**Lekcja 1: solution**

Rozwiązaniem „solution” jest to plik tekstowy z własnym, unikalnym formatem; generalnie nie jest przeznaczony do ręcznej edycji. Jest to plik leżący na najwyższym poziomie hierarchii rozwiązania Projekt jest zawarty w sensie logicznym i w systemie plików w ramach rozwiązania, które może zawierać jeden lub więcej projektów, wraz z informacjami o kompilacji, ustawieniami okna programu Visual Studio i wszelkimi różnymi plikami, które nie są skojarzone z żadnym projektem.

Rozwiązanie to obszar roboczy, który łączy w sobie wiele projektów, które zwykle są ze sobą powiązane. Konstrukcja jest bardzo przydatna w sytuacjach, gdy jakiś kod musi być współużytkowany przez wiele projektów.

Na przykład, jeśli tworzysz witrynę internetową z oddzielnym komponentem po stronie serwera, możesz podzielić go na kilka powiązanych projektów:

* Projekt zawierający wspólne interfejsy, tworzący bibliotekę DLL
* Projekt zawierający definicję bazy danych, tworzący skrypt wdrażania RDBMS
* Projekt zawierający implementację API po stronie serwera, tworzący plik wykonywalny
* Projekt witryny sieci Web tworzący skrypt wdrażania usług IIS

Te cztery projekty są ze sobą powiązane. Umieszczenie ich w jednym rozwiązaniu (solucji) umożliwia dostęp do poszczególnych części, a jednocześnie korzystanie z typowego indeksowania, które wykonuje program Visual Studio, aby uprościć wyszukiwanie pod kątem użycia i pomóc w refaktoryzacji.

Projekt jest samodzielnym dziełem, może to być aplikacja Win Forms, biblioteka klas, witryna internetowa, usługa sieciowa itp.

Jednak rozwiązanie problemu może wymagać działania wielu części, np. aplikacja Win Forms wymaga do działania Biblioteki klas, podobnie jak usługa sieciowa, ale witryna internetowa może wymagać usługi sieciowej.

Plik rozwiązania jest konceptualizowanym modelem tego. Jeśli Twoim problemem jest aplikacja Win Forms, utwórz rozwiązanie i dodaj aplikację Win Forms oraz bibliotekę.

Jeśli Twoim problemem jest witryna sieci Web, utwórz rozwiązanie i dodaj witrynę sieci Web, usługę sieci Web i bibliotekę.

Projekt zawierający bibliotekę może być taki sam w obu rozwiązaniach, co oznacza, że nie musisz powielać kodu ani śledzić zależności dla swoich projektów, ponieważ wszystko, od czego zależy, jest już w twoim rozwiązaniu.

Advantages -- Zalety

Rozwiązania pozwalają skoncentrować się na opracowywaniu i wdrażaniu projektów, zamiast przeglądać wszystkie szczegóły zarządzania plikami, komponentami i obiektami projektu. Każde rozwiązanie Visual Studio umożliwia:

* Pracuj nad wieloma projektami w tej samej instancji IDE.
* Pracuj nad elementami, korzystając z ustawień i opcji, które mają zastosowanie do całego zestawu projektów.
* Skorzystaj z Eksploratora rozwiązań, aby pomóc w opracowaniu i wdrożeniu aplikacji.
* Zarządzaj dodatkowymi plikami otwartymi poza kontekstem rozwiązania lub projektu.

**Lekcja 2: .NET Framework, .NET Core**

2002 r. Microsoft przedstawił pierwszą wersję platformy .NET (ang. .NET Framework) i dedykowanego jej języka C#. Było to przedsięwzięcie na miarę języka Javy i jej maszyny wirtualnej. Rozwijanej przez firmę Sun. Było dostępne tylko dla Windows. W obu platformach celem kompilacji nie był konkretny typ komputera i jego architektura ale wirtualne środowisko uruchomieniowe. W przypadku platformy .NET środowisko to nazywa się CLR (ang. *Common Language Runtime*).

Platforma .NET Core (rozwijana od 2016 roku) -wolna otwarta, rozwijana przez społeczność działającą pod kierunkiem Microsoftu, a jej kod źródłowy jest publicznie dostępny. Od wersji 3.0 .NET Core pozwala na tworzenie tzw. aplikacji desktopowych tzn. aplikacji z interfejsem użytkownika (okienkowych) z wykorzystaniem kontrolek WPF , jak i Windows Forms.

.NET Core różni się od .NET Framework tym, że pozwala na tworzenie oprogramowania nie tylko dla systemu Windows ale również dla Linuksa i macOS.

.NET CORE możemy C# uruchomić na wielu platformach:

* Windows,
* Linuks
* Mak
* Desktopowo,
* Na serwerze webowym
* Na urządzeniach mobilnych.

.NET obejmuje środowiska uruchomieniowe:

* NET Runtime pozwalające na uruchamianie aplikacji konsolowych
* ASP.NET CORE Runtime pozwalający na uruchamianie aplikacji Internetowych i serwerowych,
* Desktop Runtime pozwalające na uruchamianie aplikacji desktopowych

Możemy w C# pisać aplikacje na dowolną platformę np.:

* Pisać gry w Unity (system tworzenia gier w języku C#),
* Aplikacje webowe,
* Aplikacje desktopowe,
* Aplikacje mobilne (Xamarin platforma przygotowująca oprogramowanie mobilne dla różnych systemów )

Również możemy wykorzystać do tworzenia aplikacji hybrydowych: to aplikacja, która jednocześnie jest na komórkę i na Windows C#. Mając 1 kod napisany C# możemy go uruchamiać na kilku platformach.

.NET składa się z dwóch podstawowych komponentów:

* CoreCLR (pas startowy dla aplikacji napisanych C#)to środowisko uruchomieniowe CoreCLR, odpowiednik CLR w .NET Framework. Separuje on aplikacje od warstwy systemu operacyjnego, co daje kilka istotnych korzyści: pozwala chronić stabilność systemu, wprowadza mechanizmy, których nie posiada system operacyjny .NET, obejmuje mechanizm odpowiedzialny za zarządzanie pamięcią kompilator Jit (*just-in-time*) i proste typy danych i klasy niskiego poziomu.

Kod źródłowy C# nie jest kompilowany bezpośrednio do kodu maszynowego rozumianego przez procesor. Kompilacja jest dwustopniowa. W pierwszej fazie kompilator języka C# (nazywa się Roslyn) zmienia kod źródłowy języka C# w kod języka IL (intermediate language -język pośredni) -- kodu pośredniego, wspólnego dla wszystkich środowisk uruchomieniowych .Następnie zapisuje go w pliku zestawu (są to pliki DLL lub EXE). NET Core, bez względu na system operacyjny i procesor. Polecenia języka IL przypominają instrukcje języka assembler, które jednak wykonywane są w maszynie wirtualnej .NET o nazwie CoreCLR.

Druga faza to kompilacja kodu pośredniego przez CoreCLR za pomocą kompilatorów just-in-time (skrót(JIT). .NET przeprowadza kompilację drugą dopiero w momencie uruchomienia programu.

* CoreFX – to podstawowe biblioteki klas dla platformy .NET np. kolekcje, system plików, konsola, jeison, xml, asynchroniczność klient SQL, i wiele innych.

Zaletą dwustopniowej kompilacji jest to , że Microsoft może przygotować maszyny wirtualne CLR nie tylko dla systemów Windows, ale i dla Linuksa oraz systemu macOS. Ten sam kod IL może działać wszędzie, ponieważ dopiero drugi etap kompilacji generuje kod właściwy dla danego systemu operacyjnego i procesora.

**Lekcja 3: Programy najwyższego poziomu**

W C# 9 i nowszych wersjach języka istnieje coś, co jest nazywane programem najwyższego poziomu (ang. *top-level program*).

Kod programu najwyższego poziomu dla minimalnej aplikacji konsoli języka C# generuje kod Program.cs:



NET 5 i starszych wersji szablon aplikacji konsolowej generuje następujący kod:



Te dwie formy reprezentują ten sam program. Oba są prawidłowe w języku C#10.0. W przypadku korzystania z nowszej wersji wystarczy napisać treść Main. Kompilator generuje klasę Program z metodą punktu wejścia i umieszcza wszystkie instrukcje najwyższego poziomu w tej metodzie.

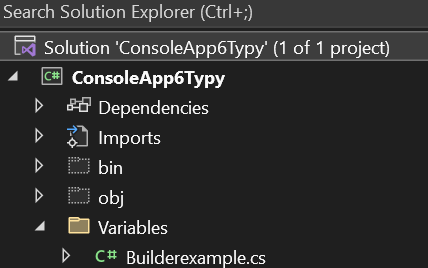
Dwie reguły dotyczące programów najwyższego poziomu:

* Ewentualne instrukcje using muszą znaleźć się na początku pliku;
* W projekcie może istnieć tylko jeden plik tego typu.

Zapisana na początku pliku instrukcja using System; nakazuje zaimportować przestrzeń nazw System. Dzięki temu możemy później skorzystać z instrukcji Console.WriteLine.

**Exercise 1** {niejawne importowanie przestrzeni nazw}

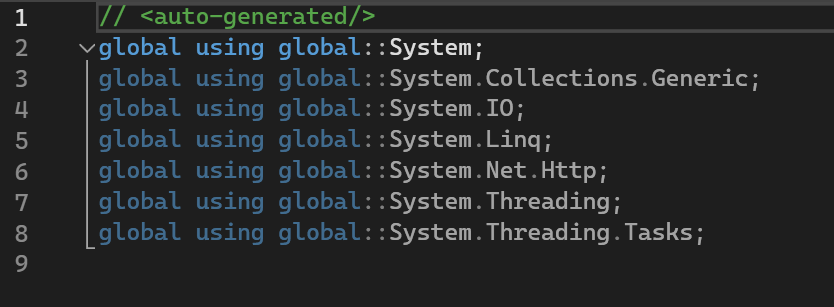
1. W okienku Eksplorator rozwiązań (Solution Explorer Ctrl+;) wybierz projekt ProgramNajwyższegoPoziomu (ConsoleApp6Typy) i kliknij przycisk Pokaż wszystkie pliki (Show All Files). Zwróć uwagę że tu też pojawią się foldery bin i obj



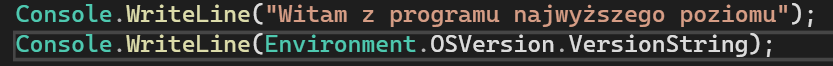
1. Rozwiń foldef obj, w nim rozwiń folder Debug, a potem folder net8.0 i otwórz plik o nazwie Program Najwyższego Poziomu.GlobalUsing.g.cs.



1. Ten plik jest automatycznie generowany przez kompilator na potrzeby projektów korzystających z .NET6 i używa funkcji nazywanej importem globalnym wprowadzonej w C# 10. Pozwala ona globalnie importować powszechnie używane przestrzenie nazw, takie jak System, do wszystkich plików z kodem.



1. W projekcie ProgramNajwyższegoPoziomu, w pliku Program.cs, zmień instrukcje tak, żeby wypisywałyinny komunikat oraz numer wersji systemu operacyjnego:





**Lekcja 4: Obsługa wiersza poleceń dotnet**

Korzystając z Interfejsu wiersza poleceń dotnet: Command line interface w skrócie CLI

wpisz polecenie

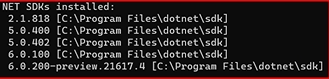


1. Mamy informację o wersji dotnet zainstalowane na komputerze, numerze kontroli wersji Commit.

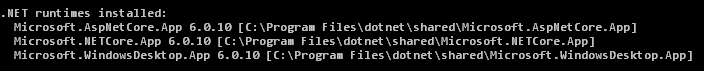
.NET SDK (Software Defelopment Kit) – zestaw bibliotek

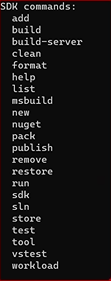


1. Wszystkich zainstalowanych wersjach SDK



1. Informacja o runtimes czyli wersjach frameworka dla których będziemy mogli uruchamiać aplikację



1. Wpisujemy polecenie  Są tu komędy które są w stanie wykonać SDK

Możemy tworzyć nowe projekty za pomocą polecenia

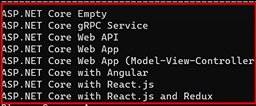
* **new** (Create a new .NET project or file).
* **build** będziemy mogli skompilować te projekty (Build a .NET project.).
* **run –** zbudujemy i uruchomimy gotowy projekt (Build and run a .NET project output)

1. Utworzenie nowego projektu:

* Dotnet new –help wyświetli listę dostępnych opcji które możemy wykonać za pomocą tego polecenia
* Wybieramy (l –list <nazwa>) czyli lista szablonów zawierających nazwę szablonu. Jeżeli nie użyjemy konkretnej nazwy zostaną wyświetlone wszystkie szablony)
* Polecenie dotnet new –list – będzie lista templates (szablonów)



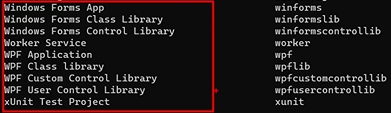
Aplikacje webowe dostępne na platformie ASP.NET



Aplikacja konsolowa



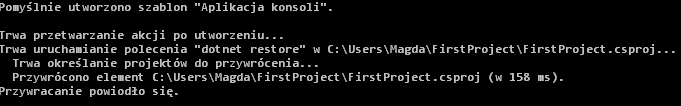
Pliki projektowe szablonów desctopowych oraz testowych

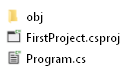


1. Utworzymy pierwszy projekt konsolowy:



Pomyślnie utworzyliśmy projekt konsolowy





W folderze FirstProject mamy plik Program.cs rozszerzenie C# plik z kodem źródłowym napisanym C#

Plik projektowy FirstProject.csproj z rozszerzeniem csproj każdy projekt taki plik będzie miał.

1. Żeby otworzyć projekt WS potrzebujemy pliku z rozszerzenie sln którego na razie nie mamy wpiszemy polecenie: dotnet new sln –name FirstProject



Znajduje się on w tym samym folderze co nasz FirstProject i jest to nasza solucja czyli rozwiązanie

1. Otwierając Visual Studio z prawej strony powinniśmy zobaczyć naszą przed chwilą stworzoną solucję. Musimy teraz nasz projekt umieścić w naszej solucji i możem zrobić to na dwa sposoby

* Uruchomić polecenie z wiersza poleceń dotnet sln add FirstProject.csproj korzystają z wiersza poleceń należy uważać na ścieżkę dostępu do tego pliku.FirstProject.csproj.
* Bozpośrednio w Visual Studio klikając prawym przyciskiem myszy na add-> Existing Project i z folderu FerstProject wybieramy plik FirstProject.csproj w solucji mamy projekt i jeden plik C# program z rozszerzeniem cs

1. Chcąc skompilować ten projekt klikamy myszą na projekt prawym przyciskiem myszy i pierwsze build lub po polsku kompiluj albo z wiersza poleceń używając polecenia dotnet build 
2. W katalogu FirstProject znajduje się po kompilacji katalog bin -> Debug -> net6.0 a wśrodku mamy skompilowane źródła naszego projektu. Interesować nas będzie plik , który zawiera skompilowany kod w formacie binarnym, plik nazwany jest asemblin Żeby go uruchomić możemy skorzystać również z wiersza poleceń CLI poleceniem zawierającym ścieżką do pliku .dll.



>dotnet FirstProject\bin\Debug\net6.0\FirstProject.dll

1. Automatyzuje zadanie budowanie I przerzucanie pakietów zewnętrznych a następnie od razu uruchomi nasz projekt 

*FirstProject>dotnet run*

**Lekcja 5: Strumienie wyjścia wejścia**

Statyczna metoda **Console.WriteLine**, którą używamy jest skrótem od metody **Console.Out.WriteLine**. Statyczna własność Out zwraca obiekt typu System.IO.TextWriter (klasa TextWriter w przestrzeni nazw System.IO), który umie m.in. wyświetlić napisy w konsoli.

**Console.Write()** to metoda która nie posiada znaku końca linii.

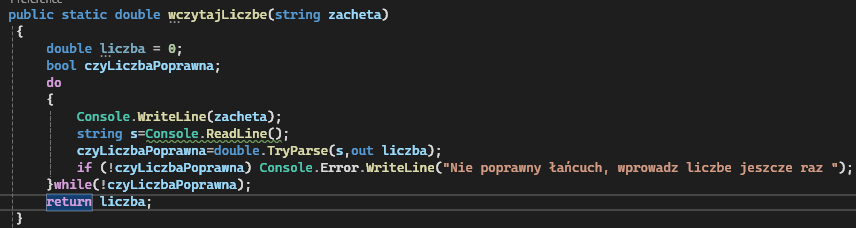
**Console.In.ReadLine()** – która pobiera tekst ze standardowego strumienia wejścia (reprezentowanego przez Console.In), czyli z klawiatury.

**Console.ReadLine()** czyli uruchamiamy bezpośrednio z klasy Console. Metoda ta czeka na wpisanie przez użytkownika ciągu znaków potwierdzonego naciśnięciem klawisza Enter. Zwraca łańcuch znaków i jeżeli chcemy uzyskać liczbę musimy uzyskaną wartość z parsować.



Konwertowanie do double metodą: **double.Parse** – ale jeżeli łańcuch jest niepoprawny, metoda double.Parse zwróci wyjątek, co spowoduje wyświetlenie komunikatu błędu i zakończenie programu bez dania użytkownikowi szansy na poprawienie własnego łańcucha.

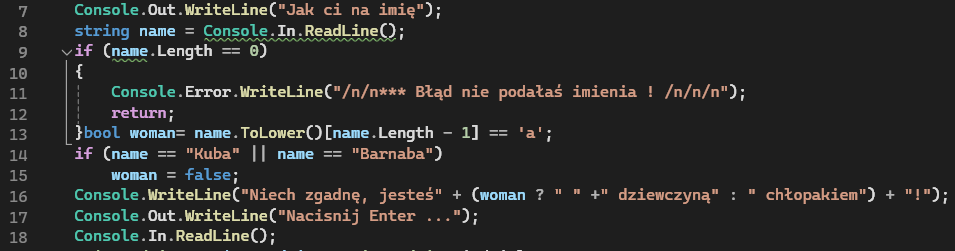
**double.TryParse** Ta metoda nie zgłasza wyjątków, a w zamian zwraca wartość logiczną mówiącą o tym czy podany łańcuch jest liczbą, czy dokładnie to co nam potrzebne. Dzięki temu możemy zrezygnować z konstrukcji try … catch. Skonwertowana liczba jest natomiast zwracana przez argument metody TryParse.



**Console.Read()** odczytujący kolejny znak ze strumienia wejściowego, wprowadzonego i potwierdzonego klawiszem Enter.

**Console.ReadKey()** zwraca naciśnięty w danej chwili klawisz (to nadaje się do tworzenia interaktywnych menu sterowanych klawiaturą.

**Console.Error()** to strumień błędów, działa analogicznie jak strumień wyjścia, drukując napisy w konsoli.



**Lekcja 6: Typy w C#**

1. Typy danych można podzielić na typy wartościowe (**ang**. *value typs*) do których zalicza się typy proste (inaczej podstawowe – **ang.** *primitive types, simple typs*), wyliczeniowe (**ang**. *enum types*) i strukturalne (**ang**. *struct types*), oraz typy odnośnikowe (referencyjne – **ang** *refrtence types*), do których należą typy klasowe, interfejsowe, delegacyjne, oraz tablicowe.
2. Typy proste można podzielić na arytmetyczne całkowite, arytmetyczne zmiennoprzecinkowe, typy char i bool.

Typy arytmetyczne całkowite to: sbyte, byte, short, ushort, int, uint, long, ulong.

Typy arytmetyczne zmiennoprzecinkowe: float, double, decimal.

**Typ char** służy do reprezentacji znaków przy czym w C# jest on 16-bitowy i zawiera znaki Unicode (Unicode to standard pozwalający na zapisanie znaków występujących w większości języków świata). Ponieważ Unicode to 16 -bitowa liczba, czasami zalicza się go do typów arytmetycznych całkowitych. Aby umieścić znak w kodzie, należy do ująć w znaki apostrofu np.: ‘a’

1. Zmienne możemy zadeklarować w dowolnym miejscu metod i klas. W klasach zmienne nazywają się polami.

Typ bool określa tylko dwie wartości logiczne true i false są używane przy konstruowaniu wyrażeń logicznych, porównywaniu danych, oraz wskazywaniu czy dana operacja zakończyła się sukcesem. W przypadku C# true i false nie mają przełożenia na wartości liczbowe czyli zmienna typu int która przechowuje np.: 0 nie można ustawić w wyrażeniu warunkowym.

*int zmienna=0;*

*if(zmienna){*

*//instrukcje Ta instrukcja w przypadku C# jest nie poprawna*

*//błąd zostanie zgłoszony już na etapie kompilacji*

*//gdyż nie istnieje domyślna konwersja z typu int na typ bool wymagany przez instrukcję if.*

*}*

**Typ wyliczeniowy** – jest określany słowem enum i pozwala na tworzenie wyliczeń – czyli określonego zbioru wartości, które będą mogły być przyjmowane przez dane tego typu. W najprostszym wypadku schemat utworzenia wyliczenia wygląda następująco:

*enum nazwa\_wyliczenia {element\_1, element\_2, element\_3, …elementN}*

Np.:

*enum Kolory {czerwony, zielony żółty}*

W rzeczywistości każde wyliczenie ma własny typ bazowy, a każdy element wyliczenia – ma wartość tego typu. Domyślnym typem bazowym jest int, a elementy są numerowane od 1. Zatem w powyższym przykładzie czerwony otrzymał wartość 1, zielony 2, a żółty 3.

Istnieje możliwość samodzielnego określenia zarówno typu bazowego, jak i wartości przypisanych poszczególnym elementom. W tym celu należy zastosować rozszerzoną definicję typu wyliczeniowego w postaci:

*enum nazwa\_typu: typ\_bazowy(element\_1=wartość=, element\_2=wartość, …, element\_N=wartoscN}*

Np.:

*enum Kolory:short{czerwony=10,zielony = 20, żółty = 30}*

taką instrukcję można też rozbić na kilka linii:

*enum Kolory:short*

*{*

*Czerwony=10,*

*Zielony=20,*

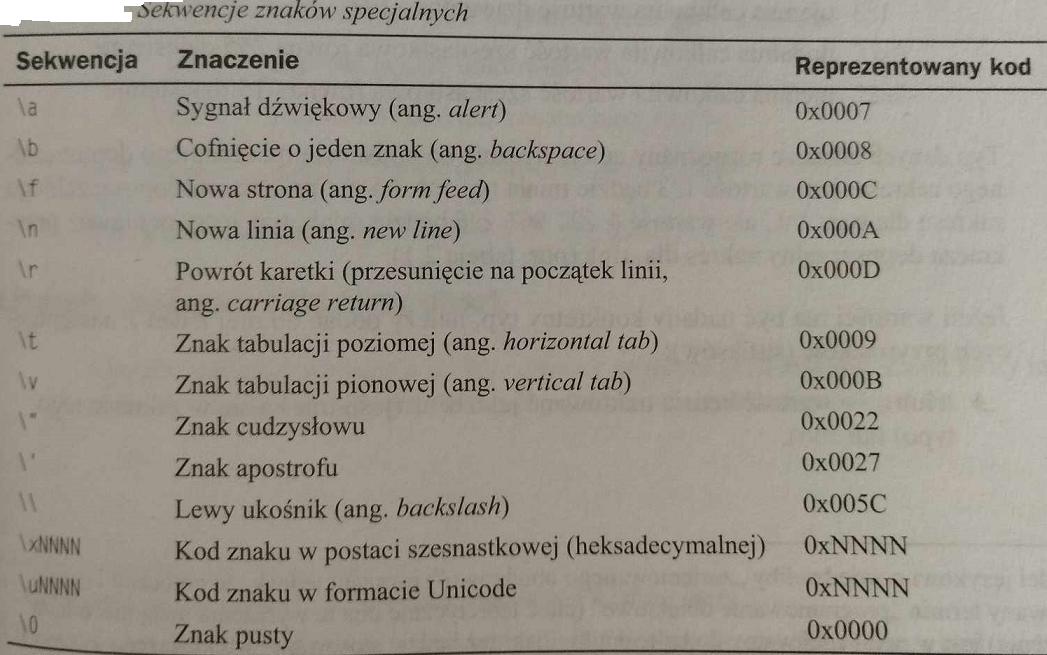
*Żółty=30*

*}*

Należy pamiętać, że typem bazowym może być taki, który reprezentuje wartości całkowite, czyli: byte, sbyte, short, ushort, int, uint, long, ulong.

**Typy strukturalne** – struktury definiowane są za pomocą słowa struct. Przypominają one klasy, choć posiadają ograniczenia

**Ciągi znaków** – do reprezentacji łańcuchów znakowych czyli napisów, służy typ string. Jest on zaliczany do typów prostych, w rzeczywistości jednak należałoby go jednak traktować jako typ referencyjny. Nie wnikając jednak w dyskusje teoretyczne, na początku nauki C# trzeba jedynie wiedzieć, że jeśli chcemy umieścić w programie łańcuch znaków, napis należy go ująć w cudzysłów.



**xTypy referencyjne** czyli obiektowe klasowe odnoszą się do programowania obiektowego

**Zapis wartości *(literały)***

Czyli stałe napisowe (**ang**. *string constant, literal constant*), to ciągi znaków reprezentujące w kodzie źródłowym programu jawne wartości. Na przykład ciąg znaków 12 jest literałem interpretowanym przez kompilator jako wartość całkowita dodatnia równa 12, zapisana w systemie dziesiętnym. Gdy napiszemy 120 to wartość jest 120 itp…

Literały całkowitoliczbowe reprezentują liczby całkowite, które mogą być poprzedzone znakiem plus (+) lub (-). Przykłady zapisu:

* 123 dodatnia całkowita wartość dziesiętna 123
* -123 ujemna całkowita wartość dziesiętna -123
* 0xFF dodatnia całkowita wartość szesnastkowa równa 255 dziesiętnie
* -0x0f ujemna całkowita wartość szesnastkowa równa – 15 dziesiętnie

Jeżeli wartości ma być nadany konkretny typ, należy dodać niej jeden z następujących przyrostków (sufiksów):

* U lub u – wartość będzie traktowana jako uint (jeśli mieści się w zakresie tego typu) lub long,
* L lub l – wartość będzie traktowana jako long lub ulong,
* UL lub ul będzie traktowana jako ulong.

Literały zmiennoprzecinkowe Literały rzeczywiste - są to ciągi cyfr zwracające separator dziesiętny (znak kropki) lub zapisane w notacji wykładniczej z literą E bądź e (patrz podane niżej przykłady). Mogą być poprzedzone znakiem plus.

Mogą być zapisywane w notacji wykładniczej, w postaci X.YeZ, gdzie X to część całkowita, Y część dziesiętna, natomiast Z wykładnik potęgi liczby 10 (można używać zarówno małej jak i wielkiej litery e). Zapis taki oznacza to samo co X.Y\*10z.

Przykłądy:

* 1.1 dodatnia wartość rzeczywista1,1
* -1.1 ujemna wartość rzeczywista -1,1
* 0.2e100 dodatnia wartość rzeczywista 20
* 0.1E2 dodatnia wartość rzeczywista10
* 2e-2 dodatnia wartość rzeczywista 0,02
* -3.4E-1 ujemna wartość rzeczywista -0,34

Wartość opisana w powyższy sposób otrzymuje standardowo typ double. Możliwe jest określenie konkretnego typu za pomocą jednego z sufiksów.

* F lub f – będzie taktowana jako float
* D lub d jako double,
* M lub m – jako decimal {przykładem takiego zapisu będzie: 1.2f czy też 22.54M

Literały znakowe char –‘a’

Literały logiczne – true, false

Literały łańcuchowe „Coś”

Literał null -- jest stosowany jako określenie wartości typu specjalnego null. Zapisywany jest jako ciąg znaków null

1. Zmienne lokalne deklarujemy w obrębie nawiasów klamrowych metod.
2. Deklarując zmienne musimy podać jej typ i nazwy oddzielone spacją:

*typ\_zmiennej nazwa\_zmiennej;*

int i; //typ całkowity czterobajtowy (32bity)

long l; //typ całkowity ośmiobajtowy (64bity)

string s; //łańcuchowy

float f;// typ rzeczywisty czterobajtowy 32 bity

double d;//typ rzeczywisty ośmiobajtowy (64 bity)

1. Wartości powinno się nadać jak najszybciej

int i=1;

long l=1L;

string s=”Helion”;

float f=1.0f;

literki na końcu niektórych wartości. Literał 1L, tak samo jak 1, to liczba o wartości równej jedności literał 1L jest typu long, 1.0f jest typu float, a 1.0 -typu double

1. Słowa kluczowe : int, long, string, float, double –nie są prawdziwymi nazwami typów to aliasy, np.: literał typu int 1 jest obiektem typu System.Int32 i mamy dostęp do metod tej struktury.

Czyli zapis:

*int i = 10;*

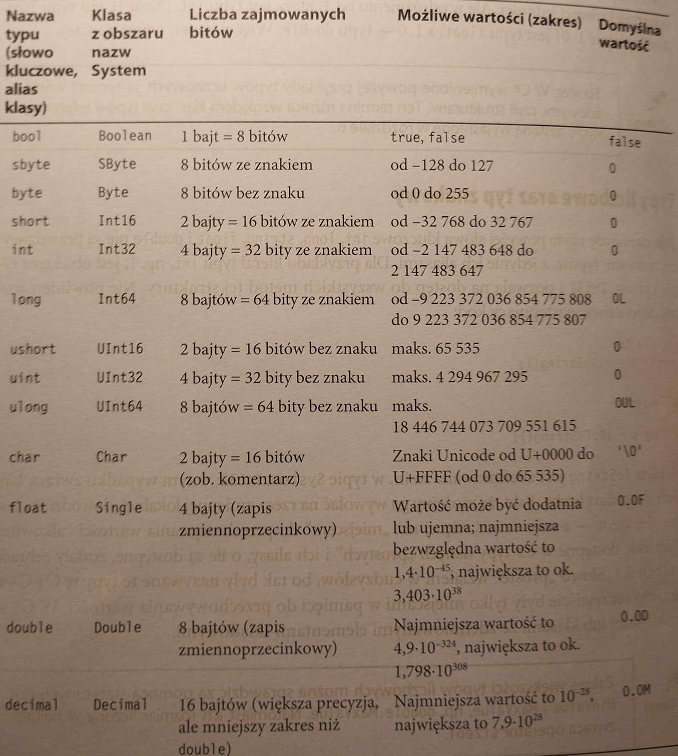
*string s = i.ToString();*

lub:

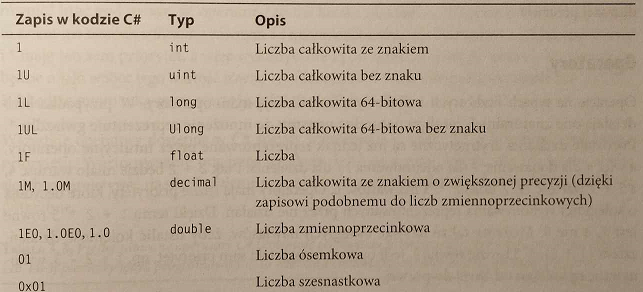
*string s=10.ToString();*

1. Metoda **ToString()** jest zdefiniowana w typie System.Int32; zwraca łańcuch z zapisaną liczbą. To że możemy ją wywołać na rzecz zmiennej lokalnej dowodzi że jest obiektem (i.ToString). Zmienne tzw: „proste” w C# są strukturami lub klasami ze zdefiniowanymi elementami składowymi.

Typy danych „prostych” dostępnych w C# i ich zakres



Stałe liczbowe w C#



1. Zakres większości typów liczbowych można sprawdzić za pomocą statycznych pól **MinValue**, **MaxValue**, np.: **double.MaxValue**. Natomiast ich rozmiar liczony w bajtach zwraca operator **sizeof**.
2. Wprowadzenie słowa kluczowego var pozwala definiować zmienne o typie ustalonym przez kompilator na podstawie wartości użytej do ich inicjalizacji. Inicjalizacja musi jednak nastąpić w tej samej linijce kodu co deklaracja. Po inicjalizacji respektują kontrolę typu. I jeżeli typ to int to nie można go już zmienić na łańcuch. Mogą to być jedynie zmienne lokalne.
3. **Słowo kluczowe var** można używać do definiowania zmiennych typów będących kolekcjami, typami parametrycznymi czy klasami i strukturami definiowanymi przez programistę a nawet kontrolek z bibliotek Forms lub WPF.
4. **Konwersja typów** czyli zmiana wartości jednego typu na inny. Używamy do tego operator rzutowania, tj. typ na który chcemy przeprowadzić konwersję, podany w nawiasach okrągłych:

*(typ)x;* //**Jawne** rzutowanie (konwersja) zmiennej x na typ wskazany w nawiasie.

Uwaga! (int)0,99=0

(double)1=1.0 odwrotnie (int)1.0=1

Poza jawną konwersją typów za pomocą operatora rzutowania w kodzie C# można stosować także konwersję niejawną tzn. taką, w której operator rzutowania nie jest jawnie obecny. Ze względów bezpieczeństwa jest ona jednak ograniczana. Niejawna konwersja z typu całkowitego na typ zmiennoprzecinkowy (generalnie do typu o większej precyzji) oraz z typu o mniejszym zakresie na typ o większym zakresie. Konwersja w drugą stronę, wiążę się z utratą informacji (precyzji), wymaga jawnego użycia operatora konwersji, a zatem świadomej decyzji programisty.

*double x=1;* //to jest ok

*Int i=0.1;* //pojawi się błąd

W przypadku operatorów dwuargumentowych, które są użyte dla argumentów różnych typów, wybierany jest typ o większej precyzji lub większym zakresie:

*1.0+1* //double

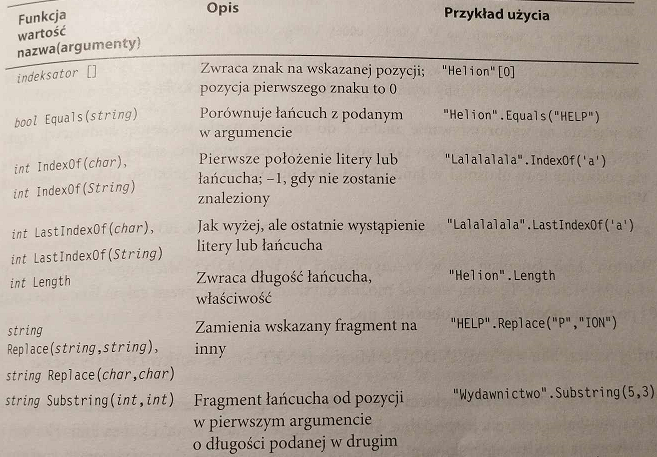
*1.0/2* //double wartość 0,5

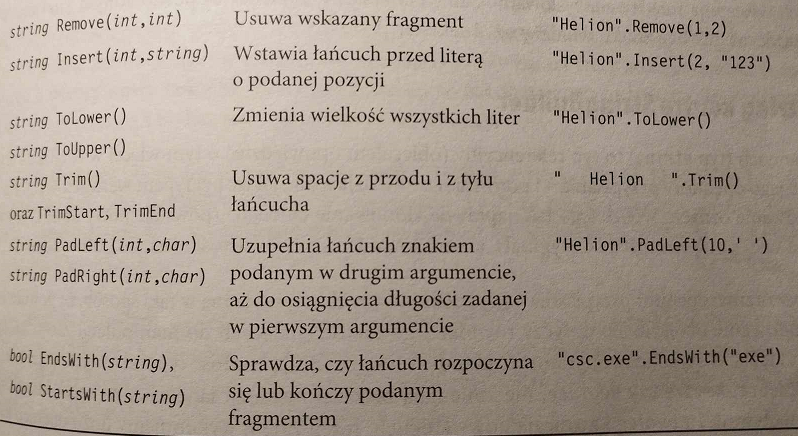
*1/2* //wartość 0

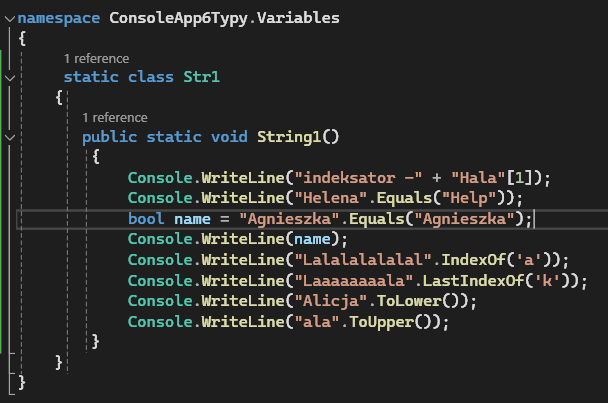
W przypadku dzielenia liczb całkowitych wynikiem będzie liczba całkowita zaokrąglona w kierunku zera.

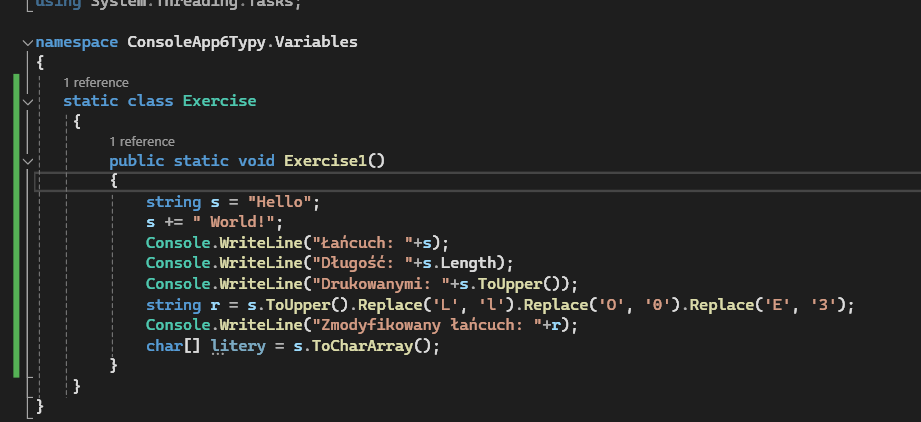
1. **Łańcuchy** są implementowane w klasie System.string (alis string) w której zostały zdefiniowane przydatne metody i właściwości. Pozwalają one na porównywanie łańcuchów

Najczęściej używane metody klasy String









**Exercise**

Zaplanuj sekwencje specjalne rozpoczynające się od lewego ukośnika (znaku backslash)\ np.

@ znak małpy na początku łańcucha ignoruje wszelkie sekwencje specjalne,

\n koniec linii

\”nie kończy łańcucha tylko wyświetla.

\b kasuje poprzedni znak

\u sekwencja definiujaca znaki Unicode (obejmuje 4 znaki alfanumeryczne),\u0048

Przykład:

*string helion = "Wydawnictwo\"\u0048\u0065\u006c\u0069\u006f\u006e";*

*Console.WriteLine(helion);*

1. **String a StringBuilder**.

String – typ referencyjny, operatory przypisania i porównania działają tak jakby był typem wartościowym. Kopiowanie oznacza klonowanie, porównanie dotyczy wartości (tekstu), a nie adresów.

Wszystkie operatory i metody, które zwracają wartość typu string, a więc m.in. +, Insert, Remove czy Replace, nie zmieniają bieżącej instancji łańcucha, na rzecz której, są wywoływane, a tworzą i zwracają nowy łańcuch. Problem pojawia się przy użyciu pętli kiedy kopiowania łańcucha jest wiele. W takich przypadkach powinniśmy używać zamiast klasy string to klasę **StringBilder**. (nie dogmatycznie).

*Console.WriteLine("Użycie klasy String");*

*string s = "abc---";*

*s += "def";*

*s = s.Replace("---", "ghi");*

*Console.WriteLine(s);//*

*Console.WriteLine("Użycie StringBilder'a....................................");*

*System.Text.StringBuilder sb = new System.Text.StringBuilder("abc---");*

*sb.Append("def");*

*sb.Replace("---","ghi");*

*Console.WriteLine(sb.ToString());*

*Console.WriteLine("efekt jest ten sam ale....");*

W przypadku typu string działanie operatora += oznacza utworzenie nowej zmiennej typu string, której adres (czyli referencja) jest zapisywany w tej samej referencji co pierwotny łańcuch. To oznacza kopiowanie. Zastępowana wartość łańcucha (ob. Typu referencyjnego) pozostaje bez referencji co oznacza, że zajmie się nim *garbage collector*. Analogicznie działa metoda Replace

Odpowiednikiem operatora += w klasie StringBuilder jest metoda Append. Dodaje do bieżącej zawartości łańcucha łańcuch wskazany w argumencie. Robi to bez kopiowania – modyfikuje bieżącą instancję obiektu. To samo Replace. Korzystanie z tej klasy jest wydajniejsze.

Jeżeli manipulacji łańcuchem jest do kilku czy kilkunastu to różnica czasu wykonania operacji w przypadku łańcucha typu string i StringBuilder jest niezauważalna.

Przykład:

*static class BExample*

*{*

*public static string Example(int count)*

*{*

*StringBuilder sb = new StringBuilder();*

*for (int i= 0; i < count; ++i)*

*{*

*sb.Append('f');*

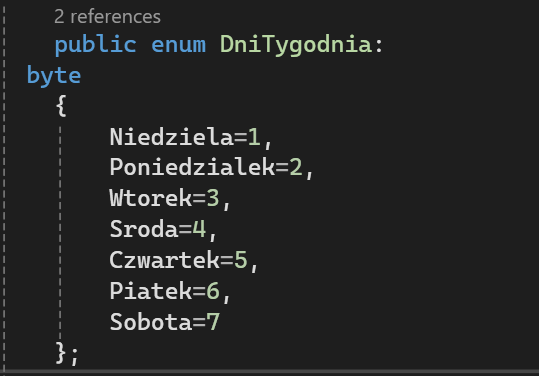
*}*

*return sb.ToString();*

*}*

*}*

1. **Formatowanie łańcuchów**: przykład sformatowania zdania używając metod Console.WriteLine i Console.Write:
2. Typ wyliczeniowy -jest stosowany do definiowania grupy powiązanych ze sobą stałych. Może zostać zdefiniowany jedynie w przestrzeni nazw lub jako pole klasy – jego definicja nie może natomiast znaleźć się wewnątrz metody. Typ wyliczeniowy jest implementowany przez klasę System. Enum, do jego definiowania będziemy używać słowa kluczowego enum. Przykład:



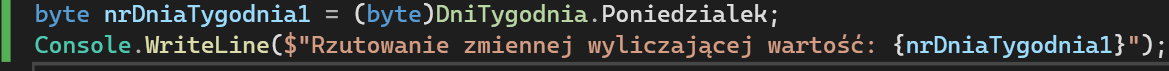


Domyślna wartość pierwszego elementu jest ustalana na 0, ale w powyższym przykładzie zamiast domyślnej wartości pierwszego elementu użyta została wartość 1. Jawne podane są również wartości wszystkich elementów. Jeśli ich nie ma, każdy kolejny element otrzymuje wartość o jeden większą: W tym przypadku pominięcie jawnych wartości niczego nie zmieni.

*Public enum DniTygodnia:*

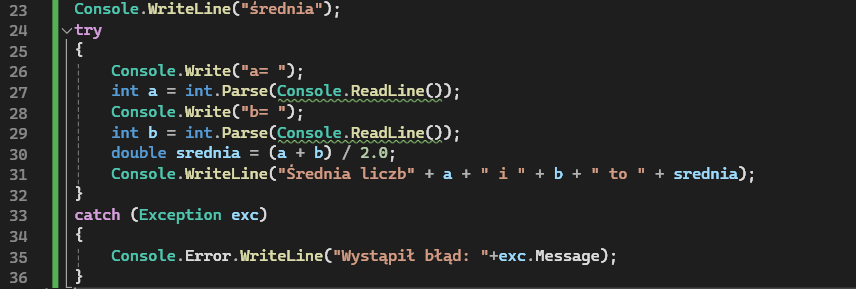
*Byte {Niedziela=1, Poniedziałek, Wtorek, Środa, Czwartek, Piątek, Sobota};*

Typ wyliczeniowy jest zawsze związany z jakimś typem liczbowym całkowitym. W przykładzie powyższym jest byte (System.Byte). Zaskakujące jest to, że wyrażenie DniTygodnia.Poniedziałek nie jest wcale typu byte a typu DniTygodnia, czyli zdefiniowanego przez nas przed chwilą typu wyliczeniowego. Każdy element może być w każdej chwili zostać zrzutowany na typ byte, np.:

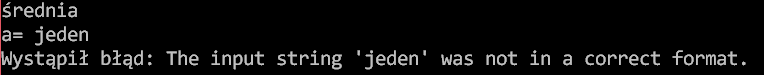


Zmienna typu DniTygodnia może przyjmować wartości zawarte w definicji typu wyliczeniowego. Definiowanie tego typu jest w istocie rozszerzeniem klasy System.Enum. W tworzonej w ten sposób klasie potomnej definiowane są publiczne pola statyczne typu zadeklarowanego w definicji. Takie dziedziczenie nie może być wykonane jawnie ponieważ System.Enum jest klasą specjalną.

1. **Obsługa błędów**: za pomocą konstrukcji **try .. catch** to kluczowy element programowania nowoczesnych języków. Należy spodziewać się niespodziewanych sytuacji i przygotować na nie program. To dotyczy zarówno interakcji z użytkownikiem jak i napisania przez programistę kodu. Wszelkie pomyłki np.: związane z wpisaniem liczby to ryzyko błędów które należy obsłużyć.



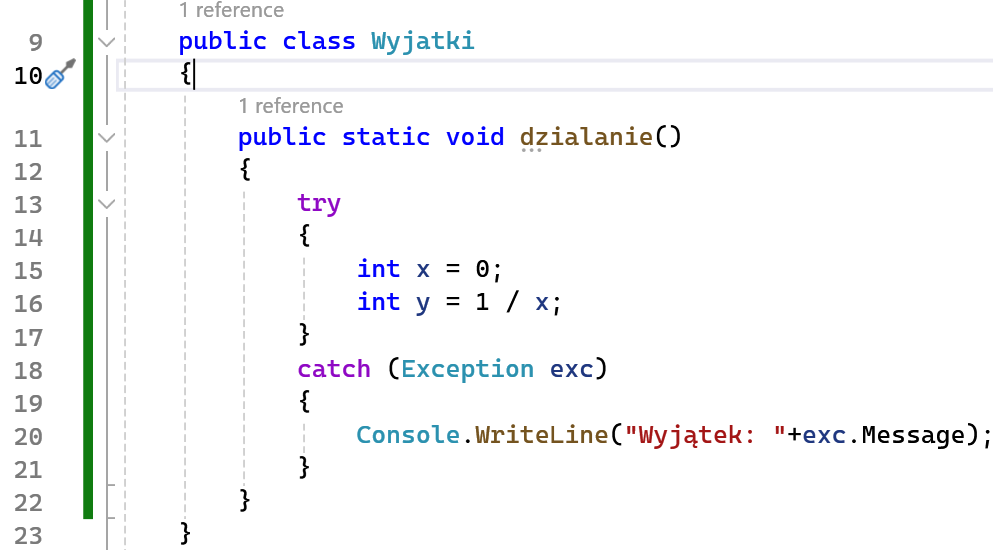
Błąd źle zapisanej wartości ma postać wyjątku (ang. exception), czyli obiektu przechowującego informacje o błędzie. Ten obiekt można przechowywać, żeby już w trakcie działania programu dowiedzieć się co spowodowało błąd i ewentualnie spróbować jakoś temu zaradzić lub chociaż wyświetlić własny komunikat, który pokaże, że panujemy nad sytuacją. Nieobsłużony błąd przez aplikację zostanie wychwycony przez platformę .NET lub .NETCore, która wyświetli komunikat błędu.



**Exceptions – wyjątki.**

Wyjątki do sygnalizowania błędów. W C# przez wyjątek można rozumieć obiekt, który przenosi wiadomość o błędzie z miejsca jego wystąpienia tzn. z miejsca, w którym wyjątek jest zgłaszany, do miejsca jego obsługi. Obszar poddany obserwacji jest otoczony blokiem try. Ewentualny wyjątek, który może tam powstać, zostanie przekazany do sekcji catch wykonywanej jedynie w takim przypadku. Po przejściu do sekcji catch nie ma możliwośći powrotu do sekcji try, pozostałe polecenia nie zostaną wykonane. Po wykonaniu poleceń w sekcji catch wątek przejdzie do poleceń znajdujących się po konstrukcji try … catch.

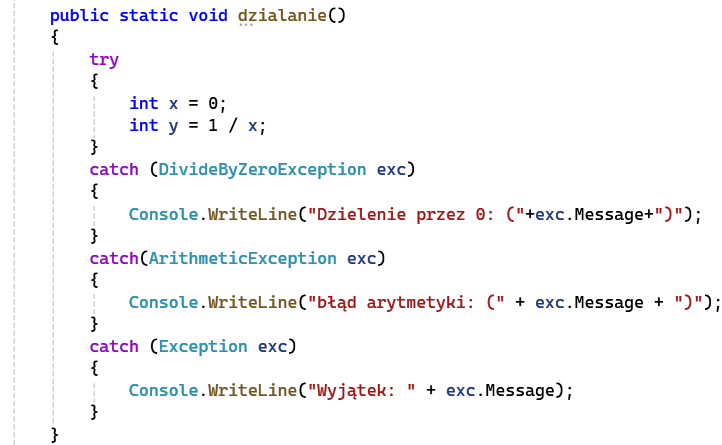
Przykład metody z blokiem try: „dzielenie przez 0:





Po sekcji try może znajdować się kilka sekcji catch wychwytujących poszczególne typy wyjątków. Poniżej ten mechanizm:

Przykład wielokrotnej sekcji catch przechwytującej coraz bardziej ogólne klasy wyjątków:



Sekcje catch powinny być ułożone od wychwytującej najbardziej szczegółową klasę wyjątku do najbardziej ogólnej, tj. do klasy wielokrotnie dziedziczącej po Exception do samego Exception. W przeciwnym razie, jeżeli na pierwszym miejscu ustawimy sekcję catch przechwytującą sekcję catch Exception , to ze względu na to, że mogą być na ten typ niejawne rzutowania wszystkie inne klasy wyjątków (jako na ich klasę bazową) automatycznie wychwytywałaby wszystkie wyjątki nie pozostawiając nic dla kolejnych sekcji catch.

|  |
| --- |
| W aplikacjach „okienkowych” Windows Forms wystąpienie nieobsłużonego wyjątku nie oznacza zakończenia działania aplikacji. Zakończy się jednak działanie metody, w której on wystąpił. |

Składnia konstrukcji obsługującej wyjątek może być rozszerzona o sekcję finally. Polecenia z tej sekcji wykonywane są w każdym przypadku, bez względu na to czy wyjątek wystąpił, czy nie. Wykonywane są także wtedy, gdy w sekcji catch znajdzie się instrukcja return lub inne polecenie kończące działanie metody. W razie wystąpienia wyjątku polecenia z tej sekcji wykonywane są po poleceniach z sekcji catch. Typowym wykorzystaniem sekcji finally jest zwalnianie zasobów, np.: zamknięcie pliku otwartego w sekcji try.

Przykład wykorzystania sekcji finally.

{

try

{

int x = 0;

int y = 1 / x;

}

catch (DivideByZeroException exc)

{

Console.WriteLine("Dzielenie przez 0: ("+exc.Message+")");

}

catch(ArithmeticException exc)

{

Console.WriteLine("błąd arytmetyki: (" + exc.Message + ")");

}

catch (Exception exc)

{

Console.WriteLine("Wyjątek: " + exc.Message);

return;

}

finally

{

Console.WriteLine("Kod wykonywany w każdym przypadku");

}

}

**Lekcja 7:Metody**

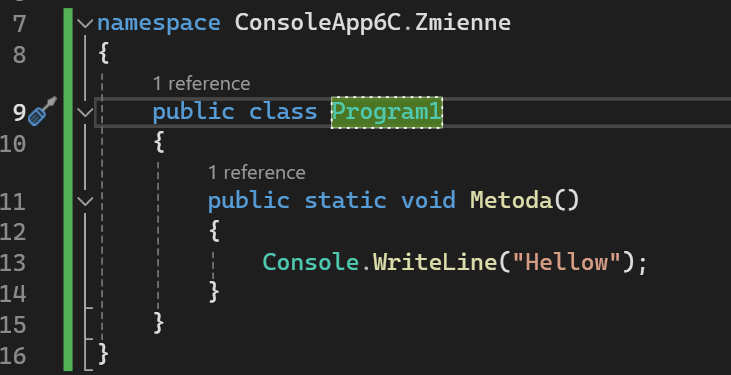
W C# nie jest możliwe definiowanie funkcji niebędących metodami jakiejś klasy.

Metody statyczne – to takie, które można wywoływać bez tworzenia instancji klasy, w której są zdefiniowane. Do definicji funkcji statycznych dodajemy modyfikator static. Funkcja main jest statyczna.

**Exercise 1**

Wywołać w funkcji main statyczną metodę metoda i wyświetlić napis Hellow.

Klasa.metoda()



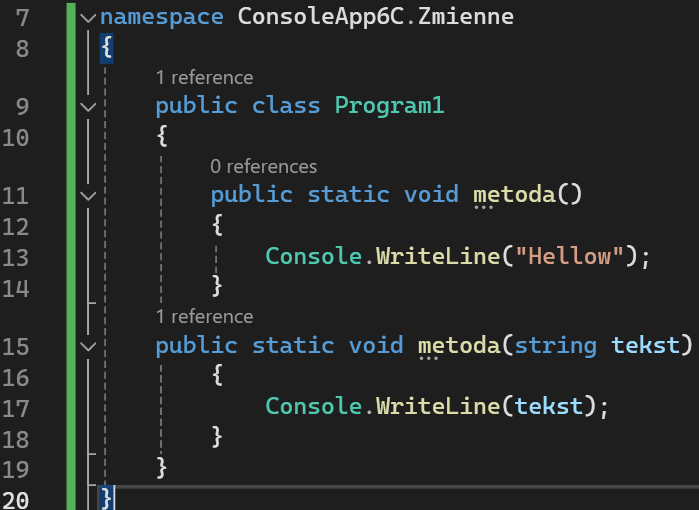
Każda metoda posiada głowę metody, typ zwracanej wartości, jej nazwę, listę parametrów formalnych (które przyjmuje). Ta część definicji nazywana głową polecenia w {} instrukcje metody nazywamy ciałem. Powyższe metoda nie przyjmuje żadnych argumentów ponieważ miejsca pomiędzy nawiasami() jest puste, nawiasy są obowiązkowe(). Elementy znajdujące się w głowie: nazwa metody i jej parametry, składają się na sygnaturę metody, po której jest ona rozpoznawalna wśród innych metod klasy.

Exercise1,a

Zdefiniuj do powyższego przykładu drugą metodę o tej samej nazwie różniącą się obecnością parametru. Będzie nim łańcuch określający treść wyświetlonego komunikatu.

Wywołanie w funkcji main



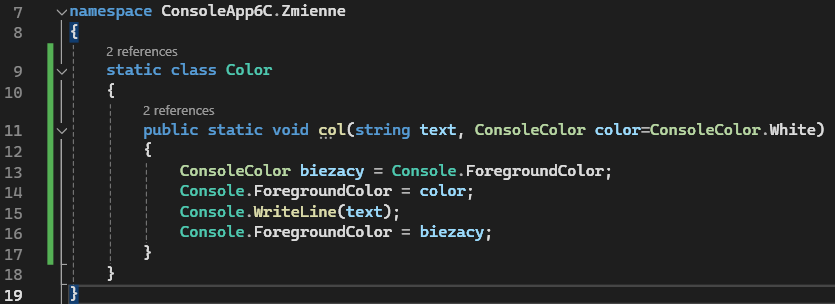


Język C# umożliwia definiowanie wielu metod o tych samych nazwach pod warunkiem, że różnią się parametrami (dzięki temu mają również inne sygnatury). Nazywa się to **przeciążeniem** metody(ang. *overload*). Niemożliwe jest natomiast definiowanie dwóch metod różniących się jedynie zwracanymi wartościami.

|  |
| --- |
| W formalnym języku informatyki rozróżnia się dwa pojęcia, - „parametry” i „argumenty metod”. Parametry (lub inaczej parametry formalne) należy postrzegać jako zmienne zdefiniowane w sygnaturze metody czyli w liście argumentów (tam wskazywane są typy i nazwy poszczególnych argumentów. W momencie wywołania metody poznajemy wartości parametrów formalnych, które kojarzone są z parametrami aktualnymi wystawionymi przy wywołaniu nazwy metody. Wartość zmiennej wartościowej lub referencja do obiektu zmiennej referencyjnej. Parametry formalne otrzymują z klonowane wartości lub referencje czyli adresy do zmiennych obiektowych . Te pierwsze znikają po wyjściu z metody (widoczne są tylko lokalnie natomiast referencje powodują zmianę pod adresem czyli w zmiennej będącej źródłem. |

**Exercise 2**

Napisz metodę z domyślną wartością argumentu. Pierwszy argument typu string wartość podana w argumencie aktualnym druga wartość domyślna np.: zmiana koloru. Wywołaj metodę dwuargumentową i jednoargumentową.

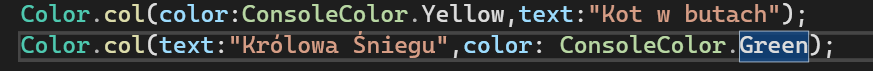


Wywołanie:



W powyższym przykładzie mamy dwie wersje metody w jednej definicji. Należy tylko pamiętać, że parametry z domyślną wartością (inaczej argumenty opcjonalne) muszą być zawsze definiowane jako ostatnie w liście parametrów.

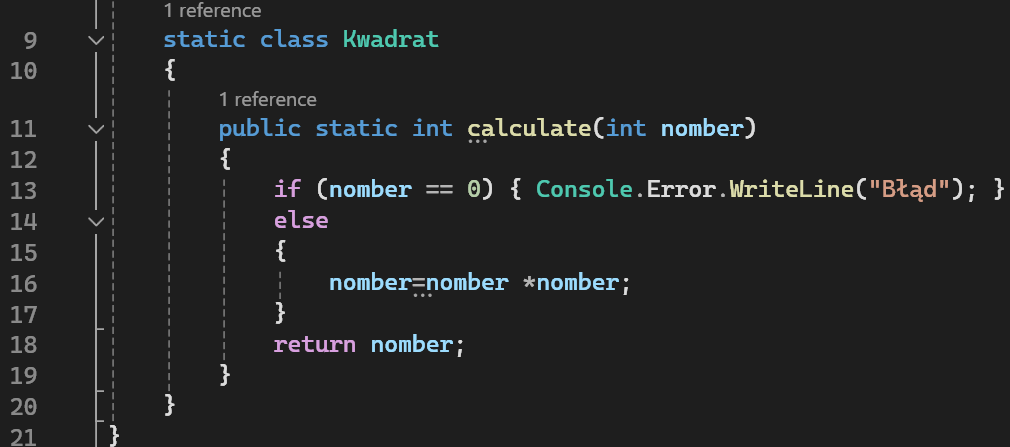
**Argumenty Nazwane**: W C# możliwe jest wskazanie parametru metody nie tylko na podstawie kolejności ale również jego nazwy, np.



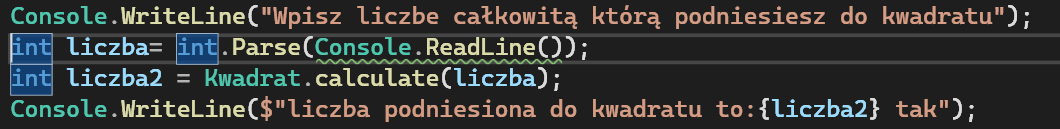
Sprawdź znaczenie kolejnych parametrów, korzystając z IntelliSense (klawisz skrótu Ctrl+Shift+ spacja).

**Exercise 3**

Metoda oblicza kwadrat liczby podanej w argumencie. Użyj return i zwróć wynik do main



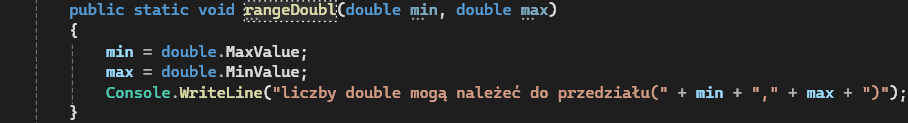
Funkcja main



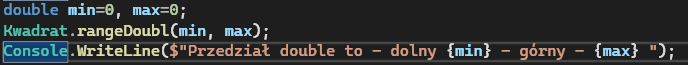
**Exercise 4**

Metoda oblicza nie jedną a kilka wartości np. dwa pierwiastki równania kwadratowego. Wiele rozwiązań a teraz sposób zwracania wartości przez argumenty. Argumenty typu referencyjnego – zmiany są widoczne w zmiennej po wyjściu z funkcji. Zmienne wartościujące czyli utworzone kopie giną przy wyjściu z metody, W C# jest możliwe potraktowanie parametru typu wartościującego (np. liczbowej) jak referencji.

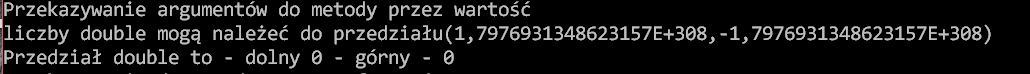
4.a. Przekazywanie argumentu do metody przez wartości.



Wywołanie w Programy.cs

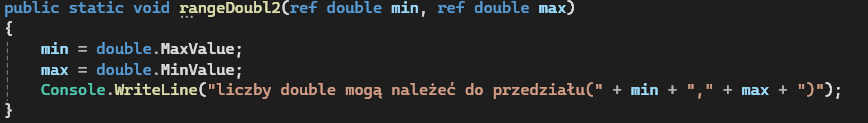


Wynik:

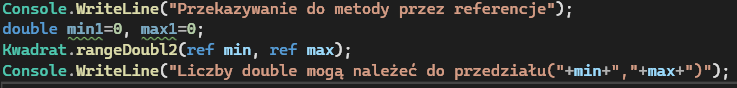


**4.b Przekazywanie argumentów do metody przez referencje**

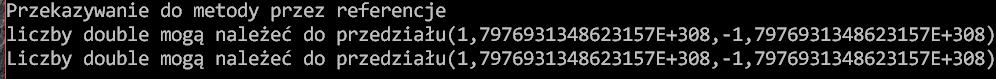
Metoda ref.



Wywołanie Program.cs



Wynik:

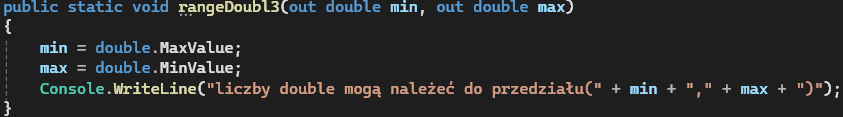


Wyświetlenia z wnętrza metody i z programy sc są takie same wartości są zapisane na adresie zmiennej.

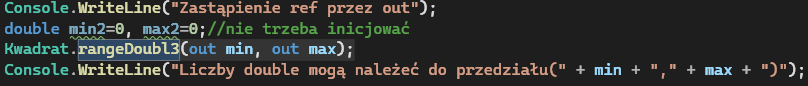
**4.c Przekazywanie argumentów do metody , zastąpienie ref przez out**

Oprócz słowa kluczowego ref można użyć w tym samym kontekście słowa kluczowego **out**. Różnica polega na tym, że out **wymusza zmianę wartości**, a ref tylko ją dopuszcza. Zmienna oznaczona przez **ref** przed przesłaniem do metody **musi zostać zainicjowana**, a oznaczona przez **out – nie**.

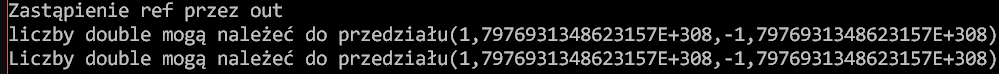
Metoda „out”



Wywołanie Program.cs



Wyniki



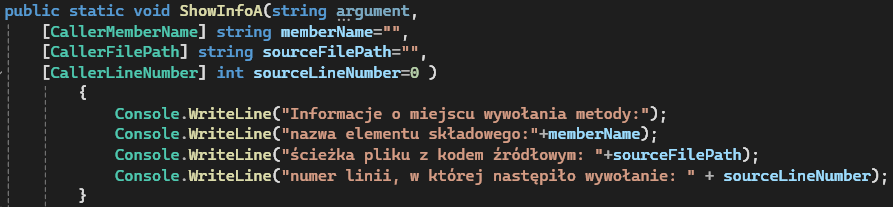
**Caller Information**

W C# można w metodzie uzyskać informację o miejscu, z którego nastąpiło jej wywołanie: nazwie metody wywołującej, ścieżce do pliku, w którym owa metoda jest zdefiniowana, i numerze linii w pliku, w której znajduje się instrukcja wywołana. Takie informacje mogą być przydatne przy raportowaniu błędów. Aplikacji w tzw. logach co jest użyteczne zwłaszcza w przypadku aplikacji działających bez bezpośredniego nadzoru użytkownika.

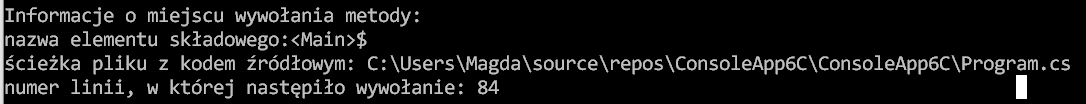
Program główny:



Metoda:



Wynik:



Trzy specjalne argumenty zostały zdefiniowane dzięki zadeklarowaniu przestrzeni nazw:

*using System.Runtime.CompilerServices;*

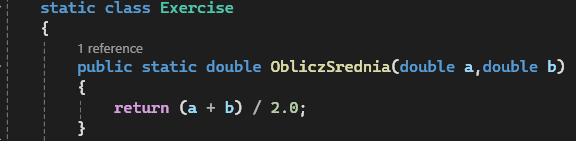
**Exercise 7**

Zdefiniuj metodę, która oblicza średnią arytmetyczną dwóch podanych w argumencie liczb typu double i zwraca jej wartość.

Program główny:



Metoda:



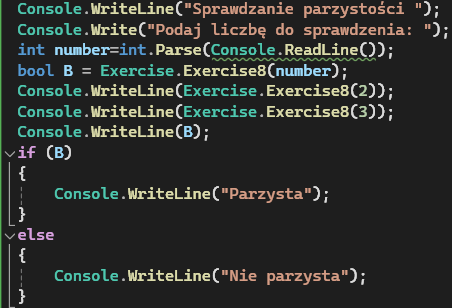
Wynik:



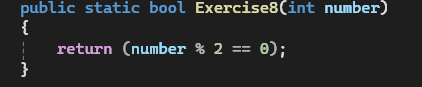
**Exercise 8**

Napisz metodę sprawdzającą parzystość liczb całkowitych podanej przez parametr, korzystając z dwuargumentowego operatora % zwracającego resztę z dzielenia. Metoda powinna zwracać wartość logiczną bool.

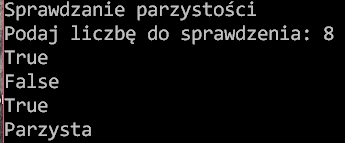
Program główny:



Metoda:



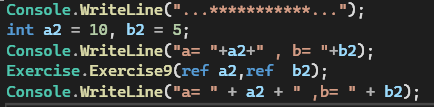
Wynik:



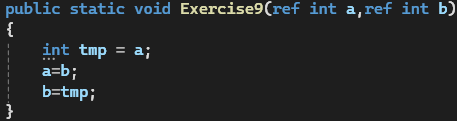
**Exercise 9**

Napisz metodę, która pobiera dwie wartości i zamienia je miejscami

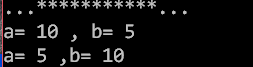
Program główny:



Metoda:



Wyniki:



**Exercise 10**

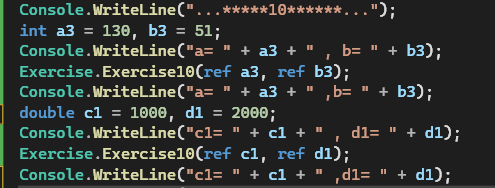
Trzeba uogólnić powyższą metodę tak aby mogła zmieniać wartości dwóch zmiennych dowolnego typu. Użyj metody ogólnej (*generic methods*).

Definiując taką metodę podajemy nazwę <typ>(zbiór parametrów formalnych)

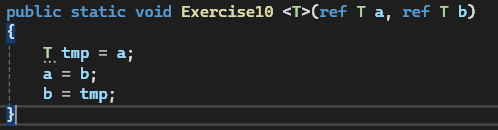
*nazwaMetody<T>(argumenty)*

Symbolu T używamy : definiując parametry, zwracaną wartość, w ciele. Symbol będzie określony w momencie wywołania.

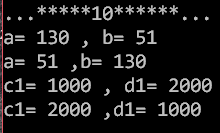
Program główny:



Metoda:

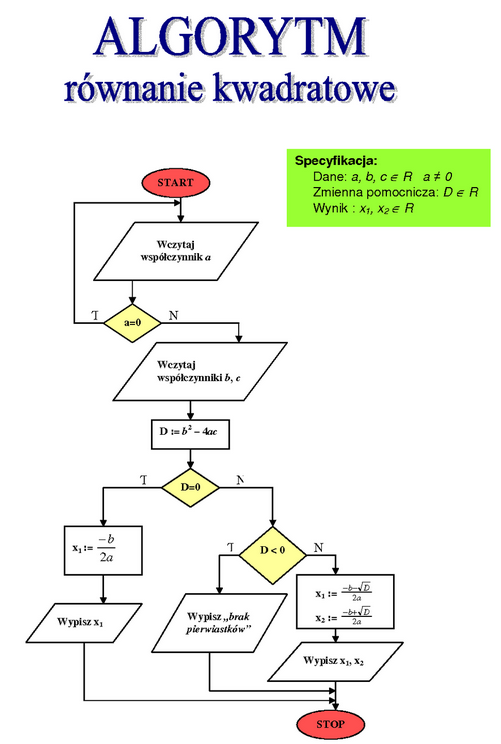


Wynik:



**Exercise 11**

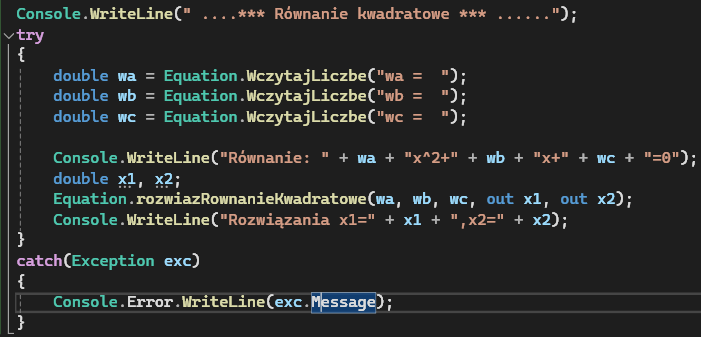
Równanie kwadratowe:



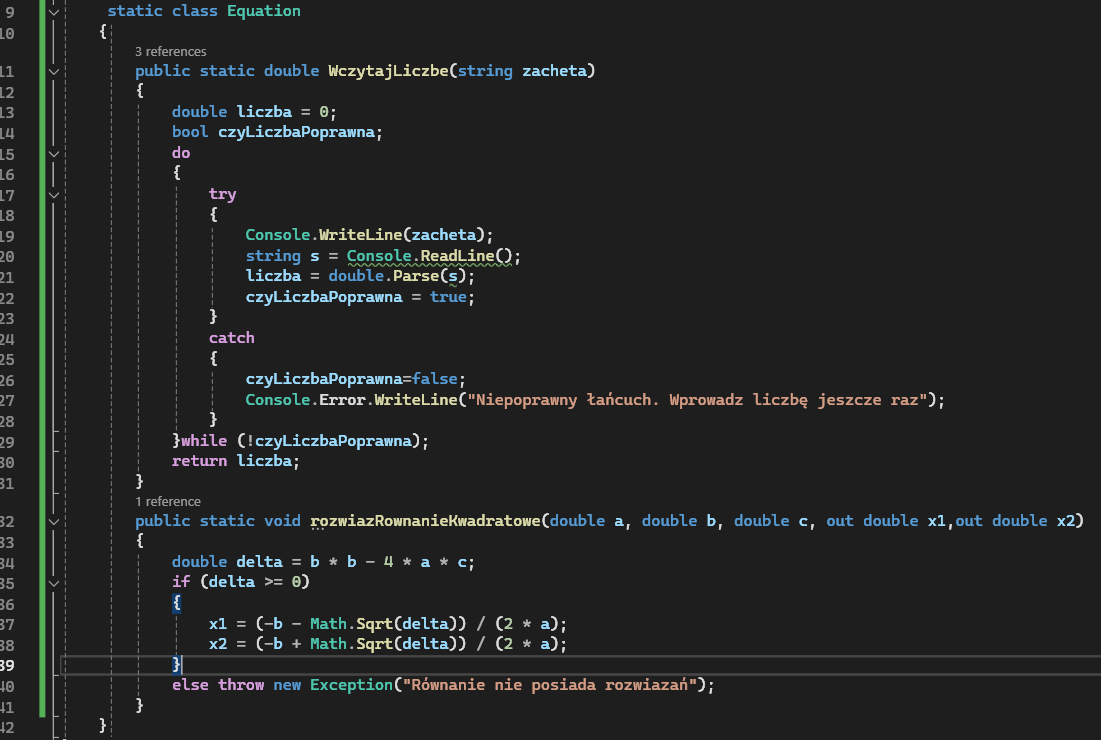
Lista kroków:

1. Wczytaj wartość współczynnika a, jeżeli jest to 0 to wyświetl komunikat nie jest to równanie kwadratowe i ponów prośbę o ponowne wpisanie. W przeciwnym przypadku wczytaj wartość współczynników b i c.
2. W tym przypadku jeżeli a≠0, przypisz delta=b2-4ac.
3. Jeżeli delta<0, wypisz komunikat: „równanie nie ma pierwiastków” i zakończ algorytm.
4. Jeżeli delta>0 to, jeżeli delta=0 oblicz pierwiastek x=-b/2\*a, wypisz wartość x i zakończ algorytm.
5. Jeżeli delta >0, przypisz **pom=c/a**
6. Jeżeli delta>0 i b>0 to oblicz pierwiastki:, x2=pom/x1
7. w przeciwnym przypadku jeżeli b≤0 to oblicz: x2= i x1=pom/x2 ,
8. Wypisz wartości wyznaczonych pierwiastków x1 i x2

Program główny:



Metoda:



**Lekcja 7: Sterowanie przepływem -instrukcje sterujące.**

**Instrukcja warunkowa if … else**

Składnia instrukcji skróconej:

*if (warunek)*

*instrukcja;*

Jest ona wykonywana wtedy kiedy warunek ma wartość *true*.

*if (warunek)*

*instrukcja;*

*else*

*alternatywna instrukcja;*

Alternatywna instrukcja jest wykonywana, gdy warunek nie jest spełniony a ściślej mówiąc wyrażenie ma wartość false.

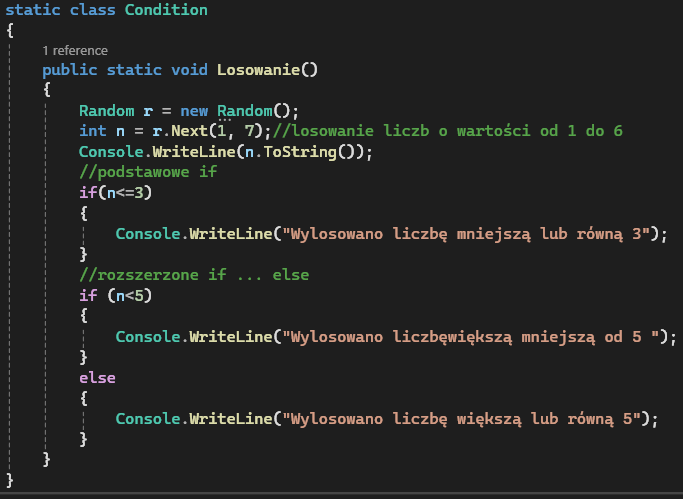
**Exercise 1**

Przykład na skróconą i klasyczną instrukcję warunkową.

Program główny:

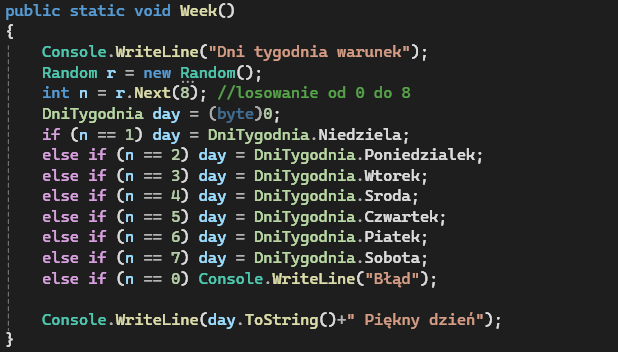


Metoda:

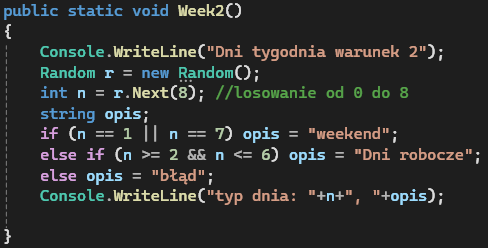


**Exercise 2.**

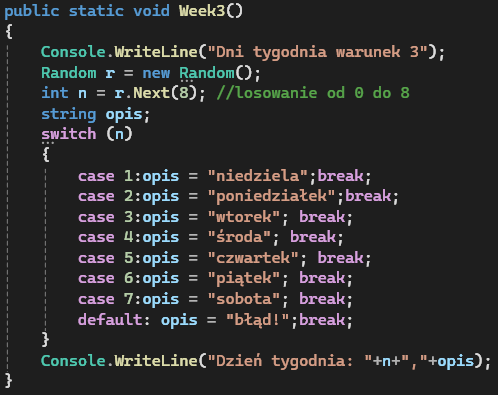
Przykład na instrukcję if … else …if sprawdzania wielu warunków (instrukcja kaskadowa).



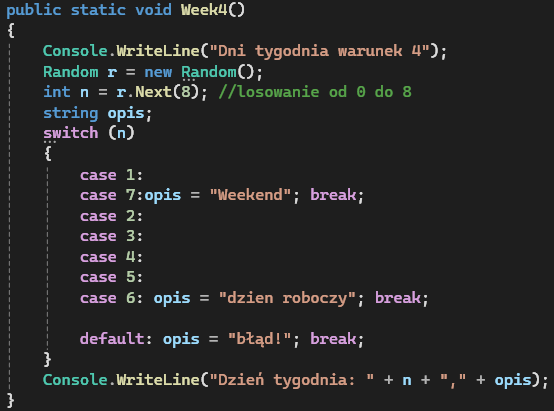
To samo zadanie gdzie użycie if else jest wygodniejsze od switcha



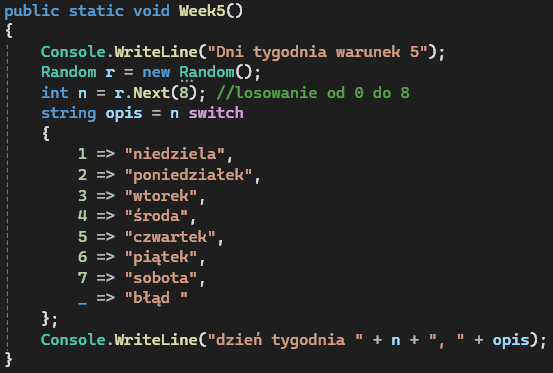
Wykorzystanie switcha



Nie efektywna ale poprawna , wykorzystanie instrukcji if



Nowa składnia switch operator =>



**Lekcja 8:Powtarzanie - rodzaje pętli**

Pętla for – struktura zapisu.

***for****(inicjacja\_indeksu; warunek; inkrementacja)*

*instrukcje;*

Typowy zapis:

***for*** *(int i=0; i< ilość; i++)*

*instrukcja;*

Pętla nieskończona;

*for (;for;)*

*instrukcja;*

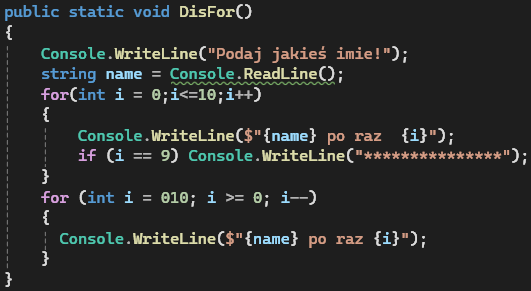
Exercise 1

Użyj pętli for do wyświetlenia imienia poraz n:

Funkcja główna:



Funkcja:



Exercise 2

Oblicz silnie iteracyjnie używając wzoru:

0!=1

1!=1

2!=1\*2=2

3!=1\*2\*3=6

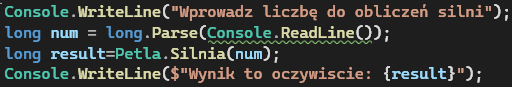
4!=1\*2\*3\*4=24

5!=1\*2\*3\*4\*5=120

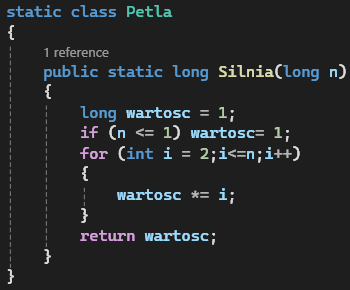
…

N!=1\*2\*3\*..\*(n-1)\*1

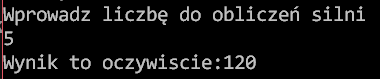
Program główny:



Funkcja licząca:



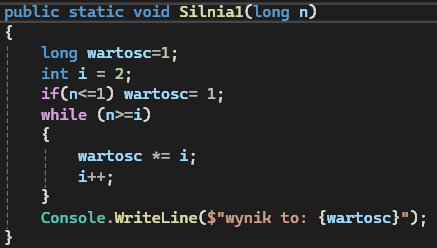
Wynik:



**Pętla while (wyrażenie)**

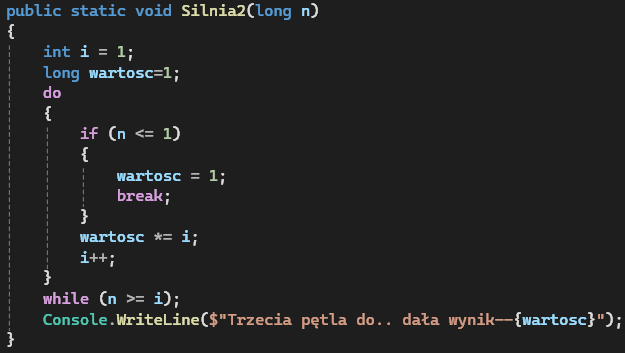
Obliczanie silni iteracyjnie przy pomocy pętli while. Warunek wykonuje się na początku imoże być tak że instrukcje nie wykonają się ani razu (przy warunku fałszywym ) Pętla chodzi przy warunkcu prawdziwym.

Wywołanie: Petla.Silnia1(num);



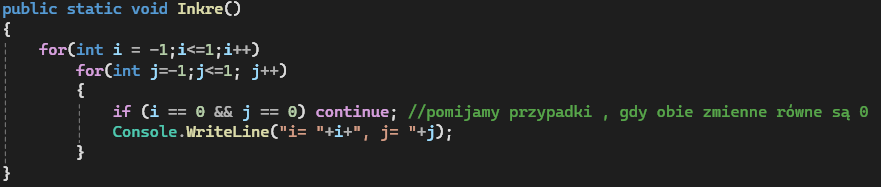
**Pętla do .. while(wyrażenie)**

Wywołanie: Petla.Silnia2(num);



Różnica między pętlami while i do ..while polega na tym, że w pierwszej warunek sprawdzany jest już przed wykonaniem instrukcji, a w do .. while dopiero po jej wykonaniu. Z tego wynika że stosując pętlę do ..while, mamy gwarancję, iż instrukcja zostanie wykonana przynajmniej raz nawet jeśli warunek niegdy nie zostanie spełniony.

**Przerywanie pętli**: Instrukcje wykonywane w każdej iteracji pętli mogą zostać przerwane za pomocą polecenia break i continue. Słowo kluczowe break całkowicie przerywa działanie pętli i przechodzi do instrukcji znajdującej się bezpośrednio za nią, continue przerywa tylko działanie bieżącej iteracji i rozpoczyna następną. W przypadku pętli for oznacza to zwiększenie indeksu zgodnie z poleceniem inkrementacji.



W tym przykładzie zostanie pokazanych tylko 8 komunikatów , przypadek gdy obie zmienne i j =0 zostanie pominięty. Gdybyśmy zamiast continue wstawili break, pominięte byłyby iteracje pętli wewnętrznej następujące po iteracji z zerowymi indeksami, ale pętla zewnętrzna byłaby kontynuowana.

**Lekcja 9: Wyjątki -przechwytywanie, filtrowanie, zgłaszane**

**Wyjątki:** w C# wyjątek to obiekt klasy Exception lub potomnej, który przenosi wiadomość o błędzie z miejsca jego wystąpienia, tzn. z miejsca, w którym jest zgłaszany, do miejsca jego obsługi.

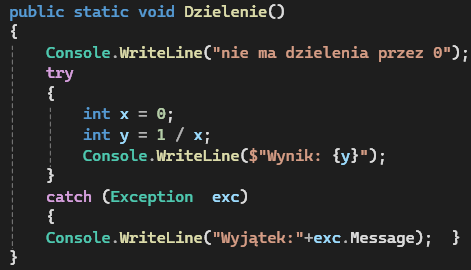
**Przechwytywanie wyjątków**: Obszar gdzie wyjątki są obserwowane występuje po słowie try i wyjątek który może tam powstać jest przenoszony do sekcji catch, wykonywanej jedynie w takim przypadku. Po przejściu do sekcji catch nie ma powrotu do sekcji try; pozostała część znajdujących się tam poleceń, nie zostanie wykonana, a po wykonaniu poleceń w sekcji catch wątek przejdzie do poleceń znajdujących się po całej konstrukcji try .. catch. To oznacza, że w sekcji try należy umieścić nie tylko te ryzykowne instrukcje, których się obawiamy, ale również instrukcje, które zależą od ich wyników – dzięki temu nie będą wykonywane, jeżeli tych wyników nie uda się uzyskać.

**Exercise 3**

Przykład konstrukcji try … catch przechwytujące wyjątek dzielenia przez 0.

Wywołanie: Exception1.Dzielenie();

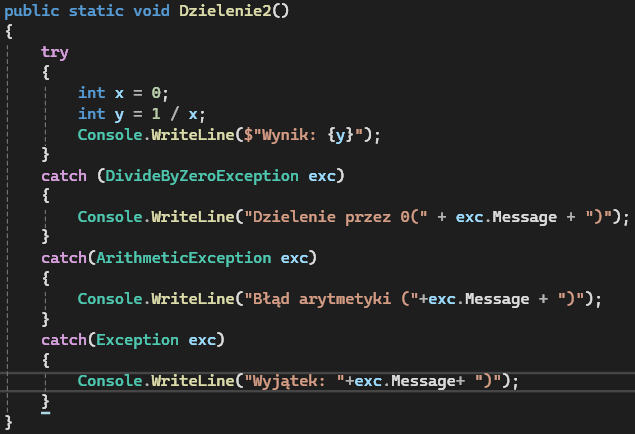
Metoda w pliku Exception1:



**Filtrowanie wyjątków** – po sekcji **try** może znajdować się kilka sekcji **catch** wychwytujących poszczególne typy wyjątków – czyli różnych klas dziedziczących po klasie Exception i w różny sposób na nie reagujących . Nazywa się to filtrowanie wyjątków

Wywołanie metody: Exception1.Dzielenie2();

Wywołana metoda:

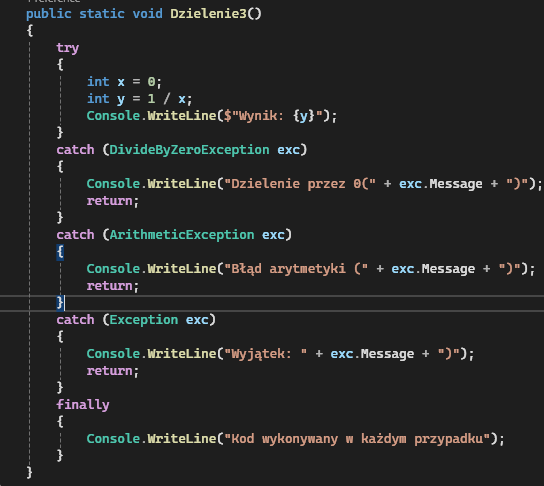


Sekcje catch powinny być ułożone od wychwytującej najbardziej szczegółową klasę wyjątku do najbardziej ogólnej, tj od klasy wielokrotnie dziedziczącej po Exception do samego Exception.

W przeciwnym razie gdybyśmy ustawili sekcję catch przechwytującą wyjątki typu Exception, to ze względu na to, że na ten typ mogą być niejawnie rzutowane wszystkie inne klasy wyjątków (jako na ich klasę bazową). Automatyczne wychwytywałaby ona wszystko nie pozostawiając dla innych.

**Sekcja finally**

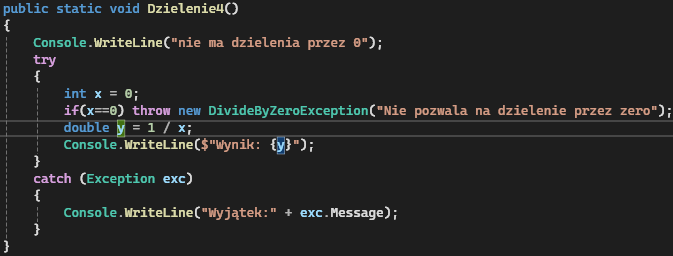
Składnia konstrukcji obsługującej wyjątek może być rozszerzona o sekcję finally. Polecenia z tej sekcji są wykonywane w każdym przypadku, bez względu na to, czy wyjątek wystąpiłby czy nie i jaki on by był. Polecenie finally wykonywane są nawet jeżeli w catch zamieszczono return (kończąca działanie metody). Polecenia z tej sekcji są wykonywane po poleceniach catch. Typowym wykorzystaniem sekcji finally jest zwalnianie zasobów np.: zamknięcie pliku otwartego w sekcji try.



Zgłaszanie wyjątków.

Służy do tego słowo *throw* z następującą po nim referencją do nowego obiektu wyjątku – tworzonego operatorem new. Kiedy wykryjemy krytyczny błąd programu i będziemy chcieli go zasygnalizować zgłoszeniem wyjątku, należy zatem w kodzie umieścić instrukcję analogiczną do poniższej:

*throw new Exception(„Komunikat opisujący błąd”);*

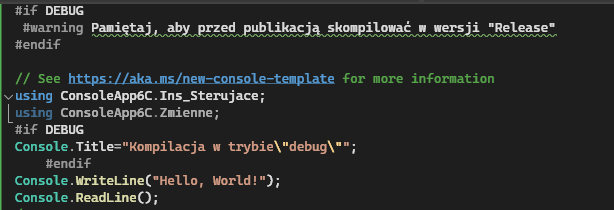


**Lekcja 10: Dyrektywy preprocesora**

Termin “dyrektywy preprocesora może być mylący dla osób znających C++, a uczących się dopiero C#. W odróżnieniu od C++ w C# nie ma osobnego etapu preprocesora podczas kompilacji kodu. Tzw. dyrektywy preprocesora są zatem analizowane równocześnie z analizą kodu C#. W efekcie w C# nie można używać dyrektyw do tworzenia makr; ich głównym zadaniem jest warunkowa kompilacja kodu.

Kompilacja warunkowa 94 strona

Przykład – wykorzystania dyrektywy do generowania ostrzeżenia, które przypomina, aby przed rozpowszechnieniem programu skompilować go z opcjami usuwającymi informacje przeznaczone dla debuggera. Poniższe dyrektywy można umieścić np.: na samym początku pliku Program.cs



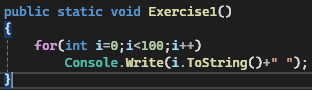
Przy każdym uruchomieniu programu tytuł okna konsoli zostanie zmieniony tak, żeby informować o tym, że działająca aplikacja została skompilowana w trybie debugowania. Tak się będzie działo nie dlatego, że wartość DEBUG jest sprawdzana za każdym razem. Została ona sprawdzona już podczas kompilacji i jeżeli warunek dyrektywy #if został spełniony, powyższa instrukcja została na stałe umieszczona w kodzie pośrednim.

**Exercise 1** pętle itp.

Wyświetl 100 liczb całkowitych od 0 do 99 oddzielonych spacjami w jednym długim wierszu, który w konsoli zostanie zawinięty do kilku wierszy.

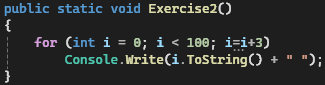
Wywołanie: Petla.Exercise1();

Metoda



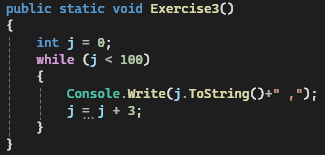
**Exercise2**

Wyświetl co trzecią liczbę z zakresu od 0 do 99:



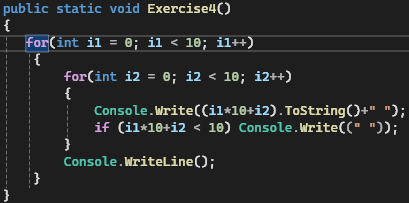
**Exercise3**

Chodź to mniej wygodne uzyskaj ten sam rezultat przy pomocy pętli while:



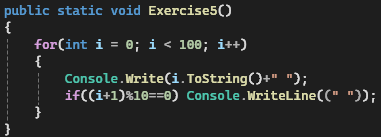
**Exercise 4.**

Wyświetlimy liczby całkowite, ułożone w dziesięciu wierszach po dziesięć liczb.

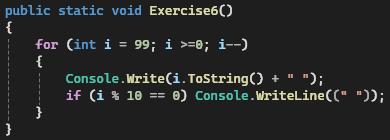


**Exercise 5**

Rozpoznanie końca dziesiątki sprawdzając czy reszta z dzielenia następnej wartości indeksu przez 10 jest równa zeru.



To samo zadanie z indeksem malejącym

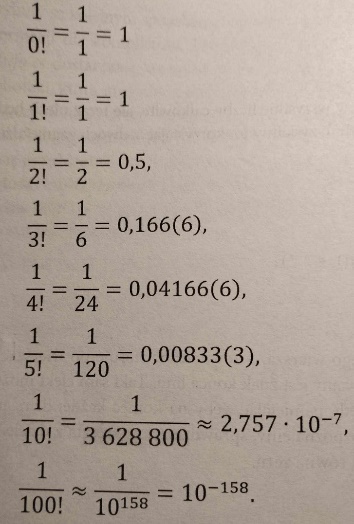


**Exercise 6 Liczba Eulera**

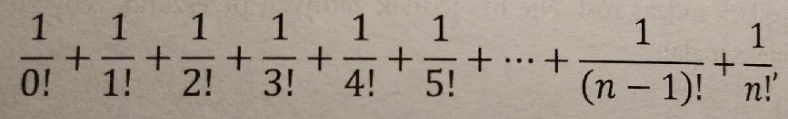
Silnia

*n!=1\*2\*3\*..\*(n-1)\*n*

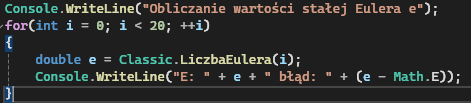
To funkcja która bardzo szybko zwiększa wartość wraz ze wzrostem argumentu. Silnia z 5 to 120, z 10 to 3 628800, a ze m100 już prawie 10158. To liczba która przekracza zakres typu long. Odwrotność silni będą wobec tego równie szybko malały do zera:



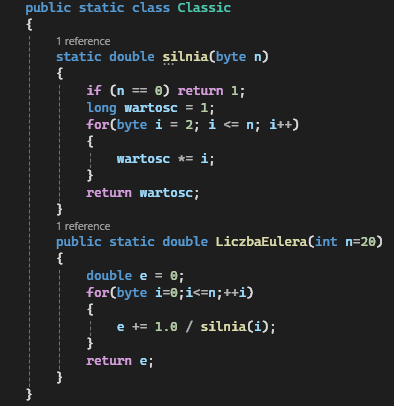
To oznacza, że jeżeli chcielibyśmy zsumować kolejne odwrotności silni, a więc obliczyć sumę szeregu to przyrosty od kolejnych takich ułamków będą coraz mniejsze i w ten sposób będą coraz mniej wpływać na wartość sumy. Już dziesiąty wyraz będzie zmieniał tylko siódmą liczbę po przecinku.



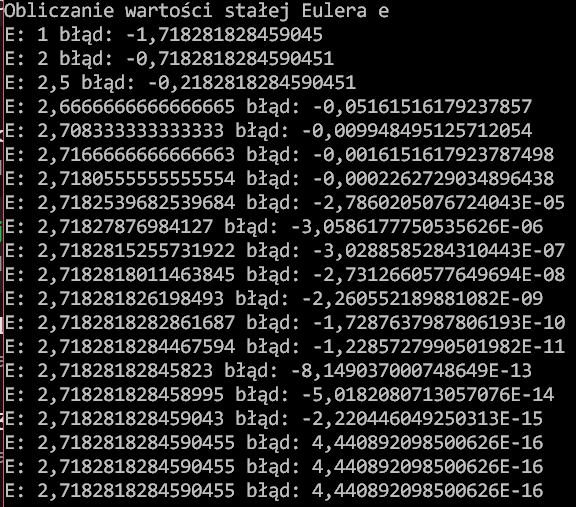
Wywołanie metody LiczbaEulera():



Metody:



Wyniki:



Metoda LiczbaEulera zwraca wartość równą w przybliżeniu 2,718281828459. To ważna stała matematyczna, nazywana liczbą Eulera. Stałą tą zwykle oznacza się literą e. Możemy ją znaleźć także wśród stałych zdefiniowanych w klasie Match we własności Math.E. To pozwoli nam sprawdzić, z jaką dokładnością my obliczamy wartość liczby e. Kolejna pętla przy wywołaniu – pętla for, zmienia liczbę sumowanych wyrazów szeregu i wyświetlając dodatkowo różnicę wyników naszych obliczeń i stałej zdefiniowanej w klasie Math.

Okazuje się że już dla kilkunastu zsumowanych wyrazów obliczona przez nas wartość liczby e stabilizuje się i kolejne wyrazy jej nie zmieniają. Obliczenia są dokładne do szesnastego miejsca po przecinku co wiąże się z dokładnością liczb typu double.

Program działa poprawnie. Każdy wyraz sumy wymaga obliczenia silni z kolejnych liczb całkowitych, co robimy korzystając z pętli for w metodzie silnia. Możemy ograniczyć liczbę działań matematycznych przez wykorzystanie faktu, że silnię dla kolejnej liczby można łatwo obliczyć, dysponując silnią dla liczby o jeden mniejszej np.:

4!=1\*2\*3\*4=(1\*2\*3)\*4=3!\*4

Zależność tę możemy zapisać rekurencyjnie:

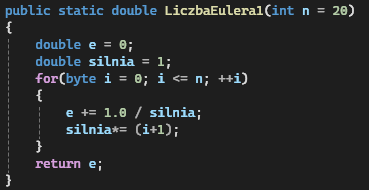
n!=(n-1)!\*n

Aby obliczyć kolejne wyrazy sumy, nie musimy osobno obliczać silni w każdym kroku. Wystarczy, że będziemy przechowywali jej wartość i w każdym kroku mnożyli ją przez kolejną liczbę całkowitą (indeks pętli)

Wywołanie metody:



Metoda licząca silnię rekurencyjnie potrzebną do obliczenia liczby Eulera:



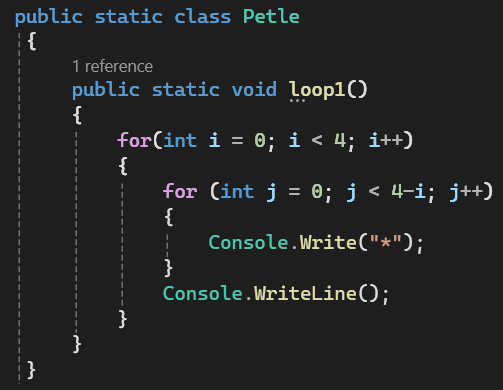
Wynik: 

**Zadania**

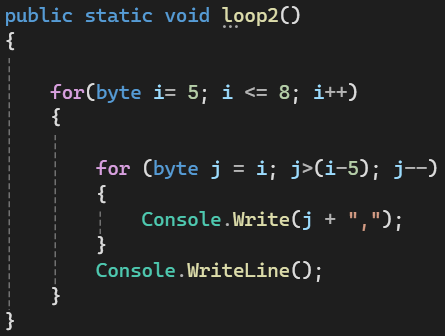
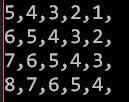
**Exercise 7**

Korzystając z wielokrotnej pętli for i instrukcji Console.Write, wyświetl w konsoli poniższe cztery wzory.

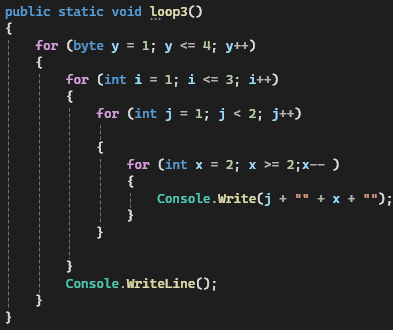
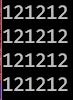
1.

2.

3.

4

**Lekcja 11. Typy wartościowe i referencyjne**

Obiektowość w C# jest wiodącym i konsekwentnym typem, nie ma typów prostych, jak w C++, wszystkie zadeklarowane przez nas zmienne są w rzeczywistości złożonymi obiektami. Nie wszystkie obiekty są instancjami klas w C# mamy do dyspozycji również struktury. Ważne jest różnica w użyciu. Klasy związane są z ze zmiennymi referencyjnymi, a struktury ze zmiennymi wartościowymi

W przypadku typów referencyjnych, a takimi są np. klasy wyjątków, obiekt będący instancją (egzemplarzem) klasy może powstać wyłącznie na stercie w wyniku użycia operatora new.

W przypadku typów wartościowych obiekty będące instancjami struktur mogą powstać wyłącznie na stosie. Programista nie ma tu żadnej dowolności. Zmienne wartościowe mogą być wprawdzie inicjowane za pomocą operatora new.

*int i=new int();*

co upodabnia ich inicjację do tworzenia instancji zmiennych referencyjnych, ale bardziej typowe jest inicjowanie za pomocą literałów lub stałych,:

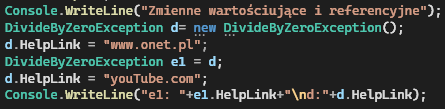
*int i=1;*

Ważną różnicą pomiędzy strukturami a klasami ujawnia się przy próbie kopiowania obiektów

*DivideByZeroException e=d;*

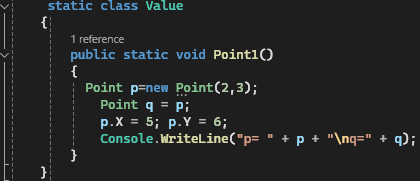
*Int j = i;*

Pierwsze z nich prowadzi tylko do utworzenia nowej referencji do wcześniej zdefiniowanego obiektu typu DivideByZeroException. W drugim tworzona jest nowa zmienna j typu int. W pierwszym przypadku nie powstaje nowy obiekt – obie referencje d i e wskazują na tę samą instancję klasy DivideByZeroException. Oznacza to, że np. zmiana własności HelpLink poprzez referencję e do tego obiektu, będzie miała taki sam efekt jak zmiana z użyciem referencji d. Użycie operatora przypisania „=” w przypadku zmiennych referencyjnych nie powoduje klonowania obiektu – nie powstała nowa instancja klasy, a tylko nowa referencja do istniejącej. Przykładowy program.:

Do zilustrowania różnicy pomiędzy klasami a strukturami zamiast typu int użyjemy struktury Point z przestrzeni System.Drawing. Typ Point opisuje punkt na płaszczyźnie i ma dwie własności X Y, które można swobodnie zmieniać po utworzeniu obiektu.

Przykład zmiany wartości :

Skopiowany obiekt typu Point jest całkowicie niezależny od oryginału. Ten sposób działania operatora przypisania powoduje, że w przypadku zmiennych wartościowych można w pełni utożsamić samą zmienną z odpowiadającymi im instancjami struktur. Struktury imitują zachowanie zmiennych prostych.

Operator porównania == w przypadku typów wartościowych zwraca wartość true gdy porównane obiekty mają taką samą wartość. Działanie operatora porównania dla typów wartościowych jest takie samo jak metody Equals. Natomiast w przypadku typów referencyjnych działanie operatora == i metody Equals różnią się od siebie. Pierwszy porównuje wartości zmiennych przechowujących referencje do obiektów. Zwraca tru jeżeli obydwie zmienne wskazują na ten sam obiekt. Metoda Equals służy do porównania stanów dwóch obiektów, zwróci prawdę jeżeli porównamy dwie zmienne wskazujące na ten sam obiekt, true zwróci również gdy obiekty będą różne ale wartości takie same.

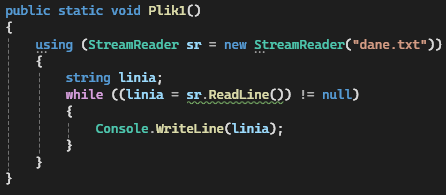
**Zwalnianie obiektów z pamięci.**

Za zwalnianie pamięci po obiektach będącymi instancjami klas odpowiedzialny jest odśmiecacz (ang. garbage collector). Cyklicznie przeszukuje pamięć sterty w poszukiwaniu obiektów, które nie są wskazywane przez żadną referencję. Jeżeli taki obiekt zostanie znaleziony jest usuwany. Użytkownik nie ma żadnego wpływu. Dzięki temu niebezpieczeństwo wycieku pamięci jest mniejsze.

Usuwając obiekty uruchamiana jest metoda Finalize. W której powinny być umieszczone polecenia zamykające zewnętrzne nie zarządzone zasoby.

Są sytuacje w których nie chcemy czekać aż odśmiecacz znajdzie zasoby pozbawione referencji i je usunie. Przykładem są klasy korzystające z plików. Chcemy, aby dostęp do nich został zamknięty jak najszybciej aby nie blokować prób ich ponownego otwarcia. W takiej sytuacji klasy definiują interfejs IDisposable, wymuszający obecność metody Dispose, w której powinny znaleźć się polecenia zwalniające wszystkie zewnętrzne zasoby używane przez obiekt. Tę metodę możemy wywołać, gdy mamy pewność, że obiekt nie jest już potrzebny. Możemy również wykorzystać ten mechanizm, kiedy nasza klasa zajmuje dużo pamięci i chcemy mieć pewność, że ta pamięć będzie zwolniona najszybciej jak to możliwe.

Słowo kluczowe using nie dotyczy tylko przestrzeni nazw ale również użyte jako modyfikator w deklaracji zmiennej referencyjnej, z inicjacją wartości, wówczas na rzecz tego obiektu zostanie automatycznie wywołana metoda Dispose wraz z zakończeniem metody w której ta zmienna lokalna jest zadeklarowana. Modyfikator using może być użyty tylko w przypadku typów implementujących interfejs IDisposable



Żeby powyższy kod zadziałał koniecznie musimy założyć plik dane.txt w miejscu gdzie jest plik wykonywalny. Dodajmy taki plik do projektu, wybierając z menu Projekt polecenie: Dodaj nowy element… - plik tekstowy  w kategorii ogólne czyli General. Aby dodany plik był kopiowany do katalogu ze skompilowanym programem należy go zaznaczyć w podoknie Eksplorator rozwiązań i w oknie właściwości zmienić jego ustawienie „Kopiuj do katalogu wyjściowego na Zawsze kopiuj . Do pliku należy wpisać w edycji kilka linijek tekstu.

Klasa StreamReader pozwala czytać z pliku tak jak ze strumienia. Żeby ten program zadziałał kursor musi być ustawiony na początku pliku. Wtedy metoda StreamReadr -ReadLine odczyta linijkę i ustawi się na początku następnej linijki, jeżeli kursor będzie na końcu pliku to warunek w pętli wskaże na null i pętla się nie wykona. Polecenie (linia=sr.ReadLine()) !=null operator przypisania „=” nie tylko zmienia wartość zmiennej linia, ale również zwraca jej nową wartość jako wynik. Możemy zatem sprawdzić, czy wynik ten jest różny od null.

**Nullable**

Tylko typ referencyjny może mieć wartość null. To znaczy, że zmienna typu referencyjnego może nie wskazywać na żaden obiekt. Zmienna wartościowa też może nie być zainicjowana, ale kompilator wychwytuje próby użycia takiej zmiennej i na to nie pozwala.

Zdefiniujemy zmienną typu Nullable<int> i zainicjujemy ją wartością 1 (literał typu int). Będzie tu niejawna konwersja z typu int na typ Nullable<int> (lub ogólniej z typu T na typ Nullable<T>).

O tym czy zmienna jest pusta, będziemy mogli się przekonać przyrównując ją do null lub sprawdzając jej właściwość HasValue.

W praktyce zamiast korzystać z właściwości HasValue, stosuje się porównanie do wartości null, np.

int i3=(in==null)? -1:ni.Value;

Tę konstrukcję można zresztą skrócić jeżeli użyjemy operatora ?? np.:

Int i4=ni??-1;