

# Java

## Prozedurale Programmierung & OOP I

---

Marcus Köhler

5. November 2018

Java-Kurs

1. Nachtrag: Arrays
2. Prozedurale Programmierung
  - If-Else
  - while
  - for
  - Funktionen
3. OOP in Java
  - Grundlagen
  - Methoden
  - Rückgabewerte
  - Der Konstruktor

## Nachtrag: Arrays

---

# Arrays

Ein *Array* (oder auch „Feld“) ist eine Menge fester Größe von gleichartigen Datentypen bzw. Objekten.

```
1 int[] i_arr = {1, 2, 4, 8};  
2 char[] c_arr = {'c', 'h', 'a', 'r'};  
3 float[] f_arr = new float[5]; //empty array  
4 String[] s_arr; //Declaration  
5 s_arr = new String[10]; //Initialization
```

Die Elemente eines *Arrays* sind mit einem Index versehen, über den man auf sie zugreifen kann;

Java-Arrays sind 0-indexed(sprich: „zero-indexed“), d.h. das erste Element hat den Index 0.

```
1 System.out.println(i_arr[0]); //prints '1'  
2 System.out.println(c_arr[3]); //prints 'r'
```

# Prozedurale Programmierung

---

- Verzweigung

- Verzweigung  
if, else, else if



- Verzweigung  
if, else, else if  
switch-case

- Verzweigung  
if, else, else if  
switch-case
- Schleifen

- Verzweigung  
if, else, else if  
switch-case
- Schleifen  
while, do-while

- Verzweigung  
if, else, else if  
switch-case
- Schleifen  
while, do-while  
for, for-each

## if & else

```
1 if(condition) {  
2     // do something if condition is true  
3 } else if(another condition){  
4     // do if "else if" condition is true  
5 } else {  
6     // otherwise do this  
7 }
```

## if & else

```
1 public class IfElseExample {  
2     public static void main(String[] args) {  
3         int myNumber = 5;  
4  
5         if(myNumber == 3) {  
6             System.out.println("Strange number");  
7         } else if(myNumber == 2) {  
8             System.out.println("Unreachable code");  
9         } else {  
10            System.out.println("Will be printed");  
11        }  
12    }  
13 }
```

# Bedingungen?

Bedingungen sind Statements, welche einen `boolean`(Wahrheitswert) darstellen. Logische Verknüpfungen werden folgendermaßen dargestellt:

- `!` Negation

# Bedingungen?

Bedingungen sind Statements, welche einen `boolean`(Wahrheitswert) darstellen. Logische Verknüpfungen werden folgendermaßen dargestellt:

- `!` Negation
- `==` Gleichheit



# Bedingungen?

Bedingungen sind Statements, welche einen `boolean`(Wahrheitswert) darstellen. Logische Verknüpfungen werden folgendermaßen dargestellt:

- `!` Negation
- `==` Gleichheit
- `!=` Ungleichheit

# Bedingungen?

Bedingungen sind Statements, welche einen `boolean`(Wahrheitswert) darstellen. Logische Verknüpfungen werden folgendermaßen dargestellt:

- `!` Negation
- `==` Gleichheit
- `!=` Ungleichheit
- `>` Größer

# Bedingungen?

Bedingungen sind Statements, welche einen `boolean`(Wahrheitswert) darstellen. Logische Verknüpfungen werden folgendermaßen dargestellt:

- `!` Negation
- `==` Gleichheit
- `!=` Ungleichheit
- `>` Größer
- `>=` Größer/Gleich

# Bedingungen?

Bedingungen sind Statements, welche einen `boolean`(Wahrheitswert) darstellen. Logische Verknüpfungen werden folgendermaßen dargestellt:

- `!` Negation
- `==` Gleichheit
- `!=` Ungleichheit
- `>` Größer
- `>=` Größer/Gleich
- `<` Kleiner

# Bedingungen?

Bedingungen sind Statements, welche einen `boolean`(Wahrheitswert) darstellen. Logische Verknüpfungen werden folgendermaßen dargestellt:

- `!` Negation
- `==` Gleichheit
- `!=` Ungleichheit
- `>` Größer
- `>=` Größer/Gleich
- `<` Kleiner
- `<=` Kleiner/Gleich

# Bedingungen?

Bedingungen sind Statements, welche einen `boolean`(Wahrheitswert) darstellen. Logische Verknüpfungen werden folgendermaßen dargestellt:

- `!` Negation
- `==` Gleichheit
- `!=` Ungleichheit
- `>` Größer
- `>=` Größer/Gleich
- `<` Kleiner
- `<=` Kleiner/Gleich

*Anmerkung:* Mehrere Bedingungen können mit `&&` (logisches UND) oder `||` (logisches ODER) verkettet werden. Die Reihenfolge ist hierbei in einigen Fällen wichtig!

# while

```
1 while(condition) {  
2     // do code while condition is true  
3 }
```

## while Beispiel

```
1 public class WhileExample {  
2  
3     public static void main(String[] args) {  
4         int a = 0;  
5         while(a <= 10) {  
6             System.out.println(a);  
7             a++; // Otherwise you would get an endless loop  
8         }  
9     }  
10 }
```



```
1 for(initial value, condition, change) {  
2     // do code while condition is true  
3 }
```

# for Beispiel

```
1 public class ForExample {  
2  
3     public static void main(String[] args) {  
4         for(int i = 0; i <= 10; i++) {  
5             System.out.print("na ");  
6         }  
7         System.out.println("BATMAN!");  
8     }  
9 }
```

Das vorangegangene Programm kann man auch mit einer `while`-Schleife umsetzen:

```
1 public class ForWhileEx{
2     public static void main(String[] args) {
3         int i = 0;
4         while(i <= 10) {
5             System.out.println("na ");
6             i++;
7         }
8         System.out.println("BATMAN!");
9     }
10 }
```

*Funktionen* sind sozusagen mehrmals nutzbare Code-Snippets:

```
public int sq(int n) {  
    return n*n;  
}
```

*Funktionen* sind sozusagen mehrmals nutzbare Code-Snippets:

```
public int sq(int n) {  
    return n*n;  
}
```

Funktionen bestehen aus einer *Signatur*

*Funktionen* sind sozusagen mehrmals nutzbare Code-Snippets:

```
public int sq(int n) {  
    return n*n;  
}
```

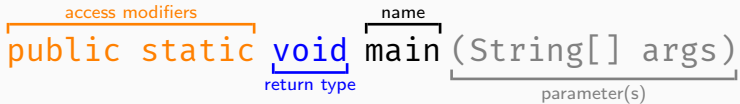
Funktionen bestehen aus einer *Signatur* und einem *Körper*.

*Funktionen* sind sozusagen mehrmals nutzbare Code-Snippets:

```
public int sq(int n) {  
    return n*n;  
}
```

Funktionen bestehen aus einer *Signatur* und einem *Körper*.  
Im OOP werden Funktionen als *Methoden* bezeichnet.

# Funktionssignaturen

  
The diagram shows the function signature `public static void main (String[] args)` with annotations above and below it. Above `public static` is an orange bracket labeled "access modifiers". Above `main` is a black bracket labeled "name". Below `void` is a blue bracket labeled "return type". Below `(String[] args)` is a grey bracket labeled "parameter(s)".

```
public static void main (String[] args)
```

Die access modifier regeln, wer die Funktion aufrufen darf.



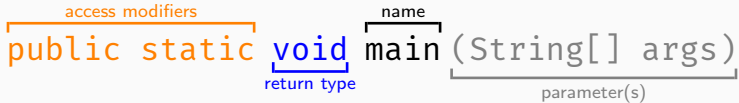
# Funktionssignaturen

The diagram shows the function signature `public static void main (String[] args)` with annotations above and below it. Above `public static` is an orange bracket labeled "access modifiers". Above `main` is a black bracket labeled "name". Below `void` is a blue bracket labeled "return type". Below `(String[] args)` is a grey bracket labeled "parameter(s)".

```
public static void main (String[] args)
```

Die `access modifier` regeln, wer die Funktion aufrufen darf.  
Der `return type` (auch: Rückgabewert) gibt an, welchen Typ das „Ergebnis“ hat.

# Funktionssignaturen

The diagram shows the function signature `public static void main (String[] args)` with three annotations: an orange bracket above `public static` labeled "access modifiers", a blue bracket below `void` labeled "return type", and a black bracket above `main` labeled "name". A black bracket below `(String[] args)` is labeled "parameter(s)".

```
public static void main (String[] args)
```

Die `access modifier` regeln, wer die Funktion aufrufen darf.

Der `return type`(auch: Rückgabewert) gibt an, welchen Typ das „Ergebnis“ hat.

Die `Parameter`(auch: Argumente) sind die Daten, die die Methode braucht bzw. erwartet.

# OOP in Java

---

# Objektorientierte Programmierung

# Class Student

```
1 public class Student {  
2  
3     // Attributes  
4     private String name;  
5     private int matriculationNumber;  
6  
7     // Methods  
8     public void setName(String name) {  
9         this.name = name;  
10    }  
11  
12    public int getMatriculationNumber() {  
13        return matriculationNumber;  
14    }  
15 }
```

# Ein Objekt erstellen

Ein primitiver Datentyp wird so erstellt:

```
1 int a; // declare a  
2 a = 273; // assign 273 to a
```

# Ein Objekt erstellen

Ein primitiver Datentyp wird so erstellt:

```
1 int a; // declare a
2 a = 273; // assign 273 to a
```

Bei Objekten funktioniert das ähnlich:

```
1 Student example = new Student();
2 // create an instance of Student
```

# Ein Objekt erstellen

Ein primitiver Datentyp wird so erstellt:

```
1 int a; // declare a
2 a = 273; // assign 273 to a
```

Bei Objekten funktioniert das ähnlich:

```
1 Student example = new Student();
2 // create an instance of Student
```

Ein Objekt einer *Klasse* wird auch *Instanz* genannt. Eine Variable, die mit einem Objekt belegt ist, heißt *Referenz*.



# Eine Methode aufrufen

```
1 public class Student {  
2  
3     private String name;  
4  
5     public String getName() {  
6         return name;  
7     }  
8  
9     public void setName(String newName) {  
10        name = newName;  
11    }  
12 }
```

Die Klasse Student (und damit das Objekt) hat zwei Methoden:  
`void printTimetable()` und `void printName()`.

# Eine Methode aufrufen

```
1 public class Main {  
2  
3     public static void main(String[] args) {  
4         Student example = new Student(); // creation  
5         example.setName("Jane"); // method call  
6         String name = example.getName();  
7         System.out.println(name); // Prints "Jane"  
8     }  
9 }
```

Man kann eine Methode eines Objekts nach der Instanzierung mittels `Referenz.methodName()`; aufrufen.

# Eine Methode aufrufen

```
1 public class Student {  
2  
3     private String name;  
4  
5     public void setName(String newName) {  
6         name = newName;  
7         printName();    // Call own method  
8         this.printName(); // Or this way  
9     }  
10  
11     public void printName() {  
12         System.out.println(name);  
13     }  
14 }
```

Ein Objekt kann seine eigenen Methoden mit `methodName()`; oder `this.methodName()`; aufrufen.

# Methoden mit Parametern

```
1 public class Calc {  
2  
3     public void add(int summand1, int summand2) {  
4         System.out.println(summand1 + summand2);  
5     }  
6  
7     public static void main(String[] args) {  
8         int summandA = 1;  
9         int summandB = 2;  
10        Calc calculator = new Calc();  
11        System.out.print("1 + 2 = ");  
12        calculator.add(summandA, summandB);  
13        // prints: 3  
14    }  
15 }
```

# Methoden mit Rückgabewerten

Eine Methode ohne Rückgabewert hat den Typ `void`:

```
1 public void add(int summand1, int summand2) {  
2     System.out.println(summand1 + summand2);  
3 }
```

Methoden mit (passenden) Rückgabewerten können Variablen zugewiesen werden:

```
1 int a, b, aTimesB;  
2 int a = 5;  
3 int b = 7;  
4 int aTimesB = mul(a, b);
```

# Konstruktoren

Ein *Konstruktor* ist eine spezielle Methode, welche den „Urzustand“ eines Objekts definiert.

Er wird aufgerufen, sobald ein Objekt instanziiert wird:

```
1 public class Calc {  
2     private int summand1;  
3     private int summand2;  
4  
5     public Calc() { // Constructor  
6         summand1 = 0;  
7         summand2 = 0;  
8     }  
9  
10    public static void main(String[] args) {  
11        Calc calc = new Calc(); //Constructor is called  
12    }  
13 }
```

# Konstruktoren mit Parametern

Ein Konstruktor kann, wie eine „normale“ Methode, auch Parameter übergeben bekommen.

Das ist nötig, wenn Anfangswerte etc. angegeben werden:

```
1 public class Calc {  
2     private int summand1;  
3     private int summand2;  
4  
5     public Calc(int x, int y) {  
6         summand1 = x;  
7         summand2 = y;  
8     }  
9 }
```

```
1 [...]  
2 Calc myCalc = new Calc(7, 9);
```

- Datenkapselung
- Vererbung
- Polymorphie
- Das `this` Keyword