Курс "Об'єктно-орієнтоване програмування"

OOP

Лекцій – 48 годин (24+24)

Лабораторних робіт – 48 годин (24+24)

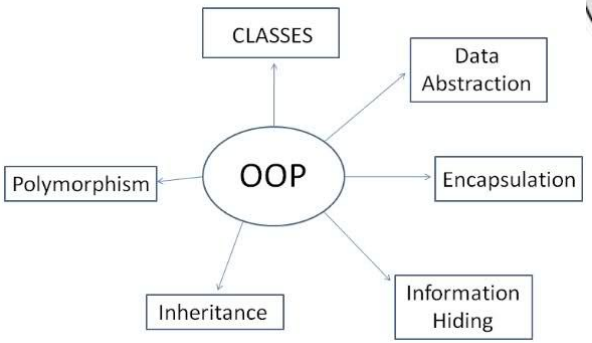
Самостійна робота – 204 годин

Кредитів - 10

Екзамен – 3 семестр (осінь)

Залік – 2 семестр (весна)

Курсова робота – 3 семестр



Мета – вивчення й практичне освоєння **методів і засобів об'єктно-орієнтованого програмування**, знайомство з **методологією об'єктно-орієнтованого аналізу та проектування (OOA & OOD)** складних програмних систем; вивчення сучасної об'єктно-орієнтованої **мови програмування C#.**

Лектор – доцент кафедри ПЗАС Попівщий Василь Іванович, 2014 – 2017 р.

Головні теми

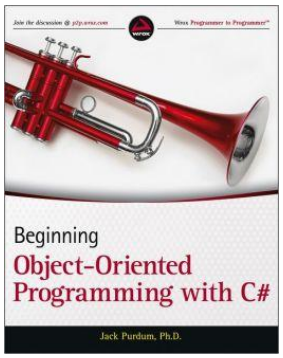
- Об’єктно-орієнтована парадигма програмування
- Побудова обєктної моделі предметної області
- Філософія .NET Framework
- Основні типи мови C#
- Класи. Спадкування. Структури
- Обробка виняткових ситуацій
- Інтерфейси
- Перевантаження операцій
- Узагальнення. Атрибути
- Делегати. Події
- Лямбда-вирази. Анонімні методи
- Мова запитів LINQ
- Динамічне зв’язування
- Паралелізм. Асинхронні функції

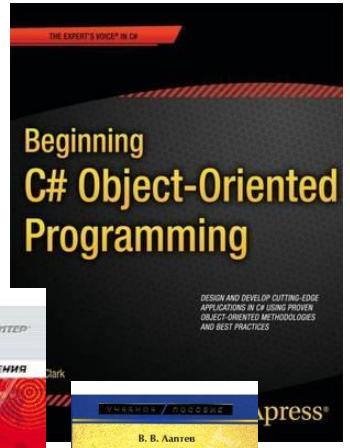
Лабораторні роботи (поточний семестр)

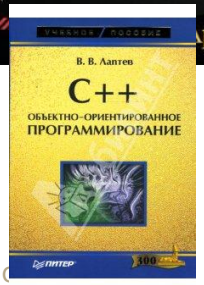
1. Мова C#. Робота з вбудованими типами даних. Застосунки консольний та Windows Forms. **Лаб. №1**
2. Цикли, рядки, файли в C#. **Лаб№2**
3. Обробка виключень в C#. **Лаб№3**
4. Використання масивів і колекцій в C#. **Лаб№4**
5. Класи в C#. Спадкування. **Лаб№5**
6. Перевантаження операцій в C#. **Лаб№6**
7. Делегати і події C#. **Лаб№7**
8. Використання елементів управління Windows Forms і діалогових вікон. **Лаб№8**

Література

1. **Пышкин Е.В. Основные концепции и механизмы объектно-ориентированного программирования** . – СПб.: BHV, 2005.
2. **Лафоре Р. Объектно-ориентированное программирование в C++** 4-е издание – СПб.: Питер, 2004
3. **Beginning Object-Oriented Programming with C#**. – Wrox, 2012
4. **Beginning C# Object-Oriented Programming.** – Apress, 2013
5. **Васильев А.Н. Java. Объектно-ориентированное программирование, 2011**
6. **Лаптев В.В. C++. Объектно-ориентированное программирование.**







Література

7. **Шилдт Герберт C# 4.0 Полное руководство**, 2011

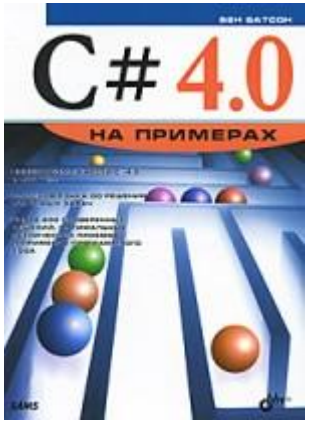
8. **Эндрю Троелсен - ЯЗЫК ПРОГРАММИРОВАНИЯ C#5.0 И ПЛАТФОРМА .NET 4.5**, 2013

9. Зиборов В.В. - Visual C# 2012 на примерах.

10. Ватсон Б. - C# 4.0 на примерах.

11. **Павловская Т.А. C#. Програм-е на языке высокого уровня**

12. **Подбельский В.В. Язык C#**

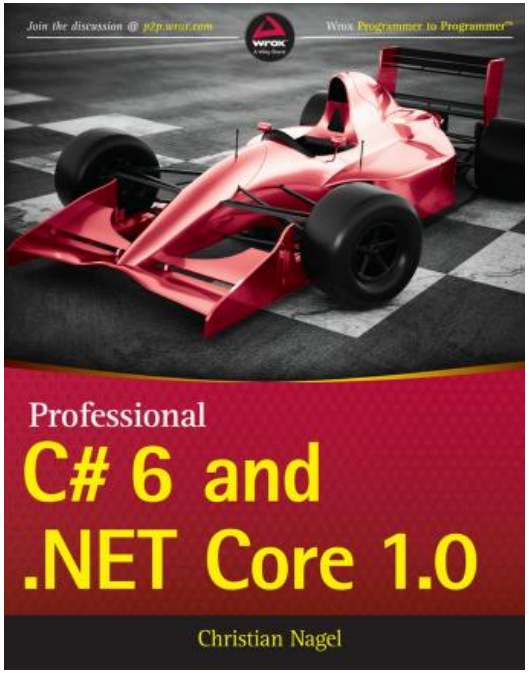


Література

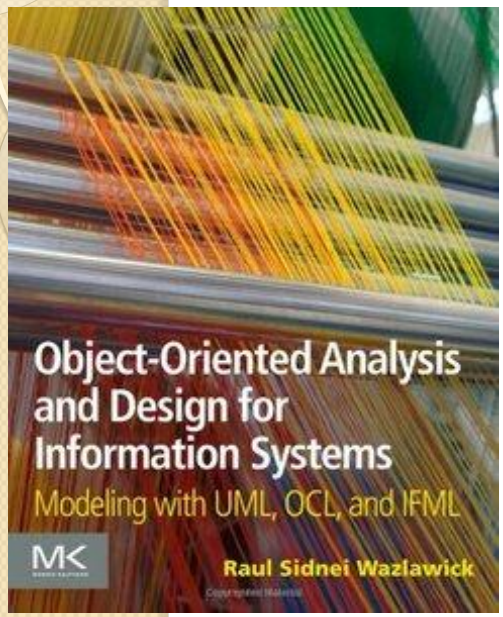
13. **Трей Нэш C# 2010. Ускоренный курс для профессионалов.**

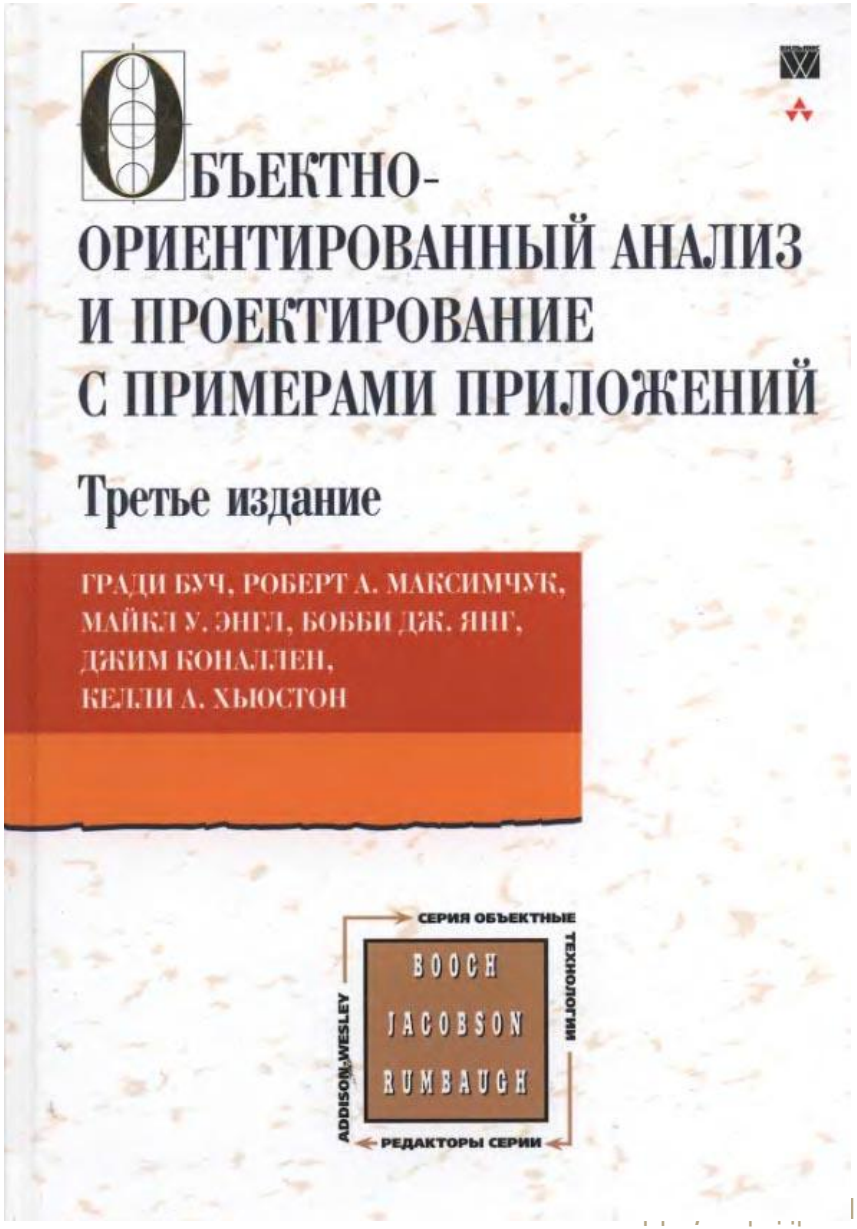
14. К. Нейгел C# 2008 и платформа .NET 3.5 для профессионалов.

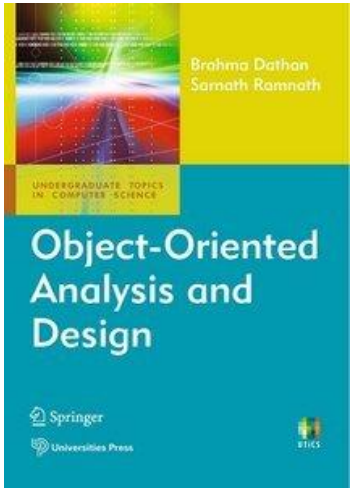
15. **Кравець П.О. Об'єктно-орієнтоване програмування**, Видавництво Лівівської політехніки, 2012

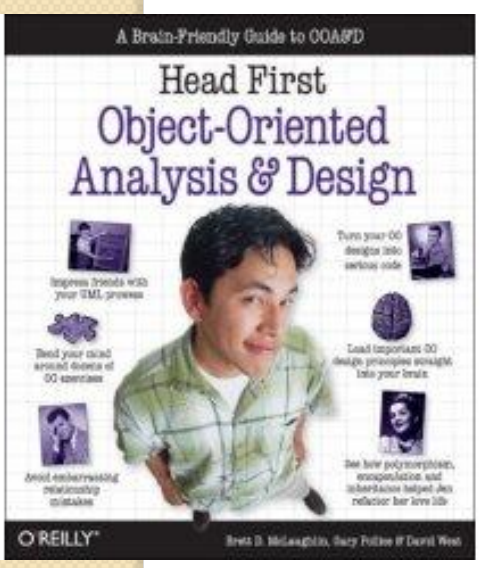


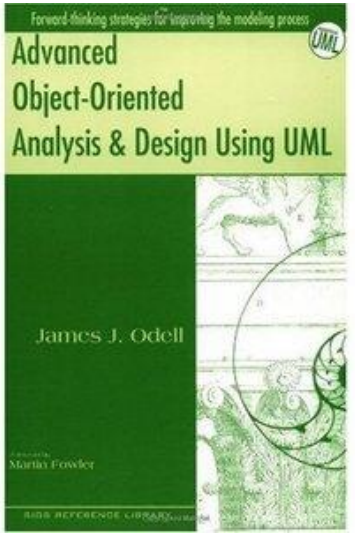
Література


Object-Oriented Analysis and Design for Information Systems
Modeling with UML, OCL, and IFML
Raul Sidnei Wazlawick


ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ И ПРОЕКТИРОВАНИЕ С ПРИМЕРАМИ ПРИЛОЖЕНИЙ
Третье издание
ГРАДИ БУЧ, РОБЕРТ А. МАКСИМЧУК, МАЙКЛ У. ЭНГЛ, БОББИ ДЖ. ЯНГ, ДЖИМ КОНАЛЛЕН, КЕЛЛИ А. ХЬЮСТОН
СЕРИЯ ОБЪЕКТНЫЕ ТЕХНОЛОГИИ
ADDISON-WESLEY
BOOCH
JACOBSON
RUMBAUGH
РЕДАКТОРЫ СЕРИИ


Brahma Dathan
Sarnath Ramnath
Object-Oriented Analysis and Design
Springer
Universities Press


A Brain-Friendly Guide to OOA&D
Head First
Object-Oriented Analysis & Design
O'REILLY


Advanced
Object-Oriented
Analysis & Design Using UML
James J. Odell
Martin Fowler

Медаль имени Ады Лавлейс присуждена Гради Бучу

Британское компьютерное общество BCS присудило ежегодно вручаемую медаль имени Лавлейс главному научному сотруднику IBM Research Гради Бучу за достижения в области архитектуры программ, технологий программирования и совместной работы.

Гради Буч широко известен как создатель объектно-ориентированного подхода к проектированию и разработке программного обеспечения, а также методов совместной разработки. Влияние, которое работы Буча оказали на развитие индустрии ИТ и его вклад в научные исследования, в том числе публикация шести завоевавших большую популярность книг, делают его достойным кандидатом на получение медали имени Лавлейс, отмечается в заявлении главы BCS Дэвида Кларка.

Медаль имени Ады Лавлейс была учреждена BCS в 1998 году и ежегодно присуждается людям, внесшим значительный вклад в развитие информационных систем. Гради Буч стал сотрудником IBM в 2003 году, после приобретения ею компании Rational Software, где он также был главным научным сотрудником. Сейчас Буч занимается главным образом теорией и практикой проектирования программ для сверхбольших систем.

Відеокурси

- **Специалист М2555 Разработка Windows - приложений для Microsoft .NET на Visual C# [2011]**
- **Специалист М10266 (М2124) Программирование на C# с использованием Microsoft .NET Framework 4 [2011]**
- Udemy - Object Oriented Programming (2014)
- AppDev - Object Oriented Programming
- AppDev - Learning To Program Using Visual C# 2008
- Lynda.com - Foundations of Programming Object-Oriented Design
- TekPub - C# Design Strategies with Jon Skeet
- Tekpub - Mastering C#4 with Jon Skeet
- thenewboston.org - C# Beginners Tutorial
- **Асинхронное программирование**

“ C# - один из наиболее популярных объектно-ориентированных языков программирования в мире. Став специалистом, пишущим на C#, Вы будете востребованы в любой стране. ”

Рейтинг мов програмування (www.tiobe.com)

Mar 2014	Mar 2013	Change	Programming Language	Ratings	Change
1	2	↑	C	17.535%	+0.39%
2	1	↓	Java	16.406%	-1.75%
3	3		Objective-C	12.143%	+1.91%
4	4		C++	6.313%	-2.80%
5	5		C#	5.572%	-1.02%
6	6		PHP	3.698%	-1.11%
7	7		(Visual) Basic	2.955%	-1.65%
8	8		Python	2.021%	-2.37%
9	11	↑	JavaScript	1.899%	+0.53%
10	16	⇈	Visual Basic .NET	1.862%	+0.97%
11	17	⇈	Transact-SQL	1.477%	+0.64%
12	69	⇈	F#	1.216%	+1.14%
13	10	↓	Perl	1.149%	-0.81%
14	9	⇊	Ruby	0.974%	-1.18%
15	15		Delphi/Object Pascal	0.881%	-0.01%

Feb 2015	Feb 2014	Change	Programming Language	Ratings	Change
1	1		C	16.488%	-1.85%
2	2		Java	15.345%	-1.97%
3	4	↑	C++	6.612%	-0.28%
4	3	↓	Objective-C	6.024%	-5.32%
5	5		C#	5.738%	-0.71%
6	9	↑	JavaScript	3.514%	+1.58%
7	6	↓	PHP	3.170%	-1.05%
8	8		Python	2.882%	+0.72%
9	10	↑	Visual Basic .NET	2.026%	+0.23%
10	-	⇈	Visual Basic	1.718%	+1.72%
11	20	⇈	Delphi/Object Pascal	1.574%	+1.05%
12	13	↑	Perl	1.390%	+0.50%
13	15	↑	PL/SQL	1.263%	+0.66%
14	16	↑	F#	1.179%	+0.59%
15	11	⇊	Transact-SQL	1.124%	-0.54%

Jan 2017	Jan 2016	Change	Programming Language	Ratings
1	1		Java	17.278%
2	2		C	9.349%
3	3		C++	6.301%
4	4		C#	4.039%
5	5		Python	3.465%
6	7	^	Visual Basic .NET	2.960%
7	8	^	JavaScript	2.850%
8	11	^	Perl	2.750%
9	9		Assembly language	2.701%
10	6	⌄⌄	PHP	2.564%
11	12	^	Delphi/Object Pascal	2.561%
12	10	⌄	Ruby	2.546%
13	54	^^	Go	2.325%
14	14		Swift	1.932%

Основні принципи ООП

- **Визначення ООП. ООП – це метод побудови програм у вигляді множини взаємодіючих об'єктів, структура і поведінка яких описані відповідними класами, і всі ці класи є компонентами ієрархії класів, яка відображає відношення спадкування.**
- Першою мовою, що підтримувала технологію ООП була мова Simula 67 (Норвезький Обчислювальний Центр, Осло).
- Наступна мова – Smalltalk (1972)
- Потім: Ada (1980), Eiffel (1986), C++ (1983), Java (1995), Pascal(Delphi, 1995), C# (2002), …
- **ООП – це одна з парадигм програмування.**

Парадигми програмування

- **Парадигма програмування – це спосіб мислення розробника програми, це спосіб концептуалізації, який визначає, як складати програми**.

Відомі парадигми програмування:

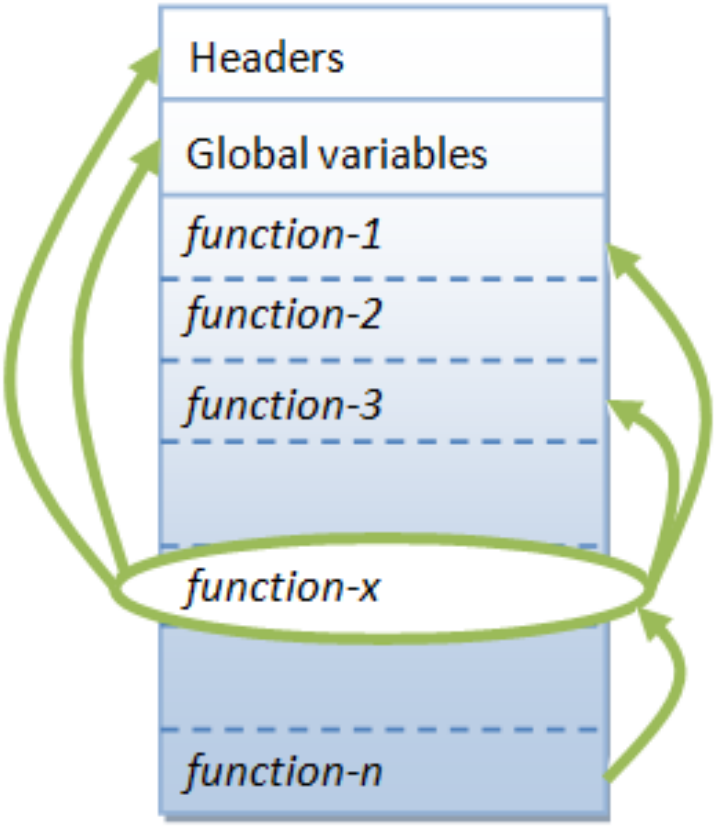
1. **Директивне (процедурне) програмування** (C, Fortran, Cobol, Pascal, ...).
2. **Об'єктно-орієнтоване програмування** (ООП).
3. **Функціональне та логічне програмування** (Lisp, Haskell, Prolog, F#).
4. **Аспектно-орієнтоване програмування**.
5. **Компонентне програмування**.

Директивне (процедурне) програмування

Традиційні процедурно-орієнтовані мови (наприклад , C і Паскаль) мають **помітні недоліки** у створенні багаторазово використовуваних (reuseble) програмних компонентів:

1. **Програми складаються з функцій. Функції часто не можуть використовуватися повторно** (функція може посилатися на заголовки , глобальні змінні та інші функції).
   - Іншими словами , функції не добре інкапсулюються як самодостатній багаторазового використовуваний блок.
2. **Процедурні мови не мають абстракцій високого рівня для вирішення реальних життєвих проблем**. Наприклад , програми C використовує конструкції низького рівня (if-else, for-loop, array, method, pointer) і важко побудувати абстракції реальних проблем , такі як Системи Управління Взаємовідносн з Клієнтами (CRM) або комп'ютерну гру футбол.

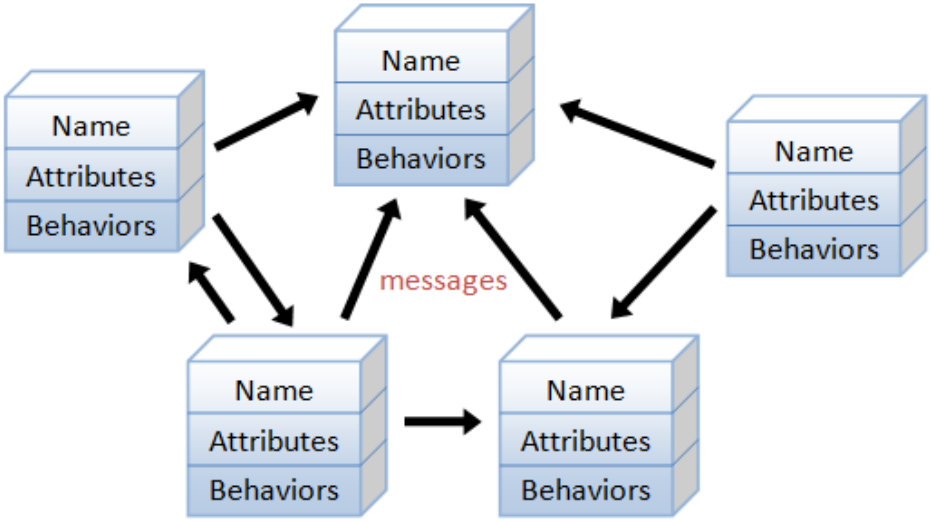
Traditional Procedural-Oriented languages



A function (in C) is not well-encapsulated

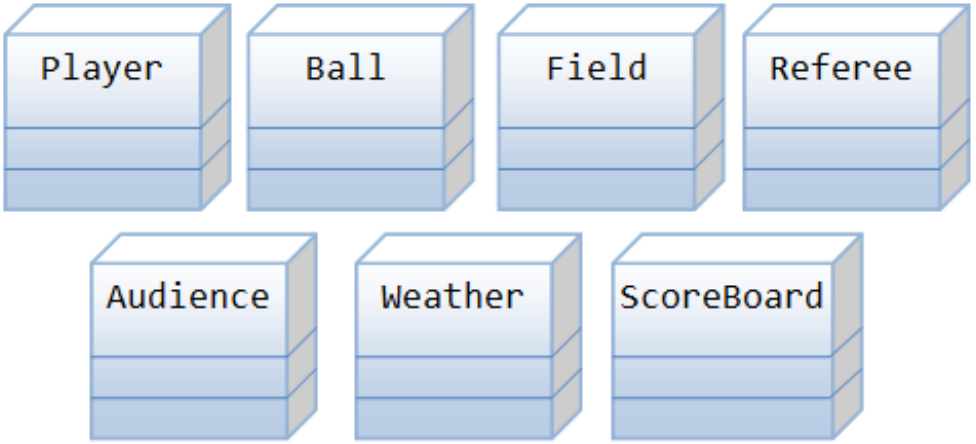
Object-Oriented Programming Languages

- **Основною одиницею ООП є клас, який інкапсулює як статичні атрибути так і динамічну поведінку в один об’єкт** (“коробку”), і визначає загальний інтерфейс для використання цих “коробок”.
- Висновок: клас краще інкапсульований (порівняно з функцією), тому його **легше повторно використовувати**.
- **Іншими словами, ООП об'єднує всередині одного об’єкту структури даних і алгоритми обробки цих даних.**



An object-oriented program consists of many well-encapsulated objects and interacting with each other by sending messages

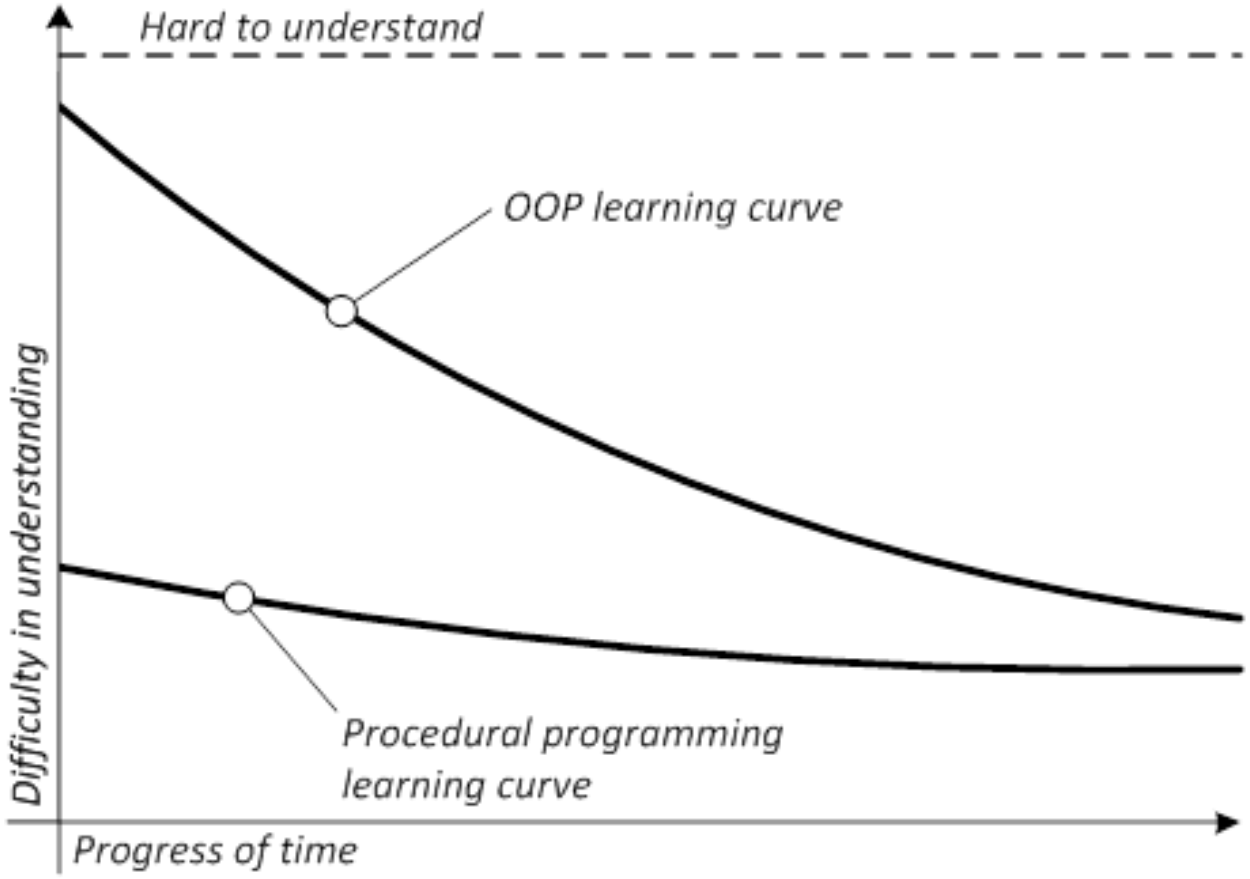
Object-Oriented Programming Languages



Classes (Entities) in a Computer Soccer Game

- **Player**: ***attributes*** include name, number, location in the field, and etc; ***operations*** include run, jump, kick-the-ball, and etc.
- ...

Hard to understand



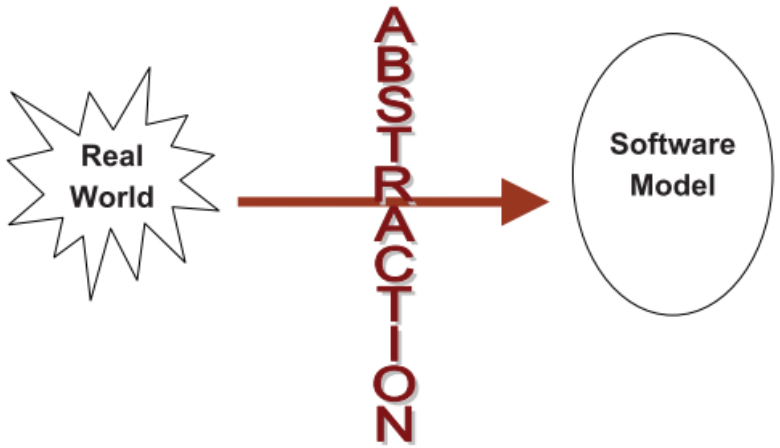
Об'єктна модель

- Кожен стиль програмування має свою **концептуальну базу.**
- Для ОО-стиля концептуальна база – це **об'єктна модель**
- Об'єктна модель складається з 5-ти головних елементів:
  - **Абстрагування (abstraction)**
  - **Інкапсуляція (encapsulation)**
  - **Поліморфізм**
  - **Модульність**
  - **Ієрархія (hierarhy)**
- **Абстрагування дозволяє виділяти у вигляді класу суттєві характеристики деякого обєкту, важливі для даної прикладної задачі**.

Об’єктна модель. Абстрагування

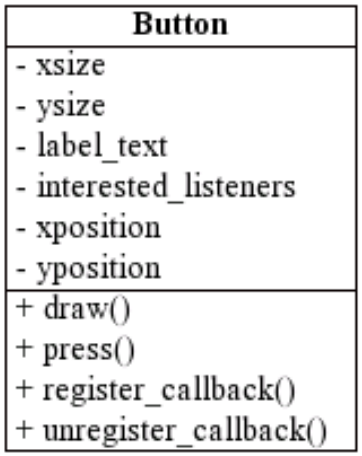
Інкапсуляція має на увазі створення межі навколо об'єкта, що розділяє його зовнішню поведінку від внутрішньої реалізації.

Інкапсуляція означає, по-перше, що атрибути і методи об’єкта асоційовані саме з цим об’єктом, а по-друге, що область видимості атрибутів і методів за замовчуванням обмежена самим об’єктом.



Об’єктна модель

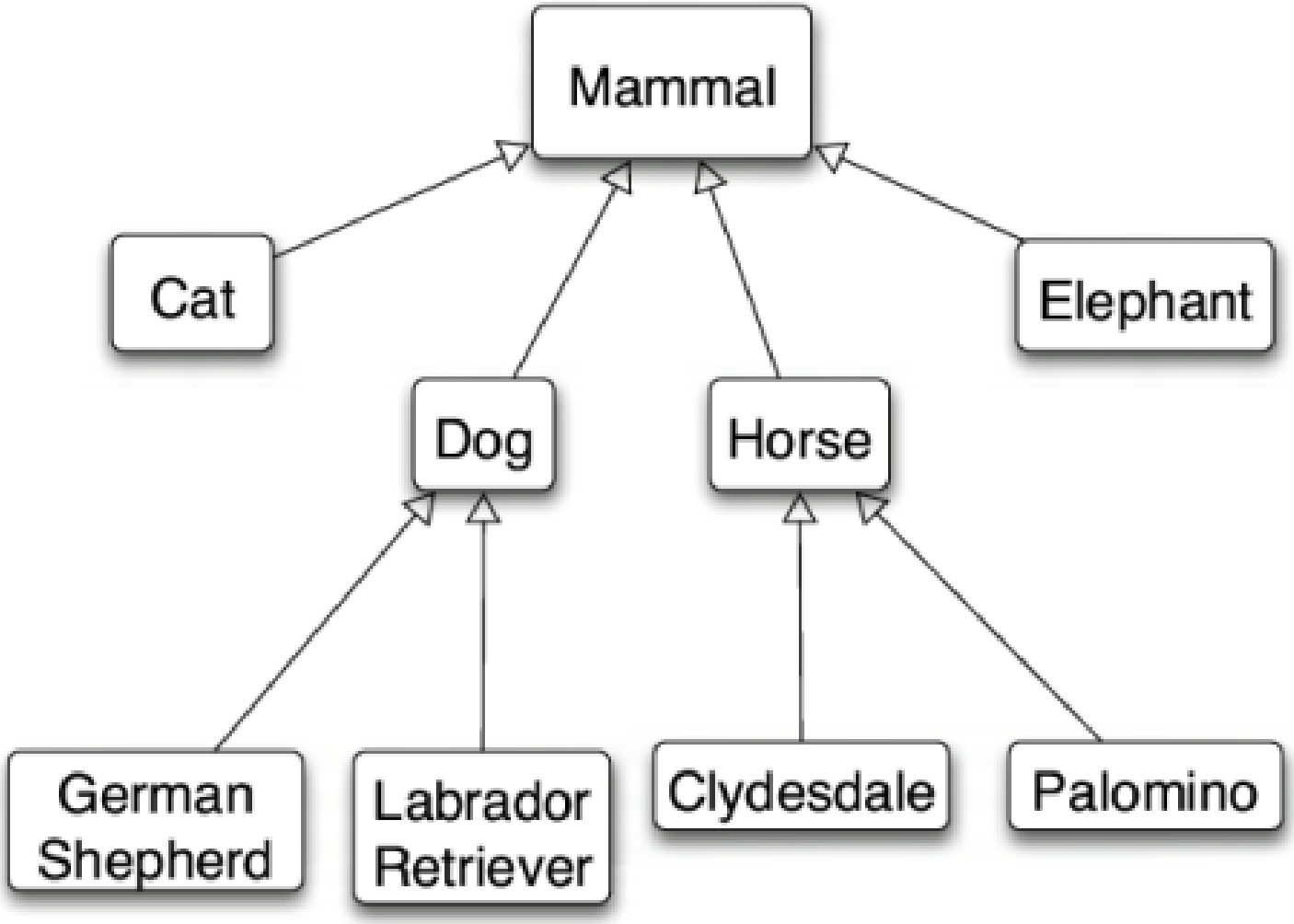
- **Абстрагування**



До речі, це частина діаграми класів мови UML (Unified Modeling Language)

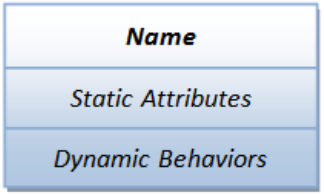
- **Інкапсуляція – це процес відділення один від одного елементів об’єкту, що визначають його внутрішній устрій і зовнішню поведінку.**
- **Ієрархія – це упорядкування абстракцій, засіб класифікації об’єктів, систематизації зв’язків між об’єктами.**
- **Модульність – це представлення системи у вигляді сукупності відокремлених сегментів, зв’язок між якими забезпечується через зв’язок між класами, що визначені в цих сегментах.**

Example of inheritance hierarchy



Об'єктна модель

- **Клас – це користувальницький тип даних (user defined type), що складається з даних (атрибутів, властивостей, полів) та операцій, що виконують з цими даними певні дії** (це методи класу, функції-члени класу).
- **Об'єкт – це екземпляр класу (instance of a class).**
- **Об'єкт – це сутність, що є контейнером для даних і управляє доступом до цих даних.**
- **Клас можна уявляти собі як креслення або шаблон, а об'єкт – як річ, що виготовлена за цим кресленням.**
- **Успадкування – це механізм, що дозволяє розширити раніше визначену сутність шляхом додавання нових можливостей.**



Класи і об'єкти

Name (Identifier)
Variables (Static attributes)
Methods (Dynamic behaviors)

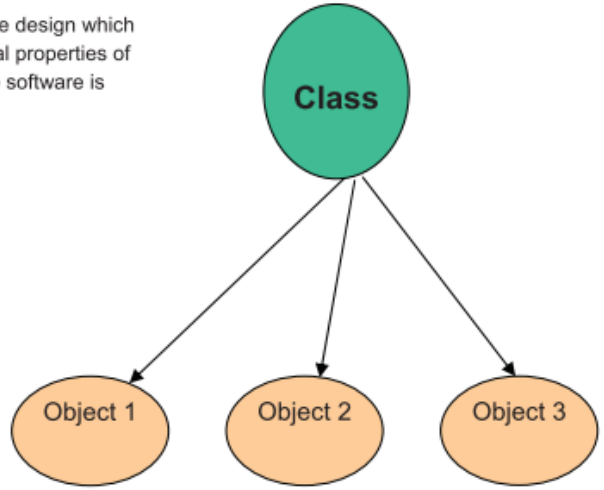
Student	Circle
name grade	radius color
getName() printGrade()	getRadius() getArea()

SoccerPlayer	Car
name number xLocation yLocation	plateNumber xLocation yLocation speed
run() jump() kickBall()	move() park() accelerate()

Examples of classes

A 'class' is a software design which describes the general properties of something which the software is modelling.

Individual 'objects' are created from the class design for each actual thing



Name
Variables
Methods

paul:Student	peter:Student
name="Paul Lee" grade=3.5	name="Peter Tan" grade=3.9
getName() printGrade()	getName() printGrade()

Two instances of the class Student

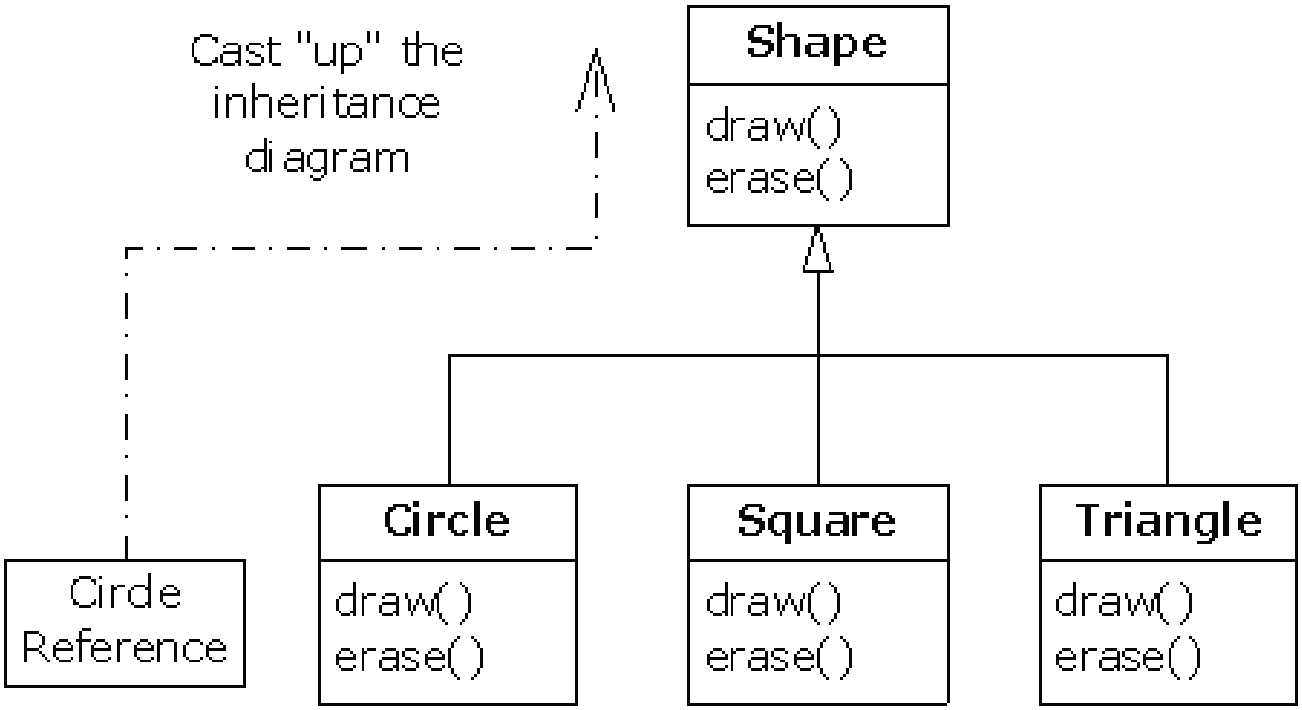
Категорії класів

- **Класи аналізу**. Вони описують абстракцію даних, що безпосередньо пов’язана з моделлю предметної області
- **Класи реалізації**. Вони виходять з внутрішніх потреб алгоритмів системи.
- **Класи проектування**. Вони описують архітектурний вибір

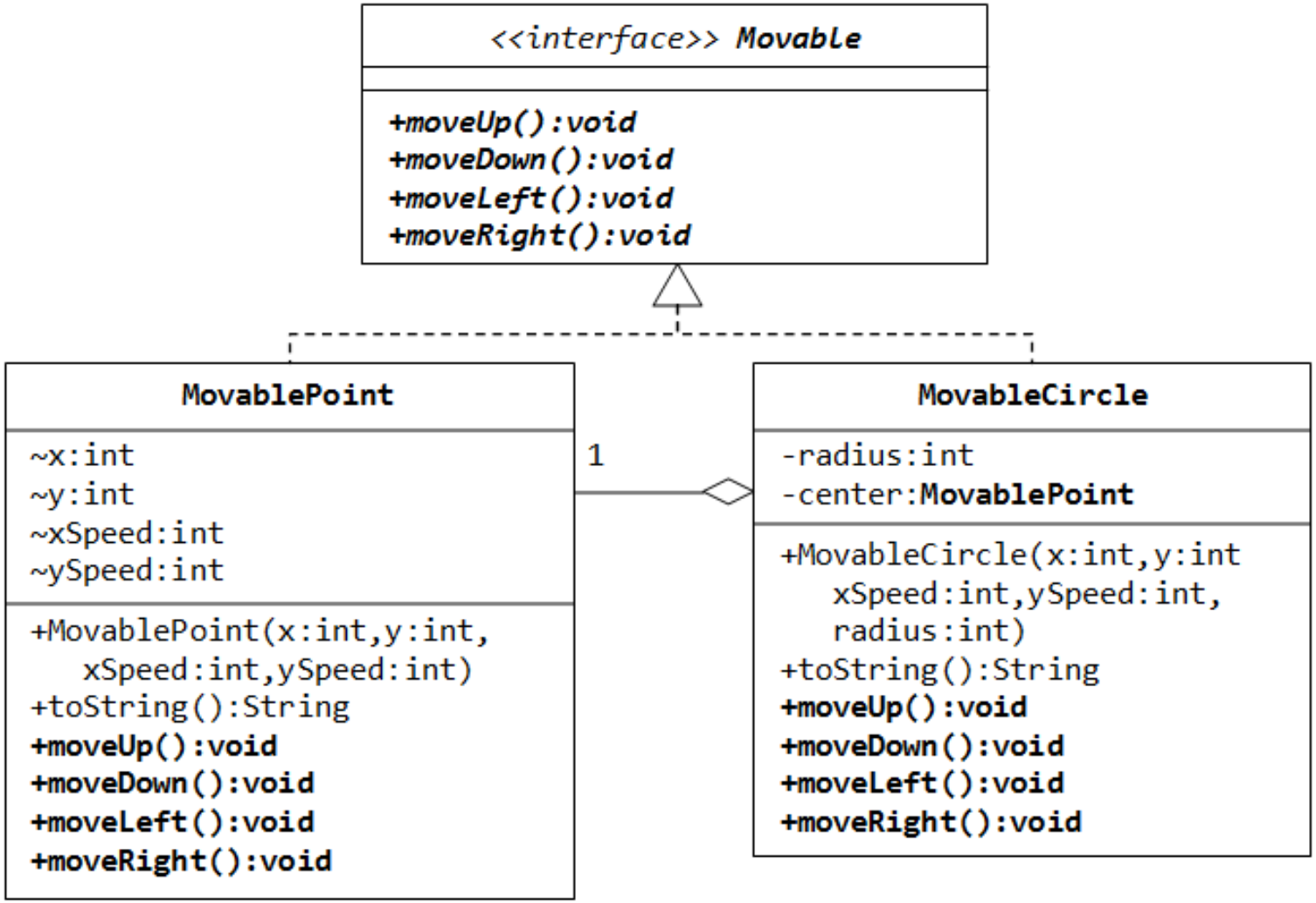
Поліморфізм

- **Поліморфізм – це здатність об’єктів реагувати на одне і те ж повідомлення (або виклик метода) згідно зі своїм типом**. (Підтримка виконання потрібних дій в залежності від типу переданого об’єкту)
- **Існують два типи поліморфізму: поліморфізм успадкування і інтерфейсний поліморфізм.**
- **При виклику методу зазвичай відповідає або метод успадкований від суперкласу, або більш спеціалізований варіант цього методу, створений в інтересах саме даного підкласу.**
- **Інтерфейсний поліморфізм не потребує наявності відносин успадкування між класами; потрібно тільки, щоб в інтерфейсах об’єктів були методи з одним і тим же ім’ям.**

Поліморфізм успадкування



Інтерфейсний поліморфізм



jjj

Polymorphism

- Normally we have this when we create an object:

```
Dog dog = new Dog();
```

- Polymorphism allows us to also do this:

```
Animal pet = new Dog();
```

  - The object reference variable can be a super class of the actual object type! (Does NOT work the other way around: Dog is an Animal but Animal is not necessarily a Dog)

Polymorphic Array Example

```
Animal[] myPets = new Animal[5];
myPets[0] = new Cat();
myPets[1] = new Cat();
myPets[3] = new Dog();
```

You can put any subclass of Animal in the Animal array!

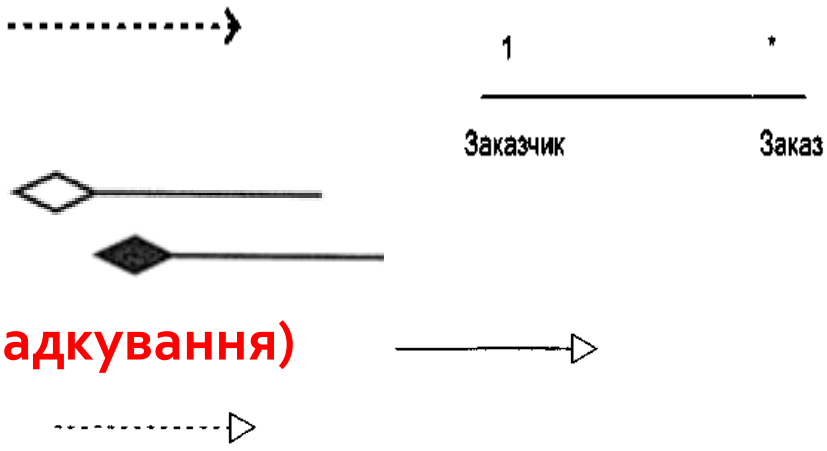
```
for (int i = 0; i < myPets.length; i++) {
    myPets[i].feed();
}
```

Polymorphic Arguments

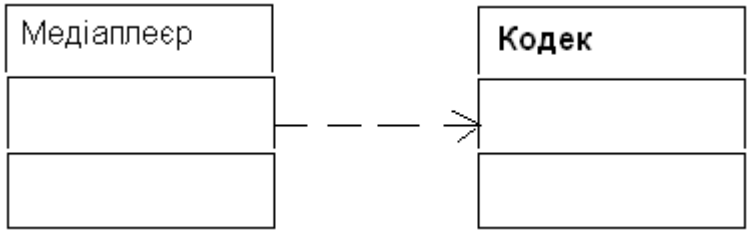
```
public class Vet {
    public void giveShot(Animal pet) {
        pet.makeNoise();
    }
}

public class PetOwner {
    Vet vet = new Vet();
    Dog dog = new Dog();
    Cat cat = new Cat();

    vet.giveShot(dog);
    vet.giveShot(cat);
}
```

Типи зв’язків між класами

- **Залежність**
- **Асоціація**
  - **Агрегація**
  - **Композиція**
- **Узагальнення (спадкування)**
- **Реалізація**

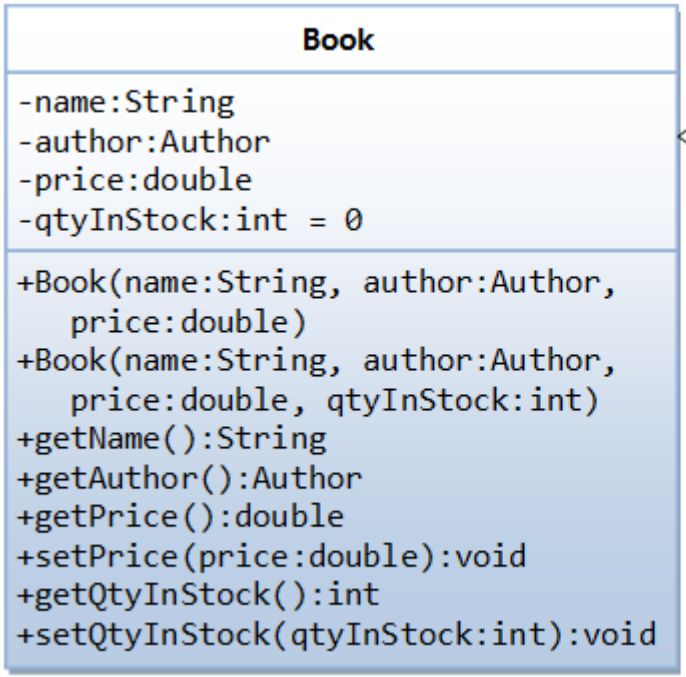


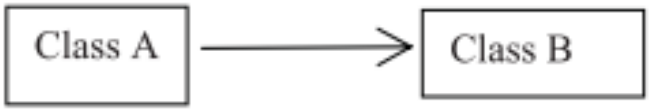
- **Залежність** (dependency, використання). **Це семантичне (смислове) відношення** між двома класами, коли зміни в одному класі (незалежному) можуть вплинути на залежний клас.

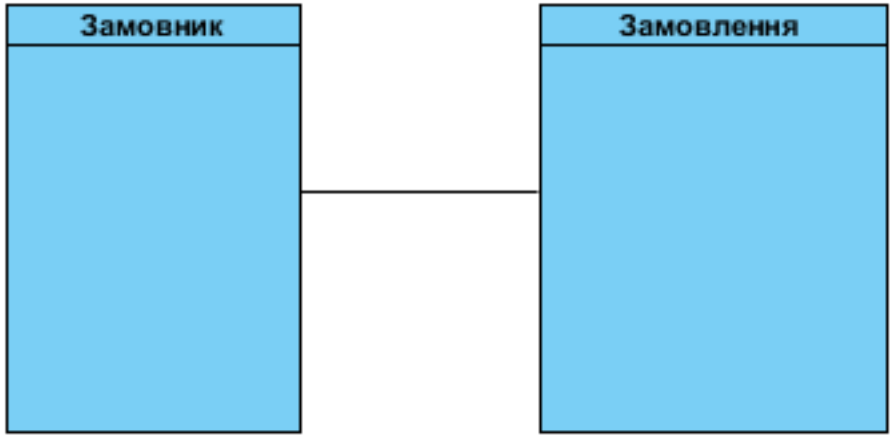


Типи зв'язків між класами

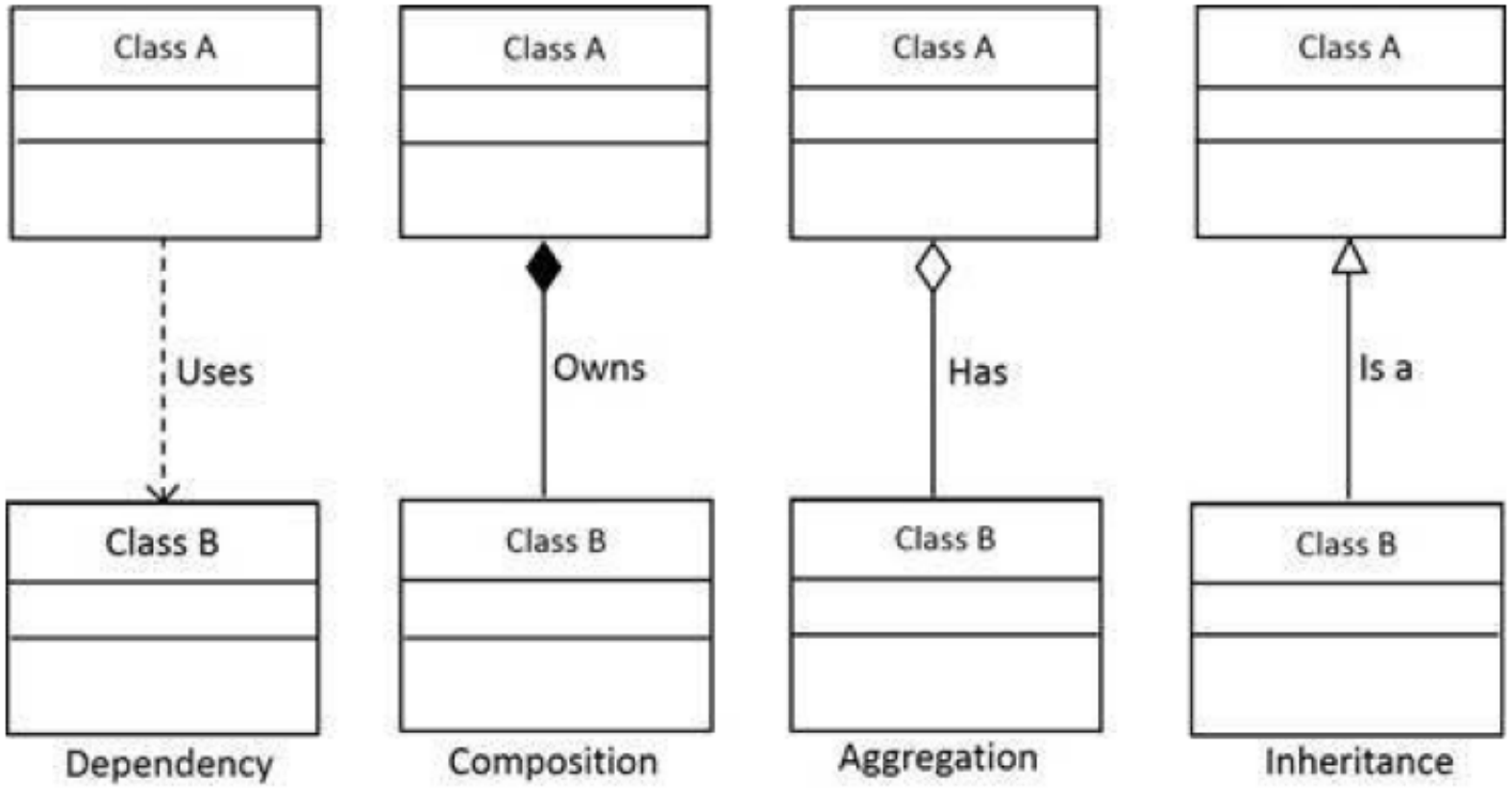
- **Асоціація – це структурний зв'язок**. (In an association Class A 'uses' objects of Class B)
- Окремі випадки асоціації – це **агрегація** і **композиція** (відношення між цілим та частинами цілого)
- Композиція – це більш тісний зв'язок.







Relationship



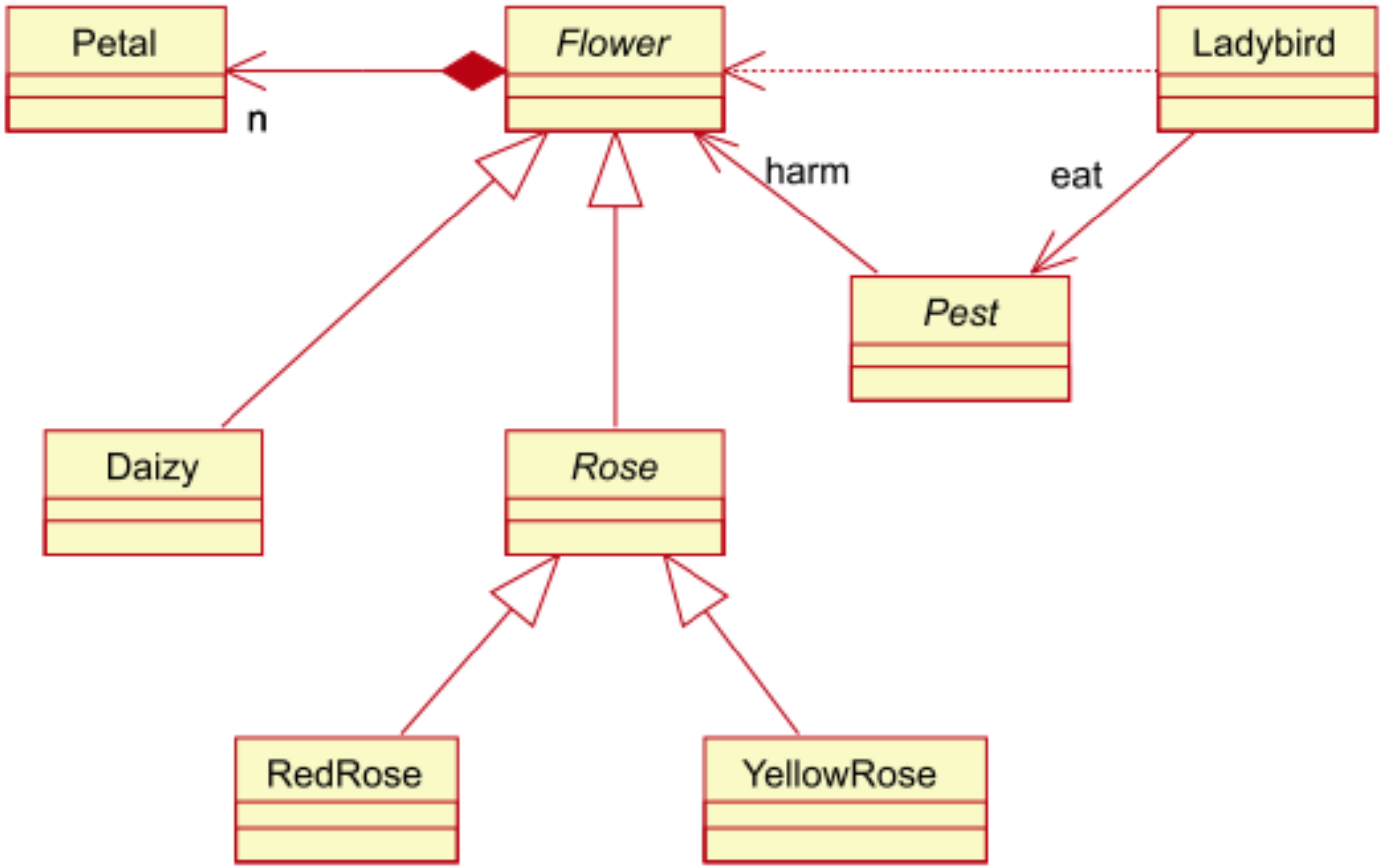
Приклад відношень між класами (1)

- **Розглянемо подібності та відмінності між наступними класами: квіти, маргаритки, червоні троянди, жовті троянди, пелюстки і сонечка**.

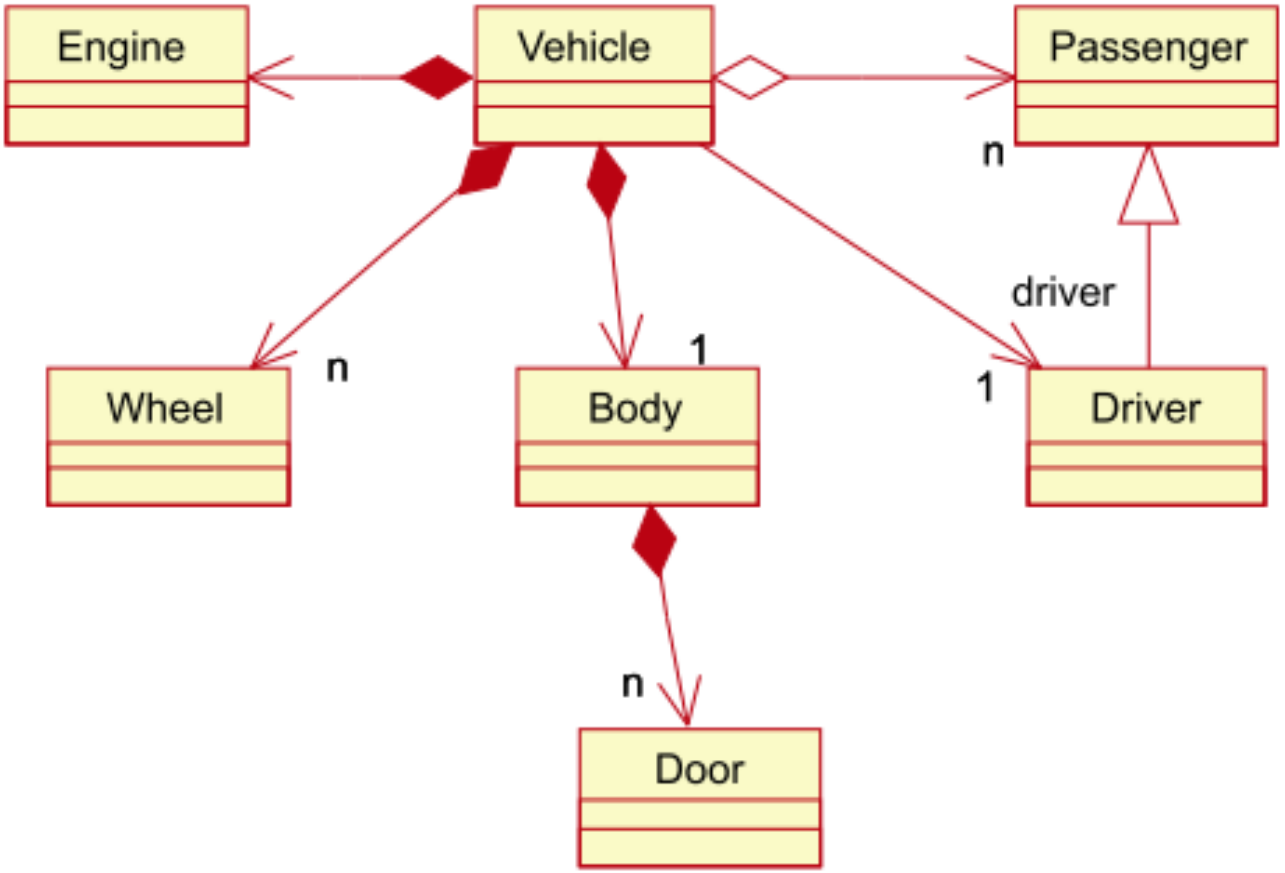
Помічаємо наступне:

- Маргаритка – квітка.
- Троянда – (інша) квітка.
- Червоні і жовті троянди – троянди.
- Пелюстки є частиною квітки.
- Сонечка харчуються шкідниками, що вражають деякі квіти.

Приклад відношень між класами (2)

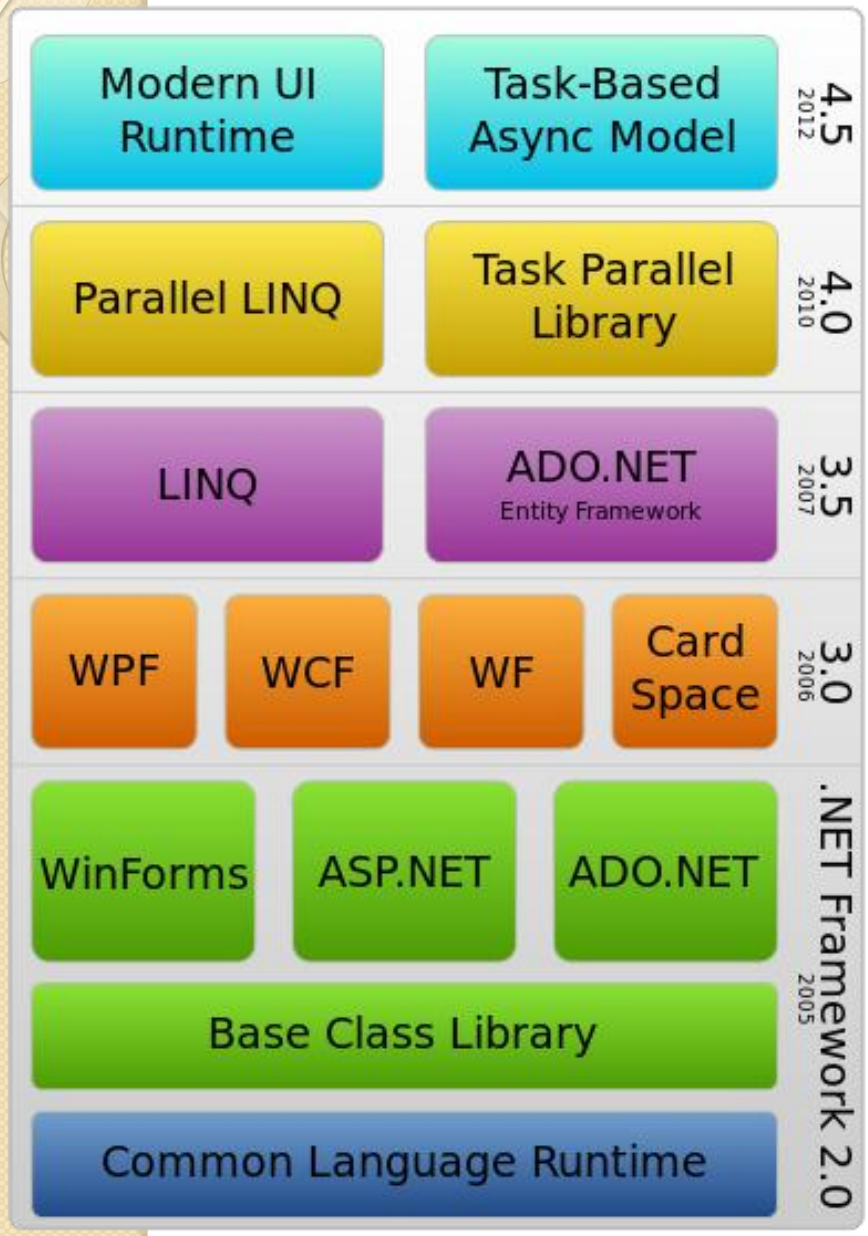


Інший приклад діаграми класів

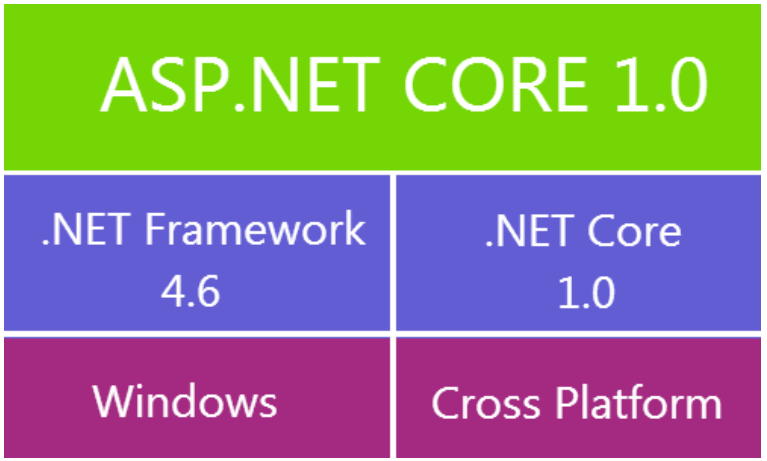


.NET Framework у основних поняттях

- **Платформа у контексті інформаційних технологій – середовище, що забезпечує виконання програмного коду**.
- **Microsoft .NET Framework – програмна технологія, запропонована фірмою Microsoft як платформа для створення Desktop-застосунків, веб-програм, мобільних застосунків**.
- **.NET Framework – це каркас, середовище для розробки, запуску і виконання програм**.
- **.NET Framework поділяється на дві основні частини** – середовище виконання (по суті віртуальна машина) – **CLR** (**Common Language Runtime**) та бібліотеку класів – **FCL** (**Framework Class Library**).
- **Однією з ідей .NET є сумісність програм, написаних різними мовами.**



The .NET Framework Stack



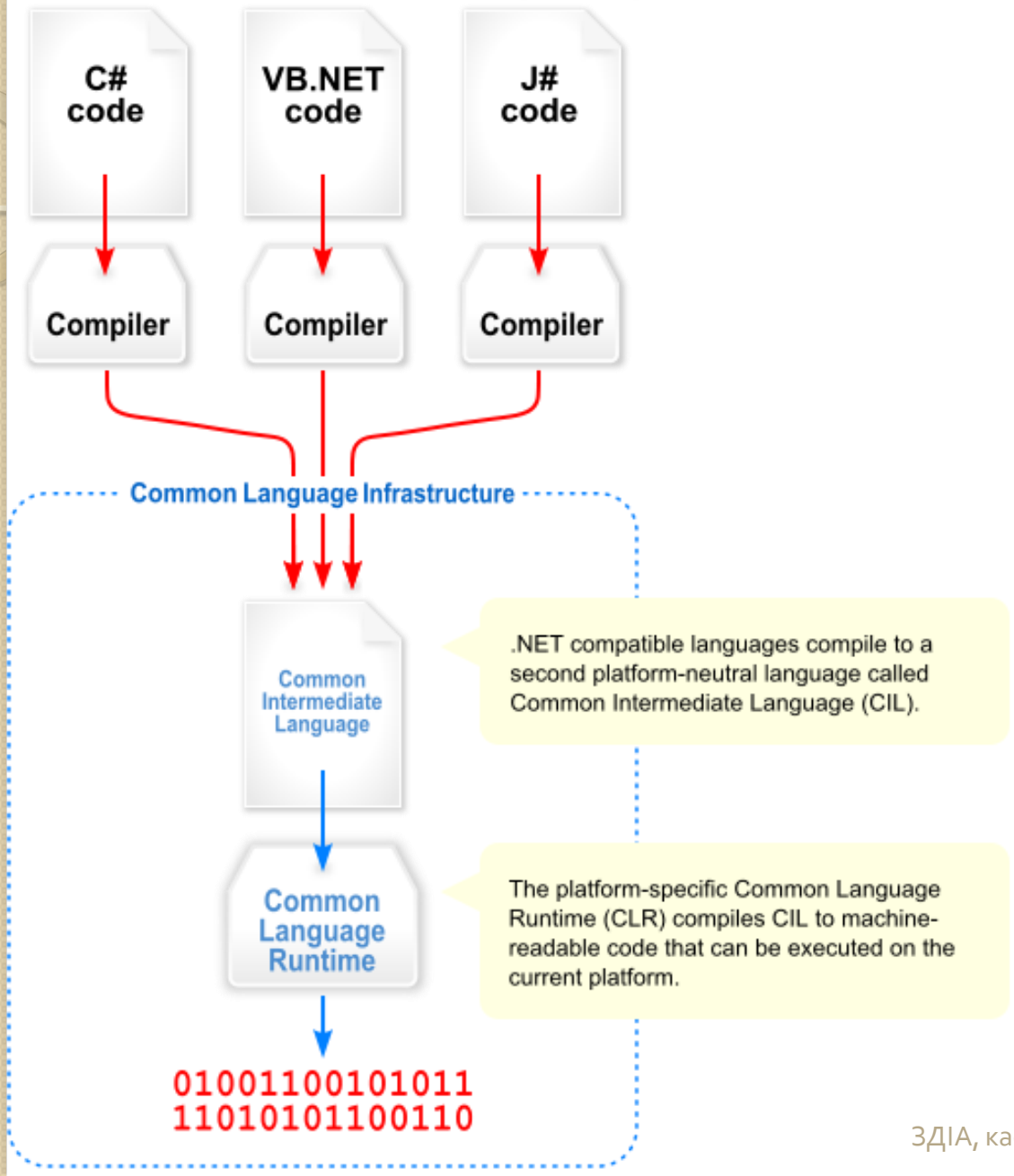
.NET Framework у основних поняттях

- **CLS (Common Language Specification)** – загальна специфікація мов програмування.
- **Керований (managed) код** – програмний код, який під час виконання здатний використовувати служби, що надаються CLR. (А є ще некерований - unmanaged)
- **FCL (або Base Class Library)** – відповідна CLS специфікації **об'єктно-орієнтована бібліотека класів, інтерфейсів і системи типів**, які включаються до складу платформи .NET. Забезпечує доступ до функціональних можливостей системи і **призначена бути основою при розробці .NET-додатків**.
- **MSIL (або IL, або CIL)** – Intermediate Language – проміжна мова Microsoft .NET. (аналог мови асемблера для байт-коду .NET), CIL – Common Intermediate Language

.NET Framework у основних поняттях

- Кросплатформеність
  - Різні версії Windows
  - Частково клони UNIX (http://www.mono-project.com)
  - Mac OS (http://www.mono-project.com)
- Підтримка декількох мов програмування
  - C#, VB, Managed C++, F#
  - ru.wikipedia.org/wiki/Список_.NET-языков
- CLI (англ. Common Language Infrastructure) - специфікація загальномовної інфраструктури
- Загальне середовище виконання для різних мов програмування
- Прозора міжмовна взаємодія
- Єдина система типів Common Type Specification – CTS
- Спрощена робота з пам'яттю
- Безпека
- Збирач сміття (Garbage Collector - GC) – відслідковує посилання на об'єкти.

Common Language Infrastructure (CLI)



How C# application is compiled and run

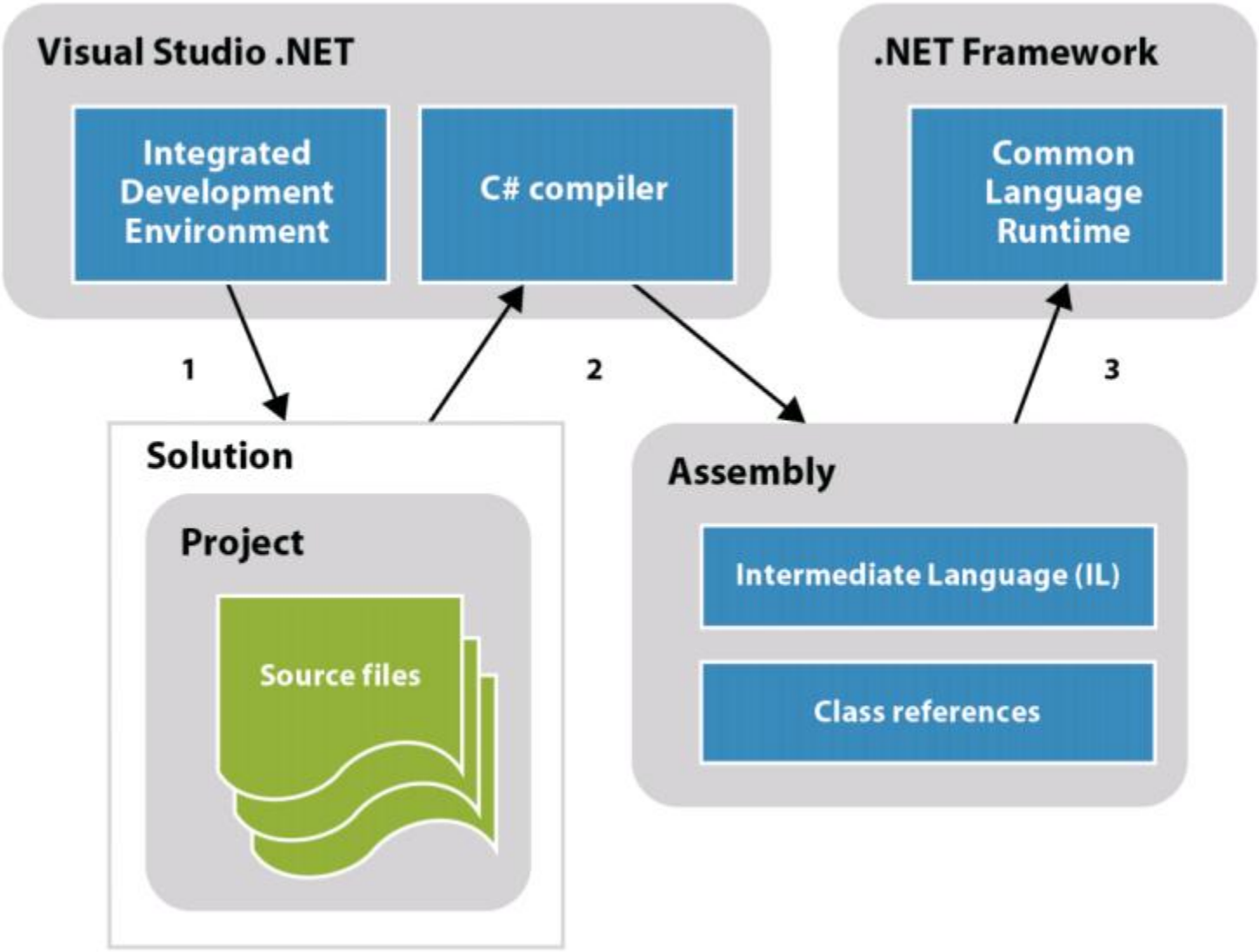
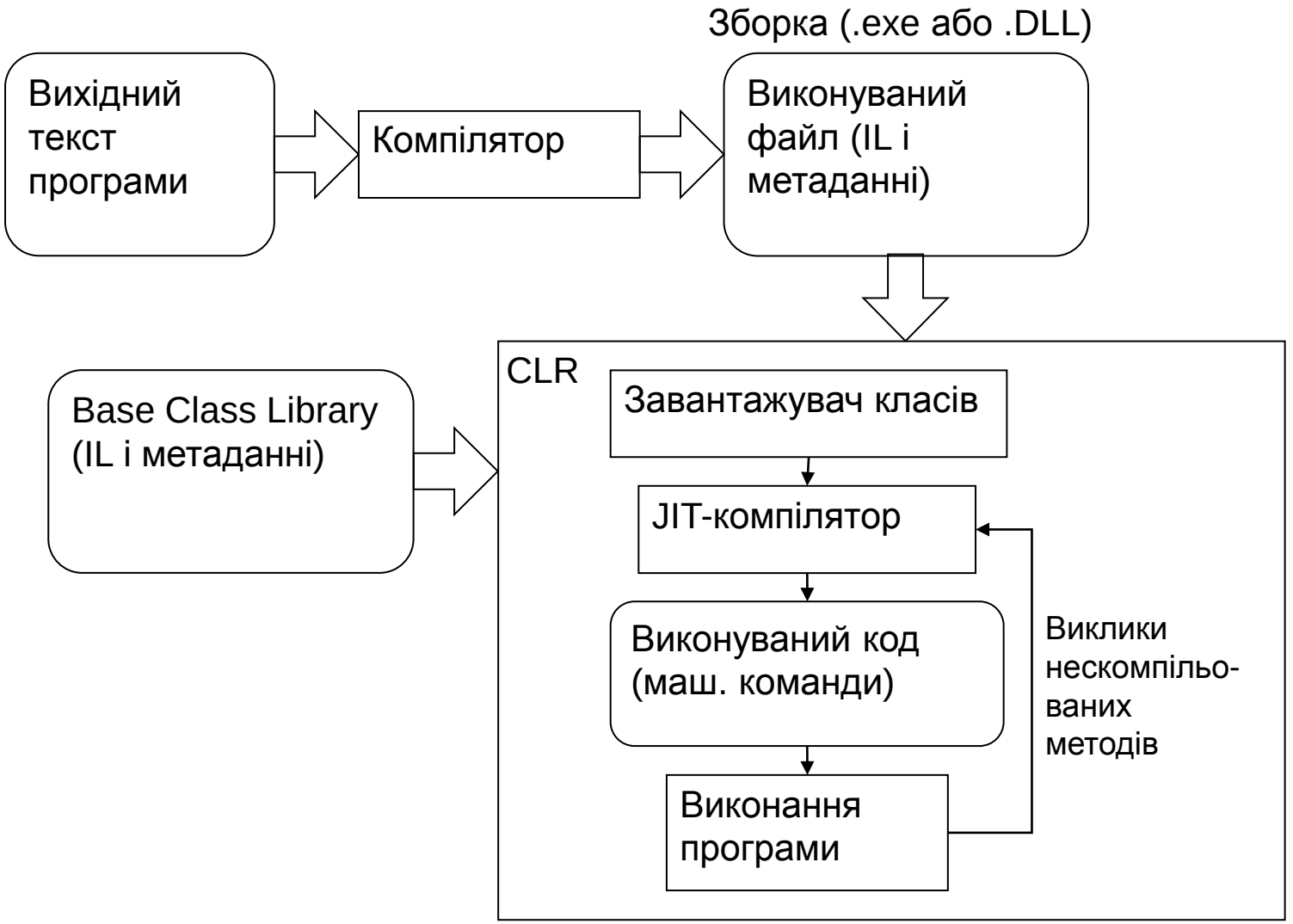


Схема виконання програми в .NET



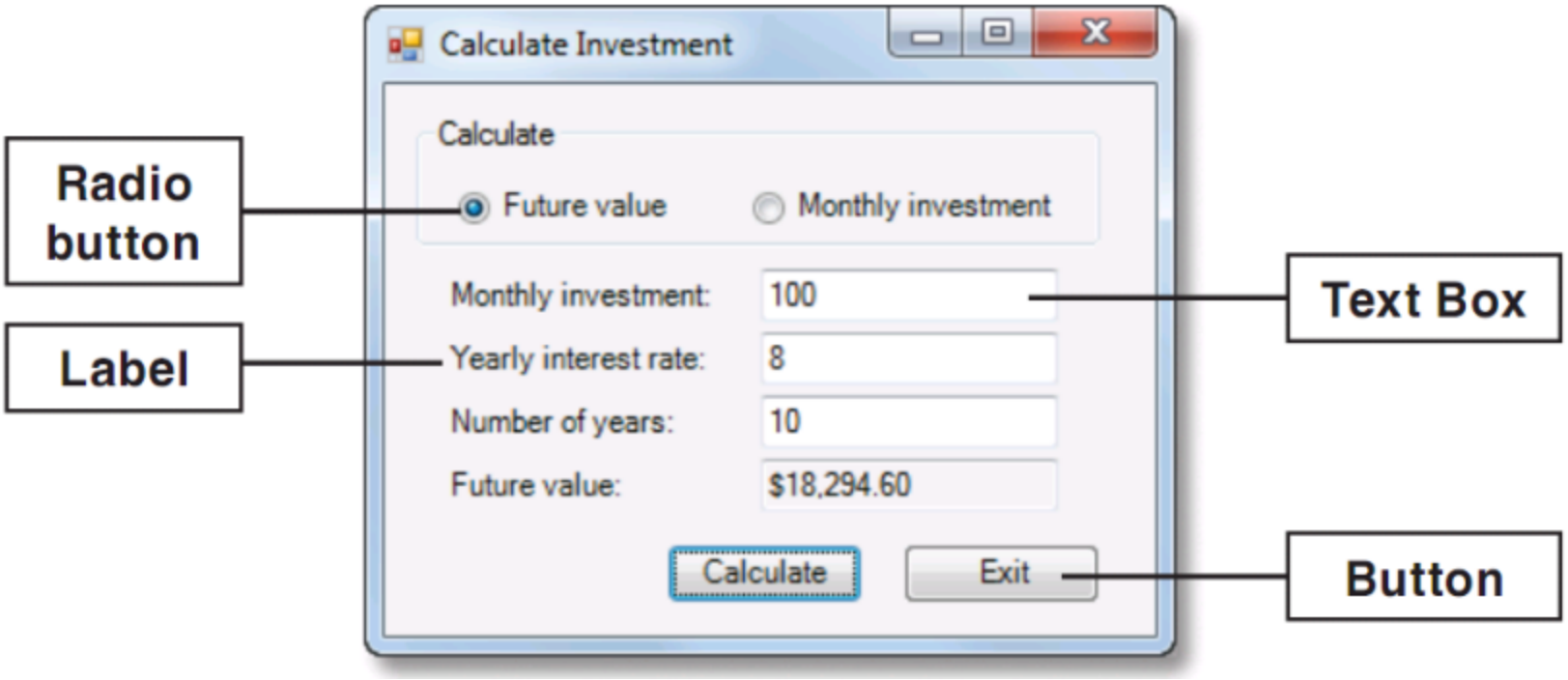
Загальні типи .NET застосунків

- Console Application

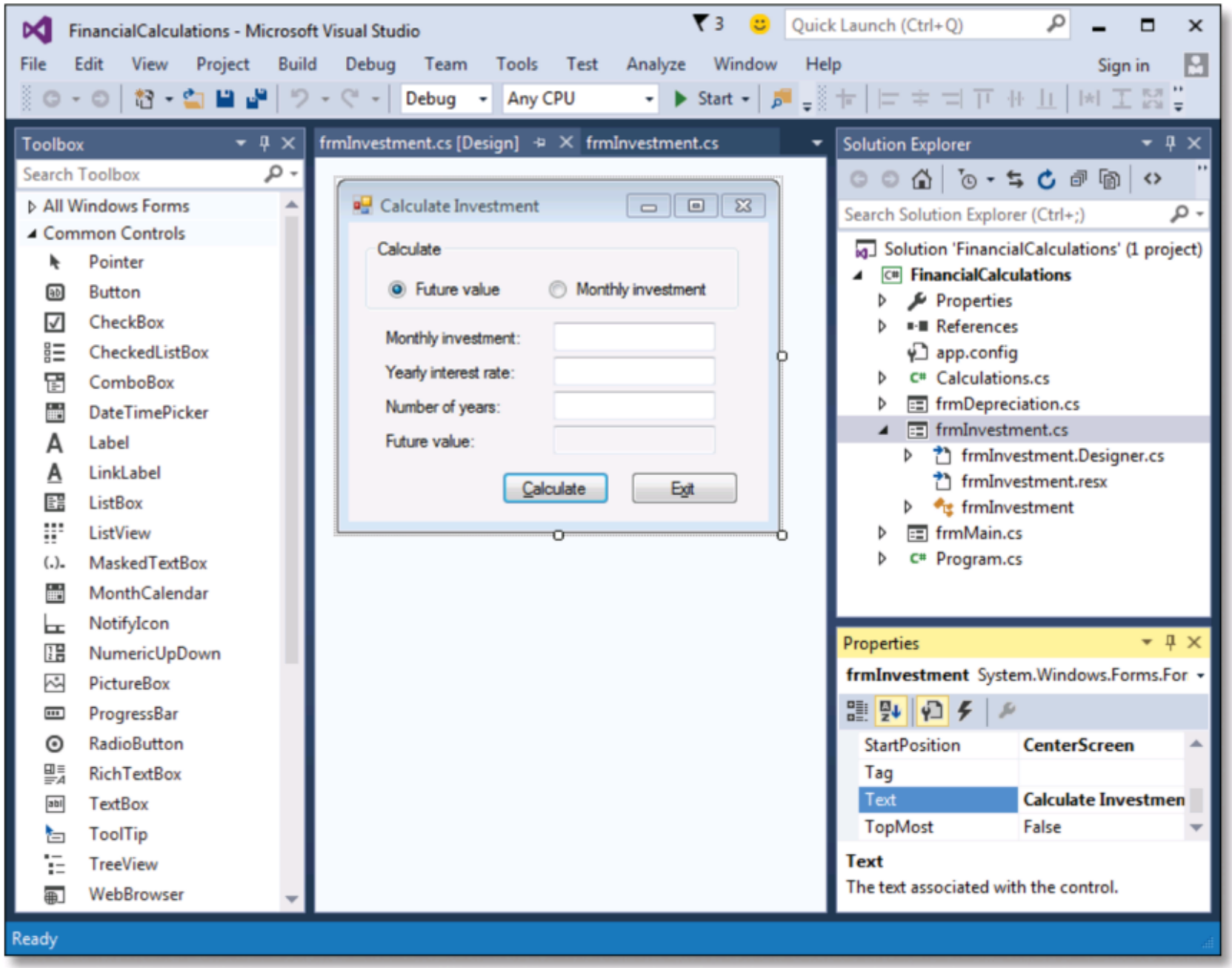
Type	Description
Windows Forms	Runs in its own window on the user's PC and consists of one or more Windows forms that provide the user interface for the application.
ASP.NET Web Forms	Runs on a web server and consists of one or more web forms that define the pages of the application. The pages are displayed in a browser on the client machine and provide the user interface for the application.
ASP.NET MVC	Runs on a web server and consists of pages that are displayed in a browser on the client machine. Unlike ASP.NET Web Forms, ASP.NET MVC uses a Model-View-Controller design pattern to create its pages.
WPF	Runs in windows on the user's PC and provides an enhanced user experience. Can also be made viewable in Internet Explorer.

- Mobile Application

Windows Forms Application



Visual Studio IDE

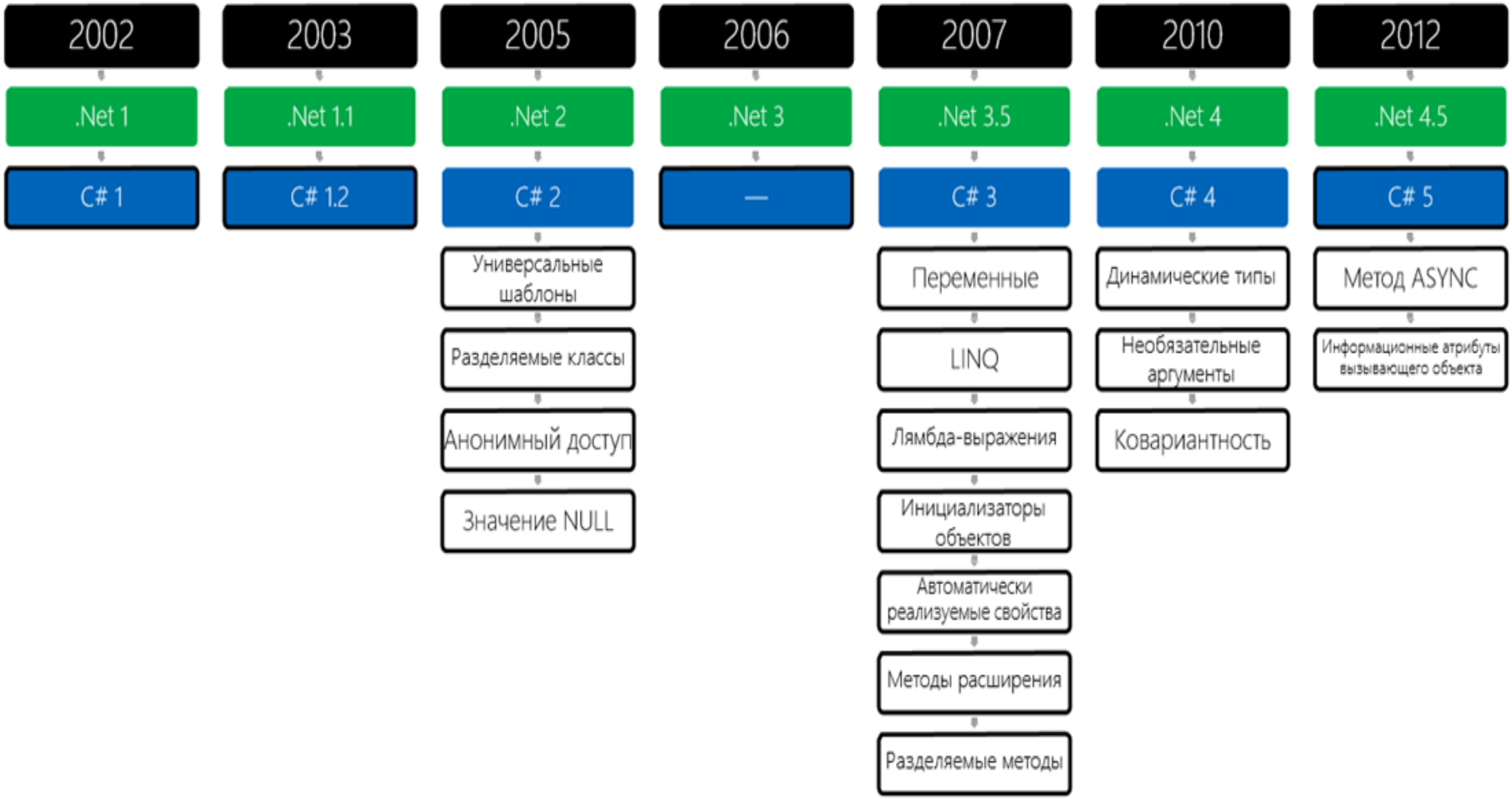


Основні поняття мови C#

- Мова C# розроблена спеціально для платформи .NET
- Головний розробник – Андреас Хейлсберг (Turbo Pascal, Delphi)
- За основу взято мови C, C++, звільнивши їх від небезпечного коду, пов'язаного з вказівниками.
- Засоби розробки:
  - Для Windows: Visual Studio, SharpDeveloper
  - Для FreeBSD та Mac OSX: Shared Source CLI
- Перша програма

```
using System;
namespace HelloWorld {
class EntryPoint
{
    static void Main()
    {
        Console.WriteLine("Hello, C# World!");
    }
}
}
```

Версії мови C#



Коментарі в C#

1) **// single-line comment**
2) **/* multi-line comment */**
3) One single-line documentation comment that starts with “///”,
4) One multi-line documentation comment that is delimited by “/**” and “*/”

```
/// <summary>Class level documentation.</summary>
class MyApp
{
  /** <summary>Program entry point.</summary>
      <param name="args">Command line arguments.</param>
   */
  static void Main(string[] args)
  {
    System.Console.WriteLine("Hello World");
  }
}
```

Ключові слова C#

abstract	enum	long	stackalloc
as	event	namespace	static
base	explicit	new	string
bool	extern	null	struct
break	false	object	switch
byte	finally	operator	this
case	fixed	out	throw
catch	float	override	true
char	for	params	try
checked	foreach	private	typeof
class	goto	protected	uint
const	if	public	ulong
continue	implicit	readonly	unchecked
decimal	in	ref	unsafe
default	int	return	ushort
delegate	interface	sbyte	using
do	internal	sealed	virtual
double	is	short	void
else	lock	sizeof	while

Contextual keywords

Some keywords are contextual, meaning they can also be used as identifiers—without an @ symbol. These are:

add	equals	join	set
ascending	from	let	value
async	get	on	var
await	global	orderby	where
by	group	partial	yield
descending	in	remove	
dynamic	into	select	

Зчитування даних з клавіатури

```
using System;
class YourAge{
static void Main(){
    string name;
    int age;
    Console.Write("Як Вас звати? ");
    name = Console.ReadLine();
    Console.WriteLine("Доброго дня, "+name+"!");
    Console.Write("Скільки Вам років? ");
    age = Int32.Parse(Console.ReadLine());
        //або age = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine("Вам "+age+" років!");
        // Очікування натискання клавіші
    Console.ReadKey();
 }
}
```

Основні поняття мови C#

- **Ідентифікатори** представляють собою імена констант, змінних, класів, методів, міток.
- Ідентифікатори можуть містити довільні Unicode-літери (в тому числі і нац. алфавітів), цифри, знак _
- В якості ідентифікаторів можуть бути зарезервовані слова, якщо їм передує символ @, наприклад, @bool
- Змінні треба ініціалізувати до першого використання

Система типів мови C#

- Всі типи поділяються на дві категорії: **Value Type** (**значущі**) та **Reference Type** (**посилальні**)
- Значущі зберігають дані безпосередньо (зберігаються в **стеку**).
- Посилальні зберігають посилання на значення (зберігаються в динамічній області пам'яті – **купі** (**heap**)).

Значущі та посилальні типи

```
int v = 123;
string s = "Hello There";
```

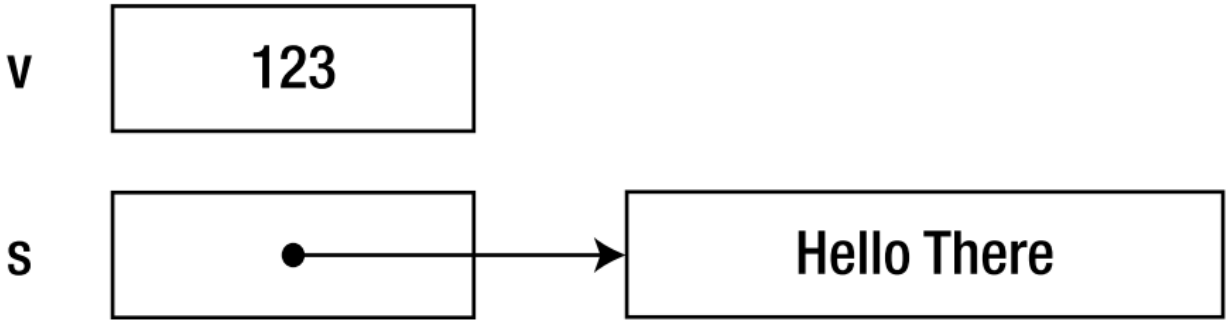
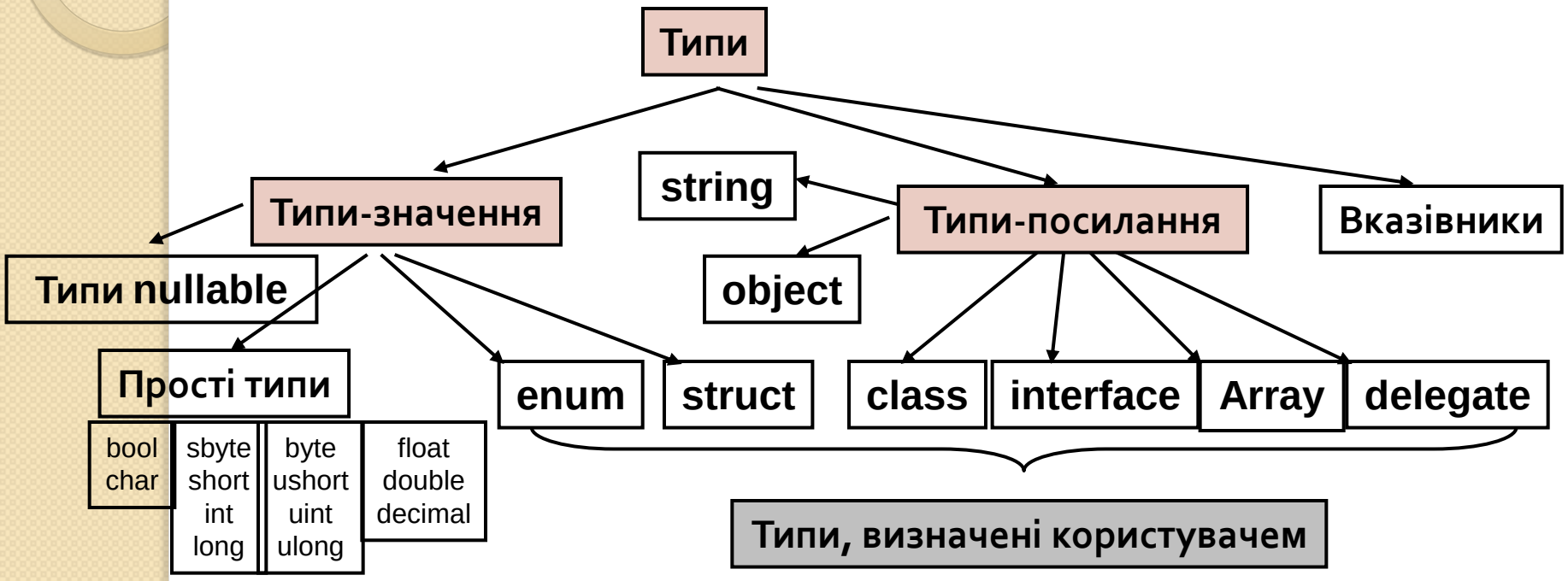


Figure 2-1. *Value and reference type allocation*

Система типів мови C#



Система типів мови C#

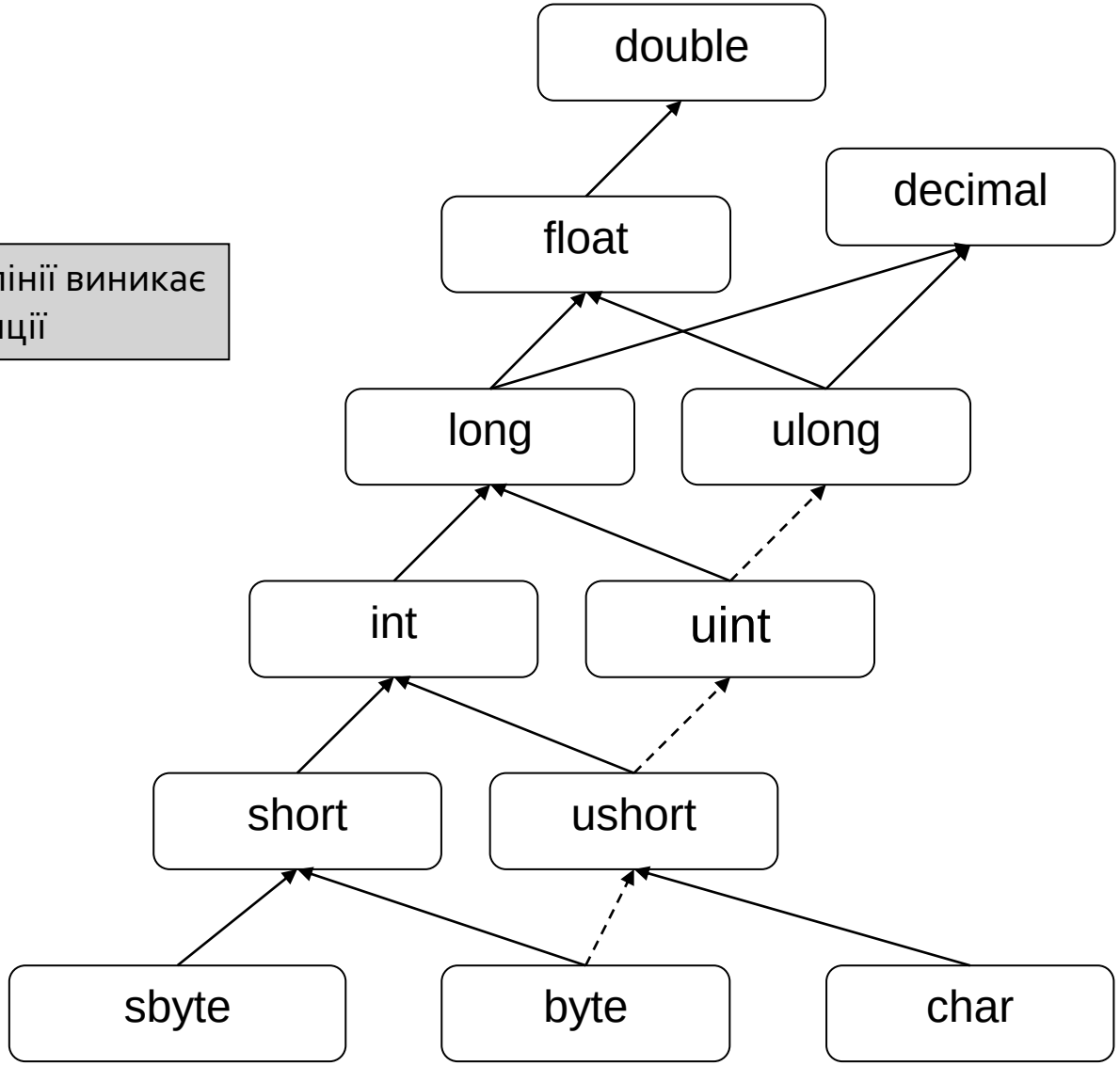
- **Базові вбудовані типи в C# не є внутрішніми по відношенню до мови, але є частиною платформи .NET Framework**
  - Наприклад, коли ви оголошуєте **int** в C#, то насправді оголошується екземпляр структури .NET – **System.Int32**
- C# має 15 вбудованих типів, 13 значущих і 2 посилальних (string та object)
- **Цілочисельні типи** (8): **sbyte** – 8-бітне зі знаком, **short** – 16-бітне зі знаком, **int** – 32-бітне зі знаком, **long** – 64-бітне зі знаком, **byte** – 8-бітне без знаку, **ushort** – 16-бітне без знаку, **uint**– 32-бітне без знаку, **ulong**– 64-бітне без знаку
- **Типи з плаваючою точкою**:
  - **float** (System.Single) – 32-бітне з плав.точкою одинар.точн . – 7 знаків
  - **double** (System.Double) – 64-бітне з плав.точк.подвій.точн. – 15-16 зн.

Приведення типів

- **Неявні** (implicit) та **явні** (explicit)
- Розширююче та звужуюче приведення
  - int x = 12345; //32-bit
  - long y = x; // Implicit conversion to 64-bit
  - short s = (short)x; //Explicit

Неявні приведення типів в C#

при відсутності лінії виникає помилка компіляції



Система типів мови C#

- **Фінансовий тип**: **decimal** (System.Decimal) – 128-бітне з плав.точкою в десятковій нотації – 28 знаків
- **Булівський тип**: **bool** (System.Boolean) – значення true та false
  - Не допускаються неявні перетворення значень bool в ціле
- **Символьний тип**: **char** (System.Char) – представляє окремий 16-бітний Unicode-символ
- **Тип object** (System.Object). **Це базовий тип, від якого спадкують всі типи C#**. Реалізує багато базових методів загального призначення, серед яких Equals(), ToString(), GetHashCode(), GetType(), …
- **Тип string** (System.String). Рядки є незмінюваними. Змінювані – клас System.Text.StringBuilder. Довжина рядка – властивість Length. Важливі операції: **конкатенація** (+), **індексація** ([ ]), **порівняння** (==).

Nullable-типи

- Мова C# підтримує концепцію типів даних, що допускають null.
- **Тип, що допускає null, може представляти всі значення типу плюс null**.
- Таким чином, якщо оголосити тип bool, що допускає null , то можна буде присвоювати значення з набору {true, false, null}.
  - Це може бути дуже зручно при роботі з реляційними базами даних, оскільки в таблицях баз даних досить часто зустрічаються стовпці, для яких значення не визначені.
- **Щоб визначити змінну типу, що допускає null, необхідно після імені необхідного типу даних додати знак питання (?), наприклад, bool? b = null;**
- Зверніть увагу, що такий синтаксис дозволено застосовувати тільки до типів значень.

Логічний тип bool

```
while(умова-продовження)
    оператор

do
  оператор
 while(умова-продовження)
```

Умова повинна мати логічний тип!

```
int i = 10;
while (i--)  // помилка!
    Console.Write(i);
```

```
if (умова)
    оператор1
else
    оператор2
```

```
if (умова)
   оператор
```

Умова – тільки тип bool

Упакування та розпакування (boxing and unboxing)

- **Процес перетворення значущого типу в посилальний називається упакуванням. Зворотній процес називається розпакуванням.**

- Приклад.

```
int v = 5;
object o = v; // boxing
v = 123;
Console.WriteLine(v+”, “+(int)o); // unboxing
```

Оператор switch

```
switch(day) {
        case 1:
                Console.Write("Понеділок");
                break;
        case 2:
        case 3:
                Console.Write("Вівторок або середа");
                break;
        default:
                Console.Write("Інший день тижня");
                break;
}
```

✓ Кожна альтернатива повинна завершуватися break, return або throw

✓switch() працює також з рядками!

Літерали

- **Це послідовності символів, що забезпечують явне представлення значень**
  - Використовуються при ініціалізації
- Розділяються літерали на арифметичні (різних типів), логічні, символьні, рядкові.
- **Арифметичні:** 25, 1.25E+08, 256L, …
  - Суфікси L – long; F – float; M – decimal; double без суфікса, але можна D, U – для uint, UL - для ulong
  - Console.Write(1.0.GetType()); //Double
- **Логічні**: true, false
- **Символьні**: ‘x’, (char)65, ‘\a’, ‘\n’
- **Рядкові**: звичайні “C:\\My Documents\\sample.txt”
  Verbatime string @”C:\My Documents\sample.txt”

Демонстраційний приклад

```
using System;
class MyApp{
    static void Main(string[] args)
    {
        int myInteger;
        string myString;
        myInteger = 17;
        myString = "\"myInteger\" is“;
        Console.WriteLine("{0} {1}.“, myString, myInteger);
        Console.ReadKey();
    }
}
```

Результат:

"myInteger" is 17

Операції

- В основному такі, як у C++

1)Арифметичні

```
float x = 3+2;
      x = 3-2;
      x = 3*2;
      x = 3/2; //1
      x = 3%2; //1
      x=3/(float)2; //1.5
```

2)Інкремент, декремент:

```
x++;  x--; ++x;  --x;
```

3)Порівняння

```
bool x=(2==3);
      x=(2!=3);
      x=(2>3);
```

4)Логічні

```
bool x=(true && false);
      x=!(true);
```

5)Порозрядні

int x=5&4; //101&100=100(4) and

x=5|4; // …=101(5) or

x=4<<1; // 100<<1=1000(8) left shift

x=4>>1; //100>>1=10(2) right shift

x=~4; //~00000100=1111101(-5)

Рядки

```
string a="Hello";
```

Concatenation

```
string b=a+" World";
```

String compare

```
bool c=(a==b); //false
```

StringBuilder class

```
using System.Text;
StringBuilder sb=new
   StringBuilder("Hello");
sb.Append(" World"); // Hello World
sb.Remove(0,5); // World
sb.Insert(0,"Bye"); // Bye World
```

String members

```
string a="String";
string b=a.Replace("i","o"); //Srtong
   b=a.Insert(0,"My "); //My String
   b=a.Remove(0,3); //ing
   b=a.Substring(0,3); //Str
   b=a.ToUpper(); //STRING
int i=a.Length; //6
```

Демонстраційний приклад

Декомпозиція і об’єднання рядків

```
using System;
class Program{
    static void Main(){
        string sent=“De gustibus non est disputandum”;
        string[] words;
        words=sent.Split(‘ ‘);
        Console.WriteLine(“words.Length=“, words.Length);
        foreach(string st in words)
                Console.Write(“{0} \t”, st);
        sent=string.Join(“-:-”, words);
        Console.WriteLine(“\n”+sent);
    }
}
```

Масиви в C#

Особливості.

- **Масив в C# - це тип даних за посиланням, масиви реалізовані як об'єкти**.
- **Ім'я масиву є вказівником** на область динамічної пам'яті, в якій послідовно розміщується набор елементів певного типу.
- **Виділення пам'яті** під елементи відбувається на етапі **ініціалізації** масиву.

Одновимірні масиви.

Оголошення: **тип[ ] ім'я = new тип[розмір];**

Наприклад, int[] a = new int[5];

На етапі оголошення масиву можна виконати його **ініціалізацію**, наприклад

int[] a = {3,1,5,7,1};//або int[] a = new int[5]{3,1,5,7,1};

Одновимірні масиви

Приклад 1. Масив заповнюється випадковими числами

```
static void Main(){
int[] ar = new int[10];
Random rand = new Random();
for(int i=0; i<10; i++){
    ar[i] = rand.Next(0,100);
    Console.Write(" {0}", ar[i]);
}
}
```

В C# елементам масиву автоматично присвоюються **значення за замовчуванням** в залежності від типу даних (для цілих – 0, для масиву об'єктів – null, для bool – false, для char – '\0', для string – пусті рядки).

Одновимірні масиви

Приклад 2. Заповнення масиву значеннями, введеними з клавіатури.

```
. . .
int[ ] ar;
Console.Write("\nВведіть розмір масиву:\t");
int size = Convert.ToInt32(Console.ReadLine());
ar = new int[size];
for(int i=0; i<ar.Length; i++){
   Console.Write("Введіть {0}-й елемент:\t", i+1);
   ar[i]=Convert.ToInt32(Console.ReadLine());
}
}
```

Масиви як спадкоємці класу Array

- **Одновимірні масиви успадковують від класу Array декілька властивостей і багато методів, які спрощують роботу з масивами.**
- **Властивості**: int Length – кількість елементів; int Rank – розмірність масиву.
- Деякі методи:
  - static int BinarySeach(),
  - static void Clear(),
  - static void Copy(),
  - static int IndexOf(),
  - static void Reverse(),
  - static void Sort(),

Багатовимірні масиви

- Існують два види багатовимірних масивів:
  - **прямокутні** і
  - **зубчасті**, або нерівні (jagged).
- Прямокутні масиви являють собою n-мірний блок пам'яті, а зубчастий масив - це масив масивів.

Прямокутні масиви

- **Оголошуються за допомогою ком, що розділяють виміри, наприклад**

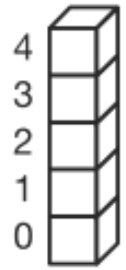
```
int[,] matrix = new int [3, 3];
```

- Метод масиву GetLength повертає довжину масиву в заданому вимірі (починаючи з нуля)

```
for (int i = 0;  i<matrix.GetLength(0);i++)
for (int j = 0;  j<matrix.GetLength(1);j++)
matrix [i, j ] =i*3 + j;
```

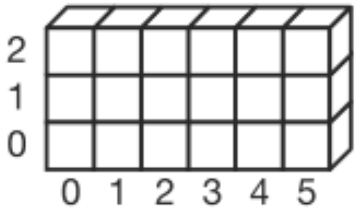
One-dimensional, rectangular, and jagged arrays

One-Dimensional Arrays

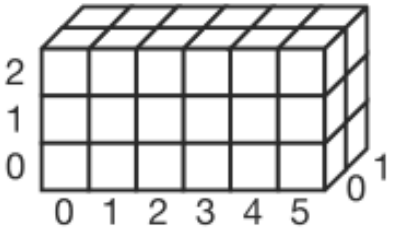


One-Dimensional
`int[5]`

Rectangular Arrays

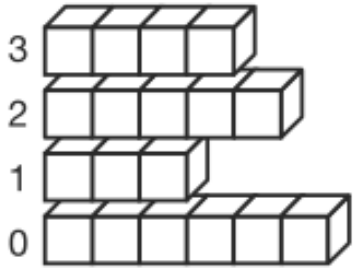


Two-Dimensional
`int[3,6]`



Three-Dimensional
`int[3,6,2]`

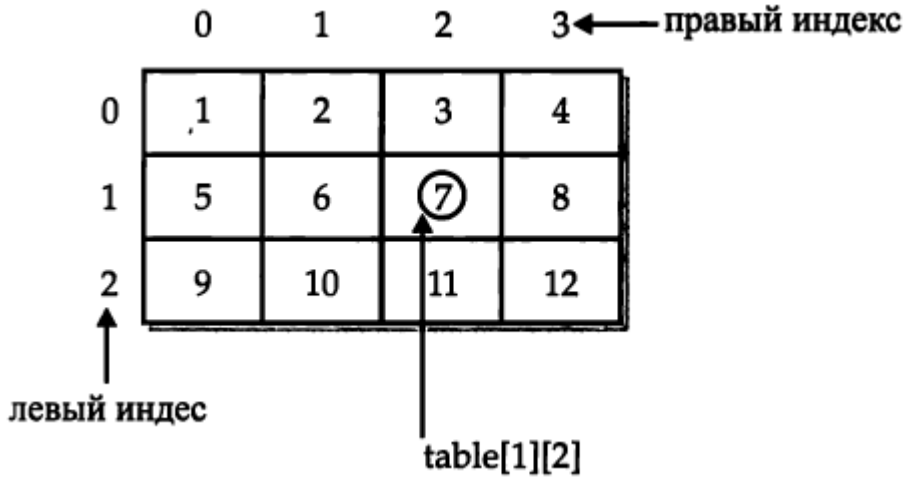
Jagged Arrays



Jagged Array
`int[4][]`

Прямокутні масиви

```
using System;
class TwoD {
static void Main () {
    int t, i;
    int[,] table = new int[3, 4];
    for(t=0; t < 3; ++t) {
    for(i=0; i < 4; ++i) {
            table[t,i] = (t*4)+i+I;
            Console.Write(table[t,i] + " ");
    }
Console.WriteLine();
}
}
}
```



Ініціалізація

```
using System;
class Squares {
static void Main () {
    int i, j;
    int[,] sqrs ={   {1,1},
                     {2,4},
                     {3,9},
                     {4,16},
                     {5,25}};
    for(i=0; i < 5; i++) {
          for(j=0; j < 2; j++)
                     Console.Write(sqrs[i,j] + " ");
          Console.WriteLine();
    }
}
}
```

Як вивести масив в DataGridView?

Нехай є масив string[,] mas розміром NxM

```
dataGridView1.RowCount = N;
dataGridView1.ColumnCount = M;
int i, j;
for(i = 0; i < N; i++)
    for(j = 0; j < M; j++)
            dataGridView1.Rows[i].Cells[j].Value = mas[i, j];
```

Властивості:

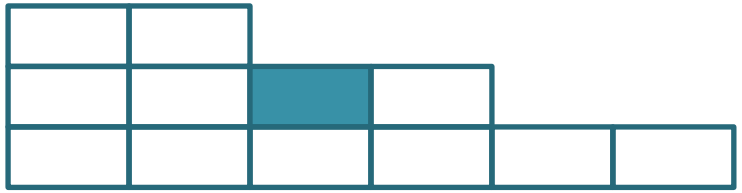
ColumnHeadersVisible = true або false

RowHeadersVisible = true або false

Зубчасті масиви

- **Ступінчастий масив являє собою масив масивів, в якому довжина кожного масиву може бути різною**.
- Отже, ступінчастий масив може бути використаний для складання таблиці з рядків різної довжини.
- **Зубчасті масиви оголошуються за допомогою послідовних квадратних дужок, що представляють кожний вимір**.

```
int[][] a = new int[3][];
for(int i=0; i<3; i++)
   a[i] = new int[2*i+2];
. . .
int p = a[1][2];
```



Класи в C#

Оголошення класу

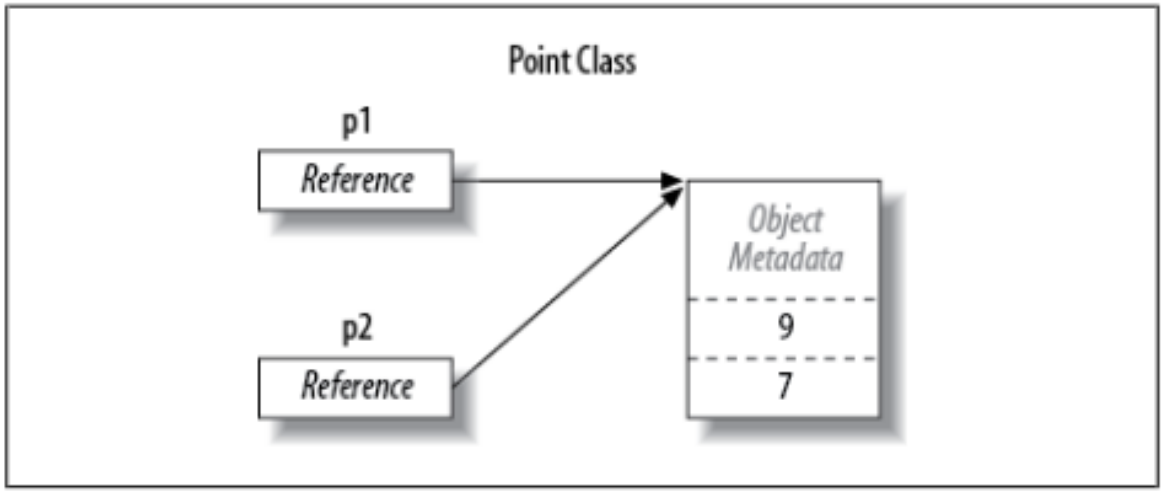
```
class Customer
{
    /* Члени класу: константи, поля, конструктори,
    деструктор, властивості, методи, індексатори,
    події, делегати, перевантажені операції  */
}
```

Перед ключовим словом class можуть стояти модифікатори: public, internal, abstract, sealed, static, unsafe, partial. (За замовчуванням internal – доступний тільки для даного проекту)

Більш складний клас може містити після імені класу *узагальнені параметри типу*, **базовий клас** і **інтерфейси**

Створення об'єктів

```
public class Point { public int X, Y; }
Point p1 = new Point();
p1.X = 7;   p1.Y = 7;
Point p2 = p1;            // Copies p1 reference
Console.WriteLine (p1.X);  // 7
Console.WriteLine (p2.X);  // 7
p1.X = 9;                 // Change p1.X
Console.WriteLine (p1.X);  // 9
Console.WriteLine (p2.X);  // 9
```



Класи в C#

- Модифікатор static для класу означає, що цей клас є контейнером констант та статичних операцій. Всі його елементи повинні бути static
- Модифікатор sealed для класу забороняє іншим класам спадкувати від цього класу (“запечатаний” клас).
- Для методу або властивості sealed означає заборону перевантаження при спадкуванні.
- Модифікатор abstract може використовуватися з класами, методами, властивостями, індексаторами та подіями.
- Він вказує на те, що реалізація об’єкта є неповна або відсутня

Модифікатор abstract

```
abstract class ShapesClass
{
    abstract public int Area();
}

class Square: ShapesClass
{
    int side = 0;
    public Square(int n)
    {
        side = n;
    }
    public override int Area()
    {
        return side*side;
    }
. . .
}
```

Часткові (partial) класи

- Ключове слово **partial** дозволяє визначити клас, структуру або інтерфейс, розподілений по декільком файлам

```
// BigClassPart1.cs
partial class BigClass
{
   public void MethodOne(){…}
}

// BigClassPart2.cs
partial class BigClass
{
   public void MethodTwo(){…}
}
```

Все компілюється в єдиний тип

Поля класу

- Поле - це змінна, яка є членом класу

```
class Octopus
{
        string name;
        public int Age = 70;
}
```

- Поле може мати модифікатори: public, private, protected, internal, readonly, static. (За замовчуванням - private)
- Звичайні поля – це поля об’єктів
- Статичні поля – це поля класу
- Read-Only поля

Демонстраційний приклад

```
using System;
class Person{
    public static int year = 2014; // поточний рік
    public int age;
    public string name;
}
class Program{
    static void Main(){
    Person who; // Посилання на об’єкт
    who = new Person(); // Об’єкт класу
    who.name = “Peter”;
    who.age = 19;
    Console.WriteLine(“Имя: {0}, Год рождения: {1} ”,
          who.name, Person.year – who.age);
    }
}
```

Властивості в C#

- **Властивість (Property) – це член класу, який використовується як поле, але при звертанні до нього використовується як метод.**
- Кожна властивість має один або два методи доступу get та set.

```
class Person
{
    private string name;
    public string Name
    {
        get{ return name;}
        set{ name = value;}
    }
. . .
}
```

Використання:

```
Person p1=new Person();
p1.Name="Петренко";
. . .
Console.WriteLine("Name
{0}", p1.Name);
```

Ключове слово value – значення, записане у властивості

Властивості в C#

- Один з аксесорів get чи set може мати модифікатор доступу
  - **private set{name = value;}** // Присвоювати можна в конструкторі

Автовластивості (Automatic properties)

- Для властивостей в C# починаючи з версії 3.0 є більш короткий синтаксис:
  - **class Person**
  - **{**
  - **public string Name{get; set;}**
  - **}**

Конструктори (1)

- Визначаються за тим же принципом, що і в мові C++
  - **Це метод, що має таке ж ім'я, що і ім'я класу, та не повертає значення**
  - **Конструктор викликається при створенні об'єкту**
- Конструктор без аргументів – це конструктор за замовчуванням

```
class CoOrds
{
   public int x, y;
   public CoOrds(){
         x = 0; y = 0;
   }
}
```

Конструктор з параметрами

```
public   CoOrds(int x, int y){
            this.x = x;
            this.y = y;
}
```

Якщо клас не має конструктора за замовчуванням, то компілятор його автоматично створює.

Конструктори (2)

- **Якщо в класі визначено конструктор (який-небудь), то конструктор за замовчуванням автоматично не створюється.**

- **Закриті конструктори**.
  - Вживаються щоб не допустити створення екземплярів класів ззовні.
  - Зазвичай він використовується в класах, що містять тільки статичні елементи.
  - Приклад:
  - **class Nlog{**
  - **private Nlog(){ }**
  - **public static double e = Math.E; // 2.71826…**
  - **}**

Конструктори (3)

- **Статичні конструктори**
  - Використовуються для ініціалізації статичних даних
  - Викликається автоматично до створення першого екземпляра
  - Не має параметрів
  - class SimpleClass{
  - static readonly long baseline;
  - static SimpleClass(){
  - baseline = DateTime.Now.Ticks;
  - }}
- **Конструктор копії**
  - class Person{
  - private string name;
  - private int age;
  - public Person(Person p){
  - name = p.name;
  - age = p.age;
  - }…}

Конструктори (4)

- Виклик одних конструкторів з інших

```
class Car
{
   private string s;
   private uint uWheels;
   public Car(string model, uint n)
   {
         s = model;
         uWheels = n;
   }
   public Car(string model):this(model, 4)
   {
   }
. . .
```

Деструктори (фіналізатори)

- Як і в C++, деструктор – це метод, що має ім'я класу з префіксом "~" попереду: ~MyClass(){…}
- Деструктори використовуються для вивільнення ресурсів у випадку, коли використовується **некерований** код
- **Для керованого коду деструктор не потрібний (ресурси звільняє "прибиральник сміття")**
- Клас може мати тільки один деструктор
- Деструктор не має параметрів
- Деструктори не можуть спадкувати і перевантажуватись
- Деструктор неявним чином викликає метод Finalize() для базового класу об'єкта

Методи в C#

- Методи визначаються в рамках оголошення класу
- Методи містять опис операцій, доступних над об'єктами класу
- Оголошення методу: **[модифікатори] тип_повернення Ім'я_методу([параметри]){//Тіло методу}**
- Модифікатори доступу: public, private, protected, internal. (За замовчуванням - private)

```
class MyApp{
  void MyPrint(){
    System.Console.Write("Hello World");
    }
    static void Main(){
          MyApp m = new MyApp();
           m.MyPrint(); // Hello World
    }
}
```

Параметри методів

```
void MyPrint(string s1, string s2)
{
  System.Console.Write(s1 + s2);
}
```

Виклик методу

```
static void Main()
{
  MyApp m = new MyApp();
  m.MyPrint("Hello", " World"); // Hello World
}
```

Ключове слово params

- Дозволяє оголошувати методи зі змінним числом параметрів
- Щоб взяти змінне число аргументів певного типу, масив з params-модифікатором може бути доданий в якості останнього параметра в списку

```
void MyPrint( params string[] s)
{
  foreach (string x in s)
    System.Console.Write(x);
}

public void Test( string arg0, float arg1, params int[] args )
{
  }
```

Перевантаження (overloading) методів

- **Можна оголосити кілька методів з однаковими іменами** але параметри яких розрізняються за типом або числом.
- Це називається **перевантаженням** методів
- Наприклад, метод System.Console.Write, має 18 реалізацій

```
void MyPrint(string s)
{
  System.Console.Write(s);
}
void MyPrint(int i)
{
  System.Console.Write(i);
}
```

Optional parameters

- **Починаючи з C# 4.0, параметри методів можуть бути оголошені як опційні, шляхом надання значення за замовчуванням для них в оголошенні методу.**
- Коли метод викликається, ці необов'язкові аргументи можуть бути опущені, щоб використовувати значення за замовчуванням

```
class MyApp
{
  void MySum(int i, int j = 0, int k = 0)
  {
    System.Console.Write(1*i + 2*j + 3*k);
  }
  static void Main()
  {
    new MyApp().MySum(1, 2); // 5
  }
}
```

Параметри за значенням та за посиланням

- **При передачі параметрів значущого типу** передається тільки локальна копія змінної, так що якщо копія зміниться, це не вплине на оригінальну змінну.

```
void Set(int i) { i = 10; }
static void Main()
{
  MyApp m = new MyApp();
  int x = 0;              // value type
  m.Set(x);               // pass value of x
  System.Console.Write(x); // 0
}
```

Параметри за значенням та за посиланням

- Для посилальних типів C# здійснює передачу в метод за посиланням, тобто самого об'єкту стан якого можна змінювати.

```
void Set(int[] i) { i = new int[] { 10 }; }
static void Main()
{
  MyApp m = new MyApp();
  int[] y = { 0 };            // reference type
  m.Set(y);                    // pass object reference
  System.Console.Write(y[0]); // 10
}
```

Ключове слово ref

- **Змінна типу значення може бути передана за посиланням за допомогою ключового слова ref**, як в виклику так і в оголошенні методу.
- Це призведе до того, що змінна передається у вигляді посилання, і, отже, змінивши її можна буде оновлювати початкове значення

```
void Set(ref int i) { i = 10; }
static void Main()
{
  MyApp m = new MyApp();
  int x = 0;              // value type
  m.Set(ref x);           // pass reference to value type
  System.Console.Write(x); // 10
}
```

Ключове слово out

- **Іноді в метод необхідно передати неприсвоєну змінну за посиланням, щоб присвоїти їй значення в самому методі.**
- Для цього використовується ключове слово out

```
void Set(out int i) { i = 10; }
static void Main()
{
  MyApp m = new MyApp();
  int x;                  // value type
  m.Set(out x);     // pass reference to unset value type
  System.Console.Write(x); // 10
}
```

Статичні методи

- Мають модифікатор static
- Працюють із статичними полями
- Належать типу, а не окремим об’єктам (екземплярам).
- Для виклику статичного методу використовується синтаксис виду **Клас.Метод(параметри)**

Індексатори

- **Якщо в клас в якості поля входе колекція, наприклад, масив елементів, то в ряді випадків зручно звертатись до елементів колекції, використовуючи індекс**

  посилання_на_об'єкт[індекс]

- Описану можливість можна реалізувати за допомогою спеціальних членів класу – індексаторів.
- **Індексатор можна вважати різновидом властивості** (є аксесори get та set)
- **При оголошенні індексатор завжди іменується службовим словом this**.
- **Тип повернуваного значення для індексатора відповідає типу елементів колекції**
- Оголошення індексатора має наступний формат:

[модифікатори] **тип this[тип_індекса індекс]{ get{…} set{…}}**

Приклад (Подбельский В.В., стор.230)

- Клас Work_hours (з індексатором) для представлення відпрацьованих годин по днях тижня. В масиві 7 елементів (понеділок, вівторок,…,неділя). Відпрацьовані години – від 0 до 14.

```
using System;
class Work_hours{
   int[ ] days; //години по днях тижня
   public Work_hours(){//конструктор
         days = new int[7];}
   public int this[int d]{// індексатор
         get{ return (d<0|d>6)?-1:days[d];}
         set{ if(d<0||d>6||value<0||value>14)
         Console.WriteLine("Недопустимо: день={0},
   години={1}!", d, value);
         else days[d] = value;}
   }
}
```

```
class Program{
    static void Main(){
    Work_hours week = new Work_hours();
    week[0]=7;
    week[2]=17;
    week[3]=7;
    week[6]=7;
    Console.WriteLine("Робочий тиждень:");
    for(int i=0;i<7;i++)
          Console.Write("days[{0}]={1} ", i, week[i]);
    Console.WriteLine();
    Console.WriteLine("days[{0}]={1}\t", 8, week[8]);
}}
```

Результат:

```
Недопустимо: день=2, години=17
Робочий тиждень:
days[0]=7 days[1]=0 days[2]=0 days[3]=7 days[4]=0
   days[5]=0 days[6]=7
  days[8]=-1
```

Індексатор для матриці

```
class Matrix
{
    public const int n = 10;
    private int[,] elements = new int[n,n];
    public int this[int i, int j]
    {
        get{ return elements[i,j];}
        set{ elements[i,j] = value;}
    }
}
```

При такому описі доступне наступне використання:

```
Matrix a = new Matrix();
a[0,0] = 1;
a[1,5] = 5;
```

Спадкування в C#

Типи спадкування

- **Спадкування реалізації** (implementation inheritance) означає, що тип походить від базового типу, отримуючи від нього всі поля-члени та методи.
  - Похідний тип адаптує реалізацію кожного метода базового класу
- **Спадкування інтерфейсу** (interface inheritance) означає, що тип успадковує тільки сигнатуру методів, але не успадковує ніякої реалізації.

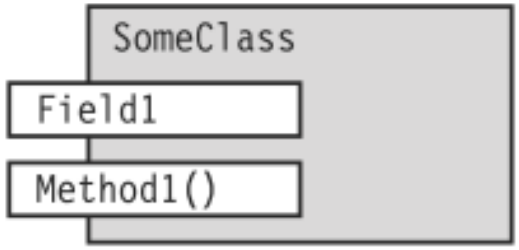
Множинне спадкування

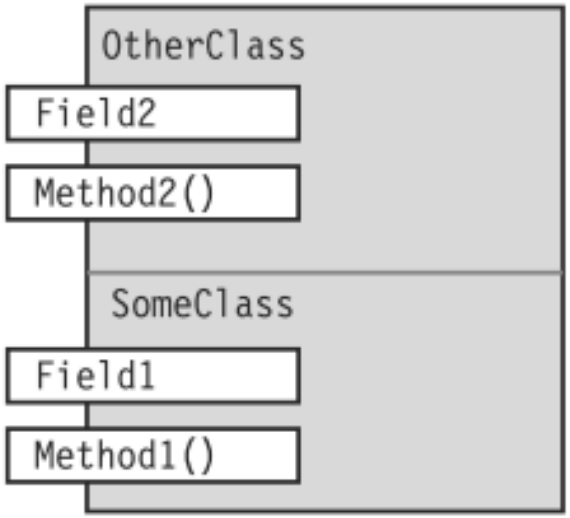
- Деякі мови, такі як C++, підтримують множинне спадкування, коли клас походить більш ніж одного базового класу.
- **Переваги множинного спадкування спірні**.
- Можна застосовувати множинне спадкування для написання надзвичайно складного, але при цьому компактного коду, що демонструє бібліотека C++ ATL.
- **Як вже згадувалось, полегшення написання стійкого коду було однією з ключових цілей проектування мови C#**.
- **Тому C# не підтримує множинне спадкування**.
- **Однак допускається множинне спадкування інтерфейсів**.

Спадкування класів

- **Успадкування дозволяє визначати новий клас, який об'єднує і розширює вже оголошений клас.**

```
                    Class-base specification
                                ↓
class OtherClass : SomeClass
{                   ↑       ↑
                    Colon   Base class
   ...
}
```





Base class and derived class

Спадкування класів

```
class SomeClass  {                    // Base class
    public string Field1 = "base class field ";
    public void Method1( string value ) {
      Console.WriteLine("Base class -- Method1:     {0}", value);
    }
  }
class OtherClass: SomeClass   {         // Derived class
    public string Field2 = "derived class field";
    public void Method2( string value ) {
      Console.WriteLine("Derived class -- Method2:  {0}", value);
    }
  }
  class Program  {
    static void Main() {
      OtherClass  oc = new OtherClass();
      oc.Method1( oc.Field1 );      // Base method with base field
      oc.Method1( oc.Field2 );      // Base method with derived field
      oc.Method2( oc.Field1 );      // Derived method with base field
      oc.Method2( oc.Field2 );      // Derived method with derived field
    }
  }
```

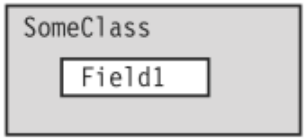
Маскування членів базового класу

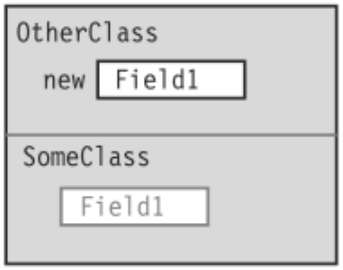
- **Похідний клас не може видалити будь-якого з членів що успадкований; однак, він може його маскувати (приховати) членом з тим же іменем.**
- **Щоб приховати поле, треба його оголосити з модифікатором new**
- **Щоб приховати успадкований метод, треба оголосити новий метод з тією ж сигнатурою.**
- **Сигнатура складається з імені і списку параметрів, але не включає тип що повертається**

.

Маскування членів базового класу

```
class SomeClass     // Base class
{
    public string Field1;
    ...
}
```






```
  class OtherClass : SomeClass // Derived class
  {
     new public string Field1; // Mask base member with same name
          . . .
  }
```

Доступ до поля базового класу (ключове слово base):

```
base.Field1
```

Приклад (Подбельский, стор. 250)

```
using System;
class Disk{ //Клас "Коло"
    protected double rad; //Радіус кола
    protected Disk(double ri){ rad = ri;}
    protected double Area{
        get{ return rad*rad*Math.PI;}
    }
}
class Ring: Disk{ //Клас "Кільце"
    new double rad; //Радіус внутрішній
    public Ring(double Ri, double ri): base(Ri){
        rad = ri;
    }
    public new double Area{
        get{ return base.Area – Math.PI*rad*rad;}
    }
```

Приклад (Подбельский, стор. 250)

```
public void print(){
        Console.WriteLine("Ring: Max_radius={0:f2},"+
"Min_radius={1:f2},"+"Area={2:f3}", base.rad, rad, Area);
}
}
class Program
{
   static void Main(){
        Ring rim = new Ring(10.0, 4.0);
        rim.print();
        Console.ReadKey();
   }
}
```

Результат:

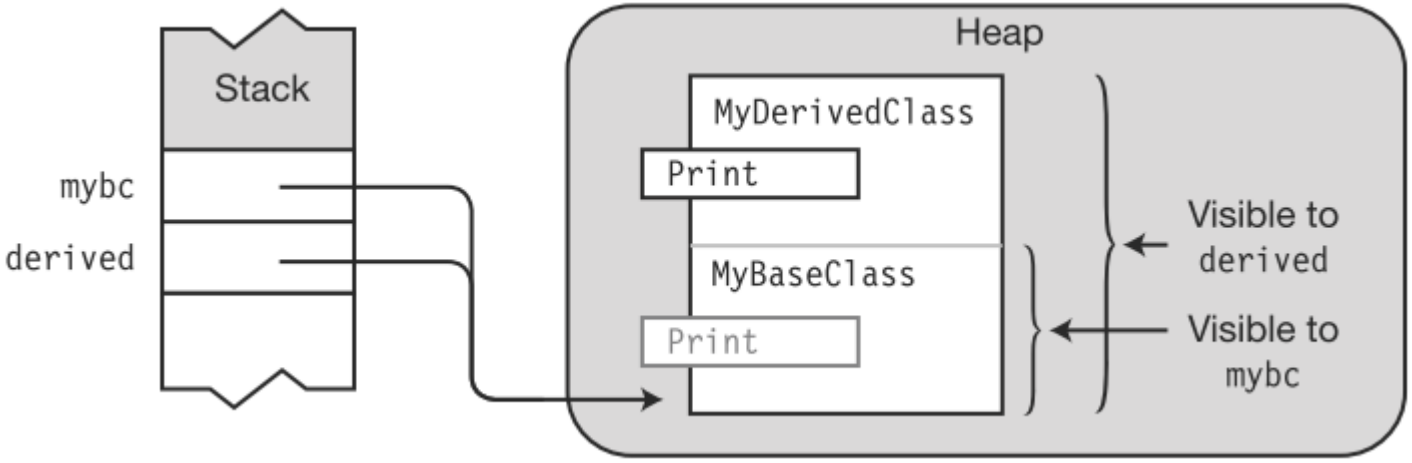
Ring: Max_radius=10.00, Min_radius=4.00, Area=263.894

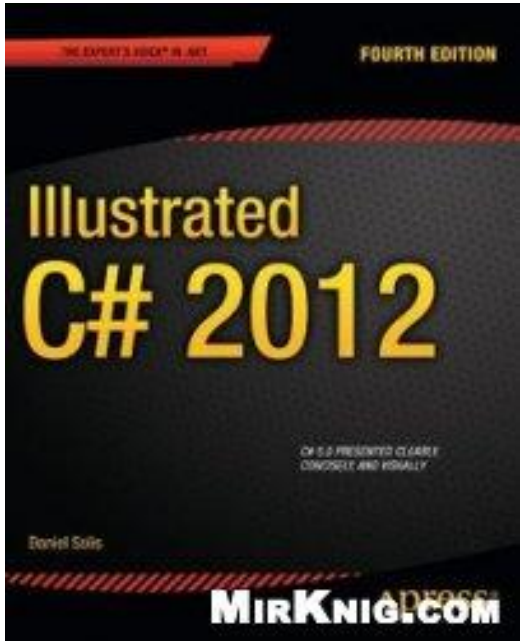
Using References to a Base Class

- **Примірник похідного класу складається з екземпляра базового класу плюс додаткових членів похідного класу.**
- **Посилання на похідний клас вказує на об'єкт всього класу, в тому числі базової частини класу.**
- **Якщо у вас є посилання на об'єкт похідного класу, ви можете отримати посилання тільки на базову частину об'єкта класу шляхом приведення до посилання на тип базового класу за допомогою оператора приведення.**

```
// Create an object.
MyDerivedClass derived = new MyDerivedClass();
 //Cast the reference.
MyBaseClass mybc = (MyBaseClass) derived;
```

Using References to a Base Class





Illustrated C# 2012, p.166 (169)

Using References to a Base Class

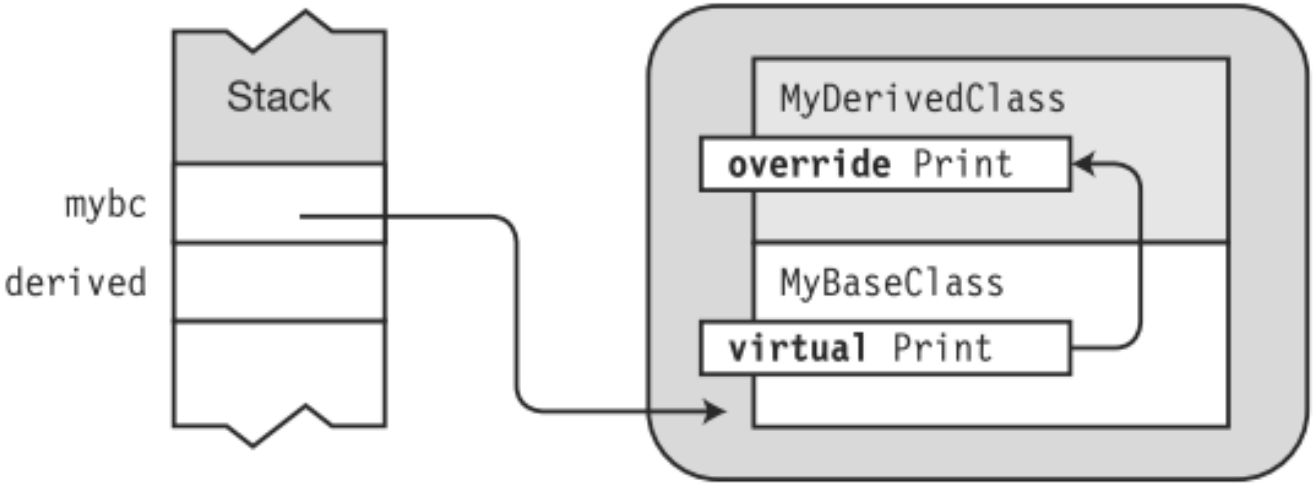
```
class MyBaseClass  {
    public void Print()  {
        Console.WriteLine("This is the base class.");
    }
}
class MyDerivedClass : MyBaseClass  {
    new public void Print()  {
        Console.WriteLine("This is the derived class.");
    }
}
class Program {
    static void Main() {
        MyDerivedClass derived = new MyDerivedClass();
        MyBaseClass mybc = (MyBaseClass)derived;
        derived.Print();          // Call Print from derived portion.
        mybc.Print();             // Call Print from base portion.
    }
}
```

Virtual and Override Methods

- **У попередньому розділі було показано, що, коли ви маєте доступ до об'єкта похідного класу, використовуючи посилання на базовий клас, ви отримаєте доступ тільки до членів з базового класу.**
- **Віртуальні методи дозволяють посиланню на базовий клас мати повний доступ до всього похідного класу.**
- **Ви можете використати посилання на базовий клас для виклику методу в похідному класі, якщо виконуються наступні умови:**
  - **Метод в похідному класі і метод в базовому класі кожен має ту ж саму сигнатуру і повернуване значення.**
  - **Метод в базовому класі позначений як віртуальний (virtual).**
  - **Метод в похідному класі позначений як перевизначений (override)**

Virtual and Override Methods

```
class MyBaseClass    // Base class
{
    virtual public void Print()
        ↑
    ...
 class MyDerivedClass : MyBaseClass  // Derived class
 {
    override public void Print()
        ↑
```



Virtual and Override Methods

```
class MyBaseClass  {
   virtual public void Print() {
      Console.WriteLine("This is the base class.");
   }  }
  class MyDerivedClass : MyBaseClass {
   override public void Print()  {
      Console.WriteLine("This is the derived class.");
   }  }
  class Program {
   static void Main() {
      MyDerivedClass derived = new MyDerivedClass();
      MyBaseClass mybc  = (MyBaseClass)derived;
       derived.Print();
       mybc.Print();
   }  }
```

```
This is the derived class.
This is the derived class.
```

Overriding Other Member Types

```
class MyBaseClass  {
    private int _myInt = 5;
    virtual public int MyProperty {
      get { return _myInt; }
    }  }
  class MyDerivedClass : MyBaseClass  {
    private int _myInt = 10;
    override public int MyProperty {
      get { return _myInt; }
    }  }
  class Program  {
    static void Main() {
      MyDerivedClass derived = new MyDerivedClass();
      MyBaseClass mybc       = (MyBaseClass)derived;
      Console.WriteLine( derived.MyProperty );
      Console.WriteLine( mybc.MyProperty );
    } }
```

```
10
10
```

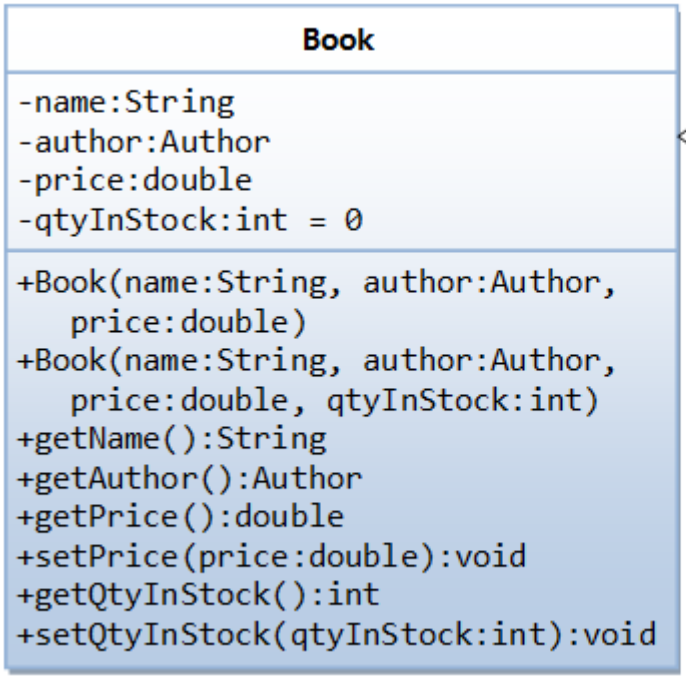
Висновки

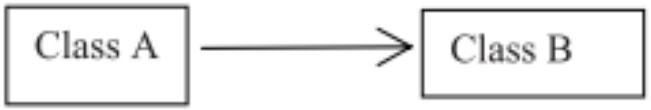
Якщо потомок (нащадок) створює метод з іменем, що співпадає з іменем метода предка, то можливі три ситуації:

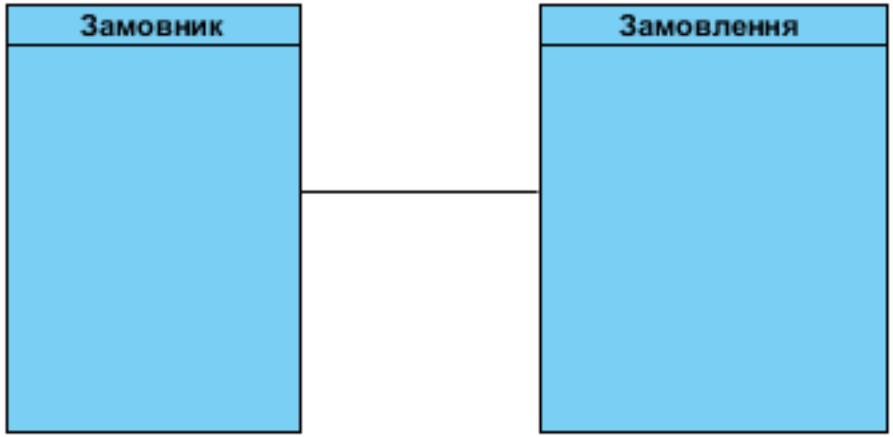
- **Перевантаження метода**. (Сигнатури не співпадають)
- **Перевизначення метода**. (Сигнатури однакові. У батька – virtual або abstract, у потомка - override)
- **Приховування метода**. (Сигнатури однакові. У потомка - new)

Типи зв'язків між класами

- **Асоціація – це структурний зв'язок**. (In an association Class A 'uses' objects of Class B)
- Окремі випадки асоціації – це **агрегація** і **композиція** (відношення між цілим та частинами цілого)
- Композиція – це більш тісний зв'язок.







Класи. Відношення композиції (1)

Приклад. (Подбельский, стор. 237)

```
using System;
class Point{
    double x, y;
    public double X {
            get{ return x;}
            set{ x=value;}
    }
     public double Y {
            get{ return y;}
            set{ y=value;}
    }
}
class Circle{
    double rad;
    Point centre=new Point();
    public double Rad {
        get{ return rad;}
        set{ rad = value;}
    }
    public double Len {
        get{ return 2*rad*Math.PI;}
    }
    public Point Centre {
        get{ return centre;}
        set{ centre = value;}
    }
    public void display(){
    Console.WriteLine("Centre:
        X={0}, Y={1}; Radius={2}" +
    "Length={3, 6:f2}", centre.X,
        centre.Y, this.rad, Len);
    }}
```

Класи. Відношення композиції (2)

```
class MyApp
{
   static void Main()
   {
         Circle rim = new Circle();
         rim.Centre.X = 10;
         rim.Centre.Y = 20;
         rim.Rad = 3.0;
         rim.display();
   }
}
```

В програмі нема об’єкта класу Point, що окремо існує. Тому це композиція.

Класи. Відношення агрегації

Клас Point як в попередньому прикладі.

```
class Circle{
    double rad;
    Point centre;
    public Circle( Point p,
    double rd)
    {
       centre = p;
       rad = rd;
    }
public double Rad{…}
public double Len{…}
public double Centre{…}
public void display(){…}
}
```

```
class MyApp
{
  static void Main()
  {
     Point pt = new Point();
     pt.X=10;
     pt.Y=20;
     Circle rim = new Circle(pt, 3.0);
     rim.display();
  }
}
```

Типи зв'язків між класами

- **Асоціація**
  - Інші назви: **"клієнт-постачальник"** , відношення **"має"** ("has"), відношення вкладеності
  - **Класи A і B знаходяться в відношенні "клієнт-постачальник", якщо одним з полів класу B є об'єкт класу A. (B – "клієнт", A – "постачальник")**
    - **Якщо в класі B створюються об'єкти класу A, або викликаються статичні методи класу A**
- **Спадкування (узагальнення)**
  - Інша назва: відношення **"являється"** ("є", **"is a"**)
- Приклади
  - Square та Rectangle
  - Car, Person та Person_of_Car (власник авто **має** автомобіль і **є** персоною)

Нові книги (травень 2014)

Название: Объектно-ориентированное мышление
Автор: Мэтт Вайсфельд
Год издания: 2014
Издательство: Питер
ISBN: 978-5-496-00793-1, 978-0321861276
Страниц: 304
Формат: PDF
Размер: 7,7 Мб (+3%)
Серия: Библиотека программиста

Объектно-ориентированное программирование - это фундамент современных языков программирования, включая C++, Java, C#, Visual Basic, .NET, Ruby и Objective-C. Кроме того, объекты лежат в основе многих веб-технологий, например javascript, Python и PHP. Объектно-ориентированное программирование обеспечивает правильные методики проектирования, переносимость кода и его повторное использование, однако для того, чтобы все это полностью понять, необходимо изменить свое мышление. Разработчики, являющиеся новичками в сфере объектно-ориентированного программирования, не должны поддаваться искушению перейти непосредственно к конкретному языку программирования (например, Objective-C, VB .NET, C++, C#, .NET или Java) или моделирования (например, UML), а вместо этого сначала уделить время освоению того, что автор книги Мэтт Вайсфельд называет объектно-ориентированным мышлением.
Несмотря на то, что технологии программирования изменяются и эволюционируют с годами, объектно-ориентированные концепции остаются прежними - при этом неважно, какой именно является платформа.

Обробка виняткових ситуацій в C#

- **Виняткова ситуація, або виняток – це помилка часу виконання в програмі (ділення на нуль і таке інше)**
- **Для обробки винятків в .NET передбачено базовий клас System.Exception та інші класи.**
- Довільний метод може **генерувати виняток** за допомогою ключового слова **throw**
- **Винятки можна обробити за допомогою блоку try…catch**
- Синтаксис оператора try:

```
try блок [блоки catch][блок finally]
```

- **Відсутнім** може бути або блоки **catch**, або блок **finally**, але не обидва одночасно

Обробка виняткових ситуацій в C#

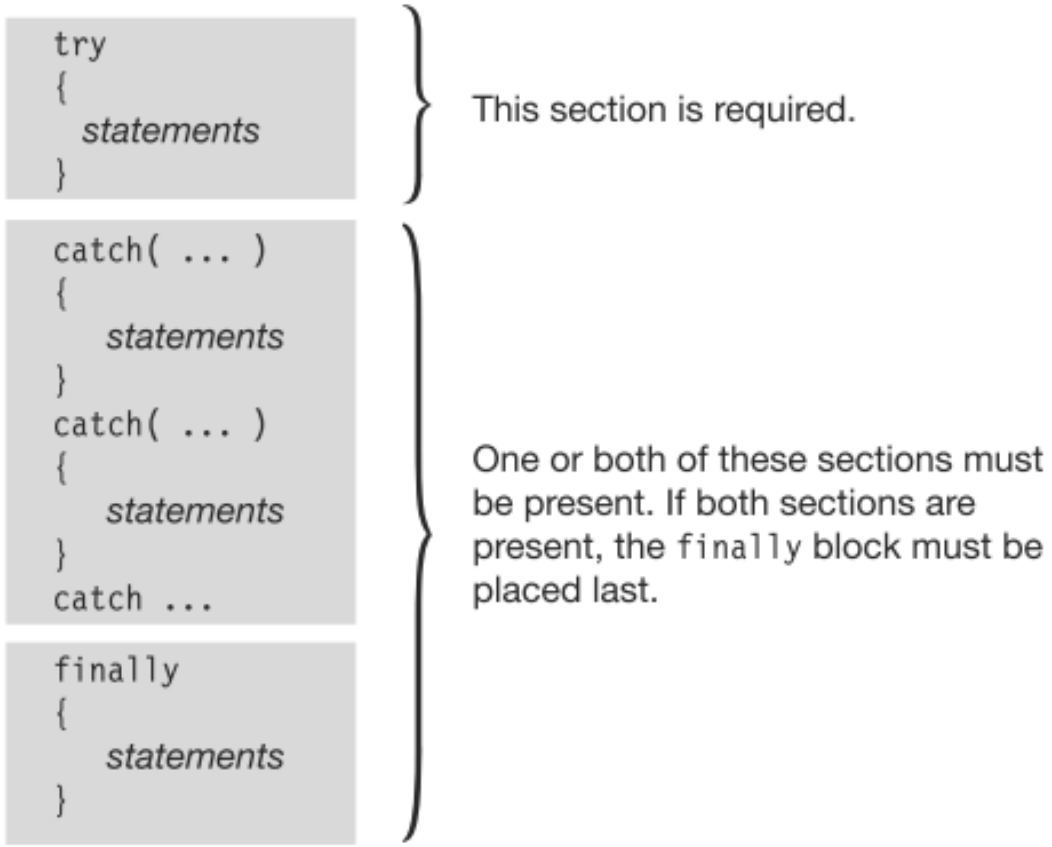
- Код, в якому може виникнути виняток (і яке ми хочемо обробити), треба помістити в try блок
- Код обробки треба помістити в блок catch
- Ми можемо в блоці catch вказувати конкретний клас винятку, який буде "уловлюватися" цим блоком
- Для одного try-блоку можна визначити декілька catch-блоків
- Код, який буде завжди виконуватися після try або try…catch-блоків треба помістити в finally-блок

Обробка виняткових ситуацій в C#

`try` block: This block contains the statements being guarded for exceptions.

`catch` clauses: This section contains the exception handlers for exceptions thrown in the `try` block.

`finally` block: This block contains code to be executed whether or not an exception is thrown in the `try` block.



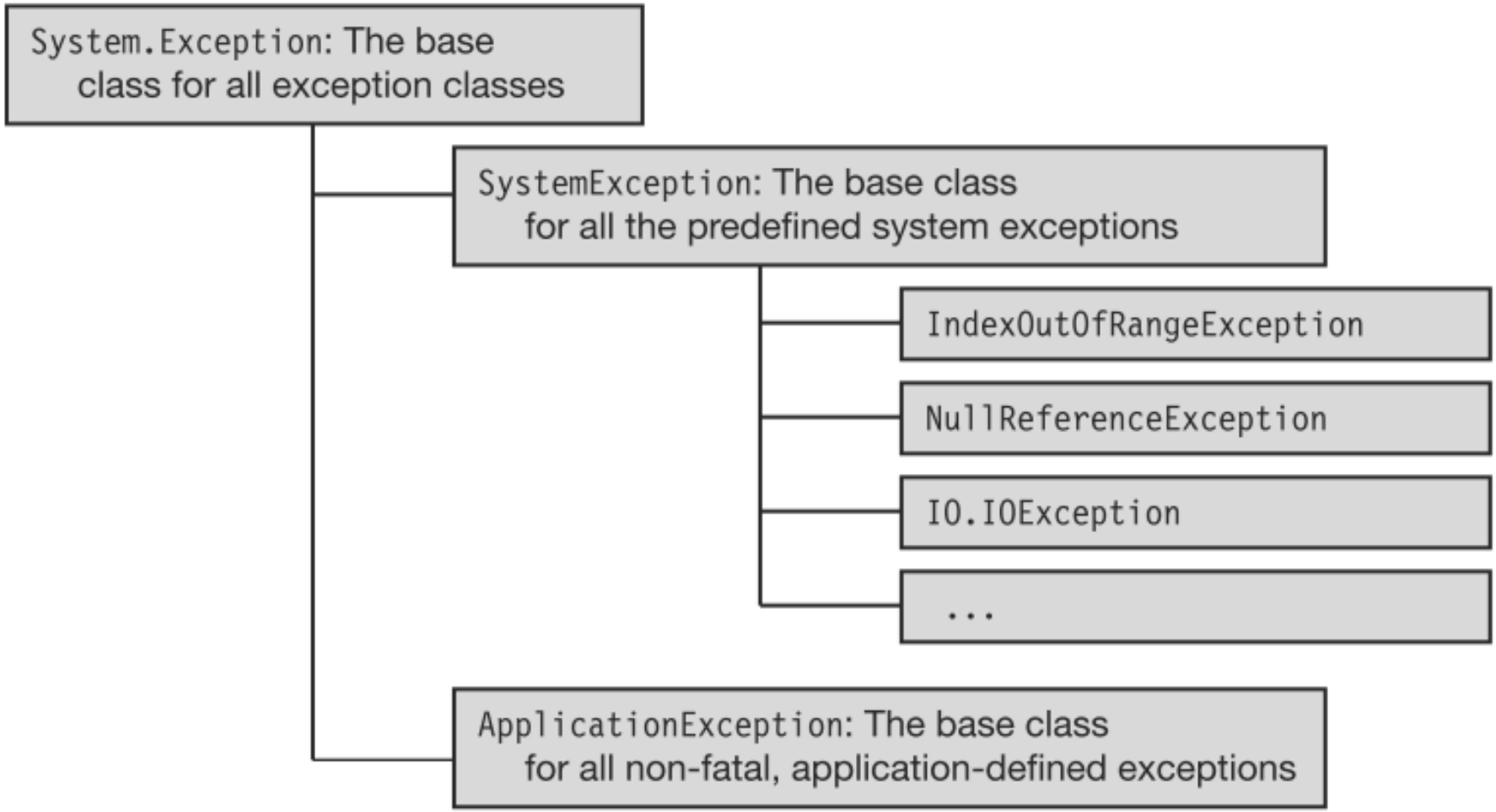
Приклад 1

```
try {
    int divisor =
    Convert.ToInt32(Console.ReadLine());
    int result = 3/divisor;
}
catch (DivideByZeroException ex)
{
    Console.WriteLine(ex.Message);
}
```

Приклад 2

```
try {
    int divisor = Convert.ToInt32(Console.ReadLine());
    int result = 3/divisor;
}
catch (DivideByZeroException)
{
    Console.WriteLine(“Attempted to divide by zero”);
}
catch (FormatException)
{
    Console.WriteLine(“Input was not in the correct format”);
}
catch (Exception)
{
    Console.WriteLine(“General catch handler”);
}
```

Ієрархія класів винятків



Деякі стандартні класи винятків

Ім'я	Опис
ArithmeticException	Помилка в арифметичних операціях
DivideByZeroException	Спроба ділення на нуль
FormatException	Спроба передати в метод аргумент невірного типу
IndexOutOfRangeException	Індекс масиву вийшов за межі діапазону
InvalidCastException	Помилка приведення типу
OverFlowException	Переповнення при виконанні арифметичної операції
StackOverFlowException	Переповнення стеку

Синтаксис обробників винятків

Три форми запису:

- **catch(тип ім'я) {. . . /*Тіло обробника */ }**
- **catch(тип) {. . . /*Тіло обробника */ }**
- **catch {. . . /*Тіло обробника */ }**
- Перша форма застосовується, коли ім'я параметра використовується в тілі обробника (наприклад для виведення інформації про виключення)
- Друга форма – важливий тільки тип
- Третя форма застосовується при перехопленні всіх винятків

Оголошення свого власного винятку (Define Your Own Custom Exception)

Ієрархія винятків в .NET Framework. Є два типи винятків:

- **Системні винятки**. Вони генеруються runtime (середовищем CLR). Їм відповідають класи, що спадкують від SystemException.
- **Прикладні винятки** (Application Exception). Вони генеруються програмою (user's program). Класи, що спадкують від ApplicationException.

Приклад

```
class InvalidArgumentException: ApplicationException
{
   public InvalidArgumentException(): base("Divide By Zero Error")
   { }
   public InvalidArgumentException(string message):
   base(message){ }
}
```

Приклад (закінчення)

```
class Program{
    static double Divide(double a, double b){
        if(b == 0)
            throw new InvalidArgumentException();
        double c = a/b;
        return c;
    }
    static void Main(){
        try{
            Console.WriteLine("In try block …");
            double d = Divide(3, 0);
            Console.WriteLine("\t Result of division: {0}", d);
        } catch (InvalidArgumentException e){
        Console.WriteLine("\t Message: ", e.Message);
}}
```

Застосунки з графічним інтерфейсом (GUI)

Для цього є два програмних інтерфейси:

- **(Старий) Інтерфейс Windows Forms**
  - Зборка System.Windows.Forms.dll
  - графічна система GDI+ (Зборка System.Drawing.dll)
- **(Новий) Інтерфейс Windows Presentation Foundation (WPF)**

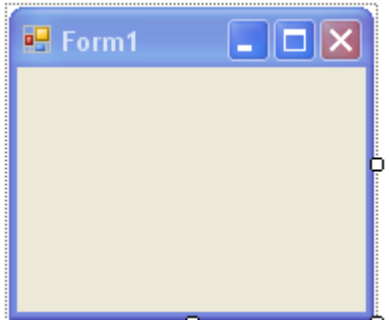
Створення застосунків Windows Forms

Приклад (Пуста форма без заголовка). Набираємо в текстовому редакторі і збираємо за допомогою компілятора командного рядка.

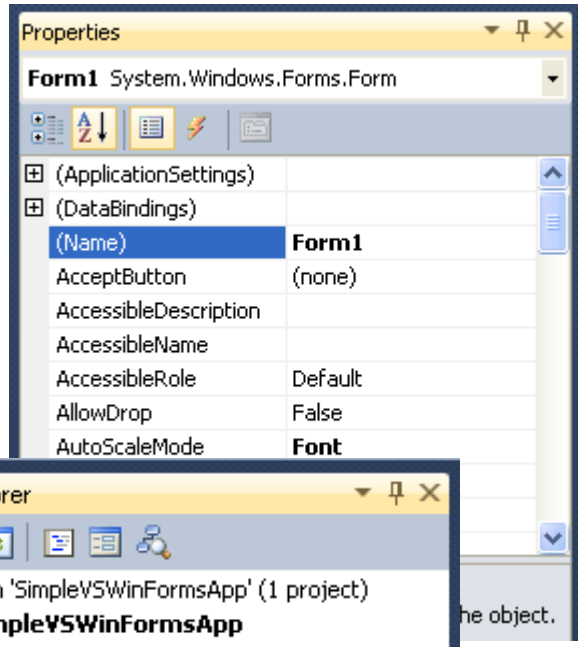
```
using System;
using System.Windows.Forms;
namespace NotepadForms
{
   public class MyForm: System.Windows.Forms.Form
   {
         public void MyForm()
         { }
            [STAThread]
            static void Main(){
                  Application.Run(new MyForm());
}}}
```

Шаблон проекту Windows Forms Application в Visual Studio

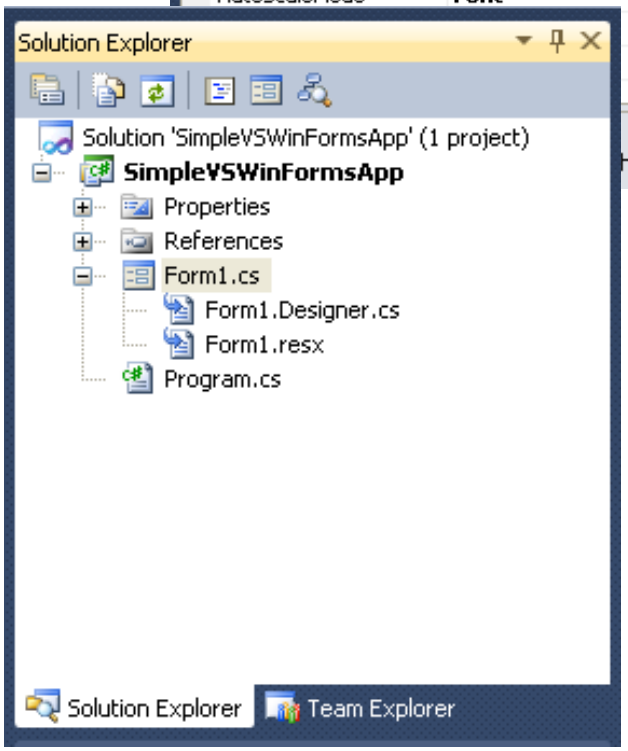
Візуальний дизайнер форм
(властивість Text форми
бажано змінити)



Вікно Properties



Вікно Solution Explorer
(Корисно виділити Form1.cs
та перейменувати в MainWindow.cs)

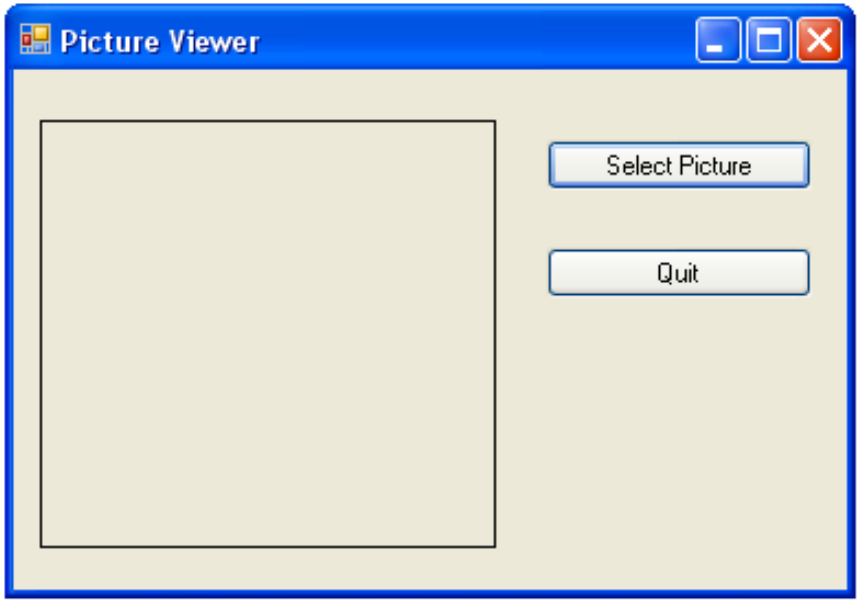


Приклад “Переглядач картинок”

Проект PictureViewer.

- **Form1.cs** перейменувати в **ViewerForm.cs**
- Властивість **Text** форми (“Form1”) замінити на “**Picture Viewer**”

Створення інтерфейсу



- PictureBox control
  - **Name**: **picShowPicture**
  - **BorderStyle**: **FixedSingle**
- Button control
  - **Name**: **btnSelectPicture**
  - **Text**: **Select Picture**
- Button control
  - **Name**: **btnQuit**
  - **Text**: **Quit**
- OpenFileDialog control (прихований)
  - **Name**: **ofdSelectPicture**
  - **Filename**: пусто (зробити пустим)
  - **Filter**: **Windows Bitmaps | *.bmp | JPEG Files | *.jpg**
  - **Title**: **Select Picture**

Приклад "Переглядач картинок"

Написання коду. Подвійне натискання на Select Picture

```
private void btnSelectPicture_Click(object sender, EventArgs e){
    if (ofdSelectPicture.ShowDialog() == DialogResult.OK){
        picShowPicture.Image =
  Image.FromFile(ofdSelectPicture.FileName);
        this.Text = string.Concat( "Picture Viewer(" +
  ofdSelectPicture.FileName + ")" );
    }
  }
```

Подвійне натискання на Quit

```
private void btnQuit_Click(object sender, EventArgs e){
        this.Close();
    }
```

Візуальна побудова системи меню

(Троельсен Язык пр-я C# 2008 и платформа .NET 3.5, стр.979)

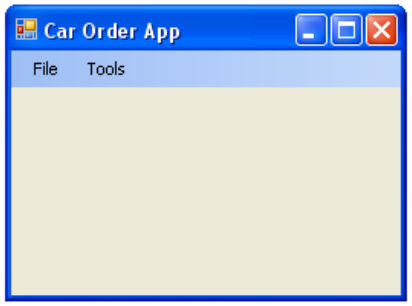
- **Перетягнути на форму елемент управління MenuStrip**
  - Visual Studio активізує редактор меню
- Натиснути маленьку трикутну піктограму справа
  - Відкриється контекстно-залежний вбудований редактор, що дозволяє виконувати декілька дій за раз
- Для прикладу вибрати опцію Insert Standard Items
  - Вбудується система меню (File, Edit, Tools, Help)
  - Відмінемо операцію (Ctrl-Z)
- **В дизайнері меню введіть пункт меню File верхнього рівня, а під ним – підменю Exit**
- Далі тиснемо кнопку з блискавкою в вікні Properties
  - Це покаже всі події, які можна обробити для вибраного елемента управління
- Вибираємо пункт меню Exit і робимо подвійне натискання на події Click. В обробнику пишемо Application.Exit();

Проектування діалогових вікон

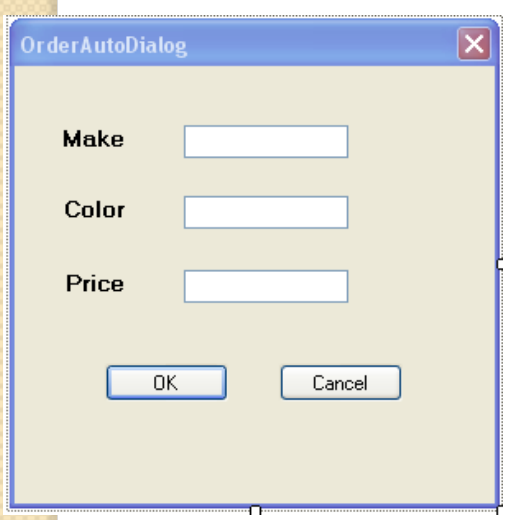
(Троельсен Язык пр-я C# 2008 и платформа .NET 3.5, стр.992)

- **В програмах з GUI діалогові вікна – це основний засіб вводу даних користувача**
- **В .NET нема базового класу “Dialog”, всі діалогові вікна в Windows Forms спадкують від класу Form**
- **Діалогове вікно конфігурується з постійним розміром**
  - Властивість **FormBorderStyle** дорівнює FormBorderStyle.FixedDialog
  - Властивості **MinimizeBox** та **MaximizeBox** – в **false**
- Крім того, якщо встановити ShowInTaskbar в false, то форма не буде показана в панелі задач Windows

Приклад. Проект CarOrderApp



- **Створити новий проект CarOrderApp**
  - Перейменувати Form1.cs в MainWindow.cs
- **Створити просте меню** File => Exit, Tools => Order Automobile
- **Створити обробники події Click для пунктів меню Exit та Order Automobile**
- **Вибрати в меню Project середовища Visual Studio пункт Add Windows Form**.
  - Назвіть нову форму OrderAutoDialog.cs
- **Сформувати наступний інтерфейс**



Поля вводу:
txtMake
txtColor
txtPrice

Кнопки:
btnOK
btnCancel

Властивості форми:
FormBorderStyle - **FixedDialog**
MinimizeBox - **false**
MaximizeBox - **false**
StartPosition - **CenterParent**
ShowInTaskbar - **false**

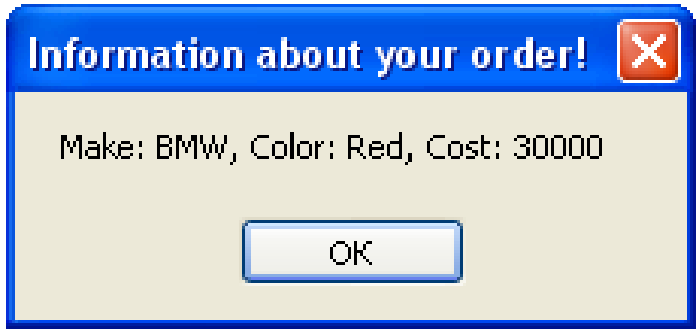
Приклад. Проект CarOrderApp

- Кнопка btnOK. Властивість DialogResult - OK
- Кнопка btnCancel. Властивість DialogResult - Cancel
- Виділити діалогову форму. **Встановити в формі кнопку вводу за замовчуванням** (<Enter>)
  - Властивість **AcceptButton** в **btnOK**
- **Відображення діалогового вікна**

```
private void
   orderAutomobileToolStripMenuItem_Click(object
   sender, EventArgs e) {
      OrderAutoDialog dlg = new OrderAutoDialog();
       if (dlg.ShowDialog() == DialogResult.OK){
```

Приклад. Проект CarOrderApp

```
//Отримати значення текстових полів (Буде помилка компіляції!!!)
   string orderInfo = string.Format("Make: {0}, Color: {1}, Cost: {2}", dlg.txtMake.Text, dlg.txtColor.Text, dlg.txtPrice.Text);
    MessageBox.Show(orderInfo, "Information about your order!");
            }
}
```



Інтерфейси в C#

Два види спадкування в ООП:

- **Спадкування реалізації** (implementation inheritance)
  - Похідний тип адаптує реалізацію кожного метода базового класу
- **Спадкування інтерфейсу** (interface inheritance) означає, що тип успадковує тільки сигнатуру методів, але не успадковує ніякої реалізації

Інтерфейси описують групу функціональних можливостей, які можуть належати довільному класу або структурі.

- **Інтерфейси можуть містити методи, властивості, події, індексатори.**
- **Інтерфейси не можуть містити поля.**
- **Члени інтерфейсу автоматично є відкритими.**

Інтерфейси в C#

(Подбельский, стор. 261)

- **Інтерфейс в C# (і, наприклад, в мові Java) – це механізм, призначений для визначення правил поведінки об'єктів іще не існуючих класів.**
- **Інтерфейс описує, які дії потрібні для об'єктів класу, але не визначає, як ці дії повинні виконуватись.**
- В оголошення інтерфейсу входять декларації (прототипи) методів, властивостей, індексаторів і подій
- Прототип метода не містить тіла, в ньому тільки заголовок методу.
- **На відміну від класів за допомогою інтерфейсів не можна визначати об'єкти.**
- На основі інтерфейсу оголошуються нові класи, і при цьому використовується механізм спадкування.
- **Говорять, що клас, побудований на базі інтерфейсу, реалізує даний інтерфейс.**

Інтерфейси в C#

Оголошення інтерфейсу

[модифікатор] interface Ім'я [: батьки] {//Тіло }

- Модифікатор може бути new, public, protected, internal, private
- Тіло містить декларації методів, властивостей, індексаторів, подій (всі за замовчуванням public)

Приклад (Подбельский)

```
using System;
interface IGeo {  //Інтерфейс геометричної фігури
   viod transform(double coef);
   void display();
}
class Circle: IGeo { //Круг
{
   double rad = 1;
   public void transform(double coef){ rad *= coef;}
```

Приклад (Подбельский)

```
  public void display(){
    Console.WriteLine("Площа круга: {0:G4}", Math.PI*rad*rad);
  }
} //End class Circle
class Cube: IGeo { //Куб
    double rib = 1; //ребро
    public void transform(double coef){ rib *=coef;}
    public void display(){
       Console.WriteLine("Об'єм куба: {0:G4}", rib*rib*rib);
    }
} //End class Cube
class Program{
    public static void report(IGeo g){
      Console.WriteLine("Дані об'єкта класу {0}:", g.GetType() );
       g.display();
    }
    public static void Main(){
```

Приклад (Подбельский)

```
public static void Main(){
    Circle cir = new Circle();
    report(cir);
    Cube cub = new Cube();
    report(cub);
    IGeo ira = cir;
    report(ira);
  }
}
```

Результат:

Дані об’єкта класу Circle:
Площа круга: 3.142
Дані об’єкта класу Cube:
Об’єм куба: 1
Дані об’єкта класу Circle:
Площа круга: 3.142

Дві стратегії реалізації інтерфейсу (Биллиг “Язык программирования C#”)

Клас, що реалізує інтерфейс, може реалізувати його методи двома способами:

- **public-методами** (як у попередньому прикладі)
- **private-методами, уточненими іменем інтерфейсу**

Розглянемо другий спосіб.

```
interface IProps{
    void Prop1(string s);
    void Prop2(string name, int val);
}
class ClainP: IProps{
    public ClainP(){ }
    void IProps.Prop1(string s){
            Console.WriteLine(s);}
    void IProps.Prop2(string name, int val){
            Console.WriteLine(“name={0}, val={1}”, name, val);}
. . .
}
```

Дві стратегії реалізації інтерфейсу

Як отримати доступ до закритих методів?

Відповідь: **Шляхом обгортання та кастингу**.

- **Обгортання**: створити public-метод, що є обгорткою закритого методу.
- **Кастинг**: приведення до інтерфейсного типу.

```
public void MyProp1(string s){ //Обгортка для Prop1()
    ((IProps)this).Prop1(s); // Кастинг
}
public void MyProp2(string s, int x){ //Обгортка для Prop2()
    ((IProps)this).Prop2(s, x); // Кастинг
}
```

Проблеми множинного спадкування

Дві основні проблеми:

- **Колізія імен**
- **Спадкування від спільного предка**

Колізія імен

Проблема колізії імен виникає, коли два або більше батьківських інтерфейсів мають методи з однаковими іменами та сигнатурою.

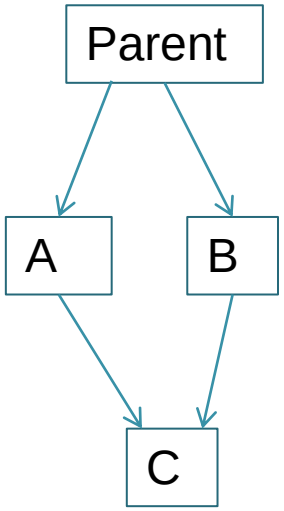
Дві стратегії "боротьби":

- **Склеювання**
  - Клас, що реалізує інтерфейси, вважає методи однаковими
- **Перейменування**
  - Методи різних інтерфейсів реалізують як закриті, а потім відкривають їх з перейменуванням.

Проблеми множинного спадкування

Спадкування від спільного предка

- Якщо клас C є спадкоємцем класів A і B, а ті, в свою чергу, є спадкоємцями класу Parent, то **клас спадкує властивості і методи свого предка Parent двічі**.
- **Це дублююче спадкування**



- Це складає проблему тільки при множинному спадкуванні класів.
- **Для інтерфейсів проблема дублюючого спадкування зводиться до проблеми колізії імен.**

Деякі вбудовані інтерфейси

1) **Впорядкованість об'єктів і інтерфейс IComparable**

- **Часто, коли створюється клас, бажано задати відношення порядку на його об'єктах**
- Такий клас слід оголосити спадкоємцем інтерфейсу IComparable
- Цей інтерфейс має усього один метод **int CompareTo(object obj)**, який повертає **+**, **-** або **0**, в залежності від виконання відношень "**більше**", "**менше**" або "**дорівнює**"
- Як правило, в класі спочатку визначають метод CompareTo, а після цього вводять перевантажені операції, щоб виконати порівняння об'єктів.

Приклад (Биллиг). Відношення порядка на прізвищах персон

```
class Person: IComparable
{
    public string fam;
    . . .
    public int CompareTo(object pers){
        const string s = "Об'єкт не є Person";
        Person p = pers as Person; //приведення до типу
        if(!p.Equals(null))
            return (fam.CompareTo(p.fam));
        throw new ArgumentException(s);
    }
    public static bool operator < (Person p1, Person p2){
        return (p1.CompareTo(p2) < 0);
    }
. . .
}
```

Операції is та as

- Вживаються при приведенні типів
- **Логічний вираз (obj is T) істинний, якщо об'єкт obj має тип T**
- **Оператор присвоювання (obj = P as T;) присвоює об'єкту obj об'єкт P, приведений до типу T, якщо таке приведення можливе, інакше об'єкту присвоюється значення null.**

2) Клонування об’єктів і інтерфейс IClonable

- Клонуванням називають процес створення копії об’єкта.
- Розрізняють два типи клонування: **поверхневе** (shallow) та **глибоке** (deep).
- **При поверхневому клонуванні копіюється сам об’єкт.**
  - **Усі значущі поля** клону отримують значення, що співпадають із значеннями полів об’єкта
  - **Усі посилальні поля** клону є посиланнями на ті ж об’єкти, на які посилається і сам об’єкт
- **При глибокому клонуванні копіюється вся сукупність об’єктів, що пов’язані взаємними посиланнями.**
  - Глибоке клонування потребує рекурсивної процедури обходу існуючої структури об’єктів

2) Клонування об’єктів і інтерфейс IClonable

- **Для поверхневого клонування достатньо скористатися методом MemberwiseClone**, який успадковано ще від прабатька object.
  - Слід пам’ятати, що **це метод захищений** (protected), тому його не можна викликати у клієнта
  - Тому клонування слід виконувати у початковому класі, зробивши “**обгортку**”

```
public Person StandartClone()
{
   Person p = (Person)this.MemberwiseClone();
   return (p);
}
```

Якщо стандартне клонування вас не влаштовує, то клас можна оголосити спадкоємцем інтерфейсу IClonable і реалізувати метод Clone.

3) Серіалізація об'єктів. Інтерфейс ISerializable

- **Під серіалізацією розуміють процес збереження об'єктів в файлах в період виконання програми**.
- **Під десеріалізацією розуміють зворотній процес**.
- Механізми серіалізації C# та .NET Framework підтримують два формати збереження даних – в бінарному файлі або в текстовому XML-файлі.
- **Серіалізація (як клонування) може бути поверхневою та глибокою.**
- **Якщо клас оголосити з атрибутом [Serializable], то в нього вбудується стандартний механізм серіалізації, який підтримує глибоку серіалізацію.**
- Якщо це не влаштовує, то клас можна оголосити спадкоємцем інтерфейсу ISerializable.

Клас з атрибутом серіалізації

Порядок такий:

- **[Serializable]**
  class MyClass
  { . . .
  }
- **Необхідно створити об'єкт, званий форматером**, який і виконує серіалізацію (метод Serialize()) і десеріалізацію.
  - **BinaryFormatter** з простору імен System.Runtime.Serialization.Formatters.Binary
  - **SoapFormatter**
- **Нам знадобиться файл та класи, які підтримують введення-виведення** (з System.IO) – наприклад FileStream

Приклад по казці Пушкіна про рибалку та рибку (Биллиг).

```
[Serializable]
class Personage{
    static int wishes; //бажання
    public string name, status, wealth; //ім'я, статус, майно
    int age;
    public Personage couple; //дружина, чоловік
    public Personage(string name, int age){
        this.name = name;
        this.age = age;
    }
. . .
}
```

Приклад (продовження)

Додамо метод “Одружитися”

```
public void marry( Personage couple)
{
        this.couple = couple;
        couple.couple = this;
        this.status = “селянин”;
        this.wealth = “рибацька сіть”;
        this.couple.status = “селянка”;
        this.couple.wealth = “корито”;
        SaveState();
}
```

Приклад (продовження)

Метод збереження стану (серіалізація).

```
void SaveState()
{
    BinaryFormatter bf = new BinaryFormatter();
    FileStream fs = new FileStream("State.bin",
            FileMode.Create, FileAccess.Write);
    bf.Serialize(fs, this);
    fs.Close();
}
```

Приклад (продовження)

Метод, що описує життя героїв казки.

```
public Personage AskGoldFish()
{
    Personage fisher = this;
    if (fisher.name == “рибак”) {
            wishes++;
            switch (wishes)
            {
                case 1: ChangeStateOne(); break;
                case 2: ChangeStateTwo(); break;
                case 3: ChangeStateThree(); break;
                default: BackState(ref fisher); break;
            }
    }
 return (fisher);
}
```

Приклад (продовження)

Починаючи з четвертого бажання все повертається в початковий стан – виконується десеріалізація графа об'єктів.

```
void BackState(ref Personage fisher)
{
   BinaryFormatter bf = new BinaryFormatter();
   FileStream fs = new FileStream("State.bin",
                  FileMode.Open, FileAccess.Read);
       fisher = (Personage)bf.Deserialize(fs);
       fs.Close();
   }
```

Приклад (закінчення)

```
class Program{
    public void TestGoldFish(){
        Personage fisher = new Personage("рибак", 70);
        Personage wife = new Personage("старуха", 70);
        fisher.marry(wife);
        Console.WriteLine("До золотої рибки"); fisher.About();
        fisher = fisher.AskGoldFish();
        Console.WriteLine("Перше бажання"); fisher.About();
        . . . . . .
    }
    static void Main(){
        TestGoldFish();
    }
}
```

Перевантаження операцій в C# (Operator Overloading)

- **Операції можуть бути перевантажені для користувацьких типів (custom types) для впровадження більш природнього синтаксису**.

```
public struct Note
{
  int value;
  public Note (int x) { value = x; }
  public static Note operator + (Note x, int s)
  {
    return new Note (x.value + s);
  }
}
```

Це перевантаження дозволяє додати int до Note

```
Note B = new Note (2);
Note CSharp = B + 2;
```

Які операції можна перевантажити

- **Всі бінарні операції** можна перевантажити (+, -, *, /, %, &, |, ^, <<, >>)
- **Всі унарні операції** можна перевантажити (+, -, !, ~, ++, --, true, false)
- **Всі операції відношення** можна перевантажити (==, !=, <, >, <=, >=), але тільки парами
- **&&, ||, (тип), =, ., ?:, ->, new, is, as, sizeof - не перевантажуються**
- Приведення до типу перевантажується з вживанням ключових слів implicit та explicit

Загальний синтаксис (перевантаження операції op)

public static retval operator op (obj1 [, obj2]) {…}

Перевантаження операцій будується на основі відкритих (public) статичних (static) методів з використанням ключового слова operator.

```
class Complex{
   private double Re, Im;
   public static Complex operator – (Complex c){
         Complex temp = new Complex();
         temp.Re = -c.Re; temp.Im = -c.Im;
         return temp;
   }
   public static Complex operator – (Complex c1, Complex c2){
         Complex temp = new Complex();
         temp.Re = c1.Re – c2.Re;
         temp.Im = c1.Im – c2.Im;
         return temp; }
. . .
```

Використання: obj2 = -obj1; . . . obj3 = obj1 – obj2;

Перевантаження операції приведення до типу

- Іноді об’єкт певного класу може вживатися в виразах, що включають дані інших типів.
  - І бажано мати можливість приведення типа класу в цільовий тип
- Існують дві форми операції приведення типу : явна і неявна.

```
public static explicit operator Цільовий_тип
(Початковий_тип v){return Значення_цільового_типу;}
```

```
public static implicit operator Цільовий_тип
(Початковий_тип v){return Значення_цільового_типу;}
```

- Якщо операція приведення до типу задана в неявній формі (implicit), то приведення викликається автоматично, якщо об’єкт зустрічається в виразі разом з значенням цільового типу.
- Явна форма – при явному приведенні: (тип)
- **Одночасно явну і неявну форми задавати не можна!**

Приклад

```
using System;
class Point3D
{
    public int x, y, z;
    public Point3D(){ }
    public Point3D(int _x, int _y, int _z){
        x=_x; y=_y; z=_z;
    }
    public static implicit operator Point3D(Point2D p2d){
        Point3D p3d = new Point3D();
        p3d.x = p2d.x;
        p3d.y = p2d.y;
        p3d.z = 0;
        return p3d;
    }
}
```

Приклад (продовження)

```
class Point2D{
    public int x, y;
    public Point2D(){ }
     public Point2D(int _x, int _y){
            x=_x; y=_y;
    }
    public static explicit operator Point2D(Point3D p3d){
            Point2D p2d = new Point2D();
            p2d.x = p3d.x;
            p2d.y = p3d.y;
            return p2d;
    }
}
```

Приклад (закінчення)

```
class Program
{
    static void Main()
    {
        Point2D p2d = new Point2D(125, 125);
        Point3D p3d;
        p3d = p2d; // Спрацьовує неявне приведення
        p3d.x = p3d.x*2;
        p3d.y = p3d.y*2;
        p3d.z = 125;
        p2d = (Point2D)p3d;
    }
}
```

Функціональний тип в C#. Делегати (delegate)

Вказівник на функцію

- **Мова C++ підтримує вказівники на функцію (function pointer), які можна передавати в якості параметрів функцій**.
- Іноді це називають зворотнім викликом (callback).
- **Наприклад, функція qsort** з стандартної бібліотеки C++ **отримує вказівник на функцію-компаратор** (comparator), яку використовують для порівняння елементів.
- **Передаючи вказівники на різні функції-компаратори, можна отримувати різний порядок сортування**.
- **В мові Java вказівники на функцію відсутні. Їх можна замінити посиланнями на об'єкти з одним методом**.
- **В мові C# для підтримки вказівників на функцію використовують делегати.**

Функціональний тип в C#. Делегати (delegate)

- Нагадаємо, що в якості типів в мові C# виступають: клас, структура, перерахування (enum), інтерфейс і делегат.
- **Стандарт мови C# виділяє три етапи застосування делегатів:**
  - **Визначення (оголошення) делегата як типу**
  - **Створення екземпляра делегата**
  - **Звертання до екземпляра делегата (виклик делегата)**
- Кожен делегат є спадкоємцем класу System.Delegate
- Синтаксис визначення делегата:

[мод-ри] delegate return_type Type_name(<параметри>)

- Приклади:

```
public delegate int[ ] Row (int num);
public delegate void Print (int[] ar);
```

Делегати

- **Кожен делегат описує множину функцій (методів) з заданою сигнатурою (визначає контракт)**
- **Делегата можна оголошувати**:
  - В просторі імен поряд з іншими оголошеннями класів і структур
  - Всередині оголошень класів і структур

- **Приклад**. Оголошення, створення екземплярів, налаштування на конкретні методи, виклик методів

```
delegate void MyDel(int x);    // Declare delegate type.
MyDel delVar, dVar;     // Create two delegate variables.
```

Instance method

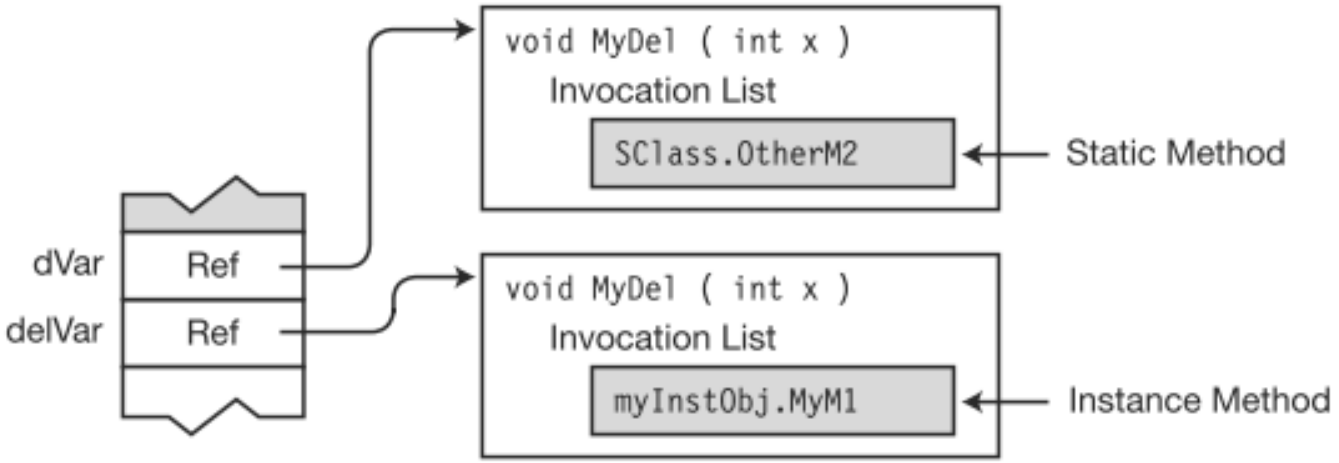
↓

```
delVar = new MyDel( myInstObj.MyM1);
dVar   = new MyDel( SClass.OtherM2 );
```

↑

Static method

Делегати



Можна використовувати спрощений синтаксис:

```
MyDel delVar = myInstObj.MyM1;
MyDel dVar   = SClass.OtherM2;
```

Виклик методів через делегат:

```
delVar(<параметри>)
dVar(<параметри>)
```

Приклад (Подбельский, стор. 343)

```
using System;
public delegate int[] Row (int num); // Делегат-тип
public delegate void Print (int[] ar); // Делегат-тип
public class Example{
   //Метод повертає масив цифр числа
   static public int[] series(int num){
        int arLen = (int)Math.Log10(num)+1;
        int[] res = new int[arLen];
        for(int i=arLen-1; i>=0; i--){
                res[i] = num % 10;
                num /= num;
        }
        return res;
   }
```

Приклад (закінчення)

```
//Метод виводе на екран значення елементів масиву
    static public void display(int[] ar){
     for(int i = 0; i<ar.Length; i++)
            Console.Write("{0}\t", ar[i]);
            Console.WriteLine();
    }
} //End of Example
class Program{
    static void Main(){
            Row delRow; // Посилання на делегат
            Print delPrint; // Посилання на делегат
            delRow = new Row(Example.series);
            delPrint = new Print(Example.display);
            int[] myAr = delRow(13579);
            delPrint(myArr); } }
```

Делегати в якості параметрів методів

Зворотні виклики

Особливість callback-методу в тому, що ви передаєте його в якості аргумента іншому методу, а не викликаєте його безпосередньо.

```
using System;
delegate int CallBackMethod(int arg1, int arg2);
class TestApp{
    public static int MyCallBack(int a, int b){
        Console.WriteLine("a={0}, b={1}", a, b);
        return 0;}
    public static void SimpleMethod(CallBackMethod cbm){
        cbm(10, 15);}
    public static void Main(){
        CallBackMethod m = new CallBackMethod(MyCallBack);
        SimpleMethod(m); }}
```

Зворотні виклики

- В попередньому прикладі метод SimpleMethod здійснює зворотній виклик через делегата cbm, який передається як аргумент

- (Троельсен, стор.386) **Багато застосунків потребують, щоб об'єкт міг звертатися зворотньо до сутності, яка його створила – через механізм зворотніх викликів.**

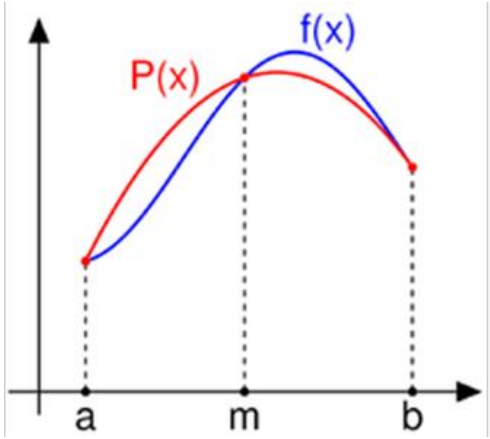
- Хоча механізми зворотнього виклику можуть застосовуватися в довільному застосунку, **вони особливо важливі в графічних інтерфейсах користувача**, де елементи управління (такі як кнопки) потребують викликів зовнішніх методів (при натисканні).

Функція вищого порядку

- Визначення (з Вікіпедії). Функція вищого порядку – це функція, що приймає в якості аргументів інші функції або повертає іншу функцію в якості результата.
- Про це є в лекціях Биллига.
- В попередньому прикладі метод SimpleMethod – функція вищого порядку

Метод Сімпсона

- **Метод Сімпсона** є одним із методів чисельного інтегрування. Названий на честь британського математика Томаса Сімпсона (1710—1761).



Формулою Сімпсона називається інтеграл від інтерполяційного многочлена другого степеня на відрізку $[a, b]$:

$$\int_a^b f(x)dx \approx \int_a^b p_2(x)dx = \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right),$$

де $f(a)$, $f((a+b)/2)$ і $f(b)$ — значення функції у відповідних точках .

Ітераційна формула

Для точнішого обчислення інтеграла проміжок [a,b] розбивають на N відрізків однакової довжини і застосовують формулу Сімпсона на кожному з них. Значення інтеграла є сумою для всіх відрізків:

$$\int_a^b f(x)dx \approx \frac{h}{3} \cdot \left(\frac{1}{2} f(x_0) + \sum_{k=1}^{N-1} f(x_k) + 2 \sum_{k=1}^{N} f\left(\frac{x_{k-1} + x_k}{2} \right) + \frac{1}{2} f(x_N) \right)$$

де $h = \frac{b-a}{N}$ величина кроку, а $x_k = a + k \cdot h$ межі відрізків.

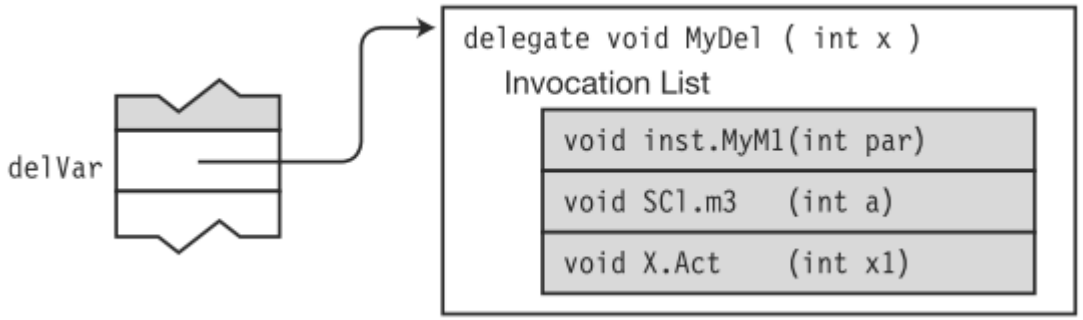
Програмна реалізація

```
class Integral
{
    public delegate double Function(double x);
    public static double integral(Function f, double a, double b,
                                  int n){
        double sum=0;
        double h=(b-a)/n;
        for(int i=0; i<n; i=i+2)
          sum=sum+(f(a+i*h)+4*f(a+(i+1)*h)+f(a+(i+2)*h))*h/3;
        return sum;
    }
}
```

Об'єднання делегатів

- **Екземпляр делегата може містити не одне посилання на метод, а список посилань на різні методи.**
- **Ви можете додати** (або видаляти) **метод за допомогою операції +=** (або -= для видалення)

```
MyDel delVar  = inst.MyM1; // Create and initialize.
delVar += SCl.m3; // Add a method.
delVar += X.Act;    // Add a method.
```



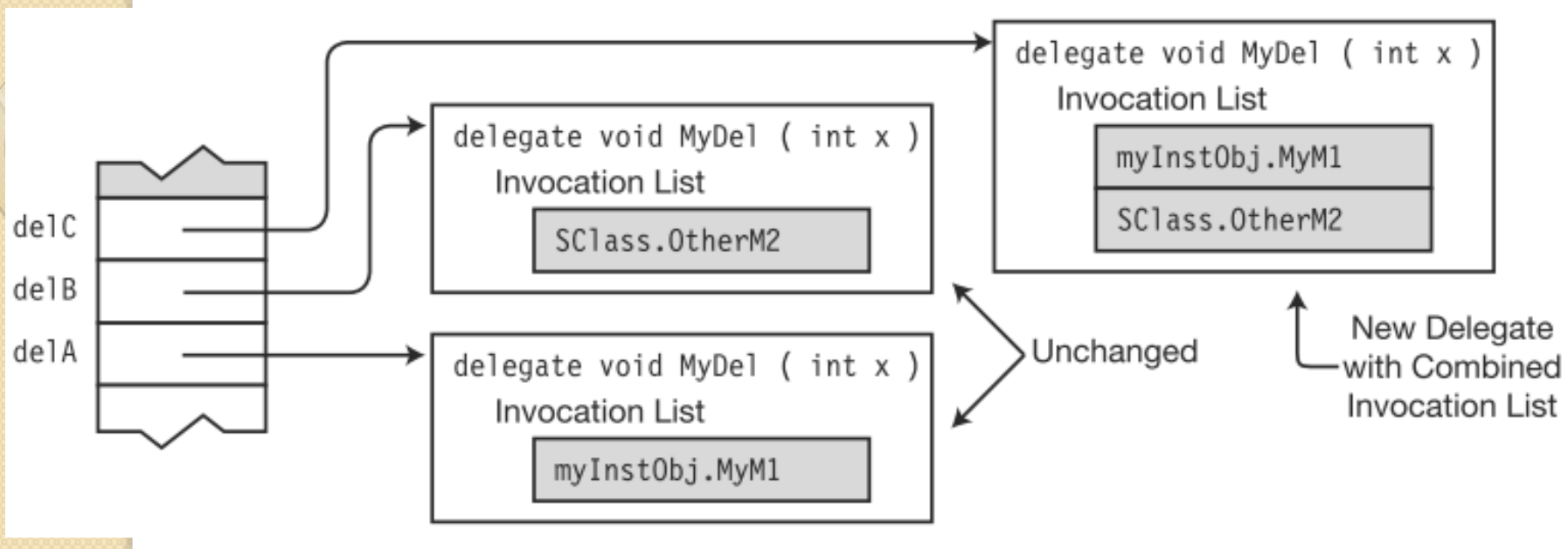
- Такі делегати називають Multicast Delegates (багатоадресні делегати)
- "За кулісами" перевантажених операцій += та -= "працюють" методи Combine та Remove класу System.Delegate

Об'єднання делегатів

- **В попередньому прикладі виклик delVar приведе до виклику методів inst.MyM1, SCl.m3 та X.Act**
- Можна видалити методи з списку: delVar -=SCl.m3;
- **Якщо return type делегата не є void, то список викликів поверне повернення останнього метода**.
- Також можна об’єднувати різні екземпляри делегата, наприклад

```
MyDel delA = myInstObj.MyM1;
MyDel delB = SClass.OtherM2;
MyDel delC = delA + delB;  // Has combined invocation list
```

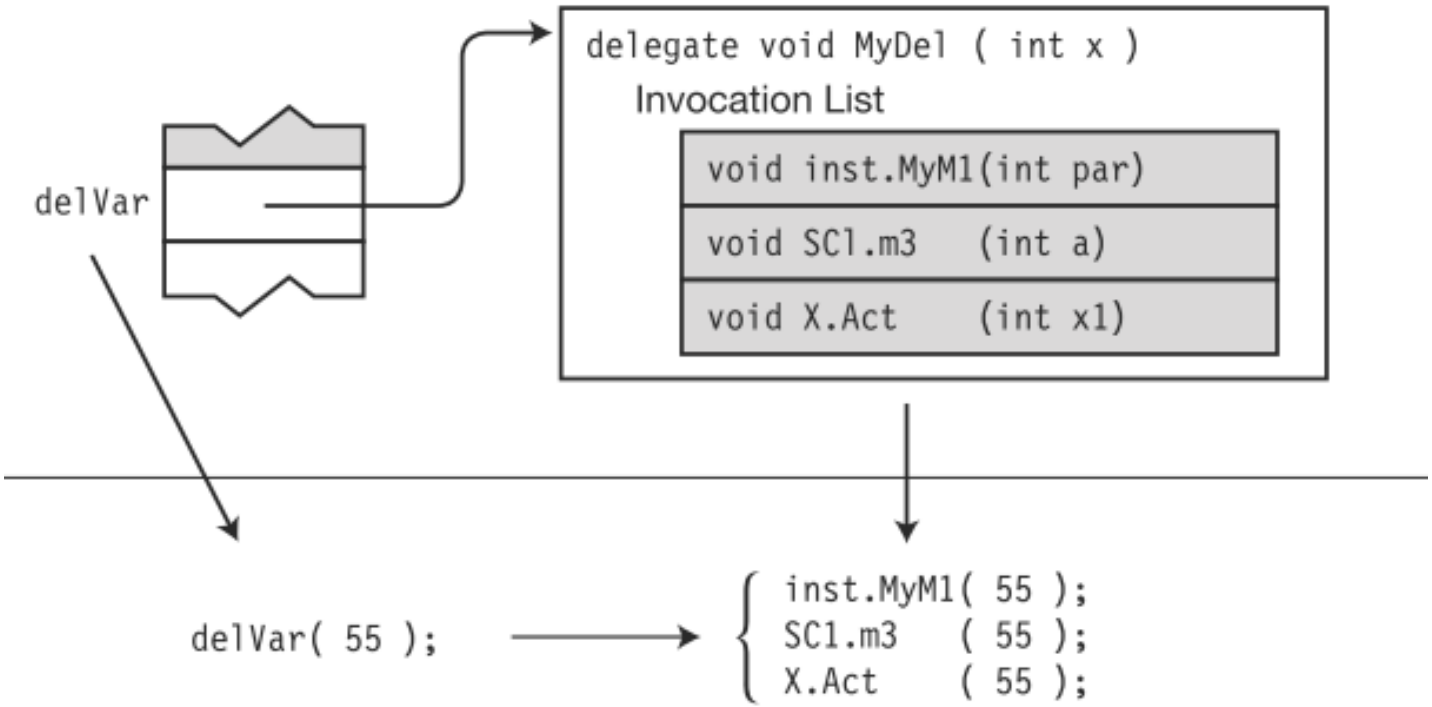
Об'єднання делегатів



- Виходячи з терміну “об'єднання делегатів” може скластися враження, що операнд делегатам можна змінювати.
- **Насправді, делегати є незмінними**.
- **Після того, як об'єкт делегата створений, він не може бути змінений.**

Invoking a Delegate

```
MyDel delVar = inst.MyM1;
delVar += SCl.m3;
delVar += X.Act;
    ...
delVar( 55 );   // Invoke the delegate.
    ...
```



Multicast delegate example

- З книги C# 5.0 in a Nutshell, p.122 (140)
- Припустимо, ви написали програму, яка зайняла багато часу для виконання.
- Ця програма може регулярно повідомляти про прогрес в роботі в результаті застосування делегата.
- У цьому прикладі процедура HardWork має параметром делегата ProgressReporter, який вона викликає, щоб вказати прогрес

Multicast delegate example (1)

```
public delegate void ProgressReporter (int percentComplete);
public class Util
{
  public static void HardWork (ProgressReporter p)
  {
    for (int i = 0; i < 10; i++)
    {
      p (i * 10);                    // Invoke delegate
      System.Threading.Thread.Sleep (100);  // Simulate hard work
    }
  }
}
```

Multicast delegate example (2)

```
class Test
{
  static void Main()
  {
    ProgressReporter  p = WriteProgressToConsole;
    p += WriteProgressToFile;
    Util.HardWork (p);
  }

  static void WriteProgressToConsole (int percentComplete)
  {
    Console.WriteLine (percentComplete);
  }

  static void WriteProgressToFile (int percentComplete)
  {
    System.IO.File.WriteAllText ("progress.txt",
                                 percentComplete.ToString());
  }
}
```

Екземпляр делегата – це вказівник на функцію?

- Ні.
- **Фактично екземпляр делегата p містить два вказівника: на об'єкт (p.Target) і на метод (p.Method)**

```
MyDel delVar  = inst.MyM1;
```

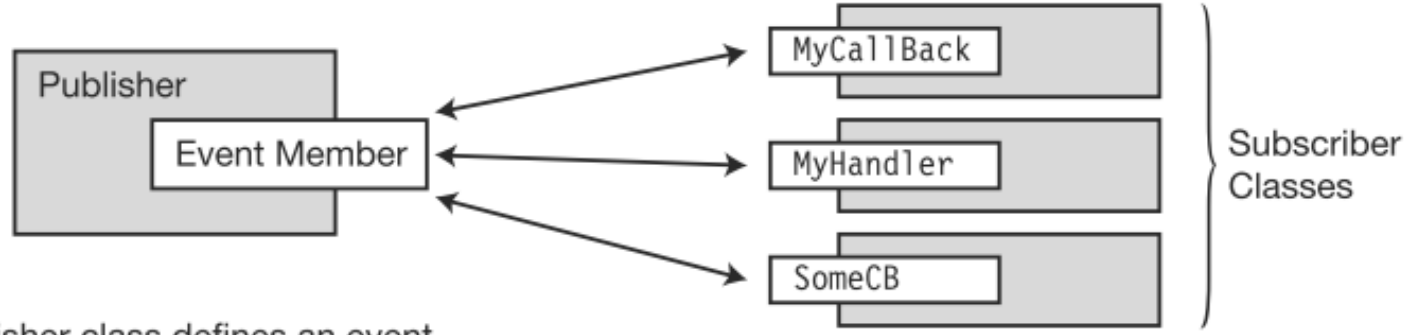
- Більш точно, екземпляр делегата містить список викликів (Invocation List) на різні методи

Події (events) в C#

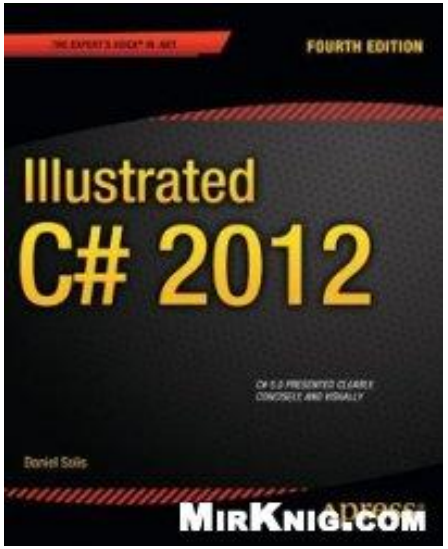
- На основі делегатів функціонує важливий механізм .NET – події (events).
- **Події можуть використовуватись для того, щоб інформувати інші класи про зміну стану даного класу.**
- Оператор посилки повідомлення виглядає так:
  ім'я_події(аргументи_для_делегата);
- **Опис події складається зі слова event, назви події та назви класу делегата, функції якого оброблятимуть дану подію.**

  [модифікатор] event ім'я_делегата ім'я_події;
- В C# використовується **модель "Публікація/Підписка"** (Publisher/Subscriber) – клас публікує ті події, які він може ініціювати, а інші класи можуть підписатися на отримання оповіщень про них.

Події (events) в C#



1. The publisher class defines an event member.
2. Subscriber classes register callback methods (handlers) to be invoked when the event is raised.
3. When the publisher raises the event, all the handlers in its list are invoked.



Illustrated C# 2012, p.345 (348)

Клас Publisher

Повинен виконати наступне:

- **Оголосити делегата**, який визначає сигнатуру обробників події.
- **Оголосити подію**: event делегат_події ім'я_події;
  - ім'я_події інколи називають **event-об'єктом**
- **Створити метод (On-процедуру),** в якому **ініціювати подію**:
  - ім'я_події(аргументи_для_делегата);
- В потрібних методах класу **викликати On-процедуру**

Клас Subscriber

Повинен виконати наступне:

- **Мати обробник події** – метод, узгоджений по сигнатурі з делегатом події
- **Мати посилання на об'єкт, який створює подію**, щоб отримати доступ до event-об'єкту
- **Зуміти приєднати обробник події до event-об'єкту** (зробити підписку)

За загальноприйнятим стилем, **делегат події приймає два аргументи**: **об'єкт відправник події** (sender), та **об'єкт типу** EventArgs (або спадкоємець від нього), що надає додаткову інформацію про подію.

Приклад (1)

Клас Metronome генерує подію (тіки таймера) кожні 3 секунди.
Клас Listener слухає тіки метронома і друкує “HEARD IT”.

```
using System;
namespace MyApp{
class Metronome{
   public event TickHandler tick; // event-об’єкт
   public EventArgs e = null;
public delegate void TickHandler(Metronome m, EventArgs e);
   public void Start(){ // On-процедура
        while (true){
        System.Threading.Thread.Sleep(3000);
        if (tick != null){
                tick(this, e); }}}} // Ініціація події
```

Приклад (2)

```
class Listener{
    public void Subscribe (Metronome m){
        m.tick += new Metronome.TickHandler(HeardIt); //підписка
    }
    private void HeardIt (Metronome m, EventArgs e){
        Console.WriteLine("HEARD IT");
    }
}
class Test{
    static void Main(){
        Metronome m = new Metronome();
        Listener l = new Listener();
        l.Subscribe(m);
        m.Start();
    }
}}
```

Ускладнений приклад (1)

- Нехай подія несе ще деяку додаткову інформацію (момент часу виникнення події).
- Для цього перевизначимо клас EventArgs. Вставимо після namespace:

```
public class TimeOfTick: EventArgs{
    private DateTime timeNow;
    public DateTime Time {
        set {timeNow = value;}
        get {return this.timeNow;}
    }
}
```

Ускладнений приклад (2)

```
class Metronome{
   public event TickHandler tick; // event-об'єкт
public delegate void TickHandler(Metronome m, TimeOfTick e);
   public void Start(){ // On-процедура
       while (true){
          System.Threading.Thread.Sleep(3000);
          if (tick != null){
                TimeOfTick tot = new TimeOfTick();
                tot.Time = DateTime.Now;
                tick(this, tot);
          }
       }
   }
}
```

Ускладнений приклад (3)

В класі Listener трохи змінимо метод HeardIt:

```
private void HeardIt (Metronome m, TimeOfTick e){
    Console.WriteLine("HEARD IT AT {0}", e.Time);
    }
```

Клас Test залишається без змін.

Узагальнення (generics) в C#

- **Шилдт Г. C# 4.0 Полное руководство**, гл.18 стр.575
- **Illustrated C# 2012**, p.415 (420)

- Узагальнення з'явились в C# 2.0
- **Узагальнення дозволяють створювати параметризовані класи (а також структури, інтерфейси, методи та делегати), в які тип передається як параметр**.

```
class Point<T>      // T – параметр типу
{
  public T x, y;
}
```

- При створенні об'єктів вказуємо конкретний тип

```
Point<short> p = new Point<short>();
```

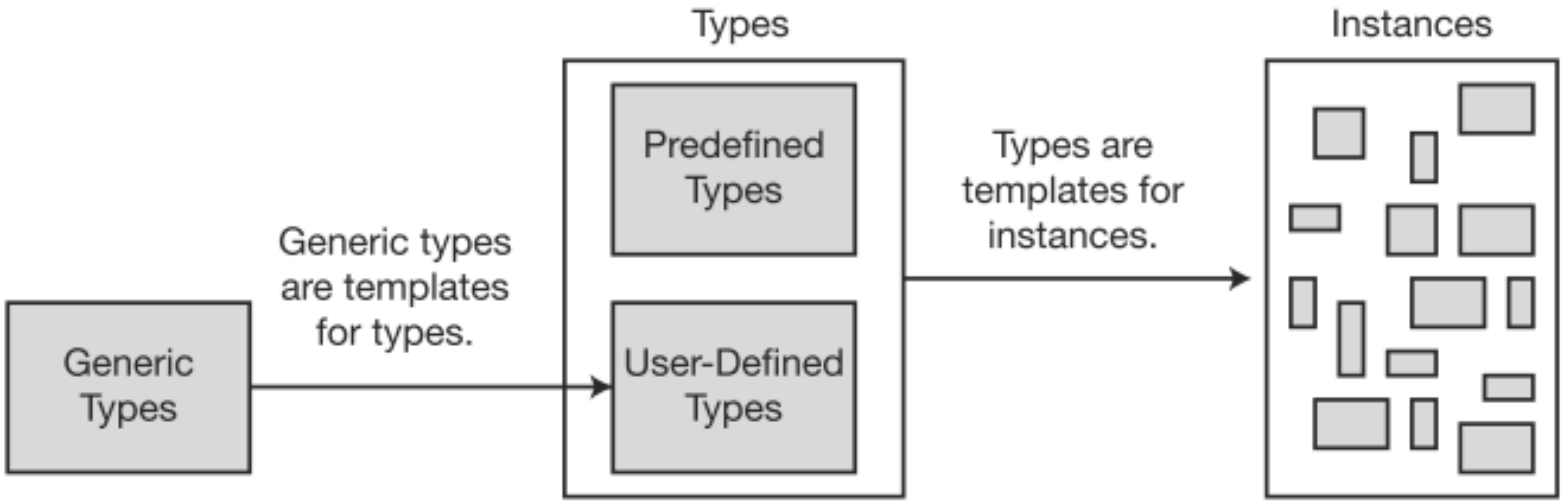
Приклад (Шилдт)

```
using System;
class Gen<T> {
    T ob;
    public Gen(T o) {
        ob = o;
    }
    public T GetOb() {
        return ob;
    }
    public void ShowType() {
        Console.WriteLine("Тип T  " + typeof(T));
    }
}
```

Приклад (закінчення)

```
class Program {
    static void Main() {
        Gen<int> iOb = new Gen<int>(102);
        iOb.ShowType();
        int v = iOb.GetOb();
        Console.WriteLine("Значення: " + v);
        Console.WriteLine();
        Gen<string> strOb = new Gen<string>("Узагальнення");
        strOb.ShowType();
        string str = strOb.GetOb();
        Console.WriteLine("Значення: " + str);
        }
}
```

Generic types are templates for types



Узагальнений клас з двома параметрами типа

```
class TwoGen<T, V>
{
   T ob1;
   V ob2;
   public TwoGen(T o1, V o2) {
         ob1 = o1;
         ob2 = o2;
   }
. . .
}
. . .
TwoGen<int, string> tgObj = new TwoGen<int, string>(119,
   “Alfa”);
```

Обмеження на тип в узагальненнях

- В попередніх прикладах параметри типу можна було замінити довільним типом даних.
- В деяких випадках корисно ввести обмеження на тип.

  Синтаксис:

  class ім'я_класу<T> where T: обмеження { . . .}

  де обмеження вказують списком через кому
- В C# передбачено декілька обмежень на типи:
  - Обмеження **на базовий клас** (**where T: ім'я_класу**)
  - Обмеження **на інтерфейс** (**where T: ім'я_інтерфейсу**)
  - Обмеження **на конструктор** (**where T: new()**)
  - Обмеження **посилального типу** (**where T: class**)
  - Обмеження **типу-значення** (**where T: struct**)

Приклад. public class List<T> where T: IComparable {

Generic-методи

```
static void Swap<T>(ref T lhs, ref T rhs)
{
   T temp;
   temp = lhs;
   lhs = rhs;
   rhs = temp;
}
. . .
int a = 10;
int b = 20;
Swap<int>(ref a, ref b);
```

Generic interfaces

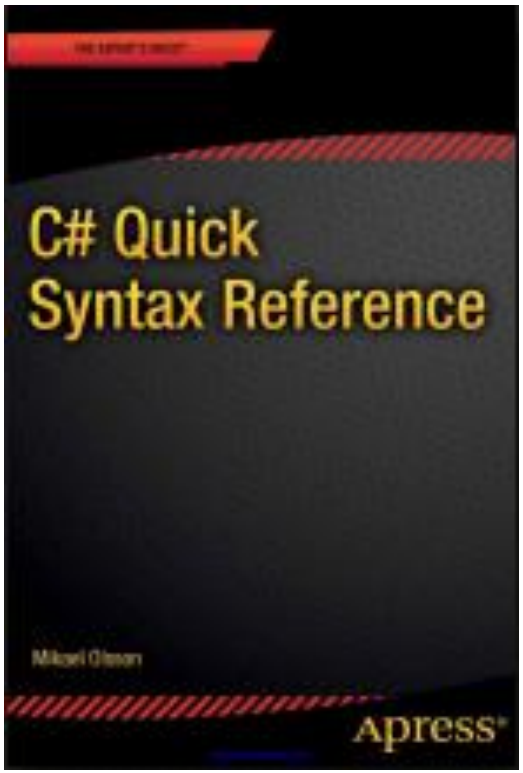
```
interface IGenericCollection<T>
{
  void store(T t);
}

// Non-generic class implementing generic interface
class Box : IGenericCollection<int>
{
  public int myBox;
  public void store(int i) { myBox = i; }
}

// Generic class implementing generic interface
class GenericBox<T> : IGenericCollection<T>
{
  public T myBox;
  public void store(T t) { myBox = t; }
}
```

Generic delegates

```
class MyClass
{
  public delegate void MyDelegate<T>(T arg);
  public void Print(string s)
  {
    System.Console.Write(s);
  }
  static void Main()
  {
    MyDelegate<string> d = Print;
  }
}
```

Generic events

```
delegate void MyDelegate<T, U>(T sender, U
    eventArgs);
event MyDelegate<MyClass, System.EventArgs>
    myEvent;
```



Узагальнені класи колекцій

- Dictionary<TKey, TValue> - зберігає пари "ключ-значення"
- HashSet<T> - множина Set, зберігає ряд унікальних значень
- LinkedList<T> - двонаправлений список
- List<T> - динамічний масив (аналог ArrayList)
- Queue<T> - черга
- Stack<T> - стек
- . . . . .

Common Members of List<T>

Name	Description
Capacity	Місткість (максимальна кількість елементів)
Count	Поточна кількість
Add	Додавання об’єкта в кінець колекції
AddRange	Додавання колекції об’єктів в кінець
BinarySearch	Пошук в сортованому списку
Clear	Видалення всіх елементів
Contains	Визначає, чи належить елемент списку
Find	Пошук елемента, що задовольняє умовам, і повертає перше входження
FindAll	Отримує всі елементи, які відповідають умовам
ForEach	Виконує вказану дію з кожним елементом у списку
Sort	Сортує елементи в списку

Приклад

```
using System;
using System.Collections.Generic;
class Program {
static void Main() {
    List<int> list = new List<int>();
    list.Add(2);
    list.Add(3);
    list.Add(7);
    for (int i = 0; i < list.Count; i++) {
          Console.WriteLine(list[i]);
    }
 }
}
```

Перелічувальний тип (enum)

- Раніше ми розглядали тільки посилальні користувацькі типи: класи та інтерфейси.
- **Однак програміст може оголошувати також користувацькі типи-значення: перелічування та структури**

Перелічувальний тип визначає набір іменованих цілих констант.

[мод-ри] enum [:base_class] {список_перелічування};

Приклади:

```
enum Days{Sat, Sun, Mon, Tue, Wed, Thu, Fri};

enum НомерПланети : uint { Церера = 1, Палада = 2, Юнона = 3, Веста = 4, Ерос = 433, Гідальго = 944}
```

Приклад

```
using System;
class Program{
enum Importance{ None, Trivial, Regular, Important, Critical};
   static void Main() {
        // ... An enum local variable.
        Importance value = Importance.Critical;
        // ... Test against known Importance values.
        if (value == Importance.Trivial) {
        Console.WriteLine("Not true");
        } else if (value == Importance.Critical) {
        Console.WriteLine("True");
        }
   }
}
```

Структури в C#

- **Структури багато в чому схожі на класи**.
- **Як і клас, структура вводить тип**.

[мод-ри] struct [:інтерфейси] { тіло_структури }

- тіло_структури : константи, поля, методи, властивості, індексатори, події, операції, конструктори, вкладені типи

Відмінності структур від класів:

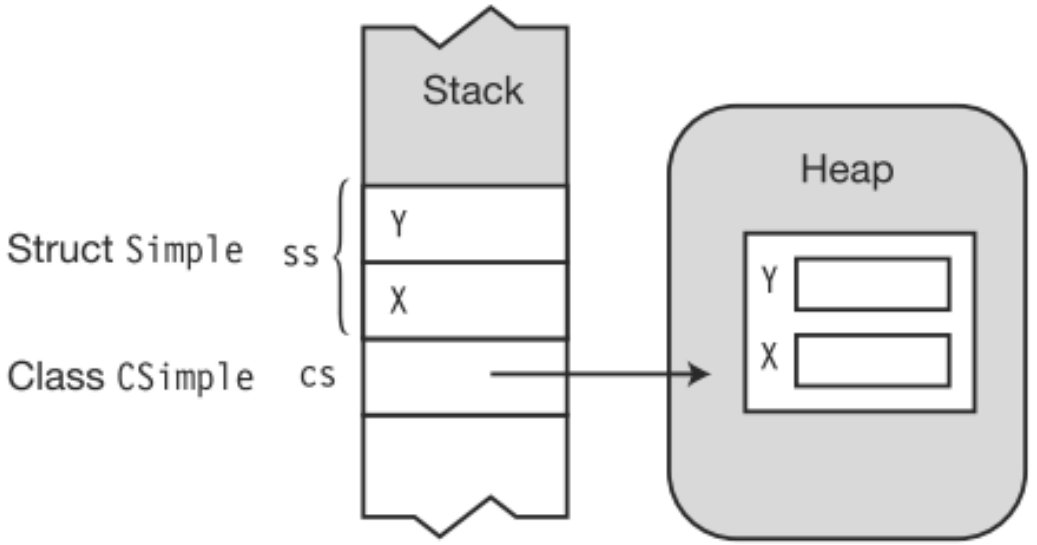
- **Структури – це значущий тип**
- **Структури не підтримують спадкування**
- **Для структур компілятор генерує конструктор без параметрів, який не можна перевизначити**
- **Поля при оголошенні не можна ініціалізувати**
- **Структури при присвоюванні копіюються**

Структури в C#

```
struct Point
{
    public int X;
    public int Y;
}
 class Program {
    static void Main() {
       Point first, second, third;
       first.X  = 10;  first.Y = 10;
       second.X = 20;  second.Y = 20;
       third.X  = first.X + second.X;
       third.Y  = first.Y + second.Y;

       Console.WriteLine( "first:   {0}, {1}“, first.X,  first.Y);
       Console.WriteLine( "second:  {0}, {1}“, second.X, second.Y);
       Console.WriteLine( "third:   {0}, {1}“, third.X,  third.Y);
    }
 }
```

Структури в C#

```
class CSimple
{
    public int X;
    public int Y;
 }
 struct Simple
 {
    public int X;
    public int Y;
  }
   class Program
   {
     static void Main()
     {
        CSimple cs = new CSimple();
        Simple  ss = new Simple();
           ...
```



Приклад (Подбельский, стор.301)

```
using System;
class PointC {
    double x = 10, y = 20;
    public PointC(){ y -=5;}
    public double X { get{ return x;} set{ x=value;} }
    public double Y { get{ return y;} set{ y=value;} }
}
class Program {
    static void Main() {
        PointC pc = new PointC();
        PointC pc1 = pc;
        pc1.X = 10.2;
        Console.WriteLine("X={0}; Y={1}", pc.X, pc.Y);
```

Приклад (закінчення)

```
        PointS ps = new PointS();
        PointS ps1 = ps;
        ps1.X = 10.2;
        Console.WriteLine("X={0}; Y={1}", ps.X, ps.Y);
    }
}
struct PointS {
    //double x = 10, y = 20;
    //public PointS(){ y -=5;}
    double x, y;
    public double X { get{ return x;} set{ x=value;} }
    public double Y { get{ return y;} set{ y=value;} }
}
```

Результат виконання:

X=10.2; Y=15;

X=0; Y=0;

Атрибути в C#

- **У C# дозволяється вводити в програму інформацію декларативного характеру у формі атрибута**, за допомогою якого визначаються **додаткові відомості (метадані), пов'язані з класом**, структурою, методом і т.д.
- **Атрибути вказуються у квадратних дужках перед тим елементом, до якого вони застосовуються**.

```
[Serializable]
class Personage {. . .}
```

- **Існують вбудовані і користувацькі атрибути**
- Після того як атрибут буде пов'язаний з програмною сутністю, він може бути **запитаний під час виконання за допомогою техніки, званої рефлексією**

Атрибути в C#

- **Атрибут підтримується класом, що спадкує від класу System.Attribute**
- **В іменах класів атрибутів прийнято вживати суфікс Attribute**
- **При оголошенні класу атрибута перед його ім'ям вказується атрибут AttributeUsage.**
  - Цей вбудований атрибут позначає типи елементів, до яких може застосовуватися і оголошений атрибут.

Створення атрибуту “Примітка”

(Шилдт, стор.563)

```
[AttributeUsage(AttributeTargets.All)]
public class RemarkAttribute : Attribute {
    string pri_remark; // базове поле
    public RemarkAttribute(string comment) {
            pri_remark = comment;
    }
    public string Remark { //властивість
                    get { return pri_remark;}
    }
}
```

Приєднання атрибута

```
[RemarkAttribute("Атрибут приєднаний до класу.")]
class UseAttrib {
        // ...
}
```

Отримання атрибутів об'єкта

Для витягу атрибуту зазвичай використовується один з двох методів.

- **Перший метод, GetCustomAttributes(), визначається в класі MemberInfo і успадковується класом Type.**
  - **Він витягає список всіх атрибутів, приєднаних до елементу**. Нижче наведена одна з його форм.
    object [] GetCustomAttributes (bool спадкування)
- **Другий метод, GetCustomAttribute (), визначається в класі Attribute**
  - **static Attribute GetCustomAttribute(MemberInfo елемент, Type тип_атрибута)**

Отримання атрибутів об’єкта

```
Type t = typeof(UseAttrib);
   // Витягти атрибут RemarkAttribute
Type tRemAtt = typeof(RemarkAttribute);
RemarkAttribute ra = (RemarkAttribute)
   Attribute.GetCustomAttribute(t, tRemAtt);
```

Повний текст прикладу дивись у Шилдта

Позиційні та іменовані параметри

- Класи атрибутів можуть мати позиційні та іменовані параметри.
- **Позиційні параметри – це параметри конструктора**
- **Іменовані параметри – нестатичні властивості для читання та запису.**

```
[AttributeUsage(AttributeTargets.Class)]
public class HelpAttribute: Attribute{
   public HelpAttribute(string url) {  // Позиційний параметр
      ...
  }
  public string Topic {                  // Іменований параметр
    get {...}
     set {...}
   }
   public string Url { get {...} }
}
```

Позиційні та іменовані параметри

- Цей клас атрибута може бути використаний у такий спосіб:

```
[Help("http://www.mycompany.com/.../Class1.htm")]
class Class1
{
    ...
}
[Help("http://www.mycompany.com/.../Misc.htm", Topic
    = "Class2")]
class Class2
{
    ...
}
```

Зарезервовані атрибути

- System.AttributeUsageAttribute
  - **призначений для опису способів використання класу атрибута**
  - **All = Assembly | Module | Class | Struct | Enum | Constructor | Method | Property | Field | Event | Interface | Parameter | Delegate | ReturnValue**
- System.Diagnostics.ConditionalAttribute
  - **використовується для визначення умовних методів**
- System.ObsoleteAttribute
  - **позначає елемент як застарілий**

Умовні методи

```
#define DEBUG
using System;
using System.Diagnostics;
class Class1
{
   [Conditional("DEBUG")]
   public static void M() {
      Console.WriteLine("Executed Class1.M");
    }
}
class Class2
{
   public static void Test() {
      Class1.M();
   }
}
```

Атрибут Obsolete

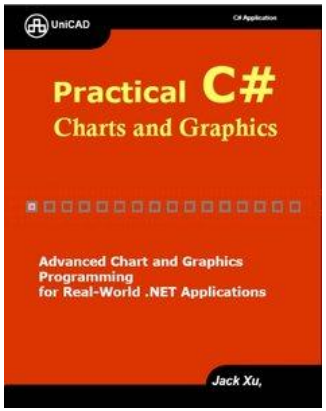
```
[Obsolete(«Цей клас є застарілим; вживайте замість нього клас B")]
class A
{
    public void F() { }
}
class B
{
    public void F() { }
}
class Test
{
    static void Main()
    {
        A a = new A();          // Попередження
        a.F();
    }
}
```

Література

Андерс Хейлсберг, Мэдс Торгерсен, Скотт Вилтамут, Питер Голд **Язык программирования C#**, 4-е издание, СПб: Питер, **2012** – 784 с.

GDI+ Custom Controls with Visual C# 2005
(p.83 – Creating the Clock Control)

Practical C# Charts and Graphics



Література

Чарльз Петцольд
Программирование для Microsoft Windows на C#, Том 1
– М.: Русская Редакция, 2002

(стр. 411 Аналоговые часы)

Петцольд - Программирование с использованием Microsoft Windows Forms, 2006

Візуалізація графічних даних засобами GDI+

- **Троельсен Э. Язык программирования C# 2005 и платформа .NET framework v2.0**, 2007, Глава 20, стор. 790
- **Троельсен Э. Язык программирования C# 2008 и платформа .NET 3.5**, 2010, стор.999
- GDI+ (Graphics Device Interface)

Головні простори імен:

- System.Drawing
  - Graphics, Кисті, пера, шрифти, Color, Point, Rectangle
- System.Drawing.Drawing2D
- System.Drawing.Imaging
- System.Drawing.Printing
- System.Drawing.Text

Клас Graphics

- **Це вхід в функціональні можливості GDI+**
- Представляє поверхню, на якій ви хочете розмістити зображення (форма, елемент управління, пам’ять)
- Має десятки членів, що допомагають відображати текст, зображення, геометричні фігури
- **Клас Graphics не допускає безпосереднього створення свого екземпляра за допомогою ключового слова new, оскільки цей клас не має відкритих конструкторів.**
  - **Створення об’єкта в обробнику події Paint**
  - **Створення об’єкта статичним методом Graphics.FromHwnd()**
  - **Створення об’єкта Graphics на основі зображення: Graphics.FromImage()**

Отримання об’єкта Graphics через подію Paint

```
private void MainWindow_Paint(object sender, PaintEventArgs e)
{
    // Отримати об’єкт Graphics для даного об’єкта Form
    Graphics g = e.Graphics;
    // Намалювати еліпс
    g.FillEllipse(Brushes.Blue, 10, 20, 150, 80);
    // Вивести текст певним шрифтом
    g.DrawString("Hello GDI+", new Font("Times New Roman", 30),
        Brushes.Red, 200, 200);
    // Намалювати лінії спеціальним пером
    using (Pen p = new Pen(Color.YellowGreen, 10))
    {
    g.DrawLine(p, 80, 4, 200, 200);
    }
}
```

Статичний метод Graphics.FromHwnd()

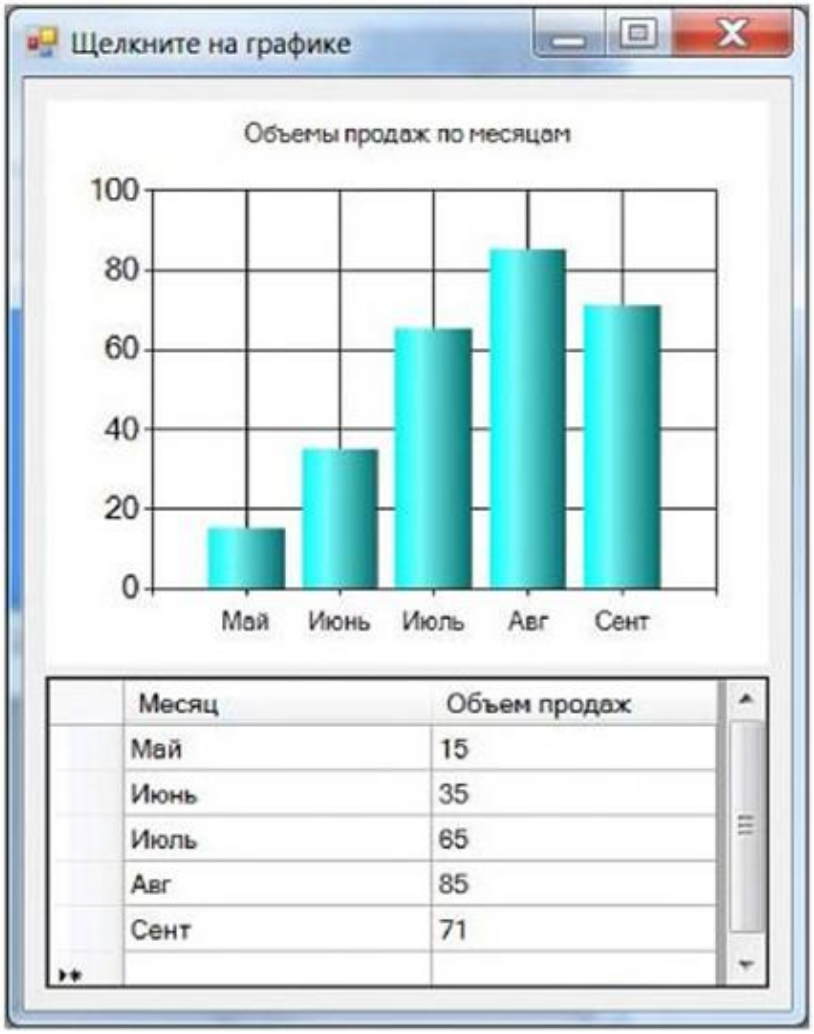
```
private void MainForm_MouseDown(object sender,
    MouseEventArgs e)
{
    Graphics g = Graphics.FromHwnd(this.Handle);
    g.FillEllipse(Brushes.Firebrick, e.X, e.Y, 10,10);
    g.Dispose();
}
```

Об'єкт Graphics на основі зображення

```
private void MainForm_Paint(object sender, PaintEventArgs
    e) {
    //Завантаження локального файлу
    Image myImageFile = Image.FromFile("myPic.jpg");

    //Створення нового об'єкта Graphics на основі зображення
    Graphics imgG = Graphics.FromImage(myImageFile);

    //Візуалізація нових даних
    imgG.FillEllipse(Brushes.DarkOrange, 50, 50, 150,150);

    //Нанесення зображення на форму
    Graphics g = e.Graphics;

    g.DrawImage(myImageFile, new PointF(0.0F, 0.0F));

    //Звільнення створеного нами об'єкта Graphics
    imgG.Dispose();
}
```

Побудова графіка з використанням Chart

Зиборов В. Visual C# 2012 на примерах, стр.190



Застосування засобів введення-виведення

- Шилдт Г. C# 4.0 Полное руководство, гл.14, стр.431

- **Введення-виведення в C# здійснюється через потоки**
- **Байтові та символьні потоки**.
  - На самому низькому рівні – байтами
  - Для символів Unicode потрібне приведення типів
- **Вбудовані потоки**:
  - **Console.In**
  - **Console.Out**
  - **Console.Error**
- **Класи потоків**
  - **Вони визначені в System.IO**
- **Клас Stream**
  - **Основним для потоків є клас System.IO.Stream**
  - **Визначає байтовий поток і є базовим для інших потоків**

Клас Stream

Метод	Опис
void Close()	Закриває поток
void Flash()	Виводе вміст потока на фізичний пристрій
int ReadByte()	Повертає цілочисельне представлення наступного байту (або -1)
int Read(byte[] buffer, int offset, int count)	Читає в буфер count байтів, починаючи з зміщення offset
long Seek(long offset, SeekOrigin origin)	Встановлює поточне положення в потоці
void WriteByte(byte value)	Виводе байт в поток виведення
void Write(byte[] buffer, int offset, int count)	Запис з буфера в поток
bool CanRead	Властивість
bool CanSeek	Властивість
bool CanWrite	Властивість
long Length	Властивість
long Position	Властивість

Класи байтових потоків

- BufferedStream
- FileStream
- MemoryStream
- UnmanageMemoryStream
- Інші потоки для виведення в стиснуті файли, сокети та канали

Класи-оболонки символьних потоків

- **Для створення символьного потоку достатньо заключити байтовий поток в один з класів-оболонок символьних потоків**
- На вершині ієрархії класів символьних потоків знаходяться абстрактні класи TextReader та TextWriter

Методи введення, визначені в TextReader

Метод	Опис
int Peek()	Отримує наступний символ, не видаляючи його
int Read()	Повертає ціле представлення символа
int Read(char[] buf, int i, int count)	Введення в масив
string ReadLine()	Введення рядка
string ReadToEnd()	Введення всіх символів

- Ці методи спроможні генерувати виняток IOException
- В класі TextWriter визначені варіанти методів Write() та WriteLine(), призначені для виведення даних усіх вбудованих типів

Спадкоємці класів TextReader та TextWriter

Клас потока	Опис
StreamReader	Призначений для введення символів з байтового потоку. Цей клас є оболонкою для байтового потоку введення
StreamWriter	Призначений для виведення символів в байтовий поток.
StringReader	Призначений для введення символів з символьного рядка
StringWriter	Призначений для виведення символів в рядок

Двійкові (бінарні) потоки

Окрім байтових та символьних потоків ще існують два класи двійкових потоків:

- BinaryReader
- BinaryWriter

Клас FileStream і байтове введення-виведення в файл

- Цей клас є похідним від класу Stream

Відкриття та закриття файлу.

В класі визначено декілька конструкторів:

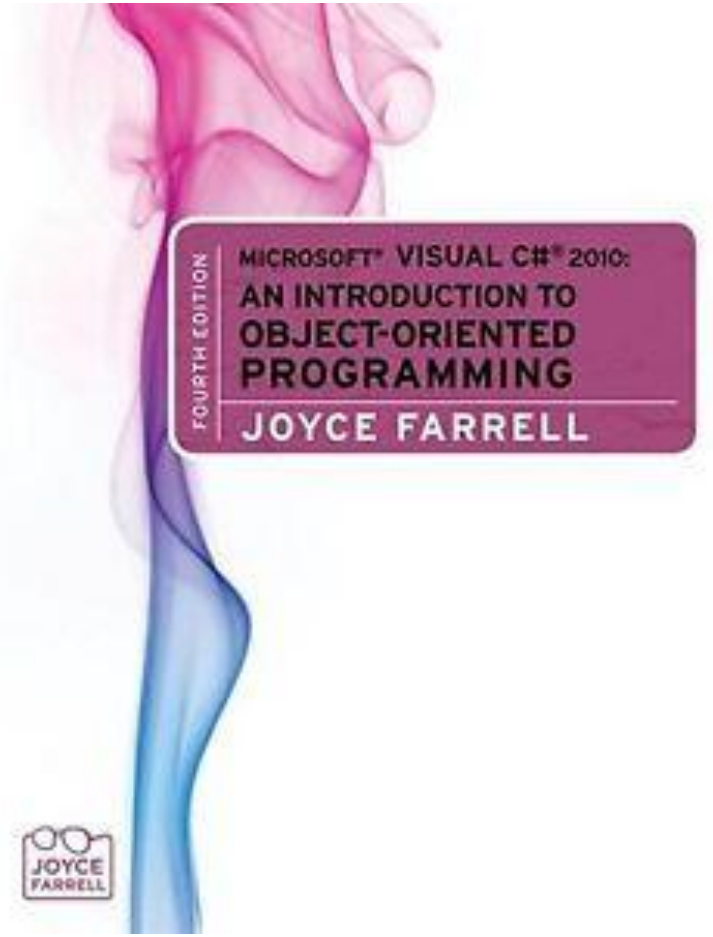
- FileStream(string шлях, FileMode режим)
  - Шлях – ім'я файлу
  - Режим – FileMode.Append (або Create, CreateNew, Open, OpenOrCreate, Truncate)
  - Приклад.
  - FileStream fin;
  - try{ fin=new FileStream("test.dat", FileMode.Open);}
  - catch(Exception e){ Console.WriteLine(e.Message);}
- FileStream(string шлях, FileMode режим, FileAccess доступ)
  - Доступ: FileAccess.Read (Write,ReadWrite)

Приклад

```
using System;
using System.IO;
public class WriteSomeText{
   public static void Main(){
    FileStream outFile = new FileStream( "SomeText.txt" , FileMode.Create,
                                  FileAccess.Write);
    StreamWriter writer = new StreamWriter(outFile);
    Console.Write( "Enter some text >> ");
      string text = Console.ReadLine();
    writer.WriteLine(text);
          // Error occurs if the next two statements are reversed
    writer.Close();
    outFile.Close();
    }
}
```

Запис в файл довільного доступу (1)

(Joyce Farrell Microsoft Visual C# 2010: An Introduction to Object-Oriented Programming, Fourth Edition, 2011, p.695)




```
using System;
using System.IO;
class Employee{
    public int EmpNum {get; set;}
    public string Name {get; set;}
    public double Salary {get; set;}
}
public class WriteSequentialFile{
   public static void Main(){
      const int END = 999;
      const string DELIM = ",";
      const string FILENAME = "EmployeeData.txt";
```

Запис в файл довільного доступу (2)

```
Employee emp = new Employee();
FileStream outFile = new FileStream(FILENAME,
                                    FileMode.Create, FileAccess.Write);
StreamWriter writer = new StreamWriter(outFile);
Console.Write("Enter employee number or " + END +
                                    " to quit >> ");
emp.EmpNum = Convert.ToInt32(Console.ReadLine());
    while(emp.EmpNum != END){
        Console.Write("Enter last name >> ");
        emp.Name = Console.ReadLine();
        Console.Write("Enter salary >> ");
        emp.Salary = Convert.ToDouble(Console.ReadLine());
        writer.WriteLine(emp.EmpNum + DELIM + emp.Name +
                                    DELIM + emp.Salary);
```

Запис в файл довільного доступу (3)

```
    Console.Write("Enter next employee number or " +
          END + " to quit >> ");
    emp.EmpNum = Convert.ToInt32(Console.ReadLine());
}
writer.Close();
outFile.Close();
}
}
```

Читання з файлу довільного доступу (1)

```
using System;
using System.IO;
class Employee{
    public int EmpNum {get; set;}
    public string Name {get; set;}
    public double Salary {get; set;}
}
public class ReadSequentialFile{
   public static void Main(){
          const char DELIM = ',';
          const string FILENAME = "EmployeeData.txt";
          Employee emp = new Employee();
          FileStream inFile = new FileStream(FILENAME,
                              FileMode.Open, FileAccess.Read);
```

Читання з файлу довільного доступу (2)

```
StreamReader reader = new StreamReader(inFile);
string recordIn;
string[] fields;
Console.WriteLine("\n{0,-5}{1,-12}{2,8}\n",
                        "Num", "Name", "Salary");
recordIn = reader.ReadLine();
  while(recordIn != null){
     fields = recordIn.Split(DELIM);
     emp.EmpNum = Convert.ToInt32(fields[0]);
     emp.Name = fields[1];
     emp.Salary = Convert.ToDouble(fields[2]);
```

Читання з файлу довільного доступу (3)

```
Console.WriteLine("{0,-5}{1,-12}{2,8}",
        emp.EmpNum, emp.Name,
        emp.Salary.ToString("C"));
        recordIn = reader.ReadLine();
        }
reader.Close();
inFile.Close();
}
}
```