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.

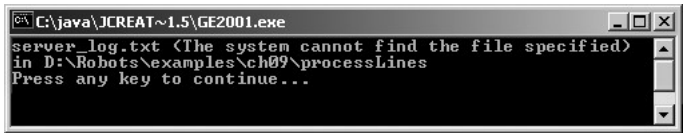
Second, it is wise to handle the exception by giving the user as much information as possible about where the program was looking for the missing file. This is done by getting and printing the **working directory** in line 19. The working directory is the directory (directories are also called folders) 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.

KEY IDEA

Every character, even spaces, corresponds to a 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.

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.

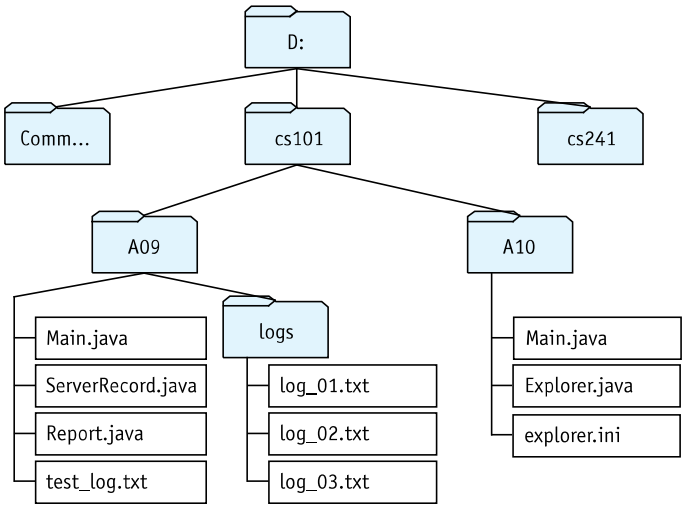
9.3.1 Filenames

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.

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.

(table 9-4) *continued*

Summary of methods in the `File` *class*

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.