

FILE INPUT AND OUTPUT

In this chapter, you will:

- ◆ Use the File class
- ◆ Understand data file organization and streams
- ◆ Use streams
- ◆ Write to and read from a file
- ◆ Write formatted file data
- ◆ Read formatted file data
- ◆ Use a variable filename
- ◆ Create random access files

Haven't I seen you spending a lot of time at the keyboard lately?" asks Lynn Greenbrier one day at Event Handlers Incorporated.

"I'm afraid so," you answer. "I'm trying to write a program that displays a month's scheduled events, one at a time. Every time I run it, I have to enter the data for every event—the host's name, the number of guests, and so on."

"You're typing all the data over and over again?" Lynn asks in disbelief. "It's time for me to show you how to save data to a file."

PREVIEWING A PROGRAM THAT USES FILE DATA

Event Handlers Incorporated stores a record of each scheduled event in a file. Any employee in the company can view the scheduled events on-screen. You will create a similar program in this chapter; however, you can now use a completed version of the `Chap16ReadEventFile` program that is saved in the `Chapter.16` folder on your Student Disk.



For convenience, the data file used in this example is stored on a floppy disk. However, business files are usually stored on a hard disk, either locally or on a network server. No matter where a data file is physically located, the process of saving and retrieving the file is the same.

To use the `Chap16ReadEventFile` class:

1. Go to the command prompt for the `Chapter.16` folder on your Student Disk, type `java Chap16ReadEventFile`, and then press **[Enter]**. This program lets you view previously stored data about events, one event at a time.
2. Click the **View Event** button. The data for the Albertson event appears and shows that the event is scheduled for the 12th day of the month with 100 guests. See Figure 16-1.



Figure 16-1 Chap16ReadEventFile program

3. Click the **View Event** button again to view the data for five additional events. Click the **Close** button to exit the program at any time. The program will also exit automatically when you reach the end of the stored data file.

USING THE FILE CLASS

Computer users use the term **file** to describe the objects that they store on permanent storage devices, such as hard, floppy, or zip disks, reels of magnetic tape, or compact discs. Some files are **data files** that contain facts and figures, such as employee numbers, names, and salaries; some files are **program files**, also called applications, that store software instructions. Other files can store graphics, text, or operating system instructions.

Although their contents vary, files have many common characteristics—each file occupies a section of disk (or other storage device) space, has a name and a specific time of creation. You can use Java's **File class** to gather file information. The File class does not provide any opening, processing, or closing capabilities for files. Rather, you use the File class to obtain information about a file, such as whether it exists or is open, its size, and its last modification date.

You must include the statement `import java.io.*` in any program that uses the File class. The `java.io` package contains all the classes you use in file processing. The File class is a direct descendant of the Object class. You can create a File object using a constructor that includes a filename; for example, `File someData = new File("data.txt");`, where `data.txt` is a file on the default disk drive. You can also specify a path for the file, as in `File someData = new File("A:\\Chapter.16\\data.txt");`, in which the argument to the constructor contains a disk drive and path. Table 16-1 lists some useful File class methods.



The *io* in `java.io` stands for *input/output*.



Recall that the backslash (`\`) is used as part of an escape sequence in Java. You must type two backslashes to indicate a single backslash to the operating system. You learned about the escape sequence in Chapter 2.

Table 16-1 Selected File class methods

Method Signature	Purpose
<code>boolean canRead()</code>	Returns <code>true</code> if a file is readable
<code>boolean canWrite()</code>	Returns <code>true</code> if a file is writeable
<code>boolean exists()</code>	Returns <code>true</code> if a file exists
<code>String getName()</code>	Returns the file's name
<code>String getPath()</code>	Returns the file's path
<code>String getParent()</code>	Returns the name of the folder in which the file can be found
<code>long length()</code>	Returns the file's size
<code>long lastModified()</code>	Returns the time the file was last modified; this time is system dependent and should be used only for comparison with other files' times, and not as an absolute time

Next you will write a class that examines a file and prints appropriate messages concerning its status.

To create a class that uses a File object:

1. Open a new file in your text editor, and then type the following first few lines of a class that checks a file's status:

```
import java.io.*;
public class CheckFile
{
    public static void main(String[] args)
    {
```

2. Enter the following line to create a File object that represents a disk file named data.txt. Although the filename on the disk is data.txt, within the program the filename is myFile.

```
File myFile = new File("data.txt");
```

3. Enter the following if...else statements to test for the file's existence. If the File object myFile exists, print its name and size, and then test whether the file can be read or written. If the file does not exist, simply print a message indicating that fact.

```
if(myFile.exists())
{
    System.out.println(myFile.getName() + " exists");
    System.out.println("The file is " +
        myFile.length () + " bytes long");
    if(myFile.canRead())
        System.out.println(" ok to read");
    if(myFile.canWrite())
        System.out.println(" ok to write");
}
else
    System.out.println("File does not exist");
```

4. Add a closing curly brace for the main() method and a closing curly brace for the class.
5. Save the file as **CheckFile.java** in the Chapter.16 folder on your Student Disk, and then compile the file.
6. Open a new file in your text editor, and then type the company name, **Event Handlers Incorporated!**. Save this file as **data.txt** in the current directory (the Chapter.16 folder on your Student Disk).
7. Run the CheckFile program using the command **java CheckFile**. The output appears in Figure 16-2. The file is 28 bytes long because each character you typed, including spaces and punctuation, consumes one byte of storage.



```
Command Prompt
A:\Chapter.16>java CheckFile
data.txt exists
The file is 28 bytes long
ok to read
ok to write
A:\Chapter.16>_
```

Figure 16-2 Output of the CheckFile program



If you added comments to the beginning of your `data.txt` file or mistyped the company name, then the total number of characters in your file might differ from 28.

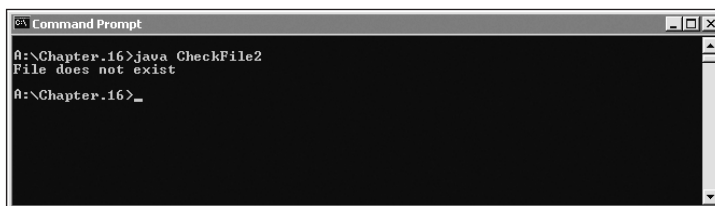
Next you will change the program to test for a file that does not exist.

To check for a nonexistent file:

1. Open the **CheckFile.java** file from the Chapter.16 folder in your text editor, and then immediately save it as **CheckFile2.java**.
2. Change the class name to **CheckFile2**.
3. Change the filename in the File constructor so that it refers to a nonexistent file:

```
File myFile = new File("nodata.txt");
```

4. Save the file, and then compile and run the program. Unless you have a file named `nodata.txt` on your Student Disk, the output looks like Figure 16-3.



```
Command Prompt
A:\Chapter.16>java CheckFile2
File does not exist
A:\Chapter.16>_
```

Figure 16-3 Output of the CheckFile2 program

In the preceding steps, the program found the file named `data.txt` because the file was physically located in the current directory from which you were working. You can check the status of files in other directories by using a File constructor with two String arguments. The first String represents a path to the filename, and the second String represents the filename. For example, `File someFile = new File("\\com\\EventHandlers","data.txt");` refers to the `data.txt` file located in the `EventHandlers` folder within the `com` folder in the root directory.

Next you will create a second data file so that you can compare its size and time stamp with the `data.txt` file.

To create a **data2.txt** file and a program for comparing **data2.txt** to **data.txt**:

1. Open a new file in your text editor, and then type a shorter version of the company name, **Event Handlers**.
2. Save the file as **data2.txt** in the Chapter.16 folder on your Student Disk.
3. Open a new file in your text editor, and then type the following first few lines of the **CheckTwoFiles** program:

```
import java.io.*;
public class CheckTwoFiles
{
    public static void main(String[] args)
    {
```

4. Enter the following code to declare two file objects:

```
File f1 = new File("data.txt");
File f2 = new File("data2.txt");
```

5. Enter the following code to determine whether both files exist. If they do, comparing their creation times determines which file has the more recent time stamp, and then the program prints the filename of that file. (*Note:* Do not add a closing curly brace for the **if** statement in this step; in the next step you will continue to add statements that belong within this **if** structure.)

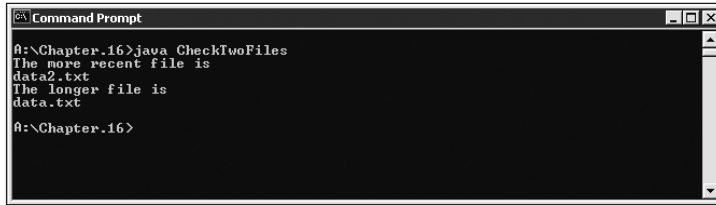
```
if(f1.exists() && f2.exists())
{
    System.out.println("The more recent file is ");
    if(f1.lastModified() > f2.lastModified())
        System.out.println(f1.getName());
    else
        System.out.println(f2.getName());
```

6. Enter additional statements within the **if** block that executes when both files exist. These statements compare the length of the files and print the name of the longer file—that is, the one that contains more bytes:

```
System.out.println("The longer file is ");
if(f1.length() > f2.length())
    System.out.println(f1.getName());
else
    System.out.println(f2.getName());
```

7. Add three curly braces—one to end the **if** statement that checks whether both files exist, one for the **main()** method, and one for the class.

8. Save the file as **CheckTwoFiles.java** in the Chapter.16 folder of your Student Disk, and then compile and run the program. The output appears in Figure 16-4. Note that the **data2.txt** file was created after the **data.txt** file, so it is more recent, but it has fewer characters.



```
A:\Chapter.16>java CheckTwoFiles
The more recent file is
data2.txt
The longer file is
data.txt
A:\Chapter.16>
```

Figure 16-4 Output of the CheckTwoFiles program

UNDERSTANDING DATA FILE ORGANIZATION AND STREAMS

Most businesses generate and use large quantities of data every day. You can store data in variables within a program, but this type of storage is temporary. When the program ends, the variables no longer exist and the data is lost. Variables are stored in the computer's main or primary memory, which is called RAM (random access memory). When you need to retain data for any significant amount of time, you must save the data on a permanent, secondary storage device such as a floppy disk, hard drive, or compact disc (CD).



Because you can erase data from files, some programmers prefer the term *persistent* storage over permanent storage. In other words, you can remove data so it is not permanent, but the data remains in the file even when the computer loses power, so, unlike RAM, the data will persist, or persevere.

Data used by businesses is stored in a data hierarchy, as shown in Figure 16-5. The smallest, useful piece of data that is of interest to most people is the character. A character is any one of the letters, numbers, or other special symbols, such as punctuation marks, you can read and to which you can assign meaning. Characters are made up of bits (the zeros and ones that represent computer circuitry), but people who use data are not concerned with whether the internal representation for an A is 01000001 or 10111110.



Java uses Unicode to represent its characters. You first learned about Unicode in Chapter 2.

When businesses use data, they group characters into fields. A **field** is a group of characters that has some meaning. For example, the characters *T*, *o*, and *m* might represent your first name. Other data fields might represent items such as last name, Social Security number, zip code, and salary.

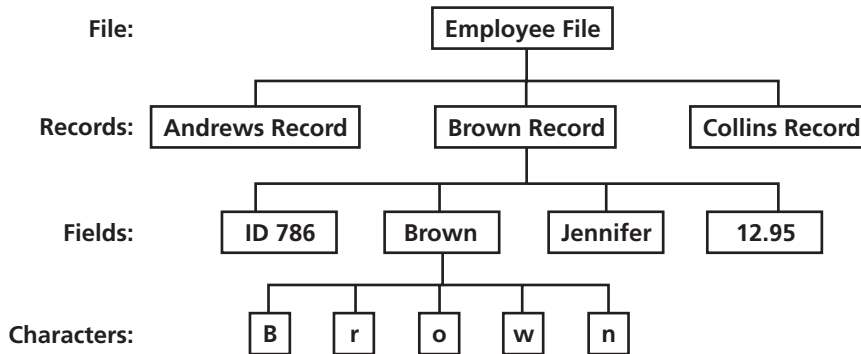


Figure 16-5 Data hierarchy

Fields are grouped together to form **records**. An individual's first and last names, Social Security number, zip code, and salary represent that individual's record. When programming in Java, you have created many classes, such as an `Employee` class or a `Student` class. You can think of the data typically stored in each of these classes as a record. These classes contain individual variables that represent data fields. A business's data records usually represent a person, item, sales transaction, or some other concrete object or event.

Records are grouped to create files. **Files** consist of related records, such as a company's personnel file that contains one record for each company employee. Some files have only a few records; perhaps your professor maintains a file for your class with 25 records—one record for each student. Other files contain thousands or even millions of records. For example, a large insurance company maintains a file of policyholders, and a mail-order catalog company maintains a file of available items.

Before a program can use a data file, the program must open the file. Similarly, when you finish using a file, the program should close the file. If you fail to close an input file—that is, a file from which you are reading data, there usually are not any serious consequences; the data still exists in the file. However, if you fail to close an output file—that is, a file to which you are writing data, the data may become inaccessible. You should always close every file you open, and you should close the file as soon as you no longer need it. When you leave a file open for no reason, you use computer resources and your computer's performance suffers. Also, particularly within a network, another program might be waiting to use the file.

While people view files as a series of records with each record containing data fields, Java views files as a series of bytes. When you perform an input operation in a program, you can picture bytes flowing into your program from an input device through a **stream**, which functions as a pipeline or channel. When you perform output, some bytes flow out of your program through another stream to an output device, as shown in Figure 16-6. A stream is an object, and like all objects, streams have data and methods. The methods allow you to perform actions such as opening, closing, and flushing (clearing) the stream.



Figure 16-6 File streams

Most streams flow in only one direction; each stream is either an input or output stream. You might open several streams at once within your program. For example, three streams are required by a program that reads a data disk. One input stream checks the data for invalid values, and then one output stream writes some records to a file of valid records; another output stream writes other records to a file of invalid records.



Random access files use streams that flow in two directions. You will use a random access file later in this chapter.

Tip

USING STREAMS

Figure 16-7 shows a partial structure of Java's Stream classes; it shows that `InputStream` and `OutputStream` are subclasses of the `Object` class. **`InputStream`** and **`OutputStream`** are abstract classes that contain methods for performing input and output. As abstract classes, these classes contain methods that must be overridden in their child classes. The capabilities of the most commonly used classes that provide input and output are summarized in Table 16-2.

Table 16-2 Description of selected classes used for input and output

Class	Description
<code>InputStream</code>	Abstract class containing methods for performing input
<code>OutputStream</code>	Abstract class containing methods for performing output
<code>FileInputStream</code>	Child of <code>InputStream</code> that provides the capability to read from disk files
<code>FileOutputStream</code>	Child of <code>OutputStream</code> that provides the capability to write to disk files
<code>PrintStream</code>	Child of <code>FilterOutputStream</code> , which is a child of <code>OutputStream</code> ; <code>PrintStream</code> handles output to a system's standard (or default) output device, usually the monitor
<code>BufferedInputStream</code>	Child of <code>FilterInputStream</code> , which is a child of <code>InputStream</code> ; <code>BufferedInputStream</code> handles input from a system's standard (or default) input device, usually the keyboard



Figure 16-7 Partial structure of the Stream classes

Java's `System` class declares a `PrintStream` object. This object is `System.out`, which you have used extensively in this book. Besides `System.out`, the `System` class defines an additional `PrintStream` object named `System.err`. The output from `System.err` and `System.out` can go to the same device; in fact, `System.err` and `System.out` are both directed to the command line on the monitor. The difference is that `System.err` usually is reserved for error messages, and `System.out` is reserved for valid output. You can direct either `System.err` or `System.out` to a new location, such as a disk file or printer. For example, you might want to keep a hard copy log of the error messages generated by a program, but direct the standard output to a disk file.

Figure 16-7 shows that the `InputStream` class is parent to `FilterInputStream`, which is parent to `BufferedInputStream`. The object `System.in` is a `BufferedInputStream` object. The `System.in` object captures keyboard input. Recall that you have used this object with its `read()` method. A **buffer** is a small memory location that you use to hold data temporarily. The `BufferedInputStream` class allows keyboard data to be held until the user presses [Enter]. That way, the user can backspace over typed characters to change the data before the program stores it. This allows the operating system—instead of your program—to handle the complicated tasks of deleting characters as the user backspaces, and then replacing the deleted characters with new ones.



Using a buffer to hold input or output improves program performance. Input and output operations are relatively slow compared to computer processor speeds. Holding input or output until there is a “batch” makes programs run faster.

You can create your own `InputStream` and `OutputStream` objects and assign to them `System.in` and `System.out`, respectively. Then you can use the `InputStream`’s `read()` method to read in one character at a time from the location you choose. The `read()` method returns an integer that represents the Unicode value of the typed character; it returns a value of `-1` when it encounters an end-of-file condition, known as EOF.



You can also identify EOF by throwing an `EOFException`. You will use this technique later in this chapter.

Next you will create `InputStream` and `OutputStream` objects so you can read from the keyboard and write to the screen. Of course, you have already written many programs that read from the keyboard and write to the screen without using these objects. By using them here with the default input/output devices, you can easily modify the `InputStream` and `OutputStream` objects at a later time; then you can use whatever input and output devices you choose.

To create a program that reads from the keyboard and writes to the screen:

1. Open a new file in your text editor, and then type the following first few lines of a program that will allow a user to input data from the keyboard and then will echo that data to the screen. The class name is `ReadKBWriteScreen`:
2. Add the following header and opening curly brace for the `main()` method. The `main()` method **throws** an `IOException` because you will perform input and output operations.

```
import java.io.*;
public class ReadKBWriteScreen
{
```

```
public static void main(String[] args) throws IOException
{
```

3. Enter the following code to declare `InputStream` and `OutputStream` objects, as well as an integer to hold each character the user types:

```
InputStream istream;
OutputStream ostream;
int c;
```

4. Enter the following code to assign the `System.in` object to `istream`, and the `System.out` object to `ostream`. Then add a prompt telling the user to type some characters.

```

istream = System.in;
ostream = System.out;
System.out.println("Type some characters ");

```

5. Use the following `try` block to read from the file. If an `IOException` occurs, you can print an appropriate message. Within the `try` block, execute a loop that reads from the keyboard until the end-of-file condition occurs (when the `read()` method returns `-1`). While there is not an end-of-file condition, send the character to the `ostream` object.



You learned about `try` blocks in Chapter 15.

```

try
{
    while((c = istream.read()) != -1)
        ostream.write(c);
}

```

6. Use the following `catch` block to handle any `IOException`:

```

catch(IOException e)
{
    System.out.println("Error: " + e.getMessage());
}

```

7. Regardless of whether an `IOException` occurs, you want to close the streams. Use the following `finally` block to ensure that the streams are closed:

```

finally
{
    istream.close();
    ostream.close();
}

```

8. Add a closing curly brace for the `main()` method and another for the class.
9. Save the file as **ReadKBWriteScreen.java** in the Chapter.16 folder on your Student Disk, and then compile and run the program. At the command line, type any series of characters and then press **[Enter]**. As you type characters, the buffer holds them until you press **[Enter]**, at which time the stored characters echo to the screen. Do not try to end the program yet.

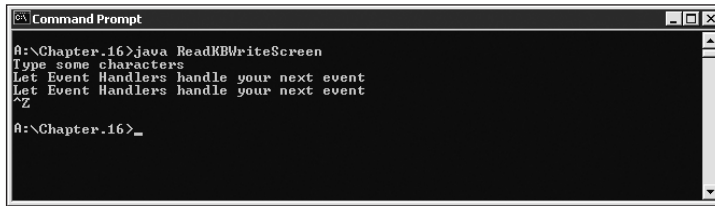
The `while` loop in the `ReadKBWriteScreen` program continues until the `read()` method returns `-1`. However, you cannot end the program by typing `-1`. Typing a minus sign (`-`) and a one (`1`) causes two additional characters to be sent to the buffer, and neither of those characters represents `-1`. Instead, you must press **[Ctrl] + Z**, which forces the `read()` method to return `-1`, and which the operating system recognizes as the end of the file. Next you will end the `ReadKBWriteScreen` program.



Pressing [Ctrl] + Z to end a program is an operating system command, not a Java command.

To end the ReadKBWriteScreen program:

1. At the command line, press **[Ctrl]+Z**, and then press **[Enter]**. The program ends. Figure 16-8 shows a typical program execution. Notice that the keystroke combination [Ctrl]+Z appears on the screen as ^Z.



```
A:\Chapter.16>java ReadKBWriteScreen
Type some characters
Let Event Handlers handle your next event
Let Event Handlers handle your next event
^Z
A:\Chapter.16>_
```

Figure 16-8 Typical execution of ReadKBWriteScreen program

2. Execute the program again. This time, type a line of characters, press **[Enter]**, and observe the echoed output. Then type another line, and press **[Enter]**. You can type as many lines as you want. Press **[Ctrl]+Z**, and then press **[Enter]**. The program will continue to echo your lines to the screen until you end the program.



Do not press [Ctrl]+C to end the program. Doing so breaks out of the program before its completion and does not properly close the files.

WRITING TO AND READING FROM A FILE

Instead of assigning files to the standard input and output devices, you can also assign a file to the `InputStream` or `OutputStream`. For example, you can read data from the keyboard and store it permanently on a disk. To accomplish this, you can construct a `FileOutputStream` object and assign it to the `OutputStream`. If you want to change a program's output device, you don't have to make any other changes to the program other than assigning a new object to the `OutputStream`; the rest of the program's logic remains the same. Java lets you assign a file to a stream object so that screen output and file output work in exactly the same manner.

You can associate a `File` object with the output stream in one of two ways:

- You can pass the filename to the constructor of the `FileOutputStream` class.

- You can create a File object passing the filename to the File constructor. Then you can pass the File object to the constructor of the FileOutputStream class.

The second method has some benefits—if you create a File object, you can use the File class methods such as `exists()` and `lastModified()` to retrieve file information. In the next set of steps you will use a FileOutputStream to write keyboard-entered data to a file you create.



Because applets are designed for distribution over the Internet, you are not allowed to use an applet to write to files on a client's workstation. Applets that write to a client's file could destroy a client's existing data.

To create a program that writes keyboard data to a file:

1. Save the ReadKBWriteScreen.java file as **ReadKBWriteFile.java** in the Chapter.16 folder on your Student Disk.
2. Change the class header to **public class ReadKBWriteFile.**
3. Position your insertion point at the end of the line that defines the ostream object (`OutputStream ostream;`), and then press **[Enter]** to start a new line. On the new line, define a File object as follows:
File outFile = new File("datafile.dat");
4. Replace the statement that assigns `System.out` to the ostream object with the statement:
ostream = new FileOutputStream(outFile);
5. Save the file, and then compile and execute the program. At the command line, type **Event Handlers handles events of all sizes**, and then press **[Enter]**. After you press **[Enter]**, the characters will not appear on the screen; instead, they are output to a file named `datafile.dat` that is written in the default directory, the current directory from which you are working—in this case, the Chapter.16 directory.
6. Press **[Ctrl]+Z**, and then press **[Enter]** to stop the program.
7. In your text editor, open the **datafile.dat** file. The characters are an exact copy of the ones you entered at the keyboard.

You could enter any number of characters to the output stream before ending the program and they would be saved in the output file. If you run the ReadKBWriteFile program again, the program will overwrite the existing `datafile.dat` file with your new data.

Reading from a File

The process you use to read data from a file is similar to the one you use to write data to a file. You can assign a File object to the input stream, as you will do in the next steps.

To read data from a file:

1. In your text editor, open the **ReadKBWriteScreen.java** file you created earlier in this chapter, and then immediately save the file as **ReadFileWriteScreen.java** in the Chapter.16 folder of your Student Disk.
2. Change the class header to **public class ReadFileWriteScreen**.
3. Position your insertion point to the right of the statement that declares the OutputStream object named ostream, and then press **[Enter]** to start a new line. On the new line, enter the following code to create a File object to refer to the datafile.dat file you created:

```
File inFile = new File("datafile.dat");
```

4. Change the statement that assigns the System.in object to istream (**istream = System.in**) so that you can use the File object for input instead of the keyboard by replacing it with the following:


```
istream = new FileInputStream(inFile);
```
5. Remove the statement that prompts the user for input;


```
System.out.println("Type some characters ");
```

 A disk file does not need a prompt.
6. Save the file, and then compile and run the program. The data you stored in the datafile.dat file ("Event Handlers handles events of all sizes") appears on the screen, and the program ends.

WRITING FORMATTED FILE DATA

You do not usually want to read data files as a series of characters. For example, you might have a data file that contains personnel records that include an employee ID number, name, and salary for each employee in your organization. Rather than reading a series of bytes, it is more useful to be able to read such a file in groups of bytes that constitute an integer, a String, and a double. You can use the DataInputStream and DataOutputStream classes to accomplish formatted input and output.

DataOutputStream objects enable you to write binary data to an OutputStream. Much of the data that you write with DataOutputStream objects is not readable in a text editor because it is not stored as characters. Instead the data is formatted correctly for its type. For example, a double with the value 123.45 is not stored as six separate readable characters that can correctly display in a text editor. Instead, numeric values are stored in a more compact form that you can read later with a DataInputStream object.

The `DataOutput` interface is implemented by `DataOutputStream`. The `DataOutput` interface includes methods such as `writeBoolean()`, `writeChar()`, `writeDouble()`, `writeFloat()`, and `writeInt()`. Each method writes data in the correct format for the data type its name indicates. You can use the method `writeUTF()` to write Unicode format strings.



The meaning of the acronym UTF is disputed by various sources. The most popular interpretations include Unicode Transformation Format, Unicode Transfer Format, and Unicode Text Format.

When you create a `DataOutputStream`, you can assign a `FileOutputStream` object to it so that your data is stored in a file. Using `DataOutputStream` with a `FileOutputStream` allows you to use the correct write method that is appropriate for your data. When you use a `DataOutputStream` connected to `FileOutputStream`, this approach is known as **chaining the stream objects**. That is, if you define a `DataOutputStream` object with a statement such as `DataOutputStream out;`, then when you call the `DataOutputStream` constructor, you pass a `FileOutputStream` object to it (for example, `out = new FileOutputStream(new FileOutputStream("someFile"));`).

In the next series of steps, you will create a full-blown project for Event Handlers Incorporated like the one you saw in the Preview activity. The program uses a GUI interface to capture data about an event from a user, and writes that data to an output file using the `DataOutput` interface. The data required includes the host's name, the date, and the number of guests. For simplicity, this program accepts event dates for the current month only, so the date field is an integer. Figure 16-9 shows a preliminary sketch of the user's interface.

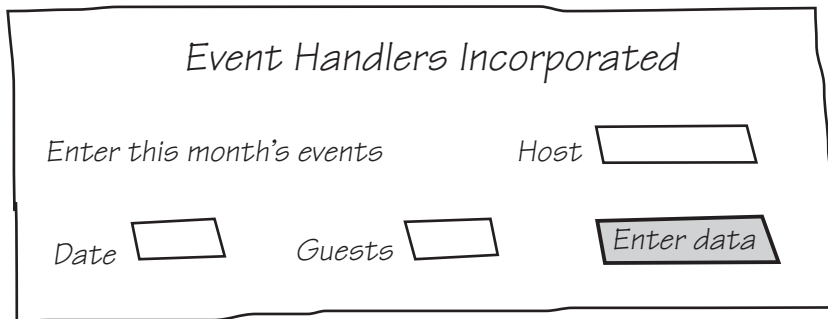


Figure 16-9 Sketch of the user's interface

To create a `JFrame` for data entry:

1. Open a new file in your text editor, and then type the following first few lines of the `CreateEventFile` class. `CreateEventFile` is a `JFrame` that reacts to a mouse click when you click an object within the `JFrame`. Therefore, you must implement `ActionListener`.


```
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class CreateEventFile extends JFrame
    implements ActionListener
{
```

2. Enter the following code to create a JLabel for the company name and a Font object to use with the company name:

```
private JLabel companyName =
    new JLabel("Event Handlers Incorporated");
Font bigFont = new Font("Helvetica", Font.ITALIC, 24);
```

3. Enter the following code to create a prompt that tells the user to enter data, and JTextFields for the host, date, and guests. Because a host's name is usually several characters long, the field for the host's name should be wider than the fields for the date and number of guests.

```
private JLabel prompt =
    new JLabel("Enter this month's events");
private JTextField host = new JTextField(10);
private JTextField date = new JTextField(4);
private JTextField guests = new JTextField(4);
```

4. Enter the following code to create a JLabel for each of the JTextFields. Include a JButton object that the user can click when a data record is completed and ready to be written to the data file. Then add a Container to hold the JFrame components.

```
private JLabel hLabel = new JLabel("Host");
private JLabel dLabel = new JLabel("Date");
private JLabel gLabel = new JLabel("Guests");
private JButton enterDataButton =
    new JButton("Enter data");
private Container con = getContentPane();
```

5. When you write the user's data to an output file, you will use the DataOutputStream class, so create a DataOutputStream object as follows:

```
DataOutputStream ostream;
```

6. Save the work you have done so far as **CreateEventFile.java** in the Chapter.16 folder on your Student Disk.

Next you will add the CreateEventFile's constructor to the class. The constructor calls its parent's constructor, which is the JFrame class constructor, and passes it a title to use for the JFrame. The constructor also attempts to open an events.dat file for output. If the open fails, the constructor's **catch** block handles the Exception; otherwise you add all the JTextFields, JLabel, and JButton Components to the JFrame.

To write the CreateEventFile class constructor:

1. In the CreateEventFile.java file, press **[Enter]** to start a new line below the statement that declares the DataOutputStream object, type the following constructor header and opening curly brace, and then call the superclass constructor:

```
public CreateEventFile()
{
    super("Create Event File");
```

2. Add the following try...catch block to handle the file creation:

```
try
{
    ostream = new DataOutputStream
        (new FileOutputStream("events.dat"));
}
catch(IOException e)
{
    System.err.println("File not opened");
    System.exit(1);
}
```



Notice the use of the System.err object to display an error message. Alternately, you can display the message on System.out.

3. After the file is open, use the following code to set the JFrame's size, choose a layout manager, and add all the necessary Components to the JFrame:

```
setSize(320,200);
con.setLayout(new FlowLayout());
companyName.setFont(bigFont);
con.add(companyName);
con.add(prompt);
con.add(hLabel);
con.add(host);
con.add(dLabel);
con.add(date);
con.add(gLabel);
con.add(guests);
con.add(enterDataButton);
```

4. To finish the JFrame constructor, enter the following code to register the JFrame as a listener for the JButton, make the JFrame visible, and set the default close operation for the JFrame. Finally, add a closing curly brace for the constructor.

```
enterDataButton.addActionListener(this);
setVisible(true);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
```

5. Save the file. Don't compile the file yet; you will add more code in the next set of steps.

When the users see the JFrame, they can enter data in each of the available JTextFields. When users complete a record for a single event, they click the JButton, which causes the actionPerformed() method to execute. This method must retrieve the text from each of the JTextFields and write it to a data file in the correct format. You will write a usable actionPerformed() method now.

To add the actionPerformed() method to the CreateEventFile program:

1. At the end of the existing code within the CreateEventFile.java file, press **[Enter]** to start a new line below the constructor method, and then type the following header for the actionPerformed() method. Within the method, create an integer variable that will hold the number of guests at an event.

```
public void actionPerformed(ActionEvent e1)
{
    int numGuests;
```

2. Use a **try** block to hold the data retrieval and the subsequent file-writing actions so that you can handle any I/O errors that occur. You will use the parseInt() method to convert the JTextField guest number to a usable integer, but you will accept the host and date fields as simple text. You can use the appropriate DataOutputStream methods to write formatted data to the output file.



You first learned about parseInt() and the Integer class in Chapter 7.

```
try
{
    numGuests = Integer.parseInt(guests.getText());
    ostream.writeUTF(host.getText());
    ostream.writeUTF(date.getText());
    ostream.writeInt(numGuests);
```

3. Continue the **try** block by removing the data from each JTextField after it is written to the file. That way, each JTextField will be clear and ready to receive data for the next record. Notice that to clear the fields, you use a pair of quotes with no space between them. Then end the **try** block.

```
    host.setText("");
    date.setText("");
    guests.setText("");
}
```

4. There are two types of Exceptions that you might want to deal with in this application. Because the host name and date fields are text, the user can enter any type of data. However, the guest field must be an integer. When you use the parseInt() method with data that cannot be converted to an integer (such

as alphabetic letters), a `NumberFormatException` error occurs. In this case, you can write an error message to the standard error device and explain the problem as follows:

```
catch(NumberFormatException e2)
{
    System.err.println("Invalid number of guests");
}
```

5. A second and more serious Exception occurs when the program cannot write the output file, so you should `catch` the potential `IOException`, print an error message, and exit using the following code:

```
catch(IOException e3)
{
    System.err.println("Error writing file");
    System.exit(1);
}
```

6. Add two closing curly braces—one for the `actionPerformed()` method and one for the class. Then save and compile the file.

Next you will create a program that uses the `CreateEventFile` JFrame. The program's only task is to instantiate a `CreateEventFile` JFrame object.

To write a program that uses a `CreateEventFile` JFrame:

1. Open a new file in your text editor, and then type the following `EventFile` program that establishes a `CreateEventFile` object:

```
public class EventFile
{
    public static void main(String[] args)
    {
        CreateEventFile cef = new CreateEventFile();
    }
}
```

2. Save the file as **EventFile.java** in the `Chapter.16` folder on your Student Disk, and then compile it using the `javac` command. When it compiles successfully, run the program, which should look like Figure 16-10.

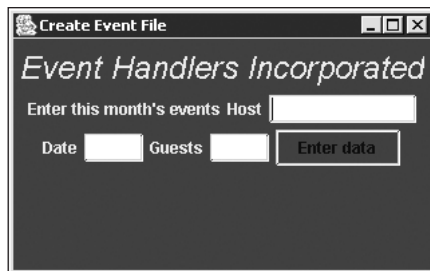


Figure 16-10 User interface for the `EventFile` program

3. Type sample data into the JTextFields in the JFrame. Specifically, type **Sagami** as the event host, on the **3(rd)**, with **150** guests. After entering the data into the three data fields, click the **Enter data** button. Your data is sent to the file, and the fields are cleared. Now you can enter a second record (make up your own record information), and then click the **Enter data** button again. Repeat this process until you have entered five data records. While entering at least one record, type non-numeric data in the guest field. Notice the error message that displays on the standard error device at the command line.
4. Click the **Close** button in the CreateEventFile JFrame to close it.
5. Examine your Student Disk using any file-management program or the DOS command-line directory command, `dir`. Confirm that your program created the `events.dat` data file in the `Chapter.16` folder on your Student Disk. You will write a program to read the file in the next series of steps.

READING FORMATTED FILE DATA

`DataInputStream` objects enable you to read binary data from an `InputStream`. The `DataInput` interface is implemented by `DataInputStream`. The `DataInput` interface includes methods such as `readByte()`, `readChar()`, `readDouble()`, `readFloat()`, `readInt()`, and `readUTF()`. In the same way that the different `write()` methods of `DataOutput` correctly format data you write to a file, each `DataInput read()` method correctly reads the type of data indicated by its name.

When you want to create a `DataInputStream` object that reads from a file, you use the same chaining technique you used for output files. In other words, if you define a `DataInputStream` object as `DataInputStream in;`, then you can associate it with a file when you call its constructor, as in `in = new DataInputStream (FileInputStream("someFile"));`.

When you read data from a file, you need to determine when the end of the file has been reached. Earlier in this chapter, you learned that you can determine EOF by checking for a return value of `-1` from the `read()` method. Alternately, if you attempt each file `read()` from within a `try` block, you can catch an `EOFException`. When you catch an `EOFException`, it means you have reached the end of the file and you should take appropriate action, such as closing the file.



Most Exceptions represent error conditions. An `EOFException` is more truly an “exception” in that most `read()` method calls do not result in EOF. For example, when a file contains 999 records, only the 1,000th, or last, `read()` for a file causes an `EOFException`.

Next you will create a JFrame in which employees of Event Handlers Incorporated can view each individual record stored in the `events.dat` file. The user interface will look like the interface used in the `CreateEventFile` JFrame, but the user will not enter data within

this JFrame. Instead, the user will click a JButton to see each succeeding record in the event.dat file.

To create a JFrame for viewing file data:

1. Open a new file in your text editor, and then type the following first few lines of the ReadEventFile class:

```
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class ReadEventFile extends JFrame
    implements ActionListener
{
```

2. Enter the following code to declare all of the JLabels, JTextFields, and associated values that will appear in the JFrame. The text of the prompt and JButton have changed, but these statements basically echo the statements in the CreateEventFile.java file.

```
private JLabel companyName =
    new JLabel("Event Handlers Incorporated");
Font bigFont = new Font("Helvetica", Font.ITALIC, 24);
private JLabel prompt = new
    JLabel("View this month's events");
private JTextField host = new JTextField(10);
private JTextField date = new JTextField(4);
private JTextField guests = new JTextField(4);
private JButton viewEventButton = new
    JButton("View Event");
private JLabel hLabel = new JLabel("Host");
private JLabel dLabel = new JLabel("Date");
private JLabel gLabel = new JLabel("Guests");
private Container con = getContentPane();
```

3. Enter the following code to declare a DataInputStream object. Then write the ReadEventFile constructor method that uses a `try...catch` block to open a file. Notice that you can chain the DataInputStream object and a FileInputStream object using the same technique you used for output. (*Note:* If you have stored your event.dat file in a location other than A:\Chapter.16, then change the location in the FileInputStream constructor in your own program.)

```
DataInputStream istream;
public ReadEventFile()
{
    super("Read Event File");
    try
    {
```

```

        istream = new DataInputStream
            (new FileInputStream
                ("A:\\Chapter.16\\events.dat"));
    }
    catch(IOException e)
    {
        System.err.println("File not opened");
        System.exit(1);
    }

```

4. After successfully opening the file, set the JFrame size, layout manager, and Font for the JFrame as follows:

```

setSize(325,200);
con.setLayout(new FlowLayout());
companyName.setFont(bigFont);

```

5. Add the JFrame's Components as follows:

```

con.add(companyName);
con.add(prompt);
con.add(hLabel);
con.add(host);
con.add(dLabel);
con.add(date);
con.add(gLabel);
con.add(guests);
con.add(viewEventButton);

```

6. Enter the following code to ensure that the JFrame listens for JButton messages, to make the JFrame visible, and to set the default close operation:

```

viewEventButton.addActionListener(this);
setVisible(true);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

```

7. Add the closing curly brace for the ReadEventFile constructor.

8. Type the beginning of the following actionPerformed() method. This method declares variables for the file field data, and then uses a try block to call the appropriate read() method for each field. Each data field then appears in the correct JTextField.

```

public void actionPerformed(ActionEvent e1)
{
    String theHost, theDate;
    int numGuests;
    try
    {
        theHost = istream.readUTF();
        theDate = istream.readUTF();
        numGuests = istream.readInt();
    }

```

```

        host.setText(theHost);
        date.setText(theDate);
        guests.setText(String.valueOf(numGuests));
    }

```

9. Code the following two `catch` blocks for the `try` block that reads the data fields. The first `catch` block catches the `EOFException` and calls a `closeFile()` method. The second `catch` block catches `IOExceptions` and exits the program if there is a problem with the file. Notice that the Exceptions have unique names (`e2` and `e3`) because you cannot declare two data items with the same name within the same method.

```

catch(EOFException e2)
{
    closeFile();
}
catch(IOException e3)
{
    System.err.println("Error reading file");
    System.exit(1);
}

```

10. Add the closing curly brace for the `actionPerformed()` method.
11. Write the following `closeFile()` method that closes the `DataInputStream` object and exits the program:

```

public void closeFile()
{
    try
    {
        istream.close();
        System.exit(0);
    }
    catch(IOException e)
    {
        System.err.println("Error closing file");
        System.exit(1);
    }
}

```

12. Add the closing curly brace for the class.
13. Save the file as **ReadEventFile.java** in the Chapter.16 folder on your Student Disk, and then compile the program using the **javac** command.

Next you will write a short host program that creates a `ReadEventFile` `JFrame` so Event Handlers employees can examine the existing file of scheduled events.

To write a program that creates a JFrame and displays file data:

1. Open a new file in your text editor, and then type the following program that instantiates a ReadEventFile object:

```
import java.io.*;
public class EventFile2
{
    public static void main(String[] args)
    {
        ReadEventFile ref = new ReadEventFile();
    }
}
```

2. Save the file as **EventFile2.java** in the Chapter.16 folder on your Student Disk, compile and execute the program, and then click the **View Event** button. Figure 16-11 shows the interface with the first record from the events.dat data file visible. (Your data might differ.) Click the **View Event** button again to see the second record. Continue clicking the **View Event** button until you reach the end of the file; when you do, the JFrame closes and the program ends.



Figure 16-11 Output of the EventFile2 program

USING A VARIABLE FILENAME

A program that reads a data file and displays its contents for you is useful. A program that can read any data file, regardless of what you name it, is even more useful. Suppose Event Handlers Incorporated keeps different event files for each month. When you execute the EventFile2 program, it would be convenient to name the file you want to display. For example, your command line might be `java EventFile2 eventsApril.dat` or `java EventFile2 October.dat`. The same program would execute, but use the data file that corresponds to the requested month. Next you will modify the EventFile2 program to accept a variable filename from the user.

To modify the EventFile2 program to use a variable filename:

1. Open the **ReadEventFile.java** file and immediately save it as **ReadNamedFile.java**. Be sure to change the class name to match.

2. Change the `ReadEventFile` constructor to the new class name and add an argument that is a `String`. The value for this argument will originate from the first element in the `String` array (`String[] args`) that you have included in every `main()` method you have written.

```
public ReadNamedFile(String fileName)
```

3. Within the `try` block of the `ReadNamedFile` constructor, remove the reference to the filename `events.dat`, and replace it with the variable `filename` that represents the `String` you will use when you call this constructor. In other words, change the `istream` assignment to the following:

```
istream = new DataInputStream
    (new FileInputStream(fileName));
```

4. Save the file, and then compile the class.
5. Open a new file in your text editor and type the following program that creates an instance of the `ReadNamedFile` class. As the argument to the `ReadNamedFile` constructor, use the first (and only) `String` in the array of `String` arguments that you will pass to the `main()` method from the command line.

```
import java.io.*;
public class EventFile3
{
    public static void main(String[] args)
    {
        ReadNamedFile ref = new ReadNamedFile(args[0]);
    }
}
```

6. Save the file as **EventFile3.java** in the Chapter.16 folder on your Student Disk, and compile the program.
7. You will test the program using the existing file `event.dat`. To execute the program, at the command line type `java EventFile3 events.dat`. The program executes, correctly showing you the events in the file as before. One at a time, view each of the records in the file just like you did in the last set of steps.
8. Attempt to execute the program using a nonexistent filename, for example `java EventFile3 noSuchFile.dat`. You should see the message “File not opened”.

CREATING RANDOM ACCESS FILES

The files you wrote to and read from in this chapter are **sequential access files**, which means that you access the records in sequential order from beginning to end. For example, if you wrote an Event record with host name Adams, and then you created an Event record with host name Brown, when you retrieve the records you see that they remain

in the original data-entry order. Businesses store data in sequential order when they use the records for **batch processing**, or processing that involves performing the same tasks with many records, one after the other. For example, when a company produces paychecks, the records for the pay period are gathered in a batch and the checks are calculated and printed in sequence.

For many applications, sequential access is inefficient. These applications, known as **real-time** applications, require that a record be accessed immediately while a client is waiting. For example, if a customer telephones a department store with a question about a monthly bill, the customer service representative does not need to access every customer account in sequence. With tens of thousands of account records to read, it would take too long to access the customer's record. Instead, customer service representatives require **random access files**, files in which records can be located in any order. Because they enable you to locate a particular record directly (without reading all of the preceding records), random access files are also called **direct access files**. You can use Java's `RandomAccessFile` class to create your own random access files.

The `RandomAccessFile` class contains the same `read()`, `write()`, and `close()` methods as `InputStream` and `OutputStream`, but it also contains a `seek()` method that lets you select a beginning position within a file before you read or write data. For example, if you declare a `RandomAccessFile` object named `myFile`, then the statement `myFile.seek(200);` selects the 200th position within the file. The 200th position represents the 201st byte because, as with Java arrays, the numbering of file positions begins at zero. The next `read()` or `write()` method will operate from the newly selected starting point.

When you store records in a file, it is often more useful to be able to access the 200th record, rather than the 201st byte. In this case, you multiply each record's size by the position you want to access. For example, if you store records that are 50 bytes long, you can access the n^{th} record using the statement `myFile.seek((n-1) * 50);`.

When you declare a `RandomAccessFile` object, you include a filename as you do with other file objects. You also include `r` or `rw` within double quotation marks as a second argument to indicate that the file is open for reading only ("`r`"), or for both reading and writing ("`rw`"). For example, `RandomAccessFile myFile = new RandomAccessFile("C:\\Temp\\someData.dat", "rw");` opens the `someData.dat` file so that either the `read()` or `write()` method can be used on the file. This feature is particularly useful in random access processing. Consider a business with 20,000 customer accounts. When the customer who has the 14,607th record in the file acquires a new telephone number, it is convenient to directly access the 14,607th record, the `read()` method confirms it represents the correct customer, and then the `write()` method writes the new telephone number to the file in the location the old telephone number was stored previously. You will demonstrate the `seek()` method in the next set of steps.

To show how the seek() method works:

1. Open a new file in your text editor, and then type the following first few lines of the `AccessRandomly` class:

```
import java.io.*;
public class AccessRandomly
{
    public static void main(String[] args) throws
        IOException
    {
```

2. Enter the following code to declare an `OutputStream` object for output, an integer to hold data temporarily, and a new `RandomAccessFile`. You will use the data file for reading only, so you include “r” as the second argument in the `RandomAccessFile` constructor.

```
OutputStream ostream;
int c;
RandomAccessFile inFile =
    new RandomAccessFile("datafile.dat", "r");
```

3. Enter the following code to assign `ostream` as the standard output device. Then try accessing the 10th file position (which is represented by the number 9). Read this position, and then display its contents.

```
ostream = System.out;
try
{
    inFile.seek(9);
    c = inFile.read();
    System.out.print("The character in position 9 is ");
    ostream.write(c);
}
```

4. Add the following `catch` clause that executes if there is trouble accessing the file, and then add a `finally` block to close the files:

```
catch(IOException e)
{
    System.out.println("Error: " + e.getMessage());
}
finally
{
    inFile.close();
    ostream.close();
}
```

5. Add two closing curly braces—one for the `main()` method and one for the class.
6. Save the file as **AccessRandomly.java** in the Chapter.16 folder on your Student Disk, and then compile it using the **javac** command.

7. Before you run the file, create a datafile.dat file by running the ReadKBWriteFile program you created earlier in this chapter. At the command line, type **java ReadKBWriteFile**, and then press **[Enter]**.
8. At the prompt, type **This is my random access file**. Press **[Enter]**, press **[Ctrl]+Z**, and then press **[Enter]** to end the program. (The *y* in *my* is the 10th character you type.)
9. At the command line, run the AccessRandomly program by typing **java AccessRandomly**. The output is the tenth character in the file, as shown in Figure 16-12.



```

A:\Chapter.16>java AccessRandomly
The character in position 9 is y
A:\Chapter.16>_

```

Figure 16-12 Output of the AccessRandomly program

In the AccessRandomly program, only one read() command was issued, yet the program accessed a byte nine positions into the file. When you access a file randomly, you do not read all the data that precedes the data you are seeking. Accessing data randomly is one of the major features that makes large data systems maintainable.

CHAPTER SUMMARY

- Files are objects that you store on permanent storage devices, such as floppy disks, CD-ROMs, or external drives. You can use the File class to gather file information.
- Data used by businesses generally is stored in a data hierarchy that includes files, records, fields, and characters.
- Java views a file as a series of bytes, and a stream as an object through which input and output data (in the form of bytes) flow. InputStream and OutputStream are abstract subclasses of Object that contain methods for performing input and output. FileInputStream and FileOutputStream provide the capability to read from and write to files. You can use the InputStream read() method to read in one character at a time.
- You can use the DataInputStream and DataOutputStream classes to accomplish formatted input and output. The DataOutput interface includes methods such as

writeBoolean(), writeChar(), writeDouble(), writeFloat(), and writeInt(). Each method writes data in the correct format for the data type its name indicates. You can use the method writeUTF() to write Unicode format strings.

- DataInputStream objects enable you to read binary data from an InputStream. The DataInput interface includes methods such as readByte(), readChar(), readDouble(), readFloat(), readInt(), and readUTF(). Each DataInput read() method correctly reads the type of data indicated by its name, such as readByte(), readChar(), or readDouble().
- You can provide a variable filename to a program using the command line.
- Random access files, or direct access files, are files in which records can be located in any order. The RandomAccessFile class contains the same read(), write(), and close() methods as InputStream and OutputStream, but it also contains a seek() method that lets you select a beginning position within a file before you read or write.

REVIEW QUESTIONS

1. Files always _____.
 - a. hold software instructions
 - b. occupy a section of storage space
 - c. remain open during the execution of a program
 - d. all of the above
2. The File class enables you to _____.
 - a. open a file
 - b. close a file
 - c. determine a file's size
 - d. all of the above
3. The _____ package contains all the classes you use in file processing.
 - a. java.file
 - b. java.io
 - c. java.lang
 - d. java.process
4. The statement `File aFile = new File("myFile");` creates a file _____.
 - a. on the disk in drive A
 - b. on the hard drive (drive C)
 - c. in the Temp folder on the hard drive (drive C)
 - d. on the default disk drive

5. The File method `canWrite()` returns a(n) _____ value.
 - a. `int`
 - b. `Boolean`
 - c. `Object`
 - d. `void`
6. Data used by businesses is stored in a data hierarchy that includes the following items, from largest to smallest:
 - a. file, field, record, character
 - b. record, file, field, character
 - c. file, record, field, character
 - d. record, field, file, character
7. A group of characters that has meaning is a _____.
 - a. file
 - b. record
 - c. field
 - d. byte
8. Files consist of related _____.
 - a. records
 - b. fields
 - c. data segments
 - d. archives
9. Before a program can use a data file, the program must _____ the file.
 - a. create
 - b. open
 - c. store
 - d. close
10. When you perform an input operation in a Java program, you use a _____.
 - a. pipeline
 - b. channel
 - c. moderator
 - d. stream

11. Most streams flow _____.
 - a. in
 - b. out
 - c. either in or out, but only in one direction
 - d. both in and out concurrently
12. The output from `System.err` and `System.out` _____ go to the same device.
 - a. must
 - b. cannot
 - c. might
 - d. might on a mainframe system, but never would on a PC
13. A small memory location that is used to temporarily hold data is a _____.
 - a. stream
 - b. buffer
 - c. bulwark
 - d. channel
14. The `read()` method returns a value of -1 when it encounters a(n) _____.
 - a. input error
 - b. integer
 - c. end-of-file condition
 - d. negative value
15. Much of the data that you write with `DataOutputStream` objects is not readable in a text editor because _____.
 - a. it does not exist in any physical sense
 - b. it is stored in a non-character format
 - c. you can read it only with a special piece of hardware called a Data Reader
 - d. Java's security features prohibit it
16. You use a `DataOutputStream` connected to `FileOutputStream` by using a method known as _____.
 - a. sequencing
 - b. iteration
 - c. piggybacking
 - d. chaining

17. When you **catch** an EOFException, it means you have _____.
 - a. failed to find the end of the file
 - b. forgotten to open a file
 - c. forgotten to close a file
 - d. reached the end of a file
18. Which of the following applications is most likely to use random file processing?
 - a. a program that schedules airline reservations
 - b. a credit card company's end-of-month billing program
 - c. a college's program that lists honor students at the end of each semester
 - d. a manufacturing company's quarterly inventory reporting system
19. The method that the RandomAccessFile class contains that does not exist in the InputStream class is _____.
 - a. read()
 - b. close()
 - c. seek()
 - d. delete()
20. You can open a RandomAccessFile object for _____.
 - a. reading
 - b. writing
 - c. both of the above
 - d. none of the above

EXERCISES

1. Create a file using any word-processing program or text editor. Write a program that displays the file's name, parent, size, and time of last modification. Save the program as **FileStatistics.java** in the Chapter.16 folder on your Student Disk.
2. Create two files using any word-processing program or text editor. Write a program that determines whether the two files are located in the same folder. Save the program as **SameFolder.java** in the Chapter.16 folder on your Student Disk.
3. Write a program that determines which, if any, of the following files are stored in the Chapter.16 folder of your Student Disk: autoexec.bat, SameFolder.java, Chap16ReadEventFile.class, and Hello.java. Save the program as **FindSelectedFiles.java** in the Chapter.16 folder on your Student Disk.
4. a. Create a JFrame that allows the user to enter a series of friends' names and phone numbers and creates a file from the entered data. Save the JFrame as **CreatePhoneList.java** in the Chapter.16 folder on your Student Disk. To use

the JFrame, create a program named **UsePhoneList.java** that instantiates a **CreatePhoneList** object.

- b. Write a program that reads the file created by the **UsePhoneList** program and displays one record at a time in a JFrame. Save the JFrame as **ReadPhoneList.java** in the Chapter.16 folder on your Student Disk. Save the program that uses this JFrame as **UsePhoneList2.java**.
5. a. Write a program for a mail-order company. The program uses a data-entry screen in which the user types an item number and a quantity. Write each record to a file. Save the program as **MailOrderWrite.java** in the Chapter.16 folder on your Student Disk.
- b. Write a program that reads the data file created by the **MailOrderWrite** program and displays one record at a time on the screen. Save the program as **MailOrderRead.java** in the Chapter.16 folder on your Student Disk.
6. a. Write a program for a mail-order company. The program uses a data-entry screen in which the user types an item number and a quantity. The valid item numbers and prices are as follows:

Item Number	Price (\$)
101	4.59
103	29.95
107	36.50
125	49.99

When the user enters an item number, check the number to make sure that it is valid. If it is valid, write a record that includes item number, quantity, price each, and total price. Save the program as **MailOrderWrite2.java** in the Chapter.16 folder on your Student Disk.

- b. Write a program that reads the data file created by the **MailOrderWrite2** program and displays one record at a time on the screen. Save the program as **MailOrderRead2.java** in the Chapter.16 folder on your Student Disk.
7. a. Write a program that allows a user to enter an integer representing a file position. Access the requested position within a file and display the character there. Save the program as **SeekPosition.java** in the Chapter.16 folder on your Student Disk.
- b. Modify the program created by the **SeekPosition** program so that you display the next five characters after the requested position. Save the program as **Seek2.java** in the Chapter.16 folder on your Student Disk.
8. a. Write a program that creates a JFrame with text fields for order processing for a tee-shirt manufacturer. Include **JTextFields** for size, color, and slogan. Write each complete record to a file. Save the JFrame as **TeeShirtWrite.java** in the Chapter.16 folder on your Student Disk. Write a program named **UseTeeShirt.java** that displays the JFrame.

- b. Write a program that reads the data file created by the TeeShirtWrite program and displays one record at a time in a JFrame on the screen. Save the program as **TeeShirtRead.java** in the Chapter.16 folder on your Student Disk. Write a program named **UseTeeShirt2.java** that displays the JFrame.
9. Write a program that allows the user to type any number of characters into a file. Then display the file contents backward. Save the program as **ReadBackwards.java** in the Chapter.16 folder on your Student Disk.
10. a. Create a JFrame that allows you to enter student data—ID number, last name, and first name. Include two buttons and instruct the user to click “Grad” or “Undergrad” after entering the data for each student. Depending on the user’s choice, write the data to either a file containing graduate students or a separate file containing undergraduate students. Name the JFrame **GradAndUndergrad.java** and create a program named **UseGradAndUndergrad.java** that instantiates a JFrame object. Save both files in the Chapter.16 folder of your Student Disk.
 - b. Create a JFrame named **StudentRead.java** that, in turn, accesses all the records in the graduate and the undergraduate files created in the GradAndUndergrad program. Write a program named **UseStudentRead.java** that instantiates a JFrame object. Save both files in the Chapter.16 folder of your Student Disk.
11. Each of the following files in the Chapter.16 folder on your Student Disk has syntax and/or logical errors. In each case, determine the problem and fix the program. Notice that DebugSixteen3.java and DebugSixteen4.java have companion files that are part of each exercise. After you correct the errors, save each file using the same filename preceded with Fix. For example, DebugSixteen1.java will become FixDebugSixteen1.java.
 - a. DebugSixteen1.java
 - b. DebugSixteen2.java
 - c. DebugSixteen3.java
 - d. DebugSixteen4.java

CASE PROJECT



Create a data-entry and retrieval system for Mowers Inc., a lawn-mowing service. Use a JFrame to enter data for the customer’s name and lawn size in square feet. An output file holds the customer name, lawn size, and fee per mowing—\$50 for lawns under 1,000 square feet and \$75 for lawns 1,000 square feet or more. Name the JFrame **MowersInc.java** and save it in the Chapter.16 folder of your Student Disk. Name the client program that creates an instance of the JFrame **UseMowersInc.java**. Retrieve the created records with a JFrame named **MowersInc2.java** and a client program named **UseMowersInc2.java**.

