Einführung in

Java

18. Dezember 1998

von

Jens Scheffler, Michael Ohr, Oliver Wagner, Thomas Much

Inhaltsverzeichnis

1 Einleitung

1.1 Allgemeine Hinweise

Dieses Skript soll in die Programmierung mit Java einführen. Zielgruppe sind dabei insbesondere diejenigen, die sehr wenig oder gar keine Programmiererfahrung besitzen. Das Skript setzt nur voraus, daß man mit seinem Computer bereits umgehen kann (Programme starten, Texte eingeben) und daß man auf ein vorinstalliertes Java zurückgreifen kann — am besten fragt man dazu einen guten Bekannten, denn die meisten Java-Systeme sind für Einsteiger leider nicht selbsterklärend. Für das Windows-Betriebssystem ist die JDK-Installation im Anhang ausführlich beschrieben. Wer bereits andere Programmiersprachen kennt, wird vieles überspringen können, und „Programmierprofis“ werden vermutlich mit einer der gängigen Java-Referenzen glücklicher. Wer sollte also was in diesem Skript lesen?

Anfänger Der folgende Abschnitt „Mein erstes Programm“ ist für all diejenigen gedacht, die keine oder sehr wenig Programmiererfahrung besitzen. Hier wird ein kleines, überschaubares Programm ausführlich besprochen, von der Eingabe bis zum Ausführen.

Fortgeschrittene Wer bereits eine andere Programmiersprache beherrscht, wird Kapitel 1 (siehe 1.2) vermutlich nur überfliegen, um einen ersten Eindruck über die Unterschiede seiner Programmiersprache zu Java zu bekommen. In Kapitel 2 (siehe 2) werden dann genaue Definitionen der Sprache Java geboten, und in Kapitel 3 (siehe 3) kann man sein neu erworbenes Wissen anhand von Übungsaufgaben vertiefen.

Profis Wer Java bereits zur Programmierung verwendet, besitzt sicherlich schon passende Literatur dazu — wenn nicht, sind im Abschnitt „Weiterführende Dokumentation“ ein paar Standardwerke aufgelistet. Ob man Java wirklich beherrscht und Problemstellungen algorithmisch lösen kann, sollten aber auch Profis mit den Aufgaben aus Kapitel 3 (siehe 3) prüfen.

1.1.1 Download und Ausdruck dieses Skripts

Auf der AIFB-Homepage sowie im `download`-Verzeichnis jedes `waifb`-Accounts liegt die Postscript-Datei `j1skript.ps`, die man mit dem Unix-Kommando `pm` einfach ausdrucken kann.

Wer die HTML-Version des Skripts zu Hause lesen möchte, kann sich auf der AIFB-Homepage das passende tar-Archiv herunterladen. Das Archiv läßt sich mit den gängigen Entpackern (WinZip, StuffIt Expander und natürlich tar) auspacken. Die Startseite `index.html` kann dann mit einem HTML-Browser (Netscape Navigator, Microsoft Internet Explorer, Lynx) geöffnet werden.

1.1.2 Installation

Die Installation des **Java Development Kits (JDK)**, also des Java-Entwicklungssystems, ist auf den verschiedenen Betriebssystemen (Windows, MacOS, Linux usw.) unterschiedlich, weshalb wir leider keine allgemeingültigen Arbeitsschritte anbieten können.

Der Anhang beschreibt aber die JDK-Installation für Windows 95 und 98 (siehe A) sehr ausführlich und ist speziell auf den AB-Pool im Rechenzentrum der Universität Karlsruhe sowie auf die „Programmieren 1"-Homepage zugeschnitten, so daß auch Einsteiger damit klarkommen sollten. Ansonsten kann auch die bestehende Java-Installation im AB-Pool problemlos genutzt werden.

Die vorgestellten Java-Quelltexte setzen Java 1.1 voraus, so daß man mit den weit verbreiteten JDK-Versionen 1.1.5 und 1.1.6 optimal für diesen Kurs ausgestattet ist.

Informationen zur Konfiguration des Rechnerzugangs per Modem von zu Hause aus befinden sich auf

`http://www.uni-karlsruhe.de/~stud/studinfo.html`

1.1.3 Weiterführende Dokumentation

Java wird oft als die Sprache des Internet bezeichnet, also ist es nur allzu logisch, daß es die vollständige Dokumentation dazu im Internet bzw. World Wide Web gibt. Folgende Adressen sollte sich jeder einmal ansehen:

- `http://java.sun.com/docs/books/tutorial/`

  Das **Java Tutorial** ist eine gute und ausführliche Einführung in Java, allerdings sollte man schon ein paar Computer-Erfahrungen mitbringen.

- `http://www.uni-karlsruhe.de/~Java/jdk1.1.5/docs/index.html`

  Das Rechenzentrum stellt eine lokale Kopie der **Java-API („Application Programming Interface")** zur Verfügung. Hier sind alle

Befehle und Funktionen, die Java bietet, aufgelistet. Wenn man also genau weiß, daß Java eine bestimmte Funktion anbietet, sich aber nicht an die Parameter erinnern kann, die diese Funktion erwartet, ist diese Adresse die erste Stelle zum nachschauen.

Wer bedrucktes Papier bevorzugt, wird in der Bibliothek fündig:

- Mary Campione, Kathy Walrath, *„The Java Tutorial: Object-Oriented Programming for the Internet"*, 2. Auflage, Addison-Wesley, Februar 1998; ISBN 0-201-31007-4

  Dies ist die gedruckte Ausgabe der oben erwähnten Online-Version. Wer des Englischen nicht mächtig ist, kann zur Not auch die deutsche Ausgabe verwenden.

- David Flanagan, *„Java in a Nutshell"*, 2. Auflage, O'Reilly, Mai 1997; ISBN 1-56592-262-X

  Java in a Nutshell ist eine gute Kurzreferenz für die Standard-Java-API. Um dieses Buch sinnvoll einsetzen zu können, sollte man allerdings schon programmieren können. Für C-Umsteiger wird ein kurzer und hervorragender Crashkurs geboten. Auch hiervon gibt es eine deutsche Übersetzung.

Schließlich gibt es im Internet (bzw. Usenet) noch zahllose Diskussionsforen rund um Java. Hier eine kleine Auswahl:

- `news:comp.lang.java.help`
- `news:comp.lang.java.programmer`
- `news:de.comp.lang.java`

1.1.4 Kontakt zu den Autoren

Die Autoren dieses Skripts sind Studenten, die bereits als Tutoren Erfahrung im Unterrichten von Java gesammelt haben. Sollten in diesem Skript Fehler oder Unklarheiten auftauchen — was leider nie ganz auszuschließen ist — sind wir unter den unten angegebenen Adressen zu erreichen. *Wichtig:* Allgemeine Fragen zu Java können immer noch am besten im Tutorium gestellt und beantwortet werden. Wir (die Autoren) bekommen auch ohne solche Fragen schon genug Post.

Autor	Email	Kapitel
Thomas Much	waifbt14@rz.uni-karlsruhe.de	1, Layout
Jens Scheffler	waifbt19@rz.uni-karlsruhe.de	2, 3, 5, Anhang
Michael Ohr	waifbt15@rz.uni-karlsruhe.de	2
Oliver Wagner	waifbt21@rz.uni-karlsruhe.de	4, Anhang

1.2 Mein erstes Programm

Wenn man an Computer denkt, fallen einem zuerst wohl Berechnungen ein, die man damit anstellen kann[1]. Jeder hat wohl schon mit einem Taschenrechner gearbeitet, und daher wollen wir mit einer sehr einfachen Rechenaufgabe anfangen.

Ziel dieses Kapitels ist es, das folgende kleine Beispielprogramm zu verstehen. Es macht nichts weiter, als 3 plus 4 zu berechnen und anschließend auf dem Bildschirm auszugeben. Im folgenden werden wir das Programm Zeile für Zeile untersuchen, in den Computer eingeben und schließlich auch ausführen lassen.

```
public class Berechnung {

  public static void main(String[] args) {

    int i;

    i = 3 + 4;

    System.out.println(i);
  }

}
```

Bitte nicht erschrecken, wenn dieser Programmtext relativ groß für eine einfache Berechnung wirkt. Wie wir später sehen, benötigt jedes Java-Programm ein paar einleitende und abschließende Worte (wie ein guter Roman), die immer gleich sind[2].

1.3 Formeln und Ausdrücke

Sehen wir uns zunächst einmal die Zeile in der Mitte an:

```
i = 3 + 4;
```

Diese Zeile sollte jeder, der schon einmal eine physikalische oder mathematische Formel gesehen hat, verstehen. Wir addieren 3 und 4 und weisen

[1] Wer hier gerade laut „Spiele“ rufen wollte, dem sei gesagt, daß es auch in Java programmierte Spiele gibt, z.B. unter der Adresse http://www.uni-karlsruhe.de/~ua4d/herpes. Allerdings wird sich der Lernende das vorliegende Skript sehr genau zu Gemüte führen müssen, um solche komplexen Programme verstehen oder schreiben zu können.

[2] Zumindest so lange, bis wir den Sinn dieses Rahmens verstanden haben und danach dann auch verändern können.

das Ergebnis der Variablen `i` zu. Natürlich sind in Java auch Formeln der Form „a=b+c“ möglich, und sobald wir die verschiedenen Operatoren von Java kennengelernt haben, können wir praktisch alle Formeln aus unserem Physikbuch direkt abtippen.

Wir sehen aber auch einen wichtigen Unterschied: Die Formel ist *mit einem Semikolon abgeschlossen*, damit der Computer weiß, wo das Ende der Formel ist. Im folgenden werden wir solche Formeln **Ausdrücke** nennen (andere Programmiersprachen verwenden hierfür die Bezeichnung **Befehl** oder **Anweisung**). *Jeder* Ausdruck muß in Java mit einem Semikolon abgeschlossen werden — wird dies vergessen, liefert uns der Java-Compiler beim Übersetzen eine Fehlermeldung.

Auch wenn Java es erlaubt, mehrere Ausdrücke pro Zeile zu schreiben, sollten wir versuchen, möglichst immer nur einen Ausdruck in jede Zeile zu packen. Andernfalls wird unser Programmtext sehr unübersichtlich, und schon eine Woche später können wir dann aus dem Programmtext nicht mehr herauslesen, was wir eigentlich programmiert haben... Wie so oft gilt also auch hier: *Weniger* (Ausdrücke pro Zeile) *ist mehr* (Übersichtlichkeit).

1.4 Zahlenbeispiele

Als nächstes sehen wir uns die Zahlen an, die wir addieren. Während es für uns fast egal ist, ob wir in einer Formel mit ganzen Zahlen rechnen oder mit solchen, die Nachkommastellen haben (wir nennen sie im folgenden „Gleitkommazahlen“[3]), ist dies für Computer ein elementarer Unterschied.

Die beiden Zahlen des Beispielprogramms sind **ganze Zahlen**, auf englisch **integer**. Hier ein paar Beispiele für Integer-Zahlen:

```
          0
          1
         -1
2147483647
```

Wenn eine ganze Zahl eine gewisse Größe über- oder unterschreitet, muß an die Zahl ein großes `L` (für **long integer**, lange Ganzzahl) angehängt werden. Die Grenze dafür liegt ungefähr bei +/- 2,1 Milliarden (siehe Kapitel 2 (siehe 2.3.1)). Es ist natürlich auch erlaubt, bei kleineren Zahlen ein `L` anzuhängen, dann wird aber meistens unnötig Speicherplatz verschwendet.

[3] Dies deutet nicht nur auf den Nachkommateil hin, sondern beschreibt auch die interne Zahlendarstellung des Computers. Wie diese interne Darstellung genau aussieht, ist Thema der Vorlesung „Info A“ bzw. „Technische Informatik“.

Der genaue Unterschied wird später noch erläutert, hier erst einmal einige Beispiele für lange Integer-Zahlen:

```
      2500000000L
-1234567890876543L
                0L
               -1L
```

Im Gegensatz zu den ganzen Zahlen besitzen **Gleitkommazahlen** einen Vor- und Nachkommateil. Ebenso können wir die wissenschaftliche **Exponentenschreibweise** verwenden. Aber Achtung: Obwohl sie Gleit*komma*zahlen heißen, müssen wir das englischen Zahlenformat mit einem Punkt als Dezimaltrenner verwenden (das englische „floating point numbers“ ist hier eindeutiger). Hier nun einige Gleitkommazahlen:

```
          0.0
          1.0
         -1.0
2147483647.0
         42.314159
         -3.7E2
          1.9E-17
```

Die Exponentenschreibweise bei den letzten beiden Zahlen sollte vom Taschenrechner her bekannt sein.

1.5 Verwendung von Variablen

Noch vor der eigentlichen Berechnung finden wir die Zeile

```
int i;
```

Damit sagen wir dem Computer, daß wir in unserem Programm in den folgenden Zeilen die Variable mit dem Namen `i` verwenden möchten. Diese Zeile, die wir **Variablendeklaration** nennen, ist auch ein Ausdruck, also steht auch hier ein abschließendes Semikolon.

`i` ist ein recht kurzer Name für eine Variable, später werden wir nach Möglichkeit immer längere — und damit aussagekräftigere — Namen verwenden, beispielsweise `summe`. Welche Namen wir für Variablen (siehe 2.4.1) wählen können, ist in Kapitel 2 (siehe 2.1.2) genauer erläutert.

Mit `int` legen wir den **Datentyp** der Variablen fest. `int` steht dabei für **integer**, d.h. (wir erinnern uns an den vorangegangenen Abschnitt) daß diese Variable ganze Zahlen im Bereich von ca. +/- 2,1 Milliarden aufnehmen kann. In Kapitel 2 wird eine Übersicht über die verschiedenen Datentypen (siehe 2.3) gegeben.

1.6 „Auf den Schirm!“

Wir möchten das Ergebnis unserer komplizierten Berechnung natürlich auch auf dem Bildschirm ausgeben. Dies geschieht mit der Zeile

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

`System.out.println` ist die Funktion (in Java **Methode** genannt), mit der man Text und Zahlen auf dem Bildschirm ausgeben kann[4]. In Klammern folgt nach dem Methodennamen dann das, was wir ausgeben wollen, in diesem Fall also die Variable `i`. Auch diese Zeile ist laut Java-Sprachdefinition ein Ausdruck, also dürfen wir auch hier das Semikolon nicht vergessen.

Die erwähnte Methode ist recht flexibel anwendbar. Wir können beispielsweise auch Text ausgeben, der in Anführungszeichen eingeschlossen wird:

```
System.out.println("Das Ergebnis ist: ");
```

Wenn man beides (Zahlen und Text) kombinieren möchte, gibt es zwei Möglichkeiten. Zum einen können wir ganz einfach zwei Zeilen verwenden:

```
System.out.print("Das Ergebnis ist: ");
System.out.println(i);
```

Bitte beachten Sie, daß in der ersten Zeile nur `print` (und nicht `println`) steht! Das bedeutet, daß nach der ersten Ausgabe *kein* Zeilenvorschub („line feed“, im Methodennamen mit „ln“ abgekürzt wiederzufinden) durchgeführt wird. Auf dem Bildschirm steht die Ausgabe also in einer Zeile, obwohl wir im Quelltext zwei Zeilen dafür verwenden:

```
Das Ergebnis ist: 7
```

Da Java eine etwas komplexere Programmiersprache ist, können wir die beiden Zeilen auch zu einer zusammenfassen:

```
System.out.println("Das Ergebnis ist: " + i);
```

Wir verknüpfen also einfach alle Teile der Ausgabe mit dem **+**-Operator, der in Java nicht nur auf Zahlen, sondern auch auf Texte angewendet werden kann!

Wichtig: Wie wir später noch sehen, kann die Vermischung von Texten und Zahlen unter Umständen zu Problemen führen. Zunächst aber können wir die hier vorgestellten Methoden bedenkenlos verwenden.

[4] Warum die Methode einen so langen und dreigeteilten Namen besitzt, werden wir erst sehr viel später sehen. Im Moment soll uns die Tatsache genügen, daß die Methode genau das tut, was wir benötigen.

1.7 Das Programmgerüst

Nun fehlt uns zum Verständnis unseres ersten Programms nur noch der Rahmen, den wir bis auf weiteres in jedem Quelltext verwenden. Dazu ist es wichtig zu wissen, daß Java zur Strukturierung der Quelltexte **Blöcke** vorsieht, die mit einer öffnenden geschweiften Klammer { begonnen und mit einer schließenden geschweiften Klammer } beendet werden. Im Quelltext können wir demnach zwei Blöcke ausmachen:

Die Klasse Vielleicht haben Sie schon irgendwo gelesen, daß Java eine **objektorientierte** Programmiersprache ist. Was das genau bedeutet, soll uns hier zunächst nicht weiter interessieren, aber wie viele objektorientierte Sprachen erzeugt Java seine Objekte aus sogenannten **Klassen.** Eine Klasse ist demnach die oberste Struktureinheit und sieht folgendermaßen aus:

```
public class Berechnung {
  // hier steht sonst der Rest des Quelltexts
}
```

Vor der öffnenden Klammer steht der Name der Klasse, in diesem Fall „Berechnung“ — so heißt schließlich auch unser Programm. *Wichtig:* Der Name der Klasse muß *exakt* (Groß-/Kleinschreibung**!**) dem Dateinamen entsprechen, unter dem wir diese Klasse speichern, wobei der Dateiname noch die Erweiterung `.java` trägt. In unserem Fall *müssen* wir die Klasse also in der Datei `Berechnung.java` speichern (siehe 1.8)**!**

Vor dem eigentlichen Namen stehen noch die beiden **Schlüsselworte** `public` und `class`. Wir wollen dies im Moment einfach akzeptieren, dürfen sie aber bei eigenen Klassen auf gar keinen Fall vergessen.

Die erlaubten Klassennamen unterliegen gewissen Regeln (siehe 2.1.2), die wir später noch kennenlernen. Wir sollten uns aber auf alle Fälle daran halten, einen Klassennamen immer mit einem *Großbuchstaben* zu beginnen.

Die Hauptroutine Innerhalb von Klassen gibt es untergeordnete Struktureinheiten, die **Methoden.** Jede Klasse besitzt bei uns dabei bis auf weiteres die Methode `main`, ohne die wir das Java-Programm nicht ausführen könnten[5]:

[5] Wenn wir später Java wirklich objektorientiert kennengelernt haben, werden wir auch Klassen ohne diese Methode benutzen — aber bis dahin dauert es noch einige Zeit.

```
public static void main(String[] args) {
  // hier steht eigentlich die Berechnung
}
```

Wir erkennen den erwähnten Methodennamen `main`. Den Rest der ersten Zeile müssen wir zunächst ganz einfach auswendig lernen — man achte dabei insbesondere auf das großgeschriebene „S“ bei `String`.

Kapitel 2 (siehe 2.2.1) geht noch einmal auf das Programmgerüst ein. Richtig verstehen werden wir es aber erst sehr viel später.

1.8 Eingeben, übersetzen und ausführen

Jetzt sind wir soweit, daß wir den Quelltext des Programms eingeben können. Danach muß der Quelltext vom **Java-Compiler** in einen interpretierbaren Code, den sogenannten **Java Bytecode**, übersetzt werden. Dieser Bytecode kann dann vom **Java-Interpreter** ausgeführt werden.

In diesem Abschnitt zeigen wir diese Vorgehensweise für das Betriebssystem Unix (oder Varianten davon, beispielsweise Linux). Bei anderen Betriebssystemen ist das Vorgehen meist recht ähnlich.

Nach dem Einloggen mit dem Symbol „KURS“ erscheint auf dem Bildschirm der Eingabeprompt („`[~]`“). Nun können wir diverse Unix-Kommandos eingeben, die letztendlich zur Ausführung des Java-Programms führen. Bei der folgenden Schritt-für-Schritt-Anleitung muß die Groß-/Kleinschreibung *unbedingt* beachtet werden! Nach jeder Zeile muß [Return] bzw. [Enter] gedrückt werden, um das Kommando auszuführen.

mkdir Berechnung Ein Unterverzeichnis mit dem Namen „`Berechnung`“ wird angelegt. Damit sorgen wir auf unserem Account für Ordnung, um auch später noch die Übersicht behalten zu können. Es ist eine gute Idee, jedes Verzeichnis nach dem Programm zu benennen.

cd Berechnung Mit diesem Kommando wechseln wir vom Heimatverzeichnis in das soeben erzeugte Verzeichnis „`Berechnung`“. Durch Eingabe von `cd` kommt man jederzeit wieder zurück ins Heimatverzeichnis, aber wir bleiben erstmal im Verzeichnis „`Berechnung`“.

fte Berechnung.java Nun rufen wir den Editor „FTE“ auf und sagen ihm gleichzeitig, daß wir die Datei `Berechnung.java` bearbeiten möchten. Da diese Datei noch nicht existiert, wird sie FTE automatisch neu anlegen.

Nun kann der Quelltext (siehe 1.2) aus diesem Kapitel Zeichen für Zeichen *exakt* abgetippt werden — oder aber man kopiert den Quelltext direkt aus dem Browser-Fenster per Copy&Paste in das FTE-Fenster.

Wenn der Quelltext eingegeben wurde, muß dieser gespeichert werden. Danach verläßt man den Editor und landet wieder beim Eingabeprompt der Unix-Shell.

javac Berechnung.java Nun lassen wir den Quelltext vom Java-Compiler (das Programm heißt `javac`) in Java-Bytecode übersetzen. Das kann ein bißchen dauern. Wenn alles geklappt hat, landen wir einfach wieder beim Eingabeprompt. Wenn Fehlermeldungen auf dem Bildschirm ausgegeben werden, haben Sie vermutlich den Quelltext nicht exakt abgetippt — versuchen Sie, den Fehler zu finden. Wie man Fehler erkennt und anhand der Fehlermeldung beseitigt, werden wir später genauer sehen.

ll Mit diesem Kommando lassen wir uns den Inhalt Verzeichnisses „`Berechnung`“, in dem wir uns immer noch befinden, anzeigen. Dort taucht unter anderem die Datei `Berechnung.class` auf. Dies ist die vom Java-Compiler erzeugte Bytecode-Datei, also das ausführbare Java-Programm.

java Berechnung Schließlich starten wir mit diesem Kommando das Java-Programm (`java` ist der Java-Interpreter). *Wichtig:* Während beim Java-Compiler der Dateiname komplett angegeben werden muß (also mit der Erweiterung `.java`), darf diese beim Java-Interpreter *nicht* angegeben werden.

Auf dem Bildschirm sollte nun nach kurzer Zeit ganz einfach eine „7“ ausgegeben werden. **Herzlichen Glückwunsch!** Experimentieren Sie ruhig noch ein wenig mit dem Programm herum (andere Zahlen, `+`-Operator ersetzen, Ausgabe verändern). Damit alle Änderungen, die Sie vornehmen, auch übersetzt werden, müssen sie immer alle Schritte ab dem oben erwähnten „`fte Berechnung.java`“ ausführen. [6]

Viel Spaß beim weiteren Lernen von Java, Kapitel 2 (siehe 2) wartet schon!

[6] Mit der Zeit werden Sie lernen, welche der Schritte Sie weglassen können bzw. wie man die Schritte optimieren kann. Um Fehler zu vermeiden, sollten Sie sich aber zunächst an diese Reihenfolge halten.

2 Grundlagen der Programmierung in Java

2.1 Grundelemente eines Java-Programmes

Das Erlernen einer Programmiersprache unterscheidet sich im Grunde nicht sonderlich vom Englisch- oder Französischunterricht in der Schule. Wir haben eine gewisse Grammatik, nach deren Regeln wir Sätze bilden können – und eine Unmenge an Vokabeln, die wir für diese Sätze brauchen. Wir formen unsere Sätze aus den gelernten Worten und können diese zu einem komplexeren Gebilde zusammenfügen – beispielsweise einer Geschichte oder einer Bedienungsanleitung für einen Toaster.

In Java (oder einer anderen Programmiersprache) funktioniert das Ganze auf dieselbe Art und Weise. Wir werden lernen, nach gewissen Regeln mit dem Computer zu „sprechen“, d.h. ihm verständlich zu machen, was er für uns zu tun hat. Damit uns dies gelingt, müssen wir uns zuerst mit gewissen Grundelementen der Sprache vertraut machen.

2.1.1 Kommentare

Wer kennt die Situation nicht? Man hat einen längere Rechnung durchgeführt, einen Artikel verfaßt oder irgendeine Skizze erarbeitet – und muß diese Arbeit nun anderen Personen erklären. Leider ist die Rechnung, der Artikel oder die Skizze schon ein paar Tage alt und man erinnert sich nicht mehr an jedes Detail, jeden logischen Schritt. Wie soll man seine Arbeit auf die Schnelle nachvollziehen?

In wichtigen Fällen hat man deshalb bereits beim Erstellen dafür gesorgt, daß andere (oder man selbst) seine Arbeit auch später noch verstehen kann. Hierzu werden Randnotizen, Fußnoten und erläuternde Diagramme verwendet — zusätzliche Kommentare also, die jedoch nicht Bestandteil des eigentlichen Papiers sind.

Auch unsere Programme werden mit der Zeit immer größer werden. Wir brauchen deshalb eine Möglichkeit, unseren Text mit erläuternden Kommentaren zu versehen. Da sich Textmarker auf dem Monitor jedoch schlecht macht, hat die Sprache Java ihre eigene Art und Weise, mit Kommentaren umzugehen:

Angenommen, Wir haben eine Programmzeile verfaßt und wollen uns später daran erinnern, was es mit dieser auf sich hat. Die einfachste Art, dies zu tun, ist die folgende:

```
a = b + c;  // hier beginnt ein Kommentar
```


Sobald der Java-Compiler die Zeichen `//` in einer Zeile findet, erkennt er einen Kommentar. Alles, was nach diesen Zeichen folgt, geht Java „nichts mehr an“ und wird vom Übersetzer ignoriert. Der Kommentar endet, wenn auch die Zeile endet.

Manchmal kann es jedoch vorkommen, daß sich Kommentare über mehr als eine Zeile erstrecken. Wir können natürlich jede Zeile mit einem Kommentarzeichen versehen, etwa wie folgt:

```
// Zeile 1
// Zeile 2
// Zeile 3
// ...
// Zeile n
```

Dies bedeutet jedoch, daß wir nicht vergessen dürfen, zu Anfang jeder Zeile die Kommentarzeichen zu machen. Wollen wir etwas am Kommentar ändern und fügen deshalb neuen Text ein, bedeutet dies also ziemliche Vorsicht. Java stellt aus diesem Grund eine zweite Form des Kommentars zur Verfügung. Wir beginnen einen mehrzeiligen Kommentar mit den Zeichen `/*` und beenden ihn schließlich mit `*/`. Zwischen diesen Zeichen kann ein beliebig langer Text stehen, wie folgendes Beispiel zeigt:

```
/* Kommentar...
   Kommentar...
   immer noch Kommentar...
   letzte Kommentarzeile...
*/
```

Wir wollen uns bezüglich der Kommentierung unserer Programme einige Dinge zur Angewohnheit machen. Viele Programme bedeuten meist auch viele Dateien — man verliert leichter den Überblick, als man denkt.

Um uns auch gleich den richtigen Stil beim Kommentieren von Java Quelltexten anzugewöhnen, wollen wir uns von Anfang an das sogenannte JavaDoc Format halten. JavaDoc ist ein sehr hilfreiches Zusatzprogramm, das Sun — die Firma, die Java sozusagen „erfunden“ hat — jedem JDK (Java Development Kit, siehe JDK-Installation unter Windows 95/98) kostenlos beifügt. Mit Hilfe von JavaDoc lassen sich nach erfolgreicher Programmierung automatisch vollständige Dokumentationen zu den erstellten Programmen generieren, was einem im nachhinein sehr viel Arbeit ersparen kann.

Der Funktionsumfang von JavaDoc ist wesentlich größer, als wir ihn im Rahmen dieser Vorlesung behandeln könnten. Außerdem wird JavaDoc von

Sun ständig weiterentwickelt — dem interessierten Leser sei die Dokumentation von JavaDoc, die in der Online Dokumentation eines jeden JDKs enthalten ist, wärmstens ans Herz gelegt.

JavaDoc Kommentare beginnen — ähnlich wie allgemeine Kommentare — stets mit der Zeichenkette `/**` und enden mit `*/`. Es hat sich eingebürgert, zu Beginn jeder Zeile des Kommentars einen zusätzliches `*` zu setzen, um Kommentare auch optisch vom Rest des Quellcodes abzusetzen.

Ein typischer JavaDoc Kommentar zu Beginn wäre zum Beispiel folgender:

```
/**
 * Dieses Programm berechnet die Lottozahlen von naechster Woche. Dabei
 * erreicht es im Schnitt eine Genauigkeit von 99,5%.
 *
 * @author Hans Mustermann
 * @date 1998-10-26
 * @version 1.0
 */
```

Schauen wir uns mal die gemachten Angaben im einzelnen an:

- Die ersten Zeilen enthalten eine allgemeine Beschreibung des vorliegenden Programms. Dabei ist vor allem darauf zu achten, daß die gemachten Angaben auch ohne den vorliegenden Quelltext Sinn machen sollten, da JavaDoc aus diesen Kommentaren später eigenständige Dateien erzeugt — im Idealfall muß jemand, der die von JavaDoc erzeugten Hilfsdokumente liest, ohne zusätzlichen Blick auf den Quellcode verstehen können, was das Programm macht und wie man es aufruft.
- Als nächstes sehen wir verschiedene Bezeichner, die stets mit dem Zeichen `@` eingeleitet werden. Hier kann man bestimmte, vordefinierte Informationen zum vorliegenden Programm angeben. Dabei spielt die Reihenfolge der Angaben keine Rolle. Auch erkennt JavaDoc mittlerweile wesentlich mehr Bezeichner als die hier aufgeführten, aber wir wollen uns mal auf die wesentlichsten konzentrieren. Die hier vorgestellten Felder sind im einzelnen:
  - **@author** der Autor des vorliegenden Programmes
  - **@date** das Erstellungsdatum des vorliegenden Programmes
  - **@version** die Versionsnummer des vorliegenden Programmes

Übrigens: Sollte jemand von euch über den passenden Quellcode zu obigem JavaDoc Kommentar stolpern, wäre ich an einer Kopie interessiert `;-)`

2.1.2 Bezeichner

Wir werden später oft in die Verlegenheit kommen, irgendwelchen Dingen einen Namen geben zu müssen – beispielsweise als Platzhalter, um eine Rechnung mit verschiedenen Werten durchführen zu können. Hierzu müssen wir jedoch wissen, wie man in Java solche Namen vergibt.

Ein Bezeichner setzt sich in Java aus folgenden Elementen zusammen:

- den **Buchstaben** des Alphabets (d.h. `a,b,c,...`). Java unterscheidet hierbei zwischen Groß- und Kleinschreibung.

  Da es sich bei Java um eine internationale Programmiersprache handelt, läßt der Sprachstandard hierbei diverse landesspezifische Erweiterungen zu. So sind etwa japanische Katakana-Zeichen, kyrillische Schrift oder auch die deutschen Umlaute gültige Buchstaben, welche in einem Bezeichner vewendet werde dürfen. Wir werden in diesem Skript auf solche Zeichen jedoch bewußt verzichten, da der Austausch derartig verfaßter Programme oftmals zu Problemen führt. So werden auf verschiedenen Betriebssystemen die deutschen Umlaute etwa unterschiedlich kodiert, so daß Quellcodes ohne gewissen Zusatzaufwand nicht mehr **portabel** sind. Für den übersetzten Bytecode — also die `class`-Dateien, die durch den Compiler `javac` erzeugt werden — ergeben sich diese Probleme nicht. Wir wollen aber auch in der Lage sein, unseren Programmtext untereinander auszutauschen.
- dem **Unterstrich** „_“
- dem **Dollarzeichen** „$“
- den **Ziffern** `0,1,2,...,9`

Bezeichner beginnen hierbei immer mit einem Buchstaben, dem Unterstrich oder dem Dollarzeichen – niemals jedoch mit einer Ziffer. Des weiteren dürfen sie kein reserviertes Wort sein, d.h. sie dürfen nicht so lauten wie einer der „Vokabeln“, welche wir in den folgenden Abschnitten lernen werden.

Folgende Beispiele zeigen gültige Bezeichner in Java:

- `Hallo_Welt`
- `_H_A_L_L_O_`
- `hallo123`

- `hallo_123`

Folgende Beispiele würden in Java jedoch zu einer Fehlermeldung führen:

- `10kleineNegerlein` – Bezeichner dürfen nicht mit Ziffern beginnen.
- `Das_war's` – das Zeichen `'` ist in Bezeichnern nicht erlaubt.
- `Hallo Welt` – Bezeichner dürfen keine Leerzeichen enthalten.
- `class` – dies ist ein reserviertes Wort.

2.1.3 Literale

Als **Literal** wird in Java allgemein die Darstellung einer der folgenden Werte in unseren Programmen bezeichnet:

- ganze Zahlen (etwa `23` oder `-166`). Hierbei zählt das Vorzeichen genaugenommen nicht zum Literal; die Negation ist vielmehr eine nachträglich durchgeführte mathematische Operation.
- Gleitkommazahlen (z.B. `3.14`)
- Wahrheitswerte (`true` und `false`)
- einzelne Zeichen in einfachen Hochkommata, so etwa `'a'`
- Zeichenketten in Anführungszeichen, etwa (`"Hallo Welt"`) und
- das sogenannten **Null-Literal**, dargestellt durch das Wortsymbol `null`.

Wir werden auf die einzelnen Punkte später genauer eingehen; momentan wollen wir uns nur merken, daß es die sogenannten Literale gibt und sie Teil eines Java-Programms sein können (und werden).

2.1.4 Wortsymbole

Wie bereits erwähnt gibt es gewisse Worte, die wir in Java nicht als Bezeichner verwenden dürfen. Diese Worte symbolisieren Befehle, die wir später kennenlernen werden. Hier eine Liste dieser Wortsymbole, die Bestandteil vieler Programme sind:

abstract	boolean	break	byte
byvalue	case	cast	catch
char	class	const	continue
default	do	double	else
extends	false	final	finally
float	for	future	generic
goto	if	implements	import
inner	instanceof	int	interface
long	native	new	null
operator	outer	package	private
protected	public	rest	return
short	static	super	switch
synchronized	this	throw	throws
transient	true	try	var
void	volatile	while	

Tabelle 1: reservierte Wortsymbole

2.1.5 Trennzeichen

Was sind *Blumento-Pferde*? Heißt es der, die oder das *Kuhliefumdenteich*? Die alten Scherzfragen aus der Vorschulzeit basieren meist auf einem Grundprinzip der Sprachen: hat man mehrere Worte, so muß man diese durch Pausen entsprechend voneinander trennen – sonst versteht keiner ihren Sinn! Schreibt man die entsprechenden Worte nieder, werden aus den Pausen Leerzeichen, Gedankenstriche und Kommata.

Auch in Java müssen wir in der Lage sein, einzelne Bezeichner und Wortsymbole voneinander zu trennen. Hierzu stehen uns mehrere Möglichkeiten zur Verfügung, die sogenannten Trennsymbole. Diese sind

- Leerzeichen
- Zeilenendezeichen (der Druck auf die `ENTER`-Taste)
- Tabulatorzeichen (die `TAB`-Taste)
- Kommentare

- die Zeichen `)`, `(`, {, }, `[`, `]` , `;`, und `,` — diese haben in der Sprache jedoch eine besondere Bedeutung. Man sollte sie deshalb nur dort einsetzen, wo sie auch hingehören.

Einzelne Wortsymbole, Literale und Bezeichner müssen durch mindestens eines der obigen Symbole voneinander getrennt werden, sofern deren Anfang und Ende nicht aus dem Kontext erkannt werden kann. Hierbei ist es Java eigentlich egal, welche Trennzeichen man verwendet (und wieviele von ihnen). Steht im Programm zwischen zwei Bezeichnern beispielsweise eine Klammer, so gilt diese bereits als Trennsymbol. Wir müssen keine weiteren Leerzeichen einfügen (können aber). Die Programmzeile

```
public static void main(String[] args)
```

wäre demnach völlig äquivalent zu folgenden Zeilen:

```
public // verwendete Trennsymbole: Leerzeichen und Kommentar
  static
/*Verwendung eines Zeilenendezeichens*/
                void
             //..
            // ..
// man kann auch mehrere Zeilenvorschuebe verwenden
  main(// hier sind nun zwei Trennzeichen: Klammer und Kommentar!
               String[] args)
```

Übersichtlicher wird der Text hierdurch jedoch nicht. Wir werden uns deshalb später auf einige Konventionen einigen, um unsere Programme lesbarer zu machen.

Die Frage, an welcher Stelle Trennsymbole wirklich gebraucht werden, läßt sich jedoch nicht allein am Auftauchen von Wortsymbolen, Literalen und Bezeichnern ausmachen. Der Ausdruck `23+24` etwa addiert in Java die Literale 23 und 24. Obwohl der Ausdruck kein Trennsymbol beinhaltet, kann Java die beiden Zahlen auseinanderhalten — sie werden durch das „+“-Zeichen getrennt, welches als eigenständiges Symbol erkannt wird.

2.1.6 Interpunktionszeichen

Was der deutschen Sprache der Punkt ist Java das Semikolon. Befehle (sozusagen die Sätze der Sprache) werden in Java immer mit einem Semikolon abgeschlossen. Fehlt dieses, liefert der Übersetzer eine Fehlemeldung der Form

```
Fehlerhaft.java:10: ';' expected.
```


Hierbei ist `Fehlerhaft.java` der Dateiname, unter dem das Programm gespeichert wurde. Die angegebene Ziffer steht für die Nummer der Zeile, in der der Fehler aufgetreten ist.

Neben dem Semikolon existieren in Java noch weitere Interpunktionszeichen. Werden mehrere Befehle zu einem **Block** zusammengefaßt, so geschieht dies, indem man vor den ersten und nach den letzten der Befehle eine geschweifte Klammer setzt. Hierzu ein Beispiel:

```
{                          //Blockbeginn
  System.out.println("B1"); // Befehl Nr. 1
  System.out.println("B2"); // Befehl Nr. 2
  System.out.println("B3"); // Befehl Nr. 3
}                          // Blockende
```

Es existieren noch weitere Interpunktionszeichen in der Sprache Java; diese werden von uns jedoch erst bei Bedarf eingeführt.

2.1.7 Operatorsymbole

Operatoren sind spezielle Symbole, die Java dazu veranlassen, bis zu drei unterschiedliche Werte – die sogenannten Operanden – miteinander zu verknüpfen. Wir unterscheiden die Operatoren nach ihrer Anzahl von Operanden:

monadische Operatoren sind Operatoren, die nur einen Operanden benötigen. Beispiele hierfür sind die Operatoren `++` oder `--`.

dyadische Operatoren haben zwei Operanden und sind die am häufigsten vorkommende Art von Operatoren. Beispiele hierfür sind etwa die Operatoren `+`, `-` oder `==`.

triadische Operatoren sind in Java äußerst selten. Einzigstes Beispiel ist der Operator `?:` (Fragezeichen-Doppelpunkt), welcher über drei Operanden verfügt.

Operatoren sind in Java mit sogenannten Prioritäten versehen, die es der Sprache ermöglichen, gewisse Reihenfolgen in der Auswertung (zum Beispiel Punkt- vor Strichrechnung) einzuhalten. Wir werden uns mit diesem Thema an anderer Stelle befassen.

2.1.8 import-Anweisungen

Viele Dinge, die wir in Java benötigen, befinden sich nicht im Kern der Sprache — beispielsweise die Bildschirmausgabe oder mathematische Standardfunktionen wie Sinus oder Cosinus. Sie wurden in sogenannte Klassen ausgelagert, welche vom Übersetzer erst bei Bedarf hinzugeladen werden. Um den Übersetzer anzuweisen, einen solchen Vorgang einzuleiten, wird eine sogenannte `import`-Anweisung verwendet. Diese macht dem Compiler die von der Anweisung bezeichnete Klasse zugänglich, d.h. er kann auf sie zugreifen und sie verwenden. Ein Beispiel hierfür sind etwa die `IOTools`, mit deren Hilfe Sie später Eingaben von der Tastatur bewerkstelligen werden. Die Klasse gehört zu einem Packet namens `Prog1Tools`, welches nicht von der Firma Sun stammt und somit nicht zu den standardmäßig eingebundenen Werkzeugen gehört. Wenn Sie die Klasse verwenden wollen, beginnen Sie Ihr Programm mit

```
import Prog1Tools.IOTools;
```

Viele Klassen, die wir vor allem zu Anfang benötigen, werden vom System automatisch als bekannt vorausgesetzt – es wird also noch eine Weile dauern, bis in unseren Programmen die `import`-Anweisung zum ersten Male auftaucht. Sie ist dennoch Grundelement eines Java-Programmes und soll an dieser Stelle deshalb erwähnt werden.

2.1.9 Zusammenfassung 2.1

Wir haben in diesem Abschnitt die verschiedenen Komponenten kennengelernt, aus denen sich ein Java-Programm zusammensetzt. Wir haben gelernt, daß es aus

- Kommentaren
- Bezeichnern
- Trennzeichen
- Wortsymbolen
- Interpunktionszeichen
- Operatoren und
- `import`-Anweisungen

bestehen kann, auch wenn nicht jede dieser Komponenten in jedem Java-Programm auftaucht. Wir haben gelernt, daß es wichtig ist, seine Programme gründlich zu dokumentieren und haben uns auf einige einfache Konventionen festgelegt, mit denen wir einen ersten Schritt in diese Richtung tun wollen. Mit diesem Wissen können wir beginnen, erste Java-Programme zu schreiben.

2.1.10 Übungsaufgaben 2.1

1. Die Zeichenfolge `dummy` hat in Java keine vordefinierte Bedeutung – sie wird also, wenn sie ohne besondere Vereinbarungen im Programmtext steht, zu einem Compilerfehler führen. Welche der folgenden Zeilen könnte also einen solchen Fehler verursachen?

```
      dummy
      dummy;
      dummy; //
//    dummy
//    dummy;
/*    dummy */
*/    dummy /*
/**/ dummy
      dummy /* */
```

2. Welche der folgenden Begriffe sind auf keinen Fall Bezeichner?

```
Karl der Grosse
Karl_der_Grosse
Karl,der_Grosse
0 Ahnung?
0_Ahnung
null_Ahnung!
1234abc
_1234abc
_1_2_3_4_abc
```

2.2 Erste Schritte in Java

Nachdem wir nun über die Grundelemente eines Java-Programmes Bescheid wissen, können wir damit beginnen, unsere ersten kleineren Programme in Java zu schreiben. Wir werden mit einigen einfachen Problemstellungen beginnen und uns langsam an etwas anspruchsvollere Aufgaben herantasten.

Es gibt in Java zwei verschiedene Wege, sich dem Computer mitzuteilen:

- Man schreibt ein Programm, eine sogenannte **Applikation**, welche dem Computer in einer Aneinanderreihung von diversen Befehlen angibt, was er genau zu tun hat. Sie werden mit dem Java-Interpreter von Sun gestartet.

  Applikationen können graphische und interaktive Elemente beinhalten, müssen dies aber nicht. Sie sind die wohl einfachste Form, in Java zu programmieren.

- Man verfaßt ein sogenanntes **Applet**, welches in eine Internetseite eingebunden wird und mit jedem aktuellen Internetbrowser gestartet werden kann. Applets sind grundsätzlich graphisch aufgebaut und enthalte zumeist eine Menge an Interaktion – das heißt sie können auf Mausklick und Tastendruck reagieren, geöffnet und geschlossen werden ...

Applets sind speziell für das Internet geschaffen und wohl der ausschlaggebende Faktor, daß sich die Sprache heute einer so großen Beliebtheit erfreut. Für einen Anfänger sind sie jedoch (noch) zu kompliziert. Wir werden deshalb mit der Programmierung von Applikationen beginnen.

2.2.1 Grundstruktur eines Java-Programms

Angenommen, wir wollen ein Programm schreiben, welches wir `Hallo Welt` nennen wollen. Als erstes müssen wir uns darüber klarwerden, daß dieser Name kein Bezeichner ist — Leerzeichen sind nicht erlaubt. Wir ersetzen deshalb die Leerzeichen durch den Unterstrich und erstellen mit unserem Editor eine Datei mit dem Namen `Hallo_Welt.java` .

Hinweis: Es gibt Fälle, in denen die Datei nicht wie das Programm heißen muß. Im allgemeinen erwartet dies der Übersetzer jedoch; wir wollen es uns deshalb von Anfang an angewöhnen.

Geben wir nun in unseren Editor folgende Zeilen ein (die Kommentare können auch wegfallen):

```
public class Hallo_Welt {                    // Klassen- bzw. Programmbeginn
    public static void main(String[] args) {  // Beginn des Hauptprogramms
        // HIER STEHT EINMAL DAS PROGRAMM...
    }                                        // Ende des Hauptprogramms
}                                            // Ende des Programms
```

Wir sehen, daß das Programm aus zwei Ebenen besteht, die wir an dieser Stelle kurz erklären wollen:

- Mit der Zeile

  ```
  public class Hallo_Welt {
  ```

  machen wir dem Übersetzer klar, daß die folgende Klasse[7] den Namen `Hallo_Welt` trägt. Der Übersetzer speichert das kompilierte Programm nun in einer Datei namens `Hallo_Welt.class` ab. Diese Zeile darf niemals fehlen, denn wir müssen unserem Programm natürlich einen Namen zuweisen.

- Es ist prinzipiell möglich, ein Programm in mehrere Abschnitte zu unterteilen. Die sogenannte **Hauptroutine** (oder auch Hauptprogramm genannte) wird mit der Zeile

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

  eingeleitet. Der Übersetzer erfährt somit, an welcher Stelle er die Ausführung des Programmes zu beginnen hat.

Es fällt auf, daß beide Zeilen jeweils mit einer geschweiften Klammer enden. Wir erinnern uns – dieses Interpunktionszeichen steht für den Beginn eines Blocks, d.h. es werden mehrer Zeilen zu einer Einheit zusammengefaßt. Diese Zeichen entbehren nicht einer gewissen Logik. Der erste Block faßt die folgenden Definitionen zu einem Block zusammen; er weist den Compiler an, sie als eine Einheit (eben das Programm bzw. die Klasse) zu betrachten. Der zweite Block umgibt die Anweisungen des Hauptprogramms und faßt sie eben zu diesem zusammen.

Achtung: Jede geöffnete Klammer (Blockanfang) muß sich irgendwann auch wieder schließen (Blockende). Wenn wir dies vergessen, wird das mit einem Übersetzungsfehler bestraft!

2.2.2 Ausgaben auf der Konsole

Wir wollen nun unser Programm so erweitern, daß es die Worte `Hallo Welt` auf dem Bildschirm ausgibt. Wir ersetzen hierzu in unserem Editor die Zeile

```
// HIER STEHT EINMAL DAS PROGRAMM...
```

durch die Zeile

[7] ein Begriff aus dem objektorientierten Programmieren; wir wollen ihn im Moment mit Programm gleichsetzen

```
System.out.println("Hallo Welt");
```

Wir sehen, daß sich diese Zeile aus mehreren Komponenten zusammensetzt

- Der Anweisung `System.out.println()`, welche den Computer anweist, einen Text auf dem Bildschirm auszugeben. Der auszugebende Text muß zwischen den runden Klammern stehen.
- dem Text `"Hallo Welt"`, der auf dem Bildschirm ausgegegben werden soll. Die Anführungszeichen tauchen beim Ausdruck übrigens nicht auf; sie markieren nur den Anfang und das Ende des Textes.
- dem Interpunktionszeichen „;“, welches jeden Befehl beeden muß. Ein vergessenes Semikolon ist wohl der häufigste Programmierfehler und unterläuft selbst alten Hasen hin und wieder ...

Wir speichern unser so verändertes Programm ab und geben in der Kommandozeile die Anweisung:

```
javac Hallo_Welt.java
```

ein. Der Compiler übersetzt unser Programm nun in eine für den Computer verständliche Form. Dies finden wir unter dem Namen `Hallo_Welt.class` wieder.

Wir wollen unser Programm nun starten. Hierzu geben wir in der Kommandozeile ein:

```
java Hallo_Welt
```

Unser Programm wird tatsächlich ausgeführt – die Worte `Hallo Welt` erscheinen auf dem Bildschirm. Ermutigt von diesem ersten Erfolg, erweitern wir unser Programm um die Zeile

```
System.out.println("Mein erstes Programm :-)");
```

und übersetzen erneut. Die neue Ausgabe auf dem Bildschirm lautet nun

```
Hallo Welt
Mein erstes Programm :-)
```


Wir sehen, daß der Befehl `System.out.println()` nach dem ausgegebenen Text einen Zeilenvorschub macht. Was ist jedoch, wenn wir dies nicht wollen?

Die einfachste Möglichkeit ist, beide Texte in einer Anweisung zu drucken. Die neue Zeile würde dann entweder

```
System.out.println("Hallo Welt Mein erstes Programm :-)");
```

oder

```
System.out.println("Hallo Welt " + "Mein erstes Programm :-)");
```

heißen. Letztere Zeile enthält einen für uns neuen Operator. Das Zeichen `+` addiert nicht etwa zwei Texte (wie sollte dies auch funktionieren). Es weist Java vielmehr an, die Texte `Hallo Welt` und `Mein erstes Programm` aneinanderzuhängen. Wir werden diesen Operator später noch zu schätzen wissen.

Eine weitere Möglichkeit, dasselbe Ergebnis zu erzielen, ist die Verwendung des Befehls `System.out.print`. Im Gegensatz zu `System.out.println` springt dieses Kommando nach der Ausführung nicht in die nächste Bildschirmzeile. Weitere Zeichen werden noch in dieselbe Zeile geschrieben. Unser so verändertes Programm sähe wie folgt aus:

```
public class Hallo_Welt {                    // Klassen- bzw. Programmbeginn
    public static void main(String[] args) {  // Beginn des Hauptprogramms
        System.out.print("Hallo Welt ");
        System.out.println("Mein erstes Programm :-)");
    }                                        // Ende des Hauptprogramms
}                                            // Ende des Programms
```

2.2.3 Schöner Programmieren in Java

Wir haben bereits gesehen, daß Programme sehr strukturiert und übersichtlich gestaltet werden können – oder unstrukturiert und chaotisch. Wir wollen uns deshalb einige goldene Regeln angewöhnen, mit denen wir unser Programm auf einen übersichtlichen und lesbaren Stand bringen.

1. *Niemals mehr als einen Befehl in eine Zeile schreiben!* Auf diese Art und Weise können wir beim späteren Lesen des Codes auch keine Anweisung übersehen.

2. Wenn *wir* einen neuen Block beginnen, *rücken* wir alle in diesem Block stehenden Zeilen um einige (beispielsweise zwei) Zeichen *nach rechts ein*. Wir haben auf diese Art und Weise stets den Überblick darüber, wie lange ein Block eigentlich geht.

3. *Das Blockendezeichen „}“ wird stets so eingerückt*, daß es mit der Zeile übereinstimmt, in der der Block geöffnet wurde. Wir können hierdurch vergessene Klammern sehr viel schneller aufspüren.

Wir wollen dies an einem Beispiel verdeutlichen. Folgendes Programm entspricht *nicht* unseren Regeln:

```
public class Unsorted {public static void main(String[] args) {
System.out.print("Ist dieses");System.out.print(" Programm eigentlich");
System.out.println(" noch "+"lesbar?");}}
```

Obwohl es nur aus wenigen Zeilen besteht, ist das Lesen dieses kurzen Programms doch schon recht schwierig. Wie sollen wir mit solchen Texten zurechtkommen, die sich aus einigen *hundert* Zeilen Programmcode zusammensetzen? Wir spalten das Programm deshalb gemäß unserer Regeln auf und rücken entsprechend ein:

```
public class Unsorted {
    public static void main(String[] args) {
        System.out.print("Ist dieses");
        System.out.print(" Programm eigentlich");
        System.out.println(" noch " + "lesbar?");
    }
}
```

Übersetzen wir das Programm und starten es, so können wir die auf dem Bildschirm erscheinende Frage eindeutig bejahen.

2.2.4 Zusammenfassung 2.2

Wir haben Applets und Applikationen eingeführt und festgelegt, daß wir mit der Programmierung von letzteren beginnen wollen. Wir haben die Grundstruktur einer Java-Applikation kennengelernt und erfahren, wie man mit den Befehlen `System.out.println` und `System.out.print` Texte auf dem Bildschirm ausgibt. Hierbei wurde auch der Operator + erwähnt, der Texte aneinanderfügen kann. Dieses Wissen haben wir angewendet, um unser erstes Java-Programm zu schreiben.

Zu guter Letzt haben wir uns auf einige Regeln geeinigt, nach denen wir unsere Programme formatieren wollen. Wir haben gesehen, wie die eingache Anwendung dieser „Gesetze“ unseren Quelltext viel lesbarer machen.

2.2.5 Übungsaufgaben 2.2

1. Schreiben Sie ein Java-Programm, welches Ihren Namen dreimal hintereinander auf dem Bildschirm ausgibt.

2. Gegeben ist folgende Java-Programm:

```
                public class Strukturuebeung
  { public static void main (String[] args){
System.out.println("Was mag ich wohl tun?")}
```

   Dieses Programm enthält zwei Fehler, welche es zu finden gilt. Versuchen Sie zuerst, diese anhand des vorliegenden Textes zu finden. Wenn Ihnen dies nicht gelingt, formatieren Sie das Programm gemäß unserer „goldenen Regeln.“ Versuchen Sie auch einmal, das fehlerhafte Programm zu übersetzen. Machen Sie sich mit den auftretenden Fehlermeldungen vertraut.

2.3 Einfache Datentypen

Natürlich reicht es uns nicht aus, einfache Meldungen auf dem Bildschirm auszugeben (dafür bräuchten wir keine Programmiersprache). Wir wollen Java benutzen können, um gewisse Effekte zu erzielen, Rechnungen durchzuführen und Probleme zu lösen – mit einem Wort zur Datenverarbeitung. Wir wollen uns aus diesem Grund darüber klarwerden, wie man in Java mit einfachen Daten (Zahlen, Buchstaben usw.) umzugehen hat.

2.3.1 Ganzzahlige Datentypen

Wie geht ein Computer mit ganzen Zahlen um? Er speichert sie als eine Folge von binären Zeichen (also 0 oder 1), welche ganzzahlige Potenzen der Zahl 2 repräsentieren. Die Zahl 23 kann etwa durch die Folge

```
1*16 + 0*8 + 1*4 + 1*2 + 1*1
```

ausgedrückt werden und wird somit durch die Binärfolge `10111` kodiert. Negative Zahlen können durch verschiedene Methoden kodiert werden; in jedem Fall benötigt man jedoch eine weitere Stelle für das Vorzeichen.

Nehmen wir an, wir haben 1 Byte – dies sind 8 Bit, also 8 binäre Stellen – zur Verfügung, um eine Zahl darzustellen. Das erste Bit benötigen wir für das Vorzeichen. Die größte Zahl, die wir mit den noch verbleibenden 7 Ziffern darstellen können, ist somit

```
1*64 + 1*32 + 1*16 + 1*8 + 1*4 + 1*2 + 1*1 = 127.
```

Aufgrund des sogenannten Zweierkomplements, in dem wir negative Zahlen darstellen, ist die kleinste negative Zahl betragsmäßig um 1 größer., d.h. in diesem Fall -128. Wir können mit 8 Bits also 127-128+1=256 verschiedene Zahlen darstellen. Hätten wir mehr Stellen zur Verfügung, würde auch unser Zahlenvorrat wachsen.

Der Datentyp `byte` repräsentiert in Java genau diese Darstellung. Eine Zahl vom Typ `byte` ist 8 Bit lang und liegt im Bereich von -128 bis +127. Im allgemeinen ist dieser Zahlenbereich viel zu klein, um damit vernünftig arbeiten zu können. Java besitzt aus diesem Grund weitere Arten von ganzzahligen Typen, die zwei, vier oder acht Byte lang sind. Die Tabelle faßt diese Datentypen zusammen.

Typname	größter Wert	kleinster Wert	Länge
`byte`	127	-128	8 Bits
`short`	32767	-32768	16 Bits
`int`	2147483647	-2147483648	32 Bits
`long`	9223372036854775807	-9223372036854775808	64 Bits

Tabelle 2: ganzzahlige Datentypen

Moderne Rechner sind meistens mit sogenannten 32Bit-Prozessoren ausgestattet, d.h. mit Rechenwerken, die mit bis zu 32 Bit auf einmal ohne erhöhten Zeitaufwand arbeiten können. Java trägt dieser Entwicklung Rechnung – wenn wir in unseren Programmen Zahlen verwenden, wird standardmäßig angenommen, daß es sich hierbei um höchstens eine 32-Bit-Zahl handelt. Wollen wir stattdessen mit 64 Bit arbeiten (also mit Zahlen im `long`-Format, so müssen wir an die Zahl die Endung `L` anhängen.

Wir wollen dies an einem Beispiel verdeutlichen. Die Befehle
`System.out.println` und
`System.out.print`
sind auch in der Lage, einfache Datentypen wie die Ganzzahlen auszu-

drucken. Wir versuchen nun, die Zahl `9223372036854775807` auf dem Bildschirm auszugeben. Hierzu schreiben wir folgendes Programm:

```
public class Longtst {
    public static void main (String[] args) {
        System.out.println(9223372036854775807);
    }
}
```

Rufen wir den Compiler mit dem Kommando `javac Longtst.java` auf, so erhalten wir folgende Fehlermeldung;

```
Longtst.java:3: Numeric overflow.
        System.out.println(9223372036854775807);
```

Was will uns der Compiler damit sagen? In Zeile 3 der Datei `Longtst.java` ist ein sogenannter Überlauf (engl. **overflow**) aufgetreten. Den Grund hierfür entnehmen wir der Tabelle. Die Zahl `9223372036854775807` ist deutlich größer als `2147483647` und kann somit nicht mehr mit 32 Bit dargestellt werden. Java verlangt jedoch, daß man derartige Fälle explizit angibt! Wir ändern die Zeile deshalb wie folgt:

```
System.out.println(92233720368547758O7L);
```

Durch die Hinzunahme der Endung `L` wird die Zahl als eine `long`-Zahl betrachtet und somit mit 64 Bit kodiert (in welche sie laut Tabelle gerade noch hineinpaßt).

2.3.2 Gleitkommatypen

Wir wollen eine einfache Rechung durchführen. Das folgende Programm soll das Ergebnis von `1/10` ausgeben. Hierzu bedienen wir uns des Divisionsoperators in Java:

```
public class Intdiv {
    public static void main (String[] args) {
        System.out.println(1/10);
    }
}
```

Wir übersetzen das Programm und führen es aus. Zu unserer Überraschung erhalten wir jedoch ein vermeintlich falsches Ergebnis – und zwar die Null! Was ist geschehen?

Um zu begreifen, was eigentlich passiert ist, müssen wir uns eines klarmachen: wir haben mit ganzzahligen Datentypen gearbeitet. Der Divisionsoperator ist in Java jedoch so definiert, daß die Division zweier ganzen Zahlen wiederum eine ganze Zahl ergibt. Wir erinnern uns an die Grundschulzeit – hier hätte `1/10` ebenfalls 0 ergeben – mit sogenanntem Rest 1. Den Rest können wir in Java mit dem %-Zeichen bestimmen. Wir ändern unser Programm entsprechend:

```
public class Intdiv {
    public static void main (String[] args) {
        System.out.print("1/10 betraegt ");
        System.out.print(1/10);
        System.out.print(" mit Rest ");
        System.out.print(1%10);
        System.out.println(" .");
    }
}
```

Das neue Programm gibt als Meldung

```
1/10 betraegt 0 mit Rest 1 .
```

aus.

Nun ist es im allgemeinen nicht wünschenswert, nur mit ganzen Zahlen zu arbeiten. Angenommen wir wollen etwa einen Geldbetrag in eine andere Währung umrechnen. Sollen die Pfennigbeträge dann etwa wegfallen? Java bietet aus diesem Grund auch die Mölichkeit, mit sogenannten **Gleitkommazahlen** (engl.: **floating point number**) zu arbeiten. Diese sind aus 32 bzw. 64 Bit aufgebaut und besitzen folgenden Zahlenumfang:

Typname	größter positiver Wert	kleinster positiver Wert	Länge
`float`	3.4028235E38	1.4E-45	32 Bits
`double`	1.7976931348623157E308	4.9E-324	64 Bits

Tabelle 3: Gleitkommatypen

Hierbei steht das E mit anschließender Zahl `X` für die Multiplikation mit 10^X, d.h. die Zahl `1.2E3` wäre in Wirklichkeit `1.2*1000=1200`. Die Zahl `1.2E-3` wäre hingegen `1.2*0.001=0.0012`. Negative Zahlen werden erzeugt, indem man vor die entsprechende Zahl ein Minuszeichen setzt.

Natürlich kann mit 32 oder auch 64 Bits nicht jede Zahl zwischen `+3.4028235E38` und `-3.4028235E38` exakt dargestellt werden. Der Computer arbeitet wieder mit Potenzen von 2, so daß selbst so einfache Zahlen wie `0.1` im Rechner nicht exakt kodiert werden können. Es wird deshalb intern eine Rundung durchgeführt, so daß wir in einigen Fällen mit Rundungsfehlern rechnen müssen. Uns soll dies für den Moment jedoch egal sein. Wir werden unser Programm nun vielmehr auf das Rechnen mit Gleitkommazahlen umstellen. Hierzu ersetzen wir lediglich die ganzzahligen Werte durch Gleitkommazahlen:

```
public class Floatdiv {
    public static void main (String[] args) {
        System.out.print("1/10 betraegt ");
        System.out.print(1.0/10.0);
        System.out.println(" .");
    }
}
```

Lassen wir das neue Programm laufen, so erhalten wir als Ergebnis:

```
1/10 betraegt 0.1 .
```

2.3.3 Sonstige einfache Datentypen

Manchmal erweist es sich als notwendig, mit einzelnen Buchstaben oder Zeichen zu arbeiten. Diese Zeichen werden in Java durch den Datentyp `char` (Abkürzung für *Character*) definiert. Ein einzelnes Zeichen wird in einfachen Hochkommata dargestellt, d.h. das Zeichen `a` hätte in Java die Darstellung `'a'`. Daten vom Typ `char` haben eine Länge von 16 Bits. Jedes Zeichen hat intern übrigens eine gewissen Numerierung (den sogenannten **Unicode**). Der Buchstabe `a` besitzt beispielsweise die Nummer `97`. Man kann Werte vom Type `char` demnach auch als ganzzahlige Werte auffassen und entsprechend miteinander verknüpfen.

A propos verknüpfen – es wird immer wieder notwendig sein, zwei Werte miteinander zu vergleichen. Ist etwa der Wert `23` größer, kleiner oder gleich einem anderen Wert? Die hierzu gegebenen Vergleichsoperatoren liefern eine Antwort, die letztendlich auf ja oder nein, wahr oder falsch, `true` oder `false` hinausläuft. Da wir diese Wahrheitswerte über eine Aussage nun einmal haben, müssen wir natürlich auch mit ihnen arbeiten können. Hierzu existiert in Java der Datentyp `boolean`. Dieser Type besitzt lediglich zwei mögliche Belegungen: `true` und `false`. Mit ihm können logische Aussagen ideal ausgewertet werden.

2.3.4 Typenumwandlungen: implizite und explizite Typkonvertierung

Manchmal kommt es vor, daß wir an einer Stelle einen gewissen Datentyp benötigen, jedoch einen anderen vorliegen haben. Wir wollen etwa die Addition

```
9223372036854775001+807
```

einer 64-Bit-Zahl und einer 32-Bit-Zahl durchführen. Der Plus-Operator ist jedoch nur für Werte des gleichen Typs definiert. Was also tun?

In unserem Fall ist dies einfach – naemlich nichts. Java erkennt, daß die linke Zahl einen Zahlenbereich hat, der den der rechten umfaßt. Das System kann also die `807` problemlos in eine `long`-Zahl umwandeln (was es auch tut). Diese Umwandlung, welche von uns unbemerkt im Hintergrund geschieht, wird als **implizite Typkonvertierung** (engl. **implicite typecast**) bezeichnet.

Implizite Typkonvertierungen treten immer dann auf, wenn ein kleinerer Zahlenbereich in einen größeren Zahlenbereich abgebildet wird, d.h. von `byte` nach `short`, von `short` nach `int`, von `int` nach `long` und so weiter. Ganzzahlige Datentypen können auch implizit in Gleitkommatypen umgwandelt werden, obwohl hierbei eventuell Rundungsfehler auftreten können. Eine implizite Umwandlung von `char` nach `int` ist ebenfalls möglich.

Manchmal kommt es jedoch auch vor, daß wir einen größeren in einen kleineren Zahlenbereich umwandeln müssen. Wir haben beispielsweise das Ergebnis einer Gleitkommarechnung (sagen wir `3.14`) und interessieren uns nur für den Anteil vor dem Komma. Wir wollen also etwa eine `double`-Zahl in `int` verwandeln.

Bei einer solchen Typenumwandlung gehen eventuell Informationen verloren – der Compiler wird dies nicht ohne weiteres tun. Er gibt uns beim Übersetzen eher eine Fehlermeldung der Form

```
Incompatible type for declaration.
Explicit cast needed to convert double to int.
```

aus. Der Grund hierfür liegt darin, daß derartige Umwandlungen häufig auf Programmierfehlern beruhen, also eigentlich überhaupt nicht beabsichtigt sind. Der Compiler geht davon aus, daß ein Fehler vorliegt und meldet dies auch.

Wir müssen dem Übersetzer also klarmachen, daß wir genau wissen, was wir tun! Dieses Verfahren wird als **explizite Typkonvertierung** (engl.

explicite typecast) bezeichnet und wird durchgeführt, indem wir den beabsichtigten Zieldatentyp in runden Klammern vor die entsprechende Zahl schreiben. In unserem obigen Beispiel würde dies etwa

```
(int) 3.14
```

bedeuten. Der Compiler erkennt, daß die Umwandlung wirklich gewollt ist, und schneidet die Nachkommastellen ab. Das Ergebnis der Umwandlung beträgt `3`.

Eine Umwandlung von `boolean` in einen anderen Datentyp ist übrigens nicht möglich – weder explizit, noch implizit.

2.3.5 Zusammenfassung 2.3

Wir haben einfache Datentypen kennengelernt, mit denen wir ganze Zahlen, Gleitkommazahlen, einzelne Zeichen und Wahrheitswerte in Java darstellen können. Wir haben erfahren, in welcher Beziehung diese Datentypen zueinander stehen und wie man sie ineinander umwandeln kann.

2.3.6 Übungsaufgaben 2.3

1. Welcher der folgenden expliziten Typkonvertierungen ist unnötig, da er im Bedarfsfalle implizit durchgeführt würde?

   - `(int) 3`
   - `(long) 3`
   - `(long) 3.1`
   - `(short) 3`
   - `(short) 3l`
   - `(double) 3l`
   - `(int) 'x'`
   - `(double)'x'`

2.4 Der Umgang mit einfachen Datentypen

Wir haben nun einfache Datentypen kennengelernt, mit denen wir jetzt etwas arbeiten wollen. Wir werden hierzu in diesem Kapitel lernen, wie man in Java Werte speichert und sie durch Operatoren miteinander verknüpft.

2.4.1 Variablen

Bis jetzt waren unsere Beispielprogramme alle recht eintönig. Das lag vor allem daran, daß wir bislang keine Möglichkeit hatten, Werte zu speichern, um dann später wieder auf sie zugreifen zu können. Genau das kann man mit Variablen erreichen.

Am besten stellt man sich eine Variable wie ein Postfach vor: Ein Postfach ist im Prinzip nichts anderes als ein Behälter, der mit einem eindeutigen Schlüssel — in der Regel die Postfachnummer — gekennzeichnet ist und in den wir etwas hineinlegen können. Später können wir dann über den eindeutigen Schlüssel — die Postfachnummer — das Postfach wieder auffinden und auf den dort abgelegten Inhalt zugreifen.

Genauso verhält es sich mit Variablen: Über einen eindeutigen Schlüssel, in diesem Fall dem Variablennamen, können wir auf Variablen zugreifen. Man kann Variablen einen bestimmten Inhalt zuweisen, und diesen später auch wieder auslesen. Das ist im Prinzip schon alles was wir brauchen, um unsere Programme etwas interessanter zu gestalten.

Dazu erstmal ein Beispiel. Angenommen, wir wollten in einem Programm ausrechnen, wieviel Geld wir in unserem Aushilfsjob als ... (je nach eigener Präferenz bitte ausfüllen) letzte Woche verdient haben. Unser Stundenlohn betrage in diesem Job 15 DM, und letzte Woche haben wir es auf 18 Stunden gebracht. Dann könnten wir mit unserem bisherigen Wissen dazu ein Programm folgender Art basteln:

```
public class StundenRechner1 {
  public static void main(String[] args) {

    System.out.print("Arbeitsstunden: ");
    System.out.println(18);
    System.out.print("Stundenlohn in DM: ");
    System.out.println(15);
    System.out.print("Damit habe ich letzte Woche soviel (in DM) verdient: ");
    System.out.println(18 * 15);

  }
}
```

Das Programm läßt sich auch anstandslos kompilieren erzeugt auch folgende (korrekte) Ausgabe:

```
Arbeitsstunden: 18
Stundenlohn in DM: 15
Damit habe ich letzte Woche soviel (in DM) verdient: 270
```


So weit, so gut. Was passiert aber nun in der darauffolgenden Woche, in der wir es nur auf 12 Stunden Arbeitszeit bringen? Eine Möglichkeit wäre, einfach die Zahl der Arbeitsstunden im Programm zu ändern – allerdings müßten wir das jetzt an zwei Stellen tun, nämlich in Zeilen 5 und 9 (Leerzeilen werden mitgezählt) jeweils den Wert 18 auf 12 ändern. Und dabei kann es nach drei (vier, fünf, ...) Wochen leicht passieren, daß wir vergessen, eine der beiden Zeilen zu ändern oder daß wir uns in einer der Zeilen vertippen. Besser wäre es also, wenn wir die Anzahl der geleisteten Arbeitsstunden nur an einer einzigen Stelle im Programm ändern müßten. Und genau hier kommen nun Variablen ins Spiel.

Um Variablen in unserem Programm verwenden zu können, müssen wir dem Java Compiler zunächst mitteilen, welche Art von Werten wir in ihr speichern wollen und wie die Variable heißen soll. Diese Anweisung bezeichnen wir auch als **Deklaration**. Um in der Analogie der Postfächer zu bleiben: Wir müssen das Postfach mit einer bestimmten Größe (Briefach, Packetfach, ...) erst einmal einrichten und es mit einer eindeutigen Postfachnummer versehen.

Eine solche Deklaration hat stets folgende Form:

```
Variablentyp Variablenname;
```

Dabei entspricht Variablentyp immer entweder einem einfachen Datentyp (`byte`, `short`, `int`, `long`, `float`, `double`, `char` oder `boolean`) oder einem Klassennamen (was das genau ist erfahren wir später). Variablenname ist eine eindeutige Zeichenkette, die den oben beschriebenen Regeln für Bezeichner entspricht. Es hat sich eingebürgert, Variablennamen in Java ohne Sonderzeichen zusammen zu schreiben, mit einem Kleinbuchstaben beginnen zu lassen, und jedes neue Wort innerhalb des Namens groß zu schreiben, wie etwa in `tollerVariablenName`. Daran wollen wir uns in Zukunft auch halten.

Um solchen Variablen nun Werte zuzuweisen, verwenden wir den **Zuweisungsoperator** `=`. Natürlich können wir einer Variable nur Werte zuweisen, die sich innerhalb des Wertebereichs des angegebenen Variablentyps befinden (siehe dazu auch Ganzzahlige Datentypen). So könnten wir zum Beispiel, um eine Variable `a` vom Typ `int` zu deklarieren und ihr den Wert 5 zuzuweisen, folgendes schreiben:

```
int a;
a = 5;
```

Da man in der Regel nach jeder Variablendeklaration auch gleich einen Wert in die Variable schreiben will, ist in Java auch folgende Kurzform erlaubt:

```
int a = 5;
```

Damit haben wir zwei Aufgaben auf einmal bewältigt, nämlich

1. **die Deklaration**, d.h. die Variable eingerichtet, und
2. **die Initialisierung** d.h. der Variablen auch gleich einen ersten Wert übergeben

Zurück zu unserem Beispiel. Wir wollten das Programm `StundenRechner1` so umschreiben, daß die Anzahl der geleisteten Arbeitsstunden nur noch an einer Stelle auftaucht. Die Lösung dafür liegt in der Verwendung einer Variablen für die Anzahl der Stunden. Das Programm sieht nun wie folgt aus:

```
public class StundenRechner2 {
  public static void main(String[] args) {

    int anzahlStunden = 12;

    System.out.print("Arbeitsstunden: ");
    System.out.println(anzahlStunden);
    System.out.print("Stundenlohn in DM: ");
    System.out.println(15);
    System.out.print("Damit habe ich letzte Woche soviel (in DM) verdient: ");
    System.out.println(anzahlStunden * 15);

  }
}
```

Zeile 4 enthält die benötigte Deklaration und gleichzeitig auch die Initialisierung der Variablen `anzahlStunden`. In den Zeilen 7 und 11 wird jetzt nur noch über den Variablennamen auf die Variable zugegriffen. Damit genügt, wenn wir nächste Woche unseren neuen Wochenlohn berechnen wollen, nur noch die Änderung in einer einzigen Programmzeile.

2.4.2 Operatoren

In der Regel will man mit Werten, die man in Variablen gespeichert hat, im Verlauf eines Programms mehr oder minder sinnvolle Berechnungen durchführen. Dazu stellt uns Java sogenannte Operatoren zur Verfügung. Auch in unserem letzten Beispielprogamm, `StundenRechner2`, haben wir schon

verschiedene Operatoren benutzt, ohne näher darauf einzugehen. Das wollen wir jetzt nachholen.

Mit Operatoren lassen sich Werte, auch **Operanden** genannt, miteinander verknüpfen. Wie schon im Kapitel Grundelemente eines Java-Programmes beschrieben, kann man Operatoren nach der Anzahl ihrer Operanden in drei Kategorien einteilen. **Einstellige** Operatoren haben einen, **zweistellige** Operatoren zwei und **dreistellige** Operatoren drei Operanden. Analog nennt bezeichnet man sie auch als **unäre**, **binäre** oder **ternäre** Operatoren bzw. als **monadische**, **dyadische** und **triadische** Operatoren.

Desweiteren muß geklärt werden, in welcher Reihenfolge Operatoren und ihre Operanden in Java Programmen geschrieben werden. Man spricht in diesem Zusammenhang auch von der **Notation** der Operatoren.

Die meisten einstelligen Operatoren werden in Java in der **Präfix** Notation verwendet. Eine Ausnahme davon bilden die Inkrement- und Dekrementoperatoren, die sowohl in **Präfix** als auch in **Postfix** Notation verwendet werden können. **Präfix** Notation bedeutet, daß der Operator vor seinem Operanden steht, also etwa

```
<Operator> <Operand>
```

Von **Postfix** Notation hingegen spricht man, wenn der Operator hinter seinem Operanden steht, also

```
<Operand> <Operator>
```

Zweistellige Operatoren in Java verwenden stets die **Infix** Notation, in der der Operator zwischen seinen zwei Operanden steht, etwa

```
<Operand> <Operator> <Operand>
```

Der einzige dreistellige Operator in Java, `?:` (siehe Vergleichs- und Bedingungsoperatoren), benutzt ebenfalls die **Infix** Notation, also

```
<Operand> ? <Operand> : <Operand>
```

Neben der Anzahl der Operanden kann man Operatoren auch nach dem **Typ** der Operanden einteilen. Die folgenden Abschnitte stellen, nach dieser (etwas längeren) Vorrede, die Operatoren von Java im einzelnen vor, gruppiert nach dem Typ ihrer Operanden.

Operator	Beispiel	Funktion
`+`	`a + b`	Addiert `a` und `b`
`-`	`a - b`	Subtrahiert `b` von `a`
`*`	`a * b`	Multipliziert `a` und `b`
`/`	`a / b`	Dividiert `a` durch `b`
`%`	`a % b`	Liefert den Rest der ganzzahligen Division `a` durch `b`

Tabelle 4: Zweistellige Arithmetische Operatoren

2.4.2.1 Arithmetische Operatoren Arithmetische Operatoren sind Operatoren, die Zahlen, also Werte vom Typ `byte`, `short`, `int`, `long`, `float`, `double` oder `char`, als Operanden erwarten. Sie sind in folgender Tabelle zusammengefaßt:

Wie man sieht, sind alle hier dargestellten Operatoren zweistellig — die Operatoren `+` und `-` können jedoch auch als einstellige Operatoren gebraucht werden und bewirken dann folgendes:

Operator	Beispiel	Funktion
`+`	`+ a`	Keine explizite Funktion, existiert nur der Symmetrie halber
`-`	`- a`	Negiert `a`

Tabelle 5: Einstellige Arithmetische Operatoren

Achtung: Der Operator `+` kann auch dazu benutzt werden, um zwei Zeichenketten zu einer einzigen zusammenzufügen. So ergibt

```
"abcd" + "efgh"
```

die Zeichenkette

```
"abcdefgh"
```

Wir wollen darauf aber erst später näher eingehen.

Eine weitere Besonderheit stellt der **Ergebnistyp** arithmetischer Operationen dar. Damit meinen wir den Typ (also `byte`, `short`, `int`, `long`, `float`,

`double` oder `char`) des Ergebnisses einer Operation, der durchaus nicht mit dem Typ der Operanden übereinstimmen muß.

Bestes Beispiel dafür sind Programmzeilen wie etwa

```
short a = 1;
short b = 2;
short c = a + b;
```

die, obwohl dem Anschein nach korrekt, beim Kompilieren zu folgender Fehlermeldung führen:

```
Incompatible type for declaration.
Explicit cast needed to convert int to short.
```

Warum? Um den Ergebnistyp einer arithmetischen Operation zu bestimmen, geht der Java Compiler wie folgt vor: Zunächst prüft er, ob einer der Operanden vom Typ `double` ist — ist dies der Fall, so ist der Ergebnistyp dieser Operation auf jeden Fall auch `double`, egal welchen Typ die übrigen Operanden haben. Analog verfährt der Compiler — in dieser Reihenfolge — mit den Typen `float` und `long`. Enthält der Ausdruck jedoch weder einen `double`, noch einen `float` oder `long`, so ist der Ergebnistyp auf jeden Fall ein `int`, unabhängig von den Typen der Operanden.

Damit wird auch klar, warum obiges Beispiel eine Fehlermeldung produziert — der Ausdruck `a + b` enthält keine der Typen `double`, `float` oder `long`, daher wird der Ergebnistyp ein `int`. Diesen versuchen wir nun ohne explizite Typkonvertierung einer Variable vom Typ `short` zuzuweisen, was zu einer Fehlermeldung führen muß, da der Wertebereich von `int` größer ist als der Wertebereich von `short`. Beheben läßt sich der Fehler jedoch ganz leicht, in dem man explizit eine Typkonvertierung erzwingt. Die Zeilen

```
short a = 1;
short b = 2;
short c = (short)(a + b);
```

lassen sich daher anstandslos kompilieren.

Was lernen wir daraus? Entweder verwenden wir ab jetzt für ganzzahlige Variablen nur noch den Typ `int` (dann haben wir nämlich den ganzen Ärger nicht mehr), oder aber wir achten bei jeder arithmetischen Operation darauf, das Ergebnis explizit in den geforderten Typ zu konvertieren. In jedem Falle aber wissen wir jetzt, wie wir Fehler dieser Art beheben können.

2.4.2.2 Bitoperatoren Um diese Kategorie von Operatoren zu verstehen, müssen wir uns zunächst klar machen, wie Werte im Computer gespeichert werden. Grundsätzlich kann ein Computer (bzw. die Elektronik, die in einem Computer enthalten ist) nur zwei Zustände unterscheiden, entweder Strom Aus oder Strom Ein. Diesen Zuständen ordnen wir nun der Einfachheit halber die Zahlenwerte 0 und 1 zu. Die kleinste Speichereinheit, in der ein Computer genau einen dieser Werte speichern kann, nennen wir ein **Bit**. Um nun beliebige Zahlen und Buchstaben darstellen zu können, faßt der Computer mehrere Bits zu neuen, größeren Einheiten zusammen. Dabei entsprechen 8 Bits einem **Byte**, 1024 Byte einem **Kilobyte**, 1024 Kilobyte einem **Megabyte** usw.

Um mit Bitwerten Berechnungen anstellen zu können, hat man nun verschiedene Operationen definiert. Auch hierbei unterscheidet man wieder zwischen unären und binären Operationen. Die einzige unäre Operation, die **Negation** (dargestellt durch das Zeichen `~`), liefert stets das Gegenteil des Eingangswertes, wie in folgender Tabelle dargestellt:

a	~a
0	1
1	0

Tabelle 6: Negation

Daneben existieren drei binäre Operationen, das logische **UND** (dargestellt durch `&`), das logische **inklusive ODER** (`|`) und das logische **exklusive ODER** (`^`), deren Ergebnistabellen im folgenden dargestellt sind:

a	b	a & b	a | b	a ^ b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Tabelle 7: UND, inklusiv ODER, exklusiv ODER

Diese Bitoperationen können auch auf mehrere Bits gleichzeitig angewendet werden. So ergibt beispielweise `1001 & 0011` die Kombination `0001`,

`1001 | 0011` dagegen `1011`.

Für all diese Operationen existieren nun natürlich auch entsprechende Operatoren in Java. Diese Bitoperatoren sind in folgender Tabelle aufgelistet:

Operator	Beispiel	Funktion
`~`	`~ a`	Negiert `a`
`&`	`a & b`	Verknüpft `a` und `b` durch ein logisches UND
`|`	`a | b`	Verknüpft `a` und `b` durch ein logisches ODER (inklusiv)
`^`	`a ^ b`	Verknüpft `a` und `b` durch ein logisches ODER (exklusiv)

Tabelle 8: Bitoperatoren

Daneben existieren noch drei **Schiebeoperatoren**, die die Bits eines Wertes um eine vorgegebene Anzahl von Stellen nach links bzw. rechts schieben, wie in folgender Tabelle aufgeführt:

Operator	Beispiel	Funktion
<<	`a << b`	Schiebt die Bits in `a` um `b` Stellen nach links und füllt mit 0-Bits auf
>>	`a >> b`	Schiebt die Bits in `a` um `b` Stellen nach rechts und füllt mit dem höchsten Bit von `a` auf
>>>	`a >>> b`	Schiebt die Bits in `a` um `b` Stellen nach rechts und füllt mit 0-Bits auf

Tabelle 9: Schiebeoperatoren

2.4.2.3 Zuweisungsoperator Eine Sonderstellung unter den Operatoren nimmt der **Zuweisungsoperator** `=` ein. Mit ihm kann man Werte einer Variablen zuordnen. Beipielsweise ordnet der Ausdruck

```
a = 3;
```

der Variablen `a` den Wert `3` zu. Um denselben Wert mehreren Variablen gleichzeitig zuzuordnen, kann man auch ganze Zuordnungsketten bilden, etwa

```
a = b = c = 5;
```

Hier wird der Wert `5` allen drei Variablen `a`, `b`, `c` auf einmal zugeordnet.

Achtung: Der Zuweisungsoperator `=` hat grundsätzlich nichts mit dem mathematischen "=" Zeichen zu tun. Ein Ausdruck der Form

```
a = a + 1
```

ist mathemetisch völlig unsinnig, die Java Anweisung

```
a = a + 1;
```

dagegen ist syntaktisch völlig korrekt und erhöht dem Wert der Variablen `a` um 1.

Will man mit dem Wert einer Variablen Berechnungen anstellen, und das Ergebnis danach in derselben Variable speichern, ist es oft lästig, denselben Variablennamen sowohl links als auch rechts des Zuweisungsoperators zu tippen. Daher bietet Java für viele binäre Operatoren auch eine verkürzende Schreibweise an. So kann man statt

```
a = a + 1;
```

auch kürzer

```
a += 1;
```

schreiben. Beide Ausdrücke sind in Java völlig äquivalent, beide erhöhen den Wert der Variablen `a` um 1. Die folgende Tabelle faßt alle möglichen abkürzenden Schreibweisen zusammen:

2.4.2.4 Vergleichs- und Bedingungsoperatoren

Eine weitere Gruppe von Operatoren bilden die sogenannten **Vergleichsoperatoren**. Diese stets binären Operatoren vergleichen ihre Operanden miteinander und geben immer ein Ergebnis vom Typ `boolean`, also entweder `true` oder `false`, zurück. Im einzelnen sind dies:

Um nun komplexe Ausdrücke zu erstellen, werden die Vergleichsoperatoren meist durch die sogenannten **Bedingungsoperatoren** verknüpft. Diese ähneln auf den ersten Blick den schon vorgestellten Bitoperatoren, allerdings

Abkürzung	Beispiel	äquivalent zu
`+=`	`a += b`	`a = a + b`
`-=`	`a -= b`	`a = a - b`
`*=`	`a *= b`	`a = a * b`
`/=`	`a /= b`	`a = a / b`
`%=`	`a %= b`	`a = a % b`
`&=`	`a &= b`	`a = a & b`
`|=`	`a |= b`	`a = a | b`
`^=`	`a ^= b`	`a = a ^ b`
`<<=`	`a <<= b`	`a = a << b`
`>>=`	`a >>= b`	`a = a >> b`
`>>>=`	`a >>>= b`	`a = a >>> b`

Tabelle 10: Abkürzende Schreibweisen für binäre Operatoren

Operator	Beispiel	liefert genau dann `true`, wenn ...
`>`	`a > b`	... `a` größer als `b` ist
`>=`	`a >= b`	... `a` größer als oder gleich `b` ist
`<`	`a < b`	... `a` kleiner als `b` ist
`<=`	`a <= b`	... `a` kleiner als oder gleich `b` ist
`==`	`a == b`	... `a` gleich `b` ist
`!=`	`a != b`	... `a` ungleich `b` ist

Tabelle 11: Vergleichsoperatoren

erwarten Bedingungsoperatoren stets Operanden vom Typ `boolean`, und ihr Ergebnistyp ist ebenfalls `boolean`. Wie bei den Bitoperatoren existieren auch hier Operatoren für logisches UND (Operator `&`), inklusives ODER (`|`) und Negation (hier ein `!`). Eine Besonderheit stellen die Operatoren `&&` und `||` dar: bei ihnen kommt das sogenannte **Short Circuit** Verfahren zum Tragen.

Beim **Short Circuit** (zu deutsch: Kurzschluß) Verfahren wird der zweite Operand nur dann ausgewertet, falls das Ergebnis der Operation nicht schon nach Auswertung des ersten Operanden klar ist. Ist also beispielsweise der Ausdruck `(a > 15) && (b < 20)` zu bewerten und der Wert der Variablen

a gerade 10, so ergibt der erste Teilausdruck (a > 15) zunächst false. Der Compiler prüft in diesem Fall den Wert der Variablen b gar nicht mehr nach, da ja der gesamte Ausdruck auch nur noch false sein kann.

Was haben wir nun davon? Zunächst kann man durch geschickte Ausnutzung des Short Circuit Verfahrens im Einzelfall die Ausführungsgeschwindigkeit des Programms deutlich steigern. Müssen an einer Stelle eines Programms zwei Bedingungen auf true überprüft werden, und ist das Ergebnis der einen Bedingung in 90% aller Fälle false, so empfiehlt es sich, diese Bedingung zuerst überprüfen zu lassen und die zweite über den Short Circuit Operator anzuschliessen. Jetzt muß die zweite Bedingung nur noch in den 10% aller Fälle überprüft werden, in denen die erste Bedingung wahr wird. In allen anderen Fällen läuft das Programm schneller ab. Desweiteren läßt sich, falls die Bedingungen nicht nur Variablen, sondern auch Aufrufe von Methoden enthalten (was das genau ist, erfahren wir später), mit Hilfe eines Short Circuit Operators erreichen, daß bestimmte Programmteile überhaupt nicht abgearbeitet werden. Doch dazu später mehr.

Die folgende Tabelle faßt alle Bedingungsoperatoren zusammen:

Operator	Beispiel	Funktion
&	a & b	Verknüpft a und b durch ein logisches UND
&&	a && b	Verknüpft a und b durch ein logisches UND mit Short Circuit
|	a | b	Verknüpft a und b durch ein logisches ODER (inklusiv)
||	a || b	Verknüpft a und b durch ein logisches ODER (inklusiv) mit Short Circuit
!	! a	Negiert a

Tabelle 12: Bedingungsoperatoren

Eine Sonderstellung unter den Vergleichs- und Bedingungsoperatoren nimmt der dreistellige Operator ?: ein. Eigentlich stellt er nur eine verkürzende Schreibweise für eine if-then Entscheidungsanweisung dar (siehe Entscheidungsanweisungen). Als ersten Operanden erwartet er einen Ausdruck mit Ergebnistyp boolean, zweiter und dritter Operand können beliebigen Typs sein. Liefert der ersten Operand true zurück, so gibt der Operator seinen zweiten Operanden zurück, liefert der erste Operand false, so ist der dritte Operand das Ergebnis der Operation.

Beispiel: Der Ausdruck

```
(a == 15) ? "a ist 15" : "a ist nicht 15"
```

liefert die Zeichenkette a ist 15, falls die Variable a den Wert 15 enthält, und die Zeichenkette a ist nicht 15 in allen anderen Fällen.

2.4.2.5 Inkrement- und Dekrementoperatoren Auch hier handelt es eigentlich nur um verkürzte Schreibweisen von häufig verwendeten Ausdrücken. Um den Inhalt der Variablen a um eins zu erhöhen, könnten wir — wie wir mittlerweile wissen — beispielsweise schreiben

```
a = a + 1;
```

Alternativ bietet sich — auch das haben wir schon gelernt — der verkürzte Zuweisungsoperator an, d.h.

```
a += 1;
```

In diesem speziellen Fall (Erhöhung des Variableninhaltes um genau 1) bietet sich jetzt eine noch kürzere Schreibweise, nämlich

```
a++;
```

Der **Inkrementoperator ++** ist also unär und erhöht den Wert seines Operanden um eins. Analog dazu erniedrigt der **Dekrementoperator --**, ebenfalls ein unärer Operator, den Wert seines Operanden um eins.

Was bleibt zu beachten? Wie bereits erwähnt, können beide Operatoren sowohl in Präfix als auch in Postfix Notation verwendet werden. Wird der Operator in einer isolierten Anweisung — wie in obigem Beispiel — verwendet, so sind beide Notationen äquivalent.

Sind Inkrement- bzw. Dekrementoperator jedoch Teil eines größeren Ausdrucks, so hat die Notation entscheidenden Einfluß auf das Ergebnis des Ausdrucks. Bei Verwendung der Präfix Notation wird der Wert der Variablen *erst* erhöht bzw. erniedrigt, und *dann* der Ausdruck ausgewertet. Analog dazu wird bei der Postfix Notation *zuerst* der Ausdruck ausgewertet und *dann* erst das Inkrement bzw. Dekrement durchgeführt.

Beispiel:

```
a = 5;
b = a++;

c = 5;
d = --c;
```

Nach Ausführung dieses Programmsegments enthält `b` den Wert `5`, da *zuerst* der Ausdruck ausgewertet wird und *dann* das Inkrement ausgeführt wird. `d` dagegen enthält den Wert `4`, da hier *zuerst* das Dekrement durchgeführt wird und *dann* der Ausdruck ausgewertet wird. (Kleine Zwischenfrage: Welchen Wert enthalten die Variablen `a` und `c`? — Für jede richtige Antwort gibt es einen feuchten Händedruck `:-)`

2.4.3 Priorität der Operatoren

Bislang haben wir alle Operatoren nur isoliert betrachtet, d.h. unsere Ausdrücke enthielten jeweils nur einen Operator. Verwendet man jedoch mehrere Operatoren in einem Ausdruck, stellt sich die Frage, in welcher Reihenfolge die einzelnen Operationen ausgeführt werden. Dies ist durch die Prioritäten der einzelnen Operatoren festgelegt. Dabei werden Operationen höherer Priorität stets vor Operationen niedrigerer Priorität ausgeführt. Haben zwei Operationen, die im gleichn Ausdruck stehen, die gleiche Priorität, so wird — außer bei Zuweisungsoperatoren — stets von links nach rechts ausgewertet. Der Zuweisungsoperator `=` sowie alle verkürzten Zuweisungsoperatoren — also `+=`, `-=`, usw. — werden, wenn sie nebeneinander in einem Ausdruck vorkommen, dagegen von rechts nach links ausgewertet.

Die folgende Tabelle enthält alle Operatoren, die wir bisher kennengelernt haben, geordnet nach deren Priorität.

Praktisch bedeutet das für uns, daß wir bedenkenlos Punkt-vor-Strich-Rechnung verwenden können, ohne uns über Prioritäten Gedanken machen zu müssen — da multiplikative Operatoren eine höhere Priorität als additive haben, werden Ausdrücke wie z.B.:

```
4 + 3 * 2
```

wie erwartet korrekt ausgewertet (hier: Ergebnis ist `10`, nicht `14`). Darüber hinaus sollten wir aber lieber ein Klammernpaar zuviel als zuwenig verwenden, um die gewünschte Ausführungsreihenfolge der Operationen zu garantieren. Mit den runden Klammern `()` können wir nämlich, genau wie in der Mathemetik, die Reihenfolge der Operationen eindeutig festlegen, gleichgültig welche Priorität ihnen zugeordnet ist.

2.4.4 Zusammenfassung 2.4

Der letzte Abschnitt war zugegebenermaßen etwas länger als die anderen, dafür haben wir aber schon eine Menge gelernt. Wir wissen nun, was Variablen sind (Analogie: Postfächer), haben zahlreiche Operatoren kennengelernt

Bezeichnung	Operator	Priorität
Unäre Operatoren in Postfix Notation	`Operand++`, `Operand--`	15
Unäre Operatoren in Präfix Notation	`++Operand`, `--Operand`, `+Operand`, `-Operand`, `~`, `!`	14
Explizite Typkonvertierung	`(typ)Operand`	13
Multiplikative Operatoren	`*`, `/`, `%`	12
Additive Operatoren	`+`, `-`	11
Schiebeoperatoren	`<<`, `>>`, `>>>`	10
Vergleichsoperatoren	`<`, `>`, `<=`, `>=`	9
Vergleichsoperatoren (Gleichheit/Ungleichheit)	`==`, `!=`	8
bitweise UND	`&`	7
bitweise exklusives ODER	`^`	6
bitweise inklusives ODER	`|`	5
logisches UND	`&&`	4
logisches ODER	`||`	3
Bedingungsoperator	`?:`	2
Zuweisungsoperatoren	`=`, `+=`, `-=`, usw.	1

Tabelle 13: Priorität der Operatoren

mit denen wir Werte — also Variablen und Konstanten — bearbeiten können und verstehen, wie man Werte und Operatoren zu Ausdrücken verknüpfen kann.

2.5 Ablaufsteuerung

Als letztes Grundelement der Sprache Java werden wir in den folgenden Abschnitten Befehle kennenlernen, mit denen wir den Ablauf unseres Programms beeinflussen können, d.h. bestimmen können, ob und in welcher Reihenfolge bestimmte Anweisungen unseres Programms ausgeführt werden. In diesem Zusammenhang ist auch der Begriff eines **Blocks** in Java zu klären: Ein **Block** enthält eine Folge von Anweisungen und wird von außen wie eine einzige Anweisung behandelt (siehe Blöcke).

Die folgenden Abschnitte erläutern Befehle zur Ablaufsteuerung geord-

net nach deren Wirkungsweise: Entscheidungsanweisungen, Schleifen und Sprungbefehle.

Achtung: Neben den hier vorgestellten Befehlen zur Ablaufsteuerung existiert noch eine weitere Gruppe solcher Befehle, die man in Zusammenhang mit Ausnahmen in Java verwendet. Diese Gruppe umfaßt die Befehle `try-catch-finally` und `throw`. Wir werden auf diese Befehle in einem späteren Teil unseres Skriptes eingehen.

2.5.1 Blöcke

In der Programmiersprache Java bezeichnet ein Block eine Folge von Anweisungen. Alle diese Anweisungen werden von außen wie eine einzige Anweisung behandelt. Blöcke können beliebig ineinander verschachtelt werden. Dabei werden Blockanfang und -ende durch geschweifte Klammern {} markiert (siehe Interpunktionszeichen). Folgender Programmausschnitt enthält beispielsweise einen großen (äußeren) Block, in den zwei (innere) Blöcke geschachtelt sind:

```
{               // Anfang des ueren Blocks
  Anweisung1;
  Anweisung2;
  {             // Anfang des ersten inneren Blocks
    Anweisung3;
    Anweisung4;
    Anweisung5;
  }             // Ende des ersten inneren Blocks
  Anweisung6;
  {             // Anfang des zweiten inneren Blocks
    Anweisung7;
    Anweisung8;
  }             // Ende des zweiten inneren Blocks
}               // Ende des ueren Blocks
```

Anzumerken bleibt, daß Variablen, die wir in unserem Programm deklarieren (siehe Variablen), immer nur bis zum Ende des aktuellen Blocks gültig sind. Man spricht in diesem Zusammenhang auch vom **Sichtbarkeitsbereich** einer Variablen. Hätten wir z.B. die Variable `a` in `Anweisung3` des obigen Beipielprogramms deklariert, so dürften wir in `Anweisung6` — und danach — nicht mehr auf `a` zugreifen, da die Variable `a` mit der schließenden geschweiften Klammer zwischen `Anweisung5` und `Anweisung6` ihre Gültigkeit verloren hätte. Man könnte auch sagen, die Variable `a` ist nur innerhalb des ersten inneren Blocks **sichtbar**.

2.5.2 Entscheidungsanweisungen

Die wohl grundlegendste Entscheidungsanweisung vieler Programmiersprachen stellt die sogenannte `if-else`-Anweisung (zu deutsch: wenn-sonst) dar. Die Syntax dieser Anweisung in Java ist in folgendem Programmcode dargestellt:

```
if (Ausdruck)
  Anweisung1;
else
  Anweisung2;
```

Trifft der Java Interpreter während der Programmausführung auf eine solche `if-else`-Anweisung, so wertet er zunächst den Ausdruck `Ausdruck` aus, dessen Ergebnistyp `boolean` sein muß. Ist das Ergebnis `true`, so wird `Anweisung1` ausgeführt, ist es `false`, so kommt `Anweisung2` zur Ausführung. Danach wird in jedem Fall mit der nächstfolgenden Anweisung fortgefahren. Es wird jedoch immer *genau eine* der beiden Anweisungen ausgeführt — `Anweisung1` und `Anweisung2` können also niemals beide gleichzeitig zur Ausführung kommen.

Will man keine Anweisungen durchführen, wenn der Ausdruck das Ergebnis `false` leifert, so kann man den `else`-Teil auch komplett weglassen.

Zu beachten ist, daß `Anweisung1` bzw. `Anweisung2` auch durch ganzen Blöcke ersetzt werden können, falls man die Ausführung mehrerer Anweisungen vom Ergebnis des Ausdrucks abhängig machen will.

Achtung: Welche Anweisungen zu welchem Block gehören wird ausschließlich durch die geschweiften Klammern festgelegt. Sind die Anweisungen im `if-` oder im `else`-Teil nicht geklammert, so gehört nur die erste Anweisung in diesen Teil, egal wie die nachfolgenden Anweisungen eingerückt sind. Im Programmausschnitt

```
if (Ausdruck) {
  Anweisung1;
  Anweisung2;
  }
else
  Anweisung3;
  Anweisung4;
Anweisung5;
```

werden also `Anweisung4` und `Anweisung5` auf jeden Fall ausgeführt, egal welches Ergebnis der Ausdruck `Ausdruck` lieferte. Es empfiehlt sich daher, `if-` und `else`-Teile *immer* in Klammern zu setzen, auch wenn sie nur aus

einer einzigen Anweisung bestehen. Nur so ist sofort ersichtlich, welche Anweisungen zu diesen Teilen gehören und welche nicht.

Eine weitere Entscheidungsanweisung stellt die `switch-case-default` Kombination dar, deren Syntax wie folgt lautet:

```
switch (Ausdruck) {
case Konstante1:
  Anweisung1;
  break;
case Konstante2:
  Anweisung2;
  break;
...
default:
  defaultAnweisung;
}
```

Hier wird zunächst der Ausdruck `Ausdruck` ausgewertet, dessen Ergebnistyp ein `byte`, `short`, `int`, `long` oder `char` sein muß. Darauf springt der Interpreter zu genau der `case-` Anweisung, die als Konstante das Ergebnis des Ausdrucks enthält. Wird das Ergebnis in keiner `case-`Anweisung gefunden, so wird die Programmausführung mit der `default`-Anweisung fortgesetzt.

Zu beachten sind die in obigem Beispiel angegebenen `break-`Anweisungen: Normalerweise fährt das Programm nach der Ausführung einer `case-` Anweisung direkt mit der Ausführung der folgenden `case-`Anweisung fort. Die `break`-Anweisung hingegen bricht die Ausführung der gesamten `switch`-Anweisung ab und setzt mit der ersten Anweisung außerhalb der `switch`-Anweisung fort (siehe Sprungbefehle).

Dazu ein Beispiel:

```
int a, b;
...
switch (a) {
case 1:
  b = 10;
case 2:
case 3:
  b = 20;
  break;
case 4:
  b = 30;
  break;
default:
  b = 40;
}
```

In dieser `switch`-Anweisung wird der Variablen `b` der Wert `20` zugewiesen, falls `a` den Wert `1`, `2` oder `3` hat. Warum? Hat `a` den Wert `1`, so wird

zunächst in die erste `case-`Anweisung gesprungen und der Variablen `b` der Wert `10` zugewiesen. Danach fährt die Bearbeitung jedoch mit der nächsten `case`-Anweisung fort, da es nicht mit einer `break`-Anweisung explizit zum Verlassen der gesamten `switch`-Anweisung aufgefordert wurde. In der zweiten `case-`Anweisung wird gar kein Befehl ausgeführt und die Bearbeitung setzt mit der dritten `case-`Anweisung fort. Jetzt wird der Variablen `b` der Wert `20` zugeordnet und anschließend die gesamte `switch-`Anweisung per `break`-Anweisung verlassen.

Enthält `a` zu Beginn der `switch`-Anweisung den Wert `4`, so wird `b` der Wert `30` zugewiesen, in allen anderen Fällen schließlich enthält `b` nach Ausführung der `switch`-Anweisung den Wert `40`.

Selbstverständlich kann die Angabe einer `default`-Anweisung auch unterbleiben, und statt einer Anweisung können `case`-Anweisungen auch beliebig viele Anweisungen oder Blöcke enthalten.

2.5.3 Schleifen

Eine weitere Gruppe der Befehle zur Ablaufsteuerung stellen die sogenannten **Schleifen** dar. Hier kann eine Anweisung bzw. ein Block von Anweisungen mehrmals hintereinander ausgeführt werden.

Der erste Vertreter dieser Schleifen ist die `for`-Anweisung, die eine **Zählschleife** implementiert. Ihre Syntax lautet:

```
for (Initialisierung; Ausdruck; Zaehlanweisung)
  Anweisung1;
```

Dabei wird zunächst im Teil `Initialisierung` eine Zählvariable initialisiert (und eventuell auch gleich deklariert), und daraufhin `Anweisung1` ausgeführt, solange der Ausdruck `Ausdruck`, dessen Ergebnistyp wieder `boolean` sein muß, den Wert `true` liefert. Dabei wird nach jeder Ausführung von `Anweisung1` zusätzlich noch die Anweisung `Zählanweisung` ausgeführt. Das hört sich jetzt alles komplizierter an, als es wirklich ist, daher am besten erstmal wieder ein Beispiel:

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

Dieses Programmstück macht nichts anderes, als die Zahlen `0` bis `9` auf dem Bildschirm auszudrucken. Wie funktioniert das? Zunächst wir die Initialisierungsanweisung `int i = 0;` ausgeführt, die die Variable `i` deklariert

und sie mit dem Wert `0` initialisiert. Als nächstes wird der Ausdruck `i < 10;` ausgewertet — dies ergibt `true`, da `i` ja gerade den Wert `0` hat, die Anweisung `System.out.println(i);` wird also ausgeführt und druckt die Zahl `0` auf den Bildschirm. Nun wird zunächst die Zählanweisung `i++` durchgeführt, die den Wert von `i` um eins erhöht, und danach wieder der Ausdruck `i < 10;` ausgewertet, auch diesmal wieder `true` als Ergebnis liefert —was zur Folge hat, daß die Anweisung `System.out.println(i);` erneut zur Ausführung kommt. Dieses Spielchen setzt sich solange fort, bis der Ausdruck `i < 10;` das Ergebnis `false` liefert, was genau dann zum ersten Mal der Fall ist, wenn die Variable `i` den Wert `10` angenommen hat, worauf die Anweisung `System.out.println(i);` nicht mehr ausgeführt wird und die Schleife beendet wird.

Analog zu dem Beispiel läßt sich beispielweise auch folgende Schleife programmieren:

```
for (int i = 9; i >= 0; i--) {
  System.out.println(i);
}
```

Hier werden nun, ihr ahnt es schon, wieder die Zahlen `0` bis `9` auf dem Bildschirm ausgegeben, diesmal jedoch in umgekehrter Reihenfolge.

Anzumerken bleibt, daß man bei solchen Zählschleifen die Anweisung wieder durch einen ganzen Block von Anweisungen ersetzen kann. Auch hier ist es — wie schon bei `if-else`-Anweisungen — sinnvoll, die zur Schleife gehörigen Anweisungen *immer* zu klammern, auch wenn nur eine Anweisung existiert, um möglichen Verwechslungen vorzubeugen.

Einen weiteren Schleifentyp stellt die **abweisende Schleife** `while` dar:

```
while (Ausdruck)
  Anweisung1;
```

Hier wird `Anweisung1` ausgeführt, solange `Ausdruck`, dessen Ergebnis — wir vermuten es ja schon — wieder vom Typ `boolean` sein muß, den Wert `true` zurückliefert.

Analog dazu wird auch bei der **nicht-abweisenden Schleife** `do-while` der Form

```
do
  Anweisung1;
while (Ausdruck);
```


die Anweisung `Anweisung1` ausgeführt, solange `Ausdruck` den Wert `true` ergibt. Der große Unterschied zwischen diesen beiden Schleifen ist die Tatsache, daß bei der abweisenden Schleife der Ausdruck `Ausdruck` *noch vor der ersten Ausführung* von `Anweisung1` überprüft wird, während bei der nicht-abweisenden Schleife der Ausdruck *erst nach der ersten Durchführung* von `Anweisung1` ausgewertet wird. Es kann daher vorkommen, daß `Anweisung1` bei der abweisenden Schleife kein einziges Mal ausgeführt wird, während `Anweisung1` bei der nicht-abweisenden Schleife auf jeden Fall mindestens einmal ausgeführt wird.

Selbstverständlich kann man in beiden Schleifenvariaten `Anweisung1` wieder durch einen kompletten Block von Anweisungen ersetzen, und auch hier empfiehlt es sich, die zur Schleife gehörigen Anweisungen stets in geschweifte Klammern zu setzen.

Ein beliebtes Informatikerspielchen (und damit auch ein wahrscheinlicher Kandidat für Übungs- und Klausuraufgaben) ist die Umwandlung einer Zählschleife in eine abweisende bzw. nicht-abweisende Schleife und umgekehrt. Dazu ein wieder ein kleines Beispiel:

Die Schleife

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

ist völlig äquivalent zum ersten Beispiel einer `for`-Schleife weiter oben in diesem Abschnitt. Wie könnte eine nicht-abweisende Schleife gleicher Funktionalität aussehen? Was bewirkt die Schleife

```
int i = 20;
while (i > 0) {
  System.out.println(i);
  i -= 2;
}
```

und wie lautet eine `for`-Schleife mit gleicher Ausgabe? Und was bewirken die Zeilen:

```
while (true) {
  System.out.println("Aloha");
}
```

Das dürft ihr euch jetzt selber aus den Fingern saugen `:-)` Solltet ihr nicht draufkommen, fragt doch mal euren Nachbarn im Rechnertutorium oder euren Tutor ...

2.5.4 Sprungbefehle

Zuletzt wollen wir uns noch mit der Klasse der sogenannten **Sprungbefehle** vertraut machen, mit denen man gezielt zu einer bestimmten Stelle im Programm springen kann. Solltet ihr schon etwas Programmiererfahrung mitbringen, eine kleine Warnung vorweg: In der Sprache Java gibt es, im Gegensatz zu anderen Programmiersprachen *keine* `goto`-Anweisung, und das ist auch gut so. Überhaupt sollte man die hier vorgestellten Befehle, insbesondere `label` und `continue` nur mit Bedacht einsetzen, denn nichts ist unübersichtlicher (und damit fehleranfälliger) als Programme, in denen ständig wild hin- und hergesprungen wird.

In Java ist es möglich, jede Anweisung durch ein sogenanntes `label` mit einem eindeutigen Namen, etwa der Form

```
Marke: Anweisung;
```

zu kennzeichnen. Dadurch wird es möglich, mit den Befehlen `break` und `continue` direkt zu den so gekennzeichneten Anweisungen zu springen.

Die Anweisung `break` haben wir schon im Zusammenhang mit der `switch`-Anweisung kennengelernt. Sie dient dazu, den gerade in Ausführung befindlichen Block bzw. die gerade in Ausführung befindliche Schlife zu unterbrechen und mit der Anweisung, die direkt nach dem Block bzw. der Schleife folgt, fortzufahren. Alternativ kann man hinter der `break`-Anweisung eine Sprungmarke angeben, die dann direkt angesprungen wird, etwa

```
break Marke;
```

Die Anweisung `continue` entspricht der `break`-Anweisung, nur daß jetzt bei Schleifen nicht mit der auf die Schleife folgenden Anweisung fortgefahren wird, sondern mit dem nächsten Schleifendurchlauf. Beispiel:

```
while (a < 100) {
  Anweisung1;
  if (a < 10)
    continue;
  Anweisung2;
}
```

ist in seiner Ausführung völlig äquivalent zu

```
while (a < 100) {
  Anweisung1;
  if (a >= 10)
    Anweisung2;
}
```


Die `return`-Anweisung nimmt wiederum eine Sonderstellung ein. Sie dient dazu, aufgerufene Methoden zu beenden und eventuell einen Rückgabewert mit auf den Weg zu geben. Da wir jedoch noch nicht wissen, was Methoden sind und wie sie aufgerufen werden, werden wir uns später nocheinmal ausführlicher mit der `return`-Anweisung befassen.

2.5.5 Zusammenfassung 2.5

Wir haben gesehen, wie wir Anweisungen zur Ablaufsteuerung dazu einsetzen können, um zu bestimmen, ob und wann andere Anweisungen in unserem Programm ausgeführt werden. Wir haben Blöcke, Entscheidungsanweisungen, Schleifen und Sprungbefehle kennengelernt.

Und nach soviel Theorie raucht uns jetzt ordentlich der Kopf. Daher stürzen wir uns in unserem nächsten Kapitel Praxisbeispiele erstmal voll in die Praxis.

3 Praxisbeispiele

3.1 Worum geht es in diesem Kapitel?

Wir haben in den vergangenen Kapiteln eine Menge über Java gelernt und sind nun theoretisch in der Lage, die ersten komplexeren Programme zu schreiben. Leider kann man eine Sprache anhand der Theorie genausowenig begreifen wie das Autofahren – wir benötigen *Praxis*.

Wir werden deshalb in den folgenden Abschnitten verschiedene Aufgabenstellungen zu lösen versuchen und uns hierbei vor allem damit befassen, wie man an ein Problem

- systematisch herangeht,
- eine Lösung sucht und
- diese in Java programmiert.

Ein Teil der folgenden Aufgaben wurde aus der Vorlesung *Programmieren in C++* des Instituts für Angewandte Mathematik in Karlsruhe übernommen. Die Autoren bedanken sich für die Inspiration :-)

3.2 *Aufgabe 1:* Teilbarkeit zum ersten

3.2.1 Aufgabenstellung 1

Gegeben sei eine dreistellige ganze Zahl. zwischen 100 und 999. Durch welche ihrer Ziffern ist diese Zahl teilbar?

3.2.2 Analyse des Problems 1

Wir haben in dieser Aufgabe zwei Nüsse zu knacken:

- Wie spalte ich eine Zahl in ihre Ziffern auf?
- Wie prüfe ich, ob eine Zahl teilbar durch eine andere ist?

Wir wollen uns mit dem ersten Problem näher beschäftigen. Nehmen wir etwa die Zahl 123 und versuchen, sie mit den uns bekannten Operatoren zu untergliedern. Wie kommen wir etwa an die Einer-Stelle heran? Wir erinnern uns an die Schulmathematik, nach der wir Division mit Rest wie folgt durchgeführt haben:


```
123 : 10 = 12 mit Rest 3.
```

Wir sehen also, daß eine simple Division uns die rechte Ziffer bescheren kann (diese ist naemlich der Rest). Wir können in Java diese Berechnung mit Hilfe des Modulo-Operators `%` durchführen.

Wie kommen wir aber an die Zehner- oder Hunderterstelle heran? Auch diese Frage haben wir mit obiger Rechnung beantwortet. Teilen wir 123 durch 10, so erhalten wir als Ergebnis 12. Die Zehnerstelle ist also um eine Position nach rechts gerückt und kann somit wieder durch den Modulo-Operator berechnet werden.

Es bleibt noch die Frage, wie wir die Teilbarkeit überprüfen. Nehmen wir zu Anfang einmal an, die Ziffer sein ungleich der Null (eine Division wäre sonst schließlich nicht möglich). In diesem Fall ist eine Zahl durch eine andere offensichtlich teilbar, wenn bei der Division kein Rest entsteht. Ist der Rest also =0, ist die Teilbarkeit erfüllt.

3.2.3 Algorithmische Beschreibung 1

Was ist ein Algorithmus? Wir wollen einen Algorithmus als eine Verfahrensvorschrift zur Lösung eines bestimmten Problems bezeichnen. Bevor wir das Programm in Java umsetzen, werden wir stets die wichtigsten Punkte auf diese Art zusammenfassen – somit schaffen wir uns einen Überblick.

Folgendes Vorgehen erscheint nach unserem Überlegungen als logisch:

1. **Initialisierung** (d.h. vorbereitende Massnahmen)
   (a) Weise der Variablen `zahl` den zu pruefenden Wert zu
2. **Bestimmung der einzelnen Ziffern**
   (a) Bestimme die Einerstelle: `einer = zahl % 10`
   (b) Bestimme die Zehnerstelle: `zehner = (zahl / 10) % 10`
   (c) Bestimme die Hunderterstelle: `hunderter = (zahl / 100) % 10`
3. **Teilbarkeitstest**
   (a) Ist `einer` ungleich null und ergibt `zahl/einer` keinen Rest, gib `einer` aus
   (b) Ist `zehner` ungleich null und ergibt die Division keinen Rest, gib `zehner` aus

(c) verfahre genauso mit den Hundertern

Wir wollen nun überlegen, wie wir dieses Programm in Java implementieren.

3.2.4 Programmierung in Java 1

Wir öffnen im Editor eine Datei `Teilbarkeit1.java` und beginnen wie üblich zuerst mit dem Programmkopf

```
/**
 @author Jens Scheffler
 @version 1.0
 */

/**
 Dieses Programm spaltet eine dreistellige Zahl in ihre Ziffern auf
 und testet, ob die Zahl durch ihre einzelnen Ziffern teilbar ist.
 */

 public class Teilbarkeit1 {
   public static void main(String[] args) {
```

Als nächstes beginnen wir mit der Initialisierung. Wir benötigen eine Zahl, die es zu testen gilt. Wir können diese entweder im Programm zuweisen oder von der Tastatur einlesen. Im letzteren Fall verwenden wir die IOTools, wie es im Anhang beschrieben wird. Wir schreiben also eine der folgenden Zeilen:

```
int zahl = 123;                    // Fest vorgegebene Zahl
int zahl = IOTools.readInteger();  // Eingabe per Tastatur
```

Als nächstes müssen wir die Zahl in ihre einzelnen Ziffern aufspalten. Hierzu können wir die Rechenvorschriften aus dem letzten Abschnitt übernehmen.

```
int einer     =  zahl       % 10;  // Bestimme die Einer
int zehner    = (zahl /  10) % 10;  // Bestimme die Zehner
int hunderter = (zahl / 100) % 10;  // Bestimme die Hunderter
```

Wir haben jetzt also die einzelnen Ziffern und die ganze Zahl in Variablen gespeichert. Kommen wir also zum letzten Punkt des Algorithmus – dem Teilbarkeitstest. Für die erste Stelle sieht dieser etwa wie folgt aus:

```
if (einer != 0 &&                  // Ist Division moeglich?
    zahl % einer == 0)             // Ist der Rest =0 ?
    System.out.println("Die Zahl " + zahl + " ist durch "
                       + einer + " teilbar!");
```


Was passiert hierbei in der Bedingung? Kann jemals eine Division durch Null auftreten, falls die Einsstelle =0 ist? Dies würde schließlich zu einem Programmabsturz führen!

Die beruhigende Antwort ist: **nein!** Der `&&`-Operator stellt nur solange Berechnungen an, wie das Ergebnis nicht feststeht. Ist `einer==0`, so ist der erste Teil der Bedingung ohnehin schon falsch – sie kann also nicht mehr erfüllt werden. Das Programm rechnet an dieser Stelle nicht weiter und der Fehler tritt nicht auf.

Zu guter letzt dürfen wir natürlich nicht vergessen, die geöffneten Klammern auch wieder zu schließen. Unser fertiges Programm sieht nun wie folgt aus:

```
import Prog1Tools.IOTools;

public class Teilbarkeit1 {
  public static void main(String[] args) {
    int zahl = IOTools.readInteger();   // Eingabe per Tastatur
    int einer     =  zahl       % 10;  // Bestimme die Einer
    int zehner    = (zahl /  10) % 10;  // Bestimme die Zehner
    int hunderter = (zahl / 100) % 10;  // Bestimme die Hunderter
    if (einer != 0 &&                   // Ist Division moeglich?
        zahl % einer == 0)              // Ist der Rest =0 ?
        System.out.println("Die Zahl " + zahl + " ist durch "
                           + einer + " teilbar!");
    if (zehner != 0 &&                  // Ist Division moeglich?
        zahl % zehner == 0)             // Ist der Rest =0 ?
        System.out.println("Die Zahl " + zahl + " ist durch "
                           + zehner + " teilbar!");
    if (hunderter != 0 &&               // Ist Division moeglich?
        zahl % hunderter == 0)          // Ist der Rest =0 ?
        System.out.println("Die Zahl " + zahl + " ist durch "
                           + hunderter + " teilbar!");
  }
}
```

Wir übersetzen das Programm mit dem Befehl `javac Teilbarkeit1.java` und starten es anschließend. Geben wir etwa die Zahl 123 ein, erhalten wir als Ausgabe

```
Die Zahl 123 ist durch 3 teilbar!
Die Zahl 123 ist durch 1 teilbar!
```

3.2.5 Vorsicht Falle 1

Welche Stolperstricke haben sich in dieser Aufgabe für uns ergeben? An folgenden Stellen treten beim Programmieren gerne Fehler auf:

- Wir vergessen das eine oder andere Semikolon. Der Compiler bedankt sich mit einer Fehlermeldung der Form
  `';' expected.`
- Wir verwechseln in einer `if`-Abfrage die Operatoren `==` und `=`. Wir erhalten die Fehlermeldung
  `Invalid left hand side of assignment.`
- Wir schreiben in der `if`-Abfrage anstelle von `&&` den `&`-Operator. Der Compiler beschwert sich zwar nicht – das Programm ist syntaktisch vollkommen korrekt. Starten wir aber das Programm und geben an der falschen Stelle eine Null ein, so erhalten wir
  `java.lang.ArithmeticException: / by zero`
  und das Programm stürzt ab.

3.2.6 Übungsaufgabe 1

Statt der Zahl wollen wir testen, ob deren Quersumme durch eine der einzelnen Ziffern teilbar ist.

3.3 *Aufgabe 2:* Teilbarkeit zum zweiten

3.3.1 Aufgabenstellung 2

Wir wollen das vorherige Problem noch einmal lösen – nur diesmal darf die Zahl beliebig lang sein ...

3.3.2 Analyse des Problems 2

Wir wissen nicht, wievielstellig die neue Zahl ist. Wie sollen wir sie also in einzelne Ziffern aufteilen?

An dieser Stelle müssen wir deshalb ein wenig umdenken. Haben wir im vorigen Abschnitt zuerst *alle* Ziffern berechnet und dann den Teilbarkeitstest gemacht, werden wir nun eine Ziffer nach der anderen betrachten müssen. Die Rechenvorschrift für das Erhalten der einzelnen Ziffern ist hierbei identisch. Wir berechnen den Rest der Division, um die Einerstelle zu erhalten. Danach teilen wir die Zahl durch 10, um alle Ziffern nach rechts zu schieben.

Diese Vorgehensweise wirft natürlich wieder zwei neue Probleme auf:

- Wenn wir die Zahl durch 10 teilen, wie sollen wir sie dann noch vergleichen?

- Wann können wir mit der Ziffernberechnung aufhören?

Der erste Fall ist relativ einfach zu lösen. Wir kopieren den Inhalt der Variable `zahl` einfach in eine andere Variable `dummy`, welche wir nun nach Belieben verändern können. Das Original bleibt uns jedoch erhalten.

Auch auf die zweite Frage (die Frage nach dem sogenannten Abbruchkriterium) ist schnell eine Antwort gefunden. Wir hören auf, sobald alle Ziffern abgearbeitet sind. Dies ist der Fall, wenn in `dummy` keine weiteren Ziffern stehen, also `dummy==0` gilt.

3.3.3 Algorithmische Beschreibung 2

1. **Initialisierung**
   (a) Weise der Variablen `zahl` den zu pruefenden Wert zu
   (b) Mache eine Kopie von `zahl` in der Variablen `dummy`
2. **Schleife**
   Wiederhole die folgenden Instruktionen, solange `dummy!=0` ist.
   (a) Bestimme die Einerstelle : `einer = dummy % 10`
   (b) Schiebe die Ziffern nach rechts: `dummy = dummy / 10`
   (c) Führe den Teilbarkeitstest durch

3.3.4 Programmierung in Java 2

Da sich das Programm in vielen Punkten nicht von dem vorherigen unterscheidet, stellen wir es gleich in einem Stück vor:

```
/**
 @author Jens Scheffler
 @version 1.0
 */

/**
 Dieses Programm spaltet eine Zahl in ihre Ziffern auf
 und testet, ob die Zahl durch ihre einzelnen Ziffern teilbar ist.
 */

 import ProgiTools.IOTools;

 public class Teilbarkeit2 {
   public static void main(String[] args) {
     // 1. INITIALISIERUNG
```

```
    // ================
    int zahl = IOTools.readInteger(); // Eingabe per Tastatur
    int dummy = zahl;                 // Kopie erstellen
    // 2. SCHLEIFE
    // ==========
    while (dummy != 0) {              // Schleifenbedingung
      int einer = dummy % 10;         // Berechne einer
      dummy = dummy / 10;             // Schiebe Ziffern nach rechts
      if (einer != 0 &                // Ist Division moeglich?
          zahl % einer == 0)          // Ist der Rest =0 ?
        System.out.println("Die Zahl " + zahl + " ist durch "
                           + einer + " teilbar!");
    }                                 // Schleifenende
  }
}
```

Wir sehen, daß wir den Teilbarkeitstest wortwörtlich aus dem Programm `Teilbarkeit1` übernehmen konnten. Ansonsten ist das Programm durch die Verwendung der Schleife sogar noch etwas kürzer geworden. Starten wir das Programm nun und geben etwa die Zahle `123456` ein, so erhalten wir folgende Ausgabe:

```
Die Zahl 123456 ist durch 6 teilbar!
Die Zahl 123456 ist durch 4 teilbar!
Die Zahl 123456 ist durch 3 teilbar!
Die Zahl 123456 ist durch 2 teilbar!
Die Zahl 123456 ist durch 1 teilbar!
```

3.3.5 Vorsicht Falle 2

Neben den bereits von Aufgabe 1 bekannten Problemen gibt es hier noch einen weiteren Punkt, auf den zu achten ist. Hatten wir im ersten Programm noch lediglich zwei Klammern geöffnet (eine für den Programmbeginn, eine für den Start der Hauptroutine), so ist durch die Schleife noch eine weitere Klammer hinzugekommen. Schließen wir diese nicht, so erhalten wir die Fehlermeldung
`'}' expected.`

Wir könnten natürlich auch auf die Idee kommen, die zu der Schleife gehörigen Klammern ganz wegzulassen. In diesem Fall würde sich die Schleife aber nur auf die erste Instruktion auswirken. Da die erste Instruktion jedoch eine neue Variable definiert und diese nun nicht mehr Teil eines eigenständigen Blockes ist, erhalten wir auf einen Schlag gleich einen Haufen von Fehlern:

```
Teilbarkeit2.java:20: Invalid declaration.
      int einer = dummy % 10;       // Berechne einer
```



```
      ^
Teilbarkeit2.java:22: Undefined variable: einer
      if (einer != 0 &              // Ist Division moeglich?
          ^
Teilbarkeit2.java:23: Undefined variable: einer
          zahl % einer == 0)        // Ist der Rest =0 ?
                  ^
Teilbarkeit2.java:23: Incompatible type for ==. Can't convert int to boolean.
          zahl % einer == 0)        // Ist der Rest =0 ?
                       ^
Teilbarkeit2.java:25: Undefined variable: einer
                      + einer + " teilbar!");
```

Von diesen fünf Fehlern ist nur einer auf unserer Unachtsamkeit begründet; die anderen ergeben sich allesamt aus dem ersten. Deshalb ein Tip der Autoren:

Niemals einen Fehler korrigieren, wenn man sich nicht hundertprozentig sicher ist, daß es sich um keinen Folgefehler handelt. Meist läßt schon die Korrektur des ersten Fehlers eine Menge anderer Fehlermeldungen verschwinden!

3.3.6 Übungsaufgabe 2

Das Programm soll so erweitert werden, daß es auch überprüft, ob die Zahl auch durch ihre Quersumme teilbar ist?

3.4 *Aufgabe 3:* Dreierlei

3.4.1 Aufgabenstellung 3

Wegen großem Verletzungspechs muß der Trainer einer Bundesligamannschaft für ein Pokalspiel drei Nachwuchsspieler aus der Amateurmannschaft rekrutieren. Er setzt sich deshalb mit dem Betreuer der Jugendlichen zusammen, der ihm die fünf verheißungsvollsten Talente vorstellt:

„Da hätten wir als erstes Al. Al ist ein wirklich guter Stürmer, aber manchmal etwas überheblich. Sie sollte auf jeden Fall Cid einsetzen, falls Al spielt. Cid ist der ruhende Pol bei uns; er sorgt dafür, daß die Jungs auf dem Teppich bleiben. Das gilt übrigens besonders auch für Daniel! Wenn sie Daniel einsetzen, darf Cid auf keinen Fall fehlen.

A propos Daniel ... Nachdem ihm Bert seine Freundin ausgespannt hat, sind die beiden nicht gut aufeinander zu sprechen. Die beiden giften sich nur an und sollten auf keinen Fall in einer Mannschaft sein. Sollten Sie aber

trotzdem Bert wollen, so müssen Sie auf jeden Fall auch Ernst einsetzen. Ernie und Bert sind ein langjähriges Team – ihr Kombinationsspiel ist einfach traumhaft!“

Welche drei Spieler sollte der Trainer nun aufstellen?

3.4.2 Analyse des Problems 3

Wie bei jeder Textaufgabe müssen wir auch hier zuerst einmal alle relevanten Informationen herausfinden, die uns die Lösung des Problems erst ermöglichen.

1. Der Trainer braucht genau drei Spieler – nicht mehr und nicht weniger!
2. Wenn Al spielt, muß auch Cid spielen.
3. Wenn Daniel spielt, muß auch Cid spielen.
4. Bert und Daniel dürfen nicht gemeinsam spielen.
5. Ernie und Bert dürfen nur gemeinsam spielen.

Wie kann man diese Informationen nun nutzen, um ein Ergebnis zu erzielen? Die erste Möglichkeit ist, sich Papier und Bleistift zu nehmen, alle Bedingungen in einem logischen Ausdruck zusammenzufassen und diesen zu vereinfachen. Dies wollen wir aber nicht tun – wir wollen Programmieren.

Die einfachste Art, alle möglichen Lösungen zu erhalten, ist wohl simples Ausprobieren. Wir kombinieren alle Spieler miteinander und schauen, welche Kombinationen die Bedingungen erfüllen. Hierzu definieren wir pro Spieler eine boolsche Variable. Ist der Inhalt der Variable `true` bedeutet dies, daß er spielt.

3.4.3 Algorithmische Beschreibung 3

Es macht wahrscheinlich mehr Sinn, sich das Programm direkt anzuschauen, als den Algorithmus zu formulieren. Trotzdem wollen wir dies kurz tun:

Konstruiere eine Schleife, die alle fünf Spieler beliebig miteinander kombiniert. Mache dann folgende Tests

- Teste, ob genau drei der fünf Variablen wahr sind
- Teste, ob C. spielt, falls A. spielt

- Teste, ob C. spielt, falls D. spielt
- Teste, ob B. und D. nicht zusammen spielen
- Teste, ob B. und E. gemeinsam spielen (falls einer spielt).

Treffen alle fünf Tests zu, gib die Kombination aus.

3.4.4 Programmierung in Java 3

Die erste Frage, die sich stellt, ist: wie können wir alle Kombinationen der fünf Variablen erhalten? Wir verschachteln hierzu fünf `do-while`-Schleifen. Folgende Schleife würde beispielsweise die Variable `bert` hintereinander auf `true` und `false` setzen.

```
boolean bert = true;
do {
  // Hier eventuelle weitere Schleifen oder Test einfuegen
  bert = !bert;
} while (bert != true);
```

Zu Beginn der Schleife ist `bert==true`, wird aber negiert, bevor die Schleifenbedingung das erste Mal überprüft wird. Auf diese Weise wird die Schleife noch ein zweites Mal für `bert==false` durchlaufen. Bevor die Bedingung ein zweites Mal abgefragt wird, setzt die Anweisung `bert=!bert;` die Variable wieder auf `true`. Die Schelife bricht ab.

Wir wollen zuerst den Programmrumpf und die fünf verschachtelten Schleifen implementieren. Es ergibt sich folgendes Listing:

```
/**
 @author Jens Scheffler
 @version 1.0
 */

/**
 Dieses Programm loest Aufgabe 3
 */

 import Prog1Tools.IOTools;

 public class threeGuys {
   public static void main(String[] args) {
     boolean al = true;
     do {
       boolean bert = true;
       do {
         boolean cid = true;
         do {
```

```
          boolean daniel = true;
          do {
            boolean ernst = true;
            do {
              // Definiere eine Variable, die das Ergebnis der Tests enthaelt
              boolean testergebnis;
// ================================
// HIER DIE FUENF TESTS EINFUEGEN!!!
// ================================
              // Ausgabe, falls testergebnis==true
              if (testergebnis)
                System.out.println("A:" + al + "  B:" + bert + "  C:"
                                   + cid + "  D:" + daniel + "  E:"
                                   + ernst);
              ernst = !ernst;                          // negiere Variable
            }
            while (ernst != true);
            daniel = !daniel;                          // negiere Variable
          } while (daniel != true);
          cid = !cid;                                  // negiere Variable
        } while (cid != true);
        bert = !bert;                                  // negiere Variable
      } while (bert != true);
      al = !al;                                        // negiere Variable
    } while (al != true);
  }
}
```

Es stellt sich nun die Frage, wie man die verbliebenen fünf Tests am besten implementiert. Beginnen wir mit dem ersten. Wir müssen die Anzahl der Variablen herausfinden, die den Wert `true` besitzen. Dies läßt sich recht einfach wie folgt bewerkstelligen:

```
int counter = 0;
if (al) counter++;
if (bert) counter++;
if (cid) counter++;
if (daniel) counter++;
if (ernst) counter++;
```

Nach Durchlaufen der letzten Zeile steht in `counter` die gesuchte Zahl der Variablen. Das Testergebnis ergibt sich durch den Vergleich `counter==3`.

Kommen wir zum zweiten Punkt: Wenn Al spielt (`if (al)`), so muß auch Cid (`testergebnis=cid`) spielen. Wir dürfen an dieser Stelle jedoch nicht vergessen, daß das Endergebnis nur stimmt, wenn *alle* einzelnen Tests korrekt sind. Sofern wir also die Variable `testergebnis` verändern, müssen wir ihren alten Wert mit einfließen lassen:

```
if (al)
  testergebnis = testergebnis & cid;
```


Der dritte Test verläuft analog zum zweiten. Im vierten Fall können wir das Testergebnis auf `false` setzen, sofern Bert und Daniel spielen (also `bert && daniel`). Fall fünf läuft auf einen Vergleich der Inhalte von `bert` und `ernst` hinaus – diese müssen gleich sein.

Hier das komplette Listing mit allen fünf Tests:

```
import Prog1Tools.IOTools;

public class threeGuys {
  public static void main(String[] args) {
    boolean al = true;
    do {
      boolean bert = true;
      do {
        boolean cid = true;
        do {
          boolean daniel = true;
          do {
            boolean ernst = true;
            do {
              // Definiere eine Variable, die das Ergebnis enthaelt
              boolean testergebnis;

              // Test 1: Zaehle die aufgestellten Spieler
              int counter = 0;

              if (al) counter++;
              if (bert) counter++;
              if (cid) counter++;
              if (daniel) counter++;
              if (ernst) counter++;

              testergebnis = (counter == 3);

              // Test 2: Wenn A spielt, spiel auch C?
              if (al)
                testergebnis = testergebnis & cid;
              // Test 3: Wenn D spielt, spielt auch C?
              if (daniel)
                testergebnis = testergebnis & cid;
              // Test 4: Wenn B spielt, darf D nicht spielen (und umgekehrt)
              if (bert && daniel)
                testergebnis = false;
              // Test 5: Spielen B und E gemeinsam?
              testergebnergebnis = testergebnis & (bert == ernst);
              // Ausgabe, falls testergebnis==true
              if (testergebnis)
                System.out.println("A:" + al + "  B:" + bert + "  C:"
                                   + cid + "  D:" + daniel + "  E:"
                                   + ernst);
              ernst = !ernst;                          // negiere Variable
            }
```

```
            while (ernst != true);
            daniel = !daniel;                          // negiere Variable
          } while (daniel != true);
          cid = !cid;                                  // negiere Variable
        } while (cid != true);
        bert = !bert;                                  // negiere Variable
      } while (bert != true);
      al = !al;                                        // negiere Variable
    } while (al != true);
  }
}
```

Übersetzen wir das Programm und lassen es laufen, so erhalten wir folgende Ausgabe:

```
A:true  B:false  C:true  D:true  E:false
A:false  B:true  C:true  D:false  E:true
```

Der Trainer kann also entweder Al, Cid und Daniel oder aber Bert, Cid und Ernie aufstellen.

3.4.5 Vorsicht Falle 3

Noch intensiver als in der vorherigen Aufgabe arbeiten wir hier mit Blöcken, geöffneten und geschlossenen Klammern. Das Hauptproblem an dieser Stelle ist deshalb wirklich, diese zum richtigen Zeitpunkt zu öffnen und wieder zu schließen. Eine strukturierte Programmierung (Einrücken!) wirkt dabei Wunder.

3.4.6 Übungsaufgabe 3

Bert und Daniel haben sich urplötzlich wieder vertragen. Welche neuen Möglichkeiten ergeben sich für den Trainer?

4 Felder

4.1 Was sind Felder?

Bisher haben wir uns immer mit sogenannten **einfachen Datentypen** beschäftigt, also z.B. mit ganzen Zahlen (Integer) oder Buchstaben (Char). Diese ermöglichen uns, Programme zu schreiben, welche sich z.B. Zahlen merken können oder mit diesen Zahlen Berechnungen durchführen. In diesem Kapitel werden wir mit einem neuen Datentyp arbeiten: Mit Feldern — englisch **Array** — aus Daten. Ein **Feld aus Daten** bedeutet, daß viele Daten, z.B. Zahlen, zu einer Einheit zusammengefaßt werden. So ein Datenfeld kann ein- oder auch mehrdimensional sein, wie wir sehen werden. Auch wenn dies auf den ersten Blick wie eine unnötige Verkomplizierung aussieht, können diese Datenfelder das Arbeiten mit Variablen sehr vereinfachen.

Angenommen wir wollen in einem Programm speichern, wieviele Telefoneinheiten an jedem einzelnen Tag im Jahr vertelefoniert wurden. Dann brauchen wir 365 verschiedene Variablen, die jeweils eine Integer Zahl speichern. Oder wir erschaffen ein Feld (Array of Integer) mit 365 Integer Variablen, das sich viel leichter programmieren und bearbeiten läßt, als 365 einzelne Variablen — quasi ein Postfach mit 365 kleinen Fächern.

4.2 Eindimensionale Felder

Als erstes müssen wir wieder eine Variable deklarieren und das geht ganz ähnlich wie bei den „normalen“ Deklarationen:

```
int[] einheiten;
```

Die Klammern hinter der Typenbezeichnung `int` machen dem Übersetzer deutlich, daß es sich bei „einheiten“ nicht um eine einzelne Zahl handelt, die gespeichert werden soll, sondern um ein Feld. Tatsächlich wird in diesem Moment (anders als bei den „einfachen Datentypen“) eine Referenz angelegt, dazu aber später mehr.

Da der Übersetzer noch nicht weiß, wie groß das Feld werden soll (wir könnten 365 Fächer haben wollen, oder nur 10), reserviert er in diesem Moment auch noch keinen Speicherplatz für unser Feld. Dies kann er erst, wenn wir uns auf eine Länge festlegen. Nachdem wir uns einmal festgelegt haben, können wir diese Einstellung nicht mehr ändern!

Die Längenfestlegung, also die Angabe, wieviele Fächer in unserem Feld angelegt werden sollen, geschieht durch einen `new`-Befehl. Mit diesem Befehl

wird unser Array erst wirklich erschaffen. In unserem Beispiel lautet der Befehl:

```
einheiten = new int[365];
```

Damit teilen wir dem Übersetzer die Größe des Feldes mit, und dieser reserviert uns nun den Speicherplatz für 365 ganze Zahlen. Von jetzt an existiert quasi unser Postfach mit 365 Fächern.

Man kann die Deklaration und die Erschaffung in vielen Fällen auch zusammenfassen, in unserem Beispiel durch:

```
int[] einheiten = new int[365];
```

Die einzelnen Fächer unseres Postfaches sind mit einer laufenden Nummer versehen:

```
einheiten[0]
einheiten[1]
einheiten[2]
...
einheiten[364]
```

Wir können also die ersten Werte zuweisen:

```
einheiten[0] = 63;
```

Damit hätten wir dem ersten Fach des Feldes den Wert 63 zugewiesen (weil wir an Neujahr für 63 Einheiten mit Mama telefoniert haben, zum Beispiel). Alle Fächer, denen wir keinen Wert zuweisen, sind von Beginn an mit dem Wert 0 gefüllt, sofern es sich um **Integer** Variablen handelt.

Wichtig ist, daß wir nicht vergessen, daß der Computer mit 0 anfängt zu zählen, wir also in der Zelle `einheiten[0]` die erste und in der Zelle `einheiten[364]` die 365. Zahl finden. Unabhängig davon bezeichnet der Computer die „Länge“ des Feldes mit 365. Diese Länge kann man bei Bedarf auch errechnen lassen. Der Ausdruck

```
int laenge = einheiten.length;
```

weist der neuen Variable `laenge` die Zahl 365 zu. `einheiten.length` ist eine Funktion des Arrays `einheiten`, aber das müssen wir erst später verstehen, wenn es zu der Situation kommt, daß wir an einer Stelle des Programms nicht wissen, wie lang ein bestimmtes Feld ist.

Anmerkung: Es gibt noch eine weitere Art, ein Datenfeld zu initialisieren. Wenn wir nämlich bereits den Inhalt aller Zellen wissen, dann können wir diese Inhalte direkt angeben bei der Initialisierung:


```
int[] tage = {31,28,31,30,31,30,31,31,30,31,30,31};
```

Mit dieser Befehlszeile wird ein Datenfeld mit Namen `tage` vom Typ `int` deklariert, in der Größe 12 initialisiert, und die 12 Zahlen werden in den 12 Fächern gemäß ihrer Reihenfolge eingespeichert — es handelt sich um die Anzahl der Tage pro Monat. Wenn ein Programm die Anzahl der Tage im März braucht, kann es diese also nach dieser Initialisierung der Variablen `a` zuweisen mit mit:

```
a = tage[2];
```

Das war bisher recht einfach, aber wir müssen noch etwas hinter die Kulissen blicken, denn sonst werden wir die Fehlermeldungen nicht verstehen, die bei den häufig gemachten Fehlern entstehen.

4.2.1 Referenzen

Jetzt brauchen wir den Begriff der **Referenz.** Felder gehören zu den **Referenz-Datentypen** (im Gegensatz zu den **einfachen Datentypen**). Felder sind nämlich eine Vorstufe zu den Objekten, die wir später noch kennenlernen werden (dann lernen wir auch, was sie von „echten“ Objekten unterscheidet).

Was ist also ein **Referenz-Datentyp**? Ganz allgemein: Eine Referenz ist ein Verweis auf die Stelle im Speicherplatz, an der das Gesuchte gefunden werden kann. Die Referenz verweist also nur auf die Speicherstelle, ohne Aussage über den Inhalt zu machen. Erzeugt man eine Referenz durch die Deklaration einer Variablen des Referenz-Datentyps, so entsteht nur der Name dieser Referenz mit dem Wert `null`, d.h. die Referenz deutet noch auf keinen Speicher! Erst bei der **Initialisierung** mit dem `new`-Befehl wird der Speicherplatz reserviert, und statt `null` zu sein, verweist die Referenz auf diese Speicherstelle (Aus anderen Programmiersprachen ist dieses Konzept als **Zeiger** oder **pointer** bekannt, da es aber gewisse Unterschiede zu diesen gibt, vermeiden wir diesen Begriff.).

Dies klingt komplizierter, als es ist. Betrachten wir ein Beispiel zu den bisher bekannten Datentypen:

```
int zahl1;
int zahl2;
```

Dies sind Deklarationenen, wie wir sie schon lange kennen: Keine Referenzen, sondern der Speicherplatz ist bereits jetzt eindeutig (die maximale Größe eines `int` ist dem Computer bekannt - vgl. Tabelle 3 in Kapitel 2).

Dagegen sind mit

```
int[] feld_A;
int[] feld_B;
```

zwei Felder deklariert. Die Länge ist unbekannt und es existieren nur zwei Referenzen, die beide den Wert `null` tragen. Die könnten wir sogar abfragen mit der Zeile:

```
if (feld_A == null) feld_A = new int[2];
```

Wenn also feld_A noch nicht initialisiert wurde (`null` ist), dann wird dies mit dem `new`-Befehl nachgeholt.

Um feld_B auch zu initialisieren schreiben wir:

```
feld_B = new int[3];
```

Jetzt haben wir `feld_A` mit zwei Fächern für `int` Zahlen und `feld_B` mit dreien.

Eigentlich macht das für uns keinen Unterschied, wenn wir ein Programm schreiben, ob es sich um einfache Datentypen oder Referenzen handelt. Wir können ganz ähnlich damit arbeiten:

```
zahl1 = 15;
zahl2 = 25;

feld_A[0] = 10;
feld_A[1] = 20;

feld_B[0] = feld_A[0] + 30;
feld_B[1] = feld_A[0] + feld_A[1];
feld_B[2] = feld_B[0] + feld_B[1];
```

Diese Zuweisungen sehen ganz normal aus, und in dem Fach `[0]` von `feld_B` steht jetzt der Wert 40, im Fach `[1]` der Wert 30 und im Fach `[2]` der Wert 70.

Aber Vorsicht, hier lauert die Tücke der Referenzen, denn während die beiden Variablen `zahl1` und `zahl2` fest im Speicher stehen und von uns immer wieder benutzt werden können, kann es uns passieren, daß wir unsere Felder „verlieren“.

Wenn wir jetzt zuweisen:

```
feld_B = feld_A;
```


betrifft dies nur die Referenzen, nicht aber die Inhalte der Fächer! `feld_B` war eine Referenz, die bisher auf ein Feld mit 3 Fächern verwies, aber durch die Gleichsetzung mit einer anderen Referenz verweist sie nun ebenfalls auf den Speicherbereich, in dem das Feld `feld_A` mit nur zwei Fächern liegt. Wie zwei Wegweiser mit unterschiedlicher Beschriftung, die auf den gleichen Ort weisen!

Das bedeutet in diesem Beispiel:

```
System.out.println(feld_B[0]);
```

wird jetzt die Zahl 10 auf den Bildschirm befördern. Der Wert 40, den wir vorhin zugewiesen hatten, steht in einer anderen Stelle im Speicher, auf die wir jetzt keine Referenz mehr haben!

Wir hatten mit zwei `new`-Befehlen zwei Felder initialisiert, haben aber jetzt nur noch eines, allerdings zeigen zwei Referenzen darauf. Auf diese Problematik, die mit allen Referenzen, die wir später noch kennenlernen, besteht, müssen wir also beim Umgang mit Feldern achten.

Der **Garbage-Collector** sucht, während das Programm läuft, nach Speicherstellen, auf die keine Referenz mehr verweist, und gibt diesen Speicher wieder frei — er löscht also die gespeicherten Werte (Es gibt Programiersprachen, da geht das nicht automatisch, z.B. „Ansi C“ — dort muß der Programmierer den Speicherplatz explizit löschen).

4.2.2 Eine Aufgabe für eindimensionale Felder

Betrachten wir unser Beispiel für Telefoneinheiten nochmal im Zusammenhang:

```
/**
 *  Dieses Programmfragment demonstriert den Umgang mit einem
 *  eindimensionalen Feld
 */

public class Telefoneinheiten {

  public static void main(String[] uebergabe) {

    int[] einheiten;                // Deklaration
    einheiten = new int[365];       // Initialisierung

    int i;                          // eine Laufvariable
```

```
    for (i = 0; i <= 364; i++) {
      einheiten[i] = 0;             // alle Fcher auf 0 setzen
    }

    einheiten[0]   = 63;   // viele
    einheiten[17]  = 16;   // einzelne
    einheiten[117] = 18;   // Zuweisungen
    // und so weiter...

    /*
     * Jetzt sollen die Daten bearbeitet werden.
     * Wir erschaffen ein Datenfeld fr die DM-Betrge, also
     * Feld bestehend aus float Zahlen:
     */

    double[] kosten;
    kosten = new double[365];

    for (i = 0; i <= 364; i++) {        // Jedem Fach in kosten
      kosten[i] = 0.12 * einheiten[i]; // wird der DM Betrag
    }                                   // zugewiesen

    double summe = 0; // Eine Hilfsvariable, um die Gesamtkosten
                      // zu erfassen

    for (i = 0; i <= 364; i++) {  // Die Quersumme aus allen
      summe = summe + kosten[i]; // Kosten wird gebildet
    }

    /*
     * Jetzt kommt die Bildschirmausgabe der Daten:
     */

    System.out.println("Der Inhalt des Feldes 'einheiten' :");
    for (i = 0; i <= 364; i++) {
      System.out.print(einheiten[i] + " - ");
    }
    System.out.println();

    System.out.println("Der Inhalt des Feldes 'kosten' :");
    for (i = 0; i <= 364; i++) {
      System.out.print(kosten[i] + " - ");
    }
    System.out.println();

    System.out.println("Gesamtkosten: " + summe + " DM");

  } // Ende des main Blocks

} // Ende des class Blocks
```


4.2.3 Übungsaufgaben für eindimensionale Felder

1. Verändern Sie den Programmtext der Klasse Telefoneinheiten so, daß das Programm nicht mehr Tagesdaten, sondern Monatsdaten erfaßt und bearbeitet. Erweitern sie den Bereich der Datenzuweisung, so daß folgende Daten erfaßt werden:

Monat	Einheiten
Januar	125
Februar	390
März	256
April	199
Mai	334
Juni	291
Juli	68
August	71
September	401
Oktober	354
November	331
Dezember	214

2. Erweitern sie den Programmtext so, daß ein neues Feld anlegt wird, das die 3 Monate mit den höchsten Kosten erfaßt, d.h. ein Feld der Länge 3 soll die Nummern der Monate speichern, in denen die drei höchsten Beträge anfallen. Erweitern Sie die Bildschirmausgabe, so daß z.B. folgende Ausgabe erscheint:

```
Der Inhalt des Feldes 'einheiten':
125 - 390 - 256 - 199 - 334 - 291 - 68 - 71 - 401 - 354 - 331 - 214 -
Der Inhalt des Feldes 'kosten':
15.0 - 46.8 - 30.72 - 23.88 - 40.08 - 34.92 - 8.16 - 8.52 - 48.12 - 42.48 - 39.72
Gesamtkosten: 364.08 DM

Die drei teuersten Monate waren:
1.) Kalendermonat 9   mit 401 Einheiten, also 48.12 DM
2.) Kalendermonat 2   mit 390 Einheiten, also 46.8 DM
3.) Kalendermonat 10   mit 354 Einheiten, also 42.48 DM
```

   Hinweis: Sie benötigen mindestens ein Feld der Länge 3. In [0] speichern Sie dann die Nummer des Faches im Feld Einheiten, welches den höchsten Inhalt hat, in [1] die mit dem zweithöchsten, usw.

3. Erhöhen sie in einer der `for`-Schleifen den Wert, bis zu dem `i` anwachsen soll, um einen Zähler. Compilieren Sie und starten Sie das Programm. Diese Fehlermeldung wird später erklärt. Wann wäre die Fehlermeldung eigentlich schon wünschenswert gewesen und warum kommt sie erst so spät?

4. Schreiben Sie ein Programm, daß ein Array verwendet, um eine endliche Zahl von Folgengliedern zu berechnen. Die Folge sei rekursiv definiert mit:

   a_1 = Wurzel aus 2 , a_{n+1} = Wurzel aus $(2+a_n)$

   Berechnen Sie mit Ihrem Programm die ersten 30 Folgenglieder und geben Sie diese auf dem Bildschirm aus. Erklären Sie die letzten beiden Werte!

   Hinweis: Zur Berechnung einer Wurzel steht Ihnen die Funktion `Math.sqrt(x)` zur Verfügung. Beispiel zur Berechnung des ersten Folgegliedes:

   ```
   double a = Math.sqrt(2.0);
   ```

4.3 Mehrdimensionale Felder

Wir haben erst die Spitze des Eisberges erkundet bisher, da wir nur Felder aus einfachen Datentypen betrachtet haben:

```
int[]
double[]
char[]
boolean[]
```

Es besteht aber die Möglichkeit, Felder auch aus Objekten oder Feldern zu konstruieren; einfacher: wir können ein Feld deklarieren, welches in jedem Fach eine Referenz, z.B. auf ein weiteres Feld, besitzt. Erweitern wir unser Telefoneinheitenbeispiel mit dem Wunsch, die Daten über 10 Jahre zu sammeln. Dann brauchen wir 10 Felder über je 365 Tage. Diese 10 Felder lassen sich auch zu einem Feld zusammenfassen. Wir erhalten ein zweidimensionales Feld und deklarieren:

```
int[][] jahrzehnt;
```

Die Initialisierung kann in einem Schritt erfolgen:

```
jahrzehnt = new int[10][365];
```


Wieder können wir beide Schritte zusammenfassen zu:

```
int[][] jahrzehnt = new int[10][365];
```

Gemäß unseres Beispiels nehmen wir hierbei an, daß es sich um 10 gleichgroße Felder handelt. Daß dies nicht so sein muß, klären wir später.

Nun können wir mit diesem zweidimensionalen Feld arbeiten, wie man es logisch erwartet. Wenn wir ein bestimmtes Fach in unserem Feld ansprechen wollen, müssen wir die „Koordinaten“ benutzen:

```
int a = jahrzehnt[7][5];
int b = jahrzehnt[3][357];
```

Hiermit wird der neuen Variablen `a` die Anzahl der Einheiten zugewiesen, die wir am Feiertag „Heilige drei Könige“ des 8. Jahres vertelefoniert haben (Man beachte, daß die Zählung bei 0 beginnt!). Die Einheiten welchen wichtigen Tages werden b zugewiesen?

Nun beschränken sich die Möglichkeiten (natürlich) nicht auf zwei Dimensionen. Wie auch in der Mathematik können wir beliebig viele Dimensionen erschaffen, nur wird es zunehmend schwieriger, dafür vernünftige Beispiele zu finden, die dann auch noch leicht verständlich sind. Die folgende Beispielaufgabe beschränkt sich daher auf zwei Dimensionen. Der Vollständigkeit halber seien hier noch ein paar Beispiele gegeben. Überdenken Sie bitte die Struktur dieses Datenfeldes und was man darin ggf. speichern könnte:

```
byte[][][][][] produktion = new byte[7][7][7][7][7];
char[][] raetsel = new char[20][20];
String[] memo = new String[20];
```

Auch das letzte Beispiel ist ein Feld aus Objekten, denn der Datentyp String ist eigentlich ein Objekt der Klasse String. Die Zeile

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

hat damit ein weiteres Geheimnis verloren: Der `main`-Methode wird bei Start des Programmes ein Feld des Datentyps `String` übergeben. Inhalt dieses Feldes sind übrigens alle Angaben, die wir hinter dem Programmaufruf an den Interpreter übergeben. Der Programmaufruf

```
java Programm Hallo 15
```

startet die main Methode der Datei `Programm.class` und übergibt ihr ein Feld, dessen Inhalt im ersten Fach der String ”Hallo” und im zweiten ”15”, ebenfalls als String, ist. Da wir nicht wissen können, wieviele Wörter, also Strings, der Benutzer beim Programmaufruf eingibt, müßten wir jetzt die Funktion `args.length` benutzen, um damit vernünftig arbeiten zu können.

4.3.1 Eine Aufgabe für mehrdimensionale Felder

Wir wollen ein Spiel programmieren, bei dem U-Boote gesucht werden müssen. Da diese Aufgabe, wie üblich, im Teamwork erledigt wird, müssen wir nur einen Teil implementieren: Unser Programmabschnitt soll die U-Boote in einem Feld speichern. Deshalb schreiben wir nur den ersten Teil der `main`-Methode, an den dann später die implementierten Action-Sequenzen angehängt werden können. Das Programm soll ein U-Boot in einem 3-Dimensionalen Feld „verstecken“ und die letzte Ortung ausgeben. Dabei soll in dem Feld nur gespeichert werden, ob ein spezifischer Bereich des Meeres derzeit von dem U-Boot belegt ist:

```
public static void main(String[] args) {
  boolean[][][] ozean;  // Deklaration des 3-D Feldes
```

Es bietet sich an, den Datentyp `boolean` zu verwenden, da nur die Information, ob sich das Boot an dieser Stelle befindet (`true`) oder nicht (`false`), wichtig ist.

```
ozean = new boolean[100][100][100];  // Initialisierung
```

Damit wird ein Würfel aus 100x100x100 kleinen Fächern initialisiert, die alle auf `false` gesetzt sind. Es zeichnet einen guten Programmierstil aus, ebenfalls alle anderen Variablen bereits im Kopfteil des Programmes zu deklarieren (daran sind TurboPascal-Programmierer natürlich gewöhnt):

```
int i,j,k;  // Laufvariablen
```

Ebenso beinhaltet ein guter Programmierstil gerade bei mehrdimensionalen Kommentare, die den Sinn der verschiedenen Ebenen klarstellen:

```
/* Die erste Koordinate gibt die Tiefe an */

ozean[60][20][14] = true;  // Das U-Boot wurde positioniert
```

Hiermit wurde nun eines der 10.000 Fächer des Felder auf `true` gesetzt. Ein Ortungsversuch könnte folgendermaßen aussehen:

```
/* Wir senden einen Ping an der Stelle [20][15] */

for (i = 0; i < 100; i++) {
  if (ozean[i][20][15])
    System.out.println("Boot geortet!");
}
```


Man beachte die Verwendung des Datentyp `boolean` in einer `if`-Anweisung: Da der Datententyp an sich ein Wahrheitswert ist, muß keine Abfrage durch den ==-Operator stattfinden!

```
/* Wir bewegen das U-Boot um ein Feld */

ozean[60][20][14] = false;
ozean[61][20][15] = true;
```

Diese Bewegungsart können wir später, wenn wir Kenntnis von Methoden haben, noch besser gestalten. Wir wiederholen den vorherigen Ping:

```
/* Wir senden einen weiteren Ping an der Stelle [20][15] */

for (i = 0; i < 100; i++) {
  if (ozean[i][20][15])
    System.out.println("Boot geortet!");
}
```

Diesmal wird der Ortungsversuch Erfolg haben, da wir das UBoot sich bewegt hat.

Doch wir können auch den gesamten Ozean mit einer Schleifen-Konstruktion absuchen:

```
/* Wir suchen den gesamten Ozean ab */

for (j = 0; j < 100; j++) {
  for (k = 0; k < 100; k++) {
    for (i = 0; i < 100; i++) {
      if (ozean[i][j][k]) {
        System.out.println("Boot geortet bei " + j
                           + "-" + k + " Tiefe: " + i);
      }
    }
  }
}

} // Ende des main Blockes
```

Wir dieses Programm noch zu einem Spiel ausbauen, sobald wir etwas mehr Kenntnisse in Java besitzen.

4.3.2 Übungsaufgaben für mehrdimensionale Felder

1. Programmieren Sie einen Wochen-Terminkalender für einen Anwalt, der für jede Stunde der sieben Tage zwischen 8:00 und 17:00 Uhr stündlich einen `String`-Eintrag zur Verfügung stellt. Tragen Sie für 13:00 Uhr täglich das Mittagessen ein und weiterhin die Termine:

Mo 8:00 bis 12:00	Verhandlung im Fall Seese gegen Ratz (BGH)
Di 10:00 bis 11:00 Uhr	Vorlesung Programmieren 1 in Java
Mi 14:00 bis 16:00 Uhr	Tutorium bei Oliver
Do 15:00 bis 16:00 Uhr	Urteilsverkündung Seese gegen Ratz (BGH)
Fr 14:00 bis 17:00 Uhr	Rechnerübung im Rechenzentrum

Dabei seien im für mehrere Stunden andauernde Termine jeweils Referenzen auf den gleichen `String` gesetzt, um Terminveränderungen zu vereinfachen. Implementieren Sie auch eine Übersichtliche Ausgabe, die nur dann die betreffende Uhrzeit angibt, wenn der entsprechende Termin belegt ist (nutzen sie dazu die `null`-Referenz!).

Eine Beispielausgabe:

```
Montag:
8:00 Uhr Verhandlung im Fall Seese gegen Ratz (BGH)
9:00 Uhr Verhandlung im Fall Seese gegen Ratz (BGH)
10:00 Uhr Verhandlung im Fall Seese gegen Ratz (BGH)
11:00 Uhr Verhandlung im Fall Seese gegen Ratz (BGH)
13:00 Uhr Mittagessen
Dienstag:
10:00 Uhr Vorlesung Programmieren 1 in Java
13:00 Uhr Mittagessen
Mittwoch:
13:00 Uhr Mittagessen
14:00 Uhr Tutorium bei Oliver
15:00 Uhr Tutorium bei Oliver
Donnerstag:
13:00 Uhr Mittagessen
16:00 Uhr Urteilsverkndung Seese gegen Ratz (BGH)
Freitag:
13:00 Uhr Mittagessen
15:00 Uhr Rechnerbung im Rechenzentrum
16:00 Uhr Rechnerbung im Rechenzentrum
17:00 Uhr Rechnerbung im Rechenzentrum
```

2. Erweitern Sie ihren Kalender für eine Kanzlei bestehend aus drei Anwälten.

4.3.3 Mehrdimensionale Felder unterschiedlicher Größe

Wir wissen, daß mehrdimensionale Felder realisiert werden, indem ein Feld aus Feldern deklariert wird, also in jedem „Fach“ eine Referenz zu finden ist. Im vorhergehenden Abschnitt waren diese Felder alle von gleicher Größe. Da aber die Referenz allein nichts über die Größe aussagt, können wir auch unterschiedlich große Felder in einem Feld zusammenfassen:


```
int[][] pascaleDreieck;
```

Dies deklariert bekanntlich ein zweidimensionales Feld.

```
pascaleDreieck = new int[5][];
```

Hiermit wird nur die erste Dimension auch initialisiert. Jetzt kann in jedem Fach der ersten Dimension ein neues Feld initialisiert und dabei jedesmal eine andere Größe verwendet werden:

```
pascaleDreieck[0] = new int[1];
pascaleDreieck[1] = new int[2];
pascaleDreieck[2] = new int[3];
pascaleDreieck[3] = new int[4];
pascaleDreieck[4] = new int[5];
```

Wir können die Fächer des nun zweidimensionalen Feldes jetzt wie gewohnt nutzen und mit den Werten des Pascal'schen Dreiecks füllen.

Es besteht auch die Möglichkeit, durch Angabe der Feldinhalte ein solches Feld in einem Schritt zu deklarieren und zu initialisieren

```
int[][] pascaleDreieck = {{         1         },
                          {       1 , 1       },
                          {     1 , 2 , 1     },
                          {   1 , 3 , 3 , 1   },
                          { 1 , 4 , 6 , 4 , 1 }};
```

Dies initialisiert (in sehr übersichtlicher Form) die erste und zweite Dimension unseres Feldes.

In Zukunft werden wir also sämtliche Binominalkoeffizienten nicht mehr mühsam berechnen, sondern uns ein Java-Programm mit einem „dreieckigen Feld“ schreiben ...

4.3.4 Übungsaufgaben zu mehrdimensionalen Feldern unterschiedlicher Größe

1. Gesucht sind die Werte der n-ten Zeile des Pascalschen Dreiecks. Schreiben Sie also ein Programm, daß z.B. die Werte der 10. Zeile ausgibt. Kreieren sie dazu ein ein „dreieckiges“ Feld aus `int`-Werten, welches mit Hilfe einer Schleife von 1 bis n initialisiert wird. Die Werte einer Zeile berechnen sich jeweils aus der vorhergehenden aus der Summe der zwei „darüberstehenden“ Zahlen, wobei die äußeren Werte immer 1 sind. Implementieren Sie also eine Schleife, die diese Berechnung über das gesamte Feld hinweg durchführt und geben Sie schließlich die

letzte Zeile aus. Experimentieren Sie mit Ihrem Programm: Wie lange braucht der Rechner z.B. um ein Feld mit 100 oder 1000 Zeilen zu berechnen? Was ist zu den Werten zu sagen? Bis zu welcher Zeilenanzahl lassen sich korrekte Werte Berechnen?

4.4 Typische Fehler beim Umgang mit Feldern

Abschließend seien noch die häufigsten Fehlerquellen im Umgang mit Feldern zusammengestellt:

Indexfehler Die Zählung beginnt mit 0. Bei einem Feld der Länge n steht der erste Wert daher im Fach [0] und der letzte in Fach [n-1]. Dies müssen wir besonders beachten, wenn wir Felder innerhalb einer Schleife bearbeiten. Die Tücke besteht darin, daß der Übersetzer diesen Fehler nicht bemerken wird, sondern erst beim Programmablauf ein Programmabruch hervorgerufen wird. Denn erst, wenn der Zugriff auf das Fach [n] erfolgt, bricht der Interpreter ab mit: „Index out of bounds exception“, da er im Speicher zu diesem Wert keinen Bereich mehr findet.

unterlassene Initialisierung Ein Referenzdatentyp muß immer erst auch initialisiert werden, bevor er genutzt werden kann. Dies geschieht erst mit dem `new`-Befehl. Findet der Übersetzer eine Programmzeile vor diesem Befehl, die mit dem deklarierten Feld arbeitet, so bricht er ab: „Variable might not have been initialized“ Die ist besonders zu beachten, wenn mehrere Dimensionen eines Feldes in verschiedenen Zeilen initialisiert werden.

Fehler bei Deklaration und Initialisierung Die Feldgröße kann nur einmal festgelegt werden, nämlich bei der Initialisierung. Die Initialisierung durch Angabe der Werte funktioniert ausschließlich, wenn Deklaration und Initialisierung in einen Befehl zusammengefast werden, wie oben beschrieben.

Verwechseln der Koordinaten Insbesonders bei mehrdimensionalen Feldern mit variabler Länge ist es wichtig, genau festzulegen, an welcher Stelle welche Komponente folgt, sprich wie lang das Feld hier ist, sonst kommt es leicht zu den beschriebenen Indexfehlern. Eine ausführliche Kommentierung ist besonders wichtig!

Nichtbeachtung des Referenzdatentyps Die Referenz auf ein Feld kann leicht kopiert werden, nicht jedoch das Feld selbst. Dazu ist ein sogenanntes **Klonen** nötig, das erst im Bereich des objektorientierten Programmierens erläutert wird.

5 Unterprogramme

Wir wollen ein einfaches Problem lösen: die Funktion $f(x,n) = x^{2n} + n^2 - nx$ mit positivem ganzzahligen n und reellem x ist zu berechnen. Folgendes Programm tut genau dies:

```
public class Eval1 {

  public static void main(String[] args) {  // Hauptprogramm
    int n = IOTools.readInteger("n=");       // lies n ein
    double x = IOTools.readDouble("x=");     // lies x ein
    double produkt = 1.0;                    // Berechnung der Potenz x^2n
    for (int i=0; i < 2*n; i++)              // ...
      produkt = produkt * x;                 // abgeschlossen
    double f_x_n = produkt + n*n - n*x;      // Berechnung von f(x,n)
    System.out.println("f(x,n)="
                       + f_x_n);             // Ausgabe des Ergebnisses
  }
}
```

Nun wollen wir das Problem etwas verkomplizieren. Statt eines einfachen x soll der Benutzer einen Bereich angeben können — ein Intervall, in dem $f(x,n)$ wie folgt ausgewertet wird:

1. Werte f am linken Randpunkt `L` des Intervalls aus.
2. Werte f am rechten Randpunkt `R` des Intervalls aus.
3. Werte f am Mittelpunkt des Intervalls (berechnet aus `(L+R)/2`) aus.
4. Gib den Mittelwert des drei Funktionswerte aus.

Wir erweitern unser Programm entsprechend. Hierbei definieren wir für die drei Auswertungen der Funktion jeweils drei eigenständige Variablen, um sie für den späteren Gebrauch zu speichern. Das entstandene Programm sieht nun so aus:

```
public class Eval2 {

  public static void main(String[] args) {  // Hauptprogramm
    int n = IOTools.readInteger("n=");       // lies n ein
    double L = IOTools.readDouble("L=");     // lies L ein
    double R = IOTools.readDouble("R=");     // lies R ein

    double produkt = 1.0;                    // Berechnung der Potenz L^2n
    for (int i=0; i < 2*n; i++)              // ...
      produkt = produkt * L;                 // abgeschlossen
    double f_L_n = produkt + n*n - n*L;      // Berechnung von f(L,n)
```



```
    System.out.println("f(L,n)="
                       + f_L_n);             // Ausgabe des Ergebnisses

    produkt = 1.0;                           // Berechnung der Potenz R^2n
    for (int i=0; i < 2*n; i++)              // ...
      produkt = produkt * R;                 // abgeschlossen
    double f_R_n = produkt + n*n - n*R;      // Berechnung von f(R,n)
    System.out.println("f(R,n)="
                       + f_R_n);             // Ausgabe des Ergebnisses

    double M = (L + R) / 2.0;                // Berechnung des Mittelpunktes
    produkt = 1.0;                           // Berechnung der Potenz M^2n
    for (int i=0; i < 2*n; i++)              // ...
      produkt = produkt * M;                 // abgeschlossen
    double f_M_n = produkt + n*n - n*M;      // Berechnung von f(M,n)
    System.out.println("f(M,n)="
                       + f_M_n);             // Ausgabe des Ergebnisses

    double mitte = (f_L_n + f_R_n + f_M_n) / 3;  // Berechnung und
    System.out.println("Mittelwert=" + mitte);   // Ausgabe des Mittelwerts
  }
}
```

Wir sehen, daß unser neues Programm wesentlich länger und unübersichtlich geworden ist. Der Grund hierfür liegt vor allem an der sich immer wiederholenden `for`-Schleife. Leider benötigen wir diese aber für die Berechnung der Funktion f. Zu schade, daß wir diese nicht wie etwa den Sinus oder Tangens als einen eigenständigen Befehl zur Verfügung stellen können! Oder können wir dies etwa doch...?

In den folgenden Abschnitten werden wir lernen, sogenannte **Methoden** (oder auch **Routinen** genannt) zu definieren. Dies sind Unterprogramme, welche vom Hauptprogramm aufgerufen werden und auch Ergebnisse zurückliefern können. Mit ihrer Hilfe werden wir Programme schreiben, die weit komplexer als obiges Problem sind — und dennoch übersichtlicher!

5.1 Theorie und Praxis

5.1.1 Was sind Methoden?

Der Java Language Specification definiert Methoden wie folgt:

> „A **method** declares executable code that can be invoked, passing a fixed number of values as arguments.“

Was haben wir uns jedoch darunter vorzustellen?

Eigentlich gehören in Java Methoden zum objektorientierten Programmieren. Einer Klasse können in Java verschiedene Eigenschaften zugeordnet

werden, zu denen eben auch diese Form von „Unterprogrammen“ gehören. Da wir noch nicht objektorientiert programmieren, befassen wir uns nur mit einem Spezialfall. Wir werden den Begriff der Methode später jedoch auch auf Objekte erweitern.

Wir definieren in Java eine Methode stets innerhalb einer Klasse (d.h. nach der ersten sich öffnenden Klammer geschweiften) und in der Form

```
public static <Rueckgabetyp> <Methodenname> ( <Parameterliste> ) {
   // HIER WIRD DEN AUSZUFUEHRENDEN CODE EINFUEGEN
}
```

Hierbei ist

- der **Rückgabetyp** einer Methode der Typenname, den die Methode als Ergebnis zurückliefern soll. Soll die Methode wie in obigem Beispiel das Ergebnis der Funktion $f(x,n) = x^{2n} + n^2 - nx$ zurückgeben, so wäre der Rückgabe mit Sicherheit ein Gleitkommatyp — also `double` oder `float`. Soll die Methode keinen Wert zurückgeben, schreiben wir als Ergebnistyp `void`.
- der **Methodenname** ein Bezeichner, unter dem die Methode von Java erkannt werden soll. Der Methodenname darf selbstverständlich kein reserviertes Wortsymbol sein. Wir werden später auf Beispiele für die Bezeichnung von Methoden zu sprechen kommen.
- die **Parameterliste** eine Menge von Werten, welche wir der Methode übergeben. Die Deklaration eines Parameters (d.h. eines vorgegebenen Wertes) entspricht im großen und ganzen der Vereinbarung einer Variable oder eines Arrays (mit dem Unterschied, daß wir keine Initialisierungswerte angeben können). Mehrere Parameter werden durch Kommata getrennt. Zu jedem Parameternamen *muß* eine Typbezeichnung angegeben werden.

Wenn wir uns den Methodenkopf etwas genauer ansehen, erkennen wir eine große Ähnlichkeit zu folgender Zeile:

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

Ist das ein Zufall? Natürlich nicht! Tatsächlich ist die Hauptroutine, welche wir bislang immer verwendet haben, nichts anderes als eine Methode. Sie hat als Rückgabetyp `void`, also liefert sie keinen Wert als Ergebnis. Der Methodenname ist `main` und die Parameterliste besteht aus einem Feld von

Strings, welches den Namen `args` traegt. Wir sehen an dieser Stelle, daß wir das Feld auch `Nasenbaer` oder `schoener_Text` hätten nennen können — es wäre auf das Gleiche hinausgekommen.

5.1.2 Wertübergabe und -rückgabe

Wir wollen nun unsere Funktion $f(x,n) = x^{2n} + n^2 - nx$ durch eine Methode berechnen lassen. Wie haben wir diese zu programmieren?

Als erstes müssen wir uns Gedanken über den Kopf der Methode machen. Welchen Rückgabettyp hat die Methode? Welche Parameter müssen wir übergeben? Und wie sollen wir sie nur benennen?

Letztgenanntes Problem dürfte relativ schnell gelöst sein – wir nennen sie einfach `f`. Dies ist schließlich der Name der Funktion, und es handelt sich hierbei sowohl um einen Bezeichner als auch um kein reserviertes Wortsymbol. Auch der Rückgabetyp ist relativ leicht geklärt. Wir haben in unserem Programm Gleitkommawerte stets durch `double`-Zahlen kodiert und werden dies deshalb auch weiterhin tun. Der Rückgabetyp ist deshalb schlicht und ergreifend `double`.

Bezüglich der Parameterliste haben wir zwei Werte, die wir der Funktion übergeben müssen:

- einen ganzzahligen Wert `n`, welchen wir im Hauptprogramm in einer Variable vom Typ `int` abgespeichert hatten.
- eine Gleitkommazahl `x`, die wir durch einen `double`-Wert kodieren.

Wir haben somit alle Informationen zusammen, um unseren Methodenkopf zu definieren. Dieser lautet nun wie folgt:

```
public static double f(double x,int n) {
```

Wie wir nun den Funktionswert $f(x,n)$ berechnen, ist klar: auf dieselbe Weise wie in den bisherigen Programmen. Ein entsprechendes Codestück könnte etwa so aussehen:

```
double produkt = 1.0;                  // Berechnung der Potenz x^2n
for (int i=0; i < 2*n; i++)            // ...
  produkt = produkt * x;               // abgeschlossen
double ergebnis = produkt + n*n - n*x; // Berechnung von f(x,n)
```

Wie machen wir Java jedoch klar, daß in der Variable `ergebnis` nun das Ergebnis unserer Rechnung steht? Wie erkennt das Programm, daß das Ergebnis nicht etwa in `produkt` steht? Um diese Frage zu klären, besitzt die Sprache das Kommando `return`. Durch den Befehl

```
return ergebnis;
```

weisen wir Java an, die Ausführung der Methode zu beenden und den Inhalt der Variable `ergebnis` als Resultat zurückzugeben. Die Variable muß natürlich vom selben Typ wie der Ergebnistyp sein oder durch implizite Typenumwandlung in den entsprechenden Typ umwandelbar sein.

Der Befehl `return` funktioniert übrigens nicht nur mit Variablen. Auch Literale, arithmetische Ausdrücke wie `a+b` oder das Ergebnis anderer Methodenaufrufe kann mit `return` zurückgeliefert werden, wenn der Typ stimmt. Hat die Methode den Rückgabetyp `void`, so steht das Kommando `return;` für das sofortige Beenden der Methode (natürlich ohne die Rückgabe irgendeines Wertes).

Wir wollen dieses Wissen verwenden und unsere Methode ohne die Verwendung einer Variable `ergebnis` formulieren. Das Ergebnis sieht wie folgt aus:

```
public static double f(double x, int n) {
  double produkt = 1.0;                  // Berechnung der Potenz x^2n
  for (int i=0; i < 2*n; i++)            // ...
    produkt = produkt * x;               // abgeschlossen
  return produkt + n*n - n*x;            // Berechnung von f(x,n)
}
```

Natürlich kann die Berechnung eines Ergebnisses sich — abhängig vom Ergebnis verschiedener Fallunterscheidungen — aus mehr als nur *einer* festgelegten Vorgehensweise erhalten werden. Nehmen wir als Beispiel die Berechnung der Fakulät einer ganzen nichtnegativen Zahl n. Diese ist

- 1, falls $n = 0$ ist und
- $n * (n-1) * (n-2) * ... * 1$ in jedem anderen Fall.

Es ist aus diesem Grund möglich, mehr als eine return-Anweisung in einer Methode zu verwenden. Folgendes Programmstück würde beispielsweise die Fakultät berechnen:

```
public static double fakultaet(int n) {
  if (n == 0)                            // Sonderfall
    return 1;
  for (int i = n-1; i > 0; i--)          // berechne n*(n-1)*...
    n = n * i;                           // fange hierzu bei n-1 an
  return n;
}
```


Wir haben in obiger Methode zwei neue Dinge getan: wir haben mehr als eine `return`-Anweisung verwendet und wir haben den Wert des übergebenen Parameters `n` verändert. Es stellt sich für uns jedoch die Frage, ob wir dies eigentlich auch *dürfen*. Was ist, falls das Hauptprogramm den in `n` gespeicherten Wert noch benötigt? Dürfen wir ihn so einfach überschreiben?

Um zu verstehen, daß wir an dieser Stelle keinen Fehler begehen, muß uns zuerst klarwerden, wie Java Werte an Methoden übergibt. Wie ein Architekt, der seine wertvollen Entwürfe im Safe verstaut, gibt auch Java niemals die originale Variable preis. Sie erstellt vielmehr eine *Kopie* des Inhalts und übergibt diese an die aufgerufene Methode. Wenn wir also in der Methode `fakultaet` den Inhalt der Variable `n` verändern, verändern wir lediglich die Kopie – nicht das Original. Wir wollen dies an folgendem kleinen Beispielprogramm verdeutlichen:

```
public class AufrufTest {
  // UNTERPROGRAMM(E)
  public static void unterprogramm(int n) {
    n = n * 5;                           // veraendere den Parameter
    System.out.println("n=" + n);  // gib diesen aus
  }
  // UNSER HAUPTPROGRAMM
  public static void main(String[] args) {
    int n = 7;                           // Startwert fuer n
    System.out.println("n= " + n); // gib diesen aus
    unterprogramm(n);                    // Unterprogramm
    System.out.println("n= " + n); // gib n erneut aus
  }
}
```

Wir übersetzen das Programm und starten es. Hierbei erhalten wir folgende Ausgabe:

```
n= 7
n=35
n= 7
```

Wie wir sehen, hat der Aufruf der Methode den Inhalt der Variable n nicht verändert; wir brauchen uns also keine Sorgen zu machen, daß irgendwelche „namensgleichen“ Variablen oder übergebene Parameter sich gegenseitig beeinflussen.

5.1.3 Vorsicht Falle!

Wir ändern unser Programm leicht ab und testen es mit einem Array als Parameter:

```
public class AufrufTest2 {
  // UNTERPROGRAMM(E)
  public static void unterprogramm(int[] n) {
    n[0] = n[0] * 5;                    // veraendere den Parameter
    System.out.println("n[0]=" + n[0]);  // gib diesen aus
  }
  // UNSER HAUPTPROGRAMM
  public static void main(String[] args) {
    int n[] = {7};                      // Startwert fuer n[0]
    System.out.println("n[0]= " + n[0]); // gib diesen aus
    unterprogramm(n);                   // Unterprogramm
    System.out.println("n[0]=" + n[0]);  // gib n erneut aus
  }
}
```

Wir haben die Integer-Variable `n` durch ein eindimensionales Feld der Länge 1 ersetzt und dieses mit dem Wert `7` initialisiert. Wir geben den Inhalt des Feldes einmal aus und starten das Unterprogramm. Dieses ändert den Inhalt seines Parameters und gibt den neuen Wert auf dem Bildschirm aus. Wir beenden das Unterprogramm und geben den Inhalt des Arrays `n` erneut auf den Bildschirm. Da Unterprogramme wie gesagt mit Kopien der Originalwerte arbeiten, erwarten wir dieselbe Ausgabe wie im letzten Abschnitt. Zu unserem Erstaunen erhalten wir jedoch

```
n[0]= 7
n[0]=35
n[0]=35
```

Was ist geschehen? Um das unerwartete Ergebnis zu verstehen, müssen wir uns ins Gedächtnis rufen, daß Arrays sogenannte **Referenzdatentypen** sind. Dies bedeutet, sie verweisen lediglich auf eine Stelle im Speicher, in dem die eigentlichen Werte abgelegt sind. Wie bei den einfachen Datentypen erstellt Java auch bei Arrays eine Kopie des originalen Wertes — dieser ist jedoch nicht das eigentliche Feld, sondern besagte *Referenz*. Unsere Kopie enthält somit lediglich einen neuen Verweis, der jedoch auf ein und dasselbe Feld von Zahlen zeigt: Wir erhalten eine sogenannte **flache Kopie.** Als wir in der Methode `unterprogramm` also den Inhalt des Feldes verändert haben, auf welches `n` zeigt, haben wir in Wirklichkeit mit den originalen Werten gearbeitet (und nicht mit einer Kopie). Diese Situation wird üblicherweise als **Seiteneffekt** bezeichnet, da „neben“ der eigentlich beabsichtigten Wirkungen noch weiter Effekte sich auswirken.

Wenn wir also die Kopie eines Arrays erstellen wollen, muessen wir dies „von Hand“ tun, wie etwa in folgender Methode:

```
public static int[] arraycopy(int[] n) {
```



```
  int[] ergebnis =                  // erzeuge ein neues Feld
    new int[n.length];              // derselben Laenge wie n
  for (int i=0; i < n.length; i++)  // kopiere alle Feldelemente
    ergebnis[i] = n[i];             // in das neue Feld
  return ergebnis;
}
```

Die Methode `arraycopy` erstellt ein neues Feld mit dem Namen `ergebnis`, welches mit dem `new`-Operator auf dieselbe Länge wie das als Parameter übergebene Feld gesetzt wird. Die Länge des Feldes erfahren wir über den Wert von `n.length`. Eine anschließende Schleife kopiert Komponente für Komponente von einem Array in das andere. Wie viele andere Dinge ist in Java übrigens auch das Kopieren eines Arrays in verallgemeinerter Form bereits vordefiniert; um die Methode `System.arraycopy` jedoch richtig verwenden zu können fehlt uns momentan noch das Wissen über objektorientiertes Programmieren.

Wir wollen unser Programm nun so verändern, daß der Aufruf des Unterprogrammes den Inhalt unseres Feldes `n` nicht beeinflußt. Hierzu bauen wir die Methode `arraycopy` in unsere Klasse ein und verwenden sie, um eine Kopie des Feldes zu erzeugen. Wir ersetzen im Hauptprogramm den Aufruf des Unterprogrammes durch folgende zwei Zeilen:

```
int kopie[] = arraycopy(n);        // erzeuge eine Kopie von n
unterprogramm(kopie);              // Unterprogramm
```

Wir erzeugen also *selbst* ein neues Feld, eine Kopie von `n`, und übergeben diese statt des Originals beim Aufruf unserer Methode. Da wir mit der Kopie nicht weiter arbeiten wollen, können wir uns übrigens die Vereinbarung einer Variable namens `kopie` ersparen und das Resultat von `arraycopy` direkt als Parameter übergeben:

```
public class AufrufTest3 {
  // UNTERPROGRAMM(E)
  public static void unterprogramm(int[] n) {
    n[0] = n[0] * 5;                    // veraendere den Parameter
    System.out.println("n[0]=" + n[0]);  // gib diesen aus
  }
  public static int[] arraycopy(int[] n) {
    int[] ergebnis =                    // erzeuge ein neues Feld
      new int[n.length];                // derselben Laenge wie n
    for (int i=0; i < n.length; i++)    // kopiere alle Feldelemente
      ergebnis[i] = n[i];               // in das neue Feld
    return ergebnis;
  }
  // UNSER HAUPTPROGRAMM
  public static void main(String[] args) {
```

```
    int n[] = {7};                      // Startwert fuer n[0]
    System.out.println("n[0]= " + n[0]); // gib diesen aus
    unterprogramm(arraycopy(n));        // Unterprogramm
    System.out.println("n[0]= " + n[0]); // gib n erneut aus
  }
}
```

Übersetzen wir nun unser Programm und lassen dieses laufen, so erhalten wir wie gewünscht als Ergebnis

```
n[0]= 7
n[0]=35
n[0]= 7
```

Wie wir sehen, haben wir auf diese Weise keine wechselseitige Beeinflussung von Originalwerten und den manipulierten Parametern mehr. Wir sehen aber auch, daß bei der Übergabe von Arrays Vorsicht geboten ist — wenn man vergisst, die Parameter zu kopieren, kann ein syntaktisch vollkommen korrektes Programm völlig falsche Ergebnisse liefern. Eine Alternative wäre es somit, daß alle Methoden, die Arrays als Parameter haben und diese verändern, das Kopieren selbst übernehmen. Versuchen Sie es und ändern Sie obiges Programm so ab, daß der simple Aufruf `unterprogramm(n)` ebenfalls zum richtigen Ergebnis führt.

5.1.4 Zusammenfassung 5.1

Anhand eines einfachen Beispiels haben wir feststellen müssen, daß auch kleine Probleme sehr schnell unübersichtlich und schwer zu programmieren werden können. Wir haben deshalb Methoden kennengelernt, mit deren Hilfe wir Programme in sinnvolle Teilabschnitte untergliedern konnten. Wir haben gelernt, daß Java seine Variablen gegen sogenannte Seiteneffekte schützt (so bezeichnet man die Beeinflussung von Variablen durch Verwendung als Parameter), indem er bei der Parameterübergabe stets nur mit Kopien arbeitet. Wir haben aber auch gesehen, daß dieses System bei Feldern versagen kann.

5.1.5 Übungsaufgaben 5.1

1. Schreiben Sie eine Methode, die den Tangens einer `double`-Zahl berechnet, welche als Parameter übergeben wird. Implementieren Sie den Tangens gemäß der Formel $tan(x) = sin(x)/cos(x)$. Sie dürfen die Methoden `Math.sin` und `Math.cos` zur Berechnung von Sinus und Cosinus verwenden. Was sie jedoch nicht dürfen, ist innerhalb der Methode auch nur eine einzige Variable zu definieren.

2. Schreiben Sie eine Methode `swap`, die den Inhalt eines eindimensionalen Arrays `a` vom Typ `int[]` „spiegelt“, das heißt das erste Element von `a` ist das letzte Element von `swap(a)` und so weiter. Hierbei dürfen keine Seiteneffekte auftreten, das heißt der Inhalt von `a` soll unverändert bleiben.

   Schreiben Sie eine weitere Methode `swap2`, welche dieselbe Funktion hat, aber als Rückgabetyp `void` besitzt. Hierzu sollen bewußt Seiteneffekte eingesetzt werden, d.h. das Ergebnis soll am Ende der Methode in `a` selbst stehen.

5.2 Rekursiv definierte Methoden

5.2.1 Motivation

Wir haben bislang gelernt, wie man Methoden definiert, wie man mit ihnen Ergebnisse berechnet und diese zurückgibt. Hierbei haben wir festgestellt, daß der Aufruf einer selbstdefinierten Methode so einfach ist wie etwa der Start von `System.out.println` oder der Sinusfunktion `Math.sin`. Wir haben auch gesehen, daß bei der Übergabe der Parameter diese auch wieder Ergebnis einer Methode sein können; so geschehen etwa in der Zeile

```
unterprogramm(arraycopy(n));       // Unterprogramm
```

in unserem letzten Programm. Wir wissen auch, daß unsere Hauptroutine `main` selbst wieder eine Methode ist, daß also Methoden wieder andere Methoden aufrufen können. Wie weit kann dieses Spielchen jedoch genau gehen? Können Methoden sich etwa auch *selbst* aufrufen?

Um diese Frage zu klären, formulieren wir ein kleines Testprogramm.

```
public class Unendlichkeit {
  // UNTERPROGRAMM(E)
  public static void unterprogramm() {
    System.out.println("Unterprogramm aufgerufen...");
    unterprogramm();           // rufe dich selbst auf
  }
  // UNSER HAUPTPROGRAMM
  public static void main(String[] args) {
    unterprogramm();
  }
}
```

Der einzige Sinn unserer Hauptroutine ist der Aufruf der Methode `unterprogramm`. Diese gibt eine Meldung auf dem Bildschirm aus und ruft danach die Methode `unterprogramm` auf. Diese gibt eine Meldung auf dem Bildschirm aus und ruft danach die Methode `unterprogramm` auf. Diese gibt dann eine...

Wie wir sehen, haben wir uns eine einfache Endlosschleife gebastelt; das Programm wird niemals beendet. Wird der Compiler dies erkennen? Wir übersetzen nun das Programm und erhalten keine Fehlermeldung – schließlich ist es syntaktisch vollkommen korrekt. Starten wir es auf dem Rechner, so erhalten wir wie erwartet die Ausgabe

```
Unterprogramm aufgerufen...
Unterprogramm aufgerufen...
Unterprogramm aufgerufen...
Unterprogramm aufgerufen...
Unterprogramm aufgerufen...
Unterprogramm aufgerufen...
Unterprogramm aufgerufen...
Unterprogramm aufgerufen...
Unterprogramm aufgerufen...
Unterprogramm aufgerufen...
...
```

Die Ausgabe endet erst, wenn wir das Programm mit `Strg+C` abbrechen. Es stellt sich natürlich die Frage, warum Java etwas derartiges nicht verbietet. Warum können Methoden sich selbst aufrufen, wenn man damit eine Endlosschleife produzieren kann?

Die Antwort liegt in dem letzten Wort der Frage: man *kann* eine Endlosschleife produzieren, *muß* dies aber nicht. Tatsächlich sind sogenannte **rekursive Methoden** oftmals der einfachste Weg, eine Problemstellung zu lösen. Wir erinnern uns etwa an unsere Methode `fakultaet`, in welcher wir ueber eine Fallunterscheidung und eine `for`-Schleife zum Ergebnis gekommen sind:

```
public static double fakultaet(int n) {
  if (n == 0)                          // Sonderfall
    return 1;
  for (int i = n-1; i > 0; i--)        // berechne n*(n-1)*...
    n = n * i;                         // fange hierzu bei n-1 an
  return n;
}
```

Mit Hilfe einer rekursiven Definition hätten wir uns eine Menge Gedankenarbeit sparen koennen; aus $n! = n*(n-1)*(n-2)*...*1 = n*(n-1)!$ können wir nämlich folgern, daß die Fakultät von n

- $= 1$ ist, falls $n = 0$ gilt und
- $= n*(n-1)!$, falls dem nicht so ist.

Diese Erkenntnis läßt sich sehr schön in eine rekursive Methode einbauen:

```
public static int fakultaet(int n) {
  if (n == 0)
    return 1;                 // Am Ende der Rekursion angekommen?
  return n * fakultaet(n-1);  // Wenn nicht, dann rechne weiter...
}
```

Ein Vergleich mit obigem Programmstück zeigt, um wieviel einfacher rekursiv definierte Methoden gestrickt sein können. Wie überall im Leben erkauft man sich hiermit natürlich auch einige Nachteile:

- Wir müssen aufpassen, daß unsere Methode **terminiert**, d.h. daß sich die Methode nicht unendlich oft aufruft. So etwas kann auch dem professionellsten Programmierer passieren, wenn er seinen Algorithmus schlampig formuliert. Dieser Nachteil wiegt aber nicht sonderlich schwer, denn bei einer `for`-Schleife kann man sich ebenfalls einen solchen Fehler einhandeln.
- Rekursiv definierte Methoden sind im allgemeinen etwas langsamer als Methoden, in denen das Problem ohne Rekursion gelöst wurde. Dies liegt daran, daß jeder Aufruf eines Unterprogramms den Computer etwas Rechenzeit kostet. Bei Programmen in der Größenordnung, die wir schreiben, bedeutet das im schlimmsten Fall einige Hundertstelsekunden.

Rekursiv definierte Methoden werden in den verschiedensten Gebieten angewandt; so wird etwa der bekannte **Quicksort-**Algorithmus, welcher eine Menge von Zahlen sortiert, im allgemeinen rekursiv definiert. Wir werden im folgenden Abschnitt ein Beispiel behandeln, in dem uns ein solcher Programmierstil von Nutzen ist.

5.2.2 Das Achtdamenproblem

5.2.2.1 Aufgabenstellung Auch wenn nicht jeder ein Großmeister des Schach sein düfte, haben wir doch wohl schon alle von diesem Spiel gehört. Wir nehmen also ein Schachbrett heran, welches bekanntlich $8*8 = 64$ Felder besitzt. Auf diesem Brett wollen wir acht Damen so verteilen, daß keine die andere schlagen kann.

Zur Lösung dieses Problems werden wir spaltenweise „von links nach rechts“ vorangehen, d.h. wir setzen unsere erste Dame in die linke Spalte. Da eine Dame senkrecht, waagerecht und diagonal schlagen kann, darf in dieser Spalte nun keine Dame mehr stehen. Wir setzen unsere nächste Dame deshalb in die nächste Zeile (und so weiter). Stehen die bereits gesetzten Damen so ungünstig, daß wir keine weitere mehr setzen können, gehen wir einfach einen Schritt zurück und versetzen die letzte Dame. Kann diese nicht versetzt werden, gehen wir wieder einen Schritt zurück und so weiter.

5.2.2.2 Lösungsidee Unsere Vorgehensweise ist also schlichtes Ausprobieren. Wir teilen den Algorithmus in drei Teilbereiche auf:

- eine Methode `bedroht`, in der wir überprüfen, ob die zuletzt gesetzte Dame im Zugbereich einer der anderen Damen steht.
- eine Methode `ausgabe`, in der wir die einmal gefundene Lösung auf dem Bildschirm ausgeben und
- eine Methode `setze`, mit der wir versuchen, die Damen an den richtigen Stellen zu plazieren.

Um die Positionen der einzelnen Damen zu speichern, definieren wir ein eindimensionales Feld der Länge 8 mit Namen `brett`. Jeder Eintrag des Feldes steht für die Position einer Dame. So steht etwa `feld[0]` für die Zeilennummer der Dame in der ersten Reihe, `feld[1]` für die Nummer der Dame in der zweiten Reihe und so weiter. Falls also etwa `feld[0` ist, so steht eine Dame in der linken oberen Ecke.

5.2.2.3 erste Vorarbeit: die Methoden `ausgabe` und `bedroht` Wir wollen uns als erstes an die Formulierung der Methode `ausgabe` machen, da diese am wenigsten Arbeit erfordert. Um eine halbwegs übersichtliche Ausgabe zu gewährleisten, wollen wir das Brett zeilenweise wie folgt auf den Bildschirm drucken:

- Steht im Feld `(i,j)` eine Dame, d.h. ist `brett[j]==i`, so drucke ein `D` auf dem Bildschirm
- Ist dem nicht so, gibt ein Leerzeichen aus.

Um später das Ergebnis leichter überprüfen zu können, trennen wir die einzelnen Spalten durch einen Balken. Die Ausgabe ist leicht durch zwei geschachtelte `for`-Schleifen zu bewerkstelligen. Folgende Methode liefert das gewünschte Ergebnis:

```
public static void ausgabe(int[] brett) {
  for (int i=0; i < 8; i++) {      // Anzahl der Zeilen
    for (int j=0; j < 8; j++)      // Anzahl der Spalten
      System.out.print("|" + ((i == brett[j]) ? 'D' : ' '));
    System.out.println("|");       // Zeilenende
  }
}
```

Wir verwenden hierbei den ternären Operator `?:`, um uns eine `if`-Abfrage zu ersparen.

Als nächstes gehen wir die Umsetzung der Methode `bedroht` an. Diese soll einen boolschen Wert zurückliefern, und zwar `true`, falls eine Bedrohung der zuletzt gesetzten Dame vorliegt. Wir müssen der Methode also neben dem Feld auch die Nummer der aktuellen Spalte als Parameter übergeben. Wir tun dies mit einem ganzzahligen Parameter namens `spalte`.

Um herauszufinden, ob die in der aktuellen Spalte gesetzte Dame durch eine andere Dame bedroht wird, müssen wir drei Tests durchführen:

1. Befindet sich in derselben Zeile noch eine andere Dame, die also waagerecht schlagen könnte? Dies wäre der Fall, wenn wir in einem vorherigen Schritt bereits eines der Elemente von `brett` auf dieselbe Zahl gesetzt haben. Es gäbe also eine Zahl `i` zwischen 0 und `spalte`, für die `brett[i]==brett[spalte]` gilt. Wir können den Test wie folgt formulieren:

   ```
   for (int i=0; i < spalte; i++)
     if (brett[i] == brett[spalte])
       return true;
   ```

2. Befindet sich in der Diagonale, die schräg nach oben links verläuft, eine Dame? Dieser Test ist nicht ganz so einfach, da wir die Testbedingung nicht so leicht wie oben angeben können.

   Wir überlegen uns deshalb an einem Beispiel, wie die Schleife auszusehen hat. Angenommen, wir befinden uns in der dritten Spalte (also `spalte==2`, da wir von der Null aus zählen) und setzen die Dame auf die fünfte Zeile (also `brett[2]==4`). Genau dann befindet sich eine Dame in der Diagonale, wenn `brett[1]==3` oder `brett[0]==2` ist. Wir

müssen also sowohl bei der Spaltenzahl als auch bei der zu überprüfenden Zeilennummer jeweils um den Wert `1` heruntergehen. Wir führen neben der Laufvariablen `i` deshalb noch eine Variable `j` ein, in der wir für jede zu prüfende Spalte die zugehörige Zeilennummer speichern. Unsere Überprüfung funktioniert nun wie folgt:

```
for (int i = spalte-1, j = brett[spalte]-1; i >= 0; i--,j--)
  if (brett[i] == j)
    return true;
```

3. Befindet sich in der Diagonale, die schräg nach unten links verläuft, eine Dame? Die Überprüfung dieser Bedingung funktioniert genau wie die andere Diagonalrichtung — mit dem Unterschied, daß wir die Variable `j` nun erhöhen statt erniedrigen müssen:

```
for (int i = spalte-1, j = brett[spalte]+1; i >= 0; i--,j++)
  if (brett[i] == j)
    return true;
```

Hat eine Situation auf dem Spielbrett alle drei Tests überstanden, so ist die zu testende Dame nicht bedroht; wir können also den Wert `false` zurückgeben. Unsere Methode sieht nun wie folgt aus:

```
public static boolean bedroht(int[] brett,int spalte) {
  // Teste als erstes, ob eine Dame in derselben Zeile steht
  for (int i=0; i < spalte; i++)
    if (brett[i] == brett[spalte])
      return true;

  // Teste nun, ob in der oberen Diagonale eine Dame steht
  for (int i = spalte-1, j = brett[spalte]-1; i >= 0; i--,j--)
    if (brett[i] == j)
      return true;

  // Teste, ob in der unteren Diagonale eine Dame steht
  for (int i = spalte-1, j = brett[spalte]+1; i >= 0; i--,j++)
    if (brett[i] == j)
      return true;

  // Wenn das Programm hier angekommen ist, steht die Dame "frei"
  return false;
}
```

5.2.2.4 die Rekursion Wir kommen nun zur letzten und allem Anschein nach schwierigsten Methode: der Methode `setze`. Wir haben uns bereits überlegt, wie der Algorithmus auszusehen hat. Wir beginnen bei `spalte=0` und setzen die Dame in die Zeile 0. Als nächstes setzen wir die

Dame in der zweiten Zeile auf das erste Feld, welches nicht besetzt ist (und so weiter). Gibt es keine Möglichkeit mehr, eine Dame zu setzen, gehen wir wieder eine Spalte zurück und versuchen, die letzte gesetzte Dame auf einen anderen Platz zu bringen. Gibt es hierfür wieder keine Möglichkeit, gehen wir wieder eine Spalte zurück...

Wie wir sehen, ist der Algorithmus ziemlich kompliziert. Die einzelnen Spalten beeinflussen sich gegenseitig und wir wissen nicht im Voraus, bis zu welcher Zeile wir etwa die Suche in der fünften Spalte durchzuführen haben. Das Resultat ist eine Verschachtelung von mindestens *acht* `for`-Schleifen mit diversen `break`-Anweisungen, bei denen man sehr leicht den Überblick verliert. Geht das nicht auch einfacher?

Dank rekursiver Programmiertechnik können wir diese Frage mit reinem Gewissen bejahen. Wenn wir uns die Vorgehensweise nämlich etwas genauer betrachten, so stellen wir eine Struktur fest, die für alle Spalten gleich ist:

1. Beginne in der ersten Zeile und setze `brett[spalte]=0`.
2. Wird die neu gesetzte Dame `bedroht`, versuche es eine Zeile tiefer.
3. Steht die neu gesetzte Dame frei, beginne wieder von vorne für die nächste Spalte.
4. Hat die Suche dort keinen Erfolg, gehe wieder zum Schritt 2. Hatte die Suche Erfolg, sind wir fertig.
5. Sind wir erfolglos bei der achten Spalte angekommen, stecken wir in einer „Sackgasse“. Leite dies Resultat wieder eine Ebene tiefer.

Wir sehen nicht nur, daß die Suche nach der passenden Dame für alle Spalten gleich aufgebaut ist; wir erkennen vielmehr auch, daß sich die „Kommunikation“ zwischen den einzelnen Spaltensuchen auf ein einfaches `true` (= die Suche war erfolgreich) bzw. `false` (= die Suche war nicht erflogreich) zurückführen läßt. Einen einzelnen boolschen Wert kann man wiederum sehr bequem von einer Methode zurückgeben lassen.

Wir definieren unsere Methode `setze` zuerst einmal so, als wollten wir die Lösung nur für eine ganz bestimmte Spalte suchen. Wir benötigen als Parameter also die Nummer der Spalte, in der wir suchen, und das Feld, in dem wir setzen sollen. Der Rückgabewert ist (wie oben gefordert) ein boolscher Wert:

```
public static boolean setze(int[] brett, int spalte) {
```

Wir wollen nun überlegen, wie wir obige fünf Schritte am besten in ein Java-Programm kleiden. „Beginne in der ersten Zeile“ und „versuche es eine Zeile tiefer“ — das klingt verdächtig nach einer Schleife! Wir formulieren also eine `for`-Schleife, die über die einzelnen Zeilennummern läuft:

```
for (int i=0; i < 8; i++) {
  brett[spalte] = i;          // Probiere jede Stelle aus
  if (bedroht(brett,spalte))  // Falls die Dame nicht frei steht
    continue;                 // versuche es an der naechsten Stelle
```

Nun haben wir innerhalb der Schleife also ein `i` gefunden, an der die Dame von keiner anderen bedroht wird. Was sagte obiger Algorithmus noch für diesen Fall? „Beginne wieder von vorne für die nächste Spalte“. Wir können also durch den rekursiven Aufruf `setze(brett,spalte+1)` das Programm anweisen, dieselben Schritte auch für die nächste Spalte durchzuführen. Wir brauchen hierbei keine Kopie des Feldes zu übergeben, da die Methode ja nur rechts von uns Veränderungen vornimmt, die uns eben nicht interessieren. Da wir natürlich auch wissen wollen, ob die Suche erfolgreich war, müssen wir das Ergebnis der Methode in einer Variable sichern:

```
boolean success =
   setze(brett,spalte+1);
```

Ist der Inhalt der Variablen `true`, so haben wir Erfolg gehabt und können unsere Suche beenden (und damit auch die Methode):

```
if (success)
  return true;
```

Andernfalls müssen wir unsere Dame weiter verschieben, also die Schleife weiterhin ausführen. Sind wir am Ende der Schleife angekommen, d.h. es gibt keine weiteren Kombinationen mehr, so stecken wir in einer Sackgasse. Der Rückgabewert ist somit `false`.

Wir haben nun alle Voraussetzungen, eine Lösung unseres Problems zu finden. Hiermit sind wir jedoch noch nicht fertig. Zwei Fragen bleiben (noch) unbeantwortet:

- Woran erkennen wir, ob wir eine Lösung gefunden haben oder noch weiterrechnen müssen?
- Terminiert unsere Methode? Haben wir uns auch wirklich keine Endlosschleife geschaffen?

Wir beschäftigen uns vorerst mit der ersten Frage, denn die zweite wird sich dann von selbst beantworten. Wir haben eine Lösung gefunden, wenn wir insgesamt acht Damen auf das Feld gesetzt haben, d.h. wenn `spalte==7` und `bedroht(brett,spalte)==false`. Wir könnten diese Abfrage in unsere `for-Schleife` einbauen, müssen dies aber nicht. Falls wir nämlich acht Damen gesetzt haben, die sich gegenseitig nicht bedrohen, wird in unserer Schleife die Methode `setze` mit dem Parameter `spalte==8` ein weiteres Mal aufgerufen. Es reicht also eine einzige Abfrage zu Anfang unserer Methode:

```
public static boolean setze(int[] brett, int spalte) {
  // Sind wir fertig?
  if (spalte == 8) {
    ausgabe(brett);
    return true;
  }

  // Suche die richtige Position fuer die neue Dame
  for (int i=0; i < 8; i++) {
    brett[spalte] = i;          // Probiere jede Stelle aus
    if (bedroht(brett,spalte)) // Falls die Dame nicht frei steht
      continue;                 // versuche es an der naechsten Stelle
    boolean success =           // moeglicher Kandidat gefunden? --
      setze(brett,spalte+1);    // teste noch die folgenden Spalten
    if (success)                // falls es geklappt hat
      return true;
  }

  // Wenn das Programm hier angekommt, stecken wir in einer Sackgasse
  return false;
}
```

Unsere fertige Methode terminiert selbstverständlich, denn die `for`-Schleifen gehen alle nur bis `i==7` und die Methode ruft sich selbst maximal achtmal hintereinander auf. Selbst wenn für jede Kombination ein Aufruf stattfinden würde (was wegen der Methode `bedroht` nicht geschieht), würde die Methode also allerhöchstens $1 + 8^8 = 16777217$ Male aufgerufen – aber natürlich ist es bei weitem nicht so viel.

5.2.2.5 Zu guter letzt...

Wir haben jetzt also eine rekursive Methode definiert, die acht Damen wie gefordert auf dem Schachbrett plaziert und die Lösung ausgibt. Sind wir damit fertig? Haben wir noch etwas vergessen?

Vor lauter Methoden und rekursivem Aufruf haben wir noch nicht daran gedacht, daß wir unser Feld noch gar nicht vereinbart haben. Auch müssen wir die Berechnung natürlich irgendwie „anstoßen“, d.h. einen ersten Aufruf von `setze` mit leerem Schachbrett und `spalte=0` ausführen. Für diese Dinge soll bei uns die Hauptroutine zuständig sein:

```
public static void main(String[] args) {
  int[] feld = {0,0,0,0,0,0,0,0}; // Initialisiere das Spielfeld
  setze(feld,0);                  // Starte die Suche am linken Rand
}
```

Wir haben nun ein Programm geschrieben, das unser gestelltes Achtdamenproblem löst (natürlich nur, sofern wir keine Fehler gemacht haben). Wir speichern das Programm in einer Datei `Achtdamen.java` ab, übersetzen es und lassen es laufen. Wir erhalten folgendes Ergebnis:

```
|D| | | | | | | |
| | | | | |D| | |
| | | | |D| | | |
| | | | | | | |D|
| |D| | | | | | |
| | | |D| | | | |
| | | | | |D| | |
| | |D| | | | | |
```

Man kann relativ leicht nachprüfen, daß obige Schachbrettkonstellation natürlich nicht die einzige mögliche Lösung ist (man muß das Brett hierzu lediglich um 90 Grad drehen). Wir wollten aber schließlich auch nur *eine* und nicht *alle* Lösungen. Diese zu erhalten ist Teil der folgenden Übungsaufgaben.

Hier noch einmal das gesamte Programm im Überblick:

```
public class Achtdamen {


  /** Testet, ob eine der Damen eine andere schlagen kann. */
  //  ------------------------------------------------------
  public static boolean bedroht(int[] brett, int spalte) {
    // Teste als erstes, ob eine Dame in derselben Zeile steht
    for (int i=0; i < spalte; i++)
      if (brett[i] == brett[spalte])
        return true;

    // Teste nun, ob in der oberen Diagonale eine Dame steht
    for (int i = spalte-1, j = brett[spalte]-1; i >= 0; i--,j--)
      if (brett[i] == j)
        return true;

    // Teste, ob in der unteren Diagonale eine Dame steht
    for (int i = spalte-1, j = brett[spalte]+1; i >= 0; i--,j++)
      if (brett[i] == j)
        return true;

    // Wenn das Programm hier angekommen ist, steht die Dame "frei"
    return false;
  }
```



```
  /** Sucht rekursiv eine Loesung des Problems. */
  //  ------------------------------------------
  public static boolean setze(int[] brett, int spalte) {
    // Sind wir fertig?
    if (spalte == 8) {
      ausgabe(brett);
      return true;
    }

    // Suche die richtige Position fuer die neue Dame
    for (int i=0; i < 8; i++) {
      brett[spalte] = i;          // Probiere jede Stelle aus
      if (bedroht(brett,spalte)) // Falls die Dame nicht frei steht
        continue;                 // versuche es an der naechsten Stelle
      boolean success =           // moeglicher Kandidat gefunden? --
        setze(brett,spalte+1);    // teste noch die folgenden Spalten
      if (success)                // falls es geklappt hat
        return true;
    }

    // Wenn das Programm hier angekommt, stecken wir in einer Sackgasse
    return false;
  }


  /** Gibt das Schachbrett auf dem Bildschirm aus. */
  //  ---------------------------------------------
  public static void ausgabe(int[] brett) {
    for (int i=0; i < 8; i++) {      // Anzahl der Zeilen
      for (int j=0; j < 8; j++)      // Anzahl der Spalten
        System.out.print("|" + ((i == brett[j]) ? 'D' : ' '));
      System.out.println("|");       // Zeilenende
    }
  }


  /** Initialisiert das Schachbrett und ruft die Methode setze auf */
  //  -------------------------------------------------------------
  public static void main(String[] args) {
    int[] feld = {0,0,0,0,0,0,0,0}; // Initialisiere das Spielfeld
    setze(feld,0);                  // Starte die Suche am linken Rand
  }
}
```

5.2.3 Zusammenfassung 5.2

Wir haben in diesem Abschnitt rekursiv definierte Methoden kennengelernt und sie verwendet, um das Achtdamenproblem mit vergleichsweise wenig Aufwand zu lösen. Hierbei haben wir erkannt, daß rekursiv definierte Methoden ein Problem oftmals sehr viel einfacher formulierbar machen. Wir

haben jedoch auch gelernt, daß der Teufel oftmals im Detail steckt. Der Compiler kann beispielsweise nicht von selbst erkennen, ob der von uns beschriebene Algorithmus tatsächlich auch terminiert ...

5.2.4 Übungsaufgaben 5.2

1. Zurück zum Achtdamenproblem. Modifizieren Sie die Methode `setze`, indem Sie die Zeile

   ```
   setze(brett,spalte+1);   // teste noch die folgenden Spalten
   ```

   durch die Zeile

   ```
   setze(brett,++spalte);   // teste noch die folgenden Spalten
   ```

   ersetzen. Liefert das Programm jetzt noch eine Lösung? Versuchen Sie, die Antwort *ohne* den Rechner zu finden.

   Machen Sie die Ersetzung rückgängig und verändern Sie das Programm so, daß es *alle* Lösungen des Problems findet. Ein kleiner Tip: Sie müssen dazu nur eine einzige Programmzeile verändern.

5.3 Sonstiges

5.3.1 Die Methode `main`

Wir haben bereits erfahren, daß unsere Hauptroutine, die Routine `main`, nach demselben Schema wie jede andere Methode aufgebaut ist. Ihr Rückgabetyp ist `void`, das heißt sie liefert kein Ergebnis zurück. Einziger Parameter ist ein eindimensionales Feld vom Typ `String`, welchem wir bislang den Namen `args` gegeben haben. Eine Sache haben wir bislang jedoch noch nicht geklärt: Was *steht* in diesem Array überhaupt?!?

Um diesen Punkt besser verstehen zu können, erinnern wir uns für einen Moment daran, wie wir unsere Programme bislang aufgerufen haben. Hieß unsere Klasse beispielsweise `schoeneKlasse`, so haben wir dies mit

```
java schoeneKlasse
```

getan. Nun kann es jedoch sein, daß wir unserem Programm selbst irgendwelche Parameter auf den Weg geben wollen. Dieser Fall ist gar nicht so ungewöhnlich und wir kennen ihn aus unserem Betriebssystem:

- Der Befehl `cp d1 d2` kopiert in Unix den Inhalt der Datei `d1` in die Datei `d2`. Wir geben diesen Befehl in einer Zeile ein und werden nicht etwa vom Programm selbst zu einer Eingabe aufgefordert. `d1` und `d2` sind also solche Parameter.
- Der Befehl `copy d1 d2` hat dieselbe Funktion in MS-DOS oder Windows 95/98. Auch hier sind wieder `d1` und `d2` Parameter.

Nun kann man natürlich argumentieren, daß diese Befehle keine Java-Programme sind. Wir halten jedoch dagegen, daß auch der `cp`-Befehl oder der `copy`-Befehl irgendwann einmal in einer Programmiersprache geschrieben worden ist. Außerdem müssen Parameter ja nicht unbedingt Dateinamen sein. Falls wir beispielsweise später einmal mit Grafiken arbeiten, möchten wir auf diese Weise villeicht die Größe eines zu zeichnenden Fensters oder die Hintergrundfarbe angeben. Wir *brauchen* also Parameter.

Natürlich läßt Java den Benutzer an dieser Stelle nicht im Stich — Sie können sich schon denken, wo die Parameter in Java abgespeichert werden. Die Antwort liegt auf der Hand: in dem Feld `args`, welches der Methode `main` übergeben wird. Die Länge des Feldes entspricht der Anzahl der übergebenen Werte (bislang war dies also immer 0). Wir wollen diesen Umstand anhand eines kurzen Beispielprogrammes verdeutlichen. Wir schreiben eine Klasse `GrussWort`, dem wir beim Aufruf den Vor- und Nachnamen als Parameter übergeben:

```
java GrussWort Manfred Mustermann
```

Dieses Programm soll folgendes ausgeben:

```
Hallo, Manfred!
Mustermann ist aber ein schoener Nachname :-)
```

Hierbei versteht sich von selbst, daß Vor- und Nachname von den Parametern abhängen. Da wir nun wissen, wie Parameter an ein Programm übergeben werden, erkennen wir

- daß der Vorname im ersten Element des Feldes (also `args[0]`) steht und
- daß der Nachname in `args[1]` gespeichert ist.

Wir können obiges Programm also sehr einfach schreiben:

```
public class GrussWort {
  public static void main(String[] args) {
    System.out.println("Hallo, " + args[0] + "!");
    System.out.println(args[1] + " ist aber ein schoener Nachname :-)");
  }
}
```

Wir übersetzen das Programm, starten es mit `java GrussWort` — und erhalten die Fehlermeldung

```
java.lang.ArrayIndexOutOfBoundsException: 0
        at GrussWort.main(GrussWort.java:3)
```

Was ist passiert? Wir haben „vergessen“, dem Programm die erwarteten zwei Parameter zu übergeben; das Feld `args` hat also die Länge `0`. Wenn wir versuchen, irgendein Element aus dem Feld zu lesen, schießen wir also automatisch über das Ziel hinaus. Das Programm bricht mit einer Fehlermeldung ab. Dasselbe passiert übrigens auch, wenn wir dem Programm nur einen Parameter übergeben (nur eine Zeile tiefer, nach der ersten Ausgabe).

Starten wir das Programm aber wie gefordert mit

```
java GrussWort Manfred Mustermann
```

so erhalten wir auch die gewünschten zwei Zeilen Hierbei ist es egal, ob wir mehr als die zwei geforderten Parameter anhängen; das Programm greift nur auf `args[0]` und `args[1]` zurück und schenkt den übrigen keinerlei Beachtung.

5.3.2 Die Klasse `java.lang.Math`

Wer bislang alle Übungsaufgaben gerechnet hat, wird in diesem Abschnitt schon einmal auf die Methoden `Math.sin` und `Math.cos` gestoßen sein, mit welchen Sinus und Cosinus einer Zahl berechnet werden sollten.

Diese Methoden gehören zu einem größeren Klasse mit dem Namen `java.lang.Math`, welche (ähnlich wie etwa die `IOTools`) eine Anzahl von vordefinierten Methoden zur Verfügung stellt. Die Klasse gehört zum Package `java.lang`, welches vom System beim Übersetzen automatisch eingebunden wird. Wir können also die Methoden verwenden, ohne sie zuvor mit eine `import`-Anweisung bekanntmachen zu müssen. So gibt also beispielsweise die Anweisung

```
System.out.println(Math.sin(1.3));
```


Name	Argumentzahl	Typ	Kurzbeschreibung
`abs`	1	`double`	Betrag eines Wertes
`abs`	1	`long`	Betrag eines Wertes
`abs`	1	`int`	Betrag eines Wertes
`acos`	1	`double`	Arcus Cosinus
`asin`	1	`double`	Arcus Sinus
`atan`	1	`doube`	Arcus Tangens
`ceil`	1	`double`	„Runde ganzzahlig auf“
`cos`	1	`double`	Cosinus
`exp`	1	`double`	E-Funktion
`floor`	1	`double`	„Runde ganzzahlig ab“
`log`	1	`double`	Logarithmus zur Basis e
`max`	2	`double`	Maximum zweier Werte
`max`	2	`long`	Maximum zweier Werte
`max`	2	`int`	Maximum zweier Werte
`min`	2	`double`	Minimum zweier Werte
`min`	2	`long`	Minimum zweier Werte
`min`	2	`int`	Minimum zweier Werte
`pow`	2	`double`	Potenzfunktion „a hoch b“
`random`	0	`double`	Zufallswert zwischen 0 und 1
`round`	1	`double`	„Runde zur nächsten Ganzzahl“
`sin`	1	`double`	Sinus
`sqrt`	1	`double`	Quadratwurzel
`tan`	1	`double`	Tangens

Tabelle 14: Einige Methoden der Klasse Math

den Sinus von `1.3` auf dem Bildschirm aus. Die folgende Tabelle faßt die wichtigsten Methoden der Klasse zusammen.

Wie ist die Tabelle nun zu lesen? Angenommen wir haben eine Zahl `x` vom Typ `double` und wollen die Wurzel dieser Zahl bestimmen. In diesem Fall finden wir in der Tabelle eine Methode `sqrt`, welche ein Argument vom Typ `double` benötigt und besagte Wurzel berechnet. Der Ergebnistyp entspricht hier immer auch dem Typ des Arguments. Wir erhalten die Wurzel also durch folgenden Aufruf:

```
double wurzel = Math.sqrt(x);
```

Nun wollen wir die erhaltene Wurzel zweimal quadrieren – also mit der Zahl vier Potenzieren (d.h. also „d hoch 4“) berechnen. Wir schlagen in der Tabelle nach und finden die Methode `pow`. Unser Aufruf sieht nun wie folgt aus:

```
double doppelQuadrat = Math.pow(wurzel,4);
```

Es sollte an dieser Stelle darauf hingewiesen werden, daß die Klasse `Math` mehr als nur die in der Tabelle aufgeführten Methoden besitzt. Um Details über diese zu erfahren wird ein Blick in die sogenannte API-Spezifikation empfohlen. Diese von Sun mit `javadoc` erstellten HTML-Seiten beschreiben den Aufbau jeder einzelnen Klasse, die im Java standardmäßig enthalten ist.

5.3.3 Zusammenfassung 5.3

Wir haben uns mit einer speziellen Methoden beschäftigt, die wir auch schon *vor* diesem Kapitel gekannt und verwendet haben. Die Methode `main`, deren Definition für uns bislang eher „schwarze Magie“ war, liegt in ihrem Aufbau, Sinn und Zweck nun offen vor uns. Wir haben unsere Kenntnisse sogar erweitert und wissen nun, wie wir einem Java-Programm selbst Parameter auf den Weg geben können.

5.3.4 Übungsaufgaben 5.3

1. Erweitern Sie das Grußwortprogramm so, daß es

   - bei der Eingabe von 0 Parametern den Satz „Bist Du stumm?“ ausgibt.
   - bei der Eingabe von einem Parameter grüßt und dann nach dem Nachnamen fragt.
   - bei der Eingabe von mehr als einem Parameter von einem doppelten bzw. mehrfachen Vornamen ausgeht (wie beispielsweise in Karl Hedwig Mustermann).

2. Nehmen Sie ein paar Ihrer vorigen Übungsprogramme zur Hand und schreiben sie ein kurzes Menü, mit dem sie diese starten können. Verwenden Sie hierzu die `IOTools`, um den Benutzer eine Zahl zwischen eins und drei eingeben zu lassen. Starten Sie bei der Zahl eins das Achtdamenproblem, bei der zwei das Grußwortprogramm mit dem Namen *Gustav Gustavson* und bei drei ein weiteres Programm ihrer Wahl.

Hinweis: Wollen Sie beispielsweise das Achtdamenproblem starten und haben sie die Klasse wie im Text `Achtdamen` genannt, so müssen Sie lediglich die Hauptroutine dieser Klasse aufrufen. In unserem Beispiel geschieht das etwa durch den Aufruf

```
Achtdamen.main(args);
```

wobei `args` ein beliebiges Feld von Zeichenketten ist (beispielsweise das Feld `args`, welches der Hauptroutine ihres neuen Menüs übergeben wurde).

A JDK-Installation unter Windows 95/98

Diese Anleitung ist bewußt einfach gehalten und auf den wohl häufigsten Fall zugeschnitten:

- Ein PC mit dem Betriebssystem Windows95 oder Windows98 soll mit
- kostenloser Software auf das Programmieren in Java vorbereitet werden.

A.1 Schritt 1 — Bezug der Software

Wer keinen Internet-Zugang vom heimischen PC installiert hat, kann sich die notwendige Software im Rechenzentrum auf Disketten kopieren. Dazu werden 8 formatierte und leere Disketten benötigt. Vorsicht, die Disketten müssen absolut fehlerfrei sein, daher sollte man lieber ein paar mehr mitnehmen!

Im Poolraum -111 oder -112 im RZ im Kursaccount, also mit `waifb###` einloggen.

netscape & [**Enter**] Damit wird ein zweites Fenster für den Netscape Browser gestartet und die Homepage von „Programmieren 1 — Java“ angezeigt.

Fenster maximieren Durch Anklicken des Knopfes mit dem großen Kästchen ganz rechts in der Kopfzeile des Netscape-Fensters, füllt der Browser jetzt den gesamten Bildschirm.

Auf der linken Seite im Inhaltsverzeichnis den Link „Info und Links zu Java“ anklicken. Damit wird die rechte Seite an die richtige Stelle versetzt. Unter „JDK1.1.x für Zuhause“ finden sich alle wichtigen Links.

Den Link „JDK1.1.5 für Windows95“ anklicken. Damit öffnet sich ein Fenster „Save as...“ und es kann hier einfach „[OK]“ angeklickt werden. Ein kleines Fenster zeigt uns wieder den Fortschritt des „Netscape Download“ mit einem kleinen blauen Balken, der sich langsam vorarbeitet.

In diesem Moment wird eine sehr große Datei in das Verzeichnis `.netscape` geladen.

Achtung! Hierbei kann ein Fehler auftreten, nämlich in der Art, daß die vom Rechenzentrum zur Verfügung gestellte Speicherkapazität für

alle Kursaccounts überschritten wird („Disc-Quota exceeded“). Das passiert, wenn zu viele Personen zugleich den JDK in ihr Kursaccount kopieren und nach der Bearbeitung nicht mehr löschen. Wenn dem so ist, hilft nur abwarten und am nächsten Tag wieder probieren...

Den Link „Programmers File Editor — ein Editor für Windows95“ anklicken. Damit öffnet sich ein weiteres Fenster „Save as...“ und wieder genügt „[OK]“ anzuklicken.

Netscape beenden Mit dem Schalter am linken Ende der Kopfzeile des Fensters mit dem Kreuz schließt sich das Fenster mit dem Browser, und es kann im x-term Fenster weitergearbeitet werden.

cd .netscape [**Enter**] Damit wird in das Netscape-Unterverzeichnis gewechselt, in das Netscape gerade eben alle Downloads abgelegt hat.

split -b 1457664 jdk115-win32.exe [**Enter**] Damit wird die große Datei in kleine Häppchen zerstückelt, die jetzt auf Disketten kopiert werden können.

Auch hierbei kann es zu dem Problem kommen, daß die „Disk-Quota exceeded“, quasi die Festplatte voll ist. In dem Fall müssen die Downloads gelöscht werden (mit dem nächsten Befehl), damit der Platz wieder freigegeben wird und die ganze Aktion ein andermal neu gestartet werden kann.

rm jdk115-win32.exe [**Enter**] Diese Datei wird damit gelöscht. Das muß noch bestätigt werden mit

y [**Enter**]

Leere Diskette einlegen

mwrite xaa a: [**Enter**] Damit wird die erste Teildatei `xaa` auf die Diskette kopiert. Kommt es zu einer Fehlermeldung („Disk Full“), ist die Diskette voll oder defekt — dann muß die nächste Diskette her, denn im AB-Pool kann nicht formatiert werden.

rm xaa [**Enter**]

y [**Enter**] Auch diese Datei kann wieder gelöscht werden.

Dieses Kopieren auf Disketten und Löschen muß nun noch sechsmal wiederholt werden:

Leere Diskette einlegen

mwrite xab a: [**Enter**]

rm xab [**Enter**]

y [**Enter**]

usw. mit den Dateien `xac`, `xad`, `xae`, `xaf`, `xag`

Leere Diskette einlegen

mwrite pfe.exe a: [**Enter**] Jetzt wird auch der Editor auf die Diskette kopiert.

rm pfe.exe [**Enter**]

y [**Enter**] Jetzt sind alle Dateien wieder gelöscht, die unnötig Platz belegen würden.

A.2 Schritt 2 — Wiederherstellen der Software zuhause

Zuhause müssen wir die JDK-Dateien nun wieder zusammenführen. Dazu öffnen wir ein MS-DOS Fenster: „Start"-Taste klicken, „Programme" anwählen und „MS-DOS Eingabeaufforderung" starten. Es öffnet sich ein Fenster mit einem „DOS-Prompt", ähnlich dem „x-term" Fenster im RZ.

cd [**Enter**] Sprung in das Hauptverzeichnis (meist `c:\` genannt).

md neu [**Enter**] Ein neues Unterverzeichnis mit dem Namen `neu` wird angelegt.

cd neu [**Enter**] Wechsel in das neue Verzeichnis.

Jetzt müssen alle Disketten aus dem RZ eingelegt werden und jeweils mit

copy a:*.* c:\neu [**Enter**] auf die Festplatte kopiert werden. Dieser Befehl muß also 8x eingegeben werden — das geht aber auch, indem man einfach die „F3" Taste drückt.

Falls im Prompt kein `C`, sondern ein anderer (Laufwerks-) Buchstabe steht, muß dieser verwendet werden!

Vor dem Wechseln der Diskette immer das Erlöschen des Lichts am Laufwerk abwarten!

Der Befehl

dir [**Enter**] zeigt jetzt die Dateien `xaa`, `xab`, `xac`, `xad`, `xae`, `xaf` und `pfe.ZIP` an.

copy /b xaa+xab+xac+xad+xae+xaf+xag jdk.exe [**Enter**] Die Datei `jdk.exe` wird aus den Einzelteilen wieder zusammengefügt.

Wer Zuhause Internet-Zugang hat, kann sich die beiden Dateien natürlich direkt in das Verzeichnis `c:\neu` kopieren. Die Datei `jdk115~1.exe` kann mit

rename jdk115~1.exe jdk.exe umbenannt werden, damit im Folgenden wieder die gleichen Dateinamen verwendet werden können.

A.3 Schritt 3 — Das Java Development Kit (JDK) installieren

In dem MS-DOS Fenster kann jetzt das Installationsprogramm aufgerufen werden:

jdk [**Enter**] Ein Fenster fragt noch mal nach, ob der JDK installiert werden soll.

[**OK**] Nach kurzer Vorbereitung startet das Installations-Setup. Das Fenster bittet, alle anderen Anwendungen zu schließen. Das MS-DOS Fenster stört nicht, also:

[**Next** >] Unter Umständen fragt das Setup hier, ob der Winsock 2 installiert werden soll.

[**Ja**] Falls es hier zu einer Fehlermeldung kommt, ist die Installation des Winsock 2 nicht nötig.

[**OK**] Eine Lizenzvereinbarung erscheint, der nach dem Lesen zugestimmt werden muß:

[**Yes**] Ein Fenster fragt, welche Komponenten installiert werden sollen. Wer genügend Platz auf der Festplatte hat, (siehe „Space available") sollte die

Checkbox „Java Sources" ebenfalls anklicken. Um die Standard Verzeichnisse anzulegen:

[**Next** >] Eine Zusammenfassung erscheint.

[**Next** >] Das Setup-Programm beginnt Dateien zu kopieren.

Schließlich erscheint (hoffentlich) eine Meldung über die Erfolgreiche Installation.

[**Finish**] Eine Readme-Datei wird angezeigt, die viele wichtige Informationen beinhaltet, die aber den Anfänger im Moment nicht interessieren.

Editor schließen Das Readme file verschwindet, das MS-DOS Fenster ist wieder sichtbar:

del x* [**Enter**]

del jdk.exe [**Enter**] Die nicht mehr benötigten Dateien sind damit gelöscht.

cd.. [**Enter**] Wechsel in das Hauptverzeichnis.

edit autoexec.bat [**Enter**] Ein Editor-Fenster mit der Datei `autoexec.bat` wird geöffnet. Sofern hier bereits eine Zeile steht, die mit „path...“ beginnt, kann man an diese die Zeichenkette `;c:\jdk1.1.5\bin` hinten anfügen, ansonsten muß

path c:\jdk1.1.5\bin als letzte Zeile in diese Datei geschrieben werden.

Editor schließen. Die Frage, ob die geänderte Datei gespeichert werden soll, natürlich mit „[Ja]“ beantworten.

autoexec.bat [**Enter**] Die Datei wird ausgeführt, wie in Zukunft bei jedem Start des Rechners. Die „path“ Variable läßt das Betriebssystem zukünftig immer im `jdk1.1.5\bin` Verzeichnis suchen, wenn der Compiler o.ä. gebraucht wird.

A.4 Schritt 4 — Den Editor PFE installieren

Wieder im MS-DOS Fenster muß der PFE ausgepackt werden:

move pfe.exe c:\jdk1.1.5 [**Enter**]

c:\jdk1.1.5\pfe [**Enter**] Die Datei, die den Editor installiert wird in das Java-Verzeichnis verlagert und die Dateien werden dort installiert. Die `pfe.exe`-Datei kann jetzt gelöscht werden, ebenso das `neu`-Verzeichnis:

del c:\jdk1.1.5\pfe.exe [**Enter**]

rmdir neu [**Enter**]

DOS-Fenster schließen. Jetzt kann man eine Verknüpfung auf den Desktop anlegen, mit der der PFE direkt aufgerufen wird:

Irgendwo auf dem Desktop mit der rechten Maustaste klicken.

„NEU“ wählen und dann „Verknüpfung“ Es erscheint ein Fenster, in das unter „Befehlszeile“ die Position unseres PFE eingegeben wird, also:

c:\jdk1.1.5\pfe32.exe und dann

„Weiter“ anklicken. Ein neues Fenster fragt nach dem Namen, der dieser Verknüpfung zugewiesen werden soll und macht einen Vorschlag. Hier kann man aber auch etwas netteres eingeben, z.B.:

„Java Editor“ oder „Programmieren 1“ Nun entsteht auf dem Desktop ein Icon, das beim Anklicken den File Editor startet.

A.5 Schritt 5 — Den Editor PFE nutzen

In der Knopfleiste des Editors stehen Funktionen wie Neue Datei, Laden, Speichern, Kopieren, Einfügen usw. zur Verfügung.

Zum Aufrufen des javac Compilers bietet sich die Funktion „Dos Command and capture output“ an, die auch über die Taste „[F11]“ erreicht wird. Dann wird nämlich ein neues Fenster erzeugt, das z.B. die Fehlermeldungen des Compilers enthält.

Durch einen Knopf in der Knopfleiste wird ein MS-DOS Fenster gestartet, in dem man die fertigen Programme dann mit dem „`java` *Dateiname*“ Kommando testen kann.

Im Menü „Options“ unter „Preferences“ in der „Category“ „File Filters“ kann die Anzeige von Java Dateien einstellen. Im Bereich „Filter String“ gibt man `*.java` und im Bereich „Description“ z.B. „JAVA Quellcode“ ein und klickt „add at top“ an. Danach werden beim Datei öffnen nur Java-Dateien angezeigt.

B Die Klasse IOTools

B.1 Tastatureingaben in Java

Java ist eine komplexe Sprache — eine Sprache, in der man wahrscheinlich fast alles Programmieren kann, wenn man nur weiß wie.

Um den großen Umfang an Funktionalität und Plattformunabhängigkeit zur Verfügung zu stellen (welchen die Sprache nun einmal hat), haben die Entwickler viele Dinge abstrahiert. Diese wurden daraufhin in so allgemeiner Form implementiert, daß insbesondere Anfänger große Schwierigkeiten haben, sie zu benutzen. Leider ist die Eingabe eine davon.

Java erhält Eingaben von Daten über sogenannte *Ströme* (engl.: streams). Man kann sich diese Ströme am besten wie einen altmodischen Nachrichtenticker vorstellen. Auf einem schmalen Streifen kommen Nachrichten an, Zeichen für Zeichen aneinandergereiht. Welche Daten und Informationen auf diesen Streifen stehen, ist dem Gerät dabei egal — es handelt sich nur um eine Ansammlung von Zeichen. Es obliegt dem Leser, die Streifen an der richtigen Stelle abzureißen und die Daten auf dem Streifen zu interpretieren.

So oder so ähnlich können wir uns auch die Ströme in Java veranschaulichen. Ein Strom besteht aus nichts weiter als einer Ansammlung von Bits, die durch das Programm in irgendeiner Form interpretiert werden müssen. Hierbei kann die Quelle dieser Zeichen verschieden sein: eine Datei auf der Festplatte, ein Dokument aus dem Internet, oder eben die Eingabe von der Tastatur. Die Methodenaufrufe, welche dem Benutzer zur Verfügung stehen, sind in allen Fällen gleich.

Haben wir es jedoch geschafft dem Computer klarzumachen, daß wir als Eingabestrom die Tastatur verwenden wollen, so sind wir damit noch lange nicht am Ziel. Das Beste, was wir dem PC mit den vordefinierten Methoden nämlich entlocken können, ist eine Kette von Zeichen — ein `String`. Wir wollen im allgemeinen jedoch keine Strings, sondern `int`-Werte, `double`-Zahlen oder eventuell einzelne Zeichen. Zwar gibt es Möglichkeiten, aus unserem `String` diese zu extrahieren; hierfür brauchen wir jedoch meistens Wissen, das über das eines Anfängers hinausgeht. Um dem Abhilfe zu schaffen, wurden die `IOTools` geschrieben.

B.2 Installation von Prog1Tools.zip

Auf den Kursrechnern im AB-Pool sind die `IOTools` bereits installiert. Möchten Sie die Klasse auch daheim unter Windows 95 verwenden, gehen Sie wie folgt vor:

1. Finden Sie heraus, wo sich auf Ihrem Rechner die Datei `classes.zip` befindet und schreiben Sie sich den zugehörigen Pfad auf (Hinweis: die Datei begindet sich üblicherweise im Unterverzeichnis `lib` des Verzeichnisses, in dem Sie Java installiert haben. Wir werden im folgenden von

   ```
   c:\jdk1.1.5\lib\classes.zip
   ```

   ausgehen. Ersetzen Sie diese Zeile nach Bedarf.

2. Besorgen Sie sich die Datei `Prog1Tools.zip` und speichern Sie sie auf Ihrer Festplatte. Schreiben Sie sich auch diesen Pfad auf. Wir gehen einmal davon aus, daß die Position der `zip`-Datei

   ```
   c:\programme\java\other\Prog1Tools.zip
   ```

   ist.

3. Öffnen Sie nun die Datei `autoexec.bat` im Verzeichnis `c:\` mit einem Editor. Fügen Sie die Zeile

   ```
   set CLASSPATH=.;c:\programme\java\other\Prog1Tools.zip;c:\jdk1.1.5\lib\classes.zip
   ```

   ein. Nun können Sie wie gewohnt mit `javac` kompilieren, ohne auf spezielle Parameter achten zu müssen. Wollen Sie die Klasse `IOTools` verwenden, müssen Sie in die erste Zeile ihres Programms nur noch

   ```
   import Prog1Tools.IOTools;
   ```

   einfügen.

B.3 Anwendung der IOTools-Methoden

Folgende Methoden sind unter anderem definiert:

- Die Methode `readInteger` liest eine Zahl vom Typ `int` von der Tastatur ein und gibt diese als Ergebnis zurück. Um beispielsweise zwei ganze Zahlen von der Tastatur einzulesen und in den Variablen `a` und `b` zu sichern, genügt folgendes Programmstück:

  ```
  int a = IOTools.readInteger();
  int b = IOTools.readInteger();
  ```

- Die Methode `readDouble` liest eine Zahl vom Typ `double` ein. Obiges Beispiel würde also für double-Zahlen wie folgt aussehen:

```
double a = IOTools.readDouble();
double b = IOTools.readDouble();
```

- Die Methode `readLong` liest eine Zahl vom Typ `long` ein. Die Methoden `readShort` und `readFloat` tun dies für die Datentypen `short` und `float`.

- Die Methode `readLine` liest eine ganze Textzeile (abgeschlossen durch den Druck auf die Eingabetaste).

- Die Methode `readString` liest ein einzelnes „Textwort“ von der Tastatur. Ein Textwort besteht aus einem String, welches weder durch Leer- noch Tabulator- bzw. Zeilenendezeichen auseinandergerissen ist. Geben wir beispielsweise die Zeile

```
Dies ist eine schoene Zeile.
```

  ein und rufen den Befehl `readString` auf, so liefert er lediglich `Dies` als Ergebnis. Um an das nächste Wort zu gelangen, muß die Methode erneut aufgerufen werden.

- Die Methode `readChar` liest ein einzelnes Zeichen, welches nicht gleich dem Leerzeichen, Zeilenendezeichen oder dem Tabulatorzeichen ist. Die Methode basiert hierbei auf der `readString`-Methode, das heißt es werden Textworte eingelesen und in ihre einzelnen Komponenten aufgespalten. Das Programmstück

```
IOTools.readChar();
char a = IOTools.readChar();
int b = IOTools.readInteger();
```

  liefert bei der Eingabe `abc123 456` also `a='b'` und `b=456`, da die Ziffern `123` noch zum ersten Textwort gehören.

- Die Methode `readBoolean` liest einen boolschen Wert ein. Hierbei ist auf Groß- und Kleinschreibung zu achten; die Eingabe `True` kodiert beispielsweise *keinen* Wert vom Typ `bool`. Es muß vielmehr `true` heißen.

Wie wir in obigen Beispielen gesehen haben, können auch mehr als eine einzulesene Information pro Zeile eingegeben werden (man muß sie lediglich durch Leerzeilen trennen). Hierbei muß man natürlich auf die Reihenfolge der Eingaben achten. Der Befehl `readInteger` wird bei der Eingabe

```
Ich gebe jetzt einmal 13 ein.
```

als Ergebnis den Wert `13` zurückgeben, da dies die erste gültige Ganzzahl ist. Die zuvor stehenden Textworte werden verworfen.

UDO6

Dieser Text wurde erzeugt mit
UDO
Release 6 Patchlevel 11
TOS

Dirk Hagedorn
Postfach 8105
D-59840 Sundern
E-Mail: DHagedorn@t-online.de
UDO ist ein Programm, welches Textdateien, die im Universal Document Format erstellt wurden, in das ASCII-, ST-Guide-, LaTeX-, Rich Text-, Pure-C-Help-, Manualpage-, HTML-, WinHelp-, Texinfo-, Linuxdoc-SGML-, LyX-, Apple-QuickView- und Turbo-Vision-Help-Format umwandeln kann.
Weitere Informationen sowie die aktuellen Versionen findet man im World Wide Web unter
http://members.aol.com/DirkHage

Index