

# Розробка функціональності класу

- Перевизначення методів Object
- Перевантаження операторів
- Особливості конструювання класів-агрегатів
- Похідні типи - наслідування реалізації

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/file-system/>

<https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions>

# Методи: визначення і перевантаження

method-declaration:

method-header method-body

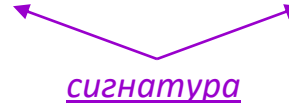
method-header:

attributes<sub>opt</sub> modifiers<sub>opt</sub> return-type member name ( parameter-list<sub>opt</sub> )

modifiers:

method-modifier

method-modifiers method-modifier

  
сигнатура

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  
=>expression  
;

# Параметри методів

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

```
void Inc(ref int x){ ++x; }  
  
void F()  
{  
    int val = 3;  
    Inc(ref val); // val == 4  
}
```

```
void Read (out int a, out int b)  
{  
    a = Console.Read();  
    b = Console.Read();  
}  
  
void F()  
{  
    int first, next;  
    Read(out first, out next);  
}
```

# Змінна кількість аргументів; аргумент–масив

**params** ідентифікатор типу **[]** ідентифікатор

- останні кілька аргументів певного типу
- без **ref** і **out**

```
void add(out int sum, params int[] val)
{
    sum = 0;
    foreach (int i in val) sum += i;
}
```

```
int s;
add(out s, 3, 5, 2, 9);
Console.WriteLine($"s : {s}");

var a = new int[] { 3, 5, 2, 9 };
add(out s, a);
Console.WriteLine($"s : {s}");
```

# Object.Equals

```
class Object
{
    public virtual Boolean Equals(object obj)
    {
        // If both references point to the same object, they must be equal.
        if (this == obj) return true;
        // Assume that the objects are not equal.
        return false;
    }

    public static Boolean Equals(object objA, object objB)
    {
        // If objA and objB refer to the same object, return true.
        if (objA == objB) return true;
        // If objA or objB is null, they can't be equal, so return false.
        if ((objA == null) || (objB == null)) return false;
        // Ask objA if objB is equal to it, and return the result.
        return objA.Equals(objB);
    }
    ....
}
```

# class Point\_s

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

# Point\_s - порівняння на рівність

```
public override bool Equals(object obj)
{
    if (obj == null) return false;
    if (GetType() != obj.GetType()) return false;
    Point_s p = (Point_s)obj;
    if (x == p.x && y == p.y) return true;
    return false;
}

public static bool operator ==(Point_s p1, Point_s p2)
{
    return Object.Equals(p1, p2);
}

public static bool operator !=(Point_s p1, Point_s p2)
{
    return !(p1 == p2);
}
```

# Point\_s : використання

```
Point_s pt = new Point_s();  
Console.WriteLine("pt : {0}", pt);  
Point_s pt1 = new Point_s(6,16);  
Console.WriteLine("pt1: {0}", pt1);
```

```
Point_s ptReal = new Point_s(10);  
Console.WriteLine("ptReal: {0}", ptReal);
```

```
Point_s pt2 = new Point_s(10, 20);  
Point_s pt3 = new Point_s(10, 20);
```

```
Console.WriteLine("pt2==pt3:{0}", pt2 == pt3);  
Console.WriteLine("pt2==pt3:{0}", pt3.Equals(pt2));  
Console.WriteLine("pt2==pt3:{0}", Equals(pt3, pt2));
```



# Перевантаження операторів

## ❑ Шаблон оператора

```
public static retval operatorop( object1 [, object2 ]){...}
```

## ❑ Шаблон оператора перетворення типу

- `public static implicit operator conv-type-out`  
`(conv-type-in operand)` – неявне перетворення
- `public static explicit operator conv-type-out`  
`(conv-type-in operand)` – явне перетворення

## ❑ Вимоги (перевантажується для типу T) :

- завжди `public static`
- для унарного оператора – тип аргумента є типом T
- для бінарного оператора – тип 1-го аргумента є типом T

## ❑ Реалізація:

- методами `op_XxxXxxx`, `op_Implicit`, `op_Explicit`
- у визначенні методу ознака `specialname`

# Оператори та їх реалізація

C# Operator	Special Method Name	Common Language Specification Method Name
+	op_UnaryPlus	Plus
-	op_UnaryNegation	Negate
~	op_OnesComplement	OnesComplement
++	op_Increment	Increment
--	op_Decrement	Decrement
+	op_Addition	Add
+=	op_AdditionAssignment	Add
-	op_Subtraction	Subtract
-=	op_SubtractionAssignment	Subtract
*	op_Multiply	Multiply
*=	op_MultiplicationAssignment	Multiply
/	op_Division	Divide
/=	op_DivisionAssignment	Divide
%	op_Modulus	Mod
%=	op_ModulusAssignment	Mod
^	op_ExclusiveOr	Xor
^=	op_ExclusiveOrAssignment	Xor

# Категорії операторів

- Primary (x) x.y f(x) a[x] new  
typeof sizeof checked unchecked
- Unary + - ! ~ ++x --x (T)x
- Multiplicative \* / %
- Additive + -
- Shift << >>
- Relational and type testing < > <= >= is as
- Equality == !=
- Logical AND &
- Logical XOR ^
- Logical OR |
- Conditional AND &&
- Conditional OR ||
- Conditional ?:
- Assignment = \*= /= %= += -= <<= >>= &= ^= |=

- можна перевантажувати
- бінарні крім присвоєння – ліво-асоціативні
- унарні, присвоєння і ?: – право-асоціативні

# Point\_op - перевантаження операторів

```
class Point_op
{
    ...
    public static Point_op operator +(Point_op p, int dx)
    {
        p.x += dx;
        return p;
    }
    public static Point_op operator +(Point_op p, Point_op dp)
    {
        p.x += dp.x;    p.y += dp.y;
        return p;
    }
    public static implicit operator Point_op(int x)
    {
        return new Point_op(x);
    }
    public static implicit operator int( Point_op p)
    {
        return p.x;
    }
}
```

# Point\_op - оператори порівняння

```
class Point_op {
    ...
    public override bool Equals (Object obj)
    {
        if(obj==null) return false;
        if(this.GetType() !=obj.GetType() ) return false;
        Point_op p=(Point_op) obj;
        if(x==p.x&&y==p.y) return true;
        return false;
    }

    public static bool operator ==(Point_op p1, Point_op p2)
    {
        return Object.Equals(p1, p2);
    }

    public static bool operator !=(Point_op p1, Point_op p2)
    {
        return !(p1== p2);
    }
}
```

# Point\_op - використання операторів

```
var pt = new Point_op();  
pt=pt + 5;  
Console.WriteLine("pt : {0}", pt);
```

```
var pt1 = new Point_op(6, 16);  
pt1 = pt1 + pt;  
Console.WriteLine("pt1: {0}", pt1);
```

```
var pt2 = new Point_op(10, 20);  
var pt3 = new Point_op(10, 20);
```

```
Console.WriteLine("pt2==pt3: {0}", pt2== pt3);  
    Console.WriteLine("pt2==pt3: {0}", pt2!= pt3);
```

# Наслідування реалізації

## Implementation Inheritance

Спосіб визначення нового типу на основі існуючого – **базового класу**:

- **безпосередній** базовий клас лише один
- наслідуються всі дані стану та функціональність крім конструкторів
- дані стану і функціональність можна розширити новими членами
- поведінку базового типу можна модифікувати шляхом **приховування** членів або **перевизначення** – надання їм поліморфних властивостей
- якщо базовий клас **abstract**, то похідний або реалізовує нереалізовані в базовому класі методи, або залишається сам абстрактним
- технічні аспекти: повторне використання коду, невеликий обсяг програм, зручність відлагодження і супроводу
- базовим може бути лише клас, а не структура
- структури не наслідують інші типи як реалізацію

# Модифікація поведінки базового типу

- ❑ *Перевизначення* методу з тією ж сигнатурою:

`virtual` у базовому типі і `override` у похідному

Реалізація методу визначається типом об'єкта, на який посилається змінна

- ❑ *Приховування* методу методом:

`new` у похідному типі (з тією ж сигнатурою, інакше - перевантаження)

Реалізація методу визначається типом посилання. Нова реалізація **не наслідується** в наступних похідних класах (наслідується з базового класу).

- ❑ *Приховування* полем, константою, властивістю або вкладеним типом.

Буде прихованим **будь-який член базового класу** з тим же ідентифікатором.

Без `new` видаватиметься попередження.



# Присвоювання і перевірка типу

```
class A {...}  
class B : A {...}  
class C: B {...}
```

## ❑ Assignments

```
A a = new A(); // static type of a: the type specified in the declaration (here A)  
               // dynamic type of a: the type of the object in a (here also A)
```

```
B b = a;      // forbidden; compilation error
```

```
a = new B(); // dynamic type of a is B  
a = new C(); // dynamic type of a is C
```

## ❑ Run-time type checks

```
a = new C();  
if (a is C) ...           // true, if the dynamic type of a is C or a subclass; otherwise false  
if (a is B) ...           // true  
if (a is A) ...           // true, but warning because it makes no sense  
  
a = null;  
if (a is C) ...           // false: if a == null, a is T always returns false
```

# Перетворення типу

## ❑ Cast

```
A a = new C();
```

```
B b = (B) a;
```

```
C c = (C) a;
```

```
a = null;
```

```
c = (C) a;           // ok → null can be casted to any reference type
```

## ❑ using **as** operator to perform certain types of conversions between compatible reference types

```
A a = new C();
```

```
B b = a as B;         // if (a is B) b = (B)a; else b = null;
```

```
C c = a as C;
```

```
a = null;
```

```
c = a as C;           // c == null
```

# class Pixel

```
class Pixel:Point
{
    private Color c;

    public Pixel() : base() { c = Color.Black; }
    public Pixel(Point p, Color cl):base(p){ c = cl; }
    public Pixel(Point p) : this(p,Color.Black) { }

    public override string ToString()
    {
        return base.ToString() + " color:" + c;
    }
    ...
}
```

# Pixel.Equals()

```
class Pixel : Point
{
    ....
    public override bool Equals(Object obj)
    {
        if (!base.Equals(obj)) return false;

        if (this.GetType() != obj.GetType()) return false;

        Pixel p = (Pixel)obj;
        if (c.Equals(p.c)) return true;
        return false;
    }

    public static bool operator ==(Pixel p1, Pixel p2)
    {
        return Object.Equals(p1, p2);
    }

    public static bool operator !=(Pixel p1, Pixel p2)
    {
        return !(p1 == p2);
    }
}
```

# Pixel - використання

```
Pixel p=new Pixel();  
Console.WriteLine("p : {0}", p);  
Pixel p1=new Pixel(new Point(10,30));  
Console.WriteLine("p1 : {0}", p1);  
Pixel p2 = new Pixel(new Point(10, 30),Color.Blue);  
Console.WriteLine("p2 : {0}", p2);  
Console.WriteLine("p1==p2 : {0}", p1==p2);  
p1 = p2;  
Console.WriteLine("p1==p2 : {0}", p1 == p2);  
p1 = null;  
Console.WriteLine("p1==p2 : {0}", p1 == p2);  
p2 = null;  
Console.WriteLine("p1==p2 : {0}", p1 == p2);  
Point pt = new Point(20, 50);  
Console.WriteLine("pt : {0}", pt);  
    pt = p;  
Console.WriteLine("pt : {0}", pt);
```

# Pixel — результат використання

```
void c-tor: Pixel_s.Pixel (0, 0) color:Black
p : Pixel_s.Pixel (0, 0) color:Black
int,int c-tor: Pixel_s.Point (10, 30)
int,int c-tor: Pixel_s.Pixel (10, 0) color:Black
int c-tor: Pixel_s.Pixel (10, 0) color:Black
p1 : Pixel_s.Pixel (10, 0) color:Black
int,int c-tor: Pixel_s.Point (10, 30)
int,int c-tor: Pixel_s.Pixel (10, 0) color:Black
int c-tor: Pixel_s.Pixel (10, 0) color:Black
p2 : Pixel_s.Pixel (10, 0) color:Blue
p1==p2 : False
p1==p2 : True
p1==p2 : False
p1==p2 : True
int,int c-tor: Pixel_s.Point (20, 50)
pt : Pixel_s.Point (20, 50)
pt : Pixel_s.Pixel (0, 0) color:Black
Press any key to continue . . .
```

# Наслідування інтерфейсів

## Інтерфейс:

- Засіб об'єднання в одну функціональну групу методів, **властивостей і подій**
- Гарантія реалізації класом оголошеної інтерфейсом поведінки
- Реалізація поліморфної поведінки незалежними типами
- Допустима реалізація членів інтерфейсу за замовчуванням (C# 8.0)

```
атрибутиopt модифікаториopt  
interface ідентифікатор: список інтерфейсівopt  
{  
    оголошення членів інтерфейсу  
}
```

# Реалізація інтерфейсів

```
interface IControl { void Paint();}

interface ISurface { void Paint();}

class SampleClass : IControl, ISurface
{
    // Both ISurface.Paint and IControl.Paint call this method
    public void Paint()
    {
        Console.WriteLine("Paint method in SampleClass");
    }
}

SampleClass sc = new SampleClass();

IControl ctrl = (IControl)sc;
ISurface srfc = (ISurface)sc;

// The following lines all call the same method
sc.Paint();
ctrl.Paint();
srfc.Paint();
```



# Явна реалізація інтерфейсів

```
public class SampleClass : IControl, ISurface
{
    void IControl.Paint() {
        System.Console.WriteLine("IControl.Paint");
    }
    void ISurface.Paint() {
        System.Console.WriteLine("ISurface.Paint");
    }
}

var obj = new SampleClass();
//obj.Paint(); // Compiler error

var c = (IControl)obj;
c.Paint(); // Calls IControl.Paint on SampleClass

var s = (ISurface)obj;
s.Paint(); // Calls ISurface.Paint on SampleClass
```

## Output:

```
IControl.Paint
ISurface.Paint
```

# Point\_cl – утворення копій

```
class Point: ICloneable {  
    ...  
    #region ICloneable Members  
    public object Clone() {  
        return new Point(x, y);  
    }  
    #endregion  
}  
  
var pt = new Point(10,10);  
Console.WriteLine("pt : {0}", pt);  
  
var pt1 = (Point)pt.Clone();  
Console.WriteLine("pt1: {0}", pt1);
```

# class Line\_s

```
class Line_s: ICloneable
{
    public Line_s()
    {
        beg = new Point(); end = new Point();
    }
    public Line_s(Point b, Point e)
    {
        beg = (Point)b.Clone();
        end = (Point)e.Clone();
    }
    public override string ToString()
    {
        return base.ToString()+" ["+beg+", "+end+" "];
    }
    private Point beg;
    private Point end;

    #region ICloneable Members
    public object Clone()
    {
        return new
            Line_s((Point)beg.Clone(), (Point)end.Clone());
    }
    #endregion
}
```

# Line\_s - використання

```
var l = new Line_s();  
Console.WriteLine("l : {0}", l);
```

```
var l1 = new Line_s(new Point(10,20),new Point(40,50));  
Console.WriteLine("l1 : {0}", l1);
```

```
var lcl = (Line_s)l.Clone();  
Console.WriteLine("lcl : {0}", lcl);
```

void c-tor: Line.Point (0, 0)

void c-tor: Line.Point (0, 0)

l : Line.Line\_s [Line.Point (0, 0), Line.Point (0, 0)]

int,int c-tor: Line.Point (10, 20)

int,int c-tor: Line.Point (100, 200)

int,int c-tor: Line.Point (10, 20)

int,int c-tor: Line.Point (100, 200)

l1 : Line.Line\_s [Line.Point (10, 20), Line.Point (100, 20)

int,int c-tor: Line.Point (0, 0)

int,int c-tor: Line.Point (0, 0)

int,int c-tor: Line.Point (0, 0)

int,int c-tor: Line.Point (0, 0)

lcl : Line.Line\_s [Line.Point (0, 0), Line.Point (0, 0)]

# Властивості - Properties

- ❑ Функціональні члени, які використовуються як поля

атрибути<sub>opt</sub> модифікатори<sub>opt</sub> тип ідентифікатор { методи доступу }

атрибути<sub>opt</sub> модифікатори<sub>opt</sub> тип інтерфейс.ідентифікатор { методи доступу }

- ❑ Оголошення в інтерфейсі

```
public interface IColoring
{
    Color Color
    {
        get;
        set;
    }
}
```

- ❑ Реалізація в класі

```
public Color Color
{
    get
    {
        return color;
    }
    set
    {
        color = value;
    }
}
```

```
public Color Color
{
    get => color;
    set => color=value;
}
```

```
public Color Color{ get; set;}
```

# Auto-Implemented Property

- Explicit implementation

```
public class Person
{
    public string FirstName
    {
        get { return firstName; }
        set { firstName = value; }
    }
    private string firstName;

    // . . . . .
}
```

- Auto-implemented property

```
public class Person
{
    public string FirstName { get; set; }

    // . . . . .
}
```

# Property using

- Validation

```
public class Person
{
    public string FirstName
    {
        get => firstName;
        set => firstName = (!string.IsNullOrEmpty(value)) ? value
                        : throw new ArgumentException("First name must not be blank");
    }
    private string firstName;
}
```

- initialization of a property to a value other than the default for its type

```
public class Person
{
    public string FirstName { get; set; } = string.Empty;
}
```

- Properties can be overridden (virtual, override).
- when **get** return private variable and optimizations are enabled, the call to the **get** accessor is **inlined** by the compiler
- a **virtual get** accessor cannot be inlined

# Доповнення функціональності

- ❑ Properties are a form of smart fields

```
class TimePeriod
{
    private double _seconds;

    public double Hours
    {
        get { return _seconds / 3600; }
        set { _seconds = value * 3600; }
    }
}
```

```
var t = new TimePeriod();
// Assigning the Hours property causes the 'set' accessor to be called
t.Hours = 24;

// Evaluating the Hours property causes the 'get' accessor to be called.
Console.WriteLine("Time in hours: " + t.Hours);
```



# Read-only property

- setting in a constructor

```
public class Person
{
    public Person(string name) => FirstName = name;

    public string FirstName { get; }
}
```

- setting by property initializer

```
public class Measurements
{
    public ICollection<DataPoint> points { get; } = new List<DataPoint>();
}
```

# Computed properties

- returning a computed value
- with out **backing store**
- when **get** return **private** variable and optimizations are enabled, the call to the get accessor is **inlined** by the compiler

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public string FullName { get { return $"{FirstName} {LastName}"; } }
}
```

- using the lambda expression syntax

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public string FullName => $"{FirstName} {LastName}";
}
```

# Cached evaluated properties

```
public class Person{
    private string firstName;
    public string FirstName
    {
        get => firstName;
        set{
            firstName = value;
            fullName = null;
        }
    }
    private string lastName;
    public string LastName
    {
        get => lastName;
        set{
            lastName = value;
            fullName = null;
        }
    }
    private string fullName;
    public string FullName
    {
        get{
            if (fullName == null)
                fullName = $"{FirstName} {LastName}";
            return fullName;
        }
    }
}
```

# Індексатори

- ❑ Функціональні члени для індексування полів

атрибути<sub>opt</sub> модифікатори<sub>opt</sub> оголошення індексатора { методи доступу }

оголошення індексатора :

тип **this** [список формальних параметрів]

тип інтерфейс.**this** [список формальних параметрів]

- ❑ Особливості:
  - Індексатори нагадують властивості(properties), за винятком того, що їхні екセスори отримують параметри
  - перевантаження за допомогою сигнатури
  - поліморфізм - **virtual** в базовому класі та **override** в похідному
  - ефективний для типів з полями-контейнерами

# Using [] notation

```
class SampleCollection<T>
{
    private var arr = new T[100];
    // Define the indexer to allow client code to use [] notation
    public T this[int i]
    {
        get {
            Console.WriteLine($"i={i}");
            return arr[i];
        }
        set => arr[i] = value;
    }
}

class Program
{
    static void Main()
    {
        var strColl = new SampleCollection<string>();

        strColl[0] = "Hello, World";
        WriteLine(strColl[0]);

        strColl[1] = "Programming Guide";
        WriteLine(strColl[1]);
    }
}
```