Karlsruher Institut für Technologie

6. Tutorium
2. Blatt, Vererbung

Tutorium 14

Péter Bohner | 07.12.2022

Inhaltsverzeichnis

Wiederholung

Was ist ein Beispiel für Narrowing Conversion?

```
byte a = (byte) 1234; // a == 214
```

Kann eine Narrowing Conversion ohne eine Casting Conversion auftreten?

Nein.

Was kann bei der Narrowing Conversion auftreten?

- Informationsverlust bezüglich Genauigkeit und Größe

Was kann bei der Widening Conversion auftreten?

- Informationsverlust bezüglich Genauigkeit
- z.B. `float a = 23456789; // a == 23456788.0`

Besitzt jedes Objekt eine toString()-Methode?

Ja, immer.

Welche Sichtbarkeiten gibt es?

- `private`, „default“, `protected`, `public`

Wiederholung

Welche Sichtbarkeit sollten Attribute standardmäßig haben?

`private`

Welche Aussagen treffen auf Datenkapselung zu?

- Variablen werden kontrolliert durch Methoden verändert
- Interne Strukturen bleiben verborgen nur Schnittstelle bekannt

Worüber soll der Zugriff auf einen abstrakten Datentypen idealerweise stattfinden?

Über eine Schnittstelle.

Welche Attribute muss ein Knoten einer einfach verketteten Liste haben?

- Referenz auf den Folgeknoten
- Element eines bestimmten Datentyps

Das trifft normalerweise auf eine Liste zu:

- dynamische Länge
- Zugriff auf einzelne Elemente erschwert
- Einfügen an gewünschten Stellen leicht
- Innere Implementierung nicht bekannt (**Geheimnisprinzip**)

Besprechung des 2. Übungsblattes

Allgemein

- Viele Teilnahmen, super! Im Schnitt (A: 22/24, B: 20/24, C: 19/24)
- Achtet auf gute (Variablen-)namen; Ohne unnötige Abkürzungen und in camelCase
- so: **calculateCheckSum()** nicht: **calcchksum()**
- Packages verwenden
- `if (a) { return true; } else { return false; }` ist unnötige Komplexität
- Formatiert bevor ihr abgibt

Keine Abgaben im Onlineeditor!

Die Abgabe ist bis auf weiteres nur mit git oder dem artemis-eclipse möglich.
Kann das jeder?

Zu den einzelnen Aufgaben

A - StringUtils

- verwendet die Methoden wieder, z.B. `word.equals(reverse(word))`
- geeignete Schleifen verwenden

Zu den einzelnen Aufgaben

B - IBAN

- OO-Modellierung war sinnvoll, aber nicht erforderlich
- Macht (private) Hilfsmethoden!
- Vermeidet Code-duplikation (z.B. countryCode)
- Sinnvolle Aufteilung der Funktionalität (lieber **toString()** als **print**))

Zu den einzelnen Aufgaben

C - MagicSquare

- Kommetiert euren code
- vermeidet Duplikationen der Logik
- Macht mehrere Klassen, MagicSquare soll keine Gottklasse sein (Matrix, Parser, etc.)

Vererbung - Einleitung

Cabriolet, Pickup, Motorrad, Sattelzug, Schwertransporter

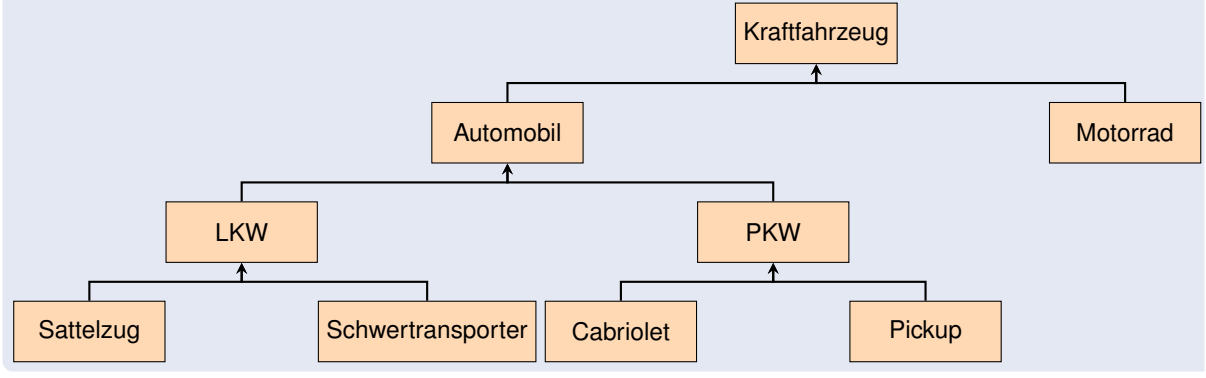
Kraftfahrzeuge

- Gibt es gemeinsame Eigenschaften?
  - z.B. Tankfüllung, PS, Kennzeichen
- Gibt es gemeinsame Verhaltensweisen?
  - fahren, bremsen
- Gibt es unterschiedliche Verhaltensweisen?
  - nicht gleiches Fahrverhalten
  - Capriolet kann Dach öffnen/schließen andere Fahrzeuge vielleicht nicht

Vererbung - Einleitung

Cabriolet, Pickup, Motorrad, Sattelzug, Schwertransporter

Findet ihr Oberbegriffe/Kategorien?



Vererbung- Einleitung

Wo kommt jetzt die Veerbung ins Spiel?

- Manche Eigenschaften/Verhaltensweisen hat jedes Fahrzeug. (Generalisierung)
  - Jedes Fahrzeug kann fahren - egal welches man gerade betrachtet.
- manche Fahrzeuge (z.B. Cabriolet) haben zusätzliche Verhaltensweisen (Spezialisierung)
- Konzept der Gemeinsamkeiten im Verhalten und/oder Eigenschaften liegt der Vererbung zu Grunde.

Vererbung - Wozu?

- Variantenbildung
- Substituierbarkeit (Unterklasse im Kontext der Oberklasse einsetzbar)
- Spezialisierung bzw. Generalisierung
- Code- Wiederverwendung (`getFuelLevel()` vielleicht überall gleich)
  - Weniger Redundanzen → höhere Wartbarkeit

Vererbung - Java

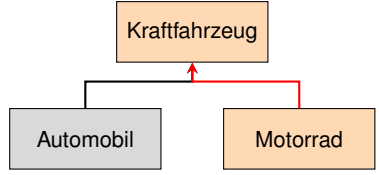
- Modelliert ist-ein-Beziehung (is-a)
  - Ein Cabriolet ist ein PKW, ein PKW ist ein Automobil und ein Automobil ist ein Kraftfahrzeug (Vererbungshierachie)
  - Jedes Unterklassenobjekt automatisch auch Oberklassenobjekt
    - Umgekehrt allerdings nicht !
- Kennzeichnung mittels Schlüsselwort **`extends`**
  - Schema: `class B extends A {...}`
- keine Mehrfachvererbung in Java (nur eine Oberklasse je Klasse)
- Jede Klasse erbt implizit von `Object` (`toString, equals, hashcode, ...`)
- Beziehung zwischen Ober- und Unterklasse
  - **nicht private** Methoden, Attribute und geschachtelte Klassen (nested classes) werden vererbt
  - Konstruktoren werden nicht vererbt
  - Unterklasse können Methoden und Attribute hinzufügen
  - Unterklassen können Methoden umdefinieren

Vererbung - Beispiel

```java
public class Kraftfahrzeug {
    private int fuelLevel;
    private LicensePlate licensePlate;

    public int getFuelLevel() {
        return this.fuelLevel;
    }
}

public class Motorrad extends Kraftfahrzeug {
    private boolean sattelTasche;
    // Auch Methoden werden geerbt. Man kann auch neue Methoden hinzufügen
    public void wheely() {...}
}
```



Vererbung - Überschreiben

- Will man Methodenverhalten von Oberklassen ändern, so kann man diese Methoden überschreiben
  - Beispiel: `toString()`-Methode
- Methodensignatur muss identisch sein (Methodenname, Parameterliste)
  - Rückgabetyp gleich oder Unterklasse vom ursprünglichen Rückgabetyp
- `@Override` Annotation stellt sicher, dass Überschreiben tatsächlich stattfindet (Sonst Compilerfehler)

Vererbung - Überschreiben

```java
public class Human {
    public void sayHello() {
        System.out.println("Hello");
    }
}

public class GermanHuman extends Human {
    @Override
    public void sayHello() {
        System.out.println("Hallo");
    }
}
```

```java
public class SpanishHuman extends Human {
    @Override
    public void sayHello() {
        System.out.println("Hola");
    }
}
```

Vererbung - Polymorphie

- Es besteht die Möglichkeit, dass Objekte einer Kindklasse wie Objekte ihrer Elternklasse auftreten können
- Erfordert eine Operation einen Parameter vom Typ einer Elternklasse, kann ohne Weiteres ein Objekt vom Typ einer Kindklasse übergeben werden
- Das Objekt der Kindklasse bietet mindestens alle Methoden und Attribute, wie ein Objekt der Elternklasse
  - Es bietet sogar noch mehr!

Vererbung - Dynamische Bindung

Dynamische Bindung

- Methodenaufrufe werden zur Laufzeit dynamisch an die Methodendefinition gebunden.
  - → Java wählt immer die „spezialisierteste“ verfügbare Methode aus!
  - Entscheidungsgrundlage: dynamischer Typ (tatsächlicher Typ des Objektes zur *Laufzeit*)

Beispiel

```
Human[] humans = {new GermanHuman(), new SpanishHuman(), new Human()};

for (Human human : humans) {
    human.sayHello();
}
```

Ausgabe:

```
Hallo Hola Hello
```

Vererbung - super

- ermöglich Zugriff auf **nicht private** Attribute und Methoden im Namensraum der Oberklasse
- hebt dynamische Bindung einmalig auf
- ermöglicht Kindklassen z.B. auf überschriebene Methode der Oberklasse trotzdem zuzugreifen

`this` vs `super`

`this`	`super`
Das aktuelle Objekt	aktuelles Objekt, aber Namensraum der Elternklasse

Vererbung - super

Beispiel

```java
public class PKW {
    private boolean isLocked;

    public void lock() {
        isLocked = true;
    }
}
```

```java
public class Cabriolet extends PKW {
    private boolean isRoofOpen;
    @Override
    public void lock() {
        super.lock();
        isRoofOpen = false;
    }
}
```

Vererbung - Konstruktoren

- Konstruktoren werden **nicht** vererbt
- Aber: Unterklassen Objekt soll immer im Kontext der Oberklasse nutzbar sein
  - Attribute der Oberklasse müssen initialisiert sein

  $\Rightarrow$ Konstruktor der Oberklasse muss beim Erstellen eines Objektes aufgerufen werden
- Im Konstruktor der Unterklasse wird als **erstes** mit `super(Parameterliste)` der Oberklassenkonstruktor aufgerufen

```java
public class Kraftfahrzeug {
    private int fuelLevel;
    public Kraftfahrzeug(int fuelLevel) {
        this.fuelLevel = fuelLevel;
    }
}
```

```java
public class Motorrad extends Kraftfahrzeug {
    private boolean sattelTasche;
    public Motorrad(int fuelLevel,
                    boolean sattelTasche) {
        super(fuelLevel);
        this.sattelTasche = sattelTasche;
    }
}
```

Vererbung - final

Vererbung verhindern durch `final`

- `final` bei Methoden verhindert das Überschreiben
- `final` bei Klassen verhindert das Ableiten der Klasse

instanceof

`instanceof` prüft, ob ein konkretes Objekt einen gewüschten Datentyp hat. (gibt boolean zurück)
Syntax: `objekt instanceof Klasse`

Beispiele

```
new Motorrad() instanceof Kraftfahrzeug;        // true
new Schwertransporter() instanceof Automobil;   // true
new Cabriolet() instanceof Motorrad;            // false
new PKW() instanceof Cabriolet;                 // false
```

Casting

Up-Cast: Typumwandlung in höhere Klasse:

```
Kraftfahrzeug k = new Motorrad(); // Harmlos!
```

Down-Cast: Typumwandlung in speziellere Klasse:
Nur wenn man sich *absolut* sicher ist, dass ein Objekt/eine Variable von einem bestimmten spezielleren Typ ist, kann man Umwandlung erzwingen:
Schema: `(Unterklasse) variable`

```
Kraftfahrzeug k = new Motorrad();
if (k instanceof Motorrad) {
    Motorrad m = (Motorrad) k;
}
```

Achtung: Down-Cast nie ohne Typüberprüfung!

Abstrakte Klassen - Einführung

Benutze das Schlüsselwort **`abstract`**, wenn:

- Keine oder unvollständige Implementierung angegeben werden soll.
- Instanziierung mit **`new`** verhindert werden soll (Objekt des Typs Säugetier ergibt keinen Sinn).
- Man nur die Schnittstelle vorgeben will (z.B. alle Kraftfahrzeug sollen Methode `drive()` anbieten. Die Implementierung ist den Unterklassen überlassen)

Abstrakte Klassen - Beispiel

Methoden in abstrakten Klassen können entweder nur den Methodenkopf definieren, oder eine (Standard-)Implementierung angeben.
Wenn die Unterklasse nicht auch abstract ist, müssen abstrakte Methoden von ihr implementiert werden.

```java
public abstract class Kraftfahrzeug {
    private int fuelLevel;

    public void refuel(int amount) { // nicht abstract
        fuelLevel += amount;
    }

    public abstract void drive(Destination destination); // <-- Semikolon nicht vergessen
}
```

Vererbung - Übung

Was ist *gültiges* Java?

Animal animal = new Mammal(); ✓
Animal animal = new Animal(); ✗
Mammal mammal = new Mammal(); ✓
Animal animal = new Dog(); ✓
Dog dog = new Mammal(); ✗

```java
abstract class Animal {
    // Attribute, Methoden,...
}
class Mammal extends Animal {
    // Attribute, Methoden,...
}
class Dog extends Mammal {
    // Attribute, Methoden,...
}
```

Aufgabe

Super Mario Bros ©

Super Mario Bros soll neu implementiert werden. Dazu sollt ihr einige **Charaktere** implementieren:

- Alle Charaktere bekommen eine `go()`-Methode.
- Jeder Charakter hat eine bestimmte Zahl an geschafften `Levels` und `gespielten Minuten`. Außerdem hat jeder Charakter eine `Laufgeschwindigkeit` von 2 m/s. Die `Sprunghöhe` wird beim Erstellen spezifiziert. Wird ein negativer Wert übergeben, beträgt die Sprunghöhe automatisch 100cm.
- `Luigi` gibt bei `go()` „Let's-a go!“ aus, Mario „It's a-me, %name!“. %name ist der Name des Charakters. Es gibt nur eine Art von Luigi, aber es kann spezielle Marios geben, z.B. Katzen-Marios. Diese haben allerdings alle dieselbe `go()`-Methode.

Lösung

```java
package edu.kit.wittemund.mariobros.character;

public abstract class Character {

    protected static final int WALK_SPEED = 2;
    private int jumpHeight = 100;
    private int completedLevels;
    private int playedMinutes;
    private String name;
```

Lösung

```
    public Character(String name, int jumpHeight) {
        this.name = name;
        if (jumpHeight >= 0) {
            this.jumpHeight = jumpHeight;
        }
        this.completedLevels = 0;
        this.playedMinutes = 0;
    }
    public String getName() {
        return this.name;
    }
    public abstract void go();
}
```

Lösung

```java
package edu.kit.wittemund.mariobros.character;

public final class Luigi extends Character {

    public Luigi(int jumpHeight) {
        super("Luigi", jumpHeight);
    }

    @Override
    public void go() {
        System.out.println("Let's-a go!");
    }
}
```

Lösung

```java
package edu.kit.wittemund.mariobros.character;

public class Mario extends Character {

    public Mario(int jumpHeight) {
        super("Mario", jumpHeight);
    }

    @Override
    public final void go() {
        System.out.println("It's a-me, " + super.getName() + "!");
    }

}
```

Bis zum nächsten Tutorium am 14.12.2022!