



Objektorientierte Programmierung mit Java – Woche 2

openHPI-Java-Team

Hasso-Plattner-Institut



Wiederholung Woche 1

openHPI-Java-Team

Hasso-Plattner-Institut

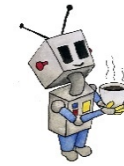


```
1  class HelloPaco{  
2      public static void main(String[] args){  
3          System.out.println("Hallo Paco");  
4      }  
5  }
```

Ausgabe:

Hallo Paco

- Ausgabefunktion über System.out.println();
- Programm startet in main()



```
1  class Robot{
2      String name;
3      int numberOfEyes;
4
5      void speak(){
6          //...
7      }
8      void walk(){
9          //...
10     }
11 }
```

} Name der Klasse

} Attribute

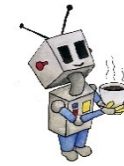
} Methoden

- Klassen sind Vorlagen für Konstrukte der realen Welt
- Klassen haben Attribute und Methoden
- Objekte sind Instanzen von Klassen



- Container für Daten
- Benötigen einen eindeutigen Bezeichner
- Format: `<Datentyp> <Bezeichner> = <Wert>;`

Name	Art	Beispiele
String	Text, Zeichenkette	<code>String name = "Robin";</code>
char	Buchstabe, Zeichen	<code>char country = 'd';</code> <code>char cedille = 'ç';</code>
int	Ganzzahl	<code>int age = 2;</code> <code>int truth = -42;</code>
double	Kommazahl	<code>double speed = 98.7;</code>



```
<Rückgabety> <Methodenbezeichner>(){  
    //mehr Quellcode  
    return <Rückgabewert>;  
}
```

- Rückgabewert muss dem Rückgabety entsprechen
- **return** gibt den Rückgabewert an die aufrufende Methode zurück



Parameter

openHPI-Java-Team

Hasso-Plattner-Institut



Datentyp Parameterbezeichner

```
1  void speak(String input){  
2      System.out.println(input);  
3  }
```

Parameter

- Variablen von Methoden, die beim Methodenaufruf befüllt werden
- Syntax:
 <Typ><Methodenbezeichner>(<Datentyp><Parameterbezeichner>)
- Zugriff innerhalb der Methode über den Parameterbezeichner



Aufruf von Methoden mit Parametern

```
1 void speak(String input){  
2     System.out.println(input);  
3 }  
4 void greet(){  
5     speak("Hallo");  
6 }
```

Parameter

Methodenname

Argument

- Syntax: <Methodenname>(<Argument>);
- Argumente werden beim Methodenaufruf übergeben
- **Parameter**: Bei der Methodendefinition
- **Argument**: Übergabewert beim Aufruf der Methode
- **Achtung!** Datentypen müssen übereinstimmen



```
1 void speak(String input){  
2     System.out.println(input);  
3 }  
4 void greet(){  
5     String text = "Hallo Paco";  
6     speak(text);  
7 }
```

- Argument kann Wert oder eine andere Variable sein
- Variablenname (`text`) muss nicht gleich dem Parameternamen in der Methodendefinition (`input`) sein
- **Achtung!** Datentypen müssen übereinstimmen



Mehrere Parameter

```
1 double add(int a, int b, double c){  
2     return a+b+c;  
3 }  
4 void calculate(){  
5     System.out.println(add(3,4,1.1));  
6 }
```

Parameter


- Übergeben von mehreren Parametern möglich
 - Werden durch Kommata getrennt
- Können unterschiedliche Datentypen haben
- **Achtung!** Reihenfolge muss exakt gleich sein
- Anzahl der Argumente muss der Anzahl der Parameter entsprechen
- Leere Klammern: Methode nimmt keine Argumente



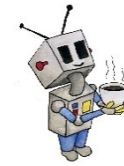
- Auch "Scope" genannt
- Schlüsselwörter **überall** verwendbar
- Selbstgewählte Bezeichner **nicht überall** verwendbar
- Parameter nur innerhalb von Methode bekannt
- Grundsätzlich gilt: geschweifte Klammern begrenzen einen Gültigkeitsbereich



```
1 void speak(String input){  
2     System.out.println(input);  
3 }  
4 void say(){  
5     System.out.println(input);  
6 }
```



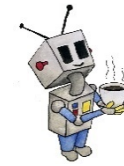
- Parameter nur innerhalb der Methode definiert



```
1 void speak(String input){  
2                       
3     System.out.println(input);  
4 }  
5 void greet() {  
6     String input = "Hallo";  
7 }
```

- Parameter nur innerhalb der Methode `speak(...)` gültig
- `input` ist außerhalb von `speak(...)` nicht bekannt
- gleicher Bezeichner kann außerhalb der Methode (z.B. in `greet()`) an andere Variable vergeben werden

Gültigkeitsbereiche: ein verwirrendes Beispiel



```
1 class Name {  
2     void name(String name){  
3         name = name + "name";  
4         System.out.println(name);  
5     }  
6     public static void main(String[] argv){  
7         Name name = new Name();  
8         String namenname = "name";  
9         name.name(namenname);  
10    }  
11 }
```

- Valider Code (leider)
- Als Hilfe für Verständlichkeit individuelle Bezeichner nehmen



Booleans: Wahr oder Falsch?

openHPI-Java-Team
Hasso-Plattner-Institut



- In Informatik wird viel über “Wahr oder Falsch”/“0 oder 1” entschieden
- Datentyp: `boolean` (kann entweder `true` oder `false` sein)
- Verwendet um Bedingungen zu überprüfen
- Syntax für die Definition:

```
boolean mybool = true;
```

- **Achtung!** Java interpretiert Zahlenwerte (z.B. 0 oder 1) nicht als `boolean`



- Einsatz: Wahrheitswert tauschen
- Symbol: !
- Ergebnis: wahr, wenn Ausdruck falsch

A	!A
Falsch	Wahr
Wahr	Falsch



- Einsatz: zwei Bedingungen müssen erfüllt sein
- Symbol: &&
- Ergebnis: wahr, wenn beide Ausdrücke wahr

A	B	A && B
Falsch	Falsch	Falsch
Falsch	Wahr	Falsch
Wahr	Falsch	Falsch
Wahr	Wahr	Wahr



- Einsatz: mindestens eine von zwei Bedingungen muss erfüllt sein
- Symbol: `||`
- Ergebnis: wahr, wenn mindestens ein Ausdruck wahr

A	B	A B
Falsch	Falsch	Falsch
Falsch	Wahr	Wahr
Wahr	Falsch	Wahr
Wahr	Wahr	Wahr



- Einsatz: die Bedingungen gleich erfüllt sein müssen
- Symbol: `==`
- Ergebnis: wahr, wenn beide Ausdrücke wahr oder beide Ausdrücke falsch

A	B	A == B
Falsch	Falsch	Wahr
Falsch	Wahr	Falsch
Wahr	Falsch	Falsch
Wahr	Wahr	Wahr

- **Achtung!** Objekte (wie Strings) werden mit `equals()` verglichen.



- Einsatz: beide Bedingungen müssen unterschiedlich erfüllt sein
- Symbol: `!=`
- Ergebnis: wahr, wenn ein Ausdruck wahr und der andere falsch

A	B	A != B
Falsch	Falsch	Falsch
Falsch	Wahr	Wahr
Wahr	Falsch	Wahr
Wahr	Wahr	Falsch



- Definierte Reihenfolge bei Auswertung von Operationen
- Negation vor UND vor ODER

Beispiele

- $! A \ \&\& \ B \iff (!A) \ \&\& \ B$
- $A \ || \ B \ \&\& \ C \iff A \ || \ (B \ \&\& \ C)$
- $A \ \&\& \ B \ || \ !C \iff (A \ \&\& \ B) \ || \ (!C)$



- Komplexere Ausdrücke durch Aneinanderreihen von Operatoren
- Hilfsspalten für Teilausdrücke hilfreich
- Beispiel: `A || B && C`

A	B	C	A B && C
Falsch	Falsch	Falsch	?
Falsch	Falsch	Wahr	?
Falsch	Wahr	Falsch	?
Falsch	Wahr	Wahr	?
Wahr	Falsch	Falsch	?
Wahr	Falsch	Wahr	?
Wahr	Wahr	Falsch	?
Wahr	Wahr	Wahr	?



Operatoren verknüpfen

- Komplexere Ausdrücke durch Aneinanderreihen von Operatoren
- Hilfsspalten für Teilausdrücke hilfreich → **D = B && C**
- Beispiel: A || B && C

A	B	C	D = B && C	A D
Falsch	Falsch	Falsch	Falsch	?
Falsch	Falsch	Wahr	Falsch	?
Falsch	Wahr	Falsch	Falsch	?
Falsch	Wahr	Wahr	Wahr	?
Wahr	Falsch	Falsch	Falsch	?
Wahr	Falsch	Wahr	Falsch	?
Wahr	Wahr	Falsch	Falsch	?
Wahr	Wahr	Wahr	Wahr	?



Operatoren verknüpfen

- Komplexere Ausdrücke durch Aneinanderreihen von Operatoren
- Hilfsspalten für Teilausdrücke hilfreich $\rightarrow D = B \ \&\& \ C$
- Beispiel: $A \ || \ B \ \&\& \ C$

A	B	C	D	$A \ \ D$
Falsch	Falsch	Falsch	Falsch	Falsch
Falsch	Falsch	Wahr	Falsch	Falsch
Falsch	Wahr	Falsch	Falsch	Falsch
Falsch	Wahr	Wahr	Wahr	Wahr
Wahr	Falsch	Falsch	Falsch	Wahr
Wahr	Falsch	Wahr	Falsch	Wahr
Wahr	Wahr	Falsch	Falsch	Wahr
Wahr	Wahr	Wahr	Wahr	Wahr

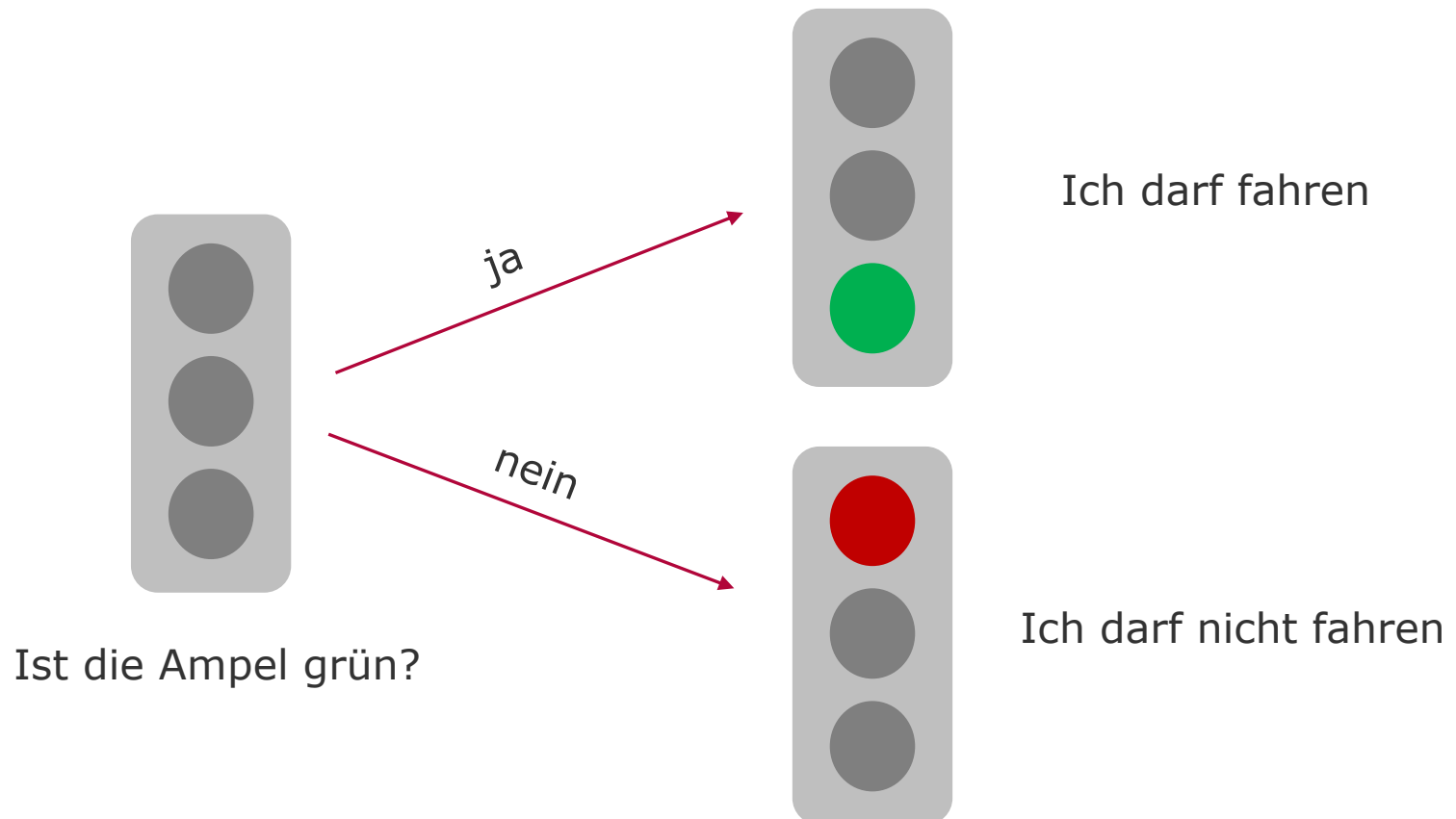


Kontrollstrukturen: Verzweigung (if/else)

openHPI-Java-Team
Hasso-Plattner-Institut



- Auswahl zwischen zwei Szenarien („wenn ..., dann ...“)
- Basiert auf Bedingung, die entweder wahr oder falsch ist (Boolean)





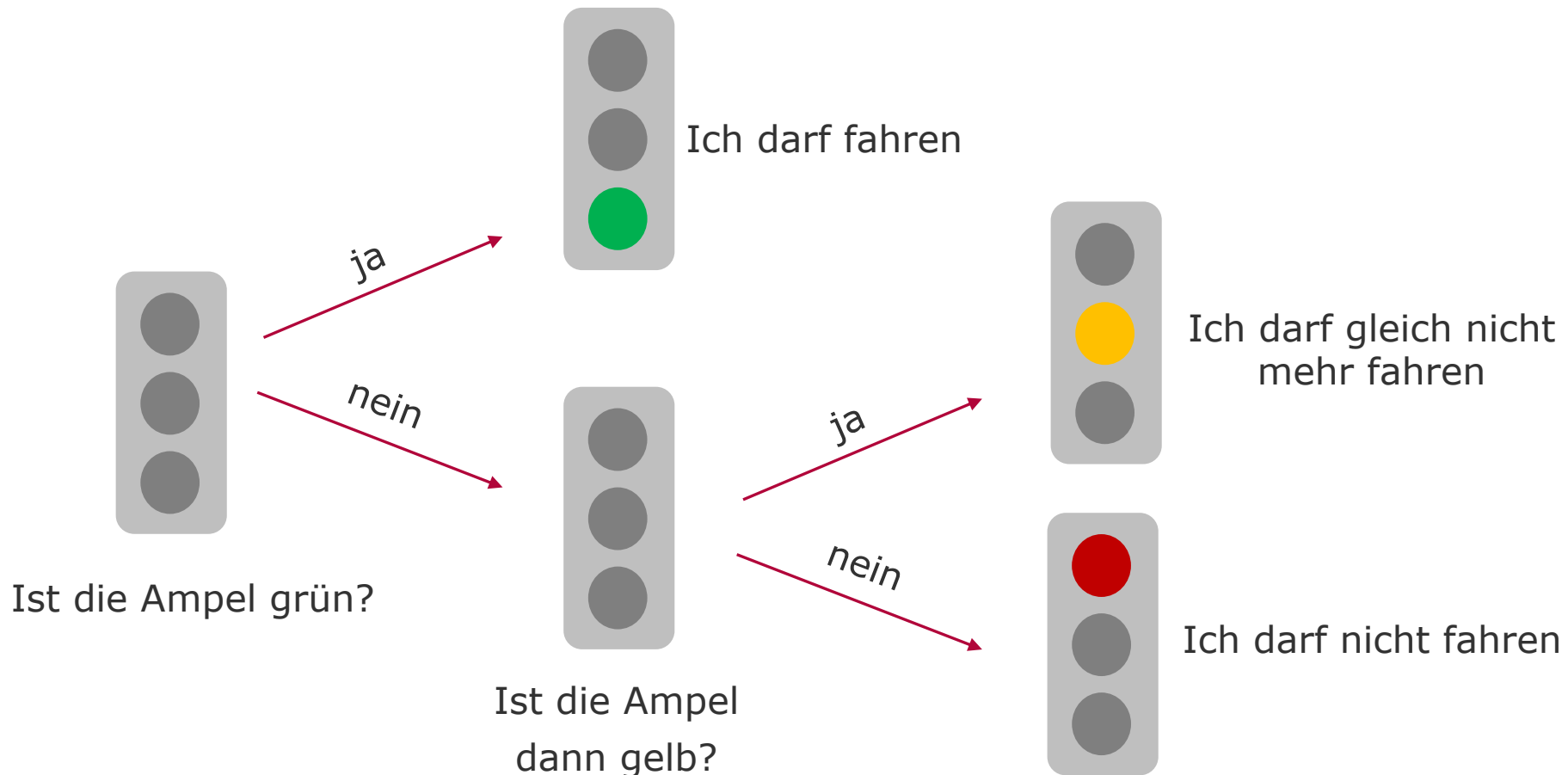
if/else im Code

```
1  if (robin.isBatteryLow()){  
2      robin.charge();  
3  } else {  
4      robin.doThings();  
5  }
```

- Hinter **if** in runden Klammern Bedingung
- in { } was passieren soll, wenn Bedingung erfüllt
- Nach **else** in { } was passieren soll, wenn Bedingung **nicht** erfüllt
- **else** kann auch weg gelassen werden
- **Wichtig!** hinter **else** nicht nochmal runde Klammern ()!



- Wenn eine Bedingung nicht erfüllt ist, aber zweite Bedingung gilt





else if (2/2)

```
1  if (robin.isBatteryLow() && robin.isInDockingStation()) {  
2      robin.chargeBattery();  
3  } else if (robin.isBatteryLow()) {  
4      robin.driveToDockingStation();  
5  } else {  
6      robin.doThings();  
7  }
```

- Programm wird von oben nach unten abgearbeitet
- → **else if** nur ausgeführt, wenn **if**-Bedingung falsch ist
- → **else** nur ausgeführt, wenn auch **else if**-Bedingungen falsch



Kontrollstrukturen: Schleifen

openHPI-Java-Team
Hasso-Plattner-Institut



- Englisch: loop
- Gleichen Code mehrmals ausführen, aber nur einmal schreiben
- Besteht aus **Schleifenkopf** mit Schleifenbedingung und **Schleifenrumpf**
- Arten:
 - Zählschleife **for** (z.B. Dreimal ausführen)
 - Kopfgesteuerte Schleife **while** (solange..., tue...)
 - Fußgesteuerte Schleife **do while** (wie **while** und immer mindestens einmal ausgeführt)
 - ...

Zählschleife (1/3)

for



```
1 for (int i = 0; i < 3; i++) {  
2  
3     //dies ist eine Zählschleife  
4     //sie wird 3 mal ausgeführt  
5  
6  
7 }
```

Schleifenbedingung

Schleifenkopf

Schleifenrumpf

Syntax des Schleifenkopfes:

for (<Zählerinitialisierung>; <Abbruchbedingung>; <Schrittweite>)

for (int i = 0; i < 3; i++)



```
1 for (int i = 0; i < 3; i++) {  
2  
3     //dies ist eine Zählschleife  
4     //sie wird 3 mal ausgeführt  
5  
6  
7 }
```

Schleifenbedingung

Schleifenkopf

Schleifenrumpf

Erklärung Schleifenkopf

`for (int i = 0; i < 3; i++)`

- Ganzzahl `i` wird zu Beginn auf 0 gesetzt (`int i = 0`)
- Vor jedem Schleifendurchlauf wird geschaut, wie groß `i` ist (`i < 3`)
- Nach jedem Schleifendurchlauf wird `i` um eins erhöht (`i++`)

Zählschleife (3/3)

for



```
1 for (int i = 0; i < 3; i++) {  
2     //dies ist ein Beispiel  
3     System.out.println("Zeile "+ i);  
4 }
```

Schleifenbedingung

Schleifenkopf

Schleifenrumpf

Ausgabe:

Zeile 0

Zeile 1

Zeile 2

kopfgesteuerte Schleife (1/3)

while



```
1 while ( i < j ){  
2  
3     //Dies ist eine  
4     //kopfgesteuerte Schleife  
5     //i oder j sollte verändert  
6     //werden  
7 }
```

Schleifenbedingung

Schleifenkopf

Schleifenrumpf

- Bei jedem Schleifendurchlauf Bedingung geprüft
- Bedingung kann beliebig sein (muss nicht `i < j` sein)
- Solange Bedingung erfüllt ist, wird Schleifenrumpf ausgeführt
- **Achtung!** Endlosschleifen möglich

kopfgesteuerte Schleife (2/3)

while



```
1 int i = 0;
2 while ( i < 3 ){
3     //So könnte man die Funktion
4     //einer for-Schleife mit einer
5     //while-Schleife darstellen
6     i++;
7 }
```

Schleifenbedingung

Schleifenkopf

Schleifenrumpf

- Auch mit **while**-Schleife gleiche Funktion wie mit einer **for**-Schleife möglich
- Initialisierung vor der Schleife, Veränderung der Variable in der Schleife

kopfgesteuerte Schleife (3/3)

while



```
1 int i = 0;
2 while ( i < 3 ){
3     //Dies ist das gleiche Beispiel
4     //wie bei der for-Schleife
5     System.out.println("Zeile "+ i);
6     i++;
7 }
```

Schleifenbedingung

Schleifenkopf

Schleifenrumpf

Ausgabe:

Zeile 0

Zeile 1

Zeile 2



```
1 do {  
2  
3     //Dies ist eine  
4     //fußgesteuerte Schleife  
5  
6 } while ( i < j );
```

} Schleifenkopf

} Schleifenrumpf

↑ Schleifenbedingung

- Nach jedem Schleifendurchlauf Bedingung geprüft
- Solange Bedingung erfüllt wird Schleifenrumpf nochmal ausgeführt
- Im Endeffekt wie **while** nur mindestens einmal ausgeführt
- **Achtung!** Endlosschleifen möglich
- Hinweis: in Praxis nur selten verwendet



Primitive Arrays

openHPI-Java-Team

Hasso-Plattner-Institut



Arrays - Erklärung

- Container für gleichartige „Dinge“ → Werte gleichen Datentyps
- Gekennzeichnet durch []
- `int[] numbers` ist ein Array von Integer → nur Integer können dort „reingepackt“ werden
- Garage (im weitesten Sinne) Array für Autos
 - Stellplätze durchnummeriert
 - Zuweisung und Zugriff über eindeutige Stellplatznummer
 - Definierte Anzahl von Stellplätzen (nicht unendlich groß)

Arrays: Deklarieren und initialisieren (1/2)



```
1  int[] numbers;  
2  numbers = new int[5];  
3  
4  
5  
6
```

- Deklaration: „Da gibt es jetzt dieses Objekt mit diesem Namen“
- Initialisierung: „Das Objekt kriegt jetzt auch einen Wert“

- Kann auch gleichzeitig geschehen:

```
    int[] numbers = new int[5];
```

- `<Datentyp>[] <arrayname> = new <gleicher Datentyp>[<Größe>];`

Arrays: Deklarieren und initialisieren (2/2)



```
1  int[] numbers = new int[5];  
2  numbers[0] = 6;  
3  numbers[1] = 33;  
4  numbers[2] = 9;  
5  numbers[3] = 0;  
6  numbers[4] = 503;
```

- Kann auch gleichzeitig mit Deklaration geschehen
`int[] numbers = {6,33,9,0,503};`
- Initialisierung mit `new` setzt alle Felder des Arrays auf Defaultwerte
- **Achtung!** In Java fängt man mit der 0 an zu zählen
- **Achtung!** In Java können primitive Arrays **nicht** ihre Größe verändern
→ nicht dynamisch weitere Elemente anfügen



```

                                0         1         2
1  String[] words = {"Hallo", "Huhu", "Na?"};
2
3  for( int i = 0; i < words.length; i++){
4      System.out.println(words[i]);
5  }
```

- Anzahl der Elemente eines Arrays mit <Arrayname>.length abfragen

Ausgabe:

Hallo ← Ausgabe für i = 0

Huhu ← Ausgabe für i = 1

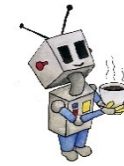
Na? ← Ausgabe für i = 2



Konstruktoren und „this“

openHPI-Java-Team

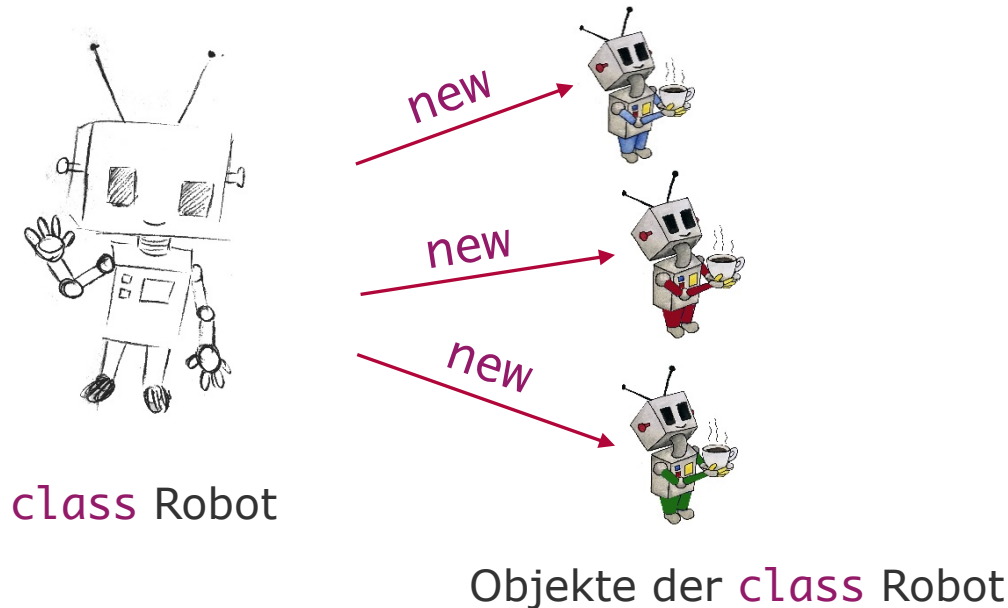
Hasso-Plattner-Institut

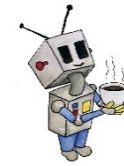


- Was macht eigentlich **new** in:

```
int[ ] numbers = new int[5]; ?
```

- **new** ruft **Konstruktor** der Klasse auf, die danach steht
- Konstrukturen **erzeugen** Objekte einer Klasse

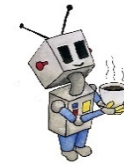




Syntax eines Konstruktors (1/4)

```
1  class Robot{  
2      String name;  
3      int numberOfEyes;  
4  
5      Robot(){  
6          this.name = "Dies ist ein Standard-Name";  
7      }  
8  }
```

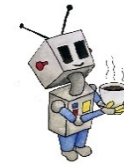
- Kein Rückgabewert
- Gleicher Bezeichner wie Klasse
- **this** wird verwendet, um auf Attribute dieser Klasse zuzugreifen
- Kann **keinen**, einen oder mehrere Parameter haben
- Beispielaufruf: Robot **robin** = new Robot();



Syntax eines Konstruktors (2/4)

```
1  class Robot{  
2      String name;  
3      int numberOfEyes;  
4  
5      Robot(String userDefinedName){  
6          this.name = userDefinedName;  
7      }  
8  }
```

- Kein Rückgabewert
- Gleicher Bezeichner wie Klasse
- **this** wird verwendet, um auf Attribute dieser Klasse zuzugreifen
- Kann keinen, **einen** oder mehrere Parameter haben
- Beispielaufruf: Robot **robert** = new Robot("Robert");

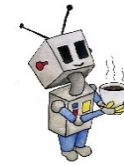


Syntax eines Konstruktors (3/4)

```
1  class Robot{
2      String name;
3      int numberOfEyes;
4
5      Robot(String name){
6          this.name = name;
7      }
8  }
```

← Gültige Bezeichnung!

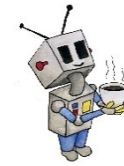
- Kein Rückgabewert
- Gleicher Bezeichner wie Klasse
- **this** wird verwendet, um auf Attribute dieser Klasse zuzugreifen
- Kann keinen, **einen** oder mehrere Parameter haben
- Beispielaufruf: Robot **robert** = new Robot("Robert");



Syntax eines Konstruktors (3/4)

```
1  class Robot{  
2      String name;  
3      int numberOfEyes;  
4  
5      Robot(String userDefinedName, int aNumber){  
6          this.name = userDefinedName;  
7          this.numberOfEyes = aNumber;  
8      }  
9  }
```

- Kein Rückgabewert
- Gleicher Bezeichner wie Klasse
- **this** wird verwendet, um auf Attribute dieser Klasse zuzugreifen
- Kann keinen, einen oder **mehrere** Parameter haben
- Beispielaufruf: Robot **ronja** = new Robot("Ronja", 2);



- Es gibt Default-Konstrukturen
 - Existiert, wenn kein Konstruktor definiert wurde
 - Keine Parameter
 - Alle Attribute haben Default-Werte (int auf 0, String auf null)
 - Wird ungültig, wenn mindestens ein Konstruktor selbst definiert

- **Achtung!** Default-Konstrukturen stehen nicht im Code

- Impliziter Code:

```
Robot(){  
    }  
}
```



Deep Dive Java: Woche 2

openHPI-Java-Team

Hasso-Plattner-Institut



Geschachtelte for-Schleifen

```
1  for (int i = 0; i < 5; i++) {  
2      for (int j = 0; j < i; j++){  
3          System.out.println(i*j);  
4      }  
5  }
```

Ausgabe:

0	6
0	0
2	4
0	8
3	12



■ Primitive Datentypen:

- Gleicher Wert: == (Achtung bei Fließkommazahlen!)

```
1 int a = 2;  
2 int b = 3;  
3 int c = 2;  
4  
5 System.out.println("a == b: " + (a == b));  
6 System.out.println("a == c: " + (a == c));
```

Ausgabe:

a == b: false

a == c: true



■ Primitive Datentypen:

- Größer als: >
- Größer gleich: >=

```
1 int a = 2;  
2 int b = 3;  
3 int c = 2;  
4  
5 System.out.println("a > b: " + (a > b));  
6 System.out.println("b > a: " + (b > a));  
7 System.out.println("a >= c: " + (a >= c));
```

Ausgabe:

a > b: false

b > a: true

a >= c: true



■ Objektdatentypen:

- Objektwerte (zum Beispiel Strings) werden mit `equals` verglichen
- Vergleich mit `==`: Referenzvergleich (Objektidentität)

```
1 String hallo1 = "Hallo";
2 String welt1  = "Welt";
3 String hallo2 = "Hallo";
4
5 System.out.println(hallo1 == hallo2);
6 System.out.println(hallo1 == hallo1);
7 System.out.println(hallo1.equals(hallo2));
```

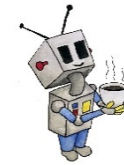
Ausgabe:

false

true

true

continue und break



```
1 int tellPosition(String[] words, String word) {
2     int output = -1;
3     for(int i = 0; i < words.length; i++){
4         if(words[i].equals("Test")) {
5             System.out.println("Skipping lines");
6             continue;
7         }
8         if(words[i].equals(word)) {
9             output = i;
10            break;
11        }
12        System.out.println("Blabla");
13    }
14    return output; // wenn Abbruch in words[]
15 }
```

- **continue:** Neustart des aktuellen Blocks (Schleifendurchlauf)
- **break:** Verlassen des aktuellen Blocks (Schleifendurchlauf)
- **return:** Verlassen der aktuellen Methode

