Namespaces und 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 {
    ...
}
```

- Eine Datei kann mehrere Namespaces enthalten.
- Ein Namespace kann über mehrere Dateien verteilt sein.
  Gleichnamige Namespaces bilden einen gemeinsamen Deklarationsraum.
- Typen die in keinem Namespace enthalten sind, kommen in Default-Namespace (Global Namespace).

Benutzung von 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;          // Benutzung ohne Qualifikation (weil using Util.Figures)
    Triangle t;
    Util.Color c;    // Benutzung mit Qualifikation
}
```

Fremde Namespaces müssen entweder

- mit *using* importiert oder
- als Qualifikation vor verwendeten Namen geschrieben werden

Fast jedes Programm benötigt Namespace System => using System;

C#-Namespaces vs. Java-Pakete

C#

Datei kann mehrere Namespaces enthalten

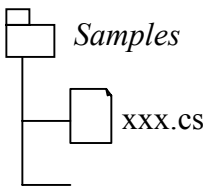
xxx.cs

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

Namespaces werden nicht auf Verzeichnisse abgebildet

xxx.cs

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



Java

Datei kann nur 1 Paketangabe enthalten

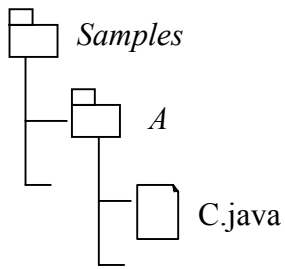
xxx.java

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

Pakete werden auf Verzeichnisse abgebildet

C.java

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



Namespaces vs. Pakete (Forts.)

C#

Es werden *Namespaces* importiert

```
using System;
```

NS werden in andere NS importiert

```
namespace A {
    using C; // gilt nur in dieser Datei
}            // für A
namespace B {
    using D;
}
```

Alias-Namen möglich

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

Wenn man explizite Qualifikation
aber kurze Namen haben will.

Java

Es werden *Klassen* importiert

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

Klassen werden in Dateien importiert

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

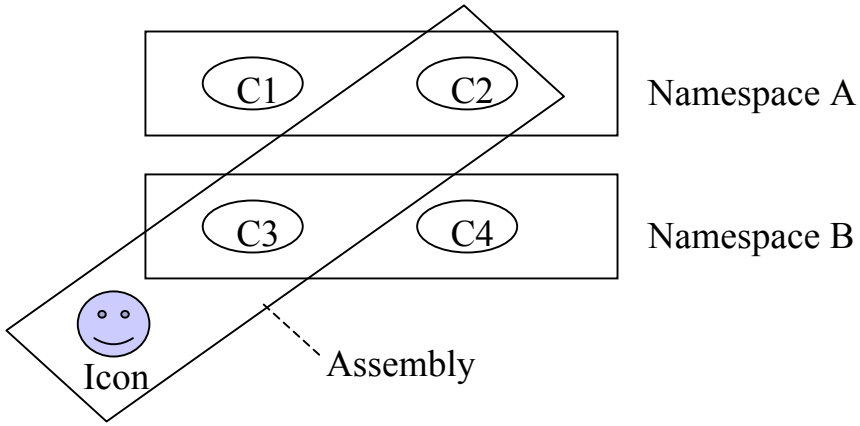
Java kennt Sichtbarkeit innerhalb Paket

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

C# kennt Sichtbarkeit in Assembly (!= Namespace)

Assemblies

Laufzeiteinheit aus mehreren Typen und sonstigen Ressourcen (z.B. Icons)



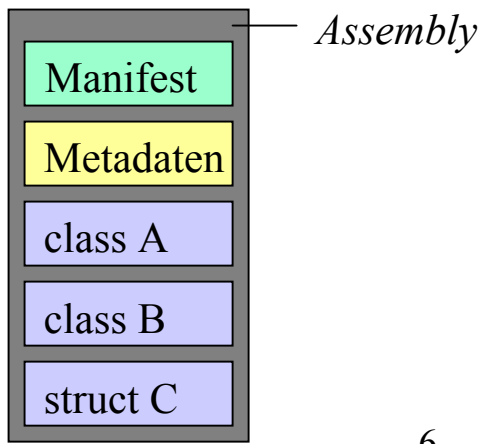
- Auslieferungseinheit: kleinere Teile als Assemblies können nicht ausgeliefert werden
- Versionierungseinheit: Alle Typen eines Assembly haben gleiche Versionsnummer

Assembly kann mehrere Namespaces enthalten.

Namespace kann auf mehrere Assemblies verteilt sein.

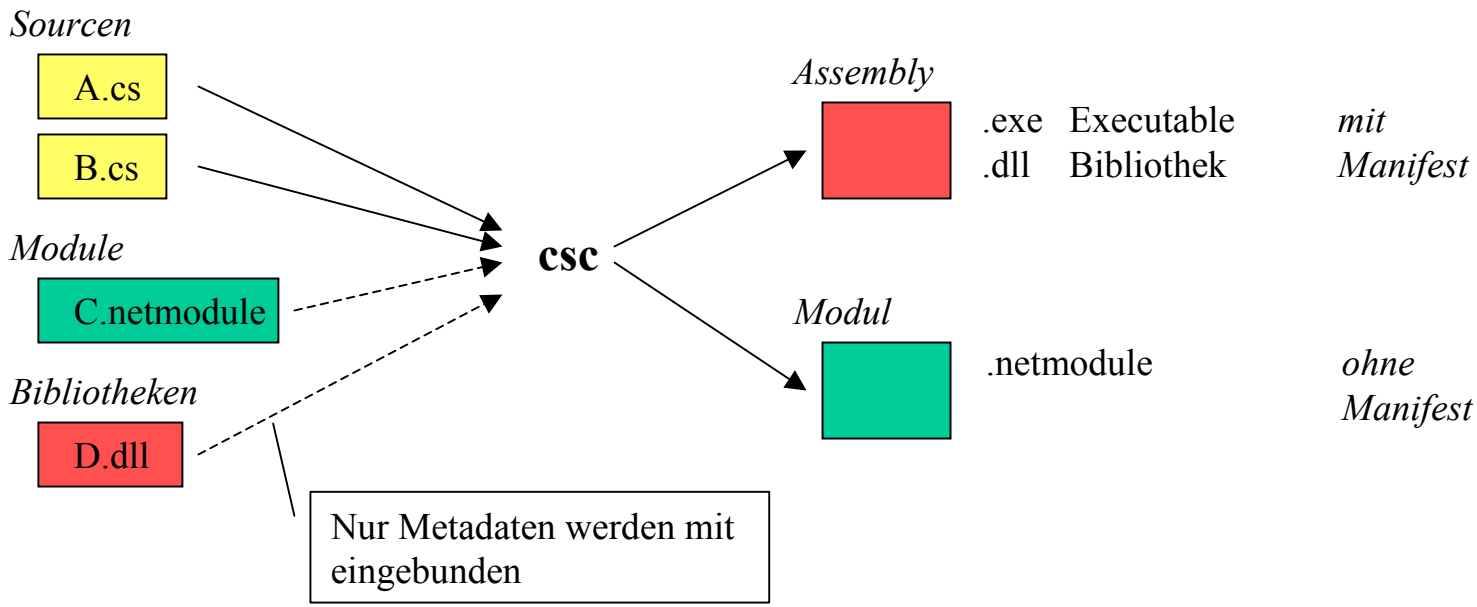
Assembly kann aus mehreren Dateien bestehen, wird aber durch "Manifest" (Inhaltsverzeichnis) zusammengehalten

Assembly ≈ JAR-File in Java
Assembly ≈ Komponente in .NET



Wie entstehen Assemblies?

Jede Compilation erzeugt ein Assembly oder ein Modul



Weitere Module/Ressourcen können mit Assembly-Linker (al) eingebunden werden

Unterschied zu Java: dort wird aus jeder Klasse ein .class-File erzeugt

Compiler-Optionen

Welche Ausgabedatei soll erzeugt werden?

/t[arget]: exe	Ausgabedatei = Executable einer Konsolenapplikation (Default)
| winexe	Ausgabedatei = Executable einer Win-GUI-Applikation
| library	Ausgabedatei = Bibliothek (DLL)
| module	Ausgabedatei = Modul (.netmodule)

/out:*name*	Name des Assembly oder des Moduls
	Default bei /t:exe *name*.exe, wobei *name* der Name der Quelldatei ist, die *Main*-Methode enthält
	Default bei /t:library *name*.dll, wobei *name* der Name der ersten Quelldatei ist
	Beispiel: csc /t:library /out:MyLib.dll A.cs B.cs C.cs

/doc:*name*	Erzeugt aus ///-Kommentaren eine XML-Datei namens *name*

Compiler-Optionen

Wie sollen Bibliotheken und Module eingebunden werden?

/r[eference]:*name*	Macht Metadaten in *name* (z.B. *xxx.dll*) in Compilation verfügbar *name* muß Metadaten enthalten.
/lib:dirpath{,dirpath}	Gibt Verzeichnisse an, in denen nach Bibliotheken gesucht wird, die mit /r referenziert werden.
/addmodule:name {,name}	Fügt angegebene Module (z.B. *xxx.netmodule*) zum erzeugten Assembly hinzu. Zur Laufzeit müssen alle diese Module im selben Verzeichnis stehen wie das Assembly, zu dem sie gehören.

Beispiel

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

Beispiele für Compilationen

csc A.cs	=> A.exe
csc A.cs B.cs C.cs	=> B.exe (wenn B.cs *Main* enthält)
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 (wobei A oder B Typen in X.dll referenzieren)
csc /addmodule:Y.netmodule A.cs	=> A.exe (Y wird zum Manifest dieses Assemblys hinzugefügt; Y.netmodule bleibt aber als eigenständige Datei erhalten)

Laden von Assemblies zur Laufzeit

Executable wird durch Programmaufruf geladen

(z.B. Aufruf von *MyApp* lädt *MyApp.exe* und führt es aus)

Bibliotheken (DLLs) werden in folgenden Verzeichnissen gesucht:

- im Application-Verzeichnis
- in allen Verzeichnissen, die in einer eventuell vorhandenen Konfigurationsdatei (z.B. *MyApp*.exe.config) unter dem <probing>-Tag angegeben sind

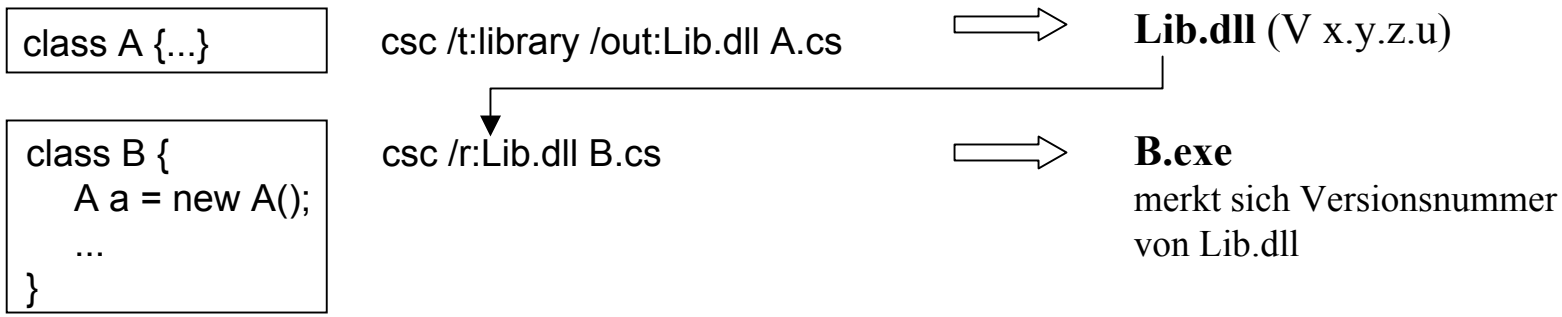
```
<configuration>
    ...
    <runtime>
        ...
        <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
            <probing privatePath="bin;bin2\subbin;bin3"/>
        </assemblyBinding>
    </runtime>
</configuration>
```

- im Global Assembly Cache (bei Shared Assemblies)

Versionierung von Assemblies

Es werden jene Bibliotheken geladen, die der erwarteten Versionsnr. entsprechen

Versionsnummer wird bei der Compilation gespeichert



Versionsnummer wird beim Laden geprüft

Aufruf: B
- lädt B.exe
- findet darin Bezug auf Lib.dll (V x.y.z.u)
- lädt Lib in Version V x.y.z.u
  (auch wenn es andere Versionen von Lib gibt)

Vermeidet "DLL Hell"

Bedeutung von "internal"

Sichtbarkeit *internal* bezieht sich darauf, was bei einer Übersetzung sichtbar ist

csc A.cs B.cs C.cs

Alle internal-Members von A, B, C sehen sich gegenseitig.

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

C sieht die internal-Members von A und B.
(A und B können in verschiedenen Sprachen geschrieben sein).