




# Von C nach Java

Institut für  
Computertechnik  
**ICT**  
Institute of  
Computer Technology

## Überblick

- Vergleich von C und Java
- Java angewendet
- Java Code-Konventionen
- Ablaufsteuerung in Java
- Fehlerbehandlung in Java



Institute of Computer Technology

2

## Vergleich von C und Java

### ■ C

- prozedurale Sprache
- Zeiger
- keine Speicherverwaltung
- Funktionen-Bibliothek
- einsetzbar für z.B. Embedded Systems (Hardware-nah) oder Systemprogrammierung
- Programme können zu effizienter Ausführung "getrimmt" werden

### ■ Java

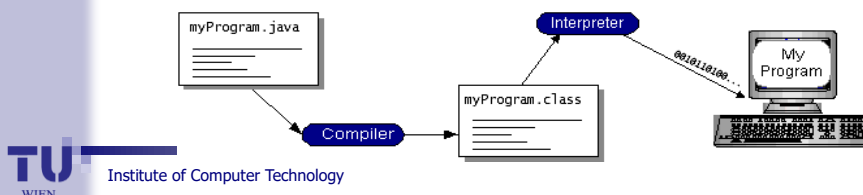
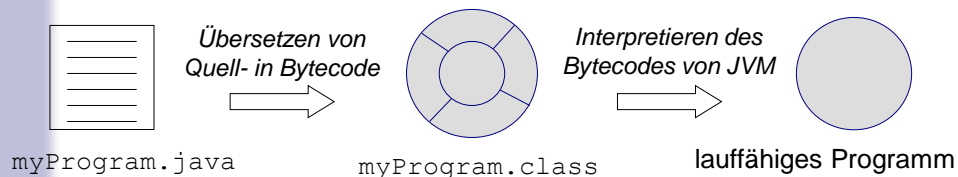
- objektorientierte Sprache mit prozeduralen Anteilen
- keine Zeiger
- Speicher wird von der Java Virtual Machine (JVM) verwaltet (Garbage Collector)
- Klassen-Bibliothek
- einsetzbar für z.B. Handy-Anwendungen, Web-Anwendungen, Verteilte Systeme
- manchmal ineffizient zur Laufzeit

## Vergleich von C und Java – prozedural

- C wird den prozeduralen Anteilen von Java gegenübergestellt.
- Die objektorientierten Anteile von Java folgen.
- Einige Anteile von Java werden in dieser VU gar nicht behandelt.

## Java angewendet

- Java-Code wird in einen **plattform-unabhängigen Bytecode** übersetzt.
- Bytecodes werden von einer **plattform-abhängigen Java Virtual Machine (JVM)** zur Laufzeit interpretiert.



5

## Java angewendet – Primitive Datentypen

### ■ C

char  
short int  
int  
long int

### ■ Java

char //16-bit Unicode  
short //16-bit integer  
int //32-bit integer  
long //64-bit integer  
byte //8-bit integer

float  
double  
long double

float //32\_bit single precision  
double //64\_bit double precision  
boolean //true or false

6

## Listen in Zusammenhang mit Generics

- Listen haben einen „generischen“ Datentyp
- Generische Datentype definiert die Elemente der Liste (z.B. String)
- Es gibt mehrere Implementierungen für Listen
  - Für die Übung ist die einfachste Implementierung der Vector.
- Beispiel

```
List<String> stringList = new Vector<String>();
```

Näheres dazu finden Sie im Foliensatz: **Generics**

## Java angewendet

- Java-Programme bestehen aus Klassen.
- Java-Anwendungen besitzen immer (zumindest) **eine** spezielle Methode: **main()**.  
(Libraries z.B. benötigen keine **main**-Methode)
  - Definiert den Zugangspunkt der Anwendung

```
public class SagHallo {  
    public static void main( String[] args ) {  
        System.out.println( "Hello World" );  
    }  
}
```

## Java angewendet

- Argumente können wie folgt an eine Java-Anwendung übergeben werden:

```
public class SagHallo {

    public static void main( String[] args ) {
        if ( args.length > 0 )
            System.out.println( "Hallo " + args[0] );
        else
            System.out.println( "Hallo" );
    }
}
```

```
C:> javac SagHallo.java
C:> java SagHallo Thomas
Hallo Thomas
C:>
```

## Java Code-Konventionen

- Konventionen sind aus vielen Gründen wichtig:
  - **80%** der **Kosten** während der **Lebensdauer** einer Software fließen in die **Wartung**.
  - Kaum eine Software wird während seiner **gesamten Lebensdauer vom ursprünglichen Programmierer** gewartet.
  - Konventionen **verbessern die Lesbarkeit** von Software und ermöglichen ein schnelleres und gründlicheres Verständnis von neuem Code.

Damit **Konventionen wirken** können,  
muss **sich jede** Person, welche Software verfasst,  
**an die Konventionen halten.**

## Java Code-Konventionen

- <http://www.oracle.com/technetwork/java/codeconventions-150003.pdf>
- Dateinamen:
  - Quell-Code: `<name>.java`
  - Bytecode: `<name>.class`

## Java Code-Konventionen – Datei-Organisation

- C

```
#include <stdio.h>;
FILE *f;
```
- Java

```
import java.io.File;
File f;
```

oder ohne "import":

```
//import java.io.File;
java.io.File f;
```

## Java Code-Konventionen

### ■ Einrückungen

```

if ( ( condition1 && condition2 )
      || ( condition3 && condition4 )
      || ! ( condition5 && condition6 ) ) {
    doSomething();
}

```

### ■ Anweisungen

```

argv++; argc--;    // VERMEIDEN!

```

```

argv++;              // besser
argc--;

```

## Java Code-Konventionen – Kommentare

### ■ Java stellt drei Kommentar-Stile zur Verfügung:

#### ■ Block-Kommentar

```

/* Dieser Kommentar geht über mehrere Zeilen -
   bis zum schließenden Symbol
   "Stern-Schrägstrich".
*/

```

#### ■ Ein-Zeilen-Kommentar

```

// Das ist ein einzeiliger Kommentar.

```

#### ■ Javadoc-Kommentar

```

/** Dieser Kommentarstil wird vom javadoc-
    Hilfsprogramm verwendet, um Dokumentation zu
    generieren (-> spezielle Syntax).
*/

```

## Java Code-Konventionen – Deklarationen

- `int level, size;`
- `int level; // besser`
- `int size;`

- **// FALSCH!**
- `int foo, fooarray[];`
- // richtig**
- `int foo;`
- `int[] fooarray;`

- `int count;`
- `...`
- `meineMethode() {`
- `if ( bedingung ) {`
- `// VERMEIDEN!`
- `int count = 0;`
- `...`
- `}`
- `...`
- `}`

## Java Code-Konventionen – Leerräume

- Leerzeilen
  - Zwischen **Methoden**
  - Zwischen **lokalen Variablen** einer Methode und der **ersten Anweisung**
  - Vor einem **Kommentar**
  - Zwischen **logischen Abschnitten** innerhalb einer Methode zur **Verbesserung der Lesbarkeit**

- Abstände

`a1=c3+d4;`     `a1 = c3 + d4;`

`for (ausdr1; ausdr2; ausdr3)`

`for ( ausdr1; ausdr2; ausdr3 )`



## Java Code-Konventionen – Namensgebung

### ■ Packages

`java.io.*;` → Kleinbuchstaben  
`java.net.ServerSocket;`

### ■ Klassen

`class String` → Hauptwörter – groß geschrieben  
`class ImageSprite` → einfach – Akronyme vermeiden

### ■ Methoden

`getValue()` → Kleinbuchstaben – sollten mit einem Verb beginnen

### ■ Variablen

`name` → Kleinbuchstaben

### ■ Konstanten

`CONST_VALUE` → Blockschrift

## Ablaufsteuerung in Java

### ■ Die Möglichkeiten der Ablaufsteuerung in C und Java sind die gleichen:

#### ■ Sequentiell:



#### ■ Auswahl:



`if [else], switch`

#### ■ Wiederholung:



`for, while, do`

#### ■ Übergang:



**Sprünge, Aufrufe**

## Ablaufsteuerung in Java

- Von **{ }** umschlossene Anweisungen bilden eine zusammengesetzte Anweisung.
  - Auch **Block** genannt.

```
if ( bedingung )
    anweisung1;
    anweisung2;
anweisung3;
```

**/\* VERMEIDEN - Kann  
zu Unklarheit  
führen! \*/**

```
if ( bedingung ) {
    anweisung1;
    anweisung2;
}
```

**// eindeutig**

## Ablaufsteuerung – Die **if**-Anweisung

- C
- Java

```
if ( ausdruck ) {
    anweisung1;
} else {
    anweisung2;
}
```

```
if ( ausdruck ) {
    anweisung1;
} else {
    anweisung2;
}
```

- Java-Beispiel:

```
if ( i % 2 == 0 ) {
    System.out.println( "gerade" );
} else {
    System.out.println( "ungerade" );
}
...
```

## Ablaufsteuerung – Die **switch**-Anweisung

- Nützlich, um zwischen einer Anzahl von möglichen Werten zu wählen.
- Die **switch**-Anweisung ist eine Kontrollstruktur, die mehrere potentielle Ausführungspfade hat.
- Unterstützte Typen der `switch`-Variable ab Java Version 1.8
  - ENUM
  - String
  - Character
  - Byte
  - Short
  - Integer

## Die **switch**-Anweisung Code Fragmente

### ■ C

```
switch( int_variable ) {  
    case konstante:  
        anweisung1;  
        anweisung2;  
        break;  
    default:  
        anweisung3;  
        anweisung4;  
}
```

### ■ Java

```
switch( switch_variable ){  
    case konstante_1:  
        ausdruck1;  
        break;  
    case konstante_2:  
        ausdruck2;  
        break;  
    default:  
        ausdruck3;  
        break;  
}
```

## Ablaufsteuerung – Schleifen

- **while**, **do** und **for** (ab Java 1.8 auch **Streams**, werden in dieser LVA aber nicht behandelt)

- C

```
while( bool_ausdruck ){
    anweisung;
}
```

- Java

```
while( bool_ausdruck ){
    anweisung;
}
```

- Java-Beispiel:

```
int i = 0;
while( i < 120 ) {
    System.out.println( "i= " + i );
    i++;
}
```

## Mögliche Schreibweisen für **for** Schleifen

```
List<String> list = new Vector<String>();
for ( int i=0; i < list.length; i++ ){
    // Das Interface List definiert die get-Methode.
    System.out.println( list.get(i) );
}
```

```
List<String> list = new Vector<String>();

// Verkürzte for-Schleife (setzt Iterable voraus).
for ( String s : list ) {
    System.out.println(s);
}
```

## Iteratoren

```
List<String> list = new Vector<>();
Iterator<String> iterator = list.iterator();

// solange es noch ein nächstes Element gibt
while( iterator.hasNext() ){

    // liefert das nächste Element
    // springt danach um eine Stelle weiter
    String element = iterator.next();

}
```

## Fehlerbehandlung in Java

- Ausnahmebehandlung durch sogenannte **Exceptions**:
  - Auslösen (**throw**) einer Ausnahme sobald ein Fehler eintritt.
  - Abfangen (**catch**) einer Ausnahme in einer übergeordneten Methode.

## Fehlerbehandlung – Was ist eine Ausnahme?

- Java-Methoden können eine Ausnahme auslösen, wenn sie aus irgendeinem Grund versagen.
  - Die Programm-Steuerung gibt unmittelbar an den entsprechenden "Exception Handler" weiter.
  - Fehlerbehandlung durch Ausnahmen ist robust.

## Fehlerbehandlung – Abfangen von Ausnahmen

- Beim Aufruf einer Methode, die eine Ausnahme auslösen könnte  
(z.B.: `java.io.Reader.read()` )

Aufruf in einem  
**try**-Block einschließen

**catch**-Block  
definieren, um die  
Ausnahme zu behandeln

```
try {  
    // kann eine I/O-Ausnahme  
    // auslösen  
    b = file1.read();  
    b = file2.read();  
    b = file3.read();  
}  
catch( IOException e ) {  
    // Behandlung der Ausnahme  
}
```

## Fehlerbehandlung – Deklarierte Ausnahmen

- Methoden geben an, welche Ausnahmen sie auslösen könnten oder selbst nicht abfangen:

```
public class URL {  
    public URL(String s) throws MalformedURLException  
    {...}  
}
```

- Diese deklarierten Ausnahmen (Checked Exceptions) müssen im Code abgefangen werden sobald die Methode verwendet wird, oder die aufrufende Methode muss auch diese Exception werfen.

## Fehlerbehandlung – Nicht deklarierte Ausnahmen

- Sind Exception (Unchecked Exceptions), welche nicht zwingend behandelt werden müssen (z.B. NullPointerException).
- Die Fehlerbehandlung wird vom Compiler nicht erzwungen.
- Können aber ebenfalls mittels eines try-catch Blocks behandelt werden.