Java-Tutorial

– Programmieren lernen –

Karsten Brodmann

www.punkt-akademie.de

(Folge 9 bis 15 - 03.01.2023)

Kontrollstrukturen

Kontrollstrukturen sind Fallunterscheidungen und Wiederholungen. Sie steuern den dynamischen Ablauf der Anweisungsfolge eines Programms durch Bedingungen. Bedingungsausdrücke sind entweder wahr (`true`) oder falsch (`false`).

Fallunterscheidungen

Mithilfe von Fallunterscheidungen werden Verzweigungen implementiert. In Abhängigkeit von Bedingungen werden alternative Anweisungsfolgen ausgeführt.

`if`-Anweisung

Die `if`-Anweisung prüft eine Bedingung. Ist sie wahr, werden davon abhängige Anweisungen ausgeführt. Mehrere Anweisungen sind syntaktisch mittels eine Anweisungsblocks zu einer Anweisung zusammenzufassen. Ein Anweisungsblock wird in geschweifte Klammern eingeschlossen.

Trifft die formulierte Bedingung nicht zu, wird nach der `if`-Anweisung mit der Programmausführung fortgefahren.

Mithilfe eines optionalen `else`-Zweiges können explizit Anweisungen angegeben werden, die nur bei einer nicht zutreffenden Bedingung ausgeführt werden sollen. Sofern nicht explizit durch geschweifte Klammern anders angegeben, bezieht sich ein `else` immer auf das direkt vorangegangene `if`.

Bedingungsausdrücke können aus Teilbedingungen bestehen die mit logischen Operatoren verknüpft werden – siehe hierzu: Skript zu Folge 7, Abschnitt Wahrheitswerte.

Syntax:

```
if (bedingung) {
    anweisungen   // auszuführen, wenn Bedingung wahr
}
else {            // OPTIONAL
    anweisungen   // auszuführen, wenn Bedingung falsch
}
```

`if`-Anweisungen können ineinander geschachtelt werden (inkl. der optionalen `else`-Zweige).

Beispiele für `if`-Anweisungen:

```
int x = -42, y;

if (x < 0) x = -x;     // Absolutbetrag

if (x < 0) y = -x;     // y auf Absolutbetrag
else       y = x;      // von x setzen
```

Ein einfach nachzuvollziehendes Beispiel soll die Fallunterscheidung mit `if`-Anweisungen illustrieren. Es wird eine Monatszahl (1 bis 12) interaktiv eingelesen. Das Programm ermittelt daraus die zugehörige Jahreszeit. Für Zahlen, die keinem Monat entsprechen, wird „unbekannte Jahreszeit“ ausgegeben.

```
1  import de.pakad.tools.StdIn;
2  import de.pakad.tools.StdOut;
3
4  public class Season {
5      public static void main(String[] args) {
6          int month = StdIn.readInt("Bitte eine Monatszahl: ");
7
8          if (month >= 3 && month <= 5)
9              StdOut.println("Frühling");
10         else if (month >= 6 && month <= 8)
11             StdOut.println("Sommer");
12         else if (month >= 9 && month <= 11)
13             StdOut.println("Herbst");
14         else if (month == 12 || month == 1 || month == 2)
15             StdOut.println("Winter");
16         else StdOut.println("unbekannte Jahreszeit");
17     }
18 }
```

Quellcode 1: Season.java (`if`)

Bedingter Ausdruck

Soll lediglich eine Zuweisung vorgenommen werden, die in Abhängigkeit einer Bedingung einer Variablen alternative Werte zuweist, kann auch kurz ein bedingter Ausdruck verwendet werden.

Syntax:

```
variable = bedinging ? wahr-ausdruck : falsch-ausdruck;
```

Als Beispiel dient nochmals der Absolutbetrag:

```
int x = -42, y;

y = x < 0 ? -x : x;
```

`switch-case`-Anweisung

Sollen mehrere Fallunterscheidungen auf Basis konkret zu benennender Werte getroffen werden, kann alternativ eine `case-switch`-Anweisung verwendet werden. Die hat eine etwas gewöhnungsbedürftige Syntax. Sie ist quasi von C/C++ übernommen.

Als Beispiel modifizieren wir das Programm `Season`.

```
1  import de.pakad.tools.StdIn;
2  import de.pakad.tools.StdOut;
3
4  public class Season {
5      public static void main(String[] args) {
6          int month = StdIn.readInt("Bitte eine Monatszahl: ");
7
8          switch (month) {
9              case 3: case 4: case 5:   StdOut.println("Frühling");
10                 break;
11             case 6: case 7: case 8:   StdOut.println("Sommer");
12                 break;
13             case 9: case 10: case 11: StdOut.println("Herbst");
14                 break;
15             case 12: case 1: case 2:  StdOut.println("Winter");
16                 break;
17             default: StdOut.println("unbekannte Jahreszeit");
18         }
19     }
20 }
```

Quellcode 2: Season.java (`case-switch`)

Die Werte für die zu unterscheidenden Fälle werden jeweils hinter einem `case` notiert. Mehrere Fälle können zusammengefasst werden, also zu einer gemeinsamen Aktion führen. So wurden hier beispielsweise die Fälle 3, 4 und 5 zusammengefasst. Für die anderen Monatszahlen ist das analog erfolgt. Die Trennung der Fälle geschieht mittels `break`. Ohne `break` fällt der zu prüfende Wert quasi von oben nach unten durch und fasst alles zusammen, was nicht entsprechend getrennt ist.

Hinweis: Mit Java 12 wurde eine erweiterte Syntax eingeführt, die in manchen Fällen einfacher sein kann. Tatsächlich mehr Möglichkeiten ergeben sich daraus jedoch nicht.

Mittels der erweiterten Syntax können wir Fälle direkt zusammenfassen, eine Zuweisung vornehmen und dabei auf `break` verzichten. Das Konstrukt ist also weniger flexibel einsetzbar. Beachten Sie auch das jetzt erforderliche Semikolon hinter den zusammenfassenden Klammern.

```
1  import de.pakad.tools.StdIn;
2  import de.pakad.tools.StdOut;
3
4  public class Season {
5      public static void main(String[] args) {
6          int month = StdIn.readInt("Bitte eine Monatszahl: ");
7
8          String result = switch (month) {
9              case 3, 4, 5   -> "Frühling";
10             case 6, 7, 8   -> "Sommer";
11             case 9, 10, 11 -> "Herbst";
12             case 12, 1, 2  -> "Winter";
13             default -> "unbekannte Jahreszeit";
14         };
15         StdOut.println(result);
16     }
17 }
```

Quellcode 3: Season.java (neues `case-switch`)

Für die Zuweisung ist hier ein spezieller Zuweisungsoperator zu verwenden, der aus zwei Einzelzeichen bestehende Pfeil.

Wiederholungen

Für die wiederholte Ausführung von Programmanweisungen dienen Schleifen. Sie steuern die Anzahl der Wiederholungen mithilfe einer Bedingung. Diese kann aus logisch verknüpften Teilbedingunen bestehen.

Wir unterscheiden die Schleifen danach, ob die Bedingung im Kopf oder im Fuß der Schleife notiert wird. Steht die Bedingung im Kopf, sprechen wir auch von abweisenden

Scheifen. Ist die Bedingung falsch, werde die in ihr formulierten Anweisungen nicht ausgeführt. Steht die Schleifenbedingung dagegen im Fuß einer Schleife, so werden die darin enthaltenen Anweisungen mindestens einmalig ausgeführt.

Schleifen können ineinander geschachtelt werden.

Als durchgängiges Beispiel für die zu behandelnden Wiederholungen habe ich eine einfache Fakultätsberechnung ausgewählt. Sie kennen Sie aus der Schule:

$$x! = \begin{cases} 1 & \text{, für } x = 0 \\ 1 \cdot \ldots \cdot n & \text{, für } x \geq 1 \end{cases}$$

Diese schulmäßige Art der Definition ist unschön. Später betrachten wir die elegantere rekursive Definition. Für den Moment mag sie aber genügen.

`while`-Schleife

Die `while`-Schleife haben wir schon bei `Collatz.java` verwendet. Ihre Syntax lautet:

```
while (bedingung)
    anweisung
```

Mehrere abhängige Anweisungen sind syntaktisch zu einem Anweisungsblock zusammenzufassen.

Die `while`-Schleife prüft im Kopf, ob die Schleifenbedingung war ist. Wenn ja, dann werden die davon abhängigen Anweisungen ausgeführt. Am Ende der Anweisungsfolge angekommen, wird wieder die Schleifenbedingung überprüft. Ist sie falsch, wird die Schleife abgebrochen und die Programmausführung wird hinter der Schleife fortgesetzt. Im anderen Fall, wenn die Schleifenbedingung noch wahr ist, werden die Anweisungen im Schleifenrumpf wiederholt ausgeführt.

```
1  import de.pakad.tools.StdIn;
2  import de.pakad.tools.StdOut;
3
4  public class Faculty {
5      public static void main(String[] args) {
6          int x = StdIn.readInt("Bitte eine natürliche Zahl: ");
7          long fac = 1;
8          while (x>1) {
9              fac *= x;
10             x--;
11         }
12         StdOut.println(fac);
13     }
14 }
```

Quellcode 4: Faculty.java (`while`)

Der Code prüft nicht explizit, ob eine Eingabezahl überhaupt zulässig ist. Eine negative Zahl, für welche die Fakultät eigentlich nicht definiert ist, führt zum Ergebnis 1. In Zeile 10 wurde eine abkürzende Schreibweise für `fac = fac * x` gewählt. Alle arithemtischen Operatoren lassen sich in diesem Sinne abkürzen.

`for`-Schleife

Die `for`-Schleife ist, wie auch die `while`-Schleife kopfgesteuert. Allerdings werden Initialisierung, Schleifenbedingung und Update direkt im Kopf notiert. Gegebenenfalls dürfen die Elemente auch entfallen.

```
1  import de.pakad.tools.StdIn;
2  import de.pakad.tools.StdOut;
3
4  public class Faculty {
5      public static void main(String[] args) {
6          int x = StdIn.readInt("Bitte eine natürliche Zahl: ");
7          long fac = 1;
8
9          for (int i=2; i<=x; i++) // Init; Bedingung; Update
10             fac *= i;
11         StdOut.println(fac);
12     }
13 }
```

Quellcode 5: Faculty.java (`for`) - Variante 1

In der Initialisierung dürfen auch Variablen vereinbart werden, deren Gültigkeits-/Sichtbarkeitsbereich auf die Schleife begrenzt ist.

Zu Beginn der Schleife wird diese initialisiert, so ein solcher Ausdruck angegeben ist. Ist die Bedingung wahr, wird der Schleifenrumpf ausgeführt. Mehrere Anweisungen sind wiederum zu einem Anweisungsblock zusammenzufassen. Das letzte Element im Schleifenkopf ist ein Ausdruck, welcher die Variable, die die Schleife steuert modifiziert. Das können auch mehrere Ausdrücke sein, wenn es beispielsweise mehrere Variabeln in der Schleifenbedingung gibt und/oder diese Variablen nicht im Rumpf modifiziert werden sollen. Steuerungsvariablen im Schleifenrumpf zu modifizieren ist ein Fehler!

Mehrere Initialisierungen und/oder Updateausdrücke werden durch Kommata getrennt notiert.

Alternativ kann die Schleife auch so formuliert werden. Das Resultat ist das gleiche. Diese Schleife kommt ohne Initialisierungsausdruck aus.

```
for (/* keine Initialisierung */; x>0; x--)
    fac *= x;
```

`for` - Variante 2

do-while-Schleife

Diese Schleife ist fußgesteuert, wird also mindestens einmal durchlaufen. Ansonsten ähnelt die Schleife in der Funktionsweise der `while`-Schleife. Nur steht eben die zu prüfende Bedingung im Schleifenfuß.

```
1  import de.pakad.tools.StdIn;
2  import de.pakad.tools.StdOut;
3
4  public class Faculty {
5      public static void main(String[] args) {
6          int x = StdIn.readInt("Bitte eine natürliche Zahl: ");
7          long fac = 1;
8
9          do {
10             fac *= x;
11             x--;
12         } while (x>1);
13         StdOut.println(fac);
14     }
15 }
```

Quellcode 6: Faculty.java (`do-while`)

Für die Aufgabenstellung ist diese Schleife ungeeignet. Selbst bei einer negativen Zahl als Eingabe, wird sie einmal durchlaufen. Das kann natürlich durch eine vorgeschaltete `if`-Bedingung abgefangen werden, ist jedoch wenig elegant.

break und continue

Schleifen können vorzeitig beendet oder fortgesetzt werden. Letzteres bedeutet, einen neuen Schleifendurchlauf zu beginnen, noch bevor die letzte Abweisung im Schleifenrumpf ausgeführt wurde.

```
1  import de.pakad.tools.StdIn;
2  import de.pakad.tools.StdOut;
3
4  public class IsPrime {
5      public static void main(String[] args) {
6          long i, n = StdIn.readInt("Zahl: ");
7
8          for(i=2; i<=n/i; i++)
9              if (n%i == 0) break;  // wenn Zahl nicht prim -> Abbruch
10         if (i > n/i)  StdOut.println(n + " ist prim.");
11         else          StdOut.println(n + " ist nicht prim.");
12     }
13 }
```

Quellcode 7: IsPrime.java

Die `break`-Anweisung, springt aus der Schleife heraus und bricht sie so ab. Für den Abbruch eines Schleifendurchlaufs und dem vorzeitigen Beginn eines neuen, gibt es die `continue`-Anweisung.

Im obigen Code verhindert das `break` unnötige Schleifendurchläufe, wenn bereits feststeht, dass eine Zahl nicht prim ist.

`continue` wird analog verwendet.