Einstieg in die Programmierung von Computern – Teil IV

Im Teil 4 der mehrteiligen Serie „Einstieg in die Programmierung von Computern“ beschäftigen wir uns mit „bedingten Anweisungen“, Schleifen, dem etwas komplexeren Datentyp „Array“ und lernen die Feinheiten der objektorientierten Programmierung in Java im Detail kennen.

Bedingte Anweisungen

Das Wort „bedingt“ in der Überschrift sagt bereits aus, dass eine Anweisung oder Anweisungen unter ganz bestimmten „Bedingungen“ ausgeführt werden. In der Informatik spricht man in diesem Fall auch von sogenannten „Kontrollstrukturen“, da der weitere Ablauf der Software durch gegebene Bedingungen gezielt kontrolliert werden kann. Die einfachste Kontrollstruktur ist die **if-Anweisung**.

Beispiele:

```
if (zahl == 0) {
  String str = "Die Zahl hat den
  Wert 0";
  System.out.println(str);
}
```

Man beachte: Wenn mehr als eine Anweisung im Kontrollblock steht, müssen die Anweisungen zwischen einer geöffneten und einer geschlossenen Klammer stehen. Bei nur einer Anweisung kann die Klammer auch entfallen (muss aber nicht).

Beispiel mit nur einer Anweisung:

```
if (zahl == 10)
   System.out.println ("Die Zahl
   hat den Wert 10");
...
```

Man kann die Bedingungen auch zusammensetzen, wie folgende Beispiele zeigen:

```
if (zahl == 10 || zahl > 20) // wenn 10
oder größer als 20
   System.out.println ("Die Zahl
   hat den Wert 10 oder ist größer
   als 20");

if (zahl1 == 10 && zahl2 >= 20) //
wenn zahl1 10 und zahl2 größer/
gleich 20
   System.out.println ("Die Zahl1
   hat den Wert 10 und die zahl2 ist
   größer/gleich 20");
```

Die bedingte Anweisung läßt sich auch mit **„else if“** bzw. **„else“** verzweigen, wie folgende Beispiele demonstrieren:

```
if (zahl >= 0) {
   String str = "Die Zahl ist größer
   als 0";
   System.out.println(str);
} else if (zahl <= 10) {
   System.out.println ("Die Zahl
   ist kleiner/gleich 10");
} else {// alle anderen Werte werden
hier behandelt!
   System.out.println ("Die Zahl
   ist");
}
```

Schleifen

Mit „Schleifen“ können Anweisungen mehrfach abgearbeitet werden. Hierzu benötigt die Schleife eine „Schleifenbedingung“ und den „Schleifenrumpf“, gegebenenfalls auch eine **break**-Anweisung (Abbruch) oder **continue**-Anweisung (weitermachen). Java

besitzt insgesamt vier Schleifentypen:

for-Schleife

Die **for-**Schleife dient zum Zählen und die Anweisungen innerhalb der Schleife werden nur ausgeführt, wenn die Bedingung stimmt.

```
for (int index = 0; index <= 10;
index++) {
   System.out.println(index);
}
```

Der Startwert von index ist 0 (Initialisierung der for-Schleife). Nun folgt die Schleifenbedingung, also „index <= 10“. Wenn index den Wert 10 durch das automatische Hochzählen erreicht hat, wird die Schleife beendet, d.h. die Anweisungen innerhalb der Schleife werden nicht mehr ausgeführt. Mit index++ wird der Schleifenzähler (index) um eins hochgezählt (geschieht immer als letzte Anweisung des Schleifenrumpfs, bevor die erste Anweisung im Rumpf wieder ausgeführt wird).

Eine Besonderheit: Die Endlosschleife. Eine Endlosschleife kann man ganz einfach mit

```
for (;;)
   System.out.println("Ich laufe
   endlos!");
```

erzeugen.

Erweiterte for-Schleife

Die erweiterte **for-**Schleife wird für das Iterieren von „Arrays“ und „Collections“ verwendet und vereinfacht die Schreibweise der for-Schleife.

```
int[] zahl = {1, 3, 5, 7, 9};

for (int n: zahl)
   System.out.println(n);
}
```

while-Schleife

Bei der **while-**Schleife werden die Anweisungen innerhalb der Schleife immer nur dann ausgeführt, wenn die Bedingung(en) stimmen.

```
int index = 0;
while (index <= 10) {
  System.out.println(index);
  index++; // identisch mit index =
index+1;

}
```

do-while-Schleife

Bei der **do-while-**Schleife wird die Bedingung erst am Ende der Schleife geprüft, die Anweisungen innerhalb der Schleife werden also mindestens einmal durchlaufen.

```
int index = 0;
do {
  System.out.println(index);
  index++;
} while (index <= 10);
```

break und continue

Mit **break** und **continue** können die Schleifen gezielt beendet bzw. weiter durchlaufen werden.

```
for (int index = 0; index <= 10;
index++) {
  System.out.println(index);

  if (index == 5)
   break; // die Schleife wird ver-
   lassen!

  index++;
}
int index = 0;
while (index <= 10) {
  if (index > 5) {
    index++;
    System.out.println(index);
    continue; // springe wieder zum
    Schleifenbeginn!
  }

  System.out.println(index);
  index++;
}
```

Arrays

Der Datentyp Array kann mehrere Werte (einfache Datentypen wie z.B. int) und Referenztypen (Objekte) speichern. Man kann sich einen Array als einen Behälter mit Schubladen vorstellen. Jede Schublade bekommt eine Nummer und kann den Wert bzw. Referenz speichern. Die erste Schublade bekommt die Nummer 0, die zweite 1 usw. Ein Array wird in Java folgenderweise deklariert:

```
int[] ungeradeZahlen;
```

Die Variable „ungeradeZahlen“ kann somit int-Werte aufnehmen. Mit der Deklaration ist aber noch kein Array erzeugt. Das geschieht entweder mit dem new-Operator, er drückt auch aus, das ein Array dynamisch angelegt wird, also genauso wie Objekte

```
int[] ungeradezahlen;
ungeradezahlen = new int[4];
```

oder

```
int[] ungeradezahlen = new int[4];
```

oder

```
int i = 4;
int[] ungeradezahlen = new int[i];
```

oder abgekürzt mit sofortiger Werte-Initialisierung `int[] ungeradezahlen = {1, 3, 5, 7, 9};`

Mit dem Attribut „length“ kann die Länge eines Arrays sehr einfach ermittelt werden:

```
int[] ungeradezahlen = {1, 3, 5, 7, 9,
11, 13, 15, 17, 19};
int länge = ungeradezahlen.length;
```

In einem kleinen Programm (siehe Kasten 1) wollen wir nun das Gelernte anwenden. Die Anweisung „if (idx%2)“ prüft, ob die Zahl eine gerade oder ungerade Zahl ist. „%“ ist in Java der sogenannte **Modulo-**Operator und teilt in unserem Beispiel den Wert der Variable „idx“ durch den Wert 2 und gibt den Rest zurück. So kann recht einfach ermittelt werden, ob eine Zahl ungerade ist oder nicht.

Arrays aus Referenztypen

Mit Arrays ist es selbstverständlich auch möglich, Referenztypen (Objekte) zu speichern.

Beispiel:

```
Auto[] autoArray = new Auto[4];
```

Im Array können nun insgesamt vier Auto-Objekte gespeichert werden.

Mehrdimensionale Arrays

Mehrdimensionale Arrays sind Arrays in Arrays und werden folgendermaßen angelegt:

Beispiel:

```
Auto[][] autoArray = new
Auto[4][3]; oder Auto[][][]auto
Array = new Auto[4][3][2];
```

Objektorientiertes Programmieren in Java

Beziehung zwischen Objekten

Die einfachste Art der Kommunikation zwischen Objekten in Java ist die sogenannte „Assoziation“ (auf deutsch: Beziehung). Wir haben ja in den letzten Teilen unseres Seminars gelernt, dass Objekte nichts anderes als Klasseninstanzen sind, d.h. Referenzvariablen einer geschriebenen Klasse. Durch die Assoziation kann ein Objekt durch einen Methodenaufruf des anderen Objektes sehr einfach eine Kommunikation stattfinden. Kommunikation bedeutet in der Tat einfach Methodenaufruf eines anderen Objektes.

Objekt A ↔ Objekt B

Die unidirektionale Beziehung

Bei der einfachen unidirektionalen Beziehung hat nur eine Klasse ein Verweis auf eine andere Klasse.

```
Class SpielerA {

  ....
}
Class SpielerB {
  SpielerA spielerA; //Verweis auf
  die Klasse „SpielerB“
}
```

Die bidirektionale Beziehung

Bei dieser Beziehungsart wird in beiden Klassen ein Verweis auf die jeweils andere Klasse erzeugt.

```
Class SpielerA {
  SpielerB spielerB; //Verweis auf
  die Klasse „SpielerA“
```

```
// Datei: JavaSeminarTeil4.java
public class JavaSeminarTeil4 {

  /*
   * Methode zum Starten eines Java-Programms.
   *
   */
  public static void main(String[] args) {

    JavaSeminarTeil4 javaSeminarTeil4 = new JavaSeminarTeil4();
    String[] ungeradeZahlen; // Array wird deklariert
    ungeradeZahlen = new String[21]; // Erzeuge den deklarierten Array

    for (int idx=1; idx < 21; idx++) {
      if (idx%2 != 0) { // Mit Modulo wird auf ungerade Zahl geprüft!
        ungeradeZahlen[idx] = "ungerade";
      } else {
        ungeradeZahlen[idx] = "gerade";
      }
    }

    javaSeminarTeil4.showUngeradeZahlen(ungeradeZahlen);
  }

  /*
   * Methode zum Anzeigen der geraden und ungeraden Zahlen bis 20.
   * Bekommt den Array als Übergabeparameter mit.
   *
   */
  private void showUngeradeZahlen(String[] ungeradeZahlen) {

    int idx = 1;

    while (idx < ungeradeZahlen.length) {
      System.out.println("Die Zahl " + idx + " ist " + ungeradeZahlen[idx]);
      idx++;
    }
  }
}
```

Kasten 1: Schleifen und Arrays.

```
// Datei: Fahrzeug.java
public class Fahrzeug {

  private int anzahlKilometer;
  priavte int anzahlReifen;

  protected void setzeKilometerstand(int kilometer) {
    this.anzahlKilometer = kilometer;
  }

  protected int setzeAnzahlReifen(int reifen) {
    this.anzahlReifen = reifen;
  }

  protected int wievieleKilometer() {
    return this.anzahlKilometer;
  }

  protected int wievieleReifen() {
    return this.anzahlReifen;
  }
}

// Datei: Auto.java
public class Auto extends Fahrzeug { }

// Datei: Motorrad.java
public class Motorrad extends Fahrzeug {

  private String motorradHaendler;

  public void motorradHaendler(String haendler) {
    this.motorradHaendler = haendler;
  }

  public String welcherMotorradHaendler() {
    return this.motorradHaendler;
  }
}

// Datei: TestVererbung.java
public class TestVererbung {

  public static void main(String[] args) {
    Auto porsche = new Auto();
    Motorrad honda = new Motorrad();

    porsche.setzeKilometerstand(500);
    porsche.setzeAnzahlReifen(4);
    honda.setzeKilometerstand(250);
    honda.setzeAnzahlReifen(2);
    honda.motorradHaendler("Bike Muenchen");

    System.out.println(porsche.wievieleKilometer());
    System.out.println(porsche.wievieleReifen());
    System.out.println(honda.wievieleKilometer());
    System.out.println(honda .wievieleReifen());
    System.out.println(honda.welcherMotorradHaendler());
  }
}
```

Kasten 2: Klassenvererbung.

```
}
Class SpielerB {
  SpielerA spielerA; // Verweis auf die Klasse „SpielerB“
}
```

Die Kommunikationsverbindung beider Klassen bzw. beider Objekte erfolgt dann ganz einfach mit:

```
SpielerA spielerA = new SpielerA(); // Erzeugen der Objektinstanz
SpielerB spielerB = new SpielerB(); // Erzeugen der Objektinstanz

spielerA.spielerB = spielerB;
spielerB.spielerA = spielerA;
```

Achtung: Bei der bidirektionalen Beziehung müssen natürlich gültige Referenzen vorhanden sein, d.h. beide Objekte müssen noch am Leben sein und nicht etwa den Wert „null“ durch die Anweisung z. B. **spielerA.spielerB = null;** besitzen.

Klassen und Vererbung

Eine wichtige und elementare Eigenschaft der Objektorientierung ist die Fähigkeit der Vererbung von Klassen. Das heißt, eine Unterklasse (Subklasse) erbt Eigenschaften (sichtbare Variablen und Methoden) einer Elternklasse (Superklasse), genauso wie im realen Leben, wo Kinder Eigenschaften und Verhalten von Eltern erben. Die Vererbung in Java wird durch das Schlüsselwort „extends“ beschrieben:

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

Durch die Vererbung werden alle sichtbaren Eigenschaften der Superklasse auf die Subklasse übertragen. Zum Beispiel kann die Subklasse die Methoden der Superklasse verwenden.
Achtung: Wenn die Implementierung einer Methode der Superklasse geändert wird, so muss die Subklasse überprüfen, ob die geänderte Methode kein Problem verursacht.
Die Subklassen binden sich somit sehr stark an die Superklasse, was bei großen Softwaresystemen oft zu großen Problemen führt. Wie man diese Probleme eleganter lösen kann, zeigt das Kapitel „Schnittstellen“.
Wir wollen anhand eines einfachen Beispiels (siehe Kasten 2) die Klassenvererbung verdeutlichen.

Polymorphie (dynamische oder späte Bindung)

Ein weiteres elementares Feature der Objektorientierung in Java ist die Fähigkeit der dynamischen Bindung. Wir habe im vorigen Kapitel gelernt, dass die Subklasse alle sichtbaren Eigenschaften der Superklasse vererbt bekommt. Das heißt, die Subklasse kann die Methoden der Superklasse auch verwenden. Es gibt jedoch viele Situationen, in denen die Implementierung der geerbten Methode in der Subklasse nicht passt. In diesem Fall kann die Subklasse die Methode der Superklasse „überschreiben“ **(Overwriting)**. Hierbei muss die Methodensignatur und der Rückgabewert exakt übereinstimmen, die Implementierung der Methode jedoch ist unterschiedlich. Der zweite Fall ist das „Überladen“ **(Overloading)** von Methoden.
Hierbei stimmt nur der Name der Methode überein, die Methodensignatur und die Implementierung der Methode jedoch ist unterschiedlich.

Beispiel Overwriting:

```
class Fahrzeug {

  public void tanken() {
    String str = "Der Tank befindet sich an der Seite!";
  }
}
class Auto extends Fahrzeug {

  public void tanken() {
    String str = "Der Tank befindet sich oben!";  }
}
```

Beispiel Overloading:

```
class Fahrzeug {

  public void tanken(int anzahl Liter) {
    System.out.println(anzahl Liter);
  }
}
class Auto extends Fahrzeug {

  private String str;

  public void tanken(String str) {
    System.out.println(str);
  }
}
```

Fazit

In diesem Teil des Programmierseminars mit Java haben wir die wichtigen Kontrollstrukturen, Schleifen, Arrays und die Klassenvererbung kennengelernt. Damit haben wir ein gutes Fundament bzw. Rüstzeug für die Java-Programmierung erarbeitet. Wir sind in der Lage, kleine Programme zu schreiben, in denen die elementaren objektorientierten Mechanismen von Java berücksichtigt werden können. Für weitere detaillierte Beschreibungen der beschriebenen Java-Features bitte ich, die umfangreichen Oracle-Webseiten zum Thema Java (http://docs.oracle.com/javase/tutorial) anzuschauen und zu lesen.

Ausblick zum Teil 5 der Serie

Im nächsten Teil geht es um abstrakte Klassen und Schnittstellen (Interfaces), mit denen fundamentale Design-Muster programmiert werden können. Es wird also richtig spannend, bleiben Sie dran. Erst mit Schnittstellen wird die objektorientierte Programmierung so richtig interessant. 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