12 Files And Multidimensional Arrays

Overview

In this chapter you will develop software to manage information about email messages. The software reads information from hard-disk files into multi-dimensional arrays. Be sure to read the first three sections of Chapter Nine on Exceptions before reading this chapter. None of the material in this chapter is required for other chapters.

- Sections 12.2-12.3 introduce basic file-handling Sun library classes.
- Section 12.4 presents the StringTokenizer and StreamTokenizer library classes.
- Section 12.5 develops multi-dimensional arrays in detail.
- Sections 12.6 completes the development of Version 1 of the email software.
- Sections 12.7-12.9 cover more topics related to arrays and files.
- Sections 12.10-12.11 have some information on rarely-used language features and on Java bytecode.

12.1 Analysis And Design For The Email Software

Your employer wants you to analyze the email messages that are sent between employees. He wants this for various reasons, among which are:

- He is worried that some employees are overburdening the email system.
- He needs to monitor content of messages to check for discriminatory acts that could cause him a lot of grief with the government if he does not detect them and do something about them.
- He wants to see if some employees are not doing their jobs, as indicated by a very low number of email messages sent.

The data describing the email is in files that are stored on a hard disk. Each day's email is in a separate file. Such a file contains two lines for each email sent that day. The first line is two words, the sender followed by the receiver (each employee has a one-word email name). The second line is the subject line of the email. The company only has forty-some employees, so one day's file only contains one or two thousand email messages. Your job is to process this file and produce various statistics describing it.

The first program you are to write is to read one day's email file and produce the following information:

- How many emails were sent that day.
- How many employees sent email that day.
- Who did not send any email at all that day.
- What was the average number of email messages sent per person.
- Which people sent at least twice as many emails as they received.
- Which people received at least twice as many emails as they sent.
- What kind of idiot sends himself email.
- Which people sent the most amount of email.
- Which people were sent the most amount of email.

Later programs will look at the subject lines themselves, even though this first program does nothing with them. So the object design should include storing the subject lines. The top-level design of the program is easily developed from this analysis, as shown in the accompanying design block.

STRUCTURED NATURAL LANGUAGE DESIGN for the main logic
1. Ask the user for the name of the file that contains one day's data and the name of the output file where the results are to be stored.
2. Open the data file for reading.
3. Read in all the information about email messages and store it in some kind of database.
4. Ask questions of that database and print out the answers to the output file.

First you need to know how to work with text files stored on a hard disk. The next two sections introduce standard library classes for this. The Sun standard library also has classes for working with files containing numbers and String objects rather than just characters; those concepts are introduced in later sections of this chapter.

12.2 FileReader And BufferedReader For Input

A text file stores on the hard disk only the characters to be displayed, plus newline and tab characters. In Word or Wordpad, make sure you select type `txt` when storing the file for a program to read, not type `doc`. The `doc` type contains special coding that specifies page numbering, margins, and other extra information that the Word software uses.

Reading from a text file

You can create an object to read characters from a text file. The statement

```
FileReader input = new FileReader ("data.txt");
```

creates a FileReader object connected to the file stored with the name `data.txt` in the current folder on the hard disk. However, this **FileReader** object is only capable of reading one character at a time or else a large array of characters.

For instance, `input.read()` returns the int Unicode value of the next char value (or -1 if at the end of the file). And if `lotsOfChars` is an array of char values, `input.read (lotsOfChars, 100, 500)` attempts to read 500 characters and store them in `lotsOfChars` starting at index 100; it returns the number of chars actually read (or -1 if at the end of the file). This is illustrated by the top part of Figure 12.1.

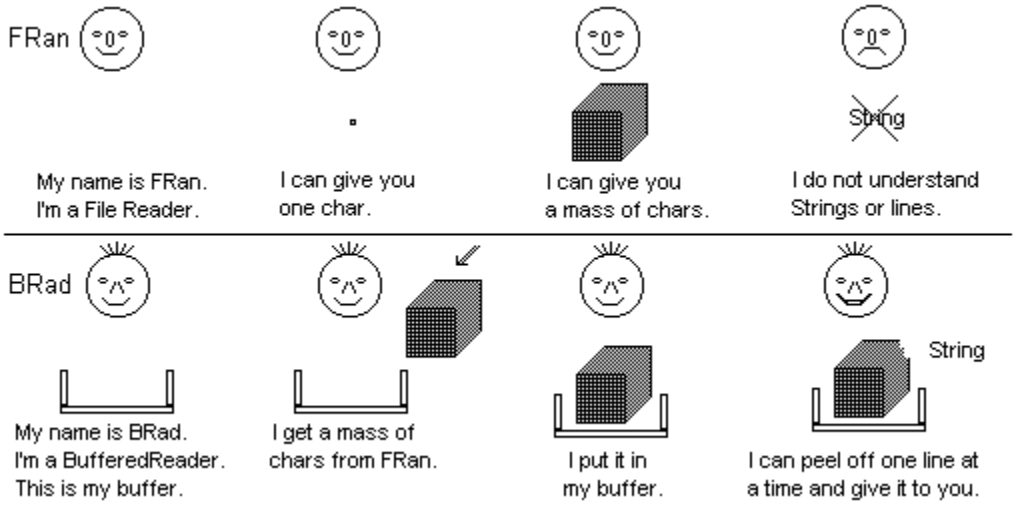

Figure 12.1 BufferedReader mediates between you and FileReader

This is convenient for some purposes, but not for text files. A FileReader has no method for reading one line of characters at a time. The statement

```
BufferedReader file = new BufferedReader (input);
```

creates out of that FileReader a **BufferedReader** object that has a `readLine` method (illustrated in the lower part of Figure 12.1). This method gets one entire line of input, discards the end-of-line marker, and returns the rest as a String of characters. If you have already reached the end of the file, this method returns the null value. You get the next line of input by using a statement such as the following:

```
String s = file.readLine();
```

The TextFileOp class (not Sun standard library)

It is useful to have a class to collect several independent class methods that work with text files. Call this utilities class **TextFileOp**. A start on this class is in Listing 12.1 (see next page). The `findDigits` method in the upper part of Listing 12.1 shows how you can read lines from a file to find the first line that begins with a digit. It tests the condition `si != null` to make sure end-of-file has not been reached, and it tests the condition `si.length() > 0` to ignore a blank line.

The BufferedReader class, the IOException class, and other classes introduced in this section and the next, are all in the java.io package. So you need `import java.io.*` at the top of a compilable file that uses these classes.

Another useful independent method would be a `compareFiles` method that tells whether two text files are identical in their contents. This requires that you repeatedly read one line from each file until you get to the end of the files. If you ever see a pair of lines that are not equal, then the files are not identical. A reasonable structured design of this method is in the accompanying design block. The lower part of Listing 12.1 contains the coding for this `compareFiles` method.

DESIGN for the compareFiles method

1. Create two file objects connected to existing physical text files.
2. Read one line from each file.
3. Repeat the following until one of the files is finished and thus the read fails...
    - 3a. If the two current lines are not equal then...
        - 3aa. Return `false` as the result.
    - 3b. Read one more line from each file.
4. Return `true` if both files are finished, otherwise return `false`.

Reading from the keyboard

You can create an object to read characters from the keyboard. The statement

```
InputStreamReader input = new InputStreamReader (System.in);
```

creates an **InputStreamReader** object connected to the keyboard, because the System class has a InputStream object `System.in` which is connected to the keyboard, and the InputStreamReader class has a constructor with an InputStream parameter. However, this InputStreamReader object can only read one character at a time (`read()` returns an int value -1 to 65535) or a large array of characters (`read(char[])` returns the number of chars read), just as for a FileReader.

Listing 12.1 The TextFileOp utilities class; more methods later

```
import java.io.*;  // for FileReader, IOException, and others

public class TextFileOp
{
   /** Find the first line that begins with a digit. */

   public static String findDigit (String fileName)
                                    throws IOException
   {  BufferedReader fi = new BufferedReader
                         (new FileReader (fileName));
      String si = fi.readLine();
      while (si != null)
      {  if (si.length() > 0 && si.charAt (0) >= '0'
                             && si.charAt (0) <= '9')
            return si;
         si = fi.readLine();
      }
      return "";
   }  //=====================


   /** Tell whether the two files have the same contents. */

   public static boolean compareFiles (String one, String two)
                                       throws IOException
   {  BufferedReader inOne = new BufferedReader
                            (new FileReader (one));
      BufferedReader inTwo = new BufferedReader
                            (new FileReader (two));
      String s1 = inOne.readLine();
      String s2 = inTwo.readLine();

      while (s1 != null && s2 != null)
      {  if ( ! s1.equals (s2))
            return false;
         s1 = inOne.readLine();
         s2 = inTwo.readLine();
      }
      return s1 == s2;  // true only when both are null
   }  //=====================
}
```

BufferedReader's constructor accepts any **Reader** object; FileReader and InputStreamReader are both subclasses of the abstract Reader class. Figure 12.2 shows the relationships among these classes. So the statement

```
BufferedReader keyboard = new BufferedReader (input);
```

creates out of that InputStreamReader an object that has a `readLine` method, just as for the file object, because both are BufferedReader objects. Now you can read whole lines at a time instead of character-by-character. The `new BufferedReader` call is analogous to an upgrade from coach to business-class -- you get the same end result, but the way you do it is much more comfortable, as in

```
String s = keyboard.readLine();
```

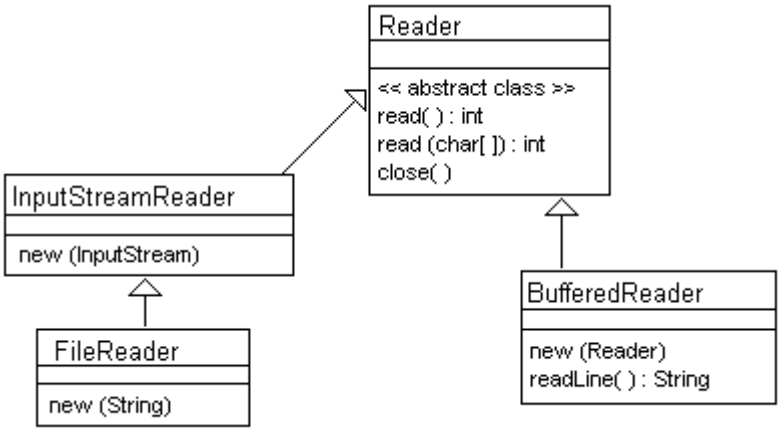

Figure 12.2 UML class diagram for Reader and some subclasses

By way of illustration, you could replace the first statement in the `findDigit` method of Listing 12.1 by the following if you want the method to be able to read from either the keyboard or a text file, depending on whether the `filename` is the empty string:

```
BufferedReader fi = fileName.length() > 0
     ? new BufferedReader (new FileReader (fileName))
     : new BufferedReader (new InputStreamReader (System.in));
```

Handling IOExceptions

Any call of the `readLine` method of the BufferedReader class could possibly throw an IOException which you must handle. So can `new FileReader`. Exception-handling is discussed in detail at the beginning of Chapter Nine. What you need to know about it for here is that you have four ways of handling this IOException:

1. Learn how to wrap each call of `readLine` or `new FileReader` in a try/catch statement. This is discussed in Chapter Nine.
2. Write a class to encapsulate the entire problem. The Buffin class in Chapter Nine does the wrapping for you -- the Buffin class is like an upgrade to first class. None of its methods throws an Exception under any circumstances.
3. If the proper handling of the Exception depends on circumstances known only to the method that calls your method, put `throws IOException` in the heading of your method and let the calling method use a try/catch statement to handle it properly.
4. Put the phrase `throws IOException` in the heading of every method that calls `readLine` or that calls a method that has that phrase in its heading. Then a failure of a file access immediately terminates the `main` method with an appropriate message.

A **buffer** is a storage location where input values are kept until asked for. A BufferedReader object obtains roughly 8192 characters at a time from the file and stores them in its private buffer in RAM. Each time you call the BufferedReader's `readLine` method, it gives you some of the characters from its buffer. As a consequence, you may have only one retrieval of information from the physical hard disk file for each hundred or so times that you execute `readLine`. Buffering is a much more efficient way of getting information than reading one or a few characters at a time.

A precondition for all exercises in this chapter is that all parameters are non-null.

Exercise 12.1 Write a TextFileOp method `public static void findCaps (String name)`: It opens a text file with a given name and then prints to `System.out` each line of that file that begins with a capital letter.
Exercise 12.2 Write a method `public int readInt()` for a subclass of BufferedReader: The executor reads one line of input from the file, parses it as an int value, and returns that value. Return 0 on end-of-file or any NumberFormatException.
Exercise 12.3 Write a TextFileOp method `public static int numChars (String name)`: It returns the number of characters in the text file of the given name.
Exercise 12.4 (harder) Revise the preceding exercise to return instead the average number of characters per line in the text file of the given name.
Exercise 12.5 (harder) Write a TextFileOp method `public static boolean descending (String name)`: It tells whether the lines in the file of the given name are in descending order (i.e., if `s1` is one line and `s2` is the line after it, then `s1.compareTo(s2)>=0`).
Exercise 12.6* Write an independent method `public static void alternate (String one, String two)`: It opens two text files with the given names and then prints the lines of those two files to `System.out`, alternating between them. That is, first line#1 of the first file, then line#1 of the second, then line#2 of the first, then line#2 of the second, etc. Stop when one file is emptied.
Exercise 12.7* Write a TextFileOp method `public static String firstOne (String name)`: It returns the lexicographically first line of the text file with the given name (i.e., a line `s1` for which `s1.compareTo(s2) <= 0` for each line `s2` in the file). It returns null if the file is empty.
Exercise 12.8** Write an application program: It gets the name of a text file from the user at the keyboard, opens the file (but opens the keyboard if the name does not have positive length), and prints every line that is the same as the line before it in the file.
Exercise 12.9** Write a TextFileOp method `public static int numWords (String name)`: It returns the number of words in the text file of the given name. The end of a word is a non-whitespace character that is followed by either whitespace or the end of the line. Whitespace is any character that is less than or equal to a blank.

12.3 FileWriter And PrintWriter For Output

Writing to a text file on a hard disk is slightly simpler than reading from one. You first create a PrintWriter object using a statement as follows, with whatever String value you want for `filename`. The FileWriter constructor can throw an IOException:

```
PrintWriter out = new PrintWriter (new FileWriter (filename));
```

This **opens** the physical file for output. Printing starts at the beginning of the file, so you lose whatever was in that disk file before you opened it. However, an alternative is to open the file for appending data: `new FileWriter (fileName, true)` means that whatever is already in the file is preserved, and what you write is added after it.

A **FileWriter** object can write one character at a time, or a whole array of characters, but not a String of characters at once. A **PrintWriter** has the `print` and `println` methods that the `System.out` variable has, to write many characters at a time. That is why you "upgrade" to the PrintWriter class. In particular, these two statements

```
out.print (whatever);
out.println (whatever);
```

can be used to print a String, character, or number. The `println` method outputs a newline character `'\n'` after printing its parameter, but the `print` method does not. Both FileWriter and PrintWriter are subclasses of the abstract **Writer** class.

It is quite important that you call the `close` method of a PrintWriter object when you are finished writing to the file, before you exit the program. If you do not, you may lose the last few lines of output. The reason is that a PrintWriter may buffer the output to cut down on the number of times it has to access the physical file. Closing the file **flushes** the buffer to the physical file (sends to the file all output that the buffer is holding onto as an efficiency measure):

```
out.close();
```

Writing to the terminal window

It is sometimes useful to have a PrintWriter object that writes to the terminal window. This can be done with the following statement. When you use this `screen` object to print something that you want to appear before a `readLine` method is executed, you should call `screen.flush()` before calling the `readLine` method:

```
PrintWriter screen = new PrintWriter (System.out);
```

A PrintWriter never throws an IOException. So there is no real need for an additional wrapper class analogous to the Buffin wrapper around a BufferedReader. Figure 12.3 shows the standard library classes introduced in this section. The `write(int)` method accepts a single int value that it casts to a char to be written to the file.

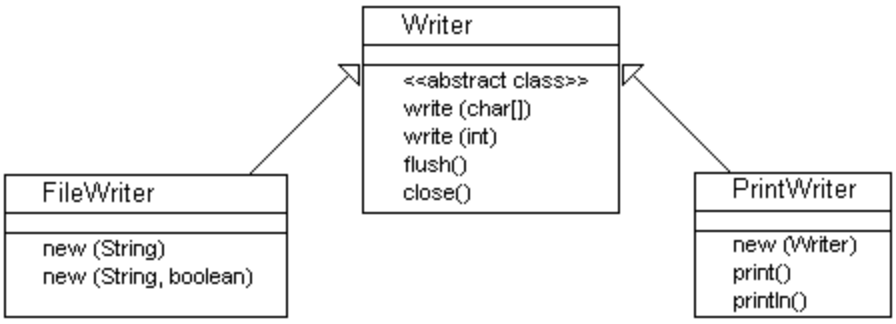

Figure 12.3 UML class diagram for text output files

More TextFileOp methods

Listing 12.2 (see next page) illustrates more class methods that work with files. The `createRandoms` method repeatedly picks a random integer in the range from 0000 to 9999 inclusive and writes it to the specified file. The `file.print` call in the inner for-loop produces a group of four random integers followed by tab characters, so that the numbers will be separated in the file and appear in nice columns when printed. Then the `file.println` call prints the fifth random number on the line and starts a new line. The outer for-loop produces the number of lines requested. The file is then closed.

The `mergeTwo` method illustrates logic for sorting information that is too massive to store in RAM. This method takes two files that have their lines in ascending order and produces a new combined file containing all the lines in both files, all in ascending order. Each iteration of the loop determines which of the two String values, `s1` or `s2`, is to be written next to the merged file. It will be the string from the first file if the second file has been completely read in (signaled by `s2` being null) or if neither file has been completely read in and `s1` comes before `s2`. In all other cases, the string from the second file will be written next. The loop continues until both input files have been read.

Exercise 12.10 Write a TextFileOp method `public static void copy (BufferedReader infile, String outName)`: The method copies every line in the given BufferedReader to the text file named `outName`.

Listing 12.2 More independent class methods working with files

```
// public class TextFileOp, continued

   /** Print the specified number of lines of random 4-digit
    *  non-negative integers, 5 integers per line. */

   public static void createRandoms (String name, int numLines)
                                     throws IOException
   {  PrintWriter file = new PrintWriter (new FileWriter (name));
      java.util.Random randy = new java.util.Random();
      for (int k = 0;  k < numLines;  k++)
      {  for (int i = 0;  i < 4; i++)
            file.print (randy.nextInt (10000) + " \t");
         file.println (randy.nextInt (10000));
      }
      file.close();
   }  //=====================


   /** Read from two BufferedReaders known to have their lines in
    *  ascending order.  Write all lines to a file named outName
    *  so that the file has its lines in ascending order.  */

   public static void mergeTwo (BufferedReader inOne,
         BufferedReader inTwo, String outName) throws IOException
   {  PrintWriter merged = new PrintWriter
                           (new FileWriter (outName));
      String s1 = inOne.readLine();
      String s2 = inTwo.readLine();
      while (s1 != null || s2 != null)
      {  if (s2 == null || (s1 != null && s1.compareTo (s2) < 0))
         {  merged.println (s1);
            s1 = inOne.readLine();
         }
         else  // the line from inTwo is next in order
         {  merged.println (s2);
            s2 = inTwo.readLine();
         }
      }
      merged.close();
   }  //=====================
```

Exercise 12.11 (harder) Write a TextFileOp method `public static void under40 (String inName, String outName)`: It opens a BufferedReader and a PrintWriter, then it writes to the output file every line in the input file that has less than 40 characters.

Exercise 12.12* Revise the `compareFiles` method in Listing 12.1 to return the first line you see in the first file that does not exactly match the corresponding line of the second file. Return null if the files are identical.

Exercise 12.13* Rewrite the `createRandoms` method in Listing 12.2 to use a single loop, with an if-statement inside the for-loop to call either `print` or `println`.

Exercise 12.14* Write a TextFileOp method `public static void capsAndSmalls (String inFile, String outOne, String outTwo)`: It reads from one named text file, prints to the second named text file each line that begins with a capital letter, and prints to the third named text file each line that begins with a lowercase letter.

Exercise 12.15** Write a TextFileOp method `public static void reverse (String fileName)`: It reads all the lines in the text file of that name and writes them back to the same file in reverse order. The file has at most 1000 lines. Use an array to store the lines between reading and writing.

12.4 The StringTokenizer And StreamTokenizer Classes

The design for the main logic of the Email software, presented in Section 12.1, is repeated below for your convenience:

STRUCTURED NATURAL LANGUAGE DESIGN for the main logic

1. Ask the user for the name of the file that contains one day's data and the name of the output file where the results are to be stored.
2. Open the data file for reading.
3. Read in all the information about email messages and store it in some kind of database.
4. Ask questions of that database and print out the answers to the output file.

This design indicates the need for several kinds of objects to implement this design:

- A BufferedReader object for input.
- A PrintWriter object for output.
- A Message object to represent one email message.
- An EmailSet object to store the number of emails from each person to each other person.

You already have BufferedReader and PrintWriter objects, so next consider what a Message object has to be able to do. You need to construct a Message object that contains the information about the next email message obtained from the BufferedReader object. And you need to be able to ask that Message object for its sender, its receiver, and its subject heading.

How will you recognize, when you construct a Message object for the next email in the BufferedReader object, that you have already come to the end of the file? A reasonable method is to have the constructor return a Message with a null sender when the end of the input is reached. That is, `someMessage.getSender()` returns null to signal the Message had no input to get data from. So we can use this sort of logic:

```
Message data = new Message (file);
while (data.getSender() != null)
{  // various statements go here to process one data value
   data = new Message (file);
}
```

The StringTokenizer methods

The Message constructor uses a new standard library class named StringTokenizer, from the `java.util` package, which includes several highly useful methods for breaking up a string of characters into its individual words:

- `new StringTokenizer (inputString)` establishes the parameter as the input string of characters for the executor. It throws a NullPointerException if the parameter is null.
- `someStringTokenizer.nextToken()` returns the next available token from the input string and advances in the sequence of tokens. It throws a java.util.NoSuchElementException if no more tokens are available.
- `someStringTokenizer.hasMoreTokens()` tells whether the input string has any tokens left that have not yet been returned by `nextToken`.
- `someStringTokenizer.countTokens()` tells how many more times you may call `nextToken` for this StringTokenizer object.

A StringTokenizer easily breaks up a String value into parts separated by blanks. These parts are called tokens, which is a more general term than "words". Specifically, a **token** within a String value is a sequence of non-whitespace characters with whitespace on each side (the beginning and end of the String value count as whitespace for purposes of this definition). Using a StringTokenizer is much easier than writing methods such as `trimFront` and `firstWord` in the StringInfo class of Chapter Six. (Whitespace includes blanks, tab characters, newline characters, and form feeds to start a new page.) Listing 12.3 (see next page) illustrates the use of a StringTokenizer object for the Email software.

The following logic illustrates the use of `hasMoreTokens` to count all tokens in a String that start with a numeric digit. The if-condition illustrates the kind of hard-to-follow logic some students produce, thinking it is classier. You would not do anything like that in your own programs, would you? An exercise has you clean it up:

```
public static int countNumerals (String s)    // independent
{  int count = 0;
   StringTokenizer data = new StringTokenizer (s);
   while (data.hasMoreTokens())
   {  if ((data.nextToken().charAt (0) - '0' + 10) / 10 == 1)
         count++;
   }
   return count;
}  //======================
```

The StreamTokenizer class

If you need more flexibility than what StringTokenizer offers, `java.io` contains the **StreamTokenizer** class, which reads in a file and separates it into tokens. It can be set to skip Java comments and to treat everything from a quote mark down to its end-quote as a single token, with proper adjustments for the backslash character. The following are a start on understanding StreamTokenizer, but you have to read the javadoc description in the java documentation on your hard disk or online to use it properly:

- `new StreamTokenizer(someReader)` creates the file processing object.
- `someStreamTokenizer.nextToken()` returns an int value such as TT_WORD (the current token is a word), TT_NUMBER (the current token is a number), TT_EOF (end-of-file has been reached), or the int equivalent of the one-character token.

Listing 12.3 The Message object class for the Email software

```
import java.util.StringTokenizer;
import java.io.BufferedReader;
import java.io.IOException;

public class Message
{
   private String itsReceiver = null;
   private String itsSubject = null;
   private String itsSender = null;


   /** Create a Message object from the next two lines of the
    *  source file.  Set sender to null at end-of-file. */

   public Message (BufferedReader source) throws IOException
   {  String s = source.readLine();
      if (s != null)
      {  StringTokenizer line = new StringTokenizer (s);
         if (line.hasMoreTokens())
         {  itsSender = line.nextToken();
            if (line.hasMoreTokens())
            {  itsReceiver = line.nextToken();
               itsSubject  = source.readLine();
            }
         }
      }
   }  //=====================


   public String getSender()
   {  return itsSender;
   }  //=====================


   public String getReceiver()
   {  return itsReceiver;
   }  //=====================


   public String getSubject()
   {  return itsSubject;
   }  //=====================
}
```

Exercise 12.16 Revise the Message constructor in Listing 12.3 to return a Message with `itsSender` being null when the first line of the input has only one token.
Exercise 12.17 Revise the body of the while-statement in the `countNumerals` method to be much clearer to the reader.
Exercise 12.18 Explain why the `countNumerals` method in the text cannot test the simpler expression `(data.nextToken().charAt (0) - '0') / 10 == 0`.
Exercise 12.19* Write an independent method `public static int numTokens (String name)`: It returns the number of tokens in the text file of the given name.
Exercise 12.20* Write an independent method `public static boolean ascending (String given)`: It tells whether all of the tokens in a given String parameter are in ascending order. Use StringTokenizer.

12.5 Defining And Using Multidimensional Arrays

One part of the Email main logic requires storing the information from a Message object in some sort of EmailSet database object. This EmailSet object class needs a constructor and a method for adding the information from a single Message object. For this Version 1 of the software, that information is just a count of emails between two specific people without trying to store the subject line.

It also makes sense to have methods to retrieve the total number of Message objects added so far, the total number of employees who were sending email that day, and the total number sent from one specific person `from` to another specific person `to`. These methods supply all you need to calculate the average number sent.

The standard way of storing one piece of information for each <from,to> pair of values is a two-dimensional array. But this requires that <from,to> be non-negative integers rather than Strings. If a Message object could produce integer-valued codes for the employees, rather than their actual email names, you could use the structure shown in Listing 12.4 for the EmailSet objects: `itsItem[from][to]` is the number sent from `from` to `to`.

Listing 12.4 The EmailSet object class, some methods postponed

```
public class EmailSet
{
   public static final int MAX = 50;
   ////////////////////////////////
   private int[][] itsItem = new int[MAX][MAX];
            // all components of the array are initially zero
   private int itsSize = 0;
            // non-zero entries are indexed 0..itsSize-1
   private int itsNumEmails = 0; // total of all non-zero entries


   /** Add the given message to the set of stored emails. */

   public void add (Message given)
   {  // logic to find <from,to> codes for sender and receiver
      itsItem[from][to]++;
      itsNumEmails++;
   }  //=====================

   /** Return the total number of emails sent. */

   public int getNumEmails()
   {  return itsNumEmails;
   }  //=====================

   /** Return the total number of employees sending email. */

   public int getNumEmployees()
   {  return itsSize;
   }  //=====================

   /** Return the number of emails sent from from to to. */

   public int numSent (int from, int to)
   {  return itsItem[from][to];
   }  //=====================
}
```

Defining multi-dimensional arrays

You declare an N-dimensional array the same way as a one-dimensional array, except that you give N empty pairs of brackets where you would otherwise give just one. In the constructor phrase of the new array, you give N pairs of brackets, each filled with the number of different index values you want to use in that position.

Examples: To declare and construct a 2-dimensional array to record a money amount owed to any of 7 different loan companies by any of 20 students, you could use this:

```
double [] [] amountOwed = new double [7] [20];
```

So `amountOwed[5][2]` is the amount owed to loan company #5 by student #2, numbering from 0 on up. Figure 12.4 shows an illustration of this.

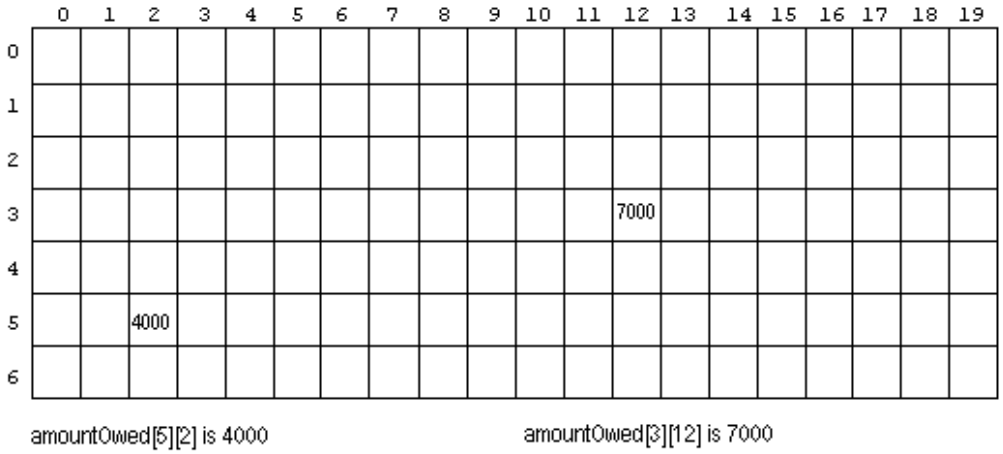

Figure 12.4 Two-dimensional array of numbers

To record a Color value for each pixel in a display 300 pixels wide and 400 pixels tall, you could use this:

```
Color [] [] itsColor = new Color [300] [400];
```

To record whether any of 12 different computer professionals are competent in the use any of 30 different pieces of software, you could use this:

```
boolean [] [] isCompetent = new boolean [12] [30];
```

To record the character in any of 40 different books on a bookshelf, on any of up to 300 pages in the book, on any of the 45 different lines on that page, in any of the 65 different character positions (reading left to right) on that line, you could use this:

```
char [] [] [] [] letter = new char [40] [300] [45] [65];
```

However, Java programmers virtually never use an array of more than two dimensions. It is almost always clearer to have a one- or two-dimensional array of objects that embody several more dimensions. For instance, a Color is actually a sequence of three int values ranging 0 to 255, each representing an RGB value (red/green/blue), so that the `itsColor` array declared above is actually a 5-dimensional array in disguise.

The book example could be done as follows, where a `bookshelf` is a 2-dimensional array of Pages, and a Page is an object that contains a rectangular array of letters:

```
Page[][] bookshelf = new Page[40][300];
class Page { private char[][] letter = new char[45][65];...}
```

Applications to EmailSet objects

All indexes in all Java arrays start from 0 and go on up. An EmailSet object stores the number of emails sent from person `from` to person `to` in the variable `itsItem[from][to]`, part of a two-dimensional array of ints. The following EmailSet methods all adapt logic you have seen before with one-dimensional arrays or other kinds of sequences. Figure 12.5 shows what this array of int values might contain if there were only five employees.

	0	1	2	3	4
0	3	2	0	6	4
1	5	0	1	8	9
2	7	4	0	2	9
3	0	8	7	0	2
4	4	1	5	8	1

Figure 12.5

You may ask an EmailSet object for the total number of emails sent to a person with a particular code number, which it could answer if it had this method, running through all permissible values of the first index:

```
public int numSentTo (int code)   // in EmailSet
{  int count = 0;
   for (int from = 0;  from < itsSize;  from++)
      count += itsItem[from][code];
   return count;
}  //=====================
```

You could ask an EmailSet object for the smallest number of emails sent from a person with a particular code, which it could answer if it had this method, running through the values of the second index:

```
public int minSentFrom (int code)  // version 1; in EmailSet
{  int min = itsItem[code][0];
   for (int to = 1;  to < itsSize;  to++)
   {  if (min > itsItem[code][to])
         min = itsItem[code][to];
   }
   return min;
}  //=====================
```

Some people find this logic easier to understand when it is written as follows, where the one-dimensional int array variable `oneRow` replaces `itsItem[code]`:

```
public int minSentFrom (int code)  // version 2; in EmailSet
{  int[] oneRow = itsItem[code];  // name the row, for clarity
   int min = oneRow[0];
   for (int to = 1;  to < itsSize;  to++)
   {  if (min > oneRow[to])
         min = oneRow[to];
   }
   return min;
}  //=====================
```

You could ask an EmailSet object for the code of the person who sent the largest number of emails to himself, which it could answer if it had the following method, running along the **diagonal** of the array (where indexes are equal). Note that there is no need to have a separate variable `max` that keeps track of the largest value seen so far (on the diagonal). All you need to track is the row and column number `code` of the largest item.

```
public int codeOfMaxSelfSent()  // in EmailSet
{  int code = 0;
   for (int k = 1;  k < itsSize;  k++)
   {  if (itsItem[code][code] < itsItem[k][k])
         code = k;
   }
   return code;
}  //=====================
```

Note that all four of the preceding methods in this section return zero if `itsSize` is zero, since int array components are always initialized to zero by the runtime system.

You could ask an EmailSet object whether any person sent more than ten emails to any one person, which it could answer if it had this method, called with the parameter 10 and running through all possible combinations of the values of both indexes:

```
public boolean anyMoreThan (int cutoff)  // in EmailSet
{  for (int from = 0;  from < itsSize;  from++)
   {  for (int to = 0;  to < itsSize;  to++)
      {  if (itsItem[from][to] > cutoff)
            return true;
      }
   }
   return false;
}  //=====================
```

Exercise 12.21 Declare and construct a 3-dimensional array representing the names of the three best friends of the 25 different students in five different classrooms. What expression represents the name of the second friend listed of student #12 in classroom #2, numbering students and classrooms from 0 up?

Exercise 12.22 Declare and construct a 2-dimensional array representing the ages of each of two children of each of two people. Then write an expression for the average age of all four children.

Exercise 12.23 Revise the `anyMoreThan` method to ignore emails one sends to oneself.

Exercise 12.24 Write an EmailSet method `public boolean allLessThan (int cutoff)`: The executor tells whether everyone sent less than `cutoff` emails to every other person. Call the `anyMoreThan` method to do most of the work.

Exercise 12.25 (harder) Write an EmailSet method `public int maxSentTo (int code)`: The executor tells the largest number of emails sent to a person with a given code.

Exercise 12.26 (harder) Write an EmailSet method `public int numPeopleSentBy (int code)`: The executor tells the number of people to whom one or more emails were sent by a person with a given code.

Exercise 12.27* Write an EmailSet method `public boolean complex (int code, int cutoff, int num)`: The executor tells whether a given person sent more than `cutoff` emails to at least `num` different people.

Exercise 12.28** Write an EmailSet method `public int howMany (int cutoff, int num)`: The executor tells the number of people who sent more than `cutoff` emails to at least `num` different people.

Exercise 12.29** Write an EmailSet method `public boolean pariah (int cutoff)`: The executor tells whether there is any person who was sent less than `cutoff` emails by all other persons. That is, tell whether there is any column of the rectangular array where all non-diagonal components are less than 5.

Part B Enrichment And Reinforcement

12.6 Implementing The Email Software With A Two-Dimensional Array

The discussion so far leads to the implementation of the Email software shown in Listing 12.5. Each line of the main logic design translates to just a few Java statements except for printing the statistics. So a `printStatistics` method goes in a helper class EmailOp to carry out that task separately.

The program can be run by entering e.g.

```
java EmailStats email09Sep02.dat results.dat
```

as the command line in the terminal window. That passes the two file names to the program in the command line arguments `args[0]` and `args[1]`. Since it is easy for a user to occasionally forget to enter one or both file names, the program begins by verifying that it has at least two String values for the file names. If it does not, it prints an appropriate message and terminates the program.

Listing 12.5 Application program for the Email software

```
import java.io.*;

public class EmailStats
{
   /** Read one day's email file and print several usage
    *  statistics about it. */

   public static void main (String[] args) throws IOException
   {
      if (args.length < 2)
         System.out.println ("Specify the two file names");
      else
      {
         // CONSTRUCT THE EMAIL FILE AND THE DATABASE
         BufferedReader inputFile = new BufferedReader
                                    (new FileReader (args[0]));
         EmailSet database = new EmailSet();

         // READ THE EMAIL FILE AND STORE IT IN THE DATABASE
         Message data = new Message (inputFile);
         while (data.getSender() != null)
         {  database.add (data);
            data = new Message (inputFile);
         }

         // PRINT THE RESULTS TO THE OUTPUT FILE
         if (database.getNumEmployees() > 0)
            EmailOp.printStatistics (database, new PrintWriter
                                     (new FileWriter (args[1])));
      }
   }  //=====================
}
```

Reading the Messages uses the standard sentinel-controlled loop: You must check each Message read to see if it signals the end-of-file condition was present when it tried to get the data. As long as it was not, add it to the database. When it signals end-of-file, the program can stop reading values and start printing results.

The `printStatistics` method call has two parameters, the database and the output file, because the database provides the results and the output file accepts the output. The next thing to be done is to design the logic for the `printStatistics` method in the EmailOp class. In the process, you will see what additional methods the EmailSet class needs.

The printStatistics method

Statistics to be printed include the number of emails sent, the number of people who sent them, and the average sent per person. The earlier Listing 12.4 has EmailSet methods that provide the information needed to calculate these statistics.

You also need to go through the employees one at a time (code numbers going from 0 up to the number of employees) and print out for each one whether he did not send any email. If he sent email, you need to print out whether he received at least twice what he sent, or he sent at least twice what he received, and whether he ever sent email to himself.

To obtain this information, you need to be able to ask an EmailSet for the name of the employee with a given code number and for the number of emails sent to or from a given code number. Call the former method `findEmployee`. You should calculate the ratio `db.numSentTo(k)/db.numSentFrom(k)` and then print `db.findEmployee(k)` if that ratio is at least 2.0 or at most 0.5. Listing 12.6 (see next page) contains an implementation of this logic for the `printStatistics` method.

Programming Style: The `printStatistics` method contains three cases in which a value is obtained from the database by a method call and stored in a simple variable to make the execution faster or the logic clearer. Two cases are `size = db.getNumEmployees()` and `largest = db.mostSent()`, to avoid making the method calls many times during the for-loops. The general principle is that, when you retrieve the same value three or more times, you should usually retrieve it once and remember it. The other case is `name = db.findEmployee(k)`. It does not save significant time, but it seems clearer to use a simple name in four places rather than the method call.

The EmailSet class

The most important addition to make to the EmailSet class is a way of shifting back and forth between the name (email address) of an employee and the code number used to store information about him in the database. When you add a given Message to the data base, you have to find the code that corresponds to `given.getSender()` and `given.getReceiver()`. And when `printStatistics` has information about a code number, it wants `findEmployee(k)`, the name corresponding to `k`, for its output.

An elementary way of doing this is to have each EmailSet object maintain another array of names entered so far. Call it `itsName`. The first name entered goes in `itsName[0]`, and 0 is its code. The second name entered goes in `itsName[1]`, and 1 is its code, and so forth. You can keep track of the total number of different names entered so far in a variable named `itsSize` (P.S. You know, don't you, that you can choose any name you like, such as `ralph`? It is just that `itsSize` is so much more informative).

Listing 12.6 The EmailOp helper class

```
class EmailOp   // helper class for Email
{
   /** Print the statistics about email usage.  Precondition:
    *  db and out are non-null, and db is not empty. */

   public static void printStatistics (EmailSet db,
                                       PrintWriter out)
   {  // PRINT OVERALL AVERAGE
      int size = db.getNumEmployees();  // for speed of execution
      out.println (db.getNumEmails() + " email sent by "
               + size + " people; average sent per person = "
               + (1.0 * db.getNumEmails()) / size);


      // PRINT THOSE WITH HEAVY IMBALANCE IN SENT VS RECEIVED
      for (int k = 0;  k < size;  k++)
      {  String name = db.findEmployee (k);
         if (db.numSentFrom (k) == 0)
            out.println (name + " did not send any email.");
         else
         {  double ratio = 1.0 * db.numSentTo (k)
                               / db.numSentFrom (k);
            if (ratio >= 2.0)
               out.println (name + " received twice the sent.");
            else if (ratio <= 0.5)
               out.println (name + " sent twice the received.");
            if (db.numSent (k, k) > 0)
               out.println (name + " sent himself email.");
         }
      }


      // PRINT THOSE WHO SENT THE MOST AMOUNT OF EMAIL
      int largest = db.mostSent();
      for (int k = 0;  k < size;  k++)
      {  if (db.numSentFrom (k) == largest)
            out.println (db.findEmployee (k) + " sent most.");
      }
      out.close();
   }  //====================
}
```

When you get a Message object, you need to search for each of the sender and the receiver of the message. So you call a private method `findCode(employee)` to ask the EmailSet object to search through the `itsName` array to see if the employee (a String value) is already in it. If so, it returns the index where it found that employee. If not, it adds the employee at the end of the list in `itsName[itsSize]` and returns that index, after incrementing `itsSize` by 1.

The public `findEmployee` method that the application uses need only return `itsName[code]`. The `mostSent` method has to find the code of the person who sent the most amount of email. It can go through each of the code values to find the value of `numSentFrom(k)` that is largest. This uses the standard logic for finding the maximum value in an array.

The finished EmailSet class is in Listing 12.7, except for the parts already presented in the earlier Listing 12.4 and the `numSentTo` method which is in the next section. Two defects left to be fixed in the exercises are (a) the `mostSent` method is unnecessarily slow in execution and (b) the program crashes if you have more than MAX employees.

Listing 12.7 The rest of the EmailSet class except numSentTo

```
// public class EmailSet continued

   private int[][] itsItem = new int[MAX][MAX]; // initially zero
   private String[] itsName = new String[MAX];
   private int itsSize = 0;
   private int itsNumEmails = 0;


   public void add (Message given)
   {  int from = findCode (given.getSender());
      int to = findCode (given.getReceiver());
      itsItem[from][to]++;
      itsNumEmails++;
   }  //====================


   /** Precondition:  0 <= code < getNumEmployees() */

   public String findEmployee (int code)
   {  return itsName[code];
   }  //====================


   private int findCode (String employee)
   {  for (int k = 0;  k < itsSize;  k++)
      {  if (employee.equals (itsName[k]))
            return k;
      }
      itsName[itsSize] = employee;
      itsSize++;
      return itsSize - 1;
   }  //====================


   public int numSentFrom (int code)
   {  int count = 0;
      for (int k = 0;  k < itsSize;  k++)
         count += itsItem[code][k];
      return count;
   }  //====================


   public int mostSent()
   {  int max = numSentFrom (0);
      for (int k = 1;  k < itsSize;  k++)
      {  if (max < numSentFrom (k))
            max = numSentFrom (k);
      }
      return max;
   }  //====================
```

Exercise 12.30 How would you rewrite Listing 12.5 and Listing 12.6 so that the PrintWriter is created within the `printStatistics` method rather than on the way to it?
Exercise 12.31 Rewrite the `mostSent` method in Listing 12.7 to avoid two calls to the same method, since that method executes a for-statement on each call.
Exercise 12.32* The variable `name` is mentioned four times in the `printStatistics` method after it is declared. How many times is it actually evaluated on each iteration?
Exercise 12.33* Perhaps the employer would at times like a shortened version of the statistics, giving only the total number sent and the average sent per person. This is to be indicated by a third command-line argument. What would you revise in Listing 12.5 and Listing 12.6 to terminate the `printStatistics` method after the one line is printed whenever the command line contains at least one more argument?
Exercise 12.34* What changes would be required in Listing 12.6 and Listing 12.7 so that the software reports those who sent the least amount of email rather than the most?
Exercise 12.35* Draw the UML class diagram for the EmailStats class.
Exercise 12.36* Draw the UML class diagram for the EmailOp class.
Exercise 12.37** Revise the Email software to terminate gracefully when there are more than MAX employees. As it is now, it crashes with an ArrayIndexOutOfBoundsException in the `findCode` method.

12.7 Using A Two-Dimensional Array Of Airline Data

Suppose you are working on software to handle a database of airline flights between various cities. Each city has a code number from 0 up through `MAX-1`, where `MAX` is a positive integer. An appropriate data structure for this information is a two-dimensional array of Flight objects stored in RAM:

```
private Flight [][] plane = new Flight [MAX][MAX];
```

So `plane[dep][arr]` is one component of this array. It is a description of the airplane flight departing from the city with code `dep` and arriving in the city with code `arr`. If that particular city-to-city combination does not have a direct flight, then null is stored in the array at that component.

The Flight class could have the following instance variables. Note that all are `final`, which means that, once the constructor assigns a value, it cannot be changed. The objects are immutable. So it does not violate encapsulation principles (and it is convenient) to make the instance variables all public:

```
// public class Flight:  five instance variables
   public final String aircraftType;
   public final double ticketPrice;
   public final int    numSeats; // available unreserved
   public final int    numReservations;
   public final long   filePosition;
```

These instance variables are self-explanatory except `filePosition`, which is explained shortly. This Flight class has one constructor and no other methods. The constructor simply initializes all fields:

```
public Flight (String type, double price, int seats,
               int reservations, long position)
{  aircraftType = type;
   ticketPrice = price;
   numSeats = seats;
   numReservations = reservations;
   filePosition = position;
}  //=====================
```

Suppose you have a FlightSet class with the `plane` array as one instance variable, a 2-dimensional array of Flight objects. The logic for a method to find the cheapest ticket price from any city to any other city could then be as follows, relying on the quite reasonable assumption that there must be at least one flight with a price under a million dollars: (1) Set `smallestSoFar` to be a million dollars. (2) Inspect each Flight object in the database and, wherever you see one that is not `null` and has a smaller ticket price than the value stored in `smallestSoFar`, replace that value. (3) return the value of `smallestSoFar`. Listing 12.8 contains this logic.

This technique of initializing `smallestSoFar` to a ridiculously high value, to assure it is replaced by the first real piece of data, is not as efficient and not as robust as the usual technique of initializing it to the first value in the data set. But the latter technique does not work as well when several values can be null, as in this example.

Listing 12.8 The FlightSet class, but with only one of its methods

```
public class FlightSet
{
   public static final int MAX = 20;
   private Flight[][] plane = new Flight[MAX][MAX];


   /** Return the fare for the flight with the lowest fare.
    *  However, return any value if there are no flights. */

   public double cheapestFare()
   {  double smallestSoFar = 1000000.0;     // a million dollars
      for (int dep = 0;  dep < MAX;  dep++)
      {  for (int arr = 0;  arr < MAX;  arr++)
         {  if (plane[dep][arr] != null
                  && plane[dep][arr].ticketPrice < smallestSoFar)
               smallestSoFar = plane[dep][arr].ticketPrice;
         }
      }
      return smallestSoFar;
   }  //=====================
}
```

Exercise 12.38 Write a FlightSet method `public int howManyArriveAt (int city)`: The executor tells how many flights arrive at a given city.
Exercise 12.39 Write a FlightSet method `public int numSeatsAvailableFrom (int city)`: The executor tells the total number of seats available (unreserved) out of a given city to all the rest of the cities.
Exercise 12.40 (harder) Write a FlightSet method `public int flightsOver()`: The executor tells the total number of flights that are more than half full.
Exercise 12.41 (harder) Write a FlightSet method `public int roundTrips()`: The executor tells how in many cases two different cities have direct flights both ways.
Exercise 12.42* Write a FlightSet method `public boolean fairlyDirect (int fromCity, int toCity)`: The executor tells whether you can get from one given city to another given city either directly or with one stopover.
Exercise 12.43** Write a FlightSet method `public void costlyDirect()`: The executor prints out all cases in which it is more expensive to fly directly from a city to another city than it is to go by way of some other city as the one stopover.
Exercise 12.44** Write a FlightSet method `public int planeTypes()`: The executor tells the number of different airplane types, computed over all flights. Hint: Study what `findCode` does in Listing 12.7.

12.8 The RandomAccessFile Class

The airline data described in the previous section is kept in an array in RAM because it is far cheaper in terms of execution time to answer requests for information using the array rather than reading it from a data file. But you have to worry that someone could kick the plug or spill a drink on the computer, thereby losing all your data. So each time a change is made in the data stored in RAM (in the `plane` array), this data should be updated in a binary file on the hard disk.

A **binary file** is one that stores values in the form that they have in RAM, not in the form that they appear on the screen. For instance, the int values 3 and 47312 are one character and five characters, respectively, when written on the screen or stored in a String value. But in RAM, every int value takes up exactly four bytes of space. Writing information in binary form executes much faster than writing it in textual form.

The RandomAccessFile class

The FlightSet class could define a binary file instance variable as follows:

```
java.io.RandomAccessFile raf = new java.io.RandomAccessFile
                      ("flight.data", "rw");
```

Random-access means that you can read from or write to any point in the file without first going through all the values at the beginning of the file, as you have to do with a sequential file. `"rw"` signals that you can both read and write with this particular file; the other choice for the second parameter is `"r"` for a read-only file. You can write all the basic kinds of values to the file if you use the appropriate method, e.g.:

```
raf.writeDouble(-3.72)      // write 8 bytes
raf.writeInt(47)            // write 4 bytes
raf.writeChar('x')          // write 2 bytes, Unicode
raf.writeChars("test case") // write that many 2-byte chars
raf.writeBoolean (done)     // write 1 byte (1==true)
```

You also have the corresponding methods for reading values, except that the String input is somewhat different: The `readLine` method only handles the basic Unicode values 0 to 255 well; it stops reading when it read in a newline or end-of-file, but it does not include the newline character in the String value returned. The following all return the value of the type specified:

```
raf.readDouble()    // read 8 bytes
raf.readInt()       // read 4 bytes
raf.readChar()      // read 2 bytes, Unicode
raf.readLine()      // read to newline or end-of-file
raf.readBoolean()   // read 1 byte; 0 is false, others true
```

Of course, you have the corresponding input and output methods for bytes, shorts, floats, and longs. The only other RandomAccessFile methods you need for now are as follows:

```
raf.close()  // disconnect when done with the file
raf.length() // return the long number of bytes in the file
raf.seek(n)  // set the point where reading or writing next
             // takes place.  n is a long value.
```

`raf.seek(n)` sets the read/write point, called the **file-pointer position**, at the byte of that number, numbering from 0 on up as always. So `raf.seek(1000);` `raf.writeInt(47)` writes four byte values at positions 1000, 1001, 1002, and 1003.

All the RandomAccessFile methods can throw an IOException or a subclass of IOException. Figure 12.6 illustrates the use of these methods.

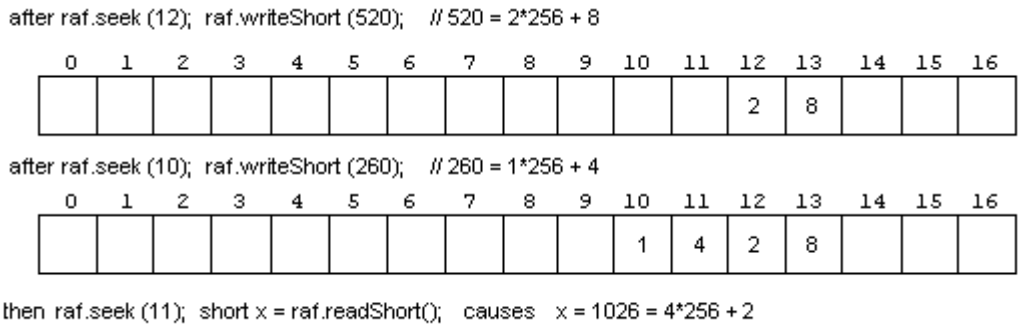

Figure 12.6 illustration of reading and writing with a random-access file

Now the purpose of the `filePosition` field in a Flight record should be clear. When you modify the data stored in a particular component of the `plane` array (replacing the existing Flight record by a new one), you backup the RAM data as follows:

```
public void writeRecord (Flight flit) throws IOException
{  raf.seek (flit.filePosition);
   raf.writeChars (flit.aircraftType + "\n");
   raf.writeDouble (flit.ticketPrice);
   raf.writeInt (flit.numSeats);
   raf.writeInt (flit.numReservations);
}  //======================
```

Each time you write to a random-access file, the file-pointer position moves forward by the number of bytes written. That is why the `writeRecord` method works without having the int values overwrite the chars and double. The same forward motion happens with reading from a random-access file.

Writing beyond the end of the file is allowed; it extends the file to that point. Reading beyond the end of the file throws an IOException of some kind. You may find the current position of the file pointer by calling `raf.getFilePointer()`; it returns a long value.

Categories of standard file classes

The Sun standard library has six basic groups of classes of file objects, all in `java.io`:

1. Reader is an abstract class whose many subclasses read into arrays of chars.
2. Writer is an abstract class whose many subclasses write from arrays of chars.
3. InputStream is an abstract class whose many subclasses read into arrays of bytes.
4. OutputStream is an abstract class whose many subclasses write from arrays of bytes.
5. StreamTokenizer is a concrete class for treating a text file as a Java program.
6. RandomAccessFile is a concrete class for random access of a text file.

Exercise 12.45 Write a FlightSet method `public void readRecord (Flight flit, long filePosition)`: The executor reads a record from the random access file `raf` at a particular file position into a particular Flight variable. The record was written by the `writeRecord` method coded in this section.

Exercise 12.46* Write an independent method `public static swapTwo (long filePosition)`: It reads two consecutive double values from the random-access file `raf` at a given file position and swaps the two values in the file.

12.9 How Buffering Is Done

Pretend for a bit that you do not have BufferedReader, only the InputStreamReader class and its subclass FileReader. The purpose of this exercise is twofold: (a) You will understand the separate roles of the BufferedReader class and the other two classes, and (b) you will develop more competence with arrays, particularly char arrays.

A structure that occurs frequently in the Sun standard library is an array of characters. This can be used to store one character in each component. A text file, for instance, is just a long sequence of characters. If it is not overly large, you can read it into a single char array and process the information in it.

FileReader is a subclass of InputStreamReader that adds only some constructors. This is to make opening a file a one-step process rather than a two- or three-step process. The main non-constructor methods you have in InputStreamReader (and thus in FileReader by inheritance) are as follows. All three can throw an IOException:

- `int read()` obtains a single character from the file. It returns -1 at end-of-file.
- `int read(char[] cbuf, int off, int len)` attempts to read `len` characters into the char array `cbuf` starting at component `off`. It returns the number of characters read, except it returns -1 if it previously reached end-of-file.
- `void close()` disconnects from the file.

You can create your own class to read values from the file 8192 characters at a time and respond to `readLine` requests. This Buffy class, shown in Listing 12.9 (see next page), has five instance variables. The `itsBuf` array is to hold 8192 characters you read from the `itsInput` file, or less if you have reached the end of the file. The `itsSize` variable keeps track of the number of useable characters in the array; this will be 8192 unless you have reached the end of the file. The `itsNextChar` variable keeps track of the index of the next available character in the array. So the characters indexed `itsNextChar... itsSize-1` have been read from the file but not "consumed" by `readLine` yet. The logic for `readLine` is rather complex, so you need to design it carefully. The accompanying design block gives a reasonable plan.

STRUCTURED NATURAL LANGUAGE DESIGN for the readLine method
If the number of "unconsumed" characters in the buffer gets below 2000 and
there are more characters in the file then...
    Move the ones you have to the front of the array.
    Go get some more characters from the file.
If you are still out of characters then...
    Return null as the answer to the readLine query.
Record in `start` the current position in the buffering array.
Count the number of characters down to the next newline character `'\n'`.
Return a String consisting of that many characters starting from `start`.

You also need a private method to fill in the buffer starting from some particular index; you could call it using `fillBufferStartingAt(front)`. It uses the `read` method from InputStreamReader to get as much as possible into the array. A call of the `read` method must supply the char array to be filled in, the first index where the filling is to begin, and the number of characters to fill in. The array will be filled completely (`itsSize == MAX`) unless you get to the end of the file first.

The logic for `readLine` uses the String constructor that accepts a slice of a char array: `new String (itsBuf, start, count)`. This logic is simplified by the assumption that no line has more than 2000 characters. It is an exercise to remove this assumption.

Listing 12.9 The Buffy class

```
import java.io.Reader;
import java.io.IOException;

public class Buffy
{
   public static final int MAX = 8192;
   ////////////////////////////////
   private Reader itsInput;        // the file or keyboard input
   private char[] itsBuf;          // chars read but not used
   private int itsSize;            // number of chars stored
   private int itsNextChar;        // position of next char
   private boolean itsAtEndOfFile; // true when no more to read


   public Buffy (Reader given) throws IOException
   {  itsInput = given;
      itsBuf = new char[MAX];
      fillBufferStartingAt (0); // get the first characters
   }  //======================

   private void fillBufferStartingAt (int loc) throws IOException
   {  itsSize = loc + itsInput.read (itsBuf, loc, MAX - loc);
      itsAtEndOfFile = itsSize < MAX;
      if (itsAtEndOfFile)
         itsInput.close();
      itsNextChar = 0;
   }  //======================

   public String readLine() throws IOException
   {  if (itsNextChar >= itsSize - 2000 && ! itsAtEndOfFile)
         downShiftAndFill();
      if (itsNextChar >= itsSize)         // no more characters
         return null;
      int start = itsNextChar;
      int count = 0;
      while (itsBuf[start + count] != '\n')
         count++;
      itsNextChar = start + count + 1;   // prepare for next call
      return new String (itsBuf, start, count);
   }  //======================

   private void downShiftAndFill() throws IOException
   {  for (int k = itsNextChar;  k < itsSize;  k++)
         itsBuf[k - itsNextChar] = itsBuf[k];
      itsSize -= itsNextChar;
      fillBufferStartingAt (itsSize);
   }  //======================
}
```

Exercise 12.47* It is actually possible for the last line in the file to not have a newline character at the end of it. Revise the condition of the while-statement in Listing 12.9 to prevent a wrong result in such a case.

Exercise 12.48** Revise the `readLine` method in Listing 12.9 so it works no matter how many characters a line has. However, insert a newline character into any line with more than 8192 characters, to break it up.

*12.10 Additional Java Language Features (*Enrichment)*

The following are features of the Java language that do not appear elsewhere in this book, but you may find them useful in your software development.

Visibility modifiers

You have only seen the two visibility modifiers public and private. Java has four levels of visibility: public, private, protected, and default. A member of a class can be marked as public, private, or protected; default visibility applies when you do not use any of them.

- **public visibility** of a member of class X: Any statement anywhere can mention the member.
- **protected visibility** of a member of class X: Only statements in the same package with X or in a subclass of X can mention the member.
- **default visibility** of a member of class X: Only statements in the same package with X can mention the member.
- **private visibility** of a member of class X: Only statements that are within X can mention the member.

Bitwise operators

Each byte value is a sequence of 8 bits and each int value is a sequence of 32 bits. Java provides several operators to work on the individual bits of byte and int values, as well as the bits of char, short, and long values. One is the **bitwise-and**, for which the symbol is `&`. This operator produces the value that has a 1-bit in each position where both of the two operands have a 1-bit, and a 0-bit in every other position. For instance, `13 & 10` is 8, because the binary notation is 00001101 for 13 and 00001010 for 10, and thus the result is 00001000 for 8.

The **bitwise-or** operator is `|`, which produces the value that has a 1-bit in each position where either one of the two operands has a 1-bit, and a 0-bit in every other position. For instance, `13 | 10` is 15, because the result is 00001111 for 15.

The **bitwise-negation** operator is `~`, which produces the value that has a 1-bit wherever the single operand has a 0-bit, and a 0-bit wherever the single operand has a 1-bit. For instance, `~13` is 242, because the result of "inverting" 00001101 for 13 is 11110010 for 242. This applies if 13 is stored in a byte variable. In general, the bitwise-negation of a byte value is 255 minus that byte value, and the bitwise-negation of a short value (16 bits; note that 2^{16} `- 1` is 65535) is 65535 minus that short value.

The **bitwise-exclusive-or** operator is `^`, which produces the value that has a 1-bit in each position where exactly one of the two operands has a 1-bit, and a 0-bit in every other position. For instance, `13 ^ 10` is 00001101 ^ 00001010 in binary, so the answer is 00000111 in binary, which is 7. In summary: Since 13 is 8+4+1 and 10 is 8+2, `13 & 10` is 8, `13 | 10` is 8+4+2+1, and `13 ^ 10` is 4+2+1.

The **left-shift** operator is <<; `x << n` shifts all the bits of x to the left by n places, filling in zeros in the places left empty. So `13 << 1` is 26 (00011010) and `13 << 2` is 52 (00110100). In general, a left-shift by n places multiplies the number by 2^n. The **right-shift** operator works as follows: `x >>> n` shifts all the bits of x to the right by n places, filling in zeros in the places left empty. 26 is 00011010 in binary, so `26 >>> 1` is 13 (00001101) and `26 >>> 2` is 6 (00000110, since the last bit "drops off the end"). In general, a right-shift by n places divides by 2^n, dropping the remainder.

Other stuff you can do in Java

- You may declare a parameter of a method as `final`, which means that the coding in the method cannot change its value.
- You may have one method whose heading is simply `static`. This initializer class method will be executed before any user of the class calls one of its methods.

Exercise 12.49* Calculate `22 & 12`, `22 | 12`, and `22 ^ 12`.
Exercise 12.50* Calculate `47 << 1`, `47 << 2`, `47 >>> 1` and `47 >>> 2`.

*12.11 Java Bytecode Commands (*Enrichment)*

The `.class` file contains primarily **bytecode**. This is a sequence of byte values to be executed by the runtime system, carrying out the instructions in the various methods of the class. The bytecode sequence for each method consists of a number of commands. Each **command** is a single byte, called the **opcode**, followed by a number of bytes, called the **operands** of the opcode. The **Java Virtual Machine** (**JVM**) executes these commands one at a time.

An **opcode** can be any of the numbers 0 through 255 (since a byte is eight bits, and 2^8 is 256). For instance, the command to add two double values is 99 and the command to subtract one int value from another is 100. Since the numbers are difficult to remember, we use a standard set of verbal clues in their place. The **mnemonic code** for 99 is `dadd` and the mnemonic code for 100 is `isub`. In general, mnemonic codes starting with `i` operate on int values and those starting with `d` operate on double values.

The number of bytes in the command depends on the particular opcode. Some opcodes require no operands at all, others require one or two, each of which might be 1, 2, 4, or 8 bytes long.

The operand stack

Each method has an **operand stack** for doing computations. When a method is called, it becomes the **current method** and its operand stack is created. When the current method finishes executing, the method that called it becomes the current method and takes up where it left off, with its own operand stack.

Some commands add values on the top of the stack, others take values off the top of the stack. The stack is a LIFO structure -- last-in-first-out. So if you add two values to the stack and then remove one, you will get the second one that you added.

The JVM typically numbers the parameters of a method in order starting from 0, then applies the next few numbers to the variables declared inside the method. As an example, consider the following method:

```
public static int diff (int x, int y)  // independent
{  int answer = x - y;
   return 2 * answer;
}  //=====================
```

The body of this method is translated into bytecode as a sequence of only 12 bytes, as you will see shortly. The three variables are given the index numbers 0 for `x`, 1 for `y`, and 2 for `answer`. Then the first statement is translated to bytecode as follows:

- `iload 0` (numerically, 21 0) is the command to load the value of local #0, which is `x`, onto the operand stack.
- `iload 1` (numerically, 21 1) is the command to load the value of local #1, which is `y`, onto the operand stack. So far, the stack contains the value of `y` on top of the value of `x`.
- `isub` (numerically, 100) is the command to pop off the top two values of the operand stack and push back onto the stack the result of subtracting the top value from the second-from-top value. So now the operand stack contains only one value, `x - y`.
- `istore 2` (numerically, 54 2) is the command to pop off the top value from the operand stack and store it in local #2, which is `answer`. Now the stack contains no values at all, but the variable named `answer` contains the value of `x - y`.

What actually appears in the bytecode to implement the first statement is therefore this sequence of seven bytes: `21 0 21 1 100 54 2`.

Standard bytecode descriptions

A compact but fairly clear description of the bytecode commands mentioned above is as follows. In these descriptions, the phrase "causes..." shows how the stack changes, e.g., for `iload`, an int value is added to the top of the stack without changing what is already there. The part before the ==> is what the stack looks like before the command is executed, the part after the ==> is what the stack looks like after the command is executed, and the ellipsis `...` indicates the part of the stack that remains unchanged.

`iload` `(21) byteVal` causes ... ==> ... intVal
// The current value stored in local variable #byteVal is pushed onto the stack.
`istore` `(54) byteVal` causes ... intVal ==> ...
// The int value on top of the stack is popped and stored into local variable #byteVal.
`isub` `(100)` causes ... intVal1 intVal2 ==> ... intVal3
// The two int values on top of the stack are popped, and intVal1 - intVal2 is pushed.

Some additional bytecode commands needed for the `diff` method are as follows:

`iconst_2` `(5)` causes ... ==> ... intVal
// The constant int value of 2 is pushed onto the stack.
`imul` `(104)` causes ... intVal1 intVal2 ==> ... intVal3
// The two int values on top of the stack are popped, and intVal1 * intVal2 is pushed.
`ireturn` `(172)` causes ... intVal ==>
// The int value on top of the stack is popped and returned to the calling method.
// The operand stack of the current method is discarded, and the returned value is
// pushed onto the operand stack of the calling method, which now becomes the
// current method.

Now the second statement in the `diff` method can be translated as

```
iconst_2;  iload 2;  imul;  ireturn;
```

which numerically is this sequence of five bytes: `5 21 2 104 172`. Other byte codes whose meaning should now be clear are as follows:

`iadd` `(96)` causes ... intVal1 intVal2 ==> ... intVal3
// The two int values on top of the stack are popped, and intVal1 + intVal2 is pushed.
`idiv` `(108)` causes ... intVal1 intVal2 ==> ... intVal3
// The two int values on top of the stack are popped, and intVal1 / intVal2 is pushed.
`irem` `(112)` causes ... intVal1 intVal2 ==> ... intVal3
// The two int values on top of the stack are popped, and intVal1 % intVal2 is pushed.

iconst_1 (4) causes ... ==> ... intVal
// The constant int value of 1 is pushed onto the stack.
iconst_0 (3) causes ... ==> ... intVal
// The constant int value of 0 is pushed onto the stack.
iconst_m1 (2) causes ... ==> ... intVal
// The constant int value of -1 is pushed onto the stack.

Branching instructions

You know that the if-statement and while-statement and others can cause execution to jump from one instruction to another, skipping those in between. This is implemented in bytecode by a **branching instruction**. The main one is `goto shortVal`, where shortVal is a two-byte whole number in the range from -32768 to +32767. The effect of `goto +20` is that the next command executed is the one that is 20 bytes forward of the goto byte in the bytecode sequence. The effect of `goto -52` is that the next command executed is the one that is 52 bytes before the goto byte in the bytecode sequence. In both cases, the new position must be the first byte in a command within the same method.

You also have six bytecode commands for a **conditional branch**. The mnemonic codes for these six are all the same except for the last two characters, which tell what kind of comparison is to be made. For instance `ifeq +80` means to pop off the top int value on the operand stack and, if it equals zero, branch forward 80 bytes from the `ifeq` byte; and `iflt -40` means to pop off the top int value on the operand stack and, if it is less than zero, branch backward 40 bytes from the `iflt` byte. The standard descriptions of these byte codes are as follows:

goto (167) shortVal causes ... ==> ...
// The stack is not changed. Branch shortVal bytes away, forward or backward.
ifeq (153) shortVal causes ... intVal ==> ...
// The int value is popped, and if intVal == 0 then branch shortVal bytes away.
iflt (155) shortVal causes ... intVal ==> ...
// The int value is popped, and if intVal < 0 then branch shortVal bytes away.

The bytecode also has `ifne` (154), `ifge` (156), `ifgt` (157), and `ifle` (158). Now consider the following method to find the smaller of two int values:

```
public static int smaller (int x, int y)  // independent
{  int answer;
   if (x >= y)
      answer = y;
   else
      answer = x;
   return answer;
}  //=====================
```

Remember than x is local variable #0, y is local variable #1, and answer is local variable #2. The mnemonic codes for this method would be as follows:

```
iload 0;  iload 1;  isub;  ifge +10;
iload 1;  istore 2;
goto +7;
iload 0;  istore 2;
iload 2;  ireturn;
```

You are probably wondering how the +10 and +7 could be calculated before the rest of the bytecode commands were written. The answer is, they could not be. You leave the **offset** amount (+10 or +7 in this case) blank until you have written enough more of the

bytecode that you can count the bytes to see how much it should be. Remember when you count that each offset amount takes up two bytes.

This section has presented only 20 of the 255 possible bytecodes. Some perform operations with float, byte, short, and long values; and some perform operations with object and array references. To see the other possibilities, you may look at `http://java.sun.com/docs/books/vmspec/html/VMSpecTOC.doc.html`, particularly the section titled "The Java Virtual Machine Instruction Set".

Assembly language

An **assembly language** is basically machine code with mnemonics for operands and other things, as well as opcodes. For instance, an assembly language for Java bytecode could specify these memory aids:

`int x, y, answer` means that `x` represents local #0, `y` represents local #1, `answer` represents local #2, and similarly wherever a byteValue operand is called for. `double u, v, result` means that `u` represents local #0, `v` represents local #2, `result` represents local #4, and similarly wherever a byteValue operand is called for (since each double value requires the space of two int values).

The counter is assumed to continue with further such "declarations"; so if both of those declarations appeared in a single method in that order, then `u` would represent local #3, `v` would represent local #5, and `result` would represent local #7.

It is then not too hard to develop an **assembler** program to translate a source file containing such mnemonics to the actual machine code. You would also save yourself the trouble of counting bytes (and re-counting if you make modifications) by specifying that an "instruction" which is some name followed by a colon denotes the location of the immediately following byte; that name could be called a **target**. The assembly program could allow a target to be mentioned directly after any of the seven branching instructions in place of the byteVal offset; the assembly program will compute the offset for you. Then the `smaller` method using double values could be expressed in assembly language as follows (`dcmpg` means compare the two double values on top of the operand stack to see which is larger):

```
            double x, y, answer;
            dload x;  dload y;  dcmpg;  ifge xLarger;
            dload y;  dstore answer;
            goto endif;
   xLarger: dload x;  dstore answer;
   endif:   dload answer;  dreturn;
```

Surely you can see how it would be much easier to write a bytecode program in assembly language and have the assembler make the conversion for you. Programmers almost uniformly wrote all programs in an assembly language up until around 1955, when FORTRAN was invented. Clearly they were hardy souls. Of course, it is even easier to write in Java itself and let the Java compiler make the translation for you.

Exercise 12.51 The assembler language for the `smaller` method for double values could have one or more instructions omitted and still have the same effect. What would the change be?
Exercise 12.52* Write the `diff` method in the assembly language described here.
Exercise 12.53* Write an assembly language method to calculate the greatest common divisor of two positive integer parameters. Hint: Repeatedly subtract the smaller from the larger until they become equal.

*12.12 About Networking Using Sockets (*Sun Library)*

The **java.net.ServerSocket** class has the following useful methods:

- `new ServerSocket(portInt, queueInt)` creates a new socket that can be used as a server for various clients. The `portInt` is the **port number** by which the ServerSocket is identified to its clients; the `portInt` can be any int value from 1024 to 65535. The `queueInt` is the number of clients requesting a connection that it can hold in its backlog queue for later processing while it is already busy with a client. The `queueInt` can be any non-negative int value.
- `someServerSocket.accept()` waits until some client socket asks for a connection (we say that it **blocks** until asked). At that point the method returns the Socket object that requested the connection.
- `someServerSocket.close()` terminates the existence of that ServerSocket.

The **java.net.Socket** class has the following useful methods:

- `new Socket(hostString, portInt)` creates a new socket that is connected to port number `portInt` on the remote host whose internet name is `hostString`.
- `someSocket.getInputStream()` returns an InputStream object for reading bytes from this client.
- `someSocket.getOutputStream()` returns an OutputStream object for writing bytes to this client.
- `someSocket.close()` terminates the existence of that ServerSocket.

All of these methods can throw a `java.io.IOException`. The constructors and the `accept` method can also throw a `java.lang.SecurityException` (which is a RuntimeException).

Example of using Sockets

The best way to understand how all of these methods interact is to see an example. My friend would like all her relatives to be able to suggest names for the baby she is expecting. So she has an account on which she runs the Telnetting application program given in Listing 12.10 (see next page). It starts by creating a Handler object. Then each time some relative telnets to that account, the Handler's `getMore` method executes.

Note that the `getMore` method is completely independent of sockets. It is simply given an input and an output by which it communicates with the relatives. It prints a list of all names suggested so far for the baby, then gets any additional suggestions the relative has and adds them to the list. When my friend wants to know the list so far, she telnets in to the account to see what it displays.

The main method creates a new Handler for the baby names and a new ServerSocket with an arbitrarily-chosen port number of 12345. It allows up to twenty relatives to be queued up waiting to make suggestions (it is a very large extended family). Then a loop executes over and over again until the server is shut down (e.g., by CTRL/C at the terminal).

The loop executes the `accept` method, which blocks execution until some relative telnets in to the server. Then it creates a BufferedReader object that accepts input from the input stream associated with that client, as well as a PrintWriter object that sends output to the output stream associated with that client. After that, it sends these two communication channels to the Handler object so it can do its job, then closes the connection and waits for another connection.

Listing 12.10 The Telnetting application program

```
import java.net.*;
import java.io.*;

public class Telnetting
{
   public static void main (String[] args)
   {  try
      {  Handler birthNames = new Handler();
         ServerSocket server = new ServerSocket (12345, 20);
         for (;;)
         {  Socket client = server.accept();  // blocks
            BufferedReader input = new BufferedReader
                (new InputStreamReader(client.getInputStream()));
            PrintWriter output = new PrintWriter
                                  (client.getOutputStream());
            birthNames.getMore (input, output);
            client.close();
         }
      }catch (IOException e)
      {  System.out.println ("could not open server or socket");
      }
   }  //======================
}
//#############################################################


import java.io.*;

public class Handler
{
   private String itsInfo = "";

   public void getMore (BufferedReader input, PrintWriter output)
   {  try
      {  if (itsInfo.length() > 0)
            output.println ("Suggestions so far:" + itsInfo);
         output.println ("What do you suggest for a baby name?");
         output.flush();
         String data = input.readLine();
         while (data != null && data.length() > 0)
         {  itsInfo += " " + data;
            output.println ("Another suggestion? ENTER to quit");
            output.flush();
            data = input.readLine();
         }
         output.close();
      }catch (IOException e)
      {  // no need for any statements here
      }
   }  //======================
}
```

If this application is running in an account named `sam.ccsu.edu`, all a relative has to do to suggest a baby name is to enter the following command in his or her terminal window:

```
telnet sam.ccsu.edu 12345
```

*12.13 About File And JFileChooser (*Sun Library)*

You can make the user's choice of which file to use more comfortable with a standard library class named **JFileChooser**. This is a subclass of JComponent in the `javax.swing` package. You create a JFileChooser object and show either an Open dialog or a Save dialog. If it returns the APPROVE_OPTION, you can access the file chosen using a statements such as the following:

```
File selected = someJFileChooser.getSelectedFile();
FileReader file = new FileReader (selected);
```

- `new JFileChooser()` creates a JFileChooser object.
- `someJFileChooser.showOpenDialog(null)` returns one of the three int values listed below, after opening a window that allows the user to choose a file to open.
- `someJFileChooser.showSaveDialog(null)` returns one of the three int values listed below, after opening a window that allows the user to save a file.
- `JFileChooser.APPROVE_OPTION` int value indicating the user selected a file.
- `JFileChooser.CANCEL_OPTION` int value indicating the user canceled out.
- `JFileChooser.ERROR_OPTION` int value indicating that some error occurred.
- `someJFileChooser.getSelectedFile()` returns the File value chosen.

A **File** object is not a file itself, only the description of a file including the drive and subfolder. You need to open a FileReader to actually access the file itself. The File class is in the `java.io` package. It allows the following useful operations:

- `someFile.exists()` tells whether there actually is a file of that name and folder.
- `someFile.canRead()` tells whether you have permission to read from this file.
- `someFile.canWrite()` tells whether you have permission to write to this file.
- `someFile.delete()` removes this file from its folder.

12.14 Review Of Chapter Twelve

Listing 12.1, Listing 12.2, and Listing 12.3 illustrate the main new library facilities discussed in this chapter.

About sequential file input and output (all are in java.io):

- A **sequential input file** is a file from which you can only obtain values in the order in which they were written; you cannot jump directly to another input position in the file.
- A **sequential output file** is a file to which you can only write values in order.
- `new BufferedReader(new FileReader(filenameString))` **opens** a text file for sequential input. The FileReader constructor can throw an IOException.
- `someBufferedReader.readLine()` gets one line of input. It can throw an IOException.
- `new BufferedReader(new InputStreamReader(System.in))` opens the keyboard for input. It can throw an IOException.
- `new PrintWriter(new FileWriter(filenameString))` opens a text file for sequential output. Any pre-existing file of the same name is lost, unless you add a second parameter value of `true` to indicate appending. It can throw an IOException.
- `somePrintWriter.flush()` clears out all characters left in the buffer, if any.
- `somePrintWriter.close()` is executed when you are done with the file.
- `somePrintWriter.print(someString)` writes the characters to the file.
- `somePrintWriter.println(someString)` writes the characters to the file followed by a newline.
- `new PrintWriter(System.out)` opens the terminal window for output.

About the java.util.StringTokenizer and java.io.StreamTokenizer classes:

- `new StringTokenizer(inputString)` creates a tokenized form of the String of characters. A **token** is a sequence of non-whitespace characters with whitespace on each side, counting beginning and end of the String as whitespace (but you can customize that definition). StringTokenizer is in the `java.util` package.
- `someStringTokenizer.hasMoreTokens()` tells whether you can get another token from the `inputString` without throwing a RuntimeException.
- `someStringTokenizer.nextToken()` returns the next available token in the `inputString` and advances in the sequence.
- `new StreamTokenizer(someReader)` creates the file processing object.
- `someStreamTokenizer.nextToken()` returns an int value that tells whether the next token is a word, number, string, end-of-file, etc. It skips Java comments. It can throw an IOException.

About random-access file input and output (from java.io):

- `new RandomAccessFile (filenameString, "rw")` opens a file for random-access. Use `"r"` instead of `"rw"` if you will not be writing to it.
- `someRandomAccessFile.writeInt(someInt)` writes one int value.
- `someRandomAccessFile.writeChar(someChar)` writes one char value.
- `someRandomAccessFile.writeDouble(someDouble)` writes one double value.
- `someRandomAccessFile.writeBoolean (someBoolean)` writes one boolean.
- `someRandomAccessFile.writeChars(someString)` writes without a newline.
- `someRandomAccessFile.readInt()` returns an int value.
- `someRandomAccessFile.readChar()` returns a char value.
- `someRandomAccessFile.readDouble()` returns a double value.
- `someRandomAccessFile.readBoolean()` returns a boolean value.
- `someRandomAccessFile.readLine()` returns a String value. It only handles the basic Unicode values 0 to 255 well, stops reading at a newline or end-of-file, and does not include the newline character in the returned value.
- `someRandomAccessFile.length()` returns a long value telling the number of bytes in the file.
- `someRandomAccessFile.seek(someLong)` sets the file pointer value to be the given long value.
- `someRandomAccessFile.close()` is executed when you are done with the file.
- Read through the documentation for the `java.io` package partially described in this chapter. Look at the API documentation at `http://java.sun.com/docs` or on your hard disk.

Other vocabulary to remember:

- An output file is **buffered** if it saves up in RAM the small chunks of data you give it until it has a big chunk of data that it can write to the hard disk or screen all at once. An input file is **buffered** if it gets one big chunk of data at a time from the hard disk and saves it in RAM until you ask for it, typically in small chunks. If you have a reason to move the saved-up output to its ultimate destination before the buffering algorithm feels like doing so, you **flush the buffer**.
- A **binary file** is one that stores values in the form that they have in RAM, not in the textual form that appears on the screen. **Random-access** means that you can read from or write to any point in the file without first going through all the values at the beginning of the file, which is what you have to do with a sequential file.
- The **file-pointer position** is the byte number at which the next read or write operation will take effect in a file. You use it to specify where you will next read or write in a random-access file.

Answers to Selected Exercises

12.1
```
public static void findCaps (String name) throws IOException
{   BufferedReader fi = new BufferedReader (new FileReader (name));
    for (String si = fi.readLine();  si != null;  si = fi.readLine())
    {   if (si.length() > 0 && si.charAt (0) >= 'A' && si.charAt (0) <= 'Z')
            System.out.println (si);
    }
}
```
12.2
```
public int readInt()
{   String si = readLine();
    try
    {   return (si == null)  ?  0  :  Integer.parseInt (si);
    }catch (NumberFormatException e)
    {   return 0;
    }
}
```
12.3
```
public static int numChars (String name) throws IOException
{   BufferedReader file = new BufferedReader (new FileReader (name));
    int count = 0;
    for (String si = file.readLine();  si != null;  si = file.readLine())
        count += si.length();
    return count;
}
```
12.4
```
public static double averageCharsPerLine (String name) throws IOException
{   BufferedReader file = new BufferedReader (new FileReader (name));
    int lines = 0;
    int count = 0;
    for (String si = file.readLine();  si != null;  si = file.readLine())
    {   lines++;
        count += si.length();
    }
    return (lines == 0)  ?  0  :  (1.0 * count) / lines;
}
```
12.5
```
public static boolean descending (String name) throws IOException
{   BufferedReader file = new BufferedReader (new FileReader (name));
    String previous = file.readLine();
    if (previous != null)
    {   for (String si = file.readLine();  si != null;  si = file.readLine())
        {   if (previous.compareTo (si) < 0)
                return false;
            previous = si;
        }
    }
    return true;
}
```
12.10
```
public static void copy (BufferedReader infile, String outName) throws IOException
{   PrintWriter outfile = new PrintWriter (new FileWriter (outName));
    for (String si = infile.readLine();  si != null;  si = infile.readLine())
        outfile.println (si);
    outfile.close();
}
```
12.11
```
public static void under40 (String inName, String outName) throws IOException
{   BufferedReader infile = new BufferedReader (new FileReader (inName));
    PrintWriter outfile = new PrintWriter (new FileWriter (outName));
    for (String si = infile.readLine();  si != null;  si = infile.readLine())
    {   if (si.length() < 40)
            outfile.println (si);
    }
    outfile.close();
}
```
12.16 Replace the line "if (line.hasMoreTokens())" by the following:
```
if (! line.hasMoreTokens())  itsSender = null;  else
```
12.17 Replace the body of the while statement by the following:
```
char ch = data.nextToken().charAt (0);
if (ch >= '0' && ch <= '9')
    count++;
```
12.18 That simpler expression will be true when the character is anything between '0' + 9 and '0' - 9, because integer division by 10 of any number from -1 to -9 yields 0.

12.21
```
String [] [] [] friend = new String [5] [25] [3];
friend [2] [12] [1]
```
12.22
```
int[][] age = new int [2][2].     (age[0][0] + age[0][1] + age[1][0] + age[1][1]) * 1.0 / 4.
```

12.23 Replace the line beginning with "if" by the following:

```
if (itsItem[from][to] > cutoff && from != to)
```

12.24

```
public boolean allLessThan (int cutoff)
{   return ! anyMoreThan (cutoff - 1);
}
```

12.25

```
public int maxSentTo (int code)  // in EmailSet
{   int max = itsItem[0][code];
    for (int from = 1;  from < itsSize;  from++)
    {   if (max < itsItem[from][code])
            max =  itsItem[from][code];
    }
    return max;
}
```

12.26

```
public int numPeopleSentBy (int code)  // in EmailSet
{   int count = 0;
    for (int to = 0;  to < itsSize;  to++)
    {   if (itsItem[code][to] > 0)
            count++;
    }
    return count;
}
```

12.30 Replace the method call in Listing 12.5 by the following:

```
printStatistics (database, args[1]);
```

Replace "PrintWriter out" in the heading of printStatistics by "String outName" and insert the following as the first statement in that method:

```
PrintWriter out = new PrintWriter (new FileOutputstream (outName));
```

12.31 Replace the body of the for-statement by the following:

```
    int numSent = numSentFrom (k);
    if (max < numSent)
        max = numSent;
```

12.38

```
public int howManyArriveAt (int city)
{   int count = 0;
    for (int dep = 0;  dep < MAX;  dep++)
    {   if (plane[dep][city] != null)
            count++;
    }
    return count;
}
```

12.39

```
public int numSeatsAvailableFrom (int city)
{   int count = 0;
    for (int arr = 0;  arr < MAX;  arr++)
    {   if (plane[city][arr] != null)
            count += plane[city][arr].numSeats;
    }
    return count;
}
```

12.40

```
public int flightsOver ()
{   int count = 0;
    for (int dep = 0;  dep < MAX;  dep++)
        for (int arr = 0;  arr < MAX;  arr++)
        {   if (plane[dep][arr] != null
                    && plane[dep][arr].numSeats < plane[dep][arr].numReservations)
                count++;
        }
    return count;
}
```

12.41

```
public int roundTrips()
{   int count = 0;
    for (int dep = 0;  dep < MAX;  dep++)
        for (int arr = dep + 1;  arr < MAX;  arr++)
        {   if (plane[dep][arr] != null && plane[arr][dep] != null)
                count++;
        }
    return count;
}
```

12.45

```
public void readRecord (Flight flit, long filePosition)
{   raf.seek (filePosition);
    flit.filePosition = filePosition;
    flit.aircraftType = raf.readLine();
    flit.ticketPrice = raf.readDouble();
    flit.numSeats = raf.readInt();
    flit.numReservations = raf.readInt();
}
```

12.51 Remove the dstore answer command in 2 places and the dload answer command in 1 place.