



LUDWIG-
MAXIMILIANS-
UNIVERSITÄT
MÜNCHEN



 mobile and
distributed systems group



Javakurs für Anfänger

Einheit 12: Mehr zur Vererbung und abstrakte Klassen

Kyrill Schmid

Lehrstuhl für Mobile und Verteilte Systeme



1. Teil: Mehr zur Vererbung

- Methoden vererben und überschreiben
- Dynamisches Binden
 - Polymorphismus

2. Teil: Abstrakte Klassen

- Motivation
- Konzept

Praxis:

- Übung zur Vererbung
 - Pizzabestellung
- Übung zu abstrakten Klassen
 - Geometrische Figuren

Lernziele

- Methoden überschreiben können
- Das Konzept der Vererbung weiter einüben
- Abstrakte Klassen kennenlernen und benutzen können

Zur Wiederholung:

- Nur **public** bzw. **protected** deklarierte Methoden werden vererbt
- **Private** Methoden werden nicht vererbt, können aber noch von vererbten Methoden benutzt werden

Geerbte Methoden können überschrieben werden, um diese

- zu **erweitern**
- komplett zu **verändern**


Überschreiben einer Methode in der Unterklasse erfolgt durch gleiche Definition der Methode aus der Oberklasse

- Gleicher **Methodenname**
- Gleiche **Parameter**
- Gleicher **Rückgabotyp**
- Verwendung der **@Override** Annotation vor der überschreibenden Methode zur Kompilierprüfung

Beispiel zum Überschreiben einer Methode aus der Oberklasse

- Zur **Erweiterung** der Funktionalität
 - Aufruf der Oberklassenmethode mittels `super.Oberklassenmethode(Parameter);`

```
//Oberklasse:  
public class Person {  
  
    private String name;  
    private String vorname;  
  
    public Person(String name, String vorname) {  
        this.name = name;  
        this.vorname = vorname;  
    }  
  
    public void sayName(){  
        System.out.println("Hi! My name is  
                            "+this.name);  
    }  
  
}
```



Erweitern

```
//Unterklasse:  
public class Student extends Person {  
  
    private int matrikelnr;  
  
    public Student(String name, String vorname,  
                   int matrikelnr){  
        super(name,vorname);  
        this.matrikelnr = matrikelnr;  
    }  
  
    @Override  
    public void sayName() {  
        super.sayName();  
        System.out.println("Ich bin ein Student!");  
    }  
  
}
```

Beispiel zum Überschreiben einer Methode aus der Oberklasse

- Um diese komplett zu **verändern**
 - Kein Aufruf der Oberklassenmethode mittels **super**

```
//Oberklasse:  
public class Person {  
  
    private String name;  
    private String vorname;  
  
    public Person(String name, String vorname) {  
        this.name = name;  
        this.vorname = vorname;  
    }  
  
    public void sayName(){  
        System.out.println("Hi! My name is  
            "+this.name);  
    }  
  
}
```

Verändern

```
//Unterklasse:  
public class Student extends Person {  
  
    private int matrikelnr;  
  
    public Student(String name, String vorname,  
        int matrikelnr){  
        super(name,vorname);  
        this.matrikelnr = matrikelnr;  
    }  
  
    @Override  
    public void sayName() {  
        System.out.println("Ich bin der  
            Student "+this.getName());  
    }  
  
}
```

Das Überschreiben einer Methoden kann verhindert werden, wenn die entspr. Methoden in der Oberklasse mit **final** deklariert wird!

Dynamisches Binden:

Stehen prinzipiell mehrere Implementierungen der gleichen Methode zur Verfügung (bspw. durch überschriebene, vererbte Methoden), kann erst zur Laufzeit entschieden werden, welche Implementierung beim Aufruf der Methode verwendet wird.

- Es wird die Implementierung in der tatsächlichen Klasse bevorzugt!
 - Falls keine Implementierung vorhanden, wird die Implementierung der kleinsten Oberklasse verwendet.

Die dynamische Bindung ist ein wichtiger Teilbereich des **Polymorphismus**

- Polymorphie (gr.: Vielgestaltigkeit) ist ein Schlüsselprinzip der OO-Programmierung
 - Bezeichner können je nach Verwendung unterschiedliche Datentypen annehmen
- Andere Bereiche von Polymorphismus:
 - Überladung
 - Generische Typen
 - Subtyping
 - ...

```
import java.util.ArrayList;
```

```
class Figur {
    void steckbrief() {
        System.out.println("Ich bin eine Figur.");
    }
}
```

Klasse Figur implementiert
void steckbrief()

```
class Kreis extends Figur {
    @Override
    void steckbrief() {
        System.out.println("Ich bin ein Kreis.");
    }
}
```

Klasse Kreis erbt von Figur
und überschreibt
void steckbrief()

```
class Dreieck extends Figur {
    @Override
    void steckbrief() {
        System.out.println("Ich bin ein Dreieck.");
    }
}
```

Klasse Dreieck erbt von Figur
und überschreibt
void steckbrief()

```
public class DynamicBinding {
    public static void main(String[] args) {
        ArrayList<Figur> figuren = new java.util.ArrayList<Figur>();
        figuren.add(new Figur());
        figuren.add(new Kreis());
        figuren.add(new Dreieck());

        for (Figur f : figuren) {
            f.steckbrief();
        }
    }
}
```

Die ArrayList hält
nur Objekte vom
Typ Figur

Kreis u. Dreieck sind auch
Figuren und werden hinzugefügt

Ausgabe:
Ich bin eine Figur.
Ich bin ein Kreis.
Ich bin ein Dreieck.

Erstellen Sie ein neues Eclipse-Projekt *Uebung12* und schreiben Sie ein Programm zum Bestellen von Pizza. Erstellen Sie dazu die folgenden Klassen:

Eine Klasse Pizza

- mit den Attributen Name (`String`) und Preis (`double`)
- Dazu die passenden Getter und Setter
- Einen Konstruktor, mit dem der Name und der Preis festgelegt werden kann

Ein Klasse Spezial (erbt von Pizza)

- Besitzt noch die Attribute Saucentyp (`String`) und Schaerfegrad (`int`)
- Dazu die passenden Getter und Setter
- Ist immer um 10% teurer als die eigentliche Pizza, daher muss die getter-Methode für den Preis diese Teuerung gleich berücksichtigen und immer 110 % des Preises zurückliefern

Eine Klasse Sonderangebot (erbt von Pizza)

- Besitzt das Attribut Rabatt (`double`)
- Muss im Konstruktor mit angegeben werden
- Dazu die passenden Getter und Setter
- Die Getter-Methode für den Preis berücksichtigt den Rabatt gleich und gibt den um den Rabatt reduzierten Preis zurück.

Testen Sie Ihr Programm, indem Sie in der Main-Methode verschiedene Pizzen anlegen und sich dann den Preis der einzelnen Pizzen ausgeben lassen

Motivation:

- Abstrakte Klassen legen gemeinsame Attribute und Methoden fest, über die alle abgeleiteten Klassen verfügen **müssen**.

Eigenschaften:

- Abstrakte Klassen beginnen mit dem Schlüsselwort `abstract`
 - `public abstract class MeineAbstrakteKlasse {...}`
- Von abstrakten Klassen können keine Objekte erzeugt werden
 - Keine `new` Anweisung!
- Aber es können eigene Klassen abgeleitet werden
 - `public class EigeneKlasse extends MeineAbstrakteKlasse{...}`
- Abstrakte Klassen können enthalten:
 - Instanzvariablen
 - Konstruktoren
 - Konkrete Methoden
 - Abstrakte Methoden, die von der ableitenden Klasse **implementiert werden müssen!**

```
//Abstrakte Klasse
public abstract class Tier {

    //Instanzvariablen
    private boolean kannFliegen;
    private boolean kannSchwimmen;

    //Konstruktor
    public Tier(boolean kannFliegen, boolean kannSchwimmen){
        this.kannFliegen=kannFliegen;
        this.kannSchwimmen=kannSchwimmen;
    }

    //Konkrete Methode
    public void doAction(){
        if (kannFliegen) System.out.println("Tier fliegt");
        if (kannSchwimmen) System.out.println("Tier schwimmt");
    }

    //Abstrakte Methoden
    public abstract void macheLaut();
    public abstract String getLieblingsEssen();
}
```

Abstrakte Klasse: Tier

Nur abstrakte Klassen **können** abstrakte Methoden haben!

Diese müssen von den abgeleiteten Klassen implementiert werden!

```
public class Vogel extends Tier {  
  
    public Vogel(){  
        super(true,false);  
    }  
  
    @Override  
    public void macheLaut() {  
        System.out.println("Piep piep");  
    }  
  
    @Override  
    public String getLieblingsEssen() {  
        return "Würmer";  
    }  
}
```

```
public class Fisch extends Tier {  
  
    public Fisch(){  
        super(false,true);  
    }  
  
    @Override  
    public void macheLaut() {  
        System.out.println("Blubb blubb");  
    }  
  
    @Override  
    public String getLieblingsEssen() {  
        return "Algen";  
    }  
}
```

Zusammenfassung:

- Nur **abstrakte** Klassen können **abstrakte** Methoden haben, müssen aber nicht
- Abstrakte Methoden beschreiben nur das **WAS** geleistet werden muss (Methodenkopf), nicht das **WIE** es geleistet wird.
- Das **WIE** (Implementierung) müssen die abgeleiteten (konkreten) Klassen erledigen
- Abstrakte Klassen werden inzwischen meist durch *Interfaces* abgelöst (nächste Stunde)

Erstellen Sie in Ihrem Projekt Uebung12 folgende Klassen:

- Eine Abstrakte Klasse `Geometrische2dFigur`
 - Hat als Attribut einen Namen (`String`) und bekommt diesen im Konstruktor übergeben
 - Besitzt Getter und Setter für den Namen
 - Hat eine abstrakte Methode `double berechneFlaeche()`;
 - Hat eine abstrakte Methode `double berechnenUmfang()`;
- Eine konkrete Klasse `Kreis`, welche von `Geometrische2dFigur` abgeleitet ist
 - Hat als Attribut einen radius (`double`)
 - Bekommt diesen zusammen mit einem Namen im Konstruktor übergeben
 - Implementiert die beiden abstrakten Methoden `berechneFlaeche` und `berechnenUmfang` gemäß eines Kreises:
 - $\text{Umfang} = 2 * \pi * \text{radius}$
 - $\text{Flaeche} = \text{radius} * \text{radius} * \pi$
- Eine konkrete Klasse `Quadrat`, welche von `Geometrische2dFigur` abgeleitet ist
 - Hat als Attribut eine Seitenlänge (`double`)
 - Bekommt diesen zusammen mit einem Namen im Konstruktor übergeben
 - Implementiert die beiden abstrakten Methoden `berechneFlaeche` und `berechnenUmfang` gemäß eines Quadrats:
 - $\text{Umfang} = 4 * \text{seitenlänge}$
 - $\text{Flaeche} = \text{seitenlänge} * \text{seitenlänge}$

- Eine konkrete Klasse **Rechteck**, welche von **Geometrische2dFigur** abgeleitet ist
 - Hat als Attribut eine Seitenlänge a (**double**) und eine Seitenlänge b (**double**)
 - Bekommt diesen zusammen mit einem Namen im Konstruktor übergeben
 - Implementiert die beiden abstrakten Methoden **berechneFlaeche** und **berechnenUmfang** gemäß eines Rechtecks:
 - $\text{Umfang} = 2 * a + 2 * b$
 - $\text{Flaeche} = a * b$;
- Schreiben Sie eine Anwendung, welche die 3 verschiedenen Figuren (Kreis, Quadrat und Rechteck) mit verschiedenen Parametern erzeugt und lassen Sie sich jeweils die Fläche und den Umfang berechnen und ausgeben.