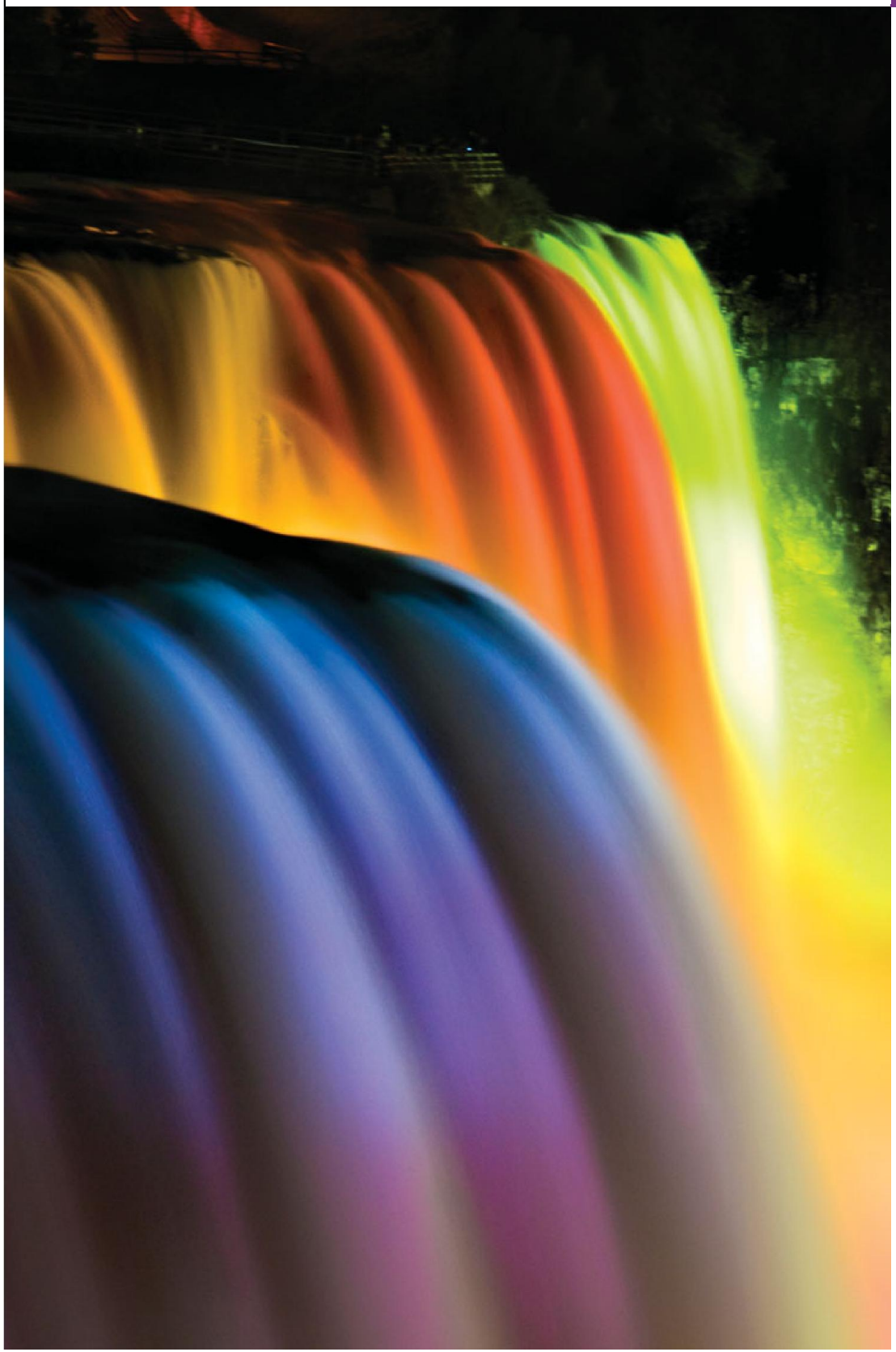
# 17

# Files, Streams and Object Serialization



*I can only assume that a "Do Not File" document is filed in a "Do Not File" file.*
—Senator Frank Church
Senate Intelligence Subcommittee Hearing, 1975

*Consciousness … does not appear to itself chopped up in bits. … A "river" or a "stream" are the metaphors by which it is most naturally described.*
—William James

## Objectives

In this chapter you'll learn:

- To create, read, write and update files.
- To retrieve information about files and directories.
- The Java input/output stream class hierarchy.
- The differences between text files and binary files.
- To use classes `Scanner` and `Formatter` to process text files.
- To use classes `FileInputStream` and `FileOutputStream` to read from and write to files.
- To use classes `ObjectInputStream` and `ObjectOutputStream` to read objects from and write objects to files.
- To use a `JFileChooser` dialog.

Outline


**17.1** Introduction
**17.2** Files and Streams
**17.3** Class `File`
**17.4** Sequential-Access Text Files
- 17.4.1 Creating a Sequential-Access Text File
- 17.4.2 Reading Data from a Sequential-Access Text File
- 17.4.3 Case Study: A Credit-Inquiry Program
- 17.4.4 Updating Sequential-Access Files

**17.5** Object Serialization
- 17.5.1 Creating a Sequential-Access File Using Object Serialization
- 17.5.2 Reading and Deserializing Data from a Sequential-Access File

**17.6** Additional `java.io` Classes
- 17.6.1 Interfaces and Classes for Byte-Based Input and Output
- 17.6.2 Interfaces and Classes for Character-Based Input and Output

**17.8** Opening Files with `JFileChooser`
**17.8** Wrap-Up

*Summary* | *Self-Review Exercises* | *Answers to Self-Review Exercises* | *Exercises* | *Making a Difference*


## 17.1 Introduction[1]

Data stored in variables and arrays is temporary—it's lost when a local variable goes out of scope or when the program terminates. For long-term retention of data, even after the programs that create the data terminate, computers use **files**. You use files every day for tasks such as writing a document or creating a spreadsheet. Computers store files on **secondary storage devices** such as hard disks, optical disks, flash drives and magnetic tapes. Data maintained in files is **persistent data**—it exists beyond the duration of program execution. In this chapter, we explain how Java programs create, update and process files.

We begin with a discussion of Java's architecture for handling files programmatically. Next we explain that data can be stored in text files and binary files—and we cover the differences between them. We demonstrate retrieving information about files and directories using class `File`, then devote several sections to the different mechanisms for writing data to and reading data from files. We show how to create and manipulate sequential-access text files. Working with text files allows you to quickly and easily start manipulating files. As you'll learn, however, it's difficult to read data from text files back into object form. Fortunately, many object-oriented languages (including Java) provide ways to write objects to and read objects from files (known as object serialization and deserialization). To demonstrate this, we recreate some of our sequential-access programs that used text files, this time by storing objects in binary files.

## 17.2 Files and Streams

Java views each file as a sequential **stream of bytes** (Fig. 17.1). Every operating system provides a mechanism to determine the end of a file, such as an **end-of-file marker** or a count of the total bytes in the file that's recorded in a system-maintained administrative data structure. A Java program processing a stream of bytes simply receives an indication from the operating system when it reaches the end of the stream—the program does *not* need to know how the underlying platform represents files or streams. In some cases, the end-

1. The techniques shown in this chapter are based on Java SE 6. Java SE 7 introduces new file-system APIs for interacting with files and directories. On the book's Companion Website (accessible via `www.pearsonhighered.com/deitel`) we've posted a version of this chapter implemented using these Java SE 7 APIs.

of-file indication occurs as an exception. In other cases, the indication is a return value from a method invoked on a stream-processing object.

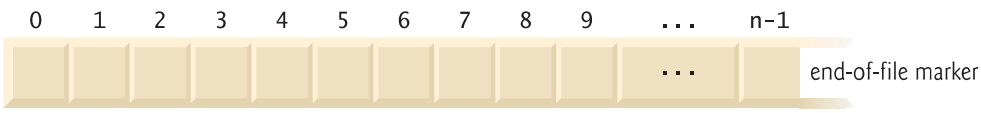



**Fig. 17.1** | Java's view of a file of *n* bytes.

### *Byte-Based and Character-Based Streams*

File streams can be used to input and output data as bytes or characters. **Byte-based streams** input and output data in its binary format. **Character-based streams** input and output data as a sequence of characters. If the value `5` were being stored using a byte-based stream, it would be stored in the binary format of the numeric value `5`, or `101`. If the value `5` were being stored using a character-based stream, it would be stored in the binary format of the character `5`, or `00000000 00110101` (this is the binary representation for the numeric value `53`, which indicates the Unicode® character `5`). The difference between the two forms is that the numeric value can be used as an integer in calculations, whereas the character `5` is simply a character that can be used in a string of text, as in `"Sarah Miller is 15 years old"`. Files that are created using byte-based streams are referred to as **binary files**, while files created using character-based streams are referred to as **text files**. Text files can be read by text editors, while binary files are read by programs that understand the file's specific content and its ordering.

### *Standard Input, Standard Output and Standard Error Streams*

A Java program **opens** a file by creating an object and associating a stream of bytes or characters with it. The object's constructor interacts with the operating system to open the file. Java can also associate streams with different devices. When a Java program begins executing, in fact, it creates three stream objects that are associated with devices—`System.in`, `System.out` and `System.err`. `System.in` (the standard input stream object) normally enables a program to input bytes from the keyboard; object `System.out` (the standard output stream object) normally enables a program to output character data to the screen; and object `System.err` (the standard error stream object) normally enables a program to output character-based error messages to the screen. Each stream can be redirected. For `System.in`, this capability enables the program to read bytes from a different source. For `System.out` and `System.err`, it enables the output to be sent to a different location, such as a file on disk. Class `System` provides methods **`setIn`**, **`setOut`** and **`setErr`** to **redirect** the standard input, output and error streams, respectively.

### *The `java.io` Package*

Java programs perform file processing by using classes from package **`java.io`**. This package includes definitions for stream classes, such as **`FileInputStream`** (for byte-based input from a file), **`FileOutputStream`** (for byte-based output to a file), **`FileReader`** (for character-based input from a file) and **`FileWriter`** (for character-based output to a file), which inherit from classes `InputStream`, `OutputStream`, `Reader` and `Writer`, respectively. Thus, the methods of the these stream classes can also be applied to file streams.

Java contains classes that enable you to perform input and output of objects or variables of primitive data types. The data will still be stored as bytes or characters behind the scenes, allowing you to read or write data in the form of `int`s, `String`s, or other types without having to worry about the details of converting such values to byte format. To perform such input and output, objects of classes **`ObjectInputStream`** and **`ObjectOutputStream`** can be used together with the byte-based file stream classes `FileInputStream` and `FileOutputStream` (these classes will be discussed in more detail shortly). The complete hierarchy of types in package `java.io` can be viewed in the online documentation at

```
download.oracle.com/javase/6/docs/api/java/io/package-tree.html
```

As you can see in the hierarchy, Java offers many classes for performing input/output operations. We use several of these classes in this chapter to implement file-processing programs that create and manipulate sequential-access files. In Chapter 27, we use stream classes extensively to implement networking applications.

In addition to the `java.io` classes, character-based input and output can be performed with classes `Scanner` and **`Formatter`**. Class `Scanner` is used extensively to input data from the keyboard—it can also read data from a file. Class `Formatter` enables formatted data to be output to any text-based stream in a manner similar to method `System.out.printf`. Appendix G presents the details of formatted output with `printf`. All these features can be used to format text files as well.

## 17.3 Class `File`

This section presents class **`File`**, which is useful for retrieving information about files or directories from disk. Objects of class `File` do not open files or provide any file-processing capabilities. However, `File` objects are used frequently with objects of other `java.io` classes to specify files or directories to manipulate.

### *Creating `File` Objects*

Class `File` provides four constructors. The one with a `String` argument specifies the `name` of a file or directory to associate with the `File` object. The `name` can contain **path information** as well as a file or directory name. A file or directory's path specifies its location on disk. The path includes some or all of the directories leading to the file or directory. An **absolute path** contains all the directories, starting with the **root directory**, that lead to a specific file or directory. Every file or directory on a particular disk drive has the same root directory in its path. A **relative path** normally starts from the directory in which the application began executing and is therefore "relative" to the current directory. The constructor with two `String` arguments specifies an absolute or relative path as the first argument and the file or directory to associate with the `File` object as the second argument. The constructor with `File` and `String` arguments uses an existing `File` object that specifies the parent directory of the file or directory specified by the `String` argument. The fourth constructor uses a `URI` object to locate the file. A **Uniform Resource Identifier (URI)** is a more general form of the **Uniform Resource Locators (URLs)** that are used to locate websites. For example, `http://www.deitel.com/` is the URL for the Deitel & Associates website. URIs for locating files vary across operating systems. On Windows platforms, the URI

```
file://C:/data.txt
```

identifies the file data.txt stored in the root directory of the C: drive. On UNIX/Linux platforms, the URI

```
file:/home/student/data.txt
```

identifies the file data.txt stored in the home directory of the user student.

Figure 17.2 lists some common File methods. The complete list can be viewed at download.oracle.com/javase/6/docs/api/java/io/File.html.

| Method | Description |
|---|---|
| boolean canRead() | Returns true if a file is readable by the current application; false otherwise. |
| boolean canWrite() | Returns true if a file is writable by the current application; false otherwise. |
| boolean exists() | Returns true if the file or directory represented by the File object exists; false otherwise. |
| boolean isFile() | Returns true if the name specified as the argument to the File constructor is a file; false otherwise. |
| boolean isDirectory() | Returns true if the name specified as the argument to the File constructor is a directory; false otherwise. |
| boolean isAbsolute() | Returns true if the arguments specified to the File constructor indicate an absolute path to a file or directory; false otherwise. |
| String getAbsolutePath() | Returns a String with the absolute path of the file or directory. |
| String getName() | Returns a String with the name of the file or directory. |
| String getPath() | Returns a String with the path of the file or directory. |
| String getParent() | Returns a String with the parent directory of the file or directory (i.e., the directory in which the file or directory is located). |
| long length() | Returns the length of the file, in bytes. If the File object represents a directory, an unspecified value is returned. |
| long lastModified() | Returns a platform-dependent representation of the time at which the file or directory was last modified. The value returned is useful only for comparison with other values returned by this method. |
| String[] list() | Returns an array of Strings representing a directory's contents. Returns null if the File object does not represent a directory. |

**Fig. 17.2** | File methods.

### *Demonstrating Class File*

Figure 17.3 prompts the user to enter the name of a file or directory, then uses class File to output information about the file or directory.

The program begins by prompting the user for a file or directory (line 12). Line 13 inputs the file name or directory name and passes it to method analyzePath (lines 17–50). The method creates a new File object (line 20) and assigns its reference to name. Line 22 invokes File method exists to determine whether the name input by the user exists

```
1  // Fig. 17.3: FileDemonstration.java
2  // File class used to obtain file and directory information.
3  import java.io.File;
4  import java.util.Scanner;
5
6  public class FileDemonstration
7  {
8     public static void main( String[] args )
9     {
10       Scanner input = new Scanner( System.in );
11
12       System.out.print( "Enter file or directory name: " );
13       analyzePath( input.nextLine() );
14    } // end main
15
16    // display information about file user specifies
17    public static void analyzePath( String path )
18    {
19       // create File object based on user input
20       File name = new File( path );
21
22       if ( name.exists() ) // if name exists, output information about it
23       {
24          // display file (or directory) information
25          System.out.printf(
26             "%s%s\n%s\n%s\n%s\n%s%s\n%s%s\n%s%s\n%s%s\n%s%s",
27             name.getName(), " exists",
28             ( name.isFile() ? "is a file" : "is not a file" ),
29             ( name.isDirectory() ? "is a directory" :
30                "is not a directory" ),
31             ( name.isAbsolute() ? "is absolute path" :
32                "is not absolute path" ), "Last modified: ",
33             name.lastModified(), "Length: ", name.length(),
34             "Path: ", name.getPath(), "Absolute path: ",
35             name.getAbsolutePath(), "Parent: ", name.getParent() );
36
37          if ( name.isDirectory() ) // output directory listing
38          {
39             String[] directory = name.list();
40             System.out.println( "\n\nDirectory contents:\n" );
41
42             for ( String directoryName : directory )
43                System.out.println( directoryName );
44          } // end if
45       } // end outer if
46       else // not file or directory, output error message
47       {
48          System.out.printf( "%s %s", path, "does not exist." );
49       } // end else
50    } // end method analyzePath
51 } // end class FileDemonstration
```

**Fig. 17.3** | `File` class used to obtain file and directory information. (Part 1 of 2.)

```
Enter file or directory name: E:\Program Files\Java\jdk1.6.0_11\demo\jfc
jfc exists
is not a file
is a directory
is absolute path
Last modified: 1228404395024
Length: 4096
Path: E:\Program Files\Java\jdk1.6.0_11\demo\jfc
Absolute path: E:\Program Files\Java\jdk1.6.0_11\demo\jfc
Parent: E:\Program Files\Java\jdk1.6.0_11\demo

Directory contents:

CodePointIM
FileChooserDemo
Font2DTest
Java2D
Laffy
Metalworks
Notepad
SampleTree
Stylepad
SwingApplet
SwingSet2
SwingSet3
```

```
Enter file or directory name: C:\Program Files\Java\jdk1.6.0_11\demo\jfc
\Java2D\README.txt
README.txt exists
is a file
is not a directory
is absolute path
Last modified: 1228404384270
Length: 7518
Path: E:\Program Files\Java\jdk1.6.0_11\demo\jfc\Java2D\README.txt
Absolute path: E:\Program Files\Java\jdk1.6.0_11\demo\jfc\Java2D\README.txt
Parent: E:\Program Files\Java\jdk1.6.0_11\demo\jfc\Java2D
```

**Fig. 17.3** | `File` class used to obtain file and directory information. (Part 2 of 2.)

(either as a file or as a directory) on the disk. If the name does not exist, control proceeds to lines 46–49 and displays a message to the screen containing the name the user typed, followed by "`does not exist`." Otherwise, the `if` statement (lines 22–45) executes. The program outputs the name of the file or directory (line 27), followed by the results of testing the `File` object with `isFile` (line 28), `isDirectory` (line 29) and `isAbsolute` (line 31). Next, the program displays the values returned by `lastModified` (line 33), `length` (line 33), `getPath` (line 34), `getAbsolutePath` (line 35) and `getParent` (line 35). If the `File` object represents a directory (line 37), the program obtains a list of the directory's contents as an array of `String`s by using `File` method `list` (line 39) and displays the list on the screen.

The first output of this program demonstrates a `File` object associated with the `jfc` directory from the JDK. The second output demonstrates a `File` object associated with

the `README.txt` file from the Java 2D example that comes with the JDK. In both cases, we specified an absolute path on our computer.

A **separator character** is used to separate directories and files in the path. On a Windows computer, the separator character is a backslash (\). On a UNIX system, it's a forward slash (/). Java processes both characters identically in a path name. For example, if we were to use the path

```
c:\Program Files\Java\jdk1.6.0_11\demo/jfc
```

which employs each separator character, Java would still process the path properly. When building `String`s that represent path information, use `File.separator` to obtain the local computer's proper separator character rather than explicitly using `/` or `\`. This constant returns a `String` consisting of one character—the proper separator for the system.

**Common Programming Error 17.1**

*Using `\` as a directory separator rather than `\\` in a string literal is a logic error. A single `\` indicates that the `\` followed by the next character represents an escape sequence. Use `\\` to insert a `\` in a string literal.*

## 17.4 Sequential-Access Text Files

Next, we create and manipulate sequential-access files in which records are stored in order by the record-key field. We begin with text files, enabling the reader to quickly create and edit human-readable files. We discuss creating, writing data to, reading data from and updating sequential-access text files. We also include a credit-inquiry program that retrieves specific data from a file.

### 17.4.1 Creating a Sequential-Access Text File

*Java imposes no structure on a file—notions such as records do not exist as part of the Java language.* Therefore, you must structure files to meet the requirements of your applications. In the following example, we see how to impose a keyed record structure on a file.

The program in Figs. 17.4, 17.5 and 17.8 creates a simple sequential-access file that might be used in an accounts receivable system to keep track of the amounts owed to a company by its credit clients. For each client, the program obtains from the user an account number and the client's name and balance (i.e., the amount the client owes the company for goods and services received). Each client's data constitutes a "record" for that client. This application uses the account number as the record key—the file will be created and maintained in account-number order. The program assumes that the user enters the records in account-number order. In a comprehensive accounts receivable system (based on sequential-access files), a sorting capability would be provided so that the user could enter the records in any order. The records would then be sorted and written to the file.

#### *Class* `AccountRecord`

Class `AccountRecord` (Fig. 17.4) encapsulates the client record information used by the examples in this chapter. `AccountRecord` is declared in package `com.deitel.ch17` (line 3), so that it can be imported into several of this chapter's examples for reuse. (Section 8.14 provides information on compiling and using your own packages.) Class `AccountRecord` contains `private` instance variables `account`, `firstName`, `lastName` and `balance` (lines 7–

10) and *set* and *get* methods for accessing these fields. Though the *set* methods do not validate the data in this example, they should do so in an "industrial-strength" system.

```
1  // Fig. 17.4: AccountRecord.java
2  // AccountRecord class maintains information for one account.
3  package com.deitel.ch17; // packaged for reuse
4
5  public class AccountRecord
6  {
7     private int account;
8     private String firstName;
9     private String lastName;
10    private double balance;
11
12    // no-argument constructor calls other constructor with default values
13    public AccountRecord()
14    {
15       this( 0, "", "", 0.0 ); // call four-argument constructor
16    } // end no-argument AccountRecord constructor
17
18    // initialize a record
19    public AccountRecord( int acct, String first, String last, double bal )
20    {
21       setAccount( acct );
22       setFirstName( first );
23       setLastName( last );
24       setBalance( bal );
25    } // end four-argument AccountRecord constructor
26
27    // set account number
28    public void setAccount( int acct )
29    {
30       account = acct;
31    } // end method setAccount
32
33    // get account number
34    public int getAccount()
35    {
36       return account;
37    } // end method getAccount
38
39    // set first name
40    public void setFirstName( String first )
41    {
42       firstName = first;
43    } // end method setFirstName
44
45    // get first name
46    public String getFirstName()
47    {
48       return firstName;
49    } // end method getFirstName
```

**Fig. 17.4** | `AccountRecord` class maintains information for one account. (Part 1 of 2.)

```
50
51      // set last name
52      public void setLastName( String last )
53      {
54         lastName = last;
55      } // end method setLastName
56
57      // get last name
58      public String getLastName()
59      {
60         return lastName;
61      } // end method getLastName
62
63      // set balance
64      public void setBalance( double bal )
65      {
66         balance = bal;
67      } // end method setBalance
68
69      // get balance
70      public double getBalance()
71      {
72         return balance;
73      } // end method getBalance
74   } // end class AccountRecord
```

**Fig. 17.4** | `AccountRecord` class maintains information for one account. (Part 2 of 2.)

To compile class `AccountRecord`, open a command window, change directories to this chapter's `fig17_05` directory (which contains `AccountRecord.java`), then type:

```
javac -d .. AccountRecord.java
```

This places `AccountRecord.class` in its package directory structure and places the package in the `ch17` folder that contains all the examples for this chapter. When you compile class `AccountRecord` (or any other classes that will be reused in this chapter), you should place them in a common directory. When you compile or execute classes that use class `AccountRecord` (e.g., `CreateTextFile` in Fig. 17.5), you must specify the command-line argument `-classpath` to both `javac` and `java`, as in

```
javac -classpath .;c:\examples\ch17 CreateTextFile.java
java -classpath .;c:\examples\ch17 CreateTextFile
```

The current directory (specified with `.`) is included in the classpath to ensure that the compiler can locate other classes in the same directory as the class being compiled. The path separator used in the preceding commands must be appropriate for your platform—a semicolon (`;`) on Windows and a colon (`:`) on UNIX/Linux/Mac OS X. The preceding commands assume that the package containing `AccountRecord` is located at in the directory `C:\examples\ch17` on a Windows computer.

### *Class* `CreateTextFile`

Now let's examine class `CreateTextFile` (Fig. 17.5). Line 14 declares `Formatter` variable `output`. As discussed in Section 17.2, a `Formatter` object outputs formatted `Strings`, us-

ing the same formatting capabilities as method `System.out.printf`. A `Formatter` object can output to various locations, such as the screen or a file, as is done here. The `Formatter` object is instantiated in line 21 in method `openFile` (lines 17–34). The constructor used in line 21 takes one argument—a `String` containing the name of the file, including its path. If a path is not specified, as is the case here, the JVM assumes that the file is in the directory from which the program was executed. For text files, we use the `.txt` file extension. If the file does not exist, it will be created. If an existing file is opened, its contents are **truncated**—all the data in the file is discarded. At this point the file is open for writing, and the resulting `Formatter` object can be used to write data to the file.

```
1  // Fig. 17.5: CreateTextFile.java
2  // Writing data to a sequential text file with class Formatter.
3  import java.io.FileNotFoundException;
4  import java.lang.SecurityException;
5  import java.util.Formatter;
6  import java.util.FormatterClosedException;
7  import java.util.NoSuchElementException;
8  import java.util.Scanner;
9
10 import com.deitel.ch17.AccountRecord;
11
12 public class CreateTextFile
13 {
14    private Formatter output; // object used to output text to file
15
16    // enable user to open file
17    public void openFile()
18    {
19       try
20       {
21          output = new Formatter( "clients.txt" ); // open the file
22       } // end try
23       catch ( SecurityException securityException )
24       {
25          System.err.println(
26             "You do not have write access to this file." );
27          System.exit( 1 ); // terminate the program
28       } // end catch
29       catch ( FileNotFoundException fileNotFoundException )
30       {
31          System.err.println( "Error opening or creating file." );
32          System.exit( 1 ); // terminate the program
33       } // end catch
34    } // end method openFile
35
36    // add records to file
37    public void addRecords()
38    {
39       // object to be written to file
40       AccountRecord record = new AccountRecord();
```

**Fig. 17.5** | Writing data to a sequential text file with class `Formatter`. (Part 1 of 3.)

```
41
42          Scanner input = new Scanner( System.in );
43
44          System.out.printf( "%s\n%s\n%s\n%s\n\n",
45             "To terminate input, type the end-of-file indicator ",
46             "when you are prompted to enter input.",
47             "On UNIX/Linux/Mac OS X type <ctrl> d then press Enter",
48             "On Windows type <ctrl> z then press Enter" );
49
50          System.out.printf( "%s\n%s",
51             "Enter account number (> 0), first name, last name and balance.",
52             "? " );
53
54          while ( input.hasNext() ) // loop until end-of-file indicator
55          {
56             try // output values to file
57             {
58                // retrieve data to be output
59                record.setAccount( input.nextInt() ); // read account number
60                record.setFirstName( input.next() ); // read first name
61                record.setLastName( input.next() ); // read last name
62                record.setBalance( input.nextDouble() ); // read balance
63
64                if ( record.getAccount() > 0 )
65                {
66                   // write new record
67                   output.format( "%d %s %s %.2f\n", record.getAccount(),
68                      record.getFirstName(), record.getLastName(),
69                      record.getBalance() );
70                } // end if
71                else
72                {
73                   System.out.println(
74                      "Account number must be greater than 0." );
75                } // end else
76             } // end try
77             catch ( FormatterClosedException formatterClosedException )
78             {
79                System.err.println( "Error writing to file." );
80                return;
81             } // end catch
82             catch ( NoSuchElementException elementException )
83             {
84                System.err.println( "Invalid input. Please try again." );
85                input.nextLine(); // discard input so user can try again
86             } // end catch
87
88             System.out.printf( "%s %s\n%s", "Enter account number (>0),",
89                "first name, last name and balance.", "? " );
90          } // end while
91       } // end method addRecords
92
```

**Fig. 17.5** | Writing data to a sequential text file with class `Formatter`. (Part 2 of 3.)

```
93      // close file
94      public void closeFile()
95      {
96         if ( output != null )
97            output.close();
98      } // end method closeFile
99   } // end class CreateTextFile
```

**Fig. 17.5** | Writing data to a sequential text file with class `Formatter`. (Part 3 of 3.)

Lines 23–28 handle the **`SecurityException`**, which occurs if the user does not have permission to write data to the file. Lines 29–33 handle the **`FileNotFoundException`**, which occurs if the file does not exist and a new file cannot be created. This exception may also occur if there's an error opening the file. In both exception handlers we call `static` method **`System.exit`** and pass the value `1`. This method terminates the application. An argument of `0` to method `exit` indicates successful program termination. A nonzero value, such as `1` in this example, normally indicates that an error has occurred. This value is passed to the command window that executed the program. The argument is useful if the program is executed from a **batch file** on Windows systems or a **shell script** on UNIX/Linux/Mac OS X systems. Batch files and shell scripts offer a convenient way of executing several programs in sequence. When the first program ends, the next program begins execution. It's possible to use the argument to method `exit` in a batch file or shell script to determine whether other programs should execute. For more information on batch files or shell scripts, see your operating system's documentation.

Method `addRecords` (lines 37–91) prompts the user to enter the various fields for each record or to enter the end-of-file key sequence when data entry is complete. Figure 17.6 lists the key combinations for entering end-of-file for various computer systems.

| Operating system | Key combination |
|---|---|
| UNIX/Linux/Mac OS X | *<Enter> <Ctrl> d* |
| Windows | *<Ctrl> z* |

**Fig. 17.6** | End-of-file key combinations.

Line 40 creates an `AccountRecord` object, which will be used to store the values of the current record entered by the user. Line 42 creates a `Scanner` object to read input from the user at the keyboard. Lines 44–48 and 50–52 prompt the user for input.

Line 54 uses `Scanner` method `hasNext` to determine whether the end-of-file key combination has been entered. The loop executes until `hasNext` encounters end-of-file.

Lines 59–62 read data from the user, storing the record information in the `AccountRecord` object. Each statement throws a **`NoSuchElementException`** (handled in lines 82–86) if the data is in the wrong format (e.g., a `String` when an `int` is expected) or if there's no more data to input. If the account number is greater than 0 (line 64), the record's information is written to `clients.txt` (lines 67–69) using method **`format`**, which can perform identical formatting to the `System.out.printf` method used extensively in earlier chapters. Method `format` outputs a formatted `String` to the output destination of

the `Formatter` object—the file `clients.txt`. The format string `"%d %s %s %.2f\n"` indicates that the current record will be stored as an integer (the account number) followed by a `String` (the first name), another `String` (the last name) and a floating-point value (the balance). Each piece of information is separated from the next by a space, and the double value (the balance) is output with two digits to the right of the decimal point (as indicated by the `.2` in `%.2f`). The data in the text file can be viewed with a text editor or retrieved later by a program designed to read the file (Section 17.4.2).

When lines 67–69 execute, if the `Formatter` object is closed, a **`FormatterClosedException`** will be thrown. This exception is handled in lines 77–81. [*Note:* You can also output data to a text file using class **`java.io.PrintWriter`**, which provides `format` and `printf` methods for outputting formatted data.]

Lines 94–98 declare method `closeFile`, which closes the `Formatter` and the underlying output file. Line 97 closes the object by simply calling method **`close`**. If method `close` is not called explicitly, the operating system normally will close the file when program execution terminates—this is an example of operating-system "housekeeping." However, you should always explicitly close a file when it's no longer needed.

### *Platform-Specific Line-Separator Characters*

Lines 67–69 output a line of text followed by a newline (`\n`). If you use a text editor to open the `clients.txt` file produced, each record might not display on a separate line. For example, in Notepad (Microsoft Windows), users will see one continuous line of text. This occurs because different platforms use different line-separator characters. On UNIX/Linux/Mac OS X, the line separator is a newline (`\n`). On Windows, it's a combination of a carriage return and a line feed—represented as `\r\n`. You can use the **`%n`** format specifier in a format control string to output a platform-specific line separator, thus ensuring that the text file can be opened and viewed correctly in a text editor for the platform on which the file was created. The method `System.out.println` outputs a platform-specific line separator after its argument. Also, regardless of the line separator used in a text file, a Java program can still recognize the lines of text and read them.

### *Class* `CreateTextFileTest`

Figure 17.7 runs the program. Line 8 creates a `CreateTextFile` object, which is then used to open, add records to and close the file (lines 10–12). The sample data for this application is shown in Fig. 17.8. In the sample execution for this program, the user enters information for five accounts, then enters end-of-file to signal that data entry is complete. The sample execution does not show how the data records actually appear in the file. In the next section, to verify that the file has been created successfully, we present a program that reads the file and prints its contents. Because this is a text file, you can also verify the information simply by opening the file in a text editor.

```
1  // Fig. 17.7: CreateTextFileTest.java
2  // Testing the CreateTextFile class.
3
4  public class CreateTextFileTest
5  {
```

**Fig. 17.7** | Testing the `CreateTextFile` class. (Part 1 of 2.)

```
6      public static void main( String[] args )
7      {
8         CreateTextFile application = new CreateTextFile();
9
10        application.openFile();
11        application.addRecords();
12        application.closeFile();
13     } // end main
14  } // end class CreateTextFileTest
```

```
To terminate input, type the end-of-file indicator
when you are prompted to enter input.
On UNIX/Linux/Mac OS X type <ctrl> d then press Enter
On Windows type <ctrl> z then press Enter

Enter account number (> 0), first name, last name and balance.
? 100 Bob Jones 24.98
Enter account number (> 0), first name, last name and balance.
? 200 Steve Doe -345.67
Enter account number (> 0), first name, last name and balance.
? 300 Pam White 0.00
Enter account number (> 0), first name, last name and balance.
? 400 Sam Stone -42.16
Enter account number (> 0), first name, last name and balance.
? 500 Sue Rich 224.62
Enter account number (> 0), first name, last name and balance.
? ^Z
```

**Fig. 17.7** | Testing the `CreateTextFile` class. (Part 2 of 2.)

| Sample data | | | |
|---|---|---|---|
| 100 | Bob | Jones | 24.98 |
| 200 | Steve | Doe | -345.67 |
| 300 | Pam | White | 0.00 |
| 400 | Sam | Stone | -42.16 |
| 500 | Sue | Rich | 224.62 |

**Fig. 17.8** | Sample data for the program in Figs. 17.5–17.7.

### 17.4.2 Reading Data from a Sequential-Access Text File

Data is stored in files so that it may be retrieved for processing when needed. Section 17.4.1 demonstrated how to create a file for sequential access. This section shows how to read data sequentially from a text file. We demonstrate how class `Scanner` can be used to input data from a file rather than the keyboard.

The application in Figs. 17.9 and 17.10 reads records from the file `"clients.txt"` created by the application of Section 17.4.1 and displays the record contents. Line 13 of Fig. 17.9 declares a `Scanner` that will be used to retrieve input from the file.

```
1  // Fig. 17.9: ReadTextFile.java
2  // This program reads a text file and displays each record.
3  import java.io.File;
4  import java.io.FileNotFoundException;
5  import java.lang.IllegalStateException;
6  import java.util.NoSuchElementException;
7  import java.util.Scanner;
8
9  import com.deitel.ch17.AccountRecord;
10
11 public class ReadTextFile
12 {
13    private Scanner input;
14
15    // enable user to open file
16    public void openFile()
17    {
18       try
19       {
20          input = new Scanner( new File( "clients.txt" ) );
21       } // end try
22       catch ( FileNotFoundException fileNotFoundException )
23       {
24          System.err.println( "Error opening file." );
25          System.exit( 1 );
26       } // end catch
27    } // end method openFile
28
29    // read record from file
30    public void readRecords()
31    {
32       // object to be written to screen
33       AccountRecord record = new AccountRecord();
34
35       System.out.printf( "%-10s%-12s%-12s%10s\n", "Account",
36          "First Name", "Last Name", "Balance" );
37
38       try // read records from file using Scanner object
39       {
40          while ( input.hasNext() )
41          {
42             record.setAccount( input.nextInt() ); // read account number
43             record.setFirstName( input.next() ); // read first name
44             record.setLastName( input.next() ); // read last name
45             record.setBalance( input.nextDouble() ); // read balance
46
47             // display record contents
48             System.out.printf( "%-10d%-12s%-12s%10.2f\n",
49                record.getAccount(), record.getFirstName(),
50                record.getLastName(), record.getBalance() );
51          } // end while
52       } // end try
```

**Fig. 17.9** | Sequential file reading using a `Scanner`. (Part 1 of 2.)

```
53          catch ( NoSuchElementException elementException )
54          {
55             System.err.println( "File improperly formed." );
56             input.close();
57             System.exit( 1 );
58          } // end catch
59          catch ( IllegalStateException stateException )
60          {
61             System.err.println( "Error reading from file." );
62             System.exit( 1 );
63          } // end catch
64       } // end method readRecords
65
66       // close file and terminate application
67       public void closeFile()
68       {
69          if ( input != null )
70             input.close(); // close file
71       } // end method closeFile
72    } // end class ReadTextFile
```

**Fig. 17.9** | Sequential file reading using a `Scanner`. (Part 2 of 2.)

```
1  // Fig. 17.10: ReadTextFileTest.java
2  // Testing the ReadTextFile class.
3
4  public class ReadTextFileTest
5  {
6     public static void main( String[] args )
7     {
8        ReadTextFile application = new ReadTextFile();
9
10       application.openFile();
11       application.readRecords();
12       application.closeFile();
13    } // end main
14 } // end class ReadTextFileTest
```

```
Account   First Name  Last Name      Balance
100       Bob         Jones            24.98
200       Steve       Doe            -345.67
300       Pam         White             0.00
400       Sam         Stone           -42.16
500       Sue         Rich            224.62
```

**Fig. 17.10** | Testing the `ReadTextFile` class.

Method `openFile` (lines 16–27) opens the file for reading by instantiating a `Scanner` object in line 20. We pass a `File` object to the constructor, which specifies that the `Scanner` object will read from the file `"clients.txt"` located in the directory from which the application executes. If the file cannot be found, a `FileNotFoundException` occurs. The exception is handled in lines 22–26.

Method `readRecords` (lines 30–64) reads and displays records from the file. Line 33 creates `AccountRecord` object `record` to store the current record's information. Lines 35–36 display headers for the columns in the application's output. Lines 40–51 read data from the file until the end-of-file marker is reached (in which case, method `hasNext` will return `false` at line 40). Lines 42–45 use `Scanner` methods `nextInt`, `next` and `nextDouble` to input an `int` (the account number), two `String`s (the first and last names) and a `double` value (the balance). Each record is one line of data in the file. The values are stored in object `record`. If the information in the file is not properly formed (e.g., there's a last name where there should be a balance), a `NoSuchElementException` occurs when the record is input. This exception is handled in lines 53–58. If the `Scanner` was closed before the data was input, an **`IllegalStateException`** occurs (handled in lines 59–63). If no exceptions occur, the record's information is displayed on the screen (lines 48–50). Note in the format string in line 48 that the account number, first name and last name are left justified, while the balance is right justified and output with two digits of precision. Each iteration of the loop inputs one line of text from the text file, which represents one record.

Lines 67–71 define method `closeFile`, which closes the `Scanner`. Method `main` is defined in Fig. 17.10 in lines 6–13. Line 8 creates a `ReadTextFile` object, which is then used to open, add records to and close the file (lines 10–12).

### 17.4.3 Case Study: A Credit-Inquiry Program

To retrieve data sequentially from a file, programs start from the beginning of the file and read all the data consecutively until the desired information is found. It might be necessary to process the file sequentially several times (from the beginning of the file) during the execution of a program. Class `Scanner` does *not* allow repositioning to the beginning of the file. If it's necessary to read the file again, the program must close the file and reopen it.

The program in Figs. 17.11–17.13 allows a credit manager to obtain lists of customers with zero balances (i.e., customers who do not owe any money), customers with credit balances (i.e., customers to whom the company owes money) and customers with debit balances (i.e., customers who owe the company money for goods and services received). A credit balance is a negative amount, a debit balance a positive amount.

#### `MenuOption` *Enumeration*

We begin by creating an `enum` type (Fig. 17.11) to define the different menu options the user will have. The options and their values are listed in lines 7–10. Method `getValue` (lines 19–22) retrieves the value of a specific `enum` constant.

```
1  // Fig. 17.11: MenuOption.java
2  // Enumeration for the credit-inquiry program's options.
3
4  public enum MenuOption
5  {
6     // declare contents of enum type
7     ZERO_BALANCE( 1 ),
8     CREDIT_BALANCE( 2 ),
9     DEBIT_BALANCE( 3 ),
10    END( 4 );
```

**Fig. 17.11** | Enumeration for the credit-inquiry program's menu options. (Part 1 of 2.)

```
11
12     private final int value; // current menu option
13
14     // constructor
15     MenuOption( int valueOption )
16     {
17        value = valueOption;
18     } // end MenuOptions enum constructor
19
20     // return the value of a constant
21     public int getValue()
22     {
23        return value;
24     } // end method getValue
25  } // end enum MenuOption
```

**Fig. 17.11** | Enumeration for the credit-inquiry program's menu options. (Part 2 of 2.)

### *`CreditInquiry` Class*

Figure 17.12 contains the functionality for the credit-inquiry program, and Fig. 17.13 contains the `main` method that executes the program. The program displays a text menu and allows the credit manager to enter one of three options to obtain credit information. Option 1 (`ZERO_BALANCE`) displays accounts with zero balances. Option 2 (`CREDIT_BALANCE`) displays accounts with credit balances. Option 3 (`DEBIT_BALANCE`) displays accounts with debit balances. Option 4 (`END`) terminates program execution.

```
1  // Fig. 17.12: CreditInquiry.java
2  // This program reads a file sequentially and displays the
3  // contents based on the type of account the user requests
4  // (credit balance, debit balance or zero balance).
5  import java.io.File;
6  import java.io.FileNotFoundException;
7  import java.lang.IllegalStateException;
8  import java.util.NoSuchElementException;
9  import java.util.Scanner;
10
11 import com.deitel.ch17.AccountRecord;
12
13 public class CreditInquiry
14 {
15    private MenuOption accountType;
16    private Scanner input;
17    private final static MenuOption[] choices = { MenuOption.ZERO_BALANCE,
18       MenuOption.CREDIT_BALANCE, MenuOption.DEBIT_BALANCE,
19       MenuOption.END };
20
21    // read records from file and display only records of appropriate type
22    private void readRecords()
23    {
24       // object to store data that will be written to file
25       AccountRecord record = new AccountRecord();
```

**Fig. 17.12** | Credit-inquiry program. (Part 1 of 4.)

```
26
27          try // read records
28          {
29             // open file to read from beginning
30             input = new Scanner( new File( "clients.txt" ) );
31
32             while ( input.hasNext() ) // input the values from the file
33             {
34                record.setAccount( input.nextInt() ); // read account number
35                record.setFirstName( input.next() ); // read first name
36                record.setLastName( input.next() ); // read last name
37                record.setBalance( input.nextDouble() ); // read balance
38
39                // if proper acount type, display record
40                if ( shouldDisplay( record.getBalance() ) )
41                   System.out.printf( "%-10d%-12s%-12s%10.2f\n",
42                      record.getAccount(), record.getFirstName(),
43                      record.getLastName(), record.getBalance() );
44             } // end while
45          } // end try
46          catch ( NoSuchElementException elementException )
47          {
48             System.err.println( "File improperly formed." );
49             input.close();
50             System.exit( 1 );
51          } // end catch
52          catch ( IllegalStateException stateException )
53          {
54             System.err.println( "Error reading from file." );
55             System.exit( 1 );
56          } // end catch
57          catch ( FileNotFoundException fileNotFoundException )
58          {
59             System.err.println( "File cannot be found." );
60             System.exit( 1 );
61          } // end catch
62          finally
63          {
64             if ( input != null )
65                input.close(); // close the Scanner and the file
66          } // end finally
67       } // end method readRecords
68
69       // use record type to determine if record should be displayed
70       private boolean shouldDisplay( double balance )
71       {
72          if ( ( accountType == MenuOption.CREDIT_BALANCE )
73             && ( balance < 0 ) )
74             return true;
75
76          else if ( ( accountType == MenuOption.DEBIT_BALANCE )
77             && ( balance > 0 ) )
78             return true;
```

**Fig. 17.12** | Credit-inquiry program. (Part 2 of 4.)

```
79
80          else if ( ( accountType == MenuOption.ZERO_BALANCE )
81             && ( balance == 0 ) )
82             return true;
83
84          return false;
85       } // end method shouldDisplay
86
87       // obtain request from user
88       private MenuOption getRequest()
89       {
90          Scanner textIn = new Scanner( System.in );
91          int request = 1;
92
93          // display request options
94          System.out.printf( "\n%s\n%s\n%s\n%s\n%s\n%s\n",
95             "Enter request", " 1 - List accounts with zero balances",
96             " 2 - List accounts with credit balances",
97             " 3 - List accounts with debit balances", " 4 - End of run" );
98
99          try // attempt to input menu choice
100         {
101            do // input user request
102            {
103               System.out.print( "\n? " );
104               request = textIn.nextInt();
105            } while ( ( request < 1 ) || ( request > 4 ) );
106         } // end try
107         catch ( NoSuchElementException elementException )
108         {
109            System.err.println( "Invalid input." );
110            System.exit( 1 );
111         } // end catch
112
113         return choices[ request - 1 ]; // return enum value for option
114      } // end method getRequest
115
116      public void processRequests()
117      {
118         // get user's request (e.g., zero, credit or debit balance)
119         accountType = getRequest();
120
121         while ( accountType != MenuOption.END )
122         {
123            switch ( accountType )
124            {
125               case ZERO_BALANCE:
126                  System.out.println( "\nAccounts with zero balances:\n" );
127                  break;
128               case CREDIT_BALANCE:
129                  System.out.println( "\nAccounts with credit balances:\n" );
130                  break;
```

**Fig. 17.12** | Credit-inquiry program. (Part 3 of 4.)

```
131             case DEBIT_BALANCE:
132                System.out.println( "\nAccounts with debit balances:\n" );
133                break;
134          } // end switch
135
136          readRecords();
137          accountType = getRequest();
138       } // end while
139    } // end method processRequests
140 } // end class CreditInquiry
```

**Fig. 17.12** | Credit-inquiry program. (Part 4 of 4.)

```
1  // Fig. 17.13: CreditInquiryTest.java
2  // This program tests class CreditInquiry.
3
4  public class CreditInquiryTest
5  {
6     public static void main( String[] args )
7     {
8        CreditInquiry application = new CreditInquiry();
9        application.processRequests();
10    } // end main
11 } // end class CreditInquiryTest
```

**Fig. 17.13** | Testing the `CreditInquiry` class.

```
Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - End of run

? 1

Accounts with zero balances:
300       Pam         White             0.00

Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - End of run

? 2

Accounts with credit balances:
200       Steve       Doe            -345.67
400       Sam         Stone           -42.16
```

**Fig. 17.14** | Sample output of the credit-inquiry program in Fig. 17.13. (Part 1 of 2.)

```
Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - End of run

? 3

Accounts with debit balances:
100       Bob         Jones           24.98
500       Sue         Rich           224.62

? 4
```

**Fig. 17.14** | Sample output of the credit-inquiry program in Fig. 17.13. (Part 2 of 2.)

The record information is collected by reading through the file and determining if each record satisfies the criteria for the selected account type. Method `processRequests` (lines 116–139 of Fig. 17.12) calls method `getRequest` to display the menu options (line 119), translates the number typed by the user into a `MenuOption` and stores the result in `MenuOption` variable `accountType`. Lines 121–138 loop until the user specifies that the program should terminate. Lines 123–134 display a header for the current set of records to be output to the screen. Line 136 calls method `readRecords` (lines 22–67), which loops through the file and reads every record.

Line 30 of method `readRecords` opens the file for reading with a `Scanner`. The file will be opened for reading with a new `Scanner` object each time this method is called, so that we can again read from the beginning of the file. Lines 34–37 read a record. Line 40 calls method `shouldDisplay` (lines 70–85) to determine whether the current record satisfies the account type requested. If `shouldDisplay` returns `true`, the program displays the account information. When the end-of-file marker is reached, the loop terminates and line 65 calls the `Scanner`'s `close` method to close the `Scanner` and the file. Notice that this occurs in a `finally` block, which will execute whether or not the file was successfully read. Once all the records have been read, control returns to method `processRequests` and `getRequest` is again called (line 137) to retrieve the user's next menu option. Figure 17.13 contains method `main`, and calls method `processRequests` in line 9.

### 17.4.4 Updating Sequential-Access Files

The data in many sequential files cannot be modified without the risk of destroying other data in the file. For example, if the name "`White`" needs to be changed to "`Worthington`," the old name cannot simply be overwritten, because the new name requires more space. The record for `White` was written to the file as

```
300 Pam White 0.00
```

If the record is rewritten beginning at the same location in the file using the new name, the record will be

```
300 Pam Worthington 0.00
```

The new record is larger (has more characters) than the original record. The characters beyond the second "`o`" in "`Worthington`" will overwrite the beginning of the next sequential

record in the file. The problem here is that fields in a text file—and hence records—can vary in size. For example, 7, 14, –117, 2074 and 27383 are all `int`s stored in the same number of bytes (4) internally, but they're different-sized fields when displayed on the screen or written to a file as text. Therefore, records in a sequential-access file are not usually updated in place. Instead, the entire file is usually rewritten. To make the preceding name change, the records before `300 Pam White 0.00` would be copied to a new file, the new record (which can be of a different size than the one it replaces) would be written and the records after `300 Pam White 0.00` would be copied to the new file. Rewriting the entire file is uneconomical to update just one record, but reasonable if a substantial number of records need to be updated.

## 17.5 Object Serialization

In Section 17.4, we demonstrated how to write the individual fields of an `AccountRecord` object into a file as text, and how to read those fields from a file and place their values into an `AccountRecord` object in memory. In the examples, `AccountRecord` was used to aggregate the information for one record. When the instance variables for an `AccountRecord` were output to a disk file, certain information was lost, such as the type of each value. For instance, if the value `"3"` is read from a file, there's no way to tell whether it came from an `int`, a `String` or a `double`. We have only data, not type information, on a disk. If the program that's going to read this data "knows" what object type the data corresponds to, then the data is simply read into objects of that type. For example, in Section 17.4.2, we know that we're inputting an `int` (the account number), followed by two `String`s (the first and last name) and a `double` (the balance). We also know that these values are separated by spaces, with only one record on each line. Sometimes we'll not know exactly how the data is stored in a file. In such cases, we want to read or write an entire object from a file. Java provides such a mechanism, called **object serialization**. A so-called **serialized object** is an object represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object. After a serialized object has been written into a file, it can be read from the file and **deserialized**—that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

### Software Engineering Observation 17.1

*The serialization mechanism makes exact copies of objects. This makes it a simple way to clone objects without having to override* `Object` *method* `clone`.

### *Classes* `ObjectInputStream` *and* `ObjectOutputStream`

Classes `ObjectInputStream` and `ObjectOutputStream`, which respectively implement the **`ObjectInput`** and **`ObjectOutput`** interfaces, enable entire objects to be read from or written to a stream (possibly a file). To use serialization with files, we initialize `ObjectInputStream` and `ObjectOutputStream` objects with stream objects that read from and write to files—objects of classes `FileInputStream` and `FileOutputStream`, respectively. Initializing stream objects with other stream objects in this manner is sometimes called **wrapping**—the new stream object being created wraps the stream object specified as a constructor argument. To wrap a `FileInputStream` in an `ObjectInputStream`, for instance, we pass the `FileInputStream` object to the `ObjectInputStream`'s constructor.

### *Interfaces* `ObjectOutput` *and* `ObjectInput`

The `ObjectOutput` interface contains method **`writeObject`**, which takes an `Object` as an argument and writes its information to an `OutputStream`. A class that implements interface `ObjectOutput` (such as `ObjectOutputStream`) declares this method and ensures that the object being output implements interface `Serializable` (discussed shortly). Correspondingly, the `ObjectInput` interface contains method **`readObject`**, which reads and returns a reference to an `Object` from an `InputStream`. After an object has been read, its reference can be cast to the object's actual type. As you'll see in Chapter 27, applications that communicate via a network, such as the Internet, can also transmit entire objects across the network.

## 17.5.1 Creating a Sequential-Access File Using Object Serialization

This section and Section 17.5.2 create and manipulate sequential-access files using object serialization. The object serialization we show here is performed with byte-based streams, so the sequential files created and manipulated will be binary files. Recall that binary files typically cannot be viewed in standard text editors. For this reason, we write a separate application that knows how to read and display serialized objects. We begin by creating and writing serialized objects to a sequential-access file. The example is similar to the one in Section 17.4, so we focus only on the new features.

### *Defining Class* `AccountRecordSerializable`

Let's begin by modifying our `AccountRecord` class so that objects of this class can be serialized. Class `AccountRecordSerializable` (Fig. 17.15) implements interface **`Serializable`** (line 7), which allows objects of `AccountRecordSerializable` to be serialized and deserialized with `ObjectOutputStreams` and `ObjectInputStreams`, respectively. Interface `Serializable` is a **tagging interface**. Such an interface does not contain methods. A class that implements `Serializable` is tagged as being a `Serializable` object. This is important, because an `ObjectOutputStream` will *not* output an object unless it *is a* `Serializable` object, which is the case for any object of a class that implements `Serializable`.

```
1  // Fig. 17.15: AccountRecordSerializable.java
2  // AccountRecordSerializable class for serializable objects.
3  package com.deitel.ch17; // packaged for reuse
4
5  import java.io.Serializable;
6
7  public class AccountRecordSerializable implements Serializable
8  {
9     private int account;
10    private String firstName;
11    private String lastName;
12    private double balance;
13
14    // no-argument constructor calls other constructor with default values
15    public AccountRecordSerializable()
16    {
17       this( 0, "", "", 0.0 );
18    } // end no-argument AccountRecordSerializable constructor
```

**Fig. 17.15** | `AccountRecordSerializable` class for serializable objects. (Part 1 of 3.)

```
19
20      // four-argument constructor initializes a record
21      public AccountRecordSerializable(
22         int acct, String first, String last, double bal )
23      {
24         setAccount( acct );
25         setFirstName( first );
26         setLastName( last );
27         setBalance( bal );
28      } // end four-argument AccountRecordSerializable constructor
29
30      // set account number
31      public void setAccount( int acct )
32      {
33         account = acct;
34      } // end method setAccount
35
36      // get account number
37      public int getAccount()
38      {
39         return account;
40      } // end method getAccount
41
42      // set first name
43      public void setFirstName( String first )
44      {
45         firstName = first;
46      } // end method setFirstName
47
48      // get first name
49      public String getFirstName()
50      {
51         return firstName;
52      } // end method getFirstName
53
54      // set last name
55      public void setLastName( String last )
56      {
57         lastName = last;
58      } // end method setLastName
59
60      // get last name
61      public String getLastName()
62      {
63         return lastName;
64      } // end method getLastName
65
66      // set balance
67      public void setBalance( double bal )
68      {
69         balance = bal;
70      } // end method setBalance
71
```

**Fig. 17.15** | `AccountRecordSerializable` class for serializable objects. (Part 2 of 3.)

```
72      // get balance
73      public double getBalance()
74      {
75         return balance;
76      } // end method getBalance
77   } // end class AccountRecordSerializable
```

**Fig. 17.15** | `AccountRecordSerializable` class for serializable objects. (Part 3 of 3.)

In a `Serializable` class, every instance variable must be `Serializable`. Non-`Serializable` instance variables must be declared **`transient`** to indicate that they should be ignored during the serialization process. *By default, all primitive-type variables are serializable.* For reference-type variables, you must check the class's documentation (and possibly its superclasses) to ensure that the type is `Serializable`. For example, `String`s are `Serializable`. By default, arrays are serializable; however, in a reference-type array, the referenced objects might not be. Class `AccountRecordSerializable` contains `private` data members `account`, `firstName`, `lastName` and `balance`—all of which are `Serializable`. This class also provides `public` *get* and *set* methods for accessing the `private` fields.

### *Writing Serialized Objects to a Sequential-Access File*

Now let's discuss the code that creates the sequential-access file (Figs. 17.16–17.17). We concentrate only on new concepts here. As stated in Section 17.2, a program can open a file by creating an object of stream class `FileInputStream` or `FileOutputStream`. In this example, the file is to be opened for output, so the program creates a `FileOutputStream` (line 21 of Fig. 17.16). The `String` argument that's passed to the `FileOutputStream`'s constructor represents the name and path of the file to be opened. Existing files that are opened for output in this manner are truncated. We chose the `.ser` file extension for binary files that contain serialized objects, but this is not required.

**Common Programming Error 17.2**

*It's a logic error to open an existing file for output when, in fact, you wish to preserve the file. Class `FileOutputStream` provides an overloaded constructor that enables you to open a file and append data to the end of the file. This will preserve the file's contents.*

```
1  // Fig. 17.16: CreateSequentialFile.java
2  // Writing objects sequentially to a file with class ObjectOutputStream.
3  import java.io.FileOutputStream;
4  import java.io.IOException;
5  import java.io.ObjectOutputStream;
6  import java.util.NoSuchElementException;
7  import java.util.Scanner;
8
9  import com.deitel.ch17.AccountRecordSerializable;
10
11 public class CreateSequentialFile
12 {
13    private ObjectOutputStream output; // outputs data to file
```

**Fig. 17.16** | Sequential file created using `ObjectOutputStream`. (Part 1 of 3.)

```
14
15      // allow user to specify file name
16      public void openFile()
17      {
18         try // open file
19         {
20            output = new ObjectOutputStream(
21               new FileOutputStream( "clients.ser" ) );
22         } // end try
23         catch ( IOException ioException )
24         {
25            System.err.println( "Error opening file." );
26         } // end catch
27      } // end method openFile
28
29      // add records to file
30      public void addRecords()
31      {
32         AccountRecordSerializable record; // object to be written to file
33         int accountNumber = 0; // account number for record object
34         String firstName; // first name for record object
35         String lastName; // last name for record object
36         double balance; // balance for record object
37
38         Scanner input = new Scanner( System.in );
39
40         System.out.printf( "%s\n%s\n%s\n%s\n\n",
41            "To terminate input, type the end-of-file indicator ",
42            "when you are prompted to enter input.",
43            "On UNIX/Linux/Mac OS X type <ctrl> d then press Enter",
44            "On Windows type <ctrl> z then press Enter" );
45
46         System.out.printf( "%s\n%s",
47            "Enter account number (> 0), first name, last name and balance.",
48            "? " );
49
50         while ( input.hasNext() ) // loop until end-of-file indicator
51         {
52            try // output values to file
53            {
54               accountNumber = input.nextInt(); // read account number
55               firstName = input.next(); // read first name
56               lastName = input.next(); // read last name
57               balance = input.nextDouble(); // read balance
58
59               if ( accountNumber > 0 )
60               {
61                  // create new record
62                  record = new AccountRecordSerializable( accountNumber,
63                     firstName, lastName, balance );
64                  output.writeObject( record ); // output record
65               } // end if
```

**Fig. 17.16** | Sequential file created using `ObjectOutputStream`. (Part 2 of 3.)

```
66              else
67              {
68                 System.out.println(
69                    "Account number must be greater than 0." );
70              } // end else
71           } // end try
72           catch ( IOException ioException )
73           {
74              System.err.println( "Error writing to file." );
75              return;
76           } // end catch
77           catch ( NoSuchElementException elementException )
78           {
79              System.err.println( "Invalid input. Please try again." );
80              input.nextLine(); // discard input so user can try again
81           } // end catch
82
83           System.out.printf( "%s %s\n%s", "Enter account number (>0),",
84              "first name, last name and balance.", "? " );
85        } // end while
86     } // end method addRecords
87
88     // close file and terminate application
89     public void closeFile()
90     {
91        try // close file
92        {
93           if ( output != null )
94              output.close();
95        } // end try
96        catch ( IOException ioException )
97        {
98           System.err.println( "Error closing file." );
99           System.exit( 1 );
100       } // end catch
101    } // end method closeFile
102 } // end class CreateSequentialFile
```

**Fig. 17.16** | Sequential file created using `ObjectOutputStream`. (Part 3 of 3.)

```
1  // Fig. 17.17: CreateSequentialFileTest.java
2  // Testing class CreateSequentialFile.
3
4  public class CreateSequentialFileTest
5  {
6     public static void main( String[] args )
7     {
8        CreateSequentialFile application = new CreateSequentialFile();
9
10       application.openFile();
11       application.addRecords();
```

**Fig. 17.17** | Testing class `CreateSequentialFile`. (Part 1 of 2.)

```
12         application.closeFile();
13      } // end main
14   } // end class CreateSequentialFileTest
```

```
To terminate input, type the end-of-file indicator
when you are prompted to enter input.
On UNIX/Linux/Mac OS X type <ctrl> d then press Enter
On Windows type <ctrl> z then press Enter

Enter account number (> 0), first name, last name and balance.
? 100 Bob Jones 24.98
Enter account number (> 0), first name, last name and balance.
? 200 Steve Doe -345.67
Enter account number (> 0), first name, last name and balance.
? 300 Pam White 0.00
Enter account number (> 0), first name, last name and balance.
? 400 Sam Stone -42.16
Enter account number (> 0), first name, last name and balance.
? 500 Sue Rich 224.62
Enter account number (> 0), first name, last name and balance.
? ^Z
```

**Fig. 17.17** | Testing class `CreateSequentialFile`. (Part 2 of 2.)

Class `FileOutputStream` provides methods for writing `byte` arrays and individual `byte`s to a file, but we wish to write objects to a file. For this reason, we wrap a `FileOutputStream` in an `ObjectOutputStream` by passing the new `FileOutputStream` object to the `ObjectOutputStream`'s constructor (lines 20–21). The `ObjectOutputStream` object uses the `FileOutputStream` object to write objects into the file. Lines 20–21 may throw an **`IOException`** if a problem occurs while opening the file (e.g., when a file is opened for writing on a drive with insufficient space or when a read-only file is opened for writing). If so, the program displays an error message (lines 23–26). If no exception occurs, the file is open, and variable `output` can be used to write objects to it.

This program assumes that data is input correctly and in the proper record-number order. Method `addRecords` (lines 30–86) performs the write operation. Lines 62–63 create an `AccountRecordSerializable` object from the data entered by the user. Line 64 calls `ObjectOutputStream` method `writeObject` to write the `record` object to the output file. Only one statement is required to write the entire object.

Method `closeFile` (lines 89–101) calls `ObjectOutputStream` method **`close`** on `output` to close both the `ObjectOutputStream` and its underlying `FileOutputStream` (line 94). The call to method `close` is contained in a `try` block. Method `close` throws an `IOException` if the file cannot be closed properly. In this case, it's important to notify the user that the information in the file might be corrupted. When using wrapped streams, closing the outermost stream also closes the underlying file.

In the sample execution for the program in Fig. 17.17, we entered information for five accounts—the same information shown in Fig. 17.8. The program does not show how the data records actually appear in the file. Remember that now we're using binary files, which are not humanly readable. To verify that the file has been created successfully, the next section presents a program to read the file's contents.

### 17.5.2 Reading and Deserializing Data from a Sequential-Access File

The preceding section showed how to create a file for sequential access using object serialization. In this section, we discuss how to read serialized data sequentially from a file.

The program in Figs. 17.18–17.19 reads records from a file created by the program in Section 17.5.1 and displays the contents. The program opens the file for input by creating a `FileInputStream` object (line 21). The name of the file to open is specified as an argument to the `FileInputStream` constructor. In Fig. 17.16, we wrote objects to the file, using an `ObjectOutputStream` object. Data must be read from the file in the same format in which it was written. Therefore, we use an `ObjectInputStream` wrapped around a `FileInputStream` in this program (lines 20–21). If no exceptions occur when opening the file, variable `input` can be used to read objects from the file.

```
1  // Fig. 17.18: ReadSequentialFile.java
2  // Reading a file of objects sequentially with ObjectInputStream
3  // and displaying each record.
4  import java.io.EOFException;
5  import java.io.FileInputStream;
6  import java.io.IOException;
7  import java.io.ObjectInputStream;
8
9  import com.deitel.ch17.AccountRecordSerializable;
10
11 public class ReadSequentialFile
12 {
13    private ObjectInputStream input;
14
15    // enable user to select file to open
16    public void openFile()
17    {
18       try // open file
19       {
20          input = new ObjectInputStream(
21             new FileInputStream( "clients.ser" ) );
22       } // end try
23       catch ( IOException ioException )
24       {
25          System.err.println( "Error opening file." );
26       } // end catch
27    } // end method openFile
28
29    // read record from file
30    public void readRecords()
31    {
32       AccountRecordSerializable record;
33       System.out.printf( "%-10s%-12s%-12s%10s\n", "Account",
34          "First Name", "Last Name", "Balance" );
35
36       try // input the values from the file
37       {
```

**Fig. 17.18** | Reading a file of objects sequentially with `ObjectInputStream` and displaying each record. (Part 1 of 2.)

```
38            while ( true )
39            {
40               record = ( AccountRecordSerializable ) input.readObject();
41
42               // display record contents
43               System.out.printf( "%-10d%-12s%-12s%10.2f\n",
44                  record.getAccount(), record.getFirstName(),
45                  record.getLastName(), record.getBalance() );
46            } // end while
47         } // end try
48         catch ( EOFException endOfFileException )
49         {
50            return; // end of file was reached
51         } // end catch
52         catch ( ClassNotFoundException classNotFoundException )
53         {
54            System.err.println( "Unable to create object." );
55         } // end catch
56         catch ( IOException ioException )
57         {
58            System.err.println( "Error during read from file." );
59         } // end catch
60      } // end method readRecords
61
62      // close file and terminate application
63      public void closeFile()
64      {
65         try // close file and exit
66         {
67            if ( input != null )
68               input.close();
69         } // end try
70         catch ( IOException ioException )
71         {
72            System.err.println( "Error closing file." );
73            System.exit( 1 );
74         } // end catch
75      } // end method closeFile
76   } // end class ReadSequentialFile
```

**Fig. 17.18** | Reading a file of objects sequentially with `ObjectInputStream` and displaying each record. (Part 2 of 2.)

The program reads records from the file in method `readRecords` (lines 30–60). Line 40 calls `ObjectInputStream` method `readObject` to read an `Object` from the file. To use `AccountRecordSerializable`-specific methods, we downcast the returned `Object` to type `AccountRecordSerializable`. Method `readObject` throws an **`EOFException`** (processed at lines 48–51) if an attempt is made to read beyond the end of the file. Method `readObject` throws a `ClassNotFoundException` if the class for the object being read cannot be located. This may occur if the file is accessed on a computer that does not have the class. Figure 17.19 contains method `main` (lines 6–13), which opens the file, calls method `readRecords` and closes the file.

```
1  // Fig. 17.19: ReadSequentialFileTest.java
2  // Testing class ReadSequentialFile.
3
4  public class ReadSequentialFileTest
5  {
6     public static void main( String[] args )
7     {
8        ReadSequentialFile application = new ReadSequentialFile();
9
10       application.openFile();
11       application.readRecords();
12       application.closeFile();
13    } // end main
14 } // end class ReadSequentialFileTest
```

```
Account   First Name  Last Name      Balance
100       Bob         Jones            24.98
200       Steve       Doe            -345.67
300       Pam         White            0.00
400       Sam         Stone          -42.16
500       Sue         Rich           224.62
```

**Fig. 17.19** | Testing class `ReadSequentialFile`.

## 17.6 Additional `java.io` Classes

This section overviews additional interfaces and classes (from package `java.io`) for byte-based input and output streams and character-based input and output streams.

### 17.6.1 Interfaces and Classes for Byte-Based Input and Output

**`InputStream`** and **`OutputStream`** are `abstract` classes that declare methods for performing byte-based input and output, respectively. We used various concrete subclasses `FileInputStream` `InputStream` and `OutputStream` to manipulate files in this chapter.

#### *Pipe Streams*

**Pipes** are synchronized communication channels between threads. We discuss threads in Chapter 26. Java provides **`PipedOutputStream`** (a subclass of `OutputStream`) and **`PipedInputStream`** (a subclass of `InputStream`) to establish pipes between two threads in a program. One thread sends data to another by writing to a `PipedOutputStream`. The target thread reads information from the pipe via a `PipedInputStream`.

#### *Filter Streams*

A **`FilterInputStream`** filters an `InputStream`, and a `FilterOutputStream` filters an `OutputStream`. **Filtering** means simply that the filter stream provides additional functionality, such as aggregating data bytes into meaningful primitive-type units. `FilterInputStream` and `FilterOutputStream` are typically extended, so some of their filtering capabilities are provided by their subclasses.

A **`PrintStream`** (a subclass of `FilterOutputStream`) performs text output to the specified stream. Actually, we've been using `PrintStream` output throughout the text to this point—`System.out` and `System.err` are `PrintStream` objects.

### *Data Streams*

Reading data as raw bytes is fast, but crude. Usually, programs read data as aggregates of bytes that form `int`s, `float`s, `double`s and so on. Java programs can use several classes to input and output data in aggregate form.

Interface `DataInput` describes methods for reading primitive types from an input stream. Classes **`DataInputStream`** and `RandomAccessFile` each implement this interface to read sets of bytes and view them as primitive-type values. Interface `DataInput` includes methods such as `readBoolean`, `readByte`, `readChar`, `readDouble`, `readFloat`, `readFully` (for `byte` arrays), `readInt`, `readLong`, `readShort`, `readUnsignedByte`, `readUnsignedShort`, `readUTF` (for reading Unicode characters encoded by Java—we discuss UTF encoding in Appendix L) and `skipBytes`.

Interface `DataOutput` describes a set of methods for writing primitive types to an output stream. Classes **`DataOutputStream`** (a subclass of `FilterOutputStream`) and `RandomAccessFile` each implement this interface to write primitive-type values as bytes. Interface `DataOutput` includes overloaded versions of method `write` (for a `byte` or for a `byte` array) and methods `writeBoolean`, `writeByte`, `writeBytes`, `writeChar`, `writeChars` (for Unicode `String`s), `writeDouble`, `writeFloat`, `writeInt`, `writeLong`, `writeShort` and `writeUTF` (to output text modified for Unicode).

### *Buffered Streams*

**Buffering** is an I/O-performance-enhancement technique. With a **`BufferedOutputStream`** (a subclass of class `FilterOutputStream`), each output statement does *not* necessarily result in an actual physical transfer of data to the output device (which is a slow operation compared to processor and main memory speeds). Rather, each output operation is directed to a region in memory called a **buffer** that's large enough to hold the data of many output operations. Then, actual transfer to the output device is performed in one large **physical output operation** each time the buffer fills. The output operations directed to the output buffer in memory are often called **logical output operations**. With a `BufferedOutputStream`, a partially filled buffer can be forced out to the device at any time by invoking the stream object's **`flush`** method.

Using buffering can greatly increase the performance of an application. Typical I/O operations are extremely slow compared with the speed of accessing data in computer memory. Buffering reduces the number of I/O operations by first combining smaller outputs together in memory. The number of actual physical I/O operations is small compared with the number of I/O requests issued by the program. Thus, the program that's using buffering is more efficient.

**Performance Tip 17.1**

*Buffered I/O can yield significant performance improvements over unbuffered I/O.*

With a **`BufferedInputStream`** (a subclass of class `FilterInputStream`), many "logical" chunks of data from a file are read as one large **physical input operation** into a memory buffer. As a program requests each new chunk of data, it's taken from the buffer. (This procedure is sometimes referred to as a **logical input operation**.) When the buffer is empty, the next actual physical input operation from the input device is performed to read in the next group of "logical" chunks of data. Thus, the number of actual physical input operations is small compared with the number of read requests issued by the program.

### *Memory-Based* `byte` *Array Steams*

Java stream I/O includes capabilities for inputting from `byte` arrays in memory and outputting to `byte` arrays in memory. A `ByteArrayInputStream` (a subclass of `InputStream`) reads from a `byte` array in memory. A `ByteArrayOutputStream` (a subclass of `OutputStream`) outputs to a `byte` array in memory. One use of `byte`-array I/O is *data validation*. A program can input an entire line at a time from the input stream into a `byte` array. Then a validation routine can scrutinize the contents of the `byte` array and correct the data if necessary. Finally, the program can proceed to input from the `byte` array, "knowing" that the input data is in the proper format. Outputting to a `byte` array is a nice way to take advantage of the powerful output-formatting capabilities of Java streams. For example, data can be stored in a `byte` array, using the same formatting that will be displayed at a later time, and the `byte` array can then be output to a file to preserve the formatting.

### *Sequencing Input from Multiple Streams*

A `SequenceInputStream` (a subclass of `InputStream`) logically concatenates several `InputStreams`—the program sees the group as one continuous `InputStream`. When the program reaches the end of one input stream, that stream closes, and the next stream in the sequence opens.

## 17.6.2 Interfaces and Classes for Character-Based Input and Output

In addition to the byte-based streams, Java provides the **`Reader`** and **`Writer`** `abstract` classes, which are Unicode two-byte, character-based streams. Most of the byte-based streams have corresponding character-based concrete `Reader` or `Writer` classes.

### *Character-Based Buffering* **`Reader`***s and* **`Writer`***s*

Classes **`BufferedReader`** (a subclass of `abstract` class `Reader`) and **`BufferedWriter`** (a subclass of `abstract` class `Writer`) enable buffering for character-based streams. Remember that character-based streams use Unicode characters—such streams can process data in any language that the Unicode character set represents.

### *Memory-Based* `char` *Array* **`Reader`***s and* **`Writer`***s*

Classes **`CharArrayReader`** and **`CharArrayWriter`** read and write, respectively, a stream of characters to a `char` array. A **`LineNumberReader`** (a subclass of `BufferedReader`) is a buffered character stream that keeps track of the number of lines read—newlines, returns and carriage-return–line-feed combinations increment the line count. Keeping track of line numbers can be useful if the program needs to inform the reader of an error on a specific line.

### *Character-Based File, Pipe and String* **`Reader`***s and* **`Writer`***s*

An `InputStream` can be converted to a `Reader` via class **`InputStreamReader`**. Similarly, an `OuputStream` can be converted to a `Writer` via class **`OutputStreamWriter`**. Class `FileReader` (a subclass of `InputStreamReader`) and class `FileWriter` (a subclass of `OutputStreamWriter`) read characters from and write characters to a file, respectively. Class **`PipedReader`** and class **`PipedWriter`** implement piped-character streams for transfering data between threads. Class **`StringReader`** and **`StringWriter`** read characters from and write characters to `String`s, respectively. A `PrintWriter` writes characters to a stream.

## 17.7 Opening Files with JFileChooser

Class **JFileChooser** displays a dialog (known as the JFileChooser dialog) that enables the user to easily select files or directories. To demonstrate this dialog, we enhance the example in Section 17.3, as shown in Figs. 17.20–17.21. The example now contains a graphical user interface, but still displays the same data as before. The constructor calls method analyzePath in line 34. This method then calls method getFile in line 68 to retrieve the File object.

```
1  // Fig. 17.20: FileDemonstration.java
2  // Demonstrating JFileChooser.
3  import java.awt.BorderLayout;
4  import java.awt.event.ActionEvent;
5  import java.awt.event.ActionListener;
6  import java.io.File;
7  import javax.swing.JFileChooser;
8  import javax.swing.JFrame;
9  import javax.swing.JOptionPane;
10 import javax.swing.JScrollPane;
11 import javax.swing.JTextArea;
12 import javax.swing.JTextField;
13
14 public class FileDemonstration extends JFrame
15 {
16    private JTextArea outputArea; // used for output
17    private JScrollPane scrollPane; // used to provide scrolling to output
18
19    // set up GUI
20    public FileDemonstration()
21    {
22       super( "Testing class File" );
23
24       outputArea = new JTextArea();
25
26       // add outputArea to scrollPane
27       scrollPane = new JScrollPane( outputArea );
28
29       add( scrollPane, BorderLayout.CENTER ); // add scrollPane to GUI
30
31       setSize( 400, 400 ); // set GUI size
32       setVisible( true ); // display GUI
33
34       analyzePath(); // create and analyze File object
35    } // end FileDemonstration constructor
36
37    // allow user to specify file or directory name
38    private File getFileOrDirectory()
39    {
40       // display file dialog, so user can choose file or directory to open
41       JFileChooser fileChooser = new JFileChooser();
42       fileChooser.setFileSelectionMode(
43          JFileChooser.FILES_AND_DIRECTORIES );
```

**Fig. 17.20** | Demonstrating JFileChooser. (Part 1 of 3.)

```
44
45          int result = fileChooser.showOpenDialog( this );
46
47          // if user clicked Cancel button on dialog, return
48          if ( result == JFileChooser.CANCEL_OPTION )
49             System.exit( 1 );
50
51          File fileName = fileChooser.getSelectedFile(); // get File
52
53          // display error if invalid
54          if ( ( fileName == null ) || ( fileName.getName().equals( "" ) ) )
55          {
56             JOptionPane.showMessageDialog( this, "Invalid Name",
57                "Invalid Name", JOptionPane.ERROR_MESSAGE );
58             System.exit( 1 );
59          } // end if
60
61          return fileName;
62       } // end method getFile
63
64       // display information about file or directory user specifies
65       public void analyzePath()
66       {
67          // create File object based on user input
68          File name = getFileOrDirectory();
69
70          if ( name.exists() ) // if name exists, output information about it
71          {
72             // display file (or directory) information
73             outputArea.setText( String.format(
74                "%s%s\n%s\n%s\n%s\n%s%s\n%s%s\n%s%s\n%s%s\n%s%s",
75                name.getName(), " exists",
76                ( name.isFile() ? "is a file" : "is not a file" ),
77                ( name.isDirectory() ? "is a directory" :
78                   "is not a directory" ),
79                ( name.isAbsolute() ? "is absolute path" :
80                   "is not absolute path" ), "Last modified: ",
81                name.lastModified(), "Length: ", name.length(),
82                "Path: ", name.getPath(), "Absolute path: ",
83                name.getAbsolutePath(), "Parent: ", name.getParent() ) );
84
85             if ( name.isDirectory() ) // output directory listing
86             {
87                String[] directory = name.list();
88                outputArea.append( "\n\nDirectory contents:\n" );
89
90                for ( String directoryName : directory )
91                   outputArea.append( directoryName + "\n" );
92             } // end else
93          } // end outer if
94          else // not file or directory, output error message
95          {
```

**Fig. 17.20** | Demonstrating `JFileChooser`. (Part 2 of 3.)

```
96             JOptionPane.showMessageDialog( this, name +
97                " does not exist.", "ERROR", JOptionPane.ERROR_MESSAGE );
98          } // end else
99       } // end method analyzePath
100   } // end class FileDemonstration
```

**Fig. 17.20** | Demonstrating `JFileChooser`. (Part 3 of 3.)

```
1  // Fig. 17.21: FileDemonstrationTest.java
2  // Testing class FileDemonstration.
3  import javax.swing.JFrame;
4
5  public class FileDemonstrationTest
6  {
7     public static void main( String[] args )
8     {
9        FileDemonstration application = new FileDemonstration();
10       application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11    } // end main
12 } // end class FileDemonstrationTest
```

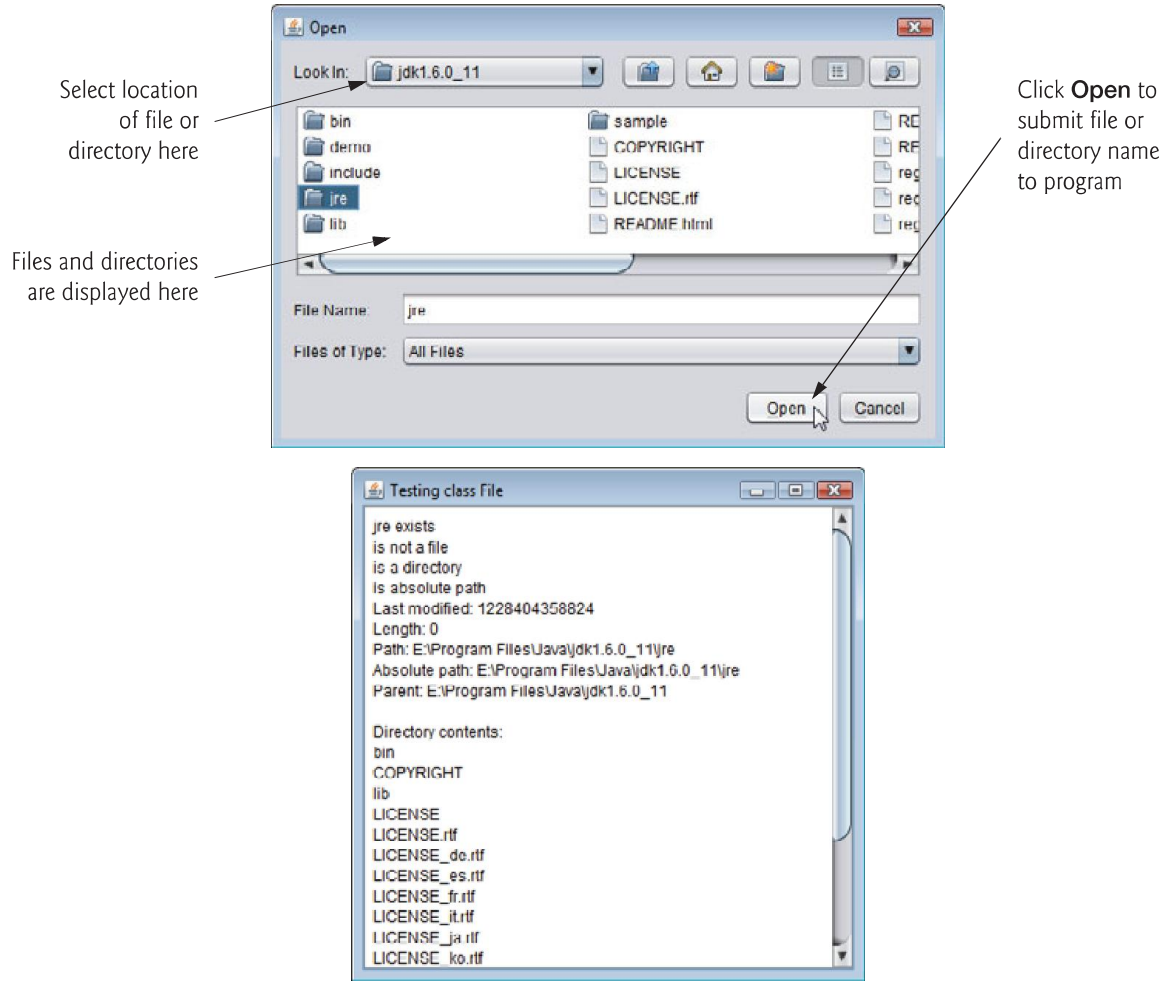



**Fig. 17.21** | Testing class `FileDemonstration`.

Method `getFile` is defined in lines 38–62 of Fig. 17.20. Line 41 creates a `JFileChooser` and assigns its reference to `fileChooser`. Lines 42–43 call method **`setFileSelectionMode`** to specify what the user can select from the `fileChooser`. For this program, we use `JFileChooser` `static` constant **`FILES_AND_DIRECTORIES`** to indicate that files and directories can be selected. Other `static` constants include **`FILES_ONLY`** (the default) and **`DIRECTORIES_ONLY`**.

Line 45 calls method **`showOpenDialog`** to display the `JFileChooser` dialog titled **Open**. Argument `this` specifies the `JFileChooser` dialog's parent window, which determines the position of the dialog on the screen. If `null` is passed, the dialog is displayed in the center of the screen—otherwise, the dialog is centered over the application window (specified by the argument `this`). A `JFileChooser` dialog is a *modal dialog* that does not allow the user to interact with any other window in the program until the user closes the `JFileChooser` by clicking the **Open** or **Cancel** button. The user selects the drive, directory or file name, then clicks **Open**. Method `showOpenDialog` returns an integer specifying which button (**Open** or **Cancel**) the user clicked to close the dialog. Line 48 tests whether the user clicked **Cancel** by comparing the result with `static` constant **`CANCEL_OPTION`**. If they're equal, the program terminates. Line 51 retrieves the file the user selected by calling `JFileChooser` method **`getSelectedFile`**. The program then displays information about the selected file or directory.

## 17.8 Wrap-Up

In this chapter, you learned how to manipulate persistent data. We compared character-based and byte-based streams, and introduced several file-processing classes from the `java.io` package. You used class `File` to retrieve information about a file or directory. You used sequential-access file processing to manipulate records that are stored in order by the record-key field. You learned the differences between text-file processing and object serialization, and used serialization to store and retrieve entire objects. The chapter concluded with a small example of using a `JFileChooser` dialog to allow users to easily select files from a GUI. The next chapter presents recursion—methods that call themselves.

## Summary

### *Section 17.1 Introduction*

- Computers use files for long-term retention of large amounts of persistent data (p. 720), even after the programs that created the data terminate.
- Computers store files on secondary storage devices (p. 720) such as hard disks.

### *Section 17.2 Files and Streams*

- Java views each file as a sequential stream of bytes (p. 720).
- Every operating system provides a mechanism to determine the end of a file, such as an end-of-file marker (p. 720) or a count of the total bytes in the file.
- Byte-based streams (p. 721) represent data in binary format.
- Character-based streams (p. 721) represent data as sequences of characters.
- Files that are created using byte-based streams are binary files (p. 721). Files created using character-based streams are text files (p. 721). Text files can be read by text editors, whereas binary files are read by a program that converts the data to a human-readable format.

- Java also can associate streams with different devices. Three stream objects are associated with devices when a Java program begins executing—`System.in`, `System.out` and `System.err`.

### *Section 17.3 Class `File`*

- Class `File` (p. 722) is used to obtain information about files and directories.
- Character-based input and output can be performed with classes `Scanner` and `Formatter`.
- Class `Formatter` (p. 722) enables formatted data to be output to the screen or to a file in a manner similar to `System.out.printf`.
- A file or directory's path (p. 722) specifies its location on disk.
- An absolute path (p. 722) contains all the directories, starting with the root directory (p. 722), that lead to a specific file or directory. Every file or directory on a disk drive has the same root directory in its path.
- A relative path (p. 722) starts from the directory in which the application began executing.
- A separator character (p. 726) is used to separate directories and files in the path.

### *Section 17.4 Sequential-Access Text Files*

- Java imposes no structure on a file. You must structure files to meet your application's needs.
- To retrieve data sequentially from a file, programs normally start from the beginning of the file and read all the data consecutively until the desired information is found.
- Data in many sequential files cannot be modified without the risk of destroying other data in the file. Records in a sequential-access file are usually updated by rewriting the entire file.

### *Section 17.5 Object Serialization*

- Java provides a mechanism called object serialization (p. 742) that enables entire objects to be written to or read from a stream.
- A serialized object (p. 742) is represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data it stores.
- After a serialized object has been written into a file, it can be read from the file and deserialized (p. 742) to recreate the object in memory.
- Classes `ObjectInputStream` (p. 722) and `ObjectOutputStream` (p. 722) enable entire objects to be read from or written to a stream (possibly a file).
- Only classes that implement interface `Serializable` (p. 743) can be serialized and deserialized.
- The `ObjectOutput` interface (p. 742) contains method `writeObject` (p. 743), which takes an `Object` as an argument and writes its information to an `OutputStream` (p. 751). A class that implements this interface, such as `ObjectOutputStream`, would ensure that the `Object` is `Serializable`.
- The `ObjectInput` interface (p. 742) contains method `readObject` (p. 743), which reads and returns a reference to an `Object` from an `InputStream`. After an object has been read, its reference can be cast to the object's actual type.

### *Section 17.6 Additional `java.io` Classes*

- `InputStream` and `OutputStream` are `abstract` classes for performing byte-based I/O.
- Pipes (p. 751) are synchronized communication channels between threads. One thread sends data via a `PipedOutputStream` (p. 751). The target thread reads information from the pipe via a `PipedInputStream` (p. 751).
- A filter stream (p. 751) provides additional functionality, such as aggregating data bytes into meaningful primitive-type units. `FilterInputStream` (p. 751) and `FilterOutputStream` are typically extended, so some of their filtering capabilities are provided by their concrete subclasses.

- A `PrintStream` (p. 751) performs text output. `System.out` and `System.err` are `PrintStreams`.
- Interface `DataInput` describes methods for reading primitive types from an input stream. Classes `DataInputStream` (p. 752) and `RandomAccessFile` each implement this interface.
- Interface `DataOutput` describes methods for writing primitive types to an output stream. Classes `DataOutputStream` (p. 752) and `RandomAccessFile` each implement this interface.
- Buffering is an I/O-performance-enhancement technique. Buffering reduces the number of I/O operations by combining smaller outputs together in memory. The number of physical I/O operations is much smaller than the number of I/O requests issued by the program.
- With a `BufferedOutputStream` (p. 752) each output operation is directed to a buffer (p. 752) large enough to hold the data of many output operations. Transfer to the output device is performed in one large physical output operation (p. 752) when the buffer fills. A partially filled buffer can be forced out to the device at any time by invoking the stream object's `flush` method (p. 752).
- With a `BufferedInputStream` (p. 752), many "logical" chunks of data from a file are read as one large physical input operation (p. 752) into a memory buffer. As a program requests data, it's taken from the buffer. When the buffer is empty, the next actual physical input operation is performed.
- A `ByteArrayInputStream` reads from a `byte` array in memory. A `ByteArrayOutputStream` outputs to a `byte` array in memory.
- A `SequenceInputStream` concatenates several `InputStreams`. When the program reaches the end of an input stream, that stream closes, and the next stream in the sequence opens.
- The `Reader` (p. 753) and `Writer` (p. 753) `abstract` classes are Unicode character-based streams. Most byte-based streams have corresponding character-based concrete `Reader` or `Writer` classes.
- Classes `BufferedReader` (p. 753) and `BufferedWriter` (p. 753) buffer character-based streams.
- Classes `CharArrayReader` (p. 753) and `CharArrayWriter` (p. 753) manipulate `char` arrays.
- A `LineNumberReader` (p. 753) is a buffered character stream tracks the number of lines read.
- Classes `FileReader` (p. 753) and `FileWriter` (p. 753) perform character-based file I/O.
- Class `PipedReader` (p. 753) and class `PipedWriter` (p. 753) implement piped-character streams for transfering data between threads.
- Class `StringReader` (p. 753) and `StringWriter` (p. 753) read characters from and write characters to `Strings`, respectively. A `PrintWriter` (p. 732) writes characters to a stream.

### *Section 17.7 Opening Files with `JFileChooser`*

- Class `JFileChooser` (p. 754) is used to display a dialog that enables users of a program to easily select files or directories from a GUI.

## Self-Review Exercises

**17.1** Determine whether each of the following statements is *true* or *false*. If *false*, explain why.

a) You must explicitly create the stream objects `System.in`, `System.out` and `System.err`.

b) When reading data from a file using class `Scanner`, if you wish to read data in the file multiple times, the file must be closed and reopened to read from the beginning of the file.

c) Method `exists` of class `File` returns `true` if the name specified as the argument to the `File` constructor is a file or directory in the specified path.

d) Binary files are human readable in a text editor.

e) An absolute path contains all the directories, starting with the root directory, that lead to a specific file or directory.

f) Class `Formatter` contains method `printf`, which enables formatted data to be output to the screen or to a file.

**17.2** Complete the following tasks, assuming that each applies to the same program:

a) Write a statement that opens file `"oldmast.txt"` for input—use `Scanner` variable `inOldMaster`.

b) Write a statement that opens file `"trans.txt"` for input—use `Scanner` variable `inTransaction`.

c) Write a statement that opens file `"newmast.txt"` for output (and creation)—use `formatter` variable `outNewMaster`.

d) Write the statements needed to read a record from the file `"oldmast.txt"`. Use the data to create an object of class `AccountRecord`—use `Scanner` variable `inOldMaster`. Assume that class `AccountRecord` is the same as the `AccountRecord` class in Fig. 17.4.

e) Write the statements needed to read a record from the file `"trans.txt"`. The record is an object of class `TransactionRecord`—use `Scanner` variable `inTransaction`. Assume that class `TransactionRecord` contains method `setAccount` (which takes an `int`) to set the account number and method `setAmount` (which takes a `double`) to set the amount of the transaction.

f) Write a statement that outputs a record to the file `"newmast.txt"`. The record is an object of type `AccountRecord`—use `Formatter` variable `outNewMaster`.

**17.3** Complete the following tasks, assuming that each applies to the same program:

a) Write a statement that opens file `"oldmast.ser"` for input—use `ObjectInputStream` variable `inOldMaster` to wrap a `FileInputStream` object.

b) Write a statement that opens file `"trans.ser"` for input—use `ObjectInputStream` variable `inTransaction` to wrap a `FileInputStream` object.

c) Write a statement that opens file `"newmast.ser"` for output (and creation)—use `ObjectOutputStream` variable `outNewMaster` to wrap a `FileOutputStream`.

d) Write a statement that reads a record from the file `"oldmast.ser"`. The record is an object of class `AccountRecordSerializable`—use `ObjectInputStream` variable `inOldMaster`. Assume class `AccountRecordSerializable` is the same as the `AccountRecordSerializable` class in Fig. 17.15

e) Write a statement that reads a record from the file `"trans.ser"`. The record is an object of class `TransactionRecord`—use `ObjectInputStream` variable `inTransaction`.

f) Write a statement that outputs a record of type `AccountRecordSerializable` to the file `"newmast.ser"`—use `ObjectOutputStream` variable `outNewMaster`.

**17.4** Find the error in each block of code and show how to correct it.

a) Assume that `account`, `company` and `amount` are declared.

```
ObjectOutputStream outputStream;
outputStream.writeInt( account );
outputStream.writeChars( company );
outputStream.writeDouble( amount );
```

b) The following statements should read a record from the file `"payables.txt"`. The `Scanner` variable `inPayable` should be used to refer to this file.

```
Scanner inPayable = new Scanner( new File( "payables.txt" ) );
PayablesRecord record = ( PayablesRecord ) inPayable.readObject();
```

## Answers to Self-Review Exercises

**17.1** a) False. These three streams are created for you when a Java application begins executing.

b) True.

c) True.

d) False. Text files are human readable in a text editor. Binary files might be human readable, but only if the bytes in the file represent ASCII characters

e) True.
f) False. Class `Formatter` contains method `format`, which enables formatted data to be output to the screen or to a file.

**17.2** a)
```
Scanner inOldMaster = new Scanner( new File ( "oldmast.txt" ) );
```
b)
```
Scanner inTransaction = new Scanner( new File( "trans.txt" ) );
```
c)
```
Formatter outNewMaster = new Formatter( "newmast.txt" );
```
d)
```
AccountRecord account = new AccountRecord();
account.setAccount( inOldMaster.nextInt() );
account.setFirstName( inOldMaster.next() );
account.setLastName( inOldMaster.next() );
account.setBalance( inOldMaster.nextDouble() );
```
e)
```
TransactionRecord transaction = new Transaction();
transaction.setAccount( inTransaction.nextInt() );
transaction.setAmount( inTransaction.nextDouble() );
```
f)
```
outNewMaster.format( "%d %s %s %.2f\n",
account.getAccount(), account.getFirstName(),
account.getLastName(), account.getBalance() );
```

**17.3** a)
```
ObjectInputStream inOldMaster = new ObjectInputStream(
   new FileInputStream( "oldmast.ser" ) );
```
b)
```
ObjectInputStream inTransaction = new ObjectInputStream(
   new FileInputStream( "trans.ser" ) );
```
c)
```
ObjectOutputStream outNewMaster = new ObjectOutputStream(
   new FileOutputStream( "newmast.ser" ) );
```
d)
```
accountRecord = ( AccountRecordSerializable ) inOldMaster.readObject();
```
e)
```
transactionRecord = ( TransactionRecord ) inTransaction.readObject();
```
f)
```
outNewMaster.writeObject( newAccountRecord );
```

**17.4** a) Error: The file was not opened before the attempt to output data to the stream.
Correction: Open a file for output by creating a new `ObjectOutputStream` object that wraps a `FileOutputStream` object.

b) Error: This example uses text files with a `Scanner`; there's no object serialization. As a result, method `readObject` cannot be used to read that data from the file. Each piece of data must be read separately, then used to create a `PayablesRecord` object.
Correction: Use `methods` of `inPayable` to read each piece of the `PayablesRecord` object.

## Exercises

**17.5** *(File Matching)* Self-Review Exercise 17.2 asked you to write a series of single statements. Actually, these statements form the core of an important type of file-processing program—namely, a file-matching program. In commercial data processing, it's common to have several files in each application system. In an accounts receivable system, for example, there's generally a master file containing detailed information about each customer, such as the customer's name, address, telephone number, outstanding balance, credit limit, discount terms, contract arrangements and possibly a condensed history of recent purchases and cash payments.

As transactions occur (i.e., sales are made and payments arrive in the mail), information about them is entered into a file. At the end of each business period (a month for some companies, a week for others, and a day in some cases), the file of transactions (called `"trans.txt"`) is applied to the master file (called `"oldmast.txt"`) to update each account's purchase and payment record. During an update, the master file is rewritten as the file `"newmast.txt"`, which is then used at the end of the next business period to begin the updating process again.

File-matching programs must deal with certain problems that do not arise in single-file programs. For example, a match does not always occur. If a customer on the master file has not made any purchases or cash payments in the current business period, no record for this customer will appear on the transaction file. Similarly, a customer who did make some purchases or cash payments could have just moved to this community, and if so, the company may not have had a chance to create a master record for this customer.

Write a complete file-matching accounts receivable program. Use the account number on each file as the record key for matching purposes. Assume that each file is a sequential text file with records stored in increasing account-number order.

a) Define class `TransactionRecord`. Objects of this class contain an account number and amount for the transaction. Provide methods to modify and retrieve these values.

b) Modify class `AccountRecord` in Fig. 17.4 to include method `combine`, which takes a `TransactionRecord` object and combines the balance of the `AccountRecord` object and the amount value of the `TransactionRecord` object.

c) Write a program to create data for testing the program. Use the sample account data in Figs. 17.22 and 17.23. Run the program to create the files `trans.txt` and `oldmast.txt` to be used by your file-matching program.

| Master file account number | Name | Balance |
|---|---|---|
| 100 | Alan Jones | 348.17 |
| 300 | Mary Smith | 27.19 |
| 500 | Sam Sharp | 0.00 |
| 700 | Suzy Green | -14.22 |

**Fig. 17.22** | Sample data for master file.

| Transaction file account number | Transaction amount |
|---|---|
| 100 | 27.14 |
| 300 | 62.11 |
| 400 | 100.56 |
| 900 | 82.17 |

**Fig. 17.23** | Sample data for transaction file.

d) Create class `FileMatch` to perform the file-matching functionality. The class should contain methods that read `oldmast.txt` and `trans.txt`. When a match occurs (i.e., records with the same account number appear in both the master file and the transaction file), add the dollar amount in the transaction record to the current balance in the master record, and write the `"newmast.txt"` record. (Assume that purchases are indicated by positive amounts in the transaction file and payments by negative amounts.) When there's a master record for a particular account, but no corresponding transaction record, merely write the master record to `"newmast.txt"`. When there's a transaction record, but no corresponding master record, print to a log file the message `"Unmatched transaction record for account number..."` (fill in the account number from the transaction record). The log file should be a text file named `"log.txt"`.

**17.6** *(File Matching with Multiple Transactions)* It's possible (and actually common) to have several transaction records with the same record key. This situation occurs, for example, when a customer makes several purchases and cash payments during a business period. Rewrite your accounts receivable file-matching program from Exercise 17.5 to provide for the possibility of handling several transaction records with the same record key. Modify the test data of `CreateData.java` to include the additional transaction records in Fig. 17.24.

| Account number | Dollar amount |
|---|---|
| 300 | 83.89 |
| 700 | 80.78 |
| 700 | 1.53 |

**Fig. 17.24** | Additional transaction records.

**17.7** *(File Matching with Object Serialization)* Recreate your solution for Exercise 17.6 using object serialization. Use the statements from Exercise 17.3 as your basis for this program. You may want to create applications to read the data stored in the `.ser` files—the code in Section 17.5.2 can be modified for this purpose.

**17.8** *(Telephone-Number Word Generator)* Standard telephone keypads contain the digits zero through nine. The numbers two through nine each have three letters associated with them (Fig. 17.25). Many people find it difficult to memorize phone numbers, so they use the correspondence between digits and letters to develop seven-letter words that correspond to their phone numbers. For example, a person whose telephone number is 686-2377 might use the correspondence indicated in Fig. 17.25 to develop the seven-letter word "NUMBERS." Every seven-letter word corresponds to exactly one seven-digit telephone number. A restaurant wishing to increase its takeout business could surely do so with the number 825-3688 (i.e., "TAKEOUT").

| Digit | Letters | Digit | Letters | Digit | Letters |
|---|---|---|---|---|---|
| 2 | A B C | 5 | J K L | 8 | T U V |
| 3 | D E F | 6 | M N O | 9 | W X Y |
| 4 | G H I | 7 | P R S | | |

**Fig. 17.25** | Telephone keypad digits and letters.

Every seven-letter phone number corresponds to many different seven-letter words, but most of these words represent unrecognizable juxtapositions of letters. It's possible, however, that the owner of a barbershop would be pleased to know that the shop's telephone number, 424-7288, corresponds to "HAIRCUT." A veterinarian with the phone number 738-2273 would be pleased to know that the number corresponds to the letters "PETCARE." An automotive dealership would be pleased to know that the dealership number, 639-2277, corresponds to "NEWCARS."

Write a program that, given a seven-digit number, uses a `PrintStream` object to write to a file every possible seven-letter word combination corresponding to that number. There are 2,187 ($3^7$) such combinations. Avoid phone numbers with the digits 0 and 1.

**17.9** *(Student Poll)* Figure 7.8 contains an array of survey responses that's hard coded into the program. Suppose we wish to process survey results that are stored in a file. This exercise requires two

separate programs. First, create an application that prompts the user for survey responses and outputs each response to a file. Use a `Formatter` to create a file called `numbers.txt`. Each integer should be written using method `format`. Then modify the program in Fig. 7.8 to read the survey responses from `numbers.txt`. The responses should be read from the file by using a `Scanner`. Use method `nextInt` to input one integer at a time from the file. The program should continue to read responses until it reaches the end of the file. The results should be output to the text file `"output.txt"`.

**17.10** *(Adding Object Serialization to the* **`MyShape`** *Drawing Application)* Modify Exercise 14.17 to allow the user to save a drawing into a file or load a prior drawing from a file using object serialization. Add buttons **Load** (to read objects from a file) and **Save** (to write objects to a file). Use an `ObjectOutputStream` to write to the file and an `ObjectInputStream` to read from the file. Write the array of `MyShape` objects using method `writeObject` (class `ObjectOutputStream`), and read the array using method `readObject` (`ObjectInputStream`). The object-serialization mechanism can read or write entire arrays—it's not necessary to manipulate each element of the array of `MyShape` objects individually. It's simply required that all the shapes be `Serializable`. For both the **Load** and **Save** buttons, use a `JFileChooser` to allow the user to select the file in which the shapes will be stored or from which they'll be read. When the user first runs the program, no shapes should be displayed on the screen. The user can display shapes by opening a previously saved file or by drawing new shapes. Once there are shapes on the screen, users can save them to a file using the **Save** button.

## Making a Difference

**17.11** *(Phishing Scanner)* Phishing is a form of identity theft in which, in an e-mail, a sender posing as a trustworthy source attempts to acquire private information, such as your user names, passwords, credit-card numbers and social security number. Phishing e-mails claiming to be from popular banks, credit-card companies, auction sites, social networks and online payment services may look quite legitimate. These fraudulent messages often provide links to spoofed (fake) websites where you're asked to enter sensitive information.

Visit McAfee® (`www.mcafee.com/us/threat_center/anti_phishing/phishing_top10.html`), Security Extra (`www.securityextra.com/`) and other websites to find lists of the top phishing scams. Also check out the Anti-Phishing Working Group (`www.antiphishing.org/`), and the FBI's Cyber Investigations website (`www.fbi.gov/cyberinvest/cyberhome.htm`), where you'll find information about the latest scams and how to protect yourself.

Create a list of 30 words, phrases and company names commonly found in phishing messages. Assign a point value to each based on your estimate of its likeliness to be in a phishing message (e.g., one point if it's somewhat likely, two points if moderately likely, or three points if highly likely). Write an application that scans a file of text for these terms and phrases. For each occurrence of a keyword or phrase within the text file, add the assigned point value to the total points for that word or phrase. For each keyword or phrase found, output one line with the word or phrase, the number of occurrences and the point total. Then show the point total for the entire message. Does your program assign a high point total to some actual phishing e-mails you've received? Does it assign a high point total to some legitimate e-mails you've received?