



# Programmierung 2

Vorlesung 1: Grundlagen, Übergang von C  
April 2025

`alexander.gepperth@cs.hs-fulda.de`



# Von C nach Java: quick & dirty



# C

- C ist eine imperative Sprache
  - Fokus auf konkreter Umsetzung: Funktionen, Schleifen, Variablen, Bedingungen,
  - maximale Freiheit für Programmierer\*innen
  - kaum Einschränkungen (Projektstruktur, Programmstruktur, ..)
- C wird direkt in Maschinensprache kompiliert



# Java

- Java ist eine objektorientierte Sprache
  - enthält alle Sprachelemente von C
  - alle wichtigen Konstrukte 1:1 übernommen
  - zusätzliche Konzepte: Klassen, Packages, Projekte
  - Freiheit der Programmierer\*innen eingeschränkt zugunsten guter Projekt/Programmstruktur
- Java wird in **Bytecode** für die **JVM** kompiliert (mehr dazu später!)



# Vom Programm zur Klasse

- Code und Daten müssen in Java stets Teile von **Klassen** sein
  - Code: **Methoden** (statt Funktionen)
  - Daten: **Attribute** (statt Variablen)
- Code außerhalb einer Klasse ist nicht möglich!
- Klassen in etwa: `structs` mit Code!
- **Beispiel:** Minimale Java- und C-Programme



# **Beispiel: Migration von C- Programm in ein Java-Projekt**

- `c_prog.c`, `JavaClass.java` im E-Learning!



# **Beispiel: wir programmieren in Java wie in C**

- `large_c_prog.c`, `LargeJavaClass.java` im E-Learning!



# Zwischenfazit

- In Java kann man wie in C programmieren

C	Java
Programmdatei mit Funktion <code>main()</code>	Datei mit Klasse, Klasse hat Methode <code>main()</code>
globale Variablen	<code>static</code> -Attribute
(globale) Funktionen	<code>static</code> -Methoden
lokale Variablen	lokale Variablen
<code>#include</code>	<code>import</code>
<code>malloc</code>	<code>new</code>
--	Klassen





# Aber: **Vorsicht!**

- Java ähnelt C sehr, quasi alle Konstrukte haben identische Syntax
- Trotzdem große Unterschiede „unter der Oberfläche“!
- Objektorientierte Programmierung mit Klassen und Instanzen erfordert Umdenken!
- **(Mehr dazu später)**



# CUT: Q&A



# Wiederholung: imperative Sprachkonzepte in Java

- Falls Sie sich mit C sehr gut auskennen, können Sie diesen Teil überspringen!



# Wiederholung: imperative Sprachkonzepte in Java

- Schleifen: while, do/while, for
- Kontrollstrukturen: if, break, continue
- Operatoren: Inkrement, Dekrement, ternärer Operator

## Kap. 2.4.5

# Inkrement/Dekrementoperatoren

- Sehr häufiger Fall: 

```
for (int i = 0; i < 10; i = i+1) { ... }
```
- es existieren spezielle Operatoren zur Erhöhung/Erniedrigung von Zahlen um 1: **++**, **--**

```
for (int i = 0; i < 10; i++) { ... }
```

```
for (int i = 10; i > 0; i--) { ... }
```

- effizienter und lesbarer als `i = i + 1` oder `i += 1`
- funktioniert für alle Zahlentypen (`int`, `long`, `double`, `float`, ...)
- Liefern Ergebnis zurück



## Kap. 2.4.5

# Inkrement/Dekrementoperatoren

- Die Operatoren ++ und - - können vor oder hinter einer Variable stehen, erhöhen/erniedrigen in beiden Fällen
- Position wichtig wenn der Rückgabewert eine Rolle spielt:

```
int i=3;  
System.out.println(i++);
```

```
int i=3;  
System.out.println(++i);
```

Ausgabe ?



## Kap. 2.4.5

# Inkrement/Dekrementoperatoren

- Die Operatoren ++ und - - können vor oder hinter einer Variable stehen, erhöhen/erniedrigen in beiden Fällen
- Position wichtig wenn der Rückgabewert eine Rolle spielt:

```
int i=3;  
System.out.println(i++);
```

3

```
int i=3;  
System.out.println(++i);
```

4



## Kap. 2.4.5

# Inkrement/Dekrementoperatoren

- Die Operatoren ++ und - - können vor oder hinter einer Variable stehen, erhöhen/erniedrigen in beiden Fällen
- Position wichtig wenn der Rückgabewert eine Rolle spielt:

```
int i=3;  
System.out.println(i++);
```

**3** `i++` gibt den alten Wert zurück

```
int i=3;  
System.out.println(++i);
```

**4** `++i` gibt den neuen Wert zurück





## Kap. 2.6

# Der ternäre Operator

- Aus Prog1 bekannt ist die `if`-Anweisung:

```
if (test == true) {  
    System.out.println ("Zugriff erteilt");  
} else {  
    System.out.println ("Zugriff verweigert!");  
}
```

- Falls nur Werte von einer Bedingung abhängen und keine Befehls(blöcke): ternärer Operator

```
System.out.println( (test == true) ? "Zugriff erteilt" :  
                    "Zugriff verweigert" );
```

## Kap. 2.6

# Der ternäre Operator

- Also allgemein: Operator liefert Wert zurück

Bedingung **?** Wert-falls-wahr **:** Wert-falls-falsch

- Klammerung oft sinnvoll damit Operatorpräzedenz klar ist

(a > b) **?** a **:** b

- Erzeugt meist kompakten aber lesbaren Code

```
int max ;  
if (a > b) {  
    max = a;  
} else {  
    max = b;  
}
```

Zum Vergleich: selber  
Effekt mit if/else!



## Kap. 2.6



# Bekannte Schleifen

- Schleifen: aus Prog1 bekannt sind
  - for
  - while

```
for (int i = 0; i < 10; i += 1) {  
    System.out.println(i);  
}
```

```
int i = 0;  
while (i < 10) {  
    System.out.println(i);  
    i += 1;  
}
```



## Kap. 2.6

# do-while-Schleife

- `for`, `while` testen am Schleifenanfang
- manchmal ist es praktisch am Ende zu testen:  
do-while-Schleife

```
int i = 0;
do {
    System.out.println(i);
    i += 1 ;
} while (i < 10) ;
```



## Kap. 2.6

# break

- Vorzeitiges und komplettes Verlassen einer Schleife: `break`
- Beispiel: kommt in einem Array `arr` der Wert 1 vor?

```
...  
wert_gefunden = false;  
for (int i = 0; i < arr.length; i += 1) {  
    if (arr[i] == 1) {  
        wert_gefunden = true ;  
    }  
}  
...  

```


- Schleife wird auf jeden Fall bis zum Ende durchlaufen
- u.U. sehr ineffizient

## Kap. 2.6

# break

- Vorzeitiges und komplettes Verlassen einer Schleife: **break**
- Beispiel: kommt in einem Array **arr** der Wert 1 vor?

```
...  
wert_gefunden = false;  
for (int i = 0; i < arr.length; i += 1) {  
    if (arr[i] == 1) {  
        wert_gefunden = true ;  
        break ;  
    }  
}  
...  
}
```

A red curved arrow originates from the 'break ;' statement and points to the closing curly brace of the 'for' loop, indicating that the loop is terminated immediately.

- Schleife wird verlassen sobald Wert gefunden ist
- u.U. viel schneller



## Kap. 2.6

# continue

- Vorzeitiges Beenden einer Schleifeniteration durch **continue**-Befehl, weiter mit nächster Iteration
- vereinfacht oft komplizierten Code, prinzipiell immer durch `if..else` ersetzbar

```
for (int i = 0; i < 100; i = i + 1) {  
    if (i % 2 == 0) {  
        continue;  
    }  
    System.out.println("Zahl ist: " + i);  
}
```

Ausgabe ?



## Kap. 2.6

# continue

- Vorzeitiges Beenden einer Schleifeniteration durch **continue**-Befehl, weiter mit nächster Iteration
- vereinfacht oft komplizierten Code, prinzipiell immer durch `if..else` ersetzbar

```
for (int i = 0; i < 100; i = i + 1) {  
    if (i % 2 == 0) {  
        continue;  
    }  
    System.out.println("Zahl ist: " + i);  
}
```

Ausgabe ?





## Kap. 2.6

# continue

- Vorzeitiges Beenden einer Schleifeniteration durch **continue**-Befehl, weiter mit nächster Iteration
- vereinfacht oft komplizierten Code, prinzipiell immer durch `if..else` ersetzbar

```
Zahl ist: 1  
Zahl ist: 3  
Zahl ist: 5  
Zahl ist: 7  
...  
Zahl ist: 99
```

Ausgabe



# Cut: Q&A



# Klassen, quick & dirty



# Was ist eine Klasse?

- Klassen definieren neue Datentypen (wie ein C-struct)
- Klassen enthalten Code (**Methoden**) und **Daten (Attribute)**, die thematisch zusammengehören
- Von jeder Klasse können beliebig viele **Instanzen** erzeugt werden



## Kap. 5.1, 5.4

# Was enthält eine Klasse?

```
public class V1 {  
  
    public int attribut ;  
  
    public V1() {  
        this.attribut = 0 ;  
    }  
  
    public V1(int k) {  
        this.attribut = k ;  
    }  
  
    public int getAttribut () {  
        return this.attribut ;  
    }  
}
```

} Attribut(e)

} Konstruktor(en)

} Methode(n)





# Was enthält eine Klasse?

- Definiert u.a.
  - einen oder mehrere **Konstruktoren**: spezielle Methoden, aufgerufen bei Instanziierung
  - **Methoden** (`main`-Methode wird bei Programmstart aufgerufen)
  - **Attribute**, potentiell mit Startwerten
  - Zugriffsrechte für Klassen, Methoden und Attribute





# OOP: Klasseninstanziierung



- Instanziierung mit `new`: erzeugt unabhängige **Instanz** der Klasse

```
V1 ref1 = new V1() ;  
V1 ref2 = new V1(5) ;
```



- Bei Instanziierung wird der Konstruktor aufgerufen, ggf. mit Argumenten

# Instanzen

- Analogie: Klasse ist Stempel, Instanz ist Abdruck
  - gibt nur einen Stempel aber viele Abdrücke
  - Abdrücke können verschieden sein (Attribute sind unterschiedlich)
  - Abdrücke sind unabhängig voneinander (da jeder eigenen Speicherbereich hat)







## Kap. 5.1, 5.4

# Instanzen

- Etwas formaler:
  - Von einer Klasse können beliebig viele Instanzen existieren
  - Änderungen in einer Instanz haben keinen Einfluss auf die anderen Instanzen
  - die Methoden können die Referenz **this** nutzen, um auf eigene Instanz zuzugreifen
  - Attribute/Methoden unterliegen **Zugriffsrechten**
  - spezielle Methoden: **Konstruktoren**





# Zugriff auf Methoden und Attribute

- Es wird stets der .-Operator benutzt
- Zugriffsrechte werden geprüft!

```
V1 ref1 = new V1() ;  
V1 ref2 = new V1(5) ;  
  
int ref1Attr = ref1.getAttribute() ;  
int ref2Attr = ref2.attribut ;  
System.out.println(ref1Attr+" "+ref2Attr) ;
```





## Kap. 5.2



# Zugriffsrechte

- Zugriffsrechte (für Attribute/Methoden/Klassen):
  - `public`: jeder kann benutzen/aufrufen
  - `private`: nur Methoden der Klasse können benutzen/aufrufen
  - `package-public`: nur Klassen im selben Paket können benutzen/aufrufen
  - `protected`: nur Methoden dieser und abgeleiteter Klassen können benutzen/aufrufen





# Beispiel für Deklaration, Instanziierung & Benutzung einer Klasse

- → V1.java (im E-Learning)



# CUT: Q&A



# ARBEIT

- Bitte bearbeiten Sie das „Vorlesungs-Quiz“ im E-Learning!



# Neue Elemente in Java



# Wichtige Java-Elemente

- Konsoleneingabe/ausgabe
- Java-Arrays
- Java-Strings
- `boolean` statt `bool`
- Beispieldatei im E-Learning:  
`JavaArraysStrings.java`





# Konsolenein/ausgabe

- Eingabe: über Instanz der Klasse  
`java.util.Scanner`
- Sehr leistungsfähig, kann von der Konsole oder aus Daten/Puffern lesen
- Wir benutzen nur die Methode `nextLine()`



# Konsolenein/ausgabe

- Ausgabe: u. A. über die Methoden  
`System.out.print()`  
`System.out.println()`
- Beide Methoden erwarten ein `String`-Argument



## Kap. 5.5

# Die Klasse String

- Instanziierung:

```
String s = "Hallo" ;
```

```
String s = new String("Hallo") ;
```

- Wichtige Methoden: `charAt()`, `length()`, `find()`
- Verkettung mit „+“-Operator:

```
int i = 5 ;
```

```
System.out.println("Hallo " + i) ;
```



## Kap. 4

# Java-Arrays

- Syntax ist etwas anders als in C:
  - 1D Arrays:

```
int[] intArray = new int [3] ;  
intArray[2] = 0 ;
```
  - 2D Arrays: Arrays von Arrays

```
int[] [] intArray2D = new int [3][4] ;  
intArray[2][1] = 9 ;
```
- Alle Arrays sind Instanzen und haben ein Attribut `length`



# Java-boololeans

- In C existiert der Typ `bool` (eigentlich `unsigned char`) mit Werten 0 und 1
- In Java existiert der spezielle Typ `boolean` mit Werten `false` und `true`
- Alle Java-Bedingungen und Schleifen arbeiten mit `boolean`-Werten



# Beispiele

- `JavaArraysStringsBooleans.java` im E-Learning



# CUT: Q&A