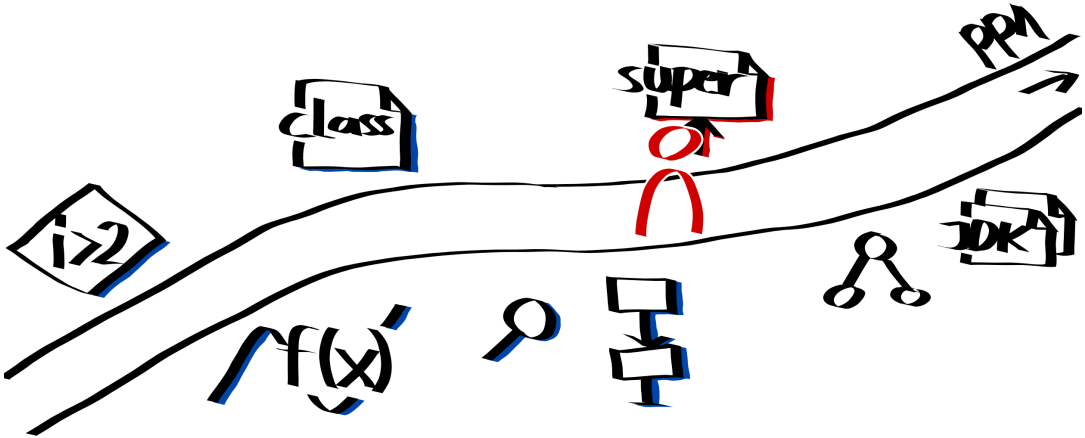
Kapitel 6: Vererbung & Fehlerbehandlung

VL 19: Vererben & Überschreiben

Wintersemester 2023/24

Stand 13. Dezember 2023, 13:00 Uhr

Wo stehen wir gerade?



Vorlesungsfeedback

Vererbung

```java
public class Vehicle {

  private double speed = 0;

  public void accelerate(double speed) {
    this.speed += speed;
  }

  public double getSpeed() {
    return speed;
  }

}
```

```java
public class Car extends Vehicle {

  private int charge = 0;

  public void chargeBattery(int charge) {
    this.charge += charge;
  }

  public int getCharge() {
    return charge;
  }

}
```

- Vehicle ist *Oberklasse* (oder: *Basisklasse*, *Superklasse*) von Car. (Generalisierung)
- Car ist *Unterklasse* (oder: *abgeleitete Klasse*) von Vehicle. (Spezialisierung)
- Car *erbt* von Vehicle.

Vererbung von Instanzmethoden

- Unterklasse hat automatisch Zugriff auf alle nicht-privaten Methoden und Variablen von Oberklasse
- Objekt besitzt im Heap alle Eigenschaften (Objektvariablen), auch geerbte

```
1 Car car = new Car();
2 car.chargeBattery(4);
3 car.accelerate(10);
4
5 System.out.println(car.getCharge());
6 System.out.println(car.getSpeed());
```

```
% java VehicleDemo
4
10.0
```

Funktioniert das?

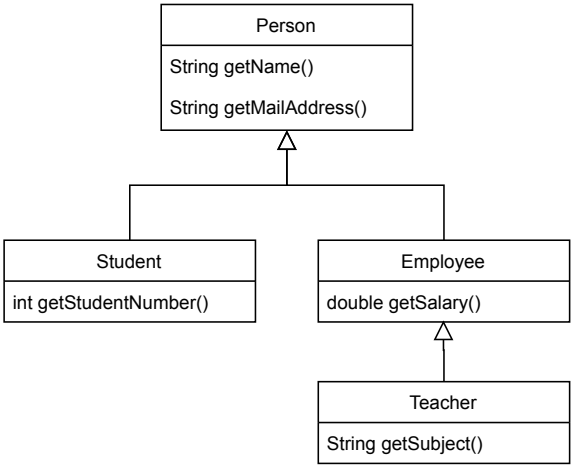
```
1 Vehicle car2 = new Car();
2 car2.accelerate(10);
3 car2.chargeBattery(4);
```

Funktioniert das?

```
1 Vehicle car2 = new Car();
2 car2.accelerate(10);
3 car2.chargeBattery(4);
```

- Objekte können in Variable vom Typ der Oberklasse gespeichert werden (Polymorphie)
- Aber: nur noch Methoden des deklarierten Typs verwendbar
  - ⇒ Zeile 2 klappt, aber Zeile 3 klappt **nicht**

Beispiele für Vererbungshierarchien: Personen



Ergibt der Methodenaufruf Sinn?

Heinrich Heine Universität Düsseldorf

```java
public static Employee richerPerson(Employee employee1, Employee employee2) {
    if(employee1.getSalary() < employee2.getSalary()) {
        return employee2;
    }
    return employee1;
}
```

```java
Teacher sascha = new Teacher("Sascha", "1234@hhu.de", 4074, "Progra");
Employee kim = new Employee("Kim", "1337@hhu.de", 5872);

Employee e1 = richerPerson(sascha, kim);
```

Ergibt der Methodenaufruf Sinn?

Heinrich Heine Universität Düsseldorf

```java
public static Employee richerPerson(Employee employee1, Employee employee2) {
    if(employee1.getSalary() < employee2.getSalary()) {
        return employee2;
    }
    return employee1;
}
```

```java
Teacher sascha = new Teacher("Sascha", "1234@hhu.de", 4074, "Progra");
Employee kim = new Employee("Kim", "1337@hhu.de", 5872);

Employee e1 = richerPerson(sascha, kim);
```

Ja!

→ Methode, die etwas mit einem Typen sinnvoll anfangen kann, kann es auch mit allen Untertypen

Ergibt der Methodenaufruf Sinn?

```java
public static Employee richerPerson(Employee employee1, Employee employee2) {
    if(employee1.getSalary() < employee2.getSalary()) {
        return employee2;
    }
    return employee1;
}
```

```java
Person alice = new Person("Alice", "1265@hhu.de");
Person bob = new Person("Bob", "1532@hhu.de");

Employee e2 = richerPerson(alice, bob);
```

Ergibt der Methodenaufruf Sinn?

Heinrich Heine Universität Düsseldorf

```java
public static Employee richerPerson(Employee employee1, Employee employee2) {
    if(employee1.getSalary() < employee2.getSalary()) {
        return employee2;
    }
    return employee1;
}
```

```java
Person alice = new Person("Alice", "1265@hhu.de");
Person bob = new Person("Bob", "1532@hhu.de");

Employee e2 = richerPerson(alice, bob);
```

Nein (Compilezeitfehler)

- Methode funktioniert nur sinnvoll mit Employee und Untertypen

Ergibt der Methodenaufruf Sinn?

```java
public static Employee richerPerson(Employee employee1, Employee employee2) {
    if(employee1.getSalary() < employee2.getSalary()) {
        return employee2;
    }
    return employee1;
}
```

```java
Teacher eve = new Teacher("Eve", "234@hhu.de", 4074, "Progra");
Person trudy = new Employee("Trudy", "007@hhu.de", 5872);

Employee e3 = richerPerson(eve, trudy);
```

Ergibt der Methodenaufruf Sinn?

Heinrich Heine Universität Düsseldorf

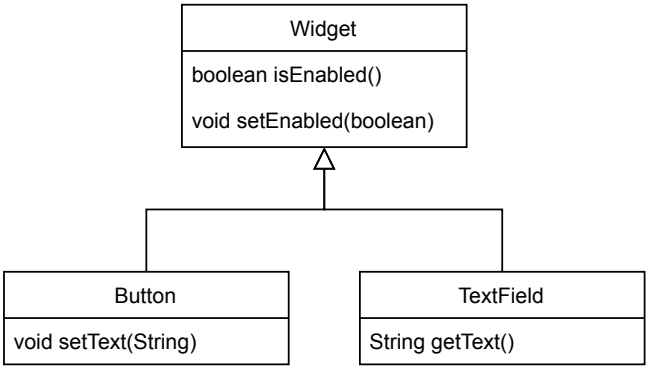
```java
public static Employee richerPerson(Employee employee1, Employee employee2) {
    if(employee1.getSalary() < employee2.getSalary()) {
        return employee2;
    }
    return employee1;
}
```

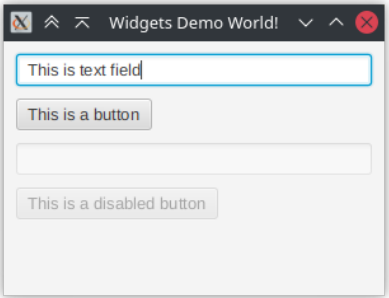
```java
Teacher eve = new Teacher("Eve", "234@hhu.de", 4074, "Progra");
Person trudy = new Employee("Trudy", "007@hhu.de", 5872);

Employee e3 = richerPerson(eve, trudy);
```

Nein (Compilezeitfehler)

→ Variablentyp zur Compilezeit ist entscheidend

Beispiele für Vererbungshierarchien: UI-Widgets





Ist diese Vererbung sinnvoll?

Heinrich Heine Universität Düsseldorf

Ein Fahrrad hat einen Sattel:

```java
import java.awt.Color;

public class Sattel {
    private Color farbe;

    public void setFarbe(Color farbe) {
        this.farbe = farbe;
    }
}
```

```java
public class Fahrrad extends Sattel {
    private double aktuelleEnergie;

    public void laden(double energie) {
        this.aktuelleEnergie +=
          ↪   energie;
    }
}
```

Ist diese Vererbung sinnvoll?

Heinrich Heine Universität Düsseldorf

Ein Fahrrad hat einen Sattel:

```java
import java.awt.Color;

public class Sattel {
    private Color farbe;

    public void setFarbe(Color farbe) {
        this.farbe = farbe;
    }
}
```

```java
public class Fahrrad extends Sattel {
    private double aktuelleEnergie;

    public void laden(double energie) {
        this.aktuelleEnergie +=
          ↪   energie;
    }
}
```

Faustregel: Vererbung nur sinnvoll, wenn eine Ist-ein-Beziehung besteht:

- Ein Fahrrad *ist ein* Fahrzeug.
- Ein Fahrrad *ist ein* Sattel.

Bessere Lösung?

Bessere Lösung?

→ Hat-ein-Beziehungen über Instanzvariablen darstellen

```java
public class GutesFahrrad {
    private double aktuelleEnergie;
    private Sattel sattel;

    public void charge(double energie) {
        this.aktuelleEnergie += energie;
    }
}
```

Liskovsches Substitutionsprinzip

Heinrich Heine Universität Düsseldorf

Regel

Wenn U von O erbt, dann müssen überall dort, wo O-Objekte verwendet werden können, auch U-Objekte verwendet werden können, ohne dass sich gewünschte Eigenschaften (z. B. Korrektheit) verändern.
Formal: Wenn für alle O-Objekte eine Eigenschaft P gilt, dann muss P auch für alle U-Objekte gelten.

Liskovsches Substitutionsprinzip

Heinrich Heine Universität Düsseldorf

Regel

Wenn U von O erbt, dann müssen überall dort, wo O-Objekte verwendet werden können, auch U-Objekte verwendet werden können, ohne dass sich gewünschte Eigenschaften (z. B. Korrektheit) verändern.
Formal: Wenn für alle O-Objekte eine Eigenschaft P gilt, dann muss P auch für alle U-Objekte gelten.

`Quadrat extends Rechteck`
Ist das eine gute Idee?

Liskovsches Substitutionsprinzip

Heinrich Heine Universität Düsseldorf

Regel

Wenn U von O erbt, dann müssen überall dort, wo O-Objekte verwendet werden können, auch U-Objekte verwendet werden können, ohne dass sich gewünschte Eigenschaften (z. B. Korrektheit) verändern.
Formal: Wenn für alle O-Objekte eine Eigenschaft P gilt, dann muss P auch für alle U-Objekte gelten.

`Quadrat extends Rechteck`
Ist das eine gute Idee?

- kommt auf konkrete Klassen an
- Gibt es `Rechteck.setWidth`, `Reckteck.setHeight`, `Reckteck.getArea`?
  $\Rightarrow$ Vererbung schlechte Idee (Warum?)[1]

[1] auch bekannt als Kreis-Ellipse-Problem

Überschreiben von Instanzmethoden

Heinrich Heine Universität Düsseldorf

- Unterklassen können Verhalten von **Instanzmethoden** der Oberklasse anpassen
  - ! Signaturen müssen exakt gleich sein
- zur Laufzeit entschieden, welche Instanzmethode in welcher Klasse aufgerufen wird (Laufzeitpolymorphismus)
  - zuerst: Definiert Klasse der Instanz die Methode?
  - wenn nein: Nachschlagen in direkter Oberklasse
    - Oberklasse schlägt ggf. in ihrer Oberklasse nach usw.
  - wenn in keiner Oberklasse vorhanden: Compilerfehler

Beispiel zum Überschreiben

Heinrich Heine Universität Düsseldorf

```java
public class Ball extends FlyingObject {

  private final double radius;

  public Ball(double radius) {
    this.radius = radius;
  }

  public String toString() {
    return "Farbe: " + getColor()
        + ", r=" + radius;
  }

}
```

```java
import java.awt.Color;

public class FlyingObject {

  private Color color;

  public FlyingObject() {
    int c = (int)(Math.random() * 255);
    this.color = new Color(c, c, c);
  }

  public Color getColor() {
    return color;
  }

  public String toString() {
    return "Farbe: " + color;
  }

}
```

Wird hier immer der Radius mit ausgegeben?

Heinrich Heine
Universität Düsseldorf

```
1 Ball ball1 = new Ball(10);
2 System.out.println(ball1.toString());
3
4 FlyingObject ball2 = new Ball(20);
5 System.out.println(ball2.toString());
```

Wird hier immer der Radius mit ausgegeben?

Heinrich Heine Universität Düsseldorf

```java
Ball ball1 = new Ball(10);
System.out.println(ball1.toString());

FlyingObject ball2 = new Ball(20);
System.out.println(ball2.toString());
```

```
% java Balls
Farbe: java.awt.Color[r=161,g=161,b=161], r=10.0
Farbe: java.awt.Color[r=41,g=41,b=41], r=20.0
```

- Java schlägt zuerst in der Klasse der `ball2`-Instanz nach

→ entscheidend ist, welche Klasse das Objekt hat, das zur Laufzeit in der Variable steht

! gilt nur für Instanzmethoden

`super` in Methoden

- Zweck: Aufruf der Implementierung in der Oberklasse
  → Wiederverwendung von Code

- hier: `super.toString()` ruft `toString()` in Oberklasse auf

```java
public class Ball extends FlyingObject {

    private final double radius;

    public Ball(double radius) {
        this.radius = radius;
    }

    public String toString() {
        return super.toString() + ", r=" + radius;
    }

}
```

Die Annotation `@Override`

Heinrich Heine Universität Düsseldorf

```
9  @Override
10 public String toString() {
11     return super.toString() + ", r=" + radius;
12 }
```

- Darf vor Methode stehen, die eine Methode überschreibt
- Warum ist das sinnvoll?

Die Annotation `@Override`

Heinrich Heine Universität Düsseldorf

```
9  @Override
10 public String toString() {
11     return super.toString() + ", r=" + radius;
12 }
```

- Darf vor Methode stehen, die eine Methode überschreibt
- Warum ist das sinnvoll?
  - Compiler prüft, ob Methode in Oberklasse oder implementiertem Interface existiert
  - Vermeidung von Tippfehlern (z. B. nicht exakt gleiche Signatur)

  $\Rightarrow$ zur Sicherheit immer `@Override` verwenden

- Programmierpraktikum: weitere Annotation (die auch Verhalten zur Laufzeit beeinflussen)

Die Klasse `Object`

Heinrich Heine Universität Düsseldorf

- Wenn Klasse nicht explizit erbt: erbt von `Object`
  - ⇒ Alle Klassen erben direkt oder indirekt von `Object`
- Object enthält u. a.[2] Standardimplementierungen von
  - `String toString()`: gibt Klassennamen und sog. Hashcode$^{\rightarrow \text{später}}$ aus
    - ⇒ wir haben die ganze Zeit schon dieses Standardverhalten überschrieben
  - `boolean equals(Object)`: prüft Gleichheit der Referenzen (wie `==`)

[2] für Interessierte: vollständige Liste unter
`https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html`

`println` revisited

```
1 System.out.println(ball1.toString());
```

- Ruft `System.out.println(String)` auf

`println` revisited

Heinrich Heine
Universität Düsseldorf

```
1 System.out.println(ball1.toString());
```

- Ruft `System.out.println(String)` auf

```
2 System.out.println(ball1);
```

`println` revisited

```
1 System.out.println(ball1.toString());
```

- Ruft `System.out.println(String)` auf

```
2 System.out.println(ball1);
```

- Ruft `System.out.println(Object)` auf
- `println` wiederum ruft `Object.toString()` auf
- `Object.toString()` wurde durch `Ball.toString()` überschrieben

Lernziele

Heinrich Heine
Universität Düsseldorf

Sie können am Ende der Woche ...

- Klassen **schreiben**, die von anderen Klassen erben.
- Polymorphie im Zusammenhang mit Vererbung **benutzen**.
- Instanzmethoden **überschreiben**.
- `super` **verwenden**, um Methoden der Oberklasse aufzurufen.

Wichtige Begriffe

Heinrich Heine Universität Düsseldorf

Vererbung

Unterklasse

hat-ein

`@Override`

extends

Vererbungshierarchie

Überschreiben

Object

Oberklasse

ist-ein

`super.toString()`