Java → C#

Vergleich der Sprachsyntax

Gemeinsamkeiten Java – C#

Die Syntax von Java und C# ist vielfach gleich:

- Jede Klasse von Object abgeleitet
  - Java: `java.lang.Object`
  - C#: `System.Object`
- Geschwungene Klammern, Klammernsetzung
- Strichpunkt als Befehlsabschluss
- `if – else`
- `(`Bedingung`) ?` Ja-Teil `:` Sonst-Teil;

Gemeinsamkeiten Java – C#

- **`while` / `do` – `while`**
- **`for` / `continue` / `break`**
- **`switch` / `case` / `break` / `default`**
  - Aber in C#:
    fall-through nicht erlaubt, außer case hat kein Statement

Gemeinsamkeiten Java – C#

- Sichtbarkeit von Variablen
- Operatoren: +,*,&&,||,+=,==,%,…
- Datentypen: int, long, short, float, double,…
  - Aber: Java boolean → C# bool
- enum
  - Aber: enum in C# nicht type-safe:

```
enum Day {MONDAY,…,SUNDAY};
Day d1 = Day.MONDAY; //OK Java u. C#
Day d2 = (Day)666;   //OK in C#, Error in Java
```

- Interfaces
- Mehrfachvererbung nicht möglich

Gemeinsamkeiten Java – C#

- Strings sind unveränderlich
  - Vereinfachung für Escape-chars in C#:

```
String s = "C:\\temp";
String s = @"C:\temp";
```

- Exceptions
  - abgeleitet von einer Exception-Klasse
  - try – catch – finally – throw
  - Aber: in C# gibt es keine throws-Anweisung
- Boxing / Unboxing
- Automatic Garbage Collection

Unterschiede
Java ↔ C#

Auf den folgenden Folien steht links Java-Code, rechts C#-Code

Konventionen

Klassennamen groß	**Klassennamen groß**
Methodennamen klein	**Methodennamen groß**

- Jeden Wortanfang groß schreiben (z.B. *ShowDialog)* – *Upper Camel Case=Pascal Case*

Style	Beispiel
Pascal Case	MyDummyName
Camel Case	myDummyName
Snake Case	MY_DUMMY_NAME
Kebab Case	my-dummy-name
Hungarian Notation	sMyDummyName

- Anfangsbuchstabe groß, außer bei Variablen, Konstanten und Feldern, die man nicht von außen sieht.
- Methoden sollten mit Verb beginnen (z.B. *GetHashCode)*
- Alles andere sollte mit Substantiv beginnen (z.B. *Size, IndexOf, Collections)*
- enum-Konstanten oder bool-Members können mit Adjektiv beginnen (*Red, Empty)*

Konstanten	klein	*size* (public-Konstanten groß, z.B. *MaxValue*)
Variablen	klein	*i, top, sum*
Felder	klein	*width, bufferLength* (public-Felder groß)
Properties	groß	*Length, FullName*
Enum-Konstanten	groß	*Red, Blue*
Methoden	groß	*Add, IndexOf*
Typen	groß	*StringBuilder* (vordefinierte Typen klein: int, string)
Namespaces	groß	*System, Collections*

Fundamentale Typen in C#

Folgende Typbezeichner sind in C# identisch:

`int`	`System.Int32`
`short`	`System.Int16`
`long`	`System.Int64`
`float`	`System.Single`
`double`	`System.Double`
`bool`	`System.Boolean`
`char`	`System.Char`
`string`	`System.String`
`object`	`System.Object`

Value Types

Nur primitive Typen (int, float,…) sind Value Types.

Man kann eigene Value Types definieren, indem man **`class`** durch **`struct`** ersetzt:

```
struct Point {…}
```

Für value types gilt:

- Parameterübergabe immer by value
- werden am Stack erzeugt
- belasten daher den Garbage Collector nicht
- erhöhen somit die Performance (bei richtigem Einsatz)

Für reference types gilt:

- „Zeiger"-Typen
- Parameterübergabe immer by reference
- werden am Heap erzeugt

struct-Variable werden jedoch ebenfalls mit new angelegt (also z.B. new Point(…)).

Parameterübergabe by reference

Call by reference ist in Java nicht möglich, in C# mit Schlüsselwort ref bzw. out.

```
public static void Swap(ref int x, ref int y)
{
   int temp = x;
   x = y;
   y = temp;
}
…
int a = 666; int b = 999;
Swap(ref a, ref b);
//value now: a->999, b→666
```

Achtung: ref muss sowohl bei der Signatur als auch beim Aufruf angegeben werden!

nullable Types

Value Type, der auch `null` als Wert annehmen kann

-	int? double? …

```
int? x = 666;
double? y = null;
if (x.HasValue) {…}
int z = x ?? -1;
double v = y.GetValueOrDefault();
```

Werden hauptsächlich verwendet, wenn C#-Klasse auf DB-Tabellen gemapped werden oder bei Web-Services.

Arrays

```
int[] arr1 = new int[66];
int arr2[] = new int[66];
```

```
int[] arr1 = new int[66];
int arr2[] = new int[66];
```

Klassen pro Source-file

Eine Klasse pro File	Beliebig viele Klassen pro File
File muss heißen wie die Klasse	Name des Files beliebig

Klasse auf mehrere Source-files aufteilen

-

File MyClass.cs:

```
public partial class MyClass
{
  private int val;
}
```

File MyClass.Impl.cs:

```
public partial class MyClass
{
  public MyClass(int x){
    this.val = x;
  }
}
```

Pakete

```
package at.grueneis.mypack;
class MyClass {…}
```

```
namespace MyPack {
   class MyClass {…}
}
```

```
import at.grueneis.mypack;
```

```
using MyPack;
```

Daher ist folgendes identisch (wenn `using System` angegeben wird)

```
String s = "abc";  //entspricht System.String
string s = "abc";
```

In C# bestimmt der Namespace nicht automatisch die Verzeichnisstruktur.

Naming Conventions einhalten!

Syntax für Ableitung

```
class A extends B
  implements MyInterface
```

```
class A : B, IMyInterface
```

extends/implements ➔ ":"

Trennen mit ","

In C# kann man bei Ableitung nicht zw. Klasse u. Interface unterscheiden ➔ Konvention, dass Interfaces mit `I` beginnen

Bei Ableitung von Klasse und Interface ➔ zuerst die Klasse, dann die Interfaces

Initialisierung der Basisklasse

```
class A extends B
{
  public A(int x, String s)
  {
    super(x);
    this.s = s;
  }
}
```

```
class A : B
{
  public A(int x, String s)
          : base(x)
  {
    this.s = s;
  }
}
```

Constructor Chaining

```
public A(String s)
{
  this(666,s);
}
public A(int x, String s)
{…}
```

```
public A(String s)
        : this(666,s) {}


public A(int x, String s)
{…}
```

Klasse von der nicht abgeleitet werden darf

```
final class Person {…}
```

```
sealed class Person {…}
```

Statischer Konstruktor

```
class MyClass{
   static {…}
}
```

```
class MyClass{
   static MyClass() {…}
}
```

main (Einsprungpunkt)

```
class MyClass{
  public static void
  main(String[] args) {…}
}
```

```
class MyClass{
  public static void
  Main(String[] args) {…}
}
```

virtuelle Methoden

```
public void f() {…}
```

```
public virtual void F() {…}
```

In Java sind alle Methoden automatisch virtuell, in C# muss das mit virtual explizit erlaubt werden.

```
@Override
public void f() {…}
```

```
public override void F(){…}
```

Für überschreibende Methoden muss in C# das Schlüsselwort override angegeben werden. Die Annotation in Java ist nicht zwingend notwendig.

Zugriffs-Modifizierer

private	**private** (innerhalb Klasse)
public	**public**
package	**internal** (innerhalb Assembly)
protected (package + Subclasses)	**protected** (abgeleitete Klassen)

Wird kein Modifizierer angegeben, ist der default in Java package, in C# ist er private (für Methoden/Variablen) bzw. internal (für Klassen)

protected hat in C# andere Bedeutung: Zugriff für alle abgeleiteten Klassen (unabhängig vom namespace)!

Konstanten

`final static int ABC=123;`	`const int ABC=123;`
`final int XYZ=987;`	`readonly int XYZ=987;`

const-Variable sind automatisch static in C#!

Properties

getter- u. setter-Methoden	Properties

```
class Person {
  private String name;

  public String getName()
  { return name; }

  public void setName(String name)
  {  this.name = name; }
}
```

```
class Person {
  private String name;
  public String Name
  {
    get { return name; }
    set { this.name = value; }
  }
}
```

Man kann get/set unterschiedliche Sichtbarkeit geben (bzw. weglassen):

```
public String getName()
  { return name; }

private void setName(String name)
  {  this.name = name; }
```

```
public String Name
{
    get { return name; }
    private set
        { this.name = value; }
}
```

Shortcut: propfull (+ 2x<Tab>)

Properties II

Folgende Kurzschreibweise ist in C# möglich (=„Autoproperties“):

```
public string Name { get; set; }
```

Shortcut: prop (+ 2x<Tab>)

Initialisierung von Autoproperties

Seit C# 6 kann man Autoproperties auch mit Werten initialisieren:

```
class Car
{
  public string Brand { get; set; } = "VW";
  public int Year { get; set; } = 2006;
  public int NrTires { get; } = 4;
}
```

```
var car = new Car();
```

car	{CSharp_Extensions.Car}
car.Brand	"VW"
car.NrTires	4
car.Year	2006

Fehlt der Setter, ist die Property unveränderlich.

casts

```
if ( x instanceof MyClass )
{
  MyClass y = (MyClass)x;
}
```

```
if ( x is MyClass )
{
  MyClass y = (MyClass)x;
  MyClass z = x as MyClass;
}
```

- Erste Variante wirft Exception, wenn cast fehlschlägt
- Bei zweiter Variante keine Exception, Ergebnis ist null

variable Parameterlisten

```
int f(Integer… vals) {…}
```

```
int f(params int[] vals){…}
```

Synchronisation von Code-Blöcken / Methoden

```
synchronized(this) {…}
```

```
lock(this) {…}
```

```
synchronized void f() {…}
```

```
[MethodImpl(MethodImplOptio
  ns.Synchronized)]
void F() {…}
```

Annotations

```
@xxx
```

```
[xxx]
```

Die Namen der eigentlichen Annotations sind in Java/C# fast immer unterschiedlich!

Serialisierung

```
class Abc implements
  Serializable {…}
```

```
[Serializable]
class Abc {…}
```

In beiden Sprachen muss nichts extra implementiert werden

Ausgabe

```
System.out.println(…);
```

```
Console.WriteLine(…);
Debug.WriteLine(…);
```

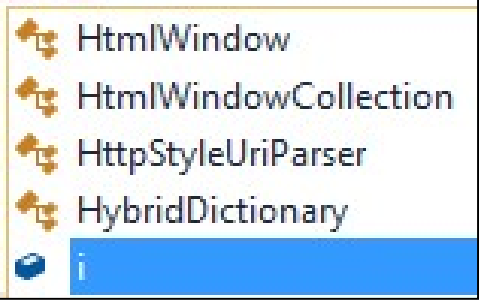
- **Debug** Erfordert **using System.Diagnostics**
- Console u. Debug schreiben beide auf die Console
  - Console schreibt auf Command-Fenster bei Konsolenapplikationen
  - Debug schreibt auf Output-Tab im VS bei Konsolenapplikationen
- **Debug.Write*XXX*()** wird beim Release-Build wegoptimiert
- Debug-Klasse hat weitere Methoden, z.B.
  Debug.WriteLineIf(…)

Formatierte Strings

- Mit **$"{<expr>}"** kann man Strings einfach formatieren
- incl. Intellisense

```
double f = 123.4567;
int i = 12345;
string s = $"f={f} i={i}";
```

```
double f = 123.4567;
int i = 12345;
string s = $"f={f} i={i";
```



- Format kann detaillierter angegeben werden:

```
double f = 123.4567;
int i = 12345;
string s = $"f={f:000#.##} i={i:d7}";
```

```
"f=0123,46 i=0012345"
```

Formatierte Strings – Formate Zahlen

- Symbol-Formate (Bsp. für int Wert = 1000000;)

Symbol	Typ	Aufruf	Ergebnis
c	Währung (currency)	$"{Wert:c}"	1.000.000,00 €
d	Dezimalzahl (decimal)	$"{Wert:d}"	1000000
e	Wissenschaftlich (scientific)	$"{Wert:e}"	1,00E+06
f	Festkommazahl (fixed point)	$"{Wert:f}"	1000000
g	Generisch (general)	$"{Wert:g}"	1000000
n	Tausender trennzeichen	$"{Wert:n}"	1.000.000,00
x	Hexadezimal	$"{Wert:x4}"	f4240

- Allgemein:
  - #: Zahl-Platzhalter
  - 0: Null-Platzhalter

Format/value	12.34	0	1000
$"{val:#.#}"	12,3		1000
$"{val:000.000}"	012,340	000,000	1000,000
$"{val:##0.0##}"	12,34	0,0	1000,0

Formatierte Strings – Datumsformate 1/2

- `DateTime clock = DateTime.Now;`

Symbol	Typ	Aufruf	Ergebnis
d	kurzes Datumsformat	$"{clock:d}"	12.09.2018
D	langes Datumsformat	$"{clock:D}"	Mittwoch, 12. September 2018
t	kurzes Zeitformat	$"{clock:t}"	11:45
T	langes Zeitformat	$"{clock:T}"	11:45:18
f	Datum & Uhrzeit komplett (kurz)	$"{clock:f}"	Mittwoch, 12. September 2018 11:45
F	Datum & Uhrzeit komplett (lang)	$"{clock:F}"	Mittwoch, 12. September 2018 11:45:18
g	Standard-Datum (kurz)	$"{clock:g}"	12.09.2018 11:45
G	Standard-Datum (lang)	$"{clock:G}"	12.09.2018 11:45:18
M	Tag des Monats	$"{clock:M}"	12. September
r	RFC1123 Datumsformat	$"{clock:r}"	Wed, 12 Sep 2018 11:45:18 GMT
s	sortierbares Datumsformat	$"{clock:s}"	2018-09-12T11:45:18
u	universell sortierbares Datumsformat	$"{clock:u}"	2018-09-12 11:45:18Z
U	universell sortierbares GMT-Datumsformat	$"{clock:U}"	Mittwoch, 12. September 2018 09:45:18
Y	Jahr/Monats-Muster	$"{clock:Y}"	September 2018

Formatierte Strings – Datumsformate 2/2

- `DateTime clock = DateTime.Now;`

Symbol	Typ	Aufruf	Ergebnis
yy	Jahr 2stellig	$"{clock:yy}"	18
yyyy	Jahr 4stellig	$"{clock:yyyy}"	2018
MM	Monat	$"{clock:MM}"	09
MMM	Monatsname (Kürzel)	$"{clock:MMM}"	Sep
MMMM	Monatsname (ausgeschrieben)	$"{clock:MMMM}"	September
dd	Tag	$"{clock:dd}"	12
ddd	Tagname (Kürzel)	$"{clock:ddd}"	Mi
dddd	Tagname (ausgeschrieben)	$"{clock:dddd}"	Mittwoch
hh	Stunde 2stellig	$"{clock:hh}"	11
HH	Stunde 2stellig (24-Stunden)	$"{clock:HH}"	11
mm	Minute	$"{clock:mm}"	45
ss	Sekunde	$"{clock:ss}"	18
fff	Millisekunden	$"{clock:fff}"	423
tt	AM oder PM (nur englisch)	$"{clock:tt}"	
zz	Zeitzone (kurz)	$"{clock:zz}"	+02
zzz	Zeitzone (lang)	$"{clock:zzz}"	+02:00
gg	Ära	$"{clock:gg}"	n. Chr.

Datum / Uhrzeit

LocalDateTime	DateTime

```
DateTime now = DateTime.Now;
DateTime d = new DateTime(2017,9,12);
DateTime nextWeek = d.AddDays(7);
string s1 = d.ToShortDateString();
string s2 = d.ToShortTimeString();
string s3 = d.ToLongDateString();
string s4 = $"{d:dd.MM.yyyy}";
int day = d.Day; //[1,31]
…
```

File Ein-/Ausgabe

- File lesen:

```
string[] lines = File.ReadAllLines(@"C:\Temp\xyz.txt");
```

- File schreiben:

```
StreamWriter outFile = new StreamWriter(@"C:\Temp\abc.txt");
outFile.WriteLine("xxx");
outFile.WriteLine("yyy");
outFile.Close();
```

- Klassen File u. StreamWriter liegt im Namespace System.IO
- ➔ using System.IO;

Collection Typen

`ArrayList`	`ArrayList, SortedList`
`LinkedList`	`-`
`ArrayList<T>` `LinkedList<T>`	`List<T>, SortedList<T>` `LinkedList<T>`

`cltn.add(666);`
`cltn.add(0,999);`
`int x = cltn.get(0);`
`int nr = cltn.size();`
	`cltn.Add(666);`
	`cltn.Insert(0,999);`
	`int x = cltn[0];`
	`int nr = cltn.Count;`

`Stack, Stack<T>`	`Stack, Stack<T>`
`Queue, Queue<T>`	`Queue, Queue<T>`

In C# erlaubt: List<int> x = new List<int>(); //Java: List<Integer>

for Schleife für Collections/Arrays

`for( int val : cltn) {…}`	`foreach( int val in cltn) {…}`

Dictionary Typen

HashMap	Hashtable
TreeMap	-
HashMap<K,V> TreeMap<K,V>	Dictionary<K,V> SortedDictionary<K,V>

map.put("abc",1);
int x = map.get("abc");
int nr = map.size();
	map["abc"] = 1;
	int x = map["abc"];
	int nr = map.Count;

Map.Entry Map.Entry<K,V>	DictionaryEntry KeyValuePair<K,V>
HashSet	-
TreeSet	-
HashSet<T> TreeSet<T>	HashSet<T>

Objekt-Initialisierer

```
class Person
{
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public int Age { get; set; }
  public override string ToString()
  {
    return $"{FirstName} {LastName} [{Age}]";
  }
}
```

Objekt initialisieren durch Angabe der public Variablen/Properties in geschwungenen Klammern

```
Person p = new Person { Age = 66, Firstname = "Hansi", Lastname = "Huber" };
```

Collection-Initialisierer

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
```

```
List<Person> persons = new List<Person> {
  new Person { Age = 66, Firstname = "Hansi", Lastname = "Huber" },
  new Person { Age=55, Firstname="Susi", Lastname="Müller" }
};
```

```
Dictionary<string, int> phoneBook = new Dictionary<string, int> {
    { "Hansi", 4711 },
    { "Pauli", 3456 },
    { "Susi", 7256 }
};
```

```
Dictionary<string, int> phoneBook = new Dictionary<string, int>
{
  ["Hansi"] = 4711,
  ["Pauli"] = 3456,
  ["Susi"] = 7256
};
```

2D-Array

- Erzeugen:

```
int[,] values = new int[3,2];
```

- Zuweisen:

```
values[1, 1] = 11;
values[2, 0] = 20;
```

- Ausgeben
  - Anzahl der Zeilen: **`GetLength(0)`**
  - Anzahl der Spalten: **`GetLength(1)`**

```
for (int i = 0; i < values.GetLength(0); i++)
{
  for (int j = 0; j < values.GetLength(1); j++)
  {
    Console.Write($"{values[i, j]}");
  }
  Console.WriteLine();
}
```