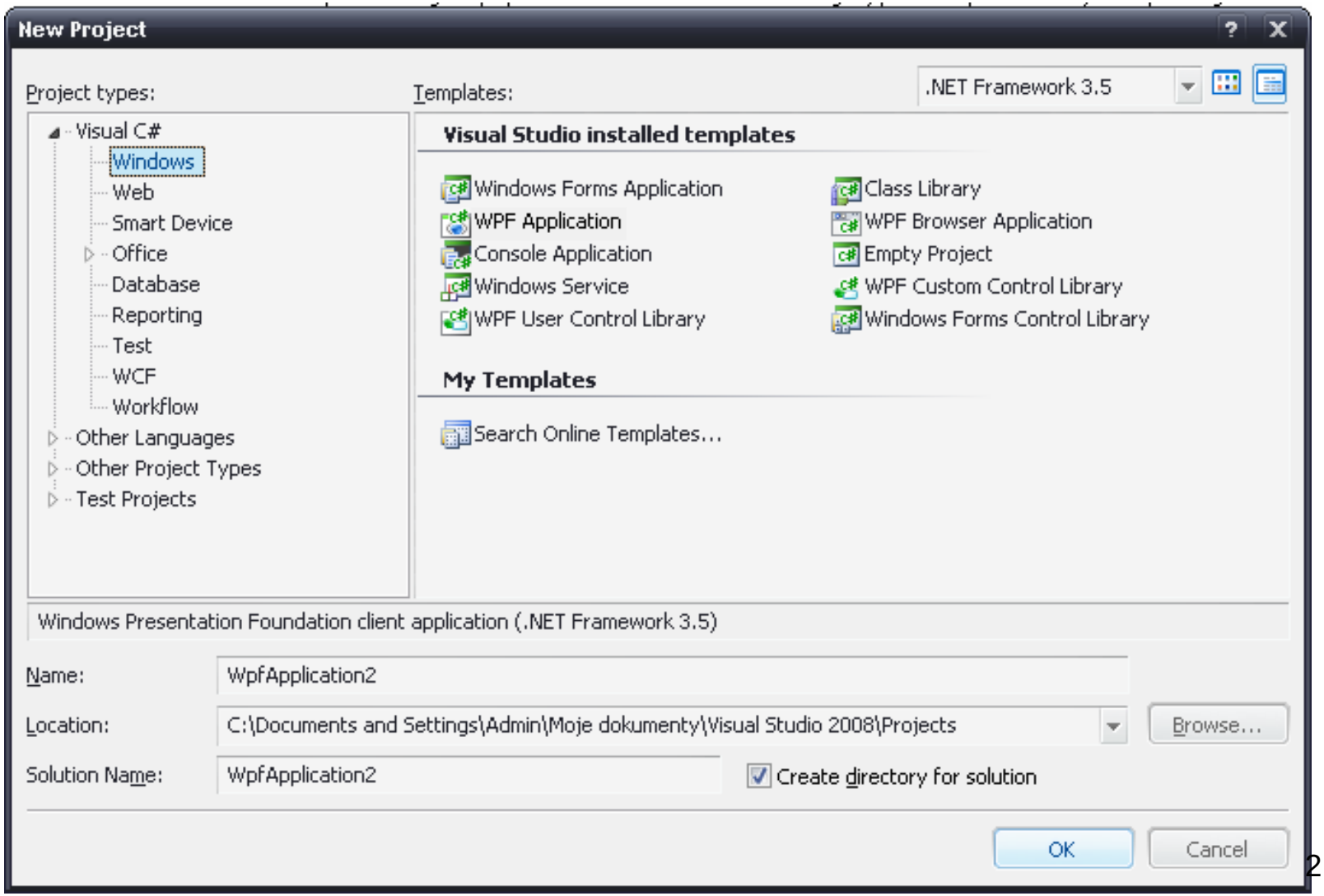
C#
WPROWADZENIE

Pierwszy program 1



Najprostszy program

```
class PierszaKlasa
{
    static void Main(string[] args)
    {
    }
}
```

Oczywiście taka aplikacja nic nie robi, zaraz po uruchomieniu (klawisz F5) zostanie zamknięta.

Metoda Main

- metoda Main jest obowiązkowym elementem programu. To od niej program rozpoczyna swe działanie i na niej je kończy.

Program musi posiadać klasę

- Nazwę klasy od słowa kluczowego class musi dzielić co najmniej jedna spacja. Klasy mogą zawierać m.in. metody, takie jak metoda Main. Zawartość klasy musi mieścić się pomiędzy klamrami { }

- class Bar { }

Słowa kluczowe

- Każdy język programowania posiada specyficzne elementy, tzw. *słowa kluczowe*. Mogą one oznaczać rozkaz czy instrukcję, które są w dany sposób interpretowane przez kompilator.

- Standardowo w środowisku Visual C# słowa kluczowe wyróżniane są kolorem.

- Do słów kluczowych C# można zaliczyć m.in. class, void, static.

- Przykładowo, słowo class oznacza *deklarację* klasy o danej nazwie. Deklaracja w kontekście języka programowania może zwyczajnie oznaczać utworzenie danego elementu (np. klasy).

Komentarze

- Najprostszym elementem każdego języka programowania są komentarze.

Podstawowym typem w języku C# są komentarze jednej linii w stylu języka C++. Przykład:

```
class Foo
{
   // to jest komentarz
   // metoda Main — tutaj zaczynamy działanie!
   static void Main(string[] args)
   {
      System.Console.WriteLine("Witaj Świecie!");
   }
}
```

Tekst znajdujący się po znakach // nie będzie brany pod uwagę przez kompilator.

Komentarze

W C# dostępne są także komentarze w stylu języka C, dzięki którym możesz „skomentować” wiele linii tekstu:

```
/*
   Tutaj jest komentarz
   Tutaj również...
   */
```

Rozpoczęcie oraz zakończenie bloku komentarza określają znaki /* oraz */.

Komentarze mogą być w sobie zagnieżdżane, przykładowo:

```
/* komentarz w stylu C
// komentarz jednej linii
 */
```

Podzespoły, metody, klasy

- Główną biblioteką w .NET Framework jest *mscorlib.dll*. Zawiera ona **przestrzeń nazw *System***, która z kolei zawiera klasę Console.

- Klasy zawierają *metody*, m.in. WriteLine, która służy do wypisywania tekstu na konsoli.

- System zależności jest dość skomplikowany

Funkcje - metody

- Funkcje jako takie nie istnieją w C#! Zamiast tego mówimy o metodach.

- Metoda jest to rodzaj operacji umożliwiającej realizację wyznaczonych czynności przez obiekty klasy.

Klasy

- O klasach można powiedzieć, iż jest to zestaw metod i atrybutów. Przykładowo, klasa Console zawiera zestaw metod służących do operowania na konsoli. Klasa jest swego rodzaju przybornikiem, paczką zawierającą przydatne narzędzia.

- Środowisko .NET Framework udostępnia szereg klas gotowych do użycia. Przykładowo, chcemy napisać program, który obliczy logarytm z danej liczby. Zamiast samemu męczyć się z pisaniem odpowiedniego kodu, możemy wykorzystać klasę Math, która jest częścią środowiska .NET Framework, i udostępniane przez nią mechanizmy.

Przestrzenie nazw

- Przestrzeń nazw skupia w sobie definicje klas dostępnych w ramach danej przestrzeni.

- W obrębie kilku przestrzeni nazw mogą istnieć klasy o tej samej nazwie.

Przykłady definicji przestrzeni nazw

```
namespace Bar
        {
    class Hello
            {

            }
        }

namespace Foo
        {
    class Hello
            {

            }
        }
```

Użycie:

```
System.Console.Read();
```

Słowo kluczowe using

- Pisanie za każdym razem nazwy przestrzeni nazw, a następnie klasy i metody może być nieco męczące. Dlatego też można wykorzystać słowo kluczowe ***using***, które informuje kompilator, że w programie będziemy korzystali z klas znajdujących się w danej przestrzeni nazw (np. System):

```
using System;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");
    }
}
```

Deklarowanie zmiennych

- Operacja utworzenia zmiennej nazywana jest deklaracją zmiennej.
- Musimy określić unikalną nazwę zmiennej, która nie może się powtarzać w obrębie danej klasy czy metody.
- Musimy również określić typ danej zmiennej, czyli zidentyfikować dane, jakie będziemy przechowywać w pamięci (tekst, liczby itp.).

```
class Foo
{
   static void Main(string[] args)
   {
      string Bar;
   }
}
```

Deklaracja kilku zmiennych

- Potrzebujemy trzech zmiennych. Jeżeli wszystkie zmienne są tego samego typu (string), możemy zadeklarować je w ten sposób:

```
string Login, FName, LName;
```

- Nazwy zmiennych musimy oddzielić znakiem przecinka.

- Z punktu widzenia kompilatora nie ma znaczenia to, w jaki sposób deklarujesz zmienne, więc równie dobrze możesz je zadeklarować w ten sposób:

```
string Login;
string FName;
string LName;
```

Przydział danych

- Przypisanie danych do zmiennej jest równie proste jak deklaracja. W tym celu używamy operatora przypisania (=):

```
class Foo
{
    static void Main(string[] args)
    {
        string Bar = "Hello World";
    }
}
```

- Inny sposób przypisania wartości

```
string Bar;

Bar = "Hello World";

Bar = "Hello my darling!";

Console.WriteLine(Bar);
```

Typy skalarne i referencyjne

- Typ danych określa charakter zmiennej — czy zmienna jest łańcuchem znaków, liczbą całkowitą, liczbą rzeczywistą lub jeszcze jakimś innym obiektem.

- W środowisku .NET Framework istnieją typy **skalarne i typy referencyjne**.

- Typy skalarne to typy proste (np. liczby całkowite, liczby rzeczywiste, znaki), wyliczenia i struktury.

- Typy referencyjne to na przykład klasy, interfejsy, delegaty i tablice.

Typy skalarne i referencyjne cd.1

- Zmienne typu skalarnego bezpośrednio zawierają przypisywane im dane, a zmienne typów referencyjnych tylko wskazują obiekty.

- W przypadku typów referencyjnych możliwe jest, że dwie zmienne wskazują ten sam obiekt. Istnieje możliwość, że operacje przeprowadzane na jednej zmiennej referencyjnej wpłyną na zmianę obiektu wskazywanego przez inną zmienną referencyjną.

- W przypadku typów skalarnych zmienna zawiera własną kopię danych i nie ma możliwości zmiany tych danych poprzez operacje na innej zmiennej.

Typy skalarne i referencyjne cd.2

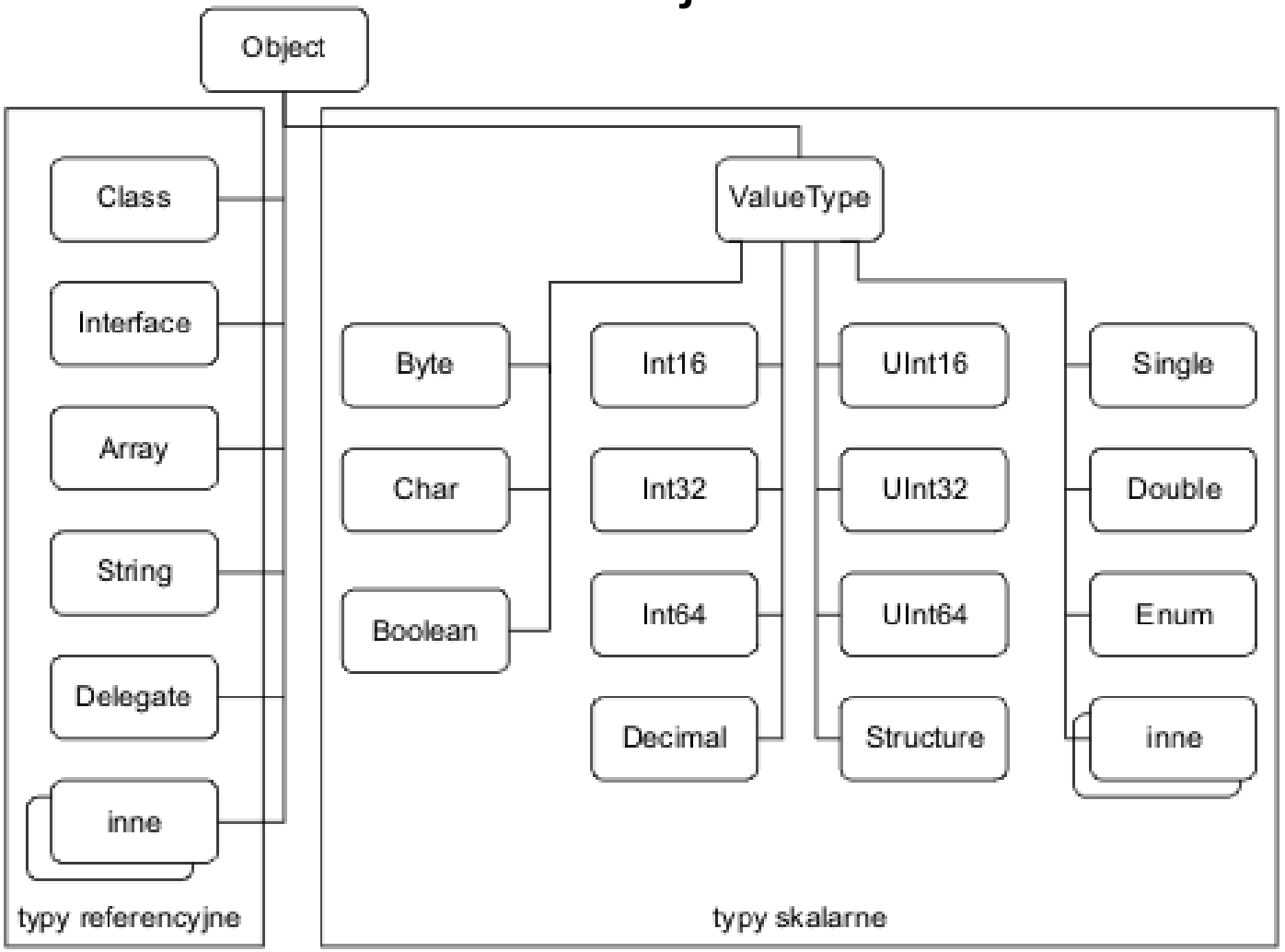
```
using System;
class Class1 {
    static void Main() {
      // definicja zmiennej całkowitoliczbowej o wartości 3
        int skalar1 = 3;
        int skalar2 = skalar1; // druga zmienna o tej samej wartości
        skalar2 = 76;
    Console.WriteLine("Typy skalarne: {0}, {1}", skalar1, skalar2);

        // deklaracja tablicy liczb zawierającej jeden element o wartości 3
        int []ref1 = {3};
        int []ref2 = ref1; // skopiowanie zmiennej, ale nie tablicy!
        ref2[0] = 76;

        Console.WriteLine("Referencje: {0}, {1}", ref1[0], ref2[0]);
     }
    }
```

Typy danych

- Najważniejsze typy danych zdefiniowane w CTS przedstawiono na ilustracji



Typy danych: **liczby całkowite**

Nazwa klasy	C#	**Opis**	**zakres**
Byte	byte	8-bitowa liczba całkowita bez znaku	0 do 255
SByte	sbyte	8-bitowa liczba całkowita ze znakiem	–128 do 127
Int16	short	16-bitowa liczba całkowita ze znakiem	–32,768 do 32,767
Int32	int	32-bitowa liczba całkowita ze znakiem	–2,147,483,648 do 2,147,483,647
Int64	long	64-bitowa liczba całkowita ze znakiem	_–9,223,372,036,854,775,808 do 9,223,372,036,854,775,807
UInt16	ushort	16-bitowa liczba całkowita bez znaku	unsigned short
UInt32	uint	32-bitowa liczba całkowita bez znaku	0 do 4,294,967,295
UInt64	ulong	64-bitowa liczba całkowita bez znaku	0 do 18,446,744,073,709,551,615

Liczby zmiennoprzecinkowe

Nazwa klasy	C#	**Opis**	**Zakres**
Single	float	liczba zmiennoprzecinkowa pojedynczej precyzji (32-bity)	–3.402823e38 do 3.402823e38
Double	double	liczba zmiennoprzecinkowa podwójnej precyzji (64-bit)	–1.7976931348623e308 do 1.7976931348623e308

Wartości logiczne

Nazwa klasy	C#	**Opis**	**Zakres**
Boolean	bool	wartość logiczna (prawda lub fałsz)	true lub false

Typy danych: **inne**

Nazwa klasy	C#	**Opis**	**Zakres**
Char	char	znak Unicode (16-bitowy)	
Decimal	decimal	96-bitowa liczba dziesiętna	
String	string	ciąg znaków Unicode o stałej długości,	

Restrykcje w nazewnictwie zmiennych

- Pierwszym znakiem nazwy zmiennej nie może być cyfra — nazwa ta musi rozpoczynać się od litery.
- Nazwa zmiennej może zawierać na początku _
- Następujące znaki nie są dozwolone w nazwach zmiennych:
  ( ) * & ^ % # @ ! / = + - [}] } ' " ; , . ?

  Platforma .NET oraz sam język C# obsługują standard kodowania znaków Unicode, dlatego w nazwach zmiennych można używać polskich znaków lub jakichkolwiek innych wchodzących w skład Unicode:
  byte _gżegżółka = 234

Stałe

- Stałe od zmiennych odróżnia to, że zawartość przydzielana jest jeszcze w trakcie pisania kodu i nie ulega późniejszym zmianom.

- Zawartości stałych nie można zmieniać w trakcie działania aplikacji. Raz przypisana wartość pozostaje w pamięci komputera aż do zakończenia działania aplikacji:

  const double Version = 1.0;
  Version = 2.0; *// w tej linii kompilator wskaże błąd*

- Stałe deklarowane są prawie identycznie jak zmienne. Jedyna różnica to konieczność poprzedzenia deklaracji słowem kluczowym ***const***.

Operacje na konsoli

- Programy na konsolę nie posiadają żadnych okien, kontrolek itp., interakcja z użytkownikiem jest słaba.

- Program może jedynie wypisywać tekst na konsoli (WriteLine()) lub odczytać tekst wpisany przez użytkownika (ReadLine()).

Prosty program w konsoli

```
using System;

  class Program
  {
      static void Main(string[] args)
      {
          Console.WriteLine("Cześć, jak masz na imię?");

          string name; // deklaracja zmiennej

          name = Console.ReadLine(); // pobranie tekstu wpisanego
  przez użytkownika

          Console.WriteLine("Miło mi " + name + ". Jak się masz?");
          Console.ReadLine();
      }
  }
```

Metody klasy Console

Wybrane metody klasy Console

Metoda	**Opis**
Beep	Umożliwia odegranie dźwięku z głośnika systemowego.
Clear	Czyści ekran konsoli.
ResetColor	Ustawia domyślny kolor tła oraz tekstu.
SetCursorPosition	Ustawia pozycję kursora w oknie konsoli.
SetWindowPosition	Umożliwia ustawienie położenia okna konsoli.
SetWindowSize	Umożliwia określenie rozmiaru okna konsoli.

Prosta aplikacja w konsoli 1

```
using System;

class Program
{
    static void Main(string[] args)
    {
        // ustaw rozmiar okna
        Console.SetWindowSize(60, 30);
        // ustaw położenie tekstu
        Console.SetCursorPosition(10, 10);
        Console.WriteLine("Hello World!");

        Console.ReadLine();
        Console.Clear();
        Console.Beep();
    }
}
```

Prosta aplikacja w konsoli 2

```
using System;

class Program
{
    static void Main(string[] args)
    {
        // nadanie wartości dla właściwości
        Console.Title = "Hello World";
        // określenie koloru tła (ciemny żółty)
        Console.BackgroundColor = ConsoleColor.DarkYellow;

        // odczyt wartości właściwości
        Console.WriteLine("Tytuł tego okna to: " + Console.Title);

        Console.ReadLine();
    }
}
```

Operatory

- W każdym języku programowania wysokiego poziomu istnieją symbole służące do sterowania programem. Takie znaki nazywane są przez programistów operatorami.

- *Symbole operatorów w języku C# są praktycznie identyczne z tymi z języka C++ oraz Java.*

Operatory porównania

Operator	**Język C#**
Nierówności	!=
Równości	==
Większości	>
Mniejszości	<
Większe lub równe	>=
Mniejsze lub równe	<=

Operatory arytmetyczne

Operator	**Język C#**
Dodawanie	+
Odejmowanie	-
Mnożenie	*
Dzielenie rzeczywiste	/
Dzielenie całkowite	/
Reszta z dzielenia	%

Operatory arytmetyczne - przykłady

```
double d = 5.0;
int i = 5;
Console.WriteLine(5.0 / 5); // 1
Console.WriteLine(-i / 2); // -2
Console.WriteLine(-d / 2); // -2.5
```

Operator inkrementacji oraz dekrementacji

- int i = 5;

```
Console.WriteLine(i++); // 5
i = 5;
Console.WriteLine(++i); // 6

i--;
--i;
```

Operatory logiczne

- Operatory logiczne często są nazywane operatorami boolowskimi (ang. *Boolean operators*). Wynika to z tego, że realizują one operacje właściwe dla algebry Boole'a.

Operator	**Język C#**
Logiczne i	&&
Logiczne lub	||
Zaprzeczenie	!

Operatory bitowe

Operator	**Język C#**
Koniunkcja	&
Zaprzeczenie	~
Alternatywa	|
Dysjunkcja (NAND)	^
Przesunięcie w lewo	<<
Przesunięcie w prawo	>>

Operatory przypisania

- foo = 4;
  bar = foo;

- x = x + 2;
  x += 2;

- Pozostałe operatory przypisania to:
  -=, *=, /=, %=, &=, |=, ^=, <<=, >>=.

Instrukcje warunkowe

- Są to konstrukcje, które służą do sprawdzania, czy dany warunek został spełniony. Jest to praktycznie podstawowy element języka programowania — dzięki instrukcjom warunkowym możemy odpowiednio zareagować na istniejące sytuacje i sterować pracą programu.

Instrukcja if

- Ogólna budowa instrukcji *if* wygląda następująco:

  ```
  if (warunek do spełnienia)
  {
  operacje do wykonania, jeżeli warunek jest prawdziwy;
  }
  ```

- Obowiązkowym elementem każdej instrukcji *if* są nawiasy okrągłe, w których musi znaleźć się warunek do sprawdzenia.

- Warunek może zostać spełniony (czyli wynik pozytywny) albo nie. Od tego zależy, czy wykonany zostanie kod znajdujący się pomiędzy klamrami.

Instrukcja if - przykład

```
static void Main(string[] args)
    {
        int x = 5;

        if (x == 5)
        {
            Console.WriteLine("Zmienna x ma wartość 5!");
        }

        Console.ReadLine();
    }
```

Jaki będzie wynik działania programu?

```
static void Main(string[] args)
    {
        int x = 6;

        if (x == 5)
         Console.WriteLine("Zmienna x ma wartość 5!");
         Console.WriteLine("Zmienna x nie ma wartości 5");

        Console.ReadLine();
    }
```

Instrukcje zagnieżdżone

- Instrukcje *if* można dowoli zagnieżdżać:

```
if (x == 5)
{
   Console.WriteLine("Zmienna x ma wartość 5!");

    if (y >= 10)
    {
        Console.WriteLine("Drugi warunek również spełniony");
    }
}
```

- Jeżeli pierwszy warunek zostanie spełniony, wykonany zostanie kod wyświetlający wiadomość na ekranie konsoli. Następnie sprawdzony zostanie kolejny warunek i — po raz kolejny — jeżeli zostanie on spełniony, wykonany zostanie odpowiedni kod.

Zastosowanie nawiasów

- int x, y, z;

```
x = 2;
y = 4;
z = 10;

if ((x == 2 || y == 4) && z > 20)
{
    Console.WriteLine("Yes, yes, yes!");
}
```

Słowo kluczowe ***else***

- ```
  // warunek sprawdza, czy jest po godzinie 18:00
  if (DateTime.Now.Hour >= 18)
  {
    Console.WriteLine("Dobry wieczór koleżanko!");
  }
  // warunek sprawdza, czy jest przed godziną 18:00
  else
  {
    Console.WriteLine("Dzień dobry koleżanko!");
  }
  ```
  ```

Instrukcja else if

- ```
  int X = DateTime.Now.Hour;

  if (X == 12)
  {
      Console.WriteLine("Jest południe!");
  }
  else if (X > 12 && X < 21)
  {
      Console.WriteLine("No cóż... już po 12:00");
  }
  else if (X >= 21)
  {
      Console.WriteLine("Oj... toż to środek nocy");
  }
  else
  {
      Console.WriteLine("Dzień dobry");
  }
  ```
  ```

Instrukcja switch

- Jest to kolejna instrukcja warunkowa języka C#. Umożliwia sprawdzanie wielu warunków.

```
int Mandat = 50;

switch (Mandat)
{
    case 10:
        Console.WriteLine("10 zł mogę zapłacić");
        break;

    case 20:
        Console.WriteLine("Oj, 20 zł to troszkę dużo");
        break;
}
```

- Słowo kluczowe break nakazuje zakończenie instrukcji switch.

Instrukcja switch – wartość domyślna

- W języku C# możemy użyć słowa kluczowego ***default***, aby odpowiednio zareagować, gdy żaden ze wcześniejszych warunków nie zostanie spełniony:

```
int Mandat = 50;

switch (Mandat)
{
    case 10:
        Console.WriteLine("10 zł mogę zapłacić");
        break;

    case 20:
        Console.WriteLine("Oj, 20 zł to troszkę dużo");
        break;

    default:
        Console.WriteLine("Niezidentyfikowana suma");
        break;
}
```

Pętle

- W języku c# wykorzystywane są pętle trzech rodzajów:
  - while
  - do while
  - for

Pętla while

- Pętla while umożliwia wielokrotne wykonywanie tego samego kodu, **dopóki jest spełniony warunek**.
- W tym momencie musimy ponownie posłużyć się operatorami logicznymi oraz porównania. Ogólna budowa pętli while wygląda następująco:

```
while (warunek zakończenia)
{
// kod do wykonania
}
```

Pętla while - przykład

- Napiszmy jakiś prosty program, który będzie wyświetlał w pętli dowolny tekst, powiedzmy — dziesięciokrotnie. Spójrz na poniższy kod:

```
using System;

class Program
{
    static void Main(string[] args)
    {
        int X = 1;

        while (X <= 10)
        {
            Console.WriteLine("Odliczanie..." + X);
            ++X;
        }
        Console.Read();
    }
}
```

Pętla do-while

- Pętla do-while zostanie wykonana co najmniej raz, ponieważ warunek jej zakończenia jest sprawdzany po wykonaniu kodu z jej ciała.
- Zapewnia wykonywanie tego samego kodu, **dopóki jest spełniony warunek**.
- ```
  do
  {
      // kod do wykonania

  }
  while (warunek zakończenia);
  ```
  ```

Pętla do-while - przykład

- ```
  int X = 20;

  do
  {
      Console.WriteLine(„Wykonanie");
      X++;
  }
  while (X < 20);
  ```
  ```

Pętla for

- Budowę pętli można zaprezentować następująco:

```
for (wartość_startowa; wartość_końcowa; licznik_pętli)
{
// kod pętli
}
```

Pętla for – przykład 1

- ```
  int i;
  for (i = 1; i <= 10; i++) // dobrze

  for (int i = 1; i <= 10; i++) // dobrze
  ```

- **Po każdej iteracji zwiększany zostaje licznik pętli (czyli wartość zmiennej i).**
- **Początkowa wartość zmiennej i jest przypisywana w nagłówku (w naszym przykładzie początkową wartością jest 1).**
- **W nagłówku określamy warunek zakończenia działania pętli (pętla będzie kontynuowana, dopóki wartość zmiennej i jest mniejsza lub równa 10).**
  ```

Pętla for – inne przykłady

Inkrementacja o dwa

```
for (int i = 1; i <= 20; i += 2)
{
    Console.WriteLine("Odliczanie..." + i);
}
```

==========

Odliczanie od góry do dołu

```
for (int i = 20; i > 0; i--)
{
    Console.WriteLine("Odliczanie..." + i);
}
```

==========

Pętla for – inne przykłady

Parametry opcjonalne

```
for (int i = 20; i > 0; )
{
    Console.WriteLine("Odliczanie..." + i);
    i -= 2;
}
```

==========

Pętla nieskończona

```
for (; ; )
```

Instrukcja break

- Instrukcja ***break*** powoduje natychmiastowe wyjście z pętli

```
int i = 0;

    for (; ; )
    {
        ++i;

        Console.WriteLine("Odliczanie..." + i);

        if (i == 20)
        {
            break;
        }
    }

    Console.Read();
```

Instrukcja continue

- Instrukcja ***continue*** umożliwia przeskoczenie do następnej iteracji.

```
for (; ; )
{
    ++i;

    // pomijamy wartości parzyste
    if (i % 2 == 0)
    {
        continue;
    }

    Console.WriteLine("Odliczanie..." + i);

    // pomijamy wartości parzyste,
    // warunkiem jest osiągnięcie wartości 101
    if (i == 101)
    {
        break;
    }
}
```

Instrukcja *foreach*

- Pętla ***foreach*** powtarza wykonywanie instrukcji lub grupy instrukcji dla każdego elementu w określonym zbiorze (np.: dla każdego znaku w łańcuchu znaków).

Składnia ***foreach***

foreach (typ identyfikator in wyrażenie) instrukcja;

gdzie:

- ***typ***
  typ danych elementu określonego zbioru (wymagane),

- ***identyfikator***
  nazwa zmiennej dla elementu zbioru (wymagane),

- ***wyrażenie***
  obiekt kolekcji lub wyrażenie tablicowe, musi istnieć możliwość konwersji typu elementu kolekcji lub tablicy na typ danych identyfikatora (wymagane),

- ***instrukcja***
  instrukcja lub grupa instrukcji w bloku, powtarzana w pętli (wymagane).

Przykład ***foreach***

```
string strNapis = "Ala ma kota";

foreach (char cZnak in strNapis)
 Console.WriteLine("", cZnak);
```

Koniec męczarni