ТЕМА 3: КОНСТРУКТОРИ. ПЕРЕВАНТАЖЕННЯ КОНСТРУКТОРІВ. ВЛАСТИВОСТІ.

План

1. Конструктори екземплярів.
2. Перевантаження конструкторів. Посилання this.
3. Ініціалізатори об’єктів.
4. Властивості.
5. Аксесори.
6. Властивості, що стиснені до виразів.
7. Рівні доступу get та set.
8. Автоматичні властивості.
9. Ініціалізатори властивостей.

1. КОНСТРУКТОРИ ЕКЗЕМПЛЯРІВ.

В C# елементом програмування є клас. Об’єкти створюються з цих класів, а функції інкапсулюються в класах як методи.

В ООП використовуються класи (типи, що містять дані та функції для маніпулювання даними) для інкапсуляції (об’єднання) даних (атрибутів) і методів (поведінки). Класи можна розглядати як шаблони для створення об’єктів (екземплярів класу).

Об’єкт є представленням чогось, що існує в реальному світі з відображенням того, чим *він є*, і того, що *він робить*. Об’єкти мають властивість приховування своєї реалізації від інших об’єктів, не зважаючи на те, що деякі об’єкти можуть взаємодіяти один з одним через чітко визначені інтерфейси.

Конструктори екземплярів виконують код ініціалізації класу або структури. Конструктор записується подібно до методу, лише в якості імені вказується ім’я типу:

```
class Panda
{
     string name;//Визначення поля
     public Panda(string n)//Визначення конструктора
     {
          name=n;//Код ініціалізації
     }
}
...
Panda P=new Panda("Мишко");//Виклик конструктора
```

Конструктори допускають застосування наступних модифікаторів:

Модифікатори доступу	`public, internal, private, protected`
Модифікатор некерованого коду	`unsafe, extern`

Клас або структура може ***перевантажувати конструктори***. Для запобігання дублювання коду один конструктор може викликати інший, використовуючи ключове слово `this`:

```
class Panda
{
     string name;
     int BrYear;
     public Panda(string n){name=n;}
     public Panda(string n, int b): this (n) { BrYear=b;}
}
```

Викликати таким чином визначений конструктор можна з різною кількістю аргументів:

```
Panda P1=new Panda("Васько");
Panda P2=new Panda("Мишко", 2015);
```

Ініціалізація об'єктів. Для полегшення створення та ініціалізації об'єктів.

Нехай маємо опис класу:

```
class Point
{
public int X { get; set; }
public int Y { get; set; }
public Point(int xVal, int yVal)
{
X = xVal;
Y = yVal;
}
public Point () { }
public void DisplayStats()
{
Console.WriteLine("[{0}, {1}]", X, Y) ;
}
}
```

Створення об'єктів:

```
// Создать объект Point с установкой каждого свойства вручную.
Point firstPoint = new Point();
firstPoint.X = 10;
firstPoint.Y = 10;
firstPoint.DisplayStats();
// Создать с использованием специального конструктора.
Point anotherPoint = new Point(20, 20);
```

```
anotherPoint.DisplayStats();
// Создать с использованием синтаксиса инициализатора объекта.
Point finalPoint = new Point { X = 30, Y = 30 };
finalPoint.DisplayStats() ;
```

Викликається стандартний конструктор, після чого ініціалізуються властивості.

Можна також викликати спеціальні конструктори разом з ініціалізацією властивостей:

```
Point pt = new Point(10, 16) {X = 100, Y = 100};//X=100,Y=100
```

Ініціалізація вкладених типів:

```
class Rectangle
{
private Point topLeft = new Point();
private Point bottomRight = new Point();
public Point TopLeft
{
get { return topLeft; }
set { topLeft = value; }
}
public Point BottomRight
{
get { return bottomRight; }
set { bottomRight = value; }
}
}
```

Традиційний підхід:

```
Rectangle r = new Rectangle();
Point p1 = new Point();
p1.X = 10;
p1.Y = 10;
r.TopLeft = p1;
Point p2 = new Point() ;
p2.X = 200;
p2.Y = 200;
```

```
r.BottomRight = p2;
```

Із застосуванням ініціалізації:

```
Rectangle myRect = new Rectangle
{
TopLeft = new Point { X = 10, Y = 10 },
BottomRight = new Point { X = 200, Y = 200}
};
```

Перевага синтаксису ініціалізації полягає у скороченні об’єму коду.

Властивості. Властивості класу виглядають як поля, але при зчитуванні або встановленні значень містять логіку. Реалізується ця логіка за допомогою відповідно блоків `get` і `set`:

```
class price
{
     private decimal cina;
     public decimal Cina
     {
          get {return cina;} //виконується при зчитуванні
          set {cina=value;} // виконується при присвоєнні
     }
}
```

Засіб доступу `get` виконується при зчитуванні властивості, а `set` – при присвоєнні. Засіб доступу `set` має неявний параметр `value` з типом властивості, що присвоюється. Хоча доступ до властивостей відбувається таким самим способом, як і до полів, проте властивості відрізняються тим, що надають програмісту повний контроль над отриманням і встановленням їх значень. Це дозволяє обирати будь-яке необхідне внутрішнє представлення, не показуючи внутрішні деталі користувачу.

Якщо вказати лише засіб доступу `get`, то властивість буде доступна лише для читання, а якщо лише `set` – тільки для запису.

З властивостями дозволено застосовувати наступні модифікатори:

Статичний модифікатор	`static`
Модифікатори доступу	`public, internal, private, protected`
Модифікатори успадковування	`new, virtual, abstract, override, sealed`
Модифікатор некерованого коду	`unsafe, extern`

Модифікатори доступу можна окремо застосовувати для засобів доступу `get` і `set`:

```
class price
{
     private decimal cina;
     public decimal Cina //властивість більш доступна
     {
          get {return cina;}
          private set {cina=value;} // засіб доступу менш доступний
     }
}
```

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
	public partial class Form1 : Form
	{
		public Form1()
		{
			InitializeComponent();
		}
	}

	public class Point
	{
		int x,y;
		public int X
		{
			set { if (value < 0) x = 0; else x = value; }
			get { return x; }
		}
		public int Y
		{
			set { if (value < 0) y = 0; else y = value; }
			get { return y; }
		}
		public Point() { }
		public Point(int x, int y):this()
		{
			X = x;
			Y = y;
		}

		public override string ToString()
		{
			return "Point(X=" + X + ", Y=" + Y + ")";
		}
	}
	public class Circle:IComparer<Circle>
	{
		protected Point center=new Point();
		int radius;
		public virtual void SetCenter(int x, int y)
		{
			center.X = x;
			center.Y = y;
		}
		public Circle(int x, int y, int radius)
		{
			SetCenter(x, y);
			this.radius = radius;
		}
		public Circle() { }
		public static void MooveRight(Circle obj)
		{
			obj.SetCenter(obj.center.X + 10, obj.center.Y);

			if (obj is ColoredCircle)
				((ColoredCircle)obj).circleColor = Color.Red;
		}

		public override string ToString()
```

```
        {
                return "Centr("+center.X+", "+center.Y+"), radius="+radius;
        }

        public int Compare(Circle x, Circle y)
        {
                if (x.radius > y.radius)
                        return 1;
                else if (x.radius < y.radius)
                        return -1;
                else return 0;
        }

}

public class ColoredCircle:Circle
{
        //new Point center=new Point();
        Color CircleColor;
        public Color circleColor
        {
                get { return CircleColor; }
                set { CircleColor = value; }
        }

        public ColoredCircle(int x, int y, int radius,  Color circleColor):base(x,y,radius)
        {
                this.circleColor = circleColor;
        }
        public override void SetCenter(int x, int y)
        {
                center.X = x;
                center.Y = y;
        }
}

public class Test<MyType>
{
        public static string ToMyString(MyType obj)
        {
                return "Tostring(" + obj.GetType().ToString() + ")=" + obj.ToString();
        }
}
}
```