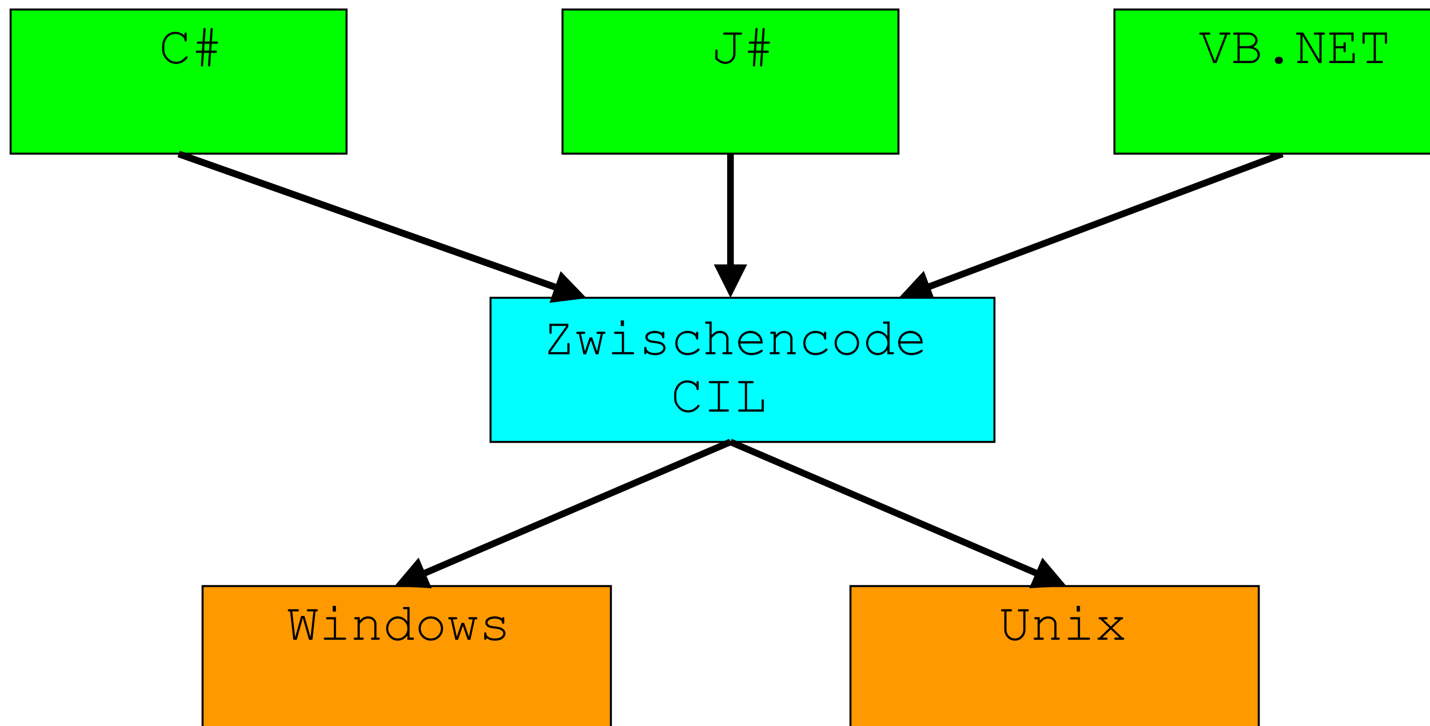
Typsystem

Plattform- und Sprachenunabhängigkeit in .NET



Auf welchen Grundlagen basiert dies?

Typsystem

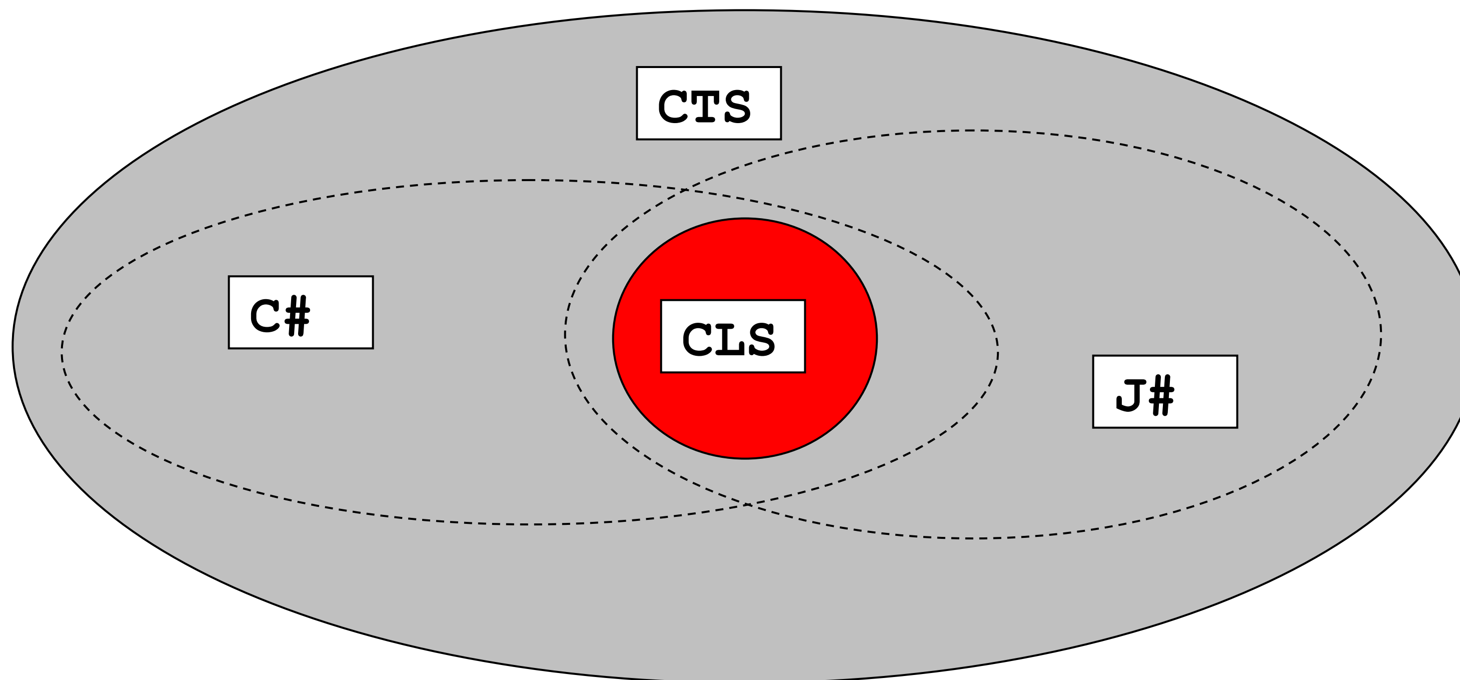
Grundlagen

- CLR Common Language Runtime
  - Laufzeitumgebung, in der alle .NET Programme ausgeführt werden
  - ist eine virtuelle Maschine
  - ist eine Stackmaschine
- CTS Common Type System
  - stellt Datentypen zur Verfügung und legt deren Formate fest
- CLS Common Language Specification
  - minimale Interoperabilitätsanforderung
- CIL Common Intermediate Language
  - Zwischensprache in die alle Programme unabhängig in welcher Sprache sie implementiert wurden übersetzt werden

Typsystem

Common Language Specification

CLS ist der minimale Anforderungskern, den jede Programmiersprache zur Interoperabilität erfüllen muss.



Typsystem

Das allgemeine Typsystem von .NET

- CTS = Common Type System
- Spezifikation, die festlegt, wie Typen zur Laufzeit deklariert, verwendet und verwaltet werden
- Ist Basis für die Interoperabilität der verschiedenen Sprachen
- CLS (Common Language Specification) definiert Regeln für Kommunikation zwischen den Sprachen
- Komponente, die diesen Regeln genügt, heißt CLS-konform
  - Attribut [assembly:CLSCompliant(true)] veranlasst Compiler CLS-Konformität zu überprüfen
- Unterscheidung in:
  - Werttypen und
  - Referenztypen

Das C# Typsystem

Aspekte des C# Typsystems

- Typen
  - Werttypen (von System.valuetype abgeleitet)
    - Einfache Typen (int,float,...)
    - Enumerationen
    - Strukturen
  - Referenztypen (von System.object abgeleitet)
    - Klassen
    - Interfaces
    - Arrays
    - Delegaten

C# ist typsicher (keine ungeprüfte Typumwandlung wie in C++ möglich), wobei in Problemsituationen eine Ausnahme geworfen wird

Das C# Typsystem

Verwendung der Typen

Bei Verwendung eines Typs kann in C# der .NET Framework-Typ oder der C#Typ (übersichtlicher) angegeben werden. Der C#-Typ ist ein Alias für den .NET Framework-Typ. Man kann z.B. statt System.Int32 auch einfach int verwenden.

Das C# Typsystem

C#-Typ	CLS-konform	.NET Framework-Typ	Standardwert
bool	ja	System.Boolean	false
byte	ja	System.Byte	0
sbyte	nein	System.SByte	0
char	ja	System.Char	\0
decimal	ja	System.Decimal	0.0m
double	ja	System.Double	0.0d
float	ja	System.Single	0.0f
int	ja	System.Int32	0
uint	nein	System.UInt32	0
long	ja	System.Int64	0
ulong	nein	Sytem.UInt64	0
object	ja	System.Object	0
short	ja	System.Int16	0
ushort	nein	System.UInt16	0
string	ja	System.String	0

Das C# Typsystem

Werttypen

- in C++ wählbar: Klasseninstanzen auf Heap oder Stack
- in C# werden Werttypen auf dem Stack abgelegt
- schneller zugreifbar (Indirektion fehlt)
- einfache Typen wie int,... sowie Enumeration und struct
- Lebensdauer durch den Gültigkeitsbereich der Variablen begrenzt
- kurze Lebensdauer, nicht von mehreren Clients benutzbar
- keine Belastung des GC => verbesserte Laufzeitleistung
- Bsp.:
```
int i =5; // tatsächliche Instanz des Typs
          // (4 Byte im Stack)
```
- Parameterübergabe in C# standardmäßig als Wert (Kopie erzeugt)
- Methode kann Wert der übergebenen Parameter nicht ändern

Das C# Typsystem

Verweistypen

- als Verweistyp deklarierte Variable enthält Verweis auf den Heap (Adresse der Instanz)
- mehrere Variable können auf dieselbe Stelle im Heap verweisen
- Instanzen werden in C# mit new angelegt
- Bsp.:

```
class A{}
A a= new A();  // a enthält Verweis auf Instanz
               // im Heap
```

- Freigabe der Instanz erfolgt automatisch mit dem GC (falls kein Verweis mehr existiert, Zeitpunkt undefiniert)
- Beim Belegen von Ressourcen bzw. bei Ressourcenknappheit bestehen Eingriffsmöglichkeiten für den Programmierer

Das C# Typsystem

Boxing und Unboxing

Wie können Werttyp-Variable in Methoden verwendet werden, die Verweistyp-Variable erwarten?

```
Bsp:  int i = 5;         // i ist Werttyp-Variable
      object obj = i;    // obj ist Instanz der Klasse
                         // object somit Verweistypvar.
```

Frage: Was passiert hier?

Boxing

- Werttyp-Variable wird in eine Verweisbox verpackt
- erfolgt oft implizit
- Unboxing ist die „Umkehrung“ dieses Vorgangs
- Unboxing ist explizit anzufordern (Typ-Casting)

```
Bsp.: int i = (int) obj;   // möglicher Datenverlust =>
                           // Ausnahme
```

Das C# Typsystem

Boxing

Stack **Heap**

```
int i = 4711;
```

4711

```
object o;
```

null

```
o = i;
```

Referenz → int 4711

Das C# Typsystem

Struktur

- ist einfacher benutzerdefinierter Datentyp
- ähnelt Klasse aber: ist Werttyp
- Syntax: [Attribute] [Zugriffsmodifikatoren] struct Bezeichner [:Interface-Liste]{Struct-Elemente}
- Strukturen unterstützen <u>keine Vererbung</u> oder Destruktoren
- Basisklasse: System.ValueType (von Object abgeleitet)
- nicht geeignet für Collections da Referenzen erwartet werden
- sind implizit sealed
- falls Konstruktor definiert, müssen alle Felder initialisiert werden
- direkte Initialisierung im struct (int i = 15;) nicht möglich

Das C# Typsystem

Struktur erzeugen

- kann mit new wie bei Klassen erfolgen
- Bsp.: struct mystruct{..}

  mystruct ms = new mystruct(10);
- falls kein eigener Konstruktor definiert dann Standardkonstruktor verfügbar (Initialisierung mit 0)
- Instanzen von Strukturen werden auf dem Stack erzeugt
- Instanziierung ohne new möglich, aber Initialisierung erforderlich

  Bsp.: mystruct ms1; ms1.feld = ...; // für alle Felder
- parameterloser Konstruktor darf nicht definiert werden
- Übergabe an Funktionen als Wert
- Bsp.: fn(ms); // Ändern von ms nicht möglich,

  // da Kopie von ms übergeben

Das C# Typsystem

Verwendung eigener Werttypen

- Werttypen existieren in vielen Programmiersprachen (Bsp.: C++, Fortran, C#, Basic, Java, ...)
- Definieren in der Regel Basistypen (int, double,...)
- Vorteil: Keine Indirektion beim Zugriff => performanter
- Verwende eigene Werttypen ausschliesslich für Typen, die den Basistypen entsprechen
  (Bsp.: Punkt P(x,y,z) in der Geometrie)
- Nachteile eigener Werttypen:
  - Probleme beim Casting auf Referenztypen (Fehlerquelle)
  - Paramterübergabe erfolgt in C# durch Kopieren
    d. h. falls Werttyp groß => Speicherbelastung
- Vorteil: Werttypen belasten Garbagecollector nicht

Das C# Typsystem

Interface

- Interface ist Vertrag über: Garantie der Implementierung von Methoden, Eigenschaften, Events und Indexer
- Referenztyp (bei Struktur beachten!)
- Alternative zur abstrakten Klasse
- Schlüsselwort: interface (deklariert Referenztyp)
- Falls eine Klasse ein Interface hat (implementiert), so muss sie "alles" implementieren
- Elemente des Interface sind standardmässig public
- Klasse kann mehrere Interfaces implementieren
- Modellieren im Hinblick auf Schnittstellen (Interfaces) ist wichtiges Engineering-Prinzip
- Beispiel class A : Interface_I {Klassenkörper} (stellt implementiert-Beziehung dar)

Das C# Typsystem

Implizite Variablendeklaration

```
- var   Deklaration einer Variablen
        ohne explizite Angabe des Typs

- Compiler ermittelt den Typ
- Bsp.:  int i = 5;
         var j = i;  // Compiler ermittelt int

- Variablen sind statisch typisiert und nicht dynamisch
- Bsp.:   var x = 11;
          x = “Test“; // Fehler
- Nachteil: Entwickler erkennt Typ nicht direkt
- var ist sinnvoll und notwendig
  z. B. anonyme Typen in LINQ
```