Einführung in die Objekt-orientierte Programmierung

Dozent: Prof. Dr. Michael Eichberg
Kontakt: michael.eichberg@dhbw.de, Raum 149B
Version: 1.0

Folien: https://delors.github.io/prog-java-oo/folien.de.rst.html

https://delors.github.io/prog-java-oo/folien.de.rst.html.pdf

Fehler melden:

https://github.com/Delors/delors.github.io/issues

1. Objekt-orientierte Programmierung mit Java

Was ist Objektorientierte Programmierung?

Definition:

OOP ist ein Programmierparadigma, das auf den Konzepten von Klassen und **Objekten** basiert, die Daten und Funktionen kapseln.

Hauptziele:

Code **erweiterbar**, **strukturierter** und wiederverwendbar gestalten.

Hauptprinzipien:

- **Kapselung** (🇺🇸 *Encapsulation*)
- **Abstraktion** (🇺🇸 *Abstraction*)
- **Vererbung** (🇺🇸 *Inheritance*)
- **Polymorphie** (🇺🇸 *Polymorphism*) („vielerlei Gestalt“)

Während es in der Anfangszeit Programmiersprachen gab, die neben der prozeduralen Programmierung insbesondere auch die objektorientierte Programmierung unterstützten, unterstützen heute fast alle Programmiersprachen auch andere Paradigmen. Insbesondere die funktionale Programmierung.

Klassen

Klasse: Ein Bauplan für Objekte, der beschreibt, welche Daten bzw. Felder und Methoden ein Objekt haben kann.

Syntax: `class <Klassenname> { ... }`

Beispiel: *Auto* ist eine Klasse.

```
class Auto {
    // Felder (gel. auch Attribute genannt)
    String marke;
    int geschwindigkeit;

    // Methoden
    void beschleunigen(int wert) {
        geschwindigkeit += wert; // Zugriff auf das Feld des Objektes
    }
}
```

Objekte und die Selbstreferenz *this*

Objekt: Eine Instanz einer Klasse.

Definition:

`this` ist eine Referenz auf das aktuelle Objekt. Es wird verwendet, um auf die Felder und Methoden des aktuellen Objekts zuzugreifen.
Wenn es keine Zweideutigkeit gibt, dann kann auf die Angabe von `this` verzichtet werden.

Beispiel:

```
class Auto {
    String marke;
    int geschwindigkeit;

    void beschleunigen(int wert) {
        this.geschwindigkeit += wert; // this. ist hier optional
    }
    String alsString() {
        return "Auto: " + this.marke + " " + /*this.*/geschwindigkeit;
    }
}
```

Objekterzeugung/Instanziierung einer Java Klasse

Um eine Objekts zu erzeugen bzw. eine Klasse zu instanziiern, wird der `new` Operator verwendet.

Dieser Operator ...

- reserviert den benötigten Speicher, und bereinigt diesen ggf.
- ruft dann den Konstruktor der Klasse auf.

Syntax: `new <Klassenname>(<Parameter>)`

Beispiel: *meinAuto* referenziert ein Objekt der Klasse *Auto*.

```
class Auto {
    String marke;           // der Standardwert ist null
    int geschwindigkeit;    // der Standardwert ist 0

    void beschleunigen(int wert) { ... }
}
```

```
var meinAuto = new Auto(); // Aufruf des impliziten Konstruktors
```


Der Konstruktor ist eine spezielle Methode, die nur beim Erzeugen eines Objekts aufgerufen wird. Wird kein Konstruktor explizit definiert, wird ein (impliziter) Standardkonstruktor verwendet.

Der implizite Konstruktor ist ein Konstruktor, der automatisch vom Java compiler generiert wird, wenn kein Konstruktor explizit definiert wurde. Der implizite Konstruktor hat keine Parameter und initialisiert die Felder mit Standardwerten.

Explizite Konstruktoren

Ein Konstruktor hat immer den Namen der Klasse und kann Parameter enthalten.

Syntax: `<Klassenname>(<Parameter>) { ... }`

Beispiel:

```
class Auto {
    String marke;           // der Standardwert ist null
    int geschwindigkeit;    // der Standardwert ist 0

    Auto(String marke, int geschwindigkeit) {
        // ⚠️ "this." ist notwendig!
        this.marke = marke;
        this.geschwindigkeit = geschwindigkeit;notwendig!
    }
}
```

```
var meinAuto = new Auto("BMW",0); // Aufruf des impliziten Konstruktors
```


Ein Konstruktor hat (in Java) keinen Rückgabewert.

Es ist möglich, mehrere Konstruktoren zu definieren, wenn diese unterschiedliche Parameter haben.

Der Konstruktor wird aufgerufen, wenn ein Objekt erzeugt wird. Ein Konstruktor kann auch andere Konstruktoren der Klasse aufrufen.

Verwendung eines Objektes

Auf Felder und Methoden eines beliebigen Objektes kann über den **Punktoperator** zugegriffen werden.

Syntax: `<Objektinstanz>.<Attribut/Methode>`

Beispiel: *meinAuto* referenziert ein Objekt der Klasse *Auto*.

```
class Auto {
    String marke;
    int geschwindigkeit;
    void beschleunigen(int wert) { ... }
}
```

```
var meinAuto = new Auto();
meinAuto.marke = "BMW";
meinAuto.beschleunigen(10);
```

Abstraktion (Abstraction)

Definition:

Abstraktion bedeutet, die wesentlichen Eigenschaften und Funktionen eines Objekts hervorzuheben und Details zu verstecken, die für die Nutzung des Objekts nicht relevant sind.

Ziel: Details und Komplexität verstecken; d. h. wir möchten von unnötigen Details abstrahieren.

Beispiel: Eine *Form*-Klasse, die über verschiedene Unterklassen wie *Kreis*, *Quadrat* und *Dreieck* abstrahiert. Alle Formen bieten eine Möglichkeit zur Berechnung der Fläche obwohl diese ggf. sehr verschieden berechnet wird.

```
abstract class Form {
    abstract double berechneFlaeche();
}
```

```
class Kreis extends Form {
    double r = 0.0;
    double berechneFlaeche() {
        return Math.PI * r * r;
    }
}
```

```
class Quadrat extends Form {
    double seite = 0.0;
    double berechneFlaeche() {
        return seite * seite;
    }
}
```

Vererbung (🇺🇸 *Inheritance*)

Definition:

Erlaubt es, eine Klasse von einer anderen abzuleiten und deren Eigenschaften und Methoden zu erben.

Vorteile:

- Wiederverwendbarkeit des Codes
- Hierarchische Strukturierung

Beispiel: *Auto* als Basisklasse und *Elektroauto* als abgeleitete Klasse

```
class Auto {
    String marke;

    void fahren() {
        System.out.println("Das Auto fährt.");
    }
}

class Elektroauto extends Auto {
    int batteriestand;
```

Polymorphie (Polymorphism)

Definition:
Fähigkeit von Objekten, verschiedene Formen anzunehmen.

Typen:
- **Überladen** von Methoden (🇺🇸 *Compile-Time Polymorphism*)
- **Überschreiben** von Methoden (🇺🇸 *Runtime Polymorphism`*)

Vorteil: Ermöglicht flexiblen und dynamischen Code

Beispiel: Methode *fahren* wird in verschiedenen Klassen unterschiedlich implementiert.

```
class Auto {
    void fahren() {
        System.out.println("Das Auto fährt.");
    }
}

class Elektroauto extends Auto {
    void fahren() { // Überschreiben der Methode
        System.out.println("Das Elektroauto fährt leise.");
```

Zusammenfassung und Vorteile von Objekt-orientierter Programmierung[1]

Kapselung:

Schützt die Daten und kontrolliert den Zugriff.

Abstraktion:

Vereinfacht die Komplexität des Codes.

Vererbung:

Ermöglicht Code-Wiederverwendung und Hierarchien.

Polymorphie:

Erlaubt flexiblen Code durch unterschiedliche Implementierungen.

[1] Diese Vorteile gelten im Wesentlichen für alle objektorientierten Programmiersprachen.

Übung

Meine Erste Klassenhierarchie

Erstelle eine einfache *Tier*-Klasse mit einer Methode *lautGeben()*. Erstelle dann die Klassen *Hund* und *Katze*, die *Tier* erweitern, und überschreibe die Methode *lautGeben()* mit unterschiedlichen Ausgaben.

Meine Erste Klassenhierarchie

Erstelle eine einfache *Tier*-Klasse mit einer Methode *lautGeben()*. Erstelle dann die Klassen *Hund* und *Katze*, die *Tier* erweitern, und überschreibe die Methode *lautGeben()* mit unterschiedlichen Ausgaben.

Kapselung (🇺🇸 *Encapsulation*)[#]

Ziel: Daten eines Objekts vor direktem Zugriff von außen schützen.

Zugriff auf Daten erfolgt über **Getter** und **Setter**.

Vorteile:
- Schutz der Datenintegrität
- Kontrollierter Zugriff auf die Daten; fördert die Wartbarkeit

```
class Auto {
    private int geschwindigkeit;

    public int getGeschwindigkeit() {
        return geschwindigkeit;
    }

    public void setGeschwindigkeit(int geschwindigkeit) {
        if (geschwindigkeit >= 0) {
            this.geschwindigkeit = geschwindigkeit;
        }
    }
```

[2] Kapselung dient vor allem dem Programming-in-the-Large. Sprachen wie zum Beispiel Python bieten diesbezüglich zum Beispiel deutlich weniger Konzepte.