStrategy,
        e.RoutedEvent.Name);
}

private void outerEllipse_MouseDown(object sender, MouseButtonEventArgs e)
{
    AddEventInfo(sender, e);
}

private void outerEllipse_PreviewMouseDown(object sender, MouseButtonEventArgs e)
{
    AddEventInfo(sender, e);
}
}

```

Обратите внимание, что ни в одном обработчике событий пузырьковое распространение не прекращается. Запустив это приложение, вы увидите окно с уникальным сообщением, которое зависит от того, где конкретно на кнопке был произведен щелчок. На рис. 31.5 показан результат щелчка на внешнем объекте `Ellipse`.

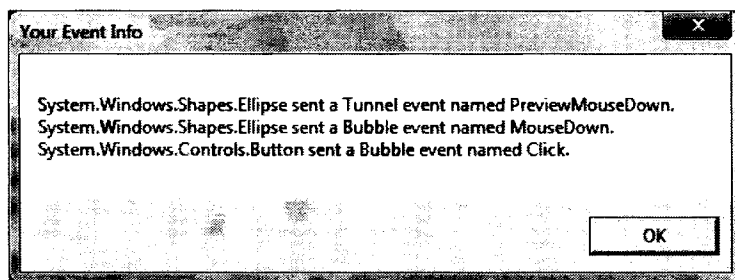


Рис. 31.5. Сначала туннельное, а затем пузырьковое распространение

Итак, почему же события WPF обычно встречаются парами (одно туннельное и одно пузырьковое)? Ответ заключается в том, что с помощью предварительного отслеживания событий можно выполнять любую специализированную логику (проверку достоверности данных, отключение пузырькового распространения и т.п.), прежде чем будет запущен пузырьковый компонент. В качестве примера предположим, что имеется элемент `TextBox`, который должен содержать только числовые данные. Если пользователь введет какие-то нечисловые данные, в обработчике события `PreviewKeyDown` можно остановить пузырьковое распространение, присвоив свойству `Handled` значение `true`.

Как и можно было предположить, при построении специального элемента управления, который содержит специальные события, событие можно реализовать так, чтобы оно могло распространяться пузырьковым (или туннельным) образом по дереву разметки XAML. В настоящей главе не рассматривается создание специальных маршрутизируемых событий (хотя этот процесс не особенно отличается от построения специального свойства зависимости). Если интересно, загляните в раздел “Routed Events Overview” (“Обзор маршрутизируемых событий”) документации .NET Framework 4.5 SDK. Там вы найдете множество полезных подсказок.

Логические деревья, визуальные деревья и стандартные шаблоны

Существует еще несколько тем, которые следует рассмотреть, прежде чем приступить к изучению построения специальных элементов управления. В частности, необходимо понять разницу между *логическим деревом*, *визуальным деревом* и *стандартным шаблоном*. При вводе разметки XAML в Visual Studio, либо в таком редакторе, как *Кахамл*, разметка становится *логическим представлением* документа XAML. Точно также, при написании кода C#, добавляющего новые элементы в *StackPanel*, новые элементы вставляются в логическое дерево. В сущности, логическое представление показывает, как содержимое будет позиционировано внутри различных диспетчеров компоновки для главного элемента *Window* (или другого корневого элемента, такого как *Page* или *NavigationWindow*).

Однако за каждым логическим деревом стоит более сложное представление, которое называется *визуальным деревом* и внутренне используется инфраструктурой WPF для корректной визуализации элементов на экране. Внутри любого визуального дерева находится вся информация шаблонов и стилей, применяемых для визуализации каждого объекта, включая все необходимые рисунки, фигуры, визуальные объекты и анимации.

Полезно уяснить разницу между логическим и визуальным деревьями, потому что при построении шаблона специального элемента, по сути, вместо всего стандартного визуального дерева (или его части) элемента управления подставляется ваш вариант. Таким образом, если необходимо, чтобы кнопка *Button* визуализировалась в виде звездообразной фигуры, можно определить новый шаблон подобного рода и включить его в визуальное дерево *Button*. Логически это тот же тип *Button*, и он поддерживает все те же самые свойства, методы и события. Но визуально он выглядит совершенно по-другому. Один лишь этот факт делает WPF исключительно удобным API-интерфейсом, учитывая, что другие инструментальные наборы потребовали бы в такой ситуации создания совершенно нового класса. В WPF достаточно просто определить новую разметку.

На заметку! Элементы управления WPF часто называются *лишенными внешности*. Это означает, что внешний вид WPF-элемента полностью независим от его поведения и допускает настройку.

Программный просмотр логического дерева

Хотя анализ логического дерева окна во время выполнения — не слишком частое действие при программировании с использованием WPF, все же стоит упомянуть, что в пространстве имен *System.Windows* определен класс *LogicalTreeHelper*, который позволяет просматривать структуру логического дерева во время выполнения. Для иллюстрации связи между логическими деревьями, визуальными деревьями и шаблонами элементов управления создайте новое приложение WPF по имени *TreesAndTemplatesApp*.

Добавьте в разметку окна два элемента *Button* и большой доступный только для чтения элемент *TextBox* с активными линейками прокрутки. Воспользуйтесь IDE-средой для обработки события *Click* каждой кнопки. Ниже показана необходимая разметка XAML:

```
<Window x:Class="TreesAndTemplatesApp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Fun with Trees and Templates" Height="518"
        Width="836" WindowStartupLocation="CenterScreen">
```

```

<DockPanel LastChildFill="True">
  <Border Height="50" DockPanel.Dock="Top" BorderBrush="Blue">
    <StackPanel Orientation="Horizontal">
      <Button x:Name="btnShowLogicalTree" Content="Logical Tree of Window"
        Margin="4" BorderBrush="Blue"
        Height="40" Click="btnShowLogicalTree_Click"/>
      <Button x:Name="btnShowVisualTree" Content="Visual Tree of Window"
        BorderBrush="Blue" Height="40" Click="btnShowVisualTree_Click"/>
    </StackPanel>
  </Border>
  <TextBox x:Name="txtDisplayArea" Margin="10"
    Background="AliceBlue" IsReadOnly="True"
    BorderBrush="Red" VerticalScrollBarVisibility="Auto"
    HorizontalScrollBarVisibility="Auto" />
</DockPanel>
</Window>

```

Внутри файла кода C# определите переменную-член `dataToShow` типа `string`. Теперь в обработчике `Click` объекта `btnShowLogicalTree` вызовите вспомогательную функцию, которая будет вызывать себя рекурсивно для заполнения строковой переменной логическим деревом `Window`. Для этого воспользуйтесь статическим методом `GetChildren()` объекта `LogicalTreeHelper`. Ниже показан код.

```

private void btnShowLogicalTree_Click(object sender, RoutedEventArgs e)
{
    dataToShow = "";
    BuildLogicalTree(0, this);
    this.txtDisplayArea.Text = dataToShow;
}

void BuildLogicalTree(int depth, object obj)
{
    // Добавить имя типа к переменной-члену dataToShow.
    dataToShow += new string(' ', depth) + obj.GetType().Name + "\n";

    // Если элемент - не DependencyObject, пропустить его.
    if (!(obj is DependencyObject))
        return;

    // Выполнить рекурсивный вызов для каждого логического дочернего элемента.
    foreach (object child in LogicalTreeHelper.GetChildren(obj as DependencyObject))
        BuildLogicalTree(depth + 5, child);
}

```

Запустив приложение и щелкнув на кнопке `Logical Tree of Window` (Логическое дерево `Window`), вы увидите в текстовой области древовидное представление — почти точную копию первоначальной разметки XAML (рис. 31.6).

Программный просмотр визуального дерева

Визуальное дерево `Window` также можно просмотреть во время выполнения с помощью класса `VisualTreeHelper` из пространства имен `System.Windows.Media`. Ниже приведена реализация обработчика события `Click` для второго элемента `Button` (`btnShowVisualTree`), где аналогичная рекурсивная логика применяется для построения текстового представления визуального дерева:

```

private void btnShowVisualTree_Click(object sender, RoutedEventArgs e)
{
    dataToShow = "";
    BuildVisualTree(0, this);
    this.txtDisplayArea.Text = dataToShow;
}

```

```

void BuildVisualTree(int depth, DependencyObject obj)
{
    // Добавить имя типа к переменной-члену dataToShow.
    dataToShow += new string(' ', depth) + obj.GetType().Name + "\n";
    // Выполнить рекурсивный вызов для каждого дочернего элемента.
    for (int i = 0; i < VisualTreeHelper.GetChildrenCount(obj); i++)
        BuildVisualTree(depth + 1, VisualTreeHelper.GetChild(obj, i));
}

```

Как показано на рис. 31.7, визуальное дерево содержит множество низкоуровневых агентов визуализации, таких как `ContentPresenter`, `AdornerDecorator`, `TextBoxLineDrawingVisual` и т.д.

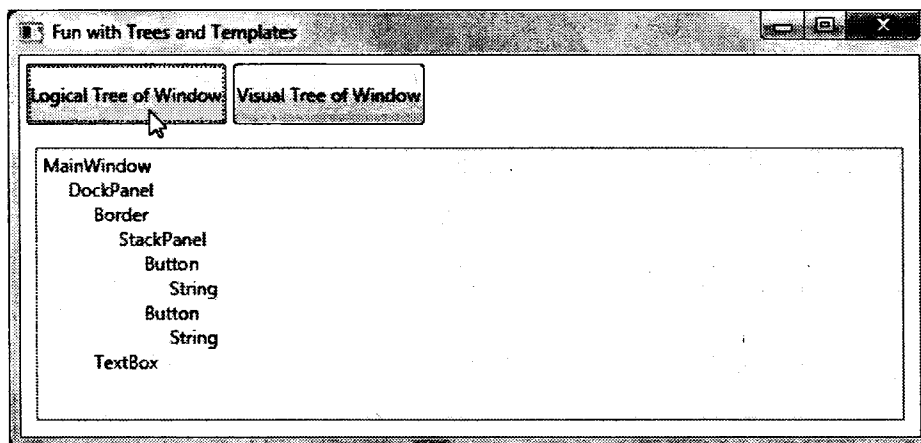


Рис. 31.6. Просмотр логического дерева во время выполнения

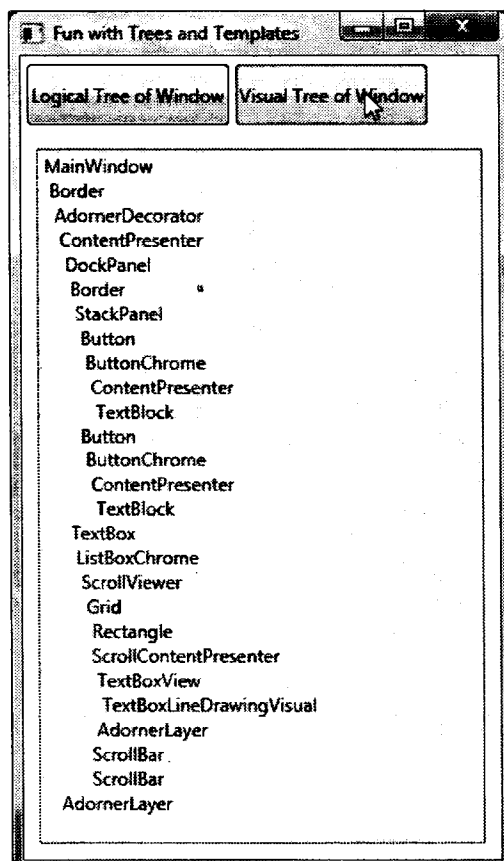


Рис. 31.7. Просмотр визуального дерева во время выполнения

Программный просмотр стандартного шаблона элемента управления

Вспомните, что визуальное дерево используется в WPF для определения визуализации элемента Window и всех содержащихся в нем элементов. Каждый элемент управления хранит собственный набор команд внутри своего стандартного шаблона. С программной точки зрения любой шаблон может быть представлен как экземпляр класса `ControlTemplate`. Кроме того, стандартный шаблон элемента можно получить через свойство `Template`, как показано ниже:

```
// Получить стандартный шаблон элемента Button.
Button myBtn = new Button();
ControlTemplate template = myBtn.Template;
```

Аналогично, можно создать в коде новый объект `ControlTemplate` и подключить его к свойству `Template` элемента управления:

```
// Подключить новый шаблон для кнопки.
Button myBtn = new Button();
ControlTemplate customTemplate = new ControlTemplate();

// Предположить, что этот метод добавляет весь код для звездообразного шаблона.
MakeStarTemplate(customTemplate);
myBtn.Template = customTemplate;
```

Хотя новый шаблон можно построить в коде, это обычно делается в XAML-разметке. Однако прежде чем приступить к построению собственных шаблонов, давайте завершим текущий пример и добавим возможность просмотра стандартного шаблона WPF-элемента во время выполнения. Это может оказаться действительно удобным способом ознакомления с общей композицией шаблона. Для начала добавьте в разметку окна новый контейнер `StackPanel` с элементами управления, стыкованный с левой стороной главной панели `DockPanel`:

```
<Border DockPanel.Dock="Left" Margin="10" BorderBrush="DarkGreen"
        BorderThickness="4" Width="358">
  <StackPanel>
    <Label Content="Enter Full Name of WPF Control" Width="340" FontWeight="DemiBold" />
    <TextBox x:Name="txtFullName" Width="340" BorderBrush="Green"
             Background="BlanchedAlmond" Height="22"
             Text="System.Windows.Controls.Button" />
    <Button x:Name="btnTemplate" Content="See Template" BorderBrush="Green"
            Height="40" Width="100" Margin="5"
            Click="btnTemplate_Click" HorizontalAlignment="Left" />
    <Border BorderBrush="DarkGreen" BorderThickness="2" Height="260"
            Width="301" Margin="10" Background="LightGreen" >
      <StackPanel x:Name="stackTemplatePanel" />
    </Border>
  </StackPanel>
</Border>
```

Обратите внимание на пустой элемент `StackPanel` по имени `stackTemplatePanel`: на него будут производиться ссылки в коде. На рис. 31.8 показано, как примерно должно выглядеть окно.

Текстовая область сверху слева позволяет вводить полностью заданное имя элемента управления WPF, находящегося в сборке `PresentationFramework.dll`. После загрузки библиотеки экземпляр объекта динамически загружается и отображается в большом квадрате внизу слева.

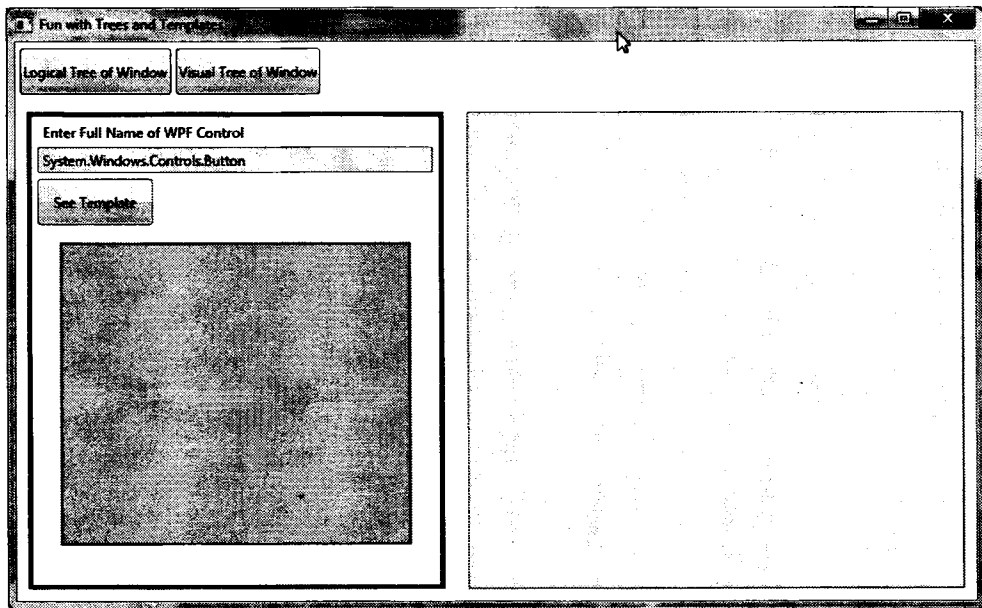


Рис. 31.8. Обновленный пользовательский интерфейс окна

И, наконец, стандартный шаблон элемента управления будет отображаться в текстовой области справа. Добавьте в свой класс C# новую переменную-член типа `Control`:

```
private Control ctrlToExamine = null;
```

Ниже приведен остальной код, который требует импорта пространств имен `System.Reflection`, `System.Xml` и `System.Windows.Markup`:

```
private void btnTemplate_Click(object sender, RoutedEventArgs e)
{
    dataToShow = "";
    ShowTemplate();
    this.txtDisplayArea.Text = dataToShow;
}

private void ShowTemplate()
{
    // Удалить элемент, находящийся в области предварительного просмотра.
    if (ctrlToExamine != null)
        stackTemplatePanel.Children.Remove(ctrlToExamine);
    try
    {
        // Загрузить PresentationFramework и создать экземпляр
        // указанного элемента управления. Установить его размеры
        // для отображения, а затем добавить в пустой контейнер StackPanel.
        Assembly asm = Assembly.Load("PresentationFramework, Version=4.0.0.0, " +
            "Culture=neutral, PublicKeyToken=31bf3856ad364e35");
        ctrlToExamine = (Control)asm.CreateInstance(txtFullName.Text);
        ctrlToExamine.Height = 200;
        ctrlToExamine.Width = 200;
        ctrlToExamine.Margin = new Thickness(5);
        stackTemplatePanel.Children.Add(ctrlToExamine);

        // Определить параметры XML для выполнения отступов.
        XmlWriterSettings xmlSettings = new XmlWriterSettings();
        xmlSettings.Indent = true;
    }
    catch { }
}
```

```

// Создать StringBuilder для хранения XAML-разметки.
StringBuilder strBuilder = new StringBuilder();

// Создать XmlWriter на основе существующих параметров.
XmlWriter xWriter = XmlWriter.Create(strBuilder, xmlSettings);

// Сохранить XAML-разметку в объекте XmlWriter на основе ControlTemplate.
XmlWriter.Save(ctrlToExamine.Template, xWriter);

// Отобразить XAML-разметку XAML в текстовом поле.
dataToShow = strBuilder.ToString();
}
catch (Exception ex)
{
    dataToShow = ex.Message;
}
}

```

Большую часть работы занимает отображение скомпилированного ресурса BAML на строку XAML. На рис. 31.9 показано финальное приложение в действии на примере вывода стандартного шаблона для элемента управления `System.Windows.Controls.DatePicker`.

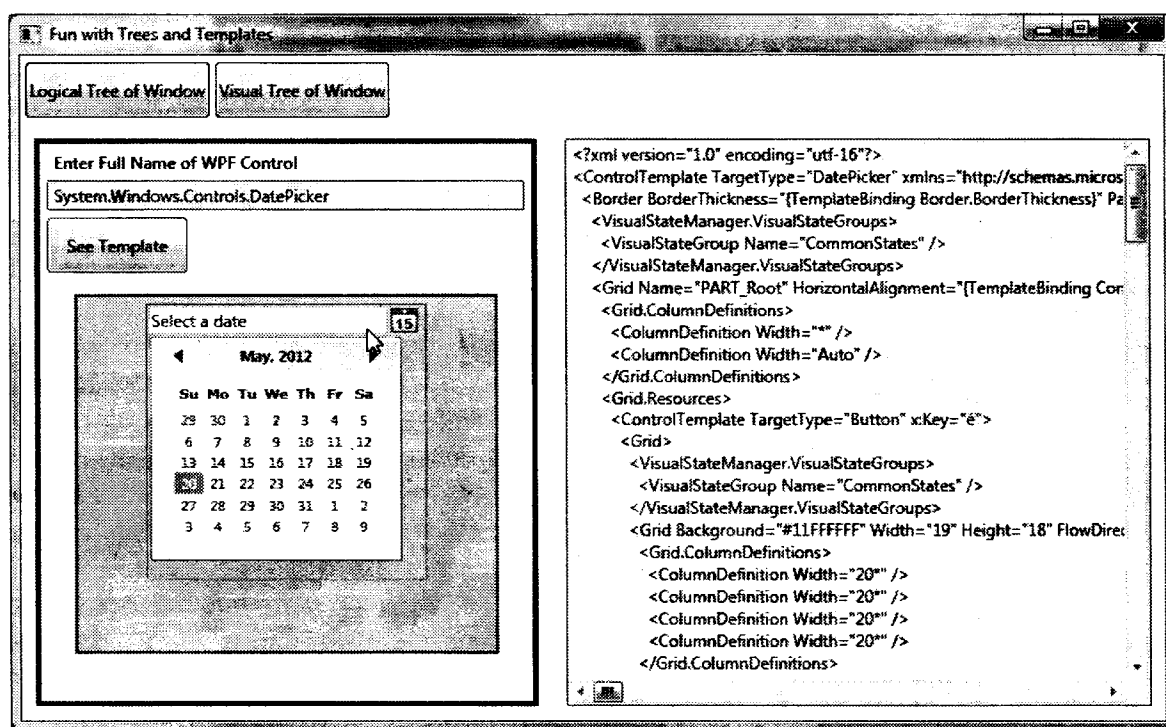


Рис. 31.9. Исследование элемента управления `ControlTemplate` во время выполнения

Теперь вы должны лучше представлять взаимосвязь логических деревьев, визуальных деревьев и стандартных шаблонов элементов управления. Оставшаяся часть главы будет посвящена построению специальных шаблонов и пользовательских элементов управления.

Исходный код. Проект `TreesAndTemplatesApp` доступен в подкаталоге `Chapter 31`.

Построение специального шаблона элемента управления с помощью инфраструктуры триггеров

Специальный шаблон для элемента управления можно создать исключительно в коде C#. При таком подходе можно добавлять данные к объекту `ControlTemplate` и затем присвоить его свойству `Template` элемента управления. Однако обычно внешний вид `ControlTemplate` определяется с помощью разметки XAML, а с помощью небольших (или крупных) фрагментов кода осуществляется управление поведением элемента во время выполнения.

В оставшейся части этой главы будет показано, как строить специальные шаблоны с помощью Visual Studio. Попутно вы узнаете об инфраструктуре триггеров WPF, о диспетчере визуального состояния (Visual State Manager — VSM) и об использовании анимации для встраивания визуальных подсказок конечным пользователям. Вы увидите, что создание сложных шаблонов с помощью только среды Visual Studio приводит к большому объему вводимого кода и выполняемой работы. Разумеется, для построения шаблонов производственного уровня лучше применять Expression Blend. Однако в настоящем издании книги описание Expression Blend не приводится, так что мы просто приступим к созданию очередной разметки.

Создайте новое приложение WPF по имени `ButtonTemplate`. В этом проекте основной интерес представляют механизмы создания и использования шаблонов, так что разметка для главного окна очень проста:

```
<Window x:Class="ButtonTemplate.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Fun with Templates" Height="350" Width="525">
  <StackPanel>
    <Button x:Name="myButton" Width="100" Height="100" Click="myButton_Click"/>
  </StackPanel>
</Window>
```

В обработчике события `Click` просто отобразите окно сообщения (вызовом `MessageBox.Show()`) с подтверждением щелчка на элементе управления. При создании шаблонов элементов управления помните, что *поведение* элемента управления неизменно, но его *внешний вид* может меняться.

В настоящее время этот элемент `Button` визуализируется с использованием стандартного шаблона, который, как было показано в последнем примере, представляет собой ресурс BAML внутри заданной сборки WPF. Когда необходимо определить собственный шаблон, вы, в сущности, заменяете стандартное визуальное дерево своим вариантом. Для начала модифицируйте определение элемента `<Button>`, указав новый шаблон с помощью синтаксиса “свойство-элемент”. Этот шаблон придаст элементу управления округлый вид:

```
<Button x:Name="myButton" Width="100" Height="100" Click="myButton_Click">
  <Button.Template>
    <ControlTemplate>
      <Grid x:Name="controlLayout">
        <Ellipse x:Name="buttonSurface" Fill = "LightBlue"/>
        <Label x:Name="buttonCaption" VerticalAlignment = "Center"
          HorizontalAlignment = "Center"
          FontWeight = "Bold" FontSize = "20" Content = "OK!"/>
      </Grid>
    </ControlTemplate>
  </Button.Template>
</Button>
```

Здесь определен шаблон, состоящий из именованного элемента Grid, который содержит именованные элементы Ellipse и Label. Поскольку в Grid не определены строки и столбцы, каждый дочерний элемент укладывается поверх предыдущего элемента управления, позволяя центрировать содержимое. Если теперь запустить приложение, можно заметить, что событие Click инициируется *только* в том случае, когда курсор мыши находится в границах Ellipse (т.е. не на углах, окружающих эллипс)! Это замечательная возможность архитектуры шаблонов WPF — нет нужды вычислять точное попадание курсора, проверять граничные условия или предпринимать другие низкоуровневые действия. Поэтому, если шаблон использовал объект Polygon для отображения какой-то необычной геометрии, можете быть уверены, что детали проверки попадания курсора будут соответствовать форме элемента управления, а не ограничивающего прямоугольника.

Шаблоны как ресурсы

В настоящее время наш шаблон включен в конкретный элемент управления Button, что ограничивает возможности его повторного использования. В идеале шаблон крутой кнопки стоило бы поместить в словарь ресурсов для многократного применения в разных проектах или, как минимум, переместить его в контейнер ресурсов для повторного использования в текущем проекте. Давайте перенесем локальный ресурс Button на уровень приложения. Для начала найдите свойство Template элемента Button в окне Properties (Свойства) — оно расположено в разделе Miscellaneous (Разные). Теперь щелкните на небольшом белом квадрате и в открывшемся контекстом меню выберите пункт Convert to New Resource... (Преобразовать в новый ресурс...), как показано на рис. 31.10. В результирующем диалоговом окне определите новый шаблон по имени RoundButtonTemplate и сохраните его на уровне Application (т.е. в файле App.xaml; рис. 31.11).

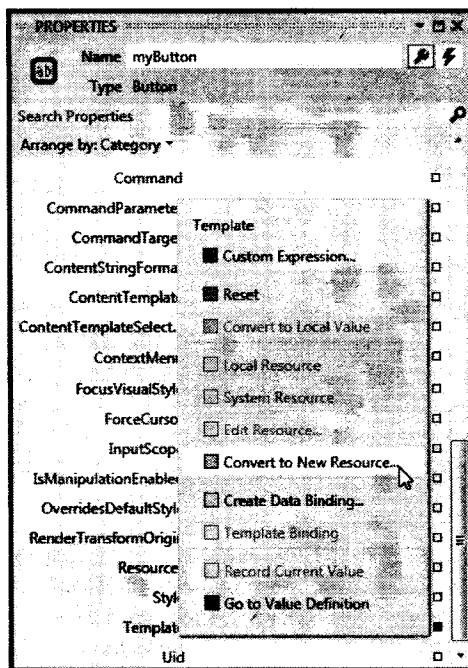


Рис. 31.10. Извлечение локального ресурса

После этого в разметке объекта Application появятся следующие данные:

```
<Application x:Class="ButtonTemplate.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  StartupUri="MainWindow.xaml">
  <Application.Resources>
    <ControlTemplate x:Key="RoundButtonTemplate">
      <Grid x:Name="controlLayout">
        <Ellipse x:Name="buttonSurface" Fill = "LightBlue"/>
        <Label x:Name="buttonCaption" VerticalAlignment = "Center"
          HorizontalAlignment = "Center"
          FontWeight = "Bold" FontSize = "20" Content = "OK!"/>
      </Grid>
    </ControlTemplate>
  </Application.Resources>
</Application>
```

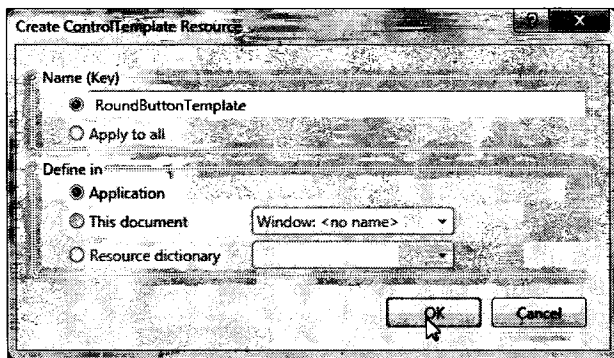


Рис. 31.11. Помещение ресурса в файл App.xaml

Теперь этот ресурс доступен всему приложению, и можно определять любое количество круглых кнопок. В целях тестирования создайте два дополнительных элемента управления Button, которые используют этот шаблон (обрабатывать событие Click для них не нужно).

```
<StackPanel>
  <Button x:Name="myButton" Width="100" Height="100"
    Click="myButton_Click"
    Template="{StaticResource RoundButtonTemplate}"></Button>
  <Button x:Name="myButton2" Width="100" Height="100"
    Template="{StaticResource RoundButtonTemplate}"></Button>
  <Button x:Name="myButton3" Width="100" Height="100"
    Template="{StaticResource RoundButtonTemplate}"></Button>
</StackPanel>
```

Встраивание визуальных подсказок с использованием триггеров

При определении специального шаблона все визуальные подсказки, реализованные стандартным шаблоном, удаляются. Например, стандартный шаблон кнопки содержит разметку, которая задает внешний вид элемента управления при возникновении определенных событий пользовательского интерфейса, таких как получение фокуса, щелчок кнопкой мыши, включение (или отключение) и т.д. Пользователи достаточно хорошо привыкли к подобным визуальным подсказкам, поскольку они придают элементу управления некоторую ощутимую реакцию. Однако шаблон RoundButtonTemplate не содержит предназначенной для этого разметки, так что вид элемента управления будет одинаков, независимо от действий мыши. В идеале элемент при щелчке на нем должен слегка менять вид (возможно, за счет изменения цвета или отбрасывания тени), чтобы уведомить пользователя об изменении визуального состояния.

Сразу после появления WPF единственным способом организации таких визуальных подсказок было добавление к шаблону любого количества триггеров, которые обычно изменяют значения свойств объекта либо запускают раскладку анимации (или делают то и другое), когда условие триггера истинно. Для примера модифицируйте разметку RoundButtonTemplate, чтобы при помещении курсора на поверхность элемента цвет переднего плана менялся на синий, а цвет фона на желтый, как показано ниже:

```
<ControlTemplate x:Key="RoundButtonTemplate" TargetType="Button">
  <Grid x:Name="controlLayout">
    <Ellipse x:Name="buttonSurface" Fill="LightBlue" />
    <Label x:Name="buttonCaption" Content="OK!" FontSize="20" FontWeight="Bold"
      HorizontalAlignment="Center" VerticalAlignment="Center" />
  </Grid>
```

```

<ControlTemplate.Triggers>
  <Trigger Property = "IsMouseOver" Value = "True">
    <Setter TargetName = "buttonSurface" Property = "Fill" Value = "Blue"/>
    <Setter TargetName = "buttonCaption" Property = "Foreground"
      Value = "Yellow"/>
  </Trigger>
</ControlTemplate.Triggers>
</ControlTemplate>

```

Вновь запустив эту программу, вы увидите, что цвет изменяется в зависимости от того, находится курсор мыши в области `Ellipse` или нет. Ниже показан другой триггер, в котором при щелчке на элементе уменьшаются размеры контейнера `Grid` (и, следовательно, всех его дочерних элементов). Добавьте следующую разметку в коллекцию `<ControlTemplate.Triggers>`:

```

<Trigger Property = "IsPressed" Value="True">
  <Setter TargetName="controlLayout"
    Property="RenderTransformOrigin" Value="0.5,0.5"/>
  <Setter TargetName="controlLayout" Property="RenderTransform">
    <Setter.Value>
      <ScaleTransform ScaleX="0.8" ScaleY="0.8"/>
    </Setter.Value>
  </Setter>
</Trigger>

```

Роль расширения разметки {TemplateBinding}

Построенный шаблон применим только к элементам `Button`, и потому разумно ожидать, что должен существовать способ установки свойств элемента `<Button>` для уникального отображения шаблона. Например, сейчас в свойстве `Fill` элемента `Ellipse` жестко закодирован синий цвет, а свойство `Content` элемента `Label` всегда содержит строковое значение `OK`. Естественно, могут понадобиться кнопки других цветов и с другими текстовыми значениями, поэтому давайте попробуем определить в главном окне следующие кнопки:

```

<StackPanel>
  <Button x:Name="myButton" Width="100" Height="100"
    Background="Red" Content="Howdy!" Click="myButton_Click"
    Template="{StaticResource RoundButtonTemplate}" />
  <Button x:Name="myButton2" Width="100" Height="100"
    Background="LightGreen" Content="Cancel!"
    Template="{StaticResource RoundButtonTemplate}" />
  <Button x:Name="myButton3" Width="100" Height="100"
    Background="Yellow" Content="Format"
    Template="{StaticResource RoundButtonTemplate}" />
</StackPanel>

```

Однако, независимо от указания для каждого элемента `Button` уникальных значений в свойствах `Background` и `Content`, вы все равно получите три синих кнопки с текстом `OK`. Проблема в том, что свойства элемента управления, использующего шаблон (`Button`), не соответствуют в точности элементам шаблона (например, свойство `Fill` элемента `Ellipse`). Кроме того, хотя элемент `Label` имеет свойство `Content`, значение, определенное в контексте `<Button>`, не направляется автоматически во внутреннее содержимое шаблона.

При построении шаблона такие проблемы можно устранить с помощью расширения разметки `{TemplateBinding}`. Это позволит захватывать настройки свойств, определенные элементом управления, который использует шаблон, и применять их для установки свойств в самом шаблоне.

Ниже приведена переработанная версия `RoundButtonTemplate`, в которой теперь это расширение разметки используется для отображения свойства `Background` элемента `Button` на свойство `Fill` элемента `Ellipse`; также здесь обеспечивается передача значения `Content` элемента `Button` в свойство `Content` элемента `Label`.

```
<ControlTemplate x:Key="RoundButtonTemplate" TargetType="Button">
  <Grid x:Name="controlLayout">
    <Ellipse x:Name="buttonSurface" Fill="{TemplateBinding Background}"/>
    <Label x:Name="buttonCaption" Content="{TemplateBinding Content}"
      FontSize="20" FontWeight="Bold"
      HorizontalAlignment="Center" VerticalAlignment="Center" />
  </Grid>
  <ControlTemplate.Triggers>
    ...
  </ControlTemplate.Triggers>
</ControlTemplate>
```

После такого обновления можно создавать кнопки разных цветов и с разным текстом (рис. 31.12).

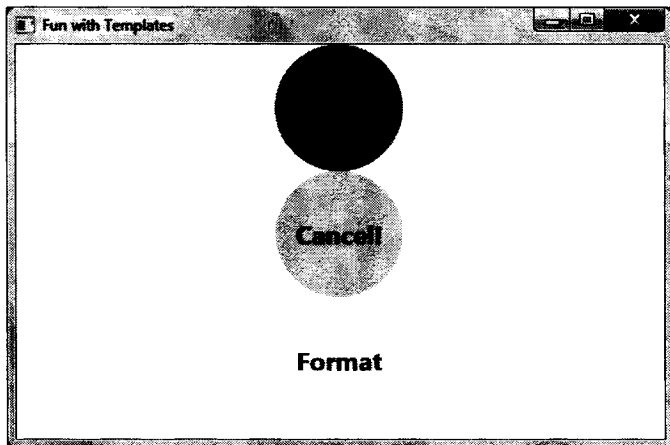


Рис. 31.12. Привязки шаблона позволяют передавать значения во внутренние элементы управления

Роль класса `ContentPresenter`

При проектировании шаблона для отображения текстового значения элемента управления использовался элемент `Label`. Подобно `Button`, он поддерживает свойство `Content`. Поэтому, если применяется расширение разметки `{TemplateBinding}`, можно определить элемент `Button` со сложным содержимым, а не с простой строкой. Например:

```
<Button x:Name="myButton4" Width="100" Height="100" Background="Yellow"
  Template="{StaticResource RoundButtonTemplate}">
  <Button.Content>
    <ListBox Height="50" Width="75">
      <ListBoxItem>Hello</ListBoxItem>
      <ListBoxItem>Hello</ListBoxItem>
      <ListBoxItem>Hello</ListBoxItem>
    </ListBox>
  </Button.Content>
</Button>
```

Для этого конкретного элемента управления все работает, как ожидалось. Но что, если необходимо передать сложное содержимое члену шаблона, который не имеет свойства Content? Когда требуется определить обобщенную область отображения содержимого в шаблоне, можно использовать не специфический тип элемента (Label или TextBox), а класс ContentPresenter. В рассматриваемом примере это делать не нужно, однако ниже показана простая разметка, иллюстрирующая построение специального шаблона, который применяет ContentPresenter для показа значения свойства Content элемента управления, использующего шаблон:

```
<!-- Этот шаблон кнопки отобразит то, что установлено
      в Content размещающей кнопки -->
<ControlTemplate x:Key="NewRoundButtonTemplate" TargetType="Button">
  <Grid>
    <Ellipse Fill="{TemplateBinding Background}" />
    <ContentPresenter HorizontalAlignment="Center"
                      VerticalAlignment="Center" />
  </Grid>
</ControlTemplate>
```

Включение шаблонов в стили

Сейчас шаблон просто определяет базовый внешний вид элемента управления Button. Тем не менее, за процесс установки базовых свойств элемента управления (содержимое, размер шрифта, вес шрифта и т.п.) отвечает сам элемент Button:

```
<!-- Сейчас Button сам должен устанавливать базовые значения свойств, а не шаблон -->
<Button x:Name="myButton" Foreground="Black" FontSize="20" FontWeight="Bold"
        Template="{StaticResource RoundButtonTemplate}" Click="myButton_Click"/>
```

При желании эти значения можно устанавливать в шаблоне. Подобным образом, по сути, создается стандартный внешний вид. И как вам, видимо, уже понятно, эта работа предназначена для стилей WPF. При создании стиля (для учета настроек базовых свойств) можно определить шаблон *внутри стиля*! Ниже приведен измененный ресурс приложения из файла App.xaml, который помечен ключом RoundButtonStyle:

```
<!-- Стиль, содержащий шаблон -->
<Style x:Key="RoundButtonStyle" TargetType="Button">
  <Setter Property="Foreground" Value="Black"/>
  <Setter Property="FontSize" Value="14"/>
  <Setter Property="FontWeight" Value="Bold"/>
  <Setter Property="Width" Value="100"/>
  <Setter Property="Height" Value="100"/>
  <!-- Далее следует сам шаблон -->
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="Button">
        <Grid x:Name="controlLayout">
          <Ellipse x:Name="buttonSurface" Fill="{TemplateBinding Background}" />
          <Label x:Name="buttonCaption" Content="{TemplateBinding Content}"
                HorizontalAlignment="Center" VerticalAlignment="Center" />
        </Grid>
        <ControlTemplate.Triggers>
          <Trigger Property="IsMouseOver" Value="True">
            <Setter TargetName="buttonSurface" Property="Fill" Value="Blue"/>
            <Setter TargetName="buttonCaption" Property="Foreground"
                  Value="Yellow"/>
          </Trigger>
        </ControlTemplate.Triggers>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

```

<Trigger Property = "IsPressed" Value="True">
  <Setter TargetName="controlLayout"
    Property="RenderTransformOrigin" Value="0.5,0.5"/>
  <Setter TargetName="controlLayout" Property="RenderTransform">
    <Setter.Value>
      <ScaleTransform ScaleX="0.8" ScaleY="0.8"/>
    </Setter.Value>
  </Setter>
</Trigger>
</ControlTemplate.Triggers>
</ControlTemplate>
</Setter.Value>
</Setter>
</Style>

```

После таких изменений можно создавать кнопочные элементы управления, устанавливая свойство `Style` следующим образом:

```

<Button x:Name="myButton" Background="Red" Content="Howdy!"
  Click="myButton_Click" Style="{StaticResource RoundButtonStyle}"/>

```

Хотя внешний вид и поведение кнопки остаются теми же самыми, внедрение шаблонов в стили удобнее тем, что оно позволяет предоставить готовый набор значений для общих свойств. На этом обзор использования Visual Studio и инфраструктуры триггеров для построения специальных шаблонов элемента управления завершен. Хотя относительно API-интерфейса Windows Presentation Foundation можно еще много чего сказать, теперь у вас есть хороший фундамент для самостоятельного изучения.

Исходный код. Проект `ButtonTemplate` доступен в подкаталоге `Chapter 31`.

Резюме

В этой главе рассматривалось множество связанных с WPF тем, которые были ориентированы на создание специальных пользовательских элементов управления. Мы начали с рассмотрения, как в WPF задействованы традиционные программные примитивы .NET — свойства и события. Вы узнали, что механизм *свойств зависимости* позволяет создавать свойства, интегрируемые в набор служб WPF (анимации, привязки данных, стили и т.п.). Связанное с ними понятие *маршрутизируемых событий* обеспечивает передачу событий вверх и вниз по дереву разметки.

Затем рассматривались отношения между *логическим* и *визуальным деревьями*. Логическое дерево, по сути, является взаимно однозначным отображением разметки, описывающей корневой элемент WPF. За этим логическим деревом находится гораздо более глубокое визуальное дерево, которое содержит детальные инструкции визуализации.

В главе также была рассмотрена роль *стандартного шаблона*. Запомните, что при построении специальных шаблонов, вы на самом деле заменяете визуальное дерево элемента управления (или его часть) собственной специальной реализацией.

ЧАСТЬ VIII

ASP.NET Web Forms

В этой части

Глава 32. Введение в ASP.NET Web Forms

Глава 33. Веб-элементы управления, мастер-страницы и темы ASP.NET

Глава 34. Управление состоянием в ASP.NET

Введение в ASP.NET Web Forms

Все рассмотренные до сих пор примеры были либо консольными, либо настольными приложениями с графическими пользовательскими интерфейсами, созданными с применением WPF. В последующих главах вы узнаете, как платформа .NET облегчает построение браузерных уровней представления с использованием технологии под названием ASP.NET. Для начала будет дан краткий обзор основных концепций веб-разработки (HTTP, HTML, написание сценариев клиентской стороны, обратные отправки) и описана роль коммерческого веб-сервера Microsoft (IIS), а также веб-сервера разработки ASP.NET (ASP.NET Development Web Server).

После краткого введения остальная часть главы будет посвящена структуре модели программирования ASP.NET Web Forms (включая однофайловую модель и модель отделенного кода) и обзору функциональности базового класса Page. Попутно вы ознакомитесь с ролью веб-элементов управления ASP.NET, структурой каталогов веб-сайта ASP.NET и применением файла `web.config` для управления веб-сайтами во время выполнения.

На заметку! Чтобы загрузить любой из проектов веб-сайтов ASP.NET, входящих в состав кода примеров для этой книги, запустите Visual Studio и выберите пункт меню `File⇒Open⇒Web Site...` (`Файл⇒Открыть⇒Веб-сайт...`). В открывшемся диалоговом окне щелкните на кнопке `File System` (Файловая система) в левой части окна и выберите папку, содержащую файлы веб-проекта. Содержимое текущего веб-приложения загрузится в IDE-среде Visual Studio.

Роль протокола HTTP

Веб-приложения значительно отличаются от настольных графических приложений. Первое очевидное отличие состоит в том, что профессиональное веб-приложение всегда подразумевает существование как минимум двух машин, объединенных в сеть: на одной находится веб-сайт, а на другой просматриваются данные с помощью веб-браузера. Разумеется, во время разработки вполне допускается, что один компьютер будет исполнять роль и браузерного клиента, и веб-сервера, обслуживающего содержимое сайта. Учитывая природу веб-приложений, объединенные в сеть машины должны договориться об используемом сетевом протоколе для определения способа отправки и получения данных. Сетевой протокол, соединяющий такие компьютеры, называется протоколом передачи гипертекста (Hypertext Transfer Protocol — HTTP).

Цикл запрос/ответ HTTP

Когда на клиентской машине запускается веб-браузер (такой как Google Chrome, Opera, Mozilla Firefox, Apple Safari или Microsoft Internet Explorer), выполняется *HTTP-запрос* для доступа к определенному ресурсу (обычно веб-странице) на удаленной серверной машине. HTTP представляет собой текстовый протокол на основе стандартной парадигмы запрос/ответ. Например, если ввести в адресной строке `http://www.facebook.com`, то программа-браузер задействует веб-технологии под названием *служба доменных имен* (Domain Name Service — DNS), которая преобразует запрошенный URL в числовое значение, которое называется *IP-адресом*. После этого браузер открывает сокетное подключение (обычно через порт 80 для незащищенных подключений) и посылает HTTP-запрос для обработки на целевом сайте.

Веб-сервер принимает входящий HTTP-запрос и может обработать любые отправленные клиентом входные значения (значения текстовых полей, флажков, переключателей и т.п.), чтобы сформировать нужный *HTTP-ответ*. Веб-программисты могут использовать любое количество серверных технологий (PHP, ASP.NET, JSP и т.д.) для динамической генерации содержимого, посылаемого в HTTP-ответе. После этого браузер клиентской стороны отображает на экране HTML-разметку, отправленную веб-сервером. На рис. 32.1 показан базовый цикл запроса/ответа HTTP.

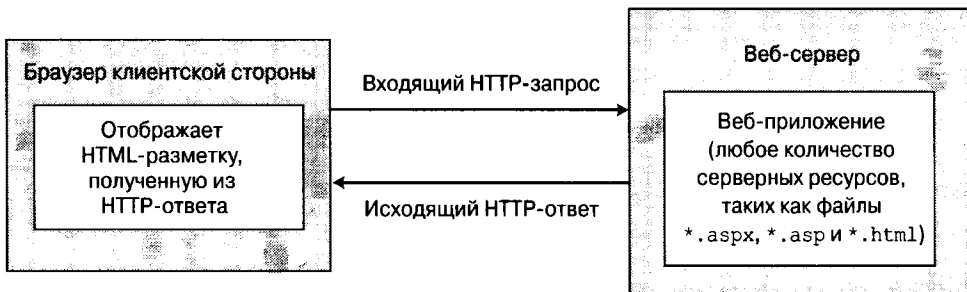


Рис. 32.1. Цикл запроса/ответа HTTP

HTTP — протокол без хранения состояния

Другой аспект веб-разработки, который заметно отличает ее от программирования традиционных настольных приложений, состоит в том, что HTTP — это, по сути, сетевой протокол без хранения состояния. После отправки с веб-сервера ответа клиентскому браузеру все их предыдущее взаимодействие забывается. В традиционном настольном приложении все совершенно не так: состояние исполняемой программы сохраняется и используется, пока пользователь не закроет главное окно приложения.

Учитывая это, на веб-разработчика возлагается ответственность за реализацию запоминания информации (такой как товары в корзине покупок, номера кредитных карт, домашние адреса), связанной с пользователями, которые в данный момент зашли на сайт. Как будет показано в главе 34, ASP.NET предлагает многочисленные способы поддержки состояния: переменные сеанса, cookie-наборы и кеш приложения, а также API-интерфейс управления профилями ASP.NET.

Веб-приложения и веб-серверы

Веб-приложение можно воспринимать как коллекцию файлов (*.htm, *.aspx, файлов изображений, файлов данных XML и т.п.), а также связанных с ними компонентов (таких как библиотека кода .NET), которые хранятся в определенном наборе каталогов

на веб-сервере. Как будет показано в главе 34, веб-приложения ASP.NET обладают специфическим жизненным циклом и предоставляют многочисленные события (вроде начального запуска или финального останова), которые можно перехватывать для выполнения специализированной обработки во время оперирования веб-сайта.

Веб-сервер — это программный продукт, отвечающий за размещение веб-приложений. Он обычно предоставляет множество взаимосвязанных служб, таких как интегрированная безопасность, FTP (File Transfer Protocol — протокол передачи файлов), обмен почтой и т.д. Службы Internet Information Services (IIS) — это веб-серверный продукт производственного уровня от Microsoft, который обладает встроенной поддержкой веб-приложений ASP.NET.

При наличии правильно установленного сервера IIS, взаимодействовать с ним можно через папку Administrative Tools (Администрирование) панели управления, дважды щелкнув на значке Internet Information Services Manager (Диспетчер служб IIS).

На рис. 32.2 показан узел Default Web Site (Веб-сайт по умолчанию) IIS, в котором находится основная часть параметров конфигурации (в предшествующих версиях IIS диспетчер выглядел по-другому).

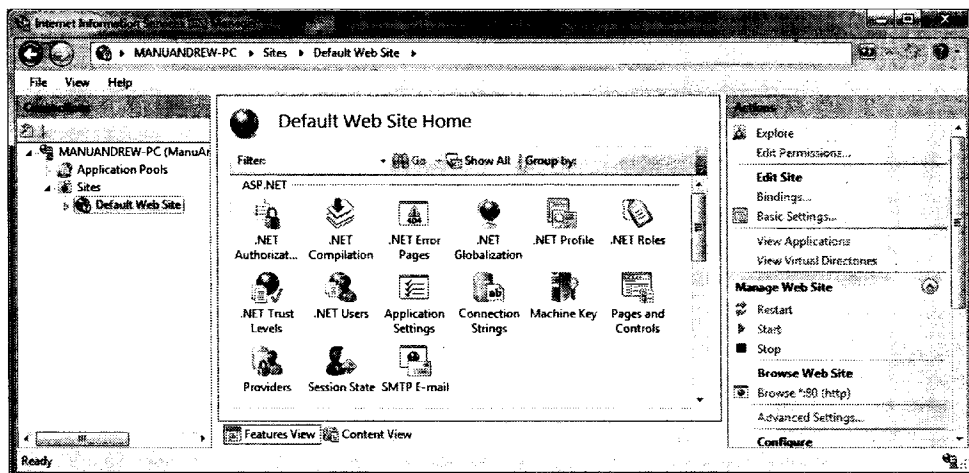


Рис. 32.2. Диспетчер служб IIS позволяет сконфигурировать поведение Microsoft IIS во время выполнения

Роль виртуальных каталогов IIS

Единственная установленная копия IIS может размещать многочисленные веб-приложения, каждое из которых располагается в своем *виртуальном каталоге*. Каждый виртуальный каталог отображается на физический каталог на жестком диске. Например, если создается новый виртуальный каталог с именем CarsAreUs, то извне он доступен по адресу вроде <http://www.MyDomain.com/CarsAreUs> (предполагается, что IP-адрес сайта зарегистрирован в DNS как www.MyDomain.com). На веб-сервере этот виртуальный каталог отображается на физический корневой каталог, где находится содержимое веб-приложения CarsAreUs.

Как будет показано далее в главе, при создании веб-приложения ASP.NET с помощью Visual Studio можно указать, чтобы IDE-среда автоматически сгенерировала новый виртуальный каталог для текущего веб-сайта. Но при необходимости можно создать виртуальный каталог и вручную, щелкнув правой кнопкой мыши на узле Default Web Site в диспетчере служб IIS и выбрав в контекстном меню пункт Add Virtual Directory (Добавить виртуальный каталог).

Веб-сервер разработки ASP.NET

В предшествующих версиях платформы .NET разработчикам ASP.NET приходилось создавать виртуальные каталоги IIS во время построения и тестирования веб-приложений. Во многих случаях такая тесная зависимость от IIS излишне усложняла командную разработку, не говоря уже о том, что многие сетевые администраторы были не в восторге от необходимости выполнять установку IIS на машине каждого разработчика.

К счастью, теперь появилась возможность пользоваться легковесным веб-сервером под названием ASP.NET Development Web Server (Веб-сервер разработки ASP.NET). Эта утилита позволяет разработчикам размещать веб-приложения ASP.NET за рамками IIS и дает возможность строить и тестировать веб-страницы из любого каталога на машине. Это довольно удобно в сценариях командной разработки, а также при построении веб-приложений ASP.NET под управлением версий Windows, которые не поддерживают установку IIS.

В большинстве примеров настоящей книги применяется веб-сервер разработки ASP.NET (через соответствующий вариант проекта Visual Studio) вместо размещения веб-содержимого в виртуальном каталоге IIS. Хотя данный подход может упростить разработку веб-приложений, имейте в виду, что этот веб-сервер не рассчитан на размещение веб-приложений производственного уровня. Он предназначен только для целей разработки и тестирования. Как только веб-приложение готово к эксплуатации, сайт следует скопировать в виртуальный каталог IIS.

На заметку! В Visual Studio имеется встроенный инструмент для копирования локального веб-приложения на производственный веб-сервер. Для этого достаточно буквально пары щелчков на кнопках. Для запуска процесса необходимо выбрать веб-проект в проводнике решений Visual Studio, а затем щелкнуть на кнопке Copy Web Site (Скопировать веб-сайт). В этот момент можно выбрать место для развертывания проекта.

Роль языка HTML

Теперь, когда настроен каталог для размещения веб-приложения и выбран веб-сервер для использования в качестве хоста, необходимо создать само содержимое. Вспомните, что веб-приложение — это просто набор файлов, составляющих функциональность сайта. Многие из этих файлов будут содержать операторы HTML (Hypertext Markup Language — язык разметки гипертекста). HTML является стандартным языком разметки для описания того, как литеральный текст, изображения, внешние ссылки и разнообразные элементы управления HTML отображаются в окне браузера на стороне клиента.

Хотя современные IDE-среды (включая Visual Studio) и платформы веб-разработки (такие как ASP.NET) генерируют значительную часть HTML-разметки автоматически, при работе с ASP.NET без знания языка HTML все же не обойтись.

На заметку! Вспомните из главы 2, что Microsoft предлагает ряд бесплатных IDE-сред из семейства продуктов Express (например, Visual C# Express). Если вас интересует веб-разработка, возможно, имеет смысл загрузить Visual Web Developer Express. Эта бесплатная IDE-среда специально ориентирована на построение веб-приложений ASP.NET с использованием C# или VB.

В этом разделе рассматриваются основы HTML, которые необходимы для понимания разметки, генерируемой моделью программирования ASP.NET Web Forms.

Структура HTML-документа

Типичный файл HTML состоит из набора дескрипторов, описывающих внешний вид и поведение веб-страницы. Базовая структура HTML-документа, как правило, не изменяется. Например, файлы *.htm (или *.html) открываются и закрываются дескрипторами `<html>` и `</html>`, в них обычно определен раздел `<body>` и т.д.

Чтобы проиллюстрировать некоторые основы HTML, откройте Visual Studio и создайте файл пустой страницы HTML, используя пункт меню `File⇒New⇒File` (Файл⇒Создать⇒Файл); обратите внимание, что веб-проект пока не создается, а просто открывается пустой файл HTML для редактирования. После этого сохраните файл под именем `default.htm` в удобном месте на жестком диске. В нем будет находиться следующая начальная разметка (точный текст может отличаться в зависимости от конфигурации Visual Studio):

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta charset="utf-8" />
  <title>Untitled Page</title>
</head>
<body>

</body>
</html>
```

Прежде всего, обратите внимание на инструкцию обработки DOCTYPE в начале файла. Она вместе с открывающим дескриптором `<html>` указывает, что содержащиеся в файле дескрипторы HTML должны соответствовать стандарту HTML 5.0. Как упоминалось ранее, традиционный HTML был очень либерален в отношении синтаксиса. Помимо нечувствительности к регистру символов, допускалось определять открывающий элемент (такой как `<br>` для перевода строки), который не имел соответствующего закрывающего элемента (`</br>` в данном случае). Стандарт HTML 5.0 представляет собой спецификацию W3C, которая добавляет в базовый язык разметки множество новых возможностей.

На заметку! По умолчанию Visual Studio проверяет все документы HTML на соответствие схеме HTML 5.0, чтобы обеспечить согласованность разметки со стандартом HTML 5. Если нужно указать альтернативную схему проверки достоверности, откройте диалоговое окно Options (Параметры), выбрав пункт меню `Tools⇒Options` (Сервис⇒Параметры), разверните узел Text Editor (Текстовый редактор), затем узел HTML и выберите узел Validation (Проверка достоверности). Если вы не хотите просматривать предупреждения, просто снимите отметку с расположенного там же флажка Show Errors (Показывать ошибки).

Область `<body>` — место, в котором определена основная часть действительно содержимого. Чтобы немного приукрасить начальную страницу, добавим к ней заголовок:

```
<head>
  <title>This is my simple web page</title>
</head>
```

Дескрипторы `<title>` применяются для указания текстовой строки, которая должна быть помещена в заголовке окна веб-браузера.

Роль форм HTML

Форма HTML — это просто именованная группа взаимосвязанных элементов пользовательского интерфейса, обычно служащих для ввода пользователем каких-нибудь данных. Не путайте HTML-форму с общей областью отображения браузера. В действительности форма HTML — это скорее *логическая группа* графических элементов управления, помещенных между дескрипторами `<form>` и `</form>`, например:

```
<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>This is my simple web page</title>
</head>
<body>
  <form id="defaultPage">
    <!-- Вставьте сюда содержимое пользовательского веб-интерфейса -->
  </form>
</body>
</html>
```

Этой форме назначен идентификатор "defaultPage". Обычно открывающий дескриптор `<form>` содержит атрибут `action`, в котором указан URL-адрес, куда отправляются данные формы, а также атрибут `method`, определяющий метод передачи данных (POST или GET). Давайте ознакомимся с разновидностями элементов, помещаемых в формы HTML (кроме обычного текста).

Инструменты визуального конструктора HTML в Visual Studio

В панели инструментов (Toolbox) среды Visual Studio имеется вкладка HTML, которая позволяет выбрать любой элемент управления HTML и поместить его на поверхность визуального конструктора HTML (рис. 32.3).

На заметку! При построении веб-страниц ASP.NET с помощью модели программирования веб-форм эти элементы управления HTML обычно не применяются для создания пользовательского интерфейса. Вместо них используются веб-элементы управления ASP.NET, которые преобразуются в корректную HTML-разметку без вашего участия. Роль веб-элементов управления рассматривается далее в этой главе.

Подобно процессу разработки WPF-приложения, эти элементы управления HTML могут перетаскиваться на поверхность визуального конструктора HTML или непосредственно в HTML-разметку. Если щелкнуть на кнопке Split (Разделить) внизу окна редактора HTML, то нижняя панель будет отображать визуальную компоновку HTML, а верхняя — соответствующий код разметки. Другое преимущество этого редактора состоит в том, что при выборе разметки или HTML-элемента пользовательского интерфейса соответствующее представление подсвечивается (рис. 32.4).

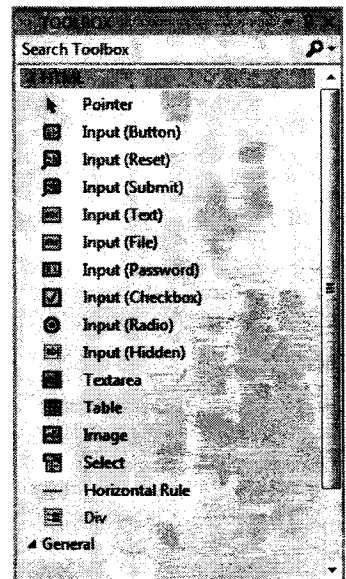


Рис. 32.3. Вкладка HTML панели инструментов

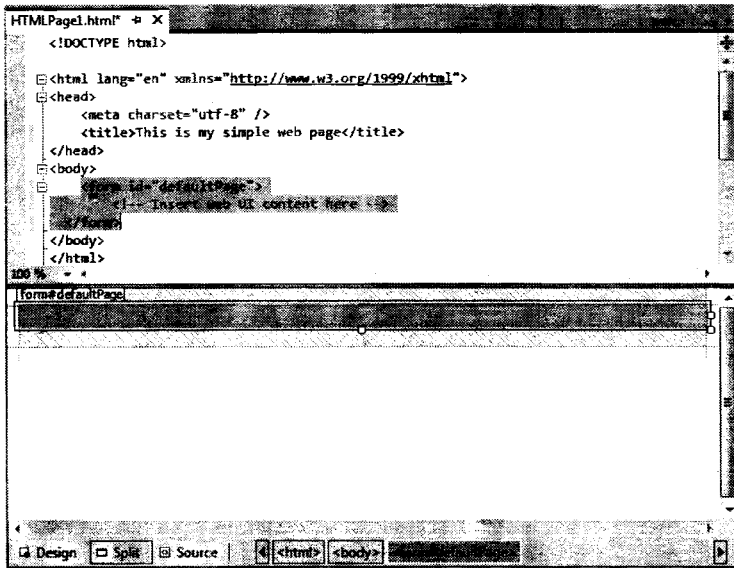


Рис. 32.4. Визуальный конструктор HTML в Visual Studio

Среда Visual Studio также позволяет редактировать внешний вид и поведение файла *.htm или отдельного элемента управления в <form> с использованием окна Properties (Свойства). Например, выбрав в раскрывающемся списке окна Properties элемент DOCUMENT, можно конфигурировать различные аспекты HTML-страницы (рис. 32.5).

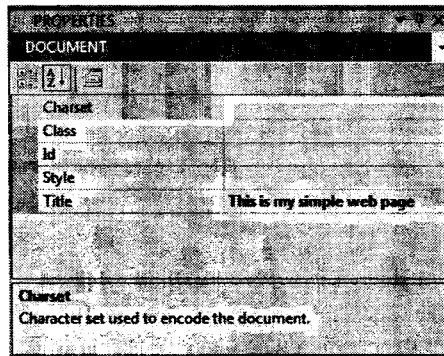


Рис. 32.5. Окно Properties в Visual Studio может применяться для конфигурирования HTML-разметки

При использовании окна Properties для конфигурирования какого-то аспекта веб-страницы IDE-среда соответствующим образом изменяет HTML-разметку. При чтении следующих глав тоже можете применять IDE-среду для редактирования HTML-страниц.

Построение HTML-формы

Теперь добавьте в область <body> первоначального файла литеральный текст, который приглашает пользователя ввести какое-нибудь сообщение. Такой текст можно вводить и форматировать непосредственно в визуальном конструкторе HTML. Здесь используется дескриптор <h1> для установки веса заголовка, <p> — для блока абзаца и <i> — для выделения текста курсивом:

```

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>This is my simple web page</title>
</head>
<body>
  <!-- Приглашение на ввод текста. -->
  <h1>Simple HTML Page</h1>
  <p>
    <br/>
    <i>Please enter a message</i>.
  </p>
  <form id="defaultPage">
  </form>
</body>
</html>

```

А теперь создадим область ввода формы. В общем случае каждый элемент управления HTML описывается с помощью атрибутов `id` (для идентификации элемента в коде) и `type` (для указания того, какой конкретно элемент управления вводом должен быть помещен в объявление `<form>`). В зависимости от того, какой виджет пользовательского интерфейса объявлен, допускается применение дополнительных специфических атрибутов, которые можно модифицировать в окне **Properties**.

Создаваемый нами пользовательский интерфейс будет содержать одно текстовое поле и две кнопки. Первая кнопка будет служить для запуска сценария на стороне клиента, а другая — для сброса входных полей формы в стандартные значения. Измените HTML-форму следующим образом:

```

<!-- Построить форму для получения пользовательской информации. -->
<form id="defaultPage">
  <p>
    Your Message:
    <input id="txtUserMessage" type="text"/></p>
  <p>
    <input id="btnShow" type="button" value="Show!"/>
    <input id="btnReset" type="reset" value="Reset"/>
  </p>
</form>

```

Обратите внимание, что каждому элементу управления в атрибуте `id` назначен подходящий идентификатор (`txtUserMessage`, `btnShow` и `btnReset`). Кроме того, каждый элемент ввода имеет дополнительный атрибут `type`, который указывает, что элемент автоматически сбрасывает все поля в их начальные значения (`type="reset"`), принимает текстовый ввод (`type="text"`) или функционирует как простая кнопка клиентской стороны, ничего не отправляющая веб-серверу (`type="button"`).

Сохраните файл, щелкните правой кнопкой мыши на поверхности визуального конструктора и выберите в контекстном меню пункт **View in Browser** (Просмотреть в браузере). На рис. 32.6 показана текущая страница в браузере Firefox.

На заметку! В случае выбора пункта **View in Browser** для HTML-файла Visual Studio автоматически запускает веб-сервер разработки ASP.NET для размещения содержимого.

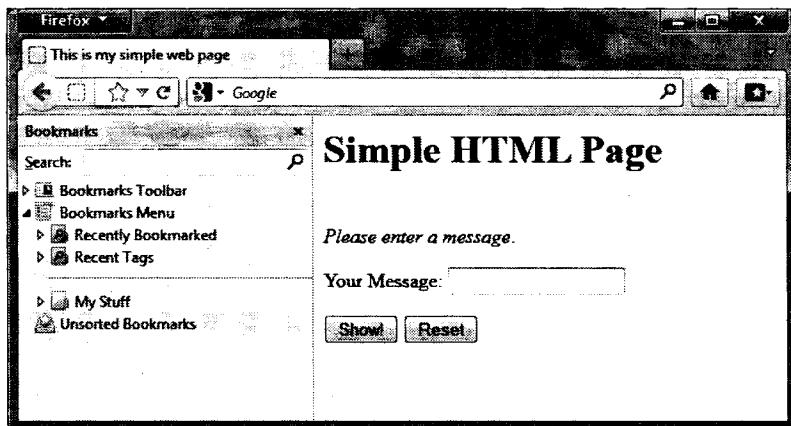


Рис. 32.6. Простая HTML-страница

Роль сценариев клиентской стороны

В дополнение к элементам графического пользовательского интерфейса, файл *.html может содержать блоки кода сценариев, которые будут обработаны запрашивающим браузером. Есть две главные причины использования сценариев на стороне клиента:

- проверка достоверности пользовательского ввода перед обратной отправкой веб-серверу;
- взаимодействие с объектной моделью документа (Document Object Model — DOM) браузера.

Относительно первой причины следует понимать, что при работе веб-приложений приходится часто выполнять процедуру полного обмена (которая называется *обратной отправкой*) с сервером для обновления HTML-разметки, визуализируемой в браузере. Хотя полностью избежать обратных отправок не удастся, нужно стремиться к минимизации трафика в сети. Один из приемов, позволяющих сэкономить количество обратных отправок, состоит в применении сценариев клиентской стороны для проверки достоверности пользовательского ввода перед отправкой данных формы веб-серверу. При обнаружении ошибки, такой как отсутствие данных в обязательном поле, можно уведомить пользователя об этом, не выполняя обратную отправку веб-серверу. (В конце концов, нет ничего более раздражающего пользователей, нежели отправка данных по медленному подключению лишь для того, чтобы получить в ответ сообщения об ошибках ввода!)

На заметку! Имейте в виду, что даже при выполнении проверки достоверности на стороне клиента (для сокращения времени реакции), аналогичная проверка также должна быть предпринята на стороне самого веб-сервера. Это гарантирует, что данные не были искажены после того, как клиент отправил их по сети. Элементы управления проверкой достоверности ASP.NET автоматически производят проверку достоверности на клиентской и серверной стороне (более подробно об этом пойдет речь в главе 33).

Сценарии клиентской стороны могут также использоваться для взаимодействия с лежащей в основе объектной моделью (DOM) самого браузера. Большинство коммерческих браузеров открывают доступ к набору объектов, которые позволяют управлять поведением браузера.

Когда браузер производит синтаксический анализ HTML-страницы, он строит в памяти дерево объектов, представляющих все содержимое веб-страницы (формы, элементы ввода и т.д.). Браузеры предоставляют API-интерфейс под названием DOM, который открывает доступ к дереву объектов и позволяет программно изменять его содержимое. Например, можно написать код JavaScript, который выполняется в браузере для получения значений из определенных элементов управления, изменения цвета элемента, динамического добавления новых элементов на страницу и т.п.

К большому сожалению, разные браузеры часто предлагают сходные, но не идентичные объектные модели. Поэтому блок сценария клиентской стороны, который взаимодействует с DOM, может работать неодинаково в разных браузерах (и потому тестирование обязательно).

В ASP.NET имеется свойство `HttpRequest.Browser`, которое позволяет определять во время выполнения возможности браузера и устройства, откуда поступил текущий запрос. Данная информация позволяет максимально оптимизировать формируемый HTTP-ответ. Но это редко бывает нужно, если только вы не реализуете специальные элементы управления, поскольку все стандартные веб-элементы управления в ASP.NET автоматически знают, как правильно визуализировать себя в различных браузерах. Эта ценная способность известна как *адаптивная визуализация*, и она реализована во всех стандартных элементах управления ASP.NET.

Для написания сценариев клиентской стороны доступны разнообразные языки программирования, но наиболее популярным из них является JavaScript. Важно помнить, что JavaScript ни в каком аспекте не является подмножеством языка Java. Хотя JavaScript и Java обладают местами похожим синтаксисом, язык JavaScript намного менее мощный, чем Java. Вдобавок все современные веб-браузеры поддерживают язык JavaScript, что делает его естественным кандидатом на реализацию логики сценариев клиентской стороны.

Пример сценария клиентской стороны

Чтобы продемонстрировать роль сценариев клиентской стороны, давайте сначала рассмотрим, как перехватываются события, посылаемые виджетами графического пользовательского интерфейса клиентской стороны. Для перехвата события `click` кнопки **Show! (Показать)** добавьте в определение виджета `btnShow` атрибут `onclick` и укажите в нем JavaScript-метод по имени `btnShow_onclick()`:

```
<input id="btnShow" type="button" value="Show!"
      onclick="return btnShow_onclick()" />
```

Теперь добавьте сразу после открывающего элемента `<head>` код функции JavaScript, которая будет вызвана при щелчке пользователя на кнопке. Используйте метод `alert()` для отображения на стороне клиента окна сообщения, которое содержит в свойстве `value` значение из текстового поля:

```
<script lang="javascript" type="text/javascript">
// 
    function btnShow_onclick() {
        alert(txtUserMessage.value);
    }
// ]]&gt;
&lt;/script&gt;</pre>
</div>
<div data-bbox="50 873 956 943" data-label="Text">
<p>Обратите внимание, что блок сценария помещен в раздел <code>CDATA</code>. Причина этого проста: если страница попадет в браузер, который не поддерживает JavaScript, этот код будет воспринят как блок комментария и проигнорирован. Разумеется, страница при этом станет менее функциональной, но зато не вызовет ошибку при визуализации в</p>
</div>
```

таком браузере. В любом случае, при последующем просмотре страницы в браузере вы сможете ввести сообщение и увидеть его в окне сообщения (рис. 32.7).

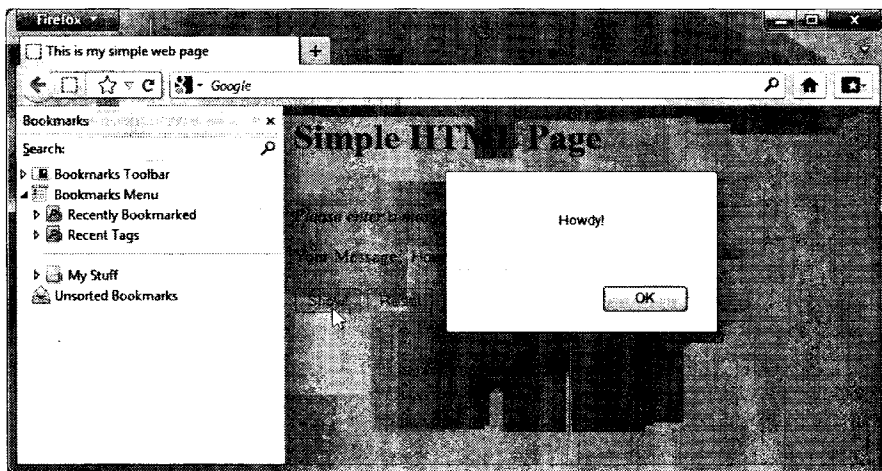


Рис. 32.7. Вызов функции JavaScript на стороне клиента

Щелчок на кнопке Reset (Сброс) приводит к очистке текстового поля, поскольку эта кнопка была определена с атрибутом `type="reset"`.

Обратная отправка веб-серверу

Эта простая HTML-страница выполняет всю свою функциональность в принимающем браузере. Однако *реальной* веб-странице необходима обратная отправка ресурсу на веб-сервере, с одновременной передачей всех введенных данных. Получив эти данные, ресурс серверной стороны может использовать их для динамического построения соответствующего HTTP-ответа.

Атрибут `action` в открывающем дескрипторе `<form>` указывает получателя входных данных формы. Такими получателями могут быть почтовые серверы, другие HTML-файлы на веб-сервере, классические файлы Active Server Pages (ASP), веб-страницы ASP.NET и т.д.

Помимо атрибута `action`, должна быть также кнопка отправки (`submit`), щелчок на которой приводит к передаче данных формы веб-приложению через HTTP-запрос. В текущем примере это не требуется, тем не менее, ниже приведен вариант разметки, где в открывающем дескрипторе `<form>` указан следующий атрибут:

```
<form id="defaultPage"
  action="http://localhost/Cars/ClassicAspPage.asp" method="GET">
  <input id="btnPostBack" type="submit" value="Post to Server!"/>
  ...
</form>
```

После щелчка на кнопке отправки данные формы посылаются файлу `ClassicAspPage.asp` по указанному URL. Если в качестве режима передачи задано `method="GET"`, данные формы присоединяются к строке запроса в виде набора пар "имя/значение", разделенных амперсандами. Наверняка вы уже видели данные подобного рода в браузере. Например:

```
http://www.google.com/search?hl=en&source=hp&q=vikings&cts=1264370773666&aq=
f&aql=&aqi=glg-zlglg-zlglg-zlg4&oq=
```

Другой метод передачи данных формы на веб-сервер определяется как `method="POST"`:

```
<form id="defaultPage"
  action="http://localhost/Cars/ClassicAspPage.asp" method="POST">
  ...
</form>
```

В этом случае данные формы не добавляются к строке запроса. При использовании метода POST данные формы не являются непосредственно видимыми внешнему миру. Что более важно, данные POST не имеют ограничения по количеству символов, в то время как многие браузеры ограничивают длину запросов GET.

Обратные отправки в ASP.NET

При создании веб-сайтов на основе ASP.NET Web Forms инфраструктура самостоятельно заботится о механизме отправки данных. Одним из многих преимуществ построения веб-сайтов с использованием ASP.NET является то, что модель программирования находится поверх стандартного управляемого событиями HTTP-протокола запрос/ответ. Поэтому вместо ручной установки атрибута `action` и определения HTML-кнопки отправки можно просто обрабатывать события веб-элементов управления ASP.NET с помощью стандартного синтаксиса C#.

Применяя эту управляемую событиями модель, можно очень легко осуществлять обратную отставку веб-серверу с помощью множества элементов управления. При необходимости это можно сделать по щелчку пользователя на переключателе, элементе в окне списка, дне в элементе управления типа календаря и т.д. В каждом случае вы просто обрабатываете соответствующее событие, а исполняющая среда ASP.NET автоматически выдает корректные HTML-данные отправки.

Исходный код. Веб-сайт SimpleWebPage доступен в подкаталоге Chapter 32.

Обзор API-интерфейса ASP.NET

К этому моменту краткий обзор разработки классических веб-приложений завершен, и вы готовы к изучению непосредственно ASP.NET. Естественно, каждая новая версия платформы .NET добавляла дополнительную функциональность к API-интерфейсу веб-программирования, и эта традиция продолжена в .NET 4.5. Однако независимо от версии .NET, с которой вы намерены работать, перечисленные ниже возможности доступны всем веб-приложениям ASP.NET.

1. Инфраструктура ASP.NET предоставляет модель *отделенного кода*, которая позволяет разделять логику презентации (HTML) и бизнес-логику (код C#).
2. Страницы ASP.NET кодируются на языках программирования .NET, а не на языках сценариев серверной стороны. Файлы кода компилируются в .NET-сборки *.dll (которые, в свою очередь, транслируются в намного более быстродействующий исполняемый код).
3. Веб-элементы управления ASP.NET могут применяться для построения пользовательского веб-интерфейса с использованием модели, подобной той, что присуща настольным оконным приложениям.
4. Веб-приложения ASP.NET могут пользоваться любыми сборками из библиотек базовых классов .NET и конструируются с применением объектно-ориентированных технологий, рассматриваемых в настоящей книге (классы, интерфейсы, структуры, перечисления и делегаты).

- Веб-приложения ASP.NET можно легко настраивать с помощью конфигурационного файла веб-приложения (`web.config`).

Первое, что здесь следует подчеркнуть: пользовательский интерфейс веб-страницы ASP.NET может быть сконструирован с применением разнообразных *веб-элементов управления*. В отличие от типичного HTML-элемента, веб-элементы управления выполняются на веб-сервере и выдают в HTTP-ответ корректные HTML-дескрипторы. Одно лишь это является огромным преимуществом ASP.NET, потому что значительно сокращает объем HTML-разметки, который необходимо писать вручную. В качестве небольшого примера предположим, что на веб-странице ASP.NET определен следующий веб-элемент управления ASP.NET:

```
<asp:Button ID="btnMyButton" runat="server" Text="Button" BorderColor="Blue"
    BorderStyle="Solid" BorderWidth="5px" />
```

С деталями объявления веб-элементов управления ASP.NET вы ознакомитесь чуть позже, а пока обратите внимание, что многие атрибуты элемента `<asp:Button>` очень похожи на свойства, знакомые вам по примерам WPF. То же самое верно для всех веб-элементов управления ASP.NET: когда в Microsoft создавали инструментальный набор веб-элементов управления, эти виджеты были намеренно спроектированы похожими на свои настольные аналоги.

Если теперь браузер обратится к `.aspx`-файлу, содержащему этот элемент управления, то элемент выдаст в выходной поток следующее объявление HTML:

```
<input type="submit" name="btnMyButton" value="Button" id="btnMyButton"
    style="border-color:Blue;border-width:5px;border-style:Solid;" />
```

Обратите внимание, что веб-элемент управления выдает стандартную HTML-разметку, которая может визуализироваться в любом браузере. Это значит, что использование веб-элементов управления ASP.NET ни в коем случае не привязывает к семейству операционных систем Microsoft или к браузеру Microsoft Internet Explorer. Веб-страницу ASP.NET можно просматривать в любой операционной системе или браузере (включая портативные устройства наподобие Apple iPhone или BlackBerry).

Выше было сказано, что веб-приложение ASP.NET компилируется в сборку `.NET`. Поэтому веб-проекты ничем не отличаются от любой `.dll`-сборки `.NET`, которая строилась в примерах, приведенных в этой книге. Скомпилированное веб-приложение состоит из кода CIL, манифеста сборки и метаданных типов. Это дает ряд значительных преимуществ, наиболее важными из которых являются выигрыш в производительности, строгая типизация и возможность микроуправления со стороны CLR (сборка мусора и т.п.).

Наконец, веб-приложения ASP.NET поддерживают программную модель, в которой разметку страницы можно отделять от кодовой базы C#, используя *файлы кода*. Такие файлы позволяют отобразить разметку на полноценную объектную модель, которая объединяется с файлом кода C# через объявления частичных классов.

Основные функциональные возможности ASP.NET 2.0 и последующих версий

Версия ASP.NET 1.0 была значительным шагом в верном направлении, а ASP.NET 2.0 предоставила множество дополнительных средств, которые позволили создавать не только динамические *веб-страницы*, но и полноценные *веб-сайты*. Ниже приведен список основных средств.

- Появление веб-сервера разработки ASP.NET (это значит, что разработчикам больше не нужно устанавливать IIS на своих машинах).

- Большое количество новых веб-элементов, обрабатывающих множество сложных ситуаций (элементы управления навигацией, элементы управления безопасностью, новые элементы управления привязкой данных и т.д.).
- Появление мастер-страниц, которые позволяют разработчикам присоединять общие области пользовательского интерфейса к набору взаимосвязанных страниц.
- Поддержка тем, которые предлагают декларативный способ изменения внешнего вида и поведения всего веб-приложения на веб-сервере.
- Поддержка веб-частей (Web Parts), которые позволяют конечным пользователям настраивать внешний вид и поведение веб-страницы и сохранять эти настройки для последующего использования (подобно порталам).
- Появление веб-ориентированной утилиты конфигурации и управления, которая обслуживает разнообразные файлы `web.config`.

Помимо веб-сервера разработки ASP.NET, одним из самых значительных новшеств ASP.NET 2.0 было появление *мастер-страниц*. Как известно, большинство веб-сайтов поддерживают согласованный внешний вид и поведение для всех страниц сайта. Взгляните на коммерческий веб-сайт вроде `www.amazon.com`. Каждая страница содержит одни и те же элементы, такие как общий заголовок, общий нижний колонтитул, общее меню навигации и т.д.

С помощью мастер-страниц можно моделировать эту общую функциональность и определять *места заполнения* для подключения других `.aspx`-файлов. Это существенно облегчает изменение общего вида сайта (изменение позиции панели навигации, смена логотипа в заголовке и т.п.) простым изменением мастер-страницы, не затрагивая другие `.aspx`-файлы.

На заметку! Мастер-страницы настолько удобны, что, начиная с версии Visual Studio 2010, все новые веб-проекты ASP.NET включают их по умолчанию.

В ASP.NET 2.0 было также добавлено множество новых веб-элементов управления, включая элементы управления, которые автоматически включают общие средства безопасности (элементы управления входом на сайт, элементы восстановления пароля и т.д.); элементы управления, которые позволяют уложить навигационную структуру поверх набора связанных `.aspx`-файлов; и дополнительные элементы управления для выполнения сложных операций привязки данных, в которых необходимые SQL-запросы могут генерироваться с помощью набора веб-элементов управления ASP.NET.

Основные функциональные возможности ASP.NET 3.5 (и .NET 3.5 SP1) и последующих версий

В платформе .NET 3.5 веб-приложения ASP.NET получили возможность использовать модель программирования LINQ (также появившуюся в .NET 3.5) и следующие веб-ориентированные средства.

- Поддержка привязки данных для сущностных классов ADO.NET (см. главу 23).
- Поддержка ASP.NET Dynamic Data. Это инфраструктура, подобная Ruby on Rails, которая может использоваться для построения веб-приложений, управляемых данными. Таблицы в базе данных представляются за счет их кодирования в виде URI веб-службы ASP.NET, а данные в таблицах автоматически визуализируются в HTML-разметку.
- Интегрированная поддержка разработки в стиле Ajax, которая, по сути, позволяет выполнять обратные микро-отправки для максимально быстрого обновления части веб-страницы.

Шаблоны проектов ASP.NET Dynamic Data, которые появились в .NET 3.5 Service Pack 1, предлагают новую модель создания сайтов, существенно связанных с реляционной базой данных. Конечно, большинство веб-сайтов в определенной мере нуждаются во взаимодействии с базами данных, но проекты ASP.NET Dynamic Data тесно связаны с ADO.NET Entity Framework и в основном ориентированы на быструю разработку сайтов, управляемых данными (подобных тем, которые строятся с помощью Ruby).

Основные функциональные возможности ASP.NET 4.0 и 4.5

В версиях .NET 4.0 и 4.5 к платформе веб-разработки Microsoft добавлены дополнительные возможности; ниже представлен список некоторых наиболее заметных веб-ориентированных средств.

- Возможность сжатия данных “состояния представления” с использованием стандарта GZIP.
- Обновленные определения браузеров для обеспечения корректной визуализации страниц ASP.NET в новых браузерах и устройствах (Google Chrome, Apple iPhone, устройства BlackBerry и т.п.).
- Возможность настройки вывода элементов управления проверкой достоверности с помощью каскадных таблиц стилей (CSS).
- Включение в библиотеку ASP.NET элемента управления Chart, который позволяет создавать страницы ASP.NET с наглядными диаграммами для сложного статистического и финансового анализа.
- Поддержка шаблонов проектов ASP.NET Model View Controller, которая уменьшает зависимость между уровнями приложения с помощью шаблона MVC (Model-View-Controller — модель-представление-контроллер). Этот подход совершенно не похож на разработку веб-сайтов и немного напоминает модель программирования веб-форм, которая рассматривается в данной книге.
- Многочисленные изменения для поддержки HTML 5.0.
- Интеграция с новыми асинхронными языковыми возможностями C# и VB.

В общем, этот список средств ASP.NET довольно показателен (а этот API-интерфейс содержит еще гораздо больше всяких возможностей, чем перечислено здесь). Если попытаться описать все средства ASP.NET, эта книга стала бы вдвое (а то и втрое) толще. Поскольку это нереально, мы рассмотрим только базовые средства ASP.NET, которые, скорее всего, будут использоваться вами каждый день. Исчерпывающие описания всех средств ASP.NET можно найти в документации .NET Framework 4.5 SDK.

На заметку! Если нужно хорошее руководство по разработке веб-приложений в ASP.NET, рекомендуется обратиться к книге Адама Фримена и Мэтью Мак-Дональда *Pro ASP.NET 4.5 in C#, Fifth Edition* (издательство Apress).

Построение однофайловой веб-страницы ASP.NET

Как было сказано, веб-страницу ASP.NET можно создать на основе одного из двух основных подходов, первый из которых заключается в построении единственного .aspx-файла, содержащего смесь кода серверной стороны и разметки HTML. В рамках такой модели однофайловой страницы код серверной стороны помещается внутрь области `<script>`, но сам код не является сценарием (например, на VBScript или JavaScript). Вместо этого код внутри блока `<script>` пишется на выбранном языке .NET (C#, Visual Basic и т.д.).

Если создаваемая веб-страница содержит очень немного кода (но значительный объем статической HTML-разметки), то модель однофайловой страницы более удобна, поскольку она позволяет видеть код и разметку в одном .aspx-файле. Кроме того, помещение процедурного кода и HTML-разметки в единый .aspx-файл обеспечивает и ряд других преимуществ.

- Страницы, написанные в соответствии с однофайловой моделью, несколько легче развертывать или отправлять другим разработчикам.
- Поскольку нет зависимости между множеством файлов, такую страницу проще переименовывать.
- Слегка упрощается манипулирование файлами в системе управления исходным кодом, поскольку все действия выполняются с единственным файлом.

Недостаток модели однофайловой страницы в том, что она приводит к появлению слишком сложных файлов, которые содержат и разметку пользовательского интерфейса, и программную логику. И все-таки мы начнем путешествие по ASP.NET именно с рассмотрения модели однофайловой страницы.

Мы создадим .aspx-файл, который выведет содержимое таблицы Inventory базы данных AutoLot (созданной в главе 21) с применением подключенного уровня (хотя, конечно, можно было бы задействовать и автономный уровень либо Entity Framework). Запустите Visual Studio и создайте новую веб-форму, выбрав пункт меню File⇒New⇒File (Файл⇒Создать⇒Файл), как показано на рис. 32.8 (в расположенном слева древовидном представлении нужно выбрать узел Web⇒C#).

Сохраните файл под именем Default.aspx в новом каталоге на жестком диске, что-бы позже его можно было легко найти (например, C:\MyCode\SinglePageModel).

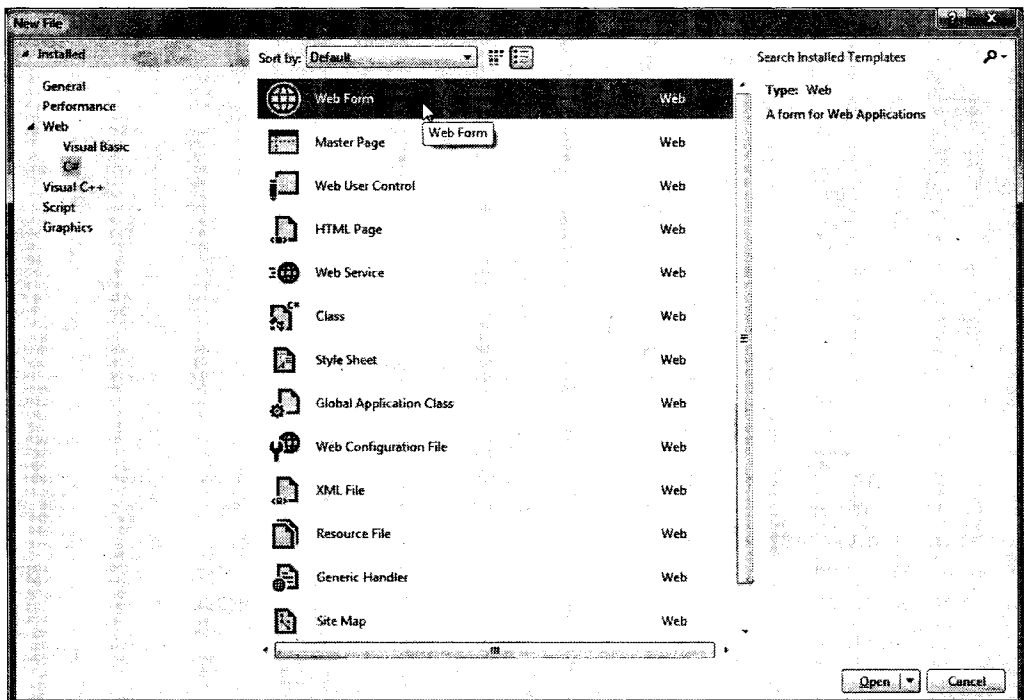


Рис. 32.8. Создание новой однофайловой страницы ASP.NET

Указание сборки AutoLotDAL.dll

Создайте с помощью проводника Windows подкаталог bin внутри папки SinglePageModel. Подкаталог со специальным именем bin — это зарегистрированное имя для механизма исполняющей среды ASP.NET. В папку \bin корневого каталога веб-сайта можно развернуть любые закрытые сборки, используемые веб-приложением. В нашем примере поместите в папку C:\MyCode\SinglePageModel\bin копию AutoLotDAL.dll (см. главу 21).

На заметку! Как будет показано далее в этой главе, при создании веб-приложения ASP.NET с помощью Visual Studio среда автоматически создает папку \bin и копирует любые ссылки на закрытые сборки.

Проектирование пользовательского интерфейса

Выберите в панели инструментов Visual Studio вкладку Standard (Стандартные) и перетащите на поверхность визуального конструктора страницы элементы управления Button, Label и GridView (виджет GridView находится на вкладке Data (Данные) панели инструментов), разместив их между открывающим и закрывающим элементами формы. Воспользуйтесь окном Properties для установки различных параметров отображения, а также подходящих имен для каждого виджета (через атрибут ID). Один из возможных вариантов компоновки показан на рис. 32.9. (В этом примере внешний вид страницы намеренно сохранен простым, чтобы минимизировать объем сгенерированной разметки, однако вы можете расширить его по своему усмотрению.)

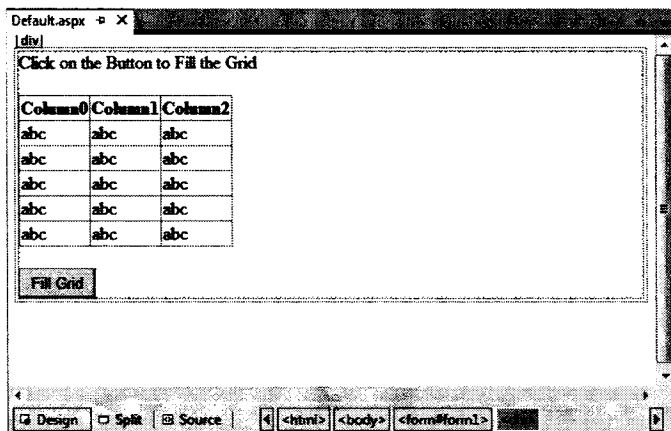


Рис. 32.9. Графический пользовательский интерфейс Default.aspx

Теперь найдите в разметке страницы раздел <form>. Обратите внимание, что каждый веб-элемент управления определен с использованием дескриптора <asp:>. После этого префикса указано имя веб-элемента ASP.NET (Label, GridView и Button). Перед закрывающим дескриптором элемента находится последовательность пар “имя/значение”, которые соответствуют настройкам, проведенным в окне Properties:

```
<form id="form1" runat="server">
<div>
  <asp:Label ID="lblInfo" runat="server"
    Text="Click on the Button to Fill the Grid">
  </asp:Label>
```

```

<br />
<br />
<asp:GridView ID="carsGridView" runat="server">
</asp:GridView>
<br />
<asp:Button ID="btnFillData" runat="server" Text="Fill Grid" />
</div>
</form>

```

Все детали веб-элементов управления ASP.NET будут рассматриваться в главе 33. А пока просто запомните, что веб-элементы управления — это объекты, обрабатываемые на веб-сервере, который автоматически возвращает их HTML-представление в исходящем HTTP-ответе. Помимо этого главного преимущества, веб-элементы управления ASP.NET имитируют модель программирования настольных приложений, при которой имена свойств, методов и событий обычно совпадают с их аналогами из Windows Forms/WPF.

Добавление логики доступа к данным

Теперь добавьте обработчик события Click для типа Button, используя окно Properties среды Visual Studio (через значок с изображением молнии). После этого в определении Button появится атрибут OnClick, которому присвоено имя введенного обработчика события Click:

```

<asp:Button ID="btnFillData" runat="server"
    Text="Fill Grid" OnClick="btnFillData_Click"/>

```

Затем в блоке <script> внутри .aspx-файла серверной стороны необходимо реализовать обработчик события Click. Обратите внимание, что его входные параметры в точности соответствуют целевому методу делегата System.EventHandler, который применялся во многих примерах этой книги:

```

<script runat="server">
    protected void btnFillData_Click(object sender, EventArgs args)
    {
    }
</script>

```

Следующий шаг связан с наполнением таблицы GridView с использованием функциональности сборки AutoLoDAL.dll. Для этого с помощью директивы <%@ Import %> нужно указать, что будет применяться пространство имен AutoLotConnectedLayer.

На заметку! При построении страницы в соответствии с однофайловой моделью кода необходимо использовать только директиву <%@ Import %>. В случае стандартного подхода с файлом кода пространства имен включаются в этот файл с помощью ключевого слова using языка C#. То же самое касается описанной ниже директивы <%@ Assembly %>.

Вдобавок посредством директивы <%@ Assembly %> понадобится информировать исполняющую среду ASP.NET о том, что эта однофайловая страница ссылается на сборку AutoLoDAL.dll (о директивах будет сказано ниже). Ниже приведена остальная логика страницы из файла Default.aspx (при необходимости измените строку соединения):

```

<%@ Page Language="C#" %>
<%@ Import Namespace = "AutoLotConnectedLayer" %>
<%@ Assembly Name = "AutoLoDAL" %>
<!DOCTYPE html>
<script runat="server">
    protected void btnFillData_Click(object sender, EventArgs args)
    {
        InventoryDAL dal = new InventoryDAL();
    }

```

```

dal.OpenConnection(@"Data Source=(local)\SQLEXPRESS;" +
    "Initial Catalog=AutoLot;Integrated Security=True");
carsGridView.DataSource = dal.GetAllInventoryAsList();
carsGridView.DataBind();
dal.CloseConnection();
}
</script>

```

Прежде чем приступить к изучению формата файла *.aspx, давайте произведем тестовый запуск. Сначала сохраните .aspx-файл. Теперь щелкните правой кнопкой мыши в любом месте визуального конструктора *.aspx и выберите в контекстном меню пункт View in Browser (Просмотреть в браузере). При этом запустится веб-сервер разработки ASP.NET, на котором будет размещена страница.

После обслуживания страницы сначала будут видны элементы управления Label и Button. Однако после щелчка на кнопке произойдет обратная отправка веб-серверу, и веб-элементы управления визуализируют соответствующие HTML-дескрипторы. На рис. 32.10 показан вывод после щелчка на кнопке Fill Grid (Заполнить сетку).

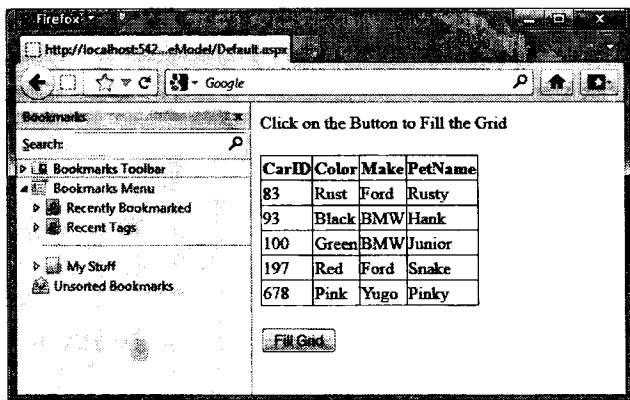


Рис. 32.10. ASP.NET предоставляет декларативную модель привязки данных

Пока что пользовательский интерфейс довольно скромен. Чтобы улучшить его, выберите в визуальном конструкторе элемент GridView, и в контекстном меню (открываемом с помощью небольшой стрелки в правом верхнем углу элемента) выберите ссылку Auto Format (Автоформат), как показано на рис. 32.11.

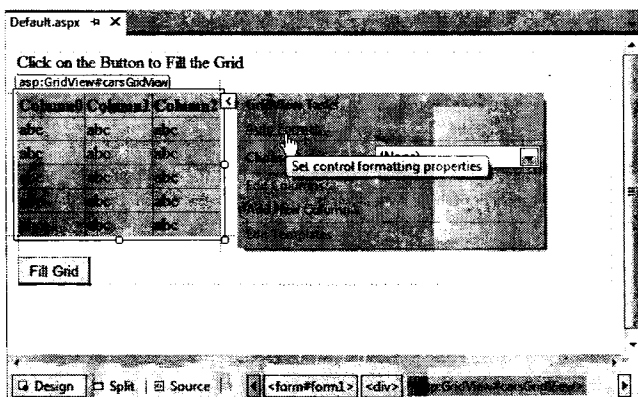


Рис. 32.11. Конфигурирование элемента управления GridView

В открывшемся диалоговом окне выберите подходящий шаблон (например, Slate). После щелчка на кнопке ОК просмотрите сгенерированное объявление элемента управления, которое существенно больше предыдущего:

```
<asp:GridView ID="carsGridView" runat="server" BackColor="White"
    BorderColor="#E7E7FF" BorderStyle="None" BorderWidth="1px" CellPadding="3"
    GridLines="Horizontal">
    <AlternatingRowStyle BackColor="#F7F7F7" />
    <FooterStyle BackColor="#B5C7DE" ForeColor="#4A3C8C" />
    <HeaderStyle BackColor="#4A3C8C" Font-Bold="True" ForeColor="#F7F7F7" />
    <PagerStyle BackColor="#E7E7FF" ForeColor="#4A3C8C" HorizontalAlign="Right" />
    <RowStyle BackColor="#E7E7FF" ForeColor="#4A3C8C" />
    <SelectedRowStyle BackColor="#738A9C" Font-Bold="True" ForeColor="#F7F7F7" />
    <SortedAscendingCellStyle BackColor="#F4F4FD" />
    <SortedAscendingHeaderStyle BackColor="#5A4C9D" />
    <SortedDescendingCellStyle BackColor="#D8D8F0" />
    <SortedDescendingHeaderStyle BackColor="#3E3277" />
</asp:GridView>
```

Снова запустив приложение и щелкнув на кнопке Fill Grid, вы увидите более интересный пользовательский интерфейс (рис. 32.12).

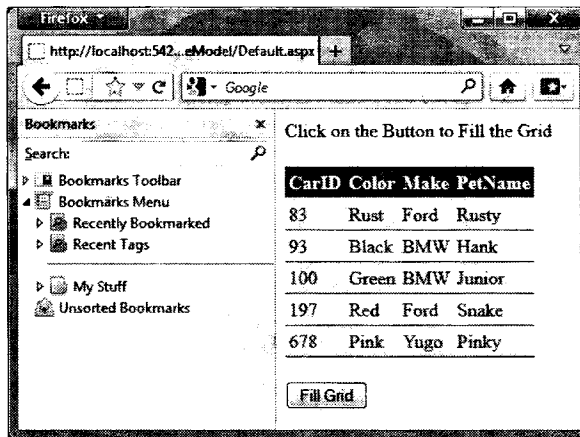


Рис. 32.12. Тестовая страница в более насыщенном оформлении

Просто, не правда ли? Разумеется, как всегда, сложности кроются в деталях, поэтому давайте немного углубимся в состав этого .aspx-файла, начав с роли директивы `<%@ Page... %>`. Имейте в виду, что рассмотренные здесь темы будут непосредственно применимы к рекомендуемой модели с файлом кода, которая описана ниже.

Роль директив ASP.NET

Типичный файл .aspx обычно начинается набором директив. Директивы ASP.NET всегда помечаются маркерами `<%@ ... %>` и могут содержать различные атрибуты, информирующие исполняющую среду ASP.NET о том, как следует обрабатывать конкретную директиву.

Каждый файл .aspx должен содержать, как минимум, директиву `<%@ Page %>`, которая служит для определения управляемого языка, используемого внутри страницы (в атрибуте `language`). Директива `<%@ Page %>` позволяет также определить имя связанного файла отделенного кода (рассматривается ниже) и т.д. Наиболее интересные атрибуты директивы `<%@ Page %>` перечислены в табл. 32.1.

Таблица 32.1. Избранные атрибуты директивы <%@ Page %>

Атрибут	Описание
CodePage	Указывает имя связанного файла отделенного кода
EnableTheming	Указывает, поддерживают ли элементы управления на .aspx-странице темы ASP.NET
EnableViewState	Указывает, поддерживается ли состояние представления между запросами страницы (более подробно это свойство рассматривается в главе 33)
Inherits	Определяет класс в файле отделенного кода, от которого наследуется страница; может быть любым классом, производным от System.Web.UI.Page
MasterPageFile	Устанавливает мастер-страницу, используемую в сочетании с текущей .aspx-страницей
Trace	Указывает, включена ли трассировка

В дополнение к директиве <%@ Page %>, файл *.aspx может содержать различные директивы <%@ Import %> для явного указания пространств имен, требуемых текущей страницей, и директивы <%@ Assembly %> для описания внешних библиотек кода, используемых сайтом (обычно расположенных в папке \bin веб-сайта).

В этом примере было указано, что применяются типы из пространства имен AutoLotConnectedLayer сборки AutoLotDAL.dll. Понятно, что если необходимо использовать дополнительные пространства имен .NET, нужно просто указать несколько директив <%@ Import %> / <%@ Assembly %>.

Помимо директив <%@ Page %>, <%@ Import %> и <%@ Assembly %>, в ASP.NET определен ряд других директив, которые могут появляться в .aspx-файле; их описание приводится позже. Примеры применения других директив вы найдете в оставшихся главах.

Анализ блока script

В однофайловой модели страницы .aspx-файл может содержать логику сценариев серверной стороны, которая выполняется на веб-сервере. В таком случае *критически важно*, чтобы все блоки кода серверной стороны были определены для выполнения на сервере с помощью атрибута runat="server". Если атрибут runat="server" не указан, исполняющая среда предполагает, что был написан блок сценария *клиентской* стороны, предназначенный для встраивания в исходящий HTTP-ответ, и генерирует исключение. Ниже показан правильный блок <script> серверной стороны.

На заметку! Все веб-элементы управления ASP.NET должны иметь в своем открывающем объявлении атрибут runat="server". В противном случае они не будут генерировать соответствующую HTML-разметку в исходящий HTTP-ответ.

```
<script runat="server">
    protected void btnFillData_Click(object sender, EventArgs args)
    {
        InventoryDAL dal = new InventoryDAL();
        dal.OpenConnection(@"Data Source=(local)\SQLEXPRESS;" +
            "Initial Catalog=AutoLot;Integrated Security=True");
        carsGridView.DataSource = dal.GetAllInventory();
        carsGridView.DataBind();
        dal.CloseConnection();
    }
</script>
```

Сигнатура этого вспомогательного метода должна выглядеть удивительно знакомой. Вспомните, что обработчик событий каждого элемента управления должен соответствовать шаблону, определенному связанным делегатом .NET. Таким делегатом является `System.EventHandler`, который может вызывать только методы, принимающие `System.Object` в первом параметре и `System.EventArgs` — во втором.

Анализ объявлений элементов управления ASP.NET

Последний момент, который нас интересует в этом первом примере — объявление веб-элементов управления `Button`, `Label` и `GridView`. Подобно классическому ASP и низкоуровневому HTML, веб-виджеты ASP.NET помещаются в область элементов `<form>`. Однако в нашем случае открывающий дескриптор `<form>` помечен атрибутом `runat="server"`, а элементы управления — дескрипторным префиксом `asp:..` Любой элемент с таким префиксом является членом библиотеки элементов управления ASP.NET и имеет соответствующее представление в виде класса C# в определенном пространстве имен .NET библиотеки базовых классов .NET. Вот что здесь находится:

```
<form id="form1" runat="server">
<div>
  <asp:Label ID="lblInfo" runat="server"
    Text="Click on the Button to Fill the Grid">
  </asp:Label>
  <br />
  <br />
  <asp:GridView ID="carsGridView" runat="server">
    ...
  </asp:GridView>
  <br />
  <asp:Button ID="btnFillData" runat="server" Text="Fill Grid"
    OnClick="btnFillData_Click"/>
</div>
</form>
```

Пространство имен `System.Web.UI.WebControls` сборки `System.Web.dll` содержит большинство веб-элементов управления ASP.NET. Открыв браузер объектов Visual Studio, в нем можно найти, например, элемент управления `DataGrid` (рис. 32.13).

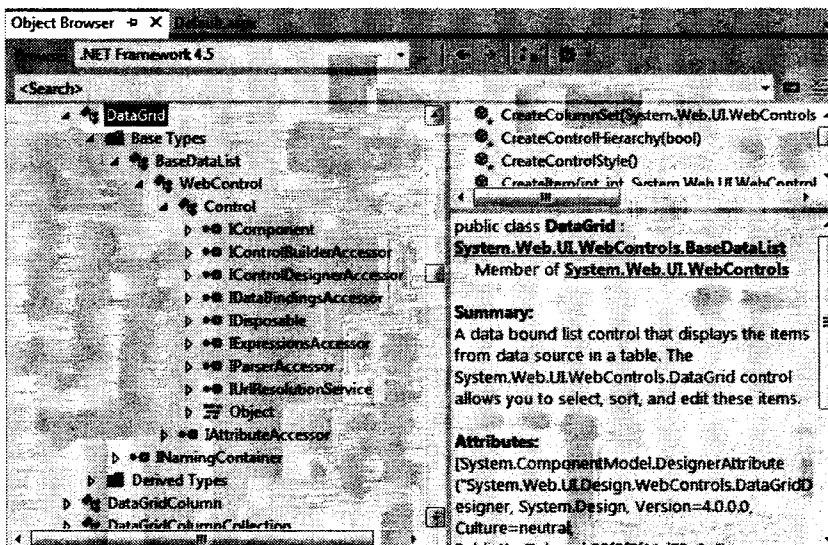


Рис. 32.13. Все объявления элементов управления ASP.NET отображаются на типы классов .NET

Как видите, на самой вершине цепочки наследования любого веб-элемента управления ASP.NET находится `System.Object`. Родительский класс `WebControl` — это общая база всех элементов управления ASP.NET, он определяет все общие свойства пользовательского интерфейса (`BackColor`, `Height` и т.д.). Класс `Control` также очень распространен, однако в нем определены члены, больше ориентированные на инфраструктуру (привязка данных, состояние представления и т.п.), а не на внешний вид дочерних элементов. Дополнительные сведения об этих классах вы узнаете в главе 33.

Исходный код. Веб-сайт `SinglePageModel` доступен в подкаталоге `Chapter 32`.

Построение веб-страницы ASP.NET с использованием файлов кода

Хотя однофайловая модель иногда бывает полезной, стандартный подход, принятый в Visual Studio (при создании нового веб-проекта), предусматривает использование приема, который называется *отделенным кодом* и позволяет разнести программный код серверной стороны и логику презентации HTML в два разных файла. Эта модель работает довольно хорошо, когда страницы содержат существенный объем кода или когда созданием одного веб-сайта занимается множество разработчиков. Модель отделенного кода предоставляет и другие преимущества, перечисленные ниже.

- Поскольку страницы с отделенным кодом обеспечивают четкое разделение HTML-разметки и кода, можно организовать параллельную работу дизайнеров над разметкой и программистов — над кодом C#.
- Код не виден дизайнерам страниц и прочему персоналу, который имеет дело только с разметкой страницы (как и можно было догадаться, специалистам по HTML-разметке не всегда интересны детали кода C#).
- Файлы кода можно использовать с несколькими `.aspx`-файлами.

Каков бы ни был выбранный подход, он никак не отражается на производительности. В действительности многие веб-приложения ASP.NET выигрывают от построения сайтов с применением обоих подходов. Для демонстрации модели с отделенным кодом давайте пересоздадим предыдущий пример, используя шаблон ASP.NET Empty Web Site (Пустой веб-сайт ASP.NET) в Visual Studio. Выберите пункт меню `File⇒New⇒Web Site (Файл⇒Создать⇒Веб-сайт)` и укажите шаблон ASP.NET Empty Web Site (рис. 32.14).

На рис. 32.14 видно, что имеется возможность выбора местоположения для нового сайта. Выбор в списке `Web location (Веб-расположение)` варианта `File System (Файловая система)` обеспечивает помещение файлов содержимого в локальный каталог с обслуживанием страниц веб-сервером разработки ASP.NET. В случае выбора вариантов `FTP` или `HTTP` сайт будет размещен внутри нового виртуального каталога, обслуживаемого IIS. В рассматриваемом примере не имеет значения, какой вариант будет выбран, но для простоты имеет смысл выбрать `File System` и указать новую папку по имени `C:\CodeBehindPageModel`.

На заметку! Шаблон проекта ASP.NET Empty Web Site автоматически включает файл `web.config`, который похож на файл `App.config` для настольного приложения. Формат этого файла рассматривается далее в этой главе.

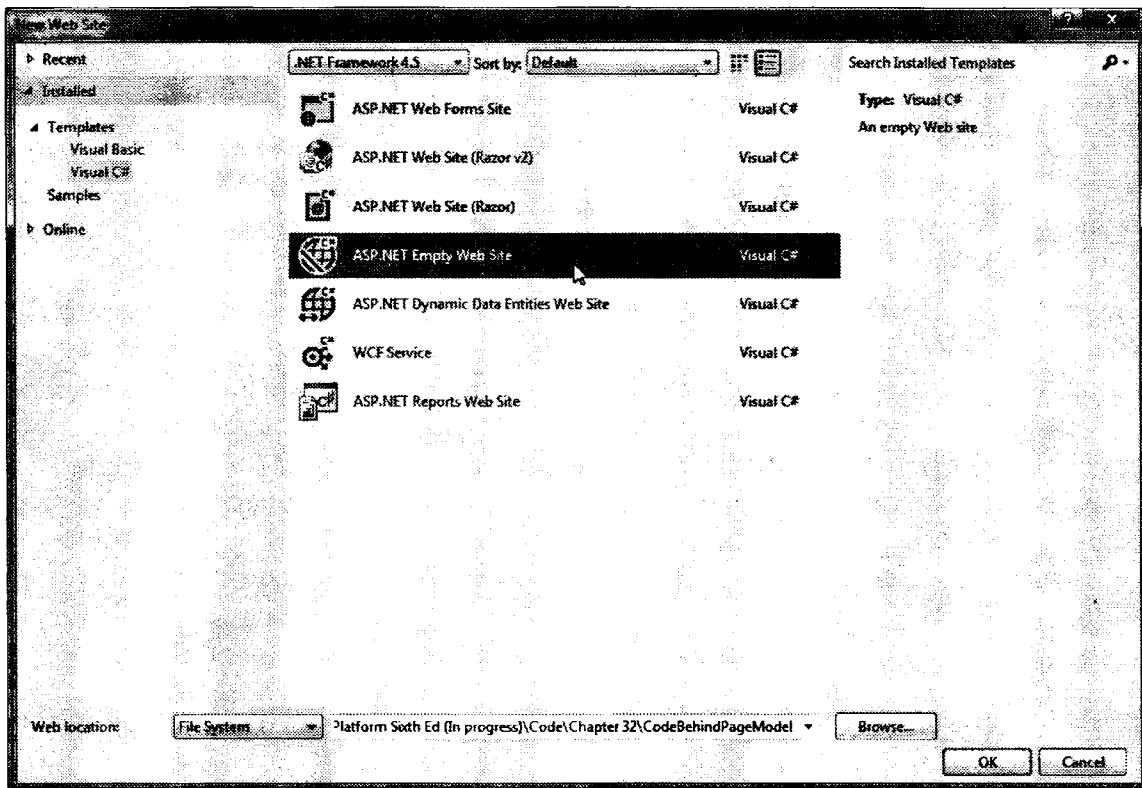


Рис. 32.14. Выбор шаблона ASP.NET Empty Web Site

Теперь, выбрав пункт меню Website⇒Add New Item... (Веб-сайт⇒Добавить новый элемент), вставьте новый элемент Web Form (Веб-форма) по имени Default.aspx. Обратите внимание, что по умолчанию флажок Place code in separate file (Поместить код в отдельный файл) автоматически отмечен — именно это и требуется (рис. 32.15).

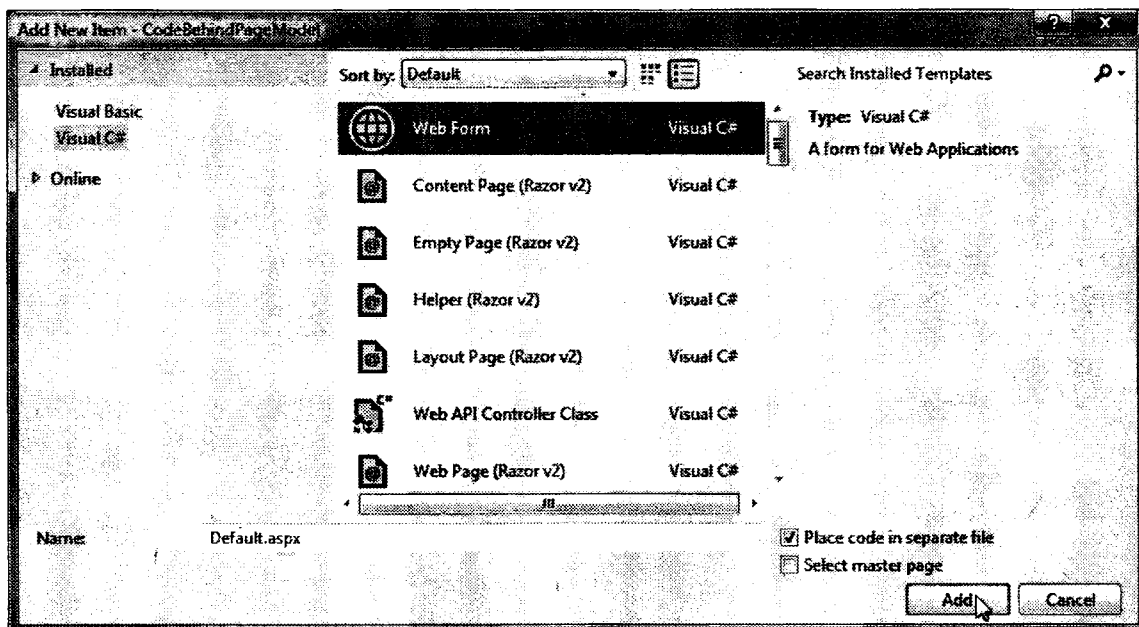


Рис. 32.15. Вставка нового элемента Web Form с разделением кода

Создайте в визуальном конструкторе пользовательский интерфейс, состоящий из элементов управления Label, Button и GridView, и настройте его в окне Properties по собственному усмотрению. Можно просто скопировать в новый .aspx-файл объявления элементов управления ASP.NET из примера SingleFilePageModel.

Учитывая, что здесь все уже знакомо, все нюансы повторно рассматриваться не будут (нужно только обеспечить вставку объявлений элементов управления между дескрипторами `<form>` и `</form>`).

Обратите внимание, что директива `<% Page %>`, используемая в модели файла кода, дополнена новыми атрибутами:

```
<% Page Language="C#" AutoEventWireup="true"
    CodeFile="Default.aspx.cs" Inherits="_Default" %>
```

Атрибут `CodeFile` используется для указания соответствующего внешнего файла, который содержит код логики для этой страницы. По умолчанию эти файлы отделенного кода именуются путем добавления суффикса `.cs` к имени файла `.aspx` (в рассматриваемом примере получается `Default.aspx.cs`). Заглянув в Solution Explorer, вы увидите, что файл отделенного кода находится в подузле под значком веб-формы (рис. 32.16).

Открыв файл отделенного кода, вы найдете в нем частичный класс, унаследованный от `System.Web.UI.Page`, с поддержкой обработки события `Load`. Обратите внимание, что имя этого класса (`_Default`) идентично атрибуту `Inherits` внутри директивы `<% Page %>`:

```
public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
    }
}
```

Ссылка на сборку AutoLotDAL.dll

Как упоминалось ранее, при создании проектов веб-приложений с использованием Visual Studio нет необходимости вручную создавать подкаталог `\bin` и копировать в него закрытые сборки. Для целей данного примера откройте через пункт меню **Website (Веб-сайт)** диалоговое окно **Add Reference (Добавить ссылку)** и укажите ссылку на `AutoLotDAL.dll`. После этого в Solution Explorer появится новая папка `\bin` (рис. 32.17).

Изменение файла кода

Обработайте событие `Click` для типа `Button`, дважды щелкнув на элементе управления `Button` в визуальном конструкторе. Как и ранее, к определению `Button` добавляется атрибут `OnClick`. Однако обработчик события серверной стороны теперь не размещается в области `<script>` файла `*.aspx`, а становится методом класса `_Default`.

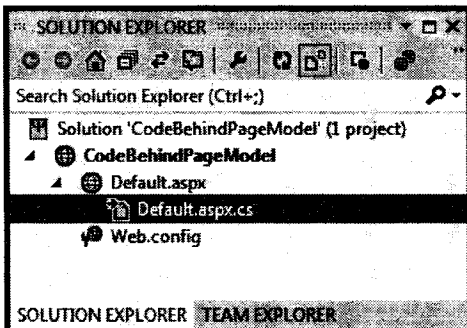


Рис. 32.16. Файл отделенного кода для заданного файла `*.aspx`

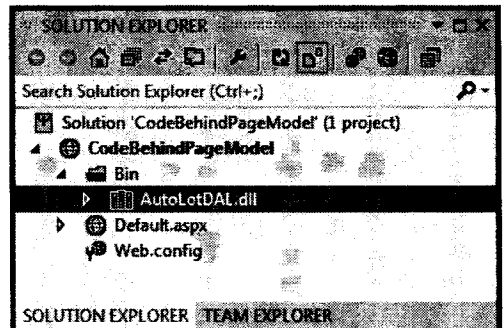


Рис. 32.17. В веб-проектах Visual Studio обычно используются специальные папки ASP.NET

Для завершения этого примера добавьте в файл отделенного кода оператор `using` для `AutoLotConnectedLayer` и реализуйте обработчик, используя прежнюю логику (здесь также может понадобиться изменить строку соединения):

```
using AutoLotConnectedLayer;
public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
    }

    protected void btnFillData_Click(object sender, EventArgs e)
    {
        InventoryDAL dal = new InventoryDAL();
        dal.OpenConnection(@"Data Source=(local)\SQLEXPRESS;" +
            "Initial Catalog=AutoLot;Integrated Security=True");
        carsGridView.DataSource = dal.GetAllInventoryAsList();
        carsGridView.DataBind();
        dal.CloseConnection();
    }
}
```

В этот момент можно запустить веб-сайт, нажав комбинацию клавиш <Ctrl+F5>. Как и прежде, запустится веб-сервер разработки ASP.NET и обслужит страницу для принимающего браузера.

Отладка и трассировка страниц ASP.NET

При разработке веб-проектов ASP.NET можно применять те же приемы отладки, что и в любых других типах проектов Visual Studio: размещать точки останова в файле отделенного кода (а также во встроенных блоках <script> внутри файла *.aspx), запускать сеанс отладки (по умолчанию нажатием клавиши <F5>) и пошагово выполнять код.

Однако для отладки веб-приложений ASP.NET сайт должен содержать правильно сконфигурированный файл `web.config`. По умолчанию все веб-проекты Visual Studio автоматически получают свой файл `web.config`, но поддержка отладки в нем изначально отключена. После запуска сеанса отладки IDE-среда запросит о необходимости изменения файла `web.config` для включения отладки. Если это нужно, элемент <compilation> в файле `web.config` будет модифицирован следующим образом:

```
<compilation debug="true" targetFramework="4.5"/>
```

Кроме того, можно также включить *поддержку трассировки* для .aspx файла, присвоив атрибуту `Trace` значение `true` внутри директивы <%@ Page %> (чтобы включить трассировку для всего сайта понадобится модифицировать файл `web.config`):

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeFile="Default.aspx.cs" Inherits="_Default" Trace="true" %>
```

После этого сгенерированная HTML-разметка будет содержать многочисленные детали, касающиеся предыдущего запроса/ответа HTTP (серверные переменные, переменные сеанса и приложения, запрос/ответ и т.п.).

Чтобы вставить собственные сообщения трассировки, можно воспользоваться свойством `Trace`, унаследованным от типа `System.Web.UI.Page`. Всякий раз, когда нужно занести в журнал специальное сообщение (из блока сценария или файла исходного кода C#), просто вызывайте статический метод `Trace.Write()`. Первый аргумент представляет имя специальной категории, а второй — сообщение трассировки.

Для примера измените обработчик Click элемента Button следующим образом:

```
protected void btnFillData_Click(object sender, EventArgs e)
{
    Trace.Write("CodeFileTraceInfo!", "Filling the grid!");
    ...
}
```

Запустите проект снова и щелкните на кнопке. В журнале появится специальная категория и специальное сообщение. На рис. 32.18 обратите внимание на выделенное сообщение с информацией трассировки.

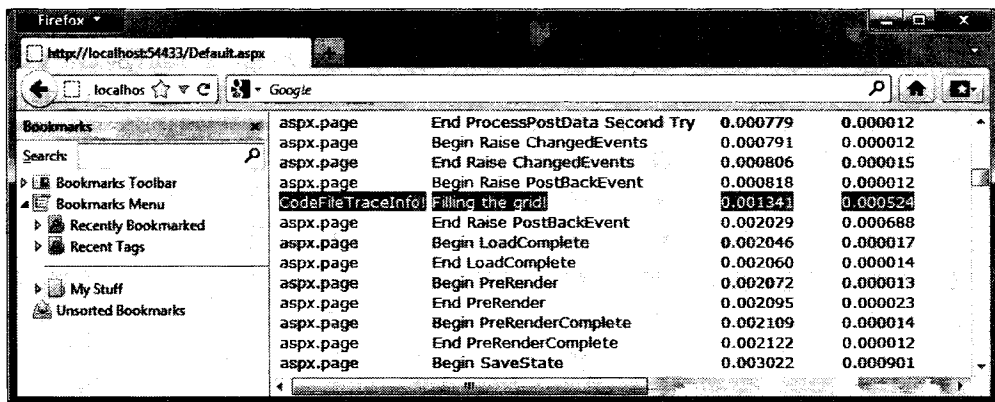


Рис. 32.18. Журнал со специальными сообщениями трассировки

К этому моменту было показано, как строить веб-страницу ASP.NET применением однофайлового подхода и подхода с отделенным кодом. Оставшийся материал этой главы посвящен анализу состава веб-проекта ASP.NET, а также способам взаимодействия с запросами/ответами HTTP и жизненному циклу класса, производного от Page. Однако прежде чем двигаться дальше, следует прояснить разницу между веб-сайтом ASP.NET и веб-приложением ASP.NET.

Исходный код. Веб-сайт CodeBehindPageModel доступен в подкаталоге Chapter 32.

Сравнение веб-сайтов и веб-приложений ASP.NET

Перед построением нового веб-приложения ASP.NET необходимо выбрать один из двух форматов проектов: *веб-сайт ASP.NET* или *веб-приложение ASP.NET*. Этот выбор определяет, как Visual Studio организует и обрабатывает стартовые файлы веб-приложения, тип начальных файлов проекта и степень контроля над результирующим составом скомпилированной сборки .NET.

Когда технология ASP.NET впервые появилась в составе .NET 1.0, единственным выбором было *веб-приложение*. Эта модель обеспечивает непосредственный контроль над именем и местоположением скомпилированной выходной сборки.

Веб-приложения удобны, когда нужно переносить старые веб-сайты .NET 1.1 в проекты .NET 2.0 и последующих версий. Они также полезны, если требуется создать единственное решение Visual Studio, которое может содержать несколько проектов (например, веб-приложение и любые связанные библиотеки кода .NET). Чтобы построить веб-приложение ASP.NET, выберите пункт меню File⇒New Project... (Файл⇒Создать проект...) и укажите шаблон из категории Web (рис. 32.19).

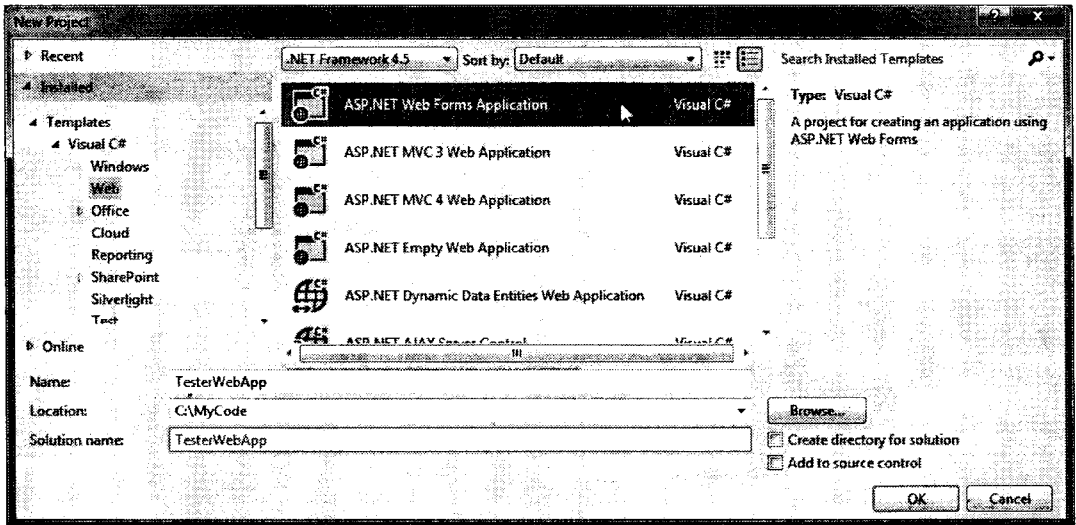


Рис. 32.19. Шаблоны веб-приложений Visual Studio

Предположим, что был создан новый проект веб-приложения ASP.NET. В нем находится большое количество стартовых файлов (назначение которых станет ясным в последующих главах). Однако наиболее важно отметить, что каждая веб-страница ASP.NET состоит из трех файлов: *.aspx (разметка), *.designer.cs (сгенерированный визуальным конструктором код C#) и основной файл кода C# (обработчики событий, специальные методы и т.п.). На рис. 32.20 показан пример.

На заметку! Поскольку шаблоны проектов ASP.NET в Visual Studio могут генерировать значительный объем стартового кода (мастер-страницы, страницы содержимого, библиотеки сценариев, страница входа и т.д.), в книге будет применяться только шаблон ASP.NET Empty Web Site (Пустой веб-сайт ASP.NET). Но когда вы закончите чтение глав по ASP.NET, обязательно создайте новый проект веб-сайта ASP.NET и первым делом просмотрите стартовый код.

В отличие от этого, шаблоны проектов веб-сайтов ASP.NET в Visual Studio, доступные через пункт меню File⇒New Web Site... (Файл⇒Создать веб-сайт...), содержат файл *.designer.cs для находящегося в памяти частичного класса. Более того, проекты веб-сайтов ASP.NET поддерживают несколько папок со специальными именами, такими как App_Code. В эту папку можно поместить любые файлы кода C# (или VB), которые не отображаются напрямую на веб-страницы, и компилятор времени выполнения при необходимости динамически скомпилирует их. Это значительное упрощение обычного процесса создания выделенной библиотеки кода .NET и указания ссылок на нее в новых проектах.

Кстати, проект веб-сайта можно перенести в неизменном виде на производственный веб-

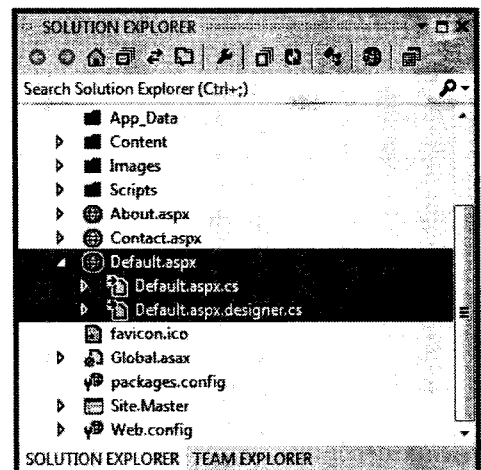


Рис. 32.20. В модели веб-приложения каждая веб-страница состоит из трех файлов

сервер без необходимости в предварительной компиляции сайта, что нужно в случае веб-приложения ASP.NET.

В этой книге используются типы проектов веб-сайтов ASP.NET, поскольку они упрощают процесс построения веб-приложений на платформе .NET. Тем не менее, независимо от выбранного подхода, вы будете иметь доступ к одной и той же модели программирования.

Структура каталогов веб-сайта ASP.NET

Новый проект веб-сайта ASP.NET может содержать некоторое количество подкаталогов со специальными именами, каждый из которых имеет определенное значение для исполняющей среды ASP.NET. Эти специальные подкаталоги описаны в табл. 32.2.

Таблица 32.2. Специальные подкаталоги ASP.NET

Подкаталог	Описание
App_Browsers	Папка для файлов определений браузеров, используемых для идентификации индивидуальных браузеров и определения их возможностей
App_Code	Папка исходного кода для компонентов или классов, которые должны компилироваться как составные части приложения. ASP.NET компилирует код в этой папке при запросе страниц. Код в папке App_Code автоматически доступен приложению
App_Data	Папка для хранения файлов Access (*.mdb), файлов SQL Express (*.mdf), файлов XML или других источников данных
App_GlobalResources	Папка для файлов *.resx, доступных программно из кода приложения
App_LocalResources	Папка для файлов *.resx, привязанных к определенной странице
App_Themes	Папка, содержащая коллекцию файлов, которые определяют внешний вид веб-страниц и элементов управления ASP.NET
App_WebReferences	Папка для прокси-классов, схем и прочих файлов, связанных с использованием веб-служб в приложении
Bin	Папка для скомпилированных закрытых сборок (.dll-файлов). Сборки из папки Bin автоматически доступны приложению

Любой из этих известных подкаталогов можно явно добавить в текущее веб-приложение, выбрав пункт меню Website⇒Add ASP.NET Folder (Веб-сайт⇒Добавить папку ASP.NET). Тем не менее, во многих случаях IDE-среда делает это автоматически, при естественном добавлении соответствующих файлов к сайту. Например, вставка нового файла класса в проект автоматически добавит в структуру каталогов папку App_Code, если она еще не существует.

Ссылка на сборки

Хотя шаблоны веб-сайтов генерируют .sln-файл для загрузки .aspx-файла в IDE-среду, теперь больше нет файла *.csproj. С другой стороны, проекты веб-приложений ASP.NET записывают все внешние сборки в файл *.csproj. Так куда же записываются внешние сборки под ASP.NET?

Ранее уже было показано, что при ссылке на закрытую сборку Visual Studio автоматически создает внутри структуры каталогов подкаталог \bin для хранения локальной копии двоичной сборки. Когда базовый код использует типы из этих библиотек кода, они автоматически загружаются по требованию.

При ссылке на разделяемую сборку, находящуюся в глобальном кеше сборок (Global Assembly Cache — GAC), Visual Studio автоматически вставляет в текущее веб-решение файл `web.config` (если его еще нет) и записывает внешнюю ссылку в элемент `<assemblies>`. Например, если выбрать пункт меню Website⇒Add Reference (Веб-сайт⇒Добавить ссылку) и указать разделяемую сборку (скажем, `System.Security.dll`), то файл `web.config` будет модифицирован следующим образом:

```
<assemblies>
  <add assembly="System.Security, Version=4.0.0.0,
    Culture=neutral, PublicKeyToken=B03F5F7F11D50A3A"/>
</assemblies>
```

Как видите, каждая сборка описывается с использованием одной и той же информации, требуемой для динамической загрузки методом `Assembly.Load()` (см. главу 15).

Роль папки App_Code

Папка `App_Code` используется для хранения файлов кода, которые не привязаны непосредственно к конкретной веб-странице (как файл отделенного кода), но должны быть скомпилированы для использования веб-сайтом. Код из папки `App_Code` будет автоматически компилироваться по мере необходимости. После этого сборка будет доступна любому другому коду веб-сайта. В этом отношении папка `App_Code` похожа на папку `Bin`, только в ней можно сохранять исходный код, а не скомпилированный. Основное преимущество такого подхода в том, что появляется возможность определять специальные типы для веб-приложения, не компилируя их отдельно.

Единственная папка `App_Code` может содержать код на разных языках программирования. Во время выполнения для генерации необходимой сборки вызывается соответствующий компилятор. Но если необходимо разбить код на части, то можно определить несколько подкаталогов для хранения любого количества файлов управляемого кода (*.`vb`, *.`cs` и т.д.).

Например, предположим, что папка `App_Code` добавлена в корневой каталог приложения веб-сайта с двумя подпапками, `MyCSharpCode` и `MyVbNetCode`, которые содержат файлы, специфичные к языку программирования. После этого данные подкаталоги можно указать в файле `web.config` с помощью элемента `<codeSubDirectories>`, вложенного в элемент `<configuration>`:

```
<compilation debug="true" strict="false" explicit="true">
  <codeSubDirectories>
    <add directoryName="MyCSharpCode" />
    <add directoryName="MyVbNetCode" />
  </codeSubDirectories>
</compilation>
```

На заметку! Папка `App_Code` также используется для хранения файлов, которые не являются языковыми, но необходимы по другим причинам (*.`xsd`, *.`wsdl` и т.д.).

Помимо `Bin` и `App_Code`, существуют еще два дополнительных специальных подкаталога — `App_Data` и `App_Themes`; оба они рассматриваются в последующих главах. Как всегда, за подробной информацией относительно остальных подкаталогов ASP.NET обращайтесь к документации .NET Framework 4.5 SDK.

Цепочка наследования для типа Page

Все веб-страницы .NET в конечном итоге наследуются от класса `System.Web.UI.Page`. Подобно любому базовому классу, этот тип предоставляет полиморфный ин-

терфейс всем производным типам. Однако тип Page — не единственный член иерархии наследования. Если найти в браузере объектов Visual Studio класс System.Web.UI.Page (внутри сборки System.Web.dll), то можно увидеть, что Page “является” классом TemplateControl, который, в свою очередь, “является” классом Control, а тот — классом Object (рис. 32.21).

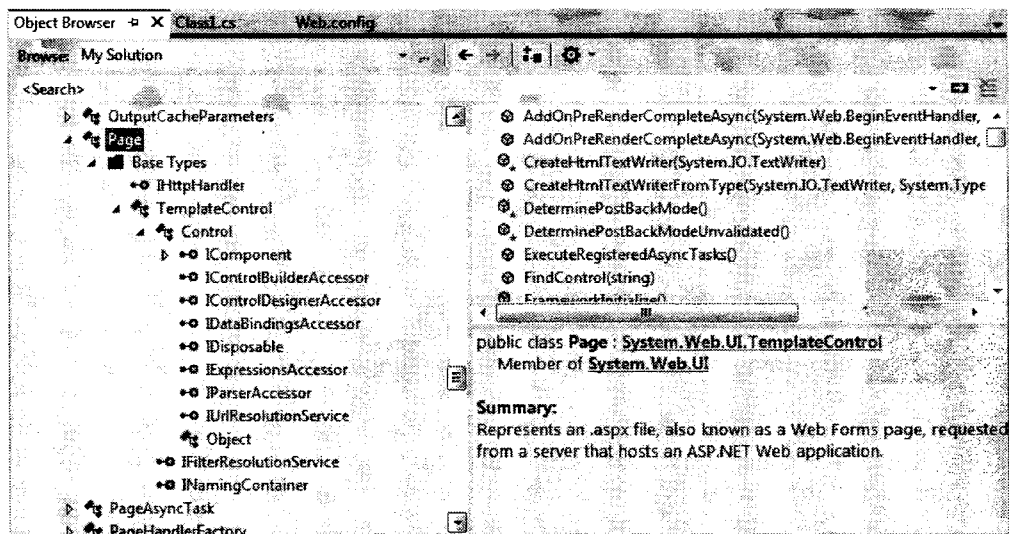


Рис. 32.21. Цепочка наследования для типа Page

Каждый из этих базовых классов добавляет свою часть функциональности в каждый файл *.aspx. В большинстве проектов используются члены, определенные внутри родительских классов Page и Control. Функциональность из класса System.Web.UI.TemplateControl интересна только при создании специальных элементов управления Web Forms или для вмешательства в процесс визуализации.

Первый родительский класс, представляющий интерес — это собственно Page. Он содержит многочисленные свойства, которые позволяют взаимодействовать с различными веб-примитивами, такими как переменные приложения и сеанса, запрос/ответ HTTP, поддержка тем и т.д. Некоторые (но, конечно, не все) основные свойства описаны в табл. 32.3.

Таблица 32.3. Избранные свойства типа Page

Свойство	Описание
Application	Позволяет взаимодействовать с данными, которые доступны по всему веб-сайту всем пользователям
Cache	Позволяет взаимодействовать с объектом кеша для текущего веб-сайта
ClientTarget	Позволяет указать, как данная страница должна визуализироваться в запрашивающем браузере
IsPostBack	Получает значение, указывающее, загружается страница в ответ на клиентскую обратную отправку или при обращении к ней в первый раз
MasterPageFile	Устанавливает мастер-страницу для текущей страницы
Request	Предоставляет доступ к текущему HTTP-запросу
Response	Позволяет взаимодействовать с исходящим HTTP-ответом

Свойство	Описание
Server	Предоставляет доступ к объекту <code>HttpServerUtility</code> , который содержит различные вспомогательные функции на стороне сервера
Session	Позволяет взаимодействовать с данными сеанса, используемыми текущей страницей
Theme	Получает или устанавливает тему, используемую текущей страницей
Trace	Предоставляет доступ к объекту <code>TraceContext</code> , который позволяет записывать в журнал специальные сообщения во время сеансов отладки

Взаимодействие с входящим HTTP-запросом

Как было показано ранее в этой главе, базовый поток веб-приложения предусматривает запрос веб-страницы клиентом, возможный ввод информации пользователем и щелчок на кнопке **Submit** (Отправить) для обратной отправки данных HTML-формы заданной веб-странице на обработку. В большинстве случаев в открывающем дескрипторе оператора `form` присутствуют атрибуты `action` и `method`, указывающие файл на веб-сервере, которому будут отправлены данные из различных HTML-виджетов, а также метод отправки этих данных (GET или POST):

```
<form name="defaultPage" id="defaultPage"
      action="http://localhost/Cars/ClassicAspPage.asp" method = "GET">
  ...
</form>
```

Все страницы ASP.NET поддерживают свойство `System.Web.UI.Page.Request`, которое обеспечивает доступ к экземпляру класса `HttpRequest` (основные члены этого класса перечислены в табл. 32.4).

Таблица 32.4. Члены класса `HttpRequest`

Член	Описание
<code>ApplicationPath</code>	Получает виртуальный корневой путь приложения ASP.NET на сервере
<code>Browser</code>	Предоставляет информацию о возможностях клиентского браузера
<code>Cookies</code>	Получает коллекцию cookie-наборов, отправленную клиентским браузером
<code>FilePath</code>	Указывает виртуальный путь для текущего запроса
<code>Form</code>	Получает коллекцию переменных HTTP-формы
<code>Headers</code>	Получает коллекцию HTTP-заголовков
<code>HttpMethod</code>	Указывает метод передачи HTTP-данных, используемый клиентом (GET/POST)
<code>IsSecureConnection</code>	Указывает, является ли подключение HTTP защищенным (т.е. HTTPS)
<code>QueryString</code>	Получает коллекцию переменных строки HTTP-запроса
<code>RawUrl</code>	Получает изначальный URL текущего запроса
<code>RequestType</code>	Указывает HTTP-метод передачи данных, применяемый клиентом (GET/POST)
<code>ServerVariables</code>	Получает коллекцию переменных веб-сервера
<code>UserHostAddress</code>	Получает IP-адрес хоста удаленного клиента
<code>UserHostName</code>	Получает DNS-имя удаленного клиента

В дополнение к этим свойствам класс `HttpRequest` содержит несколько полезных методов, часть которых перечислена ниже.

- `MapPath()`. Отображает виртуальный путь в запрошенном URL на физический путь на сервере для текущего запроса.
- `SaveAs()`. Сохраняет информацию о текущем HTTP-запросе в файле на веб-сервере, что полезно для целей отладки.
- `ValidateInput()`. Если включено средство проверки достоверности с помощью атрибута `Validate` директивы `Page`, этот метод можно вызвать для проверки всех введенных пользователем данных (включая cookie-данные) по заранее определенному списку потенциально опасных входных данных.

Получение статистики о браузере

Первым интересным аспектом типа `HttpRequest` является свойство `Browser`, которое обеспечивает доступ к лежащему в основе объекту `HttpBrowserCapabilities`. В свою очередь, объект `HttpBrowserCapabilities` содержит многочисленные члены, которые позволяют программно получить статистику о браузере, отправившем входной HTTP-запрос.

Создайте новый проект ASP.NET Empty Web Site (Пустой веб-сайт ASP.NET) по имени `FunWithPageMembers`, выбрав пункт меню `File⇒New Website` (Файл⇒Создать веб-сайт), а затем вариант `File System` в списке `Web location`.

Вначале построим пользовательский интерфейс, который позволит пользователю щелкнуть на веб-элементе управления `Button` (по имени `btnGetBrowserStats`) для просмотра разнообразных статистических данных о вызывающем браузере. Статистические сведения будут генерироваться динамически, после чего присоединяться к элементу управления `Label` (с именем `lblOutput`). Добавьте эти два элемента управления в любое место графического конструктора веб-страницы. Затем реализуйте код обработчика события `Click` для кнопки, как показано ниже:

```
protected void btnGetBrowserStats_Click(object sender, EventArgs e)
{
    string theInfo = "";
    theInfo += string.Format("<li>Is the client AOL? {0}</li>",
        Request.Browser.AOL); // Клиент AOL?
    theInfo += string.Format("<li>Does the client support ActiveX? {0}</li>",
        Request.Browser.ActiveXControls); // Поддерживает ли ActiveX?
    theInfo += string.Format("<li>Is the client a Beta? {0}</li>",
        Request.Browser.Beta); // Бета-версия?
    theInfo += string.Format("<li>Does the client support Java Applets? {0}</li>",
        Request.Browser.JavaApplets); // Поддерживает ли Java-апплеты?
    theInfo += string.Format("<li>Does the client support Cookies? {0}</li>",
        Request.Browser.Cookies); // Поддерживает ли cookie-наборы?
    theInfo += string.Format("<li>Does the client support VBScript? {0}</li>",
        Request.Browser.VBScript); // Поддерживает ли VBScript?
    lblOutput.Text = theInfo;
}
```

Здесь проверяется несколько возможностей браузера. Как и можно было предположить, очень полезно проверить возможность поддержки браузером элементов `ActiveX`, `Java`-апплетов и кода `VBScript` клиентской стороны. Если вызывающий браузер не поддерживает некоторую веб-технология, то `.aspx`-страница сможет предпринять альтернативные действия.

Доступ к входным данным формы

В классе `HttpResponse` определены также свойства `Form` и `QueryString`. Они позволяют просматривать входные данные формы в виде пар “имя/значение”. Хотя свойства `HttpRequest.Form` и `HttpRequest.QueryString` можно было бы использовать для доступа к клиентским данным формы на веб-сервере, ASP.NET предлагает для этого более элегантный объектно-ориентированный подход. Поскольку ASP.NET предоставляет веб-элементы управления серверной стороны, HTML-элементы пользовательского интерфейса можно трактовать как настоящие объекты. Таким образом, вместо того чтобы получать значение из текстового поля следующим образом:

```
protected void btnGetFormData_Click(object sender, System.EventArgs e)
{
    // Получить значение из виджета с идентификатором txtFirstName.
    string firstName = Request.Form("txtFirstName");
    // Использовать полученное значение на странице...
}
```

можно просто опросить виджет серверной стороны непосредственно через свойство `Text`, как показано ниже:

```
protected void btnGetFormData_Click(object sender, System.EventArgs e)
{
    // Получить значение из виджета с идентификатором txtFirstName.
    string firstName = txtFirstName.Text;
    // Использовать полученное значение на странице...
}
```

Этот подход не только соответствует основополагающим принципам объектно-ориентированного программирования, но также позволяет не беспокоиться о способе отправки данных формы (GET или POST) перед получением значений. Более того, работа с виджетом является безопасной в отношении типов, учитывая, что ошибки обнаруживаются на этапе компиляции, а не во время выполнения. Разумеется, это не значит, что никогда не следует использовать свойство `Form` или `QueryString` в ASP.NET; просто теперь потребность в них значительно снижена и не является обязательной.

Свойство `IsPostBack`

Еще одним очень важным членом `HttpRequest` является свойство `IsPostBack`. Вспомните, что означает обратная отправка: веб-страница отправляет данные обратно на веб-сервер по тому же самому URL. Учитывая это, свойство `IsPostBack` вернет `true`, если текущий HTTP-запрос был послан пользователем в текущем сеансе, и `false`, если пользователь обращается к странице впервые.

Необходимость в определении того, является ли текущий HTTP-запрос обратной отправкой, чаще всего возникает, когда нужно организовать выполнение блока кода, только когда доступ к странице производится в первый раз. Например, при первом обращении пользователя к `.aspx`-файлу может потребоваться заполнить данными ADO.NET-объект `DataSet` и кешировать этот объект для последующего использования. Когда вызывающий клиент вернется на ту же страницу, можно избежать излишнего обращения к базе данных (конечно, некоторые страницы могут требовать обновления `DataSet` в каждом запросе, но это другая проблема). Если `.aspx`-файл обрабатывает событие страницы `Load` (рассматривается далее в этой главе), то запрограммировать проверку условия обратной отправки можно следующим образом:

```
protected void Page_Load(object sender, EventArgs e)
{
    // Заполнить DataSet только при самом первом входе пользователя на страницу.
    if (!IsPostBack)
    {
        // Заполнить объект DataSet и кешировать его.
    }
    // Использовать кешированный объект DataSet.
}
```

Взаимодействие с исходящим HTTP-ответом

Теперь, когда вы лучше понимаете, как тип Page позволяет взаимодействовать с входящим HTTP-запросом, необходимо научиться взаимодействовать с исходящим HTTP-ответом. В ASP.NET свойство Response класса Page предоставляет доступ к экземпляру типа HttpResponse. В этом типе определен набор свойств, которые позволяют форматировать HTTP-ответ, отправляемый обратно клиентскому браузеру. В табл. 32.5 перечислены основные свойства HttpResponse.

Таблица 32.5. Свойства типа HttpResponse

Свойство	Описание
Cache	Возвращает семантику кеширования веб-страницы (см. главу 34)
ContentEncoding	Получает или устанавливает набор символов HTTP для выходного потока
ContentType	Получает или устанавливает MIME-тип HTTP для выходного потока
Cookies	Получает коллекцию HttpCookie, которая будет возвращена браузеру
Output	Позволяет осуществлять текстовый вывод в тело исходящего HTTP-ответа
OutputStream	Позволяет осуществлять двоичный вывод в тело исходящего HTTP-ответа
StatusCode	Получает или устанавливает код состояния HTTP выходных данных, возвращаемых клиенту
StatusDescription	Получает или устанавливает строку состояния HTTP выходных данных, возвращаемых клиенту
SuppressContent	Получает или устанавливает значение, указывающее, что HTTP-ответ не будет отправлен клиенту

В табл. 32.6 приведен неполный список методов, поддерживаемых типом HttpResponse.

Таблица 32.6. Методы типа HttpResponse

Метод	Описание
Clear()	Очищает все заголовки и выходное содержимое из буфера потока
End()	Отправляет весь буферизованный вывод клиенту, после чего закрывает сокетное подключение
Flush()	Отправляет весь текущий буферизованный вывод клиенту
Redirect()	Перенаправляет клиента на новый URL
Write()	Записывает значения в выходной поток содержимого HTTP
WriteFile()	Записывает файл непосредственно в выходной поток содержимого HTTP

Выдача HTML-содержимого

Пожалуй, наиболее известным аспектом типа `HttpResponse` является способность записывать содержимое непосредственно в выходной поток HTTP.

Метод `HttpResponse.Write()` позволяет передавать в поток любые HTML-дескрипторы и/или текстовые литералы. Метод `HttpResponse.WriteFile()` еще более развивает эту функциональность, позволяя указывать имя физического файла на веб-сервере, содержимое которого необходимо поместить в выходной поток (это очень удобно для быстрой выдачи содержимого существующего .htm-файла).

В целях иллюстрации предположим, что в текущий .aspx-файл добавлен еще один элемент `Button` со следующим обработчиком события `Click` серверной стороны:

```
protected void btnHttpResponse_Click(object sender, EventArgs e)
{
    Response.Write("<b>My name is:</b><br>");
    Response.Write(this.ToString());
    Response.Write("<br><br><b>Here was your last request:</b><br>");
    Response.WriteFile("MyHTMLPage.htm");
}
```

Роль этой вспомогательной функции (которая, как несложно догадаться, вызывается некоторыми обработчиками событий на стороне сервера) довольно проста. Единственное, что здесь представляет интерес — это тот факт, что метод `HttpResponse.WriteFile()` теперь выдает содержимое .htm-файла, находящегося в корневом каталоге веб-сайта на сервере.

Хотя всегда можно воспользоваться старым подходом для генерации HTML-дескрипторов и содержимого с помощью метода `Write()`, в ASP.NET этот способ применяется намного реже, чем в классическом ASP. Причина (опять-таки) связана с появлением веб-элементов управления серверной стороны. Таким образом, чтобы визуализировать блок текстовых данных в браузере, достаточно присвоить нужную строку свойству `Text` виджета `Label`.

Перенаправление пользователей

Еще одним интересным аспектом класса `HttpResponse` является способность перенаправления пользователей на новый URL, например:

```
protected void btnWasteTime_Click(object sender, EventArgs e)
{
    Response.Redirect("http://www.facebook.com");
}
```

Если этот обработчик событий вызывается через обратную отправку клиентской стороны, пользователь автоматически перенаправляется по указанному URL.

На заметку! Метод `HttpResponse.Redirect()` всегда влечет за собой взаимодействие с клиентским браузером. Если нужно просто передать управление .aspx-файлу из того же самого виртуального каталога, более эффективным будет применение метода `HttpServerUtility.Transfer()`, доступного через унаследованное свойство `Server`.

Представленного материала вполне достаточно для оценки функциональности класса `System.Web.UI.Page`. Роль базового класса `System.Web.UI.Control` будет описана в следующей главе. А теперь рассмотрим жизненный цикл объекта, производного от `Page`.

Исходный код. Веб-сайт `FunWithPageMembers` доступен в подкаталоге `Chapter 32`.

Жизненный цикл веб-страницы ASP.NET

Каждая веб-страница ASP.NET имеет фиксированный жизненный цикл. Когда исполняющая среда ASP.NET принимает входящий запрос некоторого .aspx-файла, в памяти размещается соответствующий объект производного от System.Web.UI.Page типа с помощью стандартного конструктора этого типа. Затем инфраструктура автоматически запускает последовательность событий. По умолчанию автоматически учитывается событие Load, в обработчик которого можно поместить свой специальный код:

```
public partial class Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        Response.Write("Load event fired!");
    }
}
```

Помимо события Load, страница Page может перехватывать любое из основных событий, описанных в табл. 32.7, которые перечислены в порядке их появления (подробную информацию о событиях, которые могут возникать во время жизни страницы, ищите в документации .NET Framework 4.5 SDK).

Таблица 32.7. Избранные события типа Page

Событие	Описание
PreInit	Инфраструктура использует это событие для размещения в памяти всех веб-элементов управления, применения тем, установки мастер-страницы и настройки пользовательских профилей. Это событие можно перехватить, чтобы вмешаться в процесс
Init	Инфраструктура использует это событие для установки в свойствах веб-элементов управления их предыдущих значений через обратную отправку или данные состояния представления
Load	Когда возникает это событие, страница и ее элементы управления полностью инициализированы, и их предыдущие значения восстановлены. В этот момент можно безопасно взаимодействовать с каждым веб-виджетом
Событие, инициировавшее обратную отправку	Конечно, события с таким именем нет. Это "событие" просто означает любое событие, из-за которого браузер выполнил обратную отправку веб-серверу (вроде щелчка на элементе Button)
PreRender	Вся привязка данных для элементов управления и конфигурирование пользовательского интерфейса выполнено, и элементы управления готовы визуализировать свои данные в исходящий HTTP-ответ
Unload	Страница и ее элементы управления завершили процесс визуализации, и объект страницы готов к уничтожению. В этот момент попытка взаимодействия с исходящим HTTP-ответом приведет к ошибке времени выполнения. Однако это событие можно перехватить для выполнения любой очистки на уровне страницы (заккрытие файла или подключения к базе данных, занесение в журнал необходимых записей, освобождение памяти, занимаемой объектами, и т.д.)

Как ни удивительно, но IDE-среда не поддерживает обработку других событий помимо Load. В этом случае требуется вручную реализовать в файле кода метод Page_ИмяСобытия. Например, ниже показано, как можно обработать событие Unload:

```
public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        Response.Write("Load event fired!");
    }

    protected void Page_Unload(object sender, EventArgs e)
    {
        // Больше невозможно добавлять данные в HTTP-ответ,
        // поэтому будем записывать их в локальный файл.
        System.IO.File.WriteAllText(@"C:\MyLog.txt", "Page unloading!");
    }
}
```

На заметку! Каждое событие типа Page работает в сочетании с делегатом System.EventHandler, поэтому подпрограммы, которые обрабатывают эти события, всегда принимают Object в первом параметре и EventArgs — во втором.

Роль атрибута AutoEventWireup

Чтобы организовать обработку событий для своей страницы, необходимо добавить соответствующий обработчик в блок <script> или файл отделенного кода. Однако в директиве <%@ Page %> имеется специальный атрибут по имени AutoEventWireup, который по умолчанию установлен в true:

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeFile="Default.aspx.cs" Inherits="_Default" %>
```

При таком стандартном поведении каждый обработчик события уровня страницы будет автоматически добавляться при вводе метода с соответствующим именем. Если же установить атрибут AutoPageWireup в false:

```
<%@ Page Language="C#" AutoEventWireup="false"
    CodeFile="Default.aspx.cs" Inherits="_Default" %>
```

то события уровня страницы перехватываться не будут. Этот атрибут (будучи включенным) приводит к генерации необходимой оснастки для событий внутри автоматически сгенерированного частичного класса, который был описан ранее в этой главе. Если отключить AutoEventWireup, события уровня страницы все равно можно будет обрабатывать, используя логику обработки событий C#. Например:

```
public _Default()
{
    // Выполнить явную привязку к событиям Load и Unload.
    this.Load += Page_Load;
    this.Unload += Page_Unload;
}
```

Как и можно было ожидать, обычно атрибут AutoEventWireup остается включенным.

Событие Error

Еще одним событием, которое может произойти во время жизненного цикла страницы, является Error. Это событие возникает, когда метод унаследованного от Page типа инициирует исключение, которое не было явно обработано. Предположим, что обрабатывается событие Click для данного элемента Button на странице, и внутри обра-

ботчика события (который называется `btnGetFile_Click`) предпринимается попытка вывести в HTTP-ответ содержимое локального файла.

Также предположим, что проверка существования этого файла с помощью стандартной структурированной обработки исключений *не была выполнена*. Если поместить в стандартный конструктор обработчик события `Error`, то появится последний шанс справиться с проблемой на этой странице, прежде чем конечный пользователь получит невнятное сообщение об ошибке. Взгляните на следующий код:

```
public partial class _Default : System.Web.UI.Page
{
    void Page_Error(object sender, EventArgs e)
    {
        Response.Clear();
        Response.Write("I am sorry...I can't find a required file.<br>");
        Response.Write(string.Format("The error was: <b>{0}</b>",
            Server.GetLastError().Message));
        Server.ClearError();
    }

    protected void Page_Load(object sender, EventArgs e)
    {
        Response.Write("Load event fired!");
    }

    protected void Page_Unload(object sender, EventArgs e)
    {
        // Больше невозможно добавлять данные в HTTP-ответ,
        // поэтому будем записывать их в локальный файл.
        System.IO.File.WriteAllText(@"C:\MyLog.txt", "Page unloading!");
    }

    protected void btnPostBack_Click(object sender, EventArgs e)
    {
        // Здесь ничего не происходит. Это нужно только
        // для обеспечения обратной отправки страницы.
    }

    protected void btnTriggerError_Click(object sender, EventArgs e)
    {
        System.IO.File.ReadAllText(@"C:\IDontExist.txt");
    }
}
```

Обратите внимание, что обработчик событий `Error` начинается с очистки любого содержимого, имеющегося в HTTP-ответе, и выдачи обобщенного сообщения об ошибке. Чтобы получить доступ к конкретному объекту `System.Exception`, можно воспользоваться методом `HttpServerUtility.GetLastError()`, который представлен унаследованным свойством `Server`:

```
Exception e = Server.GetLastError();
```

И, наконец, перед выходом из этого обобщенного обработчика ошибок явно вызывается метод `HttpServerUtility.ClearError()` через свойство `Server`. Это обязательно, поскольку информирует исполняющую среду, что обработка проблемы завершена, и дальнейшая обработка не требуется. Если вы забудете сделать это, конечному пользователю будет выдана страница с ошибкой времени выполнения.

К этому моменту вы должны хорошо понимать структуру страницы ASP.NET. Имея такую подготовку, можно приступить к изучению роли веб-элементов управления, тем и мастер-страниц ASP.NET — т.е. материала последующих глав. В завершение данной главы мы рассмотрим роль файла `web.config`.

Исходный код. Веб-сайт PageLifeCycle доступен в подкаталоге Chapter 32.

Роль файла `web.config`

По умолчанию все веб-приложения ASP.NET на C#, созданные в Visual Studio, автоматически снабжаются файлом `web.config`. Если возникла необходимость добавить этот файл к веб-сайту вручную (например, при работе с однофайловой моделью без создания веб-решения), выберите пункт меню Website⇒Add New Item (Веб-сайт⇒Добавить новый элемент). В любом случае в файл `web.config` можно добавлять параметры, управляющие поведением веб-приложения во время выполнения.

Вспомните, что во время рассмотрения сборок .NET (в главе 14) вы узнали, что клиентское приложение может использовать конфигурационный XML-файл, чтобы указать среде CLR, как она должна обрабатывать запросы привязки, осуществлять зондирование сборок и выполнять другие детали времени выполнения. То же самое верно и для веб-приложений ASP.NET, но с тем заметным отличием, что веб-ориентированные конфигурационные файлы всегда называются `web.config` (в отличие от конфигурационных файлов `*.exe`, которые называются по имени соответствующей клиентской исполняемой сборки).

Полная структура файла `web.config` довольно многословна. В табл. 32.8 приведен ряд наиболее интересных подэлементов, которые могут присутствовать в файле `web.config`.

Таблица 32.8. Избранные элементы файла `web.config`

Элемент	Описание
<code><appSettings></code>	Этот элемент служит для установки специальных пар “имя/значение”, которые можно программно прочитать в память для использования страницами, с помощью типа <code>ConfigurationManager</code>
<code><authentication></code>	Этот элемент, относящийся к безопасности, используется для определения режима аутентификации для данного веб-приложения
<code><authorization></code>	Этот элемент, также связанный с безопасностью, используется для определения, какие пользователи и к каким ресурсам имеют доступ на веб-сервере
<code><connectionStrings></code>	Этот элемент служит для хранения строк внешних подключений, используемых веб-сайтом
<code><customErrors></code>	Этот элемент позволяет точно указать исполняющей среде, как следует сообщать об ошибках, возникающих во время функционирования веб-приложения
<code><globalization></code>	Этот элемент используется для конфигурирования параметров глобализации для веб-приложения
<code><namespaces></code>	Этот элемент содержит список всех пространств имен, которые нужно включить, если веб-приложение предварительно компилируется новым инструментом командной строки <code>aspnet_compiler.exe</code>
<code><sessionState></code>	Этот элемент используется для управления тем, как и где данные о состоянии сеанса будут храниться исполняющей средой .NET
<code><trace></code>	Этот элемент позволяет включать и отключать поддержку трассировки для данного веб-приложения

Помимо набора, представленного в табл. 32.8, файл `web.config` может содержать дополнительные подэлементы. Подавляющее большинство этих элементов относятся к безопасности, а остальные могут пригодиться только в более сложных сценариях применения ASP.NET, таких как создание специальных HTTP-заголовков или специальных модулей HTTP (эти темы здесь не рассматриваются).

Утилита администрирования веб-сайтов ASP.NET

Хотя содержимое файла `web.config` можно модифицировать непосредственно в Visual Studio, для веб-проектов ASP.NET доступен удобный веб-ориентированный редактор, который позволяет графически редактировать многочисленные элементы и атрибуты файла `web.config`. Для запуска этого инструмента выберите пункт меню Website⇒ASP.NET Configuration (Веб-сайт⇒Конфигурация ASP.NET).

Просмотрев содержимое вкладок в верхней части страницы, легко заметить, что большинство функциональности инструмента в основном предназначено для установки параметров безопасности веб-сайта. Однако этот инструмент позволяет также добавлять параметры к элементу `<appSettings>`, определять параметры отладки и трассировки, а также устанавливать стандартную страницу с сообщением об ошибке.

Вы еще увидите этот инструмент в действии, когда он понадобится, а сейчас учтите, что данная утилита не позволяет добавлять в файл `web.config` все возможные параметры. Почти наверняка будут возникать ситуации, когда этот файл придется изменять вручную в обычном текстовом редакторе.

Резюме

Построение веб-приложения требует другого мышления по сравнению с созданием традиционных настольных приложений. В этой главе вы кратко ознакомились с некоторыми основными темами, в числе которых HTML, HTTP, роль сценариев клиентской стороны, а также серверных сценариев, использующих классический ASP. Большая часть главы была посвящена исследованию архитектуры страницы ASP.NET. Как вы видели, каждый файл `*.aspx` в проекте имеет связанный с ним класс, производный от `System.Web.UI.Page`. С помощью этого объектно-ориентированного подхода ASP.NET позволяет создавать многократно используемые объектно-ориентированные системы.

После рассмотрения ключевой функциональности, предлагаемой цепочкой наследования страницы, в этой главе было показано, что страницы в конечном итоге компилируются в сборку .NET. Глава завершилась рассмотрением роли файла `web.config` и инструмента администрирования веб-сайтов ASP.NET.

Веб-элементы управления, мастер-страницы и темы ASP.NET

В предыдущей главе основное внимание было сосредоточено на общей структуре веб-страницы ASP.NET и роли класса `Page`. В этой главе мы погрузимся в детали *веб-элементов управления*, которые составляют пользовательский интерфейс страницы. После рассмотрения общей природы веб-элемента управления ASP.NET вы получите представление о том, как правильно применять многие элементы пользовательского интерфейса, включая элементы управления проверкой достоверности и разнообразные приемы привязки данных.

Значительная часть этой главы будет посвящена роли *мастер-страниц*. Будет показано, как они обеспечивают упрощенный способ определения общего скелета пользовательского интерфейса, который повторяется среди множества страниц веб-сайта. С мастер-страницами тесно связано применение элементов управления навигацией по сайту (а также файл `*.sitemap`), которые позволяют определять навигационную структуру многостраничного сайта с помощью XML-файла на стороне сервера.

В завершение вы узнаете о роли тем ASP.NET. Концептуально темы служат той же цели, что и каскадные таблицы стилей (CSS); однако темы ASP.NET применяются на веб-сервере (в противоположность клиентскому браузеру) и потому имеют доступ к ресурсам серверной стороны.

Природа веб-элементов управления

Главным преимуществом ASP.NET является возможность собирать пользовательский интерфейс страниц с использованием типов, определенных в пространстве имен `System.Web.UI.WebControls`. Как было показано, эти элементы управления (которые называются *серверными элементами управления*, *веб-элементами управления* или *элементами управления Web Forms*) исключительно полезны в том, что автоматически генерируют необходимую HTML-разметку для запрашивающего браузера и предоставляют набор событий, которые могут быть обработаны на веб-сервере. Более того, поскольку каждый элемент управления ASP.NET имеет соответствующий класс в пространстве имен `System.Web.UI.WebControls`, им можно манипулировать в объектно-ориентированной манере.

При конфигурировании свойств веб-элемента управления в окне Properties (Свойства) среды Visual Studio изменения фиксируются в открывающем дескрипторе данного элемента в файле *.aspx как последовательность пар “имя/значение”. Таким образом, если добавить новый элемент TextBox в визуальном конструкторе и изменить его свойства ID, BorderStyle, BorderWidth, BackColor и Text, открывающий дескриптор <asp:TextBox> будет соответствующим образом модифицирован (обратите внимание, что значение Text становится внутренним текстом в области TextBox):

```
<asp:TextBox ID="txtNameTextBox" runat="server" BackColor="#C0FFC0"
    BorderStyle="Dotted" BorderWidth="3px">Enter Your Name</asp:TextBox>
```

Учитывая, что это объявление веб-элемента управления в конечном итоге становится переменной-членом из пространства имен System.Web.UI.WebControls (через цикл динамической компиляции, упомянутый в главе 32), с членами этого типа можно взаимодействовать внутри блока <script> серверной стороны или более распространенным способом — в файле отделенного кода. Таким образом, если добавить в файл *.aspx новый элемент управления Button, можно написать обработчик события Click серверной стороны, в котором будет изменяться цвет фона элемента TextBox, как показано ниже:

```
partial class _Default : System.Web.UI.Page
{
    protected void btnChangeTextBoxColor_Click(object sender, EventArgs e)
    {
        // Изменить цвет фона объекта TextBox в коде.
        this.txtNameTextBox.BackColor = System.Drawing.Color.DarkBlue;
    }
}
```

Все веб-элементы управления ASP.NET в конечном итоге являются производными от общего базового класса System.Web.UI.WebControls.WebControl, который, в свою очередь, унаследован от System.Web.UI.Control (а тот — от System.Object). Классы Control и WebControl определяют набор свойств, общих для всех элементов управления серверной стороны. Прежде чем рассматривать унаследованную функциональность, давайте формализуем понятие обработки событий серверной стороны.

Обработка событий серверной стороны

Учитывая текущее состояние World Wide Web, нельзя не принимать во внимание фундаментальную природу взаимодействия браузеров с веб-серверами. Всякий раз, когда эти две сущности взаимодействуют, возникает лишенный состояния цикл запроса/ответа HTTP. Хотя серверные элементы управления ASP.NET выполняют немалую работу, изолируя от низкоуровневых деталей протокола HTTP, всегда следует помнить, что восприятие World Wide Web как управляемой событиями сущности — всего лишь маскировка, которую обеспечивает платформа .NET. Это не имеет ничего общего с управляемой событиями моделью, которая поддерживается в основанной Windows инфраструктуре для построения графических пользовательских интерфейсов, такой как WPF.

Например, хотя в пространстве имен System.Windows.Controls из WPF и в пространстве имен System.Web.UI.WebControls из ASP.NET определены классы с одними и теми же простыми именами (Button, TextBox, Label и т.д.), они не предлагают одинаковые наборы свойств, методов или событий. Скажем, не существует способа обработать событие MouseMove серверной стороны, когда пользователь перемещает курсор над элементом управления Button из Web Forms.

Суть заключается в том, что конкретный веб-элемент управления ASP.NET будет открывать ограниченный набор событий, причем все они в конечном итоге приводят к

обратной отправке на веб-сервер. Любая необходимая обработка событий клиентской стороны потребует написания массы сценарного кода JavaScript/VBScript *клиентской стороны* для обработки механизмом выполнения сценариев запросившего браузера. Учитывая то, что ASP.NET — это в первую очередь технология серверной стороны, тема написания сценариев клиентской стороны здесь не рассматривается.

На заметку! Обработка события для определенного веб-элемента управления с использованием Visual Studio может быть выполнена в той же манере, что и для элемента управления графического пользовательского интерфейса Windows. Просто выберите необходимый виджет на поверхности визуального конструктора и щелкните на значке с изображением молнии в окне Properties.

Свойство AutoPostBack

Стоит также упомянуть, что многие веб-элементы управления ASP.NET поддерживают свойство по имени `AutoPostBack` (прежде всего, это элементы управления `CheckBox`, `RadioButton` и `TextBox`, а также любые элементы, унаследованные от абстрактного класса `ListControl`). По умолчанию упомянутое свойство установлено в `false`, что отключает немедленную обратную отправку на сервер (даже при наличии обработчика события в файле отделенного кода). В большинстве случаев это именно то поведение, которое нужно, учитывая, что такие элементы пользовательского интерфейса, как флажки, обычно не требуют функциональности обратной отправки. Другими словами, не следует выполнять обратную отправку немедленно после отметки или снятия отметки с флажка, поскольку объект страницы может получить состояние виджета внутри более естественного обработчика события `Click` для `Button`.

Однако если необходимо заставить любой из этих виджетов немедленно выполнять обратную отправку обработчику события серверной стороны, установите значение `AutoPostBack` равным `true`. Такой прием полезен, когда требуется сделать так, чтобы состояние одного виджета автоматически помещало значение в другой виджет на той же странице. Для иллюстрации предположим, что есть веб-страница, содержащая элемент `TextBox` (по имени `txtAutoPostBack`) и элемент `ListBox` (по имени `lstTextBoxData`). Ниже показана соответствующая разметка:

```
<form id="form1" runat="server">
  <asp:TextBox ID="txtAutoPostBack" runat="server"></asp:TextBox>
  <br/>
  <asp:ListBox ID="lstTextBoxData" runat="server"></asp:ListBox>
</form>
```

Теперь обработайте событие `TextChanged` элемента `TextBox`, и внутри обработчика событий серверной стороны попробуйте заполнить `ListBox` текущим значением `TextBox`:

```
partial class _Default : System.Web.UI.Page
{
    protected void txtAutoPostBack_TextChanged(object sender, EventArgs e)
    {
        lstTextBoxData.Items.Add(txtAutoPostBack.Text);
    }
}
```

Если вы запустите приложение в том виде, как есть, то при вводе текста в `TextBox` обнаружите, что ничего не происходит. Более того, если вы введете что-то в `TextBox` и перейдете к следующему элементу управления нажатием клавиши `<Tab>`, также ничего не произойдет.

Причина в том, что по умолчанию свойство `AutoPostBack` элемента `TextBox` установлено в `false`. Однако если установить это свойство в `true`:

```
<asp:TextBox ID="txtAutoPostBack" runat="server" AutoPostBack="true">
</asp:TextBox>
```

то при выходе из `TextBox` по нажатию `<Tab>` или `<Enter>` элемент `ListBox` автоматически заполнится текущим значением `TextBox`. Честно говоря, помимо необходимости наполнения элементов одного виджета в зависимости от значения другого, обычно изменять состояние свойства `AutoPostBack` виджета не придется (и даже в такой ситуации задачу можно решить внутри клиентского сценария, устраняя потребность во взаимодействии с сервером).

Базовые классы `Control` и `WebControl`

Базовый класс `System.Web.UI.Control` определяет разнообразные свойства, методы и события, которые обеспечивают возможность взаимодействия с основными (обычно не имеющими отношения к графическому пользовательскому интерфейсу) аспектами веб-элемента управления. В табл. 33.1 документированы некоторые члены, представляющие интерес.

Таблица 33.1. Избранные члены `System.Web.UI.Control`

Член	Описание
<code>Controls</code>	Это свойство получает объект <code>ControlCollection</code> , представляющий дочерние элементы управления внутри текущего элемента
<code>DataBind()</code>	Этот метод привязывает источник данных к вызывающему серверному элементу управления и всем его дочерним элементам
<code>EnableTheming</code>	Это свойство устанавливает, поддерживает ли элемент функциональность тем (стандартное значение — <code>true</code>)
<code>HasControls()</code>	Этот метод определяет, содержит ли серверный элемент управления какие-то дочерние элементы
<code>ID</code>	Это свойство получает и устанавливает программный идентификатор серверного элемента управления
<code>Page</code>	Это свойство получает ссылку на экземпляр <code>Page</code> , который содержит данный серверный элемент управления
<code>Parent</code>	Это свойство получает ссылку на родительский элемент данного серверного элемента управления в иерархии элементов управления страницы
<code>SkinID</code>	Это свойство получает или устанавливает обложку для применения к элементу управления, позволяя определять внешний вид и поведение с использованием ресурсов серверной стороны
<code>Visible</code>	Это свойство получает или устанавливает значение, указывающее на то, будет ли серверный элемент управления визуализироваться в виде элемента пользовательского интерфейса на странице

Перечисление содержащихся элементов управления

Первый аспект `System.Web.UI.Control`, который мы рассмотрим — это тот факт, что все веб-элементы управления (включая сам `Page`) наследуют коллекцию специальных элементов управления (доступную через свойство `Controls`).

Во многом подобно приложениям Windows Forms, свойство Controls предоставляет доступ к строго типизированной коллекции типов, производных от WebControl. Подобно любой коллекции .NET, она позволяет динамически добавлять, вставлять и удалять элементы во время выполнения.

Хотя формально возможно добавлять веб-элементы управления непосредственно к типу, производному от Page, проще (и надежнее) использовать элемент управления Panel. Класс Panel представляет контейнер виджетов, которые могут быть, а могут и не быть, видимыми конечному пользователю (в зависимости от значений свойств Visible и BorderStyle).

Для примера создайте новый проект ASP.NET Empty Web Site (Пустой веб-сайт ASP.NET) по имени DynamicCtrls и добавьте в проект новый элемент Web Form (Веб-форма). С помощью визуального конструктора страниц Visual Studio добавьте элемент типа Panel (с именем myPanel), который содержит в себе элементы TextBox, Button и HyperLink с произвольными именами (имейте в виду, что визуальный конструктор требует, чтобы внутренние элементы перетаскивались только в пределах пользовательского интерфейса элемента Panel). Затем за пределами Panel разместите виджет Label (по имени lblControlInfo) для помещения в него визуализированного вывода. Ниже показана одна из возможных HTML-разметок.

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Dynamic Control Test</title>
</head>
<body>
  <form id="form1" runat="server">
    <div>
      <hr />
      <h1>Dynamic Controls</h1>
      <asp:Label ID="lblTextBoxText" runat="server"></asp:Label>
      <hr />
    </div>

    <!-- Элемент Panel содержит три элемента управления -->
    <asp:Panel ID="myPanel" runat="server" Width="200px"
      BorderColor="Black" BorderStyle="Solid" >
      <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox><br/>
      <asp:Button ID="Button1" runat="server" Text="Button"/><br/>
      <asp:HyperLink ID="HyperLink1" runat="server">HyperLink
    </asp:HyperLink>
    </asp:Panel>
    <br />
  </form>
  <asp:Label ID="lblControlInfo" runat="server"></asp:Label>
</body>
</html>
```

При такой разметке поверхность визуального конструктора страницы будет выглядеть примерно так, как показано на рис. 33.1.

Предположим, что в обработчике события Page_Load() необходимо получить детальную информацию об элементах управления, содержащихся внутри Panel, и присвоить результат элементу управления Label (по имени lblControlInfo).

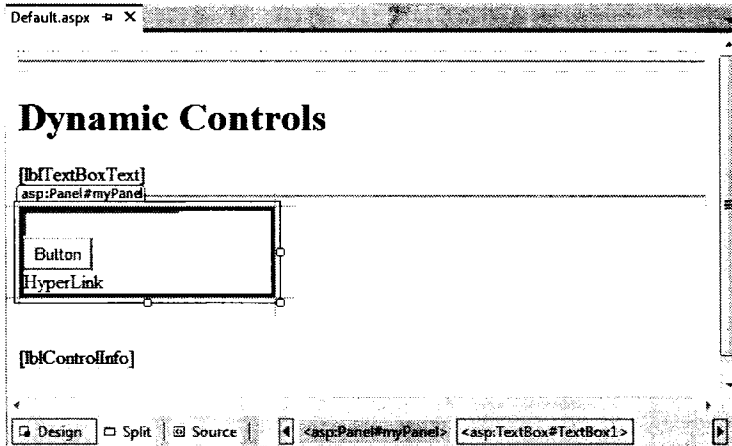


Рис. 33.1. Пользовательский интерфейс веб-страницы в проекте DynamicCtrls

Ниже приведен соответствующий код C#.

```
public partial class _Default : System.Web.UI.Page
{
    private void ListControlsInPanel()
    {
        string theInfo = "";
        theInfo = string.Format("<b>Does the panel have controls? {0}</b><br/>",
            myPanel.HasControls());

        // Получить все элементы управления в панели.
        foreach (Control c in myPanel.Controls)
        {
            if (!object.ReferenceEquals(c.GetType(),
                typeof(System.Web.UI.LiteralControl)))
            {
                theInfo += "*****<br/>";
                theInfo += string.Format("Control Name? {0}<br/>", c.ToString());
                theInfo += string.Format("ID? {0}<br>", c.ID);
                theInfo += string.Format("Control Visible? {0}<br/>", c.Visible);
                theInfo += string.Format("ViewState? {0}<br/>", c.EnableViewState);
            }
        }
        lblControlInfo.Text = theInfo;
    }

    protected void Page_Load(object sender, System.EventArgs e)
    {
        ListControlsInPanel();
    }
}
```

Здесь осуществляется проход по всем WebControl, содержащимся в Panel, с проверкой их принадлежности к типу System.Web.UI.LiteralControl; элементы этого типа пропускаются. Класс LiteralControl служит для представления литеральных дескрипторов и содержимого HTML (например,
, текстовые литералы и т.п.). Если не предпринять такой проверки, в контексте Panel будет найдено намного больше элементов управления (учитывая приведенное выше объявление *.aspx). Предполагая, что элемент управления не является литеральным HTML-содержимым, для него выводятся некоторые статистические данные. Результат можно видеть на рис. 33.2.

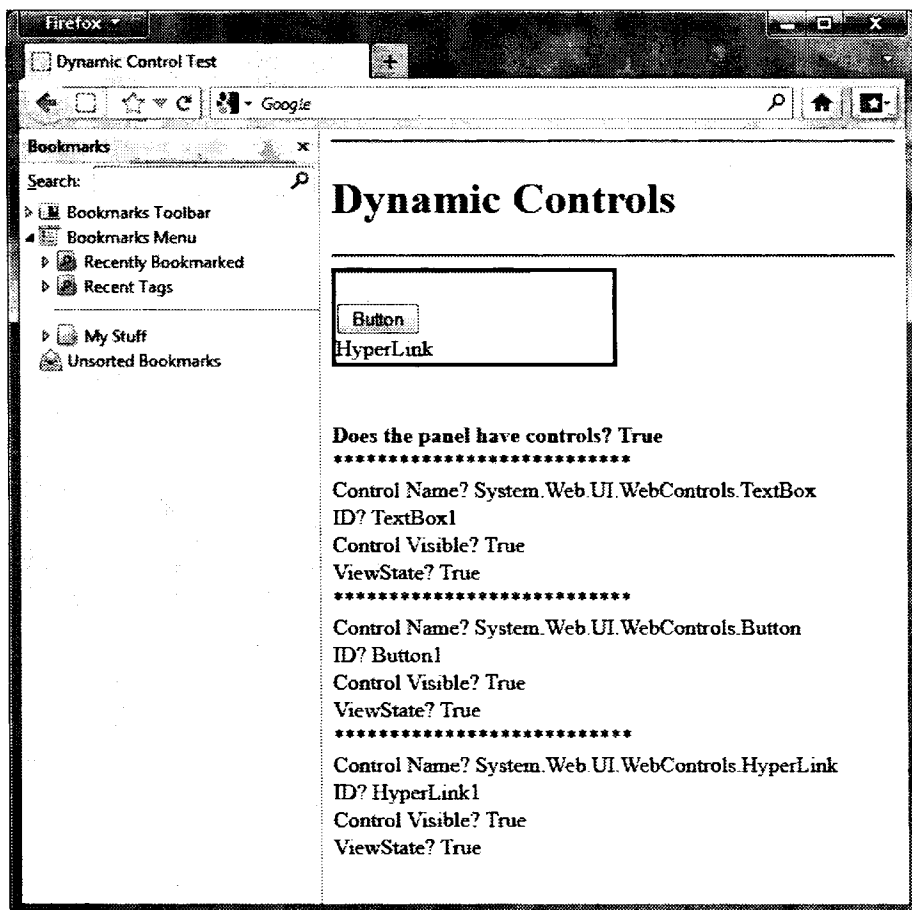


Рис. 33.2. Перечисление элементов управления во время выполнения

Динамическое добавление и удаление элементов управления

Пусть теперь необходимо изменять содержимое Panel во время выполнения. Давайте поместим на текущую страницу элемент Button (по имени `btnAddWidgets`), который будет динамически добавлять к Panel три новых элемента управления TextBox, и еще один элемент Button (по имени `btnRemovePanelItems`), который очистит Panel от всех вложенных элементов управления. Обработчики события Click для обеих кнопок показаны ниже.

```

protected void btnClearPanel_Click(object sender, System.EventArgs e)
{
    // Очистить содержимое панели, затем заново перечислить элементы.
    myPanel.Controls.Clear();
    ListControlsInPanel();
}

protected void btnAddWidgets_Click(object sender, System.EventArgs e)
{
    for (int i = 0; i < 3; i++)
    {
        // Присвоить идентификатор, чтобы можно было позднее получить
        // текстовое значение с использованием входных данных формы.
        TextBox t = new TextBox();
        t.ID = string.Format("newTextBox{0}", i);
        myPanel.Controls.Add(t);
        ListControlsInPanel();
    }
}

```


Обратите внимание, что каждому элементу `TextBox` присваивается уникальный идентификатор (`newTextBox0`, `newTextBox1` и т.д.). Запустив страницу, можно добавлять новые элементы к элементу управления `Panel` и полностью очищать содержимое `Panel`.

Взаимодействие с динамически созданными элементами управления

Получать значения из этих динамически сгенерированных элементов `TextBox` можно различными способами. Добавьте к пользовательскому интерфейсу еще один элемент управления `Button` (по имени `btnGetTextData`) и один элемент `Label` по имени `lblTextBoxData`, а также обработайте событие `Click` элемента `Button`.

Для доступа к данным в динамически сгенерированных элементах `TextBox` существует несколько вариантов. Один подход заключается в циклическом проходе по всем элементам, содержащимся во входных данных формы HTML (доступных через `HttpRequest.Form`) и накоплении текстовой информации в локальной строке `System.String`. После прохождения всей коллекции эту строку необходимо присвоить свойству `Text` нового элемента `Label`:

```
protected void btnGetTextData_Click(object sender, System.EventArgs e)
{
    string textBoxValues = "";
    for (int i = 0; i < Request.Form.Count; i++)
    {
        textBoxValues += string.Format("<li>{0}</li><br/>", Request.Form[i]);
    }
    lblTextBoxData.Text = textBoxValues;
}
```

Запустив приложение, вы обнаружите, что содержимое каждого текстового поля можно просматривать в виде довольно длинной (нечитабельной) строки. Эта строка содержит *состояние представления* каждого элемента управления на странице. Роль состояния представления рассматривается в главе 34.

Для получения более ясного вывода можно разнести текстовые данные по уникально именованным элементам (`newTextBox0`, `newTextBox1` и `newTextBox2`). Взгляните на следующую модификацию:

```
protected void btnGetTextData_Click(object sender, System.EventArgs e)
{
    // Получить текстовые поля по имени.
    string lableData = string.Format("<li>{0}</li><br/>",
        Request.Form.Get("newTextBox0"));
    lableData += string.Format("<li>{0}</li><br/>",
        Request.Form.Get("newTextBox1"));
    lableData += string.Format("<li>{0}</li><br/>",
        Request.Form.Get("newTextBox2"));
    lblTextBoxData.Text = lableData;
}
```

Используя этот подход, вы заметите, что как только запрос обработан, текстовое поле исчезнет. Причина этого кроется в не поддерживающей состояние природе HTML. Чтобы сохранять эти динамически созданные `TextBox` между обратными отправками, нужно пользоваться приемами программирования состояния ASP.NET (см. главу 34).

Функциональность базового класса WebControl

Как видите, тип `Control` обладает рядом поведенческих характеристик, не связанных с графическим пользовательским интерфейсом (коллекция элементов управления, поддержка автоматической обратной отправки и т.д.). С другой стороны, базовый класс `WebControl` предоставляет графический полиморфный интерфейс всем веб-виджетам (некоторые свойства `WebControl` перечислены в табл. 33.2).

Таблица 33.2. Избранные свойства базового класса `WebControl`

Свойство	Описание
<code>BackColor</code>	Получает или устанавливает цвет фона для веб-элемента управления
<code>BorderColor</code>	Получает или устанавливает цвет контура веб-элемента управления
<code>BorderStyle</code>	Получает или устанавливает стиль контура веб-элемента управления
<code>BorderWidth</code>	Получает или устанавливает ширину контура веб-элемента управления
<code>Enabled</code>	Получает или устанавливает значение, указывающее на то, что веб-элемент управления является доступным
<code>CssClass</code>	Позволяет назначить виджету класс, определенный в каскадной таблице стилей
<code>Font</code>	Получает информацию о шрифте веб-элемента управления
<code>ForeColor</code>	Получает или устанавливает цвет переднего плана (обычно цвет текста) веб-элемента управления
<code>Height, Width</code>	Получает или устанавливает высоту и ширину веб-элемента управления
<code>TabIndex</code>	Получает или устанавливает индекс обхода по клавише <code><Tab></code> веб-элемента управления
<code>ToolTip</code>	Получает или устанавливает всплывающую подсказку веб-элемента управления, появляющуюся при наведении на элемент курсора мыши

Почти все эти свойства самоочевидны, поэтому вместо того, чтобы рассматривать их использование по одиночке, давайте взглянем на работу сразу множества элементов `Web Forms`.

Основные категории веб-элементов управления ASP.NET

Библиотеку веб-элементов управления `ASP.NET` можно разделить на несколько обширных категорий, и все они видны в панели инструментов `Visual Studio` (предполагая, что в визуальном конструкторе открыта страница `*.aspx`), как показано на рис. 33.3.

В разделе `Standard` (Стандартные) панели инструментов находятся наиболее часто используемые элементы управления, включая `Button`, `Label`, `TextBox` и `ListBox`. В дополнение к этим замечательным элементам пользовательского интерфейса, в разделе `Standard` также доступны и более экзотические веб-элементы управления, такие как `Calendar`, `Wizard` и `AdRotator` (рис. 33.4).

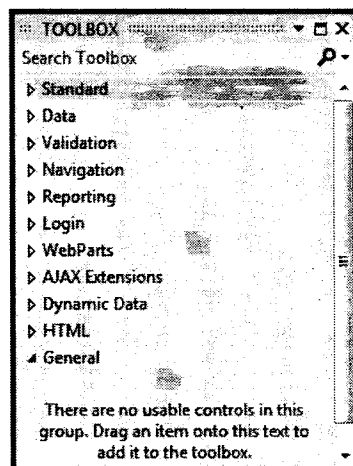


Рис. 33.3. Категории веб-элементов управления `ASP.NET`

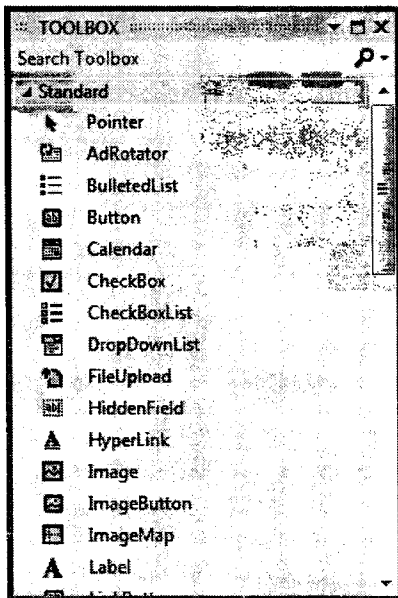


Рис. 33.4. Веб-элементы управления ASP.NET из раздела Standard

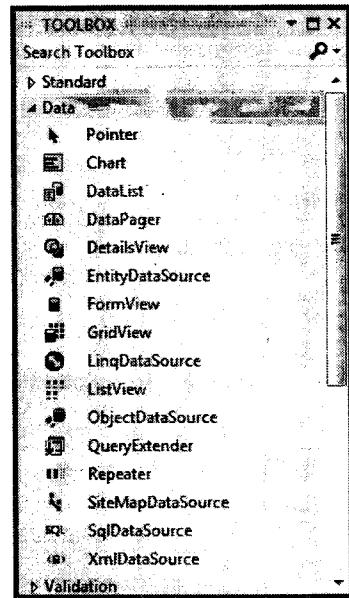


Рис. 33.5. Веб-элементы управления ASP.NET, ориентированные на данные

Раздел Data (Данные) — это место, где можно найти набор элементов управления, используемых для операций привязки данных, включая новый элемент управления ASP.NET под названием Chart, который позволяет визуализировать графики (круговые диаграммы, гистограммы и т.п.) обычно в качестве результата операции привязки (рис. 33.5).

Элементы управления проверкой достоверности ASP.NET (находящиеся в области Validation (Проверка достоверности) панели инструментов) очень интересны в том плане, что их можно конфигурировать для генерации блоков JavaScript-кода клиентской стороны, которые проверяют поля ввода на допустимость данных. Если случится ошибка проверки достоверности, пользователь увидит сообщение об ошибке и не сможет выполнить обратную отправку на сервер, пока не исправит ошибку.

Раздел Navigation (Навигация) панели инструментов — это место, где находится небольшой набор элементов управления (Menu, SiteMapPath и TreeView), которые обычно работают в сочетании с файлом *.sitemap. Как уже кратко упоминалось ранее в этой главе, элементы управления навигацией позволяют описывать структуру многостраничного сайта, используя дескрипторы XML.

По-настоящему экзотическим набором веб-элементов управления ASP.NET можно назвать элементы из раздела Login (Вход), показанные на рис. 33.6.

Эти элементы управления могут радикально упростить включение в веб-приложения базовых средств безопасности (восстановление пароля, экраны входа и т.п.). Фактически эти элементы управления настолько мощные, что даже динамически создают выделенную

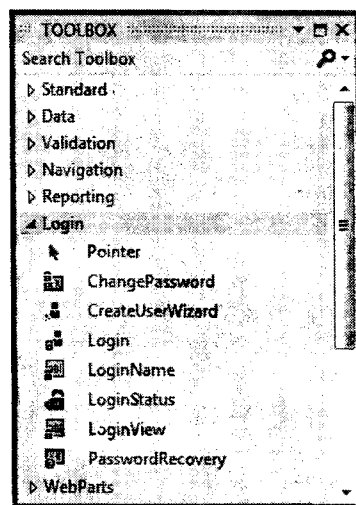


Рис. 33.6. Веб-элементы управления ASP.NET, связанные с безопасностью

базу данных для хранения регистрационных данных (в папке App_Data веб-сайта), если специфическая база данных для целей безопасности не предусмотрена.

На заметку! Остальные категории веб-элементов управления, представленные в панели инструментов Visual Studio (WebParts (Веб-части), AJAX Extensions (Расширения AJAX) и Dynamic Data (Динамические данные)), предназначены для решения более специализированных задач программирования и здесь не рассматриваются.

Несколько слов о пространстве имен `System.Web.UI.HtmlControls`

На самом деле в ASP.NET поставляются два разных набора инструментов веб-элементов управления. В дополнение к веб-элементам управления ASP.NET (внутри пространства имен `System.Web.UI.WebControls`) библиотеки базовых классов также предлагают элементы управления HTML в пространстве имен `System.Web.UI.HtmlControls`.

Элементы управления HTML — это коллекция типов, позволяющих использовать традиционные элементы управления HTML на странице Web Forms. Однако в отличие от простых HTML-дескрипторов, эти элементы являются объектно-ориентированными сущностями, которые могут быть сконфигурированы для запуска на сервере и потому поддерживают обработку событий серверной стороны. В отличие от веб-элементов управления ASP.NET, элементы HTML довольно просты по своей природе и предлагают очень небольшую функциональность сверх стандартных HTML-дескрипторов (`HtmlButton`, `HtmlInputControl`, `HtmlTable` и т.д.).

Элементы управления HTML могут быть удобны, если команда четко разделена на тех, кто занимается построением пользовательских интерфейсов HTML, и разработчиков .NET. Специалисты по HTML могут пользоваться своим веб-редактором, имея дело со знакомыми дескрипторами разметки и передавая готовые HTML-файлы команде разработки. После этого разработчики .NET могут конфигурировать эти элементы управления HTML для выполнения в качестве серверных элементов управления (щелкая правой кнопкой мыши на элементе управления HTML в Visual Studio). Это позволит разработчикам обрабатывать события серверной стороны и программно работать с виджетами HTML.

Элементы управления HTML предоставляют открытый интерфейс, который имитирует стандартные атрибуты HTML. Например, для получения информации из области ввода используется свойство `Value` вместо принятого у веб-элементов свойства `Text`. Учитывая, что элементы управления HTML не столь многофункциональны, как веб-элементы управления ASP.NET, далее в этой книге они не рассматриваются.

Документация по веб-элементам управления

На протяжении оставшейся части книги у вас будет шанс поработать с множеством веб-элементов управления ASP.NET; однако вы определенно должны уделить время ознакомлению с описанием пространства имен `System.Web.UI.WebControls` в документации .NET Framework 4.5 SDK. Там вы найдете объяснения и примеры кода для каждого члена пространства имен (рис. 33.7).

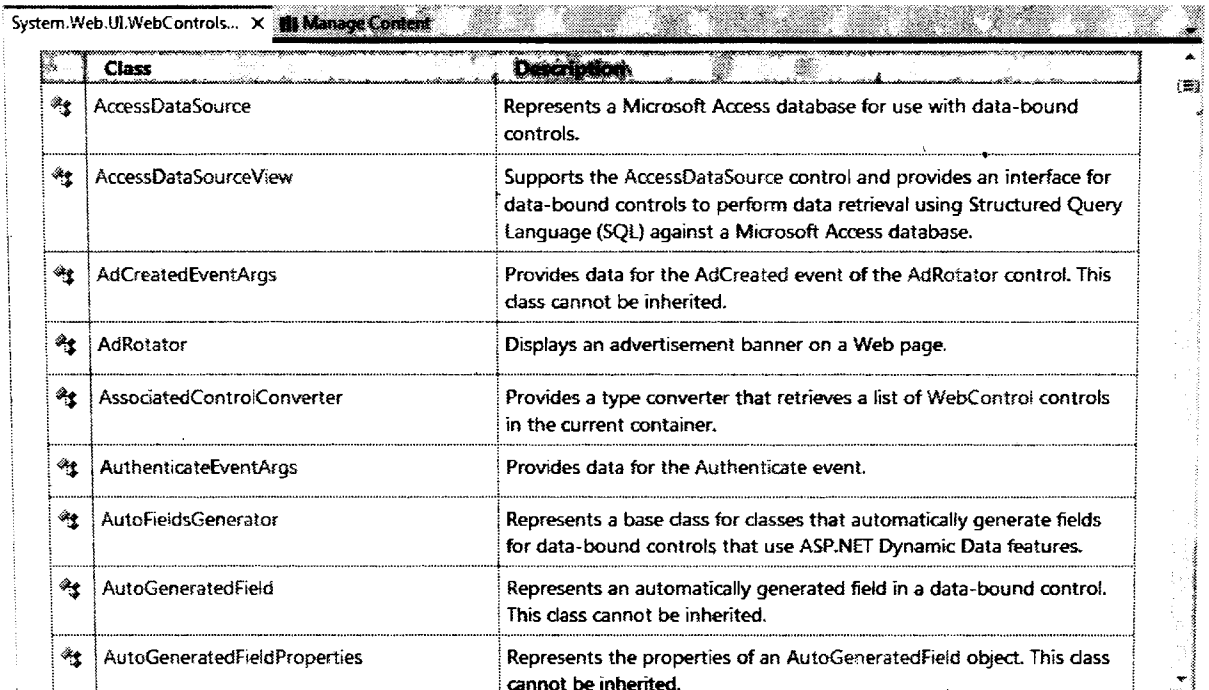
Построение примера веб-сайта ASP.NET

Учитывая, что очень много “простых” элементов управления выглядят и ведут себя подобно своим аналогам из графического пользовательского интерфейса Windows, детали базовых виджетов (`Button`, `Label`, `TextBox` и т.д.) подробно рассматриваться не будут. Вместо этого давайте построим веб-сайт, в котором иллюстрируется работа с

некоторыми из наиболее экзотичных элементов управления, а также с моделью мастер-страниц ASP.NET и особенностями механизма привязки данных. В частности, в приведенном ниже примере будут продемонстрированы следующие приемы:

- работа с мастер-страницами;
- работа с навигацией посредством карты сайта;
- работа с элементом управления GridView;
- работа с элементом управления Wizard.

Для начала создайте проект ASP.NET Empty Web Site по имени AppNetCarsSite. Обратите внимание, что мы пока не создаем новый проект веб-сайта ASP.NET, поскольку в нем добавляется множество начальных файлов, которые пока еще не рассматривались. В текущем проекте все необходимое будет добавляться вручную.



	Class	Description
	AccessDataSource	Represents a Microsoft Access database for use with data-bound controls.
	AccessDataSourceView	Supports the AccessDataSource control and provides an interface for data-bound controls to perform data retrieval using Structured Query Language (SQL) against a Microsoft Access database.
	AdCreatedEventArgs	Provides data for the AdCreated event of the AdRotator control. This class cannot be inherited.
	AdRotator	Displays an advertisement banner on a Web page.
	AssociatedControlConverter	Provides a type converter that retrieves a list of WebControl controls in the current container.
	AuthenticateEventArgs	Provides data for the Authenticate event.
	AutoFieldsGenerator	Represents a base class for classes that automatically generate fields for data-bound controls that use ASP.NET Dynamic Data features.
	AutoGeneratedField	Represents an automatically generated field in a data-bound control. This class cannot be inherited.
	AutoGeneratedFieldProperties	Represents the properties of an AutoGeneratedField object. This class cannot be inherited.

Рис. 33.7. Все веб-элементы управления ASP.NET описаны в документации .NET Framework 4.5 SDK

Работа с мастер-страницами

Многие веб-сайты обеспечивают согласованный внешний вид и поведение, распространяющийся на множество страниц (общая система навигации с помощью меню, общее содержимое заголовков и нижних колонтитулов, логотип компании и т.п.). Мастер-страница — это всего лишь страница ASP.NET, имеющая файловое расширение *.master. Сами по себе мастер-страницы не являются просматриваемыми в браузере клиентской стороны (на самом деле исполняющая среда ASP.NET не обслуживает веб-содержимое такого рода). Вместо этого мастер-страницы определяют общую компоновку пользовательского интерфейса, разделяемую всеми страницами (или их подмножеством) на сайте.

Кроме того, страница *.master определяет различные области-заполнители содержимого, которые устанавливают область пользовательского интерфейса, куда могут подключаться другие файлы *.aspx. Как будет показано, файлы *.aspx, которые включают свое содержимое в мастер-страницу, выглядят и ведут себя немного иначе, чем те

файлы *.aspx, которые рассматривались до сих пор. В частности, файлы *.aspx этого типа называются *страницами содержимого*. Страницы содержимого — это файлы *.aspx, в которых не определен HTML-элемент <form> (это работа мастер-страницы).

Однако с точки зрения конечного пользователя запрос осуществляется к заданному файлу *.aspx. На веб-сервере соответствующие файлы *.master и *.aspx смешиваются вместе — в единое объявление унифицированной HTML-страницы.

Чтобы проиллюстрировать использование мастер-страниц и страниц содержимого, начните со вставки новой мастер-страницы в веб-сайт через пункт меню Website⇒Add New Item (Веб-сайт⇒Добавить новый элемент); результирующее диалоговое окно показано на рис. 33.8.

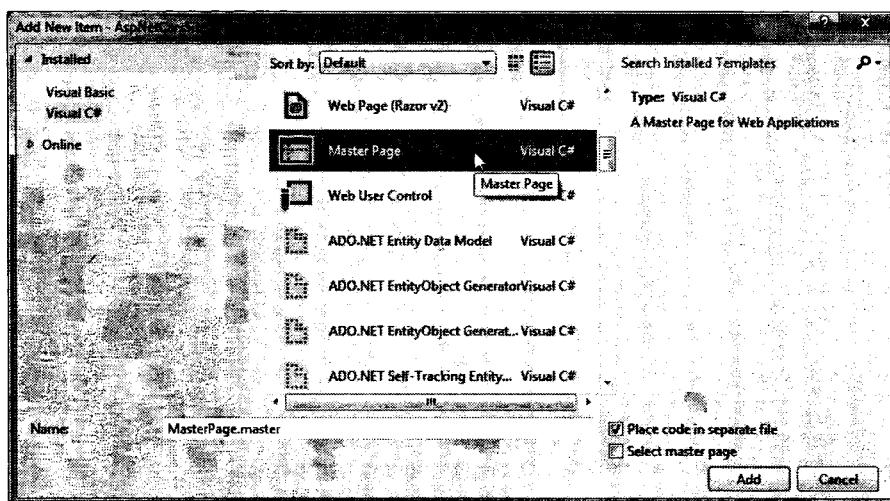


Рис. 33.8. Вставка нового файла *.master

Начальная разметка файла MasterPage.master выглядит следующим образом:

```
<%@ Master Language="C#" AutoEventWireup="true"
    CodeFile="MasterPage.master.cs" Inherits="MasterPage" %>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title> </title>
    <asp:ContentPlaceHolder id="head" runat="server">
    </asp:ContentPlaceHolder>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <asp:ContentPlaceHolder id="ContentPlaceHolder1" runat="server">
        </asp:ContentPlaceHolder>
    </div>
    </form>
</body>
</html>
```

Первое, что здесь представляет интерес — новая директива <%@ Master %>. В основном эта директива поддерживает те же атрибуты, что и директива <%@ Page %>, описанная в главе 32. Подобно типам Page, мастер-страница является производной от определенного базового класса, которым в данном случае является MasterPage. Открыв соответствующий файл кода, вы увидите там следующее определение класса:

```
public partial class MasterPage : System.Web.UI.MasterPage
{
    protected void Page_Load(object sender, EventArgs e)
    {
    }
}
```

Другим интересным моментом в разметке мастер-страницы является определение `<asp:ContentPlaceHolder>`. В эту область мастер-страницы могут подключаться виджеты пользовательского интерфейса связанного файла содержимого `*.aspx`, а не содержимое, определенное самой мастер-страницей.

Если вы намерены подключить файл `*.aspx` к этой области, контекст внутри дескрипторов `<asp:ContentPlaceHolder>` и `</asp:ContentPlaceHolder>` обычно оставляется пустым. Тем не менее, эту область можно заполнить разнообразными веб-элементами управления, которые функционируют как стандартный пользовательский интерфейс, если заданный файл `*.aspx` не предоставит специфическое содержимое. В данном примере предположим, что каждая страница `*.aspx` сайта действительно будет предоставлять специальное содержимое, и потому элементы `<asp:ContentPlaceHolder>` будут пустыми.

На заметку! В странице `*.master` может быть определено произвольное количество заполнителей содержимого. Кроме того, одна страница `*.master` может иметь вложенные страницы `*.master`.

Общий пользовательский интерфейс файла `*.master` можно строить с помощью тех же визуальных конструкторов Visual Studio, которые используются для создания файлов `*.aspx`. Для данного сайта будет добавлен описательный элемент `Label` (показывающий в качестве общего приветственного сообщения), элемент управления `AdRotator` (показывающий случайно выбранное одно из двух изображений) и элемент управления `TreeView` (позволяющий пользователю выполнять навигацию на другие области сайта). Ниже приведена возможная разметка мастер-страницы.

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title> </title>
    <asp:ContentPlaceHolder id="head" runat="server">
    </asp:ContentPlaceHolder>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <hr />
            <asp:Label ID="Label1" runat="server" Font-Size="XX-Large"
                Text="Welcome to the ASP.NET Cars Super Site!"></asp:Label>
            <asp:AdRotator ID="myAdRotator" runat="server"/>
            &nbsp;  <br />
            <br />
            <asp:TreeView ID="navigationTree" runat="server">
            </asp:TreeView>
            <hr />
        </div>
        <div>
            <asp:ContentPlaceHolder id="ContentPlaceHolder1" runat="server">
            </asp:ContentPlaceHolder>
        </div>
    </form>
</body>
</html>
```

На рис. 33.9 показано представление текущей мастер-страницы во время проектирования (обратите внимание, что область отображения элемента управления AdRotator на данный момент пуста).

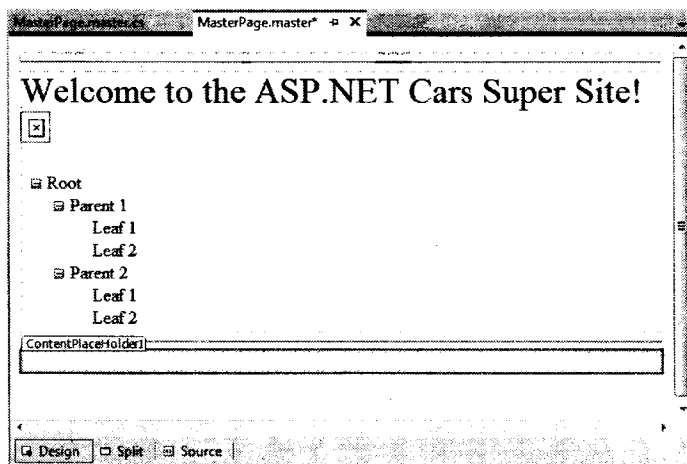


Рис. 33.9. Разделяемый пользовательский интерфейс в файле *.master

Внешний вид элемента управления TreeView можно улучшить, используя встроенный редактор элемента управления и выбрав ссылку Auto Format... (Автоформат...). Кроме того, с помощью окна Properties можно скорректировать отображение остальных элементов управления. После получения удовлетворительных результатов переходите к следующему разделу.

Конфигурирование логики навигации по сайту с помощью элемента управления TreeView

Инфраструктура ASP.NET поставляется с несколькими веб-элементами управления, поддерживающими навигацию по сайту: SiteMapPath, TreeView и Menu. Как и можно было предположить, указанные веб-виджеты могут быть сконфигурированы различными способами. Например, каждый из этих элементов управления может динамически генерировать свои узлы с применением внешнего XML-файла (или файла *.sitemap на основе XML), программно в коде или через разметку с использованием визуальных конструкторов среды Visual Studio.

Создаваемая система навигации будет динамически наполняться через файл *.sitemap. Преимущество такого подхода состоит в том, что можно определить общую структуру веб-сайта во внешнем файле и затем привязать его к элементу управления TreeView (или Menu) на лету. Таким образом, если навигационная структура веб-сайта изменится, достаточно будет просто модифицировать файл *.sitemap и перегрузить страницу. Для начала вставьте новый файл Web.sitemap в проект, выбрав пункт меню Website⇒Add New Item, в результате чего откроется диалоговое окно, показанное на рис. 33.10. Как видите, в начальном содержимом файла Web.sitemap определяется элемент самого верхнего уровня с двумя подузлами:

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >
  <siteMapNode url="" title="" description="">
    <siteMapNode url="" title="" description="" />
    <siteMapNode url="" title="" description="" />
  </siteMapNode>
</siteMap>
```

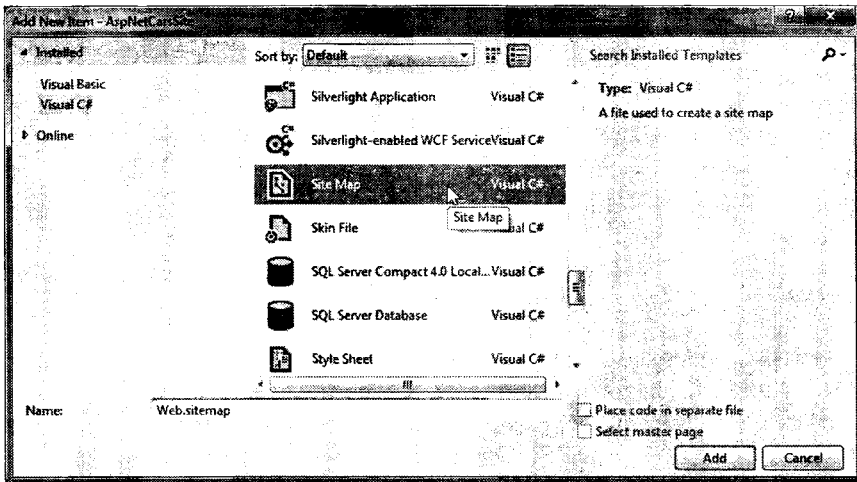



Рис. 33.10. Вставка нового файла Web.sitemap

Если привязать эту структуру к элементу управления Menu, отобразится элемент верхнего уровня с двумя подэлементами. Таким образом, для создания подэлементов необходимо просто определить новые элементы `<siteMapNode>` в контексте существующего `<siteMapNode>`. В любом случае цель заключается в определении внутри файла Web.sitemap общей структуры веб-сайта с использованием множества элементов `<siteMapNode>`. Каждый из этих элементов может определять атрибуты заголовка и URL. Атрибут URL представляет файл *.aspx, к которому будет выполнен переход, когда пользователь щелкнет на заданном элементе (или на узле TreeView). Создаваемая карта сайта будет содержать три узла (расположенные ниже узла верхнего уровня самой карты сайта):

- Home (Домой): Default.aspx
- Build a Car (Собрать автомобиль): BuildCar.aspx
- View Inventory (Просмотреть склад): Inventory.aspx

Эти три новых веб-страницы ASP.NET будут вскоре добавлены в проект. А пока нужно просто сконфигурировать файл карты сайта.

Система навигации имеет единственный элемент верхнего уровня Welcome (Добро пожаловать) с тремя подэлементами. Модифицируйте файл Web.sitemap следующим образом (имейте в виду, что каждое значение url должно быть уникальным, иначе возникнет ошибка времени выполнения):

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >
  <siteMapNode url="" title="Welcome!" description="">
    <siteMapNode url="~/Default.aspx" title="Home"
      description="The Home Page" />
    <siteMapNode url="~/BuildCar.aspx" title="Build a car"
      description="Create your dream car" />
    <siteMapNode url="~/Inventory.aspx" title="View Inventory"
      description="See what is in stock" />
  </siteMapNode>
</siteMap>
```

На заметку! Префикс ~/ перед каждой страницей в атрибуте url — это обозначение корня веб-сайта.

Теперь, несмотря на то, что можно было подумать, файл `Web.sitemap` не ассоциируется непосредственно с элементами управления `Menu` или `TreeView`, используя заданное свойство. Вместо этого файл `*.master` или `*.aspx`, содержащий виджет пользовательского интерфейса, который отобразит файл `Web.sitemap`, должен содержать компонент `SiteMapDataSource`. Этот компонент будет автоматически загружать файл `Web.sitemap` в свою объектную модель при запросе страницы. Типы `Menu` и `TreeView` затем установят свои свойства `DataSourceID` так, чтобы указывать на экземпляры `SiteMapDataSource`.

Для добавления нового компонента `SiteMapDataSource` в файл `*.master` и автоматической установки свойства `DataSourceID` можно использовать визуальный конструктор Visual Studio. Откройте встроенный редактор элемента управления `TreeView` (щелкнув на небольшой стрелке в правом верхнем углу элемента `TreeView`), раскройте список `Choose Data Source` (Выберите источник данных) и выберите пункт `<New Data Source...>` (Новый источник данных), как показано на рис. 33.11.

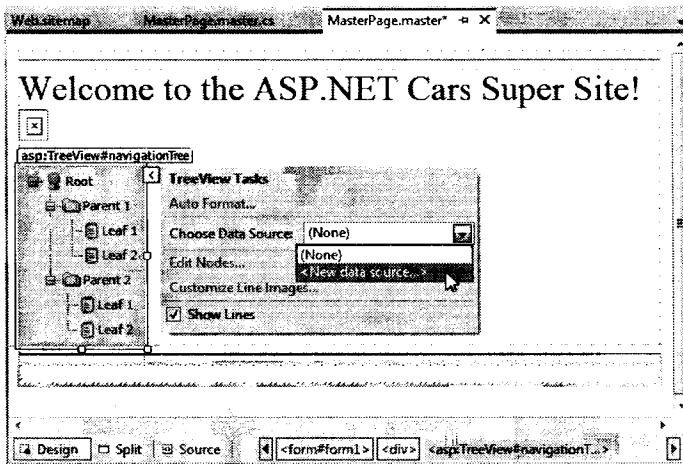


Рис. 33.11. Добавление нового компонента `SiteMapDataSource`

В результирующем диалоговом окне выберите значок `SiteMap` (Карта сайта). В результате будет установлено свойство `DataSourceID` элемента управления `Menu` или `TreeView`, и к странице добавится компонент `SiteMapDataSource`. Это все, что понадобится сделать, чтобы сконфигурировать элемент управления `TreeView` для перехода на дополнительные страницы внутри сайта. Если необходимо выполнить дополнительную обработку при выборе пользователем пункта меню, это можно сделать, обработав событие `SelectedNodeChanged` элемента управления `TreeView`. В рассматриваемом примере в этом нет необходимости, но имейте в виду, что можно выяснить, какой пункт меню был выбран, используя входные аргументы события.

Установка навигационных цепочек с помощью элемента `SiteMapPath`

Прежде чем переходить к работе с элементом управления `AddRotator`, добавьте элемент `SiteMapPath` (расположенный на вкладке `Navigation` (Навигация) панели инструментов) в файл `*.master` ниже элементов-заполнителей содержимого. Этот виджет автоматически настраивает свое содержимое на основе текущего выбора системы меню. Как вам, возможно, известно, это может предоставить полезную визуальную подсказку конечному пользователю (формально этот аспект пользовательского интерфейса называется навигационными цепочками или «хлебными крошками»). По завершении вы заметите, что при выборе пункта меню `Welcome` ⇒ `Build a Car` (Добро

пожаловать⇒Укомплектовать автомобиль) виджет SiteMapPath соответствующим образом автоматически обновится.

Конфигурирование элемента управления *AddRotator*

Роль виджета AdRotator из ASP.NET заключается в показе случайно выбранного изображения в одной и той же позиции браузера. Вспомните, что в данный момент виджет AdRotator отображается как пустой заполнитель. Этот элемент управления не может выполнять свою работу до тех пор, пока в его свойстве AdvertisementFile не будет указан исходный файл, описывающий каждое изображение. Для рассматриваемого примера источником данных будет простой XML-файл по имени Ads.xml.

Чтобы добавить XML-файл к веб-сайту, выберите пункт меню Website⇒Add New Item (Веб-сайт⇒Добавить новый элемент) и укажите вариант XML file (XML-файл). Назовите файл Ads.xml и предусмотрите уникальный элемент <Ad> для каждого изображения, которое планируется показывать. Как минимум, в элементе <Ad> должно быть указано графическое изображение для отображения (ImageUrl), URL для навигации, если изображение выбрано (TargetUrl), текст, появляющийся при наведении курсора мыши (AlternateText), и вес показа (Impressions):

```
<Advertisements>
  <Ad>
    <ImageUrl>SlugBug.jpg</ImageUrl>
    <TargetUrl>http://www.Cars.com</TargetUrl>
    <AlternateText>Your new Car?</AlternateText>
    <Impressions>80</Impressions>
  </Ad>
  <Ad>
    <ImageUrl>car.gif</ImageUrl>
    <TargetUrl>http://www.CarSuperSite.com</TargetUrl>
    <AlternateText>Like this Car?</AlternateText>
    <Impressions>80</Impressions>
  </Ad>
</Advertisements>
```

Здесь заданы два файла с изображениями (slugbug.jpg и car.gif). В результате необходимо обеспечить наличие этих файлов в корневом каталоге веб-сайта (данные файлы включены в состав загружаемого кода примеров для книги). Чтобы добавить их в текущий проект, выберите пункт меню Web Site⇒Add Existing Item (Веб-сайт⇒Добавить существующий элемент). В этот момент можно ассоциировать XML-файл с элементом управления AdRotator через свойство AdvertisementFile (в окне Properties):

```
<asp:AdRotator ID="myAdRotator" runat="server"
  AdvertisementFile="~/Ads.xml"/>
```

Позже, когда вы запустите приложение и выполните обратную отправку страницы, то получите случайно выбранный из двух файлов изображения.

Определение стандартной страницы содержимого Default.aspx

Теперь, имея готовую мастер-страницу, можно приступить к проектированию индивидуальных страниц *.aspx, которые будут определять содержимое пользовательского интерфейса для вставки в дескриптор <asp:ContentPlaceHolder> на мастер-странице. Файлы *.aspx, которые объединяются с мастер-страницей, называются *страницами содержимого* и имеют несколько ключевых отличий от нормальной автономной веб-страницы ASP.NET.

В своей основе файл *.master определяет раздел <form> финальной HTML-страницы. Поэтому существующая область <form> внутри файла *.aspx должна быть

заменена контекстом `<asp:Content>`. Хотя можно было бы модифицировать разметку в начальном файле `*.aspx` вручную, предпочтительнее вставить в проект новую страницу содержимого; для этого просто щелкните правой кнопкой мыши в любом месте на поверхности визуального конструктора файла `*.master` и выберите в контекстном меню пункт **Add Content Page** (Добавить страницу содержимого), как показано на рис. 33.12.

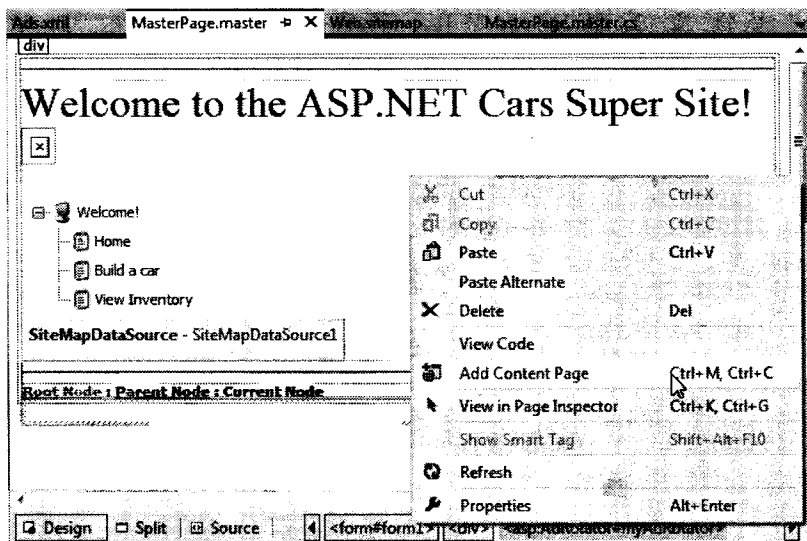


Рис. 33.12. Добавление новой страницы содержимого к мастер-странице

В результате будет сгенерирован новый файл `*.aspx` со следующей начальной разметкой:

```
<%@ Page Language="C#" MasterPageFile="~/MasterPage.master"
    AutoEventWireup="true" CodeFile="Default.aspx.cs"
    Inherits="_Default" Title="" %>

<asp:Content ID="Content1"
    ContentPlaceHolderID="head" Runat="Server">
</asp:Content>
<asp:Content ID="Content2"
    ContentPlaceHolderID="ContentPlaceHolder1" Runat="Server">
</asp:Content>
```

Первым делом, обратите внимание на то, что директива `<%@ Page %>` дополнена новым атрибутом `MasterPageFile`, который указывает на файл `*.master`. Кроме того, вместо элемента `<form>` имеется контекст `<asp:Content>` (пока пустой), в котором значение `ContentPlaceHolderID` установлено идентично компоненту `<asp:ContentPlaceHolder>` в файле мастер-страницы.

Имея эти ассоциации, страница содержимого знает, куда следует подключить ее содержимое, а содержимое мастер-страницы отображается в стиле только для чтения на странице содержимого. Нет необходимости в построении сложного пользовательского интерфейса для области содержимого `Default.aspx`. Для данного примера просто добавьте некоторый литеральный текст с базовыми инструкциями по сайту, как показано на рис. 33.13 (обратите внимание, что в правом верхнем углу страницы содержимого в визуальном конструкторе находится ссылка для переключения на связанную мастер-страницу).

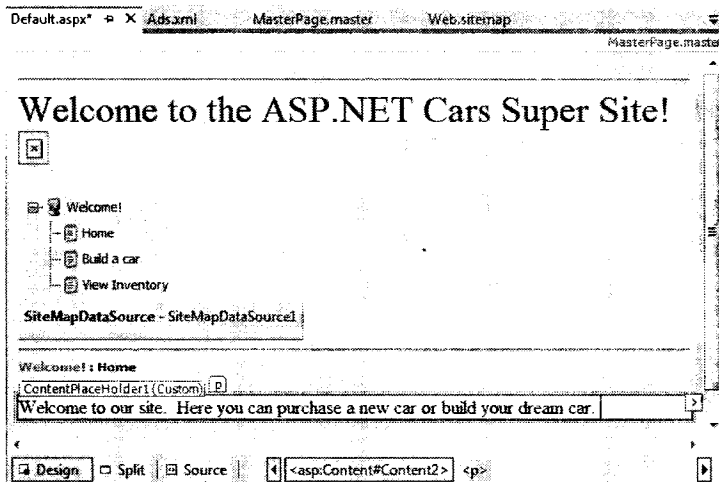


Рис. 33.13. Создание первой страницы содержимого

Теперь после запуска проекта вы увидите, что содержимое пользовательского интерфейса из файлов *.master и Default.aspx объединено в единый поток HTML-разметки. Как показано на рис. 33.14, браузер (или конечный пользователь) даже не имеет понятия о существовании мастер-страницы (обратите внимание, что браузер просто отображает HTML-разметку из Default.aspx). Кроме того, если обновить страницу (нажатием <F5>), элемент AdRotator покажет случайно выбранное одно из двух изображений.

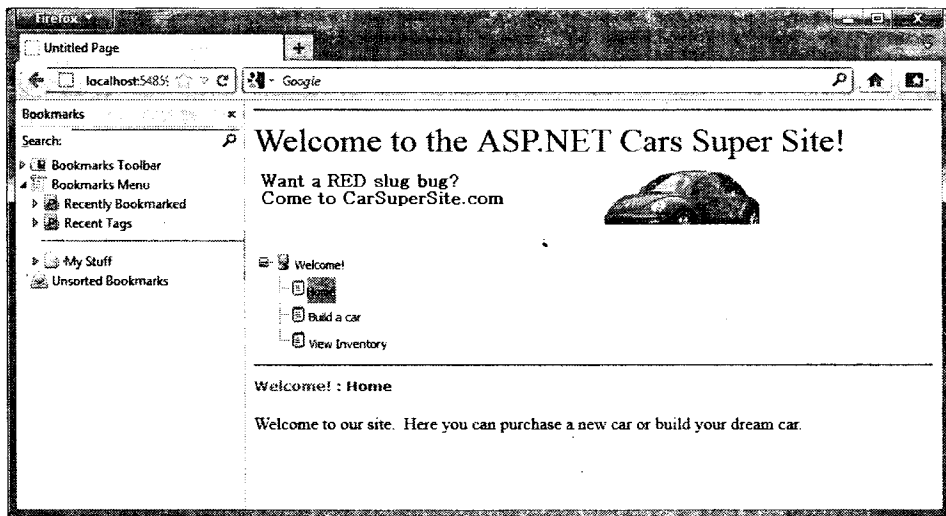


Рис. 33.14. Во время выполнения мастер-страницы и страницы содержимого визуализируются в единую форму

Проектирование страницы содержимого Inventory.aspx

Чтобы вставить в текущий проект страницу содержимого Inventory.aspx, откройте файл *.master в IDE-среде, выберите пункт меню Website⇒Add Content Page (Веб-сайт⇒Добавить страницу содержимого) и с помощью Solution Explorer переименуйте этот файл в Inventory.aspx. Назначение этой страницы содержимого состоит в отоб-

ражении записей таблицы Inventory из базы данных AutoLot внутри элемента управления GridView. Тем не менее, в отличие от предыдущей главы, этот элемент управления GridView будет сконфигурирован для взаимодействия с базой данных AutoLot, используя встроенную поддержку привязки данных.

Элемент управления GridView из ASP.NET обладает способностью представлять в разметке данные строки соединения и SQL-операторы Select, Insert, Update и Delete (или в качестве альтернативы — хранимые процедуры). Поэтому вместо написания всего необходимого кода ADO.NET вручную, можно позволить классу SqlDataSource сгенерировать разметку автоматически. С помощью визуальных конструкторов можно присвоить свойству DataSourceID элемента GridView соответствующий компонент SqlDataSource.

Выполнив несколько простых щелчков кнопкой мыши, можно настроить GridView на автоматическую выборку, обновление и удаление записей лежащего в основе хранилища данных. Хотя прием “нулевого кода” значительно сокращает общий объем кода, следует помнить, что платой за эту простоту будет утеря контроля, поэтому модель “нулевого кода” не всегда является удачным подходом для приложений масштаба предприятия. Эта модель может замечательно подойти для страниц с низким трафиком, при построении прототипа веб-сайта или для создания небольших домашних приложений.

Чтобы проиллюстрировать работу с элементом управления GridView (и логикой доступа к данным) в декларативной манере, начните с изменения страницы содержимого Inventory.aspx, добавив элемент управления Label. Затем откройте инструмент Server Explorer (через меню View (Вид)) и удостоверьтесь, что добавили соединение с данными из базы AutoLot, созданной ранее при изучении ADO.NET (процесс создания соединения с данными описан в главе 21). Теперь выберите таблицу Inventory и перетащите ее на область содержимого файла Inventory.aspx. В ответ на эти действия IDE-среда выполнит следующие шаги.

1. Файл web.config дополняется новым элементом <connectionStrings>.
2. Компонент SqlDataSource конфигурируется необходимой логикой Select, Insert, Update и Delete.
3. Свойство DataSourceID элемента GridView устанавливается в компонент SqlDataSource.

На заметку! В качестве альтернативы перетаскиванию таблиц из окна Server Explorer конфигурировать виджет GridView можно с помощью встроенного редактора (доступного через значок в верхнем правом углу). Выберите пункт <New Data Source...> в раскрывающемся списке Choose Data Source. Это активизирует мастер, который проведет через последовательность шагов для подключения этого компонента к необходимому источнику данных.

Если вы посмотрите на открывающееся объявление элемента управления GridView, то увидите, что свойство DataSourceID установлено в только что определенный компонент SqlDataSource:

```
<asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="False"
    DataKeyNames="CarID" DataSourceID="SqlDataSource1"
    EmptyDataText="There are no data records to display.">
  <Columns>
    <asp:BoundField DataField="CarID" HeaderText="CarID" ReadOnly="True"
        SortExpression="CarID" />
    <asp:BoundField DataField="Make" HeaderText="Make" SortExpression="Make" />
    <asp:BoundField DataField="Color" HeaderText="Color" SortExpression="Color" />
    <asp:BoundField DataField="PetName" HeaderText="PetName" SortExpression="PetName" />
  </Columns>
</asp:GridView>
```

Элемент `SqlDataSource` — это место, где происходит большая часть всех действий. В приведенной ниже разметке обратите внимание, что этот элемент записывает все необходимые SQL-операторы (в виде параметризованных запросов) для взаимодействия с таблицей `Inventory` базы данных `AutoLot`. Кроме того, используя синтаксис `$` свойства `ConnectionString`, этот компонент автоматически читает значение `<connectionString>` из файла `web.config`.

```
<asp:SqlDataSource ID="SqlDataSource1" runat="server"
  ConnectionString="<%$ ConnectionStrings:AutoLotConnectionString1 %%"
  DeleteCommand="DELETE FROM [Inventory] WHERE [CarID] = @CarID"
  InsertCommand="INSERT INTO [Inventory] ([CarID], [Make], [Color], [PetName])
    VALUES (@CarID, @Make, @Color, @PetName)"
  ProviderName="<%$ ConnectionStrings:AutoLotConnectionString1.ProviderName %%"
  SelectCommand="SELECT [CarID], [Make], [Color], [PetName] FROM [Inventory]"
  UpdateCommand="UPDATE [Inventory] SET [Make] = @Make,
    [Color] = @Color, [PetName] = @PetName WHERE [CarID] = @CarID">
  <DeleteParameters>
    <asp:Parameter Name="CarID" Type="Int32" />
  </DeleteParameters>
  <UpdateParameters>
    <asp:Parameter Name="Make" Type="String" />
    <asp:Parameter Name="Color" Type="String" />
    <asp:Parameter Name="PetName" Type="String" />
    <asp:Parameter Name="CarID" Type="Int32" />
  </UpdateParameters>
  <InsertParameters>
    <asp:Parameter Name="CarID" Type="Int32" />
    <asp:Parameter Name="Make" Type="String" />
    <asp:Parameter Name="Color" Type="String" />
    <asp:Parameter Name="PetName" Type="String" />
  </InsertParameters>
</asp:SqlDataSource>
```

В этот момент можно запустить веб-приложение, выбрать пункт меню `View Inventory` (Просмотреть склад) и просмотреть данные, как показано на рис. 33.15. (Обратите внимание, что элемент управления `GridView` здесь имеет уникальный внешний вид, установленный с помощью встроенного визуального конструктора).

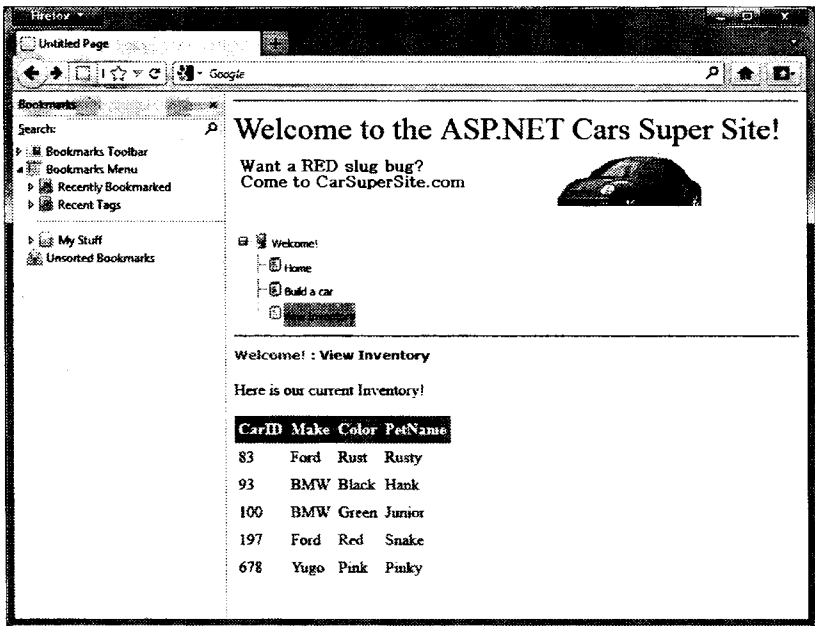


Рис. 33.15. Модель “нулевого кода” компонента `SqlDataSource`

Включение сортировки и разбиения на страницы

Элемент управления GridView можно легко сконфигурировать на выполнение сортировки (через гиперссылки имен столбцов) и разбиения на страницы (с помощью числовых гиперссылок или гиперссылок “следующая/предыдущая”). Для этого активизируйте встроенный редактор и отметьте флажки Enable Paging (Включить разбиение на страницы) и Enable Sorting (Включить сортировку), как показано на рис. 33.16.

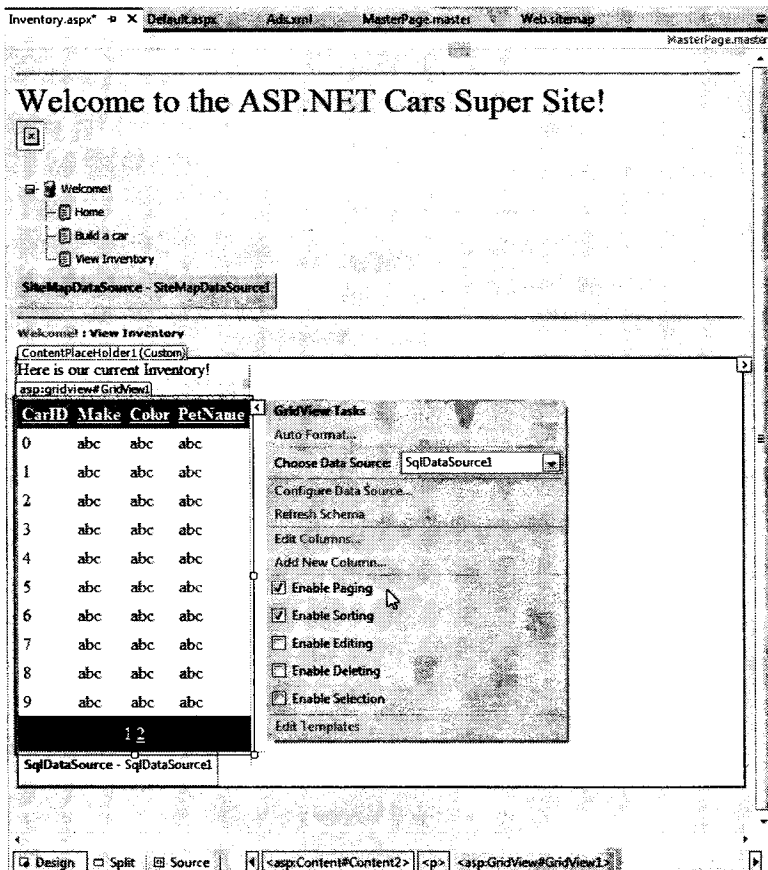


Рис. 33.16. Включение сортировки и разбиения на страницы

Теперь страница позволяет сортировать данные щелчками на именах столбцов и прокруткой данных через страницные ссылки (при условии, что в таблице Inventory присутствует достаточное количество записей).

Включение редактирования “на месте”

И последнее, что еще можно сделать на этой странице — включить поддержку редактирования “на месте” в элементе управления GridView. Учитывая, что SqlDataSource уже имеет необходимую логику Delete и Update, все, что понадобится сделать — отметить флажки Enable Deleting (Разрешить удаление) и Enable Editing (Разрешить редактирование) для элемента GridView (см. рис. 33.16). После этого на странице Inventory.aspx можно будет редактировать и удалять записи, как показано на рис. 33.17, обновляя лежащую в основе таблицу Inventory базы данных AutoLot.

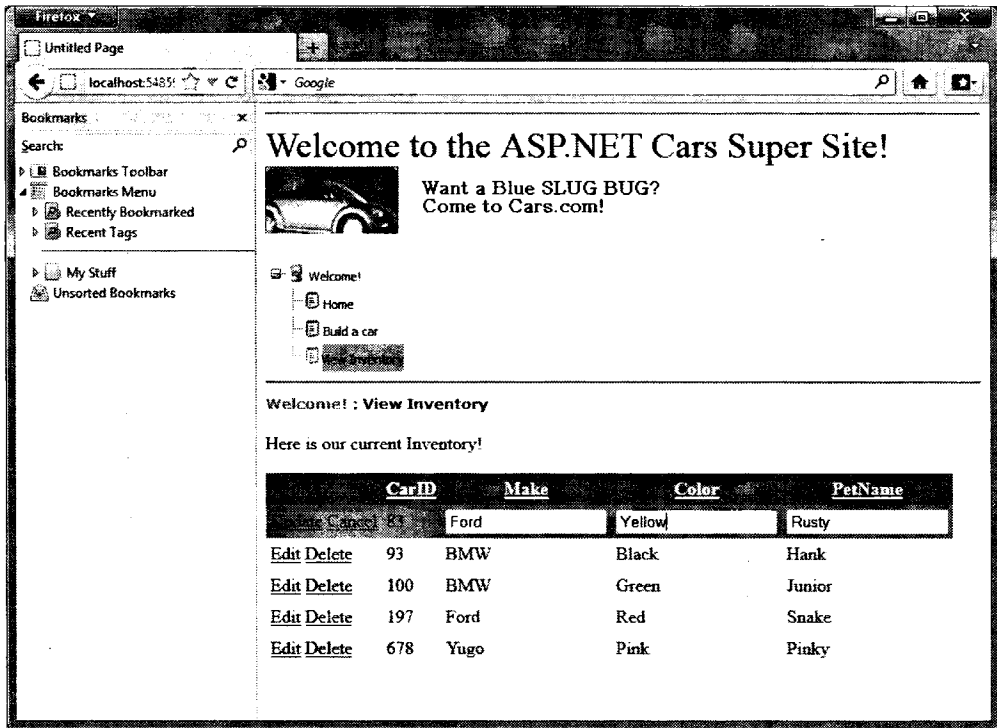


Рис. 33.17. Функциональность редактирования и удаления

На заметку! Включение редактирования "на месте" для элемента управления GridView требует наличия в таблице базы данных первичного ключа. Если активизировать эту опцию не удастся, то, скорее всего, столбец CarID не был установлен в качестве первичного ключа таблицы Inventory базы данных AutoLot.

Проектирование страницы содержимого BuildCar.aspx

В рассматриваемом примере осталось еще спроектировать страницу содержимого BuildCar.aspx. Для этого удостоверьтесь, что файл *.master открыт для редактирования и вставьте этот файл в текущий проект (с помощью пункта меню Website⇒Add Content Page; это альтернатива щелчку правой кнопкой мыши на мастер-странице проекта). Переименуйте новый файл в BuildCar.aspx в Solution Explorer.

На этой новой странице будет использоваться веб-элемент управления Wizard из ASP.NET, который обеспечивает простой способ для проведения конечного пользователя через последовательность взаимосвязанных шагов. Здесь эти шаги будут эмулировать акт комплектации автомобиля для покупки.

Поместите в область содержимого элементы управления Label и Wizard. Затем активизируйте встроенный редактор для Wizard и щелкните на ссылке Add/Remove WizardSteps (Добавить/удалить шаги мастера). Добавьте четыре шага, как показано на рис. 33.18.

После определения этих шагов вы заметите, что Wizard предлагает пустую область содержимого, куда можно перетаскивать элементы управления для текущего выбранного шага мастера.

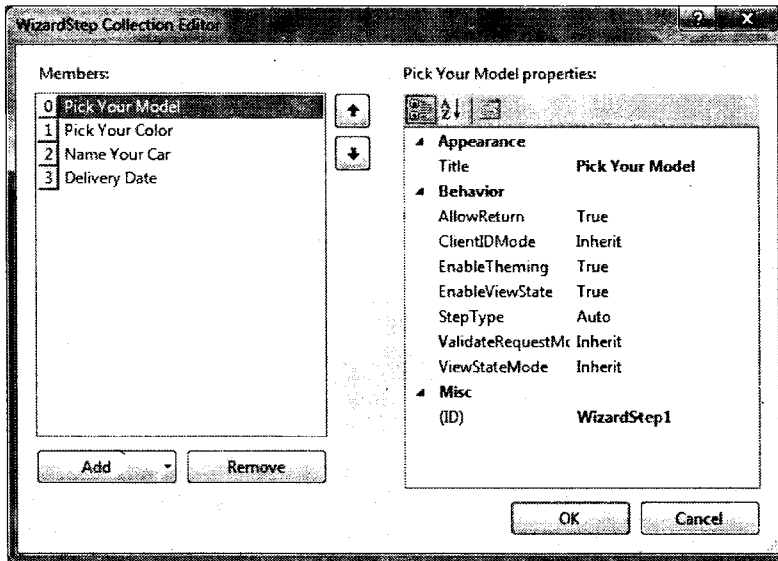


Рис. 33.18. Конфигурирование элемента управления Wizard

Для целей рассматриваемого примера модифицируем каждый шаг, добавив следующие элементы пользовательского интерфейса (не забудьте задать подходящие идентификаторы для каждого элемента в окне Properties):

- Pick Your Model (Выбор модели): элемент управления TextBox;
- Pick Your Color (Выбор цвета): элемент управления ListBox;
- Name Your Car (Название автомобиля): элемент управления TextBox;
- Delivery Date (Дата доставки): элемент управления Calendar.

Элемент управления ListBox — единственный элемент пользовательского интерфейса внутри Wizard, который требует дополнительных шагов. Выберите этот элемент в визуальном конструкторе (не забыв сначала выбрать ссылку Pick Your Color) и заполните виджет набором цветов, используя свойство Items в окне Properties. После этого вы увидите в области определения Wizard разметку вроде следующей:

```
<asp:ListBox ID="ListBoxColors" runat="server" Width="237px">
  <asp:ListItem>Purple</asp:ListItem>
  <asp:ListItem>Green</asp:ListItem>
  <asp:ListItem>Red</asp:ListItem>
  <asp:ListItem>Yellow</asp:ListItem>
  <asp:ListItem>Pea Soup Green</asp:ListItem>
  <asp:ListItem>Black</asp:ListItem>
  <asp:ListItem>Lime Green</asp:ListItem>
</asp:ListBox>
```

После определения всех шагов можно обработать событие FinishButtonClick для автоматически сгенерированной кнопки Finish (Готово). Однако имейте в виду, что кнопка Finish не видна до тех пор, пока не будет выбран завершающий шаг мастера. Выбрав последний шаг, просто выполните двойной щелчок на кнопке Finish для генерации обработчика событий. Внутри этого обработчика событий серверной стороны извлеките выбор из каждого элемента пользовательского интерфейса и постройте строку описания, которую присвойте свойству Text дополнительного элемента типа Label по имени lblOrder:

```
public partial class BuildCarPage : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
    }

    protected void carWizard_FinishButtonClick(object sender,
        WizardNavigationEventArgs e)
    {
        // Получить каждое значение.
        string order = string.Format("{0}, your {1} {2} will arrive on {3}.",
            txtCarPetName.Text, ListBoxColors.SelectedValue,
            txtCarModel.Text,
            carCalendar.SelectedDate.ToShortDateString());

        // Присвоить результирующую строку метке.
        lblOrder.Text = order;
    }
}
```

Итак, веб-приложение AspNetCarsSite готово. На рис. 33.19 показан элемент Wizard в действии.

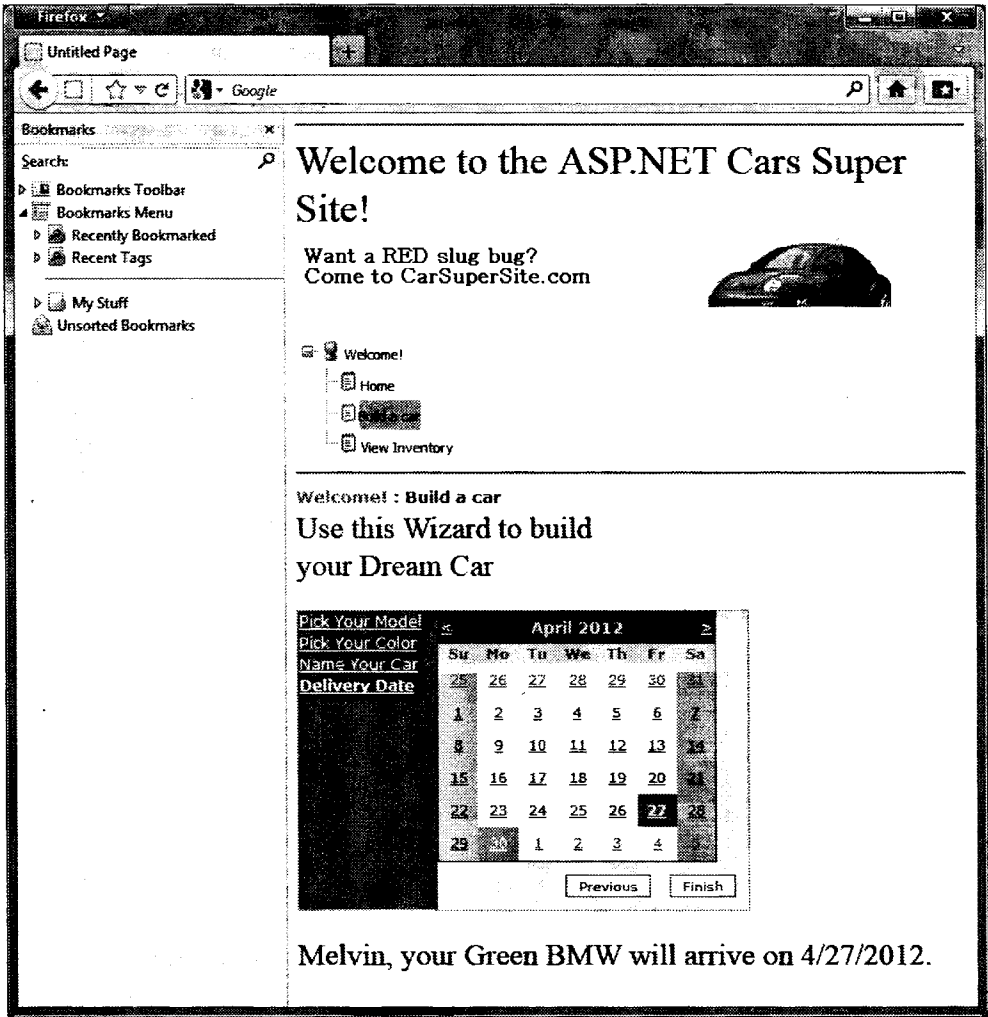


Рис. 33.19. Элемент управления Wizard в действии

На этом завершается краткое исследование разнообразных веб-элементов управления ASP.NET, мастер-страниц, страниц содержимого и навигации с помощью карты сайта. Далее мы переходим к исследованию функциональности элементов управления проверкой достоверности ASP.NET. Чтобы изолировать тематику настоящей главы, для демонстрации технологий проверки достоверности мы построим новый сайт. Тем не менее, элементы управления проверкой достоверности вполне можно добавить в текущий проект.

Исходный код. Веб-сайт `AspNetCarsSite` доступен в подкаталоге `Chapter 33`.

Роль элементов управления проверкой достоверности

Следующий набор элементов управления Web Forms, который мы рассмотрим, известен под общим названием *элементы управления проверкой достоверности*. В отличие от других элементов управления Web Forms, эти элементы не выдают HTML-разметку для визуализации, а применяются для генерации JavaScript-кода клиентской стороны в целях проверки достоверности данных формы. Как было показано в начале этой главы, проверка достоверности данных формы клиентской стороны удобна тем, что позволяет на месте проверять данные на предмет соответствия различным ограничениям, прежде чем отправлять их обратно веб-серверу, тем самым сокращая дорогостоящий циклический обмен с сервером. В табл. 33.3 приведена краткая сводка по элементам управления проверкой достоверности ASP.NET.

Таблица 33.3. Элементы управления проверкой достоверности ASP.NET

Элемент управления	Описание
<code>CompareValidator</code>	Проверяет значение в элементе ввода на равенство заданному значению другого элемента ввода или фиксированной константе
<code>CustomValidator</code>	Позволяет строить специальную функцию проверки достоверности, которая проверяет заданный элемент управления
<code>RangeValidator</code>	Определяет, находится ли данное значение в пределах заранее определенного диапазона
<code>RegularExpressionValidator</code>	Проверяет значение в ассоциированном элементе ввода на соответствие шаблону регулярного выражения
<code>RequiredFieldValidator</code>	Проверяет заданный элемент ввода на наличие значения (т.е. что он не пуст)
<code>ValidationSummary</code>	Отображает итог по всем ошибкам проверки достоверности страницы в формате простого списка, маркированного списка или одного абзаца. Ошибки могут отображаться встроенным способом и/или во всплывающем окне сообщения

Все элементы управления проверкой достоверности (кроме `ValidationSummary`) в конечном итоге являются производными от общего базового класса по имени `System.Web.UI.WebControls.BaseValidator` и потому обладают набором общих функциональных возможностей. Основные члены перечислены в табл. 33.4.

Таблица 33.4. Общие свойства элементов управления проверкой достоверности ASP.NET

Член	Описание
ControlToValidate	Получает или устанавливает элемент управления, подлежащий проверке достоверности
Display	Получает или устанавливает поведение сообщений об ошибках в элементе управления проверкой достоверности
EnableClientScript	Получает или устанавливает значение, указывающее, включена ли проверка достоверности клиентской стороны
ErrorMessage	Получает или устанавливает текст для сообщения об ошибке
ForeColor	Получает или устанавливает цвет сообщения, отображаемого при неудачной проверке достоверности

Чтобы проиллюстрировать работу с этими элементами управления проверкой достоверности, создайте новый проект ASP.NET Empty Web Site по имени ValidatorCtrls и вставьте в него новую веб-форму по имени Default.aspx. Для начала поместите на страницу четыре (именованных) элемента управления TextBox (с четырьмя соответствующими описательными элементами Label). Затем поместите на страницу рядом с каждым полем элементы управления RequiredFieldValidator, RangeValidator, RegularExpressionValidator и CompareValidator. И, наконец, добавьте элементы Button и Label. Возможный вид компоновки показан на рис. 33.20.

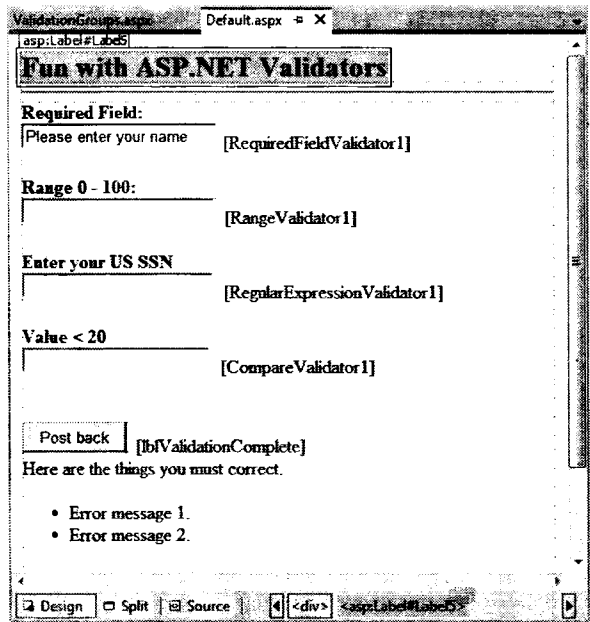


Рис. 33.20. Элементы управления проверкой достоверности обеспечат корректность данных формы перед выполнением обратной отправки

Теперь, имея начальный пользовательский интерфейс для проведения экспериментов, давайте пройдемся по процессу конфигурирования каждого элемента управления проверкой достоверности и посмотрим на конечный результат всего этого. Однако сначала необходимо модифицировать текущий файл web.config, разрешив клиентскую обработку для элементов управления проверкой достоверности.

Включение поддержки проверки достоверности с помощью JavaScript на стороне клиента

Начиная с версии ASP.NET 4.5, в Microsoft ввели новую настройку, которая может использоваться для управления реагированием элементов управления проверкой достоверности во время выполнения. Открыв текущий файл `web.config`, вы должны найти в нем следующую настройку:

```
<appSettings>
...
<add key="ValidationSettings:UnobtrusiveValidationMode" value="WebForms" />
</appSettings>
```

Когда эта настройка присутствует в файле `web.config`, веб-сайт будет проводить проверку достоверности с применением различных атрибутов данных HTML 5, а не отправлять обратно порции JavaScript-кода клиентской стороны для обработки веб-браузером. Учитывая, что в этой книге HTML 5 подробно не рассматривается, необходимо закомментировать (или удалить) эту строку, чтобы текущий пример работал корректно. Итак, закомментируйте этот раздел `<appSettings>` следующим образом:

```
<appSettings>
...
<!--
<add key="ValidationSettings:UnobtrusiveValidationMode" value="WebForms" />
-->
</appSettings>
```

Элемент управления RequiredFieldValidator

Конфигурирование `RequiredFieldValidator` осуществляется просто. Для этого соответствующим образом установите свойства `ErrorMessage` и `ControlToValidate` в окне `Properties` среды `Visual Studio`. Ниже показана результирующая разметка, которая обеспечивает проверку, что текстовое поле `txtRequiredField` не является пустым:

```
<asp:RequiredFieldValidator ID="RequiredFieldValidator1"
    runat="server" ControlToValidate="txtRequiredField"
    ErrorMessage="Oops! Need to enter data.">
</asp:RequiredFieldValidator>
```

Класс `RequiredFieldValidator` поддерживает свойство `InitialValue`. Его можно применять для проверки того факта, что пользователь ввел в связанном элементе `TextBox` какое-то значение, отличное от начального. Например, когда пользователь впервые получает страницу, можно сконфигурировать `TextBox`, чтобы он содержал значение `"Please enter your name"`. Если не установить свойство `InitialValue` элемента `RequiredFieldValidator`, исполняющая среда решит, что значение `"Please enter your name"` является допустимым. Таким образом, чтобы обязательное поле `TextBox` было допустимым только в случае, когда в нем введено значение, отличное от `"Please enter your name"`, сконфигурируйте виджет, как показано ниже:

```
<asp:RequiredFieldValidator ID="RequiredFieldValidator1"
    runat="server" ControlToValidate="txtRequiredField"
    ErrorMessage="Oops! Need to enter data."
    InitialValue="Please enter your name">
</asp:RequiredFieldValidator>
```

Элемент управления RegularExpressionValidator

Элемент управления RegularExpressionValidator может использоваться, когда к символам, введенным в заданном поле, необходимо применить шаблон. Например, чтобы обеспечить ввод в заданном поле TextBox корректного номера карточки социального страхования США, виджет может быть определен следующим образом:

```
<asp:RegularExpressionValidator ID="RegularExpressionValidator1"
  runat="server" ControlToValidate="txtRegExp"
  ErrorMessage="Please enter a valid US SSN."
  ValidationExpression="\d{3}-\d{2}-\d{4}">
</asp:RegularExpressionValidator>
```

Обратите внимание на то, как в RegularExpressionValidator определено свойство ValidationExpression. Если вы ранее не работали с регулярными выражениями, то все, что следует знать для целей рассматриваемого примера — это то, что они используются для проверки соответствия строки определенному шаблону. Здесь выражение "\d{3}-\d{2}-\d{4}" получает стандартный номер карточки социального страхования США в форме xxx-xx-xxxx (где x — десятичная цифра). Это конкретное регулярное выражение достаточно очевидно; однако предположим, что требуется проверить правильность телефонного номера, скажем, в Японии. Корректное выражение для этого случая выглядит намного сложнее: "(0\d{1,4}-|\d{0,4}\d{1,4}\d{1,4})?\d{1,4}-\d{4}". Удобно то, что при выборе свойства ValidationExpression в окне Properties доступен предварительно определенный список распространенных регулярных выражений (по щелчку на кнопке с троеточием).

На заметку! За программные манипуляции регулярными выражениями в .NET отвечают два пространства имен — System.Text.RegularExpressions и System.Web.RegularExpressions.

Элемент управления RangeValidator

В дополнение к свойствам MinimumValue и MaximumValue, в классе RangeValidator имеется свойство по имени Type. Поскольку нужно проверять пользовательский ввод на предмет вхождения в диапазон целых чисел, следует указать тип Integer (это не является стандартной установкой):

```
<asp:RangeValidator ID="RangeValidator1"
  runat="server" ControlToValidate="txtRange"
  ErrorMessage="Please enter value between 0 and 100."
  MaximumValue="100" MinimumValue="0" Type="Integer">
</asp:RangeValidator>
```

Класс RangeValidator также может использоваться для проверки вхождения в диапазон денежных значений, дат, чисел с плавающей точкой и строковых данных (стандартная установка).

Элемент управления CompareValidator

Наконец, обратите внимание, что CompareValidator поддерживает следующее свойство Operator:

```
<asp:CompareValidator ID="CompareValidator1" runat="server"
  ControlToValidate="txtComparison"
  ErrorMessage="Enter a value less than 20." Operator="LessThan"
  ValueToCompare="20" Type="Integer">
</asp:CompareValidator>
```

Учитывая, что предназначение элементов управления проверкой достоверности состоит в сравнении значения в текстовом поле и другого значения с использованием бинарной операции, не удивительно, что свойство `Operator` может принимать такие значения, как `LessThan`, `GreaterThan`, `Equal` и `NotEqual`. В `ValueToCompare` указывается значение, с которым нужно сравнивать. Обратите внимание, что атрибут `Type` установлен в `Integer`. По умолчанию `CompareValidator` выполняет сравнения со строковыми значениями.

На заметку! Элемент `CompareValidator` также может быть настроен для сравнения со значением внутри другого элемента управления `Web Forms` (а не с жестко закодированной константой) посредством свойства `ControlToCompare`.

Чтобы завершить код этой страницы, обработайте событие `Click` элемента управления `Button` и информируйте пользователя об успешном прохождении логики проверки достоверности:

```
public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
    }
    protected void btnPostBack_Click(object sender, EventArgs e)
    {
        lblValidationComplete.Text = "You passed validation!";
    }
}
```

Загрузите готовую страницу в браузер. В этот момент вы не должны увидеть каких-либо заметных изменений. Однако при попытке щелкнуть на кнопке `Submit` (Отправить) после ввода некорректных данных появится сообщение об ошибке. После ввода правильных данных сообщение об ошибке исчезнет и произойдет обратная отправка. Взглянув на HTML-разметку, визуализированную браузером, вы обнаружите, что элементы управления проверкой достоверности генерируют JavaScript-функцию клиентской стороны, которая использует специфическую библиотеку функций JavaScript, автоматически загружаемую на пользовательскую машину. Как только проверка достоверности прошла, данные формы отправляются обратно на сервер, где исполняющая среда выполняет ту же самую проверку еще раз на веб-сервере (просто чтобы удостовериться, что данные не были искажены по пути).

Кстати, если HTTP-запрос был прислан браузером, который не поддерживает JavaScript клиентской стороны, то вся проверка достоверности проходит на сервере. Таким образом, программировать элементы управления проверкой достоверности можно, не задумываясь о целевом браузере; возвращенная HTML-страница переадресует обработку ошибок веб-серверу.

Создание итоговой панели проверки достоверности

Следующая тема, касающаяся проверки достоверности, которую мы рассмотрим здесь — применение виджета `ValidationSummary`. В настоящий момент каждый элемент управления проверкой достоверности отображает свое сообщение об ошибке именно в том месте, куда он был помещен во время проектирования. Во многих случаях именно это и требуется. Однако в сложных формах с многочисленными виджетами ввода вариант с засорением формы многочисленными надписями красного цвета может не устроить. С помощью элемента управления `ValidationSummary` можно заставить все типы проверки достоверности отображать свои сообщения об ошибках в определенном месте страницы.

Первый шаг состоит в помещении элемента `ValidationSummary` в файл `*.aspx`. Дополнительно можно установить свойство `HeaderText` этого типа вместе с `DisplayMode`, которое по умолчанию отобразит список всех сообщений об ошибках в виде маркированного списка.

```
<asp:ValidationSummary id="ValidationSummary1"
    runat="server" Width="353px"
    HeaderText="Here are the things you must correct.">
</asp:ValidationSummary>
```

Затем понадобится установить свойство `Display` в `None` для всех индивидуальных элементов управления проверкой достоверности (т.е. `RequiredFieldValidator`, `RangeValidator` и т.д.). Это гарантирует отсутствие дублированных сообщений об ошибках при каждой неудаче проверки достоверности (одно — в итоговой панели, а другое — в месте расположения элемента управления проверкой достоверности). На рис. 33.21 показана итоговая панель в действии.

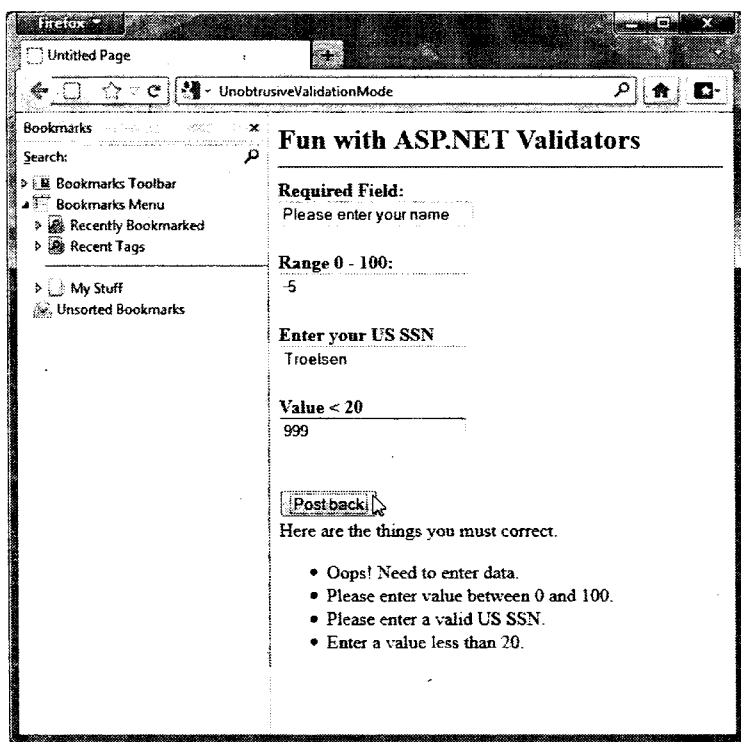


Рис. 33.21. Использование итоговой панели проверки достоверности

И последнее: если вы вместо этого хотите отображать сообщения об ошибках в окне сообщений клиентской стороны, установите свойство `ShowMessageBox` элемента управления `ValidationSummary` в `true`, а свойство `ShowSummary` — в `false`.

Определение групп проверки достоверности

Можно также определять группы, к которым относятся элементы управления проверкой достоверности. Это очень полезно, когда имеются области страницы, работающие как единое целое. Например, может существовать одна группа элементов управления в объекте `Panel`, предназначенная для ввода пользователем адреса электронной почты, и другая группа в другом объекте `Panel` — для ввода информации о кредитной

карте. С помощью групп можно конфигурировать каждый набор элементов управления для независимой проверки достоверности.

Вставьте в текущий проект новую страницу по имени `ValidationGroups.aspx`, на которой определены два элемента управления `Panel`. Первый объект `Panel` будет содержать элемент `TextBox` для некоторого пользовательского ввода (через `RequiredFieldValidator`), а второй объект `Panel` — элемент `TextBox` для ввода номера карточки социального страхования США (через `RegularExpressionValidator`). На рис. 33.22 показан возможный вариант пользовательского интерфейса.

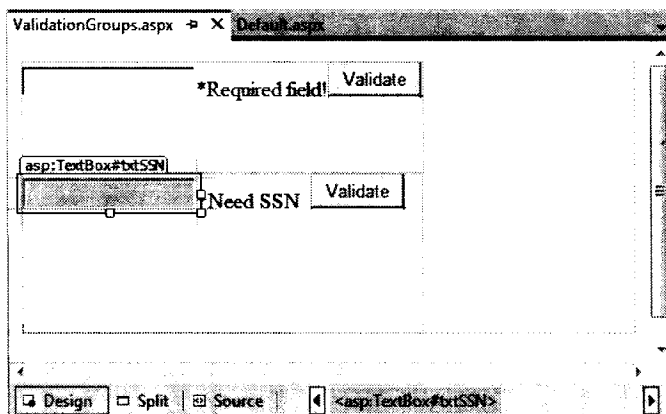


Рис. 33.22. Эти объекты `Panel` независимо конфигурируют свои области ввода

Чтобы обеспечить независимую проверку достоверности, просто включите элемент управления проверкой достоверности и проверяемый элемент управления в уникально именованную группу, используя свойство `ValidationGroup`. В следующем примере разметки обратите внимание, что применяемые здесь обработчики события `Click`, в сущности, являются пустыми заглушками в файле кода:

```
<form id="form1" runat="server">
  <asp:Panel ID="Panel1" runat="server" Height="83px" Width="296px">
    <asp:TextBox ID="txtRequiredData" runat="server"
      ValidationGroup="FirstGroup">
    </asp:TextBox>
    <asp:RequiredFieldValidator ID="RequiredFieldValidator1" runat="server"
      ErrorMessage="*Required field!" ControlToValidate="txtRequiredData"
      ValidationGroup="FirstGroup">
    </asp:RequiredFieldValidator>
    <asp:Button ID="bntValidateRequired" runat="server"
      OnClick="bntValidateRequired_Click"
      Text="Validate" ValidationGroup="FirstGroup" />
  </asp:Panel>

  <asp:Panel ID="Panel2" runat="server" Height="119px" Width="295px">
    <asp:TextBox ID="txtSSN" runat="server"
      ValidationGroup="SecondGroup">
    </asp:TextBox>
    <asp:RegularExpressionValidator ID="RegularExpressionValidator1"
      runat="server" ControlToValidate="txtSSN"
      ErrorMessage="*Need SSN" ValidationExpression="\d{3}-\d{2}-\d{4}"
      ValidationGroup="SecondGroup">
    </asp:RegularExpressionValidator>&nbsp;   </pre>
```

```

<asp:Button ID="btnValidateSSN" runat="server"
    OnClick="btnValidateSSN_Click" Text="Validate"
    ValidationGroup="SecondGroup" />
</asp:Panel>

</form>

```

Теперь щелкните правой кнопкой мыши на поверхности визуального конструктора этой страницы и выберите в контекстном меню пункт View In Browser (Просмотреть в браузере), чтобы удостовериться, что проверка данных в элементах каждой панели происходит во взаимоисключающей манере.

Исходный код. Веб-сайт ValidatorCtrls доступен в подкаталоге Chapter 33.

Работа с темами

До этих пор вы работали с многочисленными веб-элементами управления ASP.NET. Как было показано, каждый из них предоставляет набор свойств (многие из которых унаследованы от `System.Web.UI.WebControls.WebControl`), позволяющих настраивать внешний вид и поведение этих элементов пользовательского интерфейса (цвет фона, размер шрифта, стиль рамки и т.п.). Конечно, на многостраничном веб-сайте принято определять общий внешний вид и поведение виджетов различных типов. Например, все элементы `TextBox` могут быть сконфигурированы на поддержку определенного шрифта, все `Button` — чтобы иметь общий вид, а все `Calendar` — ярко-синюю рамку.

Очевидно, что было бы очень трудоемкой (и чреватой ошибками) задачей задавать одинаковые установки свойств для каждого виджета на каждой странице веб-сайта. Даже если вы в состоянии вручную обновить свойства каждого виджета пользовательского интерфейса на каждой странице, представьте, насколько утомительным может стать обновление цвета фона каждого элемента `TextBox`, когда это понадобится снова. Ясно, что должен существовать другой путь применения настроек пользовательского интерфейса на уровне всего сайта.

Один из возможных подходов, позволяющий упростить установку общего внешнего вида пользовательского интерфейса, заключается в определении *таблиц стилей*. Если у вас есть опыт веб-разработки, то вы знаете, что таблицы стилей определяют общий набор настроек пользовательского интерфейса, применяемых в браузере. Как и можно было ожидать, веб-элементы управления ASP.NET могут принимать стиль за счет присваивания значения свойству `CssStyle`.

Однако ASP.NET поставляется с дополняющей технологией для определения общего пользовательского интерфейса, которая называется *темами*. В отличие от таблиц стилей, темы применяются на веб-сервере (а не в браузере) и могут использоваться как программно, так и декларативно. Учитывая, что тема применяется на веб-сервере, она имеет доступ ко всем серверным ресурсам веб-сайта. Более того, темы определяются написанием той же разметки, которую можно найти в любом файле `*.aspx` (наверняка вы согласитесь, что синтаксис таблиц стилей чересчур лаконичный).

Вспомните из главы 32, что веб-приложения ASP.NET могут определять любое количество “специальных” каталогов, одним из которых является `App_Themes`. Этот единственный каталог может иметь подкаталоги, каждый из которых представляет одну из возможных тем веб-сайта. Например, на рис. 33.23 показан один каталог `App_Themes`, содержащий три подкаталога, каждый из которых имеет набор файлов, образующих саму тему.

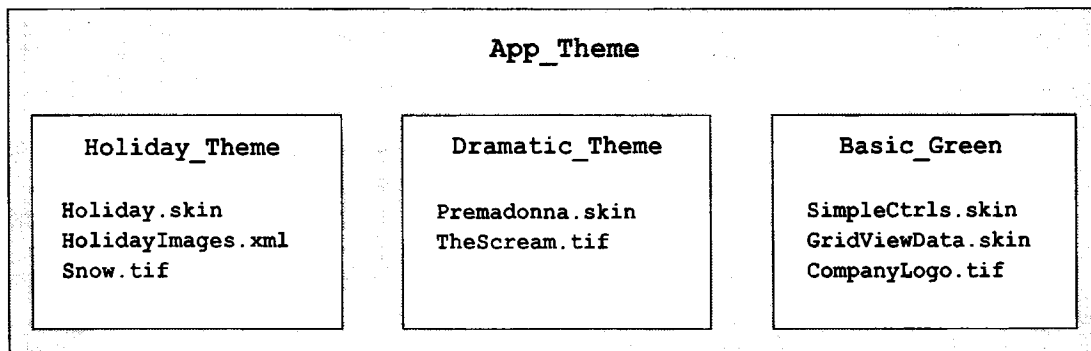


Рис. 33.23. Один каталог App_Themes может определять многочисленные темы

Файлы *.skin

Каждый подкаталог темы обязательно содержит файл *.skin. Эти файлы определяют внешний вид и поведение различных веб-элементов управления. В целях иллюстрации создайте новый проект ASP.NET Empty Web Site по имени FunWithThemes и вставьте в него новую веб-форму Default.aspx. Добавьте на эту страницу элементы управления Calendar, TextBox и Button. Конфигурировать каким-то особым образом эти элементы управления не понадобится, и точные их имена в этом примере не важны. Как будет показано, данные элементы управления будут служить целями для специальных обложек.

Добавьте новый файл *.skin (используя пункт меню Website⇒Add New Item) по имени BasicGreen.skin, как показано на рис. 33.24.

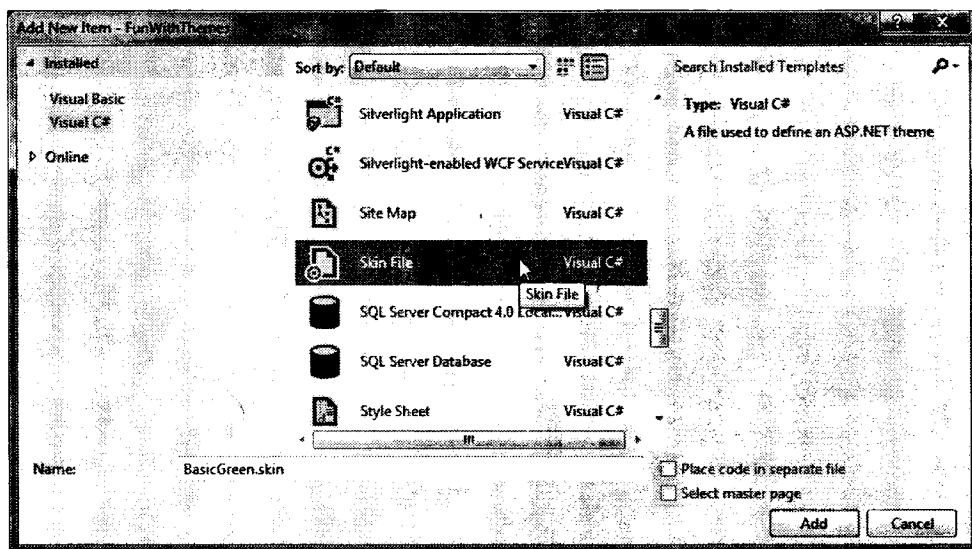


Рис. 33.24. Вставка файла *.skin

Среда Visual Studio предложит подтвердить добавление этого файла в App_Themes (что как раз и требуется). Если теперь заглянуть в Solution Explorer, в каталоге App_Themes будет виден подкаталог BasicGreen, который содержит новый файл BasicGreen.skin.

В файле *.skin определяется внешность и поведение различных виджетов с использованием синтаксиса для объявления элементов управления ASP.NET. К сожалению, в IDE-среде не предусмотрена поддержка визуального конструирования файлов *.skin.

Один из способов сокращения объема ввода состоит в добавлении к программе временного файла *.aspx (например, temp.aspx), который может применяться для построения пользовательского интерфейса виджетов с помощью визуального конструктора страниц Visual Studio.

Полученную в результате разметку можно затем скопировать в файл *.skin. При этом, однако, вы обязаны удалить атрибут ID каждого веб-элемента управления. Это должно иметь смысл, поскольку мы не определяем вид и поведение *отдельного* элемента Button (например), а *всех* элементов Button.

С учетом сказанного, вот как может выглядеть разметка для BasicGreen.skin, которая определяет внешний вид и стандартное поведение для типов Button, TextBox и Calendar:

```
<asp:Button runat="server" BackColor="#80FF80"/>
<asp:TextBox runat="server" BackColor="#80FF80"/>
<asp:Calendar runat="server" BackColor="#80FF80"/>
```

Обратите внимание, что каждый виджет по-прежнему имеет атрибут runat="server" (что обязательно), и ни одному из них не назначен атрибут ID.

Теперь определим вторую тему по имени CrazyOrange. В окне Solution Explorer щелкните правой кнопкой мыши на папке App_Themes и добавьте новую тему по имени CrazyOrange. Это приведет к созданию нового подкаталога внутри папки App_Themes сайта.

Затем щелкните правой кнопкой мыши на новой папке CrazyOrange в Solution Explorer и выберите в контекстном меню пункт Add New Item (Добавить новый элемент). В открывшемся диалоговом окне добавьте новый файл *.skin. Модифицируйте файл CrazyOrange.skin, определив уникальный внешний вид и поведение для тех же самых веб-элементов управления. Например:

```
<asp:Button runat="server" BackColor="#FF8000"/>
<asp:TextBox runat="server" BackColor="#FF8000"/>
<asp:Calendar BackColor="White" BorderColor="Black"
  BorderStyle="Solid" CellSpacing="1"
  Font-Names="Verdana" Font-Size="9pt" ForeColor="Black" Height="250px"
  NextPrevFormat="ShortMonth" Width="330px" runat="server">
  <SelectedDayStyle BackColor="#333399" ForeColor="White" />
  <OtherMonthDayStyle ForeColor="#999999" />
  <TodayDayStyle BackColor="#999999" ForeColor="White" />
  <DayStyle BackColor="#CCCCCC" />
  <NextPrevStyle Font-Bold="True" Font-Size="8pt" ForeColor="White" />
  <DayHeaderStyle Font-Bold="True" Font-Size="8pt"
    ForeColor="#333333" Height="8pt" />
  <TitleStyle BackColor="#333399" BorderStyle="Solid"
    Font-Bold="True" Font-Size="12pt"
    ForeColor="White" Height="12pt" />
</asp:Calendar>
```

После этого окно Solution Explorer должно выглядеть, как показано на рис. 33.25.

Теперь, когда сайт имеет несколько определенных тем, следующий логический шаг заключается в их применении к страницам. Как и можно было ожидать, для этого существует множество путей.

На заметку! Дизайн рассматриваемых примеров тем довольно прост (с целью экономии места на печатной странице). Вы можете развить их по своему вкусу.

Применение тем ко всему сайту

Если вы хотите обеспечить оформление каждой страницы сайта в одной и той же теме, проще всего обновить файл `web.config`.

Откройте текущий файл `web.config` и определите элемент `<pages>` внутри контекста корневого элемента `<system.web>`. Добавление атрибута `theme` к элементу `<pages>` гарантирует применение одной и той же темы ко всем страницам сайта (разумеется, значением этого атрибута должно быть имя одного из подкаталогов внутри `App_Themes`). Ниже показано основное изменение:

```
<configuration>
  <system.web>
    ...
    <pages controlRenderingCompatibilityVersion="4.5"
      theme="BasicGreen">
    </pages>
  </system.web>
</configuration>
```

Открыв эту страницу, вы увидите, что каждый виджет получил пользовательский интерфейс, определенный темой `BasicGreen`. Если изменить значение атрибута темы на `CrazyOrange` и снова запустить страницу, пользовательский интерфейс будет соответствовать заданному темой `CrazyOrange`.

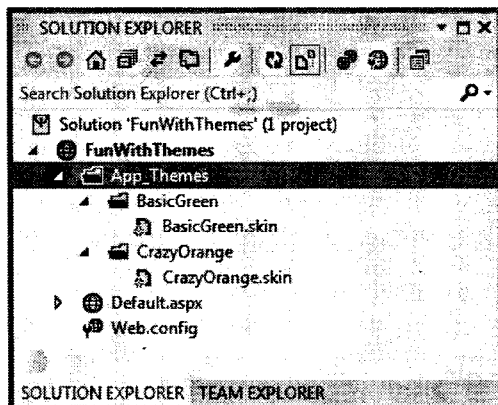


Рис. 33.25. Веб-сайт с несколькими темами

Применение тем на уровне страницы

Темы также можно применять к отдельным страницам. Это полезно в различных ситуациях. Например, возможно, в файле `web.config` определена тема для всего сайта (как описано в предыдущем разделе); однако определенной странице должна быть назначена другая тема. Для этого понадобится просто обновить директиву `<%@ Page %>`. При использовании Visual Studio средство IntelliSense отобразит все доступные темы, определенные в папке `App_Themes`.

```
<%@ Page Language="C#" AutoEventWireup="true"
  CodeFile="Default.aspx.cs" Inherits="_Default" Theme ="CrazyOrange" %>
```

Поскольку этой странице назначена тема `CrazyOrange`, а в файле `web.config` указана тема `BasicGreen`, то все страницы кроме этой будут визуализированы с применением темы `BasicGreen`.

Свойство SkinID

Иногда может потребоваться определить набор возможных внешних видов для отдельного виджета. Например, предположим, что необходимо определить два возможных пользовательских интерфейса для типа `Button` внутри темы `CrazyOrange`. В этом случае для различения каждого варианта внешнего вида служит свойство `SkinID` элемента управления внутри файла `*.skin`:

```
<asp:Button runat="server" BackColor="#FF8000"/>
<asp:Button runat="server" SkinID = "BigFontButton"
  Font-Size="30pt" BackColor="#FF8000"/>
```

Теперь при наличии страницы, которая использует тему CrazyOrange, каждому элементу Button по умолчанию будет назначена неименованная обложка Button. Если необходимо, чтобы в файле *.aspx обложку BigFontButton имели несколько разных кнопок, просто укажите SkinID внутри разметки:

```
<asp:Button ID="Button2" runat="server"
    SkinID="BigFontButton" Text="Button" /><br />
```

Программное назначение тем

И последний важный аспект касается назначения тем в коде. Это может пригодиться, когда необходимо предоставить конечным пользователям возможность выбора темы для текущего сеанса. Конечно, мы еще не рассматривали построение веб-приложений с поддержкой состояния, поэтому выбранная подобным образом тема будет утеряна между обратными отправлениями. В реальном рабочем сайте текущая тема пользователя сохраняется внутри переменной сеанса или же записывается в базу данных.

Для иллюстрации программного назначения темы добавьте к пользовательскому интерфейсу в файле Default.aspx три новых элемента управления Button, как показано на рис. 33.26. Обработайте событие Click каждого из этих элементов Button.

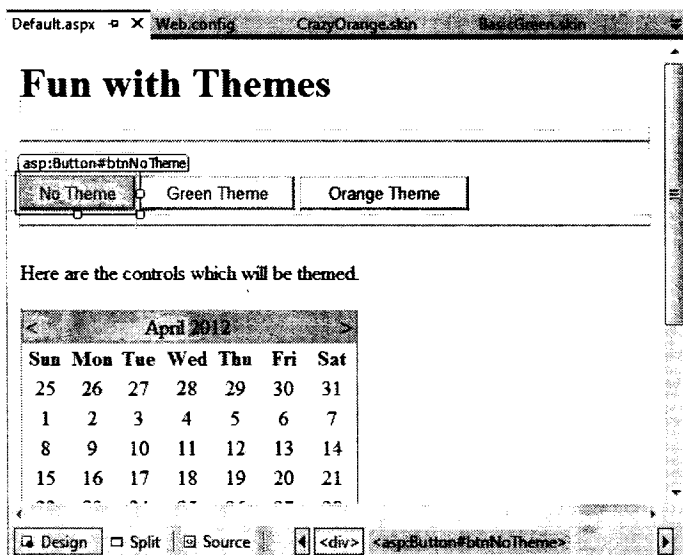


Рис. 33.26. Измененный пользовательский интерфейс примера применения тем

Теперь имейте в виду, что назначать тему программно можно только на определенных фазах жизненного цикла страницы. Обычно это делается внутри обработчика события Page_PreInit. С учетом сказанного, модифицируйте файл кода следующим образом:

```
partial class _Default : System.Web.UI.Page
{
    protected void btnNoTheme_Click(object sender, System.EventArgs e)
    {
        // Пустая строка означает, что тема не применяется.
        Session["UserTheme"] = "";
        // Снова инициализировать событие PreInit.
        Server.Transfer(Request.FilePath);
    }
}
```

```
protected void btnGreenTheme_Click(object sender, System.EventArgs e)
{
    Session["UserTheme"] = "BasicGreen";
    // Снова инициировать событие PreInit.
    Server.Transfer(Request.FilePath);
}

protected void btnOrangeTheme_Click(object sender, System.EventArgs e)
{
    Session["UserTheme"] = "CrazyOrange";
    // Снова инициировать событие PreInit.
    Server.Transfer(Request.FilePath);
}

protected void Page_PreInit(object sender, System.EventArgs e)
{
    try
    {
        Theme = Session["UserTheme"].ToString();
    }
    catch
    {
        Theme = "";
    }
}
```

Обратите внимание, что выбранная тема сохраняется в переменной сеанса (детали ищите в главе 34) по имени `UserTheme`, значение которой формально присваивается внутри обработчика событий `Page_PreInit()`. Когда пользователь щелкает на заданном элементе управления `Button`, мы вновь программно иницилируем событие `PreInit`, вызывая для этого метод `Server.Transfer()` и вновь запрашивая текущую страницу. Запустив эту страницу, вы обнаружите, что темы можно переключать щелчками на разных элементах `Button`.

Исходный код. Веб-сайт `FunWithThemes` доступен в подкаталоге `Chapter 33`.

Резюме

В этой главе рассматривалось использование разнообразных веб-элементов управления ASP.NET. Начали мы с изучения роли базовых классов `Control` и `WebControl`, после чего перешли к исследованию динамического взаимодействия с внутренней коллекцией элементов управления панели. Попутно была представлена новая модель навигации по сайту (файлы `*.sitemap` и компонент `SiteMapDataSource`), новый механизм привязки данных (через компонент `SqlDataSource` и элемент управления `GridView`), а также различные элементы управления проверкой достоверности.

Вторая половина главы была посвящена рассмотрению роли мастер-страниц и тем. Вспомните, что мастер-страницы могут использоваться для определения общей разметки для набора страниц сайта. Также вспомните, что файл `*.master` определяет любое количество мест заполнения содержимым, куда страницы содержимого подключают свое специальное содержимое пользовательского интерфейса. И, наконец, было показано, каким образом с помощью механизма тем ASP.NET декларативно или программно применять общий внешний вид и поведение пользовательского интерфейса к виджетам на веб-сервере.

Управление состоянием в ASP.NET

В предыдущих двух главах основное внимание было сосредоточено на композиции и поведении страниц ASP.NET, а также веб-элементов управления, которые они содержат. Настоящая глава основана на этой информации и посвящена роли файла `Global.asax` и лежащего в основе типа `HttpApplication`. Как будет показано, функциональность типа `HttpApplication` дает возможность перехватывать многочисленные события, которые позволяют трактовать веб-приложение как единую сущность, а не просто набор отдельных файлов `*.aspx`, управляемых мастер-страницей.

В дополнение к исследованию типа `HttpApplication`, в главе также рассматриваются все важные темы, касающиеся управления состоянием. Вы узнаете о роли состояния представления, переменных сеанса и приложения (включая кеш приложения), cookie-данных и API-интерфейса профилей ASP.NET (ASP.NET Profile API).

Проблема поддержки состояния

В начале главы 32 было указано, что HTTP является протоколом, *не поддерживающим состояние*. Это делает веб-разработку совершенно отличающейся от привычного процесса построения исполняемой сборки. Например, при разработке приложения с настольным пользовательским интерфейсом Windows можно рассчитывать на то, что переменные-члены, определенные в классе-наследнике `Form`, будут существовать в памяти до тех пор, пока пользователь явно не завершит исполняемую программу:

```
public partial class MainWindow : Window
{
    // Данные состояния!
    private string userFavoriteCar = "Yugo";
}
```

Тем не менее, в среде World Wide Web нельзя исходить из такого же удобного предположения. Чтобы удостовериться в этом, создайте новый проект ASP.NET Empty Web Site (Пустой веб-сайт ASP.NET) по имени `SimpleStateExample` и вставьте в него веб-форму. Внутри файла отделенного кода определите строковую переменную уровня страницы по имени `userFavoriteCar`:

```
public partial class _Default : System.Web.UI.Page
{
    // Данные состояния?
    private string userFavoriteCar = "Yugo";
    protected void Page_Load(object sender, EventArgs e)
    {
    }
}
```

Затем сконструируйте очень простой пользовательский интерфейс, показанный на рис. 34.1.

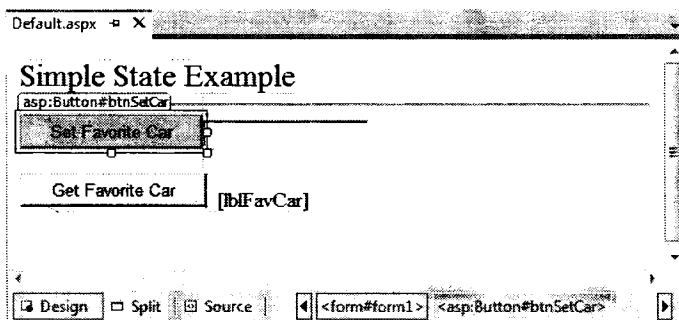


Рис. 34.1. Пользовательский интерфейс для простой страницы с состоянием

Обработчик события Click серверной стороны для кнопки Set Favorite Car (Установить предпочитаемый автомобиль) по имени btnSetCar позволяет пользователю присвоить переменной-члену типа string значение, взятое из элемента TextBox (с именем txtFavCar):

```
protected void btnSetCar_Click(object sender, EventArgs e)
{
    // Сохранить предпочитаемый автомобиль в переменной-члене.
    userFavoriteCar = txtFavCar.Text;
}
```

Обработчик события Click для кнопки Get Favorite Car (Получить предпочитаемый автомобиль) по имени btnGetCar отображает текущее значение переменной-члена внутри элемента Label (по имени lblFavCar) страницы:

```
protected void btnGetCar_Click(object sender, EventArgs e)
{
    // Отобразить значение переменной-члена.
    lblFavCar.Text = userFavoriteCar;
}
```

В приложении с графическим пользовательским интерфейсом Windows вполне корректно предполагать, что после того, как начальное значение установлено пользователем, оно запоминается на все время жизни настольного приложения. К сожалению, запустив построенное веб-приложение, вы обнаружите, что всякий раз, когда осуществляется обратная отправка на веб-сервер (по щелчку на любой кнопке), значение строковой переменной userFavoriteCar устанавливается обратно в свое начальное значение "Yugo", и потому текст в Label не изменяется.

Учитывая то, что протокол HTTP не имеет понятия о том, как автоматически запоминать данные после отправки HTTP-ответа, само собой разумеется, что объект Page уничтожается практически мгновенно. В результате, когда клиент отправляет обратно файл *.aspx, конструируется новый объект Page, который сбрасывает значение всех переменных-членов уровня страницы. Понятно, что это — серьезная проблема. Представьте, насколько бесполезной оказалась бы онлайн-торговля, если бы вся введенная ранее информация (вроде наименований товаров для приобретения) отбрасывалась при каждой отправке данных на веб-сервер. Когда необходимо запоминать информацию о пользователях, вошедших на сайт, приходится применять различные приемы управления состоянием.

На заметку! Эта проблема никоим образом не ограничивается ASP.NET. Веб-приложения Java, приложения CGI, приложения на классическом ASP и приложения PHP также сталкиваются с необходимостью решения задачи управления состоянием.

Чтобы запомнить значение строковой переменной `userFavoriteCar` между обратными отправлениями, один из подходов заключается в сохранении этого значения в *переменной сеанса*. С детальной информацией о состоянии сеанса вы ознакомитесь далее в этой главе. Однако для полноты примера внесем необходимые изменения в текущую страницу (обратите внимание, что закрытая переменная-член типа `string` больше не используется, потому ее можно закомментировать либо вообще удалить ее определение):

```
public partial class _Default : System.Web.UI.Page
{
    // Данные состояния?
    // private string userFavoriteCar = "Yugo";

    protected void Page_Load(object sender, EventArgs e)
    {
    }

    protected void btnSetCar_Click(object sender, EventArgs e)
    {
        // Сохранить значение в переменной сеанса.
        Session["UserFavCar"] = txtFavCar.Text;
    }

    protected void btnGetCar_Click(object sender, EventArgs e)
    {
        // Получить значение из переменной сеанса.
        lblFavCar.Text = (string)Session["UserFavCar"];
    }
}
```

Если теперь запустить приложение, то запись о предпочитаемом автомобиле будет сохранена между обратными отправлениями, благодаря объекту `HttpSessionState`, которым опосредованно манипулирует унаследованное свойство `Session`. Данные сеанса (которые будут подробно рассматриваться далее в этой главе) — это только один способ “запоминания” информации на веб-сайтах. В следующем разделе представлены другие возможные варианты, поддерживаемые ASP.NET.

Исходный код. Веб-сайт `SimpleStateExample` доступен в подкаталоге `Chapter 34`.

Приемы управления состоянием ASP.NET

В ASP.NET предлагается несколько механизмов для поддержки информации о состоянии внутри веб-приложений. На выбор доступны следующие общие варианты:

- использование состояния представления ASP.NET;
- использование состояния элемента управления ASP.NET;
- определение переменных уровня приложения;
- использование объекта кеша;
- определение переменных уровня сеанса;
- определение cookie-данных.

В дополнение к перечисленным выше подходам, для сохранения пользовательских данных на постоянной основе в ASP.NET предлагается готовый API-интерфейс Profile. Далее мы подробно рассмотрим детали каждого из упомянутых подходов, начиная с состояния представления ASP.NET.

Роль состояния представления ASP.NET

Понятие *состояние представления* уже несколько раз встречалось в предыдущих главах, но без формального определения, поэтому давайте проясним его сейчас. Без поддержки со стороны инфраструктуры веб-разработчики вынуждены вручную переполнять значениями виджеты ввода на форме во время процесса конструирования исходящего HTTP-ответа.

В ASP.NET мы больше не обязаны вручную собирать и заново заполнять значениями виджеты HTML, поскольку исполняющая среда ASP.NET автоматически встраивает в форму скрытое поле (по имени `__VIEWSTATE`), которое будет передаваться между браузером и определенной страницей. Данные, присваиваемые этому полю, представляют собой закодированную по алгоритму Base64 строку, содержащую набор пар “имя/значение”, которые соответствуют значениям виджетов пользовательского интерфейса на странице.

Обработчик событий `Init` базового класса `System.Web.UI.Page` — это сущность, ответственная за чтение входных значений из поля `__VIEWSTATE` для заполнения соответствующих переменных-членов в производном классе. (Именно по этой причине обращаться к состоянию виджета в контексте обработчика события `Init` страницы в лучшем случае рискованно.)

Кроме того, непосредственно перед отправкой исходящего ответа запросившему браузеру данные `__VIEWSTATE` используются для повторного заполнения виджетов формы. Очевидно, наилучшим в этом аспекте ASP.NET является то, что все это происходит без какого-либо участия с вашей стороны. Разумеется, при необходимости всегда можно взаимодействовать, изменять или отключать эту функциональность. Чтобы понять, как это делается, давайте рассмотрим конкретный пример состояния представления.

Демонстрация работы с состоянием представления

Создайте новый проект ASP.NET Empty Web Site по имени `ViewStateApp` и добавьте в него новую веб-форму. На страницу `*.aspx` поместите веб-элемент управления `ListBox` по имени `myListBox` и элемент управления `Button` по имени `btnPostBack`.

Теперь в окне **Properties** (Свойства) среды Visual Studio найдите свойство `Items` и добавьте в `ListBox` четыре элемента `Listitem`, используя ассоциированное диалоговое окно. Результирующая разметка должна быть похожа на показанную ниже:

```
<asp:ListBox ID="myListBox" runat="server">
  <asp:ListItem>Item One</asp:ListItem>
  <asp:ListItem>Item Two</asp:ListItem>
  <asp:ListItem>Item Three</asp:ListItem>
  <asp:ListItem>Item Four</asp:ListItem>
</asp:ListBox>
```

Обратите внимание, что элементы списка `ListBox` жестко закодированы внутри файла `*.aspx`. Как уже известно, перед отправкой финального HTTP-ответа все определения `<asp: >` в веб-форме ASP.NET будут автоматически визуализированы в свои HTML-представления (благодаря наличию атрибута `runat="server"`).

Директива `<% Page %>` имеет необязательный атрибут по имени `EnableViewState`, который по умолчанию установлен в `true`.

Чтобы отключить поведение состояния представления, измените директиву `<%@ Page %>` следующим образом:

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeFile="Default.aspx.cs" Inherits="_Default"
    EnableViewState =>false %>
```

Итак, что же в точности означает отключение состояния представления? Это зависит от обстоятельств. Исходя из предыдущего определения термина, могло бы показаться, что если отключить состояние представления для файла *.aspx, то значения в `ListBox` не будут запоминаться между обратными отправками на веб-сервер. Однако если запустить приложение в том виде, как оно есть, легко заметить, что информация в `ListBox` сохраняется, независимо от того, сколько раз производится обратная отправка страницы.

На самом деле, если вы просмотрите HTML-разметку, возвращенную браузеру (щелкнув правой кнопкой мыши на странице внутри браузера и выбрав в контекстном меню пункт `View Source` (Исходный код страницы)), то удивитесь еще больше, обнаружив, что скрытое поле `__VIEWSTATE` по-прежнему присутствует:

```
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
    value="/wEPDwUKLTm4MTM2MDM4NGRkqGC6gjEV25JnddkJiRmoIc10SIA=" />
```

Однако предположим, что `ListBox` заполняется динамически в файле отдельного кода, а не внутри определения HTML-дескриптора `<form>`. Для начала удалите объявления `<asp:ListItem>` из текущего файла *.aspx:

```
<asp:ListBox ID="myListBox" runat="server">
</asp:ListBox>
```

Затем заполните список элементами внутри обработчика событий `Load` в файле отдельного кода:

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        // Заполнить ListBox динамически!
        myListBox.Items.Add("Item One");
        myListBox.Items.Add("Item Two");
        myListBox.Items.Add("Item Three");
        myListBox.Items.Add("Item Four");
    }
}
```

Отправив эту обновленную страницу, вы увидите, что при первом запросе ее браузером значения `ListBox` присутствуют в том виде, как они были определены. Однако после обратной отправки `ListBox` внезапно становится пустым. Первое правило состояния представления ASP.NET заключается в том, что его работу можно заметить только тогда, когда есть виджеты, значения которых динамически генерируются в коде. Если жестко закодировать эти значения внутри дескрипторов `<form>` файла *.aspx, то состояние этих элементов будет всегда запоминаться между обратными отправками (даже если для какой-то страницы атрибут `EnableViewState` установлен в `false`).

Если идея отключения состояния представления для всего файла *.aspx кажется чересчур экстремальной, имейте в виду, что каждый потомок базового класса `System.Web.UI.Control` наследует свойство `EnableViewState`, которое существенно упрощает задачу отключения состояния представления отдельно для каждого элемента управления:

```
<asp:GridView id="myHugeDynamicallyFilledGridOfData" runat="server"
    EnableViewState="false">
</asp:GridView>
```

На заметку! Начиная с версии .NET 4.0, объемные данные состояния представления автоматически сжимаются, чтобы сократить размер этого скрытого поля формы.

Добавление специальных данных в состояние представления

В дополнение к свойству `EnableViewState`, класс `System.Web.UI.Control` предоставляет защищенное свойство по имени `ViewState`. “За кулисами” это свойство обеспечивает доступ к типу `System.Web.UI.StateBag`, представляющему все данные, которые хранятся в поле `__VIEWSTATE`. Используя индексатор типа `StateBag`, можно встраивать специальную информацию в скрытое поле формы `__VIEWSTATE` с применением набора пар “имя/значение”. Ниже показан простой пример:

```
protected void btnAddToVS_Click(object sender, EventArgs e)
{
    ViewState["CustomViewStateItem"] = "Some user data";
    lblVSValue.Text = (string)ViewState["CustomViewStateItem"];
}
```

Поскольку тип `System.Web.UI.StateBag` спроектирован для оперирования над типами `System.Object`, при доступе к значению по заданному ключу потребуются явно приводить его к типу данных, лежащему в основе (в рассматриваемом случае это `System.String`). Однако имейте в виду, что значения, помещенные в поле `__VIEWSTATE`, не могут быть объектом в буквальном смысле. В частности, допустимыми типами являются только `String`, `Integer`, `bool`, `ArrayList`, `Hashtable` и массивы всех этих типов.

Поэтому, с учетом того, что страницы `*.aspx` могут вставлять специальные порции информации в строку `__VIEWSTATE`, следующий логичный шаг заключается в том, чтобы выяснить, когда это может понадобиться. В большинстве случаев специальные данные состояния представления больше подходят для хранения специфичной для пользователя информации. Например, можно установить данные состояния представления, которые указывают, как пользователь желает видеть интерфейс элемента `GridView` (скажем, порядок сортировки). Тем не менее, данные состояния представления не слишком подходят для хранения полномасштабных пользовательских данных, таких как элементы в корзине для покупок или кешированные объекты `DataSet`. Когда нужно хранить сложную информацию подобного рода, лучше работать с данными сеанса или данными приложения. Но прежде чем перейти к ним, следует прояснить роль файла `Global.asax`.

Исходный код. Веб-сайт `ViewStateApp` доступен в подкаталоге `Chapter 34`.

Роль файла `Global.asax`

К этому моменту приложение ASP.NET может выглядеть всего лишь как набор файлов `*.aspx` и соответствующих веб-элементов управления. Хотя веб-приложение можно строить, просто связывая вместе набор веб-страниц, скорее всего, понадобится способ взаимодействия с веб-приложением как с единым целым. В этом случае в веб-приложение ASP.NET можно включить дополнительный файл `Global.asax` через пункт меню `Website⇒Add New Item (Веб-сайт⇒Добавить новый элемент)`, как показано на рис. 34.2 (обратите внимание, что нужно выбрать вариант `Global Application Class (Глобальный класс приложения)`).

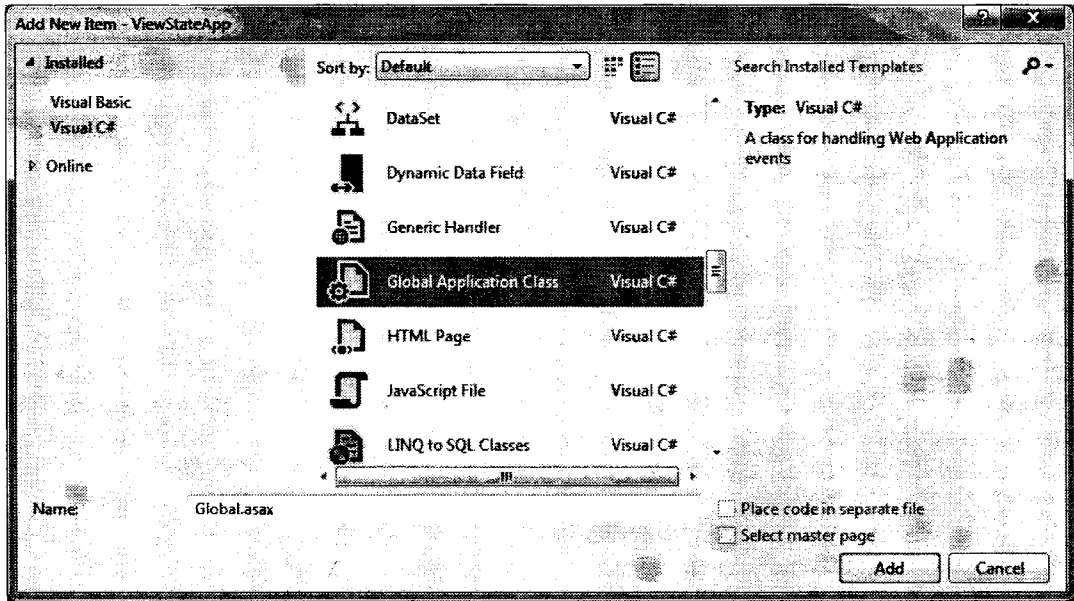


Рис. 34.2. Добавление файла Global.aspx

Попросту говоря, Global.aspx — это почти то же самое, что запускаемый двойным щелчком файл *.exe, который можно получить в мире ASP.NET, в том смысле, что этот тип представляет поведение времени выполнения самого веб-сайта. После добавления файла Global.aspx в веб-проект вы увидите, что это всего лишь блок `<script>`, содержащий набор обработчиков событий:

```
<%@ Application Language="C#" %>
<script runat="server">
    void Application_Start(object sender, EventArgs e)
    {
        // Код, выполняемый при запуске приложения.
    }

    void Application_End(object sender, EventArgs e)
    {
        // Код, выполняемый при останове приложения.
    }

    void Application_Error(object sender, EventArgs e)
    {
        // Код, выполняемый при возникновении необработанной ошибки.
    }

    void Session_Start(object sender, EventArgs e)
    {
        // Код, выполняемый при запуске сеанса.
    }

    void Session_End(object sender, EventArgs e)
    {
        // Код, выполняемый при завершении сеанса.
        // Примечание. Событие Session_End инициируется, только если в файле web.config
        // режим состояния сеанса установлен в InProc. Если режим сеанса установлен
        // в StateServer или SQLServer, это событие не возникает.
    }
</script>
```

Тем не менее, внешность может быть обманчивой. Во время выполнения код внутри блока `<script>` организуется в класс, производный от `System.Web.HttpApplication`. Следовательно, внутри любого из предоставленных обработчиков событий можно получать доступ к членам родительского класса с помощью ключевых слов `this` или `base`.

Как уже упоминалось, члены, определенные внутри `Global.asax`, представляют собой обработчики событий, которые позволяют взаимодействовать с событиями уровня приложения (и уровня сеанса). В табл. 34.1 описаны роли всех обработчиков событий.

Таблица 34.1. Основные обработчики событий пространства имен `System.Web`

Обработчик событий	Описание
<code>Application_Start()</code>	Этот обработчик событий вызывается в самом начале при запуске веб-приложения. Таким образом, данное событие инициируется только один раз за все время жизни веб-приложения. Это идеальное место для определения данных уровня приложения, используемых по всему веб-приложению
<code>Application_End()</code>	Этот обработчик событий вызывается при остановке приложения. Это происходит, когда последний пользователь покидает приложение по тайм-ауту, или когда вы вручную останавливаете приложение в IIS
<code>Session_Start()</code>	Этот обработчик событий вызывается, когда новый пользователь входит в приложение. Здесь можно устанавливать все специфические для пользователя данные, которые нужно сохранить между обратными отправками
<code>Session_End()</code>	Этот обработчик событий вызывается при завершении пользовательского сеанса (обычно по истечении предопределенного тайм-аута)
<code>Application_Error()</code>	Это глобальный обработчик ошибок, который вызывается, когда веб-приложение сгенерировало необработанное исключение

Глобальный обработчик исключений “последнего шанса”

Давайте сначала рассмотрим роль обработчика событий `Application_Error()`. Вспомните, что определенная страница может обрабатывать событие `Error`, в результате перехватывая любое необработанное исключение, которое произошло в контексте самой этой страницы. Аналогично, обработчик событий `Application_Error()` — это последнее место, где можно обработать исключение, которое не было обработано страницей. Как и в случае события `Error` уровня страницы, обращаться к определенному исключению `System.Exception` можно через унаследованное свойство `Server`:

```
void Application_Error(object sender, EventArgs e)
{
    // Получить необработанную ошибку.
    Exception ex = Server.GetLastError();

    // Обработать ошибку...

    // По завершении очистить ошибку.
    Server.ClearError();
}
```

Поскольку обработчик событий `Application_Error()` является обработчиком исключений “последнего шанса” для веб-приложения, довольно часто этот метод реализуется таким образом, что пользователь переносится на заранее определенную страницу, связанную с ошибкой, на сервере. Другие распространенные действия могут включать отправку сообщения электронной почты веб-администратору или запись во внешний журнал ошибок.

Базовый класс `HttpApplication`

Как упоминалось ранее, сценарий `Global.asax` динамически генерируется в виде класса, производного от базового класса `System.Web.HttpApplication`, который предлагает такого же рода функциональность, что и класс `System.Web.UI.Page` (без видимого пользовательского интерфейса). В табл. 34.2 документированы ключевые свойства этого класса.

Таблица 34.2. Ключевые свойства класса `System.Web.HttpApplication`

Свойство	Описание
<code>Application</code>	Это свойство позволяет взаимодействовать с данными уровня приложения, используя тип <code>HttpApplicationState</code>
<code>Request</code>	Это свойство позволяет взаимодействовать с входящим запросом HTTP, используя лежащий в основе объект <code>HttpRequest</code>
<code>Response</code>	Это свойство позволяет взаимодействовать с исходящим ответом HTTP, используя лежащий в основе объект <code>HttpResponse</code>
<code>Server</code>	Это свойство позволяет получить встроенный серверный объект для текущего запроса, используя лежащий в основе объект <code>HttpServerUtility</code>
<code>Session</code>	Это свойство позволяет взаимодействовать с данными уровня сеанса, используя лежащий в основе объект <code>HttpSessionState</code>

Поскольку файл `Global.asax` явно не документирует `HttpApplication` в качестве лежащего в основе базового класса, важно помнить, что здесь применимы все правила отношения “является”.

Различие между состоянием приложения и состоянием сеанса

В ASP.NET состояние приложения поддерживается экземпляром типа `HttpApplicationState`. Этот класс позволяет разделять глобальную информацию между всеми пользователями (и всеми страницами), работающими с приложением ASP.NET. С его помощью можно не только разделять все данные приложения между всеми пользователями сайта; в случае, если значение этих данных уровня приложения изменяется, новое значение становится видимым всем пользователям после следующей обратной отправки.

С другой стороны, состояние сеанса служит для запоминания информации, связанной с определенным пользователем (такой как товары в корзине покупок). Физически состояние сеанса пользователя представлено типом класса `HttpSessionState`. Когда новый пользователь входит в веб-приложение ASP.NET, исполняющая среда автоматически назначает ему новый идентификатор сеанса, который устаревает по умолчанию после 20 минут отсутствия активности. Таким образом, если к сайту подключено 20 000 пользователей, существует 20 000 отдельных объектов `HttpSessionState`, каждому из которых автоматически назначен уникальный идентификатор сеанса. Отношения между веб-приложением и веб-сеансом показаны на рис. 34.3.

Поддержка данных состояния уровня приложения

Тип `HttpApplicationState` позволяет разработчикам разделять глобальную информацию между множеством пользователей в приложении ASP.NET. В табл. 34.3 описаны некоторые основные члены этого типа.

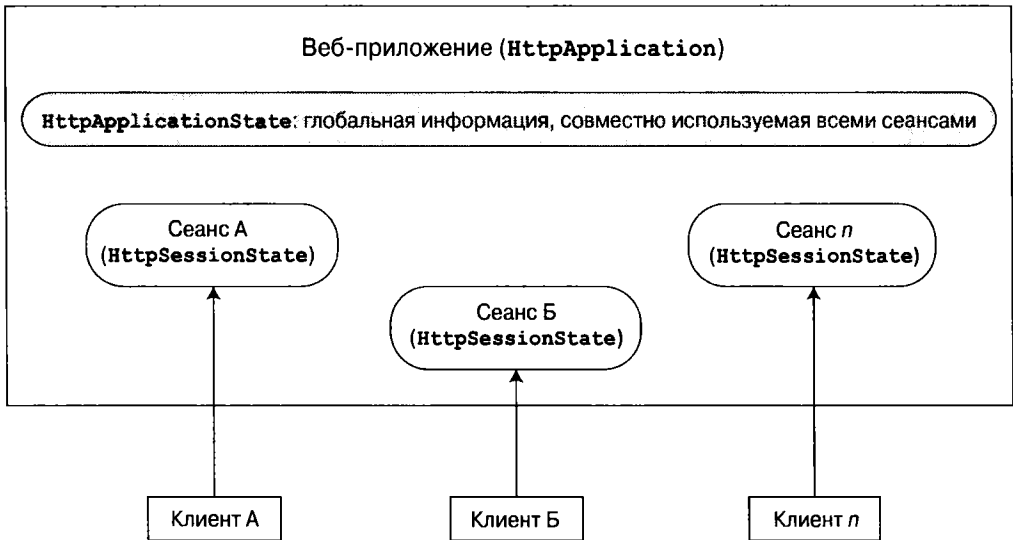


Рис. 34.3. Отличие между состоянием приложения и состоянием сеанса

Таблица 34.3. Члены типа HttpSessionState

Член	Описание
Add()	Этот метод позволяет добавить новую пару "имя/значение" к объекту HttpSessionState. Обратите внимание, что этот метод обычно не используется, а вместо него применяется индекатор класса HttpSessionState
AllKeys	Это свойство возвращает массив объектов string, которые представляют все имена в объекте HttpSessionState
Clear()	Этот метод очищает все элементы в объекте HttpSessionState. Его функциональность эквивалентна методу RemoveAll()
Count	Это свойство получает количество объектов-элементов, хранимых в объекте HttpSessionState
Lock(), Unlock()	Эти два метода используются, когда необходимо изменить набор переменных приложения в безопасной к потокам манере
RemoveAll(), Remove(), RemoveAt()	Эти методы удаляют определенный элемент (по строковому имени) из объекта HttpSessionState. Метод RemoveAt() удаляет элемент по числовому индексу

Чтобы проиллюстрировать работу с состоянием приложения, создайте проект ASP.NET Empty Web Site по имени AppState (и добавьте в него новую веб-форму). Затем вставьте новый файл Global.asax. После создания членов данных, которые могут быть разделены между всеми активными сеансами, нужно будет установить пары "имя/значение". В большинстве случаев самым естественным местом для этого является обработчик события Application_Start() в файле Global.asax.cs, например:

```
void Application_Start(Object sender, EventArgs e)
{
    // Установить некоторые переменные приложения.
    Application["SalesPersonOfTheMonth"] = "Chucky";
    Application["CurrentCarOnSale"] = "Colt";
    Application["MostPopularColorOnLot"] = "Black";
}
```

На протяжении жизненного цикла веб-приложения (т.е. пока приложение не будет остановлено вручную либо не истечет тайм-аут последнего пользователя) любой пользователь на любой странице при необходимости может обращаться к этим значениям. Предположим, что есть страница, которая отображает текущую скидку на автомобиль в элементе Label через обработчик события Click кнопки:

```
protected void btnShowCarOnSale_Click(object sender, EventArgs arg)
{
    lblCurrCarOnSale.Text = string.Format("Sale on {0}'s today!",
        (string)Application["CurrentCarOnSale"]);
}
```

Обратите внимание на то, что, как и в случае свойства ViewState, значение, возвращенное объектом HttpSessionState, должно быть приведено к корректному типу, лежащему в основе; это объясняется тем, что свойство Application оперирует с общими типами System.Object.

Теперь, учитывая, что свойство Application может хранить любой тип, становится возможным помещать в него объекты пользовательских типов (или любые объекты .NET) и хранить их в состоянии приложения. Предположим, что нужно поддерживать три переменных приложения внутри строго типизированного класса CarLotInfo, который определен следующим образом:

```
public class CarLotInfo
{
    public CarLotInfo(string s, string c, string m)
    {
        SalesPersonOfTheMonth = s;
        CurrentCarOnSale = c;
        MostPopularColorOnLot = m;
    }
    public string SalesPersonOfTheMonth { get; set; }
    public string CurrentCarOnSale { get; set; }
    public string MostPopularColorOnLot { get; set; }
}
```

Имея такой вспомогательный класс, можно модифицировать обработчик события Application_Start(), как показано ниже:

```
void Application_Start(Object sender, EventArgs e)
{
    // Поместить специальный объект в раздел данных приложения.
    Application["CarSiteInfo"] =
        new CarLotInfo("Chucky", "Colt", "Black");
}
```

После этого можно было бы получать доступ к информации, используя открытые поля данных внутри обработчика события Click серверной стороны для элемента управления Button по имени btnShowAppVariables:

```
protected void btnShowAppVariables_Click(object sender, EventArgs e)
{
    CarLotInfo appVars =
        ((CarLotInfo)Application["CarSiteInfo"]);
    string appState =
        string.Format("<li>Car on sale: {0}</li>",
            appVars.CurrentCarOnSale);
    appState +=
        string.Format("<li>Most popular color: {0}</li>",
            appVars.MostPopularColorOnLot);
}
```

```

appState +=
    string.Format("<li>Big shot SalesPerson: {0}</li>",
        appVars.SalesPersonOfTheMonth);
lblAppVariables.Text = appState;
}

```

Учитывая, что теперь данные о продаже машины представлены специальным типом класса, обработчик события Click элемента btnShowCarOnSale должен быть изменен, как показано ниже:

```

protected void btnShowCarOnSale_Click(object sender, EventArgs e)
{
    lblCurrCarOnSale.Text = String.Format("Sale on {0}'s today!",
        ((CarLotInfo)Application["CarSiteInfo"]).CurrentCarOnSale);
}

```

Модификация данных приложения

Во время выполнения веб-приложения можно программно обновлять или удалять любые либо все элементы данных уровня приложения с использованием членов типа HttpSessionState. Например, чтобы удалить определенный элемент данных, необходимо вызвать метод Remove(). Чтобы удалить все данные уровня приложения, следует вызвать RemoveAll().

```

private void CleanAppData()
{
    // Удалить один элемент по строковому имени.
    Application.Remove("SomeItemIDontNeed");

    // Уничтожить все данные приложения!
    Application.RemoveAll();
}

```

Если необходимо изменить значение существующего элемента данных уровня приложения, достаточно только выполнить новое присваивание соответствующему элементу данных. Предположим, что страница теперь имеет элемент управления Button, который позволяет пользователю изменять текущего рекордсмена продаж, прочитав его имя из элемента TextBox по имени txtNewSP. Обработчик события Click выглядит, как и следовало ожидать:

```

protected void btnSetNewSP_Click(object sender, EventArgs e)
{
    // Установить нового продавца.
    ((CarLotInfo)Application["CarSiteInfo"]).SalesPersonOfTheMonth
        = txtNewSP.Text;
}

```

Запустив веб-приложение, после щелчка на новой кнопке вы увидите, что элемент данных уровня приложения изменился. Более того, поскольку переменные приложения доступны всем пользователям на любой странице веб-приложения, то после запуска трех или четырех экземпляров веб-браузера можно заметить, что при изменении в одном экземпляре текущего рекордсмена продаж все остальные отобразят его после обратной отправки. На рис. 34.4 показан возможный вывод.

Имейте в виду, что если вы столкнетесь с ситуацией, в которой набор переменных уровня приложения должен изменяться как единое целое, то рискуете повредить данные (поскольку теоретически возможно, что данные уровня приложения будут изменяться как раз в тот момент, когда другой пользователь пытается их прочитать). Хотя можно было бы предпринять длинный путь и реализовать ручную логику блокировок с использованием потоковых примитивов из пространства имен System.Threading, тип

HttpApplicationState уже имеет следующие два метода, Lock() и Unlock(), которые автоматически обеспечивают безопасность потоков:

```
// Произвести безопасный доступ к взаимосвязанным данным приложения.
Application.Lock();
Application["SalesPersonOfTheMonth"] = "Maxine";
Application["CurrentBonusedEmployee"] = Application["SalesPersonOfTheMonth"];
Application.Unlock();
```

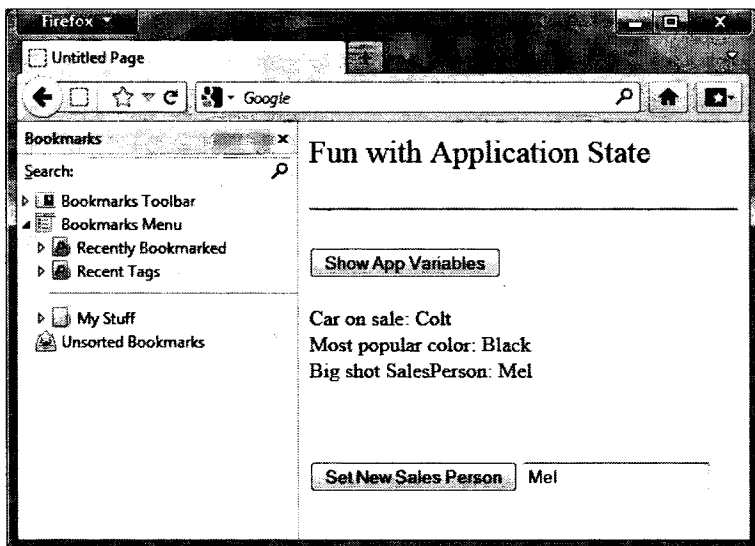


Рис. 34.4. Отображение данных приложения

Обработка останова веб-приложения

Тип `HttpApplicationState` спроектирован для поддержки значений элементов, которые он хранит до тех пор, пока не наступит одна из двух ситуаций: последний пользователь сайта выйдет по тайм-ауту (или явно) либо же кто-то остановит веб-сайт через IIS. В любом случае будет автоматически вызван метод `Application_End()` производного от `HttpApplication` типа. Внутри этого обработчика событий можно предусмотреть любой необходимый код очистки:

```
void Application_End(Object sender, EventArgs e)
{
    // Записать текущие переменные в базу
    // данных или куда-нибудь еще.
}
```

Исходный код. Веб-сайт `AppState` доступен в подкаталоге `Chapter 34`.

Работа с кешем приложения

В ASP.NET предлагается второй, более гибкий способ для обработки данных уровня приложения. Как вы помните, значения внутри объекта `HttpApplicationState` остаются в памяти до тех пор, пока веб-приложение работает и используется. Однако иногда может понадобиться хранить некоторую часть данных приложения на протя-

жении только определенного периода времени. Например, может потребоваться получить ADO.NET-объект DataSet, который действителен только в течение пяти минут. По истечении этого времени нужно будет получить обновленный объект DataSet для учета всех возможных изменений данных. Хотя формально возможно построить такую инфраструктуру средствами HttpSessionState и некоторого рода ручного мониторинга, решение этой задачи значительно упрощается за счет использования кеша приложения ASP.NET.

Как и можно было предположить по названию, объект System.Web.Caching.Cache из ASP.NET (доступный через свойство Context.Cache) позволяет определить объекты, доступные всем пользователям на всех страницах в течение фиксированного периода времени. В простейшей форме взаимодействие с кешем выглядит идентичным взаимодействию с типом HttpSessionState:

```
// Добавить элемент в кеш.
// Этот элемент *не* устареет.
Context.Cache["SomeStringItem"] = "This is the string item";
// Получить элемент из кеша.
string s = (string)Context.Cache["SomeStringItem"];
```

На заметку! Чтобы обратиться к кешу из Global.asax, следует применять свойство Context. Однако в контексте типа, производного от System.Web.UI.Page, объект Cache можно использовать напрямую через свойство Cache страницы.

Помимо индексатора, в классе System.Web.Caching.Cache определено лишь небольшое количество членов. Метод Add() можно применять для вставки в кеш нового элемента, который еще не определен (если указанный элемент уже существует, метод Add() ничего не делает). Метод Insert() также помещает элемент в кеш. Если, однако, данный элемент определен, Insert() заменяет текущий элемент новым. Учитывая, что именно такое поведение чаще всего и требуется, сосредоточим внимание исключительно на методе Insert().

Использование кеширования данных

Давайте рассмотрим пример. Создайте новый проект ASP.NET Empty Web Site по имени CacheState и добавьте в него веб-форму и файл Global.asax. Подобно элементу данных уровня приложения, поддерживаемому типом HttpSessionState, кеш может хранить любой производный от System.Object тип и часто заполняется внутри обработчика событий Application_Start(). Целью текущего примера будет автоматическое обновление содержимого DataSet каждые 15 секунд. Интересующий нас объект DataSet будет содержать текущий набор записей из таблицы Inventory базы данных AutoLot, созданной во время обсуждения ADO.NET.

С учетом сказанного выше, установите ссылку на сборку AutoLotDAL.dll (см. главу 21) и модифицируйте файл Global.asax следующим образом:

```
<%@ Application Language="C#" %>
<%@ Import Namespace = "AutoLotConnectedLayer" %>
<%@ Import Namespace = "System.Data" %>

<script runat="server">
    // Определить статическую переменную-член Cache.
    static Cache theCache;

    void Application_Start(Object sender, EventArgs e)
    {
        // Присвоить значение статической переменной theCache.
        theCache = Context.Cache;
```

```

// При запуске приложения прочитать текущие записи
// из таблицы Inventory базы данных AutoLot.
InventoryDAL dal = new InventoryDAL();
dal.OpenConnection(@"Data Source=(local)\SQLEXPRESS;" +
    "Initial Catalog=AutoLot;Integrated Security=True");
DataTable theCars = dal.GetAllInventoryAsDataTable();
dal.CloseConnection();

// Сохранить DataTable в кеше.
theCache.Insert("AppDataTable",
    theCars,
    null,
    DateTime.Now.AddSeconds(15),
    Cache.NoSlidingExpiration,
    CacheItemPriority.Default,
    new CacheItemRemovedCallback(UpdateCarInventory));
}

// Целевой метод для делегата CacheItemRemovedCallback.
static void UpdateCarInventory(string key, object item,
    CacheItemRemovedReason reason)
{
    InventoryDAL dal = new InventoryDAL();
    dal.OpenConnection(@"Data Source=(local)\SQLEXPRESS;" +
        "Initial Catalog=AutoLot;Integrated Security=True");
    DataTable theCars = dal.GetAllInventoryAsDataTable();
    dal.CloseConnection();

    // Сохранить в кеше.
    theCache.Insert("AppDataTable",
        theCars,
        null,
        DateTime.Now.AddSeconds(15),
        Cache.NoSlidingExpiration,
        CacheItemPriority.Default,
        new CacheItemRemovedCallback(UpdateCarInventory));
}
...
</script>

```

Для начала обратите внимание на определение статической переменной-члена `Cache`. Причина в том, что были определены два статических члена, которым необходим доступ к объекту `Cache`. Вспомните, что статические методы не имеют доступа к унаследованным членам, а потому использовать свойство `Context` нельзя!

Внутри обработчика событий `Application_Start()` осуществляется наполнение объекта `DataTable` и помещение его в кеш приложения. Как и можно было предположить, метод `Context.Cache.Insert()` имеет несколько перегруженных версий. В этом случае указывается значение для каждого параметра. Взгляните на следующий прокомментированный вызов `Insert()`:

```

theCache.Insert("AppDataTable", // Имя для идентификации элемента в кеше.
    theCars,                    // Объект, помещаемый в кеш.
    null,                       // Есть ли зависимости у этого объекта?
    DateTime.Now.AddSeconds(15), // Абсолютное время устаревания.
    Cache.NoSlidingExpiration,   // Не использовать скользящее устаревание (см. ниже).
    CacheItemPriority.Default,    // Уровень приоритета элемента кеша.

    // Делегат для события CacheItemRemove.
    new CacheItemRemovedCallback(UpdateCarInventory));

```

Первые два параметра просто формируют пару “имя/значение” для элемента. Третий параметр позволяет определить объект `CacheDependency` (который в данном случае равен `null`, поскольку `DataTable` ни от чего не зависит).

Параметр `DateTime.Now.AddSeconds(15)` задает абсолютное время устаревания. Это значит, что элемент будет неизбежно удален из кеша через 15 секунд. Абсолютное время устаревания удобно для элементов данных, которые постоянно должны обновляться (таких как биржевые показатели).

Параметр `Cache.NoSlidingExpiration` указывает на то, что элемент кеша не использует скользящее устаревание. Скользящее устаревание — это способ сохранения элемента в кеше в течение, как минимум, указанного времени. Например, если установить значение скользящего устаревания в 60 секунд для элемента кеша, он будет находиться там, по меньшей мере, одну минуту. Если любая веб-страница обратится к элементу кеша в течение этого времени, таймер сбрасывается, и элемент кеша существует еще 60 секунд. Если в течение этого времени никакого обращения к нему не последует, элемент из кеша удаляется. Скользящее устаревание удобно для данных, которые могут быть дорого (в смысле затрат времени) генерировать, но которые не слишком часто используются веб-страницами.

Обратите внимание, что для определенного элемента кеша указывать одновременно абсолютное и скользящее устаревание не допускается. Должно устанавливаться либо абсолютное (с помощью `Cache.NoSlidingExpiration`), либо скользящее (посредством `Cache.NoAbsoluteExpiration`) устаревание.

Наконец, как видно из сигнатуры метода `UpdateCarInventory()`, делегат `CacheItemRemovedCallback` может вызывать только методы, соответствующие следующей сигнатуре:

```
void UpdateCarInventory(string key, object item, CacheItemRemovedReason reason)
{
}
```

Итак, теперь при запуске приложения объект `DataTable` заполняется и кешируется. Каждые 15 секунд объект `DataTable` очищается, обновляется и заново вставляется в кеш. Чтобы увидеть это в действии, понадобится создать страницу, которая позволит осуществлять некоторое взаимодействие с пользователем.

Модификация файла *.aspx

На рис. 34.5 показан пользовательский интерфейс, который позволяет вводить необходимые данные для вставки новой записи в базу данных (посредством различных элементов управления `TextBox`). Обработчик события `Click` для единственного элемента управления `Button` будет закодирован для поддержки манипуляций в базе данных. Наконец, элемент управления `GridView` в нижней части страницы будет применяться для отображения набора текущих записей в таблице `Inventory`.

В обработчике события `Load` страницы сконфигурируйте элемент управления `GridView` для отображения текущего содержимого кешированного объекта `DataTable` при первом получении страницы пользователем (не забудьте импортировать в файл кода пространства имен `System.Data` и `AutoLotConnectedLayer`):

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        carsGridView.DataSource = (DataTable)Cache["AppDataTable"];
        carsGridView.DataBind();
    }
}
```

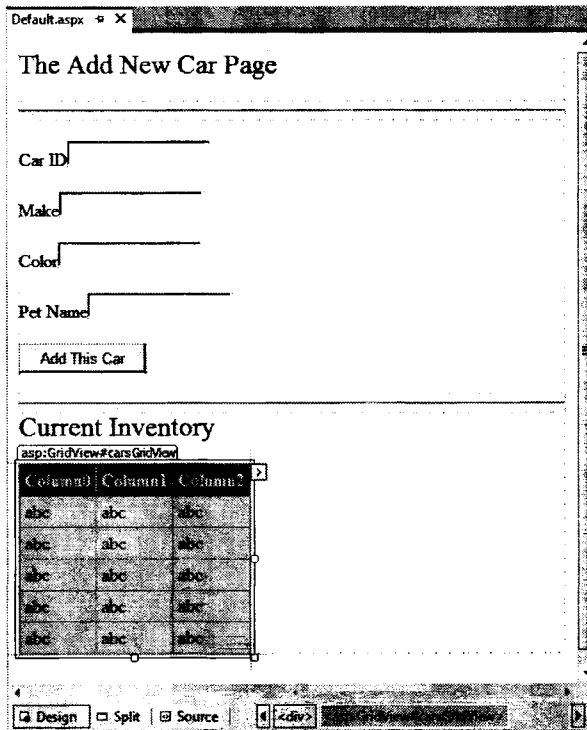



Рис. 34.5. Графический пользовательский интерфейс приложения, в котором применяется кеш

В обработчике события Click кнопки Add This Car (Добавить этот автомобиль) вставьте новую запись в базу данных AutoLot, используя тип InventoryDAL. Как только запись будет вставлена, вызовите вспомогательную функцию по имени RefreshGrid(), которая обновит пользовательский интерфейс:

```
protected void btnAddCar_Click(object sender, EventArgs e)
{
    // Обновить таблицу Inventory и вызвать RefreshGrid().
    InventoryDAL dal = new InventoryDAL();
    dal.OpenConnection(@"Data Source=(local)\SQLEXPRESS;" +
        "Initial Catalog=AutoLot;Integrated Security=True");
    dal.InsertAuto(int.Parse(txtCarID.Text), txtCarColor.Text,
        txtCarMake.Text, txtCarPetName.Text);
    dal.CloseConnection();
    RefreshGrid();
}

private void RefreshGrid()
{
    InventoryDAL dal = new InventoryDAL();
    dal.OpenConnection(@"Data Source=(local)\SQLEXPRESS;" +
        "Initial Catalog=AutoLot;Integrated Security=True");
    DataTable theCars = dal.GetAllInventoryAsDataTable();
    dal.CloseConnection();

    carsGridView.DataSource = theCars;
    carsGridView.DataBind();
}
```

Теперь, чтобы проверить использование кеша, начните с запуска текущей программы (нажав <Ctrl+F5>) и скопируйте URL, появившийся в браузере, в системный буфер обмена. Затем запустите второй экземпляр браузера и вставьте скопированный URL в его строку адреса. После этого вы должны видеть на экране два экземпляра браузера, оба с загруженной страницей Default.aspx, отображающей идентичные данные.

В первом экземпляре браузера добавьте новую запись об автомобиле. Очевидно, что в результате получится обновленный GridView, видимый в браузере, который инициировал обратную отправку.

Во втором экземпляре браузера щелкните на кнопке обновления (или нажмите <F5>). Вы не сразу увидите новый элемент, учитывая, что обработчик события Page_Load читает непосредственно из кеша. (Если вы его видите, значит, истекло 15 секунд. Либо действуйте быстрее, либо увеличьте период времени, в течение которого DataTable остается в кеше.) Подождите несколько секунд и еще раз щелкните на кнопке обновления во втором экземпляре браузера. Теперь вы должны увидеть новый элемент, учитывая, что DataTable в кеше устарел, и целевой метод делегата CacheItemRemoveCallback автоматически обновил кешированный объект DataTable.

Как видите, главное преимущество типа Cache заключается в том, что появляется шанс отреагировать, когда элемент удаляется из кеша. В рассмотренном примере определенно можно было бы обойтись без применения Cache, а просто заставить обработчик событий Page_Load() всегда читать непосредственно из базы данных AutoLot (правда, работа страницы в этом случае замедлилась бы). Тем не менее, идея должна быть ясна: кеш всегда позволяет автоматически обновлять данные, используя механизм кеширования.

Исходный код. Веб-сайт CacheState доступен в подкаталоге Chapter 34.

Поддержка данных сеанса

На этом рассмотрение данных уровня приложения и кешированных данных завершено. Далее давайте ознакомимся с ролью данных, специфичных для пользователя. Как упоминалось ранее в главе, *сеанс* — это не более чем взаимодействие конкретного пользователя с веб-приложением, представленное уникальным объектом HttpSessionState. Чтобы поддерживать информацию о состоянии для конкретного пользователя, можно применять свойство Session в классе веб-страницы или в файле Global.asax. Классический пример необходимости поддерживать данные пользователя — корзина покупок. Если 10 человек зашли в онлайн-электронный магазин, каждый из них будет иметь уникальный набор наименований товаров, который он намерен приобрести, и эти данные должны поддерживаться.

Когда новый пользователь входит в веб-приложение, исполняющая среда .NET автоматически присваивает ему уникальный идентификатор сеанса, служащий для идентификации этого конкретного пользователя. Каждый идентификатор сеанса получает свой экземпляр типа HttpSessionState для хранения специфичных для пользователя данных. Вставка или извлечение данных сеанса синтаксически идентична манипуляциям с данными приложения, например:

```
// Добавить/извлечь данные сеанса для текущего пользователя.
Session["DesiredCarColor"] = "Green";
string color = (string) Session["DesiredCarColor"];
```

В Global.asax можно перехватывать момент начала и конца сеанса через обработчики событий Session_Start() и Session_End(). Внутри Session_Start() можно свободно создавать любые элементы данных, специфичные для пользователя, в то время

как `Session_End()` позволяет выполнять любую работу, которая может понадобиться при закрытии пользовательского сеанса.

```
<%@ Application Language="C#" %>
...
void Session_Start(Object sender, EventArgs e)
{
    // Новый сеанс! При необходимости выполнить подготовительные действия.
}

void Session_End(Object sender, EventArgs e)
{
    // Пользователь вышел или отключен по тайм-ауту. При необходимости выполнить очистку.
}
```

Подобно состоянию приложения, состояние сеанса может хранить объекты любых производных от `System.Object` типов, включая специальные классы. Например, предположим, что создан новый проект ASP.NET Empty Web Site (по имени `SessionState`), в котором определен класс `UserShoppingCart`:

```
public class UserShoppingCart
{
    public string DesiredCar {get; set;}
    public string DesiredCarColor {get; set;}
    public float DownPayment {get; set;}
    public bool IsLeasing {get; set;}
    public DateTime DateOfPickUp {get; set;}

    public override string ToString()
    {
        return string.Format
            ("Car: {0}<br>Color: {1}<br>$ Down: {2}<br>Lease: {3}<br>Pick-up Date: {4}",
             DesiredCar, DesiredCarColor, DownPayment, IsLeasing,
             DateOfPickUp.ToShortDateString());
    }
}
```

Вставьте файл `Global.asax`. Внутри обработчика `Session_Start()` теперь можно присвоить каждому пользователю новый экземпляр класса `UserShoppingCart`:

```
void Session_Start(Object sender, EventArgs e)
{
    Session["UserShoppingCartInfo"] = new UserShoppingCart();
}
```

Во время посещения пользователем веб-страниц можно брать экземпляры `UserShoppingCart` и заполнять его поля специфичными для пользователя данными. Например, предположим, что имеется простая страница `*.aspx`, которая определяет набор элементов управления для ввода, соответствующих полям типа `UserShoppingCart`, элемент `Button`, используемый для установки значений, и два элемента `Label`, которые будут отображать идентификатор сеанса пользователя и информацию о сеансе (рис. 34.6).

Обработчик события `Click` серверной стороны для элемента управления `Button` достаточно прост (он извлекает значения из элементов `TextBox` и отображает данные корзины покупок в элементе управления `Label`):

```
protected void btnSubmit_Click(object sender, EventArgs e)
{
    // Установить предпочтения текущего пользователя.
    UserShoppingCart cart =
        (UserShoppingCart)Session["UserShoppingCartInfo"];
```

```

cart.DateOfPickUp = myCalendar.SelectedDate;
cart.DesiredCar = txtCarMake.Text;
cart.DesiredCarColor = txtCarColor.Text;
cart.DownPayment = float.Parse(txtDownPayment.Text);
cart.IsLeasing = chkIsLeasing.Checked;
lblUserInfo.Text = cart.ToString();
Session["UserShoppingCartInfo"] = cart;
}

```

Внутри `Session_End()` можно сохранить поля `UserShoppingCart` в базе данных или где-нибудь еще (как будет показано в конце этой главы, API-интерфейс `Profile` делает это автоматически). Кроме того, можно реализовать `Session_Error()` для перехвата любого ошибочного ввода (либо применить различные элементы управления проверкой достоверности на странице `Default.aspx` для обработки подобного рода ошибок пользователя).

В любом случае, запустив два или три экземпляра браузера с одним и тем же URL, обнаружится, что каждый пользователь может наполнять собственную корзину покупок, которая отображается на его уникальный экземпляр `HttpSessionState`.

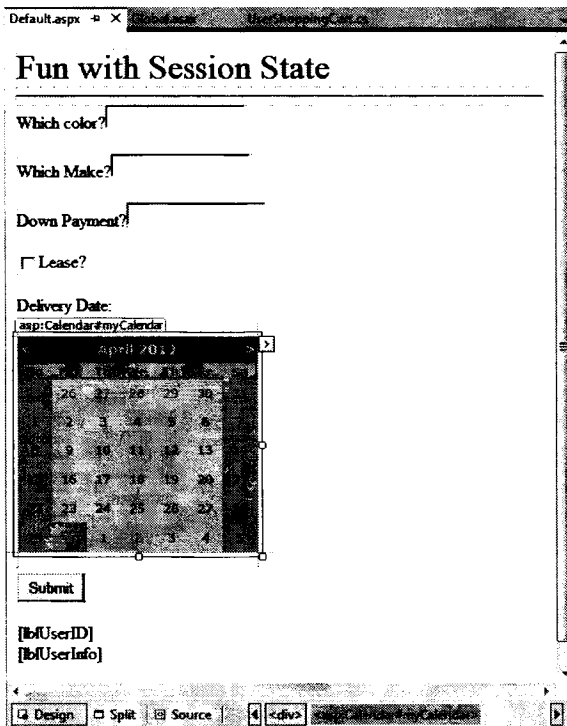


Рис. 34.6. Графический пользовательский интерфейс приложения, в котором применяется информация сеанса

Дополнительные члены класса `HttpSessionState`

Помимо индексатора, в классе `HttpSessionState` определен ряд других интересных членов. Свойство `SessionID` возвращает уникальный идентификатор текущего пользователя. Если хотите увидеть в этом примере автоматически назначенный идентификатор сеанса, обработайте событие `Load` страницы, как показано ниже:

```
protected void Page_Load(object sender, EventArgs e)
{
    lblUserID.Text = string.Format("Here is your ID: {0}",
        Session.SessionID);
}
```

Методы `Remove()` и `RemoveAll()` могут применяться для очистки элементов пользовательского экземпляра `HttpSessionState`, например:

```
Session.Remove("SomeItemWeDontNeedAnymore");
```

В классе `HttpSessionState` также определен набор членов, которые управляют политикой устаревания текущего сеанса. По умолчанию каждый пользователь располагает 20 минутами отсутствия активности, прежде чем объект `HttpSessionState` будет уничтожен. Таким образом, если пользователь входит в веб-приложение (и, следовательно, получает уникальный идентификатор сеанса), но не возвращается на сайт в течение 20 минут, то исполняющая среда предполагает, что пользователь больше не заинтересован в сайте и уничтожает его данные сеанса. Период устаревания сеанса можно изменить, установив его для каждого пользователя с помощью свойства `Timeout`. Наиболее подходящим местом для этого является метод `Session_Start()`:

```
void Session_Start(Object sender, EventArgs e)
{
    // Каждый пользователь получает 5 минут на отсутствие активности.
    Session.Timeout = 5;
    Session["UserShoppingCartInfo"]
        = new UserShoppingCart();
}
```

На заметку! Если настраивать значение `Timeout` для каждого пользователя не нужно, можете изменить 20-минутное стандартное значение для всех пользователей через атрибут `timeout` элемента `<sessionState>` внутри файла `web.config` (рассматривается в конце этой главы).

Преимущество свойства `Timeout` заключается в возможности назначения специфического тайм-аута отдельно каждому пользователю. Например, предположим, что создано веб-приложение, которое позволяет пользователям платить дифференцированную плату за различные уровни членства. Вы могли бы указать, что привилегированные пользователи должны иметь тайм-аут длительностью в один час, в то время как обычные — только 30 секунд. Такая возможность вызывает вопрос: как запомнить специфическую для пользователя информацию (вроде текущего уровня членства) между визитами на сайт? Один из вариантов предусматривает применение типа `HttpCookie`.

Исходный код. Веб-сайт `SessionState` доступен в подкаталоге `Chapter 34`.

Cookie-наборы

Следующий прием управления состоянием состоит в хранении постоянных данных внутри *cookie-наборов*, которые часто реализуются в виде текстового файла (или набора файлов), расположенного на машине пользователя. Когда пользователь входит на определенный сайт, браузер проверяет наличие на пользовательской машине *cookie-файла* для заданного URL, и если он есть, добавляет его данные к запросу HTTP.

Принимающая веб-страница серверной стороны может затем прочитать *cookie-данные* для создания графического интерфейса, основанного на предпочтениях конкретного пользователя. Наверняка вы замечали, что после посещения одного из любимых веб-сайтов он каким-то образом “запоминает”, какое содержимое вас интересует.

Причина (отчасти) в том, что в cookie-наборах, хранимых на вашем компьютере, содержится информация, касающаяся этого веб-сайта.

На заметку! Точное местоположение cookie-файлов зависит от используемого браузера и установленной операционной системы.

Очевидно, что содержимое конкретного cookie-файла варьируется в зависимости от веб-сайта, но имейте в виду, что в конечном итоге это все равно будет текстовый файл. Поэтому cookie-набор — худший выбор для хранения чувствительной информации о текущем пользователе (такой как номер кредитной карточки, пароль и т.п.). Даже если вы предпримете усилия, чтобы зашифровать данные, есть шанс, что настойчивый хакер расшифрует их и воспользуется в преступных целях. Но в любом случае, cookie-наборы играют определенную роль в разработке веб-приложений, поэтому давайте посмотрим, как ASP.NET работает с этим приемом управления состоянием.

Создание cookie-наборов

Прежде всего, следует отметить, что cookie-наборы ASP.NET могут быть сконфигурированы как постоянные или временные. *Постоянные* cookie-наборы обычно известны как классическое определение cookie-данных, которые устанавливают пары “имя/значение”, физически сохраняемые на жестком диске пользователя. *Временные* cookie-наборы (также называемые *сеансовыми* cookie-наборами) содержат те же данные, что и постоянные cookie-наборы, но пары “имя/значение” никогда не сохраняются на жестком диске пользователя, а вместо этого существуют *только* в течение периода, пока браузер открыт. Как только пользователь закрывает браузер, все данные, содержащиеся в сеансовом cookie-наборе, уничтожаются.

`System.Web.HttpCookie` — это класс, представляющий серверную сторону cookie-данных (постоянных или временных). Для создания нового cookie-набора в коде веб-страницы необходимо обратиться к свойству `Response.Cookies`. Как только новый элемент `HttpCookie` вставлен во внутреннюю коллекцию, пары “имя/значение” отправляются браузеру внутри заголовка HTTP.

Чтобы наглядно увидеть поведение cookie-наборов, создайте новый проект ASP.NET Empty Web Site по имени `CookieStateApp` и вставьте в него веб-форму с пользовательским интерфейсом, показанным на рис. 34.7.

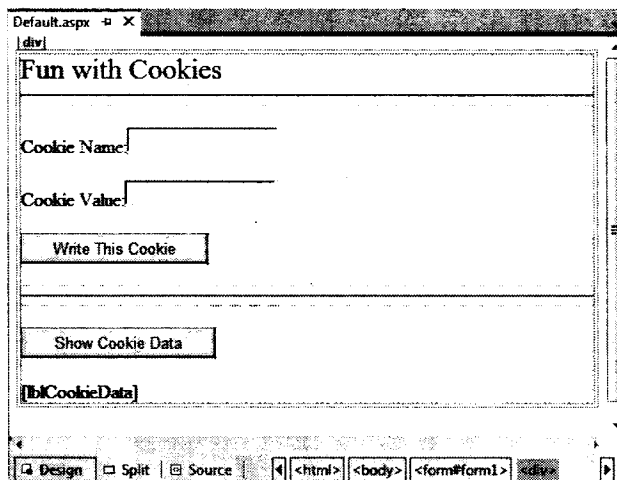


Рис. 34.7. Пользовательский интерфейс `CookieStateApp`

Внутри обработчика события Click кнопки Write This Cookie (Записать этот cookie-набор) постройте новый объект `HttpCookie` и добавьте его в коллекцию `Cookie`, представленную свойством `HttpRequest.Cookies`. Имейте в виду, что данные не будут сохраняться на жестком диске пользователя, если только вы явно не установите срок хранения в свойстве `HttpCookie.Expires`. Таким образом, следующая реализация создаст временный cookie-набор, который будет уничтожен, когда пользователь закроет браузер:

```
protected void btnCookie_Click(object sender, EventArgs e)
{
    // Создать временный cookie-набор.
    HttpCookie theCookie =
        new HttpCookie(txtCookieName.Text,
            txtCookieValue.Text);
    Response.Cookies.Add(theCookie);
}
```

А приведенный ниже код сгенерирует постоянный cookie-набор, который будет действителен в течение трех месяцев, начиная с текущей даты:

```
protected void btnCookie_Click(object sender, EventArgs e)
{
    HttpCookie theCookie =
        new HttpCookie(txtCookieName.Text,
            txtCookieValue.Text);
    theCookie.Expires = DateTime.Now.AddMonths(3);
    Response.Cookies.Add(theCookie);
}
```

Чтение входящих cookie-данных

Вспомните, что браузер полностью отвечает за доступ к постоянным cookie-наборам при переходе на ранее посещенную страницу. Если браузер решает отправить cookie-набор серверу, доступ к входящим cookie-данным в коде страницы *.aspx можно получить через свойство `HttpRequest.Cookies`. В целях иллюстрации реализуйте обработчик события Click для кнопки Show Cookie Data (Показать cookie-данные) следующим образом:

```
protected void btnShowCookie_Click(object sender, EventArgs e)
{
    string cookieData = "";
    foreach (string s in Request.Cookies)
    {
        cookieData +=
            string.Format("<li><b>Name</b>: {0}, <b>Value</b>: {1}</li>",
                s, Request.Cookies[s].Value);
    }
    lblCookieData.Text = cookieData;
}
```

Теперь, запустив приложение и щелкнув на кнопке Show Cookie Data, вы увидите, что cookie-данные действительно были опрарвлены браузером и успешно доступны в коде *.aspx на стороне сервера.

Роль элемента <sessionState>

К этому моменту главы вы ознакомились с различными способами запоминания информации о пользователях. Как было показано, программное управление данными состояния представления и приложения, кеша, сеанса и cookie-набора осуществляется более или менее сходным образом (через индекатор класса). Кроме того, в `Global.asax` имеются методы, которые позволяют перехватывать и реагировать на события, возникающие во время жизни веб-приложения.

По умолчанию ASP.NET сохраняет данные сеанса внутри процесса. Положительной стороной является предельно возможная скорость доступа к информации. Однако отрицательный момент такого решения заключается в том, что если домен приложения терпит аварию (по любой причине), то все пользовательские данные состояния разрушаются. Более того, когда данные состояния хранятся во внутрипроцессной сборке *.dll, нет возможности взаимодействовать с сетевой веб-фермой. Такой стандартный режим хранения работает достаточно хорошо, если веб-приложение развернуто на единственном веб-сервере. Однако, как и можно было предположить, подобная модель не идеальна для фермы веб-серверов, учитывая, что состояние сеанса “удерживается” внутри определенного домена приложения.

Хранение данных сеанса на сервере состояния сеансов ASP.NET

В ASP.NET исполняющую среду можно инструктировать о необходимости размещения сборки *.dll состояния сеанса в суррогатном процессе, который называется сервером состояния сеансов ASP.NET (`aspnet_state.exe`). При этом сборку *.dll можно выгрузить из `aspnet_wp.exe` в отдельный файл *.exe, который может находиться на любой из машин веб-фермы. Даже если вы намерены запускать процесс `aspnet_wp.exe` на той же машине, что и веб-сервер, будет получен выигрыш от вынесения данных состояния в отдельный процесс (т.к. это более надежно).

Чтобы использовать сервер состояния сеансов, прежде всего, запустите Windows-службу `aspnet_state.exe` на целевой машине посредством ввода следующей команды в командной строке разработчика Visual Studio (обратите внимание, что для этого понадобятся права администратора):

```
net start aspnet_state
```

В качестве альтернативы Windows-службу `aspnet_state.exe` можно запустить через значок **Services** (Службы) в группе **Administrative Tools** (Администрирование) панели управления, как показано на рис. 34.8.

Ключевое преимущество такого подхода связано с возможностью конфигурирования `aspnet_state.exe` в окне **Properties** на автоматический запуск при начальной загрузке операционной системы на машине. В любом случае, как только сервер состояния сеансов запущен, добавьте в файл `web.config` следующий элемент `<sessionState>`:

```
<system.web>
  <sessionState
    mode="StateServer"
    stateConnectionString="tcpip=127.0.0.1:42626"
    sqlConnectionString="data source=127.0.0.1;Trusted_Connection=yes"
    cookieless="false"
    timeout="20"
  />
  ...
</system.web>
```

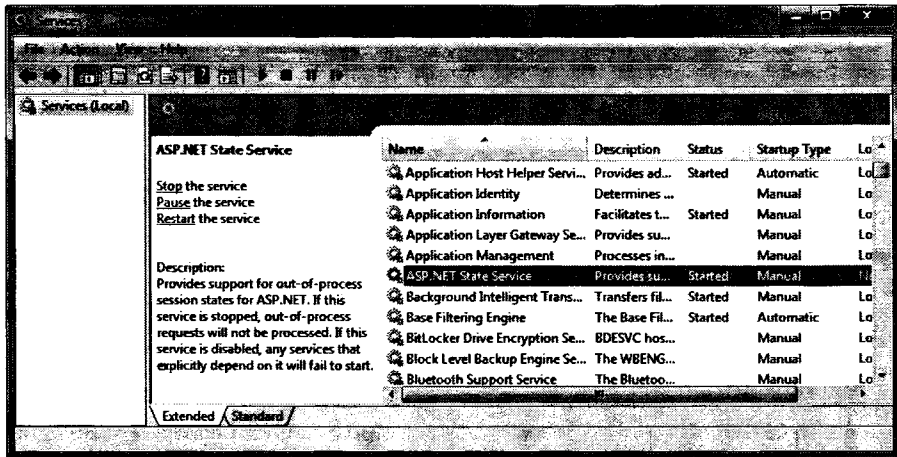



Рис. 34.8. Запуск Windows-службы aspnet_state.exe через значок Services

Вот и все! Начиная с этого момента, среда CLR будет хранить данные о сеансе внутри процесса aspnet_state.exe. Таким образом, если домен приложения, в котором размещается веб-приложение, потерпит аварию, данные сеанса останутся сохраненными. Обратите внимание, что элемент <sessionState> может также поддерживать атрибут stateConnectionString. Стандартное значение TCP/IP-адреса (127.0.0.1) указывает на локальную машину. Если же вы хотите, чтобы исполняющая среда .NET использовала службу aspnet_state.exe, функционирующую на другой машине (т.е. случай веб-фермы), можете соответствующим образом изменить это значение.

Хранение информации о сеансах в выделенной базе данных

И, наконец, если требуется максимальная степень изоляции и надежности веб-приложения, можно заставить исполняющую среду хранить все данные о состоянии сеанса внутри Microsoft SQL Server. Необходимое для этого изменение файла web.config выглядит очень просто:

```
<sessionState
  mode="SQLServer"
  stateConnectionString="tcpip=127.0.0.1:42626"
  sqlConnectionString="data source=127.0.0.1;Trusted_Connection=yes"
  cookieless="false"
  timeout="20"
/>
```

Однако перед тем как попробовать запустить ассоциированное веб-приложение, следует удостовериться, что целевая машина (указанная в атрибуте sqlConnectionString) правильно сконфигурирована.

Во время установки .NET Framework 4.5 SDK (или Visual Studio) предлагаются два файла с именами InstallSqlState.sql и UninstallSqlState.sql, расположенные по умолчанию в каталоге C:\Windows\Microsoft.NET\Framework\<версия>. На целевой машине потребуется запустить файл InstallSqlState.sql, используя инструмент наподобие Microsoft SQL Server Management Studio (который поставляется вместе с Microsoft SQL Server).

После выполнения SQL-сценария InstallSqlState.sql вы обнаружите новую базу данных SQL Server (ASPState), которая содержит набор хранимых процедур, вызываемых исполняющей средой ASP.NET, и набор таблиц, используемых для хранения данных

сеансов. (Кроме того, база данных tempdb обновляется набором таблиц, предназначенных для подкачки.) Как и можно было предположить, конфигурация веб-приложения с хранением информации сеансов в SQL Server является самой медленной из всех возможных вариантов. Преимущество заключается в том, что пользовательские данные хранятся надежнее всего (даже если происходит перезагрузка веб-сервера).

На заметку! Если для хранения сеансовых данных используется сервер состояния сеансов ASP.NET или SQL Server, необходимо удостовериться, что все специальные типы, помещенные в объект HttpSessionState, помечены атрибутом [Serializable].

Введение в API-интерфейс ASP.NET Profile

К настоящему моменту вы изучили различные приемы, позволяющие запоминать данные на уровнях пользователей и приложения. Однако многие веб-сайты требуют возможности сохранения пользовательской информации между сеансами. Например, предположим, что необходимо предоставить пользователям средство для поддержки своих учетных записей на сайте. Или, скажем, нужно сохранять экземпляры ShoppingCart между сеансами (для сайта онлайн-магазина). Или, возможно, требуется сохранять базовые пользовательские предпочтения (темы и т.п.).

Хотя для хранения такой информации определенно можно построить специальную базу данных (с несколькими хранимыми процедурами), впоследствии придется создавать специальную библиотеку кода для взаимодействия с такими объектами базы данных. Это не обязательно будет сложной задачей, но недостаток подобного решения заключается в том, что построение всей инфраструктуры возлагается полностью на вас.

Чтобы помочь справиться с такими ситуациями, в состав ASP.NET входит готовый API-интерфейс управления профилями пользователей (ASP.NET Profile API) и соответствующая система базы данных. В дополнение к обеспечению необходимой инфраструктуры, API-интерфейс Profile также позволяет определять данные, которые должны храниться непосредственно в файле web.config (с целью упрощения); однако можно хранить и любой тип, помеченный атрибутом [Serializable]. Перед тем как погрузиться в эту тему, давайте посмотрим, где API-интерфейс Profile будет хранить указанные данные.

База данных ASPNETDB.mdf

Каждый веб-сайт ASP.NET, построенный средствами Visual Studio, может поддерживать папку App_Data. По умолчанию API-интерфейс Profile (а также другие службы наподобие API-интерфейса ролей ASP.NET, который здесь не рассматривается) конфигурируется на работу с локальной базой данных SQL Server по имени ASPNETDB.mdf, расположенной в папке App_Data. Это стандартное поведение определяется настройками в файле machine.config для текущей установки .NET на машине. В действительности, когда код использует службу ASP.NET, требующую папки App_Data, файл ASPNETDB.mdf автоматически создается на лету, если он до этого не существовал.

Если же необходимо, чтобы исполняющая среда ASP.NET взаимодействовала с файлом ASPNETDB.mdf, расположенным на другой машине в сети, или предпочтительнее установить эту базу данных на экземпляре SQL Server 7.0 (или выше), потребуется вручную построить ASPNETDB.mdf с помощью утилиты командной строки aspnet_regsql.exe. Подобно любой хорошей утилите командной строки, aspnet_regsql.exe поддерживает множество опций; однако если запустить ее без аргументов (в окне командной строки разработчика):

```
aspnet_regsql
```

откроется мастер с графическим пользовательским интерфейсом, который поможет в прохождении всего процесса создания и установки базы данных ASPNETDB.mdf на выбранной машине (и версии SQL Server).

Теперь предположим, что сайт не использует локальную копию базы данных в папке App_Data. Тогда последним шагом должна быть модификация файла web.config для указания на уникальное местоположение ASPNETDB.mdf. Предположим, что файл ASPNETDB.mdf установлен на машине по имени ProductionServer. Чтобы указать API-интерфейсу Profile, где по умолчанию расположены необходимые элементы базы данных, необходимо воспользоваться следующим фрагментом файла machine.config (для изменения этих стандартных настроек можно добавить специальный файл web.config):

```
<configuration>
  <connectionStrings>
    <add name="LocalSqlServer"
      connectionString="Data Source=ProductionServer;Integrated
      Security=SSPI;Initial Catalog=aspnetdb;"
      providerName="System.Data.SqlClient"/>
  </connectionStrings>
  <system.web>
    <profile>
      <providers>
        <clear/>
        <add name="AspNetSqlProfileProvider"
          connectionStringName="LocalSqlServer"
          applicationName="/"
          type="System.Web.Profile.SqlProfileProvider, System.Web,
          Version=4.0.0.0,
          Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" />
      </providers>
    </profile>
  </system.web>
</configuration>
```

Подобно большинству файлов *.config, он выглядит намного сложнее, чем есть на самом деле. В основном, здесь определяется элемент <connectionString> с необходимыми данными, за которым следует именованный экземпляр SqlProfileProvider (это стандартный поставщик, используемый независимо от физического местоположения ASPNETDB.mdf).

На заметку! Для простоты будем предполагать, что используется автоматически сгенерированная база данных ASPNETDB.mdf, расположенная в подкаталоге App_Data веб-приложения.

Определение пользовательского профиля в web.config

Как уже упоминалось, пользовательский профиль определяется в файле web.config. Изящество этого подхода состоит в том, что с этим профилем можно взаимодействовать в строго типизированной манере, используя в файлах кода унаследованное свойство Property. Чтобы проиллюстрировать это, создайте новый проект ASP.NET Empty Web Site по имени FunWithProfiles, добавьте в него новый файл *.aspx и откройте файл web.config для редактирования.

Нашей целью будет создание профиля, который моделирует домашние адреса пользователей, находящихся в сеансе, а также общее количество их обращений к этому сайту. Не должен вызывать удивление тот факт, что данные профиля определяются внутри элемента <profile> с применением множества пар типа "имя/значение". Рассмотрим следующий профиль, созданный в контексте элемента <system.web>:

```

<profile>
  <properties>
    <add name="StreetAddress" type="System.String" />
    <add name="City" type="System.String" />
    <add name="State" type="System.String" />
    <add name="TotalPost" type="System.Int32" />
  </properties>
</profile>

```

Здесь для каждого элемента профиля указано имя и тип данных CLR (разумеется, можно было бы добавить дополнительные элементы для почтового кода, имени и т.д., но и без того идея должна быть ясна). Строго говоря, атрибут `type` не обязателен; однако по умолчанию принимается `System.String`. Как и следовало ожидать, существует много других атрибутов, которые могут быть указаны в элементе профиля для дальнейшего уточнения способа хранения этой информации в `ASPNETDB.mdf`. В табл. 34.4 описаны некоторые основные атрибуты.

Таблица 34.4. Избранные атрибуты данных профиля

Атрибуты	Примеры значений	Описание
<code>allowAnonymous</code>	<code>true</code> <code>false</code>	Ограничивает или разрешает анонимный доступ к данному значению. Если этот атрибут установлен в <code>false</code> , анонимные пользователи не имеют доступа к этому значению профиля
<code>defaultValue</code>	Строка	Значение, которое должно быть возвращено, если свойство еще не было явно установлено
<code>name</code>	Строка	Уникальный идентификатор для данного свойства
<code>provider</code>	Строка	Поставщик, используемый для управления этим значением. Переопределяет установку <code>default Provider</code> в <code>web.config</code> или <code>machine.config</code>
<code>readOnly</code>	<code>true</code> <code>false</code>	Ограничивает или разрешает доступ для записи (стандартным значением является <code>false</code> , т.е. не только для чтения)
<code>serializeAs</code>	<code>String</code> <code>XML</code> <code>Binary</code>	Формат значений при записи в хранилище данных
<code>type</code>	Элементарный тип Тип, определяемый пользователем	Элементарный тип или класс .NET. Имена классов должны быть полностью заданными (например, <code>MyApp.UserData.ColorPrefs</code>)

Мы увидим некоторые из этих атрибутов в действии, когда модифицируем текущий профиль. А пока давайте посмотрим, как программно обращаться к этим данным в коде страниц.

Программный доступ к данным профиля

Вспомните, что общее предназначение API-интерфейса ASP.NET Profile заключается в автоматизации процесса записи (и чтения) информации в выделенную базу данных. Чтобы попробовать это на практике, модифицируйте пользовательский интерфейс страницы `Default.aspx`, добавив набор элементов `TextBox` (и описательных элементов `Label`) для ввода адреса пользователя: улицы, города и штата. Кроме того, добавьте элемент управления `Button` (по имени `btnSubmit`) и еще один элемент `Label` (по имени `lblUserData`), которые будут служить для отображения постоянно хранимых данных (рис. 34.9).

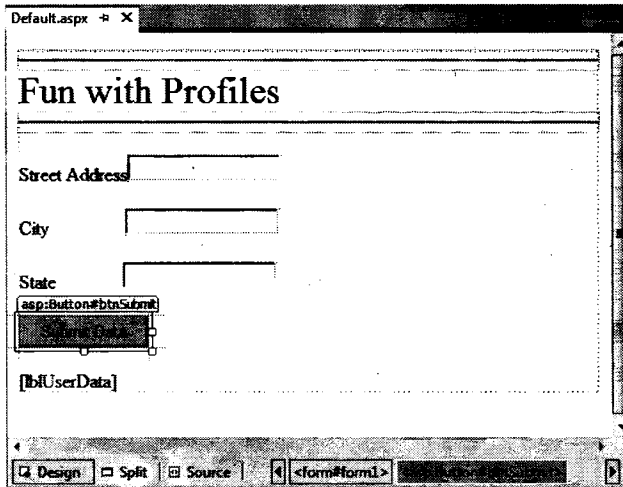


Рис. 34.9. Пользовательский интерфейс страницы Default.aspx веб-сайта FunWithProfiles

Внутри обработчика события Click кнопки Submit Data (Отправить данные) воспользуйтесь унаследованным свойством Profile для сохранения каждой единицы данных профиля на основе информации, введенной пользователем в соответствующем поле TextBox. Сохранив каждый элемент данных в ASPNETDB.mdf, прочитайте их из базы данных и сформатируйте в строку, отображаемую в элементе lblUserData типа Label. И, наконец, обработайте события страницы Load и отобразите ту же информацию в элементе Label. Таким образом, когда пользователь зайдет на страницу, он увидит текущие настройки. Ниже приведен полный код.

```
public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        GetUserAddress();
    }
    protected void btnSubmit_Click(object sender, EventArgs e)
    {
        // Здесь происходит запись в базу данных.
        Profile.StreetAddress = txtStreetAddress.Text;
        Profile.City = txtCity.Text;
        Profile.State = txtState.Text;

        // Получить настройки из базы данных.
        GetUserAddress();
    }
    private void GetUserAddress()
    {
        // Здесь происходит чтение из базы данных.
        lblUserData.Text = String.Format("You live here: {0}, {1}, {2}",
            Profile.StreetAddress, Profile.City, Profile.State);
    }
}
```

Теперь, запустив страницу, вы заметите некоторую задержку при первом запросе Default.aspx. Причина в том, что в это время создается на лету файл ASPNETDB.mdf и помещается в папку App_Data (в чем можно убедиться, обновив окно Solution Explorer и заглянув в папку App_Data).

Кроме того, вы обнаружите, что при первом входе на эту страницу элемент Label по имени lblUserData не отображает никаких данных, поскольку они еще не добавлены в соответствующую таблицу базы ASPNETDB.mdf. Как только вы введете значения в элементах управления TextBox и выполните обратную отправку на сервер, упомянутый элемент Label будет заполнен постоянными данными.

Теперь перейдем к самому интересному аспекту этой технологии. Если вы закроете браузер и перезапустите веб-сайт, то обнаружите, что ранее введенные данные профиля действительно сохранились, поскольку в Label отображается корректная информация. Это вызывает очевидный вопрос: как они были сохранены?

В рассматриваемом примере API-интерфейс Profile использует сетевую идентификацию Windows, которая получена из текущих регистрационных данных машины. Однако при построении публичных веб-сайтов (где пользователи не относятся к определенному домену) необходимо обеспечить интеграцию API-интерфейса Profile с моделью аутентификации ASP.NET на основе форм, а также поддержку понятия “анонимных профилей”, которые позволяют хранить данные профиля для пользователей, не имеющих в данный момент активной идентичности на сайте.

На заметку! Темы, связанные с безопасностью ASP.NET (такие как аутентификация на основе форм и анонимные профили), в этой книге не рассматриваются. Подробные сведения ищите в документации .NET Framework 4.5 SDK.

Группирование данных профиля и сохранение специальных объектов

В завершение этой главы необходимо дать несколько дополнительных комментариев относительно того, как данные профиля могут быть определены в файле web.config. Текущий профиль просто определяет четыре порции данных, которые представлены непосредственно типом профиля. При построении более сложных профилей может оказаться полезным группирование вместе взаимосвязанных данных под одним уникальным именем. Рассмотрим следующее изменение:

```
<profile>
  <properties>
    <group name="Address">
      <add name="StreetAddress" type="String" />
      <add name="City" type="String" />
      <add name="State" type="String" />
    </group>
    <add name="TotalPost" type="Integer" />
  </properties>
</profile>
```

На этот раз определена специальная группа по имени Address, включающая название улицы, город и штат пользователя. Чтобы обратиться к этим данным на страницах, потребуется модифицировать код, указав Profile.Address для доступа к каждому подэлементу. Например, ниже приведен обновленный метод GetUserAddress() (обработчик события Click для кнопки Submit Data потребует аналогичных изменений):

```
private void GetUserAddress()
{
  // Здесь происходит чтение из базы данных.
  lblUserData.Text = String.Format("You live here: {0}, {1}, {2}",
    Profile.Address.StreetAddress,
    Profile.Address.City, Profile.Address.State);
}
```

Прежде чем запустить этот пример, потребуется удалить ASPNETDB.mdf из папки App_Data, чтобы обеспечить обновление схемы базы данных. После этого пример веб-сайта будет выполняться без ошибок.

На заметку! Профиль может содержать столько групп, сколько вы считаете необходимым. Просто определите несколько элементов <group> внутри контекста <properties>.

И, наконец, полезно упомянуть, что профиль может также хранить (и извлекать) в ASPNETDB.mdf специальные объекты. Чтобы проиллюстрировать это, предположим, что требуется построить специальный класс (или структуру), которая будет представлять данные адреса пользователя. Единственное требование, предъявляемое API-интерфейсом Profile к такому типу — он должен быть помечен атрибутом [Serializable], например:

```
[Serializable]
public class UserAddress
{
    public string Street = string.Empty;
    public string City = string.Empty;
    public string State = string.Empty;
}
```

Имея этот класс, определение профиля может быть изменено следующим образом (обратите внимание, что специальная группа удалена, хотя это и не обязательно):

```
<profile>
  <properties>
    <add name="AddressInfo" type="UserAddress" serializeAs="Binary"/>
    <add name="TotalPost" type="Integer" />
  </properties>
</profile>
```

При добавлении к профилю типов [Serializable] атрибут type представляет собой полностью заданное имя типа, который нужно хранить. Как вы увидите в окне IntelliSense среды Visual Studio, основными вариантами являются двоичные, XML и строковые данные. Получив информацию адреса в виде пользовательского класса, понадобится модифицировать базовый код:

```
private void GetUserAddress()
{
    // Здесь происходит чтение из базы данных.
    lblUserData.Text = String.Format("You live here: {0}, {1}, {2}",
        Profile.AddressInfo.Street, Profile.AddressInfo.City,
        Profile.AddressInfo.State);
}
```

Тема, связанная с API-интерфейсом Profile, намного шире того, что изложено здесь. Например, свойство Profile в действительности инкапсулирует тип по имени ProfileCommon. С помощью этого типа можно программно получать всю информацию о конкретном пользователе, удалять (или добавлять) профили в ASPNETDB.mdf, обновлять аспекты профиля и т.д.

Кроме того, API-интерфейс Profile имеет множество точек расширения, которые позволяют оптимизировать способ обращения диспетчера профилей к таблицам базы данных ASPNETDB.mdf. Как и можно было ожидать, существует немало способов сократить количество обращений к базе данных. Заинтересованные читатели могут обратиться за подробностями в документацию .NET Framework 4.5 SDK.

Резюме

В этой главе знания ASP.NET были дополнены сведениями об использовании типа `HttpApplication`. Как было показано, этот тип предлагает набор стандартных обработчиков событий, которые позволяют перехватывать различные события уровня приложения и уровня сеанса. Большая часть этой главы была посвящена рассмотрению различных приемов управления состоянием. Вспомните, что состояние представления применяется для автоматического заполнения значениями виджетов HTML между обратными отправками определенной страницы. Кроме того, вы узнали о различии между данными уровня приложения и данными уровня сеанса, об управлении cookie-наборами и о кеше приложения ASP.NET.

Наконец, вы ознакомились с интерфейсом ASP.NET Profile API. Как было показано, эта технология предлагает готовое решение задачи сохранения пользовательских данных между сеансами. Используя конфигурационный файл `web.config` веб-сайта, можно определять любое количество элементов профиля (включая группы элементов и типы `[Serializable]`), которые будут автоматически сохраняться в базе данных `ASPNETDB.mdf`.

Предметный указатель

A

ADO (Active Data Objects), 745
ADO.NET, 794;
 Entity Framework (EF), 852
 автономный уровень, 795
 поставщики данных ADO.NET, 747; 764
API-интерфейс ASP.NET, 1205
ASP.NET Profile, 1299

B

BAML (Binary Application Markup Language), 1018
BCL (base class library), 44

C

CLI (Common Language Infrastructure), 73
COM+, 901
COM (Component Object Model), 44
CIL (Common Intermediate Language), 44; 51
CLR (Common Language Runtime), 44; 46
CLS (Common Language Specification), 44
Common Type System (CTS), 44; 46; 902
Cookie-набор, 1294

D

DCOM (Distributed Component Object Model), 900
DLR (Dynamic Language Runtime), 567; 573
DOM (Document Object Model), 885

E

Entity Framework (EF), 747; 850; 854
Entity SQL, 873
Expression Design, 1125
Extensible Application Markup Language
 (XAML), 988

F

FIFO, 342
FTP (File Transfer Protocol), 1196

G

GAC (Global Assembly Cache), 482; 490; 509
GUID (Globally unique identifier), 313; 510

H

HTML (Hypertext Markup Language), 1197
HTTP (Hypertext Transfer Protocol), 1194

I

IDE (Integrated development environment), 78
IL (Intermediate Language), 51
IIS (Internet Information Services), 1196
IntelliSense, 378; 570

J

JIT (Just-In-Time), 44; 54
JVM (Java Virtual Machine), 62

L

LIFO, 342
LINQ (Language Integrated Query), 47; 426;
 435; 436; 448; 454
LINQ to Entities, 852; 872
LINQ to XML, 886

M

MEX (Metadata exchange), 924
MSIL (Microsoft Intermediate Language), 51
MSMQ (Microsoft Message Queuing), 902
MTS (Microsoft Transaction Server), 901

N

.NET Remoting, 902

P

Parallel LINQ (PLINQ), 692
PIA (Primary interop assembly), 580
Process identifier (PID), 589

R

RCW (Runtime Callable Wrapper), 579

S

SEH (Structured exception handling), 261
SharpDevelop, 86
Silverlight, 995
SOA (Service-oriented architecture), 905

T

TLS (Thread Local Storage), 590

U

UDT (User-defined type), 58

V

Visual Studio, 570; 1031; 1092; 1111; 1118
VSM (Visual State Manager), 1186

W

WAS (Windows Activation Service), 910
WCF (Windows Communication Foundation),
 588; 899; 904
WF (Windows Workflow Foundation), 953
WinRT (Windows Runtime), 76
WPF (Windows Presentation Foundation), 77;
 991; 1042; ; 1136
WSDL (Web Service Description Language), 927

А

Адаптер данных, 746; 754; 796; 819
 Адрес, 911
 WCF, 914
 Алгоритм
 FIFO, 342
 LIFO, 342
 Аргумент
 именованный, 152
 Архитектура
 ориентированная на службы (SOA), 905
 Атрибут
 CIL, 614
 .NET, 614
 контекстный, 609
 рефлексия атрибутов с использованием
 раннего связывания, 557
 специальный, 553
 уровня сборки, 555

Б

База данных
 ASPNETDB.mdf, 1299
 Безопасность
 типов, 330; 331
 Библиотека
 .NET mscorlib.dll, 482
 базовых классов, 46
 кода (*.dll), 489
 Бизнес-процесс, 953

В

Веб-приложение, 1195; 1220
 Веб-сайт, 1220
 Веб-сервер, 1196
 Веб-службы XML, 903
 Веб-элементы управления, 1235
 Виртуальная машина Java (JVM), 62
 Выражение
 дерево выражений, 574
 лямбда-, 357; 384

Г

Глобально уникальный идентификатор
 (GUID), 510
 Глобальный кеш сборок (GAC), 69; 490; 509
 Графика
 непосредственного режима, 1098
 режима сохранения (retained mode), 1098

Д

Данные
 обмен метаданными (MEX), 924
 очередизация, 902

параллелизм, 684
 привязка, 1091
 Делегат, 59; 357; 358
 анонимный, 687
 включение группового вызова, 367
 обобщенный, 370; 381
 типа делегата .NET, 357
 типа делегата в C#, 358
 Диспетчер задач Windows, 589
 Домен приложения, 588; 599; 606
 стандартный, 600

Ж

Жизненный цикл веб-страницы, 1230

З

Запрос PLINQ, 692

И

Имя
 строгое
 генерация в Visual Studio, 513
 Индексатор, 396
 Инкапсуляция, 182; 201; 227
 Интерфейс, 57; 287; 291
 API-интерфейс ASP.NET Profile, 1299
 именованный, 310
 интерфейсный тип, 287; 298
 использование в качестве возвращаемых
 значений, 298
 использование в качестве параметров, 296
 полиморфный, 246
 реализация интерфейса, 292
 явная, 301
 специальный, 290
 Исключение
 внутреннее, 282
 времени выполнения, 252
 обработка исключений, 278
 структурированная, 261
 перехват исключений, 268
 построение специальных исключений, 274
 системное, 273
 уровня приложения, 274
 Итератор, 309

К

Квант времени, 590
 Кеш приложения, 1286
 Класс
 Animation, 1150
 AppDomain, 600
 Application, 998
 ApplicationCommands, 1069
 Array, 161

- AsyncResult, 662
- AutoResetEvent, 671
- BasicHttpBinding, 913
- BinaryFormatter, 730; 732
- BinaryReader, 703; 722
- BinaryWriter, 703; 722
- Brush, 1110
- BufferedStream, 703
- Collections, 324
- CombinedGeometry, 1108
- CompareValidator, 1264
- ComponentCommands, 1069
- Console, 111
- ContentControl, 999
- ContentPresenter, 1190
- Control, 1001; 1238
- DataColumn, 799; 800
- DataRow, 802; 837
- DataSet, 797; 835
- DataTable, 805; 811; 836
- DbCommand, 747; 764; 773
- DbConnection, 747; 764
- DbDataAdapter, 747; 764; 819
- DbDataReader, 747; 764
- DbParameter, 748; 764; 782
- DbTransaction, 748; 764
- DependencyObject, 1003
- Dictionary<TKey, TValue>, 339
- Directory, 703; 708
- DispatcherObject, 1003
- Drawing, 1121
- DrawingBrush, 1110; 1111
- DrawingGroup, 1122
- DrawingVisual, 1130
- EditingCommands, 1069
- EllipseGeometry, 1108
- Environment, 110
- Exception, 264
- File, 703; 714
- FileInfo, 703; 710
- FileStream, 703; 716
- FrameworkElement, 1002
- Geometry, 1107
- GeometryDrawing, 1122
- GeometryGroup, 1108
- GlyphRunDrawing, 1122
- HttpApplication, 1282
- HttpApplicationState, 1283; 1286
- HttpRequest, 1225
- HttpSessionState, 1293
- ImageBrush, 1111
- ImageDrawing, 1122
- LinearGradientBrush, 1110
- LineGeometry, 1108
- LinkedList<T>, 339
- List<T>, 339
- MainWindow, 1029
- MatrixTransform, 1114
- MediaCommands, 1069
- MsmqIntegrationBinding, 915
- NavigationCommands, 1069
- NetMsmqBinding, 915
- NetNamedPipeBinding, 914
- NetPeerTcpBinding, 914
- NetTcpBinding, 914
- Object, 253
- ObjectContext, 857
- OmageBrush, 1110
- Parallel, 683
- Path, 703
- PathGeometry, 1108
- Polygon, 1106
- Polyline, 1106
- Process, 591
- ProcessThread, 596
- Queue<T>, 339; 342
- RadialGradientBrush, 1110
- RangeValidator, 1264
- RectangleGeometry, 1108
- RegularExpressionValidator, 1264
- RequiredFieldValidator, 1263
- RotateTransform, 1114
- ScaleTransform, 1114
- ServiceHost, 922
- Shape, 1101
- SkewTransform, 1114
- SolidColorBrush, 1110
- SortedDictionary<TKey, TValue>, 339
- SortedList<T>, 339; 344
- Stack<T>, 339; 342
- Stream, 715
- String, 124
- StringBuilder, 129
- System.Activator, 546
- System.GC, 463
- System.Type, 534
- Task, 688
- Thread, 664
- Timeline, 1152
- Transform, 1114
- TransformGroup, 1114
- TranslateTransform, 1114
- UIElement, 1002
- VideoDrawing, 1122
- Visual, 1003; 1129
- VisualBrush, 1110; 1111

WebControl, 1238; 1243
 Window, 999
 WorkflowApplication, 960
 WorkflowInvoker, 957
 WSDualHttpBinding, 913
 WSFederationHttpBinding, 913
 WSHttpBinding, 913
Ключ
 открытый, 510
 секретный, 510
Ключевые слова C#, связанные с указателями, 418
Код
 отделенный, 1216
Коллекция, 323; 333; 339
Команды WPF, 1068
Компилятор
 csc.exe, 80
 JIT, 54
Компиляция
 оперативная (JIT), 44
Конкатенация строк, 125
Конструктор, 185
 стандартный, 185
 статический, 198
Контейнер
 безопасный в отношении типов, 331
Контракт службы, 911
Куча
 управляемая, 455
Л
Лямбда-выражение, 357; 384; 428
М
Манифест, 52
 сборки, 56
Массив, 156; 298
 зубчатый (ступенчатый), 159
 использование массивов в качестве аргументов, 160
 многомерный, 159
 параметров, 150
 прямоугольный, 159
 синтаксис инициализации массивов C#, 157
Мастер-страница, 1235; 1246
Метаданные, 52
 обмен метаданными (MEX), 924
Метод
 виртуальный, 241
 групповое преобразование методов, 369
 синтаксис, 378
 -индексатор, 391
 многомерный, 395
 перегрузка методов, 154; 156; 348

 расширяющий, 429; 436
 создание, 347
Модификатор
 out, 148
 params, 150
 ref, 149
 доступа, 205; 206
 параметров в C#, 147
Модуль, 596

Н

Наследование, 201; 202; 227
 классическое, 228

О

Обобщения, 322
Обработка исключений
 структурированная, 261
Обратный вызов, 357
Объект
 контекст объекта, 853
 представления, 818
 приложения, 105
 команд, 773
 построителей строк соединения, 772
 сериализация объектов, 702
 службы объектов, 854
 создание
 ленивое (отложенное); 476
 фабрика объектов подключений, 757
 чтения данных, 774
Очереди данных, 902
Очередь, 342
Ошибка
 на этапе компиляции, 117
 пользовательская, 261
 программная, 261

П

Параллелизм данных, 684
Параметр
 необязательный, 151
 передача параметров по значению, 147
Перегрузка операций, 396
Переменная
 внешняя, 383
 локальная, 435
 неявно типизированная, 435
Перечисление, 58; 162
Поле
 автоинкрементное, 801
Полиморфизм, 201; 203
 классический, 204
Поставщики данных ADO.NET, 747; 749

Поток, 589
 главный, 589
 переднего плана, 672
 рабочий, 590
 фоновый, 672
 Привязка, 911
 данных, 1091
 на основе HTTP, 912
 на основе MSMQ, 914
 на основе TCP, 914

Приложение
 расширяемое, 560
 браузерное, 994

Процесс, 588

Р

Распаковка (unboxing), 328
 Редактор
 Кахамл, 1020; 1021
 конфигурации проекта, 93
 трансформаций Visual Studio, 1118

Ресурс
 двоичный, 1136

Ресурсы
 логические, 1136
 объединенный словарь ресурсов, 1147
 объектные, 1136; 1142
 уровня приложения, 1146

Рефакторинг, 94

Рефлексия, 533
 атрибутов
 с использованием позднего связывания, 558
 с использованием раннего связывания, 557
 разделяемых сборок, 543

С

Сборка, 52; 65; 490
 Microsoft.CSharp.dll, 570
 PIA, 580
 WCF, 907
 атрибуты уровня сборки, 555
 взаимодействия, 578; 579
 основная (primary interop assembly, PIA), 580
 динамически загружаемая, 541; 604; 641
 закрытая, 504
 манифест сборки, 493
 метаданные сборки, 493
 подключение внешних сборок, 566
 подчиненные сборки, 494
 разделяемая, 515
 рефлексия разделяемых сборок, 543
 связанная с LINQ, 432
 ссылка на внешние сборки, 69
 статическая, 641

Сборка мусора, 455; 466
 параллельная, 462
 фоновая, 462
 Свойство, 1176
 зависимости, 1003; 1165
 построение специального свойства
 зависимости, 1170
 Связывание
 позднее, 545; 576
 Сериализация объектов, 702; 725; 737
 Службы
 Internet Information Services (IIS), 1196
 WAS, 910
 WCF, 916
 хостинг, 919
 Window, 945
 контракты служб, 911
 объектов, 854
 тестирование, 951
 типы служб, 911
 установка, 944
 Смарт-тег, 300
 Событие
 маршрутизируемое, 1176
 прямое, 1176
 пузырьковое, 1176
 туннельное, 1176
 Состояние
 представления, 1277
 проблема поддержки состояния, 1274
 Спецификация
 общезыко́вая (CLS), 44; 46; 61
 Среда DLR, 567
 Стек
 виртуальный, 616
 Страница, 994
 Строка
 базовые манипуляции строками, 125
 конкатенация строк, 125
 определение дословных строк, 127
 управляющие последовательности в
 строковых литералах, 126
 Структура, 168
 Сущности (entity), 850; 852
 клиент сущности, 854; 855

Т

Таблицы стилей, 1268
 Технология
 ADO.NET, 794
 COM, 44
 COM+, 901
 LINQ, 47; 426; 436; 454
 операции запросов LINQ, 443
 сборки, связанные с LINQ, 432

Surround With, 96

WPF, 77

Тип

анонимный, 429

вложенный, 239

динамический, 567

интерфейсный, 287

параметры типа, 335

для обобщенных интерфейсов, 336

для обобщенных членов, 336

проблемы с безопасностью типов, 330

Тип данных

C#, 116

допускающий null, 177

преобразования типов данных, 130

структура (struct), 168

Типы служб, 911

Транзакция, 790

Трансформации, 1114; 1120

редактор трансформаций Visual Studio, 1118

У

Указатели, 422

Упаковка (boxing), 327

Управляющие последовательности в строковых литералах, 126

Уровень

автономный, 796

Утилита

Class Designer, 98

Class View, 93

EdmGen.exe, 850; 860

gacutil.exe, 515

ilasm.exe, 622

ildasm.exe, 70; 359; 457

installutil.exe, 944

msbuild.exe, 1014; 1018; 1031

Object Browser, 93

peverify.exe, 623

Solution Explorer, 91

SvcConfigEditor.exe, 939; 940

svcutil.exe, 927; 928

WcfTestClient.exe, 938; 951

Ф

Фабрика

объектов подключений, 757

поставщиков данных ADO.NET, 764

Файл

App.config, 507; 908

AssemblyInfo.cs, 556

*.aspx, 1252; 1289

CILTypes.il, 628

*.config, 939

*.dll, 489

*.edmx, 856

Global.asax, 1279

Inventory.xml, 895

*.rsp, 84

*.skin, 1269

*.svc, 951

*.targets, 1016

web.config, 1233

Финализация, 469

Форма HTML, 1199

Формат BAML, 1018

Функция

обратного вызова, 357

Х

Хостинг службы WCF, 919

Хранимая процедура, 783

Ц

Цикл

do/while, 141

for, 140

foreach, 140

while, 141

Э

Экспорт данных в файл Excel, 585

Элементы управления

Ink API, 1044

WPF, 1043; 1045

веб-, 1235

проверкой достоверности ASP.NET, 1261

серверные, 1235

Я

Язык

BAML, 1018

CIL, 44; 52; 612

атрибуты, 614

директивы, 614

коды операций, 614; 632

Entity SQL, 873

F#, 49

HTML, 1197

IL, 51

IronPython, 573

IronRuby, 573

MSIL, 51

XAML, 988

интегрированных запросов (LINQ), 47