Kapitel 3
Entscheidungen

Es wäre übertrieben, zu sagen, dass ein Computer ohne die Fähigkeit, Entscheidungen zu treffen, wertlos ist. Er wäre immer noch eine extrem schnelle Rechenmaschine. Aber alles, was darüber hinausgeht, ist ohne Entscheidungen nicht möglich. Von Minesweeper bis World of Warcraft, vom Dateisystem bis zum Webbrowser: Alle Anwendungen, egal, welcher Komplexität, benötigen die Möglichkeit, den Programmablauf zu verzweigen.

Eine Möglichkeit, den Programmablauf zu verzweigen und entweder den einen oder den anderen Programmteil auszuführen, ist für jede Programmiersprache unverzichtbar. Sei es, um auf Benutzereingaben zu reagieren oder auf interne Zustände, nur sehr wenige nützliche Programme führen ihren Code linear vom Anfang bis zum Ende aus. Java bietet verschiedene Sprachkonstrukte, die den Programmablauf verzweigen und die Sie in diesem Kapitel kennenlernen werden.

3.1 Entweder-oder-Entscheidungen

Das häufigste Konstrukt, das den Programmablauf verzweigt, ist das `if`-Statement. Es trifft die Entscheidung, welcher Programmzweig ausgeführt werden soll, aufgrund eines booleschen Wertes. Das kann eine `boolean`-Variable, eine `boolean`-Methode oder ein Ausdruck mit dem Ergebnistyp `boolean` durch die Anwendung der Vergleichsoperatoren sein.

```
public int betrag(int x){
    if (x < 0){
        x = -x;
    }
    return x;
}
```

Listing 3.1 Den Betrag einer Zahl mit »if« berechnen

Die Bedingung, nach der entschieden wird, steht in Klammern nach dem Schlüsselwort `if`. Darauf folgt der Code, der ausgeführt werden soll, wenn die Bedingung wahr ist. Normalerweise handelt es sich dabei um einen Codeblock in geschweiften Klammern, wie im Beispiel gezeigt. Aber wenn nur eine Anweisung folgt, und nur dann, können Sie auf die geschweiften Klammern verzichten. Das folgende Beispiel ist mit dem in Listing 3.1 völlig gleichwertig.

```
public int betrag(int x){
    if (x < 0)
        x = -x;
    return x;
}
```

Listing 3.2 Betragsberechnung ohne Codeblock

Obwohl unter bestimmten Bedingungen auf die geschweiften Klammern verzichtet werden kann, zeigt die Erfahrung aus dem Alltag, dass es besser ist, sie immer zu setzen, auch wenn der Block nur ein Statement enthält. Der Grund dafür ist kein technischer, sondern Übersichtlichkeit und Fehlervermeidung: Will man später ein zweites Statement hinzufügen, vergisst man leicht, die Klammern zu setzen. Deshalb lautet meine Empfehlung, auch einzelne Statements zu klammern, um diesen Fehlern vorzubeugen. Das Gleiche gilt für andere Sprachkonstrukte wie `else`-Blöcke und Schleifen, aber **nicht** für Klassen- und Methodendeklarationen. Bei diesen müssen die geschweiften Klammern immer gesetzt werden.

Wie in den Beispielen gezeigt, wird ein Codeblock entweder ausgeführt oder nicht. Sie haben aber auch die Möglichkeit, entweder den einen Codeblock auszuführen oder den anderen. Dazu dient die optionale `else`-Klausel des `if`-Statements.

```
public void begruesse(String name){
    if (name != null){
        System.out.println("Hallo, " + name);
    } else {
        System.out.println("Hallo, Fremder");
    }
}
```

Listing 3.3 Persönliche Begrüßung mit »if«-»else«

Der `else`-Block wird ausgeführt, wenn die Bedingung nicht zutrifft. Es wird also immer genau einer der Codeblöcke ausgeführt, entweder der `if`-Block oder der `else`-Block.

3.1.1 Übung: Star Trek – sehen oder nicht?

Schon mit diesen einfachen Entscheidungsoptionen kann Ihnen ein Computerprogramm helfen, schwierige und wichtige Alltagsentscheidungen zu treffen. Sicherlich haben Sie bereits vom Fluch der alten Star-Trek-Filme gehört? Die Filme mit geraden Nummern (»Der Zorn des Khan«, »Zurück in die Gegenwart«, »Das unentdeckte Land«, »Der erste Kontakt«) sind sehr gute Filme, die Filme mit ungeraden Nummern … nicht. »Nemesis« fällt aus diesem Muster heraus, der Film hat zwar eine gerade Nummer, ist aber trotzdem nicht gut. Aber darum kümmern wir uns später. Für den Moment besteht Ihre Aufgabe darin, ein Programm zu schreiben, das vom Benutzer erfragt, welchen Teil der Star-Trek-Reihe er ansehen möchte, und ihn bei Eingabe einer geraden Zahl dazu ermutigt, bei Eingabe einer ungeraden Zahl aber davon abhält.

Ein- und Ausgabe

In dieser Übung, und auch in vielen weiteren, müssen Sie mit dem Benutzer über die Kommandozeile kommunizieren. Sie haben in den Beispielen bereits gesehen, wie Ihr Programm Text auf die Kommandozeile ausgeben kann: Die Methode `System.out.println` erwartet einen String-Parameter und gibt ihn auf die Standardausgabe aus, gefolgt von einem Zeilenumbruch:

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

Eine Eingabe von der Kommandozeile zu lesen, ist etwas schwieriger. Die Standardeingabe finden Sie ebenfalls im `System`-Objekt, unter dem Namen `System.in`. Die Standardeingabe liefert allerdings Binärdaten an das Programm – das ist für unsere Zwecke nicht nützlich. Um aus dem `byte`-Strom einen `char`-Strom zu machen, aus dem Sie dann Text lesen können, ist ein kleiner Vorgriff auf Kapitel 12, »Dateien, Streams und Reader«, notwendig. Sie können die Klasse `java.io.InputStreamReader` verwenden, um die Standardeingabe zu transformieren. Damit können Sie einzelne Zeichen von der Standardeingabe lesen. Noch viel praktischer wäre es aber, die Eingabe zeilenweise lesen zu können, also darauf zu warten, dass der Benutzer die ↵-Taste drückt, und dann die gesamte Eingabe in einem Schritt zu lesen. Dies leistet die Klasse `java.io. BufferedReader`, die diese Funktionalität mit der Methode `readLine` zur Verfügung stellt. Der Code, mit dem Sie an die Benutzereingabe gelangen, sieht also insgesamt folgendermaßen aus:

```
package de.kaiguenster.javaintro;
import java.io.*;
public class StarTrek {
    public static void main(String[] args) throws IOException {
        BufferedReader reader = new BufferedReader(
              new InputStreamReader(System.in));
```

```
        …
        String eingabe = reader.readLine();
        …
    }
}
```

Listing 3.4 Benutzereingabe lesen

Sie können dieses Beispiel als Vorlage für Ihre eigenen Klassen verwenden. Es wird Ihnen nicht entgangen sein, dass, abweichend von den vorherigen Beispielen, die Deklaration der `main`-Methode um den Zusatz `throws IOException` erweitert wurde. Dies ist notwendig, da die `readLine`-Methode eine `IOException` werfen kann, also einen Ein-/Ausgabefehler. Diese ist, im Gegensatz zu den anderen Exceptions, die Sie bislang gesehen haben, eine *Checked Exception*, ein Fehler, der in der Methodensignatur angegeben werden **muss**. Mehr dazu erfahren Sie in Kapitel 9 beim Thema Fehler und Fehlerbehandlung; für den Moment genügt es, zu wissen, dass Sie `throws IOException` angeben müssen, wenn Sie von der Standardeingabe lesen.

Ein letzter Hinweis, bevor Sie mit der Aufgabe loslegen: Erinnern Sie sich an die Methode `Integer.parseInt()`, die einen `String` in eine Zahl umwandelt. Die Eingabe sollte dafür in herkömmlichen arabischen Ziffern erfolgen, nicht wie in den Filmtiteln in römischen Zahlen. Diese sind etwas aufwendiger zu verarbeiten, und Sie werden sich später im Buch noch damit beschäftigen.

Und nun versuchen Sie sich an der Lösung Ihrer ersten Programmieraufgabe. Und bitte versuchen Sie es, bevor Sie bis zur Lösung weiterlesen.

Die Lösung

Da es sich um die erste Aufgabe zum Themengebiet handelt, finden Sie eine ausführliche Lösung direkt hier statt im Anhang. Betrachten Sie zunächst den Code der Beispiellösung:

```
package de.kaiguenster.javaintro;
import java.io.*;

public class StarTrek {
    public static void main(String[] args) throws IOException {
        BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));
        System.out.println("Welchen Star-Trek-Film willst du anschauen?");
        int film = Integer.parseInt(reader.readLine());
        if (film % 2 == 0){
            System.out.println("Cool! Los geht's.");
        } else {
```



```
            System.out.println("Nein, tu es nicht!");
        }
    }
}
```

Listing 3.5 Welcher Star-Trek-Film ist sehenswert? Eine Beispiellösung

Dies ist natürlich nur eine mögliche Lösung. Wenn Ihre Lösung anders aussieht, aber die Anforderungen erfüllt, ist sie deswegen nicht falsch, vorausgesetzt natürlich, Sie haben ein `if-else`-Statement verwendet, denn darum ging es ja schließlich. Vieles war bereits im Beispielcode zum Lesen der Benutzereingabe vorgegeben, der interessante Teil ist das `if`-Statement. Um zu prüfen, ob die Zahl gerade oder ungerade ist, kommt der Modulo-Operator zum Einsatz: Offensichtlich hat eine gerade Zahl geteilt durch 2 den Teilungsrest 0, eine ungerade Zahl den Rest 1. Als Bedingung für das `if`-Statement formuliert heißt das: `film % 2 == 0` (oder natürlich `film % 2 == 1`, wenn Sie die beiden Codeblöcke vertauschen).

In jedem Codeblock wird eine Ausgabe gemacht, die dem Benutzer mitteilt, ob er dabei ist, einen Fehler zu begehen, oder nicht.

3.1.2 Mehrfache Verzweigungen

Mit `if-else` lassen sich auch schon Mehrfachverzweigungen umsetzen. Dazu geben Sie in der `else`-Klausel wieder ein `if`-Statement an:

```
if (Bedingung1){
    Block1
} else if (Bedingung2){
    Block2
} else if (Bedingung3){
    Block3
} else {
    else-Block
}
```

Listing 3.6 Mehrfachverzweigung mit »else if«

Wenn Sie auf diese Art mehrere `if`-Statements verketten, dann ist sichergestellt, dass nur genau einer der gegebenen Codeblöcke ausgeführt wird. Trifft `Bedingung1` zu, wird `Block1` ausgeführt, dann wird die Programmausführung nach dem abschließenden `else`-Block fortgesetzt. Trifft `Bedingung1` nicht zu, wird `Bedingung2` geprüft usw. Nur wenn keine der Bedingungen zutraf, wird der finale `else`-Block ausgeführt. Bei dieser Konstruktion ist es wichtig, Ihre Bedingungen in der richtigen Reihenfolge anzuordnen. Betrachten Sie folgendes Beispiel:

```
int i = …
if (i > 1){
   System.out.println("i > 1");
} else if (i > 2){
    System.out.println("i > 2");
} else if …
```

Listing 3.7 Falsche Reihenfolge von »if«-Prüfungen

In diesem Codefragment ist die Reihenfolge problematisch: Der zweite Codeblock wird niemals ausgeführt. Da jede Zahl größer als 2 auch größer als 1 ist, wird der erste Block ausgeführt, und die weiteren Bedingungen werden gar nicht mehr geprüft.

3.1.3 Übung: Body-Mass-Index

In dieser Übung sollen Sie einen BMI-(Body-Mass-Index-)Rechner implementieren. Dazu müssen zunächst Größe und Gewicht des Benutzers abgefragt werden. Aus diesen Angaben können Sie den BMI nach der Formel $BMI = Gewicht \div Größe^2$ berechnen. Sie benötigen das Gewicht in Kilogramm und die Größe in Metern, aber es ist etwas einfacher, die Größe in Zentimetern eingeben zu lassen und umzurechnen, so müssen Sie sich nicht um das Eingabeformat der Dezimalzahl kümmern. Das Programm soll den berechneten BMI und eine Einordnung nach Tabelle 3.1 ausgeben.

BMI	Kategorie
< 18,5	Untergewicht
18,5–25	Normalgewicht
25–30	Übergewicht
> 30	schweres Übergewicht

Tabelle 3.1 BMI-Kategorien

Die Kategorisierung soll natürlich mit einem `if-else`-Statement realisiert werden. Denken Sie auch darüber nach, welche Datentypen Sie verwenden sollten, und bedenken Sie, welche Ergebnistypen Ihre Berechnungen haben. Die Lösung zu dieser Übung finden Sie im Anhang.

3.1.4 Der ternäre Operator

Neben dem `if`-Statement gibt es ein weiteres Sprachkonstrukt für Entweder-oder-Entscheidungen: Der sogenannte *ternäre Operator* heißt so, weil er drei Operanden statt der üblichen zwei benötigt.


```
Bedingung ? Wert1 : Wert2
```

Listing 3.8 Der ternäre Operator

Die Bedingung wird zu einem `boolean`-Wert ausgewertet. Ist dieser Wert `true`, so ist der Wert des gesamten Ausdrucks `Wert1`, ansonsten `Wert2`. Dass der Ausdruck einen Wert hat, unterscheidet ihn vom `if`-Statement. Er kann dadurch in einen größeren Ausdruck eingebettet werden, was mit `if` nicht gelingt. Der ternäre Operator kann dafür nur zwischen zwei Werten auswählen, nicht wie `if` zwischen zwei Codeblöcken.

Beide Konstrukte können also dieselben Entscheidungen treffen, haben aber unterschiedliche Einsatzgebiete.

```
int maximum = a > b ? a : b;
```

Listing 3.9 Die größere Zahl auswählen mit ternärem Operator

```
System.out.println("Die Zahl ist " + (zahl % 2 == 0 ? "gerade" : "ungerade"));
```

Listing 3.10 Ausgabe gerade/ungerade mit ternärem Operator

Beide Beispiele ließen sich natürlich auch mit `if-else` umsetzen. Das gilt für alle Anwendungen des ternären Operators, er kann immer durch `if-else` ersetzt werden. Allerdings wird der Code dadurch länger und unübersichtlicher.

```
int maximum;
if (a > b){
    maximum = a;
} else {
    maximum = b;
}
```

Listing 3.11 Die größere Zahl auswählen mit »if«-»else«

Aus einer Zeile werden sechs, und es ist so nicht mehr auf den ersten Blick erkennbar, dass der Zweck des Codes darin besteht, der Variablen `maximum` einen Wert zuzuweisen. Beachten Sie auch die Deklaration von `maximum` vor dem `if`-Statement: Sie ist notwendig, damit die Variable im richtigen Scope liegt. Würden Sie die Variable erst im `if`- und `else`-Block deklarieren, könnten Sie sie außerhalb des Blocks nicht sehen. Die größere Zahl zu finden, wäre damit sinnlos, weil Ihnen das Ergebnis nicht zur Verfügung stünde.

3.2 Logische Verknüpfungen

Bisher haben Sie Entscheidungen aufgrund einer einzelnen Bedingung getroffen: Ist Variable a größer als Variable b, ist der BMI kleiner als 18,5, hat der Star-Trek-Film eine gerade oder ungerade Zahl? Aber was, wenn eine Entscheidung von mehreren Bedingungen abhängt? Genau wie für Zahlen gibt es auch für `boolean`-Werte Operatoren. Sie erlauben es, mehrere Bedingungen logisch zu einer einzigen zu verknüpfen.

3.2.1 Boolesche Operatoren

Java kennt für `boolean`-Werte vier Operationen aus der booleschen Logik: AND, OR, XOR und die Negation (siehe Tabelle 3.2).

Boolesche Operation	Java-Operator	Beschreibung
Negation	!	Negiert den Wert; aus wahr wird falsch und umgekehrt.
AND	&, &&	Der gesamte Ausdruck ist genau dann wahr, wenn beide Operanden wahr sind.
OR	|, ||	Der gesamte Ausdruck ist dann wahr, wenn einer oder beide Operanden wahr sind.
XOR (exklusives Oder)	^	Der gesamte Ausdruck ist dann wahr, wenn genau einer der Operanden wahr ist. Er ist falsch, wenn beide Operanden falsch oder beide Operanden wahr sind.

Tabelle 3.2 Boolesche Operatoren

Für die Operationen AND und OR sind jeweils zwei Operatoren angegeben, weil sie mit oder ohne »Kurzschluss« ausgeführt werden können (mehr dazu im nächsten Abschnitt).

Genau wie bei anderen Operationen können Sie das Ergebnis der Operation einer Variablen zuweisen oder es gleich anstelle einer Variablen verwenden. Formulieren Sie komplexe Bedingungen, und prüfen Sie beispielsweise, ob ein Wert innerhalb eines Bereichs liegt:


```
double bmi = …;
if (bmi >= 18.5 && bmi < 25){
    System.out.println("Sie haben Normalgewicht.");
}
```

Listing 3.12 Prüfung, ob ein Wert in einen Bereich fällt

So können Sie nun auch noch bessere Empfehlungen geben, welchen Star-Trek-Film man anschauen sollte:

```
if (film % 2 == 0 && film != 10){
    System.out.println("Cool! Los geht's.");
} else {
    System.out.println("Nein, tu es nicht!");
}
```

Listing 3.13 Nun wird auch vor »Nemesis« gewarnt.

Wie mathematische Operationen können Sie auch boolesche Operationen zu längeren Ausdrücken verketten. Wenn Sie in einem Ausdruck mehrere Operatoren verwenden, sollten Sie immer Klammern setzen: Es gibt zwar eine Rangfolge unter den booleschen Operatoren (AND wird vor OR ausgewertet), im Gegensatz zur Regel »Punkt- vor Strichrechnung« kennt aber kaum jemand sie auswendig. Setzen Sie also Klammern, und machen Sie sich und jedem, der Ihren Code liest, das Leben leichter.

```
if ((film % 2 == 0 && film != 10) || isReboot){
    …
}
```

Listing 3.14 Die neuen Star-Trek-Filme sind (bisher) alle unterhaltsam.

3.2.2 Verknüpfungen mit und ohne Kurzschluss

Wenn der erste Operand einer AND-Verknüpfung `false` ist, dann ist der Wert des zweiten Operanden egal: Der Ausdruck kann nur noch `false` werden. Dasselbe gilt, wenn der erste Operand bei OR `true` ist: Der ganze Ausdruck ist unabhängig vom zweiten Operanden `true`.

Hier liegt der Unterschied zwischen den Operatoren mit und ohne Kurzschluss. Die Operatoren mit Kurzschluss (`&&` und `||`) werten ihren zweiten Operanden nicht aus, wenn es nicht notwendig ist. Die Operatoren ohne Kurzschluss (`&` und `|`) werten immer beide Operanden aus. Für die XOR-Operation gibt es keinen Kurzschluss-Operator, da immer beide Operanden ausgewertet werden müssen.

Für den Wert des Ausdrucks macht es keinen Unterschied, ob Sie einen Operator mit oder ohne Kurzschluss verwenden. Einen Unterschied finden Sie erst dann, wenn Ihre Bedingungen Seiteneffekte haben – normalerweise erst nach mehreren Stunden Fehlersuche. Vergleichen Sie die beiden folgenden Beispiele:

```
int zahl = 2;
if (zahl % 2 != 0 & zahl++ < 5){
    …
}
```

Listing 3.15 Verknüpfung ohne Kurzschluss

```
int zahl = 2;
if (zahl % 2 != 0 && zahl++ < 5){
    …
}
```

Listing 3.16 Verknüpfung mit Kurzschluss

In beiden Fällen wird der Rumpf des `if`-Statements nicht ausgeführt, da `zahl % 2 != 0` falsch ist. Es unterscheidet sich aber der Wert, den `zahl` anschließend hat. Im ersten Beispiel hat `zahl` am Ende des Fragments den Wert 3, denn die Anweisung `zahl++` wurde ausgeführt. Im zweiten Beispiel wird die rechte Seite des `&&` nicht ausgeführt, deshalb hat `zahl` nach wie vor den Wert 2.

Am besten sollten Sie Seiteneffekte dieser Art vermeiden.

Welchen Zweck hat der Kurzschlussoperator dann überhaupt? Er kann zum Beispiel teure Operationen vermeiden, wenn ihr Ergebnis gar nicht benötigt wird.

```
if (filmImCache() || downloadFilm()){
    starteFilm();
}
```

Listing 3.17 Teure Operationen vermeiden mit Kurzschluss

Das gezeigte Beispiel aus einem hypothetischen Videoplayer veranschaulicht dies. Wenn der Film, den Sie sehen wollen – zum Beispiel ein Teil der Star-Trek-Reihe mit gerader Zahl –, schon im Cache liegt oder erfolgreich heruntergeladen wurde, dann soll er abgespielt werden. Selbstverständlich wollen Sie den Film nicht herunterladen, wenn er schon im lokalen Cache vorhanden ist. Der Kurzschlussoperator verhindert das.

In der Praxis kommen meist die Operatoren mit Kurzschluss zum Einsatz: Auch wenn auf der rechten Seite des Operators nur ein einfacher Vergleich steht, so ist es

immer noch eine Anweisung, die nicht ausgeführt werden muss, wenn sie nicht notwendig ist. Seiteneffekte sollen innerhalb einer `if`-Bedingung sowieso vermieden werden, dann kann auch mit Kurzschluss nichts schiefgehen.

3.2.3 Übung: Boolesche Operatoren

Für alle folgenden Codefragmente gilt, dass `i`, `j` und `k` `int`-Variablen mit den Werten `i` = 0, `j` = 7 und `k` = 13 sind. Welchen Wert haben die drei Variablen, nachdem die Fragmente ausgeführt wurden?

- **Fragment 1**

```
if (i > 0 || j > 5){
    k = 10;
}
```

- **Fragment 2**

```
if (i > 0 && j > 5){
    k = 10;
}
```

- **Fragment 3**

```
if ((i > 0 && j > 5) || k < 15){
    k = 10;
}
```

- **Fragment 4**

```
if ((i > 0 || j > 5) && k > 15){
    k = 10;
}
```

- **Fragment 5**

```
if (i == 0 & j++ < 5){
    k = 10;
}
```

- **Fragment 6**

```
if (i == 0 && j++ < 5){
    k = 10;
}
```

- **Fragment 7**

```
if (i != 0 && j++ < 5){
    k = 10;
}
```

- **Fragment 8**

```
if (i != 0 & j++ < 5){
    k = 10;
}
```

Die Lösung zu dieser Übung finden Sie im Anhang.

3.2.4 Übung: Solitaire

Sie sind sicher mit der Windows-Variante des Kartenspiels Solitaire vertraut. Ihre Aufgabe in diesem Spiel ist es, Karten in Stapel zu sortieren. Dabei dürfen Sie eine Karte an eine andere anlegen, wenn der Wert der neuen Karte um genau eins niedriger ist und die Farbe der beiden Karten (Rot oder Schwarz) nicht übereinstimmt. Sie dürfen also beispielsweise eine Karo 7 an eine Pik 8 anlegen.

In dieser Übung sollen Sie eine Methode entwickeln, der Kartenfarbe (Kreuz, Pik, Herz, Karo) und Wert einer »alten« und einer »neuen« Karte übergeben werden. Die Methode soll `true` zurückgeben, wenn die neue Karte an die alte Karte angelegt werden kann. Verwenden Sie dazu die abgedruckte Vorlage oder die Klasse `SolitaireRumpf` aus den Downloads zum Buch (*www.rheinwerk-verlag.de/5111*). Dort finden Sie bereits eine `main`-Methode mit einer Reihe von Prüfungen, die Ihren Algorithmus testen. (In Kapitel 7, »Unit Testing«, werden Sie eine bessere Methode kennenlernen, um diese Art automatischer Tests zu realisieren.)

Der Einfachheit halber wird für den Kartenwert eine `int`-Variable verwendet. Die Werte von 2 bis 10 entsprechen direkt den Kartenwerten, daneben gilt Ass = 1, Bube = 11, Dame = 12, König = 13. Sie können sich darauf verlassen, dass nur gültige Werte übergeben werden. Für die Kartenfarbe werden die String-Werte `"Kreuz"`, `"Pik"`, `"Herz"` und `"Karo"` verwendet.

String-Vergleiche

Wie Sie bereits im vorherigen Kapitel gelernt haben, hat es meist nicht den gewünschten Effekt, wenn Objekte mit dem `==`-Operator verglichen werden. Das gilt auch für Strings. Statt des Vergleichsoperators verwenden Sie die Methode `equals`, um Objekte zu vergleichen. Der Vergleich einer String-Variablen mit einem String-Literal sieht aus wie folgt:

```
if ("Herz".equals(stringvariable)){…}
```

Speziell bei Strings sollten Sie immer wie gezeigt vergleichen und niemals mit `stringvariable.equals("Herz")`. Die empfohlene Art zu vergleichen hat den Vorteil, dass keine Gefahr von `null`-Werten ausgeht. `stringvariable` könnte `null` enthalten, der Aufruf `stringvariable.equals("Herz")` würde dann eine `NullPointerException` verursachen. Umgekehrt verursacht `"Herz".equals(stringvariable)` keinen Fehler, auch wenn `stringvariable` `null` enthält. Der Aufruf liefert in diesem Fall das korrekte Ergebnis `false`.

In dieser Übung müssen Sie zum ersten Mal das Ergebnis eines Methodenaufrufs zurückgeben. Das geschieht mit dem Schlüsselwort `return` und ist bereits vorgegeben. Sie müssen das Ergebnis Ihrer Prüfung nur der Variablen `ergebnis` zuweisen. Mehr zu Rückgabewerten erfahren Sie in Kapitel 5, »Klassen und Objekte«. Die Lösung zu dieser Übung finden Sie im Anhang.

```
package de.kaiguenster.javaintro.solitaire;

public class SolitaireRumpf {

    public static boolean kannAnlegen(String farbeAlt, int wertAlt,
     String farbeNeu, int wertNeu){
        //Implementieren Sie hier Ihre Lösung
        boolean ergebnis = false;
        return ergebnis;
    }

    public static void pruefeKarte(String farbeAlt, int wertAlt,
     String farbeNeu, int wertNeu, boolean erwartet){
        boolean ergebnis = kannAnlegen(farbeAlt, wertAlt, farbeNeu, wertNeu);
        System.out.print(farbeNeu + " " + wertNeu + " auf " + farbeAlt + " "
          + wertAlt + " ");
        System.out.print(ergebnis ? "ist möglich " : " ist nicht möglich");
        if (erwartet == ergebnis){
            System.out.println(" ===> RICHTIG");
        } else {
            System.out.println(" ===> FALSCH");
        }
    }

    public static void main(String[] args) {
        pruefeKarte("Herz", 8, "Pik", 7, true);
        pruefeKarte("Herz", 9, "Pik", 7, false);
        pruefeKarte("Herz", 8, "Kreuz", 7, true);
        pruefeKarte("Karo", 6, "Pik", 7, false);
        pruefeKarte("Herz", 2, "Pik", 1, true);
        pruefeKarte("Karo", 8, "Herz", 7, false);
        pruefeKarte("Karo", 6, "Herz", 7, false);
```

```
        pruefeKarte("Herz", 11, "Kreuz", 10, true);
        pruefeKarte("Pik", 8, "Karo", 7, true);
        pruefeKarte("Kreuz", 7, "Pik", 5, false);
    }
}
```

Listing 3.18 Die Vorlage für die »Solitaire«-Klasse

3.3 Mehrfach verzweigen mit »switch«

Neben `if … else if …` gibt es in Java mit dem `switch`-Statement eine weitere Möglichkeit, Mehrfachverzweigungen umzusetzen. `switch` verzweigt nach dem Wert einer einzelnen Variablen. Für jeden erwarteten Wert wird ein Codeblock angegeben.

```
switch (variable){
    case wert1:
        anweisung1;
        anweisung2;
        break;
    case wert2:
        …
        break;
    case wert3:
        …
        break;
    …
    default:
        …
}
```

Listing 3.19 Aufbau des »switch«-Statements

Die Variable, nach der die Entscheidung getroffen werden soll, wird in Klammern nach dem Schlüsselwort `switch` angeführt. Die Syntax des `switch`-Statements unterscheidet sich leider von der in Java sonst üblichen Syntax. Die verschiedenen Codezweige werden nicht jeweils von einem Paar geschweifter Klammern umschlossen, sondern der gesamte Rumpf des `switch`-Statements steht in einem Paar geschweifter Klammern, und die verschiedenen Zweige werden durch das Schlüsselwort `case` und einen möglichen Variablenwert getrennt. Eine wichtige Einschränkung ist dabei, dass die `case`-Werte keine Variablen sein dürfen, es müssen konstante Werte verwendet werden.

Die Ausführung beginnt mit dem `case`, dessen Wert dem Variablenwert entspricht. Sie endet aber nicht automatisch dort, wo der nächste `case` beginnt. `switch` muss explizit mit dem Befehl `break` unterbrochen werden, oder es führt auch alle weiteren Fälle aus, die nach dem richtigen Fall noch folgen. Das klingt zunächst nur verwirrend und unnütz, aber in Abschnitt 3.3.4, »Durchfallendes »switch««, werden Sie sehen, dass es für dieses eigenartige Verhalten durchaus sinnvolle Anwendungen gibt.

Am Ende kann noch ein `default`-Block stehen, der ausgeführt wird, falls keiner der anderen Fälle zutrifft.

3.3.1 »switch« mit Strings, Zeichen und Zahlen

Das `switch`-Statement funktioniert nicht mit jedem Datentyp; die Variable, anhand derer entschieden wird, muss eine Ganzzahl (`byte`, `short`, `int`, aber *nicht* `long`), ein Zeichen (`char`), ein enumerierter Datentyp (`enum`, siehe Listing 3.22) oder, seit Java 7, ein `String` sein. Fließkommazahlen können mit `switch` nicht verwendet werden, da sie, wie oben beschrieben, nicht problemlos auf Gleichheit geprüft werden können, und für boolesche Werte wäre `switch` sinnlos, weil es nur zwei mögliche Werte gibt.

Mit allen diesen Datentypen lassen sich Entscheidungen treffen, die auf einzelnen Werten beruhen. Zum Beispiel können Sie im oben erklärten Solitaire-Spiel die Zahlenwerte für Bildkarten wieder in den echten Kartenwert umsetzen und so den Namen einer Karte bestimmen:

```
public static String kartenName(String farbe, int wert){
    String name = farbe + " ";
    switch(wert){
        case 1:
            name += "As";
            break;
        case 11:
            name += "Bube";
            break;
        case 12:
            name += "Dame";
            break;
        case 13:
            name += "König";
            break;
        default:
            name += wert;
```

```
    }
    return name;
}
```

Listing 3.20 Kartennamen berechnen mit »switch«

Der gezeigte Code tut nur das Nötigste: Für Ass (1), Bube (11), Dame (12) und König (13) wird jeweils ein Codezweig definiert, der den Kartenwert an den `String` hängt, der bereits die Farbe enthält. Das Ergebnis lautet zum Beispiel `"Karo Dame"`. Für alle Werte, die direkt dem Kartenwert entsprechen (die Zahlen 2–10), wird im `default`-Zweig die Zahl an den String gehängt.

Mit `char`-Werten ist `switch` gut dafür geeignet, die Tastatursteuerung für ein Spiel umzusetzen:

```
switch(eingabe){
    case 'w': vorwaerts(); break;
    case 'a': links(); break;
    case 's': rueckwaerts(); break;
    case 'd': rechts(); break;
}
```

Listing 3.21 Tastendruck verarbeiten mit »switch«

In diesem Fall fehlt der `default`-Fall, denn wenn der gedrückten Taste kein Kommando zugeordnet ist, dann muss auch nichts getan werden. Die Technik eignet sich leider nicht für Spiele, in denen der Spieler die Tastenbelegung ändern kann, da die einzelnen `case` mit Konstanten identifiziert werden müssen, aber für viele einfache Spiele reicht der Code vollkommen aus.

3.3.2 Übung: »Rock im ROM«

Jedes Jahr findet eine Woche lang das berühmte Musikfestival »Rock im ROM« statt. Jeden Tag spielt dort ein anderer Headliner, und jeder Tag hat einen anderen Eintrittspreis. Tabelle 3.3 zeigt das Programm in diesem Jahr:

Tag	Headliner	Preis
Montag	Rage against the Compiler	37,50 €
Dienstag	if/else	22 €
Mittwoch	The Falling Cases	17,50 €

Tabelle 3.3 »Rock im ROM 2014« – das Programm

Tag	Headliner	Preis
Donnerstag	Blinkenlichten	21 €
Freitag	Compilaz	32,55 €
Samstag	Real Class	45 €
Sonntag	Delphi and the Oracles	35 €

Tabelle 3.3 »Rock im ROM 2014« – das Programm (Forts.)

Für diese Aufgabe sollen Sie ein Programm schreiben, das vom Benutzer abfragt, für welchen Tag er eine Karte kaufen möchte. Aktuell gibt es nur Tageskarten. Das Programm soll ausgeben, wer am ausgewählten Tag Headliner ist und wieviel die Karte kostet. Die Lösung zu dieser Übung finden Sie im Anhang.

3.3.3 Enumerierte Datentypen und »switch«

Neben den oben genannten gibt es eine ganze Gruppe von weiteren Datentypen, die Sie für das `switch`-Statement verwenden können: *enumerierte Datentypen*, sogenannte *Enums*. Enums sind Objekttypen, die nur eine definierte Menge von Werten annehmen können. Ihre Werte sind also aufzählbar oder auch enumerierbar.

Enumerierte Typen haben den Vorteil, dass keine ungültigen Werte übergeben werden können. Denken Sie an das Solitaire-Beispiel zurück. Kartenfarben wurden dort als Strings übergeben, Sie hätten in einem Anfall plötzlicher Verwirrung also auch `"Eichel"` oder `"Schellen"` übergeben können. Das hätte mit Sicherheit Fehler zur Folge gehabt. Um diese Möglichkeit sicher auszuschließen, könnten Sie die gültigen Farben als `enum` definieren:

```
public enum Farbe {
    KREUZ, PIK, HERZ, KARO;
}
```

Listing 3.22 Eine ganz einfache Enum

Enums werden normalerweise, genau wie Klassen, in einer eigenen Datei definiert, die den Namen der `enum` tragen muss, zum Beispiel *Farbe.java*. Sie enthalten im einfachen Fall nichts anderes als eine Aufzählung aller möglichen Werte, die nach Konvention in Großbuchstaben geschrieben werden. Die Werte einer `enum` sind Objekte, Sie können sie verwenden wie jedes andere Objekt.

```
Farbe alteFarbe = Farbe.KREUZ;
```

Warum sind enumerierte Datentypen an dieser Stelle interessant? Weil auch mit ihnen das `switch`-Statement funktioniert. Sie können also eigene Datentypen definieren, mit denen Sie »switchen« können.

```
Farbe kartenfarbe = …;
switch (kartenfarbe){
    case KREUZ: …; break;
    case PIK: …; break;
    …
    default: …;
}
```

Listing 3.23 »switch« mit enumerierten Typen

Mehr Details zu Enums finden Sie in Abschnitt 6.5, »Enumerationen«.

3.3.4 Durchfallendes »switch«

Eine letzte Erklärung zum `switch`-Statement fehlt noch: Warum muss jeder `case` mit einem `break` unterbrochen werden? Wieso wird ohne diese Unterbrechung der nächste Fall ausgeführt, anstatt beim nächsten `case` automatisch abzubrechen? Welchen Vorteil hat es, dass in Java die Ausführung durch die Cases »durchfällt«?

Wie weiter oben bereits angekündigt, gibt es Fälle, in denen genau dieses Verhalten nützlich ist, zum Beispiel wenn mehrere Möglichkeiten zum gleichen Verhalten führen sollen.

```
switch(eingabe){
    case "ja": //fällt durch
    case "j": //fällt durch
    case "yes": //fällt durch
    case "y": bestaetigen(); break;
    case "nein": //fällt durch
    case "no": //fällt durch
    case "n": abbrechen(); break;
}
```

Listing 3.24 Durchfallende Switches: mehrere Eingaben

Im Beispiel sehen Sie, dass alle Fälle, in denen die Ausführung durchfällt, mit einem entsprechenden Kommentar markiert sind. Wenn Sie mit durchfallenden Cases arbeiten, ist das eine enorm hilfreiche Praxis; sie sorgt für Übersicht und unterscheidet absichtliches Durchfallen klar von unabsichtlichem, wenn ein `break` einfach vergessen wurde.

Eine weitere Anwendung für diese Technik ist es, in einer Reihe von Aktionen an einer beliebigen Stelle einzusteigen, aber alle folgenden Aktionen bis zu einem definierten Endpunkt auszuführen, so zum Beispiel in der nächsten Übung.

3.3.5 Übung: »Rock im ROM« bis zum Ende

Zurück zum Musikfestival »Rock im ROM«. Nehmen Sie für diese Übung an, dass alle Besucher bis zum Ende bleiben. Interpretieren Sie die Eingabe des Benutzers als seinen Anreisetag. Auch dafür sind durchfallende Switches sehr nützlich: Sie können nun alle Headliner, die ein Besucher sehen wird, mit String-Konkatenation zusammenschreiben und seinen Gesamtpreis berechnen.

Geben Sie die Bandnamen jeweils auf einer eigenen Zeile aus. Um im String einen Zeilenumbruch zu erzeugen, verwenden Sie die Zeichenkombination `\n`. Geben Sie alle Bandnamen aus und danach, in einer neuen Zeile, den Gesamtpreis. Die Lösung zu dieser Übung finden Sie im Anhang.

3.3.6 Übung: »Rock im ROM« solange ich will

Das Programm von »Rock im ROM« sieht jetzt schon gut aus, aber nicht jeder ist ein Fan der Compilaz oder von Delphi and the Oracles. Es sollte deshalb auch noch möglich sein, ein Ticket von einem beliebigen Tag bis zu einem beliebigen Tag zu bekommen.

Das ist nicht so schwierig, wie es sich zunächst anhört. Sie müssen lediglich wissen, dass auch die `break`-Anweisung mit `if` bedingt ausgeführt werden kann. Sie können also immer noch mit `switch` beim Starttag einsteigen und dann mit einem bedingten `break` am Endtag abbrechen. Die Lösung zu dieser Übung finden Sie im Anhang.

3.3.7 Der Unterschied zwischen »switch« und »if… else if …«

Warum gibt es in Java zwei Konstruktionen, die Entscheidungen treffen, `if` und `switch`? Alles, was Sie mit `switch` tun können, könnten Sie auch mit `if` realisieren.

Gerade bei durchfallenden Switches und bedingten Unterbrechungen führt es aber zu langem und unübersichtlichem Code, wenn sie mit `if-else` umgesetzt werden, weil Sie viele Möglichkeiten mit OR verknüpft angeben müssen. Versuchen Sie, die letzte Variante des Programms von »Rock im ROM« mit `if` umzusetzen statt mit `switch`, und Sie werden schnell sehen, wo das Problem liegt.

Über den kurzen und lesbaren Code hinaus gibt es aber noch einen weiteren Unterschied: Für die Anwendungsfälle, für die `switch` gedacht ist, ist es schlicht schneller als `if`. Mit `if`-Statements muss für jede Bedingung wieder geprüft werden, ob sie wahr oder falsch ist. Bei einem `switch`-Statement erzeugt der Compiler eine soge-

nannte Sprungtabelle, in der für jeden `case` steht, wo die Ausführung fortgesetzt werden muss. Mit dem Wert, nach dem entschieden wird, muss auf diese Weise nur einmal die Sprungadresse in der Tabelle gefunden werden. Der Geschwindigkeitsunterschied ist allerdings, zugegeben, so gering, dass er für mehr als 99 % aller Java-Programme keinen Unterschied macht.

3.4 Zusammenfassung

Mit `if`, `switch` und dem ternären Operator haben Sie in diesem Kapitel drei Möglichkeiten kennengelernt, die Programmausführung zu verzweigen. Sie haben die Einsatzgebiete der verschiedenen Konstrukte gesehen und wie Sie mit logischen Operatoren mehrere Bedingungen zu einer verknüpfen.

Im nächsten Kapitel werden Sie sehen, wie Sie Codeblöcke in Java beliebig oft wiederholen können.

Kapitel 10
Arrays und Collections

Nur in sehr wenigen Anwendungen kommen Sie mit einzelnen Objekten aus: Früher oder später gelangen Sie immer an einen Punkt, an dem Sie eine ganze Liste von gleichartigen Objekten brauchen. Die verschiedenen Möglichkeiten, solche Listen von Objekten in Java darzustellen, sind Gegenstand dieses Kapitels.

In fast jeder Anwendung finden sich Stellen, an denen Sie nicht mit einem einzelnen Objekt arbeiten möchten, sondern eine ganze Reihe von gleichartigen Objekten benötigen. Mehr noch, Sie wissen in diesen Fällen auch nicht, mit wie vielen Objekten Sie es letztlich zu tun haben werden, so dass Sie keine Variablen mit Namen wie `objekt1`, `objekt2` und `objekt3` anlegen können. Einmal abgesehen von vielen anderen Nachteilen, die dieser Ansatz brächte.

Beispiele für diese Art von Problem sind schnell gefunden. Schon eine einfache Anwendung, die einen Einkaufszettel erstellt, benötigt einen Weg, mehrere und potenziell unbegrenzt viele Einträge aufzunehmen. Dasselbe gilt für eine Liste von Adressen und Geburtstagen Ihrer Freunde, für die Verbindungen bei einer Bahnreise mit Umstiegen oder, um einmal mehr zum `Musicplayer` aus früheren Kapiteln zurückzukommen, für eine Playlist, die beliebig viele Songs aufnehmen kann.

Für alle diese Aufgaben gibt es in Java zwei verschiedene Ansätze, mit der Pluralität der Daten umzugehen: *Arrays* und *Collections*. In diesem Kapitel werden wir beide genauer betrachten.

Alle hier vorgestellten Klassen haben über die hier vorgestellten Methoden hinaus eine lange Liste weiterer Methoden, die Lambda-Ausdrücke verwenden. Es handelt sich dabei um Komfortmethoden, die Sie sich ansehen sollten, nachdem Sie im nächsten Kapitel alles über Lambdas gelernt haben.

10.1 Arrays

Von den beiden Möglichkeiten, Datenlisten darzustellen, sind Arrays die primitivere. Sie haben ein weniger umfangreiches Interface als die `Collection`-Klassen, die Sie später kennenlernen werden, und sind in vielen Anwendungsfällen etwas umständlicher zu benutzen. Andererseits sind sie dadurch für manche Anwendungen perfor-

manter und haben einige Sprachmittel, die speziell auf Arrays zugeschnitten sind. So kommen zum Beispiel bei variablen Parameterlisten (siehe Abschnitt 10.3) Arrays zum Einsatz, nicht Collections. Außerdem können nur Arrays Primitive aufnehmen, während eine Collection Zahlen, Zeichen und Wahrheitswerte nur in Form ihrer Wrapper-Objekte verwenden kann, was zu einem schlechteren Speicherverhalten und langsameren Zugriffen führt. Wenn Sie also zeitkritische Berechnungen auf langen Listen von Zahlen durchführen müssen, dann ist das Array der Datentyp Ihrer Wahl.

10.1.1 Grundlagen von Arrays

Sie kennen die Grundlagen von Arrays bereits aus früheren Kapiteln, dennoch sollen sie hier noch einmal wiederholt werden. Ein Array, im deutschen auch *Feld* genannt, ist eine Liste gleichartiger Daten. Dabei hat ein Array eine bei der Instanziierung festgelegte Länge, die sich nachträglich nicht mehr ändern lässt.

Arrays instanziieren

```
int[] zahlen = new int[1000];
Song[] musiksammlung = new Song[500];
```

Listing 10.1 Arrays erzeugen

Der Typ der Variablen im Beispiel ist `int[]` (sprich `int`-Array) bzw. `Song[]`. Die Länge des Arrays ist dabei vom Typ unabhängig, die Variable `zahlen` kann sowohl ein `int[]` der Länge 0 als auch ein `int[]` der Länge `Integer.MAX_VALUE - 8` enthalten. Das ist kein Widerspruch zur Aussage oben, dass Sie die Länge eines Arrays nicht nachträglich ändern können, denn durch die erneute Zuweisung ändern Sie nicht die Länge des bestehenden Arrays, sondern erzeugen ein neues Array, das vom alten Array vollkommen unabhängig ist. Die Array-Variable kann ein Array beliebiger Länge referenzieren, da die Länge kein Teil des Typs ist.

Die Java-Syntax erlaubt es auch, die eckigen Klammern an den Variablennamen zu hängen statt an den Typ: `int zahlen[] = new int[100]`. Von dieser Variante ist aber mit Nachdruck abzuraten, denn der Typ der Variablen ist `int[]`, nicht `int`, und das sollte auch in der Deklaration klar sichtbar sein.

Maximale Länge von Arrays

Die Maximallänge von Arrays hängt von Ihrer JVM ab, liegt aber immer nahe bei `Integer.MAX_VALUE`. Es gibt leider keine zuverlässige Methode, um herauszufinden, wie groß ein Array maximal sein darf. Da der Versuch, ein zu großes Array zu allokieren (was so viel heißt, wie Speicher dafür zu reservieren), einen `OutOfMemoryError` produziert, scheidet auch einfaches Ausprobieren als Möglichkeit aus, denn wie Sie im vorigen Kapitel gelernt haben, gibt es nach einem Error keine Garantie dafür, dass die JVM stabil weiterläuft. Der beste Hinweis auf die maximale sichere Array-Größe stammt aus der Klasse `java.util.ArrayList` der Standardbibliothek:

```
private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;
```


Ein so erzeugtes Array enthält an jeder Position den Initialwert einer Variablen dieses Typs, also 0 für alle Zahlentypen, `false` für `boolean` und `null` für Objekttypen.

Bei der Instanziierung eines Arrays können Sie auch alle Werte des Arrays angegeben, so dass Sie nicht anschließend jeder Array-Position einzeln einen Wert zuweisen müssen. Dazu führen Sie die Werte in geschweiften Klammen an:

```
int[] zahlen = new int[]{1, 2, 3, 4};
```

Listing 10.2 Array mit Initialwerten instanziieren

Wenn Sie Initialwerte für das Array angeben, dann entfällt die Längenangabe in den eckigen Klammern, die Länge entspricht genau der Anzahl übergebener Initialwerte.

Speicherverwaltung von Arrays

Dass Sie die Länge eines Arrays nachträglich nicht ändern können, liegt vor allem daran, wie Arrays ihren Speicher verwalten. Wenn Sie ein Array anlegen, dann wird exakt so viel Speicher allokiert, wie für die angegebene Anzahl an Werten des deklarierten Datentyps benötigt wird. Ein `long[100]` belegt demnach 800 Byte, denn ein `long`-Wert umfasst 8 Byte, und es werden 100 davon benötigt. Dieser Speicher wird als zusammenhängender Block allokiert, wodurch der Zugriff auf ein Array-Element extrem effizient ist: Um auf das n-te Element eines Arrays zuzugreifen, wird die passende Speicherstelle berechnet als *Startadresse des Arrays + n × Größe des Datentyps*.

In dieser Hinsicht verhalten sich Objekt-Arrays anders als Arrays von primitiven Typen: Ein Objekt-Array enthält in seinem Speicherblock nicht das Objekt selbst, sondern einen Zeiger auf das Objekt, so dass der Zugriff auf ein in einem Array gespeichertes Objekt ein wenig langsamer ist als der Zugriff auf einen Primitivtyp. Dieses Vorgehen ist aber notwendig, da Objekte eben keine festgelegte Größe haben.

Zugriff auf Arrays

Auch der Zugriff auf Arrays erfolgt durch eckige Klammern. Um auf einen Eintrag des Arrays zuzugreifen, lesend oder schreibend, geben Sie den gewünschten Index in eckigen Klammern an:

```
Song[] topTen = new Song[10]{new Song(…), …};
topTen[3] = new Song("Paint it Black", "The Rolling Stones", 239);
if ("Queen".equals(topTen[0].getInterpret())){
    System.out.println("Die Welt ist noch in Ordnung");
}
```

Listing 10.3 Array-Zugriff lesend und schreibend

Ein Array-Eintrag verhält sich also aus Ihrer Sicht als Programmierer genauso wie eine einfache Variable. Sie können mit = Werte zuweisen, mit == Werte vergleichen, primitive Array-Einträge in Berechnungen verwenden und auf Felder und Methoden von Einträgen in Objekt-Arrays mit dem Punktoperator zugreifen.

Beachten Sie dabei stets, dass der erste Eintrag eines Arrays den Index 0 hat und der letzte Eintrag den Index Länge – 1. Jeder Versuch, auf einen Index außerhalb dieses Bereichs zuzugreifen, führt zu einer `IndexOutOfBoundsException`.

Die Länge eines Arrays können Sie aus dem Feld `length` auslesen. Der Zugriff auf `einArray[einArray.length - 1]` ist somit immer der Zugriff auf das letzte Array-Element, egal, wie lang das Array ist. So ergibt sich die Ihnen ebenfalls schon bekannte Schleife, die über alle Werte eines Arrays iteriert.

```
for (int i = 0; i < zahlen.length; i++){
    int zahl = zahlen[i];
    …
}
```

Listing 10.4 Iterieren über ein Array

Im weiteren Verlauf dieses Kapitels werden Sie noch eine andere Art der `for`-Schleife kennenlernen, die diesen einfachen Fall kompakter und lesbarer ausdrückt.

10.1.2 Übung: Primzahlen

Eine effiziente Methode, alle Primzahlen unterhalb einer vorgegebenen Grenze zu finden, ist das nach dem griechischen Mathematiker Eratosthenes von Kyrene benannte *Sieb des Eratosthenes*. Dabei schreiben Sie zunächst alle Zahlen auf, die kleiner als die Grenze sind. Dann gehen Sie diese Liste durch, angefangen bei der 2, der per Definition kleinsten Primzahl. Für jede Zahl, die Sie bearbeiten, streichen Sie alle Vielfachen dieser Zahl aus. Bei der 2 streichen Sie also 4, 6, 8, 10 … Ist eine Zahl bereits ausgestrichen, müssen Sie sie nicht weiter beachten, denn dann sind alle ihre Vielfachen ebenfalls gestrichen. Haben Sie alle Zahlen bearbeitet, dann sind die übrigen, nicht ausgestrichenen Zahlen Primzahlen, denn sie waren durch keine andere Zahl teilbar.

Implementieren Sie das Sieb des Eratosthenes in Java auf Basis von Arrays. Ihre Implementierung soll die Obergrenze als Parameter erhalten und ein `int[]` zurückgeben, das genau alle Primzahlen enthält, die kleiner oder gleich dieser Grenze sind. Die Obergrenze soll dabei maximal 100.000 betragen, denn wenn Sie alle Werte bis `Integer.MAX_VALUE - 8` erlauben, dann müssen Sie die Speichereinstellungen des Java-Prozesses anpassen: Das Array würde 4 GB Speicher belegen. Die Lösung zu dieser Übung finden Sie im Anhang.

10.1.3 Mehrdimensionale Arrays

Arrays müssen keine flachen Listen von Werten sein, sie können sich auch in mehr Dimensionen erstrecken. So können Sie statt einer Liste von Daten auch ein Gitter darstellen, zum Beispiel ein Schachbrett:

```
Schachfigur[][] schachbrett = new Schachfigur[8][8];
```

Listing 10.5 Ein zweidimensionales Array

Zwei Paare von eckigen Klammern erzeugen ein Array in zwei Dimensionen. Für beide Dimensionen können Sie eine Größe angeben, und das Gitter muss nicht wie im Beispiel quadratisch sein. Sie sind auch nicht auf zwei Dimensionen beschränkt, höherdimensionale Arrays sind ohne weiteres möglich. Ein `int[][][][][][][][]` wird aber schon etwas unübersichtlich, und das Maximum von 255 Dimensionen werden Sie eher selten erreichen.

Auf den Wert eines mehrdimensionalen Arrays können Sie genauso einfach zugreifen wie auf den eines eindimensionalen, Sie müssen nur die genauen Koordinaten angeben:

```
schachbrett[2][1] = new Schachfigur(Figur.BAUER, Farbe.WEISS);
```

Listing 10.6 Zugriff auf ein mehrdimensionales Array

Sie können aber auch auf eine Zeile aus dem zweidimensionalen Array zugreifen. So erhalten Sie ein eindimensionales Array desselben Typs.

```
int[] grundreiheWeiss = schachbrett[1];
int[] grundreiheSchwarz = schachbrett[6];
```

Listing 10.7 Eine Zeile aus einem mehrdimensionalen Array auslesen

Analog können Sie auch einer Zeile des Arrays einen neuen Wert zuweisen. So können mehrdimensionale Arrays entstehen, die nicht rechteckig (oder das höherdimensionale Äquivalent dazu) sind.

```
int[][] dreieck = new int[3][];
dreieck[0] = new int[1];
dreieck[1] = new int[2];
dreieck[2] = new int[3];
```

Listing 10.8 Nicht rechteckiges, zweidimensionales Array

Die Dimensionalität eines Arrays ist im Gegensatz zur Länge ein Teil des Datentyps. `int[]` und `int[][]` sind unterschiedliche Typen und nicht zuweisungskompatibel. Die Anweisung `int[][] zahlen = new int[5]` führt zu einem Compilerfehler.

10.1.4 Übung: Das Pascalsche Dreieck

Sie erinnern sich vielleicht an das *Pascalsche Dreieck* aus Ihrer Schulzeit, das die Koeffizienten für binomische Gleichungen wie $(a + b)^3 = a^3 + 3a^2b + 3ab^2 + b^3$ angibt. Die Koeffizienten sind in diesem Fall 1, 3, 3, 1. Dabei ist die Bildungsvorschrift für das pascalsche Dreieck sehr einfach: Um eine Zeile zu berechnen, summieren Sie jeweils zwei benachbarte Zahlen der Vorzeile und schreiben das Ergebnis in die neue Zeile, zwischen die beiden Zahlen.

```
      1
    1  1
  1  2  1
1  3  3  1
  1  4  6  4  1
1  5  10  10  5  1
      ...
```

Sie können nun das pascalsche Dreieck in fast beliebiger Größe als zweidimensionales Array berechnen. Schreiben Sie eine Klasse `PascalsDreieck`. Übergeben Sie dem Konstruktor, wie viele Zeilen des Dreiecks berechnet werden sollen, und berechnen Sie das gesamte Dreieck bis zu dieser Zeile. Eine Methode `getZeile` soll den Zugriff auf eine einzelne Zeile erlauben. Als Bonusaufgabe schreiben Sie eine `toString`-Methode, die das berechnete Dreieck in ungefährer Dreiecksform ausgibt. Die Lösung zu dieser Übung finden Sie im Anhang.

10.1.5 Utility-Methoden in »java.util.Arrays«

Arrays haben zwar selbst kein umfangreiches Interface, aber die Utility-Klasse `Arrays` bietet einige statische Methoden für erweiterte Funktionalität »von außerhalb« an.

Sie sehen hier eine Auswahl aus den angebotenen Werkzeugen von `Arrays`; es existieren weitere Methoden, die die Lambdas aus dem nächsten Kapitel verwenden.

Suchen und sortieren

Ein wichtiges Anliegen an Listen von Werten ist es immer, sie sortieren und in ihnen suchen zu können. `Arrays` bietet die Methoden `sort` zum Sortieren und `binarySearch` zum Suchen an, jeweils in überladenen Varianten für die verschiedenen Array-Typen.

Bei der Suche in Arrays ist immer zu beachten, dass `binarySearch` nur auf sortierten Arrays funktioniert. Auf einem unsortierten Array ist das Ergebnis der Suche nicht definiert.

Das Sortieren von primitiven Arrays ist einfach und definiert, denn alle primitiven Werte haben eine natürliche Ordnung: 1 ist immer kleiner als 17, a ist immer kleiner als z.

Anders sieht es bei Objekt-Arrays aus, denn Objekte haben von sich aus keine natürliche Ordnung. Es gibt für den Sortieralgorithmus keine Möglichkeit, ohne Hilfe zu entscheiden, ob ein Objekt kleiner oder größer ist als ein anderes. Zwei Wege stehen zur Verfügung, um ihm diese unmögliche Entscheidung abzunehmen.

Comparator

Die erste Möglichkeit ist, der `sort`-Methode einen `Comparator` zu übergeben. Eine Implementierung des Interface `Comparator` muss eine `compare`-Methode enthalten, die zwei Objekte entgegennimmt und einen `int`-Wert zurückgibt, der negativ ist, wenn das erste Objekt kleiner als das zweite ist, der 0 ist, wenn beide Objekte gleich sind, und positiv ist, wenn das erste Objekt größer ist als das zweite. Die `sort`-Methode vergleicht jeweils zwei Objekte mit dem übergebenen `Comparator` und bringt so das Array nach und nach in die richtige Reihenfolge.

```
Song[] songs = …;
Arrays.sort(songs, new Comparator(){
    public void compare(Object o1, Object o2){
        Song s1 = (Song) o1;
        Song s2 = (Song) o2;
        return s1.getInterpret().compareTo(s2.getInterpret());
    }
});
```

Listing 10.9 Songs nach Künstler sortieren

Im Beispiel wird ein Array von Songs nach Namen des Künstlers sortiert. Die `compareTo`-Methode der Klasse `String` hat Rückgabewerte mit derselben Bedeutung wie `Comparator.compare`, deshalb kann der Rückgabewert nach außen weitergereicht werden.

Es wurden inzwischen verschiedene Möglichkeiten entwickelt, um den gezeigten Code kürzer und lesbarer zu gestalten. So können Sie zum Beispiel den `Comparator` typisieren und so die beiden Casts einsparen (siehe Abschnitt 10.5.1, »Generics außerhalb von Collections«). Sie können auf Funktionsreferenzen zurückgreifen (mehr dazu im nächsten Kapitel) und damit ganz leicht ausdrücken, dass die Sortierung anhand des Interpreten erfolgen soll.

```
Arrays.sort(songs, Comparator.comparing(Song::getInterpret));
```

Listing 10.10 Vorschau auf das nächste Kapitel – sortieren mit Methodenreferenz

Wichtig ist bei der Sortierung mit einem `Comparator` immer eines: Wenn Sie anschließend das Array mit `binarySearch` durchsuchen wollen, dann müssen Sie denselben `Comparator` übergeben, denn nur so funktioniert die Suche.

```
Song[] songs = …;
Comparator c = Comparator.comparing(Song::getInterpret);
Arrays.sort(songs, c);
Arrays.binarySearch(songs, suchSong, c);
```

Listing 10.11 Nach einem Song suchen

`binarySearch` nimmt als Parameter das zu durchsuchende Array, den gesuchten Wert und – nur bei Objekt-Arrays – auch noch einen `Comparator`. Eine 0 oder eine positive Zahl als Rückgabewert ist der Index, an dem der gesuchte Wert gefunden wurde. Ein negativer Rückgabewert bedeutet, dass das gesuchte Element nicht im Array enthalten ist, und sagt gleichzeitig aus, wo es eingefügt werden müsste, um die Sortierung zu erhalten. In diesem Fall ist der Rückgabewert der Index des Einfügepunktes –1. Ein Rückgabewert von –1 bedeutet also, dass das Objekt am Anfang des Arrays eingefügt werden müsste, –2 bedeutet zwischen dem ersten und zweiten Element usw.

Comparable

Eine Alternative zum Sortieren mit `Comparator` besteht darin, auch bei Objekten eine *natürliche Ordnung* zu etablieren. Das bietet sich nicht bei allen Klassen an. Was sollte zum Beispiel die natürliche Ordnung von Songs sein? Alphabetisch nach Interpret? Nach Titel? Vielleicht nach der Länge? Nach Verkaufszahlen der Single? Songs sind keine guten Kandidaten für eine natürliche Ordnung. Anders ist das zum Beispiel für eine Klasse `Person`. Für Personen ist es im Kontext der meisten Programme natürlich, sie alphabetisch nach Namen zu sortieren. Um ihren Objekten eine natürliche Ordnung zu geben, muss eine Klasse das Interface `Comparable` und seine Methode `compareTo` implementieren:


```
public class Person implements Comparable {
    private String vorname, nachname;
    …
    public int compareTo(Object o){
        Person other = (Person) o;
        int vergleichNachname = nachname.compareTo(other.nachname);
        if (vergleichNachname == 0){
            return vorname.compareTo(other.vorname);
        } else {
            return vergleichNachname;
        }
    }
}
```

Listing 10.12 Die natürliche Ordnung von Personen – alphabetisch nach Namen

Im Beispiel ist die natürliche Ordnung von Personen zunächst eine Sortierung nach dem Nachnamen, und falls dieser identisch ist, nach dem Vornamen. Die Bedeutung des Rückgabewerts entspricht dabei der von `Comparator`: Eine negative Zahl bedeutet, dass `this` kleiner ist, eine positive, dass `this` größer ist, eine 0, dass beide gleich sind. Ein Array von solchen Objekten können Sie nun vergleichen, ohne einen `Comparator` zu verwenden:

```
Person[] leute = …;
Arrays.sort(leute);
```

Listing 10.13 Comparables sortieren

Auch wenn eine Klasse `Comparable` implementiert, können Sie Arrays trotzdem mit einem `Comparator` sortieren, wenn Sie eben nicht nach der natürlichen Ordnung sortieren möchten. Mit einem `Comparator` könnten Sie ein Array von Personen nach Alter sortieren, die `compareTo`-Methode würde in diesem Fall ignoriert.

Sortieren in mehreren Threads

Mit den Utility-Methoden aus `Arrays` haben Sie auch Ihre erste, oberflächliche Berührung mit der Multithreading-Programmierung. Die Methode `Arrays.parallelSort` erfüllt dieselbe Aufgabe wie `Arrays.sort`, kann die Sortierung aber in mehreren Threads gleichzeitig verarbeiten. Auf einem Computer mit nur einem Prozessorkern macht das keinen Unterschied, aber diese Computer werden immer seltener. Mit mehreren Prozessorkernen und für ein großes Array ist `parallelSort` signifikant schneller.

Füllen und kopieren

Dadurch, dass Arrays zusammenhängende Speicherbereiche belegen, können sie sehr effizient kopiert und gefüllt werden, sehr viel effizienter, als Sie es im Java-Code mit einer Schleife Element für Element umsetzen könnten. Aus diesem Grund gibt es in der `Arrays`-Klasse Methoden, die diese Aufgaben durch native Speicherzugriffe erfüllen.

Kopien können Sie mit den verschiedenen `copyOf`-Methoden erzeugen. Mit `copyOfRange` können Sie auch nur einen Teil eines Arrays in ein neues Array kopieren. Um ein Array ganz oder teilweise zu füllen, benutzen Sie die `fill`-Methoden.

```
int[] zahlen = …;
int[] kopie = Arrays.copyOf(zahlen, zahlen.length);
/*Parameter zwei und drei von fill geben den Bereich an, der vierte Parameter
den Füllwert.*/
Arrays.fill(zahlen, 0, 100, 0);
```

Listing 10.14 Arrays kopieren und befüllen

10.1.6 Übung: Sequenziell und parallel sortieren

Der Kasten »Sortieren in mehreren Threads« behauptet, dass parallele Sortierung auf Maschinen mit mehreren Kernen wesentlich schneller ist als sequenzielle Sortierung. Zeit, das auf die Probe zu stellen.

Schreiben Sie eine Klasse `Quader`, die drei `int`-Felder enthält: Länge, Breite und Höhe des Quaders. Außerdem soll die Klasse eine Methode `getVolumen` haben, die das Volumen des Quaders berechnet. Quader sollen durch ihr Volumen eine natürliche Ordnung haben.

Schreiben Sie dann ein Programm, das nacheinander zufällige `Quader`-Arrays der Größe 10, 100, 1.000 … 10.000.000 erzeugt. Um zufällige Quadergrößen zu erzeugen, können Sie die Klasse `java.util.Random` verwenden; limitieren Sie die Quadergröße dabei in jede Richtung auf 100.

Jedes dieser Arrays soll einmal sequenziell und einmal parallel sortiert werden. Messen Sie jeweils von beiden Sortierungen die Zeit, und geben Sie sie aus.

Vergleichen Sie, ab welcher Array-Größe parallele Sortierung Vorteile bringt. Die Lösung zu dieser Übung finden Sie im Anhang.

10.2 Die for-each-Schleife

Speziell für Arrays und Collections gibt es seit Java 5 eine weitere Variante der `for`-Schleife. Bei dieser Variante, genannt `for-each`-Schleife, entfällt die Zählvariable, und Sie erhalten nacheinander die Einträge des Arrays oder der Collection.

```
Song[] songs = …;
for (Song song : songs){
    System.out.println(song.getTitel());
}
```

Listing 10.15 Über ein Array mit der »for«-»each«-Schleife iterieren

In den Klammern der `for-each`-Schleife geben Sie keine Zählvariable an, sondern eine Variable vom Typ Ihres Arrays und, davon mit einem Doppelpunkt getrennt, das Array, aus dem die Werte genommen werden.

Neben Arrays können Sie auch alle Implementierungen des Interface `Iterable` in einer `for-each`-Schleife verwenden. Dazu gehören vor allem die verschiedenen Collections, die Sie im Verlauf dieses Kapitels kennenlernen werden.

Diese Variante der Schleife hat gegenüber der klassischen `for`-Schleife den klaren Vorteil, dass sie auf den ersten Blick sichtbar macht, was ihre Aufgabe ist: durch alle Werte des Arrays zu iterieren. Außerdem vermeiden Sie mit `for-each` potenzielle Fehler der `for`-Schleife, zum Beispiel den häufigen Fehler, dass in der Abbruchbedingung `i <= array.length` angegeben wird, was zu einer `IndexOutOfBoundsException` führt.

Andererseits ist die `for-each`-Schleife in ihren Möglichkeiten eingeschränkter als die `for`-Schleife. So können Sie zwar die Objekte im Array verändern, aber Sie können keine neuen Objekte ins Array schreiben oder vorhandene Objekte durch andere ersetzen. Auch können Sie ohne Zählvariable nichts tun, was einen Bezug zur Position im Array hat.

Die `for-each`-Schleife ist also sehr spezialisiert, erleichtert aber in dem Spezialfall, für den sie entwickelt wurde, die Programmierung erheblich.

10.3 Variable Parameterlisten

Ein Feature von Java, das ganz speziell mit Arrays funktioniert, sind die *variablen Parameterlisten*, meist kurz *Varargs* genannt. Sie erlauben es einer Methode, eine variable Anzahl von Parametern entgegenzunehmen. Ausgedrückt wird das durch drei Punkte nach dem Parametertyp.

```
public static int max(int… zahlen){
    int ergebnis = Integer.MIN_VALUE;
    for (int zahl : zahlen){
        if (zahl > ergebnis){
            ergebnis = zahl;
        }
    }
    return ergebnis;
}
```

Listing 10.16 Das Maximum beliebig vieler Zahlen finden

Diese Methode können Sie mit beliebig vielen `int`-Werten aufrufen, um deren Maximum zu finden. Innerhalb der Methode sind alle Parameter aus der Vararg-Liste als Array zu sehen.

Dieselbe Methode hätte sich auch ohne Varargs, also vor Java 5, umsetzen lassen, indem Sie ein Array als Parameter übergeben. Aber dann sähe der Aufruf zum Beispiel so aus:

```
int maximum = max(new int[]{zahl1, zahl2, zahl3, zahl4});
```

Es funktioniert, aber es ist umständlich zu schreiben und unangenehm zu lesen. Mit der variablen Parameterliste sieht es hingegen so aus:

```
int maximum = max(zahl1, zahl2, zahl3, zahl4);
```

An der Funktion hat sich nichts geändert, aber der Aufruf ist kürzer und leserlicher geworden.

Eine notwendige Einschränkung ist, dass eine Methode nur einen Vararg-Parameter haben kann und dass dieser der letzte Parameter in der Signatur sein muss. Aber auch mit dieser Einschränkung sind Varargs eine nützliche syntaktische Erweiterung.

Vorsicht: Varargs und überladene Methoden

Vorsicht ist geboten, wenn Sie Methoden mit variablen Parameterlisten überladen. Dann kann es nämlich vorkommen, dass ein Methodenaufruf nicht mehr eindeutig ist. Nehmen Sie als Beispiel folgende Methoden:

```
public int max(int zahl1, int zahl2){…}
public int max(int… zahlen){…}
```

Welche Methode ist mit dem Aufruf `max(zahl1, zahl2)` nun gemeint? Beide Signaturen passen zum Aufruf. In diesem Fall wird immer die Methode ohne Varargs bevorzugt, zwei »echte« Parameter passen besser zum Aufruf als ein Vararg-Parameter.

10.4 Collections

Collections sind neben Arrays die andere Art, in Java mit Wertlisten umzugehen. Im Gegensatz zu Arrays sind sie nicht durch native Methoden und einen zusammenhängenden Speicherbereich optimiert, sie haben auch keine eigene Syntax, um auf ihren Inhalt zuzugreifen. Collections sind ganz alltägliche Java-Objekte.

Ihre Vorteile gegenüber Arrays liegen im Benutzerkomfort. Collections haben eine umfangreichere und komfortablere API, und die verschiedenen Implementierungen des Interface `Collection` bieten viele verschiedene Möglichkeiten, für Ihren Anwendungsfall die richtige Strategie zu wählen. Außerdem, und dieser Vorteil ist nicht zu unterschätzen, müssen Sie für Collections keine Größe vorgeben, sie wachsen automatisch bei Bedarf.

Wo Arrays also systemnah und auf Performanz getrimmt sind, sind Collections in ihrer Verwendung komfortabler und besser geeignet für den Einsatz in Anwendungen, in denen es nicht auf jede Nanosekunde ankommt.

Alle Methoden von Collections arbeiten zunächst mit dem Typ `Object`. Sie können also beliebige Objekte in eine solche Collection hineinstecken und wissen beim Herauslesen nicht mit Sicherheit, welchen Typ Sie erwarten sollten. Sie können aber mit einer speziellen Notation angeben, dass diese Collection (oder auch viele andere Klassen) nur mit Objekten eines bestimmten Typs umgeht.

```
Collection<Song> playlist = new ArrayList<>();
```

Listing 10.17 Typsichere Collections

Genaueres zu dieser Notation in spitzen Klammern folgt weiter unten, aber ganz allgemein bedeutet `Collection<Song>`, dass es sich um eine `Collection` handelt, die Objekte vom Typ `Song` enthält.

Bei `java.util.Collection` handelt es sich lediglich um ein allgemeines Interface, das Methoden bereitstellt, die für die verschiedenen Spezialisierungen gleich sind. Diese Methoden ermöglichen den grundlegenden Zugriff auf alle Arten von Collections. Dazu gehören `add` und `addAll`, die der Collection ein Objekt bzw. alle Objekte einer anderen Collection hinzufügen, `contains` und `containsAll`, die prüfen, ob ein oder mehrere Objekte in der Collection enthalten sind, und `remove` und `removeAll`, die Objekte aus der Collection entfernen. Alle diese Methoden erwarten Parameter des Typs, der für die Collection in spitzen Klammern angegeben wurde. Haben Sie keinen Typ angegeben, dann wird überall `Object` erwartet, es können also beliebige Objekte übergeben werden.

Es gibt in `Collection` keine Methoden für den Zugriff auf ein bestimmtes Element, diese werden erst von manchen Spezialisierungen hinzugefügt. Der allgemeine Ansatz, an den Inhalt einer Collection heranzukommen, ist der, sie in einer `for-each`-

Schleife zu durchlaufen oder mit der Methode `iterator` ein `Iterator`-Objekt zu erzeugen (siehe Abschnitt 10.4.2, »Iteratoren«).

Falls Sie doch ein Array benötigen, können Sie mit den `toArray`-Methoden jede Collection in ein Array umwandeln. Dabei gibt es aber Feinheiten zu beachten. Die parameterlose `toArray`-Methode liefert immer ein `Object[]`, auch wenn Sie für Ihre Liste einen Typ angegeben haben. Das reicht in vielen Fällen aber nicht aus, denn eins können Sie mit Arrays nicht tun: sie casten. Auch wenn Ihr `Object[]` nur `Song`-Objekte enthält, wenn es als `Object[]` erzeugt wurde, können Sie es nicht nach `Song[]` casten. Wenn Sie aber aus einer Collection von Songs ein `Song[]` machen wollen, dann geht das sehr wohl, und zwar mit der `toArray`-Methode mit Parameter. Sie gibt immer ein Array des Typs zurück, den auch das als Parameter übergebene Array hat. Gedulden Sie sich noch ein wenig, diese Magie werde ich in Abschnitt 10.5, »Typisierte Collections – Generics«, erklären.

```
Collection<Double> liste = new ArrayList<>();
Double[] alsDoubles = liste.toArray(new Double[liste.size()]);
Number[] alsNumber = liste.toArray(new Number[liste.size()]);
Object[] alsObject = liste.toArray(new Object[liste.size()]);
Object[] auchAlsObject = liste.toArray();
```

Listing 10.18 Collections in typisierte Arrays umwandeln

Wie das übergebene Array verwendet wird, ist etwas eigen. Wenn es groß genug ist, um alle Elemente der Collection aufzunehmen, dann wird das übergebene Array gefüllt und zurückgegeben. Ist das Array aber zu klein, so wie im Beispiel, dann ist der Rückgabewert ein neues Array des gleichen Typs und mit der Größe der Collection. Natürlich können Sie auch mit dieser Methode den Inhalt der Liste nicht in einen völlig anderen Typ umwandeln. Im obigen Beispiel würde `liste.toArray(new Integer[0])` mit einer Exception scheitern.

10.4.1 Listen und Sets

Die zwei mit Abstand am häufigsten verwendeten Arten von Collections sind Lists und Sets. Es gibt daneben zwar noch weitere Spezialisierungen, sie kommen aber seltener zum Einsatz.

Listen

Listen sind die Collections, die in ihrer Funktionsweise am ehesten einem Array entsprechen. Elemente bleiben immer in der Reihenfolge, in der sie hinzugefügt werden, und Sie können ein bestimmtes Element direkt lesen (mit `get(index)`) und schreiben (mit `set(index, element)`).

Damit bieten Listen alles, was Sie auch mit einem Array erreichen können, aber mit mehr Komfort. Sie wachsen automatisch, um neue Elemente aufnehmen zu können, haben eine reichere API und ersparen Ihnen einigen Verwaltungsaufwand. So wird zum Beispiel, wenn Sie ein Element mit `remove` entfernen, die entstehende Lücke geschlossen, indem alle folgenden Elemente nachrutschen.

```
List<Song> playlist = new ArrayList<>();
//Der Playlist zwei Songs hinzufügen
playlist.add(new Song(…));
playlist.add(new Song(…));
//Den zweiten Song durch einen anderen ersetzen
playlist.set(1, new Song(…));
//Alle Songs abspielen
for (Song song : playlist){
    System.out.println(song.getTitel());
    song.play();
}
```

Listing 10.19 Arbeiten mit einer Liste

Wie Sie an den Beispielen sehen, ist auch `List` nur ein Interface, und Sie müssen die Implementierung auswählen, die Sie benutzen möchten. Bei Listen ist diese Auswahl allerdings nicht weiter schwierig, denn `ArrayList` ist fast immer die beste oder zumindest eine gute Wahl.

`ArrayList` benutzt intern ein Array, um Daten zu speichern, versteckt aber alle Schwierigkeiten vor Ihnen. Das bedeutet vor allem, dass eine `ArrayList` ihr internes Array automatisch durch ein größeres ersetzt, wenn es voll ist.

Trotz des damit verbundenen Erzeugens eines größeren Arrays und Umkopierens der enthaltenen Daten ist `ArrayList` für die meisten Fälle die performanteste `List`-Implementierung. Nur in wenigen Fällen, vor allem wenn Sie häufig Elemente aus der Mitte der Liste entfernen müssen, bietet die doppelt verkettete Liste `LinkedList` eine bessere Performance.

Egal, welche Implementierung von Liste Sie verwenden, denken Sie daran, dass es von gutem Stil zeugt, als Typ Ihrer Variablen das Interface `List` zu deklarieren, nicht die Klasse `ArrayList` oder `LinkedList`.

Sets

Sets sind eine andere Spezialisierung von Collection, die in etwa einer mathematischen Menge entspricht – wie auch der Name schon andeutet: »set« ist das englische Wort für Menge.

Das bedeutet, dass ein Set niemals mehrere identische Elemente enthalten kann und keine interne Reihenfolge hat. Dementsprechend bietet `Set` auch keine Methoden für den direkten Zugriff auf ein Element, wie `List` es tut. Um auf die Elemente eines `Set` zuzugreifen, **müssen** Sie in einer Schleife darüber iterieren.

```
Set<String> besuchteOrte = new HashSet<>();
besuchteOrte.add("Berlin");
besuchteOrte.add("Bonn");
besuchteOrte.add("Berlin");
besuchteOrte.add("Hamburg");
for (String ort : besuchteOrte){
    System.out.println(ort);
}
```

Listing 10.20 Arbeiten mit einem Set

Das Beispiel gibt Berlin, Bonn und Hamburg aus. Auch wenn Berlin zweimal hinzugefügt wurde, zählt es nur einmal. Dieses Verhalten ist auch der Grund, warum die `add`-Methoden von `Collection` einen `boolean`-Wert zurückgeben: Wenn das Element hinzugefügt wurde, ist der Rückgabewert `true`, wenn es schon vorhanden war, `false`. Sie können also beim zweiten Aufruf von `besuchteOrte.add("Berlin")` feststellen, ob wirklich ein Element hinzugefügt wurde oder nicht. Beachten Sie außerdem, dass die Ausgabe nicht in einer festgelegten Reihenfolge erfolgt und es auch keine Garantie dafür gibt, dass die Reihenfolge gleich bleibt.

Die meistbenutzte Implementierung von `Set` ist `HashSet`, das die Einmaligkeit der enthaltenen Elemente durch deren Hashcode sicherstellt. Wenn Sie ein Set verwenden wollen, ist es deshalb wichtig, dass Ihre Klassen `hashCode` und `equals` überschreiben. Betrachten Sie dazu das folgende Beispiel:

```
Set<Song> meineMusik = new HashSet<>();
meineMusik.add(new Song("The One and Only", "Chesney Hawkes", 220));
meineMusik.add(new Song("The One and Only", "Chesney Hawkes", 220));
```

Listing 10.21 Der feine Unterschied mit oder ohne »hashCode«

Wenn die Klasse `Song` die `hashCode`-Methode nicht überschreibt, dann ist »The One and Only« am Ende des Beispiels zweimal im Set enthalten, denn die von `Object` geerbte `hashCode`-Implementierung gibt für verschiedene Objekte verschiedene Werte zurück, auch wenn die Objekte inhaltsgleich sind. Wenn Sie aber in `Song` Ihre eigene `hashCode`-Methode schreiben, die den Code aus Titel, Interpret und Länge berechnet, dann wird der zweite Song nicht hinzugefügt, weil er jetzt als identisch erkannt werden kann.

`Set` hat ein weiteres Subinterface `SortedSet`, das ebenfalls nur eine nennenswerte Implementierung hat: `TreeSet`. `SortedSet` unterscheidet sich in seiner Verwendung nicht von einem herkömmlichen `Set`, der einzige Unterschied ist, dass es die enthaltenen Elemente sortiert und garantiert, dass sie beim Iterieren in der richtigen Reihenfolge ausgegeben werden. Die Reihenfolge kann entweder die natürliche Reihenfolge sein, wenn die Elemente `Comparable` implementieren, oder sie kann durch einen `Comparator` festgelegt werden, den Sie dem `TreeSet` im Konstruktor übergeben.

10.4.2 Iteratoren

Das Interface `Collection` bietet eine weitere Methode, die dazu dient, über alle Werte der Collection zu iterieren: die Methode `iterator`, die ein Objekt vom Typ `Iterator` zurückgibt.

Iteratoren sind der objektorientierte Weg, die Werte einer Collection zu durchlaufen. Ihre Methode `hasNext` prüft, ob ein weiteres Element vorhanden ist, `next` gibt das nächste Element zurück. Mit einer Schleife können Sie so alle Elemente betrachten.

```
List<Song> songs = …;
Iterator<Song> it = songs.iterator();
while (it.hasNext){
    Song song = it.next();
    …
}
```

Listing 10.22 Iterieren mit »Iterator«

Seit mit Java 5 die `for-each`-Schleife eingeführt wurde, ist es nur noch selten nötig, selbst mit einem `Iterator` zu arbeiten. Das neue Schleifenkonstrukt ist kürzer und weniger anfällig für Programmierfehler.

Iteratoren haben aber dennoch einen Vorteil, der in manchen Situationen zum Tragen kommt: Sie sind keine Sprachkonstrukte, sondern Objekte, und als solche können sie in Feldern gespeichert, als Parameter an Methoden übergeben und aus Methoden zurückgegeben werden. Konkret bedeutet das für Sie, dass Sie nicht in einem Durchlauf über alle Elemente iterieren müssen, wie mit einer Schleife, sondern einzelne Elemente auslesen können, um dann zu einem späteren Zeitpunkt an derselben Stelle fortzusetzen.

```
public boolean spieleSchlafmodus(Iterator<Song> songs, int zeit) {
    int gesamtzeit = 0;
    while (gesamtzeit < zeit) {
        if (songs.hasNext()) {
            Song song = songs.next();
```

```
            gesamtzeit += song.getLaenge();
            spieleSong(song);
        } else {
            return false;
        }
    }
    return songs.hasNext();
}
```

Listing 10.23 Der Schlafmodus für den Musikspieler

In diesem Beispiel hätten Sie es ohne ein `Iterator`-Objekt wesentlich schwerer. Die Methode `spieleSchlafmodus` soll so lange Musik spielen, bis eine übergebene Zeit erreicht ist, und dann den aktuellen Song zu Ende spielen, denn nichts ist schlimmer, als wenn die Musik mitten im Lied abbricht. Wenn Sie den Schlafmodus das nächste Mal aktivieren, dann übergeben Sie denselben `Iterator`, und es geht mit dem nächsten Song weiter.

Ein weiterer, gelegentlicher Vorteil von Iteratoren ist, dass Sie beim Iterieren Elemente aus der `Collection` entfernen können, indem Sie am Iterator `remove` rufen. Aber Vorsicht, nicht alle Iteratoren unterstützen `remove`, diese Methode wirft manchmal eine `UnsupportedOperationException`.

10.4.3 Übung: Musiksammlung und Playlist

Schreiben Sie zwei Klassen, `Musiksammlung` und `Playlist`. Beide Klassen sollen Methoden anbieten, die Songs hinzufügen und Songs entfernen, und sie sollen mit einer `for-each`-Schleife iterierbar sein. Der Unterschied zwischen beiden Klassen ist der, dass `Playlist` denselben Song mehrmals enthalten kann, `Musiksammlung` aber nicht.

Versuchen Sie bei der Programmierung, doppelten Code zu vermeiden. Alles, was beide Klassen gemeinsam haben, sollten Sie auch nur einmal schreiben. Die Lösung zu dieser Übung finden Sie im Anhang.

10.5 Typisierte Collections – Generics

In den Codebeispielen zu Collections haben Sie schon gesehen, wie Sie für eine Collection einen Typ angeben können.

```
List<Number> zahlen = new ArrayList<Number>();
```

oder

```
List<Number> zahlen = new ArrayList<>();
```


In spitzen Klammern geben Sie am Variablentyp an, welchen Typ die Collection enthalten soll. Beim Konstruktoraufruf können Sie den Typ wiederholen, seit Java 7 ist das aber nicht mehr notwendig, und Sie können stattdessen ein leeres Paar spitze Klammern angeben, den sogenannten *Diamantoperator*. Der Compiler erkennt dann aus dem Kontext, welcher Typ dort stehen sollte. Diese Typangabe kann auch bei vielen anderen Klassen genutzt werden (siehe Abschnitt 10.5.1, »Generics außerhalb von Collections«), aber Collections bieten einen sehr anschaulichen Einstieg.

Die Typangabe bewirkt nicht nur, dass der Collection keine inkompatiblen Typen hinzugefügt werden können, sondern sie ändert den Typ der Rückgabewerte und Parameter. So kann schon der Compiler sicherstellen, dass Sie gar nicht mit falschen Typen arbeiten können, und Sie bekommen aus einer Collection immer den richtigen Typ heraus und müssen nicht von `Object` auf den richtigen Typ casten.

Wie funktioniert die Typisierung?

Dass Sie nicht casten müssen, wenn Sie typisierte Collections verwenden, bedeutet nicht, dass es zu keinen Casts kommt. Die Typangaben werden vom Compiler umgesetzt, indem er Casts für Rückgabewerte einfügt und prüft, ob alle als Parameter übergebenen Werte wirklich vom richtigen Typ sind.

Zur Laufzeit ist eine `Collection<Song>` also nicht von einer untypisierten `Collection` zu unterscheiden.

Typisierte Collections sind keine besondere Abart von Collections und auch keine Spezialisierungen. Sie sind zur Laufzeit nicht von untypisierten Collections zu unterscheiden. Deswegen heißt dieses Sprachfeature auch *Generics*: Generische Collections können als typisierte Container verwendet werden, ohne dass Sie für jeden Inhaltstyp eine Spezialisierung benötigen. Oft wird auch der Begriff *typsicher* verwendet, denn genau das erreichen Sie mit der Typangabe: die Sicherheit, dass nur dieser Typ als Inhalt einer Collection verwendet werden kann.

Zur Compilezeit verhalten *Generics* sich aber ähnlich wie Spezialisierungen. Sie können eine typisierte Collection einer untypisierten Collection-Variablen zuweisen und an Methoden mit einem untypisierten Collection-Parameter übergeben.

```
List<Song> typisiert = new ArrayList<>();
List untypisiert = typisiert;
untypisierteMethode(typisiert);
…
public void untypisierteMethode(List untypisiert){…}
```

Listing 10.24 Typisierte Collections können immer als untypisierte verwendet werden.

Die Typisierung lässt sich überlisten

Dass *typisierte* Listen *untypisierten* Variablen zugewiesen werden können, führt leider dazu, dass man die Typeinschränkung umgehen kann. Betrachten Sie folgendes Beispiel:

```
List<Song> typisiert = new ArrayList<>();
List untypisiert = typisiert;
untypisiert.add("Ich bin ein String");
Song song = typisiert.get(0);
```

Der Code kompiliert fehlerfrei, es werden nur erlaubte Operationen ausgeführt. Es ist völlig legal, `untypisiert` einen String hinzuzufügen, denn an dieser Variablen sind keine Typinformationen vorhanden. Erinnern Sie sich, durch die Typangabe wird keine spezielle Liste erzeugt, Sie bringen nur den Compiler dazu, Casts und Prüfungen einzufügen. Da `untypisiert` keine Typangabe macht, prüft der Compiler auch nicht. Aber `untypisiert` und `typisiert` referenzieren dasselbe `List`-Objekt.

Damit ist der Zugriff in der letzten Zeile für den Compiler zwar korrekt, `typisiert` hat schließlich den Typ `Song`, aber zur Laufzeit kommt es zu einer `ClassCastException`, weil das erste Element eben kein `Song`, sondern ein `String` ist.

Diese Art Fehler ist ärgerlich, aber weitgehend vermeidbar, wenn Sie einfachen Grundregen folgen:

1. Weisen Sie niemals eine typisierte Collection einer untypisierten Variablen zu, auch nicht als Methodenparameter.
2. Wenn Sie es doch tun müssen, zum Beispiel, weil Sie eine Methode schreiben, die auf Collections mit beliebigen Typen operiert, dann vermeiden Sie es, neue Elemente hinzuzufügen.

10.5.1 Generics außerhalb von Collections

Collections sind aber nicht die einzigen Klassen, die von Generics profitieren. Sie sind lediglich ein prominentes Beispiel, denn bei ihnen ist es sehr anschaulich, dass sie den angegebenen Typ enthalten.

Auch bei Klassen, die keine Container für andere Objekte sind, kann die Typangabe einen großen Vorteil bringen. Fast immer, wenn eine Klasse oder eine Methode mit dem Typ `Object` arbeitet, kann sie durch Generics verbessert werden, denn durch die Typangabe wird der Code lesbarer, und Typfehler können bereits vom Compiler erkannt werden statt erst zur Laufzeit.

Nehmen Sie zum Beispiel das Ihnen schon bekannte Interface `Comparator`: Sie können auch für einen `Comparator` einen Typ angeben und so klarmachen, welche Typen er miteinander vergleicht:


```
Comparator<Song> songsNachLaenge = new Comparator<Song>(){
    public int compare(Song o1, Song o2) {
        return o1.getLaengeInSekunden() -
               o2.getLaengeInSekunden();
    }
};
```

Listing 10.25 Ein »Comparator« nur für Songs

Zwei Vorteile, einen `Comparator` speziell für Songs zu haben, werden aus dem obigen Beispiel sofort ersichtlich: Zum einen ist am Variablentyp `Comparator<Song>` erkennbar, welche Objekte mit diesem `Comparator` verglichen werden können, zum anderen sparen Sie in der `compare`-Methode die Casts, da der angegebene Typ in die Methodensignatur übernommen wird.

Ein dritter Vorteil zeigt sich, wenn Sie den `Comparator` mit Utility-Methoden wie der Sortiermethode `Collections.sort` verwenden. Der Compiler kann nun prüfen, ob `Collection` und `Comparator` überhaupt kompatibel sind.

```
List<Song> songs = …;
List<String> strings = …;
Collections.sort(songs, songsNachLaenge); //funktioniert
Collections.sort(strings, songsNachLaenge); //Compilerfehler!!!
```

Listing 10.26 Typsicherheit durch Generics

Ohne eine Typangabe würde der zweite Aufruf von `Collections.sort` fehlerlos kompilieren, erst zur Laufzeit käme es zu einer `ClassCastException`. Die Typangaben helfen Ihnen in diesem Fall, Fehler schon früh in Ihrem Entwicklungszyklus zu erkennen, und sichern Sie dagegen ab, später neue Fehler einzubauen. Deswegen, und weil Ihr Code kürzer und lesbarer wird, sollten Sie überall, wo es möglich ist, auch Typangaben verwenden.

Viele weitere Klassen und Methoden im JDK lassen sich mit Typparametern verwenden. Dazu gehören zum Beispiel auch die Utility-Methoden der Klassen `Arrays` und `Collections`. Sie erkennen Klassen und Methoden, die Typparameter akzeptieren, im Javadoc immer daran, dass sie einen Typ in spitzen Klammern als Parameter oder Rückgabewert deklarieren. Häufig haben diese Typparameter einbuchstabige Namen wie `<T>` für Typ oder `<E>` für Element. Häufig kommt dafür auch die erweiterte Syntax für Typparameter zum Einsatz, die Sie im nächsten Abschnitt kennenlernen werden. So sieht zum Beispiel die Signatur von `Collections.sort` aus, die sicherstellt, dass Sie eine Liste nur mit einem kompatiblen `Comparator` sortieren können:

```
public static <T> void sort(List<T> list, Comparator<? super T> c)
```

10.5.2 Eigenen Code generifizieren

Sie wissen nun, wie Sie mit »generifizierten« Klassen arbeiten können, auch über Collections hinaus. Aber auch Ihr eigener Code kann auf viele verschiedene Arten davon profitieren, wenn Sie typsichere Klassen und Methoden erzeugen. Sie können sowohl ganze Klassen als auch einzelne Methoden typsicher machen, wobei es hierbei keine großen syntaktischen Unterschiede gibt.

Generische Methoden und beschränkte Typen

Es ist nicht schwierig, eine Methode zu schreiben, die einen typisierten Parameter erwartet oder einen typisierten Wert zurückgibt. Sie können für Parameter oder Rückgabewerte, die einen Typparameter unterstützen, diesen in der Methodensignatur angeben.

```
public List<Integer> parseAlleInts(List<String> strings){
    List<Integer> ergebnis = new ArrayList<>();
    for (String s : strings){
        ergebnis.add(Integer.parseInt(s));
    }
    return ergebnis;
}
```

Listing 10.27 Eine Liste von Strings in Zahlen umwandeln

Die Beispielmethode erwartet eine Liste von Strings als Parameter und gibt eine Liste von Integern zurück. So weit funktioniert alles wie erwartet.

Etwas schwieriger wird die Situation leider, wenn Sie Vererbungsbeziehungen zwischen den Typen typisierter Klassen betrachten. Betrachten Sie dazu einmal mehr das Beispiel, mit dem die Vererbung angefangen hat: das Beispiel aus dem Tierreich.

Sie können einer Collection vom Typ `Tier` zwar einen `Hund` oder eine `Katze` hinzufügen, aber Sie können ein Objekt vom Typ `Collection<Hund>` nicht einer Variablen vom Typ `Collection<Tier>` zuweisen, denn für diese Zuweisung beachtet der Compiler die Vererbungsbeziehung zwischen den Typen nicht.

Unterschied zwischen typisierten Collections und Arrays

Die Zuweisung `Tier[] tiere = new Hund[5]` ist problemlos möglich, da der Compiler hier die Vererbung berücksichtigt. Leider ergibt sich daraus auch ein Fehlerrisiko, denn Code wie der folgende kompiliert fehlerfrei und verursacht erst zur Laufzeit eine `ArrayStoreException`, weil eine `Katze` nicht in ein `Hund[]` passt.

```
Tier[] tiere = new Hund[5];
tiere[0] = new Katze();
```


Ein Beispiel macht das Problem klarer:

```
public List<Tier> sortiereNachGewicht(List<Tier> tiere){
    Collections.sort(tiere, new Comparator<Tier>(){
        public int compare(Tier o1, Tier o2) {
            if (o1.getGewicht() > o2.getGewicht()){
                return 1;
            } else if (o1.getGewicht() < o2.getGewicht()){
                return -1;
            } else {
                return 0;
            }
        }
    });
    return tiere;
}
//Funktioniert
sortiereNachGewicht(new ArrayList<Tier>());
//Funktioniert nicht
sortiereNachGewicht(new ArrayList<Hund>());
```

Listing 10.28 Tiere nach Gewicht sortieren

Diese Methode sortiert eine Liste von Tieren nach ihrem Gewicht. Die Schwierigkeit steckt dabei im Aufruf: Sie können diese Methode mit einer Liste vom Typ `Tier` rufen, aber nicht mit einer Liste vom Typ `Hund`, denn obwohl `Hund` eine Spezialisierung von `Tier` ist, kann eine `List<Hund>` nicht dem Parametertyp `List<Tier>` zugewiesen werden. Zum Glück ist das kein unlösbares Problem, aber die Syntax wird an dieser Stelle etwas unschön. Sie können für eine Typangabe eine Beschränkung angeben, einen sogenannten *Upper Bound*, und so festlegen, dass alle Spezialisierungen einer Oberklasse erlaubt sind. Sie können die Signatur der Methode `sortiereNachGewicht` so anpassen:

```
public List<? extends Tier> sortiereNachGewicht(List<? extends Tier> tiere)
```

Listing 10.29 Typangaben mit Upper Bound

Die Schreibweise `<? extends Tier>` bedeutet, dass die Liste einen beliebigen Typ haben kann, es muss sich aber um eine Spezialisierung von `Tier` handeln. Innerhalb der Methode ändert sich nichts, die Liste `tiere` kann wie eine `List<Tier>` behandelt werden. Sie können nun aber auch Listen vom Typ `Hund` oder `Katze` übergeben. Eine Einschränkung: Sie können einer mit Einschränkung deklarierten Liste keine Elemente hinzufügen, da der Compiler nicht prüfen kann, ob sie den richtigen Typ für diese Liste haben.

Dummerweise zwingt Sie die Änderung des Parameters in diesem Fall auch dazu, den Rückgabewert zu ändern. Sie geben dasselbe `List`-Objekt zurück, das als Parameter übergeben wird, und dafür steht keine andere Typinformation zur Verfügung. Dadurch müssen Sie, wenn Sie den Rückgabewert einer Variablen zuweisen, den beschränkten Typ verwenden, weil der Compiler keine bessere Information hat.

```
List<Dackel> dackel = new ArrayList<Dackel>()
List<? extends Tier> sortierteDackel = sortiereNachGewicht(dackel);
```

Diese Lösung ist besser als die vorige, Sie können nun auch Listen von Hunden oder Katzen sortieren. Aber zufriedenstellend ist auch das noch nicht, denn die genaue Typinformation für den Rückgabewert geht verloren. Wir brauchen eine Möglichkeit, dem Compiler zu sagen, dass die zurückgegebene Liste denselben Typ hat wie die als Parameter übergebene.

Auch das ist möglich, es bedarf dazu nur einer weiteren Variante der Generics-Syntax. Sie können in der Methodensignatur einen benannten Typparameter deklarieren:

```
public <T extends Tier> List<T> sortiereNachGewicht(List<T> tiere){
    Collections.sort(tiere, new Comparator<T>(){
        public int compare(T o1, T o2) {
            if (o1.getGewicht() > o2.getGewicht()){
                return 1;
            } else if (o1.getGewicht() < o2.getGewicht()){
                return -1;
            } else {
                return 0;
            }
        }
    });
    return tiere;
}
```

Listing 10.30 Methode mit benanntem Typparameter

Zu lesen ist sie dies so: Der Typ `T` ist eine Spezialisierung der Klasse `Tier`. Die Methode erwartet eine Liste dieses Typs `T` als Parameter und gibt eine Liste des Typs `T` zurück. Der wichtige Unterschied zu vorher ist, dass `T` an den Stellen, an denen es vorkommt, **denselben Typ** meint. Wenn Sie eine `List<Dobermann>` als Parameter übergeben, dann legen Sie dadurch den Typparameter `T` fest, und der Rückgabewert wird automatisch auch `List<Dobermann>`. Rufen Sie die Methode an einer anderen Stelle mit einer `List<Dackel>` auf, dann bekommen Sie auch eine `List<Dackel>` zurück.

Wenn Sie sich den Methodenrumpf anschauen, stellen Sie fest, dass Sie den Typparameter `T` auch dort noch verwenden können. Sie können Variablen vom Typ `T` deklarieren und mit ihnen arbeiten, als sei `T` eine echte Klasse. Das Einzige, was nicht möglich ist, ist, neue Instanzen von `T` durch einen Konstruktor zu erzeugen. Da Konstruktoren nicht vererbt werden, hat der Compiler keine Informationen darüber, welche Konstruktoren die Klasse `T` haben könnte.

Mehr zu Typvariablen

Sie können für eine Methode auch mehrere Typvariablen deklarieren, indem Sie sie mit Kommata trennen:

```
public <T extends Tier, U extends Number> tuEtwas(List<T> tiere,
  List<U> zahlen)
```

Für diese Anwendung ist es schwierig, Beispiele zu finden, die nicht sehr speziell sind. Bessere Beispiele dafür finden Sie bei Klassen mit generischen Typen, wie zum Beispiel der Klasse `Map`, die Sie weiter unten kennenlernen werden.

Sie können auch mehrere Beschränkungen auf eine Typvariable legen und so sicherstellen, dass übergebene Werte nicht nur der richtigen Klasse angehören, sondern außerdem ein oder mehrere Interfaces implementieren.

```
public <T extends Tier & Fleischfresser> void jage(T jaeger, Tier beute)
```

Nur fleischfressende Tiere können jagen, weder fleischfressende Pflanzen noch pflanzenfressende Tiere haben diese Fähigkeit. Dass nur ein fleischfressendes Tier als Jäger in Frage kommt, wird durch einen generischen Typ sichergestellt, der gleichzeitig durch `Tier` und `Fleischfresser` beschränkt ist.

Als Letztes gibt es noch die Möglichkeit, eine Beschränkung in die andere Richtung anzugeben. In diesem Fall soll die Typvariable keine Spezialisierung der angegebenen Klasse sein, sondern eine Oberklasse der Klasse. Dies wird durch das Schlüsselwort `super` dargestellt. Die Typangabe `<T super Dackel>` würde demnach die Klassen `Dackel`, `Hund`, `Tier` und `Object` akzeptieren.

Sie kennen mit `Collections.sort` schon eine Methode, die einen *Lower Bound* verwendet.

```
public static <T> void sort(List<T> list, Comparator<? super T> c)
```

Listing 10.31 Methodensignatur mit Lower Bound

Die `sort`-Methode verlangt eine Liste vom Typ `T` (unbeschränkt) und einen `Comparator` vom Typ `T` oder einer Oberklasse von `T`. Das ist deshalb sinnvoll, weil zum Beispiel eine Liste von Dackeln auch von einem `Comparator` für Hunde oder Tiere sortiert werden könnte.

Die PECS-Regel

Eine hilfreiche Faustregel, wann für Collections ein Typparameter mit Upper Bound verwendet wird und wann einer mit Lower Bound, ist die PECS-Regel: **Producer extends, Consumer super**. Damit ist Folgendes gemeint:

- Wenn die typisierte Collection ein Producer ist, ihr also Objekte entnommen und verarbeitet werden, dann wird sie mit einem Upper Bound (`? extends Tier`) deklariert. So kann immer ein Element der Collection einer Variablen vom Typ der Schranke zugewiesen und damit gearbeitet werden.
- Ist die Collection dagegen ein Consumer, werden ihr also Objekte hinzugefügt, dann wird sie mit einem Lower Bound (`? super Tier`) deklariert, denn so können neue Objekte vom Typ der Schranke immer hinzugefügt werden.

Generische Klassen

Klassen sind nicht schwieriger zu generifizieren als Methoden, es ändert sich nur die Stelle, an der Sie die Typvariablen angeben. Bei einer Klasse folgen die Typparameter dem Klassennamen und müssen immer benannt sein, Fragezeichen sind nicht erlaubt. So deklarierte Typparameter können im gesamten Klassenrumpf verwendet werden als Methodenparameter, Rückgabewerte oder Feldtypen.

```
public class Tierarzt<T extends Tier>{
    private List<T> patienten = new ArrayList<>();
    public void neuerPatient(T neu){
        patienten.add(neu);
    }
    public List<T> getPatienten(){
        return patienten;
    }
}
```

Listing 10.32 Der Tierarzt, Alleskönner oder Spezialist?

Der `Tierarzt` aus dem Beispiel akzeptiert einen Typparameter, der von `Tier` erbt. Er kann so als `Tierarzt<Katze>` zum Spezialisten werden, der nur Katzen behandelt. Sie wissen dann, dass seine Patientenliste nur Katzen enthalten kann. Geben Sie keinen Typ an – Typparameter sind immer optional –, dann wird der angegebene Upper Bound als Typ verwendet, `T` entspricht also `Tier`.

Ein weiterer Punkt, den es bei Klassen zu beachten gilt, sind typisierte Interfaces. Genau wie bei Klassen können Sie auch bei Interfaces Typparameter angeben. Für einen solchen Typparameter können Sie, wenn Sie das Interface implementieren, einen konkreten Typ angeben.

Sie haben das schon bei der anonymen Implementierung von `Comparator` in den Beispielen oben gesehen: `new Comparator<Tier>(){…}` implementiert das Interface `Comparator` mit dem Typ `Tier`, wodurch die Methodenparameter der Methode `compare(T o1, To2)` auf Tier festgelegt werden. Ebenso können Sie auch einen Typ angeben, wenn Sie in einer nichtanonymen Klasse ein typisiertes Interface implementieren. `Comparable` ist ein häufiges Beispiel dafür:

```
public class Quader implements Comparable<Quader>{
    …
    public int compareTo(Quader other){
        return getVolumen() - other.getVolumen();
    }
}
```

Listing 10.33 Implementieren eines typisierten Interface

Die Klasse `Quader` aus Abschnitt 10.1.6, »Übung: Sequenziell und parallel sortieren«, war schon dort `Comparable`. Dadurch, dass die Klasse aber nun `Comparable<Quader>` implementiert, wird klar, dass ein Quader nur mit anderen Quadern verglichen werden kann. Welchen Sinn hätte es auch, einen Quader mit einem String zu vergleichen? Die von `Comparable` vorgeschriebene `compareTo`-Methode übernimmt den Typparameter und stellt so sicher, dass Quader nur mit Quadern verglichen werden können.

Warum ist »Class« generisch?

Es ist etwas verwunderlich, dass ausgerechnet die Klasse `Class` einen Typparameter hat. Warum hat die Klasse, die Informationen über Klassen enthält, einen Typparameter? Es gibt dafür mehrere Gründe, die alle mit der Reflection-API zusammenhängen. Zusammengefasst ist der häufigste Grund einfach, zu erzwingen, dass ein passendes Klassenobjekt als Parameter übergeben wird. So können Sie unter anderem das Problem umgehen, dass Sie keine neuen Instanzen der generischen Typen erzeugen können. Sie müssen nur wissen, welche Konstruktoren oder Factory-Methoden die Klasse hat, denn wenn Sie Reflection benutzen, unterstützt Sie der Compiler nicht mehr.

```
private static <T extends Number> List<T> addNumbersToList(List<T> zahlen,
Class<T> klasse, String... werte) throws Exception{
    Method m = klasse.getMethod("valueOf", String.class);
    for (String wert : werte){
        zahlen.add((T) m.invoke(null, wert));
    }
    return zahlen;
}
```

Diese Methode benutzt das übergebene Klassenobjekt, um eine Reihe von Strings in den Zahlentyp umzuwandeln, den die übergebene Liste hat, und fügt sie der Liste hinzu. Übergeben Sie eine `List<Integer>`, werden Integer erzeugt, für ein Objekt vom Typ `List<Double>` sind es Doubles usw. Der Typparameter an `Class` stellt sicher, dass Sie nur das Klassenobjekt übergeben können, das zum Listentyp passt. Die Methode funktioniert, weil alle im JDK enthaltenen Spezialisierungen von `Number` eine Methode namens `valueOf` enthalten.

10.5.3 Übung: Generisches Filtern

Schreiben Sie ein Interface `Filter` mit einer Methode `filter`. Diese Methode erhält ein einzelnes Objekt und gibt einen `boolean`-Wert zurück, der bestimmt, ob dieses Objekt in einer gefilterten Liste enthalten sein soll oder nicht.

Schreiben Sie im Interface außerdem eine statische Methode `wendeFilterAn`, die eine Collection und einen Filter als Parameter erwartet. Sie soll aus der Collection alle Elemente entfernen, für die der übergebene Filter `true` liefert. Stellen Sie durch eine Typvariable sicher, dass die übergebene Liste und der übergebene Filter zueinanderpassen.

Zuletzt erzeugen Sie zwei Implementierungen von `Filter`. Der `UngeradeZahlenFilter` soll alle ungeraden Zahlen aus einer Collection von Integer-Werten entfernen. Der `KurzeStringFilter` soll alle Strings entfernen, die kürzer als zehn Zeichen sind. Die Lösung zu dieser Übung finden Sie im Anhang.

10.6 Maps

Obwohl es nicht das `Collection`-Interface erweitert, gehört das folgende wichtige Interface in denselben Kontext: `Map`. Maps werden im Deutschen manchmal als *assoziative Arrays* bezeichnet, wenn man darauf besteht, Fachbegriffe einzudeutschen, und dieser Name ist nicht abwegig. Ein Array ordnet ein Objekt einer Zahl zu, dem Index. Eine `Map` ordnet ein Objekt, den Wert, einem anderen Objekt zu, dem Schlüssel. Sie erlaubt es, Zuordnungen von einem beliebigen Typ zu einem beliebigen anderen Typ vorzunehmen. Häufig werden Strings als Schlüssel verwendet, jedoch kann jeder Objekttyp als Schlüssel dienen.

Sie haben eine `Map` schon ganz am Anfang des Buches in Aktion gesehen, im Eingangsbeispiel WordCount:

```
private Map<String, Integer> wordCounts = new HashMap<>();
private void count(InputStream source){
```



```
    try(Scanner scan = new Scanner(source)){
        scan.useDelimiter("[^\\p{IsAlphabetic}]+");
        while (scan.hasNext()){
            String word = scan.next().toLowerCase();
            totalCount++;
            wordCounts.put(word, wordCounts.getOrDefault(word, 0) + 1);
        }
    }
}
```

Listing 10.34 Noch einmal der Wortzähler

Eine `Map` hat nicht einen, sondern zwei Typparameter, den ersten für den Schlüssel und den zweiten für den Wert. Die `Map wordCount` ist eine Zuordnung von einem String-Schlüssel zu einem Integer-Wert.

Die `put`-Methode fügt ein Element hinzu (und überschreibt das alte Element, falls dem Schlüssel schon ein Wert zugeordnet war), die `get`-Methode gibt den einem Schlüssel zugeordneten Wert zurück. Die im Beispiel gezeigte Methode `getOrDefault` hat dieselbe Funktion. Sie können aber zusätzlich einen Default-Wert übergeben, der zurückgegeben wird, falls der angegebene Schlüssel nicht in der `Map` enthalten ist.

Die am häufigsten verwendete Implementierung von `Map` ist die im Beispiel gezeigte `HashMap`. Wie der Name andeutet, verwendet sie den Hashcode des Schlüssels, um die gesuchten Werte wiederzufinden. Das macht die Zugriffe auf eine `HashMap` sehr effizient, denn sie muss nicht komplett durchsucht werden, um herauszufinden, wo der Wert liegen könnte – eine Berechnung mit dem Hashcode liefert die richtige Stelle.

Zuordnungen von einem Objekt zum anderen werden häufig benötigt, schon für einfache Anwendungen, beispielsweise für eine Geburtstagsliste, die einem Namen ein Geburtsdatum zuordnet:

```
public class Geburtstagsliste {
    private Map<String, LocalDate> geburtstage = new TreeMap<>();

    public void fuegeGeburtstagHinzu(String name, LocalDate geburtstag){
        geburtstage.put(name, geburtstag);
    }

    public LocalDate findeGeburtstag(String name){
        return geburtstage.get(name);
    }

    public void schreibeGeburtstage(){
```

```
        for(Map.Entry<String, LocalDate> entry : geburtstage.entrySet()){
            System.out.println(entry.getKey() + ": " + entry.getValue());
        }
    }
}
```

Listing 10.35 Eine Geburtstagskarte – tolles Wortspiel, oder?

Sie sehen, wie einfach es ist, eine Geburtstagsliste mit einer `Map` zu erstellen. Die Methode `schreibeGeburtstage` enthält noch ein weiteres Feature von `Map`: Auch über sie können Sie iterieren. Die Methode `entrySet` liefert ein Set von `Map.Entry`-Objekten, die Key (Schlüssel) und Value (Wert) eines Eintrags enthalten. Die Methoden `keySet` und `values` (nicht `valueSet`!) liefern eine Collection aller Schlüssel bzw. eine Collection aller Werte.

10.6.1 Übung: Lieblingslieder

Schreiben Sie im `Musicplayer` eine Klasse `Lieblingslieder`, die für jeden Benutzer, identifiziert durch einen Benutzernamen, eine Liste seiner Lieblingslieder verwaltet. Es soll einem Benutzer möglich sein, der Liste ein Lied hinzuzufügen, sich alle für ihn gespeicherten Lieder zurückgeben zu lassen und die Liste komplett zu entfernen. Die Lösung zu dieser Übung finden Sie im Anhang.

10.7 Zusammenfassung

Sie haben in diesem Kapitel mehrere Wege kennengelernt, um mit Sammlungen von Objekten umzugehen. Sie haben in einer größeren Detailtiefe als zuvor vieles über Arrays gelernt, haben die verschiedenen Arten von Collections gesehen und den Unterschied zwischen ihnen und Arrays dargestellt bekommen. Sie haben außerdem den Umgang mit Typparametern gelernt, den sogenannten Generics, und wie sie Ihnen den Umgang mit Collections und vielen weiteren Klassen erleichtern.

Im nächsten Kapitel wird es zunächst mit Arrays und Collections weitergehen, denn Lambda-Ausdrücke kommen in diesem Umfeld gehäuft zum Einsatz.

Anhang B

Lösungen zu den Übungsaufgaben

B.1 Lösungen zu 2.2.4: Ausdrücke und Datentypen

	Ausdruck	Datentyp	Wert
1.	`7`	`int`	7
2.	`7.0`	`double`	7.0
3.	`7L`	`long`	7
4.	`5 + 9`	`int`	14
5.	`5 + 9.0`	`double`	14.0
6.	`5 + 9.0f`	`float`	14.0
7.	`5d + 9.0f`	`Double`	14.0
8.	`(byte) 5 + (short) 9`	`int`	14
9.	`17 << 2`	`int`	68
10.	`7L * 2`	`long`	14
11.	`5 / 2`	`int`	2
12.	`5 / 2.0`	`double`	2,5
13.	`int i = 0; i--;`	`int`	–1

Tabelle B.1 Ausdrücke und Datentypen

B.2 Lösung zu 3.1.3: Body-Mass-Index

```
package de.kaiguenster.javaintro.bmirechner;
import java.io.*;
import java.text.DecimalFormat;
import java.text.NumberFormat;
```

```
public class BMIRechner {

    public static void main(String[] args) throws IOException {
        //Zuerst werden die Nutzereingaben gelesen
        BufferedReader reader =
         new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Bitte geben Sie Ihre Größe in cm an");
        int groesseCM = Integer.parseInt(reader.readLine());
        System.out.println("Bitte geben Sie Ihr Gewicht in kg an");
        int gewicht = Integer.parseInt(reader.readLine());

        //Dann wird gerechnet
        double groesse = (double) groesseCM / 100;
        double bmi = gewicht / (groesse * groesse);

        //Zuletzt wird das Egebnis ausgegeben
        // Mit DecimalFormat wird der BMI für die Augabe formatiert
        NumberFormat formatter = new DecimalFormat("##.##");
        System.out.println("Ihr BMI ist " + formatter.format(bmi));
        if (bmi < 18.5){
            System.out.println("Damit haben Sie Untergewicht");
        } else if (bmi < 25){
            System.out.println("Damit haben Sie Normalgewicht");
        } else if (bmi < 30){
            System.out.println("Damit haben Sie Übergewicht");
        } else {
            System.out.println("Damit haben Sie schweres Übergewicht");
        }
    }
}
```

Listing B.1 BMI-Rechner

Das Programm verwendet fast ausschließlich Elemente, die Sie bereits kennen. Die einzige Ausnahme davon stellt die Klasse `NumberFormat` dar. Sie formatiert Zahlen für die Ausgabe und ist für die Lösung der Aufgabe nicht notwendig, es wird dadurch nur der ausgegebene Text verschönert. Sie werden sie in Kapitel 8, »Die Standardbibliothek«, näher kennenlernen.

Interessanter ist an dieser Stelle die Reihenfolge der `if`-Bedingungen: Es wird von der ersten Kategorie zur letzten geprüft. So ist garantiert, dass nur der richtige Codeblock ausgeführt wird.

B.3 Lösung zu 3.2.3: Boolesche Operatoren

- **Fragment 1**

```
if (i > 0 || j > 5){
    k = 10;
}
```

`i` ist zwar nicht größer als 0, aber `j` ist größer als 5. Somit ist die OR-Verknüpfung der beiden Bedingungen wahr, und der Rumpf des `if`-Konstrukts wird ausgeführt. Der Endzustand ist `i = 0`, `j = 7`, `k = 10`.

- **Fragment 2**

```
if (i > 0 && j > 5){
    k = 10;
}
```

Hier sind beide Bedingungen AND-verknüpft, der Gesamtausdruck ist falsch, da `i > 0` falsch ist. Endzustand: `i = 0`, `j = 7`, `k = 13`.

- **Fragment 3**

```
if ((i > 0 && j > 5) || k < 15){
    k = 10;
}
```

Die AND-Verknüpfung `i > 0 && j > 5` ist zwar falsch, aber der geklammerte Teilausdruck wird mit `k < 15` OR-verknüpft, also ist der Gesamtausdruck wahr. Endzustand: `i = 0`, `j = 7`, `k = 10`.

- **Fragment 4**

```
if ((i > 0 || j > 5) && k > 15){
    k = 10;
}
```

Die OR-Verknüpfung in Klammern ist wahr, weil `j > 5` wahr ist. Der Wert wird aber mit `k > 15` (falsch) AND-verknüpft, der Gesamtausdruck ist damit falsch. Endzustand: `i = 0`, `j = 7`, `k = 13`.

- **Fragment 5**

```
if (i == 0 & j++ < 5){
    k = 10;
}
```

Die beiden Bedingungen werden mit dem kurzschlussfreien `&`-Operator verknüpft, der Seiteneffekt `j++` wird deshalb in jedem Fall ausgeführt. Die Gesamtbedingung ist aber falsch, denn `j++` ist nicht kleiner als 5. Endzustand: `i = 0`, `j = 8`, `k = 13`.

- **Fragment 6**

```
if (i == 0 && j++ < 5){
    k = 10;
}
```

  In diesem Fragment wird zwar der &&-Operator mit Kurzschluss verwendet, da aber auf der linken Seite eine wahre Bedingung steht, muss die rechte Seite mit ihrem Seiteneffekt ausgeführt werden, um den gesamten Ausdruck auszuwerten. Endzustand: `i = 0`, `j = 8`, `k = 13`.

- **Fragment 7**

```
if (i != 0 && j++ < 5){
    k = 10;
}
```

  Jetzt steht auf der linken Seite des &&-Operators eine falsche Bedingung, deshalb wird die rechte Seite nicht ausgeführt. Außerdem ist der Gesamtausdruck falsch. Endzustand: `i = 0`, `j = 7`, `k = 13`.

- **Fragment 8**

```
if (i != 0 & j++ < 5){
    k = 10;
}
```

  Im letzten Fragment ist zwar die Gesamtbedingung falsch, da aber der &-Operator benutzt wurde, wird der Seiteneffekt `j++` ausgeführt. Endzustand: `i = 0`, `j = 8`, `k = 13`.

B.4 Lösung zu 3.2.4: Solitaire

```
package de.kaiguenster.javaintro.solitaire;

public class Solitaire {

    public static boolean kannAnlegen(String farbeAlt, int wertAlt,
     String farbeNeu, int wertNeu){
        boolean altRot = "Herz".equals(farbeAlt) || "Karo".equals(farbeAlt);
        boolean neuRot = "Herz".equals(farbeNeu) || "Karo".equals(farbeNeu);
        return altRot ^ neuRot && wertNeu + 1 == wertAlt;
    }

    public static void pruefeKarte(String farbeAlt, int wertAlt,
     String farbeNeu, int wertNeu, boolean erwartet){
        …
    }
```



```
    public static void main(String[] args) {
        …
    }
}
```

Listing B.2 Die Lösung der Solitaire-Übung

Der größte Teil des Codes war in diesem Fall vorgegeben und enthielt auch nur wenig Neues gegenüber den anderen Beispielen. Zwei Dinge sind aber eine Erwähnung wert. Das ist zum einen die Methode `System.out.print`. Sie erzeugt genau wie `System.out.println` eine Ausgabe auf die Kommandozeile, erzeugt aber im Anschluss keinen Zeilenumbruch. So können mehrere Ausgaben in eine Zeile geschrieben werden. Zum anderen ist es der Vergleich `erwartet == ergebnis`. `boolean`-Werte sind vergleichbar wie alle anderen primitiven Typen auch, hier ist es aber das erste Mal, dass in den Beispielen davon Gebrauch gemacht wird.

Der interessantere Teil besteht natürlich in der Methode, die zu entwickeln war. Es müssen zwei Bedingungen geprüft werden: ob der Wert der neuen Karte um genau eins niedriger ist als der Wert der alten Karte und ob die neue Karte rot ist, falls die alte Karte schwarz war, oder umgekehrt.

Beginnen wir mit der zweiten Bedingung. Sie lässt sich einfacher ausdrücken als: »wenn beide Karten unterschiedliche Farben haben«. Gemeint sind Farben im herkömmlichen Sinn, nicht Kartenfarben, also Rot oder Schwarz. So lässt sich die Bedingung noch anders ausdrücken: »Wenn genau eine der beiden Karten rot ist.« Im Alltag würde man es zwar so nicht sagen, aber es ist dennoch auf den ersten Blick korrekt: Um die Karte anlegen zu können, muss eine der beiden Karten rot sein und die andere schwarz. In dieser dritten Formulierung klingt die Bedingung schon ähnlich wie der XOR-Operator ^: Eine Karte muss rot sein, aber nicht beide dürfen es sein.

Mit diesen Vorüberlegungen gewappnet, können Sie eine der Bedingungen in Programmcode umsetzen. Zunächst müssen Sie feststellen, ob eine Karte rot ist. Das ist der Fall, wenn Ihre Kartenfarbe entweder Herz oder Karo ist. In Code ist das mit dem OR-Operator umzusetzen:

```
boolean altRot = "Herz".equals(farbeAlt) ||
                 "Karo".equals(farbeAlt);
```

Listing B.3 Prüfen, ob eine Karte rot ist

Diese Prüfung führen Sie für beide Karten durch und speichern das Ergebnis jeweils in einer `boolean`-Variablen. Beachten Sie dabei, dass die Strings wie beschrieben mit der `equals`-Methode verglichen werden und dass der OR-Vergleich mit Kurzschluss verwendet wird. Wenn schon bekannt ist, dass die Kartenfarbe Herz ist, ist der Vergleich mit Karo überflüssig.

Im nächsten Schritt ist noch zu prüfen, ob die Farben der Karten unterschiedlich waren. Im Beispiel ist diese Prüfung mit dem ^-Operator umgesetzt als `altRot ^ neuRot`, Sie könnten die beiden Variablen aber ebenso gut auf Ungleichheit prüfen, das Ergebnis wäre dasselbe: `altRot != neuRot`.

Die andere Bedingung ist schneller umzusetzen. Es gibt keinen Vergleichsoperator, der prüft, ob ein Wert um genau eins größer ist als der andere. Ersatzweise müssen Sie prüfen, ob ein Wert gleich dem anderen plus eins ist: `wertNeu + 1 == wertAlt`. Da beide Bedingungen zutreffen müssen, sind sie noch mit AND zu verknüpfen und ist das Ergebnis dann wie gezeigt zurückzugeben:

```
boolean ergebnis = (altRot ^ neuRot) && wertNeu + 1 == wertAlt;
return ergebnis;
```

Listing B.4 Das Endergebnis der Prüfung berechnen und zurückgeben

B.5 Lösung zu 3.3.2: »Rock im ROM«

```
package de.kaiguenster.javaintro.rockimrom;
import java.io.*;
public class RockImROM {
    public static void main(String[] args) throws IOException {
        System.out.println("An welchem Tag kommst du?");
        BufferedReader in = new BufferedReader(
         new InputStreamReader(System.in));
        String wochentag = in.readLine();
        String headliner = null;
        double preis = 0;
        switch(wochentag){
            case "Montag":
                headliner = "Rage against the Compiler";
                preis = 37.5;
                break;
            case "Dienstag":
                headliner = "if/else";
                preis = 22;
                break;
            …
            default:
                System.out.println("Den Tag kenn ich nicht: " + wochentag);
                System.exit(1);
        }
        System.out.println("Der Headliner am " + wochentag + " ist "
```



```
         + headliner);
        System.out.println("Kartenpreis: " + preis + "€");
    }

}
```

Listing B.5 »Rock im ROM«

Aus Platzgründen fehlen hier die Tage Mittwoch bis Sonntag, es passiert dort aber nichts anderes als auch schon am Montag und Dienstag. Wie Sie Text zum Benutzer ausgeben und Benutzereingaben einlesen, ist inzwischen ein alter Hut, der interessante Teil kommt danach. Zunächst werden für Headliner und Preis Variablen definiert. Da lokale Variablen keinen Default-Wert haben, muss ihnen ein initialer Wert von `null` bzw. `0` zugewiesen werden.

Anschließend wertet ein `switch`-Statement aus, welchen Tag der Benutzer eingegeben hat, und setzt die beiden Variablen entsprechend dem Programm. Der `default`-Block wird ausgeführt, wenn der Benutzer keinen gültigen Wochentag eingegeben hat. In diesem Fall gibt das Programm eine Fehlermeldung aus und beendet sich sofort.

Nach dem Ende von `switch` wird aus den gesetzten Variablen eine Ausgabe zusammengesetzt. Anstatt Variablen zu setzen und die Ausgabe erst am Ende zu machen, könnten Sie auch in jedem `case` die Ausgabe machen. Die gezeigte Variante hat den Vorteil, dass der Code zur Ausgabe, der ja in allen Fällen gleich ist, nur einmal geschrieben wird. Doppelten Code zu vermeiden, ist ein weiterer Aspekt von gutem Programmierstil: Das Programm wird kürzer und übersichtlicher, und wenn Sie später etwas ändern wollen, zum Beispiel einen anderen Text ausgeben, müssen Sie die Änderung nur einmal vornehmen und nicht siebenmal.

B.6 Lösung zu 3.3.5: »Rock im ROM« bis zum Ende

```
package de.kaiguenster.javaintro.rockimrom;
import java.io.*;
public class RockImROM {
    public static void main(String[] args) throws IOException {
        System.out.println("An welchem Tag kommst du?");
        BufferedReader in = new BufferedReader(
         new InputStreamReader(System.in));
        String wochentag = in.readLine();
        String headliner = "";
        double preis = 0;
        switch(wochentag){
```

```
            case "Montag":
                headliner += "Rage against the Compiler\n";
                preis += 37.5;
            …
            case "Sonntag":
                headliner += "Delphi and the Oracles\n";
                preis += 35;
                break;
            default:
                System.out.println("Den Tag kenn ich nicht: " + wochentag);
                System.exit(1);
        }
        System.out.println("Die Headliner sind:\n" + headliner);
        System.out.println("Kartenpreis: " + preis + "€");
    }
}
```

Listing B.6 »Rock im ROM« die Zweite

Viel hat sich gegenüber der Vorversion gar nicht geändert. Die beiden Variablen werden nun nicht mehr an jedem Wochentag gesetzt, es wird zu ihrem Wert addiert bzw. sie werden konkateniert. Ein kleiner, aber gemeiner Fallstrick ist dabei, dass die Variable `headliner` nun nicht mehr mit `null` initialisiert wird, sondern mit dem leeren String `""`. Diese Änderung ist notwendig, da sonst das Wort »null« im zusammengesetzten Ergebnis auftaucht.

Außerdem sind fast alle `break`-Befehle verschwunden. Aber nur fast alle: Am letzten Tag muss die Ausführung noch immer unterbrochen werden, sonst fällt sie in den `default`-Block durch und beendet ebenfalls das Programm mit einer Fehlermeldung.

B.7 Lösung zu 3.3.6: »Rock im ROM« solange ich will

```
package de.kaiguenster.javaintro.rockimrom;
import java.io.*;
public class RockImROM {
    public static void main(String[] args) throws IOException {
        System.out.println("An welchem Tag kommst du?");
        BufferedReader in = new BufferedReader(
         new InputStreamReader(System.in));
        String starttag = in.readLine();
        System.out.println("Und bis wann bleibst du?");
        String endtag = in.readLine();
```



```
        String headliner = "";
        double preis = 0;
        switch(starttag){
            case "Montag":
                headliner += "Rage against the Compiler\n";
                preis += 37.5;
                if ("Montag".equals(endtag)){
                    break;
                }
            case "Dienstag":
                headliner += "if/else\n";
                preis += 22;
                if ("Dienstag".equals(endtag)){
                    break;
                }
            …
            case "Sonntag":
                headliner += "Delphi and the Oracles\n";
                preis += 35;
                break;
            default:
                System.out.println("Unbekannter Tag oder Endtag liegt vor
                 Starttag: " + starttag + " - " + endtag);
                System.exit(1);
        }
        System.out.println("Die Headliner sind:\n" + headliner);
        System.out.println("Kartenpreis: " + preis + "€");
    }
}
```

Listing B.7 »Rock im ROM« die Dritte

Auch diesmal hat sich gegenüber der vorherigen Version nicht viel geändert. Am Anfang des Programms werden An- und Abreisetag abgefragt. In jedem `case`-Block befindet sich nun ein `break`, das nur dann ausgeführt wird, wenn der Tag der Endtag ist.

Jetzt wird auch das `break` am Sonntag nicht mehr immer ausgeführt, sondern nur, wenn Sonntag der Endtag ist. Es gibt nur zwei Fälle, in denen die Ausführung hier noch zum `default`-Block durchfallen kann: Entweder es wurde ein ungültiger Start- oder Endtag eingegeben, oder der Endtag liegt vor dem Starttag. In beiden Fällen ist es richtig, das Programm mit einer Fehlermeldung zu beenden. Die Fehlermeldung hat sich entsprechend geändert und weist auf beide Fehlermöglichkeiten hin.