C# : Визначення типів за допомогою класів

Клакович Л.М.

1. Способи визначення типів: клас, структура, інтерфейс.
2. Огляд членів класу
3. Модифікатори доступу
4. Поля класу. Ініціалізація
5. Поля `readonly` та `const`
6. Статичні члени класу
7. Конструктори
8. Методи класу

Визначення типів

Загальна схема визначення користувацьких типів: класів, структур та інтерфейсів

атрибути $_{opt}$ *модифікатори* $_{opt}$
class *ідентифікатор* : *ідентифікатор базового класу* $_{opt}$, *список інтерфейсів* $_{opt}$ {
визначення членів класу
}

атрибути $_{opt}$ *модифікатори* $_{opt}$
struct *ідентифікатор*: *список інтерфейсів* $_{opt}$ {
визначення членів типу
}

Неявне наслідування від
`System.ValueType`.

атрибути $_{opt}$ *модифікатори* $_{opt}$
interface *ідентифікатор*: *список інтерфейсів* $_{opt}$ {
оголошення членів інтерфейсу
}

Модифікатор:
- **new, abstract, sealed**

Модифікатори доступу:
- **public, protected, internal, private**

Члени класу

- **Поля (fields)** – змінні, які зберігають **дані про стан об'єкта**. Модифікатори ***static***, ***readonly*** і ***const*** визначають спосіб використання цих даних.
- **Методи (methods)** – код, який задає дії над даними об'єктів (функціональність).
- **Властивості (properties)** – методи, які з точки зору клієнта класу виглядають як поля, а насправді надають варіанти опосередкованого доступу до даних об'єкта; реалізуються аксесорними методами ***get*** і ***set***.
- **Індексатори (indexers)** – подібні до методів засоби доступу до полів об'єкта як до елементів деякого масиву за значенням порядкового індексу; реалізуються аксесорними методами ***get*** і ***set***.
- **Делегати(delegates)** – типи, спеціалізовані ***delegate***; їхні екземпляри інкапсулюють список виклику, елементи якого(один або кілька) є методами, доступними для виклику. Тобто делегати є засобом опосередкованого виклику методів, причому на момент компілювання мають бути заданими лише типи результату і параметрів, а конкретні реалізації методів можуть призначатися в процесі виконання програми.
- **Події (events)** – спеціалізовані ***events*** екземпляри-делегати; разом з оголошенням свого типу-делегата вони публікуються (***public-***специфіковані) класом, об'єкти якого при виконанні певних умов через делегати-події викликтимуть потрібний метод (обробітник події) іншого класу, який в своїх методах підписався на конкретну подію.
- **Оператори (operators)** – перевантажені для класу стандартні оператори, які дають змогу використовувати об'єкти класу подібно до об'єктів вбудованих типів.
- **Вкладені (nested) типи** – як правило, private-специфіковані для створення об'єктів лише для внутрішнього використання.

Модифікатори доступу

- **Визначають рівень доступу *(accessibility level)* до члена класу**
  - ***public*** – необмежений доступ; член класу доступний як в методах свого класу, так і поза визначенням класу
  - ***protected*** – член класу недоступний скрізь поза визначенням класу крім ієрархії похідних класів
  - ***private*** – член класу може використовуватися лише в методах свого класу і є недоступним поза визначенням класу
  - ***internal*** – поза визначенням класу член класу доступний лише в поточній збірці компілювання, де знаходиться визначення класу
  - ***protected internal*** – член класу доступний в поточній збірці, де знаходиться визначення класу, або в ієрархії похідних класів
- **Визначають область доступності *(accessibility domain)* визначень типів і їхніх членів**
- **Модифікатор для всіх типів і їх членів задається явно або неявно (за замовчуванням).**

За замовчуванням:

типи верхнього рівня – ***public***

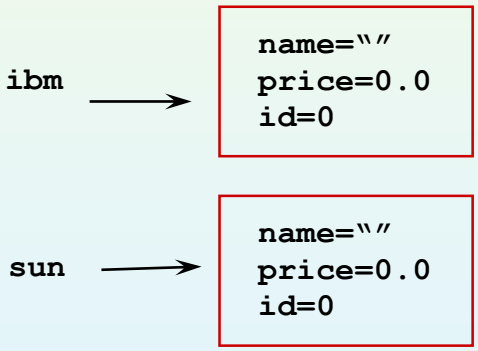
вкладені типи і члени типів – ***private***

Створення об”єктів

- Оператор **new** при створенні об’єкту класу:
  - В heap виділяється пам”ять під об’єкт (для всіх полів + додаткова інформація) і адреса цієї пам’яті присвоюється змінній
  - Викликається відповідний конструктор

```
class Product
{
  string name;
  double price;
  uint    id;
  ...
}
```

```
Product ibm = new Product();
Product sun = new Product();
```



Поля класу. Ініціалізація

- **Полям присвоюється значення за замовчуванням при створенні об”єкта:**
  - `0` для числових типів
  - `false` для **`bool`**
  - `'\x0000'` для **`char`**
  - `null` для посилань
- **Допускається ініціалізація полів при оголошенні**
- **Природньо ініціалізувати в конструкторах**

instance fields →

```
class Rational
{
   int numerator;
   int denominator=1;
   ...
}
```

a

numerator	0
denominator	1

set to default values →

```
Rational a = new Rational();
```

Поля readonly та const

[<модифікатор доступу>] [static] [const | readonly]

<Тип> <ідент. Змін.> [=<Вираз1>][,...]

- **readonly** – тільки для читання.
  Ініціалізація – в конструкторі, далі – незмінна.
  Може бути довільного типу.
- **const** – за замовчуванням – статичні. Лише вбудованих типів, `string` та `enum`.
  Ініціалізація при оголошенні.

```
class MagicNumbers
{
        public const double pi = 3.1415;
        public readonly int YorNumb;
        public MagicNumbers (int init) { YorNumb=init;}
}
```

```
MagicNumbers mg=new MagicNumbers(45);
Console.WriteLine("pi = {0}, everything else = {1}",
                  MagicNumbers.pi, mg.YouNumb);
```

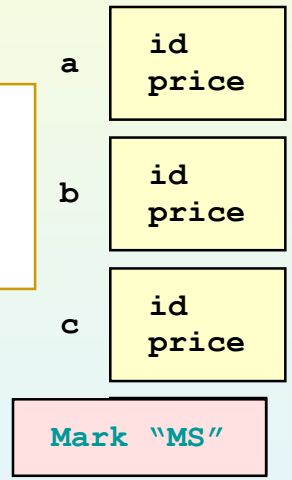
Статичні члени класу

- **Статичні поля**
  - Це дані, спільні для всіх об’єктів цього класу, зберігаються в одному екземплярі
  - Утворення та ініціалізація відбувається при завантаженні програми (за замовчуванням, при оголошенні, статичним конструктором)
  - Доступ через екземпляр класу заборонений
  - Константи є статичними полями
  - Існують константи до кінця виконання програми

```
class Product
{
  private int id;
  private double price;

  public static string Mark;
}
```

```
Product a = new Product();
Product b = new Product();
Product c = new Product();
```




```
Product.Mark = “MS”;
```

Статичні члени класу

Статичні методи

- Виклик за назвою класу (без врахування існування об'єктів)
- Можуть використовувати лише статичні дані класу та статичні методи
- Статичний конструктор викликається до утворення першого об'єкта класу, першого виклику статичного методу чи доступу до статичного поля
- Main() – статичний метод

```
public class A{
   public static int X=10;
   public const int Y=20;
   private static int Z;
   public static A(){ z=110;}
    public static void f(){
       Console.WriteLine
       ("X={0},Y={1},Z={2}",X,Y,Z);
    }
}
```

```
public class B{
    public static void Main(){
       Console.WriteLine("A.X={0}",A.X);
      A.f();
    }
}
```

```
public class Math
{
  public static double Sqrt(double d) ...
  public static double Sin (double a) ...
  public static double Log (double d) ...
  ...
}
```

Конструктори

- **Визначення конструктора**

  модифікатори_доступу **ід класу (*список параметрів*$_{opt}$) {....}**

- **Утворення об'єкта**

  ід класу ід об'єкта = new ід класу (*список аргументів*$_{opt}$);

- **Ініціалізатори:** **this(...), base(...)**

- **Особливості C#**
  - може бути кілька
  - лише при повній відсутності компілятор створить конструктор за замовчуванням
  - особливості визначення в структурах
- **Статичний к-тор** (закритий за замовчуванням)

  static ід класу () {....}
- **Закриті к-тори (private)**
- **Ініціалізація readonly полів**

```
using System;
class ConstructorApp{
    ConstructorApp(){Console.WriteLine("I'm the c-tor");}
    public static void Main(){
      ConstructorApp app = new ConstructorApp();
    }
}
```

class Point_s

```
class Point_s {
   public Point_s(int xx, int yy){
      x = xx;
      y = yy;
      Console.WriteLine("int,int c-tor: "+ x+", "+y+ ")“);
   }
   public Point_s(int x):this(x,0){
      Console.WriteLine("int c-tor: "+x+", "+ y + ")“);
   }
   public Point_s(){
      Console.WriteLine("void c-tor: "+x+", " + y + ")“);
   }
      ....
   private int x=0;
   private int y=0;
}
```

Властивості - Properties

Функціональні члени, які використовуються як поля

модифікатори$_{opt}$ *тип* *ідентифікатор* **{** *методи доступу* **}**

- Властивості містять переваги public і private даних
  - мають прозорий синтаксис доступу як public поля
  - забезпечують можливість виявлення помилок як private поля

Оголошення **Name** властивості

реалізація

```
class Person
{
  public string Name
  {
    ...
  }
  private string name;
}
```

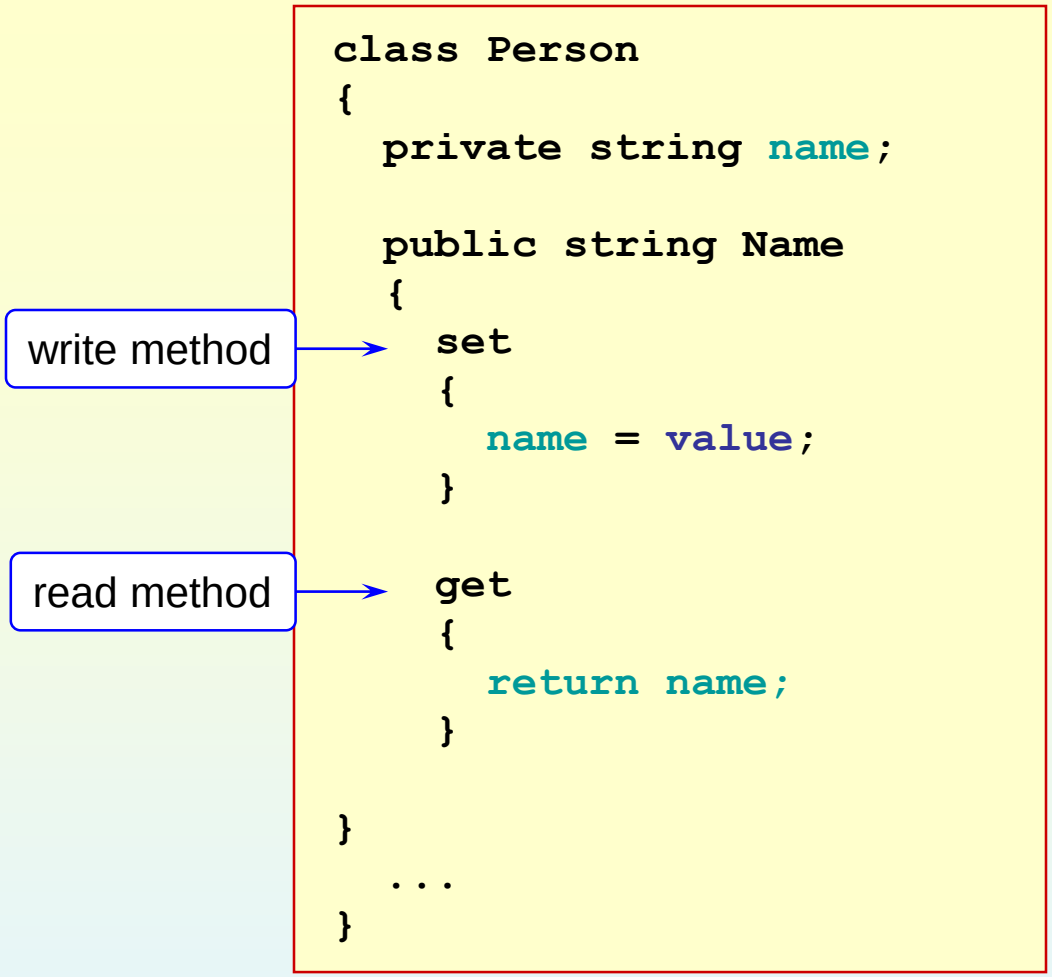
```
Person p = new Person();

p.Name = "Ann";

string n = p.Name;
```

запис

читання



Властивості дозволяють обмежувати доступ:

- read-only (тільки get)
- write-only (тільки set)
- read and write (get та set)

```
class Person
{
  private DateTime dob;

  public int Age
  {
    get
    {
      int age = DateTime.Today.Year - dob.Year;

      DateTime birthday = dob.AddYears(age);

      if (birthday > DateTime.Today)
        age--;

      return age;
    }
  }
  ...
}
```

Індексатори

- Функціональні члени для індексування полів
- Аналог оператора [ ] в C++
- Індексатори компілюються у властивість `Item`

Модифікатори$_{opt}$ *тип* **this** [*список формальних параметрів*]
{
методи доступу
}

- `Polygon` містить масив вершин – об'єктів типу **`Point`**

```
class Polygon
{
  Point[] vertices;
  ...
}
```

use indexer to write

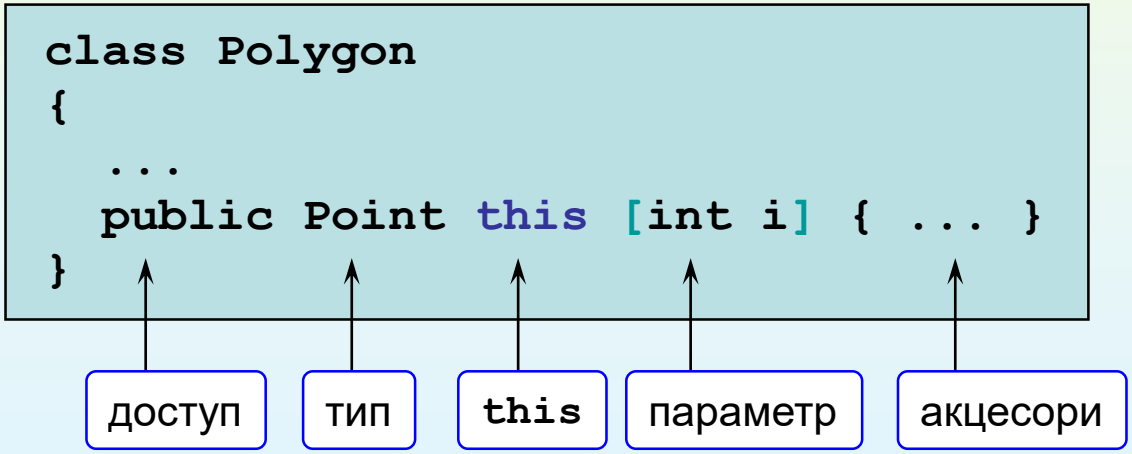
use indexer to read

```
Polygon triangle = new Polygon(3);

triangle[0] = new Point(0,0);
triangle[1] = new Point(2,0);
triangle[2] = new Point(1,2);

for (int i = 0; i < triangle.Size; i++)
  Console.WriteLine(triangle[i]);
```

- **Специфіка визначення індексатора:**
  - рівень доступу
  - тип даних, що буде проіндексовано
  - **`this`**
  - параметри в [ ]
  - акцесори в { }



```
class Polygon
{
  Point[] vertices;

  public Point this [int i]
  {
    set
    {
      vertices[i] = value;
    }
    get
    {
      return vertices[i];
    }

    ...
  }
  ...
}
```

Параметри індексатора

- можуть бути довільного типу
- повинні передаватися як значення, без використання **ref** або **out**

```
class Department
{
  public Employee this [string name] ...
  ...
}
```

```
Department d = new Department();
...
Employee e = d["Ann"];
```

- індексатори можуть мати один або більше параметрів

```
class Matrix
{
  public int this [int row, int column] ...
  ...
}
```

```
Matrix m = new Matrix();
...
int v = m[1,2];
```

- **Індексатори можуть бути перевантажені** (повинні відрізнятися кількістю або типом параметрів)

```
class Matrix
{
  public int[] this [int row] ...

  public int   this [int row, int column] ...

  ...
}
```

Методи: визначення і перевантаження

method-declaration:

method-header *method-body*

method-header:

***modifiers*$_{opt}$ *return-type* *member_name* (*parameter-list*$_{opt}$)**

сигнатура

modifiers:

method-modifier

method-modifiers *method-modifier*

method-modifier:

```
new
public
protected
internal
private
static
virtual
sealed
override
abstract
extern
```

return-type:

type

`void`

member-name:

identifier

interface-type . *identifier*

method-body:

block

;

Параметри методів

1. За замовчуванням – аргумент передається як значення, тобто метод матиме власну копію аргумента
2. ***Наслідок 1:*** **аргумент типу-посилання** передається в метод як посилання (адреса), тобто зміна аргумента в методі змінить значення об'єкта.
3. **`ref`** і **`out`** –модифікація аргумента як посилання (як для значень, так і посилань)
   - **`ref`** вхідний аргумент, перед викликом має бути ініціалізованим
   - **`out`** вихідний аргумент, перед закінченням роботи методу має отримати значення; якщо перед викликом не був ініціалізований, то в методі не може бути зчитаним

```
class Calc{
   public void Add(int x, int y, ref int ans)
   {
        ans=x+y;
   }

   public void AddVal(out int v)
   {
        v+=10;
   }
};
```

```
class App
{
       public static void Main()
       {
              Calc m=new Calc();
              int x,y=0;
              m.Add(90,90,ref y);
              //y=180
              x=2;
              m.AddVal(out x);
              //x=12
       }
}
```

Аргумент - масив

- **Дозволяє передавати цілий набір параметрів (невизначену кількість) як одне ціле**
- **Визначення:**

  `params` ідент. типу `[]` ідентифікатор

  Приклад : params object[] list

```
public void DisplayArr(string msg, params int [] list)
{
        Console.Writeline(msg);
        for(int i=0;i<list.Length;i++)
                Console.Writeline(list[i]);
}
```

```
// використання:
        int[] intAr=new int [3] {10,11,56};
        m.DisplayArr("1",intAr);
        m.DisplayArr("2",78,89,76);
        m.DisplayArr("3",7,8,8,9,7,6);
```

Параметри і перевантаження методів

- **Допустимі перевантаження**

```
–public void ArgConcordance (ref int p){…}
–public void ArgConcordance (int p){…}

–public void ArgConcordance (out Point p){…}
–public void ArgConcordance (Point p){…}
```

- **Недопустиме перевантаження**

```
–public void ArgConcordance (ref Point p){…}
–public void ArgConcordance (out Point p){…}
```

Створення типів через Struct

- Структури можуть містити *методи, поля, індексатори, властивості, оператори, події*
- Структури можуть мати *різні конструктори*, але не можуть мати *деструктора*
- *Дефолтний конструктор* для структури визначити не можна. Він надається автоматично і змінити його не можна
- Структури не можуть наслідувати інші структури чи класи.
- Структури не можуть використовуватися як базовий тип для інших структур та класів
- Структури можуть реалізовувати (наслідувати) один або декілька інтерфейсів.
- Методи структури не можуть мати специфікаторів *abstract, virtual, or protected.*
- При утворенні об’єктів структур можна не використовувати оператора *new*. Використання оператора new приведе лише до виклику відповідного конструктора.
- Якщо не використовувати оператора new при утворення змінних структури, поля структури залишаються непроініціалізовані і не можуть бути використані доти, доки не будуть проініціалізовані.

Приклад використання структур: DataTime

```
class Program
{
    static void Main()
    {
            // DateTime is a struct.
            DateTime date = new DateTime(2000, 1, 1);

            // When you assign a DateTime, a separate copy is
created.
            DateTime dateCopy = date;

            // The two structs have the same values.
            Console.WriteLine(date);
            Console.WriteLine(dateCopy);

            // The copy is not affected when the original changes.
            date = DateTime.MinValue;
            Console.WriteLine(dateCopy);
    }
}
```

Клас та структура – оператор присвоєння

```
class ShapeInfo
{
  public string infoString;
}
```

```
struct Rectangle
{
  // The Rectangle structure contains a reference type member.
  public ShapeInfo rectInfo;

  public int rectTop, rectLeft, rectBottom, rectRight;

  public Rectangle(string info, int top, int left, int bottom, int right)
  {
    rectInfo = new ShapeInfo(info);
    rectTop = top;  rectBottom = bottom;
    rectLeft = left; rectRight = right;
  }

  public void Display()
  {
    Console.WriteLine("String = {0}, Top = {1}, Bottom = {2}, " +
      "Left = {3}, Right = {4}",
      rectInfo.infoString, rectTop, rectBottom, rectLeft, rectRight);
  }
}
```

```
static void ValueTypeContainingRefType()
{
  // Create the first Rectangle.
  Console.WriteLine("-> Creating r1");
  Rectangle r1 = new Rectangle("First Rect", 10, 10, 50, 50);

  // Now assign a new Rectangle to r1.
  Console.WriteLine("-> Assigning r2 to r1");
  Rectangle r2 = r1;

  // Change some values of r2.
  Console.WriteLine("-> Changing values of r2");
  r2.rectInfo.infoString = "This is new info!";
  r2.rectBottom = 4444;

  // Print values of both rectangles.
  r1.Display();
  r2.Display();
}
```

```
-> Creating r1
-> Assigning r2 to r1
-> Changing values of r2
String = This is new info!, Top = 10, Bottom = 50, Left = 10, Right = 50
String = This is new info!, Top = 10, Bottom = 4444, Left = 10, Right = 50
```