Platforma .NET

„Skrót wykładu”

Dr inż. Krzysztof Smółka

ksmolka@p.lodz.pl

Niniejsze materiały są jedynie skrótem treści przedstawionych na wykładzie i nie obejmują wszystkich wiadomości i umiejętności wymaganych w ramach zaliczenia przedmiotu.

W materiałach znajdują się również przykłady z komentarzami, które ze względu na ograniczenia czasowe nie zostały przedstawione na wykładzie.

Politechnika Łódzka

1. **CEL**
- Student powinien poznać podstawową terminologią, przeznaczenie i logiką platformy .NET. oraz umieć tworzyć aplikacje z wykorzystaniem języków programowania platformy .NET

2. **WYMAGANIA WSTĘPNE**
- Podstawy informatyki, Podstawy programowania, Programowanie obiektowe, Bazy danych

Efekty

1. Student poprawnie identyfikuje technologie, metody, języki programowania i narzędzia związane z platformą .NET i potrafi je omówić.
2. Student potrafi - zgodnie z zadaną specyfikacją - zaprojektować oraz zrealizować prostą, typową aplikację i stronę www z wykorzystaniem platformy .NET, używając właściwych metod, technik i narzędzi.
3. Student potrafi w swoich aplikacjach zastosować elementy technologii ADO.NET.

3. **WYKŁAD**
- Podstawowe terminy, przeznaczenie i logika platformy .NET,
- Funkcje platformy .NET Framework (klient, serwer oraz usługi),
- Podstawowe komponenty .NET Framework (maszyna wirtualna CLR, biblioteka klas podstawowych BCL)
- Programowanie zorientowane obiektowo w języku VB.NET,
- Podstawy języka C# (literały, operatory, klasa Object, cechy OOP, wyjątki, kolekcje, interfejsy generyczne, delegaty, strumienie, asynchronizm)
- Tworzenie aplikacji WWW z wykorzystaniem ASP.NET
- Programowanie dostępu do baz danych z wykorzystaniem ADO.NET
- ĆWICZENIA LABORATORYJNE
- Tworzenie aplikacji z wykorzystaniem języków programowania platformy .NET
- Tworzenie aplikacji WWW z wykorzystaniem ASP.NET i ADO.NET

Literatura podstawowa:

1. Lars Powers, Mike Snell, Microsoft Visual Studio 2008. Księga eksperta, Helion 2009
2. Wei-Meng Lee, C# 2008. Warsztat programisty, Helion 2010
3. Rod Stephens, Visual Basic 2008. Warsztat programisty, Helion 2009
4. Bill Evjen, Scott Hanselman, Devin Rader, ASP.NET 4 z wykorzystaniem C# i VB. Zaawansowane programowanie, Helion 2011
5. Dokumentacja środowiska Visual Studio .NET w aktualnej wersji i materiały MSDN (Microsoft Developer Network) - kolekcja stron dla społeczności programistów i deweloperów, skupionych wokół technologii firmy Microsoft.

Literatura uzupełniająca:

1. Kevin Hoffman, Microsoft Visual C# 2005. Księga eksperta, Helion 2007
2. Andrew Troelsen, Język C# 2008 i platforma .NET 3.5, PWN 2011
3. Heinrich Gantenbein, Greg Dunn, Amit Kalani, Chris Payne, Thiru Thangarathinam, Microsoft Visual Basic .NET 2003. Księga eksperta , Helion 2006

Platforma .NET

WPROWADZENIE

.NET Framework

- **.NET Framework**, w skrócie .NET (wym. dot net) – platforma programistyczna opracowana przez Microsoft, obejmująca środowisko uruchomieniowe (Common Language Runtime – CLR) oraz biblioteki klas dostarczające standardowej funkcjonalności dla aplikacji.

.NET Framework

- Technologia ta nie jest związana z żadnym konkretnym językiem programowania, a programy mogą być pisane w jednym z wielu języków – na przykład C++/CLI, C#, J#, Delphi 8 dla .NET, Visual Basic .NET.

.NET Framework

Zadaniem platformy .NET Framework jest zarządzanie różnymi elementami systemu:

- kodem aplikacji,
- pamięcią
- i zabezpieczeniami.

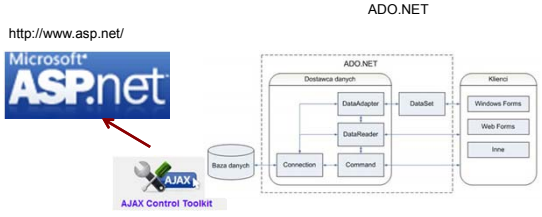
- W środowisku tym można tworzyć oprogramowanie działające po stronie serwera internetowego (IIS) oraz pracujące na systemach, na które istnieje działająca implementacja tej platformy.

Bloki składowe platformy .NET (CLI **Common Language Infrastructure**)

- **CLR** (*ang. Common Language Runtime*) odpowiedzialny za lokalizowanie, wczytywanie oraz zarządzanie typami .NET. To trzon całej platformy .NET ponieważ to właśnie do CLR należy zadanie kompilowania i uruchamiania kodu zapisanego językiem kodu pośredniego (CIL).
- **CTS** (*ang. Common Type System*) jest odpowiedzialny za opis wszystkich danych udostępnianych przez środowisko uruchomieniowe.
- **CLS** (*ang. Common Language Specification*) to zbiór zasad definiujących podzbiór wspólnych typów precyzujących zgodność kodu binarnego z dostępnymi kompilatorami .NET

Technologie

- Platforma .NET niesie ze sobą kilka pochodnych technologii. Można tu wymienić ADO.NET, ułatwiający dostęp do baz danych, oraz ASP.NET, służąca do budowania dynamicznych stron WWW i wiele innych.



Technologie

- Windows Presentation Foundation (WPF, nazwa kodowa Avalon) – nazwa silnika graficznego i API bazującego na .NET 3, wchodzącego w skład WinFX.
- WPF integruje interfejs użytkownika, grafikę 2D i 3D, multimedia, dokumenty (nazwa kodowa Metro) oraz generowanie/rozpoznawanie mowy (do aplikacji sterowanych głosem).

- API w WPF opiera się na języku XML, dokładniej na jego implementacji o nazwie XAML. Całość jest zawarta w nowym API WinFX, zaś graficzna część GUI wykorzystuje grafikę wektorową, budowaną z użyciem akceleratorów grafiki 3D i efektów graficznych.
- Rozwiązanie to jest podobne do Quartz z Mac OS X.

Technologi

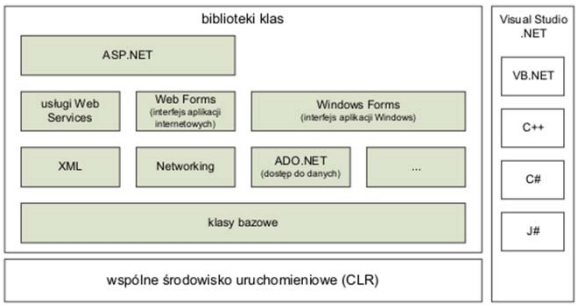
- Windows Communication Foundation, w skrócie WCF, to nowy podsystem komunikacyjny przeznaczony do łatwej implementacji komunikacji między aplikacjami na jednej lub wielu maszynach.

- WCF jest częścią .NET Framework 3.0, który był częścią Windows Vista.

Komponenty, CLR

Główne składniki platformy .NET to

- wspólne środowisko uruchomieniowe
- oraz nowe, hierarchicznie zorganizowane biblioteki klas, które ułatwiają obsługę graficznych interfejsów użytkownika, dostęp do baz danych i plików oraz komunikację sieciową.



Dla nowych wersji .NET, schemat można rozbudować…

.NET 4 I 1/2

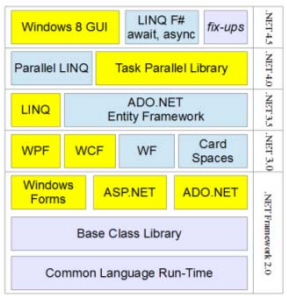
Zmiany

- Ogólnie nowe zmiany można podzielić na kilka obszarów platformy .NET.

- Niewątpliwie pierwszym elementem są zmiany związane z Windows 8, czyli aplikacje Metro (ang. *.NET for Metro style Apps*).

- Zmiany wprowadzono także do zbioru klas bazowych.

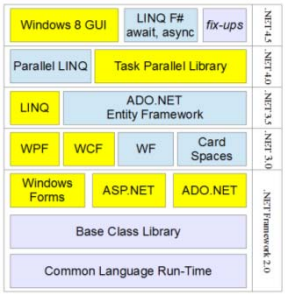
- Wprowadzono pomniejsze zmiany w .NET, ale unowocześnieniu uległy również rozwiązania programowania równoległego, a także pakiety związane z obsługą sieci oraz aplikacji WEB.



Zmiany

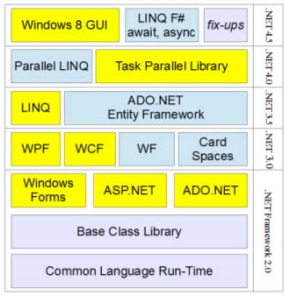
Microsoft uaktualnił też inne pakiety wchodzące w skład .NET, a są to:

- pakiety do tworzenia GUI Windows Presentation Foundation (WPF),
- technologia komunikacji Windows Communication Foundation (WCF)
- oraz system zarządzania procesami, czyli Workflow Foundation (WF).



Zmiany

- Jeśli chodzi o języki programowania związane z .NET, to nie ma tak dużych zmian w dwóch głównych językach platformy, czyli Visual Basic oraz C#.
- Wprowadzono dwa nowe słowa kluczowe, aby usprawnić programowanie asynchroniczne.
- Większą zmianą jest wprowadzenie obsługi zapytań LINQ do języka F#, co z pewnością zwiększy popularność tego języka.

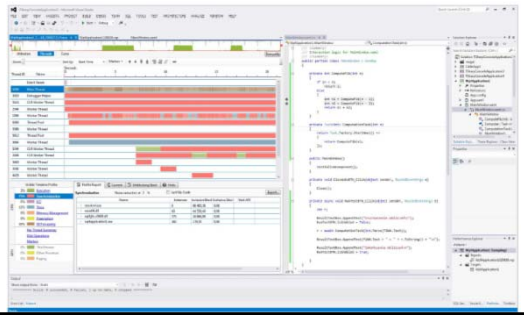


Ogólnie o nowościach w 4.5 .NET

- Dwa dodatkowe słowa kluczowe **async** oraz **await** upraszczają problem obsługi synchronicznych oraz asynchronicznych metod, która stosuje się w przetwarzaniu równoległym.
- *Wbrew pozorom nie chodzi w tym momencie o skomplikowane i wyrafinowane algorytmy, ale o bardziej prozaiczne zadania jak np. wczytywanie większego pliku XML czy też odczyt danych z sieci Internet.*
- Użycie dwóch nowych słów kluczowych upraszcza tworzenie programów asynchronicznych w stosunku do poprzednich wydań .NET, gdzie problem ten był rozwiązywany na poziomie biblioteki TPL – Task Parallel Library.
- *W dokumentacji do Visual Studio 2012 znajduje się przykład wykorzystania **await** i **async** do ściągania danych ze strony WWW w tle, bez blokowania zdarzeń w oknie głównym.*

Ogólnie o nowościach w 4.5 .NET

Wizualizacja współbieżności w programie w Visual Studio 2012 Ultimate



Platforma .NET

„Języki programowania”

Politechnika Łódzka

Języki programowania

- Główną cechą, która wyróżnia platformę .NET wśród innych środowisk programistycznych, jest wsparcie **wielu języków programowania**.
- Umożliwia to grupom programistów o różnych umiejętnościach tworzenie komponentów oprogramowania w wybranych językach, a następnie połączenie opracowanych komponentów w dobrze współpracującą całość.
- Do języków platformy .NET zalicza się języki: Visual Basic .NET, Visual C++ oraz Visual C#.

C#

Visual C# .NET

- C# to jedyny język programowania Microsoft, który został zaprojektowany od samego początku specjalnie dla platformy .NET i wspólnego środowiska uruchomieniowego CLR.
- Chociaż CLR obsługuje wiele języków, to tylko C# był projektowany równolegle z CLR. Te dwie technologie miały na siebie duży wpływ, przez co C# świetnie nadaje się do pisania kodu zarządzanego.
- Kod zarządzany komponentów platformy .NET, takich jak biblioteki klas i środowisko programistyczne ASP.NET, został napisany właśnie w języku C#.

Visual C# .NET

- C# jest znacznie prostszym językiem niż C++, jednak - jak sama nazwa wskazuje - należy do rodziny języków C.
- Oznacza to, że ma wiele cech wspólnych z C/C++, których nie mają języki takie jak Visual Basic.
- Na przykład C# rozróżnia wielkie i małe litery, a VB nie.
- C# wymaga od programistów jawnej konwersji pomiędzy typami danych, a Visual Basic dokonuje niektórych konwersji automatycznie.

Visual C# .NET

- Składnia języka C# jest podobna do składni języków C++ i Java.
- C# ma jednak w stosunku do C++ kilka dodatkowych cech obiektowych, takich jak właściwości, atrybuty, delegaty czy zdarzenia.
- W języku C# można także tworzyć kod „niebezpieczny”. W C# można na przykład uzyskać bezpośredni dostęp do pamięci zaalokowanej dla bufora i przeglądać tę pamięć przy użyciu wskaźników.

Visual Basic .NET

Visual Basic .NET

- Visual Basic jest uważany za najbardziej popularny język programowania aplikacji dla Windows. (!!! ???)
- W wersji Visual Basic .NET wprowadzono do języka wiele zmian.
- Zmiany objęły między innymi sposób deklaracji zmiennych i funkcji, sposób tworzenia i usuwania obiektów, domyślny sposób przekazywania parametrów funkcji, sposób wywoływania procedur.

Visual Basic .NET

- Największe zmiany dotyczą chyba sposobu obsługi błędów - została usunięta stosowana do tej pory obsługa błędów, często nazywana
- „*on error goto hell*”.

- Visual Basic .NET w pełni wspiera strukturalną obsługę wyjątków.

Visual Basic .NET

- Dzięki możliwości współdziałania platformy .NET Framework ze starszymi technologiami, kod napisany w Visual Basic .NET może wywoływać istniejący kod, napisany w starszych wersjach języka Visual Basic (i odwrotnie), dzięki czemu aplikacje na platformie .NET mogą wykorzystywać starsze moduły aplikacji.

Visual Basic .NET

- Na platformę .NET przeniesiono jedynie główną odmianę języka Visual Basic.

- *Nie został przeniesiony język skryptowy **Visual Basic Scripting Edition** (znany pod nazwą **VBScript**), wykorzystywany do tworzenia skryptów administracyjnych, stron ASP i dynamicznej zawartości stron internetowych*
- *oraz język **Visual Basic for Applications (VBA)** - język skryptowy wykorzystywany do pisania makr dla aplikacji rodziny Office (i nie tylko).*

Wprowadzenie do programowania

Wprowadzenie

- Platforma .NET została opracowana dla systemów operacyjnych firmy MS Windows.
- Zatem u podstaw działania jest system operacyjny.

- Druga warstwa to wspólne środowisko uruchomieniowe
- – CLR (ang. Common Language Runtime).

- Środowisko to odpowiada w 90% maszynie wirtualnej Javy.

Wprowadzenie

- Środowisko CLR w szczególności odpowiada za:

- **zarządzanie pamięcią** – to właśnie tu znajduje się GarbageCollector.

- Platforma .NET podobnie jak inne języki wprowadziła zarządzalny model pamięci.
- Oznacza to, że programista nie ma bezpośredniego dostępu do pamięci operacyjnej komputera – pomiędzy kodem programu a systemem operacyjnym jest warstwa pośrednia, która optymalizuje proces dostępu do pamięci, dba o minimalizowanie fragmentacji sterty, ale przede wszystkim chroni system przed wyciekami pamięci spowodowanymi nieprawidłową pracą aplikacji,

Wprowadzenie

- Środowisko CLR w szczególności odpowiada za:
- **kontrola typów** – otóż posługiwanie się polimorficznymi kolekcjami i klasami bardzo łatwo powoduje błędne przekazanie jako argumentu wywołań obiektu klasy, której typ w danym kontekście jest nieodpowiedni,

Wprowadzenie

- Środowisko CLR w szczególności odpowiada za:
- **tutaj także ma miejsce obsługa wyjątków** – czyli faktyczne przekazywanie informacji do programu działającego programu, że dany wyjątek wystąpił.
- Kod programu nie ma bezpośredniego dostępu do zasobów sprzętowych – to platforma .NET wyposażona w tysiące klas zaimplementowanych ukrywa w swoim wnętrzu niezarządzane, niskopoziomowe szczegóły współpracy z systemem operacyjnym.

Wprowadzenie

- **Kolejną warstwę stanowią tzw. klasy bazowe.**
- Jeszcze przed powstaniem platformy .NET biblioteki MS liczyły setki klas – otrzymały one wspólną nazwę MFC (MicroSoft Foundation Classes).
- Obecnie platforma .NET liczy grubo ponad 500 tysięcy klas.
- Część z nich w sposób przejrzysty i niezmienny dostarcza rozwiązań w zakresie najważniejszych i popularnych operacji – te klasy możemy określić mianem bazowych.

Wprowadzenie

- **Kolejną warstwę stanowią tzw. klasy bazowe.**
- Do grona takich właśnie klas zaliczają się:
- **kolekcje** – kolekcje typów ogólnych i generycznych.
  - Pozwalają na przetwarzanie zbiorów danych, przeszukiwanie ich, dodawanie i usuwanie elementów.
  - Znajdują się tutaj popularne struktury danych: Lista, Lista wiązana, Tablice asocjacyjne, Stos, Kolejka itd.,

Wprowadzenie

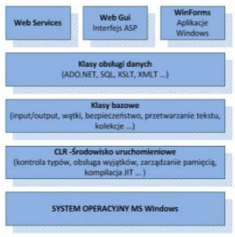
- **Kolejną warstwę stanowią tzw. klasy bazowe.**
- Do grona takich właśnie klas zaliczają się:
- **klasy bazowe** to także zagadnienia dotyczące przetwarzania tekstu.
- Składanie łańcuchów tekstowych, wyszukiwanie wzorców, dynamiczna realokacja tekstu, formatowanie ciągów, konwersje to operacje tak powszechne, że stanowią fundament pracy programów,

Wprowadzenie

- **Kolejną warstwę stanowią tzw. klasy bazowe.**
- Do grona takich właśnie klas zaliczają się:
- **obsługa wejścia/wyjścia** – czyli zagadnienia związane z wyświetlaniem danych na monitorze ekranowym, przechwytywanie zdarzeń z urządzeń wskazujących, odczyt danych z dysków, sieci w oparciu o rozmaite protokoły, implementacje protokołów siecowych, zapis danych w dowolne miejsce.

Wprowadzenie

- **Kolejny wyższy wyróżniony poziom to wyspecjalizowane klasy dostępu do danych oraz mechanizmy ich przetwarzania.**



- Odnajdziemy tutaj zatem ADO.NET – zestaw klas i interfejsów do efektywnego przetwarzania danych z relacyjnych baz danych. A także szeroko rozumianą obsługę danych w postaci XML.

PODSTAWOWE ELEMENTY JĘZYKA (-ÓW)

Pierwszy program

C#
```
using System;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Witamy w świecie programowania!");
            Console.ReadKey();
        }
    }
}
```


Pierwszy program

VB
```
Imports System

Module Module1
    Sub Main()
        Console.WriteLine("Witamy w świecie programowania!")
        Console.ReadKey()
    End Sub
End Module
```


Komentarze

C#
```
using System;
class Przywitanie // krótki komentarz
{
    static void Main()
    { /* dłuższy komentarz, zwierający wiele
         linii tekstu, lepiej wstawić jako
         komentarz blokowy */
        Console.WriteLine("Witamy w świecie programowania!");
    }
}
```


Komentarze

VB
```
Imports System

Module Module1
    Sub Main()
            ' w VB komentarz na kilka linii można uzyskać
            ' wyłącznie poprzez powtórzenie znaku
            ' na początku ' każdej linii komentarza
        Console.WriteLine("Witamy w świecie programowania!")
        Console.ReadKey()
    End Sub
End Module
```

Zmienne i typy danych

C#
```
using System;
class Przywitanie // krótki komentarz
{
   static void Main()
   {
            int calkowita = 76;
           double rzeczywista;
           rzeczywista = calkowita;
           Console.WriteLine("{0}, {1}", calkowita, rzeczywista);
   }
}
```


Zmienne i typy danych

VB
```
Imports System

Module Module1
   Sub Main()
      Dim calkowita As Integer = 76
      Dim rzeczywista As Double
      rzeczywista = calkowita
      Console.WriteLine("{0}, {1}", calkowita, rzeczywista)
   End Sub
End Module
```


Typy skalarne i referencyjne

C#
```
using System;
class Class1
{
   static void Main()
   {
      // definicja zmiennej całkowitoliczbowej o wartości 3
      int skalar1 = 3;
      int skalar2 = skalar1; // druga zmienna o tej samej wartości
      skalar2 = 76;
      // poniżej znajduje się deklaracja tablicy liczb całkowitych
      // zawierającej jeden element o wartości 3

      int []ref1 = {3};
      int []ref2 = ref1; // skopiowanie zmiennej, ale nie tablicy!
      ref2[0] = 76;

      Console.WriteLine("Typy skalarne: {0}, {1}", skalar1, skalar2);
      Console.WriteLine("Referencje: {0}, {1}", ref1[0], ref2[0]);
   }
}
```


Typy skalarne i referencyjne

VB
```
Imports System
Module Module1
   Sub Main()
      ' definicja zmiennej całkowitoliczbowej o wartości 3
      Dim skalar1 As Integer = 3
      Dim skalar2 As Integer = skalar1 ' druga zmienna o tej samej wartości
      skalar2 = 76

      ' poniżej znajduje się deklaracja tablicy liczb całkowitych
      ' zawierająca jeden element o wartości 3
      Dim ref1() As Integer = {3}
      Dim ref2() As Integer = ref1 ' skopiowanie zmiennej, ale nie tablicy!
      ref2(0) = 76

      Console.WriteLine("Typy skalarne: {0}, {1}", skalar1, skalar2)
      Console.WriteLine("Referencje: {0}, {1}", ref1(0), ref2(0))
   End Sub
End Module
```

W wyniku działania powyższego programu, w konsoli zostanie wyświetlony następujący tekst:

```
Typy skalarne: 3, 76
Referencje: 76, 76
```


Tablice

C#
```
using System;
class Class1
{
   static void Main()
   {
      char[] tab = new char[3]; // tablica znaków o długości 3
      //przypisanie wartości do poszczególnych elementów tablicy
      tab[0] = 'a';
      tab[1] = 'b';
      tab[2] = 'c';
      Console.WriteLine("Długość tablicy znaków: {0}", tab.Length);
   }
}
```


Tablice

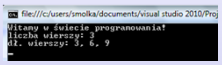
VB
```
Imports System
Module Module1
   Sub Main()
      Dim tab(2) As Char
      'przypisanie wartości do poszczególnych elementów tablicy
      tab(0) = "a"
      tab(1) = "b"
      tab(2) = "c"
      Console.WriteLine("Długość tablicy znaków: {0}", tab.Length)
   End Sub
End Module
```

Tablice

C#
```
using System;
class Class1
{
    static void Main()
    {
      // tablica jednowymiarowa o długości 3
      int[] tab1 = new int[] { 1, 2, 3 };
      // prostokątna tablica dwuwymiarowa (2x3)
      int[,] tab2 = new int[,] { { 1, 2, 3 }, { 4, 5, 6 } };
      // prostokątna tablica trójwymiarowa (10x2x100), bez podanych wartości
      int[, ,] tab3 = new int[10, 2, 100];
      // tworzenie tablicy postrzępionej o 3 wierszach
      int[][] j = new int[3][];
      j[0] = new int[] { 1, 2, 3 }; // elementy 1. wiersza
      j[1] = new int[] { 1, 2, 3, 4, 5, 6 }; // elementy 2. wiersza
      j[2] = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 }; // elementy 3. wiersza
      Console.WriteLine("liczba wierszy: {0}", j.Length);
      Console.WriteLine("dł. wierszy: {0}, {1}, {2}", j[0].Length, j[1].Length,
j[2].Length);
    }
}
```


Tablice

VB
```
Imports System
Module Module1
    Sub Main()
        ' tablica jednowymiarowa o długości 3
        Dim tab1() As Integer = {1, 2, 3}
        ' prostokątna tablica dwuwymiarowa (2x3)
        Dim tab2(,) As Integer = {{1, 2, 3}, {4, 5, 6}}
        ' prostokątna tablica trójwymiarowa (10x2x100), bez podanych wartości
        Dim tab3(9, 1, 99) As Integer
        ' tworzenie tablicy postrzępionej o 3 wierszach
        Dim j(2)() As Integer
        j(0) = New Integer() {1, 2, 3} ' elementy 1. wiersza
        j(1) = New Integer() {1, 2, 3, 4, 5, 6} ' elementy 2. wiersza
        j(2) = New Integer() {1, 2, 3, 4, 5, 6, 7, 8, 9} ' elementy 3. wiersza
        Console.WriteLine("liczba wierszy: {0}", j.Length)
        Console.WriteLine("dł. wierszy: {0}, {1}, {2}", j(0).Length, j(1).Length,
j(2).Length)
    End Sub
End Module
```



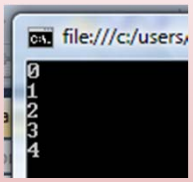
Polecenie while

C#
```
while ( wyrażenie ) instrukcje
```

VB
```
While wyrażenie
        instrukcje
End While
```


Polecenie while

C#
```
using System;
namespace ConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 0;
            int zm = 5;
            while (i < zm)
            {
                Console.WriteLine(i);
                i++; // zwiększenie zmiennej i o 1
            }
            Console.ReadKey();
        }
    }
}
```



Tablice

VB
```
Module Module1

    Sub Main()
        Dim i As Integer = 0
        Dim zm As Integer = 5
        While i < zm
            Console.WriteLine(i)
            i += 1 ' zwiększenie zmiennej i o 1
        End While
        Console.ReadKey()
    End Sub

End Module
```

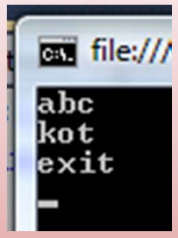
Polecenie do...while

C#
```
do
      instrukcje
while ( wyrażenie )
```

VB
```
Do
    instrukcje
Loop While wyrażenie
```

Polecenie do...while

C#
```
using System;
namespace ConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            string s;
            do
            {
                s = Console.ReadLine();
            }
            while (s != "exit");
        }
    }
}
```



Polecenie do...while

VB
```
Module Module1
    Sub Main()
        Dim s As String
        Do
            s = Console.ReadLine()
        Loop Until (s = "exit")
        Console.ReadKey()
    End Sub
End Module
```

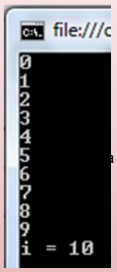

Polecenie for

- Pętla **for** jest rozbudowaną wersją pętli while.
- Ma ona następującą postać:

C#
```
for (instrukcje_inicjujące ; warunek ; instrukcje_krokowe) instrukcje
```


Polecenie for

C#
```
using System;
namespace ConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            {
                int i;
// Jeśli chcemy pominąć powyższą linię, to poniższą                    zapisać
 // for (int i = 0; i < 10; i++)
                for (i = 0; i < 10; i++)
                {
                    Console.WriteLine(i);
                }
                Console.WriteLine("i = {0:G}\n", i);
                Console.ReadKey();
            }
        }
    }
}
```



Polecenie for

C#
```
using System;
namespace ConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            {
                int i;
// Jeśli chcemy pominąć powyższą linię, to poniższą pętlę                    sać
 // for (int i = 0; i++ < 10; i++)
                for (i = 0; i++ < 10; i++)
                {
                    Console.WriteLine(i);
                }
                Console.WriteLine("i = {0:G}\n", i);
                Console.ReadKey();
            }
        }
    }
}
```

Polecenie for

VB
```
For licznik [ As typ_danych ] = start To koniec [Step krok ]
  instrukcje
Next [ licznik ]
```

Polecenie for

VB
```
Module Module1
    Sub Main()
        Dim zm As Integer = 5
        Dim i As Integer
        ' Jeśli chcemy pominąć powyższą linię,
        ' to poniższą pętlę For można zapisać
        ' For i As Integer = 0 To (zm - 1)

        For i = 0 To (zm - 1) Step 1
            Console.WriteLine(i)
        Next

        Console.ReadKey()
    End Sub
End Module
```

Funkcje

- **Szkielet funkcji:**

C#
```
typ_danych nazwa_funkcji (typy i nazwy argumentów funkcji)
{
        ciało funkcji
        return zwracana_wartość;
}
```

VB
```
Function nazwa_funkcji(typy i nazwy argumentów funkcji) As typ_danych
        ciało funkcji
        Return zwracana_wartość
End Function
```

C#
```
(int a, int b)
```

VB
```
(ByVal a As Integer, ByVal b As Integer)
```


Funkcje

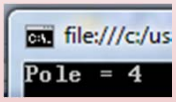
- **Szkielet funkcji:**

C#
```
typ_danych nazwa_funkcji (typy i nazwy argumentów funkcji)
{
        ciało funkcji
        return zwracana_wartość;
}
```

VB
```
Function nazwa_funkcji(typy i nazwy argumentów funkcji) As typ_danych
        ciało funkcji
        Return zwracana_wartość
End Function
```

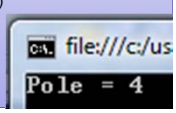

Funkcje

C#
```
using System;
namespace ConsoleApplication
{
    class Program
    {
        // definicja funkcji
        static int Pole(int a, int b)
        {
            return a * b;
        }
        static void Main()
        {
// wywołanie funkcji i przypisanie zwracanej przez nią wartości do zmiennej typu int
            int PoleProstokata;
            PoleProstokata = Pole(2, 2);
            Console.WriteLine("Pole = {0:G}", PoleProstokata);
            Console.ReadKey();
        }
    }
}
```

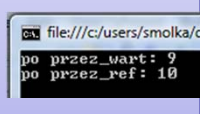


Funkcje

VB
```
Module Module1
    ' definicja funkcji
    Function Pole(ByVal a As Integer, ByVal b As Integer) As Integer
        Return (a * b)
    End Function

    Sub Main()
        ' wywołanie funkcji i przypisanie zwracanej przez nią
        ' wartości do zmiennej typu Integer
        Dim PoleProstokata As Integer
        PoleProstokata = Pole(2, 2)
        Console.WriteLine("Pole = {0:G}", PoleProstokata)
        Console.ReadKey()
    End Sub

End Module
```



VB
```
Module Module1

    Sub przez_wart(ByVal a As Integer)
        'zwiększamy "a" o jeden, ale nie będzie widać tego na zewnątrz
        a += 1
    End Sub

    Sub przez_ref(ByRef a As Integer)
        'zwiększamy "a" o jeden i będzie to widać na zewnątrz
        a += 1
    End Sub

    Sub Main()
        Dim p As Integer = 9

        przez_wart(p)
        Console.WriteLine("po przez_wart: {0}", p)
        przez_ref(p)
        Console.WriteLine("po przez_ref: {0}", p)
        Console.ReadKey()
    End Sub

End Module
```



Definiowanie klas

- Do definiowania klas w językach C# i Visual Basic służy słowo kluczowe **class**, po którym podajemy nazwę definiowanej klasy.
- Potem następuje blok zwany ciałem klasy, czyli definicja zawartości klasy.

C#
```
class nazwa_klasy
{
        // ciało klasy
}
```

VB
```
Class nazwa_klasy
        ' ciało klasy
End Class
```


Definiowanie klas

- Aby utworzyć *egzemplarz* (zwany też ***instancją***) naszej klasy, piszemy zwykłą konstrukcję deklaracji zmiennej, w której jako typ danych podajemy nazwę naszej klasy. Każdy egzemplarz klasy należy zainicjować, używając słowa kluczowego **new**.

C#
```
nasz_typ egzemplarz = new nasz_typ();
```

VB
```
Dim egzamplarz As nasz_typ = new nasz_typ
```

Definicja zmiennej powoduje przydzielenie pewnej ilości pamięci operacyjnej komputera dla nowej zmiennej. Operator **new** tworzy w tym obszarze pamięci nowy egzemplarz obiektu.
To tak, jakby na podstawie planów konstrukcyjnych (klasy) wykonano nowy egzemplarz obiektu (inicjowaną zmienną).

Zmienne i funkcje składowe

C#
```
class ModelSamochodu
{
    // funkcje składowe – na razie puste
    public void Jazda(double Droga)
    {
    }
    public void Skrec(double IleStopni)
    {
    }

    // zmienne składowe
    private double Predkosc;
    private double PrzejechanaDroga;
    private double SkrecenieKol;
    public string Nazwa;
    public string Kolor;
}
```


Zmienne i funkcje składowe

VB
```
Class ModelSamochodu
    ' funkcje składowe – na razie puste
    Public Sub Jazda(ByVal Droga As Double)
    End Sub
    Public Sub Skrec(ByVal IleStopni As Double)
    End Sub

    ' zmienne składowe
    Private Predkosc As Double
    Private PrzejechanaDroga As Double
    Private SkrecenieKol As Double
    Public Nazwa As String
    Public Kolor As String
End Class
```


Zmienne i funkcje składowe

C#
```
public void Jazda(double Droga)
{
    PrzejechanaDroga+=Droga;
}
public void Skrec(double IleStopni)
{
    SkrecenieKol+=IleStopni;
    if (SkrecenieKol>45)
        SkrecenieKol=45;
    else
        if (SkrecenieKol<-45)
            SkrecenieKol=-45;
}
```


Zmienne i funkcje składowe

VB
```
Public Sub Jazda(ByVal Droga As Double)
    PrzejechanaDroga += Droga
End Sub
Public Sub Skrec(ByVal IleStopni As Double)
    SkrecenieKol += IleStopni
    If (SkrecenieKol > 45) Then
        SkrecenieKol = 45
    ElseIf SkrecenieKol < -45 Then
        SkrecenieKol = -45
    End If
End Sub
```

79

C#

```
using System;
class ModelSamochodu
{
    // funkcje składowe
    public void Jazda(double Droga)
    {
        PrzejechanaDroga+=Droga;
    }
    public void Skrec(double IleStopni)
    {
        SkrecenieKol+=IleStopni;
        if (SkrecenieKol>45)
            SkrecenieKol=45;
        else
            if (SkrecenieKol<-45)
            SkrecenieKol=-45;
    }

    // zmienne składowe
    private double Predkosc;
    private double PrzejechanaDroga;
    private double SkrecenieKol;
    public string Nazwa;
    public string Kolor;
}
```

```
class Class1
{
    static void Main()
    {
        ModelSamochodu model1 = new ModelSamochodu();
        ModelSamochodu model2 = new ModelSamochodu();
        model1.Kolor = "czerwony";
        model2.Kolor = "czarny";
        model1.Jazda(1.32);
    }
}
```

80

VB

```
Imports System
Class ModelSamochodu
    ' funkcje składowe
    Public Sub Jazda(ByVal Droga As Double)
        PrzejechanaDroga += Droga
    End Sub
    Public Sub Skrec(ByVal IleStopni As Double)
        SkrecenieKol += IleStopni
        If (SkrecenieKol > 45) Then
            SkrecenieKol = 45
        ElseIf SkrecenieKol < -45 Then
            SkrecenieKol = -45
        End If
    End Sub
    ' zmienne składowe
    Private Predkosc As Double
    Private PrzejechanaDroga As Double
    Private SkrecenieKol As Double
    Public Nazwa As String
    Public Kolor As String
End Class
```

```
Module Module1
    Sub Main()
        Dim model1 As ModelSamochodu = New ModelSamochodu
        Dim model2 As ModelSamochodu = New ModelSamochodu
        model1.Kolor = "czerwony"
        model2.Kolor = "czarny"
        model1.Jazda(1.32)
    End Sub
End Module
```


Zmienne i funkcje składowe

C#

```
public void Jazda(double Droga)
{
    this.PrzejechanaDroga+=Droga;
}
```

VB

```
Public Sub Jazda(ByVal Droga As Double)
    Me.PrzejechanaDroga += Droga
End Sub
```


Konstruktory

- Metoda konstruktora w języku C# ma nazwę taką samą jak nazwa klasy.
- Natomiast w języku Visual Basic metoda konstruktora ma nazwę New.
- Istnieje możliwość przeciążania konstruktorów.

C#

```
public ModelSamochodu(string KolorFabryczny)
{
    Kolor = KolorFabryczny;
}
```

VB

```
Public Sub New(ByVal KolorFabryczny As String)
    Kolor = KolorFabryczny
End Sub
```


Konstruktory

C#

```
public ModelSamochodu(string KolorFabryczny)
{
    Kolor = KolorFabryczny;
}
……..
ModelSamochodu model1 = new ModelSamochodu("czerowny");
ModelSamochodu model2 = new ModelSamochodu("czarny");
```

VB

```
Public Sub New(ByVal KolorFabryczny As String)
    Kolor = KolorFabryczny
End Sub

Dim model1 As ModelSamochodu = new ModelSamochodu("czerwony")
Dim model2 As ModelSamochodu = new ModelSamochodu("czarny ")
```


Przeciążanie

C#

```
public ModelSamochodu(string KolorFabryczny)
{
    Kolor = KolorFabryczny;
}
public ModelSamochodu()
{
    Kolor = "rdzawy";
}
```

VB

```
Public Sub New(ByVal KolorFabryczny As String)
    Kolor = KolorFabryczny
End Sub

Public Sub New()
    Kolor = "rdzawy"
End Sub
```

Dziedziczenie

- Nowa klasa **dziedziczy** całą zawartość klasy podstawowej - stąd nazwa tej techniki programowania.

C#
```
class ModelTerenowy : ModelSamochodu
```

VB
```
Class ModelTerenowy Inherits ModelSamochodu
```


Dziedziczenie

C#
```
public ModelTerenowy() : base()
{
    BlokadaMechanizmu = false;
}
public ModelTerenowy(string kolor) : base(kolor)
{
    BlokadaMechanizmu = false;
}
```


Dziedziczenie

VB
```
Public Sub New()
    MyBase.New()
    BlokadaMechanizmu = False
End Sub
Public Sub New(ByVal KolorFabryczny As String)
    MyBase.New(KolorFabryczny)
    BlokadaMechanizmu = False
End Sub
```


Zasłanianie i zastępowanie

C#
```
class ModelTerenowy : ModelSamochodu
{
    ...

    public new void Skrec(double IleStopni)
    {
        SkrecenieKol += IleStopni;
        if (SkrecenieKol > 60)
            SkrecenieKol = 60;
        else
            if (SkrecenieKol < -60)
                SkrecenieKol = -60;
    }
}
```


Zasłanianie i zastępowanie

VB
```
Class ModelTerenowy
    Inherits ModelSamochodu

    ...

    Public Shadows Sub Skrec(ByVal IleStopni As Double)
        SkrecenieKol += IleStopni
        If (SkrecenieKol > 60) Then
            SkrecenieKol = 60
        ElseIf SkrecenieKol < -60 Then
            SkrecenieKol = -60
        End If
    End Sub
End Class
```


Modyfikatory

- Modyfikatory w C# i VB .NET Deklarując metody, właściwości i funkcje w C#/VB.NET, można stosować następujące modyfikatory:

Overloads (VB.NET)
Wykorzystywane jest podczas deklarowania wielu funkcji o tej samej nazwie, ale innych parametrach. Proszę spojrzeć na poniższy przykład:

-
```
Overloads Sub
Show(ByVal theChar As Char)
End Sub
```

-
```
Overloads Sub
Show (ByVal theInteger As Integer)
End Sub

Overloads Sub
Show (ByVal theDouble As Double)
End Sub
```

W C# nie trzeba w analogicznym wypadku stosować dodatkowego słowa kluczowego.

Modyfikatory

- **Overrides (VB.NET) / override (C#)**
  Określa, że dany element przesłania identyczny element w klasie bazowej.

  Overridable (VB.NET) / virtual (C#)
  Określa, że element może być przesłonięty w klasie dziedziczącej.

  NotOverridable (VB.NET) / sealed (C#)
  Określa, że element nie może być przesłonięty w klasie dziedziczącej.

  MustOverride (VB.NET) / abstract (C#)
  Określa, że element musi być przesłonięty w klasie dziedziczącej. Jeżeli cała klasa zdefiniowana jest jako abstract, to nie można utworzyć jej instancji.

  Przykładem klasy abstrakcyjnej jest TextReader, przeznaczona do odczytu sekwencji znaków. Nie można utworzyć jej instancji, ale ma ona dwie specjalizacje - StringReader, która czyta sekwencję ze zmiennej łańcuchowej i StreamReader, czytającą ze strumienia ze zdefiniowanym sposobem kodowania znaków - których można już używać w kodzie.

Modyfikatory

- **Shadows (VB.NET) / new (C#)**
  Określa, że element przesłania inny element z klasy bazowej, odmiennego typu.
  Praktycznie rzadko stosowane:

```
Public Class CBase
Public Z As Integer
End Class

Public Class CDerived
Inherits CBase

Public Shadows Z As String
End Class
```

W klasie CDerived Z jest typu String, mimo że w klasie bazowej jest inaczej.

Przestrzenie nazw

- Platforma Microsoft .NET Framework zawiera bogaty zestaw gotowych klas - tak zwaną bibliotekę klas.
- Klasy te zapewniają dostęp do podstawowych funkcji systemu operacyjnego i stanowią komponenty do budowy aplikacji, własnych kontrolek (elementów graficznego interfejsu użytkownika) i innych, bardziej rozbudowanych komponentów.
- *Ponieważ biblioteka ta jest bardzo rozbudowana, poszczególne jej części zostały umieszczone w różnych **przestrzeniach nazw**.*
- Dzięki hierarchicznemu uporządkowaniu wszystkich obiektów systemu operacyjnego, klas i funkcji, odnalezienie potrzebnych komponentów jest bardzo łatwe.

Przestrzenie nazw

- *Przestrzenie nazw mają także inną zaletę - zawężają zakres obowiązywania nazw typów.*
- Ponieważ podawanie pełnej nazwy klasy wraz z przestrzeniami nazw byłoby męczące, istnieje możliwość podania kompilatorowi, które przestrzenie nazw chcemy uznawać za domyślne.
- W języku C# służy do tego słowo kluczowe **using**, natomiast w języku Visual Basic słowo **Imports**, umieszczane na początku programu.
-

Przestrzenie nazw

C#

```
using System;
class Class1
{
    static void Main()
    {
        // gdyby nie using na początku, musielibyśmy
        // napisać System.Console.WriteLine
        Console.WriteLine("jakis tekst");
    }
}
```

VB

```
Imports System
Module Module1
    Sub Main()
        ' gdyby nie Imports na początku, musielibyśmy
        ' napisać System.Console.WriteLine
        Console.WriteLine("jakis tekst")
    End Sub
End Module
```


Właściwości

C#

```
class WskaznikPostepu
{
    private int postep;
    public int Postep
    {
        get
        {
            return postep;
        }
        set
        {
            postep = value;
            if (postep < 0) postep = 0;
            if (postep > 100) postep = 100;
        }
    }
}
```

```
WskaznikPostepu wsk = new WskaznikPostepu();
wsk.Postep = -10;
System.Console.Write("Postęp: {0}", wsk.Postep);
```

```
Postęp: 0
```

Właściwości

VB
```
Class WskaznikPostepu
    Private _postep As Integer
    Public Property Postep() As Integer
        Get
            Postep = _postep
        End Get
        Set(ByVal value As Integer)
            _postep = value
            If _postep < 0 Then _postep = 0
            If _postep > 100 Then _postep = 100
        End Set
    End Property
End Class
```

```
Dim wsk As WskaznikPostepu = New
WskaznikPostepu wsk.Postep = -10
System.Console.Write("Postęp: {0}", wsk.Postep)
```

```
Postęp: 0
```


Właściwości

C#
```
class prostokat
{
    public double x;
    public double y;
    public  double Pole
    {
        get
        {
            return x*y;
        }
    }
    public  double Obwod
    {
        get
        {
            return 2* (x+y);
        }
    }
}
```


Właściwości

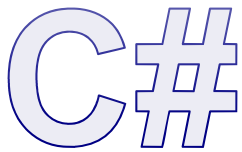
VB
```
Class prostokat
    Public x As Double
    Public y As Double
    Public ReadOnly Property Pole() As Double
        Get
            Return x * y
        End Get
    End Property
    Public ReadOnly Property Obwod() As Double
        Get
            Return 2 * (x + y)
        End Get
    End Property
End Class
```


Przestrzenie nazw

C#
```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.Show("Klik!");
}
```

```
// w pliku .cs:
this.button1.Click += new System.EventHandler(this.button1_Click);
```

VB
```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs)
                    Handles Button1.Click
    MessageBox.Show("Klik!")
End Sub
```

C#

NAZEWNICTWO
KONWENCJA
SŁOWA KLUCZOWE

Nazewnictwo

- W języku C# do nazywania klas, metod i zmiennych można korzystać z dużych i małych liter, cyfr oraz podkreślenia _. Kompilator C# odróżnia wielkość liter.
- Tworzone nazwy powinny odzwierciedlać przeznaczenie nazywanego elementu, a co za tym idzie minimalizować konieczność pisania komentarzy.
- Nazwa w C# nie może zaczynać się od cyfry.
- Nie mogą zostać również użyte słowa zastrzeżone: wartości logiczne, wartość null oraz pozostałe słowa kluczowe.

Konwencja

Kilka podstawowych zasad nazewnictwa w języku C#:

- **klasy** – rzeczownik rozpoczynający się z wielkiej litery (Ticket, Browser, User),
- **zmienne** – zwykle rzeczownik rozpoczynający się małą literą (value, numberOfTickets),
- **stałe** – rzeczownik pisany wielkimi literami, wyrazy oddzielone przez _ (MAXIMUM_VALUE_FOR_MACHINE),
- **metody** – przeważnie czasowniki i wyrażenia czasownikowe (Add, Modify, Remove),

Konwencja

Kilka podstawowych zasad nazewnictwa w języku C#:

- **nazwy wielowyrazowe** – każdy kolejny wyraz rozpoczynamy z dużej litery i bez spacji, tzw. Notacja Pascalowa (ComponentModel, HttpFileCollection) lub wielbłądzia (firstName, httpFileCollection),
- **pliki** – nazwa pliku jest taka sama jak klasy publicznej w nim zawartej,
- **przestrzenie nazw** – rzeczowniki, duże litery (System. Data).

Konwencja

- Podkreślenia są dopuszczalne, jednak zwykle w nazwach identyfikatorów nie występują podkreślenia, myślniki ani inne znaki niealfanumeryczne.
- Ponadto w języku C#, inaczej niż w starszych językach, nie korzysta się z notacji węgierskiej (polegającej na poprzedzaniu nazw skrótem reprezentującym typ danych).
  - *Pozwala to uniknąć modyfikowania nazw zmiennych po zmianie typu danych lub niespójności powstających w sytuacji, gdy programista używający notacji węgierskiej nie dostosuje odpowiednio przedrostka określającego typ.*
- Niektóre identyfikatory (jest ich niewiele), na przykład **Main**, mają w języku C# specjalne znaczenie.

Konwencja, metody

- Składniowo **metoda** w języku C# to nazwany blok kodu dodany za pomocą deklaracji metody (na przykład static void Main()), po której zwykle następują instrukcje w nawiasach klamrowych.
- Metody wykonują obliczenia lub operacje.
- Miejsce, w którym rozpoczyna się wykonywanie programów w języku C#, to **metoda Main**.
- Jej deklaracja zaczyna się od członu **static void Main()**. Gdy wywołujemy program, program się uruchomi, znajdzie metodę Main i rozpocznie wykonywanie pierwszej instrukcji.
- Choć deklaracje metody Main mogą przyjmować różną postać, zawsze potrzebny są modyfikator static i nazwa metody, Main.

Konwencja, metody

- Język C# wymaga, aby metoda Main zwracała wartość void lub wartość typu int. Ponadto ta metoda może nie przyjmować żadnych parametrów lub pobierać jedną tablicę łańcuchów znaków.

```
static int Main(string[] args)
{
//…
}
```

- Inaczej niż w starszych językach opartych na C, w języku C# w nazwie metody Main występuje wielkie *M*. Pozwala to zachować zgodność z NotacjąPascalową stosowaną w nazwach w języku C#.
- Parametr args to tablica łańcuchów znaków reprezentujących argumenty wprowadzone w wierszu poleceń.
- Jednak (inaczej niż w językach C i C++) pierwszym elementem tej tablicy nie jest nazwa programu, ale pierwszy parametr z wiersza poleceń pojawiający się po nazwie pliku wykonywalnego.
- Aby pobrać kompletne polecenie użyte do uruchomienia programu, wykorzystać można właściwość **System.Environment.CommandLine.**
- Wartość typu int zwracana tu przez metodę Main to kod statusu informujący o tym, czy wykonanie programu zakończyło się sukcesem. Niezerowa wartość zwykle oznacza błąd.

Konwencja

Wskazówki (nazwy)

- **PRZEDKŁADAJ** jasność nad zwięzłość w trakcie tworzenia identyfikatorów.
- **NIE** stosuj skrótów w nazwach identyfikatorów.
- **NIE** stosuj akronimów, chyba że są powszechnie używane. Nawet wtedy korzystaj z nich tylko w razie konieczności.
- **STOSUJ** dwie wielkie litery dla dwuliterowych akronimów. Wyjątkiem jest pierwsze słowo w identyfikatorach w notacjiWielbłąda.
- **STOSUJ** wielką literę tylko dla pierwszego znaku w akronimach obejmujących przynajmniej trzy litery. Wyjątkiem jest pierwsze słowo w identyfikatorach w notacjiWielbłąda.
- **NIE** stosuj wielkich liter w akronimach na początku identyfikatorów w notacjiWielbłąda.
- **NIE** stosuj notacji węgierskiej (nie zapisuj typu zmiennej w jej nazwie).

Słowa kluczowe

Większość słów kluczowych jest jednocześnie słowami zarezerwowanymi, tzn. takimi, których nie można używać jako identyfikatorów. Poniżej znajduje się lista wszystkich słów zarezerwowanych języka C#

- abstract
- as
- base
- bool
- break
- byte
- case
- catch
- char
- checked
- class
- const
- continue
- decimal
- default
- delegate
- do
- double
- else
- enum
- event
- explicit
- extern
- false
- finally
- fixed
- float
- for
- foreach
- goto
- if
- implicit
- in
- int
- interface
- internal
- is
- lock
- longnamespace
- new
- null
- object
- operator
- out
- override
- params
- private
- protected
- public
- readonly
- ref
- return
- sbyte
- sealed
- short
- sizeof
- stackalloc
- static
- string
- struct
- switch
- this
- throw
- true
- try
- typeof
- uint
- ulong
- unchecked
- unsafe
- ushort
- using
- virtual
- void
- volatile
- while

Kontekstowe słowa kluczowe

Niektóre słowa kluczowe są **kontekstowe**, tzn. można ich używać w roli identyfikatorów (bez przedrostka @). Oto lista tych słów:

- add
- ascending
- async
- await
- by
- descending
- dynamic
- equals
- from
- get
- global
- group
- in
- into
- join
- let
- nameof
- on
- orderby
- partial
- remove
- select
- set
- value
- var
- when
- where
- yield

Po wersji C# 1.0 do języka C# nie dodano żadnych nowych zarezerwowanych słów kluczowych.

Jednak w niektórych konstrukcjach wprowadzonych w późniejszych wersjach używane są kontekstowe słowa kluczowe, które mają specjalne znaczenie wyłącznie w określonych miejscach. Poza tymi lokalizacjami kontekstowe słowa kluczowe nie mają specjalnego znaczenia.

Dzięki temu większość kodu dostosowanego do wersji C# 1.0 jest zgodna także z późniejszymi standardami.

C#

LITERAŁY

Literały

W C# stosuje się literały, czyli znaki podawane bezpośrednio w kodzie określające typ danych (gdy nie jest on jednoznacznie określony).

Wyróżniamy 6 rodzajów literałów:

- całkowite,
- rzeczywiste,
- logiczne,
- znakowe,
- napisowe,
- literał null.

Literały

- Standardowo literały całkowite są typu **int** i mogą zostać zapisane w postaci dziesiętnej, ósemkowej oraz szesnastkowej.
- *W platformie .NET zrezygnowano z zapisu w systemie ósemkowym.*
- W przypadku **systemu szesnastkowego** wykorzystuje się przedrostek **0x** lub **0X**, po którym występuje liczba szesnastkowa (złożona z cyfr 0-9 oraz liter od a/A do f/F).
- Uzyskanie liczby całkowitej **typu long** realizuje się z wykorzystaniem przyrostka **l** lub **L** (*duża litera zapewnia lepszą czytelność*).

Literały

- Literały zmiennoprzecinkowe mogą zostać wyrażone w systemie dziesiętnym lub szesnastkowym.
- Domyślnie literały zmiennoprzecinkowe są typu double.
- Na liczbę zmiennoprzecinkową dziesiętną składa się: część całkowita, kropka, część ułamkowa, symbol wykładnika potęgi oraz jego wartość.
- Dodatkowo, podobnie jak w przypadku literałów całkowitych można dodać przyrostek **f** lub **F** celem narzucenia **typu float**.
- Można również dodać **d** lub **D** dla typu **double**, co zwiększy czytelność kodu……

Literały

- Przyrostek D jest zbędny, ponieważ wszystkie literały z kropką dziesiętną są uznawane za typ double. A kropkę dziesiętną zawsze można dodać do literału liczbowego:

  double x = 4.0;
- Najbardziej przydatne są przyrostki F i M (decimal), które powinno się zawsze stosować do literałów typu float i decimal.
- Bez przyrostka F poniższa instrukcja nie przeszłaby kompilacji, ponieważ literał 4.5 zostałby uznany za typ double, który nie obsługuje niejawnej konwersji na float:

  float f = 4.5F;

Literały

- Do reprezentowania dowolnych znaków alfanumerycznych **Unicode** stosuje się typ znakowy **char**.
- Są one umieszczane pomiędzy znakami **pojedynczego cudzysłowu** i mogą być zapisane w postaci kodu **UTF-16** w systemie dziesiętnym lub szesnastkowym (poprzedzone znakami **\u**) oraz po prostu za pomocą konkretnego znaku.

Literały

Dodatkowo występują specjalne sekwencje poprzedzone znakiem \ :

- \n – przejście do nowej linii,
- \r – powrót karetki,
- \t – tabulacja,
- \b – backspace,
- \f – nowa strona,
- \" – cudzysłów,
- \' – apostrof,
- \\ – backslash.

Literały

W celu deklaracji zmiennych typu podstawowego stosuje się zapis

nazwa_typu nazwa_zmiennej;

Z kolei do inicjalizacji korzystamy z zapisu

nazwa_zmiennej = wartość;

Inicjalizacja może zostać również dokonana od razu w miejscu deklaracji.

Literały

```
// Przykłady deklaracji i inicjacji zmiennych
int liczba = 1;
float cena;
char znak;
cena = 2.7f;
znak = 'a';
```

Literały

Typy obiektowe w C# to referencje (odnośniki) do obiektów lub wartości null.

Takie referencje mogą istnieć samodzielnie.

Wyróżnia się następujące typy obiektowe:
- typy tablicowe,
- typy klas,
- typy interfejsów,
- typy delegatów.

Literały

W C# do obsługi ciągów znaków stosuje się klasę **System.String**.

- W celu ułatwienia posługiwania się tą klasą zdefiniowano jej alias: **string**, który został awansowany do rangi słowa kluczowego języka.
- Inicjalizacja łańcucha wygląda następująco:

```
string nazwisko = "Kowalski";
```


Literały

- Obiekt stworzony przez klasę string jest reprezentowany przez konkretny, stały napis oraz posiada właściwość immutable, czyli jest niezmienny.

- **Oznacza to, że taki obiekt nie może być modyfikowany – przy próbie zmiany wartości obiektu tworzony jest nowy obiekt zainicjowany nową wartością.**

Literały

- Klasa string obsługuje konkatenację, czyli łączenie ciągów znakowych.
- Realizuje się to za pomocą operatora + lub +=.

```
string nazwisko = "Kowalski";
nazwisko += " Marian";
Console.WriteLine(nazwisko);
```


Literały

- Istnieje również możliwość łączenia ciągów tekstowych z liczbami:

```
string word = "napis";
int number = 2;
string s1 = number + number + word; //s1 => 4napis
string s2 = word + number + number; //s2 => napis22
```

```
//Przykład…
string word = "napis";
int number = 2;
string s1 = number;
```

Literały

Klasa string oferuje szereg wbudowanych metod ułatwiających operacje na tekście:
- **operator ==** nadpisanie operatora daje wartość true, jeśli dwa łańcuchy tekstowe zawierają ten sam zestaw znaków,
- **char[] ToCharArray()** – metoda zwraca tablicę znaków z danego ciągu tekstowego,
- **int IndexOf(char c)** – metoda odnajduje pierwsze wystąpienie znaku,
- **bool contains(string s)** – metoda sprawdza, czy wartość argumentu znajduje się w zadanym ciągu tekstowym i zwraca wartość logiczną,

Literały

Klasa string oferuje szereg wbudowanych metod ułatwiających operacje na tekście:

- **bool EndsWith(string s)** – metoda zwraca, czy dany łańcuch tekstowy zawiera inny,
- **int Length** – właściwość, która zlicza i zwraca liczbę znaków w zadanym ciągu,
- **string Replace(char old, char new)** – metoda tworzy obiekt typu string powstały z podstawienia w ciągu, z którego była wywołana wystąpień znaku zadanego przez pierwszy argument znakiem zadanym w drugim argumencie,
- Istnieje też: **string Replace(string old, char string)**

Literały

Klasa string oferuje szereg wbudowanych metod ułatwiających operacje na tekście:

- **string Substring(int start)** – metoda tworzy obiekt typu string, będący częścią ciągu, dla którego została wywołana. Nowy rozpoczyna się od znaku znajdującego się na pozycji zadanej argumentem,
- Istnieje też **string Substring(int start, int Length)**

Literały

Klasa string oferuje szereg wbudowanych metod ułatwiających operacje na tekście:

- **string ToLower()** – metoda zwraca nowy obiekt typu string, w którym wszystkie duże litery zostały zastąpione małymi,
- **string ToUpper()** – metoda zwraca nowy obiekt typu string, w którym wszystkie małe litery zostały zastąpione dużymi,
- **string Trim()** – metoda zwraca nowy obiekt typu string, który pozbawiony jest znaków odstępu znajdujących się na początku i końcu.

Przykłady zastosowania klasy string

```
string line = "woRID";

line = line.ToUpper(); //line => "WORLD"

string part = line.Substring(1); //part => „ORLD”

line = part.ToLower(); //part => "orld"

bool similar = line.StartsWith("or"); //similar => True
```


Przykłady zastosowania klasy string

```
using System;
class Example
{
   public static void Main()
   {
      string s1 = "The quick brown fox jumps over the lazy dog";
      string s2 = "fox";
      bool b = s1.Contains(s2);
      Console.WriteLine("'{0}' is in the string '{1}': {2}", s2, s1, b);
      if (b) {
         int index = s1.IndexOf(s2);
         if (index >= 0)
            Console.WriteLine("'{0} begins at character position {1}", s2, index + 1);
      }
   }
}
```

```
'fox' is in the string 'The quick brown fox jumps over the lazy dog': True
'fox begins at character position 17
```


C#

OPERATORY

Operatory

- W C# podobnie jak w innych językach programowania do wykonywania obliczeń, przypisania i innych operacji na danych stosuje się operatory.
- Operatory przyjmują jeden lub więcej argumentów oraz zwracają nową wartość.
- Typowe operatory w C#:

Operatory

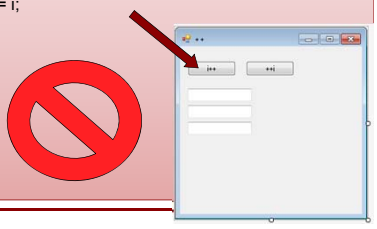
Symbol	Działanie	Przykład
+	dodawanie	liczba1 + liczba2
-	odejmowanie	liczba1 – liczba2
*	mnożenie	liczba1 * liczba2
/	dzielenie	liczba1 / liczba2
%	reszta z dzielenia	liczba1 % liczba2
++	inkrementacja przedrostkowa/przyrostkowa	++liczba, liczba++
--	dekrementacja przedrostkowa/przyrostkowa	--liczba, liczba--
!	negacja	!argument

Operatory

Symbol	Działanie	Przykład
==	równość	argument1 == argument2
!=	nierówność	argument1 != argument2
<, <=	mniejszy, mniejszy lub równy	argument1 <= argument2
>, >=	większy, większy lub równy	argument1 >= argument2
&&	koniunkcja logiczna (AND)	argument1 && argument2
||	alternatywa logiczna (OR)	argument1 || argument2
=	przypisanie	zmienna1 = zmienna2

Operatory

```
private void button1_Click(object sender, EventArgs e)
    {
        int i = 0;
        this.textBox1.Text = i++;
        this.textBox2.Text = i++;
        this.textBox3.Text = i;
    }
```



Operatory

```
private void button1_Click(object sender, EventArgs e)
    {
        int i = 0;
        this.textBox1.Text = (i++).ToString();
        this.textBox2.Text = (i++).ToString();
        this.textBox3.Text = (i).ToString();
    }
```


Operatory

```
private void button1_Click(object sender, EventArgs e)
    {
        int i = 0;
        this.textBox1.Text = i++.ToString();
        this.textBox2.Text = i++.ToString();
        this.textBox3.Text = i.ToString();
    }
```

Operatory

```
private void button2_Click(object sender, EventArgs e)
    {
        int i = 0;
        this.textBox1.Text = ++i.ToString();
        this.textBox2.Text = ++i.ToString();
        this.textBox3.Text = i.ToString();
    }
```

C#

TYPY
KONWERSJA
RZUTOWANIE

Typy

W języku C# obecne są następujące typy predefiniowane:

Typy wartościowe
- Liczbowe
- liczby całkowite ze znakiem (sbyte, short, int, long);
- liczby całkowite bez znaku (byte, ushort, uint, ulong);
- liczby rzeczywiste (float, double, decimal).
- Logiczny (bool).
- Znakowy (char).

Typy referencyjne
- Łańcuchowy (string).
- Obiektowy (object).

Konwersja

W języku C# można konwertować zgodne ze sobą typy. Wynikiem konwersji zawsze jest nowa wartość utworzona z istniejącej wartości.
Konwersja może być **niejawna** lub **jawna**.
Pierwsze odbywają się **automatycznie**, a drugie wymagają **rzutowania**.

W poniższym przykładzie niejawnie konwertujemy typ int na long (który ma dwa razy większą pojemność bitową od typu int) oraz jawnie rzutujemy typ int na short (który ma dwa razy mniejszą pojemność bitową od typu int):

```
int x = 12345; // int to 32-bitowa liczba całkowita
long y = x; // niejawna konwersja na 64-bitowy typ całkowitoliczbowy
short z = (short)x; // jawna konwersja na 16-bitowy typ całkowitoliczbowy
```


Konwersja

Konwersja **niejawna** jest dozwolona, gdy spełnione są dwa następujące warunki:
- kompilator ma gwarancję, że operacja zawsze się powiedzie;
- w wyniku operacji nie dojdzie do utraty informacji (*#znaczącej*).

Natomiast konwersja **jawna** jest konieczna, gdy spełniony jest jeden z poniższych warunków:
- kompilator nie ma gwarancji, że operacja zawsze się powiedzie;
- w wyniku konwersji może dojść do utraty informacji.

Jeżeli kompilator stwierdza, że konwersja nigdy się nie powiedzie, zabronione są oba rodzaje konwersji.

Rzutowanie

W zależności od zadanych argumentów wyniki operacji na danych mogą przyjmować typy:

int – jeśli żaden z argumentów nie jest typu double, float lub long,
long – jeśli co najmniej jeden z argumentów jest typu long i żaden nie jest typu double lub float,
float – jeśli co najmniej jeden jest typu float i żaden nie jest typu double,
double – jeśli co najmniej jeden z argumentów jest typu double.

Przykłady rzutowania

```
byte a = 0;
int b = 8;
float c = 1.2f;
double d = 10.0;
int k = a + b;          // a + b jest typu int
float m = a + b;        // a + b jest typu float
float n = b*c;          // b * c jest typu float
int p = a/k;            // a / k jest typu int
float f = d;            // zapis niepoprawny!
decimal d = -1.23       // bez przyrostka M ta instrukcja nie przejdzie kompilacji, poprawnie:
                        decimal d = -1.23M
```


Rzutowanie

- Konwersja typów odbywa się automatycznie.
- W sytuacji gdy kompilator nie może w jednoznaczny sposób ustalić typu (np. próba przypisania liczby zmienno-przecinkowej do zmiennej typu całkowitego), wówczas niezbędna jest operacja rzutowania.

Rzutowanie

- Rzutowanie realizowane jest poprzez wstawienie żądanego typu danych w nawiasach przed konkretną wartością.
- Zostanie zwrócona wartość o takim typie, na jaki rzutujemy.
- Można rzutować typ liczbowy na inny typ liczbowy oraz typ obiektowy na inny typ obiektowy (tylko na typ klasy będącej wyżej w hierarchii klas).
- Rzutowanie typów podstawowych może powodować utratę danych.

Rzutowanie

```
// Przykłady rzutowania

long a = 5;
int b = 6;
float c = 3.2f;
byte d = 56;

b = (int)(a + b);          // b => 11
a = (long)(b + c);         // a => 14
d = (byte)(d + 200);       // d => 0 (utrata informacji)
```


Przepełnienie całkowitoliczbowe

- Podczas wykonywania działań na wartościach całkowitoliczbowych może dojść do przepełnienia.
- Standardowo takie zdarzenie jest wyciszane — nie jest zgłaszany wyjątek, tylko dochodzi do „zawinięcia" wartości, tak jakby obliczenia wykonano na większym typie całkowitoliczbowym i odrzucono dodatkowy znaczący bit.
- Na przykład dekrementacja najmniejszej możliwej wartości typu int powoduje zwrócenie największej możliwej wartości tego typu:

```
int a = int.MinValue;
a--;
Console.WriteLine (a == int.MaxValue); // prawda
```


Przepełnienie całkowitoliczbowe

- Operator **checked** nakazuje systemowi wykonawczemu generowanie wyjątku **OverflowException** zamiast niepostrzeżenie doprowadzać do przepełnienia, gdy wyrażenie lub instrukcja przekroczy arytmetyczny limit wartości danego typu.
- Operator ten działa na wyrażenia z operatorami ++, --, +, - (jedno i dwuargumentowy), *, / oraz operatorami jawnej konwersji działającymi na liczbach całkowitych.
- Operator **checked** nie działa na typy double i float (których przepełnienie jest oznaczane specjalną wartością „nieskończoną", o czym szerzej piszemy nieco dalej) ani decimal (który jest zawsze kontrolowany).

Przepełnienie całkowitoliczbowe

- Operator **checked** można zastosować zarówno do wyrażenia, jak i do bloku instrukcji, np.:

```
int a = 1000000;
int b = 1000000;
int c = checked (a * b); // sprawdza tylko wyrażenie


checked // sprawdza wszystkie wyrażenia
{ // w bloku instrukcji
...
c = a * b;
...
}
```


Przepełnienie całkowitoliczbowe

- Można się posłużyć też operatorem **unchecked**. Poniższy kod nie spowoduje zgłoszenia wyjątków, nawet jeśli zostanie skompilowany z dodatkiem przełącznika kompilacji /checked+:

```
int x = int.MaxValue;
int y = unchecked (x + 1);
unchecked { int z = x + 1; }
```

- Niezależnie od ustawień przełącznika kompilatora /checked wyrażenia obliczane w czasie kompilacji są zawsze sprawdzane pod kątem występowania przepełnień — chyba że programista zastosuje operator unchecked:

```
int x = int.MaxValue + 1; // błąd kompilacji
int y = unchecked (int.MaxValue + 1); // brak błędów
```


Metoda TryParse()

- Od wersji C# 2.0 wszystkie proste typy liczbowe udostępniają statyczną metodę TryParse().
- Działa ona bardzo podobnie do metody Parse(), ale zamiast zgłaszać po nieudanej konwersji wyjątek, zwraca wartość false

```
double number;
string input;
System.Console.Write("Wprowadź liczbę: ");
input = System.Console.ReadLine();
if (double.TryParse(input, out number))
{
   // Poprawna konwersja — teraz można korzystać z liczby
   // …
}
else
{
   System.Console.WriteLine( "Wprowadzony tekst nie jest poprawną liczbą.");
}
```

```
Wprowadź liczbę: czterdzieści-dwa
Wprowadzony tekst nie jest poprawną liczbą.
```

Ważną różnicą między metodami Parse() i TryParse() jest to, że metoda TryParse() nie zgłasza wyjątku po niepowodzeniu.

C#

DEKLARACJE I ZASIĘG

Zmienne

- W C# deklarację zmiennych można tworzyć w dowolnym miejscu w kodzie.
- Zaleca się, aby zmienne były jednocześnie deklarowane i inicjowane jak najbliżej miejsca ich użycia.
- Pozwala to na zachowanie naturalnego stylu kodowania.

```
// Przykład deklaracji i inicjacji zmiennej typu String
string message = "Witamy w systemie bankowym";
```


Zmienne

- Pojęcie zasięgu określa widoczność zmiennych oraz czas ich życia w ramach tego zadanego zasięgu.
- W C# zasięg jest ograniczony przez pozycje nawiasów klamrowych { … }, między którymi znajdują się instrukcje złożone.
- Zmienna widoczna jest od miejsca jej deklaracji do końca instrukcji złożonej, w której się znajduje.
- Wewnątrz instrukcji złożonych mogą występować kolejne, zagnieżdżone instrukcje, w których nie mogą występować takie same identyfikatory zmiennych.

Zmienne

Zakres zmiennej lub stałej lokalnej obejmuje bieżący blok. W danym bloku i zagnieżdżonych w nim blokach może się znajdować tylko jedna deklaracja zmiennej lokalnej o określonej nazwie. Na przykład:

```
static void Main()
{
int x;
 {
 int y;
 int x; // błąd - zmienna x jest już zdefiniowana
 }
 {
 int y; // OK - zmiennej y nie ma w tym zakresie
 }
 Console.Write (y); // błąd - zmienna y jest poza zakresem
}
```


C#

TABLICE

Tablice

- Tablica jest strukturą danych, w której występuje ciąg (tego samego typu) obiektów albo zmiennych typu prostego.
- Zbiór takich elementów posiada nazwę, a dostęp do konkretnych wartości uzyskuje się poprzez nawiasy kwadratowe [], czyli operator indeksowania.
- Deklaracja tablicy polega na dodaniu zaraz po nazwie typu lub nazwie zmiennej operatora indeksowania (typ[] zmienna / typ zmienna[]).
- Tablice mogą być wielowymiarowe – zwiększana jest wtedy liczba nawiasów kwadratowych.

Tablice

```
// Przykłady deklaracji tabel
// tablica jednowymiarowa
        bool[] wartosciLogiczne;
// macierz (może być więcej wymiarów)
        Bilet[,] bilety;
// Tablica nieregularna - poszarpana.
        int[][] liczby;
```

- Tak zadeklarowana tablica jest jedynie referencją do tablicy, co oznacza, że pamięć dla niej nie została jeszcze zaalokowana.
- Takie odniesienie wskazuje na miejsce w pamięci, gdzie dane będą przechowywane.

Tablice

- W celu alokacji pamięci dla tablicy konieczne jest jawne podanie elementów tablicy w nawiasach klamrowych w miejscu jej deklaracji lub wykorzystanie operatora **new**.
- W przypadku drugiego rozwiązania w nawiasach kwadratowych podajemy liczbę elementów tablicy w danym wymiarze.

```
// Alokacja pamięci i wypełnianie tabel danymi

int[] liczby = new int[10];
double[,] dane = new double[3,5];

string[] porty = {"Warszawa", "Paryż", "Berlin", "Marsylia",
"Rzym", "Londyn" };
```


Tablice

- Jeśli tworzona jest tablica i nie są podawane jej elementy, to domyślnie przyjmują one wartość 0 (dla typów prostych), false (dla typów logicznych) lub null (dla typów obiektowych).
- Do przypisywania wartości elementom wykorzystuje się indeksowanie.
- W przypadku tablic dwuwymiarowych pierwszy indeks wskazuje na numer wiersza (tablica[numerWiersza, numerKolumny]).

```
// Indeksowanie tabel
int[,] tab = new int[9, 9];
tab[2, 5] = 11;
```

Tablice

```
// Dostęp do tablicy dwuwymiarowej
int m = 3;
int n = 5;
int[][] matrix = {
new[] {4, 8, 2, 1, 0},
new[] {2, 6, 1, 9, 1},
new[] {5, 7, 9, 1, 2}};
for (int i = 0; i < m; i++)
{
        for (int j = 0; j < n; j++)
        {
                Console.Write(matrix[i][j] + " ");
        }
        Console.WriteLine();
}
```


Tablice

- Możliwe jest również tworzenie tzw. **tablic poszarpanych** (ang. *jagged*).
- Są to tablice dwuwymiarowe, w których poszczególne wiersze mogą mieć inną liczbę elementów (liczba wierszy jest stała).
- Liczbę kolumn w danym wierszu uzyskuje się poprzez zastosowanie zapisu [numerWiersza].Length.

Tablice

- Dodatkowo, każda tablica posiada składową **Length** informującą o ilości elementów danej tablicy.
- W C# tablice numerowane są od zera, dlatego też maksymalnym dozwolonym indeksem tablicy jest wartość Length – 1.
- Zmienna ta – w połączeniu z indeksowaniem – jest pomocna przy realizowaniu dostępu do tablicy (zapobiega przed odwoływaniem się poza zakres tablicy).
- W przypadku odwołania do nieistniejącego elementu program zatrzyma się, wyrzucając wyjątek **IndexOutOfRangeException**.

Tablice

```
// Wykorzystanie Length
public static void Main()
{
string[] airport = { "Warszawa", "Paryż", "Berlin", "Tokio",
"Marsylia", "Rzym", "Londyn" };

for (int i = 0; i < airport.Length; i++)
        {
        Console.WriteLine(i + ". " + airport[i]);
        }
}
```


Tablice

```
// Pobieranie długości wierszy w tabelach

int[][] tab = { new int[] { 2, 4, 6 }, new int[] { 100, 5 } };
for (int i = 0; i < tab.Length; i++)
{
        for (int j = 0; j < tab[i].Length; j++)
        {
        Console.Write(tab[i][j] + " ");
        }
        Console.WriteLine();
}
```


Tablice

- Dla tablic dostępne są też dodatkowe metody przeznaczone do manipulowania elementami.
- Niektóre z tych metod to: Sort(), Reverse() i Clear().

```
class ProgrammingLanguages
{
static void Main()
{
string[] languages = new string[]{"C#", "COBOL", "Visual Basic",
"Pascal","Fortran", "Lisp", "J#"};
System.Array.Sort(languages);
string searchString = "COBOL";
int index = System.Array.BinarySearch(languages, searchString);
System.Console.WriteLine("Język przyszłości, "+ $"{ searchString }, jest dostępny pod indeksem
{ index }.");
System.Console.WriteLine();
System.Console.WriteLine($"{ "Pierwszy element",-20 }\t{ "Ostatni element",-20 }");
System.Console.WriteLine($"{ "----------",-20 }\t{ "----------",-20 }");
System.Console.WriteLine($"{ languages[0],-20 }\t{ languages[languages.Length-1],-20}");
System.Array.Reverse(languages);
System.Console.WriteLine($"{ languages[0],-20 }\t{ languages[languages.Length-1],-20}");
// Zauważ, że poniższa instrukcja nie usuwa elementów z tablicy.
// Zamiast tego do wszystkich elementów przypisywana jest wartość domyślna.
System.Array.Clear(languages, 0, languages.Length);
System.Console.WriteLine($"{ languages[0],-20 }\t{ languages[languages.Length-1],-20}");
System.Console.WriteLine($"Po wywołaniu Clear wielkość tablicy to: { languages.Length }");
}}
```

```
Język przyszłości, COBOL, jest dostępny pod indeksem 2.

Pierwszy element     Ostatni element
----------           ----------
C#                   Visual Basic
Visual Basic         C#

Po wywołaniu Clear wielkość tablicy to: 9
```


C#

KLASA OBJECT

KLASA OBJECT

- Wszystkie klasy tworzone w języku C# posiadają jeden wspólny korzeń, jakim jest klasa **Object**.
- Tworząc dowolną klasę, będzie ona dziedziczyła właśnie po klasie **Object**.
- Typ object (System.Object) reprezentuje najwyższą klasę stanowiącą podstawę wszystkich typów. Do typu object można rzutować w górę każdy inny typ.

KLASA OBJECT

Jak bardzo jest to przydatne, pokażemy na przykładzie stosu. Jest to struktura danych zgodna z zasadą LIFO (ang. last in, first out — ostatni na wejściu, pierwszy na wyjściu).

Każdy stos obsługuje dwie operacje: push do wstawiania obiektów na stos i pop do usuwania obiektów ze stosu.

Oto prosta implementacja takiej struktury, w której można przechowywać maksymalnie 10 obiektów:

```
public class Stack
{
  int position;
  object[] data = new object[10];
  public void Push (object obj) { data[position++] = obj; }
  public object Pop() { return data[--position]; }
}
```


KLASA OBJECT

```
public class Stack
{
  int position;
  object[] data = new object[10];
  public void Push (object obj) { data[position++] = obj; }
  public object Pop() { return data[--position]; }
}
```

Dzięki temu, że klasa Stack pracuje z typem object, na stosie możemy umieszczać egzemplarze dowolnego typu:

```
Stack stack = new Stack();
stack.Push ("kiełbasa");
string s = (string) stack.Pop(); // rzutowanie w dół, więc musi być wykonane jawnie
Console.WriteLine (s); // kiełbasa
```


KLASA OBJECT

Typ object to typ referencyjny, ponieważ jest typem klasowym.

Mimo to można na niego rzutować także typy wartościowe, np. int.

Jest to cecha języka C# zwana unifikacją (ang. unification), a poniże przedstawiono przykład jej wykorzystania:

```
stack.Push (3);
int three = (int) stack.Pop();
```

KLASA OBJECT

Konsekwencją takiej hierarchii jest też fakt, że wszystkie klasy w C# posiadają następujące metody, pierwotnie zaimplementowane w klasie **Object**:

- **bool Equals(Object obj)**
- **bool Equals(Object objA, Object objB)**
- **void Finalize()**
- **int GetHashCode()**
- **Type GetType()**
- **Object MemberwiseClone()**
- **ReferenceEquals(Object objA, Object objB)**
- **string ToString()**

KLASA OBJECT

bool Equals(Object obj)

- Metoda **Equals(Object obj)**, jako składowa klasy **Object**, jest dostępna dla wszystkich obiektów.
- Zadaniem tej metody jest sprawdzenie równości obiektu wywołującego i obiektu **obj**.
- Jest odpowiednikiem operatorów **==** oraz **!=** dla typów prostych.
- Dla referencji, w większości przypadków, należy stosować metodę **Equals()**, ponieważ działa w inny sposób niż operator **==**.

KLASA OBJECT

bool Equals(Object obj)

- Metoda wywoływana dla obiektu, jako parametr przyjmuje obiekt porównywany.

- Dla każdej **niepustej** referencji **object** zachodzi:
- **object.Equals(null) == false;**
- **object.Equals(object) == true;**

KLASA OBJECT

void Finalize()

- Zadaniem metody **Finalize()** jest zwalnianie zasobów użytych w danej instancji obiektu.
- Metoda oznaczona **Finalize()** jest wirtualną metodą chronioną.
- Jej implementacja polega na zdefiniowaniu destruktora klasy.
- Kompilator sam dokonuje konwersji do postaci destruktora.
- Domyślna implementacja zarówno bezparametrowego konstruktora, jak i destruktora jest pusta i automatycznie definiowana podczas kompilacji.

KLASA OBJECT

void Finalize()

- Platforma .NET jest wyposażona w mechanizm automatycznego zarządzania pamięcią.
- Klasą odpowiedzialną za porządek w dostępie do pamięci i zwalnianiu zbędnych obszarow jest **GarbageCollector**. Zarządca pamięci jest integralną częścią platformy .NET i to on, na podstawie licznika referencji (ang. *Reference counting*), rozpoznaje, czy dany obiekt może zostać zwolniony.

KLASA OBJECT

int GetHashCode()

- Zadaniem metody **GetHashCode()** jest rozróżnianie obiektów.
- Metoda na podstawie informacji zgromadzonych w obiekcie klasy, bądź z nim związanych, ustala zwracaną wartość.
- Ma to szczególne znaczenie w przypadku stosowania kolekcji przechowujących obiekty, wykorzystujących technikę haszowania.
- W trakcie kiedy program jest wykonywany, metoda dla danego obiektu zwróci tę samą wartość.

KLASA OBJECT

int GetHashCode()

- Domyślna implementacja metody nie gwarantuje unikalności wartości zwracanych dla rożnych obiektów.
- Co więcej, .NET Framework nie gwarantuje, że domyślna implementacja metody **GetHashCode()** będzie taka sama dla rożnych wersji .NET Framework.
- Metoda ta jest powiązana z metodą **Equals().**
- Nadpisanie którejkolwiek z nich wymusza na programiście nadpisanie drugiej.

KLASA OBJECT

int GetHashCode()

- Zachodzi tutaj zależność, ktora mówi, że jeśli dla dwóch obiektow wywołanie metody Equals() zwraca wartość true, to wartości wywołania na rzecz tych obiektów metody
- GetHashCode() będą równe.
- Drugie istotne założenie to fakt, że dla danego obiektu wartość zwracana z metody GetHashCode() nie może się zmieniać w trakcie działania programu.
- Zmiana taka może wystąpić po zmianie wartości obiektu.
- Dobra implementacja metody haszującej gwarantuje wydajne działanie tablicy haszującej – można osiągnąć stały czas dostępu do wybranego elementu.

KLASA OBJECT

Type GetType()

- W przypadku kiedy dysponujemy konkretnym obiektem, istnieje możliwość uzyskania szczegółowych informacji dotyczących jego typu.
- W tym celu konieczne jest określenie, do jakiej klasy obiekt należy.
- Metoda **GetType()** zwraca instancję klasy **Type** obiektu ją wywołującego.
- Obiekt zwracany poprzez metodę **GetType()** udostępnia informacje o klasie w trakcie działania programu.

KLASA OBJECT

Type GetType()

- Klasa **Type** dostarcza komplet szczegółowych danych o pochodzeniu klasy, jej metodach, polach, implementowanych interfejsach, informacji o *assembly,* do którego przynależy klasa, i wiele innych.

KLASA OBJECT

Object MemberwiseClone()

- Metoda **Object MemberwiseClone()** służy do tworzenia obiektu identycznego, jak obiekt wywołujący.
- Tak stworzona kopia nie wskazuje na obiekt kopiowany, ale jest tej samej klasy.

KLASA OBJECT

Object MemberwiseClone()

- Domyślnie realizowane jest kopiowanie płytkie, czyli bajt po bajcie.
- Jest ono wystarczające w przypadkach, gdy pola obiektu są typu prostego.
- W przypadku, gdy klasa obiektu zawiera referencje do innych klas, należy dokonać przesłonięcia metody.
- **MemberwiseClone()** jest metodą oznaczoną jako **protected**, dlatego też metoda przesłaniająca musi być publiczna.

KLASA OBJECT

Object MemberwiseClone()

- Dodatkowo, klasa, w której tworzymy metodę clone(), musi implementować interfejs ICloneable.
- Wspomniana metoda ma za zadanie utworzyć nowy obiekt identyczny z obecnym, ale kompletnie niezależny – jak łatwo się domyśleć, dla złożonych obiektów referencyjnych kopiowanie pól powinno odbywać się rekurencyjnie, aż do typów prostych.
- Od mechanizmu klonowania zasadniczo oczekujemy tworzenia „głębokich kopii" oraz relacji równoważności pomiędzy obiektem a jego klonem. Klonowanie obiektów bardzo przypomina użycie konstruktorów kopiujących w języku C++.

KLASA OBJECT

bool ReferenceEquals(Object objA, ObjectB)

- Statyczna metoda **ReferenEquals(Object objA, Object objB)** pozwala na sprawdzenie, czy dwie referencje (**objA** i **objB**) wskazują na dokładnie ten sam obiekt w pamięci.

KLASA OBJECT

string ToString()

- Metoda **ToString()** zwraca opis obiektu wywołującego – ciąg znakow, który opisuje dany obiekt.

- Domyślna implementacja metody

ToString(): return GetType().FullName;

- W przypadku wywołania na rzecz **MyClass** zwróci wartość:

PrzestrzenNazw.MyClass

KLASA OBJECT

string ToString()

- Metoda **ToString** jest niewątpliwie najczęściej wywoływaną ze wszystkich metod w aplikacjach zbudowanych na platformie .NET.
- Domyślne zachowanie operatora **+** pomiędzy obiektami i łańcuchami tekstowymi powoduje automatyczne wywoływanie metody **ToString** na wszystkich obiektach.
- Jest to niezwykle wygodne.
- Własna implementacja metody **ToString()** bardzo ułatwia programowanie w języku C#.
- **Zalecane jest wręcz, by każda klasa była w nią wyposażona.**

KLASY

Klasy

Klasa jest najczęściej używanym rodzajem typów referencyjnych. Prosty przykład klasy:

```
class NazwaKlasy
{
}
```

Klasy

Bardziej złożone klasy mogą dodatkowo zawierać następujące składniki:

Przed słowem kluczowym class

- **atrybuty** i **modyfikatory klasy**.
- Modyfikatory niezagnieżdżonych klas to: public, internal, abstract, sealed, static, unsafe oraz partial.

Za nazwą klasy generyczne parametry typu, nazwa klasy bazowej oraz interfejsy.

W klamrze składowe klasy (metody, własności, indeksatory, zdarzenia, pola, konstruktory, przeciążone operatory, typy zagnieżdżone oraz finalizator).

Pola

Pole to zmienna będąca składową klasy lub struktury.

Pola mogą mieć następujące modyfikatory:

- **Statyczny** static
- **Dostępu** public internal private protected
- **Dziedziczenia** new
- **Niebezpiecznego kodu** unsafe
- **Tylko do odczytu** readonly
- **Wątkowy** volatile

```
class Octopus
{
        string name;
        public int Age = 10;
}
```


Pola

Modyfikator readonly

Modyfikator readonly uniemożliwia zmienianie wartości zmiennej po utworzeniu obiektu.

Wartość takiemu polu można przypisać tylko w deklaracji lub konstruktorze zawierającego to pole typu.

Pola

Inicjalizacja pól

Inicjalizacja pól jest opcjonalna.

Pole niezainicjalizowane ma wartość domyślną (0, \0, null lub false).

Inicjalizatory pól są wykonywane przed konstruktorami:

```
public int Age = 10;
```


Pola

Deklarowanie wielu pól naraz

Dla wygody można zadeklarować wiele pól tego samego typu za pomocą listy elementów rozdzielanych przecinkami.

Jest to wygodny sposób deklarowania pól o takich samych atrybutach i z takimi samymi modyfikatorami. Na przykład:

```
static readonly int legs = 8,
                    eyes = 2;
```


Metody

Metoda wykonuje pewną czynność podzieloną na serię instrukcji.

Może przyjmować od wywołującego dane wejściowe w postaci parametrów oraz zwracać dane do wywołującego za pomocą typu zwrotnego.

Metoda może mieć typ zwrotny void, tzn. nie zwraca żadnej wartości do wywołującego.

Ponadto metoda może zwracać dane poprzez parametry ref i out.

Metody

Sygnatura metody nie może się powtarzać w obrębie danego typu.

Sygnatura składa się z nazwy metody i typów parametrów (nie należą do niej nazwy parametrów ani typ zwrotny).

Metody mogą mieć następujące modyfikatory:

Statyczny static

Dostępu public internal private protected

Dziedziczenia new virtual abstract override sealed

Częściowej metody partial

Kodu niezarządzanego unsafe extern

Kodu asynchronicznego async

Metody wyrażeniowe (C# 6)

Metody zawierające tylko jedno wyrażenie, takie jak ta:

```
int Foo (int x) { return x * 2; }
```

można zapisywać zwięźlej jako **metody wyrażeniowe** (ang. *expression-bodied method*). W ich składni pozbywamy się klamry i słowa kluczowego return, a dodajemy strzałkę:

```
int Foo (int x) => x * 2;
```

Funkcje wyrażeniowe mogą też mieć typ zwrotny void:

```
void Foo (int x) => Console.WriteLine (x);
```


Przeciążanie metod

Typ może przeciążać metody (zawierać kilka metod o takiej samej nazwie), pod warunkiem że każda z nich ma inną sygnaturę.

Na przykład wszystkie poniższe metody mogą się znajdować w jednym typie:

```
void Foo (int x) {...}
void Foo (double x) {...}
void Foo (int x, float y) {...}
void Foo (float x, int y) {...}
```


Przeciążanie metod

Natomiast poniższe pary metod nie mogą koegzystować w jednym typie, ponieważ typ zwrotny i modyfikator params nie wchodzą w skład sygnatury:

```
void Foo (int x) {...}
float Foo (int x) {...} // błąd kompilacji

void Goo (int[] x) {...}
void Goo (params int[] x) {...} // błąd kompilacji
```

*Za pomocą **params** — słowo kluczowe, można określić parametr metody, który przyjmuje zmienną liczbę argumentów.*

```
public class MyClass
{
    public static void UseParams1(params int[] list)
    {
        for (int i = 0; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
        Console.WriteLine();
    }

    public static void UseParams2(params object[] list)
    {
        for (int i = 0; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
        Console.WriteLine();
    }

    static void Main()
    {
        // You can send a comma-separated list of arguments of the specified type.
        UseParams1(1, 2, 3, 4);
        UseParams2(1, 'a', "test");
        // A params parameter accepts zero or more arguments.
        // The following calling statement displays only a blank line.
        UseParams2();

        // An array argument can be passed, as long as the array
        // type matches the parameter type of the method being called.
        int[] myIntArray = { 5, 6, 7, 8, 9 };
        UseParams1(myIntArray);

        object[] myObjArray = { 2, 'b', "test", "again" };
        UseParams2(myObjArray);

        // The following call causes a compiler error because the object
        // array cannot be converted into an integer array.
        // UseParams1(myObjArray);

        // The following call does not cause an error, but the entire
        // integer array becomes the first element of the params array.
        UseParams2(myIntArray);
    }
}
```

```
Output:
1 2 3 4
1 a test

5 6 7 8 9
2 b test again
System.Int32[]
```

Parametry

Metoda może mieć zbiór parametrów. Określają one argumenty, które muszą być tej metodzie przekazane.

W poniższym przykładzie metoda Foo ma zdefiniowany jeden parametr o nazwie p typu int:

```
static void Foo (int p)
{
p = p + 1; // zwiększenie wartości o jeden
Console.WriteLine (p); // drukuje wartość p na ekranie
}
static void Main()
{
Foo (8); // wywołanie Foo z argumentem 8
}
```


Parametry

Sposób przekazywania parametrów można kontrolować za pomocą modyfikatorów ref i out.

Modyfikator parametru	Sposób przekazania	Zmienna musi mieć przypisaną wartość
(Brak)	Przez wartość	Na *wejściu*
ref	Przez referencję	Na *wejściu*
out	Przez referencję	Na *wyjściu*

Przekazywanie argumentów przez wartość.

W języku C# argumenty są domyślnie **przekazywane przez wartość**, co jest zresztą najczęstszym przypadkiem. Ten sposób przekazywania wartości polega na tym, że do metody przekazywana jest kopia podanej wartości:

```
class Test
{
static void Foo (int p)
{
p = p + 1; // zwiększenie wartości o 1
Console.WriteLine (p); // drukuje wartość p na ekranie
}
static void Main()
{
int x = 8;
Foo (x); // utworzenie kopii x
Console.WriteLine (x); // x nadal będzie mieć wartość 8
}
}
```


Przekazywanie argumentów przez wartość.

Przekazanie argumentu typu referencyjnego przez wartość powoduje skopiowanie *referencji*, nie samego obiektu. W poniższym przykładzie metoda Foo pracuje na tym samym obiekcie typu StringBuilder, ktory utworzyła metoda Main, ale ma do niego własną niezależną *referencję*. Innymi słowy - sb i fooSB to osobne zmienne wskazujące ten sam obiekt typu StringBuilder:

```
class Test
{
static void Foo (StringBuilder fooSB)
{
fooSB.Append ("test");
fooSB = null;}
static void Main()
{
StringBuilder sb = new StringBuilder();
Foo (sb);
Console.WriteLine(sb.ToString()); // test
}}
```

Jako że fooSB jest *kopią* referencji, ustawienie jej na null nie pociąga za sobą takiego samego skutku dla sb.

(Gdyby jednak argument fooSB zadeklarowano i wywołano z modyfikatorem ref, to zmienna sb również *otrzymałaby* wartość null).

Modyfikator ref

Aby **przekazać argument przez referencję**, w języku C# należy posłużyć się modyfikatorem ref.

W poniższym przykładzie p i x odnoszą się do tego samego miejsca w pamięci:

```
class Test
{
static void Foo (ref int p)
{
  p = p + 1; // zwiększenie wartości o 1
  Console.WriteLine(p); // drukuje wartość p na ekranie
}
static void Main()
{
  int x = 8;
  Foo (ref x); // nakazujemy Foo pracować bezpośrednio z x
  Console.WriteLine(x); // x ma teraz wartość 9
}}
```

W tym przypadku zmiana wartości p pociąga za sobą zmianę wartości x.
Modyfikator ref musi się znajdować się zarówno w definicji, jak i w wywołaniu metody. Dzięki temu dobrze widać, co się dzieje w kodzie.

Modyfikator ref - przykład

```
class Test
{
static void Swap (ref string a, ref string b)
{
string temp = a;
a = b;
b = temp;
}
static void Main()
{
string x = „kot";
string y = „pies";
Swap (x, y);
Console.WriteLine (x);
Console.WriteLine (y);
}
}
```

Modyfikator ref - przykład

```
class Test
{
static void Swap (ref string a, ref string b)
{
string temp = a;
a = b;
b = temp;
}
static void Main()
{
string x = „kot";
string y = „pies";
Swap (ref x, ref y);
Console.WriteLine (x);
Console.WriteLine (y);
}
}
```

```
// Konsola:
pies
kot
```


Modyfikator out

Argument out jest jak argument ref z dwoma wyjątkami:

- Nie musi mieć przypisanej wartości, zanim zostanie przekazany do funkcji.
- Musi mieć przypisaną wartość, zanim **wyjdzie** z funkcji.

Modyfikator out jest najczęściej używany w celu zwrócenia więcej niż jednej wartości przez metodę.

```
class Test
{
static void Split (string name, out string firstNames, out string lastName)
{
int i = name.LastIndexOf (' ');
firstNames = name.Substring (0, i);
lastName = name.Substring (i + 1);}
static void Main()
{
string a, b;
Split ("Stevie Ray Vaughan", out a, out b);
Console.WriteLine (a); // Stevie Ray
Console.WriteLine (b); // Vaughan
}}
```


Implikacje przekazywania przez referencję

Gdy przekazujemy argument przez referencję, to tworzymy nową nazwę dla miejsca przechowywania wartości wskazywanej przez inną zmienną, a nie tworzymy nową lokalizację do przechowywania wartości.

W poniższym przykładzie zmienne x i y reprezentują ten sam egzemplarz:

```
class Test
{
static int x;
static void Main()
{
        Foo (out x);
}
static void Foo (out int y)
{
        Console.WriteLine (x); // x wynosi 0
        y = 1; // modyfikacja y
        Console.WriteLine (x); // x wynosi 1
}
}
```


Parametry opcjonalne

W C# 4.0 wprowadzono możliwość deklarowania parametrów opcjonalnych w metodach, konstruktorach i indeksatorach.

Parametr jest opcjonalny, gdy ma podaną wartość domyślną:

```
void Foo (int x = 23) { Console.WriteLine (x); }
```

Parametry opcjonalne można opuszczać w wywołaniach metody:

```
Foo(); // 23
```

W tym przypadku nastąpiło *przekazanie domyślnego argumentu* 23 do opcjonalnego parametru x - kompilator wcieli wartość 23 do skompilowanego kodu po stronie *wywołującego*.

Powyższe wywołanie metody Foo jest semantycznie identyczne z poniższym, ponieważ kompilator wstawia wartość domyślną parametru opcjonalnego zawsze, gdy jest taka potrzeba:

```
Foo (23);
```


Parametry opcjonalne

Wartość domyślna parametru opcjonalnego musi być określona przez wyrażenie stałe lub bezparametrowy konstruktor typu wartościowego.

Parametrów opcjonalnych nie można oznaczać modyfikatorami ref i out.

Parametry obowiązkowe muszą znajdować się *przed* opcjonalnymi zarówno w deklaracji metody, jak i w jej wywołaniu (wyjątkiem są argumenty params, które także w tym przypadku muszą znajdować się na końcu).

```
void Foo (int x = 0, int y = 0) { Console.WriteLine (x + ", " + y); }
void Test()
{
        Foo(1);
}
```

```
Konsola:
1, 0
```

Aby otrzymać odwrotny efekt, tzn. przekazać wartość domyślną do x, a argument do y, należałoby połączyć parametry opcjonalne z **argumentami nazwanymi**.

Argumenty nazwane

Argumenty można identyfikować nie tylko wg ich pozycji na liście, ale i wg nazw. Na przykład:

```
void Foo (int x, int y) { Console.WriteLine (x + ", " + y); }
void Test()
{
Foo (x:1, y:2); // 1, 2
}
```

Argumenty nazwane mogą występować w dowolnej kolejności.

Dwa poniższe wywołania metody Foo są semantycznie identyczne:

```
Foo (x:1, y:2);
Foo (y:2, x:1);
```

Argumenty nazwane

Jest drobna różnica polegająca na tym, że wyrażenia argumentów są obliczane w kolejności występowania w miejscu *wywołania* metody. Ma to znaczenie tylko w przypadku współzależnych wyrażeń ze skutkami ubocznymi, jak poniższe oznaczające 0, 1:

```
int a = 0;
Foo (y: ++a, x: --a); // najpierw zostanie wykonane działanie ++a
```

Oczywiście nie należy pisać takiego kodu!

Argumenty pozycyjne i nazwane można mieszać:

```
Foo (1, y:2);
```


Argumenty nazwane

Jest jednak pewne ograniczenie: argumenty pozycyjne muszą występować przed nazwanymi. Nie można więc wywołać metody Foo w następujący sposob:

```
Foo (x:1, 2); // błąd kompilacji
```

Argumenty nazwane są szczególnie przydatne w połączeniu z parametrami opcjonalnymi:

```
void Bar (int a = 0, int b = 0, int c = 0, int d = 0) { ... }
```

W jej wywołaniu można przekazać tylko wartość dla parametru d:

```
Bar (d:3);
```


Słowo kluczowe var - niejawne określanie typów zmiennych lokalnych

Zmienne często deklaruje się i inicjalizuje za jednym razem.

Jeżeli kompilator może wydedukować typ z wyrażenia inicjalizacyjnego, to w miejsce deklaracji typu można wpisać słowo kluczowe var (wprowadzone w C# 3.0). Na przykład:

```
var x = "cześć";
var y = new System.Text.StringBuilder();
var z = (float)Math.PI;
```

Te definicje zmiennych są równoważne z poniższymi:

```
string x = "cześć";
System.Text.StringBuilder y = new System.Text.StringBuilder();
float z = (float)Math.PI;
```


Słowo kluczowe var - niejawne określanie typów zmiennych lokalnych

Zmienne o niejawnie określonym typie podlegają statycznej kontroli typów. Na przykład poniższy kod spowoduje powstanie błędu kompilacji:

```
var x = 5;
x = "hello"; // błąd kompilacji; x jest typu int
```

Słowo kluczowe var może pogorszyć czytelność kodu, jeśli *odczytanie typu z samej deklaracji zmiennej jest niemożliwe*. Na przykład:

```
Random r = new Random();
var x = r.Next();
```

Jakiego typu jest zmienna x?

W typach anonimowych, w niektórych przypadkach użycie słowa kluczowego var jest obowiązkowe.

Kod niebezpieczny

Oznaczając typ, składową typu lub blok instrukcji słowem kluczowym **unsafe**, programista zapewnia sobie możliwość używania typów wskaźnikowych i wykonywania na pamięci operacji w stylu języka C++ w danym zakresie.
Oto przykład szybkiego przetwarzania mapy bitowej za pomocą wskaźników:

```
unsafe void BlueFilter (int[,] bitmap)
{
  int length = bitmap.Length;
  fixed (int* b = bitmap)
  {
    int* p = b;
    for (int i = 0; i < length; i++)
    *p++ &= 0xFF;
  }
}
```


Konstruktory egzemplarzy

Konstruktor wykonuje kod inicjalizacyjny na klasie lub strukturze. Definiuje się go jak metodę, tylko nazwę i typ zwrotny redukuje się do postaci nazwy typu, do którego konstruktor należy:

```
public class Panda
{
  string name; // definicja pola
  public Panda (string n) // definicja konstruktora
  {
    name = n; // kod inicjalizacyjny (ustawienie wartości pola)
  }
}
...
Panda p = new Panda („Halinka"); // wywołanie konstruktora
```

Konstruktory egzemplarzy mogą mieć następujące modyfikatory:
Dostępu public internal private protected
Kodu niezarządzanego unsafe extern

Przeciążanie konstruktorów

Klasy i struktury mogą przeciążać konstruktory. Aby uniknąć powielania kodu, jeden konstruktor może wywoływać inny za pomocą słowa kluczowego this:

```
using System;
public class Wine
{
  public decimal Price;
  public int Year;
  public Wine (decimal price) { Price = price; }
  public Wine (decimal price, int year) : this (price) { Year = year; }
}
```

Gdy jeden konstruktor wywołuje inny, to ten *wywoływany konstruktor* zostaje wykonany pierwszy.

Do drugiego konstruktora można przekazać *wyrażenie*:

public Wine (decimal price, DateTime year) : this (price, year.Year) { }

W wyrażeniu nie można używać referencji this, aby np. wywołać metodę egzemplarza. (Ta zasada została wprowadzona, ponieważ na tym etapie obiekt nie jest jeszcze zainicjalizowany przez konstruktor, więc wykonywanie metod na tym obiekcie nie może się udać).

Można natomiast wywoływać metody statyczne.

Niejawne konstruktory bez parametrów i kolejność inicjonowania pól

Niejawne konstruktory bez parametrów

Dla klas kompilator C# automatycznie generuje publiczny konstruktor bez parametrów, ale wtedy i tylko wtedy, gdy programista sam nie zdefiniuje żadnego konstruktora. Jeśli programista zdefiniuje jakikolwiek konstruktor, automatyczne generowanie zostaje wstrzymane.

Konstruktor i kolejność inicjalizowania pól

Polom można nadać wartości domyślne w deklaracji:

```
class Player
{
int shields = 50; // pierwsza inicjalizacja
int health = 100; // druga inicjalizacja
}
```

Inicjalizacja pól następuje *przed* wykonaniem konstruktora i w kolejności występowania deklaracji.

Konstruktory niepubliczne

Konstruktor nie musi być publiczny.

Niepubliczne konstruktory najczęściej tworzy się w celu kontrolowania procesu tworzenia egzemplarzy przez wywołania metody statycznej. Metoda ta może zwracać obiekt z puli, zamiast za każdym razem tworzyć nowy egzemplarz, albo na podstawie otrzymanych na wejściu argumentów może zwracać obiekty różnych podklas:

```
public class Klasa1
{
Klasa1() {} // konstruktor prywatny
public static Klasa1 Create (...)
{
// kod zwracający egzemplarzy klasy Klasa1
...
}
}
```


Inicjalizatory obiektów

Aby uprościć inicjalizację obiektów, wszystkie jego dostępne pola i własności można ustawiać przez **inicjalizator obiektu** bezpośrednio po zakończeniu procesu konstrukcji. Dla przykładowej klasy:

```
public class Bunny
{
public string Name;
public bool LikesCarrots;
public bool LikesHumans;
public Bunny () {}
public Bunny (string n) { Name = n; }
}
```

Przy użyciu inicjalizatorów obiektów można tworzyć obiekty klasy Bunny w następujący sposób:

// jeśli konstruktor nie ma parametrów, można opuścić pusty nawias

```
Bunny b1 = new Bunny { Name="Bo", LikesCarrots=true, LikesHumans=false };
Bunny b2 = new Bunny ("Bo") { LikesCarrots=true, LikesHumans=false };
```


Inicjalizatory obiektów

```
Bunny b1 = new Bunny { Name="Bo", LikesCarrots=true, LikesHumans=false };
Bunny b2 = new Bunny ("Bo") { LikesCarrots=true, LikesHumans=false };
```

Te instrukcje utworzenia obiektów b1 i b2 są równoznaczne z poniższym kodem:

```
Bunny temp1 = new Bunny(); // temp1 to nazwa wygenerowana przez kompilator
temp1.Name = "Bo";
temp1.LikesCarrots = true;
temp1.LikesHumans = false;
Bunny b1 = temp1;
Bunny temp2 = new Bunny ("Bo");
temp2.LikesCarrots = true;
temp2.LikesHumans = false;
Bunny b2 = temp2;
```

Zmienne tymczasowe są wykorzystywane po to, by w razie wystąpienia wyjątku podczas inicjalizacji nie pozostał nam częściowo niezainicjalizowany obiekt.

Inicjalizatory obiektów zostały wprowadzone w C# 3.0.

Referencja this

Referencja this odnosi się do samego egzemplarza. W poniższym przykładzie metoda Marry za pomocą referencji this ustawia pole Matepartnera:

```
public class Panda
{
public Panda Mate;
public void Marry (Panda partner)
{
Mate = partner;
partner.Mate = this;  }}
```

Referencja this pozwala też odróżnić zmienną lokalną lub parametr od pola.

```
public class Test
{
string name;
public Test (string name) { this.name = name; }  }
```

Referencji this można używać tylko w niestatycznych składowych klas i struktur.

Finalizatory

Finalizatory to metody klasowe wywoływane, zanim system usuwania nieużytków odzyska pamięć zajmowaną przez nieużywany obiekt. Składnia finalizatora składa się z nazwy klasy z przedrostkiem ~:

```
class Class1
{
~Class1()
{
...
}}
```

W istocie jest to składnia do przesłaniania metody Finalize klasy Object. Kompilator rozwinie tę deklarację do następującej postaci:

```
protected override void Finalize()
{
...
base.Finalize();}
```

Finalizatory mogą mieć następujące modyfikatory:
Kodu niezarządzanego unsafe

Metody i typy częściowe

Typy częściowe to technika pozwalająca dzielić definicje typów na części — najczęściej kilka różnych plików. Często spotykaną sytuacją jest wygenerowanie części klasy przez jakiś automat (np. z szablonu Visual Studio) i dodanie do niej przez programistę pozostałych potrzebnych składników. Na przykład:

```
// PaymentFormGen.cs - wygenerowany automatycznie
partial class PaymentForm { ... }
// PaymentForm.cs - napisany ręcznie
partial class PaymentForm { ... }
```

Każda część musi zawierać w deklaracji słowo kluczowe partial, tzn. poniższy kod jest nieprawidłowy:

```
partial class PaymentForm {}
class PaymentForm {}
```

Części nie mogą zawierać składowych kolidujących ze składowymi innych części. Kompilator niczego nie gwarantuje w odniesieniu do kolejności inicjalizowania pól w częściowych deklaracjach typów.

Metody częściowe

Typ częściowy może zawierać **metody częściowe**, które są dla programisty punktami zaczepienia do wpisania własnej implementacji. Na przykład:

```
partial class PaymentForm // w pliku wygenerowanym automatycznie
{
...
partial void ValidatePayment (decimal amount);
}
partial class PaymentForm // w pliku pisanym ręcznie
{
...
partial void ValidatePayment (decimal amount)
{
if (amount > 100)
...
}}
```

Metoda częściowa składa się z dwóch części: **definicji** i **implementacji**. Definicja jest zazwyczaj automatycznie tworzona przez generator kodu, a implementację pisze ręcznie programista.
Jeżeli implementacja nie zostanie dostarczona, wygenerowana definicja metody częściowej zostaje usunięta (wraz z wywołującym ją kodem). Dzięki temu można pozwolić automatowi generować większą liczbę punktów zaczepienia bez obaw, że doprowadzi to do rozdęcia kodu.

Metody częściowe muszą mieć typ zwrotny void i są niejawnie definiowane jako prywatne.

Operator nameof (C# 6)

Operator nameof zwraca nazwę symbolu (typu, składowej, zmiennej itd.) w postaci łańcucha:

```
int count = 123;
string name = nameof (count); // nazwa to "count"
```

Zaletą tego operatora w porównaniu z podaniem po prostu łańcucha jest statyczna kontrola typów. Takie narzędzia, jak np. Visual Studio rozpoznają odwołanie do symbolu, więc jeśli zmienimy interesujący nas symbol, to zmienią się także wszystkie odwołania do niego.

Jeśli trzeba określić nazwę składowej typu, np. pola lub własności, należy dodać nazwę tego typu. Ta technika działa zarówno dla składowych statycznych, jak i dla egzemplarza:

```
string name = nameof (StringBuilder.Length);
```

Wartością tego wyrażenia jest łańcuch "Length".

Aby został zwrócony łańcuch "StringBuilder.Length", należałoby napisać:

```
nameof (StringBuilder) + "." + nameof (StringBuilder.Length);
```


CECHY OOP

CECHY OOP

Można wyróżnić 4 główne cechy, które charakteryzują programowanie zorientowane obiektowo:

- **enkapsulacja,**
- **polimorfizm,**
- **dziedziczenie,**
- **abstrakcja.**

Enkapsulacja

- Enkapsulację (lub inaczej hermetyzację) można w skrócie określić jako ukrywanie implementacji.
- Oznacza to, że w OOP implementacja operacji wykonywanych przez obiekt jest ukryta przed użytkownikiem
- Program komunikuje się z danymi obiektu tylko poprzez metody tego obiektu, które chronią i przekazują wartości zmiennych pomiędzy metodami.
- Do przekazywania i przyjmowania informacji wykorzystuje się interfejsy.

Polimorfizm

- Polimorfizm (inaczej wielopostaciowość) polega na oddzieleniu wyrażeń w kategoriach typów.
- W prostszy sposób można stwierdzić, że to samo polecenie może mieć wiele znaczeń – jeden przedmiot może posiadać wiele kształtów czy kolorów. Na tym mechanizmie opiera się większość wzorców projektowych.
- Dzięki niemu metody mogą mieć te same nazwy, ale wykonywać rożne zadania.
- Polimorfizm odbywa się z wykorzystaniem:
  - dziedziczenia, przeciążenia, interfejsów.

Dziedziczenie – klasy

- Dziedziczenie jest mechanizmem charakteryzującym programowanie obiektowe, który pozwala na wielokrotne wykorzystanie klas.
- Realizowane jest to przez utworzenie nowej klasy w oparciu o już istniejącą.
- Klasa bazowa posiada ogólne przeznaczenie, natomiast każda nowa klasa jest bardziej specjalistyczna.

- Teoretycznie w drzewie dziedziczenia można umieścić nieograniczoną liczbę klas.
- C# jest językiem, w którym obowiązuje **dziedziczenie po jednym typie** (dotyczy to także języka CIL, do którego kompilowany jest kod w języku C#). To oznacza, że klasa nie może dziedziczyć bezpośrednio po dwóch klasach.


```
public class Pdaltem : object
{
// ...
}
public class Appointment : Pdaltem
{
// ...
}
public class Contact : Pdaltem
{
// ...
}
public class Customer : Contact
{
//
}
```

Klasy dziedziczące jedna po drugiej tworzą łańcuch dziedziczenia.

W wyniku dziedziczenia każda składowa klasy bazowej występuje też w łańcuchu klas pochodnych.

Jeśli nie została podana inna klasa bazowa, wszystkie klasy domyślnie dziedziczą po klasie **object**.

W klasach pochodnych nie ma dostępu do składowych zadeklarowanych w klasie bazowej jako prywatne.

- W rzadkich sytuacjach, w których potrzebna jest struktura klas z wielodziedziczeniem, możliwym rozwiązaniem jest zastosowania **agregacji**.
- Wtedy zamiast tworzyć jedną klasę jako pochodną od innej, można umieścić w danej klasie instancję typu drugiej klasy.

Dziedziczenie

```
public class PdaItem
{
// …
}
public class Person
{
// …
}
```

```
public class Contact : PdaItem
{
private Person InternalPerson { get; set; }
public string FirstName
{
get { return InternalPerson.FirstName; }
set { InternalPerson.FirstName = value; }
}
public string LastName
{
get { return InternalPerson.LastName; }
set { InternalPerson.LastName = value; }
}
// …
}
```


CECHY OOP

Konstruktory a dziedziczenie

- Przy definiowaniu konstruktora klasy pochodnej należy użyć specjalnej konstrukcji **base()**.
- Konstrukcja ta jest wywołaniem konstruktora klasy bazowej (wraz z jego argumentami).

```
public class Zwierze
{
  protected string nazwa;
  public Zwierze(string nazwa)
  {
    this.nazwa = nazwa;
  }
}
```

```
public class Kot : Zwierze
{
  protected string rasaKota;
  public Kot(string rasaKota) : base("kot")
  {
    this.rasaKota = rasaKota;
  }
}
```


CECHY OOP

Virtual i override

- Budowanie hierarchii klas doprowadza często programistę do sytuacji, gdzie klasa pochodna ma zaimplementowaną metodę, której podstawowa implementacja już istnieje w klasie bazowej.
- Może wówczas pojawić się naturalne pytanie o to, która z metod zostanie wywołana? Czy ta z klasy bazowej, czy nadrzędnej?

CECHY OOP

Virtual i override

- Otóż domyślnie wywoływana jest metoda wynikająca z zadeklarowanego typu obiektu.
- Jeśli chcemy, aby kompilator „wiedział", że ma wywoływać metodę z klasy obiektu, a nie z klasy bazowej – musi ona być w klasie bazowej określona jako **virtual**, natomiast w klasie dziedziczącej oznaczona jako **override.**

CECHY OOP

Abstrakcja

- Abstrakcja jest to pewien wzorzec, który może zostać użyty do tworzenia konkretnych instancji czy zachowań.
- W C# istnieją klasy oraz metody abstrakcyjne nie posiadające konkretnej implementacji, opisujące tylko pewien ogólny rodzaj klas czy metod.

CECHY OOP

Klasa abstrakcyjna

- Klasa abstrakcyjna jest to klasa, która nie posiada instancji w postaci obiektów.
- Najczęściej stosowana jest do tworzenia interfejsów – modelowania na ogólnym poziomie.
- Klasa abstrakcyjna posiada jedynie zachowania i pola charakterystyczne dla danego typu.
- Nie można utworzyć obiektów tej klasy, ale można po niej dziedziczyć.
- Klasę abstrakcyjną definiuje się przez poprzedzenie nazwy klasy słowem kluczowym **abstract**.

Metoda abstrakcyjna

- Metoda abstrakcyjna jest metodą, w której nazwa jest poprzedzona słowem kluczowym **abstract**.
- Metoda taka nie posiada implementacji – występuje tylko nagłówek zakończony średnikiem.
- Nie można definiować abstrakcyjnych konstruktorów oraz metod statycznych.
- Oznaczenie dowolnej metody jako abstrakcyjnej automatycznie powoduje, że cała klasa jest abstrakcyjna.


```
public abstract class Podroz
{
    public void Start()
    {
        Console.WriteLine("Miłej podróży");
    }
    public abstract void Podrozuj();
}

public class PodrozOkretem : Podroz
{
    public override void Podrozuj()
    {
        Console.WriteLine("Płynę statkiem");
    }
    ……..
}
```



```
abstract class FiguraGeometryczna
{
        public abstract void Inicjalizuj(double[] wspolrzedne);
        public abstract double ObliczPolePowierzchni();
}
class Okrag : FiguraGeometryczna
{
  public double x1, y1, r;
  public override void Inicjalizuj(double[] wspolrzedne)
  {
    x1 = wspolrzedne[0];
    y1 = wspolrzedne[1];
    r = wspolrzedne[2];
  }
  public override double ObliczPolePowierzchni()
  {
    return Math.PI*r*r;
  }
}
```



```
abstract class FiguraGeometryczna
{
        public abstract void Inicjalizuj(double[] wspolrzedne);
        public abstract double ObliczPolePowierzchni();
}
class Prostokat :FiguraGeometryczna
{
   public double x1, y1, x2, y2;
   public override void Inicjalizuj(double[] wspolrzedne)
   {
     x1 = wspolrzedne[0];
     y1 = wspolrzedne[1];
     x2 = wspolrzedne[2];
     y2 = wspolrzedne[3];
   }
  public override double ObliczPolePowierzchni()
  {
    return Math.Abs(x2 - x1)*Math.Abs(y2 - y1);
  }}
```



```
abstract class FiguraGeometryczna
{
        public abstract void Inicjalizuj(double[] wspolrzedne);
        public abstract double ObliczPolePowierzchni();
}
class Trojkat : FiguraGeometryczna
{
   public double x1, y1, x2, y2, x3, y3;
   public override void Inicjalizuj(double[] wspolrzedne)
   {
     x1 = wspolrzedne[0];    y1 = wspolrzedne[1];
     x2 = wspolrzedne[2];    y2 = wspolrzedne[3];
     x3 = wspolrzedne[4];    y3 = wspolrzedne[5];
   }
  public override double ObliczPolePowierzchni()
  {
    return 0.5 *Math.Abs(x1*y2 + x2*y3 + x3*y1 - x3*y2 - x1*y3-x2*y1);
  }
}
```



```
class TestFigurGeometrycznych
{
  public static void Main()
  {
    List<FiguraGeometryczna> figury = new List<FiguraGeometryczna>();
    Prostokat prostokat = new Prostokat(); prostokat.Inicjalizuj(new double[]{0, 0, 5, 10});
    Trojkat trojkat = new Trojkat();   trojkat.Inicjalizuj(new double[] { 0, 0, 1, 0, 0, 1 });
    Okrag okrag = new Okrag();      okrag.Inicjalizuj(new double[] { 0, 0, 4});
    figury.Add(prostokat);
    figury.Add(trojkat);
    figury.Add(okrag);
    foreach (FiguraGeometryczna figura in figury)
    {
      Console.WriteLine("Pole powierzchni = " +
      figura.ObliczPolePowierzchni());
    }
} }
```

CECHY OOP

KLASA ZAGNIEŻDŻONA

- C# umożliwia tworzenie klas zagnieżdżonych.
- Są to klasy, które mogą być umieszczane wewnątrz innych klas.
- Zasięg pojedynczej klasy zagnieżdżonej jest ograniczony zasięgiem klasy zawierającej.
- Dostęp do klasy wewnętrznej wynika z użytych kwalifikatorów dostępu.

CECHY OOP

```
class Outside
{
 private static int number = 1;
 internal void Method()
 {
  Inside inside = new Inside();
  inside.Dolt();
 }
 class Inside
 {
  internal void Dolt()
  {
   Console.WriteLine(„Witam ze środka“);
   Console.WriteLine(„Wartosc z zewnątrz: „ + number);
  }
 }
}
```

```
public static void Main(string[] ars)
{
  Outside outside = new Outside();
  outside.Method();
}

Wynik działania powyższego programu:
// Witam ze środka
// Wartosc z zewnątrz: 1
```


SPECYFIKATORY KLAS

W C# przy definicji klas używa się specyfikatorów, które poprzedzają słowo kluczowe class:

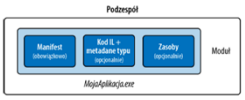
- **abstract** – tworzona klasa jest abstrakcyjna, nie posiada wystąpień,
- **internal** – klasy dostępne jedynie w danym assembly – bez wyspecyfikowania, jest to domyślna sytuacja,
- **public** – charakteryzuje klasy dostępne publicznie, czyli z każdego miejsca w programie bądź z innych programów, które zawierają referencję do assembly klasy,
- **sealed** – specyfikator określający, że dana klasa nie może być dziedziczona,
- **static** – nie można utworzyć ich instancji.

Podzespół

Podzespół to podstawowa jednostka wdrożeniowa na platformie .NET Framework i jednocześnie kontener dla wszystkich typów.

Podzespół zawiera skompilowane typy wraz z ich kodem IL (Intermediate Language), zasobami środowiska uruchomieniowego oraz informacjami pomagającymi w wersjonowaniu, zapewnianiu bezpieczeństwa i odwoływaniu się do innych podzespołów.

Ponadto podzespół definiuje granice dla ustalania typu oraz uprawnień.



Podzespół przechowywany w pojedynczym pliku

Podzespół

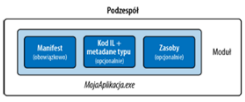
Podzespół zawiera cztery rodzaje elementów:

Manifest podzespołu

- Dostarcza informacje o środowisku uruchomieniowym platformy .NET Framework, takiejak: nazwa podzespołu, jego wersja, wymagane uprawnienia oraz inne podzespoły, do których się odwołuje.

Manifest aplikacji

- Dostarcza systemowi operacyjnemu informacje takie jak sposób wdrożenia podzespołu oraz wskazuje, czy wymagane są uprawnienia administratora.



Podzespół przechowywany w pojedynczym pliku

Podzespół

Podzespół zawiera cztery rodzaje elementów:

Skompilowane typy

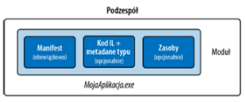
- Skompilowany kod IL oraz metadane typów zdefiniowanych w podzespole.

Podzespoły

- Inne dane osadzone wewnątrz podzespołu, takie jak obrazy oraz lokalizowane ciągi tekstowe.

Z wymienionych powyżej tylko *manifest podzespołu* jest obowiązkowy, choć podzespół niemal zawsze zawiera skompilowane typy (jeżeli nie jest odwołaniem podzespołu w środowisku WinRT).

Struktura podzespołu jest podobna niezależnie od tego, czy to plik wykonywalny, czy biblioteka.



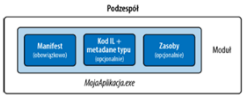
Podzespół przechowywany w pojedynczym pliku

Podzespół

- Manifest aplikacji to plik w formacie XML dostarczający systemowi operacyjnemu informacje o podzespole.
- Manifest aplikacji .NET ma element główny o nazwie assembly zdefiniowany w przestrzeni nazw XML urn:schemas-microsoft-com:asm.v1:

```
<?xml version="1.0" encoding="utf-8"?>
        <assembly manifestVersion="1.0"
xmlns="urn:schemas-microsoft-com:asm.v1">
        <!-- zawartość manifestu -->
```

- </assembly>

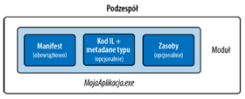


Podzespół przechowywany w pojedynczym pliku

Podzespół

Przedstawiony poniżej manifest informuje system operacyjny o konieczności żądania uprawnień administratora:

```
<?xml version="1.0" encoding="utf-8"?>
<assembly manifestVersion="1.0" xmlns="urn:schemas-microsoft-
com:asm.v1">
<trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
<security>
<requestedPrivileges>
<requestedExecutionLevel level="requireAdministrator" />
</requestedPrivileges>
</security>
</trustInfo>
</assembly>
```



Podzespół przechowywany w pojedynczym pliku

SPECYFIKATORY DLA PÓL I METOD

W przypadku pól i metod specyfikatory pełnią głównie funkcję określającą poziom dostępu.

Wyróżniamy następujące specyfikatory:

- **public** – oznacza, że dana składowa klasy jest publiczna; jest to równoznaczne ze stwierdzeniem, że jest ona dostępna z każdego miejsca programu,
- **private** – oznacza, że składowa jest prywatna; będzie ona dostępna jedynie wewnątrz klasy, w której została zdefiniowana, i ukryta dla całej reszty programu,

SPECYFIKATORY DLA PÓL I METOD

- **protected** – pole, właściwość lub metoda klasy jest chroniona; dostęp do składowej jest możliwy w klasach dziedziczących po klasie, tak, jakby to była ich składowa prywatna; poza hierarchią klasy dostęp do składowej jest niemożliwy,

SPECYFIKATORY DLA PÓL I METOD

- **static**:
  - pole statyczne – pole ze specyfikatorem **static** będzie wspólne dla wszystkich obiektów klasy, a dane istnieć będą niezależnie od istnienia obiektu,
  - metoda lub właściwość statyczna – metody i właściwości ze specyfikatorem **static** mogą być wywoływane na rzecz klasy bez tworzenia instancji obiektu; w metodach tego typu nie można odwoływać się do niestatycznych elementów klasy,

SPECYFIKATORY DLA PÓL I METOD

- **readonly** – zabezpiecza pole klasy przed zmianą przypisanej mu referencji, przypomina w działaniu słowo kluczowe **const**, ale w odniesieniu do typów referencyjnych,
- **internal** – zastosowanie tego kwalifikatora zabezpiecza dostęp tylko dla obiektów z tego samego podzespołu (*assembly*), do pola, właściwości lub metody klasy,

SPECYFIKATORY DLA PÓL I METOD

- **protected internal** – jest to hybryda oznaczająca sumę dostępności wynikających z dwóch specyfikatorów; klasy dziedziczące mają dostęp do składowej klasy wg dostępu chronionego, a dodatkowo, w ramach danego podzespołu składowe są dostępne dla wszystkich klas.

NOWOŚĆ w C# 7.2

- **private protected** wskazuje, że element może być używany przez zawierające klasy lub klasy pochodne, które są zadeklarowane w tym samym zestawie. Gdy **protected internal** zezwala na dostęp klas pochodnych **lub** klasy, które znajdują się w tym samym zestawie, private protected ogranicza dostęp do typów pochodnych zadeklarowanych w tym samym zestawie.

SPECYFIKATORY DLA PÓL I METOD

- Poza przedstawionymi kwalifikatorami istnieje także słowo kluczowe **const**.
- Słowo to może poprzedzać w deklaracji dowolne pole klasy, uzupełniając informację wynikającą z kwalifikatorów dostępu.
- Znaczenie słowa **const** jest takie, że dane pole może utrzymać niezmienną wartość – kłopot w tym, że ta wartość musi być stała w czasie kompilacji – czyli **inicjalizowana inline**.

SPECYFIKATORY DLA PÓL I METOD

```
public class Test
{
public static int number = 0;
public static void Add(int nb)
{
  number = number + nb;
  Console.WriteLine(number);
}
public static void Main()
{
  Test.number = Test.number + 200;
  Console.WriteLine(Test.number);

  Test.Add(100);
  Console.WriteLine(Test.number);
}
}
```

Powyższy program wyświetli następujące wartości:
// 200
// 300
// 300

SPECYFIKATORY DLA PÓL I METOD

```
class Point { protected int x; protected int y; }

class DerivedPoint: Point
{
  static void Main()
  {
   DerivedPoint dpoint = new DerivedPoint();
   dpoint.x = 10;
   dpoint.y = 15;
   Console.WriteLine("x = {0}, y = {1}", dpoint.x, dpoint.y);
  }
}
```

// Output: x = 10, y = 15

SPECYFIKATORY DLA PÓL I METOD

```
using System;
namespace Protected_Specifier
{
  class access
  {
    protected string name;
    public void print()
    {
      Console.WriteLine("\nMy name is " + name);
    }
  }
  class Program
  {
    static void Main(string[] args)
    {
      access ac = new access();
      Console.Write("Enter your name:\t");
      ac.name = Console.ReadLine();
      ac.print();
      Console.ReadLine();
    }
  }
}
```

// błąd: name jest protected

SPECYFIKATORY DLA PÓL I METOD

```
using System;
namespace Protected_Specifier
{
  class access
  {
    protected string name;
    public void print()
    {
      Console.WriteLine("\nMy name is " + name);
    }
  }
  class Program : access // Inherit access class
  {
    static void Main(string[] args)
    {
      Program p=new Program();
      Console.Write("Enter your name:\t");
      p.name = Console.ReadLine();
      p.print();
      Console.ReadLine();
    }
  }
}
```

// ok

SPECYFIKATORY DLA PÓL I METOD

```
using System;
namespace Internal_Access_Specifier
{
  class access
  {
    internal string name;
    public void print()
    {
      Console.WriteLine("\nMy name is " + name);
    }
  }
  class Program
  {
    static void Main(string[] args)
    {
      access ac = new access();
      Console.Write("Enter your name:\t");
      ac.name = Console.ReadLine();
      ac.print();
      Console.ReadLine();
    }
  }
}
```

// ok

SPECYFIKATORY DLA PÓL I METOD

```
class A
{
   protected int x = 123;
}

class B : A
{
   static void Main()
   {
      A a = new A();
      B b = new B();

      a.x = 10;
   }
}
```

// Error CS1540, because x can only be accessed by classes derived from A.

SPECYFIKATORY DLA PÓL I METOD

```
class A
{
   protected int x = 123;
}

class B : A
{
   static void Main()
   {
      A a = new A();
      B b = new B();

      b.x = 10;
   }
}
```

// OK

SPECYFIKATORY DLA PÓL I METOD

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

   class DerivedPoint: Point
   {
      static void Main()
      {
         DerivedPoint dpoint = new DerivedPoint();

         dpoint.x = 10;
         dpoint.y = 15;
         Console.WriteLine("x = {0}, y = {1}", dpoint.x, dpoint.y);
      }
   }
```

// błąd

SPECYFIKATORY DLA PÓL I METOD

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

   class DerivedPoint: Point
   {
      static void Main()
      {
         DerivedPoint dpoint = new DerivedPoint();

         dpoint.x = 10;
         dpoint.y = 15;
         Console.WriteLine("x = {0}, y = {1}", dpoint.x, dpoint.y);
      }
   }
```

// Output: x = 10, y = 15

SPECYFIKATORY DLA PÓL I METOD

```
class Point
   {
      protected int x;
      protected int y;
   }

   class DerivedPoint: Point
   {
      static void Main()
      {
         DerivedPoint dpoint = new DerivedPoint();

         dpoint.x = 10;
         dpoint.y = 15;
         Console.WriteLine("x = {0}, y = {1}", dpoint.x, dpoint.y);
      }
   }
```

// Output: x = 10, y = 15

SPECYFIKATORY DLA PÓL I METOD

```
class Person
{
   public string Name { get; set; }
   public int Age { get; set; }

   public string ToString()
   {
     return "Person: " + Name + " " + Age;
   }
}

namespace WindowsFormsApplication1 {
   public partial class Form1 : Form
   {
      public Form1()
      {   InitializeComponent(); }
      private void Form1_Load(object sender, EventArgs e)   {
         Person p=new Person();
         p.Name="Kris";
         p.Age=39;
         Text = p.ToString();
}}}
```

// Błąd, dlaczego?

SPECYFIKATORY DLA PÓL I METOD

```
class Person
{
   public string Name { get; set; }
   public int Age { get; set; }

   public override string ToString()
   {
     return "Person: " + Name + " " + Age;
   }
}

namespace WindowsFormsApplication1 {
   public partial class Form1 : Form
   {
      public Form1()
      {   InitializeComponent(); }
      private void Form1_Load(object sender, EventArgs e)   {
         Person p=new Person();
         p.Name="Kris";
         p.Age=39;
         Text = p.ToString();
}}}
```

// Kris 39

POLIMORFIZM CD

Polimorfizm cd

- Polimorfizm pozwala zdefiniować w podklasie jej własną wersję metody zdefiniowanej w jej klasie bazowej, za pomocą procesu nazywanego przesłanianiem metody (*method overriding*).

Polimorfizm cd

- Jeśli chcemy w klasie bazowej zdefiniować metodę, która może (ale nie musi) być przesłaniana przez podklasę, musimy oznaczyć ją słowem kluczowym **virtual**.
- Modyfikator **override** jest wymagany, aby rozszerzyć lub zmodyfikować abstrakcyjną (**abstract**) lub **wirtualną** (virtual) implementację metody odziedziczone a także właściwość lub zdarzenie (event).
- Zwróćmy uwagę, że każda przesłonięta metoda może wykorzystywać domyślne działanie za pomocą słowa kluczowego **base**.
- Dodatkowo, warto pamiętać, że słowo kluczowe **sealed** uniemożliwia innym typom rozszerzania zachować danej klasy przez dziedziczenie (tzw. pieczętowanie składowych klasy).

Polimorfizm cd

- W języku C# można użyć udogodnienia będącego logicznym przeciwieństwem przesłaniania – chodzi o cieniowanie (ang. shadowing).
- Z formalnego punktu widzenia, jeśli w klasie pochodnej zdefiniowano składową, która jest taka sama, jak składowa w klasie bazowej, klasa pochodna *cieniuje* wersję nadrzędną.
- W praktyce zdarza się, że tworzymy podklasy z klas, których nie wykonaliśmy sami – czasem nie mamy też dostępu do tych klas i nie dopiszemy słowa **virtual**……zatem **override** nie zdaje egzaminu……
- Wówczas można załączyć słowo kluczowe **new** do problematycznej składowej. Słowo kluczowe **new** można zastosować do dowolnego typu składowego odziedziczonego po klasie bazowej (pola, stałej, składowej statycznej, właściwości itp.).
- Robiąc to, jawnie stwierdzamy, że implementacja typu pochodnego ma ukryć wersję nadrzędną (*ucinamy pępowinę ☺ ???*).

Platforma .NET 283

SPECYFIKATORY DLA PÓL I METOD

```
using System;
namespace ConsoleApplication1
{
   class KlasaBazowa
   {
      public void Metoda_1()
      {
      Console.WriteLine("Klasa bazowa - Metoda_1");
      }
   }

   class KlasaPochodna : KlasaBazowa
   {
      public void Metoda_2()
      {
      Console.WriteLine("Klasa pochodna - Metoda_2");
      }
   }
```

```
class Program
{
  static void Main(string[] args)
  {
   KlasaBazowa kbazowa = new KlasaBazowa();
   KlasaPochodna kpochodna = new KlasaPochodna();
   KlasaBazowa kbazowa_kpochodna = new KlasaPochodna();

   kbazowa.Metoda_1();
   kpochodna.Metoda_1();
   kpochodna.Metoda_2();
   kbazowa_kpochodna.Metoda_1();
                        //tylko metoda 1 dostępna
    Console.ReadKey();
}}}
```

```
Klasa bazowa - Metoda_1
Klasa bazowa - Metoda_1
Klasa pochodna - Metoda_2
Klasa bazowa - Metoda_1
```

Platforma .NET 284

SPECYFIKATORY DLA PÓL I METOD

```
using System;
namespace ConsoleApplication1
{
   class KlasaBazowa
   {
      public void Metoda_1()
      {
      Console.WriteLine("Klasa bazowa - Metoda_1");
      }
      public void Metoda_2()
      {
      Console.WriteLine("Klasa bazowa - Metoda_2");
      }
   }

   class KlasaPochodna : KlasaBazowa
   {
      public void Metoda_2() //warning.....
      {
      Console.WriteLine("Klasa pochodna - Metoda_2");
      }
   }
```

```
class Program
{
  static void Main(string[] args)
  {
   KlasaBazowa kbazowa = new KlasaBazowa();
   KlasaPochodna kpochodna = new KlasaPochodna();
   KlasaBazowa kbazowa_kpochodna = new KlasaPochodna();

   kbazowa.Metoda_1();
   kbazowa.Metoda_2();
   kpochodna.Metoda_1();
   kpochodna.Metoda_2();
   kbazowa_kpochodna.Metoda_1();
   kbazowa_kpochodna.Metoda_2();

    Console.ReadKey();
}}}
```

```
Klasa bazowa - Metoda_1
Klasa bazowa - Metoda_2
Klasa bazowa - Metoda_1
Klasa pochodna - Metoda_2
Klasa bazowa - Metoda_1
Klasa bazowa - Metoda_2
```

Platforma .NET 285

SPECYFIKATORY DLA PÓL I METOD

```
using System;
namespace ConsoleApplication1
{
   class KlasaBazowa
   {
      public void Metoda_1()
      {
      Console.WriteLine("Klasa bazowa - Metoda_1");
      }
      public void Metoda_2()
      {
      Console.WriteLine("Klasa bazowa - Metoda_2");
      }
   }

   class KlasaPochodna : KlasaBazowa
   {
      public new void Metoda_2()
      {
      Console.WriteLine("Klasa pochodna - Metoda_2");
      }
   }
```

```
class Program
{
  static void Main(string[] args)
  {
   KlasaBazowa kbazowa = new KlasaBazowa();
   KlasaPochodna kpochodna = new KlasaPochodna();
   KlasaBazowa kbazowa_kpochodna = new KlasaPochodna();

   kbazowa.Metoda_1();
   kbazowa.Metoda_2();
   kpochodna.Metoda_1();
   kpochodna.Metoda_2();
   kbazowa_kpochodna.Metoda_1();
   kbazowa_kpochodna.Metoda_2();

    Console.ReadKey();
}}}
```

```
Klasa bazowa - Metoda_1
Klasa bazowa - Metoda_2
Klasa bazowa - Metoda_1
Klasa pochodna - Metoda_2
Klasa bazowa - Metoda_1
Klasa bazowa - Metoda_2
```

Platforma .NET 286

SPECYFIKATORY DLA PÓL I METOD

```
using System;
namespace ConsoleApplication1
{
   class KlasaBazowa
   {
      public void Metoda_1()
      {
      Console.WriteLine("Klasa bazowa - Metoda_1");
      }
      public void Metoda_2()
      {
      Console.WriteLine("Klasa bazowa - Metoda_2");
      }
   }

   class KlasaPochodna : KlasaBazowa
   {
      public override void Metoda_1()
      {
      Console.WriteLine("Klasa pochodna- Metoda_1");
      }
      public new void Metoda_2()
      {
      Console.WriteLine("Klasa pochodna - Metoda_2");
      }
   }
```

```
class Program
{
  static void Main(string[] args)
  {
   KlasaBazowa kbazowa = new KlasaBazowa();
   KlasaPochodna kpochodna = new KlasaPochodna();
   KlasaBazowa kbazowa_kpochodna = new KlasaPochodna();

   kbazowa.Metoda_1();
   kbazowa.Metoda_2();
   kpochodna.Metoda_1();
   kpochodna.Metoda_2();
   kbazowa_kpochodna.Metoda_1();
   kbazowa_kpochodna.Metoda_2();

    Console.ReadKey();
}}}
```

```
Błąd, bo metoda_1 w klasie bazowej nie jest
ani wirtualna, ani abstrakcyjna.....
```

Platforma .NET 287

SPECYFIKATORY DLA PÓL I METOD

```
using System;
namespace ConsoleApplication1
{
   class KlasaBazowa
   {
      public virtual void Metoda_1()
      {
      Console.WriteLine("Klasa bazowa - Metoda_1");
      }
      public void Metoda_2()
      {
      Console.WriteLine("Klasa bazowa - Metoda_2");
      }
   }

   class KlasaPochodna : KlasaBazowa
   {
      public override void Metoda_1()
      {
      Console.WriteLine("Klasa pochodna- Metoda_1");
      }
      public new void Metoda_2()
      {
      Console.WriteLine("Klasa pochodna - Metoda_2");
      }
   }
```

```
class Program
{
  static void Main(string[] args)
  {
   KlasaBazowa kbazowa = new KlasaBazowa();
   KlasaPochodna kpochodna = new KlasaPochodna();
   KlasaBazowa kbazowa_kpochodna = new KlasaPochodna();

   kbazowa.Metoda_1();
   kbazowa.Metoda_2();
   kpochodna.Metoda_1();
   kpochodna.Metoda_2();
   kbazowa_kpochodna.Metoda_1();
   kbazowa_kpochodna.Metoda_2();

    Console.ReadKey();
}}}
```

```
Klasa bazowa - Metoda_1
Klasa bazowa - Metoda_2
Klasa pochodna - Metoda_1
Klasa pochodna - Metoda_2
Klasa pochodna - Metoda_1
Klasa bazowa - Metoda_2
```

Platforma .NET 288

SPECYFIKATORY DLA PÓL I METOD

```
using System;
namespace ConsoleApplication1
{
   class KlasaBazowa
   {
      public virtual void Metoda_1()
      {
      Console.WriteLine("Klasa bazowa - Metoda_1");
      }
      public void Metoda_2()
      {
      Console.WriteLine("Klasa bazowa - Metoda_2");
      }
   }

   class KlasaPochodna : KlasaBazowa
   {
      public override void Metoda_1()
      {
      Console.WriteLine("Klasa pochodna- Metoda_1");
      }
      public new void Metoda_2()
      {
      Console.WriteLine("Klasa pochodna - Metoda_2");
      }
   }
```

```
class Program
{
  static void Main(string[] args)
  {
   KlasaBazowa kbazowa = new KlasaBazowa();
   KlasaPochodna kpochodna = new KlasaPochodna();
   KlasaBazowa kbazowa_kpochodna = new KlasaPochodna();

   kbazowa.Metoda_1();
   kbazowa.Metoda_2();
   kpochodna.Metoda_1();
   kpochodna.Metoda_2();
   kbazowa_kpochodna.Metoda_1();
   kbazowa_kpochodna.Metoda_2();

   ((KlasaBazowa)kpochodna).Metoda_1();
   ((KlasaBazowa)kpochodna).Metoda_2();

    Console.ReadKey();
}}}
```

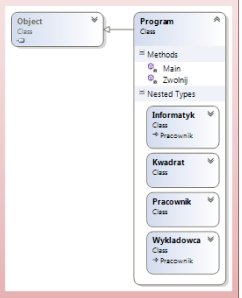
```
Klasa bazowa - Metoda_1
Klasa bazowa - Metoda_2
Klasa pochodna - Metoda_1
Klasa pochodna - Metoda_2
Klasa pochodna - Metoda_1
Klasa bazowa - Metoda_2
Klasa pochodna - Metoda_1
Klasa bazowa - Metoda_2
```

Polimorfizm cd

- Na koniec należy pamiętać, że nadal istnieje możliwość wyzwalania implementacji klasy bazowej ze składowej cieniowanej za pomocą jawnego rzutowania.

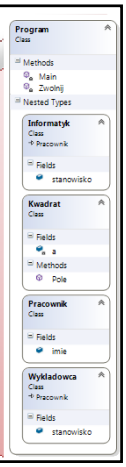
SPECYFIKATORY DLA PÓL I METOD

```
using System;
namespace ConsoleApplication2
{  class Program
   {
      public class Pracownik
      {
         public string imie;
      }
      public class Informatyk : Pracownik
      {
         public string stanowisko = "informatyk";
      }
      public class Wykladowca : Pracownik
      {
         public string stanowisko = "wykladowca";
      }
      public class Kwadrat
      {
         double a;
         public double Pole()
         { return a * a; }       }
      static void Zwolnij(Pracownik prac)
      {
         //usuń z bazy
         //zabierz ołówek i temperówkę
      }
```



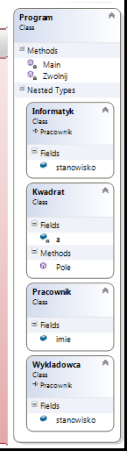
SPECYFIKATORY DLA PÓL I METOD

```
static void Main(string[] args)
   {
      object krzysiek = new Wykladowca();
```



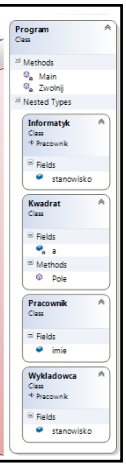
SPECYFIKATORY DLA PÓL I METOD

```
static void Main(string[] args)
   {
      object krzysiek = new Wykladowca(); //OK
      krzysiek.imie = "Krzysiek";
```



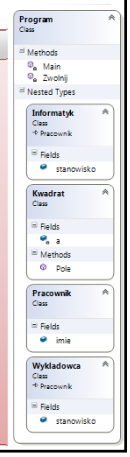
SPECYFIKATORY DLA PÓL I METOD

```
static void Main(string[] args)
   {
      object krzysiek = new Wykladowca(); //OK
      // krzysiek.imie = "Krzysiek"; // błąd, bo w object nie ma zmiennej imie
```



SPECYFIKATORY DLA PÓL I METOD

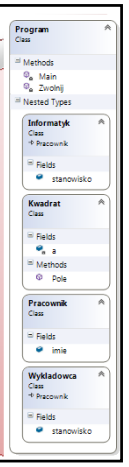
```
static void Main(string[] args)
   {
      object krzysiek = new Wykladowca(); //OK
      // krzysiek.imie = "Krzysiek"; // błąd, bo w object nie ma zmiennej imie

      Pracownik franek = new Informatyk();
      franek.imie = "Franek";
      franek.stanowisko = "Programista";

      Informatyk wojtek = new Informatyk();
      wojtek.imie = "wojtek";
      wojtek.stanowisko = "Programista";

      Wykladowca piotrek = new Wykladowca();
```



Platforma .NET

SPECYFIKATORY DLA PÓL I METOD

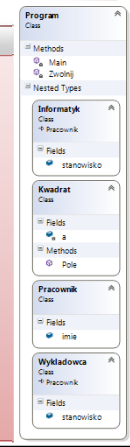
```
static void Main(string[] args)
    {
      object krzysiek = new Wykladowca(); //OK
     // krzysiek.imie = "Krzysiek"; // błąd, bo w object nie ma zmiennej imie

      Pracownik franek = new Informatyk();
      franek.imie = "Franek";
      // franek.stanowisko = "Programista"; // błąd, bo w Pracownik nie ma stanowiska

      Informatyk wojtek = new Informatyk();
      wojtek.imie = "wojtek";
      wojtek.stanowisko = "Programista";

      Wykladowca piotrek = new Wykladowca();

      Zwolnij(franek);
      Zwolnij(wojtek);
      Zwolnij(piotrek);
      Zwolnij(krzysiek);
```



Platforma .NET

SPECYFIKATORY DLA PÓL I METOD

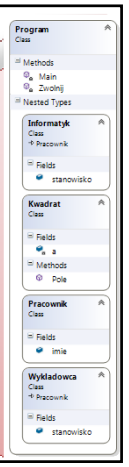
```
static void Main(string[] args)
    {
      object krzysiek = new Wykladowca(); //OK
     // krzysiek.imie = "Krzysiek"; // błąd, bo w object nie ma zmiennej imie

      Pracownik franek = new Informatyk();
      Informatyk wojtek = new Informatyk();
      Wykladowca piotrek = new Wykladowca();

      Zwolnij(franek);
      Zwolnij(wojtek);
      Zwolnij(piotrek);
      Zwolnij(krzysiek);

//1.
// dzięki relacji pokrewieństwa możemy przekazać do metody potomków klasy Pracownik;
// wykorzystamy rzutowanie niejawne
// jak widać powyżej dostaliśmy niezłe narzędzie do budowania klas i ich pochodnych oraz ich metod
```



Platforma .NET

SPECYFIKATORY DLA PÓL I METOD

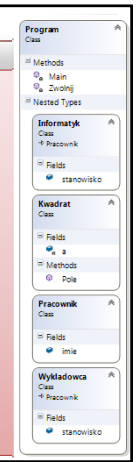
```
static void Main(string[] args)
    {
      object krzysiek = new Wykladowca(); //OK
     // krzysiek.imie = "Krzysiek"; // błąd, bo w object nie ma zmiennej imie

      Pracownik franek = new Informatyk();
      Informatyk wojtek = new Informatyk();
      Wykladowca piotrek = new Wykladowca();

      Zwolnij(franek);
      Zwolnij(wojtek);
      Zwolnij(piotrek);
      ~~Zwolnij(krzysiek);~~

//1.
// dzięki relacji pokrewieństwa możemy przekazać do metody potomków klasy Pracownik;
// wykorzystamy rzutowanie niejawne
// jak widać powyżej dostaliśmy niezłe narzędzie do budowania klas i ich pochodnych oraz ich metod
//2.
//dla Zwolnij(krzysiek); kompilator zgłosi błąd, choć obiekt może mieć kompatybilny typ...
// jak zatem zwolnić krzyśka?
```



Platforma .NET

SPECYFIKATORY DLA PÓL I METOD

```
static void Main(string[] args)
    {
      object krzysiek = new Wykladowca(); //OK

      Pracownik franek = new Informatyk();
      Informatyk wojtek = new Informatyk();
      Wykladowca piotrek = new Wykladowca();

      Zwolnij(franek);
      Zwolnij(wojtek);
      Zwolnij(piotrek);

      // może zatem jawne rzutowanie w dół
      Zwolnij((Wykladowca)krzysiek);

      //Rzutowanie jawne odbywa się w czasie wykonywania kodu!!!
      //Czy może się nie udać????

      Object k = new Kwadrat();
      Zwolnij((Wykladowca)k);

     //W trakcie wykonania: Nie można rzutować obiektu typu 'Kwadrat' na typ 'Wykladowca'.
```

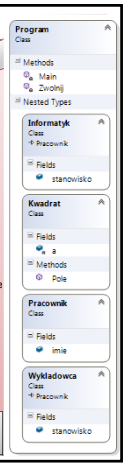


Platforma .NET

SPECYFIKATORY DLA PÓL I METOD

```
static void Main(string[] args)
    {
      object krzysiek = new Wykladowca(); //OK
      Pracownik franek = new Informatyk();
      franek.imie = "Franek";
      Informatyk wojtek = new Informatyk();
      Wykladowca piotrek = new Wykladowca();

      Zwolnij(franek);        Zwolnij(wojtek);        Zwolnij(piotrek);

      // może zatem jawne rzutowanie w dół
      Zwolnij((Wykladowca)krzysiek);

      //Może się nie udać, zatem można wykorzystać wyjątki lub...
      //Użyc słowa kluczowego as, który sprawdza kompatybilność i zwraca ewentualnie null, w czasie
      Kwadrat kwadrat = krzysiek as Kwadrat;
      if (kwadrat == null)
        Console.WriteLine("Krzysiek nie jest kwadratem");
      Pracownik prac = krzysiek as Pracownik;
      if (prac != null)
        Console.WriteLine("Krzysiek jest pracownikiem");
      //można użyć też is
      if(piotrek is Pracownik)
        Console.WriteLine("Piotrek jest pracownikiem");

      Console.ReadKey();     } } }
```

```
Krzysiek nie jest kwadratem
Krzysiek jest pracownikiem
Piotrek jest pracownikiem
```



Platforma .NET

300

SPECYFIKATORY DLA PÓL I METOD

```
using System;

namespace ConsoleApplication2
{
  class Program
  {
    public class Pracownik
    {
      public string imie;
    }
    public class Informatyk : Pracownik
    {
      public string stanowisko = "informatyk";
    }
    public class Wykladowca : Pracownik
    {
      public string stanowisko = "wykladowca";
    }
    public class Kwadrat
    {
      double a;
      public double Pole()
      {
        return a * a;
      }
    }

    static void Zwolnij(Pracownik prac)
    {
      //usuń z bazy
      //zabierz ołówek i temperówkę
    }
```

```
    static void Main(string[] args)
    {
      object krzysiek = new Wykladowca();
      // krzysiek.imie = "Krzysiek"; // błąd, bo w object nie ma zmiennej imie
      Pracownik franek = new Informatyk();
      franek.imie = "Franek";
      //franek.stanowisko = "Programista";

      Informatyk wojtek = new Informatyk();
      Wykladowca piotrek = new Wykladowca();

      // dzięki relacji pokrewieństwa możemy przekazać do metody potomków klasy Pracownik;
      // wykorzystamy rzutowanie niejawne
      // jak widać poniżej dostaliśmy niezłe narzędzie do budowania klas i ich pochodnych oraz ich metod
      Zwolnij(franek);
      Zwolnij(wojtek);
      Zwolnij(piotrek);
      //Zwolnij(krzysiek); kompilator zgłosi błąd, choć obiekt może mieć kompatybilny typ...
      // może zatem jawne rzutowanie w dół
      Zwolnij((Wykladowca)krzysiek);
      //Rzutowanie jawne odbywa się w czasie wykonywania kodu!!!
      //Zatem można wykorzystać wyjątki lub...
      //Użyc słowa kluczowego as, który sprawdza kompatybilność i zwraca ewentualnie null, w czasie działania kodu


      //Object k = new Kwadrat();
      //Zwolnij((Wykladowca)k); // błąd w trakcie wykonywania

      Kwadrat kwadrat = krzysiek as Kwadrat;
      if (kwadrat == null)
        Console.WriteLine("Krzysiek nie jest kwadratem");
      Pracownik prac = krzysiek as Pracownik;
      if (prac != null)
        Console.WriteLine("Krzysiek jest pracownikiem");
      //można użyć też is
      if(piotrek is Pracownik)
        Console.WriteLine("Piotrek jest pracownikiem");

      Console.ReadKey();

    }
  }
}
```

SPECYFIKATORY DLA PÓL I METOD

```
class GraphicsClass
{
   public virtual void DrawLine() { }
   public virtual void DrawPoint() { }
   public virtual string DrawRectangle(int q)
   {
      return "prostokąt";
   }
}
class YourDerivedGraphicsClass : GraphicsClass
{
   public override string DrawRectangle(int q)
   {
      return "kwadrat";
   }
   public string DrawRectangle(double q)
   {
      return "kwadrat";
   }
} // zmiana w bazie może wpłynąć naszej klasy
```


PRZYKŁADY

(KOLEJNE ZAJĘCIA)

Test

```
using System;
namespace ConsoleApplication3
{
   class Program
   {
      public class Pracownik
      {
         public override string ToString()
         {
            return "klasa Pracownik";
         }
      }
      public class Wykladowca:Pracownik
      {
         public override string ToString()
         {
            return "klasa Wykladowca";
         }
      }
```

```
static void Main(string[] args)
      {
         Pracownik krzysiek = new Wykladowca();

         Console.WriteLine(krzysiek.ToString());

         Console.ReadKey();
      }
   }
}
```

```
klasa Wykladowca
```


Test

```
using System;
namespace ConsoleApplication3
{
   class Program
   {
      public class Pracownik
      {
         public override string ToString()
         {
            return "klasa Pracownik";
         }
      }
      public class Wykladowca:Pracownik
      {
         public new string ToString()
         {
            return "klasa Wykladowca";
         }
      }
```

```
static void Main(string[] args)
      {
         Pracownik krzysiek = new Wykladowca();

         Console.WriteLine(krzysiek.ToString());

         Console.ReadKey();
      }
   }
}
```

```
klasa Pracownik
```


Test

```
using System;
namespace ConsoleApplication3
{
   class Program
   {
      public class Pracownik
      {

         public new string ToString()
         {
            return "klasa Pracownik";
         }
      }

      public class Wykladowca:Pracownik
      {

      }
```

```
static void Main(string[] args)
      {
         object krzysiek = new object();

         Console.WriteLine(krzysiek.ToString());

         Console.ReadKey();
      }
   }
}
```

```
System.Object
```


Test

```
using System;
namespace ConsoleApplication3
{
   class Program
   {
      public class Pracownik
      {

         public override string ToString()
         {
            return "klasa Pracownik";
         }
      }

      public class Wykladowca:Pracownik
      {

      }
```

```
static void Main(string[] args)
      {
         object krzysiek = new Wykladowca();

         Console.WriteLine(krzysiek.ToString());

         Console.ReadKey();
      }
   }
}
```

```
klasa Pracownik
```

Test

```
using System;
namespace ConsoleApplication3
{
   class Program
   {
      public class Pracownik
      {

         public new string ToString()
         {
            return "klasa Pracownik";
         }
      }

      public class Wykladowca:Pracownik
      {

      }
```

```
static void Main(string[] args)
      {
         object krzysiek = new Wykladowca();

         Console.WriteLine(krzysiek.ToString());

         Console.ReadKey();
      }
   }
}
```

```
ConsoleApplication3.Program+Wykladowca
```


INTERFEJSY – KOMUNIKACJA

INTERFEJSY – KOMUNIKACJA

- Interfejsy to zbiory operacji opisujących zachowania – ustalają, jakie żądania można wywołać pod adresem obiektu.

Nie zmienia to faktu, że gdzieś będzie musiał istnieć kod, który te żądania będzie realizował ☺

- Interfejsy służą do zdefiniowania sygnatur metod.
- Do pojedynczej klasy można dołączyć jeden lub wiele interfejsów.
- Klasa, która implementuje interfejs, dostarcza definicji wszystkich jego metod – inaczej – staje się klasą abstrakcyjną.

INTERFEJSY – KOMUNIKACJA

Definicja interfejsu :

```
interface NazwaInterfejsu
{
///blok definicji
}
```

Interfejs musi być publiczny. W bloku definicji interfejsu znajdują się deklaracje stałych (pól interfejsu) oraz nagłówki metod.

Kod metod definiowany jest w klasach, które implementują ten interfejs.

Wszystkie pola interfejsu są stałymi publicznymi i statycznymi, natomiast wszystkie metody są publiczne i abstrakcyjne.

W platformie .NET bardzo została przyjęta konwencja, aby nazwy interfejsów zaczynały się z dużej litery.

INTERFEJSY – KOMUNIKACJA

Dziedziczenie – interfejsy

- Podobnie jak klasy, interfejsy mogą być dziedziczone. Interfejs nie może dziedziczyć po klasie, ale może po innym interfejsie.
- Tutaj, podobnie jak w klasach, używa się w tym celu dwukropka:

interface INazwa : InterfejsBazowy

- W przeciwieństwie do dziedziczenia klas, w przypadku interfejsów możliwe jest dziedziczenie wielokrotne.

INTERFEJSY – KOMUNIKACJA

```
using System;
 public interface IShape
 {
  // Dla uproszczenia - brak innych metod.
  double Area();
  int Sides { get; }
 }
 public interface IShapeDisplay
 {
  void Display();
 }
```

INTERFEJSY – KOMUNIKACJA

```
public class Square : IShape, IShapeDisplay
{
 private int InSides;
 public int SideLength;

 public int Sides
 {
  get { return InSides; }
 }

 public double Area()
 {
  return ((double)
          (SideLength * SideLength));
 }
```

```
 public double Circumference()
 {
  return ((double) (Sides * SideLength));
 }

 public Square()
 {
  InSides = 4;
 }

  public void Display()
 {
 Console.WriteLine("\nDisplaying Square information:");
 Console.WriteLine("Side length: {0}", this.SideLength);
 Console.WriteLine("Sides: {0}", this.Sides);
 Console.WriteLine("Area: {0}", this.Area());
 }
}
```


INTERFEJSY – KOMUNIKACJA

```
public class Multi
{
 public static void Main()
 {
  Square mySquare = new Square();
  mySquare.SideLength = 7;

  mySquare.Display();
 }
}
```


INTERFEJSY – KOMUNIKACJA

```
public interface IPływanie
{
    void Płyń();
}

public interface IBieganie
{
    void Biegnij();
}

public interface IStrzelaj
{
    void Strzelaj();
}
.....................
```


INTERFEJSY – KOMUNIKACJA

```
public class Pływak : Sportowiec, IPływanie
{
  public Pływak(String imię, String nazwisko) : base(imię, nazwisko)
  {}
  public void Płyń()
  {
     Console.WriteLine("Płynę");
  }
  public override void WalczOMedal()
  {
     Płyń();
  }
}
```


INTERFEJSY – KOMUNIKACJA

```
public class Maratonczyk : Sportowiec, IBieganie
{
 public Maratonczyk(String imię, String nazwisko): base(imię, nazwisko)
 {
 }
 public void Biegnij()
 {
    Console.WriteLine(„Biegnę w maratonie");
 }
 public override void WalczOMedal()
 {
    Biegnij();
 }
}
```


INTERFEJSY – KOMUNIKACJA

```
public class Pięcioboista : Sportowiec, IBieganie, Ipływanie, IStrzelanie
{
public Pięcioboista(String imię, String nazwisko) : base(imię, nazwisko)
{}
public void Biegnij()
{
Console.WriteLine(„Rozpoczynam konkurencję: bieg z przeszkodami");
}
```

INTERFEJSY – KOMUNIKACJA

```
public void Płyń()
{
Console.WriteLine(„Płynę stylem dowolnym“);}
public void Strzelaj()
{
Console.WriteLine(„Strzelam“);}
public void RozpocznijSzermierkę()
{
Console.WriteLine(„Rozpoczynam konkurencję szermierki“);}
public void RozpocznijJazdęKonną()
{
Console.WriteLine(„Rozpoczynam konkurencję: jazda konna“);}
public override void WalczOMedal()
{
Płyń(); RozpocznijSzermierkę(); RozpocznijJazdęKonną(); Biegnij(); Strzelaj();}}
```


INTERFEJSY – Przykład

```
// Oto definicja interfejsu IEnumerator z przestrzeni nazw System.Collections:
public interface IEnumerator
{
bool MoveNext();
object Current { get; }
void Reset();
}
// Implementacja interfejsu polega na zdefiniowaniu publicznych implementacji
// wszystkich jego składowych:
internal class Countdown : IEnumerator
{
int count = 11;
public bool MoveNext() => count-- > 0;
public object Current => count;
public void Reset() { throw new NotSupportedException(); }
}
```


INTERFEJSY – Przykład

```
internal class Countdown : IEnumerator
{
int count = 11;
public bool MoveNext() => count-- > 0;
public object Current => count;
public void Reset() { throw new NotSupportedException(); }
}

// Obiekt można niejawnie rzutować na typ dowolnego interfejsu,
// który implementuje. Na przykład:

IEnumerator e = new Countdown();
while (e.MoveNext())
  Console.Write (e.Current); // 109876543210
```


INTERFEJSY – Rozszerzanie interfejsu

```
// Interfejsy mogą być tworzone na podstawie innych interfejsów. Na przykład:

public interface IPierwszy { void Metoda1(); }
public interface IDrugi : IPierwszy { void Metoda2(); }

// Interfejs IDrugi „dziedziczy” wszystkie składowe z interfejsu IPierwszy.
// Innymi słowy: typy implementujące IDrugi muszą implementować też
// składowe interfejsu IPierwszy.
```


INTERFEJSY – Jawna implementacja interfejsu

```
// Implementacja wielu interfejsów może skutkować kolizją sygnatur składowych.
// W takim przypadku rozwiązaniem problemu jest jawna implementacja
// składowej.
interface I1 { void Metoda(); }
interface I2 { int Metoda(); }
public class MojaKlasa: I1, I2
{
public void Metoda()
{
Console.WriteLine ("Implementacja składowej I1.Metoda w klasie MojaKlasa.");
}
int I2. Metoda()
{
Console.WriteLine ("Implementacja składowej I2.Metoda w klasie MojaKlasa.");
return 42;
}}
```

OK,
wszystko fajnie, ale jak wywołać jedną i drugą metodę…

INTERFEJSY – Jawna implementacja interfejsu

Ze względu na konflikt sygnatur metod **Metoda** z interfejsów I1 i I2 w klasie MojaKlasa w sposób bezpośredni zdefiniowano implementację tej metody z interfejsu I2.

Dzięki temu metody te mogą współistnieć w jednej klasie. Jedynym sposobem na wywołanie jawnie zaimplementowanej składowej jest dokonanie rzutowania na typ jej interfejsu:

```
MojaKlasa egzemplarz = new MojaKlasa ();

egzemplarz.Metoda(); // Implementacja metody I1.Metoda z klasy MojaKlasa.

((I1)egzemplarz ).Metoda(); // Implementacja metody I1.Metoda z klasy MojaKlasa.
((I2)egzemplarz ).Metoda(); // Implementacja metody I2.Metoda z klasy MojaKlasa.
```

INTERFEJSY – Wirtualna implementacja składowych interfejsu

Niejawnie zaimplementowana składowa interfejsu jest domyślnie zapieczętowana. W klasie bazowej musi więc zostać oznaczona modyfikatorem dostępu virtual lub abstract, aby można ją było przesłonić.

```
public interface IUndoable { void Undo(); }
public class TextBox : IUndoable
{public virtual void Undo() =>  ("TextBox.Undo");}
public class RichTextBox : TextBox
{public override void Undo() => Console.WriteLine ("RichTextBox.Undo");}
```

Dla wywołania składowej interfejsu przez klasę bazową lub interfejs wybierana jest implementacja z podklasy:

```
RichTextBox r = new RichTextBox();
r.Undo(); // RichTextBox.Undo
((IUndoable)r).Undo(); // RichTextBox.Undo
((TextBox)r).Undo(); // RichTextBox.Undo
```

Jawnie zaimplementowana składowa interfejsu nie może być oznaczona modyfikatorem virtual ani nie można jej przesłonić w normalny sposób. Istnieje jednak możliwość jej **reimplementowania**.

INTERFEJSY – Podsumowanie

- Interfejsy to inny rodzaj typów danych. Są one jedną z odmian typów, które nie dziedziczą po klasie System.Object (ma to znaczenie tylko teoretyczne ☺).
- **Interfejsy, w odróżnieniu od klas, nie umożliwiają tworzenia instancji.**
- Instancja interfejsu jest dostępna tylko za pomocą referencji do obiektu z implementacją danego interfejsu.
- **Nie można użyć operatora new do interfejsu. Związane jest z tym to, że interfejsy nie mogą zawierać konstruktorów ani finalizatorów.**
- Ponadto w interfejsach nie można tworzyć składowych statycznych.
- **Interfejsy przypominają więc klasy abstrakcyjne. Obie te struktury nie pozwalają tworzyć instancji.**

Porównanie klas abstrakcyjnych i interfejsów

Klasy abstrakcyjne	Interfejsy
Nie można bezpośrednio tworzyć ich instancji. Trzeba utworzyć instancję klasy pochodnej.	Nie można bezpośrednio tworzyć ich instancji. Trzeba utworzyć instancję typu z implementacją interfejsu.
Klasa pochodna od abstrakcyjnej albo sama musi być abstrakcyjna, albo musi zawierać implementację wszystkich składowych abstrakcyjnych.	W typach z implementacją interfejsu trzeba zaimplementować wszystkie jego składowe.
Można dodawać nowe składowe nieabstrakcyjne, dziedziczone w klasach pochodnych. Nie powoduje to naruszenia zgodności z wcześniejszą wersją kodu.	Dodanie nowych składowych do interfejsu narusza zgodność z wcześniejszą wersją kodu.
Można deklarować zarówno właściwości, jak i pola.	Można deklarować właściwości, ale już nie pola.
Można tworzyć składowe instancji, wirtualne, abstrakcyjne lub statyczne. Można też dodać implementację składowych nieabstrakcyjnych; zostanie ona wykorzystana w klasach pochodnych.	Wszystkie składowe są automatycznie traktowane jak abstrakcyjne, dlatego nie mogą obejmować implementacji.
Klasa pochodna może dziedziczyć tylko po jednej klasie bazowej.	W typie z implementacją można zaimplementować wiele interfejsów.

INTERFEJSY – Podsumowanie

- Interfejsy są ważnym aspektem programowania obiektowego w języku C#. Zapewniają podobny mechanizm jak klasy abstrakcyjne, ale nie mają ograniczenia w postaci możliwości dziedziczenia tylko po jednym typie. W jednej klasie można zaimplementować wiele interfejsów.

Wskazówki

- **PRZEDKŁADAJ** klasy nad interfejsy. Korzystaj z klas abstrakcyjnych do oddzielenia kontraktów (opisujących, co typ robi) od szczegółów implementacji (określających, jak typ to robi).
- **ROZWAŻ** zdefiniowanie interfejsu, jeśli chcesz dodać obsługę funkcji z interfejsu w typach, które dziedziczą już po innym typie.
- Interfejsów używaj do reprezentacji typów o niezależnych implementacjach.

STRUKTURY

Struktury

- Prawie wszystkie wbudowane typy języka C#, na przykład bool i decimal, to typy bezpośrednie.
- Wyjątkami są typy referencyjne string i object. W platformie dostępnych jest też wiele dodatkowych typów bezpośrednich.
- Ponadto programiści mogą definiować własne typy tego rodzaju.
- Aby zdefiniować niestandardowy typ bezpośredni, należy zastosować składnię podobną do tej używanej do definiowania klas i interfejsów.
- Najważniejszą różnicą jest to, że w definicji typu bezpośredniego należy podać słowo kluczowe **struct**,

Struktury

Struktury są podobne do klas, ale różnią się od nich następującymi cechami:

- Struktura jest typem wartościowym, podczas gdy klasa jest typem referencyjnym.
- Struktury nie mogą dziedziczyć po innych strukturach (nie licząc faktu, że niejawnie pochodzą od klasy object, a dokładniej System.ValueType).
- Struktura może mieć te same składowe co klasa, z wyjątkiem:
  - konstruktora bezparametrowego,
  - inicjalizatorów pól,
  - finalizatora,
  - składowych wirtualnych i chronionych.

Struktury należy użyć, gdy potrzebny jest typ o semantyce wartościowej. Dobrymi przykładami struktur są typy liczbowe, w przypadku których skopiowanie wartości przy przypisaniu jest bardziej naturalne niż referencji.

Struktury - Semantyka tworzenia struktur

Semantyka tworzenia struktury przedstawia się następująco:

- Bezparametrowy konstruktor, którego nie można przesłonić, jest generowany niejawnie. Przeprowadza on bitowe zerowanie wszystkich pól.
- Jeśli programista zdefiniuje konstruktor struktury, musi jawnie przypisać wartość każdemu polu.
- W strukturach nie można się posługiwać inicjalizatorami pól.

Poniżej przykład deklaracji i wywołania konstruktorów struktury:

```
public struct Point
{
int x, y;
public Point (int x, int y) { this.x = x; this.y = y; }
}
...
Point p1 = new Point (); // p1.x i p1.y będą miały wartość 0
Point p2 = new Point (1, 1); // p1.x i p1.y będą miały wartość 1
```


Struktury - Semantyka tworzenia struktur

Policz błędy w poniższym przykładzie:

```
public struct Point
{
int x = 1;
int y;
public Point() {}
public Point (int x) {this.x = x;}
}
```


Struktury - Semantyka tworzenia struktur

Policz błędy w poniższym przykładzie:

```
public struct Point
{
int x = 1; // nieprawidłowe: inicjalizator pola
int y;
public Point() {} // nieprawidłowe: konstruktor bezparametrowy
public Point (int x) {this.x = x;} // nieprawidłowe: brak wartości dla pola y
}
```

Gdyby zmieniono słowo kluczowe struct na class, błędy zniknęłyby ☺


```
struct Angle
    {
        public Angle(int degrees, int minutes, int seconds)
        {
            Degrees = degrees;   Minutes = minutes; Seconds = seconds;
        }
//Używanie przeznaczonych tylko do odczytu i automatycznie implementowanych właściwości z języka C# 6.0.
        public int Degrees { get; }
        public int Minutes { get; }
        public int Seconds { get; }
        public Angle Move(int degrees, int minutes, int seconds)
        {
            return new Angle(
            Degrees + degrees,
            Minutes + minutes,
            Seconds + seconds);
        }
    }
```

```
// Deklaracja klasy, czyli typu referencyjnego.
class Coordinate
        {
            public Angle Longitude { get; set; }
            public Angle Latitude { get; set; }
        }
...
Coordinate c = new Coordinate();
c.Latitude = c.Latitude.Move(300, 12, 45);
int d = c.Latitude.Degrees;
```



```
struct Angle
    {
     public Angle(int degrees, int minutes, int seconds)
       {
          _Degrees = degrees;
          _Minutes = minutes;
          _Seconds = seconds;
       }
       public int Degrees { get { return _Degrees; } }
       readonly private int _Degrees;
       public int Minutes { get { return _Minutes; } }
       readonly private int _Minutes;
       public int Seconds { get { return _Seconds; } }
       readonly private int _Seconds;
       // …
    }
```

// Tworzenie właściwości tylko do odczytu
// w starszej wersji….

```
private protected class Coordinate
        {
            private Angle _longitude;
            private Angle _latitude;
            public Angle Longitude { get => _longitude; set => _longitude = value; }
            public Angle Latitude { get => _latitude; set => _latitude = value; }
        }
```

WYJĄTKI

WYJĄTKI

Duża część błędów może zostać wykryta na etapie kompilacji.

Kompilator jest w stanie sprawdzić poprawność leksykalną kodu.

Podczas kompilacji odbywa się sprawdzenie mnóstwa rzeczy, np.:

- czy odwołania do klas, metod właściwości są poprawne
- czy programista dokonał dozwolonej konwersji typów
- czy wywołania metod miały prawidłową liczbę argumentów
- czy zmienne użyte w metodach i klasach były odpowiednio wcześniej zainicjalizowane

WYJĄTKI

Istnieje jednak wiele typów błędów, których odnalezienie bądź przewidzenie przez kompilator jest niemożliwe.

Przykładami takich błędów mogą być:

- dzielenie przez zero dla wczytanych liczb
- błąd w operacjach I/O podczas odczytu danych z dysku
- przerwanie strumienia sieciowego w trakcie komunikacji
- podanie w polu interpretowanym jako liczba wartości nieliczbowej

WYJĄTKI

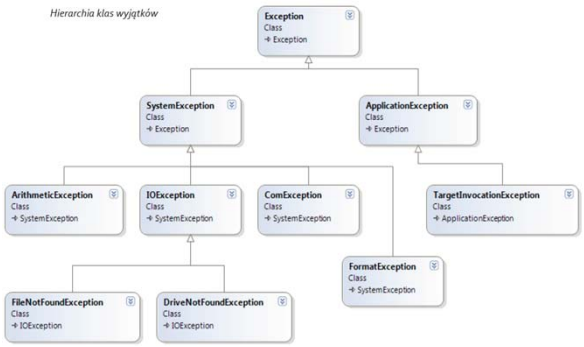
- Wyjątki tworzymy przy pomocy słowa kluczowego **throw** oraz operatora **new**.
- Wszystkie wyjątki, jakie mogą wystąpić w programie, muszą być podklasą klasy **Exception**.

Drzewo hierarchii wyjątków ma dwa główne konary:

- **SystemException** – wyjątki typu *runtime*, związane z systemem operacyjnym, np. brak pamięci, błąd I/O, błąd w przetwarzaniu wyrażeń matematycznych
- **ApplicationException** – wyjątki związane z działaniem aplikacji. Zaleca się, by klasy wyjątków we własnych aplikacjach definiowanych przez programistów dziedziczyły właśnie z tej klasy.

WYJĄTKI

Hierarchia klas wyjątków



Klasa Exception

Klasa **Exception** jest bazową klasą dla wszystkich wyjątków.

Zawiera kluczowe pola pozwalające na przechwycenie pełnej informacji o nieoczekiwanym zdarzeniu, które zmieniło standardowy bieg programu.

Są to:

- **StackTrace** - stos wywołań, ślad miejsca, skąd nastąpiło wyrzucenie wyjątku
- **InnerException** - enkapsulowany wyjątek zawierający szczegóły niższej warstwy
- **Message** – komunikat tekstowy stanowiący faktyczny opis wyjątku. Komunikaty te dla standardowych wyjątków są lokalizowane w platformie .NET
- **Source** – nazwa aplikacji lub obiektu, który wywołał błąd

Klasa Exception

- **HelpLink** – adres URN (ang. *unified resource name*) lub URL (ang. *unified resource locator*) do pliku opisującego dany wyjątek, jego źródło – ogólnie rzecz biorąc związanego z wyjątkiem
- **HResult** – wartość liczbowa – kod błędu związanego z wyjątkiem, ma to związek z koniecznością zachowania spójności z bibliotekami i rozwiązaniami niezarządzanymi
- **TargetSite** – pozwala pobrać nazwę metody, w której wystąpiło wyrzucenie wyjątku

Najczęściej używane typy wyjątków

Poniższe typy wyjątków są powszechnie wykorzystywane w CLR i platformie .NET Framework. Można je zgłaszać lub używać ich jako klas bazowych do definiowania własnych typów wyjątków.

System.ArgumentException
Rodzaj wyjątku zgłaszany, gdy funkcja zostanie wywołana z bezsensownym argumentem. Najczęściej sygnalizuje błąd programu.

System.ArgumentNullException
Podklasa klasy ArgumentNullException używana, gdy argument funkcji (nieoczekiwanie) ma wartość null.

Najczęściej używane typy wyjątków

System.ArgumentOutOfRangeException
Podklasa klasy ArgumentException używana, gdy argument (zazwyczaj liczbowy) jest za duży lub za mały. Wyjątek tego typu może np. zostać zgłoszony, gdy ktoś przekaże ujemną liczbę do funkcji przyjmującej tylko dodatnie wartości.

System.InvalidOperationException
Typ wyjątków zgłaszanych, gdy stan obiektu uniemożliwia metodzie prawidłowe działanie, bez względu na konkretne wartości argumentów. Przykładem może być próba odczytu nieotwartego pliku albo pobieranie następnego elementu z wyliczenia, którego lista została zmieniona w czasie trwania iteracji.

Najczęściej używane typy wyjątków

System.NotSupportedException
Typ wyjątków zgłaszany w celu zasygnalizowania, że określony element funkcjonalności nie jest obsługiwany. Dobrym przykładem jest wywołanie metody Add na kolekcji, dla której IsReadOnly zwraca wartość true.

System.NotImplementedException
Typ wyjątków oznaczający, że funkcja jeszcze nie została zaimplementowana.

System.ObjectDisposedException
Typ wyjątków zgłaszanych, gdy obiekt, na którym wywołano funkcję, został usunięty.

Najczęściej używane typy wyjątków

Innym często używanym typem wyjątków jest **NullReferenceException**.

System CLR zgłasza go, gdy programista chce użyć składowej obiektu o wartości null (co wskazuje na błąd w kodzie programu).

Wyjątek typu NullReferenceException można zgłosić bezpośrednio (w celach testowych) w następujący sposób:

throw null;.

Wyrzucanie wyjątków

Wyrzucanie wyjątków realizowane jest za pomocą słowa kluczowego **throw**:

- **throw new TypWyjątku();**

Taka konstrukcja powoduje przerwanie wykonywania danej metody.

Klasa wyjątku definiuje cztery bazowe konstruktory.

Wyrzucanie wyjątków

```
public class MyException : ApplicationException
{
    public MyException()
    {}
    public MyException(string message) : base(message)
    {}
    public MyException(string message, System.Exception inner) :
    base(message, inner)
    {}
    protected MyException( SerializationInfo info,
            StreamingContext context) : base(info, context)
    {}
}
```


Komentarze wyjątków

Język C# w żaden sposób nie wymusza na programiście konieczności poinformowania użytkownika metody, że może ona spowodować wystąpienie wyjątku.

Dlatego istotną sprawą jest to, by własne metody wyrzucające wyjątki zaopatrzyć w stosowny kod dokumentacyjny.

Komentarze wyjątków

```
/// <summary>
/// Oblicza pole koła.
/// </summary>
/// <param name="promien">promień koła</param>
/// <returns>pole powierzchni</returns>
/// <exception cref="ArgumentException"> wyrzucany dla ujemnego
promienia</exception>
public static double ObliczPoleKola(double promien)
{
     if (promien < 0)
    {
       throw new ArgumentException("Promień nie może być ujemny");
    }
    return Math.PI*promien*promien;
}
```


Nazewnictwo wyjątków

Konwencja nazewnicza wyjątków wygląda następująco:

- nazwa_wlasna + Exception

np. **InvalidArgumentException**, **FileNotFoundException**.

W bloku **catch** podczas przechwytywania wyjątku stosuje się najczęściej krótką nazwę dla obiektu wyjątku poprzez utworzenie skrótu pierwszych liter wyjątku, np.:

- DivideByZeroExcption – dbze
- InvalidArgumentException – iae
- FileNotFoundException – fnfe

Przechwytywanie wyjątków

Mechanizm przechwytywania wyjątków:

- W celu implementacji mechanizmu obsługi wyjątku w metodzie, która nie powinna być przerwana wyjątkiem, umieszczamy blok **try**, który przechwytuje wyjątki.
- Wewnątrz tego bloku podejmowane są próby wykonania konkretnych metod.
- Jeśli zakończą się one niepowodzeniem, zostaje wyrzucony wyjątek, który trafia do bloku **catch**.

Przechwytywanie wyjątków

- W momencie wejścia do bloku **catch** wyjątek uważany jest za obsłużony, a procedura obsługi wyjątku zajmuje się problemem.
- **Należy unikać przechwytywania wyjątków, jeśli nie wiadomo, co z nimi zrobić.**
- **Ważna jest też kolejność….**

Przechwytywanie wyjątków

- **Należy pamiętać o nadrzędnej regule, że wyjątki służą do obsługi sytuacji wyjątkowych**,
- zatem jeżeli np. w formularzu użytkownik powinien wypełnić zestaw wymaganych pól, to przed przetworzeniem ich należy wypełnienie takich pól sprawdzić (walidacja formularza), a nie obsługiwać wyjątki typu **NullPointerException**.
- *Kolejną istotną sprawą jest to, by odróżniać wyjątki od wartości zwracanych z funkcji. Bardzo łatwo wpaść w pułapkę i sterowanie normalnym tokiem działania programu zacząć opierać o wyjątki.*
- *Wyjątki podobnie jak instrukcja **goto** stanowią sposób na przerwanie działania kodu w dowolnym miejscu i bezwarunkową „ucieczkę”.*

```
public class Divisibility {
public static void Main() {
    string line = "2";
    try
    {
        int number = int.Parse(line);
        Console.WriteLine("Liczba " + number);
        if (number%7 != 0) {  Console.WriteLine(" nie"); }
        Console.WriteLine(" jest podzielna przez 7");
    }
    catch (ArgumentNullException ane)
    {
        Console.WriteLine("Nie zainicjalizowany argument"); }
    catch (FormatException fe)
    {
      Console.WriteLine("Ciąg nie jest liczbą"); }
    catch (Exception e)
    {
      Console.WriteLine("Jakiś błąd!"); }
} }
```

Efektem powyższego przykładu będzie:
// Liczba 2 nie jest podzielna przez 7
Jeśli wprowadzić by następującą modyfikację w 3 wierszu:
String line = "napis";
program wyświetli na ekranie następującą informację:
// Ciąg nie jest liczbą

try-cach-finally

- Jako uzupełnienie bloku try-catch można umieścić sekcję finally.
- W bloku **finally** umieszczane są procedury takie jak czyszczenie ekranu, zamykanie plików czy połączeń sieciowych.
- Blok finally dodaje do programu odrobinę przewidywalności: CLR zawsze próbuje go wykonać. Można go wykorzystać do wykonywania czynności porządkowych, np. zamykania połączeń sieciowych.

Blok finally

Blok finally jest zawsze wykonywany, niezależnie od tego, czy wyjątek zostanie zgłoszony i czy blok try zostanie wykonany do końca.
W blokach finally najczęściej wpisuje się kod porządkujący.

Blok finally zostaje wykonany:
- po zakończeniu wykonywania bloku catch;
- po wyjściu sterowania z bloku try z powodu wykonania instrukcji skoku (np. return albo goto);
- po zakończeniu wykonywania bloku try.

Wykonanie bloku finally może zostać uniemożliwione tylko przez pętlę nieskończoną i przez nagłe zakończenie procesu.

Blok finally

W kolejnym przykładzie otwarty plik *zawsze* zostanie zamknięty, bez względu na to, czy:
- blok try zostanie wykonany normalnie;
- wykonywanie zakończy się przedwcześnie z powodu tego, że plik jest pusty (EndOfStream);
- podczas odczytu pliku zostanie zgłoszony wyjątek IOException.

Blok finally

```
static void ReadFile()
{
StreamReader reader = null; // w przestrzeni nazw System.IO
try
{
reader = File.OpenText ("file.txt");
if (reader.EndOfStream) return;
Console.WriteLine (reader.ReadToEnd());
}
finally
{
if (reader != null) reader.Dispose();
}}
```

W tym przykładzie zamknęliśmy plik przez wywołanie metody Dispose obiektu klasy StreamReader. Wywoływanie tej metody na obiekcie w bloku finally jest standardową stosowaną w.NET Framework techniką, która jest bezpośrednio obsługiwana w C# poprzez instrukcję **using**.

Blok finally - Instrukcja using

Wiele klas zawiera niezarządzane zasoby, takie jak: uchwyty do plików, uchwyty graficzne czy połączenia z bazami danych. Klasy te implementują interfejs System.IDisposable, który definiuje jedną bezparametrową metodę o nazwie Dispose służącą do porządkowania zasobów. Instrukcja using zapewnia elegancką składnię do wywoływania metody Dispose na obiektach implementujących interfejs IDisposable w blokach finally:

Poniższy kod:

```
using (StreamReader reader = File.OpenText ("file.txt"))
{
...
}
```

jest równoważny z tym:

```
{
StreamReader reader = File.OpenText ("file.txt");
try
{...}
finally
{
 if (reader != null)
((IDisposable)reader).Dispose();}}
```


Zgłaszanie wyjątków

Wyjątki mogą być zgłaszane przez system wykonawczy lub przez instrukcje znajdujące się w kodzie użytkownika. W poniższym przykładzie metoda Display zgłasza wyjątek System.ArgumentNullException:

```
class Test
{
static void Display (string name)
{
if (name == null)
throw new ArgumentNullException (nameof (name));
Console.WriteLine (name);
}
static void Main()
{
try { Display (null); }
catch (ArgumentNullException ex)
{
Console.WriteLine ("Przechwycono wyjątek.");}}}
```


Ponawianie zgłoszenia wyjątku

Wyjątek można przechwycić i zgłosić jeszcze raz w następujący sposób:

```
try { ... }
catch (Exception ex)
{
// zapisanie błędu w dzienniku
...
throw; // ponowienie zgłoszenia wyjątku
}
```

Gdybyśmy instrukcję throw zamienili na throw ex, to przykład i tak by działał, tylko własność StackTrace nowego wyjątku nie opisywałaby już oryginalnego błędu.

Ponowienie w ten sposób zgłoszenia wyjątku umożliwia jego zarejestrowanie bez **pochłonięcia**.

Ponawianie zgłoszenia wyjątku

Przy okazji zyskujemy możliwość wycofania się z obsługi wyjątku, gdyby sytuacja okazała się inna, niż się spodziewaliśmy:

```
using System.Net;
...
string s = null;
using (WebClient wc = new WebClient())
try { s = wc.DownloadString ("http://www.albahari.com/nutshell/"); }
catch (WebException ex)
{
if (ex.Status == WebExceptionStatus.Timeout)
Console.WriteLine ("Timeout");
else
throw; }
// brak możliwości obsługi innych rodzajów wyjątków
//WebException, więc ponawiamy zgłoszenie
```


Ponawianie zgłoszenia wyjątku

Innym typowym działaniem jest ponowienie zgłoszenia z bardziej specyficznym typem wyjątku. Na przykład:

```
try
{
... // pobranie danych DateTime z XML
}
catch (FormatException ex)
{
throw new XmlException ("Nieprawidłowe dane DateTime.", ex);
}
```

Zwróć uwagę, że przy tworzeniu wyjątku XmlException w drugim argumencie przekazaliśmy oryginalny wyjątek ex. Argument ten przekazuje wartość dla własności InnerException nowego wątku i jest pomocny w diagnostyce. Prawie wszystkie typy wyjątków udostępniają podobny konstruktor. *Mniej* specyficzny typ wyjątku można zgłosić w sytuacji ograniczonego zaufania, aby nie ujawnić szczegółów technicznych.

Filtry wyjątków (C# 6)

W C# 6.0 wprowadzono możliwość używania w klauzuli catch **filtrów wyjątków**, które to filtry definiuje się za pomocą klauzuli when:

```
catch (WebException ex) when (ex.Status == WebExceptionStatus.Timeout)
{
...
}
```

Gdyby w tym przykładzie został zgłoszony wyjątek typu WebException, to nastąpiłoby obliczenie wartości wyrażenia logicznego znajdującego się za słowem kluczowym when.

W przypadku fałszywego wyniku blok catch zostałby zignorowany i nastąpiłoby sprawdzenie następnych klauzul catch.

Filtry wyjątków (C# 6)

Użycie filtrów wyjątków sprawia, że ma sens przechwytywanie tego samego wyjątku kilka razy:

```
catch (WebException ex) when (ex.Status==WebExceptionStatus.Timeout)
{ ... }
catch (WebException ex) when
(ex.Status==WebExceptionStatus.SendFailure)

{ ... }
```

Wyrażenie logiczne w klauzuli when może mieć skutki uboczne, jak np. metoda rejestrująca wyjątekw dzienniku w celach diagnostycznych.

```
class Test
{
static int Calc (int x) => 10 / x;
static void Main()
{
int y = Calc (0);
Console.WriteLine (y);}}

class Test
{
static int Calc (int x) => 10 / x;
static void Main()
{
try
{
int y = Calc (0);
Console.WriteLine (y);
}
catch (DivideByZeroException ex)
{
Console.WriteLine ("x nie może mieć wartości zero.");
}
Console.WriteLine ("Program zakończył działanie z powodzeniem.");
}
}
```

Wynik:
x nie może mieć wartości zero.
Program zakończył działanie z powodzeniem.

Jest to prosty przykład ilustrujący technikę obsługi wyjątków.
W praktyce można by było to lepiej rozwiązać przez sprawdzenie wprost, czy dzielnik nie ma wartości zero, przed wywołaniem Calc.
Błędy, którym można zapobiec, lepiej jest wykrywać zawczasu niż pozostawiać do wykrycia w blokach try-catch, ponieważ obsługa wyjątków pochłania przynajmniej setki cykli zegara.

KOLEKCJE

KOLEKCJE

- Kolekcje (znane w innych językach programowania jako kontenery) są to specjalne klasy obiektów, które służą do przechowywania obiektów.
- Klasy udostępniają mechanizmy do wstawiania, przeglądania oraz pobierania obiektów.

Typy kolekcji platformy można podzielić na następujące kategorie:

- interfejsy definiujące standardowe protokoły kolekcji;
- gotowe do użycia klasy kolekcji (listy, słowniki itd.);
- klasy bazowe do pisania kolekcji specjalnie dostosowanych do potrzeb konkretnych aplikacji.

KOLEKCJE

W .NET kolekcje dostępne są w dwóch podstawowych przestrzeniach:

- **System.Collections**
- **System.Collections.Generic**.

Znajdują się tam odpowiednio kolekcje klas ogólnych i generycznych – czyli kolekcje konkretnych typów.

Klasy generyczne *pozwalają na tworzenie* ***obiektów tej samej klasy dla różnych typów danych****. Dopiero przy tworzeniu obiektu klasy generycznej podajemy typ danych na którym będzie operował.*

KOLEKCJE

Najważniejsze klasy wśród kolekcji ogólnych:

- **ArrayList**,
- **Hashtable**
- oraz abstrakcyjna klasa **CollectionBase**.

Natomiast dla kolekcji generycznych to odpowiednio:

- **List<T>**,
- **Dictionary<TKey, TValue>**
- **Collection<T>**.

Kolekcje generyczne

- Kolekcje generyczne zostały dodane w wersji 2.0 do C# i CLR.
- Z uwagi na bezpieczeństwo typów i uwolnienie od konieczności odpakowywania obiektów (rzutowania z klasy **Object** na konkretny typ) kolekcje generyczne są znacznie bardziej wydajne i preferowane wszędzie tam, gdzie ich użycie jest możliwe.

Kolekcje generyczne

Najważniejsze cechy kolekcji generycznych:

- Pozwalają optymalizować kolekcje pod konkretne zadania
- W swej konstrukcji zapewniają bezpieczeństwo typów (ang. *type safety*)
- Poprawiają wydajność
- Język C# umożliwia definiowanie interfejsów, klas, metod, zdarzeń i delegatów parametryzowanych typem.

Kolekcje generyczne

Najważniejsze cechy kolekcji generycznych:

- Klasy generyczne mogą uzyskać ograniczenia stosowania metod dla konkretnych typów (np. metoda dodająca element do kolekcji wymagała będzie jako argumentu konkretnego typu obiektu)
- Informacja o typie, który jest wykorzystywany w kolekcji, może być uzyskana w czasie działania programu dzięki mechanizmowi refleksji

Implementacja kolekcji w .NET

Ważną cechą kolekcji jest to, że dostarczają podobną i prostą obsługę niezależenie od tego, jaki typ danych jest przetwarzany.

Implementacja kolekcji w .NET

Implementacja kolekcji może odbywać się w trzech obszarach:

- Interfejsy abstrakcyjne, właściwości i operacje kolekcji – np. przejście po wszystkich elementach w oderwaniu od konkretnych implementacji
- Algorytmy metody przetwarzające kolekcje (wyszukiwanie, sortowanie itp.) zwykle zdefiniowane dla pewnego rodzaju kolekcji
- Implementacje klasy będące implementacjami odpowiednich interfejsów

Tablice vs kolekcje

- Do przechowywania obiektów możemy wykorzystywać zarówno kolekcje, jak i tablice.
- Kolekcje posiadają jednak szereg cech, dzięki którym ich zastosowanie jest dużo bardziej elastyczne niż jest to w przypadku tablic.
- Jakich???

Tablice vs kolekcje

- Faktem jest, że tablice poza możliwością przechowywania obiektów (a właściwie referencji) mogą również przechowywać zmienne typów podstawowych.

Tablice vs kolekcje

- Jednak rozmiar tablic jest stały i z góry ustalony przy ich tworzeniu.
- W przypadku kolekcji rozmiar nie jest określony.

- W specyficznych przypadkach użycie tablicy jest zasadne, np. przekazywanie danych pomiędzy kartą graficzną a pamięcią, jednakże w typowych zastosowaniach kolekcje są dużo lepszym rozwiązaniem. Warto pamiętać, że rozmiar kolekcji, mimo że nie jest ustalony, można optymalizować w momencie alokacji – podczas konstrukcji można przekazać zmienną informującą o zakładanym rozmiarze kolekcji (ang. *Initial Capacity*).

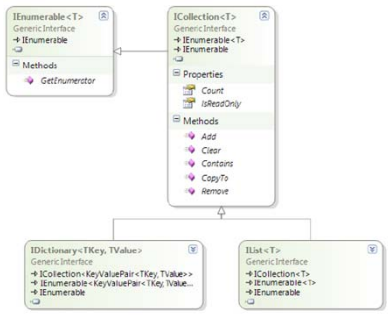
- Ważną cechą kolekcji jest również to, że posiadają wiele metod służących do obsługi przechowywania danych.

NAJWAŻNIEJSZE INTERFEJSY GENERYCZNE

Podstawą kolekcji generycznych w .NET są interfejsy:

- **IEnumerable<T> –** definiuje mechanizm umożliwiający przejrzenie wszystkich elementów zgromadzonych w kolekcji
- **ICollection<T> –** oferuje kluczową funkcjonalność w zakresie dostępu i modyfikacji zbioru obiektów. Interfejs kolekcji jest bazą dla dwóch kolejnych niezwykle istotnych interfejsów:
  - **IDictionary<TKey, TValue> –** definiuje słownik dowolnego typu w oparciu o technikę hashowania
  - **IList<T> –** interfejs listy elementów.

NAJWAŻNIEJSZE INTERFEJSY GENERYCZNE



Klasa List

- Klasa **List** stanowi implementację interfejsu **IList**. Zawiera zestaw kilkunastu metod doskonale ułatwiających operowanie po zbiorze elementów.

- *Klasa ta w zupełności zastępuje tablicę elementów, pozwalając dodatkowo przeszukiwać, sortować i modyfikować kolekcję.*

Klasa List

Najważniejsze metody klasy to:

- **Add** – dodanie elementu
- **AddRange** – dodanie kolekcji
- **Find** – wyszukiwanie elementu
- **GetRange** – dynamiczne pobranie fragmentu listy
- **Sort** – sortowanie
- **ToArray** – dynamiczne utworzenie tablicy zawierającej wszystkie elementy w liście

Klasa List

- Najpierw lista typu **List<T>** musi zostać stworzona, następnie z wykorzystaniem metody **Add()** dodawane są do niej obiekty.
- Do obiektów z takiej listy można się odwołać tak samo jak w tablicach za pomocą indeksu obiektu i operatora indeksowania **[]**.
- Podobnie jak w tablicach występuje tutaj właściwość umożliwiająca odczytanie liczby elementów, co pozwala na stworzenie zabezpieczenia przed odwołaniem się do nieistniejącego obiektu.
- Właściwość ta nosi nazwę **Count**.

Klasa LinkedList

- Klasa **LinkedList** to lista podwójnie łączona implementująca interfejs **ICollection**.
- W przypadku takiej listy każdy z elementów posiada określoną pozycję oraz referencję do następnika i poprzednika.
- Elementy posiadają kolejność zgodną z kolejnością ich dodawania, a ich wartości mogą się powtarzać.

Klasa LinkedList

Zawiera dedykowane metody związane z obsługą list takie jak:

- **AddFirst(e)** – wstawia element na początek listy
- **AddLast(e)** – wstawia element na koniec listy
- **AddAfter(e)** – wstawia element po wybranym elemencie
- **AddBefore(e)** – wstawia element przed wybranym elementem
- **Clear** – czyści kolekcję
- **First** – właściwość pozwalająca pobrać pierwszy element z listy
- **Last** – pobiera ostatni element z listy
- **RemoveFirst()** – usuwa z listy pierwszy element
- **RemoveLast()** – usuwa ostatni element z listy
- **Remove(e)** – jeśli znajdzie element, to usuwa go z listy i zwraca **true**.

Klasa Stack

- Klasa **Stack** podobnie jak **LinkedList** implementuje interfejs **ICollection**.
- Klasa **Stack** reprezentuje stos – strukturę danych, która realizuje kolejkę LIFO (ang. *Last In First Out*).
- Stos zapewnia, że pierwszy element dodany do kolekcji (w przypadku stosu używa się sformułowania *położony na stosie*, ang. *push*) będzie pobrany z niej jako ostatni.

Klasa Stack

Poza standardowymi metodami wynikającymi z implementacji kolekcji stos oferuje trzy kluczowe metody:

- **Push** – wstawia element na „górę” stosu
- **Pop** – pobiera element z góry i usuwa go ze stosu
- **Peek** – pozwala odczytać wartość elementu znajdującego się na górze stosu

Klasa Queue

Klasa **Queue** reprezentuje kolejkę FIFO (ang. *First In First Out*).

- Klasa kolejki implementuje interfejs **ICollection**, zatem można w niej odnaleźć metody pozwalające na iterowanie elementów, odnalezienie rozmiaru bądź konwersję do tablicy.
- Kolejka, podobnie jak stos, zabezpiecza właściwą kolejność w odczycie danych, zawiera przy tym minimalny zestaw trzech operacji.

Klasa Queue

Charakterystyczne metody kolejki to:

- **Enqueue** – dodaje element na koniec kolejki
- **Dequeue** – służy do pobrania z kolejki elementu, który najdłużej się w niej znajduje
- **Peek** – pozwala odczytać wartość elementu znajdującego się na początku kolejki

Klasa HashSet

Klasa **HashSet** jest tablicą z kodowaniem mieszanym (potocznie zwaną haszmapą).

- Elementy w **HashSet** nie mogą się powtarzać oraz nie posiadają określonej pozycji.
- Porządek takich elementów nie jest również uzależniony od kolejności ich dodawania.
- Specjalna funkcja nadaje każdemu elementowi unikalny indeks.
- Powoduje to, że operacje zajmują zwykle stały czas, nawet przy dużych tablicach.

Klasa Dictionary

- Klasa **Dictionary** jest tablicą typu asocjacyjnego.
- Przetwarzanie obiektów odbywa się za pomocą pary klucz-wartość.
- Klucze w tej kolekcji muszą być unikalne, wartości natomiast już dowolne.
- Znając konkretny klucz, istnieje możliwość znalezienia powiązanej z nim wartości.

Klasa Dictionary

Dodawanie elementów do kolekcji odbywa się poprzez metodę:

- **Add(klucz, wartość);**

Wprowadzona wartość jest kojarzona z podanym kluczem i według tego klucza można uzyskać dostęp do elementu poprzez operator indeksowania tablic:

- [klucz];

Interfejs IEnumerator

Interfejs IEnumerator

Każda kolekcja produkuje swój własny obiekt o interfejsie **IEnumerator**.

Interfejs **IEnumerator** zawiera metody takie jak:

- **bool MoveNext()** – przechodzi do następnego elementu w kolekcji, zwraca **true**, jeśli takowy istnieje
- **<T> Current** – właściwość umożliwiająca pobranie bieżącego elementu.

Podstawowe operacje na kolekcjach

W ramach interfejsu **ICollection** można wyróżnić jeszcze podstawowe metody takie jak:

- **int Count** – zwraca liczbę elementów
- **bool Contains (T item)** – sprawdza, czy w kolekcji znajduje się dany element
- **void Add(T item)** – dodaje element do kolekcji
- **bool Remove(T item)** – usuwa obiekt z kolekcji, zwraca **true**, jeśli operacja się powiedzie.

Operacje grupowe

Operacje grupowe

- Oprócz opisanych wcześniej operacji w interfejsie **ICollection** zostały zaimplementowane operacje grupowe takie jak:
- **void Clear()** – usuwa wszystkie elementy kolekcji
- **void CopyTo(T array)** –kopiuje zawartość do tablicy

Interfejs **IList** dodatkowo dodaje kilka ciekawych metod grupowych:

- **bool AddRange(IEnumerable c)** – dodaje wszystkie elementy ze zbioru **c**
- **void RemoveAt(int index)** – usuwa element o podanym indeksie
- **IEnumerable<T> Intereset(IEnumerable c)** – zostawia tylko te elementy, które są jednocześnie zawarte w zbiorze **c**
- **T[]ToArray()** – zwraca tablice obiektów zawartych w kolekcji
- **List<T> FindAll(Predicate<T> predicate)** – zwraca listę elementów pasujących do wzorca

Polimorfizm w kolekcjach

Dzięki zastosowaniu polimorfizmu parametrycznego typ obiektu nie musi być ustalany w trakcie kompilacji – jest to realizowane automatycznie.

Mechanizm ten powala więc tworzyć abstrakcyjne programy mające w parametrze nieokreśloną z góry kolekcję.

W zależności od tego, jaką kolekcję otrzymają, będą działać inaczej.

Test

```
using System;
using System.Collections;
using System.Diagnostics;
namespace ConsoleApplication_Kolekcje
{
   class Program
   {
   static void Main(string[] args)
      {
         ArrayList MojaLista = new ArrayList();

         MojaLista.Add("Krzysiek");
         MojaLista.Add("Kot");
         MojaLista.Add(123);
         MojaLista.Add(3.14);

         for (int i = 0; i < MojaLista.Count; i++)
         {
            Debug.WriteLine("Indeks: {0} wartość: {1}", i, MojaLista[i]);
         }
}}}
```

```
Indeks: 0 wartość: Krzysiek
Indeks: 1 wartość: Kot
Indeks: 2 wartość: 123
Indeks: 3 wartość: 3.14
```

Przy pomocy Add() dodajemy do listy kolejne elementy, raz typu string, później liczbę stałoprzecinkową i wreszcie ? liczbę rzeczywistą.

Jest to absolutnie dopuszczalne, gdyż parametr metody Add() jest typu object, a jak wiadomo wszystkie typy .NET Framework dziedziczą po tym typie (konkretnie jest to alias do klasy System.Object).

Test

```
using System;
using System.Collections;
using System.Diagnostics;
namespace ConsoleApplication_Kolekcje
{
   class Program
   {
   static void Main(string[] args)
      {
         ArrayList MojaLista = new ArrayList();

         MojaLista.Add(1);
         MojaLista.Add(2);
         MojaLista.Add(3);
         MojaLista.Add(4);

         int liczba = MojaLista[3];

         Debug.WriteLine(liczba);
}}}
```

```
1>------ Build started: Project: ConsoleApplication_Kolekcje, Configuration: Debug Any CPU ------
1>c:\users\smolka\documents\visual studio 2015\Projects\ConsoleApplication_Kolekcje\ConsoleApplication_Kolekcje\Program.cs(31,26,31,39): error CS0266: Cannot implicitly convert type 'object' to 'int'. An explicit conversion exists (are you missing a cast?)
========== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped ==========
```

Kolekcja działa na typie System.Object, a jak wiemy Object to ostateczna klasa w C#, z której dziedziczy wszystko.

Przez takie właśnie działanie nasze dane, które do nich przesyłamy muszą przejść tzw. proces opakowania (ang. Boxing).

W przykładzie pracujemy na kolekcji ArrayList, gdzie dodajemy elementy typu *int*.

Kiedy wkładamy je do środka naszej kolekcji ArrayList opakowane są one w typ Object, jednak jeśli teraz napiszemy naszą instrukcję to otrzymamy błąd kompilacji.

Musimy jawnie wykonać proces wypakowania (ang. Unboxing) do zmiennej typu int.

Test

```
using System;
using System.Collections;
using System.Diagnostics;
namespace ConsoleApplication_Kolekcje
{
   class Program
   {
   static void Main(string[] args)
      {
         ArrayList MojaLista = new ArrayList();

         MojaLista.Add(1);
         MojaLista.Add(2);
         MojaLista.Add(3);
         MojaLista.Add(4);

         int liczba=(int)MojaLista[3];

         Debug.WriteLine(liczba);
}}}
```

```
4
```

Przy każdym wywołaniu metody **Add()** program musi wykonać opakowywanie (ang. *boxing*) typów, a przy wyświetlaniu - odpakowywanie. Przy dużej ilości danych trwa to dość długo, na tyle długo iż opłacalne jest wykorzystanie w tym momencie typów generycznych.

Dlatego jeżeli zamierzasz umieścić w kolekcji dane tego samego typu, lepiej wykorzystać klasy oferowane przez przestrzeń nazw **System.Collection.Generic**. Znajdują się tam klasy, odpowiedniki klas **Stack** oraz **Queue**, które działają szybciej na danych tego samego typu.

Nie ma w przestrzeni nazw **System.Collection.Generic** klasy **ArrayList**. Zamiast tego możemy jednak wykorzystać klasę **List**, która jest właściwie generycznym odpowiednikiem **ArrayList**.

Test

```
using System;
using System.Collections;
using System.Diagnostics;
namespace ConsoleApplication_Kolekcje
{
   class Program
   {
   static void Main(string[] args)
      {
         ArrayList MojaLista = new ArrayList();
         for (int i = 1; i <= 5; i++)
         {
            MojaLista.Add(i);
         }

         MojaLista.Add("Kot");

         foreach (var element in MojaLista)
         {
            Debug.WriteLine(element);
         }
}}}
```

```
1
2
3
4
5
Kot
```

Drugi problem ze standardowymi listami to **bezpieczeństwo typów**.
Skoro metoda **Add()** w naszej kolekcji **ArrayList** przyjmuje, jako parametr typ **Object** to w zasadzie możemy tam wpakować wszystko.
To są właśnie powody, dla których powstały typy generyczne zapewniające pełne bezpieczeństwo typów oraz poprawienie wydajności działania programów jak się zresztą za chwilę przekonamy.
Problem bezpieczeństwa typów można jednak rozwiązać nie używając generyczności – tworząc np. własną klasę, która implementuje niegeneryczny interfejs IEnumerable, który udostępnia nam metodę GetEnumerator(), dzięki której możemy na naszym obiekcie typu zastosować później pętlę foreach. Pomimo faktu, iż zapewnimy w naszej kolekcji bezpieczeństwo typów to nie unikniemy operacji pakowania i rozpakowania.

Metody generyczne

```
using System;
using System.Collections;
using System.Diagnostics;
namespace ConsoleApplication_Kolekcje
{
   class Program
   {
      class MojaKlasaGeneric<T>
      {
         public void Add(T X)
         {
            Console.WriteLine("{0}", X);
         }
      }
   static void Main(string[] args)
      {
       MojaKlasaGeneric<string> MojaKolekcja
                        = new MojaKlasaGeneric<string>();
       MojaKolekcja.Add("Hello World!");
}}}
```

Istnieje możliwość tworzenia własnych klas generycznych. Jedyne, co musimy zrobić, to na końcu nazwy dodać frazę <T>.

Przy tworzeniu egzemplarza klasy kompilator będzie wymagał, aby podać również typ danych, na których ma ona operować.

Przyjęło się, że przy deklarowaniu typów generycznych stosujemy frazę <T>. Kompilator nie wymusza jednak takiego nazewnictwa, więc równie dobrze możemy napisać: class Generic<Type> {}.

Z przyzwyczajenia programiści nazywają go według następującej konwencji:
T – typ
K – klucz
V – wartość

```
Hello World!
```


Tworzenie typów generycznych

```
using System;
using System.Collections;
using System.Diagnostics;
namespace ConsoleApplication_Kolekcje
{
   class Program
   {
    static void MetodaGeneryczna<T>(T Element)
     {
        Debug.WriteLine("{0}", Element);
     }

    static void Main(string[] args)
     {
        MetodaGeneryczna("Krzysiek");
        MetodaGeneryczna(5);
        MetodaGeneryczna<int>(12);
}}}
```

Istnieje również możliwość deklarowania metod generycznych.

Ich tworzenie wygląda bardzo podobnie.

Wywołując taką metodę, możesz, aczkolwiek nie musisz, podawać typu danych.

Kompilator domyśli się typu na podstawie przekazanych parametrów.

```
Krzysiek
5
12
```


Korzystanie z list

```
using System;
using System.Collections;
using System.Diagnostics;
using System.Collections.Generic;
namespace ConsoleApplication_Kolekcje
{
   class Program
   {
    static void Main(string[] args)
     {
        List<int> lista = new List<int>();
        for (int i = 0; i < 5; i++)
        {
           lista.Add(i);
        }
        lista.Insert(7, 123);
        for (int i = 0; i < lista.Count; i++)
        {
           Debug.WriteLine(lista[i] + " ");
        }
}}}
```

```
System.ArgumentOutOfRangeException was unhandled
  HResult=-2146233086
  Message=Indeks i długość muszą mieścić się w granicach elementu Lista.
Nazwa parametru: index
  ParamName=index
  Source=mscorlib
(...)
```


Korzystanie z list

```
using System;
using System.Collections;
using System.Diagnostics;
using System.Collections.Generic;
namespace ConsoleApplication_Kolekcje
{
   class Program
   {
    static void Main(string[] args)
     {
        List<int> lista = new List<int>();
        for (int i = 0; i < 5; i++)
        {
           lista.Add(i);
        }
        lista.Insert(2, 123);
        for (int i = 0; i < lista.Count; i++)
        {
           Debug.WriteLine(lista[i] + " ");
        }
}}}
```

```
0
1
123
2
3
4
```


Korzystanie z list

```
using System;
using System.Collections;
using System.Diagnostics;
using System.Collections.Generic;
namespace ConsoleApplication_Kolekcje
{
 static void MetodaGeneryczna<T1,T2>(T1 Element1, T2 Element2)
     {
        Debug.WriteLine("{0} i {1}", Element1, Element2);
     }
class Program
   {
    static void Main(string[] args)
     {
        MetodaGeneryczna(123,"Krzysiek");

}}}
```

```
123 i Krzysiek
```

```
MetodaGeneryczna1(123,"Krzysiek");
```

```
void Program.MetodaGeneryczna1<int, string>(int Element1, string Element2)
```


Używanie klasy List<T>

```
using System;
using System.Collections.Generic;
class Program
{
    static void Main()
    {
    List<string> list = new List<string>();
    // Listy są automatycznie wydłużane w reakcji
    // na dodawanie elementów.
    list.Add("Apsik");
    list.Add("Wesołek");
    list.Add("Gapcio");
    list.Add("Mędrek");
    list.Add("Śpioszek");
    list.Add("Nieśmiałek");
    list.Add("Gburek");
    list.Sort();
    Console.WriteLine( $"W porządku alfabetycznym pierwszy krasnal to { list[0] }, a „
          + $"ostatnim jest { list[6] }.");
    list.Remove("Gburek");
}}
```

```
W porządku alfabetycznym pierwszy krasnal to Apsik, a ostatnim jest Wesołek.
```

Używanie dopełnienia bitowego do wyniku zwróconego przez metodę BinarySearch()

```
using System;
using System.Collections.Generic;
class Program
{
    static void Main()
    {
    List<string> list = new List<string>();
    int search;
    list.Add("public");
    list.Add("protected");
    list.Add("private");
    list.Sort();
    search = list.BinarySearch("protected internal");
    if (search < 0)
    {
        list.Insert(~search, "protected internal");
    }
    foreach (string accessModifier in list)
    {
        Console.WriteLine(accessModifier);
    }
}}
```

```
private
protected
protected internal
public
```

Aby znaleźć na liście List<T> konkretny element, można się posłużyć metodami Contains(), IndexOf(), LastIndexOf() i BinarySearch().
Pierwsze trzy z tych metod przeszukują kolekcję, począwszy od pierwszego elementu (lub ostatniego w przypadku metody LastIndexOf()), i sprawdzają każdą wartość do momentu znalezienia tej szukanej. Czas działania tych metod jest proporcjonalny do liczby elementów, które trzeba sprawdzić do czasu natrafienia na szukaną wartość.
Zauważ, że klasy kolekcji nie wymagają, by elementy były unikatowe.
Jeśli dwa elementy kolekcji (lub większa ich liczba) są identyczne, metoda IndexOf() zwraca indeks pierwszego ze znalezionych elementów, a metoda LastIndexOf() — ostatniej znalezionej wartości.
Metoda BinarySearch() posługuje się znacznie szybszym algorytmem wyszukiwania binarnego, ale wymaga, by elementy były posortowane. Przydatną cechą metody BinarySearch() jest to, że jeśli element nie zostanie znaleziony, metoda zwraca ujemną liczbę całkowitą.
Dopełnienie bitowe (~) tej liczby to indeks pierwszego elementu większego od szukanego lub, jeśli żadna wartość nie jest większa od szukanej, łączna liczba elementów. To zapewnia wygodny mechanizm wstawiania nowych wartości na listę w konkretnym miejscu, aby zachować porządek sortowania.

Wyszukiwanie wielu elementów za pomocą metody FindAll()

```
using System;
using System.Collections.Generic;
class Program
{
static void Main()
{
        List<int> list = new List<int>();
        list.Add(1);
        list.Add(2);
        list.Add(3);
        list.Add(2);
        List<int> results = list.FindAll(Even);
        foreach (int number in results)
        {
            Console.WriteLine(number);
        }
}
public static bool Even(int value) => (value % 2) == 0;
}
```

Czasem trzeba znaleźć wiele elementów na liście, a kryteria są bardziej skomplikowane niż sprawdzanie konkretnej wartości.
Na potrzeby tej sytuacji w klasie System.Collections.Generic.List<T> udostępniono metodę FindAll(). Przyjmuje ona parametr typu Predicate<T>, który określa delegat (referencję do metody).

```
2
2
```


Wstawianie elementów do kolekcji typu Dictionary<TKey, TValue>

```
using System;
using System.Collections.Generic;
class Program
{
    static void Main()
    {
// Kod dla wersji C# 6.0 (w starszych wersjach zastosuj składnię {"Error", ConsoleColor.Red}).
    var colorMap = new Dictionary<string, ConsoleColor>
    {
        ["Error"] = ConsoleColor.Red,
        ["Warning"] = ConsoleColor.Yellow,
        ["Information"] = ConsoleColor.Green
    };
    colorMap["Verbose"] = ConsoleColor.White;
    colorMap["Error"] = ConsoleColor.Cyan;
// …
}
}
```

W pierwszym przypisaniu z danym kluczem nie jest powiązana żadna wartość ze słownika. Wtedy klasa słownika wstawia nową wartość o podanym kluczu.
W drugim przypisaniu element o podanym kluczu już istnieje. Wtedy kod nie wstawia dodatkowego elementu, ale zastępuje wcześniejszą wartość typu ConsoleColor powiązaną z kluczem "Error" nową wartością — ConsoleColor.Cyan.

Wstawianie elementów do kolekcji typu Dictionary<TKey, TValue>

```
using System;
using System.Collections.Generic;
class Program
{
static void Main()
{
   // Kod dla wersji C# 6.0 (we wcześniejszych wersjach zastosuj składnię {"Error", ConsoleColor.Red}).
   Dictionary<string, ConsoleColor> colorMap =
   new Dictionary<string, ConsoleColor>
   {
        ["Error"] = ConsoleColor.Red,
        ["Warning"] = ConsoleColor.Yellow,
        ["Information"] = ConsoleColor.Green,
        ["Verbose"] = ConsoleColor.White
   };
   Print(colorMap);
}
private static void Print(
   IEnumerable<KeyValuePair<string, ConsoleColor>> items)
   {
        foreach (KeyValuePair<string, ConsoleColor> item in items)
        {
            Console.ForegroundColor = item.Value;
            Console.WriteLine(item.Key);
        }}}
```

```
Error
Warning
Information
Verbose
```


DELEGATY

DELEGAT TO OBIEKT „WIEDZĄCY", JAK WYWOŁAĆ METODĘ.

Delegaty

- W nowoczesnych interfejsach graficznych współpraca z użytkownikiem opiera się o model sterowany zdarzeniami.
- Współczesne programy prezentują dane i oczekują na działanie użytkownika.
- Rodzaje wykonywanych zdarzeń mogą być rozmaite:
  - naciśnięcie przycisku,
  - wybór pola listy,
  - zmiana zawartości pola tekstowego,
  - załadowanie okienka formularza,
  - odliczanie wewnętrznego timera.

Delegaty

- Zdarzenia i delegaty są ściśle ze sobą powiązane – przyjęta na platformie .NET konwencja pozwala z wybranym zdarzeniem aplikacji powiązać zdefiniowaną metodę obsługi.
- Mechanizmem umożliwiającym wiązanie metod obsługi zdarzeń do samych zdarzeń są najczęściej **delegaty**.

Deklaracja delegatów

- Delegaty z formalnego punku widzenia są prostym typem referencyjnym, który umożliwia zdefiniowanie sygnatury pojedynczej metody.
- Delegat może wskazywać/używać dowolnej metody o zgodnej sygnaturze.
- Można tutaj odnaleźć analogię do wskaźnika do funkcji – czyli mechanizmu umożliwiającego zdefiniowanie dowolnej funkcji o zadanej sygnaturze, tak by później można było się do niej elastycznie odwołać.

Deklaracja delegatów

Deklarowanie delegata wygląda następująco:

[kwal_dostępu] delegate TypZwracany NazwaDelegata(Parametry pars);

- opcjonalny kwalifikator dostępu,
- słowo kluczowe języka delegate,
- typ zwracany,
- definiowana przez programistę nazwa delegata,
- lista parametrów.

- Delegat może być zdefiniowany zarówno wewnątrz, jak i na zewnątrz kodu klasy.

Inicjalizacja i wywołanie delegatów

- Inicjalizacja delegata polega na przypisaniu do zmiennej typu określonego nazwą delegata, konkretnej funkcji o pasującej sygnaturze.
- Zatem dla zdefiniowanego w poprzednim punkcie delegata o nazwie **NazwaDelegata** inicjalizacja pełna wygląda następująco:

```
NazwaDelegata zmienna = new NazwaDelegata(metoda);
```

- W języku C# dozwolona jest także krótsza forma inicjalizacji, która polega na przypisaniu do zmiennej bezpośrednio nazwy metody:

```
NazwaDelegata zmienna = metoda;
```


Inicjalizacja i wywołanie delegatów

- Gdy mamy utworzoną zmienną typu delegata, możemy ją wywołać w sposób zarówno synchroniczny, jak i asynchroniczny.
- Wywołanie synchroniczne metody delegata może się odbyć na dwa sposoby.
- Pierwszym z nich jest użycie metody **Invoke**, dostępnej dla każdego delegata. Wewnątrz metody **Invoke** należy podać parametry wynikające z deklaracji delegata:

```
zmienna.Invoke("parametr1", 144);
```

- Alternatywnym sposobem wywołania delegata jest użycie składni identycznej jak w przypadku wywoływania metod:

```
zmienna("parametr1", 144);
```


Anonimowa metoda delegata

- Podczas inicjalizacji zmiennej delegata należy powiązać ze zmienną metodę.
- Język C# umożliwia powiązanie do zmiennej typu delegowanego zarówno jawnie zdefiniowanej metody, jak również tzw. **metody anonimowej**.

Anonimowa metoda delegata

- **Metoda anonimowa** to wewnętrzny blok kodu, który nie posiada zdefiniowanej nazwy w programie.
- Do takiego bloku można swobodnie przekazać parametry, jednakże nie można go w sposób jawny wywołać.
- Metody anonimowe najczęściej występują w parze z delegatami – jedynym sposobem wywołania takiej metody pozostaje wywołanie samego delegata.

- *Warto pamiętać, że dłuższe bloki metod anonimowych mogą utrudnić analizowanie i testowanie kodu – wkrótce zaprezentowany zostanie mechanizm zdarzeń, który zabezpiecza bezpośrednie uruchomienie metody delegowanej w dużo bardziej elegancki sposób.*

Anonimowa metoda delegata

- Przeanalizujemy definiowanie delegata anonimowego dla typu zdefiniowanego następująco:

delegate void TestowyDelegat(string wiadomosc);


```
delegate void TestowyDelegat(string s);
public static void Main()
{
        // inicjalizacja delegata z jawnie zdefiniowaną metodą
        TestowyDelegat d1 = PrzykladowaMetodaDelegata;
        // inicjalizacja delegata metodą anonimową
        TestowyDelegat del1 = delegate(string informacja)
        {
                Console.WriteLine("Anonimowo: " + informacja);
        };
        // wywołanie delegata d1
        d1("HELLO");
        // wywołanie delegata d2
        del1("WORLD");
}

static void PrzykladowaMetodaDelegata(string k)
        {
            System.Console.WriteLine(k);
        }
```


Operator lambda

- MS .NET Framework 3.0 uzupełnił język C# między innymi o **operator lambda**.
- Operator lambda to sposób na bardziej zwięzłe definiowanie metod anonimowych.
- Przedstawiony wcześniej przykład można zapisać w następującej postaci:

```
TestowyDelegat del2 =
informacja => Console.WriteLine("Anonimowo: " + informacja);
```


Operator lambda

Składnia **operatora lambda** wygląda następująco:

- nazwa_parametru – jeśli jest ich więcej, to muszą zostać ujęte w nawiasy.
- Typ zmiennej jest określony definicją delegata, nazwa może być dowolna i obowiązuje dalej w bloku lambda,
- operator lambda =>,
- blok kodu związany z obsługą delegata.

Używanie delegatów parametryzowanych

- Delegaty mogą podlegać parametryzacji typem.

- Taka elastyczność umożliwia na zdefiniowanie odwołań do metod obsługi, które w momencie inicjalizacji zostaną uzupełnione informacją o typie danych.

- *Przykładem takiego rozwiązania może być np. sortowanie kolekcji. Z jednej strony można sobie wyobrazić rozmaite implementacje metody sortującej kolekcje.*
- *Drugim wymiarem jest fakt, że kolekcja podlega parametryzacji typem – zatem metoda sortująca może być również parametryzowana.*

Używanie delegatów parametryzowanych

```
// Przykład parametryzowania metody
public delegate void SortujListe<T>(List<T> list);
public static void Main()
{
        List<int> liczby = new List<int>();
        SortujListe<int> sortowanie1 = SortujListeQuickSortem;
        SortujListe<int> sortowanie2 = SortujBombelkowo;
}
private static void SortujBombelkowo(List<int> list)
{
        // ...
}
private static void SortujListeQuickSortem(List<int> list)
{
        // ...
}
```


Delegaty zbiorowe

- Wszystkie egzemplarze delegatów mają zdolność **multiemisji** (ang. *multicast*). Oznacza to, że egzemplarz delegatu może się odnosić nie tylko do jednej metody, ale do całej listy metod.
- Dzięki temu mechanizm delegatów pozwala na wywołanie rozmaitych metod w ramach jednego wywołania delegata.
- W tym celu w języku C# zostały przeciążone operatory arytmetyczne **+**, **+=**.
- Takie rozwiązanie pozwala na wysłanie komunikatu do wielu odbiorców.
- Jest to przejrzysty i efektywny mechanizm.
- Delegaty posiadają - przechowywaną wewnętrznie - informację o tym, jakie metody obsługi zostały z nimi powiązane.
- Programista nie musi się kłopotać o wywołanie tych metod, wystarczy, że wywoła delegata.
- Delegaty są **niezmienne**, więc użycie operatorów += i -= w rzeczywistości oznacza utworzenie *nowego* egzemplarza delegatu i przypisanie go do istniejącej zmiennej.


```
// Zastosowanie delegatów zbiorowych
delegate void Del(string s);
class TestClass
{
   static void Hello(string s)
   {
      System.Console.WriteLine("  Hello, {0}!", s);   }
   static void Goodbye(string s)
   {
      System.Console.WriteLine("  Goodbye, {0}!", s);   }

   static void Main()
   {
      Del a, b, c, d;

      // Create the delegate object a that references the method Hello:
      a = Hello;

      // Create the delegate object b that references the method Goodbye:
      b = Goodbye;
```



```
// Zastosowanie delegatów zbiorowych

      // The two delegates, a and b, are composed to form c:
      c = a + b;

      // Remove a from the composed delegate, leaving d,
      // which calls only the method Goodbye:
      d = c - a;

      System.Console.WriteLine("Invoking delegate a:");
      a("A");
      System.Console.WriteLine("Invoking delegate b:");
      b("B");
      System.Console.WriteLine("Invoking delegate c:");
      c("C");
      System.Console.WriteLine("Invoking delegate d:");
      d("D");
   }
}
```

```
Invoking delegate a:
  Hello, A!
Invoking delegate b:
  Goodbye, B!
Invoking delegate c:
  Hello, C!
  Goodbye, C!
Invoking delegate d:
  Goodbye, D!
```


Zdarzenia

- Aplikacje z interfejsem GUI opierają swe działanie…
- o zdarzenia.

- W języku C# obiekt może publikować zestaw zdarzeń, które inne klasy mogą subskrybować.
- Jest to elegancki, wbudowany mechanizm wzorca Observer.
- Bezpośredni dostęp do zmiennej zdefiniowanej przez delegata umożliwia jej modyfikację z zewnątrz.
- Zatem jeżeli mamy dodaną kolekcję powiązanych metod obsługi delegata, to jedną nieopatrzną instrukcją można odłączyć całą kolekcję.

Zdarzenia

- Mechanizm zdarzeń zabezpiecza przed taką sytuacją – dzięki czemu mamy pewność, że uruchomienie delegata jest możliwe tylko wewnątrz klasy, w której się on znajduje.

Zdarzenia w języku C# obsługiwane są przez Delegaty. Popularne przykłady to:

- Button.Click;
- TextBox.TextChanged;
- Timer.Tick …

Inicjalizacja i wywołanie delegatów

- Zdarzenia bardzo często dotyczą graficznych kontrolek, jednakże jeśli chodzi o sam język C#, słowo kluczowe **event** - związane z obsługą zdarzeń – nie jest bezpośrednio związane z interfejsem graficznym.
- Zdarzeniem mogą być otrzymane dane w gnieździe sieciowym, bądź porcie szeregowym, bądź dowolna sytuacja w programie, której wystąpienie nie jest określone w sposób sekwencyjny.

Inicjalizacja i wywołanie delegatów

- Definiowanie zdarzenia posiada następującą składnię:
- Kwalifikator_dostępu event nazwa_typu_delegata NazwaZdarzenia
- Jak widać w powyższej definicji – definiując zdarzenie, należy odwołać się do delegata.
- Delegat może być zdefiniowany przez nas, bądź można wykorzystać gotowe delegaty.
- Poniżej przykład zawierający definicję zdarzenia oraz wywołanie zdarzenia (poinformowanie subskrybentów).

Inicjalizacja i wywołanie delegatów

```
// Definicja oraz wywołanie zdarzenia
public class Winda
{
public delegate void ZmianaPiętra(int numerPietra);
public event ZmianaPiętra OnZmianaPiętra;
private int biezacePietro = 0;
public void JedzWGore(int numerPietra)
{
    for (int i = biezacePietro; i < numerPietra; i++)
    {
        // obsługa windy
        // publikacja zdarzenia
        this.OnZmianaPiętra(i);
        }
    }
}
```


Inicjalizacja i wywołanie delegatów

```
// Zastosowanie klasy obserwującej zdarzenie
public class ObserwatorWindy
{
public static void Main()
{
    Winda winda = new Winda();
    winda.OnZmianaPiętra += ObserwujWinde;
    winda.JedzWGore(7);
}
private static void ObserwujWinde(int numerPietra)
{
        Console.WriteLine("Winda jest na {0} piętrze", numerPietra);
        Console.ReadKey();
}
}
```


Odłączanie nasłuchu zdarzeń i delegatów

Wracając do subskrypcji i publikacji:

- Zdarzenia są definiowane w klasie publikującej.
- Obiekty klasy publikującej udostępniają listę zdarzeń, którymi mogą być zainteresowane inne klasy.
- Wywołanie delegatów odbywa się na rzecz klasy subskrybującej.

Odłączanie nasłuchu zdarzeń i delegatów

- Aby dołączyć nowy obiekt do obsługi zdarzenia, należy posłużyć się operatorem **+=**.
- Odłączenie obsługi zdarzenia to odpowiednio operator **-=**.
- *Zdarzenia w odróżnieniu od delegatów nie mają zdefiniowanego operatora przypisania (=), w tym przypadku możliwa jest wyłącznie subskrypcja i rezygnacja z niej.*

```
// Odłączanie nasłuchu
public delegate void GranieMuzyki();
public class OdlaczanieDelegatow
{
public static void Main()
{
      GranieMuzyki granieMuzyki = GranieNaTrąbce;
      granieMuzyki += GranieNaGitarze;
      granieMuzyki -= GranieNaGitarze;
      granieMuzyki();
      Console.ReadKey();
}
public static void GranieNaTrąbce()
{
      Console.WriteLine("granie na trąbce");
}
public static void GranieNaGitarze()
{
      Console.WriteLine("granie na gitarze");
} }
```

```
granie na trąbce
```


Asynchroniczne wywoływanie zdarzeń

- Jedną z najbardziej interesujących cech obsługi zdarzeń to fakt, że dowolny delegat może zostać wywołany w sposób asynchroniczny.
- Oznacza to, że klasa publikująca tylko wysyła komunikat do subskrybentów i nie czeka na przetworzenie przez nich zdarzenia.
- Takie działanie zabezpiecza płynną pracę klasy publikującej zdarzenia. Jest tutaj wewnętrznie wykorzystany mechanizm wielowątkowości.
- Do wywołania zdarzenia w sposób asynchroniczny służy metoda **BeginInvoke**.


```
// Przykład wywołania asynchronicznego
public class Winda
{
public delegate void ZmianaPiętra(int numerPietra);
public event ZmianaPiętra OnZmianaPiętra;
private int biezacePietro = 0;
private AsyncCallback callback = ZmianaPietraEndInvoke;
public void JedzWGore(int numerPietra)
{
      for (int i = biezacePietro; i < numerPietra; i++)
      {
            // obsługa windy // publikacja zdarzenia
            this.OnZmianaPiętra.BeginInvoke(i, callback, this);
      }
}
private static void ZmianaPietraEndInvoke(IAsyncResult ar)
{
Console.WriteLine(
      "Obserwatorzy przetworzyli zdarzenie asynchroniczne");
} }
```


Inicjalizacja i wywołanie delegatów

Jeśli chodzi o wyniki zwrotne, to są dwie możliwe sytuacje:

- Klasa publikująca wyłącznie informuje subskrybentów i nie jest zainteresowana wynikami ich działania.
- Klasa publikująca potrzebuje w swym działaniu otrzymać/przetworzyć wyniki asynchronicznie wywołanych metod – w tym celu musi się ona posłużyć wywołaniem zwrotnym tzw. callback'iem.

Delegaty a interfejsy

Każdy problem dający się rozwiązać za pomocą delegatu można też rozwiązać przy użyciu interfejsu.

```
public delegate int Transformer (int x);


class Util
{
    public static void Transform (int[] values, Transformer t)
    {
    for (int i = 0; i < values.Length; i++)
                values[i] = t (values[i]);
    }
}

class Test
{
    static void Main()
    {
    int[] values = { 1, 2, 3 };
    Util.Transform (values, Square); // podczepienie do metody Square
    foreach (int i in values)
          Console.Write (i + " "); // 1 4 9
    }
    static int Square (int x) => x * x;

}
```

```
public interface ITransformer
{
    int Transform (int x);
}
public class Util
{
    public static void TransformAll (int[] values, ITransformer t)
    {
    for (int i = 0; i < values.Length; i++)
                values[i] = t.Transform (values[i]);
    }
}
class Squarer : ITransformer
{
    public int Transform (int x) => x * x;
}
...
static void Main()
{
    int[] values = { 1, 2, 3 };
    Util.TransformAll (values, new Squarer());
    foreach (int i in values)
          Console.WriteLine (i);
}
```


Delegaty a interfejsy

Każdy problem dający się rozwiązać za pomocą delegatu można też rozwiązać przy użyciu interfejsu.

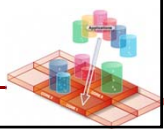
Delegat może być lepszym rozwiązaniem od interfejsu, jeśli spełniony jest przynajmniej jeden z poniższych warunków:

- ☐ interfejs zawiera definicję tylko jednej metody;
- ☐ potrzebna jest możliwość skorzystania z multiemisji;
- ☐ subskrybent musi zaimplementować interfejs wiele razy.

ASYNCHRONICZNOŚĆ WBUDOWANA W JĘZYK

Wielozadaniowość

- Zastosowanie wielowątkowości ma na celu zwiększenie wydajności i skrócenia czasu reakcji dla wszystkich typów aplikacji.
- Wątki (threads) – wydzielone jednostki zadaniowe tworzone poprzez proces macierzysty w celu wykonania określonych zadań.
- Pozwalają wykonywać pewne długotrwałe operacje w tle, podczas gdy operacje pracujące w pierwszym planie pozostają w stanie gotowości.
- *Nieodpowiednie stosowanie wielowątkowości może jednak zaszkodzić…*



Sposoby przetwarzania

Asynchroniczne zadania można przetwarzać następująco:

- Uruchomienie asynchronicznego procesu i ciągłe **sprawdzanie obiektu IAsyncResult** w celu zbadania, czy zadanie zostało zakończone
- **Skojarzenie procesu z tzw. uchwytem** (Threading.Handle).

Rozwiązanie to umożliwia uruchomienie dowolne wybranych procesów asynchronicznych – następnie można poczekać (np. metoda WHandle.WaitAll()) , jak wszystkie lub część się zakończy w celu dokończenia głównego zadania.
Ta metody wymaga dokładnego zaplanowania harmonogramu.
Jednocześnie program (główny wątek) **zatrzyma się** podczas wspomnianego oczekiwania.

Sposoby przetwarzania

Asynchroniczne zadania można przetwarzać następująco:

- Przypisanie do uruchomionego procesu **funkcji zwrotnej** – pozwala to na równoległe wykonywanie innych zadań.

Po zakończeniu asynchronicznego zadania, jest wywoływana metoda zwrotna, która może „posprzątać" po procesie i powiadomić inne części aplikacji o wykonaniu zadania.

- Wykorzystać słowa kluczowe **async** i **await**.

Modele przetwarzania

Wskazane metody można przypisać do określonych modeli:

- Model programowania asynchronicznego - **Asynchronous Programming Model** (w skrócie **APM**) (oparty na parze metod: jedna służy do rozpoczęcia realizacji operacji Begin, a druga do pobrania danych po jej zakończeniu End)
- Model asynchroniczny bazujący na zdarzeniach - **Event-based Asynchronous Pattern** (w skrócie **EAP**). Klasa działająca według tego wzorca udostępnia metodę rozpoczynającą operację oraz odpowiadające jej zdarzenie, które jest zgłaszane po zakończeniu operacji.

Modele przetwarzania

Wskazane metody można przypisać do określonych modeli:

- Model **TPL** (**Task Parallel Library**) – oparty na metodzie Wait oraz właściwości Result.
- + rozszerzenie TPL (4.5 i C# 5.0), które dotyczy wyłącznie sposobów oczekiwania na zakończenie aktualnie realizowanych operacji asynchronicznych - nie określa natomiast, jak te operacje mają być rozpoczynane i kończone.
- Model ten jest określany angielskim terminem ***awaitable***, a jest on obsługiwany przez asynchroniczne możliwości języka C# (chodzi o słowa kluczowe **async** oraz **await**).

- Od wersji 4.0 wykorzystujemy głównie Task Parallel Library (TPL), która opiera się na koncepcji zadań, która reprezentuje operację asynchroniczną. W .NET Framework TPL jest preferowanym interfejsem API do pisania kodu wielowątkowego, asynchronicznego i równoległego.

C# - async i await

ASYNCHRONICZNOŚĆ WBUDOWANA W JĘZYK

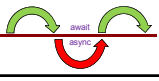
C# - async i await

Zasady:

- Metody będą opatrzone modyfikatorem **async** oraz każde wywołanie asynchroniczne zostaje poprzedzone słówkiem **await**.
- Kod strukturalnie niczym nie różni się od kodu synchronicznego, nie wliczając w to użycia tych dwóch słów kluczowych.
- Wszystkie konstrukcje językowe (try, using) działają tutaj poprawnie.

C# - async i await

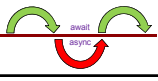
- Słowo kluczowe **async** niejako włącza słowo kluczowe **await** w metodzie.
- Nie powoduje ono w żaden sposób wykonania metody w wątku z puli.
- Jeśli uruchomimy taką metodę, wykona się ona po prostu synchronicznie.
- W momencie, gdy kompilator natrafi na słowo kluczowe **await**, to w tym miejscu może wprowadzić asynchroniczność.
- **await** przyjmuje jeden argument, którym jest tzn. typ **awaitable**.



C# - async i await

Podczas wykonania sprawdzany jest stan obiektu **awaitable**:

- Jeśli dostępne są już rezultaty, metoda wykonuje się dalej w sposób synchroniczny.
- Jeśli nie, to metoda asynchroniczna jest przerywana, a wywołanie wraca do metody wołającej metodę asynchroniczną – pozwalając jej się dalej wykonywać.
- Dzięki temu m.in. nasz interfejs się nie zawiesza.
- W momencie, gdy **awaitable** zasygnalizuje, że operacja już się zakończyła, wznawiane jest wykonanie metody asynchronicznej od ostatniego miejsca przystanku.
- *Czym jest to awaitable?*



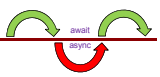
C# - async i await

awaitable to obiekty, które mogą współpracować z async, tutaj tzw. *taski*.

Typ zwracany jest jednym z następujących typów:

- **Task<TResult>**: jeśli metoda zawiera instrukcję return, w której argument jest typu TResult.
- **Task**: kiedy nie chcemy nic zwrócić, ale chcemy mieć możliwość czekania („awaitowania")
- **void**: nie można nic zwrócić, ani na nic zaczekać.

Uwaga: (Task(Of T) -> Visual Basic .NET)



C# - async i await – bez asynchoniczności

```
int AccessTheWebSynch()
{
    HttpClient client = new HttpClient();

    string getString = client.GetStringAsync("http://msdn.microsoft.com").Result;

    DoIndependentWork(this.lblLenghtSynchr);

     string urlContents = getString;

     return urlContents.Length;
}

private void DoIndependentWork(Label lbl)
      {
          lbl.Text = "Working.....";
      }
```

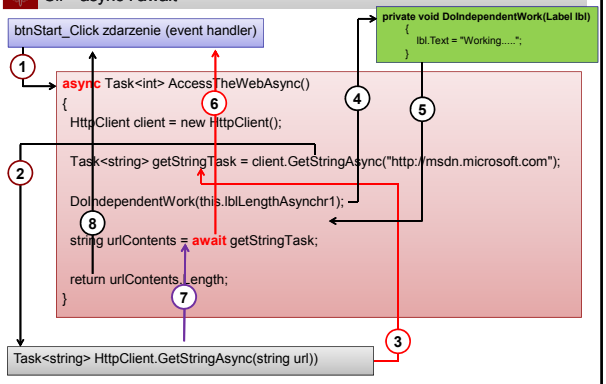
C# - async i await

```
async Task<int> AccessTheWebAsync1()
{
  HttpClient client = new HttpClient();

  Task<string> getStringTask = client.GetStringAsync("http://msdn.microsoft.com");

  DoIndependentWork(this.lblLengthAsynchr1);

  string urlContents = await getStringTask;

  return urlContents.Length;
}
```


C# - async i await

```
async Task<int> AccessTheWebAsync2()
{
  DoIndependentWork(this.lblLengthAsynchr2);

  HttpClient client = new HttpClient();

  string urlContents = await client.GetStringAsync("http://msdn.microsoft.com");

  return urlContents.Length;
}
```


C# - async i await



C# - async i await – kilka wątków

- Inną ciekawą możliwością, o której warto wspomnieć, jest komponowanie awaiterów.
- Można uruchomić kilka operacji na raz i czekać na skończenie wszystkich, bądź dowolnej z nich.

```
public async Task<int> GetGrandTotalPrice()
{
        Task<int> t1 = GetOrderTotalPrice(1);
        Task<int> t2 = GetOrderTotalPrice(2);
        await Task.WhenAll(t1, t2);
        return t1.Result + t2.Result;
}
```


C# - async i await – kilka wątków

```
public async void ProcessWriteMult()
{
   string folder = @"tempfolder\";
   List<Task> tasks = new List<Task>();
   List<FileStream> sourceStreams = new List<FileStream>();
   try
   {
      for (int index = 1; index <= 10; index++)
      {
         string text = "In file " + index.ToString() + "\r\n";
         string fileName = "thefile" + index.ToString("00") + ".txt";
         string filePath = folder + fileName;
         byte[] encodedText = Encoding.Unicode.GetBytes(text);

         FileStream sourceStream = new FileStream(filePath,FileMode.Append, FileAccess.Write,
               FileShare.None, bufferSize: 4096, useAsync: true);

         Task theTask = sourceStream.WriteAsync(encodedText, 0, encodedText.Length);
         sourceStreams.Add(sourceStream);

         tasks.Add(theTask);
      }

      await Task.WhenAll(tasks);
   }
   finally
   {
      foreach (FileStream sourceStream in sourceStreams)
      {
         sourceStream.Close();
      }
   }
}
```


C# - async i await – anulowanie

- Platforma .NET definiuje standardowy mechanizm służący do anulowania długotrwałych operacji. Operacje zapewniające możliwość anulowania pobierają argument typu **CancellationToken**.
- Jeśli ustawimy go w stan anulowania, to jeśli tylko pojawi się taka możliwość, operacja zostanie przerwana wcześniej, a nie będzie wykonywana do końca.
- Sam typ **CancellationToken** nie udostępnia żadnych metod służących do zainicjowania anulowania — anulowanie jest obsługiwane przez odrębny obiekt — **CancellationTokenSource**.
- *Uwaga: jeśli korzystamy z mechanizmu anulowania, trzeba być przygotowanym na to, że jego użycie nie da żadnego efektu.*

C# - async i await – anulowanie

```
CancellationTokenSource cts;

private async void btnStartAsynchCancel_Click
                                (object sender, EventArgs e)
{
 cts = new CancellationTokenSource();
 txtResults.Clear();
 try
  {
    cts.CancelAfter(5000);
    await AccessTheWebAsync(cts.Token);
    txtResults.Text += "\r\nDownloads succeeded.\r\n";
  }
  catch (OperationCanceledException)
  {
    txtResults.Text += "\r\nDownloads canceled.\r\n";
  }
  catch (Exception)
  {
    txtResults.Text += "\r\nDownloads failed.\r\n";
  }
 cts = null;
}

private void btnCancel_Click(object sender, EventArgs e)
{
    if (cts != null)
        cts.Cancel();
}
```

```
async Task AccessTheWebAsync(CancellationToken ct)
{
  HttpClient client = new HttpClient();
  List<string> urlList = SetUpURLList();
  foreach (var url in urlList)
  {
    HttpResponseMessage response = await
                                client.GetAsync(url, ct);
    byte[] urlContents = await
                response.Content.ReadAsByteArrayAsync();
    txtResults.Text +=
            String.Format("\r\nLength of the downloaded
                string: {0}.\r\n", urlContents.Length);
  }
}

private List<string> SetUpURLList()
{
  List<string> urls = new List<string>
  {
  "http://msdn.microsoft.com",
  "http://msdn.microsoft.com/apps/br211380.aspx",
  "http://msdn.microsoft.com/en-us/library/hh290136.aspx",
  };
 return urls;
}
```

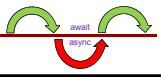

Podsumowanie

- Istnieje coraz więcej klas dających wsparcie programowaniu asynchronicznemu.

Obszar aplikacji	**Obsługa interfejsów API, która obejmuje metody asynchroniczne**
Dostęp do sieci Web	HttpClient
Praca z plikami	StorageFile, StreamWriter, StreamReader, XmlReader
Praca z obrazami	MediaCapture, BitmapEncoder, BitmapDecoder
Programowanie WCF	Operacje synchroniczne i asynchroniczne
Bazy danych	SqlCommand

Podsumowanie

- Użycie przedstawionego rozszerzenia językowego nie jest oczywiście limitowane tylko do celów interfejsowych.
- Możemy go także używać do asynchronicznego wczytywanie czy zapisywanie plików oraz do tych wszystkich sytuacji, gdzie pojawia się potrzeba wielowątkowości.
- **async** i **await** znacząco upraszcza obsługę asynchroniczności w interfejsie użytkownika.
- Kod staje się prosty, czytelny i co ważne zarządzalny.



TEST

Test

```
using System.Diagnostics;
using System.Threading.Tasks;
namespace ConsoleApplicationTest
{
   class Program
   {
      static void Main(string[] args)
      {
         Task theTask = ProcessAsync(); // A
         int x = 0;  // B
         theTask.Wait(); // C - Waits for the task to complete execution
      }

      static async Task ProcessAsync()
      {
         var result = await DoSomethingAsync();  // D
         int y = 0;  // E
      }

      static async Task<int> DoSomethingAsync()
      {
         Debug.WriteLine("before"); // F
         await Task.Delay(10000); // G
         Debug.WriteLine("after"); // H
         return 5; // I
} }}
```

Kod będzie się wykonywał w kolejności:
a) A D E B C…
b) A B C D E…
c) A D F G H …
d) A D F G B …

Test

```
using System.Diagnostics;
using System.Threading.Tasks;
namespace ConsoleApplicationTest
{
   class Program
   {
      static void Main(string[] args)
      {
         Task theTask = ProcessAsync(); // A {1}
         int x = 0;  // B {5}
         theTask.Wait(); // C {6} - Waits for the task to complete execution
      }

      static async Task ProcessAsync()
      {
         var result = await DoSomethingAsync();  // D {2}
         int y = 0;  // E {9}
      }

      static async Task<int> DoSomethingAsync()
      {
         Debug.WriteLine("before"); // F {3}
         await Task.Delay(10000); // G {4}
         Debug.WriteLine("after"); // H {7}
         return 5; // I {8}
} }}
```

Kod będzie się wykonywał w kolejności:
a) A D E B …
b) A B C D …
c) A D F G H …
d) A D F G B …

a) A D F G B C H I E

Podsumowanie

Metody asynchroniczne stosuje się:

- gdy brakuje wolnych wątków;
- gdy aplikacja będzie szybciej/lepiej działać przy użyciu metod asynchronicznych;
- gdy operacja opiera się w głównej mierze na operacjach I/O (zapis/odczyt) na dysku lub na operacjach sieciowych, które mogą zablokować wątek poprzez wolne działanie;
- chce się zaimplementować mechanizm zatrzymywania lub przerywania długich żądań (np. przesyłanie plików).

Podsumowanie

Słowa kluczowe — async, await, Task

- **async** — tym słowem kluczowym oznacza się metodę asynchroniczną. Dzięki temu kompilator wie, że będzie to metoda asynchroniczna. Metoda asynchroniczna według konwencji powinna mieć na końcu nazwy słowo **async**.
- **await** — informuje o metodzie, która czeka na dane zwracane w sposób asynchroniczny. Słowo **await** może być użyte tylko w metodzie asynchronicznej oznaczonej słowem **async**. Dzięki słowu **await** kompilator nie blokuje wątku ani wykonywania innych części kodu na czas pobierania danych.
- **Task** — jest to typ danych zwracanych przez metodę asynchroniczną. Typ Task<typ_zwracany> informuje kompilator o danych zwracanych asynchronicznie.

Podsumowanie

- Klasa Task stanowi jednak coś więcej niż jedynie opakowanie dla puli wątków. Zarówno jej, jak i powiązanych z nią typów tworzących bibliotekę TPL (Task Parallel Library) można z powodzeniem używać w znacznie większej liczbie sytuacji.
- Biblioteka TPL została wprowadzona w .NET 4.0, jednak w kolejnej wersji platformy - 4.5 - zyskała znacząco większe znaczenie, gdyż asynchroniczne mechanizmy języka wprowadzone w C# 5.0 są w stanie operować bezpośrednio na obiektach zadań.
- Z tego powodu w najnowszej wersji .NET Framework bardzo wiele API zostało rozbudowanych o obsługę operacji asynchronicznych bazujących na wykorzystaniu zdarzeń.

Kluczowymi elementami biblioteki TPL są dwie klasy:

- Task oraz dziedzicząca po niej klasa Task<T>.
- Klasa bazowa Task reprezentuje pewną operację, której wykonanie może zająć trochę czasu.
- Klasa Task<T> stanowi rozszerzenie reprezentujące operację, która po wykonaniu zwraca pewien wynik (typu T).
- (Nieogólna klasa Task nie zwraca żadnego wyniku. Stanowi ona asynchroniczny odpowiednik typu void).
- Warto zwrócić uwagę, że nie są to pojęcia, które muszą być powiązane wątkami.

Asynch

```
using System;
using System.Net.Http;
using System.Threading.Tasks;
using System.Windows.Forms;
namespace WindowsForms_Asyn
{
    public partial class frmTest : Form
    {
        public frmTest()
        {
            InitializeComponent();
        }
        private void DoIndependentWork(Label lbl)
        {
            lbl.Text = "Working....";
        }
        int AccessTheWebSynch()
        {
            HttpClient client = new HttpClient();

            string getString = client.GetStringAsync("http://msdn.microsoft.com").Result;

            DoIndependentWork(this.lblLenghtSynchr);

            string urlContents = getString;

            return urlContents.Length;
        }
        async Task<int> AccessTheWebAsync1()
        {
            HttpClient client = new HttpClient();

            Task<string> getStringTask = client.GetStringAsync("http://msdn.microsoft.com");

            DoIndependentWork(this.lblLengthAsynchr1);

            string urlContents = await getStringTask;

            return urlContents.Length;
        }
        async Task<int> AccessTheWebAsync2()
        {
            DoIndependentWork(this.lblLengthAsynchr2);

            HttpClient client = new HttpClient();

            string urlContents = await client.GetStringAsync("http://msdn.microsoft.com");

            return urlContents.Length;
        }

        private void btnStartSynchr_Click(object sender, EventArgs e)
        {
            lblLenghtSynchr.Text = (AccessTheWebSynch()).ToString();
        }

        private async void btnStart1_Click(object sender, EventArgs e)
        {
            lblLengthAsynchr1.Text = (await AccessTheWebAsync1()).ToString();
        }

        private async void btnStart2_Click(object sender, EventArgs e)
        {
            lblLengthAsynchr2.Text = (await AccessTheWebAsync2()).ToString();
        }

        private void tmrProgress_Tick(object sender, EventArgs e)
        {
            if (++prbProgress.Value == prbProgress.Maximum)
                prbProgress.Value = 0;
        }

    }
}
```

Test

```
public partial class MainWindow : Window
{
    // . . .
    private async void startButton_Click(object sender, RoutedEventArgs e)
    {
        // A
        Task<int> getLengthTask = AccessTheWebAsync();

        // B
        int contentLength = await getLengthTask;

        // C
        resultsTextBox.Text +=
            String.Format("\r\nLength of the downloaded string: {0}.\r\n", contentLength);
    }

    async Task<int> AccessTheWebAsync()
    {
        // D
        HttpClient client = new HttpClient();
        Task<string> getStringTask =
            client.GetStringAsync("http://msdn.microsoft.com");

        // E
        string urlContents = await getStringTask;

        // F
        return urlContents.Length;
    }
}
```

W jakiej kolejności będzie wykonywał się kod po wywołaniu zdażenia - startButton_Click?

Test

```
public partial class MainWindow : Window
{
    // . . .
    private async void startButton_Click(object sender, RoutedEventArgs e)
    {
        // A – {1}
        Task<int> getLengthTask = AccessTheWebAsync();

        // B – {4}
        int contentLength = await getLengthTask;

        // C – {6}
        resultsTextBox.Text +=
            String.Format("\r\nLength of the downloaded string: {0}.\r\n", contentLength);
    }

    async Task<int> AccessTheWebAsync()
    {
        // D – {2}
        HttpClient client = new HttpClient();
        Task<string> getStringTask =
            client.GetStringAsync("http://msdn.microsoft.com");

        // E – {3}
        string urlContents = await getStringTask;

        // F – {5}
        return urlContents.Length;
    }
}
```

START

KONIEC

W jakiej kolejności będzie wykonywał się kod po wywołaniu zdażenia - startButton_Click?

Platforma .NET

(...)

„ADO .NET”

Dr inż. Krzysztof Smółka

ksmolka@p.lodz.pl

Politechnika Łódzka

Dostęp do baz danych za pomocą ADO.NET

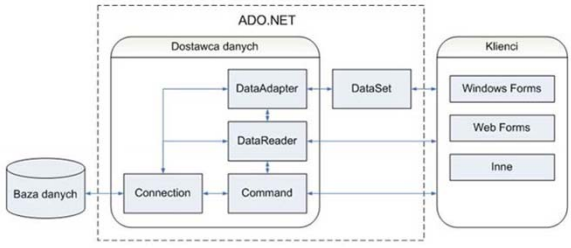
- Do dostępu do baz danych w .NET Framework służy biblioteka o nazwie ADO.NET (tworząca przestrzeń nazw ***System.Data***).
- Biblioteka ta jest także integralnym składnikiem .NET Compact Framework, dzięki czemu programiści, którzy tworzą wersje aplikacji dla urządzeń przenośnych, nie muszą zmieniać kodu źródłowego odpowiedzialnego za dostęp do baz danych.

Dostęp do baz danych za pomocą ADO.NET

- ADO.NET umożliwia dostęp do praktycznie każdej bazy danych z aplikacji napisanej w dowolnym języku obsługiwanym w .NET Framework.
- Funkcjonalność tę osiągnięto rozdzielając bibliotekę ADO.NET na dwa komponenty –
- zestaw *dostawców danych* oraz uniwersalny obiekt *DataSet*.

Dostawca danych

Zestaw obiektów zwany *dostawcą danych* (*Data Provider*) pośredniczy w wymianie danych pomiędzy obiektami DataSet a bazami danych.



Dostawca danych

W podstawowej instalacji .NET Framework dostępni są następujący dostawcy danych:

- dostawca danych dla SQL Server (przestrzeń nazw *System.Data.SqlClient)* - obsługa baz danych SQL Server w wersji 7.0 lub nowszej,
- dostawca danych dla OLEDB (przestrzeń nazw *System.Data.OleDb)* - połączenie z dowolnym źródłem danych OLEDB,
- dostawca danych dla ODBC (przestrzeń nazw *System.Data.Odbc)* - obsługa źródeł danych dostępnych za pośrednictwem sterowników ODBC,
- dostawca danych dla Oracle (przestrzeń nazw *System.Data.OracleClient)* - obsługa baz danych firmy Oracle.

Dostawca danych

- Dostawcy danych dla OLEDB oraz dla ODBC są **dostawcami uniwersalnymi** - mogą zapewnić dostęp do dowolnej bazy danych, z którą można komunikować się za pośrednictwem OLEDB lub ODBC.
- Większą wydajność mają jednak dedykowani dostawcy danych, jakimi są dostawcy dla SQL Server oraz dla Oracle.
- Dedykowanych dostawców danych dla innych systemów baz danych można pobrać od producentów tych systemów baz danych.

Dostawca danych

Każdy dostawca danych składa się z następujących obiektów:

- **obiekt *Connection*** (SqlConnection, OleDbConnection, OdbcConnection, OracleConnection) - reprezentuje połączenie z konkretną bazą danych.
- Posiada wszystkie informacje potrzebne do połączenia się z bazą, uwierzytelnienia jako określony użytkownik i prowadzenia dalszej komunikacji,

Dostawca danych

Każdy dostawca danych składa się z następujących obiektów:

- **obiekt *Command*** (SqlCommand, OleDbCommand, OdbcCommand, OracleCommand) - służy do wykonywania poleceń dotyczących danych zgromadzonych w bazie.
- Polecenie może zostać zdefiniowane albo jako zapytanie SQL, które ma zostać wykonane, albo jako nazwa *procedury przechowywanej* zapisanej w bazie.
- Właściwość *Parameters* obiektu *Command* zawiera kolekcję parametrów przekazywanych do polecenia.
- Komunikacja z bazą danych odbywa się za pośrednictwem obiektu *Connection,*

Dostawca danych

Każdy dostawca danych składa się z następujących obiektów:

- **obiekt *DataReader*** (SqlDataReader, OleDbDataReader, OdbcDataReader, OracleDataReader) - służy do odczytu danych zwróconych w wyniku wykonania zapytania zapisanego w obiekcie *Command*.
- Dane przesyłane są porcjami - po jednym wierszu,

Dostawca danych

Każdy dostawca danych składa się z następujących obiektów:

- **obiekt *DataAdapter*** (SqlDataAdapter, OleDbDataAdapter, OdbcDataAdapter, OracleDataAdapter) — pośredniczy w komunikacji pomiędzy obiektem *DataSet* a bazą danych.

Dostawca danych zawiera także inne obiekty, umożliwiające obsługę transakcji i automatyczne konfigurowanie obiektów *Command,* a także obiekty ułatwiające przekazywanie parametrów poleceń do bazy danych, obsługę wyjątków i błędów oraz definiowanie uprawnień.

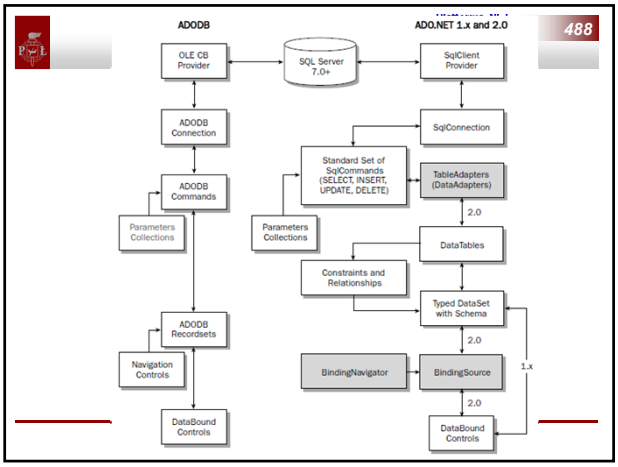
- ,

Obiekt DataSet

- **Obiekt *DataSet*** służy do przechowywania w pamięci operacyjnej komputera danych pobranych z różnych źródeł.
- W obiekcie *DataSet* można odzwierciedlić pełną strukturę bazy danych - *tabele* składające się z kolumn i wierszy, *ograniczenia* określające, jaką postać mogą przyjmować dane znajdujące się w tabelach oraz *relacje* opisujące wzajemne powiązania pomiędzy danymi zawartymi w tabelach.

Obiekt DataSet

- W przekazywaniu danych pomiędzy źródłem danych a obiektem ***DataSet*** pośredniczy obiekt ***DataAdapter*** odpowiedniego dostawcy danych.
- Ponieważ dane, przechowywane w obiekcie *DataSet,* są niezależne od jakichkolwiek zewnętrznych źródeł danych, model pracy z danymi w .NET często nazywany jest modelem „odłączonym" - aplikacja może połączyć się z bazą danych, wczytać potrzebne informacje do obiektu *DataSet* i zamknąć połączenie.
- Po zmodyfikowaniu zawartości obiektu *DataSet* aplikacja może ponownie połączyć się z bazą danych i przekazać jej zmodyfikowane rekordy.



LINQ

- **LINQ (czyt. Link)** - Language Integrated Query - "zapytania zintegrowane z językiem programowania". część technologii Microsoft .NET, która została opracowana przez Andersa Hejlsberga – znanego z zaprojektowania języka Delphi i języka C#.
- Technologia LINQ umożliwia zadawanie pytań na obiektach.
- Składnia języka LINQ jest prosta i przypomina SQL.

LINQ

Technologia LINQ pojawiła się wraz z platformą .NET w wersji 3.5. LINQ to mechanizm ułatwiający pracę z danymi po stronie aplikacji. Dane mogą pochodzić z:

- bazy danych (LINQ to sql),
- dokumentów xml (LINQ to xml),
- zwykłych obiektów (LINQ to objects).

Według twórców języka C#, w niektórych przypadkach zapytanie LINQ może o 40% skrócić ilość kodu w stosunku do tradycyjnych metod dostępu do danych!!!

LINQ

Być może macie dość hymnów pochwalnych istniejących w sieci na cześć LINQ ☺

Zatem oddzielmy mity od rzeczywistości:

- LINQ nie zamieni najbardziej skomplikowanego zapytania w zapytanie jednolinijkowe.
- LINQ nie oznacza, że już nigdy nie będziesz musiał się zajmować zwykłym SQL-em.
- LINQ nie sprawi magicznie, iż nagle staniesz się geniuszem w zakresie korzystania z danych.

Pomimo wszystkich powyższych stwierdzeń LINQ jest nadal świetnym sposobem wyrażania zapytań w środowisku zorientowanym obiektowo. Nie jest to zatem złoty środek, ale *bardzo* funkcjonalne narzędzie w arsenale programisty.

LINQ

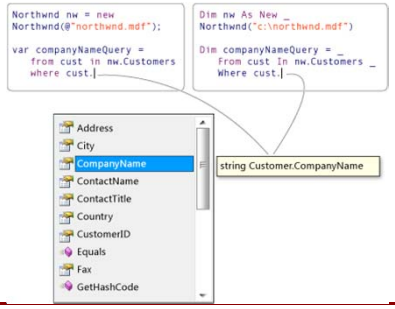
- LINQ stanowi warstwę abstrakcji nad różnymi źródłami danych.
- LINQ posiada pełne wsparcie dla transakcji, widoków oraz procedur przechowanych.
- Jeśli LINQ ma działać musi znać mapę całej bazy danych.
- Zapytanie LINQ zwraca kolekcję z przestrzeni nazw typów ogólnych.
- Kolekcja ta może być modyfikowana, a następnie zwrócona do źródła.
- Dzięki temu zachowywana jest pełna kontrola typów danych i ich konwersji w poszczególnych mechanizmach pośredniczących w pobieraniu danych.

LINQ

```
// Przykład wybrania w zapytaniu wszystkich obiektów
// z właściwością SomeProperty mniejszą niż 10,
int someValue = 5;
var results = from c in someCollection
        let x = someValue * 2
        where c.SomeProperty < x
        select new {c.SomeProperty, c.OtherProperty};

foreach (var result in results)
        { Console.WriteLine(result); }
```


LINQ

Pełne sprawdzanie i obsługa technologii IntelliSense



Standardowe operatory zapytań LINQ

Rodzaj operacji	Metody rozszerzające
Pobieranie danych	Select, SelectMany
Sortowanie	OrderBy, ThenBy, OrderByDescending, ThenByDescending, Reverse
Filtrowanie	Where
Operacje arytmetyczne	Aggregate, Average, Count, LongCount, Max, Min, Sum
Konwersja	Cast, OfType, ToArray, ToDictionary, ToList, ToLookup, ToSequence
Pobieranie elementu	Element DefaultIfEmpty, ElementAt, ElementAtOrDefault, First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault
Porównywanie	SequenceEqual
Tworzenie	Empty, Range, Repeat
Grupowanie	GroupBy
Łączenie	GroupJoin, Join
Wybór i pomijanie elementów	Skip, SkipWhile, Take, TakeWhile
Kwantyfikatory	All, Any, Contains
Operacje na zbiorach	Concat, Distinct, Except, Intersect, Union

Standardowe operatory zapytań LINQ

- **Select** – operator ten jest używany, by wybrać z kolekcji odpowiedniego rodzaju dane (zbiory/podzbiory danego obiektu)
- **SelectMany** – jest używany, by dokonać wyświetlenia w przypadku relacji jeden-do-wielu, czyli jeśli obiekt w kolekcji zawiera inną kolekcję jako część danych, SelectMany będzie użyte do wybrania całej pod-kolekcji.
- **Where** – operator Where pozwala zdefiniować zbiór zasad dla każdego obiektu w kolekcji. Wszystkie te obiekty, które nie pasują do wybranej reguły, są odfiltrowywane.

Standardowe operatory zapytań LINQ

- **Sum** – używany do otrzymywania sumy
- **Min** – używany do otrzymania minimalnej wartości
- **Max** – używany do otrzymania maksymalnej wartości
- **Average** – używany do otrzymania średniej
- **Aggregate** – używany do stworzenia wyrażenia agregującego wszystkie elementy w kolekcji

Standardowe operatory zapytań LINQ

- **Join** – operator Join to operator bazujący na dwóch kolekcjach i tworzy z nich obiekt wynikowy.
- **GroupJoin** – operator dla dokonania połączenia grupowego. Tak jak w przypadku operatora Select, wynik łączenia jest instancją nowej klasy z wszystkimi członkami obiektów źródłowych lub ich podzbiorem.
- **Take** – operator Take jest używany do wybrania pierwszych n obiektów z kolekcji
- **TakeWhile**
- **Skip**
- **SkipWhile**

Standardowe operatory zapytań LINQ

- **OfType** – operator używany do wybrania elementów konkretnego typu
- **Concat** – operator tworzący konkatenacje dwóch kolekcji
- **OrderBy** – sortuje wyniki po wybranym elemencie kolekcji – według określonego klucza
- **OrderByDescending** – sortuje w odwrotnej kolejności niż standardowo
- **ThenBy** – określa, po czym sortować w następnym etapie
- **ThenByDescending**
- **Reverse** – odwraca kolekcje
- **GroupBy**

Standardowe operatory zapytań LINQ

- **Distinct** – usuwa duplikacje wartości kluczy w kolekcji
- **Union** – operacja 'suma' – zwraca połączone ze sobą dwie kolekcje
- **Intersect** – operacja 'iloczyn' – zwraca część wspólną dwóch kolekcji
- **Except**
- **EqualAll** – sprawdza, czy wszystkie elementy w kolekcji są takie same

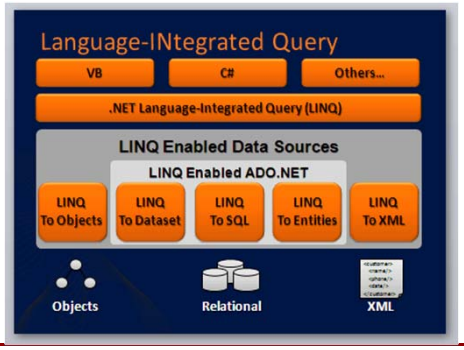
Standardowe operatory zapytań LINQ

- **First** – zwraca pierwszy element kolekcji
- **FirstOrDefault** – zwraca pierwszy element kolekcji lub NULL, gdy kolekcja jest pusta
- **Last** – zwraca ostatni element kolekcji
- **LastOrDefault** – zwraca ostatni element kolekcji lub NULL, gdy kolekcja jest pusta
- **Single**
- **ElementAt** – zwraca element o wybranym indeksie (index) z kolekcji

Standardowe operatory zapytań LINQ

- **Any** – sprawdza, czy którykolwiek element kolekcji spełnia warunek podany w nawiasach
- **All** – sprawdza, czy wszystkie elementy kolekcji spełniają warunek podany w nawiasach
- **Contains** – sprawdza, czy kolekcja zawiera element podany w argumencie
- **Count** – zlicza elementy kolekcji

LINQ



Dostawcy - LINQ

- Większość przypadków korzysta z dostawcy ***LINQ to Objects***, natomiast zakres stosowania jest dużo szerszy, jak np. ***LINQ to Entities***, czyli dostawcy współpracującego z bazami danych.

- Poniżej zamieszczone są krótkie opisy niektórych innych technologii związanych z LINQ.
- Ich lista nie jest bynajmniej kompletna, gdyż każdy może tworzyć nowych dostawców LINQ.

Dostawcy - LINQ

- Platforma Entity Framework jest to technologia dostępu do danych dostarczana jako jeden z elementów .NET Framework, umożliwiająca kojarzenie baz danych z warstwą obiektów.
- Obsługuje ona wiele różnych rodzajów baz danych.
- EF bazuje na danych typu IQueryable<T>.

- Ponieważ interfejs IQueryable<T> nie jest używany wyłącznie przez Entity Framework, dlatego też do obsługi danych tego typu używany jest standardowy zbiór metod rozszerzeń zdefiniowanych w klasie Queryable należącej do przestrzeni nazw System.Linq, choć mechanizm ten został zaprojektowany w taki sposób, by każdy dostawca mógł korzystać z własnych operatorów.

Dostawcy - LINQ

- Kolejną technologią dostępu do danych jest **LINQ to SQL**.
- W odróżnieniu do Entity Framework została ona opracowana konkretnie z myślą o bazie danych Microsoft SQL Server.
- Cechuje ją nieco inna filozofia działania: została ona zaprojektowana jako wygodny API służący do korzystania z danych przechowywanych na serwerze bazy danych, a nie jako warstwa pomiędzy bazą i obiektami aplikacji; dlatego też nie dysponuje ona rozbudowanymi możliwościami odwzorowywania struktury informacji w bazie i projektu naszego dziedziny.

- LINQ to SQL udostępnia obiekty reprezentujące konkretne tabele bazy danych. Te obiekty tabel implementują interfejs IQueryable<T>, a zatem pod względem pisania zapytań LINQ to SQL działa podobnie do EF.

Dostawcy - LINQ

Klient WCF Data Services

- WCF Data Services zapewniają możliwości udostępniania i korzystania z danych za pośrednictwem protokołu HTTP, używając przy tym standardowego protokołu OData — ang. Open Data Protol.
- Prezentuje on dane przy użyciu języka XML lub JSON i definiuje sposoby zapisu zapytań zawierających operacje filtrowania, porządkowania oraz korzystających ze złączeń.
- Kliencka część tej technologii zawiera dostawcę LINQ korzystającego z interfejsu IQueryable<T>.
- Niemniej jednak ponieważ standard OData pozwala na zapis jedynie stosunkowo niewielu rodzajów zapytań, zatem dostawca ten udostępnia jedynie nieznaczną część standardowych operatorów LINQ.

Dostawcy - LINQ

- **Parallel LINQ (PLINQ)**
- Dostawca *Parallel LINQ* jest podobny do ***LINQ to Objects*** pod tym względem, że także korzysta z obiektów i delegatów, a nie z drzew wyrażeń i tłumaczenia zapytań.
- Jednak kiedy zaczniemy go prosić o zwracanie danych, to wszędzie tam, gdzie jest to możliwe, będzie starał się działać wielowątkowo, używając przy tym puli wątków, by w możliwe wydajny sposób wykorzystywać zasoby procesora.

Dostawcy - LINQ

- **LINQ to XML**

LINQ to XML nie jest dostawcą LINQ. ☺

Wspominam tu o nim, ponieważ jego nazwa sprawia, że można go uznać za dostawcę.

- W rzeczywistości jest to jednak API, którego możliwości funkcjonalne pozwalają na tworzenie i przetwarzanie dokumentów XML.
- Nosi on nazwę LINQ to XML, gdyż został zaprojektowany, by ułatwić tworzenie zapytań LINQ operujących na dokumentach XML; jednak zadanie to realizuje, przetwarzając dokumenty XML do postaci modeli obiektów .NET.

Dostawcy - LINQ

- **LINQ to XML**
- Biblioteka klas .NET Framework udostępnia dwa odrębne API służące do tego celu: oprócz LINQ to XML dostępny jest także XML Document Object Model (w skrócie DOM; czyli model obiektów dokumentu).

- DOM bazuje na standardzie niezależnym od używanej platformy, dlatego też nie jest idealnie dopasowany do rozwiązań stosowanych w .NET, a w porównaniu z większością klas biblioteki .NET Framework wydaje się on czasem dosyć dziwny…….

Dostawcy - LINQ

- **LINQ to XML**
- LINQ to XML został zaprojektowany w całości z myślą o .NET Framework, dlatego też znacznie lepiej integruje się ze standardowymi rozwiązaniami stosowanymi w języku C#.
- Dotyczy to także bezproblemowej współpracy z technologią LINQ, którą zapewnia, udostępniając metody pobierające wszelkie informacje z dokumentów i udostępniające je w postaci danych typu IEnumerable<T>.
- To właśnie dzięki temu LINQ to XML może odwoływać się do LINQ to Objects w celu definiowania i wykonywania zapytań.

PRZYKŁAD

Przykład - LINQ

- Związanie rozszerzeń LINQ z interfejsem IEnumerable<> oznacza, że zwykłe kolekcje platformy .NET (takie jak List<>), które implementują ten interfejs, mogą być źródłem danych w technologii LINQ to Objects.
- Jeżeli zatem dysponujemy kolekcją obiektów, możemy je dowolnie filtrować, sortować, analizować w dowolny sposób, jak również łączyć z inną kolekcją.

Przykład - LINQ

```
// klasa Osoba
class Osoba
{
  public int Id;
  public string Imię, Nazwisko;
  public int NumerTelefonu;
  public int Wiek;
}
// źródło danych
static List<Osoba> listaOsob = new List<Osoba>
{
  new Osoba { Id = 1, Imię = "Jan", Nazwisko = "Kowalski", NumerTelefonu = 7272024, Wiek = 39 },
  new Osoba { Id = 2, Imię = "Andrzej", Nazwisko = "Kowalski", NumerTelefonu = 7272020, Wiek = 29 },
  new Osoba { Id = 3, Imię = "Maciej", Nazwisko = "Bartnicki", NumerTelefonu = 7272021, Wiek = 42 },
  new Osoba { Id = 4, Imię = "Witold", Nazwisko = "Mocarz", NumerTelefonu = 7272022, Wiek = 26 },
  new Osoba { Id = 5, Imię = "Adam", Nazwisko = "Kowalski", NumerTelefonu = 7272023, Wiek = 6 },
  new Osoba { Id = 6, Imię = "Ewa", Nazwisko = "Mocarz", NumerTelefonu = 7272025, Wiek = 11 }
};
```


Przykład - LINQ

```
// klasa Osoba
class Osoba
{
  public int Id;
  public string Imię, Nazwisko;
  public int NumerTelefonu;
  public int Wiek;
}
// źródło danych
static List<Osoba> listaOsob = new List<Osoba>
{
  new Osoba { Id = 1, Imię = "Jan", Nazwisko = "Kowalski", NumerTelefonu = 7272024, Wiek = 39 },
  new Osoba { Id = 2, Imię = "Andrzej", Nazwisko = "Kowalski", NumerTelefonu = 7272020, Wiek = 29 },
  new Osoba { Id = 3, Imię = "Maciej", Nazwisko = "Bartnicki", NumerTelefonu = 7272021, Wiek = 42 },
  new Osoba { Id = 4, Imię = "Witold", Nazwisko = "Mocarz", NumerTelefonu = 7272022, Wiek = 26 },
  new Osoba { Id = 5, Imię = "Adam", Nazwisko = "Kowalski", NumerTelefonu = 7272023, Wiek = 6 },
  new Osoba { Id = 6, Imię = "Ewa", Nazwisko = "Mocarz", NumerTelefonu = 7272025, Wiek = 11 }
};
```


Przykład – LINQ - Pobieranie danych (filtrowanie i sortowanie)

```
var listaOsobPełnoletnich = from osoba in listaOsob
  where osoba.Wiek >= 18
  orderby osoba.Wiek
  select osoba;

List<Osoba> podlista = listaOsobPełnoletnich.ToList<Osoba>();

Console.WriteLine("Lista osób pełnoletnich:");
foreach (var osoba in listaOsobPełnoletnich)
  Console.WriteLine(osoba.Imię + " " + osoba.Nazwisko + " (" + osoba.Wiek + ")");

Console.WriteLine("Wiek najstarszej osoby: " +
  listaOsobPełnoletnich.Max(osoba => osoba.Wiek));
Console.WriteLine("Średni wiek osób pełnoletnich: " +
  listaOsobPełnoletnich.Average(osoba => osoba.Wiek));
Console.WriteLine("Suma lat osób pełnoletnich: " +
  listaOsobPełnoletnich.Sum(osoba => osoba.Wiek));
```

Przykład – LINQ - Grupowanie danych w zapytaniu

```
var grupyOsobOTymSamymNazwisku = from osoba in listaOsob
    group osoba by osoba.Nazwisko into grupa
    select grupa;

Console.WriteLine("Lista osób pogrupowanych według nazwisk:");

foreach (var grupa in grupyOsobOTymSamymNazwisku)
{
  Console.WriteLine("Grupa osób o nazwisku " + grupa.Key);
  foreach (Osoba osoba in grupa)   Console.WriteLine(osoba.Imię+ " " + osoba.Nazwisko);
}
```


Przykład – LINQ - Łączenie zbiorów danych

```
// Dwa pierwsze zapytania tworzą dwa zbiory, które łączone są w trzecim zapytaniu

var listaOsobPelnoletnich = from osoba in listaOsob
    where osoba.Wiek>=18
    orderby osoba.Wiek
    select new { osoba.Imię, osoba.Nazwisko, osoba.Wiek };

var listaKobiet = from osoba in listaOsob
   where osoba.Imię.EndsWith("a")
   select new { osoba.Imię, osoba.Nazwisko, osoba.Wiek };

var listaPelnoletnich_I_Kobiet = listaOsobPelnoletnich.Concat(listaKobiet);
```


Przykład – LINQ - Łączenie zbiorów danych

```
// Jeżeli chcemy się pozbyć zdublowanych elementów,
// należy użyć metody rozszerzającej Distinct:
var listaPelnoletnich_I_Kobiet = listaOsobPelnoletnich.Concat(listaKobiet).Distinct();

// Ten sam efekt, tj. sumę mnogościową z wyłączeniem powtarzających się elementów,
// można uzyskać, tworząc unię kolekcji za pomocą metody rozszerzającej Union:
var listaPelnoletnich_I_Kobiet = listaOsobPelnoletnich.Union(listaKobiet);

// Oprócz sumy mnogościowej można zdefiniować również iloczyn dwóch zbiorów
// (część wspólną). Służy do tego metoda rozszerzająca Intersect:
var listaKobietPelnoletnich = listaOsobPelnoletnich.Intersect(listaKobiet);

// Łatwo się domyślić, że możliwe jest również uzyskanie różnicy zbiorów.
// Dla przykładu znajdźmy listę osób pełnoletnich niebędących kobietami:
var listaPelnoletnichNiekobiet = listaOsobPelnoletnich.Except(listaKobiet);
```


Przykład – LINQ - Łączenie zbiorów danych z różnych źródeł

```
// Na potrzeby ilustracji tworzymy 2 kolekcje z listy listaOsob:
var listaTelefonów = from osoba in listaOsob
    select new {osoba.Id, osoba.NumerTelefonu};
var listaPersonaliów = from osoba in listaOsob
    select new {osoba.Id, osoba.Imię, osoba.Nazwisko};

var listaPersonaliówZTelefonami = from telefon in listaTelefonów
  join personalia in listaPersonaliów
  on telefon.Id equals personalia.Id
  select new
  {
    telefon.Id,
    personalia.Imię,
    personalia.Nazwisko,
    telefon.NumerTelefonu
};
```


Przykład – LINQ - Możliwość modyfikacji danych źródła

```
// Jeżeli w wyniku zapytania LINQ utworzymy listę osób pełnoletnich:
var ListaOsobPelnoletnich = from osoba in listaOsob
   where osoba.Wiek >= 18
   orderby osoba.Wiek
   select osoba;

// to dane zgromadzone w nowej kolekcji listaOsobPelnoletnich nie są kopiowane.
// Warunkiem jest jednak, że elementy kolekcji-źródła, czyli w powyższym przykładzie
// klasa Osoba jest typu referencyjnego, czyli jest właśnie klasą, a nie strukturą. Do nowej
// kolekcji dołączane są referencje z oryginalnego zbioru. To oznacza, że dane można
// modyfikować, a zmiany będą widoczne także w oryginalnej kolekcji.

Osoba pierwszyNaLiscie = nowaListaOsobPelnoletnich.First<Osoba>();
pierwszyNaLiscie.Imię = "Karol";
pierwszyNaLiscie.Nazwisko = "Bartnicki";
pierwszyNaLiscie.Wiek = 31;
// Po tej zmianie wyświetlmy dane z oryginalnego źródła (tj. listaOsob), a przekonamy
// się, że Witolda Mocarza (26) zastąpił Karol Bartnicki (31).
```


Przykład – LINQ - Możliwość modyfikacji danych źródła

```
// W przypadku LINQ to Objects polecenie umieszczone za operatorem select można
// zmienić w taki sposób, aby dane były kopiowane do nowych obiektów, tj.

var nowaListaOsobPelnoletnich = from osoba in listaOsob
    where osoba.Wiek >= 18
   orderby osoba.Wiek
   select new Osoba { Id = osoba.Id, Imię = osoba.Imię,
                      Nazwisko = osoba.Nazwisko,
                      NumerTelefonu = osoba.NumerTelefonu,
                      Wiek = osoba.Wiek };


// Wówczas cała Nasza filozofia bierze w łeb i edycja kolekcji będącej wynikiem zapytania
// nie wpływa na zawartość oryginału. ☺
```

Przykład – LINQ - Przenoszenie danych z kolekcji do pliku XML

```
private void Przenoszenie()
{
    XDocument xml = new XDocument(
            new XDeclaration("1.0","utf-8","yes"),
            new XElement("ListaOsob",
                    from osoba in listaOsob
                    orderby osoba.Wiek
                    select new XElement("Osoba",
                            new XAttribute("Id", osoba.Id),
                            new XElement("Imię", osoba.Imię),
                            new XElement("Nazwisko", osoba.Nazwisko),
                            new XElement("NumerTelefonu", osoba.NumerTelefonu),
                            new XElement("Wiek", osoba.Wiek)
                    )
            )
    );
xml.Save("ListaOsob.xml");
}
```


Przykład – LINQ - Tworzenie kolekcji na bazie danych z pliku XML

```
private void Pobieranie1()
{
    XDocument xml = XDocument.Load("ListaOsob.xml");
    IEnumerable<Osoba> listaOsobPelnoletnich =
            from osoba in xml.Descendants("Osoba")
            select
                    new Osoba() {
                            Id = int.Parse(osoba.Attribute("Id").Value),
                            Imię = osoba.Element("Imię").Value,
                            Nazwisko = osoba.Element("Nazwisko").Value,
                            NumerTelefonu = int.Parse(osoba.Element("NumerTelefonu").Value),
                            Wiek = int.Parse(osoba.Element("Wiek").Value) };

    string s = "Lista osób pełnoletnich:\n";
    foreach (Osoba osoba in listaOsobPelnoletnich) s += osoba.Imię + " " + osoba.Nazwisko
            + " (" + osoba.Wiek + ")\n";
    MessageBox.Show(s);
}
```


Przykład – LINQ - Budowanie kolekcji łańcuchów na podstawie informacji z pliku XML

```
private void Pobieranie2(object sender, EventArgs e)
{
    XDocument xml = XDocument.Load("ListaOsob.xml");
    IEnumerable<string> listaOsobPelnoletnich =
            from osoba in xml.Descendants("Osoba")
            where int.Parse(osoba.Element("Wiek").Value) >= 18
            orderby osoba.Element("Imię").Value
            select osoba.Element("Imię").Value+" "+osoba.Element("Nazwisko").Value;

string s = "Lista osób pełnoletnich:\n";

foreach (string personalia in listaOsobPelnoletnich) s += personalia + "\n";

MessageBox.Show(s);
}
```


TEST

Korzystanie z list

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Diagnostics;
namespace ConsoleApplication_Kolekcje
{
   class Program
   {
      static void Main(string[] args)
      {

         List<int> MojaListaTest = new List<int>();

         MojaListaTest.Add(1);
         MojaListaTest.Add(2);
         MojaListaTest.Add("kot");

         int zmienna = (int)MojaListaTest[1];

         Debug.WriteLine(zmienna);
}}}
```

Co wyświetli się w konsoli (Debug)?

Czy coś tu jest niepotrzebne?

Korzystanie z list

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Diagnostics;
namespace ConsoleApplication_Kolekcje
{
   class Program
   {
      static void Main(string[] args)
      {

         List<int> MojaListaTest = new List<int>();

         MojaListaTest.Add(1);
         MojaListaTest.Add(2);
         MojaListaTest.Add("kot");

         int zmienna = (int)MojaListaTest[1];

         Debug.WriteLine(zmienna);
}}}
```

Co wyświetli się w konsoli (Debug)?

Czy coś tu jest niepotrzebne?

```
2
```

Korzystanie z list

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Diagnostics;
namespace ConsoleApplication_Kolekcje
{
    class Program
    {
        static void Main(string[] args)
        {

            Queue<string> MojaListaTest = new Queue<string>();

            MojaListaTest.Enqueue("jajko");
            MojaListaTest.Enqueue("kura");

            Debug.WriteLine(MojaListaTest.Dequeue());
            Debug.WriteLine(MojaListaTest.Dequeue());

}}}
```

Dequeue - removes and returns the object at the beginning of the Queue<T>.

Enqueue - adds an object to the end of the Queue<T>.

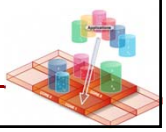
Co będzie pierwsze, jajko czy kura?

```
jajko
kura
```


ASYNCHRONICZNOŚĆ WBUDOWANA W JĘZYK

Wielozadaniowość

- Zastosowanie wielowątkowości ma na celu zwiększenie wydajności i skrócenia czasu reakcji dla wszystkich typów aplikacji.
- Wątki (threads) – wydzielone jednostki zadaniowe tworzone poprzez proces macierzysty w celu wykonania określonych zadań.
- Pozwalają wykonywać pewne długotrwałe operacje w tle, podczas gdy operacje pracujące w pierwszym planie pozostają w stanie gotowości.
- *Nieodpowiednie stosowanie wielowątkowości może jednak zaszkodzić…*

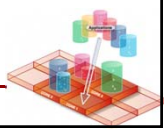


Wielozadaniowość

Bez względu jakie zadanie wykonujemy, w środowisku wielowątkowym zawsze mamy do czynienia z klasą **Thread** w takiej lub innej postaci.

Zasadniczo z tworzeniem poprawnej aplikacji wielowątkowej związane są dwa pojęcia:

- **Harmonogram wątków** – należy wiedzieć, jak uruchomić, zatrzymać, zawiesić i zakończyć wątki w swojej aplikacji
- **Rywalizacja o zasoby między wątkami**



Synchronizacja wątków i ich walka o zasoby

- Wszystkie obecnie wersje Microsoft Windows wykorzystują tzw. **wielozadaniowość z wywłaszczeniem**.
- Oznacza to, że aktualnie pracujący wątek może zostać przerwany (wywłaszczony), aby umożliwić wykonywanie się innemu wątkowi.
- Rozwiązanie takie znacznie redukuje ilość sytuacji, w których system się zawiesza lub wstrzymuje pracę z powodu nieodpowiedniego zachowania się jakiejś aplikacji.
- Wadą wielozadaniowości z wywłaszczeniem jest to, że musimy się orientować, jakie są tego konsekwencje.
- **Kluczową sprawą jest synchronizacja.**

Czym dysponujemy na platformie .NET

MOŻLIWOŚCI

Klasy i metody

W ramach biblioteki wątków rdzenia .NET Framework dostępne są różnorakie sposoby ułatwiające zarządzanie rywalizacją o współdzielone zasoby w aplikacji wielowątkowej, jak i kwestiami czasu i synchronizacji, np. klasy:

- Mutex,
- Monitor,
- Semaphore,
- AutoResetEvent,
- WaitHandle,
- słowo kluczowe lock itd.

Klasy i metody

Klasy udostępniają metody, które pozwalają na asynchroniczne wywołania metod, np. klasa SqlCommand:

- BeginExecuteNonQuery
- EndExecuteNonQuery
- BeginExecuteReader
- EndExecuteReader

- Powyższe metody asynchroniczne wykorzystują referencje do obiektu, z klasy, która implementuje interfejs **IAsyncResult**.
- Niektóre z metod przyjmują w postaci parametru egzemplarz klasy **AsyncCallback**.
- Klasa ta umożliwia wskazywanie metod, które mają być wykonywane po zakończeniu przetwarzania asynchronicznego zadania.

Klasy i metody

Dla egzemplarzy wspomnianej klasy SqlCommand znajdziemy również asynchroniczne odpowiedniki „klasycznych" metod:

- dla ExecuteNonQuery → ExecuteNonQueryAsync
- dla ExecuteReader → ExecuteReaderAsync

Powyższe metody asynchroniczne zwracają typ Task<T> i są określane jako awaitable („poczekalskie" ☺).

Komponenty

Można wykorzystać również komponent **BackgroundWorker**

- umożliwia asynchroniczne wywołanie (.RunWorkerAsync) metody zdarzeniowej (.DoWork),
- a jednocześnie umożliwia przerywanie metody (.CancelAsync),
- raportowanie jej postępu (.ProgressChanged)
- i obsługę jej zakończenia (.RunWorkerCompleted).

Co wybrać???

SPOSOBY PRZETWARZANIA

Sposoby przetwarzania

Asynchroniczne zadania można przetwarzać następująco:

- Uruchomienie asynchronicznego procesu i ciągłe **sprawdzanie obiektu IAsyncResult** w celu zbadania, czy zadanie zostało zakończone

- **Skojarzenie procesu z tzw. uchwytem** (Threading.Handle).

Rozwiązanie to umożliwia uruchomienie dowolne wybranych procesów asynchronicznych – następnie można poczekać (np. metoda WHandle.WaitAll()) , jak wszystkie lub część się zakończy w celu dokończenia głównego zadania.

Ta metody wymaga dokładnego zaplanowania harmonogramu.

Jednocześnie program (główny wątek) **zatrzyma się** podczas wspomnianego oczekiwania.

Sposoby przetwarzania

Asynchroniczne zadania można przetwarzać następująco:

- Przypisanie do uruchomionego procesu **funkcji zwrotnej** – pozwala to na równoległe wykonywanie innych zadań.

Po zakończeniu asynchronicznego zadania, jest wywoływana metoda zwrotna, która może „posprzątać" ☺ po procesie i powiadomić inne części aplikacji o wykonaniu zadania.

- Wykorzystać słowa kluczowe **async** i **await**.

Modele przetwarzania

Wskazane metody można przypisać do określonych modeli:

- Model programowania asynchronicznego - **Asynchronous Programming Model** (w skrócie **APM**) (oparty na parze metod: jedna służy do rozpoczęcia realizacji operacji Begin, a druga do pobrania danych po jej zakończeniu End)
- Model asynchroniczny bazujący na zdarzeniach - **Event-based Asynchronous Pattern** (w skrócie **EAP**). Klasa działająca według tego wzorca udostępnia metodę rozpoczynającą operację oraz odpowiadające jej zdarzenie, które jest zgłaszane po zakończeniu operacji.

Modele przetwarzania

Wskazane metody można przypisać do określonych modeli:

- Model **TPL** (**Task Parallel Library**) – oparty na metodzie Wait oraz właściwości Result.
- + rozszerzenie TPL (4.5 i C# 5.0), które dotyczy wyłącznie sposobów oczekiwania na zakończenie aktualnie realizowanych operacji asynchronicznych - nie określa natomiast, jak te operacje mają być rozpoczynane i kończone.
- Model ten jest określany angielskim terminem ***awaitable***, a jest on obsługiwany przez asynchroniczne możliwości języka C# (chodzi o słowa kluczowe **async** oraz **await**).

- Od wersji 4.0 wykorzystujemy głównie Task Parallel Library (TPL), która opiera się na koncepcji zadań, która reprezentuje operację asynchroniczną. W .NET Framework TPL jest preferowanym interfejsem API do pisania kodu wielowątkowego, asynchronicznego i równoległego.

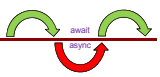
C# - async i await

ASYNCHRONICZNOŚĆ WBUDOWANA W JĘZYK

C# - async i await

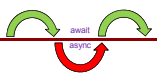
Zasady:

- Metody będą opatrzone modyfikatorem **async** oraz każde wywołanie asynchroniczne zostaje poprzedzone słówkiem **await**.
- Kod strukturalnie niczym nie różni się od kodu synchronicznego, nie wliczając w to użycia tych dwóch słów kluczowych.
- Wszystkie konstrukcje językowe (try, using) działają tutaj poprawnie.
- *Nazwa metody asynchronicznej, zgodnie z Konwencją, kończy się sufiksem "Async".*



C# - async i await

- Słowo kluczowe **async** niejako włącza słowo kluczowe **await** w metodzie.
- Nie powoduje ono w żaden sposób wykonania metody w wątku z puli.
- Jeśli uruchomimy taką metodę, wykona się ona po prostu synchronicznie.
- W momencie, gdy kompilator natrafi na słowo kluczowe **await**, to w tym miejscu **może** wprowadzić asynchroniczność.
- **await** przyjmuje jeden argument, którym jest tzn. typ **awaitable**.



C# - async i await

Podczas wykonania sprawdzany jest stan obiektu **awaitable**:

- Jeśli dostępne są już rezultaty, metoda wykonuje się dalej w sposób synchroniczny.
- Jeśli nie, to metoda asynchroniczna jest przerywana, a wywołanie wraca do metody wołającej metodę asynchroniczną – pozwalając jej się dalej wykonywać.
- Dzięki temu m.in. nasz interfejs się nie zawiesza.
- W momencie, gdy **awaitable** zasygnalizuje, że operacja już się zakończyła, wznawiane jest wykonanie metody asynchronicznej od ostatniego miejsca przystanku.

- *Czym jest to awaitable?*

C# - async i await

awaitable to obiekty, które mogą współpracować z async, tutaj tzw. *taski.*

Typ zwracany jest jednym z następujących typów:

- **Task<TResult>**: jeśli metoda zawiera instrukcję return, w której argument jest typu TResult.
- **Task**: kiedy nie chcemy nic zwrócić, ale chcemy mieć możliwość czekania („awaitowania")
- **void**: nie można nic zwrócić, ani na nic zaczekać.

Uwaga: (Task(Of T) -> Visual Basic .NET)

C# - async i await – bez asynchoniczności

```
int AccessTheWebSynch()
{
   HttpClient client = new HttpClient();

   string getString = client.GetStringAsync("http://msdn.microsoft.com").Result;

   DoIndependentWork(this.lblLenghtSynchr);

    string urlContents = getString;

    return urlContents.Length;
}

private void DoIndependentWork(Label lbl)
      {
         lbl.Text = "Working.....";
      }
```

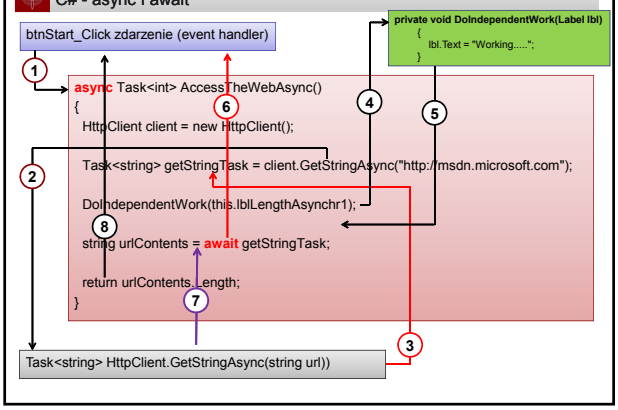

C# - async i await

```
async Task<int> AccessTheWebAsync1()
{
  HttpClient client = new HttpClient();

  Task<string> getStringTask = client.GetStringAsync("http://msdn.microsoft.com");

  DoIndependentWork(this.lblLengthAsynchr1);

  string urlContents = await getStringTask;

  return urlContents.Length;
}
```


C# - async i await

```
async Task<int> AccessTheWebAsync2()
{
  DoIndependentWork(this.lblLengthAsynchr2);

  HttpClient client = new HttpClient();

  string urlContents = await client.GetStringAsync("http://msdn.microsoft.com");

  return urlContents.Length;
}
```


C# - async i await



C# - async i await – kilka wątków

- Inną ciekawą możliwością, o której warto wspomnieć, jest komponowanie awaiterów.
- Można uruchomić kilka operacji na raz i czekać na skończenie wszystkich, bądź dowolnej z nich.

```
public async Task<int> GetGrandTotalPrice()
{
        Task<int> t1 = GetOrderTotalPrice(1);
        Task<int> t2 = GetOrderTotalPrice(2);
        await Task.WhenAll(t1, t2);
        return t1.Result + t2.Result;
}
```


C# - async i await – kilka wątków

```
public async void ProcessWriteMult()
{
   string folder = @"tempfolder\";
   List<Task> tasks = new List<Task>();
   List<FileStream> sourceStreams = new List<FileStream>();
   try
   {
      for (int index = 1; index <= 10; index++)
      {
         string text = "In file " + index.ToString() + "\r\n";
         string fileName = "thefile" + index.ToString("00") + ".txt";
         string filePath = folder + fileName;
         byte[] encodedText = Encoding.Unicode.GetBytes(text);

         FileStream sourceStream = new FileStream(filePath,FileMode.Append, FileAccess.Write,
               FileShare.None, bufferSize: 4096, useAsync: true);

         Task theTask = sourceStream.WriteAsync(encodedText, 0, encodedText.Length);
         sourceStreams.Add(sourceStream);

         tasks.Add(theTask);
      }

      await Task.WhenAll(tasks);
   }
   finally
   {
      foreach (FileStream sourceStream in sourceStreams)
      {
         sourceStream.Close();
      }
   }
}
```


C# - async i await – anulowanie

- Platforma .NET definiuje standardowy mechanizm służący do anulowania długotrwałych operacji. Operacje zapewniające możliwość anulowania pobierają argument typu **CancellationToken**.
- Jeśli ustawimy go w stan anulowania, to jeśli tylko pojawi się taka możliwość, operacja zostanie przerwana wcześniej, a nie będzie wykonywana do końca.
- Sam typ **CancellationToken** nie udostępnia żadnych metod służących do zainicjowania anulowania — anulowanie jest obsługiwane przez odrębny obiekt — **CancellationTokenSource**.
- *Uwaga: jeśli korzystamy z mechanizmu anulowania, trzeba być przygotowanym na to, że jego użycie nie da żadnego efektu.*

C# - async i await – anulowanie

```
CancellationTokenSource cts;

private async void btnStartAsynchCancel_Click
                                (object sender, EventArgs e)
{
 cts = new CancellationTokenSource();
 txtResults.Clear();
 try
  {
    cts.CancelAfter(5000);
    await AccessTheWebAsync(cts.Token);
    txtResults.Text += "\r\nDownloads succeeded.\r\n";
  }
  catch (OperationCanceledException)
  {
    txtResults.Text += "\r\nDownloads canceled.\r\n";
  }
  catch (Exception)
  {
    txtResults.Text += "\r\nDownloads failed.\r\n";
  }
 cts = null;
}

private void btnCancel_Click(object sender, EventArgs e)
{
    if (cts != null)
        cts.Cancel();
}
```

```
async Task AccessTheWebAsync(CancellationToken ct)
{
  HttpClient client = new HttpClient();
  List<string> urlList = SetUpURLList();
  foreach (var url in urlList)
  {
     HttpResponseMessage response = await
                                   client.GetAsync(url, ct);
     byte[] urlContents = await
                response.Content.ReadAsByteArrayAsync();
     txtResults.Text +=
                String.Format("\r\nLength of the downloaded
                   string: {0}.\r\n", urlContents.Length);
     }
  }

private List<string> SetUpURLList()
 {
   List<string> urls = new List<string>
   {
   "http://msdn.microsoft.com",
   "http://msdn.microsoft.com/apps/br211380.aspx",
   "http://msdn.microsoft.com/en-us/library/hh290136.aspx",
   };
 return urls;
}
```


PODSUMOWANIE

Podsumowanie

- Istnieje coraz więcej klas dających wsparcie programowaniu asynchronicznemu.

Obszar aplikacji	**Obsługa interfejsów API, która obejmuje metody asynchroniczne**
Dostęp do sieci Web	HttpClient
Praca z plikami	StorageFile, StreamWriter, StreamReader, XmlReader
Praca z obrazami	MediaCapture, BitmapEncoder, BitmapDecoder
Programowanie WCF	Operacje synchroniczne i asynchroniczne
Bazy danych	SqlCommand

Podsumowanie

- Użycie przedstawionego rozszerzenia językowego nie jest oczywiście limitowane tylko do celów interfejsowych.
- Możemy go także używać do asynchronicznego wczytywanie czy zapisywanie plików oraz do tych wszystkich sytuacji, gdzie pojawia się potrzeba wielowątkowości.
- **async** i **await** znacząco upraszcza obsługę asynchroniczności w interfejsie użytkownika.
- Kod staje się prosty, czytelny i co ważne zarządzalny.

TEST

Test

```
using System.Diagnostics;
using System.Threading.Tasks;
namespace ConsoleApplicationTest
{
   class Program
   {
      static void Main(string[] args)
      {
         Task theTask = ProcessAsync(); // A
         int x = 0;  // B
         theTask.Wait(); // C - Waits for the task to complete execution
      }

      static async Task ProcessAsync()
      {
         var result = await DoSomethingAsync();  // D
         int y = 0;  // E
      }

      static async Task<int> DoSomethingAsync()
      {
         Debug.WriteLine("before"); // F
         await Task.Delay(10000); // G
         Debug.WriteLine("after"); // H
         return 5; // I
} }}
```

Kod będzie się wykonywał w kolejności:
a) A D E B C…
b) A B C D E…
c) A D F G H …
d) A D F G B …

Test

```
using System.Diagnostics;
using System.Threading.Tasks;
namespace ConsoleApplicationTest
{
   class Program
   {
      static void Main(string[] args)
      {
         Task theTask = ProcessAsync(); // A {1}
         int x = 0;  // B {5}
         theTask.Wait(); // C {6} - Waits for the task to complete execution
      }

      static async Task ProcessAsync()
      {
         var result = await DoSomethingAsync();  // D {2}
         int y = 0;  // E {9}
      }

      static async Task<int> DoSomethingAsync()
      {
         Debug.WriteLine("before"); // F {3}
         await Task.Delay(10000); // G {4}
         Debug.WriteLine("after"); // H {7}
         return 5; // I {8}
} }}
```

Kod będzie się wykonywał w kolejności:
a) A D E B …
b) A B C D …
c) A D F G H …
d) A D F G B …

a) A D F G B C H I E

PODSUMOWANIE

Podsumowanie

Metody asynchroniczne stosuje się:

- gdy brakuje wolnych wątków;
- gdy aplikacja będzie szybciej/lepiej działać przy użyciu metod asynchronicznych;
- gdy operacja opiera się w głównej mierze na operacjach I/O (zapis/odczyt) na dysku lub na operacjach sieciowych, które mogą zablokować wątek poprzez wolne działanie;
- chce się zaimplementować mechanizm zatrzymywania lub przerywania długich żądań (np. przesyłanie plików).

Podsumowanie

Słowa kluczowe — async, await, Task

- **async** — tym słowem kluczowym oznacza się metodę asynchroniczną. Dzięki temu kompilator wie, że będzie to metoda asynchroniczna. Metoda asynchroniczna według konwencji powinna mieć na końcu nazwy słowo **async**.
- **await** — informuje o metodzie, która czeka na dane zwracane w sposób asynchroniczny. Słowo **await** może być użyte tylko w metodzie asynchronicznej oznaczonej słowem **async**. Dzięki słowu **await** kompilator nie blokuje wątku ani wykonywania innych części kodu na czas pobierania danych.
- **Task** — jest to typ danych zwracanych przez metodę asynchroniczną. Typ Task<typ_zwracany> informuje kompilator o danych zwracanych asynchronicznie.

Podsumowanie

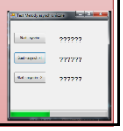
- Klasa Task stanowi jednak coś więcej niż jedynie opakowanie dla puli wątków. Zarówno jej, jak i powiązanych z nią typów tworzących bibliotekę TPL (Task Parallel Library) można z powodzeniem używać w znacznie większej liczbie sytuacji.
- Biblioteka TPL została wprowadzona w .NET 4.0, jednak w kolejnej wersji platformy - 4.5 - zyskała znacząco większe znaczenie, gdyż asynchroniczne mechanizmy języka wprowadzone w C# 5.0 są w stanie operować bezpośrednio na obiektach zadań.
- Z tego powodu w najnowszej wersji .NET Framework bardzo wiele API zostało rozbudowanych o obsługę operacji asynchronicznych bazujących na wykorzystaniu zdarzeń.

Kluczowymi elementami biblioteki TPL są dwie klasy:

- Task oraz dziedzicząca po niej klasa Task<T>.
- Klasa bazowa Task reprezentuje pewną operację, której wykonanie może zająć trochę czasu.
- Klasa Task<T> stanowi rozszerzenie reprezentujące operację, która po wykonaniu zwraca pewien wynik (typu T).
- (Nieogólna klasa Task nie zwraca żadnego wyniku. Stanowi ona asynchroniczny odpowiednik typu void).
- Warto zwrócić uwagę, że nie są to pojęcia, które muszą być powiązane wątkami.

Asynch

```
using System;
using System.Net.Http;
using System.Threading.Tasks;
using System.Windows.Forms;
namespace WindowsForms_Asyn
{
  public partial class frmTest : Form
  {
    public frmTest()
    {
      InitializeComponent();
    }
    private void DoIndependentWork(Label lbl)
    {
      lbl.Text = "Working....";
    }
    int AccessTheWebSynch()
    {
      HttpClient client = new HttpClient();

      string getString = client.GetStringAsync("http://msdn.microsoft.com").Result;

      DoIndependentWork(this.lblLenghtSynchr);

      string urlContents = getString;

      return urlContents.Length;
    }
    async Task<int> AccessTheWebAsync1()
    {
      HttpClient client = new HttpClient();

      Task<string> getStringTask = client.GetStringAsync("http://msdn.microsoft.com");

      DoIndependentWork(this.lblLengthAsynchr1);

      string urlContents = await getStringTask;

      return urlContents.Length;
    }
    async Task<int> AccessTheWebAsync2()
    {
      DoIndependentWork(this.lblLengthAsynchr2);

      HttpClient client = new HttpClient();

      string urlContents = await client.GetStringAsync("http://msdn.microsoft.com");

      return urlContents.Length;
    }

    private void btnStartSynchr_Click(object sender, EventArgs e)
    {
      lblLenghtSynchr.Text = (AccessTheWebSynch()).ToString();
    }

    private async void btnStart1_Click(object sender, EventArgs e)
    {
      lblLengthAsynchr1.Text = (await AccessTheWebAsync1()).ToString();
    }

    private async void btnStart2_Click(object sender, EventArgs e)
    {
      lblLengthAsynchr2.Text = (await AccessTheWebAsync2()).ToString();
    }

    private void tmrProgress_Tick(object sender, EventArgs e)
    {
      if (++prbProgress.Value == prbProgress.Maximum)
        prbProgress.Value = 0;
    }
  }
}
```



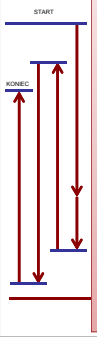
TEST

Test

```
public partial class MainWindow : Window
{
  // . . .
  private async void startButton_Click(object sender, RoutedEventArgs e)
  {
    // A
    Task<int> getLengthTask = AccessTheWebAsync();

    // B
    int contentLength = await getLengthTask;

    // C
    resultsTextBox.Text +=
      String.Format("\r\nLength of the downloaded string: {0}.\r\n", contentLength);
  }

  async Task<int> AccessTheWebAsync()
  {
    // D
    HttpClient client = new HttpClient();
    Task<string> getStringTask =
      client.GetStringAsync("http://msdn.microsoft.com");

    // E
    string urlContents = await getStringTask;

    // F
    return urlContents.Length;
  }
}
```

W jakiej kolejności będzie wykonywał się kod po wywołaniu zdażenia - startButton_Click?

Test




```
public partial class MainWindow : Window
{
    // . . .
    private async void startButton_Click(object sender, RoutedEventArgs e)
    {
        // A – {1}
        Task<int> getLengthTask = AccessTheWebAsync();

        // B – {4}
        int contentLength = await getLengthTask;

        // C – {6}
        resultsTextBox.Text +=
            String.Format("\r\nLength of the downloaded string: {0}.\r\n", contentLength);
    }


    async Task<int> AccessTheWebAsync()
    {
        // D – {2}
        HttpClient client = new HttpClient();
        Task<string> getStringTask =
            client.GetStringAsync("http://msdn.microsoft.com");

        // E – {3}
        string urlContents = await getStringTask;

        // F – {5}
        return urlContents.Length;
    }
}
```

W jakiej kolejności będzie wykonywał się kod po wywołaniu zdażenia - startButton_Click?

KONIEC