


# Робота з масивами C#

# Масиви в С#

- ▶ Масиви в С# зі сторони синтаксису практично не відрізняються від масивів в С, С++ і Java.
- ▶ Однак внутрішньо масив в С# влаштований як тип даних, похідний від класу `System.Array`.
- ▶ Формально масив визначається як набір елементів, доступ до яких виконується за допомогою числового індексу.

# Оголошення масиву

- ▶ Масиви оголошуються шляхом поміщення квадратних дужок ([ ]) після типу даних для елементів цього масиву.



```
// Масив символних рядків із 10 елементами {0, 1,..., 9}  
string[] booksOnCOM;  
booksOnCOM = new string[10];  
// Масив символних рядків із 2 елементами {0, 1}  
string[] booksOnPLI = new string[2];  
// Масив символних рядків із 100 елементів {0, 1,..., 99}  
string[] booksOnDotNet = new string[100];
```




Таке оголошення масива приведе до помилки компілятора:

// При визначенні масиву фіксованого розміру

// ми зобов'язані використовувати ключове слово **new**

int[4] ages = {30, 54, 4, 10}; // **Помилка!**

Розмір масива задається при його створенні, а не оголошенні.



```
// Буде автоматично створений масив із 4 елементів.  
// Зверніть увагу на відсутність ключового слова new  
// і на порожні квадратні дужки  
int[] ages = {20, 22, 23, 0};
```

# Заповнення масива в С#

- ▶ Заповнити масив можна, перерахуваючи елементи послідовно у фігурних дужках.
- ▶ А можна використовувати для цієї цілі числові індекси.

// Використовуємо послідовне  
//перечислення елементів масиву:

```
string[] firstNames = new string[5] {"Steve",  
"Gina", "Swallow", "Baldy", "Gunner"};
```

// Використовуємо числові індекси:

```
string[] firstNames = new string[5];  
firstNames[0] = "Steve";  
firstNames[1] = "Gina";  
firstNames[2] = "Swallow";  
firstNames[3] = "Baldy";  
firstNames[4] = "Gunner";
```




# Важлива відмінність між масивами C++ та C#

- ▶ В C# елементам масиву автоматично присвоюється значення за замовчуванням у залежності від використовуваного для них типу даних.


Наприклад, для масиву цілих чисел всім елементам буде на самому початку присвоєно значення 0, для об'єктів — значення NULL і т. д.


# Багатовимірні масиви

- ▶ Окрім масивів із одним виміром в С# підтримуються також дві основні різновидності багатовимірних масивів.
- ▶ Перший різновид багатовимірних масивів інколи називають **"прямокутним масивом"**. Такий тип масиву утворюється простим складанням декількох вимірів. При цьому всі рядки і стовпці у даному масиві будуть однакової довжини.




```
// Прямокутний багатовимірний масив
int[,] myMatrix;
myMatrix = new int[6, 6];
// Заповнюємо масив 6 на 6
for (int i = 0; i < 6; i++)
    for (int j = 0; j < 6; j++)
        myMatrix[i, j] = i*j;
// Виводимо елементи багатовимірного масиву на
//системну консоль
for (int i = 0 ; i < 6; i++)
{
    for (int j = 0; j < 6; j++)
    {
        Console.Write(myMatrix[i, j] + "\t");
    }
    Console.WriteLine();
}
```

- 
- ▶ Другий тип багатовимірного масиву можна назвати «**ламаним**» (jagged). Такий масив містить у якості внутрішніх елементів деяку кількість внутрішніх масивів, кожен із яких може мати свій унікальний внутрішній розмір.



```
// «Ламаний» багатовимірний масив (масив
// із масивів). У нашому випадку – це масив
// із п`яти внутрішніх масивів різного
// розміру
int[,] myJagArray = new int[5][]:
// Створюємо «ламаний» масив
for (int i = 0; i < myJagArray. Length; i++)
{
myJagArray[i] = new int[i + 7];
}
```



```
// Виводимо кожний рядок на системну консоль
// (як ми пам`ятаємо, кожному елементу
// присвоюється значення по замовчуванню – у нашому
// випадку 0)
for (int i = 0; i < 5; i++)
{
    Console.Write("Length of row {0} is {1}:\t", i,
myJagArray[i].Length);
    for (int j = 0; j < myJagArray[i].Length; j++)
    {
        Console.Write(myJagArray[i,j] + " ");
    }
    Console.WriteLine();
}
```

# Базовий клас `System.Array`

- ▶ Всі найбільш важливі відмінності між масивами в C++ та C# полягають у тому, що в C# всі масиви є похідними від базового класу **`System.Array`**. За рахунок цього будь-який масив в C# успадковує велику кількість корисних методів і властивостей, які сильно спрощують роботу програміста.



# Таблиця методів

Метод класу	Призначення
BinarySearch()	Цей статичний метод можна використовувати тільки тоді, коли масив реалізує інтерфейс <code>IComparer</code> , у цьому випадку метод, дозволяє знайти елемент масиву.
Clear()	Цей статичний метод дозволяє очистити діапазон елементів (числові елементи отримають значення 0, а посилання на об'єкти – null).
CopyTo()	Використовується для копіювання елементів із вихідного масиву в масив призначення.
GetEnumerator()	Повертає інтерфейс <code>IEnumerator</code> для вказаного масиву.



Метод класу	Призначення
<code>GetLength()</code> <code>.Length</code>	Метод <code>GetLength()</code> використовується для визначення кількості елементів у вказаному вимірі масиву. <code>Length</code> – це властивість тільки для читання, за допомогою якого можна отримати кількість елементів масиву.
<code>GetLowerBound()</code> <code>GetUpperBound()</code>	Ці методи використовуються для визначення верхньої і нижньої границі обраного вами виміру масиву.
<code>GetValue()</code> <code>SetValue()</code>	Повертає або встановлює значення вказаного індексу для масиву. Цей метод перевантажений для нормальної роботи як і з одновимірними, так і з багатовимірними масивами.
<code>Reverse()</code>	Цей статичний метод дозволяє розставляти елементи одновимірного масиву у зворотному порядку.
<code>Sort()</code>	Сортує одновимірний масив вбудованих типів даних. Якщо елементи масиву підтримують інтерфейс <code>IComparer</code> , то за допомогою цього методу ви зможете виконувати сортування і ваших користувацьких типів даних.

# Приклад

```
// Створюємо декілька масивів символьних
// рядків і експериментуємо із членами
// System.Array
class Arrays
{
    public static int Main(string[] args)
    {
        // Масив символьних рядків
        string[] firstNames = new string[5];
        firstNames = {"Steve", "Gina", "Swallow",
            "Baldy", "Gunner"};
    }
}
```

// Виводимо імена у відповідності із  
порядком

// елементів у масиві

```
Console.WriteLine("Here is the array:");
```

```
for (int i = 0; i < firstNames.Length; i++)
```

```
    Console.Write(firstNames[i] + "\t");
```

// Розставляємо елементи в оберненому

// порядку за допомогою статичного

// методу Reverse()



```
Array.Reverse(firstNames );
```

```
// ...і знову виводимо імена
```

```
Console.WriteLine("Here is the array once  
reversed:");
```

```
for (int i = 0; i < firstNames.Length; i++)
```

```
    Console. Write(firstNames[i] + "\t");
```


```
// А тепер видаляємо всіх, крім
```

```
// Steve
```

```
Console.WriteLine("Cleared out all but  
one...");
```

```
Array.Clear(firstNames, 1, 4);
```

```
for (int i = 0; i < firstNames.Length; i++)  
{  
    Console.WriteLine(firstNames[i] + "\t\n");  
}  
return 0;  
}  
}
```



# Робота з рядками в C#

# Рядки в C#

- ▶ `String` (рядки Unicode) — це вбудований тип даних C#.
- ▶ Всі рядки у світі C# і .NET походять від одного базового класу — `System.String`.
- ▶ `System.String` забезпечує безліч методів, які дозволяють виконувати за вас всю чорну роботу: повернути кількість символів у рядку, знайти підрядок, перетворити всі символи у рядкові або прописні і т.д.

Метод класу	Призначення
Length	Ця властивість повертає довжину вказаного рядку.
Concat()	Цей статичний метод, класу String повертає новий рядок, «склеєний» із двох вихідних.
CompareTo()	Порівнює два рядки.
Copy()	Цей статичний метод створює нову копію існуючого рядка.
Format()	Використовується для форматування рядка з використанням інших примітивів (числових даних, інших рядків) і виразів для підстановки виду {0}.
Insert()	Використовується для внесення рядка всередину існуючого.
PadLeft() PadRight()	Ці методи дозволяють заповнити («набити») рядок вказаними символами.
Remove() Replace()	Ці методи дозволяють створити копію рядка із внесеними змінами (видаленими або заміненними символами).
ToUpper() ToLower()	Ці методи використовуються для отримання копії рядка, в якому всі символи стануть рядковими або прописними.



# Зверніть увагу

- ▶ Хоч *string* — це тип посилання даних, при використанні операторів рівності (`==` і `!=`) відбувається порівняння значень рядкових об'єктів, а не адрес цих об'єктів в оперативній пам'яті.
- ▶ Оператор складання (+) в C# перевантажений таким чином, що при використанні до рядкових об'єктів він викликає метод `Concat()`.

# Керуючі послідовності

- ▶ В С#, як і в С, і в С++, і в Java рядки можуть містити будь-яку кількість керуючих послідовностей (escape characters).

# Приклад

```
// Застосування керуючих
//послідовностей - \t, \\", \n та інших
string anotherString;
anotherString = "Every programming book
    need \"Hello World\"";
Console.WriteLine("\t" + anotherString);

anotherString =
    "c:\\CSharpProjects\\Strings\\string.cs";
Console.WriteLine("\t" + anotherString);
```

Керуючі послідовності	Призначення
\'	Вставити одинарні лапки в рядок
\"	Вставити подвійні лапки в рядок
\\	Вставити в рядок зворотній слеш. Особливо корисно при роботі із шляхом у файлових системах.
\a	Запустити системне сповіщення (Alert).
\b	Повернутися на одну позицію (Backspace).
\f	Почати наступну сторінку (Form feed).
\n	Вставити новий рядок (New line).
\r	Вставити повернення каретки (carriage Return).
\t	Вставити горизонтальний символ табуляції (horizontal Tab).
\u	Вставити символ Unicode.
\v	Вставити вертикальний символ табуляції (Vertical tab).
\0	Порожній символ (NULL).

# Виведення службових СИМВОЛІВ

- Крім керуючих послідовностей, в C# передбачений також спеціальний префікс **@** для дослівного виведення рядків незалежно від наявності у них керуючих послідовностей. Рядки із цим префіксом називаються «дослівними» (verbatim strings). Це — дуже зручний засіб у багатьох ситуаціях:

// Префікс @ вимикає обробку

// керуючих послідовностей

```
string finalString = @"\n\tString file:  
'C:\CSnarpProjects\Strings\string.cs';  
Console.WriteLine(finalString);
```

# Використання `System.Text.StringBuilder`

- ▶ При роботі з рядками в C# необхідно пам'ятати дуже важливу річ: **значення рядка не може бути зміненим.**
- ▶ Всі методи, здавалося б, які змінюють рядок, насправді лише повертають її змінену копію.

# Приклад

// Вносимо зміни в рядок? Насправді

// ні...

```
System.String strFixed = "This is how I began  
life";
```


```
Console.WriteLine(strFixed);
```

```
string upperVersion = strFixed.ToUpper();
```

// Повертає «прописну» копію strFixed

```
Console.WriteLine(strFixed);
```


```
Console.WriteLine(upperVersion);
```

- 
- ▶ Робота з копіями копій може набриднуть. Тому в C# існує клас, який дозволяє змінювати рядки напряму – це клас `StringBuilder`, визначений у просторі імен `System.Text`. Він багато в чому нагадує `CString` в MFC.
  - ▶ Всі зміни, які ви вносите в об'єкт цього класу, швидко у ньому відображаються, що в більшості випадків більш ефективно, ніж працювати із множиною копій.



```
// Демонстрація використання класу StringBuilder
using System;
using System.Text; // Тут проживає StringBuilder!
class StringApp
{
    public static int Main(string[] args)
    {
        // Створюємо об'єкт StringBuilder та змінюємо
        // його вміст
        StringBuilder myBuffer = new StringBuilder("I am a
buffer");
        myBuffer.Append(" that just got longer...");
        Console.WriteLine(myBuffer);
        return 0;
    }
}
```

- ▶ Окрім додавання клас `StringBuilder` дозволяє і інші операції, наприклад видалення певних символів або їх заміна.
- ▶ Після того як ви отримали потрібний вам результат, для зручності можна викликати метод `ToString()`, щоб перевести вміст об'єкту `StringBuilder` у звичайний тип даних `String`.



```
using System;
using System.Text;
class StringApp
{
    public static int Main(string[] args)
    {
        StringBuilder myBuffer = new StringBuilder("I am a
buffer");
        myBuffer.Append(" that just got longer...");
        Console.WriteLine(myBuffer) ;
        myBuffer.Append("and even longer.");
        Console. WriteLine(myBuffer);
        // Змінюємо всі букви на прописні
        string theReallyFinalString = myBuffer.ToString().
ToUpper();
        Console.WriteLine(theReallyFinalString);
        return 0; } }
```



# Перерахування в C#


- Часто буває зручним створити набір значущих імен, які будуть представляти числове значення.

// Створюємо перерахування

```
enum EmpType  
{  
    Manager, // = 0  
    Grunt,    // = 1  
    Contractor, // = 2  
    VP        // = 3  
}
```

// Елементи перерахування можуть мати  
// довільні числові значення

```
enum EmpType  
{  
    Manager = 10,  
    Grunt = 1,  
    Contractor = 100,  
    VP = 99  
}
```

- 
- ▶ При компіляції компілятор просто підставляє замість елементів перерахування відповідні числові значення.
  - ▶ По замовчуванню для цих числових значень компілятор використовує тип даних `Int`. Однак ніщо не заважає явним чином оголосити компілятору, що потрібно використовувати інший тип даних, наприклад, `byte`.

// Замість елементів перерахування будуть  
// підставлятися числові значення типу byte

```
enum EmpType : byte
```

```
{
```

```
    Manager = 10,
```

```
    Grunt = 1,
```

```
    Contractor = 100,
```

```
    VP = 9
```

```
}
```

- Точно таким же чином можна використовувати будь-який із основних цілочисельних типів даних C# (byte, sbyte, short, ushort, int, uint, long, ulong).



# Базовий клас System.Enum

- ▶ Всі перерахунки в С# утворюються від єдиного базового класу System.Enum. Звичайно ж, у цьому базовому класі передбачені методи, які можуть істотно спростити роботу з перерахунками.

# GetUnderlyingType ( )

- Це статичний метод, який дозволяє отримувати інформацію про те, який тип даних використовується для представлення числових значень елементів перерахування:

```
// Отримуємо тип числових даних  
// перерахування (у нашому прикладі це буде  
// System.Byte)  
Console.WriteLine(Enum.  
GetUnderlyingType(typeof(EmpType)));
```

# Enum.Format ( )

- ▶ Статичний метод, який може отримувати значущі імена елементів перерахування по їх числовим значенням.
- ▶ У нашому прикладі змінній типу EnumType відповідало ім'я елементу перерахування Contractor (тобто ця змінна перетворювалась у числове значення 100).
- ▶ Для того щоб дізнатися, якому елементу відповідає це числове значення, необхідно викликати метод Enum.Format, вказати тип перечислення.

Числове значення (у нашому випадку  
через змінну) і прапорець  
форматування (у нашому випадку — G,  
що значить вивести як тип string)

```
// Цей код виводить на системну
```

```
// консоль рядок "You are a Contractor"
```

```
EmpType fred;
```

```
fred = EmpType.Contractor;
```

```
Console.WriteLine("You are a {0}", Enum.Format(typeof(  
    EmpType), fred, "G"));
```


# GetValues()

- ▶ Статичний метод, який повертає екземпляр `System.Array`, при цьому кожному елементу масиву буде відповідати член вказаного перерахування.

```
// Отримуємо інформацію про  
кількість
```

```
// елементів в перерахуванні
```

```
Array obj =  
Enum.GetValues(typeof(EmpType));  
Console.WriteLine("This enum has {0}  
members.", obj.Length);
```



```
// А тепер виводимо імена елементів
// перерахування у форматі string і
// відповідні їм числові значення
foreach(EmpType e in obj)
{
    Console. Write("String name: {0}", Enum.
    Format (typeof (EmpType), e, "G"));
    Console. Write(" ({0})", Enum. Format(typeof(
    EmpType), e, "D"));
    Console. Write(" hex: {0}\n", Enum. Format (
    typeof (EmpType), e, "X"));
}
```

# Результат роботи програми

```
1 using System;
2 enum EmpType : byte {
3     Manager = 10,
4     Grunt = 1,
5     Contractor = 100,
6     VP = 9
7 }
8 public class Program
9 {
10     public static void Main()
11     {
12         Array obj = Enum.GetValues(typeof(EmpType));
13         Console.WriteLine("This enum has {0} members.", obj.Length);
14         foreach(EmpType e in obj) {
15             Console.WriteLine("String name: {0}", Enum.Format(typeof(EmpType), e, "G"));
16             Console.WriteLine("({0})", Enum.Format(typeof(EmpType), e, "D"));
17             Console.WriteLine("hex: {0}\n", Enum.Format(typeof(EmpType), e, "X"));
18         }
19     }
20 }
```

```
This enum has 4 members.
String name: Grunt (1) hex: 01
String name: VP (9) hex: 09
String name: Manager (10) hex: 0A
String name: Contractor (100) hex: 64
```



# IsDefined

- ▶ Властивість класу `System.Enum`, яка дозволяє визначити, чи є обраний вами символьний рядок елементом вказаного перерахування.
- ▶ Наприклад, припустимо, що потрібно дізнатися, чи є значення `Salesperson` елементом перерахування `EmpType`.





```
// Чи є в EmpType елемент
```

```
// Salesperson?
```

```
if (Enum.IsDefined(typeof (EmpType), "Salesperson"))
```

```
    Console.WriteLine("Yes, we have sales people.");
```

```
else
```

```
    Console.WriteLine("No, we have no profits...");
```

- ▶ Перерахування C# підтримують роботу з великою кількістю перевантажених операторів, які можуть виконувати різні операції з числовими значеннями змінних.

```
// Якому із цих двох змінних-членів
```

```
// перерахування відповідає більше числове  
значення?
```

```
EmpType Joe = EmpType.VP;
```

```
EmpType Fran = EmpType.Grunt;
```

```
if (Joe < Fran)
```

```
    Console.WriteLine("Joe's value is less than Fran's");
```


```
else
```

```
    Console.WriteLine("Fran's value is less than Joe's");
```

# Структури в C#


# Визначення структур в C#

- ▶ Структури C# можна розглядати як деякий особливий різновид класів.
- ▶ Для структур можливо створювати конструктори (тільки приймають параметри).
- ▶ Структури можуть реалізувати інтерфейси.


- 
- ▶ Структури можуть містити будь-яку кількість внутрішніх членів.
  - ▶ Для структур C# не існує єдиного базового класу (тип `System.Structure` в C# не передбачений).
  - ▶ Опосередковано всі структури є похідними від типу `ValueType`.

# Приклад

```
// Спочатку нам знадобиться наше  
// перерахування  
enum EmpType : byte  
{  
    Manager = 10, Grunt = 1, Contractor =  
    100,  
    VP = 9  
}  
struct EMPLOYEE  
{
```



```
public EmpType title; // Одне із полів структури –  
// перерахування, визначене вище  
public string name;  
public short deptID;  
}  
class StructTester  
{  
    public static int Main(string[] args)  
    {  
        // Створюємо і присвоюємо значення fred'у  
        EMPLOYEE fred;  
        fred.deptID = 40;  
        fred.name = "Fred";  
        fred.title = EmpType.Grunt;  
        return 0;  
    }  
}
```

- 
- ▶ Цілком можливо, що в реальному застосунку для більш зручного присвоєння значення членам структури доведеться визначити свій власний конструктор або декілька конструкторів.
  - ▶ Потрібно пам'ятати, що не можна перевизначать конструктор для структури по замовчуванню – той конструктор, який не приймає параметрів.



- 
- ▶ Всі ваші конструктори обов'язково повинні приймати один або декілька параметрів.

// Для структур можна визначити

// конструктори, але всі вони повинні


// приймати параметри

```
struct EMPLOYEE
```


```
{
```

// Поля

```
public EmpType title;
```



```
public string name;  
public short deptID;  
// Конструктор  
public EMPLOYEE (EmpType et, string n,  
short d)  
{  
    title = et;  
    name = n;  
    deptID = d;  
}  
}
```

- 
- За допомогою такого визначення структури, у якому передбачений конструктор, ви можете створювати нових співробітників наступним чином:


```
class StructTester
```

```
{
```


```
// Створюємо mary і присвоюємо їй
```

```
// значення за допомогою конструктору
```

```
public static int Main(string[] args)
```



```
{  
    // Для виклику нашого  
    // конструктора ми зобов'язані  
    // використовувати ключове слово  
    // new  
    EMPLOYEE mary = new EMPLOYEE(EmpType.VP, "Mary",  
10);  
    return 0;  
}  
}
```

- 
- Структури можуть бути використані у якості вхідних та вихідних методами параметрів.

# Упакування та розпакування

- ▶ Упакування та розпакування – це найбільш зручний спосіб перетворення структурного типу у вказівник, і навпаки.
- ▶ Основне призначення структур – можливість отримання деяких переваг об'єктної орієнтації, але при більш великій продуктивності за рахунок розміщення у стеці.

- ▶ Щоб перетворити структуру у вказівник на об'єкт, необхідно упакувати її екземпляр:

```
// Створюємо та пакуємо нового
```

```
// співробітника
```

```
EMPLOYEE stan = new
```

```
EMPLOYEE(EmpType.Grunt, "Stan", 10);
```


```
object stanInBox = stan;
```

```
//stanInBox відноситься до адресних типів
```

```
// даних, але при цьому зберігає
```

```
// внутрішні значення вихідного типу
```

```
// даних EMPLOYEE
```

- 
- Можна використовувати `stan` у всіх випадках, коли потрібний об'єкт, і при необхідності виконувати розпакування:

//Так як ми раніше виконали


//пакування даних, ми можемо

//розпакувати їх і виконувати операції

//із вмістом

```
public void UnboxThisEmployee(object o)
```





```
{  
    // Виконуємо розпакування в структуру  
    // EMPLOYEE для отримання доступу  
    // до всіх полів  
    EMPLOYEE temp = (EMPLOYEE)o;  
    Console.WriteLine(temp.name + "is alive!");  
}
```

- 
- ▶ Виклик цього методу може мати наступний вигляд:

```
// Передаємо запакованого співробітника на  
// обробку
```

```
t.UnboxThisEmployee(stanInBox);
```

- ▶ Компілятор C# при необхідності автоматично виконує упаковання. Тому дозволяється передача об'єкту stan (типу EMPLOYEE) напряму:

```
// Stan буде упакований автоматично
```

```
t.UnboxThisEmployee(stan);
```



# Користувачський простір імен в C#


# Створення користувацького простору імен

- ▶ Часто буває дуже корисним згрупувати використовувані у програмі типи даних у спеціально створений для цієї цілі простір імен.
- ▶ В С# ця операція виконується за допомогою ключового слова `namespace`.

# Приклад

```
// shapeslib.cs
namespace MyShapes
{
    using System;
    // Клас Circle
    public class Circle { // Цікаві методи }
```

```
public class Hexagon // Клас Hexagon
{
    // Більш цікаві методи
}
public class Square // Клас Square
{
    // Ще більш цікаві методи
}
}
```

- 
- ▶ Простір імен `MyShapes` діє як контейнер для всіх цих типів.
  - ▶ Можна розбити єдиний простір імен `C#` на декілька фізичних файлів. Для цього достатньо просто визначити у різних файлах один і той же простір імен і помістити у ньому визначення класів.



```
// circle.cs
```

```
namespace MyShapes
```

```
{
```

```
    using System;
```

```
    // Клас Circle
```

```
    class Circle { // Цікаві методи }
```

```
}
```

Подібним чином визначаються `hexagon.cs` і `square.cs`.




- ▶ Якщо потрібно буде використовувати ці класи всередині іншого застосунку, зручніше всього це зробити за допомогою ключового слова **using**:

```
namespace MyApp
{
    using System;
    using MyShapes;
    class ShapeTester
    {
        public static void Main()
        {
            // Всі ці об'єкти були визначені
            // у просторі імен MyShapes
            Hexagon h = new Hexagon();
            Circle c = new Circle();
            Square s = new Square();
        }
    }
}
```

# Використання простору імен для вирішення конфлікту між іменами класів

- ▶ Простір імен може бути використаний для вирішення конфлікту між іменами об'єктів у тих ситуаціях, коли у нашому застосунку використовуються різні об'єкти з однаковими іменами.



```
// Ще один простір імен для
// геометричних фігур
namespace My3DShapes
{
    using System;
    // Клас 3D Circle
    class Circle{}
    // Клас 3D Hexagon
    class Hexagon{}
    // Клас 3D Square
    class Square{}
}
```

// У коді є двозначність!

namespace MyApp

```
{
    using System;
    using MyShapes;
    using My3DShapes;
    class ShapeTester
    {
        public static void Main()
        { // Невідомо, до об'єктів якого простору імен ми
          //звертаємося
          Hexagon h = new Hexagon();
          Circle c = new Circle();
          Square s = new Square();
        }
    }
}
```

- ▶ Простіше всього позбутися від подібних конфліктів, вказавши для кожного класу його повне ім'я разом з ім'ям відповідного простору імен:


// Конфлікт вирішено

```
public static void Main()  
{  
    My3DShapes.Hexagon h = new  
        My3DShapes.Hexagon();  
    My3DShapes.Circle c = new  
        My3DShapes.Circle();  
    My3DShapes.Square s = new  
        MyShapes.Square();  
}
```

# Використання псевдонімів для імен класів

- ▶ Ще одна можливість позбутися від конфліктів імен – використовувати для імен класів псевдоніми.

```
namespace MyApp  
{  
    using System;  
    using MyShapes;  
    using My3DShapes;
```



```
// Створюємо псевдонім для класу з  
// іншого простору імен  
using The3DHexagon = My3DShapes.Hexagon  
;  
class ShapeTester  
{  
    public static void Main()  
    {  
        Hexagon h = new Hexagon();  
        Circle c = new Circle();  
        Square s = new Square();
```



```
// Створюємо об'єкт за допомогою
```

```
// псевдоніма
```

```
    The3DHexagon h2 = new The3DHexagon();
```

```
    }
```

```
}
```

```
}
```



# Вкладені простори імен

- ▶ Можна без будь-яких обмежень вкладати один простір імен в інший.
- ▶ Такий спосіб дуже часто використовується у бібліотеках базових класів .NET для організації типів.

# Приклад

```
// Класи для геометричних фігур  
// розташовані у просторі імен  
// Chapter2Types.My3DShapes  
namespace Chapter2Types  
{  
    namespace My3DShapes  
    {  
        using System;
```

// Κλας 3D Circle

class Circle{

// Κλας 3D Hexagon

class Hexagon{

// Κλας 3D Square

class Squared

}

}