Programowanie wizualne .NET 2

Programowanie Wizualne

Paweł Wojciechowski

Instytut Informatyki, Politechniki Poznańskiej

2022

Typy wyliczeniowe

- typ wyliczeniowy

```
enum DniTygodnia { Poniedzialek, Wtorek, Sroda, Czwartek, Piatek, Sobota, Niedziela}

enum Owoce: byte { Jablko, Gruszka, Sliwka}
```

- domyślnym typem jest `int`
- atrybut `[Flags]`

```
[Flags]
enum Owoce : byte { Jablko = 2, Gruszka=1, Sliwka }

public class TypWyliczeniowy
{
    public static void Main()
    {
        Owoce o = Owoce.Sliwka | Owoce.Gruszka;
        Console.WriteLine("{0}\t{0:d}", o);
    }
}
```

Pytanie 1: jaka będzie wartość stałej Sliwka?

```
enum Owoce: byte { Jablko = 1, Gruszka, Sliwka}

enum Owoce: byte { Jablko = 999, Gruszka=998, Sliwka }

Owoce o = Owoce.Sliwka;
```

Struktury

- „lekka” wersja obiektu
- są zawsze wartościami (na stosie, kopiowanie)
- mogą zawierać interfejsy, funkcje składowe i konstruktory z argumentami
- nie mogą inicjować pól (innych niż statyczne i stałe) w deklaracji struktury
- nie mogą mieć domyślnego konstruktora
- nie mogą dziedziczyć po innej strukturze/klasie i nie można z nich dziedziczyć (są zapieczętowane)

```
struct Pracownik
{
    public string Imie;
    public string Nazwisko;

    public Pracownik(string imie )
    {
        Imie = imie;
        Nazwisko = "Nowak";
    }
}
```

Struktury (2)

```
struct Pracownik
{
    public string Imie;
    public string Nazwisko;

    public Pracownik(string imie )
    {
        Imie = imie;
        Nazwisko = "Nowak";
    }

    public void Print()
    {
        Console.WriteLine($"{Imie} {Nazwisko}");
    }
}

Pracownik pracownik;
//pracownik.Imie = "Jan";
//pracownik.Nazwisko = "Kowalski";

Pracownik p = new Pracownik();
Pracownik p2 = new(); //C#9

Console.WriteLine("default constructor {0}",p.Nazwisko);

p.Print();
p2.Print();
pracownik.Print(); //ok po odkomentowaniu dwóch linii wyżej
```

Struktury (3)

- Struktury mogą być oznaczone jako read-only (C#7.2) – stają się niezmienialne (*immutable*) – inicjalizacja na etapie konstruktora. Korzyści: wydajność?
- Struktury tylko do odczytu mogą mieć właściwości ale również tylko do odczytu
- C#7.2 – dodano ref struct – nie mogą odwołać się poza stos. Inne ograniczenia m.in. :
  - nie mogą być używane jako typy w non-ref struct
  - nie mogą być używane w metodach async, iteratorach, tablicach, wyrażeniach lambda, czy funkcjach lokalnych
- Od C# 8.0 metody, właściwości itd. mogą być oznaczane jako read-only

```
public readonly struct Rect
{
    public int Width { get;  }
    public int Height { get; set; } //error
}


public struct Circle
{
    public int Radius;
    public readonly int Position { get; }
}
```

Struktury (4)

- Zmiany w C#10:
  - można tworzyć bezargumentowy konstruktor (wcześniej zabronione)
  - bezargumentowy konstruktor musi inicjalizować wszystkie pola
  - pola mogą być inicjowane w momencie deklaracji

```
struct Point
{
        public int X = 5;
}
```

Pytanie z innej beczki: czy można w jednej klasie zdefiniować dwie metody różniące się tylko modyfikatorami parametrów?

```
static int Add(ref int x) { /* */ }
static int Add(out int x) { /* */ }
```

Typ Nullable

- typ który rozszerza typy wartościowe o wartość null
- dziedziczenie po `System.Nullable<T>` inny zapis `T?`
- właściwość `HasValue` pozwala sprawdzić, czy zmienna ma wartość inną niż `null`

```
bool? b = null;

if (b.HasValue)
{
    Console.WriteLine("b ma wartość" + b);
}
```

- operator `??` - dotyczy nie tylko wartości `Nullable<T>`

```
bool odp = b ?? false;
```

Nullable Reference Types (C#8)

- Nullable<T> działa tylko gdy T jest typem wartościowym
- dla typów referencyjnych non-nullable generowane jest ostrzeżenie w przypadku próby użycia wartości null
- null forgiving operator (!) – brak ostrzeżenia (warning)

```
Point p = null!;

Point? p2 = null;
Point p3 = p2!;
```

```
class P{}

class Program
{
    static void Main(string[] args)
    {
#nullable enable
        Point p = null;

        Point? p2 = null;
        Point p3 = p2;
    }
}
```

Nullable – lifted operators

```
int? a = 10;
int? b = null;
int? c = 10;

a++;        // a is 11
a = a * c;  // a is 110
a = a + b;  // a is null
```

```
int? aa = 10;
Console.WriteLine($"{aa} >= null is {aa >= null}");
Console.WriteLine($"{aa} < null is {aa < null}");
Console.WriteLine($"{aa} == null is {aa == null}");
```

```
// Output:
// 10 >= null is False
// 10 < null is False
// 10 == null is False
```

- predefiniowane operatory są przeciążone żeby wspierały typy T? (lifted operators) i wynik jest null jeśli jeden lub obie zmienne mają wartość null
- porównania >, >=, <, <= - jeśli jeden lub obie zmienne mają wartość null – wynikiem jest False
- operator == jeśli oba są null – zwraca true, jeśli jeden – false
- operator != false jeśli oba są null, jeśli tylko jeden - true

Nullable Boolean logical operators

x	y	x&y	x|y
true	true	true	true
true	false	false	true
true	null	null	true
false	true	false	true
false	false	false	false
false	null	false	null
null	true	null	true
null	false	false	null
null	null	null	null

Krotki (Tuples)

- wcześniej była klasa `System.ValueTuple`
- możliwości inicjalizacji

- porównywanie krotek
- nazwy atrybutów
- C#7 odrzuty (*discard*)

```
static (string Imie, string Nazwisko, int wiek) GetPerson()
{
    return ("Jan", "Nowak", 33);
}

(_, var Nazwisko6, var Wiek6) = GetPerson();
```

```
(string, string, int) pracownik = ("Jan", "Nowak", 33);
var pracownik2 = ("Jan", "Nowak", 33);
(var Imie, var Nazwisko, var wiek)  = ("Jan", "Nowak", 33);
var (Imie2, Nazwisko2, wiek2)  = ("Jan", "Nowak", 33);
var pracownik3 = (Imie: "Jan", Nazwisko: "Nowak", Wiek: 33);
(string Imie, string Nazwisko, int Wiek) pracownik4 = ("Jan", "Nowak", 33);
Console.WriteLine( $"{pracownik4.Item2} {pracownik4.Wiek}");

if (pracownik.CompareTo(pracownik2) == 0)
{
    Console.WriteLine( "Ten sam pracownik");
}

string Imie5 = "Jan";
string Nazwisko5 = "Nowak";
int Wiek5 = 33;

//C#7.1
var pracownik5 = (Imie5, Nazwisko5, Wiek5);
Console.WriteLine($"{pracownik5.Imie5} {pracownik5.Nazwisko5}");
```

Obiektowość

Typy referencyjne i wartościowe

```
public class Point
{
    public int X;
    public int Y;
}
```

```
public struct Point
{
    public int X;
    public int Y;
}
```

```
public static void DodajX( Point p) => p.X++;

public static void DodajXref(ref Point p) => p.X++;

public static void NowyPunkt(Point p) => p = new Point();

public static void NowyPunktref(ref Point p) => p = new Point();
```

```
Point punkt = new Point();

Console.WriteLine($"({punkt.X}:{punkt.Y})");
DodajX(punkt);
Console.WriteLine($"1: ({punkt.X}:{punkt.Y})");
DodajXref(ref punkt);
Console.WriteLine($"2: ({punkt.X}:{punkt.Y})");
NowyPunkt(punkt);
Console.WriteLine($"3: ({punkt.X}:{punkt.Y})");
NowyPunktref(ref punkt);
Console.WriteLine($"4: ({punkt.X}:{punkt.Y})");
```

Pytanie 2: co zostanie wyświetlone dla struktury Point, a co dla klasy Point?

Konstruktor

- klasa na początku ma domyślny konstruktor bezargumentowy
- wszystkie pola i właściwości są inicjowane wartościami domyślnymi dla typów
- konstruktor nie zwraca żadnego typu nawet typu void
- po utworzeniu własnego konstruktora, domyślny bezargumentowy „znika”

Pytanie 3: czy poniższy kod jest poprawny?

```
public class Osoba
{
    public string imie;
    public Osoba(string imie)
    {
        imie = imie;
    }
}
```

Pytanie 4: czy poniższy kod jest poprawny?

```
public class Osoba
{
    public string imie;
    public string nazwisko;
    public Osoba(string imie)
    {
        this.imie = imie;
    }
    public Osoba(string nazwisko)
    {
        this.nazwisko = nazwisko;
    }
}
```

Konstruktor: łączenie wywołań

```
public class Osoba
{
    public string imie;
    public string nazwisko;
    public int wiek;

    public Osoba(string imie, string nazwisko, int wiek)
    {
        this.imie       = imie;
        this.nazwisko   = nazwisko;
        this.wiek       = wiek;
        Console.WriteLine("ctor 1");
    }

    public Osoba( string imie, string nazwisko): this(imie, nazwisko, 0)
    {
        Console.WriteLine("ctor 2");
    }

    public Osoba( int wiek): this( "","", wiek)
    {
        Console.WriteLine("ctor 3");
    }
}
```

Pytanie 4: jaka będzie kolejność wywołania konstruktorów?

```
Osoba o = new Osoba(20);
```

Pytanie 5: czy w konstruktorach można używać argumentów nazwanych?

```
Osoba o = new Osoba(imie: "Jurek", nazwisko: "Nowak", wiek: 20);
```

Zastosowanie słowa `static`

- statyczne pola klasy i metody
- statyczna klasa – może zawierać tylko statyczne metody i składowe
- statyczny konstruktor:
  - wywoływany zawsze przed jakimkolwiek użyciem danej klasy
  - zawsze bezargumentowy
  - tylko 1
  - bez modyfikatora dostępu
- importowanie statycznych funkcji (C#6)

```
using static System.Console;

WriteLine("Ala ma kota");
```

```
public class Osoba
{
    static Osoba()
    {

    }
}
```

Modyfikatory dostępu

Caller's location	`public`	`protected internal`	`protected`	`internal`	`private protected`	`private`
Within the class	✔	✔	✔	✔	✔	✔
Derived class (same assembly)	✔	✔	✔	✔	✔	✘
Non-derived class (same assembly)	✔	✔	✘	✔	✘	✘
Derived class (different assembly)	✔	✔	✔	✘	✘	✘
Non-derived class (different assembly)	✔	✘	✘	✘	✘	✘

- **`private`** – dostęp ograniczony wyłącznie do klasy zawierającej
- **`protected`** – dostęp ograniczony do klasy zawierającej oraz klas dziedziczących po klasie zawierającej
- **`public`** – dostęp nieograniczony
- **`internal`** – dostęp ograniczony do pakietu (assembly) zawierającego typ
- **`protected internal`** – dostęp ograniczony do pakietu lub typu dziedziczącego po klasie zawierającej typ (w lub poza pakietem)
- (C#7.2) **`private protected`** – dostęp ograniczony do klasy zawierającej oraz typów dziedziczących po klasie zawierającej wewnątrz pakietu

- domyślnie:
  - typy – `internal`
  - składowe - `private`

Pytanie 6: czy można definiować prywatną klasę?

Właściwości (Properties)

```
public class Punkt
{
    private int x;
    private int y;

    public int GetX() { return x; }
    public void SetX(int _x) { x = _x; }
}
```

```
public class Punkt
{
    private int x;
    private int y;

    public int X
    {
        get { return x;  }
        set { x = value; }
    }

}
```

- zaleta:

```
p.X++;
p.SetX(p.GetX() + 1);
```

Właściwości (2)

- mogą być tylko do odczytu – nie zawierają sekcji `set`
- mogą być tylko do zapisu – nie zawierają sekcji `get`
- C#9 – dodano setters działające tylko na etapie inicjalizacji `init`
- właściwości mogą być statyczne
- właściwości automatyczne

```
public class Punkt
{
    public int X { get; set; }
    public int Y { get; set; }
}
```

- od C#6 można definiować właściwości automatyczne tylko do odczytu
- właściwości automatyczne są inicjowane wartością domyślną dla typu
- od C#6 właściwości automatyczne (tylko one) mogą być inicjalizowane wartością

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

Pytanie 7: czy poniższy zapis jest poprawny?

```
public int X { get; } = 1;
```

```
class Base
{
    internal readonly int Field;
    internal int Property
    {
        get => Field;
        init => Field = value; // Okay
    }

    internal int OtherProperty { get; init; }
}

class Derived : Base
{
    internal readonly int DerivedField;
    internal int DerivedProperty
    {
        get => DerivedField;
        init
        {
            DerivedField = 42;  // Okay
            Property = 0;       // Okay
            Field = 13;         // Error Field is readonly
        }
    }

    public Derived()
    {
        Property = 42;  // Okay
        Field = 13;     // Error Field is readonly
    }
}
```


https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-9.0/init

Inicjalizowanie obiektów

```
public class Punkt
{
    public int X { get; set; } = 1;
    public int Y { get; set; } = 1;
}
```

- publiczne pola i właściwości można inicjalizować w następujący sposób:

```
Punkt p = new Punkt() { X = 20, Y = 20 };
Punkt p2 = new Punkt { X = 20, Y = 20 };
```

- po dodaniu konstruktora z parametrami, powyższe wywołania będą niepoprawne

```
public Punkt( int x, int y)
{
    X = x;
    Y = y;
}
```

Pytanie 8: jakie wartości będą miały właściwości X i Y przy poniższym wywołaniu?

```
Punkt p = new Punkt(10, 10) { X = 20, Y = 20 };
```

Stałe (const) vs pola tylko do odczytu (readonly)

- stała może być inicjalizowana TYLKO w momencie deklaracji
- stała nie może być statyczna – const domyślnie oznacza też static

```
public const double PI = 3.14;
```

- pola tylko do odczytu mogą być inicjowane w konstruktorach

```
public readonly double PI = 3.14;
```

Pytanie 9: gdzie będą inicjalizowane statyczne pola tylko do odczytu?

Pytanie 10: czy poniższe deklaracje są poprawne?

```
public readonly int Y { get; set; }
public readonly int X { get; }
```

Dziedziczenie

Dziedziczenie

- C# pozwala na dziedziczenie po jednej klasie

```
public class Pracownik
{ … }

public class Kierownik: Pracownik
{   }
```

- klasę można zapieczętować, czyli zabronić dziedziczenia po niej – słowo kluczowe `sealed` (np. wszystkie typy proste są zapieczętowane)

```
public sealed class Kierownik: Pracownik
```

- wywołanie konstruktora klasy nadrzędnej - `base`

Kolejność wywoływania konstruktorów

```
public class Pracownik
{
    public Pracownik() =>  Console.WriteLine("P");
}

public class Kierownik: Pracownik
{
    public Kierownik() => Console.WriteLine("K");
}

public class Dyrektor: Kierownik
{
    public Dyrektor()
    {
        Console.WriteLine("D");
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        Dyrektor d = new Dyrektor();
    }
}
```

Pytanie 1: co zostanie wypisane na ekranie?

Kolejność wywoływania konstruktorów - base

```
public class Pracownik
{
    …
    public Pracownik( string imie)
    {
        Console.WriteLine("P_s");
    }
}

public class Kierownik : Pracownik
{
    …
    public Kierownik(string dzial)
    {
        Console.WriteLine("K_s");
    }
}

public class Dyrektor: Kierownik
{
    public Dyrektor():base( "???")
    {
        Console.WriteLine("D");
    }
}
public class Program
{
    public static void Main(string[] args)
    {
        Dyrektor d = new Dyrektor();
    }
}
```

Pytanie 2: po dodaniu konstruktorów z parametrami (takimi samymi) do klas `Pracownik` i `Kierownik`, do którego z nich odwoła się `base`?

Pytanie 3: zakładając, że w klasie `Pracownik` jest pole/właściwość np.:
`protected int ID;`
, któremu jest nadawana wartość w konstruktorze `Pracownik()` i `Kierownik()`. Która wartość zostanie przypisana obiektowi?

Pytanie 4: czy pole `public readonly` może być modyfikowane w konstruktorze klasy dziedziczącej?

Klasy (typy) wewnętrzne

```
public class KlasaZewnetrzna
{
    private int pole;
    public int Pole { get { return pole; } }
    public class KlasaWewnetrzna
    {
        public KlasaWewnetrzna( KlasaZewnetrzna klasa)
        {
            klasa.pole = 10;
        }
    }
}
public class KlasyWewnetrzne
{
    public static void Main()
    {
        KlasaZewnetrzna k = new KlasaZewnetrzna();
        Console.WriteLine(k.Pole);
        KlasaZewnetrzna.KlasaWewnetrzna kk =
                  new KlasaZewnetrzna.KlasaWewnetrzna(k);
        Console.WriteLine(k.Pole);
    }
}
```

Pytanie 5: Czy klasa wewnętrzna może odwoływać się do pól prywatnych klasy zewnętrznej i na odwrót?

```
public class KlasaZewnetrzna
{
    public int Zmienna { get { return KlasaWewnetrzna.zmienna; } }
    public class KlasaWewnetrzna
    {
        private static int zmienna;
        public KlasaWewnetrzna( KlasaZewnetrzna klasa)
        {
            zmienna = 11;
        }
    }
}
```

Polimorfizm

```
public class Ksztalt
{
    public void PrintIt() => Console.WriteLine( "PrintIt Ksztalt ");
}

public class Wielokat: Ksztalt
{
    public void PrintIt() => Console.WriteLine("PrintIt Wielokat");
}

public class Kwadrat: Wielokat
{
    public void PrintIt() => Console.WriteLine("PrintIt Kwadrat");
}

public class Polimorfizm
{
    public static void Main()
    {
        Kwadrat k = new Kwadrat();
        k.PrintIt();
        Wielokat w = (Wielokat)k;
        w.PrintIt();

    }
}
```

Pytanie 6: Co zostanie wyświetlone na ekranie?

Polimorfizm – słowa kluczowe virtual/override

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

public class Wielokat: Ksztalt
{
    public override void PrintIt() => Console.WriteLine("PrintIt Wielokat");
}

public class Kwadrat: Wielokat
{
    public override void PrintIt() => Console.WriteLine("PrintIt Kwadrat");
}

public class Polimorfizm
{
    public static void Main()
    {
        Kwadrat k = new Kwadrat();
        k.PrintIt();
        Wielokat w = (Wielokat)k;
        w.PrintIt();
        Ksztalt ks = k as Ksztalt;
        ks.PrintIt();
    }
}
```

Pytanie 7: W jaki sposób wywołać metodę klasy nadrzędnej w takim przypadku?

Polimorfizm - virtual/override

- nie można użyć słowa override jeśli w klasie nadrzędnej metoda nie była oznaczone jako virtual
- operator as zmienna k zostanie zrzutowana na Ksztalt, a jeśli jest to niemożliwe do ks zostanie przypisane null

```
Ksztalt ks = k as Ksztalt;
```

- operator ?. Elvis operator

```
Ksztalt ksztalt = new Ksztalt();
Kwadrat k2 = ksztalt as Kwadrat;
k2?.PrintIt();
```

- użycie słowa kluczowego new

```
public new void PrintIt()
```

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

public class Wielokat: Ksztalt
{
    public sealed override void PrintIt() => Console.WriteLine("PrintIt Wielokat");
}

public class Kwadrat: Wielokat
{
    public new void PrintIt() => Console.WriteLine("PrintIt Kwadrat");
}

public class Polimorfizm
{
    public static void Main()
    {
        Kwadrat k = new Kwadrat();
        k.PrintIt();
        Wielokat w = (Wielokat)k;
        w.PrintIt();
        Ksztalt ks = k as Ksztalt;
        ks.PrintIt();
    }
}
```

Klasy abstrakcyjne (abstract)

- nie można tworzyć obiektów tej klasy
- mogą zawierać implementacje metod
- metody abstrakcyjne – wymuszenie implementacji metody w klasie pochodnej

Pytanie 8: Czy można zdefiniować metodę abstrakcyjną w klasie innej niż abstrakcyjna?