

# Die Sprache C# 5. Teil

- Generische Typen
- Constraints bei generischen Typen
- Vererbung, Null-Werte, etc.

## Generische Typen (ab C# 2.0)

- Der formale Parametertyp legt den Wertebereich fest, in dem die Argumente beim Aufruf liegen müssen

```
void foo (int k) ...  
foo(3); // 3 ∈ {int}
```

- Zur Laufzeit wird der Variablen ein konkreter Wert (von diesem Typ) zugewiesen
- Führt zu robusteren Programmen, weil
  - der Aufgerufene sich darauf verlassen kann, dass Werte in einem definierten Bereich liegen
  - der Aufrufer anhand der Schnittstelle erkennt (Compiler überprüft), welche Werte zulässig sind
- Aber es erschwert die Entwicklung von Bibliotheksfunktionen, die unabhängig vom Typ sein sollen
  - z.B. Sortierungs, Collections, ..

# Der Object-Trick

```
class Buffer {  
    private Object[] data;  
    public void Put(Object x) {...}  
    public Object Get() {...}  
}
```

## ■ Nutzt den Umstand, dass

- die Oberklasse aller Klassen vom Typ **Object** ist
- die Zuweisungskompatibilität entlang der Vererbungshierarchie gegeben ist

## ■ Nachteile

- Typumwandlungen nötig

```
buffer.Put(3); // Boxing kostet Zeit
```

```
int x = (int)buffer.Get(); // Typumwandlung kostet Zeit
```

- Homogenität kann nicht erzwungen werden

```
buffer.Put(3); buffer.Put(new Rectangle());
```

```
Rectangle r = (Rectangle)buffer.Get(); // kann zu Laufzeitfehler führen!
```

- Klasse pro Typ IntBuffer, RectangleBuffer, ... führt zu Code Duplikaten

# Variable Typen - Platzhalter Typ

- Nicht nur der Wert sondern auch der Typ kann variabel sein.
- Ermöglichung von Programmen, bei denen der konkrete Typ offen gelassen wird, bzw. erst später festgelegt wird.

```
void foo(Element e)
```

- Meist gemeinsamer Parametertyp pro Klasse

- Dieser Platzhaltertyp wird bei der Definition der Klasse folgendermaassen angegeben

```
class Bar<Element> {  
    foo(Element e) ...  
}
```

- Der konkrete Typ wird erst bei der Deklaration der Klasse festgelegt

```
Bar<int> b = new Bar<int>();  
b.foo(3);
```

# Generische Klasse Buffer

generischer Typ

Platzhaltertyp

```
class Buffer<Element> {
    private Element[] data;
    public Buffer(int size) {...}
    public void Put(Element x) {...}
    public Element Get() {...}
}
```

- geht auch für Structs und Interfaces
- Platzhaltertyp *Element* kann wie normaler Typ verwendet werden

## Benutzung

```
Buffer<int> a = new Buffer<int>(100);
a.Put(3);           // nur int-Parameter erlaubt; kein Boxing
int i = a.Get();    // keine Typumwandlung nötig
```

```
Buffer<Rectangle> b = new Buffer<Rectangle>(100);
b.Put(new Rectangle()); // nur Rectangle-Parameter erlaubt
Rectangle r = b.Get();  // keine Typumwandlung nötig
```

## Vorteile

- Homogene Datenstruktur mit Compilezeit-Typprüfung
- Effizienz (kein Boxing, keine Typumwandlungen)

Generizität auch in Ada, Eiffel, C++ (Templates), Java 1.5

## Buffer mit Prioritäten

```
class Buffer <Element, Priority> {  
    private Element[] data;  
    private Priority[] prio;  
    public void Put(Element x, Priority prio) {...}  
    public void Get(out Element x, out Priority prio) {...}  
}
```

## Verwendung

```
Buffer<int, int> a = new Buffer<int, int>();  
a.Put(100, 0);  
int elem, prio;  
a.Get(out elem, out prio);
```

```
Buffer<Rectangle, double> b = new  
Buffer<Rectangle, double>();  
b.Put(new Rectangle(), 0.5);  
Rectangle r; double prio;  
b.Get(out r, out prio);
```

## Annahmen über Platzhaltertypen werden als Basistypen ausgedrückt

```
class OrderedBuffer <Element, Priority> where Priority: IComparable {  
    Element[] data;  
    Priority[] prio;  
    int lastElem;  
    ...  
    public void Put(Element x, Priority p) {  
        int i = lastElem;  
        while (i >= 0 && p.CompareTo(prio[i]) > 0) {data[i+1] = data[i];  
prio[i+1] = prio[i]; i--;}  
        data[i+1] = x; prio[i+1] = p;  
    }  
}
```

Interface oder Basisklasse

Erlaubt Operationen auf Elemente von Platzhaltertypen

## Verwendung

Platzhaltertyp muss IComparable unterstützen

```
OrderedBuffer<int, int> a = new OrderedBuffer<int, int>();  
a.Put(100, 3);
```



# Mehrere Constraints möglich

```
class OrderedBuffer <Element, Priority>
  where Element: MyClass
  where Priority: IComparable
  where Priority: ISerializable {
    ...
    public void Put(Element x, Priority p) {...}
    public void Get(out Element x, out Priority p) {...}
  }
```

## Verwendung

muss Unterklasse von MyClass sein

muss IComparable und ISerializable unterstützen

```
OrderedBuffer<MySubclass, MyPrio> a = new OrderedBuffer<MySubclass, MyPrio>();
...
a.Put(new MySubclass(), new MyPrio(100));
```

## Zum Erzeugen neuer Objekte in einem generischen Typ

```
class Stack<T, E> where E: Exception, new() {  
    T[] data = ...;  
    int top = -1;  
    public void Push(T x) {  
        if (top >= data.Length)  
            throw new E();  
        else  
            data[++top] = x;  
    }  
}
```

spezifiziert, dass der Platzhalter E einen parameterlosen Konstruktor haben muss.

## Verwendung

```
class MyException: Exception {  
    public MyException(): base("stack overflow or underflow") {}  
}
```

```
Stack<int, MyException> stack = new Stack<int, MyException>();  
...  
stack.Push(3);
```

```
class Buffer <Element>: List<Element> {  
    ...  
    public void Put(Element x) {  
        this.Add(x); // Add wurde von List geerbt  
    }  
}
```

kann auch generische Interfaces implementieren

## Von welchen Klassen darf eine generische Klasse erben?

■ von einer gewöhnlichen Klasse `class T<X>: B {...}`

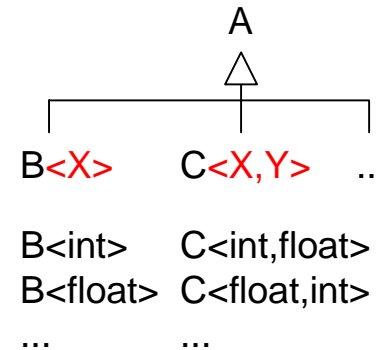
■ von einer konkretisierten generischen Klasse `class T<X>: B<int> {...}`

■ von einer generischen Klasse mit gleichem Platzhalter `class T<X>: B<X> {...}`

## Zuweisung von T<x> an gewöhnliche Oberklasse

```
class A {...}  
class B<X>: A {...}  
class C<X,Y>: A {...}
```

```
A a1 = new B<int>();  
A a2 = new C<int, float>();
```

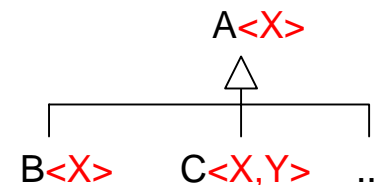


## Zuweisung von T<x> an generische Oberklasse

```
class A<X> {...}  
class B<X>: A<X> {...}  
class C<X,Y>: A<X> {...}
```

```
A<int> a1 = new B<int>();  
A<int> a2 = new C<int, float>();
```

```
A<int> a3 = new B<short>();
```



erlaubt, wenn korrespondierende Platzhalter durch denselben Typ ersetzt wurden

jedoch verboten !!!

# Überschreiben von Methoden

```
class Buffer<Element> {  
    ...  
    public virtual void Put(Element x) {...}  
}
```

## Wenn von konkretisierter Klasse geerbt

```
class MyBuffer: Buffer<int> {  
    ...  
    public override void Put(int x) {...}  
}
```

Put<Element> von Buffer<Element> geerbt

## Wenn von generischer Klasse geerbt

```
class MyBuffer<Element>: Buffer<Element> {  
    ...  
    public override void Put(Element x) {...}  
}
```

## Folgendes geht nicht (man kann keinen Platzhaltertyp erben)

```
class MyBuffer: Buffer<Element> {  
    ...  
    public override void Put(Element x) {...}  
}
```

- Konkretisierter generischer Typ kann wie normaler Typ verwendet werden

```
Buffer<int> buf = new Buffer<int>(20);  
object obj = buf;  
  
if (obj is Buffer<int>)  
    buf = (Buffer<int>) obj;  
  
Type t = typeof(Buffer<int>);  
Console.WriteLine(t.Name); // => Buffer[System.Int32]
```

- Reflection liefert auch die konkreten Platzhaltertypen!

Unterschied zu Java: zur Laufzeit keine Typeninformation mehr vorhanden  
-> Generics in Java werden auch als "Erasures" bezeichnet

## Methoden, die mit verschiedenen Datentypen arbeiten können

```
static void Sort<T> (T[] a) where T: IComparable {  
    for (int i = 0; i < a.Length-1; i++) {  
        for (int j = i+1; j < a.Length; j++) {  
            if (a[j].CompareTo(a[i]) < 0) {  
                T x = a[i]; a[i] = a[j]; a[j] = x;  
            }  
        }  
    }  
}
```

kann beliebige Arrays  
sortieren, solange die  
Elemente IComparable  
implementieren

### Benutzung

```
int[] a = {3, 7, 2, 5, 3};  
...  
Sort<int>(a);    // a == {2, 3, 3, 5, 7}
```

```
string[] s = {"one", "two", "three"};  
...  
Sort<string>(s);    // s == {"one", "three", "two"}
```

Meist weiss der Compiler aus den Parametern welchen Typ er  
für den Platzhalter einsetzen muss, so dass man einfach schreiben kann:

```
Sort(a);    // a == {2, 3, 3, 5, 7}
```

```
Sort(s);    // s == {"one", "three", "two"}
```

# Generische Delegates

```
delegate bool Check<T>(T value);  
  
class Payment {  
    public DateTime date;  
    public int amount;  
}  
  
class Account {  
    ArrayList payments = new ArrayList();  
    public void Add(Payment p) { payments.Add(p); }  
    public int AmountPaid(Check<Payment> matches) {  
        int val = 0;  
        foreach (Payment p in payments)  
            if (matches(p)) val += p.amount;  
        return val;  
    }  
}
```

Es wird eine Prüfmethode  
übergeben, die für jedes  
Payment prüft, ob es in Frage  
kommt

```
bool PaymentsAfter(Payment p) {  
    return DateTime.Compare(p.date, myDate) >= 0;  
}  
...  
myDate = new DateTime(2003, 11, 1);  
int val = account.AmountPaid(new Check<Payment>(PaymentsAfter));
```

```
int val = account.AmountPaid(delegate(Payment p) {  
    return DateTime.Compare(p.date, new DateTime(2003,11,1)) >= 0;  
});
```

als anonyme  
Methode



## Nullsetzen eines Werts

```
void Foo<T>() {  
    T x = null;    // Fehler  
    T y = 0;       // Fehler  
    T z = T.default; // ok! 0, '\0', false, null  
}
```

## Abfragen auf null

```
void Foo<T>(T x) {  
    if (x == null) {  
        Console.WriteLine(x + " == null");  
    } else {  
        Console.WriteLine(x + " != null");  
    }  
}
```

für Referenztypen führt `x == null` Vergleich durch  
für Werttypen liefert `x == null` den Wert *false*

```
Foo(3);           // 3 != null  
Foo(0);           // 0 != null  
Foo("Hello");     // Hello != null  
Foo<string>(null); // == null
```

# Was geschieht hinter den Kulissen?

```
class Buffer<Element>
{...}
```

Compiler erzeugt IL-Code für Klasse Buffer mit Platzhalter für Element.

## Konkretisierung mit Werttypen

```
Buffer<int> a = new Buffer<int>();
```

```
Buffer<int> b = new Buffer<int>();
```

```
Buffer<float> c = new Buffer<float>();
```

CLR erzeugt zur Laufzeit **neue Klasse** Buffer<int>, in der Element durch int ersetzt wird.

Verwendet vorhandenes Buffer<int>.

CLR erzeugt zur Laufzeit **neue Klasse** Buffer<float>, wird.

## Konkretisierung mit Referenztypen

```
Buffer<string> a = new Buffer<string>();
```

```
Buffer<string> b = new Buffer<string>();
```

```
Buffer<Node> b = new Buffer<Node>();
```

CLR erzeugt zur Laufzeit **neue Klasse** Buffer<object>, die mit allen Referenztypen arbeiten kann.

Verwendet vorhandenes Buffer<object>.

Verwendet vorhandenes Buffer<object>.

# Unterschiede zu anderen Sprachen

## C++ ähnliche Syntax

```
template <class Element>
class Buffer {
    ...
    void Put(Element x);
}
Buffer<int> b1;
Buffer<int> b2;
```

- **Compiler** erzeugt für **jede** Konkretisierung eine neue Klasse
- keine Constraints, weniger typsicher

## Java

- ab Version 1.5

- Platzhalter können nur durch Referenztypen ersetzt werden
- durch versteckte Type-Casts implementiert (kostet Laufzeit)
- Reflection liefert keine exakte Typinformation

# Die .NET-Architektur

- Begriffe, Plattformen, Features, allg. Betrachtungen, ...
- Common Type System
- Intermediate Language
- Common Language Subsystem

## ■ Managed Code

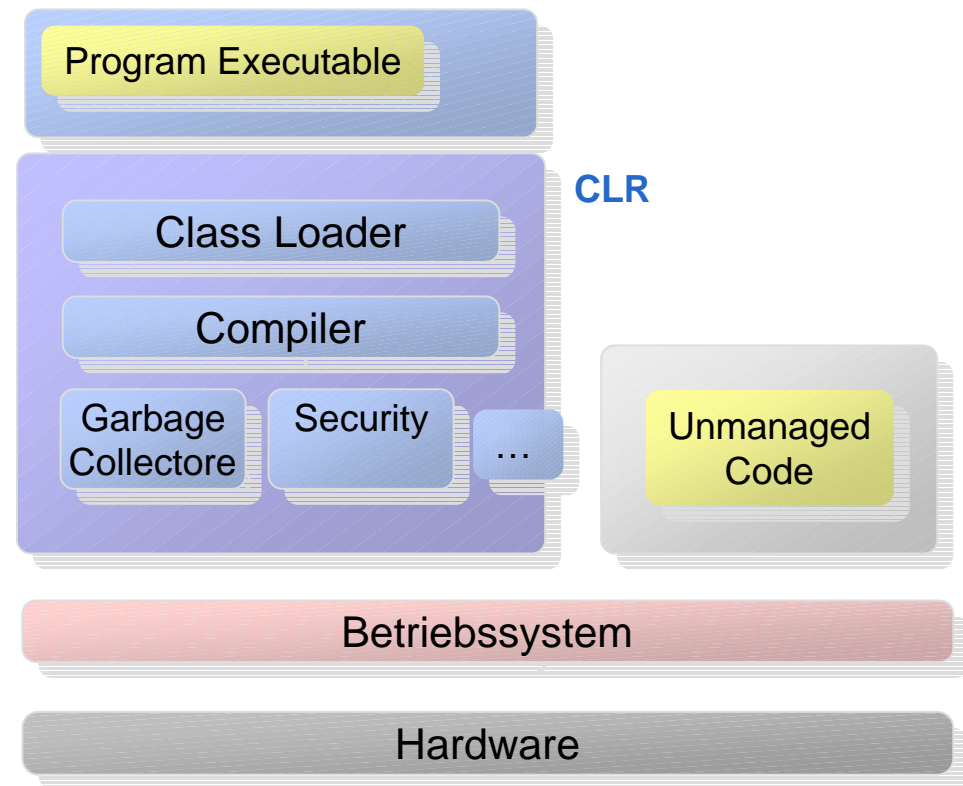
- Wird von virtueller Maschine ausgeführt
- Basiert auf dem .NET Framework

## ■ NET Framework (CLR)

- Laufzeitumgebung
- Tools
- Klassenbibliothek

## ■ Unmanaged Code

- Win32-Code, I32 o.ä.
- Vista!
- W7



# Common Language Runtime (CLR)

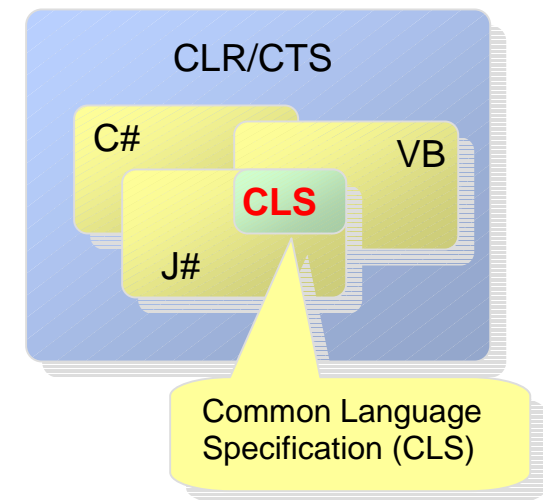
## ■ Tools

- Laden und Verwalten von Assemblies
- Compiler
- Diverse Tools
- Verwaltung von geteilten Assemblies

## ■ Standard-Assemblies

## ■ Sprachübergreifendes Typsystem (**CTS**)

- Vorhandene Typen in allen .NET-Sprachen nutzbar
- Interoperabilität zwischen Sprachen gewährleistet



# Eine gemeinsame Klassenbibliothek

## System.Web

### Services

Description

Discovery

Protocols

Caching

Configuration

### UI

HtmlControls

WebControls

Security

SessionState

## System.WinForms

Design

ComponentModel

## System.Drawing

Drawing2D

Printing

Imaging

Text

## System.Data

ADO

Design

SQL

SQLTypes

## System.Xml

XSLT

XPath

Serialization

## System

Collections

Configuration

Diagnostics

Globalization

IO

Net

Reflection

Resources

Security

ServiceProcess

Text

Threading

Runtime

InteropServices

Remoting

Serialization

# Common Type System (CTS) und CLS

- CTS-Gemeinsames Typ-System

- wird in Sprachtypen übersetzt

- C# alle in System deklarierte Typen

- `System.Int32` -> `int`
- `System.Double` -> `double`

CTS Type

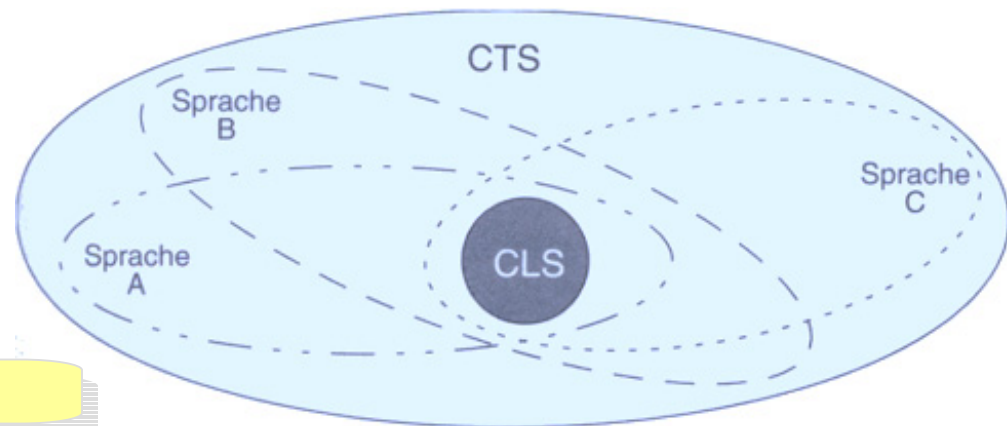
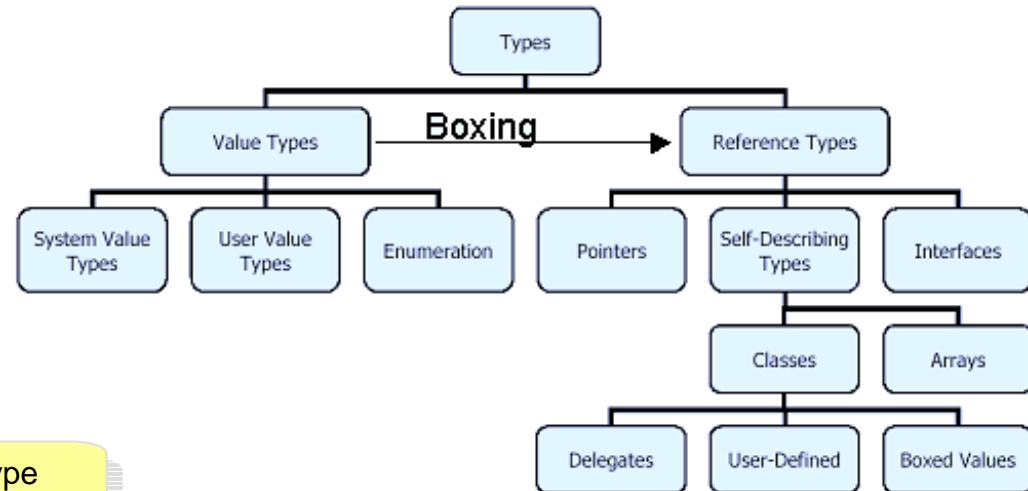
C# Type

- i.A. von Sprache nur Subset der Typen unterstützt

- Common Language Specification (**CLS**)  
= gemeinsamer Satz von Typen + Regeln

AssemblyInfo.cs

- kann mit `[assembly: CLSCompliant(true)]` erzwungen werden





- Compiler erzeugen keinen nativen Code sondern eine prozessorunabhängige Zwischensprache

## IL - Intermediate Language

- IL unterscheidet sich von „reinen“ Assemblersprachen
  - komplexe Datentypen und Objekte sind fester Bestandteil
  - Konzepte wie Vererbung und Polymorphie werden von vornherein unterstützt
- IL Code wird unter Kontrolle der Common Language Runtime ausgeführt
- CLR
  - führt Sicherheitsüberprüfungen aus
  - übernimmt Speicherverwaltung und Fehlerbehandlung (→ GC, Exceptions)
  - führt Versionsprüfungen aus
  - ...
- Dieser Code wird deshalb als "Managed Code" bezeichnet

# CLI - Common Language Infrastructure

- CLI = Grundgerüst der .NET-Architektur (Subset der CLR)
- wird im ECMA-Standard 335 beschrieben (2<sup>nd</sup> ed., Dec.2002)  
<http://www.ecma-international.org/publications/standards/Ecma-335.htm>

## ■ Microsofts .NET-Framework Implementierung

- enthält Common Language Runtime (CLR subset)  
= Microsofts Implementierung des CLI-Standards
- erfüllt den Standard vollständig
- bietet aber noch einiges darüber hinaus, z.B.
  - *ASP.NET*
  - *ADO.NET*
  - *Web Services*
  - *erweiterte Klassenbibliothek*
  - ...



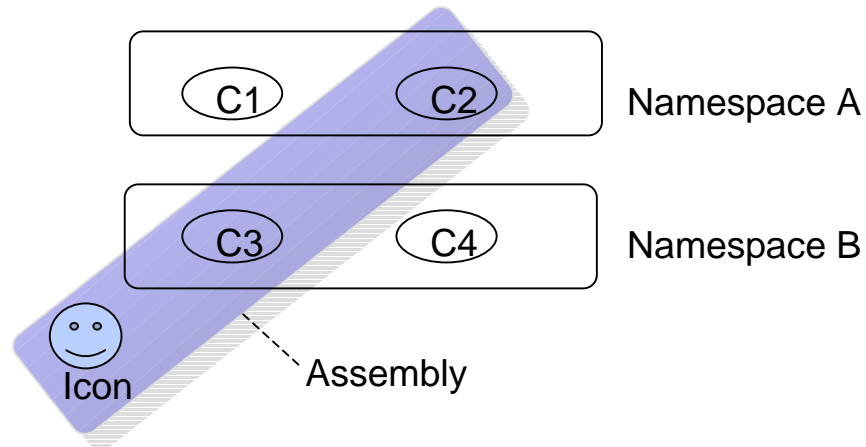
Z.Z. einzige weitere  
Implementation ist das  
mono Projekt auf Linux,  
Mac, Windows

# Definiert durch die CLI

- einheitliche Zwischensprache ⇒ **IL**
- einheitliches Typsystem ⇒ **CTS**
- Einheiten für Auslieferung ⇒ **Assembly**
- erweiterbare Typinformation ⇒ **Metadaten**
- Integration vieler Programmiersprachen ⇒ **CLS**
- Code-basiertes Sicherheitskonzept ⇒ **CAS**
- automatische Speicherbereinigung ⇒ **GC**
- Just-in-Time-Übersetzung (*NIE* Interpretation!) ⇒ **JIT**
- .NET Framework (aber ohne GUI und DB Klassen)

# Assemblies und Module

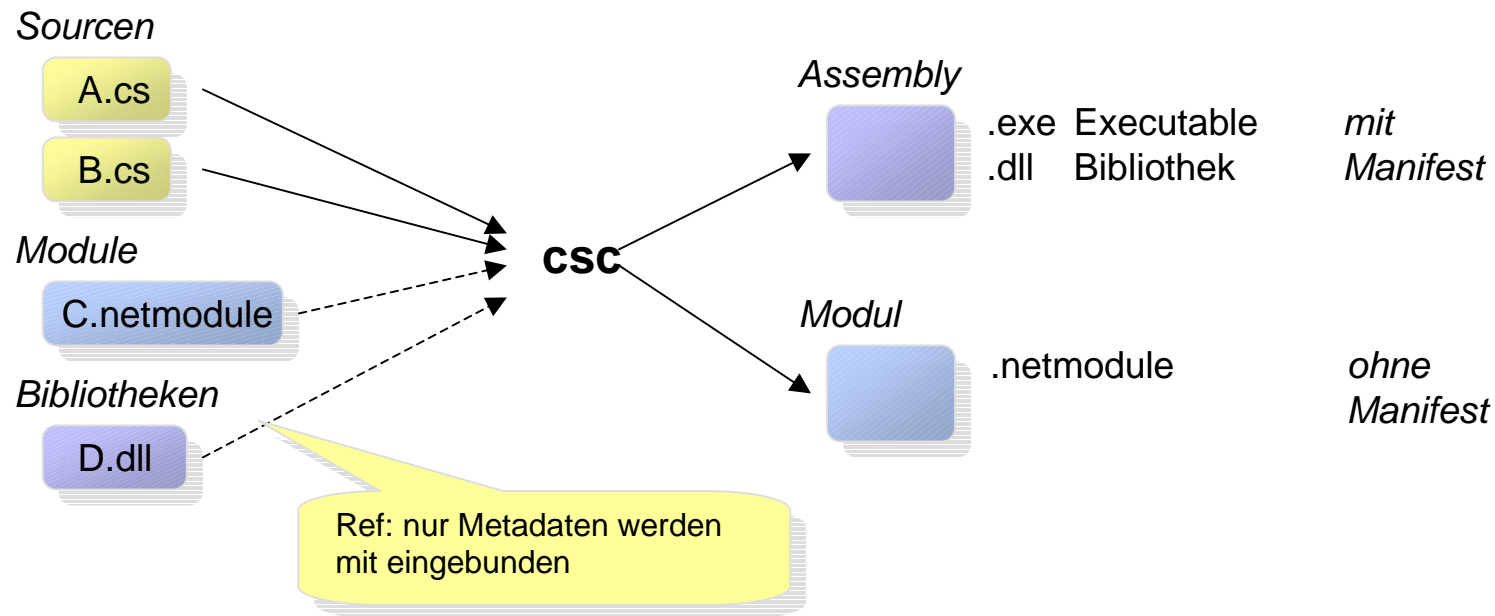
- Laufzeiteinheit aus mehreren Typen und sonstigen Ressourcen (z.B. Icons)



- **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

# Wie entstehen Assemblies?

Jede Compilation erzeugt ein Assembly oder ein Modul



- Weitere Module/Ressourcen können mit Assembly-Linker eingebunden werden
- Unterschied zu Java: aus jeder Klasse ein .class-File erzeugt

## ■ Modul (*managed module*) =

- **physische Einheit**: 1 .NET-Modul = 1 .NET-PE-Datei
- enthält Typdefinitionen mit Metadaten & IL-Code der Methoden
- wird vom Compiler erzeugt

## ■ Assembly =

- **logische Einheit** für:  
Auslieferung, Kapselung, Versionierung, Sicherheit
- .NET-Komponente (im Sinne der komponenten-orientierten Softwareentwicklung)
- fasst Module und Ressourcdateien zusammen (siehe ⇒ **Manifest**)
- wird vom Compiler oder Assembly Linker (al.exe) erzeugt

## ■ Manifest

- speichert Informationen über die Teile eines Assemblies  
z.B. welche Dateien gehören dazu, wo sind diese zu finden, exportierte Typen, ...

# Modul = Übersetzungseinheit

## Bestandteile

### ■ PE Header

- (Portable Executable)
- Dateityp (GUI, CUI, DLL, ...)
- Zeitstempel
- Primär für unmanaged Code

### ■ CLR Header

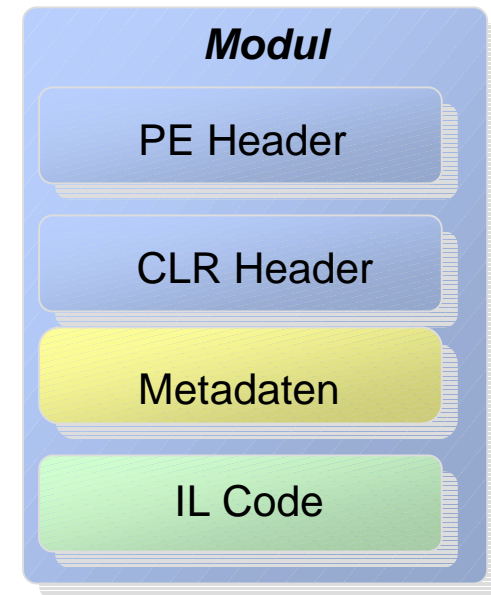
- Benötigte CLR Version
- Ort der Metadaten etc.
- Einstiegspunkt

...

### ■ Metadaten

- Definierte Typen und Member
- Referenzierte Typen und Member

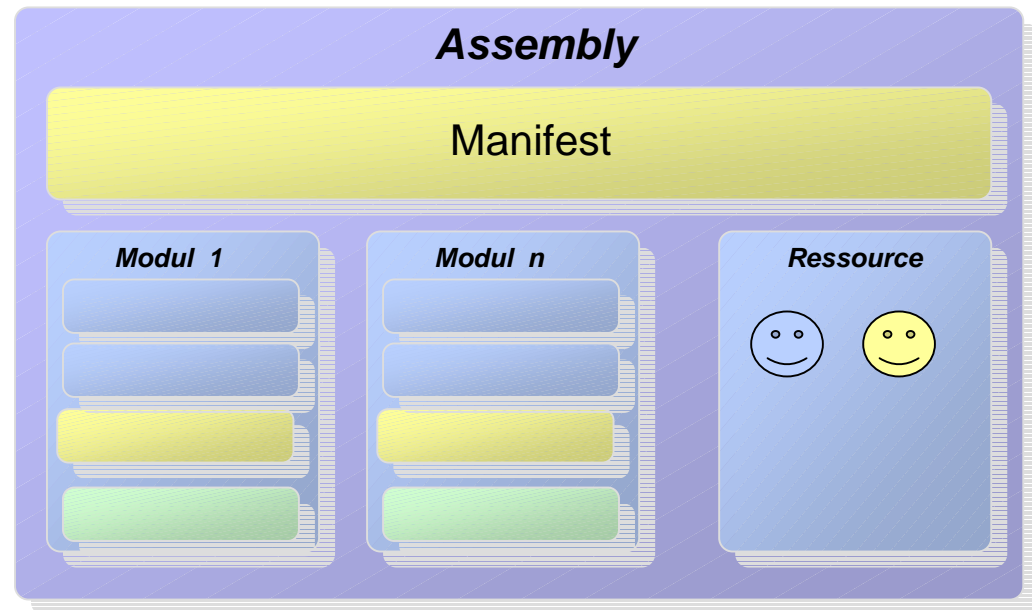
### ■ IL: Der übersetzte Code





# Assembly= Verteilungs-/Versionierungseinheit

- Zusammenfassung von
  - 1 oder mehreren Modulen
  - Weitere Ressourcen (Graphiken, HTML-Dateien, etc.)
- Struktur wird durch ein Manifest beschrieben
  - Name
  - Shared Name
  - Version
  - Hash
  - Referenzierte Assemblies
  - ...
- Assembly Linker (AL.exe)



# Assembly Kategorien

## ■ Private Assembly

- Assembly kann nur von genau einer Anwendung benutzt werden
- im gleichen Verzeichnis oder Unterverzeichnis

## ■ Shared Assembly

- Assembly kann global von allen Anwendungen benutzt werden
- im Global Assembly Cache

- Identifikation anhand eines einfachen Names
- Keine Versionsüberprüfung (nur auf Dateinamen)
- Alle zusammengehörigen Dateien in einem Verzeichnis
  - Anwendung kann mittels Copy verteilt werden
  - Standardmässig befinden sich Hilfsassemblies (dll) und Anwendung (exe) im gleichen Verzeichnis
  - Weitere Unterverzeichnisse können per .config-Datei definiert werden
  - Suchheuristik (auch "Culture"-abhängig)
    - -> *Culture der Assembly de-CH Verzeichnis de-CH wird (zuerst) nach DLL gesucht.*

# 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:

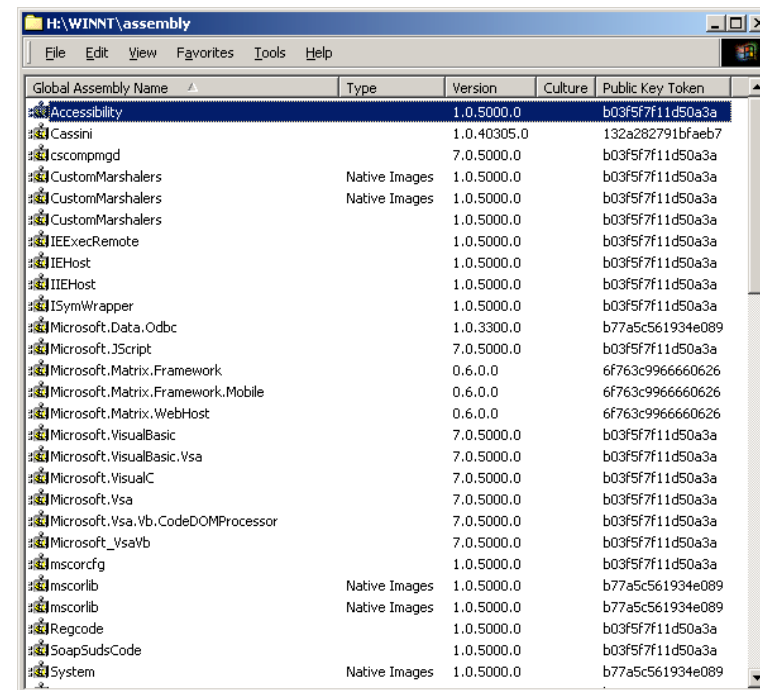
- im Application-Verzeichnis
- in allen Verzeichnissen, die in einer eventuell vorhandenen Konfigurationsdatei (z.B. *MyApp.exe.config*) unter dem <probing>-Tag angegeben sind

```
<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)

# Shared Assemblies

- Identifikation über einen **Strong Name**
- Versionsüberprüfung durch die Runtime
- Installation im Global Assembly Cache  
(→ SDK-Tool gacutil.exe)
  - systemweiter “Speicherbereich”
  - normale Dateien
  - keine Registry-Einträge, o. ä.
- In Explorer direkt darstellbar
  - <WINHOME>\assembly



| Global Assembly Name              | Type          | Version     | Culture | Public Key Token |
|-----------------------------------|---------------|-------------|---------|------------------|
| Accessibility                     |               | 1.0.5000.0  |         | b03f5f7f11d50a3a |
| Cassini                           |               | 1.0.40305.0 |         | 132a282791bfaeb7 |
| cscompmgd                         |               | 7.0.5000.0  |         | b03f5f7f11d50a3a |
| CustomMarshalers                  | Native Images | 1.0.5000.0  |         | b03f5f7f11d50a3a |
| CustomMarshalers                  | Native Images | 1.0.5000.0  |         | b03f5f7f11d50a3a |
| CustomMarshalers                  |               | 1.0.5000.0  |         | b03f5f7f11d50a3a |
| IEExecRemote                      |               | 1.0.5000.0  |         | b03f5f7f11d50a3a |
| IEHost                            |               | 1.0.5000.0  |         | b03f5f7f11d50a3a |
| IEHost                            |               | 1.0.5000.0  |         | b03f5f7f11d50a3a |
| ISymWrapper                       |               | 1.0.5000.0  |         | b03f5f7f11d50a3a |
| Microsoft.Data.Odbc               |               | 1.0.3300.0  |         | b77a5c561934e089 |
| Microsoft.JScript                 |               | 7.0.5000.0  |         | b03f5f7f11d50a3a |
| Microsoft.Matrix.Framework        |               | 0.6.0.0     |         | 6f763c9966660626 |
| Microsoft.Matrix.Framework.Mobile |               | 0.6.0.0     |         | 6f763c9966660626 |
| Microsoft.Matrix.WebHost          |               | 0.6.0.0     |         | 6f763c9966660626 |
| Microsoft.VisualBasic             |               | 7.0.5000.0  |         | b03f5f7f11d50a3a |
| Microsoft.VisualBasic.Vsa         |               | 7.0.5000.0  |         | b03f5f7f11d50a3a |
| Microsoft.VisualBasic             |               | 7.0.5000.0  |         | b03f5f7f11d50a3a |
| Microsoft.Vsa                     |               | 7.0.5000.0  |         | b03f5f7f11d50a3a |
| Microsoft.Vsa.Vb.CodeDOMProcessor |               | 7.0.5000.0  |         | b03f5f7f11d50a3a |
| Microsoft.VsaVb                   |               | 7.0.5000.0  |         | b03f5f7f11d50a3a |
| mscorlib                          |               | 1.0.5000.0  |         | b03f5f7f11d50a3a |
| mscorlib                          | Native Images | 1.0.5000.0  |         | b77a5c561934e089 |
| mscorlib                          | Native Images | 1.0.5000.0  |         | b77a5c561934e089 |
| RegCode                           |               | 1.0.5000.0  |         | b03f5f7f11d50a3a |
| SoapSudsCode                      |               | 1.0.5000.0  |         | b03f5f7f11d50a3a |
| System                            | Native Images | 1.0.5000.0  |         | b77a5c561934e089 |

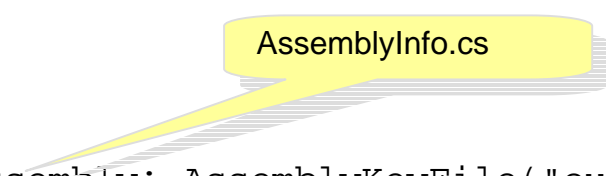
# Shared Assemblies - Strong Name

- Eindeutigkeit des Namens wird mit Hilfe der Public-Key-Verschlüsselung garantiert

- Strong Name = Identität + Public Key
- Attribut "**Originator**" im Manifest

- Vorgehen:

- Keyfile erstellen (→ `sn.exe -k outf.snk`)
- Im Sourcecode des Shared Assemblies Attribut `[assembly: AssemblyKeyFile("outf.snk")]` angeben
- Beim Kompilieren des Shared Assemblies wird der Public Key im Manifest eingetragen
- Client, der das Assembly referenziert, erhält beim Kompilieren einen Hashwert d. Public Key (→ `publickeytoken` in seinem Manifest)
- Eintragen in Global-Assembly Cache:  
`gacutil /i <assembly-name>`



AssemblyInfo.cs

- Client wird standardmässig an die Version des Shared Assemblies gebunden, die in seinem Manifest eingetragen ist
- Dieses Verhalten kann per `.config`-Datei übersteuert werden (→ später)

# Versionierung

# Versionierung - "Side by Side" Execution

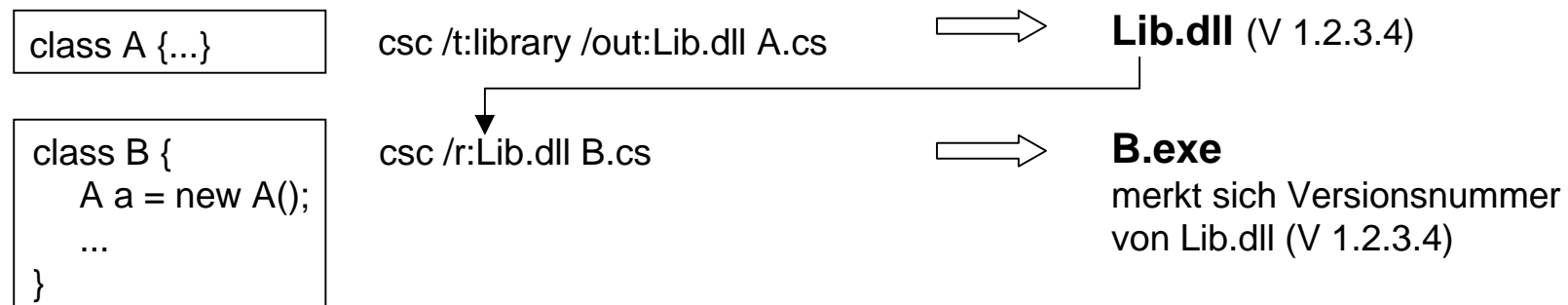
- Die Möglichkeit mehrere Versionen derselben Assembly im Speicher zu halten
- Programme die unterschiedliche Versionen brauchen müssen nicht unbedingt strikt aufwärtskompatibel sein
  - Können auf derselben Maschine laufen
  - Können sogar in demselben Process laufen
- Versionen von Assemblies werden in der CLR durch Versionsnummer und Public Key gekennzeichnet.



# Versionierung von Assemblies

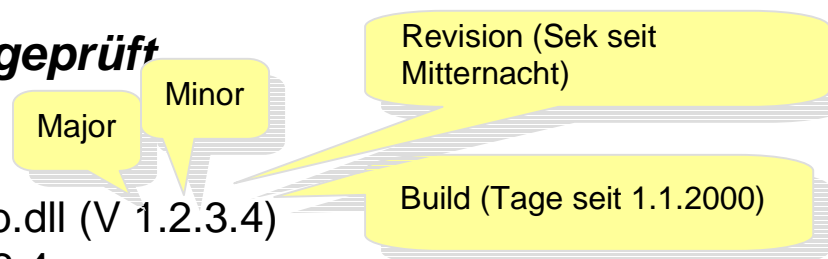
Es werden jene Bibliotheken geladen, die der erwarteten Versionsnr. entsprechen

## ***Versionsnummer wird bei der Compilation gespeichert***



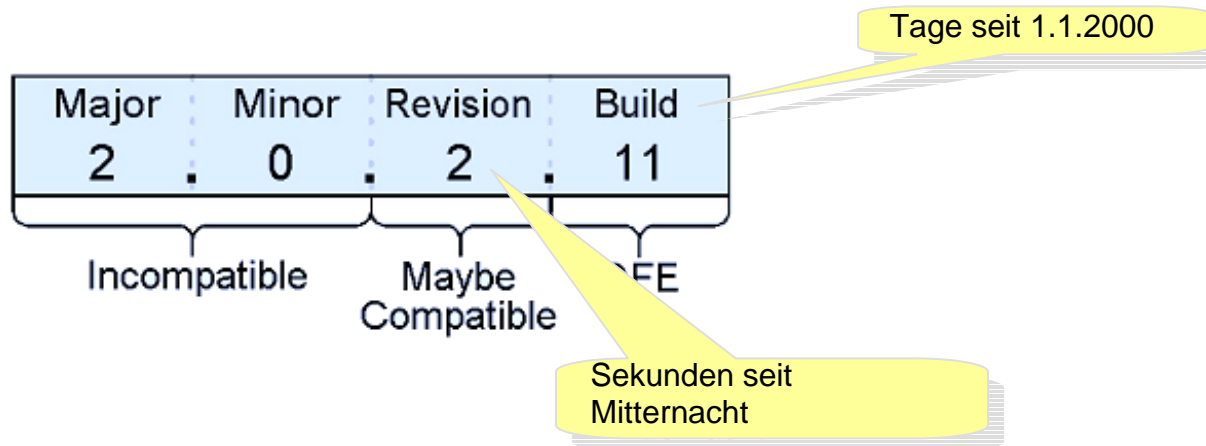
## ***Versionsnummer wird beim Laden geprüft***

- Aufruf: B
- lädt B.exe
  - findet darin Bezug auf Lib.dll (V 1.2.3.4)
  - lädt Lib in Version V 1.2.3.4  
(auch wenn es andere Versionen von Lib gibt)



Vermeidet "DLL (Versionen-) Hell" aber mehrere "DLL (Varianten) Hell"

# Versionierung



- Ein Shared Assembly ist grundsätzlich inkompatibel zum Client, wenn sich die Major- oder Minor-Version ändert
  - Beispiel: neues Produktrelease
  - Runtime wirft eine Type Load Exception
- Ein Shared Assembly kann kompatibel zum Client sein, wenn sich die Revision bei gleich bleibender Major- und Minor-Version ändert
  - Beispiel: Servicepack
  - Runtime versucht, das Assembly mit der höchsten Revisions- und Buildnummer zu laden
- Als Attribut angegeben `[assembly: AssemblyVersion("1.2.*")]`

AssemblyInfo.cs

- Ein Shared Assembly ist grundsätzlich kompatibel zum Client, wenn sich nur die Buildnummer ändert
  - In diesem Fall liegt ein sogenannter Quick Fix Engineering (QFE) vor
  - Beispiel: Security Hotfix
  
- Runtime versucht immer, die Assembly mit der höchsten Revisions- und Buildnummer zu laden

- Es soll eine ganz bestimmte Version eines Assembly geladen werden
- Angegeben müssen dafür:
  - Name: Name des Assemblies
  - Originator: Public Key des Assemblies, um Eindeutigkeit zu gewährleisten
  - oldVersion: Version, die nicht geladen werden sollen  
(→ ein Stern kennzeichnet alle Versionen)
  - newVersion: Version des Assemblies, das geladen werden soll
- Beispiel für die Option **bindingRedirect**
  - Neue Version 2.0.1.0 ist inkompatibel zu 1.0.0.0
  - ohne neu zu kompilieren können Clients dennoch die Version 2.0.0.0 benutzen
- Es kann auch angegeben werden, dass eine spezielle Version der CLR benötigt wird: **requiredRuntime, supportedRuntime**

# Versionierungs Beispiele

```
<?xml version = "1.0"?>
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="Reverser"
                           publicKeyToken="2c47419262627255" />
        <bindingRedirect oldVersion="1.0.0.0"
                           newVersion="2.0.0.0" />
      </dependentAssembly>
      <probing privatePath="Reverse;String" />
    </assemblyBinding>
  </runtime>
</configuration>
```

```
<configuration>
  <startup>
    <requiredRuntime version="v2.0.50727" />
    <supportedRuntime version="v1.1.4322" />
  </startup>
</configuration>
```

Es ist CLR 2.0 verlangt es  
läuft aber auch unter  
CLR 1.1

# CLR Versionen

■ Framework Versionen 1.1, 2.0, 3.0, 3.5, 4, 4.5

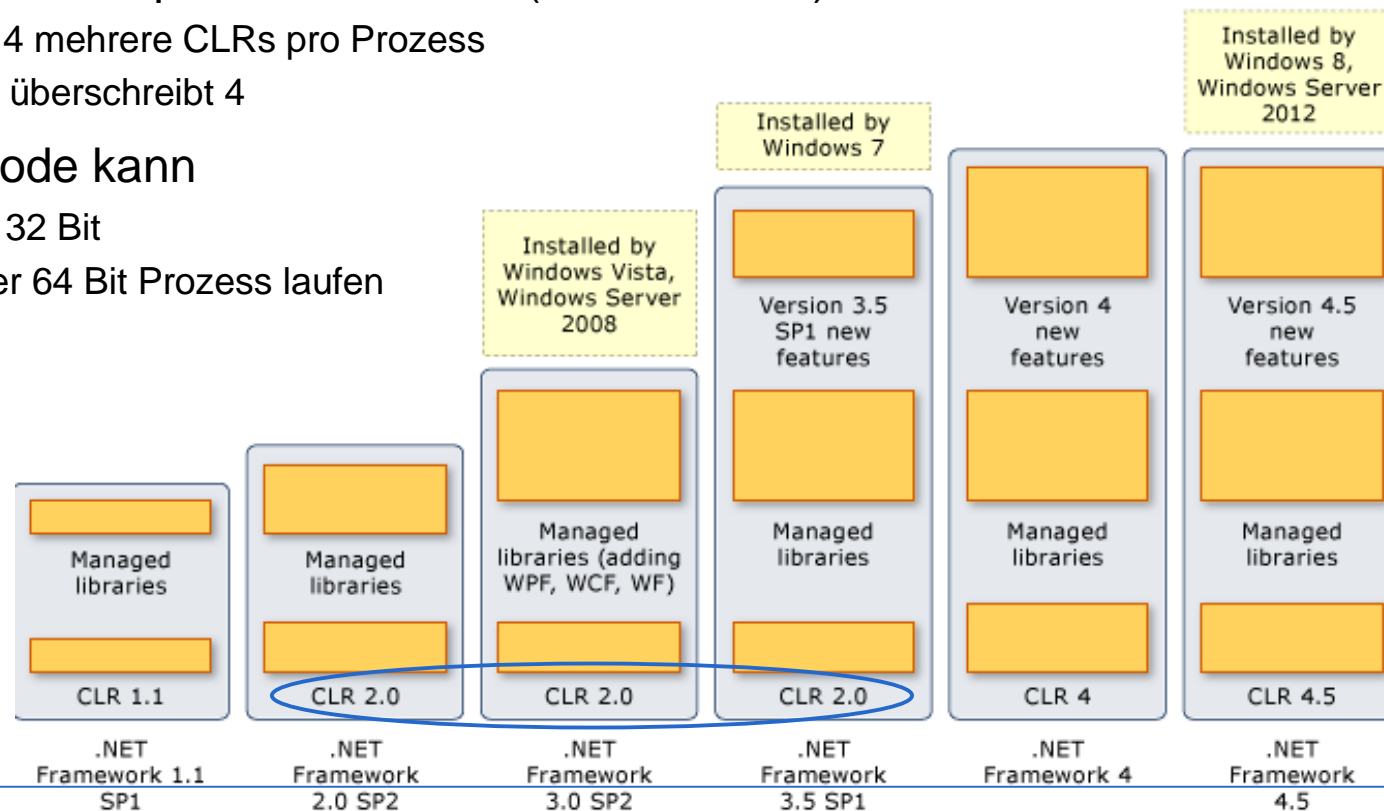
■ CLR Versionen 1.1, 2.0, 4, 4.45

■ CLR werden parallel installiert (bis und mit 4)

- Ab 4 mehrere CLR pro Prozess
- 4.5 überschreibt 4

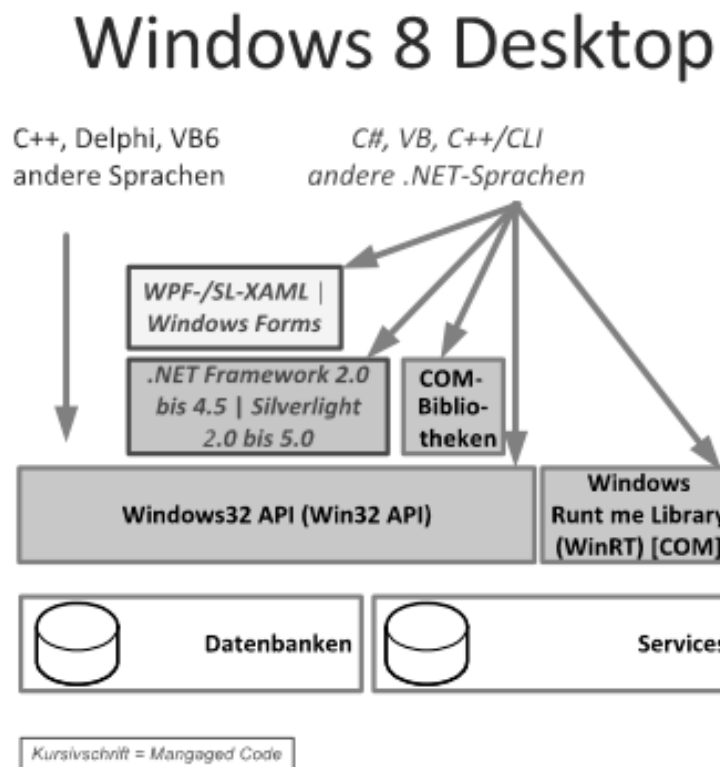
■ Der Code kann

- als 32 Bit
- oder 64 Bit Prozess laufen

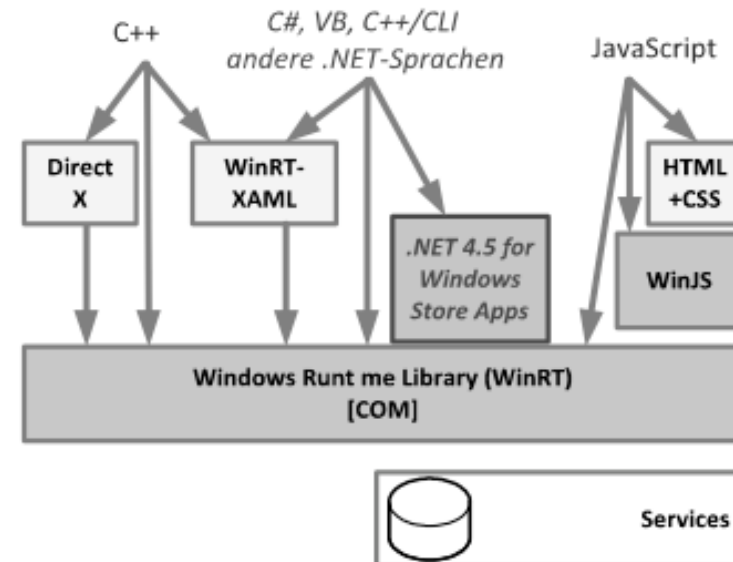


## ■ WinRT (Library nur auf Windows 8)

- Ähnlich wie .NET Standardklassenbibliothek
- mit z.T. unterschiedlichen Methoden Signaturen oder anderen Namespaces
- Nur für Metro Apps

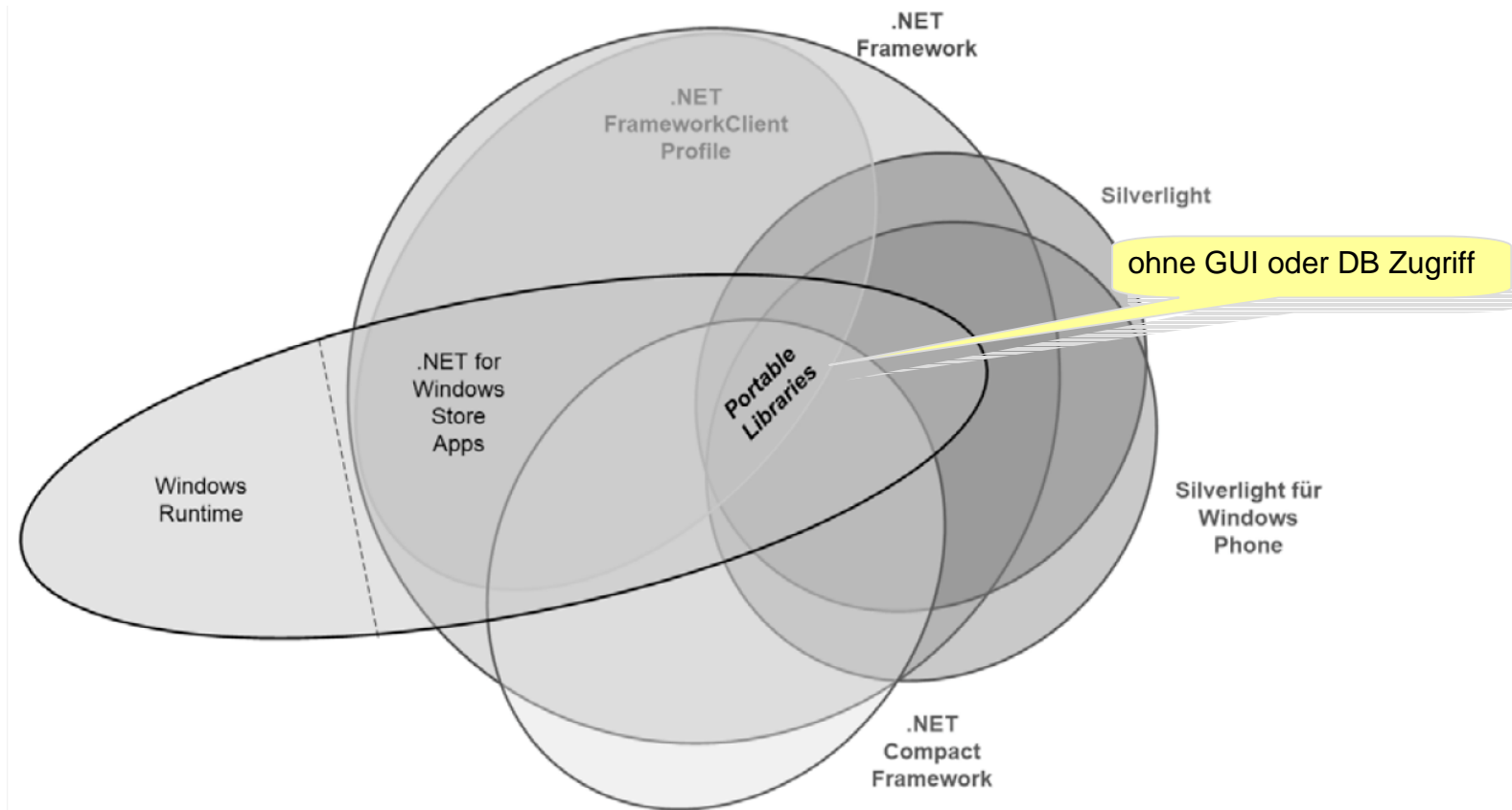


### Windows 8 Metro Apps



© Dr. Holger Schwichtenberg, www.IT-Visions.de, 2011-2012

- Je nach Umgebung steht ein anderes Subset an Bibliotheken zur Verfügung

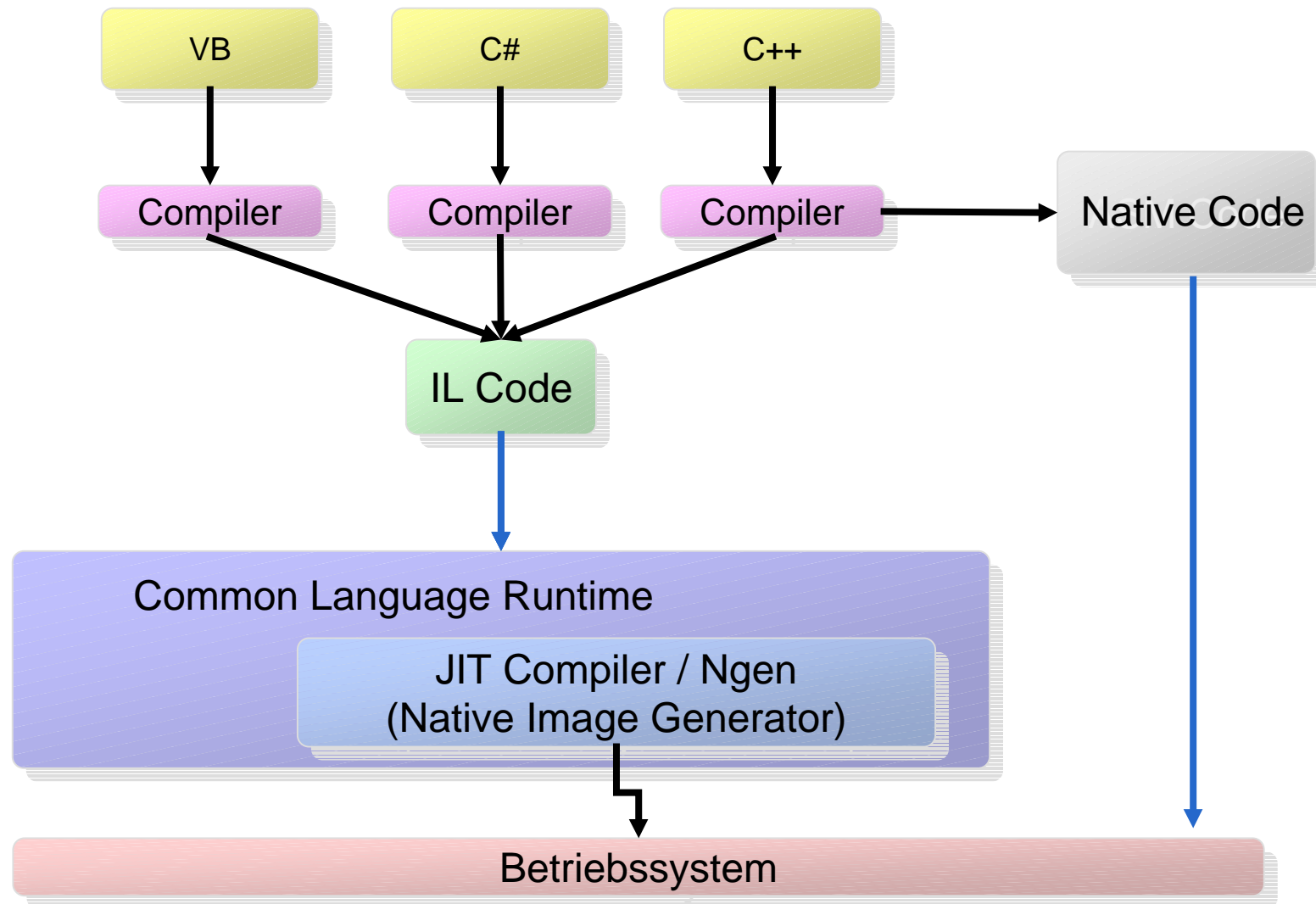


© Dr. Holger Schwichtenberg, [www.IT-Visions.de](http://www.IT-Visions.de), 2011-2012



# Übersetzen und Ausführen von Programmen

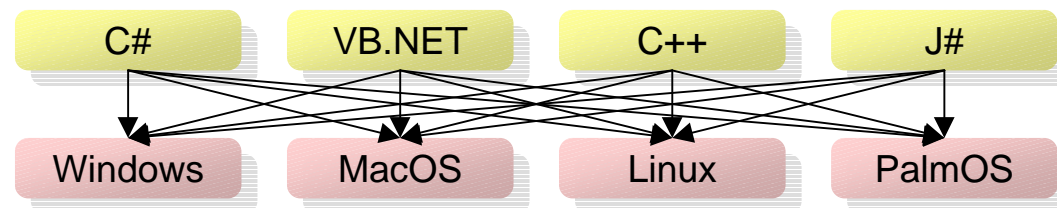
# Compilation



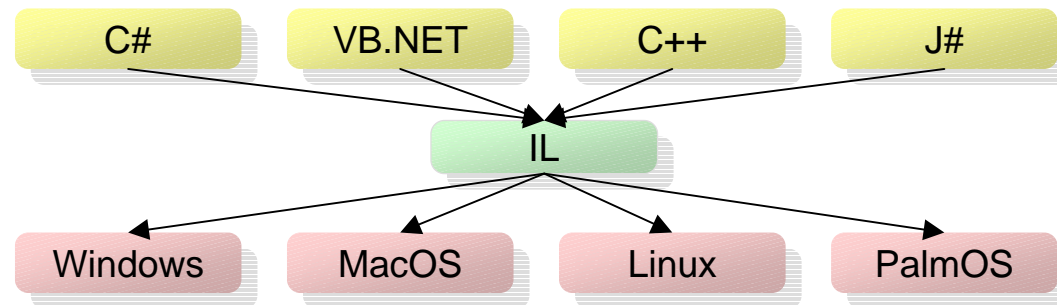
# Vorteile einer virtuellen Maschine

## ■ einfachere Portierbarkeit (Plattform- und Sprachunabhängigkeit)

- *ohne VM:* je ein Compiler pro Sprache und Plattform (z.B.  $4 \times 4 = \underline{16}$ )



- *mit VM:* Übersetzung in Zwischensprache (unter .NET: IL)  
ein Compiler pro Sprache und  
eine CLR (JIT-Compiler) pro Plattform (z.B.  $4 + 4 = \underline{8}$ )



- Kompaktheit des Zwischencodes
- mehr Möglichkeiten zur Optimierung des Maschinencodes

# Vorteile einer virtuellen Maschine

## ■ Sprachen werden gleichwertig, da alle Compiler IL-Code erzeugen

- „eine C# Klasse kann von einer VB.NET Klasse abgeleitet sein“
- einheitliche Fehlerbehandlung

## ■ Compilerbau wird einfacher

- kein Typsystem zu implementieren
- Code kann relativ einfach für IL erzeugt werden
- Sprachen sind per „Definition“ interoperabel

## ■ Gemeinsame Klassenbibliothek für alle Sprache

- .NET Framework

# .NET Virtuelle Maschine - Vergleich

## ■ .NET basiert - wie Java - auf einer "virtuellen Maschine"

- Eine in Software implementierte CPU
- Befehle werden JIT-übersetzt

## ■ Was hat die CLR, was die JVM nicht hat?

- Objekte am Stack (Records, Unions), benutzerdefinierte Werttypen
- Referenzparameter
- Varargs
- Funktionszeiger
- Blockmatrizen
- Überlaufprüfung
- Tail Calls
- ...

Programme (C#, C++, ...)

CLR

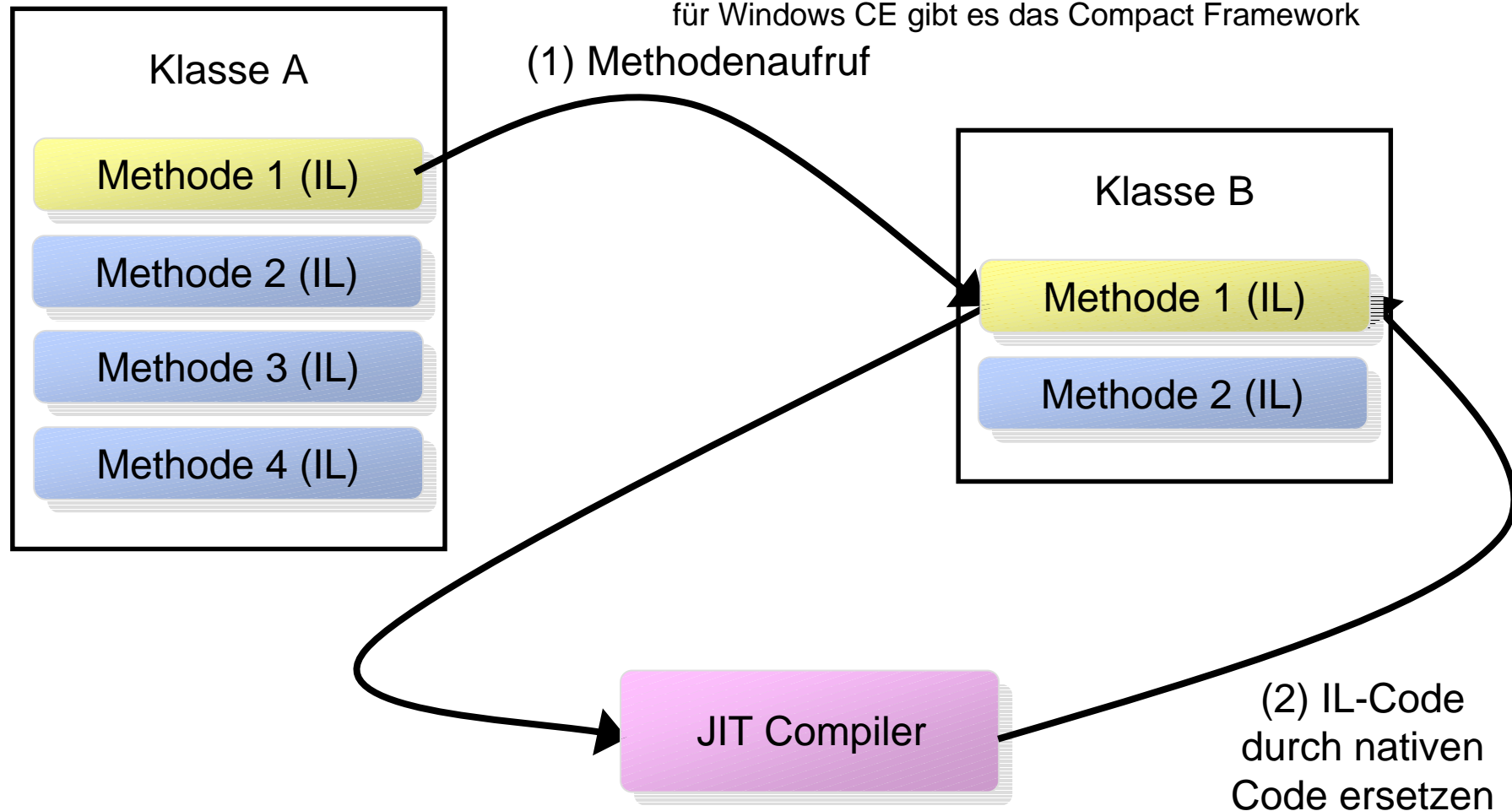
z.B. Intel-Prozessor

CLR bietet bessere Unterstützung für verschiedene Programmiersprachen - dafür läuft die JVM auf mehr Plattformen

- Erzeugt aus IL Code ein Native Executable zur Laufzeit (Just in Time)
- Output ist abhängig von
  - CPU Typ
  - Betriebssystemversion
  - Exakte Identität des Assemblies
  - Exakte Identität aller referenzierten Assemblies
  - Command-line Schaltern
- Statt zur Laufzeit kann auch mittels NGen für Plattform vorkompiliert werden.

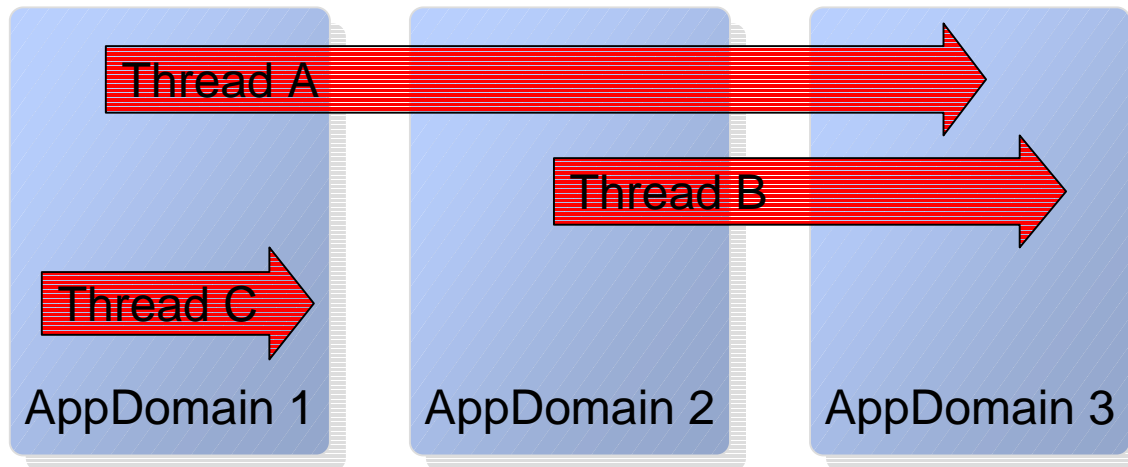
# Just In Time Compilation

IL-Code wird vor der Ausführung immer (!) durch Compiler in echten Maschinencode übersetzt  
Unabhängigkeit von Hardwareplattformen  
für Windows CE gibt es das Compact Framework



# Applikationsdomänen

- sind isolierte Bereiche im CLR-Prozess (= OS-Prozess)  
(*leichtgewichtige Unterprozesse*, aber  $\neq$  Threads)

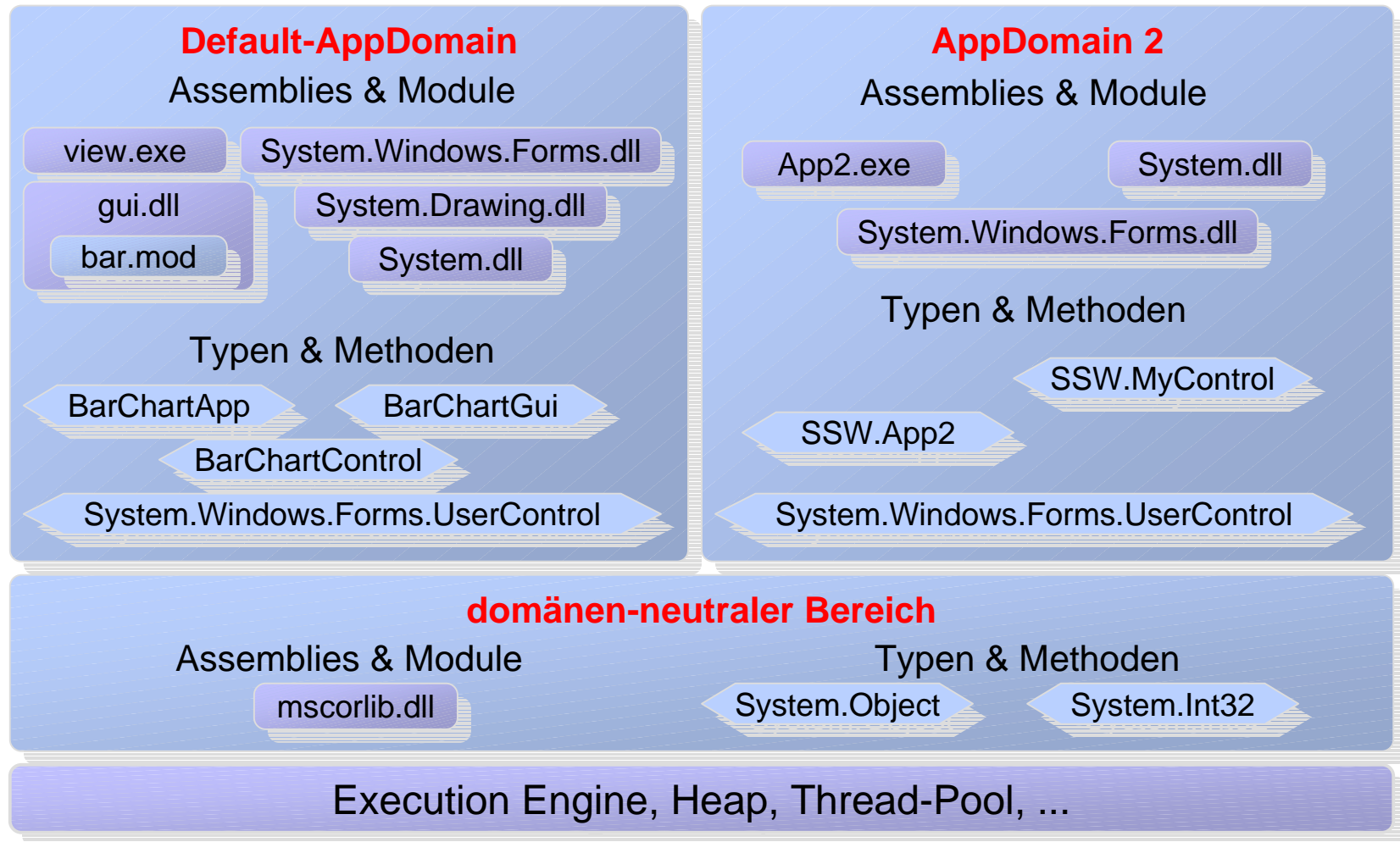


- schirmen Applikationen gegeneinander ab
- umfassen (und isolieren)
  - Assemblies, Module und Typen einer Applikation
  - Methodentabellen der geladenen Typen
  - statische Felder und Objekte der geladenen Typen



## ... Applikationsdomänen

### Windows-Prozess



# AppLauncher: AppDomains erzeugen

*AppLauncher.cs:*

```
using System; using System.Threading;
```

```
class AppLauncher {  
    static void Main () {  
        for (;;) {  
            string appURI = Console.ReadLine();  
  
            if (appURI == "exit") break;  
  
            try {  
                AppThread at = new AppThread(appURI);  
                Thread t =  
                    new Thread(new ThreadStart(at.Execute));  
                t.Start();  
            } catch (Exception e) { . . . }  
        }  
    }  
}
```

```
class AppThread {  
    AppDomain domain;  
    string uri;  
  
    public AppThread (string appURI) {  
        uri = appURI;  
        domain = AppDomain.CreateDomain(uri);  
    }  
  
    public void Execute () {  
        try { domain.ExecuteAssembly(uri); }  
        catch (Exception e) { . . . }  
    }  
}
```

# Intermediate Language

# Wieso IL erlernen?

## ■ “Na und? Wieso soll ich mich darum kümmern? Ich kann C#!”

- Ist nicht C# die einzig wahre Sprache für .NET?

## ■ Gründe um IL zu kennen

- ILDASM: .NET "developer's best friend" (heute eher Reflector)
  - *Disassembliert jede Assembly, inklusive Microsoft's!*
  - *Disassemblierter code: IL*
- selbst C# nutzt die CLI nicht vollständig aus
  - *Auto-generated Naming Patterns (get\_Property, etc)*
  - *Default Method Parameters*
  - *Exception Filters*
- Besseres Verständnis was der Compiler erzeugt und was "abgeht"
- Ein Grund um Assembler-Kurse zu rechtfertigen
- Just plain fun! - na ja.
- Dokumentation von IL im SDK:  
C:\Program Files\Microsoft.NET\FrameworkSDK\v1.1\Tool Developers Guide\docs\Partition III  
IL.doc

## Was ist eine Virtuelle Maschine (VM)?

- Eine in Software implementierte CPU
- Befehle werden interpretiert / JIT-übersetzt
- andere Beispiele: Java-VM, Smalltalk-VM, Pascal P-Code

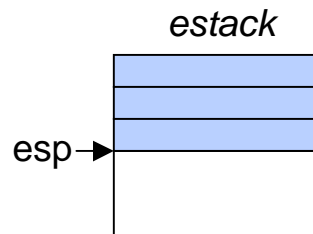
Programme (C#, C++, ...)

CLR

z.B. Intel-Prozessor

## Die CLR ist eine Stackmaschine

- keine Register
- stattdessen *Expression Stack* (auf den Werte geladen werden)



max. Grösse wird für jeder Methode in den Metadaten gespeichert

esp ... expression stack pointer

## Die CLR führt JIT-übersetzten Bytecode aus

- jede Methode wird erst direkt vor der ersten Ausführung übersetzt (= just-in-time)
- Operanden werden in IL symbolisch adressiert (aus Informationen in Metadaten)

# Arbeitsweise einer Stackmaschine

## Beispiel

Anweisung  $i = i + j * 5;$

Angenommene Werte von  $i$  und  $j$

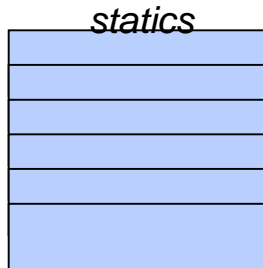
| <i>locals</i> |   |     |
|---------------|---|-----|
| 0             | 3 | $i$ |
| 1             | 4 | $j$ |

## Abarbeitung

| <i>Befehle</i> | <i>Stack</i> |  |
|----------------|--------------|--|
| ldloc.0        | 3            | lade lokale Variable von Adresse 0 auf den Stack (d.h. $i$ ) |
| ldloc.1        | 3 4          | lade lokale Variable von Adresse 1 auf den Stack (d.h. $j$ ) |
| loc.i4.5       | 3 4 5        | lade Konstante 5   |
| mul            | 3 20         | multipliziere die obersten beiden Stackelemente              |
| add            | 23           | addiere die obersten beiden Stackelemente                    |
| stloc.0        |              | speichere oberstes Stackelement auf Adresse 0                |

Am Ende jeder Anweisung ist der Expression Stack wieder leer!

## Globale Variablen



- werden in der CLR zu statischen Klassenfeldern der Programmklasse
- leben während der gesamten Programmausführung (= während Programmklasse geladen ist)
- Adressierung über Metadaten-Tokens  
z.B.  $\text{Idsfld } T_{fld}$  lädt den Wert des von  $T_{fld}$  referenzierten statischen Felds auf den *estack*

Metadaten-Token sind 4 Byte grosse Werte, die Zeilen in Metadaten-Tabellen referenzieren.

|                        |  |
|------------------------|--|
| Token-Type<br>(1 Byte) | Index in Metadaten-Tabelle<br>(3 Byte) |
|------------------------|--|

## Methodenzustand

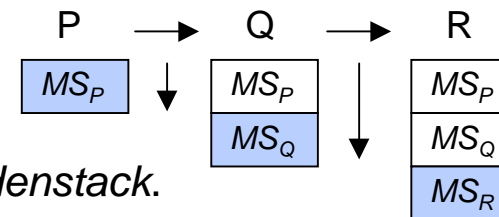
### ■ verwaltet eigene Bereiche für

- Argumente (*args*)
- lokale Variablen (*locals*)
- Expression Stack (*estack*)

### ■ jeder Methodenaufruf hat seinen eigenen Methodenzustand (method state, *MS*)

### ■ Methodenzustände werden kellerartig verwaltet.

Daher spricht man auch von *Stack Frame* und *Methodenstack*.

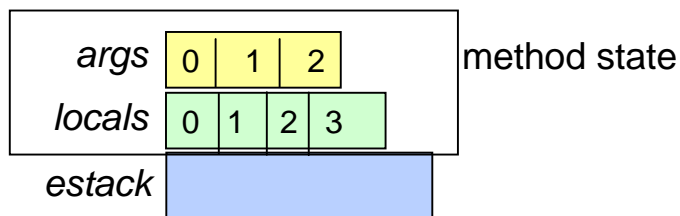


### ■ Jeder Parameter und jede lokale Variable belegen ein Fach ("Slot") von typabhängiger Grösse.

### ■ Adressen sind fortlaufende Nummern, die der Deklarationsreihenfolge entsprechen

z.B. *ldarg.0* lädt den Wert des ersten Methodenarguments auf den *estack*

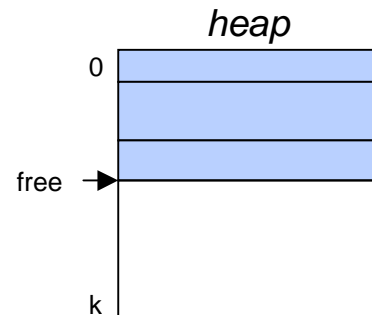
*ldloc.2* lädt den Wert der dritten lokalen Variablen auf den *estack*





## Heap

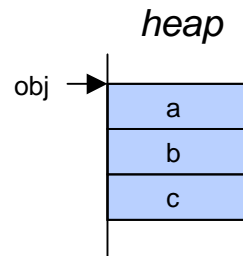
- enthält Klassen- und Array-Objekte



- neue Objekte werden an der Stelle *free* angelegt und *free* wird erhöht; durch die IL-Befehle *newobj* und *newarr*
- Objekte werden vom Garbage Collector freigegeben

## Klassen-Objekte

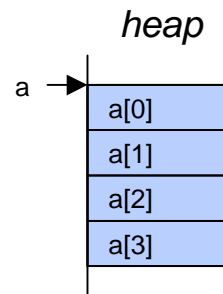
```
class X {  
    int a, b;  
    char c;  
}  
X obj = new X;
```



■ Adressierung durch Field-Token relativ zu *obj*

## Array-Objekte

```
int[] a = new int[4];
```



■ Adressierung durch Indexwert relativ zu *a*

■ nur eindimensionale Arrays können mit den speziellen IL-Anweisung `newarr`, `ldlen`, `ldelem`, `stelem` bearbeitet werden.

## Common Intermediate Language (IL) (hier nur eine Untermenge notwendig)

- sehr kompakt: die meisten Befehle sind nur 1 Byte lang
- meist ungetypt; Typangaben in den Befehlen beziehen sich auf Ergebnistyp

*ungetypt*

```
ldloc.0  
starg.1  
add
```

*getypt*

```
ldc.i4.3  
ldelem.i2  
stelem.ref
```

## Befehlsformat

Sehr einfach im Vergleich zu Intel, PowerPC oder SPARC

```
Code      = { Instruction }.  
Instruction = opcode [ operand ].
```

```
opcode ... 1 oder 2 Byte  
operand ... primitiver Datentyp oder Metadaten-Token
```

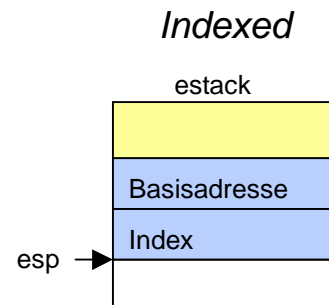
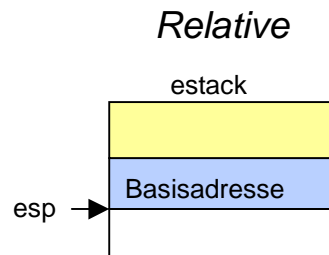
## Beispiele

|             |            |                                       |
|-------------|------------|---------------------------------------|
| 0 Operanden | add        | hat zwei implizite Operanden am Stack |
| 1 Operand   | ldc.i4.s 9 |                                       |

## Adressierungsarten

Wie kann man Operanden in Befehlen ansprechen?

| Adressierungsart   | Beispiel          |  |
|--------------------|-------------------|--|
| ■ <b>Immediate</b> | ldc.i4 123        | für Konstanten   |
| ■ <b>Arg</b>       | ldarg.s 5         | für Methodenargumente  |
| ■ <b>Local</b>     | ldloc.s 12        | für lokale Variablen   |
| ■ <b>Static</b>    | ldsfld <i>fld</i> | für statische Felder ( <i>fld</i> = Metadaten-Token)               |
| ■ <b>Stack</b>     | add               | für geladene Werte am <i>estack</i>                                |
| ■ <b>Relative</b>  | ldfld <i>fld</i>  | für Objektfelder (Objektzeiger liegt am <i>estack</i> )            |
| ■ <b>Indexed</b>   | ldelem.i4         | für Arrayelemente (Arrayzeiger und Index liegen am <i>estack</i> ) |



# Instruktionssatz der CLR

## Laden und Speichern von Methodenargumenten

|                  |                 |  |
|------------------|-----------------|--|
| <b>ldarg.s</b> b | ...<br>..., val | <u>Load</u><br>push(args[b]);            |
| <b>ldarg.n</b>   | ...<br>..., val | <u>Load</u> (n = 0..3)<br>push(args[n]); |
| <b>starg.s</b> b | ..., val<br>... | <u>Store</u><br>args[b] = pop();         |

### Operandentypen

b ... unsigned byte

i ... signed integer

T ... Metadaten-Token

## Laden und Speichern lokaler Variablen

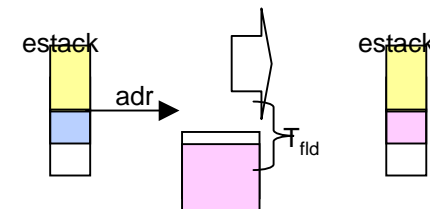
|                  |                 |   |
|------------------|-----------------|---|
| <b>ldloc.s</b> b | ...<br>..., val | <u>Load</u><br>push(locals[b]);               |
| <b>ldloc.n</b>   | ...<br>..., val | <u>Load</u> (n = 0..3)<br>push(locals[n]);    |
| <b>stloc.s</b> b | ..., val<br>... | <u>Store</u><br>locals[b] = pop();            |
| <b>stloc.n</b>   | ..., val<br>... | <u>Store</u> (n = 0..3)<br>locals[n] = pop(); |

## Laden und Speichern globaler Variablen

|               |           |                 |  |
|---------------|-----------|-----------------|--|
| <b>ldsfld</b> | $T_{fld}$ | ...<br>..., val | <u>Load static variable</u><br><code>push(statics[T<sub>fld</sub>]);</code>    |
| <b>stsfld</b> | $T_{fld}$ | ..., val<br>... | <u>Store static variable</u><br><code>statics[T<sub>fld</sub>] = pop();</code> |

## Laden und Speichern von Objektfeldern

|              |           |                      |  |
|--------------|-----------|----------------------|--|
| <b>ldfld</b> | $T_{fld}$ | ..., obj<br>..., val | <u>Load object field</u><br><code>obj = pop(); push(heap[obj+T<sub>fld</sub>]);</code>                   |
| <b>stfld</b> | $T_{fld}$ | ..., obj, val<br>... | <u>Store object field</u><br><code>val = pop(); obj = pop();<br/>heap[obj+T<sub>fld</sub>] = val;</code> |

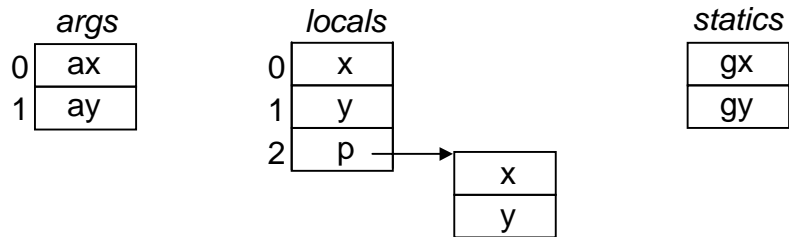


# Instruktionssatz der CLR

## Laden von Konstanten

|                        |                      |  |
|------------------------|----------------------|--|
| <b>ldc.i4</b> <i>i</i> | ...<br>..., <i>i</i> | <u>Load constant</u><br>push( <i>i</i> );                    |
| <b>ldc.i4.n</b>        | ...<br>..., <i>n</i> | <u>Load constant</u> ( <i>n</i> = 0..8)<br>push( <i>n</i> ); |
| <b>ldc.i4.m1</b>       | ...<br>..., -1       | <u>Load minus one</u><br>push(-1);                           |
| <b>ldnull</b>          | ...<br>..., null     | <u>Load null</u><br>push(null);                              |

# Beispiele: Laden und Speichern



|                   | Code   | Bytes            | estack                 |
|-------------------|--|------------------|------------------------|
| <b>ax = ay;</b>   | ldarg.1<br>starg.s 0   | 1<br>2           | ay<br>-                |
| <b>x = y;</b>     | ldloc.1<br>stloc.0   | 1<br>1           | y<br>-                 |
| <b>gx = gy;</b>   | ldsfd T <sub>fld_gy</sub><br>stsfld T <sub>fld_gx</sub>                    | 5<br>5           | gy<br>-                |
| <b>p.x = p.y;</b> | ldloc.2<br>ldloc.2<br>ldfld T <sub>fld_y</sub><br>stfld T <sub>fld_x</sub> | 1<br>1<br>5<br>5 | p<br>p p<br>p p.y<br>- |



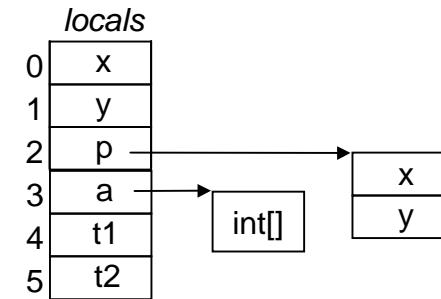
# Instruktionssatz der CLR

## Arithmetik

|            |                                   |   |
|------------|-----------------------------------|---|
| <b>add</b> | ..., val1, val2<br>..., val1+val2 | <u>Add</u><br>push(pop() + pop());              |
| <b>sub</b> | ..., val1, val2<br>..., val1-val2 | <u>Subtract</u><br>push(-pop() + pop());        |
| <b>mul</b> | ..., val1, val2<br>..., val1*val2 | <u>Multiply</u><br>push(pop() * pop());         |
| <b>div</b> | ..., val1, val2<br>..., val1/val2 | <u>Divide</u><br>x = pop(); push(pop() / x);    |
| <b>rem</b> | ..., val1, val2<br>..., val1%val2 | <u>Remainder</u><br>x = pop(); push(pop() % x); |
| <b>neg</b> | ..., val<br>..., -val             | <u>Negate</u><br>push(-pop());                  |

# Beispiele: Arithmetik

|             | Code     | Bytes | Stack |
|-------------|----------|-------|-------|
| $x + y * 3$ | ldloc.0  | 1     | x     |
|             | ldloc.1  | 1     | x y   |
|             | ldc.i4.3 | 1     | x y 3 |
|             | mul      | 1     | x y*3 |
|             | add      | 1     | x+y*3 |



|        |          |   |     |
|--------|----------|---|-----|
| $x++;$ | ldloc.0  | 1 | x   |
|        | ldc.i4.1 | 1 | x 1 |
|        | add      | 1 | x+1 |
|        | stloc.0  | 1 | -   |

|        |           |   |      |
|--------|-----------|---|------|
| $x--;$ | ldloc.0   | 1 | x    |
|        | ldc.i4.m1 | 1 | x -1 |
|        | add       | 1 | x-1  |
|        | stloc.0   | 1 | -    |

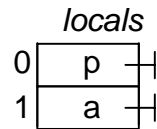
|         |                       |   |         |
|---------|-----------------------|---|---------|
| $p.x++$ | ldloc.2               | 1 | p       |
|         | dup                   | 1 | p p     |
|         | ldfld T <sub>px</sub> | 5 | p p.x   |
|         | ldc.i4.1              | 1 | p p.x 1 |
|         | add                   | 1 | p p.x+1 |
|         | stfld T <sub>px</sub> | 5 | -       |

|          |           |   |            |
|----------|-----------|---|------------|
| $a[2]++$ | ldloc.3   | 1 | a          |
|          | ldc.i4.2  | 1 | a 2        |
|          | stloc.s 5 | 2 | a          |
|          | stloc.s 4 | 2 | -          |
|          | ldloc.s 4 | 2 | a          |
|          | ldloc.s 5 | 2 | a 2        |
|          | ldloc.s 4 | 2 | a 2 a      |
|          | ldloc.s 5 | 2 | a 2 a 2    |
|          | ldelem.i4 | 1 | a 2 a[2]   |
|          | ldc.i4.1  | 1 | a 2 a[2] 1 |
|          | add       | 1 | a 2 a[2]+1 |
|          | stelem.i4 | 1 | -          |

## Objekterzeugung

|                           |  |   |
|---------------------------|--|---|
| <b>newobj</b> $T_{ctor}$  | ... [ arg0, ...,<br>argN ]<br>..., obj | <u>New object</u><br>erzeugt neues Objekt vom durch Constructor-Token angegebenen Typ und führt dann den Konstruktor aus (Argumente am Stack) |
| <b>newarr</b> $T_{eType}$ | ..., n<br>..., arr                     | <u>New array</u><br>erzeugt Array mit Platz für n Elemente vom durch Type-Token angegebenen Typ   |

# Beispiele: Objekterzeugung

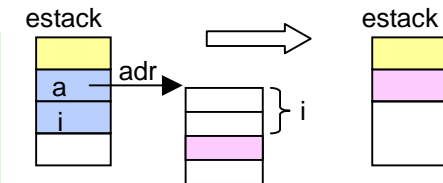


|                        | Code                    | Bytes | Stack |
|------------------------|-------------------------|-------|-------|
| Person p = new Person; | newobj T <sub>P()</sub> | 5     | p     |
|                        | stloc.0                 | 1     | -     |
| int[] a = new int[5];  | ldc.i4.5                | 1     | 5     |
|                        | newarr T <sub>int</sub> | 5     | a     |
|                        | stloc.1                 | 1     | -     |

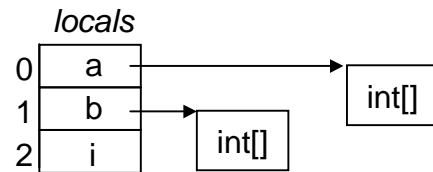
# Instruktionssatz der CLR

## Arrayzugriff

|   |                         |  |
|---|-------------------------|--|
| <b>ldelem.u2</b><br><b>ldelem.i4</b><br><b>ldelem.ref</b> | ..., adr, i<br>..., val | <u>Load array element</u><br>i = pop(); adr = pop();<br>push(heap[adr+i]);<br>Typ des Ergebnisses am Stack:<br>char, int, Objektreferenz                 |
| <b>stelem.i2</b><br><b>stelem.i4</b><br><b>stelem.ref</b> | ...,adr, i, val<br>...  | <u>Store array element</u><br>val=pop(); i=pop(); adr=pop()/4+1;<br>heap[adr+i] = val;<br>Typ des zu speichernden Elements:<br>char, int, Objektreferenz |
| <b>ldlen</b>  | ..., adr<br>..., len    | <u>Get array length</u>  |



# Beispiel: Arrayzugriff



|                       | <i>Code</i> | <i>Bytes</i> | <i>Stack</i> |
|-----------------------|-------------|--------------|--------------|
| <b>a[i] = b[i+1];</b> | ldloc.0     | 1            | a            |
|                       | ldloc.2     | 1            | a i          |
|                       | ldloc.1     | 1            | a i b        |
|                       | ldloc.2     | 1            | a i b i      |
|                       | ldc.i4.1    | 1            | a i b i 1    |
|                       | add         | 1            | a i b i+1    |
|                       | ldelem.i4   | 1            | a i b[i+1]   |
|                       | stelem.i4   | 1            | -            |

# Instruktionssatz der CLR

## Stackmanipulation

|            |                           |  |
|------------|---------------------------|--|
| <b>pop</b> | ..., val<br>...           | <u>Remove topmost stack element</u><br>dummy = pop();                  |
| <b>dup</b> | ..., val<br>..., val, val | <u>Duplicate topmost stack element</u><br>x = pop(); push(x); push(x); |

## Sprünge

|                               |                  |   |
|-------------------------------|------------------|---|
| <b>br</b> <b>i</b>            | ...<br>...       | <u>Branch unconditionally</u><br>pc = pc + i  |
| <b>b&lt;cond&gt;</b> <b>i</b> | ..., x, y<br>... | <u>Branch conditionally</u><br>(<cond> = eq   ge   gt   le   lt   ne.un)<br>y = pop(); x = pop();<br>if (x cond y) pc = pc + i; |

*pc* markiert aktuelle  
Instruktion;  
*i* (Sprungdistanz)  
relativ zum Beginn der  
nächsten Instruktion

## Methodenaufruf

|             |                   |   |   |
|-------------|-------------------|---|---|
| <b>call</b> | $T_{\text{meth}}$ | ... [ arg0, ...<br>argN ]<br>... [ retVal ] | <u>Call method</u><br>nimmt Argumente von <i>estack</i> des<br>Rufers und gibt sie an <i>args</i> des<br>Gerufenen;<br>nimmt Rückgabewert von <i>estack</i><br>des Gerufenen und legt ihn auf<br><i>estack</i> des Rufers |
| <b>ret</b>  |                   | ...<br>...                                  | <u>Return from method</u>   |

## Sonstiges

|              |                 |                        |
|--------------|-----------------|------------------------|
| <b>throw</b> | ..., exc<br>... | <u>Throw exception</u> |
|--------------|-----------------|------------------------|



# Bedingte und unbedingte Sprünge

## Unbedingte Sprünge

```
br offset
```

## Bedingte Sprünge

```
... load operand1 ...  
... load operand2 ...  
beq offset
```

if (operand1 == operand2) br offset

|        |                          |
|--------|--------------------------|
| beq    | jump on equal            |
| bge    | jump on greater or equal |
| bgt    | jump on greater than     |
| ble    | jump on less or equal    |
| blt    | jump on less than        |
| bne.un | jump on not equal        |

# Beispiel: Sprünge

| <i>locals</i> |   |
|---------------|---|
| 0             | x |
| 1             | y |

|              | <i>Code</i> | <i>Bytes</i> | <i>Stack</i> |
|--------------|-------------|--------------|--------------|
| if (x > y) { | ldloc.0     | 1            | x            |
| ...          | ldloc.1     | 1            | x y          |
| }            | ble ...     | 5            | -            |
| ...          |             |              |              |

# Beispiel

|                     |  |                     |  |
|---------------------|--|---------------------|--|
| void Main ()        |  | static void Main () |  |
| int a, b, max, sum; |  |                     |  |
| {                   |  |                     |  |
| if (a > b)          |  | 0: ldloc.0          |  |
|                     |  | 1: ldloc.1          |  |
|                     |  | 2: ble 7 (=14)      |  |
| max = a;            |  | 7: ldloc.0          |  |
|                     |  | 8: stloc.2          |  |
|                     |  | 9: br 2 (=16)       |  |
| else max = b;       |  | 14: ldloc.1         |  |
|                     |  | 15: stloc.2         |  |
| while (a > 0) {     |  | 16: ldloc.0         |  |
|                     |  | 17: ldc.i4.0        |  |
|                     |  | 18: ble 15 (=38)    |  |
| sum = sum + a * b;  |  | 23: ldloc.3         |  |
|                     |  | 24: ldloc.0         |  |
|                     |  | 25: ldloc.1         |  |
|                     |  | 26: mul             |  |
|                     |  | 27: add             |  |
|                     |  | 28: stloc.3         |  |
| a--;                |  | 29: ldloc.0         |  |
|                     |  | 30: ldc.i4.1        |  |
|                     |  | 31: sub             |  |
|                     |  | 32: stloc.0         |  |
| }                   |  | 33: br -22 (=16)    |  |
| }                   |  | 38: return          |  |

## Adressen

a ... 0  
b ... 1  
max ... 2  
sum ... 3

## ■ .class

- interface: class is actually an interface
- Access control: public, private
- String handling: ansi, autochar, unicode
- beforefieldinit: don't type-init on static method calls

```
.class private auto ansi beforefieldinit App
    extends [mscorlib]System.Object
{
    // ...
} // end of class App
```

## ■ .field

- Access control: public, assembly, family, famandassem, famorassem, private
- initonly: constant field ("readonly" in C#)
- literal: constant value; inline replacement when used
- static: one instance for all type instances

```
.class private auto ansi beforefieldinit App
    extends [mscorlib]System.Object
{
    .field private string message
    .field private static object[] cachedValues
} // end of class App
```

## ■ .method

- Method name:
  - *.ctor, .cctor: Spezieller Name für Konstruktoren*

```
...  
.method private hidebysig static void Main() IL managed  
{  
  .entrypoint  
  .maxstack 1  
  ldstr    "Hello, IL"  
  call     void [mscorlib]System.Console::WriteLine(string)  
  ret  
} // end of method App::Main  
...
```


# Methoden, Parameterübergabe auf dem Stack

Alle Namen müssen mit Assembly qualifiziert und vollständig aufgelöst angegeben werden:

`[assembly]namespace.class::Method`  
`[mscorlib]System.Object::WriteLine`

```
string s1, s2  
.  
.  
.  
If (s1.CompareTo(s2) == 0) ...
```

Lokale Variablen und Parameter  
auf dem Stack, Resultat auch.



```
// IL  
.locals init ([0] string s1, [1] string s2)  
.  
.  
.  
ldloc.0  
ldloc.1  
callvirt instance int32 [mscorlib]System.String::CompareTo(string)  
ldc.i4.0  
bne.un.s endif
```

# IL assembler - Hello IL

## ■ ILAsm (IL Assembler) nahe an reinem IL

- `ilasm.exe` == MASM for .NET
- Bestandteil des FrameworkSDK,
- Assembly language
  - *IL Opcodes und Operanden*
  - *Assembler Direktiven*
  - *Nimmt Rücksicht auf CLI (objects, interfaces, etc)*

```
.assembly extern mscorlib { }  
.assembly Hello { }  
  
.class private auto ansi beforefieldinit App  
    extends [mscorlib]System.Object  
{  
    .method private hidebysig static void Main() IL managed  
    {  
        .entrypoint  
        .maxstack 1  
        ldstr    "Hello, IL"  
        call     void [mscorlib]System.Console::WriteLine(string)  
        ret  
    } // end of method App::Main  
} // end of class App
```



# Hello World übersetzen und ausführen

## ■ Compiling, Running

```
C:\Prg\Demos>ilasm Hello.il
```

```
Microsoft (R) .NET Framework IL Assembler. Version 1.0.3705.0  
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.  
Assembling 'Hello.il' , no listing file, to EXE --> 'Hello.EXE'  
Source file is ANSI
```

```
Assembled method App::Main  
Creating PE file
```

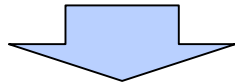
```
Emitting members:  
Global  
Class 1 Methods: 1;  
Writing PE file  
Operation completed successfully
```

```
C:\Prg\Demos>hello  
Hello, IL
```

# DAS Beispiel: Hello, .NET-World!

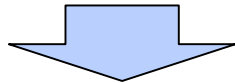
*HelloWorld.cs:*

```
class HelloWorldApp {  
    static void Main () {  
        System.Console.WriteLine("Hello, .NET-World!");  
    }  
}
```

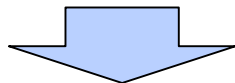


*Übersetzen und Assembly erzeugen (mit C#-Compiler):*

> csc HelloWorld.cs



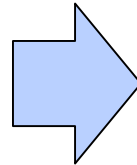
*Assembly:* HelloWorld.exe (3072  
Byte!)



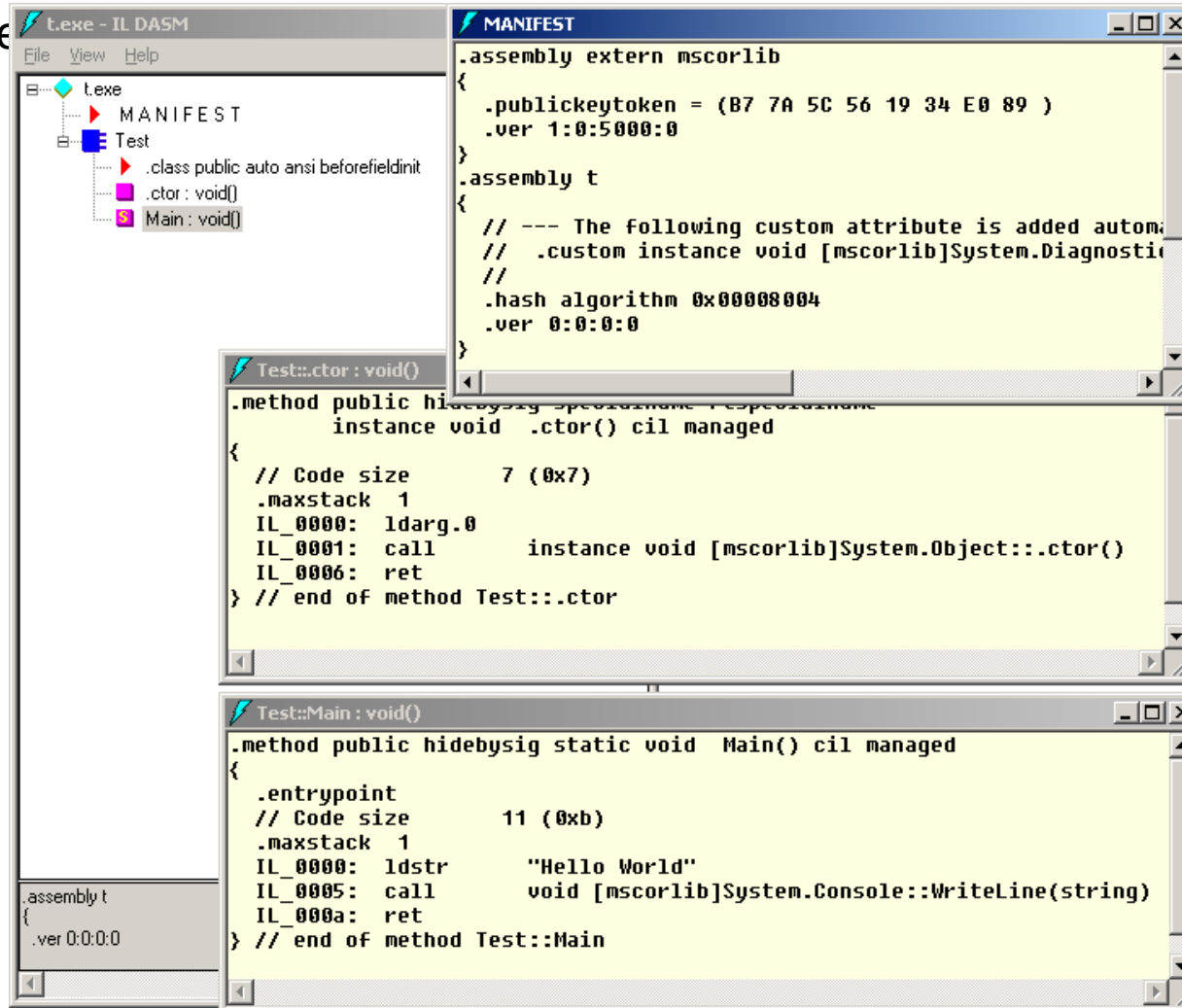
*Metadaten und IL-Code  
betrachten*

*(mit IL-Disassembler):*

> ildasm HelloWorld.exe



## ■ "Disasse



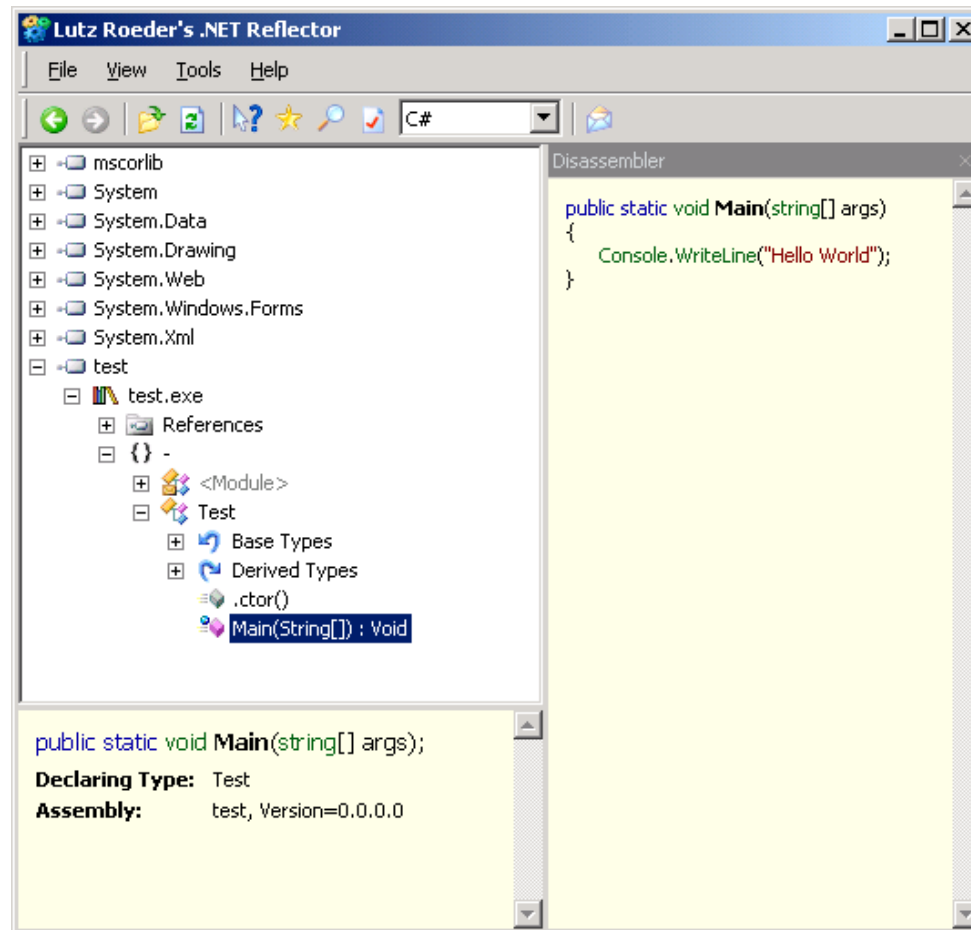
The screenshot displays the ILDASM interface with three main panes:

- Left Pane (Tree View):** Shows the assembly structure. The 'Test' class is selected, showing its methods: `.ctor : void()` and `Main : void()`.
- Top Right Pane (Manifest):** Displays the assembly manifest for 't.exe'. It includes the public key token `(B7 7A 5C 56 19 34 E0 89 )` and version `1:0:5000:0`. It also shows a custom attribute for `[mscorlib]System.Diagnostics.DebuggableAttribute` with a hash algorithm of `0x00008004` and version `0:0:0:0`.
- Bottom Right Pane (Method Disassembly):** Shows the disassembly of the `Test::Main : void()` method. The code is as follows:
 

```
.method public hidebysig static void Main() cil managed
{
    .entrypoint
    // Code size      11 (0xb)
    .maxstack 1
    IL_0000: ldstr      "Hello World"
    IL_0005: call       void [mscorlib]System.Console::WriteLine(string)
    IL_000a: ret
} // end of method Test::Main
```

# .NET Reflector oder ILSpy oder dotPeek

- Erzeugt aus Code eine Repräsentation in IL, C#, VB.NET



## ■ Architektur

- Common Language Runtime (CLR)
- Common Intermediate Language (IL)
- Common Type System (CTS)
- Assembly Format, Versionierung

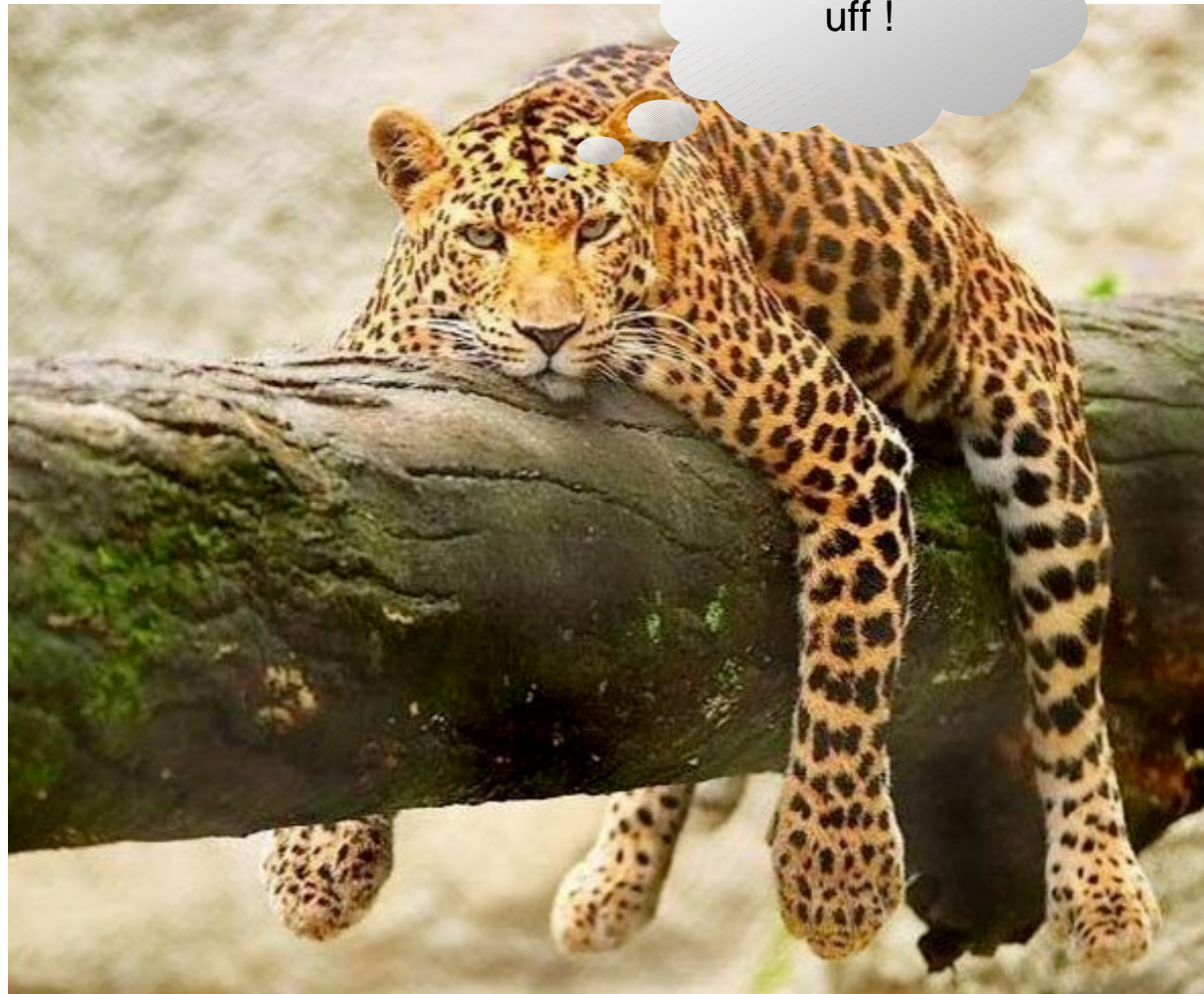
## ■ Common Language Runtime

- Ausführungssystem
- Security, Garbage Collection

## ■ Intermediate Language

- Assembler für .NET
- Typen unterstützt
- Stackmaschine
- Wird zur Ladezeit in Maschinen-Code übersetzt

# Fragen?



# Anhang: Überblick Tools

|                                     |                       |
|-------------------------------------|-----------------------|
| ■ C#-Compiler                       | ⇒ <b>csc.exe</b>      |
| ■ Visual Basic .NET-Compiler        | ⇒ <b>vbc.exe</b>      |
| ■ J# Compiler                       | ⇒ <b>vjc.exe</b>      |
| ■ JScript Compiler                  | ⇒ <b>jsc.exe</b>      |
| ■ IL-Disassembler                   | ⇒ <b>ildasm.exe</b>   |
| ■ GUI-Debugger                      | ⇒ <b>dbgclr.exe</b>   |
| ■ .NET Framework Configuration Tool | ⇒ <b>mscorcfg.msc</b> |
| ■ Assembly Cache Viewer             | ⇒ <b>sfusion.dll</b>  |
| ■ Global Assembly Cache Utility     | ⇒ <b>gacutil.exe</b>  |
| ■ Strong Name Tool                  | ⇒ <b>sn.exe</b>       |
| ■ Code Access Security Policy Tool  | ⇒ <b>caspol.exe</b>   |

# Anhang A: wichtige Verzeichnisse

## ■ Windowsspezifische Verzeichnisse

- *Windows*: z.B. c:\WINDOWS\ , c:\WINNT\
- *Programs*: z.B. c:\Program Files\ , c:\Programme\
- *User*: z.B. c:\Documents and Settings\*UserName*\ ,  
c:\Dokumente und Einstellungen\*BenutzerName*\

## ■ .NET-spezifische Verzeichnisse

- *.NET-Runtime*:  
z.B. *Windows*\Microsoft.NET\Framework\v1.0.3705
- *.NET-SDK*:  
z.B. *Programs*\Microsoft.NET\FrameworkSDK oder  
*Programs*\Microsoft Visual Studio .NET\FrameworkSDK
- Global Assembly Cache (GAC): z.B. *Windows*\assembly\



# Anhang B: Konfigurationsdateien

## ■ Maschinenkonfiguration

- *.NET-Runtime*\CONFIG\machine.config

## ■ Applikationskonfiguration

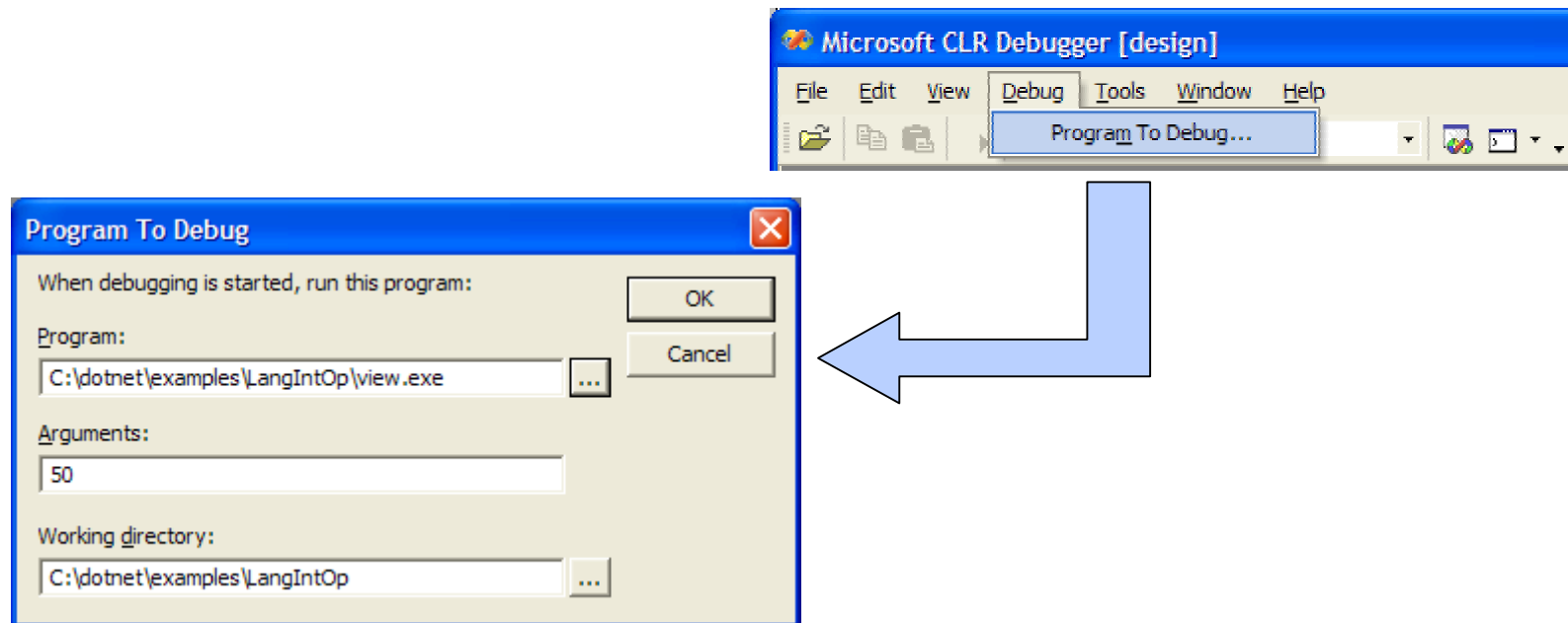
- im Applikationsverzeichnis
- Dateiname = Applikationsdateiname + .config  
z.B. MyApp.exe → MyApp.exe.config

## ■ Sicherheitspolitik

- Ebene ENTERPRISE: *.NET-Runtime*\CONFIG\enterprisesec.config
- Ebene MACHINE: *.NET-Runtime*\CONFIG\security.config
- Ebene USER: z.B.  
*User*\Application Data\Microsoft\CLR Security Config\v1.0.03705\security.config
- Ebene APPDOMAIN: mit AppDomain.SetAppDomainPolicy ( ... )

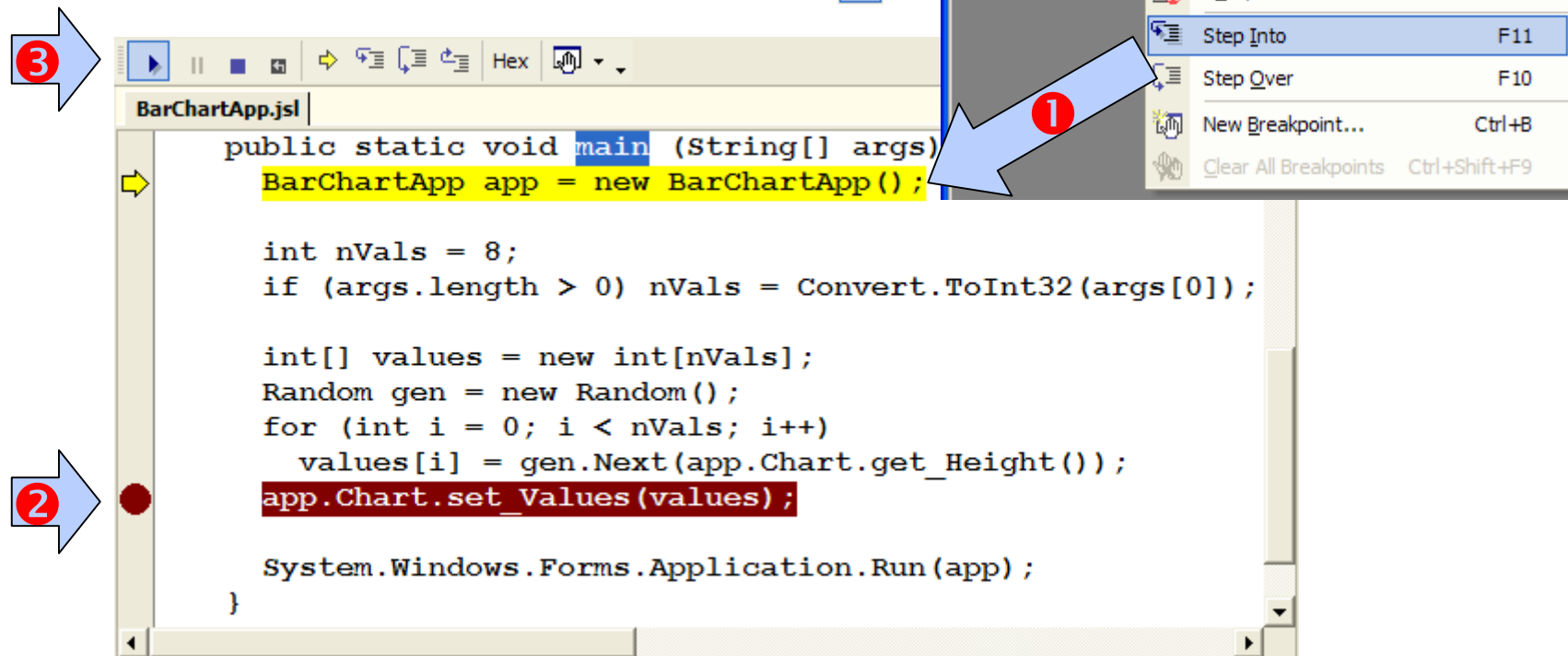
# Anhang C: GUI-Debugger

- GUI-Debugger starten
  - `.NET-SDK\GuiDebug\DbgCLR.exe`
- Programm auswählen

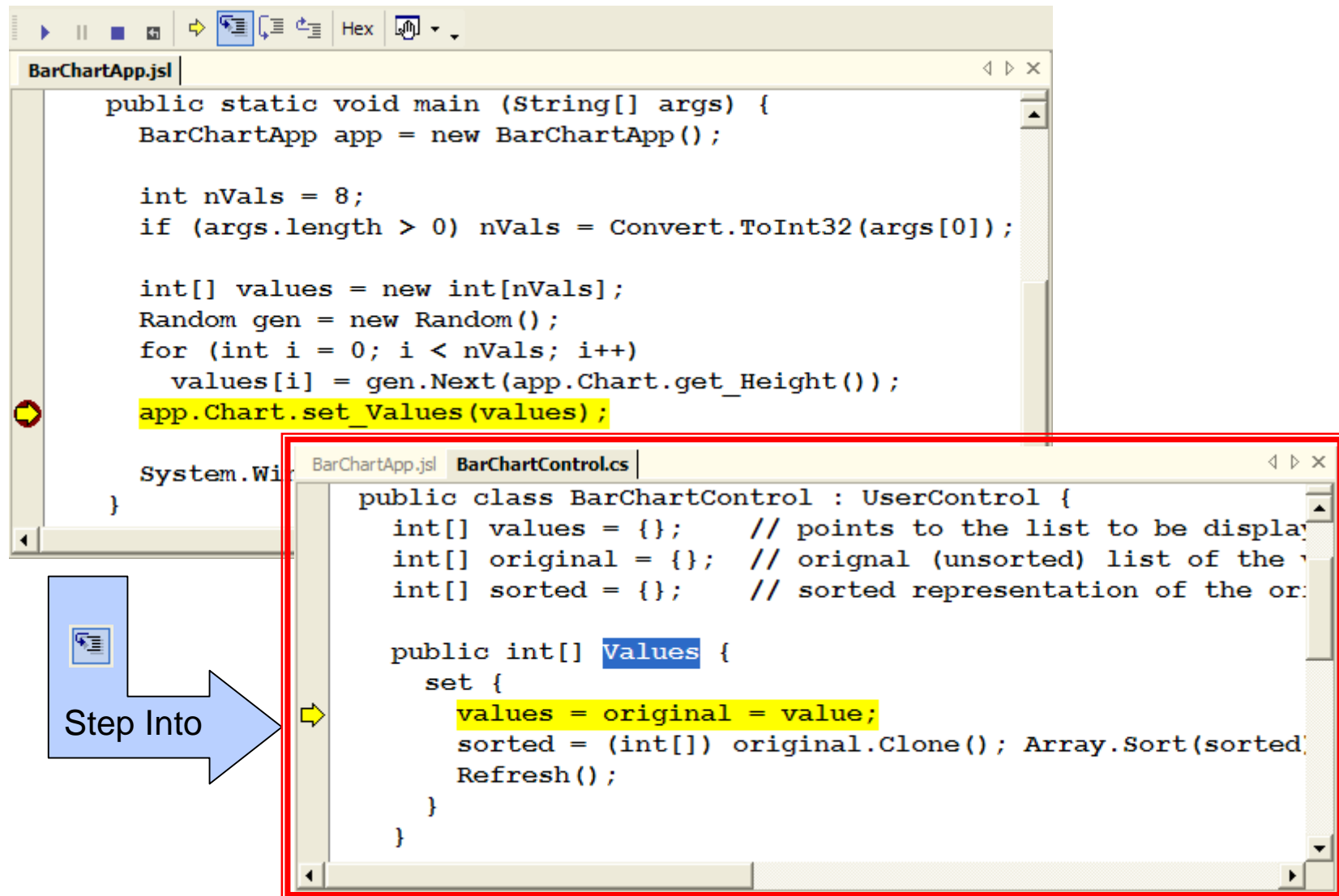


# GUI-Debugger

- 1 Debugging starten
  - *Step Into (F11)*
- 2 Breakpoint setzen
  - *Klick in linken Rand*
- 3 bis Breakpoint laufen lassen

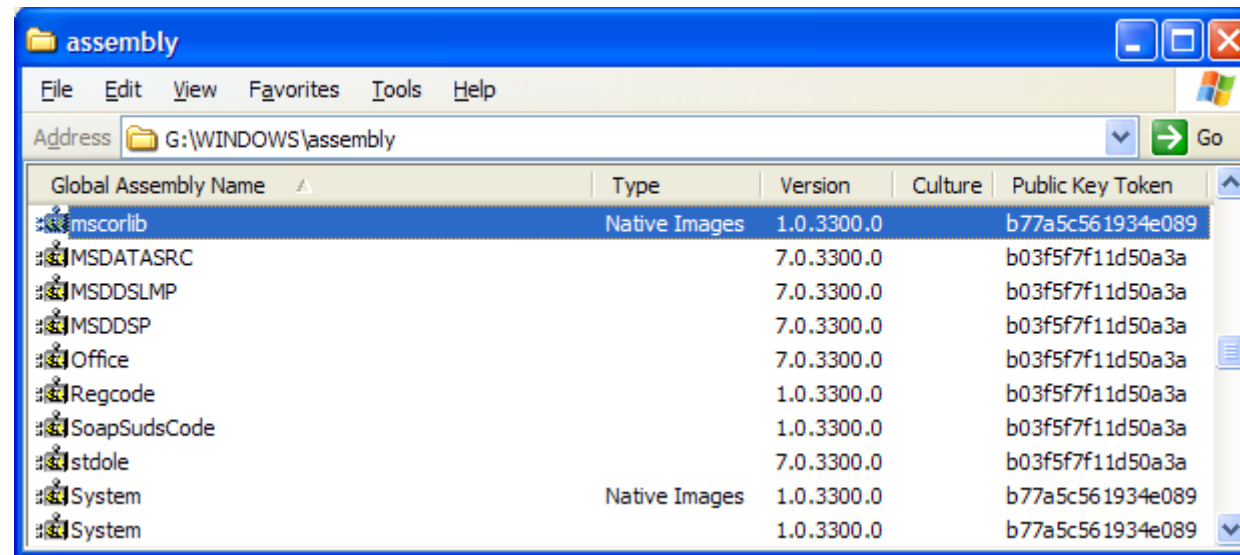


# Cross-Language Debugging



# Anhang D: Assembly Cache Viewer

- stellt Global Assembly Cache (GAC) im Windows Explorer wie gewöhnliches Verzeichnis dar



- tatsächliche Pfade: z.B.

- Assembly **mscorlib**:  
GAC\*\NativeImages1\_v1.0.3705\mscorlib\  
1.0.3300.0\_\_b77a5c561934e089\_be4960bd\mscorlib.dll
- Assembly **System**:  
GAC\*\GAC\System\1.0.3300.0\_\_b77a5c561934e089\System.dll

\*) siehe Teil 2: Die .NET-  
Architektur,  
Anhang A: wichtige Verzeichnisse

# Global Assembly Cache Utility (gacutil.exe)

Kommandozeilen-Werkzeuge zum

- Assemblies im GAC installieren

> gacutil **-i** GlobalLib.dll

- Assemblies aus GAC entfernen

> gacutil **-u** GlobalLib

- Inhalt des GAC anzeigen

> gacutil **-l**

- unterstützt auch Referenzzähler auf globale Assemblies

> gacutil **-ir** GlobalLib.dll FILEPATH c:\Apps\MyApp.exe

> gacutil **-ur** GlobalLib FILEPATH c:\Apps\MyApp.exe

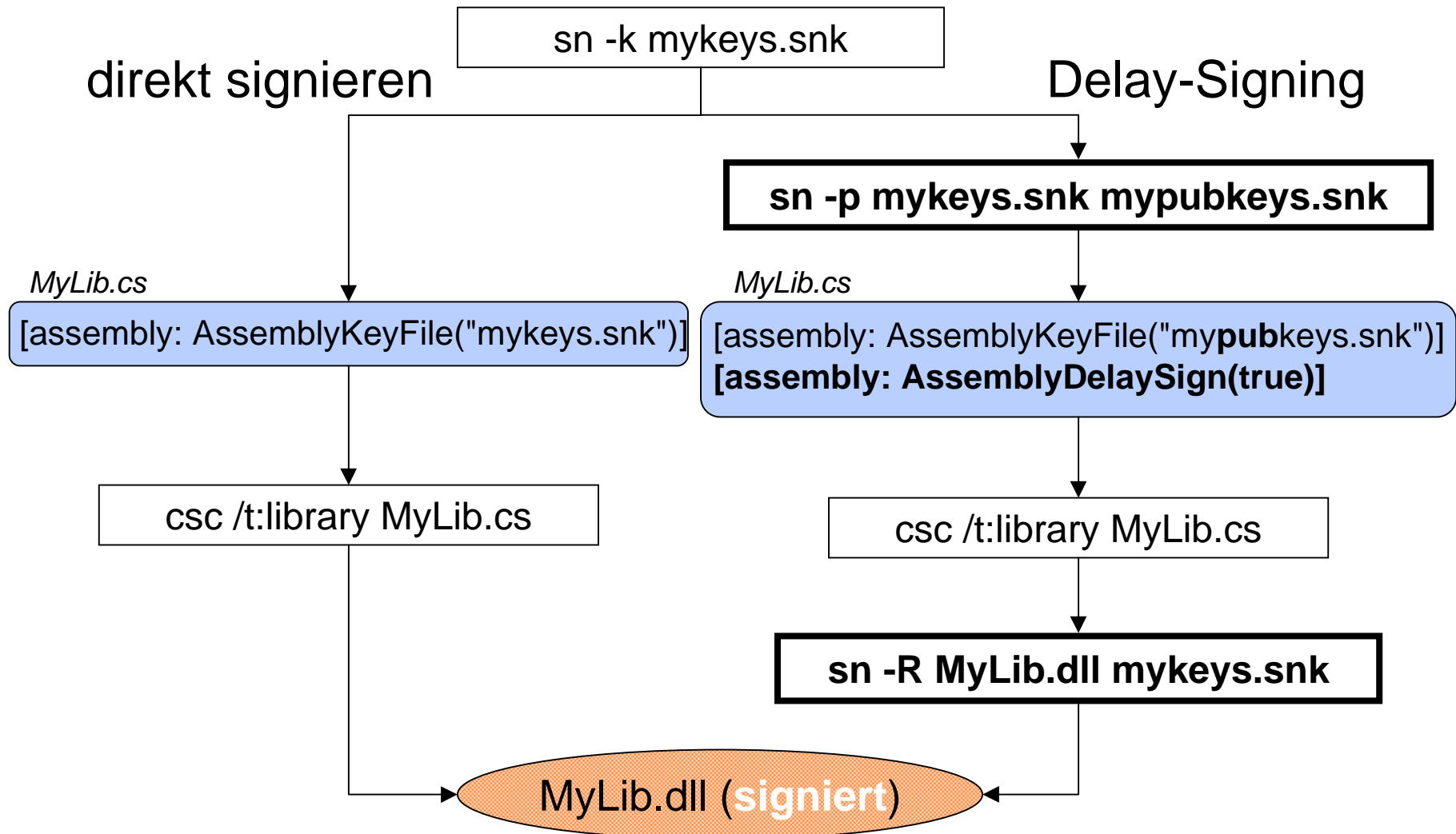
> gacutil **-lr**

- solange Referenz vorhanden, kann Assembly nicht aus GAC entfernt werden

# Anhang E:Strong Name Tool (sn.exe)

Kommandozeilen-Werkzeug zum

- Erzeugen von privaten & öffentlichen Schlüsselpaaren  
> sn **-k** mykeys.snk
- Extrahieren des öffentlichen Schlüssels auf einem Paar  
> sn **-p** mykeys.snk mypubkey.snk
- verspäteten Signieren von Assemblies  
> sn **-R** MyLib.dll mykeys.snk
- Wozu Signieren?
  - ein "starker" Assemblyname (*strong name*) besteht aus:  
Namen, Versionsnummer, Sprachmerkmal, öffentl. Schlüssel  
z.B.            MyLib, Version=1.2.745.18000, Culture=en-US,  
                  PublicKeyToken=13a3f300fff94cef
  - ein Assemblyname wird erst durch Signieren "stark"
  - nur signierte Assemblies können im GAC installiert werden





Welche Ausgabedatei soll erzeugt werden?

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

**/out:*name*** Name des Assembly oder des Moduls

Name der Quelldatei ist,

Name der ersten

Default bei /t:exe *name.exe*, wobei *name* der

die *Main*-Methode enthält

Default bei /t:library *name.dll*, wobei *name* der

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*

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)