# Einführung in Java

Arne Hüffmeier

Michelle Liebers, Dennis Hoffmann

Tilman Lüttje, Jean Wiele

Angelehnt an Java-Vorkurs der Freitagsrunde

## Motivation für Methoden

- helfen, den Code zu strukturieren
- erlaubt Wiederverwendung von Code
  - Redundanz wird vermieden
  - Fehleranfälligkeit sinkt

## Methoden

- können direkt einen Effekt haben, aber kein Ergebnis liefern

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

  → Text auf dem Bildschirm

- können keinen direkten Effekt haben, aber ein Ergebnis liefern

```
int ersteSumme = calcSum(erstesArray);
```

  → Ergebnis von *calcSum* wird in *ersteSumme* gespeichert

## Aufbau einer Methode

```
public static <Rueckgabetyp> <Name> (<Parameterliste>) { <Anweisungen> }
```

## Methodenkopf

- public { private, protected}
  Beschränkt die Sichtbarkeit der Methode. Aktuell erstmal public verwenden.
- static
  Zeigt das diese Methode benutzt werden kann ohne das vorher mit new eine Instanz erzeugt werden muss (genaueres beim Thema Objekte)
- **Rückgabetyp**
  Was soll die Methode zurück gaben ? (z.b. int, String) Wenn keine Rückgabe gewünscht ist muss hier void stehen.

## Aufbau einer Methode

```
public static <Rueckgabetyp> <Name> (<Parameterliste>) { <Anweisungen> }
```

## Methodenkopf

- **Name**
  Name der Methode. Man sollte eine treffenden Namen wählen.
- **Parameterliste**
  beliebig viele Parameter. (z.b. int a).
  Wenn kein Parameter benötigt wird kann hier auch nichts stehen.

## Aufbau einer Methode

```
public static <Rueckgabetyp> <Name> (<Parameterliste>) { <Anweisungen> }
```

## Methodenrumpf

Es können hier beliebige Anweisungen stehen.
Wenn ein **Rückgabetyp** benötigt wird passiert dieses mittels
return <Variable>

```java
public static int berechneFlaeche (int a, int b) {
    int flaeche = a * b;
    return flaeche;
}
```

```java
public static int berechneFlaeche(int a, int b) {
    return a * b;
}
```

```java
public static void sayHello(String name) {
    System.out.println("Hallo" + name);
        //bei void ist kein return notwendig
}
```

```java
public static void sayHello(){
    System.out.println("Hallo" +name );
}

public static void main(String[] args) {
    String name = "Hannes";
    sayHello();
}
```

- Welche Variablen kennt meine Methode?
  - "Globale Variablen": in der Datei definierte Variablen
  - die übergebenen Parameter
- Was kann man machen, wenn man zusätzliche Variablen braucht?
  $\rightarrow$ dann definiert man "lokale Variablen" in der Methode

**Merke!**

- Variablen sind genau in dem Bereich gültig, in dem sie deklariert worden sind.
- "Globale Variablen" werden innerhalb der äußersten geschweiften Klammern dekalariert
  $\rightarrow$ daher sind sie in der ganzen Datei gültig

# Verdecken

lokale Variablen können (globale) Variablen *verdecken*

- lokale Variablen sind Parameter oder innerhalb der Methode deklarierte Variablen
- lokale Variablen können genauso heißen wie globale Variablen
- im Falle des Verdeckens
  - hat man keinen Zugriff auf den Wert der globalen Variable
  - kann keine (versehentliche) Änderung des Werts der globalen Variable vorgenommen werden
    $\rightarrow$ ohne Verdecken kann man auf globale Variablen zugreifen

```
public class Geometrie {



    public static void main(String[] args) {
        int a = 3;
        int b = 2;
        int umfang = 2 * a + 2 * b;
        int c = berechneFlaeche(a,b);
        System.out.println("Rechteck " + a + ", " + b
        + ": Flaeche = " + c + ", Umfang = " + umfang);
    }
}
```

- ```
    public static int berechneFlaeche(int a, int b) {
        return a * b;
    }
  ```

- ```
    public static int berechneFlaeche(int a, int b) {
        int umfang = a + 1;
        return a * b;
    }
  ```

- ```
  public static int berechneFlaeche(int b) {
        a = a + 1;
        return a * b;
    }
  ```
  ```

```
public class Geometrie {
    static int extra;

    public static int berechneFlaeche(int a, int b){
        extra = 20;
        return a * b;
    }

    public static void main(String[] args) {
        extra = 10;
        int a = 3;
        int b = 2;
        int umfang = 2 * a + 2 * b;
        int c = berechneFlaeche(a,b);
        System.out.println("Rechteck " + a + ", " + b
        + ": Flaeche = " + c + ", Umfang = " + umfang);
        System.out.println("extra = "+extra);
    }
}
```

## Overloading - Überladen von Methoden

Es kann mehrere (unterschiedlicher) Methoden mit dem selben Namen geben → *Overlading*

## Einschränkung:

Methoden mit dem selben Namen müssen trotzdem eindeutig identifizierbar sein.

- mittels unterschiedlicher Parameteranzahl
- mittels unterschiedlichen Parametertypen

Unterschiedliche Rückgabetypen reichen für eine Eindeutigkeit nicht aus!

# Overloading - Beispiel

Die Klasse Geometrie soll für Kreise erweitert werden.

## *berechneFlaeche* für Kreise?

```
public static double berechneFlaeche(int radius, int pi) {
    return pi * radius * radius;
}
```

**Falsch!**

- Der Standardfall ist bereits eine Methode *berechneFlaeche* mit zwei Parametern vom Typ int.
- Die Variablennamen der Parameterliste können nicht zur Unterscheidung von Methoden verwendet werden.

  → daher keine eindeutige Identifizierung möglich

## besser!

```
public static double berechneFlaeche (int radius) {
    return 3.141 * radius * radius;
}
```

# Abschließende Bemerkungen

## Fragen für die Vorgehensweise

- Welche Parameter werden benötigt?
- Welches Ergebnis soll geliefert werden?
- Wie komme ich mit den Parametern auf das Ergebnis?

## häufige Fehlerquellen

- Vergessen eines Statements im Methodenkopf
- falsche Typen in Parameterliste
- falscher Rückgabetyp
- *return*-Statement vergessen
- Kann man Methoden innerhalb von anderen Methoden deklarieren?
  $\rightarrow$ **Nein**, aber man kann andere Methoden aufrufen.

# Abschließende Bemerkungen

Ab jetzt bis zum ende eures Studium gilt:

**In die main Methode werden nur andere Methoden aufrufen.**
**Also kein Berechnungen und auch kein for oder while.**

# Methoden aus der eigenen Datei

## Aufruf von Methoden innerhalb der Datei

Für die Belegung der Parameterliste kommt es zur Übergabe von:

- vorhandenen Variablen
- festen Werten

## ohne Rückgabewert

```
public static void main(String[] args) {
int a = 3;
meineMethode(a);
}
```

## mit Rückgabewert

```
public static void main(String[] args) {
int b = meineMethode(3);
}
```

# Methoden aus anderen Dateien

## ohne Rückgabewert

```
public static void main(String[] args) {
int meineVar = 3;
MeineKlasse.meineMethode(meineVar);
}
```

## mit Rückgabewert

```
public static void main(String[] args) {
int meineVarA = 3;
int meineVarB = MeineKlasse.meineMethode(meineVarA);
}
```

## Besonderheit: rekursive Methoden

Rekursion nennt man die Definition einer Funktion durch sich selbst

- z.B. bei der rekursiven Definition von Folgen
- Rekursive Methoden rufen sich immer wieder selbst auf, bis die Abbruchbedingung erreicht ist.

$$a_n = 2 * a_{n-1}$$

$$a_0 = 1$$

# Rekursion - Beispiele

kleiner Gauß: 1 + 2 + 3 + . . . + n-1 + n

- Bildungsvorschrift: $a_n = a_n + a_{n-1}$
- Abbruchbedingung: $a_1 = 1$

```
public class Gauss{
  public static void main(String[] args) {
    System.out.println(gauss(10));
  }

  public static int gauss (int n) {
    if (n <= 1) {
      return 1;
    }
    return n + gauss(n-1);
  }
}
```

# Rekursion und Iteration

Alles, was wir mit Rekursion lösen können, können wir auch mit Schleifen lösen.
Rekursion ist meistens allerdings übersichtlicher.

## rekursive Lösung

```
public static int gauss (int n) {
  if (n <= 1) {
    return 1;
  }
  return n + gauss(n-1);
}
```

## iterative Lösung

```
public static int gauss (int n) {
  int x = 0;
  for (int i = 0; i <= n; i++) {
    x += i;
  }
  return x;
}
```

# Warum dann Rekursion nutzen?

Wieso sollten wir Rekursion nutzen, wenn wir auch Schleifen nutzen könnten?

Weil Rekursion im Normalfall einfacher zu verwenden und zu verstehen ist. Sobald es sich verzweigt, wird es mit einer Schleife nicht mehr schön zu lösen sein, ohne Hilfsstrukturen.

# Der Unterschied liegt im Detail

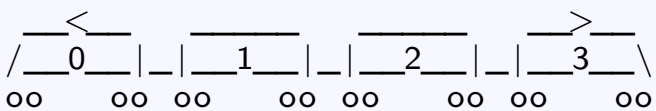
Die Einfachheit der Rekursion hat aber auch ihren Preis. Es geht zulasten der Performance, wie die Tabelle belegt.

| | Rekursion | Iteration | Δ |
|---|---|---|---|
| fib(10) | 9.294 ns | 1.555 ns | 6 |
| fib(45) | 3.834.252.031 ns | 3.587 ns | 1.007.600 |

Java kann Schleifen etwas schneller abarbeiten.

# Einführung in die Objekte

Wir erinnern uns an unseren Zug?



```
int[] zug = new int[4];
```

Es ist schon doof, dass der Zug nur einen Wert speichern kann, also bei uns seine Fahrgäste in den Waggons. Wäre es nicht schön, wenn er auch noch wissen würde, ob der Waggon erste oder zweite Klasse ist?

Wir lernen nun, wie wir uns eigene Datentypen erstellen!

# Der Traum vom eigenen Datentyp

Wie müsste der ideale Waggon sein?
Er müsste haben:

- eine Passagierzahl
- eine Klasse (erste oder zweite)

Unser Zug sollte aus solchen Waggons bestehen und nicht aus Integer.

# Datentyp: Waggon

So sähe der Waggon in Java dann aus:

```
class Waggon {
    int passagiere;
    int klasse;

    Waggon() {
        passagiere = 0;
        klasse = 0;
    }
}
```

## Objekt erklärt

Waggon() { } ist der Konstruktor von Waggon. Jedes Objekt braucht einen.
Ein Konstruktor ist mit einer Methode zu vergleichen, welche ein Waggon erstellt.

Ein Objekt ist eine Instanz einer Klasse, welche eigene Variablen, Attribute genannt, und eigene Methoden besitzt, welche nur das Objekt, in unserem Fall der Waggon, anwenden kann.

Damit Java weiß wo es unseren Waggon findet ist es nötig die Datei mit einem import in unsere Main Datei einzubinden.

# Unser neuer Zug

Nun haben wir einen Waggon, damit kann unser Zug modernisiert werden

```
Waggon[] zug;
zug = new Waggon[4];
```

Wir müssen nun aber noch die Waggons in Unserem Zug erstellen.

```
for (int i = 0; i < zug.length; i++) {
    zug[i] = new Waggon();
}
```

# Das verstehe ich nicht

Also, das ist wie folgt

```
zug[i] = new Waggon();
```

## new Object() erklärt

zug[i] ist der **Waggon** in unserem Zug, der an Stelle **i** stehen soll.

Waggon() ruft den Konstruktor von **Waggon** auf. Es wird also ein Waggon produziert.

new Waggon() sagt Java soll einen neuen **Waggon** im Speicher erstellen, welchen wir an der Stelle **i** in unserem Zug haben wollen.
Wir sprechen hier davon, dass wir einen neues Objekt der Klasse Waggon instanzieren.

# Was bringt mir das?

Das ist doch nur noch komplizierter!

*Kann sein, aber wir könnten die Wagen nun mit ihrer Klasse beim Erstellen auszeichnen.*

Wie? *So:*

```
class Waggon {
int passagiere;
  int klasse;

  Waggon(int x) {
  klasse = x;
  passagiere = 0;
  }
}
```

## Der Konstruktor erklärt

Waggon(int x) ist ein Konstruktor, der eine Zahl entgegen nimmt. Das geht, da ein Konstruktor einer Methode sehr ähnlich ist, man kann ihn beliebig viele Argumente entgegennehmen lassen, und im Konstruktor verschiedene Anweisungen ausführen lassen.

# Unser neuer Zug

Wir haben nun einen fabrikneuen Zug der Länge 4.

```
Waggon[] zug = new Waggon[4];
for (int i = 0; i < zug.length; i++) {
    zug[i] = new Waggon(2); //zweite Klasse Wagen
}
```

Aber wie bekommen wir nun an die Fahrgäste in den Waggon?
So:

```
zug[0].passagiere = 10;
```

Nun haben wir im ersten Waggon 10 Passagiere.

# Das verstehe ich nicht

Also, das ist ganz einfach:

```
zug[x].passagiere = 10;
```

## Punktnotation erklärt

zug[x] ist ein Waggon, der an Position x unseres Zuges.

zug[x]. sagt, wir wollen auf etwas zugreifen, welches dem **Waggon** gehört.

zug[x].passagiere sagt, wir wollen auf das Attribut **passagiere** unseres Waggons zugreifen.
In dem Beispiel weisen wir ihm den Wert 10 zu.

# Wir bauen einen Zug

```
Waggon[] zug = new Waggon[4];

for (int i = 0; i < zug.length; i++) {
    zug[i] = new Waggon(2);
}

zug[0].klasse = 1;
zug[0].passagiere = 5; //kann sich kaum einer leisten

for (int i = 1; i < zug.length; i++) {
    zug[i].passagiere = 30;
}
```

Es ist also ganz einfach, einen 2. Klasse Wagen in einen 1. Klasse Wagen zu verwandeln

Der Bahn gefällt das überhaupt nicht, dass jeder die Klassierung ihrer Wagen ändern kann.
Das muss man ändern!

# Der neue Waggon

```
public class Waggon {
    private int passagiere;
    private int klasse;

    public Waggon(int x) {
        klasse = x;
        passagiere = 0;
    }
}
```

Nun kann keiner mehr die Klasse unseres Waggon ändern!

## Die neuen Wörter erklärt

**private** bedeutet, keiner außer der Waggon selber kann auf das Attribut zugreifen.

**public** jeder kann darauf zugreifen.
In unserem Fall kann jeder also einen Waggon erstellen.

Diese Schlüsselwörter lassen sich auch für Methoden und Konstruktoren verwenden.

# Sicherheit

Wieso kann ich nicht mehr die Klasse angucken?

Weil wir sie geschützt haben. Niemand außer dem Waggon selbst kann jetzt noch darauf zugreifen.

Wie komme ich wieder an die Informationen?

Über Methoden, die darauf zugreifen.

# getter und setter

Um auf private Attribute zugreifen zu können, gibt es in Java getter und setter Methoden.
Diese werden im Normalfall nach der Variable benannt, auf die sie zugreifen.

```
class Waggon {
    private int passagiere;

    public Waggon() {
        passagiere = 0;
    }

    public void setPassagiere(int x) {
        passagiere = x;
    }

    public int getPassagiere() {
        return passagiere;
    }
}
```

Eine getter-Methode liefert den Wert des zugehörigen Attributes
Eine setter-Methode hingegen setzt den Wert der Variable.

# Was bringt mir das?

Wir wollen nicht immer, dass man einfach eine Angabe ändern kann.
Zum Beispiel darf man einen Erster-Klasse Wagen als Zweiter-Klasse Wagen nutzen, aber aus einem Zweiter-Klasse Wagen darf man keinen Erster-Klasse Wagen machen.
In Java sähe es z.B. so aus

```
private int klasse;

public int getKlasse() {
    return klasse;
}

public void setKlasse(int x) {
    if(x >= klasse) {
        klasse = x;
    }
}
```

# Fehlt da nicht was?

Wieso haben die ganzen Methoden kein static mehr?

Weil das static ausdrückt, es handelt sich um eine statische Funktion, also etwas, das unabhänging vom Zustand des Objektes ist.
Ohne das Static ist es von der aktuellen Instanz der Klasse abhängig.

**Merke**

Statische Methoden geben bei gleicher Eingabe immer das Gleiche zurück.

Nicht statische Methoden sind vom Zustand des Objektes abhängig, können also bei gleicher Eingabe unterschiedliches zurück geben.

# Der neue Zug

Wir erstellen eine Klasse Zug. Was muss er haben?
Ersteinmal die Attribute und der Konstruktor

```
class Zug {
    private String name;
    private Waggon[] waggon;

    public Zug(String name, int laenge) {
        this.name = name;
        waggon = new Waggon[laenge];
        for (int i = 0; i < laenge; i++) {
         waggon[i] = new Waggon(2);
        }
    }
```

## Erklärung des Zuges

**this** bezeichnet die aktuelle Instanz der Klasse Zug, in der wir uns befinden.

**this.name** sagt dem Compiler, welchen Namen wir meinen. In diesem Fall das Attribut der Klasse Zug, da name als Parameter das Attribut der Klasse überlagert.

# Der neue Zug

## Die Methoden

```
/**
* Liefert Fahrgastzahl des angeforderten Waggons.
*/
public int getFahrgast(int waggon) {
return this.waggon[waggon].getPassagiere();
}

/**
* Es steigen x Passagiere in den Waggon ein.
* Also werden es nur mehr.
*/
public void einsteigen(int x, int waggon) {
x += this.waggon[waggon].getPassagiere();
this.waggon[waggon].setPassagiere(x);
}

/**
* Es steigen x Passagiere aus dem Waggon aus.
* Also werden es nur x weniger.
*/
public void aussteigen(int x, int waggon) {
x = this.waggon[waggon].getPassagiere() - x;
this.waggon[waggon].setPassagiere(x);
}
}
```

# Neue Möglichkeiten

Nun können wir auch Züge anders darstellen.
Aber warum muss ich immer einen new Waggon() ins Array setzen?
Dazu erstellen wir nun erstmal eine Methode getAlleFahrgaeste()

```
/**
 * Gibt die Fahrgastzahl aller Waggons zusammen aus.
 */
public int getAlleFahrgaeste() {
    int x = 0;
    for (int i = 0; i < waggon.length; i++
        x += waggon[i].getPassagiere();
    }
    return x;
}
```

Als nächstes testen wir folgendes bei unserem jetzigen Zug

```
Zug ice = new Zug("ICE",10);
ice.einsteigen(30,1); //30 Passagiere in Wagen 1
System.out.println(ice.getAlleFahrgaeste());
```

Wir haben also nur 30 Fahrgäste im ganzen Zug.

# Neue Möglichkeiten

Nun schreiben wir den Konstruktor des Zuges um

```
public Zug(String name, int laenge) {
    this.name = name;
    this.waggon = new Waggon[laenge];
    Waggon tmp = new Waggon(2);
    for (int i = 0; i < laenge; i++) {
        waggon[i] = tmp;
    }
}
```

Und testen jetzt die Anweisung von eben.

```
Zug ice = new Zug("ICE",10);
ice.einsteigen(30,1); //30 Passagiere in Wagen 1
System.out.println(ice.getAlleFahrgaeste());
```

Was stellen wir fest?

Unser Zug hat nun 300 Fahrgäste im Zug, obwohl nur 30 eingestiegen sind.

**Wieso?**

# Die Referenz

Wir hatten in jeden Feld unseres Arrays den gleichen Waggon angegeben. Java hat also jedes Array auf den Waggon verweisen lassen.
Das ist, als ob wir 10 Fernbedienungen, in dem Fall die Arrayeinträge, auf nur einen Fernseher eingestellt hätten.
Wenn nun eine Fernbedienung den Sender wechselt, ist für alle anderen auch der Sender gewechselt.

Da das aber nicht gut für unseren Zug ist, wird jedes Feld des Arrays mit einem neuen Waggon initialisiert.

```
Waggon a = new Waggon(1);
Waggon b = a;

System.out.println(a.getKlasse());
System.out.println(b.getKlasse());

b.setKasse(2);

System.out.println(a.getKlasse());
System.out.println(b.getKlasse());
```

```
Waggon a = new Waggon(1);
Waggon b = a;

System.out.println(a);
System.out.println(b);
```

```
Waggon a = new Waggon(1);
Waggon b = new Waggon(1);

System.out.println(a);
System.out.println(b);
```