Einführung in die Objekt-orientierte Programmierung

Dozent: Prof. Dr. Michael Eichberg

Kontakt: michael.eichberg@dhbw.de, Raum 149B

Version: 1.1.1

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

Kontrollfragen: https://delors.github.io/prog-java-oo/kontrollfragen.de.rst.html

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

1. Objekt-orientierte Programmierung mit Java

Was ist Objektorientierte Programmierung?

Definition: Objektorientierte Programmierung (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*) (nächster Foliensatz)
- **Polymorphie** (*Polymorphism*) („vielerlei Gestalt“) (nächster Foliensatz)

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

Klassen

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

Syntax:

```
<Modifikator>* class <Klassenname> {
  (<Attributdeklarationen>|<Methodendeklarationen>)*
}
```

Beispiele:

`Auto` ist eine Klasse.

```
class Auto {
    // Attribute (gel. auch Felder (bzw. Fields) genannt)
    private String marke;
    private int geschwindigkeit; // _aktuelle_ Geschwindigkeit

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

`Button` (bei der Modellierung grafischer Benutzeroberflächen) ist eine Klasse.

```
class Button {
    private String text;
    private int state; // 0: normal, 1: pressed, 2: disabled

    void registerListener() { ... }
}
```

`BigDecimal` (zur Repräsentation von Dezimalzahlen mit „beliebiger“ Präzision) ist eine Klasse.

```
class BigDecimal {
    private int scale;
    private int precision;
    void add(BigDecimal b) { ... }
}
```

`File` (zum Zugriff auf Dateien) ist eine Klasse.

```
class File {
    private String name;
    private long size;
    void read() { ... }
}
```

Klassen ermöglichen es uns über konkrete Objekte zu **abstrahieren**: Klassen sind eine Beschreibung vieler Objekte mit gleichen Eigenschaften und Verhalten.

Durch die Verwendung von Sichtbarkeiten (insbesondere `private` und ggf. `protected`) ist der Zugriff auf die Attribute und Methoden einer Klasse von außen kontrollierbar.

Wir sprechen hier von **Kapselung**.

Die privaten Daten eines Objekts (und ggf. einiger Methoden) sind also geschützt und können nur über die Methoden der Klasse manipuliert werden. Dabei können alle Objekte einer Klasse auf die Attribute eines anderen Objektes derselben Klasse zugreifen.

Pro Java Datei (Datei mit der Endung .java), darf nur eine Klasse mit dem Modifikator `public` enthalten sein. Die Datei muss den Namen der Klasse haben.

Objekte

Sichtweisen:

- Objekte sind Instanzen von Klassen, die durch den Bauplan der Klasse definiert sind und durch spezifische Werte der Attribute charakterisiert werden.
- Objekte haben eine Identität und einen konkreten, eigenen Zustand.

Objekterzeugung/Instanziierung einer Java Klasse

Um ein Objekt zu erzeugen bzw. eine Klasse zu instanziiern, wird der `new` Operator verwendet. Der `new` Operator ...

- reserviert den benötigten Speicher für die Attribute, und stellt sicher, dass alle Attribute mit dem Defaultwert initialisiert sind.
- ruft dann den *Konstruktor* der Klasse auf.

  Der Konstruktor ist eine spezielle Methode, die einmalig beim Erzeugen eines Objekts aufgerufen wird und der Initialisierung des Objekts dient.

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 Attribute mit Standardwerten.

Variablen, die auf Objekte verweisen

Beispiel: Deklaration und Initialisierung einer Objektvariablen.

```
class Rectangle {
    int width;
    int height;
    int x;
    int y;
}
```

```
Rectangle a = new Rectangle();
```

bzw.

```
var b = new Rectangle();
```

- Objektvariablen sind *Referenzvariablen* und werden durch den Klassennamen gefolgt von einem Variablennamen deklariert.

  Syntax: `<Klassenname> <Variablenname>`

- Die Referenzvariable speichert eine Referenz (wie bei Feldern/Arrays) auf ein Objekt der Klasse.
- Bei Objekten gelten die gleichen Regeln beim Kopieren und Vergleichen wie bei Arrays (z. B. bzgl. des `==` Operators).
- Objektvariablen können überall dort deklariert werden, wo auch andere Variablen (für primitive Datentypen) deklariert werden können.
- Wie bei Arrays ist der Standardwert für Objektvariablen `null` und bedeutet, dass diese Variable auf *kein* Objekt verweist.
- Der Typ der Variablen ist durch die Klasse bestimmt.
- Der Name des Typs ist somit der Klassenname.
- Überall, wo ein primitiver Typ verwendet werden kann, kann auch der Typ eines Objektes verwendet werden.

Beispiel:

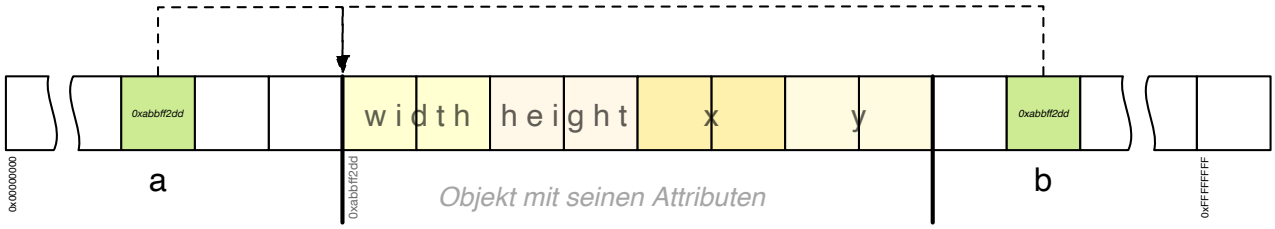
```
boolean compare(Rectangle a, Rectangle b) { } // Parameter
Person copy(Person a) { }                     // Rückgabetyp
double sum(Rectangle[] rectangles) { }        // Arrays
double intersect(Circle... circles) { }       // Varags
class Student { private String s; }           // Attribute

Car c = new Car();
Rectangle[] rs = new Rectangle[10];
Rectangle[] rs = new Rectangle[]{new Rectangle(), new Rectangle()};
```

Exemplarische Speicherbelegung

```
Rectangle a = new Rectangle();
Rectangle b = a;
```

Arbeitsspeicher (konzeptionell)



Objekt mit seinen Attributen

Bereinigung von Objekten

Verweist keine Referenzvariable mehr auf ein Objekt im Speicher, dann wird es automatisch vom *Garbage Collector* aus dem Speicher entfernt, d. h. der Entwickler muss sich nicht explizit um die Speicherbereinigung kümmern.

Es ist insbesondere nicht notwendig, Referenzvariablen auf `null` zu setzen.

Java unterstützt automatische *Garbage Collection*.

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 Attribute und Methoden des aktuellen Objekts zuzugreifen.

Beispiel:

```
1  class Auto {
2      String marke;
3      int geschwindigkeit;
4
5      void beschleunigen(int wert) {
6          this.geschwindigkeit += wert; // this. ist hier optional
7      }
8      String alsString() {
9          return "Auto: " + this.marke + " " + /*this.*/geschwindigkeit
10     }
11 }
```

Wenn es keine Zweideutigkeit gibt, dann kann auf die Angabe von `this` verzichtet werden. (Lokale Parameter und Variable überschatten ggf. Attribute der Klasse mit dem gleichen Namen.)

Explizite Konstruktoren

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

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

Beispiel:

```
1  class Auto {
2      String marke;           // der Standardwert ist null
3      int geschwindigkeit;    // der Standardwert ist 0
4
5      Auto(String marke, int geschwindigkeit) {
6          // ⚠ "this." ist notwendig,
7          //   zur Unterscheidung von Parameter und Attribut
8          this.marke = marke;
9          this.geschwindigkeit = geschwindigkeit;notwendig!
10     }
11 }
1  var meinAuto = new Auto("BMW",0); // Aufruf des Konstruktors
```

Ein Konstruktor kann auch andere Konstruktoren der Klasse aufrufen. Der „Methodenname“ der anderen Konstruktoren ist in diesem Fall `this`.

Beispiel:

```
1  class Auto {
2      String marke;
3      int geschwindigkeit;
4
5      Auto(String marke) { this.marke = marke; }
6
7      Auto(String marke, int geschwindigkeit) {
8          this(marke); // Aufruf des anderen Konstruktors
9
10         this.geschwindigkeit = geschwindigkeit;
11     }
12 }
1  void main() {  new Auto("VW", 0); }
```

Achtung!

Die Anzahl der Dinge, die vor dem Aufruf eines Konstruktors gemacht werden können, ist sehr begrenzt, damit eine konsistente Initialisierung gewährleistet ist.

Es ist zum Beispiel nicht möglich andere Methoden des Objekts aufzurufen (d. h. Aufrufe auf `this` sind nicht möglich).

Hinweis

Erst seit Java 22 ist es überhaupt möglich, dass vor dem Aufruf eines anderen Konstruktors Code ausgeführt werden darf.

Beispiel (seit Java 22):

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

    Auto(String marke) { this.marke = marke; }

    Auto(String marke, int geschwindigkeit) {
        if (geschwindigkeit < 0)
            throw new IllegalArgumentException("Geschw. < 0");
        this.geschwindigkeit = geschwindigkeit;
        this(marke); // Aufruf des anderen Konstruktors
    }
}
```

Initialisierungsfolge

1. Initialisierung der Attribute mit Standardwerten
2. Aufruf des Konstruktors des expliziten Konstruktors, wenn angegeben sonst des impliziten Konstruktors.
   (Dies führt ggf. zu weiteren Konstruktoraufrufen.)

Verwendung eines Objektes

Auf sichtbare Attribute 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);
```

Kapselung (🇺🇸 *Encapsulation*)[1]

Ziel: Daten eines Objekts vor direktem Zugriff von außen schützen. Best Practice: Zugriff auf Daten erfolgt über öffentliche **Getter** (Methoden, die mit `get` anfangen) und **Setter** (Methoden, die mit `set` anfangen). Alle Attribute (außer Konstanten) sollten **privat** sein.

Vorteile:

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

```
1 class Auto {
2     private int geschwindigkeit;
3     public int getGeschwindigkeit() { return geschwindigkeit; }
4     public void setGeschwindigkeit(int geschwindigkeit) {
5         if (geschwindigkeit >= 0) {
6             this.geschwindigkeit = geschwindigkeit;
7 }   }   }
```

Achtung!

Die Benennung von Gettern und Setter - wie dargestellt - ist umbedingt einzuhalten. Dies ist eine so etablierte Konvention, dass sie in den meisten modernen IDEs und vielen Tools automatisch unterstützt wird und auch von erweiterten Sprachkonstrukten genutzt wird.

Die Unterstützung von Sichtbarkeitskonzepten variieren von Programmiersprache zu Programmiersprache sehr stark. Sprachen wie zum Beispiel Python bieten diesbezüglich zum Beispiel deutlich weniger oder gar keine Konzepte. Obwohl fast alle Sprachen zumindest grundlegende Mechanismen für die Unterscheidung zwischen privaten und öffentlichen Daten und Methoden bieten. Sprachen wie Scala bieten jedoch noch weit ausgefeiltere Konzepte.

---

[1] Kapselung dient vor allem dem Programming-in-the-Large.

Übung

1.1. Bibliothek - Grundgerüst

Entwickeln Sie die Klassen `Buch`, `Exemplar`, `Benutzer`, `Bibliothek`.

Die Klassen haben die folgenden Attribute:

Bücher: Titel, ISBN, Jahr, Autoren
Exemplare: Exemplar-Nummer, Regal, Position
Benutzer: Benutzer-Nummer, Vorname, Nachname
Bibliotheken: Name der zugehörigen Institution, Standort

Es soll weiterhin gelten:

- Ein Buch hat max. 10 Exemplare.
- Ein Exemplar kann durch max. einen Benutzer ausgeliehen sein.
- Eine Bibliothek hat max. 100 Bücher und max. 20 Benutzer.

Übung

1.2. Bibliothek

- Entwickeln Sie Konstruktoren und folgende Methoden für die jeweiligen Klassen:

  Buch: `addExemplar(Exemplar ex, int nr)`, um ein Exemplar hinzuzufügen
  Exemplar: `verleihe(Benutzer b)`, um ein Buch auszuleihen
  Biblitohek: `addBenutzer(Benutzer b)`, `addBuch(Buch b)`
  Alle Klassen: `print()`, um alle Attribute auf der Kommandozeile auszugeben

- Entwickeln Sie eine `main()`-Methode, die eine Bibliothek der DHBW Mannheim erzeugt mit dem Standort Coblitzallee. Der Bibliothek sollen mindestens zwei Bücher und zwei Benutzer und jedem Buch mindestens ein Exemplar zugeordnet werden. Jeweils ein Exemplar ist an einen Benutzer verliehen.

  Abschließend soll die `main()`-Methode alle Informationen der Bibliothek über die Kommandozeile ausgeben.

Fehlerbehandlung und Validierung der Eingaben sind *noch nicht* notwendig.

Übung

1.3. Patientenakte - die Klasse Patient

- Die Klasse Patient hat folgende Attribute das Geburtsdatum (`String geburtsdatum`), einen `Namen (String name)`, ein Gewicht in Kilogramm (`double gewicht`) und eine Größe in Zentimetern (`int groesse`).
- Definieren Sie einen Konstruktor, der es ermöglicht einen Patienten wie folgt zu erzeugen: `new Patient("24.12.2024", "Max Müller", 180, 80d)`
- In einer (externen) `main` Methode:
  - Legen Sie mehrere Patienten an und speichern Sie diese in einem Array `patienten`.
  - Schreiben Sie eine Methode, die die Durchschnittsgröße aller Patienten berechnet. Rufen Sie die Methode auf und geben Sie das Ergebnis aus.

Achten Sie ggf. auf die Datentypen bei den Berechnungen.

Übung

1.4. Patient mit Getter und Setter Methoden

- Machen Sie alle Attribute der Klasse `Patient` `private`.
- Implementieren Sie für jedes Attribut eine Getter Methode.
  D. h. eine Methode, die einfach den Wert des Attributs direkt zurückgibt.
- Passen Sie Ihre Main-Methode entsprechend an.

Zur Erinnerung

Ein Getter fängt immer mit `get` an und hat den Namen des Attributs als Suffix (z. B. `getGeburtsdatum`).

Übung

1.5. Patient und Arzt

- Legen Sie eine Klasse Arzt an, die ein privates Attribut vom Typ Array („Feld“) von Patienten hat.
- Fügen Sie der Klasse Arzt eine Methode `addPatient` hinzu, die ein neues Array erzeugt, das alle bisherigen Patienten des Arztes und den neuen Patienten enthält. Stellen Sie sicher, dass der Patient nur einmal hinzugefügt wird. Sollte der Patient schon in der Liste sein, dann passiert nichts.

  (Gehen Sie für diese Aufgabe davon aus, dass jeder echte Patient immer nur durch genau ein Objekt repräsentiert wird.)
- Verschieben Sie die Methode zur Berechnung der Durschnittsgröße in die Klasse Arzt, um die Durschnittsgröße aller Patienten des Arztes zu berechnen.

Sie können ggf. zum Vergrößern des Arrays die Methode `java.util.Arrays.copyOf` verwenden.