

JAVA I

Grundlagen der Programmierung

Teil 2 von 5:

Objektorientierte Programmierung

Letzte Änderung: 13.03.2017

Teil 2 - Objektorientierte Programmierung

1. Klassen und Objekte

2. Methoden und Member

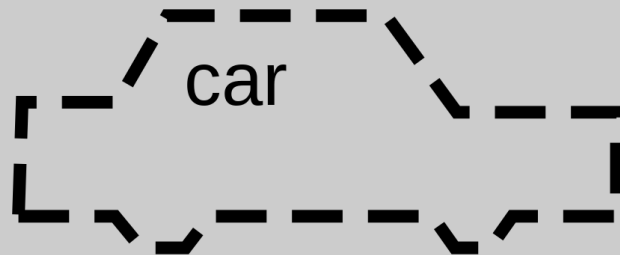
3. Klassenvariablen und Klassenmethoden

4. Einfaches Fehlerhandling

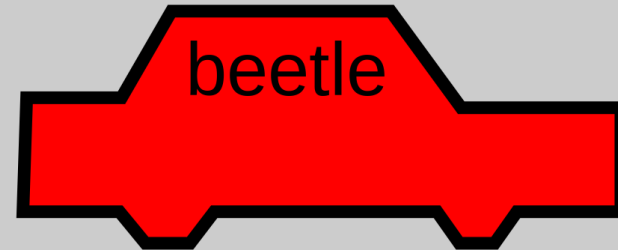
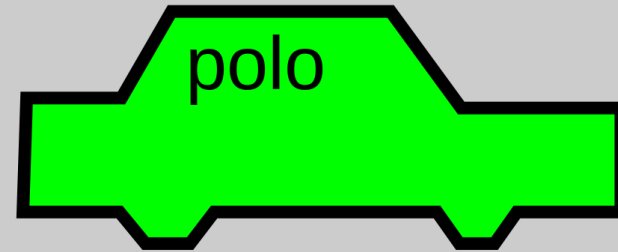
5. Exkurs „Planung und Entwurf“

1. Klassen und Objekte

class

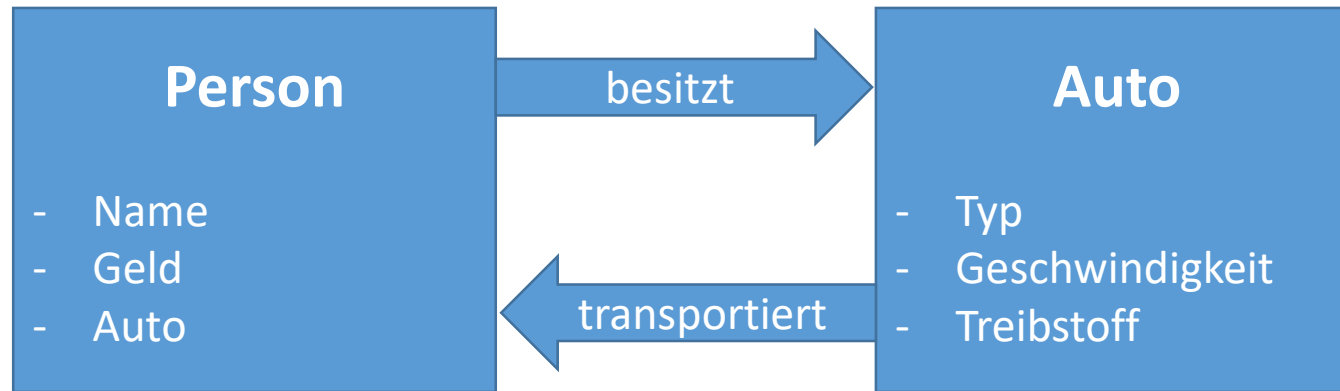


objects



1. Klassen und Objekte

Habe ich etwas (ist es fester Bestandteil einer Klasse)
oder **Benutze** ich etwas (ist es schon eine andere Klasse)



1. Klassen und Objekte

1.1. Klassen als Bauplan der Objekte

- Eine Datei sollte nur eine Klasse beinhalten
- Eine Klasse hat denselben Namen wie die Datei

*Beispiel Datei „**Car.java**“:*

```
public class Car {  
}
```

- Eine Klasse ist eine Beschreibung, eine Definition von Dingen mit
 - gemeinsamer Struktur
 - gemeinsames Verhalten
 - verschiedener Ausprägung

1. Klassen und Objekte

1.2. Objekte und ihre Identität

- Mit dem Schlüsselwort **new** wird von einer Klasse ein **Objekt** erzeugt
- Hinter dem Klassennamen gehört eine **Klammer** (*Erklärung folgt mit den Konstruktoren*)
- Grundsätzlich geschieht dies analog zu der Erzeugung einer primitiven Variablen

```
Car myCar;  
myCar = new Car();
```

```
Car yourCar = new Car();
```

- Jedes Objekt hat eine eigene **Identität**, technisch gesehen einen eigenen Platz im Speicher.
- Selbst wenn alle Werte gleich sind, sind es die Objekte daher NICHT.
- Daher kann man **KEINE Vergleiche mit ==** vornehmen!

1. Klassen und Objekte

1.3. Referenzvariablen (call by reference)

- Die Variable eines Objekts enthält nicht das Objekt selbst, sondern eine **Referenz auf seinen Speicherplatz**.
- Dies nennt man **call by referenz** – im Gegensatz zu **call by value** *bei primitiven Datentypen*

```
Car myCar;  
myCar = new Car();  
Car myCarToo = myCar;
```

```
int x = 10;  
int y = x;    // Wert wurde kopiert!  
y = 20;       // x ist immer noch 10!
```

- Beide **Objektvariablen** (*myCar* und *myCarToo*) referenzieren auf das selbe Objekt, **denselben Speicherplatz**
- In diesem Fall ist der Vergleich mit **==** zulässig, da es **dasselbe** ist.

1. Klassen und Objekte

1.4. Default-/Standardwerte ohne Initialisierung

- Wird eine Variable nur definiert, aber nicht initialisiert, hat sie einen Default-Wert

```
boolean b;    // DefaultWert:    false
int i;        // DefaultWert:    0
float f;      // DefaultWert:    0.0
char c;       // DefaultWert:    '\0000'    (das „Quadrat“...)

Car myCar;    // DefaultWert:    null        (quasi „NICHTS“)
```

- Ohne Initialisierung wird kein Speicher für Objekte belegt, daher ist es **null**
- Ein nicht initialisiertes Objekt zu benutzen führt zur beliebten **NullPointerException**

2. Methoden und Member

2.1. Methoden, Parameter und Rückgabewert

- Eine Methode kann beliebig viele **Parameter** (Variablen oder Objekte) bekommen
- Eine Methode hat **genau einen Rückgabewert**, der auch **void** = **NICHTS** sein kann
- Parameter werden IMMER **Call-By-Value** übergeben (siehe Kommentare im Beispiel)

```
[Zugriffslevel] [Rückgabewert] [Methodenname] ([Parameterliste]) {  
    [Methodenrumpf]  
}  
  
public boolean doSomething(int i, Car car) {  
    i = 25;           // Aenderungen betreffen NICHT die Variablen außerhalb der Methode  
    car.setSpeed(30); // Aenderung betrifft das Objekt auch außerhalb der Methode!  
    car = new Car();  // neuer Speicherplatz! / keine Auswirkung außerhalb der Methode  
    return true;      // gib zurueck: ja, es wurde irgendwas gemacht  
}
```

2. Methoden und Member

2.2. Methoden verwenden

- Methoden einer Klasse werden vom Objekt durch die **Punktnotation** aufgerufen.
- Wie bei anderen Ausdrücken, kann der Rückgabewert in einer Variable gespeichert werden

```
public class Car {  
  
    public static void main(String[] args) {  
        car myCar = new Car();  
        boolean result = myCar.doSomething(10, "something");  
    }  
  
    public boolean doSomething(int i, String s) { [...] }  
}
```

2. Methoden und Member

2.3. Member und Zugriff beschränken

- Klassen haben Variablen, die **Member**, **Attribute** oder auch **Eigenschaften** der Klasse genannt werden
- Jedes Objekt hat zur Lebenszeit seine **eigenen Werte** für seine Member
- Member können primitive Typen oder auch Objekte sein
- Man kann **public Member** deklarieren, auf die auch per Punktnotation zugegriffen werden kann -> ABER:
- **private** **Member** sollten **nach außen nicht sichtbar** sein – durch den **Modifizierer „private“**
- **public** Im Gegensatz dazu waren **Klassen** und **Methoden** bisher immer **öffentlich verfügbar**

```
public class Car {  
    private int speedMax;  
    private String type;  
    public String everybodyCanChangeMe;  
}
```

```
Car myCar = new Car();  
myCar.everybodyCanChangeMe = "Bad Dev was here :-)"
```

2. Methoden und Member

2.4. Zugriff auf Member (GETTER)

- Um Member eines Objekts abzufragen benutzt man **Zugriffsmethoden**

```
public class Car {  
  
    private int speedMax;  
    private String type;  
  
    public int getSpeedMax() {                // call by value!  
        return speedMax;                    // es wird eine Kopie zurückgegeben  
    }  
  
    public String getType() {                // call by reference!  
        return type;                        // der Versuch, "type" außerhalb zu  
                                            // verändern wirft einen Fehler!  
    }  
}
```

2. Methoden und Member

2.5. Konstruktor und Zugriff mit *this*

- Der Konstruktor wird bei Erstellung des Objekts aufgerufen und kann **Initialisierungs-Parameter** fordern
- Mit dem Schlüsselwort `this` kann auf das aktuell aufrufende Objekt verwiesen werden

```
public class Car {  
  
    private int speedMax;  
    private String myType;  
  
    public Car(int speed, String type) {  
        this.speedMax = speed;  
        this.myType = type;  
    }  
}
```

```
Car myCar = new Car(200, "Opel Zafira");
```

```
// Beispiel-Initialisierung
```

2. Methoden und Member

2.6. Modifizierungsmethoden mit und ohne Prüfung (SETTER)

- Bei reinen Datenobjekten ist ein „naiver“ SETTER gebräuchlich
- Oft ist eine Prüfung des übergebenen Parameter-Werts sinnvoll, bevor ein Member wirklich geändert wird

```
public class Car {  
    private int fuel;  
  
    public void setFuel(int fuel) { this.fuel = fuel; } // Tank koennte überlaufen!  
  
    public boolean addFuel(int additionalFuel) {  
        if (additionalFuel > 0 && this.fuel + additionalFuel < 50) {  
            this.fuel += additionalFuel;  
            return true;  
        }  
        return false;  
    }  
}
```

3. Klassenvariablen und Klassenmethoden

3.1. Klassenvariablen

- Klassenvariablen werden **von allen Objekten einer Klasse gemeinsam** verwendet

```
public class Car {  
    private static int built;  
  
    public Car() {  
        built++;    // Konstruktor zählt um 1 hoch  
    }  
}
```

- Bei jedem neuen Objekt wird die Klassenvariable **carsBuilt** hochgezählt

3. Klassenvariablen und Klassenmethoden

3.2. Klassenmethoden

- Klassenmethoden können aufgerufen werden, ohne dass ein Objekt der Klasse existiert
- Sie werden durch Punktnotation direkt von der Klasse erfragt

```
public class Car {  
    private static int built;  
  
    public Car() {    built++;    }           // Konstruktor zählt um 1 hoch  
  
    public static int getBuilt() {  
        return built;  
    }  
}  
  
int carsBuilt = Car.getBuilt();           // Beispiel-Aufruf
```


3. Klassenvariablen und Klassenmethoden

3.3. Besonderheiten beim Zugriff innerhalb der Klasse

```
public class Car {  
  
    private static int built;  
    private int speedMax;  
  
    public Car() {  
        built++;           // Zugriff ok!  
    }                     // da jedes Objekt die Klasseneigenschaften kennt  
  
    public void setSpeedMax(int speedMax) {  
        this.speedMax = speedMax;  
    }  
  
    public static void changeSpeedMax() {  
        speedMax += 10;    // Beides führt zu einem FEHLER,  
        setSpeedMax(100); // denn eine Klassenmethode weiß nicht  
    }                     // welches Objekt hier gemeint ist!  
}
```

3. Klassenvariablen und Klassenmethoden

3.4. die main-Methode

- Die main-Methode ist also eine Klassenmethode
- Zusätzlich wird die umschließende Klasse dadurch **ausführbar**
- Jede Klasse mit einer main-Methode kann als **Start des Programms** genutzt werden

```
public class HelloWorld {  
    public static void main(String[] args) {  
        // PROGRAMM GESTARTET...  
        System.out.println("Hello, World");  
    }  
}
```

4. Einfaches Fehlerhandling

Wenn ein Fehler passiert, wird das JAVA-Programm normalerweise sofort beendet. Wenn man dies nicht will, kann man kritische Programmteile absichern und bei einem Fehler reagieren.

```
try {  
    // in diesem Block koennte etwas schief gehen,  
    // dann wird ein Fehler „geschmissen“  
} catch (Throwable th) {  
    // der Fehler wird „abgefangen“  
    // und dieser Block wird alternativ ausgeführt  
}
```

WICHTIG: Diese Prozedur soll auch helfen Fehler besser zu finden. Daher sollte man vermeiden zu viel zusammenhängenden Code innerhalb eines „try“ Blocks zu schreiben!

5. Exkurs „Planung und Entwurf“

5.1. Lastenheft und Pflichtenheft

Das Lastenheft beschreibt Anforderungen von einem Kunden an die Software. Das Pflichtenheft beschreibt alle Feature der Software, auf die sich Kunde und Entwickler geeinigt haben. Gegebenenfalls sind hier auch Hardwareanforderungen und weitere Details definiert. Beide Dokumente sind sehr umfangreich und meist auch Grundlage für einen Vertrag...

Diese Art der Planung ist ein guter allererster Schritt wenn man einen Auftrag bekommt oder ein großes Projekt durchführen will – ***aber für unsere Zwecke fiel zu weit gegriffen!***

5. Exkurs „Planung und Entwurf“

5.2. Designdokument und UML-Diagramme

Das (Software) Designdokument (**SDD**) ähnelt Lasten und Pflichtenheft, ist aber nicht so umfangreich und bei weitem nicht so strikt reglementiert. In Kombination mit **UML**-Diagrammen sind damit auch kleinen Programme VOR dem programmieren besser vorbereitet und sehr klar definiert. Sind SDD und UML gut, muss man beim Programmieren nicht mehr darüber nachdenken, was man überhaupt machen will.

Die Reihenfolge bei der Erstellung von SDD und UML ist nicht starr, manchmal ergibt sich durch das UML ein Detail im SDD. Übertragungen von Erkenntnis vom SDD zum UML sind der Regelfall.

Sollte man jedoch erst während des Programmierens zu viele Änderungen an SDD oder in den UML vornehmen müssen, zeugt das von... **suboptimaler Planung** oder unerwarteten Änderungen der Anforderungen...

5. Exkurs „Planung und Entwurf“

5.3. Beispiel: Aufgabe 4 von Teil 1:

Schreibe ein Programm zur Bestimmung des *kleinsten gemeinsamen Vielfaches (kgV)* zweier natürlicher Zahlen. Die Berechnung soll ohne Beteiligung des größten gemeinsamen Teilers erfolgen.

Beispiel: Zahl1 = 5, Zahl2 = 7, kgV = 35

5. Exkurs „Planung und Entwurf“

5.4. Umwandlung in ein Designdokument

Gegeben sind zwei natürliche Zahlen (**a** und **b**).
Gesucht wird das kleinste gemeinsame Vielfache.

Dies bedeutet ein Vielfaches **A** der **Zahl a** muss dasselbe ergeben wie das Vielfache **B** der **Zahl b**!

Lösungsansatz ausformulieren:

Wir starten mit dem Wert der jew. Zahl (**A = Wert a** und **B = Wert b**).
Anschließend ***erhöhen*** wir ***den kleineren Wert*** von **A** bzw. **B**
um den Wert a bei A bzw. b bei B **solange bis A und B gleich sind.**

5. Exkurs „Planung und Entwurf“

5.5. UML-Diagramme

Für dieses Beispiel:

das Ablaufdiagramm / Flussdiagramm

