Chapter 6

File Processing

Introduction

In Chapter 3 we discussed how to construct a `Scanner` object to read input from the console. Now we will look at how to construct `Scanner` objects to read input from files. The idea is fairly straightforward, but Java does not make it easy to read from input files. This is unfortunate because many interesting problems can be formulated as file-processing tasks. Many introductory computer science classes have abandoned file processing altogether and left the topic for the second course because it is considered too advanced for novices.

There is nothing intrinsically complex about file processing, but Java was not designed for it and the designers of Java have not been particularly eager to provide a simple solution. They did, however, introduce the `Scanner` class as a way to simplify some of the details associated with reading files. The result is that file reading is still awkward in Java, but at least the level of detail is manageable.

Before we start writing file-processing programs, we have to explore some issues related to Java exceptions. Remember that exceptions are errors that halt the execution of a program. In the case of file processing, trying to open a file that doesn't exist or trying to read beyond the end of a file generates an exception.

6.1 File-Reading Basics

In this section we'll look at the most basic issues related to file processing. What are files and why do we care about them? What are the most basic techniques for reading files in a Java program? Once you've mastered these basics, we'll move on to a more detailed discussion of the different techniques you can use to process files.

Data, Data Everywhere

People are fascinated by data. When the field of statistics emerged in the nineteenth century, there was an explosion of interest in gathering and interpreting large amounts of data. Mark Twain reported that the British statesman Benjamin Disraeli complained to him, "There are three kinds of lies: lies, damn lies, and statistics."

The advent of the Internet has only added fuel to the fire. Today, every person with an Internet connection has access to a vast array of databases containing information about every facet of our existence. Here are just a few examples:

- If you visit http://www.landmark-project.com and click on the link for "Raw Data," you will find data files about earthquakes, air pollution, baseball, labor, crime, financial markets, U.S. history, geography, weather, national parks, a "world values survey," and more.
- At http://www.gutenberg.org you'll find thousands of online books, including the complete works of Shakespeare and works by Sir Arthur Conan Doyle, Jane Austen, H.G. Wells, James Joyce, Albert Einstein, Mark Twain, Lewis Carroll, T.S. Eliot, Edgar Allan Poe, and many others.
- A wealth of genomic data is available from sites like http://www.ncbi.nlm.nih.gov/guide/. Biologists have decided that the vast quantities of data describing the human genome and the genomes of other organisms should be publicly available to everyone to study.
- Many popular web sites, such as the Internet Movie Database, make their data available for download as simple data files (see http://www.imdb.com/interfaces).
- The U.S. government produces reams of statistical data. The web site http://www.fedstats.gov provides a lengthy list of available downloads, including maps and statistics on employment, climate, manufacturing, demographics, health, crime, and more.

Files and File Objects

When you store data on your own computer, you store it in a *file*.

> **File**
>
> A collection of information that is stored on a computer and assigned a particular name.

As we have just noted, every file has a name. For example, if you were to download the text of *Hamlet* from the Gutenberg site, you might store it on your computer

Did You Know?

Origin of Data Processing

The field of data processing predates computers by over half a century. It is often said that necessity is the mother of invention, and the emergence of data processing is a good example of this principle. The crisis that spawned the industry came from a requirement in Article 1, Section 2, of the U.S. Constitution, which indicates that each state's population will determine how many representatives that state gets in the House of Representatives. To calculate the correct number, you need to know the population, so the Constitution says, "The actual Enumeration shall be made within three Years after the first Meeting of the Congress of the United States, and within every subsequent Term of ten Years, in such Manner as they shall by Law direct."

The first census was completed relatively quickly in 1790. Since then, every 10 years the U.S. government has had to perform another complete census of the population. This process became more and more difficult as the population of the country grew larger. By 1880 the government discovered that using old-fashioned hand-counting techniques, it barely completed the census within the 10 years allotted to it. So the government announced a competition for inventors to propose machines that could be used to speed up the process.

Herman Hollerith won the competition with a system involving punched cards. Clerks punched over 62 million cards that were then counted by 100 counting machines. This system allowed the 1890 tabulation to be completed in less than half the time it had taken to hand-count the 1880 results, even though the population had increased by 25 percent.

Hollerith struggled for years to turn his invention into a commercial success. His biggest problem initially was that he had just one customer: the U.S. government. Eventually he found other customers, and the company that he founded merged with competitors and grew into the company we now know as International Business Machines Corporation, or IBM.

We think of IBM as a computer company, but it sold a wide variety of data-processing equipment involving Hollerith cards long before computers became popular. Later, when it entered the computer field, IBM used Hollerith cards for storing programs and data. These cards were still being used when one of this book's authors took his freshman computer programming class in 1978.

in a file called `hamlet.txt`. A file name often ends with a special suffix that indicates the kind of data it contains or the format in which it has been stored. This suffix is known as a *file extension.* Table 6.1 lists some common file extensions.

Table 6.1 Common File Extensions

Extension	Description
`.txt`	text file
`.java`	Java source code file
`.class`	compiled Java bytecode file
`.doc`	Microsoft Word file
`.xls`	Microsoft Excel file
`.pdf`	Adobe Portable Document File
`.mp3`	audio file
`.jpg`	image file
`.zip`	compressed archive
`.html`	hypertext markup language files (most often web pages)
`.exe`	executable file

Files can be classified into text files and binary files depending on the format that is used. Text files can be edited using simple text editors. Binary files are stored using an internal format that requires special software to process. Text files are often stored using the `.txt` extension, but other file formats are also text formats, including `.java` and `.html` files.

To access a file from inside a Java program, you need to construct an internal object that will represent the file. The Java class libraries include a class called `File` that performs this duty.

You construct a `File` object by passing in the name of a file, as in the following line of code:

```
File f = new File("hamlet.txt");
```

Once you've constructed the object, you can call a number of methods to manipulate the file. For example, the following program calls a method that determines whether a file exists, whether it can be read, what its length is (i.e., how many characters are in the file), and what its absolute path is (i.e., where it is stored on the computer):

```
1 // Report some basic information about a file.
2
3 import java.io.*; // for File
4
5 public class FileInfo {
6     public static void main(String[] args) {
7         File f = new File("hamlet.txt");
```

```
8          System.out.println("exists returns " + f.exists());
9          System.out.println("canRead returns " + f.canRead());
10         System.out.println("length returns " + f.length());
11         System.out.println("getAbsolutePath returns "
12                                + f.getAbsolutePath());
13     }
14 }
```

Notice that the program includes an import from the package `java.io`, because the `File` class is part of that package. The term "io" (or I/O) is jargon used by computer science professionals to mean "input/output." Assuming you have stored the file `hamlet.txt` in the same directory as the program, you'll get output like the following when you run the program:

```
exists returns true
canRead returns true
length returns 191734
getAbsolutePath returns C:\data\hamlet.txt
```

The fact that we use a call on `new` to construct a `File` object can be misleading. We aren't constructing an actual file by constructing this object. The `File` object is an internal object that allows us to access files that already exist on the computer. Later in the chapter we will see how to write a program that creates a file as output.

Table 6.2 lists some useful methods for `File` objects.

Reading a File with a `Scanner`

The `Scanner` class that we have been using since Chapter 3 is flexible in that `Scanner` objects can be attached to many different kinds of input (see Figure 6.1).

Table 6.2 Useful Methods of `File` Objects

Method	Description
`canRead()`	Whether or not this file exists and can be read
`delete()`	Deletes the given file
`exists()`	Whether or not this file exists on the system
`getAbsolutePath()`	The full path where this file is located
`getName()`	The name of this file as a `String`, without any path attached
`isDirectory()`	Whether this file represents a directory/folder on the system
`isFile()`	Whether this file represents a file (nonfolder) on the system
`length()`	The number of characters in this file
`renameTo(file)`	Changes this file's name to the given file's name

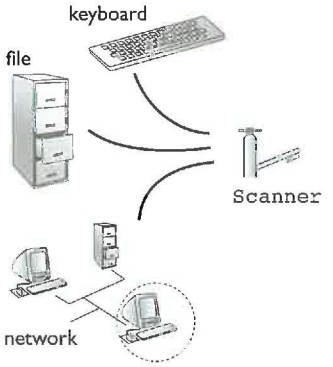

Figure 6.1 Scanners can be connected to many input sources.

You can think of a `Scanner` object as being like a faucet that you can attach to a pipe that has water flowing through it. The water can come from various sources. For example, in a house you'd attach a faucet to a pipe carrying water from the city water supply or from a well, but faucets in places like mobile homes and airplanes have different sources of water.

Thus far, we have been constructing `Scanner` objects by passing `System.in` to the `Scanner` constructor:

```
Scanner console = new Scanner(System.in);
```

This line of code instructs the computer to construct a `Scanner` that reads from the console (i.e., pauses for input from the user).

Instead of passing `System.in` to the constructor, you can pass a `File` object:

```
File f = new File("hamlet.txt");
Scanner input = new Scanner(f);
```

In this case the variable `f` is not necessary, so we can shorten this code to the following:

```
Scanner input = new Scanner(new File("hamlet.txt"));
```

This line of code, or something like it, will appear in all of your file-processing programs. When we were reading from the console window, we called our `Scanner` variable `console`. When you read from an input file, you may want to call the variable `input`. Of course, you can name the variable anything you want, as long as you refer to it in a consistent way.

Unfortunately, when you try to compile a program that constructs a `Scanner` in this manner, you'll run into a snag. Say you write a `main` method that begins by opening a `Scanner` as follows:

```
// flawed method--does not compile
public static void main(String[] args) {
    Scanner input = new Scanner(new File("hamlet.txt"));
    ...
}
```

This program does not compile. It produces a message like the following:

```
CountWords.java:8:
unreported exception java.io.FileNotFoundException;
must be caught or declared to be thrown
        Scanner input = new Scanner(new File(hamlet.txt));
                                    ^
1 error
```

The issue involves exceptions, which were described in Chapter 3. Remember that exceptions are errors that prevent a program from continuing normal execution. In this case the compiler is worried that it might not be able to find a file called `hamlet.txt`. What is it supposed to do if that happens? It won't have a file to read from, so it won't have any way to continue executing the rest of the code.

If the program is unable to locate the specified input file, it will generate an error by throwing what is known as a `FileNotFoundException`. This particular exception is known as a *checked exception.*

Checked Exception

An exception that *must* be caught or specifically declared in the header of the method that might generate it.

Because `FileNotFoundException` is a checked exception, you can't just ignore it. Java provides a construct known as the `try/catch` statement for handling such errors (described in Appendix C), but it allows you to avoid handling this error as long as you clearly indicate the fact that you aren't handling it. All you have to do is include a *`throws` clause* in the header for the `main` method to clearly state the fact that your `main` method might generate this exception.

`throws` Clause

A declaration that a method will not attempt to handle a particular type of exception.

Here's how to modify the header for the `main` method to include a `throws` clause indicating that it may throw a `FileNotFoundException`:

```
public static void main(String[] args)
        throws FileNotFoundException {
    Scanner input = new Scanner(new File("hamlet.txt"));
    ...
}
```

With the `throws` clause, the line becomes so long that we have to break it into two lines to allow it to fit in the margins of the textbook. On your own computer, you will probably include all of it on a single line.

After this modification, the program compiles. Once you've constructed the `Scanner` so that it reads from the file, you can manipulate it like any other `Scanner`. Of course, you should always prompt before reading from the console to give the user an indication of what kind of data you want, but when reading from a file, you don't need to prompt because the data is already there, stored in the file. For example, you could write a program like the following to count the number of words in *Hamlet*:

```
1 // Counts the number of words in Hamlet.
2
3 import java.io.*;
4 import java.util.*;
5
6 public class CountWords {
7     public static void main(String[] args)
8             throws FileNotFoundException {
9         Scanner input = new Scanner(new File("hamlet.txt"));
10        int count = 0;
11        while (input.hasNext()) {
12            String word = input.next();
13            count++;
14        }
15        System.out.println("total words = " + count);
16    }
17 }
```

Note that you have to include an import from `java.util` for the `Scanner` class and an import from `java.io` for the `File` class. The program generates the following output:

```
total words = 31956
```

Common Programming Error

Reading Beyond the End of a File

As you learn how to process input files, you are likely to write programs that accidentally attempt to read data when there is no data left to read. For example, the `CountWords` program we just saw uses a `while` loop to keep reading words from the file as long as there are still words left to read. After the `while` loop, you might include an extra statement to read a word:

```
while (input.hasNext()) {
    String word = input.next();
    count++;
}
String extra = input.next(); // illegal, no more input
```

This new line of code causes the computer to try to read a word when there is no word to read. Java throws an exception when this occurs:

```
Exception in thread "main" java.util.NoSuchElementException
        at java.util.Scanner.throwFor(Scanner.java:817)
        at java.util.Scanner.next(Scanner.java:1317)
        at CountWords.main(CountWords.java:15)
```

As usual, the most important information appears at the end of this list of line numbers. The exception indicates that the error occurred in line 15 of `CountWords`. The other line numbers come from the `Scanner` class and aren't helpful.

If you find yourself getting a `NoSuchElementException`, it is probably because you have somehow attempted to read beyond the end of the input. The `Scanner` is saying, "You've asked me for some data, but I don't have any such value to give you."

Common Programming Error

Forgetting `"new File"`

Suppose that you intend to construct a `Scanner` the way we've learned:

```
Scanner input = new Scanner(new File("hamlet.txt"));
```

But you accidentally forget to include the `File` object and instead write this line of code:

```
Scanner input = new Scanner("hamlet.txt"); // not right
```

Continued on next page

Continued from previous page

The line of code may seem correct because it mentions the name of the file, but it won't work because it doesn't include the `File` object.

Normally, when you make a mistake like this Java warns you that you have done something illegal. In this case, however, you'll get no warning from Java. This is because, as you'll see later in this chapter, it is possible to construct a `Scanner` from a `String`, in which case Java reads from the `String` itself.

If you were to make this mistake in the `CountWords` program, you would get the following output:

```
total words = 1
```

The program would report just one word because the `String "hamlet.txt"` looks like a single word to the `Scanner`. So, whenever you construct a `Scanner` that is supposed to read from an input file, make sure that you include the call on `new File` to construct an appropriate `File` object.

6.2 Details of Token-Based Processing


VideoNote

Now that we've introduced some of the basic issues involved in reading an input file, let's explore reading from a file in more detail. One way to process a file is token by token.

Token-Based Processing

Processing input token by token (i.e., one word at a time or one number at a time).

Recall from Chapter 3 the primary token-reading methods for the `Scanner` class:

- `nextInt` for reading an `int` value
- `nextDouble` for reading a `double` value
- `next` for reading the next token as a `String`

For example, you might want to create a file called `numbers.dat` with the following content:

```
308.2 14.9 7.4
2.8

3.9 4.7 -15.4
2.8
```

You can create such a file with an editor like Notepad on a Windows machine or TextEdit on a Macintosh. Then you might write a program that processes this input file and produces some kind of report. For example, the following program reads the first five numbers from the file and reports their sum:

```
1 // Program that reads five numbers and reports their sum.
2
3 import java.io.*;
4 import java.util.*;
5
6 public class ShowSum1 {
7     public static void main(String[] args)
8             throws FileNotFoundException {
9         Scanner input = new Scanner(new File("numbers.dat"));
10
11        double sum = 0.0;
12        for (int i = 1; i <= 5; i++) {
13            double next = input.nextDouble();
14            System.out.println("number " + i + " = " + next);
15            sum += next;
16        }
17        System.out.println("Sum = " + sum);
18    }
19 }
```

This program uses a variation of the cumulative sum code from Chapter 4. Remember that you need a `throws` clause in the header for `main` because there is a potential `FileNotFoundException`. The program produces the following output:

```
number 1 = 308.2
number 2 = 14.9
number 3 = 7.4
number 4 = 2.8
number 5 = 3.9
Sum = 337.19999999999993
```

Notice that the reported sum is not 337.2. This result is another example of a roundoff error (also described in Chapter 4).

The preceding program reads exactly five numbers from the file. More typically, you'll continue to read numbers as long as there are more numbers to read while a `while` loop is executing. Remember that the `Scanner` class includes a series of `hasNext` methods that parallel the various `next` methods. In this case, `nextDouble` is

being used to read a value of type `double`, so you can use `hasNextDouble` to test whether there is such a value to read:

```
1 // Reads an input file of numbers and prints the numbers and
2 // their sum.
3
4 import java.io.*;
5 import java.util.*;
6
7 public class ShowSum2 {
8     public static void main(String[] args)
9             throws FileNotFoundException {
10        Scanner input = new Scanner(new File("numbers.dat"));
11
12        double sum = 0.0;
13        int count = 0;
14        while (input.hasNextDouble()) {
15            double next = input.nextDouble();
16            count++;
17            System.out.println("number " + count + " = " + next);
18            sum += next;
19        }
20        System.out.println("Sum = " + sum);
21    }
22 }
```

This program would work on an input file with any number of numbers; `numbers.dat` happens to contain eight. This version of the program produces the following output:

```
number 1 = 308.2
number 2 = 14.9
number 3 = 7.4
number 4 = 2.8
number 5 = 3.9
number 6 = 4.7
number 7 = -15.4
number 8 = 2.8
Sum = 329.29999999999995
```

Structure of Files and Consuming Input

We think of text as being two-dimensional, like a sheet of paper, but from the computer's point of view, each file is just a one-dimensional sequence of characters. For example, consider the file `numbers.dat` that we used in the previous section:

```
308.2 14.9 7.4
2.8


3.9 4.7 -15.4
2.8
```

We think of this as a six-line file with text going across and down and two blank lines in the middle. However, the computer views the file differently. When you typed the text in this file, you hit the Enter key to go to a new line. This key inserts special "new line" characters in the file. You can annotate the file with `\n` characters to indicate the end of each line:

```
308.2 14.9 7.4\n
2.8\n
\n
\n
3.9 4.7 -15.4\n
2.8\n
```

Common Programming Error

Reading the Wrong Kind of Token

It's easy to write code that accidentally reads the wrong kind of data. For example, the `ShowSum1` program always reads exactly five `doubles` from the input file `numbers.dat`. But suppose that the input file has some extraneous text in it:

```
308.2 14.9 7.4
hello
2.8

3.9 4.7 –15.4
2.8
```

The first line of the file contains three numbers that the program will read properly. But when it attempts to read a fourth number, the computer finds that the next token in the file is the text "hello". This token cannot be interpreted as a `double`, so the program generates an exception:

```
number 1 = 308.2
number 2 = 14.9
number 3 = 7.4
Exception in thread "main" java.util.InputMismatchException
```

Continued on next page

Continued from previous page

```
        at java.util.Scanner.throwFor(Scanner.java:819)
        at java.util.Scanner.next(Scanner.java:1431)
        at java.util.Scanner.nextDouble(Scanner.java:2335)
        at ShowSum1.main(ShowSum1.java:13)
```

Once again, the useful line number appears at the bottom of this list. The last line indicates that the exception occurred in line 13 of the `ShowSum1` class. The other line numbers are from the `Scanner` class and aren't helpful.

You saw earlier that when you attempt to read beyond the end of a file, the `Scanner` throws a `NoSuchElementException`. In the case of this program that attempts to read the wrong kind of token, the `Scanner` throws an `InputMismatchException`. By paying attention to the kind of exception the `Scanner` throws, you can get better feedback about what the problem is.

When you mark the end of each line in your program, you no longer need to use a two-dimensional representation. You can collapse this text to a one-dimensional sequence of characters:

```
308.2 14.9 7.4\n2.8\n\n\n3.9 4.7 -15.4\n2.8\n
```

This sequence is how the computer views the file: as a one-dimensional sequence of characters including special characters that represent "new line". On some systems, including Windows machines, there are two different "new line" characters, but we'll use just `\n` here—objects like `Scanner` handle these differences for you, so you can generally ignore them. (For those who are interested, the brief explanation is that Windows machines end each line with a `\r` followed by a `\n`.)

When it is processing a file, the `Scanner` object keeps track of the current position in the file. You can think of this as an *input cursor* or pointer into the file.

Input Cursor

A pointer to the current position in an input file.

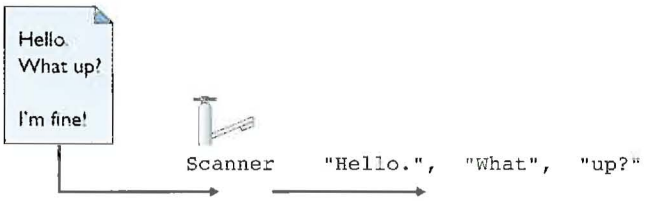

Figure 6.2 Scanners treat files as one-dimensional strings of characters and convert their contents into a series of whitespace-separated tokens.

When a `Scanner` object is first constructed, the cursor points to the beginning of the file. But as you perform various next operations, the cursor moves forward. The `ShowSum2` program from the last section processes the file through a series of calls on `nextDouble`. Let's take a moment to examine in detail how that works. Again, when the `Scanner` is first constructed, the input cursor will be positioned at the beginning of the file (indicated with an up-arrow pointing at the first character):

```
308.2 14.9 7.4\n2.8\n\n\n3.9 4.7 -15.4\n2.8\n
▲
input
cursor
```

After the first call on `nextDouble` the cursor will be positioned in the middle of the first line, after the token "308.2":

```
308.2 14.9 7.4\n2.8\n\n\n3.9 4.7 -15.4\n2.8\n
     ▲
     input
     cursor
```

We refer to this process as *consuming input*.

Consuming Input

Moving the input cursor forward past some input.

The process of consuming input doesn't actually change the file, it just changes the corresponding `Scanner` object so that it is positioned at a different point in the file.

The first call on `nextDouble` consumes the text "308.2" from the input file and leaves the input cursor positioned at the first character after this token. Notice that this leaves the input cursor positioned at a space. When the second call is made on `nextDouble`, the `Scanner` first skips past this space to get to the next token, then consumes the text "14.9" and leaves the cursor positioned at the space that follows it:

```
308.2 14.9 7.4\n2.8\n\n\n3.9 4.7 -15.4\n2.8\n
         ▲
         input
         cursor
```

The third call on `nextDouble` skips that space and consumes the text "7.4":

```
308.2 14.9 7.4\n2.8\n\n\n3.9 4.7 -15.4\n2.8\n
             ▲
             input
             cursor
```

At this point, the input cursor is positioned at the new line character at the end of the first line of input. The fourth call on `nextDouble` skips past this new line character and consumes the text "2.8":

```
308.2 14.9 7.4\n2.8\n\n\n3.9 4.7 -15.4\n2.8\n
                   ↑
                   input
                   cursor
```

Notice that when it skipped past the first new line character, the input cursor moved into data stored on the second line of input. At this point, the input cursor is positioned at the end of the second line of input, because it has consumed the "2.8" token. When a fifth call is made on `nextDouble`, the `Scanner` finds three new line characters in a row. This isn't a problem for the `Scanner`, because it simply skips past any leading white-space characters (spaces, tabs, new line characters) until it finds an actual token. So, it skips all three of these new line characters and consumes the text "3.9":

```
308.2 14.9 7.4\n2.8\n\n\n3.9 4.7 -15.4\n2.8\n
                            ↑
                            input
                            cursor
```

At this point the input cursor is positioned in the middle of the fifth line of input. (The third and fourth lines were blank). The program continues reading in this manner, consuming the remaining three numbers in the input file. After it reads the final token, "2.8", the input cursor is positioned at the new line character at the end of the file:

```
308.2 14.9 7.4\n2.8\n\n\n3.9 4.7 -15.4\n2.8\n
                                        ↑
                                        input
                                        cursor
```

If you attempted to call `nextDouble` again, it would throw a `NoSuchElementException` because there are no more tokens left to process. But remember that the `ShowSum2` program has a `while` loop that calls `hasNextDouble` before it calls `nextDouble`. When you call methods like `hasNextDouble`, the `Scanner` looks ahead in the file to see whether there is a next token and whether it is of the specified type (in this case, a `double`). So, the `ShowSum2` program will continue executing until it reaches the end of the file or until it encounters a token that cannot be interpreted as a `double`.

When the input cursor reaches the new line character at the end of the file, the `Scanner` notices that there are no more `double` values to read and returns `false` when `hasNextDouble` is called. That return stops the `while` loop from executing and causes the program to exit.

Scanner objects are designed for file processing in a forward manner. They provide a great deal of flexibility for looking ahead in an input file, but no support for reading the input backward. There are no "previous" methods, and there's no mechanism for resetting a Scanner back to the beginning of the input. Instead, you would have to construct a new Scanner object that would be positioned at the beginning of the file. We will see an example of this technique in the case study at the end of the chapter.

Scanner Parameters

Novices are sometimes surprised that the input cursor for a Scanner does not reset to the beginning of the file when it is passed as a parameter to a method. For example, consider the following variation of the ShowSum1 program. It has a method that takes a Scanner as input and an integer specifying how many numbers to process:

```
1 // Demonstrates a Scanner as a parameter to a method that
2 // can consume an arbitrary number of tokens.
3
4 import java.io.*;
5 import java.util.*;
6
7 public class ShowSum3 {
8     public static void main(String[] args)
9             throws FileNotFoundException {
10        Scanner input = new Scanner(new File("numbers.dat"));
11        processTokens(input, 2);
12        processTokens(input, 3);
13        processTokens(input, 2);
14    }
15
16    public static void processTokens(Scanner input, int n) {
17        double sum = 0.0;
18        for (int i = 1; i <= n; i++) {
19            double next = input.nextDouble();
20            System.out.println("number " + i + " = " + next);
21            sum += next;
22        }
23        System.out.println("Sum = " + sum);
24        System.out.println();
25    }
26 }
```

The main method creates a Scanner object that is tied to the numbers.dat file. It then calls the processTokens method several times, indicating the number of tokens

to process. The first call instructs the method to process two tokens. It operates on the first two tokens of the file, generating the following output:

```
number 1 = 308.2
number 2 = 14.9
Sum = 323.09999999999997
```

The second call on the method indicates that three tokens are to be processed. Some people expect the method to process the first three tokens of the file, but that's not what happens. Remember that the `Scanner` keeps track of where the input cursor is positioned. After the first call on the method, the input cursor is positioned beyond the first two tokens. So the second call on the method processes the next three tokens from the file, producing the following output:

```
number 1 = 7.4
number 2 = 2.8
number 3 = 3.9
Sum = 14.1
```

The final call on the method asks the `Scanner` to process the next two tokens from the file, so it ends up processing the sixth and seventh numbers from the file:

```
number 1 = 4.7
number 2 = -15.4
Sum = -10.7
```

The program then terminates, never having processed the eighth number in the file.

The key point to remember is that a `Scanner` keeps track of the position of the input cursor, so you can process an input file piece by piece in a very flexible manner. Even when the `Scanner` is passed as a parameter to a method, it remembers how much of the input file has been processed so far.

Paths and Directories

Files are grouped into *folders*, also called *directories*. Directories are organized in a hierarchy, starting from a root directory at the top. For example, most Windows machines have a disk drive known as `C:`. At the top level of this drive is the *root* directory, which we can describe as `C:\`. This root directory will contain various top-level directories. Each top-level directory can have subdirectories, each of which also has subdirectories, and so on. All files are stored in one of these directories. The description of how to get from the top-level directory to the particular directory that stores a file is known as the *path* of the file.

File Path

A description of a file's location on a computer, starting with a drive and including the path from the root directory to the directory where the file is stored.

We read the path information from left to right. For example, if the path to a file is `C:\school\data\hamlet.txt`, we know that the file is on the `C:` drive in a folder called `school` and in a subfolder called `data`.

In the previous section, the program used the file name `numbers.dat`. When Java encounters a simple name like that (also called a *relative* file path), it looks in the *current directory* to find the file.

> **Current Directory (a.k.a. Working Directory)**
>
> The directory that Java uses as the default when a program uses a simple file name.

The default directory varies with the Java environment you are using. In most environments, the current directory is the one in which your program appears. We'll assume that this is the case for the examples in this textbook.

You can also use a *fully qualified*, or complete, file name (sometimes called an *absolute file path*). However, this approach works well only when you know exactly where your file is going to be stored on your system. For example, if you are on a Windows machine and you have stored the file in the `C:\data` directory, you could use a file name like the following:

```
Scanner input = new Scanner(new File("C:/data/numbers.dat"));
```

Notice that the path is written with forward-slash characters rather than backslash characters. On Windows you would normally use a backslash, but Java allows you to use a forward slash instead. If you wanted to use a backslash, you would have to use a `\\` escape sequence. Most programmers choose the simpler approach of using forward-slash characters because Java does the appropriate translation on Windows machines.

You can also specify a file using a *relative path*. To write a relative path you omit the drive specification at the beginning of the string. You can still specify subdirectory relationships that will be relative to the current directory. For example, the relative path `"data/numbers.dat"` indicates a file called `numbers.dat` in a subdirectory of the working directory called `data`.

Sometimes, rather than writing a file's path name in the code yourself, you'll ask the user for a file name. For example, here is a variation of `ShowSum2` that prompts the user for the file name:

```
1 // Variation of ShowSum2 that prompts for a file name.
2
3 import java.io.*;
4 import java.util.*;
5
6 public class ShowSum4 {
7     public static void main(String[] args)
8             throws FileNotFoundException {
```

```
 9          System.out.println("This program will add a series");
10          System.out.println("of numbers from a file.");
11          System.out.println();
12
13          Scanner console = new Scanner(System.in);
14          System.out.print("What is the file name? ");
15          String name = console.nextLine();
16          Scanner input = new Scanner(new File(name));
17          System.out.println();
18
19          double sum = 0.0;
20          int count = 0;
21          while (input.hasNextDouble()) {
22              double next = input.nextDouble();
23              count++;
24              System.out.println("number " + count + " = " + next);
25              sum += next;
26          }
27          System.out.println("Sum = " + sum);
28      }
29  }
```

Notice that the program has two different `Scanner` objects: one for reading from the console and one for reading from the file. We read the file name using a call on `nextLine` to read an entire line of input from the user, which allows the user to type in file names that have spaces in them. Notice that we still need the `throws FileNotFoundException` in the header for `main` because even though we are prompting the user to enter a file name, there won't necessarily be a file of that name.

If we have this program read from the file `numbers.dat` used earlier, it will produce the following output:

```
This program will add a series
of numbers from a file.

What is the file name? numbers.dat

number 1 = 308.2
number 2 = 14.9
number 3 = 7.4
number 4 = 2.8
number 5 = 3.9
number 6 = 4.7
number 7 = -15.4
number 8 = 2.8
Sum = 329.29999999999995
```

The user also has the option of specifying a full file path:

```
This program will add a series
of numbers from a file.

What is the file name? C:\data\numbers.dat

number 1 = 308.2
number 2 = 14.9
number 3 = 7.4
number 4 = 2.8
number 5 = 3.9
number 6 = 4.7
number 7 = -15.4
number 8 = 2.8
Sum = 329.29999999999995
```

Notice that the user doesn't have to type two backslashes to get a single backslash. The `Scanner` object that reads the user's input is able to read it without escape sequences.

A More Complex Input File

Suppose an input file contains information about how many hours each employee of a company has worked. It might look like the following:

```
Erica 7.5 8.5 10.25 8 8.5
Erin 10.5 11.5 12 11 10.75
Simone 8 8 8
Ryan 6.5 8 9.25 8
Kendall 2.5 3
```

Suppose you want to find out the total number of hours worked by each individual. You can construct a `Scanner` object linked to this file to solve this task. As you start writing more complex file-processing programs, you will want to divide the program into methods to break up the code into logical subtasks. In this case, you can open the file in `main` and write a separate method to process the file.

Most file processing will involve `while` loops, because you won't know in advance how much data the file contains. You'll need to write different tests, depending on the particular file being processed, but they will almost all be calls on the various `hasNext` methods of the `Scanner` class. You basically want to say, "While you have more data for me to process, let's keep reading."

In this case, the data is a series of input lines that each begins with a name. For this program you can assume that names are simple, with no spaces in the middle.

That means you can read them with a call on the `next` method. As a result, the `while` loop test involves seeing whether there is another name in the input file:

```
while (input.hasNext()) {
    process next person.
}
```

So, how do you process each person? You have to read that person's name and then read that person's list of hours. If you look at the sample input file, you will see that the list of hours is not always the same length. For example, some employees worked on five days, while others worked only two or three days. This unevenness is a common occurrence in input files. You can deal with it by using a nested loop. The outer loop will handle one person at a time and the inner loop will handle one number at a time. The task is a fairly straightforward cumulative sum:

```
double sum = 0.0;
while (input.hasNextDouble()) {
    sum += input.nextDouble();
}
```

When you put the parts of the program together, you end up with the following complete program:

```
 1 // This program reads an input file of hours worked by various
 2 // employees and reports the total hours worked by each.
 3
 4 import java.io.*;
 5 import java.util.*;
 6
 7 public class HoursWorked {
 8     public static void main(String[] args)
 9             throws FileNotFoundException {
10         Scanner input = new Scanner(new File("hours.dat"));
11         process(input);
12     }
13
14     public static void process(Scanner input) {
15         while (input.hasNext()) {
16             String name = input.next();
17             double sum = 0.0;
18             while (input.hasNextDouble()) {
19                 sum += input.nextDouble();
20             }
21             System.out.println("Total hours worked by " + name
22                                + " = " + sum);
```

```
23          }
24      }
25  }
```

Notice that you need to put the `throws FileNotFoundException` in the header for `main`. You don't need to include it in the `process` method because the code to open the file appears in method `main`.

If you put the input data into a file called `hours.dat` and execute the program, you get the following result:

```
Total hours worked by Erica = 42.75
Total hours worked by Erin = 55.75
Total hours worked by Simone = 24.0
Total hours worked by Ryan = 31.75
Total hours worked by Kendall = 5.5
```

6.3 Line-Based Processing


VideoNote

So far we have been looking at programs that process input token by token. However, you'll often find yourself working with input files that are line-based: Each line of input represents a different case, to be handled separately from the rest. These types of files lend themselves to a second style of file processing called *line-based processing*.

> **Line-Based Processing**
>
> The practice of processing input line by line (i.e., reading in entire lines of input at a time).

Most file processing involves a combination of line- and token-based styles, and the `Scanner` class is flexible enough to allow you to write programs that include both styles of processing. For line-based processing, you'll use the `nextLine` and `hasNextLine` methods of the `Scanner` object. For example, here is a program that echoes an input file in uppercase:

```
1  // Reads a file and echoes it in uppercase.
2
3  import java.io.*;
4  import java.util.*;
5
6  public class EchoUppercase {
7      public static void main(String[] args)
8              throws FileNotFoundException {
9          Scanner input = new Scanner(new File("poem.txt"));
10         while (input.hasNextLine()) {
```

```
11              String text = input.nextLine();
12              System.out.println(text.toUpperCase());
13          }
14      }
15  }
```

This loop reads the input line by line and prints each line in uppercase until it runs out of lines to process. It reads from a file called `poem.txt`. Suppose that the file has the following contents:

```
My candle burns at both ends
It will not last the night;
But ah, my foes, and oh, my friends -
It gives a lovely light.

   --Edna St. Vincent Millay
```

If you run the preceding program on this input, it will produce the following output:

```
MY CANDLE BURNS AT BOTH ENDS
IT WILL NOT LAST THE NIGHT;
BUT AH, MY FOES, AND OH, MY FRIENDS -
IT GIVES A LOVELY LIGHT.

   --EDNA ST. VINCENT MILLAY
```

Notice that you could not have accomplished the same task with token-based processing. In this example, the line breaks are significant because they are part of the poem. Also, when you read a file token by token, you lose the spacing within the line because the `Scanner` skips any leading whitespace when it reads a token. The final line of this input file is indented because it is the name of the author and not part of the poem. That spacing would be lost if the file was read as a series of tokens.

`String` Scanners and Line/Token Combinations

In the last section we looked at a program called `HoursWorked` that processed the following data file:

```
Erica 7.5 8.5 10.25 8 8.5
Erin 10.5 11.5 12 11 10.75
Simone 8 8 8
Ryan 6.5 8 9.25 8
Kendall 2.5 3
```

The data are line-oriented, each employee's information appearing on a different line of input, but this aspect of the data wasn't incorporated into the program. The program processed the file in a token-based manner. However, that approach won't always work. For example, consider a slight variation in the data in which each line of input begins with an employee ID rather than just a name:

```
101 Erica 7.5 8.5 10.25 8 8.5
783 Erin 10.5 11.5 12 11 10.75
114 Simone 8 8 8
238 Ryan 6.5 8 9.25 8
156 Kendall 2.5 3
```

This addition seems like a fairly simple change that shouldn't require major changes to the code. Recall that the program uses a method to process the file:

```
public static void process(Scanner input) {
    while (input.hasNext()) {
        String name = input.next();
        double sum = 0.0;
        while (input.hasNextDouble()) {
            sum += input.nextDouble();
        }
        System.out.println("Total hours worked by " + name +
                           " = " + sum);
    }
}
```

Suppose you add a line of code to read the employee ID and modify the `println` to report it:

```
public static void process(Scanner input) {
    while (input.hasNext()) {
        int id = input.nextInt();
        String name = input.next();
        double sum = 0.0;
        while (input.hasNextDouble()) {
            sum += input.nextDouble();
        }
        System.out.println("Total hours worked by " + name +
                           " (id#" + id + ") = " + sum);
    }
}
```

When you run this new version of the program on the new input file, you'll get one line of output and then Java will throw an exception:

```
Total hours worked by Erica (id#101) = 825.75
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:819)
        at java.util.Scanner.next(Scanner.java:1431)
        at java.util.Scanner.nextInt(Scanner.java:2040)
        ...
```

The program correctly reads Erica's employee ID and reports it in the `println` statement, but then it crashes. Also, notice that the program reports Erica's total hours worked as 825.75, when the number should be 42.75. Where did it go wrong?

If you compute the difference between the reported sum of 825.75 hours and the correct sum of 42.75 hours, you'll find it is equal to 783. That number appears in the data file: It's the employee ID of the second employee, Erin. When the program was adding up the hours for Erica, it accidentally read Erin's employee ID as more hours worked by Erica and added this number to the sum. That's also why the exception occurs—on the second iteration of the loop, the program tries to read an employee ID for Erin when the next token in the file is her name, not an integer.

The solution is to somehow get the program to stop reading when it gets to the end of an input line. Unfortunately, there is no easy way to do this with a token-based approach. The loop asks whether the `Scanner` has a next `double` value to read, and the employee ID looks like a `double` that can be read. You might try to write a complex test that looks for a `double` that is not also an integer, but even that won't work because some of the hours are integers.

To make this program work, you need to write a more sophisticated version that pays attention to the line breaks. The program must read an entire line of input at a time and process that line by itself. Recall the `main` method for the program:

```
public static void main(String[] args)
        throws FileNotFoundException {
    Scanner input = new Scanner(new File("hours2.dat"));
    process(input);
}
```

If you incorporate a line-based loop, you'll end up with the following method:

```
public static void main(String[] args)
        throws FileNotFoundException {
    Scanner input = new Scanner(new File("hours2.dat"));
    while (input.hasNextLine()) {
```

```
        String text = input.nextLine();
        processLine(text);
    }
}
```

Reading the file line by line guarantees that you don't accidentally combine data for two employees. The downside to this approach is that you have to write a method called `processLine` that takes a `String` as a parameter, and you have to pull apart that `String`. It contains the employee ID, followed by the employee name, followed by the numbers indicating how many hours the employee worked on different days. In other words, the input line is composed of several pieces (tokens) that you want to process piece by piece. It's much easier to process this data in a token-based manner than to have it in a `String`.

Fortunately, there is a convenient way to do this. You can construct a `Scanner` object from an individual `String`. Remember that just as you can attach a faucet to different sources of water (a faucet in a house attached to city or well water versus a faucet on an airplane attached to a tank of water), you can attach a `Scanner` to different sources of input. You've seen how to attach it to the console (`System.in`) and to a file (passing a `File` object). You can also attach it to an individual `String`. For example, consider the following line of code:

```
Scanner input = new Scanner("18.4 17.9 8.3 2.9");
```

This code constructs a `Scanner` that gets its input from the `String` used to construct it. This `Scanner` has an input cursor just like a `Scanner` linked to a file. Initially the input cursor is positioned at the first character in the `String`, and it moves forward as you read tokens from the `Scanner`.

The following short program demonstrates this solution:

```
1  // Simple example of a Scanner reading from a String.
2
3  import java.util.*;
4
5  public class StringScannerExample {
6      public static void main(String[] args) {
7          Scanner input = new Scanner("18.4 17.9 8.3 2.9");
8          while (input.hasNextDouble()) {
9              double next = input.nextDouble();
10             System.out.println(next);
11         }
12     }
13 }
```

This program produces the following output:

```
18.4
17.9
8.3
2.9
```

Notice that the program produces four lines of output because there are four numbers in the `String` used to construct the `Scanner`.

When a file requires a combination of line-based and token-based processing, you can construct a `String`-based `Scanner` for each line of the input file. Using this approach, you end up with a lot of `Scanner` objects. You have a `Scanner` object that is keeping track of the input file, and you use that `Scanner` to read entire lines of input. In addition, each time you read a line of text from the file, you construct a mini-`Scanner` for just that line of input. You can then use token-based processing for these mini-`Scanner` objects, because each contains just a single line of data.

This combination of line-based and token-based processing is powerful. You will find that you can use this approach (and slight variations on it) to process a large variety of input files. To summarize, this approach involves a two-step process:

1. Break the file into lines with a `Scanner` using calls on `hasNextLine` and `nextLine`.
2. Break apart each line by constructing a `Scanner` just for that line of input and making calls on token-based methods like `hasNext` and `next`.

Following this approach, if you have a file composed of n lines of input, you end up constructing $n + 1$ different `Scanner` objects–one for each of the n individual lines (step 2) and one extra `Scanner` that is used to process the overall file line by line (step 1).

In the `HoursWorked` program, each input line contains information for a single employee. Processing the input line involves making a `Scanner` for the line and then reading its various parts (employee ID, name, hours) in a token-based manner. You can put this all together into a new version of the program:

```
1 // Variation of HoursWorked that includes employee IDs.
2
3 import java.io.*;
4 import java.util.*;
5
6 public class HoursWorked2 {
7     public static void main(String[] args)
8             throws FileNotFoundException {
9         Scanner input = new Scanner(new File("hours2.dat"));
```

```
10          while (input.hasNextLine()) {
11              String text = input.nextLine();
12              processLine(text);
13          }
14      }
15
16      // processes the given String (ID, name, and hours worked)
17      public static void processLine(String text) {
18          Scanner data = new Scanner(text);
19          int id = data.nextInt();
20          String name = data.next();
21          double sum = 0.0;
22          while (data.hasNextDouble()) {
23              sum += data.nextDouble();
24          }
25          System.out.println("Total hours worked by " + name +
26                             " (id#" + id + ") = " + sum);
27      }
28  }
```

Notice that the `main` method includes line-based processing to read entire lines of input from the file. Each such line is passed to the `processLine` method. Each time the program calls `processLine`, it makes a mini-`Scanner` for just that line of input and uses token-based processing (calling the methods `nextInt`, `next`, and `nextDouble`).

This new version of the program produces the following output:

```
Total hours worked by Erica (id#101) = 42.75
Total hours worked by Erin (id#783) = 55.75
Total hours worked by Simone (id#114) = 24.0
Total hours worked by Ryan (id#238) = 31.75
Total hours worked by Kendall (id#156) = 5.5
```

While this version of the program is a little more complex than the original, it is much more flexible because it pays attention to line breaks.

6.4 Advanced File Processing

In this section we'll explore two advanced topics related to file processing: producing output files and guaranteeing that files can be read.

Output Files with `PrintStream`

All of the programs we've studied so far have sent their output to the console window by calling `System.out.print` or `System.out.println`. But just as you can read input from a file instead of reading from the console, you can write output to a file

instead of writing it to the console. There are many ways to accomplish this task. The simplest approach is to take advantage of what you already know. You've already learned all about how `print` and `println` statements work, and you can leverage that knowledge to easily create output files.

If you look at the Java documentation, you will find that `System.out` is a variable that stores a reference to an object of type `PrintStream`. The `print` and `println` statements you've been writing are calls on methods that are part of the `PrintStream` class. The variable `System.out` stores a reference to a special `PrintStream` object that is tied to the console window. However, you can construct other `PrintStream` objects that send their output to other places. Suppose, for example, that you want to send output to a file called `results.txt`. You can construct a `PrintStream` object as follows:

```
PrintStream output = new PrintStream(new File("results.txt"));
```

This line of code looks a lot like the one we used to construct a `Scanner` tied to an input file. In this case, the computer is creating an output file. If no such file already exists, the program creates it. If such a file does exist, the computer overwrites the current version. Initially, the file will be empty. It will end up containing whatever output you tell it to produce through calls on `print` and `println`.

The line of code that constructs a `PrintStream` object can generate an exception if Java is unable to create the file you've described. There are many reasons that this might happen: You might not have permission to write to the directory, or the file might be locked because another program is using it. Like the line of code that creates a file-based `Scanner`, this line of code potentially throws a `FileNotFoundException`. Therefore, Java requires you to include the `throws` clause in whatever method contains this line of code. The simplest approach is to put this line in `main`. In fact, it is common practice to have the `main` method begin with the lines of code that deal with the input and output files.

Once you have constructed a `PrintStream` object, how do you use it? You should already have a good idea of what to do. We have been making calls on `System.out.print` and `System.out.println` since Chapter 1. If you recall everything you know about `System.out` you'll have a good idea of what to do, but for this program, you will call `output.print` instead of `System.out.print` and `output.println` instead of `System.out.println`.

As a simple example, remember that in Chapter 1 we looked at the following variation of the simple "hello world" program that produces several lines of output:

```
1 public class Hello2 {
2    public static void main(String[] args) {
3       System.out.println("Hello, world!");
4       System.out.println();
```

```
5         System.out.println("This program produces four");
6         System.out.println("lines of output.");
7     }
8 }
```

Here is a variation that sends its output to a file called `hello.txt`:

```
1  // Variation of Hello2 that prints to a file.
2
3  import java.io.*;
4
5  public class Hello4 {
6     public static void main(String[] args)
7           throws FileNotFoundException {
8        PrintStream output =
9           new PrintStream(new File("hello.txt"));
10       output.println("Hello, world.");
11       output.println();
12       output.println("This program produces four");
13       output.println("lines of output.");
14    }
15 }
```

When you run this new version of the program, a curious thing happens. The program doesn't seem to do anything; no output appears on the console at all. You're so used to writing programs that send their output to the console that this might seem confusing at first. We don't see any output in the console window when we run this program because the output was directed to a file instead. After the program finishes executing, you can open up the file called `hello.txt` and you'll find that it contains the following:

```
Hello, world.

This program produces four
lines of output.
```

The main point is that everything you've learned to do with `System.out`, you can also do with `PrintStream` objects that are tied to files.

You can also write methods that take `PrintStream` objects as parameters. For example, consider the task of fixing the spacing for a series of words. Say that you have a line of text with erratic spacing, like the following line:

```
a       new  nation,     conceived  in     liberty
```

Suppose that you want to print this text with exactly one space between each pair of words:

```
a new nation, conceived in liberty
```

How do you do that? Assume that you are writing a method that is passed a `String` to echo and a `PrintStream` object to which the output should be sent:

```
public static void echoFixed(String text, PrintStream output) {
    ...
}
```

You can construct a `Scanner` from the `String` and then use the `next` method to read one word at a time. Recall that the `Scanner` class ignores whitespace, so you'll get just the individual words without all of the spaces between them. As you read words, you'll need to echo them to the `PrintStream` object. Here's a first attempt:

```
Scanner data = new Scanner(text);
while (data.hasNext()) {
    output.print(data.next());
}
```

This code does a great job of deleting the long sequences of spaces from the `String`, but it goes too far: It eliminates all of the spaces. To get one space between each pair of words, you'll have to include some spaces:

```
Scanner data = new Scanner(text);
while (data.hasNext()) {
    output.print(data.next() + " ");
}
```

This method produces results that look pretty good, but it prints an extra space at the end of the line. To get rid of that space so that you truly have spaces appearing only between pairs of words, you'll have to change the method slightly. This is a classic fencepost problem; you want to print one more word than you have spaces. You can use the typical solution of processing the first word before the loop begins and swapping the order of the other two operations inside the loop (printing a space and then the word):

```
Scanner data = new Scanner(text);
output.print(data.next());
while (data.hasNext()) {
    output.print(" " + data.next());
}
```

This version of the program works well for almost all cases, but by including the fencepost solution, which echoes the first word before the loop begins, you've introduced an assumption that there is a first word. If the `String` has no words at all, this call on `next` will throw an exception. So, you need a test for the case in which the `String` doesn't contain any words. If you also want this program to produce a complete line of output, you'll have to include a call on `println` to complete the line of output after printing the individual words. Incorporating these changes, you get the following code:

```
public static void echoFixed(String text, PrintStream output) {
    Scanner data = new Scanner(text);
    if (data.hasNext()) {
        output.print(data.next());
        while (data.hasNext()) {
            output.print(" " + data.next());
        }
    }
    output.println();
}
```

Notice that you're now calling `output.print` and `output.println` instead of calling `System.out.print` and `System.out.println`. An interesting aspect of this method is that it can be used not only to send output to an output file, but also to send it to `System.out`. The method header indicates that it works on any `PrintStream`, so you can call it to send output either to a `PrintStream` object tied to a file or to `System.out`.

The following complete program uses this method to fix the spacing in an entire input file of text. To underscore the flexibility of the method, the program sends its output both to a file (`words2.txt`) and to the console:

```
1  // This program removes excess spaces in an input file.
2
3  import java.io.*;
4  import java.util.*;
5
6  public class FixSpacing {
7      public static void main(String[] args)
8              throws FileNotFoundException {
9          Scanner input = new Scanner(new File("words.txt"));
10         PrintStream output =
11             new PrintStream(new File("words2.txt"));
12         while (input.hasNextLine()) {
13             String text = input.nextLine();
14             echoFixed(text, output);
```

```
            echoFixed(text, System.out);
        }
    }

    public static void echoFixed(String text,
                                 PrintStream output) {
        Scanner data = new Scanner(text);
        if (data.hasNext()) {
            output.print(data.next());
            while (data.hasNext()) {
                output.print(" " + data.next());
            }
        }
        output.println();
    }
}
```

Consider the following input file:

```
    four          score     and
seven     years ago     our
   fathers brought  forth on this      continent
a        new nation,      conceived  in     liberty
  and  dedicated    to  the  proposition     that
   all    men     are      created     equal
```

Using this input, the program produces the following output file, called `words2.txt`:

```
four score and
seven years ago our
fathers brought forth on this continent
a new nation, conceived in liberty
and dedicated to the proposition that
all men are created equal
```

The output also appears in the console window.

Guaranteeing That Files Can Be Read

The programs we have studied so far assume that the user will provide a legal file name. But what if the user accidentally types in the name of a file that doesn't exist or that can't be read for some reason? In this section we explore how to guarantee that a file can be read.

Let's explore how you might handle the task of prompting the user for a file name in the console window. If the user does not input a legal file name, you can keep

prompting until the user does enter a legal name. The `File` class has a method called `canRead` that you can use to test whether a file exists and can be read. You can print an error message each time the file can't be read, until you get a good file name. This situation turns out to be a fencepost problem. If you end up prompting the user n times, you'll want to produce $n - 1$ error messages. You can use the classic fencepost solution of prompting for the file name once before the loop:

```
System.out.print("input file name? ");
File f = new File(console.nextLine());
while (!f.canRead()) {
    System.out.println("File not found. Try again.");
    System.out.print("input file name? ");
    f = new File(console.nextLine());
}
```

This code could be included in your `main` method, but there is enough code here that it makes sense to put it in its own method. Because it is prompting the user for input, the code requires a `Scanner` for reading from the console. It can return a `Scanner` that is tied to the input file to process.

When you try to create a method that includes this code, you again run into the problem of checked exceptions. Even though we are being very careful to make sure that the file exists and can be read, the Java compiler doesn't know that. From its point of view, this code might throw an exception. You still need to include a `throws` clause in the method header, just as you've been doing with `main`:

```
public static Scanner getInput(Scanner console)
        throws FileNotFoundException {
    System.out.print("input file name? ");
    File f = new File(console.nextLine());
    while (!f.canRead()) {
        System.out.println("File not found. Try again.");
        System.out.print("input file name? ");
        f = new File(console.nextLine());
    }
    // now we know that f is a file that can be read
    return new Scanner(f);
}
```

Here is a variation of the `CountWords` program that prompts for a file name:

```
1  // Variation of CountWords that prompts for a file name.
2
3  import java.io.*;
4  import java.util.*;
```

```
5
6  public class CountWords2 {
7      public static void main(String[] args)
8              throws FileNotFoundException {
9          Scanner console = new Scanner(System.in);
10         Scanner input = getInput(console);
11
12         // and count words
13         int count = 0;
14         while (input.hasNext()) {
15             String word = input.next();
16             count++;
17         }
18         System.out.println("total words = " + count);
19     }
20
21     // Prompts the user for a legal file name; creates and
22     // returns a Scanner tied to the file
23     public static Scanner getInput(Scanner console)
24             throws FileNotFoundException {
25         System.out.print("input file name? ");
26         File f = new File(console.nextLine());
27         while (!f.canRead()) {
28             System.out.println("File not found. Try again.");
29             System.out.print("input file name? ");
30             f = new File(console.nextLine());
31         }
32         // now we know that f is a file that can be read
33         return new Scanner(f);
34     }
35 }
```

The following log of execution shows what happens when the user types in some illegal file names, ending with a legal file name:

```
input file name? amlet.txt
File not found. Try again.
input file name? hamlet.dat
File not found. Try again.
input file name? humlet.txt
File not found. Try again.
input file name? hamlet.txt
Total words = 31956
```

The code for opening a file is fairly standard and could be used without modification in many programs. We refer to this as *boilerplate code*.

Boilerplate Code

Code that tends to be the same from one program to another.

The `getInput` method is a good example of the kind of boilerplate code that you might use in many different file-processing programs.

6.5 Case Study: Zip Code Lookup


VideoNote

Knowing the distance between two locations turns out to be extremely helpful and valuable. For example, many popular Internet dating sites allow you to search for people on the basis of a target location. On Match.com, you can search for potential matches within a particular radius of a given city or zip code (5 miles, 10 miles, 15 miles, 25 miles, and so on). Obviously this is an important feature for a dating site because people are most interested in dating other people who live near them.

There are many other applications of this kind of proximity search. In the 1970s and 1980s there was an explosion of interest in what is known as direct mail marketing that has produced what we now call junk mail. Proximity searches are very important in direct mail campaigns. A local store, for example, might decide to mail out a brochure to all residents who live within 5 miles of the store. A political candidate might pay a membership organization like The Sierra Club or the National Rifle Association a fee to get the mailing addresses of all its members who live within a certain distance of a town or a city district.

Massive databases keep track of potential customers and voters. Direct-mail marketing organizations often want to find the distance between one of these individuals and some fixed location. The distance calculations are done almost exclusively with zip codes. There are over 40,000 five-digit zip codes in the United States. Some zip codes cover rural areas that are fairly large, but more often a zip code determines your location in a city or town to within a fraction of a mile. If you use the more specific Zip + 4 database, you can often pinpoint a location to within a few city blocks.

If you do a web search for "zip code database" or "zip code software" you will find that there are many people selling the data and the software to interpret the data. There are also some free databases, although the data aren't quite as accurate. The U.S. Census Bureau is the source of much of the free data.

To explore this application, let's write a program that finds all the zip codes within a certain proximity of another zip code. A web site like Match.com could use the logic of this program to find potential dates within a certain radius. You'd simply start with the zip code of interest, find all the other zip codes within a particular distance, and then find all the customers who have those zip codes. We don't have access to a massive dating database like Match.com, so we'll be working on just the first part of this task, finding the zip codes that are within a specified distance.

As we noted earlier, some free zip code databases are available online. Our sample program uses data compiled by software developer Schuyler Erle, whose data are distributed free through a Creative Commons license (obtained from http://www.boutell.com/zipcodes/).

We have reformatted the data to make it more convenient for us to work with it (a process known as *data munging*). We will be working with a file called `zipcode.txt` that has a series of 3-line entries, one for each zip code. The first line contains the zip code, the second line contains the city and state, and the third line contains two numbers that represent the latitude and longitude of the zip code. For example, the following is an entry for one of the authors' home zip codes:

```
98104
Seattle, WA
47.60252 -122.32855
```

The overall task is to prompt the user for a target zip code and a proximity and to show all zip codes within the given proximity of the target. Here is a first attempt at pseudocode for the overall task:

```
introduce program to user.
prompt for target zip code and proximity.
display matching zip codes from file.
```

This approach doesn't quite work. To display a match, you have to compare the target location to each of the different zip codes in the data file. You'll need the latitude and longitude information to make this comparison. But when you prompt the user, you're just asking for a zip code and proximity. You could alter the program to prompt for a latitude and longitude, but that wouldn't be a very friendly program for the user. Imagine if Match.com required you to know your latitude and longitude in order for you to search for people who live near you.

Instead, you can use the zip code data to find the latitude and longitude of the target zip code. As a result, you'll have to search the data twice. The first time through you will be looking for the target zip code, so that you can find its coordinates. The second time through you will display all the zip codes that are within the distance specified by the user. Here is a new version of the pseudocode:

```
introduce program to user.
prompt for target zip code and proximity.
find coordinates for target zip code.
display matching zip codes from file.
```

Introducing the program and prompting for the target zip code and proximity are fairly straightforward tasks that don't require detailed explanation. The real work of the program involves solving the third and fourth steps in this pseudocode. Each of these steps is sufficiently complex that it deserves to be included in a static method.

First consider the problem of finding the coordinates for the target zip code. You need to set up a `Scanner` to read from the file, and then you need to call the method that will do the search. But what information should the searching method return? You want the coordinates of the target zip code (the latitude and longitude). Your method can't return two different values, but these coordinates appear on a single line of input, so you can return that line of input as a `String`. That means that your main method will include the following code:

```
Scanner input = new Scanner(new File("zipcode.txt"));
String targetCoordinates = find(target, input);
```

The method should read the input file line by line, searching for the target zip code. Remember that each entry in the file is composed of three different lines. As a result, you need a slight variation of the standard line-processing loop that reads three lines each time through the loop:

```
public static String find(String target, Scanner input) {
    while (input.hasNextLine()) {
        String zip = input.nextLine();
        String city = input.nextLine();
        String coordinates = input.nextLine();
        ...
    }
    ...
}
```

As you read various zip code entries, you want to test each to see whether it matches the target. Remember that you need to use the `equals` method to compare strings for equality. If you find a match, you can print it and return the coordinates:

```
public static String find(String target, Scanner input) {
    while (input.hasNextLine()) {
        String zip = input.nextLine();
        String city = input.nextLine();
        String coordinates = input.nextLine();
        if (zip.equals(target)) {
            System.out.println(zip + ": " + city);
            return coordinates;
        }
    }
    ...
}
```

This method isn't complete because you have to consider the case in which the target zip code doesn't appear in the file. In that case, you exit the loop without having returned a value. There are many things the program could do at this point, such as printing an error message or throwing an exception. To keep things simple, let's instead return a set of fake coordinates. If the program returns a latitude and longitude of (0, 0), there won't be any matches unless the user asks for an outrageously high proximity (over 4,000 miles):

```
public static String find(String target, Scanner input) {
    while (input.hasNextLine()) {
        String zip = input.nextLine();
        String city = input.nextLine();
        String coordinates = input.nextLine();
        if (zip.equals(target)) {
            System.out.println(zip + ": " + city);
            return coordinates;
        }
    }
    // at this point we know the zip code isn't in the file
    // we return fictitious (no match) coordinates
    return "0 0";
}
```

This method completes the first of the two file-processing tasks. In the second task, you have to read the file and search for zip codes within the given proximity. The `Scanner` doesn't have a reset option for going back to the beginning of the file. Instead, you have to construct a second `Scanner` object that will be used for the second pass. Thus, your code in `main` will look like the following:

```
input = new Scanner(new File("zipcode.txt"));
showMatches(targetCoordinates, input, miles);
```

The code for finding matches involves a similar file-processing loop that reads three lines of input at a time, printing matches as it finds them:

```
public static void showMatches(String targetCoordinates,
                               Scanner input, double miles) {
    // compute lat1 and long1
    System.out.println("zip codes within " + miles + " miles:");
    while (input.hasNextLine()) {
        String zip = input.nextLine();
        String city = input.nextLine();
        String coordinates = input.nextLine();
        // compute lat2 and long2
        double distance = distance(lat1, long1, lat2, long2);
```

```
        if (distance <= miles) {
            // print zip code
        }
    }
}
```

Again, this is an incomplete version of the method. It indicates that before the loop begins you will compute two values known as `lat1` and `long1` that represent the latitude and longitude of the target coordinates. Inside the loop you compute values for `lat2` and `long2` that represent the latitude and longitude of the next entry from the data file. The latitude and longitude are stored in a `String`. You can construct a `Scanner` for each `String` that can be used to pull out the individual tokens. You also need to fill in the details of printing. This is a good place to use a `printf` to format the output:

```
public static void showMatches(String targetCoordinates,
                               Scanner input, double miles) {
    Scanner data = new Scanner(targetCoordinates);
    double lat1 = data.nextDouble();
    double long1 = data.nextDouble();
    System.out.println("zip codes within " + miles + " miles:");
    while (input.hasNextLine()) {
        String zip = input.nextLine();
        String city = input.nextLine();
        String coordinates = input.nextLine();
        data = new Scanner(coordinates);
        double lat2 = data.nextDouble();
        double long2 = data.nextDouble();
        double distance = distance(lat1, long1, lat2, long2);
        if (distance <= miles) {
            System.out.printf("    %s %s, %3.2f miles\n",
                              zip, city, distance);
        }
    }
}
```

This addition almost completes the program. The preceding code calls a method called `distance` that is intended to compute the distance between two points, given their latitude and longitude. This problem was included as Programming Project 5 in Chapter 3. You can use the following standard formula:

Let φ_1, λ_1, and φ_2, λ_2 be the latitude and longitude of two points, respectively. $\Delta\lambda$, the longitudinal difference, and $\Delta\sigma$, the angular difference/distance in radians, can be determined from the spherical law of cosines as:

$$\Delta\sigma = \arccos(\sin\varphi_1 \sin\varphi_2 + \cos\varphi_1 \cos\varphi_2 \cos\Delta\lambda)$$

We won't dwell on the math involved here, but a short explanation might be helpful. Imagine forming a triangle by connecting two points with the North Pole. From the two latitudes, you can compute the distance from each point to the North Pole. The difference between the two longitudes tells you the angle formed by these two sides of the triangle. You may recall from geometry class that if you know two sides and the angle between them, then you can compute the third side. We are using a special version of the law of cosines that works for spheres to compute the length of the third side of the triangle (which is the line connecting the two points on our sphere). We have to convert from degrees into radians and we have to include the radius of our sphere (in this case the Earth). The resulting calculation is included in the final version of the program.

Here is the complete version of the program:

```
1  // This program uses a file of zip code information to allow a user
2  // to find zip codes within a certain distance of another zip code.
3
4  import java.util.*;
5  import java.io.*;
6
7  public class ZipLookup {
8     // radius of sphere. Here it's the Earth, in miles
9     public static final double RADIUS = 3956.6;
10
11    public static void main(String[] args)
12          throws FileNotFoundException {
13       giveIntro();
14       Scanner console = new Scanner(System.in);
15
16       System.out.print("What zip code are you interested in? ");
17       String target = console.next();
18       System.out.print("And what proximity (in miles)? ");
19       double miles = console.nextDouble();
20       System.out.println();
21
22       Scanner input = new Scanner(new File("zipcode.txt"));
23       String targetCoordinates = find(target, input);
24       input = new Scanner(new File("zipcode.txt"));
25       showMatches(targetCoordinates, input, miles);
26    }
27
28    // introduces the program to the user
29    public static void giveIntro() {
30       System.out.println("Welcome to the zip code database.");
31       System.out.println("Give me a 5-digit zip code and a");
```

```
32          System.out.println("proximity, and I'll tell you where");
33          System.out.println("that zip code is located, along");
34          System.out.println("with a list of other zip codes");
35          System.out.println("within the given proximity.");
36          System.out.println();
37      }
38
39      // Searches for the given string in the input file; if found,
40      // returns the coordinates; otherwise returns (0, 0)
41      public static String find(String target, Scanner input) {
42          while (input.hasNextLine()) {
43              String zip = input.nextLine();
44              String city = input.nextLine();
45              String coordinates = input.nextLine();
46              if (zip.equals(target)) {
47                  System.out.println(zip + ": " + city);
48                  return coordinates;
49              }
50          }
51          // at this point we know the zip code isn't in the file
52          // we return fictitious (no match) coordinates
53          return "0 0";
54      }
55
56      // Shows all matches for the given coordinates within the
57      // given number of miles
58      public static void showMatches(String targetCoordinates,
59                                     Scanner input, double miles) {
60          Scanner data = new Scanner(targetCoordinates);
61          double lat1 = data.nextDouble();
62          double long1 = data.nextDouble();
63          System.out.println("zip codes within " + miles + " miles:");
64          while (input.hasNextLine()) {
65              String zip = input.nextLine();
66              String city = input.nextLine();
67              String coordinates = input.nextLine();
68              data = new Scanner(coordinates);
69              double lat2 = data.nextDouble();
70              double long2 = data.nextDouble();
71              double distance = distance(lat1, long1, lat2, long2);
72              if (distance <= miles) {
73                  System.out.printf("    %s %s, %3.2f miles\n",
74                                    zip, city, distance);
75              }
```

```
76          }
77      }
78
79      // Returns spherical distance in miles given the latitude
80      // and longitude of two points (depends on constant RADIUS)
81      public static double distance(double lat1, double long1,
82                                    double lat2, double long2) {
83          lat1 = Math.toRadians(lat1);
84          long1 = Math.toRadians(long1);
85          lat2 = Math.toRadians(lat2);
86          long2 = Math.toRadians(long2);
87          double theCos = Math.sin(lat1) * Math.sin(lat2) +
88              Math.cos(lat1) * Math.cos(lat2) * Math.cos(long1 - long2);
89          double arcLength = Math.acos(theCos);
90          return arcLength * RADIUS;
91      }
92  }
```

Here is a sample execution:

```
Welcome to the zip code database.
Give me a 5-digit zip code and a
proximity, and I'll tell you where
that zip code is located, along
with a list of other zip codes
within the given proximity.

What zip code are you interested in? 98104
And what proximity (in miles)? 1

98104: Seattle, WA
zip codes within 1.0 miles:
    98101 Seattle, WA, 0.62 miles
    98104 Seattle, WA, 0.00 miles
    98154 Seattle, WA, 0.35 miles
    98164 Seattle, WA, 0.29 miles
    98174 Seattle, WA, 0.35 miles
```

There is an old saying that you get what you pay for, and these zip code data are no exception. There are several web sites that list zip codes within a mile of 98104, and they include many zip codes not included here. That's because the free zip code information is incomplete. Each of those web sites gives you the option of obtaining a better database for a small fee.

Chapter Summary

Files are represented in Java as `File` objects. The `File` class is found in the `java.io` package.

A `Scanner` object can read input from a file rather than from the keyboard. This task is achieved by passing `new File(`*filename*`)` to the `Scanner`'s constructor, rather than passing `System.in`.

A checked exception is a program error condition that must be caught or declared in order for the program to compile. For example, when constructing a `Scanner` that reads a file, you must write the phrase `throws FileNotFoundException` in the `main` method's header.

The `Scanner` treats an input file as a one-dimensional stream of data that is read in order from start to end. The input cursor consumes (moves past) input tokens as they are read and returns them to your program.

A `Scanner` that reads a file makes use of the various `hasNext` methods to discover when the file's input has been exhausted.

`Scanner`s can be passed as parameters to methods to read part or all of a file, since they are objects and therefore use reference semantics.

A file name can be specified as a relative path such as `data/text/numbers.dat`, which refers to a file called `numbers.dat` that exists in the `data/text/` subfolder of the current directory. Alternatively, you can specify a full file path such as `C:/Documents and Settings/user/My Documents/data/text/numbers.dat`.

In many files, input is structured by lines, and it makes sense to process those files line by line. In such cases, it is common to use nested loops: an outer loop that iterates over each line of the file and an inner loop that processes the tokens in each line.

Output to a file can be achieved with a `PrintStream` object, which is constructed with a `File` and has the same methods as `System.out`, such as `println` and `print`.

Self-Check Problems

Section 6.1: File-Reading Basics

1. What is a file? How can we read data from a file in Java?
2. What is wrong with the following line of code?

```
Scanner input = new Scanner("test.dat");
```

3. Which of the following is the correct syntax to declare a `Scanner` to read the file `example.txt` in the current directory?

   a. `Scanner input = new Scanner("C:\example.txt");`
   b. `Scanner input = new Scanner(new File("example.txt"));`
   c. `Scanner input = new File("\\example.txt");`
   d. `File input = new Scanner("/example.txt");`
   e. `Scanner input = new Scanner("C:/example.txt");`

4. Write code to construct a `Scanner` object to read the file `input.txt`, which exists in the same folder as your program.

Section 6.2: Details of Token-Based Processing

5. Given the following line of input, what tokens does a `Scanner` break the line apart into?

```
welcome...to the matrix.
```

a. `"welcome", "to", "the", "matrix"`
b. `"welcome...to the matrix."`
c. `"welcome...to", "the", "matrix."`
d. `"welcome...", "to", "the matrix."`
e. `"welcome", "to the matrix"`

6. Given the following line of input, what tokens does a `Scanner` break the line apart into?

```
in fourteen-hundred 92
columbus sailed the ocean blue :)
```

a. `"in", "fourteen-hundred", "92"`
b. `"in", "fourteen-hundred", "92", "columbus", "sailed", "the", "ocean", "blue", ":)"`
c. `"in", "fourteen", "hundred", "92", "columbus", "sailed", "the", "ocean", "blue"`
d. `"in", "fourteen-hundred", "92\ncolumbus", "sailed", "the", "ocean", "blue :)"`
e. `"in fourteen-hundred 92", "columbus sailed the ocean blue :)"`

7. How many tokens are there in the following input, and what `Scanner` method(s) can be used to read each of the tokens?

```
Hello there,how are you?
I am "very well", thank you.
12 34 5.67 (8 + 9) "10"
```

8. What is wrong with the following line of code?

```
Scanner input = new Scanner(new File("C:\temp\new files\test.dat"));
```

(Hint: Try printing the `String` in this line of code.)

9. Answer the following questions about a Java program located on a Windows machine in the folder `C:\Documents and Settings\amanda\My Documents\programs`:

a. What are two legal ways you can refer to the file `C:\Documents and Settings\amanda\My Documents\programs\numbers.dat`?
b. How can you refer to the file `C:\Documents and Settings\amanda\My Documents\programs\data\homework6\input.dat`?
c. How many, and in what legal, ways can you refer to the file `C:\Documents and Settings\amanda\My Documents\homework\data.txt`?

10. Answer the following questions about a Java program located on a Linux machine in the folder `/home/amanda/Documents/hw6`:

a. What are two legal ways you can refer to the file `/home/amanda/Documents/hw6/names.txt`?
b. How can you refer to the file `/home/amanda/Documents/hw6/data/numbers.txt`?
c. How many legal ways can you refer to the file `/home/amanda/download/saved.html`?

11. The following program contains 6 mistakes! What are they?

```
1  public class Oops6 {
2      public static void main(String[] args) {
3          Scanner in = new Scanner("example.txt");
4          countWords(in);
5      }
6
7      // Counts total lines and words in the input scanner.
8      public static void countWords(Scanner input) {
9          Scanner input = new Scanner("example.txt");
10         int lineCount = 0;
11         int wordCount = 0;
12
13         while (input.nextLine()) {
14             String line = input.line();     // read one line
15             lineCount++;
16             while (line.next()) {           // tokens in line
17                 String word = line.hasNext;
18                 wordCount++;
19             }
20         }
21     }
22 }
```

Section 6.3: Line-Based Processing

12. For the next several questions, consider a file called `readme.txt` that has the following contents:

```
6.7        This file has
        several input lines.

  10 20         30   40

test
```

What would be the output from the following code when it is run on the `readme.txt` file?

```
Scanner input = new Scanner(new File("readme.txt"));
int count = 0;
while (input.hasNextLine()) {
    System.out.println("input: " + input.nextLine());
    count++;
}
System.out.println(count + " total");
```

13. What would be the output from the code in the previous exercise if the calls to `hasNextLine` and `nextLine` were replaced by calls to `hasNext` and `next`, respectively?

14. What would be the output from the code in the previous exercise if the calls to `hasNextLine` and `nextLine` were replaced by calls to `hasNextInt` and `nextInt`, respectively? How about `hasNextDouble` and `nextDouble`?

15. Write a program that prints itself to the console as output. For example, if the program is stored in `Example.java`, it will open the file `Example.java` and print its contents to the console.

16. Write code that prompts the user for a file name and prints the contents of that file to the console as output. Assume that the file exists. You may wish to place this code into a method called `printEntireFile`.

17. Write a program that takes as input lines of text like the following:

```
This is some
text here.
```

The program should produce as output the same text inside a box, as in the following:

```
+--------------+
| This is some |
| text here.   |
+--------------+
```

Your program will have to assume some maximum line length (e.g., 12 in this case).

Section 6.4: Advanced File Processing

18. What object is used to write output to a file? What methods does this object have available for you to use?

19. Write code to print the following four lines of text into a file named `message.txt`:

```
Testing,
1, 2, 3.

This is my output file.
```

20. Write code that repeatedly prompts the user for a file name until the user types the name of a file that exists on the system. You may wish to place this code into a method called `getFileName`, which will return that file name as a `String`.

21. In Problem 16, you wrote a piece of code that prompted the user for a file name and printed that file's contents to the console. Modify your code so that it will repeatedly prompt the user for the file name until the user types the name of a file that exists on the system.

Exercises

1. Write a method called `boyGirl` that accepts a `Scanner` that is reading its input from a file containing a series of names followed by integers. The names alternate between boys' names and girls' names. Your method should compute the absolute difference between the sum of the boys' integers and the sum of the girls' integers. The input could end with either a boy or girl; you may not assume that it contains an even number of names. For example, if the input file contains the following text:

```
Erik 3 Rita 7 Tanner 14 Jillyn 13 Curtis 4 Stefanie 12 Ben 6
```

Then the method should produce the following console output, since the boys' sum is 27 and the girls' sum is 32:

```
4 boys, 3 girls
Difference between boys' and girls' sums: 5
```

2. Write a method called `evenNumbers` that accepts a `Scanner` reading input from a file with a series of integers, and report various statistics about the integers to the console. Report the total number of numbers, the sum of the numbers, the count of even numbers and the percent of even numbers. For example, if the input file contains the following text:

```
5 7 2 8 9 10 12 98 7 14 20 22
```

Then the method should produce the following console output:

```
12 numbers, sum = 214
8 evens (66.67%)
```

3. Write a method called `negativeSum` that accepts a `Scanner` reading input from a file containing a series of integers, and print a message to the console indicating whether the sum starting from the first number is ever negative. You should also return `true` if a negative sum can be reached and `false` if not. For example, suppose the file contains the following text:

```
38 4 19 -27 -15 -3 4 19 38
```

Your method would consider the sum of just one number (38), the first two numbers (38 + 4), the first three numbers (38 + 4 + 19), and so on to the end. None of these sums is negative, so the method would produce the following output and return `false`:

```
no negative sum
```

If the file instead contains the following numbers:

```
14 7 -10 9 -18 -10 17 42 98
```

The method finds that a negative sum of −8 is reached after adding the first six numbers. It should output the following to the console and return `true`:

```
sum of -8 after 6 steps
```

4. Write a method called `cointCoins` that accepts a `Scanner` representing an input file whose data is a series of pairs of tokens, where each pair begins with an integer and is followed by the type of coin, which will be "pennies" (1 cent each), "nickels" (5 cents each), "dimes" (10 cents each), or "quarters" (25 cents each), case-insensitively. Add up the cash values of all the coins and print the total money. For example, if the input file contains the following text:

```
3 pennies 2 quarters 1 Pennies 23 NiCkeLs 4 DIMES
```

For the input above, your method should produce the following output:

```
Total money: $2.09
```

5. Write a method called `collapseSpaces` that accepts a `Scanner` representing an input file as its parameter, then reads that file and outputs it with all its tokens separated by single spaces, collapsing any sequences of multiple spaces into single spaces. For example, consider the following text:

```
many   spaces   on   this   line!
```

If this text were a line in the file, the same line should be output as follows:

```
many spaces on this line!
```

6. Write a method called `readEntireFile` that accepts a `Scanner` representing an input file as its parameter, then reads that file and returns its entire text contents as a `String`.

7. Write a method called `flipLines` that accepts a `Scanner` for an input file and writes to the console the same file's contents with each pair of lines reversed in order. If the file contains an odd number of lines, leave the last line unmodified. For example, if the file contains:

```
Twas brillig and the slithy toves
did gyre and gimble in the wabe.
All mimsey were the borogroves,
and the mome raths outgrabe.
```

your method should produce the following output:

```
did gyre and gimble in the wabe.
Twas brillig and the slithy toves
and the mome raths outgrabe.
All mimsey were the borogroves,
```

8. Write a method called `doubleSpace` that accepts a `Scanner` for an input file and a `PrintStream` for an output file as its parameters, writing into the output file a double-spaced version of the text in the input file. You can achieve this task by inserting a blank line between each line of output.

9. Write a method called `wordWrap` that accepts a `Scanner` representing an input file as its parameter and outputs each line of the file to the console, word-wrapping all lines that are longer than 60 characters. For example, if a line contains 112 characters, the method should replace it with two lines: one containing the first 60 characters and another containing the final 52 characters. A line containing 217 characters should be wrapped into four lines: three of length 60 and a final line of length 37.

10. Modify the preceding `wordWrap` method so that it outputs the newly wrapped text back into the original file. (Be careful—don't output into a file while you are reading it!) Also, modify it to use a class constant for the maximum line length rather than hard-coding 60.

11. Modify the preceding `wordWrap` method so that it only wraps whole words, never chopping a word in half. Assume that a word is any whitespace-separated token and that all words are under 60 characters in length.

12. Write a method called `stripHtmlTags` that accepts a `Scanner` representing an input file containing an HTML web page as its parameter, then reads that file and prints the file's text with all HTML tags removed. A tag is any text between the characters < and >. For example, consider the following text:

```
<html>
<head>
<title>My web page</title>
</head>
<body>
<p>There are many pictures of my cat here,
as well as my <b>very cool</b> blog page,
which contains <font color="red">awesome
stuff about my trip to Vegas.</p>
```

```
Here's my cat now:<img src="cat.jpg">
</body>
</html>
```

If the file contained these lines, your program should output the following text:

```
My web page

There are many pictures of my cat here,
as well as my very cool blog page,
which contains awesome
stuff about my trip to Vegas.

Here's my cat now:
```

You may assume that the file is a well-formed HTML document and that there are no < or > characters inside tags.

13. Write a method called `stripComments` that accepts a `Scanner` representing an input file containing a Java program as its parameter, reads that file, and then prints the file's text with all comments removed. A comment is any text on a line from `//` to the end of the line, and any text between `/*` and `*/` characters. For example, consider the following text:

```
import java.util.*;

/* My program
by Suzy Student */
public class Program {
    public static void main(String[] args) {
        System.out.println("Hello, world!"); // a println
    }

    public static /* Hello there */ void foo() {
        System.out.println("Goodbye!"); // comment here
    } /* */
}
```

If the file contained this text, your program should output the following text:

```
import java.util.*;

public class Program {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
```

```
    public static  void foo() {
        System.out.println("Goodbye!");
    }
}
```

14. Write a method called `printDuplicates` that takes as a parameter a `Scanner` containing a series of lines. Your method should examine each line looking for consecutive occurrences of the same token on the same line and print each duplicated token, along with the number of times that it appears consecutively. Nonrepeated tokens are not printed. You may ignore the case of repetition across multiple lines (such as if a line ends with a given token and the next line starts with the same token). You may assume that each line of the file contains at least 1 token of input. For example, consider the following input:

```
hello how how are you you you you
I I I am Jack's Jack's smirking smirking smirking smirking revenge
bow wow wow yippee yippee  yo yippee  yippee yay yay yay
one fish two fish red fish blue fish
It's the Muppet Show, wakka wakka wakka
```

Your method should produce the following output:

```
how*2 you*4
I*3 Jack's*2 smirking*4
wow*2 yippee*2 yippee*2 yay*3

wakka*3
```

15. Write a method called `coinFlip` that accepts a `Scanner` representing an input file of coin flips that are heads (H) or tails (T). Consider each line to be a separate set of coin flips and output the number and percentage of heads in that line. If it is more than 50%, print "You win!". Consider the following file:

```
H T H H T
T t    t  T h  H
```

For the input above, your method should produce the following output:

```
3 heads (60.0%)
You win!

2 heads (33.3%)
```

16. Write a method called `mostCommonNames` that accepts a `Scanner` representing an input file with names on each line separated by spaces. Some names appear multiple times in a row on the same line. For each line, print the most commonly occurring name. If there's a tie, use the first name that had that many occurrences; if all names are unique, print the first name on the line. For example, if the file has this input:

```
Benson Eric   Eric  Kim  Kim Kim   Jenny  Nancy Nancy  Paul  Paul
Ethan  Jamie Jamie Alyssa Alyssa Helene  Helene Jessica Jessica
```

For the input above, your method should produce the following output:

```
Most common: Kim
Most common: Jamie
```

17. Write a method called `inputStats` that accepts a `Scanner` representing an input file and reports the number of lines, the longest line, the number of tokens on each line, and the length of the longest token on each line. If the file contains the following text:

```
Beware the Jabberwock, my son,
the jaws that bite, the claws that catch,
Beware the JubJub bird and shun
the frumious bandersnatch.
```

For the input above, your method should produce the following output:

```
Line 1 has 5 tokens (longest = 11)
Line 2 has 8 tokens (longest = 6)
Line 3 has 6 tokens (longest = 6)
Line 4 has 3 tokens (longest = 13)
Longest line: the jaws that bite, the claws that catch,
```

18. Write a method called `plusScores` that accepts a `Scanner` representing an input file containing a series of lines that represent student records. Each student record takes up two lines of input. The first line has the student's name and the second line has a series of plus and minus characters. Below is a sample input:

```
Kane, Erica
--+-+
Chandler, Adam
++-+
Martin, Jake
+++++++
```

For each student you should produce a line of output with the student's name followed by a colon followed by the percent of plus characters. For the input above, your method should produce the following output:

```
Kane, Erica: 40.0% plus
Chandler, Adam: 75.0% plus
Martin, Jake: 100.0% plus
```

19. Write a method called `leetSpeak` that accepts two parameters: a `Scanner` representing an input file, and a `PrintStream` representing an output file. Convert the input file's text to "leet speak," where various letters are replaced by other letters/numbers, and output the new text to the given output file. Replace `"o"` with `"0"`, `"l"` (lowercase "L") with `"1"` (the number one), `"e"` with `"3"`, `"a"` with `"4"`, `"t"` with `"7"`, and an `"s"` at the end of a word with `"Z"`. Preserve the original line breaks from the input. Also wrap each word of input in parentheses. For example, if the input file contains the following text:

```
four score and
seven years ago our
```

```
fathers brought forth on this continent
a new nation
```

For the input above, your method should produce the following in the output file:

```
(f0ur) (sc0r3) (4nd)
(s3v3n) (y34rZ) (4g0) (0ur)
(f47h3rZ) (br0ugh7) (f0r7h) (0n) (7hiZ) (c0n7in3n7)
(4) (n3w) (n47i0n)
```

Programming Projects

1. Students are often asked to write term papers containing a certain number of words. Counting words in a long paper is a tedious task, but the computer can help. Write a program that counts the number of words, lines, and total characters (not including whitespace) in a paper, assuming that consecutive words are separated either by spaces or end-of-line characters.

2. Write a program that compares two files and prints information about the differences between them. For example, consider a file `data1.txt` with the following contents:

```
This file has a great deal of
text in it which needs to

be processed.
```

Consider another file `data2.txt` that exists with the following contents:

```
This file has a grate deal of
text in it which needs to

bee procesed.
```

A dialogue with the user running your program might look like the following:

```
Enter a first file name: data1.txt
Enter a second file name: data2.txt
Differences found:
Line 1:
< This file has a great deal of
> This file has a grate deal of

Line 4:
< be processed.
> bee procesed.
```

3. Write a program that prompts the user for a file name, assuming that the file contains a Java program. Your program should read the file and print its contents properly indented. When you see a left-brace character ({) in the file,

increase your indentation level by four spaces. When you see a right-brace character (}), decrease your indentation level by four spaces. You may assume that the file has only one opening or closing brace per line, that every block statement (such as `if` or `for`) uses braces rather than omitting them, and that every relevant occurrence of a { or } character in the file occurs at the end of a line. Consider using a class constant for the number of spaces to indent (4), so that it can easily be changed later.

4. Write a program that reads a file containing data about the changing popularity of various baby names over time and displays the data about a particular name. Each line of the file stores a name followed by integers representing the name's popularity in each decade: 1900, 1910, 1920, and so on. The rankings range from 1 (most popular) to 1000 (least popular), or 0 for a name that was less popular than the 1000th name. The following lines are a sample of the file format:

```
Sally 0 0 0 0 0 0 0 0 0 0 886
Sam 58 69 99 131 168 236 278 380 467 408 466
Samantha 0 0 0 0 0 0 272 107 26 5 7
Samir 0 0 0 0 0 0 0 0 920 0 798
```

Your program should prompt the user for a name and search the file for that name:

```
This program allows you to search through the
data from the Social Security Administration
to see how popular a particular name has been
since 1900.

Name? Sam
```

If the name is found, the program should display data about the name on the screen:

```
Statistics on name "Sam"
    1900: 58
    1910: 69
    1920: 99
    1930: 131
    ...
```

This program is more fun and challenging if you also draw the name's popularity on a `DrawingPanel` as a line graph. Plot the decades on the *x*-axis and the popularity on the *y*-axis.

5. Write a program that plays a game where a player is asked to fill in various words of a mostly complete story without being able to see the rest. Then the user is shown his/her story, which is often funny. The input for your program is a set of story files, each of which contains "placeholder" tokens surrounded by < and >, such as:

```
One of the most <adjective> characters in fiction is named
"Tarzan of the <plural-noun> ." Tarzan was raised by a/an
<noun> and lives in the <adjective> jungle in the
heart of darkest <place> .
```

The user is prompted to fill in each of the placeholders in the story, and then a resulting output file is created with the placeholders filled in. For example:

```
Input file name? story1.txt
Please enter an adjective: silly
Please enter a plural noun: socks
Please enter a noun: tree
Please enter an adjective: tiny
Please enter a place: Canada
```

The resulting output story would be:

```
One of the most silly characters in fiction is named
"Tarzan of the socks ." Tarzan was raised by a/an
tree and lives in the tiny jungle in the
heart of darkest Canada .
```