4 Die grundlegenden Programmstrukturen

4.1 Die Sequenz

Ein Prozessor ist zu einem großen Teil damit beschäftigt, in zeitlicher Abfolge Befehl um Befehl eines Programms abzuarbeiten, wir sprechen von einem **sequentiellen** Prozess (im Gegensatz beispielsweise zu einem Lebewesen, in dem viele Prozesse miteinander (parallel) ablaufen). Darum ist in jeder Programmiersprache die Sequenz eine fundamentale Programmstruktur. Wir können diese Abfolge auch bildlich darstellen (Abb. 4.1).

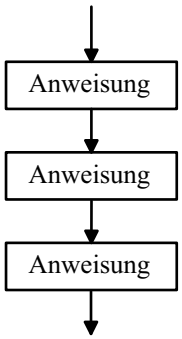

Abb. 4.1 *Die Sequenz*

Bereits unser erstes Programm zur Berechnung des Preises bestand aus einer Sequenz von Anweisungen. Um nochmals zu erleben, was eine Sequenz ist, wollen wird die Turtle anweisen, eine fünfstufige Treppe zu zeichnen und programmieren dazu jeden einzelnen Schritt explizit aus.

```
// TuEx1.java

import ch.aplu.turtle.*;
```

```
class TuEx1
{
  public static void main(String[] args)
  {
    Turtle john = new Turtle();   // Objekterzeugung

    john.forward(20);  // Gehe 20 Schritte vorwärts
    john.right(90);    // Drehe nach rechts
    john.forward(20);
    john.left(90);

    john.forward(20);
    john.right(90);
    john.forward(20);
    john.left(90);

    john.forward(20);
    john.right(90);
    john.forward(20);
    john.left(90);

    john.forward(20);
    john.right(90);
    john.forward(20);
    john.left(90);

    john.forward(20);
    john.right(90);
    john.forward(20);
    john.left(90);
  }
}
```

Das Programm ist zwar wenig elegant, aber wir erhalten das gewünschte Resultat (Abb. 4.2).

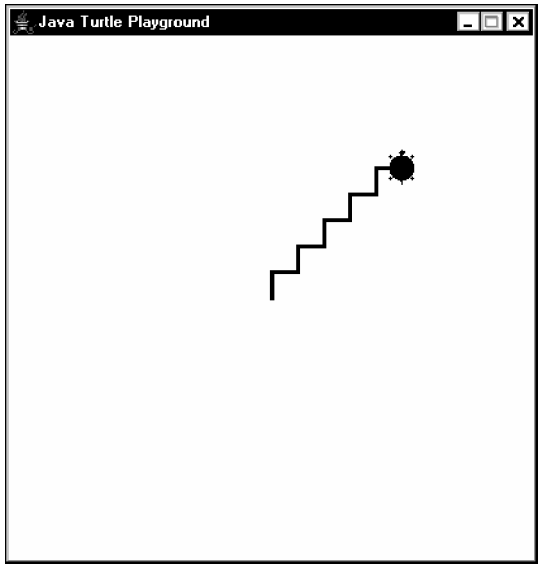

Abb. 4.2 *Die Ausgabe von TuEx1*

4.2 Die Iteration (Wiederholung)

Das Wiederholen von Anweisungsblöcken gehört zu den fundamentalen Aufgaben eines Computers. Man kann sogar davon sprechen, dass die Rechenmaschinen zu diesem Zweck erfunden wurden: Sie sollten dieselben Abläufe mit verschiedenen Werten immer und immer wieder abarbeiten und dazu den Menschen von mühsamer Routine entlasten. In Java werden Programmblöcke mit der linken geschweiften Klammer { eingeleitet und mit der rechten geschweiften Klammer } beendet. Die Klammern treten also immer paarweise auf. Im Innern eines Programmblocks können sich selbstverständlich weitere Programmblöcke befinden. Bei der Wiederholung wird im Allgemeinen ein gewisser Programmblock mehrmals durchlaufen (im Spezialfall aber auch einmal oder keinmal). Damit der Durchlauf beendet wird, muss eine **Bedingung** formuliert werden. Unter einer Bedingung versteht man einen Ausdruck, der **wahr** oder **falsch** ist. In Java gibt es drei Varianten, diese Bedingung anzuordnen: Zu Beginn oder am Ende des Wiederholblocks oder als for-Schleife.

4.2.1 Die while-Struktur

Bei dieser Wiederholstruktur wird vor dem Eintritt in den Wiederholblock (manchmal auch **Körper** der Wiederholung genannt) eine Bedingung auf ihren Wahrheitsgehalt geprüft. Falls die Bedingung wahr ist, wird der Wiederholblock ausgeführt und nachher zur Bedingungs-

prüfung zurückgekehrt. Umgangssprachlich formuliert handelt es sich um eine Struktur der Art:

```
Solange ( bedingung erfüllt ) führe aus
{
   Anweisungsblock
}
```

Ablaufdiagramme (Flussdiagramme) sind zwar in der Informatik verpönt, in diesem Zusammenhang liefern sie aber einen unübertrefflichen Einblick in den zeitlichen Ablauf (Abb. 4.3).

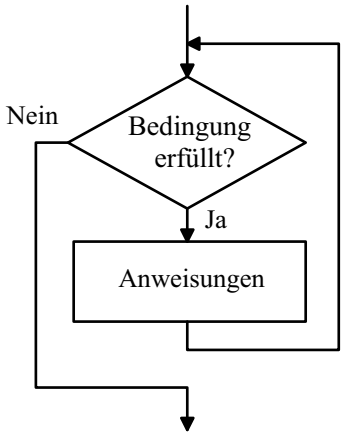

Abb. 4.3 *while-Struktur*

Da die Bedingung vor dem Eintritt in den Wiederblock geprüft wird, ist es auch möglich, dass der Block gar nie ausgeführt wird. Man spricht von einer **vorprüfenden** Wiederholung. Auf Grund der Rückführung des Programmablaufs sagt man statt while-Struktur auch oft while-**Schleife**.

Mit Hilfe der while-Struktur wird das Programm, welches die Turtle eine 5-stufige Treppe zeichnen lässt, wesentlich einfacher und eleganter.

```
// TuEx2.java

import ch.aplu.turtle.*;

class TuEx2
{
  public static void main(String[] args)
  {
    Turtle john = new Turtle();
```

```
    int i = 0;
    while (i < 5)
    {
       john.forward(20);
       john.right(90);
       john.forward(20);
       john.left(90);
       i = i + 1;
    }
  }
}
```

Wir benötigen dazu einen **Schleifenzähler** i, der bei jedem Durchgang durch die Schleife um 1 erhöht wird. Erreicht i den Wert 5, so ist die Laufbedingung i < 5 nicht mehr wahr und die Wiederholung wird abgebrochen.

Für die Anordnung der Blockklammern gibt es zwei übliche Konventionen, nämlich

```
while ( bedingung )
{
  Anweisungsblock
}
```

oder

```
while ( bedingung ) {
  Anweisungsblock
}
```

Beide haben Vor- und Nachteile: Die erste Version ergibt übersichtliche Blöcke, da beginnende und schließende Klammern immer übereinander stehen. Bei der zweiten Version gewinnt man eine Zeile (auf der nur eine Klammer steht). Obschon die zweite Version in der professionellen Java-Programmiergemeinde bevorzugt wird, wählen wir in diesem Buch die erste Version, welche die Ablauflogik wesentlich klarer zum Ausdruck bringt.

Ein zweites Beispiel stammt aus der Zinseszinsrechnung und behandelt bereits ein Problem, das wir ohne Programmierung nicht ganz leicht lösen könnten. Wir wollen herausfinden, wie lange wir ein Kapital zu einem bestimmten Zinssatz anlegen müssen, bis es sich verdoppelt hat. Wir verwenden dabei die Möglichkeiten, über das Console-Fenster einen Wert einzulesen, auf einen Tastendruck zu warten und das Programm zu beenden. Da die Methoden der Klasse `Console` mehrmals verwendet werden, ist es angebracht, eine Console-Referenz `c` zu erstellen, die mehrmals verwendet wird.

Statt der Zunahme eines Kapitals, könnten wir die Zunahme der Bevölkerung eines Landes bei gegebener jährlicher Zuwachsrate betrachten. Für das hier betrachtete Wachstumsverhalten gilt, dass die Zunahme pro Zeitschritt proportional zur aktuellen Größe ist, was das Verhalten von vielen Systemen richtig beschreibt und zu einem exponentiellen Wachstum führt.

```
// Zins.java

import ch.aplu.util.*;

class Zins
{
  public static void main(String[] args)
  {
    Console c = Console.init();
    double anfangskapital = 200;
    c.print("Zinssatz? ");
    double zinssatz = c.readDouble();
    int n = 0; // Jahre
    double kapital = anfangskapital;
    while (kapital < 2 * anfangskapital)
    {
      kapital = kapital * (1 + zinssatz/100);
      n++;
    }
    c.println("Laufzeit mind. " + n + " Jahre.");
    c.print("Zum Beenden eine Taste druecken");
    c.getKeyWait();
    c.terminate();
}
```

Bemerkungen:

- Der Text, welcher mit `print` ausgeschrieben wird, kann auf einfache Art mit dem Pluszeichen zusammengesetzt werden. Wir nennen dies **Stringkonkatenation**. Der Text muss immer in Anführungszeichen gesetzt werden, die Werte von numerischen Variablen werden automatisch in die Stringform umgesetzt
- In der Mathematik werden algebraische Größen mit einem einzigen Buchstaben bezeichnet, eventuell mit hoch- oder tiefstehenden Zeichen oder Zahlenindizes. Dies ist deswegen nötig, weil zwei hintereinander geschriebene Buchstaben als Produkt der Größen aufgefasst werden. In Java ist das Multiplikationszeichen * obligatorisch, so dass Variablennamen mit mehreren Buchstaben nicht nur erlaubt sind, sondern wegen der besseren Übersicht sogar bevorzugt werden.
- Wir verwenden die Leerschläge in großzügiger Art, um den Quelltext übersichtlich zu gestalten. Insbesondere werden Leerschläge vor und nach dem Gleichheitszeichen, nach Kommas (bei Parameteraufzählungen), nach Strukturierungsschlüsselwörtern (while usw.) und bei mathematischen Ausdrücken gemacht. Weitere Regeln sind aus den Programmbeispielen ersichtlich.
- Blöcke werden konsequent mit 2 Leerschlägen eingerückt und Zeilen innerhalb eines Blocks immer exakt untereinander begonnen
- Wir verwenden die in Java übliche abkürzende Schreibweise `n++` für `n = n + 1`. Entsprechend kann man statt `n = n -1` abkürzend `n--` schreiben

- Ein schlimmer Anfängerfehler ist grundsätzlich den Strichpunkt nach jeder Zeile zu setzen. Ein Strichpunkt am Ende der while-Zeile führt zwar zu einem syntaktisch korrekten Programm, da der Compiler dies als eine **leere Anweisung** betrachtet. Zur Laufzeit bleibt aber das Programm in der while-Schleife „hängen“, weil diese leere Anweisung als while-Block aufgefasst wird, in der auf die Laufbedingung kein Einfluß genommen wird.

> ! **Über die Formatierung von Programmen kann man unterschiedlicher Meinung sein, innerhalb eines Quellprogramms hält man sich aber konsequent an die sich selbst auferlegten Konventionen.**

Wir erkennen auch einen Nachteil der Computerlösung: Die Vermutung liegt nahe, dass die Verdoppelungszeit unabhängig vom gewählten Anfangskapital ist, also nur vom Zinssatz abhängt. Wir können die Vermutung zwar überprüfen, indem wir in einigen Testläufen das Anfangskapital ändern. Ganz offensichtlich ist dies aber kein Beweis für die Allgemeingültigkeit unserer Vermutung.

4.2.2 Die do-while-Struktur

Im Gegensatz zur while-Struktur erfolgt hier der Test für die Laufbedingung erst am Ende des Wiederholblocks. Man spricht von einer **nachprüfenden** Wiederholung. Umgangssprachlich formuliert handelt es sich um eine Struktur der Art:

```
Führe aus
{
   Anweisungsblock
} solange ( bedingung erfüllt )
```

Im Flussdiagramm ist der Ablauf klar ersichtlich (Abb. 4.4).

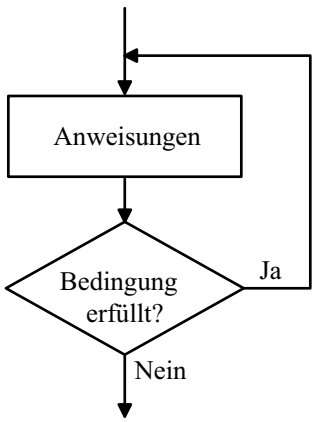

Abb. 4.4 *do-while-Struktur*

Da die Auswertung der Bedingung erst am Schluss erfolgt, wird der Wiederholblock mindestes einmal durchlaufen, was oft ein Nachteil ist. Aus diesem Grund wird die do-while-Struktur im Vergleich zur while-Struktur selten eingesetzt. Als Beispiel wählen wir wieder die 5-stufige Treppe.

```
// TuEx3.java

import ch.aplu.turtle.*;

class TuEx3
{
  public static void main(String[] args)
  {
    Turtle john = new Turtle();
    int i = 0;
    do
    {
       john.forward(20);
       john.right(90);
       john.forward(20);
       john.left(90);
       i++;
    }  while (i < 5);
  }
}
```

Man beachte, dass es sich bei der Bedingung um die Lauf- und nicht die Abbruchbedingung handelt. In vielen bekannten Programmiersprachen ist dies bei nachprüfenden Wiederholungen (repeat-until) gerade umgekehrt.

4.2.3 Die for-Struktur

Falls man in einer wiederholenden Struktur einen Schleifenzähler benötigt, kann die for-Struktur die while-Struktur vorteilhaft ersetzen. Dies ist bei numerisch orientierten Problemen oft der Fall, beispielsweise im Zusammenhang mit Vektoren. Das Zeichnen der Treppe mit 5 Stufen gehört auch zu Problemen, bei denen die for-Struktur angebracht ist. TuEx4 ist also die beste Version des Treppenprogramms.

```
// TuEx4.java

import ch.aplu.turtle.*;

class TuEx4
{
  public static void main(String[] args)
  {
    Turtle john = new Turtle();
```

```
    for (int i = 0; i < 5; i++)
    {
       john.forward(20);
       john.right(90);
       john.forward(20);
       john.left(90);
    }
  }
}
```

Für das anschauliche Verständnis der for-Struktur ist wieder ein Flussdiagramm geeignet (Abb. 4.5). Grau hinterlegt sind die Teile, welche in der for-Klammer angegeben werden.

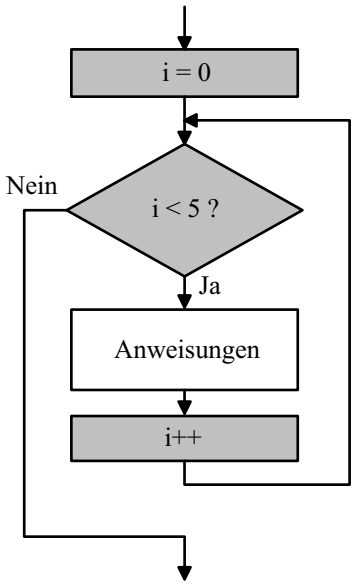

Abb. 4.5 *for-Struktur*

Man erkennt, dass es sich um eine spezielle, automatisierte Form einer while-Struktur handelt. Die Syntax ist sehr gewöhnungsbedürftig und kann leicht zu schwer auffindbaren Fehlern führen. Folgendes ist zu beachten:

- Der Schleifenzähler kann auch vor dem Eintritt in die for-Schleife deklariert werden und wird in der for-Klammer nur noch initialisiert. Er ist dann aber auch nach dem Ende der for-Schleife noch „sichtbar“. Es gehört aber zum guten Programmierstil, den Schleifenzähler in der for-Schleife zu „kapseln“, indem man ihn in und nicht außerhalb der for-Klammer deklariert

- Der Schleifenzähler wird im vorliegenden Fall bei jedem Durchlauf um 1 erhöht. Auch wenn es theoretisch möglich wäre, wird er nicht zusätzlich verändert, weil man die Auswirkungen schlecht durchblickt
- Man beachte, dass wie bei der while-Struktur ein unachtsam gesetzter Strichpunkt am Ende der for-Bedingung zu einer leeren Anweisung führt, die als Wiederholblock aufgefasst wird. Der eigentlich vorgesehene Wiederholblock befindet sich dann trotz der Einrückung außerhalb der for-Schleife, was syntaktisch richtig ist, aber zu einem bösartigen Laufzeitfehler führt.

Wer häufig in anderen Programmiersprachen programmiert, muss sich daran gewöhnen, dass es sich im Gegensatz zu vielen bekannten Programmiersprachen um die Laufbedingung und nicht um die Endbedingung (for-to-Struktur) handelt.

4.3 Die Selektion (Auswahl)

Programmverzweigungen auf Grund von bestimmten Bedingungen gehören zum grundlegenden Befehlssatz aller Prozessoren. Auf höhere Programmiersprachen übertragen heißt dies, dass ein logischer (boolescher) Ausdruck ausgewertet wird und je nach Ausgang ein Programmblock ausgeführt wird oder nicht.

4.3.1 Die if-Stuktur (einseitige Auswahl)

Am Flussdiagramm lässt sich der Ablauf wiederum am besten überblicken (Abb. 4.6).

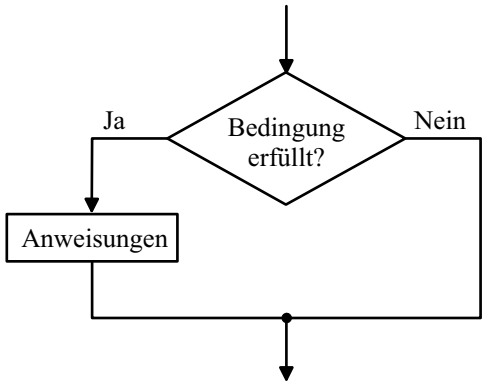

Abb. 4.6 *if-Struktur*

Falls die Bedingung wahr ist, verzweigt das Programm in den if-Block. Nachher wird es an derselben Stelle zusammengeführt, wie wenn die Bedingung nicht erfüllt ist. Mit anderen Worten, auf Grund der Bedingung wird also ein zusätzlicher Programmblock ausgeführt.

Umgangssprachlich handelt es sich um eine Struktur der Art

```
Falls ( bedingung erfüllt ) dann
{
   Anweisungsblock
}
```

Als Beispiel lösen wir eine Aufgabe aus dem Gebiet der Spiele. Wir werfen 1000 mal zwei Würfel (Doppelwurf) und betrachten jeweils die Augensumme. Dabei zählen wir, wie oft die Augensumme 8 beträgt. Es ist zu erwarten, dass diese Zahl von Versuchreihe zu Versuchreihe etwas unterschiedlich ausfällt. Interessant ist es, eine Vermutung über den Mittelwert aufzustellen. Für „Fleißaufgaben“ dieser Art ist der Computer hervorragend geeignet, denn er kann das Würfeln Millionen Mal schneller als der Mensch durchführen.

Im Programm lenken wir mit `Console.init()` zuerst alle Ausgaben in ein Console-Fenster um. Anschließend deklarieren wir einige Variablen, die wir im Programm benötigen. In der Simulation bedeutet das Werfen der beiden Würfel, dass der Computer zwei Zufallszahlen zwischen 1 und 6 generieren muss. Dazu verwenden wir die Java-Klasse `Random` und fordern mit der Methode `nextInt(6)` die nächste Zufallszahl im Bereich 0 bis 5 an. Wir verwenden auch neu den Operator ==, der zwei Größen auf Gleichheit überprüft.

In Java ist es zwingend nötig, die Zuweisung (mit dem Operator =) vom Vergleich (mit dem Operator ==) zu unterscheiden. Es gehört zu den bekanntesten Programmierfehlern, die Verdoppelung des Gleichheitszeichens beim Vergleich zu vergessen. In anderen Programmiersprachen wird für die Unterscheidung eine etwas vernünftigere Schreibweise verwendet.

```
// Wuerfeln.java

import java.util.*;
import ch.aplu.util.*;

class Wuerfeln
{
  public static void main(String[] args)
  {
    Console.init();
    int summe = 8;
    int wuerfe = 1000;

    int m, n;
    int bingo = 0;

    Random rnd = new Random();

    for (int i = 0; i < wuerfe; i++)
    {
      m = rnd.nextInt(6); // Zufallszahl 0 <= m < 6
      n = rnd.nextInt(6);
      m++; // Augenzahl 1. Wuerfel
```

```
      n++; // Augenzahl 2. Wuerfel
      if (n + m == summe)
      {
         bingo++;
      }
    }
    System.out.print("Auf " + wuerfe + " Wuerfe hatte ich "
                     + bingo + "x die Summe " + summe);
  }
}
```

Der if-Block besteht aus einer einzigen Anweisung, die den Wert von `bingo` um eins erhöht. Falls, wie hier, der Block nur aus einer einzigen Anweisung besteht, können die Blockklammern auch weggelassen werden. Man läuft aber dann in Gefahr, dass beim Hinzufügen einer weiteren Anweisung im if-Block die Klammern vergessen werden, was zu einem schwer auffindbaren Laufzeitfehler führt, da das Programm syntaktisch richtig bleibt.

Das Programm zeigt auch, dass es zum guten Programmierstil gehört, für numerische Werte, wie die Totalzahl der Würfe, eine Variable am Anfang des Programms zu deklarieren, statt sie irgendwo im Innern des Programms „hart zu verdrahten“. Dies gilt insbesondere dann, wenn man diese Zahl an mehreren Stellen verwendet. (Solche Zahlen werden ironisch auch **magic numbers** genannt.) Es ist nämlich unwahrscheinlich, dass man bei einer nachträglichen Modifikation des Werts alle Stellen auffindet, an denen dieser verändert werden muss. Meist wird man sich zufrieden geben, die erste gefundene Stelle zu modifizieren, was zu bösen Laufzeitfehlern führen kann.

4.3.2 Die if-else-Struktur (zweiseitige Auswahl)

Auch hier verschaffen wir uns einen Überblick mit einem Flussdiagramm (Abb. 4.7).

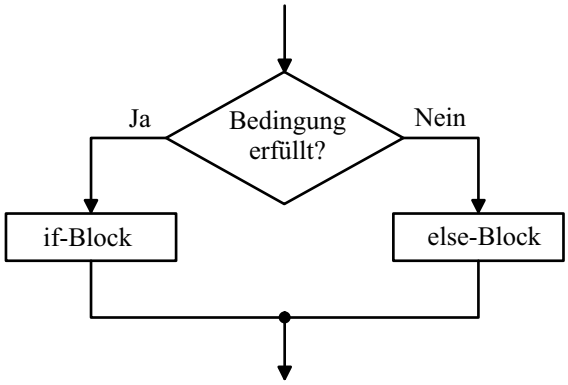

Abb. 4.7 *if-else-Struktur*

Umgangssprachlich können wir schreiben

```
Falls ( bedingung erfüllt ) dann
{
   if-Anweisungsblock
}
sonst
{
   else-Anweisungsblock
}
```

Im folgenden Programm zeichnet die Turtle einen Halbkreis (eigentlich ein fünfzigseitiges Polygon) im Gegenuhrzeigersinn und dann einen Halbkreis im Uhrzeigersinn.

```
// TuEx5.java

import ch.aplu.turtle.*;

class TuEx5
{
  public static void main(String[] args)
  {
    Turtle john = new Turtle();

    for (int i = 0; i < 100; i++)
    {
      john.forward(2);
      if (i < 50)
        john.left(3.6);  // Drehe 3.6 Grad
      else
        john.right(3.6);
    }
  }
}
```

Bei der Verschachtelung von if-else-Strukturen geschehen leicht Überlegungsfehler. Aus diesem Grund sind tiefe Verschachtelungen, d.h. weitere if-else-Strukturen innerhalb eines if- oder else-Blocks zu vermeiden. Falls man, wie im obigen Beispiel, die Blockklammern weglässt, ist es besonders gefährlich, einen weiteren else-Block einzufügen. Will man beispielsweise die Turtle-Farbe lediglich beim ersten Halbkreis verändern, sobald die Turtle eine Höhe von 25 überschreitet, so setzt man aus Fahrlässigkeit

```
// TuEx5a.java

import ch.aplu.turtle.*;
import java.awt.Color;

class TuEx5a
```

```
{
  public static void main(String[] args)
  {
    Turtle john = new Turtle();

    for (int i = 0; i < 100; i++)
    {
      john.forward(2);
      if (i < 50)
        john.left(3.6); // Drehe 3.6 Grad
        if (john.getY() > 25)
          john.setColor(Color.red);
      else
        john.right(3.6);
    }
  }
}
```

Das Programm ergibt keinen Syntaxfehler. Aber das `else` gehört trotz des Einrückens zum zweiten `if`, was aber nicht die Absicht war (**tangling-else problem**). Abhilfe schafft das korrekte Setzen von Blockklammern.

```
// TuEx5b.java

import ch.aplu.turtle.*;
import java.awt.Color;

class TuEx5b
{
  public static void main(String[] args)
  {
    Turtle john = new Turtle();

    for (int i = 0; i < 100; i++)
    {
      john.forward(2);
      if (i < 50)
      {
        john.left(3.6); // Drehe 3.6 Grad
        if (john.getY() > 25)
          john.setColor(Color.red);
      }
      else
        john.right(3.6);
    }
  }
}
```

Die Gefahr besteht nicht, wenn man immer, auch bei if- und else-Blöcken mit nur einer Anweisung, Blockklammern setzt.

4.3.3 Die switch-Struktur (Mehrfachauswahl)

Falls eine Variable mehrere verschiedene Werte annehmen kann und je nach Wert ein anderer Programmblock ausgeführt werden soll, bietet sich die switch-Struktur anstelle einer geschachtelten if-else-Struktur an. Die Logik der switch-Struktur stammt aus den Zeiten der Assembler-Programmierung und mutet deswegen etwas archaisch an.

Zuerst wird der Ausdruck der switch-Anweisung ausgewertet und dann der Reihe nach mit fest definierten (konstanten) Werten (auch Labels genannt) verglichen. Ergibt sich eine Übereinstimmung, so springt das Programm zum entsprechenden case-Block. Nachher läuft das Programm allerdings beim darunter stehenden case-Block weiter. Meist ist dies unerwünscht und man zwingt das Programm mit der break-Anweisung, die switch-Struktur zu verlassen.

Auch hier zeigt das Flussdiagramm (Abb. 4.8) das Verhalten anschaulich. Der Defaultblock wird ausgeführt, falls der Ausdruck a zu keiner Übereinstimmung mit den angegebenen Labels führt. Dieser Block kann auch weggelassen werden und das Programm springt bei fehlender Übereinstimmung zur nächsten Anweisung. Falls das `break` in einem case-Block fehlt, so spricht man vom **Durchfallen** in den nächsten Block. Erst beim nächsten `break` wird die switch-Strutkur verlassen. In seltenen Fällen ist dies erwünscht und sollte besonderes dokumentiert werden, oft wird aber das `break` vergessen, was zu einem schwerwiegenden Laufzeitfehler führt.

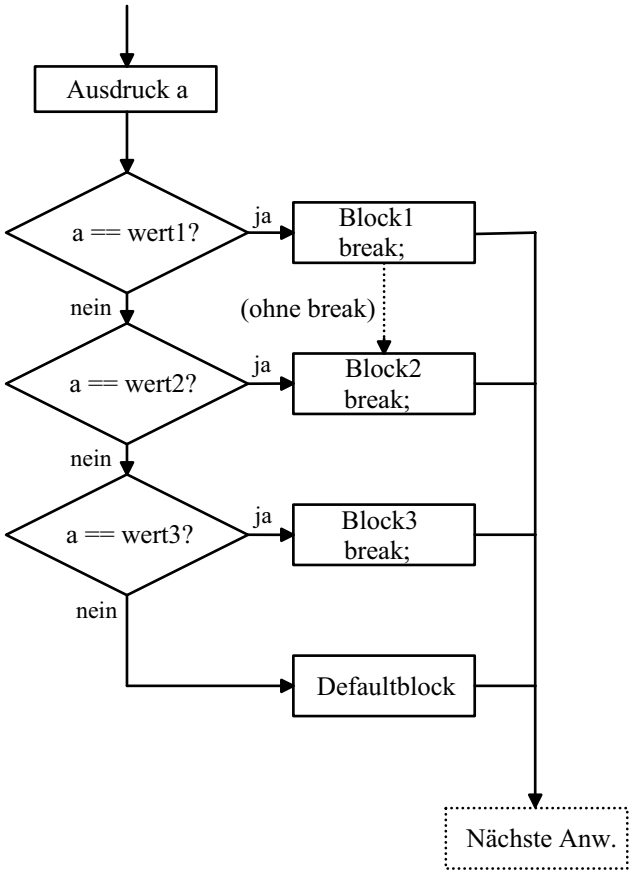

Abb. 4.8 *switch-Struktur*

Symbolisch für die Lernkurve beim Erlernen von Java soll die Turtle eine Treppe zeichnen, deren erster Tritt eine Stufenhöhe von 80 und deren zweiter Tritt eine Stufenhöhe von 40 aufweist. Nach der hohen Einstiegsschwelle soll die Stufenhöhe nur noch 20 betragen.

```
// TuEx6.java

import ch.aplu.turtle.*;

class TuEx6
{
  public static void main( String[] args )
  {
    Turtle john = new Turtle();
    for ( int i = 0; i < 5; i++ )
    {
      switch ( i )
```

```
      {
        case 0:
          john.forward( 80 );
          break;
        case 1:
          john.forward( 40 );
          break;
        default:
          john.forward( 20 );
          break;
      }
      john.right( 90 );
      john.forward( 20 );
      john.left( 90 );
    }
  }
}
```

Das `break` im Default-Block ist nicht nötig, wird aber zur Sicherheit hingesetzt, damit die Reihenfolge der case/default ohne Einfluss auf die Programmlogik verändert werden kann.

In `switch()` sind leider nur Variablen vom Typ `int`, `short`, `byte` oder `char` zulässig.

4.4 Varianten der for-Struktur

Die for-Struktur gehört zu den wichtigen Programmstrukturen von Java, insbesondere auch, weil sie sehr verschieden aufgebaut werden kann. Obschon es sich um eine lustige intellektuelle Herausforderung handelt, geistreiche for-Strukturen zu erfinden, zählt man exotisch aufgebaute for-Schleifen zum trickreichen Programmieren. War es in den Siebziger- und Achtzigerjahren des letzten Jahrhunderts unter Studierenden „in“, die kürzeste Variante eines Algorithmus durch Anwendung von Programmiertricks zu präsentieren, sind vielfach aus den Tricks Stolpersteine geworden, die man besser meidet.

> ! **Die Verwendung von Programmiertricks ist nicht Zeichen für fortgeschrittene professionelle Programmierung, sondern führt zu schwierig les- und wartbaren Programmen und macht deshalb nur in gut dokumentierten Ausnahmefällen Sinn.**

Da wir aber auch in der Lage sein müssen, Programme, die von anderen Programmierern geschrieben wurden, zu lesen und zu verstehen, besprechen wir hier die bekanntesten Varianten der for-Struktur. Sie gehen davon aus, dass sie die allgemeine Form

```
for (Startweisung; Laufbedingung; Schleifenanweisung)
```

```
{
   Schleifenkörper
}
```

besitzt, die als Ablaufschema in Abb. 4.9 dargestellt ist.

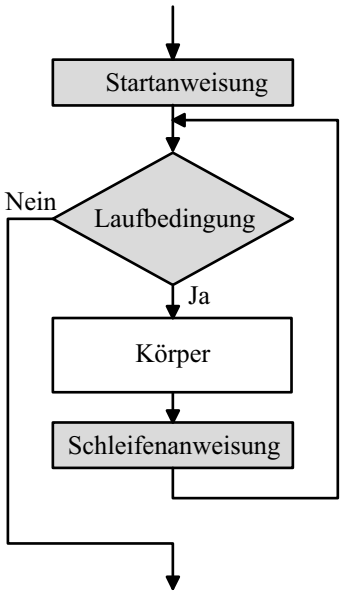

Abb. 4.9 *Allgemeine for-Strukur*

4.4.1 Mehrere Schleifenvariable

Schreiben wir mehrere Anweisungen, nur durch den **Kommaoperator** getrennt, hintereinander, so wird dies als eine einzige Anweisung betrachtet, die von links nach rechts abgearbeitet wird. In der for-Klammer muss der erste und letzte Term ein einziger Ausdruck sein. Setzen wir den Kommaoperator ein, so können wir bereits in der for-Klammer kompliziertere Operationen ausführen lassen. Typisch ist die Verwendung von zwei Schleifenvariablen, beispielsweise ergibt Programm `ForEx1` folgendes Resultat:

```
i: 0 k: 10
i: 1 k: 9
i: 2 k: 8
i: 3 k: 7
i: 4 k: 6
```

```
// ForEx1.java

import ch.aplu.util.*;

class ForEx1
{
  public static void main(String[] args)
  {
    int i, k;
    for (i = 0, k = 10; i < k; i++, k--)
    {
      Console.println("i: " + i + " k: " + k);
    }
  }
}
```

4.4.2 Weglassen einzelner Teile

Es ist erlaubt, in der for-Klammer einzelne Teile wegzulassen oder durch eine Leeranweisung zu ersetzen. Beispielsweise kann man die Initialisierung der Schleifenvariablen vor Eintritt in die Klammer erledigen:

```
int i = 0;
for (; i < 5; i++ )
```

oder die Schleifenanweisung in den Körper verlegen

```
for (i = 0; i < 5)
{
  ...
  i++;
}
```

4.4.3 Endlosschleife

Auf den ersten Blick mag eine Endlosschleife als unsinnig betrachtet werden, da man damit ein Programm erhält, das nie abbricht. Es gibt aber durchaus Möglichkeiten, eine Endlosschleife abzubrechen, beispielsweise durch eine `break`-Anweisung oder in ereignisgesteuerten Programmen durch einen Event. Im folgenden Programm versucht die Turtle verzweifelt nach Hause zu kommen, wobei sie sich nach jedem Schritt in einer zufälligen Richtung weiterbewegt. Das folgende Programm bricht ab, sobald die Turtle die Ziellinie y = 100 überschreitet. Sie gibt die Suche nach 20 Schritten auf. Es wird die Methode `Math.random()` verwendet, die bei jedem Aufruf eine Zufallszahl zwischen 0 und 1 liefert.

```
// ForEx2.java

import ch.aplu.turtle.*;

class ForEx2
{
  public static void main(String[] args)
  {
    Turtle john = new Turtle();

    int n = 0;
    for (;;)
    {
      john.forward(10);
      john.right(180 * (Math.random()-0.5));
      if (john.getY() > 100)
      {
        john.label("   Got home");
        break;
      }
      if (n++ == 20)
      {
        john.label("   Got lost");
        break;
      }
    }
  }
}
```

4.4.4 for-Schleife ohne Körper

Da die Schleifenanweisung am Ende des Wiederholblocks ausgeführt wird, kann man sie dazu benützen, Anweisungen auszuführen, die sich normalerweise im Schleifenkörper befinden. Dies macht natürlich, wenn überhaupt, nur bei sehr einfachen Anweisungen einen Sinn. Will man beispielsweise die Summe der Quadratzahlen von 1 bis 4 berechnen, so schreibt man klassisch

```
int sum = 0;
for (int i = 1; i <= 4; i++)
{
  sum = sum + i*i;
}
```

Zieht man alle Trickregister, so könnte man das Programm wesentlich kürzer, aber damit wesentlich unleserlicher schreiben. Das folgende Beispiel ist deshalb lustig, aber nicht nachahmenswert.

```
// ForEx3.java

import ch.aplu.util.*;

class ForEx3
{
  public static void main(String[] args)
  {
    int sum = 0;
    for (int i = 1; i <= 4; sum += i*i++);
    Console.println("Summe der Quadratzahlen 1..4: "
                    + sum);
  }
}
```