

ZPR PWr – Zintegrowany Program Rozwoju Politechniki Wrocławskiej

Aplikacje webowe na platformę .NET

W05 – Język C#: typy, zmienne, instrukcje,
metody itp.

Syllabus

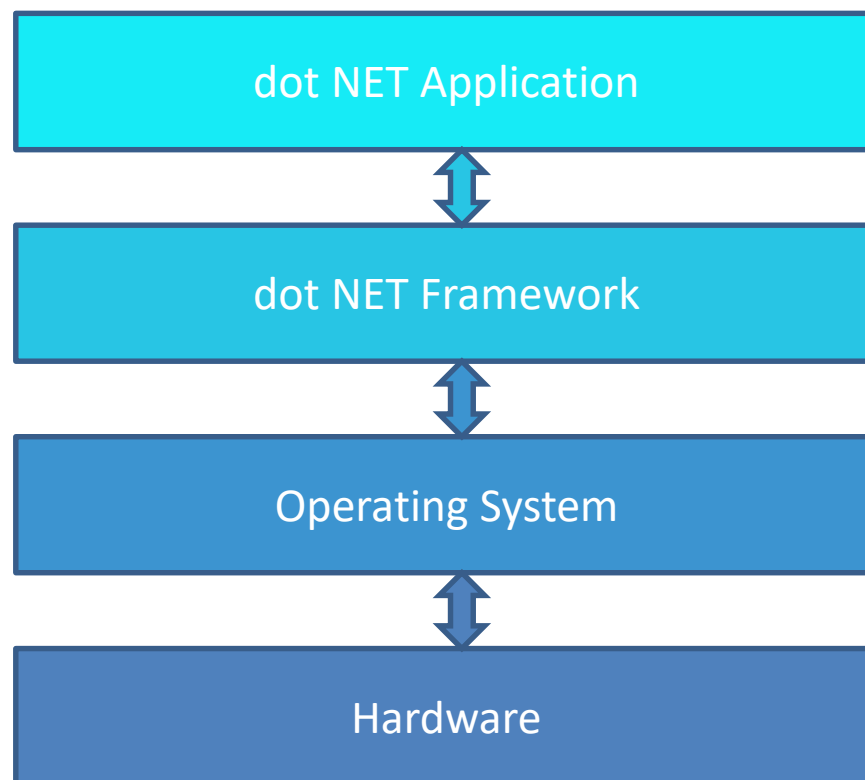
- Frameworki Microsoft
- Komenda `dotnet`
- Visual Studio Community
- Visual Studio Code
- Aplikacja konsolowa – operacje we/wy
- Podstawowe typy danych,
- Literały
- Rzutowanie/konwersja w prostych typach
- Identyfikatory
- Słowa kluczowe
- Konwencje notacyjne
- Zmienne,
- Typ **string**
- Kategorie typów:
 - typy wartościowe
 - referencyjne,
- Typy nullable
- Niejawny typ danych – **var**
- Anonimowe typy danych
- Krotki
- Tablice:
 - Jednowymiarowe
 - Wielowymiarowe
 - Tablice `tablic`
- Metody w `System.Arrays`
- Operatory dwu argumentowe
- Złożone operatory przypisania
- Instrukcje sterujące:
 - **if, if else**
 - **while**
 - **do while**
 - **for**
 - **foreach**
 - **switch**
- Instrukcje **break, continue, goto**
- Operatory dla wyrażeń logicznych
- Operatory `?:, ??, ?.`
- Dyrektywy kompilatora
- Operatory na bitach
- Nagłówek metody
- Parametry przekazywane przez wartość
- Parametry `ref`
- Parametry `out`
- Parametry `in`
- Parametry `params`
- Parametry opcjonalne
- Wywoływanie metody
- Argumenty nazwane
- Przeciążanie metod
- Określanie wywoływanej metody

MICROSOFT FRAMEWORKI

Framework

- **Framework** albo **platforma programistyczna** – szkielet do budowy [aplikacji](#). Definiuje on strukturę aplikacji oraz ogólny mechanizm jej działania, a także dostarcza zestaw [komponentów](#) i [bibliotek](#) ogólnego przeznaczenia do wykonywania określonych zadań. Programista tworzy aplikację, rozbudowując i dostosowując poszczególne komponenty do wymagań realizowanego projektu, tworząc w ten sposób gotową aplikację [Wikipedia].
- Cechy:
 - odwrócenie sterowania
 - w odróżnieniu od aplikacji oraz bibliotek, przepływ sterowania jest narzucany przez framework, a nie przez użytkownika
 - domyślne zachowanie
 - domyślna konfiguracja frameworka musi być użyteczna i dawać sensowny wynik, zamiast być zbiorem pustych operacji do nadpisania przez programistę.
 - rozszerzalność
 - poszczególne komponenty frameworka powinny być rozszerzalne przez programistę, jeśli ten chce rozbudować je o niezbędne mu dodatkowe funkcje.
 - zamknięta struktura wewnętrzna
 - programista może rozbudowywać framework, ale nie poprzez modyfikację domyślnego kodu.

Ogólna idea frameworków Microsoft



- W tym przypadku framework rozumiany jako biblioteka/warstwa pośrednicząca
 - Komunikowanie się tylko między sąsiadującymi warstwami

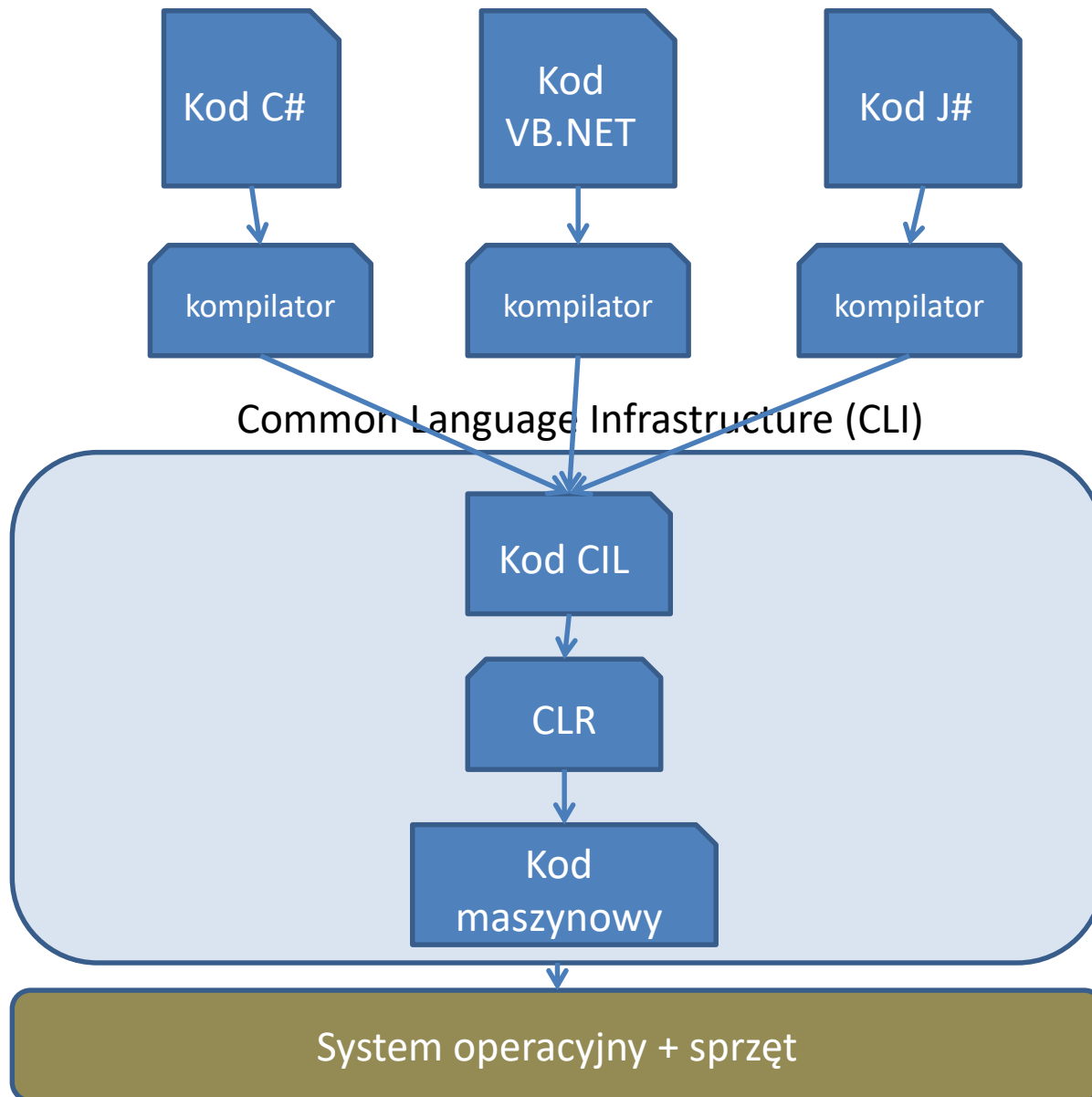
Platformy – historia, teraźniejszość

- **.NET Framework** – pierwszy framework, zamknięty, zorientowany na system operacyjny Windows, duży.
 - strony web, serwisy i aplikacje na Windows
- **.NET Core** – open source [GitHub], otwarty na dowolny system operacyjny, elementy społecznościowe, mniejszy.
 - Windows, Linux, macOS, do wszelakich rodzajów aplikacji
- **.NET** – obecny standard, ma łączyć dwa powyższe, oparty na .NET Core. W praktyce pewne rozwiązania z .NET Framework zostały zastąpione innymi rozwiązaniami. Np. Web Forms zastąpiono ASP.NET Core Blazor lub Razor Pages. Inne, jak Windows Communication Foundation (WCF), są wspierane tylko dla systemu Windows.
- **Xamarin/Mono** – zamknięty, dla systemów mobilnych
- **.NET Standard** – zbiór bibliotek współdzielonych między powyższymi platformami (od pewnych wersji)
- Inne pochodne .NET Framework, np. .NET Compact Framework (dla Windows CE na telefony komórkowe)

Wersje frameworków

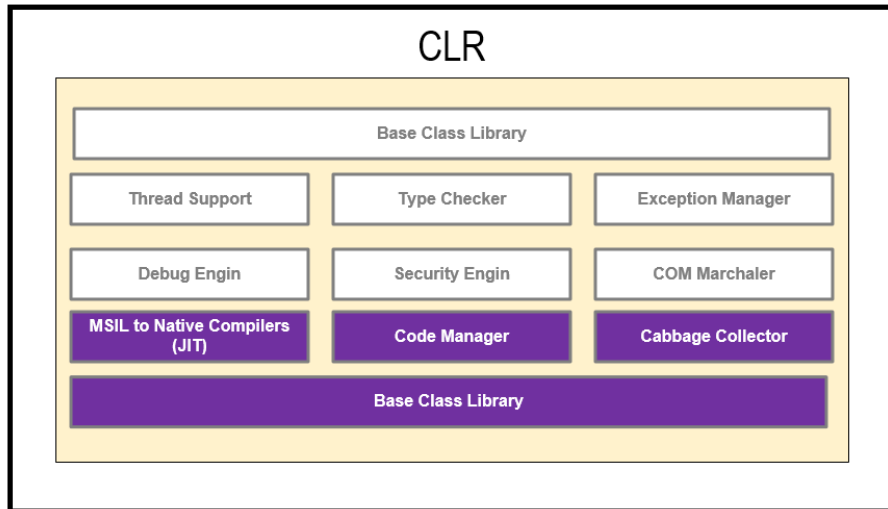
- .NET Framework:
 - najstarszy: 1.0 – 2002-02-13
 - bieżący: 4.8.0 Build 3928 - 2019-07-25
 - przyszły: ? (migracja do .NET)
- .NET Core:
 - najstarszy: 1.0 – 2016-06-27
 - bieżący: 3.1 - 2020-01-15
- .NET 5 – wersja 5.0.17 - 2022-05-10
 - Pominięto w numeracji wartość 4, aby nie mylić z .NET Framework
 - <https://docs.microsoft.com/pl-pl/dotnet/core/dotnet-five>
 - Zakończono wsparcie
- .NET 6 – wersja 6.0.10 - 2022-10-11
- .NET 7 – planowane na 2022-11
- .NET 8 – planowane na 2023-11
- https://en.wikipedia.org/wiki/.NET_Framework
- <https://en.wikipedia.org/wiki/.NET>

Common Language Infrastructure (CLI)

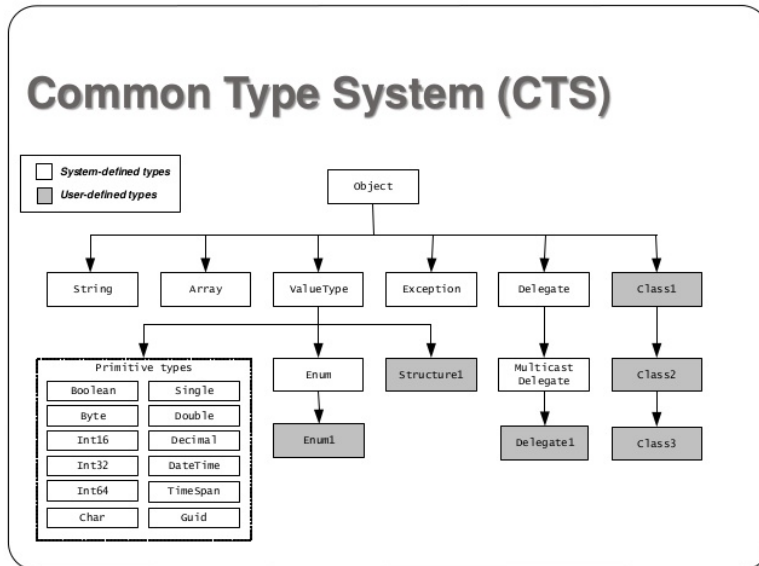


- CIL (**Common Intermediate Language**) – język pośredni do którego kompilowane są kody z języków kompatybilnych z .NET
- CLR – **Common Language Runtime** – kompiluje kod CIL do kodu maszynowego (JIT – **just in time**) i wykonuje go.

Bloki składowe platformy .NET



- **CLR** (*ang. Common Language Runtime*) odpowiedzialny za lokalizowanie, wczytywanie oraz zarządzanie typami .NET. To trzon całej platformy .NET ponieważ to właśnie do CLR należy zadanie kompilowania i uruchamiania kodu zapisanego językiem kodu pośredniego (CIL).
- **CTS** (*ang. Common Type System*) jest odpowiedzialny za opis wszystkich danych udostępnianych przez środowisko uruchomieniowe.
- **CLS** (*ang. Common Language Specification*) to zbiór zasad definiujących podzbiór wspólnych typów precyzujących zgodność kodu binarnego z dostępnymi kompilatorami .NET
- Źródło: <https://www.knowsh.com/Notes/210311/Interview-Questions-On-ASPNET--Terminology-CLR-CTS-CLS-MSIL-Managed-Code>



Języki zgodne z platformą .NET

- Obecnie ponad 40 języków programowania jest zgodnych z .NET. Często znane języki musiały być zmienione, aby dostosować je do wymagań tej platformy, stąd w nazwie dodawana jest końcówka „.NET”. Najbardziej znane, to: [C#](#), [Visual Basic .NET](#), [F#](#), [C++/CLI](#), [J#](#) (wariant języka [Java](#) opracowany przez Microsoft), [JScript .NET](#) (kompilowany wariant języka JScript)

Zalecane środowiska

- Visual Studio 2019 (dla .Net 5)
 - Community
- Visual Studio 2022 (dla .Net 6)
- Visual Studio Code
- Visual Studio for Mac
- Wersja konsolowa - komenda:
 - `dotnet`
Działa w każdym z wcześniej wymienionych systemów operacyjnych.

Przydatne linki

- Microsoft .NET: <https://dotnet.microsoft.com/>
- C# Guide: <https://docs.microsoft.com/en-us/dotnet/csharp/>
- Już historyczne:
 - Film „.NET Framework and .NET Core” krótka charakterystyka różnic między tymi platformami”
<https://www.youtube.com/watch?v=OkeM7XVwEdA>
 - Film nt. „Co to jest .NET Core”
<https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet>

KOMENDA DOTNET

Korzystanie z konsoli systemu operacyjnego

- Instalacja .NET 6.0:
 - <https://dotnet.microsoft.com/download>
- po uruchomieniu konsoli Windows (lub Linux) wydać komendę dotnet

```
Wiersz polecenia

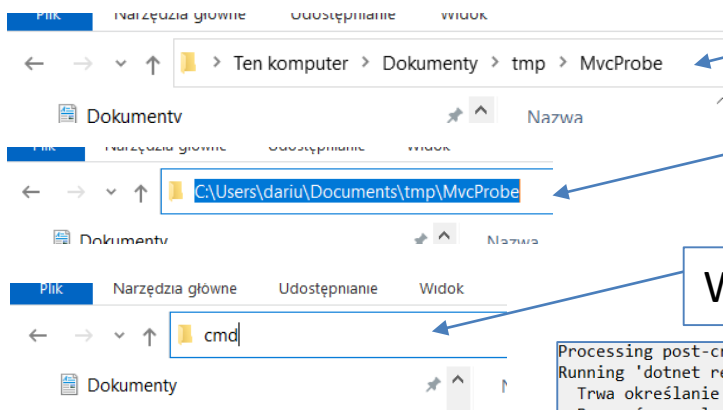
C:\Users\dariu>dotnet

Usage: dotnet [options]
Usage: dotnet [path-to-application]

Options:
  -h|--help          Display help.
  --info             Display .NET information.
  --list-sdks        Display the installed SDKs.
  --list-runtimes    Display the installed runtimes.

path-to-application:
  The path to an application .dll file to execute.

C:\Users\dariu>dotnet --version
5.0.208
```



Za ścieżką kliknąć myszką

Wpisać „cmd”

Wcisnąć [Enter]

```
Processing post-creation actions...
Running 'dotnet restore' on C:\Users\dariu\Documents\tmp\MvcProbe\
Trwa określanie projektów do przywrócenia...
Przywrócono element C:\Users\dariu\Documents\tmp\MvcProbe\MvcPro
Restore succeeded.

C:\Users\dariu\Documents\tmp\MvcProbe>dir

Volume in drive C is Windows
Volume Serial Number is 8291-76D4

Directory of C:\Users\dariu\Documents\tmp\MvcProbe

2021-11-01 12:17 <DIR>      .
2021-11-01 12:17 <DIR>      ..
2021-11-01 12:17          162 appsettings.Development.json
2021-11-01 12:17          192 appsettings.json
2021-11-01 12:17 <DIR>      Controllers
2021-11-01 12:17 <DIR>      Models
2021-11-01 12:17          141 MvcProbe.csproj
2021-11-01 12:17 <DIR>      obj
2021-11-01 12:17          716 Program.cs
2021-11-01 12:17 <DIR>      Properties
2021-11-01 12:17 <DIR>      1 827 Startup.cs
2021-11-01 12:17 <DIR>      Views
2021-11-01 12:17 <DIR>      wwwroot
2021-11-01 12:17          5 File(s)      3 038 bytes
                8 Dir(s)  22 719 356 928 bytes free
```

- Pozwala wykonywać działania za pomocą komend np.:
 - dotnet new mvc
- Stworzenie nowego szkieletu projektu ASP.NET MVC Core

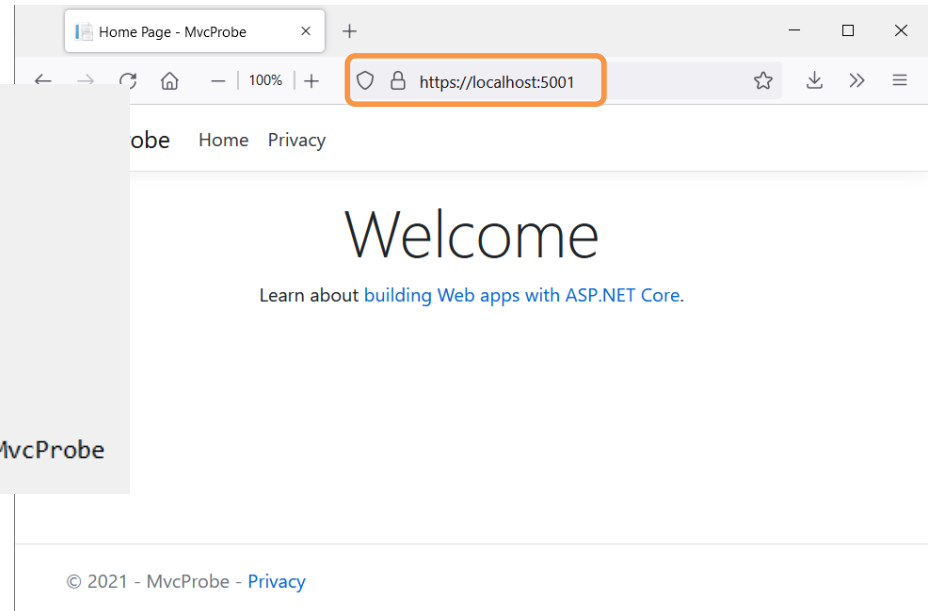
Pełna dokumentacja komend dotnet:

<https://docs.microsoft.com/pl-pl/dotnet/core/tools/>

dotnet build

- dotnet build
 - Skompilowanie projektu
- dotnet run
 - Uruchomienie projektu
- Po tym ciągu komend można w przeglądarce otworzyć stronę wskazaną podczas uruchamiania:
 - <https://localhost:5001/>
- [Ctrl]+C w konsoli wyłącza serwer

```
C:\Users\dariu\Documents\tmp\MvcProbe>dotnet run
Trwa kompilowanie...
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Users\dariu\Documents\tmp\MvcProbe
```



Rozszerzalność dotnet

- Komenda `dotnet` jest rozszerzalna.
- Np. po dodaniu pakietu EF poprzez:
 - Globalnie przez: `dotnet tool install --global dotnet-ef`
 - Lub lokalnie przez:
`dotnet add package Microsoft.EntityFrameworkCore.Design`
- Można używać komend do migracji baz danych:
 - `dotnet ef migrations add <name>`
 - `dotnet ef database update <name>`

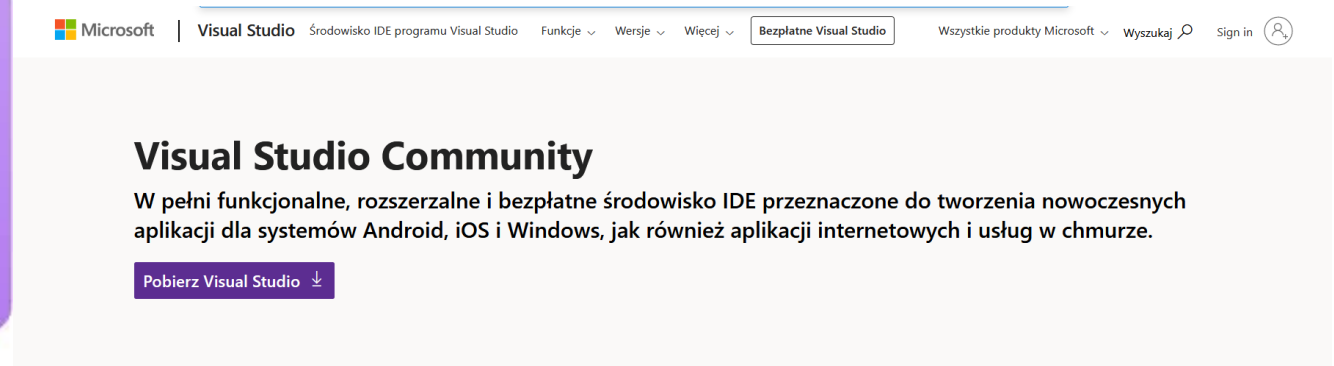
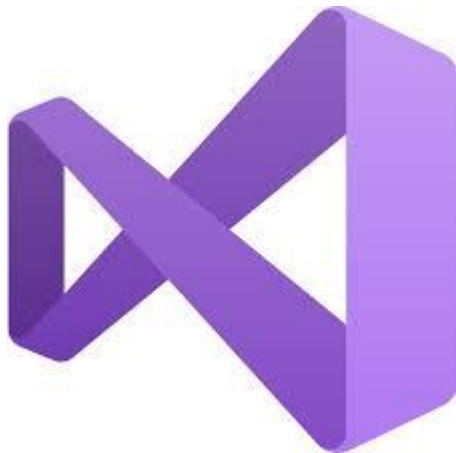
`dotnet` a konsola pakietów NuGet

- W ramach Visual Studio 2019/2022 można włączyć konsolę pakietów NuGet, w której komendy są podobne jak dla `dotnet`, ale oczywiście bez „`dotnet`”.
- Używanie komend `dotnet` pozwala na tworzenie własnych środowisk deweloperskich, w różnych środowiskach operacyjnych, w różnych językach programowania.
- Można „łatwo” dodać programowanie w .Net do istniejących otwartych platform deweloperskich.
 - Przykładem Visual Studio Code

VISUAL STUDIO 2019 COMMUNITY

Visual Studio 2019 Community

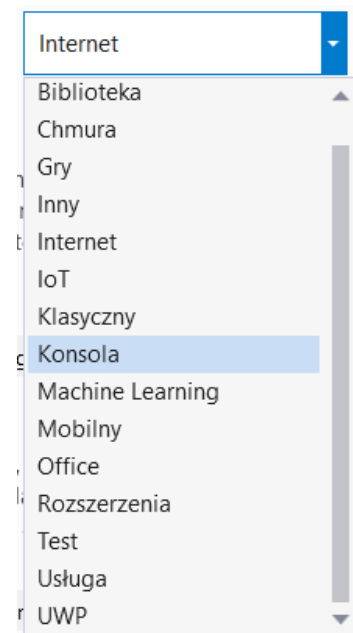
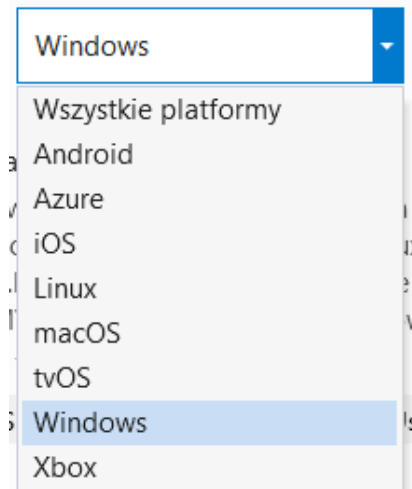
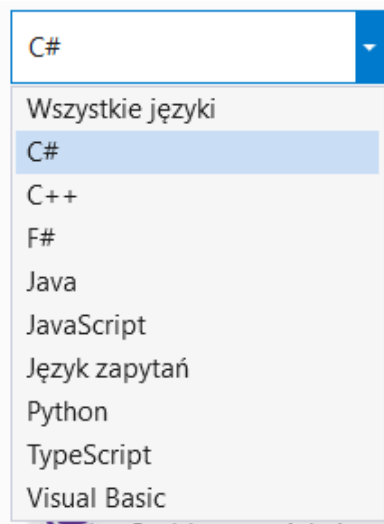
- <https://visualstudio.microsoft.com/pl/vs/community/>
- Darmowe do własnego wykorzystania
- Istnieją też płatne wersje:
 - Professional
 - Enterprise
- 8 listopada 2021r - premiera VS 2022...
 - Na ePortalu dokument „Dodanie .Net 5 do Visual Studio 2022 Community”



Wszystko, czego potrzebujesz w jednym miejscu

Tworzenie projektu

- Podczas tworzenia projektu/rozwiązania w trakcie wybierania szablonu wybieramy konkretnie:
 - Język programowania
 - Platformę (system operacyjny)
 - Typ projektu
- Dużo kombinacji tych trzech elementów, w dodatku po wybraniu składowych nadal jest do wyboru wiele rodzajów projektów
- Domyślnie jest tworzony projekt z jednym rozwiązaniem. W rozwiązaniu można mieć wiele projektów, np. każdy wytworzony innym szablonem projektu
- Szablon dołącza odpowiednie biblioteki, zespoły, powiązania, tworzy początkowe pliki z kodem i innymi danymi (np. konfiguracją)
- Na tym wykładzie będzie używany głównie typ Konsola i Internet
- Zrzuty z ekranu są raz z ustawieniem języka polskiego raz z ustawieniem języka angielskiego



Projekt konsolowy

- W wersji .NET Core

Aplikacja konsoli (.NET Core)

Projekt służący do tworzenia aplikacji wiersza poleceń, którą można uruchomić w środowisku .NET Core w systemach Windows, Linux i MacOS.

C# Linux macOS Windows Konsola

Program.cs

```
1 using System;
2
3 namespace ConsoleEmpty
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             Console.WriteLine("Hello World!");
10        }
11    }
12 }
13
```

Eksplorator rozwiązań

Przeszukaj: Eksplorator rozwiązań (Ctrl+;)

Rozwiązanie „ConsoleEmpty” (liczba projektów)

- C# ConsoleEmpty
 - Zależności
 - Struktury
 - Microsoft.NETCore.App
 - C# Program.cs
 - Program
 - Main(string[]) : void

Lista błędów

Cale rozwiązanie 0 Błędy 0 Ostrzeżenia 1 Komunikat Kompilacja + IntelliSense Przeszukaj listę błędów

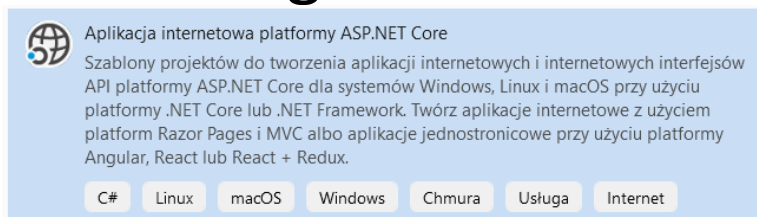
Kod	Opis	Projekt	Plik	W...	Stan...
IDE0060	Usunąć nieużywany parametr „args”	ConsoleEmpty	Program.cs	7	Aktywne

Konsola debugowania

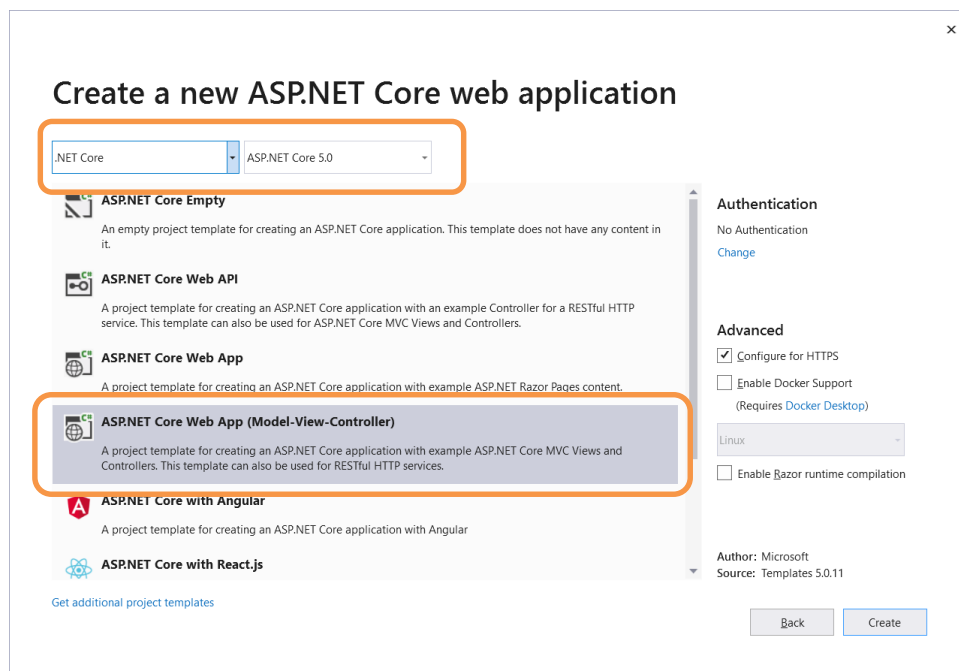
```
Hello World!
C:\Users\daniu\source\repos\ConsoleEmpty\ConsoleEmpty>
z kodem 0.
```

Aplikacja internetowa 1/2

- Również głównie w ASP .NET Core

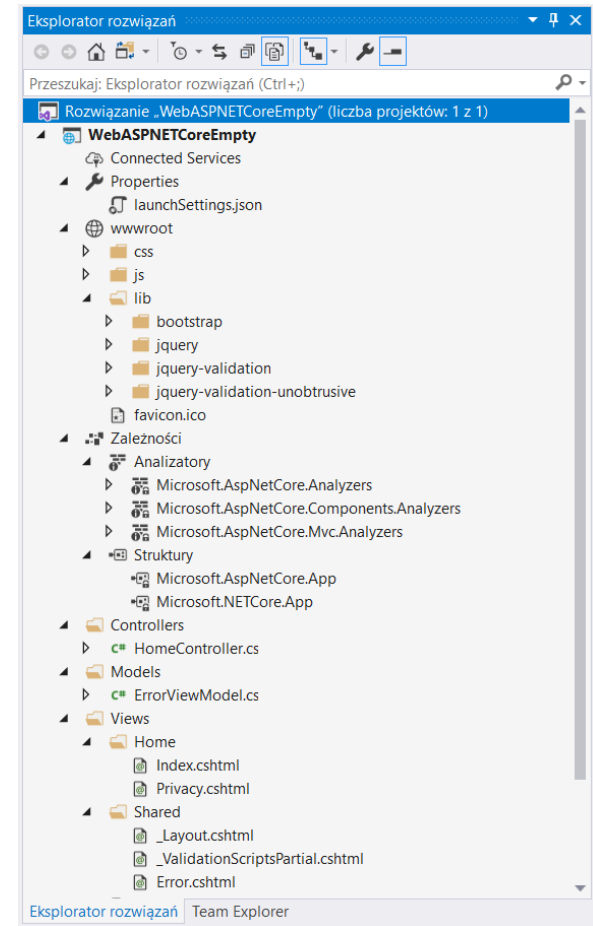
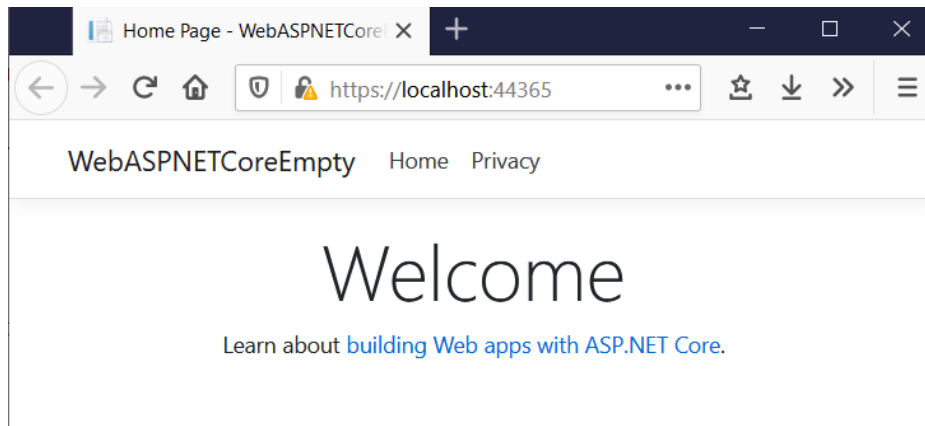
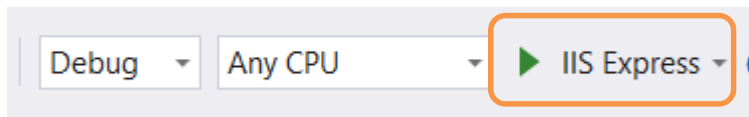


- Po nadaniu nazwy następuje uszczegółowienie szablonu, który ma być użyty, dla przykładu użyjmy „Aplikacja internetowa (Model-View-Cotroller)”
- Można też zmienić rodzaj frameworku i jego wersję



Aplikacja internetowa 2/2

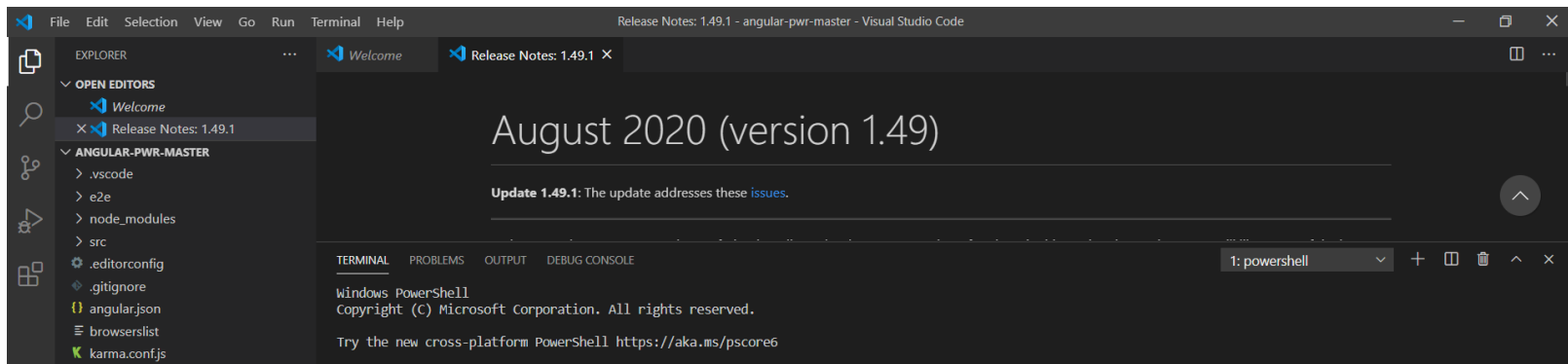
- Struktura dużo bardziej rozbudowana.
- Uruchomienie „IIS Express” otworzy w przeglądarce stronę z projektowaną stroną internetową.



VISUAL STUDIO CODE

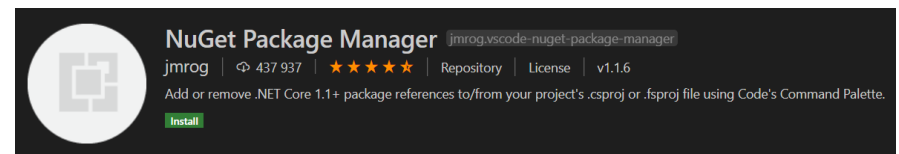
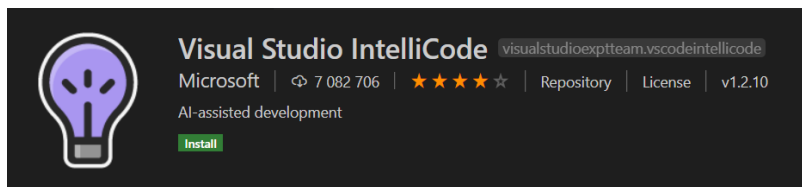
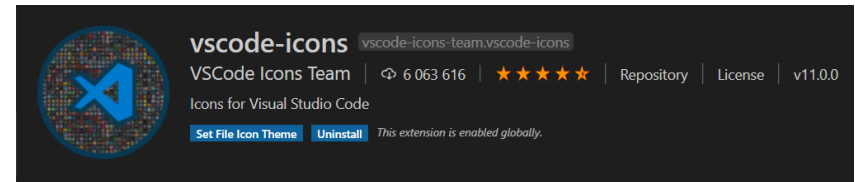
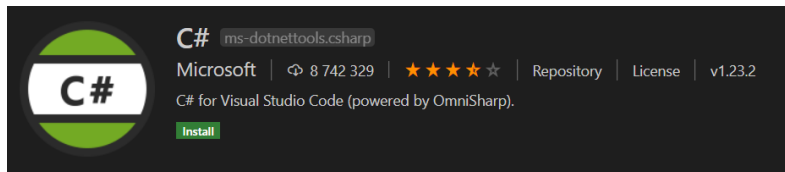
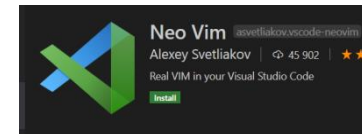
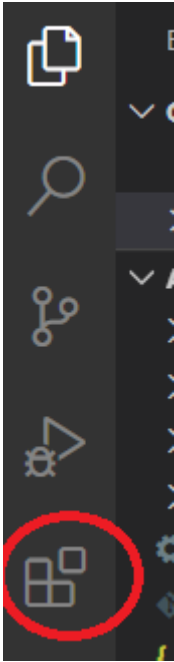
Visual Studio Code

- Charakterystyka Visual Studio Code:
 - Darmowy
 - Desktopowy
 - Otwarty kod
 - Gebuggowanie
 - Zarządzanie wersja przez Git
 - IntelliSense
 - Zarządzanie wycinkami kodu (snippet)
 - Refaktoryzacja Kodu
 - Duże repozytorium rozszerzeń
 - Animacje pokazujące działanie rozszerzeń
 - Wiele operacji wykonuje się za pomocą linii komend



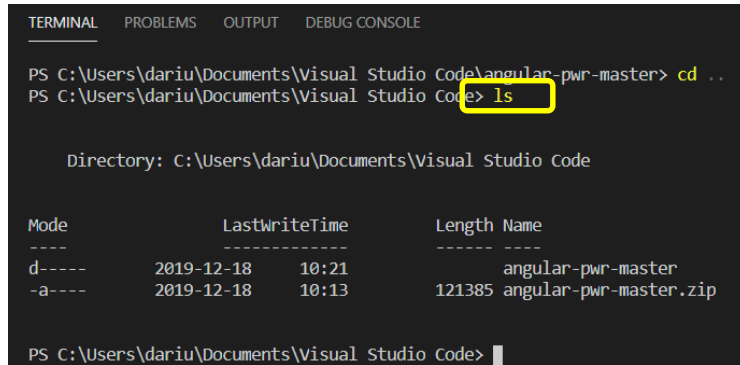
VS Code - extensions

- Z ważnych dla programowania w C#:
 - C#
 - vscode-icons
 - Visual Studio IntelliCode
 - NuGet Package Manager
- Inne ciekawe:
 - Azure Repos
 - Live Share – współdzielenie ekranu
 - GitLens — Git supercharged



VS Code – Terminal 1/2

- Ciekawy link „Intro to VSCode for C# Developers - From Installation to Debugging”:
- <https://www.youtube.com/watch?v=r5dt19Uq9V0>



The screenshot shows a VS Code terminal window with the following content:

```
TERMINAL  PROBLEMS  OUTPUT  DEBUG CONSOLE

PS C:\Users\dariu\Documents\Visual Studio Code\angular-pwr-master> cd ..
PS C:\Users\dariu\Documents\Visual Studio Code> ls

Directory: C:\Users\dariu\Documents\Visual Studio Code

Mode                LastWriteTime         Length Name
----                -
d-----          2019-12-18 10:21             angular-pwr-master
-a----          2019-12-18 10:13          121385 angular-pwr-master.zip

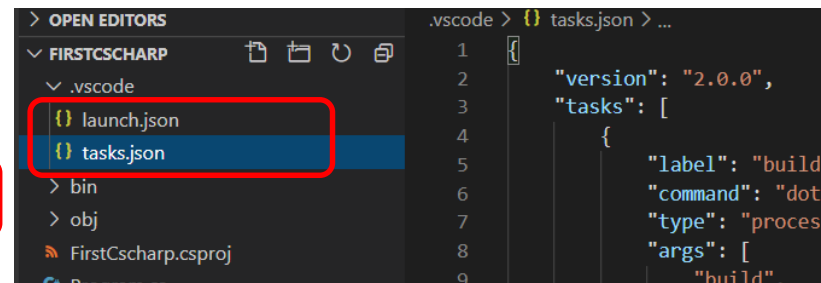
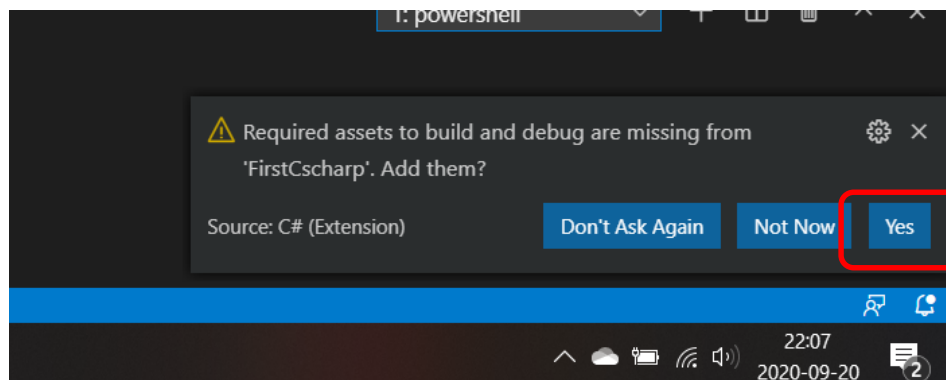
PS C:\Users\dariu\Documents\Visual Studio Code>
```

- Terminal = cmd, czyli należy korzystać z komendy dotnet

```
mkdir FirstCSharp
cd FirstCSharp
dotnet new sln -n "FirstCsharpSln"
dotnet new console -n "FirstCscharp,"
111,53 ms (C:\Users\dariu\Documents\Visual Studio Code\FirstCsharp\FirstCscharp\FirstCscharp.csproj).
```

```
dotnet sln FirstCsharpSln.sln add .\FirstCscharp\FirstCscharp.csproj
cd .\FirstCscharp\
code .
```

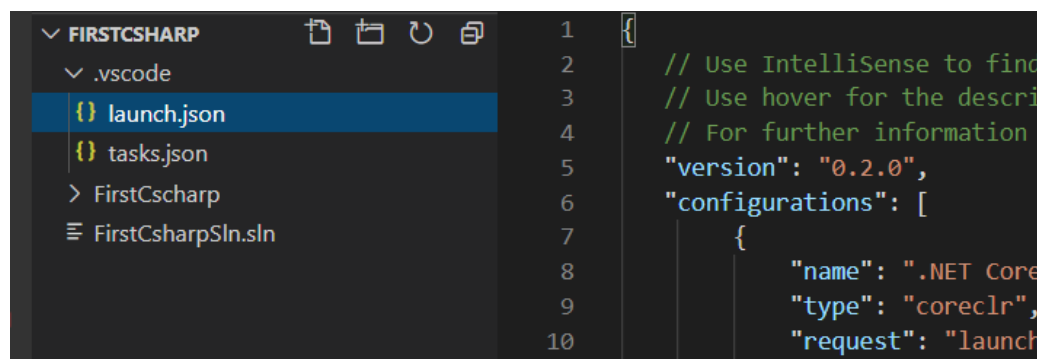
VS Code – Terminal 2/2



- Okienko propozycji pojawi się po chwili
 - Dodanie uruchamiania itd. dla projektu
- To samo na poziomie solucji (zamknąć VS Code)

cd ..

code .



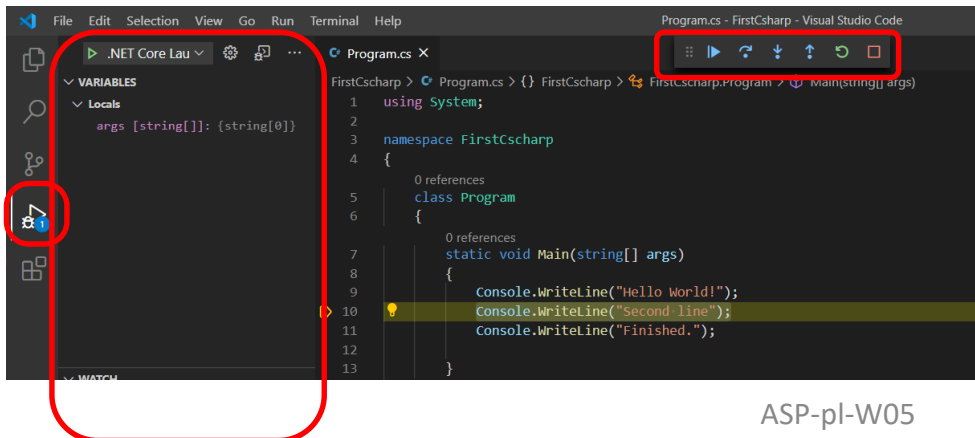
Działanie VS Code dla C#

- Działają skróty klawiszowe, ale trochę inne niż w MS VS Community
 - Można zainstalować inne, np. z Notepad++
- Działa rozwijanie skrótów typu „cw”, „prop”
- Uruchamianie programu w terminalu:

```
cd FirstCsharp
dotnet run
```

```
PS C:\Users\dariu\Documents\Visual Studio Code\FirstCsharp> cd .\FirstCsharp\
PS C:\Users\dariu\Documents\Visual Studio Code\FirstCsharp\FirstCsharp> dotnet run
Hello World!
```

- Można uruchamiać w trybie debuggowania (F5)
 - np. wcześniej w kolumnie przed numeracją linii ustawić breakpoint
 - Pojawi się na środku menu do debuggowania
 - Po lewej można włączyć okienka do przeglądania zmiennych itp.
- Z konsoli można dodawać pakiety Nuget: `dotnet add package <NazwaPakietu>`



JĘZYK C# - PODSTAWY

Uwagi ogólne o języku C#



- Wiele wersji języka z kompatybilności wstecz
 - W niektórych przypadkach na wykładzie będzie się pojawiać informacja, od której wersji języka wprowadzony został omawiany element języka.
- Obecnie funkcjonuje wersja 10.0 z dokumentacją (obsługiwany jest w ramach .NET 6 i instalowana w ramach VS 2022)
- Co rok/dwa lata ma powstawać kolejna wersja języka (w nieskończoność?)
- Wielkość liter ma znaczenie w identyfikatorach!
- Język C# to połączenie idei języków Java i C++
 - Usunięcie „złych” cech, zostawienie „dobrych” plus nowe pomysły.
- Nazwy plików nie muszą odpowiadać nazwom klas.
 - Oczywiście wskazane aby w wielu przypadkach tak jednak było.
- W pliku mogą być definicje więcej niż jednej klasy
 - Nie trzeba tworzyć tyle plików ile będzie klas
- W przypadku klas **częściowych** kod może być rozbity na więcej plików.

C# - pierwszy program konsolowy



- Metodą od której aplikacja zaczyna działanie jest: **static void** Main(**string**[] args)
- Metoda musi być w jakiejś klasie np. Program.
- Klasa **musi być w jakiejś przestrzeni nazw** np. ConsoleFirst

```
namespace ConsoleFirst
{
    class Program
    {
        static void Main(string[] args)
        {
            // Main method code
        }
    }
}
```




- Odpowiednie klasy i obiekty do operowania na konsoli znajdują się w przestrzeni `System`.
- Należy zatem włączyć ją do aplikacji za pomocą poniższej komendy na początku pliku źródłowego `using System;`
- Klasa do obsługi konsoli: `System.Console`
- Metody w tej klasie (statyczne) do wypisywania do strumienia powiązanego z konsolą: `Write`, `WriteLine`.
- Pierwsza powoduje, że po wypisaniu tekstu, będącego argumentem, kursor pozostaje za ostatnim znakiem, druga – kursor przechodzi do następnej linii.

Korzystanie z konsoli - przykład



```
using System;

namespace ConsoleFirst
{
    class Program
    {
        static void Main(string[] args)
        {
            System.Console.WriteLine("First line, full path");
            Console.WriteLine("second line");
            Console.Write("polish letters - zażółć");
            Console.WriteLine(" jaźń");
        }
    }
}
```

 Konsola debugowania programu Microsoft

```
First line, full path
second line
polish letters - zażółć jaźń
C:\Users\dariu\source\repos\Console
```



- W celach prezentacji innych elementów wykorzystane będą bezparametryczne metody statyczne bez zwracanego wyniku postaci:
public static void *NazwaMetody*()
- Metody te będą wywoływane w metodzie `Main`. Zapisywane będą w ramach przykładowej klasy (np. `Program`).
- Więcej o tworzeniu metod, parametrach itd. będzie na kolejnych slajdach i wykładach.
- Na kolejnych slajdach prezentowane będą tylko kody tych metod z założeniem, że zostają wywołane w metodzie `Main`, tak jak na przykładzie na następnym slajdzie.

Rozbicie na metody - przykład



```
using System;

namespace ConsoleFirst
{
    class Program
    {
        public static void Test1()
        {
            System.Console.WriteLine("First line, full path");
            Console.WriteLine("second line");
        }

        public static void TestConcat()
        {
            Console.Write("polish letters - zażółć");
            Console.WriteLine(" jaźń");
        }

        static void Main(string[] args)
        {
            Test1();
            TestConcat();
        }
    }
}
```

```
First line, full path
second line
polish letters - zażółć jaźń
```

Deklarowanie zmiennych lokalnych - krótko

C# == Java

- W ramach kodu metody.
- Składnia:
typZmiennej nazwaZmiennej;
- Można również w tej samej linii zainicjować zmienną:
typZmiennej nazwaZmiennej=wartość;
- Nazwa zmiennej to poprawny identyfikator.
- Przykładowe typy: `int`, `double`, `char`, `bool`,
`string`.

Wczytywanie danych z konsoli

- Metoda `ReadLine` wczytuje ze standardowego strumienia dane aż do wciśnięcia klawisza `<Enter>` i zwraca w wyniku ciąg znaków w postaci obiektu typu **string**.
- Istnieją inne metody, które nie będą na wykładzie używane np. `Read`, `ReadKey`.

```
public static void TestReadLine()  
{  
    string firstStr;  
    string secondStr;  
    Console.Write("Input first string:");  
    firstStr = Console.ReadLine();  
    Console.WriteLine("Input second string:");  
    secondStr = Console.ReadLine();  
}
```

```
Input first string:any string  
Input second string:  
This is a new line
```

Formatowanie złożone

- Do wersji C# 6.0 w celu wypisania danych dostępne jest **formatowanie złożone**. Polega na oznaczaniu w wypisywanym ciągu znaków miejsca na wartości, które są kolejnymi argumentami funkcji `WriteLine` numerowanymi od zera. Argumentów tych można użyć wiele razy.

```
public static void TestCompositeFormatting()
{
    string firstStr="a cat";
    string secondStr="Alice";
    Console.WriteLine("{1} has {0}. Again {1} has {0}.", firstStr, secondStr);
}
```

```
Alice has a cat. Again Alice has a cat.
```

<https://docs.microsoft.com/pl-pl/dotnet/standard/base-types/composite-formatting>

Interpolacja łańcucha znaków

- Od wersji C# 6.0 wprowadzono interpolację łańcucha znaków.
- Łańcuch znaków musi być wtedy poprzedzony znakiem dolara \$.
- Zamiast numerów w klamrach, wpisuje się nazwy zmiennych.
 - Dokładniej: poprawne wyrażenie zwracające wynik, który można wypisać.

```
public static void TestStringInterpolation()
{
    string firstStr = "a cat";
    string secondStr = "Alice";
    int value = 4;
    Console.WriteLine($"{secondStr} has {firstStr}");
    Console.WriteLine($"value = {value*4+value/4}");
}
```

```
Alice has a cat
value = 17
```

<https://docs.microsoft.com/pl-pl/dotnet/csharp/tutorials/string-interpolation>

Komentarze

C# == Java

- Komentarze nie ulegają kompilacji do kodu wykonywalnego
- Komentarze XML-owe mogą być kompilowane przez różne narzędzia celem wytworzenia dokumentacji.
 - Może to być wykonywane nawet w trybie online.
- Zwykły komentarz jednowierszowy: od dwóch ukośników „/ /” do końca linii.
- Zwykły blok komentarza: między „/ *” a „* /”
- XML-owy komentarz jednowierszowy: od trzech ukośników „/ / /” do końca linii.
- XML-owy blok komentarza: między „/ * *” a „* * /”

Komentarze - przykład

C# == Java

```
/// <summary>
/// Comment Test - only for presentation
/// </summary>
Odwołania: 0
public static void TestComment()
{
    string firstStr = "a cat"; // first test variable
    string secondStr = /* inner comment */ "Alice";
    /* comment block can
       have many
       lines */

    /* comment block generated
     * by VS adds
     * stars on lines' begin */
    int value = 4;
    Console.WriteLine($"{secondStr} has {firstStr}");
    Console.WriteLine($"value = {value * 4 + value / 4}");
    TestCommentXML();
}
```

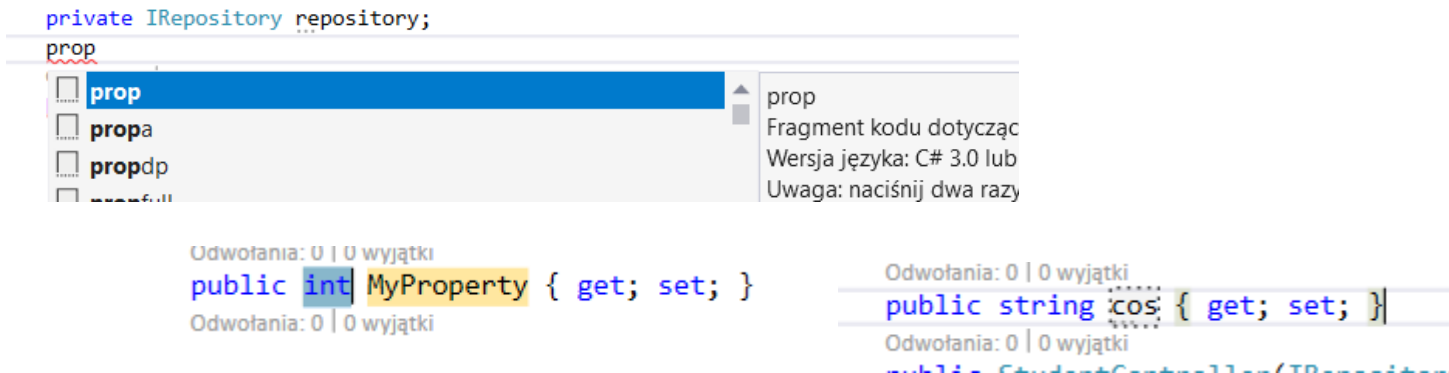
 void Program.TestCommentXML()

Empty method. This is only for presentation

```
/** |
 * <summary> <b>Empty method</b>.
 * This is only for presentation
 * </summary>
 **/
1 odwołanie
public static void TestCommentXML()
{
}
}
```

IntelliSense

- Forma automatycznego uzupełniania zawartego w [Microsoft Visual Studio](#) oraz [Visual Studio Code](#). Jednocześnie służy jako dokumentacja i ujednolniczenie dla nazw [zmiennych](#), [funkcji](#) i [metod](#).
- Stara się podpowiadać najlepsze możliwe dokończenie tekstu z kontekstu, z ostatnich wyborów itd.
- Bardzo dobrze będzie się sprawdzać w ramach ASP .Net
- *Code snippets*: Posiada kilka przydatnych skrótów po których naciśnięcie dwukrotnie <TAB> powoduje rozwinięcie ich w dłuższy fragment kodu
 - np. „prop”<TAB><TAB> tworzy
public int MyProperty { **get; set;** }
kursor ustawia na zaznaczony napis „**int**”, po którego zmianie wciśnięcie <TAB> przeskakuje na zaznaczony napis „MyProperty”. Zmiana nazwy właściwości i naciśnięcie <Enter> kursor znajdzie się za klamrą zamykającą.
 - „cw”<TAB> dla „System.Console.WriteLine”



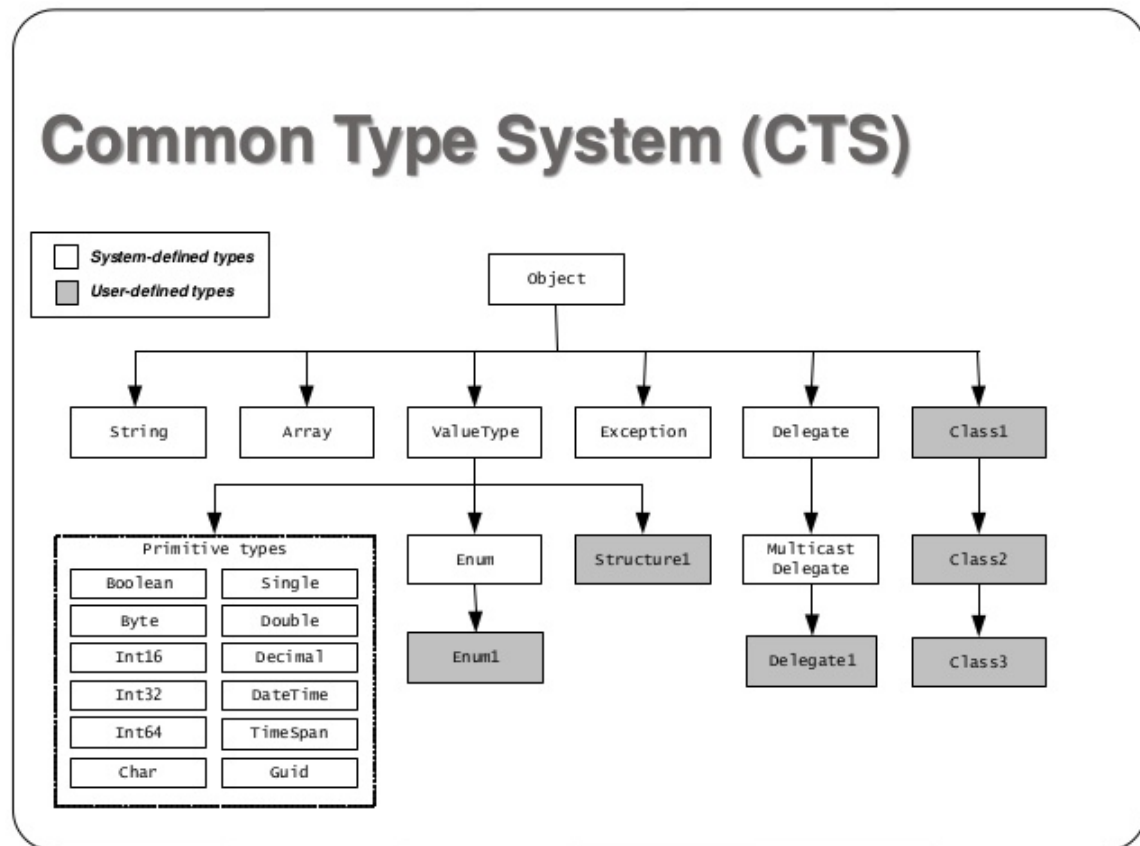
<https://docs.microsoft.com/pl-pl/visualstudio/ide/visual-csharp-code-snippets?view=vs-2019>

C#

TYPY DANYCH, LITERAŁY ITP.

Podstawowe typy danych

- Podstawowe typy:
 - liczby całkowite
 - liczby zmiennoprzecinkowe
 - znak
 - logiczny



Typy całkowitoliczbowe

Typ C#	Typ .NET	Liczba bitów	Zakres	Wartość domyślna
sbyte	<code>System.SByte</code>	8	$[-2^7, 2^7-1]$	0
short	<code>System.Int16</code>	16	$[-2^{15}, 2^{15}-1]$	0
int	<code>System.Int32</code>	32	$[-2^{31}, 2^{31}-1]$	0
long	<code>System.Int64</code>	64	$[-2^{63}, 2^{63}-1]$	0L
byte	<code>System.Byte</code>	8	$[0, 2^8-1]$	0
ushort	<code>System.UInt16</code>	16	$[0, 2^{16}-1]$	0
uint	<code>System.UInt32</code>	32	$[0, 2^{32}-1]$	0
ulong	<code>System.UInt64</code>	64	$[0, 2^{64}-1]$	0L

Typy zmiennoprzecinkowe

Binarne zmiennoprzecinkowe

Typ C#	Typ .NET	Liczba bitów	Zakres	Cyfry znaczące	Wartość domyślna
float	System.Single	32	eps.: $1.4 \cdot 10^{-45}$ zakres: $\pm 3.43 \cdot 10^{38}$	7	0.0F
double	System.Double	64	eps.: $5.0 \cdot 10^{-324}$ zakres: $\pm 1.7 \cdot 10^{308}$	15/16	0.0D

Dziesiętne zmiennoprzecinkowe

Typ C#	Typ .NET	Liczba bitów	Zakres	Cyfry znaczące	Wartość domyślna
decimal	System.Decimal	128	eps.: $1.0 \cdot 10^{-28}$ zakres: $\pm 7.9 \cdot 10^{28}$	28/29	0.0M

Typ logiczny i znakowy

Logiczny

Typ C#	Typ .NET	Liczba bitów	Wartości	Wartość domyślna
bool	<code>System.Boolean</code>	8	true, false	false

Znakowy (znaki Unicode)

Typ C#	Typ .NET	Liczba bitów	Wartości	Wartość domyślna
char	<code>System.Char</code>	16	Kody Unicode (od 0 do 65535)	<code>'\0'</code>

Typ **char** reprezentuje jednostkę kodową UTF-16

Typy dla daty, czasu i przedziału czasu

Typ C#	Typ .NET	Opis
DateTime	<code>System.DateTime</code>	Reprezentacja daty i czasu od 0:00:00am 1/1/01 Do 11:59:59pm 12/31/9999
TimeSpan	<code>System.TimeSpan</code>	Łańcuch znaków Unicode

Predefiniowane typy referencyjne

Typ C#	Typ .NET	Opis
object	<code>System.Object</code>	Korzeń drzewa typów w CTS. Wszystkie inne typy w CTS (również wartościowe) są wyprowadzone z object
string	<code>System.String</code>	Łańcuch znaków Unicode

Typ **string** reprezentuje sekwencję jednostek kodowych UTF-16

Literały dla liczb

- Literał to zapis stałej wartości danego typu.
- Literały dla liczb:
 - **sbyte, short, int, byte, ushort, uint** – ciąg cyfr
 - **long, ulong** – ciąg zakończony literą 'L' lub 'l'
 - **double** – zapis liczby rzeczywistej (może być z kropką i ewentualnie wykładnikiem po literze 'E' lub 'e'), bez litery na końcu lub z literą 'D' lub 'd'
 - **float** – podobnie jak double, ale na końcu litera 'F' lub 'f'
 - **decimal** – podobnie jak double, ale na końcu litera 'M' lub 'm'
- Dla liczb całkowitych w różnych podstawach – poprzez przedrostki:
 - Bez przedrostku – zapis dziesiętny
 - „0b” – binarny
 - „0x” – szesnastkowy
- Dla przejrzystości zapisu istnieje (od C# 7) separator cyfr – znak podkreślenia '_', którym można oddzielać grupy cyfr
- Podczas zamiany liczb na **string** istnieją modyfikatory pozwalające w analogiczny sposób prezentować liczbę w wybrany sposób.

C# == Java



Literały dla liczb - przykłady

```
static void DoubleVsDecimal()
{
    double x = 0.1;
    double xx = x + x + x;
    Console.WriteLine("{0:R}", xx); // 0,3000000000000000004

    decimal y = 0.1M;
    decimal yy = y + y + y;
    Console.WriteLine("{0}", yy); // 0,3
}
```

```
0,300000000000000004
0,3
```

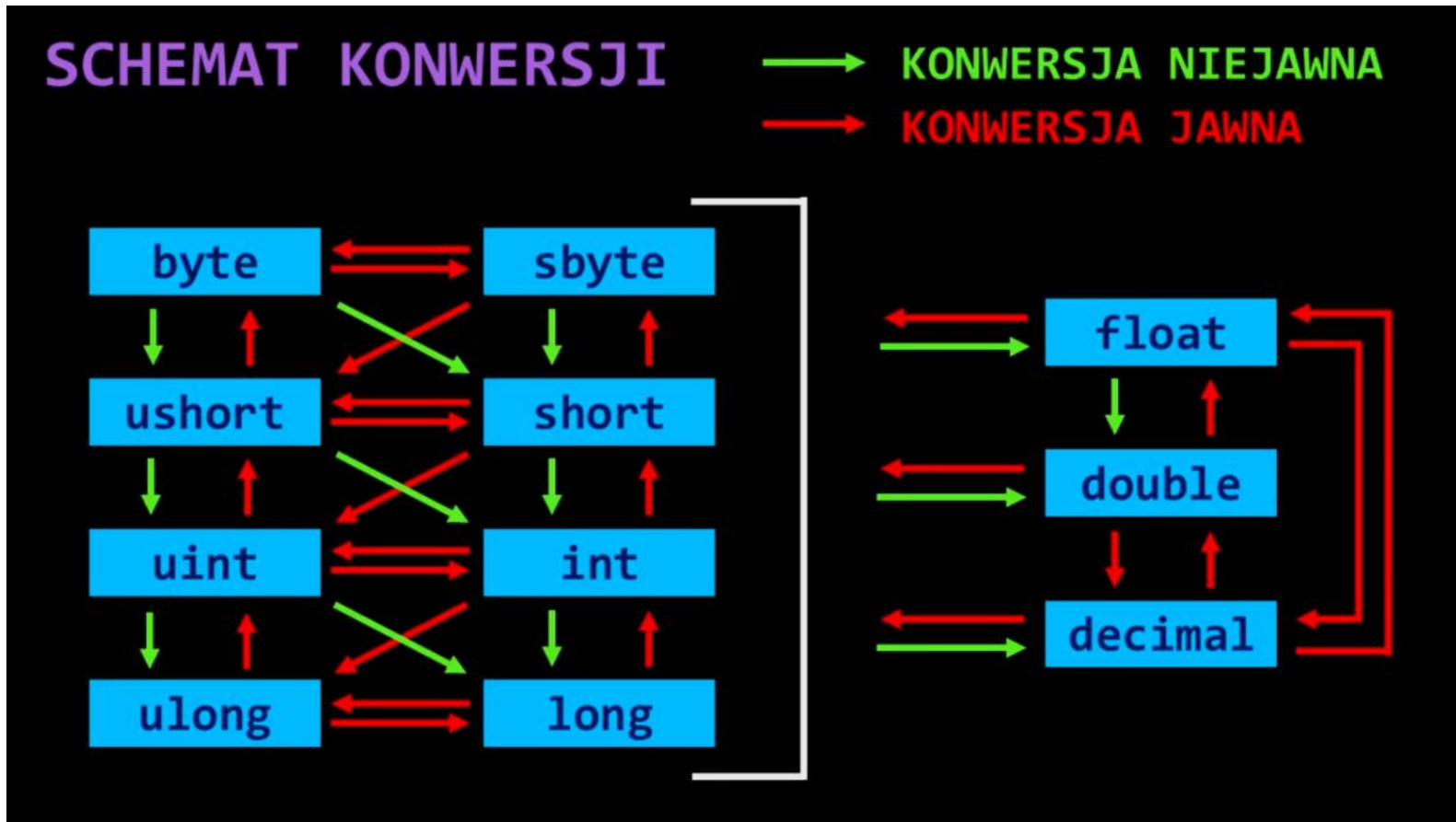
```
static void Literals()
{
    int x = 0b01010;
    long a = 1_234_567_890; // separators using
    float b = 1.0f; // without 'f' compilation error
    float c = 123.45e10f;
    decimal d = 123456789;
    System.Console.WriteLine($"d = {a}");
    System.Console.WriteLine($"d = {a:X}");
}
```

```
d = 1234567890
d = 499602D2
```

Rzutowanie

C# == Java

- Jeśli wartość wyrażenia liczbowego jest typu „węższego” niż zmienna, pod którą tą wartość podstawiamy, to następuje niejawne rzutowanie na „szerszy” typ.
- Jeśli wartość wyrażenia liczbowego jest typu „szerszego” niż zmienna, pod którą tą wartość podstawiamy, to należy jawnie rzutować na węższy typ.



Źródło: <https://www.youtube.com/watch?v=ax87ZsyoaRI>

Konwersje - przykład

- Uwaga na przekraczanie zakresów!

C# == Java

```
static void Castings()  
{  
    int a = 1234;  
    uint b = (uint) a;  
    long c = a;  
    c = b;  
    float x = c;  
    System.Console.WriteLine("x="+x);  
    double d = x;  
    d = 20e100;  
    x = (float)d;  
    System.Console.WriteLine("d="+d);  
    System.Console.WriteLine("x="+x);  
}
```

x=1234
d=2E+101
x=?



- Skończona sekwencja zero lub więcej znaków to łańcuch znaków (ang. string).
- Literały tekstowe w C# można zapisać na kilka sposobów:
 - znaki ujęte w dwa cudzysłowy, gdzie lewy ukośnik interpretowany jest jako „sekwencja ucieczki”
 - Jako **dosłowny literał tekstowy**, czyli podobnie jak poprzednio, ale poprzedzone znakiem '@'. Jedynie dwa cudzysłowy jeden po drugim są interpretowane jako jeden w literale



Łańcuchy znaków/Literały tekstowe - demo

```
1 reference
static void StringProbe1()
{
    Console.WriteLine("it is a tab \t and newline \n second line");
}


1 reference
static void StringProbe2()
{
    Console.WriteLine(@"it is not a tab \t and not a newline \n this line
quotation marks ""and then the second ""
end");
}
```



```
it is a tab      and newline
second line
it is not a tab \t and not a newline \n this line
quotation marks "and then the second "
end
```

C#

IDENTYFIKATORY

- **Identyfikator** to ciąg znaków zaczynający się od litery lub znaku podkreślenia '_', po którym następuje ciąg liter, cyfr, znaków podkreślenia.
- **Pojedynczy znak podkreślenia** jest zarezerwowany do specjalnych celów.
- Język C# **rozdziela małe i wielkie litery**, stąd identyfikatory `ident`, `Ident`, `IDent`, `idenT` są dla niego różne.
- Istnieją **słowa kluczowe**, których nie wolno używać jako identyfikatorów.
- Istnieją **kontekstowe słowa kluczowe**, których nie wolno używać jako identyfikatorów tylko w pewnych kontekstach (np. w wyrażeniu języka LINQ zbliżonym do języka SQL) 

Słowa kluczowe/kontekstowe słowa kluczowe

- Tabela z [1] str 41.

Tabela 1.1. Słowa kluczowe języka C#

abstract	enum	long	static
add* (1)	equals* (3)	nameof* (6)	string
alias* (2)	event	namespace	struct
as	explicit	new	switch
ascending* (3)	extern	null	this
async* (5)	false	object	throw
await* (5)	finally	on* (3)	true
base	fixed	operator	try
bool	float	orderby* (3)	typeof
break	for	out	uint
by* (3)	foreach	override	ulong
byte	from* (3)	params	unchecked
case	get* (1)	partial* (2)	unsafe
catch	global* (2)	private	ushort
char	goto	protected	using
checked	group* (3)	public	value* (1)
class	if	readonly	var* (3)
const	implicit	ref	virtual
continue	in	remove* (1)	void
decimal	int	return	volatile
default	interface	sbyte	where* (2)
delegate	internal	sealed	when* (6)
descending* (3)	into* (3)	select* (3)	while
do	is	set* (1)	yield* (2)
double	join* (3)	short	
dynamic* (4)	let* (3)	sizeof	
else	lock	stackalloc	

* Kontekstowe słowa kluczowe. Numer w nawiasie (*n*) wskazuje, w której wersji języka dodano określone kontekstowe słowo kluczowe.

Formaty identyfikatorów



- **NotacjaPascalowa** – każde słowo w identyfikatorze zaczynamy wielką literą, a potem następują małe (nawet jeśli słowo jest skrótem pisanym wielkimi literami jak np. HTTP) np.
 - `ComponentModel, HttpFileConnection, Configuration`
- **notacjaWielbłada**: pierwsze słowa z małych liter, potem jak w notacji pascalowej:
 - `firstName, quota, httpFileCollection`
- **Notacja węgierska**: identyfikator poprzedzany literowymi skrótami typów:
 - `iLicznik, dPlaca, piArrayOfInt`
 - **NIE STOSOWANA w C#**

Konwencja firmy Microsoft



- NotacjaPascalowa używana jest do nazw:
 - Klasy
 - Metody
 - Funkcji
 - Właściwości
 -
- notacjaWielbłąda używana jest do nazw:
 - Zmiennych lokalnych
 - Pól
 - Argumentów
- Identyfikatory pisane wielkimi literami używane są do nazw:
 - Stałych

Specjalne zasady identyfikatorów

- Microsoft sobie zastrzegł możliwość używania identyfikatorów zaczynających się od dwóch znaków podkreślenia, chociaż obecnie używa tylko 4 takie identyfikatory: `__arglist`, `__makeref`, `__reftype`, `__refvalue`.
- Ze względu na możliwość dołączania bibliotek z innych języków istnieje możliwość używania identyfikatorów, które są słowami kluczowymi. Należy je wtedy poprzedzać znakiem '@' np.:
 - `@return`, `@throw()`



C#

ZMIENNE, TYP STRING

Zmienne

- Zmienna to nazwa wskazująca wartość.
- Zmienna ma swój typ określany podczas jej deklaracji.
- Deklaracja zmiennej to:
`<TypZmiennej> <NazwaZmiennej>`
- Zamiast jednej zmiennej można zadeklarować kilka oddzielonych przecinkiem, jeśli mają ten sam typ.
- Występują tylko zmienne lokalne w ciele metod itp.
 - Nie ma zmiennych globalnych
- Podczas deklaracji można również zainicjować wartość zmiennej dodając znak '=' oraz wyrażenie zgodne z typem zmiennej.
- Niezainicjowane zmienne nie mogą być używane jako r-wartości:
 - r-wartość: (w uproszczeniu) mogą być po prawej stronie operatora przypisania '='

Zmienne - przykłady

```
static void Variables()  
{  
    int xyz = 10, abc;  
    double a = 2.3;  
    //xyz = abc;           // compile error:  
    // variable abc is uninitialized, it cannot be  
    // on the right side of =  
    abc = xyz;  
    int b = 23 * abc - 4;  
    a += 3;  
}
```

Typ **string**

- Typ do pamiętania ciągów znaków.
- Zapamiętany w zmiennej ciąg znaków jest **niemodyfikowalny**. Jedyna możliwość jego zmiany to stworzenie nowego ciągu i podstawienie pod zmienna typu **string**.
- Istnieje wiele gotowych metod do tworzenie zmodyfikowanych stringów:
 - **static string string.Format(...)**
 - **static string string.Concat(...)**
 - **static string string.Compare(...)**
 - **bool** `StartsWith(...)`
 - **bool** `EndsWith(...)`
 - **string** `ToLower()`
 - **string** `ToUpper()`
 - **string** `Trim(...)`
 - **string** `Replace(...)`
 - itd.
- Zdefiniowane są też operatory `+` oraz `+=` do konkatencji napisów.

Metody na typie string - przykłady

C# == Java

```
static void StringExample()  
{  
    string str1 = "abc Value";  
    str1 = str1.ToUpper();  
    System.Console.WriteLine(str1);  
    str1 = string.Concat(str1, "=xyz");  
    System.Console.WriteLine(str1);  
    string str2 = "some lines";  
    str2 += " and" + " a word";  
    System.Console.WriteLine(str2);  
}
```

```
ABC VALUE  
ABC VALUE=xyz  
some lines and a word
```

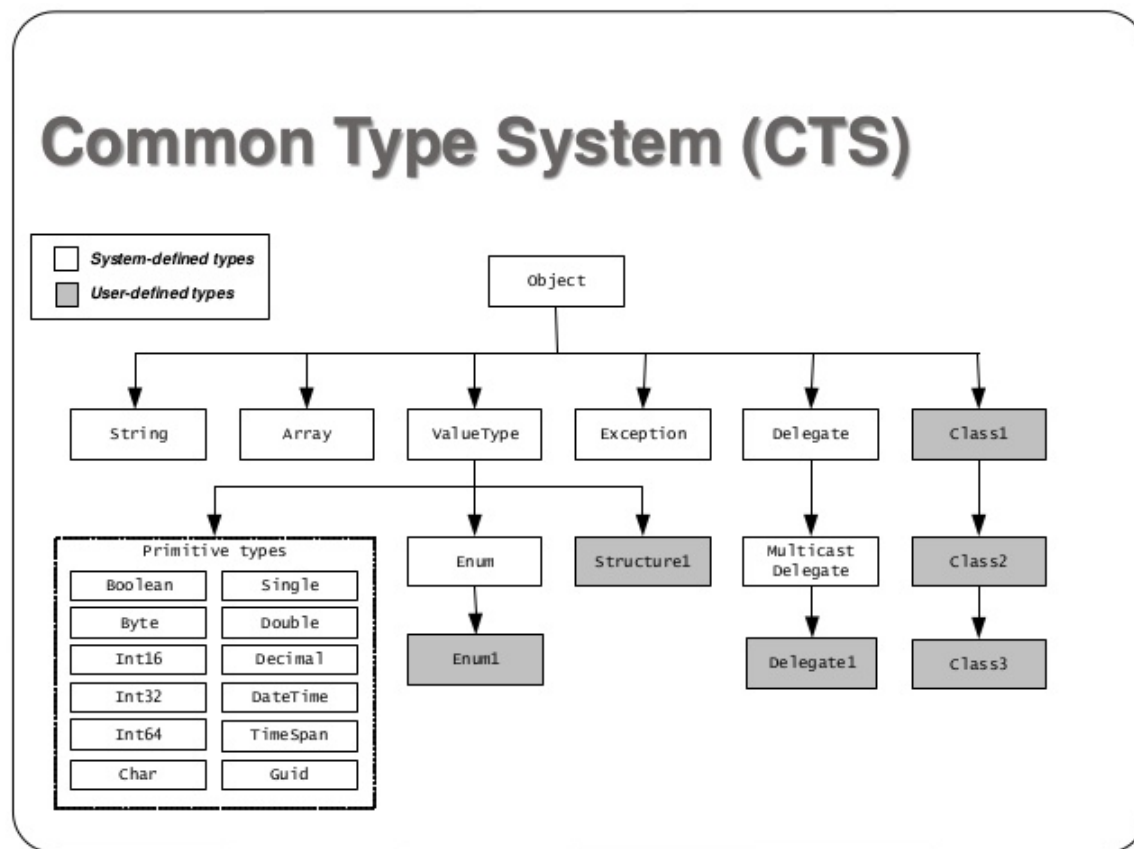
C#

KATEGORIE TYPÓW, WYBRANE CIEKAWE TYPY

Kategorie typów



- Wszystkie typy należą do jednej z dwóch kategorii – typów bezpośrednich/wartościowych (ang. *value types*) i typów referencyjnych (ang. *reference types*).
- Typy bezpośrednio kopiowane są zawsze przez **wartość**.
- Typy referencyjne kopiowane są przez **referencję** (wartość referencji).



Typy bezpośrednie/wartościowe

- Zmienna to nazwa (adres) komórek pamięci, które zawierają wartość zmiennej
- Należą do nich wszystkie typy podstawowe (bez **string**).
- Tworzone w pamięci na stosie, stąd nie powinny być duże.
- Można tworzyć własne typy bezpośrednie, podobnie do klas (ale ponieważ nie są to obiekty, to uboższe w możliwości).



Typy bezpośrednie - prezentacja



```
int a=123;  
double x=1.0  
int b=10;  
char znak='y';  
bool isCorrect=false;
```

stos		
isCorrect	1A04	false
znak	1A05	y
b	1A07	10
x	1A0B	1.0
a	1A13	123

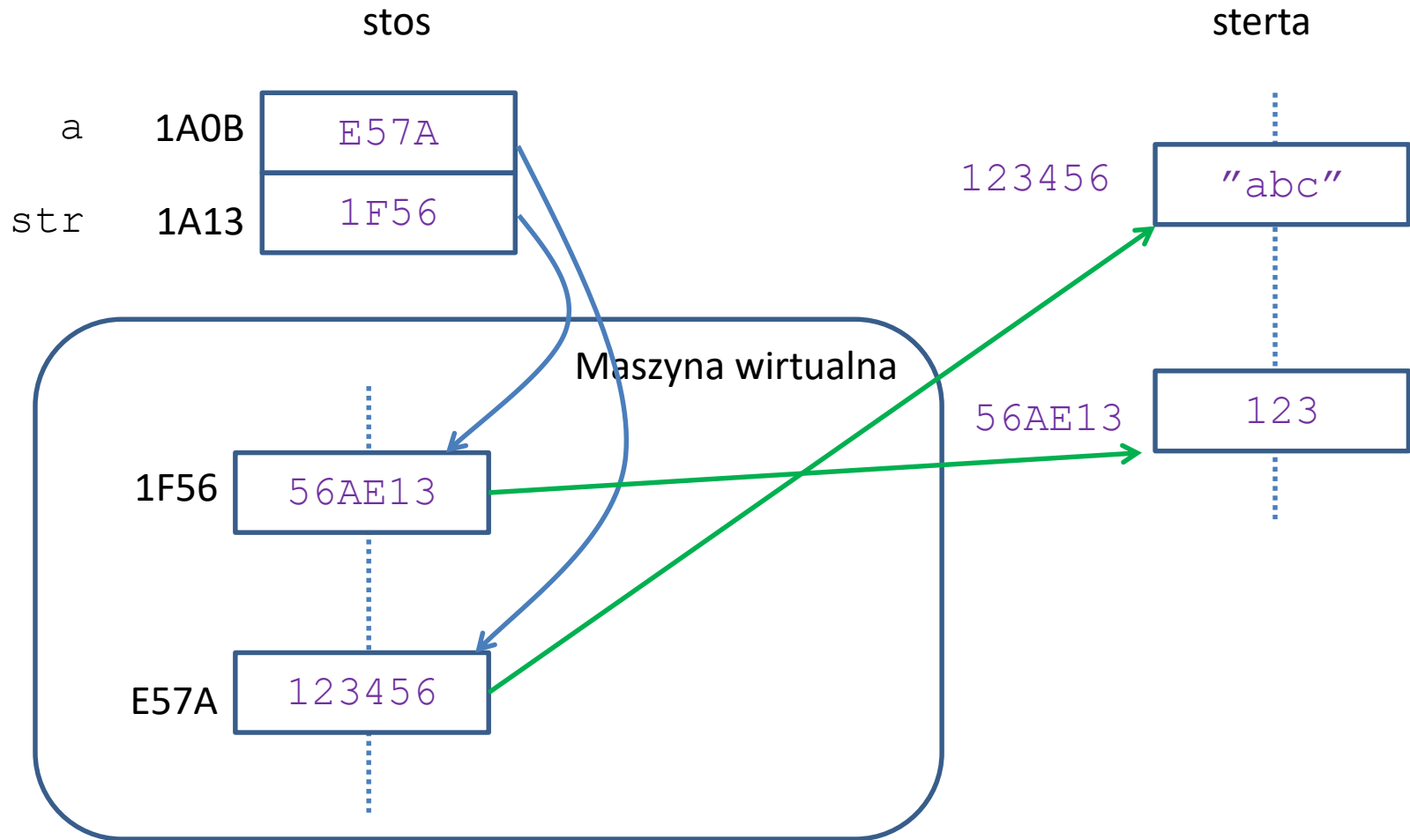
- **Zmienna typu referencyjnego** to adres komórki pamięci na **stosie**, w którym jest referencja do wartości zapamiętanej na **stercie**.
- Nie jest to bezpośredni adres, lecz „numer” w tablicy referencji maszyny wirtualnej.
- Wszystkie obiekty są pamiętane jako zmienne referencyjne, w tym również klasy **string**.
- Oczywiście można tworzyć własne typy referencyjne.

Typy referencyjne - prezentacja



```
string str="abc";
```

```
int? a=123;
```

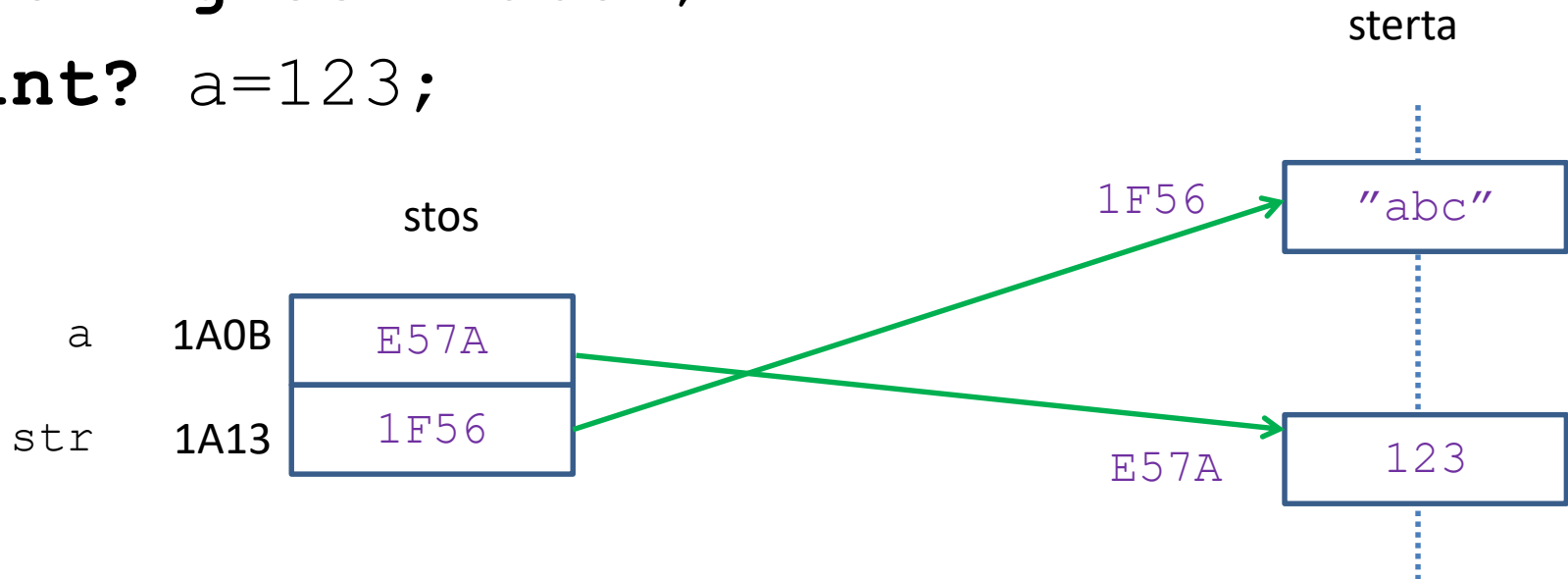


Typy referencyjne – prezentacja uproszczona

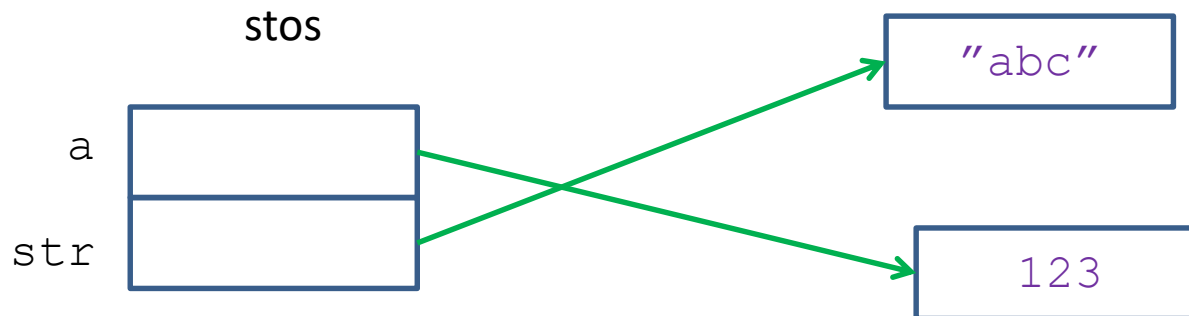


```
string str="abc";
```

```
int? a=123;
```



Typy referencyjne – prezentacja uproszczona v.2



Typy **nullowalne**



- Typy **nullowalne** to w sumie anonimowe typy referencyjne opakowujące typy strukturalne.
- Powstają przez dodanie znaku ‘?’ tuż po nazwie typu wartościowego.
- W razie potrzeby kompilator używa zmiennej tego typu jako referencji lub jako wartości bezpośredniej.
- W tym momencie można takiej zmiennej przypisać wartość **null**.
 - Przydaje się w operacjach na bazach danych, gdy kolumna z liczbą dopuszcza wartość pustą.
 - Lub gdy jakiś parametr może być nieokreślony.

Typy nullable - demonstracja



```
static void Nullable()
{
    int x = 1;
    //x = null; // compile error
    int? y = 2;
    y = null;
    System.Console.WriteLine(x == y);
    y = 1;
    System.Console.WriteLine(x == y);
}
```

False
True


Niejawny typ danych - **var**

C# == Java

- Słowo kluczowe **var** wstawione w miejsce typu podczas deklaracji zmiennej oznacza, że kompilator ma wywnioskować typ na podstawie wyrażenia za znakiem '='
- W środowiskach programistycznych można łatwo sprawdzić, jaki typ wywnioskował kompilator ustawiając myszkę nad nazwą zmiennej:



```
static void Var()  
{  
    var x = 1 + 3;  
    var str = "ciąg znaków";  
    var str2 = str.ToUpper();  
}  
  
Odwólar  
static void main(string[] args)
```

 **class System.String**
Represents text as a sequence of UTF-16 code units.

Anonimowe typy danych



- Niejawny typ danych jest bardzo przydatny przy **anonimowych typach danych**.
- Typy anonimowe tworzą się poprzez konstrukcję jednego obiektu (ew. kolekcji obiektów) przez operator **new** oraz nawiasy klamrowe, w których podawane są rozdzielone przecinkami nazwę pola, znak '=' oraz wartość pola.
- Typy anonimowe zostały w wielu sytuacjach zastąpione **krotkami**.
 - Ale niektóre metody wymagają jednak typów anonimowych.

```
static void Anonymous()  
{  
    var book1 = new { title = "The War of the Worlds", year = 1898 };  
    var book2 = new { title = "The Hobbit or There and Back Again", year = 1937 };  
    System.Console.WriteLine($"{book1.title} from year {book1.year}");  
    System.Console.WriteLine($"{book2.title} from year {book2.year}");  
}
```

```
The War of the Worlds from year 1898  
The Hobbit or There and Back Again from year 1937
```

Krotki



- Krotki to „zamiennik” typów anonimowych.
 - Ale niektóre metody wymagają jednak typów anonimowych.
- **Krotka nienazwana:** Ogólnie to kilka wartości (elementów) rozdzielone przecinkami ujęte w nawiasy okrągłe.
 - (wart1, wart2, ..., wartN)
- **Krotka nazwana:** elementy krotki mogą być **nazwane**, wówczas przed wartością jest identyfikator (nazwa elementu) i dwukropek
 - (nazwa1:wart1, nazwa2:wart2, ..., nazwaN:wartN)
- Wiele sposobów użycia/dostępu do elementów krotek
- 1) Przepisywanie krotki do **osobno deklarowanych zmiennych**
- 2) Przepisywanie krotki do osobno deklarowanych **wcześniej** zmiennych
- 3) Przepisywanie krotki do osobno deklarowanych zmiennych z **niejawnie określonym typem**

```
(string name1, string author1, double price1) = ("Pan Tadeusz", "Adam Mickiewicz", 120.0);
System.Console.WriteLine($"Lsiążka {name1} autora {author1} w cenie {price1} zł.");

string name2;
string author2;
double price2;
(name2, author2, price2) = ("Pan Tadeusz", "Adam Mickiewicz", 120.0);
System.Console.WriteLine($"Lsiążka {name2} autora {author2} w cenie {price2} zł.");

(var name3, var author3, var price3) = ("Pan Tadeusz", "Adam Mickiewicz", 120.0);
System.Console.WriteLine($"Lsiążka {name3} autora {author3} w cenie {price3} zł.");
```

```
Book Hothouse by Brian W. Aldiss for 120 zł.
Book Hothouse by Brian W. Aldiss for 120 zł.
Book Hothouse by Brian W. Aldiss for 120 zł.
```

Krotki - dostęp



- 4) Przepisywanie krotki do osobno deklarowanych zmiennych z **niejawnie** określonym typem **wszystkich** zmiennych
- 5) Deklarowanie **nazwanej** krotki, przypisanie jej wartości (z nienazwanej krotki) i dostęp do elementów za pomocą nazw
- 6) Przypisywanie krotki z nazwanymi elementami do elementu o niejawnie określonym typie i dostęp do elementów krotki na podstawie nazw.
- 7) Przypisywanie krotki z elementami bez nazw do jednej zmiennej o niejawnie określonym typie i dostęp do elementów krotki na podstawie numerowanych właściwości.

```
var (name4, author4, price4) = ("Pan Tadeusz", "Adam Mickiewicz", 120.0);
System.Console.WriteLine($"Lsiążka {name4} autora {author4} w cenie {price4} zł.");

(string Name, string Author, double Price) bookInfo5 = ("Pan Tadeusz", "Adam Mickiewicz", 120.0);
System.Console.WriteLine($"Lsiążka {bookInfo5.Name} autora {bookInfo5.Author} w cenie {bookInfo5.Price} zł.");

var bookInfo6 = (Name: "Pan Tadeusz", Author: "Adam Mickiewicz", Price: 120.0);
System.Console.WriteLine($"Lsiążka {bookInfo6.Name} autora {bookInfo6.Author} w cenie {bookInfo6.Price} zł.");

var bookInfo7 = ("Pan Tadeusz", "Adam Mickiewicz", 120.0);
System.Console.WriteLine($"Lsiążka {bookInfo7.Item1} autora {bookInfo7.Item2} w cenie {bookInfo7.Item3} zł.");
```

```
Book Hothouse by Brian W. Aldiss for 120 zł.
Book Hothouse by Brian W. Aldiss for 120 zł.
Book Hothouse by Brian W. Aldiss for 120 zł.
Book Hothouse by Brian W. Aldiss for 120 zł.
```

Krotki - dostęp



- 8) Przypisywanie krotki z nazwanymi elementami do elementu o niejawnie określonym typie i dostęp do elementów krotki na podstawie numerowanych właściwości.
 - Inne użycie zmiennej z przypadku 6)
- 9) Pomijanie fragmentów krotek za pomocą znaku podkreślenia (joker)
- 10) Nazwy elementów krotek mogą zostać wywnioskowane na podstawie nazw zmiennych i właściwości (od C# 7.0)
- Krotki mogą być **parametrami funkcji** lub jej **wynikiem**.
- Zapis typu krotki jak dla `bookInfo5`.
 - Mogą być też same typy bez nazw.
- Od C# 7.3 krotki **można porównywać** za pomocą `==` i `!=`
 - Pod uwagę brane są **kolejne** elementy krotki (nazwy są **pomijane**), występuje rzutowanie typów (`int` na `float` itp.)
- Krotki to typ **wartościowy** `System.ValueTuple`.
 - I tylko dla nich można używać przedstawiony zapis nawiasowy
 - Istnieje typ `System.Tuple`, który jest typem **referencyjnym**.

```
var bookInfo8 = (Name: "Pan Tadeusz", Author: "Adam Mickiewicz", Price: 120.0);
System.Console.WriteLine($"Lsiążka {bookInfo8.Item1} autora {bookInfo8.Item2} w cenie {bookInfo8.Item3} zł.");

//(string Name, _, double Price) bookInfo9 = ("Pan Tadeusz", "Adam Mickiewicz", 120.0);
//System.Console.WriteLine($"Lsiążka {bookInfo9.Name} autora {bookInfo9.Item2} w cenie {bookInfo9.Item3} zł.");

string nameX= "Pan Tadeusz";
string authorX= "Adam Mickiewicz";
double priceX=120.0;
var bookInfoX=(nameX, authorX, priceX);
System.Console.WriteLine($"Lsiążka {bookInfoX.nameX} autora {bookInfoX.authorX} w cenie {bookInfoX.Item3} zł.");
```

```
Book Hothouse by Brian W. Aldiss for 120 zł.
Book Hothouse for 120 zł.
Book Hothouse by Brian W. Aldiss for 120 zł.
```

- Zapis z nawiasami kwadratowymi poprzedzonymi nazwą typu, po których następuje nazwa zmiennej np.:

```
int [] tablicaInt;
```

- Tablice wielowymiarowe:

```
int [,] –tablica dwuwymiarowa
```

```
int [, ,] – tablica trzowymiarowa
```



- Tablice tablic:

```
int [] [] – tablica tablic
```

```
int [] [] [] – tablica tablic tablic
```

- Dowolne mieszanie typów tablicowych:

```
int [,] [,] – dwuwymiarowa tablica tablic  
dzwuwymiarowych.
```

- Tablica to typ **referencyjny**, jest obiektem.
- Indeksy tablic zaczynają się od 0.

Tablice jednowymiarowe

- Tworzenie tablicy:

```
int[] tab=new int[10];
```

- Jeśli **nie zainicjujemy** wartości tablicy wprost, będą to wartości **domyślne** dla typu elementu.
- Inicjowanie wartości początkowych (w drugim przypadku rozmiar tablicy musi się zgadzać z ilością elementów):

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

```
int[] tab2=new int[4]{1,2,3,4};
```

- Inicjowanie tablicy poza deklaracją:

```
int[] tab;
```

```
tab=new int[10];
```

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

- Korzystanie z elementów tablicy za pomocą **akcesora tablicy**, czyli indeksu w notacji nawiasowej:

```
tab[3]=5*tab[1];
```

- Użycie indeksu **spoza zakresu** kończy się zgłoszeniem **wyjątku**.
- Aktualna długość dostępna poprzez właściwość `Length`, więc dostęp do ostatniego elementu to tablica `tab`:

```
tab[tab.Length-1];
```

- (C# 8) Indeksowanie od końca `tab[^x] == tab[tab.Length-x]`, czyli `tab[^1]` to ostatni element tablicy, `tab[^2]` –przedostatni, itd.





- Inicjowanie samodzielne:
`int[,] cells = { {0, 1, 2}, {3, 4, 5} } ;`
- Lub
`int[,] cells = new
int[2, 3] { {0, 1, 2}, {3, 4, 5} } ;`
- Liczba elementów w każdym wymiarze **musi być** ustalona.
- Dostęp do elementu:
`cells[1, 1] = 2 + cells[0, 0] ;`
- Właściwość `Length` oznacza liczbę wszystkich elementów (dla zmiennej `cells` to 6).
- Właściwość `Rank` zwraca liczbę wymiarów.
- Metoda `GetLength(dimension)` zwraca liczbę elementów w podanym wymiarze.

Klasa **string**

- Klasa ta to w pewnym sensie tablica, stąd ma zaprogramowany akcesor dla tablic.

```
string str=„Jacek”;
```

```
System.Console.Write(str[2]); // 'c'
```

Posiada też metodę to przekształcenia na tablicę char-ów:

```
char [] charArr=str.ToArray();
```

OPERATORY, INSTRUKCJE

Operatory dwu argumentowe

- Dla typów całkowitoliczbowych (priorytetami):

$*$, $/$, $\%$

$+$, $-$,

$=$

- Dla typów rzeczywistych

$*$, $/$

$+$, $-$

$=$

- Do zmiany kolejności działań służą nawiasy okrągłe ``(`` oraz ``)``
- Wykonanie wywołań metod są zawsze od lewej do prawej niezależnie z jakimi operatorami używane:



`A() + B() * C()`

najpierw będzie obliczenie metod w kolejności `A()`, `B()`, `C()`, a dopiero później działania wg priorytetów. (przykład w kodzie)

- W typach rzeczywistych pamiętać o niedokładności wyników.
- W wyniku niepoprawnej operacji matematycznej wartością zmiennej może być `NaN` (not a number) lub `-Infinity` lub `Infinity`.

Kolejność wykonywania wywołań

```
static int orderCounter = 0;

1 reference
static int A()
{
    orderCounter++;
    Console.WriteLine($"A() - >orderCounter={orderCounter}");
    return 4;
}

1 reference
static int B()
{
    orderCounter++;
    Console.WriteLine($"B() -> orderCounter={orderCounter}");
    return 5;
}

1 reference
static int C()
{
    orderCounter++;
    Console.WriteLine($"C() -> orderCounter={orderCounter}");
    return 6;
}

1 reference
static void OrderInExpression()
{
    #if VERBOSE
        Console.WriteLine("Testing "+nameof(OrderInExpression));
    #endif

    Console.WriteLine(A() + B() * C());
}
```

```
A() - >orderCounter=1
B() -> orderCounter=2
C() -> orderCounter=3
34
```

Złożone operatory przypisania

- Złożone operatory przypisania:

`+=`, `-=`, `*=`, `/=`, `%=`

- Operatory inkrementacji i dekrementacji:

`++`, `--`

- przed zmienną (pre –inkrementacja/dekrementacja)
- po zmiennej (post –inkrementacja/dekrementacja)

Instrukcje sterujące

- **if** (<wyrażenieLogiczne>)
 <instrukcjaJeśliPrawda>
- **if** (<wyrażenieLogiczne>)
 <instrukcjaJeśliPrawda>
else
 <instrukcjaJeśliFałsz>
- **while** (<wyrażenieLogiczne>)
 <instrukcjaJeśliPrawda>
- **do**
 <instrukcjaRazOrazJeśliPrawda>
while (<wyrażenieLogiczne>)

if, if else, while - przykład

```
static void CollatzProblem(int n)
{
    if (n < 1) return;
    while (n != 1)
    {
        Console.Write(n+" ");
        if (n % 2 == 0)
            n /= 2;
        else
            n = 3 * n + 1;
    }
    Console.WriteLine(n);
}

static void CollatzProblemTest()
{
    CollatzProblem(17);
    CollatzProblem(15);
    CollatzProblem(-5);
}
```

```
17 52 26 13 40 20 10 5 16 8 4 2 1
15 46 23 70 35 106 53 160 80 40 20 10 5 16 8 4 2 1
C:\Users\daniu\source\repos\ControlStatements\Centr
```


do/while - przykład

```
static int BinarySearch(int[] arr, int value)
{
    int minIdx = 0;
    int maxIdx = arr.Length - 1;
    do
    {
        int midIdx = (minIdx + maxIdx) / 2;
        if (arr[midIdx] == value)
            return midIdx;
        else if (arr[midIdx] < value)
            minIdx = midIdx+1;
        else
            maxIdx = midIdx-1;
    }
    while (minIdx < maxIdx);
    return minIdx;
}

static void BinarySearchTest()
{
    Console.WriteLine(BinarySearch(new int[] { 1, 3, 5, 7, 9, 11, 13, 15 }, 5));
    Console.WriteLine(BinarySearch(new int[] { 1, 3, 5, 7, 9, 11, 13, 15 }, 15));
    Console.WriteLine(BinarySearch(new int[] { 1, 3, 5, 7, 9, 11, 13, 15 }, 1));
    Console.WriteLine(BinarySearch(new int[] { 1, 3, 5, 7, 9, 11, 13, 15 }, 0));
    Console.WriteLine(BinarySearch(new int[] { 1, 3, 5, 7, 9, 11, 13, 15 }, 8));
}
```

2
7
0
0
4

Instrukcje sterujące

- **for** (<inicjowanieDlaFor>; <wyrażenieLogiczne>; <iteratorDlaFor>)
 <instrukcjaJeśliPrawda>
- **foreach** (<typ> <zmienna> **in** <kolekcja>)
 <instrukcja>
- <Kolekcja> to obiekt odpowiedniej klasy lub tablica.
- <Typ> musi być zgodny z typem kolekcji. Można użyć **var**
- <zmienna> jest **niezmienna** w ciele pętli **foreach** 

Pętla for oraz foreach

```
static void ForLoopTest()  
{  
    int[] arr = { 1, 3, 5, 7 };  
    int sum = 0;  
    for (int i = 0; i < arr.Length; i++)  
        sum += arr[i];  
    Console.WriteLine(sum);  
    sum = 0;  
    foreach (int value in arr)  
        sum += value;  
    Console.WriteLine(sum);  
    for (int i = 0; i < arr.Length; i++)  
        arr[i]++;  
    //foreach (int value in arr)  
    //    value++; // error: variable - constant in body of foreach loop  
}
```



16
16

Instrukcje sterujące

- **switch** (<wyrażenieNadrzędne>) {
 case <stałeWyrażenie1>:
 <listaInstrukcji>
 <instrukcjaSkoku>
 case <stałeWyrażenie2>:
 <listaInstrukcji>
 <instrukcjaSkoku>
 ...
 default :
 <listaInstrukcji>
 <instrukcjaSkoku>
}
- Stałe wyrażenie musi być typu całkowitoliczbowego, logicznego, znakowego lub **string**
- Jeśli jest jakakolwiek instrukcja, musi być ogranicznik (**break** lub **goto**)
- Instrukcje **goto** (dla **switch**):
 - **goto case** <stałeWyrażenie>;
 - **goto default**;



Instrukcja switch/case - przykład

```
static void SwitchInstruction(string line)
{
    switch (line)
    {
        case "Good":
        case "OK": Console.WriteLine("All right");
            break;
        case "Super": Console.Write("Extra - ");
            goto case "OK";
        case "Can be":
            Console.WriteLine("It is fine");
            break;
        default:
            Console.WriteLine("I don't understand...");
            break;
    }
}

static void SwitchInstructionTest()
{
    SwitchInstruction ("Good");
    SwitchInstruction ("Can be");
    SwitchInstruction ("Super");
    SwitchInstruction ("So-so");
}
```



```
All right
It is fine
Extra - All right
I don't understand...
```

Switch/case c.d. 1



Inne postacie instrukcji switch/case:

- Argumentem `switch(X)` może być zmienna dowolnego typu (nawet typu generycznego). Wówczas przypadkami `case` są wyrażenia oznaczające deklarację zmiennej. Jeśli w danym przypadku możliwe jest rzutowanie na dany typ, następuje wykonanie tegoż przypadku.

```
private static void ShowCollectionInformation<T>(T coll){
    switch (coll){
        case Array arr:
            Console.WriteLine($"An array with {arr.Length} elements.");
            break;
        case IEnumerable<int> ieInt:
            Console.WriteLine($"Average: {ieInt.Average(s => s)}");
            break;
        case System.Collections.IList list:
            Console.WriteLine($"{list.Count} items");
            break;
        case IEnumerable ie:
            string result = "";
            foreach (var e in ie)
                result += $"{e} ";
            Console.WriteLine(result);
            break;
        case null:
            Console.WriteLine("Null passed to this method.");
            break;
        default:
            Console.WriteLine($"A instance of type {coll.GetType().Name}");
            break;
    }
}
```

<https://docs.microsoft.com/pl-pl/dotnet/csharp/language-reference/keywords/switch>

Switch/case c.d. 2



Inne postacie instrukcji **switch/case**:

- Rozszerzając poprzednią postać instrukcji **switch/case** można dodać warunek uruchomienia przypadku poprzez klauzulę **when**), w którym będzie dowolne wyrażenie logiczne zawierające najczęściej zmienną zadeklarowanej w danym przypadku **case**.

```
private static void ShowShapeInfo(Shape sh) {  
    switch (sh)  
    {
```

<https://docs.microsoft.com/pl-pl/dotnet/csharp/language-reference/keywords/switch>

```
// Note that this code never evaluates to true.
```

```
    case Shape shape when shape == null:
```

```
        Console.WriteLine($"An uninitialized shape (shape == null)");
```

```
        break;
```

```
    case null:
```

```
        Console.WriteLine($"An uninitialized shape");
```

```
        break;
```

```
    case Shape shape when sh.Area == 0:
```

```
        Console.WriteLine($"The shape: {sh.GetType().Name} with no dimensions");
```

```
        break;
```

```
    case Square sq when sh.Area > 0:
```

```
        Console.WriteLine("Information about square:");
```

```
        Console.WriteLine($"    Length of a side: {sq.Side}");
```

```
        Console.WriteLine($"    Area: {sq.Area}");
```

```
        break;
```

```
    case Rectangle r when r.Length == r.Width && r.Area > 0:
```

```
        Console.WriteLine("Information about square rectangle:");
```

```
        Console.WriteLine($"    Length of a side: {r.Length}");
```

```
        Console.WriteLine($"    Area: {r.Area}");
```

```
        break;
```


Instrukcje **break**, **continue**, **goto**

Instrukcje do specjalnego zastosowania w pętlach:

- **break**
 - Przerywa działanie pętli (tylko tej najbardziej wewnętrznej, w której znajduje się instrukcja **break**). Sterowanie przechodzi o kolejnej instrukcji za pętlą.
- **continue**
 - Przerywa działanie pętli (tylko tej najbardziej wewnętrznej, w której znajduje się instrukcja **continue**). Sterowanie przechodzi do początku pętli. Jeśli jest to pętla **for** następuje wykonanie instrukcji iteracyjnych tej pętli, a następnie w każdym przypadku następuje sprawdzenie warunku kontynuacji wykonania pętli.
- Instrukcja **goto** ogólna (ZABRONIONA):
 - **goto** <identyfikator>

Operatory

- Operatory porównywania
`<, >, <=, >=, ==, !=`
- Operatory logiczne
`&&, ||, ^ (XOR), ! (negacja)`
- Operator warunkowy („krótki if”)?:
`<warunek>?<wyrażenieDlaTrue>:<wyrażenieDlaFalse>`
- Operator ??
`<wyrażenie>??<wyrażenieJeśliNull>`
- Operator ?.
`<wyrażenieTypReferencyjny>?.<MetodaLubWłaściwość>`



jeśli wyrażenie przed operatorem ?. będzie **null**, to reszta się nie wykona i wartością będzie **null**. Operator ten często łączy w wyrażeniu się z operatorem ??


`str?.Length??0`

dla `str` równego **null** i tak wynikiem będzie 0, a nie wyjątek `System.NullReferenceException`

Operatory - przykłady

```
static bool IsText(string str)
{
    return (str?.Length ?? 0) != 0;
}

static void ShortIfsTest()
{
    Random rnd = new Random();
    int x = rnd.Next(100);
    string str = x < 50 ? "to low" : "to high";
    Console.WriteLine($"drawn {x} so {str}");
    Console.WriteLine($"jest tekst {IsText(null)}");
    Console.WriteLine($"jest tekst {IsText("")}");
    Console.WriteLine($"jest tekst {IsText("it is")}");
}
```



```
drawn 57 so to high
is a text: False
is a text: False
is a text: True
```

```
drawn 41 so to low
is a text: False
is a text: False
is a text: True
```

Dyrektywy preprocesora



- Dyrektywy kompilatora są brane pod uwagę tylko w czasie kompilacji i mają wpływ na ten proces. Mogą włączać/wyłączać kod, który ma podlegać kompilacji, generować ostrzeżenia lub błędy kompilacji.
 - `#define`
 - `#undef`
 - `#if`
 - `#endif`
 - `#elif`
 - `#else`
 - `#error`
 - `#warning`
 - `#pragma`
 - `#line`
 - `#line default`
- Regiony nie są w zasadzie dyrektywą kompilatora, lecz środowiska programistycznego (np. Visual Studio 2019 Community) i pozwalają związać pewną partię kodu do jednej linijki z komentarzem w ramach edytora tekstowego kodu.
 - `#region`
 - `#endregion`

<https://docs.microsoft.com/dotnet/csharp/language-reference/preprocessor-directives/>

Operatory na bitach

- Operatory bitowe:
 $\gg, \ll, |, \&, ^, \sim$ (negacja binarna)
- Operatory przypisania bitowe:
 $\gg=, \ll=, |=, \&=, ^=$
- Wynikiem działania operatorów jest co najmniej typ **int**.

Operatory binarne - przykłady

```
public static string MyBinary(int value)
{
    const int len = 32;
    string ret = Convert.ToString(value, 2);
    while (ret.Length < len)
        ret = "0" + ret;
    return ret;
}
```

```
static void BinaryOperators()
```

```
{
    int value = 10;
    int mask = 3;
    Console.WriteLine($"value = {MyBinary(value)}");
    Console.WriteLine($"mask  = {MyBinary(mask)}");
    Console.WriteLine("-----");
    Console.WriteLine($"v | m = {MyBinary(value | mask)}");
    Console.WriteLine($"v & m = {MyBinary(value & mask)}");
    Console.WriteLine($"v ^ m = {MyBinary(value ^ mask)}");
    Console.WriteLine($"~v   = {MyBinary(~value)}");
    Console.WriteLine($"v >> 2= {MyBinary(value >>2)}");
    Console.WriteLine($"v << 2= {MyBinary(value << 2)}");
}
```

```
value = 00000000000000000000000000001010
mask  = 00000000000000000000000000000011
-----
v | m = 00000000000000000000000000001011
v & m = 00000000000000000000000000000010
v ^ m = 00000000000000000000000000001001
~v    = 11111111111111111111111111111010
v >> 2= 00000000000000000000000000000010
v << 2= 0000000000000000000000000000101000
```

METODY

Nagłówek metody

- Nagłówek metody

```
<typZwracany>  
  <nazwaMetody> ([<listaParametrówFormalnych>])
```

- Lista parametrów formalnych to oddzielone przecinkami kolejne parametry wyrażone jako:

```
[<modyfikator>] <typParametru>  
<nazwaParametru>
```

- Wywołanie metody:

```
<nazwaMetody> (listaArgumentów)
```

- Lista argumentów musi być **zgodna** z listą parametrów formalnych (szczegóły dalej)
- Nazwa metody i nazwy parametrów to poprawne identyfikatory
- Metoda zawsze jest związana z typem (klasą lub obiektem) na rzecz którego może być wywołana.
- Standard Microsoftu zakłada użycie notacji pascalowej dla nazw metod i wielbłądziej dla parametrów.



Przekazywanie parametrów przez wartość

- Jeśli nie ma przy parametrze modyfikatora oznacza, że jest przekazywany przez wartość (kopia wartości trafia do procedury):
 - Dla typów wartościowych to kopia wartości
 - Dla typów referencyjnych to kopia referencji
- Ten rodzaj przekazywania parametrów pozwala umieszczać wyrażenia jako argumenty.
- Zmienne w wyrażeniach parametrów muszą być zainicjowane.
- To oznacza, że modyfikacja tych kopii wartości podczas wykonywania procedury nie ma wpływu na wartość podaną w wywołaniu procedury
- W typach referencyjnych znaczy to, że możemy jednak zmienić zawartość, na którą wskazują

Przekazywanie parametrów przez referencję - **ref**



- Gdy przed typem parametru dopiszemy słowo kluczowe **ref**, „przekazana będzie referencja”. W kodzie oznacza to, że korzystanie z parametru formalnego oznacza korzystanie z oryginalnej zmiennej przekazanej jako parametr.
- Nie można używać wyrażeń, tylko zmiennych.
- Zmienne muszą być zainicjowane.
- Nie można używać właściwości (będzie w przyszłości)
- Zmiana wartości parametru w czasie działania procedury zmienia oryginalną wartość.
 - Zmiana jest po powrocie z wywołania metody
 - Uważać podczas programowania współbieżnego
- Podczas wywoływania metody przed argumentem również trzeba wpisać słowo kluczowe **ref**.

Parametry wyjściowe - **out**



- Gdy przed typem parametru dopiszemy słowo kluczowe **out**, parametr **musi być zmieniony** w trakcie działania metody. W kodzie oznacza to, że korzystanie z parametru formalnego oznacza korzystanie z oryginalnej zmiennej przekazanej jako parametr.
- Nie można używać wyrażeń, tylko zmiennych.
- Nie można używać właściwości (będzie w przyszłości)
- Zmienne NIE muszą być zainicjowane.
 - Wręcz kompilator zakłada, że nie są zainicjowane, więc nie mogą być używane jako r-wartość.
- Zmiana wartości parametru w czasie działania procedury zmienia oryginalną wartość.
- Podczas wywoływania metody przed argumentem również trzeba wpisać słowo kluczowe **out**.

Parametry - przykład

```
static void DiffForwardingInt(int a, ref int b, out int c)
{
    a++;
    b++;
    //c++; //error, c can be not initialized
    c = 50; // without initialization - compile error
}

static void DiffForwardingIntTest()
{
    int x = 10;
    int y = 100;
    int z; // can be not initialized
    Console.WriteLine($"x={x}, y={y}");
    DiffForwardingInt(x, ref y, out z);
    Console.WriteLine($"x={x}, y={y}, z={z}");
}
```

```
x=10, y=100
x=10, y=101, z=50
```



Referencje tylko do odczytu - **in**



- Dla **typów wartościowych** istnieje możliwość przekazania zmiennych jako referencja, ale tylko z możliwością odczytu poprzez słowo kluczowe **in** przed typem parametru.
- Ma to sens przy dużych pamięciowo typach wartościowych.

Przekazywanie typów strukturalnych



- Komentarz: jak działa operator przypisania dla tego typu

```
using System.Drawing;
...
static void DiffForwardingStruct(Point p1, ref Point p2, out Point p3)
{
    p1.X++;
    p2.X++;
    p1 = new Point(5, 5);
    //p2 = new Point(6, 6);
    p3 = new Point(7, 7);
}
static void DiffForwardingStructTest()
{
    Point pkt1 = new Point(1, 1);
    Point pkt2 = new Point(10, 10);
    Point pkt3;
    Console.WriteLine($"pkt1={pkt1}, pkt2={pkt2}");
    roznePrzekazywaniaStruct(pkt1, ref pkt2, out pkt3);
    Console.WriteLine($"pkt1={pkt1}, pkt2={pkt2}, pkt3={pkt3}");
}
```

```
pkt1={X=1,Y=1}, pkt2={X=10,Y=10}
pkt1={X=1,Y=1}, pkt2={X=6,Y=6}, pkt3={X=7,Y=7}
```

Przekazywanie typów referencyjnych



- Komentarz: jak działa operator przypisania dla tego typu

```
class CPoint
{
    public int x,y;
    public CPoint(int x, int y) { this.x = x; this.y = y; }
    public override string ToString()
    {
        return $"{{x={x},y={y}}}" ;
    }
}

static void DiffForwardingRefType(CPoint p1, ref CPoint p2, out CPoint p3)
{
    p1.x++;
    p2.x++;
    p1 = new CPoint(5, 5);
    //p2 = new CPoint(6, 6);
    p3 = new CPoint(7, 7);
}

static void DiffForwardingRefTypeTest()
{
    CPoint pkt1 = new CPoint(1, 1);
    CPoint pkt2 = new CPoint(10, 10);
    CPoint pkt3;
    Console.WriteLine($"pkt1={pkt1}, pkt2={pkt2}");
    DiffForwardingRefType(pkt1, ref pkt2, out pkt3);
    Console.WriteLine($"pkt1={pkt1}, pkt2={pkt2}, pkt3={pkt3}");
}
```

```
pkt1={x=1,y=1}, pkt2={x=10,y=10}
pkt1={x=2,y=1}, pkt2={x=11,y=10}, pkt3={x=7,y=7}
```



Parametry opcjonalne

- Parametry opcjonalne w metodzie (lub metoda z wartościami domyślnymi) pozwala na nie podawanie wszystkich parametrów. Parametry, które nie podamy zostaną zainicjowane wartościami podanymi w definicji metody.
- Nie można używać jako **ref/out**
- Wszystkie parametry opcjonalne muszą zgrupowane na końcu listy parametrów.

```
static void OptionalParameters(int number, string name, int marker = 0, string ident = "noValue")
{
    Console.WriteLine($"{number},{name},{marker},{ident}");
}
static void OptionParametersTest()
{
    OptionalParameters(1, "Kaczyński", 2, "Nadprezydent");
    OptionalParameters(2, "Nowak", 1);
    OptionalParameters(3, "Kowalski");
}
```

```
1,Kaczyński,2,Nadprezydent
2,Nowak,1,noValue
3,Kowalski,0,noValue
```




Tablice parametrów - **params**

- Ostatni argument, jeśli jest tablicą elementów o tym samym typie, może być oznaczony słowem **params**.
- Umożliwia on podczas wywołania podawać elementy takiej tablicy oddzielone przecinkami
- Tablica parametrów nie może być z modyfikatorami **in/out/ref** ani nie może mieć wartości domyślnych

```
static string ForwardingParams (string name, params int[] tab)
{
    int sum = 0;
    foreach(int value in tab)
    {
        sum += value;
    }
    return name + sum;
}

static void ForwardingParamsTest()
{
    int[] tab = new int[] { 1, 2, 3, 4 };
    Console.WriteLine(ForwardingParams("as a variable tab: ", tab));
    Console.WriteLine(ForwardingParams("as an array: ", new int[] { 1, 2, 3, 4 }));
    Console.WriteLine(ForwardingParams("as parameters: ", 1, 2, 3, 4));
}
```

```
as a variable tab: 10
as an array: 10
as parameters: 10
```



Wartości zwracane przez metodę

- Jeśli jako typ zwracany jest słowo kluczowe **void**, nic nie jest zwracane (procedura)
- Jeśli nie ma modyfikatora **ref** przed typem zwracanym przez metodę, to zwracana jest kopia wartości umieszczona po słowie kluczowym **return**.
 - Zachowanie względem typów wartościowych/referencyjnych analogiczne jak dla parametrów.
- Jeśli jest modyfikator **ref**, to nie jest tworzona kopia, ale przekazywana referencyjnie wartość. To znaczy, że ta część pamięci musi nadal „istnieć” po zakończeniu wykonywania metody. Czyli nie może to być zmienna lokalna wywoływanej metody (ale może to być zmienna lokalna wywołującej metody).
- Przypisanie wyniku takiej metody może być tylko na zmienną, która jest zmienną referencyjną. Taka zmienna w deklaracji ma typ poprzedzony słowem **ref** i musi być zainicjowana podczas deklaracji (np. poprzez wywołanie metody zwracającej referencję). W kolejnych instrukcjach taka zmienna nie może być już zmieniana.



Argumenty nazwane

- Standardowo parametry należy podawać w kolejności w jakiej występują w nagłówku funkcji
- Jeśli pamięta się nazwy parametrów, można ich użyć podczas wywołania, dokładnie w formacie:
`<modyfikator> <nazwaParametru>:<wartośćParametru>`
- Najpierw mogą być argumenty nienazwane w poprawnej kolejności, natomiast argumenty nazwane muszą być **na końcu**.
 - Chyba, że argument nazwany jest „na swoim miejscu”, to nie zalicza się do tej reguły.
- Wtedy kolejność parametrów nazwanych nie jest istotna.
- Mechanizm ten skraca kod, gdy w metodzie istnieje wiele parametrów domyślnych, z których chcemy ustawić tylko niewielką część.

Argumenty nazwane - przykład



```
static void OptionalParameters(int number, string name, int marker = 0, string ident =  
"noValue")  
{  
    Console.WriteLine($"{number},{name},{marker},{ident}");  
}  
  
static void NamedArgumentsTest()  
{  
    OptionalParameters(5, marker: 5, name: "Suweren");  
    OptionalParameters(ident: "rycerz", name: "Zawisza Czarny", number: 6);  
    OptionalParameters(7, "Żuk", 0, "robaczek");  
        // you have to remember what is the value of marker parameter  
    OptionalParameters(7, "Żuk", ident: "robaczek");  
}
```

```
5,Suweren,5,noValue  
6,Zawisza Czarny,0,rycerz  
7,Żuk,0,robaczek  
7,Żuk,0,robaczek
```

Przeciążanie metod



- Może istnieć wiele metod o tej samej nazwie, jednak muszą się różnić liczbą parametrów lub ich typami.
- Metody z parametrami domyślnymi to w zasadzie kilka metod o tej samej nazwie.
- Nie można przeciążać metody tylko poprzez typ zwracanego wyniku.



- W przypadku przeciążania metod, używania parametrów domyślnych itp. kompilator stara się dopasować typy argumentów (z ewentualnym rzutowaniem) to parametrów formalnych.
- Jeśli pasuje wersja metody z parametrami opcjonalnymi i bez, kompilator wybierze wersję bez parametrów.
- Jeśli wystąpi niejednoznaczność, będzie błąd kompilacji. Żeby naprowadzić kompilator na właściwą metodę, która ma być użyta, należy wykonać jawne rzutowanie na typy parametrów metody.
- Niejednoznaczności pojawiają się na etapie wywołania metod, a nie ich definicji.

Określanie wywoływanej metody - przykład



```
static void OptionalParameters(int number, string name, int marker = 0, string ident = "noValue")
{
    Console.WriteLine($"{number},{name},{marker},{ident}");
}

static void OptionalParameters(int nr, string name, int smt = 5)
{
    Console.WriteLine($"nr={nr},name={name},smt={smt}");
}

static void metodyPrzeciazzoneTest() {
{
    OptionalParameters(1, "Kaczyński", 2, "Nadprezydent");
    OptionalParameters(2, "Nowak", 1);
    //OptionalParameters(3, "Kowalski");
    OptionalParameters(number: 3, "Kowalski");
    OptionalParameters(nr: 3, "Kowalski");
}
}
```

```
1,Kaczyński,2,Nadprezydent
nr=2,name=Nowak,smt=1
3,Kowalski,0,noValue
nr=3,name=Kowalski,smt=5
```