Java

Kontrollstrukturen

D. A. Waldvogel

4. Kontrollstrukturen

Kontrollflußanweisungen
Die Anweisungen innerhalb eines Programms werden sequentiell abgearbeitet. Diese Abarbeitungsreihenfolge (der Kontrollfluß) kann durch Kontrollflußanweisungen gezielt durch den Programmierer beeinflußt werden. Java verfügt über die aus anderen Programmiersprachen bekannten Kontrollflußanweisungen, und die Syntax ist stark an C bzw. C++ angelehnt.

4.1 Bedingte-Anweisung

Eine der häufigsten Problemstellungen in einem Programm ist, den Programmfluß in Abhängigkeit von einer oder mehreren Bedingungen zu steuern. Die *if*-Anweisung wird bei bedingten Kontrollflußverzweigungen verwendet. Die Syntax der *if*-Anweisung lautet:

```
if (<logischeBedingung>)
{
   <Anweisung-1>
}
[else
{
    <Anweisung-2>]
}
```

Die *logischeBedingung* wird bewertet. Ergibt sie wahr (true), werden die Anweisungen direkt hinter dem *if* ausgeführt. Ergibt die *logischeBedingung* falsch (false), werden die Anweisungen hinter dem *else* ausgeführt.
Man kann die *if*-Ánweisung auch ohne den *else*-Zweig schreiben. Ist in einem solchen Fall die *logischeBedingung* falsch, wird der geklammerte Anweisungsblock übersprungen, und das Programm wird hinter der geschlossenen Klammer fortgeführt.

Im Bedingungsteil der Regel darf nur ein Ausdruck vom Datentyp `boolean` stehen. Es ist zu beachten, daß *Anweisung-1* auch für einen Block von Anweisungen stehen kann. Sollen also mehrere Anweisungen innerhalb einer *if*-Anweisung ausgeführt werden, so sind diese durch geschweifte Klammern einzuschließen.
Beispiel für eine *if*-Anweisung:

```
int i;
if (i > 5)
    System.out.println("i ist größer als 5");
else
    System.out.println("i ist kleiner als oder gleich 5");
```

Vergleichsoperatoren
Im Ausdruck *i>5* wird ein Vergleich durchgeführt. Das Ergebnis eines solchen Vergleichs ist entweder wahr oder falsch. > ist ein Vergleichsoperator und legt fest, welche Art von Vergleich durchgeführt werden soll.
Java kennt folgende Vergleichsoperatoren:

Zeichen	**Bedeutung**
<	kleiner als
>	größer als
<=	kleiner oder gleich
>=	größer oder gleich
Instanceof	Typ-Vergleich
==	gleich
!=	ungleich

Logische Operatoren
Wenn Sie Ausdrücke zusammenstellen, müssen oft mehrere Vergleiche zusammengefaßt werden.

So untersucht der Ausdruck *(x==2 && y<3),* ob x gleich 2 ist und ob y kleiner als 3 ist. Erst wenn beide Bedingungen erfüllt sind, wird die Anweisung hinter dem if ausgeführt, ansonsten die Anweisung hinter dem *else.*
Java kennt folgende logische Operatoren

Operator	**Bedeutung**
&&	UND
||	ODER
!	NICHT

Es können beliebig viele Aussagen durch logische Operatoren verknüpft werden.

Das folgende Beispiel zeigt eine Variante der *if-else*-Anweisung. Hierbei wird mit einer Folge von *else if* eine Kette von Abfragen aufgebaut

```
int z=4;
if (z== 1)
{
        System.out.println("Z=1");
}
else if (z==2)
{
        System.out.println("Z=2");
}
else if (z==3)
{
        System.out.println("Z=3");
}
else
{
        System.out.println("Z ist <1 oder >3");
}
```

Statt dieser geschachtelten *if*-Anweisungen verwendet man auch häufig die switch-Anweisung.

Switch-Anweisung
Sollen Mehrfachentscheidungen getroffen werden, so kann die switch-Anweisung verwendet werden. Hier wird untersucht, ob ein Ausdruck einer von mehreren Konstanten gleicht. Gleicht der Ausdruck keiner der angegebenen Konstanten, kann ein sonst-Zweig vorgesehen werden. Das Schlüsselwort `case` steht vor jeder Konstanten, die mit dem Ausdruck verglichen werden sollen, das Schlüsselwort `default` leitet den sonst-Zweig ein. Als Beispiel betrachten wir eine switch-Anweisung, die testet, welches Zeichen ein Benutzer eingegeben hat:

Beispiel
Die formale Syntax lautet:

```
switch (<Ausdruck>)      {
    case <konstanterAusdruck>:
        <Anweisung> [<Anweisung> ...]

    case <konstanterAusdruck>:
        <Anweisung> [<Anweisung> ...]
    ...
    default:
        <Anweisung> [<Anweisung> ...]   }
```

Zuerst wird der *Ausdruck* hinter dem Schlüsselwort *switch* ausgewertet. Dabei muß der Typ dieses Ausdrucks *char, byte, short* oder *int* sein, ansonsten meldet der Compiler einen Fehler. Der ermittelte

Wert wird dann mit jedem der konstanten Ausdrücke hinter den *case*-Anweisungen verglichen. Wird ein passender Ausdruck gefunden, wird die Programmausführung an dieser Stelle fortgeführt und läuft dann bis zum Ende der Switch-Anweisung oder bis zum nächsten break. Wenn kein Fall zutrifft wird die Anweisung hinter *default* ausgeführt.
Man beachte den Doppelpunkt hinter <*konstanter Ausdruck*>

Frage: Wenn ich folgendes Programm habe und ich rufe es mit dem Wert 9 auf, welches Ergebnis erscheint?

```
class switchtest
{
        public static void main(String args[])
        {
                int i=Integer.parseInt(args[0]);
                switch(i)
                {
                        case 0: case 2: case 4: case 2*3: case 8:
                                System.out.println("Gerade Zahl");
                                break;
                        case 1: case 3: case 5: case 7:
                        case 9:
                                System.out.println("Ungerade Zahl");
                                break;
                        default: system.out.println("Größer neun, test nicht durchgeführt");
                }
        }
}
```

Das Programm liest den Wert ein, der untersucht werden soll.
Mit dem Java-Befehl *integer.parseInt(args[0])* wird die Zahl 9 aus der Kommandozeile, die im Array args als Zeichenfolge vorhanden ist, in einen entsprechenden Integerwert umgewandelt und auf die Variable i gewiesen.
switch vergleicht die Variable i, die 9 enthält, mit den nun folgenden konstanten Ausdrücken, um zu ermitteln, welchen Wert i hat.
Ein konstanter Ausdruck ist auch 2*3.
Die *break*-Anweisung unterbricht die Programmausführung und springt hinter die *switch*-Anweisung. Dieses *break* ist bei den geraden Zahlen nötig, da das Programm sonst in die anderen *case*s laufen würde.
=>*Es wird ausgeschrieben "Ungerade Zahl".*

4.2 Schleifenanweisungen

Um Teile eines Programmes mehrfach auszuführen, benutzt man Schleifen.
Java verfügt über drei unterschiedliche Schleifenkonstrukte. Alle drei Formen sind aus anderen Programmiersprachen bekannt. Die *for*-Schleife kann alternativ auch mit Hilfe der *while*-Schleife bzw. der *do-while*-Schleife formuliert werden:

4.2.1 Die for-Schleife

Eine Eigenschaft der *for*-Schleife ist, dass man bereits zu Beginn der Schleife wissen muss, wie oft sie durchlaufen werden soll.
Die formale Syntax der *for*-Schleife lautet:

```
for (startwert; [<logischerAusdruck>]; schleifenwert)
    <Anweisung>
```

In den Klammern hinter dem Schlüsselwort *for* wird festgelegt, wie oft die Anweisung wiederholt wird. Zuerst wird der *startwert* initialisiert. Dann wird *logischerAusdruck* überprüft. Ist er wahr, werden die *Anweisung* durchlaufen. Im letzten Schritt wird der s*chleifenwert* ausgewertet.

Von da an werden immer wieder *logischerAusdruck - anweisung - schleifenwert* abgearbeitet, bis der logische Ausdruck falsch wird.

Beispiel
```
int i;
for (i = 1; i <= 10; i++)
{
        System.out.print(i + "  ");
}
System.out.println("\n" + i );
```

Auf dem Bildschirm erscheint folgende Ausgabe
```
1     2     3     4     5     6     7     8     9     10
11
```

Beim Durchlaufen der Schleife wird die Variable i zuerst mit dem Wert 1 initialisiert. Dann wird die Bedingung *i<=10* überprüft. Stimmt die Aussage, wird die Ausschreibeanweisung ausgeführt. Zum Schluß wird die Variable *i* um 1 hochgezählt. Dann beginnt die Auswertung der Schleife wieder bei der Bedingung, bis sie irgendwann nicht mehr wahr ist.
Nachdem alle Durchläufe erfolgt sind, hat unsere Variable *i* den Wert 11.

Statt *i++* könnte man auch schreiben *i=i+1* (Inkrementoperator).
Das Gegenstück ist der Dekrementoperator *i--*. Er verringert den Wert einer numerischen Variablen um 1.

Beispiel: `for (int i = 10; i >= 1; i--)`

Im Beispiel sehen sie auch, dass die Laufvariable *i* vom Typ (*int*) auch in der *for*-Schleife definiert werden darf. Dann ist Variable *i* aber nur innerhalb der Schleife definiert, d.h. das letzte
System.out.println("\n" + i);
im obigen Beispiel würde eine Fehlermeldung erzeugen, da *i* außerhalb der Schleife verwendet wird.

Inkrement- und Dekrementoperatoren gibt es als Präfix- und Postfix-Operatoren. Steht der Operator vor der Variablen *++i* wird der Operator angewandt, bevor der Wert des Ausdrucks zurückgegeben wird, ansonsten danach (*i++)*

Beispiel:
```
int i=4711;
System.out.print (i++ + "  " + ++i + "  " + i);
System.out.print (i-- + "  " + --i + "  " + i);
```

Ergebnis:
```
4711  4713  4713
4713  4711  4711
```

for-Schleifen können auch geschachtelt werden.

Beispiel:
```
for (int i=1; i<=3; i++)
{
        for (int j=1; j<=5; j++)
        {
                system.out.print(i*j + "\t");
        }
}
```
Auf dem Bildschirm erscheint folgende Ausgabe
```
1     2     3     4     5
```

```
2     4     6     8     10
3     6     9     12    15
```

Frage: Wie gehe ich nun vor, wenn ich ein Programm mit einer geschachtelten for-Schleife entwickeln will, dass folgende Ausgabe liefert:

```
      1     2     3     4     5
      2     4     6     8     10
      3     6     9     12    15
      4     8     12    16    20
      5     10    15    20    25
      6     12    18    24    30
```

Lösung:

4.3 while-Schleife

Die while-Schleife ist eine Struktur, mit der man Programmteile so lange wiederholen kann, bis eine Bedingung nicht mehr zutrifft.

```
while ([<logischerAusdruck>])
{
    <Anweisung>
}
```

LogischerAusdruck wird bewertet. Ist er wahr (true), wird der Block *Anweisung* direkt hinter dem *while* ausgeführt. Ergibt *LogischerAusdruck* falsch (false), wird die Programmausführung direkt hinter der schließenden Klammer der *while*-Schleife fortgesetzt. Wenn *LogischerAusdruck* bereits beim ersten Mal falsch ist, kann es sein, dass sie überhaupt nicht durchlaufen wird. Um die *while*-Schleife zu verlassen, muß man dafür sorgen, dass irgendwann die Bedingung nicht mehr erfüllt ist.

Beispiel: Umrechnungstabelle von Altgrad in Bogenmaß

```
final static float pi=3.14159f;
float x;
int i=0;
while (i <= 360)
{
            x = pi* i / 180;
            System.out.println(i + " : " + x);
            i++;
}
```

Vor Beginn der *while*-Schleife definiert man eine Variable so, dass man zum ersten Mal in die Schleife hinein läuft. Am Ende der Schleife sorgt man dafür, dass sich diese Variable ändert. Irgendwann hat sie einen Wert erreicht, der dafür sorgt, dass logischer Ausdruck nicht mehr wahr ist.
So verwendet man *while*-Schleife zum Einlesen einer Datei. Dabei wird so lange gelesen, bis keine Daten mehr vorhanden sind.

do-while-Schleife

Die do-Schleife ist eng mit der *while*-Schleife verwandt. Der Unterschied zwischen den beiden besteht im Zeitpunkt der Prüfung, ob weitergemacht werden soll oder nicht. Bei *do* findet die Prüfung am Ende der Schleife statt.. Die Schleife wird daher mindestens einmal durchlaufen, während es bei der *while*-Schleife vorkommen kann, dass sie überhaupt nicht durchlaufen wird.
Die allgemeine Form einer *do*-Schleife sieht so aus

```
do
    <Anweisung>
while ([<logischerAusdruck>]);
```

Beispiel einer do-Schleife

```
final static float pi=3.14159f;
float x;
int i=0;
do {
            x = pi* i / 180;
            System.out.println(i + " : " + x);
            i++;
} while (i <= 360);
```

Do leitet die Schleife ein. Die Schleife enthält einen Inkrementoperator (i++), der dazu dient, dass irgendwann i=361 wird und die Schleife beendet ist.
beachten Sie das Semikolon hinter dem while-Statement.

4.4 Literatur

Literatur:

- Helmut Erlenkötter und Volker Reher: Java - HTML, Skripts, Applets und Anwendungen, rowohlt:Hamburg 1997, 49-71
- Peter Buhrmann und Thomas Kretzberg: Java: interaktiv im World Wide Web, Addison-Wesley:Bonn 1997, S. 19-23.

4.5 Kontrollfragen

1.Gegeben sei die folgende *if*-Abfrage:

x=2, y=3, z=4
If (Bedingung)
{ System.out.println("wahr");
}
else
{ System.out.println("falsch");
}

Was liefert diese *if*-Abfrage bei folgenden Bedingungen ?

a) (x==2 && y<=3)
b) (x==2 || y!=2 && z*z>=16)
c) (y==3 && ! (z==1))
d) (x!=2 && y>3)
e) (x<y || x>z)

2. Schreiben Sie ein Programm, dass alle ungeraden Zahlen bis 100 addiert. Den Wert 100 übergeben Sie als Parameter an das Programm.

3. Notieren Sie die folgenden Aussagen in Java-Schreibweise für eine *while*-Schleife!

a) Solange *summe* ungleich 24
b) Solange *zahl* größer oder gleich x
c) Solange *x* minus *y* ungleich 234
d) Solange der Rest von *a* durch *b* ungleich 0

4. Was ist der Output des folgenden Programms ?

```
class schleife
{
        public static void main(String args[])
        { long i=5, j=4;
                while (i<=10 && j % 3 !=0)
```

```
                {
                        System.out.println(i+","+j);
                        i++;
                        j++;
                }
        }
}
```

5. Welche Aussagen zur *if*-Anweisung sind richtig?
   a) Der else-Zweig ist optional.
   b) Geschachtelte if-Anweisungen sind unzulässig.
   c) Innerhalb des if- bzw. else-Zweiges dürfen Anweisungsblöcke stehen.
   d) Der Bedingungsteil der Regel kann neben Ausdrücken des Typs boolean auch Ausdrücke des Typs byte, short, int, long enthalten.

6. Können alle Schleifenanweisungen durch die while-Schleife nachgebildet werden?
   a) Ja
   b) Nein