10 Klassen

Wie wir gesehen haben, dienen Arrays dazu, mehrere Datenelemente zu einer größeren Einheit zu gruppieren. Klassen erfüllen einen ähnlichen Zweck: Sie sind wie Arrays selbst definierte Datentypen, die es erlauben, mehrere Elemente zu einem neuen Objekt zusammenzufassen und unter einem gemeinsamen Namen anzusprechen. Im Gegensatz zu Arrays, die aus lauter gleichartigen Elementen bestehen, können Klassen aber aus verschiedenartigen Elementen aufgebaut sein.

10.1 Deklaration und Verwendung

Beginnen wir am besten mit einem Beispiel: Angenommen wir wollen ein Datum verarbeiten, das aus einem Tag, einem Monatsnamen und einer Jahreszahl besteht, z.B. 13. November 2006. Um es abzuspeichern, brauchen wir drei Variablen:

```
int day;
String month;
int year;
```

Wenn wir nun allerdings `n` Exemplare eines Datums benötigen, müssen wir `n` mal drei Variablen deklarieren, was nicht nur schreibaufwendig ist, sondern uns auch bei der Namensgebung in Verlegenheit bringt, weil ja jede Variable einen anderen Namen bekommen muss.

Hier kommen Klassen ins Spiel. Mit Klassen können wir mehrere Variablen wie `day`, `month` und `year` zu einem neuen Typ zusammenfassen, den wir z.B. `Date` nennen und von dem wir beliebig viele Exemplare deklarieren können.

Deklaration von Klassen

Eine Klasse `Date` wird in Java folgendermaßen deklariert:

```
class Date {
    int day;
    String month;
    int year;
}
```

Die Deklaration beginnt mit dem Schlüsselwort class. Darauf folgt der Name der Klasse und eine Liste von Variablendeklarationen in geschwungenen Klammern. Die Variablen in einer Klasse nennt man *Felder* oder *Attribute*. Klassen werden auf der äußersten Ebene einer Datei deklariert, also

```
class Date {...}          // my classes
class Time {…}
...
class MyProg {...}        // my main program
```

Wie man sieht, ist auch MyProg eine Klasse, allerdings eine, die auch Methoden enthält. Klassen mit Methoden werden wir uns in Kapitel 11 näher ansehen. Einstweilen betrachten wir Klassen nur als Sammlung von Daten.

Verwendung von Klassen

Wir können nun die Klasse Date wie jeden anderen Datentyp verwenden, um Variablen zu deklarieren:

```
Date x, y;
```

Die beiden Variablen x und y sind vom Typ Date und bestehen jeweils aus den Feldern day, month und year. Man kann auf diese Felder zugreifen, indem man sie durch einen Punkt vom Namen der Variablen trennt, zu der sie gehören (man *qualifiziert* die Feldnamen mit dem Variablennamen):

```
x.day = 13;
x.month = "November";
x.year = 2006;
```

In gleicher Weise kann man auf die Felder von y mittels y.day, y.month und y.year zugreifen; y.day bedeutet das Feld day der Variablen y und x.day bedeutet das Feld day der Variablen x.

Variablen, deren Typ eine Klasse ist, enthalten Zeiger (Referenzen). Eine Date-Variable enthält also einen Zeiger auf ein Date-Objekt (siehe Abb. 10.1).

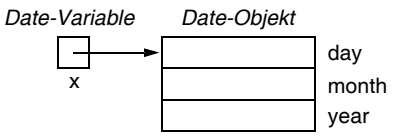


Abb. 10.1 *Date-Variable als Zeiger auf ein Date-Objekt*

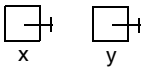
Eine Klasse ist ein Typ, ihre Werte bezeichnet man als Objekte, wobei die Klasse die Struktur ihrer Objekte vorgibt.

Erzeugung von Objekten

Wie Arrayobjekte müssen auch Objekte einer Klasse mittels new erzeugt werden, bevor man auf sie zugreifen kann. Die folgenden Bilder zeigen schrittweise, was dabei geschieht. Durch die Deklaration

```
Date x, y;
```

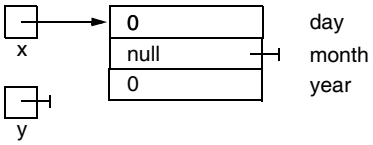
werden lediglich zwei *Objektvariablen* angelegt, die aber noch nirgendwo hinzeigen (sie haben den Wert null):



Nun erzeugen wir ein Date-Objekt und weisen seine Adresse der Variablen x zu:

```
x = new Date();
```

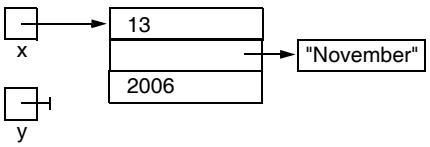
Die Variable x zeigt jetzt auf das neue Date-Objekt, dessen Felder bei der Erzeugung automatisch mit den Werten 0 bzw. null initialisiert wurden. Die Variable y zeigt nach wie vor auf kein Objekt:



Als Nächstes weisen wir den Feldern von x Werte zu:

```
x.day = 13;
x.month = "November";
x.year = 2006;
```

Das Date-Objekt sieht nun so aus:



Freigabe von Objekten

Objekte einer Klasse werden wie Arrayobjekte automatisch durch den *Garbage Collector* freigegeben. Sobald ein Objekt nicht mehr durch einen Zeiger referenziert wird, sammelt es der Garbage Collector ein und stellt seinen Speicherplatz wieder zur Verfügung. Der Java-Programmierer muss sich also nicht um die

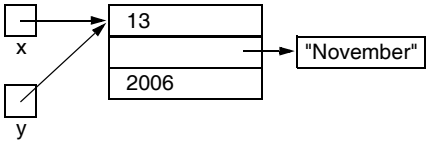
Freigabe von Objekten kümmern und kann so tun, als ob er einen unendlich großen Speicher zur Verfügung hätte, von dem er ständig neue Objekte mittels new anfordern kann. Das ist ein enormer Vorteil gegenüber Sprachen wie *C++*, in denen der Programmierer seine Objekte explizit freigeben muss und dabei Gefahr läuft, Fehler zu begehen, die äußerst schwierig zu finden sind.

Zuweisungen zwischen Objektvariablen

Objektvariablen können einander zugewiesen werden, falls sie den gleichen Typ haben. Was geschieht dabei? Die Zuweisung

```
y = x;
```

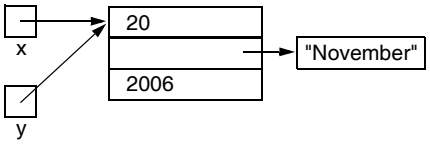
weist der Variablen y den Wert von x zu, also den Zeiger auf das Objekt, auf das x verweist.



Die beiden Variablen x und y zeigen jetzt auf dasselbe Objekt. Wenn man nun ein Feld von y verändert, z.B. durch

```
y.day = 20;
```

so verändert man gleichzeitig auch das entsprechende Feld von x. Da x und y auf dasselbe Objekt zeigen, bezeichnen die Felder x.day und y.day dieselbe Speicherzelle, die nun nach der Zuweisung den Wert 20 enthält.



Zuweisungskompatibilität

Zuweisungen zwischen Objektvariablen sind in Java nur dann erlaubt, wenn die Variablen denselben Typ haben, d.h., wenn ihre Typen durch denselben Namen bezeichnet werden. Man nennt diese Art der Typkompatibilität *Namensäquivalenz*. Im Gegensatz dazu verwenden manche Sprachen (z.B. *Modula-3*) *Strukturäquivalenz*, was bedeutet, dass zwei Typen dann gleich sind, wenn sie dieselbe

Struktur haben, also aus Feldern desselben Typs aufgebaut sind. Namensäquivalenz ist einfacher zu definieren und auch für den Compiler einfacher zu prüfen als Strukturäquivalenz.

Unsere beiden Variablen x und y sind beide vom Typ Date. Aufgrund der Namensäquivalenz haben sie daher denselben Typ. Betrachten wir hingegen folgende Deklarationen

```
class Date1 {
    int d;
    String m;
    int y;
}
Date1 z;
```

so sind x und z nicht vom selben Typ, weil x den Typ Date und z den Typ Date1 hat. Man kann z also nicht an x zuweisen. In Sprachen mit Strukturäquivalenz wären die Typen von x und z kompatibel, weil sie trotz unterschiedlicher Namen dieselbe Struktur aufweisen. Eine Zuweisung von z an x wäre dort erlaubt.

Der Wert null, den wir schon bei Arrays kennen gelernt haben und der so viel bedeutet wie "zeigt auf kein Objekt", darf jeder Objektvariablen zugewiesen werden, zum Beispiel

```
x = null;
```

Das Objekt, auf das x zeigte, hat nach dieser Zuweisung eine Referenz weniger. Wenn kein Zeiger mehr auf das Objekt zeigt, wird es vom Garbage Collector eingesammelt.

Vergleiche

Objektvariablen können auf Gleichheit und Ungleichheit verglichen werden:

```
if (x == y) ...
if (x != y) ...
```

Dabei finden allerdings nur Zeigervergleiche und keine Wertvergleiche statt. Es wird also geprüft, ob x und y auf dasselbe Objekt zeigen. Wenn man feststellen will, ob zwei Objekte denselben *Wert* haben, muss man eine selbst geschriebene Vergleichsfunktion benutzen, z.B.

```
static boolean isEqual (Date x, Date y) {
    return x.day == y.day && x.month.equals(y.month) && x.year == y.year;
}
…
if (isEqual(x, y)) ...
```

Man beachte, dass die Funktion isEqual die beiden month-Felder mittels der String-Methode equals vergleicht (die wir in Kapitel 9 kennen gelernt haben)

```
x.month.equals(y.month)
```

weil

```
x.month == y.month
```

lediglich einen Zeigervergleich durchführen würde; wir wollen aber einen Wertvergleich.

Klassen versus Arrays

Sowohl Klassen als auch Arrays bestehen aus mehreren Elementen und werden durch Zeiger referenziert. Was ist der Unterschied zwischen beiden und wann soll man welches Konstrukt verwenden? Tab. 10.1 fasst die Unterschiede zusammen:

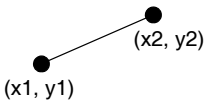
Tab. 10.1 *Unterschiede zwischen Arrays und Klassen*

Arrays	Klassen
Bestehen aus mehreren *gleichartigen* Elementen, z.B. aus lauter *int*-Werten.	Können aus mehreren *verschiedenartigen* Feldern bestehen, z.B. aus einem *int*-Wert und einem *String*-Wert.
Die Elemente eines Arrays haben keinen Namen, sondern werden über einen Index angesprochen, z.B. a[3].	Die Felder einer Klasse haben einen Namen und werden über diesen Namen angesprochen, z.B. x.day.
Die Anzahl der Elemente wird bei der Erzeugung des Arrayobjekts festgelegt.	Die Anzahl der Felder wird bei der Deklaration der Klasse festgelegt.

Man sollte daher Arrays verwenden, wenn man es mit lauter gleichartigen Elementen zu tun hat, die keinen eigenen Namen haben (z.B. eine Tabelle von Zahlen). Hingegen sollte man Klassen verwenden, wenn man verschiedenartige Elemente hat und diese über einen Namen ansprechen will.

Beispiel: Datenstruktur für Linien

Als Beispiel wollen wir nun eine Datenstruktur entwerfen, die Linien in einem Grafikeditor repräsentiert. Eine Linie besteht aus zwei Endpunkten:



Jeder Punkt hat zwei Koordinaten x und y. Da ein Punkt ein Objekt darstellt, das wir auch im Programm als solches ansprechen wollen, fassen wir seine Daten (d.h. die beiden Koordinaten) zu einer Klasse Point zusammen:

```
class Point {
    int x, y;
}
```

Wir könnten zwar einen Punkt im Prinzip auch als Array mit zwei Elementen darstellen, allerdings ist eine Klasse besser geeignet, da man so die beiden Koordinaten durch Namen ansprechen kann.

Auch eine Linie ist ein Objekt, das wir im Programm als solches ansprechen wollen. Es besteht aus zwei Punkten, also fassen wir zwei Punkte zu einer neuen Klasse Line zusammen:

```
class Line {
    Point p1, p2;
}
```

Wir können nun ein Linienobjekt mit seinen beiden Endpunkten folgendermaßen erzeugen:

```
Line line = new Line();
line.p1 = new Point(); line.p1.x = 10; line.p1.y = 20;
line.p2 = new Point(); line.p2.x = 30; line.p2.y = 50;
```

Das ergibt folgende Datenstruktur (siehe Abb. 10.2):

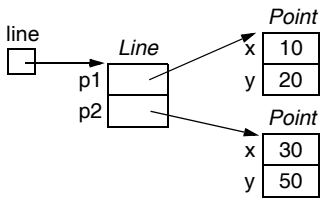


Abb. 10.2 *Ein Line-Objekt bestehend aus zwei Point-Objekten*

Wie man sieht, können Objekte über Zeiger zu einer Datenstruktur zusammengefügt werden. Solche *dynamischen Datenstrukturen* ermöglichen es, beliebig komplexe Dinge der realen Welt zu modellieren. In Kapitel 12 werden wir näher darauf eingehen. In Kapitel 11 werden wir auch sehen, dass die Erzeugung und Initialisierung von Objekten durch so genannte *Konstruktoren* wesentlich vereinfacht werden kann. Um eine Linie wie oben zu erzeugen und zu initialisieren, können wir dann einfach schreiben:

```
Line line = new Line(new Point(10, 20), new Point(30, 50));
```

10.2 Methoden mit mehreren Rückgabewerten

Methoden können in Java nur einen einzigen Wert an ihren Rufer zurückliefern. Wenn wir mehrere Werte zurückgeben wollen, müssen wir diese in eine Klasse verpacken und ein Objekt dieser Klasse zurückgeben. Dies wollen wir uns nun in einem Beispiel ansehen.

Beispiel: Umrechnung von Sekunden auf Stunden, Minuten und Sekunden

Nehmen wir an, wir wollen eine Funktion convert(seconds) schreiben, die Sekunden in Uhrzeiten umrechnen soll. Sie soll also berechnen, wie viele Stunden, Minuten und Sekunden dem Parameter seconds entsprechen. Da eine Funktion nicht drei Werte zurückgeben kann, müssen wir die Rückgabewerte zu einer Klasse Time zusammenfassen:

```
class Time {
    int h;      // hours
    int m;      // minutes
    int s;      // seconds
}
```

Die Funktion convert(seconds) kann nun ihren Parameter mittels Division und Modulorechnung in Stunden, Minuten und Sekunden zerlegen und diese Informationen in einem Time-Objekt verpackt an den Rufer zurückgeben. Auf diese Weise haben wir die Einschränkung umgangen, dass Funktionen nur einen einzigen Rückgabewert liefern dürfen.

```
static Time convert (int seconds) {
    Time t = new Time();
    t.h = seconds / 3600;
    t.m = (seconds % 3600) / 60;
    t.s = seconds % 60;
    return t;
}
```

Time-Objekte sind aber nicht nur deshalb nützlich, weil wir mit ihnen die Beschränkung von Funktionen auf einen einzigen Rückgabewert umgehen können. Die Klasse Time ist eine sinnvolle Abstraktion an sich. Wir können zum Beispiel Operationen für den Vergleich oder die Addition von Uhrzeiten definieren und dann mehrere Time-Objekte mit diesen Operationen verknüpfen. Darauf werden wir in Kapitel 11 näher eingehen.

Um zu zeigen, wie Klassen in den Kontext eines Programms eingebettet werden, geben wir nun ein vollständiges Programm an, in dem Time und convert deklariert und benutzt werden.

```
class Time {
    int h, m, s;
}

class MainProgram {

    static Time convert (int seconds) {
        Time t = new Time();
        t.h = seconds / 3600;
        t.m = (seconds % 3600) / 60;
        t.s = seconds % 60;
        return t;
    }

    public static void main (String[] arg) {
        Out.print("enter seconds>");
        int seconds = In.readInt();
        while (In.done()) {
            Time t = convert(seconds);
            Out.println(t.h + ":" + t.m + ":" + t.s);
            Out.print("enter seconds>");
            seconds = In.readInt();
        }
    }

}
```

10.3 Kombination von Klassen und Arrays

In größeren Anwendungen kommen meist verschiedene Arten von Objekten vor. Daher werden in diesen Anwendungen auch unterschiedliche Klassen zu ihrer Modellierung deklariert. Oft werden Klassen dabei auch mit Arrays kombiniert, wie etwa im folgenden Beispiel.

Nehmen wir an, wir wollen ein Telefonbuch verwalten, in dem Personennamen und Telefonnummern gespeichert sind. Abstrakt gesehen, ist ein Telefonbuch eine Tabelle der folgenden Art:

	name	phone
0	...	...
	Maier	876 8878
	Mayr	543 2343
	Meier	656 2332
999	...	...

Leider können wir diese Tabelle nicht als zweidimensionales Array implementieren, weil die Elemente unterschiedliche Typen haben: Namen sind Strings, Tele-

fonnummern sind vom Typ int. Wir können aber Arrays und Klassen kombinieren, um zu einer geeigneten Datenstruktur zu kommen. Dabei haben wir sogar zwei Möglichkeiten, zwischen denen wir wählen können:

- Ein Array aus Zeilenobjekten
- Ein Objekt bestehend aus zwei Spaltenarrays

Ein Array aus Zeilenobjekten

Jede Zeile des Telefonbuchs kann als Objekt einer Klasse Entry modelliert werden:

```
class Entry {  // a single line in the phone book
    String name;
    int phone;
}
```

Das gesamte Telefonbuch ist dann ein Array solcher Zeilenobjekte:

```
Entry[] book = new Entry[1000];
```

Dies entspricht der Datenstruktur in Abb. 10.3:

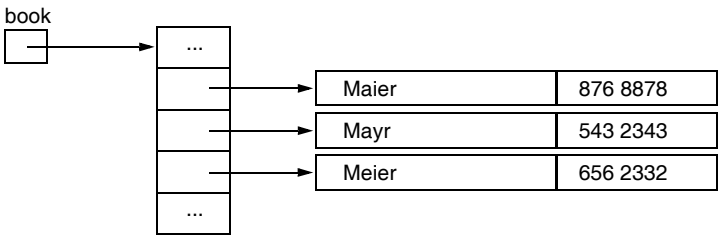


Abb. 10.3 *Telefonbuch als Array von Zeilenobjekten*

Um auf den Eintrag i zuzugreifen, schreiben wir book[i].name und book[i].phone. Beachten Sie, dass jedes Zeilenobjekt vor seiner Benutzung mit new erzeugt werden muss, also

```
book[i] = new Entry();
book[i].name = ...;
book[i].phone = ...;
```

Das Telefonbuch-Array selbst wurde bereits bei seiner Deklaration mittels new erzeugt.

Ein Objekt bestehend aus zwei Spaltenarrays

Anstatt ein Array aus Objekten zu verwenden, können wir auch ein Objekt bestehend aus zwei Arrays benutzen. Statt also die Tabelle in Zeilen zu zerlegen, zerle-

gen wir sie jetzt in Spalten. Die erste Spalte ist ein Array von Personennamen, die zweite Spalte ein Array von Telefonnummern. Die beiden Arrays fassen wir zu einer Klasse PhoneBook zusammen, die folgendermaßen aussieht:

```
class PhoneBook {
    String[] name;     // column of names
    int[] phone;       // column of phone numbers
}
```

Wir deklarieren eine Variable book für das Telefonbuch und müssen vor ihrer Benutzung wieder das PhoneBook-Objekt und die beiden darin enthaltenen Arrays erzeugen:

```
PhoneBook book = new PhoneBook();
book.name = new String[1000];
book.phone = new int[1000];
```

Dadurch bauen wir die in Abb. 10.4 gezeigte Datenstruktur auf.

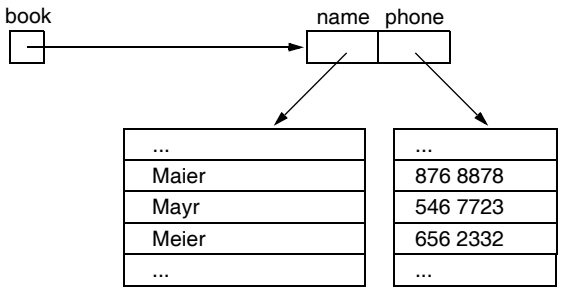


Abb. 10.4 *Telefonbuch als Objekt bestehend aus zwei Spaltenarrays*

Um auf die Person i zuzugreifen, schreiben wir book.name[i] und book.phone[i]. Beachten Sie bitte den Unterschied zur vorherigen Variante. Wir wählen hier zuerst die Spalte aus (name oder phone) und dann erst die Zeile. Bei der vorigen Variante haben wir zuerst die Zeile und dann die Spalte ausgewählt. Welche Datenstruktur sollte also gewählt werden? Beide sind gleich effizient. Die erste ist hier allerdings vorzuziehen, da ein Telefonbuch aus logischer Sicht eher eine Sammlung von Personen-Objekten ist als zwei getrennte Arrays von Namen und Telefonnummern.

Eintragen und Suchen von Personen im Telefonbuch

Um das Beispiel zu vervollständigen, entwerfen wir nun noch eine Methode enter(name, phone), mit der wir Namen und Telefonnummern in das Telefonbuch eintragen, und eine Methode lookup(name), mit der wir die Telefonnummer zu einem gegebenen Namen abfragen können. Wir wählen als Datenstruktur ein Array aus Zeilenobjekten, also die erste der beiden oben beschriebenen Varianten.

Das Programm hat folgende Struktur:

```
class Entry {
    String name;
    int phone;
}

class MainProgram {
    static Entry[] book;          // the phone book (global variable!)
    static int nEntries = 0;      // current number of entries in the book

    static void enter (String name, int phone) {...}
    static int lookup (String name) {...}
    public static void main (String[] arg) {...}
}
```

Das Telefonbuch book muss sowohl für die Methode enter als auch für die Methode lookup() zugreifbar sein. Daher deklarieren wir es auf Klassenebene; book ist also eine globale Variable. Aus Gründen, die in Kapitel 11 erklärt werden, müssen wir die Variable mit dem Zusatz static deklarieren.

Im Array book sind nicht unbedingt alle Zeilen gefüllt, daher benötigen wir eine zweite Variable nEntries, die angibt, wie viele Einträge das Telefonbuch bereits hat. Am Anfang ist das Telefonbuch leer. Wir initialisieren daher nEntries mit 0.

Die Methode enter bekommt als Parameter einen Namen und eine Telefonnummer und erzeugt daraus einen neuen Eintrag im Telefonbuch. Dabei muss man darauf achten, dass das Telefonbuch nicht überläuft, dass also nicht mehr Personen eingetragen werden, als Platz haben. Wenn man versucht, einen Namen in ein bereits volles Telefonbuch einzutragen, geben wir eine Fehlermeldung aus und ignorieren den Eintrag (in Kapitel 19 werden wir eine bessere Fehlerbehandlungstechnik kennen lernen). Die Implementierung von enter sieht folgendermaßen aus:

```
static void enter (String name, int phone) {
    if (nEntries >= book.length)
        Out.println("-- phone book full, entry " + name + " ignored");
    else {
        Entry e = new Entry();
        e.name = name; e.phone = phone;
        book[nEntries] = e;
        nEntries++;
    }
}
```

Um die Telefonnummer zu einem gegebenen Namen zu suchen, schreiben wir eine Funktion lookup, die alle Einträge des Telefonbuchs sequenziell durchsucht und den als Parameter mitgegebenen Namen mit dem Namen jeder Zeile vergleicht. Wird der Name gefunden, geben wir die entsprechende Telefonnummer zurück, wird er nicht gefunden, liefern wir -1, um anzuzeigen, dass der Name nicht im Telefonbuch steht:

```
static int lookup (String name) {
    int i = 0;
    while (i < nEntries && !name.equals(book[i].name))
        i++;
    // i == nEntries || name.equals(book[i].name)
    if (i == nEntries) return -1;  // not found
    else return book[i].phone;
}
```

Beachten Sie bitte, dass die beiden Abfragen im Kopf der while-Schleife nicht vertauscht werden dürfen. Wir dürfen erst dann auf book[i] zugreifen, wenn wir sicher sind, dass i < nEntries ist, dass also der Index im erlaubten Bereich liegt.

Nun fehlt noch das Hauptprogramm, das wir in der Methode main implementieren. Wir lesen dazu das Telefonbuch von einer Datei ein und fragen anschließend den Benutzer wiederholt nach einem Namen, für den wir dann die entsprechende Telefonnummer suchen.

```
public static void main (String[] arg) {
    String name;
    int phone;
    book = new Entry[1000];
    //------- read the phone book from file "PhoneBookFile"
    In.open("PhoneBookFile");
    name = In.readWord();
    while (In.done()) { // while a name could be read
        phone = In.readInt();
        enter(name, phone);
        name = In.readWord();
    }
    In.close();
    //------- search names in the phone book
    for (;;) {
        Out.print("Name>");  // ask the user to enter a name
        name = In.readWord();
        if (!In.done()) break;
        phone = lookup(name);
        if (phone == -1)
            Out.println(name + " unknown");
        else
            Out.println("phone number is " + phone);
    }
}
```

Um von einer Datei zu lesen, öffnen wir sie mit In.open. Anschließend können wir von ihr mit In.readWord und In.readInt lesen. Am Ende schließen wir sie wieder mit In.close, wodurch In.readWord anschließend wieder von der Tastatur liest. Nach jeder Leseoperation zeigt In.done an, ob das Lesen erfolgreich war oder nicht.

Die Methoden enter und lookup gehören eigentlich zu den Daten des Telefonbuchs. Man sollte sie deshalb zu Bestandteilen der Klasse PhoneBook machen. Im nächsten Kapitel werden wir sehen, wie das geht.

Übungsaufgaben

1. *Artikelverwaltung*. Schreiben Sie ein Programm, das eine Verkaufszahlendatei mit folgender Syntax einliest:

```
Datei = { Artikel }.
Artikel = artikelNr einzelPreis { menge } "0".
```

artikelNr, einzelPreis und menge sind positive ganze Zahlen größer 0. Ihr Programm soll die Umsätze aller Artikel berechnen und auf dem Bildschirm ausgeben. Für folgende Eingabe

```
102700      999   1 3 1 2 4     0
102701     3250   2 13 4 1 1    0
102702     1190   2 1           0
...
```

soll diese Ausgabe erzeugt werden:

```
Artikel-Nummer     Umsatz
   102700           10989
   102701           68250
   102702            3570
            ...
```

Schreiben Sie jeweils eine Methode zum Lesen und zum Ausgeben eines Artikels:

```
static Article readArticle() {...}
static void printArticle(Article a) {...}
```

wobei Article eine Klasse ist, die die Daten eines Artikels enthält.

2. *Schneiden von Rechtecken*. Schreiben Sie eine Methode intersection(r1, r2), die das Schnittrechteck der beiden Rechtecke r1 und r2 berechnet und zurückgibt. Rechtecke sollen durch eine Klasse

```
class Rectangle {
    int x, y;       // left top corner
    int width;      // width of rectangle
    int height;     // height of rectangle
}
```

beschrieben werden. Wenn sich r1 und r2 nicht schneiden, soll intersection als Ergebnis null liefern.

3. *Datumsberechnung*. Implementieren Sie eine Methode `dayDiff(d1, d2)`, die zwei Datumsangaben `d1` und `d2` der Klasse `Date` als Parameter bekommt und ihre Differenz in Tagen zurückgibt. Berücksichtigen Sie auch Schaltjahre.

4. *Kundenkartei*. Die Kundenkartei einer Firma enthält pro Kunden einen Datensatz, der den Namen des Kunden, seine Kundennummer, seine Privat- und Geschäftsadresse sowie eine Liste der im laufenden Jahr erfolgten Bestellungen enthält. Jede Bestellung besteht aus einer Bestellmenge, einer Artikelnummer und einem Artikelpreis. Adressen bestehen aus Straße, Hausnummer, Postleitzahl und Ort.

   Modellieren Sie diese Kartei mittels Arrays und Records und stellen Sie einen Ausschnitt davon grafisch dar. Definieren Sie ein geeignetes Eingabeformat, mit dem diese Daten von einer Datei gelesen werden können. Schreiben Sie ein Programm, das Eingabedaten in diesem Format liest, die Kartei damit befüllt und schließlich in ansprechender Form wieder ausgibt.