

# Uvod v C#

Drugi del

## Contents

- Inheritance
- Interfaces
- Delegates
- Exceptions
- Namespaces and Assemblies
- Attributes
- Threads
- XML Comments

# Dedovanje

# Sintaksa

```
class A {                               // base class
    int a;
    public A() {...}
    public void F() {...}
}
```

```
class B : A {                             // subclass (inherits from A, extends A)
    int b;
    public B() {...}
    public void G() {...}
}
```

- B inherits *a* and *F()*, it adds *b* and *G()*
  - constructors are not inherited
  - inherited methods can be overridden (see later)
- Single inheritance: a class can only inherit from one base class, but it can implement multiple interfaces.
- A class can only inherit from a class, not from a struct.
- Structs cannot inherit from another type, but they can implement multiple interfaces.
- A class without explicit base class inherits from *object*.

# Prirejanje in preverjanje tipov

```
class A {...}
class B : A {...}
class C: B {...}
```

## Assignments

```
A a = new A();    // static type of a: the type specified in the declaration (here A)
                  // dynamic type of a: the type of the object in a (here also A)
a = new B();      // dynamic type of a is B
a = new C();      // dynamic type of a is C

B b = a;         // forbidden; compilation error
```

## Run time type checks

```
a = new C();
if (a is C) ...    // true, if dynamic type of a is C or a subclass; otherwise false
if (a is B) ...    // true
if (a is A) ...    // true, but warning because it makes no sense

a = null;
if (a is C) ...    // false: if a == null, a is T always returns false
```

# Kaste preverjenih tipov

## Cast

```
A a = new C();  
B b = (B) a;           // if (a is B) stat.type(a) is B in this expression; else exception  
C c = (C) a;
```

```
a = null;  
c = (C) a;           // ok → null can be casted to any reference type
```

## as

```
A a = new C();  
B b = a as B;         // if (a is B) b = (B)a; else b = null;  
C c = a as C;
```

```
a = null;  
c = a as C;          // c == null
```

# Prekrivanje metod

Only methods that are declared as **virtual** can be overridden in subclasses

```
class A {  
    public      void F() {...} // cannot be overridden  
    public virtual void G() {...} // can be overridden in a subclass  
}
```

Overriding methods must be declared as **override**

```
class B : A {  
    public      void F() {...} // warning: hides inherited F() → use new  
    public      void G() {...} // warning: hides inherited G() → use new  
    public override void G() { // ok: overrides inherited G  
        ... base.G(); // calls inherited G()  
    }  
}
```

- Method signatures must be identical
  - same number and types of parameters (including function type)
  - same visibility (public, protected, ...).
- Properties and indexers can also be overridden (virtual, override).
- Static methods cannot be overridden.

# Dinamično povezovanje (poenostavljeno)

```
class A {  
    public virtual void WhoAreYou() { Console.WriteLine("I am an A"); }  
}  
  
class B : A {  
    public override void WhoAreYou() { Console.WriteLine("I am a B"); }  
}
```

**A message invokes the method belonging to the **dynamic type** of the receiver**  
(not quite true, see later)

```
A a = new B();  
a.WhoAreYou();           // "I am a B"
```

Every method that can work with *A* can also work with *B*

```
void Use (A x) {  
    x.WhoAreYou();  
}
```

```
Use(new A());           // "I am an A"  
Use(new B());           // "I am a B"
```

# Skrivanje

Members can be declared as **new** in a subclass.

They *hide* inherited members with the same name.

```
class A {  
    public int x;  
    public void F() {...}  
    public virtual void G() {...}  
}
```

```
class B : A {  
    public new int x;  
    public new void F() {...}  
    public new void G() {...}  
}
```

```
B b = new B();  
b.x = ...;           // accesses B.x  
b.F(); ... b.G();   // calls B.F and B.G
```

```
((A)b).x = ...;     // accesses A.x !  
((A)b).F(); ... ((A)b).G(); // calls A.F and A.G !
```



# Dinamično povezovanje (s skrivanjem)

```
class A {  
    public virtual void WhoAreYou() { Console.WriteLine("I am an A"); }  
}  
  
class B : A {  
    public override void WhoAreYou() { Console.WriteLine("I am a B"); }  
}  
  
class C : B {  
    public new virtual void WhoAreYou() { Console.WriteLine("I am a C"); }  
}  
  
class D : C {  
    public override void WhoAreYou() { Console.WriteLine("I am a D"); }  
}
```

```
C c = new D();  
c.WhoAreYou();    // "I am a D"
```

```
A a = new D();  
a.WhoAreYou();    // "I am a B" !!
```

# Fragile base class problem

## Initial situation

```
class LibraryClass {
    public void CleanUp() { ... }
}
class MyClass : LibraryClass {
    public void Delete() { ... erase the hard disk ... }
}
```

## Later: vendor ships new version of *LibraryClass*

```
class LibraryClass {
    string name;
    public virtual void Delete() { name = null; }
    public void CleanUp() { Delete(); ... }
}
```

- In Java the call *myObj.CleanUp()* would erase the hard disk!
- In C# nothing happens, as long as *MyClass* is not recompiled. *MyClass* still relies on the old version of *LibraryClass* (**Versioning**)  
→ old *CleanUp()* does not call *LibraryClass.Delete()*.
- If *MyClass* is recompiled, the compiler forces *Delete* to be declared as *new* or *override*.

# Konstruktorji in dedovanje

## Implicit call of the base class constructor

```
class A {  
    ...  
}
```

```
class B : A {  
    public B(int x) {...}  
}
```

```
B b = new B(3);
```

### OK

- default constr. A()
- B(int x)

```
class A {  
    public A() {...}  
}
```

```
class B : A {  
    public B(int x) {...}  
}
```

```
B b = new B(3);
```

### OK

- A()
- B(int x)

```
class A {  
    public A(int x) {...}  
}
```

```
class B : A {  
    public B(int x) {...}  
}
```

```
B b = new B(3);
```

### Error!

- no explicit call of the A() constructor
- default constr. A() does not exist

## Explicit call

```
class A {  
    public A(int x) {...}  
}
```

```
class B : A {  
    public B(int x)  
        : base(x) {...}  
}
```

```
B b = new B(3);
```

### OK

- A(int x)
- B(int x)

# Vidnost *protected* in *internal*

<b>protected</b>	Visible in declaring class and its subclasses (more restrictive than in Java)
<b>internal</b>	Visible in declaring assembly (see later)
<b>protected internal</b>	Visible in declaring class, its subclasses and the declaring assembly

## Example

```
class Stack {
    protected int[] values = new int[32];
    protected int top = -1;
    public void Push(int x) {...}
    public int Pop() {...}
}
class BetterStack : Stack {
    public bool Contains(int x) {
        foreach (int y in values) if (x == y) return true;
        return false;
    }
}
class Client {
    Stack s = new Stack();
    ... s.values[0] ... // compilation error!
}
```

# Abstraktni razredi

## Example

```
abstract class Stream {  
    public abstract void Write(char ch);  
    public void WriteString(string s) { foreach (char ch in s) Write(s); }  
}  
  
class File : Stream {  
    public override void Write(char ch) {... write ch to disk ...}  
}
```

## Note

- Abstract methods do not have an implementation.
- Abstract methods are implicitly *virtual*.
- If a class has abstract methods it must be declared *abstract* itself.
- One cannot create objects of an abstract class.

# Abstraktne lastnosti in indekserji

## Example

```
abstract class Sequence {  
    public abstract void Add(object x);           // method  
    public abstract string Name { get; }         // property  
    public abstract object this [int i] { get; set; } // indexer  
}  
  
class List : Sequence {  
    public override void Add(object x) {...}  
    public override string Name { get {...} }  
    public override object this [int i] { get {...} set {...} }  
}
```

## Note

- Overridden indexers and properties must have the same get and set methods as in the base class

# Zapečateni razredi

## Example

```
sealed class Account : Asset {  
    long val;  
    public void Deposit (long x) { ... }  
    public void Withdraw (long x) { ... }  
    ...  
}
```

## Note

- *sealed* classes cannot be extended (same as *final* classes in Java), but they can inherit from other classes.
- *override* methods can be declared as *sealed* individually.
- Reason:
  - Security (avoids inadvertent modification of the class semantics)
  - Efficiency (methods can possibly be called using static binding)

# Vmesniki (interfaces)



# Sintaksa

```
public interface IList : ICollection, IEnumerable {  
    int Add (object value);           // methods  
    bool Contains (object value);  
    ...  
    bool IsReadOnly { get; }         // property  
    ...  
    object this [int index] { get; set; } // indexer  
}
```

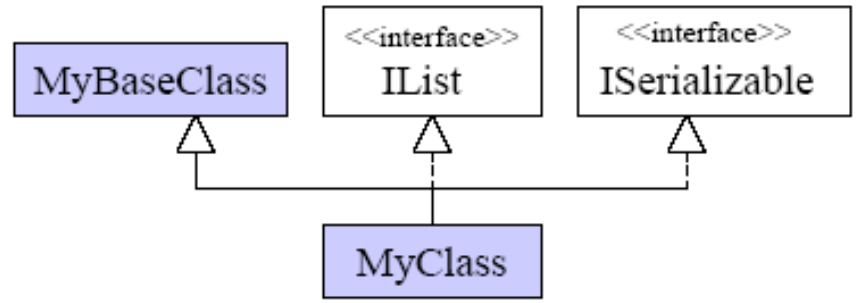
- Interface = purely abstract class; only signatures, no implementation.
- May contain **methods**, **properties**, **indexers** and **events** (no fields, constants, constructors, destructors, operators, nested types).
- Interface members are implicitly *public abstract (virtual)*.
- Interface members must not be *static*.
- Classes and structs may implement multiple interfaces.
- Interfaces can extend other interfaces.

# Implementacija vmesnikov

```
class MyClass : MyBaseClass, IList, ISerializable {  
    public int Add (object value) {...}  
    public bool Contains (object value) {...}  
    ...  
    public bool IsReadOnly { get {...} }  
    ...  
    public object this [int index] { get {...} set {...} }  
}
```

- A class can inherit from a *single base class*, but implement *multiple interfaces*. A struct cannot inherit from any type, but can implement multiple interfaces.
- Every interface member (method, property, indexer) must be *implemented* or *inherited* from a base class.
- Implemented interface methods must *not* be declared as *override*.
- Implemented interface methods can be declared *virtual* or *abstract* (i.e. an interface can be implemented by an abstract class).

# Delo z vmesniki



Assignments:

```
MyClass c = new MyClass();
IList list = c;
```

Method calls:

```
list.Add("Tom"); // dynamic binding => MyClass.Add
```

Type checks:

```
if (list is MyClass) ... // true
```

Type casts:

```
c = list as MyClass;
c = (MyClass) list;

ISerializable ser = (ISerializable) list;
```

# Primer

```
interface ISimpleReader {  
    int Read();  
}
```

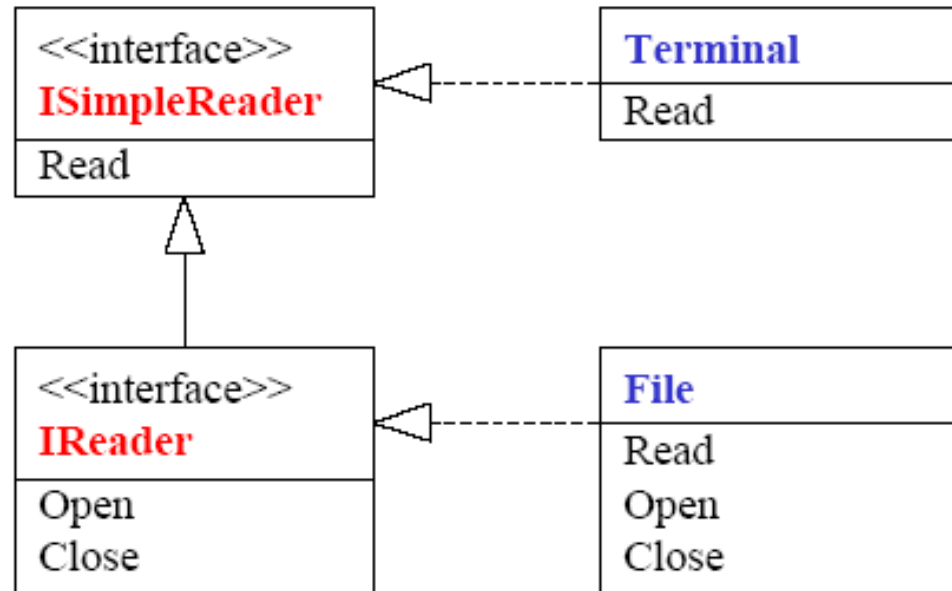
```
interface IReader : ISimpleReader {  
    void Open(string name);  
    void Close();  
}
```

```
class Terminal : ISimpleReader {  
    public int Read() { ... }  
}
```

```
class File : IReader {  
    public int Read() { ... }  
    public void Open(string name) { ... }  
    public void Close() { ... }  
}
```

```
ISimpleReader sr = null;    // null can be assigned to any interface variable  
sr = new Terminal();  
sr = new File();
```

```
IReader r = new File();  
sr = r;
```



# Delegati in dogodki

# Delegati

Delegates are reference types which allow indirect calls to methods.

A delegate instance holds references to some number of methods, and by invoking the delegate one causes all of these methods to be called.

The usefulness of delegates lies in the fact that the functions which invoke them are blind to the underlying methods they thereby cause to run (see, for instance, the discussion of events, below).

# Delegati

From this brief description, it can be seen that delegates are functionally rather similar to C++'s 'function pointers'.

However, it is important to bear in mind two main differences. Firstly, delegates are reference types rather than value types. Secondly, some single delegates can reference multiple methods

# Delegate Declaration and Instantiation

Each delegate is limited to referencing methods of a particular kind only. The type is indicated by the delegate declaration - the input parameters and return type given in the delegate declaration must be shared by the methods its delegate instances reference.

To illustrate this: a delegate specified as below can be used to refer only to methods which have a single String input and no return value:

```
public delegate void Print (String s);
```



Suppose, for instance, that a class contains the following method:

```
public void realMethod (String myString) {  
    // method code  
}
```

Another method in this class could then instantiate the 'Print' delegate in the following way, so that it holds a reference to 'realMethod':

```
Print delegateVariable = new Print(realMethod);
```

# Razredi - dogodki

- Jezikovna podpora dogodkom
- Ključna beseda **event** nam omogoča določiti delegata, ki bo ustrezal poklicanim metodam ob nastopu nekega dogodka.

```
public event AlarmEventHandler Alarm;
```

- Dogodek sprožimo s klicem dogodka

```
Alarm();
```

- Odjemalci se prijavijo na dogodek z operatorjem +=

```
eventSource.Alarm += new  
AlarmEventHandler(eventListener.AlarmRang);
```

# Deklaracija delegata, prireditev metode delegatu

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);
```

Calling a delegate variable

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

# Prerejanje različnih metod

Every matching method can be assigned to a delegate variable

```
void SayGoodBye(string sender) {  
    Console.WriteLine("Good bye from " + sender);  
}  
  
greetings = new Notifier(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 parameter, etc.

# Creating a delegate value

```
new DelegateType (obj.Method)
```

- A delegate variable stores a method and its receiver, but no parameters !  
`new Notifier(myObj.SayHello);`
- *obj* can be *this* (and can be omitted)  
`new Notifier(SayHello)`
- *Method* can be *static*. In this case the class name must be specified instead of *obj*.  
`new Notifier(MyClass.StaticSayHello);`
- *Method* must not be *abstract*, but it can be *virtual*, *override*, or *new*.
- *Method* signature must match the signature of *DelegateType*
  - same number of parameters
  - same parameter types (including the return type)
  - same parameter kinds (ref, out, value)

# Multicast delegati

A delegate variable can hold multiple values at the same time

```
Notifier greetings;  
greetings = new Notifier(SayHello);  
greetings += new Notifier(SayGoodBye);
```

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

```
greetings -= new Notifier(SayHello);
```

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

## Note

- if the multicast delegate is a function, the value of the last call is returned
- if the multicast delegate has an out parameter, the parameter of the last call is returned

# Events = Special Delegate Variables

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

```
class View1 {  
    public View1(Model m) { m.notifyViews += new Notifier(this.Update1); }  
    void Update1(string sender) { Console.WriteLine(sender + " was changed"); }  
}  
class View2 {  
    public View2(Model m) { m.notifyViews += new Notifier(this.Update2); }  
    void Update2(string sender) { Console.WriteLine(sender + " was changed"); }  
}
```

```
class Test {  
    static void Main() {  
        Model m = new Model(); new View1(m); new View2(m);  
        m.Change();  
    }  
}
```

Why events instead of normal delegate variables?

Only the class that declares the event can fire it (better abstraction).

# Izjeme (exceptions)



# Stack *try*

```
FileStream s = null;
try {
    s = new FileStream(curName, FileMode.Open);
    ...
} catch (FileNotFoundException e) {
    Console.WriteLine("file {0} not found", e.FileName);
} catch (IOException) {
    Console.WriteLine("some IO exception occurred");
} catch {
    Console.WriteLine("some unknown error occurred");
} finally {
    if (s != null) s.Close();
}
```

- *catch* clauses are checked in sequential order.
- *finally* clause is always executed (if present).
- Exception parameter name can be omitted in a *catch* clause.
- Exception type must be derived from *System.Exception*.  
If exception parameter is missing, *System.Exception* is assumed.

# System.Exception

## Properties

e.Message	the error message as a string; set in <i>new Exception(msg)</i> ;
e.StackTrace	trace of the method call stack as a string
e.Source	the application or object that threw the exception
e.TargetSite	the method object that threw the exception
...	

## Methods

e.ToString()	returns the name of the exception
...	

# *Throwing an Exception*

## **By an invalid operation (implicit exception)**

Division by 0

Index overflow

Access via a null reference

...

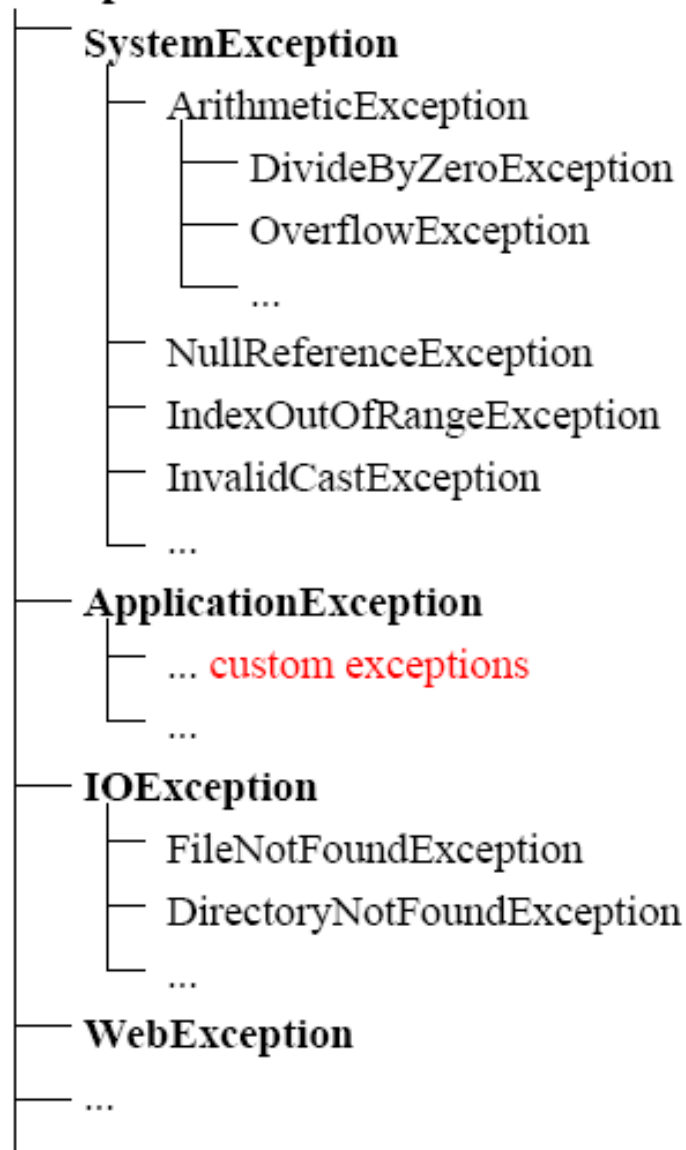
## **By a throw statement (explicit exception)**

```
throw new FunnyException(10);
```

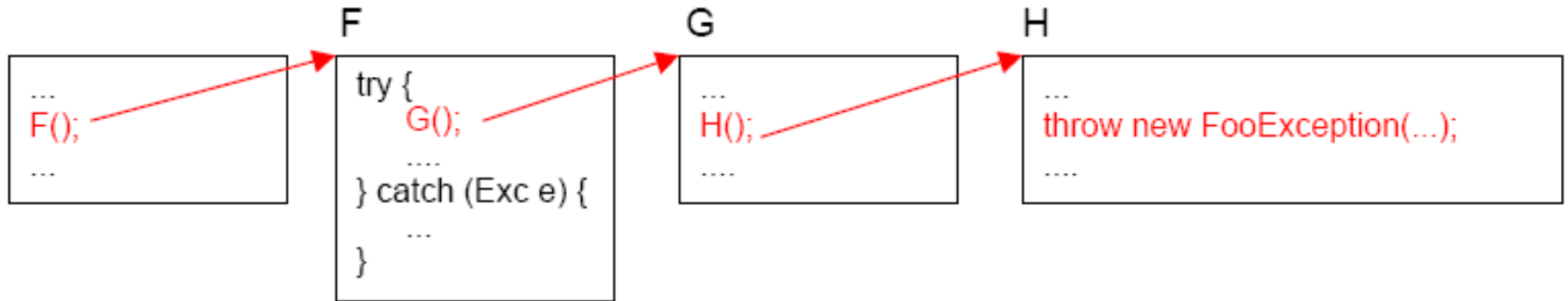
```
class FunnyException : ApplicationException {  
    public int errorCode;  
    public FunnyException(int x) { errorCode = x; }  
}
```

# Hierarhija izjem

## Exception



# Searching for a catch Clause



Caller chain is traversed backwards until a method with a matching catch clause is found.  
If none is found => Program is aborted with a stack trace

**Exceptions don't have to be caught in C#** (in contrast to Java)

No distinction between

- *checked exceptions* that have to be caught, and
- *unchecked exceptions* that don't have to be caught

Advantage: convenient

Disadvantage: less robust software

# No Throws Clause in Method Signature

## Java

```
void myMethod() throws IOException {  
    ... throw new IOException(); ...  
}
```

Callers of *myMethod* must either

- catch *IOException* or
- specify *IOExceptions* in their own signature

## C#

```
void myMethod() {  
    ... throw new IOException(); ...  
}
```

Callers of *myMethod* may handle *IOException* or not.

- + convenient
- less robust

# *Namespaces and Assemblies*

# C# Namespaces vs. Java Packages

## C#

A file may contain multiple namespaces

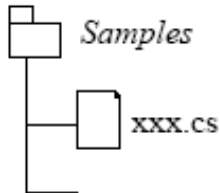
*xxx.cs*

```
namespace A {...}  
namespace B {...}  
namespace C {...}
```

Namespaces and classes are not mapped to directories and files

*xxx.cs*

```
namespace A {  
    class C {...}  
}
```



## Java

A file may contain just 1 package

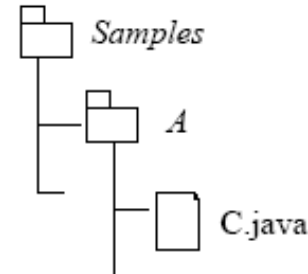
*xxx.java*

```
package A;  
...  
...
```

Packages and classes are mapped to directories and files

*C.java*

```
package A;  
class C {...}
```





# Namespaces vs. Packages (continued)

## C#

Imports *namespaces*

```
using System;
```

Namespaces are imported in other Namesp.

```
using A;  
namespace B {  
    using C;  
    ...  
}
```

Alias names allowed

```
using F = System.Windows.Forms;  
...  
F.Button b;
```

for explicit qualification and short names.

## Java

Imports *classes*

```
import java.util.LinkedList;  
import java.awt.*;
```

Classes are imported in files

```
import java.util.LinkedList;
```

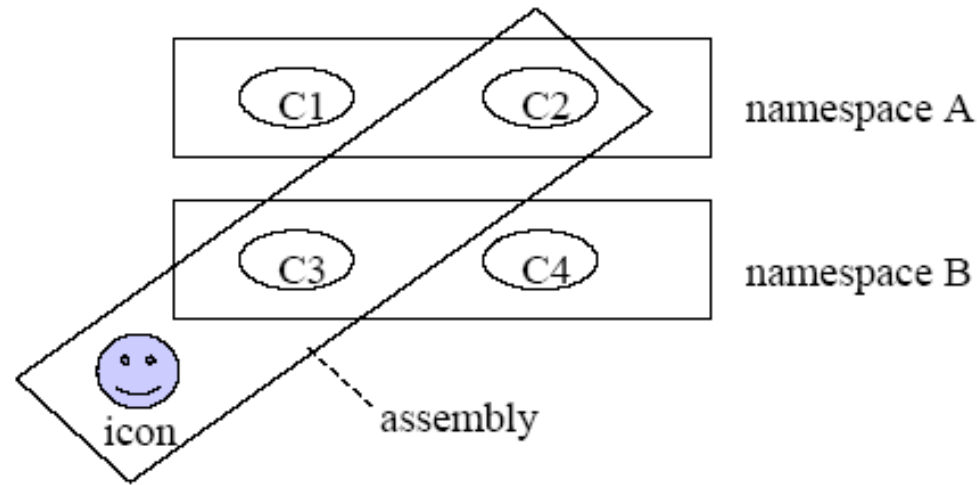
Java has visibility *package*

```
package A;  
class C {  
    void f() {...} // package  
}
```

C# has only visibility *internal* (!= namespace)

# Assemblies

Run time unit consisting of types and other resources (e.g. icons)



- Unit of deployment: assembly is smallest unit that can be deployed individually
- Unit of versioning: all types in an assembly have the same version number

Often: 1 assembly = 1 namespace = 1 program

But:

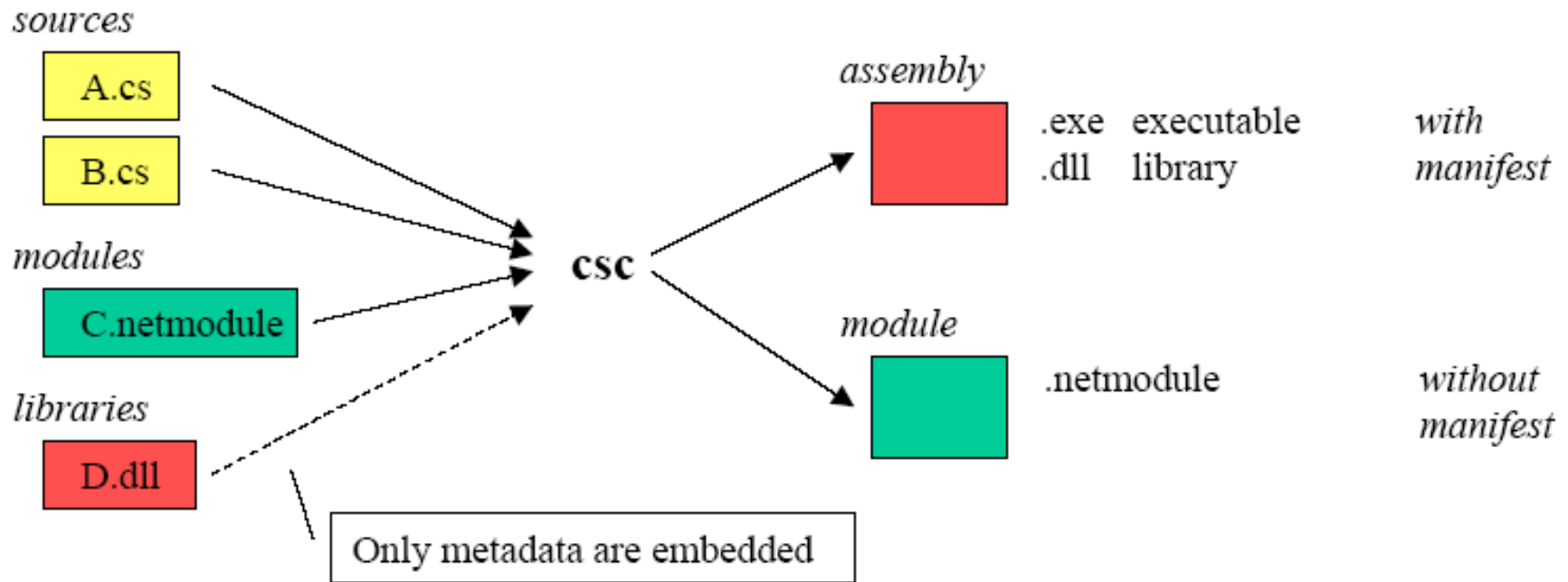
- one assembly may consist of multiple namespaces.
- one namespace may be spread over several assemblies.
- an assembly may consist of multiple files, held together by a *manifest* ("table of contents")

Assembly JAR file in Java

Assembly Component in .NET

# Kako se tvorijo assemblies

Every compilation creates either an *assembly* or a *module*



Other modules/resources can be added with the assembly linker (al)

Difference to Java: Java creates a \*.class file for every class

# Opcije prevajalnika

Which output file should be generated?

<b>/t[arget]: exe</b>	output file = console application (default)
<b>winexe</b>	output file = Windows GUI application
<b>library</b>	output file = library (DLL)
<b>module</b>	output file = module (.netmodule)

<b>/out:name</b>	specifies the name of the assembly or module
default for /t:exe	<u>name.exe</u> , where <i>name</i> is the name of the source file containing the <i>Main</i> method
default for /t:library	<u>name.dll</u> , where <i>name</i> is the name of the first source file
Example:	csc /t:library /out:MyLib.dll A.cs B.cs C.cs

<b>/doc:name</b>	generates an XML file with the specified name from <i>///</i> comments
------------------	--

# Opcije prevajalnika

How should libraries and modules be embedded?

**/r[eference]:*name*** makes metadata in *name* (e.g. *xxx.dll*) available in the compilation.  
*name* must contain metadata.

**/lib:dirpath{,dirpath}** specifies the directories, in which libraries are searched that are referenced by /r.

**/addmodule:name {,name}** adds the specified modules (e.g. *xxx.netmodule*) to the generated assembly.  
At run time these modules must be in the same directory as the assembly to which they belong.

## Example

```
csc /r:MyLib.dll /lib:C:\project A.cs B.cs
```

# Primeri prevajanj

`csc A.cs`  $\Rightarrow$  `A.exe`

`csc A.cs B.cs C.cs`  $\Rightarrow$  `B.exe` (if *B.cs* contains *Main*)

`csc /out:X.exe A.cs B.cs`  $\Rightarrow$  `X.exe`

`csc /t:library A.cs`  $\Rightarrow$  `A.dll`

`csc /t:library A.cs B.cs`  $\Rightarrow$  `A.dll`

`csc /t:library /out:X.dll A.cs B.cs`  $\Rightarrow$  `X.dll`

`csc /r:X.dll A.cs B.cs`  $\Rightarrow$  `A.exe` (where *A* or *B* reference types in *X.dll*)

`csc /addmodule:Y.netmodule A.cs`  $\Rightarrow$  `A.exe` (*Y* is added to this assembly)

# Atributi

# Atributi

- **Atributi** so deklarativni elementi jezika
  - pridevniki so na primer tudi deklarativni elementi
  - omogočajo dekoracijo elementov kode z dodatno informacijo
    - assembly, modul, tip, član tipa, return value, parameter
- Atributi posplošujejo koncept deklarativnega elementa
  - atributi niso omejeni na vnaprej določeno množico
  - lahko se zmišljujemo svoje nove attribute
  - atributi in njihove vrednosti so dosegljivi v času prevajanja in v času izvajanja

```
[Obsolete("This class is obsolete")]  
class A  
{  
    public void F() {}  
}
```



# Atributi

## User-defined metainformation about program elements

- Can be attached to types, members, assemblies, etc.
- Extend predefined attributes such as *public*, *sealed* or *abstract*.
- Are implemented as classes that are derived from *System.Attribute*.
- Are stored in the metadata of an assembly.
- Often used by CLR services (serialization, remoting, COM interoperability)
- Can be queried at run time.

## Example

```
[Serializable]  
class C {...} // makes the class serializable
```

Also possible to attach multiple attributes

```
[Serializable] [Obsolete]  
class C {...}
```

```
[Serializable, Obsolete]  
class C {...}
```

# Attributes parameters

## Example

```
[Obsolete("Use class C1 instead", IsError=true)] // causes compiler message saying
public class C {...}                          // that C is obsolete
```

positional parameter

name parameters  
come after pos. parameters

Positional parameter = parameter of the attribute's constructor

Name parameter = a property of the attribute

## Attributes are declared as classes

```
public class ObsoleteAttribute : Attribute { // class name ends with "Attribute"
    public string Message { get; }          // but can be used as "Obsolete"
    public bool IsError { get; set; }
    public ObsoleteAttribute() {...}
    public ObsoleteAttribute(string msg) {...}
    public ObsoleteAttribute(string msg, bool error) {...}
}
```

## Valid variants:

```
[Obsolete]
[Obsolete("some Message")]
[Obsolete("some Message", false)]
[Obsolete("some Message", IsError=false)]
```

value must be a constant

# Atributi

## Nekateri vnaprej definirani atributi

<b>Ime atributa</b>	<b>Opis</b>
<code>Browsable</code>	Vključenost lastnosti v okno lastnosti
<code>Serializable</code>	Določa naj bo razred ali struktura serializabilna
<code>Obsolete</code>	Določa, da je nek element kode zastarel
<code>Conditional</code>	Določa vključenost metode v odvisnosti od predproc. simbola

# Atributi

## Zgled uporabe atributa Conditional

```
#define Debug
public class Debug {
    [Conditional("Debug")]
    public static void Assert(bool cond, String s) {
        if (!cond) {
            throw new AssertionError(s);
        }
    }
    void DoSomething() {
        ...
        // If Debug is not defined, the next line is
        // not even called
        Assert((x == y), "X should equal Y");
        ...
    }
}
```

# Še en primer: atribut Conditional

Allows a conditional call of methods

```
#define debug // preprocessor command

class C {

    [Conditional("debug")] // only possible for void methods
    static void Assert (bool ok, string errorMsg) {
        if (!ok) {
            Console.WriteLine(errorMsg);
            System.Environment.Exit(0); // graceful program termination
        }
    }

    static void Main (string[] arg) {
        Assert(arg.Length > 0, "no arguments specified");
        Assert(arg[0] == "...", "invalid argument");
        ...
    }
}
```

*Assert* is only called, if *debug* was defined.

Also useful for controlling trace output.

# Tvoji lastni atributi

## Declaration

```
[AttributeUsage(AttributeTargets.Class|AttributeTargets.Interface, Inherited=true)]
class Comment : 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"; }
}
```

## Use

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

## Querying the attribute at run time

```
class Attributes {
    static void Main() {
        Type t = typeof(C);
        object[] a = t.GetCustomAttributes(typeof(Comment), true);
        Comment ca = (Comment)a[0];
        Console.WriteLine(ca.Text + ", " + ca.Author);
    }
}
```

search should  
also be continued  
in subclasses

# Niti (threads)

# Sodelujoče niti (participating threads)

```
public sealed class Thread {
    public static Thread CurrentThread { get; }    // static methods
    public static void Sleep(int milliseconds) {...}
    ...
    public Thread(ThreadStart startMethod) {...} // thread creation

    public string Name { get; set; }              // properties
    public ThreadPriority Priority { get; set; }
    public ThreadState ThreadState { get; }
    public bool IsAlive { get; }
    public bool IsBackground { get; set; }
    ...
    public void Start() {...}                    // methods
    public void Suspend() {...}
    public void Resume() {...}
    public void Join() {...}                    // caller waits for the thread to die
    public void Abort() {...}                  // throws ThreadAbortException
    ...
}

public delegate void ThreadStart();           // parameterless void method

public enum ThreadPriority {AboveNormal, BelowNormal, Highest, Lowest, Normal}
public enum ThreadState {Aborted, Running, Stopped, Suspended, Unstarted, ...}
```



# Primer

```
using System;
using System.Threading;

class Printer {
    char ch;
    int sleepTime;

    public Printer(char c, int t) {ch = c; sleepTime = t;}

    public void Print() {
        for (int i = 0; i < 100; i++) {
            Console.Write(ch);
            Thread.Sleep(sleepTime);
        }
    }
}

class Test {
    static void Main() {
        Printer a = new Printer('.', 10);
        Printer b = new Printer('*', 100);
        new Thread(new ThreadStart(a.Print)).Start();
        new Thread(new ThreadStart(b.Print)).Start();
    }
}
```

**The program runs until the last thread stops.**

# Stanja niti

```
Thread t = new Thread(new ThreadStart(P));
Console.WriteLine("name={0}, priority={1}, state={2}", t.Name, t.Priority, t.ThreadState);
t.Name = "Worker"; t.Priority = ThreadPriority.BelowNormal;
t.Start();
Thread.Sleep(0);
Console.WriteLine("name={0}, priority={1}, state={2}", t.Name, t.Priority, t.ThreadState);
t.Suspend();
Console.WriteLine("state={0}", t.ThreadState);
t.Resume();
Console.WriteLine("state={0}", t.ThreadState);
t.Abort();
Thread.Sleep(0);
Console.WriteLine("state={0}", t.ThreadState);
```

## Output

```
name=, priority=Normal, state=Unstarted
name=Worker, priority=BelowNormal, state=Running
state=Suspended
state=Running
state=Stopped
```

# Primer za *Join*

```
using System;
using System.Threading;

class Test {

    static void P() {
        for (int i = 1; i <= 20; i++) {
            Console.Write('-');
            Thread.Sleep(100);
        }
    }

    static void Main() {
        Thread t = new Thread(new ThreadStart(P));
        Console.Write("start");
        t.Start();
        t.Join();
        Console.WriteLine("end");
    }
}
```

## Output

start-----end

# Medsebojno izobčenje (sinhronizacija)

lock Statement

```
lock(Variable) Statement
```

Example

```
class Account {                               // this class should behave like a monitor
    long val = 0;

    public void Deposit(long x) {
        lock (this) { val += x; } // only 1 thread at a time may execute this statement
    }

    public void Withdraw(long x) {
        lock (this) { val -= x; }
    }
}
```

Lock can be set to any object

```
object semaphore = new object();
...
lock (semaphore) { ... critical region ... }
```

No synchronized methods like in Java

# Razred Monitor

lock(v) Statement

is a shortcut for

```
Monitor.Enter(v);  
try {  
    Statement  
} finally {  
    Monitor.Exit(v);  
}
```

# Wait in Pulse

```
Monitor.Wait(lockedVar);  
Monitor.Pulse(lockedVar);  
Monitor.PulseAll(lockedVar);
```

```
wait() in Java (in Java lockedVar is always this)  
notify() in Java  
notifyAll() in Java
```

## Example

*Thread A*

```
1 lock(v) {  
  ...  
2  Monitor.Wait(v); 5  
  ...  
}
```

*Thread B*

```
3 lock(v) {  
  ...  
4  Monitor.Pulse(v);  
  ...  
} 6
```

1. *A* comes to *lock(v)* and proceeds because the critical region is free.
2. *A* comes to *Wait*, goes to sleep and releases the lock.
3. *B* comes to *lock(v)* and proceeds because the critical region is free.
4. *B* comes to *Pulse* and wakes up *A*. There can be a context switch between *A* and *B*, but not necessarily.
5. *A* tries to get the lock but fails, because *B* is still in the critical region.
6. At the end of the critical region *B* releases the lock; *A* can proceed now.

# Primer: sinhroniziran medpomnilnik

```
class Buffer {  
    const int size = 4;  
    char[] buf = new char[size];  
    int head = 0, tail = 0, n = 0;  
  
    public void Put(char ch) {  
        lock(this) {  
            while (n == size) Monitor.Wait(this);  
            buf[tail] = ch; tail = (tail + 1) % size; n++;  
            Monitor.Pulse(this);  
        }  
    }  
  
    public char Get() {  
        lock(this) {  
            while (n == 0) Monitor.Wait(this);  
            char ch = buf[head]; head = (head + 1) % size;  
            n--;  
            Monitor.Pulse(this);  
            return ch;  
        }  
    }  
}
```

If producer is faster

Put  
Put  
Put  
Put  
Get  
Put  
Get  
...

If consumer is faster

Put  
Get  
Put  
Get  
...

# Komentarji XML



# Posebni komentarji (podobno javadoc)

## Example

```
/// ... comment ...  
class C {  
    /// ... comment ...  
    public int f;  
  
    /// ... comment ...  
    public void foo() {...}  
}
```

## Compilation `csc /doc:MyFile.xml MyFile.cs`

- *Checks if comments are complete and consistent*  
e.g. if one parameter of a method is documented, all parameters must be documented;  
Names of program elements must be spelled correctly.
- *Generates an XML file with the commented program elements*  
XML can be formatted for the Web browser with XSL

# Primer komentirane kode

```
/// <summary> A counter for accumulating values and computing the mean value.</summary>
class Counter {
    /// <summary>The accumulated values</summary>
    private int value;

    /// <summary>The number of added values</summary>
    public int n;

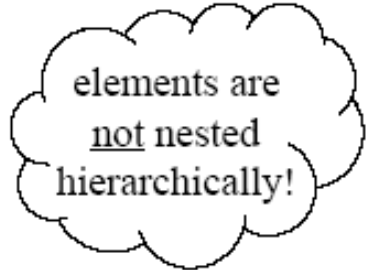
    /// <summary>Adds a value to the counter</summary>
    /// <param name="x">The value to be added</param>
    public void Add(int x) {
        value += x; n++;
    }

    /// <summary>Returns the mean value of all accumulated values</summary>
    /// <returns>The mean value, i.e. <see cref="value"/> / <see cref="n"/></returns>
    public float Mean() {
        return (float)value / n;
    }
}
```

# Tvorjena XML datoteka

```
<?xml version="1.0"?>
<doc>
  <assembly>
    <name>MyFile</name>
  </assembly>
  <members>
    <member name="T:Counter">
      <summary> A counter for accumulating values and computing the mean value.</summary>
    </member>
    <member name="F:Counter.value">
      <summary>The accumulated values</summary>
    </member>
    <member name="F:Counter.n">
      <summary>The number of added values</summary>
    </member>
    <member name="M:Counter.Add(System.Int32)">
      <summary>Adds a value to the counter</summary>
      <param name="x">The value to be added</param>
    </member>
    <member name="M:Counter.Mean">
      <summary>Returns the mean value of all accumulated values</summary>
      <returns>The mean value, i.e. <see cref="F:Counter.value"/> / <see cref="F:Counter.n"/></returns>
    </member>
  </members>
</doc>
```

XML file can be viewed in HTML using Visual Studio.



elements are  
not nested  
hierarchically!

# XML oznake (XML tags)

## Predefined Tags

### Main tags

`<summary>` *short description of a program element* `</summary>`

`<remarks>` *extensive description of a program element* `</remarks>`

`<param name="ParamName">` *description of a parameter* `</param>`

`<returns>` *description of the return value* `</returns>`

### Tags that are used within other tags

`<exception [cref="ExceptionType"]>` *used in the documentation of a method: describes an exception* `</exception>`

`<example>` *sample code* `</example>`

`<code>` *arbitrary code* `</code>`

`<see cref="ProgramElement">` *name of a crossreference link* `</see>`

`<paramref name="ParamName">` *name of a parameter* `</paramref>`

## User-defined Tags

Users may add arbitrary tags, e.g. `<author>`, `<version>`, ...

# Povzetek C#

- **Familiar**
- **Safe**
  - Strong static typing
  - Run time checks
  - Garbage Collection
  - Versioning
- **Expressive**
  - Object-oriented (classes, interfaces, ...)
  - Component-oriented (properties, events, assemblies, ...)
  - Uniform type system (boxing / unboxing)
  - Enumerations
  - Delegates
  - Indexers
  - **ref** and **out** parameters
  - Value objects on the stack
  - Threads and synchronization
  - Exceptions
  - User attributes
  - Reflection
  - ...

# Odprtokodni c#

## **MCS: The Ximian C# compiler**

The Mono C# compiler is considered feature C# 1.0 complete at this point and mature. MCS is able to compile itself and many more C# programs (there is a test suite included that you can use). It is routinely used to compile Mono, roughly 1.7 million lines of C# code.

The compiler is also fairly fast. On a IBM ThinkPad t40 it compiles 18,000 lines of C# code per second.

Work on C# 2.0 has started: some pieces of it are available on the standard compiler with the -2 switch (iterators, method conversions) and some others are available on the `gmcs' branch on CVS (generics)

## **Obtaining MCS**

The Mono C# compiler is part of the `mcs' module in the Mono CVS you can get it from our [Anonymous CVS](#) server, or you can get nightly [download page](#).

# Mono razhroščevalnik

The screenshot shows the Mono IDE interface. The main window displays a C# source file with the following code:

```
249 Console.WriteLine("Compilation succeeded");
250 if (Report.Warnings > 0) {
251     Console.WriteLine("- {0} warning(s)", Report.Warnings);
252 }
253 Console.WriteLine();
254 return 0;
255 } else {
256     Console.WriteLine("Compilation failed: {0} error(s), {1} warnings",
257         Report.Errors, Report.Warnings);
258     return 1;
259 }
260 }
261
262 static public void LoadAssembly (string assembly, bool soft)
263 {
264     Assembly a;
265     string total_log = "";
```

A breakpoint is set at line 257. The status bar at the bottom indicates the program is stopped at driver.cs:257 at 0x0817ff9a.

The Modules window shows the following loaded modules:

Module Name	✓	□	□	□
libgc.so.1	✓	□	□	✓
libnsl.so.1	✓	□	□	✓
libpthread.so.0	✓	□	□	✓
librt.so.1	✓	□	□	✓
libc.so.6	✓	□	□	✓
ld-linux.so.2	✓	□	□	✓
System	✓	✓	✓	□
mcs	✓	✓	✓	□
Svstem.Xml	✓	✓	✓	□

The Command window shows the following register values:

```
%eax = 0x832ad50
%ebx = 0x8143c18
%ecx = 0x0
%edx = 0xffff8b815
%esi = 0x0
%edi = 0xbffffe44
%ebp = 0xbffffc4c
%esp = 0xbffffc0c
%eip = 0x817ff9a
%eflags = 0x382
```

# Eclipse in mono

