Chapter 2: Using File Streams

In This Chapter

- **Understanding streams**
- **Reading text files**
- **Writing text files**
- **Reading binary files**
- **Writing binary files**

I/O, I/O, it's off to work I go.

Or so goes the classic song, which pretty much sums up the whole purpose of computers. Without I/O, computers — and the programs that run on them — would be worthless.

Imagining any useful computer program that doesn't do some form of I/O is hard. Even the very first program presented in this book — the classic Hello, World! program — does I/O: It displays a simple message on-screen.

In this chapter, you find out about Java's most fundamental technique for getting data into and out of programs: streams. You've been working with streams all along in this book. When you use the `System.out.print` or `System.out.println` method to display text on the console, you're actually sending data to an output stream. And when you use a `Scanner` object to get data from `System.in`, you're reading data from an input stream.

In this chapter, you build on what you already know about stream I/O and see how it can be used to read and write data to disk files.

Understanding Streams

A *stream* is simply a flow of characters to and from a program. The other end of the stream can be anything that can accept or generate a stream of characters, including a console window, a printer, a file on a disk drive, or even another program.

Streams have no idea of the structure or meaning of your data; a stream is just a sequence of characters. In later chapters in Book VIII, you find out how to work with data at a higher level, by using databases and XML.

You can roughly divide the world of Java stream I/O into two camps:

✦ **Character streams:** *Character streams* read and write text characters that represent strings. You can connect a character stream to a text file to store text data on disk. Typically, text files use special characters called *delimiters* to separate elements of the file. For example:

  • A *comma-delimited file* uses commas to separate individual fields of data.

  • A *tab-delimited file* uses tabs to separate fields.

  You can usually display a text file in a text editor and make some sense of its contents.

✦ **Binary streams:** *Binary streams* read and write individual bytes that represent primitive data types. You can connect a binary stream to a *binary file* to store binary data on disk. The contents of a binary file makes perfect sense to the programs that read and write them. However, if you try to open a binary file in a text editor, the file's contents look like gibberish.

Conceptually, the trickiest part of understanding how streams work is getting your mind around all the different classes. Java has more than 60 classes for working with streams. Fortunately, you only need to know about a few of them for most file I/O applications. In the rest of this chapter, I tell you about the most important classes for working with character and binary streams.

All the classes in this chapter are in the `java.io` package. So programs that work with file streams include an `import java.io.*` statement.

Reading Character Streams

To read a text file through a character stream, you usually work with the following classes:

✦ **`File`:** The `File` class, which is covered in detail in the preceding chapter, represents a file on disk. In file I/O applications, the main purpose of the `File` class is to identify the file you want to read from or write to.

✦ **`FileReader`:** The `FileReader` class provides basic methods for reading data from a character stream that originates from a file. It provides methods that let you read data one character at a time. You won't usually work directly with this class. Instead, you create a `FileReader` object to connect your program to a file, and then pass that object to the constructor of the `BufferedReader` class, which provides more efficient access to the file. (This class extends the abstract class `Reader`, which is the base class for a variety of different classes that can read character data from a stream.)

- **`BufferedReader`:** This class “wraps” around the `FileReader` class to provide more efficient input. This class adds a *buffer* to the input stream that allows the input to be read from disk in large chunks rather than one byte at a time. This can result in a huge improvement in performance. The `BufferedReader` class lets you read data one character at a time or a line at a time. In most programs, you read data one line at a time, and then use Java’s string-handling features to break the line into individual fields.

Table 2-1 lists the most important constructors and methods of these classes.

Table 2-1 The BufferedReader and FileReader Classes

Constructors	*Description*
`BufferedReader (Reader in)`	Creates a buffered reader from any object that extends the `Reader` class. Typically, you pass this constructor a `FileReader` object.
`FileReader(File file)`	Creates a file reader from the specified `File` object. Throws `FileNotFoundException` if the file doesn’t exist or if it is a directory rather than a file.
`FileReader(String path)`	Creates a file reader from the specified pathname. Throws `FileNotFoundException` if the file doesn’t exist or if it is a directory rather than a file.
Methods *(from various classes)*	***Description***
`void close()`	Closes the file. Throws `IOException`.
`int read()`	Reads a single character from the file and returns it as an integer. Returns `–1` if the end of the file has been reached. Throws `IOException`.
`String readLine()`	Reads an entire line and returns it as a string. Returns `null` if the end of the file has been reached. Throws `IOException`.
`void skip(long num)`	Skips ahead the specified number of characters.

In the following sections, you find out how to read a file named `movies.txt` that contains one line for ten of my favorite movies. Each line of the file contains the title of the movie, a tab, the year the movie was released, another tab, and the price I paid for it at the video store. Here’s the contents of the file:

```
It's a Wonderful Life➪1946➪14.95
The Great Race➪1965➪12.95
Young Frankenstein➪1974➪16.95
The Return of the Pink Panther➪1975➪11.95
Star Wars➪1977➪17.95
The Princess Bride➪1987➪16.95
Glory➪1989➪14.95
```

```
Apollo 13➪1995➪19.95
The Game➪1997➪14.95
The Lord of the Rings: The Fellowship of the Ring➪
   2001➪19.95
```

(In this list, the arrows represent tab characters.) Later in this chapter, I show you a program that writes data to this file.

TIP

If you create this file with a text editor, make sure your text editor correctly preserves the tabs.

Creating a BufferedReader

The normal way to connect a character stream to a file is to create a `File` object for the file using one of the techniques presented in the preceding chapter. Then, you can call the `FileReader` constructor to create a `FileReader` object and pass this object to the `BufferedReader` constructor to create a `BufferedReader` object. For example:

```
File f = new File("movies.txt");
BufferedReader bfIn = new BufferedReader(
                    new FileReader(f));
```

Here, a `BufferedReader` object is created to read the `movies.txt` file.

Reading from a character stream

To read a line from the file, you use the `readLine` method of the `BufferedReader` class. This method returns `null` when the end of the file is reached. As a result, testing the string returned by the `readLine` method in a `while` loop to process all the lines in the file is common.

For example, this code snippet reads each line from the file and prints it to the console:

```
String line = bfIn.readLine();
while (line != null)
{
    System.out.println(line);
    line = bfIn.readLine();
}
```

After you read a line of data from the file, you can use Java's string handling features to pull out the individual bits of data from the line. In particular, you can use the `split` method to separate the line into the individual strings that are separated by tabs. Then, you can use the appropriate parse methods (such as `parseInt` and `parseDouble`) to convert each string to its correct data type.

For example, here's a routine that converts a line read from the `movies.txt` file to the title (a string), year (an `int`), and price (a `double`):

```
String[] data = line.split("\t");
String title = data[0];
int year = Integer.parseInt(data[1]);
double price = Double.parseDouble(data[2]);
```

TIP

After the entire file is read, you can close the stream by calling the `close` method:

```
bfIn.close();
```

Reading the movies.txt file

Listing 2-1 shows a complete, albeit simple, program that reads the `movies.txt` file and prints the contents of the file to the console.

Listing 2-1: Reading from a Text File

```
import java.io.*;                                                →  1
import java.text.NumberFormat;

public class ReadFile
{
    public static void main(String[] args)
    {
        NumberFormat cf = NumberFormat.getCurrencyInstance();

        BufferedReader bfIn = getReader("movies.txt");           → 10

        Movie movie = readMovie(bfIn);                           → 12
        while (movie != null)                                    → 13
        {
            String msg = Integer.toString(movie.year);
            msg += ": " + movie.title;
            msg += " (" + cf.format(movie.price) + ")";
            System.out.println(msg);
            movie = readMovie(bfIn);
        }
    }                                                            → 21

    private static BufferedReader getReader(String name)         → 23
    {
        BufferedReader bfIn = null;
        try
        {
            File file = new File(name);
            bfIn = new BufferedReader(
                new FileReader(file) );
        }
        catch (FileNotFoundException e)
        {
```

continued

Listing 2-1 (continued)

```
            System.out.println("The file doesn't exist.");
            System.exit(0);
        }
        catch (IOException e)
        {
            System.out.println("I/O Error");
            System.exit(0);
        }
        return bfIn;
    }

    private static Movie readMovie(BufferedReader bfIn)           →45
    {
        String title;
        int year;
        double price;
        String line = "";
        String[] data;

        try
        {
            line = bfIn.readLine();
        }
        catch (IOException e)
        {
            System.out.println("I/O Error");
            System.exit(0);
        }

        if (line == null)
            return null;
        else
        {
            data = line.split("\t");
            title = data[0];
            year = Integer.parseInt(data[1]);
            price = Double.parseDouble(data[2]);
            return new Movie(title, year, price);
        }
    }

    private static class Movie                                     → 75
    {
        public String title;
        public int year;
        public double price;

        public Movie(String title, int year, double price)
        {
            this.title = title;
            this.year = year;
            this.price = price;
        }
    }
}
```

If you run this program, the following output is displayed on the console:

```
1946: It's a Wonderful Life ($14.95)
1965: The Great Race ($12.95)
1974: Young Frankenstein ($16.95)
1975: The Return of the Pink Panther ($11.95)
1977: Star Wars ($17.95)
1987: The Princess Bride ($16.95)
1989: Glory ($14.95)
1995: Apollo 13 ($19.95)
1997: The Game ($14.95)
2001: The Lord of the Rings: The Fellowship of the Ring
   ($19.95)
```

Because I've already explained most of this code, the following paragraphs provide just a roadmap to this program:

→ **1** The program begins with `import java.io.*` to import all the Java I/O classes used by the program.

→**10** The program uses a method named `getReader` to create a `BufferedReader` object that can read the file. The name of the file is passed to this method as a parameter. Note that in a real program, you'd probably get this filename from the user via a `JFileChooser` dialog box or some other means. In any event, the `BufferedReader` object returned by the `getReader` method is saved in a variable named `bfIn`.

→**12** Another method, named `readMovie`, is used to read each movie from the file. This method returns a `Movie` object — `Movie` is a private class that's defined later in the program. If the end of the file has been reached, this method returns `null`.

→**13** A `while` loop is used to process each movie. This loop simply builds a message string from the `Movie` object, displays it on the console, and then calls `readMovie` to read the next movie in the file.

→**21** The program ends without closing the file. That's okay, though, because the file is closed automatically when the program that opened it ends. If the program were to go on with other processing after it was finished with the file, you'd want to close the file first.

→**23** The `getReader` method creates a `BufferedReader` object for the filename passed as a parameter. If any exceptions are thrown while trying to create the `BufferedReader`, the program exits.

→**45** The `readMovie` method reads a line from the reader passed as a parameter, parses the data in the line, creates a `Movie` object from the data, and returns the `Movie` object. If the end of the file is reached, this method returns `null`. The statement that reads the line from the file is enclosed in a `try/catch` block that exits the program if an I/O error occurs.

→75 The `Movie` class is a private inner class that defines the movie objects. To keep the class simple, it uses public fields and a single constructor that initializes the fields.

Writing Character Streams

The usual way to write data to a text file is to use the `PrintWriter` class, which as luck has it you're already familiar with: It's the same class that provides the `print` and `println` methods used to write console output. As a result, the only real trick to writing output to a text file is figuring out how to connect a print writer to a text file. To do that, you work with three classes:

- **`FileWriter`:** The `FileWriter` class connects to a `File` object but provides only rudimentary writing ability.
- **`BufferedWriter`:** This class connects to a `FileWriter` and provides output buffering. Without the buffer, data is written to disk one character at a time. This class lets the program accumulate data in a buffer and writes the data only when the buffer is filled up or when the program requests that the data be written.
- **`PrintWriter`:** This class connects to a `Writer`, which can be a `BufferedWriter`, a `FileWriter`, or any other object that extends the abstract `Writer` class. Most often, you connect this class to a `BufferedWriter`.

The `PrintWriter` class is the only one of these classes whose methods you usually use when you write data to a file. Table 2-2 lists the most important constructors and methods of this class.

Table 2-2 The PrintWriter, BufferedWriter, and FileWriter Classes

Constructors	*Description*
`PrintWriter(Writer out)`	Creates a print writer for the specified output writer.
`PrintWriter(Writer out, boolean flush)`	Creates a print writer for the specified output writer. If the second parameter is true, the buffer is automatically flushed whenever the `println` method is called.
`BufferedWriter(Writer out)`	Creates a buffered writer from the specified writer. Typically, you pass this constructor a `FileWriter` object.
`FileWriter(File file)`	Creates a file writer from the specified `File` object. Throws `IOException` if an error occurs.
`FileWriter(File file, boolean append)`	Creates a file writer from the specified `File` object. Throws `IOException` if an error occurs. If the second parameter is true, data is added to the end of the file if the file already exists.

Constructors	Description
`FileWriter(String path)`	Creates a file writer from the specified pathname. Throws `IOException` if an error occurs.
`FileWriter(String path, boolean append)`	Creates a file writer from the specified pathname. Throws `IOException` if an error occurs. If the second parameter is true, data is added to the end of the file if the file already exists.
PrintWriter Methods	***Description***
`void close()`	Closes the file.
`void flush()`	Writes the contents of the buffer to disk.
`int read()`	Reads a single character from the file and returns it as an integer. Returns `–1` if the end of the file has been reached. Throws `IOException`.
`void print(value)`	Writes the value, which can be any primitive type or any object. If the value is an object, the object's `toString()` method is called.
`void println(value)`	Writes the value, which can be any primitive type or any object. If the value is an object, the object's `toString()` method is called. A line break is written following the value.

Connecting a PrintWriter to a text file

To connect a character stream to an output file, you first create a `File` object for the file as I describe in the preceding chapter. Then, you call the `PrintWriter` constructor to create a `PrintWriter` object you can use to write to the file. This constructor wraps around a `BufferedWriter` object, which in turn wraps around a `FileWriter` object like this:

```
File file = new File("movies.txt");
PrintWriter pwOut =
    new PrintWriter(
        new BufferedWriter(
            new FileWriter(file) ) );
```

If you find this a little confusing, that's good! That makes me feel a little better, because I find it a little confusing too. The basic idea going on here is that each of the classes is adding a capability to the class it wraps. At the bottom is the `FileWriter` class, which has the ability to write characters to a file. The `BufferedWriter` class adds buffering to the mix, saving data in a buffer until it makes sense to write it all out to the file in one big spurt. And the `PrintWriter` class adds basic formatting capabilities, like adding line endings at the end of each line and converting primitive types to strings.

Both the `FileWriter` and the `PrintWriter` classes have an optional `boolean` parameter you can use to add extra capabilities to the file stream.

If you specify true in the FileWriter constructor, the file is *appended* if it exists. That simply means that any data in the file is retained; data you write to the file in your program is simply added on to the end of the file. Here's a PrintWriter constructor that appends data to its file:

```
File file = new File("movies.txt");
PrintWriter pwOut =
    new PrintWriter(
        new BufferedWriter(
            new FileWriter(file, true )))// append mode
```

If you specify false instead of true, or if you leave this parameter out altogether, an existing file is deleted, and its data is lost.

The boolean parameter in the PrintWriter class has less dire consequences. It simply tells the PrintWriter class that it should tell the BufferedWriter class to flush its buffer whenever you use the println method to write a line of data. Although this option might decrease the efficiency of your program by a small amount, it also makes the program a little more reliable because it reduces the odds of losing data because your program or the whole computer crashes while unwritten data is in the buffer.

Unfortunately, the code for specifying this option looks a little goofy because of the way the constructors for the BufferedWriter and FileWriter classes are nested:

```
File file = new File("movies.txt");
PrintWriter pwOut =
    new PrintWriter(
        new BufferedWriter(
           new FileWriter(file) ), true); ////mode flush
```

TIP

If all these nested constructors make your head spin, you can always construct each object separately and use variables to keep track of them. Here's an example that does that, and turns on append mode for the FileWriter and flush mode for the PrintWriter:

```
FileWriter fw = new FileWriter(file, true);
BufferedWriter bw = new BufferedWriter(fw);
PrintWriter pwOut = new PrintWriter(bw, true);
```

If you find this coding technique easier to understand, by all means use it.

Writing to a character stream

After you successfully connect a character stream to a file, writing data to it is as easy as writing text to the console. You just use the print and println methods exactly as if you're writing to the console.

One minor complication is that if you're writing data to a text file in a delimited format, you have to include statements that write the delimiter characters to the file. For example, suppose the title and year for a movie you want to write to the text file are stored in String variables named `title` and `year`. This snippet of code writes these fields with a tab delimiter between them:

```
System.out.print(title);
System.out.print("\t");
System.out.println(year);
```

Here, the last item to be written is written with the `println` method rather than the `print` method. That ends the current line.

If you prefer to be a little more efficient, you can build a string representing the entire line, and then write the line all at once:

```
String line = title + "\t" + year;
System.out.println(line);
```

This way is a little more efficient than the previous version, but not as much as you'd think. In most cases, the `BufferedWriter` holds your text in a buffer until the `println` method is called anyway.

TIP

If you didn't specify the flush option when you created the `PrintWriter` object, you can still periodically force any data in the buffer to be written to disk by calling the `flush` method:

```
pwOut.flush();
```

Also, when you're finished writing data to the file, you can close the file by calling the `close` method:

```
pwOut.close();
```

Writing the movies.txt file

Listing 2-2 shows a complete program that writes lines to a text file. The data written is taken from an array that's hard-coded into the file, but you can easily imagine how to obtain the data from the user by prompting for console input or using text fields in a Swing application.

Listing 2-2: Writing to a Text File

```
import java.io.*;

public class WriteFile
{
    public static void main(String[] args)                    → 5
```

continued

Listing 2-2 (continued)

```
    {
        Movie[] movies = getMovies();

        PrintWriter pwOut = openWriter("movies.txt");
        for (Movie m : movies)
            writeMovie(m, pwOut);
        pwOut.close();
    }

    private static Movie[] getMovies()                           → 15
    {
        Movie[] movies = new Movie[10];

        movies[0] = new Movie("It's a Wonderful Life", 1946, 14.95);
        movies[1] = new Movie("The Great Race", 1965, 12.95);
        movies[2] = new Movie("Young Frankenstein", 1974, 16.95);
        movies[3] = new Movie("The Return of the Pink Panther", 1975,
               11.95);
        movies[4] = new Movie("Star Wars", 1977, 17.95);
        movies[5] = new Movie("The Princess Bride", 1987, 16.95);
        movies[6] = new Movie("Glory", 1989, 14.95);
        movies[7] = new Movie("Apollo 13", 1995, 19.95);
        movies[8] = new Movie("The Game", 1997, 14.95);
        movies[9] = new Movie("The Lord of the Rings: The Fellowship
               of the Ring", 2001, 19.95);

        return movies;
    }

    private static PrintWriter openWriter(String name)           → 40
    {
        try
        {
            File file = new File(name);
            PrintWriter pwOut =
                new PrintWriter(
                    new BufferedWriter(
                        new FileWriter(file) ), true );
            return pwOut;
        }
        catch (IOException e)
        {
            System.out.println("I/O Error");
            System.exit(0);
        }
        return null;
    }

private static void writeMovie(Movie m,                          → 58
    PrintWriter pwOut)
    {
        String line = m.title;
        line += "\t" + Integer.toString(m.year);
        line += "\t" + Double.toString(m.price);
        pwOut.println(line);
    }

    private static class Movie                                  → 67
    {
        public String title;
        public int year;
        public double price;

        public Movie(String title, int year, double price)
        {
            this.title = title;
            this.year = year;
            this.price = price;
        }
    }
}
```

Because all the coding elements in this program have already been explained in this chapter, the following paragraphs just provide a roadmap to the major part of the program:

→ 5 The `main` method begins by calling a method named `getMovies`, which returns an array of `Movie` objects to be written to the file. (The `Movie` class is defined as an inner class later in the program.) Then, it calls `openWriter`, which creates a `PrintWriter` object the program can use to write data to the file. Next, it uses an enhanced `for` loop to call the `writeMovie` method for each movie in the array. This method accepts a `Movie` object that contains the movie to be written and a `PrintWriter` object to write the movie to. Finally, the `PrintWriter` is closed.

→15 The `getMovies` method returns an array of `Movie` objects that are written to a file. In a real-life program, you probably do something other than hard-code the movie information in this method. For example, you might prompt the user to enter the data or use a Swing frame to get the data.

→40 The `openWriter` method creates a `PrintWriter` object for the filename passed to it as a parameter. The `PrintWriter` uses a buffer that's flushed each time `println` is called.

→58 The `writeMovie` method accepts as parameters a `Movie` object to be written and the `PrintWriter` the movie should be written to. It creates a string that includes the title, a tab, the year, another tab, and the price. Then, it writes the string to the file.

→67 The `Movie` class is an inner class that defines a movie object. This class simply consists of three public fields (title, year, and price) and a constructor that initializes the fields.

Reading Binary Streams

Binary streams are a bit tougher to read than character streams, but not much. The biggest obstacle to pass when you're reading a binary stream is that you need to know exactly the type of each item that was written to the file. If any incorrect data is in the file, the program won't work. So you need to ensure the file contains the data your program expects it to contain.

To read a binary file, you usually work with the following classes:

- **`File`:** Once again, you use the `File` class to represent the file itself.
- **`FileInputStream`:** The `FileInputStream` is what connects the input stream to a file.
- **`BufferedInputStream`:** This class adds buffering to the basic `FileInputStream`, which improves the stream's efficiency and gives it a moist and chewy texture.
- **`DataInputStream`:** This is the class you actually work with to read data from the stream. The other `Stream` classes read a byte at a time. This class knows how to read basic data types, including primitive types and strings.

Table 2-3 lists the vital constructors and methods of these classes.

Table 2-3 The BufferedReader and FileReader Classes

Constructors	*Description*
`BufferedInputStream (InputStream in)`	Creates a buffered input stream from any object that extends the `InputStream` class. Typically, you pass this constructor a `FileInputStream` object.
`DataInputStream (InputStream in)`	Creates a data input stream from any object that extends the `InputStream` class. Typically, you pass this constructor a `BufferedInputStream` object.
`FileInputStream (File file)`	Creates a file input stream from the specified `File` object. Throws `FileNotFoundException` if the file doesn't exist or if it is a directory rather than a file.
`FileInputStream (String path)`	Creates a file input stream from the specified path-name. Throws `FileNotFoundException` if the file doesn't exist or if it is a directory rather than a file.
DataInputStream Methods	***Description***
`boolean readBoolean()`	Reads a `boolean` value from the input stream. Throws `EOFException` and `IOException`.
`byte readByte()`	Reads a `byte` value from the input stream. Throws `EOFException` and `IOException`.
`char readChar()`	Reads a `char` value from the input stream. Throws `EOFException` and `IOException`.

DataInputStream Methods	*Description*
`double readDouble()`	Reads a `double` value from the input stream. Throws `EOFException` and `IOException`.
`float readFloat()`	Reads a `float` value from the input stream. Throws `EOFException` and `IOException`.
`int readInt()`	Reads an `int` value from the input stream. Throws `EOFException` and `IOException`.
`long readLong()`	Reads a `long` value from the input stream. Throws `EOFException` and `IOException`.
`short readShort()`	Reads a `short` value from the input stream. Throws `EOFException` and `IOException`.
`String readUTF()`	Reads a string stored in UTF format from the input stream. Throws `EOFException`, `IOException`, and `UTFDataFormatException`.

The following sections present programs that read and write data in a binary file named `movies.dat` that contains information about movies. Each record in this file consists of a UTF string containing the movie's title, an `int` representing the year the movie was released, and a `double` representing the price I paid for the movie at my local discount video store. Although the format of this file is different than the `movies.txt` file shown earlier in this chapter, the file contains the same data. You can refer to the earlier section "Reading Character Streams" to see a listing of the movies in this file.

Creating a DataInputStream

To read data from a binary file, you want to connect a `DataInputStream` object to an input file. To do that, you use a `File` object to represent the file, a `FileInputStream` object that represents the file as an input stream, a `BufferedInputStream` object that adds buffering to the mix, and finally a `DataInputStream` object to provide the methods that read various data type. The constructor for such a beast looks like this:

```
File file = new File("movies.dat");
DataInputStream dsIn = new DataInputStream(
       new BufferedInputStream(
         new FileInputStream(file) ) );
```

If all the nesting makes you nauseous, you can do it this way instead:

```
File file = new File("movies.dat");
FileInputStream fs = new FileInputStream(file);
BufferedInputStream bs = new BufferedInputStream(fs);
DataInputStream dsIn = new DataInputStream(bs);
```

Either way, the effect is the same.

Reading from a data input stream

With binary files, you don't read an entire line into the program and parse it into individual fields. Instead, you use the various read methods of the `DataInputStream` class to read the fields one at a time. To do that, you have to know the exact sequence in which data values appear in the file.

For example, here's a code snippet that reads the information for a single movie and stores the data in variables:

```
String title = dsIn.readUTF();
int year = dsIn.readInt();
double price = dsIn.readDouble();
```

Note that the read methods all throw `EOFException` if the end of the file is reached and `IOException` if an I/O error occurs. So you need to call these methods inside a `try/catch` block that catches these exceptions. The `readUTF` method also throws `UTFDataFormatException`, but that exception is a type of `IOException`, so you probably don't need to catch it separately.

The read methods are usually used in a `while` loop to read all the data from the file. When the end of the file is reached, `EOFException` is thrown. You can then catch this exception and stop the loop. One way to do that is to use a `boolean` variable to control the loop:

```
boolean eof = false;
while (!eof)
{
    try
    {
        String title = dsIn.readUTF();
        int year = dsIn.readInt();
        double price = dsIn.readDouble();
        // do something with the data here
    }
    catch (EOFException e)
    {
        eof = true;
    }
    catch (IOException e)
    {
        System.out.println("An I/O error has
            occurred!");
        System.exit(0);
    }
}
```

Here, the `boolean` variable `eof` is set to `true` when `EOFException` is thrown, and the loop continues to execute as long as `eof` is `false`.

After you read a line of data from the file, you can use Java's string handling features to pull out the individual bits of data from the line. In particular, you can use the `split` method to separate the line into the individual strings that are separated by tabs. Then, you can use the appropriate parse methods to parse each string to its correct data type.

For example, here's a routine that converts a line read from the `movies.txt` file to the title (a string), year (an `int`), and price (a `double`):

```
String[] data = line.split("\t");
String title = data[0];
int year = Integer.parseInt(data[1]);
double price = Double.parseDouble(data[2]);
```

TIP

After the entire file has been read, you can close the stream by calling the `close` method:

```
dsIn.close();
```

This method also throws `IOException`, so you want to place it inside a `try/catch` block.

Reading the movies.dat file

Now that you've seen the individual elements of reading data from a binary file, Listing 2-3 presents a complete program that uses these techniques. This program reads the `movies.dat` file, creates a `Movie` object for each title, year, and price value, and prints a line on the console for the movie. If you run this program, the output looks exactly like the output from the text file version presented earlier in this chapter, in the section "Reading the movies.txt file."

Listing 2-3: Reading from a Binary File

```
import java.io.*;
import java.text.NumberFormat;

public class ReadBinaryFile
{
    public static void main(String[] args)                    → 6
    {
        NumberFormat cf = NumberFormat.getCurrencyInstance();

        DataInputStream dsIn = getStream("movies.dat");

        boolean eof = false;
        while (!eof)
        {
            Movie movie = readMovie(in);
            if (movie == null)
```

continued

Listing 2-3 (continued)

```
                eof = true;
            else
            {
                String msg = Integer.toString(movie.year);
                msg += ": " + movie.title;
                msg += " (" + cf.format(movie.price) + ")";
                System.out.println(msg);
            }
        }
        closeFile(dsIn);
    }
    private static DataInputStream getStream(String name)          → 28
    {
        DataInputStream dsIn = null;                              → 30
        try
        {
            File file = new File(name);
            dsIn = new DataInputStream(
                   new BufferedInputStream(
                     new FileInputStream(file) ) );
        }
        catch (FileNotFoundException e)
        {
            System.out.println("The file doesn't exist.");
            System.exit(0);
        }
        catch (IOException e)
        {
            System.out.println("I/O Error creating file.");
            System.exit(0);
        }
        return dsIn;
    }

    private static Movie readMovie(DataInputStream in)             → 51
    {
        String title = "";
        int year = 0;;
        double price = 0.0;;

        try
        {
            title = dsIn.readUTF();
            year = dsIn.readInt();
            price = dsIn.readDouble();
        }
        catch (EOFException e)
        {
            return null;
        }
        catch (IOException e)
        {
            System.out.println("I/O Error");
            System.exit(0);
        }
        return new Movie(title, year, price);
    }
    private static void closeFile(DataInputStream dsIn)          → 76
    {
        try
        {
            dsIn.close();
        }
        catch(IOException e)
        {
            System.out.println("I/O Error closing file.");
            System.out.println();
        }
    }

    private static class Movie                                      → 89
    {
        public String title;
        public int year;
        public double price;

        public Movie(String title, int year, double price)
        {
            this.title = title;
            this.year = year;
            this.price = price;
        }
    }
}
```

The following paragraphs describe what each method in this program does:

→ **6** The `main` method is intentionally kept simple so it can focus on controlling the flow of the program rather than doing the detail work of accessing the file. As a result, it calls a method named `getStream` to get a data input stream object to read the file. Then, it uses a `while` loop to call a method named `readMovie` to get a movie object. If the `Movie` object isn't `null`, the movie's data is then printed to the console. Finally, when the loop ends, a method named `closeFile` is called to close the file.

→**28** The `getStream` method creates a `DataInputStream` object for the filename passed as a parameter. If any exceptions are thrown, the program exits. dsIn must be declared and initalized outside of try-catch since it is used in `return dsIn;`

→**51** The `readMovie` method reads the data for a single movie and creates a `Movie` object. If the end of the file is reached, the method returns `null`.

→**76** The `closeFile` method closes the input stream.

→**89** As in the other programs in this chapter, the `Movie` class is defined as a private inner class.

Writing Binary Streams

To write data to a binary file, you use the following classes:

- **`FileOutputStream:`** The `FileOutputStream` class connects to a `File` object and creates an output stream that can write to the file. However, this output stream is limited in its capabilities: It can write only raw bytes to the file. In other words, it doesn't know how to write values such as `int`s, `double`s, or strings.
- **`BufferedOutputStream:`** This class connects to a `FileOutputStream` and adds output buffering.
- **`DataOutputStream:`** This class adds the ability to write primitive data types and strings to a stream.

Table 2-4 lists the essential constructors and methods of these classes.

Table 2-4 The DataOutputStream, BufferedOutputStream, and FileOutputStream Classes

Constructors	*Description*
`DataOutputStream (OutputStream out)`	Creates a data output stream for the specified output stream.
`BufferedIOutputStream (OutputStream out)`	Creates a buffered output stream for the specified stream. Typically, you pass this constructor a `FileOutputStream` object.
`FileOutputStream (File file)`	Creates a file writer from the file. Throws `FileNotFoundException` if an error occurs.
`FileOutputStream(File file, boolean append)`	Creates a file writer from the file. Throws `FileNotFoundException` if an error occurs. If the second parameter is true, data is added to the end of the file if the file already exists.
`FileOutputStream (String path)`	Creates a file writer from the specified pathname. Throws `FileNotFoundException` if an error occurs.
`FileOutputStream(String path, boolean append)`	Creates a file writer from the specified pathname. Throws `FileNotFoundException` if an error occurs. If the second parameter is true, data is added to the end of the file if the file already exists.
DataOutputStream Methods	***Description***
`void close()`	Closes the file.
`void flush()`	Writes the contents of the buffer to disk.
`int size()`	Returns the number of bytes written to the file.
`void writeBoolean (boolean value)`	Writes a `boolean` value to the output stream. Throws `IOException`.

DataInputStream Methods	*Description*
`void writeByte(byte value)`	Writes a `byte` value to the output stream. Throws `IOException`.
`void writeChar(char value)`	Writes a `char` value to the output stream. Throws `IOException`.
`void writeDouble(double value)`	Writes a `double` value to the output stream. Throws `IOException`.
`void writeFloat(float value)`	Writes a `float` value to the output stream. Throws `IOException`.
`void writeInt(int value)`	Writes an `int` value to the output stream. Throws `IOException`.
`void writeLong(long value)`	Writes a `long` value to the output stream. Throws `IOException`.
`void writeShort(short value)`	Writes a `short` value to the output stream. Throws `IOException`.
`void writeUTF(String value)`	Writes a string stored in UTF format to the output stream. Throws `EOFException`, `IOException`, and `UTFDataFormatException`.

Creating a DataOutputStream

Creating a `DataOutputStream` object requires yet another one of those crazy nested constructor things:

```
File file = new File(name);
DataOutputStream dsOut = new DataOutputStream(
        new BufferedOutputStream(
          new FileOutputStream(file) ) );
```

If you prefer, you can unravel the constructors like this:

```
File file = new File(name);
FileOutputStream fos = new FileOutputStream(file);
BufferedOutputStream bos = new
   BufferedOutputStream(fos);
DataOutputStream dsOut = new DataOutputStream(bos);
```

The `FileOutputStream` class has an optional `boolean` parameter you can use to indicate that the file should be appended if it exists. To use this feature, call the constructors like this:

```
File file = new File(name);
DataOutputStream out = new DataOutputStream(
        new BufferedOutputStream(
          new FileOutputStream(file, true) ) );
```

If you specify `false` instead of `true` or leave the parameter out altogether, an existing file is deleted and its data is lost.

Writing to a binary stream

After you successfully connect a `DataOutputStream` to a file, writing data to it is simply a matter of calling the various write methods to write different data types to the file. For example, the following code writes the data for a `Movie` object to the file:

```
dsOut.writeUTF(movie.title);
dsOut.writeInt(movie.year);
dsOut.writeDouble(movie.price);
```

Of course, these methods throw `IOException`. As a result, you have to enclose them in a `try/catch` block.

TIP

If you included the `BufferedOutputStream` class in the stream, it accumulates data in its buffer until it decides to write the data to disk. If you want, you can force the buffer to be written to disk by calling the `flush` method, like this:

```
dsOut.flush();
```

Also, when you finish writing data to the file, close the file by calling the `close` method, like this:

```
dsOut.close();
```

Both the `flush` and `close` methods also throw `IOException`, so you need a `try/catch` to catch the exception.

Writing the movies.dat file

Listing 2-4 presents a program that writes the `movies.dat` file from an array of `Movie` objects whose values are hard-coded into the program.

Listing 2-4: Writing to a Text File

```
import java.io.*;

public class WriteBinaryFile
{
    public static void main(String[] args)                        → 5
    {
        Movie[] movies = getMovies();
        DataOutputStream dsOut = openOutputStream("movies.dat");
        for (Movie m : movies)
            writeMovie(m, dsOut);
        closeFile(dsOut);
    }

    private static Movie[] getMovies()                            → 14
    {
        Movie[] movies = new Movie[10];

        movies[0] = new Movie("It's a Wonderful Life", 1946, 14.95);
        movies[1] = new Movie("The Great Race", 1965, 12.95);
        movies[2] = new Movie("Young Frankenstein", 1974, 16.95);
        movies[3] = new Movie("The Return of the Pink Panther", 1975,
                11.95);
        movies[4] = new Movie("Star Wars", 1977, 17.95);
        movies[5] = new Movie("The Princess Bride", 1987, 16.95);
        movies[6] = new Movie("Glory", 1989, 14.95);
        movies[7] = new Movie("Apollo 13", 1995, 19.95);
        movies[8] = new Movie("The Game", 1997, 14.95);
        movies[9] = new Movie("The Lord of the Rings: The Fellowship
                of the Ring", 2001, 19.95);
        return movies;
    }

private static DataOutputStream
    openOutputStream(String name)                                  → 39
    {
        DataOutputStream dsOut = null;
        try
        {
            File file = new File(name);
            out = new DataOutputStream(
                    new BufferedOutputStream(
                      new FileOutputStream(file) ) );
            return dsOut;           // is this line needed here?
        }
        catch (IOException e)
        {
            System.out.println(
                "I/O Exception opening file.");
            System.exit(0);
        }
        return dsOut;
    }

private static void writeMovie(Movie m,                           → 59
    DataOutputStream dsOut)
    {
        try
        {
            dsOut.writeUTF(m.title);
            dsOut.writeInt(m.year);
            dsOut.writeDouble(m.price);
        }
        catch (IOException e)
        {
            System.out.println(
                "I/O Exception writing data.");
            System.exit(0);
        }
    }

    private static void closeFile(DataOutputStream dsOut)          → 76
    {
```

continued

Listing 2-4 (continued)

```
        try
        {
            dsOut.close();
        }
        catch (IOException e)
        {
            System.out.println("I/O Exception closing file.");
            System.exit(0);
        }
    }

    private static class Movie                                     → 89
    {
        public String title;
        public int year;
        public double price;

        public Movie(String title, int year, double price)
        {
            this.title = title;
            this.year = year;
            this.price = price;
        }
    }
}
```

Because this chapter explains all the coding elements in this program, the following paragraphs just provide a roadmap to the major part of the program:

→ 5 The `main` method calls `getMovies` to get an array of `Movie` objects. Then, it calls `openOutputStream` to get an output stream to write data to the file. Then, an enhanced `for` loop calls `writeMovie` to write the movies to the file. Finally, it calls `closeFile` to close the file.

→14 The `getMovies` method creates an array of movies to be written to the file.

→39 The `openOutputStream` method creates a `DataOutputStream` object so the program can write data to the file.

→59 The `writeMovie` method accepts two parameters: the movie to be written and the output stream to write the data to.

→76 The `closeFile` method closes the file.

→89 Once again, the `Movie` class is included as an inner class.