Chapter 9 Input and Output

After studying this chapter, you should be able to:

- Use the `Scanner` class to read information from files and the `PrintWriter` class to write information to files
- Locate files using absolute and relative paths
- Use the `Scanner` class to obtain information from the user
- Prompt users for information and check it for errors
- Give users more control over programs by using command interpreters
- Put commonly used classes in a package
- Use a dialog box to obtain a filename from the user
- Display an image stored in a file

Computers excel at managing large amounts of data. The data might consist of credit card transactions at your bank, student records at your school, the card catalog at your local library, or a song on your computer. Such data is stored in a file on a hard drive or similar device.

In this chapter, we will learn how to work with files: How to open a file, process the data, and close the file.

These same techniques form the basis for interacting with the program's user via text. The program displays text to the user; the user types text to the program. Learning these basic techniques still has value, even in an age of graphical user interfaces. Writing a graphical user interface is difficult and, for many programs, simply isn't worth the trouble. Text interfaces are often sufficient.

9.1 Basic File Input and Output

KEY IDEA

Files are often organized using records.

Each time you visit a Web page with your browser, the Web server responsible for delivering that page records what it does in a log. On a busy Web server, this log can grow to include millions of records. On the computer hosting my personal home page, one week of log entries during a quiet time of year resulted in more than 360,000 records. A **record** refers to one entry within the file consisting of several related pieces of information. Each piece of information in a record is called a **field**. For example, a typical record[1] in the server's log contains the information shown in Table 9-1. This particular record shows that someone at the University of Massachusetts looked at a graphic on my Web page on August 19, 2005.

(table 9-1)

Information from a typical record in a Web server's log

Field Contents	Meaning
`128.119.246.74`	The **IP address**, or Internet Protocol address, of the computer requesting the Web page.
`vinci5.cs.umass.edu`	The **host name** of the computer requesting the Web page. The host name and the IP address are largely interchangeable. One is easier for computers; the other is easier for people.
`2005/8/19@11:24:14`	The date and time the Web page was served.
`GET`	The command that came from the browser requesting the page. Other commands include `POST` (used for pages with forms) and `PUT` (used for uploading data).
`/~bwbecker/mandel/ Gods_Eye_Heart.GIF`	The specific file that was requested. In this case, it isn't a Web page at all but a graphic that is part of a Web page. Once you know the name of the Web server (`www.cs.uwaterloo.ca`), you can reconstruct the requested URL and look at it with a browser (*www.cs.uwaterloo.ca/~bwbecker/mandel/Gods_Eye_Heart.GIF*).
`200`	The completion code. A code that begins with `2` indicates that the request completed normally.
`135215`	The size of the requested file. If the server encountered an error, the size is replaced with a dash (–).

In this chapter, we will write a series of programs that can be used to explore a Web server's log. If you have a personal home page, you may want to obtain a log to see what you can learn about who is accessing your page and how frequently your page is requested.

[1] The format of the record has been adjusted slightly. The program that does so is included with the examples for this chapter in the directory `formatLog`. The changes consist of removing several uninteresting fields, looking up the IP address to obtain the host name, and reformatting the date.

When I used these programs to explore the server log for my personal home page, I was surprised by how many times my page was accessed—612 times in one week! A little further investigation revealed that at least 140 of these were generated by search engines building their databases. I noticed that a professor at my alma mater accessed my Web page, presumably to find my e-mail address (I received an e-mail from him later that week). I was also surprised at the number of international hits (including Finland, South Africa, Australia, Israel, Singapore, Bosnia/Herzegovina, Netherlands, and Mexico). It was interesting to speculate how these visitors found my home page and what kind of information they were seeking.

9.1.1 Reading from a File

The program in Listing 9-1 provides a first look at a program that processes a file, which involves three important steps:

- Lines 13–21 locate the file on the disk drive and construct a `Scanner` object to obtain the information it contains. This process is called **opening** a file.
- Lines 24–29 process the file one record at a time, printing selected records. It uses two methods in the `Scanner` class, `hasNextLine` and `nextLine`. Obtaining data from a file is called **reading** a file. The information obtained from the file is called the program's **input**.
- Line 32 **closes** the file when it is no longer being used.

These three steps will be explored in detail in the following sections.

FIND THE CODE

ch09/processLines/

Listing 9-1: *A program to read a Web server's log and print records containing a given string*

```
1  import java.io.File;
2  import java.io.FileNotFoundException;
3  import java.util.Scanner;
4
5  /** Read a Web server's log record by record. Print those records that contain the
6  *  substring "bwbecker".
7  *
8  *  @author Byron Weber Becker */
9  public class ReadServerLog
10 {
11   public static void main(String[] args)
12   { // Open the file.
13     Scanner in = null;
14     try
15     { File file = new File("server_log.txt");
16       in = new Scanner(file);
17     } catch (FileNotFoundException ex)
```

PATTERN

Open File for Input

Listing 9-1: *A program to read a Web server's log and print records containing a given string* (continued)

```
18      { System.out.println(ex.getMessage());
19        System.out.println("in " + System.getProperty("user.dir"));
20        System.exit(1);
21      }
22
23      // Read and process each record.
24      while (in.hasNextLine())
25      { String record = in.nextLine();
26        if (record.indexOf("bwbecker") >= 0)      // author's Web pages
27        { System.out.println(record);
28        }
29      }
30
31      // Close the file.
32      in.close();
33    }
34  }
```

PATTERN

Process File

Opening a File

Conceptually, opening a file is simple. All we want to do is execute the following two lines:

```
File file = new File("server_log.txt");
Scanner in = new Scanner(file);
```

The first line creates a `File` object that describes where the program should look for the file named `server_log.txt`. The second line creates an object used to access the file at that location.

If only it were that simple. In reality, things can go wrong. The most common problem, and the only one that throws a checked exception, is when the file is not at the expected location. The programmer may have misspelled the name as `serverlog.txt`, the file may have been moved, the program may be running in an unexpected location, or the file may not have been created yet. In any of these cases, the `Scanner` constructor will throw a `FileNotFoundException`. Handling this exception expands the two lines we need to execute into nine lines in Listing 9-1.

LOOKING BACK

Exceptions were discussed in Section 8.4.

First, we need to introduce a `try-catch` statement around the `Scanner` constructor call to handle the `FileNotFoundException`. We will need the `Scanner` object outside of the `try-catch` statement, and so it is declared in line 13.

KEY IDEA

The working directory is useful information when a file is not found.

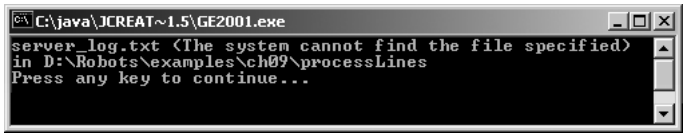
Second, it is wise to handle the exception by giving the user as much information as possible about where the program was looking for the missing file. This is done by getting and printing the **working directory** in line 19. The working directory is the directory (directories are also called folders) from which a program begins looking for a file. The working directory is set when the program begins execution. The working directory can be obtained with the query `System.getProperty("user.dir")`.

This code results in a message similar to the one shown in Figure 9-1. The message says the system started looking for the file with a disk drive labeled `D:`. That drive has a directory named `Robots`. Inside `Robots` is a directory named `examples`, which contains a directory named `ch09`. Inside `ch09` is `processLines`. That is the directory where the program expected to find the file named `server_log.txt`.

For the time being, we'll assume that in such circumstances you will simply move the file to the directory where the system expects to find it. Later, we will learn how to open files in other locations.

(figure 9-1)

Example of the message printed when a file is not found



Processing a File

Lines 24–29 in Listing 9-1 are responsible for processing the data in the file. Many files, including the server log, are organized as one record per line of text, as shown in Figure 9-2. The requested filenames are shortened so that each record fits on one line.

(figure 9-2)

Four records from the `server_log.txt` *file*

```
131.107.0.106 tide536.microsoft.com 2005/8/19@11:24:13 GET /~zqu/…enu.jpg 301 354
128.119.246.74 vinci5.cs.umass.edu 2005/8/19@11:24:14 GET /~bwb…rt.GIF 200 135215
210.8.90.45 cam1.gw.connect.com.au 2005/8/19@11:24:16 GET /~hza…zed.jpg 200 54297
131.107.0.106 tide536.microsoft.com 2005/8/19@11:24:16 GET /~zqu/…enu.jpg 302 326
```

The `nextLine` method, used in line 25 of Listing 9-1, retrieves one line from the file. With each repetition of the loop, it obtains the next line. This continues as long as `hasNextLine` returns `true`. When the last line has been read, `hasNextLine` will return `false` and the loop will stop.

Finally, the `if` statement contained within the loop prints out only those lines that contain the string `bwbecker`—that is, it prints out the log records pertaining to the author's Web pages.

Closing a File

Files use significant resources. Closing the file with the `close` method (line 32) allows the system to free up those resources for other uses.

9.1.2 Writing to a File

KEY IDEA

Writing a file is the opposite of reading it.

The previous program simply displays selected records in the console window. If a large number of records are selected, the first records will scroll out of view long before the last records are displayed. An alternative is for the program to copy the selected records to their own file. The process of creating a file and placing records in it is called **writing** a file. The information written is called the program's **output**. The terms *input* and *output* are often used together and abbreviated as **I/O**.

The program in Listing 9-2 is the same as the previous program except that it writes the selected records to a file named `bwbecker.txt` instead of printing them on the console. As with reading, there are three steps to writing the file:

- The file is opened at line 18 by constructing an instance of `PrintWriter`. This object opens the file and provides methods for writing data to it. Like opening a file to read it, a `FileNotFoundException` can be thrown. Therefore, the constructor call is placed inside the `try-catch` statement but the variable, `out`, is declared earlier, in line 15.
- The selected records are written to the file, one record at a time, in line 29. The `PrintWriter` class provides the same methods as `System.out`, including `print`, `println`, and `printf`.
- Finally, the file is closed at line 36.

These changes are shown in bold.

FIND THE CODE

ch09/processLines/

Listing 9-2: *A program that writes matching records to a file*

```
1  import java.io.File;
2  import java.io.FileNotFoundException;
3  import java.io.PrintWriter;
4  import java.util.Scanner;
5
6  /** Read a Web server's access log record by record. Write those records that contain the
7  *  substring "bwbecker" to a file.
8  *
9  *  @author Byron Weber Becker */
10 public class WriteMatchingLines
11 {
12   public static void main(String[] args)
```

Listing 9-2: *A program that writes matching records to a file* (continued)

```
13    { // Open the files.
14      Scanner in = null;
15      PrintWriter out = null;
16      try
17      { in = new Scanner(new File("server_log.txt"));
18        out = new PrintWriter("bwbecker.txt");
19      } catch (FileNotFoundException ex)
20      { System.out.println(ex.getMessage());
21        System.out.println("in " + System.getProperty("user.dir"));
22        System.exit(1);
23      }
24
25      // Read and process each record.
26      while (in.hasNextLine())
27      { String record = in.nextLine();
28        if (record.indexOf("bwbecker") > 0)
29        { out.println(record);
30        }
31
32      }
33
34      // Close the files.
35      in.close();
36      out.close();
37    }
38  }
```

Open File for Input
Open File for Output

Process File

KEY IDEA

Ensure that all data is written by calling `close` *before the program ends.*

Java does not always write information to the file immediately. By collecting information from several calls to `print` and `println` and writing them all at once, substantial gains in efficiency can be realized. This process is called **buffering**. Some information may not be written to the file at all if the program ends at the wrong time. To prevent this, you should always call the `close` method after you are done writing to the file. It is an error to call a print method after `close` has been called.

What happens if the preceding code is executed again and the file `bwbecker.txt` already exists? The existing file and all the information within it will be deleted, as a new file with the same name is created.

Sometimes you would rather append new data to the end of an existing file. In that case, an extra step is required. Replace line 18 with the following two lines:

```
FileWriter fw = new FileWriter("bwbecker.txt", true);
out = new PrintWriter(fw);
```

The first line constructs an object that opens the file so that new data will be appended to it. However, the `FileWriter`'s methods only write individual characters; we still want to be able to use the `print` methods in `PrintWriter`. Fortunately, the two classes can work together to provide this capability.

LOOKING AHEAD

Java's I/O classes are designed to work together. More details in Section 9.7.

If your program uses these two lines but the specified file does not exist, a new file will be created.

9.1.3 The Structure of Files

Consider the following records from the inventory file of a computer store. The four fields are quantity on hand, part identifier, description, and price.

```
10 002D9249 Computer  1595.99
5 293E993C Keyboard   24.99
12 0003922M Monitor   349.99
```

The two programs examined in this chapter so far read such files as lines of text. In fact, text has a richer structure.

A file is a sequence of characters. The characters that are displayed visibly on the screen include letters, numbers, and punctuation, such as `y`, `M`, `8`, and `?`. Each of these is represented in the computer using a unique value.

Some characters are less obvious, such as spaces. They are represented on the screen as empty space. In the computer, however, they are represented by a value, just as `M` and `y` are represented by a value. For clarity, we will often show a space as a single dot in the middle of the line (·). Most word processors have a similar feature to help users understand how a document is formatted.

KEY IDEA

Every character, even spaces, corresponds to a value.

Another less obvious character is the tab character. Like the space character, a tab is also displayed by blank space. The length of that blank space, however, depends on a number of factors. But no matter how long the space is, it is represented in the computer as a single value. For clarity, we will show a tab character with a small arrow: →.

Finally, the end of a line is also represented by a character. The exact value used depends on the computer's operating system, and some use a sequence of two characters. Fortunately, the `Scanner` and `PrintWriter` classes allow us to ignore this detail most of the time. We will refer to this character as the **newline character**. It is displayed on the screen by moving the **insertion point**—the point where the next character is displayed—to the left side of the screen and down one line. For clarity, we will show the newline character as a down and left arrow: ↵.

The space, tab, and newline characters are collectively known as **whitespace** because they appear as white space when printed on a white sheet of paper.

Finally, we will represent the end of the file[2] with □.

With these conventions, the three inventory records are shown as follows:

```
10·002D9249·Computer→1595.99↵
5·293E993C·Keyboard→24.99↵
12·0003922M·Monitor→349.99↵
□
```

KEY IDEA

Lines are divided into tokens separated by delimiters.

Lines of text are often divided into groups of characters called **tokens**. The characters that divide one token from the next are called **delimiters**. The most common delimiters are white space characters. Using white space as delimiters, the previous lines each contain four tokens. Dividing the line into tokens enables us to obtain the information it contains more flexibly.

Data Acquisition Methods

The `Scanner` class provides methods to read a file token by token as well as line by line. The `next` method will read the next token, returning it as a `String`. Calling the `next` method on the inventory records will return, in order, the strings `10`, `002D9249`, `Computer`, and so on.

Another method, `nextInt`, will attempt to read the next token and convert it to an integer before returning it. If the next token can't be converted to an integer, `nextInt` will throw an `InputMismatchException`. A third method, `nextDouble`, behaves similarly except that it attempts to convert the next token to a `double` value. These methods can be described as **data acquisition methods** because they are used to acquire data from the file.

A program fragment that reads the inventory records and prints a simple report is shown in Listing 9-3. It assumes a `Scanner` object named `in` has already been created.

FIND THE CODE

ch09/inventoryReport/

Listing 9-3: *A program fragment that reads the tokens in an inventory record*

```
1 // Open the file.
2 while (in.hasNextLine())
3 { int quantity = in.nextInt();
4   String partID = in.next();
5   String description = in.next();
6   double cost = in.nextDouble();
7   in.nextLine();
8
```

[2] Actually, the end of the file is not a character in the same way that a space or newline is a character. Nevertheless, showing it as a character is a useful fiction.

Listing 9-3: *A program fragment that reads the tokens in an inventory record* (continued)

```
 9      // Print in a different order, including a calculated value.
10      System.out.printf("%-15s%5d%8.2f%10.2f%n", description,
11                        quantity, cost, quantity * cost);
12  }
```

LOOKING BACK

The `printf` method was discussed in Section 7.2.4.

As this program reads the file, the `Scanner` object maintains a **cursor** that marks its position. The cursor divides the file into two parts: the part that has already been read, and the part that has not. The cursor is positioned just before the first character when the file is opened.

Table 9-2 traces part of the execution of the previous program. It shows the position of the cursor with a diamond (♦) in the column labeled "Input."

(table 9-2)

Tracing the partial execution of the program fragment in Listing 9-3

Statement	Input	quantity	partID	descr	cost
	♦10·002D9249·Computer→1595.99↵				
2 `while (in.hasNextLine())`					
	♦10·002D9249·Computer→1595.99↵				
3 `{ int quantity = in.nextInt();`					
	10♦·002D9249·Computer→1595.99↵	10			
4 `String partNum = in.next();`					
	10·002D9249♦·Computer→1595.99↵	10	002D9249		
5 `String description = in.next();`					
	10·002D9249·Computer♦→1595.99↵	10	002D9249	Computer	
6 `double cost = in.nextDouble();`					
	10·002D9249·Computer→1595.99♦↵	10	002D9249	Computer	1595.99
7 `in.nextLine();`					
	10·002D9249·Computer→1595.99↵ ♦5·293E993C·Keyboard→24.99↵	10	002D9249	Computer	1595.99
10 `System.out.printf...` *2* `while (in.hasNextLine())`					
	10·002D9249·Computer→1595.99↵ ♦5·293E993C·Keyboard→24.99↵	10	002D9249	Computer	1595.99

Statement	Input	quantity	partID	descr	cost
`3 { int quantity = in.nextInt();`					
	`10·002D9249·Computer1→1595.99↵` `5♦·293E993C·Keyboard→24.99↵`	5	002D9249	Computer	1595.99

(table 9-2) *continued*

Tracing the partial execution of the program fragment in Listing 9-3

Beginning at the top of Table 9-2, the `while` statement calls `hasNextLine` to determine whether additional text comes after the cursor. `hasNextLine` does not move the cursor.

When the `nextInt` method executes, it begins at the cursor and looks ahead at the following characters. It skips any leading delimiters such as spaces, and then examines the characters until the next delimiter character is found. In this example, these characters are `10`, which can be interpreted as an integer. The cursor is therefore moved just past the token, and the integer `10` is returned. If the characters cannot be interpreted as an integer, an exception is thrown and the cursor does not move.

The `Scanner` class contains methods to read and interpret the next token for many types. They all behave essentially the same as `nextInt`:

- Skip delimiting characters.
- Examine the characters up to the next delimiter.
- If the examined characters can be interpreted as the specified type, move the cursor beyond them and return the token as the specified type. If the characters cannot be interpreted as the specified type, throw a `InputMismatchException` and leave the cursor's position unchanged.

KEY IDEA

`nextLine` *does not skip leading white space. Other* `next` *methods do.*

The exception is the `nextLine` method. It does not skip leading white space and returns the rest of the line rather than a token.

The most commonly used data acquisition methods in the `Scanner` class are shown in Table 9-3. In this table, each method is followed by a description and examples.

(table 9-3)

Data acquisition methods in the `Scanner` *class*

Method	Description and Examples

`int nextInt()`	Examines the next token in the input, skipping any leading delimiters. If the token can be interpreted as an `int`, the cursor is moved past the token and the `int` value is returned. Otherwise, an `InputMismatchException` is thrown and the cursor is not moved. Examples:
	
	Initial Situation / Returns / Final Situation
	`ABC♦··10·DEF↵` / `10` / `ABC··10♦·DEF↵`
	`ABC♦·-15↵` / `-15` / `ABC·-15♦·↵`
	`ABC♦·ten·DEF↵` / `Exception` / `ABC♦·ten·DEF↵`
	`ABC♦·↵10·DEF` / `10` / `ABC·↵10♦·DEF`
	
	Please note that the last example contains a newline character ↵ in the middle of the line. A text editor would show this as two lines.

`double nextDouble()`	Like `nextInt`, but attempts to interpret the token as a `double`. Examples:
	
	Initial Situation / Returns / Final Situation
	`ABC♦··10.5·DEF↵` / `10.5` / `ABC··10.5♦·DEF↵`
	`ABC♦·-1.5E3·DEF↵` / `-1500.0` / `ABC·-1.5E3♦·DEF↵`
	`ABC♦·10·DEF↵` / `10.0` / `ABC·10♦·DEF↵`
	`ABC♦·ten·DEF↵` / `Exception` / `ABC♦·ten·DEF↵`

`boolean nextBoolean()`	Like `nextInt`, but attempts to interpret the token as a `boolean`. Examples:
	
	Initial Situation / Returns / Final Situation
	`ABC♦··true·DEF↵` / `true` / `ABC··true♦·DEF↵`
	`ABC♦·FALSE↵` / `false` / `ABC·FALSE♦·↵`
	`ABC♦·truest·DEF↵` / `Exception` / `ABC♦·truest·DEF↵`

`String next()`	Reads the next token and returns it as a `String`. Examples:
	
	Initial Situation / Returns / Final Situation
	`ABC♦··xyz·DEF↵` / `"xyz"` / `ABC··xyz♦·DEF↵`
	`ABC♦·FALSE↵` / `"FALSE"` / `ABC·FALSE♦·↵`
	`ABC♦·10·DEF↵` / `"10"` / `ABC·10♦·DEF↵`
	`ABC♦□` / `Exception` / `ABC♦□`
	`ABC♦↵··xyz·DEF` / `"xyz"` / `ABC↵··xyz♦·DEF`

`String nextLine()`	Reads and returns as a `String` all the characters from the cursor up to the next newline character or the end of the file, whichever comes first. Moves the cursor past the characters that were read and the following newline, if there is one. `nextLine` does not skip leading delimiters. Examples:
	
	Initial Situation / Returns / Final Situation
	`ABC♦··xyz·DEF↵` / `"··xyz·DEF"` / `ABC··xyz·DEF↵♦`
	`ABC♦··xyz·DEF□` / `"··xyz·DEF"` / `ABC··xyz·DEF♦□`
	`ABC♦□` / `Exception` / `ABC♦□`

KEY IDEA

Choose a method based on the desired return type.

Many tokens may be read with more than one method. For example, the token `10` can be read with `nextInt`, `nextDouble`, and `next`. It can also be read with `nextLine`, which may also include additional tokens. The difference is in the type returned. `nextInt` returns the token as an `int`, ready to be assigned to an integer variable. `next`, on the other hand, returns it as a `String`, which can be assigned to a variable of type `String` but not a variable of type `int`.

Data Availability Methods

KEY IDEA

`hasNextInt` is used to determine if calling `nextInt` will succeed.

In addition to the data access methods shown in Table 9-3, the `Scanner` class has data availability methods. `hasNextLine` is one of these methods. **Data availability methods** are used to determine whether data of a given type is available. Each of the data acquisition methods have a corresponding data availability method that can be used to determine if calling the data acquisition method will succeed.

These methods include `hasNext`, `hasNextInt`, `hasNextDouble`, `hasNextBoolean`, and `hasNextLine`. They all return `boolean` values.

For an example of using a data availability method, consider again the Web server log. Normally, the last token of the record is an integer specifying the size of the data served. However, if the server encounters an error and cannot serve the requested data, the log will contain a dash (`-`). If `nextInt` is called on such a record, an exception will be thrown.

Instead, use `hasNextInt` to determine if an integer is available. If it is, call `nextInt` to acquire it. If `hasNextInt` returns `false`, we can read the information another way. This is shown in Listing 9-4 in lines 10–15.

Listing 9-4: *A code fragment to read individual tokens in a Web server's log, accounting for either an integer size or a dash (–) in the last token*

```
1   while(in.hasNextLine())
2   { String ipAddress = in.next();
3     String hostName = in.next();
4     String when = in.next();
5     String cmd = in.next();
6     String url = in.next();
7     int completionCode = in.nextInt();
8
9     // Read the size of the served page. Set size to 0 if there was an error recorded.
10    int size = 0;
11    if (in.hasNextInt())
12    { size = in.nextInt();                    // Read the size.
13    } else
```

Listing 9-4: *A code fragment to read individual tokens in a Web server's log, accounting for either an integer size or a dash (–) in the last token* (continued)

```
14        { in.next();                                  // Skip the dash.
15        }
16        in.nextLine();                                // Move cursor to next line.
17
18        // Process the data.
19    }
```

9.2 Representing Records as Objects

Reading one record can become complex, as Listing 9-4 indicates. The complexity only gets worse as the number of fields and the number of exceptions increase as in lines 11–15. All that code can obscure our understanding of the enclosing loop and the code processing the records.

An excellent way to address these issues is to write helper methods. An even better solution is to write a new class so that each record can be represented as an object. The helper methods go in that class.

KEY IDEA

Represent records as objects.

9.2.1 Reading Records as Objects

For an example of representing a record as an object, consider the `ServerRecord` class shown in Listing 9-5. The helper method to read the file is actually the constructor. It takes a `Scanner` object as a parameter and uses it to read the information for one server log record. The code opening the file, the loop reading multiple records, and the code closing the file are in another class (see Listing 9-6).

Instance variables in `ServerRecord` correspond to the fields in the record. Each field is stored in the appropriate variable when it is read.

Note that the date and time from the record is stored as a `DateTime` object. Furthermore, the `DateTime` class has a constructor taking a `Scanner` object as a parameter. This allows the `ServerRecord` constructor to quickly and easily delegate reading the date and time to the `DateTime` class in line 25.

This technique assumes that the constructor is called with the `Scanner`'s cursor positioned immediately before the record. When the constructor finishes executing, the cursor must be immediately after the record, ready for the next record to be read.

KEY IDEA

The constructor begins and ends with the cursor at the beginning of a record.

`ServerRecord` should also provide methods required to process the record. Examples might include methods to get the size of the served page, to determine if the URL contains a specified string, to determine if the hostname contains a specified string, or to get the date the page was served.

The `ServerRecord` class also includes a method named `write` that writes the record to a file in the same format in which it was read. This allows the program to read its own files. `write` takes a `PrintWriter` object as its parameter. As with the reading of the file, the responsibility for opening and closing the file rests with the calling code (see Listing 9-6).

FIND THE CODE

ch09/processRecords/

Listing 9-5: *A class representing records in a Web server's log*

```
1  import java.util.Scanner;
2  import java.io.PrintWriter;
3  import becker.util.DateTime;
4
5  /** Represent one server log record.
6  *
7  *  @author Byron Weber Becker */
8  public class ServerRecord extends Object
9  {
10   private String ipAddress;
11   private String hostName;
12   private DateTime when;
13   private String cmd;
14   private String url;
15   private int completionCode;
16   private int size = 0;
17   private boolean error = true;     // Assume an error until proven otherwise.
18
19   /** Construct an object representing one server record using information read from a file.
20   *  @param in An open file, positioned at the beginning of the next record. */
21   public ServerRecord(Scanner in)
22   { super();
23     this.ipAddress = in.next();
24     this.hostName = in.next();
25     this.when = new DateTime(in);
26     this.cmd = in.next();
27     this.url = in.next();
28     this.completionCode = in.nextInt();
29     if (in.hasNextInt())
30     { this.size = in.nextInt();
31       this.error = false;
32     }
33
```

LOOKING AHEAD

A class like `ServerRecord` is an excellent candidate for a library. See Section 9.6.

PATTERN

Construct Record from File

Listing 9-5: *A class representing records in a Web server's log* (continued)

```
34        // Get ready to read the next record
35        in.nextLine();
36     }
37
38     /** Write the record to a file in the same format it was read.
39      * @param out An open output file. */
40     public void write(PrintWriter out)
41     { out.print(this.ipAddress + " " + this.hostName + " ");
42       out.print(this.when.toString() + " " + this.cmd + " ");
43       out.print(this.url + " " + this.completionCode + " ");
44       if (this.error)
45       { out.print("-");
46       } else
47       { out.print(this.size);
48       }
49       out.println();
50     }
51
52     // Some methods have been omitted.
53  }
```

Listing 9-6: *Client code to read server records and write selected records to a file*

FIND THE CODE

ch09/processRecords/

```
1  import java.io.*;
2  import java.net.*;
3  import java.util.Scanner;
4
5  /** Read a Web server's access log. Write selected records to a file.
6  *
7  *   @author Byron Weber Becker */
8  public class ReadServerRecords
9  {
10   public static void main(String[] args)
11   { // Open the files.
12     Scanner in = null;
13     PrintWriter out = null;
14     try
15     { in = new Scanner(new File("server_log.txt"));
16       out = new PrintWriter("largeFiles.txt");
17     } catch (FileNotFoundException ex)
```

PATTERN

Open File for Input
Open File for Output

Listing 9-6: *Client code to read server records and write selected records to a file* (continued)

```
18      { System.out.println(ex.getMessage());
19        System.out.println("in " + System.getProperty("user.dir"));
20        System.exit(1);
21      }
22
23      // Read and process each record.
24      while (in.hasNextLine())
25      { ServerRecord sr = new ServerRecord(in);
26        if (sr.getSize() >= 25000)
27        { sr.write(out);
28        }
29      }
30
31      // Close the files.
32      in.close();
33      out.close();
34    }
35  }
```

Process File

9.2.2 File Formats

KEY IDEA

Every program using a file must agree on the file's format.

Many files are used by more than one program. For example, the Web server writes the log file while various reporting programs read it. These programs need to agree on how the file is organized: the order of the fields within the record, which delimiters are used to separate tokens, and so on. The organization of the file is known as the **file format**.

To better appreciate the effect the file format has on the program, let's consider again the simple file format for the computer store inventory file. Recall that it had four fields, as shown in the following example records:

```
10 002D9249 Computer  1595.99
5 293E993C Keyboard   24.99
12 0003922M Monitor   349.99
```

A constructor to read these records is quite simple:

FIND THE CODE

ch09/fileFormat/

Construct Record from File

```
public Inventory1(Scanner in)
{ super();
  this.quantity = in.nextInt();
  this.partID = in.next();
  this.description = in.next();
  this.price = in.nextDouble();
  in.nextLine();
}
```

However, this code assumes that each field consists of a single token. If the description were `LCD Monitor` instead of simply `Monitor`, this would not work because `in.next()` would read `LCD`. The call to `nextDouble()` would attempt to turn the string `Monitor` into a `double`, and fail.

KEY IDEA

Simple changes to the file format can make a big difference in the code that reads it.

The simplest way to handle this change is to change the file format. By putting single token fields such as quantity, price, and part identifier first, and putting the multiple token field (description) last, the description can be read using `nextLine`; in other words, order the record as shown in the following example:

```
12 0003922M 349.99 LCD Monitor
```

Code to read this file format can be found in `ch09/fileFormat/Inventory2.java`.

However, suppose that there is a second multiple token field, such as the name of the supplier. If we simply add it on to the end of the record, we have no reliable way of knowing where one field ends and the next begins unless we use a different delimiter that does not appear in either field, such as a colon (`:`). This is shown in the following record:

```
12 0003922M 349.99 LCD Monitor : ACME Computer Distributors
```

Such a record could be read with code such as the following. It reads the description a token at a time, building up the description until the delimiter is found. It then reads the last multiple token field with `nextLine`, trimming off any leading or trailing blanks.

FIND THE CODE

ch09/fileFormat/

```
public Inventory3(Scanner in)
{  // Code to read quantity, part identifier, and price is omitted
   this.description = "";
   String token = in.next();
   while (!token.equals(":"))
   { this.description += " " + token;
     token = in.next();
   }
   this.distributor = in.nextLine().trim();
}
```

The `Scanner` class takes this idea one step further by allowing us to specify the delimiters it uses. If we replace each white space delimiter with a colon, for example, then even multiword phrases are treated as a single token. Consider the following record:

```
12:M0003922:349.99:LCD Monitor:ACME Computer Distributors:
```

This record can be read by calling `in.useDelimiter(":")` immediately after opening the file. The `ServerRecord` constructor can now read each of the tokens with a single call, as follows:

FIND THE CODE

ch09/fileFormat/

```
public Inventory4(Scanner in)
{ this.quantity = in.nextInt();
  this.partID = in.next();
  this.price = in.nextDouble();
  this.description = in.next();
  this.distributor = in.next();
  in.nextLine();                    // Move to the next line of the file.
}
```

Because newline characters are no longer delimiters, the colon at the end of the record and the call to `nextLine` are required.

KEY IDEA

Design the file format to make your code easy to read, write, and understand.

There is one more file format variation that bears mentioning: Simply place each multiple token field like `description` and `distributor` on its own line. No law requires a record to use only one line. This simple idea of placing each multiple token field on its own line helps keep both the file and the code easy to read, write, and understand. To make the file easier to read, you may want to place a blank line between each pair of records.

9.3 Using the File Class

To open a file, a `File` object must be constructed and given the name of the file, such as `server_log.txt`. The resulting object can be passed to a `Scanner` constructor, but it can also be useful by itself. The sections that follow investigate valid filenames, explain how to specify file locations, and discuss the methods this class provides.

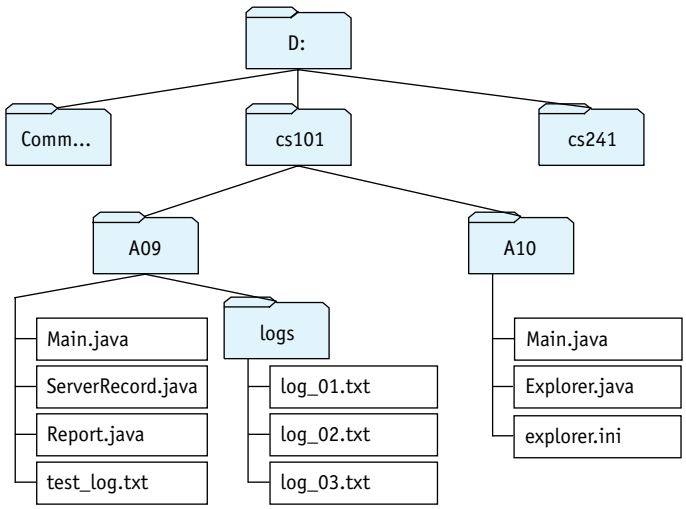
9.3.1 Filenames

You can't name a file anything you want because some characters are not allowed. The Windows operating system, for example, does not allow a filename to contain any of the following characters: \ / : * ? " < > |.

Filenames often have an **extension**, such as `.txt`. An extension is whatever follows the last period in the name, and is often used to identify the kind of information stored in the file. For example, a file with an extension of `.html` contains a Web page, whereas a file with an extension of `.jpg` means it contains a graphic.

9.3.2 Specifying File Locations

Modern computers use a hierarchical system for locating directories and files. The hierarchy is depicted as an upside-down tree with branches, as shown in Figure 9-3. Directories can contain either files (white) or other directories (green). For example, the directory `cs101` contains two other directories, `A09` and `A10`. The directory `A09` contains four files, including `ServerRecord.java`. `A09` also includes a directory, `logs`, which includes three additional files.



(figure 9-3)

Hierarchical file system in which folder icons represent directories and boxes represent files

An **absolute path** begins with the root of the tree (`D:`) and specifies all of the directories between it and the desired file. For example, the hierarchy shown in Figure 9-3 contains two files named `Main.java`. The following statement uses an absolute path to specify one of them:

```
File f = new File("D:/cs101/A09/Main.java");
```

The directories in the path are separated with a special character, typically `/` (Unix and Macintosh) or `\` (Windows). Java will accept either, but `/` is easier because `\` is Java's escape character for strings.

Files can also be specified with a **relative path** from the program's working directory. Suppose the current working directory is `A09`. A name without a prefix specifies a file in that directory—for example, `test_log.txt`. You can also name a file in a subdirectory of the working directory—for example, `logs/log_01.txt`. The special name `..` specifies the parent directory. The following statement uses a relative path to specify the initialization file in `A10`:

```
File init = new File("../A10/explorer.ini");
```

Relative paths are most useful when the program's location and the file's location are related. If the program moves to a new location (such as submitting it electronically to be marked), the file should also move. Absolute paths are more useful when the location of the file is independent of the location of the program using it.

Knowing your program's working directory is a key to using relative paths effectively. You can find it with the following statement:

```
System.out.println(System.getProperty("user.dir"));
```

The `System` class maintains a map of keys and properties for the running program. The string "`user.dir`" is the key for the working directory property. Other keys include "`user.name`" (the user's account name); "`os.name`" (the computer's operating system); and "`line.separator`" (the character or sequence of characters separating lines in a file, represented earlier with ↵).

9.3.3 Manipulating Files

The `File` constructor can have either an absolute or a relative path as its argument. The resulting object represents a path to a file or a directory. The file or directory may or may not exist.

A `File` object can both answer a number of useful questions about the path it represents and perform a number of operations on the file system. Some of these operations are summarized in Table 9-4. Technically, a directory is a special kind of file. The online documentation often uses "file" to refer to either; Table 9-4 does the same.

(table 9-4)

Summary of methods in the `File` *class*

Method	Description
`boolean canRead()`	Determines whether this program has permission to read from the file.
`boolean canWrite()`	Determines whether this program has permission to write to the file.
`boolean delete()`	Deletes the file or directory. Directories must be empty before they can be deleted. Returns `true` if successful.
`boolean exists()`	Determines whether the file exists.
`String getAbsolutePath()`	Gets the absolute path for this file.
`File getParentFile()`	Gets a `File` object representing this file's parent directory. Returns `null` if this file doesn't have a parent.
`boolean isFile()`	Determines whether the path specifies a file.

(table 9-4) *continued*

Summary of methods in the `File` *class*

Method	Description
`boolean isDirectory()`	Determines whether the path specifies a directory.
`long length()`	Gets the number of characters in the file.
`boolean mkDir()`	Makes the directory represented by this `File`. Returns `true` if successful.

9.4 Interacting with Users

Without input from users, most programs are not worth writing. A word processor that always typed the same essay, regardless of what the user wanted to say, would be worthless. A game program that didn't react to the game player's decisions would be boring.

This section and the next will discuss how to interact with the user of your program to modify how the program behaves. As an example, we will modify the program in Listing 9-6, which currently processes a server log, printing only those records resulting from serving a file larger than or equal to 25,000 bytes. Our new program will ask the user which log file to process and the minimum served file size to report. In particular, we want to implement the following pseudocode:

```
String fileName = ask the user for the log file to open
int minSize = ask the user for the minimum served file size
open the log file named in fileName
while (log file has another line)
{ ServerRecord sr = new ServerRecord(log file);
  if (sr.getSize() >= minSize)
  { print the record
  }
}
close the log file
```

9.4.1 Reading from the Console

Fortunately, the techniques we learned to read from a file can also be used to read information from the console. We still use the `Scanner` class, but we construct the `Scanner` object slightly differently, as follows:

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

`System.in` is an object similar to `System.out`. `Scanner` uses it to read from the console. Unlike opening a file, we are not required to catch any exceptions.

Before we implement the pseudocode discussed earlier, consider the sample program shown in Listing 9-7. It illustrates the important elements of reading from the console. The result of running this program is shown in Figure 9-4.

FIND THE CODE

ch09/readConsole/

Listing 9-7: *A short program demonstrating reading from the console*

```
1  import java.util.Scanner;
2
3  public class ReadConsole
4  {
5    public static void main(String[] args)
6    { Scanner cin = new Scanner(System.in);
7
8      System.out.print("Enter an integer: ");
9      int a = cin.nextInt();
10     System.out.print("Enter an integer: ");
11     int b = cin.nextInt();
12
13     System.out.println(a + "*" + b + " = " + a * b);
14   }
15 }
```

The program begins by creating a new `Scanner` object used to read from the console. It's named `cin`, short for "console input."

KEY IDEA

Prompt the user when input is expected.

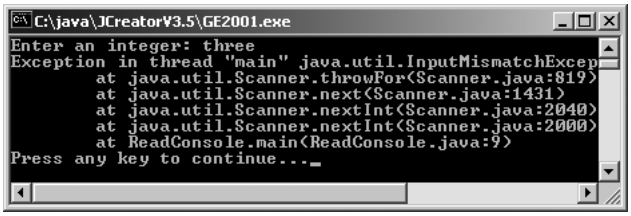
At lines 8 and 10, the program prints a **prompt** for the user. The prompt informs the user that input is expected. In Figure 9-4, the user responded to the first prompt with `3↵`. That is, the user entered the digit 3 and the Enter key. The Enter key is the user's cue to the program that it should read the input and process it. The program waits to read the input until Enter is pressed. The online documentation uses the term **block**, which means to wait for input.

(figure 9-4)

Result of running the program shown in Listing 9-7

9.4.2 Checking Input for Errors

Unfortunately, if the user misreads the prompt and enters `three↵`, the program will throw an exception, as shown in Figure 9-5.



(figure 9-5)

Result of entering data with an inappropriate type

The program can be protected from such errors with the code shown in Listing 9-8. The loop in lines 9–19 verifies that the next token is an integer. If it is not, the program reads it and displays a helpful message. This action gives the user the opportunity to try again. When an integer is entered, it is read in line 12, and the loop ends with the `break` in line 14.

Listing 9-8: *Rewriting Listing 9-7 to check for input errors*

```
1  import java.util.Scanner;
2
3  public class ReadConsoleChecked
4  {
5    public static void main(String[] args)
6    { Scanner cin = new Scanner(System.in);
7
8      int a = 0;
9      while (true)
10     { System.out.print("Enter an integer: ");
11       if (cin.hasNextInt())
12       { a = cin.nextInt();
13         cin.nextLine();                    // consume remaining input
14         break;
15       } else
16       { String next = cin.nextLine();    // consume the error
17         System.out.println(next + " is not an integer such as 10 or -3.");
18       }
19     }
20     // Repeat lines 8–19, but read the data entered into variable b.
33
34     System.out.println(a + " * " + b + " = " + a * b);
35   }
36 }
```

FIND THE CODE

ch09/readConsole/

PATTERN

Error-Checked Input

The need to repeat essentially identical code to read the second integer suggests that a method should be written. Such a method does not rely on any instance variables and can therefore be a class method. In fact, a whole set of similar methods will be needed.

LOOKING BACK

Class methods were discussed in Section 7.5.2

We can place them in a class named `Prompt` and call them as shown in the following example:

```
int a = Prompt.forInt("Enter an integer: ");
```

KEY IDEA

The `Prompt` class must be used for all of the console input—or none of it.

Listing 9-9 shows the beginning of the class. Notice that line 9 declares a `static` (class) variable used to read from the console. One consequence of this decision is that *all* input from the console must be obtained with the methods in this class. Using more than one `Scanner` object to read from the same source (that is, `System.in`) will not work reliably.

LOOKING AHEAD

The problem set will ask you to add to this library.

Listing 9-9 also includes the methods `forInputFile` and `forInputScanner`. The first method uses the `File` class to verify that a string entered by the user specifies a file that exists and can be read by this program. The second method uses the first to open the specified file using `Scanner`. Putting this code in its own class has the following advantages:

- We can avoid writing it anew for each program that asks the user for a file or integer to process.
- We can put the `try-catch` statement here, rather than cluttering the main program with it.
- If the methods need enhancing or debugging, there is only one place that requires attention.

FIND THE CODE

ch09/userIO/

Listing 9-9: *A class providing error-checked reading of an integer and a filename*

```
1  import java.util.Scanner;
2  import java.io.*;
3
4  /** A set of useful static methods for interacting with a user via the console.
5  *
6  *  @author Byron Weber Becker */
7  public class Prompt extends Object
8  {
9    private static final Scanner in = new Scanner(System.in);
10
11   /** Prompt the user to enter an integer.
12   *  @param prompt The prompting message for the user.
13   *  @return The integer entered by the user. */
14   public static int forInt(String prompt)
15   { while (true)
16     { System.out.print(prompt);
17       if (Prompt.in.hasNextInt())
18       { int answer = Prompt.in.nextInt();
19         Prompt.in.nextLine(); // consume remaining input
```

PATTERN

Error-Checked Input

Listing 9-9: *A class providing error-checked reading of an integer and a filename* (continued)

```
20          return answer;
21        } else
22        { String input = Prompt.in.nextLine();
23          System.out.println("Error: " + input
24                + " not recognized as an integer such as '10' or '-3'.");
25        }
26      }
27    }
28
29    /** Prompt the user for a file to use as input.
30    *  @param prompt The prompting message for the user.
31    *  @return A File object representing a file that exists and is readable. */
32    public static File forInputFile(String prompt)
33    { while (true)
34      { System.out.print(prompt);
35        String name = in.nextLine().trim();
36        File f = new File(name);
37        if (!f.exists())
38        { System.out.println("Error: " + name + " does not exist.");
39        } else if (f.isDirectory())
40        { System.out.println("Error: " + name + " is a directory.");
41        } else if (!f.canRead())
42        { System.out.println("Error: " + name + " is not readable.");
43        } else
44        { return f;
45        }
46      }
47    }
48
49    /** Prompt the user for a file to use as input.
50    *  @param prompt The prompting message for the user.
51    *  @return A Scanner object ready to read the file specified by the user. */
52    public static Scanner forInputScanner(String prompt)
53    { try
54      { return new Scanner(Prompt.forInputFile(prompt));
55      } catch (FileNotFoundException ex)
56      { // Shouldn't happen, given the work we do in forInputFile.
57        System.out.println(ex.getMessage());
58        System.exit(1);
59      }
60      return null; // for the compiler
61    }
62  }
```

Using `Prompt`

The completed program for interacting with the user to ask for a specific Web server log file to process and the minimum size of returned page to print in a report is shown in Listing 9-10. Notice that it uses the `Prompt` class in lines 11 and 14.

FIND THE CODE

ch09/userIO/

Listing 9-10: *A program that processes a Web server log based on user input*

```
1  import java.util.Scanner;
2
3  /** List files in a user-specified Web server log that meet a minimum size criteria.
4  *  Report the number of files that are printed.
5  *
6  *  @author Byron Weber Becker */
7  public class ListFilesBySize
8  {
9    public static void main(String[] args)
10   { // Prompt for the file to process.
11     Scanner in = Prompt.forInputScanner("Web server log name: ");
12
13     // Get the minimum size from the user.
14     int minSize = Prompt.forInt("Minimum served file size: ");
15
16     // Process the files.
17     int count = 0;
18     while (in.hasNextLine())
19     { ServerRecord sr = new ServerRecord(in);
20       if (sr.getSize() >= minSize)
21       { System.out.println(sr.toString());
22         count++;
23       }
24     }
25
26     // Close the input file and report the count.
27     in.close();
28     System.out.println(count + " files served were at least "
29                                + minSize + " bytes.");
30   }
31 }
```

9.5 Command Interpreters

The techniques for interacting with users shown in the previous section are adequate if the user must always enter the same information in the same order each time the program is run. There are many programs, however, for which more flexibility is desired. One way to achieve more flexibility (without the work of implementing a graphical user interface) is to write a **command interpreter**. A command interpreter repeatedly waits for the user to enter a command, and then it executes the command.

9.5.1 Using a Command Interpreter

To illustrate the principles involved, we will write a program called `LogExplorer`, which will process a Web server log, selecting records that meet criteria set by the user. The user interface will allow the user to:

- Specify a string to find in the client host name field.
- Specify a minimum served file size.
- Specify whether each matching record is shown.
- Specify whether the number of matching records is shown.
- Display a help message.
- Process a specified file with the current settings.

Furthermore, the structure of the program will make it easy to add functionality. An example of running the program is shown in Figure 9-6. The prompt for a command is >. Information required by the command is entered on the same line. As you can see, commands are usually terse.

(figure 9-6)

Running a program that uses a command interpreter

9.5.2 Implementing a Command Interpreter

The pseudocode for a command interpreter is as follows:

LOOKING AHEAD

A graphical user interface uses a similar loop, called an "event loop."

```
while (the quit command has not been received)
{ get a command
  execute the command
}
```

We get the command by prompting the user to enter a command and reading it from the console. Executing the command is done with a cascading-`if` statement, often in a separate method.

Unfortunately, a pair of commands like *host* and *host <string>* complicates matters. The problem lies with determining whether the user has entered a string following the *host* command. It would seem that calling `hasNext()` would easily resolve that question, but it doesn't—`Scanner` will keep looking up to the end of the "file" to see if there is a token present. But when scanning `System.in` (the console), there is no end of file. `Scanner` waits for whatever the user types in next (ignoring white space, including Enter). When the user does type something, `Scanner` returns `true`. `hasNextLine` has similar issues.

We can solve this problem by reading the input a line at a time—but then we have to find out what is on the line, as indicated by the following pseudocode:

Command Interpreter

```
while (the quit command has not been received)
{ System.out.print("> ");                    // Prompt for a command.
  String line = in.nextLine();
  get the command and argument (if there is one) out of the line
  execute the command
}
```

KEY IDEA

The `Scanner` class can also process strings.

Now there is the problem of extracting the information on the line to find the command (such as *host*, *min*, or *p*) and the argument (such as *googlebot.com* or *1500000*), if there is one. Fortunately, `Scanner` can scan strings in addition to files. For example, the following code will print "`host true googlebot.com`".

```
Scanner s = new Scanner("host googlebot.com");
System.out.print(s.next());
System.out.print(s.hasNext());
System.out.print(s.next());
```

The following code fragment will print "`host false`".

```
Scanner s = new Scanner("host");
System.out.print(s.next());
System.out.print(s.hasNext());
```

By creating a `Scanner` for each line we read, we can determine exactly what is on it. With this insight, we can structure the command interpreter as follows:

```
Scanner cin = new Scanner(System.in);
while (the quit command has not been received)
{ System.out.print("> ");
  String line = cin.nextLine();
  Scanner lineScanner = new Scanner(line);
  String cmd = lineScanner.next();
  if (cmd is "host" and line has another token)
  { remember given hostname for next search
  } else if (cmd is "host")
  { next search will be for all hosts
  } else if (cmd is "min" and line has an integer)
  { remember given minimum size for the next search
  } else if (cmd is "p" and line has another token)
  { process the given file with the settings given by previous commands
  ...
  } else
  { error message
  }
}
```

These ideas are implemented in the class shown in Listing 9-11. The command interpreter is at lines 23–31; it delegates the task of executing the commands to `executeCmd`, lines 35–59.

The heart of the actual application is the `processFile` method. It's overloaded, with one method taking a `String` parameter (the filename) and another taking a `Scanner` object. The first one handles the messy details of opening the file and then calls the second one, which actually does the work. It reads each record in the log, printing and counting those that match the criteria. The task of deciding which records match is delegated to `includeRecord`.

`LogExplorer` works as shown, but the design could be improved by separating the user interface from the rest of the program. Section 9.5.3 explains how.

Listing 9-11: *The* `LogExplorer` *program with an integrated command interpreter*

```
1  import java.io.*;
2  import java.util.Scanner;
3
4  /** Explore a Web server log by displaying/counting records meeting user-specified criteria.
5  *
6  *   @author Byron Weber Becker */
7  public class LogExplorer extends Object
8  {
```

FIND THE CODE

ch09/logExplorer/

Listing 9-11: *The* `LogExplorer` *program with an integrated command interpreter* (continued)

```
 9    // Search criteria
10    private String searchHost = "";          // String to find in hostname.
11    private int minSize = 0;                 // Minimum size of returned page.
12
13    private boolean done = false;            // Received quit command yet?
14    private boolean displayNum = true;       // Display # of matching records?
15    private boolean displayRec = true;       // Display each matching record?
16
17    /** Create a new explorer object; displays all log records by default. */
18    public LogExplorer()
19    { super();
20    }
21
22    /** Interpret the commands entered by the user. */
23    public void cmdInterpreter()
24    { this.displayHelp();
25      Scanner cin = new Scanner(System.in);
26      while (!this.done)
27      { System.out.print("> ");
28        String line = cin.nextLine();
29        this.executeCmd(line);
30      }
31    }
32
33    /** Execute one line entered by the user.
34    *   @param cmdLine The one line of command and optional arguments to execute. */
35    private void executeCmd(String cmdLine)
36    { Scanner line = new Scanner(cmdLine);
37      if (line.hasNext())
38      { String cmd = line.next();
39        if (cmd.equals("host") && line.hasNext())
40        { this.searchHost = line.next();
41        } else if (cmd.equals("host"))
42        { this.searchHost = "";
43        } else if (cmd.equals("min") && line.hasNextInt())
44        { this.minSize = line.nextInt();
45        } else if (cmd.equals("p") && line.hasNext())
46        { this.processFile(line.next());
47        } else if (cmd.equals("q"))
48        { this.done = true;
49        } else if (cmd.equals("help"))
50        { this.displayHelp();
51        } else if (cmd.equals("dr") && line.hasNextBoolean())
```

Command Interpreter

Listing 9-11: *The* `LogExplorer` *program with an integrated command interpreter* (continued)

```
52          { this.displayRec = line.nextBoolean();
53          } else if (cmd.equals("dn") && line.hasNextBoolean())
54          { this.displayNum = line.nextBoolean();
55          } else
56          { System.out.println("Command '" + line + "' not recognized.");
57          }
58        }
59      }
60
61      /** Process a log file via the specified Scanner object. Display and count each record that
62      *   matches the criteria set previously.
63      *   @param in A scanner for the input file to process. */
64      private void processFile(Scanner in)
65      { int count = 0;
66        while (in.hasNextLine())
67        { ServerRecord sr = new ServerRecord(in);
68          if (this.includeRecord(sr))
69          { if (this.displayRec)
70            { this.displayRecord(sr);
71            }
72            count++;
73          }
74        }
75        if (this.displayNum)
76        { this.displayCount(count);
77        }
78      }
79
80      /** Process the specified file.
81      *   @param fName The name of the file to process. */
82      private void processFile(String fName)
83      { Scanner in = null;
84        try
85        { in = new Scanner(new File(fName));
86          this.processFile(in);
87          in.close();
88        } catch (FileNotFoundException ex)
89        {  System.err.println("Can't find file " +
90              System.getProperty("user.dir") + "/" + fName + ".");
91        }
92      }
93
94      /** Determine whether a record should be included in the report. Include records that meet
```

Listing 9-11: *The* `LogExplorer` *program with an integrated command interpreter* (continued)

```
95      *  ALL the specified criteria. If a criterion wasn't set (for example, no client host name was
96      *  specified), we need a way to ignore it, typically with an "or" condition. */
97      private boolean includeRecord(ServerRecord sr)
98      { return (this.searchHost.length() == 0 ||
99                sr.hostnameContains(this.searchHost))
100               && sr.getSize() >= this.minSize;
101     }
102
103     /** Display one record to the user.
104     *  @param sr The record to display. */
105     private void displayRecord(ServerRecord sr)
106     { System.out.printf("%-15s %s%n",
107             sr.getIPAddress(), sr.getClientHostname());
108       System.out.printf(" %5d%10d%5s %s%n",
109             sr.getCompletionCode(), sr.getSize(),
110             sr.getCommand(), sr.getURL());
111     }
112
113     /** Display the number of matching records.
114     *  @param count The number of matching records to display. */
115     private void displayCount(int count)
116     { System.out.println();
117       System.out.println(count + " records met the search criteria.");
118     }
119
120     /** Display a help message. */
121     private void displayHelp()
122     { final String helpFmt = " %-14s %s%n";
123       final PrintStream out = System.out;
124       out.println("General Commands:");
125       out.printf(helpFmt, "q", "Quit");
126       out.printf(helpFmt, "help", "Display this help message");
127       out.printf(helpFmt, "p <string>", "Process specified file");
128       out.println();
129       out.println("Commands that affect which records are included:");
130       out.printf(helpFmt, "host <string>", "Hostnames including...");
131       out.printf(helpFmt, "host", "All client hostnames");
132       out.printf(helpFmt, "url <string>", "Requested URLs including...");
133       out.printf(helpFmt, "url", "All URLs");
134       out.printf(helpFmt, "min <int>", "Served pages with a minimum size");
135       out.println();
136       out.println("Commands that affect how records are displayed:");
137       out.printf(helpFmt, "dn <boolean>", "Display number of records");
```

Listing 9-11: *The* `LogExplorer` *program with an integrated command interpreter* (continued)

```
138       out.printf(helpFmt, "dr <boolean>", "Display records");
139       out.println();
140     }
141 }
```

9.5.3 Separating the User Interface from the Model

The `LogExplorer` program shown in Listing 9-10 combines the code that solves the problem (the model) with the code that interacts with the user (the user interface, composed of a view and controller). By correctly separating these two aspects of the program, we can achieve the following benefits:

- Separating these aspects makes the program easier to understand. In all but the simplest programs, it is easier to understand a program when each class has a specific focus. In this case, the model should focus on determining which records to display, and the user interface should focus on interacting with the user.
- Separating these aspects enables attaching different user interfaces to the model without changing the model. For example, an experienced user may use the program with a terse command interpreter that allows him to work quickly. An inexperienced user may use the program with a command interpreter that reads a command and then prompts for additional information. When a graphical user interface becomes available, the command interpreter can be replaced by it without changing the model.

LOOKING BACK

Models, views, and controllers were discussed in Section 8.6.2.

Identifying Parts That Belong to the User Interface

We can begin by reviewing Listing 9-11 and identifying the parts that have to do with communicating with the user. They include the following:

- The `cmdInterpreter` method in lines 22–31
- The `executeCmd` method in lines 33–59
- The `displayRecord` method in lines 103–111
- The `displayCount` method in lines 113–118
- The `displayHelp` method in lines 120–140
- The `processFile` method's error message at lines 89–91
- The other `processFile` method's two `if` statements that determine whether each record or the total number of records is displayed to the user (lines 69 and 75)

- Three instance variables—`done`, `displayNum`, and `displayRec`—that control user interface functions (lines 13–15)

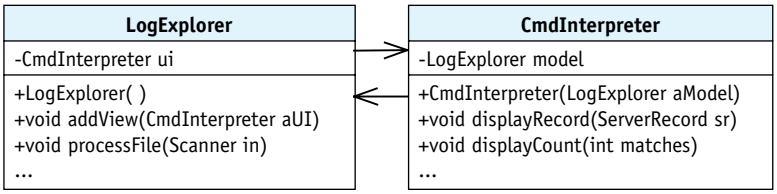
All of these elements will be moving to a new class named `CmdInterpreter`. With these parts gone, there won't be much left to the model (`LogExplorer`).

Setting Up Relationships between Classes

For these two classes to work together, `LogExplorer` will need to call the `displayRecord` and `displayCount` methods in `CmdInterpreter`. Similarly, the `CmdInterpreter` class will need to call methods such as `processFile` in the `LogExplorer` class. This implies that we need to set the classes up as shown in Figure 9-7.

(figure 9-7)

Structuring the `LogExplorer` program



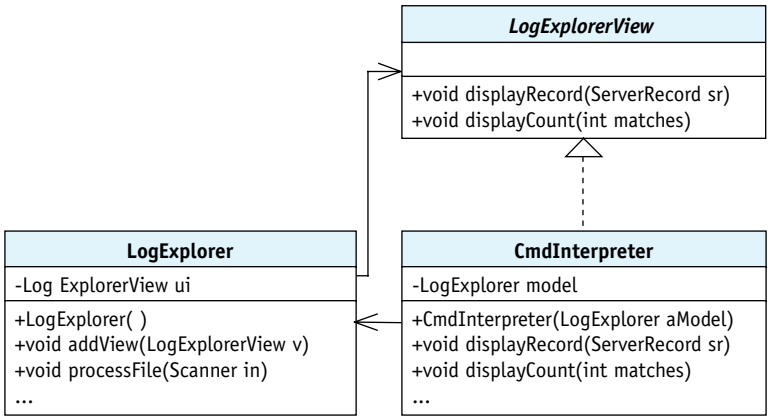
LOOKING BACK

Interfaces were discussed in Section 7.6.

However, implementing this class diagram will not allow us to change user interfaces, as we claimed earlier. With this implementation, the only user interface that can be used with `LogExplorer` is a class named `CmdInterpreter`. To fix this, we will define a Java interface declaring the methods `displayRecord` and `displayCount`. If `LogExplorer` uses this interface, then any user interface that implements it can be used with `LogExplorer`. Therefore, the high-level class diagram for the program will be the one shown in Figure 9-8.

(figure 9-8)

Structuring the `LogExplorer` program with a Java interface



The program can be set up with these relationships using the following code fragments. The `main` method has the responsibility to create both the `LogExplorer` and the `CmdInterpreter` objects, as follows:

```
1  public static void main(String[] args)
2  { LogExplorer explorer = new LogExplorer();
3    CmdInterpreter cmd = new CmdInterpreter(explorer);
4
5    cmd.cmdInterpreter();          // Run the command interpreter.
6  }
```

Notice that a reference to the `LogExplorer` object is passed to the `CmdInterpreter` constructor. This is used to set up the "`CmdInterpreter` has-a `LogExplorer`" relationship shown in Figure 9-8.

The "`LogExplorer` has-a `LogExplorerView`" relationship is set up when the `CmdInterpreter` constructor calls `LogExplorer`'s `addView` method with itself as the parameter, as follows:

```
1  public class CmdInterpreter extends Object
2                               implements LogExplorerView
3  {
4    private LogExplorer model;
5
6    public CmdInterpreter(LogExplorer aModel)
7    { super();
8      this.model = aModel;
9      this.model.addView(this);
10   }
11 }
```

The `CmdInterpreter` class implements the `LogExplorerView` interface in line 2, as expected by the `LogExplorer` class.

Finishing Up

Just a little bit of cleanup remains to complete the reorganization of the `LogExplorer` program.

LOOKING AHEAD

Programming Project 9.10 asks you to complete the implementation.

First, the original user interface simply set the `searchHost` and `minSize` instance variables directly. Now that these variables and the code that sets them are in different classes, we'll need to add some methods to `LogExplorer` to give appropriate access to the variables. The methods, of course, are called from the user interface.

Second, the `processFile` method cannot remain `private`. The user interface will need to call it at the appropriate times.

Third, the original program determined inside the `processFile` method whether to display each matching record and whether to display the count of matching records. This, however, is more appropriately a function of the user interface. One way to handle this is for `processFile` to always call `displayRecord` and `displayCount`. These methods, in the user interface, can each include an `if` statement to determine whether they do anything.

9.6 Constructing a Library

In this chapter, the `ServerRecord` and `Prompt` classes have been used in several different programs. So far, this has been handled by simply making a copy of the class for each program that uses it. This, however, is not a good idea. Suppose the class needs to change because a bug is found, an enhancement is needed, or the file format is changed. In each of these cases, multiple copies of the class must be changed—an error-prone process that is a waste of time and energy.

KEY IDEA

Most programming languages have a way to make a library of reusable code. In Java, it's done with packages.

A better solution is to place reusable code—such as `ServerRecord` and `Prompt`—into a **library**. A library is a collection of resources that are meant to be used in many different programs. The concept of a library is implemented in Java with **packages**. A package is a collection of related classes that may contain subpackages. The classes in `becker.robots` constitute a library, as do the classes in `becker.util` and `java.awt`. A program accesses the classes in a package with the `import` statement.

Understanding how programs are compiled and run is important background for understanding how to use packages. Integrated development environments often hide these details, so we begin with a brief tour of compiling and running programs using a command line.

9.6.1 Compiling without an IDE

Most computers have a way to run programs from a command line. A simple program (one that uses only imports from the `java` and `javax` packages or no imports at all) can be compiled and run, as shown in Figure 9-9. The example is `ListFilesBySize`, taken from Listing 9-10 with two minor changes. First, the `ServerRecord` class has been modified to use a `String` instead of the `DateTime` class (to simplify the first example; it will be put back afterwards). Second, the program itself has been modified to print only the number of matching records to reduce the amount of output.

(figure 9-9)

Compiling a program with no `import` *statements*

The five commands shown perform the following actions:

- `cd` changes into a new directory specified by the path `D:/cs101/ch09/userIO/`.
- `ls` lists the contents of the directory, showing the data file and the three `.java` files that make up the program.
- `javac` runs the Java compiler to translate the `.java` files into a form more easily understood by the computer. It puts the translation of each file into a file with the same name but replaces `.java` with `.class`. The `*.java` means any file in the working directory that ends with `.java`.
- `ls` lists the contents of the directory again. It shows the same files as before plus the newly created `.class` files.
- `java` runs the compiled program. It is told where to search for the program's `.class` files with `-classpath D:/cs101/ch09/userIO/`. The first part, `-classpath`, tells Java that the following path is the directory to search. The following three lines are the result of running the program.

A single dot (`.`) is an abbreviation for the current working directory. Therefore, a more succinct replacement for `-classpath D:/cs101/ch09/userIO/` is `-classpath .`.

With this background, we are now ready to return to understanding how to easily reuse classes by placing them in packages.

9.6.2 Creating and Using a Package

The `package` statement places a class into a named package. For example, the `Robot` class begins with the following statement:

```
package becker.robots;
```

Classes in the `becker.robots` package can be used by including the familiar statement `import becker.robots.*;`.

The package name should be unique. The recommended way to make it unique is to reverse your e-mail address. The author could use the reverse of `bwbecker@learningwithrobots.com`, as shown in the following package statement:

```
package com.learningwithrobots.bwbecker;
```

The `package` statement must be the first statement in the class, before any `import` statements or the `class` statement. However, comments may come before the `package` statement.

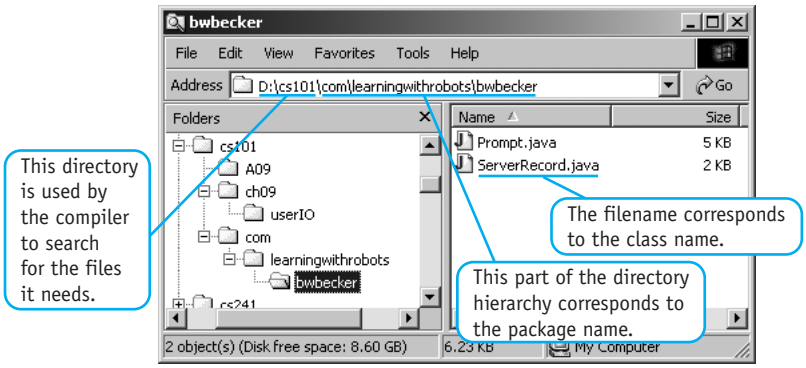
The `package` statement places constraints on the location of the file containing the source code. With the above `package` statement, the `ServerRecord` class must be located in the file `com/learningwithrobots/bwbecker/ServerRecord.java`. Notice that this path and name has three parts:

- The package name, but with the dots replaced with the directory separator character `/`
- The name of the class, `ServerRecord`
- The extension, `.java`

This is shown in Figure 9-10. Notice that the path beginning with `com/learn...` is not the entire path. We will need to tell the compiler about the first part, `D:/cs101/`.

(figure 9-10)

File locations for a package named `com.learningwithrobots.bwbecker`



In summary, compared to the program compiled in Figure 9-9, we need to make the following changes:

- Add a `package` statement to `ServerRecord.java` and a similar one to `Prompt.java`.
- Move `ServerRecord.java` and `Prompt.java` to the locations specified by their new `package` statements. The location of `ServerRecord.java` is shown in Figure 9-10.
- Modify `ListFilesBySize.java` to `import` the classes in the new package.

We can compile the revised program as shown in Figure 9-11.

```
bwbecker

>cd d:/cs101/ch09/userIO/

>ls
ListFilesBySize.java  server_log.txt

>javac -sourcepath d:/cs101/ *.java

>ls
ListFilesBySize.class  server_log.txt
ListFilesBySize.java

>java -classpath "d:/cs101/;." ListFilesBySize
Server log to process: server_log.txt
Minimum file size: 1000000
5 files are at least 1000000 bytes.

>
```

(figure 9-11)

Compiling the `ListFilesBySize` *program using packages*

The five commands shown perform the following actions:

- `cd` changes to the directory containing our program.
- `ls` shows that the directory now contains only `ListFilesBySize`. The other files have been moved to the library to facilitate easy reuse by many programs.
- `javac` runs the Java compiler. The compiler is told where to search for the packages it needs with `-sourcepath D:/cs101/`. When the compiler attempts to import `com.learningwithrobots.bwbecker.ServerRecord`, it looks in `D:/cs101/com/learningwithrobots/bwbecker/`.
- `ls` shows that only the `.class` file for `ListFilesBySize` has been added to the current directory. The other files were also compiled, but their `.class` files were left with the corresponding `.java` files.
- `java` runs the program. Now, because our class files are in several different places, the `-classpath` option is more complex. It lists two paths, separated by a semicolon (`;`). The second path is the current working directory, abbreviated with "`.`". Because of the semicolon, the entire list must be placed in double quotes. The first path specifies where to start the search for classes in the `bwbecker` package. The second path specifies where to find `ListFilesBySize`.

The preceding explanation may seem complex, but it's worth it because we can now write many programs that use the `ServerRecord` and `Prompt` classes. No matter how many programs use them, we have only one copy of the `.java` files—that is, only one copy to enhance or debug. Furthermore, we can use them simply by specifying a source path and a class path to the compiler. For commonly used classes, these are big advantages.

9.6.3 Using `.jar` Files

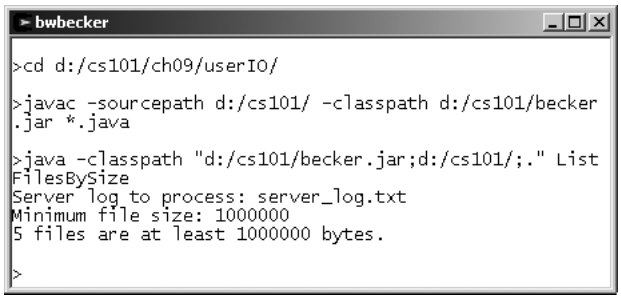
You may remember that we simplified `ServerRecord` somewhat to avoid using the `DateTime` class. To address that issue, assume that `ServerRecord` once again imports and uses `becker.util.DateTime`.

The complication is that we don't have the source code to `DateTime` like we have for `ServerRecord`. The `.class` files for `DateTime`, the robot classes, and many others are all stored in a single file named `becker.jar`. Storing them all in a `.jar` file makes them easier to distribute to other people. The `.jar` extension means **Java Archive**.

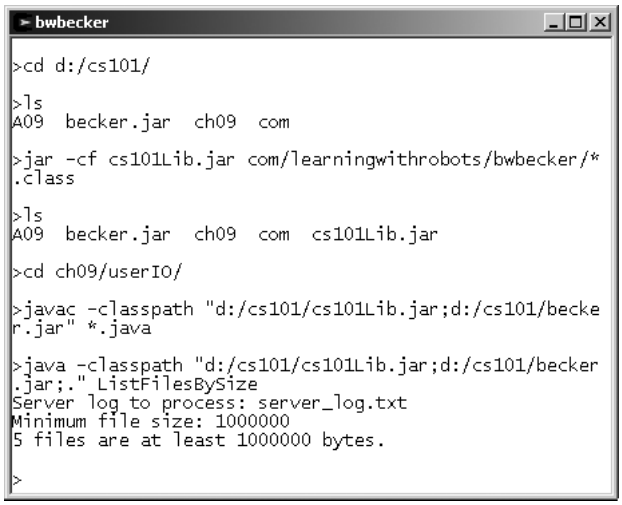
The `javac` command will need those `.class` files to compile the program. We tell it where to look for them by adding the `-classpath` option. Likewise, the `java` command will need the classes to run the program. Therefore, we must add the `.jar` file to its classpath as well. All this is shown in Figure 9-12. The `.jar` file is in the directory `D:/cs101/`.

(figure 9-12)

Compiling with classes in a `.jar` file



We may also want to put our own classes into a `.jar` file to make them easier to manage. This is done with a command named `jar`, as shown in Figure 9-13.



(figure 9-13)

Creating and using a `.jar` *file*

The crucial differences from what we've done before are as follows:

- `cd` changes to the directory we've been using for the source and class path for our package, `D:/cs101`.
- `jar` creates a new `.jar` file named `cs101Lib.jar`. The `.class` files to put into it are specified with the last part of the command, `*.class`. The asterisk (`*`) means to include every file ending in `.class`. In this situation, that would be `ServerRecord.class` and `Prompt.class`.
- `javac` and `java` now use class paths that include the new `.jar` file instead of `D:/cs101/`.

Fortunately, integrated development environments, which you probably use, know about source paths and class paths. Look for these terms among the IDE's settings; the documentation should explain how to include the path or `.jar` file for your personal library.

9.7 Streams (advanced)

Java's input and output library is based on the concept of **streams**. A stream is an ordered collection of information that moves from a **source** to a destination, or **sink**. It is similar to a stream of water flowing from a source (a spring or a lake) to a sink (the ocean).

Streams can be categorized using three questions:

- Is it an **input stream**, which carries information from a source to the program, or an **output stream**, which carries information from the program to a sink?

- Is it a **character stream**, which carries information in the form of 16-bit Unicode characters (usually human readable), or a **byte stream**, which carries binary information such as images or sounds in 8-bit bytes?
- Is it a **provider stream** that provides information from a source or to a sink, or is it a **processing stream** that processes or transforms information as it flows between a source and a sink?

Take `Scanner` as an example. `Scanner` is an input stream because the program uses it to get input from a source. It reads information as characters, either from a user or a file, and is thus a character stream. Finally, `Scanner` is a processing stream because it processes consecutive characters in the stream into higher-level constructs, such as an entire string or an integer. To do this, it makes use of an underlying provider stream. When we use `Scanner` to read from the console, for example, the provider stream is `System.in`.

`PrintWriter` is categorized much like `Scanner` except that it is an output stream that writes characters rather than bytes. It, too, is a processing stream because it processes higher-level constructs into the individual characters it writes out. Like `Scanner`, it uses an underlying provider stream to do the actual writing.

The philosophy of the Java I/O (input/output) library is that each class should do one thing well, and that each class should combine easily with other classes to obtain the strengths of both. Covering all the possibilities is beyond the scope of this textbook. Instead, we will provide an orientation and direct interested readers to other sources, such as the online Java Tutorial at *http://java.sun.com/docs/books/tutorial/index.html*.

9.7.1 Character Input Streams

The core class for character input is `Reader`. It's core method is `read`, which reads one or more characters from the source. `Reader` is extended to form four provider streams, corresponding to four different kinds of sources:

- `FileReader`: reads characters from a file
- `StringReader`: reads characters from a `String`; ultimately, this allows you to use `next`, `nextInt`, and similar methods in `Scanner` to read information from a `String` as easily as you can from a file.
- `CharArrayReader`: similar to `StringReader`
- `PipedReader`: reads characters that are produced by another thread in the program

Using these provider streams directly is not very easy. All they really provide is the ability to read a sequence of characters. A number of processing streams are provided to help form those characters into useful information. To use a processing stream, you must first construct a provider stream for it to use. Then, pass the provider stream to

the processing stream's constructor. For example, `BufferedReader` is a processing stream that has a method to read an entire line of characters at once. It could be used like this:

```
// Construct a buffered reader that processes input from a file
FileReader fileIn = new FileReader("phoneBook.txt");
BufferedReader buffIn = new BufferedReader(fileIn);

// Read and process each line of characters in the file.
String line = buffIn.readLine();
while (line != null)
{  // process the line here
   line = buffIn.readLine();
}
buffIn.close();
```

The processing streams that work with character input streams include:

- `BufferedReader`: reads an entire line of characters at once
- `LineNumberReader`: reads an entire line at once and provides a count of the lines read so far
- `PushbackReader`: allows a program to read some characters, examine them, and then push them back on the stream to be read again at a later time
- `Scanner`: divides the input stream into tokens and provides conversion of each token into appropriate types, such as `int`

`Scanner` is the most sophisticated of the processing streams. As we have already seen, it provides many methods to convert the raw stream of characters into useful information. It also has some convenience constructors to make it easier to use. For example, one constructor takes a string as an argument and automatically uses it to construct a `StringReader` to use as the source. Another constructor takes a `File` object as an argument and automatically constructs a `FileReader` to use as the source.

9.7.2 Character Output Streams

There are many similarities between character input streams and character output streams. Just as `Reader` is the core class for input, `Writer` is the core class for output, providing a `write` method to write one or more characters to the sink. It is extended four times, each class corresponding to four different kinds of sinks. The four subclasses are `FileWriter`, `StringWriter`, `CharArrayWriter`, and `PipedWriter`. A `FileWriter` writes characters to a file, of course. A `StringWriter` appends the characters to a string.

Only two interesting processing streams are associated with character output streams. One is `BufferedWriter`, which collects a large number of characters and then writes them all at once using its provider stream. `BufferedWriter` is typically combined

with a `FileWriter` because writing only one character at a time to a file is tremendously inefficient.

The other interesting processing stream is `PrintWriter`, which has already been discussed. It adds methods such as `print`, `println`, and `printf` to convert types such as `int` and `double` into individual characters.

9.7.3 Byte Streams

The structure of the byte input streams is similar to character input streams. A core class, `InputStream`, is extended four times to provide bytes from files, string buffers, byte arrays, and pipes. A similar set of classes provide processing streams to buffer the stream, count the lines, and push back information previously read.

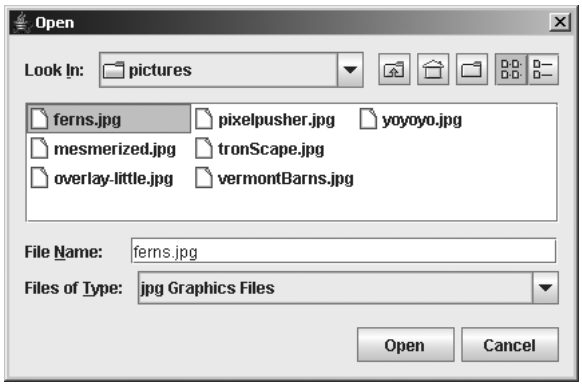
Similarly, the byte output streams mirror the character output streams. The core class, `OutputStream`, is extended to write bytes to files, byte arrays, and pipes. `BufferedOutputStream` and `PrintStream` are analogous to `BufferedWriter` and `PrintWriter`.

9.8 GUI: File Choosers and Images

Files are used with graphical user interfaces in a number of ways. One is to ask the user for a filename with an easy-to-use dialog box, as shown in Figure 9-14. Another is to read an image from a file and paint it on the screen. The image could have been created with a separate editor or taken with a camera.

(figure 9-14)

Dialog box displayed by `JFileChooser` *to choose a file*



9.8.1 Using `JFileChooser`

Using a professional dialog box to obtain a filename from a user is easy, thanks to the libraries that come with Java. The test program in Listing 9-12 displays a dialog box like the one shown in Figure 9-14.

In lines 13–14, the dialog box is created and shown to the user. The method `showOpenDialog` is meant for opening existing files. Another method, `showSaveDialog`, is for saving a file. The difference is the text placed on the buttons and the title bar.

The user's action is returned as an integer from `showOpenDialog`, and could be `APPROVE_OPTION` (the user chose a file), `CANCEL_OPTION` (the user cancelled the dialog without choosing a file), or `ERROR_OPTION` (an error occurred). If the user chooses a file, the full path and filename can be obtained with the code shown in line 19. Of course, your program should open and use the file instead of only printing the name on the console.

Listing 9-12: *A program demonstrating the use of* `JFileChooser`

```
1  import javax.swing.JFileChooser;
2  import java.io.File;
3
4  /** A program testing the operation of JFileChooser.
5  *
6  *  @author Byron Weber Becker */
7  public class Main extends Object
8  {
9    public static void main(String[] args)
10   { System.out.println("Ready to get a filename.");
11
12     // construct the dialog and show it to the user
13     JFileChooser chooser = new JFileChooser();
14     int result = chooser.showOpenDialog(null);
15
16     if (result == JFileChooser.APPROVE_OPTION)
17     { // Open the file and use it
18       System.out.println("You chose '" +
19               chooser.getSelectedFile().getPath());
20     }
21   }
22 }
```

FIND THE CODE

ch09/fileChooser/

Typically your program will be interested only certain kinds of files. For example, the next section shows how to display certain kinds of images on the screen. These images are normally stored in files that end with an extension of either `.gif` or `.jpg`. With the help of the `FileExtensionFilter` class, shown in Listing 9-13, `JFileChooser` will show only the relevant classes. Use it by adding the following lines between lines 13 and 14 in Listing 9-12:

```
chooser.addChoosableFileFilter(
          new FileExtensionFilter(".jpg", "jpg Graphics Files"));
chooser.addChoosableFileFilter(
          new FileExtensionFilter(".gif", "gif Graphics Files"));
```

`FileExtensionFilter` works by overriding the `accept` method in its superclass. `JFileChooser` calls this method once for each file or directory in the current directory. If `accept` returns `true`, the file or directory is displayed so the user can choose it.

FIND THE CODE

ch09/fileChooser/

Listing 9-13: *A filter used by* `JFileChooser` *to show only files with the specified extension*

```
1  import javax.swing.filechooser.FileFilter;
2  import java.io.File;
3
4  /** A class used to filter out some files so that JFileChooser only shows files with a
5  *  specified extension.
6  *
7  *  @author Byron Weber Becker */
8  public class FileExtensionFilter extends FileFilter
9  {
10   private String ext;
11   private String descr;
12
13   /** Accept files ending with the given extension.
14   *  @param extension The extension to accept (e.g., ".jpg")
15   *  @param description A description of the file accepted */
16   public FileExtensionFilter(String extension,
17                              String description)
18   { super();
19     this.ext = extension.toLowerCase();
20     this.descr = description;
21   }
22
23   /** Decide whether or not the given file should be displayed. In our case, include
24   *  directories as well as files with a name ending in the specified extension.
25   *  @param f A description of one file.
26   *  @return True if the file should be displayed to the user; false otherwise. */
```

Listing 9-13: *A filter used by* `JFileChooser` *to show only files with the specified extension* (continued)

```
27    public boolean accept(File f)
28    { return f.isDirectory() ||
29             f.getName().toLowerCase().endsWith(this.ext);
30    }
31
32    /** Return the description of the files accepted.
33    *  @return A description of the files this filter accepts.*/
34    public String getDescription()
35    { return this.descr;
36    }
37 }
```

9.8.2 Displaying Images from a File

The program in Listing 9-12 can be easily modified to display an image from a file. The program currently prints the name of the file selected by the user (see lines 18–19). The new program will replace lines 18–19 with the following code to create a component that displays an image it reads from a file, and then shows that component in a frame. Notice that the filename is obtained from the chooser and passed to `ImageComponent`'s constructor.

LOOKING BACK

This code uses the Display a Frame pattern. See Section 1.7.5.

```
// Create a component to display an image
ImageComponent imageComp = new
     ImageComponent(chooser.getSelectedFile().getPath());

// Display the image component in a frame
JFrame f = new JFrame("Image");
f.setContentPane(imageComp);
f.setSize(500, 500);
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
f.setVisible(true);
```

The source code for `ImageComponent` is more interesting and is shown in Listing 9-14. It is passed the name of the image file via the parameter in the constructor. A class from the Java library, `ImageIcon`, is used to read the image from the file. Supported types of images include `.gif`, `.jpg`, and `.png`. Once the image is loaded, the preferred size for the component is set to the image's size. If the preferred size is not set, the `JFrame` will make it so small that it can't be seen.

As with previous extensions of `JComponent`, the `paintComponent` method is overridden to do the painting. In the past, the `Graphics` parameter, `g`, has been used to call such methods as `drawRect` and `fillOval`. Here, it is used in line 24 to paint the

image read from the file. The second and third parameters give the desired location of the upper-left corner of the image. The zero values shown here put the image in the upper-left corner of the component.

`drawImage` is overloaded. Another version includes two more parameters to specify the painted image's width and height. This is useful if you want to scale the image. It is also possible to use `drawImage` to draw a background image and then add details on top with calls to `drawRect` and similar methods.

FIND THE CODE

ch09/displayImage/

Listing 9-14: *A new kind of component that displays an image from a file*

```
1  import java.awt.*;
2  import javax.swing.*;
3
4  /** A component that paints an image stored in a file.
5  *
6  * @author Byron Weber Becker */
7  public class ImageComponent extends JComponent
8  {
9    private ImageIcon image;
10
11   /** Construct the new component.
12   * @param fileName The file where the image is stored. */
13   public ImageComponent(String fileName)
14   { super();
15     this.image = new ImageIcon(fileName);
16     this.setPreferredSize(new Dimension(
17                           this.image.getIconWidth(),
18                           this.image.getIconHeight()));
19   }
20
21   /** Paint this component, including its image. */
22   public void paintComponent(Graphics g)
23   { super.paintComponent(g);
24     g.drawImage(this.image.getImage(), 0, 0, null);
25   }
26 }
```

9.9 Patterns

9.9.1 The Open File for Input Pattern

Name: Open File for Input

Context: You need to read information stored in a file.

Solution: Open the file and use `Scanner` to obtain the individual tokens within the file. The following template applies:

```
Scanner «in» = null;
try
{ «in» = new Scanner(new File(«fileName»));
} catch (FileNotFoundException ex)
{ System.out.println(ex.getMessage());
  System.out.println("in " + System.getProperty("user.dir"));
  System.exit(-1);
}
«statements to read file»
«in».close();
```

Consequences: The file is opened for reading. If the file does not exist, an exception is thrown and the program stops.

Related Patterns:

- The Open File for Output pattern is used to write information to a file.
- The Process File pattern depends on this pattern to open the file.

9.9.2 The Open File for Output Pattern

This pattern is almost identical to Open File for Input.

9.9.3 The Process File Pattern

Name: Process File

Context: You need to process all of the records in a data file, one after another.

Solution: Use an instance of `Scanner` to provide the records, one at a time. A `while` loop that tests for the end of the file controls the processing.

```
Scanner in = this.openFile(«fileName»);
while (in.hasNextLine())
{ «read one record»
  «process one record»
}

in.close();
```

LOOKING AHEAD

A factory method is simply a `static` *method that creates and returns an object. Listing 12-11 contains an example.*

A record is often represented by an instance of a class. Reading it is often best done in a constructor or factory method belonging to that class.

Consequences: The file is read from beginning to end. If it must be processed again, the file must be reopened to reset the file cursor back to the beginning of the file.

Related Patterns:

- This pattern uses the Open File for Input pattern to open the file. In the above template, the pattern would be implemented in the `openFile` method.
- The action of reading one record is often delegated using the Construct Record from File pattern.

9.9.4 The Construct Record from File Pattern

Writing this pattern is an exercise for the student in Written Exercise 9.2.

9.9.5 The Error-Checked Input Pattern

Name: Error-Checked Input

Context: Input from the user is required. Because the user may enter erroneous data, error checking is appropriate.

Solution: The pattern to use depends on the amount of error checking required. If only the correct type of data (integer, double, and so on) matters, then a simpler pattern will suffice. In the following, *«hasNext»* is a method such as `hasNextInt` or `hasNextDouble` in the `Scanner` class, and *«next»* is the corresponding method to get the next value, such as `nextInt` or `nextDouble`.

```
Scanner in = new Scanner(System.in);
...
System.out.print(«initialPrompt»);
while (!in.«hasNext»())          // type might be Int or Double or ...
{ in.nextLine();                 // skip offending input
  System.out.print(«errorPrompt»);
}
«type» «varName» = in.«next»();
```

If the value entered matters as well, then a more complex pattern is appropriate:

```
Scanner in = new Scanner(System.in);
...
«type» answer = «initialValue»;   // will contain the answer
boolean ok = false;
System.out.print(«prompt»);
while (!ok)
{ if (in.«hasNext»())
  { answer = in.«next»();
    if («testForCorrectValues»)
    { ok = true;
    } else
    { System.out.print(«incorrectValueErrorPrompt»);
    }
  } else
  { System.out.print(«incorrectTypeErrorPrompt»);
    in.nextLine();
 }
}
```

Consequences: The user will be repeatedly prompted until a correct value is input. There is no provision for the user to cancel the operation or otherwise break out of the loop. Due to the amount of code, this pattern is best contained in a method.

Related Pattern: This pattern may be used by the Command Interpreter pattern.

9.9.6 The Command Interpreter Pattern

Name: Command Interpreter

Context: You are writing a program in which the user can give commands from a prompt. You need to interpret the commands and execute code corresponding to each one.

Solution: Implement a command interpreter with the following pseudocode.

```
show initial instructions
while (the user is not done)
{ display current program state or prompt
  get a command from the user
  interpret and execute the command
}
```

Getting the command from the user is often as simple as getting one word or token using `Scanner`. The step to interpret and execute the command is usually performed with a cascading-`if` statement. A helper method should be used if more than one or two lines of code are needed to execute the command.

Consequences: By using a cascading-`if` statement, the task of interpreting and executing the commands is given a regular structure. This makes it easier to understand the program and to extend it with new commands.

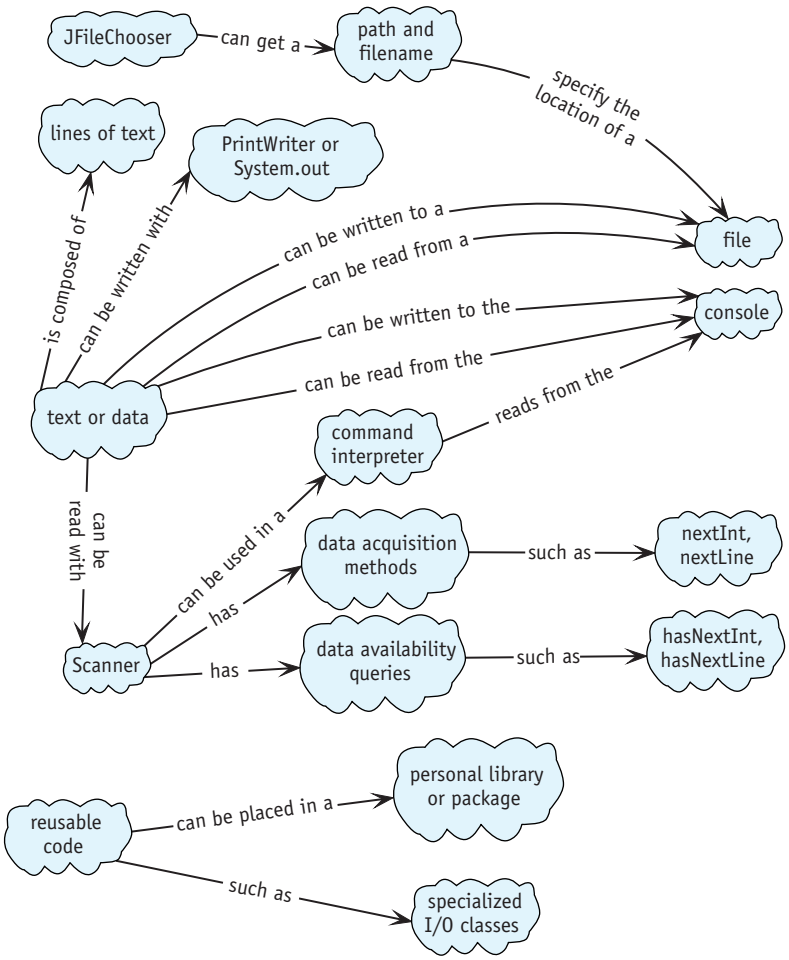
Related Pattern: The Error-Checked Input pattern is an important part of making the command interpreter respond appropriately to user errors.

9.10 Summary and Concept Map

Programs may read information from either the user (via the console) or a file. Reading is typically done with an instance of `Scanner`, which has a variety of methods to acquire data and convert it to an appropriate type. Another set of methods can determine if data of a specified type is available.

Programs may also write information to either the console, using `System.out`, or a file, using `PrintWriter`. The location of a file is specified by a path and filename. Paths may be either relative to the current working directory or absolute.

It is usually appropriate to represent a record in a file with a class, delegating the input and output to methods in the class. If several programs use the same file, it is appropriate to put the class into a package to make reuse easy.



Problem Set

Written Exercises

9.1 Review Listing 9-4. Explain why the `else`-clause in lines 13–15 could be removed with no change in the function of the program.

9.2 Write the Construct Record from File pattern. See Section 9.2.1 for background and examples.

9.3 Describe how to modify the command interpreter in Listing 9-11 to allow the user to enter either a word or a single character for each command. For example, to set the minimum file size, the user can enter `m`, `M`, `min`, or even `MiN` followed by the desired size.

9.4 Consider writing a simple telephone book program, which will keep names and telephone numbers of friends and businesses you call frequently in a file. The program itself should repeatedly prompt for a name, displaying all of the matching names and their associated telephone numbers.

a. Develop two different file formats for your program. Describe both, indicating which one you consider the better design and why.

b. Write pseudocode for the program. Include prompting, opening and closing files, and the search algorithm.

c. Suppose the specification is modified so that the program prints only the first name it finds, if it finds one at all. Consider the following pseudocode and describe the bug(s) in each algorithm.

```
while (true)
{ read the next record
  if (at the end of the file)
  { print message
    exit the loop
  }
  if (found the name)
  { print message
    exit the loop
  }
}
```

```
while (not at end of file)
{ read the next record
  if (found the name)
  { print message
    exit the loop
  }
}
if (at end of file)
{ print not found message
}
```

Programming Exercises

9.5 The package `becker.xtras.hangman` includes classes implementing the game of Hangman. Figure 7-13 has an image of the user interface. Extend `SampleHangman` and override the `newGame()` method to open a file, choose a random phrase, and then call the other `newGame` method with the chosen phrase. Create a file with the phrases. Create a `main` method, as shown in the package overview, to run your program.

9.6 Write a program that reads a text file and writes it to a new file, performing one of the following transformations. (*Note:* Other than the described transformation, the two files should be identical.)

a. Write the new file entirely in uppercase letters.

b. Double space the new file.

c. Make an identical copy of the file except for a statement at the end telling how many characters, words (tokens), and lines it contains. For the purpose of counting characters, ignore newline characters.

d. Reverse the characters in each line of the file. If the first line of input is "It was a dark and stormy night." the first line of output should be ".thgin ymrots dna krad a saw tI".

e. Put a prefix such as > at the beginning of each line of output.

f. Output only those lines that contain a given string.

g. Output the first *n* lines of the input file.

Programming Projects

9.7 Write a calculator that accepts input like the one shown in Figure 9-15. The program has a variable that stores the calculation as performed so far. When a line begins with a number, that number goes in to the variable. An operation such as + or * is remembered so that the next number can be combined appropriately with the number currently stored in the variable. An equal sign causes the current value of the variable to be printed. A line that starts with an operator such as / continues to use the number in the variable from previous operations. In addition to the operators shown, implement subtraction.

(figure 9-15)

Calculator program

9.8 Write your own version of the `Prompt` class (see Listing 9-9).

a. Implement the `forInt`, `forInputFile`, and `forInputScanner` shown in Listing 9-9.

b. Implement `forBoolean`, `forDouble`, and `forToken`. Each take a prompt as a parameter and return a `boolean`, `double`, or single `String` token, respectively.

c. Implement `forInt` with three parameters: a prompt, a minimum value, and a maximum value. The method returns an integer value between the minimum and the maximum, inclusive. The method invocation `forInt("Enter your choice", 1, 5)` should produce the prompt "`Enter your choice [1..5]: `". Entering text that is outside of this range or is not an integer should produce a prompt explaining the error.

d. Implement `forInt` with two parameters: a prompt and a default value. The method returns the integer value entered by the user, or the default value if the user only hits Enter. Of course, if the user enters something else, an appropriate error message is displayed and the user is given another opportunity. The method invocation `forInt("Enter your choice", 0)` should produce the prompt "`Enter your choice (0): `". (*Hint*: To detect only Enter, you will need to read the response using `nextLine`,

removing leading and trailing blanks with the `trim` method in `String`. If the result is not empty, you will need to determine if it contains an integer. Check the constructors in `Scanner` for ideas.)

e. Implement `forString` with two parameters: a prompt and a list of valid values. The method returns the entered string, but only if the response appears in the list of valid values. If it does not, print the list of valid values and ask the user to try again. (*Hint*: The list of valid values could be either a string with appropriate delimiters—such as "`|exit|stop|go|`"—or one of the collection classes discussed in Section 8.5.)

f. Put the `Prompt` class in an appropriately named package for easy reuse. In a different directory, create a simple program that uses at least one of the methods from your package. Run it.

9.9 Write a class named `HangmanTextUI` that can be used to play a text-based version of Hangman. Your program will have a similar structure to the `LogExplorer` program described in Section 9.5. You may use the `SampleHangman` class in `becker.xtras.hangman` to implement the actual game.

9.10 Complete the reorganization of the model and user interface for the `LogExplorer` program as described in Section 9.5.3.

a. Complete the class diagram shown in Figure 9-8.

b. If necessary, download the files for `/ch09/logExplorer2/` and complete the program it contains. It includes the original `LogExplorer` class plus the `LogExplorerView` Java interface and a graphical user interface. The `main` method will ask you which interface you would like to use.

Modifying the program as described will allow you to choose between the command interpreter and the graphical user interface when the program runs. The graphical user interface does not require any changes as long as the methods named `setSearchHost` and `setMinSize` are provided in `LogExplorer`.

9.11 Implement all the parts of Programming Exercise 9.6, providing a command interpreter to specify which transformation should be performed. Commands to the command interpreter should be of the following form:

```
reverse <in> [<out>]
prefix <str> <in> [<out>]
first <int> <in> [<out>]
```

`<in>` is the name of an input file. `<out>` is the name of an output file. Placing `<out>` in square brackets means that entering an output file is optional, in which case output is displayed on the screen. `<str>` and `<int>` indicate that a string or an integer is expected, respectively.

9.12 Write a program to display a slide show on the computer's display by combining the programs in Section 9.8. Modify the `ImageComponent` class shown in Listing 9-14 to include the method `setImage(String fileName)`. When

called, the component should read the image from the named file and display it. Recall the function of `repaint`, as discussed in Section 6.7.2. Set the preferred size of the component to 500 x 500 instead of basing it on a specific image.

a. Use `JFileChooser` to obtain a file named by the user. Each record in the file will contain an image filename and the number of seconds to display the image. The image files are assumed to be in the same directory as the file listing them (which may be different from the program's working directory).

b. Use `JFileChooser` to obtain the list of images to show. `JFileChooser` can allow the user to select several files at the same time by holding down the Shift or Control keys while selecting files. To enable this behavior, the programmer must call the chooser's `setMultiSelectionEnabled` method before the chooser is shown. The list of files chosen can be retrieved with the following statement:

```
List<File> fNames = Arrays.asList(
                                    chooser.getSelectedFiles());
```

Use a *foreach* loop, as discussed in Section 8.5.1, to access each file. Use `Thread.sleep` to pause for two seconds between each image.

9.13 A cipher transforms text to conceal its meaning. One of the simplest ciphers is the Caesar cipher, which replaces each letter in the message with the letter *n* positions away in the alphabet, where *n* remains constant. For example, when *n* = 3, `A` is replaced with `D`, `B` with `E`, `C` with `F`, and so on. Letters at the end of the alphabet will wrap around to be replaced with letters at the beginning of the alphabet. See Figure 9-16.

(figure 9-16)

Caesar cipher

Using *n* = 3 to encode the message `MEET AT DAWN` results in the encoded message `PHHW DW GDZQ`. One can do this in a program by placing the letters in a string. For each letter in the message, find its position, *p*, in the string and write the letter at (*p* + *n*) % 26 to the output file. Any character not in the string is written as itself. For example, period (`.`) will appear identically in both the input and the output.

a. Write a program implementing the Caesar cipher described above except that the string includes all the letters present on your keyboard. Your program should ask the user for an offset, an input file, and an output file. When your program is complete, encode a message with *n* = 5. You should be able to decode the message by running the program again with *n* = -5. Be careful with the `%` operator, however. If the first operand is negative, the answer will be negative as well.

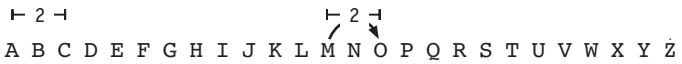
b. The Caesar cipher is easy to break because there are so few combinations to try. One could easily write a program to simply try each value of *n*. A better approach asks the user for a key, for example `SMOKESTACK`, and a value of *n*. Insert the letters in the key into a string, ignoring duplicates. Then add all the remaining characters to encode in order, skipping any that are already present. For example, with the key `SMOKESTACK`, one would have the following string: `SMOKETACBDFGHIJLNPQRUVWXYZ`. Now encode the message as with the Caesar cipher. Of course, this is more effective if the string contains lowercase letters, digits, punctuation, and so on. It is also more effective if the key contains some of these characters as well. Write a program implementing this encoding scheme. Verify that you can use the same program to decode the message.

c. The keyed cipher in part (b) is still relatively easy to break using letter frequencies. Assuming the coded message is written in English, the characters that appear most often in the coded message are likely to be the most frequently occurring English letters: `E`, `T`, `A`, `R`, and `N`.

   One way around this is use a key again. Suppose the key is `CIPHER` and the message to encode is `MEET AT DAWN`. The first letter of the message, `M`, is encoded as `O` using an offset of 2 because there are two letters between the beginning of the alphabet and `C`, the first letter of the key. See Figure 9-17.

(figure 9-17)

First step of using a key

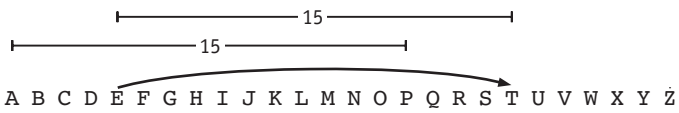


Similarly, the second letter of the message, `E`, is encoded using the second letter of the key to determine the offset. The offset is 8 because there are 8 letters between the beginning of the alphabet and `I`. Therefore `E` is encoded as `M`. See Figure 9-18.

(figure 9-18)

Second step of using a key

Notice, however, that the second `E` in `MEET` is encoded differently than the first one. That's because the third letter of the key is `P`. Therefore, an offset of 15 is used instead of an offset of `8`, encoding `E` as `T`. See Figure 9-19.



(figure 9-19)

Using an offset of 15 to encode `E` *as* `T`

When the end of the key is reached, simply wrap around to the beginning to encode the next letter of the message. Implement this improved algorithm in the class `Cipher3.java`.