«ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ»
МІНІСТЕРСТВА ОСВІТИ І НАУКІ УКРАЇНИ

С.Ю. Борю

Посібник із самостійного вивчення базових основ мови програмування C#

Базові основи

навчально-методичний посібник для студентів
природничих спеціальностей

Затверджено вченою радою
ЗНУПротокол № ___ від _____

Запоріжжя2025

УДК: 004.4з (075.8)

ББК: з-973.2-D18 я7зБ 848

Борю С.Ю. Посібник із самостійного вивчення базових основ мови програмування C#. Навчально-методичний посібник для студентів природничо-наукових спеціальностей – Запоріжжя; ЗНУ, 2025, – 101 с.

Розглядаються базові основи об'єкто-орієнтованого програмування у .NET мовою C#. Наведеться велика кількість прикладів та фрагментів програм. Посібник орієнтований на самостійну роботу студентів природничих спеціальностей.

Рецензент – Гоменюк С.І.

Відповідальний за випуск – Матвейшена Н.В.

ЗМІСТ

1. Введення в C

1.1 Мова C# та платформа .NET

Сьогодні мова програмування C# вважається однією з найпотужніших мов, що швидко розвиваються і затребуваних в IT-галузі. На ньому пишуться різні програми: від невеликих «десктопних» програм до великих веб-порталів і веб-сервісів, що обслуговують щодня мільйони користувачів.

Порівняно з іншими мовами C# досить молодий, але в той же час він уже пройшов великий шлях. Перша версія мови вийшла разом із релізом Microsoft Visual Studio .NET у лютому 2002 року. Поточною версією мови є версія C# 12, яка підтримується в рамках платформи .NET 8. Версія C# 12 була випущена разом із .NET 8 14 листопада 2023 року.

Мова C# є об'єктно-орієнтованою і в цьому плані вона багато перейняла з мов Java і C++. Наприклад, C# підтримує інкапсуляцію, поліморфізм, успадкування, навантаження операторів, статичну типізацію. Об'єктно-орієнтований підхід дозволяє вирішити завдання з побудови великих, але в той же час гнучких, масштабованих і додатків, що розширюються. C# продовжує активно розвиватися і з кожною новою версією з'являється все більше цікавих функціональностей, як, наприклад, лямбди, динамічне зв'язування, асинхронні методи і т.д. C# є мовою з подібним синтаксисом і близький, у цьому відношенні, до C++ і Java.

В даний час технології. Текст даної роботи заснований на матеріалах сайтів https://msdn.microsoft.com/library, http://metanit.com, http://www.intuit.ru та, в першу чергу, призначений для самостійного вивчення мови програмування C# на платформі .NET.

1.1.1. Роль платформи.

Коли говорять C#, часто мають на увазі технології платформи .NET (WPF, ASP.NET). І навпаки, коли говорять .NET, нерідко мають на увазі C#. Однак, хоча ці поняття пов'язані, ототожнювати їх не так. Мова C# була створена спеціально для роботи з фреймворком .NET, проте саме поняття .NET дещо ширше.

Фреймворк .NET представляє потужну платформу створення додатків. Можна виділити такі основні риси:

- Підтримка кількох мов. Основою платформи є загальномовне середовище виконання Common Language Runtime (CLR), завдяки чому. При компіляції код будь-якою з цих мов компілюється у складання загальною мовою CIL (Common Intermediate Language) – свого роду асемблер платформи .NET. Тому ми можемо зробити окремі модулі однієї програми окремими мовами.
- Кросплатформеність. .NET є платформою, що переноситься (з деякими обмеженнями). Наприклад, остання версія платформи на даний момент .NET Framework підтримується на більшості сучасних

Windows (Windows 10/8.1/8/7/Vista/XP). Завдяки проекту Mono можна створювати програми, які працюватимуть і на інших ОС сімейства Linux, у тому числі на мобільних платформах Android та iOS.

- Потужна бібліотека класів. .NET представляє єдину всім підтримуваних мов бібліотеку класів. І який би додаток ми не збиралися писати на C# - текстовий редактор, чат або складний веб-сайт - так чи інакше ми використовуємо бібліотеку класів .NET.
- Різноманітність технологій. Загальномовне середовище виконання CLR та базова бібліотека класів є основою цілого стеку технологій, які розробники можуть задіяти при побудові тих чи інших додатків. Наприклад, для роботи з базами даних у цьому стеку технологій призначено технологію ADO.NET. Для побудови графічних програм із багатим насиченим інтерфейсом – технологія WPF. Для створення веб-сайтів – ASP.NET та ін.

Також слід відзначити таку особливість мови C# і фреймворку .NET, як автоматичне складання сміття. Це означає, що програмісту здебільшого не доведеться, на відміну C++, дбати про звільнення пам'яті. Вищезгадане загальномовне середовище CLR сама викличе збирач сміття та очистить пам'ять.

1.1.2. Керований та некерований код

Нерідко програму, створену на C#, називають керованим кодом (managed code). Це означає, що ця програма створена на основі платформи .NET і тому управляється загальномовним середовищем CLR, яке завантажує програму і при необхідності очищує пам'ять. Але є також програми, наприклад, створені мовою C++, які компілюються над загальну мову CIL, як C# чи VB.NET, а звичайний машинний код. У цьому випадку .NET не керує програмою. Зазначимо, що платформа .NET надає можливості для взаємодії з некерованим кодом.

1.1.3. JIT-компіляція

Як зазначалося вище, код C# компілюється в додатки або збірки з розширеннями exe або dll мовою CIL. Далі при запуску на виконання подібної програми відбувається JIT-компіляція (Just-In-Time) у машинний код, який потім виконується. При цьому, оскільки програма може бути великою і містити багато інструкцій, в даний момент компілюватиметься лише та частина програми, до якої безпосередньо йде звернення. Якщо буде звернення до іншої частини коду, вона буде скомпілювана з CIL в машинний код. А вже скомпільована частина програми зберігається до завершення роботи програми. У результаті це підвищує продуктивність. По суті, це все, що коротенько треба знати про платформу .NET.

1.2. Початок роботи. Visual Studio

Для створення програми (програми) мовою C# потрібно:

- По-перше, текстовий редактор, з якого створюється файл вихідного коду програми.
- По-друге, компілятор, для компіляції програмного коду в додаток exe.
- По-третє, фреймворк .NET, для JIT компіляції та виконання робочої програми.

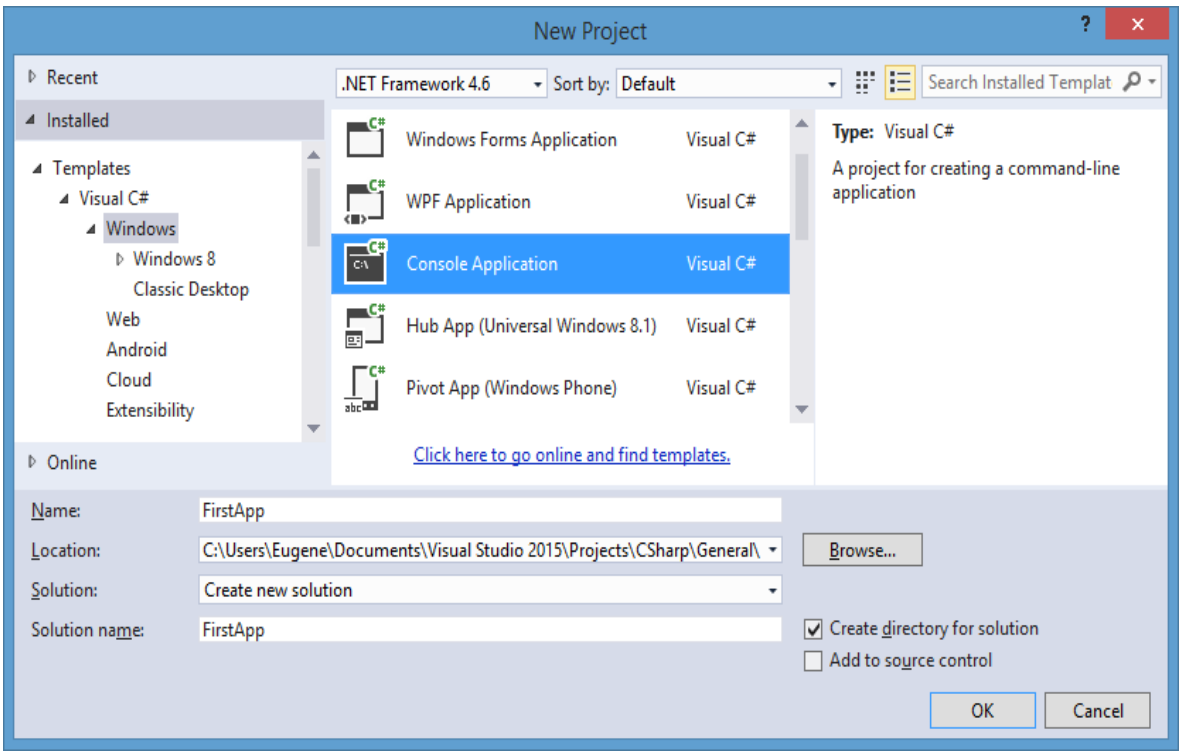
Щоб полегшити записи вихідного коду програми, а також тестування та налагодження програмного коду зазвичай використовують спеціальні середовища розробки, зокрема Visual Studio.

Для створення програм на C# можна використовувати безкоштовне та повнофункціональне середовище розробки - Visual Studio Community 2022, яке можна завантажити за наступною адресою:Microsoft Visual Studio 2022 (https://www.visualstudio.com/en-us/downloads). Також можна використовувати Visual Studio.

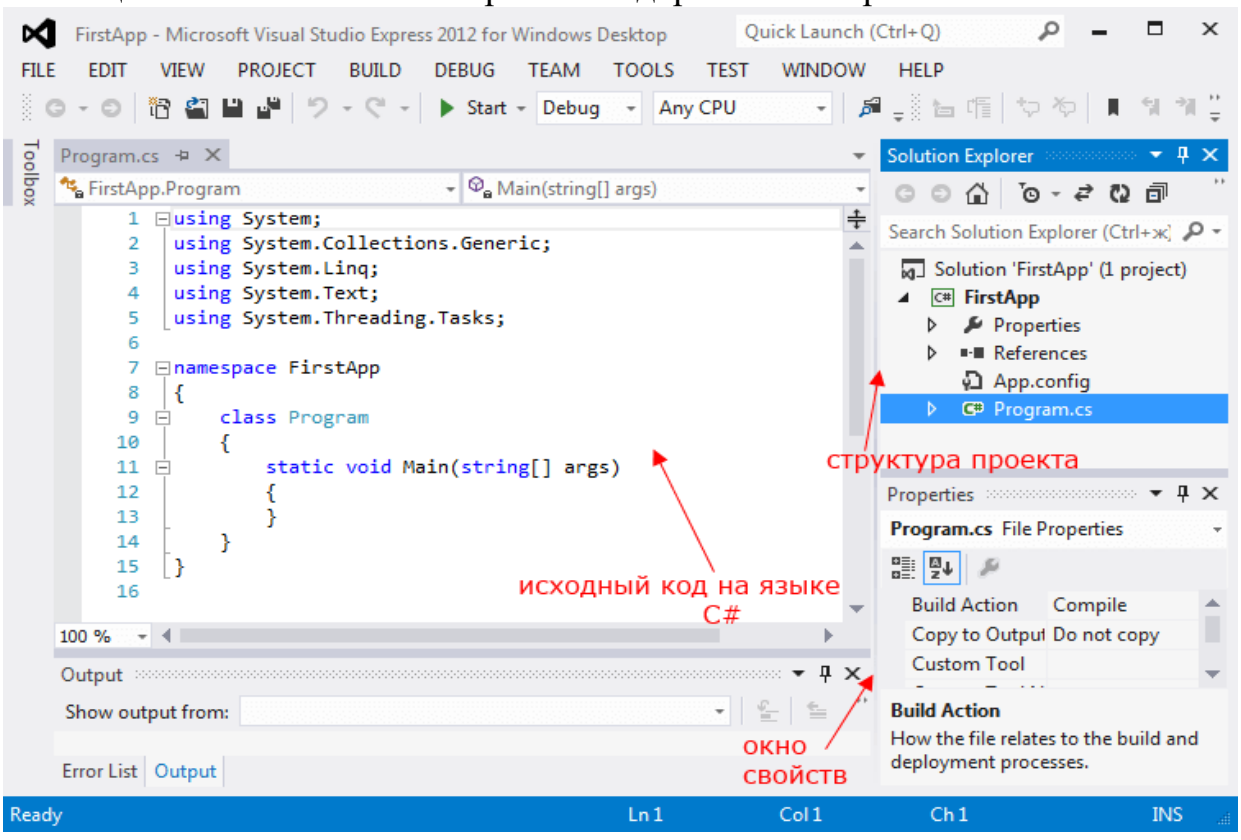
При інсталяції Visual Studio на комп'ютер з операційною системою WINDOWS будуть встановлені всі необхідні інструменти для розробки програм, у тому числі фреймворк .NET 4.8.

Розглянемо створення першої (дуже простої, «консольної») програми. Спочатку запустимо на виконання ("відкриємо") Visual Studio і вгорі в рядку меню виберемо пункт File (Файл) -> New (Створити) -> Project (Проект). Відкриється діалогове вікно створення нового проекту.

Тут у центрі виберемо пункт Console Application, оскільки перша програма буде консольною. Внизу в полі Name введемо назву проекту (не використовуйте "національний" абетка). У цьому прикладі FirstApp. І натиснемо ОК.



Після цього Visual Studio створить та відкриє новий проект:



У "великому полі" в центрі, яке, по суті, представляє поле (вікно) спеціалізованого текстового редактора, знаходиться за замовчуванням вихідний код C#. Згодом його змінюють інший код.

Праворуч знаходиться вікно Solution Explorer, в якому можна побачити структуру проекту, що розробляється. В даному випадку згенерована «за умовчанням» структура:

- вузол "Properties або Властивостей" - він "зберігає" файли властивостей додатків і поки не потрібен;
- вузол "References" - це вузол містить складання dll, які додані в проект за замовчуванням.

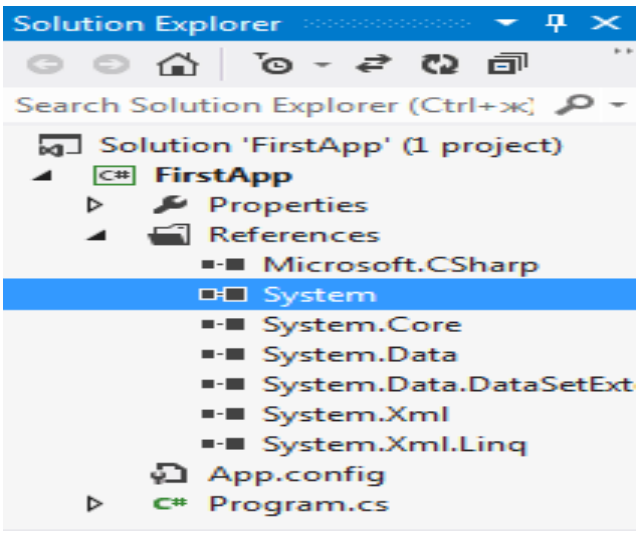
Ці збірки містять класи бібліотеки .NET, які використовуватиме C#. Однак не завжди всі збирання потрібні. Непотрібні бібліотеки можна видалити, в той же час, якщо потрібно - додати.

Далі йде файл конфігурації App.config (поки він не використовується і не цікавий) і безпосередньо сам файл коду програми Program.cs. Саме цей файл і відкритий для редагування та перегляду в центральному вікні.

Розберемо, що цей код означає:

```
/*початок секції просторів імен, що підключаються*/
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
/*кінець секції просторів імен, що підключаються*/
namespace FirstApp {/*оголошення нового простору імен*/
class Program {/*оголошення нового класу*/
static void Main(string[] args) {/*оголошення нового
методу*/
}/* кінець оголошення нового методу*/
} /* кінець оголошення нового класу*/
}/* кінець оголошення нового простору імен*/
```

На початку файлу записані директиви using, після яких записуються назви просторів імен, що підключаються.**Простір імен**є організацією класів у загальні блоки. Наприклад, першому рядку using System; підключається простір іменSystem, що містить фундаментальні та базові класи платформи .NET. Фізично простори імен знаходяться в бібліотеках dll, що підключаються, які можна побачити у вікні Solution Explorer, відкривши вузол References:



Там можна побачити бібліотеку System.dll, яка містить класи з простору імен System. Однак точної відповідності між просторами імен та назвами файлів dll немає.

Другим рядком знову ж таки підключається вкладений простір імен System.Collections.Generic: тобто в просторі імен System визначено простір імен Collections, а вже в ньому простір імен Generic.

І оскільки C# має Сі-подібний синтаксис, кожен рядок завершується крапкою з комою, а кожен блок коду поміщається у фігурні дужки.

Далі починається вже власне «наш» простір імен, який буде створювати окрему збірку або програму, що виконується: спочатку йде ключове слово namespace, після якого назва простору імен. За промовчанням Visual Studio дає назву проекту. Далі всередині фігурних дужок йде блок простору імен.

Простір імен може включати інші місця або класи. В даному випадку за умовчанням згенеровано один клас – Program. Класи оголошуються схожим способом – спочатку йде ключове слово class, а потім назва класу, і далі блок самого класу у фігурних дужках.

Клас може містити різні змінні, методи, властивості та інші інструкції. В даному випадку оголошено один метод Main. Зараз "він порожній і нічого не робить" - текст програми відсутній.

У програмі на C# метод Main є вхідною точкою програми, з нього починається все виконання програми (йому операційна система передає управління при старті програми). Він обов'язково має бути присутнім у програмі саме у такому «виді» (з такою сигнатурою), як ця сторока була «згенерована» Visual Studio.

Слово static вказує, що метод Main – статичний, а слово void – що він не повертає жодного значення. Для виклику статичного методу не потрібен об'єкт, що стерже цей метод (статичний метод "завжди знаходиться в пам'яті").

Далі в дужках йдуть параметри методу - string[] args - це масив args, який зберігає значення типу string, тобто рядки. В даному випадку параметри методу не використовуються, але в реальній програмі це параметри, які передаються при запуску програми з консолі - параметр командного рядка.

Змінимо весь цей код на наступний:

```
using System;
```

```
namespace FirstApp {
class Program {
static voidMain(string[] args) {
            Calculator calc = New Calculator();
calc.Add(2, 3);
}
}
   // оголошення нового класу
class Calculator {
public void Add(int x, int y) {
int z = x + y;
Console.WriteLine("Сума {0} і {1} дорівнює {2}", x, y,
z);
Console.ReadLine();
}
}
}
```

Порівняно з автоматично згенерованим кодом внесено кілька змін.

По-перше, прибрано підключення непотрібних просторів імен, оскільки вони у разі не потрібні.

По-друге, додано в наш простір імен новий клас - Calculator, який має один метод Add. Цей метод приймає як параметри два цілих числа - x і y і повертає їхню суму. Значення суми виводиться на консоль за допомогою Console.WriteLine. Метод Console.ReadLine використовується для очікування введення від користувача (щоб вікно консолі не закривалося відразу після виведення результату).

У цих двох рядках коду виконано звернення до класу Console, що знаходиться у просторі імен System. Цей простір підключено директивою using.

Необов'язково підключати простір імен директивою using. Можна видалити перший рядок, але для виклику методів використовуваних класів необхідно вказати «повний шлях» до використовуваного класу. Необхідно кодувати: System.Console.ReadLine().

Після оголошення нового класу можна використовувати його в методі Main:

```
Calculator calc = New Calculator(); // Створення об'єкта
// Нового класу
calc.Add(2, 3); // Виклик методу Add нового класу
```

Далі буде описано, як створюються об'єкти та чому саме такий синтаксис використовується. Зараз досить розуміти, що всі дії, з яких починається програма, записуються у методі Main. Метод Main – це «вхідна точка» у програму.

Тепер можна «запустити» на виконання програму – натиснувши клавішу F5 або на панелі інструментів, натиснувши на зелену стрілку. І якщо все зроблено правильно, відкриється вікно консолі, де буде виведено число 5 - тобто сума чисел 2 і 3.

Отже, перша програма (програма) створена. Його можна "знайти" на жорсткому диску в папці проекту в каталозі bin\Debug (воно буде називатися на ім'я проекту, і матиме розширення exe). "Створену програму" потім вже можна виконувати без Visual Studio, а також переносити програму на інші комп'ютери з операційною системою WINDOWS, де є підтримка технології .NET.

Тих же результатів можна досягти без використання Visual Studio. Використовуючи будь-який текстовий редактор, можна створити файл (наприклад pgm.cs), що містить програму мовою C#. Потім виконавши «у консольному режимі»:

```
%windir%\Microsoft.NET\Framework\v4.0.30319\csc.exe
pgm.cs
```

отримаємо файл pgm.exe, що містить "робочу програму".

Далі розглядаються основні складові синтаксису мови C#.

1.3. Контрольні питання

1 Основні можливості Фреймворку.
2 Що таке керований та некерований код?
3 У чому полягає основна ідея JIT-компіляції?
4 Програма Visual Studio Community є платною чи безкоштовною?
5 Що необхідно для створення програми (програми) мовою C#?
6 Навіщо використовують директиви using?
7 Який метод у програмі на C# є вхідною точкою програми?
8 Які найчастіше використовувані методи містить клас Console?
9 У якому каталозі записується збірка програми, створеної за допомогою програми Visual Studio?
10 Як створити власну програму без використання програми Visual Studio?

2. Основи програмування на C#

2.1. Типи даних та змінні

Як і в багатьох мовах програмування, у C# є своя система типів даних, яка використовується для роботи зі змінним. Вона представлена такими типами:

- **`bool`**: зберігає значення true або false Представлений системним типом System.Boolean
- **`byte`**: зберігає ціле число від 0 до 255 і займає 1 байт. Представлений системним типом System.Byte
- **`sbyte`**: зберігає ціле число від -128 до 127 і займає 1 байт. Представлений системним типом System.SByte
- **`short`**: зберігає ціле число від -32768 до 32767 і займає 2 байти. Представлений системним типом System.Int16
- **`ushort`**: зберігає ціле число від 0 до 65535 і займає 2 байти. Представлений системним типом System.UInt16
- **`int`**: зберігає ціле число від -2147483648 до 2147483647 і займає 4 байти. Представлений системним типом System.Int32
- **`uint`**: зберігає ціле число від 0 до 4294967295 і займає 4 байти. Представлений системним типом System.UInt32
- **`long`**: зберігає ціле число від -9223372036854775808 до 9223372036854775807 і займає 8 байт. Представлений системним типом System.Int64
- **`ulong`**: зберігає ціле число від 0 до 18446744073709551615 і займає 8 байт. Представлений системним типом System.UInt64
- **`float`**: зберігає число з плаваючою точкою від -3.4*1038 до 3.4*1038 і займає 4 байти. Представлений системним типом System.Single
- **`double`**: зберігає число з точкою, що плаває від ±5.0*10324 до ±1.7*10308 і займає 8 байти. Представлений системним типом System.Double
- **`decimal`**: зберігає десяткове дробове число. Якщо вживається без десяткової коми, має значення від 0 до +/- 79228162514264337593543950335; якщо з комою, то від 0 до +/- 7,9228162514264337593543950335 з 28 розрядами після коми і займає 16 байт. Представлений системним типом System.Decimal
- **`char`**: зберігає одиночний символ у кодуванні Unicode і займає 2 байти. Представлений системним типом System.Char
- **`string`**: зберігає набір символів Unicode Представлений системним типом System.String
- **`object`**: може зберігати значення будь-якого типу даних і займає 4 байти на 32-розрядній платформі та 8 байт на 64-розрядній платформі. Представлений системним типом System.Object, який є базовим для всіх інших типів та класів .NET.

2.1.1. Оголошення змінних

Загальний спосіб оголошення змінних наступний:

```
тип_даних назва_змінної;
```

Наприклад, int x; У цьому вся виразі оголошується змінна x типу int, нею «виділяється оперативна пам'ять для зберігання «значення». Тобто x зберігатиме деяке ціле число, закодоване (як правило) чотирма байтами.

Як ім'я змінної може виступати будь-яка довільна назва, яка задовольняє наступним вимогам:

- ім'я має містити не більше 255 символів
- Ім'я може містити будь-які цифри, літери англійського алфавіту та символ підкреслення, при цьому перший символ в імені має бути буквою або символом підкреслення
- в імені не повинно бути знаків пунктуації та пропусків
- ім'я не може бути ключовим словом мови C#

Оголосивши змінну, можна «тут же» надати їй значення або ініціалізувати її. Варіанти оголошення змінних:

```
boolisEnabled = true;
intx;
doubley=3.0;
stringhello = "Hello World";
stringhello1 = @ "Hello World";
charc='s';
inta=4;
intz=a+5;
```

2.1.2. Використання суфіксів та префікса @

При присвоєнні числових значень треба пам'ятати деякі угоди. При присвоєнні змінним типу float і decimal чисел з плаваючою точкою, Visual Studio розглядає ці цифри як значення типу double. Щоб конкретизувати, що це значення має розглядатися як float, необхідно використовувати суфікси f і m відповідно для float і decimal:

```
floatb =30.6f;
decimald=334.8m;
strings1="C:\temp\test.txt";
string  s2=@" C:\temp\test.txt";
```

Оголошення та ініціалізацію рядків можна виконувати різними способами. При цьому часто використовуються «літерали». Літерал - це, грубо кажучи, "зафіксоване" в коді програми "значення" рядка.

Є два типи рядкових літералів у C# - стандартні (regular) і дослівні (verbatim).

Стандартні літерали в C# схожі з такими в більшості мов програмування - вони обрамляються в подвійні лапки ("), а також можуть містити спеціальні символи (власне подвійні лапки ("), зворотний сліш (\), перенесення рядка (carriage return - CR), подача рядка (line feed - інші).

Дослівні літерали дозволяють майже те саме, що й стандартні, проте дослівний літерал закінчується на перших, не продубльованих подвійних лапках. Щоб вставити в дослівний літерал подвійні лапки, потрібно їх продублювати (""). Також, на відміну від стандартного літерала, у дослівному літералі можуть бути символи повернення каретки та перенесення рядка без екранування. Для використання дослівного літерала необхідно вказати @ перед відкриттям лапкою. Нижче в таблиці зібрані приклади, що демонструють різницю між описаними типами літералів.

Стандартний літерал	Дослівний літерал	Результуючий рядок
`"Hello"`	`@ "Hello"`	`Hello`
`"Зворотний сліш: \\"`	`@"Зворотний сліш: \"`	`Зворотний сліш: \`
`"Подвійна лапка:"`	`@"Подвійна лапка: """`	`Подвійна лапка: "`
`"CRLF:\r\n Після CRLF"`	`@"CRLF:Після CRLF"`	`CRLF:Після CRLF`

Слід зазначити, що стандартний і дослівний літерали існують лише програміста і компілятора C#. Як тільки код скомпільований, всі літерали наводяться до єдиного способу зберігання оперативної пам'яті.

Список спеціальних символів, які потребують екранування:

\' - одинарна лапка, використовується для оголошення літералів типу System.Char

- подвійна лапка, використовується для оголошення рядкових літералів.

\\ — зворотний сліш

\0 - null-символ в Юнікоді

\a - символ Alert (№7)

\ b - символ Backspace (№ 8)

\f —зміна сторінки FORM FEED (№12)

\n - переклад рядка (№10)

\r - повернення каретки (№13)

\t - горизонтальна табуляція (№9)

\v - вертикальна табуляція (№ 11)

Uxxxx - символ Юнікоду з шістнадцятковим кодом xxxx

\xn[n][n][n] — символ Юнікоду з шістнадцятковим кодом nnnn, версія попереднього пункту зі змінною довжиною цифр коду

\Uxxxxxxxx - символ Юнікоду з шістнадцятковим кодом xxxxxxxx, використовується для виклику сурогатних пар.

Насправді символи \a, \f, \v, \x і \U використовуються досить рідко.

2.1.3. Використання системних типів

Вище, при перерахуванні всіх базових типів даних, кожного згадувався*системний тип*. Назва вбудованого типу, по суті, є скороченим позначенням системного типу. Наприклад, наступні змінні будуть еквівалентні за типом:

```
int a = 4;
System.Int32 b = 4;
```

2.1.4. Неявна типізація

Раніше явно вказували тип змінних, наприклад, int x;. І компілятор «вже знав», що x зберігає ціле значення.

Однак можна використовувати неявну типізацію:

```
varstroka = "Hell to World";
varc = 20;

Console.WriteLine(c.GetType().ToString());
Console.WriteLine(stroka.GetType().ToString());
```

Для неявної типізації замість назви типу даних використовують ключове слово var. Потім вже за компіляції компілятор сам «виводить» тип даних з присвоєного значення.

У прикладі використовувався вираз

Console.WriteLine(c.GetType().ToString());

яке дозволяє дізнатися виведений тип змінної c. Так як, «за замовчуванням», всі цілі значення розглядаються як значення типу int, то, тому, змінна c матиме тип int або System.Int32

Ці змінні подібні до «звичайних», однак вони мають деякі обмеження.

По-перше, не можна спочатку оголосити змінну, що неявно типізується, а потім її ініціалізувати:

```
// Цей код працює
inta;
a = 20;

// Цей код не працює
varc;
c = 20;
```

По-друге, не можна вказати як значення змінної, що неявно типізується, «значення» null:

```
// Цей код не працює
varc=null;
```

Для значення null компілятор зможе «вивести» тип даних.

2.2. Область видимості (контекст) змінних

Кожна змінна доступна у межах певного контексту чи області видимість. Поза цим контекстом змінної вже немає.

Існують різні контексти:

- **Контекст класу.**Змінні, визначені на рівні класу, доступні у будь-якому методі цього класу
- **Контекст методу**. Змінні, визначені лише на рівні методу, є локальними і доступні лише рамках даного методу. В інших методах вони недоступні

- **Контекст блоку коду**. Змінні, визначені на рівні блоку коду, також локальні і доступні тільки в рамках даного блоку. Поза своїм блоком коду вони не дістануться.

Наприклад, нехай клас Program визначений так:

```
classProgram { // початок контексту класу
static int a = 9; // змінна рівня класу
staticvoid Main(string[] args) { // початок контексту // методу Main
int b = a – 1; // змінна рівня методу
{ // початок контексту блоку коду
int c = b – 1; // змінна рівня блоку // коду
} // кінець контексту блоку коду, // змінна з знищується
        //оскільки у наступному операторі - не можна, //
змінна c визначено у блоці коду
Console.WriteLine(c);
        //оскільки у наступному операторі - не можна, //
змінна d визначено іншому методі
Console.WriteLine(d);
Console.Read();
} // кінець контексту методу Main, // змінна b знищується
void Display() { // початок контексту методу Display
// змінна a визначена у тих класу, // тому доступна
int d = a + 1;
} // кінець конексту методу Display, // змінна d
знищується
}// кінець контексту класу, змінна a знищується
```

Тут виразно чотири змінних: a, b, c, d. Кожна їх існує у своєму контексті. Змінна a існує в контексті всього класу Program і доступна в будь-якому місці та блоці коду у методах Main та Display.

Змінна b існує лише у межах методу Main. Так само як і змінна d існує у межах методу Display. У методі Main не можна звернутися до змінної d, оскільки вона визначена іншому контексті.

Змінна c існує тільки в блоці коду, межами якого є фігурні дужки, що відкриває і закриває. Поза його межами змінна c не існує і до неї не можна звернутися.

Зазвичай межі різних контекстів можна асоціювати з фігурними дужками, що відкриваються і закриваються, як у даному випадку, які задають межі блоку коду, методу, класу.

При роботі зі змінними треба враховувати, що локальні змінні, визначені в методі або блоці коду, приховують (екранують) змінні рівня класу, якщо їх імена збігаються:

```
classProgram {
static int a = 9; // змінна рівня класу
static void Main(string[] args) {
int a = 5; // приховує змінну a // яка оголошена на рівні
класу
```

```
Console.WriteLine(a); // 5
}
}
```

При оголошенні змінних також треба враховувати, що в одному контексті не можна визначити кілька змінних з тим самим ім'ям.

2.3. Перетворення базових типів даних

При розгляді типів даних вказувалося, які значення може мати той чи інший тип і скільки байт пам'яті може займати. Можна написати, наприклад, так:

```
byte a = 4;
int b = a + 70;
```

Але важливо розуміти, що цей запис не еквівалентний наступній (хоча результат буде той самий):

```
byte a = 4;
byte b = (byte) (a + 70);
```

У першому прикладі компілятор застосовує перетворення типів: він перетворює дані типу byte типу int. Цей тип перетворень називається розширюючим (widening), оскільки значення типу byte розширює свій розмір до розміру типу int чи System.Int32. За таких перетворень, як правило, проблем не виникає.

Крім розширювальних перетворень є ще й звужують (narrowing). У другому випадку виходить звужувальне перетворення.

Або, припустимо, треба присвоїти змінній типу byte суму двох змінних типу int:

```
int a = 4;
int b = 6;
byte c = a+b;
```

Незважаючи на те, що сума (число 10) вкладається в діапазон типу byte, Visual Studio все одно відобразить помилку. І щоб уникнути цієї ситуації, треба застосувати явне перетворення на тип**byte**.

```
byte c = (byte) a + b;
```

2.3.1. Явні та неявні перетворення

У разі розширення перетворення, компілятор сам виконував всі перетворення даних. Ці перетворення були неявними (implicit conversion).

При явних перетвореннях (explicit conversion) програміст повинен застосувати операцію перетворення - (тип). Кодується операція перетворення типів так: перед значенням вказується в дужках тип, до якого треба привести це значення:

```
int a = 4;
int b = 6;
byte c = (byte) (a + b);
```

2.3.2. Втрата даних та ключове слово checked

Розглянемо іншу ситуацію: що буде, наприклад, у такому разі:

```
int a = 33;
int b = 600;
byte c = (byte) (a + b);
```

Результатом буде число 121, так число 633 не потрапляє у допустимий діапазон для типу byte, і старші біти будуть усікатися (обкидатись). У результаті вийде число 121 і виняток не буде вибрано. Тому, при перетвореннях, треба це враховувати.

В даному випадку можна або взяти такі значення чисел a і b, які в сумі дадуть число не більше 255, або можна вибрати замість byte інший тип даних, наприклад, int.

У загальному випадку, невідомо, які значення будуть мати числа a і b. Щоб уникнути подібних ситуацій, C# має ключове слово checked:

```
try {
int a = 33;
int b = 600;
byte c = checked((byte)(a + b));
Console.WriteLine(c);
}catch (OverflowException ex) {
Console.WriteLine(ex.Message);
}
```

При використанні ключового слова checked програма викидає виняток про переповнення. Тому його обробки, у разі, використовується конструкція try...catch. Докладніше дана конструкція та обробка винятків розглядається пізніше, а поки що треба знати, що в блок try включаються дії, у яких потенційно може виникнути помилка, а блоці catch – код обробки помилки.

2.4. Операції мови C#

C# використовується більшість операцій, які застосовуються і в інших мовах програмування. Операції бувають*унарними*(виконуються над одним операндом),*бінарними*- над двома операндами та*тернарними*- Виконуються над трьома операндами. Операндом є змінна чи значення (наприклад, число), що бере участь в операції. Розглянемо всі види операцій.

2.4.1. Математичні операції

- `+операція складання двох чисел: z = x + y`
- `-операція віднімання двох чисел: z = xy`
- операція множення двох чисел: z = x * y
- `/операція поділу двох чисел: z=x/y`
- `%одержання залишку від поділу двох чисел: z=x%y`

- **++**(Префіксний інкремент)
  z=++y (спочатку значення змінної y збільшується на 1, та був її значення присвоюється змінної z)
- **++**(Постфіксний інкремент)
  z=y++ (спочатку значення змінної y надається змінної z, а потім значення змінної y збільшується на 1)
- **-**(Префіксний декремент)
  z=--y (спочатку значення змінної y зменшується на 1, а потім її значення присвоюється змінною z)
- **-** (Постфіксний декремент)
  z=y--(спочатку значення змінної y надається змінної z, а потім значення змінної y зменшується на 1)

Приклади:

```
stringhello = "hello" + "world";
//результат дорівнює "hello world"

intx1 = 2 + 4; //Результат дорівнює 6
intx2 = 10 – 6; //Результат дорівнює 4
intx3 = 10*6; //Результат дорівнює 60
doublex4 = 10.0/4.0; //Результат дорівнює 2.5
doublex5 = 10.0% 4.0; //Результат дорівнює 2
inty1 = 5;
intz1 = ++y1; // z1 = 6; y1=6
inty2 = 5;
intz2 = y2++; // z2 = 5; y2=6
inty3 = 5;
intz3 = -y3; // z3 = 4; y3=4
inty4 = 5;
intz4 = y4--; // z4 = 5; y4=4
```

2.4.2. Логічні операції над числами

Логічні операції над числами виконуються порозрядно. Числа розглядаються в двійковому поданні, наприклад, 2 у двійковому поданні 10 і має два розряди, число 7 - 111 і має три розряди.

- **&**(логічне множення)

  Множення проводиться порозрядно, і якщо в обох операнда значення розрядів дорівнює 1, то операція повертає 1, інакше повертається число 0. Наприклад:

```
intx1 = 2; //010
inty1 = 5;//101
Console.WriteLine(x1 & y1); // виведе 0

intx2 = 4; //100
```

```
int y2 = 5; //101
Console.WriteLine(x2 & y2); // виведе 4
```

У першому випадку ми два числа 2 і 5. 2 у двійковому вигляді представляє число 010, а 5 - 101. Порозрядно помножимо числа (0*1, 1*0, 0*1) й у результаті отримаємо 000.

У другому випадку у нас замість двійки число 4, у якого в першому розряді 1, так само як і у числа 5, тому отримаємо (1*1, 0*0, 0 *1) = 100, тобто число 4 у десятковому форматі.

- **|**(логічне додавання)

Схоже на логічне множення, операція також здійснюється за двійковими розрядами, але тепер повертається одиниця, якщо хоча б у одного числа в даному розряді є одиниця. Наприклад:

```
int x1 = 2; //010
int y1 = 5;//101
Console.WriteLine(x1|y1); // виведе 7 - 111
int x2 = 4; //100
int y2 = 5;//101
Console.WriteLine(x2 | y2); // виведе 5 - 101
```

- **^**(логічне виключне АБО)

Також цю операцію називають XOR, нерідко її застосовують для простого шифрування:

```
int x = 45; // Значення, яке треба зашифрувати – //
Двійковій формі 101101
int key = 102; //Нехай це буде ключ -//в двійковій формі
1100110
int encrypt = x ^ key; //Результатом буде // число
1001011 чи 75
Console.WriteLine("Зашифроване число: +encrypt);

int decrypt = encrypt^key; // Результатом буде
                           // вихідне число 45
Console.WriteLine("Розшифроване число:" + decrypt);
```

Тут знову ж таки проводяться порозрядні операції. Якщо значення поточного розряду в обох чисел різні, то повертається 1, інакше повертається 0. Таким чином, отримуємо з 9^5 як результат число 12. І щоб розшифрувати число, застосовуємо ту ж операцію до результату.

- **~**(логічне заперечення)

Ще одна порозрядна операція, яка інвертує всі розряди: якщо значення розряду дорівнює 1, воно стає рівним нулю, і навпаки.

```
int x = 9;
Console.WriteLine(~x);
```

2.4.3. Операції зсуву

Операції зсуву також виконуються над розрядами чисел. Зсув може відбуватися праворуч і ліворуч.

- `X << y – зсуває число x ліворуч на y розрядів.`
  Наприклад, 4<<1 зрушує число 4 (яке в двійковому поданні 100) на один розряд вліво, тобто в результаті виходить 1000 або число 8 у десятковому поданні.
- `X >> y`- Зсуває число x вправо на y розрядів.
  Наприклад, 16>>1 зсуває число 16 (яке в двійковому поданні 10000) на один розряд праворуч, тобто в результаті виходить 1000 або число 8 у десятковому поданні.

Таким чином, якщо вихідне число, яке треба зрушити в ту чи іншу строну, ділиться на два, то фактично виходить множення чи поділ на два. Тому подібну операцію можна використовувати замість безпосереднього множення чи поділу на два.

2.4.4. Операції порівняння

У операціях порівняння порівнюються два операнда і повертається значення типу bool - true, якщо порівняння вірне, і false - інакше.

- **==**
  порівнює два операнди на рівність: z = x==y; z дорівнює true, якщо x дорівнює y, інакше z дорівнюватиме false.
- **!=**
  `z = x! = y; z дорівнює true, якщо x не дорівнює y, інакше z дорівнюватиме false.`
- **<**
  `z = x<y; z дорівнює true, якщо x менше y, інакше z дорівнюватиме false.`
- **>**
  `z = x> y; z дорівнює true, якщо x більше y, інакше z дорівнюватиме false.`
- **<=**
  `z = x<=y; z дорівнює true, якщо x менше або дорівнює y, інакше z дорівнюватиме false.`
- **>=**
  z = x>=y; z дорівнює true, якщо x більше або дорівнює y, інакше z дорівнюватиме false.

Також у C# визначено логічні оператори, які повертають значення типу bool. Частина їх було розглянуто вище у тих порозрядних операцій над числами.

Нижче розглядаються оператори під час операцій над булевими значеннями:

- **|**

z = x | y; z дорівнює true, якщо x, або y (або x, y y) рівні true, інакше z буде дорівнює false.

- **&**

  z = x&y; z дорівнює true, якщо і x, і y дорівнюють true, інакше z дорівнюватиме false.

- **!**

  z =! Y; z дорівнює true, якщо y дорівнює false, інакше z дорівнюватиме false.

- **^**

  z = x^y; z дорівнює true, якщо або x, або y (але не одночасно) дорівнюють true, інакше z дорівнюватиме false.

- **||**

  z = x||y; z дорівнює true, якщо x, або y (або x, y y) рівні true, інакше z буде дорівнює false.

- **&&**

  z = x&&y; z дорівнює true, якщо і x, і y дорівнюють true, інакше z дорівнюватиме false.

Тут дві пари операцій та || (а також & і &&) виконують схожі дії, однак вони не рівнозначні.

У виразі z = x | y; обчислюватимуться обидва значення - x і y.

У виразі ж z = x | | y; спочатку буде обчислюватися значення x, і якщо воно дорівнює true, то обчислення значення y вже не має сенсу, так як у нас у будь-якому випадку вже z буде одно true. Значення y буде обчислюватися тільки в тому випадку, якщо x дорівнює false

Те саме стосується пари операцій &/&&. У виразі z=x&y; обчислюватимуться обидва значення - x і y.

У виразі ж z=x&&y; спочатку буде обчислюватися значення x, і якщо воно дорівнює false, то обчислення значення y вже не має сенсу, так як у нас у будь-якому випадку вже z буде дорівнює false. Значення y буде обчислюватися тільки в тому випадку, якщо x дорівнює true.

Тому операції || і && більш зручні у обчисленнях, оскільки дозволяють скоротити час на обчислення значення виразу, і тим самим підвищують продуктивність. А операції | & більше підходять для виконання порозрядних операцій над числами.

Приклади:

```
bool x1 = (5 > 6) | (4 < 6);
// 5 > 6 – false, 4 < 6 – true, тому повертається true

bool x2 = (5> 6) | (4> 6);
// 5 > 6 - false, 4 > 6 - false, тому повертається //
false
```

```
boolx3 = (5 > 6) && (4 < 6);
// 5 > 6 - false, 4 < 6 - true, тому повертається //
false

boolx4 = (50 > 6) && (4/2 < 3);
// 50 > 6 - true, 4/2 < 3 - true, тому повертається //
true

boolx5 = (5 > 6) ^ (4 < 6);
// 5 > 6 - false, 4 < 6 - true, тому повертається // true

boolx6 = (50>6)^(4/2<3);
// 50 > 6 - true, 4/2 < 3 - true, тому повертається
// false
```

2.4.5. Операції присвоєння

- **=**
  найпоширеніша операція просто прирівнює одне значення іншому: z = x;
- **+=**
  z+=y; змінної z надається результат складання z та y
- **-=**
  z-=y; змінної z надається результат віднімання y з z
- ***=**
  z * = y; змінної z надається результат твору z та y
- **/=**
  z/=y; змінної z надається результат поділу z на y
- **%=**
  z%=y; змінної z надається залишок від розподілу z на y
- **&=**
  z&=y; змінної z надається значення z&y
- **|=**
  z|=y; змінної z надається значення z|y
- **^=**
  z^=y; змінної z надається значення z^y
- **<<=**
  z<<=y; змінної z надається значення z<<y
- **>>=**
  z>>=y; змінної z надається значення z>>y

2.5. Масиви

Масив представляє набір однотипних змінних. Оголошення масиву схоже на оголошення змінної:*тип_змінної[] назва_масиву*. Наприклад,

```
int[]nums = New int [4];
nums [0] = 1;
nums[1] = 2;
nums[2] = 3;
nums[3] = 5;
Console.WriteLine(nums[3]);
```

Тут спочатку оголосили масив nums, який зберігатиме дані типу int. Далі використовуючи операцію new, виділили пам'ять 4 елементів масиву: new int[4]. Число 4 ще називається довжиною масиву.

Відлік елементів масиву починається з 0, тому в даному випадку, щоб звернутися до четвертого елемента в масиві, треба використовувати nums [3].

І оскільки масив визначено лише 4 елементів, то не можна звернутися, наприклад, до шостому елементу: nums[5] = 5;. Якщо так зробити, то вийде виняток IndexOutOfRangeException.

У попередньому прикладі спочатку створили масив, а потім визначили для всіх елементів значення. Передбачено альтернативний спосіб ініціалізації масивів:

```
// ці два способи рівноцінні
int[]nums2 = new int[] {1, 2, 3, 5};

int[]nums3 = {1, 2, 3, 5};
```

Тут відразу вказую всі елементи масиву, при цьому довжина масиву обчислюється автоматично.

2.5.1. Багатовимірні масиви

Масиви бувають одновимірними та багатовимірними. У попередніх прикладах створювали одновимірні масиви. Створимо двомірний масив:

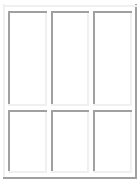
```
int[]nums1 = new int[] {0, 1, 2, 3, 4, 5};

int[,] nums2 = { { 0, 1, 2 }, { 3, 4, 5 } };
```

Візуально обидва масиви можна представити так:

Одновимірний масив nums1

0	1	2	3	4	5

Двовимірний масив nums2



Оскільки масив nums2 двовимірний, він є простою таблицею. Оголошення тривимірного масиву могло б виглядати так:

```
int[,,]nums3 = new int[2, 3, 4];
```

2.5.2. Масив масивів

Від багатовимірних масивів треба відрізняти масив масивів:

```
int[][]nums = new int[3][];
nums[0] = new int[2];
nums[1] = new int[3];
nums[2] = new int[5];
```

Тут дві групи квадратних дужок вказують, що це масив масивів, тобто такий масив, який у свою чергу містить інші масиви. В даному випадку масив nums містить три масиви. Причому розмірність кожного з цих масивів може збігатися.

Можна використовувати як масиви і багатовимірні масиви:

```
int[][,] nums = new int[3][,] {
new int[,] { {1,2}, {3,4} },
new int[,] { {1,2}, {3,6} },
new int[,] {{1,2}, {3,5}, {8, 13}}
};
```

Так тут у нас масив із трьох масивів, причому кожен із цих масивів представляє двомірний масив.

2.5.3. Деякі методи та властивості масивів

- Властивість Length: дозволяє отримати кількість елементів масиву
- Властивість Rank: дозволяє отримати розмірність масиву
- Метод Array.Reverse: змінює порядок проходження елементів масиву на зворотний

Метод Array.Sort: сортує елементи масиву.

Приклади використання:

```
int[]nums1 = new int [] {8, 1, 5, 3, 4, 2};
intlength = nums1.Length;
Console.WriteLine("кількість елементів: {0}", length);
intrank = nums1.Rank;
Console.WriteLine("розмірність масиву: {0}", rank);
Array.Reverse(nums1);
Array.Sort(nums1);
Console.WriteLine(nums1[1]);
```

2.5.4. Простий приклад використання масиву

```
// Оголошуємо двовимірний масив
int[,] myArr = new int [4, 5];
Random ran = новий Random();
// Ініціалізуємо та роздруковуємо даний масив
for(int i = 0; i < 4; i++) {
    for (int j = 0; j < 5; j++) {
       myArr[i, j] = ran.Next(1, 15);
       Console.Write("{0}\t", myArr[i, j]);
    }
```

```
    Console.WriteLine();
}
```

Зверніть особливу увагу на спосіб оголошення та використання двовимірного масиву. Якщо вам доводилося раніше програмувати на C, C++ чи Java, будьте особливо уважні, оголошуючи чи організуючи доступом до багатовимірним масивів в C#. У цих мовах програмування розміри масиву та індекси вказуються в окремих квадратних дужках, тоді як C# вони вказуються в «одних» квадратних дужках, розділені комою.

```
   int[] nums1 = new int[] { 0, 10, 20, 30, 40, 50};
   int[,] nums2 = new int[2,3] { { 11, 12, 13 }, { 21,
22, 23 } };
       for(int k=0; k<nums1.Length; k++)
            Console.WriteLine("{0} {1}", k, nums1[k]);
       Console.WriteLine("**********************");
       for(int i=0; i<2; i++) {
            for(int j=0; j<3; j++)
                 Console.Write("{0}\t", nums2[i,j]);
            Console.Write("\n");
       }
```

2.6. Умовні конструкції та оператор goto

Умовні конструкції - один із базових компонентів багатьох мов програмування. Вони «розгалужують» роботу програми одним із шляхів залежно від певних умов.

У мові C# використовуються такі умовні конструкції: if..else та switch..case

2.6.1. Конструкція if/else

Конструкція if/else перевіряє істинність певної умови та залежно від результатів перевірки виконує певний код:

```
intnum1 = 8; int num2 = 6;
if(num1 > num2) {
Console.WriteLine("Число {0} більше числа {1}", num1,
num2);
}
```

Після ключового слова if записується умова. Якщо ця умова виконується, то виконується код, записаний далі після фігурних дужок. Як умови виступають операції порівняння та операції з логічними змінними та виразами.

У прикладі перше число більше другого, тому вираз num1 > num2 істинно і повертає значення true, отже, управління переходить до рядка

```
   Console.WriteLine ("Число {0} більше числа {1}", num1,
num2);
```

Якщо необхідно, щоб при недотриманні умови виконувались будь-які дії, достатньо додати блок else:

```
intnum1 = 8;
```

```
intnum2 = 6;
if(num1 > num2) {
Console.WriteLine("Число {0} більше числа {1}", num1,
num2);
} else {
Console.WriteLine("Число {0} менше числа {1}", num1,
num2);
}
```

Допустимою також є конструкція else if:

```
   intnum1 = 8;
   intnum2 = 6;
   if(num1 > num2) {
   Console.WriteLine("Число {0} більше числа {1}", num1,
num2);
   }
   else if(num1 <num2) {
   Console.WriteLine("Число {0} менше числа {1}", num1,
num2);
        } else {
   Console.WriteLine("Число num1 дорівнює числу num2");
   }
```

Використовуючи оператори логічними значеннями, можна записувати «складні» умови:

```
intnum1 = 8;
intnum2 = 6;
if(num1 > num2 && num1>8) {
Console.WriteLine("Число {0} більше числа {1}", num1,
num2);
}
```

У разі блок if буде виконуватися, якщо num1 > num2 дорівнює true і num1>8 дорівнює true.

2.6.2. Конструкція switch

Конструкція switch/case аналогічна конструкції if/else, оскільки дозволяє обробити відразу кілька умов:

```
Console.WriteLine("Натисніть Y або N");
stringselection = Console.ReadLine();
switch(Selection) {
case "Y":
Console.WriteLine("Ви натиснули букву Y");
break;
case "N":
Console.WriteLine("Ви натиснули букву N");
break;
```

```
default:
Console.WriteLine("Ви натиснули невідому літеру");
break;
}
```

Після ключового слова switch у дужках записується вираз, значення якого підлягає порівнянню. Значення цього виразу послідовно порівнюється зі значеннями, розміщеними після операторів case. І якщо збіг буде знайдено, то виконуватиметься певний блок case.

Якщо необхідно, в кінці блоку case ставиться оператор break, що дозволяє уникнути перевірку та виконання інших блоків case.

Якщо необхідно обробити ситуацію «збіги не знайдені», необхідно додати блок default, як у вище наведеному прикладі.

2.6.3. Тернарна операція

Тернарна операція має наступний синтаксис: [перший операнд - умова]? [другий операнд] : [третій операнд]

Тут присуджують три операнди. Залежно від умови, тернарна операція повертає другий або третій операнд. Якщо умова true, то повертається другий операнд; якщо умова false, то третя. Наприклад:

```
intx=3;
inty=2;
Console.WriteLine("Натисніть + або -");
stringselection = Console.ReadLine();

intz = selection=="+"? (x+y): (xy);
Console.WriteLine(z);
```

Тут результатом тернарної операції є змінна z. Якщо вводимо "+", то z дорівнюватиме другому операнду - (x + y). Інакше z дорівнюватиме третьому операнду.

2.6.4 Оператор goto – безумовний перехід

Наявний C# оператор goto є класичний оператор безумовного переходу. Коли в програмі зустрічається оператор goto, то "управління передається" безпосередньо до того місця, на яке вказує цей оператор. Він уже давно «вийшов із вжитку» у програмуванні, оскільки сприяє створенню так званого «макаронного» коду. Хоча в деяких випадках він виявляється зручним та дає певні переваги. Головний недолік оператора goto, з погляду проектування та програмування, полягає в тому, що він вносить у програму безладдя і робить її незручною. Але іноді застосування оператора goto може швидше прояснити, ніж заплутати хід виконання програми.

Для виконання оператора goto потрібна мітка - дійсний C# ідентифікатор з двокрапкою. Мітка повинна знаходитися в тому ж методі, де і оператор goto, а також у межах тієї ж області дії.

Приклад використання оператора goto:

```
usingSystem;
```

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace ConsoleApplication1 {
    class Program {
        static void Main(string[] args) {
           // Звичайний цикл for вивідний числа від 1 до
5
           Console.WriteLine("Звичайний цикл for:");
           for (int i = 1; i <= 5; i++)
                Console.Write("\t{0}",i);
          // Реалізуємо те саме за допомогою оператора
// goto
          Console.WriteLine("\n\nA тепер використовуємо
"+" goto:");
            int j = 1;
        link1:
            Console.Write("\t{0}",j);
            j++;
            if (j <= 5) goto link1;
            Console.ReadLine();
        }
    }
}
```

«Репутація» оператора goto така, що, як правило, його застосування засуджується. Насправді цей оператор зручно застосовувати для «виходу» з глибоко вкладених внутрішніх циклів.

2.7. Цикли

Цикли також є керуючими конструкціями, дозволяючи в залежності від певних умов виконувати деяку дію багато разів. У C# є такі види циклів:

- **for**
- **foreach**
- **while**
- **do...while**

2.7.1. Цикл for

Цикл for має таке формальне визначення:

```
for([ініціалізація лічильника]; [умова]; [зміна
лічильника]) {
// дії
}
```

Розглянемо стандартний цикл**for**:

```
for(int i = 0; i < 9; i++) {
Console.WriteLine("Квадрат числа {0} дорівнює {1}", i, i
* i);
}
```

Перша частина оголошення циклу – int i = 0 – створює та ініціалізує лічильник i. Лічильник необов'язково має представляти тип int. Це може бути інший числовий тип, наприклад, float. У прикладі - перед виконанням циклу значення лічильника дорівнюватиме 0. В даному випадку це оголошення змінної і для контексту - тіло циклу.

Друга частина - умова, за якої виконуватиметься цикл. У разі цикл буде виконуватися, поки і досягне значення 9.

І третина - збільшення лічильника на одиницю. Знову ж таки необов'язково збільшувати на одиницю. Можна, наприклад, зменшувати: i--.

У результаті блок циклу виконається 9 разів (поки значення і стане рівним 9).

Необов'язково вказувати всі частини під час оголошення циклу. Наприклад, можна написати так:

```
inti = 0;
for(; ;) {
Console.WriteLine("Квадрат числа {0} дорівнює {1}", ++i,
i * i);
System.Threading.Thread.Sleep(500);
}
```

Формально визначення циклу залишилося тим самим, тільки тепер частини (блоки) у визначенні порожні: for(;;). Немає ініціалізованої змінної-лічильника, немає умови, тому цикл працюватиме вічно – це «нескінченний цикл».

Можна опустити не всі блоки:

```
inti = 0;
for(; i<9;) {
Console.WriteLine("Квадрат числа {0} дорівнює {1}", ++i,
i * i);
}
```

Цей приклад, по суті, еквівалентний першому прикладу: є лічильник i, тільки створений він поза циклом. Є умова виконання циклу. І є збільшення лічильника вже в самому блоці**for**. Пременная i в цьому випадку продовжує «існувати» і після закінчення роботи циклу.

2.7.2. Цикл foreach

Цикл foreach призначений для перебору елементів у контейнерах. Формальне оголошення циклу foreach:

```
foreach(тип_даних назва_змінної in контейнер) {
// дії
}
```

Наприклад:

```
int[]array = new int[] { 1, 2, 3, 4, 5 };
```

```
foreach(int i in array) {
Console.WriteLine(i);
}
```

Тут як контейнер виступає масив даних типу int. Тому оголошуємо змінну типу i int.

Подібні дії можна зробити і за допомогою циклу for:

```
int[]array = new int[] { 1, 2, 3, 4, 5 };
for(int i = 0; i < array.Length; i++) {
Console.WriteLine(array[i]);
}
```

У той же час цикл for більш гнучкий порівняно з foreach. Якщо foreach послідовно витягує елементи контейнераі лише для читання, то в циклі for ми можемо перескакувати на кілька елементів вперед залежно від збільшення лічильника, а також можемо змінювати елементи:

```
int[]array = new int[] { 1, 2, 3, 4, 5 };
for(int i = 0; i < array.Length; i++) {
array[i] = array[i] * 2;
Console.WriteLine(array[i]);
}
```

2.7.3. Цикл do

У циклі do спочатку виконується код циклу, потім відбувається перевірка умови в інструкції while. І поки ця умова є істинною, цикл повторюється. Наприклад:

```
inti = 6;
do {
Console.WriteLine(i);
i--;
}while (i > 0);
```

Тут код циклу (тіло циклу) спрацює 6 разів, доки і не стане рівним нулю. Але важливо зазначити, що цикл**do**гарантує хоча б одноразове виконання дій тіла циклу, навіть якщо умова в інструкції**while**не буде істинно. Тобто якщо

```
inti = -1;
do {
Console.WriteLine(i);
i--;
}while (i > 0);
```

то тіло циклу однаково один раз виконається.

2.7.3. Цикл while

На відміну від циклу do цикл while відразу перевіряє істинність деякої умови, і якщо умова істинна, то код циклу виконується:

```
inti = 6;
while(i > 0) {
```

```
Console.WriteLine(i);
i--;
}
```

2.7.4. Оператори continue та break

Іноді виникає ситуація, коли потрібно вийти з циклу, не чекаючи на його завершення. У цьому випадку можна скористатись оператором break.

Наприклад:

```
int[]array = new int[] {1, 2, 3, 4, 12, 9};
for(int i = 0; i < array.Length; i++) {
if (array[i] > 10) break;
Console.WriteLine(array[i]);
}
```

У тілі циклу виконується перевірка «чи більше і елемент масиву значення 10». У зв'язку з цим, на консоль не виведуть значення останніх двох елементів масиву. Для array[4] умова array[4] > 10 виконується тому буде виконано оператор break. Цикл завершиться "достроково".

Якщо необхідно, щоб за певної умови цикл не завершувався, а просто переходив до наступного елементу (перепустка ітерації), то можна скористатися оператором continue:

```
int[]array = new int[] {1, 2, 3, 4, 12, 9};
for(int i = 0; i < array.Length; i++) {
if (array[i] > 10) continue;
Console.WriteLine(array[i]);
}
```

У цьому прикладі число 12 (значення array[4] рівне 12) не буде надруковано.

2.8. Приклад програми сортування масиву

Познайомившись із циклами, змінними, умовними конструкціями та масивами, вже можна писати примітивні програми. Розглянемо програму сортування масиву, що реалізує алгоритм сортування «бульбашкою».

Створимо новий консольний додаток. І змінимо код файлу Program.cs на наступний:

```
using System;
namespace SortApp {
class Program {
static void Main(string[] args) {
            // Введення чисел
int[] nums = new int[7];
Console.WriteLine("Введіть сім чисел");
for (int i = 0; i < nums.Length; i++) {
Console.Write("{0}-е число: ", i + 1);
               nums[i] = Int32.Parse(Console.ReadLine());
}
```

```
            // сортування
int temp;
for (int i = 0; i < nums.Length-1; i++) {
for (int j = i + 1; j <nums.Length; j++) {
if (nums[i] > nums[j]) {
temp = nums [i];
nums[i] = nums[j];
nums[j] = temp;
}
}
}

            // Висновок
Console.WriteLine("Виведення відсортованого "+"масиву");
for (int i = 0; i < nums.Length; i++) {
Console.WriteLine(nums[i]);
}
Console.ReadLine();
}
}
```

Вся програма умовно поділена на три блоки: введення чисел, сортування та виведення відсортованого масиву. Спочатку в циклі вводимо всі числа в масив. У зв'язку з тим, що метод Console.ReadLine() повертає текстовий рядок, а масив необхідно поміщати цілі числа, необхідно «текстовий запис числа» перевести «ціле число» з допомогою методу Int32.Parse(Console.ReadLine()).

Потім масив сортуємо методом бульбашки: виконуємо проходи масивом і порівнюючи сусідні елементи. Якщо елемент з меншим індексом більший за елемент з великим індексом, то міняємо елементи місцями.

Наприкінці виводимо всі елементи.

2.9. Методи

Якщо змінні зберігають деякі значення, то методи містять набір операторів, які виконують певні дії.

Загальне визначення методів виглядає так:

```
[модифікатори] тип_повертаного_значення назва_метода
([параметри]) {
// Тіло методу
}
```

Модифікатори та параметри необов'язкові.

Розглянемо з прикладу методу Main:

```
static voidMain(string[] args) {
Console.WriteLine("привіт світ!");
}
```

Ключове слово "static" є модифікатором. Далі йде тип значення, що повертається. У разі ключове слово void вказує на те, що метод нічого не повертає. Такий метод ще називається процедурою.

Далі йде назва методу – Main та у дужках параметри – string[] args. І в фігурні дужки укладено тіло методу – всі дії, які він виконує.

Приклад деяких процедур:

```
static voidMethod1() {
Console.WriteLine("Method1");
}

voidMethod2() {
Console.WriteLine("Method2");
}
```

Модифікатор static дозволяє використовувати метод, не створюючи об'єкт класу, в якому визначено цей метод. Фактично це зазвичай означає, що static метод "не використовує" змінні "своїста класу".

2.9.1. Функції

На відміну від процедур, функції повертають певне значення. Наприклад, визначимо пару функцій:

```
intFactorial() {
return 1;
}

stringHello() {
return "Hell to World";
}
```

У функції в якості типу значення, що повертається замість void використовується будь-який інший тип. У разі типи int і string. Функції також відрізняються тим, що обов'язково повинен бути використаний оператор return, після якого записується значення, що повертається функцією.

Значення, що повертається, завжди повинно мати той же тип, що записаний в «заголовку» визначення функції.

2.9.2. Використання методів у програмі

Коли методи визначені, їх можна використовувати у програмі. Щоб викликати метод у програмі, треба вказати ім'я методу, а після нього у дужках значення його фактичних параметрів:

```
static voidMain(string[] args) {
string message = Hello(); // Виклик першого методу

Console.WriteLine(message);

    Sum(); // Виклик другого методу

Console.ReadLine();
```

```
}
static stringHello() {
return "Hell to World!";
}
static voidSum() {
int x = 2;
int y = 3;
Console.WriteLine("{0} + {1} = {2}", x, y, x+y);
}
```

Тут визначено два методи. Перший метод Hello повертає значення типу string. Тому можна привласнити це значення будь-якої змінної типу string: string message = Hello();

Другий метод – процедура Sum – просто складає два числа і виводить результат на консоль.

2.10. Параметри методів

У попередніх прикладах використовувалися методи без параметрів. Розглянемо, як використовуються параметри методів. Існує два способи передачі параметрів метод у мові C#: за значенням і за посиланням.

Найбільш простий спосіб передачі параметрів представляє передача за значенням:

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

При виклику цього в програмі обов'язково треба передати місце параметрів значення, які відповідають типу параметра:

```
static voidMain(string[] args) {
int x = 10;
int z = Sum (x, 15);
Console.WriteLine(z);
Console.ReadLine();
}
```

2.10.1. Модифікатор out

Вище використовувалися вхідні параметри. Але параметри можуть бути вихідними. Щоб зробити вихідним параметром, перед ним ставиться модифікатор out:

```
static voidSum(int x, int y, out int a) {
a = x + y;
}
```

На відміну від попереднього варіанту, тут результат повертається не оператором return, а через вихідний параметр. Приклад використання у програмі:

```
static voidMain(string[] args) {
```

```
int x = 10;
int z;
    Sum(x, 15, out z);
Console.WriteLine(z);
Console.ReadLine();
}
```

*Зверніть увагу, що модифікатор out вказується як при оголошенні методу, так і при його виклику*у методі Main.

Також зверніть увагу, що методи, які використовують такі параметри, обов'язково повинні надавати їм певне значення. Тобто наступний код буде неприпустимим, тому що в ньому для out-параметра не вказано жодного значення:

```
static voidSum(int x, int y, out int a) {
Console.WriteLine(x+y);
}// це приклад помилкового коду
```

Практика використання таких параметрів полягає в тому, що по суті можна повернути з методу не одне розраховане значення, а кілька. Наприклад:

```
static voidMain(string[] args) {
int x = 10;
int area;
int perimetr;
    GetData(x, 15, out area, out perimetr);
Console.WriteLine("Площа:" + area);
Console.WriteLine("Периметр:" + perimetr);

Console.ReadLine();
}
static voidGetData(int x, int y, out int area, out int
perim) {
area = x * y;
perim = (x + y) * 2;
}
```

Тут метод GetData приймає сторони прямокутника. Два вихідні параметри використовуються для отримання значень площі та периметра прямокутника в основній програмі.

2.10.2. Передача параметрів за посиланням та модифікатор ref

При надсиланні параметрів за посиланням перед параметрами використовується модифікатор ref:

```
static voidMain(string[] args) {
int x = 10;
int y = 15;
Addition(ref x, y); // виклик методу
Console.WriteLine(x);
```

```
Console.ReadLine();
}
// Визначення методу
static voidAddition(ref int x, int y) {
x += y;
}
```

Причому, як і у випадку з out, ключове слово ref використовується як при визначенні методу, так і при його виклику.

У чому різниця двох способів передачі параметрів? При передачі значення метод отримує не саму змінну, а її копію. А при передачі параметра посилання метод отримує адресу змінної в пам'яті. І, таким чином, якщо в методі змінюється значення параметра, що передається за посиланням, то також змінюється значення змінної, яка передається на його місце. Наприклад:

```
static voidMain(string[] args) {
//Початкові значення змінних a та b
int a = 5;
int b = 6;
Console.WriteLine("Початкове значення змінної" + "a =
{0}", a);
//Передача змінних за значенням
//Після виконання цього коду як і a = 5, // оскільки
передали лише її копію
AdditionVal(a, b);
Console.WriteLine("Змінна a після передачі по" +
                                "значення дорівнює = {0}",
a);

//Передача змінних за посиланням
    //Після виконання цього коду a = 11, //оскільки
передали саму змінну
AdditionRef(refa, b);
Console.WriteLine("Змінна a за посиланням по " +
"значення дорівнює = {0}", a);
Console.ReadLine();
}
// передача за посиланням
static voidAdditionRef(ref int x, int y) {
x = x + y;
    Console.WriteLine("x + y = {0}", x);
}
// передача за значенням
static voidAdditionVal(int x, int y) {
x = x + y;
    Console.WriteLine("x + y = {0}", x);
}
```

Тут у методі Main у нас є дві змінні: a та b. Є два методи, які приймають два параметри: x та y. В обох методах значення параметра x дорівнює сумі x та y.

У методі Main підставляються місце параметрів x і y змінні a і b відповідно. У першому випадку змінна передається за значенням, тобто передається копія цієї змінної, і вона змінюється. У другому випадку передається покажчик на цю змінну пам'яті. І так як в методі AdditionRef значення параметра x змінюється, то змінна, що передається на його місце, а теж змінює своє значення.

Коли треба передавати аргументи за посиланням, а коли за значенням? Якщо необхідно змінити змінну або навіть кілька змінних в одному методі, слід передавати аргументи за посиланням. Також слід передавати за посиланням великі об'єкти, навіть якщо не треба їх змінювати, оскільки створення їхньої копії знижує продуктивність програми.

2.10.3. Необов'язкові параметри

C# дозволяє використовувати необов'язкові параметри. Для таких параметрів необхідно вказувати значення за промовчанням. Також слід враховувати, що після першого необов'язкового параметра всі наступні параметри повинні бути необов'язковими:

```
static intOptionalParam(int x, int y, int z=5, int s=4) {
return x + y + z + s;
}
```

Так як останні два параметри оголошені як необов'язкові, то можливо при виклику методу один з них або обидва параметри не вказувати:

```
static  void Main(string[] args) {
OptionalParam(2, 3);
OptionalParam(2,3,10);
Console.ReadLine();
}
```

2.10.4. Іменовані параметри

У попередніх прикладах при викликі методів значення параметрів передавалися в порядку оголошення цих параметрів у методі. Але можна порушити такий порядок, використовуючи іменовані параметри:

```
static intOptionalParam(int x, int y, int z=5, int s=4)
{
return x + y + z + s;
}
static voidMain(string[] args) {
OptionalParam(x:2, y:3);

//Необов'язковий параметр z використовує // значення за
замовчуванням
OptionalParam(y:2,x:3,s:10);

Console.ReadLine();
}
```

Як видно з прикладу, параметр, що передається в метод, можна вказати на будь-якому місці списку, але перед ним необхідно написати ім'я формального параметра методу і вказати символ : (двокрапка).

2.11. Масив параметрів та ключове слово params

У всіх попередніх прикладах під час виклику методів вказувалося постійне число параметрів (узгоджене з кодом методу). Але, використовуючи ключове слово params, можна передавати методу невизначену кількість параметрів:

```
static voidAddition(params int[] integers) {
int result = 0;
for (int i = 0; i < integers.Length; i++) {
        result += integers[i];
}
Console.WriteLine(result);
}

static voidMain(string[] args) {
Addition(1, 2, 3, 4, 5);

int[] array = new int[] { 1, 2, 3, 4 };
Addition(array);

Addition();
Console.ReadLine();
}
```

Як видно з прикладу, місце параметра з модифікатором params можна передати як окремі значення, так і масив значень, або взагалі не передавати параметри. Код методу, використовуючи «властивість». Length з'ясовує кількість переданих йому параметрів.

Однак, цей спосіб має обмеження: після параметра з модифікатором params не можна вказувати інші параметри. Тобто таке визначення методу неприпустимо:

```
// це помилка
static voidAddition(params int[] integers, int x,
string mes) {
}
```

Цей спосіб передачі параметрів треба відрізняти від передачі масиву як параметр:

```
// передача параметра з params
static voidAddition(params int[] integers) {
int result = 0;
for (int i = 0; i < integers.Length; i++) {
        result += integers[i];
}
```

```
Console.WriteLine(result);
}
// передача масиву
static voidAdditionMas(int[] integers, int k) {
int result = 0;
for (int i = 0; i < integers.Length; i++) {
result += (integers[i]*k);
}
Console.WriteLine(result);
}

static voidMain(string[] args) {
Addition(1, 2, 3, 4, 5);

int[] array = new int[] { 1, 2, 3, 4 };
AdditionMas(array, 2);

Console.ReadLine();
}
```

Так як метод AdditionMas приймає як параметр масив без ключового слова params, то при його виклик нам обов'язково треба передати як параметр ім'я масив, а не список параметрів.

2.12. Рекурсивні функції

Рекурсивна функція представляє таку конструкцію, коли він функція викликає себе (явно чи опосередковано).

Традиційно, розглянемо функцію, яка обчислює факторіал цілого числа:

```
static intFactorial(int x) {
if (x <= 1) {
return 1;
} else {
return x * Factorial (x - 1);
}
}
```

Умова (x<=1) припиняє рекурсивний виклик. Інакше відбувається множення вхідного в функцію числа на результат виконання цієї функції, в яку в якості параметра передається число x-1. Тобто відбувається рекурсивний спуск. Так виконується до тих пір, поки значення вхідного параметра не дорівнює одиниці.

Обчислити факторіалу можна і без застосування рекурсії:

```
static intFactorial2(int x) {
int result=1;
for (int i = 1; i <= x; i++) {
result *= i;
```

```
}
return result;
}
```

Слід зазначити, що такі методи вичлікування факторіалів дадуть правильні результати лише з невеликих значень аргументу (x <= 12 для 32-х бітної архітектури).

```
for (int k=1; k<13; k++)
            Console.WriteLine(" "+k+" "+Factorial2(k) );


    static int Factorial2(int x) {
      int result=1;
      int kkk = 0;
      try {
         for (int i = 1; i <= x; i++) {
            kkk=i;
            result = checked (result * i);
         }
      } catch (OverflowException ex) {
          Console.WriteLine("kkk="+kkk+" "+ex.Message);
      }
      return result;
    }
```

```
РЕЗУЛЬТАТ:
1 1
2 2
3 6
4 24
5 120
6 720
7 5040
8 40320
9 362880
10 3628800
11 39916800
12 479001600
kkk=13 Переповнення внаслідок виконання арифметичної
операції.
13 479001600
```

Іншим поширеним показовим прикладом рекурсивної функції є функція, що обчислює числа Фіббоначчі. n-й член послідовності Фібоначчі визначається за формулою: f(n)=f(n-1) + f(n-2), причому f(0)=0, а f(1)=1.

```
static intFibonachi(int n) {
if (n == 0) {
return 0;
```

```
}
if (n == 1) {
return 1;
} else {
return Fibonachi(n - 1) + Fibonachi(n - 2);
}
}
```

2.13. Перерахування enum

Крім примітивних типів даних C# є такий тип як enum або перерахування. Перерахування представляють набір логічно пов'язаних констант. Оголошення переліку відбувається за допомогою оператора enum. Далі йде назва перерахування, після якого вказується тип перерахування - він обов'язково має представляти цілий тип (byte, int, short, long). Якщо тип явно не вказано, то використовується тип int. Потім йде список елементів перерахування через кому:

```
enumdays {
Monday,
Tuesday,
wednesday,
thursday,
friday,
saturday,
sunday
}

enumtime : byte {
morning,
afternoon,
evening,
night
}
```

У цих прикладах кожному елементу перерахування присвоюється ціле значення, причому перший елемент матиме значення 0, другий - 1 і так далі. Можна також явно вказати значення елементів, або вказавши значення першого елемента:

```
enumoperation {
add = 1, // кожен наступний елемент за замовчуванням //
збільшується на одиницю
subtract // цей елемент дорівнює 2
multiply, // дорівнює 3
divide // дорівнює 4
}
```

Можна для всіх елементів явно вказати їх значення:

```
enumoperation {
add = 2,
subtract = 4,
multiply = 8,
divide = 16
}
```

Приклад перерахування у реальній програмі:

```
classProgram {
enum Operation {
add = 1,
subtract,
multiply,
divide
}

static void MathOp (double x, double y, Operation op) {
double result = 0.0;

switch (op) {
case Operation.add:
result = x + y;
break;
case Operation.subtract:
result = x - y;
break;
case Operation.multiply:
result = x * y;
break;
case Operation.divide:
result = x/y;
break;
}

Console.WriteLine("Результат операції дорівнює {0}",
result);
}

static void Main(string[] args) {
// Тип операції задаємо за допомогою константи //
Operation.add, яка дорівнює 1
MathOp(10, 5, Operation.add);
// Тип операції задаємо за допомогою константи //
Operation.multiply, яка дорівнює 3
MathOp(11, 5, Operation.multiply);
```

```
Console.ReadLine();
}
}
```

У прикладі задано перелік Operation, який представляє арифметичні операції. Метод MathOp як параметри приймає два числа і типи операції. В основному методі Main двічі викликається процедура MathOp, передаючи в неї два числа та тип операції.

2.14. Структури

Крім базових елементарних типів даних і перерахувань C# є і складовий тип даних, який називається структурою. Структури можуть містити у собі звичайні змінні та методи.

Для прикладу створимо структуру Book, в якій зберігатимуться змінні для назви, автора та року видання книги. Крім того, структура міститиме метод для виведення інформації про книгу на консоль:

```
structBook {
public string name;
public string author;
public int year;

public void Info() {
Console.WriteLine("Книга '{0}' (автор {1}) була "+"
видана в {2} році", name, author, year);
}
}
```

Щоб можна було використовувати змінні та методи структури з будь-якого місця програми, ми ставимо перед змінними та методом модифікатор доступу.

Використовуємо структуру практично:

```
usingSystem;

namespaceStructures {
class Program {
static void Main(string[] args) {
Book book;
book.name = "Війна та мир";
book.author = "Л. Н. Толстой";
book.year = 1869;

//Виведемо інформацію про книгу book на екран
book.Info();

Console.ReadLine();
}
}
```

```
struct Book {
public string name;
public string author;
public int year;
public void Info() {
Console.WriteLine("Книга '{0}' (автор {1}) була"+ "
видана в {2} році", name, author, year);
}
}
}
```

Структуру можна задати як усередині простору імен (як у цьому випадку), так і всередині класу, але не всередині методу.

По суті структура Book представляє новий тип даних. Можна також використовувати масиви структур:

```
Book[] books=new Book[3];
books[0].name = "Війна та мир";
books[0].author = "Л. Н. Толстой";
books[0].year = 1869;

books[1].name = "Злочин і покарання";
books [1]. author = "Ф. М. Достоєвський";
books[1].year = 1866;

books[2].name = "Батьки та діти";
books [2]. author = "І. С. Тургенєв";
books[2].year = 1862;

foreach (Book b in books) {
b.Info();
}
```

2.14.1. Конструктори у структурах

Крім звичайних методів структура може містити спеціальний метод - *конструктор*, який виконує якусь початкову ініціалізацію об'єкта, наприклад, надає всім полям деякі значення за промовчанням. В принципі для структури необов'язково використовувати конструктор:

```
Book book1;
```

Однак у цьому випадку, щоб повноцінно використовувати структуру, обов'язково необхідно проініціалізувати усі її поля:

```
book1.name = "Війна та мир";
book1.author = "Л. Н. Толстой";
book1.year = 1869;
```

Виклик конструктора виконується автоматично при «створенні» об'єкта (змінної типу структури) і дозволяє автоматично проініціалізувати поля

структури значеннями за промовчанням (наприклад, для числових даних – це число 0):

```
Book book2 = New Book (); // Використання конструктора
```

Виклик конструктора має наступний синтаксис:

new назва_структури([список_параметрів]).

Визначимо «свій» конструктор:

```
structBook {
public string name;
public string author;
public int year;

    // конструкор
public Book(string n, string a, int y) {
name = n;
author = a;
year = y;
}
public void Info() {
Console.WriteLine("Книга '{0}' (автор {1}) була"+ "
видана в {2} році", name, author, year);
}
}
```

Конструктор є звичайним методом, тільки не має значення, що повертається, і його назва завжди збігається з ім'ям структури або класу.

Використання цього методу:

```
Book book = new Book ("Війна і мир", "Л. Н. Толстой",
1869);
book.Info();
```

Тепер не треба «вручну» надавати значення полям структури – їхню ініціалізацію виконав конструктор.

2.15. Обробка винятків

Іноді під час виконання програми виникають помилки, які важко передбачити чи передбачити. Наприклад, під час передачі файлу через мережу може несподівано обірватися мережне підключення. Такі ситуації називаються**винятками**. Мова C# надає розробникам можливості обробки таких ситуацій. Для цього C# призначена конструкція try...catch...finally. У разі виключення середовище CLR шукає блок catch, який може обробити цей виняток. Якщо такого блоку не знайдено, то користувачеві відображається повідомлення про необроблений виняток, а подальше виконання програми зупиняється. Щоб подібна аварійна зупинка програми не сталася, необхідно використовувати блок try..catch. Наприклад:

```
static voidMain(string[] args) {
int[] a = new int[4];
```

```
try {
a[5] = 4; // тут виникне виняток, // оскільки ми в масиві
лише 4 //елемента
Console.WriteLine("Завершення блоку try");
}
catch (Exception ex) {
Console.WriteLine("Помилка:" + ex.Message);
} finally {
Console.WriteLine("Блок finally");
}
Console.ReadLine();
}
```

При використанні блоку try...catch..finally спочатку виконуються всі інструкції між операторами try та catch. Якщо при цьому виникає виняток, то звичайний порядок виконання зупиняється та управління переходить до інструкцій catch. У цьому прикладі виникне виняток у блоці try, тому що відбувається спроба надати значення шостому елементу масиву a . Але в масиві всього 4 елементи. При виконанні операції a[5] = 4; виконання блоку програми переривається, і управління передається до блоків (функцій) catch.

Інструкція catch має наступний синтаксис:

catch (тип_виключення ім'я_змінної) {}.

У прикладі, параметром, що передається функції, оголошується змінна ex, яка має тип Exception. Якщо виняток не є винятком типу, вказаного в інструкції catch, то воно не обробляється, і програма аварійно зачаровується.

Тип Exception є базовим класом всім типів винятків, то вираз catch (Exception ex) буде обробляти майже всі винятки. Вся обробка виключення у цьому прикладі зводиться до виведення на консоль повідомленні про виключення, яке можна отримати, використовуючи властивість message класу Exception.

Далі у будь-якому випадку виконуюсь оператори блоку finally. Однак цей блок необов'язковий і його можна опускати. Якщо під час програми винятків не виникне, то програма нічого очікувати виконувати блок catch, відразу перейде до блоку finally, якщо він є.

2.15.1. Обробка кількох винятків

Насправді часто розмежовують обробку різних типів винятків. Це здійснюється кодуванням додаткових блоків catch:

```
static voidMain(string[] args) {
try {

}
catch (FileNotFoundException e) {
        // Обробка винятку, що виник при
        // відсутність файлу
}
```

```
catch (IOException e) {
// Обробка винятків введення-виведення
}
Console.ReadLine();
}
```

Якщо виникає виняток певного типу, воно обробляється відповідним блоком catch.

При цьому більш приватні винятки слід поміщати на початку, і лише потім більш загальні винятки. Наприклад, спочатку обробляється виняток IOException, і лише потім Exception (оскільки IOException успадковується від класу Exception).

2.15.2. Оператор throw

Щоб "згенерувати" виняток використовується оператор throw. Тобто за допомогою цього оператора програміст може створювати (генерувати, викидати) виняток у процесі виконання програми.

Наприклад, програміст бажає, щоб у програмі при введенні рядка не допускалися рядки більше 6 символів. Якщо довжина рядка більше 6 символів – «викидається» виняток:

```
static voidMain(string[] args) {
try {
string message = Console.ReadLine();
if (message.Length > 6) {
throw new Exception( "Довжина рядка більше 6 символів");
}
}
catch (Exception e) {
Console.WriteLine("Помилка:" + e.Message);
}
Console.ReadLine();
}
```

2.15.3. Обробка винятків та умовні конструкції

Ряд виняткових ситуацій може бути передбачений програмістом. Наприклад, нехай програма передбачає введення числа та виведення його квадрата:

```
static voidMain(string[] args) {
Console.WriteLine("Введіть число");
int x = Int32.Parse(Console.ReadLine());
x * = x;
Console.WriteLine("Квадрат числа:" + x);
Console.Read();
}
```

Якщо користувач введе не числові символи, а рядок інших символів, програма завершиться аварійно. Тут можна застосувати блок try..catch, щоб

опрацювати можливу помилку. Але набагато оптимальніше було перевірити допустимість перетворення рядка на ціле число:

```
static voidMain(string[] args) {
Console.WriteLine("Введіть число");
int x;
string input = Console.ReadLine();
if (Int32.TryParse(input, out x)) {
x * = x;
Console.WriteLine("Квадрат числа:" + x);
} else {
Console.WriteLine("Некоректне введення");
}
Console.Read();
}
```

Метод Int32.TryParse() повертає true, якщо перетворення можна здійснити, і false - якщо не можна. При допустимості перетворення змінна x міститиме введене число. Це приклад як, не використовуючи try...catch, можна обробити можливу виняткову ситуацію.

З погляду продуктивності використання блоків try..catch менш ефективне, ніж застосування умовних конструкцій. Тому, по можливості, замість try..catch краще використовувати умовні конструкції на перевірку виняткових ситуацій.

2.15.4. Фільтри винятків when

C# 6.0 (Visual Studio 2015) була додана така функціональність, як фільтри винятків. Вони дозволяють обробляти винятки залежно від певних умов:

```
intx = 1;
inty = 0;

try {
int result = x/y;
    // Інші оператори
}catch(Exception ex) when (y==0) {
Console.WriteLine("y не повинен дорівнювати 0");
}
catch(Exception ex) {
Console.WriteLine(ex.Message);
}
```

У даному випадку буде викинуто виняток лише за умови умови: y==0. Тут два блоки catch, але оскільки першого блоку зазначено умова з допомогою ключового слова when, то спрацює перший блок catch. Якби y не було 0, то спрацював би другий блок catch у разі помилок в інших операторах.

2.16. Робота з консоллю та клас Console

Для взаємодії з консоллю зазвичай використовується клас Console. Раніше у прикладах використовувався виведення на консоль за допомогою методу WriteLine. Розглянемо цей клас докладніше.

Клас Console представляє ряд методів для взаємодії з консоллю:

- `Beep`: подача звукового сигналу
- `Clear`: очищення консолі
- `WriteLine`: виведення рядка тексту, включаючи символ повернення каретки (тобто переклад на новий рядок)
- `Write`: виведення рядка тексту, але без символу повернення каретки
- `ReadLine`: зчитування рядка тексту із вхідного потоку
- `Read`: зчитування введеного символу у вигляді числового коду цього символу. За допомогою перетворення до типу**char**можна отримати введений символ

`ReadKey`: зчитування клавіші клавіатури

ConsoleKeyInfo key= Console.ReadKey();

`Key`	Повертає клавішу консолі, представлену поточним об'єктом`ConsoleKeyInfo`.
`KeyChar`	Повертає символ Юнікоду, представлений поточним об'єктом`ConsoleKeyInfo`.
`Modifiers`	Повертає побітове поєднання значень із перерахування`System.ConsoleModifiers`, що вказує, чи одночасно з клавішею консолі натиснуті керуючі клавіші SHIFT, ALT або CTRL.

ConsoleKey – перелік визначає стандартні клавіші консолі (наведені не всі):

Ім'я члена	Опис
`Backspace`	Кнопка BACKSPACE.
`Tab`	Кнопка TAB.
`Enter`	Кнопка ВВЕДЕННЯ.
`Pause`	Клавіша PAUSE.
`Escape`	Кнопка ESC (ESCAPE).
`Spacebar`	Клавіша ПРОБІЛ.
`PageUp`	Клавіша PAGE UP.
`PageDown`	Клавіша PAGE DOWN.

Ім'я члена	Опис
End	Клавіша END.
Home	Клавіша HOME.
LeftArrow	Клавіша стрілка вліво.
UpArrow	Клавіша СТРІЛКА ВВЕРХ.
RightArrow	Клавіша стрілка вправо.
DownArrow	Клавіша Стрілка вниз.
PrintScreen	Клавіша PRINT SCREEN.
Insert	Кнопка INS (INSERT).
Delete	Кнопка DEL (DELETE).
Help	Кнопка HELP.
D0	Кнопка 0.
D1	Клавіша 1.
D2	Клавіша 2.
D3	Кнопка 3.
D4	Клавіша 4.
D5	Клавіша 5.
D6	Клавіша 6.
D7	Клавіша 7.
D8	Клавіша 8.
D9	Клавіша 9.
A	Кнопка A.
B	Кнопка B.
C	Клавіша C.
D	Кнопка D.
E	Кнопка E.
F	Кнопка F.
G	Клавіша G.
H	Кнопка H.
I	Клавіша I.
J	Клавіша J.
K	Клавіша K.
L	Клавіша L.
M	Кнопка M.
N	Кнопка N.
O	Кнопка O.

Ім'я члена	Опис
P	Клавіша P.
Q	Клавіша Q.
R	Клавіша R.
S	Клавіша S.
T	Кнопка T.
U	Кнопка U.
V	Клавіша V.
W	Клавіша W.
X	Кнопка X.
Y	Клавіша Y.
Z	Клавіша Z.
LeftWindows	Ліва клавіша з емблемою Windows
RightWindows	Права клавіша з емблемою Windows
Applications	Клавіша контекстного меню
Sleep	Клавіша переведення комп'ютера в режим сну.
NumPad0	Кнопка 0 на цифровій клавіатурі.
NumPad1	Клавіша 1 на цифровій клавіатурі.
NumPad2	Клавіша 2 на цифрову клавіатуру.
NumPad3	Кнопка 3 на цифровій клавіатурі.
NumPad4	Клавіша 4 на цифрову клавіатуру.
NumPad5	Клавіша 5 на цифрову клавіатуру.
NumPad6	Клавіша 6 на цифрову клавіатуру.
NumPad7	Клавіша 7 на цифрову клавіатуру.
NumPad8	Клавіша 8 на цифрову клавіатуру.
NumPad9	Клавіша 9 на цифрову клавіатуру.
Multiply	Клавіша знак множення.
Add	Клавіша плюс.
Separator	Клавіша роздільника.
Subtract	Кнопка мінус.
Decimal	Клавіша десяткового роздільника.
Divide	Клавіша знак розподілу.
F1	Кнопка F1.
F2	Кнопка F2.
F3	Кнопка F3.
F4	Кнопка F4.

Ім'я члена	Опис
F5	Кнопка F5.
F6	Кнопка F6.
F7	Кнопка F7.
F8	Кнопка F8.
F9	Кнопка F9.
F10	Кнопка F10.
F11	Кнопка F11.
F12	Кнопка F12.

Перелік ConsoleKey зазвичай використовується у структурі System.ConsoleKeyInfo, що повертається методомConsole.ReadKeyдля вказівки того, яку клавішу консолі було передано.

У цьому прикладі використовується перелік ConsoleKey, щоб показати користувачеві, яку клавішу він натиснув.

```
using System;
using System.Text;
public class ConsoleKeyExample {
   public static void Main() {
      ConsoleKeyInfo input;
      do {
         Console.WriteLine("Press a key, "+ " together
with Alt, Ctrl, або Shift.");
         Console.WriteLine("Press Esc to exit.");
         input = Console.ReadKey(true);

         StringBuilder output = новий StringBuilder(
                        String.Format("You pressed {0}",
input.Key.ToString()));
         bool modifiers = false;

         if ((input.Modifiers & ConsoleModifiers.Alt)
== ConsoleModifiers.Alt) {
            output.Append(", together with " +
ConsoleModifiers.Alt.ToString());
            modifiers = true;
         }
         if ((input.Modifiers &
ConsoleModifiers.Control) == ConsoleModifiers.Control){
            if (modifiers) {
               output.Append(" and ");
            } else {
```

```
                output.Append(", together with ");
                modifiers = true;
             } output.Append(
                   ConsoleModifiers.Control.ToString());
          }
          if ((input.Modifiers & ConsoleModifiers.Shift)
== ConsoleModifiers.Shift) {
             if (modifiers) {
                output.Append(" and ");
             }
             else {
                output.Append(", together with ");
                modifiers = true;
             }
             output.Append(
ConsoleModifiers.Shift.ToString());
          }
          output.Append(".");
          Console.WriteLine(output.ToString());
          Console.WriteLine();
       } while (input.Key! = ConsoleKey.Escape);
   }
}
```

Крім того, клас Console, має властивості, які дозволяють керувати консоллю. Деякі з них:

- BackgroundColor: колір фону консолі
- ForegroundColor: колір шрифту консолі
- BufferHeight: висота буфера консолі
- BufferWidth: ширина буфера консолі
- Title: заголовок консолі

WindowHeightта WindowWidth: висота та ширина консолі відповідно

Розглянемо невелику програму. Воно буде приймати два числа, введені користувачем, і відображатиме їхню суму. Додаток буде мати наступний код:

```
class Program {
static void Main(string[] args) {
        // встановлення зеленого кольору шрифту
Console.ForegroundColor=ConsoleColor.DarkGreen;

try {
do {
Console.WriteLine("Введіть перше число");
int num1 = Int32.Parse(Console.ReadLine());
Console.WriteLine("Введіть друге число");
int num2 = Int32.Parse(Console.ReadLine());
Console.WriteLine("Сума чисел {0} та {1}"+
```

```
                    " дорівнює {2}", num1, num2, num1 +
num2);
Console.WriteLine("Для виходу натисніть"+ " Escape; "+
                                        "для продовження "+" -
будь-яку іншу клавішу");
}
while (Console.ReadKey().Key != ConsoleKey.Escape);
}
catch (Exception ex) {
Console.WriteLine(ex.Message);
Console.ReadLine();
}
}
}
```

Спочатку встановлюємо колір шрифту консолі. Усі доступні кольори зберігаються у списку ConsoleColor. Далі поміщаємо весь код у блок try, оскільки може виникнути виняток у ході перетворення рядка в число (якщо буде виконано введення нечислових символів). Вихід із циклу do...while організований натисканням клавіші Escape (дуже погане рішення).

Члени переліку ConsoleColor:

Ім'я члена	Опис
Black	Чорний колір.
Blue	Синій колір.
Cyan	Блакитний колір (синьо-зелений).
DarkBlue	Темно-синій колір.
DarkCyan	Темно-блакитний колір (чорний синьо-зелений).
DarkGray	Темно-сірий колір.
DarkGreen	Темно-зелений колір.
DarkMagenta	Темно-пурпуровий колір (темний фіолетово-червоний).
DarkRed	Темно-червоний колір.
DarkYellow	Темно-жовтий колір (коричнево-жовтий).
Gray	Сірий колір.
Green	Зелений колір.
Magenta	Пурпуровий колір (фіолетово-червоний).
Red	Червоний колір.
White	Білий колір.
Yellow	Жовтий колір.

За допомогою методу Int32.Parse, введена сторінка перетворюється на число: int num1 = Int32.Parse(Console.ReadLine());

2.17. Типи значень та типи посилань

Раніше розглядалися такі елементарні типи даних: int, byte, double, string, object та інших. Мова C# так само підтримує складні типи: структури, перерахування, класи. Всі ці типи даних можна умовно розділити на «типи значень», ще звані значимими типами (**value types**) та «посилальні типи» (**reference types**).

Типи значень:

- Цілочисленні типи (byte, sbyte, char, short, ushort, int, uint, long, ulong)
- Типи з плаваючою комою (float, double)
- Тип decimal
- Тип bool
- Перерахування enum
- Структури (struct)
- **Посилальні типи:**
- Тип object
- Тип string
- Класи class
- Інтерфейси interface
- Делегати delegate

У чому між ними відмінності? Для цього треба зрозуміти організацію пам'яті у .NET. У .NET пам'ять ділиться на два типи: стек (stack) та купа (heap). Всі типи значень похідні від типу System.ValueType і розміщують своє значення в стеку. Стек являє собою структуру даних, яка «зростає знизу вгору»: кожен новий елемент, що додається, поміщаються поверх попереднього. Час життя таких змін обмежений їх контекстом. Фізично стек – це деяка область пам'яті в адресному просторі.

Коли програма запускається виконання, покажчик стека встановлюється те щоб він уазивал в кінець блоку пам'яті, зарезервованого для стека. При розміщенні даних у стек покажчик встановлюється таким чином, що знову вказує на нове вільне місце. Наприклад:

```
private voidSomeMethodVal(int t) {
int x = 5;
int y = 6;
int z = x * y * t;
}
```

При виклику цього методу в стек будуть розміщуватися змінні t, x, y та z. Вони визначаються у тих даного методу. Коли метод відпрацює, всі ці змінні знищуються і пам'ять очищається.

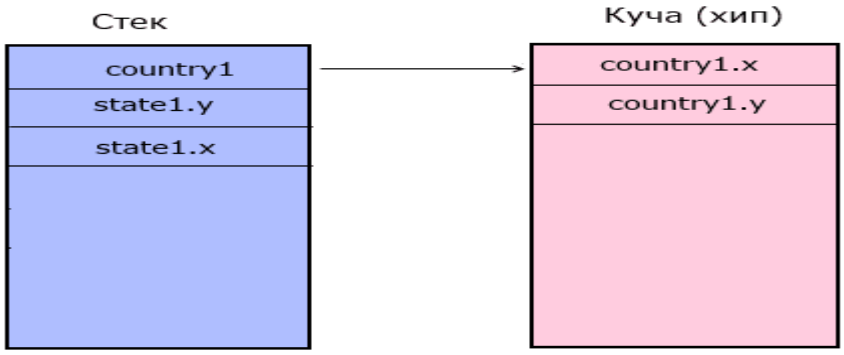
Типи посилань зберігаються в купі (або хіпі), яку можна представити як невпорядкований набір різнорідних об'єктів. Фізично це решта пам'яті, яка доступна процесу.

При створенні об'єкта посилання типу в стеку міститься посилання на адресу в купі (хіпі). Коли об'єкт посилання типу перестає використовуватися, посилання зі стека видаляється, і пам'ять очищається. Після цього в справу вступає автоматичний збирач сміття: він «бачить», що на об'єкт у хіпі немає більше посилань і видаляє цей об'єкт (тобто звільняє пам'ять для подальшого використання).

Розглянемо невеликий приклад, де буде використано тип значень у вигляді структури та тип посилань у вигляді класу.

```
classProgram {
private static void Main(string[] args) {
State state1 = New State();
// State - структура, її дані розміщені у стеку
Country country1 = new Country();
// Country - клас, в стек міститься посилання // адресу
в хіпі
// а в хіпі розташовуються всі дані об'єкта // country1
}
}
structState {
public int x;
public int y;
}
classCountry {
public int x;
public int y;
}
```

Тут у методі Main у стеку виділяється пам'ять для об'єкта state1. Далі в стеку створюється посилання об'єкта country1 (Country country1), а з допомогою конструктора виділяється місце у хіпі (new Country()). Посилання в стеку для об'єкта country1 представлятиме адресу на місце в хіпі, за яким розміщений даний об'єкт.



Таким чином, у стеку виявляться всі поля структури state1 та адреса на всі поля об'єкта country1 у хіпі.

2.17.1. Копіювання значень

Тип даних треба враховувати під час копіювання значень. При наданні даних об'єкту значного типу він отримує копію даних. При наданні даних об'єкту посилання типу відбувається присвоєння не копії об'єкта, яке адреси. Наприклад:

```
private static voidMain(string[] args) {
State state1 = New State(); // Структура State
    State state2 = New State();
state2.x = 1;
state2.y = 2;
state1 = state2;
    state2.x = 5; // state1.x=1, як і раніше
Console.WriteLine(state1.x); // 1
Console.WriteLine(state2.x); // 5

Country country1 = new Country(); // Клас Country
Country country2 = new Country();
country2.x = 1;
country2.y = 4;
country1 = country2;
    country2.x = 7;
// тепер і country1.x = 7, тому що обидві посилання // і
country1 і country2 вказують на один // об'єкт у хіпі
Console.WriteLine(country1.x); // 7
Console.WriteLine(country2.x); // 7

Console.Read();
}
```

Так як state1 – структура, то при присвоєнні state1 = state2 вона отримує копію структуриstate2. А об'єкт класу country1 при присвоєнні country1 = country2; отримує адресу посилання

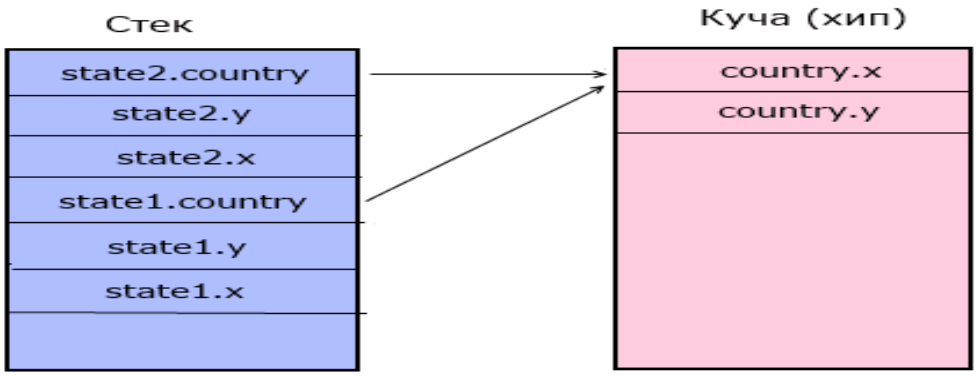
той самий об'єкт, який вказує country2. Тому зі зміною country2, так само змінюватиметься і country1.

2.17.2. Посилальні типи всередині типів значень

Розглянемо приклад, коли всередині структури є змінна посилання, наприклад, об'єкт якого-небудь класу:

```
classProgram {
private static void Main(string[] args) {
State state1 = New State();
      State state2 = New State();

state2.country = New Country();
state2.country.x = 5;
state1 = state2;
state2.country.x = 8;/* тепер і state1.country.x=8,
оскільки state1.country та state2.country вказують на
один об'єкт у хіпі */
Console.WriteLine(state1.country.x); // 8
Console.WriteLine(state2.country.x); // 8

Console.Read();
}
}
structState {
public int x;
public int y;
public Country country;
}
classCountry {
public int x;
public int y;
}
```

Змінні типів посилань у структурах також зберігають у стеку адресу посилання на об'єкт у хіпі. І за присвоєння двох структур state1 = state2; структураstate1 також отримає посилання на об'єкт country у хіпі. Тому зміна state2.country спричинить також зміну state1.country.



2.17.4. . Об'єкти класів як параметри методів

Організацію зберігання об'єктів у пам'яті слід враховувати під час передачі параметрів за значенням та за посиланням. Якщо параметри методів є об'єктами класів, то використання параметрів має деякі особливості. Наприклад, створимо метод, який як параметр приймає об'єкт Person:

```
classProgram {
static void Main(string[] args) {
Person p = new Person { name = " Tom " , age = 23 };
        ChangePerson(p);
Console.WriteLine(p.name); // Alice
Console.WriteLine(p.age); // 23
Console.Read();
}

static void ChangePerson(Person person) {
// "спрацює" - змінить поле вхідного об'єкта
person.name = "Alice";
        // "спрацює" тільки в рамках даного методу
person = new Person { name = " Bill " , age = 45 };
Console.WriteLine(person.name); // Bill
}
}
classPerson {
public string name;
public int age;
}
```

При передачі об'єкта класу за значенням, метод передається копія посилання на об'єкт. Ця копія вказує на той самий об'єкт, що й вихідне посилання, тому можна змінити окремі поля та властивості об'єкта, але не можемо змінити сам об'єкт (точніше посилання на об'єкт). Тому в прикладі вище "спрацює" лише рядок person.name = "Alice".

А інший рядок person = new Person { name = " Bill " , age = 45 } створить новий об'єкт у пам'яті, і person тепер вказуватиме новий об'єкт у пам'яті. Якщо

після цього його змінити, це ніяк не вплине на посилання p у методі Main, оскільки посилання p все ще вказує на старий об'єкт у пам'яті.

Але при передачі параметра за посиланням (за допомогою ключового слова ref) метод як аргумент передається сам адресу на об'єкт у пам'яті. Тому можна змінити як поля та властивості об'єкта, так і сам об'єкт:

```
classProgram {
static void Main(string[] args) {
Person p = new Person { name = " Tom " , age = 23 };
ChangePerson(ref p);
Console.WriteLine(p.name); // Bill
Console.WriteLine(p.age); // 45
Console.Read();
}
static void ChangePerson(ref Person person) {
// «спрацює»
person.name = "Alice";
// «спрацює»
        person = new Person { name = " Bill " , age = 45
};
}
}
classPerson {
public string name;
public int age;
}
```

Операціяnew створить новий об'єкт у пам'яті, і тепер посилання person (вона посилання p з методу Main) буде вказувати вже на новий об'єкт у пам'яті.

2.18. Контрольні питання

1. Основні типи даних, які у мові C#.
2. Що таке стандартний та дослівний рядковий літерал?
3. Як вставити в дослівний літерал подвійну лапку?
4. Що таке неявна типізація?
5. Що визначає «контекст методу»?
6. Призначення ключове слово checked.
7. У чому відмінність логічних операцій та || (А також & & &)?
8. Коли виникає виняток IndexOutOfRangeException?
9. Що таке масив масивів?
10. Чи є у складі мови C# оператор goto?
11. Що таке тернарна операція?
12. У чому принципова різниця циклів for і foreach?
13. Призначення операторів continue та break?

14. Призначення та використання модифікатора out?
15. Призначення та використання модифікатора ref?
16. Що таке необов'язкові параметри та іменовані параметри?
17. Призначення та використання ключового слова params?
18. Що таке рекурсивна функція?
19. Структури та їх застосування.
20. Конструктори у структурах.
21. Призначення операторів блоку finally під час обробки винятків.
22. Що вважається ефективнішим з погляду продуктивності програми використання операторів обробки винятків чи використання умовних конструкцій для обробки помилкових ситуацій?
23. Які типи даних відносяться до типів посилань?

3. Складання сміття, управління пам'яттю та вказівники

3.1. Складальник сміття в C#

Раніше у темі «2.17. Типи значень і типи посилань» розглядалися окремі типи даних та їх розташування в оперативній пам'яті. Так, при використанні змінних типів значень у методі всі значення цих змінних потрапляють у стек. Після завершення роботи методу стек очищається.

При використанні ж типів посилань, наприклад, об'єктів класів, для них також буде відводитися місце в стеку, тільки там буде зберігатися не значення, а адреса на ділянку пам'яті в хіпі або купі, в якому вже і буду знаходитися самі значення даного об'єкта. Якщо об'єкт класу перестає використовуватися, при очищенні стека посилання на ділянку пам'яті також «очищається». Але це не призводить до негайного «очищення» самої ділянки пам'яті в купі. Згодом збирач сміття (garbage collector) "побачить", що на цю ділянку пам'яті більше немає посилань, і очистить його.

Наприклад:

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

    private static void Test() {
        Country country = new Country();
        country.x = 10;
        country.y = 15;
    }
}
class Country {
    public int x;
    public int y;
}
```

У методі Test створюється об'єкт Country. За допомогою оператора new у купі для зберігання об'єкта CRL виділяє ділянку пам'яті. У стек додає адресу на цю ділянку пам'яті. У головному методі Main викликається метод Test. Після того, як Test «відпрацює», місце в стеку очищається, а збирач сміття очищає раніше виділену під зберігання об'єкта country ділянку пам'яті.

Складальник сміття не запускається відразу після видалення стека посилання на об'єкт, розміщений у купі. Він запускається в той час, коли середовище CLR виявить потребу, наприклад, коли програмі потрібна додаткова пам'ять.

Як правило, об'єкти в купі розташовуються невпорядковано, між ними можуть бути «порожнечі». Купа досить сильно фрагментована. Тому після очищення пам'яті в результаті чергового складання сміття, об'єкти, що

залишилися, переміщаються в один безперервний блок пам'яті. Разом з цим відбувається оновлення посилань, щоб правильно вказувати на нові адреси об'єктів. Цей процес зазвичай називають "стисненням пам'яті".

Слід зазначити, що з великих об'єктів існує своя купа - Large Object Heap. У цю купу поміщаються об'єкти, розмір яких більше 85 000 байт. Особливість цієї купи полягає в тому, що при складанні сміття стиснення пам'яті не проводиться (через великі витрати, пов'язані з розміром об'єктів).

На стиснення зайнятого простору оперативної пам'яті потрібен час, протягом якого програма не працює. Але вважається, що завдяки подібному підходу відбувається оптимізація застосування та прискорення його роботи. Тепер, щоб знайти вільне місце в купі середовищі CLR не потрібно шукати ділянки порожнього простору купи серед зайнятих блоків. Їй достатньо звернутися до покажчика купи, який вказує на вільну ділянку пам'яті. Це суттєво зменшує кількість звернень до пам'яті та загалом прискорює роботу програми.

Крім того, щоб знизити витрати від роботи збирача сміття, всі об'єкти в купі поділяються на покоління. Усього існує три покоління об'єктів: 0, 1 та 2-ге.

До покоління 0 належать нові об'єкти, які ще жодного разу не піддавалися збиранню сміття. До покоління 1 відносяться об'єкти, які пережили одну збірку, а до покоління 2 - об'єкти, що пройшли більше одного складання сміття.

Коли збирач сміття починає роботу, він спочатку аналізує об'єкти з покоління 0. Ті об'єкти, які залишаються актуальними після очищення, підвищуються до покоління 1.

Якщо після обробки об'єктів покоління 0 досі необхідна додаткова пам'ять, то збирач сміття приступає до об'єктів з покоління 1. Ті об'єкти, на які вже немає посилань, знищуються, а ті, які, як і раніше, актуальні, підвищуються до покоління 2.

Оскільки об'єкти з покоління 0 є «молодими» і часто перебувають у адресному просторі пам'яті поруч друг з одним, їх видалення проходить із найменшими витратами.

3.1.1. Клас System.GC

Функціонал збирача сміття у бібліотеці класів .NET представляє клас System.GC. Через статичні методи цей клас дозволяє звертатися до збирача сміття. Як правило, потреба у явному застосуванні цього класу відсутня. Найбільш поширеним випадком його використання є складання сміття при роботі з некерованими ресурсами, при інтенсивному виділенні великих обсягів пам'яті, при яких необхідне таке ж швидке звільнення.

Розглянемо деякі методи та властивості класу System.GC:

Метод AddMemoryPressure інформує середовище CLR про виділення великого обсягу некерованої пам'яті, яку слід врахувати під час планування складання сміття. У зв'язку з цим методом використовується метод RemoveMemoryPressure, який вказує на CLR, що раніше виділена пам'ять звільнена, і її не треба враховувати при складання сміття.

Метод Collect приводить у дію механізм складання сміття. Перевантажені версії методу дозволяють вказати покоління об'єктів, аж до якого треба зробити сміття.

Метод GetGeneration(Object) дозволяє визначити номер покоління, до якого належить переданий як параметр об'єкт

Метод GetTotalMemory повертає об'єм пам'яті (в байтах), яке зайнято в купі, що керується.

Метод WaitForPendingFinalizers припиняє роботу поточного потоку до звільнення всіх об'єктів, для яких здійснюється складання сміття

Приклад роботи із цими методами System.GC:

```
//.................................
long totalMemory = GC.GetTotalMemory(false);

GC.WaitForPendingFinalizers();
GC.Collect();
//.....................................
```

Перед викликом складання сміття через GC.Collect слід викликати метод GC.WaitForPendingFinalizers, який дозволить об'єктам, що звільняються, виконати код з очищення пам'яті, який визначений у методах і деструкторах даних об'єктів.

За допомогою перевантажених версій методу GC.Collect можна виконати більш точне налаштування складання сміття. Так, його перевантажена версія приймає як параметр число - номер покоління, до якого треба виконати очищення. Наприклад, GC.Collect(0) - видаляються лише об'єкти покоління 0.

Ще одна перевантажена версія приймає ще й другий параметр – перерахування GCCollectionMode. Цей перелік може набувати трьох значень:

- Default: значення за промовчанням для цього переліку (Forced)
- Forced: викликає негайне виконання складання сміття

Optimized: дозволяє збиральнику сміття визначити, чи є поточний момент оптимальним для збирання сміття.

Наприклад, негайне складання сміття до першого покоління об'єктів: GC.Collect(1, GCCollectionMode.Forced);

3.2. Фіналізовані об'єкти

Більшість об'єктів програми C#, що використовуються, відносяться до «керованих» або managed-коду. Вони легко очищаються збирачем сміття. Однак разом з тим існують об'єкти, які задіяють "некерований код" (низькорівневі файлові дескриптори, мережеві підключення тощо). Такі «некеровані» об'єкти звертаються до операційної системи API через служби PInvoke. Складальник сміття може «командувати» керованими об'єктами, однак він не знає, як видаляти некеровані об'єкти. І тут розробник повинен сам

реалізовувати механізми очищення оперативної пам'яті лише на рівні програмного коду.

Звільнення некерованих ресурсів передбачає реалізацію одного з двох механізмів:

- Створення деструктора
- Реалізація класом інтерфейсу System.IDisposable

3.2.1. Створення деструкторів

Метод деструктора має ім'я класу (як і конструктор), перед яким стоїть знак тильди (~). Наприклад, створимо деструктор класу`Person`:

```
public class Person {
    ~Person() {
        Console.Beep();
    }
}
```

Деструктор на відміну конструктора немає модифікаторів доступу. У разі деструкторі з метою демонстрації просто викликається метод – «подати звуковий сигнал», але у реальних програмах в деструкторі програмується логіка звільнення некерованих ресурсів.

Однак фактично при очищенні збирач сміття викликає не деструктор, а метод Finalize класу Person. Компілятор C# компілює деструктор у конструкцію, яка еквівалентна наступній:

```
protected override void Finalize() {
    try {
        // Тут ідуть інструкції деструктора
    } finally {
            base.Finalize();
     }
}
```

Метод Finalize визначений у базовому всім типів класі Object, але його не можна перевизначити стандартним способом. Фактична його реалізація відбувається створення деструктора.

Використовуючи в програмі клас Person, після її завершення можна буде почути голосовий сигнал:

```
class Program {
    static void Main(string[] args) {
        Test();
        Console.ReadLine();
    }
     private static void Test() {
        Person p = new Person();
    }
}
```

Зверніть увагу, що навіть після завершення методу Test і відповідно видалення з стека посилання на об'єкт Person у купі, може не наслідувати негайний виклик деструктора. Лише після завершення всієї програми гарантовано відбудеться очищення пам'яті та виклик деструктора.

На рівні пам'яті це виглядає так: збирач сміття при розміщенні об'єкта в купі визначає, чи підтримує даний об'єкт метод Finalize. І якщо об'єкт має метод Finalize, то покажчик на нього зберігається у спеціальній таблиці, що називається черга фіналізації. Коли настає момент складання сміття, збирач «бачить», що цей об'єкт має бути знищений. І, якщо він має метод Finalize, він копіюється в ще одну таблицю і остаточно знищується, лише при наступному проході збирача сміття.

Тут можлива наступна проблема: а якщо нам негайно треба викликати деструктор і звільнити всі пов'язані з об'єктом некеровані ресурси? У цьому випадку, можливо, використовувати другий підхід - реалізацію інтерфейсу IDвихідного.

3.2.2. Інтерфейс IDisposable

Інтерфейс IDisposable повідомляє один єдиний метод Dispose, в якому при реалізації інтерфейсу в класі має відбуватися звільнення некерованих ресурсів. Наприклад:

```
class Program {
    static void Main(string[] args) {
        Test();
        Console.ReadLine();
    }

    private static void Test() {
        Person p = null;
        try {
            p = new Person();
        } finally {
            if (p! = null) {
                p.Dispose();
            }
        }
    }
}
public class Person : IDisposable {
    public void Dispose() {
        Console.Beep();
    }
}
```

У цьому коді використовується конструкція **try...finally**. По суті, ця конструкція по функціоналу еквівалентна наступним двом рядкам коду:

```
Person p = new Person();
p.Dispose();
```

Але конструкцію try...finally краще використовувати при виклику методу Dispose, тому що вона гарантує, що навіть у разі виникнення виключення відбудеться звільнення ресурсів у методі Dispose.

Синтаксис C# також пропонує синонім конструкцію для автоматичного виклику метод Dispose - конструкцію using:

```
using(Person p = new Person()) {
}
```

3.2.3. Комбінування підходів

Вище було розглянуто два підходи. Який із них кращий? З одного боку, метод Dispose дозволяє будь-якої миті часу викликати звільнення пов'язаних ресурсів, з другого - програміст, використовує клас, може забути поставити у коді виклик методу Dispose. Щоб поєднувати плюси обох підходів, можна використовувати «комбінований підхід». Як цей «підхід», фірма Microsoft пропонує наступний формалізований шаблон:

```
public classSomeClass: IDisposable {
    private bool disposed=false;

    // Реалізація інтерфейсу IDisposable.
    public void Dispose() {
        Dispose(true);
           //пригнічуємо фіналізацію
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing) {
        if (!disposed) {
            if (disposing) {
                // Звільняємо керовані ресурси
            }
            // звільняємо некеровані об'єкти
            disposed=true;
        }
    }

    // Деструктор
    ~SomeClass() {
        Dispose (false);
    }
}
```

Логіка очищення реалізується перевантаженою версією методу Dispose (bool disposing). При виклику деструктора як параметр disposing передається

значення false, щоб уникнути очищення керованих ресурсів, оскільки немає впевненості у тому стані – невідомо що досі перебувають у пам'яті. І тут залишається покладатися на деструктори цих ресурсів. В обох випадках також звільняються некеровані ресурси.

Ще один важливий момент - виклик у методі Dispose методу GC.SuppressFinalize(this). GC.SuppressFinalize не дозволяє системі виконати викликати метод Finalize для цього об'єкта.

Таким чином, навіть якщо розробник не використовує у програмі метод Dispose, все одно відбудеться очищення та звільнення ресурсів.

3.2.4. Загальні рекомендації щодо використання Finalize та Dispose

Деструктор слід реалізовувати тільки в тих об'єктів, яким він дійсно необхідний, оскільки метод Finalize дуже впливає на продуктивність.

Після виклику методу Dispose необхідно блокувати у об'єкта виклик методу Finalize за допомогою GC.SuppressFinalize

При створенні похідних класів від базових, які реалізують інтерфейс IDisposable, слід також викликати метод Dispose базового класу:

```
public classDerived: Base {
  private bool IsDisposed = false;
  protected override void Dispose(bool disposing){
        if (! IsDisposed) {
            if (disposing) {
              // Звільнення керованих ресурсів
            }
            IsDisposed = true;
        }
    // Звернення до методу Dispose базового класу
        base.Dispose(disposing);
   }
}
```

Віддавайте перевагу комбінованого шаблону, що реалізує як метод Dispose, так і деструктор.

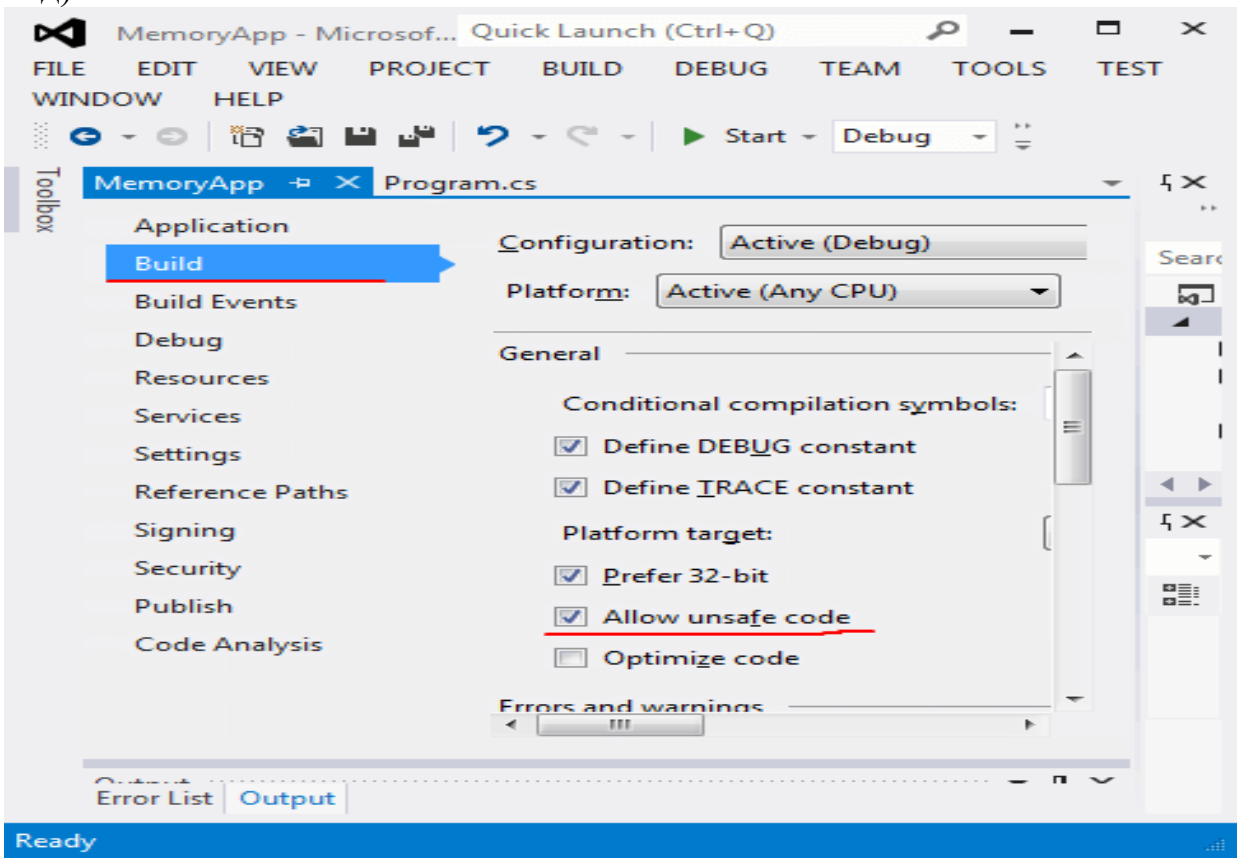
3.3. Вказівники

Покажчики дозволяють отримати доступ до певних осередків оперативної пам'яті і зробити певні маніпуляції зі значеннями, що зберігаються в цьому осередку.

У мові C# покажчики дуже рідко використовуються, однак у деяких випадках можна вдаватися до них для оптимізації програм. Код, який застосовує покажчики, називають небезпечним кодом. Однак це не означає, що він становить якусь небезпеку. Просто під час роботи з ним всі дії з використання пам'яті, зокрема з її очищення, лягати повністю на програміста, а чи не середу CLR. З погляду CLR такий код не безпечний, оскільки середовище

неспроможна перевірити цей код, що тягне у себе підвищення ймовірності появи різноманітних помилок у програмі.

Щоб використовувати небезпечний код у C#, треба насамперед вказати проекту, що він буде використаний з небезпечним кодом. Для цього необхідно встановити в налаштуваннях проекту відповідний прапор – в меню «Project» викликати «Властивості проекту». Потім у меню "Build" встановити прапорець "Allow unsafe code" (Дозволити небезпечний код):



Тепер можна працювати з небезпечним кодом та покажчиками.

3.3.1. Ключове слово unsafe

Блок коду або метод, у якому використовуються покажчики, позначається ключовим словом unsafe:

```
static void Main(string[] args) {
      // блок коду, який використовує покажчики
    unsafe {

    }
}
```

Метод, який використовує покажчики:

```
unsafe private static void PointerMethod() {
```

```
}
```

Також за допомогою unsafe можна оголошувати структури:

```
unsafestruct State {


}
```

3.3.2. Операції * та &

Ключовою під час роботи з покажчиками є операція *, яку часто називають*операцією розіменування*. Операція розіменування дозволяє отримати або встановити значення за адресою, на яку вказує вказівник. Для отримання адреси змінної застосовується операція & - "взяття адреси":

```
static voidMain(string[] args) {
    unsafe {
        int * x; // Визначення покажчика
        int y = 10; // Визначаємо змінну
        x = &y; // покажчик x тепер вказує // на адресу
змінної y
        Console.WriteLine(*x); // 10
        y = y + 20;
        Console.WriteLine(*x);// 30
        * x = 50;
        Console.WriteLine(y); // Змінна y=50
    }
    Console.ReadLine();
}
```

При оголошенні покажчика фіксується тип int * x; - у разі оголошується покажчик на ціле число. Але крім типу int можна використовувати й інші: sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal або bool. Також можна оголошувати покажчики на типи enum, структури та інші покажчики.

Вираз x = &y; дозволяє отримати адресу змінної y та встановити на неї покажчик x. До цього покажчик x ні на що не вказував.

Після цього всі операції з y будуть впливати на значення, одержуване через покажчик x і навпаки, оскільки вони вказують на ту саму область у пам'яті.

Для отримання значення, яке зберігається в області пам'яті, яку вказує покажчик x, використовується вираз *x.

3.3.3. Отримання адреси

Використовуючи перетворення покажчика до цілого типу, можна отримати адресу пам'яті, на яку вказує покажчик:

```
int* x; // Визначення покажчика
inty = 10; // Визначаємо змінну
x = &y; // покажчик x тепер вказує на адресу // змінної y
```

```
// Отримаємо адресу змінної y
uintaddr = (uint) x;

Console.WriteLine("Адреса змінної y: {0}", addr);
```

Оскільки значення адреси - це ціле число, але в 32-разрядных системах діапазон адрес 0 до ~4000000000, то отримання адреси використовується перетворення на тип uint, long чи ulong. Відповідно на 64-розрядних системах діапазон доступних адрес набагато більше, тому в даному випадку краще використовувати ulong, щоб уникнути помилки переповнення.

3.3.4. Операції із вказівниками

Крім операції розіменовування до покажчиків застосовні ще деякі арифметичні операції (+, ++, -, --, +=, -=) і перетворення. Наприклад, можна перетворити число на покажчик:

```
int* x; // Визначення покажчика
inty = 10; // Визначаємо змінну
x = &y; // покажчик x тепер вказує на адресу // змінної y

// Отримаємо адресу змінної y
uintaddr = (uint) x;
Console.WriteLine("Адреса змінної y: {0}", addr);

byte * bytePointer = (byte *) addr + 4; // отримати
покажчик // наступного байта //після addr
Console.WriteLine("Значення int за адресою {0}: {1}",
addr+4, *bytePointer);

//зворотна операція
uintoldAddr = (uint)bytePointer - 4; // віднімаємо чотири
// байта, // так як bytePointer - // покажчик на байт
int* intPointer = (int *) oldAddr;
Console.WriteLine("Значення int за адресою {0}: {1}",
                                  oldAddr, * intPointer);

// Перетворення на тип double
double* doublePointer = (double*)addr + 4;
Console.WriteLine("Значення double за адресою {0}: {1}",
addr + 4, * doublePointer);
```

Так як x - покажчик на об'єкт int, який займає 4 байти, то можна отримати наступний за ним байт за допомогою виразу byte * chp = (byte *) addr + 4;. Тепер покажчик bytePointer вказує на наступний байт. Можна створити й інший покажчик double* doublePointer = (double*)addr + 4; тільки цей покажчик вже буде вказувати на наступні 8 байт, так як тип double займає 8 байт.

Щоб назад отримати вихідну адресу, використовується вираз bytePointer - 4. Тут bytePointer - це покажчик, а не число, і операції віднімання та додавання будуть відбуватися відповідно до правил арифметики покажчиків. Наприклад:

```
char*charPointer = (char *) 123400;
charPointer + = 4; // 123408
Console.WriteLine("Адреса {0}", (uint)charPointer);
```

Хоча до покажчика додаємо число 4, але підсумкова адреса збільшиться на 8, оскільки обсяг об'єкта char - 2 байти,(2*4=8). Подібним чином діє додавання з іншими типами покажчиків:

```
double* doublePointer = (double*)123000;
doublePointer = doublePointer+3; // 123024
Console.WriteLine("Адреса {0}", (uint)doublePointer);
```

Аналогічно працює віднімання: doublePointer -= 2 встановить в покажчику doublePointer як адресу число 123008

3.3.5. Покажчик на інший покажчик

Приклад оголошення та використання покажчика на покажчик:

```
static voidMain(string[] args) {
    unsafe {
        int* x; // Визначення покажчика
        int y = 10; // Визначаємо змінну

        x = &y; // покажчик x тепер містить (вказує)
адресу // змінної y
        int** z = &x; // покажчик z тепер вказує на //
адресу, яка вказує // на покажчик x
        **z = **z + 40; // зміна покажчика z спричинить
// зміну змінної y
        Console.WriteLine(y); // Змінна y=50
        Console.WriteLine(**z); // Змінна **z=50
    }
    Console.ReadLine();
}
```

3.4. Покажчики на структури, члени класів та масиви

3.4.1. Вказівники на типи та операція ->

Крім покажчиків на прості типи можна використовувати покажчики структури. Для доступу до полів структури, яку вказує покажчик, використовується операція ->:

```
classProgram {
```

```
    static void Main(string[] args) {
        unsafe {
            Person person;
            person.age = 29;
            person.height = 176;
            Person* p = &person;
            p-> age = 30;
            Console.WriteLine(p->age);

            // Розіменування покажчика
            (* p). height = 180;
            Console.WriteLine((*p).height);
        }
    }
}
public structPerson {
    public int age;
    public int height;
}
```

Звертаючись до покажчика p-> age = 30; можна отримати чи встановити значення властивості структури, яку вказує покажчик. Зверніть увагу, що написати p.age=30 не можна, оскільки p - це структура Person, а покажчик на структуру.

Альтернативою є операція розіменування: (*p).height = 180;

3.4.2. Вказівники на масиви та stackaloc

За допомогою ключового слова stackaloc можна виділити пам'ять під масив у стеку. Виділення пам'яті в стеку, як правило, призводить до підвищення швидкодії виконання програми. Розглянемо, наприклад, обчислення факторіалу (досить екзотичним способом):

```
unsafe{
    const int size = 7;
    int * factorial = stackalloc int [size]; // виділяємо
//пам'ять у //стеку під сім //об'єктів int
    int * p = factorial;

    *(p++)= 1; // привласнюємо першому осередку значення
1 і
               // збільшуємо покажчик на 1
    for (int i = 2; i <= size; i++, p++) {
        // Розрахуємо факторіал числа
        * p = p [-1] * i;
    }
    for (int i = 1; i <= size; ++i) {
```

```
        Console.WriteLine(factorial[i-1]);
    }
}
```

Оператор stackalloc працює подібно до оператора**new**, повертаючи покажчик на масив цілих:`int * factorial = stackalloc int [size];`.

Для маніпуляцій з масивом створюємо покажчик p: int* p = factorial;, який вказує на перший елемент масиву, в якому всього 7 елементів

Далі починаються операції з покажчиком до розрахунку факторіалу. Оскільки факторіал 1 дорівнює 1, то присвоюємо першому елементу, який вказує покажчик p, одиницю з допомогою операції розіменування: *(p++)= 1;

Щоб записати деяке значення за адресою, що зберігає покажчик, треба використовувати вираз: * p = 1. Але, крім цього, виконується також інкремент покажчика p++. Тобто спочатку першому елементу масиву надається одиниця, потім покажчик p зміщується на одиницю і починає вказувати вже на другий елемент. Це можна записати й іншими операторами:

```
* p = 1;
p++;
```

Щоб отримати попередній елемент («зміститися назад»), можна використовувати операцію декремента: Console.WriteLine(*(--p));. Зверніть увагу, що операції *(--p) і *(p--) розрізняються, тому що в першому випадку спочатку йде зміщення покажчика, а потім його розіменування. А в другому випадку – навпаки.

Потім обчислюється факторіал решти шести чисел: *p=p[1]*i;. Звернення до покажчиків як масивів, представляє альтернативу операції розіменовування для отримання значення. У разі отримуємо значення попереднього елемента.

І на закінчення, використовуючи покажчик factorial, виводимо значення факторіалів всіх семи чисел.

3.4.3. Оператор fixed та закріплення покажчиків

Раніше розглянули, як створювати покажчики на типи значень, наприклад, int чи структури. Однак крім структур у C# є ще й класи, які, на відміну від типів значень, зберігають усі пов'язані значення в купі. У роботу цих класів може будь-якої миті втрутитися збирач сміття, який періодично очищає купу. Щоб фіксувати весь час роботи покажчики на об'єкти класів використовується оператор fixed.

Допустимо, у нас є клас Person:

```
public classPerson {
    public int age;
    public int height;
}
```

Зафіксуємо покажчик за допомогою оператора fixed:

```
unsafe {
    Person person = new Person();
    person.age = 28;
    person.height = 178;
    // блок фіксації покажчика
    fixed(int* p = &person.age) {
        if (*p < 30) {
            * p = 30;
        }
    }
    Console.WriteLine(person.age); // 30
}
```

Оператор fixed створює блок, у якому фіксується покажчик на полі об'єкта person. Після завершення блоку fixed закріплення зі змінних знімається, і вони можуть бути схильні до складання сміття.

Крім адреси змінної можна також ініціалізувати покажчик, використовуючи масив, рядок або буфер фіксованого розміру:

```
unsafe {
    int[] nums = {0, 1, 2, 3, 7, 88};
    string str = "Привіт світ";
    fixed(int* p = nums) {
……..
    }
    fixed(char * p = str) {…….
    }
}
```

При ініціалізації покажчиків на рядок слід враховувати, що покажчик повинен мати тип char*.

3.5. Контрольні питання

1. Призначення збирача сміття C#.
2. Призначення та функціонал класу System.GC.
3. Призначення та правила створення деструкторів.
4. Що таке код небезпечний?
5. Призначення ключового слова unsafe.
6. Призначення та використання операції * та &.
7. Основні операції із покажчиками.
8. Операція ->, її призначення та використання.
9. Призначення ключового слова "stackaloc".
10. Призначення оператора fixed.

4. Dynamic Language Runtime

4.1. DLR C#. Ключове слово dynamic

Формально мова C# відноситься до статично типізованих мов. Але в останніх версіях мови було додано деякі динамічні можливості. Так, починаючи з .NET 4.0 було додано нову функціональність під назвою DLR (Dynamic Language Runtime). DLR представляє середовище виконання динамічних мов, наприклад, мов IronPython і IronRuby.

Щоб зрозуміти значення цього нововведення, потрібно усвідомлювати різницю між мовами зі статичною та динамічною типизацією. У мовах зі статичною типізацією виявлення всіх типів та його членів - властивостей та методів відбувається на етапі компіляції. А в динамічних мовах системі нічого не відомо про властивості та методи типів аж до виконання.

Завдяки цьому середовищу DLR C# може створювати динамічні об'єкти, члени яких «виявляються» на етапі виконання програми, та використовувати їх разом із традиційними об'єктами зі статичною типізацією.

Ключовим моментом використання DLR C# є застосування типів dynamic. Це ключове слово дозволяє перевірити типи під час компіляції. Крім того, об'єкти, оголошені як динамічний, можуть протягом роботи програми змінювати свій тип. Наприклад:

```
class Program {
    static void Main(string[] args) {
        dynamic x = 3; // тут x - цілісне int
        Console.WriteLine(x);

        x = "Привіт світ"; // x - рядок
        Console.WriteLine(x);

        x = new Person() {Name = "Tom", Age = 23}; //x-
об'єкт Person
        Console.WriteLine(x);

        Console.ReadLine();
    }
}
class Person {
    public string Name {get;set;}
    public int Age {get; set; }

    public override string ToString() {
        return Name + "," + Age.ToString();
    }
```

```
}
```

Незважаючи на те, що змінна x змінює тип свого значення кілька разів, цей код працюватиме. При цьому використання типів dynamic відрізняється від застосування ключового слова var. Для змінної, оголошеної ключовим словом var, тип «виводиться» під час компіляції і потім під час виконання більше змінюється.

Також можна знайти спільне між використанням dynamic та типом object. Якщо попередньому прикладі замінити dynamic на object: object x = 3;, то результат буде той самий. Однак тут є відмінності. Наприклад:

```
objectobj = 24;
dynamicdyn = 24;
obj + = 4; // так не можна
dyn += 4; // а так можна
```

Рядок obj + = 4; помилкова, оскільки операція += може бути застосована до типів object. Зі змінною, оголошеною як dynamic, це можливо, тому що її тип буде відомий тільки під час виконання (що можливо призведе до помилки часу виконання…).

Ще одна відмінна риса використання dynamic полягає в тому, що це ключове слово можна застосувати не тільки до змінних, але й до властивостей та методів. Наприклад:

```
classPerson {
    public string Name {get;set;}
    public dynamic Age { get; set; }

// Виводимо зарплату залежно від переданого формату
    public dynamic getSalary(dynamic value, string
format) {
        if (format=="string") {
            return value + "рублів";
        }
        else if (format == "int") {
            return value;
        } else {
            return 0.0;
        }
    }

    public override string ToString() {
        return Name + "," + Age.ToString();
    }
```

```
}
```

У класі Person визначено динамічну властивість Age, тому за завдання значення цій властивості можна писати і person.Age=33, і person.Age="тридцять три". Обидва варіанти будуть правильними.

Також є метод getSalary, що повертає значення dynamic. Наприклад, залежно від параметра можна «повернути» або строкове подання суми доходу чи чисельне. Також метод приймає dynamic як параметр. Таким чином, можна передати як значення доходу як ціле, так і дробове число. Подивимося на конкретне застосування:

```
dynamicperson1 = new Person() { Name = "Том", Age = 27};
Console.WriteLine(person1);
Console.WriteLine(person1.getSalary(28.09, "int"));

dynamicperson2 = new Person() { Name = "Білл", Age =
"Двадцять два роки"};
Console.WriteLine(person2);
Console.WriteLine(person2.getSalary(30, "string"));
```

4.2. DynamicObject та ExpandoObject

Цікаві можливості при розробці C# і .NET з використанням DLR надає простір імен System.Dynamic і зокрема клас ExpandoObject. Він дозволяє створювати динамічні об'єкти, на зразок тих, що використовуються в javascript:

```
dynamicviewbag = New System.Dynamic.ExpandoObject();
viewbag.Name = "Tom";
viewbag.Age = 46;
viewbag.Languages = новий List<string> { "english",
"german",
                                        "french"};
Console.WriteLine("{0} - {1}", viewbag.Name,
viewbag.Age);
foreach(var lang in viewbag.Languages)
    Console.WriteLine(lang);

// Оголошуємо метод
viewbag.IncrementAge = (Action<int>) (x => viewbag.Age +=
x);
viewbag.IncrementAge(6); // збільшуємо вік на 6 років
Console.WriteLine("{0} - {1}", viewbag.Name,
viewbag.Age);
```

Консольний висновок:

```
Tom - 46
english
german
french
Tom – 52
```

У динамічного об'єкта ExpandoObject можна оголосити будь-які властивості, наприклад, Name, Age, Languages, які можуть представляти різні об'єкти. Крім того, можна задати методи за допомогою делегатів.

4.2.1. DynamicObject

На ExpandoObject за своєю дією схожий інший клас – DynamicObject. Він також дозволяє ставити динамічні об'єкти. Тільки в цьому випадку необхідно створити свій клас, успадкувавши його від DynamicObject і реалізувати його методи:

- `TryBinaryOperation(): виконує бінарну операцію між двома об'єктами. Еквівалентно стандартним бінарним операціям, наприклад, доданню x + y)`
- `TryConvert(): Здійснює перетворення до певного типу. Еквівалентно базовому перетворенню C#, наприклад, (SomeType) obj`
- `TryCreateInstance(): створює екземпляр об'єкта`
- `TryDeleteIndex(): видаляє індексатор`
- `TryDeleteMember(): видаляє властивість або метод`
- `TryGetIndex(): отримує елемент індексу через індексатор. C# може бути еквівалентно наступному виразу int x = collection[i]`
- `TryGetMember(): отримуємо значення якості. Еквівалентно зверненню до властивості, наприклад, string n=person.Name`
- `TryInvoke(): виклик об'єкта як делегат`
- `TryInvokeMember(): виклик методу`
- TrySetIndex(): встановлює елемент індексу через індексатор. C# може бути еквівалентно наступному виразу collection[i] = x;
- `TrySetMember(): встановлює властивість. Еквівалентно присвоєння властивості значення person.Name = "Tom"`

`TryUnaryOperation()`:виконує унарну операцію подібно до унарних операцій в C#: x++

Кожен з цих методів має ту саму модель визначення: всі вони повертають логічне значення, що показує, чи вдало пройшла операція. Як перший параметр, всі вони приймають об'єкт зв'язувача або binder. Якщо метод представляє виклик індексатора або об'єкта, які можуть приймати параметри, то як другий параметр використовується масив object[] - він зберігає передані в метод або індексатор аргументи.

Майже всі операції, крім встановлення та видалення властивостей та індексаторів, повертають певне значення (наприклад, значення властивості). У цьому випадку застосовується третій параметр out object vaue, який призначений для зберігання об'єкта, що повертається.

Наприклад, визначення методу TryInvokeMember():

```
public virtualbool TryInvokeMember(InvokeMemberBinder
```

```
binder,
                        object[] args, out object result)
```

Параметр InvokeMemberBinder binder є «зв'язувачем» - отримує властивості та методи об'єкта, object[] args зберігає аргументи, що передаються, out object result призначений для зберігання вихідного результату.

Розглянемо з прикладу. Створимо клас динамічного об'єкта:

```
class PersonObject : DynamicObject {
    Dictionary<string, object> members = new
Dictionary<string, object>();

    //Встановлення властивості
    public override bool TrySetMember(SetMemberBinder
binder, object value) {
        members[binder.Name] = value;
        return true;
    }
    // отримання властивості
    public override bool TryGetMember(GetMemberBinder
binder, out object result){
        result = null;
        if (members.ContainsKey(binder.Name)) {
            result = members[binder.Name];
            return true;
        }
        return false;
    }
    // виклик методу
    public override bool
TryInvokeMember(InvokeMemberBinder binder, object[] args,
out object result) {
        dynamic method = members[binder.Name];
        result = method((int)args[0]);
        return result != null;
    }

}
```

Клас успадковується від DynamicObject, оскільки безпосередньо створювати об'єкти DynamicObject не можна. Тут перевизначаються три успадковані методи.

Для зберігання всіх членів класу, як властивостей, і методів, визначено словник Dictionary<string, object> members. Ключами тут є назви властивостей та методів, а значеннями – значення цих властивостей.

У методі TrySetMember() проводиться установка якості:

```
bool TrySetMember(SetMemberBinder binder, object value)
```

Параметр binder зберігає назвою властивості, що встановлюється (binder.Name), a value - значення, яке йому треба встановити.

Для отримання значення властивості перевизначено метод TryGetMember:

```
boolTryGetMember(GetMemberBinder binder, out object
result)
```

Знову ж таки binder містить назву властивості, а параметр result буде містити значення отримуваної властивості.

Для виклику методів визначено метод TryInvokeMember:

```
public overridebool TryInvokeMember(InvokeMemberBinder
binder, object[] args, out object result) {
    dynamic method = members[binder.Name];
    result = method((int)args[0]);
    return result != null;
}
```

Спочатку за допомогою bindera одержуємо метод і потім передаємо йому аргумент args[0], попередньо привівши його до типу int, і результат методу зберігаємо у параметрі result. Тобто в даному випадку мається на увазі, що метод прийматиме один параметр типу int і повертатиме якийсь результат.

Тепер застосуємо клас у програмі:

```
static voidMain(string[] args) {
    dynamic person = new PersonObject();
    person.Name = "Tom";
    person.Age = 23;
    Func<int, int> Incr = delegate (int x)
{person.Age+=x; return person.Age; };
    person.IncrementAge = Incr;
    Console.WriteLine("{0} - {1}", person.Name,
person.Age); // Tom - 23
    person.IncrementAge(4);
    Console.WriteLine("{0} - {1}", person.Name,
person.Age); // Tom - 27
    Console.Read();
}
```

Вираз person.Name = "Tom" буде викликати метод TrySetMember, який як другий параметр буде передаватися рядок "Tom".

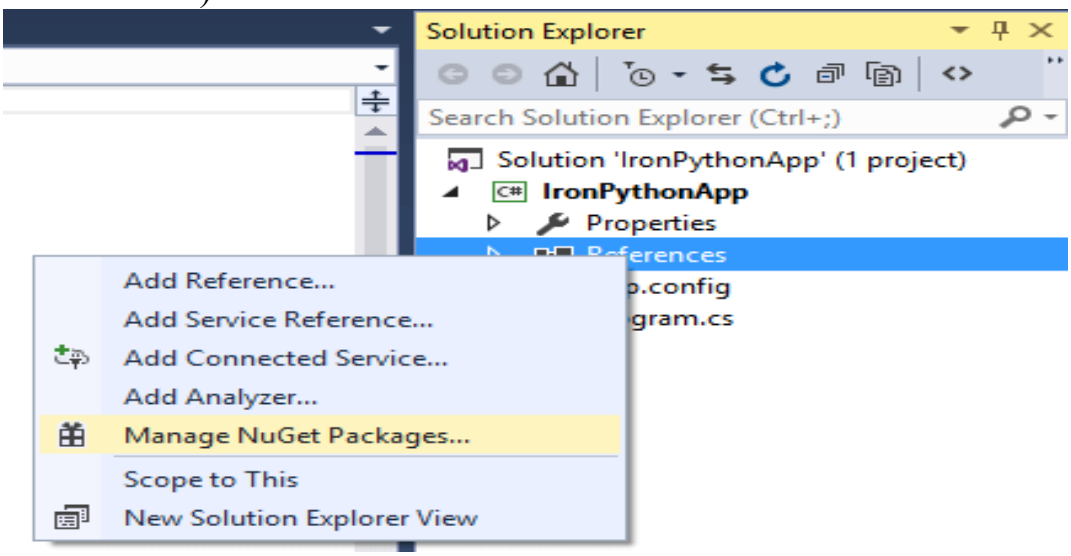
Вираз return person.Age; викликає метод TryGetMember.

Також об'єкт person має метод IncrementAge, який представляє дії анонімного делегата delegate (int x) { person.Age+=x; return person.Age; }. Делегат приймає число x, збільшує на це число властивість Age та повертає нове значення person.Age. І при виклику цього методу відбуватиметься звернення до методу TryInvokeMember. І, таким чином, відбудеться збільшення значення властивості person.Age.

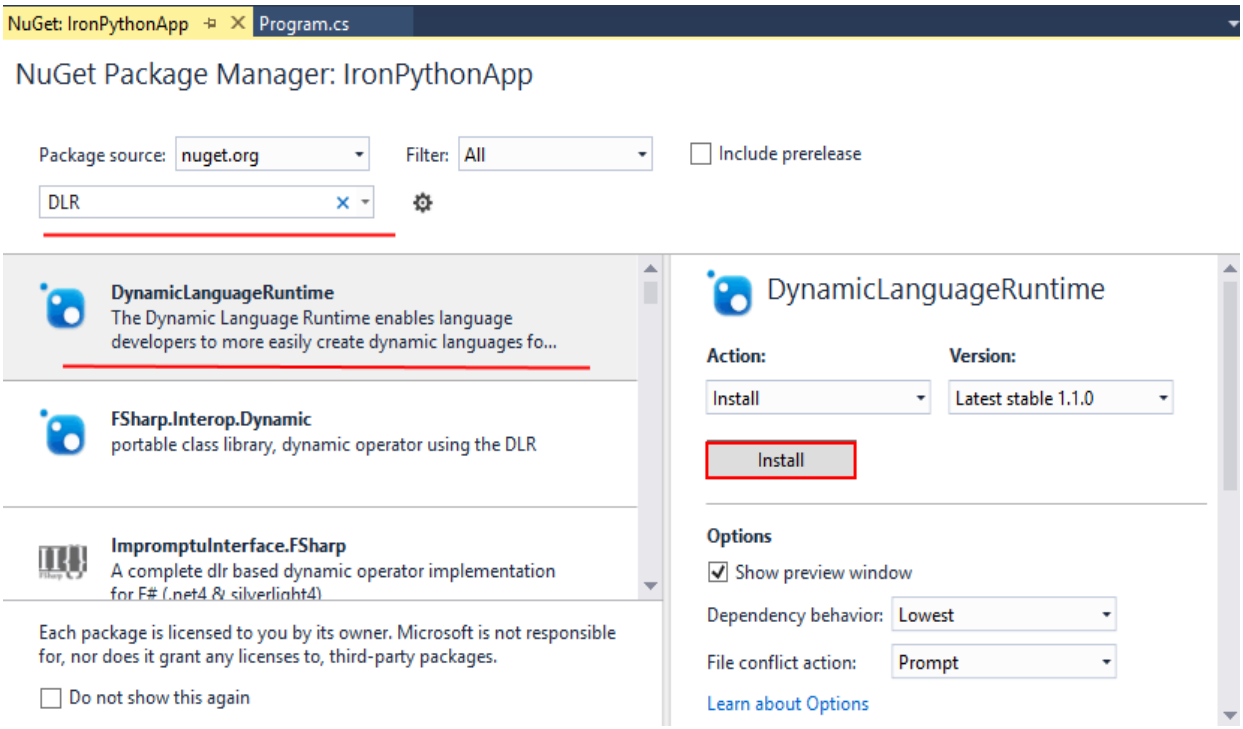
4.3. Використання IronPython у .NET

Одним з ключових переваг DLR є підтримка таких динамічних мов як IronPython і IronRuby. Ці «динамічні мови», можливо, не часто використовуються, однак є сфери, де їхнє застосування є доцільним та виправданим. Наприклад, написання клієнтських сценаріїв. Можливо, користувач програми захоче внести якусь додаткову поведінку до програми і для цього може використовуватися IronPython. Створення клієнтських сценаріїв широко поширене в наші дні, багато програм і навіть ігор підтримують додавання клієнтських сценаріїв, написаних різними мовами. Крім того, можливо, є бібліотеки Python, функціональність яких може бути відсутня в .NET. У цьому випадку можна використовувати IronPython.

Розглянемо на прикладі застосування IronPython. Для початку необхідно додати до проекту кілька пакетів, використовуючи пакетний менеджер NuGet. Для цього необхідно натиснути у вікні проекту на вузол References правою кнопкою миші і вибрати в списку пункт Manage NuGet Packages... (Управління NuGet-пакетами):

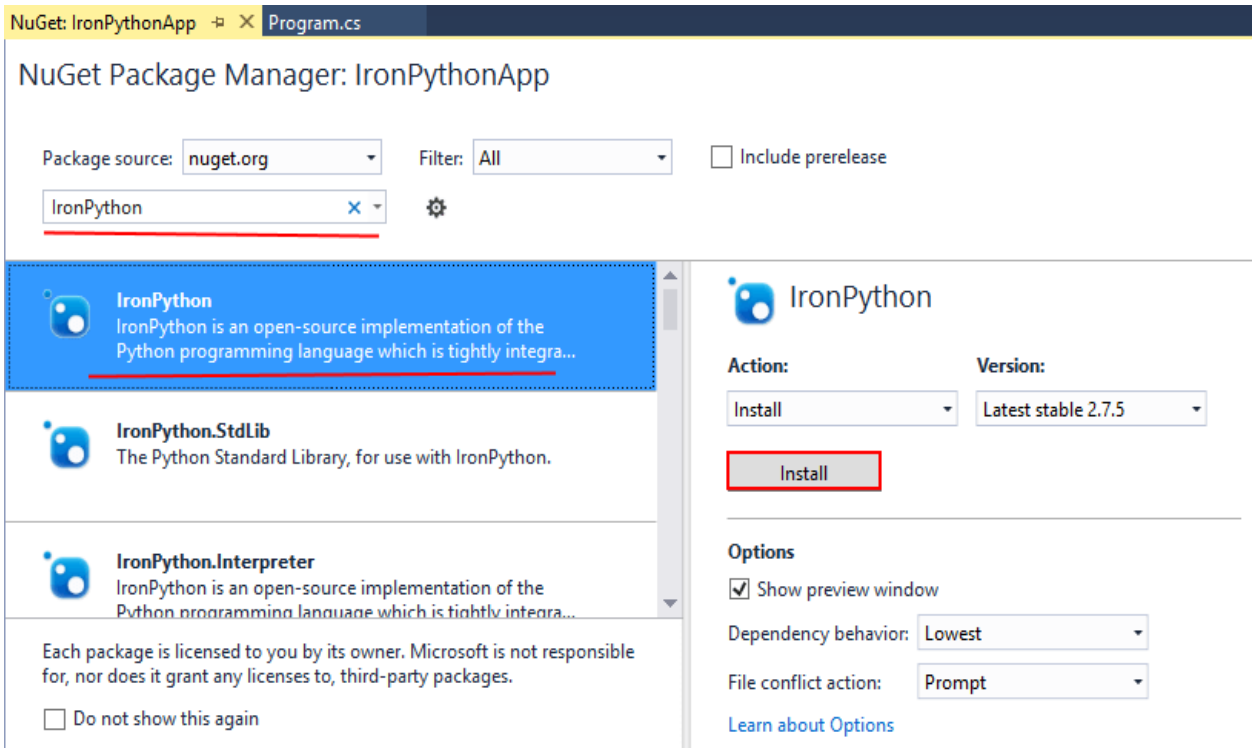


Відкриється вікно пакетного менеджера. Щоб знайти потрібний пакет, введемо в поле пошуку "DLR", і менеджер відобразить ряд результатів, з яких перший - пакет DynamicLanguageRuntime необхідно встановити.



Після цього до проекту у вузол References додається бібліотека Microsoft.Scripting.

Тепер потрібно додати необхідні пакети для IronPython. Для цього введемо в поле пошуку "IronPython" і після цього встановимо однойменний пакет:



Після інсталяції пакета у вузлі References додається бібліотека IronPython.

Тепер напишемо примітивну демонстраційну програму:

```
using System;
using IronPython.Hosting;
```

```
using Microsoft.Scripting.Hosting;

namespace IronPythonApp {
    class Program {
        static void Main(string[] args) {
            ScriptEngine engine =
Python.CreateEngine();
            engine.Execute("print 'hello, world'");
        }
    }
}
```

Консольний висновок:

```
hello, world
```

Тут використовується вираз print 'hello, world' мови Python, яке виводить на консоль рядок. Для створення "движка", що виконує скрипт, застосовується клас ScriptEngine. Його метод Execute() буде виконувати скрипт.

Створимо файл hello.py, звичайний текстовий файл із кодом на мові Python:

```
print 'hello, world'
```

Виконаємо його із програми C#:

```
ScriptEngine engine = Python.CreateEngine();
engine.ExecuteFile("D://hello.py");

Console.Read();
```

4.3.1. ScriptScope

Об'єкт ScriptScope дозволяє взаємодіяти зі скриптом, отримуючи чи встановлюючи його змінні, отримуючи посилання функції. Наприклад, напишемо найпростіший скрипт*hello2.py*, який використовує змінні:

```
x = 10
z = x + y

print z
```

Тепер напишемо програму, яка взаємодіятиме зі скриптом:

```
int yNumber = 22;
ScriptEngine engine = Python.CreateEngine();
ScriptScope scope = engine.CreateScope();
scope.SetVariable("y", yNumber);
engine.ExecuteFile("D://hello2.py", scope);
dynamic xNumber = scope.GetVariable("x");
dynamic zNumber = scope.GetVariable("z");

Console.WriteLine("Сума {0} і {1} дорівнює: {2}",
                          xNumber, yNumber, zNumber);
```

Об'єкт ScriptScope за допомогою методу SetVariable дозволяє встановити змінні у скрипті, а за допомогою методу GetVariable() – отримати їх.

Консольний висновок:

```
32
Сума 10 та 22 дорівнює: 32
```

4.3.2. Виклик функцій із IronPython

Визначимо скрипт*factorial.py*, який містить функцію, що обчислює факторіал числа:

```
def factorial(number):
    result = 1
    for i in xrange(2, number + 1):
        result *= i

    return result
```

Тепер звернемося до цієї функції у коді C#:

```
using System;
using IronPython.Hosting;
using Microsoft.Scripting.Hosting;

namespace IronPythonApp {
    class Program {
        static void Main(string[] args) {
            Console.WriteLine("Введіть число:");
            int x = Int32.Parse(Console.ReadLine());

            ScriptEngine engine = Python.CreateEngine();
            ScriptScope scope = engine.CreateScope();

            engine.ExecuteFile("D://factorial.py",
scope);
            dynamic function =
scope.GetVariable("factorial");
            // Викликаємо функцію і отримуємо результат
            dynamic result = function(x);
            Console.WriteLine(result);

            Console.Read();
        }
    }
}
```

Отримати об'єкт функції можна так само, як і змінну: scope.GetVariable("factorial"); Потім із цим об'єктом працюємо так само, як і з будь-яким іншим методом.

Консольний висновок:

```
Введіть число:
```

```
5
120
```

4.4. Контрольні питання

1. Призначення ключового слова dynamic
2. Призначення класів DynamicObject та ExpandoObject
3. Чи можна використовувати мову Python у .NET?
4. Призначення об'єкта ScriptScope

5. Складання .NET

5.1. Роль збірок у додатках .NET

Коли створюється програма (в результаті компіляції у Visual Studio або консолі), результатом цієї роботи є файл exe або dll (залежно від вибраних налаштувань). Цей файл називається збиранням програми. Складання є базовою структурною одиницею в .NET, на рівні якої проходить контроль версій, розгортання та конфігурація програми.

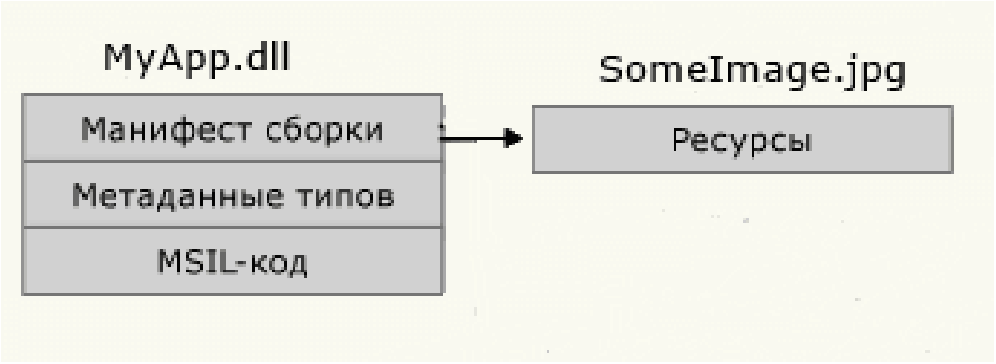
Складання «фіксують» всю бібліотеку класів.

Складання мають такі складові:

- Маніфест, який містить метадані збирання
- Методи типів. Використовуючи ці метадані, збірка визначає розташування типів у файлі програми, а також місця розміщення їх у пам'яті.
- Власне код програми мовою MSIL, який компілюється код C#
- Ресурси

Всі ці компоненти можуть знаходитися в одному файлі, і тоді збірка представляє єдиний файл у форматі exe або dll.

Але можливе зберігання цих компонентів у окремих файлах. Наприклад, є основний файл exe, який має метадані збірки та типів і який використовує додаткові файли ресурсів – різні зображення, звукові файли, допоміжні модулі, елементи інтерфейсу, локалізовані для різних культур.



5.1.1. Маніфест збірки

Ключовим компонентом збирання є її маніфест. Якщо збірка не має маніфесту, то укладений у ній код MSIL виконуватися не буде. Маніфест може перебувати в одному файлі з виконуваним кодом збірки, а може також розміщуватися в окремому файлі.

Маніфест зберігає такі дані:

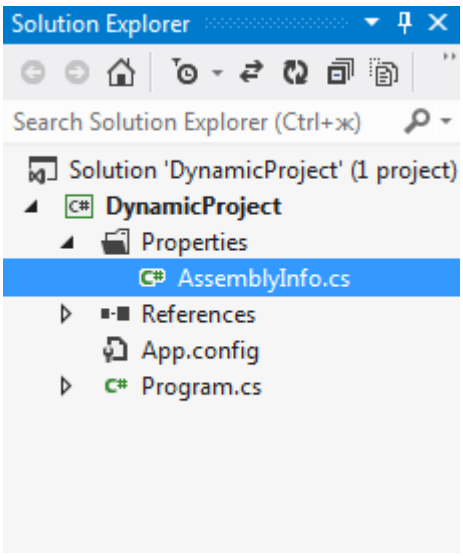
- Ім'я збирання
- Номер версії: основний та додатковий номери. Використовується для керування версіями
- Мова та регіональні параметри: інформація про мову та регіональні параметри, які підтримує складання
- Інформація про суворе ім'я: відкритий ключ видавця
- Список всіх файлів збирання: хеш та ім'я кожного з файлів, що входять до збирання
- Список посилань на інші збирання, які використовує поточне збирання

Список посилань на типи, що використовуються зборкою

Таким чином, маніфест дозволяє системі визначити всі файли, що входять до збирання, зіставити посилання на типи, ресурси, збирання з їх файлами, керувати контролем версій.

5.1.2. Атрибути збирання

За промовчанням Visual Studio під час створення проекту додає файл*AssemblyInfo.cs*, який можна знайти у вузлі Properties:



Це звичайний файл мовою C# і може виглядати так:

```
usingSystem.Reflection;
usingSystem.Runtime.CompilerServices;
usingSystem.Runtime.InteropServices;

// General Information про assembly is controlled //
through the following
// Set of attributes. Зміни цих параметрів значення //
modify the information
// associated with an assembly.
[assembly: AssemblyTitle("DynamicProject")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("DynamicProject")]
[assembly: AssemblyCopyright("Copyright © 2014")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]

// Setting ComVisible to false makes the// types in this
assembly not visible
// to COM components. Якщо ви потрібні для access// a
type in this assembly from
// COM, натиснути ComVisible attribute to true on that //
type.
[assembly: ComVisible(false)]

// Натисніть на GUID для ID ID // typelib if this project
is exposed to COM
[assembly: Guid("8794dabf-2f91-423d-87c4-6317e2913be7")]

// Version information for assembly consists// of
following four values:
```

```
//
// Major Version
// Minor Version
// Build Number
// Revision
//
// Ви можете зазначити всі значення або ви можете//
default build and revision numbers
// by using the '*' як shown below:
// [assembly: AssemblyVersion("1.0.*")]
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

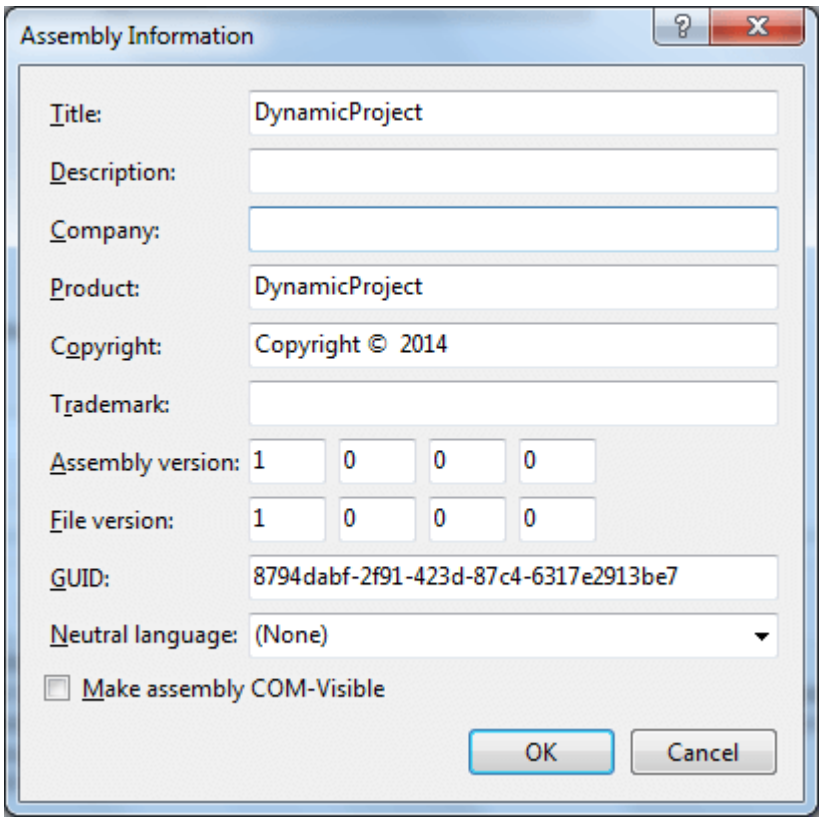
Цей файл містить налаштування маніфесту збирання. Атрибути типу [assembly: AssemblyVersion("1.0.0.0")] встановлюють значення маніфесті. Префікс assembly: перед атрибутом вказує, що це атрибут рівня складання. В даному випадку – атрибут номера версії складання.

Виходячи з назви, призначення багатьох атрибутів очевидне:

- `AssemblyCompany`: назва компанії
- `AssemblyConfiguration`: конфігурація збірки (Retail або Debug)
- `AssemblyCopyright`: авторське право на програму
- `AssemblyDefaultAlias`: стандартний псевдонім, який використовується при посиланні на цю збірку з інших збірок
- `AssemblyDescription`: короткий опис збірки
- `AssemblyProduct`: інформація про продукт
- `AssemblyTitle`: назва збірки як інформаційного продукту
- `AssemblyTrademark`: відомості про торгову марку
- `AssemblyCulture`: задає мову та регіональні параметри, що підтримуються збиранням. Приклад - встановлення російської локалі: `[assembly:AssemblyCultureAttribute("ru")])`
- `AssemblyInformationalVersion`: повна версія збірки
- `AssemblyVersion`: версія складання

`AssemblyFileVersion`: номер версії файлу Win32 За умовчанням збігається з версією збирання.

Також є графічний спосіб задання інформації про складання. Для цього необхідно натиснути у вікні Solution Explorer (браузер рішень) на назву проекту правою кнопкою миші і вибирати в меню пункт Properties (Властивості). У вікні натиснемо кнопку Assembly Information..., і отримаємо вікно зміни атрибутів:



Після будь-яких змін, зроблених у цьому вікні, відповідною інформацією буде автоматично оновлено файл AssemblyInfo.cs.

5.2. Складання, що розділяються. Додавання збірки до GAC

За способом взаємодії з іншими збираннями та програмами збирання можна розділити на дві категорії: закриті та розділяються.

Закриті збирання- це звичайні збірки програми, які створюються Visual Studio. Наприклад, при створенні бібліотеки класів dll створюється закрите складання. Згодом цю закриту збірку можна використовувати, підключивши її до іншого проекту. Для цього її можна просто «покласти поруч із виконуваним файлом» і додати до проекту посилання на неї через Add Reference. На одній робочій станції може бути десяток додатків, які використовують різні копії однієї і тієї ж збирання.

Якщо необхідно видалити програму, то можна також видалити закриту збірку, яку він використовує. Це не позначиться на роботі інших програм цієї робочої станції.

Інакше справа з збірками, що розділяються. За умовчанням при створенні проекту visual Studio вже додає в проект посилання на ряд збірок, що розділяються. Це можна побачити, відкривши вузол References (Посилання). Наприклад, Microsoft.CSharp.dll, System.dll, System.Core.dll - це все збірки, що розділяються.

*Складання, що розділяються*знаходяться у глобальному кеші збірок (Global Assembly Cache). Розташування кешу збірок відрізняється залежно від версії .NET, встановленої на локальній машині. До .NET 4.0 глобальний кеш

знаходився в каталозі C: Windows assembly. Починаючи з версії .NET 4.0 кеш збірок розміщується по дорозі C:\Windows\Microsoft.NET\assembly\GAC_MSIL

5.2.1. Суворе ім'я складання

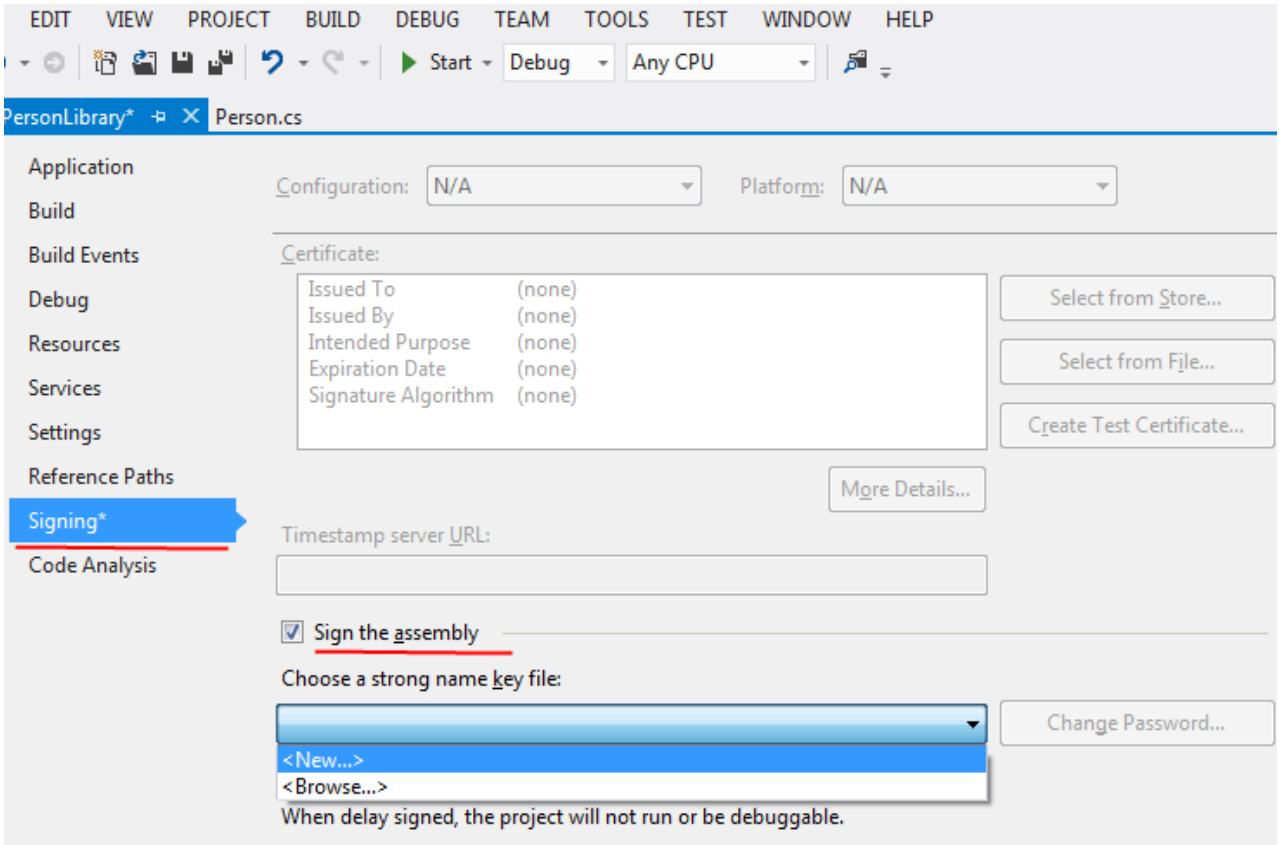
Помістити збірку в GAC (глобальний кеш) можливо, якщо ця збірка має "суворе ім'я". До складу суворого імені входять такі компоненти:

- Ім'я збирання без розширення;
- Номер версії. Завдяки розмежуванню за версією можна використовувати різні версії однієї і тієї ж збірки;
- Відкритий ключ;
- необов'язкове значення для локалі (при локалізації складання);
- Цифровий підпис, який створюється за допомогою хеш-значення вмісту складання та значення секретного ключа. Секретний ключ є файлом з розширенням *.snk.;

Завдяки строгому імені гарантується унікальність збирання у глобальному кеші.

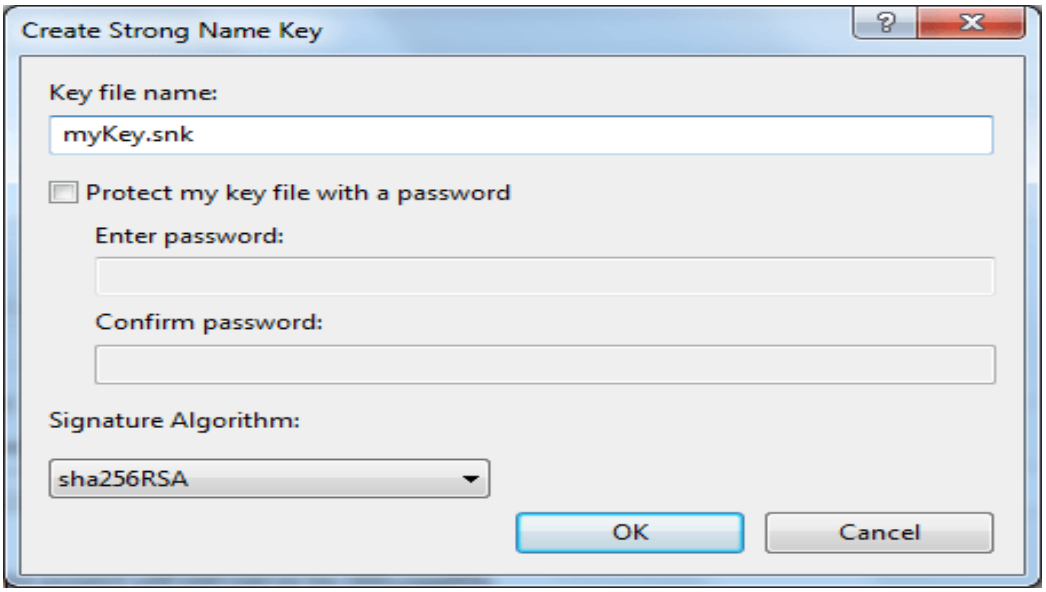
Щоб створити суворе ім'я, можна скористатися інструментарієм Visual Studio.

Допустимо, створено проект за типом Class Library (Бібліотека класів). І тепер необхідно підписати складання (яке компілюватиметься) строгим ім'ям. Для цього необхідно натиснути у вікні Solution Explorer (Обозреватель рішень) на ім'я проекту правою кнопкою миші і в меню вибрати пункт Properties (Властивості). На вкладці властивостей вибрати пункт Signing:

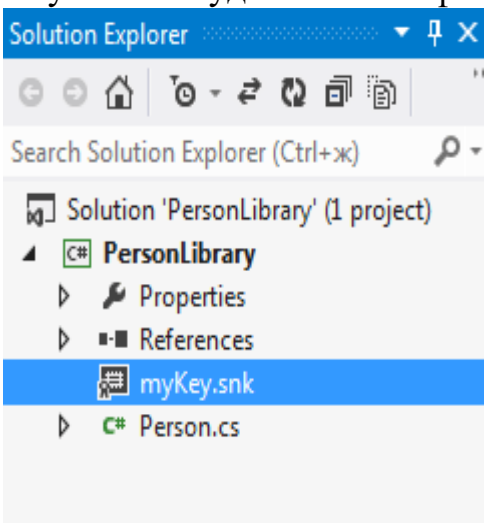


Необхідно встановити прапорець*Sign the assembly (Підписати збірку)*, як показано малюнку.

Щоб створити новий секретний ключ, потрібно вибрати у списку New. Після цього відкриється вікно налаштувань секретного ключа:



Дайте новому ключу якесь ім'я і натисніть ОК. Після цього у структурі проекту можна буде побачити файл ключа:



Після встановлення складання строгого імені її можна додавати до GAC. Для цього скористайтесь утилітою (яка йде в комплекті з .NET Framework) під назвою gacutil.exe.

Відкрийте командний рядок із правами адміністратора. Знайдіть розташування утиліти gacutil.exe на локальній машині. Вона може бути, наприклад, у каталозі

```
C:\Program Files\Microsoft SDKs\Windows\v8.0A\bin\NETFX
4.0 Tools.
```

«Перейдіть» до цього каталогу:

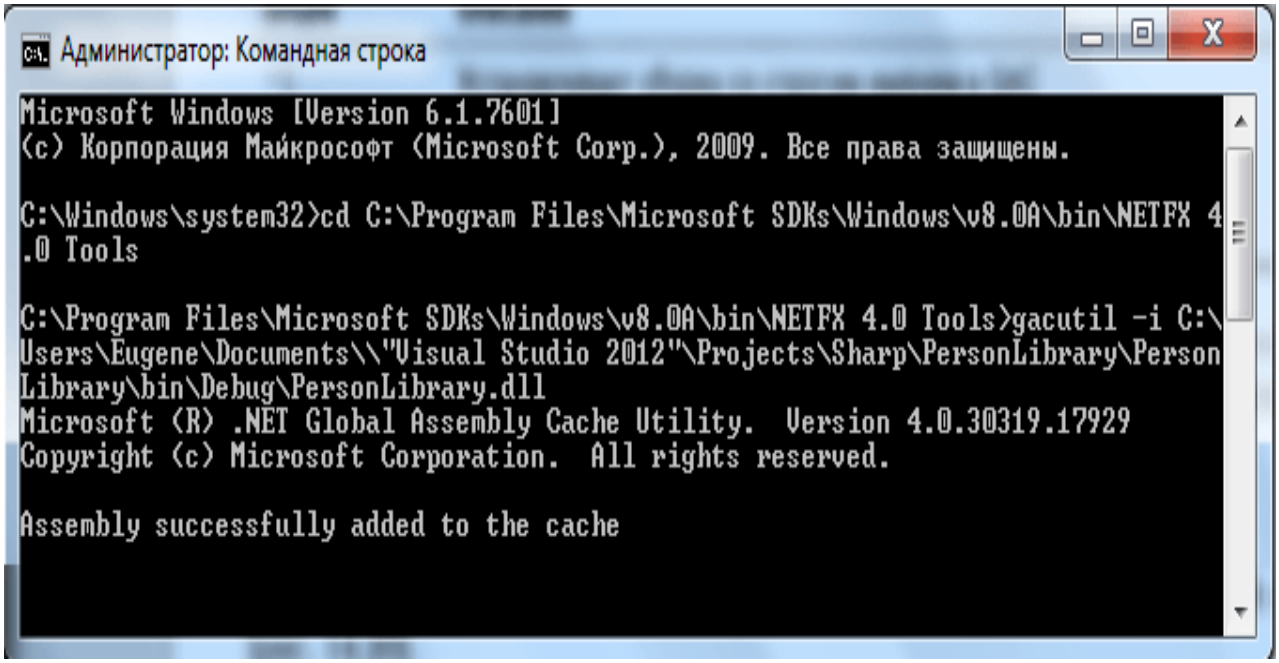
```
C:\Windows\system32>cd C:\Program Files\Microsoft
SDKs\Windows\v8.0A\bin\NETFX 4.0 Tools
```

Скористайтеся однією з команд цієї утиліти. Найбільш використовувані команди:

- *-i ім'я_складання*- встановлення складання в GAC
- *-l*- виведення всього списку збірок до GAC
- *-u ім'я_складання*- видалення складання з GAC

У командному рядку вводимо команду на додавання:

```
C:\Program Files\Microsoft SDKs\Windows\v8.0A\bin\
NETFX 4.0 Tools>gacutil -i Повне_ім'я_зборки
```



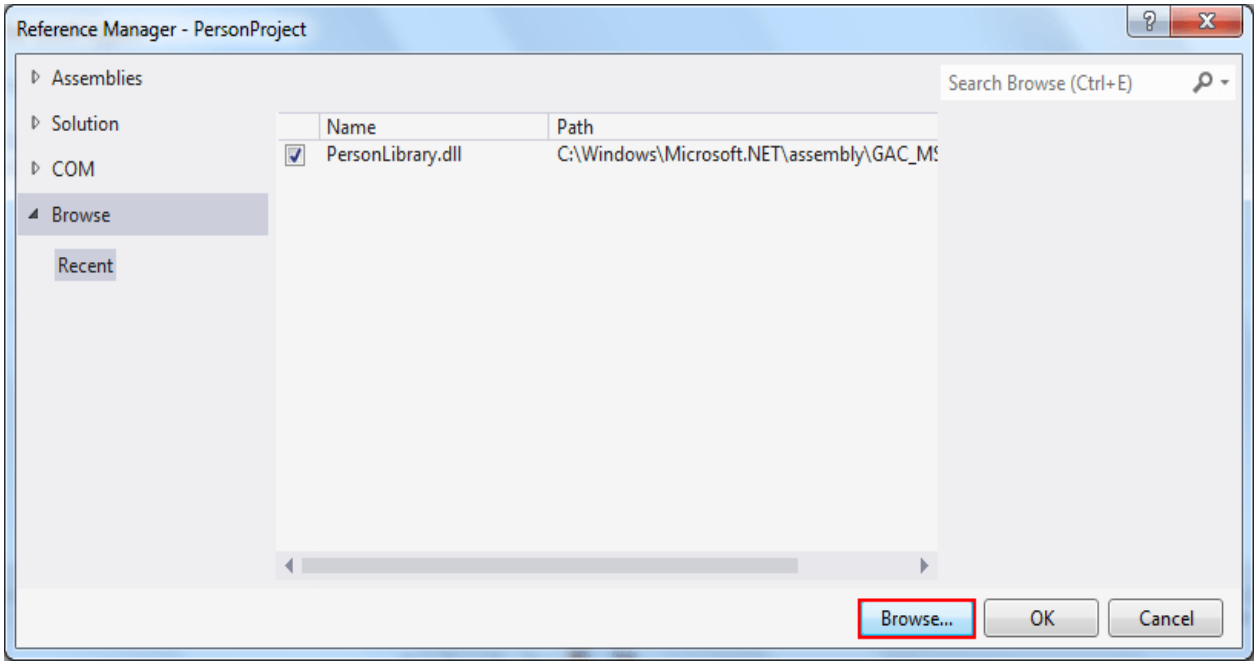
Як видно, на малюнку, у цьому прикладі повне ім'я збирання *C:\Users\Eugene\Documents\Visual Studio 2012\Projects\Sharp\PersonLibrary\PersonLibrary\bin\Debug\PersonLibrary.dll*.

І якщо додавання пройшло успішно, то на консолі відобразиться:

```
Assembly successfully added to the cache
```

Після додавання до папки C:\Windows\Microsoft.NET\assembly\GAC_MSIL можна знайти додану збірку - для неї буде створено окремий каталог, як і для інших збірок, який буде носити коротке ім'я збірки.

Тепер можна використовувати цю збірку із GAC. Для цього створимо якийсь проект і у вікні Solution Exlplorer (Обозреватель рішень) натиснемо на вузол Referenses (Посилання). У меню виберемо Add Reference... (Додати посилання):



У вікні додавання посилання на складання натиснемо внизу кнопку Browse (Огляд) і знайдемо в GAC цю збірку. Після цього вона буде додана до проекту, і її можемо використати.

5.3. Контрольні питання

1. Роль збірок у додатках .NET
2. Що таке файл збирання програми?
3. Що містить маніфест збирання?
4. Що таке метадані типів?
5. Що таке атрибути збирання?
6. Як змінюються та задаються атрибути складання?
7. Що таке складання, що розділяються?
8. Що необхідно для розміщення складання в GAC?
9. Що таке суворе ім'я збирання?
10. Як створити суворе ім'я збирання?

Література

1. Зеленський О. С., Лисенко В. С. Розробка Windos-додатків на мові C# : навч. посіб. Ч. 1. Кривий Ріг : Держ. ун-т економіки і технологій, 2023. 160 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi78/0058684.pdf.
2. Зеленський О. С., Лисенко В. С. Розробка Windos-додатків на мові C# : навч. посіб. Ч. 2. Кривий Ріг : Держ. ун-т економіки і технологій, 2023. 357 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi78/0058674.pdf.
3. Бичков О. С., Іванов Є. В. Об'єктно-орієнтоване програмування мовою C# : підручник. Київ : Київ. ун-т, 2018. 208 с.
4. Решевська К. С., Лісняк А. О., Борю С. Ю. Об'єктно-орієнтоване програмування : навч. посіб. Запоріжжя : ЗНУ, 2020. 94 с. URL: http://ebooks.znu.edu.ua/files/metodychky/2020/06/0045039.doc.
5. Бобков В. Б., Грудзинський Ю. Є., Крилов К. В. Програмування - 2. Об'єктно-орієнтоване програмування : навч. посіб. Київ : КПІ ім. Ігоря Сікорського, 2023. 77 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi84/0063780.pdf.
6. Гришанович Т. О., Глинчук Л. Я. Основи об'єктно-орієнтованого програмування : навч. посіб. / ВНУ ім. Лесі Українки. Луцьк : ВНУ ім. Лесі Українки, 2022. 120 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi70/0051132.pdf.
7. Дібрівний О. А., Гребенюк В. В. Вступ до об'єктно орієнтованого програмування C# : навч. посіб. Київ : Державний університет телекомунікацій, 2018. 190 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi73/0053671.pdf.
8. Олефіренко Н. В., Курганський А. Р., Остапенко Л. П., Пономарьова Н. О. Об'єктно-орієнтоване програмування мовою C# : навч. посіб. Харків : ХНПУ ім. Г. С. Сковороди, 2024. 254 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi84/0063778.pdf.