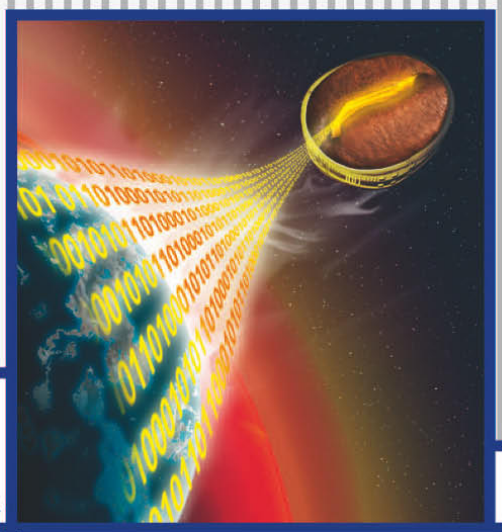
WWW.PEARSON-STUDIUM.DE
CWS
COMPANION WEBSITE



it informatik

Reinhard Schiedermeier

Programmieren mit Java

2., aktualisierte Auflage

PEARSON Studium

Kontrollstrukturen

3

ÜBERBLICK

» Der Begriff „Kontrollstrukturen“ bezeichnet alle Sprachmittel, die die Ausführungsreihenfolge von Anweisungen steuern. Aus struktureller Sicht liegen die Kontrollstrukturen eine Abstraktionsebene über den bisher betrachteten, einfachen Anweisungen (Variablendefinition, Wertzuweisung, Ausgabeanweisung). Mithilfe von Kontrollstrukturen lassen sich Algorithmen implementieren, die interessante und komplexe Berechnungen ausführen. „Von Hand“ ließen sich diese nicht mehr mit annehmbarem Aufwand durchführen.

- Im ersten Abschnitt 3.1 werden zunächst **Alternativen** in Form von if-Anweisungen vorgestellt. Alternativen erlauben es, untergeordnete Anweisungen wahlweise auszuführen oder zu überspringen. Die Entscheidung darüber wird von einer Bedingung gesteuert, die sich auf den vorhergehenden Programmverlauf bezieht. Programme wickeln damit nicht mehr einfach alle Anweisungen nacheinander ab, sondern nehmen, abhängig von Vorgabedaten, unterschiedliche Wege durch den Code.
- Als Bedingungen standen im ersten Abschnitt nur einfache Größenvergleiche zwischen Zahlenwerten zur Verfügung. Im Abschnitt 3.2 wird zuerst der Typ boolean für **Wahrheitswerte** eingeführt. Die Bedingungen von if-Anweisungen erweisen sich als Ausdrücke, deren Wert ein boolean-Ergebnis ist. Mit relationalen und logischen Operatoren lassen sich aus einfachen Bedingungen immer komplexere zusammenfügen.
  Es erweist sich, dass sich Bedingungen und arithmetische Ausdrücke aus der Sicht von Java nur im Typ des Ergebnisses unterscheiden und ansonsten einheitlichen Berechnungsvorschriften folgen.
- Neben den if-Anweisungen sind **Schleifen** die zweite, wichtige Art von Kontrollstrukturen. In Abschnitt 3.3 werden zunächst einfache while-Schleifen eingeführt. Durch die Freiheit verschiedene Kontrollstrukturen ineinander zu schachteln, lassen sich mit wenigen Sprachmitteln sehr komplexe Abläufe formulieren.
- Abschnitt 3.4 stellt die beiden Anweisungen break und continue vor, die in erster Linie im Zusammenhang mit Schleifen nützlich sind. Obwohl sie auf der einen Seite manchmal zu kürzerem Quelltext führen, ziehen sie auch möglicherweise unübersichtliche Abläufe nach sich.
- Variablen wurden bisher immer zu Beginn des Codes definiert und im Rest des Programms verwendet. Sie können aber auch *innerhalb* von Kontrollstrukturen definiert werden und stehen dann nur dort zur Verfügung. Daraus ergeben sich unterschiedliche **Gültigkeitsbereiche**, die in Abschnitt 3.5 besprochen werden. Eingeschränkte Gültigkeitsbereiche tragen ganz entscheidend dazu bei, Abhängigkeiten zwischen verschiedenen Programmteilen zu vermeiden und führen damit zu flexiblerem Code.
- In Abschnitt 3.6 wird eine zweite Art von Schleifen, die sogenannten **for-Schleifen**, vorgestellt. Konzeptionell bieten for-Schleifen gegenüber while-Schleifen keine neuen Möglichkeiten, erlauben aber in vielen Fällen kompakten und prägnanten Code. Dem steht eine etwas barocke Syntax und Semantik gegenüber, die letztlich aus Vorläufersprachen von Java stammt und sich über lange Zeit hinweg eingebürgert hat.

- Schließlich werden im letzten Abschnitt 3.7 **switch-Anweisungen** eingeführt. Ebenso wie for-Schleifen sind switch-Anweisungen verhältnismäßig komplizierte Kontrollstrukturen und stammen gleichfalls aus der älteren Programmiersprache C. switch-Anweisungen werden nicht allzu oft gebraucht, erlauben aber in bestimmten Situationen eine übersichtliche Formulierung von längeren Fallunterscheidungen. «

Einfache Anweisungen

Im vorhergehenden Kapitel wurden Variablendefinitionen, Wertzuweisungen und Ausgabeanweisungen vorgestellt. Diese drei Arten von Anweisungen werden als „einfache Anweisungen“ bezeichnet, weil sie nicht mehr in kleinere Anweisungen zerlegt werden können.

In diesem Kapitel werden Kontrollstrukturen eingeführt. Kontrollstrukturen sind „zusammengesetzte Anweisungen“, die als Bausteine wieder vollständige, untergeordnete Anweisungen enthalten.

Sequenz = Anweisungsfolge

Schon das schlichte Aneinanderreihen von Anweisungen lässt sich als Kontrollstruktur sehen, als eine „Anweisungsfolge“ oder „Sequenz“. Eigentlich ganz selbstverständlich werden die Anweisungen einer Sequenz genau in der Reihenfolge ausgeführt, in der sie im Quelltext stehen.

Algorithmus = Rechenvorschrift

Mit den Kontrollstrukturen dieses Kapitels lassen sich komplexe, tief geschachtelte Abläufe konstruieren. Eine solche Rechenvorschrift wird als **Algorithmus** bezeichnet.

Einen neuen Algorithmus zu erfinden ist keine triviale Angelegenheit. In erster Linie ist dazu Erfindungsreichtum und Kreativität nötig. Es gibt kein „Rezept“, nach dem sich ein neuer Algorithmus ohne viel Nachdenken und durch bloßes Anwenden von einfachen Regeln „konstruieren“ ließe.

Darüber hinaus braucht man eine klare Vorstellung davon, welche Bausteine eines Algorithmus in einer konkreten Programmiersprache überhaupt formuliert werden können und welche nicht. Ein Ziel dieses Kapitels ist es, Ihnen diese Vorstellung zu vermitteln.

Schließlich spielt auch Erfahrung eine Rolle. Bestimmte Muster und Schemata wiederholen sich und können, einmal verstanden, bei neuen Problemen in ähnlicher Form erneut angewendet werden. Auch dabei hilft Ihnen dieses Kapitel. Sie werden eine Reihe von Algorithmen kennenlernen und vielleicht an anderer Stelle und in einem ganz anderen Zusammenhang wieder umsetzen können.

3.1 if-Anweisungen

if: Anweisung ausführen oder auslassen

In einer **if**-Anweisung (auch „Alternative“, „bedingte Anweisung“ oder „Verzweigung“) wird eine untergeordnete Anweisung nur dann ausgeführt, wenn eine bestimmte Voraussetzung („Bedingung“, *condition*) erfüllt ist. Andernfalls wird die untergeordnete Anweisung ausgelassen.

Eine **if**-Anweisung wird folgendermaßen geschrieben:

```
if ( condition )
    statement
```

Das Wort **if** und die runden Klammern sind fester Bestandteil der **if**-Anweisung. Beachten Sie, dass die **if**-Anweisung, für sich betrachtet, nicht mit einem Semikolon abgeschlossen wird. Allerdings kann das Semikolon Teil der untergeordneten Anweisung *statement* sein.

Die folgende **if**-Anweisung gibt den Text „snow“ aus, wenn die Variable **temperature** einen negativen Wert hat. Andernfalls wird nichts ausgegeben.

```
if(temperature < 0)
    System.out.println("snow");
```

In dieser **if**-Anweisung wird zuerst der Wert der Variablen **temperature** geprüft:

- Ist der Wert negativ, dann wird die nachfolgende Ausgabeanweisung ausgeführt.
- Andernfalls, wenn der Wert null oder positiv ist, wird die Ausgabeanweisung ausgelassen.

Bedingung

Bedingung = Ausdruck mit ja/nein-Ergebnis

Die Bedingung in den runden Klammern einer **if**-Anweisung ist eine neue Art von Ausdruck. Entweder er „trifft zu“ oder er „trifft nicht zu“. Eine dritte Möglichkeit außer diesen beiden gibt es nicht. Ein derartiger Ausdruck heißt „**boolean**-Ausdruck“, das Ergebnis seiner „Berechnung“ ist ein **Wahrheitswert**. **boolean**-Ausdrücke werden in Abschnitt 3.2 im Detail erklärt.

Vergleichsoperatoren

Relationale Operatoren

In der Bedingung einer **if**-Anweisung können zwei Ausdrücke verglichen werden. Dazu wird ein **relationaler Operator** („Vergleichsoperator“) benutzt. Java kennt die folgenden relationalen Operatoren:[1]

<	echt kleiner
<=	kleiner oder gleich
>	echt größer
>=	größer oder gleich
==	gleich
!=	nicht gleich

1 Die Schreibweise der letzten beiden Vergleichsoperatoren ist etwas gewöhnungsbedürftig. Für den Test auf Gleichheit mit dem Operator == wird manchmal irrtümlich nur ein einzelnes Gleichheitszeichen geschrieben, wie aus der Mathematik gewohnt. Unglücklicherweise ist die Konstruktion mit einem Gleichheitszeichen unter Umständen auch noch syntaktisch zulässig und wird vom Compiler übersetzt, hat aber dann die falsche Wirkung.

Gegensatz zu arithmetischen Operatoren

Die relationalen Operatoren bilden eine neue Gruppe von Operatoren, neben den schon bekannten arithmetischen Operatoren. Sie verknüpfen je zwei Rechenausdrücke, liefern als Ergebnis aber einen Wahrheitswert und keinen Zahlenwert.

Priorität und Assoziativität

Wie die arithmetischen Operatoren hat jeder relationale Operator eine charakteristische Priorität und Assoziativität (siehe Anhang B). Allgemein binden die Vergleichsoperatoren von links nach rechts und schwächer als die Rechenoperatoren. Der folgende Vergleich ist syntaktisch korrekt und trifft zu:

2 + 3 < 2 * 3 → 5 < 6

Beispiel: Größter von drei Werten

Den Einsatz von **if**-Anweisungen zeigt das folgende Beispiel: In den drei Variablen **a**, **b** und **c** seien drei beliebige Zahlenwerte gespeichert. Der größte der drei Werte soll in die Variable **max** kopiert und der Wert dieser Variablen dann ausgegeben werden.

Dieses Problem lässt sich mit folgendem Algorithmus lösen, der zunächst umgangssprachlich formuliert wird:

1. Zunächst wird angenommen, dass **a** der größte Wert sei.
2. Anschließend wird dieser erste mögliche Maximalwert mit **b** verglichen. Falls **b** tatsächlich größer ist, wird der Maximalwert auf **b** korrigiert.
3. In der Variablen **max** ist jetzt der größere der Werte **a** und **b** gespeichert. Dieser Wert wird jetzt mit **c** verglichen. Falls **c** noch größer ist, wird der Maximalwert erneut korrigiert, diesmal auf **c**.
4. Schließlich steht hier in **max** der größte der drei Werte. Er kann ausgegeben werden.

Das entsprechende Java-Programm lautet:

```
class Max3 {
   public static void main(String[ ] args) {
      int a = Integer.parseInt(args[0]);
      int b = Integer.parseInt(args[1]);
      int c = Integer.parseInt(args[2]);
      int max = a;                    // 1.
      if(b > max)                     // 2.
         max = b;
      if(c > max)                     // 3.
         max = c;
      System.out.println(max);        // 4.
   }
}
```

Die Nummern der einzelnen Schritte des umgangssprachlich formulierten Algorithmus sind als Kommentare im Quelltext eingefügt.

Zweiseitige if-Anweisungen

if/else: Entscheidung zwischen zwei Anweisungen

Eine Erweiterung der einfachen einseitigen **if**-Anweisung ist die zweiseitige **if**-Anweisung mit folgendem Aufbau:

```
if ( condition )
   statement
else
   statement
```

Wie **if** ist auch **else** ein Schlüsselwort.

Genau eine Anweisung wird ausgeführt

Zuerst wird die *condition* geprüft. Trifft sie zu, wird die erste Anweisung ausgeführt, andernfalls die zweite Anweisung. In jedem Fall wird genau eine der beiden Anweisungen durchlaufen, aber keinesfalls alle beide oder überhaupt keine.

Im folgenden Beispiel wird die Quadratwurzel von **x** ausgegeben, wenn **x** größer oder gleich 0 ist, und ansonsten der Text „undefiniert“:

```
double x = ...;
if(x >= 0)
   System.out.println(Math.sqrt(x));
else
   System.out.println("undefiniert");
```

Vergleich von Floatingpoint-Werten

Relationale Operatoren sind polymorph

Die Vergleichsoperatoren können ganze Zahlen und Floatingpoint-Werte vergleichen. Gemischte Operandentypen werden über implizite Typkonversion angeglichen, wie bei den arithmetischen Operatoren.

Rundungsfehler bei Floatingpoint-Werten

Die **int**-Arithmetik ist frei von Rundungsfehlern, nicht aber die Floatingpoint-Arithmetik. Deshalb liefert der Vergleich von **double**-Werten mit **==** und **!=** manchmal unerwartete Ergebnisse, wie das folgende Beispiel zeigt:

```
double a = 1.0/3.0;
double b = 10 + a - 10;
if(a == b)
   System.out.println("gleich");
else
   System.out.println("verschieden");
```

Obwohl **a** und **b** rechnerisch gleich sein sollten, gibt das Programm **verschieden** aus.

Floatingpoint-Werte mit Bereichen vergleichen

Im Allgemeinen ist es ratsam, bei Floatingpoint-Vergleichen Toleranzgrenzen vorzusehen, statt exakte Werte abzuprüfen. Mit dem folgenden Vergleich[2]

```
if(Math.abs(a - b) < 1E-10)     // statt a == b
   ...
```

2 Die Bibliotheksfunktion abs liefert den Absolutwert des Argumentes.

gibt das Programms **gleich** aus. Die Wahl eines passenden Toleranzwertes ist nicht immer ganz einfach.

Geschachtelte if-Anweisungen

Eine **if**-Anweisung kontrolliert eine oder zwei untergeordnete Anweisungen. Auch die ganze **if**-Anweisung selbst ist wieder eine Anweisung. Im Gegensatz zu einfachen Anweisungen, wie Definitionen und Wertzuweisungen, bezeichnet man eine **if**-Anweisung als „zusammengesetzte Anweisung", weil sie als Bestandteile untergeordnete Anweisungen enthält.

Kontrolle über beliebige andere Anweisung

if-Anweisungen können geschachtelt werden. Die untergeordnete Anweisung ist in diesem Fall selbst wieder eine **if**-Anweisung.

if-Anweisung kontrolliert andere if-Anweisung

Im folgenden Beispiel ist eine geschachtelte **if**-Anweisung sinnvoll anwendbar: Eine quadratische Gleichung $ax^2 + bx + c = 0$ hat keine, eine oder zwei Lösungen, abhängig vom Vorzeichen der Diskriminante $d = b^2 - 4ac$. Das folgende Programm gibt die Lösungen aus:

```
class SolveSqrt {
   public static void main(String[ ] args) {
      double a = Double.parseDouble(args[0]);
      double b = Double.parseDouble(args[1]);
      double c = Double.parseDouble(args[2]);
      double d = b*b - 4*a*c;
      if(d < 0)
         System.out.printf("no solution%n");
      else
         if(d == 0)
            System.out.printf("1 solution: %g%n",
               -b/(2*a));
         else
            System.out.printf("2 solutions: %g, %g%n",
               (-b + Math.sqrt(d))/(2*a),
               (-b - Math.sqrt(d))/(2*a));
   }
}
```

Das gleiche Ergebnis könnte ohne geschachtelte **if**-Anweisungen berechnet werden:

```
   ...
      double d = b*b - 4*a*c;
      if(d < 0)
         System.out.printf("no solution%n");
      if(d == 0)
         System.out.printf("1 solution: %g%n",
            -b/(2*a));
      if(d > 0)
         System.out.printf("2 solutions: %g, %g%n",
            (-b + Math.sqrt(d))/(2*a),
            (-b - Math.sqrt(d))/(2*a));
```

Die zweite Lösung ist allerdings weniger effizient: In jedem Fall werden drei Vergleiche ausgeführt. In der ersten Lösung sind nur ein oder zwei Vergleiche notwendig. Das spielt bei diesen Beispielprogrammen sicher keine Rolle. Später kommen aber Objekte ins Spiel, deren Vergleich durchaus Rechenzeit kosten kann.

Untergeordnete **if**-Anweisungen können im ersten Zweig der übergeordneten **if**-Anweisung stehen oder im zweiten Zweig oder auch in beiden.

Blöcke als Anweisungsgruppe

if-Anweisung kontrolliert nur eine oder zwei Anweisungen

In der bisher gezeigten Form kontrolliert eine **if**-Anweisung nur eine oder zwei untergeordnete Anweisungen. Oft möchte man aber eine ganze Liste von Anweisungen von einer einzigen Bedingung abhängig machen.

Zu diesem Zweck lässt sich eine Folge von beliebig vielen Anweisungen mit geschweiften Klammern zu einem **Block** gruppieren:

Gruppieren einer Sequenz zu einem Block

```
{
   statement
   statement
    ...
}
```

Nach außen wird ein Block wie eine einzige, allerdings zusammengesetzte Anweisung behandelt und kann beispielsweise komplett einer **if**-Anweisung untergeordnet werden. Das vorhergehende Programm kann etwas beschleunigt werden, wenn die Quadratwurzel der Diskriminante im Fall von zwei Lösungen nur einmal berechnet und dann zweimal benutzt wird:

```
...
if(d == 0)
   ...
else {
   d = Math.sqrt(d);
   System.out.printf("2 solutions: %g, %g%n",
      (-b + d)/(2*a),
      (-b - d)/(2*a));
}
...
```

Der **else**-Zweig besteht hier aus einem Block, der zwei Anweisungen enthält.

Leerer Block

Die Klammern eines Blocks dürfen beliebig viele Anweisungen enthalten. Das schließt einen leeren Block ein, der überhaupt keine Anweisungen enthält:

```
{}
```

Der leere Block hat keine Wirkung im Programm. Er ist gleichwertig mit der leeren Anweisung, die nur aus einem Semikolon besteht:

```
;
```

Abgesehen von der Gruppierung eingeschlossener Anweisungen haben Blöcke noch weitere Eigenschaften, die in Abschnitt 3.5 diskutiert werden.

Dangling Else

Die Möglichkeit zur Schachtelung ein- und zweiseitiger **if**-Anweisungen hat ein eher lästiges, syntaktisches Problem zur Folge: In einer Anweisung der Gestalt[3]

Syntaktische Unregelmäßigkeit

if(*condition1*) **if**(*condition2*) *statement1* **else** *statement2*

könnte man das einzelne **else** dem ersten oder dem zweiten **if** zuordnen. Man bezeichnet ein **else** in dieser Stellung deshalb als *dangling else*. Die beiden Interpretationen entsprechen den folgenden Programmfragmenten, in denen die beabsichtigte Struktur durch Einrückung deutlich gemacht ist:

Interpretationsmöglichkeiten

```
if (condition1)
    if (condition2)
        statement1
    else
    statement2
```

```
if (condition1)
    if (condition2)
        statement1
    else
        statement2
```

Der Compiler ignoriert allerdings die Einrückung vollkommen, er orientiert sich ausschließlich am Programmtext. Dieser ist aber in beiden Fällen genau der Gleiche.

Entscheidung in Java

Eine Mehrdeutigkeit ist nicht zulässig, deshalb gilt folgende Regel: Ein **else** bezieht sich immer auf das *textuell letzte freie* **if** im selben Block.[4] Ein „freies **if**“ ist noch keinem **else** zugeordnet. Von den beiden oben gezeigten Interpretationen gilt also die rechte.

Trotz der eindeutigen Regelung werden geschachtelte **if**-Anweisungen leicht falsch gelesen. Im Zweifelsfall sollte man zusätzliche Block-Klammern einfügen, selbst wenn das formal nicht nötig wäre:

Explizites Klammern vermeidet Fehler

```
if(condition1)  {
      if(condition2)
            statement1
      else
            statement2
}
```

Hier gibt es keinen Zweifel mehr über die Struktur der geschachtelten **if**-Anweisung. Ebenfalls mit Block-Klammern kann ein *dangling else* bei Bedarf dem ersten **if** zugeordnet werden:

3 Hier mit Absicht ohne Zeilenumbruch abgedruckt.

4 Die gleiche Regel gilt auch in allen anderen Programmiersprachen, in denen if-Anweisungen diese syntaktische Gestalt haben und in denen deshalb ebenfalls das Problem des *dangling else* auftritt.

```
if(condition1) {
    if(condition2)
       statement1
}
else
    statement2
```

if-Kaskade

Tiefe Verschachtelung von if-Anweisungen

if-Anweisungen können beliebig tief geschachtelt werden. Man spricht dann von einer „**if**-Kaskade". Eine **if**-Kaskade kann zum Beispiel benutzt werden, um einen einzelnen Wert mit einer Liste von Möglichkeiten zu vergleichen.

Im folgenden Beispiel wird eine Monatszahl (1 = Januar bis 12 = Dezember) in der Variablen **month** geliefert. In **days** soll die Anzahl der Tage des betreffenden Monats gespeichert werden, wobei Schaltjahre unberücksichtigt bleiben:[5]

```
if(month == 1)
   days = 31;
else
   if(month == 2)
      days = 28;
   else
      if(month == 3)
         days = 31;
      ...
         else
            if(month == 12)
               days = 31;
```

Layout für if-Kaskaden

Rückt man in **if**-Kaskaden konsequent mit jeder Schachtelungsebene weiter nach rechts ein, dann passt die ganze Konstruktion bald nicht mehr auf eine normale Textseite. Stattdessen kann man in diesem Fall das Einrückungsschema aufgeben und die **if**-Anweisungen untereinander anordnen. Für den Compiler ist das ohnehin belanglos, und ein Leser kommt damit vielleicht besser zurecht:

```
if(month == 1)
    days = 31;
else if(month == 2)
    days = 28;
else if(month == 3)
    days = 31;
...
else if(month == 12)
    days = 31;
```

5 Diese Konstruktion dient hier nur zur Illustration. Das Problem lässt sich mit anderen Mitteln eleganter lösen.

3.2 Wahrheitswerte

Datentyp boolean

Typ boolean für Wahrheitswerte

Die Bedingung in einer **if**-Anweisung ist ein Ausdruck mit einem Wahrheitswert als Ergebnis. Wahrheitswerte sind in Java ein eigenständiger Typ mit der Bezeichnung **boolean**. **boolean** steht gleichrangig neben **int** und **double**, ist aber im Gegensatz zu diesen kein numerischer Typ.

Es gibt nur zwei **boolean**-Werte, die die Bedeutung „wahr“ und „falsch“ haben. Die beiden **boolean**-Konstanten heißen

true für „wahr“ und
false für „falsch“.

boolean-Literale

Explizit im Programmtext genannte Konstanten werden als **Literale** bezeichnet. **true** und **false** sind die beiden vordefinierten **boolean**-Literale. Zahlenkonstanten, also Konstanten der numerischen Typen sind ein Sonderfall von allgemeinen Literalen und werden „Numerale“ genannt (siehe Seiten 41 und 52).

Der Typ **boolean** ist nicht kompatibel zu **int** und **double**, oder umgekehrt, denn **boolean**-Werte haben weder eine numerische Interpretation noch eine Reihenfolge.[6]

Relationale Operatoren

Relationale Operatoren liefern boolean-Ergebnis

Die bereits weiter oben vorgestellten relationalen Operatoren akzeptieren Zahlen als Operanden und liefern einen **boolean**-Wert als Ergebnis.

Ein Ausdruck mit einem relationalen Operator wird nach genau dem gleichen Schema ausgewertet wie ein arithmetischer Ausdruck. Zum Beispiel binden im Ausdruck

```
2 + 3 < 2*3
```

Auswertungsreihenfolge

die arithmetischen Operatoren stärker als der Vergleichsoperator und werden darum vorrangig angewendet. Der Vergleich „trifft zu“, deshalb ist das Ergebnis **true**:

```
2 + 3 < 2*3 → 5 < 6 → true
```

6 Gelegentlich ist man versucht, die Zahl 0 wie den Wert false zu behandeln. true entspräche dann einer 1 oder einer anderen, von Null verschiedenen Zahl. Diese Deutung mag auf elektrotechnischer Ebene noch einen Sinn ergeben, aus programmlogischer Sicht entbehrt sie aber jeder Grundlage.

Logische Operatoren

Logische Operatoren verknüpfen Wahrheitswerte

Neben den arithmetischen und relationalen Operatoren gibt es **logische Operatoren**. Diese verknüpfen Wahrheitswerte und liefern als Ergebnis einen neuen Wahrheitswert:

&&	And, logisches Und: Liefert **true** genau dann, wenn alle beiden Operanden **true** sind.
||	Or, inklusives logisches Oder: Liefert **true** genau dann, wenn wenigstens ein Operand **true** ist.
^	Xor, exklusives logisches Oder: Liefert **true**, wenn genau einer der Operanden **true** ist.
!	Not, logisches Nicht: Liefert **true** genau dann, wenn der Operand **false** ist.

Zusammengesetzte Bedingungen

Mit den logischen Operatoren lassen sich zusammengesetzte Bedingungen formulieren, wie zum Beispiel „x ist größer als -5 und x ist kleiner als $+5$“:[7]

```
(x > -5)  && (x < 5)
```

Die tiefe **if**-Kaskade auf Seite 74 zur Berechnung der Tage eines Monats lässt sich damit viel kürzer fassen:

```
if((month == 4) || (month == 6) || (month == 9) || (month == 11))
    days = 30;
else
    if(month == 2)
        days = 28;
    else
        days = 31;
```

Priorität und Assoziativität

Die binären logischen Operatoren (**&&**, **||**, **^**) binden schwächer als die arithmetischen und relationalen Operatoren. Wie diese sind sie links-assoziativ.

Das logische Not (**!**) ist ein unärer Operator und bindet sehr stark, ebenso wie die unären arithmetischen Vorzeichenoperatoren. Einzelheiten finden Sie in der kompletten Operatorentabelle in Anhang B.

Wahrheitstabellen

Operanden für binäre logische Operatoren

Die binären logischen Operatoren lassen sich mit Wahrheitstabellen vollständig definieren. Das ist praktikabel, weil es für zwei **boolean**-Operanden überhaupt nur vier Wertekombinationen gibt:

```
true,   true
true,   false
false,  true
false,  false
```

7 Es gibt keinen „Bereichsoperator“ für eine Bedingung der Art „x liegt zwischen -5 und $+5$“.

Tabellen für And, Or, Xor

Ein beliebiger, binärer logischer Operator ordnet jeder dieser vier Wertekombinationen einen der Ergebniswerte **true** und **false** zu. Die Wahrheitstabelle für And lautet zum Beispiel:

```
true  && true  → true
true  && false → false
false && true  → false
false && false → false
```

Entsprechend definieren die folgenden Wahrheitstabellen Or:

```
true  || true  → true
true  || false → true
false || true  → true
false || false → false
```

und Xor:

```
true  ^ true  → false
true  ^ false → true
false ^ true  → true
false ^ false → false
```

16 verschiedene binäre logische Operatoren

Insgesamt gibt es überhaupt nur $2^4 = 16$ unterschiedliche Tabellen und dementsprechend auch nur 16 verschiedene binäre logische Operatoren. And, Or und Xor sind drei davon. Unter den verbleibenden 13 finden sich zum Beispiel die in der Elektronik bekannten Nand- und Nor-Verknüpfungen.

Operatorgruppen

Bisher wurden drei Operatorgruppen eingeführt. Die Gruppen unterscheiden sich in den Typen der Operanden und in den Typen der Ergebnisse:

Arithmetische Operatoren: numerisch → numerisch

```
+ - * / %
```

Relationale Operatoren: numerisch → boolean

```
< > <= => == !=
```

Logische Operatoren: boolean → boolean

```
&& || ^ !
```

Eine weitere Möglichkeit fehlt in dieser Tabelle: Auch **boolean**-Werte lassen sich mit `==` und `!=` auf Gleichheit und Ungleichheit prüfen. Die anderen vier relationalen Operatoren (`<`, `>`, `<=`, `>=`) sind für Wahrheitswerte nicht definiert, weil **true** und **false** keiner Reihenfolge unterliegen.

Teilweise und vollständige Auswertung

Abhängige Teilbedingungen

Bei zusammengesetzten Bedingungen hängen die verschiedenen Einzelbedingungen oft voneinander ab. Im Beispiel

```
(d != 0) && (x/d > 0)
```

prüft die linke Teilbedingung `d != 0`, ob die Division in der rechten Teilbedingung `x/d > 0` überhaupt zulässig ist.

Auswertungsschema arithmetischer Operatoren unpassend

Die arithmetischen Operatoren berechnen zuerst beide Operanden und verknüpfen sie anschließend zum Ergebnis. Wenn das logische And ebenso vorgehen würde, dann würde das Programm für `d = 0` bereits beim Versuch den rechten Operanden auszurechnen durch null teilen und mit einem Fehler abbrechen. Das widerspricht der Intuition: Für `d = 0` sollte die Gesamtbedingung

```
(d != 0) && (x/d > 0)
```

nicht zutreffen, weil schon die linke Teilbedingung nicht gilt.

Teilweise Auswertung logischer Operatoren

Um dieser Vorstellung entgegenzukommen, arbeitet das logische And anders als die arithmetischen und relationalen Operatoren: Nur dann, wenn der linke Operand `true` liefert, wird der rechte Operand ausgewertet und schließlich verknüpft. Andernfalls ist das Ergebnis `false`, ohne dass der rechte Operand überhaupt betrachtet wird. Diese Arbeitsweise wird als **teilweise Auswertung** bezeichnet. Allgemein wird bei teilweiser Auswertung zunächst nur ein Teil der Operanden berechnet. Je nach Ergebnis werden dann die übrigen Operanden ignoriert oder ebenfalls ausgewertet.

Der Unterschied zwischen vollständiger und teilweiser Auswertung wird deutlich, wenn man die Operanden des logischen And in temporären `boolean`-Variablen zwischenspeichert. Bei vollständiger Auswertung werden erst *beide* Operanden berechnet und dann getestet:

```
boolean left = (d != 0);
boolean right = (x/d > 0);
if(left) {
   if(right)
      ...
}
```

Bei teilweiser Auswertung wird erst ein Operand berechnet und getestet, der zweite nur falls nötig:

```
boolean left = (d != 0);
if(left) {
   boolean right = (x/d > 0);
   if(right)
      ...
}
```

Auswertung bricht ab, sobald Ergebnis bekannt

Teilweise Auswertung ist bei And sinnvoll, wie die letzten beiden Zeilen der Wahrheitstabelle zeigen. Wenn der linke Operand **false** ist, ist das Ergebnis immer **false**. Der Wert des rechten Operanden spielt dabei keine Rolle:

```
true  && true  → true
true  && false → false
false && true  → false
false && false → false
```

Ähnlich liegen die Verhältnisse bei Or: Wenn der linke Operand **true** ist, ist das Ergebnis **true**, unabhängig vom rechten Operanden, wie man den ersten beiden Zeilen der Wahrheitstabelle entnehmen kann:

```
true  || true  → true
true  || false → true
false || true  → true
false || false → false
```

Keine teilweise Auswertung für Xor

Im Gegensatz dazu ist bei Xor das Endergebnis auch mit Kenntnis des linken Operanden immer noch offen: Der rechte Operand muss in jedem Fall ausgewertet werden, teilweise Auswertung reicht nicht aus:

```
true  ^ true  → false
true  ^ false → true
false ^ true  → true
false ^ false → false
```

Teilweise Auswertung bei drei Operatoren

In Java werden nur And, Or und der „bedingte Operator“ (Seite 86) teilweise ausgewertet. Alle anderen Operatoren werden vollständig ausgewertet.

boolean-Variablen

boolean-Variablen

Variablen können von jedem Typ definiert werden, einschließlich **boolean**. Eine **boolean**-Variable kann die beiden Werte **true** und **false** speichern, wie im folgenden Beispiel:

```
boolean isOk;
isOk = true;
```

Die beiden Anweisungen lassen sich in einer Definition mit Initialisierung zusammenfassen:

```
boolean isOk = true;
```

Wertzuweisung von Bedingungen

Logische Ausdrücke können nicht nur in Bedingungen verwendet, sondern zum Beispiel auch an **boolean**-Variablen zugewiesen werden:

```
boolean validTemperature = celsius > -273.16;
boolean percentageOutOfRange = (percent < 0) || (percent > 100);
```

boolean-Variablen in Bedingungen

boolean-Variablen sind ihrerseits (sehr einfache) logische Ausdrücke und können, ohne Vergleich, in Bedingungen eingesetzt werden:

```
if(percentageOutOfRange)
   System.out.println("cannot  continue ...");
```

3.3 Schleifen

Wiederholen von Anweisungen

Schleifen sind Kontrollstrukturen zur wiederholten Ausführung von Anweisungen. Eine einfache Schleifenkonstruktion ist die **while-Schleife** mit folgendem Aufbau:

```
while ( condition )
  statement
```

boolean-Ausdruck steuert Ablauf

Wie bei der **if**-Anweisung ist *condition* ein **boolean**-Ausdruck und *statement* eine beliebige Anweisung. Diese Anweisung wird auch als „Schleifenrumpf" bezeichnet, der Vorspann als „Schleifenkopf".

Eine **while**-Schleife arbeitet folgendermaßen: Zuerst wird die Bedingung geprüft. Trifft sie zu, wird der Schleifenrumpf einmal ausgeführt. Dann folgt ein neuer Test der Bedingung. Trifft die Bedingung schließlich nicht mehr zu, wird der Rumpf übersprungen und die ganze Schleife beendet.

Beispiel: Wertebereich abzählen

Durchlauf Zahlenbereich

Eine der häufigsten Anwendungen von Schleifen ist der schrittweise Durchlauf eines Wertebereiches, beginnend mit einem Startwert und bis zu einem Endwert. Dafür wird eine Hilfsvariable definiert, der „Schleifenzähler". Vor der Schleife wird der Zähler auf einen Startwert gesetzt und dann im Rumpf erhöht. Im Kopf wird geprüft, ob der Zähler noch unter dem Endwert liegt:

```
int counter = 0;
while(counter < 10)
   counter = counter + 1;
```

Anzahl Durchläufe

Der Rumpf der Schleife wird 10-mal ausgeführt. Im ersten Durchlauf gilt **counter = 0**, im letzten **counter = 9**. Für **counter = 10** wird der Schleifenrumpf nicht mehr durchlaufen, weil im Kopf der Vergleich **10 < 10** nicht zutrifft und die ganze Schleife damit beendet ist.

Wie bei **if**-Anweisungen können mehrere Anweisungen in den Rumpf einer Schleife gestellt werden, wenn sie als Block zusammengefasst sind:

```
int counter = 0;
while(counter < 10) {
   System.out.println(counter);
   counter = counter + 1;
}
```

Diese Schleife gibt nacheinander die Werte $0, 1, 2, \ldots, 9$ aus. Die Reihenfolge der beiden Anweisungen im Rumpf ist entscheidend. Tauscht man sie, erhält man die Ausgabe $1, 2, 3, \ldots, 10$.

Reihenfolge von Anweisungen

Beispiel: Zahlensumme

Teile des ersten Beispielprogramms auf Seite 21 werden jetzt klarer. Die Bedingung **i <= n** im Schleifenkopf sorgt dafür, dass der Rumpf mit dem Wert **n** ein letztes Mal durchlaufen wird, sodass alle Zahlen bis einschließlich **n** aufsummiert werden.

Rückblende zum Einführungsbeispiel

Beispiel: Größter gemeinsamer Teiler (Differenzalgorithmus)

Ein anderes Beispiel für den Einsatz einer Schleife ist die Berechnung des größten gemeinsamen Teilers („ggT“) von zwei gegebenen, positiven[8] ganzen Zahlen *m* und *n*. Dafür gibt es verschiedene Algorithmen, unter denen der sogenannte „Differenzalgorithmus“ zwar einfach, allerdings auch ineffizient ist.

Der Algorithmus selbst ist simpel: Die kleinere der beiden Zahlen *m* und *n* wird von der größeren subtrahiert. Das wird so lange fortgesetzt, bis zwei gleiche Werte übrig bleiben. Das ist der ggT der ursprünglichen Zahlen.

Simpler Algorithmus für ggT

```
class DifferenceGCD {
   public static void main(String[ ] args) {
      int m = Integer.parseInt(args[0]);
      int n = Integer.parseInt(args[1]);
      while(m != n)
         if(m > n)
            m = m - n;
         else
            n = n - m;
      System.out.println(m);
   }
}
```

Beispiel: Größter gemeinsamer Teiler (Euklids Algorithmus)

Ein wesentlich schnellerer Algorithmus zur Berechnung des ggT ist „Euklids Algorithmus“:[9] Die größere der beiden Zahlen *m* und *n* wird durch die kleinere geteilt. Wenn die Division aufgeht, ist die kleinere Zahl der ggT und der Algorithmus endet. Andernfalls werden *m* und *n* durch Divisor und Divisionsrest ersetzt und das Verfahren wird wiederholt.

Effizienter Algorithmus für ggT

```
class EuclidGCD {
   public static void main(String[ ] args) {
      int m = Integer.parseInt(args[0]);
```

8 Negative Startwerte werden in dieser einfachen Fassung nicht berücksichtigt.
9 Dieser zählt zu den ältesten bekannten Algorithmen überhaupt.

```
        int n = Integer.parseInt(args[1]);
        int r = m%n;
        while(r > 0) {
            m = n;
            n = r;
            r = m%n;
        }
        System.out.println(n);
    }
}
```

Es spielt kein Rolle, ob beim Start des Algorithmus m oder n die größere Zahl ist. Die erste Modulus-Operation bringt sie in die richtige Reihenfolge.

Beispiel: Collatzfolge

Zahlenfolge mit interessanten Eigenschaften

Die Collatzfolge oder auch „$3n+1$“-Folge ist eine Zahlenfolge mit interessanten Eigenschaften.

Zunächst wird eine beliebige positive Startzahl z_0 gewählt. Der Rest der Folge ergibt sich nach folgenden Regeln:

- Für ein gerades z_n ist das nächste Element der Folge halb so groß: $z_{n+1} = \frac{1}{2} z_n$.
- Für ein ungerades z_n ist das nächste Element $z_{n+1} = 3z_n + 1$. Diese zweite Formel gibt der Folge den Namen.

Zyklus als Ende

Aus der Startzahl $z_0 = 6$ ergibt sich zum Beispiel die Folge

$$6,\ 3,\ 10,\ 5,\ 16,\ 8,\ 4,\ 2,\ 1,\ 4,\ 2,\ 1,\ \ldots$$

Die drei Zahlen 4, 2, 1 bilden offenbar einen Zyklus, der sich endlos wiederholt.

Empirische Beobachtungen

Inzwischen wurden viele Startzahlen ausprobiert. Dabei hat sich herausgestellt, dass es anscheinend nur den einen Zyklus 4, 2, 1 gibt, in den über kurz oder lang jede Folge mündet. Einen mathematischen Beweis dafür gibt es aber bisher nicht.

Das folgende Programm berechnet die $3n+1$-Folge für eine gegebene Startzahl, bis zum ersten Mal eine 1 auftaucht.

```
class Collatz {
    public staticvoid main(String[ ] args) {
        int n = Integer.parseInt(args[0]);
        while(n != 1) {          // weiter, solange keine 1
            if(n%2 == 0)         // ist n gerade?
                n = n/2;         // ja: halbieren
            else
                n = 3*n + 1;     // nein
            System.out.println(n);
        }
    }
}
```

Eigenschaften der Collatzfolge

Länge einer Folge

Als „Länge“ der Collatzfolge mit Startzahl n soll die Anzahl der Elemente von der Startzahl (einschließlich) bis zur ersten 1 (ausschließlich) bezeichnet werden. Die Folge mit $n = 6$ hat zum Beispiel die Länge 8. Um die Länge einer Folge zu berechnen, wird ein Zähler **counter** verwendet, der im Schleifenrumpf hochgezählt wird:

```
class CollatzLength {
    public static void main(String[ ] args) {
        int n = Integer.parseInt(args[0]);
        int counter = 0;                    // Zähler initialisieren
        while(n != 1) {
            if(n%2 == 0)
                n = n/2;
            else
                n = 3*n + 1;
            counter = counter + 1;          // Zähler erhöhen
        }
        System.out.println(counter);        // Zähler ausgeben
    }
}
```

Größtes Element einer Folge

Das Programm soll noch einmal erweitert werden, um das größte Element („Maximum“) einer Folge zu bestimmen und auszugeben. Die Folge mit $n = 6$ hat das Maximum 16.

Zu diesem Zweck wird eine weitere Variable **max** eingeführt, deren Wert im Programmablauf das bisher größte Element ist. Initialisiert wird **max** mit der Startzahl selbst. Immer wenn ein neu berechnetes Element größer als der bisherige Rekordhalter **max** ist, wird dieser aktualisiert. Am Ende enthält **max** das größte Element der Folge.

```
class CollatzMax {
    public static void main(String[ ] args) {
        int n = Integer.parseInt(args[0]);
        int counter = 0;
        int max = n;                        // bisher größtes Element
        while(n != 1) {
            if(n%2 == 0)
                n = n/2;
            else
                n = 3*n + 1;
            counter = counter + 1;
            if(n > max)                     // neuer Rekord?
                max = n;                    // ja: aufzeichnen
        }
        System.out.println(counter);
        System.out.println(max);            // größtes Element ausgeben
    }
}
```

Die Startzahl $n = 27$ produziert eine besonders lange Folge mit 111 Elementen und dem Maximum 9232.

Operatorzuweisungen

Syntaktische Kurzform

In vielen Programmen kommen Wertzuweisungen der folgenden Gestalt vor:

variable = variable operator expression;

variable ist dabei links und rechts des Zuweisungszeichens dieselbe Variable, *operator* ein binärer Operator und *expression* ein beliebiger Ausdruck. Diese Form der Zuweisung lässt sich kürzer schreiben als **Operatorzuweisung**:

variable operator = expression;

Eine Operatorzuweisung ist eine syntaktische Kurzform, die Schreibarbeit spart und die Leserlichkeit verbessert. Die Sprache Java wird damit aber nicht ausdrucksfähiger.

Im vorhergehenden Beispielprogramm kann die Operatorzuweisung an zwei Stellen eingesetzt werden:

```
...
while(n != 1) {
    if (n%2 == 0)
        n /= 2;          // vorher n = n/2;
    else
        n = 3*n + 1;
    counter += 1;        // vorher counter = counter + 1;
    if (n > max)
        max = n;
}
...
```

Die Wertzuweisung **n = 3*n + 1;** lässt sich nicht als Operatorzuweisung formulieren, weil sie nicht die Form *variable* **=** *variable operator expression***;** hat.

Inkrement- und Dekrementoperator

Zählen in Einerschritten

In Schleifen werden Variablen oft in Einerschritten nach oben oder nach unten gezählt. Mit Operatorzuweisungen kann das folgendermaßen geschrieben werden:

***variable* += 1;**
***variable* –= 1;**

Speziell für dieses schrittweise Zählen gibt es eine weitere Kurzform mit dem **Inkrementoperator** ++ beziehungsweise dem **Dekrementoperator** **--**:

***variable*++;**
***variable*– –;**

Die folgenden drei Anweisungen sind äquivalent:[10]

```
variable = variable + 1;
variable += 1;
variable++;
```

Im vorhergehenden Beispielprogramm kann der Inkrementoperator verwendet werden, um den Zähler hochzuzählen:

```
...
while(n != 1) {
    if (n%2 == 0)
        n /= 2;
    else
        n = 3*n + 1;
    counter++;          // vorher counter += 1;
    if (n > max)
        max = n;
}
...
```

Inkrementoperator als Anweisung oder Ausdruck

Die Konstruktion **variable**++ spielt in einer weiteren Hinsicht eine Zwitterrolle: Sie kann einerseits als Anweisung benutzt werden, wie vorher gezeigt. Sie kann aber auch als Ausdruck verwendet werden und liefert dann den Wert der Variablen *vor* dem Inkrementieren. Das Entsprechende gilt für **variable--**. Im folgenden Beispiel hat **a** am Ende den Wert **2** und **b** den Wert **1**:

```
int a = 1;
int b = a++;
```

Prä- und Postfixstellung

Zudem können die beiden Operatoren ++ und -- vor und nach der Variablen stehen.

- Der **Präfixoperator** inkrementiert (dekrementiert) *zuerst* und liefert *dann* den Wert.
- Der **Postfixoperator** liefert *zuerst* den Wert und inkrementiert (dekrementiert) *dann*.

Das folgende Beispiel zeigt alle vier Möglichkeiten (die Variablen **a**, **b**, **c** und **d** sind mit **1** initialisiert):

```
int e = ++a;        // e = 2, a = 2
int f = b++;        // f = 1, b = 2
int g = --c;        // g = 0, c = 0
int h = d--;        // h = 1, d = 0
```

10 Tatsächlich gibt es subtile Unterschiede im Zusammenhang mit impliziten Typkonversionen: Für Variablen der Typen byte, short und char ist die erste Form nicht zulässig, die zweite und dritte dagegen schon.

Verschachtelte Ausdrücke mit mehreren Inkrement- oder Dekrementoperatoren sind oft schwer zu durchschauen und schlecht lesbar, wie das folgende Negativbeispiel zeigt:

```
int a = 1;
int b = a++ + ++a;  // b = ?
```

Der Gewinn an Schreibersparnis ist dagegen meist vernachlässigbar. Aus diesem Grund sollten die beiden Operatoren nicht in zusammengesetzten Ausdrücken benutzt werden.

Bedingter Operator

Dreistelliger Operator

Ein interessanter Operator ist der dreistellige „bedingte Operator" (*conditional operator*) mit der Schreibweise

```
condition ? expression : expression
```

Die beiden Zeichen **?** und **:** trennen die drei Operanden.

Zuerst wird der erste Operand, der **boolean**-Ausdruck *condition*, ausgewertet. Wenn er **true** liefert, wird der mittlere Operand ausgewertet und liefert das Ergebnis. Andernfalls liefert der rechte Operand das Ergebnis.

Um die Bestandteile optisch ganz eindeutig zu trennen, wird die Bedingung oft in Klammern gesetzt. Formal wäre das meistens nicht nötig, weil der bedingte Operator eine verhältnismäßig niedere Priorität hat. Im folgenden Programmfragment wird beispielsweise **b = 1** zugewiesen:

```
int a = 5;
int b = (a > 0)? 1: 2;
```

In jedem Fall wird genau einer der beiden hinteren Operanden ausgewertet und der jeweils andere nicht. Beide müssen den gleichen Typ haben.[11]

Ähnlichkeit mit if-Anweisung

Der Ablauf ähnelt einer zweiseitigen **if**-Anweisung. Der bedingte Operator bildet aber einen Ausdruck und liefert einen Wert, während eine **if**-Anweisung keinen Wert hat. Die Zuweisung

```
variable = condition? expression1: expression2;
```

ist äquivalent zu:

```
if(condition)
    variable = expression1;
else
    variable = expression2;
```

11 Genauer gesagt: sie müssen beide kompatibel zum Kontext des Ausdrucks sein. So ist zum Beispiel die Zuweisung double d = condition? 2: 2.5; zulässig.

Teilweise Auswertung

Ebenso wie die logischen Operatoren And und Or wertet der bedingte Operator teilweise aus: Die drei Operanden werden nicht alle drei vorher berechnet, sondern nur zwei davon nach Bedarf.

Die frühere Fassung des Programms von Seite 85 lässt sich mit dem bedingten Operator noch kompakter schreiben:

```
...
while(n != 1) {
   n = (n%2 == 0)?  n/2:  3*n + 1;
   counter++;
   max = (n > max)?  n:  max;
}
...
```

Der zweite der beiden bedingten Operatoren in diesem Beispiel schießt aber sicher über das Ziel hinaus. Die ursprüngliche Fassung mit einer **if**-Anweisung war leichter lesbar.

Geschachtelte Schleifen

Schleife kontrolliert andere Schleife

Eine **while**-Schleife ist als Ganzes selbst eine zusammengesetzte Anweisung. Sie kann damit als Baustein in anderen Kontrollstrukturen eingesetzt werden. Insbesondere lassen sich Schleifen schachteln. Das folgende Programm druckt eine 1×1-Tabelle aus:

```
01 class MultTable {
02    public static void main(String[ ] args) {
03       int x = 1;
04       int y;
05       while(x <= 10) { // äußere Schleife
06          y = 1;
07          while(y <= 10) { // innere Schleife
08             System.out.printf("%4d", x*y);
09             y++;
10          }
11          System.out.println();
12          x++;
13       }
14    }
15 }
```

Äußere und innere Schleife

Die beiden Schleifen werden als „äußere“ und „innere“ Schleife bezeichnet.

Der Rumpf der äußeren Schleife (Zeilen 6–12) wird 10-mal durchlaufen mit den Werten **x** = **1**, **2**, **3**, . . . , **10**.

Für jeden einzelnen Durchgang der äußeren Schleife wird der Rumpf der inneren Schleife (Zeilen 8 und 9) wiederum 10-mal durchlaufen mit den Werten **y = 1**, **2**, **3**, . . . , **10**. Insgesamt wird die Ausgabeanweisung **printf** also $10 \cdot 10$-mal = 100-mal ausgeführt.

Hotspots

Häufig ausgeführte Anweisungen werden als *Hotspots* eines Programms bezeichnet. An Hotspots wirken sich selbst geringe Performance-Unterschiede oft spürbar auf das Laufzeitverhalten des ganzen Programms aus. Nicht ganz zu Unrecht wird deshalb manchmal etwas plakativ behauptet, dass 90% der Laufzeit eines Programms von 10% des Codes verbraucht werden.

3.4 break und continue

Unterbrechung als einfache Anweisung

break und **continue** sind Anweisungen, die nur im Rumpf von Schleifen[12] verwendet werden dürfen. Beide sind eigenständige, elementare Anweisungen und stehen auf der gleichen Stufe wie Definitionen, Wertzuweisungen und Ausgabeanweisungen.

Schleifenabbruch mit break

Abbruch einer Schleife als Ganzes

Sobald innerhalb eines Schleifenrumpfes die Anweisung

```
break;
```

erreicht wird, wird die Schleife sofort beendet. Der Rest des Rumpfes wird nicht mehr ausgeführt, die Bedingung im Schleifenkopf nicht mehr geprüft und das Programm nach der Schleife fortgesetzt.

Vereinfachung mancher Schleifenkonstruktionen

Eine **break**-Anweisung ist manchmal nützlich, um umständliche Konstruktionen zu vereinfachen. Im folgenden Beispiel wird eine unbekannte Anzahl von Eingabewerten verarbeitet, bis eine Null erscheint und das Ende markiert.

```
while(true) {
   int n;
   // n ermitteln ...
   if(n == 0)
      break;
   // n verarbeiten ...
}
```

Ein anderes Beispiel ist Euklids Algorithmus zur Berechnung des ggT auf Seite 81, der zunächst folgendermaßen implementiert wurde:

```
...
int r = m%n;
while(r > 0) {
   m = n;
   n = r;
   r = m%n;
}
...
```

12 Eine Ausnahme gibt es: break ist auch in switch-Anweisungen zulässig.

Die Berechnung des Divisionsrestes kommt in zwei Kopien vor. Mit einer **break**-Anweisung lässt sich das vermeiden:

```
...
while(true) {
   int r = m%n;
   if(r == 0)
      break;
   m = n;
   n = r;
}
...
```

In diesem Beispiel ist die Rechenzeit-Einsparung sicher vernachlässigbar. Bei größeren Programmen können aber solche Vereinfachungen durchaus lohnenswert sein.

Im letzten Beispiel lautet die Schleifenbedingung einfach **true**. Eigentlich wäre das eine Endlosschleife, die in Wahrheit aber mit **break** beendet wird. Auch diese, bei flüchtiger Betrachtung irreführende Konstruktion weist darauf hin, dass **break** zurückhaltend eingesetzt werden sollte.

Schleifenkurzschluss mit continue

Abbruch eines Schleifendurchgangs

Ähnlich wie **break** unterbricht **continue** den regulären Ablauf von Schleifen: Sobald ein **continue** erreicht wird, wird der Rest des Schleifenrumpfes übersprungen und dann mit der normalen Abwicklung der Schleife fortgefahren. Die Schleife wird hier also nicht abgebrochen, sondern vorzeitig der nächste Durchgang gestartet.

Auch mit **continue** können Schleifenkonstruktionen gekürzt und vereinfacht werden. Im folgenden Beispiel wird eine Folge von Eingabewerten verarbeitet. Negative Werte sollen dabei ignoriert werden:

```
while(true) {
   int n;
   // n ermitteln ...
   if(n < 0)
      continue;
   // n verarbeiten ...
}
```

Syntaktische Kurzform für längere Konstruktion

break und **continue** können immer durch **if**-Anweisungen ersetzt werden. Nur wenn die **if**-Anweisungen recht lang und tief verschachtelt werden, schaffen **break** oder **continue** mehr Klarheit.

Ein Sonderfall sind *foreach*-Schleifen (Abschnitt 6.4 und 12.5): Dort bietet **break** die einzige Möglichkeit, um den regulären Schleifenablauf vorzeitig zu beenden, weil es keine explizite Schleifenbedingung gibt.

3.5 Gültigkeitsbereiche

Zusätzliche Eigenschaft eines Blocks

Wie auf Seite 72 beschrieben, können Folgen beliebig vieler Anweisungen mit geschweiften Klammern zu einem **Block** zusammengefasst und dieser als eine einzelne Anweisung behandelt werden. Die Folgen dieser Konstruktion gehen tiefer, als man auf den ersten Blick vermuten würde.

Variablendefinition endet mit Block

In einem Block sind alle Arten von Anweisungen erlaubt, und damit auch Variablendefinitionen. Eine Variablendefinition gilt ab dem Ort der Definition bis zum Ende des Blocks, in dem sie steht. Außerhalb dieses Blocks gilt die Definition dagegen nicht mehr.

Der Quelltextausschnitt, in dem eine Definition gilt, wird als **Gültigkeitsbereich** (*scope*) bezeichnet. Das folgende Beispielprogramm berechnet die Summe von Quadratzahlen. Dabei wird jedes einzelne Quadrat in der Hilfsvariablen **square** zwischengespeichert.[13]

```
01 class SquareSum
02 {
03    public static void main(String[ ] args)
04    {
05       int n = 10;
06       int sum = 0;
07       int loop = 0;
08       while(loop < n)
09       {
10          loop++;
11          int square = loop*loop;
12          sum += square;
13       }
14       System.out.println(sum);
15    }
16 }
```

Der Gültigkeitsbereich der Variablen **loop** beginnt in diesem Beispiel mit der Definition in Zeile 7 und endet mit Zeile 15. Demgegenüber ist **square** nur in den Zeilen 11–13 gültig. In Zeile 13 endet der Block, in dem **square** definiert ist, und damit auch der Gültigkeitsbereich der Variablen.

Gültigkeitsbereich bezogen auf Quelltext

Ein Gültigkeitsbereich ist ein Ausschnitt des Quelltextes. Bildlich gesprochen könnte ein Gültigkeitsbereich im Quelltext mit einem Stift markiert werden.

Gültigkeitsbereiche werden vom Compiler überprüft und ausgewertet. Im fertig übersetzten und ausführbaren Bytecode spielen sie keine Rolle mehr.

Namenskonflikte

Namenskollision von Definitionen

Im Gültigkeitsbereich einer Variablen können untergeordnete Blöcke eingebettet sein. Diese zählen mit zum Gültigkeitsbereich. Im vorhergehenden Programm

13 Die Variable square ist eigentlich unnötig. Sie ist hier nur zur Illustration eingefügt.

wird das stillschweigend unterstellt, weil zum Beispiel in Zeile 11 die Variable **loop** verwendet wird, obwohl diese außerhalb des unmittelbar umgebenden Blocks definiert ist.

Das zieht die Gefahr von Namenskonflikten nach sich. Angenommen, die Variable **square** im vorhergehenden Beispiel würde in **n** umbenannt:

```
int n = 10;
int sum = 0;
int loop = 0;
while(loop < n) {
   loop++;
   int n = loop*loop;  // Fehler - n doppelt definiert
   sum += n;
}
```

Mehrfache Definitionen verboten

Einerseits liegt die kommentierte Zeile im Gültigkeitsbereich der äußeren Definition von **n**, andererseits wird eine neue Variable mit dem gleichen Namen **n** definiert. Java verbietet das: An jedem Punkt im Code darf nur eine einzige Definition einer Variablen gelten.[14]

Keine Kollision in getrennten Gültigkeitsbereichen

Definitionen gleichnamiger Variablen in getrennten Gültigkeitsbereichen sind kein Problem. Die beiden Variablen **tmp** im folgenden Beispiel haben „zufällig" den gleichen Namen, sind aber ansonsten völlig unabhängig voneinander:

```
while(...) {
   int tmp;
   ...
}
while(...) {
   int tmp;
   ...
}
```

Lebensdauer

Lebensdauer als Zeitintervall

Variablen kann eine **Lebensdauer** zugesprochen werden, die mit dem Gültigkeitsbereich nichts zu tun hat. Die Lebensdauer einer Variablen ist das Zeitintervall, in dem die Variable zur Laufzeit existiert.

Inkarnationen von Variablen

Eine Variable, die in einem Schleifenrumpf definiert ist, wird beim Ablauf des Programms möglicherweise vielfach geschaffen und wieder freigegeben. Im Programm **SquareSum** auf Seite 90 gibt es zehn aufeinanderfolgende „Inkarnationen" von **square**, eine in jedem Schleifendurchlauf.

14 Hier unterscheidet sich Java beispielsweise von C und C++, wo eine zweite Definition zulässig ist und die erste Definition „überschattet". Die Regelung von Java ist vielleicht weniger flexibel, verringert aber die Gefahr von Missverständnissen.
Allerdings gibt es in Java verschiedene Arten von Variablen, die sich zum Teil auch mit gleichen Namen nicht ins Gehege kommen. An dieser Stelle geht es aber nur um lokale Variablen, sodass diese Betrachtung hier noch keine Rolle spielt.

Inkarnationen isoliert

Die verschiedenen Inkarnationen leben völlig isoliert voneinander. **square** wird jedes Mal komplett neu erzeugt. Es gibt keine Möglichkeit, auf den Wert einer früheren Inkarnation zuzugreifen.

Das Entstehen und Auflösen von Variablen führt zu keinem nennenswerten Performanceverlust, weil dieser Vorgang im Laufzeitsystem äußerst effizient abgewickelt wird und in der Praxis kaum messbar ist.

3.6 for-Schleifen

for-Schleifen für Bereichsdurchlauf

In vielen Schleifen wird mit einem Zähler ein Wertebereich durchlaufen. Für diese Anwendung gibt es die **for-Schleifen**. Aufbau und Arbeitsweise von **for**-Schleifen wirken auf den ersten Blick etwas kompliziert. Dennoch eignen sie sich für viele Zwecke gut wegen der kompakten Schreibweise.

Schematisch sehen **for**-Schleifen folgendermaßen aus:

```
for ( startstatement ; condition ; nextstatement )
   statement
```

Komplizierte Semantik

Zu Beginn wird die Startanweisung *startstatement* einmal ausgeführt. Dann wird die Bedingung *condition* geprüft. Falls das Ergebnis **true** ist, wird zuerst *statement* und dann die sogenannte „Fortschalt-Anweisung“ *nextstatement* ausgeführt.[15] Anschließend beginnt der nächste Schleifendurchgang mit dem Test der Bedingung. Die Schleife endet, sobald die Bedingung **false** liefert. Das folgende Codefragment gibt zum Beispiel die Werte **0** bis **9** aus:

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

Beispiele

Zahlensumme mit for-Schleife

Das einführende Beispielprogramm zur Berechnung der Zahlensumme (Seite 21) lässt sich mit einer **for**-Schleife folgendermaßen schreiben:

```
int n = 4;
int sum = 0;
for(int counter = 1;  counter <= n;  counter++)
   sum += counter;
System.out.println(sum);
```

Diese Anwendung ist typisch für **for**-Schleifen: In der Startanweisung wird eine „Laufvariable“ definiert. In der Bedingung wird die Laufvariable mit einem Endwert verglichen. Die Fortschalt-Anweisung gibt der Laufvariablen den nächsten

15 Diese Darstellung ist vereinfacht. *startstatement* und *nextstatement* können nicht nur einzelne Anweisungen, sondern ganze Listen von Anweisungen und Ausdrücken sein, die dann der Reihe nach um ihres Seiteneffekts willen ausgewertet werden. Solche Konstruktionen sind aber kaum jemals notwendig und zudem unübersichtlich.

Wert. Im Schleifenrumpf wird der eigentliche Zweck der Schleife erfüllt und die Laufvariable in irgendeiner Form verwendet.

Geschachtelte for-Schleifen

Eine andere passende Einsatzmöglichkeit für **for**-Schleifen ist die Ausgabe der 1×1-Tabelle (siehe Seite 87):

```
class MultTable {
   public static void main(String[ ] args) {
      for(int x = 1;  x <= 10;  x++) {
         for(int y = 1;  y <= 10;  y++)
            System.out.printf("%4d", x*y);
         System.out.println();
      }
   }
}
```

for-Schleife ohne „Zähler"

Auch Euklids Algorithmus (Seite 81) zur ggT-Berechnung kann mit einer **for**-Schleife formuliert werden:

```
for(int r = m%n;  r != 0;  r = m%n) {
   m = n;
   n = r;
}
```

In diesem Beispiel wird die **for**-Schleife eigentlich zweckentfremdet, weil die Zählvariable hier nichts „zählt".

Das ursprüngliche **Collatz**-Programm (Seite 82) kann mit einer **for**-Schleife neu geschrieben werden:

```
for(int n = Integer.parseInt(args[0]);
      n != 1;
      n = (n%2 == 0)? n/2: 3*n + 1)
   System.out.println(n);
```

Dieses Beispiel ist nur noch schwer lesbar. Eine einfachere und längere Formulierung mit weniger syntaktischen Abkürzungen wäre sicher vorzuziehen.

Gegenüberstellung mit while-Schleifen

for- und while-Schleifen gegenseitig ersetzbar

Jede **for**-Schleife kann durch eine gleichwertige **while**-Schleife ersetzt werden und umgekehrt. Das Codefragment

```
for(startstatement; condition; nextstatement)
   statement
```

ist annähernd äquivalent zu:[16]

16 In der Ersatzdarstellung ist das *nextstatement* Teil des Schleifenrumpfes, in der for-Schleife dagegen nicht. Dementsprechend wirkt ein continue unterschiedlich.

```
{
   startstatement;
   while(condition)
   {
      statement
      nextstatement;
   }
}
```

Gültigkeitsbereich der Zählvariable

Wichtig sind die geschweiften Klammern, die die ganze **while**-Schleife einfassen. In einer **for**-Schleife ist der Gültigkeitsbereich einer Definition im *startstatement* auf die Schleife selbst begrenzt. Diese Regelung erlaubt mehrere aufeinanderfolgende **for**-Schleifen mit gleich benannten Laufvariablen:

```
for(int i = 0; ...)
   ...
for(int i = 0; ...)
   ...
```

Ohne die zusätzlichen Klammern in der Ersatzdarstellung mit **while**-Schleifen wäre das nicht zulässig, weil die Definitionen der Laufvariablen außerhalb der Rümpfe stehen und damit kollidieren würden.

3.7 switch-Anweisungen

Eine **switch-Anweisung** entspricht im Prinzip einer einfachen Art von **if**-Kaskade, in der das Ergebnis eines bestimmten Ausdrucks nacheinander mit einer Liste von festen Werten verglichen wird. Im folgenden Beispiel wird der Name einer Zahl zwischen 1 und 3 ausgegeben:

```
switch(n) {
  case 1:
    System.out.println("one");
    break;
  case 2:
    System.out.println("two");
    break;
  case 3:
    System.out.println("three");
    break;
}
```

Grundsätzlich sind **switch**-Anweisungen folgendermaßen aufgebaut:

```
switch ( expression )
{
   case label :
      statement ...
   case label :
      statement ...
   ...
}
```

Syntaktische Bestandteile

Im Kopf der **switch**-Anweisung steht ein Ausdruck (**expression**), im Rumpf folgt eine Liste von Zweigen (*switch case*). Jeder Zweig beginnt mit einem **case**-Label, gefolgt von Anweisungen.

Abwicklung einer switch-Anweisung

Eine **switch**-Anweisung wird folgendermaßen abgewickelt: Zuerst und nur einmal wird der Ausdruck **expression** im Kopf berechnet. Dann wird das Ergebnis unter den **case**-Labeln gesucht. Schließlich werden die Anweisungen im passenden Zweig ausgeführt, bis eine **break**-Anweisung erreicht wird. Die **break**-Anweisung beendet die ganze **switch**-Anweisung.

if-Kaskaden unübersichtlich

Eine **if**-Kaskade dient als Verteiler auf mehrere verschiedene Ablaufwege. Das Beispiel auf Seite 74 zeigt eine tief verschachtelte **if**-Kaskade zur Berechnung der Tage eines Monats, wobei der Monat als Nummer gegeben ist:

```
if(month == 1)
   days = 31;
else
   if(month == 2)
     days = 28;
   else
     if(month == 3)
        days = 31;
     ...
        else
           if(month == 12)
              days = 31;
```

switch-Anweisung ersetzt if-Kaskaden

Die Konstruktion ist recht sperrig und lädt förmlich zu Fehlern ein. Sie lässt sich mit einer **switch**-Anweisung übersichtlicher schreiben:

```
switch(month) {
  case 1:
    days = 31;
    break;

  case 2:
    days = 28;
    break;

  case 3:
    ...
```

```
  case 12:
    days = 31;
    break;
}
```

case-Label

Eindeutige, konstante case-Label

Die **case**-Label in einer **switch**-Anweisung müssen eindeutig sein. Zudem müssen alle **case**-Label *vom Compiler berechenbar* sein. Es sind also nur konstante Ausdrücke erlaubt, die aus **final**-Variablen und Literalen bestehen.

Wenn keines der **case**-Label passt, wird die ganze **switch**-Anweisung ohne Auswirkung beendet.

Zweig mit mehreren Labeln

Die **case**-Label müssen in keiner bestimmten Reihenfolge genannt werden, abgesehen vom **default**-Label (siehe nächster Abschnitt). Mehrere **case**-Label pro Zweig sind zulässig. Das vorhergehende Beispiel lässt sich wie folgt vereinfachen:

```
switch(month) {
  case 1:
  case 3:
  case 5:
  case 7:
  case 8:
  case 10:
  case 12:
    days = 31;
    break;

  case 2:
    days = 28;
    break;

  case 4:
  case 6:
  case 9:
  case 11:
    days = 30;
    break;
}
```

Defaultzweig

default-Label passt auf jeden Wert

Das besondere Label **default** dient als eine Art Joker. Es darf nur einmal vorkommen, muss dann als Letztes stehen und passt zu jedem vorher nicht genannten Wert.

Das vorhergehende Beispiel lässt sich mit einem Defaultzweig weiter kürzen:

```
switch(month) {
  case 2:
    days = 28;
    break;

  case 4:
  case 6:
  case 9:
  case 11:
    days = 30;
    break;

  default:   // Defaultzweig: alle restlichen Monate
    days = 31;
    break;
}
```

Der Defaultzweig entspricht einem abschließenden **else** in einer **if**-Kaskade.

Jede **switch**-Anweisungen sollte einen Defaultzweig vorsehen, selbst wenn dieser „eigentlich nicht möglich“ ist.

Defaultzweig immer sinnvoll

Fall through

Die Anweisungen eines Zweiges werden der Reihe nach ausgeführt, bis eine **break**-Anweisung oder das Ende der **switch**-Anweisung erreicht wird.

Abgrenzung von Zweigen

In der Regel endet jeder Zweig mit einer **break**-Anweisung. Sollte diese aber fehlen, so wird das nachfolgende **case**-Label übergangen und mit den Anweisungen des *nachfolgenden Zweiges* fortgefahren.[17] Diese Verhalten wird als *Fall through* bezeichnet.

Fehlendes break

Im Beispiel

```
switch(n) {
  case 1:
    System.out.println("one");
    // kein break - fall through
  case 2:
    System.out.println("two");
    // kein break - fall through
  case 3:
    System.out.println("three");
    // kein break - Ende des switch
}
```

werden für $n = 1$ alle drei Ausgaben durchlaufen.

17 Ob dieses Verhalten besonders sinnvoll ist, sei dahingestellt. Wie einige andere Sprachelemente stammt auch die switch-Anweisung aus der älteren Programmiersprache C und bringt eine vielleicht in effizienten Code übersetzbare, aber aus heutiger Sicht dennoch recht barocke Semantik mit.

Fall through selten sinnvoll

Fall through ist selten sinnvoll. In der Regel endet *jeder* Zweig einer **switch**-Anweisung mit einer **break**-Anweisung. Das gilt auch für den letzten Zweig, in dem das **break** technisch gesehen nicht nötig wäre, weil die **switch**-Anweisung anschließend ohnedies beendet ist. Bei späteren Änderungen besteht so weniger Gefahr, dass es zu einem unbeabsichtigten *Fall through* kommt.

In den wenigen Situationen, in denen ein *Fall through* wirklich sinnvoll ist, sollte das unbedingt in einem Kommentar deutlich gemacht und begründet werden.

switch-Anweisung und Blöcke

Definitionen in switch-Anweisungen vermeiden

Der Rumpf einer **switch**-Anweisung begrenzt einen Block. Allerdings hat die etwas eigenartige Abwicklung von **switch**-Anweisungen unübersichtliche Folgen für die Gültigkeitsbereiche von Variablen. Im Rumpf von **switch**-Anweisungen sollte man deshalb *besser keine* Variablen definieren.

Geschachtelte switch-Anweisungen

Eine **switch**-Anweisung ist selbst eine zusammengesetzte Anweisung. Damit können **switch**-Anweisungen geschachtelt werden. Sinnvoll ist das, um zum Beispiel Tabellen in Code umzusetzen. Angenommen, eine Funktion $f(x, y)$ sei durch folgende Tabelle definiert:

	x = 0	*x = 1*	*x = 2*
y = 0	4	Undefiniert	5
y = 1	5	7	6
y = 2	Undefiniert	5	8

Die Tabelleneinträge folgen keiner einfachen Rechenvorschrift. Das nächste Programmfragment gibt in der Variablen **isDefined** Auskunft, ob die Funktion überhaupt definiert ist und schreibt gegebenenfalls den Funktionswert in die Variable **value**. Die Defaultfälle sind hier weggelassen.

```
boolean isDefined = true;
int value = -1;
switch(x) {
  case 0:
    switch(y) {
      case 0:  value = 4;  break;
      case 1:  value = 5;  break;
      case 2:  isDefined = false; break;
    }
  case 1:
    switch(y) {
      case 0:  isDefined = false; break;
      case 1:  value = 6;  break;
      case 2:  value = 5;  break;
    }
  case 2:
    switch(y) {
      case 0:  value = 5;  break;
```

```
    case 1:  value = 6;  break;
    case 2:  value = 8;  break;
  }
}
```

Diese Konstruktion kann sicher nicht als elegant bezeichnet werden. Der sehr regelmäßige Aufbau erlaubt aber einen direkten Vergleich mit der Tabelle und vereinfacht zum Beispiel die Fehlersuche. Eine entsprechende **if**-Kaskade wäre kaum kürzer, aber sicher schwerer zu überschauen.

switch-Typen

Der Ausdruck im Kopf einer **switch**-Anweisung und die **case**-Label müssen den gleichen Typ haben. Dabei sind **int**, Aufzählungstypen (siehe Abschnitt 4.14) und Strings erlaubt, aber nicht **boolean** oder **double**.[18] Ein Beispiel zum **switch** über Aufzählungswerte finden Sie auf Seite 159.

switch-Typen, Strings

7

ZUSAMMENFASSUNG

In diesem Kapitel haben Sie gelernt:

- Wie mit if-Anweisungen einzelne Programmteile durchlaufen oder weggelassen werden können
- Dass Bedingungen, die if-Anweisungen steuern, nichts anderes als Ausdrücke sind, deren Ergebnis ein Wahrheitswert (Typ boolean) ist
- Wie sich einfache Vergleiche mit logischen Operatoren zu komplexen Bedingungen verknüpfen lassen
- Wie mit while-Schleifen Programmabschnitte wiederholt durchlaufen werden können
- Dass sich Kontrollstrukturen beliebig tief verschachteln lassen und damit komplexe Abläufe formuliert werden können
- Wie Schleifen mit break und continue vorzeitig abgebrochen oder kurzgeschlossen werden können
- Dass Variablen innerhalb von Kontrollstrukturen, sozusagen „vor Ort“, definiert werden können und damit der Gültigkeitsbereich verkleinert wird
- Wie mit for-Schleifen Zahlenbereiche durchlaufen werden können
- Dass mit switch-Anweisungen Fallunterscheidungen manchmal übersichtlicher als mit langen if-Kaskaden implementiert werden können

18 Ein switch für die zwei möglichen boolean-Werte wäre keine Vereinfachung. Der Test von Floatingpoint-Werten auf Gleichheit ist wegen Rundungsfehlern problematisch.

Lösung

Übungsaufgaben

3a. Median

Schreiben Sie ein Programm **Median**, das drei Zahlen a, b und c von der Kommandozeile einliest und den mittleren der drei Werte wieder ausgibt.

Hier ist nicht der arithmetische Mittelwert $\frac{a+b+c}{3}$ gemeint. Denken Sie sich a, b und c der Größe nach geordnet. **Median** gibt die Zahl aus, die dann in der Mitte steht.

Ein Beispiel:

```
$ java Median 2 1 2
2
```

Versuchen Sie mit möglichst wenigen Vergleichen auszukommen.

3b. Zahlenvergleiche

Das Programm **Monoton** erhält vier ganze Zahlen auf der Kommandozeile. Das Programm soll einen der drei folgenden Buchstaben ausgeben:

- **S**, wenn jede Zahl echt größer als die vorhergehende Zahl ist (S = streng monoton steigend).
- **M**, wenn jede Zahl größer oder gleich groß wie die vorhergehende Zahl ist (M = monoton steigend).
- **N** ansonsten (N = nicht monoton steigend).

Einige Beispiele:

```
$ java Monoton 0 1 2 4
S
$ java Monoton 1 3 3 4
M
$ java Monoton 1 3 4 1
N
```

Lösen Sie die Aufgabe ohne zusammengesetzte Bedingungen, das heißt ohne logische Operatoren.

3c. Zahlentests

Das Programm **Select23** erhält vier ganze Zahlen auf der Kommandozeile. Das Programm soll **true** ausgeben, wenn ...

- genau eine der Zahlen durch 2 und
- genau eine der Zahlen durch 3 teilbar ist.

Andernfalls soll **false** ausgegeben werden. Einige Beispiele:

```
$ java Select23 4 5 15 7
true
$ java Select23 4 5 5 6
false
$ java Select23 1 6 11 7
true
$ java Select23 15 7 7 5
false
```

Lösen Sie die Aufgabe ohne zusammengesetzte Bedingungen, das heißt ohne logische Operatoren.

3d. Fibonacci-Zahlen

Die „Fibonacci-Reihe“ ist eine Folge von ganzen Zahlen f_0, f_1, f_2, ... Die ersten beiden Fibonacci-Zahlen sind $f_0 = 1$ und $f_1 = 1$. Jede weitere Zahl ergibt sich aus der Summe der beiden Vorgänger:

$$f_n = f_{n-2} + f_{n-1} \quad \text{für} \quad n \geq 2$$

Der Anfang der Fibonacci-Reihe lautet $1, 1, 2, 3, 5, 8, \ldots$

Schreiben Sie ein Programm **Fibonacci**, das eine Zahl $n \geq 0$ akzeptiert und die entsprechende Fibonacci-Zahl f_n berechnet. Ignorieren Sie arithmetischen Überlauf.

Zwei Beispiele:

```
$ java Fibonacci 5
8
$ java Fibonacci 0
1
```

3e. Logische Ausdrücke

Geben Sie für jeden der folgenden umgangssprachlich beschriebenen Ausdrücke einen entsprechenden Java-Ausdruck an. Der Java-Ausdruck soll **true** liefern, wenn die Bedingung zutrifft, und ansonsten **false**. **i**, **j** und **k** sind **int**-Variablen, **b**, **c** und **d** sind **boolean**-Variablen.

Versuchen Sie, möglichst kurze Ausdrücke zu finden.[19]

1. **i**, **j** und **k** sind alle verschieden von null.
2. **i** ist durch 17 teilbar und echt positiv.
3. **j** ist ungerade und liegt zwischen 20 und 40.
4. **k** ist entweder Vielfaches von 3 und 5 oder Vielfaches von 5 und 7 oder Vielfaches von 5 und 11.
5. Genau eines von **b**, **c** und **d** ist **true**.
6. **b**, **c** und **d** sind alle drei **true** oder alle drei **false**.

3f. Kleinstes gemeinsames Vielfaches

Schreiben Sie ein Programm **LCM**, das das „kleinste gemeinsame Vielfache“ von zwei natürlichen Zahlen berechnet und ausgibt. **LCM** erhält die zwei Zahlen auf der Kommandozeile und gibt das kgV aus. Einige Beispiele:

```
$ java LCM 6 9
18
$ java LCM 9 6
18
$ java LCM 11 11
11
$ java LCM 1 17
17
$ java LCM 19 17
323
```

Der ggT und das kgV von zwei Zahlen hängen eng zusammen. Nutzen Sie diese Abhängigkeit.

19 Im Allgemeinen ist das kein erstrebenswertes Ziel. Ein langer, lesbarer Ausdruck ist besser als ein kurzer, schwer verständlicher.

3g. Nand und Nor

Es gibt in Java keine Operatoren für die binären logischen Verknüpfungen Nand (**true**, wenn nicht beide Operanden gleichzeitig **true** sind) und Nor (**true**, wenn keiner der beiden Operanden **true** ist).

- Stellen Sie für Nand und Nor Wahrheitstabellen auf.
- Geben Sie äquivalente Java-Ausdrücke an, die mit der minimalen Anzahl logischer Java-Operatoren auskommen. Klammern zählen nicht als Operatoren.
- Wenn es entsprechende Operatoren gäbe, könnten sie teilweise ausgewertet werden oder nicht?

3h. Primfaktoren

Schreiben Sie ein Programm **PrimeFactors**, das die Primfaktorenzerlegung einer Eingabezahl $n \geq 2$ berechnet. Dazu werden alle potenziellen Teiler nacheinander ausprobiert. Wenn ein echter Teiler gefunden wurde (denken Sie an den Modulus-Operator), wird er ausgegeben und die Zahl n entsprechend verringert. Die Suche endet, wenn n auf den Wert 1 gefallen ist. Die Primfaktoren werden in steigender Größe ausgegeben. Ein Beispiel:

```
$ java PrimeFactors 1668
2
3
139
```

Ignorieren Sie arithmetischen Überlauf.

3i. Text-Diamanten

Ein „Text-Diamant“ besteht aus einer rautenförmigen Anordnung von Sternen vor einem Hintergrund von Punkten. Hier sehen Sie einen Text-Diamanten der Größe 5:

```
..*..
.***.
*****
.***.
..*..
```

Schreiben Sie das Programm **Diamond**, das auf der Kommandozeile eine ungerade natürliche Zahl erhält und einen Diamanten dieser Höhe und Breite ausgibt. Drei Beispiele:

```
$ java Diamond 5
..*..
.***.
*****
.***.
..*..
$ java Diamond 11
.....*.....
....***....
...*****...
..*******..
.*********.
***********
.*********.
..*******..
...*****...
....***....
.....*.....
$ java Diamond 1
*
```

Die folgende Anweisung gibt ein einzelnes Zeichen aus, in diesem Fall einen Stern:

```
System.out.print("*");
```

Die Anweisung[20]

```
System.out.println ();
```

beginnt eine neue Zeile. Zum Beispiel produzieren die Anweisungen

```
System.out.print(".");
System.out.print("*");
System.out.print(".");
System.out.println ();          // weiter in die nächste Zeile
```

die Ausgabe

```
.*.
```

20 Der Bezeichner println kann gelesen werden als *print and start new line*.

3j. Perfekte Zahlen

Eine Zahl nennt man eine „Perfekte Zahl“, wenn sie gleich der Summe aller ihrer ganzzahligen Teiler ist (einschließlich der 1, ohne sie selbst). Die ersten beiden Perfekten Zahlen sind

$$6 = 3 + 2 + 1 \quad \text{und} \quad 28 = 14 + 7 + 4 + 2 + 1$$

Finden Sie drei weitere Perfekte Zahlen.

3k. Zahlensysteme

Schreiben Sie ein Programm **BaseXto10**, das Zahlen aus einem fremden Zahlensystem mit der Basis b in das Zehnersystem umrechnet. Zur Vereinfachung gilt $b < 10$. Eine Zahl n in einem fremden Zahlensystem mit der Basis b wird bezeichnet mit n_b. Zahlen ohne Angabe einer Basis gelten automatisch im Zehnersystem. Zwei Beispiele zur Umrechnung:

$$\begin{aligned} 2041_5 &= 2 \cdot 5^3 + 0 \cdot 5^2 + 4 \cdot 5^1 + 1 \cdot 5^0 \\ &= 2 \cdot 125 + 4 \cdot 5 + 1 \cdot 1 = 250 + 20 + 1 = 271_{10} \\ 11111110_2 &= 254_{10} \end{aligned}$$

BaseXto10 erhält zwei Argumente: Das erste Argument nennt eine Zahl in einem fremden Zahlensystem, das zweite Argument die Basis des fremden Zahlensystems. Zwei Beispiele:

```
$ java BaseXto10 2041 5
271
$ java BaseXto10 11111110 2
254
```

Schreiben Sie ein zweites Programm **Base10toX**, das eine Zahl aus dem Zehnersystem in ein fremdes Zahlensystem mit der Basis b umrechnet (wieder gilt $b < 10$). Gehen Sie analog zur ersten Teilaufgabe vor. Zwei Beispiele:

```
$ java Base10toX 271 5
2041
$ java Base10toX 254 2
11111110
```