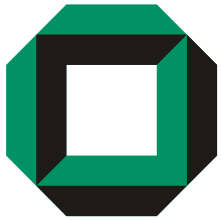


Universität Karlsruhe (TH)
Forschungsuniversität · gegründet 1825

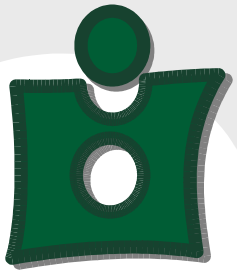
Moderne Entwicklungsumgebungen am Beispiel .NET

Die .NET-Architektur
Teil 1

Ali Jannesari
Institut für Programmstrukturen und Datenorganisation (IPD)

Inhalt

- **Common Language Infrastructure (CLI)**
- **Common Intermediate Language (CIL)**
- **Common Type System (CTS)**
- Common Language Specification (CLS)
- Metadaten
  - Attribute
  - Reflection
- Assemblies
- Virtual Execution System (VES)

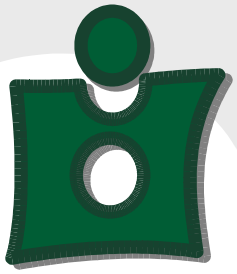


CLI - Common Language Infrastructure

- CLI = Grundgerüst der .NET-Architektur
- wird im ECMA-Standard 335 beschrieben (3rd edition June 2005)
  http://www.ecma-international.org/publications/standards/Ecma-335.htm
- ISO 35.060
  - 35 Information technology. Office machines
  - 060 Languages used in information technology
  - ISO/IEC 23271:2003
  - ISO/IEC TR 23272:2003
- CLI ähnelt in vielen Punkten den folgenden Technologien:
  - Object Management Group's (OMG's)
    - Common Object Request Broker Architecture (CORBA)
  - Microsoft's Component Object Model
    - COM/Distributed COM (DCOM)
  - Java™ and related technologies

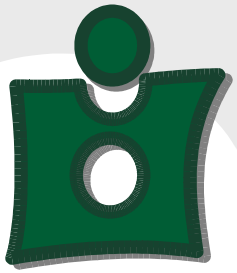
CLI - Common Language Infrastructure (ECMA-335)

- Schicht I: Architektur und Konzepte
  - Beschreibt die grundlegende Architektur der CLR, und stellt die Definition des Common Type Systems (CTS) bereit, das virtuelle Ausführungssystem (VES), und die Common Language Specification (CLS).
- Schicht II: Metadaten und Semantik
  - Stellt die Beschreibung der Metadaten zur Verfügung: physikalische Anordnung (als Dateiformat), logischer Inhalt (als Menge von Tabellen und Beziehungen), und die Semantik (aus der Sicht eines hypothetischen Assemblers, ilasm).
- Schicht III: CIL Befehlssatz
  - Detaillierte Beschreibung des Common Intermediate Language (CIL) Befehlssatzes.



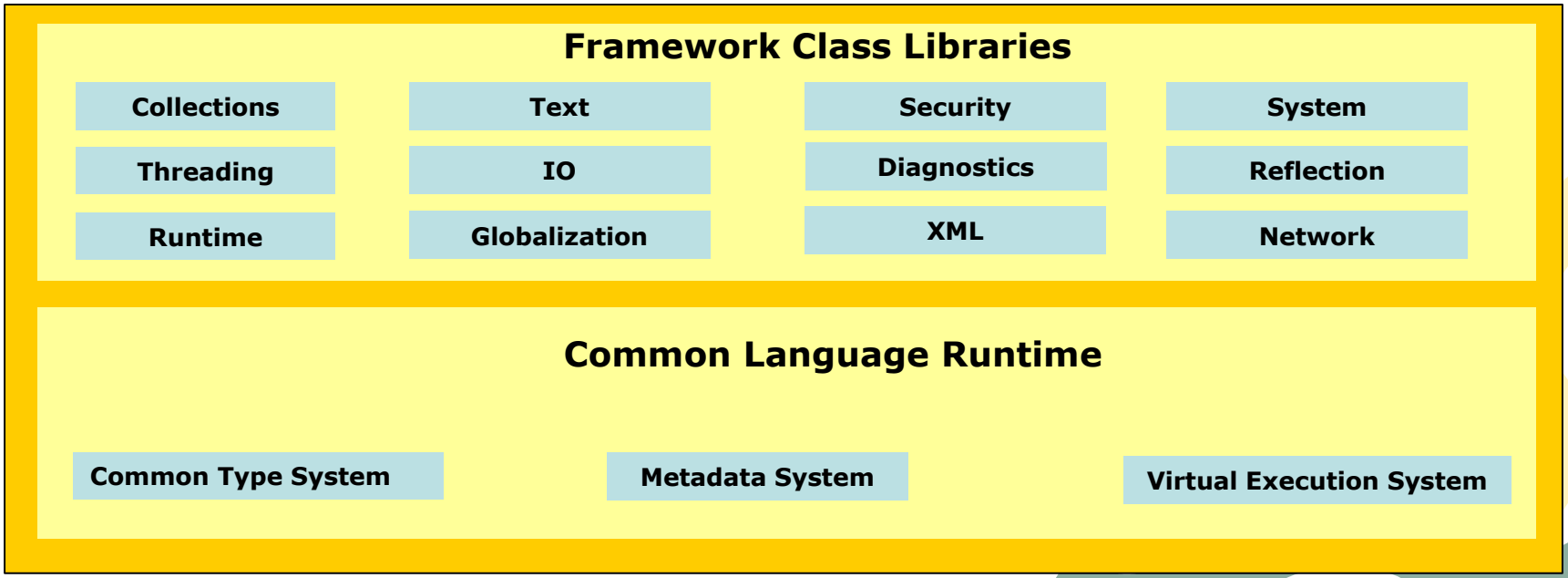
CLI - Common Language Infrastructure (ECMA-335)

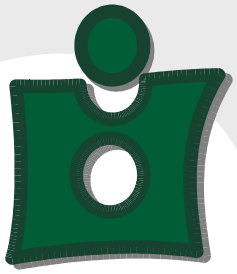
- Schicht IV: Profile und Bibliotheken
  - Stellt eine Übersicht der CLI-Bibliotheken und eine detaillierte Beschreibung der Klassen, Schnittstellen- und Werttypen im XML-Format zur Verfügung.
- Schicht V: Annexes
  - Enthält einige in CIL implementierte Beispielprogramme, Informationen über spezifische Implementierung eines Assemblers, eine maschinen-lesbare Beschreibung des CIL-Befehlssatzes und Werkzeuge zur CIL- Manipulation.



Common Language Infrastructure (CLI)

- Die CLI enthält zwei Hauptbestandteile:
- Common Language Runtime (CLR)
  - Eine Verwaltete Ausführungsumgebung, welche von verschiedenen Sprachen verwendet werden kann.
- Framework Class Libraries (FCL)
  - Services und Strukturen um komplexe Anwendungen zu entwickeln.





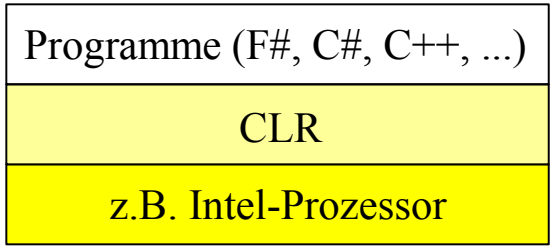
Common Language Infrastructure (CLI)

- Common Type System:
  - Kern der CLR
    - Beschreibt Datentypen von .NET und ihr Verhalten
- Metadata System:
  - Typen beschreiben sich selbst
  - sprachunabhängig
  - Metadaten können auch von andren Tools (IDEs /Debuggern) verwendet werden
- Virtual Execution System:
  - Implementiert das CTS-Modell
  - Lädt .NET Programme und führt diese aus
  - Späte Bindung

Common Language Infrastructure (CLI)

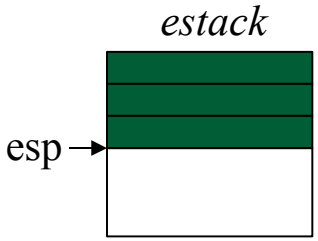
Was ist eine Virtuelle Maschine (VM)?

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



Die CLR ist eine Kellermaschine

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



max. Größe wird für jeder Methode in den Metadaten gespeichert

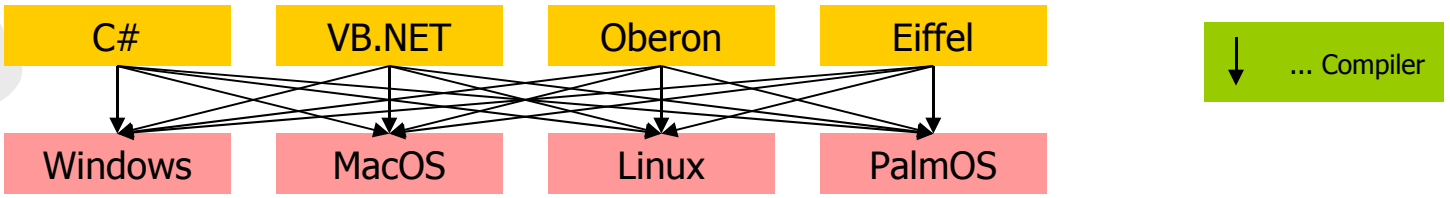
esp ... expression stack pointer

Die CLR führt JIT-übersetzen Bytecode aus

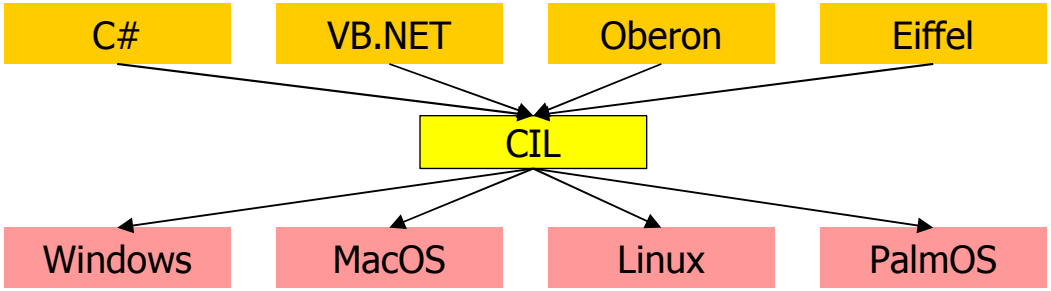
- jede Methode wird erst direkt vor der ersten Ausführung übersetzt (= just-in-time)
- Operanden werden in IL symbolisch adressiert (aus Informationen in Metadaten)

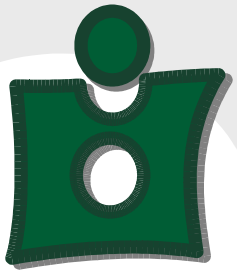
Vorteile einer virtuellen Maschine

- **Einfachere Portierbarkeit (Plattform- und Sprachunabhängigkeit)**
  - **ohne *VM*:** je ein Compiler pro Sprache und Plattform (z.B. 4×4=**16**)



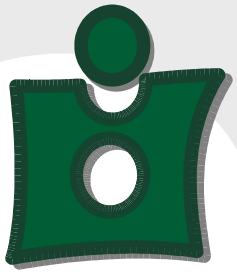
  - **mit VM**: Übersetzung in Zwischensprache (unter .NET: CIL)
    ein Compiler pro Sprache und
    eine CLR (JIT-Compiler) pro Plattform (z.B. 4+4=8)





Vorteile einer virtuellen Maschine

- **Kompaktheit des Zwischencodes**
  - Kompakter als Quellcode
  - Kompakter als Native-Code
- **Optimierter Code**
  - mehr Möglichkeiten zur Optimierung des Maschinencodes
  - Analysierung der Zielmaschinenkonfiguration von JIT-Compiler
    - Laufzeitoptimierung
    - Ressourcenverbrauch

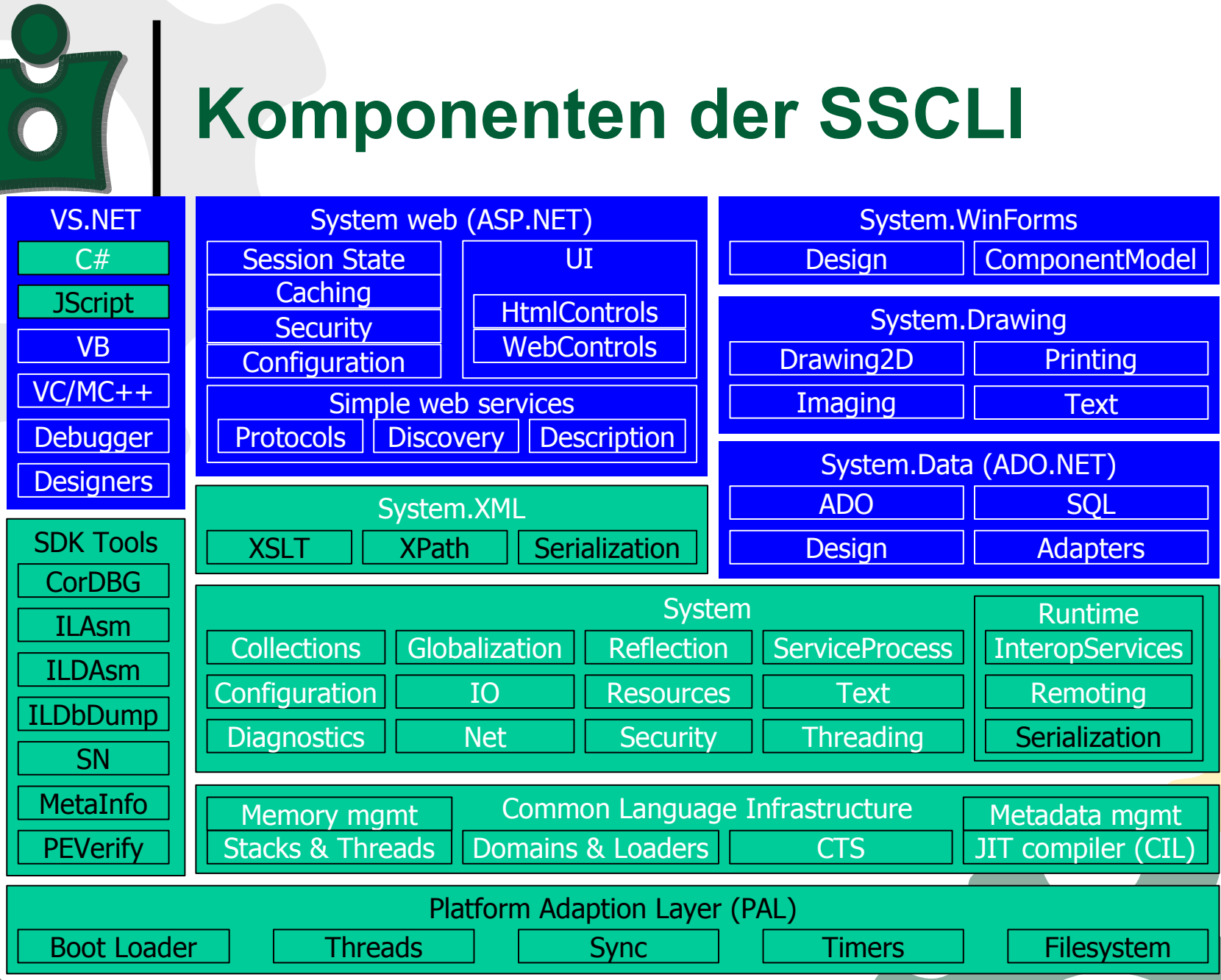


Plattformen & Implementierungen

- Microsoft Windows (95, 2000, XP, ...):
  - **Microsoft .NET-Framework (**+Compact Framework**)**:
    enthält Common Language Runtime (CLR)
    = Microsofts Implementierung des CLI-Standards

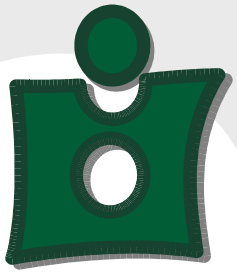
    http://msdn.microsoft.com/netframework
- Microsoft Windows XP, FreeBSD, Mac OS X 10.2:
  - **SSCLI** (a.k.a. Rotor): Shared Source Common Language Infrastructure
    http://msdn.microsoft.com/net/sscli
    http://www.sscli.net (Rotor Community Site)
- Linux, Unix (Solaris):
  - **Project Mono** by Ximian (Gnome) und Novell:
    http://www.go-mono.org
  - **DotGNU**:
    http://www.gnu.org/projects/dotgnu

Komponenten der SSCLI



Quelle: Stutz, Neward, Shilling: *Shared Source CLI Essentials*, OReilly, 2003

ECMA Standard | zusätzlich in Rotor enthalten | in kommerzieller CLR enthalten



Common Intermediate Language (CIL)

- Die CLI versteht nur eine Zwischensprache: Common Intermediate Langugae (CIL)
- CIL-Anweisungen sind vom Typ der Argumente völlig unabhängig (Gegensatz zu Java-ByteCode)
- Beispiel:

Java-Bytcode (Integer)

```
iload_0
iload_1
iadd
istore_2
```

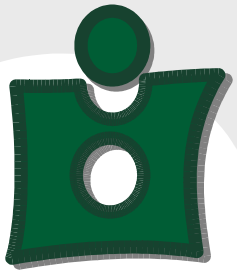
Java-Bytecode (Floats)

```
fload_0
fload_1
fadd
fstore_2
```

CIL (Floats und Integer)

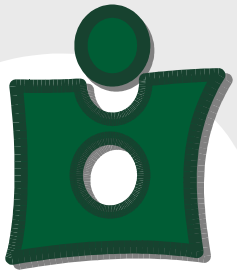
```
ldloc.0
ldloc.1
add
stloc.2
```

- Vereinfachung der Erzeugung der CIL für Compiler
  - System.Reflection.Emit
- Erschwert die Arbeit der JIT-Compiler
  - Typbestimmung
- Stack Transition Diagram: Was vor und nach dem Ausführung jedes Befehls am Stack sein soll.



Common Intermediate Language (CIL)

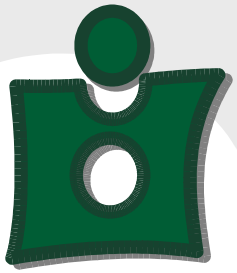
- CIL-Anweisungen arbeiten nur mit folgende Typen auf dem Keller (Kellertypen):
  - Für Ganzzahlen: native int (i), int32 (i4) und int64 (i8)
  - Für Gleitkommazahlen: F
  - Für Zeiger: Objektzeiger (object), verwalteter Zeiger (&), unverwalteter Zeiger (native unsigned int und *)
- Alle andren Typen, die die CLR verwendet (CLR-Typen), werden durch die entsprechenden Lade- und Speicherbefehle am Übergang zum/vom Keller umgewandelt.
- Befehle: ldloc, stloc, ldarg, starg verwenden nur eine generische form (int32) auf dem Keller
  - CLR wird die Bestimmung von Quell- und Zieltyp übernehmen
- Andere Befehle geben den Typ der Speicherzelle explizit
  - Indirekt laden: ldind.<Typ> z.B. ldind.i2
- Explizit Konvertierung zwischen Kellertypen und CLR-Typen
  - Conv.<Zieltyp> z.B. conv.i1 (nach int8 konvertieren)
- Stack-Typen als vorzeichenlose Ganzzahltypen
  - durch Anhängen des Zusatzes .un an CIl-anweisung: z.B. :Div.un, add.ovf.un



Common Intermediate Language (CIL)

- CLI-Beispiel: „**HelloWorld.IL**“

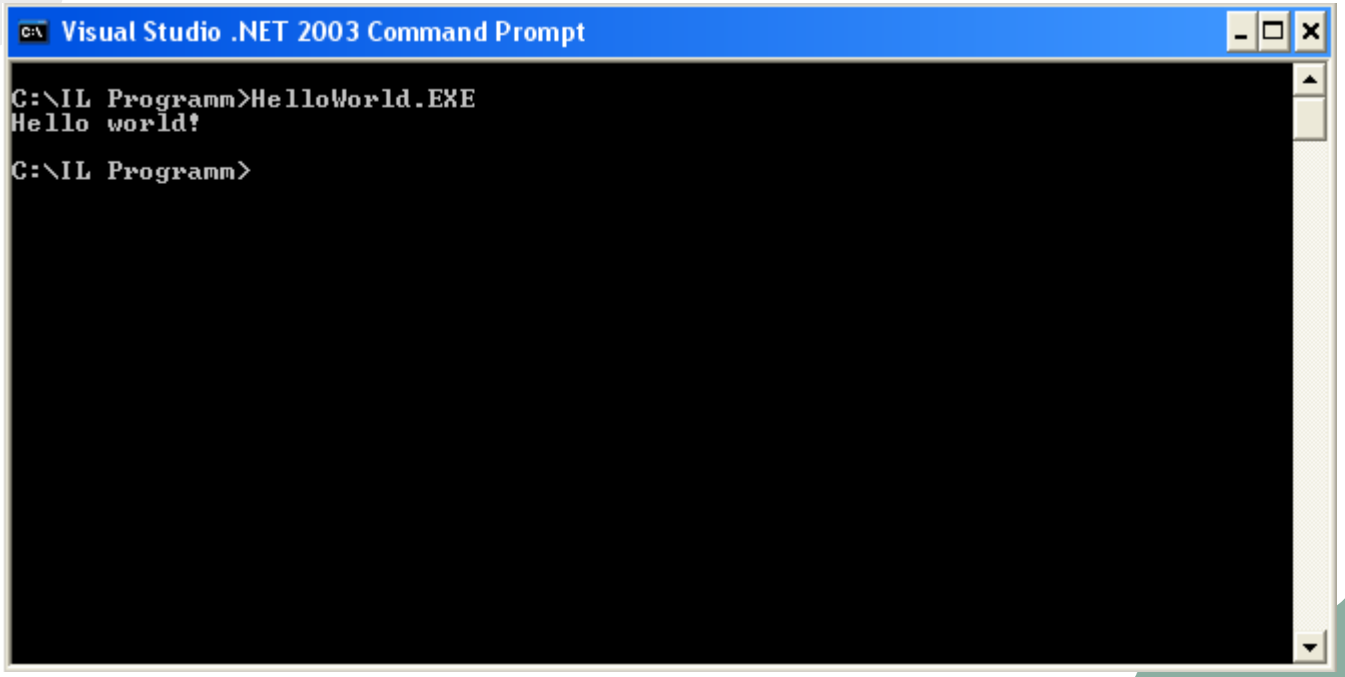
```
.assembly extern mscorlib {}
.assembly HelloWorld {}
.method static private void myMain() cil managed
{
      .entrypoint
      .maxstack 1
      ldstr "Hello world!"
      call void [mscorlib]System.Console::WriteLine(
      class System.String)
      ret
}
```



Common Intermediate Language (CIL)

CLI-Beispiel : „HelloWorld“

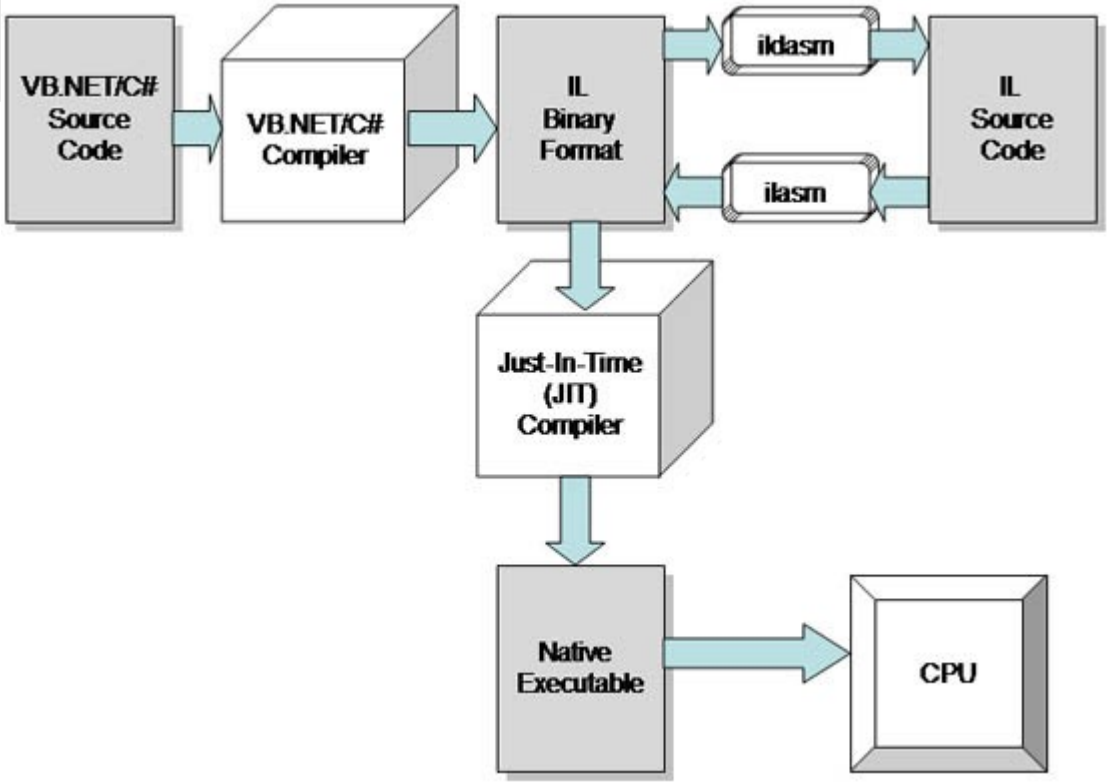
- C:\IL Programm> ilasm HelloWorld.IL
- C:\IL Programm> HelloWorld.EXE



Common Intermediate Language (CIL)

CLI-Beispiel: „HelloWorld.IL“

- C:\IL Programm> ilasm HelloWorld.IL
- C:\IL Programm> HelloWorld.EXE



DAS Beispiel: Hello, .NET-World!

HelloWorld.cs:

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

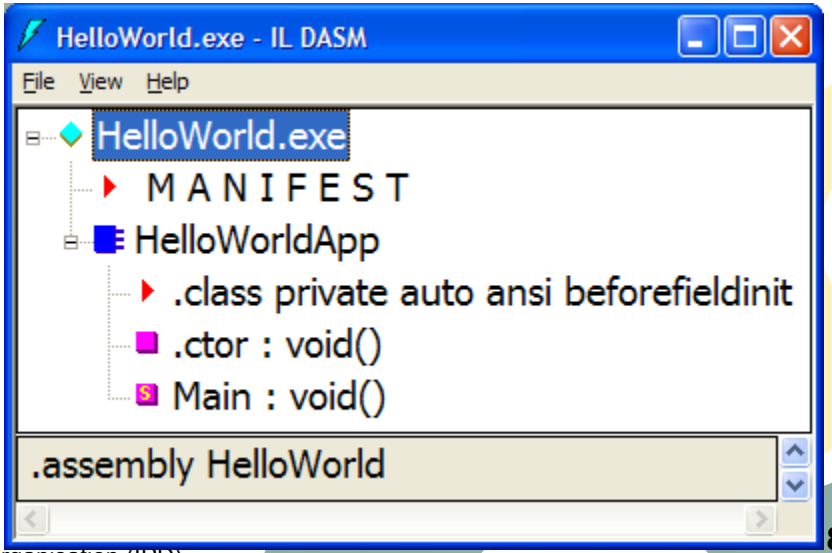
Übersetzen und Assembly erzeugen (mit C#-Compiler):

```
> csc HelloWorld.cs
```

Assembly: HelloWorld.exe (3072 Byte!)

Metadaten und CIL-Code betrachten (mit IL-Disassembler):

```
> ildasm HelloWorld.exe
```



Common Intermediate Language (CIL)

IL-Assembler-Code (= Metadaten + CIL) der Methode Main:

```
.method private hidebysig static void  Main() cil managed {
    .entrypoint
    .maxstack  1
    ldstr "Hello, .NET-World!"
    call void [mscorlib]System.Console::WriteLine(string)
    ret
}
```

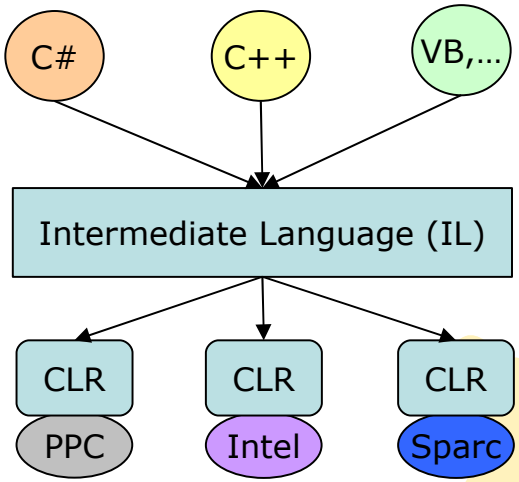
Manifest von HelloWorld.exe:

```
.assembly extern mscorlib {
    .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
    .ver 1:0:3300:0
}
.assembly HelloWorld {
    .hash algorithm 0x00008004
    .ver 0:0:0:0
}
.module HelloWorld.exe
```

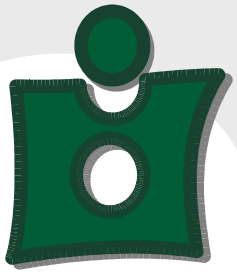
Common Type System (CTS)

Warum CTS?

- Plattformunabhängigkeit und Sprachunabhängigkeit
- **Mehrere Sprachen Microsoft bietet an:**
  - C#
  - VisualBasic
  - Managed C++
  - JScript
- **Von Dritten werden u.a. bereitgestellt**:
  - APL
  - COBOL
  - Eiffel
  - Fortran
  - Haskell
  - Java
  - ML
  - Oberon
  - Pascal
  - Perl
  - Python
  - Scheme
  - Smalltalk



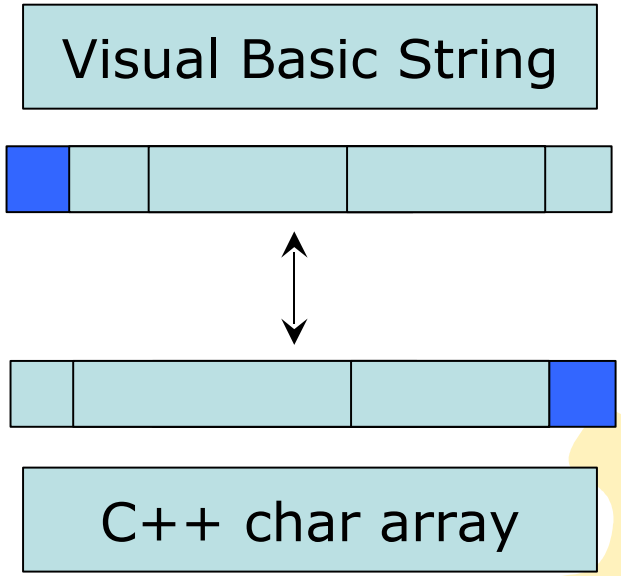
- **Alle diese Sprachen sollen miteinander arbeiten können!**



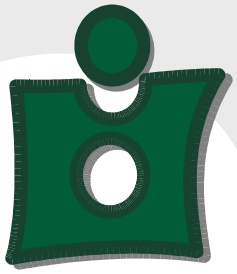
Common Type System (CTS)

Das Problem der Typeninkompatibilität !

- VB (nicht VB.NET!) merkt sich die Länge des Strings selbst in der Laufzeitumgebung.
- C++: Null-Zeichen gibt das ende eines Strings an.
- **Problem**: Ruft VB eine C++ Funktion auf, muss das String-Format immer angepasst werden.

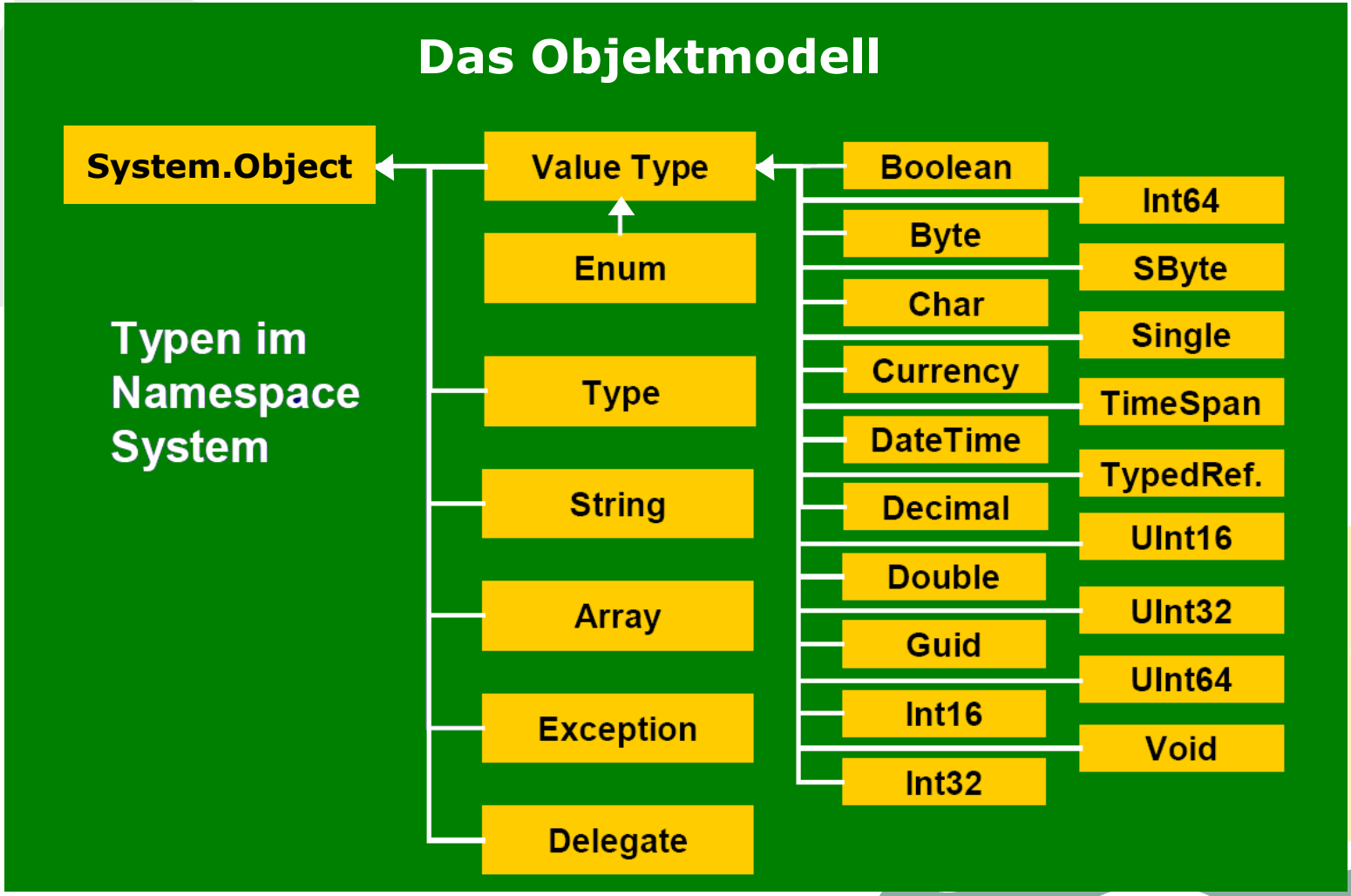


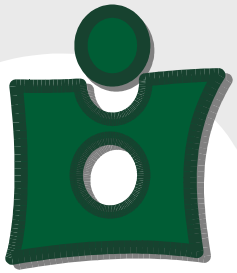
- **Lösung**: Ein gemeinsames Typsystem ➔ **CTS (CLR Type)**



Common Type System (CTS)

CLR Type: Alles ist ein Objekt

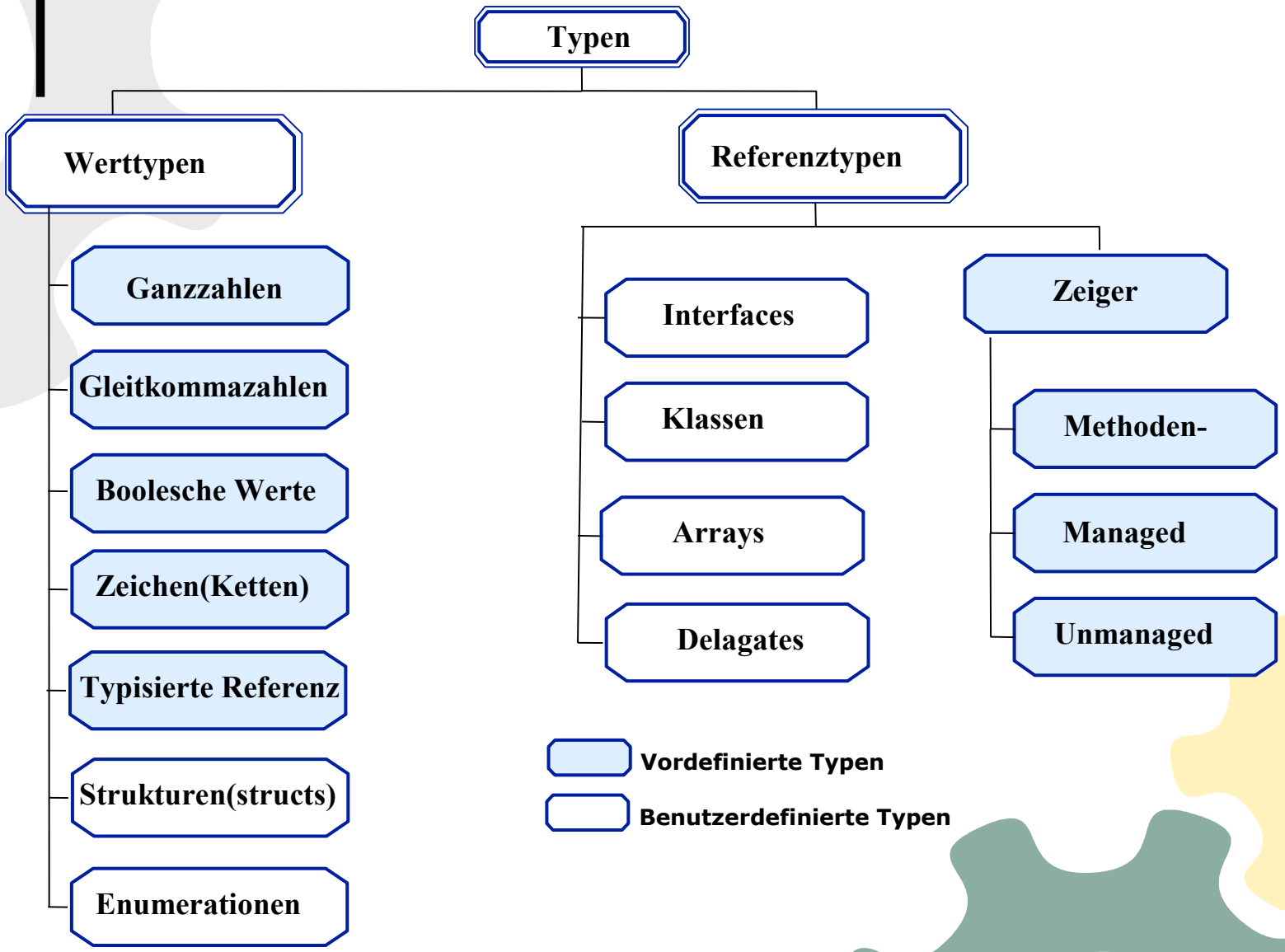


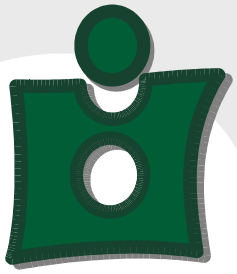


Common Type System (CTS)

- CTS Handelt sich mit zwei Entitätsarten: Objekten und Werten
  - Objekte sind Self-Typing und haben eindeutige Identität
- Common Type System (CTS) unterstützt die **objekt-orientierte Programmierung (OOP)** Sprachen so gut wie die **funktionale und prozedurale Programmierung** Sprachen
- Unterstützt Mehrfache Schnittstellenvererbung (Subtyping) und einfache Vererbung (Subclassing)
- zwei grundsätzlich verschiedene Arten:
  - **Wertetypen und Referenztypen**
- **Wertetypen** – Repräsentieren einen Wert und deren Objekte sind direkt an der von der Variablen bezeichneten Stelle im Speicher.
- **Referenztypen** – Als Verweis auf ein Objekt am Halde realisiert sind.

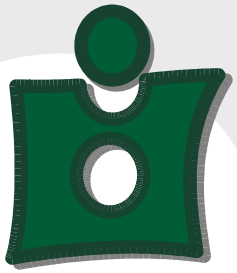
Common Type System (CTS)





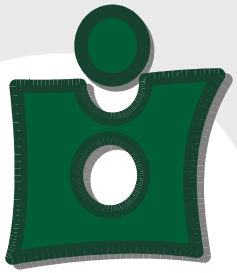
Common Type System (CTS)-Wertetypen

- **Vordefinierte (Built-in) Werttypen**
  - Boolean
  - Character - Unicode
  - 8, 16, 32, 64 bit signed and unsigned integers
  - 32, 64 floating point numbers
  - Maschinenabhängige Typen (native int, native unsigned int)
- **Benutzerdefinierte Werttypen**
  - Structures (C#)
    - struct Point {int x, int y}
    - System.ValueType
  - Enumerations (C#)
    - enum Month {January = 1, …}
    - System.Enum
  - Werttypen sind sealed (keine Erweiterungen).
- sind in IL-Assembler mit Schlüsselwort “valuetype” bezeichnet.



Common Type System (CTS)- Referenztypen

- **Vordefinierte Referenztypen**
  - Object Types
    - System.Object (Base Class, built-in)
    - System.String (Built-in)
  - Pointer Types
    - Managed , Unmanaged und Methoden (Function)
- **Benutzerdefinierte Referenztypen(C#)**
  - Class
    - class Point: IPoint, IChanged
    - System.Object
  - Interface
    - public interface IPoint
  - Delegate
    - public delegate void ADelegate();
    - System.MulticastDelegate
  - Array
    - int [ ] a;
    - System.Arrays
- sind in IL-Assembler mit Schlüsselwort “class” bezeichnet.



Common Type System (CTS) Werttypen vs. Referenztypen

Werttypen

- Nicht auf dem Halde (Heap) sondern an „Ort und Stelle“: entweder am Methodenkeller (lokalen Variablen) oder direkt im Objekt (ein Feld eines Objekts)
- Schneller Zugriff aber größeren Speicherplatzverbrauch an Methodenkeller
- Eignen sich zur kleine Strukturen und Zwischenergebnissen
- Keine Belastung für Garbage Collctor

Referenztypen

- Immer auf der Halde
- 4 oder 8 Byte für den Zeiger zum Objekt

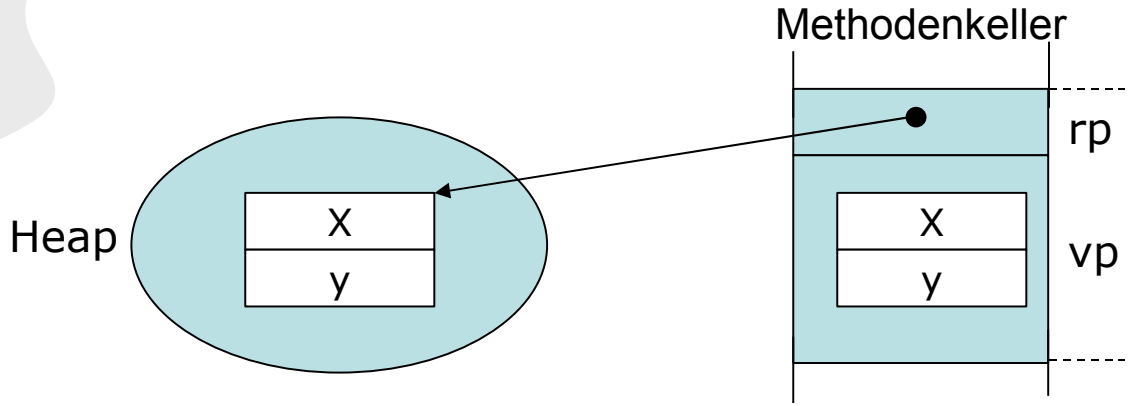
Referenztyp	**Werttyp**

.class **RPoint** extends **System.Object** {
.field int32 x
.field int32 y
}
	.class **RPoint** extends **System.ValueType** {
	.field int32 x
	.field int32 y
	}

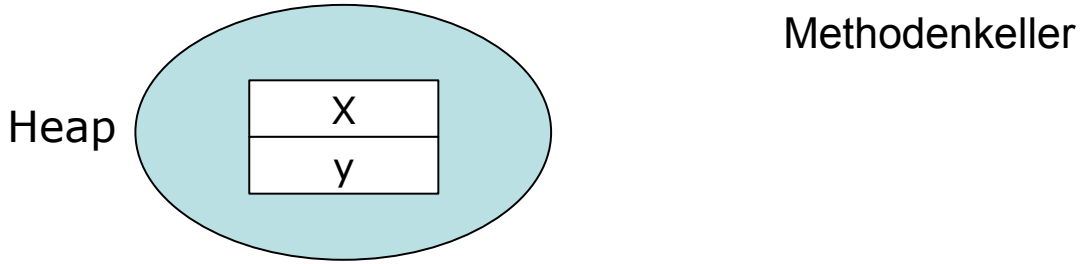
Common Type System (CTS) Werttypen vs. Referenztypen

- Erzeugen zwei Lokalen Variabeln:

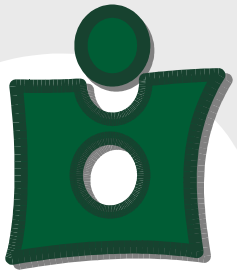
```
.locals (class RPoint rp, Valuetype VPoint vp)
```



- Nach Verlassen der Methode:



- Wenn GC aktiviert wird, wird das Objekt auf der Halde entfernt



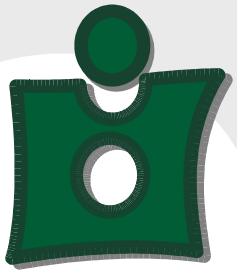
Common Type System (CTS) Vordefinierte Typen

Objektzeiger (Object)

- Verweist auf Objekte von Klassen oder Arrays als Ganzes auf dem Halde (deren Anfänge im Speicher).
- Anlegen mit der Anweisungen: newobj oder newarr
- Als Methodenargument und –ergebnisse, als Objekt- und Klassenfelder, als lokalen Variablen und Array-Elemente
- Unter Kontrolle der Speicherverwaltung und typsicher
- Keine Arithmetische Operationen erlaubt
- Beispiel:

```
.filed valuetype Point p
.field Class Node[ ] nodeArr

.method valuetype Point m ( class Node n) {
.locals (valuetype Point V_0, class Node V_1)
}
```

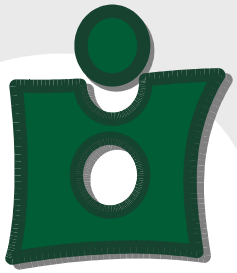


Common Type System (CTS) Vordefinierte Typen

Verwaltete Zeiger (Managed Pointers)

- Verweist ins Innere von Objekten z. B. auf eine Instanzvariable eines Objektes oder ein Element eines Arrays (was mit Objektzeiger verboten ist)
- Typisiert und typsicher: Informationen über den referenzierten Typ sind gegeben, die zur Laufzeit überprüft werden können
- Unter der Kontrolle der Speicherverwaltung: Ihre Werte werden angepasst falls sich die Position der Zielobjekte im Speicher ändert.
- Als Methodenargument und lokalen Variablen aber nicht als Objekt- oder Klassenfelder oder Array-Elemente
- Durch Anhängen des &-Zeichens an die Bezeichnung des Typs
- Beispiel:

```
.methpd instance void int32 RefParMethod( int32& i, valuetype Point& p,
    class Node& n)  { }
```

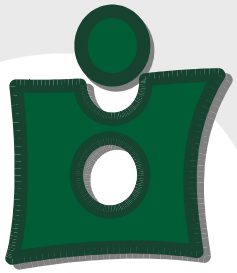


Common Type System (CTS) Vordefinierte Typen

Unverwaltete Zeiger (Unmanaged Pointers)

- Typ-unsicher und nicht verifizierbar (wie normale Zeiger in C, C++: eine Adresse im Speicher)
- Keine Speicherverwaltung
- Dürfen nicht auf Klassen und Arrays am Heap verweisen. (Keine Beeinträchtigung für GC)
- Im CLR wird als „**natural unsigned int**“ oder durch Anhängen des Zeichens * an eine Typbezeichnung definiert
- Beispiel (in C# mit Schlüsselwort **unsafe**):

```
unsafe {
        Int x = 10;
        Int* pX = &x;
}
```



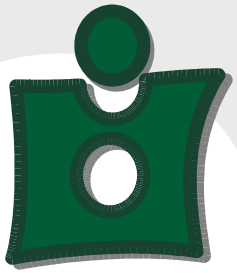
Common Type System (CTS) Vordefinierte Typen

Transiente Zeiger

- Intern für die CLR und unsichtbar für die Benutzer
- Zwischenstufe zwischen verwalteten Zeigern und unverwalteten Zeigern
- Wird überall von CLR verwendet, wo eine der beiden Zeigerarten (verwaltete oder unverwaltete) erwartet.
- z. B. wird durch dieser Instruktionen ein Transiente Zeiger erzeugt: ldloca (Liefert die Adresse des lokalen Variablen)
- **Methodenzeiger**
- Durch das Schlüsselwort **method**, den Ergebnistyp, den durch * ersetzten Methodennamen und die Argumenttypen gekennzeichnet:
- Beispiel:

```
.field Method int32 * (int32&) intMeth
```

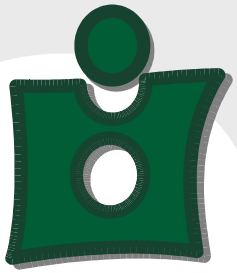
  - Ein Feld „inMeth“ als Zeiger auf Methoden, die einen int32-Wert liefern und einen Referenzparameter vom Typ int32 erhalten.



Common Type System (CTS) Vordefinierte Typen

Maschinenabhängige Typen (Wortlänge)

- Verschiebung der Größe der gespeicherten Werte von der Übersetzungszeit in die Laufzeit
- Alle generische Typen: Native-Typen, object, &, *
- Die Optimale Größe für den Jeweiligen Prozessoren wird in Laufzeit durch CLR verwendet (die Wortlänge wird an die Zielplatform angepasst):
    - native int auf Intel-Pentium-Prozessor ➔ 32 Bit
    - native int auf IA64-Prozessor 64 Bit ➔ 64 Bit



Common Type System (CTS) Benutzerdefinierbare Werttypen

Struktur (Struct)

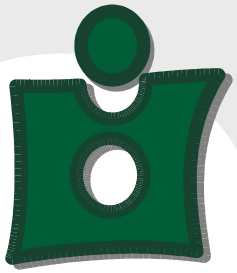
- Attribut **auto**: die Reihenfolge der Datenfelder (Objektlayout) wird optimal für die Zielplatform durch die CLR festgelegt
- Beispiel:

```
.class auto sealed Person extends System.ValueType {
.field string name
.field int32 age
}
```

- Attribut **sequential**: Felder in einer bestimmte Reihenfolge anzuordnen

```
.class sequential sealed TwoNums extends System.ValueType {
.field float32 f
.field  int32 i
}
```

- Keine Attribut: Deklarationsreihenfolge wird angenommen



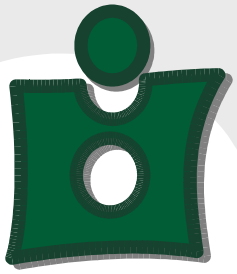
Common Type System (CTS) Benutzerdefinierbare Werttypen

Vereinigung (Union)

- Attribut **explicit**: Auf Keinen fall das Speichelayout des Objekts an CLR überlassen und die Angabe der Speicherposition der Felder zwingend vorschreiben .
- Beispiel:

```
.class explict sealed IntFloat extends System.ValueType {
.field [0] float32 f
.field [0] int32 i
}
```

- Mit Attribut **explicit** ein Überlappen der Speicherbereiche verschiedener Felder ist erlaubt aber mit **sequential** nicht.

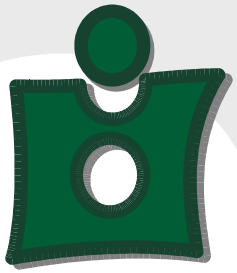


Common Type System (CTS) Benutzerdefinierbare Werttypen

Enumeration

- Erweitert System.**Enum**, was direkt von System.Valuetype abgeleitet ist
- Typbestimmung der Konstanten und die Größe der Objekte des Enumerationstyp ist möglich
- Bespiel:

```
.class sealed Color extends System.Enum {
.field static valuetype Color red = int32(0x00000000)
.field static valuetype Color green = int32(0x00000001)
.field static valuetype Color blue = int32(0x00000002)
}
```

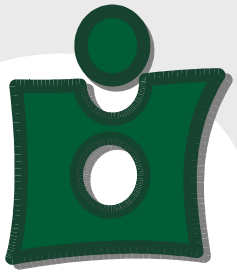


Common Type System (CTS) Benutzerdefinierbare Referenztypen

Klassen

- Erweitert System.**Object**
- Fassen Felder und Methoden zusammen
- Jede Klasse muss einen Konstruktor haben (.ctor())
- Bespiel:
- C#: class Bar { }
- IL:

```
.class Bar extends System.Object {
    .method instance void .ctor () {
      Ldarg.0 // this-Zeiger (Argument 0) auf Stack laden
      Call instance void System.Object::.ctor
      Ret
      }
}
```

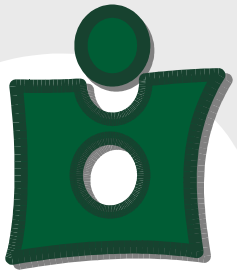


Common Type System (CTS) Benutzerdefinierbare Referenztypen

Klassen

- **Erzeugung eines Objektes**
  - Mit Befehl **newobj** z. B.: **newobj** instance void Bar::.ctor()
  - Speicherplatzreservierung und Initialisierung untrennbar in eine Anweisung. (Einfacher für die CLR-Verifizierer)
  - Eine Objektzeiger auf dem Stack zur Verweisung der neue Objekt
- **Methoden**
  - Attribut **instance**: wird bei jedem Aufruf als erstes Argument ein Zeiger auf einem Objekt am Heap übergeben. (Objektmethodenaufruf)
  - Attribut **virtual**:: die Methode kann in abgeleitete Klassen überschrieben werden.

```
.class Bar extends System.Object {
    .method static void foo () {}          // statische Klassenmethode
    .method instance virtual void goo () {} //überschriebbare Objektmethode
    .method instance void hoo () {} //nicht überschriebbare Objektmethode
}
```



Common Type System (CTS) Benutzerdefinierbare Referenztypen

Klassen

- **Methodenaufruf in der CLR:**
  - Früh Aufruf (statisch-Bindung, zur Compilzeit an die Methode gebunden werden) mit call-Befehl:

```
.locals ( class Bar bar)
    call void Bar:: foo()        //statisch
    ldloc.0
    call void Bar:: goo()        //statisch (call-Befehl)
    ldloc.0
    call void Bar:: hoo()        //statisch
```

  - Sapät Aufruf (dynamisch-Bindung, zur Laufzeit) mit callvirt-Befehl:

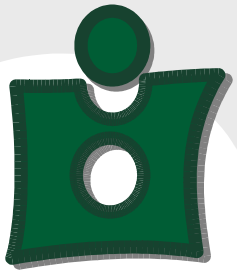
```
    ldloc.0
    callvirt void Bar:: goo()    //dynamisch
    ldloc.0
    callvirt void Bar:: hoo()   //statisch (nicht virtuell)
```

Common Type System (CTS) Benutzerdefinierbare Referenztypen

Interfaces

- Werden mit dem Schlüsselwort **.class** und dem zusätzlichen Attribut **interface** und **abstarct** deklariert
- Dürfen keine
  - Instanzvariablen, nicht überschreibbare Methoden und inneren Typen haben.
- Können
  - Klassenvariablen (nicht möglich in C#), statische Methoden (nicht möglich in C#) und virtuelle Objektmethoden definieren.

```
.class interface abstract ILockable {
.method public abstarct instance virtual void Lock () { }
}
.class Door extends System.Object implements ILockable {
.method publicabstarct instance virtual void Lock () {…}
}
```



Common Type System (CTS) Benutzerdefinierbare Referenztypen

Arrays

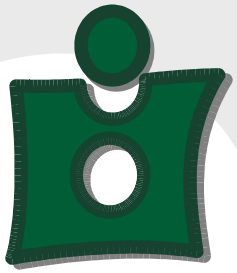
- Arrays werden von System.Array abgeleitet
- **Vektoren**: Index bei 0 beginnt (eindimensionalen Arrays)
  - newarr, ldelem, stelem

C#	CIL
int[] a; a = new int [6]; // eindimensionalen Arrays	.locals (int32[] a) ldc.i4.6 **newarr** System.Int32 stloc a

- **Arrays als Objekte**: Index nicht bei 0 beginnt (mehrdimensionalen Arrays)
  - newobj

C#	CIL

int[,] ab;
 	
ab = new int [2,3]; // zweidimensionalen
//Block-Arrays
	.locals (int32[0,…,0...] ab)
	
	ldc.i4.2
	ldc.i4.3
	newobj instance void int32[0…,0…]::.ctor(int32,int32)
	stloc ab



Common Type System (CTS) Benutzerdefinierbare Referenztypen

Delegates

- Typsichere Variante von Methodeanzeiger
- Methodenzeiger werden in einer von CLR Compilergenierirten Klasse gekapselt

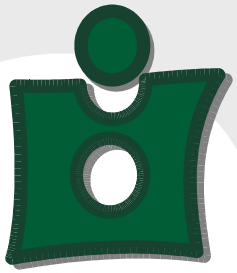
C#

```
Delegate int Adder (int a, int b);
```

CIL

```
.class seald Adder extends System.MulticastDelegate {
.method instance void .ctor(object receiver, native int method) runtime { }
.method virtual instance int32 Invoke(int32 a, int32 b) runtime { }
.method virtual  instance class System.IAsyncResult BeginInvoke
        (int32 a, int32 b,  class System.IAsyncResult acb, object asyncState) runtime { }
.method virtual instance int32 EndInvoke (class System.IAsyncResult result) runtime { }
}
```

- runtime { } : Kein CIL-Code, die CLR wird das Verhalten der Methoden bestimmen.
- Invoke(): zum Aufruf der im Delegate gekapselten Methode
- BeginInvoke() und EndInvoke(): Asynchrone Aufruf der Delegate-Methode



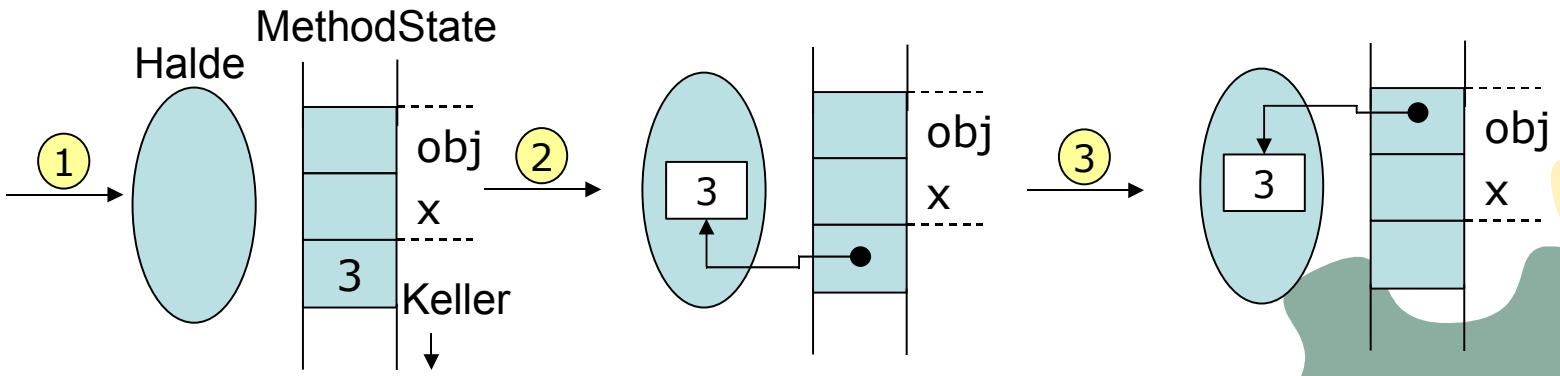
Common Type System (CTS) Boxing und Unboxing

- Instanzen von Wertetypen werden "verpackt“ (boxed) um Objektreferenzen auf Wertetypen zu unterstützen
- Jeder Werttyp hat zwei Repräsentationen:
  - Raw (Roh): das Werttypobjekt entweder am Stack oder direkt in ein Objekt am Heap
  - Boxed (verpackt): Das Werttypobjekt durch ein neues Referenztypobjekt am Heap ersetzt wurde
- CIL-Anweisungen: box und unbox
- Das verpackte Objekt ist ein unabhängiger Clone
- Verpackte Objekte können zurück in Werteinstanzen gewandelt werden (unboxing)
- Wertetypen sind bis zur Umwandlung keine wirklichen Objekte
- System.Object ist Universaltyp

Common Type System (CTS) Boxing und Unboxing

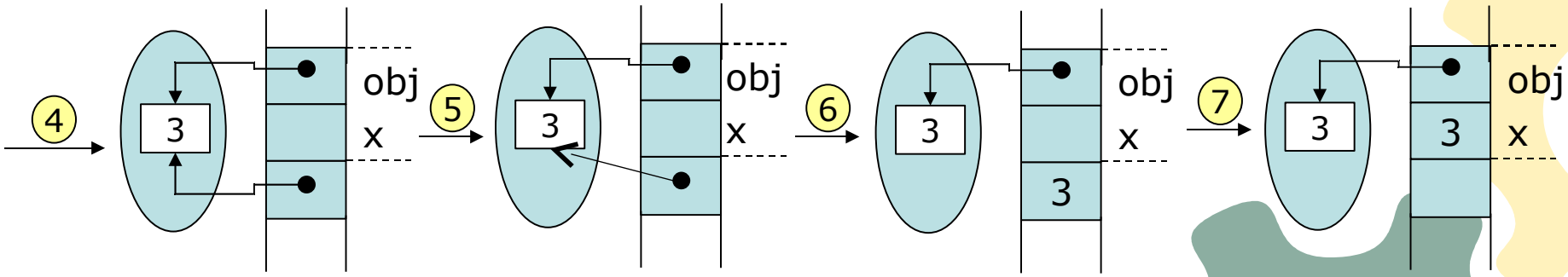
Beispiel: C# CIL

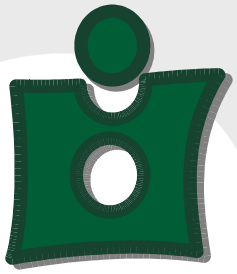
C#	CIL
	.local (object obj, int32 x)
object obj = 3; //Boxing	ldc.i4.3 // 1 **box** System.Int32 // 2 stloc obj // 3
int x = (int) obj; // Unboxing	ldloc obj // 4 **unbox** System.Int32 // 5 ldind.i4 // 6 stloc x // 7



Common Type System (CTS) Boxing und Unboxing

C#	CIL
	.local (object obj, int32 x)
object obj = 3; //Boxing	Ldc.i4.3 // 1 **box** System.Int32 // 2 Stloc obj // 3
int x = (int) obj; // Unboxing	ldloc obj // 4 **unbox** System.Int32 // 5, in verwalteter Zeiger auf Wert im Objekt umwandeln ldind.i4 // 6 stloc x // 7





Common Type System (CTS) Boxing und Unboxing

- **Demo 2: Boxing und Unboxing**

Common Type System (CTS)

Gleichheit und Identität von Objekten

- Zwei Objekte sind gleich, wenn deren Inhalte gleich sind
- Zwei Objekte sind identisch, wenn sie die gleiche Instanz referenzieren
- Gleichheit und Identität können sich über die virtuelle Methode System.Object.Equals definieren:
  - identisch: System.Object.Equals = true z.B. (A,B)
  - gleich: System.Object.Equals.Value = true z.B. (A,C)