Programowanie wizualne .NET

Programowanie Wizualne

Paweł Wojciechowski

Instytut Informatyki, Politechniki Poznańskiej

2023

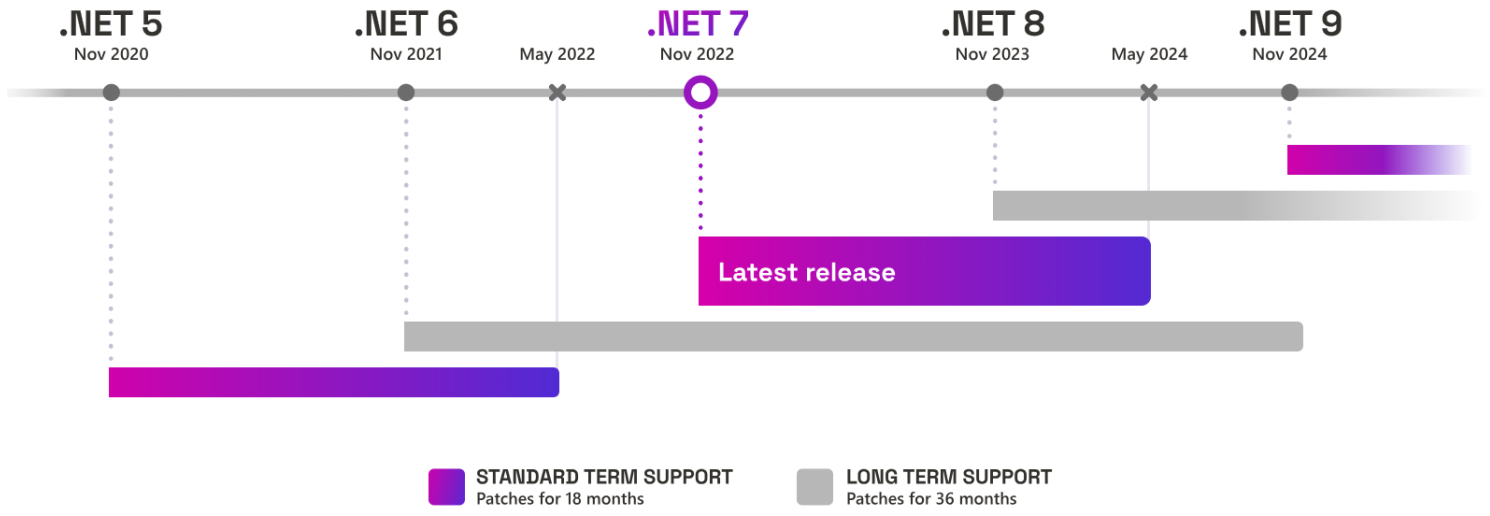
.NET główne cechy

- Microsoft .NET (w tym C#) powstał ok 2002 roku
- Cechy platformy .NET:
  - współpraca z istniejącym kodem (dzięki .NET Standard)
  - wsparcie dla wielu języków programowania
  - pełna integracja języka – wsparcie międzyjęzykowe: dziedziczenia, obsługi wyjątków, debugowania
  - pełna biblioteka klas bazowych
  - uproszczony model wdrażania – brak wpisu do rejestru

https://learn.microsoft.com/en-us/lifecycle/products/microsoft-net-and-net-core

.NET cykl życia

- Pojęcia:
  - .NET Standard
  - .NET Framework
  - .NET Core
  - .NET
  - .NET Runtime
- https://learn.microsoft.com/en-us/lifecycle/products/microsoft-net-and-net-core

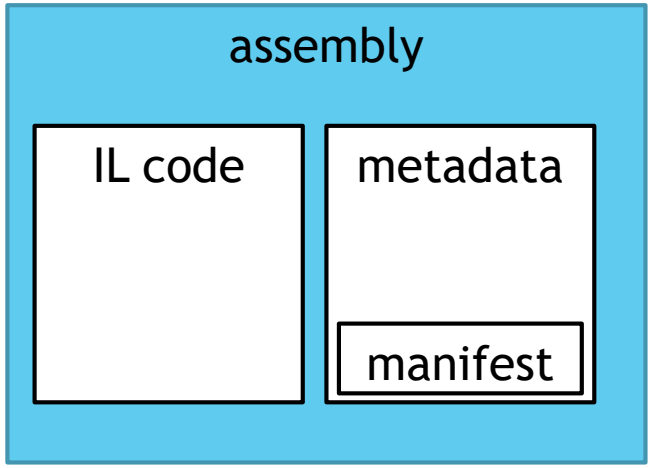


Podstawowe składowe .NET

- Common Language Runtime – warstwa uruchomieniowa – lokalizacja, zarządzanie i wczytywanie obiektów .NET., zarządzanie pamięcią, koordynacja działania wątków itp..
- Common Type System – definiuje wszystkie typy obsługiwane przez CLR i interakcję między nimi
- Common Language Specification – wymagania odnośnie języka programowania
- Klasy bazowe dostępne w .NET np. ASP.NET, WCF, WPF, ADO.NET itd..

Pojęcie Assembly

- program/biblioteka napisana w .NET jest kompilowana do IL (Intermediate Language)/CIL (C – Common)/MSIL
- metadata – kompletny opis wszystkich typów włącznie z wszystkimi jego elementami (metody, zdarzenia, właściwości itd.)
- manifest – opis całego pakietu



Common Type System (CTS)

- każdy pakiet może zawierać dowolną liczbę *typów:*
  - klasa
  - interfejs
  - struktura
  - typ wyliczeniowy (enumeration)
  - delegat (odpowiednik wskaźnika na funkcję)
- składowe typów:

  constructor, finalizer, static constructor, typ zagnieżdżony, operator, metoda, właściwość (property), indexer, pole (field), stała (constant), zdarzenie
  dekorator składowej np. zakres widzialności
- nazwy typów CTS: System.Single, VB: Single, C#: float

Common Language Specification

- CLS jest zbiorem zasad opisujących minimalny ale kompletny zbiór cech, które muszą zostać spełnione aby kompilator .NET był w stanie wygenerować kod dla CLR.
- CLS jest najczęściej podzbiorem CTS
- Specyfikacja CLS jest interesująca przede wszystkim dla osób piszących kompilatory
- Specyfikacja CLS jest zbiorem reguł, przy czym najważniejsza brzmi:

reguły CLS dotyczą tylko tych części typu, które są widoczne poza definiującym je pakietem (assembly).

- Atrybut

```
[assembly: CLSCompliant(true)]
[CLSCompliant(true)]
```

Common Language Runtime

- kolekcja usług umożliwiająca uruchomienie kodu IL na docelowej platformie
- kod CIL jest kompilowany przez JIT na docelową platformę
- istnieje możliwość wcześniejszej prekompilacji kodu w czasie instalacji pakietu – wykorzystanie np. dla dużych, wymagających aplikacji

.NET poza systemem Windows

- projekt Mono
- Xamarin – projekt wyrósł na bazie Mono i umożliwia tworzenie wieloplatformowych interfejsów GUI dla aplikacji mobilnych. SDK jest darmowe, ale cały Xamarin już nie.
- Microsoft .NET (Core)– skupiono się na wieloplatformowej implementacji bibliotek związanych z dostępem do danych, aplikacji web-owych, serwisów web-owych
- wieloplatformowy edytor kodu Visual Studio Code

Pierwszy program C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Wyklad1
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = default;
            return x;
        }
    }
}
```

Inne wersje metody main:

```
static int Main(string[] args)  { return 0; }

static void Main() { }

static int Main() { return 0; }
```

Uwagi:

- argumenty
- argumenty w VS
- zwracana wartość
- C# 9 top-level statesments

Metoda main - Quiz

Pytanie 1:

```
class Program2
{
    static void Main( string[] args)
    {
        Console.WriteLine("program 2");
    }

    static int Main(string[] args)
    {
        return 0;
    }

}
```

Pytanie 2:

```
class Program
{
    static int Main(string[] args)
    {
        int x = default;
        return x;
    }
}

class Program2
{
    static void Main(string[] args)
    {
        Console.WriteLine("program 2");
    }

}
```

Top-level statement

- tylko jeden plik może być utworzony w tej konwencji
- program nie może używać punktu wejścia (entry – point)
- Taki kod nie może być zamknięty w ramach namespace
- Wciąż mamy dostęp do argumentów uruchomieniowych (tablica args)
- Typy mogą być definiowane po konstrukcji top-level statement – definicja przed prowadzi do błędu kompilacji

Global using statement (C#10)

- Możemy podać wszystkie biblioteki które mają być wykorzystywane przez każdą z klas
- global using System;
- Wszystkie takie biblioteki muszą być podane przed innymi bibliotekami
- Domyślne biblioteki – zależne od rodzaju projektu
- Użycie można zmienić w pliku projektu <ImplicitUsing> wartości enabled:disabled
- Wszystkie biblioteki są zdefiniowane w katalogu \obj\Debug\net6.0 w pliku *ProjectName*.GlobalUsing.g.cs

Przestrzenie nazw z zakresem dla pliku
File scoped namespaces (C#10)

```
namespace Wykład_1;
internal class Class1
{
}
```

```
namespace Wykład_1
{
    internal class Class1
    {
    }
}
```

Klasa System.Console

- Console.ReadLine()

```
string input = Console.ReadLine();
```

- Console.WriteLine()

```
int x = 100;
Console.WriteLine("Program 2\nWprowadź tekst:");
string input = Console.ReadLine();
Console.WriteLine( "Twój {1} tekst to: {0}", input, x);
```

- formatowanie danych
  - c – format dla waluty
  - d – liczby dziesiętne
  - e – format wykładniczy
  - f – liczby zmiennoprzecinkowe
  - n – liczby z przecinkami
  - x – format heksadecymalny

Klasa System.Console

```
Console.WriteLine("{{0:c}}:  {0:c}", x);                                        {0:c}:  255,33 zł
Console.WriteLine("{{0:c1}}:  {0:c1}", x);                                      {0:c1}:  255,3 zł
Console.WriteLine("{{0}}:  {0}", x.ToString("c",CultureInfo.CreateSpecificCulture("en-us"))); {0}:  $255.33
Console.WriteLine("{{0:d}}:  {0:d}", (int)x);                                   {0:d}:  255
Console.WriteLine("{{0:d4}}:  {0:d4}", (int)x);                                 {0:d4}:  0255
Console.WriteLine("{{0:e}}:  {0:e}", x);                                        {0:e}:  2,553300e+002
Console.WriteLine("{{0:e2}}:  {0:e2}", x);                                      {0:e2}:  2,55e+002
Console.WriteLine("{{0:f}}:  {0:f}", x);                                        {0:f}:  255,33
Console.WriteLine("{{0:f3}}:  {0:f3}", x);                                      {0:f3}:  255,330
Console.WriteLine("{{0:n}}:  {0:n}", 123456789.ToString("N"));                  {0:n}:  123 456 789,00
Console.WriteLine("{{0:n}}:  {0:n3}", 123456789.ToString("N3"));                {0:n3}:  123 456 789,000
Console.WriteLine("{{0:x}}:  {0:x}", 255);                                      {0:x}:  ff
Console.WriteLine("{{0:X}}:  {0:X}", 255);                                      {0:X}:  FF
```

Podstawowe typy danych

- typy wartościowe ValueTypes:
  - typy proste
    - liczby znakiem: sbyte, short, int, long
    - liczby bez znaku: byte, ushort, uint, ulong
    - char
    - zmiennoprzecinkowe: float, double, decimal (128 bitowe wysokiej precyzji)
    - logiczne: bools
  - wyliczeniowe: enum
  - struktury
  - nullable
- typy referencyjne:
  - klasy
  - string
  - interfejsy
  - tablice
  - delegaty

https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/types-and-variables

Typy danych

- wszystkie typy dziedziczą po klasie Object – domyślnie mają zaimplementowane metody: `ToString(), Equals(), GetHashCode(), GetType()`
- typy wartościowe dziedziczą dodatkowo po klasie ValueType
- wszystkie typy proste są zapieczętowane (sealed)
- typy liczbowe mają właściwości: `MinValue/MaxValue`
- typy zmiennoprzecinkowe mają dodatkowo: `PositiveInfinity, NegativeInfinity, Epsilon` (najmniejsza wartość dodatnia, która jest większa od 0)
- typ `char` ma metody statyczne do sprawdzania czy znak jest liczbą, znakiem kontrolnym itd.
- typy liczbowe i `boolean` mają metodę `Parse` i `TryParse`
- Biblioteka `System.Numerics` ma dodatkowo typy: `BigInteger, Complex` i inne

Deklaracja zmiennych

Pytanie 3: czy tak można użyć zmiennej a?

```
static void Main()
{
    int y = 1;
    int a;
    Console.WriteLine(a);
}
```

- (C#7.1) użycie słowa `default`

```
int a = default;
```

- (C#7) separator liczbowy

```
int i = 123_345_000;
int j = 0b0010_0000;
int k = 0x000A_00FF;
```

Klasa String

- metody: `Compare()`, `Contains()`, `Trim()`, `ToUpper()`, `ToLower()`, `Format()`, `Equals()`, `Split()`
- właściwość: `Length`
- Łączenie obiektów klasy `String`: operator `+` (`+=`), metoda `Concat()`
- definiowanie łańcuchów „dosłownych”

```
string s = @"Ala ma ""kota""\t, a kot ma Alę\n";
```

- obiekty klasy string są niezmienialne -> za każdym razem tworzony jest nowy obiekt -> klasa `StringBuilder`
- C#6 string interpolation

```
string name = "Ala";
int age = 10;
string s1 = string.Format("{0} ma kota i {1} lat.", name, age);
string s2 = $"{name} ma kota i {age} lat.";
string s3 = $"{name.ToLower()} ma kota i {age+1} lat.";
```

- C#8 połączenie verbatim z interpolation: `$@` albo `@$`

Konwersja typów

```
static int Dodaj( int x, int y) { return x + y; }
static void Main()
{
    byte b1 = 10;
    byte b2 = 20;

    Console.WriteLine( Dodaj(b1, b2));
}
byte b3 = Dodaj(b1, b2); //błąd kompilacji
```

Pytanie 4: Czy taka metoda jest poprawna?

```
static short Dodaj2( short x, short y) { return x + y; }
```

- zawężanie zakresu powoduje wystąpienie błędu – potrzebna jest konwersja
- jawna konwersja ignoruje przekroczenie zakresu
- checked – generuje wyjątek przy przekroczeniu zakresu

```
byte b3 = checked( (byte)Dodaj(b1, b2));
```

- opcję checked można włączyć w ustawieniach projektu – wyłączenie bloku słowem kluczowym unchecked

Niejawnie typowane zmienne lokalne

- słowo var

```
static int Dodaj( int x, int y) { return x + y; }
static void Main()
{
    int i = 0;
    bool b = true;
    string s = "ala ma kota";
    int z = Dodaj(1, 2);
}
```

```
static int Dodaj( int x, int y) { return x + y; }
static void Main()
{
    var i = 0;
    var b = true;
    var s = "ala ma kota";
    var z = Dodaj(1, 2);
}
```

- Jak nie używać zmiennych niejawnie typowanych:
  - jako typu pola w klasie
  - jako parametru wejściowego metody albo wartości zwracanej
  - bez natychmiastowego przypisania wartości
- takie zmienne są silnie typowane!!!

```
var i = 0;
i = 11; // OK

i = "ala"; //ERROR
```

Pętle z dopasowywaniem wzorców (C#7,C#9)

- słowo kluczowe `is` normalnie służy do sprawdzania, czy obiekt implementuje interfejs/jakiego jest typu

```
public struct Kwadrat
{
    public double Bok { get; }

    public Kwadrat( double bok)
    {
        Bok = bok;
    }
}

public struct Kolo
{
    public double Promien { get; }

    public Kolo( double promien)
    {
        Promien = promien;
    }
}
```

```
static double PolePowierzchni( object s)
{
    if ( s is Kolo )
    {
        var o = (Kolo) s;
        return o.Promien * o.Promien * Math.PI;
    }
    else if ( s is Kwadrat)
    {
        var k = (Kwadrat)s;
        return k.Bok * k.Bok;
    }

    throw new Exception("Obiekt nie został rozpoznany");
}
```

Pętle z dopasowywaniem wzorców (C#7)

- nowa składnia z operatorem is

```
static double PolePowierzchni2( object s)
{
    if (s is Kolo o)
        return o.Promien * o.Promien * Math.PI;
    else if (s is Kwadrat k)
        return k.Bok * k.Bok;

    throw ...
}
```

- nowa składnia switch/case – w C#7 dodano m.in. możliwość sprawdzenia typu

```
static double PolePowierzchni3( object s)
{
    switch( s)
    {
        case Kwadrat k:
            return k.Bok * k.Bok;
        case Kolo o:
            return o.Promien * o.Promien * Math.PI;
        default:
            throw new Exception();
    }
}
```

```
static double PolePowierzchni4( object s)
{
    switch( s)
    {
        case Kwadrat k when k.Bok <= 0:
        case Kolo o when o.Promien <= 0:
            return 0;

        ...
    }
}
```

```
static double PolePowierzchni(object s)
{
    if (s is Kolo kolo)
    {
        return kolo.Promien * kolo.Promien * Math.PI;
    }
    else if (s is Kwadrat kwadrat)
    {
        return kwadrat.Bok * kwadrat.Bok;
    }

    throw new Exception("Object not recognized");
}

public struct Kwadrat
{
    public double Bok { get; }

    public Kwadrat(double bok)
    {
        Bok = bok;
    }
}

public struct Kolo
{
    public double Promien { get; }
    public Kolo(double promien)
    {
        Promien = promien;
    }
}
```

Błąd: Użyto nieprzypisanej zmiennej lokalnej „kolo”

```
static double PolePowierzchni(object s)
{
    if (s is Kolo k)
    {
        return k.Promien * k.Promien * Math.PI;
    }
    else if (s is Kwadrat k)
    {
        return k.Bok * k.Bok;
    }

    throw new Exception("Object not recognized");
}
```

Błąd: Element lokalny lub parametr o nazwie „k” nie może zostać zadeklarowany w tym zakresie, ponieważ ta nazwa jest już użyta w otaczającym zakresie lokalnym do zdefiniowania elementu lokalnego lub parametru

```
static double PolePowierzchni(object s)
{
    if (s is Kolo kolo)
    {
        return kolo.Promien * kolo.Promien * Math.PI;
    }
    else if (s is Kwadrat kwadrat)
    {
        Console.WriteLine($"{kolo.Promien}");
        return kwadrat.Bok * kwadrat.Bok;
    }

    throw new Exception("Object not recognized");
}
```

Instrukcja switch C#8

```
string GetColor( string color)
{
    switch (color)
    {
        case "red": return "#FF0000";
        default: return "";

    }
}

string GetColor2(string color)
{
    return color switch
    {
        "red" => "#FF0000",
        _ => "",
    };
}


//C#9
char c = 't';
if ( c is >= 'a' and <= 'z' or >= 'A' and <= 'Z')
    Console.WriteLine("c is character");
```

Relational patterns C#9

```
static decimal GetGroupTicketPriceDiscount(int groupSize, DateTime visitDate)
    => (groupSize, visitDate.DayOfWeek) switch
    {
        ( <= 0, _) => throw new ArgumentException("Group size must be positive."),
        (_, DayOfWeek.Saturday or DayOfWeek.Sunday) => 0.0m,
        ( >= 5 and < 10, DayOfWeek.Monday) => 20.0m,
        ( >= 10, DayOfWeek.Monday) => 30.0m,
        ( >= 5 and < 10, _) => 12.0m,
        ( >= 10, _) => 15.0m,
        _ => 0.0m,
    };
```

Tablice w C#

- deklaracja tablicy

```
int[] tab = new int[3];
```

- inicjalizacja tablicy

```
int[] tab2 = new int[] { 1, 2, 3 };
```

- tablice wielowymiarowe

```
int[,] mtab = new int[2, 2] { { 0, 1 },{ 1, 2 } };

int[][] mtab2= new int[3][];

for( int i=0; i< mtab2.Length; i++)
{
    mtab2[i] = new int[i + 1];
}
```

- tablice mogą być argumentami albo wartościami zwracanymi przez metody
- Domyślne metody: `Clear(), Sort(), Reverse(), Length`

Pytanie 5: Czy można tak użyć elementu tablicy?

```
static void Main()
{
    int[] tab = new int[3];

    Console.WriteLine(tab[0]);
}
```

Pytanie 6: Czy poniższe tablice są poprawnie zainicjalizowane?

```
int[] tab3 = new int[2] { 1, 2, 3 };

int[] tab4 = new int[4] { 1, 2, 3 };
```

Modyfikatory parametrów metod

- Pytanie 7: jakie będą wartości zmiennych i oraz j w ostatniej linii programu?

```
static int Dodaj( int x, int y)
{
    int wynik = x + y;

    x = 1;
    y = 2;
    return wynik;
}

static void Main()
{
    int i = 3;
    int j = 4;
    Console.WriteLine(Dodaj(i, j));
    Console.WriteLine(i + " " + j);

}
```

Modyfikatory parametrów metod

- Pytanie 7: jakie będą wartości zmiennych i oraz j w ostatniej linii programu?

```
static int Dodaj( int x, int y)
{
    int wynik = x + y;

    x = 1;
    y = 2;
    return wynik;
}

static void Main()
{
    int i = 3;
    int j = 4;
    Console.WriteLine(Dodaj(i, j));
    Console.WriteLine(i + " " + j);

}
```

Modyfikatory parametrów metod: out

- modyfikator out:
  - parametr wyjściowy MUSI być przypisany w ciele metody
  - przy wywołaniu trzeba podać jawnie modyfikator
  - zmienna nie musi być zainicjalizowana

```
static void Dodaj2( int x, int y, out int wynik)
{
    wynik = x + y;
}
static void Main()
{
    int i = 3;
    int j = 4;
    int wynik;
    Dodaj2(i, j, out wynik);
}
```

```
static void Main()
{
    int i = 3;
    int j = 4;
    Dodaj2(i, j, out int wynik2);
    Console.WriteLine(wynik2);
}
```

  - od C#7 zmienna nie musi być zadeklarowana
  - odrzucenie zmiennej (discard)

```
Dodaj2(i, j, out _);
```

Modyfikatory parametrów metod: ref

- modyfikator ref:
  - przy wywołaniu trzeba podać jawnie modyfikator
  - zmienna musi być zainicjalizowana

```
static int Dodaj3(ref int x, ref int y)
{
    int wynik = x + y;

    x = 1;
    y = 2;
    return wynik;
}

static void Main()
{
    int i = 3;
    int j = 4;

    Console.WriteLine(Dodaj3(ref i, ref j));
    Console.WriteLine(i + " " + j);
}
```

Modyfikatory parametrów metod: ref

- C#7 modyfikator ref – wartość zwracana odwołania
- zmienna lokalna nie może zostać zwrócona
- funkcjonalność nie działa z metodami async

```
static int[] tab = new int[] { 1, 2, 3, 4 };

public static ref int ZnajdzParzysta()
{
    for( int i=0 ; i < tab.Length; i++)
    {
        if (tab[i] % 2 == 0)
            return ref tab[i];
    }
    return ref tab[0];
}

static void Main()
{
    foreach( var it in tab)
        Console.Write( $"{it} ");
    Console.WriteLine();
    ref int refValue = ref ZnajdzParzysta();
    refValue += 1;
    foreach (var it in tab)
        Console.Write($"{it} ");
    Console.WriteLine();
}
```

Modyfikatory parametrów metod: params

```
static int Suma( int x, int y, params int[] wartosci)
{
    int sum = x + y;
    foreach (var i in wartosci)
        sum += i;
    return sum;
}

Console.WriteLine( Suma( 1,2, 3, 4, 5));
Console.WriteLine( Suma( 3,4));
```

Pytanie 8: Czy można w ten sposób używać modyfikatora params?

```
static int Suma( params int[] wartosci, params float[] wartosci2)

static int Suma( params int[] wartosci, int x, int y)
```

Modyfikatory parametrów metod: in

- w C#7.2 wprowadzono modyfikator `in`
- służy on do przekazywania przez referencję bez możliwości zmiany – oszczędność pamięci
- przy wywołaniu metody z argumentami typu in nie trzeba podawać modyfikatora (w przeciwieństwie do modyfikatorów ref i out)
- nie ma sensu używać tego modyfikatora do typów referencyjnych
- w przypadku struktur nie można bezpośrednio zmieniać wartości pól (dla klas można)

```
public struct Person
{
    public string Imie;
    public string Nazwisko;
}
```

```
static string ShowPerson(in Person p)
{
    // p.Imie = "Ola";
    // p = new Person();

    return p.Imie + " " + p.Nazwisko;
}
```

```
Person p = new Person();
p.Imie = "Ala";
p.Nazwisko = "Nowak";

Console.WriteLine(ShowPerson( p));
```

Wartości domyślne parametrów i parametry nazwane

- wartości domyślne

```
static int oblicz(int x, int y = 1)
{
    return x + y;
}

static void Main()
{
    int z = 3;
    Console.WriteLine(oblicz( z ));
}
```

Pytanie 9: Jaki będzie wynik działania programu gdy dodamy poniższy kod?

```
static int oblicz(int x)
{
    return x * x;
}
```

- parametry nazwane:
  - można nadać wartości parametrów metody w dowolnej kolejności używając nazw zmiennych np.:

```
Console.WriteLine(oblicz( y: z, x:3 ));
```

Pytanie 10: Które wywołania są poprawne? – uwaga od C# 7.2 nastąpiła zmiana

```
oblicz( 1, x: 3);
oblicz( 1, y: 3);
oblicz( y: 3, 1);
```