

# C#

## Podstawy programowania – cz. 1



# HELLO

## Michał Sosnowski

Senior C# Developer at OKE Software







Jest to obiektowy język programowania zaprojektowany w latach 1998–2001 dla firmy Microsoft, która do dziś zajmuje się rozwojem języka. Obecnie najnowszą wersją jest wersja 11.

Nazwa C# wzięła się z tego że jest rozwinięciem języka C++ gdzie ten był rozwinięciem języka C - cztery plusy zapisano tak by tworzyły razem znak #





Jest to platforma programistyczna  
opracowana przez Microsoft obejmująca  
środowisko uruchomieniowe m.in.  
kompilator oraz biblioteki klas  
dostarczające standardowe  
funkcjonalności dla aplikacji.

A large, stylized ".NET" logo in white, centered within a solid blue square. The logo is composed of a small dot followed by the letters "NET" in a bold, sans-serif font.



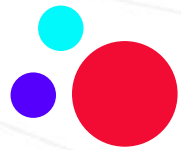
# .NET Framework, .NET Core

## .NET Framework

Pakiet bibliotek służący do tworzenia aplikacji na systemy Windows. Jego najnowsza (i ostatnia) wersja to 4.8.1.

## .NET Core

Otwartoźródłowy pakiet bibliotek pozwalający na tworzenie aplikacji konsolowych i webowych (zarówno backend jak i frontend) na wielu systemach.



APP  
MODELS

## .NET FRAMEWORK

WPF

Windows  
Forms

ASP.NET

## .NET CORE

UWP

ASP.NET Core

## XAMARIN

iOS

OS X

Android

## .NET STANDARD LIBRARY

One library to rule them all

## COMMON INFRASTRUCTURE

Compilers

Languages

Runtime components



W 2020 Microsoft wprowadził wersję piątą .NET, która zunifikowała .NET Framework oraz .NET Core. **Obecnie najnowszą wersją .NET jest wersja 7.**

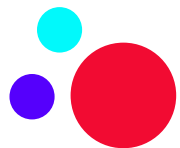
Mimo to .NET Framework wciąż jest rozwijany o m.in. funkcję poprawiające bezpieczeństwo głównie dla wciąż rozwijanych i utrzymywanych aplikacji desktopowych oraz serwerowych napisanych w oparciu o niego.





# .NET – A unified platform



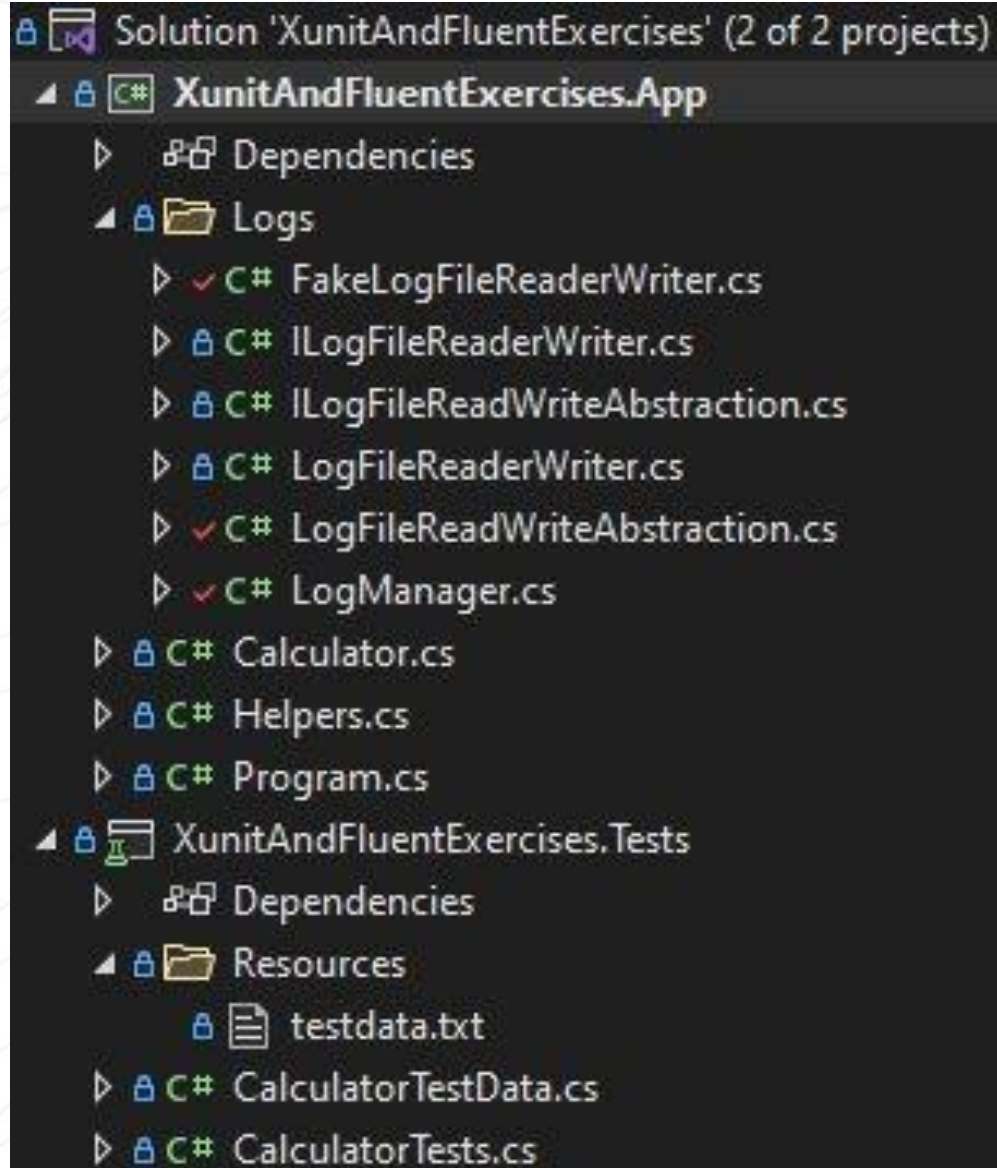


# Praca z projektami w Visual Studio

Projekt to podstawowy kontener używany przez deweloperów do organizowania plików kodu źródłowego i innych zasobów. Plik o rozszerzeniu .csproj zawiera informacje o zasobach wchodzących w skład projektu, odwołania do zewnętrznych bibliotek lub innych projektów oraz o typie projektu (aplikacja konsolowa, biblioteka, aplikacja okienkowa itp.)

Jeśli plik .csproj nie zawiera informacji o zasobach wtedy Visual Studio dołącza do obiektu wszystkie zasoby które są w tym samym katalogu co plik .csproj poza katalogami bin i obj które zawierają pliki wynikowe które utworzył kompilator.

Jest to kontener, który Visual Studio używa do organizowania projektów. Plik rozwiązania który posiada rozszerzenie .sln, zawiera informacje o wszystkich projektach jaki rozwiązanie zawiera. Po otwarciu pliku rozwiązania przez Visual Studio ten próbuje automatycznie załadować wszystkie zawarte w rozwiązaniu projekty.



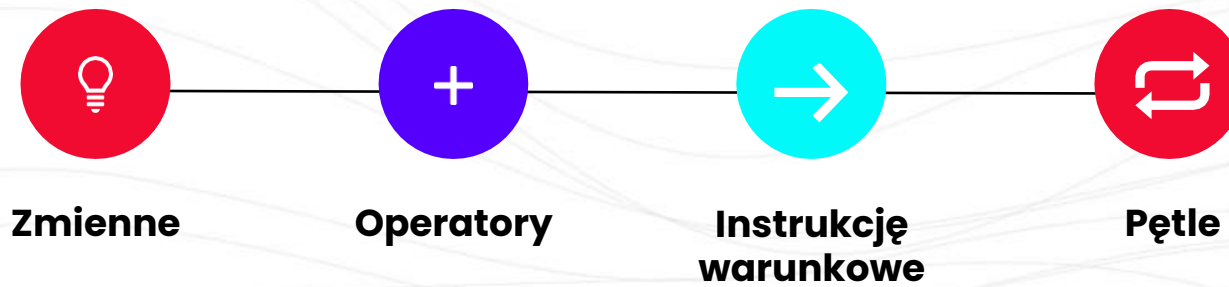


# Thank you!

You can find me at  
#SLACK or [sosna321@gmail.com](mailto:sosna321@gmail.com)

# C#

## Podstawy programowania – cz. 2





Jest to obszar w pamięci komputera posiadający jakąś wartość. W języku C# każda zmienna ma określony typ danych od którego zależy jak dużo będzie przechowywać pamięci jaki jest jej zakres wartości oraz jakie operacje będzie można wykonać na niej wykonać.

[Variables - C# language specification | Microsoft Learn](#)





# Podstawowe typy zmiennych

**int** – typ przechowujący liczby całkowite z zakresu od  $-2\,147\,483\,648$  do  $+2\,147\,483\,647$  ( $-2^{32}$ ,  $2^{32}$ ), zajmuje 4 bajty w pamięci

**long** – typ przechowujący liczby całkowite z zakresu od  $-2^{64}$  do  $2^{64}$ , zajmuje 8 bajtów w pamięci

**float** – typ przechowujący liczby zmiennoprzecinkowe, może przechowywać 8 miejsc po przecinku, zajmuje 4 bajty pamięci

**double** – typ przechowujący liczby zmiennoprzecinkowe o podwójnej precyzji może przechowywać do 16 miejsc po przecinku zajmuje 8 bajtów pamięci

**bool** – typ przechowujący wartość true lub false, zajmuje 1 bajt pamięci

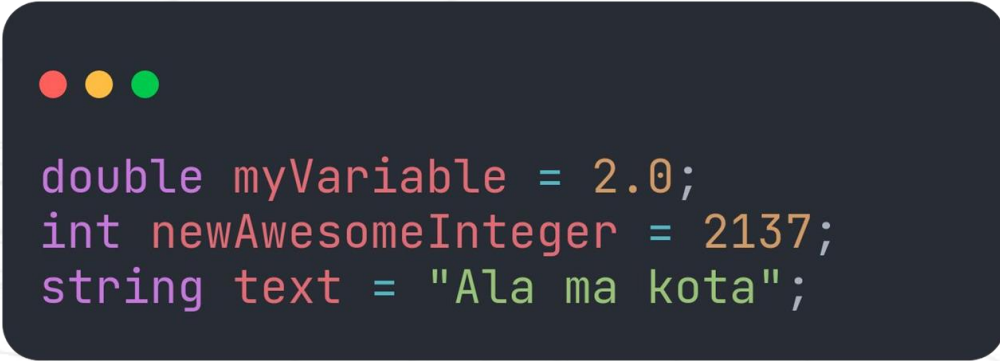
**char** – typ przechowujący znak tekstowy, zajmuje 1 bajt pamięci

**string** – typ przechowujący ciąg znaków tekstowych, ma zmienny rozmiar



# Definiowanie zmiennej

Aby zdefiniować nową zmienną należy napisać typ zmiennej jaką chcemy utworzyć a następnie jej nazwę. Nazwa nie może zaczynać się od cyfry.



```
double myVariable = 2.0;  
int newAwesomeInteger = 2137;  
string text = "Ała ma kota";
```

# Typy wartościowe i referencyjne

## Typy wartościowe

Są to typy które mają stały rozmiar pamięci, są umieszczane na stosie oraz do funkcji i metod jest przekazywana ich kopia.

## Typy referencyjne

To są typy które nie mają stałego rozmiaru pamięci. W praktyce na stosie umieszczany jest adres pamięci który prowadzi do wartości natomiast na sterckie umieszczany jest ten obszar pamięci. Do funkcji i metod jest przekazywany ten obszar (wyjątkiem tutaj jest typ string który w tym przypadku jest traktowany jako typ wartościowy).



# Jak ćwiczyć podstawy programowania po zajęciach?

- [Write your first code using C# \(Get started with C#, Part 1\) - Training | Microsoft Learn](#)
- <https://www.codewars.com/>



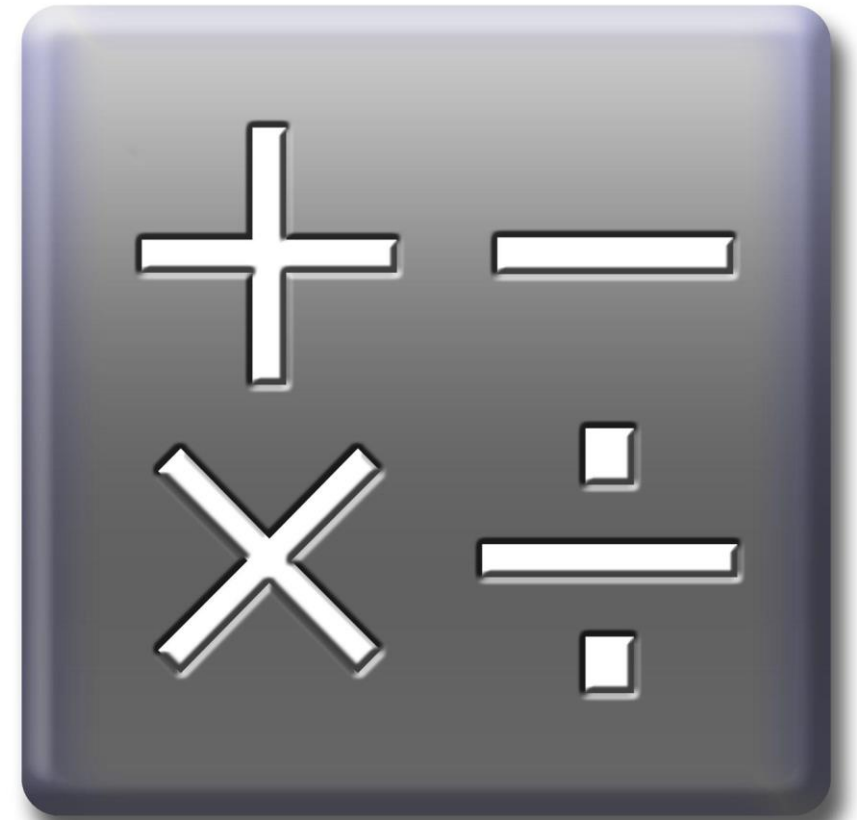
[infoShareAcademy.com](https://infoShareAcademy.com)

**infoShare**  
ACADEMY



# **Operator**

Operatory to symbole które informują jakie działania logiczne czy matematyczne mają zostać wykonane.





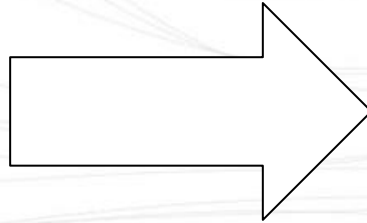
# Rodzaje operatorów

- = - operator przypisania, przypisuje wartość do zmiennej
- + - operator dodawania
- - - operator odejmowania
- \* - operator mnożenia
- / - operator dzielenia
- % - operator modulo, zwraca resztę z dzielenia dwóch liczb
- == - operator porównania - zwraca true jeśli wartości są równe
- != - operator rozróżnienia - zwraca true jeśli wartości są różne



# Skrócone operatory

```
int a = 5;  
a = a + 2;  
a = a - 1;  
a = a * 5;  
a = a / 2;
```




```
int a = 5;  
a += 2;  
a -= 1;  
a *= 5;  
a /= 2;
```



# Inkrementacja i dekrementacja

Inkrementacja i dekrementacja jest to zmienienie wartości zmiennej o 1. Możemy je podzielić także na pre-inkrementacje i post-inkrementacje oraz pre-dekrementacje i post-dekrementacje. Post-inkrementacja najpierw zwróci wartość a następnie podniesie jej wartość o 1 natomiast pre-inkrementacja podniesie wartość o 1 a potem zwróci wartość.



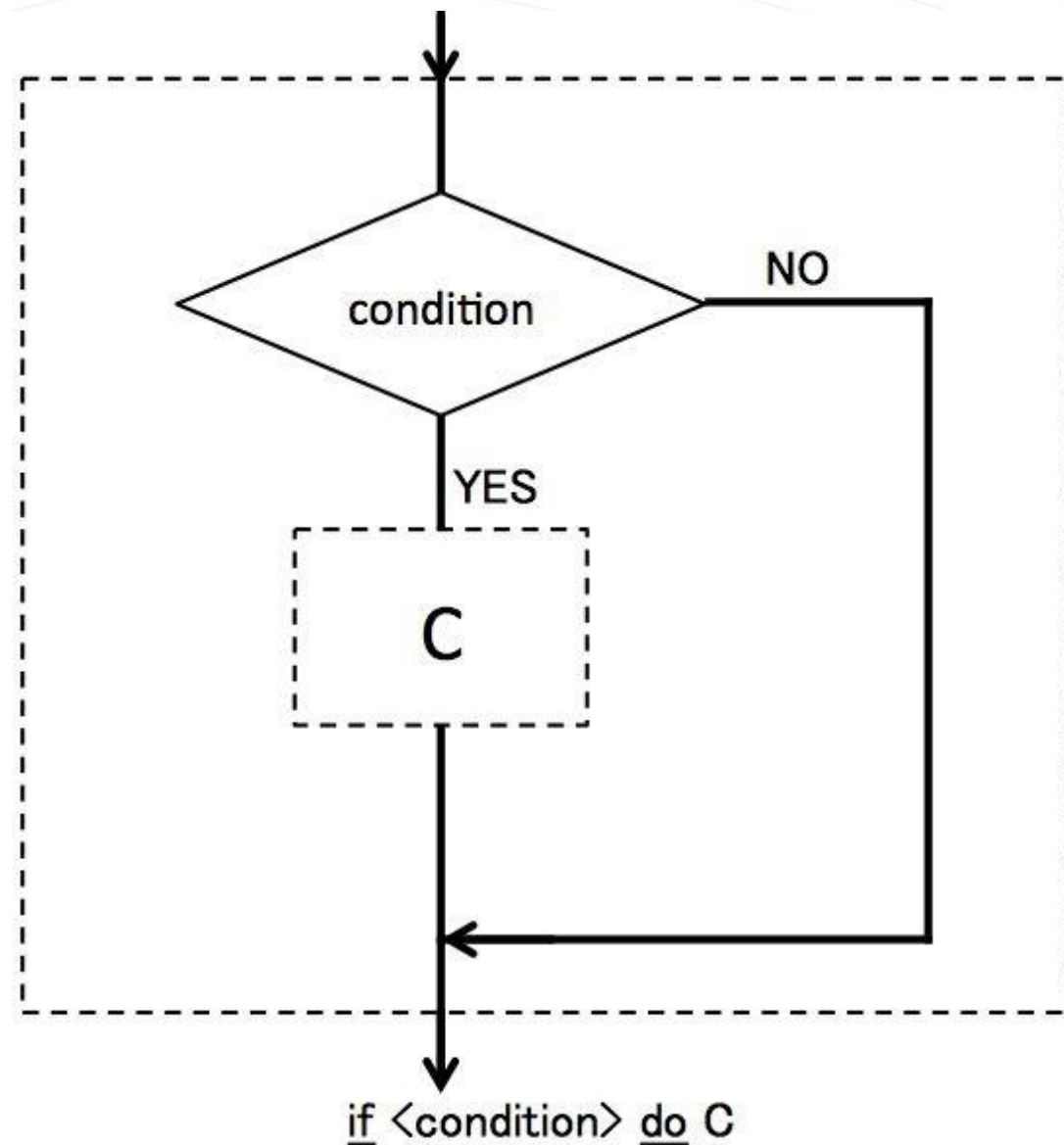
```
int i = 4;
Console.WriteLine($"Wartość: {i++}");
// Wartość: 4
Console.WriteLine($"Wartość: {++i}");
// Wartość: 6
Console.WriteLine($"Wartość: {--i}");
// Wartość: 5
Console.WriteLine($"Wartość: {i--}");
// Wartość: 5
Console.WriteLine($"Wartość: {i}");
// Wartość: 4
```





# Instrukcje warunkowe

Są to instrukcje w języku C#, które pozwalają zdefiniować nam kod który ma wykonać się po spełnieniu jakiegoś warunku.





Instrukcja warunkowa która służy do definiowania kodu który ma zostać wykonany po spełnieniu jakiegoś warunku. Warunek jest to działanie logiczne które zwraca wartość bool lub nawet zmienna typu bool. Mówimy o spełnieniu warunku jeśli ma on wartość true.

```
if(i == 0)
{
    Console.WriteLine("Liczba jest równa 0");
}
```

Instrukcja pisana wraz z instrukcją if –  
definiuje kod który ma się wykonać  
alternatywnie jeśli warunek zdefiniowany  
w ifie nie zostanie spełniony.

```
if(i == 0)
{
    Console.WriteLine("Zmienna równa 0");
}
else
{
    Console.WriteLine("Zmienna różna od 0");
}
```



## else if

Możemy połączyć instrukcję if i else tak aby zdefiniować kod, który wykona się gdy pierwszy warunek nie zostanie spełniony jednak spełniony zostanie warunek alternatywny.

```
if(i == 0)
{
    Console.WriteLine("Zmienna równa 0");
}
else if(i > 0)
{
    Console.WriteLine("Zmienna jest dodatnia");
}
else
{
    Console.WriteLine("Zmienna jest ujemna");
}
```



# Operatory logiczne

Pozwalają wykonywać operację na zmiennych lub wartościach typu bool, co pozwala nam na łączenie kilku warunków w jeden.

! – operator negacji – odwraca wartość

&& – operator AND – zwraca true tylko jeśli wszystkie wartości są true

|| – operator OR – zwraca true jeśli jedna z wartości jest true





# Operator logiczne

b	!b
true	false
false	true

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

a	b	a    b
true	true	true
true	false	true
false	true	true
false	false	false



Jest to instrukcja która sprawdza serię warunków dla jednej zmiennej. Zmienna ta musi jednoznacznie wartość liczby całkowitej więc może być typu int, long, char i enum.

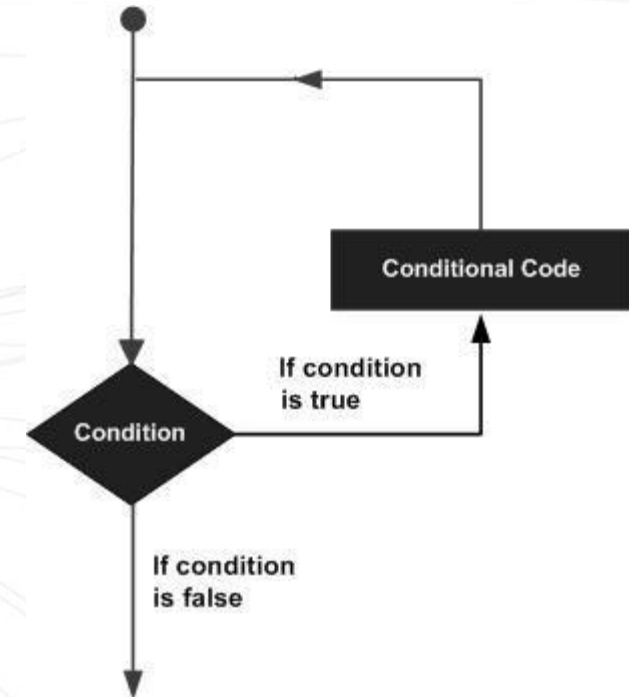
```
switch (zmienna)
{
    case 1: // jeśli zmienna równa 1
        WykonajInstrukcjeJeden();
        break; // ta instrukcja jest ważna inaczej program nie opuści switcha
    case 2:
        WykonajInstrukcjeTwo();
        break;
    case 3:
        WykonajInstrukcjeThree();
        break;
    default: // jeśli zmienna nie ma żadnych powyższych wartości
        Console.WriteLine("Nie mam instrukcji");
        break;
}
```



[infoShareAcademy.com](https://infoShareAcademy.com)

**infoShare**  
ACADEMY

Pętle wykonują wskazane instrukcje  
do momentu aż spełniony jest  
wskazany warunek.





# while

Podstawowa pętla w której definiujemy instrukcje które mają się wykonywać dopóki spełniony zostanie warunek.

```
int i=0;
while(i<5)
{
    Console.WriteLine($"i={i}");
    ++i;
}

// OUTPUT
// i=0
// i=1
// i=2
// i=3
// i=4
```



# Break i continue

## Break

Instrukcja break przerywa i natychmiast opuszcza całą pętlę niezależnie od tego czy jest spełniony jej warunek czy nie.

## Continue

Instrukcja continue przerywa obecny przebieg pętli i przechodzi do następnego obiegu jeśli warunek jest spełniony.



# do-while

Pętla podobna do while z tą różnicą że warunek sprawdzany jest po obiegu pętli a nie przed dlatego kod w pętli do-while zawsze wykona się co najmniej raz.

```
do
{
    Console.WriteLine("I tak się raz wykonam");
}while(false);

// Output
// I tak się raz wykonam
```



Jest to pętla w której definiujemy przy jej inicjalizacji instrukcję początkową, warunek oraz instrukcję po zakończeniu obiegu pętli.

*for(operacja do wykonania na starcie; warunek; operacja do wykonania po zakończeniu obiegu)*

```
for (int i = 0; i < 5; ++i)
{
    Console.WriteLine(i);
}
```

```
// OUTPUT
// 0
// 1
// 2
// 3
// 4
```



# For vs while



```
for (int i = 0; i < 5; ++i)
{
    Console.WriteLine(i);
}
```



```
int i = 0
while (i < 5)
{
    Console.WriteLine(i);
    ++i;
}
```

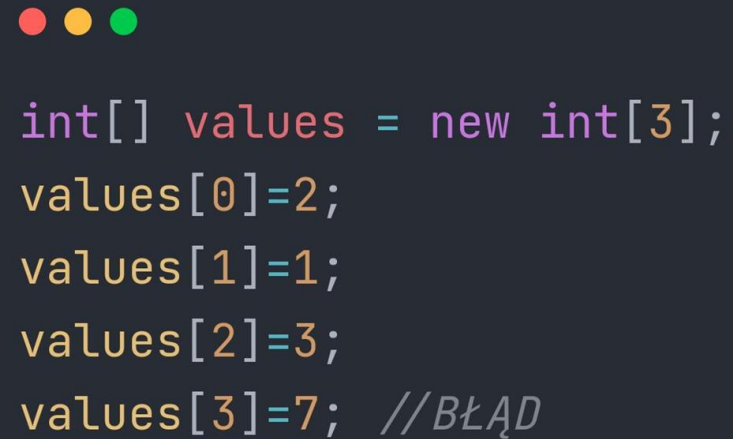


# Kolekcje

Są to klasy i struktury które pozwalają na przechowywanie wielu wartości.

Podstawową taką strukturą jest tablica  
- struktura która może przechowywać zdefiniowaną wcześniej liczbę wartości wskazanego typu.

Tablica posiada właściwość `.Length` która zwraca rozmiar tablicy czy ile zawiera elementów.



```
int[] values = new int[3];  
values[0]=2;  
values[1]=1;  
values[2]=3;  
values[3]=7; //BŁĄD
```





# Foreach

Jest to pętla która pozwala na wykonywanie kodu na elementach tablicy czy kolekcji.

W pętli foreach definiujemy zmienną będącą typu taki jak elementy kolekcji.

Przy każdym obiegu pętli zmienna ta przyjmuje wartość kolejnego elementu z kolekcji.

```
int suma = 0;
foreach(int item in values)
{
    suma += item;
}
```



# Foreach vs For

```
int suma = 0;
foreach(int item in values)
{
    suma += item;
}
```

```
int suma = 0;
for(int i=0; i < values.Length; ++i)
{
    suma += values[i];
}
```

# Thank you!

You can find me at  
#SLACK or [sosna321@gmail.com](mailto:sosna321@gmail.com)

# C#

## Podstawy programowania – cz. 3



[infoShareAcademy.com](https://infoShareAcademy.com)

**infoShare**  
ACADEMY





# Funkcja

Jest to fragment kodu który może być wywołany wiele razy z wielu miejsc kodu – funkcja składa się z typu jaki zwraca (lub void jeśli nie zwraca nic), nazwy pod którą ma być widoczna, parametrów jakie przyjmuje oraz kodu funkcji. Jeśli funkcja ma zwracać jakiś typ to każde możliwe jej zakończenie musi zwracać wartość instrukcją return lub wyrzucać wyjątek.

```
int Add(int a, int b)
{
    return a + b;
}

void PrintError(string error)
{
    Console.WriteLine($"Błąd! {error}");
}

int sum = Add(2,3);
// 5

PrintError("Nieznany format");
// Błąd! Nieznany format
```



Klasa jest to struktura opisująca obiekt danego typu, jakie ma właściwości, co może zrobić oraz jakie informacje może przechowywać.

Instancję danej klasy nazywamy obiektem.

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

    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public double GetLengthFromZero()
    {
        return Math.Sqrt(x * x + y * y);
    }
}
```



Obiekt nowej klasy inicjalizujemy  
poprzez słowo kluczowe **new**



```
Point point = new Point(2,3);  
Point anotherPoint = new(1,2);
```



# Konstruktor

Jest to specjalna metoda która jest wywoływana w momencie kiedy tworzona jest nowa instancja danej klasy. Klasa może posiadać kilka konstruktorów z różnymi zestawami parametrów. Zazwyczaj w konstruktorze przypisujemy parametry do pól.

```
public Point(int x, int y)
{
    this.x = x;
    this.y = y;
}

public Point()
{
    this.x = 0;
    this.y = 0;
}

Point p1 = new(2,3);
Point p2 = new(0,0);
```



# Modyfikatory dostępu

Klasy, Metody, pola i właściwości posiadają modyfikatory dostępu, które określają dostęp do nich dla innych obiektów. Wyróżniamy 4 modyfikatory dostępu

**private** – dostępność tylko dla aktualnego obiektu

**protected** – dostępność tylko dla obiektów które dziedziczą po klasie w której zdefiniowane została klasa \ pole \ właściwość \ metoda

**public** – dostępność dla wszystkich obiektów

**internal** – dostępność dla wszystkich obiektów znajdujących się w obrębie tego samego assembly (czyli w praktyce tylko w obrębie tego samego projektu)


Modyfikator dostępu umieszczamy przed definicją klasy/pola/właściwości czy metody, jeśli tego nie zrobimy zastosowany zostanie domyślny modyfikator czyli **private**. (Dla klas domyślny modyfikator to **internal**)



Jest to funkcja będąca wewnątrz klasy i mogąca być wywołana z obiektu. Metody mają dostęp do wszystkich pól, właściwości i innych metod klasy w której się znajdują.

```
public double GetLengthFromZero()  
{  
    return Math.Sqrt(x * x + y * y);  
}
```

Są to zmienne znajdujące się w klasie.



```
private int x;  
private int y;
```



# Właściwości

Jest to mechanizm, który pozwala na odczytywanie i modyfikowanie pól.

Tworzenie standardowej właściwości bez odniesienia do pola spowoduje niejawne utworzenie takiego pola.

```
public class Book
{
    public string Title {get; set;}
    public string Author {get; set;}
    public int PublishedYear {get; set;}
}
```



```
public class Student
{
    public string Name {get;set;}
    public bool Enrolled {get;set;}
}
```

```
public class Student
{
    private string name;

    public string Name
    {
        get
        {
            return name;
        }
        set
        {
            name = value;
        }
    }

    private string enrolled;

    public bool Enrolled
    {
        get
        {
            return enrolled;
        }
        set
        {
            enrolled = value;
        }
    }
}
```



# Backing fields

Jeśli we właściwości odnosimy się jawnie do istniejącego pola wtedy mówimy o tzw. backing fields.

Zazwyczaj używamy tego mechanizmu żeby zawrzeć dodatkową walidację wartości jaką chcemy nadać polu naszego obiektu lub inną logikę, która ma się wykonać jeśli pole zostanie zmodyfikowane (np. odświeżenie okna z aplikacją).

```
public class Book
{
    private string title;

    public string Title
    {
        get
        {
            return title;
        }
        set
        {
            if(!string.IsNullOrEmpty(value))
            {
                title = value;
            }
        }
    }
    ...

    private int publishedYear;
    public int PublishedYear
    {
        get
        {
            return publishedYear;
        }
        set
        {
            if(value <= DateTime.Now.Year)
            {
                publishedYear = value;
            }
        }
    }
}
```





# Klasa statyczna

Jest to klasa, z której nie możemy tworzyć obiektów, możemy wywoływać metody bezpośrednio definicji klasy.

```
public static class Calculator
{
    public static Add(int x, int y)
    {
        return x + y;
    }

    public static Subtract(int x, int y)
    {
        return x - y;
    }
}
```



# Zasięg zmiennych

Jeśli zmienna została zadeklarowana wewnątrz funkcji czy metody to tylko tam będzie ona dostępna - nazywamy wtedy taką zmienną lokalną.

W przypadku klasy domyślnie zmienna będzie dostępna w całej klasie ale dodając do niej modyfikator dostępu (public, protected, internal) możemy ją udostępnić również innym klasom które mają jakieś zależności z naszą klasą.

```
void MyFunc()
{
    int b = 5;
    ...
}

// błąd kompilacji
Console.WriteLine(b);
```



# Parametry ref, out

Domyślnie parametry do metod i funkcji są przekazywane przez wartość więc w praktyce do funkcji trafiają kopie zmiennych. Możemy jednak zdefiniować by parametry były przekazywane przez referencję używając słów kluczowych ref i out.

Różnica jest taka że parametr zadeklarowany jako ref musi być zadeklarowany przed wywołaniem metody i może być zmodyfikowany w metodzie natomiast out nie musi być zadeklarowany przed wywołaniem metody i musi być zmodyfikowany w metodzie.

```
void Increment(ref int value)
{
    value += 1;
}

bool TryGetInt(string input, out int value)
{
    value = 0;
    if(string.IsNullOrEmpty(input))
    {
        return false;
    }
    ...
    return true;
}

int counter = 0;
Increment(ref counter);
Console.WriteLine(counter); // 1
if(TryGetInt("23 kolumny", out int value))
{
    Console.WriteLine(value); // 23
}
```



# Parametry in

Zadeklarowanie parametru ze słowem kluczowym in oznacza, że wartość tego parametru nie będzie zmieniana w funkcji. W przypadku typów referencyjnych oznacza to że referencja parametru nie będzie zmieniana, ale zawartość pamięci wciąż może być zmieniona. Stosuje się deklaracje parametrów in w celu zwiększenia wydajności.

```
void Enroll(in Student student)
{
    // poniższa linia spowodowałaby błąd
    // student = new Student();

    student.Enrolled = true;
}

var student = new Student
{
    Name = "Susan",
    Enrolled = false
};

Enroll(student);
```



# Operacje na stringach







# Operacje na stringach

Typ string posiada wiele metod pozwalających na wykonywanie zaawansowanych operacji na stringach.

String w języku C# jest typem referencyjnym, który jednak zachowuje się jak typ wartościowy. Jest jednak niemutowalny, dlatego nie możemy go zmienić po nadaniu wartości. Zamiast tego musimy utworzyć nowego stringa.

# ToLower i ToUpper

## ToLower

Zwraca nowego stringa który ma zamienione duże litery na małe.

```
string input = "Ala ma Kota";  
string loweredInput = input.ToLower();  
Console.WriteLine(loweredInput); //ala ma kota
```

## ToUpper

Zwraca nowego stringa, który ma zamienione małe litery na duże.

```
string input = "Ala ma Kota";  
string upperedInput = input.ToUpper();  
Console.WriteLine(upperedInput); //ALA MA KOTA
```



Usuwa białe znaki z początku i końca stringa lub znaki z początku lub końca podane w parametrach metody.

```
string input = "    Ala ma Kota    ";  
string trimmedInput = input.Trim();  
Console.WriteLine(trimmedInput); //Ala ma kota
```



# Substring

Zwraca string będący fragmentem stringa z którego wywołaliśmy metodę. Przyjmuje parametr będący indeksem znaku od którego ma się zacząć fragment oraz opcjonalny parametr definiujący jak długi ma być zwrócony string.

```
string text = "Opowieści z doliny smoków";  
string output1 = text.Substring(9); // " z doliny smoków";  
string output1 = text.Substring(12, 6); // "doliny"
```

Zwraca indeks pierwszego wystąpienia znaku lub stringa podanego w parametrze.



```
string text = "Opowieści z doliny smoków";  
int index = text.IndexOf('z'); // 10  
int dragonindex = text.IndexOf("smok"); // 19
```





# Replace

Zwraca nowy string który ma zastąpione fragmenty stringa podane w pierwszym parametrze nowym fragmentem podanym w drugim parametrze.

```
string title = "Ala ma kota";  
string altTitle = title.Replace("Ala", "Kasia");  
Console.WriteLine(altTitle);  
// Kasia ma kota  
string anotherTitle = title.Replace("kota", "psa");  
Console.WriteLine(anotherTitle);  
// Ala ma psa
```



Zwraca nowy string będący połączeniem stringów podanych w parametrach w kolejności ich występowania.

```
string title = string.Concat("Opowieści", "z", "dołiny", "smoków");  
//Opowieścizdolinysmoków
```

Zwraca nowy string będący połączeniem stringów podanych w parametrach w kolejności ich występowania z występującym znakiem między łączonymi stringami podanym w pierwszym parametrze.



```
string title = string.Join(' ', "Opowieści", "z", "doliny", "smoków");  
//Opowieści z doliny smoków
```

## IsNullOrEmpty

Zwraca wartość bool ustawioną na true jeśli string podany w parametrze jest wartości null lub string.Empty

```
string nothing = string.Empty;  
if(string.IsNullOrEmpty(nothing))  
{  
    Console.WriteLine("Pusto");  
}
```

## IsNullOrWhitespace

Zwraca wartość bool ustawioną na true jeśli string podany w parametrze jest wartości null lub zawiera wyłącznie białe znaki.

```
string nothing = "\t\t\n \t";  
if(string.IsNullOrWhiteSpace(nothing))  
{  
    Console.WriteLine("Pusto");  
}
```



# **Wyjątki, rzutowanie, boxing, unboxing**



Jest to konwertowanie jednego typu danych na inny. Dzieli się na dwa rodzaje – niejawny gdzie dokonujemy konwersji, np. mniejszego na większy typ danych czy konwersji klasy pochodnej na klasę bazową, oraz jawny gdzie należy użyć operatora rzutowania lub predefiniowanej metody do konwertowania typów.



# ToString

Metoda, którą posiada każdy obiekt

- zwraca string będący

reprezentacją wartości obiektu.




```
double value = 420.69;  
string text = value.ToString(); // "420.69"
```



# Rzutowanie

Jest to mechanizm który przekształca większy typ danych na mniejszy.




```
double value = 420.69;  
int valueWithoutDecimals = (int)value; // 420
```



# Boxing Unboxing

Boxing jest to mechanizm który polega na przekształceniu typu wartościowego w typ referencyjny natomiast unboxing polega na przekształceniu tego typu spowrotem w typ wartościowy.



```
double value = 420.69;  
object o = value;  
  
double unboxedValue = (double)o;
```





Wyjątek jest to obiekt który jest odpowiedzią na nieoczekiwane zdarzenie do których dochodzi w trakcie wykonywania programu. W języku C# reprezentowane są przez klasy które pośrednio lub bezpośrednio dziedziczą z klasy System.Exception

Każdy wyjątek składa się z komunikatu błędu oraz tzw. stacktrace czyli miejsca gdzie wyjątek został wyrzucony i jakie metody zostały wykonane aby program doszedł do miejsca.



# Popularne wyjątki

**IOException** – obsługa problemów związanych z odczytywaniem i zapisywaniem plików

**IndexOutOfRangeException** – obsługa wyjątków związanych z tablicami, a ściślej z wyjściem indeksu poza dopuszczalny zakres

**NullReferenceException** – obsługa wyjątków związanych z typem pustym (wywołanie metody z nulla)

**DivideByZeroException** – obsługa wyjątków związanych z dzieleniem przez zero

**InvalidCastException** – obsługa wyjątków związanych z niepoprawnym rzutowaniem

**OutOfMemoryException** – obsługa wyjątków związanych z niewystarczającą ilością pozostałej pamięci

**StackOverflowException** – obsługa wyjątków związanych z przepełnieniem stosu



# Obsługa wyjątków

Wyjątki obsługujemy poprzez bloki try-catch-finally.

W bloku try umieszczamy kod, który może wyrzucić wyjątek natomiast w bloku catch umieszczamy kod, który ma się wykonać w momencie wyrzucenia wyjątku.

W bloku finally umieszczamy kod, który ma się wykonać niezależnie od tego czy wyjątek został wyrzucony czy nie.

```
try
{
    // kod który może wyrzucić wyjątek
}
catch
{
    // kod który ma się wykonać po wystąpieniu wyjątku
}
finally
{
    // kod który ma się wykonać niezależnie
    // czy wystąpił wyjątek czy nie
}
```

# Thank you!

You can find me at  
#SLACK or [sosna321@gmail.com](mailto:sosna321@gmail.com)