



Polsko-Japońska Wyższa Szkoła  
Technik Komputerowych

Piotr Tronczyk

U K O

Visual Studio .NET

VB.NET i C#

K M K T

2 0 0 6

# Spis treści

<b>Spis tabel</b> . . . . .	iv
<b>Spis rysunków</b> . . . . .	vi
<b>Rozdział 1. Laboratorium 1</b> . . . . .	1
1.1. Wprowadzenie do .NET Framework . . . . .	1
1.1.1. Common Language Runtime (CLR) . . . . .	1
1.1.2. Microsoft Intermediate Language (MSIL) . . . . .	2
1.1.3. Podstawowe typy danych . . . . .	3
1.2. Aplikacje konsolowe . . . . .	5
1.2.1. Pierwsze aplikacje w VB.NET . . . . .	5
1.2.2. Pierwsze aplikacje w C# . . . . .	8
1.3. Operatory relacji . . . . .	10
1.3.1. Operatory relacji w VB .NET . . . . .	10
1.3.2. Operatory relacji w C# . . . . .	10
1.4. Operatory logiczne i bitowe . . . . .	11
1.4.1. Operatory logiczne i bitowe w VB .NET . . . . .	11
1.4.2. Operatory logiczne i bitowe w C# . . . . .	12
1.5. Instrukcje warunkowe . . . . .	13
1.5.1. Instrukcje warunkowe w VB .NET . . . . .	13
1.5.2. Instrukcje warunkowe w C# . . . . .	15
<b>Rozdział 2. Laboratorium 2</b> . . . . .	18
2.1. Łańcuchy znaków w .NET . . . . .	18
2.1.1. Porównywanie napisów . . . . .	18
2.1.2. Kopiowanie i konkatencja łańcuchów znaków . . . . .	20
2.1.3. Dodawanie, usuwanie oraz zastępowanie ciągów znaków . . . . .	21

---

2.1.4. Zmiana wielkości liter . . . . .	23
2.1.5. Formatowanie napisów . . . . .	23
2.1.6. Formatowanie daty . . . . .	24
2.1.7. Wycinanie białych znaków . . . . .	26
2.1.8. Dopelnianie łańcuchów znaków . . . . .	27
2.1.9. Klasa <code>StringBuilder</code> . . . . .	28
2.2. Pętle . . . . .	30
2.2.1. Konstrukcja <code>For...Next</code> VB.NET . . . . .	30
2.2.2. Konstrukcja <code>for</code> C# . . . . .	31
2.2.3. Konstrukcja <code>Do While...Loop</code> VB.NET . . . . .	35
2.2.4. Konstrukcja <code>Do...Loop While</code> VB.NET . . . . .	35
2.2.5. Konstrukcja <code>Do...Loop Until</code> VB.NET . . . . .	36
2.2.6. Konstrukcja <code>while</code> C# . . . . .	36
2.2.7. Konstrukcja <code>do...while</code> C# . . . . .	37
<b>Rozdział 3. Laboratorium 3</b> . . . . .	<b>38</b>
3.1. Tablice . . . . .	38
3.1.1. Tablice w VB.NET . . . . .	38
3.1.2. Tablice w C# . . . . .	42
3.2. Funkcje matematyczne . . . . .	45
3.3. Obsługa wyjątków . . . . .	45
3.3.1. Wyjątki VB.NET . . . . .	46
3.3.2. Wyjątki C# . . . . .	49
<b>Rozdział 4. Laboratorium 4</b> . . . . .	<b>51</b>
4.1. Klasy . . . . .	51
4.1.1. Klasy w VB.NET . . . . .	52
4.1.2. Klasy w C# . . . . .	56
4.1.3. Przeciążanie metod . . . . .	61
4.1.4. Właściwości (properties) . . . . .	62
<b>Rozdział 5. Laboratorium 5</b> . . . . .	<b>65</b>
5.1. Aplikacje Okienkowe . . . . .	65
5.2. Prosty kalkulator . . . . .	67
5.2.1. Kod VB.NET . . . . .	68
5.2.2. Kod C# . . . . .	71

---

5.3. Zegarek elektroniczny . . . . .	73
5.3.1. Kod C# . . . . .	74
5.3.2. Kod VB.NET . . . . .	75
5.4. Obsługa menu . . . . .	77
5.5. Kontrolka ProgressBar . . . . .	78
5.6. Technika przeciągnij i upuść (drag & drop) . . . . .	79
<b>Rozdział 6. Laboratorium 6 . . . . .</b>	<b>82</b>
6.1. GDI+ . . . . .	82
6.1.1. Przestrzenie nazw GDI+ . . . . .	82
6.1.2. Klasa Graphics . . . . .	85
6.1.3. Obiekty graficzne . . . . .	87
6.2. Zegar analogowy . . . . .	89
6.2.1. Rysowanie tarczy zegara . . . . .	90
6.2.2. Godziny . . . . .	91
6.2.3. Wskazówki . . . . .	94
<b>Rozdział 7. Laboratorium 7 . . . . .</b>	<b>97</b>
7.1. Programowanie obiektowe . . . . .	97
7.1.1. Klasa Complex (liczby zespolone) . . . . .	100
7.1.2. Zbiór Mandelbrota . . . . .	102
<b>Słowniczek skrótów . . . . .</b>	<b>106</b>

# Spis tabel

1.1.1 Zestawienie typów danych dla VB.NET, C# oraz ich odpowiedniki w CTS . . . . .	4
1.3.1 Zestawienie podstawowych operatorów relacji dla VB .NET . . . . .	10
1.3.2 Zestawienie podstawowych operatorów relacji dla C# . . . . .	11
1.4.1 Zestawienie podstawowych operatorów logicznych dla VB .NET . . . . .	11
1.4.2 Zestawienie podstawowych operatorów logicznych dla C# . . . . .	12
2.1.1 Znaczenie wartości zwracanych przez metodę Compare . . . . .	18
2.1.2 Formatowanie liczb . . . . .	24
2.1.3 Własne znaczniki formatowania liczb . . . . .	24
2.1.4 Znaczniki formatowania daty . . . . .	24
2.1.5 Znaczniki formatowania daty . . . . .	25
2.1.6 Właściwości klasy <code>StringBuilder</code> . . . . .	28
2.1.7 Metody klasy <code>StringBuilder</code> . . . . .	28
3.2.1 Funkcje matematyczne . . . . .	45
3.3.1 Wyjątki . . . . .	48
4.1.1 Modyfikatory klasowe . . . . .	60
6.1.1 Wybrane klasy przestrzeni nazw <code>System.Drawing</code> . . . . .	83
6.1.2 Wybrane struktury przestrzeni nazw <code>System.Drawing</code> . . . . .	83
6.1.3 Wybrane klasy przestrzeni nazw <code>System.Drawing.Drawing2D</code> . . . . .	84
6.1.4 Wybrane enumeratory przestrzeni nazw <code>System.Drawing.Drawing2D</code> . . . . .	84
6.1.5 Wybrane klasy przestrzeni nazw <code>System.Drawing.Printing</code> . . . . .	84
6.1.6 Wybrane metody klasy <code>Graphics</code> . . . . .	86
6.1.7 Podstawowe obiekty graficzne GDI+ . . . . .	87
6.1.8 Struktura <code>Color</code> . . . . .	88

6.1.9 Style czcionki <code>FontStyle</code> . . . . .	88
---	----

# Spis rysunków

1.1.1 Schemat komunikacji . . . . .	1
1.1.2 Schemat wykonania . . . . .	2
1.2.1 Tworzenie nowego projektu . . . . .	5
2.1.1 Wynik działania metody <code>PadLeft</code> oraz <code>PadRight</code> . . . . .	27
2.1.2 Wynik działania programu korzystającego z klasy <code>StringBuilder</code> . . . . .	28
2.2.1 Wynik działania kodu zagadki . . . . .	34
2.2.2 Wynik działania kodu zagadki ( <code>++i</code> ) . . . . .	34
3.1.1 Wynik działania programu . . . . .	44
5.1.1 Tworzenie nowego projektu dla aplikacji okienkowej . . . . .	66
5.1.2 Utworzona aplikacja VB.NET . . . . .	66
5.2.1 Wygląd okienka . . . . .	67
5.2.2 Komunikat o błędzie . . . . .	70
5.3.1 Wygląd okienka programu zegarek . . . . .	73
5.3.2 Wybór metod w oknie kodu . . . . .	77
5.4.1 Kontrolka menu na formie. . . . .	77
5.5.1 Zegar cyfrowy (menu oraz pasek postępu). . . . .	79
5.6.1 Zegar cyfrowy (paleta kolorów). . . . .	80
6.2.1 Zegar analogowy. . . . .	89
6.2.2 Zegar analogowy pierwszy tarcza. . . . .	91
6.2.3 Elipsa. . . . .	91
6.2.4 Elipsa układ współrzędnych okna. . . . .	92
6.2.5 Zegar analogowy rysowanie godzin. . . . .	93
6.2.6 Zegar analogowy wygląd aplikacji (tarcza z godzinami). . . . .	94

6.2.7 Obrót punktu względem innego punktu. . . . .	95
7.1.1 Dodawanie nowej klasy. . . . .	101
7.1.2 Zbiór Mandelbrota. . . . .	102
7.1.3 Zbiór Mandelbrota (skalowanie). . . . .	103
7.1.4 Zbiór Mandelbrota (tworzenie okna programu). . . . .	103



---

## Rozdział 1

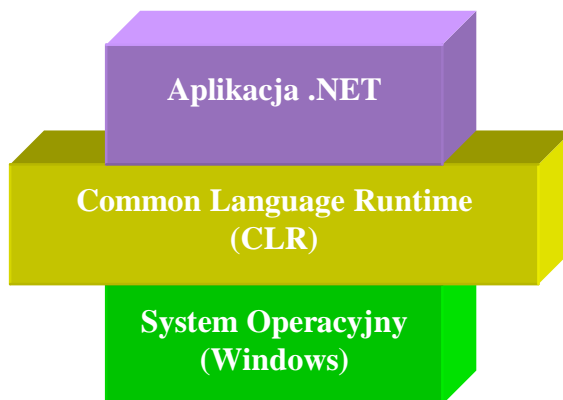
---

# Laboratorium 1

## 1.1. Wprowadzenie do .NET Framework

### 1.1.1. Common Language Runtime (CLR)

Najważniejszą cechą środowiska .NET jest **wspólne środowisko uruchomieniowe** (ang. *Common Language Runtime*, w skrócie CLR). Jest to warstwa znajdująca się 'ponad' systemem operacyjnym, obsługująca wykonanie wszystkich aplikacji środowiska .NET. Programy napisane na platformę .NET komunikują się z systemem operacyjnym poprzez CLR.



Rys. 1.1.1. Schemat komunikacji

CLR to podstawa całego systemu .NET Framework. Wszystkie języki środowiska .NET (na przykład C# czy Visual Basic .NET), a także wszystkie biblioteki klas obecne

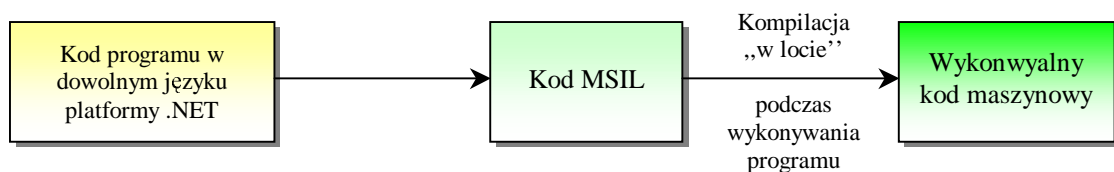
w .NET Framework (ASP.NET, ADO.NET i inne) oparte są na CLR. Ponieważ nowe, tworzone przez firmę Microsoft oprogramowanie, także oparte jest na .NET Framework, każdy, kto chce korzystać ze środowiska Microsoft, prędzej czy później będzie musiał zetknąć się z CLR.

Środowisko CLR kompiluje i wykonuje zapisany w standardowym języku pośrednim Microsoft (MSIL) kod aplikacji zwany kodem zarządzanym (ang. *managed code*), zapewniając wszystkie podstawowe funkcje konieczne do działania aplikacji. Podstawowym elementem CLR jest standardowy zestaw typów danych, wykorzystywanych przez wszystkie języki oparte na CLR, a także standardowy format metadanych, służących do opisu oprogramowania wykorzystującego te typy danych. CLR zapewnia także mechanizmy umożliwiające pakowanie kodu zarządzanego w jednostki zwane podzespołami.

W CLR wbudowane są także mechanizmy kontroli bezpieczeństwa wykonywania aplikacji — bezpieczeństwo oparte na uprawnieniach kodu (ang. *Code Access Security*, w skrócie CAS) oraz bezpieczeństwo oparte na rolach (ang. *Role-Based Security*, w skrócie RBS).

### 1.1.2. Microsoft Intermediate Language (MSIL)

Kompilując aplikację środowiska .NET, napisaną w dowolnym języku wchodzącym w skład środowiska (np. C# czy VB.NET), nie jest dokonywana konwersja na wykonywalny kod binarny, ale tworzony jest kod pośredni, nazywany MSIL lub IL, który jest zrozumiały dla warstwy CLR.



Rys. 1.1.2. Schemat wykonania

MSIL to kod dość podobny do zestawu instrukcji procesora. Obecnie nie istnieje jednak żaden sprzęt, który mógłby bezpośrednio wykonywać kod MSIL (nie jest jednak wykluczone, że w przyszłości taki sprzęt powstanie). Na razie kod MSIL musi być tłumaczony na język maszynowy procesora, na którym ma być uruchomiony.

Kompilacja kodu źródłowego języka wyższego poziomu na kod pośredni jest podstawową techniką, wykorzystywaną przez nowoczesne kompilatory. Kompilatory pakietu Visual Studio tłumaczą kod źródłowy różnych języków na taki sam kod pośredni, który następnie kompilowany jest na kod maszynowy przez jeden wspólny kompilator. To właśnie ten kod maszynowy stanowił finalny kod aplikacji przed wprowadzeniem środowiska .NET Framework.

Przenaszalność nie jest jedyną zaletą stosowania języka pośredniego. Odmienne niż w przypadku kodu maszynowego, który może zawierać wskaźniki do dowolnych adresów, kod MSIL może przed uruchomieniem zostać sprawdzony pod względem bez-

pieczeństwa typów. Podnosi to poziom bezpieczeństwa i daje większą niezawodność, gdyż działanie takie pozwala na wykrycie pewnych rodzajów błędów oraz wielu prób ataków.

Najczęściej stosowaną metodą kompilacji kodu MSIL na kod natywny jest załadowanie przez CLR podzespołu do pamięci, a następnie kompilacja każdej metody w momencie pierwszego jej wywołania. Ponieważ każda metoda kompilowana jest tylko w momencie pierwszego uruchomienia, proces kompilacji nazywa się kompilacją w samą porę (ang. *just-in-time compilation*, w skrócie JIT).

Kompilacja JIT umożliwia kompilowanie tylko tych metod, które są rzeczywiście wykorzystywane. Jeśli metoda została załadowana do pamięci razem z całym podzespolem, ale nigdy nie została wywołana, pozostanie w pamięci komputera w postaci MSIL. Skompilowany kod maszynowy nie jest zapisywany z powrotem na dysk twardy — przy ponownym uruchomieniu aplikacji kod MSIL będzie musiał zostać ponownie skompilowany.

Inną metodą kompilacji jest wygenerowanie całego kodu binarnego danego podzespołu z użyciem narzędzia NGEN (ang. *Native Image Generator*, w skrócie NGEN), dostępnego w .NET Framework SDK. Narzędzie to, uruchamiane poleceniem `ngen.exe`, kompiluje cały podzespół i umieszcza jego kod maszynowy w obszarze zwanym pamięcią podręczną obrazów kodu natywnego (ang. *Native Image Cache*, w skrócie NIC). Pozwala to na szybsze uruchamianie aplikacji, ponieważ podzespoły nie muszą już być kompilowane metodą JIT.

Kompilacja kodu MSIL na kod maszynowy pozwala na sprawdzenie bezpieczeństwa typów danych. Proces ten, zwany weryfikacją, sprawdza kod MSIL oraz metadane metod pod kątem prób niepowołanego uzyskania dostępu do zasobów systemu. Na tym etapie sprawdzane są także ustawienia bezpieczeństwa dla kodu. Administrator systemu może wyłączyć tę funkcję, jeśli nie jest ona potrzebna.

### 1.1.3. Podstawowe typy danych

Język programowania to połączenie składni oraz zbioru słów kluczowych, umożliwiające definiowanie danych oraz operacji przeprowadzanych na tych danych. Różne języki różnią się pod względem składni, jednak podstawowe pojęcia są dość podobne — większość języków obsługuje takie typy danych jak liczba całkowita czy łańcuch znaków i umożliwia porządkowanie kodu w metody oraz zbieranie metod i danych w klasy. Przy zachowaniu odpowiedniego poziomu abstrakcji, możliwe jest zdefiniowanie zestawu typów danych niezależnego od składni języka. Zamiast łączyć składnię (syntaktykę) i semantykę, można je określić oddzielnie, co pozwoli na zdefiniowanie większej liczby języków korzystających z tych samych pojęć (typów danych). Takie właśnie podejście zastosowano w CLR. Wspólny zestaw typów danych (ang. *Common Type System*, w skrócie CTS) nie jest związany z żadną składnią lub słowami kluczowymi — definiuje jedynie zestaw typów danych, który może być wykorzystywany przez wiele języków. Każdy język zgodny z CLR może używać dowolnej składni, ale musi korzystać przynajmniej z części typów danych zdefiniowanych przez CTS.

Zestaw typów danych definiowany przez CTS należy do głównych składników CLR.

Każdy język programowania oparty na CLR może udostępniać programiście te typy danych we właściwy sobie sposób. Twórca języka może skorzystać tylko z niektórych typów danych, może też definiować własne typy danych. Jednak większość języków wszechstronnie korzysta z CTS.

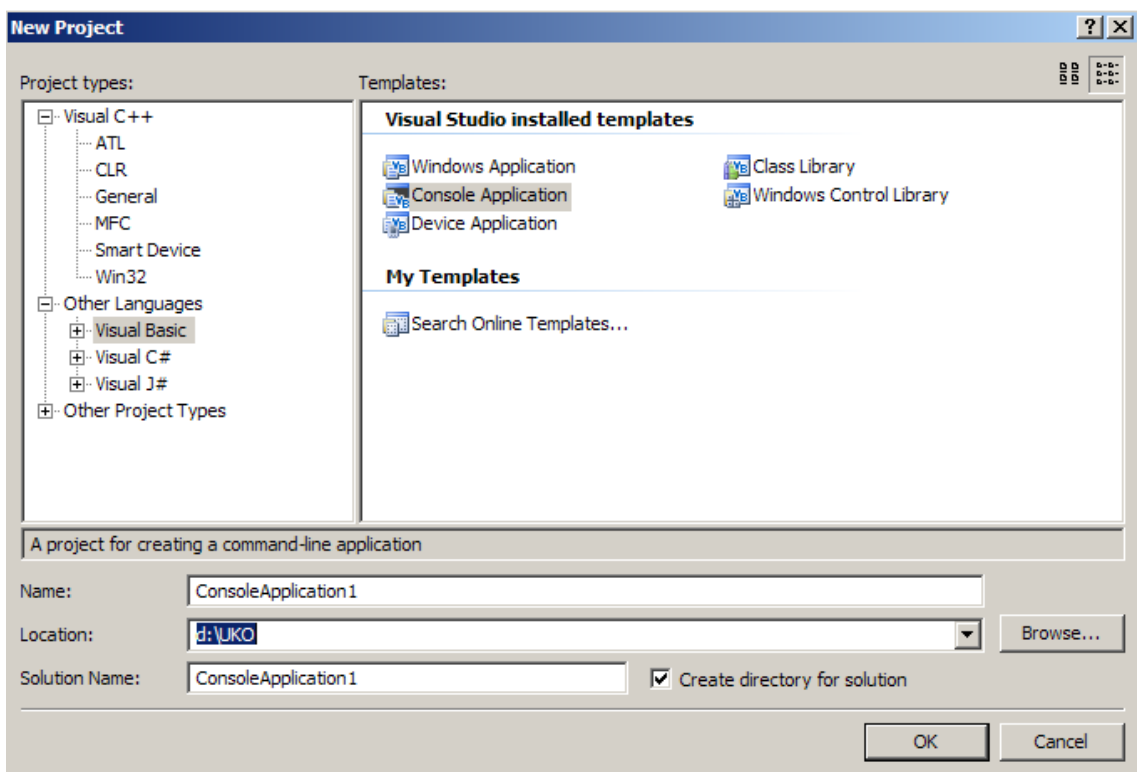
Tabela 1.1.1. Zestawienie typów danych dla VB.NET, C# oraz ich odpowiedniki w CTS

VB.Net	C#	.NET	Rozmiar	Komentarz
Boolean	bool	Boolean	1	wartość logiczna (prawda lub fałsz)
Char	char	Char	2	znak Unicode
<b>Liczby całkowite</b>				
Byte	byte	Byte	1	8-bitowa liczba całkowita bez znaku
SByte	sbyte	SByte	1	8-bitowa liczba całkowita ze znakiem
Short	short	Int16	2	16-bitowa liczba całkowita ze znakiem
Integer	int	Int32	4	32-bitowa liczba całkowita ze znakiem
Long	long	Int64	8	64-bitowa liczba całkowita ze znakiem
<b>Liczby zmiennoprzecinkowe</b>				
Single	float	Single	4	pojedynczej precyzji
Double	double	Double	8	podwójnej precyzji
Decimal	decimal	Decimal	12	96-bitowa liczba dziesiętna

## 1.2. Aplikacje konsolowe

### 1.2.1. Pierwsze aplikacje w VB.NET

Pierwsza aplikacja środowiska .NET zostanie przedstawiona na przykładzie języka Visual Basic .NET. Po uruchomieniu środowiska z menu File wybieramy opcję New, a następnie Project.



Rys. 1.2.1. Tworzenie nowego projektu

Na ekranie powinno pojawić się okienko przedstawione na Rys. 1.2.1 (wygląd okienka może być różny w zależności od wersji środowiska Visual Studio tutaj przedstawiono okienko w wersji 2005). Wybieramy Visual Basic z listy Project Types, następnie wybieramy Console Application z listy Templates. W pole Name wpisujemy nazwę aplikacji, w pozycji Location wybieramy położenie projektu na dysku i zatwierdzamy guzikiem OK.

W wyniku zostanie wygenerowany następujący szablon aplikacji konsolowej dla języka Visual Basic .NET.

```
Module Module1
    Sub Main()

        End Sub
End Module
```

Procedura `Sub Main` jest punktem rozpoczęcia wykonywania aplikacji konsolowej w języku `VB.NET`. Dopiszmy linijkę kodu do wygenerowanego szkieletu aplikacji:

```
Module Module1
    Sub Main()
        Console.WriteLine("Hello World!")
    End Sub
End Module
```

Aby skompilować oraz wykonać aplikację z menu `Debug` wybieramy opcję `Start`, lub aby wykonać aplikację bez debugowania wciskamy kombinację `Ctrl+F5`. Powinniśmy zobaczyć okienko konsoli zawierające słowa `Hello World!`, a aby zakończyć działanie aplikacji naciskamy dowolny klawisz.

Teraz przejdziemy do trochę bardziej interaktywnej aplikacji, której zadaniem będzie zapytanie użytkownika o imię, a następnie przywitanie go używając wprowadzoną przez użytkownika informację.

```
Module Module1
    Sub Main()
        Console.Write("Jak masz na imię? ")
        Dim name As String = Console.ReadLine()
        Console.WriteLine("Witaj {0}", name)
    End Sub
End Module
```

W powyższym przykładzie do wypisania tekstu zamiast metody `WriteLine` klasy `Console` użyliśmy metody `Write`. Różnica polega na tym, że po wypisaniu tekstu kursor teraz nie przechodzi do nowej linii. Kolejna linia programu zawiera deklarację zmiennej `name` typu `String`, na której przypisujemy wartość zwróconą przez metodę `ReadLine()` klasy `Console`. Zmienne mogą przechowywać różne typy danych, np. zmienna typu `Integer` może przechowywać liczby całkowite, typ `String` natomiast przechowuje zmienne w postaci łańcucha znaków. Ostatnia linijka kodu wypisuje tekst na ekranie używając metody `WriteLine`, zastępując występujący w napisie znacznik `{0}` wartością zmiennej przekazanej jako pierwszy parametr (w naszym przypadku `name`).

```
Console.WriteLine("Witaj {0}", name)
```

Jeśli pierwszym parametrem metody `WriteLine` w przekazanym napisie jest znacznik `{n}`, to kompilator zastępuje go wartością `(n+1)` zmiennej występującej po napisie – w naszym przypadku znacznik `{0}` zastępowany jest `(0+1)`-ą zmienną, czyli `name`. Jeżeli zmienna `name` zawiera np. napis `Janek`, to CLR podczas wykonania programu zinterpretuje wywołanie tej funkcji jako:

```
Console.WriteLine("Witaj Janek")
```

Linia kodu zawierająca słowo kluczowe Dim

```
Dim name As String = Console.ReadLine()
```

spowoduje zadeklarowanie zmiennej o nazwie `name` typu `String`, natomiast znak `=` oznacza w tym przypadku instrukcję przypisania, gdzie zmiennej `name` zostaje przypisana wartość zwracana przez metodę `ReadLine()`, a w naszym przypadku będzie to tekst wprowadzony z klawiatury przez użytkownika.

W języku `Visual Basic .Net` istnieje również możliwość deklarowania stałych. Różnica w stosunku do deklarowania zmiennych jest taka, że wartość stałej w trakcie realizacji programu nie może ulec zmianie, czyli nie można jej przypisać żadnej nowej wartości, natomiast wartość przypisywana jest już w momencie deklaracji. Na przykład, deklaracja stałej `PI` może wyglądać następująco:

```
Dim PI As Double = 3.1415
```

Na zmiennych typu liczbowego możemy wykonywać operacje matematyczne. Cztery podstawowe operatory to: `+`, `-`, `*`, `/`. Napiszmy teraz aplikację, która pobierze od użytkownika dwie wartości, a następnie wyświetli wynik działania przedstawionych operatorów matematycznych na tych wartościach.

Zakładamy, że wprowadzany przez użytkownika tekst jest liczbą, a więc zakładamy w tym momencie, że wprowadzone dane będą poprawne (w przeciwnym razie program zgłosi nam wyjątek, czyli błąd). Sprawą kontroli poprawności wprowadzanych danych i obsługą wyjątków zajmiemy się później.

```
Module Module1
  Sub Main()
    Dim liczba1 As Double
    Dim liczba2 As Double
    Console.Write("Podaj pierwszą liczbę: ")
    liczba1 = Console.ReadLine()
    Console.Write("Podaj drugą liczbę: ")
    liczba2 = Console.ReadLine()
    Console.WriteLine("{0} + {1} = {2}", -
      liczba1, liczba2, liczba1 + liczba2)
    Console.WriteLine("{0} - {1} = {2}", -
      liczba1, liczba2, liczba1 - liczba2)
    Console.WriteLine("{0} * {1} = {2}", -
      liczba1, liczba2, liczba1 * liczba2)
    Console.WriteLine("{0} / {1} = {2}", -
      liczba1, liczba2, liczba1 / liczba2)
  End Sub
End Module
```

Zauważmy pojawienie się znaku `_` w metodzie `WriteLine`. Znak ten oznacza w `Visual Basic`'u, że chcemy złamać wiersz i kontynuować instrukcję w nowym wierszu. Zapis bez znaku `_` będzie traktowany jako błędny.

```
Console.WriteLine("{0} + {1} = {2}",  
    liczba1, liczba2, liczba1 + liczba2)
```

Powyższa linia kodu spowoduje zgłoszenie błędu składni, edytor podświetli miejsce wystąpienia błędu.

### 1.2.2. Pierwsze aplikacje w C#

Teraz zajmiemy się językiem `C#` i postaramy się napisać przedstawione do tej pory programy z wykorzystaniem tego języka.

Rozpoczniemy standardowo od aplikacji wypisującej na konsoli tekst `Hello World!`. W tym celu musimy utworzyć nowy projekt, tym razem wybierając język `C#`. Zostanie wygenerowany szkielet aplikacji konsolowej w języku `C#`, który możemy, podobnie jak w poprzednim przypadku języka `VB .NET`, uzupełnić o linię zawierającą kod wypisujący tekst na konsoli:

```
using System;  
using System.Collections.Generic;  
using System.Text;  
  
namespace ConsoleApplication1 {  
    class Program  
    {  
        static void Main(string [] args)  
        {  
            Console.WriteLine("Hello World!");  
        }  
    }  
}
```

Zwróćmy uwagę, że w odróżnieniu od `VB .NET`, linijka zawierająca instrukcję zakończona jest średnikiem.

Kolejną modyfikacją będzie dodanie interakcji z użytkownikiem, czyli napiszemy analogiczny program jak w przypadku `VB .NET`, który zapyta użytkownika o imię i następnie wyświetli odpowiednie powitanie zawierające wprowadzony przez użytkownika tekst.



```

namespace ConsoleApplication1 {
    class Program
    {
        static void Main(string [] args)
        {
            Console.WriteLine("Jak masz na imię ? ");
            string name = Console.ReadLine();
            Console.WriteLine("Witaj {0}", name);
        }
    }
}

```

Jedyna istotna różnica w stosunku do kodu napisanego w VB .NET polega na sposobie deklarowania zmiennych. W przypadku C# zmienną name deklarujemy podając najpierw nazwę typu (tutaj string), a następnie nazwę zmiennej.

Kolejny przykład:

```

namespace ConsoleApplication1 {
    class Program
    {
        static void Main(string [] args)
        {
            double liczba1;
            double liczba2;
            Console.WriteLine("Podaj pierwszą liczbę: ");
            liczba1 = double.Parse(Console.ReadLine());
            Console.WriteLine("Podaj drugą liczbę: ");
            liczba2 = double.Parse(Console.ReadLine());
            Console.WriteLine("{0} + {1} = {2}",
                liczba1, liczba2, liczba1 + liczba2);
            Console.WriteLine("{0} - {1} = {2}",
                liczba1, liczba2, liczba1 - liczba2);
            Console.WriteLine("{0} * {1} = {2}",
                liczba1, liczba2, liczba1 * liczba2);
            Console.WriteLine("{0} / {1} = {2}",
                liczba1, liczba2, liczba1 / liczba2);
        }
    }
}

```

Tak samo jak w przypadku VB .NET, dla typów liczbowych dostępne są podstawowe operatory, czyli możemy przepisać program pobierający dwie liczby i wypisujący wynik operacji dodawania, odejmowania, mnożenia oraz dzielenia.

Różnica w zapisie pomiędzy kodem VB .NET oraz C# jest widoczna przy instrukcjach pobierających dane od użytkownika.

```
liczba1 = double.Parse(Console.ReadLine());
```

Ponieważ wartość zwracana przez metodę `ReadLine()` jest typu `string`, pisząc program w `C#` musimy dokonać konwersji pobranej wartości na tym `double`, w przypadku `VB .NET` konwersja ta była dokonana niejawnie.

W celu dokonania konwersji skorzystamy z metody `Parse`, zdefiniowanej na typie `double`, która stara się przekonwertować napis na liczbę. Oczywiście jest, że w przypadku wprowadzenia napisu, który nie może zostać poprawnie przekonwertowany na liczbę, program podczas wykonania zgłosi wyjątek.

## 1.3. Operatory relacji

Instrukcje warunkowe wykonywane są w zależności od wyniku jaki przyjmuje pewne wyrażenie logiczne. Ważną grupą operatorów są operatory relacji, pozwalające określić w jakiej relacji są ze sobą zmienne. Wynikiem działania operatora relacji jest wartość `true` lub `false`.

### 1.3.1. Operatory relacji w `VB .NET`

Język `VB .NET` dostarcza sześć podstawowych operatorów relacji:

Tabela 1.3.1. Zestawienie podstawowych operatorów relacji dla `VB .NET`

Operator	Znaczenie
=	sprawdzenie równości
>	większe niż
>=	większe równe
<>	różne
<	mniejsze niż
<=	mniejsze równe

Przykładowe działanie operatorów relacji:

<code>Dim num1 As Integer = 1, num2 As Integer = 4</code>			
...			
<code>num1 = num2</code>	wynik <code>false</code>		<code>num1 &lt; num2</code> wynik <code>true</code>
<code>num1 &lt;&gt; num2</code>	wynik <code>true</code>		<code>num1 &lt;= num2</code> wynik <code>true</code>
<code>num1 &gt; num2</code>	wynik <code>false</code>		<code>num1 &gt;= num2</code> wynik <code>false</code>

Operatory relacji mogą być stosowane tylko do kompatybilnych typów, a więc np. nie można dokonać porównania zmiennej typu `Integer` ze zmienną typu `Boolean`.

### 1.3.2. Operatory relacji w `C#`

Język `C#` dostarcza również sześć podstawowych operatorów relacji:

Tabela 1.3.2. Zestawienie podstawowych operatorów relacji dla C#

Operator	Znaczenie
==	sprawdzenie równości
>	większe niż
>=	większe równe
!=	różne
<	mniejsze niż
<=	mniejsze równe

Przykładowe działanie operatorów relacji:

```
int num1 = 1, num2 = 4;
...
num1 == num2    wynik false
num1 != num2    wynik true
num1 > num2     wynik false
num1 < num2     wynik true
num1 <= num2   wynik true
num1 >= num2   wynik false
```

## 1.4. Operatory logiczne i bitowe

Operatory logiczne i bitowe są używane do wyliczania wartości wyrażeń logicznych, oraz do operacji logicznych na bitach.

### 1.4.1. Operatory logiczne i bitowe w VB .NET

Zestawienie operatorów dla języka Visual Basic .NET:

Tabela 1.4.1. Zestawienie podstawowych operatorów logicznych dla VB .NET

Operator	Znaczenie
And	logiczne lub bitowe AND
Or	logiczne lub bitowe OR
Xor	logiczne lub bitowe XOR
Not	logiczne lub bitowe NOT
AndAlso	logiczne AND forma z leniwym wyliczaniem wartości
OrElse	logiczne OR forma z leniwym wyliczaniem wartości

Jeżeli operator zostanie zastosowany do zmienny typu `Boolean` wynikiem będzie wartość `true` lub `false`, natomiast zastosowanie operatora do liczb typu `Integer` zwróci jako wynik liczbę bo zastosowaniu operatora logicznego na poszczególnych bitach operandów (nie dotyczy operatorów `AndAlso` oraz `OrElse`).

Operatory `AndAlso` oraz `OrElse` stosują tak zwaną *leniwą* metodę wyliczania wartości logicznej. W przypadku operatora `AndAlso`, jeżeli pierwszy operand posiada

wartość **false**, to drugi nie jest już liczony i wynikiem jest oczywiście wartość **false**. W przypadku operatora **OrElse**, jeżeli pierwszy operand przyjmuje wartość **true**, to drugi nie jest wyliczany i całe wyrażenie przyjmuje wartość **true**. Przykłady użycia operatorów logicznych:

```
Dim num1 As Double = 1, num2 As Double = 3
Dim b As Boolean = false
...
(num1 > num2) And (num2 > 1)    wynik false
(num1 = num2) Or (num1 < 2)     wynik true
Not b                            wynik true
```

Przykłady użycia operatorów bitowych:

```
Dim num1 As Double = 1, num2 As Double = 3
...
num1 And num2    wynik 1
num1 Or num2     wynik 3
num1 Xor num2    wynik 2
```

Zastosowanie operatorów bitowych można zobrazować w następujący sposób: liczba 1 w zapisie binarnym reprezentowana jest jako 01, natomiast liczba 3 jako 11. Wykonanie operacji **And** da w wyniku pierwszy bit nowej liczby 0 **And** 1, czyli 0, oraz drugi bit nowej liczby jako 1 **And** 1, czyli 1. Nowa liczba wynosi więc 1. Dla operatora **Or** postępowanie jest analogiczne, z tym, że 0 **Or** 1 daje 1, 1 **Or** 1 daje 1, więc nowa liczba w zapisie binarnym to 11, czyli w zapisie dziesiętnym 3.

### 1.4.2. Operatory logiczne i bitowe w C#

Zestawienie operatorów dla języka C#:

Tabela 1.4.2. Zestawienie podstawowych operatorów logicznych dla C#

Operator	Znaczenie
&&	logiczne AND
	logiczne OR
&	bitowe AND
	bitowe OR
^	bitowe XOR
!	bitowe NOT

Przykłady użycia operatorów logicznych i bitowych:

```
int num1 = 1, num2 = 3;
bool b = false;
...
num1 & num2           wynik 1
(num1 > num2) || !b   wynik true
```

## 1.5. Instrukcje warunkowe

Przedstawione do tej pory przykłady programów były mało ciekawe, ponieważ wynik ich działania był z góry znany. W praktycznym programowaniu nie można obejść się bez instrukcji warunkowych, które pozwalają na wykonanie odpowiednich fragmentów kodu w zależności od spełnienia, lub nie, pewnych warunków logicznych.

### 1.5.1. Instrukcje warunkowe w VB .NET

Podstawową instrukcją warunkową w języku VB .NET to instrukcja `If`. Składnia tej instrukcji wygląda następująco:

```
If <warunek logiczny> Then
    <blok instrukcji>
Else
    <blok instrukcj>
End If
```

Klauzula `Else` przedstawiona powyżej jest opcjonalna, a typowy przykład zastosowania instrukcji `If` ma postać

```
If i=5 Then
    Console.WriteLine("zmienna i przyjęła wartość 5")
End If
```

Przedstawiony powyżej kod wypisze na konsoli komunikat tylko w przypadku, kiedy zmienna `i` przyjmie wartość 5, czyli gdy wyliczony warunek logiczny `i = 5` przyjmie wartość `true`.

```
If i=5 Then
    Console.WriteLine("zmienna i przyjęła wartość 5")
Else
    Console.WriteLine("zmienna i nie przyjęła wartość 5")
End If
```

Powyższy kod w zależności od spełnienia warunku logicznego `i = 5`, lub jego nie spełnienia, wypisze na konsoli odpowiedni tekst.

Kolejną wersją instrukcji warunkowej If jest zastosowanie klauzuli ElseIf.

```
If i=5 Then
    Console.WriteLine("zmienna i przyjęła wartość 5")
ElseIf i=6
    Console.WriteLine("zmienna i nie przyjęła wartość 6")
Else
    Console.WriteLine("zmienna i nie przyjęła ani wartości 5 ani 6")
End If
```

Ponieważ konstrukcja If ...Then ...Else jest także instrukcją, to można ją zagnieździć w innej instrukcji If.

```
If i>5 Then
    If i=6 Then
        Console.WriteLine("zmienna i przyjęła wartość 6")
    Else
        Console.WriteLine("zmienna i jest > 5 ale <> 6")
    End If
Else
    Console.WriteLine("zmienna i przyjęła wartość <= 5")
End If
```

### Konstrukcja Select ... Case

Jeżeli trzeba dokonać sprawdzenia zajścia jakiś warunków dla pewnej zmiennej, to zamiast konstrukcji If ... Then ... ElseIf, wygodnie jest zastosować konstrukcję Select Case

```
Select <zmienna typu podstawowego>
    Case <wyrażenie1>
        <instrukcje>
    Case <wyrażenie2>
        <instrukcje>
        ....
        <inne bloki Case>
    Case Else
        <instrukcje>
End Select
```

Przykładowe zastosowanie konstrukcji `Select Case`:

```
Module Module1
    Sub Main()
        Dim i As Integer
        i = Console.ReadLine()
        Select Case i
            Case 1
                Console.WriteLine("Wprowadzono 1")
            Case 2
                Console.WriteLine("Wprowadzono 2")
            Case 3 To 5
                Console.WriteLine("Wprowadzono wartość <3,5>")
            Case Else
                Console.WriteLine("Wprowadzona wartość > 5 lub < 1")
        End Select
    End Sub
End Module
```

Kiedy użytkownik wprowadzi 1 wykona się pierwszy blok: `Case` wypisze tekst `wprowadzono 1`. Podobnie będzie dla 2. W przypadku wpisania 3, 4 lub 5, wykona się trzeci blok `Case`, natomiast kiedy wprowadzona liczba będzie mniejsza od 1 lub większa od 5, wykonana zostanie ostatnia klauzula, czyli `Case Else`.

### 1.5.2. Instrukcje warunkowe w C#

```
if (<warunek logiczny >){
    <blok instrukcji >
}
else{
    <blok instrukcj >
}
```

Typowe zastosowanie konstrukcji `If`.

```
if (i == 5)
    Console.WriteLine("Zmienna i przyjęła wartość 5");
```

W przypadku, kiedy po instrukcji `if` występuje tylko jedna instrukcja, można opuścić nawiasy klamrowe. Używane są one tylko w sytuacji, gdy w momencie spełnienia warunku wykonany ma być blok kilku instrukcji.

```
if (i == 5)
    Console.WriteLine("Zmienna i przyjęła wartość 5");
else
    Console.WriteLine("Zmienna i nie przyjęła wartości 5");
```

Powyższy kod w zależności od spełnienia warunku logicznego  $i = 5$ , lub jego nie spełnienia, wypisze na konsoli odpowiedni tekst.

Zagnieżdżenie instrukcji `if`:

```
if (i > 5)
{
    if (i == 6)
        Console.WriteLine("Zmienna i przyjęła wartość 6");
    else
        Console.WriteLine("Zmienna i jest > 5 ale < 6");
}
else
    Console.WriteLine("Zmienna i przyjęła wartość <= 5");
```



## Konstrukcja switch

Odpowiednikiem konstrukcji `Select Case` w VB .NET jest w języku C# konstrukcja `switch`.

```
switch (<zmienna typu podstawowego>)
{
    case <wyrażenie1>:
        <instrukcje>;
        break;
    case <wyrażenie2>:
        <instrukcje>;
        break;
    ....
    <inne bloki case>
    default:
        <instrukcje>;
        break;
}
```

Przykład zastosowania konstrukcji `switch`:

```
i = int.Parse(Console.ReadLine());
switch (i)
{
    case 1:
        Console.WriteLine("Wprowadzono 1");
        break;
    case 2:
        Console.WriteLine("Wprowadzono 2");
        break;
    case 3:
    case 4:
    case 5:
        Console.WriteLine("Wprowadzono wartość <3,5>");
        break;
    default:
        Console.WriteLine("Wprowadzona wartość > 5 lub < 1");
        break;
} //koniec bloku switch
```

Różnice składniowe pomiędzy konstrukcjami `Select Case` a `switch` w VB .NET oraz w C# można łatwo zauważyć analizując przykłady. Parę słów komentarza wymaga zastosowanie słowa kluczowego `break` w konstrukcji `switch`. Kiedy spełniony jest warunek dla jednej z klauzul `case`, (np. założmy że zmienna `i` posiada wartość 2, wypisany zostanie więc napis: `wprowadzono 2`). Gdyby usunąć słowo kluczowe `break`, wystąpiłby błąd składni (dla osób programujących w C lub w C++: brak słowa `break` w konstrukcji `switch` języka C lub C++ spowodowałby wykonanie kolejnych klauzul `case`, tak jakby warunek był spełniony).

---

## Rozdział 2

---

# Laboratorium 2

### 2.1. Łańcuchy znaków w .NET

Biblioteka `.NET Framework` dostarcza narzędzia do pracy z łańcuchami znaków. Narzędzia te są wspólne dla wszystkich języków środowiska `.NET`, między innymi dla `VB.NET` oraz `C#`.

#### 2.1.1. Porównywanie napisów

Metoda `Compare` klasy `String` dokonuje porównania dwóch napisów i zwraca wynik w postaci liczby całkowitej. Zwrócona wartość ma następujące znaczenie:

Tabela 2.1.1. Znaczenie wartości zwracanych przez metodę `Compare`

wartość	znaczenie
mniejsza od zera	pierwszy napis jest mniejszy od drugiego
zero	napisy są sobie równe
większe od zera	pierwszy napis jest większy od drugiego

Inną możliwością jest użycie metody `CompareTo` dla zmiennej typu `String` (metoda umożliwia porównanie obiektu z napisem). Zwracana wartość jest identyczna jak w przypadku metody `Compare`

## Przykładowy kod VB.NET

Przykładowy kod porównuje dwa napisy i wyświetla wynik na konsoli:

```
Sub Main()  
    Dim napis1 As String  
    Dim napis2 As String  
    Dim wynik As Integer  
    Console.WriteLine("Podaj pierwszy napis: ")  
    napis1 = Console.ReadLine()  
    Console.WriteLine("Podaj drugi napis: ")  
    napis2 = Console.ReadLine()  
    wynik = String.Compare(napis1, napis2)  
    Console.WriteLine("Wynik porównania : {0}", wynik)  
End Sub
```

## Zastosowanie metody CompareTo

```
Sub Main()  
    Dim napis1 As String  
    Dim napis2 As String  
    Dim wynik As Integer  
    Console.WriteLine("Podaj pierwszy napis: ")  
    napis1 = Console.ReadLine()  
    Console.WriteLine("Podaj drugi napis: ")  
    napis2 = Console.ReadLine()  
    wynik = napis1.CompareTo(napis2)  
    Console.WriteLine("Wynik porównania : {0}", wynik)  
End Sub
```

## Przykładowy kod dla C#

Przykładowy kod porównuje dwa napisy i wyświetla wynik na konsoli:

```
static void Main(string[] args)  
{  
    string napis1;  
    string napis2;  
    int wynik;  
    Console.WriteLine("Podaj pierwszy napis: ");  
    napis1 = Console.ReadLine();  
    Console.WriteLine("Podaj drugi napis: ");  
    napis2 = Console.ReadLine();  
    wynik = String.Compare(napis1, napis2);  
    Console.WriteLine("Wynik porównania : {0}", wynik);  
}
```

### Zastosowanie metody CompareTo

```
static void Main(string [] args)
{
    string napis1;
    string napis2;
    int wynik;
    Console.WriteLine("Podaj pierwszy napis: ");
    napis1 = Console.ReadLine();
    Console.WriteLine("Podaj drugi napis: ");
    napis2 = Console.ReadLine();
    wynik = napis1.CompareTo(napis2);
    Console.WriteLine("Wynik porównania : {0}", wynik);
}
```

#### 2.1.2. Kopiowanie i konkatencja łańcuchów znaków

Metoda `Concat` dodaje do siebie dwa napisy (dopisuje drugi na końcu pierwszego) i jako wynik zwraca napis będący sklejeniem dwóch napisów. Metoda `Copy` kopiuje zawartość jednego napisu do drugiego.

#### Przykładowy kod VB.NET

Przykładowy kod sklejący dwa napisy:

```
Sub Main()
    Dim napis1 As String
    Dim napis2 As String
    napis1 = "abcde"
    napis2 = "123"
    napis1 = String.Concat(napis1, napis2)
    Console.WriteLine(napis1)
End Sub
```

Przykładowy kod kopujący napis:

```
Sub Main()
    Dim napis1 As String
    Dim napis2 As String
    napis1 = "abcde"
    napis2 = String.Copy(napis1)
    Console.WriteLine(napis2)
End Sub
```

### Przykładowy kod C#

Przykładowy kod sklejający dwa napisy:

```
static void Main(string [] args)
{
    string napis1;
    string napis2;
    napis1 = "abcd";
    napis2 = "123";
    napis1 = string.Concat(napis1, napis2);
    Console.WriteLine(napis1);
}
```

Przykładowy kod kopiujący napis:

```
static void Main(string [] args)
{
    string napis1;
    string napis2;
    napis1 = "abcde";
    napis2 = String.Copy(napis1);
    Console.WriteLine(napis2);
}
```

#### 2.1.3. Dodawanie, usuwanie oraz zastępowanie ciągów znaków

Metoda `Insert` wstawia jeden napis wewnątrz drugiego na podanej pozycji. Metoda `Remove` usuwa określoną ilość znaków zaczynając od podanej pozycji, a następnie zwraca nowy łańcuch znaków będący wynikiem zastosowanej operacji. Metoda `Replace` zastępuje jeden znak innym (w całym napisie)

### Przykładowy kod VB.NET

Metoda `Insert`

```
Sub Main()
    Dim napis1 As String
    Dim napis2 As String
    napis1 = "abcde"
    napis2 = "123"
    napis1 = napis1.Insert(2, napis2)
    Console.WriteLine(napis1)
End Sub
```

Wynikiem jest wypisanie na konsoli napisu `ab123cde`.

Metoda Remove

```
Sub Main()
    Dim napis1 As String
    napis1 = "ab123cde"
    napis1 = napis1.Remove(2, 3)
    Console.WriteLine(napis1)
End Sub
```

Wynikiem jest wyświetlenie na konsoli napisu abcd.

Metoda Replace

```
Sub Main()
    Dim napis1 As String
    napis1 = "ababab"
    napis1 = napis1.Replace("b", "12")
    Console.WriteLine(napis1)
End Sub
```

Wynikiem jest wyświetlenie na konsoli napisu a12a12a12

**Przykładowy kod C#**

Metoda Insert

```
static void Main(string [] args)
{
    string napis1;
    string napis2;
    napis1 = "abcde";
    napis2 = "123";
    napis1 = napis1.Insert(2, napis2);
    Console.WriteLine(napis1);
}
```

Wynikiem jest wypisanie na konsoli napisu ab123cde.

Metoda Remove

```
static void Main(string [] args)
{
    string napis1;
    napis1 = "ab123cde";
    napis1 = napis1.Remove(2, 3);
    Console.WriteLine(napis1);
}
```

Wynikiem jest wyświetlenie na konsoli napisu abcd.

## Metoda Replace

```
static void Main(string [] args)
{
    string napis1;
    napis1 = "ababab";
    napis1 = napis1.Replace("b", "12");
    Console.WriteLine(napis1);
}
```

Wynikiem jest wyświetlenie na konsoli napisu a12a12a12

### 2.1.4. Zmiana wielkości liter

Metoda ToUpper dokonuje konwersji liter w napisie na wielkie, natomiast metoda ToLower na małe litery.

#### Przykładowy kod VB.NET

działanie metod ToUpper oraz ToLower

```
Sub Main()
    Dim napis1 As String = "abcde"
    Dim napis2 As String = "ABCDE"
    napis1 = napis1.ToUpper()
    napis2 = napis2.ToLower()
    Console.WriteLine("ToUpper: {0}, ToLower: {1}", -
        napis1, napis2)
End Sub
```

#### Przykładowy kod C#

działanie metod ToUpper oraz ToLower

```
static void Main(string [] args)
{
    string napis1 = "abcd";
    string napis2 = "ABCD";
    napis1 = napis1.ToUpper();
    napis2 = napis2.ToLower();
    Console.WriteLine("ToUpper: {0}, ToLower: {1}",
        napis1, napis2);
}
```

### 2.1.5. Formatowanie napisów

Przekazując parametry do metody WriteLine lub budując nowy napis korzystając z metody Format klasy String możemy dokładniej sprecyzować jaki sposób wyświe-

tlania wartości nas interesuje. W poniższej tabeli przedstawiono znaczniki dostępne przy formatowaniu liczb:

Tabela 2.1.2. Formatowanie liczb

znacznik	typ	format	wynik 1.2345	wynik 12345
c	waluta	0:c	1,23 zł	12 345,00 zł
d	dziesiętny	0:d	System.FormatException	12345
e	wykładniczy	0:e	1,234500e+000	1,234500e+004
f	kropka dziesiętna	0:f	1,23	12345,0
g	ogólny	0:g	1,2345	12345
n	liczba	0:n	1,23	12 345,00
x	szesnastkowy	0:x	System.FormatException	3039

Można również tworzyć własne wzorce formatowania liczb.

Tabela 2.1.3. Własne znaczniki formatowania liczb

znacznik	typ	format	wynik 1234.56
0	zero lub cyfra	0:00.000	1234,560
#	cyfra	0:#.##	1234.56
.	kropka dziesiętna	0:0.0	1234,6
,	separator tysięcy	0:0,0	1 235

### 2.1.6. Formatowanie daty

Znaczniki definiujące format daty są zależne od ustawień międzynarodowych w systemie. Poniższa tabela prezentuje dostępne znaczniki formatowania daty i wynik ich działania dla systemu polskiego.

Tabela 2.1.4. Znaczniki formatowania daty

znacznik	typ	wynik 2006-10-11 0:58:09
d	krótki format daty	2006-10-11
D	długi format daty	10 listopada 2006
t	krótki format godziny	0:58
T	długi format godziny	0:58:09
f	data i czas	10 listopada 2006 0:58
F	data i czas pełny	10 listopada 2006 0:58:09
g	domyślny format daty	2006-10-11 0:58
G	domyślny format daty długi	2006-10-11 0:58:09
M	dzień / miesiąc	10 listopada
r	format zgodny z RFC1123	Fri, 10 Nov 2006 00:58:09 GMT
Y	miesiąc / rok	listopad 2006



Tak jak w przypadku liczb istnieje możliwość zdefiniowania własnego formatu daty z wykorzystaniem znaczników przedstawionych w poniższej tabeli:

Tabela 2.1.5. Znaczniki formatowania daty

znacznik	typ	wynik 2006-10-11 22:58:09
dd	dzień	10
ddd	krótka nazwa dnia	Pt
dddd	długa nazwa dnia	piątek
hh	godzina	10
GG	godzina format 24	22
mm	minuty	58
MM	miesiąc	11
MMM	miesiąc krótka nazwa	lis
MMMM	miesiąc długa nazwa	listopad
ss	sekundy	09
yy	rok 2 cyfry	06
yyyy	rok 4 cyfry	2006
:	separator np 0:hh:mm:ss	22:58:09
/	separator np 0:dd/MM/yyyy	10-11-2006

### Przykładowy kod dla VB.NET

Przykład użycia znaczników formatowania:

```
Sub Main()
    Dim liczba As Double = 1234.56
    Dim napis As String
    Console.WriteLine("Format liczby 0:c {0:c}", liczba)
    Console.WriteLine("Format liczby 0:n {0:n}", liczba)
    Console.WriteLine("Format liczby 0:f {0:f}", liczba)
    Console.WriteLine("Format liczby 0:e {0:e}", liczba)
    Console.WriteLine("Format liczby 0:e {0:##,###.##0}", liczba)
    napis = String.Format("{0:e}", liczba)
    Console.WriteLine(napis)
    Console.WriteLine()
    Dim data As Date = DateTime.Now
    Console.WriteLine("Format daty 0:d {0:d}", data)
    Console.WriteLine(" -
        "Format daty 0:dddd MM yyyy: {0:dddd MM yyyy}", -
        data)
End Sub
```

## Przykładowy kod dla C#

Przykład użycia znaczników formatowania:

```
static void Main(string [] args)
{
    double liczba = 1234.56;
    string napis;
    Console.WriteLine("Format liczby 0:c {0:c}", liczba);
    Console.WriteLine("Format liczby 0:n {0:n}", liczba);
    Console.WriteLine("Format liczby 0:f {0:f}", liczba);
    Console.WriteLine("Format liczby 0:e {0:e}", liczba);
    Console.WriteLine("Format liczby 0:e {0:##,###.##0}", liczba);
    napis = string.Format("{0:e}", liczba);
    Console.WriteLine(napis);
    Console.WriteLine();
    DateTime data = DateTime.Now;
    Console.WriteLine("Format daty 0:d {0:d}", data);
    Console.WriteLine(
        "Format daty 0:dddd MM yyyy: {0:dddd MM yyyy}",
        data);
}
```

### 2.1.7. Wycinanie białych znaków

Metoda Trim usuwa białe znaki z początku i końca napisu.

## Przykładowy kod dla VB.NET

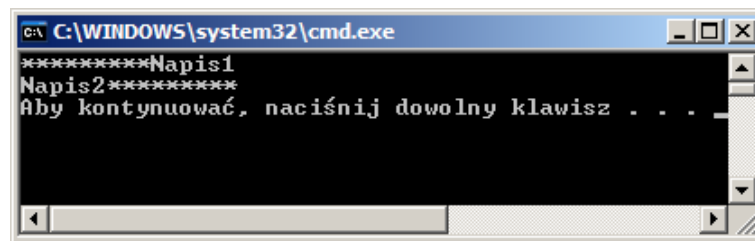
```
Sub Main()
    Dim napis As String = " Napis "
    Console.WriteLine(napis)
    napis = napis.Trim()
    Console.WriteLine(napis)
End Sub
```

## Przykładowy kod dla C#

```
static void Main(string [] args)
{
    string napis = " Napis ";
    Console.WriteLine(napis);
    napis = napis.Trim();
    Console.WriteLine(napis);
}
```

### 2.1.8. Dopełnianie łańcuchów znaków

Dwie metody `PadLeft` oraz `PadRight`, mogą zostać użyte w celu dopełnienia łańcucha znaków, odpowiednio z lewej lub prawej strony. Działanie metod jest takie, że jako parametr podawana jest długość łańcucha wynikowego oraz znak którym ma zostać wypełniony łańcuch, wynikiem działania jest łańcuch znaków o podanej długości natomiast brakujące elementy zostają wypełnione podanym znakiem.



Rys. 2.1.1. Wynik działania metody `PadLeft` oraz `PadRight`

#### Przykładowy kod dla VB.NET

```
Sub Main()
    Dim napis1 As String = "Napis1"
    Dim napis2 As String = "Napis2"
    napis1 = napis1.PadLeft(15, "*")
    napis2 = napis2.PadRight(15, "*")
    Console.WriteLine(napis1)
    Console.WriteLine(napis2)
End Sub
```

#### Przykładowy kod dla C#

```
static void Main(string [] args)
{
    string napis1 = "Napis1";
    string napis2 = "Napis2";
    napis1 = napis1.PadLeft(15, '*');
    napis2 = napis2.PadRight(15, '*');
    Console.WriteLine(napis1);
    Console.WriteLine(napis2);
}
```

Zauważmy, że w odróżnieniu od VB.NET w przypadku C# drugi parametr metod `Pad` zawierający znak przekazywany jest nie w cudzysłowie `"`, ale pomiędzy znakami apostrofu (jest to informacja, że wartość jest typu `char`).

### 2.1.9. Klasa `StringBuilder`

Klasa `StringBuilder` reprezentuje zmienny łańcuch znaków. Może on być modyfikowany przy pomocy metod `Append`, `Insert`, `Remove` oraz `Replace`.

Kiedy budujemy napis wykonując wiele różnych operacji np. konkatencji, zamiany znaków, ze względów wydajności lepiej jest stosować klasę `StringBuilder` zamiast klasy `String`.

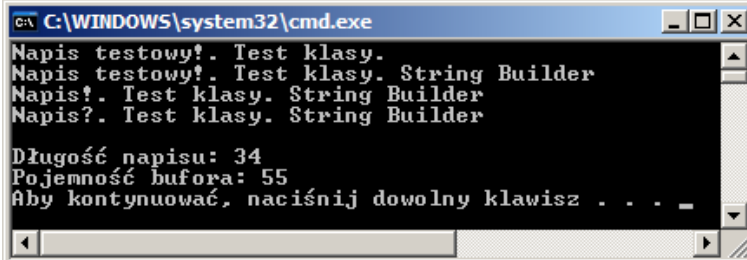
Klasa `StringBuilder` jest zdefiniowana w przestrzeni nazw `System.Text`.

Tabela 2.1.6. Właściwości klasy `StringBuilder`

Właściwość	Opis
<code>Capacity</code>	Reprezentuje maksymalną ilość znaków, która może być przechowywana
<code>Chars</code>	Znak na określonej pozycji
<code>Length</code>	Ilość znaków
<code>MaxCapacity</code>	Zwraca maksymalną pojemność

Tabela 2.1.7. Metody klasy `StringBuilder`

Metoda	Opis
<code>Append</code>	Dopisuje łańcuch znaków na końcu
<code>AppendFormat</code>	Dopisuje sformatowany łańcuch znaków na końcu
<code>EnsureCapacity</code>	Ustawienie pojemności
<code>Append</code>	Dopisuje łańcuch znaków na końcu
<code>Insert</code>	Wstawia łańcuch znaków na podanej pozycji
<code>Remove</code>	Usuwa zakres znaków z łańcucha
<code>Replace</code>	Zastępuje wszystkie wystąpienia znaku innym



```

C:\WINDOWS\system32\cmd.exe
Napis testowy!. Test klasy.
Napis testowy!. Test klasy. String Builder
Napis!. Test klasy. String Builder
Napis?. Test klasy. String Builder

Długość napisu: 34
Pojemność bufora: 55
Aby kontynuować, naciśnij dowolny klawisz . . .

```

Rys. 2.1.2. Wynik działania programu korzystającego z klasy `StringBuilder`

## Przykładowy kod dla VB.NET

```
Imports System.Text
Module Module1
    Sub Main()
        Dim builder As StringBuilder
        builder = New StringBuilder("Napis testowy!", 20)
        Dim cap As Integer = builder.EnsureCapacity(55)

        builder.Append(". Test klasy.")
        Console.WriteLine(builder)
        builder.Insert(27, " String Builder")
        Console.WriteLine(builder)
        builder.Remove(5, 8)
        Console.WriteLine(builder)
        builder.Replace("!", "?")
        Console.WriteLine(builder)
        Console.WriteLine()
        Console.WriteLine("Długość napisu: {0}", -
            builder.Length.ToString())
        Console.WriteLine("Pojemność bufora: {0}", -
            builder.Capacity.ToString())

    End Sub
End Module
```

## Przykładowy kod dla C#

```
using System.Text;
...
static void Main(string[] args)
{
    StringBuilder builder;
    builder = new StringBuilder("Napis testowy!", 20);
    int cap = builder.EnsureCapacity(55);

    builder.Append(". Test klasy.");
    Console.WriteLine(builder);
    builder.Insert(27, " String Builder");
    Console.WriteLine(builder);
    builder.Remove(5, 8);
    Console.WriteLine(builder);
    builder.Replace("!", "?");
    Console.WriteLine(builder);
    Console.WriteLine();
    Console.WriteLine("Długość napisu: {0}",
        builder.Length.ToString());
    Console.WriteLine("Pojemność bufora: {0}",
        builder.Capacity.ToString());
}
```

## 2.2. Pętle

Pętle są przydatnymi konstrukcjami języka, szczególnie kiedy trzeba wykonać pewne zadanie iteracyjne, powtórzyć wykonanie pewnego fragmentu kodu kilkakrotnie (zazwyczaj do momentu aż nie zostanie spełniony pewien warunek).

Najbardziej typowym i spotykanym rodzajem pętli jest konstrukcja `for`

### 2.2.1. Konstrukcja `For...Next` VB.NET

Typowe zastosowanie konstrukcji `For` w języku VB.NET

```
For <zmienna> = start To koniec Step <krok>  
    <blok instrukcji>  
Next
```

Gdzie `<zmienna>` – zmienna typu liczbowego reprezentująca licznik pętli zwiększany w każdym kroku domyślnie o jeden, `start` – wartość początkowa, `koniec` – wartość końcowa, `krok` – wartość o jaką będzie zwiększony licznik.

przykład programu wyświetlającego na konsoli liczby kolejno od jeden do dziesięć:

```
Sub Main()  
    Dim i As Integer  
    For i = 1 To 10  
        Console.WriteLine(i)  
    Next  
End Sub
```

Program wyświetlający liczby od dziesięć do jeden:

```
Sub Main()  
    Dim i As Integer  
    For i = 10 To 1 Step -1  
        Console.WriteLine(i)  
    Next  
End Sub
```

W pierwszym przykładzie zmienna `i` typu `Integer` inicjalizowana jest wartością 1, następnie wykonywany jest blok znajdujący się wewnątrz instrukcji `For`, po wykonaniu bloku testowany jest warunek osiągnięcia przez zmienną `i` wartości większej od 10, jeżeli warunek nie jest spełniony, zmienna `i` zwiększana jest o jeden (w tym przykładzie brak słowa kluczowego `Step`) i wykonanie pętli jest kontynuowane z nową wartością zmiennej `i`.

W drugim przykładzie zmienna `i` typu `Integer` inicjalizowana jest wartością 10, następnie wykonywany jest blok znajdujący się wewnątrz instrukcji `For`, po wykonaniu bloku testowany jest warunek osiągnięcia przez zmienną `i` wartości mniejszej od 1,

jeżeli warunek nie jest spełniony, zmienna *i* zmniejszana jest o jeden (w tym przykładzie wartość -1 występująca po słowie **Step** mówi o ile zmienić wartość zmiennej *i*), wykonanie pętli jest kontynuowane z nową wartością zmiennej *i*.

Istnieje możliwość opuszczenia pętli zanim wartość zmiennej *i* osiągnie wartość **koniec**, służy do tego instrukcja **Exit For** (jest to jednak mało elegancki sposób programowania i należy unikać tej konstrukcji, ponieważ pętla **for** wyraża akcję którą wykonujemy założoną z góry ilość razy i wszelkie „sztuczne” przerywanie jej działania oznacza, że powinniśmy zastosować w tym miejscu inną konstrukcję pętli).

Przykład zastosowania instrukcji **Exit For**

```
Sub Main()  
    Dim i As Integer  
    For i = 1 To 10  
        If i > 5 Then  
            Exit For  
        End If  
        Console.WriteLine(i)  
    Next  
End Sub
```

### 2.2.2. Konstrukcja **for** dla C#

Ogólnych schemat konstrukcji **for** dla języka C#:

```
for ( <zmienna_opc>; <warunek_opc>; <iterator_opc> )  
{  
    <blok instrukcji>  
}
```

Konstrukcja **for** dla języka C# jest znacznie bardziej rozbudowana niż w przypadku VB.NET.

<zmienna\_opc> – opcjonalna zmienna, która może być potraktowana jako licznik pętli tak jak w przypadku VB.NET (może w tym miejscu może wystąpić deklaracja zmiennej), <warunek\_opc> – wyrażenie, które zwraca wartość logiczną (zwracana wartość **false** oznacza koniec wykonania pętli), <iterator\_opc> – wyrażenie zwiększające licznik pętli.

Najlepiej zilustrować konstrukcję `for` języka `C#` na przykładzie kodu, proste i oczywiste zastosowanie konstrukcji `for` do wypisania kolejnych liczb od 1 do 10 wersja pierwsza:

```
static void Main(string [] args)
{
    int i;
    for (i=1; i<=10; i++)
    {
        Console.WriteLine(i);
    }
}
```

Ponieważ `<zmienna_opc>` może być wyrażeniem zawierającym deklarację zmiennej, możemy zapisać kod wyświetlający kolejne liczby w następujący sposób:

```
static void Main(string [] args)
{
    for (int i=1; i<=10; i++)
    {
        Console.WriteLine(i);
    }
}
```

Teraz zmienna `i` jest zadeklarowana jako lokalna wewnątrz pętli `for`, próba odwołania się do niej poza pętlą zakończy się błędem informującym o tym, że nazwa `i` nie istnieje w bieżącym kontekście (jest dostępna tylko wewnątrz pętli).

Ponieważ wszystkie składowe konstrukcje `for` są opcjonalne nasz przykład możemy zapisać jeszcze w inny sposób:

```
static void Main(string [] args)
{
    int i=1;
    for (; i<=10; i++)
    {
        Console.WriteLine(i);
    }
}
```

Iterator pętli `<iterator_opc>` jest też opcjonalny, możemy zapisać kod jeszcze w inny sposób:



```
static void Main(string [] args)
{
    int i=1;
    for (; i<=10; )
    {
        Console.WriteLine(i);
        i++;
    }
}
```

lub:

```
static void Main(string [] args)
{
    int i=1;
    for (; i<=10; )
    {
        Console.WriteLine(i++);
    }
}
```

Ponieważ wszystkie parametry konstrukcji `for` są opcjonalne możemy napisać wersję pętli `for` w następujący sposób:

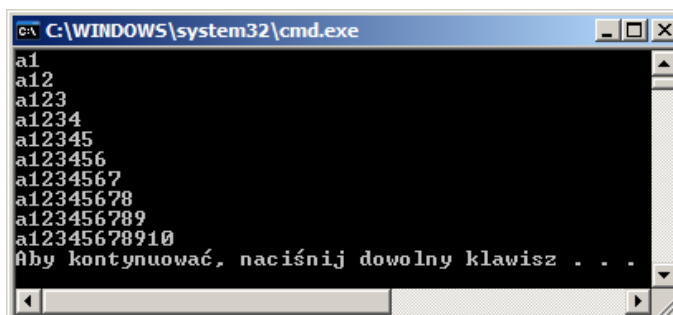
```
static void Main(string [] args)
{
    int i=1;
    for (;;)
    {
        Console.WriteLine(i++);
    }
}
```

Program ten spowoduje wykonanie pętli nieskończonej!, brak wyrażenia `<warunek_opc>` powoduje, że warunek jest zawsze prawdziwy i wykonują się kolejne iteracje.

Zagadka. Co będzie wynikiem wykonania następującego kodu:

```
static void Main(string [] args)
{
    int i = 1;
    for (string a="a"; i<=10; )
    {
        Console.WriteLine(a+=i++);
    }
}
```

Wynik będzie następujący:



```

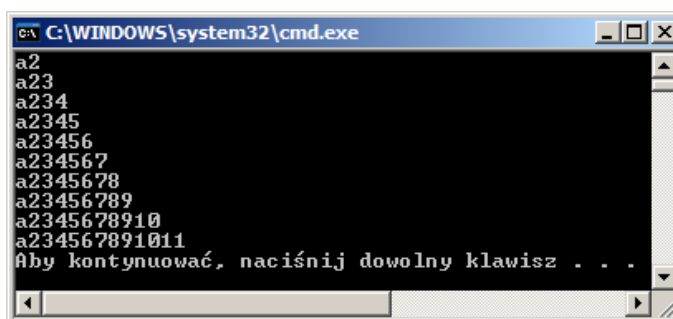
C:\WINDOWS\system32\cmd.exe
a1
a12
a123
a1234
a12345
a123456
a1234567
a12345678
a123456789
a12345678910
Aby kontynuować, naciśnij dowolny klawisz . . .

```

Rys. 2.2.1. Wynik działania kodu zagadki

Troche komentarza. Najpierw deklarujemy zmienną `i` oraz dokonujemy inicjalizacji tej zmiennej wartością 1. W pętli `for` deklarujemy lokalną zmienną `a` typu `string` przypisując jej wartość w postaci łańcucha znaków zawierającego jeden znak, mianowicie literę `a`. Warunkiem zakończenia pętli jest osiągnięcie przez zmienną `i` wartości 10. Wewnątrz pętli wykonywana jest metoda `WriteLine` klasy `Console`, której jako parametr przekazujemy wyrażenie `a+=i++` operator `+=` jest operatorem przypisania (napis `x+=1` jest równoważny napisowi `x=x+1`, ponieważ konstrukcja `i++` powoduje zwrócenie wartości zmiennej `i` a następnie zwiększenie jej o jeden po prawej stronie wyrażenia otrzymujemy jako wynik wartość zmiennej `i` (w pierwszym kroku pętli będzie to 1) następnie operator `+` dla zmiennej typu `string` oznacza operator konkatencji, czyli do łańcucha znaków zawartego w zmiennej `a` zostaje dodany łańcuch znaków reprezentujący „napisową” reprezentację liczby będącej wartością zmiennej `i` (dokonywana jest nie jawna konwersja typu `int` na `string` operacja ta zawsze jest dobrze określona nie tak jak w przypadku odwrotnym), nowa wartość zmiennej `a` zawiera więc napis `"a1"`, który wypisywany jest na konsoli, a następnie wykonanie pętli jest kontynuowane.

Operator `++` zawiera dwie formy przyrostkową, oraz przedrostkową. Powyższy przykład zawiera formę przyrostkową (konstrukcja `i++`). Forma przedrostkowa (postaci `++i`) powoduje, że najpierw zostanie zwiększona wartość zmiennej `i` a dopiero potem zostanie zwrócona jej wartość.



```

C:\WINDOWS\system32\cmd.exe
a2
a23
a234
a2345
a23456
a234567
a2345678
a23456789
a2345678910
a234567891011
Aby kontynuować, naciśnij dowolny klawisz . . .

```

Rys. 2.2.2. Wynik działania kodu zagadki (`++i`)

### 2.2.3. Konstrukcja Do While...Loop VB.NET

Ogólna struktura konstrukcji Do While Loop w języku VB.NET wygląda następująco:

```
Do While <wyrażenie logiczne>  
    <blok instrukcji>  
Loop
```

Blok instrukcji wewnątrz konstrukcji Do While wykonuje się tak długo jak długo wynikiem wyliczenia wyrażenia logicznego jest wartość true. Kod wypisujący na konsoli kolejne liczby od 1 do 10 z wykorzystaniem konstrukcji Do While wygląda następująco:

```
Sub Main()  
    Dim i As Integer = 1  
    Do While i <= 10  
        Console.WriteLine(i)  
        i = i + 1  
    Loop  
End Sub
```

### 2.2.4. Konstrukcja Do...Loop While VB.NET

Konstrukcja ta jest podobna do konstrukcji Do While...Loop z tą różnicą, że warunek nie jest sprawdzany przed pierwszą iteracją (pętla zawsze wykona się przynajmniej jeden raz).

```
Do  
    <blok instrukcji>  
Loop While <wyrażenie logiczne>
```

Przykład zastosowania konstrukcji Do...Loop While do wypisania kolejnych liczb od 1 do 10:

```
Sub Main()  
    Dim i As Integer = 1  
    Do  
        Console.WriteLine(i)  
        i = i + 1  
    Loop While i <= 10  
End Sub
```

### 2.2.5. Konstrukcja Do...Loop Until VB.NET

Konstrukcja Do...Loop Until jest podobna do konstrukcji Do...Loop While z tą różnicą, że instrukcje wewnątrz pętli są wykonywane dopóki warunek logiczny nie przyjmie wartości true. Schemat konstrukcji Do...Loop Until:

```
Do
    <blok instrukcji >
Loop Until <wyrażenie logiczne >
```

Przykład zastosowania konstrukcji Do...Loop Until do wypisania kolejnych liczb od 1 do 10:

```
Sub Main()
    Dim i As Integer = 1
    Do
        Console.WriteLine(i)
        i = i + 1
    Loop Until i > 10
End Sub
```

Tak jak w przypadku konstrukcji Do...Loop While pętla Do...Loop Until wykona się przynajmniej raz.

### 2.2.6. Konstrukcja while C#

Struktura konstrukcji while

```
while (<wyrażenie logiczne >)
{
    <blok instrukcji >;
}
```

Przykład zastosowania konstrukcji while wypisujący kolejne liczby od 1 do 10:

```
static void Main(string [] args)
{
    int i = 0;
    while (i <= 10)
    {
        Console.WriteLine(i);
        i = i + 1;
    }
}
```

Przyjrzyjmy się innemu zapisowi powyższego programu:

```
static void Main(string [] args)
{
    int i = 0;
    while (i++ < 10)
    {
        Console.WriteLine(i);
    }
}
```

Najpierw inicjalizujemy zmienną `i` wartością zero, pierwsze wykonanie pętli wylicza wyrażenie logiczne `i++<10`, ponieważ konstrukcja `i++` zwraca wartość zmiennej `i` a następnie dokonuje inkrementacji zmiennej `i` wykonywane jest porównanie `0<10` w wyniku otrzymujemy wartość `true` następnie zmienna `i` jest zwiększana o jeden i w tym momencie ma wartość 1. Instrukcja `Console.WriteLine(i)` wypisuje więc napis 1. Na koniec kiedy zmienna `i` ma wartość 9 wyrażenie logiczne `i<10` jest dalej spełnione czyli wykonywana jest najpierw inkrementacja zmiennej `i`, zmienna `i` posiada teraz wartość 10, zostaje wypisana na ekran i w kolejnym kroku pętli warunek `i<10` jest już fałszywy ponieważ zmienna `i` ma wartość 10.

### 2.2.7. Konstrukcja `do...while` C#

Struktura konstrukcji `do...while`

```
do
{
    <blok instrukcji >;
}while (<wyrażenie logiczne >);
```

Przykład zastosowania konstrukcji `do...while` wypisujący kolejne liczby od 1 do 10:

```
static void Main(string [] args)
{
    int i = 1;
    do
    {
        Console.WriteLine(i);
        i = i + 1;
    } while (i <= 10);
}
```

Oczywiście w tym przypadku możemy też skorzystać z operatora `++`.

---

## Rozdział 3

---

# Laboratorium 3

### 3.1. Tablice

Tablica jest strukturą danych zawierającą zmienne tego samego typu. CLR środowiska .NET Framework wspiera tworzenie tablic jedno oraz wiele wymiarowych.

#### 3.1.1. Tablice w VB.NET

Każda tablica VB.NET jest obiektem dziedziczącym z klasy `System.Array`. Tablice deklaruje się w następujący sposób:

```
Dim <identyfikator>(<rozmiar tablicy>) As <typ>
```

Deklaracja tablicy do przechowania 10 wartości typu `Integer`:

```
Dim tablicaLiczbaCalkowitych(9) As integer
```

Powyższy fragment kodu deklaruje tablicę mogącą przechować 10 wartości typu `Integer`, elementy tablicy indeksowane są od 0 do 9. W tym przypadku rozmiar tablicy jest stały i został podany w nawiasach przy deklaracji zmiennej typu tablicowego.

Możemy również zadeklarować zmienną tablicową bez podawania jej rozmiaru, rozmiar zostanie ustalony podczas inicjalizacji tablicy:

```
Dim tab() As Integer  
tab = New Integer() {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

Powyższy kod deklaruje najpierw zmienną `tab` jako tablicę liczb typu `Integer`, następnie w nawiasach klamrowych występują oddzielone przecinkami kolejne elementy tablicy.

Kiedy deklaracja i inicjalizacja tablicy są rozdzielone, inicjalizację trzeba wykonać podając w nawiasach klamrowych elementy tablicy.

Odwołanie się do elementu tablicy odbywa się poprzez operator indeksowania (`i`) gdzie `i` oznacza indeks elementu tablicy do którego chcemy się odwołać. Pamiętajmy, że indeks pierwszego elementu to 0

Przykładowy program korzystający z tablicy:

```
Sub Main()  
    Dim i As Integer = 0  
    Dim tab() As Integer  
    tab = New Integer() {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}  
    For i = 0 To 9  
        Console.WriteLine(tab(i))  
    Next  
End Sub
```

Powyższy program jest dość prosty, ale musieliśmy na sztywno umieścić rozmiar tablicy w pętli `For`. Wcześniej wspomnieliśmy, że wszystkie klasy `.NET Framework` dziedziczą z klasy `System.Array`, w której znajduje się wiele przydatnych metod i właściwości. jedną z bardzo przydatnych właściwości klasy `System.Array` jest `Length` gdzie przechowywana jest informacja o rozmiarze tablicy. Możemy napisać kod dla powyższego przykładu zastępując sztywny rozmiar tablicy użyty w pętli `For` odwołaniem do właściwości `Length`

```
Sub Main()  
    Dim i As Integer = 0  
    Dim tab() As Integer  
    tab = New Integer() {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}  
    For i = 0 To tab.Length - 1  
        Console.WriteLine(tab(i))  
    Next  
End Sub
```

Ponieważ właściwość `Length` zawiera rozmiar tablicy a elementy indeksowane są od 0, w pętli `for` pojawia się `Length - 1`

Możemy teraz zaprezentować inną formę funkcji `Main` od której rozpoczyna się wykonanie aplikacji konsolowej napisanej w języku `VB.NET`.

```
Sub Main(ByVal args As String())  
  
End Sub
```

Zauważmy, że procedura `Main` posiada parametr o nazwie `args` (nazwa zmiennej może być dowolna) typu tablica wartości typu `String`. Dodatkowo pojawiło się słowo kluczowe `ByVal`, które zostanie omówione później.

Zmienna `args` będzie zawierała parametry przekazane naszej aplikacji z linii poleceń. Możemy napisać program, który wyświetli wszystkie parametry z jakimi została wywołana nasza aplikacja:

```
Sub Main(ByVal args As String())
    Dim i As Integer
    For i = 0 To args.Length - 1
        Console.WriteLine(args(i))
    Next
End Sub
```

Powyższy przykład wykorzystuje pętlę typu `For`. Język `VB.NET` posiada jednak wygodniejszą konstrukcję do iteracji poprzez elementy tablicy, mianowicie konstrukcję `For Each`

```
For Each <identyfikator> In <tablica lub kolekcja>
    <blok instrukcji>
Next
```

Zamieńmy kod wyświetlający elementy tablicy tak, aby wykorzystywał konstrukcję `For Each`:

```
Sub Main()
    Dim tab As Integer()
    tab = New Integer() {1, 2, 3, 4, 5}
    Dim i As Integer
    For Each i In tab
        Console.WriteLine(i)
    Next
End Sub
```

Zmienna `i` w każdej iteracji zawiera wartość kolejnego elementu tablicy `tab`. Zmienna `i` musi być tego samego typu co elementy tablicy (w naszym przypadku `Integer`).

- Zmienna `i` może być wykorzystana tylko do odczytania wartości elementu tablicy, nie można przy jej pomocy zmienić zawartości tablicy. Jeżeli zachodzi konieczność modyfikacji elementów tablicy podczas iteracji trzeba skorzystać z konstrukcji `For...Next`,
- Konstrukcja `For Each` służy do iteracji po elementach tablicy jak i kolekcji. Pojęcie kolekcji zostanie omówione później,
- Klasa `String` jest także kolekcją znaków.

Możemy zastosować konstrukcję `For Each` do iteracji poprzez kolejne znaki wchodzące w skład łańcucha znaków. Poniższy program wypisuje poszczególne litery wchodzące w skład napisu:



```

Sub Main()
    Dim napis As String = "Jakiś napis"
    Dim c As Char
    For Each c In napis
        Console.WriteLine(c)
    Next
End Sub

```

Poniższy kod pokazuje zastosowanie tablicy tablic, do wyświetlenia kalendarza.

```

1  Sub Main()
2      Dim i As Integer
3      Dim Miesiace(), Dni() As String
4      Dim Kalendarz()() As Integer = New Integer(11)() {}
5      Dim Miesiac, Dzień As Integer
6      Miesiace = New String(11) {"Styczeń", "Luty", "Marzec", -
7                                "Kwiecień", "Maj", "Czerwiec", -
8                                "Lipiec", "Sierpień", "Wrzesień", -
9                                "Październik", "Listopad", "Grudzień"}
10     Dni = New String(6) {"Pn", "Wt", "Śr", "Cz", -
11                          "Pt", "So", "Nie"}
12     For Miesiac = 0 To Miesiace.Length - 1
13         Dzień = DateTime.DaysInMonth(Year(Now), Miesiac + 1)
14         Kalendarz(Miesiac) = New Integer(Dzień - 1) {}
15         For i = 0 To Dzień - 1
16             Kalendarz(Miesiac)(i) = i + 1
17         Next i
18     Next Miesiac
19     Dim d As DateTime
20     For Miesiac = 0 To Miesiace.Length - 1
21         Console.WriteLine(Miesiace(Miesiac))
22         d = New DateTime(Year(Now), Miesiac + 1, 1)
23         i = d.DayOfWeek ' 0 - nd, 1 - pn ...
24         If (i = 0) Then
25             i = 6
26         Else
27             i = i - 1
28         End If
29         For Dzień = 0 To Dni.Length - 1
30             Console.Write("{0} ", Dni(Dzień))
31         Next
32         Console.WriteLine()
33         For Dzień = 0 To i - 1
34             Console.Write(" ".PadRight(3))
35         Next
36         For Dzień = 0 To Kalendarz(Miesiac).Length - 1
37             Console.Write("{0:00} ", Kalendarz(Miesiac)(Dzień))
38             If (i + 1) Mod 7 = 0 Then
39                 Console.WriteLine()
40             End If
41             i = i + 1
42         Next Dzień
43         Console.WriteLine()
44         Console.WriteLine()
45     Next Miesiac
46 End Sub

```

### 3.1.2. Tablice w C#

Każda tablica C# jest obiektem dziedziczącym z klasy `System.Array`. Tablice deklaruje się w następujący sposób:

```
<typ>[<rozmiar tablicy >] <identyfikator >;
```

Deklaracja tablicy do przechowania 10 wartości typu `Integer`:

```
int [10] tablicaLiczbCalkowitych;
```

Powyższy fragment kodu deklaruje tablicę mogącą przechować 10 wartości typu `Integer`, elementy tablicy indeksowane są od 0 do 9. W tym przypadku rozmiar tablicy jest stały i został podany w nawiasach przy deklaracji zmiennej typu tablicowego.

Podawanie rozmiaru w deklaracji tablicy różni się w języku C# w stosunku do języka VB.NET. Deklarując tablicę w VB.NET podawaliśmy maksymalny indeks tablicy, natomiast w C# podajemy ilość elementów.

Tak jak w przypadku VB.NET możemy zadeklarować tablicę bez podawania jej rozmiaru, a następnie dokonać jej inicjalizacji:

```
int [] tab;
tab = new int [10] {1,2,3,4,5,6,7,8,9,10};
```

Możemy również utworzyć tablicę dynamicznie nie inicjalizując jej:

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

Operatorem indeksowania w przypadku C# jest operator `[]`. przykładowy program deklaruje zmienną `tab` jako tablicę liczb typu `int`, następnie tworzy nowy obiekt reprezentujący tablicę i wypełnia elementy tablicy kolejnymi liczbami by na końcu używając pętli `for` wyświetlić zawartość tablicy:

```
static void Main(string [] args)
{
    int [] tab;
    tab = new int [10];
    for (int i=0; i<tab.Length; i++)
        tab[i]=i+1;
    for (int i = 0; i < tab.Length; i++)
        Console.WriteLine(tab[i]);
}
```

Tak samo jak w przypadku VB.NET w języku C# istnieje konstrukcja `foreach` służąca do wykonania operacji iteracji na elementach tablicy lub kolekcji.

```
foreach (<typ> <identyfikator> in <tablica lub kolekcja >)
    <blok instrukcji >
```

Analogiczny kod jak w przypadku VB.NET używający konstrukcji `foreach`

```
static void Main(string [] args)
{
    string s = "Jakiś napis";

    foreach (char c in s)
        Console.WriteLine(c);
}
```

Zauważmy, że deklaracja funkcji `Main` języka C# od której rozpoczyna się wykonanie aplikacji konsolowej, zawiera argument o nazwie `args` typu `string []`.

Tak jak w przypadku VB.NET jest to tablica łańcuchów znaków zawierająca parametry przekazane w linii poleceń podczas uruchamiania programu.

Przykładowy kod wykorzystuje pętlę `foreach` do wyświetlenia parametrów przekazanych z linii poleceń:

```
static void Main(string [] args)
{
    if (args.Length < 1)
        Console.WriteLine("Brak parametró wywołania!");
    else
    {
        Console.WriteLine("Przekazana parametry:");
        foreach (string c in args)
        {
            Console.WriteLine(c);
        } //foreach
    } //else
} //Main
```

Tablice wielowymiarowe deklaruje się podobnie do tablic jednowymiarowych:

```
int [,] tablica2Wymiarowa;
tablica2Wymiarowa = new int [10,10];
```

Kod wykorzystujący tablice tablic:

```

1  static void Main(string[] args)
2  {
3      int Dzień, Miesiąc, i;
4      string[] Miesiące, Dni;
5      int[][] Kalendarz = new int[12][];
6      Miesiące = new string[12] {"Styczeń", "Luty", "Marzec", "Kwiecień",
7                                "Maj", "Czerwiec", "Lipiec", "Sierpień",
8                                "Wrzesień", "Październik", "Listopad",
9                                "Grudzień"};
10     Dni = new string[7] {"Pn", "Wt", "Śr", "Cz", "Pt", "So", "Nie"};
11     for ( Miesiąc = 0; Miesiąc < Miesiące.Length; Miesiąc++)
12     {
13         Dzień = DateTime.DaysInMonth(DateTime.Now.Year, Miesiąc + 1);
14         Kalendarz[Miesiąc] = new int[Dzień];
15         for (i = 0; i < Dzień; i++)
16             Kalendarz[Miesiąc][i] = i + 1;
17     } // for Miesiąc
18     for (Miesiąc = 0; Miesiąc < Miesiące.Length ; Miesiąc++)
19     {
20         Console.WriteLine(Miesiące[Miesiąc]);
21         DateTime d = new System.DateTime(DateTime.Now.Year, Miesiąc + 1, 1);
22         i = (int)d.DayOfWeek;
23         i = i == 0 ? 6 : i - 1;
24         // 0 - nd, 1 - pn ...
25         for (Dzień = 0; Dzień < Dni.Length; Dzień++)
26             Console.Write("{0} ", Dni[Dzień]);
27         Console.WriteLine();
28         Console.Write("".PadRight(3*i));
29         for (Dzień = 0; Dzień < Kalendarz[Miesiąc].Length; Dzień++)
30         {
31             Console.Write("{0:00} ", Kalendarz[Miesiąc][Dzień]);
32             if ((i++ + 1) % 7 == 0)
33                 Console.WriteLine();
34         }
35         Console.WriteLine();
36         Console.WriteLine();
37     } // for Miesiąc
38 } // Main

```

```

C:\WINDOWS\system33...
Styczeń
Pn Wt Śr Cz Pt So Nie
    01
02 03 04 05 06 07 08
09 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31

Luty
Pn Wt Śr Cz Pt So Nie
    01 02 03 04 05
06 07 08 09 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28

```

Rys. 3.1.1. Wynik działania programu

## 3.2. Funkcje matematyczne

Poza podstawowymi operacjami matematycznymi, platforma .NET Framework posiada bibliotekę `Math` pozwalającą na wykorzystanie bardziej złożonych funkcji matematycznych.

Poniższa tabelka zawiera zestaw funkcji matematycznych:

Tabela 3.2.1. Funkcje matematyczne

funkcja	użycie
<code>Math.Abs()</code>	zwraca wartość bezwzględną z liczby <code>Math.Abs(-10)</code> zwraca 10
<code>Math.Ceiling()</code>	sufit z liczby <code>Math.Ceiling(4.56)</code> zwraca 5
<code>Math.Floor()</code>	podłoga z liczby <code>Math.Floor(4.56)</code> zwraca 4
<code>Math.Max()</code>	zwraca większą z liczb <code>Math.Max(4,2)</code> zwraca 4
<code>Math.Min()</code>	zwraca mniejszą z liczb <code>Math.Min(4,2)</code> zwraca 2
<code>Math.Pow()</code>	zwraca liczbę podniesioną do potęgi <code>Math.Pow(2,3)</code> zwraca $2^3$ czyli 8
<code>Math.Round()</code>	zaokrągla liczbę do podanej liczby miejsc <code>Math.Round(1.23456,3)</code> zwraca 1.235
<code>Math.Sign()</code>	zwraca znak liczby <code>Math.Sign(-4)</code> zwraca -1
<code>Math.Sqrt()</code>	pierwiastek kwadratowy z liczby <code>Math.Sqrt(4)</code> zwraca 2

## 3.3. Obsługa wyjątków

Obsługa wyjątków jest mechanizmem wbudowanym w platformę .NET Framework, pozwala na wykrycie oraz obsłużenie wyjątków występujących w aplikacji podczas wykonania.

Wyjątki mogą być spowodowane błędami w kodzie programu, lub mogą być zgłaszane przez system operacyjny np nie znaleziono pliku, który program chce otworzyć.

Platforma .NET Framework zawiera domyślny mechanizm obsługi wyjątków, przerywając wykonywanie programu i zgłaszając wystąpienie wyjątku.

Brak obsługi wyjątków w aplikacji nie jest więc eleganckim sposobem programowania, użytkownik oczekuje od aplikacji tego, że nawet w sytuacji wystąpienia jakiegoś błędu program pozwoli na bezpieczne zakończenie jego pracy i np, zapisani wyników dotychczasowej pracy.

### 3.3.1. Wyjątki VB.NET

Język VB.NET dostarcza konstrukcję do obsługi wyjątków:

```
Try
    <blok_instrukcji>
Catch e As <typ>
    <blok_instrukcji>
Finally
    <blok_instukcji>
End Try
```

Jeżeli wystąpi wyjątek w bloku Try sterowanie przenoszone jest do bloku Catch na końcu niezależnie od tego czy wystąpił wyjątek czy nie, wykonywane są instrukcje zawarte w bloku Finally.

Bloki Catch oraz Finally są opcjonalne w języku VB.NET ale przynajmniej jeden z bloków musi wystąpić.

```
Sub Main()
    Dim liczba1 As Integer
    Dim liczba2 As Integer
    Dim wynik As Integer
    liczba1 = 10
    liczba2 = 0
    Try
        wynik = liczba1 / liczba2
        Console.WriteLine(wynik)
    Catch
        Console.WriteLine("błąd!")
    End Try
End Sub
```

Powyższy kod wykonuje w bloku Try instrukcję powodującą wygenerowanie wyjątku dzielenia przez zero. Jeżeli program nie zawierałby bloku Try jego wykonanie zostałoby przerwane z komunikatem zgłaszającym wystąpienie wyjątku. Kod zawiera jednak blok Try w momencie wystąpienia wyjątku sterowanie zostanie przeniesione do bloku Catch i zostanie wypisany napis błąd, a program będzie kontynuował działanie.

Powyższy kod nie zawiera jednak informacji na temat wyjątku jaki wystąpił, wiemy tylko że miało miejsce wystąpienie wyjątku i możemy poinformować użytkownika o wystąpieniu jakiegoś błędu, ale nie możemy go sprecyzować.

Przyjrzyjmy się zmodyfikowanej wersji kodu:

```

Sub Main()
  Dim liczba1 As Integer
  Dim liczba2 As Integer
  Dim wynik As Integer
  liczba1 = 10
  liczba2 = 0
  Try
    wynik = liczba1 / liczba2
    Console.WriteLine(wynik)
  Catch e As Exception
    Console.WriteLine(e.Message)
  End Try
End Sub

```

Pojawił się dodatkowy kod w bloku `Catch` mianowicie wygląda on teraz `Catch e As Exception`, można go potraktować jako zadeklarowanie zmiennej `e` typu `Exception`, w momencie kiedy wystąpi wyjątek zmienna `e` będzie zawierała dodatkowe informacje na temat wyjątku.

`Exception` jest obiektem reprezentującym dowolny wyjątek jaki może wystąpić. Procedurę obsługi wyjątków możemy skonstruować w bardziej dokładny sposób rozpatrując jakie typy wyjątków chcemy obsłużyć i w jaki sposób.

```

Sub Main()
  Dim liczba1 As Integer
  Dim liczba2 As Integer
  Dim wynik As Integer
  liczba1 = 10
  liczba2 = 0
  Try
    wynik = liczba1 / liczba2
    Console.WriteLine(wynik)
  Catch e As OverflowException
    Console.WriteLine(e.Message)
  Catch ee As Exception
    Console.WriteLine("wystąpił wyjątek")
  End Try
End Sub

```

Operacja dzielenia przez zero liczby całkowitej spowoduje wygenerowanie wyjątku `OverflowException`, wykona się więc instrukcja `Console.WriteLine(e.Message)`, gdyby jednak zastąpić instrukcję dzielenia jakimś innym kodem powodującym powstanie wyjątku wykonałby się blok `Catch ee As Exception`

```

Sub Main()
  Dim liczba1 As Integer
  Dim wynik As Integer
  liczba1 = 10
  Try
    wynik = "abc"
    Console.WriteLine(wynik)
  Catch e As OverflowException
    Console.WriteLine(e.Message)
  Catch ee As Exception
    Console.WriteLine("wystąpił wyjątek")
  End Try
End Sub

```

Powyższy kod nie wygeneruje wyjątku `OverflowException`, lecz wyjątek konwersji napisu `abc` na liczbę typu `Integer`, gdyby zabrakło bloku `Catch ee As Exception` program zostałby przerwany przez domyślną procedurę obsługi wyjątków, ponieważ wyjątek, który wystąpił jest typu `InvalidCastException`.

Podstawowe typy wyjątków:

Tabela 3.3.1. Wyjątki

wyjątek
<code>OutOfMemoryException</code>
<code>NullReferenceException</code>
<code>InvalidCastException</code>
<code>ArrayTypeMismatchException</code>
<code>IndexOutOfRangeException</code>
<code>ArithmeticException</code>
<code>DevideByZeroException</code>
<code>OverFlowException</code>

### Generowanie wyjątków

Wyjątki są dobrym mechanizmem zgłaszania błędów przez procedury. Zamiast pisać funkcję która zwraca np. kod reprezentujący czy wykonanie powiodło się czy nie, a następnie testowania tego kodu, możemy w procedurze lub funkcji w przypadku niepowodzenia wygenerować wyjątek, który w części kodu wywołującym daną funkcję lub procedurę wykorzysta blok `Try`.

```

Sub Main()
  Try
    ProceduraTestowa ()
  Catch e As Exception
    Console.WriteLine(e.Message)
  End Try
End Sub

Sub ProceduraTestowa ()
  Throw New DivideByZeroException("dzielenie przez zero!")
End Sub

```

Popatrzy ma trochę zmodyfikowany kod:



```
Sub Main()  
  Try  
    ProceduraTestowa()  
  Catch e As Exception  
    Console.WriteLine(e.Message)  
  End Try  
End Sub  
  
Sub ProceduraTestowa()  
  Dim liczba As Integer  
  Try  
    liczba = "aaa"  
  Catch ex As Exception  
    Console.WriteLine("Wyjątek!")  
    Throw  
  End Try  
End Sub
```

Procedura Main wywołuje procedurę ProceduraTestowa, która w wyniku swojego działania powoduje powstanie wyjątku, zawiera ona jednak blok Try gdzie wyjątek jest obsługiwany, natomiast po obsłużeniu wyjątku korzysta z instrukcji Throw do przekazania informacji o wyjątku wyżej, czyli do miejsca gdzie została wywołana, w naszym przypadku blok Try procedury Main. Możemy teraz w procedurze Main obsłużyć jeszcze raz ten wyjątek, który powstał w wyniku wykonania procedury ProceduraTestowa.

### 3.3.2. Wyjątki C#

Ponieważ język C# jest językiem platformy .NET, również dostępny jest w nim mechanizm obsługi wyjątków. Tak samo jak w przypadku VB.NET fragment kodu powodujący wystąpienie wyjątku, jeżeli nie zostanie ujęty w blok try spowoduje domyślną obsługę wyjątku, która doprowadzi do przerwania wykonywania aplikacji.

Ogólna struktura konstrukcji try dla C# wygląda następująco:

```
try  
{  
  <blok_instrukcji>  
}  
catch (<typ> <zmienna>)  
{  
  <blok_instrukcji>  
}  
finally  
{  
  <blok_instrukcji>  
}
```

Tak jak w przypadku VB.NET bloki catch oraz finally są opcjonalne, ale musi wystąpić przynajmniej jeden z nich.

Przykładowy program z obsługą wyjątków:

```
static void Main(string[] args)
{
    int liczba1 = 10, liczba2 = 0;
    int wynik;
    try
    {
        wynik = liczba1 / liczba2;
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
}
```

Tak jak w przypadku VB.NET możemy generować wyjątki programistycznie:

```
class Program
{
    static void Main(string[] args)
    {
        try
        {
            ProceduraTestowa();
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
        }
    }
    static void ProceduraTestowa()
    {
        throw new
            DivideByZeroException("dzielenie przez zero!");
    }
}
```

---

## Rozdział 4

---

# Laboratorium 4

### 4.1. Klasy

Klasa jest pewnym abstrakcyjnym modelem używanym do definiowania nowego typu danych. Klasa może składać się z danych, operacji, które można wykonać na danych (zwanymi metodami) oraz właściwościami (metodami dostępu do danych). Jako przykład możemy posłużyć się klasą `String` zdefiniowaną w przestrzeni `System` biblioteki klas środowiska `.NET Framework` (FCL).

Klasa `String` zawiera tablicę znaków (dane) i dostarcza różne operacje (metody), które mogą być wykonane na danych, np `ToLowerCase()`, `Trim()` itd. Klasa zawiera też właściwości (umożliwiające dostęp do danych) np `Length`.

Różnica pomiędzy metodą a właściwością polega na tym, że metoda jest funkcją lub procedurą wykonującą jakieś operacje na danych wchodzących w skład klasy, natomiast właściwość możemy potraktować tak jakby była to zwykła zmienna zdefiniowana wewnątrz klasy, do której możemy się odwołać (przypisać jej wartość, odczytać wartość). Nie wszystkim właściwością możemy przypisać wartość (właściwośći tylko do odczytu), zależne to jest od tego co autor klasy uznał za stosowne, jeżeli zdefiniował jakąś z właściwośći jako tylko do odczytu, możemy pobrać jej wartość natomiast nie możemy jej przypisać. Zostanie to omówione dalej na konkretnych przykładach.

Klasa jest pewną abstrakcją danych. Konkretna realizacja klasy nazywa się **obiektem**. Obiekt tworzony jest przy pomocy operatora `new`. Zmienna która przechowuje dane na temat stworzonego obiektu nazywa się **referencją**.

Zakładając, że mamy zdefiniowaną klasę o nazwie `KlasaTestowa`, stworzenie obiektu w języku `VB.NET` wygląda następująco:

```
Dim jakisObiekt As New KlasaTestowa()
```

natomiast w C#

```
KlasaTestowa jakisObiekt = new KlasaTestowa();
```

Różnica pomiędzy klasami a bezpośrednimi typami danych jest taka, że obiekt jest typem referencyjnym (przekazywanym do metod jako referencja) podczas gdy zmienne typu podstawowego przekazywane są jako kopia (lepiej będzie to widać na przykładach).

Klasa może zawierać dane przechowywane w zmiennych nazywanych polami. Pola mogą być zmiennymi typu podstawowego, lub referencjami do innych obiektów.

Metody z punktu widzenia programisty są to po prostu funkcje lub procedure zdefiniowane wewnątrz klasy, wykonujące jakies operacje na danych klasy. Metoda tak jak procedura, czy funkcja może posiadać parametry.

Podczas tworzenia obiektu za pomocą operatora **new** wywoływany jest konstruktor klasy (więcej na ten temat dalej na konkretnych przykładach).

#### 4.1.1. Klasy w VB.NET

Klasę w języku VB.NET deklarujemy przy pomocy słowa kluczowego **Class** i deklarację kończymy znacznikiem **End Class**

```
Class KlasaTestowa
    ' pola, właściwości, metody
End Class
```

#### Metody

W języku VB.NET mamy dwa typy metod: procedury oraz funkcje. Różnica pomiędzy procedurą a funkcją jest taka, że procedurę deklarujemy używając słowa kluczowego **Sub**, natomiast funkcję słowa kluczowego **Function**. Procedura nie zwraca żadnej wartości jako wynik swojego działania, natomiast funkcja zwraca wartość.

Poniższy fragment kodu przedstawia procedurę w języku VB.NET

```
Sub ProceduraTestowa()
    Console.WriteLine("procedura")
End Sub
```

Powyższa procedura nie pobiera parametrów, jedyne co robi to wypisuje na konsoli napis procedura. Oczywiście procedura może pobierać parametry:

```
Sub ProceduraTestowa(ByVal tekst As String)
    Console.WriteLine("procedura: {0}", tekst)
End Sub
```

Powyższa procedura pobiera jeden parametr o nazwie napis typu `String`, a następnie wyświetla na konsoli napis będący złożeniem napisu „procedura” oraz przekazanego parametru.

Słowo kluczowe `ByVal` oznacza, że parametr będzie przekazany do procedury przez zmienną (zostanie utworzona jego kopia).

Przyjrzyjmy się fragmentowi kodu:

```
Sub Main()
    Dim i As Integer
        i = 2
        ProceduraTestowa(i)
        Console.WriteLine(i)
End Sub

Sub ProceduraTestowa(ByVal i As Integer)
    i = i + 1
End Sub
```

W procedurze `Main` mamy deklarację zmiennej `i` typu `Integer`, jest to lokalna zmienna procedury `Main`. Procedura `ProceduraTestowa` pobiera parametr o nazwie `i` typu `Integer` poprzez wartość (`ByVal`), zmienna `i` jest lokalna wewnątrz procedury `ProceduraTestowa`. Efektem działania procedury `ProceduraTestowa` jest zwiększenie o jeden wartości zmiennej `i` (lokalnej dla procedury), która została przekazana jako kopia. Po zakończeniu swojego działania zmienna ta przestaje istnieć. Procedura `Main` wypisze więc na konsoli wartość 2, ponieważ taką wartość przypisaliśmy do zmiennej lokalnej `i` procedury `Main`.

Dokonajmy drobnej modyfikacji w kodzie:

```
Sub Main()
    Dim i As Integer
        i = 2
        ProceduraTestowa(i)
        Console.WriteLine(i)
End Sub

Sub ProceduraTestowa(ByRef i As Integer)
    i = i + 1
End Sub
```

Tym razem procedura `ProceduraTestowa` pobiera parametr nie przez wartość, ale przez referencję (zmienną), nie jest tworzona kopia tylko przekazywana jest referencja. Można powiedzieć że zmienna `i` wewnątrz procedury `Proceduratestowa` „pokazuje” na miejsce w pamięci, gdzie znajduje się zmienna `i` procedury `Main`. Wykonując więc operację `i = i + 1`, tak naprawdę dokonuje modyfikacji zmiennej `i` zadeklarowanej w procedurze `Main`. Tym razem efektem działania programu będzie wyświetlenie wartości 3.

## Funkcje

Funkcja nie różni się zasadniczo od procedury, parametry przekazywane są w identyczny sposób. Jedyną różnicą polega na tym, że procedura zwraca pewną wartość (typ zwracanej wartości określany jest w deklaracji funkcji) przy pomocy słowa kluczowego `Return`

```
Sub Main()  
    Dim wynik As Integer  
    wynik = Suma(2, 4)  
    Console.WriteLine(wynik)  
End Sub  
  
Function Suma(ByVal liczba1 As Integer, -  
              ByVal liczba2 As Integer) As Integer  
    Return liczba1 + liczba2  
End Function
```

Powyższy kod zawiera definicję funkcji `Suma`, która pobiera dwa parametry typu liczba całkowita i jako wynik działania zwraca ich sumę.

Postaramy się teraz zbudować klasę `Student`, która będzie zawierała informacje o studencie, imię, nazwisko, oraz oceny z trzech przedmiotów, będzie udostępniała metodę zwracającą średnią ocen.

```
Class Student  
    Dim oceny(2) As Integer  
    Function Srednia() As Double  
        Dim suma As Integer = 0  
        Dim i As Integer  
        For Each i In oceny  
            suma = suma + i  
        Next  
        Return suma / oceny.Length()  
    End Function  
  
    Sub WystawOceny()  
        oceny(0) = 3  
        oceny(1) = 5  
        oceny(2) = 4  
    End Sub  
End Class  
  
Sub Main()  
    Dim s As New Student()  
    s.WystawOceny()  
    Console.WriteLine(s.Srednia())  
End Sub
```

Klasa ta jest dość mało elastyczna, przechowuje informacje na temat ocen z trzech przedmiotów. Jedyne co można zrobić to przy pomocy metody `Srednia()` otrzymać wynik średniej ocen. Zauważmy, że pole klasy `oceny` nie jest dostępne (nie jest możliwa jego modyfikacja). Tablica `oceny` jest ukryta dla osoby korzystającej z klasy `Student`, jest to dość ważna cecha programowania obiektowego, pisząc własną klasę mamy pewność, że osoba korzystająca z niej nie będzie miała dostępu do składowych pól klasy.

## Konstruktory

Powyższy przykład nie zawiera jeszcze informacji na temat imienia oraz nazwiska studenta. Zmodyfikujemy go dodając brakujące pola oraz dodając konstruktory.

Konstruktor jest z punktu widzenia programisty procedurą. Procedura ta ma nazwę `New`.

```
Class Student
    Dim oceny(2) As Integer
    Dim nazwisko As String

    Public Sub New()
        WystawOceny()
        nazwisko = "brak"
    End Sub
    Function PobierzNazwisko() As String
        Return nazwisko
    End Function

    Function Srednia() As Double
        Dim suma As Integer = 0
        Dim i As Integer
        For Each i In oceny
            suma = suma + i
        Next
        Return suma / oceny.Length()
    End Function
    Sub WystawOceny()
        oceny(0) = 3
        oceny(1) = 5
        oceny(2) = 4
    End Sub
End Class

Sub Main()
    Dim s As New Student()
    Console.WriteLine("{0} : średnia ocen {1}", -
        s.PobierzNazwisko(), s.Srednia())
End Sub
```

## Konstruktory

Konstruktor jest wołany w momencie tworzenia obiektu przy pomocy operatora `New`. Konstruktor w języku VB.NET nosi nazwę `New`, możemy zdefiniować dowolnie wiele różnych konstruktorów różniących się listą argumentów, jeżeli klasa nie zawiera konstruktora, tworzony jest dla niej konstruktor domyślny.

```

Class Student
  Dim oceny(3) As Integer
  Dim nazwisko As String
  Public Sub New()
    nazwisko = "brak"
    WystawOceny()
  End Sub
  Public Sub New(ByVal nazwisko As String)
    Me.nazwisko = nazwisko
    WystawOceny()
  End Sub
  Public Function Srednia() As Double
    Dim suma As Integer = 0
    For Each i As Integer In oceny
      suma += i
    Next
    Return suma / oceny.Length()
  End Function
  Sub WystawOceny()
    oceny(0) = 3
    oceny(1) = 5
    oceny(2) = 4
  End Sub
  Public Function PobierzNazwisko() As String
    Return Me.nazwisko
  End Function
End Class

Sub Main()
  Dim s As New Student()
  Console.WriteLine("{0} : {1}", -
    s.PobierzNazwisko(), s.Srednia())
  s = New Student("Iksiński")
  Console.WriteLine("{0} : {1}", -
    s.PobierzNazwisko(), s.Srednia())
End Sub

```

Klasa `Student` posiada dwa konstruktory o nazwie `New`, różniące się listą parametrów. Konstruktor pobierający parametr o nazwie `nazwisko`, odwołuje się do składowej klasy o nazwie `nazwisko` korzystając z operatora `Me`, aby wskazać, że chodzi w tym miejscu o składową klasy `nazwisko`, a nie parametr `nazwisko`.

#### 4.1.2. Klasy w C#

Klasę w języku `C#` deklarujemy przy pomocy słowa kluczowego `class`, a ciało klasy zawieramy w nawiasach klamrowych:

```

class KlasaTestowa
{
}

```

### Metody

Podobnie jak w języku `VB.NET` w języku `C#` mamy dwa typy metod: procedury, oraz funkcje. Różnica polega w sposobie deklaracji. Procedura języka `C#` może zostać potraktowana jako specjalna funkcja, która nie zwraca żadnej wartości, w kodzie zaznaczamy to deklarując ją jako zwracającą typ `void`.



```
static void ProceduraTestowa()  
{  
    Console.WriteLine("procedura");  
}
```

Powyższa procedura nie pobiera parametrów, a w wyniku jej działania jest wypisanie w oknie konsoli komunikatu „procedura”.

Możliwe jest deklarowanie procedur zawierających parametry:

```
static void ProceduraTestowa(string tekst)  
{  
    Console.WriteLine("procedura: {0}", tekst);  
}
```

Komentarza wymaga użycie słowa kluczowego **static**, oznacza ono zadeklarowanie statycznej metody, w skrócie można napisać, że do statycznych składowych klasy można odwoływać się bez tworzenia obiektu. Metoda `ProceduraTestowa` została zadeklarowana wewnątrz klasy `Program`, klasa ta reprezentuje naszą aplikację, i istnieje dokładnie jedna instancja naszego programu. Nie jest tworzony obiekt na podstawie klasy `Program`, wszystkie metody muszą być więc zadeklarowane w tej klasie jako statyczne.

```
class Program  
{  
    static void Main(string[] args)  
    {  
        ProceduraTestowa("Test");  
    }  
    static void ProceduraTestowa(string tekst)  
    {  
        Console.WriteLine("procedura: {0}", tekst);  
    }  
} // class
```

Domyślnie parametry przekazywane są przez wartość (tworzona jest kopia parametru przekazywanego do procedury lub funkcji), istnieje również możliwość tak jak w języku `VB.NET` przekazania parametru przez referencje, musimy w tym celu skorzystać ze słowa kluczowego **ref**. Język `C#` jest trochę bardziej wymagający w stosunku do języka `VB.NET`, nie wystarczy tylko zadeklarować procedurę podając w jej deklaracji, że parametr ma być przekazywany jako referencja, musimy również przy wywołaniu procedury jawnie napisać, że chodzi nam o przekazanie parametru przez referencje:

```
static void Main(string[] args)
{
    int i = 2;
    ProceduraTestowa(ref i);
    Console.WriteLine(i);
}
static void ProceduraTestowa(ref int parametr)
{
    parametr = parametr + 1;
}
```

Dodatkowo możemy zadeklarować w metodzie przekazanie parametru przez referencje używając słowa kluczowego `out`:

```
static void Main(string[] args)
{
    int i=2;
    bool b;
    ProceduraTestowa(ref i, out b);
    Console.WriteLine(i);
}
static void ProceduraTestowa(ref int parametr, out bool b)
{
    parametr = parametr + 1;
    b = true;
}
```

Różnica pomiędzy przekazaniem parametru z wykorzystaniem słowa kluczowego `ref` a `out` jest taka, że w przypadku przekazywania jako `ref` zmiennej musi zostać nadana wartość przed przekazaniem jej jako parametr (w kodzie powyżej `int i = 2`, natomiast przekazując parametr jako `out` nie musimy nadawać mu wartości przed przekazaniem do procedury lub funkcji, natomiast procedura lub funkcja musi przypisać temu parametrowi wartość.

Jeżeli mamy przekazać do procedury lub funkcji kilka parametrów tego samego typu, możemy to zrobić przy pomocy tablicy parametrów z wykorzystaniem słowa kluczowego `params`:

```
static void Main(string[] args)
{
    Console.WriteLine(Suma(1, 2));
    Console.WriteLine(Suma(1, 2,3));
}
static int Suma(params double[] tab)
{
    int suma = 0;
    foreach (int i in tab)
    {
        suma += i;
    }
    return suma;
}
```

## Funkcje

Funkcje w języku C# podlegają tym samym zasadom co procedury z tą różnicą, że zwracają wartość przy pomocy słowa kluczowego `return`:

```
static void Main(string [] args)
{
    int i=2, j=3;
    Console.WriteLine(Suma(i, j));
}
static int Suma(int a, int b)
{
    return a + b;
}
```

## Klasa Student

Tak jak w przypadku VB.NET zadeklarujemy podobną klasę o nazwie `Student`:

```
class Student
{
    int [] oceny = new int [3];
    public double Srednia ()
    {
        int suma = 0;
        foreach (int i in oceny)
        {
            suma += i;
        }
        return suma / oceny.Length;
    }
    public void WystawOceny ()
    {
        oceny [0] = 3;
        oceny [1] = 5;
        oceny [2] = 4;
    }
}
class Program
{
    static void Main(string [] args)
    {
        Student s = new Student ();
        s.WystawOceny ();
        Console.WriteLine(s.Srednia ());
    }
}
```

Zwróćmy uwagę na pojawienie się słowa kluczowego `public` w deklaracji metod `Srednia`, oraz `WystawOceny`, są to tak zwane modyfikatory klasy. Zasadniczo w języku C# dostępne jest siedem modyfikatorów, w tym momencie omówimy jedynie cztery podstawowe kontrolujące poziom dostępu:

Tabela 4.1.1. Modyfikatory klasowe

modyfikator	znaczenie
public	metoda lub składowa klasy jest w pełni dostępna z poziomu innych typów
internal	
private	,metoda lub składowa klasy jest dostępne tylko wewnątrz klasy
protected	metoda lub składowa jest dostępna w klasie oraz klasach pochodnych

## Konstruktory

Klasy języka **C#** zawierają konstruktory, które są wywoływane w momencie tworzenia obiektu, w odróżnieniu od języka **VB.NET** konstruktor w języku **C#** posiada taką samą nazwę jak klasa. Przyjrzyjmy się następującemu programowi:

```

class Student
{
    int [] oceny = new int [3];
    string nazwisko;
    public Student ()
    {
        nazwisko = "brak";
        WystawOceny ();
    }
    public Student (string nazwisko)
    {
        this.nazwisko = nazwisko;
        WystawOceny ();
    }
    public double Srednia ()
    {
        int suma = 0;
        foreach (int i in oceny)
        {
            suma += i;
        }
        return suma / oceny.Length;
    }
    public string PobierzNazwisko ()
    {
        return nazwisko;
    }
    void WystawOceny ()
    {
        oceny [0] = 3;
        oceny [1] = 5;
        oceny [2] = 4;
    }
}
class Program
{
    static void Main (string [] args)
    {
        Student s = new Student ("Iksiński");
        Console.WriteLine (" {0} : {1} ", s.PobierzNazwisko (),
            s.Srednia ());
    }
}

```

Klasa `Student` posiada dwa konstruktory o nazwie `Student`, różniące się listą parametrów (zostanie to opisane dokładnie w dalszej części). Pierwsza wersja konstruktora bez parametrów, nadaje składowej `nazwisko` wartość „brak”, a następnie wywołuje metodę `WystawOceny`, która tym razem nie jest zadeklarowana jako publiczna i nie jest dostępna z poziomu funkcji `Main` klasy `Program`, gdzie tworzymy obiekt klasy `Student`. Druga wersja konstruktora, zawiera jeden parametr o nazwie `nazwisko`. Zwróćmy uwagę, że składowa klasy posiada taką samą nazwę jak nazwa parametru konstruktora, aby pokazać, że po lewej stronie operacji przypisania ma znaleźć się `nazwisko` zadeklarowane jako składowa klasy, używamy słowa kluczowego `this`, które oznacza instancję tego obiektu. Napis `this.nazwisko` oznacza, że chodzi nam o składową o nazwie `nazwisko` z bieżącej klasy.

### 4.1.3. Przeciążanie metod

Języki obiektowe dostarczają mechanizmy umożliwiające przeciążanie metod, ogólnie idea przeciążania polega na możliwości napisania kilku wersji tej samej metody (o takiej samej nazwie), różniące się typem parametrów, ale zwracany typ musi być taki sam dla wszystkich wersji. Dobrym przykładem jest metoda `WriteLine` klasy `Console`, która posiada 19 różnych wariantów.

#### Przykład VB.NET

```
Sub Main ()
    Console.WriteLine(IleZnakow("abc"))
    Console.WriteLine(IleZnakow(1234))
End Sub

Function IleZnakow(ByVal s As String) As Integer
    Return s.Length()
End Function
Function IleZnakow(ByVal i As Integer) As Integer
    Dim s As String
    s = String.Format("{0}", i)
    Return IleZnakow(s)
End Function
```

#### Przykład C#

```
static void Main(string[] args)
{
    Console.WriteLine(IleZnakow("abc"));
    Console.WriteLine(IleZnakow(1234));
}
static int IleZnakow(string s)
{
    return s.Length;
}
static int IleZnakow(int i)
{
    string s = string.Format("{0}", i);
    return IleZnakow(s);
}
```

Zdefiniowaliśmy dwie metody o nazwie `IleZnakow`, różniące się typem pobieranego parametru, pierwsza z nich pobiera wartość typu `string`, druga typu `int`. Obie metody zwracają ilość znaków, pierwsza w napisie, druga ilość cyfr w liczbie. Zauważmy, że druga wersja metody `IleZnakow` korzysta z wersji metody, która jako parametr pobiera napis.

#### 4.1.4. Właściwości (properties)

Właściwości mogą być traktowane jak „wirtualne” składowe klasy. Dla użytkownika klasy właściwości wyglądają jak zwykłe składowe klasy, ale wewnątrz klasy możemy na nie spojrzeć jak na specyficzne metody.

#### Właściwości w VB.NET

Deklarując właściwość w języku VB.NET korzystamy ze słowa kluczowego `Property` po którym występuje nazwa właściwości oraz jej typ, kolejne są bloki `Get` oraz `Set` są wykorzystywane odpowiednio jako metody do pobrania wartości oraz ustawienia wartości.

```
Class Osoba
  Dim _wiek As Integer
  Public Property Wiek() As Integer
  Get
    Return Me._wiek
  End Get
  Set(ByVal value As Integer)
    If value < 0 Then
      _wiek = 0
    ElseIf value > 100 Then
      _wiek = 100
    Else
      _wiek = value
    End If
  End Set
End Property
End Class
Sub Main()
  Dim o As New Osoba()
  o.Wiek = 24
  Console.WriteLine(o.Wiek)
End Sub
```

Dodatkowo możemy zadeklarować właściwość tylko do odczytu lub taką której możemy tylko przypisać wartość natomiast nie możemy jej odczytać, różnica w deklaracji polega na tym, że jeśli chcemy zadeklarować właściwość tylko do odczytu potrzebujemy wyłącznie bloku `Get` oraz przy definicji właściwości musi pojawić się słowo kluczowe `ReadOnly`.

```
Class Osoba
  Dim _wiek As Integer
  Public ReadOnly Property Wiek() As Integer
  Get
    Return Me._wiek
  End Get
  End Property
End Class
```

Kiedy chcemy zadeklarować właściwość, z możliwością ustawienia jej wartości, ale bez możliwości odczytania jej, postępujemy podobnie tym razem jednak użyjemy słowa kluczowego `WriteOnly` oraz zadeklarujemy wyłącznie blok `Set`

```
Public WriteOnly Property Wiek() As Integer
Set(ByVal value As Integer)

End Set
End Property
```

### Właściwości w C#

Właściwość zdefiniowana w klasie języka **C#** składa się z deklaracji typu oraz nazwy (tak jak w przypadku zwykłego pola klasy), różnica polega na tym, że z każdą związane są odpowiednie bloki `set` oraz `get` gdzie deklarujemy metody odpowiedzialne za nadanie oraz pobranie wartości. Rozważmy przykładową klasę `Osoba`, która zawiera składową typu `int` o nazwie `wiek`. Jeżeli składowa ta była zadeklarowana jako pole klasy i udostępniona publicznie, użytkownik klasy mógłby nadać jej dowolną wartość (zakładamy, że poprawny wiek ma być z przedziału `[0, 100]`). Oczywiście można by nie udostępniać publicznie składowej `wiek` i napisać specjalną metodę np o nazwie `UstawWiek`, która zmieniała by wartość składowej `wiek` dokonując odpowiedniego sprawdzenia, czy wartość jaką chcemy przypisać jest z odpowiedniego zakresu. Bardziej elegancką jednak formą jest jednak ta oferowana przez mechanizm właściwości.

```
class Osoba
{
    int wiek;
    public int Wiek
    {
        get
        {
            return this.wiek;
        }
        set
        {
            if (value < 0)
                this.wiek = 0;
            else if (value > 100)
                this.wiek = 100;
            else
                this.wiek = value;
        }
    }
} // class Osoba
class Program
{
    static void Main(string[] args)
    {
        Osoba o = new Osoba();
        o.Wiek = 25;
        Console.WriteLine(o.Wiek);
    }
}
```

Zauważmy, że w kodzie programu, tworząc obiekt typu `Osoba`, do właściwości `Wiek` odwołujemy się tak jakby była to składowa klasy, może ona wystąpić zarówno po lewej stronie operacji przypisania, wykonywana jest wtedy część kodu z bloku `set`, lub po prawej stronie gdzie zwracana jest wartość (wykonywany jest blok `get`).