Namespaces and Assemblies

Namespaces

File: XXX.cs

```
namespace A {
    ...
    namespace B {  // voller Name: A.B
        ...
    }
}
```

File: YYY.cs

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

namespace C {...}
```

File: ZZZ.cs

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

- A file can declare multiple namespaces.
- A namespace can be re-opened in another file.
- Types that are not declared in any namespace, are considered to be in a default namespace (global namespace).

Using Other Namespaces

Color.cs

```
namespace Util {
    enum Color {...}
}
```

Figures.cs

```
namespace Util.Figures {
    class Rect {...}
    class Circle {...}
}
```

Triangle.cs

```
namespace Util.Figures {
    class Triangle {...}
}
```

```
using Util.Figures;

class Test {
    Rect r;          // without qualification (because of using Util.Figures)
    Triangle t;
    Util.Color c;    // with qualification
}
```

Foreign namespaces

- must either be imported (e.g. *using Util;)*
- or specified in a qualified name (e.g. *Util.Color*)

C# Namespaces vs. Java Packages

C#

A file may contain multiple namespaces

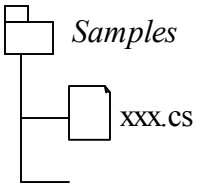
xxx.cs

```
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

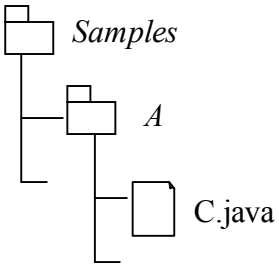
xxx.java

```
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.

```
namespace A {
    using C; // imports C into A
}            // only in this file
namespace B {
    using D;
}
```

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;
```

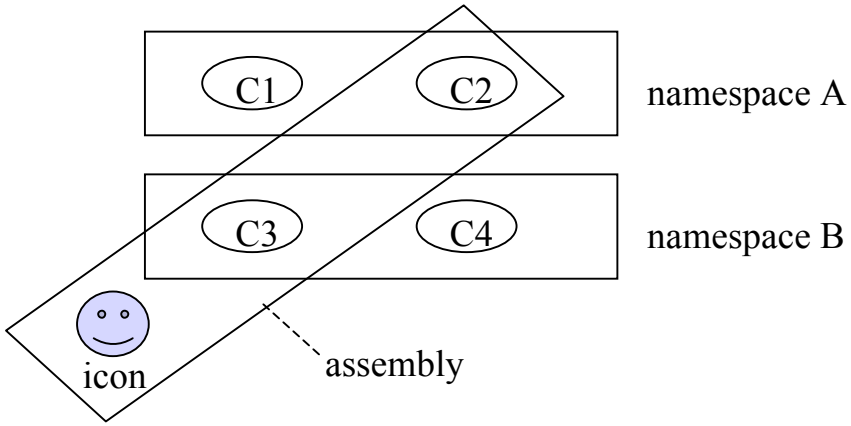
Java has visibility *package*

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

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

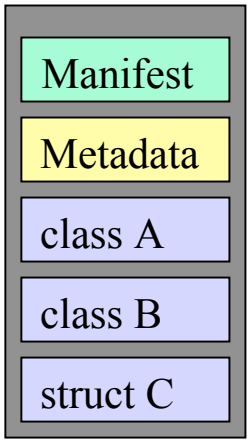
Assemblies

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



An assembly is a
- 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

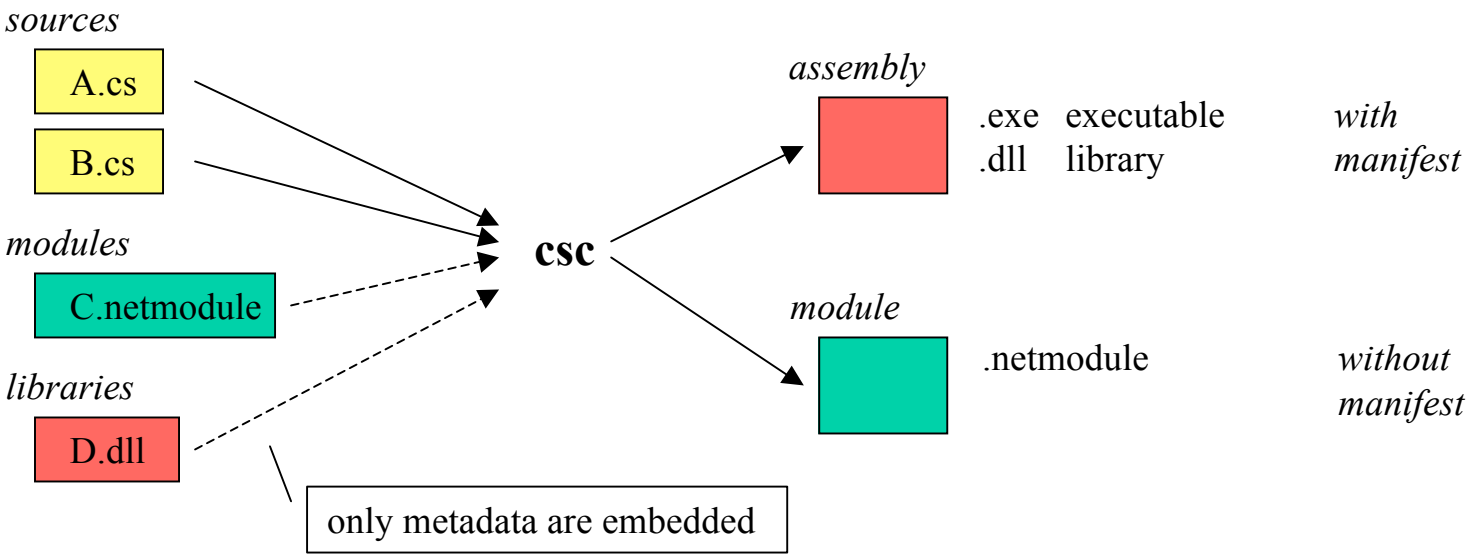
Often: 1 assembly = 1 namespace = 1 program
But:
- one assembly may consist of multiple namespaces
- one namespace may be spread over several assemblies
- an assembly may consist of multiple files, held together by a *manifest* ("table of contents")



Assembly ≈ JAR file in Java
Assembly ≈ Component in .NET

How Are Assemblies Created?

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

Compiler Options

Which output file should be generated?

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

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

/doc:*name*	generates an XML file with the specified name from /// comments

Compiler Options (continued)

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
```

Examples for Compilations

csc A.cs	=> A.exe
csc A.cs B.cs C.cs	=> B.exe (if *B.cs* contains a *Main* method)
csc /out:X.exe A.cs B.cs	=> X.exe
csc /t:library A.cs	=> A.dll
csc /t:library A.cs B.cs	=> A.dll
csc /t:library /out:X.dll A.cs B.cs	=> X.dll
csc /r:X.dll A.cs B.cs	=> A.exe (where *A* or *B* reference types in *X.dll*)
csc /addmodule:Y.netmodule A.cs	=> A.exe (*Y* is added to this assembly; but *Y.netmodule* remains as a separate file)

Loading Assemblies at Runtime

Executables are loaded when a program is invoked from the shell
(e.g., invocation of *MyApp* loads *MyApp.exe* and executes it)

Libraries (DLLs) are searched in the following directories:

- in the application directory
- in all directories that are specified in a configuration file (e.g. *MyApp.exe.config*) under the <probing> tag

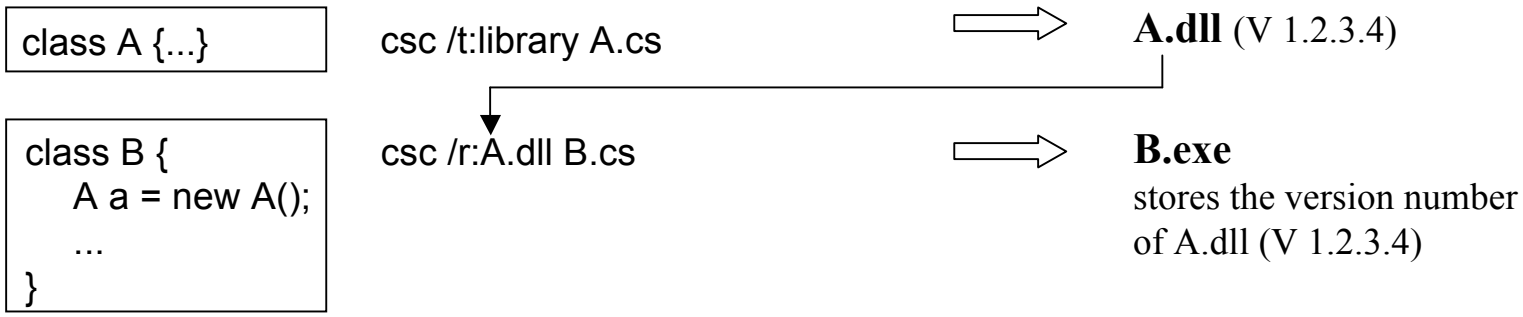
```
<configuration>
    ...
    <runtime>
        ...
        <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
            <probing privatePath="bin;bin2\subbin;bin3"/>
        </assemblyBinding>
    </runtime>
</configuration>
```

- in the Global Assembly Cache (for "shared assemblies")

Versioning of Assemblies

Causes those libraries to be loaded, which have the expected version number

Version number is stored in the assembly by the compiler



Version number is checked when the assembly is loaded

Call: B
- loads B.exe
- finds a reference to A.dll (V 1.2.3.4)
- loads A.dll in version V 1.2.3.4
  (even if there are other versions of A.dll)

avoids "DLL hell"

Private and Public Assemblies

Private Assembly

- is used by only one application
- resides in the application directory
- does not have a "strong name"
- cannot be signed

Public Assembly (or shared assembly)

- can be used by all applications
- resides in the Global Assembly Cache (GAC)
- has a "strong name"
- can be signed
- GAC can hold assemblies with the same name but different version numbers

Strong Names

Consist of 4 parts

- the *name* of the assembly (e.g. A.dll)
- the *version number* of the assembly (e.g. 1.0.1033.17)
- the *culture* of the assembly (System.Globalization.CultureInfo)
- the *public key* of the assembly

can be set with attributes

```
using System.Reflection;
[assembly:AssemblyVersion("1.0.1033.17")]
[assembly:AssemblyCulture("en-US")]
[assembly:AssemblyKeyFile("myKeyFile.snk")]
class A {
    ...
}
```

default: 0.0.0.0
default: neutral

Version number

Major.Minor.Build.Revision

Build and *Revision* can be specified as * in the *AssemblyVersion* Attribute:

```
[assembly:AssemblyVersion("1.0.*")]
```

The compiler chooses suitable values then.

Signing Assemblies

Using Public Key Cryptography

1. Generate a key file with *sn.exe* (Strong Name Tool)

```
sn  /k  myKeyFile.snk
```

myKeyFile.snk is generated and contains:
- public key (128 bytes)
- private key (436 bytes)

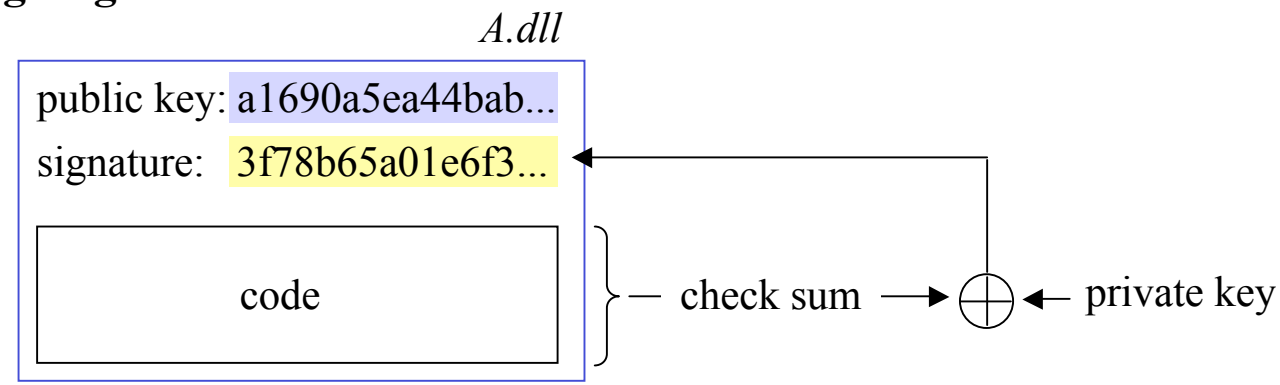
2. Sign the assembly with the AssemblyKeyFile attribute

```
[assembly:AssemblyKeyFile("myKeyFile.snk")]
public class A {...}
```

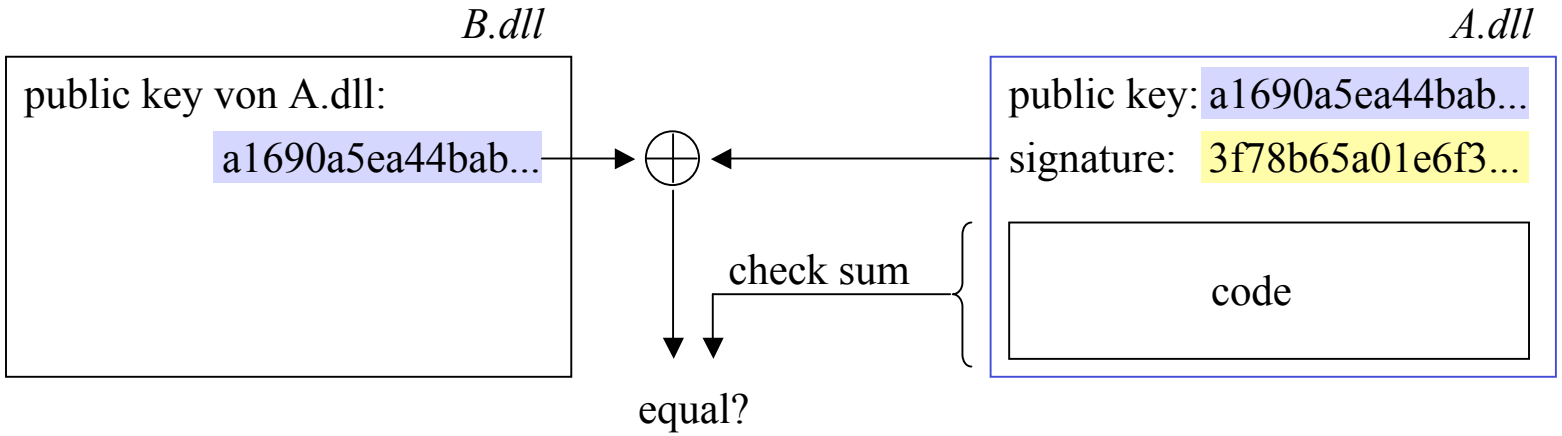
During compilation
- the assembly is signed with the private key
- the public key is stored in the assembly

Checking the Signature

Signing



Checking (during loading)



Visibility Modifier "internal"

Visibility *internal* refers to what is visible during a compilation

csc A.cs B.cs C.cs — All *internal* members of *A*, *B*, *C* see each other.

csc /addmodule:A.netmodule,B.netmodule C.cs

C sees the *internal* members of *A* and *B*

(*A* and *B* may be written in different languages)