2 Strukturierte Programmierung

Kapitelübersicht

In diesem Kapitel beschäftigen wir uns mit den Grundlagen von Algorithmen. Ein *Algorithmus*, oder eine *Routine*, ist eine exakt definierte Prozedur zur Lösung eines gegebenen Problems, also eine genau angegebene Abfolge von Anweisungen oder Einzelschritten. In der Sprache der Objektorientierung bildet ein Algorithmus eine „Methode“.

Bei einem Algorithmus spielen die zeitliche Abfolge der Anweisungen, logische Bedingungen (Abb. 2.1) und Wiederholungen eine wesentliche Rolle. Das Wort „Algorithmus“ leitet sich ab von dem Namen des bedeutenden persisch-arabischen Mathematikers Al-Chwarizmi[1] (ca. 780 – ca. 850). Dessen mathematisches Lehrbuch *Über das Rechnen mit indischen Ziffern*, um 825 erschienen und etwa 300 Jahre später (!) vom Arabischen ins Lateinische übersetzt, beschrieb das Rechnen mit dem im damaligen Europa unbekannten Dezimalsystem der Inder.[2]

[1] http://de.wikipedia.org/wiki/Al-Chwarizmi

[2] George Ifrah: *The Universal History of Numbers.* John Wiley & Sons, New York 2000, S. 362

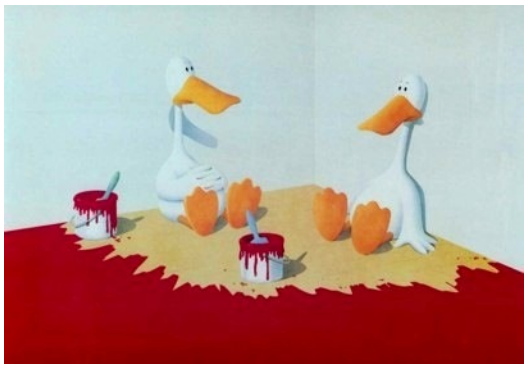


Abbildung 2.1. Für einen korrekt funktionierenden Algorithmus spielen Logik und zeitliche Reihenfolge der Einzelschritte eine wesentliche Rolle. ©1989 Michael Bedard *„The Failure of Marxism“*

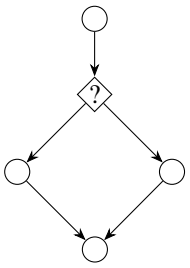
Auf diese Weise wurde das Wort Algorithmus zu einem Synonym für Rechenverfahren.

2.1 Logik und Verzweigung

2.1.1 Die Verzweigung

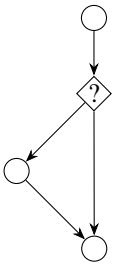
Die *Verzweigung* (auch: *Selektion*, *Auswahlstruktur* oder **`if`**-*Anweisung*) eine Anweisung, die abhängig von einer Bedingung zwischen alternativen Anweisungsfolgen auswählt. In Java hat sie die folgende Syntax:

```
if ( Bedingung ) {
    Anweisungen;
} else {
    Anweisungen;
}
```



Falls die Bedingung wahr ist (d.h. *Bedingung* = **`true`**), wird der Block direkt nach der Bedingung (kurz: der „**`if`**-Zweig“) ausgeführt, ansonsten (*Bedingung* = **`false`**) der Block nach **`else`** (der „**`else`**-Zweig“). Der **`else`**-Zweig kann weggelassen werden, falls keine Anweisung ausgeführt werden soll, wenn die Bedingung falsch ist:

```
if ( Bedingung ) {
    Anweisungen;
}
```



Die Bedingung ist eine logische Aussage, die entweder wahr (**`true`**) oder falsch (**`false`**) ist. Man nennt sie auch einen *Boole'schen Ausdruck*. Ein Beispiel für eine Verzweigung liefert die folgende Applikation:

```
import javax.swing.*;
```

```java
/** bestimt, ob ein Test bei eingegebener Note bestanden ist.
 */
public class Testergebnis {
  public static void main( String[] args ) {
    double note;
    String ausgabe;

    // Eingabeblock:
    JTextField[] feld = {new JTextField()};
    Object[] msg = {"Geben Sie die Note ein (1, ..., 6):", feld[0]};
    int click = JOptionPane.showConfirmDialog(null, msg, "Eingabe", 2);

    // Eingabe vom Typ String nach int konvertieren:
    note = Double.parseDouble( feld[0].getText() );           // (1)

    // Ergebnis bestimmen:
    if ( note <= 4 ) {
       ausgabe = "Bestanden";
    } else {
       ausgabe = "Durchgefallen";
    }

    JOptionPane.showMessageDialog(null,ausgabe);
  }
}
```

Diese Applikation erwartet die Eingabe einer Note und gibt abhängig davon aus, ob die Klausur bestanden ist oder nicht.

2.1.2 Logische Bedingungen durch Vergleiche

Durch die `if`-Anweisung wird eine Entscheidung auf Grund einer bestimmten *Bedingung* getroffen. Bedingungen für zwei Werte eines primitiven Datentyps können in Java durch *Vergleichsoperatoren* gebildet werden:

Algebraischer Operator	Operator in Java	Beispiel	Bedeutung
=	==	x == y	x ist gleich y
≠	!=	x != y	x ist ungleich y
>	>	x > y	x ist größer als y
<	<	x < y	x ist kleiner als y
≧	>=	x >= y	x ist größer gleich y
≦	<=	x <= y	x ist kleiner gleich y

Merkregel 5. Der Operator == wird oft verwechselt mit dem Zuweisungsoperator =. Der Operator == muss gelesen werden als „ist gleich“, der Operator = dagegen als „wird“ oder „bekommt den Wert von“.

Eine wichtige Einschränkung muss man bei den algebraischen Vergleichsoperatoren erwähnen:

Merkregel 6. Alle algebraischen Vergleichsoperatoren können im Allgemeinen nur zwei Ausdrücke von elementarem Datentyp vergleichen. Zwei Strings sollten dagegen nur mit der Methode `equals` verglichen werden:

```
string1.equals(string2).
```

Der Vergleichsoperator == funktioniert bei Strings nicht immer! Entsprechend sollte man auch den Ungleichheitsoperator != bei Strings nicht verwenden, sondern

```
!string1.equals(string2)
```

2.1.3 Logische Operatoren

Logische Operatoren werden benutzt, um logische Aussagen zu verknüpfen oder zu negieren. Technisch gesehen verknüpfen sie Boolesche Werte (**`true`** und **`false`**) miteinander. Java stellt die Grundoperationen UND, ODER, XOR („exklusives ODER“ oder „Antivalenz“) und NICHT zur Verfügung.

Logisches AND (&&)

Der &&-Operator führt die logische UND-Verknüpfung durch. Er ergibt **`true`**, wenn seine beiden Operanden **`true`** ergeben, ansonsten **`false`**.

Logisches OR (||)

Der ||-Operator führt die logische ODER-Verknüpfung durch. Er ergibt **`true`**, wenn einer seiner beiden Operanden **`true`** ergibt, ansonsten **`false`**.

Das Exklusive Oder XOR (^ oder !=)

Der XOR-Operator ^ führt die logische Verknüpfung des Exklusiven ODER durch. Hierbei ist A ^ B **`true`**, wenn A und B *unterschiedliche* Wahrheitswerte haben, und **`false`**, wenn sie gleiche Wahrheitswerte haben. Für Boole'sche Werte ist das äquivalent zu dem Ungleichheitsoperator !=. Im Unterschied zum XOR ^ ist dieser Operator jedoch nicht für bitweise Operationen geeignet, die wir im Folgenden betrachten werden.

Logische Negation (!)

Der Negations-Operator ! wird auf nur einen Operanden angewendet. Er ergibt **`true`**, wenn sein Operand **`false`** ist und umgekehrt.

Die Wahrheitstabellen in Tab. 2.1 zeigen die möglichen Ergebnisse der logischen Operatoren. Die erste Spalte ist beispielsweise folgendermaßen zu lesen: Wenn Aussage a den Wert **`false`** besitzt und b ebenfalls, dann ist a && b **`false`**, a || b ist **`false`**, usw.
Manchen fällt es leichter, sich Wahrheitstabellen mit 0 und 1 zu merken, wobei 0 = **`false`** und 1 = **`true`** gilt, siehe Tab. 2.2. Diese „Mathematisierung“ von **`false`** und **`true`** hat die verblüffende Konsequenz, dass wir mit den logischen Operatoren &&, || und ^ algebraisch rechnen können, wenn wir sie wie folgt durch die Grundrechenarten ausdrücken:

$$a \;\&\&\; b = a \cdot b, \qquad a \,||\, b = a+b-ab, \qquad (a \;!\!= b) = (a+b) \,\%\, 2, \qquad !a = (a+1) \,\%\, 2, \quad (2.1)$$

a	*b*	*a* && *b*	*a* || *b*	*a* != *b*	!*a*
false	false	false	false	false	true
false	true	false	true	true	–"–
true	false	false	true	true	false
true	true	true	true	false	–"–

Tabelle 2.1. Wahrheitstabellen für verschiedene logische Operatoren.

a	*b*	*a* && *b*	*a* || *b*	*a* != *b*	!*a*
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Tabelle 2.2. Wahrheitstabellen für verschiedene logische Operatoren.

mit $a, b \in \{0, 1\}$. (Probieren Sie es aus!) Daher spricht man völlig zurecht von einer „Algebra", und zwar nach ihrem Entdecker von der *Boole'schen Algebra*. Dass Computer mit ihrer Binärlogik überhaupt funktionieren, liegt letztlich an der Eigenschaft, dass die Wahrheitswerte eine Algebra bilden. Man erkennt an (2.1) übrigens direkt, dass

$$!a = (a\,!{=}1) \qquad \text{für } a \in \{0, 1\}. \tag{2.2}$$

Anwendung im Qualitätsmanagement

Betrachten wir als eine Anwendung das folgende Problem des Qualitätsmanagements eines Produktionsbetriebs. Es wird jeweils ein Los von 6 Artikeln, aus dem eine Stichprobe von drei zufällig ausgewählten Artikeln durch drei Roboter automatisiert qualitätsgeprüft wird. Die Artikel in dem Los sind von 0 bis 5 durchnummeriert. Das Programm soll so aufgebaut sein, dass es drei Zahlen `a1`, `a2` und `a3` zufällig aus der Menge $\{0, 1, \ldots, 5\}$ bestimmt, wobei der Wert von `a1` die Nummer des Artikels beschreibt, den Roboter Nr. 1 prüft, der Wert von `a2` den von Roboter Nr. 2 zu prüfenden Artikel, und `a3` den von Roboter Nr. 3 zu prüfenden Artikel. Werden für (`a1`, `a2`, `a3`) beispielsweise die drei Werte (3, 0, 5) bestimmt, also

$$(\texttt{a1}, \texttt{a2}, \texttt{a3}) = (3, 0, 5),$$

so soll Roboter 1 den Artikel 3 prüfen, Roboter 2 den Artikel 0 und Roboter 3 den Artikel 5. Für den Ablauf der Qualitätsprüfung ist es nun notwendig, dass die Werte von `a1`, `a2` und `a3` unterschiedlich sind, denn andernfalls würden mehrere Roboter einen Artikel prüfen müssen. Falls also nicht alle Zahlen unterschiedlich sind, soll das Programm eine Warnung ausgeben.

Zufallszahlen mit `Math.random()`. Um das Programm zu schreiben, benötigen wir zunächst eine Möglichkeit, eine Zufallszahl aus einem gegebenen Zahlbereich zu bekommen. In Java geschieht das am einfachsten mit der Funktion **`Math.random()`**. Das ist eine Funktion aus der Klasse `Math`, die einen zufälligen Wert z vom Typ **`double`** von 0 (einschließlich) bis 1 ausschließlich ergibt. Mit der Anweisung

```
double z = Math.random();
```

erhält die Variable z also einen Zufallswert aus dem Einheitsintervall $[0, 1[$, also $0 \leqq z < 1$. Was ist zu tun, um nun daraus eine ganze Zahl x aus dem Bereich $[0, 6[$ zu bilden? Zwei Schritte sind dazu nötig, einerseits muss der Bereich vergrößert werden, andererseits muss der **`double`**-Wert

zu einem `int`-Wert konvertiert werden. Der Bereich wird vergrößert, indem wir den z-Wert mit 6 multiplizieren, also $x = 6 \cdot z$. Damit ist x ein zufälliger Wert mit $0 \leqq x < 6$. Daraus kann durch einfaches Casten ein ganzzahliger Wert aus der Menge $\{0, 1, \ldots, 5\}$ gemacht werden. Zu einer einzigen Zeile zusammengefasst können wir also mit

```
int x = (int) ( 6 * Math.random() );
```

eine Zufallszahl x mit $x \in \{0, 1, \ldots, 5\}$ erhalten. Auf diese Weise können wir also den drei Zahlen `a1`, `a2` und `a3` zufällige Werte geben.

Fallunterscheidung mit logischen Operatoren. Wie wird nun geprüft, ob eine Warnmeldung ausgegeben werden soll? Da wir schon in der Alltagssprache sagen: „Wenn nicht alle Zahlen unterschiedlich sind, dann gib eine Warnung aus“, müssen wir eine `if`-Anweisung verwenden. Um die hinreichende Bedingung für die Warnmeldung zu formulieren, überlegen wir uns kurz, welche Fälle überhaupt eintreten können.

1. Fall: *Alle drei Zahlen sind unterschiedlich.* Als logische Aussage formuliert lautet dieser Fall:

   ```
   a1 != a2 && a1 != a3 && a2 != a3
   ```
   ist wahr.

2. Fall: *Genau zwei Zahlen sind gleich.* Als logische Aussage ist das der etwas längliche Ausdruck

   ```
   (a1 == a2 && a1 != a3 && a2 != a3) ||
   (a1 != a2 && a1 == a3 && a2 != a3) ||
   (a1 != a2 && a1 != a3 && a2 == a3)
   ```
   ist wahr.

   $\Longleftrightarrow$

   ```
   (a1 == a2 && a1 != a3) ||
   (a1 != a2 && a1 == a3) ||
   (a2 == a3 && a1 != a3)
   ```
   ist wahr.

3. Fall: *Alle drei Zahlen sind gleich.* Mit logischen Operatoren ausgedrückt:

   ```
   a1 == a2 && a1 == a3 && a2 == a3
   ```
   ist wahr.

Für unsere Warnmeldung benötigen wir nun jedoch gar nicht diese genaue Detaillierung an Fallunterscheidungen, also die Kriterien für die genaue Anzahl an übereinstimmenden Zahlen. Wir müssen nur prüfen, ob *mindestens* zwei Zahlen übereinstimmen. Die entsprechende Bedingung lautet:

```
a1 == a2 || a1 == a3 || a2 == a3
```
ist wahr.

Das deckt die Fälle 2 und 3 ab. Entsprechend ergibt sich das Programm wie folgt.

```
public class Qualitaetspruefung {
   public static void main(String[] args) {
      // 3 zufällig zur Prüfung ausgewählte Artikel aus {0, 1, ..., 5}:
      int a1 = (int) ( 6 * Math.random() );
      int a2 = (int) ( 6 * Math.random() );
      int a3 = (int) ( 6 * Math.random() );
```

```
        if( a1 == a2 || a1 == a3 || a2 == a3 ) {
           System.out.print("WARNUNG! Ein Artikel wird mehrfach geprueft: ");
        }
        System.out.println("(" + a1 + ", " + a2 + ", " + a3 + ")");
    }
}
```

2.1.4 Die `if-else-if`-Leiter

Möchte man eine endliche Alternative von Fällen abfragen, also mehrere Fälle, von denen nur einer zutreffen kann, so kann man mehrere **`if`**-Anweisungen verschachteln zu einer sogenannten *kaskadierten Verzweigung* oder „**`if-else-if`**-Leiter“ *(if-else-ladder)*. Beipielsweise kann man zur Ermittlung der (meteorologischen) Jahreszeit abhängig vom Monat den folgenden Quelltextausschnitt *(„Snippet“)* verwenden:

```
int monat = 5; // Mai, als Beispiel
String jahreszeit = "";
if (monat == 12 || monat == 1 || monat == 2) {
   jahreszeit = "Winter";
} else if (monat == 3 || monat == 4 || monat == 5) {
   jahreszeit = "Frühling";
} else if (monat == 6 || monat == 7 || monat == 8) {
   jahreszeit = "Sommer";
} else if (monat == 9 || monat == 10 || monat == 11) {
   jahreszeit = "Herbst";
} else {
   jahreszeit = "- unbekannter Monat -";
}
System.out.println("Im Monat " + monat + " ist " + jahreszeit);
```

2.1.5 Gleichheit von double-Werten

Prpft man zwei double-Werte auf Gleichheit, so kann es zu unerwarteten Ergebnissen führen. Beispielsweise ergibt der Vergleich `(0.4 - 0.3 == 0.1)` den Wert `false`, wie man schnell durch folgendes Quelltextfragment selber überprüfen kann:

```
double x = 0.4, y = 0.3;
System.out.println(x - y == 0.1);
```

Ursache ist die Tatsache, dass Werte vom Datentyp `double` als binäre Brüche gespeichert werden, und speziell `0.4` und `0.3` aber keine endliche Binärbruchentwicklung besitzen (vgl. S. 133).

2.1.6 Abfragen auf Zahlbereiche

Betrachten wir nun ein kleines Problem, bei dem wir zweckmäßigerweise mehrere Vergleiche logisch miteinander verknüpfen. Für eine eingegebene Zahl x soll ausgegeben werden, ob gilt

$$x \in [0, 10[\cup [90, 100]$$

```
import javax.swing.*;
```

```
/** bestimmt, ob eine eingegebene Zahl x in der Menge [0,10[∪[90,100] ist. */
public class Zahlbereich {
  public static void main( String[] args ) {
    double x;
    boolean istInMenge;
    String ausgabe;

    // Eingabeblock:
    JTextField[] feld = {new JTextField()};
    Object[] msg = {"Geben Sie eine Zahl ein:", feld[0]};
    int click = JOptionPane.showConfirmDialog(null, msg, "Eingabe", 2);

    // Eingabe nach double konvertieren:
    x = Double.parseDouble( feld[0].getText() );

    // Booleschen Wert bestimmen:
    istInMenge  = ( x >= 0 ) && ( x < 10 );
    istInMenge |= ( x >= 90 ) && ( x <= 100 );
    // Wert für Variable ausgabe bestimmen:
    if ( istInMenge ) {
       ausgabe = "Zahl " + x + '\u2208 [0,10[ \u222A [90,100]';
    } else {
       ausgabe = "Zahl " + x + '\u2209 [0,10[ \u222A [90,100]';
    }

    JOptionPane.showMessageDialog(null,ausgabe);
  }
}
```

Zu beachten ist, dass nun zur Laufzeit des Programms nach der Eingabe der Wert der Booleschen Variable `istInMenge` auf **`true`** gesetzt ist, wenn eine der beiden &&-verknüpften Bedingungen zutrifft.

Abfragen auf einen Zahlbereich für `int`-Werte

Möchte man in Java überprüfen, ob ein **`int`**-Wert n in einem bestimmten Bereich von a bis b liegt, also ob $n \in [a,b]$, so kann man sich die Darstellung des Zweierkomplements (der Darstellung von **`int`**- und **`long`**-Werten) zunutze machen und den Wahrheitswert mit nur einem Vergleich durchführen:

$$\texttt{(a<=n \&\& n<=b)} \iff \texttt{(n - a - 0x80000000 <= b - a - 0x80000000)} \qquad (2.3)$$

Hierbei ist `0x80000000` $= -2^{31} =$ `-Integer.MAX_VALUE-1`. Damit ist der Wert `x - a - 0x80000000` für jeden Integerwert `x` stets positiv.

„Short-Circuit“-Evaluation und bitweise Operatoren

Java stellt die UND- und ODER-Verknüpfungen in zwei verschiedenen Varianten zur Verfügung, nämlich mit Short-Circuit-Evaluation oder ohne.

Bei der *Short-Circuit-Evaluation* eines logischen Ausdrucks wird ein weiter rechts stehender Teilausdruck nur dann ausgewertet, wenn er für das Ergebnis des Gesamtausdrucks noch von Bedeutung ist. Falls in dem Ausdruck a && b also bereits a falsch ist, wird zwangsläufig immer auch a && b falsch sein, unabhängig von dem Resultat von b. Bei der Short-Circuit-Evaluation wird in diesem Fall b gar nicht mehr ausgewertet. Analoges gilt bei der Anwendung des ODER-Operators.

Die drei Operatoren &, | und ^ können auch als bitweise Operatoren auf die ganzzahligen Datentypen **`char`**, **`byte`**, **`short`**, **`int`** und **`long`** angewendet werden. Hierbei wirkt die logische Operation jeweils auf jedes einzelne Bit der Binärdarstellung (bei **`char`** ist der benutzte Wert der Unicode-Wert des Zeichens). Z.B. ergibt 25 ^ 31 = 6, denn

$$\begin{array}{r} 25_{10} = 1\,1\,0\,0\,1_2 \\ \hat{}\,31_{10} = 1\,1\,1\,1\,1_2 \\ \hline 0\,0\,1\,1\,0_2 \end{array} \tag{2.4}$$

Einen Überblick über die bitweisen Operatoren in Java gibt Tabelle A.2 auf S. 142.

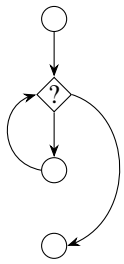
2.2 Iterationen: Wiederholung durch Schleifen

Eine der wichtigsten Eigenschaften von Computern ist ihre Fähigkeit, sich wiederholende Tätigkeiten immer und immer wieder auszuführen. Eine wichtige Struktur zur Programmierung von Wiederholungen sind die „Iterationen“ oder Schleifen. Es gibt in den meisten Programmiersprachen drei Schleifenkonstrukte, die jeweils bestimmten Umständen angepasst sind, obwohl man im Prinzip mit einer einzigen auskommen kann. In Java ist diese allgemeine Schleifenstruktur die **`while`**-Schleife.

2.2.1 Die `while`-Schleife

Bei einer Iteration oder *Schleife* wird eine bestimmte Abfolge von Anweisungen (der *Schleifenrumpf*) wiederholt, solange eine bestimmte Bedingung (die *Schleifenbedingung*) wahr ist. In Java wird eine Schleifenstruktur durch die folgende Syntax gegeben:

```
while ( Bedingung ) {
    Anweisung i_1 ;
    ...
    Anweisung i_n ;
}
```



Um das Prinzip der **`while`**-Schleife zu demonstrieren, betrachten wir den folgenden logischen Programmausschnitt („Pseudocode“), der einen Algorithmus zur Erledigung eines Einkaufs beschreibt:

```
while (es existiert noch ein nichterledigter Eintrag auf der Einkaufsliste) {
   erledige den nächsten Eintrag der Einkaufsliste;
}
```

Man macht sich nach kurzem Nachdenken klar, dass man mit dieser Schleife tatsächlich alle Einträge der Liste erledigt: Die Bedingung ist entweder wahr oder falsch. Ist sie wahr, so wird der nächste offene Eintrag erledigt, und die Bedingung wird erneut abgefragt. Irgendwann ist

der letzte Eintrag erledigt – dann ist aber auch die Bedingung falsch! Die Schleife wird also nicht mehr durchlaufen, sie ist beendet.

Merkregel 7. In einer Schleife sollte stets eine Anweisung ausgeführt werden, die dazu führt, dass die Schleifenbedingung (irgendwann) falsch wird. Ansonsten wird die Schleife nie beendet, man spricht von einer *Endlosschleife*. Außerdem: Niemals ein Semikolon direkt nach dem Wort **while**! (Einzige Ausnahme: do/while, s.u.)

2.2.2 Die Applikation *Guthaben*

Die folgende Applikation verwaltet ein Guthaben von 100 €. Der Anwender wird solange aufgefordert, von seinem Guthaben abzuheben, bis es aufgebraucht ist.

```
import javax.swing.*;

/** Verwaltet ein Guthaben von 100 €. */
public class Guthaben {
  public static void main( String[] args ) {
    int guthaben = 100;
    int betrag = 0;
    String eingabe, text;

    while (guthaben > 0) {
       text  = "Ihr Guthaben: " + guthaben + "€";
       text += "\nAuszahlungsbetrag:";
       // Eingabe:
       eingabe = JOptionPane.showInputDialog(text);
       betrag = Integer.parseInt(eingabe);
       guthaben -= betrag;
    }

    text  = "Ihr Guthaben ist aufgebraucht!";
    text += "\nEs beträgt nun " + guthaben + " €.";
    JOptionPane.showMessageDialog(null,text);
  }
}
```

In diesem Programm werden in der main-Methode zunächst die verwendeten Variablen deklariert (bis auf diejenigen des Eingabeblocks). Dabei ist die Variable `guthaben` der „Speicher“ für das aktuelle Guthaben, das mit 100 € initialisiert wird. Die beiden Variablen `betrag` und `text` werden später noch benötigt, sie werden zunächst mit 0 bzw. dem leeren String initialisiert.

Als nächstes wird eine **while**-Schleife aufgerufen. Sie wird solange ausgeführt, wie das Guthaben positiv ist. Das bedeutet, dass beim ersten Mal, wo das Guthaben durch die Initialisierung ja noch 100 € beträgt, der Schleifenrumpf ausgeführt wird. Innerhalb der Schleife wird zunächst die Variable `text` mit der Angabe des aktuellen Guthabens belegt, die dann als Informationszeile in dem Eingabeblock verwendet wird. Nachdem der Anwender dann eine Zahl eingegeben hat, wird diese zu einem **int**-Wert konvertiert und vom aktuellen Guthaben abgezogen. Sollte nun das Guthaben nicht aufgebraucht oder überzogen sein, so wird die Schleife erneut ausgeführt und der Anwender wieder zum Abheben aufgefordert. Nach Beendigung der

Schleife wird schließlich eine kurze Meldung über das aktuelle Guthaben ausgegeben und das Programm beendet.

2.2.3 Umrechnung Dezimal- in Binärdarstellung

Es folgt eine weitere, etwas mathematischere Applikation, die eine `while`-Schleife verwendet. Hierbei geht es um die Umrechnung einer Zahl in Dezimaldarstellung in die Binärdarstellung.

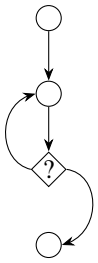
```
import javax.swing.JOptionPane;


/**
 * Berechnet zu einer eingegebenen positiven Dezimalzahl
 * die Binärdarstellung als String.
 */
public class DezimalNachBinaer {
  public static void main( String[] args ) {
    int dezimalzahlEingabe = 0;        // Dezimalzahl
    int naechsteBinaereZiffer = 0;  // Binärziffer
    String binaer = "";
    String ausgabe = ""; // Binärzahl


   // Eingabe:
   String textEingabe = JOptionPane.showInputDialog(
"Bitte geben Sie eine Dezimalzahl ein.");
    dezimalzahlEingabe = Integer.parseInt( textEingabe );

    while (dezimalzahlEingabe > 0) {
       naechsteBinaereZiffer = dezimalzahlEingabe % 2;
       // Ziffer vor den bisher aufgebauten String setzen:
       binaer = naechsteBinaereZiffer + binaer;
       dezimalzahlEingabe /= 2;
    }

    ausgabe = textEingabe + "im Dezimalsystem entspricht "
+ binaer + " im Binärsystem.";
    JOptionPane.showMessageDialog(null,ausgabe);
  }
}
```

Die Schleife besteht aus drei Schritten: Zunächst wird die Ziffer (entweder 0 oder 1) von dem aktuellen Wert der Zahl z bestimmt, danach wird diese Ziffer vor den aktuellen String `binaer` gesetzt, und schließlich wird z ganzzahlig durch 2 dividiert. Dies wird solange wiederholt, wie z positiv ist (oder anders ausgedrückt: solange *bis* $z = 0$ ist).

2.2.4 Die `do/while`-Schleife

Die `do/while`-Schleife ist der `while`-Schleife sehr ähnlich. Allerdings wird hier Schleifenbedingung *nach* Ausführung des Schleifenblocks geprüft. Die Syntax lautet wie folgt.

```
do {
   Anweisungsblock;
} while ( Bedingung );
```



Merkregel 8. Bei einer `do/while`-Schleife wird der Schleifenrumpf stets mindestens einmal ausgeführt. Bei der Syntax ist zu beachten, dass nach der Schleifenbedingung ein Semikolon gesetzt wird.

Als Beispiel betrachten wir die folgende Applikation, die eine kleine Variation der „Guthabenverwaltung“ darstellt.

```
import javax.swing.*;
/** Verwaltet ein Guthaben von 100 €.
 */
public class GuthabenDoWhile {
  public static void main( String[] args ) {
    double guthaben = 100.0, betrag = .0;   // (1)
    String text = "";

    do {
       text  = "Ihr Guthaben: " + guthaben + "€";
       text += "\nAuszahlungsbetrag:";
       // Eingabe:
       eingabe = JOptionPane.showInputDialog(text);
       betrag = Double.parseDouble( eingabe );
       guthaben -= betrag;
    } while ( guthaben > 0 );

    text  = "Ihr Guthaben ist aufgebraucht!";
    text += "\nEs betr\u00E4gt nun " + guthaben + " \u20AC.";
    JOptionPane.showMessageDialog(null,text);
  }
}
```

Hier wird der Schleifenrumpf, also die Frage nach der Auszahlung, mindestens einmal ausgeführt — sogar wenn das Guthaben anfangs nicht im Positiven wäre . . .

2.2.5 Die `for`-Schleife

Eine weitere Schleifenkonstruktion ist die `for`-Schleife. Ihre Die Syntax lautet

```
for ( start; check; update ) {
   ...;
}
```

Hier bezeichnet *start* also die einmalig zum Start der Schleife durchzuführenden Initialisierungen

von Variablen, *check* die vor jeder Iteration zu überprüfende Schleifenbedingung und *update* stets am Ende des Schleifenrumpfs auszuführenden Anweisungen. Die `for`-Schleife eignet sich besonders zur Implementierung einer *Zählschleife*, also für Wiederholungen, die durch einen Index oder einen Zähler gesteuert werden und bei denen bereits zum Schleifenbeginn klar ist, wie oft sie ausgeführt werden.

```
for ( int Zähler = Startwert; Zähler <= Maximum; Aktualisierung ) {
   Anweisungsblock;
}
```

Beispielsweise kann man eine Zählvariable `i` von einem Startwert bis zu einen Endwert wie folgt hochzählen:

```
for (int i = 0; i < 10; i++) {
   System.out.print(i + ", ");
}
```

oder runterzählen:

```
for (int i = 10; i > 0; i--) {
   System.out.print(i + ", ");
}
```

In diesem Schleifenkonstrukt werden also im Gegensatz zu den `while`-Schleifen die für die Verwaltung einer Schleife wichtigen Anweisungen übersichtlich im Schleifenkopf aufgelistet. Auch mit einer `for`-Schleife kann man eine Endlosschleife programmieren, wenn nämlich die Update-Anweisung (und die Anweisungen im Schleifenrumpf) nicht dazu führen, dass die Schleifenbedingung falsch wird. Beispielsweise: `for(int i=10; i>0; i++)` ...

Als Beispiel betrachten wir die folgende Applikation, in der der Anwender fünf Zahlen (z.B. Daten aus einer Messreihe oder Noten aus Klausuren) eingeben kann und die deren Mittelwert berechnet und ausgibt. Nebenbei lernen wir, wie man **`double`**-Variablen initialisieren kann, Eingabefelder vorbelegt und Dezimalzahlen für die Ausgabe formatieren kann.

```
import javax.swing.*;
import java.text.DecimalFormat;


/**
 * Berechnet den Mittelwert 5 einzugebender Daten.
 */
public class Mittelwert {
  public static void main( String[] args ) {
    int max = 5;  // maximale Anzahl Daten
    double mittelwert, wert = 0.0, gesamtsumme = .0;                //(1)
    String ausgabe = "";
```

```
        // Eingabeblock:
        JTextField[] feld = {
          new JTextField("0"), new JTextField("0"), new JTextField("0"),
          new JTextField("0"), new JTextField("0")                    //(2)
        };
        Object[] msg = {"Daten:",feld[0],feld[1],feld[2],feld[3],feld[4]};
        int click = JOptionPane.showConfirmDialog(null, msg, "Eingabe", 2);
```

```
        for ( int i = 0; i < max; i++ ) {
           // Konvertierung des nächsten Datenwerts von String nach double:
           wert = Double.parseDouble( feld[i].getText() );            //(3)
           // Addition des Datenwerts zur Gesamtsumme:
           gesamtsumme += wert;
        }
```

```
        // Berechnung des Mittelwerts:
        mittelwert = gesamtsumme / max;                               //(4)
        // Bestimmung des Ausgabeformats:
        DecimalFormat zweiStellen = new DecimalFormat("#,##0.00");    //(5)
        // Ausgabe des Ergebnisses:
        ausgabe  = "Mittelwert der Daten: ";
        ausgabe += zweiStellen.format( mittelwert );                  //(6)
        JOptionPane.showMessageDialog(null,ausgabe);
      }
    }
```

Wie Sie sehen, findet das Hochzählen des Zählers (hier `i`) nicht wie bei der **`while`**-Schleife im Anweisungsblock statt, sondern bereits im Anfangsteil der Schleife.

Programmablauf

Zunächst werden in der main-Methode die Variablen initialisiert. In Anmerkung (1) sind zwei Arten angegeben, wie man Variablen **`double`**-Werte zuweisen kann, nämlich mit `wert = 0.0` oder auch mit `.0`; eine weitere wäre auch mit `0.0d`.

In Anmerkung (2) werden Textfelder erzeugt, die bei der Anzeige eine Vorbelegung haben, hier mit `"0"`. Erst nach OK-Klicken des Eingabefeldes wird dann die **`for`**-Schleife ausgeführt.

Innerhalb der **`for`**-Schleife werden die Werte der Textfelder in (3) nach **`double`** konvertiert und direkt auf die Variable `gesamtsumme` aufddiert. Nach Beendigung der Schleife ist der Wert in `gesamtsumme` also die Gesamtsumme aller in die Textfelder eingegebenen Werte. Danach wird der Mittelwert in (4) berechnet.

Formatierung von Zahlen

In der Zeile von Anmerkung (5) wird das Ergebnis unserer Applikation mit Hilfe der Klasse `DecimalFormat` auf zwei Nachkommastellen, einem Tausendertrenner und mindestens einer Zahl vor dem Komma formatiert. In dem „Formatierungsmuster“ bedeutet `#`, dass an dieser Stelle eine Ziffer nur angezeigt wird, wenn sie ungleich 0 ist, während für `0` hier auch eine 0 angezeigt wird. Mit der Anweisung (6)

```
zweiStellen.format( x );
```

wird der **double**-Wert x mit zwei Nachkommastellen dargestellt; sollte er Tausenderstellen haben, so werden diese getrennt, also z.B. 12.123.123,00. Andererseits wird mindestens eine Stelle vor dem Komma angegeben, also 0,00023.

Etwas verwirrend ist, dass man bei der Formatierungsanweisung das Muster in der englischen Notation angeben muss, also das Komma als Tausendertrenner und den Punkt als Dezimaltrenner — bei der Ausgabe werden dann aber die Landeseinstellungen des Systems verwendet.

2.2.6 Wann welche Schleife verwenden?

Obwohl wir mit unserem Beispielprogramm eine Schleife kennengelernt haben, die in allen drei möglichen Varianten **while**, **for** und **do/while** implementiert werden kann, ist die **while**-Konstruktion die allgemeinste der drei. Mit ihr kann jede Schleife formuliert werden.

Merkregel 9. Die **for**- und die **do/while**-Schleife sind jeweils ein Spezialfall der **while**-Schleife: Jede **for**-Schleife und jede **do/while**-Schleife kann durch eine **while**-Schleife ausgedrückt werden (umgekehrt aber nicht unbedingt).

Betrachten Sie also die **while**-Schleife als *die* eigentliche Schleifenstruktur; die beiden anderen sind Nebenstrukturen, die für gewisse spezielle Fälle eingesetzt werden können.

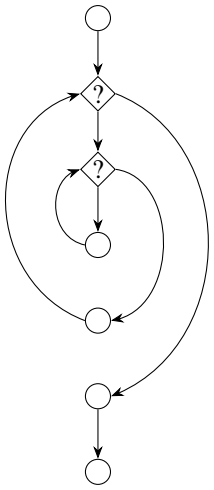
Schleife	Verwendung
while	prinzipiell alle Schleifen; insbesondere diejenigen, für die die Anzahl der Schleifendurchläufe beim Schleifenbeginn nicht bekannt ist, sondern sich dynamisch innerhalb der Schleife ergibt
for	Schleifen mit Laufindex, für die die Anzahl der Wiederholungen vor Schleifenbeginn bekannt ist
do/while	Schleifen, die *mindestens einmal* durchlaufen werden müssen und bei denen die Schleifenbedingung erst nach dem Schleifenrumpf geprüft werden soll.

Tabelle 2.3. Die Schleifenkonstruktionen von Java und ihre Verwendung

2.3 Tiefe Schleifen

Grundsätzlich kann man eine Schleife behandeln wie eine einzelne Anweisung. Natürlich eine, die mehrere Iterationen ausführt. Grundsätzlich kann man also mehrere Schleifen nacheinander ausführen, aber eben auch ineinander verschachteln.

```
...
while (?) {
    ...
    while (?) {
        ...;
    }
    ...;
}
...;
```



Solche verschachtelten Schleifen nennen wir *tiefe Schleifen*. Sie sind ein mächtiges Werkzeug zur Programmierung komplexer Algorithmen mit relativ wenig Quelltext. Wir werden zwei Anwendungsfälle betrachten, das Zeichnen von „zweidimensionalen" Strukturen und die Brute-Force-Suche.

2.3.1 Zweidimensionale Zeichenraster

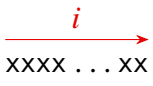
Gegeben sei das Problem, eine Zeile von 10 Zeichen hintereinander auf der Konsole auszugeben. Da hier also zehnmal derselbe Befehl aufgerufen werden soll, bietet sich dafür die **for**-Schleife an:

```
char zeichen = 'x';
// Gibt 10 Zeichen in einer Reihe aus:
int anzahl = 10;
for (int i = 1; i <= anzahl; i++) {
   System.out.print(zeichen);
}
System.out.println();
```

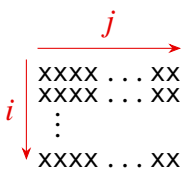
Diese Schleife bewirkt die Ausgabe:

```
xxxxxxxxxx
```

Wollen wir nun ein Rechteck von Zeichen ausgeben, also ein Zeichenraster mit vorgegebener Reihen- und Spaltenzahl, so können wir auf die obige Schleife zurückgreifen. Abstrakt gesehen erzeugen wir mit der Schleife einen linearen Strom von Zeichen, den die Zählvariable i steuert:



Die Zählvariable können wir hier als eindimensionale Koordinate auffassen. Bei einem Rechteck wollen wir diesen Strom selber in einen übergeordneten Strom wiederholen, d.h. wir brauchen *zwei* Koordinaten i und j:



(Wir haben hier gemäß Konvention die Koordinate i im Gegensatz zu oben nun in vertikaler Richtung orientiert, wir hätten es natürlich auch anders machen können.) Programmiertechnisch gesehen brauchen wir also zwei verschachtelte Schleifen mit unterschiedlichen Zählvariablen, wobei die äußere Schleife die Zeilen mit i zählt und die innere die Spalten mit j:

```
char zeichen = 'x';
// Gibt 5 x 20 Zeichen als Rechteck aus:
int reihen  = 5;
int spalten = 20;
for (int i = 1; i <= reihen; i++) {
   for (int j = 1; j <= spalten; j++) {
      System.out.print(zeichen);
   }
   System.out.println(); // nach jeder Reihe einen Zeilenumbruch
}
```

Die Ausgabe ist hier also ein Zeichenraster mit 5 Zeilen und 20 Spalten:

```
xxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxx
```

Wollen wir die Maße des ausgegebenen Rechtecks variieren, so müssen wir lediglich die Werte der Variablen `reihen` und `spalten` verändern, die für die Schleifen eine Art Parameter darstellen.

Insgesamt erkennen wie also, dass wir eine zweidimensionale Ratserstruktur durch zwei verschachtelte Schleifen programmieren können.

2.3.2 Brute-Force

Eine Brute-Force-Suche (*brute force* – engl.: „rohe Gewalt“), auch *erschöpfende Suche* oder Exhaustion genannt, ist ein allgemeines Verfahren, um ein Problem durch vollständiges Aufzählen oder Durchprobieren aller Möglichkeiten zu lösen. Zum Beispiel verwendet man Brute-Force, um ein Passwort zu knacken oder die Erfüllbarkeit einer aussagenlogischen Formel zu prüfen. Grundsätzlich kann eine Suche mit Brute-Force natürlich nur gelingen, wenn die Anzahl der zu durchlaufenden Möglichkeiten nur endlich ist. Die Menge der Möglichkeiten nenn man in diesem Zusammenhang oft auch „Suchraum“.

Beispiel 2.1. *(Knacken einer vierstelligen PIN)* Die üblichen Debitkarten haben eine vierstellige numerische PIN. Eine PIN nimmt daher einen der 10 000 möglichen Werte 0, 1, . . . , 9999 an, wobei wir vereinfachend für Zahlen kleine als 1 000 die fehlenden Stellen mit führenden Nullen auffüllen (z.B. steht 52 für die PIN 0052). In dem folgenden Quelltextausschnitt wird zunächst eine zufällige PIN aus dem Bereich $\{0, 1, \ldots, 9999\}$ erzeugt und dann mit Brute-Force gefunden.

```
// Knacken einer PIN:
int pin = (int) (10000 * Math.random());

int n = 0;
while (n != pin) {
   n++;
}
System.out.println("Die PIN lautet: " + n);
```

Die Brute-Force-Methode ist hier so einfach, da der zu durchlaufende Suchraum eine endliche Menge von Zahlen ist. □

Um ein Passwort zu knacken, reicht eine einfache Schleife wie für das Herausfinden einer PIN nicht aus. Der Grund liegt darin, dass ein Passwort aus mehreren Zeichen besteht, die nicht mehr durch einen einfachen Zahlbereich dargestellt werden können. Für ein Passwort mit einer einzigen Stelle allerdings klappt dies. Untersuchen wir zunächst diesen Fall, den allgemeinen können wir dann daraus ableiten.

Beispiel 2.2. *(Knacken eines einstelligen Passworts)* Ein Passwort besteht im allgemeinen Fall aus Unicode-Zeichen. Das ermöglicht es uns, systematisch alle Zeichen darzustellen, indem wir den Codepoint, d.h. eine ganze Zahl, nehmen und durch einen Cast nach `char` das entsprechende Zeichen erhalten: Zum Beispiel ergibt

```
System.out.println( (char) 65 );
```

die Ausgabe ‘A’, denn 65 ist der Codepoint von ‘A’; vgl. Abschnitt A.1 ab Seite 129. Wir beschränken uns hier auf den Zeichensatz „Basic Latin“ von Codepoint 33 bis 256. (Alle Codepoints kleiner als 33 stellen Steuerzeichen dar.) Die Brute-Force-Suche können wir dann mit einer einfachen Schleife durchführen, die mit dem Codepoint 33 startet und höchstens bis zum Codepoint 256 läuft:

```
// Knacken eines einstelligen Passworts:
String passwort = "A";
String tipp;

int i = 33; // Codepoint als Index
do {
   // Codepoint i casten und in einen String umformen:
   tipp = "" + (char) i;
   i++;
} while ( !tipp.equals(passwort) && i < 256 );

System.out.println("Das Passwort lautet: " + tipp);
```

Das jeweils zu testende Passwort wird als String in der Variablen `entschlüsselt` gespeichert. Hierbei müssen wir nur das gerade erhaltene Zeichen in einen String umwandeln, was mit der Addition zum leeren String geschehen kann. □

Beispiel 2.3. *(Knacken eines zweistelligen Passworts)* Mit dem Ansatz aus Beispiel 2.2 für ein einstelliges Passwort können wir nun auch ein zweistelliges Passwort knacken. Wir müssen nun systematisch für jeden einzelnen Buchstaben an der ersten Stelle für die zweite Stelle alle Möglichkeiten durchlaufen, also eine innere Schleife. Die Brute-Force-Suche können wir dann mit einer einfachen Schleife durchführen, die mit dem Codepoint 33 startet und höchstens bis zum Codepoint 255 läuft:

```
// Knacken eines zweistelligen Passworts:
String passwort = "Ab";
String tipp;
int j, k;

int i = 33;
do {
   j = 33;
```

```
        do {
            // Codepoints i, j casten und in einen String umformen:
            tipp = "" + (char) i + (char) j;
            j++;
        } while ( !tipp.equals(passwort) && j < 256 );
        i++;
    } while ( !tipp.equals(passwort) && i < 256 );

    System.out.println("Das Passwort lautet: " + tipp);
```

Diese Brute-Force-Suche können wir auf n-stellige Passwörter verallgemeinern, indem wir n Schleifen verschachteln und jeweils eine neue Codepoint-Variable i, j, k, ... von 33 bis 255 durchlaufen lassen, aber abbrechen, wenn wir das Passwort entdeckt haben. □

2.4 Arrays

Arrays sind eines der grundlegenden Objekte des wichtigen Gebiets der so genannten „Datenstrukturen“, das sich mit der Frage beschäftigt, wie für einen bestimmten Zweck mehr oder weniger komplexe Daten gespeichert werden können. Die Definition eines Arrrays und seine Erzeugung in Java sind Gegenstand der folgenden Abschnitte.

2.4.1 Was ist ein Array?

Ein *Array*[3] (auf Deutsch manchmal: *Reihung* oder *Feld*) ist eine Art Container von mehreren Datenelementen desselben Typs. Die Elemente eines Arrays kann man unter demselben Namen mit einem so genannten *Index* ansprechen („adressieren“). Dieser Index wird manchmal auch „Adresse“ oder „Zeiger“ (engl. *pointer*) des Elements in dem Array genannt.

Stellen Sie sich z.B. vor, Sie müssten die die Zu- und Abgänge eines Lagers für jeden Werktag speichern. Mit den Techniken, die wir bisher kennengelernt haben, hätten Sie nur die Möglichkeit, eine Variable für jeden Tag anzulegen: `zugang1`, `zugang2`, usw. Abgesehen von der aufwendigen Schreibarbeit wäre so etwas sehr unhandlich für weitere Verarbeitungen, wenn man beispielsweise den Bestand als die Summe der Zugänge der Woche wollte:

```
bestand = zugang1 + zugang2 + ...+ zugang5;
```

Falls man dies für den Jahresbestand tun wollte, hätte man einiges zu tippen ...

Die Lösung für solche Problemstellungen sind Arrays. Wir können uns ein Array als eine Liste nummerierter Kästchen vorstellen, so dass das Datenelement Nummer i sich in Kästchen Nummer i befindet.

`array` =	E	i	n	␣	A	r	r	a	y
	↑	↑	↑	↑	↑	↑	↑	↑	↑
Index =	0	1	2	3	4	5	6	7	8

Hier haben wir Daten des Typs `char`, und jedes Datenelement kann über einen Index angesprochen werden. So ist beispielsweise `'A'` über den Index 4 zugreifbar, also hier `array[4]`. Zu beachten ist: Man fängt in einem Array stets bei 0 an zu zählen, d.h. das erste Kästchen trägt die Nummer 0!

[3] *array* – engl. für: Anordnung, Aufstellung, (Schlacht-)Reihe, Aufgebot