Einstieg in die Programmierung von Computern – Teil V

Im Teil 5 behandeln wir nun „abstrakte Klassen“ und „Schnittstellen (Interfaces)“, mit denen die objektorientierte Java-Programmierung richtig interessant und spannend wird. Erst Schnittstellen ermöglichen es, fundamentale Designmuster in der Programmierung einzusetzen.

Abstrakte Klassen

Im letzten Teil haben wir die Vererbung als elementare Eigenschaft von Klassen besprochen. Das Schlüsselwort **„extends“** hinter dem Klassennamen sagt aus, dass diese Klasse von einer anderen Klasse (Klassenname nach dem Schlüsselwort) sichtbare Eigenschaften erbt.

```
public class KindKlasse extends VaterKlasse {
  ...
}
```

Wir konnten nun von der Beispielklasse „Kindklasse“ oder auch von der „Vaterklasse“ Objektinstanzen mit der Anweisung

```
KindKlasse kind = new Kindklasse();
VaterKlasse vater = new Vaterklasse();
```

erzeugen. Die Vaterklasse ist hier eine sogenannte „konkrete“ Klasse, von der Objektinstanzen erstellt werden können. Es gibt jedoch Fälle, in denen man mit einer Klasse lediglich ausdrücken möchte, dass diese bestimmte Eigenschaften besitzt, die sie an andere Klassen vererben kann, ohne dass jedoch von dieser Klasse selbst Objektinstanzen erstellt werden können. Diese Art von Klassen, von denen keine Objektinstanzen erstellt werden können, nennt man **„abstrakte“** Klassen und werden mit dem Schlüsselwort **„abstract“** gekennzeichnet. Demnach sind diese das Gegenteil von „konkreten“ Klassen, von denen Objektinstanzen erstellt werden können. Die abstrakten Klassen dienen lediglich als Modellierungs-Klasse in der Vererbungshierarchie.


```
public abstract class AbstrakteKlasse {
  ...
}
```

Schauen wir uns dazu ein kleines Beispiel an. Eine Klasse „Fahrzeug“ soll als abstrakte Klasse dienen, die den konkreten Klassen wie z. B. „Auto“, „Motorrad“, „Fahrrad“ Fahrzeugeigenschaften vererben kann, d. h. es können nur von den konkreten Klassen Objekte erzeugt werden, was in der Tat auch Sinn macht. Eine Objektinstanz von „Fahrzeug“ wäre unsinnig. Was sollte man mit diesem Objekt anfangen? Ein Auto-Objekt ist konkret und man kann mit diesem Objekt auch konkret etwas anfangen. Eine Fahrzeug-Klasse besitzt dabei die Vorgaben für eine Subklasse. Die Subklassen Auto, Motorrad und Fahrrad erben die Methoden der Fahrzeug-Klasse bzw. müssen diese implementieren. Ein Exemplar der Fahrzeug-Klasse selbst muss nicht existieren. Den dazugehörigen Quellcode sehen Sie im Kasten 1.

Wie man sehen kann, wird im Beispiel aus Kasten 1 genau eine Methode in der abstrakten Klasse nicht implementiert. Die Implementierung wird somit der konkreten Klasse überlassen. Dieses wird durch das Schlüsselwort „abstrakt“ vor dem Methodennamen ausgedrückt. Denn: Woher soll die Fahrzeug-Klasse wissen, wie der Zusammenbau eines Autos funktioniert? Das kann nur die konkrete Auto-Klasse selbst wissen, somit müssen alle abstrakten Methoden in der konkreten Klasse implementiert werden. Die Subklasse „Auto“ implementiert die abstrakten Methoden der abstrakten Klasse Fahrzeug und nimmt das Abstrakte damit weg.

Achtung: Implementiert eine Subklasse nicht alle geerbten abstrakten Methoden, so muss diese Klasse selbst wieder abstrakt sein. Ist mindestens eine Methode abstrakt, so ist die ganze Klasse automatisch abstrakt.

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

  private String antriebsart;
  private int anzahlPlaetze;

  // Keine abstrakte Methode
  public void setze_Antriebsart(String antriebsart) {
      this.antriebsart = antriebsart;
  }

  // Keine abstrakte Methode
  public String hole_Antriebsart() {
      return this.antriebsart;
  }

  // Abstrakte Methode
  // (muss in Subklasse überrschrieben werden)
  public abstract void zusammenbauen();
}

// Abstrakte Klasse Fahrzeug
public class Auto extends Fahrzeug {

  private String antriebsart;

  // Die abstrakte Methode der abstrakten Klasse
  // muss hier implementiert werden.
  public void zusammenbauen() {
      // hier stehen die Anweisungen
      // der Methode.
  }
}
```

Kasten 1: Abstrakte Klasse.

Schnittstellen (Interfaces)

In Java kann eine Klasse maximal nur von einer Eltern-Klasse erben. Man könnte zu der Meinung gelangen, dass das Erben von nur einer Eltern-Klasse zu einer großen Einschränkung führt. Das ist jedoch nicht der Fall. Wir werden gleich sehen, dass eine Mehrfachvererbung doch indirekt mittels Schnittstellen möglich ist. Schnittstellen sind eine Abart von abstrakten Klassen und stellen ein fundamentales Programmiervehikel in Java zur Verfügung. In einem Interface werden alle Methoden mit Signatur nur aufgeführt, aber nicht implementiert. Die Implementierung wird den Klassen überlassen, die das Interface implementieren. Ein Interface enthält neben diversen Datenelementen lediglich die abstrakten Methoden. Nun kommt das Entscheidende: Eine Klasse kann mehrere Interfaces implementieren, damit ist dann doch eine Art Mehrfachvererbung möglich. In Java wird ein Interface mit dem Schlüsselwort **„interface“** gekennzeichnet.

```
public interface Schnittstelle1 {

  public void methode1();
}

public interface Schnittstelle2 {

  public void methode2();
}
```

Eine Klasse, die nun das oder mehrere Schnittstellen implementiert, wird so deklariert:

```
public class Beispielklasse implements Schnittstelle1, Schnittstelle2, ..., SchnittstelleN {
```

```
// Datei: Gebaeude.java
public abstract class Gebaeude {

  private int stockwerke;

  public void setzeAnzahlStockwerke(int stockwerke) {
      this.stockwerke = stockwerke;
  }

  public int anzahlStockwerke() {
      return this.stockwerke;
  }
}

// Datei: Heizung.java
public interface Heizung {

      public String artHeizung();
}

// Datei: Energiesparhaus.java
public class Energiesparhaus extends Gebaeude implements Heizung {

  private static final String SOLAR_HEIZUNG = "Solar-Heizung";

  public String artHeizung() {
      return this.SOLAR_HEIZUNG;
  }
}

// Datei: Hochhaus.java
public class Hochhaus extends Gebaeude implements Heizung {

  private static final String FERNWAERME_HEIZUNG = "Fernwaerme-Heizung";

  public String artHeizung() {
      return this.FERNWAERME_HEIZUNG;
  }
}

// Datei: Main.java zum testen des Beispiels
public class Main {

  public static void main(String[] args) {

      Hochhaus deutscheBank = new Hochhaus();
      deutscheBank.setzeAnzahlStockwerke(25);
      Energiesparhaus passivhaus = new Energiesparhaus();
      passivhaus.setzeAnzahlStockwerke(2);

      System.out.println("Die deutsche Bank hat " +
              deutscheBank.anzahlStockwerke() + " Stockwerke.");
      System.out.println("Die deutsche Bank hat eine " +
              deutscheBank.artHeizung());

      System.out.println("Das Passivhaus hat " + passivhaus.anzahlStockwerke()
              + "S tockwerke.");
      System.out.println("Das Passivhaus hat eine " + passivhaus.artHeizung());
  }
}
```

Kasten 2: Beispiel mit abstrakter Klasse und Interface.

Eine Zuweisung ist dann erlaubt, wenn die zugewiesene Instanz das Interface implementiert

```
public class Klasse1 {
  ...
}

public interface Interface1 {
  ...
}

public class Klasse2 implements Interface1 {
  ...
}

public class IrgendeineKlasse {

Interface1 a;
  // Die folgende Zuweisung ist erlaubt.
  a = new Klasse2();
```

```
public void methode1() {
  // Anweisungen der Methode
}

public void methode1() {
  // Anweisungen der Methode
}

// weitere Methoden der Klasse
  Subinterfaces Beispielklasse
}
```

Achtung: Eine Klasse, die eine oder mehrere Schnittstellen implementiert, muss alle abstrakten Methoden der Schnittstelle(n) implementieren, sonst gibt es ein Compilerfehler. Nach der Implementierung eines Interfaces kann eine Klasse wie eine Subklasse des Types Interface behandelt werden. Das bedeutet, dass auch Variablen den Typ eines Interfaces besitzen können.

```
    // Die folgende Zuweisung ist
       nicht erlaubt.
    a = new Klasse1();

    ...
}
```

Subschnittstellen (Subinterfaces)

Eine **Subschnittstelle** bzw. **Subinterface** ist die Erweiterung eines anderen Interfaces, d. h. ein Interface kann wiederum von einem anderen Interface bzw. mehreren Interfaces erben. Die Erweiterung wird wie bei der Vererbung durch das Schlüsselwort **„extends“** gekennzeichnet.

```
public interface Interface1 extends Interface2 {

  public void methodenName();

  ...
}
```

Eine Klasse, die nun das Interface „Interface1“ implementiert, muss die Methoden von beiden Schnittstellen (Interface1 und Interface2) implementieren.
Beispiel für eine Schnittstelle, die mehrere Schnittstellen erweitert:

```
public interface Interface1 extends Interface2, Interface3 {
  public void methodenName();

  ...
}
```

Statische Variablen und Methoden

Klassenvariablen sind eng mit ihrem Objekt verbunden. Wird ein Objekt erzeugt, erhält es einen eigenen Satz von Klassenvariablen, die zusammen den Zustand des Objekts widerspiegeln. Ändert eine Objektmethode den Wert einer Klassenvariablen in einem Objekt, so hat dies keine Auswirkungen auf andere Objekte.
Nicht immer ist diese Verhalten erwünscht. Man möchte zudem Attribute und Methoden unabhängig von einem Objektzustand verwenden. Diese genannten Eigenschaften sind keinem konkreten Objekt zugeordnet, sondern lediglich der Klasse. Diese Art von Zugehörigkeit wird in Java durch **„statische Eigenschaften“** unterstützt. Da sie zu keinem Objekt gehören (wie Objekteigenschaften), werden diese auch **„Klasseneigenschaften“** genannt.
In Java werden statische Eigenschaften mit dem Schlüsselwort **„static“** versehen. Für den Zugriff werden dann statt der Referenzvariablen einfach Klassennamen verwendet.

Beispiel:

```
public class Namen {

  public static final int MAX_
  LAENGE_NAMEN = 14;

  public static String holeNa-
  men() {
    return "Thomas Burgard";
  }
}
```

Die Variable „MAX_LAENGE_NAMEN“ ist mit dem Modifizierer **„final“** gekennzeichnet, da „MAX_LAENGE_NAMEN“ eine Konstante ist, deren Wert später nicht mehr verändert werden soll. Die Verwendung der statischen Eigenschaften der Klasse Namen ist folgendermaßen:

```
public class StaticVerwendung {
  public static void main(String[]
  args) {

    int maxLaenge = Namen.
    MAX_LAENGE_NAMEN;
    System.out.println("Max
    Länge eines Namens ist: " +
    maxLaenge);

    System.out.println("Mein
    Name ist: " + Namen.holeNa-
    men );
}
```

Im Beispiel ist gut zu sehen, dass die Konstante „MAX_LAENGE_NAMEN“ direkt mit „Namen. MAX_LAENGE_NAMEN “ verwendet werden kann. Man braucht vorher keine Objektinstanz mit **„Namen namen = new Namen()“** erstellen.
Es gilt: Statische Attribute und als statisch deklarierte Methoden innerhalb von Klassen können ohne Objektinstanz verwendet werden.

Vererbte Konstanten bei Schnittstellen

Schnittstellen können Variablen besitzen, die jedoch, wie wir gesehen haben, immer automatisch statisch und final, also Konstanten sind. Diese Konstanten können einer anderen Schnittstelle vererbt werden.
Es gilt:

- Interfaces vererben ihre Eigenschaften an die Subinterfaces.
- Ein Interface kann von mehreren Interfaces erben, die wiederum jeweils ein bestimmtes Attribut von einem anderen gemeinsamen Interface beziehen.
- Konstanten dürfen von Subinterfaces wieder überschrieben werden.
- Interfaces können von mehreren Interfaces die Attribute gleichen Namens übernehmen, auch wenn sie den gleichen Wert haben. Für die Verwendung muss dann ein qualifizierter Name verwendet werden, der angibt, welches Attribut gemeint ist.

Zum Schluss wollen wir uns noch ein Beispiel im Kasten 2 ansehen, in dem ein abstrakte Klasse und ein Interface eingesetzt werden. Insgesamt werden fünf Dateien verwendet.

Fazit

Abstrakte Klassen und Schnittstellen (Interfaces) geben der Java-Programmierung elegante Lösungsmöglichkeiten, die viele Probleme leicht lösen lassen. Wie wir im nächsten Teil sehen werden, können damit sogenannte „Design-Muster“ erstellt werden, die in der Objekt orientierten Programmierwelt sehr entscheidend sind und elegante Lösungen zu immer wieder kehrenden Problemen bieten.

Ausblick zum Teil 6 der Serie

Wir lernen abstrakte Klassen und Interfaces noch genauer kennen und werden damit ein paar „Design-Muster“ erstellen, die die Zusammenhänge dann richtig erklären. Außerdem lernen wir die sogenannten „Pakete“ kennen, mit denen der Sourcecode einer Applikation optimal strukturiert. ZT

ZT Adresse

Thomas Burgard Dipl.-Ing. (FH)
Softwareentwicklung & Webdesign
Bavariastraße 18b
80336 München
Tel.: 089 540707-10
Fax: 089 540707-11
info@burgardsoft.de
www.burgardsoft.de