

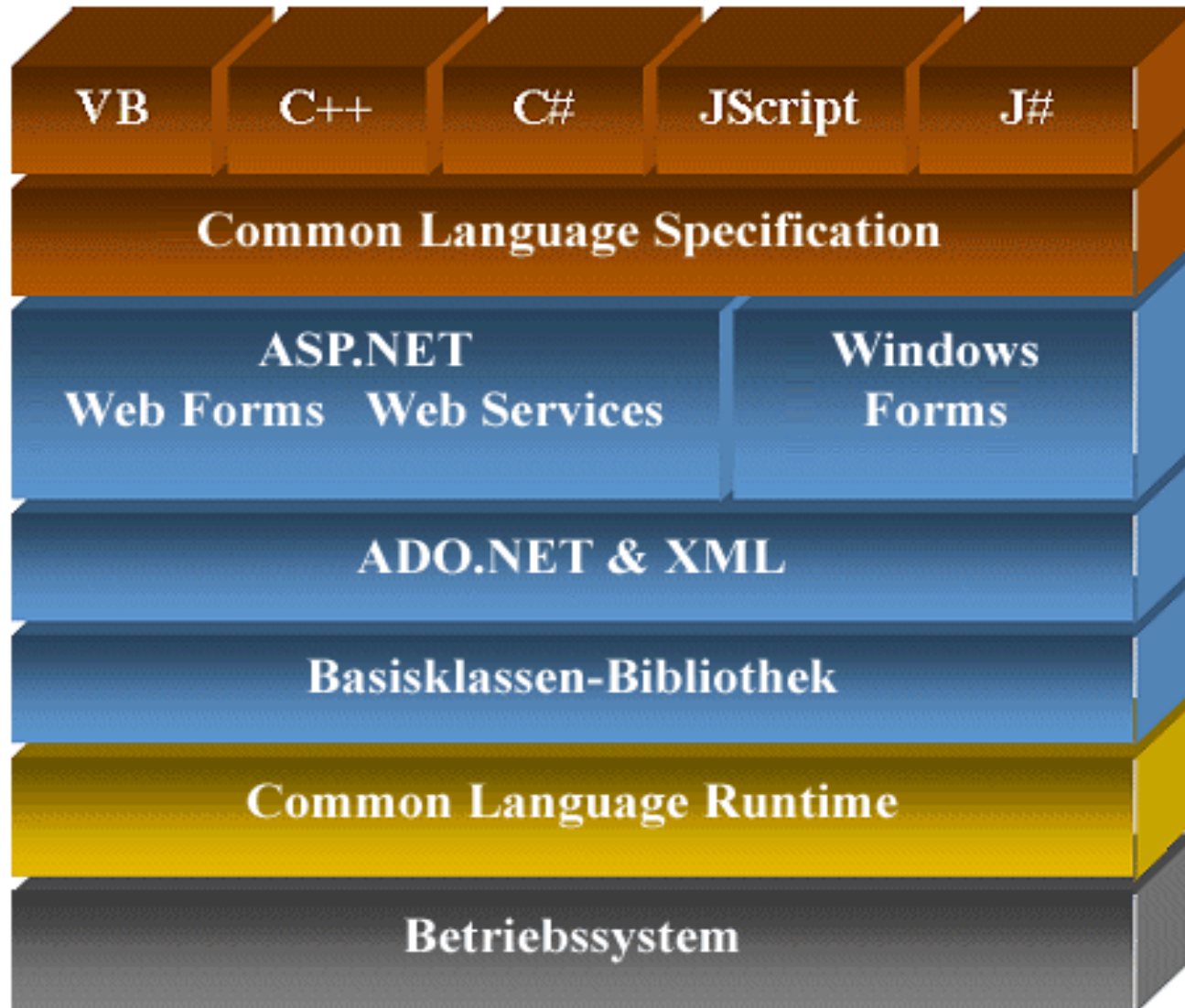
Einführung in die Programmiersprache C#



C# Übersicht

- Einführung in das .NET Framework
- Konzepte / Architektur
- Einführung in C#, speziell Unterschiede zu Java

.NET: Framework, Sprachen und Werkzeuge



.NET Framework

- **Common Language Runtime (CLR)** mit einer für alle .NET gemeinsamen Common Intermediate Language (CIL). Z.B. ist der Garbage Collector in der CLR implementiert
- **Common Language Specification (CLS)** und Common Type System (CTS). Alle .NET-Sprachen basieren auf gemeinsamen Basis-Typen
- **Umfangreiche (Klassen-) Bibliothek** z.B. Für grafische Oberfläche, Web-Anwendungen, Datenbank, Sockets, XML, Multi-Threading, Kryptographie usw.

ein Sprachenmix

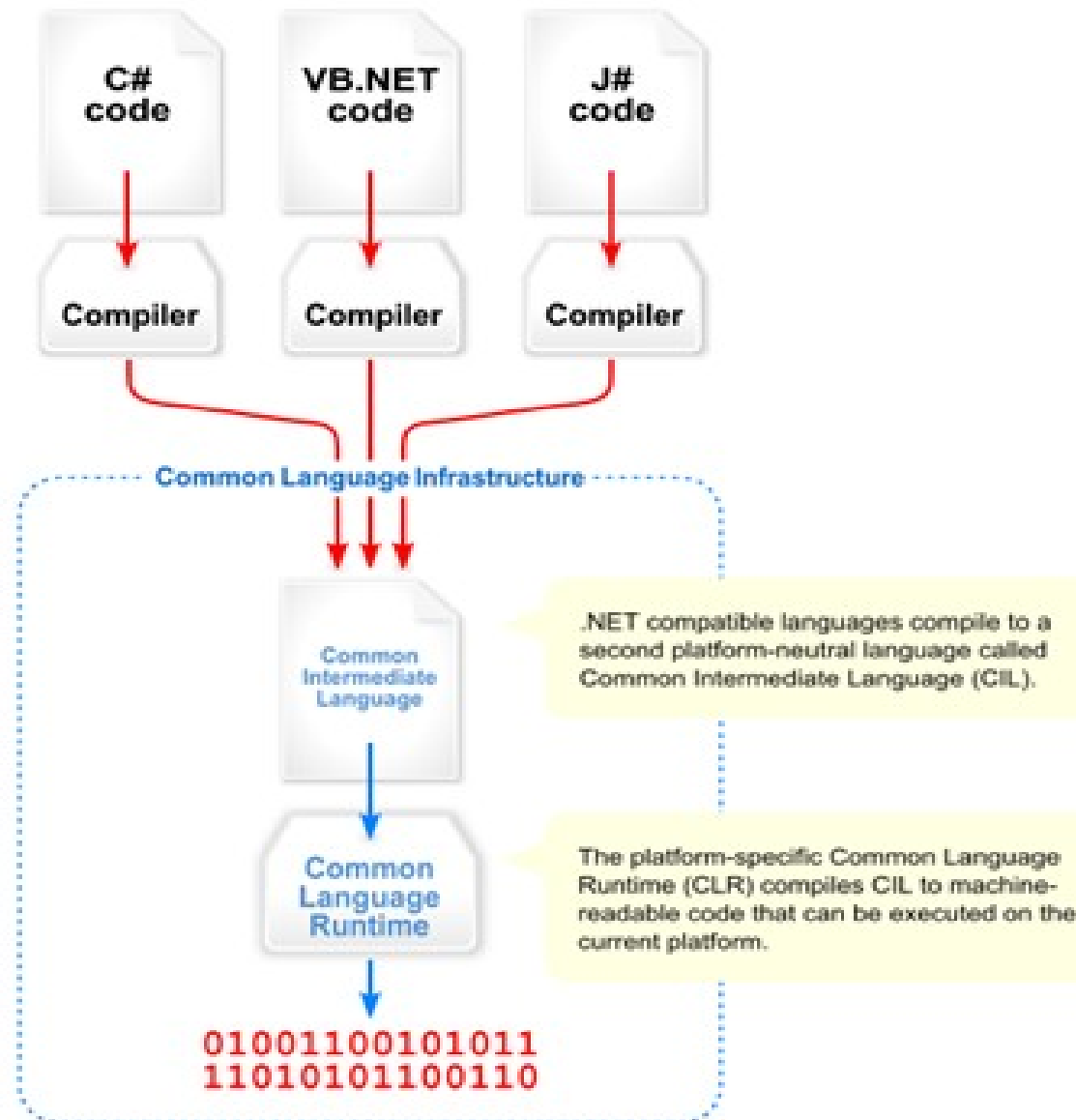
- Eine .NET-Anwendung kann in unterschiedlichen Sprachen geschrieben werden (z. B. C#, J#, C++/CLI, Visual Basic .NET)
- Eine Klasse, die in C# geschrieben ist, kann von einer Klasse in Visual Basic 2005 beerbt werden. Beide Klassen können Daten miteinander austauschen und Ausnahmen weiterreichen.
- Es gibt unter .NET keine „bevorzugte“ Programmiersprache. Vorteil: Jeder kann in der Sprache seiner Wahl programmieren.
- Die Klassenbibliothek, das Typsystem und die Laufzeitumgebung ist für alle .NET Sprachen gleich.

Common Intermediate Language

- Die CIL ist der „Befehlssatz“ der Virtual Machine von .NET
 - D.h. .NET-Anwendungen sind plattform-unabhängig in CIL geschrieben
 - Der CIL-Code sichert die Kompatibilität zwischen den verschiedenen .NET Programmiersprachen
 - CIL ist eine „objektorientierte Assemblersprache“

```
.method public static void Main() cil managed
{
    .entrypoint
    .maxstack 1
    ldstr "Hallo Welt!"
    call void [mscorlib]System.Console::WriteLine(string)
    ret
}
```

Common Language Runtime



Einfaches Beispiel

- C# Code: Person.cs

```
1  class Person
2  {
3      private double gehalt;
4      public double getGehalt() {
5          return gehalt;
6      }
7
8      public void setGehalt(int g) {
9          gehalt = g;
10     }
11
12     public void gehaltErhoehen() {
13         gehalt = gehalt + 100;
14     }
15 }
```

Person
gehalt: double
gehaltErhoehen() setGehalt getGehalt

C# vs. Java

- Streng typisierte objekt-orientierte Programmiersprache
- wird übersetzt in Intermediate Language (IL):
ähnlich Java-Bytecode
- wird ausgeführt von CLR – ähnlich JVM
- Anforderungen:
 - Architekturunabhängigkeit
 - Sprachunabhängigkeit

C# vs. Java

- keine Header-Dateien
- Mehrfachvererbung von Schnittstellen (nicht von Implementierungen)
- keine globalen Funktionen oder Konstanten (alles in Klassen)
- Arrays und Strings mit festen Längen und Zugriffskontrolle
- Alle Variablen müssen vor der ersten Verwendung initialisiert werden
- alle Objekte erben von Object-Klasse
- Objekte werden auf dem Heap erzeugt (mit dem Schlüsselwort new)
- Garbage Collector, Reflection
- Thread Unterstützung, Synchronisation
- Generics

Erste Schritte

- Hello World

```
1 using System;
2
3 public class Hello {
4     public static void Main() {
5         Console.WriteLine("Hello World");
6     }
7 }
```

- alle Klassen der .NET-Architektur im System-
Namespace (using -Anweisung)
- Main-Methode als Einstiegspunkt:

public static void Main(string[] args) { ... }

BankLibrary

```
1 namespace BankLibrary {  
2     public class Bank {  
3         public static int deposit(Account account,  
4             int amount){  
5             return account.deposit(amount);  
6         }  
7     }  
8  
9     public class Account {  
10         private int balance = 0;  
11  
12         public int deposit(int amount) {  
13             balance += amount; return balance;  
14         }  
15  
16         public int withdraw(int amount) {  
17             balance -= amount; return balance;  
18         }  
19     }  
20 }
```

> mcs -target:library Bank.cs

BankClient

```
1 using System;
2
3 namespace Customer {
4     class Customer {
5         public static void Main(string[] args) {
6             BankLibrary.Account account =
7                 new BankLibrary.Account();
8             Console.WriteLine("deposit 3 -> account = {0}",
9                 BankLibrary.Bank.deposit(account, 3));
10            Console.WriteLine("deposit 5 -> account = {0}",
11                BankLibrary.Bank.deposit(account, 5));
12        }
13    }
14 }
```

> mcs -r:Bank.dll Customer.cs

> mono Customer.exe

Kommentare

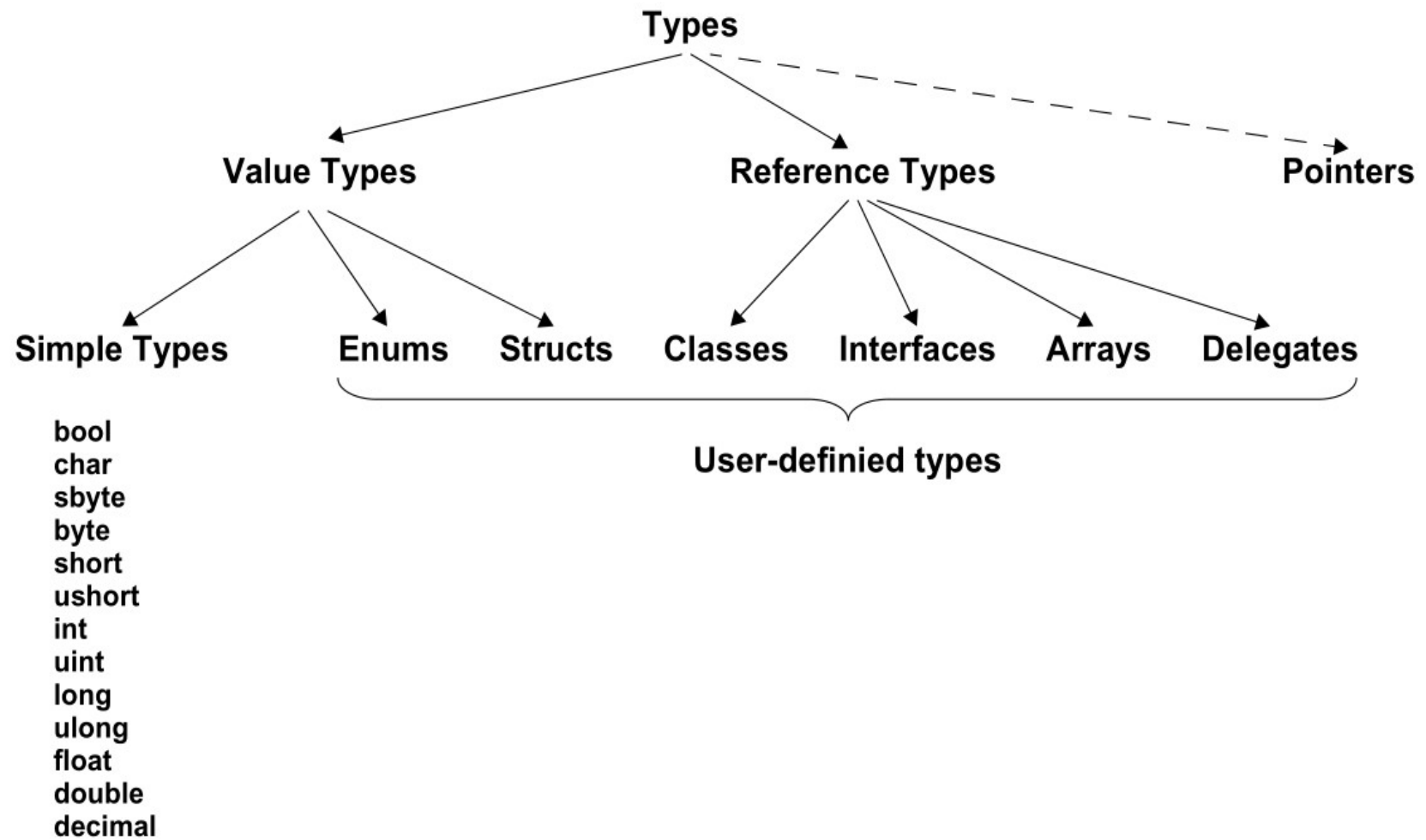
- Kommentierter Bereich

```
/* Dies ist ein Kommentar,  
   der über mehrere  
   Zeilen verläuft */
```

- Zeilenkommentar

```
int x = 1; // Anfangswert  
// ab jetzt folgen Initialisierungen
```

Typesystem



Primitive Typen

- byte , short , int , long , float , double , bool , char (Unicode)

wie in Java

- vorzeichenlose Typen: byte , ushort , uint , ulong
- decimal : 128 Bit Zahl
- Konstanten: wie in C/C++

const int i = 3;

Typen

- Enum

Liste von Konstanten mit Namen

```
1 public enum Farbe { Gelb=1, Rot=2, Gruen=4 };  
2  
3 Farbe farbe = Farbe.Rot | Farbe.Gelb;  
4 if ((farbe & Farbe.Rot) != 0) { ... }
```

- Array

```
1 int[] array = new int[3];  
2 int[] array = new int[] { 1, 2, 3 };  
3 int[] array = { 1, 2, 3 };  
4 int len = array.Length  
5  
6 int[][] 2D_array = new int[2][];  
7 a[0] = new int[3]; a[1] = new int[4];
```

Strings

- Objekte der Klasse string sind konstante, invariante Objekte
 - bei Modifikationen wird neues string-Objekt als Rückgabewert geliefert
 - Aneinanderhängen mit "+"
 - Indizierung: s[i], Länge der Zeichenkette: s.Length
- Vergleich von string-Objekten

```
1 string s1 = "Hello";
2 string s2 = string.Copy(s1);
3
4 if (s1 == s2) { ... }           // true (1)
5 if ((Object)s1 == (Object)s2) { ... } // false (2)
6 if (s1.Equals(s2)) { ... }      // true (3)
```

Strings

- Strings parsen

```
1 char[] trennzeichen = {' ', ',', ':'};  
2 // oder:  
3 string s_trennzeichen = " ,:";  
4 char[] trennzeichen = s_trennzeichen.ToCharArray();  
5  
6 string test = "Vorname,Nachname Strasse:PLZ,Ort";  
7 string[] parts = test.Split(trennzeichen);
```

- Ein-/Ausgabe (Console)

```
1 using System;  
2  
3 Console.Write("kein Newline am Ende");  
4 Console.WriteLine("diesmal mit Newline");  
5 Console.WriteLine("Heute ist der {0}. {1}", 22, "Januar");  
6  
7 string input = Console.ReadLine();
```

Strukturen

- C# unterstützt Strukturen:
 - ähnlich C/C++
- Strukturen werden in C# auf dem Stack angelegt (value type)
 - Klassen (class) werden immer auf dem Heap erzeugt
- Keine Vererbung möglich, Strukturen können aber Interfaces implementieren

```
1 public struct Account {  
2     public int balance;  
3     public Account(int amount) { balance = amount; }  
4     public void Withdraw(int amount) { balance -= amount; }  
5 }
```

Boxing

- Value types (primitive Type, Strukturen, Enums) können in ein Objekt (Boxing) und wieder zurück gewandelt werden (Unboxing)
- Nachteile:
 - Wrapper-Objekt muss (auf dem Heap) erzeugt werden
 - Man erkennt nicht auf Anhieb, dass das Verfahren teuer ist

```
1 Object obj = 333;           // boxing (explizit)
2 int i = (int) obj;          // unboxing
3
4 Stack stack = new Stack();   // Stack enthält Objekte
5 stack.Push(i);              // boxing (implizit)
6 int j = (int) stack.Pop();   // unboxing
```

Switch-Anweisung

- Kontrollfluss muss explizit festgelegt werden
 - break (oder return, goto, throw) muss am Ende jeder case-Anweisung stehen
 - falls kein case zutrifft: default-Label
- als Switch-Typ ist auch ein String erlaubt:

```
1 switch(name) {  
2     case "Zaphod Beeblebrox":  
3         Console.WriteLine( "Hello Zaphod" );  
4     break;  
5     case "Ford Prefect":  
6         Console.WriteLine( "Hi" );  
7     break;  
8 }
```

foreach-Anweisung

- foreach-Anweisung arbeitet auf allen Objekten, die das Interface `System.Collections.IEnumerable` implementieren.
- Auch Arrays implementieren dieses Interface

```
1 foreach (Object o in collection) { ... }  
2  
3 foreach (int i in array) { ... }
```

Assemblies, Namespaces

- Namensräume (namespaces) wie in Java:
Trennung mit “ . ”
 - “syntactic sugar” für lange Klassennamen
 - Namensräume importieren mit using-Schlüsselwort
- Eine Assembly besteht aus mehreren Dateien (einem Projekt), die zu einer .exe- (Executable) oder .dll-Datei (Bibliothek) kompiliert werden
 - definieren einen eigenen Namensraum
 - verschiedene Versionen einer Assembly können parallel existieren

Zugriffslevel

- private (Zugriff nur innerhalb der Klasse, wie in Java)
- internal (Zugriff innerhalb der Assembly)
- protected (Zugriff innerhalb der Klasse und abgeleiteter Klassen)
- internal protected (wie protected, zusätzlich im Assembly)
- public (Zugriff immer erlaubt)

Methoden

- Standardmäßig wird “call by value” verwendet
- out -Parameter: “call by reference”
 - übergebene Variable braucht nicht initialisiert worden sein
 - schreibender Zugriff auf Variablen des Aufrufers notwendig

```
1  class Test
2  {
3      public static void Main() {
4          Int zufallszahl;
5          random (out zufallszahl);
6          ...
7      }
8      public static void random(out int r) {
9          r = ...;
10     }
11 }
```

Methoden

- ref -Parameter: ebenfalls “call by reference”
 - übergebene Variable muss zuvor vom Aufrufer initialisiert worden sein (“in-out”-Parameter)

```
1  class Test
2  {
3      public static void Main() {
4          int a = 1, b = 2;
5          swap (ref a, ref b);
6      }
7      public static void swap(ref int a, ref int b) {
8          int temp = a;
9          a = b;
10         b = temp;
11     }
12 }
```

Beliebige Anzahl von Parametern

- Das Schlüsselwort `params` ermöglicht eine variable Anzahl von Parametern:

```
1  class Test
2  {
3      public static void Main() {
4          int ergebnis = add(1, 2, 3, 4);
5          ..
6      }
7      public static int add(params int[] array) {
8          int sum = 0;
9          foreach (int i in array)
10             sum += i;
11         return sum;
12     }
13 }
```

Überladen von Operatoren

```
1 class Score : IComparable {
2     int value;
3
4     public static bool operator == (Score x, Score y) {
5         return x.value == y.value;
6     }
7
8     public static bool operator != (Score x, Score y) {
9         return x.value != y.value;
10    }
11
12    public int CompareTo(Object o) {
13        return value - ((Score)o).value;
14    }
15 }
16
17 Score a = new Score(5);
18 Score b = new Score(5);
19 Object c = a, d = b;
20
21 if (a == b) { ... } // true
22 if (c == d) { ... } // false
23
24 if ((Object)a == (Object)b) { ... } // false
25 if (((IComparable)c).CompareTo(d) == 0) { ... } // true
```

Properties

- Properties werden wie Variablen angesprochen
 - Zugriffe werden in Methodenaufrufe von Get- und Set-Funktionen umgesetzt
 - “virtuelle Variablen”: existieren nicht, werden semantisch vorgetäuscht

```
1 class C {  
2     private string s = "(not defined)";  
3  
4     public string S {  
5         get { return this.s; }  
6         set {  
7             if (value == null) throw new Exception("null");  
8             this.s = value.ToUpper();  
9         }  
10    }  
11  
12    public static void Main(string[] args) {  
13        C c = new C();  
14        c.S = "Hello World";  
15        Console.WriteLine(c.S);  
16    }  
17 }
```

Properties: Indexer

- Objekte wie Arrays behandeln
- jedes Element wird über get/set-Methoden angesprochen
- Mehrere Argumente pro Indexer und auch mehrere Indexer pro Klasse möglich

```
1 class Skyscraper {  
2     Story[] stories;  
3  
4     public Story this [int index] {  
5         get { return stories [index]; }  
6         set {  
7             if (value != null)  
8                 stories [index] = value;  
9         }  
10    }  
11    ...  
12 }  
13  
14 Skyscraper empireState = new Skyscraper(...);  
15 empireState[102] = new Story ("The Top One", ...);
```

Konstruktoren & Destruktoren

- Konstruktoren wie in Java
 - statische Konstruktoren möglich (“static Klassenname()”)
 - Überladen von Konstruktoren möglich
- Destruktoren werden vom Garbage Collector aufgerufen (kein delete!)
 - Zeitpunkt des Aufrufs kann nicht vorhergesagt werden
 - Es ist nicht garantiert, dass der Destruktor überhaupt aufgerufen wird

Events und Delegates

- Delegates sind eine Art typsicherer objekt-orientierter Funktionszeiger
- können mehrere Methoden (Handler) enthalten
- Sprachunterstützung für Event-Verarbeitung:

```
1  // Typ-Definition
2  delegate void MouseEventHandler(int x, int y);
3  // Event-Erzeuger
4
5  class Window {
6      public event MouseEventHandler mouseChange;
7  }
8
9  // Event-Verbraucher
10 class MouseControl {
11     public MouseControl(Window window) {
12         window.mouseChange +=
13             new MouseEventHandler(myhandler);
14     }
15
16     private void myhandler(int x, int y) {...}
17 }
18
19 Window.mouseChange(4,25); // -> MouseControl.myhandler()
```

Klassen, Interfaces, Vererbung

- Übernommen von C++ ; Syntax identisch für Klassen und Interfaces
- Jedoch keine Vererbung unter Angabe von Zugriffsrechten
- Eine Klasse kann max. von einer anderen Klasse erben, aber mehrere Interfaces implementieren

Die Klasse Object

- von alle anderen Klassen abgeleitet
- Mit GetType() kann der Typ der Klasse abgefragt werden (Reflection)
- Virtuelle Methoden ToString() und Equals()

```
1 class C { ... }  
2  
3 // bedeutet implizit  
4 class C : Object { ... }
```

Interfaces

- Eine Klasse kann nur von einer Basisklasse erben, aber mehrere Interfaces implementieren
- Keine Vererbung bei Strukturen, aber Implementierung von Interfaces möglich

```
1 public interface ITeller {  
2     void Next();  
3 }  
4 public interface IIterator {  
5     void Next();  
6 }  
7  
8 public class Clark : ITeller, IIterator {  
9     void ITeller.Next() { }  
10    void IIterator.Next() { }  
11 }
```

Polymorphie und dynamisches Binden

- In Java: alle Methoden sind virtuell
- In C#: Schlüsselwort `virtual` zur Kennzeichnung von virtuellen Methoden (wie in C++)
- Schlüsselwort `override` zum Überschreiben virtueller Methoden
- Schlüsselwort `new` zum Überschreiben nicht-virtueller Methoden

Polymorphie und dynamisches Binden

```
1 N n = new N ();
2 n.foo(); // foo der Klasse N
3 ((D)n).foo(); // foo der Klasse D
4 ((B)n).foo(); // foo der Klasse D (nicht B!)
```

```
1  abstract class A {
2      public abstract void MyMethod();
3
4      public abstract int X {
5          get;
6          set;
7      }
8  }
9
10 public DerivedFromA: A {
11     public override void MyMethod() { ... }
12
13     public override int X {
14         get { return ... }
15         set { ... = value; }
16     }
17 }
```

Reflection

- Zugriff auf Klasseninformationen über die Type-Klasse

```
1 // Object obj = new C();  
2 Type t = obj.GetType(); // Alternative 1: Object.GetType()  
3 Type t = typeof(C); // Alternative 2: typeof()  
4 Type t = Type.GetType("C"); // Alt. 3: Type.GetType()  
5 // Format: <namespace>.<classname>, <assemblyname>
```

- Zugriff auf Metainformationen:
Type.GetConstructors() , Type.GetMethods() ,
Type.GetProperties() , Type.GetFields()
- Methodenaufruf (Konstruktoraufruf analog)

```
1 MethodInfo m = t.GetMethod ("F", new Type[] { });  
2 m.Invoke (obj, null)
```

Generics

- Erlaubt die Definition parametrisierter Typen

```
1 class Queue<T> {  
2     T[] cell = new T[50];  
3     void append(T item) { ... }  
4 }
```

- Vorteile gegenüber Pseudo-Generizität mit gemeinsamer Basisklasse

```
6 class Stack {  
7     Object[] stack = new Object[50];  
8  
9     void push(Object item) { ... }  
10    Object pop() { ... }
```

- Casts werden benötigt

```
14 String x = (String)mystack.pop();
```

- keine Typsicherheit

```
17 mystack.push(1);  
18 String x = (String)mystack.pop(); // Exception!
```


Generics

- Instantiierung in C# zur Laufzeit; bei C++ zur Compile-/Link-Zeit
- C++ verwendet spezialisierte Versionen für jede Instanziierung (heterogene Übersetzung): effizient, aber Problem der Code-Explosion
- Java bildet Generics auf Object ab und fügt Casts ein (homogene Übersetzung): keine Änderungen an JVM notwendig, aber ineffizient
- C# setzt spezialisierte Versionen für alle primitiven Typen ein, aber den gleichen Methoden-Code für Reference Types

References

1. C# Microsoft Programming Guide:

<http://msdn.microsoft.com/en-us/library/67ef8sbd.aspx>

2. Object-oriented Programming in C#

<http://people.cs.aau.dk/~normark/oop-csharp/pdf/all.pdf>