Programowanie wizualne .NET 3

Programowanie Wizualne

Paweł Wojciechowski

Instytut Informatyki, Politechniki Poznańskiej

2018

Funkcje lokalne (C#7,C#8)

- W C#7 wprowadzono możliwość tworzenia funkcji lokalnych – funkcji wewnątrz metody
- Są one zawsze prywatne – nie można podać modyfikatora
- Nie wspierają przeciążania
- W C#8 funkcje lokalne mogą być statyczne – nie mają dostępu do pól klasy
- Zastosowania?

```
public class Point
{
    public int X { get; set; }
    public int Y { get; set; }
    public override string ToString() => $"({X}:{Y})";
}

public class Shape
{
    public Point LeftTopCorner { get; set; }
    public Point RightBottomCorner { get; set; }

    public float Area()
    {
        SwapIfNeeded();
        return CountArea();

        float CountArea()
        {
            return (RightBottomCorner.X - LeftTopCorner.X) * (RightBottomCorner.Y - LeftTopCorner.Y);
        }

        void SwapIfNeeded()
        {
            int minX = Math.Min(LeftTopCorner.X, RightBottomCorner.X);
            int maxX = Math.Max(LeftTopCorner.X, RightBottomCorner.X);
            int minY = Math.Min(LeftTopCorner.Y, RightBottomCorner.Y);
            int maxY = Math.Max(LeftTopCorner.Y, RightBottomCorner.Y);

            LeftTopCorner.X = minX;
            LeftTopCorner.Y = minY;
            RightBottomCorner.X = maxX;
            RightBottomCorner.Y = maxY;
        }
    }
}
```

Funkcje lokalne – C#8

```
public class Shape
{
    public Point LeftTopCorner { get; set; }
    public Point RightBottomCorner { get; set; }

    public float Area()
    {
        SwapIfNeeded(LeftTopCorner, RightBottomCorner);
        return CountArea( LeftTopCorner, RightBottomCorner);

        static float CountArea(Point p1, Point p2)
        {
            return (p1.X - p2.X) * (p1.Y - p2.Y);
       }

        static void SwapIfNeeded( Point p1, Point p2)
        {
            int minX = Math.Min(p1.X, p2.X);
            int maxX = Math.Max(p1.X, p2.X);
            int minY = Math.Min(p1.Y, p2.Y);
            int maxY = Math.Max(p1.Y, p2.Y);

            p1.X = minX;
            p1.Y = minY;
            p2.X = maxX;
            p2.Y = maxY;
        }
    }
}
```

Interfejsy

- typ, który definiuje składowe abstrakcyjne
- zgodnie z konwencją nazwy interfejsów zaczynają się literą I
- różnice między klasami abstrakcyjnymi a interfejsami:
  - można dziedziczyć po jednej klasie ale implementować wiele interfejsów
  - klasa abstrakcyjna może zawierać implementacje metod – interfejs może zawierać domyślne implementacje metod od wersji C#8
  - przy deklaracji interfejsu nie używa się modyfikatorów dostępu – z założenia wszystko powinno być publiczne. Dopuszczalne jest używanie innych modyfikatorów ale użycie ich jest dość niejasne – oprócz użycia w metodach statycznych i domyślnych implementacjach

```
public interface IPrintable
{
    void PrintIt();
}

public class Ksztalt:IPrintable …


IPrintable ip = new Kwadrat();
ip = k as IPrintable;
ip = (IPrintable)w;
```

Interfejsy – dopuszczalne formy użycia

- odwołania do składowych interfejsu

```
public interface IPrintable
{
    void PrintIt();
}
```

```
Kwadrat k = new Kwadrat();
IPrintable ip = k as IPrintable;
ip.PrintIt()
```

- działają operatory `is` i `as`
- mogą być „zwracane” przez metody

```
public IPrintable GetInterface()
{ … }
```

- mogą być parametrami metod

```
public void SetInterface( IPrintable ip)
{ … }
```

- można tworzyć hierarchię interfejsów

Interfejsy – jawna implementacja

```
public interface IPrintable
{
    void PrintIt();
}

public interface IPrinterPrintable
{
    void PrintIt();
}

public interface IScreenPrintable
{
    void PrintIt();
}

public class Ksztalt : IPrintable, IPrinterPrintable, IScreenPrintable
{
    public virtual void PrintIt() => Console.WriteLine("PrintIt Ksztalt ");
}
```

Interfejsy – jawna implementacja (2)

```
public class Ksztalt : IPrintable, IPrinterPrintable, IScreenPrintable
{
    public virtual void PrintIt() => Console.WriteLine("PrintIt Ksztalt ");
}

public class Ksztalt : IPrintable, IPrinterPrintable, IScreenPrintable
{
    void IPrintable.PrintIt()
    {
        Console.WriteLine("Printable print");
    }

    void IPrinterPrintable.PrintIt()
    {
        Console.WriteLine("PrinterPrintable print");
    }

    void IScreenPrintable.PrintIt()
    {
        Console.WriteLine("ScreenPrintable print");
    }

    public virtual void PrintIt()
    {
        Console.WriteLine("PrintIt Ksztalt");
    }
}
```

Pytanie 9: Które metody zostaną wywołane?

```
Kwadrat k = new Kwadrat();
k.PrintIt();
((IPrintable)k).PrintIt();
((IPrinterPrintable)k).PrintIt();
((IScreenPrintable)k).PrintIt();
```

Domyślna implementacja metod w interfejsie (C# 8)

```
public interface IDraw
{
    public void Draw()
    {
        Console.WriteLine("IDraw rysuje");
    }
}

public class Kwadrat : IDraw
{
}

public static void Main()
{
    Kwadrat k = new Kwadrat();

    k.Draw(); //błąd
    ((IDraw)k).Draw(); //OK

}
```

Nie można w interfejsie umieszczać właściwości niestatycznych z zainicjowanymi wartościami

To jest poprawny kod, bo ma implementacje:

```
private int Prop
{
    get { return 10; }
    set { ZmiennaStatic = value; }
}
```

Tak nie można:

```
public int BlednaProp { get; set; } = 1;
```

Różnice między Interfejsami a klasami abstrakcyjnymi

- Interfejsy nie mogą mieć niestatycznych konstruktorów
- Klasa może dziedziczyć po 1 klasie abstrakcyjnej i implementować wiele różnych interfejsów

```
public interface IDraw
{
    public static object Canvas { get; set; }
    public void Draw()
    {
        Console.WriteLine("IDraw rysuje");
    }

    static IDraw()
    {
        Canvas = new object();
    }
}
```

Interfejsy standardowe: IEnumerable i IEnumerator

- pętla foreach potrafi „iterować” po obiektach, które implementują interfejs IEnumerable

```
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

- interfejs IEnumerator umożliwia przeglądanie kolekcji

```
public interface IEnumerator
{
    bool MoveNext();
    object Current { get; }
    void Reset();
}
```

- oba interfejsy są zdefiniowane w System.Collections
- W C#9 foreach sprawdza jeszcze metody rozszerzone w celu znalezienia implementacji metody GetEnumerator()

Interfejsy standardowe: IEnumerable i IEnumerator (2)

```
public class Pracownik
{
    public string Imie { get; set; }
    public string Nazwisko { get; set; }
    public int Wiek { get; set; }
}

public class Firma:IEnumerable
{
    private Pracownik[] pracownicy;

    public Firma()
    {
        pracownicy = new Pracownik[] …
    }

    public System.Collections.IEnumerator GetEnumerator()
    {
        return pracownicy.GetEnumerator();
    }
}

Firma firma = new Firma();
foreach( var p in firma)
     Console.WriteLine(p);
```

Interfejsy standardowe: IEnumerable i IEnumerator (3)

- Słowo kluczowe `yield`

```
public System.Collections.IEnumerator GetEnumerator()
{
    yield return pracownicy[1];
    yield return pracownicy[2];
    yield return pracownicy[0];
    yield return pracownicy[3];
    yield return pracownicy[2];
}
```

- Iteratory nazwane

```
public IEnumerable GetByName()
{
    yield return pracownicy[3]
    …
}
```

```
public IEnumerable GetByAge(bool ascending)
{
    if ( ascending )
    {
        yield return pracownicy[0];
    }
    else
    {
        yield return pracownicy[3];
    }
}
```

```
foreach (var p in firma.GetByName())
    Console.WriteLine(p);
foreach (var p in firma.GetByAge(true))
    Console.WriteLine(p);
```

Interfejsy standardowe: `ICloneable`

- interfejs do tworzenia kopii obiektu

```
public interface ICloneable
{
    object Clone();
}
```

- do tworzenia płytkich kopii jest metoda klasy `object MemberwiseClone()`

Interfejsy standardowe: IComparable

- interfejs pozwalający na sortowanie tablic zawierających obiekty danej klasy

```
public interface IComparable
{
    int CompareTo(object o);
}
```

wartości <0 this jest PRZED o
wartość ==0 this jest równe o
wartości >0 this jest PO o

```
public class Pracownik:IComparable
{
    public int CompareTo(object o)
    {
        Pracownik p = o as Pracownik;
        return this.Nazwisko.CompareTo(p.Nazwisko);
    }
}
```

Interfejsy standardowe: IComparer

- pozwala definiować różne porządki sortowania

```
public interface IComparer
{
    int Compare(object o1, object o2);
}
```

```
Array.Sort(pracownicy, new  PracownikComparerByAge());
```

```
public class PracownikComparerByAge : IComparer
{
    public int Compare(object o1, object o2)
    {
        Pracownik p1 = o1 as Pracownik;
        Pracownik p2 = o2 as Pracownik;
        return p1.Wiek.CompareTo(p2.Wiek);
    }
}

public class PracownikComparerByAgeAndName : IComparer
{
    public int Compare(object o1, object o2)
    {
        …
        if (p1.Wiek != p2.Wiek)
            return p1.Wiek.CompareTo(p2.Wiek);
        else
            return p1.Nazwisko.CompareTo(p2.Nazwisko);
    }
}
```

Iterfejsy – C#8

- W C#8 wprowadzono możliwość implementacji metod w definicji interfejsu ☹

```
public interface IPrintable
{
    void Print()
    {
        Console.WriteLine("Print from interface");
    }
}

public class Some : IPrintable
{
    public void Print()
    {
        Console.WriteLine("print from class");
    }
}
```

Co zostanie wyświetlone na ekranie? Która metoda będzie użyta w poszczególnych przypadkach?

```
Some s = new Some();
s.Print();
IPrintable ip = s as IPrintable;
((IPrintable)s).Print();
ip?.Print();
```

Object Initialization Syntax - sprostowanie

```
public class Shape
{
    public Point LeftTopCorner { get; set; }
    public Point RightBottomCorner { get; set; }

    internal int X { get; set; }

    internal int field;
 }
```

- czy takie wywołanie jest poprawne?

```
Shape shape = new Shape() { X = 1, field = 1 };
```

Rekordy – records C#9

- Rekordy są typami referencyjnymi
- Z założenia są *immutable*

```
public record Car
{
    public string Producer { get; init; }
    public string Model { get; init; }

    public Car( string prod, string model)
    {
        Producer = prod;
        Model = model;
    }
}
```

```
Car c = new Car("Renault", "Clio");
Console.WriteLine( c.GetHashCode());
c.Producer = "Opel"; //Error – właściwość nie ma set

Console.WriteLine(c);
```

Rekordy

```
public class Car
{
    public string Producer { get; set; }
    public string Model { get; set; }

    public Car( string prod, string model)
    {
        Producer = prod;
        Model = model;
    }
}
```

```
Car c = new Car("Renault", "Clio");
Console.WriteLine( c.GetHashCode());
c.Producer = "Opel";

Console.WriteLine(c);
Console.WriteLine(c.GetHashCode());
```

```
58225482
ConsoleApp5.Car
58225482
```

```
public record Car
{
    public string Producer { get; set; }
    public string Model { get; set; }

    public Car( string prod, string model)
    {
        Producer = prod;
        Model = model;
    }
}
```

```
2001746275
Car { Producer = Opel, Model = Clio }
-1287039415
```

Krótsza implementacja – chociaż zachowanie będzie inne

```
public record CarRecord( string Producer, string Model);
```

Rekordy –konstruktor

```
public record Car
{
    public string Producer { get; set; }
    public string Model { get; set; }

    public Car(string prod, string model)
    {
        Producer = prod;
        Model = model;
    }

    public Car()
    {

    }
}
```

Konstruktor bezparametrowy: czy taki zapis ma sens?

Rekordy uwagi

- Rekordy mogą być abstrakcyjne
- C# 10 record = record class – wprowadzono jeszcze record struct
- Porównywanie rekordów – po wartościach
- Konstrukcja with

```
public record Point(int X, int Y)
{
    public int Zbase { get; set; }
};
public record NamedPoint(string Name, int X, int Y) : Point(X, Y)
{
    public int Zderived { get; set; }
};

public static void Main()
{
    Point p1 = new NamedPoint("A", 1, 2) { Zbase = 3, Zderived = 4 };

    Point p2 = p1 with { X = 5, Y = 6, Zbase = 7 }; // Can't set Name or Zderived
    Console.WriteLine(p2 is NamedPoint);  // output: True
    Console.WriteLine(p2);
    // output: NamedPoint { X = 5, Y = 6, Zbase = 7, Name = A, Zderived = 4 }

    Point p3 = (NamedPoint)p1 with { Name = "B", X = 5, Y = 6, Zbase = 7, Zderived = 8 };
    Console.WriteLine(p3);
    // output: NamedPoint { X = 5, Y = 6, Zbase = 7, Name = B, Zderived = 8 }
}
```


https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/record