

MS .NET i C#

wykład
2009/2010

Literatura

- E. Gunnerson “Programowanie w języku C#”, Mikom
- K. Burton “.NET CLR – księga eksperta”, Helion
- MSDN Library

“Zadania” MS .NET

- Jednolite środowisko obiektowe niezależnie od lokalności/zdalności kodu
- Unikanie konfliktu wersji
- Bezpieczeństwo wykonywania kodu
- Lepsza wydajność niż języki skryptowe
- Ujednolicenie technik programistycznych (lokalnie/zdalnie)
- Łatwość integracji

Czym jest MS .NET?

- Zarządzane (*ang.* managed) środowisko stanowiące warstwę pośrednią między programem a systemem operacyjnym
- Zalety:
 - większe bezpieczeństwo,
 - obsługa wielu języków,
 - skalowalność,
 - łatwiejsze wdrażanie i zarządzanie,
 - uproszczenie wielu aspektów tworzenia aplikacji
- Wady:
 - szybkość działania,
 - zasobochłonność,
 - dyskusyjne wsparcia na innych platformach (Mono, Grasshopper)

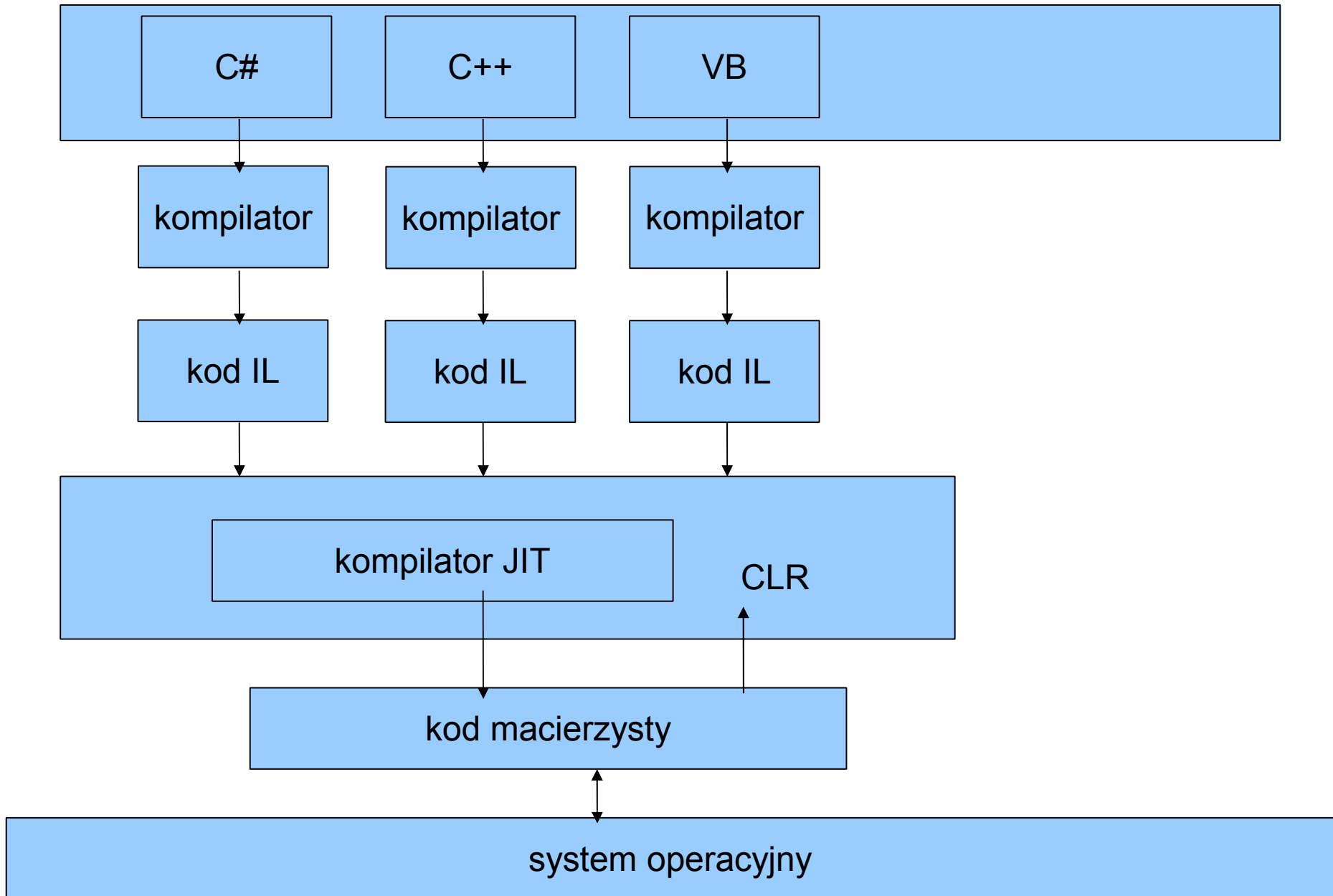
Składniki MS .NET

- Common Language Runtime
- .NET Framework class library

Pozycja C#

- C# to język opracowany dla efektywnego wykorzystania możliwości platformy .NET
- Wykorzystuje konstrukcje zbliżone do C++ i Javy

Koncepcja .NET



Bezpieczeństwo

- zarządzanie pamięcią (np. kwestie odwołania do niezaalokowanej pamięci)
- bezpieczeństwo typów (np. kwestie konwersji do typów niekompatybilnych)
- inicjalizacja zmiennych, kontrola stosu, nadzór nad wskaźnikami do funkcji
- weryfikacja kodu MSIL przed wykonaniem (dostęp do pamięci, zgodność typów)
- kontrola uprawnień na poziomie kodu, a nie użytkownika

Wielojęzyczność

- obsługa C#, C++ (wersja C++/CLI), VB, J#
- obecność elementów standaryzujących konstrukcje języków: Common Language Specification, Common Type System
- programy tłumaczone na język MS Intermediate Language
- możliwość współpracy między modułami napisanymi w różnych językach

Skalowalność

- możliwość automatycznego dostosowania środowiska do usług sprzętu na którym jest uruchomione

Wdrażanie i zarządzanie

- możliwe koegzystowanie różnych wersji komponentów (brak problemu *DLL hell*)
- wbudowane mechanizmy bezpiecznej identyfikacji modułów
- metadane i samoopisujący się kod - moduły same zawierają informacje o wymaganych komponentach, CLR zajmuje się sprawdzeniem czy wszystko jest dostępne

Prostsze tworzenie aplikacji

- Wiele przydatnych predefiniowanych obiektów
- Automatyczne oczyszczanie pamięci
- Usprawniona obsługa wyjątków
- Dziedziczenie wszystkich obiektów od klasy *Object*
- Dostęp do informacji o typie podczas pracy aplikacji
- Nowe instrukcje i konstrukcje

Szybkość i zasobochłonność

- Kompilacja JIT a generacja kodu w chwili instalacji
- Moduł CLR
- Odzyskiwanie pamięci
- Operacje opakowywania i rozpakowywania (*ang.* boxing/unboxing)

Inne platformy

- Problemy z przeniesieniem kodu na inne systemy operacyjne...
- Możliwe rozwiązania:
 - Natywna implementacja .NET na inne platformy (Mono)
 - Warstwa pośrednicząca, np. tłumacząca kod IL na Java bytecode
 - Kwestia serwerów (np. mod_mono dla Apache)

Więcej o zarządzaniu pamięcią

- Zarządzane sterty: dla obiektów 'zwykłych' i 'dużych'. Dla obiektów dużych nie występuje kompaktowanie
- Zarządzanie oparte na następujących obserwacjach:
 - nowe obiekty żyją krócej
 - nowe obiekty są ze sobą często powiązane
 - kompaktowanie fragmentu sterty jest szybsze niż całości

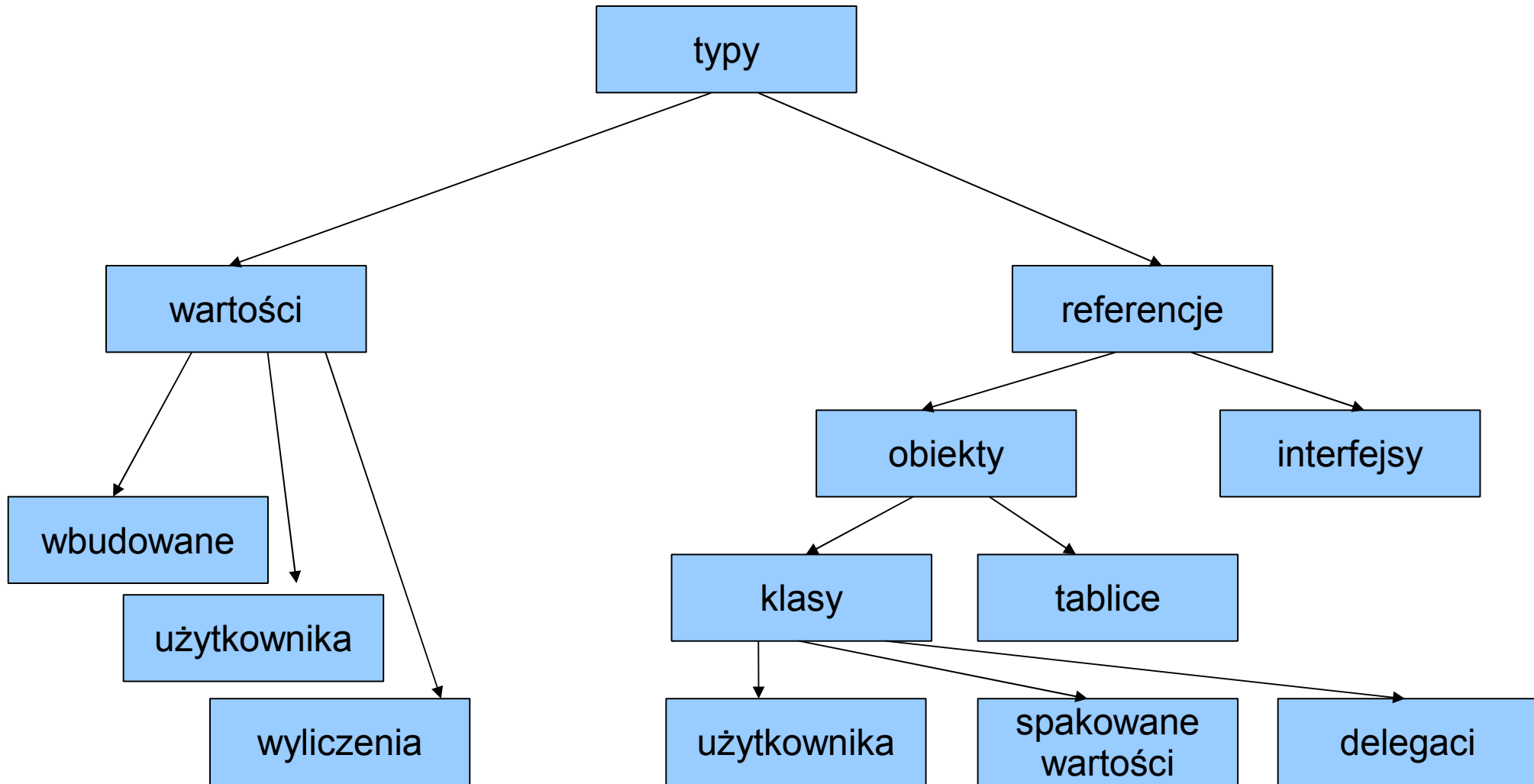
Więcej o zarządzaniu pamięcią

- Obiekty są dzielone na generacje: 0 (najnowsze), 1 i 2
- Zwalnianie pamięci i kompaktowanie odbywa się gdy nie ma miejsca na obiekty generacji 0. Zwalniane są obiekty generacji 0, jeśli nie zwolni to wystarczająco pamięci, przetwarzane są obiekty kolejnych generacji
- Obiekty które przetrwały są przesuwane do generacji o 1 wyższej

Więcej o metadanych

- Znajdują się w jednym pliku, obok kodu w MSIL
- Opisują:
 - moduł (nazwa, wersja, lokalizacja, eksportowane typy, moduły konieczne do pracy, wymagane zezwolenia do wykonania),
 - typy (nazwa, widoczność, klasa bazowa, interfejsy, elementy składowe klasy)
 - atrybuty - dodatkowe informacje wpływające na zachowanie programu przy wykonywaniu

Więcej o Common Type System



Wartości a referencje

- Niewielkie
- Alokowane na stosie
- Kopiowane przy przekazywaniu do funkcji
- Porównywane bit po bicie
- Definiowane słowem kluczowym `struct`
- Wywodzi się z `System.ValueType`
- Mogą być duże
- Alokowane na sterckie
- Do funkcji przekazywany jest wskaźnik
- Porównywane adresy
- Definiowane słowem kluczowym `class`
- Wywodzi się z `System.Object`

Wbudowane typy wartości

- Wartości logiczne
- Znaki alfanumeryczne (Unicode)
- Liczby całkowite (z i bez znaku)
- Liczby zmiennoprzecinkowe (32 i 64bit)
- Liczby zależne od komputera

Więcej o modułach (assembly)

- Są elementem któremu przyznawane są uprawnienia
- Określają przestrzeń nazw dla typów
- Są elementem dla którego określa się wersję
- Są jednostkami na które można podzielić program i używać w zależności od aktualnych potrzeb
- Zawierają: manifest modułu (konieczny), metadane, kod MSIL, zasoby.
- Mogą składać się z jednego bądź więcej plików

Zalety modułów

- Rozwiązanie problemu konfliktów wersji: umożliwiają określenie wymogów dotyczących wersji oraz pozwalają na jednoczesne instalowanie różnych wersji tego samego komponentu

Dobre praktyki - nazwy

- Nie wszystkie języki rozpoznają wielkość liter
- Należy stosować konwencję Pascal, z wyjątkiem nazw zmiennych (parametry funkcji pola klas, zmienne lokalne), gdzie stosuje się konwencję Camel
- Nie należy stosować notacji węgierskiej
- Biblioteki DLL odpowiadające modułom należy nazywać Firma.Komponent.dll
- Przestrzenie nazw należy nazywać Firma.Produkt.ElementProduktu

Dobre praktyki - nazwy

- Nie poprzedzać nazw klas przedrostkiem (wyjątek - nazw interfejsu)
- Zaleca się kończyć nawet klasy nazwą klasy bazowej

Istotne konwencje C#

- Brak wskaźników
- Odwołania do pól i funkcji składowych zawsze przy pomocy operatora `.` (kropka)
- Używa przestrzeni nazw zamiast plików nagłówkowych
- Brak elementów globalnych
- Funkcja `Main` jest statyczną składową klasy

Modyfikacja wartości przekazywanych – słowa `ref` i `out`

- Umożliwiają modyfikację wartości zmiennych przekazywanych do funkcji
- `out` umożliwia przekazanie do funkcji wartości niezainicjalizowanych

Modyfikacja wartości przekazywanych – przykład 1

```
using System;
public class MyClass
{
    public static void TestRef(ref char i)
    {
        i = 'b';
    }

    public static void TestNoRef(char i)
    {
        i = 'c';
    }

    public static void Main()
    {
        char i = 'a'; // inicjalizacja
        TestRef(ref i); // przekazanie jako ref
        TestNoRef(i);
    }
}
```

Modyfikacja wartości przekazywanych – przykład 2

```
using System;
public class MyClass
{
    public static int TestOut(out char i)
    {
        i = 'b';
        return -1;
    }

    public static void Main()
    {
        char i; //nie ma potrzeby inicjalizacji
        Console.WriteLine(TestOut(out i));
        Console.WriteLine(i);
    }
}
```

Typy wartości użytkownika – konstrukcja `struct`

- Przekazywane przez wartość
- Nie mogą brać udziału w procesie dziedziczenia
- Nie mogą mieć konstruktora domyślnego
- Pola inicjalizowane wartością 0 (tylko jeśli wartość nie jest tworzona operatorem `new`)
- Obiekty mogą być tworzone albo jako
`nazwa_typu nazwa_obiektu;`
albo
`nazwa_typu nazwa_obiektu = new
nazwa_typu (par1, par2, ..., parN);`

struct - przykład

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

    public int x;
    public int y;
}
```

```
Point pt1; //inicjalizowany zerami
Point pt2 = new Point(12, 28);
```

Typy wyliczeniowe – konstrukcja `enum`

- Oparte na typach całkowitych (domyślnie `int`, mogą być inne - oprócz `char`)
- Domyślnie rozpoczyna się od 0 i jest zwiększana dla kolejnych składników
- Mogą być stosowane jako flagi bitów
- Wywodzi się z `System.Enum`

enum – przykłady 1

```
enum LineStyle
{
    solid,
    dotted,
    dashed
}
enum LineStyle : byte
{
    solid = 0,
    dotted = 8,
    dashed = 16
}
enum LineStyle
{
    solid = 0,
    dotted = solid + 8,
    dashed = dotted * 2
}
```


enum – przykłady 2

[Flags]

```
enum LineStyle
{
    thick = 0x00000001,
    dotted = 0x00000002,
    dashed = 0x00000004
}
```

Klasy użytkownika – konstrukcja

`class`

- Umożliwia tworzenie złożonych typów
- Umożliwia dziedziczenie
- Nie ma dziedziczenia wielokrotnego, ale klasa może implementować wiele interfejsów
- Definicja klasy może być rozbita na wiele plików (słowo kluczowe `partial`)
- Obiekty mogą być tworzone tylko jako
`nazwa_typu nazwa_obiektu = new`
`nazwa_typu (par1, par2, ..., parN);`

Modyfikatory dostępu

- Na poziomie klasy:
 - `public`
 - `internal`
- Na poziomie składowej klasy
 - `public`
 - `internal protected`
 - `internal`
 - `protected`
 - `private`
- Dostępność na poziomie klasy ogranicza dostępności na poziomie składowych

Konstruktory

- Obiektu – wywoływane przy tworzeniu obiektu klasy
- Statyczne – służą do inicjalizacji klasy (a nie obiektu) zanim jakkolwiek obiekt klasy zostanie utworzony i zanim nastąpią odwołania do statycznych składowych

Konstruktory obiektu

- W przypadku niezdefiniowania konstruktora domyślnego, kompilator utworzy go. Pola inicjalizowane są wtedy wartością 0
- Konstruktor typu podstawowego może zostać wywołany przy użyciu słowa `base`
- Inny konstruktor tego samego typu może zostać wywołany przy użyciu słowa `this`
- Inicjalizację pól można przeprowadzić bądź w konstruktorze, bądź też przy ich deklaracji
- Prywatne – uniemożliwiają utworzenie obiektu klasy (jeśli nie ma konstruktorów publicznych)

Konstruktory obiektu - przykład

```
public Cylinder(double r, double h)
: base(r, h)
{
}
```

```
public Point(): this(0, 20)
{
}
```

```
public class MyClass
{
    public int counter = 100;
}
```

Konstruktory statyczne

- Nie mają modyfikatorów dostępu i argumentów
- Nie mogą być wywołane jawnie
- Użytkownik nie ma wpływu na to kiedy konstruktor zostanie wywołany w programie

Konstruktory statyczne - przykład

```
class MyClass
{
    // Static constructor:
    static MyClass()
    {
        Console.WriteLine("Static constructor");
    }

    public static void MyMethod()
    {
        Console.WriteLine("MyMethod invoked.");
    }
}
```


Destruktory

- Działają na innej zasadzie niż w C++, są wywoływane w momencie oczyszczania pamięci
- Odpowiadają metodzie `Finalize`
- „Ręczne” pisanie metody `Finalize` jest możliwe tylko w językach nie mających destruktorów
- Destruktor zawsze wywołuje metodę `Finalize` z klasy bazowej
- Nie ma określonego momentu ani kolejności wywołania metod `Finalize`, nie ma pewności że w ogóle zostaną wywołane
- W celu 'ręcznego zwolnienia zasobów' należy użyć metody `Dispose` (interfejs `IDisposable`)

Destruktory

```
~MyClass ()
{
    // Perform some cleanup operations here.
}

protected override void Finalize()
{
    try
    {
        // Perform some cleanup operations here.
    }
    finally
    {
        base.Finalize();
    }
}
```

Elementy statyczne – słowo static

- Metody (w tym konstruktory)
- Pola

Wywołanie wyłącznie na rzecz klasy, nie obiektu

Stałe

- Używając słowa `const` - wyłącznie typy wbudowane, zapisane jako literały – wartość musi być znana w momencie kompilacji. Ustawiane poprzez deklarację
- Używając słowa `readonly` – wartość może być ustawiona w konstruktorze lub poprzez deklarację i nie może być później zmieniona. Może mieć różne wartości w różnych konstruktorach

Stałe – przykład

```
class MyClass
{
    public int x;
    public readonly int y = 25; // w deklaracji
    public readonly int z;
    public const int v = 5;

    public MyClass()
    {
        z = 24; // w konstruktorze
    }

    public MyClass(int p1, int p2, int p3)
    {
        x = p1;
        y = p2;
        z = p3;
    }
}
```

Dziedziczenie 1

- Nie występuje dziedziczenie wielokrotne, ale można stosować tzw. interfejsy
- Funkcje wirtualne w klasach pochodnych przesłaniające funkcje w klasie bazowej muszą być poprzedzone słowem kluczowym `override` lub `new`
- Nie-wirtualne funkcje w klasach pochodnych przesłaniające funkcje w klasie bazowej muszą być poprzedzone słowem kluczowym `new`
- Klasy abstrakcyjne można tworzyć słowem kluczowym `abstract` (klasa abstrakcyjna nie musi mieć metod abstrakcyjnych, ale metody abstrakcyjne mogą występować tylko w klasie abstrakcyjnej)

Dziedziczenie 2

- Można uniemożliwić dziedziczenie po klasie słowem kluczowym `sealed`
- Można uniemożliwić przesłonięcie metody słowem kluczowym `sealed`
- Funkcje klasy bazowej mogą być wywoływane przy użyciu słowa kluczowego `base`

Dziedziczenie – przykład 1

```
class Square
{
    public double x;
    public Square(double x)
    {
        this.x = x;
    }
    public virtual double Area()
    {
        return x*x;
    }
}

class Cube: Square
{
    public Cube(double x): base(x) {}
    public override double Area()
    {
        return (6*(base.Area()));
    }
}
```


Dziedziczenie – przykład 2

```
class A
{
    public void F() {}
    public virtual void G() {}
}
class B: A
{
    new public void F() {}
    public override void G() {}
}
```

Dziedziczenie – przykład 3

```
using System;
class A
{
    public virtual void F() { Console.WriteLine("A.F"); }
}
class B: A
{
    public override void F() { Console.WriteLine("B.F"); }
}
class C: B
{
    new public virtual void F() { Console.WriteLine("C.F"); }
}
class D: C
{
    public override void F() { Console.WriteLine("D.F"); }
}
class Test
{
    static void Main() {
        D d = new D();
        A a = d;
        B b = d;
        C c = d;
        a.F();
        b.F();
        c.F();
        d.F();
    }
}
```

B.F
B.F
D.F
D.F

Dziedziczenie – przykład 4

```
abstract class A
{
    public abstract void F();
}
abstract class B: A
{
    public void G() {}
}
class C: B
{
    public override void F() {
        // actual implementation of F
    }
}
```

Dziedziczenie – przykład 5

```
using System;
class A
{
    public virtual void F() {}
    public virtual void G() {}
}
class B: A
{
    sealed override public void F() {
        Console.WriteLine("B.F");
    }
    override public void G() {
        Console.WriteLine("B.G");
    }
}
class C: B
{
    override public void G() {
        Console.WriteLine("C.G");
    }
}
```

Dziedziczenie – przykład 6

```
sealed class MyClass
{
    MyClass () {}
}

//błąd
class MyNewClass : MyClass
{
}
```

Interfejsy

- Koncepcja zbliżona do klasy abstrakcyjnej której wszystkie składowe są typu `abstract`
- Klasa może implementować więcej niż jeden interfejs
- Również struktury mają możliwość implementowania interfejsu
- Interfejsy można opierać na innych interfejsach
- Klasa implementująca interfejs musi zdefiniować wszystkie funkcje tego interfejsu

Interfejsy – przykład 1

```
public class DiagramObject
{
    public DiagramObject() {}
}
```

```
interface IScalable
{
    void ScaleX(float factor);
    void ScaleY(float factor);
}
```

```
public class TextObject: DiagramObject, IScalable
{
    public void ScaleX(float factor)
    {
    }
    public void ScaleY(float factor)
    {
    }
}
```

Interfejsy – przykład 2

```
TextObject text = new TextObject("bla");  
text.ScaleX(0.5F)  
IScalable scalable = (IScalable) text;  
scalable.ScaleY(0.5F)
```


Interfejsy – przykład 3

```
interface IList
{
    int Count { get; set; }
}
interface ICounter
{
    void Count(int i);
}
interface IListCounter: IList, ICounter {}
class C
{
    void Test(IListCounter x) {
        x.Count(1); // // ?
        x.Count = 1; // ?
        ((IList)x).Count = 1; // ?
        ((ICounter)x).Count(1); // ?
    }
}
```

Jawna implementacja interfejsu

- W przypadku gdy dwa interfejsy deklarują funkcje o tej samej nazwie, a ich implementacja ma być rozdzielona
- W przypadku potrzeby ukrycia implementacji interfejsu
- Nie może mieć modyfikatorów dostępu

Jawna implementacja interfejsu – przykład 1

```
interface IFoo
{
    void Execute();
}

interface IBar
{
    void Execute();
}

class Tester : IFoo, IBar
{
    public void Execute() {}
    void IFoo.Execute()
    {
        //IFoo code
    }
    void IBar.Execute()
    {
        //IBar code
    }
}
```

Jawna implementacja interfejsu – przykład 2

```
Tester tester = new Tester();
```

```
tester.Execute() //błąd
```

```
IFoo iFoo = (IFoo)tester;  
iFoo.Execute();
```

```
IBar iBar = (IBar)tester;  
iBar.Execute();
```

Rozpoznawanie typu obiektu

- operator `typeof` zwraca typ obiektu
- funkcja `Object.GetType()` zwraca typ obiektu
- operator `is` sprawdza czy dany obiekt może zostać przekonwertowany do danego typu
- operator `as` dokonuje takiej konwersji, jeśli nie jest możliwa zwraca `null`

Rozpoznawanie typu obiektu – przykład 1

```
using System;
using System.Reflection;

public class MyClass
{
    public int intI;
    public void MyMeth()
    {
    }

    public static void Main()
    {
        Type t1 = typeof(MyClass);

        MyClass mc = new MyClass();
        Type t2 = mc.GetType();
    }
}
```

Rozpoznawanie typu obiektu – przykład 2

```
class Class1
{
class Class2
{

public class IsTest
{
    public static void Test (object o)
    {
        Class1 a;
        Class2 b;
        if (o is Class1)
        {
            Console.WriteLine ("o jest typu Class1");
            a = (Class1)o;
            // do something with a
        }else if (o is Class2)
        {
            Console.WriteLine ("o jest typu Class2");
            b = (Class2)o;
            // do something with b
        }else
        {
            Console.WriteLine ("o jest innego typu.");
        }
    }
}
```

Rozpoznawanie typu obiektu – przykład 3

```
using System;
class MyClass1
{
}
class MyClass2
{
}

public class IsTest
{
    public static void Main()
    {
        object [] myObjects = new object[6];
        myObjects[0] = new MyClass1();
        myObjects[1] = new MyClass2();
        myObjects[2] = "hello";
        myObjects[3] = 123;
        myObjects[4] = 123.4;
        myObjects[5] = null;

        for (int i=0; i<myObjects.Length; ++i)
        {
            string s = myObjects[i] as string;
            Console.Write ( "{0}:", i );
            if (s != null)
                Console.WriteLine ( "'" + s + "'" );
            else
                Console.WriteLine ( "to nie tekst" );
        }
    }
}
```


Instrukcje sterujące

- Instrukcje `if`, `while`, `do...while` i `for` wymagają wyrażenia typu `bool`
- W instrukcji `switch` każdy blok musi kończyć się instrukcją `break` lub `goto case`
- Instrukcja `foreach` umożliwia wygodne tworzenie pętli. Nie powinna być używana do modyfikacji, a tylko do odczytu. Jeśli iterująca zmienna jest typem wartości, jest tylko do odczytu.

foreach – przykład 1

```
int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};  
foreach (int element in numbers)  
{  
    System.Console.WriteLine(element);  
}
```

foreach – przykład 2

```
foreach (DiagramObject d in dArray)
{
    if (d is IScalable)
    {
        IScalable scalable = (IScalable) d;
        scalable.ScaleX(0.5F);
    }
}
```

Obsługa błędów

- Podstawowym mechanizmem obsługi błędów są wyjątki
- Wszystkie klasy wyjątków dziedziczą po `System.Exception`
- Można użyć wielu następujących po sobie sekcji `catch`, sekcja `catch` bez parametru łapie wszystkie wyjątki
- Złapany wyjątek można rzucić dalej instrukcją `throw`, również złapany w sekcji bez parametru
- Sekcja `finally` umożliwia wykonanie instrukcji bezpośrednio po bloku `try`, również w przypadku rzucenia wyjątku

Obsługa błędów – przykład 1

```
class MyClass
{
    public static void Main()
    {
        MyClass x = new MyClass();
        try
        {
            string s = null;
            x.MyFn(s);
        }

        // Węższy:
        catch (ArgumentNullException e)
        {
        }

        // Szerszy:
        catch (Exception e)
        {
        }

    }

    public void MyFn(string s)
    {
        if (s == null)
            throw new ArgumentNullException();
    }
}
```

Obsługa błędów – przykład 2

```
public static void Main()
{
    int x;
    try
    {
        x = 123;    // źle.
        // ...
    }
    catch
    {
        // ...
    }
    Console.Write(x);    // błąd
}
```

Obsługa błędów – przykład 3

```
object o2 = null;
try
{
    int i2 = (int) o2;
}
catch (InvalidCastException e)
{
    // jakieś procedury...
    throw (e);
}
catch
{
    // jakieś procedury...
    throw;
}
```

Obsługa błędów – przykład 4

```
public class EHClass
{
    public static void Main ()
    {
        try
        {
            // sprawdzany kod...
            throw new NullPointerException();
        }

        finally
        {
            // zawsze wykonane
        }
        catch (NullPointerException e)
        {
            // wyjątek węższy
        }

        catch
        {
            // wyjątek szerszy
        }

    }
}
```


Błędy nadmiaru

- Możliwe jest decydowanie o wychwytywaniu błędów nadmiaru
- Instrukcje `checked` i `unchecked` umożliwiają odrębne traktowanie fragmentów kodu
- Sprawdzane są wyniki następujących operacji na liczbach całkowitych:
++ - - (unary) + - * /
oraz jawnych konwersji między typami całkowitymi

Błędy nadmiaru - przykład

```
class Test
{
    public static void Main()
    {
        checked
        {
            byte a=55;
            byte b=210;
            byte c = (byte) (a+b);
        }
    }
}
```

```
class Test
{
    public static void Main()
    {
        unchecked
        {
            byte a=55;
            byte b=210;
            byte c = (byte) (a+b);
        }
    }
}
```

Operatory 1

- Wbudowane są operatory dotyczące typów `int`, `uint`, `long`, `ulong`, `float`, `double` i `decimal`

- Operatory można przeciążać. Przeciążalne są:
jednoargumentowe:

`+` `-` `!` `~` `++` `--` `true` `false`

dwuargumentowe:

`+` `-` `*` `/` `%` `&` `|` `^` `<<` `>>`

porównania (parami):

`==` `!=` `>` `<` `>=` `<=`

- Nie przeciąża się:

`&&` `||` (ale korzystają z `&` `|` `true` i `false`)

`[]` (ale można definiować indeksatory)

`()` (ale można definiować operatory konwersji)

`+=` `-=` `*=` `/=` `%=` `&=` `|=` `^=` `<<=` `>>=` (ale

korzystają z przeciążalnych operatorów)

`=` `.` `?:` `->` `new` `is` `as` `sizeof` `typeof`

Operatory 2

- Operator musi być publiczną statyczną funkcją składową
- Przynajmniej jeden argument operatora musi być tego samego typu co klasa w której jest definiowany

Operatory - przykład

```
public class IntVector
{
    public IntVector(int length) {}
    public int Length {}
    public int this[int index] {}
    public static IntVector operator ++(IntVector iv) {
        for (int i = 0; i < iv.Length; i++)
            iv[i] = iv[i] + 1;
        return iv;
    }
}
class Test
{
    static void Main() {
        IntVector iv1 = new IntVector(4); // 4 x 0
        IntVector iv2;
        iv2 = iv1++; // ???
        iv2 = ++iv1; // ???
    }
}
```

Operatory - przykład

```
public class IntVector
{
    public IntVector(int length) {}
    public int Length {}
    public int this[int index] {}
    public static IntVector operator ++(IntVector iv) {
        IntVector temp = new IntVector(iv.Length);
        for (int i = 0; i < iv.Length; i++)
            temp[i] = iv[i] + 1;
        return temp;
    }
}
class Test
{
    static void Main() {
        IntVector iv1 = new IntVector(4); // 4 x 0
        IntVector iv2;
        iv2 = iv1++; // iv2 4 x 0, iv1 4 x 1
        iv2 = ++iv1; // iv2 4 x 2, iv1 4 x 2
    }
}
```

Konwersje

- Niejawne – dla wartości: brak utraty zakresu (ale możliwy precyzji). Również do klasy bazowej i zaimplementowanego interfejsu.
- Jawne – możliwa utrata zakresu
- Użytkownika:
 - publiczna statyczna funkcja składowa klasy
 - parametr lub wartość zwracana są typu takiego, w jakim konwersja jest deklarowana
 - dla określenia czy konwersja jest niejawna czy jawna używa się słów kluczowych `explicit` i `implicit`

Konwersje - przykład

```
using System;
public struct Digit
{
    byte value;
    public Digit(byte value)
    {
        if (value < 0 || value > 9)
            throw new ArgumentException();
        this.value = value;
    }
    public static implicit operator byte(Digit d)
    {
        return d.value;
    }
    public static explicit operator Digit(byte b)
    {
        return new Digit(b);
    }
}
```


Tablice

- Tablice mogą być jednowymiarowe, wielowymiarowe, poszarpane
- Układ w pamięci nie jest określony
- W przypadku tablicy typów referencyjnych obiekty nie są tworzone w momencie utworzenia tablicy
- Dziedziczą po `System.Array`, co daje funkcje takie jak `Sort`, `IndexOf`, `LastIndexOf`, `BinarySearch` i `Reverse`
- Konwersja jednej tablicy do drugiej może nastąpić jeśli mają one tę samą liczbę wymiarów, obie są tablicami typów referencyjnych i istnieją odpowiednie konwersje między tymi typami

Tablice - przykład

```
int[] store = new int[50];  
string[] names = new string[50];  
  
int[][] a = new int[100][];  
for (int i = 0; i < 100; i++) a[i] = new int[5];  
  
int[,] b = new int[100, 5];
```

Łańcuchy znakowe

- Typ `System.String`
- Jest niemodyfikowalny
- Obsługuje liczne operacje, np. `Compare`, `StartsWith`, `EndsWith`, `IndexOf`, `LastIndexOf`, `Concat`, `Insert`, `Remove`, `Replace`, `Split`, `Substring`, `ToLower`, `ToUpper` itp.
- Zawiera zawsze znaki w Unicode, ale są funkcje do konwersji z/do innego kodowania
- Można formatować łańcuch statyczną metodą `Format` lub klasą `StringBuilder`

Łańcuchy znakowe – metoda Format

- `public static string Format(string format, params object[] args);`
- **argument** `format` zawiera elementy formatujące o postaci `{index[, alignment] [:formatString]}`
gdzie:
`index` – indeks obiektu który ma być formatowany
`alignment` – opcjonalne pole określające minimalną szerokość sformatowanego pola, ujemna wartość wyrównuje do lewej, dodatnia do prawej
`formatString` – opcjonalny argument (dla typów wbudowanych lub jeśli obiekt obsługuje interfejs `IFormattable`)

Łańcuchy znakowe – `formatString` dla liczb

- składnia `Axx` gdzie A może być:
 - C - waluta
 - D - całkowity
 - E - wykładniczy
 - F - stałoprzecinkowy
 - G - optymalny
 - N – z rozdzieleniem tysięcy
 - P - procentowy
 - R – gwarantujący konwersję powrotną
 - X - szesnastkowy
- Można też używać małych liter
- `xx` – precyzja, znaczenie w zależności od A

Łańcuchy znakowe – przykład 1

```
String s = String.Format("Brad's dog has {0,-8:G} fleas.", 42);
```

Brad's dog has 42_____ fleas.

```
String.Format(
```

```
"The identity of {0} has been accessed {1} times.",
```

```
    Identity.Name,
```

```
    nAccesses);
```

Łańcuchy znakowe - StringBuilder

- Umożliwia konstruowanie łańcucha poprzez modyfikacje istniejącego (jest typu mutable)

StringBuilder - przykład

```
string s = "I will not buy this record.";
char[] separators = new char[]{' '};
StringBuilder sb = new StringBuilder();
int number = 1;
foreach (string sub in s.Split(separators))
{
    sb.AppendFormat("{0}: {1} ", number++, sub);
}
Console.WriteLine("{0}", sb);
```

1: I 2: will 3: not 4: buy 5: this 6: record.

Właściwości

- Umożliwiają dostęp do pól przy pomocy funkcji, wywoływanych tak jak pola
- Mogą umożliwiać odczyt, zapis bądź jedno i drugie
- Mogą być wirtualne i abstrakcyjne
- W klasach pochodnych przesłania się całą właściwość, nie można przesłonić jedynie zapisu/odczytu
- Mogą być statyczne

Właściwości – przykład 1

```
public abstract class DrawingObject
{
    public abstract string Name
    {
        get;
    }
}
class Circle : DrawingObject
{
    string name = "Circle";
    public override string Name
    {
        get
        {
            return (name);
        }
    }
}
class Test
{
    public static void Main()
    {
        DrawingObject d = new Circle();
        Console.WriteLine("Name: {0}", d.Name);
    }
}
```

Właściwości – przykład 2

```
public class BaseClass
{
    private string name;
    public string Name{
        get{
            return name;
        }
        set{
            name = value;
        }
    }
}
public class DerivedClass : BaseClass
{
    private string name;
    public new string Name{
        get{
            return name;
        }
        set{
            name = value;
        }
    }
}
```

Indeksatory

- Umożliwiają odwoływanie się do obiektu jak do tablicy
- Mogą symulować tablice wielowymiarowe
- Można indeksować za pomocą różnych typów (np. łańcuchów znakowych)
- Działają podobnie jak właściwości
- Nie odpowiada im osobne słowo kluczowe, deklaruje się je za pomocą słowa kluczowego `this`

Indeksatory - przykład

```
class Grid
{
    const int NumRows = 26;
    const int NumCols = 10;
    int[,] cells = new int[NumRows, NumCols];
    public int this[char c, int col] {
        get {
            c = Char.ToUpper(c);
            if (c < 'A' || c > 'Z') {
                throw new ArgumentException();
            }
            if (col < 0 || col >= NumCols) {
                throw new IndexOutOfRangeException();
            }
            return cells[c - 'A', col];
        }
        set {
            c = Char.ToUpper(c);
            if (c < 'A' || c > 'Z') {
                throw new ArgumentException();
            }
            if (col < 0 || col >= NumCols) {
                throw new IndexOutOfRangeException();
            }
            cells[c - 'A', col] = value;
        }
    }
}
```

Atrybuty 1

- Umożliwiają podanie dodatkowych informacji służących m.in. do opisanie (skomentowania) kodu czy zmiany funkcjonowania środowiska uruchomieniowego
- Mogą zostać wydobyte podczas wykonywania programu
- Atrybut umieszcza się w nawiasach kwadratowych przed deklaracją do której się odnosi
- Argumenty atrybutu mogą być rozpoznawane po pozycji lub nazwie
- Można zastosować wiele atrybutów, umieszcza się je w jednej parze nawiasów rozdzielone przecinkiem, lub każdy w osobnej parze

Atrybuty 2

- W przypadku niejednoznaczności do którego elementu deklaracji należy zastosować atrybut, można wyrazić to jawnie, np:

```
[SomeAttr] int Method1( string s ) // do metody  
[method: SomeAttr] int Method1( string s ) //  
do metody  
[return: SomeAttr] int Method1( string s ) //  
do wartości zwracanej
```

- **Możliwe modyfikatory:** assembly, module, class, struct, interface, enum, delegate, method, parameter, field, property – indexer, property – get, property – set, event – field, event – property, event – add, event – remove
- Można tworzyć własne atrybuty dziedzicząc od `System.Attribute`

Atrybuty 3

- Tworząc własną klasę atrybutu należy podać gdzie można ją zastosować. Używa się tu atrybutu `AttributeUsage` i typu wyliczeniowego `AttributeTargets`, przyjmującego wartości: `Assembly`, `Module`, `Class`, `Struct`, `Enum`, `Constructor`, `Method`, `Property`, `Field`, `Event`, `Interface`, `Parameter`, `Return`, `Delegate`, `All`, `ClassMembers`
- Określa on również czy dany atrybut można zastosować wielokrotnie w danym miejscu (parametr `AllowMultiple`)

Atrybuty - przykład

```
using System;
[AttributeUsage(AttributeTargets.Class|AttributeTargets.Struct,
    AllowMultiple=true)]
public class Author : Attribute
{
    public Author(string name)
    {
        this.name = name; version = 1.0;
    }
    public double version;
    string name;
    public string GetName()
    {
        return name;
    }
}

[Author("H. Ackerman")]
class FirstClass
{
    /*...*/
}

class SecondClass // no Author attribute
{
    /*...*/
}

[Author("H. Ackerman"), Author("M. Knott", version=1.1)]
class Steerage
{
    /*...*/
}
```

Atrybuty – przykład c.d.

```
class AuthorInfo
{
    public static void Main()
    {
        PrintAuthorInfo (typeof (FirstClass));
        PrintAuthorInfo (typeof (SecondClass));
        PrintAuthorInfo (typeof (Steerage));
    }
    public static void PrintAuthorInfo (Type t)
    {
        Console.WriteLine ("Author information for {0}", t);
        Attribute[] attrs = Attribute.GetCustomAttributes (t);
        foreach (Attribute attr in attrs)
        {
            if (attr is Author)
            {
                Author a = (Author)attr;
                Console.WriteLine ("    {0}, version {1:f}",
                    a.GetName (), a.version);
            }
        }
    }
}
```

Delegacje 1

- Odpowiednik wskaźników do funkcji
- Dziedziczą po System.Delegate
- Delegacje mogą wskazywać na:
 - metodę statyczną
 - obiekt i metodę niestaticzną
 - inną delegację
- Mogą mieć modyfikatory dostępu
- Mogą mieć słowo kluczowe `new`
- Przy pomocy delegacji można wywołać metodę jeśli jest ona kompatybilna z delegacją, tzn. ma tę samą liczbę parametrów, parametry są tych samych typów, w tej samej kolejności, z tymi samymi modyfikatorami oraz typ zwracany jest ten sam

Delegacje 2

- Delegacje można łączyć przy pomocy operatorów `+`, `+=`, `-`, `-=`
- Wywołanie pojedynczej delegacji jest identyczne z wywołaniem funkcji, w przypadku wywołania łańcucha obowiązują następujące zasady:
 - wywołania następują po kolei
 - każda funkcja dostaje ten sam zestaw zmiennych, jeśli występują parametry typu `ref` lub `out`, zmiany wprowadzone przez jedną funkcję będą widziane w następnej
 - wartością zwracaną jest wartość ostatniej z funkcji
 - nieobsłużony wyjątek w którejś z funkcji powoduje, że kolejne nie są wywoływane

Delegacje – przykład 1

```
delegate int D1(int i, double d);  
class A  
{  
    public static int M1(int a, double b) {...}  
}  
class B  
{  
    delegate int D2(int c, double d);  
    public static int M1(int f, double g) {...}  
    public static void M2(int k, double l) {...}  
    public static int M3(int g) {...}  
    public static void M4(int g) {...}  
  
    void test()  
    {  
        D1 del1;  
        D2 del2;  
        del1 = A.M1;  
        del2 = A.M1;  
        del1 = M1;  
        del2 = M1;  
        del1 = M2; //error  
        int x = del2(2, 6);  
    }  
}
```

Delegacje – przykład 2

```
delegate void D(int x);  
class C  
{  
    public static void M1 (int i) {...}  
    public static void M2 (int i) {...}  
}  
class Test  
{  
    static void Main() {  
        D cd1 = new D(C.M1); // M1  
        D cd2 = new D(C.M2); // M2  
        D cd3 = cd1 + cd2; // M1 + M2  
        D cd4 = cd3 + cd1; // M1 + M2 + M1  
        D cd5 = cd4 + cd3; // M1 + M2 + M1 + M1 + M2  
    }  
}
```

Delegacje – przykład 3

```
delegate void D(int x);  
class C  
{  
    public static void M1(int i) {...}  
    public void M2(int i) {...}  
}  
class Test  
{  
    static void Main() {  
        D cd1 = new D(C.M1); // metoda statyczna  
        Test t = new C();  
        D cd2 = new D(t.M2); // metoda obiektu  
        D cd3 = new D(cd2); // delegacja  
    }  
}
```

Delegacje – przykład 4

```
delegate void D(int x);
class C{
    public static void M1 (int i) { /* ... */ }
    public static void M2 (int i) { /* ... */ }
    public void M3 (int i) { /* ... */ }
}
class Test{
    static void Main() {
        D cd1 = new D(C.M1);
        cd1(-1); // M1
        D cd2 = new D(C.M2);
        cd2(-2); // M2
        D cd3 = cd1 + cd2;
        cd3(10); // M1 i M2
        cd3 += cd1;
        cd3(20); // M1, M2 i M1
        C c = new C();
        D cd4 = new D(c.M3);
        cd3 += cd4;
        cd3(30); // M1, M2, M1 i M3
        cd3 -= cd1; // usunięcie ostatniego M1
        cd3(40); // M1, M2 i M3
        cd3 -= cd4;
        cd3(50); // M1 i M2
    }
}
```


Delegacje – przykład 5

```
public delegate void Del(string message);  
  
public static void DelegateMethod(string message)  
{  
    System.Console.WriteLine(message);  
}  
  
Del handler = DelegateMethod;  
  
handler("Hello World");  
  
public void MethodWithCallback(int param1,  
    int param2, Del callback)  
{  
    callback("The number is: " +  
        (param1 + param2).ToString());  
}  
  
MethodWithCallback(1, 2, handler);
```

Zdarzenia 1

- Sygnalizują zajście określonych zmian
- Są deklarowane przy użyciu delegacji
- Kroki:
 - deklaracja delegacji. Definiuje ona jakie argumenty są przekazywane do metody obsługującej zdarzenie. Możliwe są różne argumenty, ale specyfikacja .NET wymaga argumentów `Object sender` i `EventArgs e`. Można wykorzystać wbudowaną delegację `EventHandler`
 - deklaracja zdarzenia – podobna do deklaracji pola typu delegacji, ale poprzedzona słowem kluczowym `event`
 - wywołanie zdarzenia – tak jak wywołanie delegacji. Należy sprawdzić czy nie jest `null`. Wywołanie możliwe jest tylko z klasy deklarującej zdarzenie

Zdarzenia 2

- Dołączanie się do zdarzenia: zdarzenie wygląda z zewnątrz jak pole, ale możliwe są tylko operacje $+=$ i $-=$

Zdarzenia – przykład 1

```
public delegate void ChangedEventHandler(object sender, EventArgs e);
public class ListWithChangedEvent: ArrayList
{
    public event ChangedEventHandler Changed;
    protected virtual void OnChanged(EventArgs e)
    {
        if (Changed != null)
            Changed(this, e);
    }
    public override int Add(object value)
    {
        int i = base.Add(value);
        OnChanged(EventArgs.Empty);
        return i;
    }
}
class EventListener
{
    private ListWithChangedEvent List;
    public EventListener(ListWithChangedEvent list)
    {
        List = list;
        List.Changed += new ChangedEventHandler(ListChanged);
    }
    private void ListChanged(object sender, EventArgs e)
    {
        Console.WriteLine("This is called when the event fires.");
    }
    public void Detach()
    {
        List.Changed -= new ChangedEventHandler(ListChanged);
        List = null;
    }
}
```

Zdarzenia – przykład 1c.d.

```
class Test
{
    // Test the ListWithChangedEvent class.
    public static void Main()
    {
        // Create a new list.
        ListWithChangedEvent list = new ListWithChangedEvent();

        // Create a class that listens to the list's change event.
        EventListener listener = new EventListener(list);

        // Add and remove items from the list.
        list.Add("item 1");
        list.Clear();
        listener.Detach();
    }
}
```

Zdarzenia – przykład 2

```
public class ListWithChangedEvent: ArrayList
{
    public event EventHandler Changed;
    protected virtual void OnChanged(EventArgs e)
    {
        if (Changed != null)
            Changed(this,e);
    }
    public override int Add(object value)
    {
        int i = base.Add(value);
        OnChanged(EventArgs.Empty);
        return i;
    }
}
class EventListener
{
    private ListWithChangedEvent List;
    public EventListener(ListWithChangedEvent list)
    {
        List = list;
        List.Changed += new EventHandler(ListChanged);
    }
    private void ListChanged(object sender, EventArgs e)
    {
        Console.WriteLine("This is called when the event fires.");
    }
    public void Detach()
    {
        List.Changed -= new EventHandler(ListChanged);
        List = null;
    }
}
```

przestrzeń System

- podstawowe klasy i klasy bazowe

Klasy wyjątków

- `System.Exception`
 - `System.SystemException`
 - `System.ArgumentException`
 - `System.ArgumentNullException`
 - `System.ArgumentOutOfRangeException`
 - `System.ArithmeticException`
 - `System.FormatException`
 - `System.IndexOutOfRangeException`
 - `System.InvalidCastException`
 - `System.NullReferenceException`
 - `System.ApplicationException`

klasa Exception

- **System.Object**
 - System.Exception
- `public class Exception : ISerializable`
- **Własności:**
 - `protected int HRESULT {get; set;}`
 - `public virtual string Message {get;}`
 - `public virtual string Source {get; set;}`
- **Metody:**
 - `public Exception();`
 - `public Exception(string message);`
- **Domyślny konstruktor ustwia własność Message automatycznie**

Klasa Object

- **Metody:**

- public virtual bool Equals(object);
- public virtual int GetHashCode();
- public virtual string ToString();

Object.Equals

- Domyślna implementacja sprawdza referencje
- Powinna być stosowana do porównania wartości
- Nie może rzucać wyjątków
- Zaleca się również implementację GetHashCode
- Implementacja musi spełniać pewne warunki:
 - `x.Equals(x)` zawsze zwraca `true`
 - `x.Equals(y)` zwraca to samo co `y.Equals(x)`
 - `(x.Equals(y) && y.Equals(z))` zwraca `true` wtedy i tylko wtedy gdy `x.Equals(z)` zwraca `true`
 - `x.Equals(null)` zwraca `false`

Operatory równości dla klasy Object

- Nie są składowymi klasy Object
 - bool operator ==(object x, object y);
 - bool operator !=(object x, object y);
- domyślnie porównują referencję
- nie można porównywać obiektów o których wiadomo w momencie kompilacji że są różne
- zalecane przeładowanie tylko dla typów niemutowalnych
- nie powinny rzucać wyjątków
- powinny być przeładowane oba

operator == - przykład

```
using System;
class Test
{
    public static void Main()
    {
        // Rownosc wartosci: True
        Console.WriteLine((2 + 2) == 4);

        // Rownosc referencji: rozne obiekty, ta sama wartosc: False
        object s = 1;
        object t = 1;
        Console.WriteLine(s == t);

        // lancuch znakowy
        string a = "hello";
        string b = String.Copy(a);
        string c = "hello";

        // porownanie wartosci referencji do literalu i obiektu: True
        Console.WriteLine(a == b);

        // porownanie referencji;
        // a jest referencja do literalu, b do obiektu: False
        Console.WriteLine((object)a == (object)b);

        // porownanie referencji, poniewaz literaly sa identyczne,
        //zostaly 'spakowane' w jeden obiekt: True
        Console.WriteLine((object)a == (object)c);
    }
}
```

True
False
True
False
True

Object.GetHashCode

- Definiuje funkcję mieszającą dla danego typu
- Stosowana w algorytmach mieszających
- Domyślna implementacja jest prawidłowa jeśli Equals operuje na zasadzie porównania referencji
- Nie może rzucać wyjątków
- Musi spełniać następujące warunki:
 - jeśli dwa obiekty tego samego typu mają tę samą wartość, zwraca ten sam kod
 - zwracany kod powinien mieć losowy rozkład

Object.ToString

- Domyślna implementacja zwraca nazwę typu
- W klasach pochodnych zazwyczaj używana do zwracania wartości obiektu, z uwzględnieniem lokalizacji

Klasa Console

- **System.Object**
 - System.Console
- **Własności**
 - public static TextReader In {get;}
 - public static TextWriter Out {get;}
 - public static TextWriter Error {get;}
- **Metody**
 - public static void SetIn(TextReader newIn);
 - public static void SetOut(TextWriter newOut);
 - public static void SetError(TextWriter newError);
 - public static int Read();
 - public static string ReadLine();
 - public static void Write(string, params object[]);
 - public static void WriteLine(string, params object[]);

Console.Read, Console.ReadLine

- Read zwraca wczytany znak, lub -1 jeśli nie ma już więcej znaków
- ReadLine zwraca wczytaną linię, bez znaku kończącego, lub null jeśli nie ma więcej znaków

Console.Write, Console.WriteLine

- Wypisują tekst używając formatowania jak w `String.Format`
- Istnieją przeciążone metody dla typów wbudowanych
- `WriteLine` dodaje znak końca linii

klasa Convert

- Umożliwia przeprowadzanie konwersji pomiędzy typami wbudowanymi
- Niektóre konwersje (np z/do typu `Char`) nie są możliwe i powodują wyjątek `InvalidCastException`
- Wyjątek nie jest rzucany przy utracie precyzji, ale jest rzucany `System.OverflowException` jeśli wynik nie mieści się w danym typie
- Metody mają postać `ToNazwaTypu`

struktura DateTime

- System.Object
 - System.ValueType
 - System.DateTime
- Własności:
 - Date, Day, DayOfWeek, DayOfYear, Month etc
- Metody:
 - Add, AddDays, AddMinutes etc.
 - Compare
 - Parse
 - ToLongDateString
 - ToLongTimeString
 - ToString
 - Subtract
- Operatory
 - porównania

struktura DateTime cd

- Przechowuje datę i czas od roku 1 do 9999, z rozdzielczością 100 nanosekund (1 tick)
- Używa struktury TimeSpan do operacji dodawania i odejmowania
- Obliczenia są sensowne tylko dla struktur utworzonych w tej samej strefie czasowej

interfejs Cloneable

- Metody:
 - `Object clone()` ;
- Zwraca płytką bądź głęboką (bardziej sensowne) kopię obiektu
- Zwracany obiekt musi być tego samego typu co klasa implementująca interfejs
- Do płytkiej kopii można użyć metody `MemberwiseClone`

interfejs Comparable

- Metody:
 - `int compareTo(object obj);`
- Zwracana wartość: ujemna jeśli this mniejsze od obj, 0 jeśli równe, dodatnia jeśli większe
- Zasady podobne do `Object.Equals()`
- Każdy obiekt jest większy niż null

interfejs IDisposable

- Metody:
 - `void Dispose();`
- Używany do zwolnienie niezarządzanych zasobów
- Musi dopuszczać wielokrotne wywołanie metody

klasa Math

- **System.Object**
 - System.Math
- **Pola:**
 - `public const double E;`
 - `public const double PI;`
- **Metody:**
 - `Abs, Cos, Exp, Log, Log10, Min, Max, Pow, Sqrt etc.`
 - metody są statyczne

klasa Random

- System.Object
 - System.Random
- Metody:
 - `public Random();`
 - `public Random(int Seed);`
 - `public virtual int Next();`
 - `public virtual int Next(int maxValue);`
 - `public virtual int Next(int minValue, int maxValue);`
- Konstruktor domyślny inicjalizuje generator wartością czasu systemowego

klasa Type

- **System.Object**
 - System.Reflection.MemberInfo
 - System.Type
- **Własności:**
 - BaseType, FullName, IsAbstract, IsArray, IsClass, IsEnum, IsInterface, IsPublic, IsSealed, IsValueType **etc.**
- **Metody:**
 - GetConstructor, GetField, GetInterface, GetMember, GetMethod, GetProperty **etc.**

klasa ValueType

- System.Object
 - System.ValueType
- Obiekt typu wartości może zostać opakowany tak, aby wyglądał jak obiekt typu referencji (boxing). Opakowany obiekt może zostać rozpakowany (unboxing)

boxing - przykład

```
using System;
class TestBoxing
{
    public static void Main()
    {
        int i = 123;
        object o = i; // niejawne opakowanie
        i = 456; // zmiana wartości i
        Console.WriteLine
            ("The value-type value = {0}", i);
        Console.WriteLine
            ("The object-type value = {0}", o);
    }
}
```

The value-type value = 456
The object-type value = 123

unboxing - przykład

```
using System;
public class UnboxingTest
{
    public static void Main()
    {
        int intI = 123;

        // Boxing
        object o = intI;

        try
        {
            int intJ = (short) o; // powinno być int intJ = (int) o;
            Console.WriteLine("Unboxing OK.");
        }

        catch (InvalidCastException e)
        {
            Console.WriteLine("{0} Error: Incorrect unboxing.", e);
        }
    }
}
```

przestrzeń System.Collections

- klasy:
 - ArrayList
 - Hashtable
 - Queue
 - SortedList
 - Stack
- interfejsy:
 - ICollection
 - IComparer
 - IDictionary
 - IEnumerable
 - IEnumerator
 - IList
- struktury:
 - DictionaryEntry

interfejs IEnumerable

- `public interface IEnumerable`
- **Metody:**
 - `IEnumerator GetEnumerator();`

interfejs IEnumerator

- `public interface IEnumerator`
- **Własności:**
 - `object Current {get;}`
- **Metody:**
 - `bool MoveNext();`
 - `void Reset();`
- Nie może być używany do modyfikowania kolekcji
- Pozycja startowa to tuż przed początkiem kolekcji
- Po przekroczeniu końca kolekcji `MoveNext` zwraca `false`
- Zmiany kolekcji niszczą enumerator, wywołanie `MoveNext` lub `Reset` rzuca wyjątek

interfejs ICollection

- `public interface ICollection :
IEnumerable`
- **Własności:**
 - `public virtual int Count {get;}`
- **Metody:**
 - `void CopyTo(Array array, int index);`

interfejs IList

- `public interface IList :
ICollection, IEnumerable`
- **Własności**
 - `public virtual object this[int index]
{get; set;}`
- **Metody**
 - `public virtual int Add(object value);`
 - `public virtual void Insert(int index,
object value);`
 - `public virtual void RemoveAt(int
index);`
 - `public virtual void Clear();`

interfejs IDictionary

- `public interface IDictionary :
ICollection, IEnumerable`
- **Własności**
 - `public virtual ICollection Keys {get;}`
 - `public virtual ICollection Values
{get;}`
 - `Object this [Object key] { get; set; }`
 - `void Add (Object key, Object value)`
 - `void Clear ()`

klasa ArrayList

- **System.Object**
 - System.Collections.ArrayList
- `public class ArrayList : IList, ICollection, IEnumerable, ICloneable`
- **Własności**
 - `public virtual int Capacity {get; set;}`
- **Metody:**
 - `public ArrayList();`
 - `public ArrayList(int capacity);`
 - `public ArrayList(ICollection c);`
 - `public virtual IEnumerator GetEnumerator();`
 - `public virtual void Sort();`

klasa ArrayList - przykład

```
using System;
using System.Collections;
public class SamplesArrayList {

    public static void Main() {

        // tworzy nowa ArrayList.
        ArrayList myAL = new ArrayList();
        myAL.Add("Hello");
        myAL.Add("World");
        myAL.Add("!");

        // Wyszwietla wlascnosci i wartosci.
        Console.WriteLine( "myAL" );
        Console.WriteLine( "\tCount: {0}", myAL.Count );
        Console.WriteLine( "\tCapacity: {0}", myAL.Capacity );
        Console.Write( "\tValues:" );
        PrintValues( myAL );
    }

    public static void PrintValues( IEnumerable myList ) {
        System.Collections.IEnumerator myEnumerator =
            myList.GetEnumerator();
        while ( myEnumerator.MoveNext() )
            Console.Write( "\t{0}", myEnumerator.Current );
        Console.WriteLine();
    }
}
```

```
myAL
Count: 3
Capacity: 16
Values: Hello World !
```

klasa Hashtable

- **System.Object**
 - **System.Collections.Hashtable**
- `public class Hashtable : IDictionary, ICollection, IEnumerable, ISerializable, IDeserializationCallback, ICloneable`
- **Metody:**
 - `public virtual void Add(object key, object value);`
 - `public virtual Object this [Object key] { get; set; }`

klasa Hashtable – przykład

```
using System;
using System.Collections;
public class SamplesHashtable {

    public static void Main() {

        // tworzy nowa Hashtable.
        Hashtable myHT = new Hashtable();
        myHT.Add("First", "Hello");
        myHT.Add("Second", "World");
        myHT.Add("Third", "!");

        // Wyszwietla wlascnosci i wartosci Hashtable.
        Console.WriteLine( "myHT" );
        Console.WriteLine( "  Count:      {0}", myHT.Count );
        Console.WriteLine( "  Keys and Values:" );
        PrintKeysAndValues( myHT );
    }

    public static void PrintKeysAndValues( Hashtable myList ) {
        IDictionaryEnumerator myEnumerator = myList.GetEnumerator();
        Console.WriteLine( "\t-KEY-\t-VALUE-" );
        while ( myEnumerator.MoveNext() )
            Console.WriteLine( "\t{0}:\t{1}",
                myEnumerator.Key, myEnumerator.Value );
        Console.WriteLine();
    }
}
```

```
myHT
Count: 3
Keys and Values:
-KEY- -VALUE-
Third: !
Second: World
First: Hello
```


klasa Queue

- **System.Object**
 - System.Collections.Queue
- `public class Queue : ICollection, IEnumerable, ICloneable`
- **Metody:**
 - `public virtual void Enqueue(object obj);`
 - `public virtual object Dequeue();`
 - `public virtual object Peek();`

Klasa Queue - przykład

```
using System;
using System.Collections;
public class SamplesQueue {

    public static void Main() {
        Queue myQ = new Queue();
        myQ.Enqueue( "The" );
        myQ.Enqueue( "quick" );
        myQ.Enqueue( "brown" );
        myQ.Enqueue( "fox" );
        Console.Write( "Queue values:" );
        PrintValues( myQ, '\t' );
        Console.WriteLine( "(Dequeue) \t{0}", myQ.Dequeue() );
        Console.Write( "Queue values:" );
        PrintValues( myQ, '\t' );
        Console.WriteLine( "(Dequeue) \t{0}", myQ.Dequeue() );
        Console.Write( "Queue values:" );
        PrintValues( myQ, '\t' );
        Console.WriteLine( "(Peek) \t{0}", myQ.Peek() );
        Console.Write( "Queue values:" );
        PrintValues( myQ, '\t' );
    }

    public static void PrintValues( IEnumerable myCollection,
        char mySeparator ) {
        System.Collections.IEnumerator myEnumerator =
            myCollection.GetEnumerator();
        while ( myEnumerator.MoveNext() )
            Console.Write( "{0}{1}", mySeparator, myEnumerator.Current );
        Console.WriteLine();
    }
}
```

```
Queue values: The quick brown fox
(Dequeue) The
Queue values: quick brown fox
(Dequeue) quick
Queue values: brown fox
(Peek) brown
Queue values: brown fox
```

klasa Stack

- **System.Object**
 - System.Collections.Stack
- `public class Stack : ICollection, IEnumerable, ICloneable`
- **Metody:**
 - `public virtual void Push(object obj);`
 - `public virtual object Pop();`
 - `public virtual object Peek();`

przestrzeń System.IO

- klasy:
 - BinaryReader
 - BinaryWriter
 - BufferedStream
 - Directory
 - DirectoryInfo
 - File
 - FileInfo
 - FileStream
 - Path
 - Stream
 - StreamReader
 - StreamWriter
 - StringReader
 - StringWriter
 - TextReader
 - TextWriter
 - wyjątki

klasa Stream

- **System.Object**
 - System.MarshalByRefObject
 - System.IO.Stream
- `public abstract class Stream : MarshalByRefObject, IDisposable`
- **Reprezentuje sekwencję bajtów**
- **Można czytać ze strumienia, pisać do strumienia, zmieniać pozycję**

klasa Stream 2

- **Własności:**

- public abstract bool CanRead {get;}
- public abstract bool CanWrite {get;}
- public abstract bool CanSeek {get;}
- public abstract long Length {get;}¹
- public abstract long Position {get; set;}¹

- **Metody:**

- protected Stream();

¹Rzucają wyjątek `NotSupportedException` jeśli operacja nie jest obsługiwana przez strumień

metody odczytu

- `public virtual int ReadByte();`
- `public abstract int Read(byte[] buffer, int offset, int count);`
- `public virtual IAsyncResult BeginRead(byte[] buffer, int offset, int count, AsyncCallback callback, object state)`
- pierwsza czyta jeden bajt i go zwraca, lub -1 jeśli koniec strumienia
- druga czyta `count` bajtów i wstawia do `buffer` począwszy od `offset` znaku. Zwraca liczbę odczytanych znaków
- trzecia wykonuje operacje asynchronicznie, umożliwia podanie funkcji która będzie wywołana po zakończeniu odczytu (`callback`) oraz obiektu identyfikującego dany odczyt (`state`)

metody zapisu

- `public virtual void WriteByte(byte value);`
- `public abstract void Write(byte[] buffer, int offset, int count)`
- `public virtual IAsyncResult BeginWrite(byte[] buffer, int offset, int count, AsyncCallback callback, object state)`

inne metody

- `public abstract long Seek(long offset, SeekOrigin origin);`
- `public abstract void Flush();`
- `public virtual void Close();`
- **pierwsza zmienia pozycję o offset w stosunku do origin (Begin, Current, End)**
- **druga powoduje przesłanie zawartości bufora do miejsca docelowego**
- **trzecia zamyka strumień (wywołując wcześniej Flush)**

klasa BufferedStream

- **System.Object**
 - **System.MarshalByRefObject**
 - **System.IO.Stream**
 - **System.IO.BufferedStream**
- `public sealed class BufferedStream : Stream`
- **metody:**
 - `public BufferedStream(Stream stream);`
 - `public BufferedStream(Stream stream, int bufferSize);`
- **zapewnia automatyczne buforowanie operacji odczytu i zapisu strumienia**

klasa FileStream

- System.Object
 - System.MarshalByRefObject
 - System.IO.Stream
 - System.IO.FileStream
- `public class FileStream : Stream`
- `public FileStream(IntPtr handle, FileAccess access);`
- `public FileStream(string path, FileMode mode);`
- `public FileStream(string path, FileMode mode, FileAccess access);`
- `handle: uchwyt OS, path: ścieżka bezwzględna lub względna, FileMode: Append, Create, CreateNew, Open, OpenOrCreate, Truncate, FileAccess: Read, ReadWrite, Write`

klasa FileStream 2

- **Własności:**

- public virtual IntPtr Handle {get;}
 - public string Name {get;}

- **Metody:**

- public virtual void Lock(long position, long length);
 - public virtual void Unlock(long position, long length);

- **Lock i Unlock blokują/odblokowują możliwość modyfikacji fragmentu (od position do position + length) strumienia. Nie ograniczają możliwości odczytu.**

klasa File

- **System.Object**
 - System.IO.File
- `public sealed class File`
- **Metody:**
 - `public static FileStream Create(string path);`
 - `public static FileStream Create(string path, int bufferSize);`
 - `public static void Delete(string path);`
 - `public static bool Exists(string path);`
 - `public static FileStream Open(string path, FileMode mode);`
 - `public static FileStream Open(string path, FileMode mode, FileAccess access);`
 - `public static FileAttributes GetAttributes(string path);`
 - `public static DateTime GetCreationTime(string path);`
 - `public static void SetAttributes(string path, FileAttributes fileAttributes);`
 - `public static void SetCreationTime(string path, DateTime creationTime);`

klasa File - przykład

```
using System;
using System.IO;
using System.Text;

class Test
{
    public static void Main()
    {
        string path = @"c:\temp\MyTest.txt";

        // usuwamy plik jesli istnieje.
        if (File.Exists(path))
        {
            File.Delete(path);
        }

        // Tworzymy plik.
        using (FileStream fs = File.Create(path, 1024))
        {
            Byte[] info = new UTF8Encoding(true).GetBytes("Some text");
            // Dodajemy dane do pliku.
            fs.Write(info, 0, info.Length);
        }

        // Otwieramy strumien i czytamy.
        using (StreamReader sr = File.OpenText(path))
        {
            string s = "";
            while ((s = sr.ReadLine()) != null)
            {
                Console.WriteLine(s);
            }
        }
    }
}
```

klasa FileInfo

- **System.Object**
 - System.MarshalByRefObject
 - System.IO.FileSystemInfo
 - System.IO.FileInfo
- `public sealed class FileInfo :
FileSystemInfo`
- **Własności:**
 - `public DirectoryInfo Directory {get;}`
 - `public string DirectoryName {get;}`
 - `public override bool Exists {get;}`
 - `public long Length {get;}`
 - `public override string Name {get;}`

klasa File Info 2

- **Metody:**

- public FileInfo (string fileName);
- public FileStream Create();
- public StreamWriter CreateText();
- public override void Delete();
- public FileStream Open (FileMode mode, FileAccess access);
- public StreamReader OpenText();
- public FileInfo CopyTo (string destFileName);

klasa FileInfo – przykład 1

```
using System;
using System.IO;

class Test
{
    public static void Main()
    {
        string path = @"c:\temp\MyTest.txt";
        FileInfo fil = new FileInfo(path);

        if (!fil.Exists)
        {
            //Tworzymy plik.
            using (StreamWriter sw = fil.CreateText())
            {
                sw.WriteLine("Hello");
                sw.WriteLine("And");
                sw.WriteLine("Welcome");
            }
        }

        //Open the file to read from.
        using (StreamReader sr = fil.OpenText())
        {
            string s = "";
            while ((s = sr.ReadLine()) != null)
            {
                Console.WriteLine(s);
            }
        }
    }
}
```

klasa FileInfo – przykład 2

```
try
{
    string path2 = path + "temp";
    FileInfo fi2 = new FileInfo(path2);

    //Upewniamy sie ze plik docelowy nie istnieje.
    fi2.Delete();

    //Kopiujemy plik.
    fi1.CopyTo(path2);
    Console.WriteLine("{0} skopiowano do {1}.", path, path2);

    //Usuwamy nowy plik.
    fi2.Delete();
    Console.WriteLine("{0} zostal usuniety.", path2);
}
catch (Exception e)
{
    Console.WriteLine("Blad: {0}", e.ToString());
}
}
```

klasa Directory

- **System.Object**
 - **System.IO.Directory**
- `public sealed class Directory`
- **Metody:**
 - `public static DirectoryInfo CreateDirectory(string path);`
 - `public static void Delete(string path, bool recursive);`
 - `public static bool Exists(string path);`
 - `public static DirectoryInfo GetParent(string path);`
 - `public static string[] GetDirectories(string path, string searchPattern);`
 - `public static string[] GetFiles(string path, string searchPattern);`
 - `public static void Move(string sourceDirName, string destDirName);`

klasa Directory - przykład

```
using System;
using System.IO;

class Test
{
    public static void Main()
    {
        try
        {
            // Tylko podkatalogi zaczynajace sie od "p."
            string[] dirs = Directory.GetDirectories("c:\", "p*");
            Console.WriteLine("Liczba katalogow zaczynajacych sie
                od p wynosi {0}.", dirs.Length);
            foreach (string dir in dirs)
            {
                Console.WriteLine(dir);
            }
        }
        catch (Exception e)
        {
            Console.WriteLine("Blad: {0}", e.ToString());
        }
    }
}
```

klasa DirectoryInfo

- **System.Object**
 - **System.MarshalByRefObject**
 - **System.IO.FileSystemInfo**
 - **System.IO.DirectoryInfo**
- `public sealed class DirectoryInfo :
FileSystemInfo`
- **Własności:**
 - `public override string Name {get;}`
 - `public override bool Exists {get;}`
 - `public DirectoryInfo Parent {get;}`

klasa DirectoryInfo 2

- **Metody:**

- public DirectoryInfo (string path);
- public void Create();
- public void Delete (bool recursive);
- public DirectoryInfo
CreateSubdirectory (string path);
- public DirectoryInfo[]
GetDirectories (string searchPattern);
- public FileInfo[] GetFiles (string
searchPattern);
- public void MoveTo (string
destDirName);

klasa DirectoryInfo - przykład

```
using System;
using System.IO;

class Test
{
    public static void Main()
    {
        try
        {
            DirectoryInfo di = new DirectoryInfo(@"c:\");
            // Tylko podkatalogi zawierające "p."
            DirectoryInfo[] dirs = di.GetDirectories("*p*");
            Console.WriteLine("Liczba katalogow z p: {0}",
                dirs.Length);

            // Pliki w podkatalogach zawierające "e."
            foreach (DirectoryInfo diNext in dirs)
            {
                Console.WriteLine("Liczba plikow w {0} z e w nazwie
                    wynosi {1}", diNext,
                    diNext.GetFiles("*e*").Length);
            }
        }
        catch (Exception e)
        {
            Console.WriteLine("The process failed: {0}", e.ToString());
        }
    }
}
```

klasa BinaryReader

- **System.Object**
 - **System.IO.BinaryReader**
- `public class BinaryReader : IDisposable`
- **Właściwości:**
 - `public virtual Stream BaseStream {get;}`
- **Metody:**
 - `public BinaryReader(Stream input, Encoding encoding);`
 - `public virtual int Read();`
 - `public virtual int Read(byte[] buffer, int index, int count);`
 - `public virtual char ReadChar();`
 - `public virtual char[] ReadChars(int count);`
 - **metody dla pozostałych typów wbudowanych**

klasa BinaryWriter

- **System.Object**
 - System.IO.BinaryWriter
- `public class BinaryWriter :
IDisposable`
- **Metody:**
 - `public BinaryWriter(Stream output,
Encoding encoding);`
 - `public BinaryWriter(Stream output,
Encoding encoding);`
 - `public virtual void Write(char ch);`
 - `public virtual void Write(char[]
chars);`
 - metody dla innych typów wbudowanych

klasa TextReader

- **System.Object**
 - System.MarshalByRefObject
 - System.IO.TextReader
- `public abstract class TextReader : MarshalByRefObject, IDisposable`
- **Metody:**
 - `protected TextReader();`
 - `public virtual int Read();`
 - `public virtual int Read(char[] buffer, int index, int count);`

klasa TextWriter

- **System.Object**
 - System.MarshalByRefObject
 - System.IO.TextWriter
- `public abstract class TextWriter : MarshalByRefObject, IDisposable`
- **Metody:**
 - `protected TextWriter();`
 - `public virtual void Write(char value);`
 - `public virtual void Write(char[] buffer);`
 - metody dla pozostałych typów wbudowanych...

klasy StreamReader i StringReader

- **System.Object**
 - System.MarshalByRefObject
 - System.IO.TextReader
 - System.IO.StreamReader
- `public class StreamReader : TextReader`
- **Właściwości**
 - `public virtual Stream BaseStream {get;}`
 - `public virtual Encoding CurrentEncoding {get;}`
- **Metody:**
 - `public StreamReader(Stream stream);`
- **System.Object**
 - System.MarshalByRefObject
 - System.IO.TextReader
 - System.IO.StringReader
- `public class StringReader : TextReader`
- **Metody:**
 - `public StringReader(string s);`

klasa StreamWriter

- **System.Object**
 - **System.MarshalByRefObject**
 - **System.IO.TextWriter**
 - **System.IO.StreamWriter**
- `public class StreamWriter :
TextWriter`
- **Metody:**
 - `public StreamWriter(Stream stream);`
 - `public StreamWriter(string path);`
 - `public override void Write(char
value);`
 - `public override void Write(char[]
buffer);`
 - `public override void Write(string
value);`

klasa StringWriter

- **System.Object**
 - **System.MarshalByRefObject**
 - **System.IO.TextWriter**
 - **System.IO.StringWriter**
- `public class StringWriter :
TextWriter`
- **Metody:**
 - `public StringWriter();`
 - `public StringWriter(StringBuilder sb);`
 - `public override void Write(char
value);`
 - `public override void Write(string
value);`
 - `public virtual StringBuilder
GetStringBuilder();`

klasa Path

- **System.Object**
 - **System.IO.Path**
- `public sealed class Path`
- **Pola:**
 - `public static readonly char DirectorySeparatorChar;`
 - `public static readonly char[] InvalidPathChars;`
 - `public static readonly char PathSeparator;`
 - `public static readonly char VolumeSeparatorChar;`
- **Metody:**
 - `public static string GetDirectoryName(string path);`
 - `public static string GetDirectoryName(string path);`
 - `public static string GetFileName(string path);`
 - `public static string Combine(string path1, string path2);`

klasa Path - przykład

```
string fileName = @"C:\mydir.old\myfile.ext";
string path = @"C:\mydir.old\";
string extension;

extension = Path.GetExtension(fileName);
Console.WriteLine("GetExtension('{0}') zwraca '{1}'",
    fileName, extension);

extension = Path.GetExtension(path);
Console.WriteLine("GetExtension('{0}') zwraca '{1}'",
    path, extension);
```


klasa Path – przykład 2

```
using System;
using System.IO;

public class ChangeExtensionTest {
    public static void Main() {
        string path1 = "c:\\temp";
        string path2 = "subdir\\file.txt";
        string path3 = "c:\\temp.txt";
        string path4 = "c:^*&)(_=@#'\\"^&#2.*(.txt";
        string path5 = "";
        string path6 = null;

        CombinePaths(path1, path2);
        CombinePaths(path1, path3);
        CombinePaths(path3, path2);
        CombinePaths(path4, path2);
        CombinePaths(path5, path2);
        CombinePaths(path6, path2);
    }

    private static void CombinePaths(string p1, string p2) {
        try {
            string combination = Path.Combine(p1, p2);
            Console.WriteLine("Polaczenie '{0}' i '{1}' daje: {2}'{3}'",
                p1, p2, Environment.NewLine, combination);
        } catch (Exception e) {
            Console.WriteLine("Nie mozna laczyc '{0}' i '{1}' bo: {2}{3}",
                p1, p2, Environment.NewLine, e.Message);
        }
        Console.WriteLine();
    }
}
```

klasa Path – przykład 2 cd

Polaczenie 'c:\temp' i 'subdir\file.txt' daje:
'c:\temp\subdir\file.txt'

Polaczenie 'c:\temp' i 'c:\temp.txt' daje:
'c:\temp.txt'

Polaczenie 'c:\temp.txt' i 'subdir\file.txt' daje:
'c:\temp.txt\subdir\file.txt'

Polaczenie 'c:^*&)(_=@#\^.*(.txt' i 'subdir\file.txt' daje:
'c:^*&)(_=@#\^.*(.txt\subdir\file.txt'

Polaczenie " i 'subdir\file.txt' daje:
'subdir\file.txt'

Nie mozna laczyć " i 'subdir\file.txt' bo:
Value cannot be null.
Parameter name: path1

Wyjątki

- System.Object
 - System.Exception
 - System.SystemException
 - System.IO.IOException
 - System.IO.DirectoryNotFoundException
 - System.IO.EndOfStreamException
 - System.IO.FileLoadException
 - System.IO.FileNotFoundException
 - System.IO.PathTooLongException

Wątki

- **System.Object**
 - System.Threading.Thread
- **public sealed class Thread**
- **Konstruktor:**
 - `public Thread(ThreadStart start);`
- **Własności:**
 - `public static Thread CurrentThread {get;}`
 - `public bool IsAlive {get;}`
 - `public bool IsThreadPoolThread {get;}`
 - `public string Name {get; set;}`
 - `public ThreadPriority Priority {get; set;}`
 - `public ThreadState ThreadState {get;}`

Wątki

- **Metody:**

- `public void Abort();`
- `public bool Join(int millisecondsTimeout);`
- `public void Resume();`
- `public static void Sleep(int millisecondsTimeout);`
- `public void Start();`
- `public void Suspend();`

- **Delegacje:**

- `public delegate void ThreadStart();`

```

using System;
using System.Threading;

public class ServerClass {
    public void InstanceMethod() {
        Console.WriteLine("ServerClass.InstanceMethod is running
            on another thread.");
        Thread.Sleep(3000);
        Console.WriteLine("The instance method called by the worker
            thread has ended.");
    }

    public static void StaticMethod() {
        Console.WriteLine("ServerClass.StaticMethod is running
            on another thread.");
        Thread.Sleep(5000);
        Console.WriteLine("The static method called by the worker
            thread has ended.");
    }
}

public class Simple {
    public static int Main(String[] args) {
        Console.WriteLine("Thread Simple Sample");
        ServerClass serverObject = new ServerClass();
        Thread InstanceCaller = new Thread(new ThreadStart(
            serverObject.InstanceMethod));
        InstanceCaller.Start();
        Console.WriteLine("The Main() thread calls this after
            starting the new InstanceCaller thread.");
        Thread StaticCaller = new Thread(new ThreadStart(
            ServerClass.StaticMethod));
        StaticCaller.Start();
        Console.WriteLine("The Main() thread calls this after
            starting the new StaticCaller threads.");

        return 0;
    }
}

```

```

public class Work
{
    public static void Main()
    {
        Thread newThread = new Thread(
            new ParameterizedThreadStart(Work.DoWork));
        newThread.Start(42);

        Work w = new Work();
        newThread = new Thread(
            new ParameterizedThreadStart(w.DoMoreWork));
        newThread.Start("The answer.");
    }

    public static void DoWork(object data)
    {
        Console.WriteLine("Static thread procedure. Data='{0}'",
            data);
    }

    public void DoMoreWork(object data)
    {
        Console.WriteLine("Instance thread procedure. Data='{0}'",
            data);
    }
}

```

```

Static thread procedure. Data='42'
Instance thread procedure. Data='The answer'

```

```
using System;
using System.Threading;

public class ThreadWithState {
    private string boilerplate;
    private int value;

    public ThreadWithState(string text, int number) {
        boilerplate = text;
        value = number;
    }

    public void ThreadProc() {
        Console.WriteLine(boilerplate, value);
    }
}

public class Example {
    public static void Main() {
        ThreadWithState tws =
            new ThreadWithState("This report displays the number
                {0}.", 42);
        Thread t = new Thread(new ThreadStart(tws.ThreadProc));
        t.Start();
        Console.WriteLine("Main thread does some work, then waits.");
        t.Join();
        Console.WriteLine("Independent task has completed; main
            thread ends.");
    }
}
```



```
using System;
using System.Threading;

public class ThreadWithState {
    private string boilerplate;
    private int value;
    private ExampleCallback callback;

    public ThreadWithState(string text, int number,
        ExampleCallback callbackDelegate)
    {
        boilerplate = text;
        value = number;
        callback = callbackDelegate;
    }
    public void ThreadProc() {
        Console.WriteLine(boilerplate, value);
        if (callback != null)
            callback(1);
    }
}

public delegate void ExampleCallback(int lineCount);

public class Example {
    public static void Main() {
        ThreadWithState tws = new ThreadWithState(
            "This report displays the number {0}.", 42,
            new ExampleCallback(ResultCallback));
        Thread t = new Thread(new ThreadStart(tws.ThreadProc));
        t.Start();
        Console.WriteLine("Main thread does some work, then waits.");
        t.Join();
        Console.WriteLine("Independent task has completed; main thread
            ends.");
    }
    public static void ResultCallback(int lineCount) {
        Console.WriteLine("Independent task printed {0} lines.",
            lineCount);
    }
}
```

Synchronizacja

- **System.Object**
 - System.Threading.Monitor
- **public sealed class Monitor**
- **Metody:**
 - `public static void Enter(object obj);`
 - `public static void Exit(object obj);`
 - `public static void Pulse(object obj);`
 - `public static bool TryEnter(object obj, int millisecondsTimeout);`
 - `public static bool Wait(object obj, int millisecondsTimeout);`

```

using System;
using System.Threading;
using System.Collections;
namespace MonitorCS1
{
    class MonitorSample
    {
        const int MAX_LOOP_TIME = 1000;
        Queue m_smpQueue;
        public MonitorSample()
        {
            m_smpQueue = new Queue();
        }
        public void FirstThread()
        {
            int counter = 0;
            lock(m_smpQueue)
            {
                while(counter < MAX_LOOP_TIME)
                {
                    Monitor.Wait(m_smpQueue);
                    m_smpQueue.Enqueue(counter);
                    Monitor.Pulse(m_smpQueue);
                    counter++;
                }
            }
        }
        public void SecondThread()
        {
            lock(m_smpQueue)
            {
                Monitor.Pulse(m_smpQueue);
                while(Monitor.Wait(m_smpQueue, 1000))
                {
                    int counter = (int)m_smpQueue.Dequeue();
                    Console.WriteLine(counter.ToString());
                    Monitor.Pulse(m_smpQueue);
                }
            }
        }
    }
}

```

```
//Return the number of queue elements.
public int GetQueueCount()
{
    return m_smplQueue.Count;
}

static void Main(string[] args)
{
    //Create the MonitorSample object.
    MonitorSample test = new MonitorSample();
    //Create the first thread.
    Thread tFirst = new Thread(new ThreadStart(
        test.FirstThread));
    //Create the second thread.
    Thread tSecond = new Thread(new ThreadStart(
        test.SecondThread));
    //Start threads.
    tFirst.Start();
    tSecond.Start();
    //wait to the end of the two threads
    tFirst.Join();
    tSecond.Join();
    //Print the number of queue elements.
    Console.WriteLine("Queue Count = " +
        test.GetQueueCount().ToString());
}
}
}
```

```
using System;
using System.Collections;
using System.Threading;

namespace MonitorCS2
{
    class MonitorSample
    {
        private Queue m_inputQueue;

        public MonitorSample()
        {
            m_inputQueue = new Queue();
        }

        public void AddElement(object qValue)
        {
            //Lock the queue.
            Monitor.Enter(m_inputQueue);
            //Add element
            m_inputQueue.Enqueue(qValue);
            //Unlock the queue.
            Monitor.Exit(m_inputQueue);
        }

        public bool AddElementWithoutWait(object qValue)
        {
            //Determine whether the queue is locked
            if(!Monitor.TryEnter(m_inputQueue))
                return false;
            m_inputQueue.Enqueue(qValue);

            Monitor.Exit(m_inputQueue);
            return true;
        }
    }
}
```

```

public bool WaitToAddElement(object qValue, int waitTime)
{
    //Wait while the queue is locked.
    if(!Monitor.TryEnter(m_inputQueue, waitTime))
        return false;
    m_inputQueue.Enqueue(qValue);
    Monitor.Exit(m_inputQueue);

    return true;
}

public void DeleteElement(object qValue)
{
    Monitor.Enter(m_inputQueue);
    int counter = m_inputQueue.Count;
    while(counter > 0)
    {
        //Check each element.
        object elm = m_inputQueue.Dequeue();
        if(!elm.Equals(qValue))
        {
            m_inputQueue.Enqueue(elm);
        }
        --counter;
    }
    Monitor.Exit(m_inputQueue);
}

public void PrintAllElements()
{
    Monitor.Enter(m_inputQueue);
    IEnumerator elmEnum = m_inputQueue.GetEnumerator();
    while(elmEnum.MoveNext())
    {
        Console.WriteLine(elmEnum.Current.ToString());
    }
    Monitor.Exit(m_inputQueue);
}

```

```
static void Main(string[] args)
{
    MonitorSample sample = new MonitorSample();

    for(int i = 0; i < 30; i++)
        sample.AddElement(i);
    sample.PrintAllElements();
    sample.DeleteElement(0);
    sample.DeleteElement(10);
    sample.DeleteElement(20);
    sample.PrintAllElements();
}
}
```

Synchronizacja 2

- **System.Object**
- **System.MarshalByRefObject**
- **System.Threading.WaitHandle**
- **System.Threading.Mutex**
- **Konstruktor:**
 - `public Mutex();`
- **Metody:**
 - `public void ReleaseMutex();`
 - `public static bool WaitAll(WaitHandle[] waitHandles);`
 - `public static int WaitAny(WaitHandle[] waitHandles);`
 - `public virtual bool WaitOne();`


```
using System;
using System.Threading;

class Test
{
    private static Mutex mut = new Mutex();
    private const int numIterations = 1;
    private const int numThreads = 3;

    static void Main()
    {
        for(int i = 0; i < numThreads; i++)
        {
            Thread myThread = new Thread(new ThreadStart(MyThreadProc));
            myThread.Name = String.Format("Thread{0}", i + 1);
            myThread.Start();
        }
    }
    private static void MyThreadProc()
    {
        for(int i = 0; i < numIterations; i++)
        {
            UseResource();
        }
    }

    private static void UseResource()
    {
        mut.WaitOne();

        Console.WriteLine("{0} has entered the protected area",
            Thread.CurrentThread.Name);

        Thread.Sleep(500);

        Console.WriteLine("{0} is leaving the protected area\r\n",
            Thread.CurrentThread.Name);

        mut.ReleaseMutex();
    }
}
```

ThreadPool

- **Metody:**

- `public static void GetAvailableThreads(out int workerThreads, out int completionPortThreads);`
- `public static void GetMaxThreads(out int workerThreads, out int completionPortThreads);`
- `public static bool QueueUserWorkItem(WaitCallback callBack, object state);`
- `public static RegisteredWaitHandle RegisterWaitForSingleObject(WaitHandle waitObject, WaitOrTimerCallback callBack, object state, int millisecondsTimeoutInterval, bool executeOnlyOnce);`

- **Delegacje:**

- `[Serializable]`
`public delegate void WaitCallback(object state);`

```
using System;
using System.Threading;
public class Example {
    public static void Main() {
        // Queue the task.
        ThreadPool.QueueUserWorkItem(new WaitCallback(ThreadProc));

        Console.WriteLine("Main thread does some work, then sleeps.");
        // If you comment out the Sleep, the main thread exits before
        // the thread pool task runs. The thread pool uses background
        // threads, which do not keep the application running. (This
        // is a simple example of a race condition.)
        Thread.Sleep(1000);

        Console.WriteLine("Main thread exits.");
    }

    // This thread procedure performs the task.
    static void ThreadProc(Object stateInfo) {
        // No state object was passed to QueueUserWorkItem, so
        // stateInfo is null.
        Console.WriteLine("Hello from the thread pool.");
    }
}
```

```
using System;
using System.Threading;

public class TaskInfo {
    public RegisteredWaitHandle Handle = null;
    public string OtherInfo = "default";
}

public class Example {
    public static void Main(string[] args) {
        // The main thread uses AutoResetEvent to signal the
        // registered wait handle, which executes the callback
        // method.
        AutoResetEvent ev = new AutoResetEvent(false);

        TaskInfo ti = new TaskInfo();
        ti.OtherInfo = "First task";
        // The TaskInfo for the task includes the registered wait
        // handle returned by RegisterWaitForSingleObject. This
        // allows the wait to be terminated when the object has
        // been signaled once (see WaitProc).
        ti.Handle = ThreadPool.RegisterWaitForSingleObject(
            ev, new WaitOrTimerCallback(WaitProc), ti, 1000, false
        );

        // The main thread waits three seconds, to demonstrate the
        // time-outs on the queued thread, and then signals.
        Thread.Sleep(3100);
        Console.WriteLine("Main thread signals.");
        ev.Set();

        // The main thread sleeps, which should give the callback
        // method time to execute. If you comment out this line, the
        // program usually ends before the ThreadPool thread can execute.
        Thread.Sleep(1000);
        // If you start a thread yourself, you can wait for it to end
        // by calling Thread.Join. This option is not available with
        // thread pool threads.
    }
}
```

```

// The callback method executes when the registered wait times out,
// or when the WaitHandle (in this case AutoResetEvent) is signaled.
// WaitProc unregisters the WaitHandle the first time the event is
// signaled.
public static void WaitProc(object state, bool timedOut) {
    // The state object must be cast to the correct type, because
    // the signature of the WaitOrTimerCallback delegate specifies
    // type Object.
    TaskInfo ti = (TaskInfo) state;

    string cause = "TIMED OUT";
    if (!timedOut) {
        cause = "SIGNALLED";
        // If the callback method executes because the WaitHandle is
        // signaled, stop future execution of the callback method
        // by unregistering the WaitHandle.
        if (ti.Handle != null)
            ti.Handle.Unregister(null);
    }

    Console.WriteLine("WaitProc( {0} ) executes on thread {1};
        cause = {2}.", ti.OtherInfo,
        Thread.CurrentThread.GetHashCode().ToString(),
        cause
    );
}
}

```

Timer

- `System.Object`
 - `System.MarshalByRefObject`
 - `System.Threading.Timer`
- `public sealed class Timer :`
`MarshalByRefObject, IDisposable`
- `Metody:`
 - `public Timer(TimerCallback callback, object state, int dueTime, int period);`
 - `public bool Change(int dueTime, int period);`
- `Delegacje:`
 - `[Serializable]`
`public delegate void TimerCallback(object state);`

```
using System;
using System.Threading;

class TimerExampleState
{
    public int counter = 0;
    public Timer tmr;
}

class App
{
    public static void Main()
    {
        TimerExampleState s = new TimerExampleState();
        TimerCallback timerDelegate = new TimerCallback(CheckStatus);
        Timer timer = new Timer(timerDelegate, s, 1000, 1000);
        s.tmr = timer;
        while(s.tmr != null)
            Thread.Sleep(0);
        Console.WriteLine("Timer example done.");
    }

    static void CheckStatus(Object state)
    {
        TimerExampleState s =(TimerExampleState)state;
        s.counter++;
        Console.WriteLine("{0} Checking Status {1}.",
            DateTime.Now.TimeOfDay, s.counter);
        if(s.counter == 5)
        {
            (s.tmr).Change(10000, 100);
            Console.WriteLine("changed...");
        }
        if(s.counter == 10)
        {
            Console.WriteLine("disposing of timer...");
            s.tmr.Dispose();
            s.tmr = null;
        }
    }
}
```

Serializacja

- Zapisywanie obiektów do pamięci nieulotnej
- Przekazywanie obiektów pomiędzy aplikacjami
- Automatyczne zapisywanie i odtwarzanie obiektów, drzew obiektów, obsługa pętli odwołań
- Wymagany jest atrybut Serializable (nie jest dziedziczony)
- Domyślnie zapisywane są wszystkie pola, również prywatne
- Można wskazać pola które nie są zapisywane
- Można zastąpić domyślną procedurę serializacji implementując interfejs Serializable
- Można serializować do formatu binarnego lub SOAP

[Serializable]

```
public class MyObject {  
    public int n1 = 0;  
    public int n2 = 0;  
    public String str = null;  
}
```

```
MyObject obj = new MyObject();  
obj.n1 = 1;  
obj.n2 = 24;  
obj.str = "Some String";  
IFormatter formatter = new BinaryFormatter();  
Stream stream = new FileStream("MyFile.bin", FileMode.Create,  
    FileAccess.Write, FileShare.None);  
formatter.Serialize(stream, obj);  
stream.Close();
```

```
IFormatter formatter = new BinaryFormatter();  
Stream stream = new FileStream("MyFile.bin", FileMode.Open,  
    FileAccess.Read, FileShare.Read);  
MyObject obj = (MyObject) formatter.Deserialize(stream);  
stream.Close();
```

[Serializable]

```
public class MyObject  
{  
    public int n1;  
    [NonSerialized] public int n2;  
    public String str;  
}
```

```
[Serializable]
public class MyObject : ISerializable
{
    public int n1;
    public int n2;
    public String str;

    public MyObject()
    {
    }

    protected MyObject(SerializationInfo info, StreamingContext context)
    {
        n1 = info.GetInt32("i");
        n2 = info.GetInt32("j");
        str = info.GetString("k");
    }
    [SecurityPermissionAttribute(SecurityAction.Demand,
        SerializationFormatter=true)]
    public virtual void GetObjectData(SerializationInfo info,
StreamingContext context)
    {
        info.AddValue("i", n1);
        info.AddValue("j", n2);
        info.AddValue("k", str);
    }
}
```

```
[Serializable]
public class ObjectTwo : MyObject
{
    public int num;

    public ObjectTwo() : base()
    {
    }

    protected ObjectTwo(SerializationInfo si, StreamingContext context)
        : base(si, context)
    {
        num = si.GetInt32("num");
    }
    [SecurityPermissionAttribute(SecurityAction.Demand,
    SerializationFormatter=true)]
    public override void GetObjectData(SerializationInfo si,
    StreamingContext context)
    {
        base.GetObjectData(si, context);
        si.AddValue("num", num);
    }
}
```

```

using System;
using System.IO;
using System.Collections;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization;

[Serializable]
public sealed class Singleton : ISerializable
{
    private static readonly Singleton theOneObject = new Singleton();

    public String someString;
    public Int32 someNumber;

    private Singleton()
    {
        someString = "This is a string field";
        someNumber = 123;
    }

    public static Singleton GetSingleton()
    {
        return theOneObject;
    }

    void ISerializable.GetObjectData(
        SerializationInfo info, StreamingContext context)
    {
        info.SetType(typeof(SingletonSerializationHelper));
    }
}

[Serializable]
internal sealed class SingletonSerializationHelper : IObjectReference
{
    // GetRealObject is called after this object is deserialized.
    public Object GetRealObject(StreamingContext context)
    {
        return Singleton.GetSingleton();
    }
}

```

```
class App
```

```
[STAThread]
```

```
static void Main()
```

```
{
```

```
    FileStream fs = new FileStream("DataFile.dat", FileMode.Create);
```

```
    try
```

```
    {
```

```
        BinaryFormatter formatter = new BinaryFormatter();
```

```
        Singleton[] a1 = { Singleton.GetSingleton(), Singleton.GetSin
```

```
        // This displays "True".
```

```
        Console.WriteLine(
```

```
            "Do both array elements refer to the same object? " +
```

```
            (a1[0] == a1[1]));
```

```
        formatter.Serialize(fs, a1);
```

```
        fs.Position = 0;
```

```
        Singleton[] a2 = (Singleton[]) formatter.Deserialize(fs);
```

```
        // This displays "True".
```

```
        Console.WriteLine("Do both array elements refer to the same o
```

```
            + (a2[0] == a2[1]));
```

```
        // This displays "True".
```

```
        Console.WriteLine("Do all array elements refer to the same ob
```

```
            + (a1[0] == a2[0]));
```

```
    }
```

```
    catch (SerializationException e)
```

```
    {
```

```
        Console.WriteLine("Failed to serialize. Reason: " + e.Message
```

```
        throw;
```

```
    }
```

```
    finally
```

```
    {
```

```
        fs.Close();
```

```
    }
```

```
}
```