Streams, File I/O, and Networking 10

I'll note you in my book of memory.

—WILLIAM SHAKESPEARE, *Henry VI, Part II*

I/O refers to program input and output. Input can be taken from the keyboard or from a file. Similarly, output can be sent to the screen or to a file. In this chapter, we explain how you can write your programs to take input from a file and send output to a file. By doing so, you will be able to retain your data and objects long after your program ends its execution.

OBJECTIVES

After studying this chapter, you should be able to

- Describe the concept of an I/O stream
- Explain the difference between text files and binary files
- Save data, including objects, in a file
- Read data, including objects, from a file
- Transmit data over a network

PREREQUISITES

You will need some knowledge of exception handling, as described in Chapter 9, to understand this chapter. Having some knowledge of inheritance will also be helpful. Only Section 10.5 and the Case Study in Section 10.3 of this chapter actually require that you know about arrays, interfaces, and inheritance (covered in Chapters 7 and 8). You can, of course, skip this section until you learn about these topics. You may cover Section 10.3 after Sections 10.4 and 10.5 if you wish. Many readers may choose to skip the coverage of binary files. Only selected Programming Projects in the rest of this book require knowledge of anything covered in this chapter.

Details of the prerequisites are as follows:

Section	Prerequisite
10.1 An Overview of Streams and File I/O	Chapters 1 through 6.
10.2 Text-File I/O	Sections 9.1 and 10.1. Also some knowledge of inheritance from Chapter 8.
10.3 Techniques for Any File	Section 10.2 and Chapter7 for the Case Study.
10.4 Basic Binary-File I/O	Sections 9.2 and 10.2. You do not need Section 10.3.

10.5 Binary-File I/O with Objects and Arrays	Section 10.4 and Chapters 7 and 8. You do not need Section 10.3.
10.6 Graphics Supplement	Sections 9.4 and 10.3.
10.7 Network Communication with Streams	Sections 9.1, 10.1, and 10.2.

10.1 AN OVERVIEW OF STREAMS AND FILE I/O

Fish say, they have their Stream and Pond,

But is there anything Beyond?

—RUPERT BROOKE, *Heaven*

In this section, we give you a general introduction to file I/O. In particular, we explain the difference between text files and binary files. The Java syntax for file I/O statements is given in subsequent sections of this chapter.

The Concept of a Stream

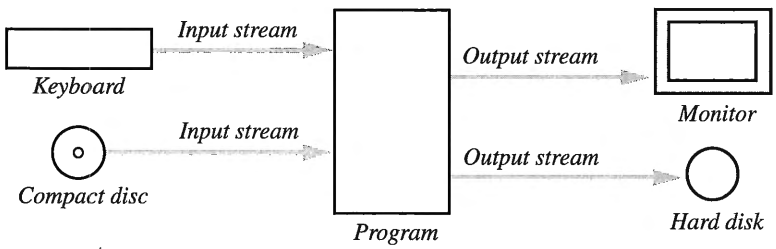
Files can store programs, music, pictures, video, and so on

You are already using files to store your Java classes and programs, your music, your pictures, and your videos. You can also use files to store input for a program or to hold output from a program. In Java, file I/O, as well as simple keyboard and screen I/O, is handled by streams. A **stream** is a flow of data. The data might be characters, numbers, or bytes consisting of binary digits. If the data flows *into your program*, the stream is called an **input stream.**If the data flows *out of your program*, the stream is called an **output stream.** For example, if an input stream is connected to the keyboard, the data flows from the keyboard into your program. If an input stream is connected to a file, the data flows from the file into your program. Figure 10.1 illustrates some of these streams.

A stream is a flow of data into or out of a program

In Java, streams are implemented as objects of special stream classes. Objects of the class `Scanner`, which we have been using for keyboard input, are input streams. The object `System.out` is an example of an output stream

FIGURE 10.1 Input and Output Streams



that we have also used. In this chapter, we discuss streams that connect your program to files instead of to the keyboard or display.

RECAP Streams

A stream is an object that either

- Delivers data from your program to a destination, such as a file or the screen, or
- Takes data from a source, such as a file or the keyboard, and delivers the data to your program.

Why Use Files for I/O?

Data in a file remains after program execution ends

The keyboard input and screen output we have used so far deal with temporary data. When the program ends, the data typed at the keyboard and left on the screen go away. Files provide you with a way to store data permanently. The contents of a file remain until a person or program changes the file.

An input file can be used over and over again by different programs, without the need to type the data again for each program. Files also provide you with a convenient way to deal with large quantities of data. When your program takes its input from a large input file, it receives a lot of data without user effort.

Text Files and Binary Files

All of the data in any file is stored as binary digits (bits)—that is, as a long sequence of 0s and 1s. However, in some situations, we do not think of a file's contents as a sequence of binary digits. Instead, we think of them as a sequence of characters. Files that are thought of as sequences of characters, and that have streams and methods to make the binary digits look like characters to your program and your text editor, are called **text files.** All other files are called **binary files.** Each kind of file has its own streams and methods to process them.

Two kinds of files: text files and binary files

Your Java programs are stored in text files. Your music files and picture files are binary files. Since text files are sequences of characters, they usually appear the same on all computers, so you can move your text files from one computer to another with few or no problems. The contents of binary files are often based on numbers. The structure of some binary files is standardized so that they can also be used on a variety of platforms. Many picture files and music files fall into this category.

Java programs can create or read both text files and binary files. Writing a text file and writing a binary file require similar steps. Likewise, reading a text file is similar to reading a binary file. The kind of file, however, determines which classes we use to perform the input and output.

The one big advantage of text files is that you can create, look at, and edit them by using a text editor. You have already done this when writing a Java program. With binary files, all the reading and writing must normally be done by a special program. Some binary files must be read by the same type of computer and with the same programming language that originally created them. However, Java binary files are platform independent; that is, with Java, you can move your binary files from one type of computer to another, and your Java programs will still be able to read the binary files.

Each character in a text file is represented as 1 or 2 bytes, depending on whether the system uses ASCII or Unicode. When a program writes a value to a text file, the number of characters written is the same as if they were written to a display using `System.out.println`. For example, writing the `int` value 12345 to a text file places five characters in the file, as shown in Figure 10.2. In general, writing an integer places between 1 and 11 characters in a text file.

Binary files store all values of the same primitive data in the same format. Each is stored as a sequence of the same number of bytes. For example, all `int` values occupy 4 bytes each in a binary file, as Figure 10.2 also illustrates. A Java program interprets these bytes in very much the same way that it interprets a data item, such as an integer, in the computer's main memory. That is why binary files can be handled so efficiently.

FAQ Should I use a text file or a binary file?

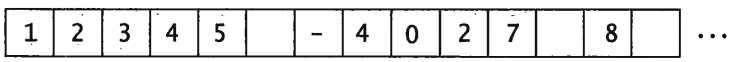
Use a text file if you want a text editor to either create a file that a program will read or read a file that a program created. In other cases, consider a binary file, as it usually occupies less space.

REMEMBER Input and Output Terminology

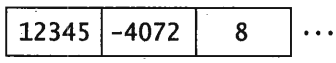
The word *input* means that data moves into your program, not into the file. The word *output* means that data moves out of your program, not out of the file.

FIGURE 10.2A Text File and a Binary File Containing the Same Values

A text file



A binary file



SELF-TEST QUESTIONS

1. Why would anybody write a program that sends its output to a file instead of to the screen?
2. When we discuss input, are we referring to data moving from the program to a file or from a file to the program?
3. What is the difference between a text file and a binary file?

10.2 TEXT-FILE I/O

Proper words in proper places,

make the true definition of a style.

—JONATHAN SWIFT, Letter to a Young Clergyman (January 9,1720)

In this section, we give a description of the most common ways to perform text-file I/O in Java.

Creating a Text File

The class `PrintWriter` in the Java Class Library defines the methods that we will need to create and write to a text file. This class is the preferred one for writing to a text file. It is in the package `java.io`, so we will need to begin our program with an `import` statement. Actually, we will be using other classes as well, so we will import them also, as you will see soon.

Opening a file connects it to a stream

Before we can write to a text file, we must connect it to an output stream. That is, we **open** the file. To do this, we need the name of the file as a string. The file has a name like `out.txt` that the operating system uses. We also must declare a variable that we will use to reference the stream associated with the actual file. This variable is called the **stream variable.** Its data type in this case is `PrintWriter`. We open a text file for output by invoking `PrintWriter`'s constructor and passing it the file name as its argument. Since this action can throw an exception, we must place the call to the constructor within a `try` block.

The following statements will open the text file `out.txt` for output:

```
String fileName = "out.txt";//Could read file name from user
PrintWriter outputStream = null;
try
{
    outputStream = new PrintWriter(fileName);
}
catch(FileNotFoundException e)
{
    System.out.println("Error opening the file " + fileName);
    System.exit(0);
}
```

The class `FileNotFoundException` must also be imported from the package `java.io`.

File name versus stream variable

Note that the name of the file, in this case `out.txt`, is given as a `String` value. Generally, we would read the name of the file instead of using a literal. We pass the file name as an argument to the `PrintWriter` constructor. The result is an object of the class `PrintWriter`, which we assign to our stream variable `outputStream`.

When you connect a file to an output stream in this way, your program always starts with an empty file. If the file `out.txt` already exists, its old contents will be lost. If the file `out.txt` does not exist, a new, empty file named `out.txt` will be created.

Because the `PrintWriter` constructor might throw a `FileNotFoundException` while attempting to open the file, its invocation must appear within a `try` block. Any exception is caught in a `catch` block. If the constructor throws an exception, it does not necessarily mean that the file was not found. After all, if you are creating a new file, it doesn't already exist. In that case, an exception would mean that the file could not be created because, for example, the file name was already being used for a folder (directory) name.

After we open the file—that is, connect the file to the stream—we can write data to it. The method `println` of the class `PrintWriter` works the same for writing to a textfile as the method `System.out.println` works for writing to the screen. `PrintWriter` also has the method `print`, which behaves just like `System.out.print`, except that the output goes to a text file.

Now that the file is open, we always refer to it by using the stream variable, not its name. Our stream variable `outStream` references the output stream—that is, the `PrintWriter` object—that we created, so we will use it when we invoke `println`. Notice that `outStream` is declared outside of the `try` block so that it is available outside of the block. Remember, anything declared in a block—even within a `try` block—is local to the block.

Let's write a couple of lines to the text file. The following statements would come after the previous statements that opened the file:

```
outputStream.println("This is line 1.");
outputStream.println("Here is line 2.");
```

Rather than sending output to a file immediately, `PrintWriter` waits to send a larger packet of data. Thus, the output from a `println` statement, for example, is not sent to the output file right away. Instead, it is saved and placed into an area of memory called a **buffer**, along with the output from other invocations of `print` and `println`. When the buffer is too full to accept more output, its contents are written to the file. Thus, the output from several `println` statements is written at the same time, instead of each time a `println` executes. This technique is called **buffering**; it allows faster file processing.

After we finish writing the entire text file, we disconnect the stream from the file. That is, we **close** the stream connected to the file by writing

Closing a file disconnects it from a stream

```
outputStream.close();
```

Closing the stream causes the system to release any resources used to connect the stream to the file and to do some other housekeeping. If you do not close a stream, Java will close it for you when the program ends. However, it is safest to close the stream by explicitly calling the method `close`. Recall that when you write data to a file, the data might not immediately reach its destination. Closing an output stream forces any pending output to be written to the file. If you do not close the stream and your program terminates abnormally, Java might not be able to close it for you, and data can be lost. The sooner you close a stream, the less likely it is that this will happen.

If your program writes to a file and later reads from the same file, it must close the stream after it is through writing to the file and then reopen the file for reading. (Java does have a class that allows a file to be open for both reading and writing, but we will not cover that class in this book.) Note that all stream classes, such as `PrintWriter`, include a method named `close`.

The calls to `println` and `close` need not be within a `try` block, as they do not throw exceptions that must be caught. Listing 10.1 contains a simple but complete program that creates a text file from data read from the user. Notice that the lines shown in the resulting text file look just as they would if we wrote them to the screen.We can read this file either by using a text editor or by using another Java program that we will write a little later.

LISTING 10.1 Writing Output to a Text File *(part 1 of 2)*

```
import java.io.PrintWriter;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class TextFileOutputDemo
{
    public static void main(String[] args)
    {
        String fileName = "out.txt"; //The name could be read from
                                     //the keyboard.
        PrintWriter outputStream = null;
        try
        {
            outputStream = new PrintWriter(fileName};
        }
        catch(FileNotFoundException e)
        {
            System.out.println("Error opening the file" +
                               fileName);
            System.exit(0);
        }
```

(continued)

LISTING 10.1 Writing Output to a Text File *(part 2 of 2)*

```
        System.out.println("Enter three lines of text:");
        Scanner keyboard = new Scanner(System.in);
        for (int count = 1; count <= 3; count++)
        {
            String line = keyboard.nextLine();
            outputStream.println(count + " " + line);
        }
        outputStream.close();
        System.out.println("Those lines were written to " +
                           fileName);
    }
}
```

Sample Screen Output

```
Enter three lines of text:
A tall tree
in a short forest is like
a big fish in a small pond.
Those lines were written to out.txt
```

Resulting file

```
1 A tall tree
2 in a short forest is like
3 a big fish in a small pond.
```

You can use a text editor to read this file.

RECAP Creating a Text File

SYNTAX

```
// Open the file PrintWriter
Output_Stream_Name = null;
try
{
    Output_Stream_Name = new PrintWriter(File_Name);
}
catch(FileNotFoundException e)
{
    Statements_Dealing_With_The_Exception
}
```

(continued)

```
// Write the file using statements of either or both of the
// following forms:
Output_Stream_Name.println(...);
Output_Stream_Name.print(...);
// Close the file
Output_Stream_Name.close();
```

EXAMPLE

See Listing 10.1.

PROGRAMMING TIP A Program Should Not Be Silent

A program that creates a file should inform the user when it has finished writing the file. Otherwise, you will have written a **silent program**, and the user will wonder whether the program has succeeded or has encountered a problem. This advice applies to both text files and binary files.

REMEMBER A File Has Two Names in a Program

Every file used by a program, whether for input or for output, appears to have two names: the actual file name used by the operating system and the name of the stream that is connected to the file. The file name is used when connecting the file to a stream. The stream name is used thereafter to work with the file. The stream name does not exist after the program ends its execution, but the actual file name persists. Note that, since a stream variable can have aliases, a file actually can have more than two names. However, distinguishing between a file name and a stream name is what is important here.

FAQ What are the rules for naming files?

The rules for how you spell file names depend on your operating system, not on Java. When you pass a file name to a Java constructor for a stream, you are not giving the constructor a Java identifier. You are giving the constructor a string corresponding to the file name. Most common operating systems allow you to use letters, digits, and the dot symbol when spelling file names. Many operating systems allow other characters as well, but letters, digits, and the dot symbol are enough for most purposes. A suffix, such as `.txt` in `out.txt`, has no special meaning to a Java program. We are using the suffix to indicate a text file, but that is just a common convention. You can use any file names that are allowed by your operating system, but you should be aware that some operating systems will hide the suffix by default.

GOTCHA A try Block Is a Block

Look again at the program in Listing 10.1. It is not an accident or a minor stylistic concern that caused us to declare the variable `outputStream` outside the `try` block. Suppose we were to move that declaration inside the `try` block, like this:

```
try
{
    PrintWriter outputStream = new PrintWriter(fileName);
}
```

This replacement looks innocent enough, but it makes `outputStream` a local variable in the `try` block. Thus, we would not be able to use `outputStream` outside the `try` block. If we did, we would get an error message saying that `outputStream` is an undefined identifier. ■

Chapter 8 suggested that you define a method `toString` in your classes. This method returns a string representation of the data in an instance of the class. The methods `print` and `println` of `System.out` invoke `toString` automatically when given an object as an argument. This is also true for `print` and `println` of an object of `PrintWriter`.

For example, we could add a `toString` method to the class `Species` in Chapter 5. Recall that this class defines three instance variables: `name`, `population`, and `growthRate`. Thus, we could define `toString`, as follows:

Overriding `toString` in the class `Species`

```
public String toString()
{
    return ("Name = " + name + "\n" +
            "Population = " + population + "\n" +
            "Growth rate = " + growthRate + "%");
}
```

Now, if we write the statements

```
Species oneRecord = new Species("Calif. Condor", 27, 0.02);
System.out.println(oneRecord.toString());
```

in a program, they will produce the output

```
Name = Calif. Condor
Population = 27
Growth rate = 0.02%
```

We also know that

```
System.out.println(oneRecord);
```

will invoke automatically, and so it will produce the same output.

The same is true if we were to write to a text file. Either of the statements

```
outputStream.println(oneRecord.toString());
```

or

```
outputStream.println(oneRecord);
```

will write the previously shown output to the text file attached to the stream `outputStream`.

Extra code on the Web

The program in the file `TextFileSpeciesOutputDemo`, which is in the source code available on the book's Web site, illustrates this fact.

PROGRAMMING TIP Define the Method `toString` for Your Classes

Since the methods `print` and `println` automatically invoke the method `toString`, regardless of whether they belong to the object `System.out` or to an object of an output stream, it is a good idea to define `toString` for your classes. ■

GOTCHA Overwriting a File

When you open a text file or a binary file for output, you always begin with an empty file. If no existing file has the given name, the constructor that you invoke will create an empty file with that name. But if an existing file has the given name, all of its data will be eliminated. Any new output will be written to this existing file. Section 10.3 shows how to test for the existence of a file so that you can avoid accidentally overwriting it. ■

Appending to a Text File

Appending, or adding, data to the end of a file

The way we opened a text file for output in Listing 10.1 ensures that we always begin with an empty file. If the named file already exists, its old contents will be lost. Sometimes, however, that is not what we want. We may simply want to add, or **append**, more data to the end of the file. To append program output to the text file named by the `String` variable `fileName`, we would connect the file to the stream `outputStream` as follows:

```
outputStream =
     new PrintWriter(new FileOutputStream(fileName, true));
```

Since `PrintWriter` does not have an appropriate constructor for this task, we need some help from the class `FileOutputStream`, which we must import from the package java.io. The second argument (`true`) of `FileOutputStream`'s constructor indicates that we want to add data to the file, if it exists already. So if the file already exists, its old contents will remain, and the program's output will be placed after those contents. But if the file does not already exist, Java will create an empty file of that name and append the output to the end of this empty file. The effect in this case is the same as in Listing 10.1.

When appending to a text file in this way, we would still use the same `try` and `catch` blocks as in Listing 10.1. A version of the program in Listing 10.1 that appends to the file `out.txt` is in the program `AppendTextFile.java` included with the source code for this book available on the Web.

Extra code on the Web

RECAP Opening a Text File for Appending

You can create a stream of the class `PrintWriter` that appends data to the end of an existing text file.

SYNTAX

```
PrintWriter Output_Stream_Name = new PrintWriter(new
FileOutputStream(File_Name, true));
```

EXAMPLE

```
PrintWriter outputStream = new PrintWriter(new
FileOutputStream("out.txt", true));
```

After this statement, you can use the methods `println` and `print` to write to the file, and the new text will be written after the old text in the file. (In practice, you might want to separate the declaration of the stream variable and the invocation of the constructor, as we did in Listing 10.1, so you can handle a `FileNotFoundException` that might be thrown when opening the file.)

SELF-TEST QUESTIONS

4. Write some code that will create a stream named `outStream` that is an object of the class `PrintWriter` and that connects this stream to a text file named `sam.txt` so that your program can send output to the file. If the file `sam.txt` does not exist, create a new empty file. However, if a file named `sam.txt` already exists, erase its old contents so the program can start with an empty file with that name.

5. Repeat Question 4, but if the file `sam.txt` exists already, write the new data after the old contents of the file.

6. Repeat Question 4, but read the name of the file from the user.

7. What kind of exception might be thrown by the following statement, and what would be indicated if this exception were thrown?

```
PrintWriter outputStream = new PrintWriter("out.txt");
```

Reading from a Text File

Listing 10.2 contains a simple program that reads data from a text file and writes it to the screen. The file `out.txt` is a text file that could have been created by either a person using a text editor or a Java program—such as the one in Listing 10.1—using the class `PrintWriter`. Notice that we use the class

LISTING 10.2 Reading Data from a Text File

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;

public class TextFileInputDemo
{
    public static void main(String[] args)
    {
        String fileName = "out.txt";
        Scanner inputStream = null;
        System.out.println("The file " + fileName +
                           "\ncontains the following lines:\n");
        try
        {
            inputStream = new Scanner(new File(fileName));
        }
        catch(FileNotFoundException e)
        {
            System.out.println("Error opening the file " +
                               fileName);
            System.exit(0);
        }
        while (inputStream.hasNextLine())
        {
            String line = inputStream.nextLine();
            System.out.println(line);
        }
        inputStream.close();
    }
}
```

Screen Output

```
The file out.txt
contains the following lines;
1 A tall tree
2 in a short forest is like
3 a big fish in a small pond.
```

Scanner, the same class we have been using to read data from the keyboard. Recall that we passed System.in as an argument to Scanner's constructor.

Unfortunately, we cannot pass a file name to Scanner's constructor. Although Scanner does have a constructor that takes a string argument, the string is interpreted as data, and not the name of a file. Scanner, however, does have a constructor that accepts as an argument an instance of another standard class, File, and File has a constructor to which we can pass a file name. (The next section will describe the class File in more detail.) So a statement of the following form will open the file for input:

```
Scanner Stream_Name = new Scanner(new File(File_Name));
```

Using Scanner to open a text file for input

If your program attempts to open a file for reading, but there is no such file, Scanner's constructor will throw a FileNotFoundException. As you saw earlier in this chapter, a FileNotFoundException is also thrown in certain other situations.

Notice that the program in Listing 10.2 has some general similarities to the program in Listing 10.1 that created a text file. Each opens the file using try-catch blocks, does something with the file, and then closes the file. Let's look at the statements in Listing 10.2 that read and display the entire file:

Reading and displaying an entire text file

```
while (inputStream.hasNextLine())
{
    String line = inputStream.nextLine();
    System.out.println(line);
}
```

This loop reads and then displays each line in the file, one at a time, until the end of the file is reached. The screen output shown in Listing 10.2 assumes that the file out.txt is the one we created in Listing 10.1.

All the methods of Scanner that we have used before are still available to us and work in the same way. Some of these methods, including nextLine, are described in Figure 2.7 of Chapter 2. However, we have not used the method hasNextLine before. It returns true if another line in the file is available for input. Figure 10.3 summarizes this method and a few other analogous methods.

RECAP Reading a Text File

SYNTAX

```
// Open the file
Scanner Input_Stream_Name = null;
try
{
    Input_Stream_Name = new Scanner(new File(File_Name));
}
```

(continued)

VideoNote
Writing and reading a text file

```
catch(FileNotFoundException e)
{
    Statements_Dealing_With_The_Exception
}
// Read the file using statements of the form:
Input_Stream_Name.Scanner_Method();
// Close the file
Input_Stream_Name.close();
```

EXAMPLE

See Listing 10.2.

SELF-TEST QUESTIONS

8. Write some code that will create a stream named `textStream` that is an object of the class `PrintWriter` and that connects the stream to a text file named `dobedo` so that your program can send output to this file.

9. Suppose you run a program that writes to the text file `dobedo`, using the stream defined in the previous self-test question. Write some code that will create a stream named `inputStream` that can be used to read from this text file in the ways we discussed in this section.

10. What is the type of a value returned by the method `next` in the class `Scanner`? What is the type of a value returned by the method `nextLine` in the class `Scanner`?

FIGURE 10.3 Additional Methods in the Class `Scanner` (See also Figure 2.7)

Scanner_Object_Name`.hasNext()` Returns true if more input data is available to be read by the method `next`.
Scanner_Object_Name`.hasNextDouble()` Returns true if more input data is available to be read by the method `nextDouble`.
Scanner_Object_Name`.hasNextInt()` Returns true if more input data is available to be read by the method `nextInt`.
Scanner_Object_Name`.hasNextLine()` Returns true if more input data is available to be read by the method `nextLine`.

10.3 TECHNIQUES FOR ANY FILE

An ounce of prevention is worth a pound of cure.

—COMMON SAYING

This section discusses some techniques that we can use with both text files and binary files, even though our examples here will involve text files. We begin by describing the standard class `File`, which we used in the previous section when reading from a text file.

The Class `File`

The class `File` provides a way to represent file names in a general way. A string such as `"treasure.txt"` might be a file name, but it has only string properties, and Java does not recognize it as a file name. On the other hand, if you pass a file name as a string to the constructor of the class `File`, it produces an object that can be thought of as the name of a file. In other words, it is a system-independent abstraction rather than an actual file. For example, the object

```
new File("treasure.txt")
```

is not simply a string. It is an object that "knows" it is supposed to name a file.

A `File` object represents the name of a file

Although some stream classes have constructors that accept a string as the name of a file, some do not. Some stream classes have constructors that accept only a `File` object as an argument. We already saw in the previous section that when reading a text file, we cannot pass the file's name as a string to `Scanner`'s constructor. Instead, we passed it an object of the class `File`. The class `PrintWriter`, which we used to write a text file, has a constructor that accepts a string as the name of a file, as well as a constructor that accepts an instance of `File` to specify a particular file.

Before we go on to describe some of `File`'s methods, let's look at an example that reads the name of a file from the user. Although our example uses a text file, we can do something similar with binary files.

PROGRAMMING EXAMPLE Reading a File Name from the Keyboard

Thus far, we have assigned a quoted string to a `String` variable and used it as the file name in our programs. However, you might not know what the file name will be when you write a program, so you may want to have the user enter the file name at the keyboard when the program is run. This task is easy to do. Simply have the program read the file name into the `String` variable. This technique is illustrated in Listing 10.3. That program is just like the one

Let the user enter the file name at the keyboard

in Listing 10.2, but it replaces the assignment to `filename` with statements that read the file name from the keyboard.

LISTING 10.3 Reading a File Name and Then the File

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;

public class TextFileInputDemo2
{
    public static void main(String[] args)
    {
        System.out.print("Enter file name: ");
        Scanner keyboard = new Scanner(System.in);
        String fileName = keyboard.next();
        Scanner inputStream = null;
        System.out.println("The file " + fileName + "\n" +
                           "contains the following lines:\n");
        try
        {
            inputStream = new Scanner(new File(fileName));
        }
        catch(FileNotFoundException e)
        {
            System.out.println("Error opening the file " +
                               fileName");
            System.exit(0).
        }
        while (inputStream.hasNextLine())
        {
            String line = inputStream.nextLine();
            System.out.println(line);
        }
        inputStream.close();
    }
}
```

Sample Screen Output

```
Enter file name: out.txt
The file out.txt
contains the following lines:
1 A tall tree
2 in a short forest is like
3 a big fish in a small pond.
```

Notice that our new program reads input from two different places. The `Scanner` object named `keyboard` is used to read the name of the file from the keyboard. The `Scanner` object named `inputStream` is connected to the specified file and is used to read data from that file.

Using Path Names

When writing a file name as an argument to a constructor for opening a file in any of the ways we have discussed, the file is assumed to be in the same directory (folder) as the one in which the program is run. However, we could specify the directory that contains the file, if it is different than the location of the program, by writing a **path name** instead of only the name of the file. A **full path name**, as the name suggests, gives a complete path name, starting from the root directory. A **relative path name** gives the path to the file, starting with the directory containing your program. The way you specify path names depends on your particular operating system, and we will discuss only some of the details here.

A path name specifies the folder containing a file

An example of a typical UNIX path name is

```
/user/smith/homework1/data.txt
```

To create an input stream connected to this file, we would write

```
Scanner inputStream = new Scanner(new
            File("/user/smith/homework1/data.txt"));
```

Windows uses \ instead of / in path names. A typical Windows path name is

```
C:\homework\hw1\data.txt
```

To create an input stream connected to this file, we would write

```
Scanner inputStream = new Scanner(
            new File("C:\\homework\\hw1\\data.txt"));
```

Notice that we must use \\ in place of \, as Java will otherwise interpret a backslash paired with another character—for example, \h—as an escape character.

Although we normally must be careful about using a backslash in a quoted string, this problem does not occur when reading a name from the keyboard. Suppose we run a program like the one in Listing 10.3, and suppose part of the dialogue with the user is as follows:

```
Enter file name:
C:\homework\hw1\data.txt
```

This path name will be understood. The user need not type double backslashes, as in

```
C:\\homework\\hw1\\data.txt
```

In fact, the use of \\ in input might produce an incorrect reading of the file name. When the user enters input at the keyboard, Java "understands" that \h is the backslash character followed by an h and not an escape character, because everything the user types consists of characters.

One way to avoid these escape-character problems altogether is always to use UNIX conventions when writing path names. A Java program will accept a path name written in either Windows or UNIX format, even if it is run on a computer whose operating system does not match the format used. Thus, an alternative way to create an input stream connected to the Windows file

```
C:\homework\hw1\data.txt
```

is the following:

```
Scanner inputStream = new Scanner( new
                File("C:/homework/hw1/data.txt"));
```

Methods of the Class File

You can use methods of the class File to check properties of files. You can check things like whether an existing file either has a specified name or is readable.

Suppose you create a File object and name it fileObject using the following code:

```
File fileObject = new File("treasure.txt");
```

Recall that a File object is not a file. Rather, fileObject is a system-independent abstraction of a file's path name—treasure.txt, in this case.

A File object can check whether a file by that name exists or is readable

After creating fileObject, you can then use the File method exists to test whether any existing file has the name treasure.txt. For example, you can write

```
if (!fileObject.exists())
    System.out.println("No file by that name.")
```

If there already is such a file, you can use the method canRead to see whether the operating system will let you read from the file. For example, you can write

```
if (!fileObject.canRead())
    System.out.println("Not allowed to read from that file.");
```

Files can be tagged as readable or not readable

Most operating systems let you designate some files as not readable or as readable only by certain users. The method canRead provides a good way to check whether you or somebody else has made a file nonreadable—either inadvertently or intentionally.

We could add the following statements to the program shown in Listing 10.3 to check that we have a text file ready as input, right after we read the file's name:

```
File fileObject = new File(fileName);
boolean fileOK = false;
while (!fileOK)
{
    if (!fileObject.exists( ))
        System.out.println("No such file");
    elseif (!fileObject.canRead( ))
            System.out.println("That file is not readable.");
    else
        fileOK = true;
    if (!fileOK)
    {
        System.out.println("Enter file name again:");
        fileName = keyboard.next( );
        fileObject = new File(fileName);
    }
}
```

Extra code on the Web

We have made this change to the program in Listing 10.3 and saved it in the file `FileClassDemo.java`, which is included in the source code for this book on its Web site.

The method `canWrite` is similar to `canRead`, except that the former checks to see whether the operating system will allow you to write to the file. Most operating systems let you designate some files as not writable or as writable only by certain users. Figure 10.4 lists these methods and some others that are in the class `File`.

RECAP The Class `File`

The class `File` represents file names. The constructor for the class `File` takes a string as an argument and produces an object that can be thought of as the file with that name. You can use the `File` object and methods of the class `File` to answer questions such as the following: Does the file exist? Does your program have permission to read the file? Does your program have permission to write to the file? Figure 10.4 summarizes some of the methods for the class `File`.

EXAMPLE

```
File file Object = new File("stuff.txt");
if (!fileObject.exists())
    System.out.println("There is no file named "+
                       "stuff.txt.");
else if (!fileObject.canRead())
    System.out.println("File stuff.txt is "+
                       "not readable.");
```

FIGURE 10.4 Some Methods in the Class `File`

`public boolean canRead()` Tests whether the program can read from the file.
`public boolean canWrite()` Tests whether the program can write to the file.
`public boolean delete()` Tries to delete the file. Returns true if it was able to delete the file.
`public boolean exists()` Tests whether an existing file has the name used as an argument to the constructor when the `File` object was created.
`public String getName()` Returns the name of the file. (Note that this name is not a path name, just a simple file name.)
`public String getPath()` Returns the path name of the file.
`public long length()` Returns the length of the file, in bytes.

FAQ What is the difference between a file and a `File` object?

A file is a collection of data stored on a physical device such as a disc.
A `File` object is a system-independent abstraction of a file's path name.

SELF-TEST QUESTION

11. Write a complete Java program that asks the user for a file name, tests whether the file exists, and—if the file does exist—asks the user whether or not it should be deleted and then does as the user requests.

Defining a Method to Open a Stream

Imagine that we want to write a method that opens a file. We will use a text file here and open it for output, but the idea is applicable to any kind of file. Our method has a `String` parameter that represents the file name, which could be either a literal or a string read from the user, as shown earlier. The following method creates an output stream, connects it to the named text file, and returns the stream:

```
public static PrintWriter openOutputTextFile
    (String fileName) throws FileNotFoundException
{
    PrintWriter toFile = new PrintWriter(fileName);
    return toFile;
}
```

We could invoke this method as follows:

```
PrintWriter outputStream = null;
try
{
    outputStream = openOutputTextFile("data.txt");
}
<appropriate catch block(s)>
```

and go on to use `outputStream` to write to the file. A simple program demonstrating this technique is in the file `OpenFileDemo.java` included with the source code for this book available on the Web.

Extra code on the Web

What if we had written the method as a `void` method so that instead of returning an output stream, it had the stream as a parameter? The following method looks reasonable, but it has a problem:

```
// This method does not do what we want it to do.
public static void openFile(String fileName,
    PrintWriter stream) throws FileNotFoundException
{
    stream = new PrintWriter(fileName);
}
```

Let's consider, for example, the following statements that invoke the method:

```
PrintWriter toFile = null;
try
{
    openFile("data.txt", toFile);
}
```

After this code is executed, the value of `toFile` is still `null`. The file that was opened in the method `openFile` went away when the method ended. The problem has to do with how Java handles arguments of a class type. These arguments are passed to the method as memory addresses that cannot be changed. The state of the object at the memory address normally can be changed, but the memory address itself cannot be changed. Thus, you cannot change the value of `toFile`.

This applies only to arguments of methods. If the stream variable is either an instance variable or declared locally within the body of the method, you can open a file and connect it to the stream and this problem will not occur. Once a stream is connected to a file, you can pass the stream variable as an argument to a method, and the method can change the file.

CASE STUDY Processing a Comma-Separated Values File

A **comma-separated values** or **CSV** file is a simple text format used to store a list of records. A comma is used as a delimiter to separate the fields (also referred to as columns) for each record. This format is commonly used to transfer data between a spreadsheet, database, or custom application program. As an example, consider a cash register that saves a log of the day's sales in a CSV file named `Transactions.txt`. The text file contains the following data:

```
SKU,Quantity,Price,Description
4039,50,0.99,SODA
9100,5,9.50,T-SHIRT
1949,30,110.00,JAVA PROGRAMMING TEXTBOOK
5199,25,1.50,COOKIE
```

The first line of the file is a header that identifies the fields. The first field is the SKU, or stock-keeping unit. This is a unique identifier associated with every product sold. The second field is the quantity of the SKU sold in the transaction. The third field is the price of one unit, and the last field is a description of the item sold. For example, the second line of the file indicates that 50 sodas were sold at a price of $0.99 each and the SKU for the soda is 4039.

There are many ways we might process the data, but in this case study we present a simple strategy to read every field in the file, output each transaction in a more English-like format, and compute the total sales for the cash register. The general algorithm is as follows:

1. Read and skip the header line of the file
2. Repeat while we have not reached the end of the file
   a. Read an entire line (one record) from the file as a String
   b. Create an array of strings from the line where `array[0]` is the value of the first field, `array[1]` is the value of the second field, etc.
   c. Convert any numeric fields from the array of strings to the appropriate numeric data type
   d. Process the fields.

Step 2b of the algorithm might seem to be difficult. At this step we have just read a line from the file. For the first line of our example we will have read `"4039,50,0.99,SODA"` into a variable of type `String`. Since a comma is used to separate every field, we could search the string for the first comma, extract the substring from the beginning of the string to the comma to extract the first field, and repeat this process for every successive field. However, the `split` method associated with the `String` class does all this for us and is defined as follows:

```
public String[] split(String delimiter)
```

The method splits the string around matches of `delimiter` and returns an array of the strings separated by `delimiter`. The `delimiter` parameter is interpreted as a regular expression, which is a flexible way to match patterns. For our purposes we will simply use this as a string that contains our delimiter character of a comma. The following example illustrates the `split` method.

```
String line = "4039,50,0.99,SODA"
String[] ary = line.split(",");
System.out.println(ary[0]);          // Outputs 4039
System.out.println(ary[1]);          // Outputs 50
System.out.println(ary[2]);          // Outputs 0.99
System.out.println(ary[3]);          // Outputs SODA
```

Listing 10.4 applies the technique to the sales transaction scenario. All that remains is to read the file, parse the quantity into an integer, parse the price into a double, and add the business logic to compute total sales by accumulating the product of the quantity sold multiplied by the price of the item. In the program, we used `System.out.printf` to format the price and totals with two decimal places.

LISTING 10.4 Processing a Comma-Separated Values File Containing Sales Transactions *(part 1 of 2)*

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.File;
import java.util.Scanner;
public class TransactionReader
{
public static void main(String[] args)
    {
        String fileName = "Transactions.txt";
        try
        {
        Scanner inputStream = new Scanner(new File(fileName));
        // Skip the header line by reading and ignoring it
        String line = inputStream.nextLine();
        // Total sales
        double total = 0;
        // Read the rest of the file line by line
        while (inputStream.hasNextLine())
        {
            // Contains SKU,Quantity,Price,Description
            line = inputStream.nextLine();
```

(continued)

LISTING 10.4 Processing a Comma-Separated Values File Containing Sales Transactions *(part 2 of 2)*

```
                // Turn the string into an array of strings
                String[] ary = line.split(",");
                // Extract each item into an appropriate
                // variable
                String SKU = ary[0];
                int quantity = Integer.parseInt(ary[1]);
                double price = Double.parseDouble(ary[2]);
                String description = ary[3];
                // Output item
                System.out.printf("Sold %d of %s (SKU: %s) at "+
                    "$%1.2f each.\n",
                    quantity, description, SKU, price);
                // Compute total
                total += quantity * price;
            }
            System.out.printf("Total sales: $%1.2f\n",total);
            inputStream.close( );
        }
        catch(FileNotFoundException e)
        {
            System.out.println("Cannot find file " + fileName);
        }
        catch(IOException e)
        {
            System.out.println("Problem with input from file " +
            fileName);
        }
    }
}
```

Sample Screen Output

```
Sold 50 of SODA (SKU: 4039) at $0.99 each.
Sold 5 of T-SHIRT (SKU: 9100) at $9.50 each.
Sold 30 of JAVA PROGRAMMING TEXTBOOK (SKU: 1949) at
$110.00 each.
Sold 25 of COOKIE (SKU: 5199) at $1.50 each.
Total sales: $3434.50
```

10.4 BASIC BINARY-FILE I/O

The White Rabbit put on his spectacles. "Where shall I begin, please your Majesty?" he asked.

"Begin at the beginning," the King said, very gravely, "And go on till you come to the end: then stop."

—LEWIS CARROLL, *Alice's Adventures in Wonderland*

We will use the stream classes `ObjectInputStream` and `ObjectOutputStream` to read and write binary files. Each of these classes has methods to read or write data one byte at a time. These streams can also convert numbers and characters to bytes that can be stored in a binary file. They allow your program to be written as if the data placed in the file, or read from the file, were made up not just of bytes but of either items of any of Java's primitive data types—such as `int`, `char`, and `double`—or strings or even objects of classes you define, as well as entire arrays. If you do not need to access your files via a text editor, the easiest and most efficient way to read data from and write data to files is to use `ObjectOutputStream` to write to a binary file and `ObjectInputStream` to read from the binary file.

We begin by creating a binary file, and then discuss how to write primitive-type data and strings to the file. Later, we will investigate using other objects and arrays as data for input and output.

Creating a Binary File

If you want to create a binary file to store values of a primitive type, strings, or other objects, you can use the stream class `ObjectOutputStream`. Listing 10.5 shows a program that writes integers to a binary file. Let's look at the details shown in that program.

Note that the substance of the program is in a `try` block. Any code that does binary-file I/O in the ways we are describing can throw an `IOException`. Your programs can catch any `IOException` that is thrown, so that you get an error message and the program ends normally.

We can create an output stream for the binary `file numbers.dat` as follows:

```
ObjectOutputStream outputStream =
    new ObjectOutputStream(new FileOutputStream
            ("numbers.dat"));
```

Opening a binary file for output

As with text files, this process is called opening the (binary) file. If the file `numbers.dat` does not already exist, this statement will create an empty file named `numbers.dat`. If the file `numbers.dat` already exists, this statement will erase the contents of the file, so that the file starts out empty. The situation is basically the same as what happens when opening a text file, except that we're using a different class here.

Note that the constructor for `ObjectOutputStream` cannot take a string argument, but the constructor for `FileOutputStream` can. Moreover, `ObjectOutputStream` does have a constructor that will accept an object of

LISTING 10.5 Using `ObjectOutputStream` to Write to a File *(part 1 of 2)*

```
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Scanner;

public class BinaryOutputDemo
{
    public static void main(String[] args)
    {
        String fileName = "numbers.dat";
        try
        {
            ObjectOutputStream outputStream =
            new ObjectOutputStream(new
                        FileOutputStream(fileName));
            Scanner keyboard = new Scanner(System.in);

            System.out.println("Enter nonnegative integers.");
            System.out.println("Place a negative number at the "+
                               "end.");

            int anInteger;
            do
            {
                anInteger = keyboard.nextInt();
                outputStream.writeInt(anInteger);
            } while (anInteger >= 0);

            System.out.println("Numbers and sentinel value");
            System.out.println("written to the file " + fileName);
            outputStream.close();
        }
        catch(FileNotFoundException e)
        {
            System.out.println("Problem opening the file " +
            fileName);
        }
        catch(IOException e)
        {
            System.out.println("Problem with output to file " +
            fileName);
        }
    }
}
```

A binary file is closed in the same way as a text file.

(continued)

LISTING 10.5 Using `ObjectOutputStream` to Write to a File *(part 2 of 2)*

Sample Screen Output

```
Enter nonnegative integers.
Place a negative number at the end.
1 2 3 -1
Number and sentinel value
written to the file numbers.dat
```

The binary file after the program is run:

This file is a binary file. You cannot read this file using a text editor.

1	2	3	-1

The -1 in this file is a sentinel value. Ending a file with a sentinel value is not essential, as you will see later.

`FileOutputStream` as its argument. So in much the same way that we passed a `File` object to the `Scanner` constructor when we read from a text file, we pass a `FileOutputStream` object to the constructor of `ObjectOutputStream`. Note that the `ObjectOutputStream` constructor can throw an `IOException` and the `FileOutputStream` constructor can throw a `FileNotFoundException`.

Exceptions are possible

Writing Primitive Values to a Binary File

An object of the class `ObjectOutputStream` does not have a method named `println`, as objects do when writing to either a text file or the screen. However, it does have a method named `writeInt` that can write a single `int` value to a binary file, as well as other output methods that we will discuss shortly. So once we have the stream `outputStream` of the class `ObjectOutputStream` connected to a file, we can write integers to the file, as in the following statement from Listing 10.5:

```
outputStream.writeInt(anInteger);
```

The method `writeInt` can throw an `IOException`.

Listing 10.5 shows the numbers in the file `numbers.dat` as if they were written in a human-readable form. That is not how they are actually written, however. A binary file has no lines or other separators between data items. Instead, items are written to the file in binary—each as a sequence of bytes—one immediately after the other. These encoded values typically cannot be read using an editor. Realistically, they will make sense only to another Java program.

You can use a stream from the class `ObjectOutputStream` to write values of any primitive type. Each primitive data type has a corresponding write method in the class `ObjectOutputStream`, such as `writeLong`, `writeDouble`, `writeFloat`, and `writeChar`.

Writing values of any primitive type to a binary file

The method `writeChar` can be used to write a single character. For example, the following would write the character `'A'` to the file connected to the stream named `outputStream`:

```
outputStream.writeChar('A');
```

The method `writeChar` has one possibly surprising property: It expects its argument to be of type `int`. So if you have an argument of type `char`, the `char` value will be type cast to an `int` before it is given to the method `writeChar`. Thus, the previous invocation is equivalent to the following one:

```
outputStream.writeChar((int)'A');
//You can omit the type cast
```

After you finish writing to a binary file, you close it in the same way that you close a text file. Here, the statement

```
outputStream.close();
```

closes the binary file.

Figure 10.5 summarizes some methods in the class `ObjectOutputStream`, including those that we have not discussed as yet. Many of these methods can throw an `IOException`.

RECAP Creating a Binary File

SYNTAX

```
try
{
    // Open the file
    ObjectOutputStream Output_Stream_Name =
        new ObjectOutputStream(new FileOutputStream(File_Name));
    // Write the file using statements of the form:
    Output_Stream_Name.MethodName(Argument); // See Figure 10.5
    // Close the file
    Output_Stream_Name.close();
}
catch(FileNotFoundException e)
{
    Statements_Dealing_With_The_Exception
}
catch(IOException e)
{
    Statements_Dealing_With_The_Exception
}
```

EXAMPLE

See Listing 10.5.

FIGURE 10.5 Some Methods in the Class `ObjectOutputStream` *(part 1 of 2)*

`public ObjectOutputStream(OutputStream streamObject)`
Creates an output stream that is connected to the specified binary file. There is no constructor that takes a file name as an argument. If you want to create a stream by using a file name, you write either

```
new ObjectOutputStream(new FileOutputStream(File_Name))
```

or, using an object of the class `File`,

```
new ObjectOutputStream(new FileOutputStream(
                       new File(File_Name)))
```

Either statement creates a blank file. If there already is a file named *File_Name*, the old contents of the file are lost.

The constructor for `FileOutputStream` can throw a `FileNotFoundException`. If it does not, the constructor for `ObjectOutputStream` can throw an `IOException`.

`public void writeInt(int n) throws IOException`
Writes the `int` value `n` to the output stream.

`public void writeLong(long n) throws IOException`
Writes the `long` value `n` to the output stream.

`public void writeDouble(double x) throws IOException`
Writes the `double` value `x` to the output stream.

`public void writeFloat(float x) throws IOException`
Writes the `float` value `x` to the output stream.

`public void writeChar(int c) throws IOException`
Writes a `char` value to the output stream. Note that the parameter type of `c` is `int`. However, Java will automatically convert a `char` value to an `int` value for you. So the following is an acceptable invocation of `writeChar`:

```
outputStream.writeChar('A');
```

`public void writeBoolean(boolean b) throws IOException`
Writes the `boolean` value `b` to the output stream.

`public void writeUTF(String aString) throws IOException`
Writes the string `aString` to the output stream. `UTF` refers to a particular method of encoding the string. To read the string back from the file, you should use the method `readUTF` of the class `ObjectInputStream`. These topics are discussed in the next section.

(continued)

FIGURE 10.5 Some Methods in the Class `ObjectOutputStream` *(part 2 of 2)*

`public void writeObject(Object anObject) throws IOException, NotSerializableException, InvalidClassException` Writes `anObject` to the output stream. The argument should be an object of a serializable class, a concept discussed later in this chapter. Throws a `NotSerializableException` if the class of `anObject` is not serializable. Throws an `InvalidClassException` if there is something wrong with the serialization. The method `writeObject` is covered later in this chapter.
`public void close() throws IOException` Closes the stream's connection to a file.

SELF-TEST QUESTIONS

12. Write some Java code to create an output stream of type `ObjectOutputStream` that is named `toFile` and is connected to a binary file named `stuff.data`.

13. Give three statements that will write the values of the three `double` variables `x1`, `x2`, and `x3` to the file `stuff.data`. Use the stream `toFile` that you created for the previous question.

14. Give a statement that will close the stream `toFile` created for the previous two questions.

15. What import statement(s) do you use when creating a binary file?

Writing Strings to a Binary File

Use `writeUTF` for strings

To write a string to a binary file, we use the method `writeUTF`. For example, if `outputStream` is a stream of type `ObjectOutputStream`, the following will write the string `"Hi Mom"` to the file connected to that stream:

```
outputStream.writeUTF("Hi Mom");
```

Of course, with any of the write methods, you can use a variable of the appropriate type (in this case, the type `String`) as an argument to the method, instead of using a literal.

You may write output of different types to the same binary file. For example, you may write a combination of `int`, `double`, and `String` values. However, mixing types in a binary file does require special care, so that later the values in the file can be read correctly. In particular, we need to know the

order in which the various types appear in the file, because, as you will see, we use a different method to read data of each different type.

FAQ What does UTF stand for?

To write an `int` to a stream of the class `ObjectOutputStream`, you use `writeInt`; to write a `double`, you use `writeDouble`; and so forth. However, to write a string, you use `writeUTF`. There is no method called `writeString` in `ObjectOutputStream`. Why this funny name `writeUTF`? UTF stands for Unicode Text Format. That is not a very descriptive name. Here is the full story:

Recall that Java uses the Unicode character set, which includes many characters used in languages whose character sets are very different from that of English. Most text editors and operating systems use the ASCII character set, which is the character set normally used for English and for typical Java programs. The ASCII character set is a subset of the Unicode character set, so the Unicode character set has many characters you do not need. For English-speaking countries, the Unicode way of encoding characters is not a very efficient scheme. The UTF coding scheme is an alternative scheme that codes all of the Unicode characters but favors the ASCII character set. It does so by giving short, efficient codes for ASCII characters but inefficient codes for the other Unicode characters. However, if you do not use the other Unicode characters, this is a good deal.

Some Details About `writeUTF`

The method `writeInt` writes integers into a file, using the same number of bytes—that is, the same number of 0s and 1s—to store any integer. Similarly, the method `writeLong` uses the same number of bytes to store each value of type `long`. However, these methods use different numbers of bytes. The situation is the same for all the other write methods that write primitive types to binary files. The method `writeUTF`, however, uses varying numbers of bytes to store different strings in a file. Longer strings require more bytes than shorter strings. This condition can present a problem to Java, because data items in a binary file have no separators between them. Java can make this approach work by writing some extra information at the start of each string. This extra information tells how many bytes are used to write the string, so that `readUTF` knows how many bytes to read and convert. (The method `readUTF` will be discussed a little later in this chapter, but as you may have already guessed, it reads a `String` value.)

The situation with `writeUTF` is even a little more complicated than we've just described, however. Notice that we said that the information at the start of the string code in the file tells how many *bytes* to read, *not how many characters are in the string*. These two figures are not the same. With the UTF way of encoding, different characters are encoded in different numbers of bytes. However, all the ASCII characters are stored in just one byte. If you are using only ASCII characters, therefore, this difference is more theoretical than real to you.

Reading from a Binary File

If you write to a binary file by using `ObjectOutputStream`, you can read from that file by using the stream class `ObjectInputStream`. Figure 10.6 gives some of the most commonly used methods for this class. If you compare these methods with the methods for `ObjectOutputStream` given in Figure 10.5, you will see that each output method has a corresponding input method. For example, if you write an integer to a file by using the method `writeInt` of `ObjectOutputStream`, you can read that integer from the file by using the method `readInt` of `ObjectInputStream`. If you write a number to a file by using the method `writeDouble` of `ObjectOutputStream`, you can use the method `readDouble` of `ObjectInputStream` to read it from the file, and so on.

You open a binary file for reading using `ObjectInputStream` in a manner similar to what you have already seen for `ObjectOutputStream`. The program in Listing 10.6 opens a binary file and connects it to a stream named `inputStream` as follows:

Opening a binary file for input

```
ObjectInputStream inputStream =
    new ObjectInputStream(new FileInputStream(fileName));
```

Note that this statement is like the analogous one in Listing 10.5, except that here we use the classes `ObjectInputStream` and `FileInputStream` instead of `ObjectOutputStream` and `FileOutputStream`, respectively. Once again, the class that defines the methods we need does not have a constructor that accepts a string as an argument. In addition, the constructor for `FileInputStream` can throw a `FileNotFoundException`, which is a kind of `IOException`. If the `FileInputStream` constructor succeeds, the constructor for `ObjectInputStream` can throw a different `IOException`.

`ObjectInputStream` allows you to read input of different types from the same file. So you may read a combination of, for example, `int` values, `double` values, and `String` values. However, if the next data item in the file is not of the type expected by the reading method, the result is likely to be unpleasant. For example, if your program writes an integer using `writeInt`, any program that reads that integer should read it using `readInt`. If you instead use `readLong` or `readDouble`, for example, your program will misbehave in unpredictable ways.

Reading multiple types

FIGURE 10.6 Some Methods in the Class `ObjectInputStream` *(part 1 of 2)*

`ObjectInputStream(InputStream streamObject)`

Creates an input stream that is connected to the specified binary file. There is no constructor that takes a file name as an argument. If you want to create a stream by using a file name, you use either

```
new ObjectInputStream(new FileInputStream(File_Name))
```

or, using an object of the class `File`,

```
new ObjectInputStream(new FileInputStream(
                      new File(File_Name)))
```

The constructor for `FileInputStream` can throw a `FileNotFoundException`. If it does not, the constructor for `ObjectInputStream` can throw an `IOException`.

`public int readInt() throws EOFException, IOException`

Reads an `int` value from the input stream and returns that `int` value. If `readInt` tries to read a value from the file that was not written by the method `writeInt` of the class `ObjectOutputStream` (or was not written in some equivalent way), problems will occur. If the read goes beyond the end of the file, an `EOFException` is thrown.

`public long readLong() throws EOFException, IOException`

Reads a `long` value from the input stream and returns that `long` value. If `readLong` tries to read a value from the file that was not written by the method `writeLong` of the class `ObjectOutputStream` (or was not written in some equivalent way), problems will occur. If the read goes beyond the end of the file, an `EOFException` is thrown.

Note that you cannot write an integer using `writeLong` and later read the same integer using `readInt`, or to write an integer using `writeInt` and later read it using `readLong`. Doing so will cause unpredictable results.

`public double readDouble() throws EOFException, IOException`

Reads a `double` value from the input stream and returns that `double` value. If `readDouble` tries to read a value from the file that was not written by the method `writeDouble` of the class `ObjectOutputStream` (or was not written in some equivalent way), problems will occur. If the read goes beyond the end of the file, an `EOFException` is thrown.

`public float readFloat() throws EOFException, IOException`

Reads a `float` value from the input stream and returns that `float` value. If `readFloat` tries to read a value from the file that was not written by the method `writeFloat` of the class `ObjectOutputStream` (or was not written in some equivalent way), problems will occur. If the read goes beyond the end of the file, an `EOFException` is thrown.

Note that you cannot write a floating-point number using `writeDouble` and later read the same number using `readFloat`, or write a floating-point number using `writeFloat` and later read it using `readDouble`. Doing so will cause unpredictable results, as will other type mismatches, such as writing with `writeInt` and then reading with `readFloat` or `readDouble`.

(continued)

FIGURE 10.6 Some Methods in the Class `ObjectInputStream` *(part 2 of 2)*

`public char readChar() throws EOFException, IOException` Reads a `char` value from the input stream and returns that `char` value. If `readChar` tries to read a value from the file that was not written by the method `writeChar` of the class `ObjectOutputStream` (or was not written in some equivalent way), problems will occur. If the read goes beyond the end of the file, an `EOFException` is thrown.
`public boolean readBoolean() throws EOFException, IOException` Reads a `boolean` value from the input stream and returns that `boolean` value. If `readBoolean` tries to read a value from the file that was not written by the method `writeBoolean` of the class `ObjectOutputStream` (or was not written in some equivalent way), problems will occur. If the read goes beyond the end of the file, an `EOFException` is thrown.
`publicString readUTF() throws IOException, UTFDataFormatException` Reads a `String` value from the input stream and returns that `String` value. If `readUTF` tries to read a value from the file that was not written by the method `writeUTF` of the class `ObjectOutputStream` (or was not written in some equivalent way), problems will occur. One of the exceptions `UTFDataFormatException` or `IOException` can be thrown.
`Object readObject() throws ClassNotFoundException, InvalidClassException, OptionalDataException, IOException` Reads an object from the input stream. Throws a `ClassNotFoundException` if the class of a serialized object cannot be found. Throws an `InvalidClassException` if something is wrong with the serializable class. Throws an `OptionalDataException` if a primitive data item, instead of an object, was found in the stream. Throws an `IOException` if there is some other I/O problem. The method `readObject` is covered in Section 10.5.
`public void close() throws IOException` Closes the stream's connection to a file.

LISTING 10.6 Using `ObjectInputStream` to Read from a File *(part 1 of 2)*

Assumes the program in Listing 10.4 was already run

```
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.io.EOFException;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Scanner;
```

(continued)

LISTING 10.6 Using `ObjectInputStream` to Read from a File *(part 2 of 2)*

```
public class BinaryInputDemo
{
    public static void main(String[] args)
    {
        String fileName = "numbers.dat";
        try
        {
            ObjectInputStream inputStream =
                  new ObjectInputStream(new FileInputStream(fileName));
            System.out.println("Reading the nonnegative integers");
            System.out.println("in the file " + fileName);
            int anInteger = inputStream.(readInt);
            while (anInteger >= 0)
            {
                System.out.ptintln(anInteger);
                anInteger = inputStream.readInt();
            }
            System.out.println("End of reading from file.");
            inputStream.close();
        }
        catch(FileNotFoundException e)
        {
            System.out.println("Problem opening the file " + fileName);
        }
        catch(EOFException e)
        {
            System.out.println("Problem reading the file " + fileName);
            System.out.println("Reached end of the file.");
        }
        catch(IOException e)
        {
            System.out.println("Problem reading the file " + fileName);
        }
    }
}
```

Screen Output

```
Reading the nonnegative integers
in the file number.dat
1
2
3
End of reading from file.
```

Notice that the sentinel value `-1` *is read from the file but is not displayed on the screen.*

RECAP Reading from a Binary File

SYNTAX

```
try
{
    // Open the file
    ObjectInputStream Input_Stream_Name =
          new ObjectInputStream(new FileInputStream
                                (File_Name));
    // Read the file using statements of the form:
    Input_Stream_Name.MethodName(Argument); // See Figure 10.6
    // Close the file
    Input_Stream_Name.close();
}
catch(FileNotFoundException e)
{
    Statements_Dealing_With_The_Exception
}
catch(EOFException e)
{
    Statements_Dealing_With_The_Exception
}
catch(IOException e)
{
    Statements_Dealing_With_The_Exception
}
```

EXAMPLE

See Listing 10.6.

RECAP `FileInputStream` and `FileOutputStream`

In this book, we use the classes `FileInputStream` and `FileOutputStream` for their constructors and nothing else. Each of these two classes has a constructor that takes a file name as an argument. We use these constructors to produce arguments for the constructors for stream classes, such as `ObjectInputStream` and `ObjectOutputStream`, that do not take a file name as an argument. Below are examples of using `FileInputStream` and `FileOutputStream`:

(continued)

```
ObjectInputStream fileInput = new ObjectInputStream(
                    new FileInputStream("rawstuff.dat"));
ObjectOutputStream fileOutput = new ObjectOutputStream(
                     new FileOutputStream("nicestuff.dat"));
```

We used similar statements in Listing 10.6 and Listing 10.5, respectively.

The constructors for `FileInputStream` and `FileOutputStream` can throw an exception in the class `FileNotFoundException`. A `FileNotFoundException` is a kind of `IOException`.

GOTCHA Using `ObjectInputStream` to Read a Text File

Binary files and text files encode their data in different ways. Thus, a stream that expects to read a binary file, such as a stream in the class `ObjectInputStream`, will have problems reading a text file. If you attempt to read a text file using a stream in the class `ObjectInputStream`, your program will either read "garbage values" or encounter some other error condition. Similarly, if you attempt to use `Scanner` to read a binary file as if it were a text file, you will also get into trouble. ■

SELF-TEST QUESTIONS

16. Write some Java code to create an input stream of type `ObjectInputStream` that is named `fromFile` and is connected to a file named `stuff.data`.

17. Give three statements that will read three numbers of type `double` from the file `stuff.data`. Use the stream `fromFile` that you created in the previous question. Declare three variables to hold the three numbers.

18. Give a statement that will close the stream `fromFile` that you created for the previous two questions.

19. Can you use `writeInt` to write a number to a file and then read that number using `readLong`? Can you read that number using `readDouble`?

20. Can you use `readUTF` to read a string from a text file?

21. Write a complete Java program that asks the user for the name of a binary file and writes the first data item in that file to the screen. Assume that the first data item is a string that was written to the file with the method `writeUTF`.

The Class EOFException

An EOFException can end a loop

Many of the methods that read from a binary file will throw an `EOFException` when they try to read beyond the end of a file. As illustrated by the program in Listing 10.7, the class `EOFException` can be used to test for the end of a file when you are using `ObjectInputStream`. In that sample program, the statements that read the file are placed within a `while` loop whose boolean expression is true. Although this loop appears to be "infinite," it does come to an end: When the end of the file is reached, an exception is thrown, ending the entire `try` block and passing control to the `catch` block.

LISTING 10.7 Using `EOFException` *(part 1 of 2)*

Assumes the program in Listing 10.4 was already run

```
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.io.EOFException;
import java.io.FileNotFoundException;
import java.io.IOException;

    public class EOFExceptionDemo
    {
        public static void main(String[] args)
        {

            String fileName = "numbers.dat";
            try
            {
                ObjectInputStream inputStream =
                     new ObjectInputStream(new
                     FileInputStream(fileName));
                System.out.println("Reading ALL the integers");
                System.out.printin("in the file " + fileName);
                try
                {
                    while (true)        The loop ends when an
                    {                   exception is thrown.

                        int anInteger = inputStream.readInt();
                        System.out.println(anInteger);
                    }
                }
                catch(EOFException e)
```

(continued)

LISTING 10.7 Using `EOFException` *(part 2 of 2)*

```
            {
                System.out.println("End of reading from file.");
            }
            inputStream.close();
        }
        catch(FileNotFoundException e)
        {
            System.out.println("Cannot find file " + fileName);
        }
        catch(IOException e)
        {
            System.out.println("Problem with input from file " +
            fileName);
        }
    }
}
```

Screen Output

```
Reading ALL the integers
in the file numbers.dat
1
2
3
-1
End of reading from file.
```

When you use `EOFException` *to end reading, you can read files that contain any kind of integers, including the* `-1` *here, which is treated just like any other integer.*

It is instructive to compare the program in Listing 10.7 with the similar program in Listing 10.6. The one in Listing 10.6 checks for the end of a file by testing for a negative number. This approach is fine, but it means that you cannot store negative numbers in the file, except as a sentinel value. On the other hand, the program in Listing 10.7 uses `EOFException` to test for the end of a file. Thus, it can handle files that store any kind of integers, including negative integers.

RECAP The `EOFException` Class

When reading primitive data from a binary file using the methods listed in Figure 10.6 for the class `ObjectInputStream`, if your program attempts to read beyond the end of the file, an `EOFException` is thrown. This exception can be used to end a loop that reads all the data in a file.

The class `EOFException` is derived from the class `IOException`. So every exception of type `EOFException` is also of type `IOException`.

PROGRAMMING TIP Always Check for the End of a File

Nothing good happens if your program reads beyond the end of a file. Exactly what occurs will depend on the details of your program: It might enter an infinite loop or end abnormally. Always be sure that your program checks for the end of a file and does something appropriate when it reaches that point. Even if you think your program will not read past the end of the file, you should provide for this eventuality, just in case things do not go exactly as you planned.

GOTCHA Checking for the End of a File in the Wrong Way

Different file-reading methods—usually in different classes—check for the end of a file in different ways. Some throw an exception of the class `EOFException` when they try to read beyond the end of a file. Others return a special value, such as `null`. When reading from a file, you must be careful to test for the end of a file in the correct way for the method you are using. If you test for the end of a file in the wrong way, one of two things will probably happen: Either your program will go into an unintended infinite loop or it will terminate abnormally.

Not all methods in all classes will throw an `EOFException` when they try to read beyond the end of a file. For the classes discussed in this book, the rule is as follows: If your program is reading primitive data from a binary file, it will throw an `EOFException`. If, however, your program is reading from a text file, it will return some special value, such as `null`, at the end of the file, and no `EOFException` will be thrown.

RECAP Detecting the End of Data in a Binary File

You can read all of the data in a binary file in one of two ways:

- By detecting a sentinel value written at the end of the file
- By catching an `EOFException`

PROGRAMMING EXAMPLE Processing a File of Binary Data

Listing 10.8 contains a program that does some simple processing of data. It asks the user for two file names, reads the numbers in the input file, doubles them, and writes the resulting numbers to the output file. This programming task is not very complicated, but it does employ many standard programming

LISTING 10.8 Processing a File of Binary Data *(part 1 of 3)*

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.EOFException;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Scanner;

public class Doubler
{
    private ObjectInputStream inputStream = null;
    private ObjectOutputStream outputStream = null;

    /**
    Doubles the integers in one file and puts them in another file.
    */
    public static void main(String[] args)
    {
        Doubler twoTimer = new Doubler();
        twoTimer.connectToInputFile();
        twoTimer.connectToOutputFile();
        twoTimer.timesTwo();
        twoTimer.closeFiles();
        System.out.println("Numbers from input file");
        System.out.println("doubled and copied to output file.");
    }
    public void connectToInputFile()
    {
        String inputFileName =
            getFileName("Enter name of input file:");
        try
        {
            inputStream = new ObjectInputStream(
                              new FileInputStream(inputFileName));
        }
        catch(FileNotFoundException e)
        {
            System.out.println("File " + inputFileName +
                               " not found.");
            System.exit(0);
        }
        catch(IOException e)
        {
            System.out.println("Error opening input file" +
                               inputFileName);
```

(continued)

LISTING 10.8 Processing a File of Binary Data *(part 2 of 3)*

```
            System.exit(0);
        }
    }
    private String getFileName(String prompt)
    {
        String fileName = null;
        System.out.println(prompt);
        Scanner keyboard = new Scanner(System.in);
        fileName = keyboard.next();

        return fileName;
    }
    public void connectToOutputFile()
    {
        String outputFileName =
                        getFileName("Enter name of output file:");
        try
        {
            outputStream = new ObjectOutputStream(
                            new FileOutputStream(outputFileName));
        }
        catch(IOException e)
        {
            System.out.println("Erroropeningoutputfile" +
                                outputFileName);
            System.out.println(e.getMessage());
            System.exit(0);
        }
    }
```

A class used in a real-life application would usually transform the input data in a more complex way before writing it to the output file. Such a class likely would have additional methods.

```
    public void timesTwo()
    {
        try
        {
            while (true)
            {
                int next = inputStream.readInt();
                outputStream.writeInt(2 * next);
            }
        }
```

(continued)

LISTING 10.8 Processing a File of Binary Data *(part 3 of 3)*

```
    catch(EOFException e)
    {
        //Do nothing. This just ends the loop.
    }
    catch(IOException e)
    {
        System.out.println(
                        "Error: reading or writing files.");
        System.out.println(e.getMessage());
        System.exit(0);
    }
    }
    public void closeFiles()
    {
        try
        {
            inputStream.close();
            outputStream.close();
        }
        catch(IOException e)
        {
            System.out.println("Error closing files " +
                               e.getMessage());
            System.exit(0);
        }
    }
}
```

techniques for handling file I/O. In particular, note that the variables for stream objects connected to the files are instance variables, and note that the task is divided into subtasks assigned to various methods.

We have made the `try` blocks small so that, when an exception is thrown, it is caught in a nearby `catch` block. If we had fewer—and larger—`try` blocks, it would have been harder to decide what part of the code had thrown an exception.

VideoNote
Writing and reading a binary file

GOTCHA Exceptions, Exceptions, Exceptions

Many situations involving a text file that do not throw exceptions during I/O do throw exceptions when a binary file is involved. So you will need to do more exception handling when working with binary files than when working with text files. For example, closing a text file connected to a stream of type `PrintWriter` does not cause an exception to be thrown. However, closing a binary-file stream of the types we have been discussing can cause an `IOException` to be thrown. When doing binary-file I/O as we have described

it, almost anything can cause an exception to be thrown. Also, the methods `writeObject` and `readObject` can throw a long list of exceptions, which you can see by checking Figures 10.5 and 10.6. ■

SELF-TEST QUESTIONS

22. Suppose that you want to create an input stream and connect it to the binary file named `mydata.dat`. Will the following code work? If not how can you write something similar that does work?

```
ObjectInputStream inputStream = new
    ObjectInputStream("mydata.dat");
```

23. Does the class `FileInputStream` have a method named `readInt`? Does it have one named `readDouble`? Does it have one named `readUTF`?

24. Does the class `FileOutputStream` have a constructor that accepts a file name as an argument?

25. Does the class `ObjectOutputStream` have a constructor that accepts a file name as an argument?

26. When opening a binary file for output in the ways discussed in this chapter, might an exception be thrown? What kind of exception? What are the answers to these questions if we open a binary file for input instead of for output?

27. Suppose that a binary file contains exactly three numbers written to the file using the method `writeDouble` of the class `ObjectOutputStream`. Suppose further that you write a program to read all three numbers using three invocations of the method `readDouble` of the class `ObjectInputStream`. If your program invokes `readDouble` a fourth time, what will happen?

28. The following code appears in the program in Listing 10.7:

```
try
{
    while (true)
    {
            int anInteger = inputStream.readInt();
            System.out.println(anInteger);
    }
}
catch(EOFException e)
{
    System.out.println("End of reading from file.");
}
```

Why doesn't this code really include an infinite loop?

29. Write a complete Java program that will display all the numbers in a binary file named `temperatures.dat` on the screen, one per line. Assume that the entire file was written with the method `writeDouble`.

10.5 BINARY-FILE I/O WITH OBJECTS AND ARRAYS

In this section, we cover I/O for binary files involving objects of a class and arrays, which, you'll recall, are really objects. We will use the classes `ObjectInputStream` and `ObjectOutputStream`.

Binary-File I/O with Objects of a Class

We have seen how to write primitive values and strings to a binary file, and how to read them again. How would we write and read objects other than strings? We could, of course, write an object's instance variables to a file and invent some way to reconstruct the object when we read the file. Since an instance variable could be another object that itself could have an object as an instance variable, however, completing this task sounds formidable.

Fortunately, Java provides a simple way—called **object serialization**—to represent an object as a sequence of bytes that can be written to a binary file. This process will occur automatically for any object that belongs to a class that is **serializable.** Making a class serializable is easy to do. We simply add the two words `implementsSerializable` to the heading of the class definition, as in the following example:

Objects of a serializable class can be written to a binary file

```
public class Species implements Serializable
```

Actually, we do need to worry about some details involving a class's instance variables, but since those details will not be relevant to our example, we will postpone discussing them until the next section. We will also discuss the meaning of serialization at that time.

`Serializable` is an interface in the Java Class Library within the package `java.io`. The interface is empty, so we have no additional methods to implement. Although an empty interface might seem useless, this one tells Java to make the class serializable. We make the interface available to our program by including the following `import` statement:

```
import java.io.Serializable;
```

In Listing 10.9, we have made the class `Species`, which we saw in Listing 5.17 of Chapter 5, serializable and added constructors and a `toString` method. Let's use the class `Species` to illustrate the reading and writing of objects to a binary file.

We can write objects of a serializable class to a binary file by using the method `writeObject` of the class `ObjectOutputStream` and then read those objects from the file by using the method `readObject` of the class `ObjectInputStream`. Listing 10.10 provides an example of this process. To

LISTING 10.9 The Class Species Serialized for Binary-File I/O

This is a new, improved definition of the class `Species` *and replaces the definition in Listing 5.19 of Chapter 5.*

```
import java.io.Serializable;
import java.util.Scanner;

/**
Serialized class for data on endangered species.
*/
public class Species implements Serializable
{
    private String name;
    private int population;
    private double growthRate;

    public Species()
    {
        name = null;
        population = 0;
        growthRate = 0;
    }

    public Species(String initialName, int initialPopulation,
                   double initialGrowthRate)
    {
        name = initialName;
        if (initialPopulation >= 0)
            population = initialPopulation;
        else
        {
            System.out.println("ERROR: Negative population.");
            System.exit(0);
        }

        growthRate = initialGrowthRate;
    }
    public String toString()
    {
        return ("Name = " + name + "\n" +
                "Population = " + population + "\n" +
                "Growth rate = " + growthRate + "%");
    }
        <Other methods are the same as those in Listing 5.19 of Chapter 5,
        but they are not needed for the discussion in this chapter.>
}
```

These two words and the `import` *statement make this class serializable.*

LISTING 10.10 File I/O of Class Objects *(part 1 of 3)*

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class ClassObjectIODemo
{
    public static void main(String[]args)
    {
        ObjectOutputStream outputStream = null;
        String fileName = "species.records";

        try
        {
            outputStream = new ObjectOutputStream(
                           new FileOutputStream(fileName));
        }
        catch(IOException e)
        {
            System.out.println("Error opening output file " +
                               fileName + ".");
            System.exit(0);
        }
        Species califCondor =
                    new Species("Calif. Condor", 27, 0.02);
        Species blackRhino =
                    new Species("Black Rhino", 100, 1.0);
        try
        {
            outputStream.writeObject(califCondor);
            outputStream.writeObject(blackRhino);
            outputStream.Close();
        }
        catch(IOException e)
        {
            System.out.println("Error wring to file " +
                               fileName + ".");
            System.exit(0)
        }

        System.out.println("Records sent to file " +
                           fileName + ".");
        System.out.println(
              "Now let's reopen the file and echo " +
              "the records.");
```

(continued)

LISTING 10.10 File I/O of Class Objects *(part 2 of 3)*

```
        ObjectInputStream inputStream = null;

        try
        {
            inputStream = new ObjectInputStream(
                          new FileInputStream("species.records"));
        }
        catch(IOException e)
        {
            System.out.println("Error opening input file " +
                               fileName + ".");
            System.exit(0);
        }
        Species readOne = null, readTwo = null;

        try                                    Notice the type casts.
        {
            readOne = (Species)inputStream.readObject();
            readTwo = (Species)inputStream.readObject();
            inputStream.close();
        }
                                   A separate catch block for each type of
        catch(Exception e)         exception would be better. We use only one
        {                          to save space.
            System.out.println("Error reading from file " +
                               fileName + ".");
            System.exit(0);
        }

        System.out.println("The following were read\n" +
                           "from the file " + fileName + ".");
        System.out.println(readOne);
        System.out.println();
        System.out.println(readTwo);

        System.out.println("End of program.");
    }
}
```

Sample Screen Output

```
Records sent to file species.records.
Now let's reopen the file and echo the records.
The following were read
from the file species.records.
Name = Calif. Condor
Population = 27
Growth rate = 0.02%
```

(continued)

LISTING 10.10 File I/O of Class Objects *(part 3 of 3)*

```
Name = Black Rhino
Population = 100
Growth rate 1.0%
End of program.
```

write an object of the class `Species` to a binary file, we pass the object as the argument to the method `writeObject`, as in

```
outputStream.writeObject(oneSpecies);
```

where `outputStream` is a stream of type `ObjectOutputStream` and `oneSpecies` is of type `Species`.

REMEMBER Writing Objects to a Binary File

You can write objects to a binary file using `writeObject` only if their class is serializable. When checking the documentation for predefined classes, see whether the class implements `Serializable`.

`readObject` recovers an object in a binary file

An object written to a binary file with the method `writeObject` can be read by using the method `readObject` of the stream class `ObjectInputStream`, as illustrated by the following example taken from Listing 10.10:

```
readOne = (Species)inputStream.readObject();
```

Here, `inputStream` is a stream of type `ObjectInputStream` that has been connected to a file that was filled with data using `writeObject` of the class `ObjectOutputStream`. The data consists of objects of the class `Species`, and the variable `readOne` is of type `Species`. Note that `readObject` returns its value as an object of type `Object`. If you want to use it as an object of type `Species`, you must type cast it to the type `Species`.

Before we leave this example, let's clarify a potential misconception. The class `Species` has a method `toString`. This method is needed so that output to the screen and output to a text file—using `println`—appears correctly. However, the method `toString` has nothing to do with object I/O to a binary file. Object I/O with a binary file would work fine even if a class did not override `toString`.

Some Details of Serialization

Our introduction to serialization in the previous section omitted some details that we must cover now, in order to arrive at a complete definition of a serializable class. Recall that we mentioned that an instance variable could be another object that itself could have an object as an instance variable. In fact, the class `Species` has a string as an instance variable. When a serializable class has instance variables

Properties of a serializable class

of a class type, the class for those instance variables should also be serializable, and so on for all levels of instance variables within classes.

Thus, a class is serializable if all of the following are true:

- The class implements the interface `Serializable`.
- Any instance variables of a class type are objects of a serializable class.
- The class's direct super class, if any, is either serializable or defines a default constructor.

For example, the class `Species` implements the interface `Serializable` and has a string as an instance variable. The class `String` is serializable. Since any subclass of a serializable class is serializable, a class derived from `Species` would also be serializable.

What is the effect of making a class serializable? In a sense, there is no direct effect on the class, but there is an effect on how Java performs file I/O with objects of the class. If a class is serializable, Java assigns a **serial number** to each object of the class that it writes to a stream of type `ObjectOutputStream`. If the same object is written to the stream more than once, after the first time, Java writes only the serial number for the object, rather than writing the object's data multiple times. This feature makes file I/O more efficient and reduces the size of the file. When the file is read using a stream of type `ObjectInputStream`, duplicate serial numbers are returned as references to the same object. Note that this condition means that, if two variables contain references to the same object, and you write the objects to the file and later read them from the file, the two objects that are read will be references to the same object. So nothing in the structure of your object data is lost when you write the objects to the file and later read them.

Some classes aren't serializable for security reasons

Serializability sounds great. So why aren't all classes made serializable? In some cases, it is for security reasons. The serial-number system makes it easier for programmers to get access to the object data written to secondary storage. In other cases, writing objects to secondary storage may not make sense, since they would be meaningless when read again later. For example, if the object contains system-dependent data, the data may be meaningless later.

Array Objects in Binary Files

Arrays are objects, and so can be written to a binary file

Since Java treats arrays as objects, you can use `writeObject` to write an entire array to a binary file, and you can use `readObject` to read it from the file. When doing so, if the array has a base type that is a class, the class should be serializable. This means that, if you store all your data for one serializable class in a single array, you can write all your data to a binary file using one invocation of `writeObject`.

For example, suppose that `group` is an array of `Species` objects. If `toFile` is an instance of `ObjectOutputStream` that is associated with a binary file, we can write the array to that file by executing the statement

```
toFile.writeObject(group);
```

After creating the file, we can read the array by using the statement

```
Species[] myArray = (Species[])fromFile.readObject();
```

where `fromFile` is an instance of `ObjectInputStream` that is associated with the file that we just created.

Note that the base-class type, `Species`, is serializable. Note also the type cast when reading the array from the file. Since `readObject` returns its value as an object of type `Object`, it must be type cast to the correct array type, `Species[]` in this case.

Listing 10.11 contains a sample program that writes and reads an array using a binary file. Notice that we declared the arrays `oneArray` and `anotherArray` outside of the `try` blocks so that they exist both outside and

LISTING 10.11 File I/O of an Array Object *(part 1 of 2)*

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class ArrayIODemo
{
    public static void main(String[] args)
    {
        Species[] oneArray = new Species[2];
        oneArray[0] = new Species("Calif. Condor", 27, 0.02);
        oneArray[1] = new Species("Black Rhino", 100, 1.0);

        String fileName = "array.dat";

        try
        {
            ObjectOutputStream outputStream =
                  new ObjectOutputStream(
                      new FileOutputStream(fileName));
            outputStream.writeobject(oneArray);
            outputStream.close();
        }
        catch(IOException e)
        {
            System.out.println("Error writing to file " +
                               fileName + ".");
            System.exit(0);
        }
        System.out.println("Array written to file " +
                           fileName + " and file is closed.");
```

(continued)

LISTING 10.11 File I/O of an Array Object *(part 2 of 2)*

```
        System.out.println("Open the file for input and " +
                           "echo the array.")
        Species[] anotherArray = null;
        try
        {
            ObjectInputStream inputStream =
                        new objectInputStream(
                                new FileInputStream(fileName));
            anotherArray = (Species[])inputStream.readobject();
            inputStream.close();
        }
        catch(Exception e)
        {
            System.out.println("Error reading file " +
                                fileName + ".");
            System.exit(0);
        }
        System.out.println("The following were read from " +
                           "the file " + fileName + ":");
        for (int i = 0; i < anotherArray.length; i++)
        {
            System.out.println(anotherArray[i]);
            System.out.println();
        }
        System.out.println("End of program.");
    }
}
```

Note the type cast.

A separate `catch` *block for each type of exception would be better. We use only one to save space.*

Sample Screen Output

```
Array written to file array.dat and file is closed.
Open the file for input and echo the array.
The following were read from the file array.dat:
Name = Calif. Condor
Population = 27
Growth rate = 0.02%

Name = black Rhino
Population = 100
Growth rate = 1.0%

End of program.
```

inside these blocks. Also note that `anotherArray` is initialized to `null`, not `newSpecies[2]`. Had we allocated a new array here, it would have been replaced by the subsequent call to `readObject`. That is, `readObject` creates a new array; it does not fill an existing one.


VideoNote
Using a binary file of objects and arrays

SELF-TEST QUESTIONS

30. How do you make a class serializable?

31. What is the return type for the method `readObject` of the class `ObjectInputStream`?

32. What exception(s) might be thrown by the method `writeObject` of the class `ObjectOutputStream`? (Consult the documentation for the Java Class Library.)

33. What exception(s) might be thrown by the method `readObject` of the class `ObjectInputStream`? (Consult the documentation for the Java Class Library.)

10.6 NETWORK COMMUNICATION WITH STREAMS


VideoNote
Networking with streams

Since in order to speak, one must first listen, learn to speak by listening.

—MEVLANA RUMI

We have seen that the `Scanner` class can be used to read data from a file or from the keyboard. Amazingly, classes such as `Scanner` and `PrintWriter` can be used with any kind of data stream! To illustrate this flexibility, here we introduce communication over a network using streams. As you will see, the code is nearly identical to reading from or writing to a file except instead the program will be reading from or sending to a computer on the Internet. This level of abstraction is made possible thanks to the principle of polymorphism discussed in Chapter 8.

When computers want to communicate with one another over a network, each computer must speak the same language, or protocol. The most common protocol today is TCP/IP, or the Transmission Control Protocol and Internet Protocol. It is used as the basis for most Internet communications. TCP is a stream-based protocol in which packets of data are transmitted from the sender to the receiver. TCP is considered a reliable protocol because it guarantees that data from the sender is received in the same order in which it was sent. The program that is waiting for a connection is called the server and the program that initiates the connection is called the client. An alternate

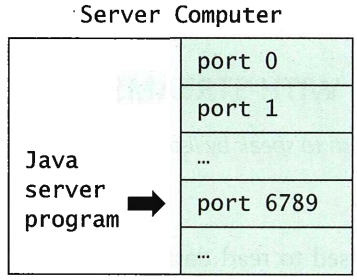
protocol is UDP, or the User Datagram Protocol. In UDP, packets of data are transmitted but no guarantee is made regarding the order in which the packets are received or if they will even be received. Although Java provides support for UDP, we will only introduce TCP in this section.

Network programming is implemented in Java using sockets. A socket describes one end of the connection between two programs over the network. A socket consists of an address that identifies the remote computer and a port for both the local and remote computer. The port is assigned an integer value between 0 and 65,535 that is used to identify which program should handle data received from the network. Two applications may not bind to the same port. Typically, ports 0 to 1,024 are reserved for use by well-known services implemented by your operating system.

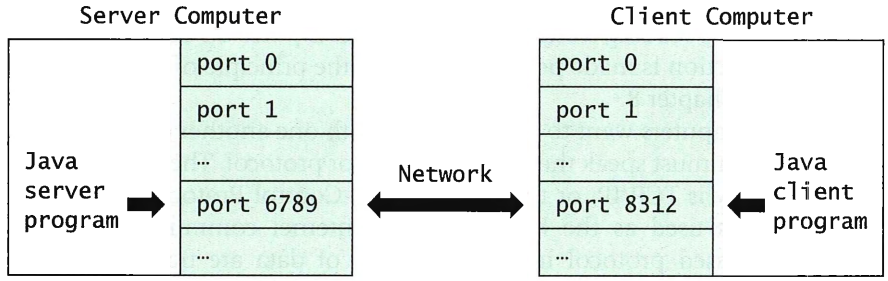
The process of client/server communication is shown in Figure 10.7. First, the server waits for a connection by listening on a specific port. When a client connects to this port, a new socket is created that identifies the remote

FIGURE 10.7 Client/Server Network Communication via Subjects

1. The Java server program listens and waits for a connection on port 6789. Different programs may be listening on other ports.



2. The Java client program connects to the server on port 6789. It uses a local port that is assigned automatically, in this case, port 8312.



3. The Java server program can now communicate over a socket bound locally to port 6789 and remotely to the client's address at port 8312, while the client communicates over a socket bound locally to port 8312 and remotely to the server's address at port 6789.

computer, the remote port, and the local port. A similar socket is created on the client. Once the sockets are created on both the client and the server, data can be transmitted using streams in the same way we read from and write to files.

Listing 10.12 shows how to create a simple server that listens on port 6789 for a connection. Once it receives a connection a new socket is returned by the `accept()` method. From this socket we create a `Scanner` and `PrintWriter`, just as if we were reading from a text file except the source of the stream comes from the socket. The `ServerSocket` and `Socket` classes are in the `java.net` package. Once the streams are created, the server expects the client to send a line of text. The server waits for the name with a call to `nextLine()` just like we would use to read from the keyboard or file, except this time the server waits for the text to come from the network. Once the text is received the server sends back a message. We should *flush* the stream when we want the data to be sent. Finally, the server closes the streams and sockets.

LISTING 10.12 Network Server Program

```
import java.util.Scanner;
import java.io.InputStreamReader;
import java.io.DataOutputStream;
import java.io.PrintWriter;
import java.net.Socket;
import java.net.ServerSocket;

public class SocketServer
{
 public static void main(String[] args)
 {
  String s;
  Scanner inputStream = null;
  PrintWriter outputStream = null;
  ServerSocket serverSocket = null;
  try
  {
        // Wait for connection on port 6789
        System.out.println("Waiting for a connection.");
        serverSocket = new ServerSocket(6789);
        Socket socket = serverSocket.accept();
        // Connection made, set up streams
        inputStream = new Scanner(new
           InputStreamReader(socket.getInputStream()));
             outputStream = new PrintWriter(new
           DataOutputStream(socket.getOutputStream()));

        // Read a line from the client
        s = inputStream.nextLine();
```

```
    // Output text to the client
    outputStream.println("Well, ");
    outputStream.println(s +
            " is a fine programming language!");
    outputStream.flush();
    System.out.println("Closing connection from "
              + s);
    inputStream.close();
    outputStream.close();
    }
    catch (Exception e)
    {
       // If any exception occurs, display it
       System.out.println("Error " + e);
    }
  }
}
```

After running the client program in Listing 10.13.

Screen Output

```
Waiting for a connection.
Closing connection from Java
```

Listing 10.13 shows how to create a client that connects to our server. First, we create a socket with the name of the computer running the server along with the corresponding port of 7654. If the server program and client program are running on the same computer then you can use *localhost* as the name of the machine. Otherwise, the hostname should be set to the name of the computer (e.g., my.server.com). After a connection is made, the client creates stream objects, sends its name, waits for a reply, and prints the reply.

LISTING 10.13 Network Client Program

```
import java.util.Scanner;
import java.io.InputStreamReader;
import java.io.DataOutputStream;
import java.io.PrintWriter;
import java.net.Socket;

public class SocketClient
{
```

```
public static void main(String[] args)
{
 String s;
 Scanner inputStream = null;
 PrintWriter outputStream = null;
 try
 {
  // Connect to server on same machine, port 6789
  Socket clientSocket = new Socket("localhost",6789);
  // Set up streams to send/receive data
  inputStream = new Scanner(new
    InputStreamReader(clientSocket.getInputStream()));
  outputStream = new PrintWriter(new
    DataOutputStream(clientSocket.getOutputStream()));

  // Send "Java" to the server
  outputStream.println("Java");
  outputStream.flush(); // Sends data to the stream

  // Read everything from the server until no
  // more lines and output it to the screen
  while (inputStream.hasNextLine())
  {
    s = inputStream.nextLine();
    System.out.println(s);
  }
  inputStream.close();
  outputStream.close();
 }
 catch (Exception e)
 {
  // If any exception occurs, display it
  System.out.println("Error " + e);
 }
}
}
```

After connecting to the server program in Listing 10.12.

Screen Output

```
Well,
Java is a fine programming language!
```

Note that the socket and stream objects throw checked exceptions. This means that their exceptions must be caught or declared in a throws block. If you run the server program in Listing 10.12 then you will notice that the server waits, or blocks, at the `serverSock.accept()` call until a client connects to it. The server also blocks at the `nextLine()` call if data from the socket is not yet available. This behavior makes it difficult for a server to handle connections with more than one client. After a connection is made with the first client, the server will not respond if a second client wishes to connect. The solution to this problem is to use threads. Threads allow a program to run concurrently, which is like multiple copies of a program running simultaneously. In a thread-based solution, each thread on the server program would handle communication with a different client that connects to it. Threads are not covered in this book but are discussed in more advanced texts.

As one final example to illustrate the flexibility of streams and the power of polymorphism, Java has an `URL` class that can read the contents of a website into a stream. To use the class, import `java.net.URL`, create the URL, and then use the stream when creating a `Scanner` object. From that point on, reading from the `Scanner` will read data from the URL. A code snippet is shown below that would output the HTML of www.wikipedia.org.

```
URL website = new
      URL("http://www.wikipedia.org");
Scanner inputStream = new Scanner(
      new InputStreamReader(
website.openStream()));

while (inputStream.hasNextLine())
{
  String s = inputStream.nextLine();
  System.out.println(s);
}
inputStream.close();
```

SELF-TEST QUESTIONS

34. What is the purpose of a port in the context of a socket?

35. Modify the client so it sends two numbers, then reads a number and prints it. Modify the server so it reads two numbers, computes the sum, then sends the sum back to the client.

10.7 GRAPHICS SUPPLEMENT

The "real world" is the only world.

—GRAFFITI

Graphical user interfaces, or GUIs, are not just for cute little demonstrations. GUIs are meant to be interfaces for real application programs. This section provides one small example of a `JFrame` GUI that manipulates files.

PROGRAMMING EXAMPLE A JFrame GUI for Manipulating Files

The program in Listing 10.14 uses a `JFrame` to create a GUI that lets you do some housekeeping with your text files. The GUI has three buttons and two text fields. You can type a file name in the indicated text field. If you then click the `Show first line` button, the first line of the file is displayed in the other text field as a reminder of what is in the file. However, if the file does not exist or is not readable, you get a message explaining the situation instead. And if the file is not a text file, you will get gibberish.

LISTING 10.14 A File Organizer GUI *(part 1 of 5)*

```
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JTextField;
import java.awt.Color;
import java.awt.Container;
import java.awt.FlowLayout;
import java.awt.Graphics;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class FileOrganizer extends JFrame implements ActionListener
{
    public static final int WIDTH = 400;
    public static final int HEIGHT = 300;
    public static final int NUMBER_OF_CHAR = 30;

    private JTextField fileNameField;
    private JTextField firstLineField;

    public FileOrganizer()
    {
```

(continued)

LISTING 10.14 A File Organizer GUI *(part 2 of 5)*

```
        setSize(WIDTH, HEIGHT);
        WindowDestroyer listener = new WindowDestroyer();
        addWindowListener(listener);
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());

        JButton showButton = new JButton("Show first line");
        showButton.addActionListener(this);
        contentPane.add(showButton);

        JButton removeButton = new JButton("Remove file");
        removeButton.addActionListener(this);
        contentPane.add(removeButton);

        JButton resetButton = new JButton("Reset");
        resetButton.addActionListener(this);
        contentPane.add(resetButton);

        fileNameField = new JTextField(NUMBER_OF_CHAR);
        contentPane.add(fileNameField);
        fileNameField.setText("Enter file name here.");

        firstLineField = new JTextField(NUMBER_OF_CHAR);
        contentPane.add(firstLineField);
}
public void actionPerformed(ActionEvent e)
{
    String actionCommand = e.getActionCommand();
    if (actionCommand.equals("Show first line"))
        showFirstLine();
    else if (actionCommand.equals("Remove file"))
            removeFile();
    else if (actionCommand.equals("Reset"))
        resetFields();
    else
        firstLineField.setText("Unexpected error.");
}
private void showFirstLine()
{
    Scanner fileInput = null;
    String fileName = fileNameField.getText();
    File fileObject = new File(fileName);

    if (!fileObject.exists())
        firstLineField.setText("No such file");
    else if (!fileObject.canRead())
        firstLineField.setText("That file is not readable.");
    else
    {
```

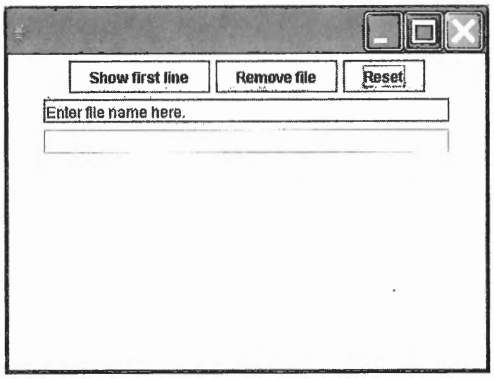
(continued)

LISTING 10.14 A File Organizer GUI *(part 3 of 5)*

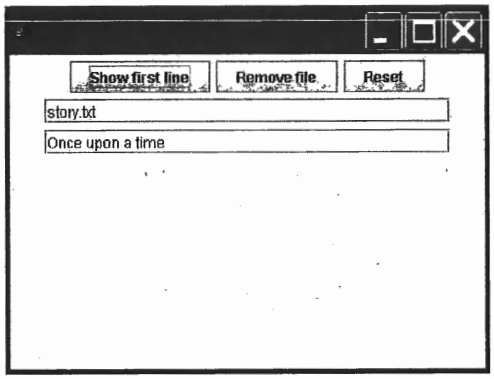
```
        try
        {
            fileInput = new Scanner(fileObject);
        }
        catch(FileNotFoundException e)
        {
            firstLineField.setText("Error opening the file " +
                                   fileName);
        }
        String firstLine = fileInput.nextLine();
        firstLineField.setText(firstLine);
        fileInput.close();
    }
}
private void resetFields()
{
    fileNameField.setText("Enter file name here.");
    firstLineField.setText("");
}
      private void removeFile()
      {
          Scanner fileInput = null;
          String firstLine;
          String fileName = fileNameField.getText();
          File fileObject = new File(filename);

          if (!fileObject.exists())
              firstLinefield.setText("No such file");
          else if (!fileObject.canWrite())
              firstLineField.setText("Permission denied.");
          else
          {
              if (fileObject.delete())
                  firstLineField.setText("File deleted.");
              else
                  firstLineField.setText("Could not delete file.");
          }
      }
      public static void main(String[] args)
      {
          FileOrganizer gui = new FileOrganizer();
          gui.setVisible(true);

      }
  }
```

LISTING 10.14 A File Organizer GUI *(part 4 of 5)*

Screen Output Showing GUI's State Initially or After the Reset Button is Clicked

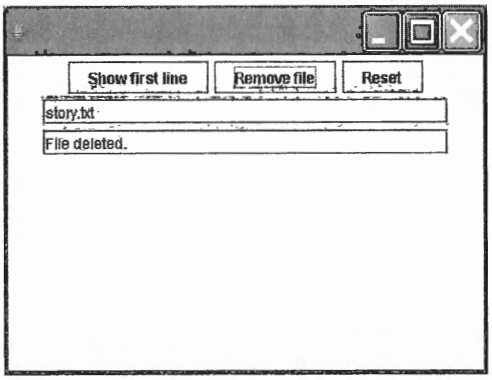


Screen Output After Entering the File Name and Clicking the Show first line Button



Assumes that the first line in the file is as shown

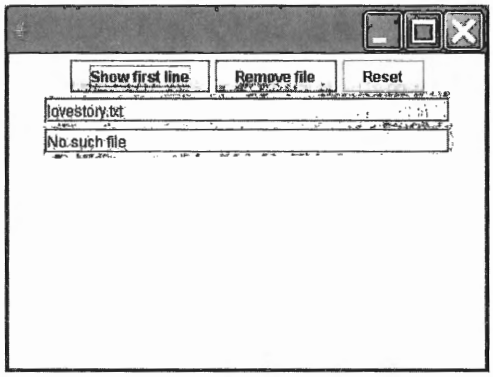
Screen Output After Entering the Remove line Button



(continued)

LISTING 10.14 A File Organizer GUI *(part 5 of 5)*

Screen Output After Entering the File Name and Clicking the Show first line Button



Assumes that the named file does not exist

If you want to delete a file, you type the file name in the text field and click the **Remove file** button. As long as the file exists and is writable, it is deleted and a confirmation message is displayed. If the file does not exist or is not writable—which normally also means you cannot delete it—an appropriate message appears instead. Finally, the **Reset** button restores the text fields to their original contents. Let's look at some code.

Within the constructor of our class, we define the three buttons and the two text fields. Having two text fields means that the file name can remain visible while any output is given in the other text field. To differentiate the text fields, we initialize the one for the file name as follows:

```
fileNameField.setText("Enter file name here.");
```

Chapter 13, which is on the Web, shows how to label a text field so that the label is outside the text field rather than in it.

The method `actionPerformed` identifies the button click and takes the appropriate action by calling one of three private methods within our class. The method `showFirstLine` retrieves the file name from the text field and, after checking that the file exists and is readable, opens the file for input. The method then reads the first line, displays it in the text field, and finally closes the file. The statements for these latter three steps are

```
String firstLine = fileInput.readLine();
firstLineField.setText(firstLine);
fileInput.close();
```

Before entering another file name, the user should click the `Reset` button, which sets the two text fields as follows:

```
fileNameField.setText("Enter file name here.");
firstLineField.setText("");
```

The second statement sets the text field contents to the empty string, which sets the field to blank. (The GUI will give correct answers even if the **Reset** button is not clicked, but the output will be clearer if it is clicked.) These two statements are in the private method `resetFields` that `actionPerformed` calls when the **Reset** button is clicked.

Finally, if the **Remove file** button is clicked, `actionPerformed` calls the private method `removeFile`. After checking that the file exists and is readable, `removeFile` deletes the file by using the following code:

```
if (fileObject.delete())
    firstLineField.setText("File deleted.");
else
    firstLineField.setText("Could not delete file.");
```

The invocation `fileObject.delete()` tries to delete the file. If `delete` is successful, it returns true; otherwise it returns false.

In this example, we have defined a `main` method within our class to create and display the GUI. Whether you choose to place `main` within this class or another one depends on the situation and to some extent on personal taste.

SELF-TEST QUESTIONS

36. The class `FileOrganizer` in Listing 10.12 includes a `main` method that creates and displays a GUI. Is it legal to instead do this within a `main` method of a separate class without deleting the `main` method in the class `FileOrganizer`?

37. In Listing 10.12, could you avoid having to catch exceptions simply by adding a `throws` clause to all methods that might throw an exception?

GOTCHA An Applet Cannot Manipulate Files on Another Computer

For reasons of security, an applet is limited in what it can do. Since an applet is designed to be embedded in a Web page and run from another computer some place else on the World Wide Web, the Java designers did not give applets the ability to manipulate files on a remote computer. Hence, you cannot write an applet version of the `JFrame` GUI in Listing 10.14 that will read and delete files on the computer on which it is viewed. You can, however, write an applet version that, when run with the applet viewer on your own computer, will manipulate files on your computer. In that case, however, you may as well use a `JFrame` GUI.

PROGRAMMING TIP When to Define GUI Components as Instance Variables

When GUI components must be available to more than one method within a class, defining them as instance variables is more convenient than passing them as arguments to several methods. In Listing 10.14, for example, two text fields—`fileNameField` and `firstLineField`—are created and initialized by the constructor and used in all of the other methods in the class except `main`. These text fields are defined as instance variables.

However, notice the buttons that the constructor defines. Since they are used only in the constructor, they can be local to that method. Although the method `actionPerformed` is responsible for a button's action, it does not reference any buttons by name. Instead, an action command is obtained from the action event that is generated when a button is clicked. This action command is dealt with entirely within `actionPerformed`. There is no reason in this case for the buttons to be instance variables. ■

CHAPTER SUMMARY

- Files that are considered to be strings of characters and that look like characters to your program and to a text editor are called text files. All other files are called binary files.
- Your program can use the class `PrintWriter` to write to a text file and can use the class `Scanner` to read from a text file.
- When reading from a file, you should always check for the end of a file and do something appropriate if the end of the file is reached. The way you test for the end of a file depends on whether your program is reading from a text file or a binary file.
- You can read a file name from the keyboard into a variable of type `String` and use that variable in place of a file name.
- The class `File` can be used to see whether an existing file has a given name. It can also be used to see whether your program is allowed to read the file or write to the file.
- Your program can use the class `ObjectOutputStream` to write to a binary file and can use the class `ObjectInputStream` to read from a binary file.
- Your program can use the method `writeObject` of the class `ObjectOutputStream` to write class objects or arrays to a binary file. Objects or arrays can be read from a binary file using the method `readObject` of the class `ObjectInputStream`.
- In order to use the methods `writeObject` of the class `ObjectOutputStream` and `readObject` of the class `ObjectInputStream`, the class whose objects are written to a binary file must implement the `Serializable` interface.
- You can connect a `PrintWriter` or `Scanner` to a `socket` to read and write data over a network.

Exercises

1. Write a program that will write the Gettysburg Address to a text file. Place each sentence on a separate line of the file.
2. Modify the program in the previous exercise so that it reads the name of the file from the keyboard.
3. Write some code that asks the user to enter either of the words append or new. According to the user response, open either an existing text file to which data can be appended or a new, empty text file to which data can be written. In either case, the file's name is a string given by the variable `fileName`.
4. Write a program that will record the purchases made at a store. For each purchase, read from the keyboard an item's name, its price, and the number bought. Compute the cost of the purchase (number bought times price), and write all this data to a text file. Also, display this information and the current total cost on the screen. After all items have been entered, write the total cost to both the screen and the file. Since we want to remember all purchases made, you should append new data to the end of the file.
5. Modify the class `LapTimer`, as described in Exercise 13 of the previous chapter, as follows:

   - Add an attribute for a file stream to which we can write the times
   - Add a constructor

   ```
   LapTimer(n, person, fileName)
   ```

   for a race having *n* laps. The name of the person and the file to record the times are passed to the constructor as strings. The file should be opened and the name of the person should be written to the file. If the file cannot be opened, throw an exception.
6. Write a class `TelephoneNumber` that will hold a telephone number. An object of this class will have the attributes

   - `areaCode`—a three-digit integer
   - `exchangeCode`—a three-digit integer
   - `number`—a four-digit integer

   and the methods

   - `TelephoneNumber(aString)`—a constructor that creates and returns a new instance of its class, given a string in the form xxx-xxx-xxxx or, if the area code is missing, xxx-xxxx. Throw an exception if the format is not valid. (*Hint*: To simplify the constructor, you can replace each hyphen in the telephone number with a blank. To accept a telephone number containing hyphens, you could process the string one character at a time or learn how to use `Scanner` to read words separated by a character—such as a hyphen—other than whitespace.)

- `toString`—returns a string in either of the two formats shown previously for the constructor.

Using a text editor, create a text file of several telephone numbers, using the two formats described previously. Write a program that reads this file, displays the data on the screen, and creates an array whose base type is `TelephoneNumber`. Allow the user to either add or delete one telephone number. Write the modified data on the text file, replacing its original contents. Then read and display the numbers in the modified file.

7. Write a class `ContactInfo` to store contact information for a person. It should have attributes for a person's name, business phone, home phone, cell phone, e-mail address, and home address. It should have a `toString` method that returns this data as a string, making appropriate replacements for any attributes that do not have values. It should have a constructor `ContactInfo(aString)` that creates and returns a new instance of the class, using data in the string `aString`. The constructor should use a format consistent with what the `toString` method produces.

   Using a text editor, create a text file of contact information, as described in the previous paragraph, for several people. Write a program that reads this file, displays the data on the screen, and creates an array whose base type is `ContactInfo`. Allow the user to do one of the following: change some data in one contact, add a contact, or delete a contact. Finally, write over the file with the modified contacts.

8. Write a program that reads every line in a text file, removes the first word from each line, and then writes the resulting lines to a new text file.

9. Repeat the previous exercise, but write the new lines to a new binary file instead of a text file.

10. Write a program that will make a copy of a text file, line by line. Read the name of the existing file and the name of the new file—the copy—from the keyboard. Use the methods of the class `File` to test whether the original file exists and can be read. If not, display an error message and abort the program. Similarly, see whether the name of the new file already exists. If so, display a warning message and allow the user to either abort the program, overwrite the existing file, or enter a new name for the file.

11. Suppose you are given a text file that contains the names of people. Every name in the file consists of a first name and last name. Unfortunately, the programmer who created the file of names had a strange sense of humor and did not guarantee that each name was on a single line of the file. Read

this file of names and write them to a new text file, one name per line. For example, if the input file contains

```
Bob Jones Fred
Charles Ed
Marston
Jeff
Williams
```

the output file should be

```
Bob Jones
Fred Charles
Ed Marston
Jeff Williams
```

12. Suppose that you have a binary file that contains numbers whose type is either `int` or `double`. You don't know the order of the numbers in the file, but their order is recorded in a string at the beginning of the file. The string is composed of the letters *i* (for `int`) and *d* (for `double`) in the order of the types of the subsequent numbers. The string is written using the method `writeUTF`.

    For example, the string `"iddiiddd"` indicates that the file contains eight values, as follows: one integer, followed by two doubles, followed by two integers, followed by three doubles. Read this binary file and create a new text file of the values, written one to a line.

13. Suppose that we want to store digitized audio information in a binary file. An audio signal typically does not change much from one sample to the next. In this case, less memory is used if we record the change in the data values instead of the actual data values. We will use this idea in the following program.

    Write a program `StoreSignal` that will read positive integers, each of which must be within 127 of the previous integer, from the keyboard (or from a text file, if you prefer). Write the first integer to a binary file. For each subsequent integer, compute the difference between it and the integer before it, cast the difference to a `byte`, and write the result to the binary file. When a negative integer is encountered, stop writing the file.

14. Write a program `RecoverSignal` that will read the binary file written by `StoreSignal`, as described in the previous exercise. Display the integer values that the data represents on the screen.

15. Even though a binary file is not a text file, it can contain embedded text. To find out if this is the case, write a program that will open a binary file and

read it one byte at a time. Display the integer value of each byte as well as the character, if any, that it represents in ASCII.

Technical details: To convert a byte to a character, use the following code:

```
char[] charArray = Character.toChars(byteValue);
```

The argument `byteValue` of the method `toChars` is an `int` whose value equals that of the byte read from the file. The character represented by the byte is `charArray[0]`. Since an integer is four bytes, `byteValue` can represent four ASCII characters. The method `toChars` tries to convert each of the four bytes to a character and places them into a `char` array. We are interested in just the character at index 0. If a byte in the file does not correspond to a character, the method will throw an `IllegalArgumentException`. If the exception is thrown, display only the byte value and continue on to the next byte.

PRACTICE PROGRAMS

Practice Programs can generally be solved with a short program that directly applies the programming principles presented in this chapter.


VideoNote
Processing a file

1. Write a program that searches a file of numbers and displays the largest number, the smallest number, and the average of all the numbers in the file. Do not assume that the numbers in the file are in any special order. Your program should obtain the file name from the user. Use either a text file or a binary file. For the text-file version, assume one number per line. For the binary-file version, use numbers of type `double` that are written using `writeDouble`.

2. Write a program that reads a file of numbers of type `int` and writes all the numbers to another file, but without any duplicate numbers. Assume that the numbers in the input file are already ordered from smallest to largest. After the program is run, the new file will contain all the numbers in the original file, but no number will appear more than once in the file. The numbers in the output file should also be sorted from smallest to largest. Your program should obtain both file names from the user. Use either a text file or a binary file. For the text-file version, assume one number per line. For the binary-file version, use numbers of type `int` that are written using `writeInt`.

3. The following is an old word puzzle: "Name a common word, besides tremendous, stupendous, and horrendous, that ends in dous." If you think about this for a while it will probably come to you. However, we can also solve this puzzle by reading a text file of English words and outputting the word if it contains "dous" at the end. The text file "words.txt" contains

VideoNote
Solution to Practice Program 3

87314 English words, including the word that completes the puzzle. This file is available online with the source code for the book. Write a program that reads each word from the text file and outputs only those containing "dous" at the end to solve the puzzle.

4. The Social Security Administration maintains an actuarial life table that contains the probability that a person in the United States will die (http://www.ssa.gov/OACT/STATS/table4c6.html). The death probabilities from this table for 2009 are stored in the file `LifeDeathProbability.txt` that is included with the book's source code. There are three values for each row the age, death probability for a male, and death probability for a female. For example, the first five lines are:

```
0  0.006990 0.005728
1  0.000447 0.000373
2  0.000301 0.000241
3  0.000233 0.000186
4  0.000177 0.000150
```

This says that a 3-year-old female has a 0.000186 chance of dying that year.

Write a program that simulates how long you will live. The basic idea is to generate random numbers for each age until you "die." For example, if you are a 1-year-old male, then if a random number is <= 0.000447 then the simulation will say you will live to age 1. Otherwise if the random number is > 0.000447 then go to age 2 and generate another random number. If it is < 0.000301 then the simulation will say you will live to age 2, etc. The program should input your sex and age to determine which probabilities to use. The program should then simulate to what age you will live by starting with the death probability for the age and sex entered in the file. If the simulation reaches age 120 then stop and predict that the user will live to 120. This program is merely a simulation and will give different results each time it is run.

5. The following is a list of scores for a game. Enter them into a text file.

```
14401
3094
39201
57192
4948
55854
84
95
430
5502
65816
4994
7712
```

Write a program that opens this file, reads in each score as an integer, and remembers the highest score. After all the scores have been read, output the high score (65816 in this example, but it could be different if the file had different scores in it).

PROGRAMMING PROJECTS

Programming Projects require more problem-solving than Practice Programs and can usually be solved many different ways.

1. Write a program that checks a text file for several formatting and punctuation matters. The program asks for the names of both an input file and an output file. It then copies all the text from the input file to the output file, but with the following two changes: (1) Any string of two or more blank characters is replaced by a single blank; (2) all sentences start with an uppercase letter. All sentences after the first one begin after either a period, a question mark, or an exclamation mark that is followed by one or more whitespace characters.

2. Write a program similar to the one in Listing 10.10 that can write an arbitrary number of `Species` objects to a binary file. (`Species` appears in Listing 5.19 of Chapter 5.) Read the file name and the data for the objects from a text file that you create by using a text editor. Then write another program that can search a binary file created by your first program and show the user the data for any requested endangered species. The user gives the file name and then enters the name of the species. The program either displays all the data for that species or gives a message if that species is not in the file. Allow the user to either enter additional species' names or quit.

3. Write a program that reads from a file created by the program in the previous programming project and displays the following information on the screen: the data for the species having the smallest population and the data for the species having the largest population. Do not assume that the objects in the file are in any particular order. The user gives the file name.

4. Programming Project 2 asks you, among other things, to write a program that creates a binary file of objects of the class `Species`. Write a program that reads from a file created by that program and writes the objects to another file after modifying their population figures as they would be in 100 years. Use the method `predictPopulation` of the class `Species`, and assume that you are given each species' growth rate.

5. Text messaging is a popular means of communication. Many abbreviations are in common use but are not appropriate for formal communication.

Suppose the abbreviations are stored, one to a line, in a text file named `abbreviations.txt`. For example, the file might contain these lines:

```
lol
:)
iirc
4
u
ttfn
```

Write a program that will read a message from another text file and surround each occurrence of an abbreviation with <> brackets. Write the marked message to a new text file.

For example, if the message to be scanned is

> How are u today? Iirc, this is your first free day. Hope you are having fun! :)

the new text file should contain

> How are <u> today? <Iirc>, this is your first free day. Hope you are having fun! <:)>

6. Modify the `TelephoneNumber` class described in Exercise 6 so that it is serializable. Write a program that creates an array whose base type is `TelephoneNumber` by reading data from the keyboard. Write the array to a binary file using the method `writeObject`. Then read the data from the file using the method `readObject` and display the information to the screen. Allow the user to change, add, or delete any telephone number until he or she indicates that all changes are complete. Then write the modified telephone numbers to the file, replacing its original contents.

7. Revise the class `Pet`, as shown in Listing 6.1 of Chapter 6, so that it is serializable. Write a program that allows you to write and read objects of type `Pet` to a file. The program should ask the user whether to write to a file or read from a file. In either case, the program next asks for the file name. A user who has asked to write to a file can enter as many records as desired. A user who has asked to read from a file is shown all of the records in the file. Be sure that the records do not scroll by so quickly that the user cannot read them. (*Hint*: Think of a way to pause the program after a certain number of lines are displayed.)

8. Write a program that reads records of type `Pet` from a file created by the program described in the previous programming project and displays the following information on the screen: the name and weight of the heaviest pet, the name and weight of the lightest pet, the name and age of the youngest pet, and the name and age of the oldest pet.

9. The UC Irvine Machine Learning repository contains many datasets for conducting computer science research. One dataset is the Haberman's Survival dataset, available at http://archive.ics.uci.edu/ml/datasets/Haberman's+Survival and also included online with the source code for the book. The file "haberman.data" contains survival data for breast cancer patients in comma-separated value (CSV) format. The first field is the patient's age at the time of surgery, the second field is the year of the surgery, the third field is the number of positive axillary nodes detected, and the fourth field is the survival status. The survival status is 1 if the patient survived 5 years or longer and 2 if the patient died within 5 years.

   Write a program that reads the CSV file and calculates the average number of positive axillary nodes detected for patients who survived 5 years or longer, and the average number of positive axillary nodes detected for patients who died within 5 years. A significant difference between the two averages suggests whether or not the number of positive axillary nodes detected can be used to predict survival time. Your program should ignore the age and year fields for each record.

10. In many races competitors wear a RFID tag on their shoes or bibs. When the racer crosses a sensor a computer logs the racer's number along with the current time. Sensors can be placed along the course to accurately calculate the racer's finish time or pace and also to verify that the racer crossed key checkpoints. Consider such a system in use for a half-marathon running race, which is 13.1 miles. In this problem there are only three sensors: at the start, at the 7-mile point, and at the finish line.

    Here is sample data for three racers. The first line is the gun time in the 24-hour time format (HH MM SS). The gun time is when the race begins. Subsequent lines are recorded by sensors and contain the sensor ID (0=start, 1=midpoint, 2=finish) followed by the racer's number followed by the time stamp. The start time may be different than the gun time because sometimes it takes a racer a little while to get to the starting line when there is a large pack.

```
08 00 00
0,100,08 00 00
0,132,08 00 03
0,182,08 00 15
1,100,08 50 46
1,182,08 51 15
1,132,08 51 18
2,132,09 34 16
2,100,09 35 10
2,182,09 45 15
```

    Create a text file with a sample race log. Write a program that reads the log data into an appropriate array structure. The program should then allow a user to enter a racer's number and it should output the racer's

overall finish place, race split times in minutes/mile for each split (i.e., the time between sensors), and the overall race time and overall race pace. For a more challenging version modify your program so that it works with an arbitrary number of sensors placed at different locations along the course instead of just three locations. You will need to specify the mile marker for each sensor.

11. Based on the log file described in Programming Project 10 write a program to detect cheating. This could occur if:

    - A racer misses a sensor, which is a sign that the racer may have taken a shortcut.
    - A race split is suspiciously fast, which is a sign that the racer may have hopped in a vehicle. In this case, a race split faster than 4:30 per mile can be considered suspicious.

    The output should be a list of suspected cheaters along with the reason for suspicion.

12. Write a Java program that serves as a primitive web browser. For this assignment it merely needs to input a server name and display the HTML that is sent by the web server. A web server normally listens on port 80. Upon connection the server expects to be sent a string that identifies what web page to receive (use / for the root) and what protocol is used. The next line is the Host and then a blank line. For example, to get the default page on Wikipedia the Java program would connect to port 80 and send:

    ```
    GET / HTTP/1.1
    Host: www.wikipedia.org
    (blank line)
    ```

    The Wikipedia server would then send back the HTML for the site which your program should display in text. For a more challenging program, parse and render the HTML in a human-friendly format instead of printing out the raw HTML.

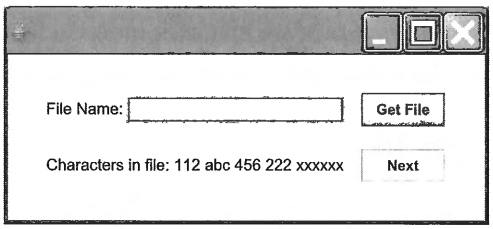
Graphics

13. Repeat any of the previous programming projects using a `JFrame` graphical user interface.

Graphics

14. Write an application or applet that uses a text field to get the name of a file, reads the file byte by byte, and displays the bytes as characters. (Exercise 15 describes how to convert a byte value to a character.) Display the first 20 characters in a label. If a byte does not correspond to a legal character, display a space instead. Clicking the `Next` button reads and displays the next 20 characters in the file. The GUI might look like the sketch in Figure 10.8.

FIGURE 10.8 GUI for Programming Project 13



15. Write an application or applet that implements a simple text editor. Use a text field and a button to get the file. Read the entire file as characters and display it in a `JTextArea`. The user will then be able to make changes in the text area. Use a **Save** button to get the contents of the text area and write that over the original file. Graphics

    Technical Details: Read each line from the file and then display the line in the text area using the method `append(aString)`. The method `getText` will return all of the text in the text area in a string that then can be written to the file. The following statement will make the text area referenced by `editorTextArea` scrollable:

    ```
    JScrollPane scrollPane = new
        JScrollPane(editorTextArea);
    ```

    Then add `scrollPane` to your GUI, just as you would any other component. The text area `editorTextArea` does not need to be added to the GUI, but it can be used.

Answers to Self-Test Questions

1. If a program sends its output to the screen, the output goes away when (or soon after) the program ends. If your program sends its output to a file, the file will remain after the program has finished running. A program that sends its output to a file has preserved its output for use long after the program ends its execution. The contents of a file remain until a person or program changes the file.

2. From a file to the program.

3. All files contain binary data. Text files can be thought of as a sequence of characters. You can write to and read a text file by using a text editor or your

own Java program. All other files are called binary files. Data in these files cannot conveniently be read using a text editor, but some standard binary files—such as certain music or picture files—can be created and read on any platform. Java binary files are platform independent, but other binary files might depend on the programming language and type of computer used to create them.

4. `PrintWriter outStream = new PrintWriter("sam.txt");`

5. ```
   PrintWriter out Stream = new
   PrintWriter(new FileOutputStream("sam.txt", true));
   ```

6. ```
   System.out.print("Enter file name: ");
   Scanner keyboard = new Scanner(System.in);
   String fileName = keyboard.next();
   PrintWriter outStream = new PrintWriter(fileName);
   ```

7. A `FileNotFoundException` would be thrown if the file could not be opened because, for example, there is already a directory (folder) named `out.txt`. Note that if the file does not exist but can be created, no exception is thrown. If you answered `IOException`, you are not wrong, because a `FileNotFoundException` is an `IOException`. However, the better answer is the more specific exception class, namely, `FileNotFoundException`.

8. `PrintWriter textStream = new PrintWriter("dobedo");`

9. `Scanner inputStream = new Scanner(new File("dobedo"));`

10. Both methods return a value of type `String`.

11. The code for the following program is included with the source code for this book available on the Web:

```
import java.io.File;
import java.util.Scanner;
public class Question11
{
    public static void main(String[] args)
    {
        System.out.print("Enter a file name and I will ");
        System.out.println("tell you whether it exists.");
        Scanner keyboard = new Scanner(System.in);
        String name = keyboard.next();
        File fileObject = new File(name);
        if (fileObject.exists())
        {
            System.out.println("I found the file " + name);
            System.out.println("Delete the file?");
            String ans = keyboard.next();
            if (ans.equalsIgnoreCase("yes"))
```

```
            {
                System.out.println("If you delete the file " +
                name);
                System.out.println("all data in the file " + "will
                be lost.");
                System.out.println("Delete?");
                ans = keyboard.next();
                if (ans.equalsIgnoreCase("yes"))
                {
                    if (fileObject.delete())
                        System.out.println("File deleted.");
                    else
                        System.out.println("Can't delete.");
                    }
                    else
                        System.out.println("File not deleted.");
                    }
                    else
                        System.out.println("File not deleted.");
            }
            else
                    System.out.println("I cannot find " + name);
        }
    }
```

12.
```
ObjectOutputStream toFile = new ObjectOutputStream(
    new FileOutputStream("stuff.data"));
```

13.
```
toFile.writeDouble(x1);
toFile.writeDouble(x2);
toFile.writeDouble(x3);
```

14.
```
toFile.close();
```

15.
```
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
```

16.
```
ObjectInputStream fromFile = new ObjectInputStream(
    new FileInputStream("stuff.data"));
```

17.
```
double x1 = fromFile.readDouble();
double x2 = fromFile.readDouble();
double x3 = fromFile.readDouble();
```

18.
```
fromFile.close();
```

19. No. If a number is written to a file using `writeInt`, it should be read only by `readInt`. If you use `readLong` or `readDouble` to read the number, something will go wrong.

20. You should not use `readUTF` to read a string from a text file. You should use `readUTF` only to read a string from a binary file. Moreover, the string should have been written to that file using `writeUTF`.

21. The following program is included with the source code for this book available on the Web:

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.ObjectInputStream;
import java.io.IOException;
import java.util.Scanner;
public class Question21
{
    public static void main(String[] args)
    {
        //To use fileName in a catch block,
        //declare it outside of the try block
        String fileName = null;
        try
        {
           System.out.println("Enter file name:");
           Scanner keyboard = new Scanner(System.in);
           fileName = keyboard.next();
           ObjectInputStream inputStream =
                          new ObjectInputStream(
                          new FileInputStream(fileName));
           System.out.println("The first thing in the file");
           System.out.println(fileName + " is");
           String first = inputStream.readUTF();
           System.out.println(first);
           inputStream.close();
        }
        catch(FileNotFoundException e)
        {
            System.out.println("Problem opening the file "+
                               "fileName);
        }
        catch(EOFException e)
        {
            System.out.println("Unexpected end of file.");
        }
        catch(IOException e)
        {
            System.out.println("Problem with input from "+
                               " file fileName);
        }
    }
}
```

22. It will not work because no constructor of `ObjectInputStream` has a parameter of type `String`. The correct way to accomplish the desired effect is by using

```
ObjectInputStream inputStream = new ObjectInputStream(
    new FileInputStream("mydata.dat"));
```

23. The class `FileInputStream` does not have any of the methods `readInt`, `readDouble`, or `readUTF`.

24. Yes.

25. No.

26. When opening a binary file for either output or input in the ways discussed in this chapter, a `FileNotFoundException` and other `IOExceptions` can be thrown.

27. An `EOFException` will be thrown.

28. Because when the end of the file is reached, an exception will be thrown, and that will end the entire `try` block.

29. The following program is included with the source code for this book available on the Web:

```
import java.io.ObjectInputStream;
import java.io.EOFException;
import java.io.IOException;
public class Question29
{
    public static final String FILE_NAME = "temperatures.dat";
    public static void main(String[] args)
    {
        try
        {
            ObjectInputStream inputStream =
                        new ObjectInputStream(
                            new FileInputStream(FILE_NAME));
            System.out.println("Numbersfromthefile " +
                               FILE_NAME + ":");
            try
            {
                while (true)
                {
                    double number = inputStream.readDouble();
                    System.out.println(number);
                }
            }
            catch(EOFException e)
```

```
                {
                    //Do nothing
                }
                System.out.println("End of reading from file.");
                inputStream.close();
            }
            catch(IOException e)
            {
                System.out.println("Problem reading from file.");
            }
        }
    }
```

30. You add the two words `implementsSerializable` to the heading of the class definition. Any objects that are instance variables of the class must belong to a serializable class.

31. The return type is `Object`, which means that the returned value usually needs to be type cast to its "true" class.

32. `InvalidClassException`, `NotSerializableException`, and `IOException`.

33. `ClassNotFoundException`, `InvalidClassException`, `StreamCorrupted Exception`, `OptionalDataException`, and `IOException`.

34. The port identifies the program on the computer that should get data received by the computer over the network.

35. The relevant code on the client should look something like:

```
outputStream.println(3);
outputStream.println(4);
outputStream.flush();
int sum = inputstream.nextInt();
System.out.println(sum);
```

The relevant code on the client should look something like:

```
int i = inputstream.nextInt ();
int j = inputstream.nextInt ();
outputStream.println(i+j);
outputStream.flush();
```

36. Yes, it is perfectly legal. Try it.

37. This would not be good style, but in this case it is not even possible. You can add the `throws` clause to the method `showFirstLine` and eliminate the `try` and `catch` blocks in that method. However, you would then have to catch the exception in the method `actionPerformed`. You cannot add a `throws` clause to the method `actionPerformed`. See the Gotcha section at the end of the graphics supplement of Chapter 9 entitled "A `throws` Clause Is Not Allowed in `actionPerformed`." There are no other `catch` blocks to eliminate.