



2.2 Wybrane elementy C#, BCL

2.1 Microsoft .NET Framework: CLR, podzespoły, biblioteka klas

2.2 Elementy języka C#, BCL, typy uogólnione, obsługa XML

2.3 Dostęp do danych za pomocą ADO.NET, LINQ, Entity Framework

2.4 Wytwarzanie aplikacji .NET: podstawy ASP.NET

2.5 Usługi Web

- Dokumentowanie kodu
- Elementy C#
 - cechy wybranych typów: *System.String*, wyliczenia, klasy, struktury
 - błędy i wyjątki
 - interfejsy
 - delegaty i zdarzenia
 - metody rozszerzające, typy implikowane, typy anonimowe
- Elementy biblioteki klas bazowych (*Base Class Library*)
 - kolekcje, typy ogólne (Generics)
 - IO
 - Refleksje , atrybuty
 - Obsługa XML
 - Serializacja



Struktura programu C#

```
// Namespace Declaration
using System;
namespace MyProgram {
// helper class
class OutputClass
{
    string myString;
// Constructor
public OutputClass(string inputString)
{
    myString = inputString;
}
// Instance Method
public void printString()
{
    Console.WriteLine("{0}", myString);
}
}
// Program start class
class InteractiveWelcome
{
// Main begins program execution.
public static void Main(string[] args)
{
// Write to console/get input
Console.Write("What is your name?: ");
Console.Write("Hello, {0}! ", Console.ReadLine());
Console.WriteLine("Welcome to the C# Tutorial!");
// Instance of OutputClass
OutputClass outCl = new OutputClass("This is printed by the output class.");
// Call Output class' method
outCl.printString()
}
```



Variables in C#

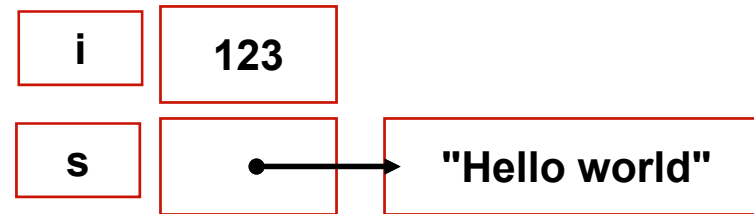
Type of Variable	Possible Contents
Non-nullable value type	A value of that exact type
Nullable value type	A null value or a value of that exact type
object	A null reference, a reference to an object of any reference type, or a reference to a boxed value of any value type
Class type	A null reference, a reference to an instance of that class type, or a reference to an instance of a class derived from that class type
Interface type	A null reference, a reference to an instance of a class type that implements that interface type, or a reference to a boxed value of a value type that implements that interface type
Array type	A null reference, a reference to an instance of that array type, or a reference to an instance of a compatible array type
Delegate type	A null reference or a reference to an instance of that delegate type



2.2.1 Type System - przykłady

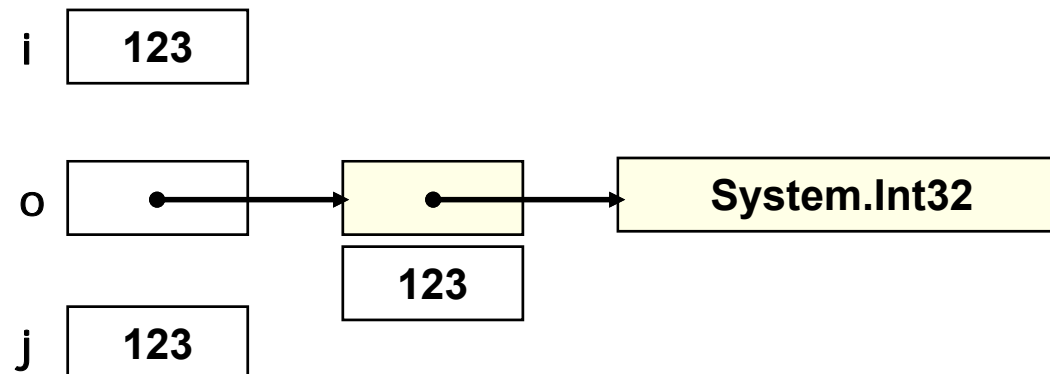
- Typy wartościowe
- Typy referencyjne

```
int i = 123;  
string s = "Hello world";
```



- Technika pakowania (Boxing and unboxing)

```
int i = 123;  
object o = i;  
int j = (int)o;
```





Typ referencyjny wbudowany: string (System.String)

Łańcuchy :

- łańcuch jest sekwencją wartości typu char
- porównywane są według wartości
- niezmiennie (ang. immutable) – zmodyfikowane są zapisywane pod innymi adresami niż oryginalne

```
public sealed class String : IComparable, ICloneable,
    IConvertible, IEnumerable{
    public char this[int index] {get;}
    public int Length {get;}
    public static int Compare(string strA, string strB);
    public static int CompareOrdinal(string strA, string
    strB);
    public static string Format(string format, object
    arg0);
    public int IndexOf(string);
    public int IndexOfAny(char[] anyOf);
    public int LastIndexOf(string value);
    public string PadLeft(int width, char c);
    public string[] Split(params char[] separator);
    public string Substring(int startIndex, int length);
    ...
}
```

```
string s = "Hi there.";
Console.WriteLine( "{0}", s.ToUpper() );           // Print uppercase copy
Console.WriteLine( "{0}", s );                     // String is unchanged
```



Typ referencyjny wbudowany - string (System.String)

Metody	Opis
Compare()	porównanie dwóch łańcuchów
Concat()	utworzenie nowego łańcucha z jednego lub większej liczby
Copy()	utworzenie nowego łańcucha przez przekopiowanie zawartości
Chars[]	indekser łańcucha
Insert()	zwraca nowy łańcuch z dostawionym nowym łańcuchem
Remove()	usunięcie z łańcucha określonej liczby znaków
Split()	rozbicie łańcucha na podłańcuchy przy założonym zbiorze ograniczników
StartsWith()	wskazuje czy łańcuch rozpoczyna się od określonych znaków
Substring()	wyłuskanie podłańcucha
ToLower()	zwraca kopię łańcucha składającego się z małych liter
Trim()	wyrzucenie określonego zbioru znaków z początku i końca łańcucha

```
Console.WriteLine("=> String concatenation:");  
string s1 = "Programming the ";  
string s2 = " PsychoDrill ";  
string s3 = string.Concat(s1, s2);  
Console.WriteLine(s3);  
Console.WriteLine();
```



Stałe łańcuchowe - obsługa znaków specjalnych

Łańcuchy standardowe i dosłowne

```
string rst1 = "Hi there!";  
string vst1 = @"Hi there!";
```

```
string rst2 = "It started, \"Four score and seven...\"";  
string vst2 = @"It started, \"\"Four score and seven...\"\"";
```

```
string rst3 = "Value 1 \t 5, Val2 \t 10";           // Interprets tab esc sequence  
string vst3 = @"Value 1 \t 5, Val2 \t 10";         // Does not interpret tab
```

```
string rst4 = "C:\\Program Files\\Microsoft\\";  
string vst4 = @"C:\Program Files\Microsoft\";
```

```
string rst5 = " Print \x000A Multiple \u000A Lines";  
string vst5 = @" Print  
Multiple  
Lines";
```



Class StringBuilder

Klasa `System.Text.StringBuilder`

- konstruowanie i przetwarzanie łańcuchów składowanych w buforze
- bardziej efektywne w przetwarzaniu łańcuchów

```
public sealed class StringBuilder {  
  
    public int Capacity {get; set;}  
    public int Length {get; set;}  
  
    StringBuilder Append(...);  
    StringBuilder AppendFormat(...);  
    StringBuilder Insert(int index, ...);  
    StringBuilder Remove(int startIndex, int  
        length);  
    StringBuilder Replace(char oldChar, char  
        newChar);  
  
    string ToString();  
}
```

```
using System.Text;  
StringBuilder sb = new StringBuilder("Hi there.");  
Console.WriteLine("{0}", sb); // Print string  
sb.Replace("Hi", "Hello");    // Replace a substring  
Console.WriteLine("{0}", sb); // Print changed string
```



Typy wyliczeniowe (System.Enum)

```
public abstract class Enum : IComparable,  
    IFormattable, IConvertible
```

- Typy zawierające zbiór nazwanych stałych
- Łatwiejsze czytanie kodu – przypisanie łatwo identyfikowalnych nazw do wartości;
- Łatwość pisania kodu – IntelliSense podpowiada listę możliwych wartości;
- Łatwość utrzymania kodu - pozwala zdefiniować zbiór stałych i zadeklarować zmienne, które będą akceptować wartości tylko z tego zbioru

```
Color a = Color.Red;  
Color b = Color.Green;  
Color c = Color.Blue;  
Console.WriteLine("Wartości typu Color: ");  
foreach(string s in Enum.GetNames(typeof(Color)))    funkcja GetNames z klasy System.Enum  
    Console.WriteLine(s);  
Console.WriteLine("Blue jest wartością typu Color ?: {0}",  
    Enum.IsDefined(typeof(Color), "Blue"));  
Console.WriteLine("Yellow jest wartością typu Color ?: {0}",  
    Enum.IsDefined(typeof(Color), "Yellow"));
```

```
enum Color { Red, Green, Blue };
```



Tablice

Struktury danych

- jednorodność - jest złożona z elementów tego samego typu zwanego typem podstawowym tablicy
- swobodny dostęp - wszystkie elementy tablicy są jednakowo dostępne - w dowolnej kolejności i w jednakowym czasie
- stała (niezmienna) ilość elementów
- ciągłość - zajmuje ciągły obszar pamięci
- **Pojedynczy element dostępny za pomocą indeksu**

- **Deklaracji zmiennej tablicowej**
 - `typ [] nazwa;`
- **Utworzenie tablicy - obiektu tablicowego**
 - `typ [] nazwa =
 new typ[ilosc_Elementow];`
 - lub**
 - `nazwa = new typ[liczba_Elementow];`



Operacje na tablicach

- **Metody Copy, Clone, CopyTo klasy Array**

```
int[] tab1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };  
int[] tab2 = {11,12,13,14,15,16,17,18,19 };  
Array.Copy(tab1,2,tab2,4,4);
```

```
foreach (int i in tab2)  
{  
    Console.WriteLine("{0}, ",i);  
}
```

11, 12, 13, 14, 3, 4, 5, 6, 19,

- **Metoda Sort klasy Array**

```
int[] tab1 = { 4, 1, 83, 41, 53, 36, 47, 18};  
Array.Sort(tab1);
```

```
foreach (int i in tab1)  
{  
    Console.WriteLine("{0}, ", i);  
}
```

1, 4, 18, 36, 41, 47, 53, 83,

- **Metoda Reverse klasy Array**

- Exists, FindLast, FindAll,
FindIndex, FindLastIndex, ...



2.2.2 Typy referencyjne użytkownika - klasy

```
[modyfikator_dostępu] class NazwaKlasy  
{  
    // ...  
}
```

Składowe klasy

Pola lub zmienne składowe

```
[modyfikator_dostępu] typ_pola nazwa_pola [= wyrażenie];
```

Metody składowe (funkcjonalność)

```
[modyfikator_dostępu] typ_zwracany nazwa_metody(argumenty)  
{  
    ciało_metody  
}
```



Kinds of members a class

Member	Description
Constants	Constant values associated with the class
Fields	Variables of the class
Methods	Computations and actions that can be performed by the class
Properties	Actions associated with reading and writing named properties of the class
Indexers	Actions associated with indexing instances of the class like an array
Events	Notifications that can be generated by the class
Operators	Conversions and expression operators supported by the class
Constructors	Actions required to initialize instances of the class or the class itself
Destructors	Actions to perform before instances of the class are permanently discarded
Types	Nested types declared by the class



Korzystanie z obiektu

- Tworzymy przy pomocy operatora **new**

```
Klasa1 zmienna1;  
zmienna1 = new Klasa1();
```

lub

```
Klasa1 zmienna1 = new Klasa1();
```

- Odwołanie się do składowych

```
zmienna1.X = 20;           //odwołanie się do pola  
zmienna1.Metoda1();       //wywołanie metody
```

- Ukrywanie informacji – modyfikatory dostępu

private	Dostępne tylko z metod danej klasy
public	Ogólnie dostępne
protected	dostępne dla klas i potomków klasy
internal	dostęp dla klas z pakietu
internal protected	dostęp dla klas z pakietu lub potomnych

Metody klasy **object**

- *ToString* - domyślna wersja tej metody zwraca w pełni kwalifikowaną nazwę klasy
- *Equals* - domyślnie porównuje referencje
- *GetHashCode* - wartość skrótu dla obiektu



Konstruktor

- Metoda klasy
- Metoda wywoływana tuż po utworzeniu obiektu
- Posiada taką samą nazwę jak klasa w której jest zdefiniowany
- Nie określamy wartości zwracanej
- Wywoływany przez operator *new*
- Można go przeciążyć

```
class NazwaKlasy{
    public NazwaKlasy(){ //konstruktor bezargumentowy
        ...
    }
    public NazwaKlasy(int arg1,double arg2){
        //konstruktor z argumentami
        ...
    }
}

NazwaKlasy x1 = new NazwaKlasy();
NazwaKlasy x1 = new NazwaKlasy(2, 3.3);
```



Klasy częściowe - wieloplikowe

```
//Plik1.cs
partial class Klasa1
{
    //częściowa definicja klasy
}
```

```
//Plik2.cs
partial class Klasa1
{
    //częściowa definicja klasy
}
```

Wszystkie części muszą być dostępne w czasie kompilacji
Klasy częściowe używane są w środowisku Visual Studio przez generatory kodu;
mogą być również przydatne w przypadku pracy grupowej



Sposoby przesyłania argumentów do metody

- Przesyłanie przez **wartość**

```
static void Nazwa(int x){...}
```

```
int a = 5;  
Nazwa(a);
```

- Przesyłanie przez **referencję**

```
static void Nazwa(ref int x){...}
```

```
int a = 5;  
Nazwa(ref a);
```

- Przesyłanie jako **parametr wyjściowy**

```
static void Nazwa(out int x){...}
```

```
int a;  
Nazwa(out a);
```



Właściwości (Properties)

- Definiuje „wirtualny” atrybut
 - get - właściwość do odczytu
 - set - właściwość do zapisu
 - argument o nazwie **value**

```
[modyfikator dostępu] typ Nazwa
{
    get{...}
    set{...}
}
```

```
class Student {
    ...
    private int numerIndeksu;
    //definicja właściwości
    public int NumerIndeksu {
        get { return numerIndeksu; }
        set { numerIndeksu = value; }
    }
}

Student s = new Student();
s.NumerIndeksu = 1234;
int nr = s.NumerIndeksu;
Console.WriteLine(s.NumerIndeksu);
```

- Pojęcie reprezentuje pewien atrybut klasy, ale w celu jego implementacji trzeba wywołać pewne funkcje
- Jedną wartość można w prosty sposób obliczyć na podstawie drugiej
- Dostęp do pola klasy musi być kontrolowany w celu zapewnienia spójności (poprawności) obiektu
- Ukrycie implementacji (typu)



Indekser

- Define property on **this** and add square brackets

```
public class Skyscraper {
    Story[] stories;
    public Story this [int index] {
        get { return stories[index]; }
        set { if (value!=null) {
                stories[index] = value;
            }
        }
    }
}

SkyScraper searsTower = new SkyScraper();
searsTower[155] = new Story("Observation Deck");
searsTower[0] = new Story("Entrance");
```



Struktury – struct

- są zawsze wartościami (na stosie, kopiowanie)
- mogą zawierać pola, interfejsy, funkcje składowe i konstruktory z argumentami
- można definiować struktury częściowe i definiować wewnątrz nich metody częściowe
- są zawsze zapieczętowane
- możemy sterować dostępem do składowych za pomocą modyfikatorów dostępu

Modyfikator	Opis
<code>public</code>	dostępne zewsząd (domyślny)
<code>private</code>	tylko wewnątrz struktury / klasy
<code>protected</code>	dla klas dziedziczących
<code>internal</code>	tylko w pakiecie
<code>protected internal</code>	dla klas dziedziczących w pakiecie



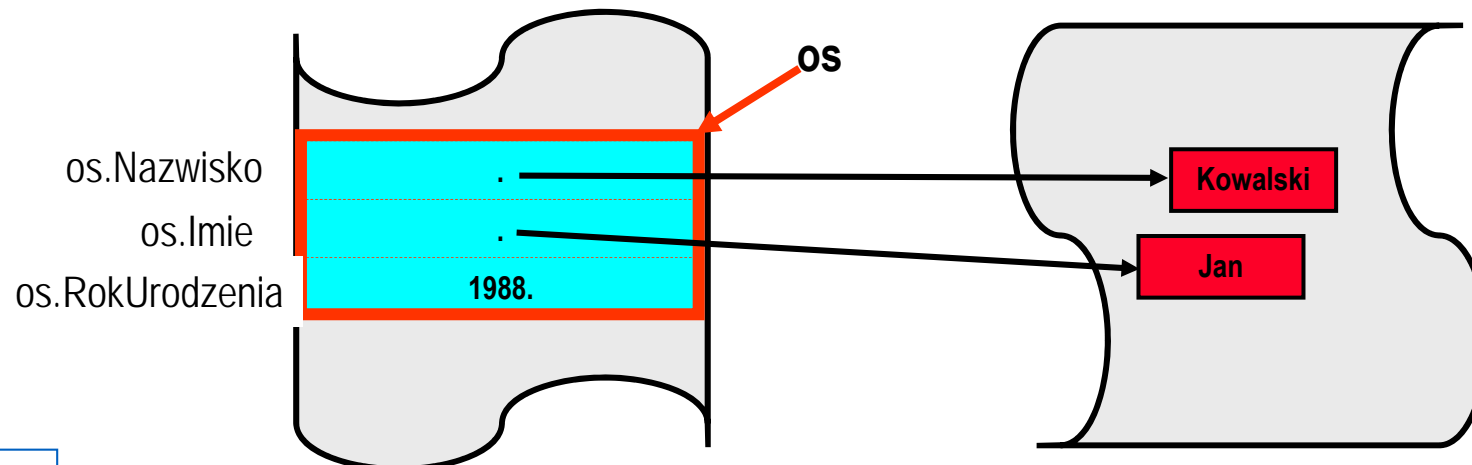
Struktury - przykład

- Definicja typu strukturalnego

```
struct Osoba{  
    public string Imie,Nazwisko;  
    public int RokUrodzenia;  
}
```

- Użycie typu strukturalnego

```
Osoba os;  
os.Imie = "Jan";  
os.Nazwisko = "Kowalski";  
os.RokUrodzenia = 1988;  
...  
Console.WriteLine("Pan(i) {0} {1}", os.Imie, os.Nazwisko);
```



Zarządzana sterła



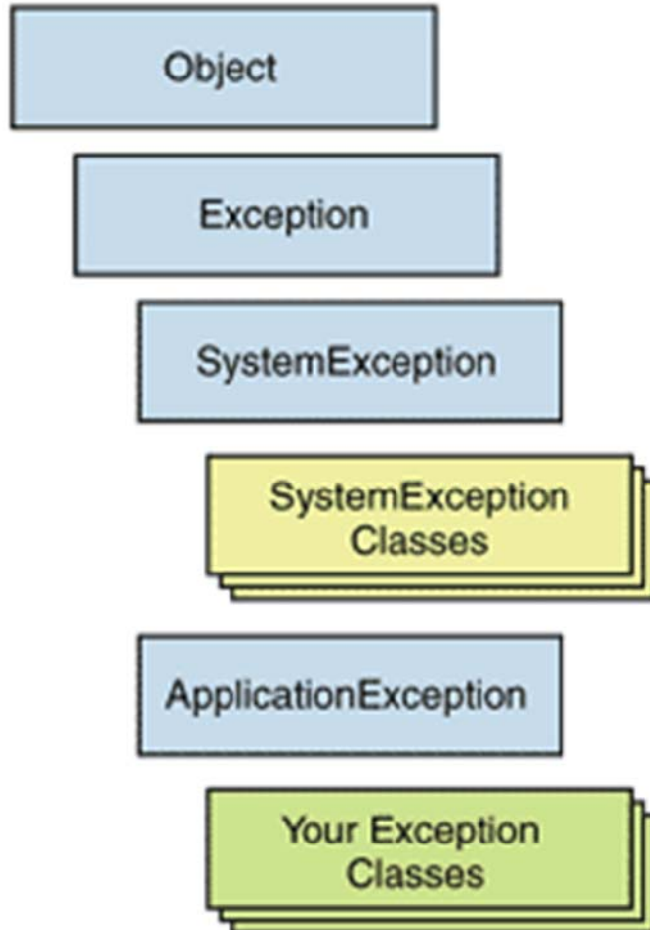
2.2.3 Exception Class Hierarchy

- **SystemException**

- Coding error, Operation error

- **ApplicationException**

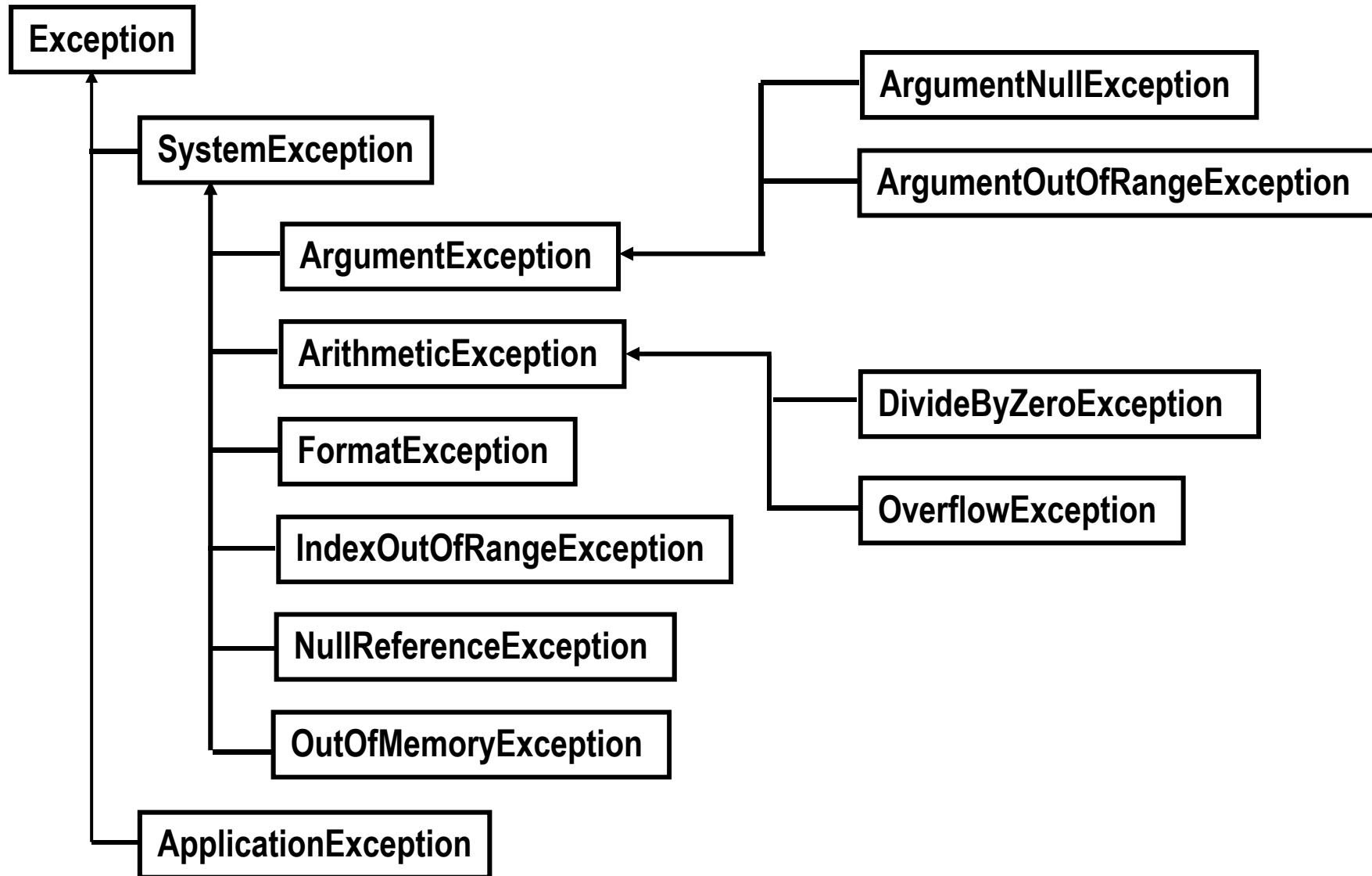
- Business logic violation



```
try
{
    // Some code that could throw an exception.
}
catch (TypeAException e)
{
    // Code to do any processing needed.
    // Rethrow the exception
    throw;
}
catch (TypeBException e)
{
    // Code to do any processing needed.
    // Wrap the current exception in a more relevant
    // outer exception and rethrow the new exception.
    throw(new TypeCException(strMessage, e));
}
finally
{
    // Code that gets executed regardless of whether
    // an exception was thrown.
}
```



Typy wyjątków





Obsługa wyjątków - użycie bloków try i catch

```
using System;
```

```
class ExceptionTestClass
```

```
{
```

```
    public static void Main()
```

```
    {
```

```
        int x = 0;
```

```
        try
```

```
        {
```

```
            int y = 100/x;
```

```
        }
```

```
        catch (ArithmeticException e)
```

```
        {
```

```
            Console.WriteLine("ArithmeticException Handler: {0}", e.ToString());
```

```
        }
```

```
        catch (Exception e)
```

```
        {
```

```
            Console.WriteLine("Generic Exception Handler: {0}", e.ToString());
```

```
        }
```

```
    }
```

```
}
```

```
/*
```

This code example produces the following results:

```
ArithmeticException Handler: System.DivideByZeroException: Attempted to divide by zero.
```

```
    at ExceptionTestClass.Main()
```

```
*/
```

Wyjątki klas bardziej szczegółowych powinny być przechwytywane wcześniej



2.2.4 Interfejs

- Zbiór funkcji pod wspólną nazwą
- Słowo kluczowe *interface*
- Same deklaracje - brak implementacji
- Składowe interfejsu mogą być metodami, właściwościami, zdarzeniami lub indeksami
- Wszystkie składowe publiczne (przez domyślność)
- Interfejs może dziedziczyć po wielu interfejsach
- Klasa (interfejs) może implementować wiele interfejsów
- Klasa musi implementować wszystkie metody swoich bazowych interfejsów

```
interface IControl
{
    void Paint();
}
interface ITextBox: IControl
{
    void SetText(string text);
}
interface IListBox: IControl
{
    void SetItems(string[] items);
}
interface IComboBox: ITextBox, IListBox {}

interface IDataBound
{
    void Bind(Binder b);
}
public class EditBox: IControl, IDataBound
{
    public void Paint() {...}
    public void Bind(Binder b) {...}
}
```



Interfejs - przykład

```
EditText editBox = new EditText();  
IControl control = editBox;  
IDataBound dataBound = editBox;
```

```
object obj = new EditText();  
IControl control = (IControl)obj;  
IDataBound dataBound = (IDataBound)obj;
```

```
interface INazwa{  
    void f();  
    String Wlasciwosc{  
        get;  
        set;  
    }  
    event EventHandler zdarzenie;  
    //int i; - błąd  
}
```

Klasę implementującą jeden lub wiele interfejsów można traktować jako klasę należącą do wielu typów. Może być postrzegana jak **instancja typu każdego z implementowanych interfejsów**.

Możliwość traktowania odmiennych klas w podobny sposób, jeśli implementują ten sam interfejs

```
class Klasa : INazwa {  
    String s;  
    public virtual void f(){...}  
    public virtual String Wlasciwosc {  
        get {return s;}  
        set {s = value;}  
    }  
    public virtual event  
        EventHandler zdarzenie;  
}  
//Słowo virtual jest opcjonalne
```



Jawna implementacja interfejsu

```
interface Interfejs1 {
    void f();
}

interface Interfejs2 {
    void f();
}

class Klasa : Interfejs1, Interfejs2 {
    public void f() {
        Console.WriteLine(
            "Implementacja w sposób niejawny");
    }

    void Interfejs2.f() {
        Console.WriteLine(
            "Implementacja w sposób jawny");
    }
}
```

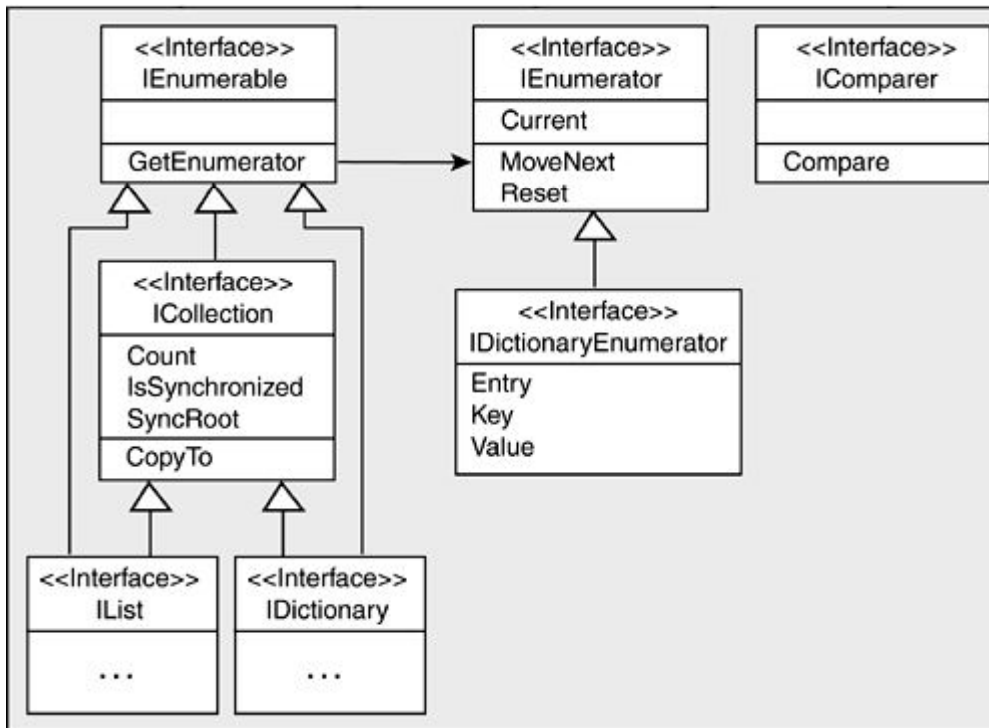


Interfejsy standardowe - *System.Collections*

ICollection	Interfejs bazowy dla wszystkich klas kolekcji.
IComparer	Metody do porównywania dwóch obiektów
IDictionary	Reprezentacja danych w formie kolekcji par klucz-wartość
IEnumerator, IEnumerable	Iteracyjne przeszukiwanie elementów kolekcji
IHashCodeProvider	Generowanie skrótów dla obiektów
ICollection	Operacje na listach
...	



Diagram interfejsów System.Collections



```
// implementacja IEnumerable
public IEnumerator GetEnumerator ( )
{
    return ( IEnumerator ) this ;
}

// implementacja IEnumerator
public void Reset ( ) { .... }
public bool MoveNext ( ) { ... }
public object Current ( )
{
    get { .... }
}
}
```

Interface ICollection, Interface ICollection<T>

- **Basic interface for collections**

```
public interface ICollection : IEnumerable {
// Properties
int Count {get;} // number of elements
bool IsSynchronized {get;} // collection synchronised?
object SyncRoot {get;} // returns object for synchronisation
void CopyTo(Array a, int index); // copies the elements into array (starting at position index)
...
}
```



Interfejsy ogólne: IEnumerable and IEnumerator (1)

- Anything which is enumerable is represented by interface *IEnumerable*
- *IEnumerator* realizes an iterator

```
interface IEnumerable {  
    IEnumerator GetEnumerator();  
}
```



```
interface IEnumerator {  
    object Current {get;}  
    bool MoveNext();  
    void Reset();  
}
```

- **IEnumerable<T>**
 - **IEnumerator<T> GetEnumerator()**
- **IEnumerator<T>**
 - **T Current { get; }**



IComparable and IComparer - sorting

Porównanie obiektów

IComparable is interface for types with order

```
public interface IComparable {
    int CompareTo(object obj); // <0 if this < obj, 0 if this == obj, >0 if this > obj
}

public interface IComparable<T> {
    int CompareTo(T obj); // <0 if this < obj, 0 if this == obj, >0 if this > obj
}
```

IComparer is interface for the realization of compare operators

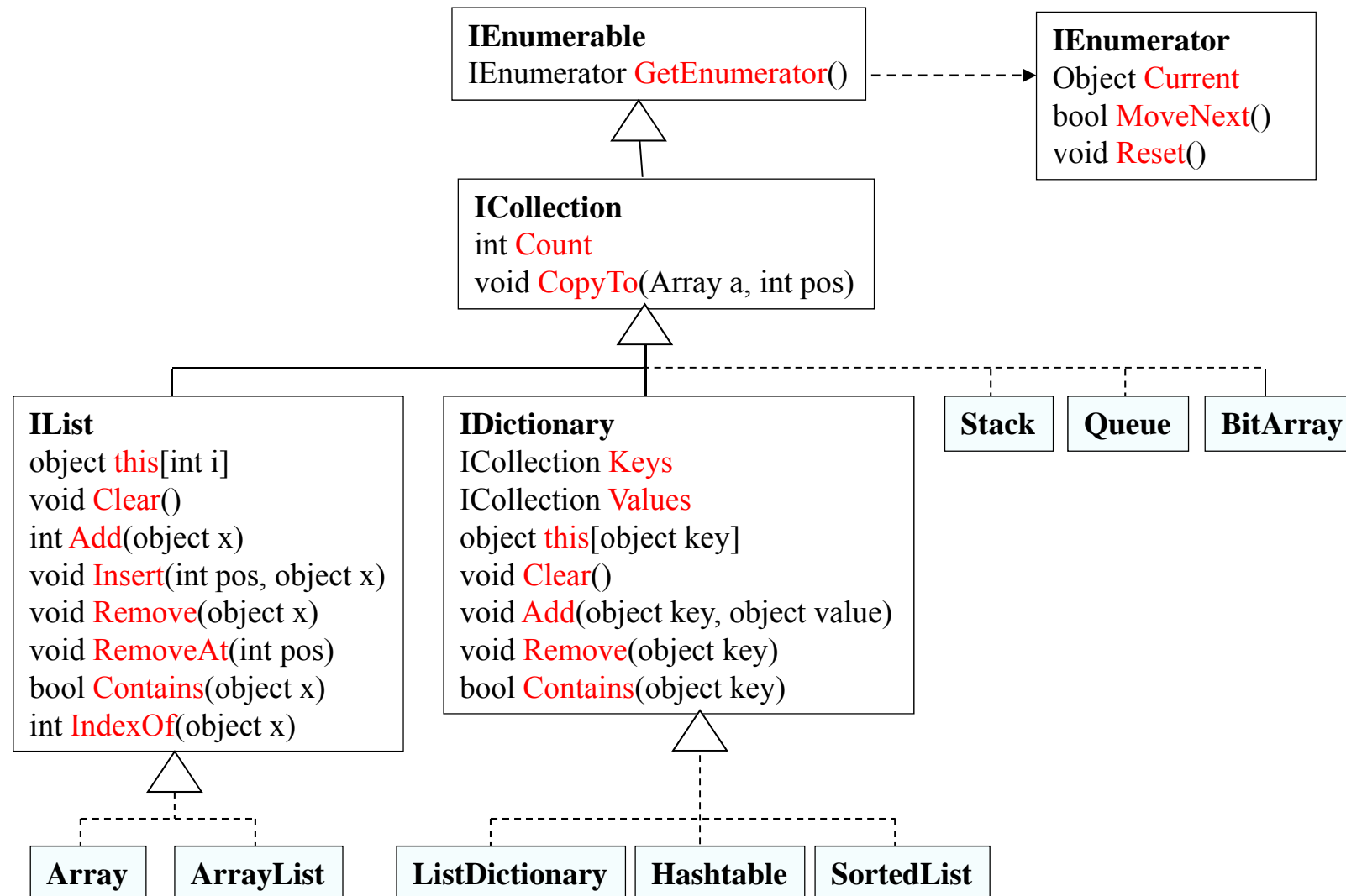
```
public interface IComparer {
    int Compare(object x, object y); // <0 if x < y, 0 if x == y, >0 if x > y
}

public interface IComparer<T> {
    int Compare(T x, T y); // <0 if x < y, 0 if x == y, >0 if x > y
}
```

- **IComparable<T>**
 - int CompareTo(T other)
- **IComparer<T>**
 - int Compare(T x, T y)



2.2.5 Klasy kolekcji - *System.Collections*





System.Collections - przykład

- ArrayList

```
using System;
using System.Collections;
namespace Kolekcje
{
class Element
{
public int Liczba;
public Element(int liczba)
{
this.Liczba = liczba;
}
}
}
```

```
class Program
{
static void Main(string[] args)
{
Element element_1 = new Element(10);
Element element_2 = new Element(20);
ArrayList lista = new ArrayList();
lista.Add(element_1);
lista.Add(element_2);
Console.WriteLine("Elementy: ");
foreach (Element element in lista)
{
Console.WriteLine(element.Liczba + " ");
}
Console.ReadKey();
}
}
```



Typy ogólne, uniwersalne (Generics)

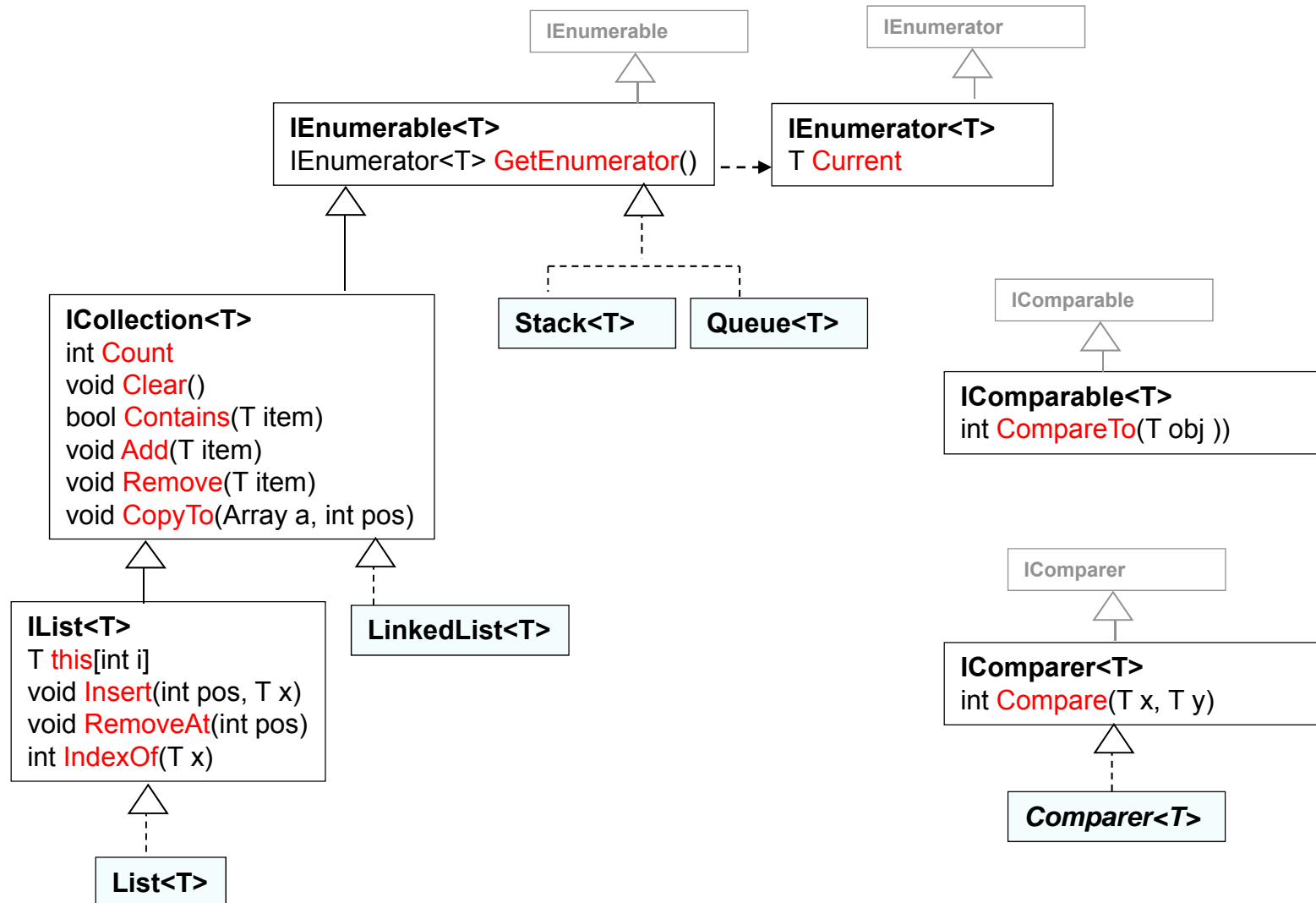
- Typy uniwersalne mogą być stosowane do implementacji klas, struktur, interfejsów i metod. Obsługują wiele typów danych. Parametry typów są umieszczone w nawiasach ostrych <> w deklaracji uniwersalnego typu.

```
class Klasa {  
    public static void Swap<T>(ref T a, ref T b) {  
        T tmp = a;  
        a = b;  
        b = tmp;  
    }  
    public static T Max<T>(T a, T b)  
        where T:      IComparable  
    {  
        if(a.CompareTo(b) > 0)  
            return a;  
        return b;  
    }  
}
```



System.Collections.Generic

System.Collections.Generic dostarcza klas uniwersalnych dla kolekcji.





System.Collections.Generic - przykład

```
using System;
using System.Collections.Generic;

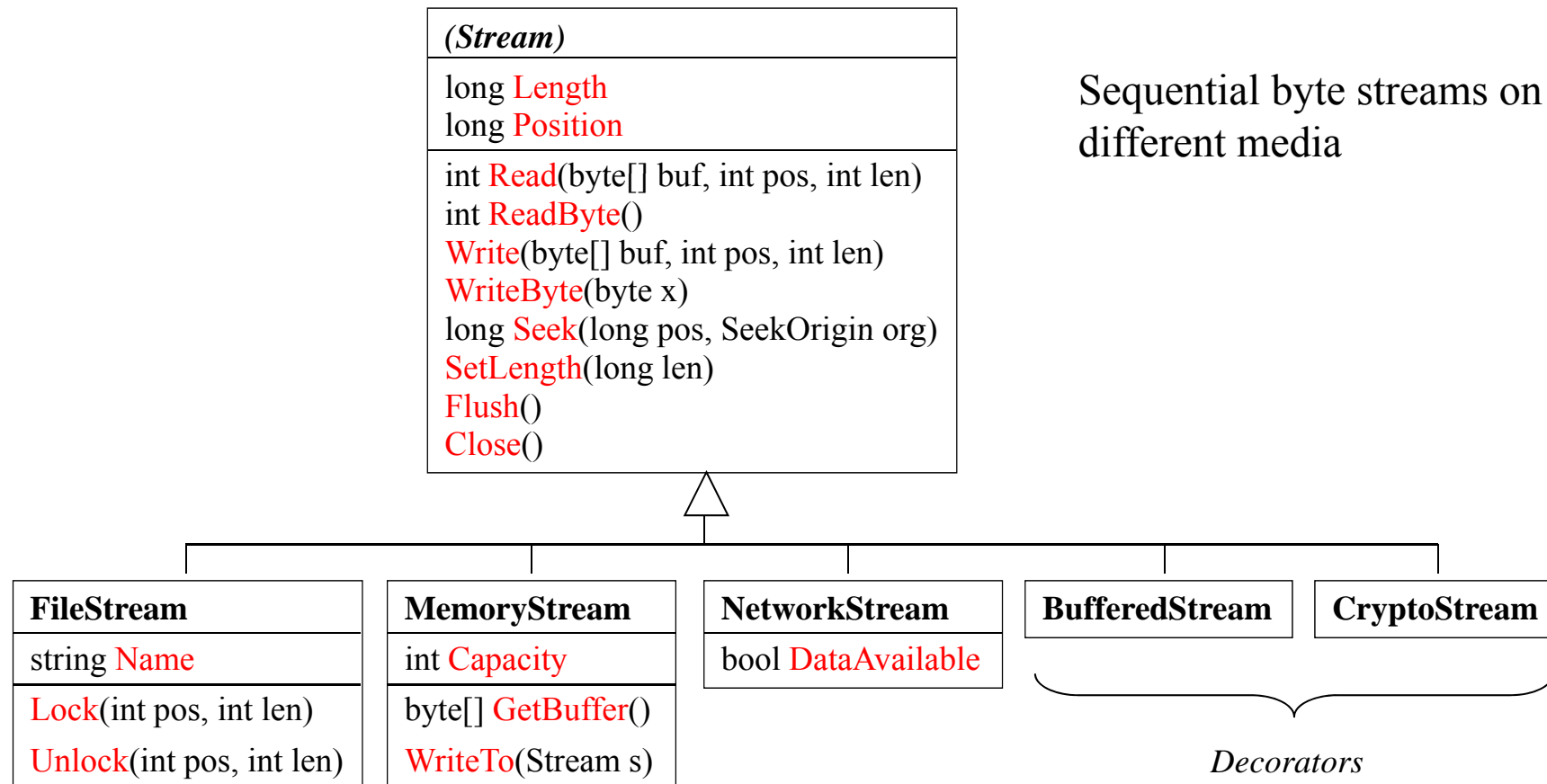
namespace KolekcjeGeneryczne
{
    class Program
    {
        static void Main(string[] args)
        {
            List<int> lista = new List<int>();
            lista.Add(2);
            lista.Add(5);

            int suma = lista[0] + lista[1];

            Console.WriteLine("Suma wynosi: " + suma);
            Console.ReadKey();
        }
    }
}
```



2.2.6 Strumienie *System.IO*

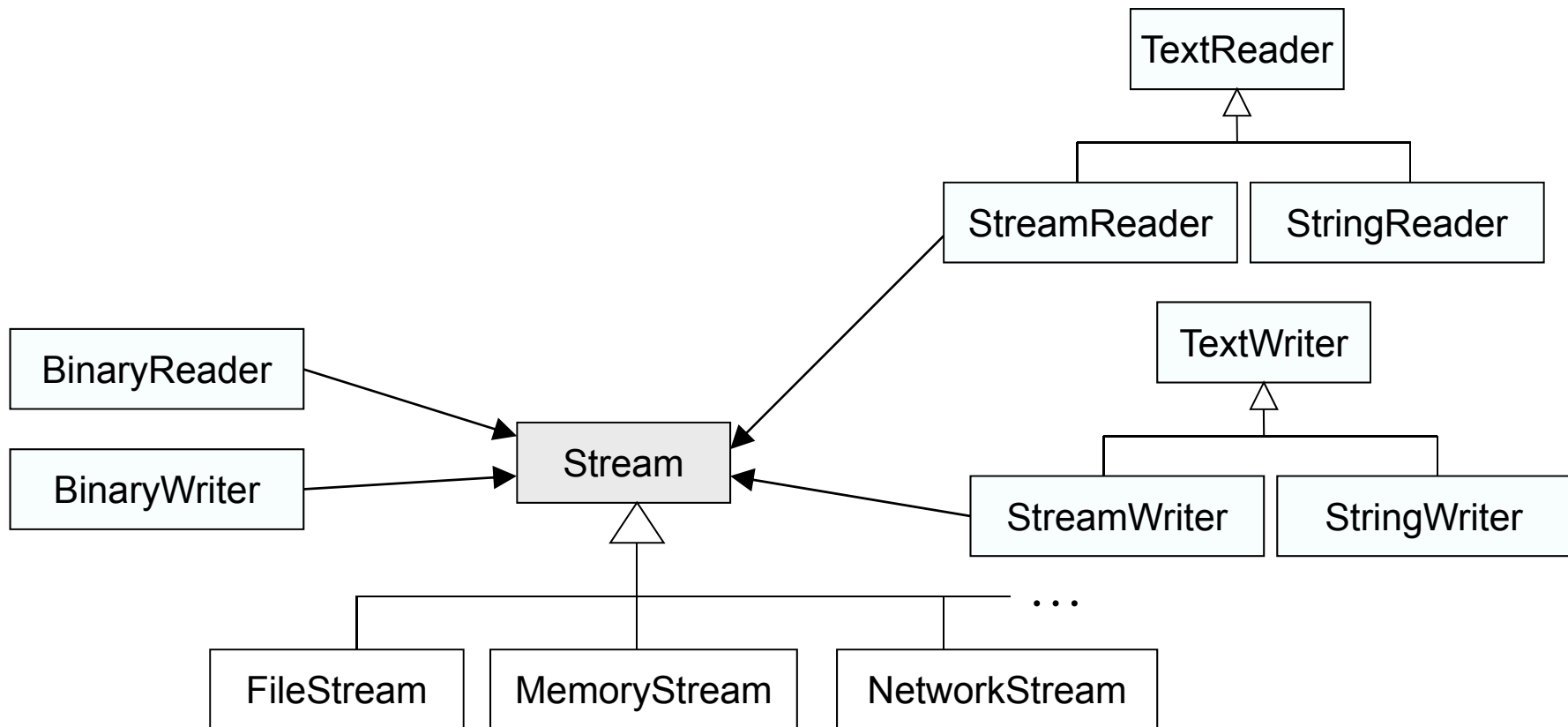


- **Podstawowe operacje na strumieniach:**

- **czytanie danych (reading)** - pobieranie danych ze strumienia i umieszczanie ich w pewnej strukturze danych
- **zapis danych (writing)** - wstawienie danych do strumienia z pewnej struktury danych
- **ustawienie bieżącej pozycji w strumieniu (seeking)**



Klasy używane do odczytu/zapisu z/do strumienia





Class Stream

```
public abstract class Stream : MarshalByRefObject, IDisposable
```

```
{  
    public abstract bool CanRead { get; }  
    public abstract bool CanSeek { get; }  
    public abstract bool CanWrite { get; }  
  
    public abstract int Read(out byte[] buff, int offset, int count);  
    public abstract void Write(byte[] buff, int offset, int count);  
    public virtual int ReadByte();  
    public virtual void WriteByte(byte value);  
  
    public virtual IAsyncResult BeginRead(...);  
    public virtual IAsyncResult BeginWrite(...);  
    public virtual int EndRead(...);  
    public virtual int EndWrite(...);  
  
    public abstract long Length { get; }  
    public abstract long Position { get; set; }  
    public abstract long Seek(long offset, SeekOrigin origin);  
  
    public abstract void Flush();  
    public virtual void Close();  
    ...  
}
```

- Elementary properties of stream
- Synchronous reading and writing
- Asynchronous reading and writing
- Length and actual position
- Positioning
- Flush and close



FileStream - przykład

```
using System;
using System.IO;

class Test {
    static void Copy(string from, string to, int pos) {
        try {
            FileStream sin = new FileStream(from, FileMode.Open);
            FileStream sout = new FileStream(to, FileMode.Create);
            sin.Seek(pos, SeekOrigin.Begin); // if Seek after end of file then
                                           // pos = end of file

            int ch = sin.ReadByte();
            while (ch >= 0) {
                sout.WriteByte((byte)ch);
                ch = sin.ReadByte();
            }
            sin.Close();
            sout.Close();
        } catch (FileNotFoundException e) {
            Console.WriteLine("-- file {0} not found", e.FileName);
        }
    }

    static void Main(string[] arg) {
        Copy(arg[0], arg[1], 10);
    }
}
```



Writer Classes

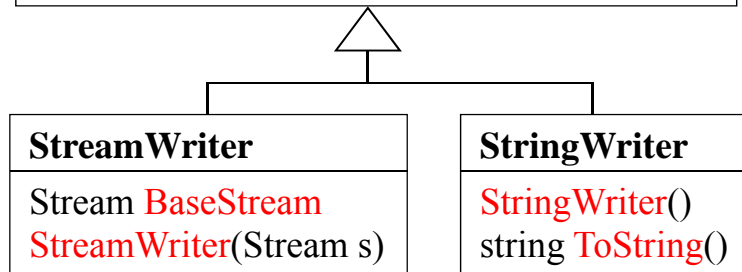
(*TextWriter*)

string **NewLine**
Write(bool b)
Write(char c)
Write(int i)
...
Write(string format, params object[] arg)
WriteLine()
WriteLine(bool b)
WriteLine(char c)
WriteLine(int i)
...
WriteLine(string format, params object[] arg)
Flush()
Close()

BinaryWriter

Stream **BaseStream**
BinaryWriter(Stream s)
BinaryWriter(Stream s, Encoding e)
Write(bool b)
Write(char c)
Write(int i)
...
long **Seek**(int pos, SeekOrigin org)
Flush()
Close()

Output of standard types in
binary format



formatted text output

```
using System;
using System.IO;
class Test {
    // writes command line arguments to a file
    static void Main(string[] arg) {
        FileStream s = new FileStream("xxx.txt",
            FileMode.Create);
        StreamWriter w = new StreamWriter(s);
        for (int i = 0; i < arg.Length; i++)
            w.WriteLine("arg[{0}] = {1}", i, arg[i]);
        w.Close();
    }
}
```



Operacje na strukturze systemu plików

- **Directory**
 - służy do bezpośrednich operacji na plikach i katalogach
- **File**
 - udostępnia metody do operacji na plikach
- **Path**
 - operacje na tekście zawierającym informacje o ścieżce dostępu do pliku lub katalogu
- **FileSystemWatcher**
 - ustawienie kontroli na pliku lub katalogu



Class Directory

```
public sealed class Directory {  
    public static DirectoryInfo CreateDirectory(string path); // creates directories and subdirectories  
    public static void Move(string src, string dest); // moves directory src to dest  
    public static void Delete(string path); // deletes an empty directory  
    public static void Delete(string path, bool recursive); // deletes directory with contents  
    public static bool Exists(string path); // checks if directory exists  
    public static string[] GetFiles(string path); // returns all file names in path  
    public static string[] GetFiles(string path, string searchPattern);  
    public static string[] GetDirectories(string path); // returns all directory names in path  
    public static string[] GetDirectories(string path, string searchPattern);  
    public static DirectoryInfo GetParent(string path); // returns the parent directory  
    public static string GetCurrentDirectory(); // returns current working directory  
    public static void SetCurrentDirectory(string path); // sets current working directory  
    public static string[] GetLogicalDrives(); // returns names of logical drives (e.g. "c:\")  
    public static DateTime GetCreationTime(string path); // returns creation date & time  
    public static DateTime GetLastAccessTime(string path);  
    public static DateTime GetLastWriteTime(string path);  
    public static void SetCreationTime(string path, DateTime t); // sets creation date & time  
    public static void SetLastAccessTime(string path, DateTime t);  
    public static void SetLastWriteTime(string path, DateTime t);  
}
```



Class DirectoryInfo

```
public sealed class DirectoryInfo : FileSystemInfo {
    //----- constructor
    public DirectoryInfo(string path); // path specifies the directory
    //----- properties
    public override string Name { get; } // returns directory name without the path
    public override bool Exists { get; } // indicates if this directory exists
    public DirectoryInfo Parent { get; } // returns the parent directory
    public DirectoryInfo Root { get; } // returns the root directory
    //----- methods
    public void Create(); // create a new directory, if it does not exist
    public DirectoryInfo CreateSubdirectory(string path); // creates a subdirectory
    public void MoveTo(string destDir); // moves this directory to destDir
    public void Delete(); // deletes this directory, if it is empty
    public void Delete(bool recursive); // deletes this directory and its contents
    public FileInfo[] GetFiles(); // returns all files in this directory
    public FileInfo[] GetFiles(string pattern); // returns matching files in this directory
    public DirectoryInfo[] GetDirectories(); // returns all directories in this directory
    public DirectoryInfo[] GetDirectories(string pattern); // returns all matching directories
    public FileSystemInfo[] GetFileSystemInfos(); // returns all files and directories
    public FileSystemInfo[] GetFileSystemInfos(string pattern); // returns files and directories for pattern
    public override ToString(); // returns the path given in the constructor
}
```



DirectoryInfo - przykład

```
using System;
using System.IO;
namespace DirectoryInfo1
{
    class Test {

        // list all files in the path to the current directory
        static void Main(string[] args) {
            DirectoryInfo d = new DirectoryInfo(".");
            while (d != null) {
                Console.WriteLine(d.FullName);
                FileInfo[] files = d.GetFiles();
                foreach (FileInfo f in files)
                    Console.WriteLine("  {0}, created: {1}", f.Name,
                                      f.CreationTime.ToString("D"));
                d = d.Parent;
            }
        }
    }
}
```



Class File

```
public sealed class File {
    public static FileStream Open(string path, FileMode mode);
    public static FileStream Open(string path, FileMode m, FileAccess a);
    public static FileStream Open(string p, FileMode m, FileAccess a, FileShare s);
    public static FileStream OpenRead(string path);
    public static FileStream OpenWrite(string path);
    public static StreamReader OpenText(string path); // returns Reader for reading text
    public static StreamWriter AppendText(string path); // returns Writer for appending text
    public static FileStream Create(string path); // create a new file
    public static FileStream Create(string path, int bufferSize);
    public static StreamWriter CreateText(string path);
    public static void Move(string src, string dest);
    public static void Copy(string src, string dest); // copies file src to dest
    public static void Copy(string src, string dest, bool overwrite);
    public static void Delete(string path);
    public static bool Exists(string path);
    public static FileAttributes GetAttributes(string path);
    public static DateTime GetCreationTime(string path);
    public static DateTime GetLastAccessTime(string path);
    public static DateTime GetLastWriteTime(string path);
    public static void SetAttributes(string path, FileAttributes fileAttributes);
    public static void SetCreationTime(string path, DateTime creationTime);
    public static void SetLastAccessTime(string path, DateTime lastAccessTime);
    public static void SetLastWriteTime(string path, DateTime lastWriteTime);
}
```

```
FileInfo {
    FileInfo(string path)
    string Name
    string FullName
    string DirectoryName
    DirectoryInfo Directory
    int Length
    FileAttributes Attributes
    DateTime CreationTime
    bool Exists
    FileStream OpenRead()
    FileStream OpenWrite()
    FileStream Create()
    StreamReader OpenText()
    StreamWriter CreateText()
    Delete()
    CopyTo(string path)
    MoveTo(string path)
    ...
}
```



2.2.7 Delegacje i zdarzenia (delegate,event)

- Definiuje nowy typ referencyjny
- Zastępuje wskaźniki do funkcji
- Sygnatura i typ wartości zwracanej delegacji musi być identyczny jak funkcji którą reprezentuje
- Dwa typy delegacji
 - *System.Delegate*
 - *System.MulticastDelegate*
- Delegacja – metody
 - Invoke - ()
 - Combine - +=
 - Remove - -=

```
delegate void myDelegate(int a, int b) ;  
myDelegate operation = new myDelegate(Add) ;  
operation += new myDelegate(Multiply) ;  
operation(1,2) ;
```



Delegacje

- **Delegacja** (*delegate*) - to specjalny **typ danych** umożliwiający „przechowywanie” referencji do metody o wspólnej sygnaturze.
- **Cel delegacji** : pozwala wywołać dowolną metodę (o sygnaturze zgodnej z sygnaturą delegacji) z dowolnego obiektu (~ jak wskaźniki do funkcji w C, C++);

– obsługa wywołań zwrotnych (*callback*)

Obsługa zdarzeń



- **Deklarowanie delegatów**

- Sygnatura delegacji jest taka sama jak sygnatura metody, którą można wywołać za pośrednictwem tej delegacji; jest poprzedzona słowem kluczowym **delegate**.

```
public delegate void AlarmEventHandler(object sender, EventArgs e);
```



Definiowanie i używanie delegacji

- **Tworzenie i wywoływanie delegatów**
 - po zadeklarowaniu delegata należy stworzyć egzemplarz klasy delegata i przypisać do niego metodę
 - następnie można wywołać określoną funkcję posługując się zmienną typu delegata.
 - środowisko .NET obsługuje właściwe działanie delegacji

- **Obiekt delegacji pozwala na pośrednie wywołanie metody**
 - zawiera referencję do metod
 - wszystkie metody wywoływane przez ten sam delegat muszą mieć takie same parametry i zwracać ten sam typ



Przykład

Declaration of a delegate type

```
delegate void Notifier (string sender); // ordinary method signature  
// with the keyword delegate
```

Declaration of a delegate variable

```
Notifier greetings;
```

Assigning a method to a delegate variable

```
void SayHello(string sender) {  
    Console.WriteLine("Hello from " + sender);  
}
```

```
greetings = new Notifier(SayHello); // or just: greetings = SayHello;
```

Calling a delegate variable

```
greetings("John"); // invokes SayHello("John") => "Hello from John"
```



Assigning Different Methods

Every matching method can be assigned to a delegate variable

```
void SayGoodBye(string sender) {  
    Console.WriteLine("Good bye from " + sender);  
}
```

```
greetings = SayGoodBye;
```

```
greetings("John");           // SayGoodBye("John") => "Good bye from John"
```

Note

- A delegate variable can have the value *null* (no method assigned).
- If *null*, a delegate variable must not be called (otherwise exception).
- Delegate variables are first class objects: can be stored in a data structure, passed as a parameter, etc.



Multicast Delegates

A delegate variable can hold multiple methods at the same time

```
Notifier greetings;  
greetings = SayHello;  
greetings += SayGoodBye;
```

```
greetings("John");      // "Hello from John"  
                        // "Good bye from John"
```

```
greetings -= SayHello;
```

```
greetings("John");      // "Good bye from John"
```



Zdarzenia

- Źródło zdarzeń - ten który publikuje
- Odbiorca zdarzeń - subskrybent
- Wzorzec projektowy obserwator

```
class NazwaKlasy
{
    ...
    public event NazwaDelegacji NazwaZdarzenia;
    ...
}
```

- Definiuje dwie metody publiczne
 - Add
 - Remove
- Definiuje jedną metodę protected
 - Invoke



Event = Special Delegate Field

```
class Model {  
    public event Notifier notifyViews;  
    public void Change() { ... notifyViews("Model"); }  
}
```

```
class View {  
    public View(Model m) { m.notifyViews += Update; }  
    void Update(string sender) { Console.WriteLine(sender + " was changed"); }  
}
```

```
class Test {  
    static void Main() {  
        Model model = new Model();  
        new View(model); ...  
        model.Change();  
    }  
}
```

Why events instead of normal delegate variables?

- Only the class that declares the event can fire it (better encapsulation)
- Other classes may change event fields only with += or -=



Event Handling in the .NET Library

In the .NET library delegates for event handling have the following signature

```
delegate void SomeEvent (object source, MyEventArgs e);
```

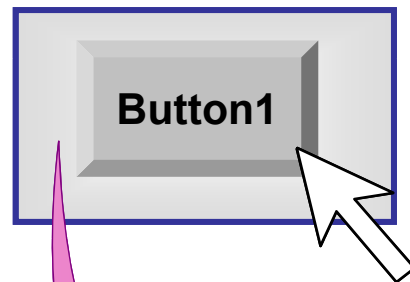
Result type: **void**

- 1. parameter: Sender of the event (type *object*)
- 2. parameter: event (subclass of *System.EventArgs*)

```
public class EventArgs {  
    ...  
}
```



Event Model in the .NET Framework



Invokes the delegate

```
this.button1.Click += new  
System.EventHandler(this.button1_Click);
```

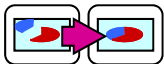
```
private void button1_Click(object sender,  
System.EventArgs e)  
{  
...  
}
```

Delegate calls the associated procedure

Application-level Event Handling
– Web Forms

Delegate

- Delegate Model
 - Connect event sender and event receiver
 - Single and multicast delegates
- Event Delegates Are Multicast
- Event Wiring
 - Register event handler with event sender





Events

Example: Component-Side

- Define the event signature as a delegate

```
public delegate void EventHandler(object sender,  
                                EventArgs e);
```

- Define the event and firing logic

```
public class Button {  
    public event EventHandler Click;  
  
    protected void OnClick(EventArgs e) {  
        // This is called when button is clicked  
        if (Click != null) Click(this, e);  
    }  
}
```



Events

Example: User-Side

- Define and register an event handler

```
public class MyForm: Form {
    Button okButton;

    static void OkClicked(object sender, EventArgs e) {
        ShowMessage("You pressed the OK button");
    }

    public MyForm() {
        okButton = new Button(...);
        okButton.Caption = "OK";
        okButton.Click += new EventHandler(OkClicked);
    }
}
```



Metody anonimowe

- Konstrukcja ta umożliwia utworzenie obiektu delegacji bez konieczności wcześniejszego definiowania metody

```
delegate void MojaDelegacja(string s, int i);
```

```
MojaDelegacja d1 = new MojaDelegacja(Test.F);  
d1("wywołanie", 1);
```

```
Test t1 = new Test() { Nazwa = "Obiekt1" };  
MojaDelegacja d2 = new MojaDelegacja(t1.G);  
d2("wywołanie", 2);
```

```
MojaDelegacja d4 = delegate(string s1, int x) {  
    Console.WriteLine("Metoda anonimowa z parametrami {0} oraz {1}", s1, x);  
};
```

```
d4("wywołanie", 5);  
class Test {  
    public string Nazwa { set; get; }  
    public static void F(string napis, int n) {  
        Console.WriteLine("Metoda statyczna: {0} oraz {1}", napis, n);  
    }  
    public void G(string napis, int n) {  
        Console.WriteLine("Metoda zwykła: {0} oraz {1}" + ", na rzecz obiektu {2}", napis, n, Nazwa);  
    }  
}
```



2.2.8 Metody rozszerzające, niejawna ścisła kontrola typów, typy anonimowe, wyrażenia lambda

VB9

XML Literals

Relaxed Delegates

Anonymous Types

Extension Methods

Lambdas

Object Initialisers

Implicit Typing

Partial Methods

C# 3.0

Collection Initialisers

Automatic Properties



Extension Methods

Extension methods enable us to **"add" methods to existing types** without creating a new derived type, recompiling, or otherwise modifying the original type.

Extension methods are a special kind **of static method**, but they are called as if they were instance methods on the extended type.

For client code written in C# and Visual Basic, there is no apparent difference between calling an extension method and the methods that are actually defined in a type.

The most common extension methods are the **LINQ standard query operators**.



Metody rozszerzające (Extension Methods)

Dodawanie nowych metod do istniejących klas bez modyfikacji ich kodu

```
static class RozszerzenieDouble{
    public static double Potega(this double x, int n){
        double p = 1;
        for (int i = 0; i < (n < 0 ? -n : n); i++)
            p *= x;
        return n < 0 ? 1 / p : p;
    }
}
```

```
class Program{
    static void Main(string[] args){
        double a = 2;
        Console.WriteLine(a.Potega(2));
        Console.WriteLine((2.0).Potega(-2));
    }
}
```



Implicitly typed local variables

C# 3.5 introduces a new **var** keyword that can be used in place of the type name when performing **local** variable declarations. The **var** keyword always generates a strongly typed variable reference. Rather than require the developer to explicitly define the variable type, the **var** keyword instead tells the compiler to **infer the type of the variable** from the expression used to initialize the variable when it is first declared. The **var** keyword can be used to reference any type in C#.

Example:

```
var name = "ABC XYZ";  
var age = 33;  
var male = true;
```



```
string name = "ABC XYZ";  
int age = 33;  
bool male = true
```

The compiler will infer the type of the "name", "age" and "male" variables based on the type of their initial assignment value.

Limitations:

- only local variables
- the variable must be initialized in the declaration
- the initialization expression must not be an anonymous function without casting
- the variable must not be initialized to null
- only one variable can be declared in the statement
- the type must be compile-time type



Typy implikowane (implicit typing)

```
var a = 4;           // int
var b = 'M';        // char
var c = 1234L;      // long
var d = "Ala ma kota"; // string
...

a = 4654434L;       // long -> int
d = 4.5;             // błąd
```

**tylko zmienne lokalne w funkcjach -
nie mogą być składowymi klas**



Anonymous Types

Anonymous types provide a convenient way to encapsulate a set of read-only properties into a single object without having to first explicitly define a type.

The type name is **generated by the compiler** and is not available at the source code level. The type of the properties is **inferred** by the compiler.

The following example shows an anonymous type being initialized with two properties called Price and Message.

```
var v = new { Price = 105, Message = "XYZ" };
```

Anonymous types are typically used in the **select clause** of a **query expression** to return a subset of the properties from each object in the **source sequence**.

Anonymous types are created by using the **new** operator with an object initializer.



Anonymous Types

```
public class Customer
{
    public string Name;
    public Address Address;
    public string Phone;
    public List<Order> Orders;
    ...
}
```

```
public class Contact
{
    public string Name;
    public string Phone;
}
```

```
Customer c = GetCustomer(...);
Contact x = new Contact { Name = c.Name, Phone = c.Phone };
```

```
Customer c = GetCustomer(...);
var x = new { Name = c.Name, Phone = c.Phone };
```

```
Customer c = GetCustomer(...);
var x = new { c.Name, c.Phone };
```



Lambda Expressions

A **lambda expression** is an **anonymous function** that can contain expressions and statements, and can be used to create delegates or expression tree types.

All lambda expressions use the **lambda operator** `=>`, which is read as "goes to".

The **left side** of the lambda operator specifies the **input parameters** (if any) and the **right side** holds **the expression** or **statement block**.

Example:

```
var maxAge = objPerson.Max(p => p.Age).ToString();
```

One of the things that make Lambda expressions particularly powerful from a framework developer's perspective is that they can be compiled as either a **code delegate** (in the form of an IL based method) or as an **expression tree object** which can be used at runtime to analyze, transform or optimize the expression.



Wyrażenia lambda

(lista_argumentów) => {instrukcje_definiujące_funkcję};

```
delegate double Delgacja1();
```

```
Delgacja1 wyr1 = () => {  
    Console.WriteLine("Funkcja bezparametrowa");  
    return 2.0;  
};  
Console.WriteLine(wyr1());
```

```
delegate int Delgacja2(int x, int y);
```

```
Delgacja2 wyr2 = (a, b) => a + b;  
Console.WriteLine("3 + 4 = {0}", wyr2(3, 4));
```

```
delegate double Delgacja3(double x);
```

```
Delgacja3 wyr3 = a => a * a;  
Console.WriteLine("Kwadrat liczby 3 wynosi {0}", wyr3(3));
```



2.2.9 Refleksja i atrybuty

System.Reflection, System.Attribute

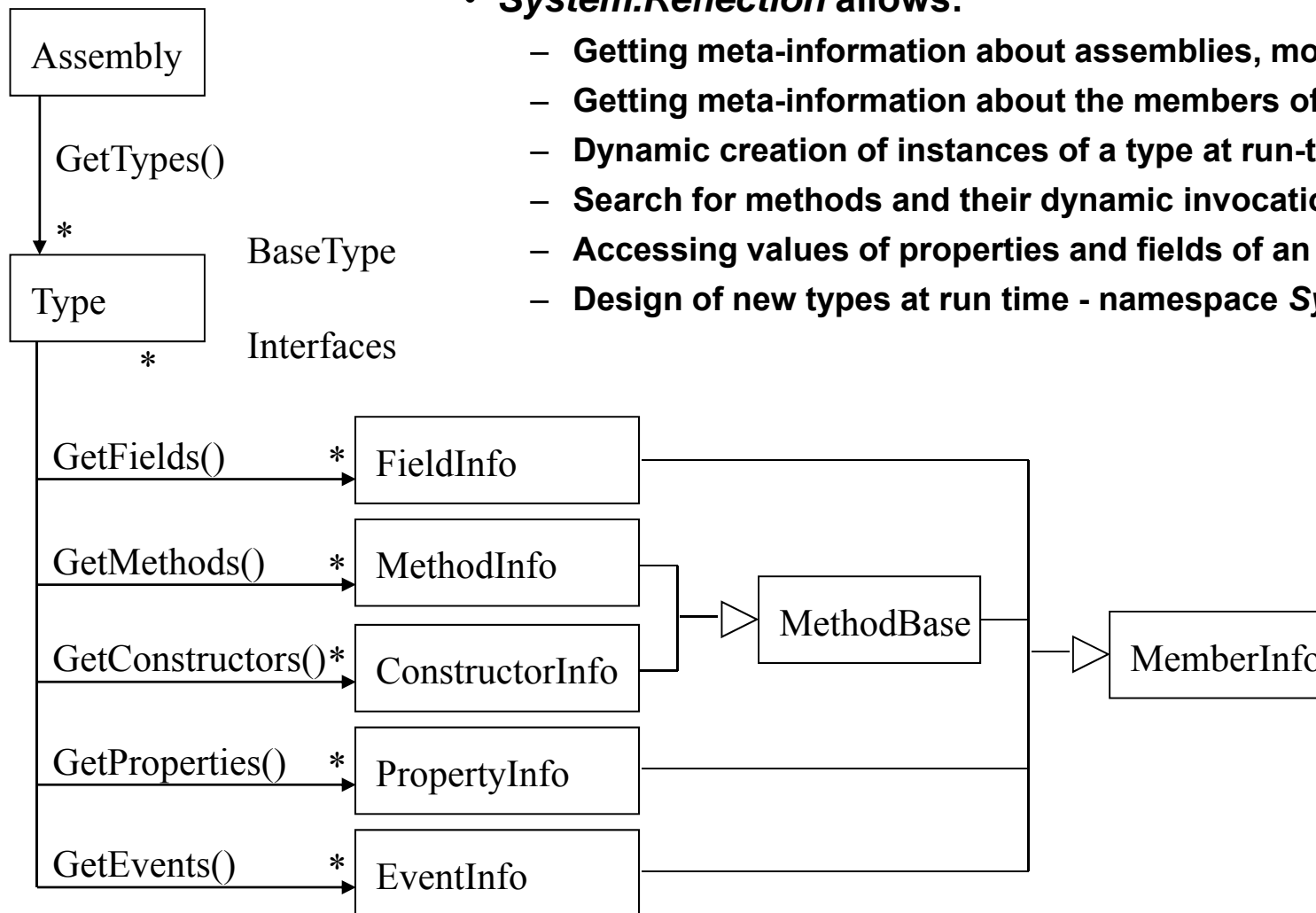
- Uzyskiwanie informacji o typie w czasie działania programu
- Przestrzeń nazw *System.Reflection*
- Klasa *System.Type*
 - Dziedziczy po *MemberInfo*
 - **Właściwości** (*IsAbstract*, *IsArray*, *IsNestedPrivate*, *IsNestedPublic*, ...)
 - **Metody** (*GetType()*, *GetConstructors()*, *GetEvents()*, *GetMethods()*, *FindMembers()*, ...) - zwracają obiekty, których typ jest określony przez jedną z klas znajdujących się w przestrzeni nazw *System.Reflection*

Class	Description
Assembly	represents an assembly, which is a reusable, versionable, and self-describing building block
AssemblyName	describes an assembly's unique identity in full
EventInfo	discovers the attributes of an event and provides access to event metadata
FieldInfo	discovers the attributes of a field and provides access to field metadata
MemberInfo	obtains information about the attributes of a member and provides access to member metadata
MethodInfo	obtains information about the attributes of a member and provides access to member metadata
ParameterInfo	discovers the attributes of a parameter and provides access to parameter metadata
...	



Reflection Class Hierarchy

- **Permits access to meta-information of types at run-time**
- ***System.Reflection* allows:**
 - **Getting meta-information about assemblies, modules and types**
 - **Getting meta-information about the members of a type**
 - **Dynamic creation of instances of a type at run-time**
 - **Search for methods and their dynamic invocation at run-time**
 - **Accessing values of properties and fields of an object**
 - **Design of new types at run time - namespace *System.Reflection.Emit***





Class Type

- Type used for meta-description of all types in the run-time system
- Provides access to the meta-information about its members

```
public abstract class Type : MemberInfo, IReflect {
```

```
    public abstract string FullName {get;};  
    public abstract Type BaseType {get;};  
    public Type[] GetInterfaces();
```

- Type name
- Direct base type
- List of implemented interfaces

```
    public bool IsAbstract {get;};  
    public bool IsClass {get;};  
    public bool IsPublic {get;};
```

- Properties of type

...

```
    public ConstructorInfo[] GetConstructors();  
    public virtual EventInfo[] GetEvents();  
    public FieldInfo[] GetFields();  
    public MethodInfo[] GetMethods();  
    public PropertyInfo[] GetProperties();
```

- Getting constructors, events, fields, methods, properties
- Optionally parameterized by **BindingFlags**

...



Klasa *System.Type*

- Metoda *Object.GetType()*

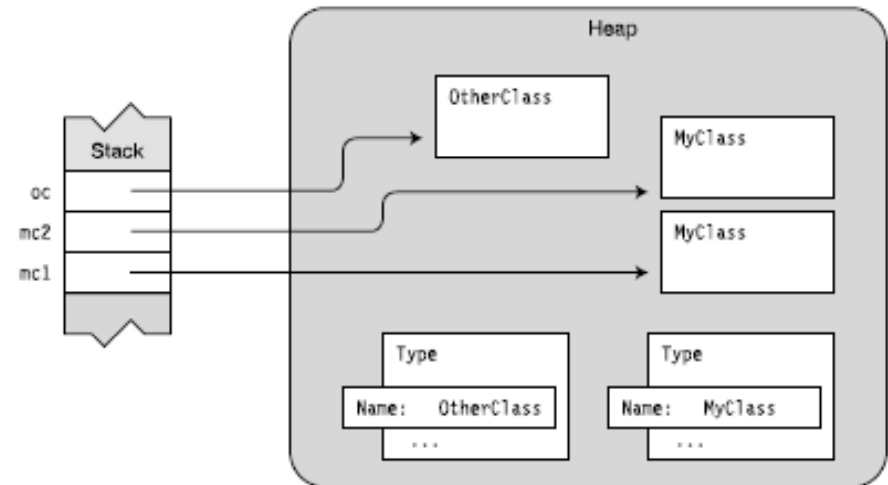
```
MojaKlasa obj = new MojaKlasa();  
Type t = obj.GetType();
```

- operator *typeof()*

```
Type t = typeof(NazwaTypu);
```

- metoda statyczna klasy *Type* - *GetType()*

```
using System;  
public class MyBaseClass: Object { }  
public class MyDerivedClass: MyBaseClass { }  
public class Test {  
    public static void Main() {  
        MyBaseClass myBase = new MyBaseClass();  
        MyDerivedClass myDerived = new MyDerivedClass();  
        Console.WriteLine("mybase: Type is {0}", myBase.GetType());  
        Console.WriteLine("myDerived: Type is {0}", myDerived.GetType());  
    }  
}
```





Refleksja - przykład

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Reflection;

namespace Page_227
{
    class SomeClass
    {
        public int Field1;
        public int Field2;

        public void Method1()
        {
        }

        public int Method2()
        {
            return 1;
        }
    }
}
```

```
class Program
{
    static void Main()
    {
        Type t = typeof( SomeClass );
        FieldInfo[] fi = t.GetFields();
        MethodInfo[] mi = t.GetMethods();

        foreach ( FieldInfo f in fi )
            Console.WriteLine( "Field : {0}", f.Name );

        foreach ( MethodInfo m in mi )
            Console.WriteLine( "Method: {0}", m.Name );
    }
}
```

```
file:///D:/...
Field : Field1
Field : Field2
Method: Method1
Method: Method2
Method: ToString
Method: Equals
Method: GetHashCode
Method: GetType
```



Class Assembly

- **Class *Assembly*** loads assemblies and their meta-data
- **Provides access to its meta-data**

- **Klasa *System.Reflection.Assembly***

- *Load()*
- *LoadFrom()* - pozwala wprowadzić ścieżkę bezwzględną

```
public class Assembly {
```

```
    public static Assembly Load(string name);
```

```
    public virtual string FullName {get;}
```

```
    public virtual string Location {get;}
```

```
    public virtual MethodInfo EntryPoint {get;}
```

```
    public Module[] GetModules();
```

```
    public virtual Type[] GetTypes();
```

```
    public virtual Type GetType(string typeName);
```

```
    public object CreateInstance(string typeName);
```

```
    ...
```

```
}
```

```
Assembly.LoadFrom("c:\\Katalog\\Przykladowy.Zestaw.dll");
```

```
Assembly a = Assembly.Load(@"NazwaZestawu, Version=1.0.982.23972,  
    PublicKeyToken=null, Culture="");
```

- Loading an assembly
- Name, storage location, entry point of the assembly
- Getting modules and all in the assembly defined types
- Getting type with name typeName
- Creation of an object of type typeName



Refleksja – przykład 2

```
namespace Hello {  
    using System;  
    public class HelloWorld {  
        public static void Main(string[] args) {  
            Console.WriteLine("HelloWorld");  
        }  
        public override string ToString() {  
            return "Example HelloWorld";  
        }  
    }  
}  
  
// Loading the assembly "HelloWorld.exe":  
Assembly a = Assembly.Load("HelloWorld");  
  
// Print all existing types in a given assembly  
Type[] types = a.GetTypes();  
foreach (Type t in types)  
    Console.WriteLine(t.FullName);  
  
// Print all existing methods of a given type  
Type hw = a.GetType("Hello.HelloWorld");  
MethodInfo[] methods = hw.GetMethods();  
foreach (MethodInfo m in methods)  
    Console.WriteLine(m.Name);
```

Hello.HelloWorld

GetType
ToString
Equals
GetHashCode



Późne wiązanie – ładowanie klasy on-line

- Klasa *System.Activator*

Plug-iny

```
Assembly a = null;
a = Assembly.Load("NazwaZestawu");
Type typ = a.GetType("Przestrzen.Klasa");
object obj = Activator.CreateInstance(typ);
MethodInfo mi = typ.GetMethod("NazwaMetody");
mi.Invoke(obj, null); //null - bez argumentów
object[] parametry = new object[2];
parametry[0] = "Napis";
parametry[1] = 4;
mi = typ.GetMethod("MetodaZParametrami");
mi.Invoke(obj, parametry);
```



Atrybuty - *System.Attribute*

- **Atrybuty** - mechanizm, pozwalający na dodanie dodatkowej informacji opisującej poszczególne elementy kodu – metadane.
- **Dostęp do metadanych jest możliwy dzięki mechanizmowi refleksji,**
 - Predefiniowane atrybuty
 - Tworzenie własnych atrybutów

Przykłady:

- **[Conditional]** – powoduje, że wywołanie metody jest zależne, od zdefiniowania jakiegoś identyfikatora przy pomocy dyrektywy `#define`. Atrybut ten jest często wykorzystywany do metod diagnostycznych, sprawdzających pewne asercje, wypisujących że zostało osiągnięte pewne miejsce w naszym programie.
- **[Serializable]** – stosowany do klas, umożliwia zapisywanie stanu obiektu danej klasy do strumienia.
- **[WebMethod]** – dodanie tego atrybutu do metody klasy reprezentującej usługę sieciową (XML Web service), powoduje że dana metoda jest dostępna dla klientów danej usługi.
- **[DllImport]** – pozwala wywołać dowolną eksportowaną funkcję z biblioteki DLL kodu niezarządzanego. Ten atrybut dodajemy do sygnatury metody, która niejako stanowi punkt wejścia do żądanej metody



Atrybuty

- Metadane - adnotacje do kodu
- Klasy dziedziczące po *System.Attribute*
- Mogą opisywać
 - podzespoły, metody, typy, klasy interfejsy, pola, zdarzenia, właściwości
- Nazwa jest domyślnie rozszerzana przez przyrostek *Attribute*

```
[Serializable] //atrybut dotyczy klasy NaszaKlasa
public class NaszaKlasa
{
    [Obsolete("Użyj innej metody")] //atrybut dotyczy metody Metoda1
    public void Metoda1() { }
}

[assembly: AssemblyCompany("Nazwa firmy")]
```



Defining Your Own Attributes

Declaration

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface, Inherited=true)]
class CommentAttribute : Attribute {
    string text, author;
    public string Text { get {return text;} }
    public string Author { get {return author;} set {author = value;} }
    public Comment (string text) { this.text = text; author = "HM"; }
}
```

Usage

```
[Comment("This is a demo class for Attributes", Author="XX")]
class C { ... }
```

Querying the attribute at runtime

```
class Attributes {

    static void Main() {
        Type t = typeof(C);
        object[] a = t.GetCustomAttributes(typeof(Comment), true);
        foreach (Comment ca = a) {
            Console.WriteLine(ca.Text + ", " + ca.Author);
        }
    }
}
```



Atrybuty - przykład

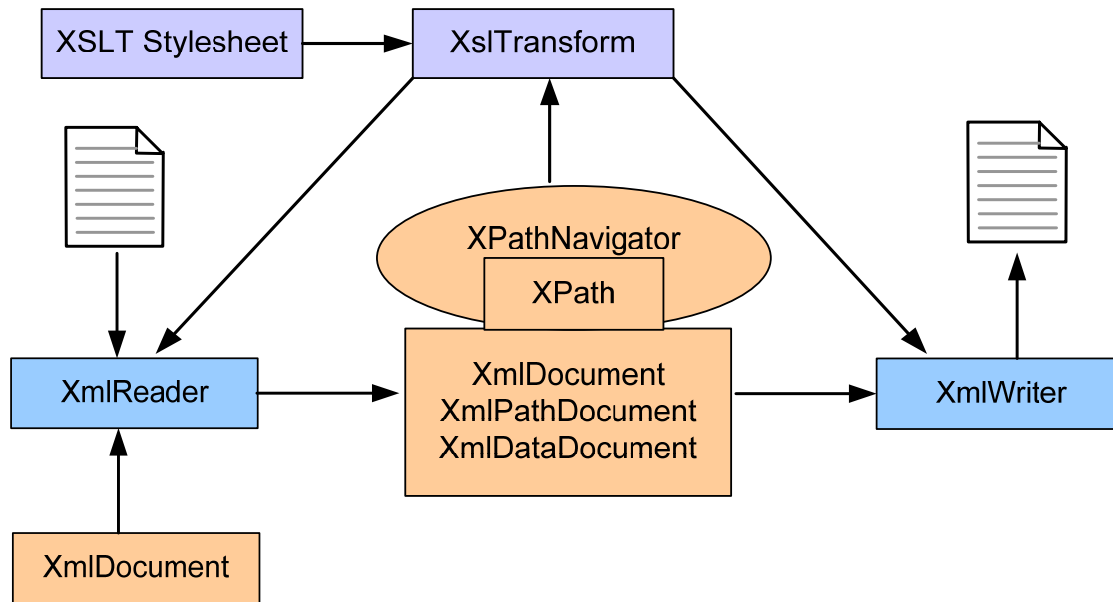
```
#define DoTrace
using System;
using System.Diagnostics;

namespace AttributesConditional
{
    class Program
    {
        [Conditional( "DoTrace" )]
        static void TraceMessage( string str )
        {
            Console.WriteLine( str );
        }

        static void Main()
        {
            TraceMessage( "Start of Main" );
            Console.WriteLine( "Doing work in Main." );
            TraceMessage( "End of Main" );
        }
    }
}
```



2.2.10 Przetwarzanie dokumentów XML



- **XmlReader:** Reading XML data
- **XmlDocument, XmlNode:** Object model for XML data (DOM)
- **XmlWriter:** Writing XML data
- **XPathNavigator:** XPath selections
- **XsltTransform:** Transformation of XML documents



Odczyt i zapis XML w .NET

Abstrakcyjne klasy **XmlReader** i **XmlWriter** po których dziedziczą:

- **XmlTextReader** – zapewnia odczyt danych XML z pliku, sprawdzając czy dokument jest dobrze uformowany, lecz bez możliwości wykorzystania W3C Schema. Odczytując dokument jesteśmy w stanie poruszać się tylko do przodu, nie ma możliwości cofnięcia się do poprzednich linii
- **XmlNodeReader** – zapewnia dokładnie tą samą funkcjonalność co poprzednik, lecz odczytuje dane tylko z określonego węzła drzewa.
- **XmlValidationReader** – to samo co **XmlTextReader** lecz z możliwością sprawdzenia poprawności danych względem reguł zawartych w W3C Schema.
- **XmlTextWriter** – zapewnia zapisywanie danych do pliku XML, także bez możliwości cofnięcia się do poprzednich linii

```
XmlTextReader xtw = new XmlTextReader("Sample1.xml");
while(xtw.Read())
{
    Console.WriteLine(xtw.NodeType.ToString() + ":" + xtw.Name + ":" + xtw.Value);
    if(xtw.HasAttributes)
    {
        for(int i=0; i<xtw.AttributeCount; i++)
        {
            xtw.MoveToAttribute(i);
            Console.WriteLine(xtw.NodeType.ToString() + ":" + xtw.Name + ":" + xtw.Value);
        }
        xtw.MoveToElement();
    }
}
```



Odczyt i zapis XML w .NET (2)

```
XmlTextWriter xtw = new XmlTextWriter("Sample2.xml", null);  
xtw.Formatting=Formatting.Indented;  
xtw.WriteStartDocument();  
xtw.WriteComment("to plik utworzony poprzez obiekt XmlTextWriter");  
xtw.WriteStartElement("Książki");  
xtw.WriteStartElement("Książka");  
xtw.WriteAttributeString("wydawnictwoId", "3");  
xtw.WriteStartElement("ISBN");  
xtw.WriteString("777-7777-777");  
xtw.WriteEndElement();  
xtw.WriteEndElement();  
xtw.WriteEndElement();  
xtw.WriteEndDocument();  
xtw.Close();
```



Struktura dokumentu DOM

- **DocumentElement**: wierzchołek drzewa dokumentu;
- **Node**: każdy z węzłów (w tym elementy);
- **Element a węzeł**: element to jeden z występujących w dokumencie DOM rodzajów węzłów (m.in. węzły tekstowe, atrybutowe).
 - Węzły **elementowe**: zwykle zawierają węzły potomne: elementowe, tekstowe lub obu rodzajów.
 - Węzły **atrybutowe**: informacja podległa konkretnemu węzłowi elementowemu. Nie są określane jako potomkowie.
 - Węzeł **dokumentu**: rodzic dla wszystkich pozostałych węzłów.
 - **CData**: odpowiada sekcji znakowej, zawierającej zawartość nie przeznaczoną do przetwarzania;
 - **Comment**: (komentarz); **ProcessingInstruction** (instr. przetwarzania);
 - **DocumentFragment**: stosowany jako węzeł roboczy przy budowaniu lub rekonstruowaniu dokumentu XML.
 - **Ponadto**: Entity, EntityReference, Notation.



DOM - Przykład

Przykład wykorzystujący klasę *XmlDocument* środowiska .NET Framework do pobrania nazwy artysty i tytułu z pierwszego elementu compact-disc, znajdującego się w elemencie items

```
using System;
using System.Xml;
public class Test{
    public static void Main(string[] args){
        XmlDocument doc = new XmlDocument();
        doc.Load("test.xml");
        XmlElement firstCD = (XmlElement)doc.DocumentElement.FirstChild;
        XmlElement artist = (XmlElement)
            firstCD.GetElementsByTagName("artist")[0];
        XmlElement title =
            (XmlElement) firstCD.GetElementsByTagName("title")[0]
        Console.WriteLine("Artysta={0}, Tytuł={1}", artist.InnerText,
            title.InnerText);
    }
}
```



XPath w.NET - przykład

Klasy XPathDocument i XPathNavigator

Przykład, w którym zastosowano klasę **XPathDocument** do pobrania nazwy artysty i tytułu z pierwszego elementu compact-disc, znajdującego się w elemencie items

```
using System;
using System.Xml.XPath;
public class Test{
    public static void Main(string[] args){
        XPathDocument doc = new XPathDocument("test.xml");
        XPathNavigator nav = doc.CreateNavigator();
        XPathNodeIterator iterator = nav.Select("/items/compact-disc[1]/artist
| /items/compact-disc[1]/title");
        iterator.MoveNext();
        Console.WriteLine("Artysta={0}", iterator.Current);
        iterator.MoveNext();
        Console.WriteLine("Tytuł={0}", iterator.Current);
    }
}
```



Przekształcanie dokumentów XML: Język XSL (*eXtensible Stylesheet Language*)

XML – opisuje strukturę i semantykę, nie opisuje sposobów prezentacji informacji

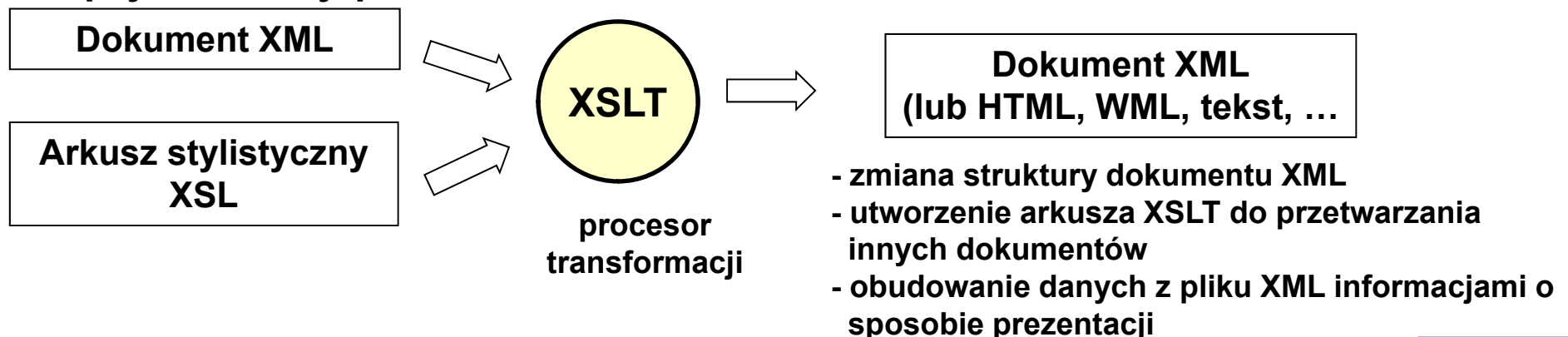
Oddzielenie warstwy informacji od prezentacji uzyskuje się poprzez dołączanie arkusza stylów.

- **CSS - Cascading Style Sheets** (level1 i Level2) - związany głównie z HTML
- **XSL - eXtensible Stylesheet Language** - opracowany dla dokumentów XML.

Obejmuje:

- **XSLT (*XSL Transformations*)** - język przekształceń o charakterze deklaratywnym, oparty na regułach przekształceń (*rule-based*) drzew XML
- **XSL FO (*XSL Formating Objects*)** – język opisu formatu

Koncepcja realizacji przekształceń





XSLT w .NET - przykład

Postać dokumentu wyjściowego:

```
<?xml version="1.0" encoding="utf-8" ?>
<Towar>
    <Książka>
        <Cena></Cena>
        <Autor></Autor>
    </Książka>
</Towar>
```

Szablon XSLT:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.0" xmlns="http://www.w3.org/1999/XSL/Transform">
<xsl:template match ="/">
    <Towary>
        <xsl:apply-templates/>
    </Towary>
</xsl:template>
<xsl:template match ="Książka">
    <Książka>
        <Cena>
            <xsl:value-of select ="Cena"/>
        </Cena>
        <Autor>
            <xsl:value-of select ="Autor/Imie"/><xsl:text> </xsl:text>
            <xsl:value-of select ="Autor/Nazwisko"/>
        </Autor>
    </Książka>
</xsl:template>
</xsl:stylesheet>
```

Sample3.xslt



XSLT w .NET – przykład (2)

Zastosowanie szablonu:

```
XPathDocument xpd = new XPathDocument("Sample1.xml");  
FileStream fs = new  
    FileStream("wysyłka.xml", FileMode.OpenOrCreate, FileAccess.Write, FileShare.  
        e.ReadWrite);  
XslTransform xsl = new XslTransform();  
xsl.Load("Sample3.xslt");  
xsl.Transform(xpd, null, fs);  
fs.Close();
```

Dokument wyjściowy XML:

```
<?xml version="1.0" encoding="utf-8"?>  
<Towary>  
    <Książka>  
        <Cena>981</Cena>  
        <Autor>Tom Archer</Autor>  
    </Książka>  
    ...  
</Towary>
```

wysyłka.xml



2.2.11 Serializacja obiektów

Serializacja jest procesem zamiany obiektów lub kolekcji obiektów na strumień bajtów.

Wykorzystywana do zapisywania stanu systemu w celu późniejszego odtworzenia lub przy przesyłaniu obiektów przez sieć.

Deserializacja jest procesem odwrotnym.

- **Serializacja binarna** – klasa **BinaryFormatter** do serializowania struktur danych do postaci strumienia binarnego. Tworzy dokładną reprezentację obiektu, zawierającą również informacja o typach.
- **Serializacja protokołu SOAP** – klasa **SoapFormatter** do serializowania typu danych do postaci strumienia XML zgodnego ze standardami protokołu SOAP
- **Serializacja do formatu XML** – klasa **XMLSerializer** do serializowania danych do postaci dokumentu XML



Serializacja binarna

```
using System;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;
using System.IO;
// Store Chair objects in a Hashtable
Hashtable ht = new Hashtable();
// Chair and Upholstery must have [Serializable] attribute
Chair ch = new Chair(100.00D, "Broyhill", "10-09");
ch.myUpholstery = new Upholstery("Cotton");
ht.Add("10-09", ch);
// Add second item to table
ch = new Chair(200.00D, "Lane", "11-19");
ch.myUpholstery = new Upholstery("Silk");
ht.Add("11-19", ch);
```

Deklaracje klas *Chair* i *Upholstery* muszą zawierać atrybut [Serializable]

Proces serializacji tworzy graf obiektów – głęboka kopia obiektów.

Możliwość selektywnego wyłączania składowych klas [NonSerialized]

```
public Upholstery myUpholstery;
```

Nadaje się do przechowywania stanu systemu, komunikacji przez *Remoting*

```
// (1) Serialize // Create a new file; if file exists it is overwritten
FileStream fs= new FileStream("c:\\chairs.dat", FileMode.Create);
BinaryFormatter bf= new BinaryFormatter();
bf.Serialize(fs,ht); fs.Close();
// (2) Deserialize binary file into a Hashtable of objects
ht.Clear();
// Clear hash table.
fs = new FileStream("c:\\chairs.dat", FileMode.Open);
ht = (Hashtable) bf.Deserialize(fs); // Confirm objects properly recreated
ch = (Chair)ht["11-19"];
Console.WriteLine(ch.myUpholstery.Fabric); // "Silk"
fs.Close();
```



Serializacja binarna (2)

- **Serialization and Deserialization Events**

Event	Attribute	Description
OnSerializing	[Serializing]	Occurs before objects are serialized. Event handler is called for each object to be serialized.
OnSerialized	[Serialized]	Occurs after objects are serialized. Event handler is called once for each object serialized.
OnDeserializing	[Deserializing]	Occurs before objects are deserialized. Event handler is called once for each object to be deserialized.
OnDeserialized	[Deserialized]	Occurs after objects have been deserialized. Event handler is called for each deserialized object.

```
public class Chair
{
    // other code here
    [OnDeserialized]
    void OnDeserialized(StreamingContext context)
    {
        // Edit vendor name after object is created
        if (MyVen == "Lane") MyVen = "Lane Bros.";
    }
}
```



XML Serialization in the .NET Framework

System.Xml.Serialization

XmlRootAttribute

XmlElementAttribute

XmlAttributeAttribute

XmlArrayAttribute

XmlArrayItemAttribute

Serializacja XML-owa

- Serializacja XML-owa serializuje publiczne zmienne i właściwości.
- Wykorzystanie atrybutów przypisanych do zmiennych lub właściwości jako znaczników w dokumencie XML.

Deserializacja XML-owa

- zbudowanie hierarchii obiektów na podstawie dokumentu XML.

xsd.exe – do generowania klas na podstawie schematu



Przykład: Using XmlSerializer to Create an XML File

```
using System.Xml;
using System.Xml.Serialization;
// other code here ...
public class movies
{
    public movies() // Parameterless constructor is required
    { }
    public movies(int ID, string title, string dir, string pic,
                  int yr, int movierank)
    {
        movieID = ID;
        movie_Director = dir;
        bestPicture = pic;
        rank = movierank;
        movie_Title = title;
        movie_Year = yr;
    }

    // Public properties that are serialized
    public int movieID
    {
        get { return mID; }
        set { mID = value; }
    }
    public string movie_Title
    {
        get { return mTitle; }
        set { mTitle = value; }
    }
}
```

```
public int movie_Year
{
    get { return mYear; }
    set { mYear = value; }
}

public string movie_Director
{
    get { return mDirector; }
    set { mDirector = value; }
}
public string bestPicture
{
    get { return mbestPicture; }
    set { mbestPicture = value; }
}
[XmlElement("AFIRank")]
public int rank
{
    get { return mAFIRank; }
    set { mAFIRank = value; }
}
private int mID;
private string mTitle;
private int mYear;
private string mDirector;
private string mbestPicture;
private int mAFIRank;
}
```



Przykład: Using XmlSerializer to Create an XML File

```
// (1) Create array of objects to be serialized
movies[] films = {new movies(5,"Citizen Kane","Orson Welles",
                            "Y", 1941,1 ),
                  new movies(6,"Casablanca","Michael Curtiz",
                            "Y", 1942,2)};

// (2) Create serializer
// This attribute is used to assign name to XML root element
XmlAttribute xRoot = new XmlRootAttribute();
xRoot.ElementName = "films";
xRoot.Namespace = "http://www.corecsharp.net";
xRoot.IsNullable = true;

// Specify that an array of movies types is to be serialized
XmlSerializer xSerial = new XmlSerializer(typeof(movies[]),
                                           xRoot);

string filename=@"c:\oscarwinners.xml";

// (3) Stream to write XML into
TextWriter writer = new StreamWriter(filename);
xSerial.Serialize(writer,films);
```



Przykład: wynik serializacji

```
<?xml version="1.0" standalone="yes"?>
<films>
  <movies>
    <movie_ID>5</movie_ID>
    <movie_Title>Citizen Kane </movie_Title>
    <movie_Year>1941</movie_Year>
    <movie_DirectorID>Orson Welles</movie_DirectorID>
    <bestPicture>Y</bestPicture>
    <AFIRank>1</AFIRank>
  </movies>
  <movies>
    <movie_ID>6</movie_ID>
    <movie_Title>Casablanca </movie_Title>
    <movie_Year>1942</movie_Year>
    <movie_Director>Michael Curtiz</movie_Director>
    <bestPicture>Y</bestPicture>
    <AFIRank>1</AFIRank>
  </movies>
</films>
```



Atrybuty kontrolujące XML serializację

- Elementy tworzone na podstawie serializowanych obiektów klasy domyślnie otrzymują nazwy zgodne z nazwami reprezentowanych właściwości.
- Istnieją Atrybuty serializacji, za pomocą których możemy przykrywać domyślne nazwy elementów.
 - Poprzedni przykład - nazwa właściwości *rank* została zastąpiona wyrażeniem *AFIRank*

Serialization attributes

XmlElement	Zmienna lub właściwość zostanie podczas serializacji zapisana jako węzeł
XmlAttribute	Is attached to a property or field and causes it to be rendered as an attribute within an element. Example: <code>XmlAttribute("movie_ID")</code> Result: <code><movies movie_ID=„5”></code>
XmlIgnore	Causes the field or property to be excluded from the XML.
XmlText	Causes the value of the field or property to be rendered as text. No elements are created for the member name. Example: <code>[XmlText]</code> <code>public string movie_Title{</code> Result: <code><movies movie_ID=„5”>Citizen Kane</code>