Advanced Programming Methods
Lecture 3

Lectures 3+4 Bibliografy

- C# Microsoft Programming Guide:
  http://msdn.microsoft.com/en-us/library/67ef8sbd.aspx

  C# free tutorials and books at:
http://www.cs.ubbcluj.ro/~craciunf/MetodeAvansateDeProgramare/ResourceCSharp/

Contents

Fundamentals of .NET and C#

The .NET Framework consists of:

- a runtime called the *Common Language Runtime* (CLR),

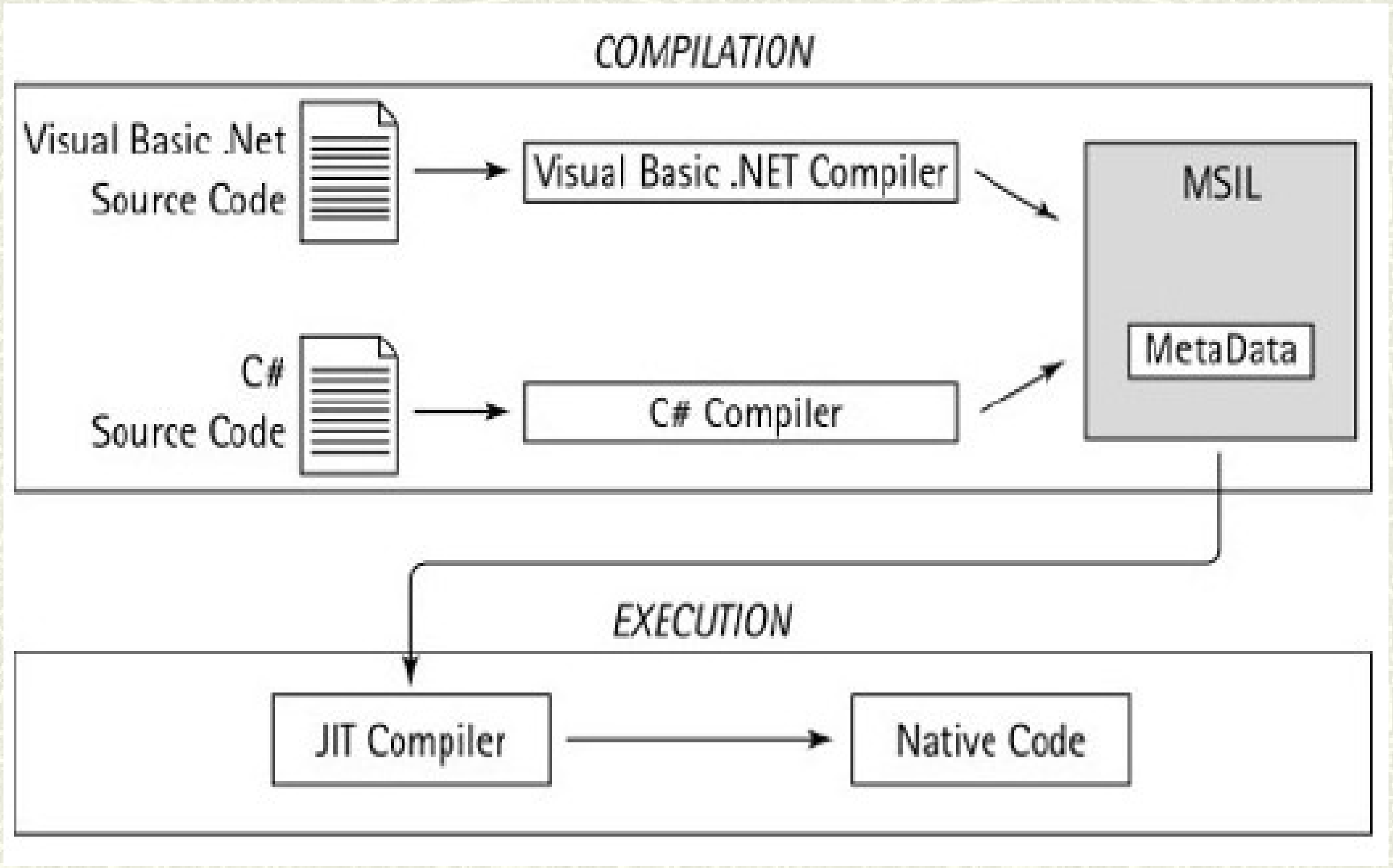
- a set of libraries:
  - core libraries (*mscorlib.dll, System.dll, System.Xml.dll, System.Core.dll*),
  - applied libraries (Windows forms, ADO.NET, etc).

The CLR is the runtime for executing *managed code*.

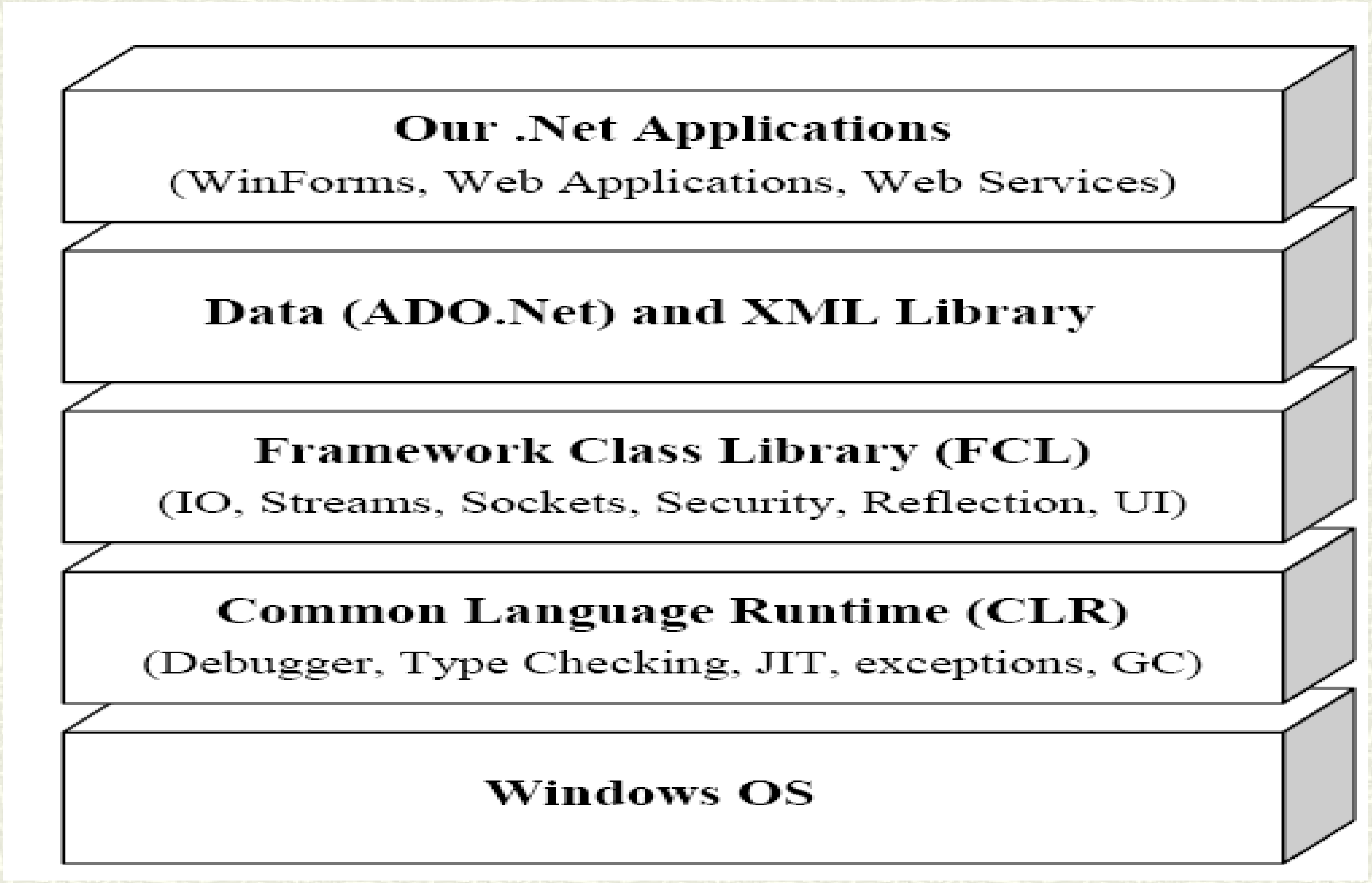
C# is one of several *managed languages* that get compiled into managed code.

Managed code is packaged into an *assembly* (an executable file (an *.exe*) or a library (a *.dll*)), along with type information, or *metadata*.

Fundamentals of .NET and C#



Fundamentals of .NET and C#


Our .Net Applications
(WinForms, Web Applications, Web Services)
Data (ADO.Net) and XML Library
Framework Class Library (FCL)
(IO, Streams, Sockets, Security, Reflection, UI)
Common Language Runtime (CLR)
(Debugger, Type Checking, JIT, exceptions, GC)
Windows OS

C# basics

//Java

```
class Test{
 public static void main(String args[]){
   System.out.println("Hello");
  }
}
```

//C#

```
using System;
class Test{
  public static void Main(String args[]){
   Console.WriteLine("Hello");
  }
}
```

4 different possible entry points:

```
static void Main(){}     static void Main(String[] args){}
static int Main(){}      static int Main(String[] args){}
```

C# compiler

- The C# compiler compiles source code, specified as a set of files with the .cs extension, into an assembly.
- A normal console or Windows *application* has a Main method and is an *.exe*.
- A library is a *.dll* and is equivalent to an *.exe* without an entry point.
- The name of the C# compiler is *csc.exe*

```
csc Test.cs
```

It produces an application named *Test.exe*

```
csc target:library /out:datastruct.dll List.cs Stack.cs
```

Data Types

C# Type	.NET Type	Size in bytes	Description
byte	Byte	1	0 .. 255
sbyte	SByte	1	-128 … 127
short	Int16	2	-32 768 … 32767
ushort	UInt16	2	0 … 65535
int (default)	Int32	4	-2147483648 … 2147483647
uint	UInt32	4	0 … 4294967295
long	Int64	8	-9 223 372 036 854 775 808 … 9 223 372 036 854 775 807
ulong	UInt64	8	0 … 18 446 744 073 709 551 615
float	Single	4	$\pm 1.5 * 10^{-45} \ldots \pm 3.4 * 10^{38}$
double (default)	Double	8	$\pm 5.0 * 10^{-324} \ldots \pm 1.7 * 10^{308}$
bool	Boolean	1	true, false

Operators

Precedenc e	Operators
1.	(), ., ->, [ ], ++, --, new, typeof
2.	sizeof, +, -, ! (unary), ~, (cast)
3.	*, /, %
4.	+, -
5.	<<, >>
6.	<, >, <=, >=, is, as
7.	==, !=
8.	&
9.	^
10.	|

C# Statements

if … else statement

```
if (boolean expresion)
  statement or block of statements
[else
  statement or block of statements]
```

switch … case statement

```
switch(integral or string expression)
{
  case constant - expression:
    breaking or jump statement
  //some other case blocks
  //...
  default:
    statements
    breaking or jump statement
}
```

Flow Control Statements

For loop

```
   for(initialization; condition; iteration-clause)
     statement or block of statements
```

Do … while loop

```
   do
     statement or block of statements
   while (boolean expression);
```

While loop

```
   while (boolean expression)
     statement or block of statements
```

Foreach loop

```
foreach (<type of elements in collection> <identifier> in <array or collection>)
   statement or block of statements
```

Jump Statements

- **`break`** ends the execution of the body of a **`while`** loop, **`for`** loop, or **`switch`** statement.
- **`continue`** skips the remaining statements in the loop and starts from the next iteration.
- **`goto`** transfers execution to another label within the statement block.

Constants

A *constant* is a field whose value can never change. A constant is evaluated statically at compile time and its value is literally substituted by the compiler whenever used.

It can be one of the following types: `sbyte, byte, short, ushort, int, uint, long, ulong, float, double, decimal, bool, char, string`, or an `enum` type.

A constant is declared with the `const` keyword and must be initialized with a value.

```
   const string Message = "Hello World";
```

Java

```
 final String message;
 …
 message=“Hello World”;
```

Arrays

1-dimensional array declaration

```
 <data type> [] <identifier>=new <data type>[<size>]
int[] arr=new int[20];
```

N-dimensional arrays

Rectangular arrays are declared using commas to separate each dimension

```
int [,] matrix = new int [2, 3];
```

The `GetLength` method of an array returns the length for a given dimension (starting at 0)

```
for (int i = 0; i < matrix.GetLength(0); i++)
    for (int j = 0; j < matrix.GetLength(1); j++)
        matrix[i, j] = i * 3 + j;
```

Initialization:

```
int[,] matrix = new int[,] { {0,1,2},
                 {3,4,5} };
```

Arrays

- *Jagged arrays* are declared using successive square braces to represent each dimension.

```
int [][] matrix = new int [2][];
```

The inner dimensions are not specified in the declaration. Each inner array can have an arbitrary length. Each inner array is implicitly initialized to null, so it must be created manually:

```
for (int i = 0; i < matrix.Length; i++) {
  matrix[i] = new int [3];   // create inner array
  for (int j = 0; j < matrix[i].Length; j++)
     matrix[i][j] = i * 3 + j;
}
```

Initialization:

```
int[][] matrix = new int[][] {
           new int[] {0,1,2},
           new int[] {3,4,5}
           };
```

Enum

An `enum` is a special value type that specifies a group of named numeric constants.

```
public enum BorderSide { Left, Right, Top, Bottom }
BorderSide topSide = BorderSide.Top;
bool isTop = (topSide == BorderSide.Top);   // true
```

Each enum member has an underlying integral value. By default:

Underlying values are of type int.

The constants 0, 1, 2... are automatically assigned, in the declaration order of the enum members.

It is possible to specify an alternative integral type:

```
public enum BorderSide : byte { Left, Right, Top, Bottom }
```

It is also possible to specify an explicit underlying value for each(or some) enum member:

```
public enum BorderSide : byte { Left=1, Right=2, Top=10, Bottom }
```

Classes

A class in C# is declared using the `class` keyword:

```
class Person{
 //fields
 //methods
 //properties
}
```

Object creation:

```
Person person=new Person();
```

Field declaration:

```
[static] [access modifier][new][readonly] Type field_name [=init_val];
```

The `readonly` modifier prevents a field from being modified after construction. A read-only field can be assigned only in its declaration or within the constructor.

Constructor /Destructor

Constructor

```
[access modifier] ClassName ([list of parameters]){…}
```

constructor can be overloaded

if no constructor is defined, the compiler automatically generates a default constructor

Destructor

```
~ClassName(){…}
```

The destructor is called automatically when the object is about to be destroyed (when the garbage collector is about to destroy the object).

Destructor/Finalize

- `Finalize ()` method

Each class inherits from the Object class that contains a `Finalize()` method. This method is guaranteed to be called when the object is garbage collected. The `Finalize()` method can be overriden.

```
protected override void Finalize(){…}
```

C# compiler internally converts the destructor to the Finalize() method.

Destructors are not used very much in common C# programming practice.

Methods

```
[access_m] [static][inheritance_m] type method_name(list_params )
{
   //body of the method
}
```

List of parameters: `[modifier] type1 param1, [modifier] type2 param2,…`

Modifier	Passed by
None	Value
`ref`	Reference
`out`	Reference

Methods

```
class Test{
   static void swap(int a, int b){
    int c=a;
    a=b;
    b=c;
   }
   static void Main(){
    int x=23, y=45;
    Console.WriteLine(“x={0} y={1}”,x,y);
    swap(x,y);
        Console.WriteLine(“After x={0} y={1}”,x,y);
   }
}
```

```
x=23 y=45
After x=23 y=45
```

Methods

```
class Test{
   static void swapRef(ref int a,ref int b){
    int c=a;
    a=b;
    b=c;
   }
   static void Main(){
    int x=23, y=45;
    Console.WriteLine(“x={0} y={1}”,x,y);
    swapRef(ref x,ref y);
        Console.WriteLine(“After x={0} y={1}”,x,y);
   }
}
```

```
x=23 y=45
After x=45 y=23
```

Methods

```
class Test{
   static void testOut( int a,out int b){
    b=a*a;
   }
   static void Main(){
    int x=2, y;
    Console.WriteLine("x={0} y=",x);
    testOut(x,out y);
        Console.WriteLine("After x={0} y={1}",x, y);
   }
}
```

```
x=2 y=
After x=2 y=4
```

Access modifiers

private	Access only within the same class
protected internal	Access only from the same project or from subclasses
internal	Access only from the same project
protected	Access from subclasses
public	No restriction

Default modifier:

•fields/methods: private

•classes, interfaces: internal

Properties

Properties look like fields from the outside but act like methods on the inside.

A property is declared like a field, but with a `{ get {} set {} }` block added.

`get` and `set` denote property *accessors*. The `get` accessor runs when the property is read. It must return a value of the property's type. The `set` accessor is run when the property is assigned. It has an implicit parameter named `value` of the property's type that is typically assigned to a field.

//Java

```
class Person{
   private String name;
   //…
   public String getName() {return name; }
   public void setName(String name) {this.name=name;}
}
```

Properties

//C#

```
class Person{
    private String name;
    //…
    public String Name() {
        get{ return name; }
        set{ name=value;}
    }
}
```

A property is *read-only* if it specifies only a get accessor, and it is *write-only* if it specifies only a set accessor.

A property usually has a dedicated backing field to store the underlying data.

property can also be computed from other data.

Properties

```
public class Stock {
   string symbol;
   double price;
   long sharesOwned;
   public Stock (string symbol, double price, long shares) {
       this.symbol = symbol;
       this.price = price;
       this.sharesOwned = shares; }
   public double Price {
       get { return price; }
       set { price = value; } }
   public string Symbol {
        get { return symbol; } }
   public long SharesOwned {
        get { return sharesOwned; } }
   public double Worth { get { return price*SharesOwned; } }
}
```

Indexers

Indexers provide a natural syntax for accessing elements in a class or struct that encapsulate a list or dictionary of values. Indexers are similar to properties, but are accessed via an index `argument` rather than a property name.

```
public class Portfolio {
    Stock[] stocks;
    public Portfolio (int numberOfStocks) {
        stocks = new Stock [numberOfStocks];
    }
    public int NumberOfStocks {
        get { return stocks.Length; }
     }
    public Stock this [int index]         // indexer
    {
        get { return stocks [index]; }
        set { stocks [index] = value; }
    }
 }
```

Indexers

A type may declare multiple indexers (overloaded indexers).

The Portfolio class can be extended to also allow a stock to be returned by a symbol:

```
public class Portfolio {
    ...
    public Stock this[string symbol] {
         get { foreach (Stock s in stocks)
              if (s.Symbol == symbol) return s;
               return null;
         }
    }
```

//An indexer can take any number of parameters.

```
    public Stock this [string symbol, string exchange] {
         get { ... }
         set { ... }
     }

 }
```

Inheritance

In C# the : is used for inheritance.

```
class Person{ … }
class Student : Person { … }
```

Abstract classes and methods: `abstract` keyword

```
abstract class Element{
      public abstract int CompareTo(Element e);
}
```

Interfaces: `interface` keyword

```
public interface IComparable{
      int CompareTo(Object o);
}
```

Inheritance

C# supports single class inheritance, and multiple interface implementations.

```
class A { … }
interface B{…}
interface C : B {…}          //interface inheritance
interface CC {…}
class SA : A, B, CC  { … }  //class inheritance, interface
                     //implementation
```

Virtual methods

A function marked as `virtual` can be overridden by subclasses wanting to provide a specialized implementation. Methods, properties, indexers, and events can all be declared virtual:

```
public class Person{
        private String address;
        public virtual String getAddress(){…};
}
```

A subclass overrides a virtual method by applying the `override` modifier

```
public class Student:Person{
        public override String getAddress(){…}
}
```

It is possible to hide a member (field/method/property) deliberately, using the `new` modifier to the member in the subclass.

```
public class Student:Person{
        private new String address;
        public new String getAddress(){…}
}
```

`base` keyword

It is used to explicitly call the constructor of the base class.

```
class SubClass : BaseClass{
   SubClass(int id):base (){ //explicitly call the default
   constructor
   //…
   }
   SubClass(String n, int id): base(n){
   //…
   }
}
```

`sealed` keyword

A sealed method cannot be overridden by a subclass.

```
class AA{
   public virtual int f(){…}
}
class BB: AA{
   public sealed override int f(){…}
}
class CC: BB{
   public  override int f(){…}    // compile time error
}
```

A sealed class implicitly seals all the virtual functions and prevents subtyping.

```
sealed class DD{ … }
```

Static constructor

Static constructor

```
 public class Set{
   static int capacity;
   static int instances=0;
   static Set(){
     capacity=20;
   }
   // …
 }
```

A static constructor executes once per type, rather than once per instance.

A static constructor executes before any instances of the type are created and before any other static members are accessed.

A type can define only one static constructor.

The static constructor has no parameters and it has the same name as the class.

It can access only the static members from the class.

Static field assignments occur before the static constructor is called, in the declaration order in which the fields appear.

Static Classes

Static classes

A class can be marked `static`, indicating that it contains only static members (fields, methods, properties).

```
static class EncodingUtils{
 public static String encode(String txt){…}
   public static String decode(String txt){…}
}
```

A static class cannot be subclassed.

```
System.Console
System.Math
```

Partial Classes

Partial classes

Partial classes allow a class definition to be split—typically across multiple files.

A common usage is for a partial class to be auto-generated from some source, and for that class to be augmented with manually added methods.

```
// PaymentFormGen.cs - auto-generated
partial class PaymentForm { ... }
// PaymentForm.cs - manually added
partial class PaymentForm { ... }
```

Each participant must have the `partial` declaration.

Participants cannot have conflicting members.

Partial classes are resolved entirely by the compiler, which means that each participant must be available at compile time and must reside in the same assembly.

Partial Methods

Partial methods

A partial class may contain *partial methods*.

```
// PaymentFormGen.cs - auto-generated
partial class PaymentForm { ...
 partial void ValidatePayment(decimal amount);
}
// PaymentForm.cs - manually
partial class PaymentForm { ...
  partial void ValidatePayment(decimal amount) {
    if (amount > 100) ...
 }
}
```

A partial method consists of two parts: a *definition* and an *implementation*. The definition is typically written by a code generator, and the implementation is typically manually added. If an implementation is not provided, the definition of the partial method is compiled away.

`Partial` methods must be `void` and are implicitly `private`.

as and is Operators

The `as` operator performs a downcast that evaluates to null if the downcast fails:

```
Person pers=new Person();
Student st= pers as Student();     //st is null
Person pers1=new Student();
Student st1=pers1 as Student();    //st is not null
```

The `is` operator tests whether a downcast would succeed (i.e., whether an object derives from a specified class or implements an interface). It is often used to test before downcasting.

```
 if ( pers1 is Student){ … }
```

Implementing interfaces

```
public interface ILog{
   void Write(String mess);
}
public interface IFile{
   void Write(String mess);
}
public class MyFile:ILog, IFile{
   public void Write(String mess){
    Console.WriteLine(mess);
   }
   void ILog.Write(String mess){
    Console.WriteLine(“ILog: {0}“, mess);
   }
}
…
void Main(){
   ILog ilog=new MyFile();
   IFile ifile=new MyFile();
   ifile.Write(“ana”);        //ana
   ilog.Write(“ana”);     //ILog: ana
}
```

Object class

All classes inherits from Object class (from System namespace).

```
public class Object {
   public extern Type GetType( );
   public virtual bool Equals (object obj);
   public virtual int GetHashCode( );
   public virtual string ToString( );
   protected override void Finalize( );
   public static bool Equals (object objA, object objB);
   public static bool ReferenceEquals (object objA, object objB);
   //…
}
```