



LUDWIG-  
MAXIMILIANS-  
UNIVERSITÄT  
MÜNCHEN



 mobile and  
distributed systems group



# Javakurs für Anfänger

Einheit 02: Klassen & Objekte

Kyrill Schmid

Lehrstuhl für Mobile und Verteilte Systeme



## 1. Teil: Klassen

- Grundstruktur einer Java-Klasse
  - Eigenschaften (Attribute)
  - Variablen und Datentypen
  - Verhalten (Methoden)

## 2. Teil: Objekte

- Objekte erzeugen
- Konstruktoren
- Getter/Setter Methoden

### Praxis:

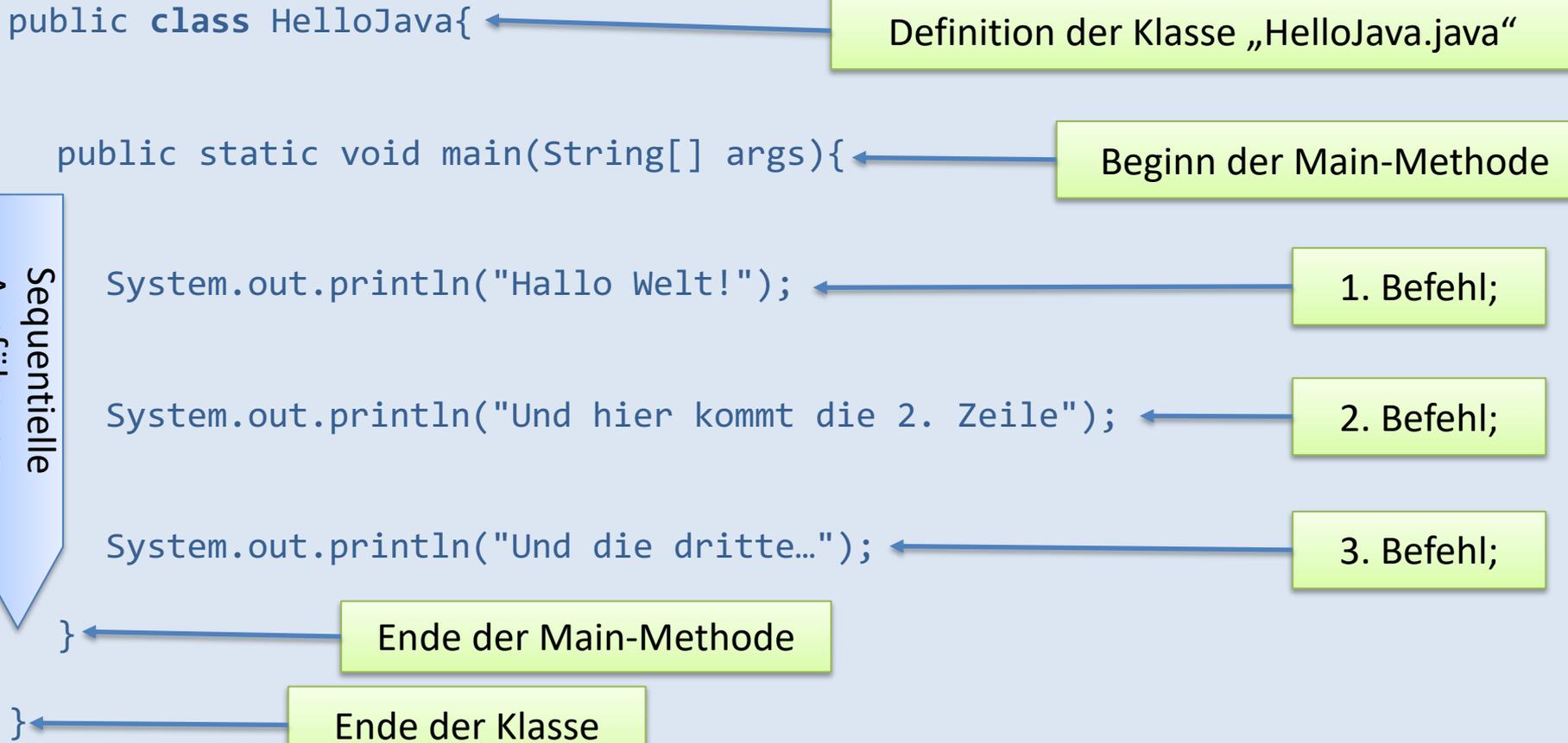
- Die Klasse Auto schreiben
- Auto-Objekte erzeugen
- Methoden aufrufen
- Programmieraufgabe

### Lernziele

- Objektorientierung kennenlernen
- Zusammenhang zwischen Klassen und Objekten verstehen
- Erste Programmierung mit Objekten

**Bisher:** Nur eine einfache Klasse mit `main`-Methode

- `Main`-Methode bildet den Einstiegspunkt für unser Programm (ausführbarer Code)



**Bisher:** Nur eine einfache Klasse mit `main`-Methode

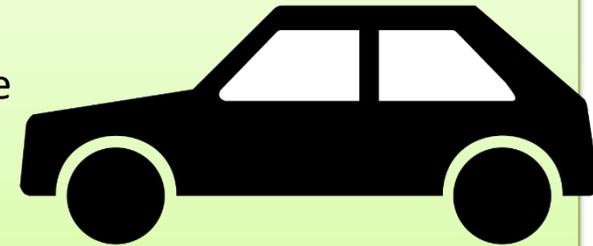
```
public class HelloJava{  
    public static void main(String[] args){  
        System.out.println("Hallo Welt!");  
    }  
}
```

Nun: Schnell raus aus der `main`-Methode

- Komplexere Programme erfordern Strukturmechanismen
- Daher: **Objektorientierung** als vereinfachte Sichtweise auf komplexe Systeme
  - Klassen bieten eine **vereinfachte Sicht** der realen Welt
  - **Kapselung** von *Eigenschaften* und *Verhalten* gleichartiger Objekte in Klassen
  - **Wartbarkeit** durch Begrenzung von Änderungen auf einzelne Klassen
  - **Wiederverwendung** von bereits vorhandenen Klassen
- Java Programme bestehen i.d.R. aus **Objekten**, die über **Methodenaufrufe** miteinander **kommunizieren**

## Klasse:

- Stellt ein Konzept bzw. Plan gleichartiger Objekte dar (Bsp.: Auto)
  - Welche **Eigenschaften** (Attribute) haben die Objekte der Klasse
    - (Bsp.: Name, Preis,...)
  - Welches **Verhalten** (Methoden) bieten die Objekte der Klasse
    - (Bsp.: fahren, bremsen,...)
- Beschreibt dadurch einen Teil der Realität
- Wird vom Programmierer geschrieben, gespeichert und kompiliert



## Objekt (= Instanz einer Klasse):

- Wird beim Ausführen des Programms erzeugt und spätestens beim Beenden verworfen
- Existiert nur im Speicher
- Wird nach dem vorgesehenen Konzept bzw. dem Plan seiner Klasse erstellt:
  - Bekommt Werte für seine Attribute
    - (Bsp.: „Audi Quattro“, 30.000 Euro,...)
  - Verhält sich wie in seiner Klasse definiert
    - (Bsp.: fahren, bremsen, ...)



```

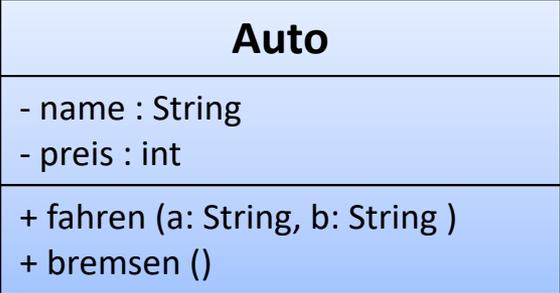
1 // Package Deklaration
2 package java.fuer.anfaenger;
3
4 // Import externe Klassen
5 import java.util.*;
6
7 // Klassendefinition
8 public class Auto{
9
10     // Instanzvariablen
11     private String name;
12     private int preis;
13
14     // Konstruktor
15     public Auto(String name, int preis){
16         this.name = name;
17         this.preis = preis;
18     }
19
20     // Methoden
21     public void fahren(String a, String b){
22         System.out.println("Das Auto faehrt von "+a+" nach "+b);
23     }
24
25     public void bremsen(){
26         System.out.println("Ich bremse!");
27     }
28
29 }

```

**Eigenschaften (Attribute)**

**Konstruktor (kommt später)**

**Fähigkeiten (Methoden)**



Die Eigenschaften der Objekte einer Klasse (Instanzen) werden durch **Instanzvariablen** (auch Feldvariablen genannt) definiert:

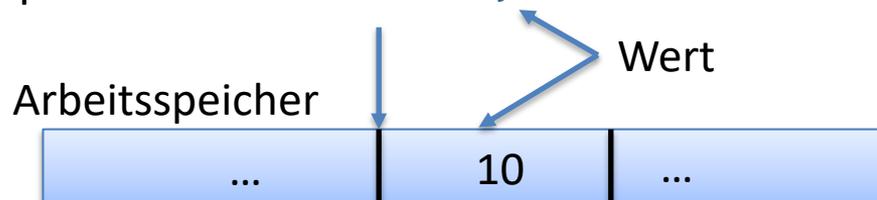
- Beschreiben die Eigenschaften (Attribute) eines Objekts
- Speichern Daten und halten so den aktuellen **Objektzustand**
- Werden zusammen mit dem Objekt erzeugt und initialisiert
  - Mit Defaultwerte oder über eigene Konstruktoren
- Ein direkter Zugriff von außerhalb der Klasse sollte vermieden werden!
  - Prinzip der Datenkapselung (Objekt behält Kontrolle über seinen Zustand)
  - Daher: `private` Deklaration
  - Grund: Erweiterbarkeit bzw. Wartbarkeit
- Beispiele:
  - `private String name;`
  - `private int preis;`

Neben Instanzvariablen gibt es noch andere Arten von Variablen:

- Lokale Variablen
- Parameter (für Methodenaufrufe)

Variablen dienen als ansprechbarer Platzhalter zum Speichern von Daten

- Haben immer einen **Datentyp** (int, String, Objekt,...) und einen **Namen**
  - Werden dadurch *deklariert*: `Datentyp variablenName;`
  - Aussagekräftigen Namen mit kleinem Anfangsbuchstaben wählen
- Reserviert festen Speicherplatz auf dem Hauptspeicher (je nach Datentyp)
  - Kann mit Namen angesprochen werden
  - Werte können dort abgelegt und verändert werden
  - Wertzuweisung (*initialisieren*): `variablenName = wert;`
- Deklarieren und initialisieren in einem Schritt möglich
- Beispiel: `int meineZahl = 10;`



Der Datentyp bestimmt auch die Größe des reservierten Speicherbereichs

- Unterscheidung zwischen *Primitiven*- und Referenztypen

## Primitive Typen

- **8** primitive Datentypen in Java vordefiniert
- Erfordern wenig Speicherplatz und geringen Aufwand für Compiler und Interpreter => Geschwindigkeitsvorteile

Datentyp	Verwendung	Größe in Bit	Wertebereich
<b>boolean</b>	Wahrheitswert	8	false, true
<b>char</b>	Zeichen	16	0 – 65.535
<b>byte</b>	Ganzzahl	8	-128 bis 127
<b>short</b>	Ganzzahl	16	- 32.768 bis 32.767
<b>int</b>	Ganzzahl	32	-2.147.483.648 bis 2.147.483.647
<b>long</b>	Ganzzahl	64	-2 <sup>63</sup> bis 2 <sup>63</sup> -1
<b>float</b>	Kommazahl	32	Genauigkeit ca. 7 Kommastellen
<b>double</b>	Kommazahl	64	Genauigkeit ca. 15 Kommastellen

## Referenztypen

- Jede Klasse (mit mehreren Attributen)
  - Können sich aus verschiedenen Elementen zusammensetzen
  - Bsp.: String, Array, Auto, ...

Jede Variable hat eine **Lebensspanne** und eine **Sichtbarkeit** während der Abarbeitung des Programms

- **Lebensspanne** (Von der Erzeugung bis zum Ende)
  - Ist im Speicher repräsentiert
  - Werte können gelesen und geschrieben werden
- **Sichtbarkeit**
  - Variable kann nur in ihrem Sichtbarkeitsbereich verwendet werden  
=> Ein Zugriff erfolgt damit nur während der Lebensspanne
- Sichtbarkeit und Lebensspanne müssen nicht identisch sein!
  - Stichwort: (Temporäre) Überschattung
- Sichtbarkeit  $\leq$  Lebensspanne

## Methoden

- beschreiben die Fähigkeiten bzw. das Verhalten der Klasse/Objekt
- Operationen mit Attributen der Klasse/Objekt

## Aufbau (Allgemein):

```
<Modifier> Rückgabetyp methodName(ParameterTyp parameter1,...){  
    //Methodenrumpf: Das was die Methode machen soll  
}
```

- **Modifier** (optional): verändert Eigenschaft der Methode
  - Für den Anfang: `public`
- **Rückgabetyp**: gibt den Datentyp des Ergebnisses an (`int`, `String`, `Auto`,...)
  - Im Methodenrumpf muss der entsprechende `return` Befehl kommen!
  - `void` für keine Wertrückgabe (kein `return` Befehl)
- **Methodennamen**: frei wählbar (aussagekräftig, kleiner Anfangsbuchstabe)
  - Liste an Parametertyp + Parameter: Datentyp + variablenname

//Beispiele:

```
public void fahreVonAnachB(String a, String b){  
    System.out.println("Ich fahre von "+a+" nach "+b);  
}
```

Mit Parameter

```
public void fahren(){  
    System.out.println("Ich fahre...");  
}
```

Ohne Rückgabewert

```
public int wieWeitGefahren(){  
    return distanz; //muss als Instanzvariable (int) vorliegen  
}
```

Ohne Parameter

Mit Rückgabewert

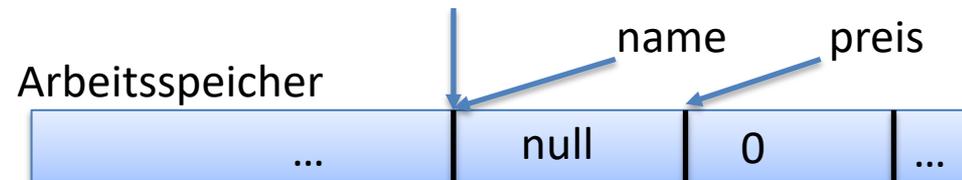
Also: return Befehl

Ein Objekt ist eine eindeutige **Instanz** seiner Klasse

- Verfügt über dessen Attribute und Methoden
- Wird mit dem **new** Operator erzeugt, Bsp.:  

```
Klassenname meinObjekt = new Klassenname();
```
- Besitzt:
  - *Zustand*: Aktuelle Werte der Instanzvariablen
  - *Verhalten*: Methoden
  - *Identität*: Eindeutige Speicheradresse

```
Auto mein_audi = new Auto();
```



Beispiel: Klasse Auto

- Besitzt Instanzvariablen `String name` und `int preis`
- Ein neues Auto „mein\_audi“ anlegen mit: `Auto mein_audi = new Auto();`
- Problem: Das neue Objekt „mein\_audi“ hat bisher nur Defaultwerte für den Namen und den Preis, nämlich `null` und `0`

Wir benötigen einen **Konstruktor** zum Erzeugen von Objekten

- Aufgabe: Bringe das Objekt in einen definierbaren **Anfangszustand** beim Erzeugen
  - Anfangsbelegung der Instanzvariablen
  - Kann frei vom Entwickler programmiert werden
  - Ist eine spezielle Methode der Form:

```
public Klassenname (Parametertyp Parameter1,...){  
    //Anweisungen  
}
```

- Es können mehrere Konstruktoren definiert werden
  - Der „richtige“ Konstruktor wird nach Anzahl und Typen der Parameter aufgerufen
    - Dazu später mehr...
- Ist kein Konstruktor definiert wird vom Compiler automatisch ein Default-Konstruktor erzeugt
  - Besitzt keine weitere Anweisungen
- Der entspr. Konstruktor wird bei der Objekterzeugung mit **new** aufgerufen

Wollen wir ein Auto mit einem bestimmten Namen und einem bestimmten Preis erzeugen, schreiben wir unseren eigenen Konstruktor, der die Instanzvariablen belegt.

```
public class Auto{  
  
    //Instanzvariablen  
    private String name;  
    private int preis;  
  
    //Konstruktor  
    public Auto(String name, int preis){  
  
        this.name = name;  
        this.preis = preis;  
    }  
}
```

Klasse: Auto

Instanzvariablen: Beschreiben Attribute der Klasse

Konstruktor zur Belegung der Attribute beim Erzeugen

Ein Auto kann (bzw. muss) nun durch folgenden Aufruf erzeugt werden:

- `Auto mein_auto = new Auto(„Fiat Punto“,9000);`

Das Schlüsselwort `this` wird als Selbstreferenz bezeichnet

- Verweist auf das Objekt auf dem die Methode aufgerufen wurde
- Der Typ ist immer die Klasse in der die Methode definiert ist
- Um auf Instanzvariablen zuzugreifen: `this.instanzvariable`
  - Beugt versehentliche Verwechslungen mit lokalen Variablen oder Parametern vor
  - Bsp.: `this.name = name;`

Für den Moment: Die Verwendung von `this`:

- `this`: bezieht sich auf das aktuelle Objekt
- `this.variable`: greift auf **Instanzvariable des eigenen Objekts** zurück
- `this.variable = variable;` Setzt den Wert der **Instanzvariable** mit dem Wert der **lokalen Variable** bzw. des Parameters

Zur Erinnerung:

- Ein direkter Zugriff auf Instanzvariablen von außerhalb der Klasse sollte vermieden werden (`private` Deklaration zur Kapselung von Daten)

Deshalb 2 Möglichkeiten zum setzen (schreiben):

- Bei Objekterzeugung über den Konstruktor (wie eben gesehen)
- Oder im weiteren Programmverlauf über spezielle „**setter**“-Methoden

Möglichkeit zum Lesen der Instanzvariablen:

- Über spezielle „**getter**“-Methoden

Innerhalb einer Klasse kann immer auf die privaten Instanzvariablen direkt zugegriffen (lesen u. schreiben) werden mit `objekt.instanzvariable`, bzw. `this.instanzvariable`

Allgemeiner Aufbau von Setter:

```
public void setIv_Name(IV_Typ iv_Name){  
    this.iv_name = iv_name;  
}
```

Allgemeiner Aufbau von Getter:

```
public IV_Typ getIv_Name (){  
    return iv_name;  
}
```

Diese eindeutigen (einfachen) Methoden können von Eclipse automatisch auf Basis der Instanzvariablen generiert werden

- *Source -> generate Getters and Setters...*

```
// Instanzvariablen

private String name;
private int preis;

// Setter Methoden für name und preis:

public void setName(String name) {
    this.name = name;
}
public void setPreis(int preis) {
    this.preis = preis;
}

// Getter Methoden für name und preis:

public String getName() {
    return name;
}
public int getPreis() {
    return preis;
}
```

Lösen Sie bitte selbstständig die folgende Programmieraufgabe zu *Klassen & Objekten* in Java:

- Erstellen Sie eine Hunde-Farm mit 3 unterschiedlichen Hunden
- Jeder Hund hat einen Namen (String), eine Rasse (String) und eine Größe in cm (int)
- Jeder Hund kann laufen und bellen
  
- Ablauf:
  - Jeder der Hunde soll einmal bellen und laufen
  - Da Sie der Besitzer sind, wollen Sie sich auch immer anzeigen lassen welcher Hund gerade bellt.
  - Beim Laufen interessiert sie nur, ob ein Hund läuft.
  
- Achten Sie auf eine objektorientierte Umsetzung entsprechend dem Kapselungsprinzip
  - `private` Instanzvariablen und `public` Methoden