CHAPTER 10

File I/O

10 File I/O

As a leaf is carried by a stream, whether the stream ends in a lake or in the sea, so too is the output of your program carried by a stream not knowing if the stream goes to the screen or to a file.

Washroom Wall of a
Computer Science Department (1995)

Introduction

In this chapter, we explain how you can write your programs to take input from a file and send output to a file. This chapter covers the most common ways of doing file I/O in Java. However, it is not an exhaustive study of Java I/O classes. The Java I/O class library contains bewilderingly many classes and an exhaustive treatment of all of them would be a book by itself.

Prerequisites

You need only some of Chapter 9 on exception handling to read this chapter. You do not need Chapters 6, 7, or 8 on arrays, inheritance, and polymorphism, except for in the final subsection, which covers writing and reading of arrays to binary files. If you have not yet covered some basic material on one-dimensional arrays, you can, of course, simply omit this last subsection.

You may postpone all or part of this chapter if you wish. Nothing in the rest of this book requires any of this chapter.

10.1 Introduction to File I/O

Good Heavens! For more than forty years I have been speaking prose without knowing it.

Molière, *Le Bourgeois Gentilhomme*

In this section we go over some basic concepts about file I/O before we go into any Java details.

STREAMS

stream

A **stream** is an object that allows for the flow of data between your program and some I/O device or some file. If the flow is into your program, the stream

is called an **input stream**. If the flow is out of your program, the stream is called an **output stream**. If the input stream flows from the keyboard, then your program will take input from the keyboard. If the input stream flows from a file, then your program will take its input from that file. Similarly, an output stream can go to the screen or to a file.

input stream

output stream

Although you may not realize it, you have already been using streams in your programs. The `System.out` that you have already used (in such things as `System.out.println`) is an output stream connected to the screen. `System.in` is an input stream connected to the keyboard. You used `System.in` in expressions like the following:

`System.out`
`System.in`

```
BufferedReader keyboard =
                    new BufferedReader(new InputStreamReader(System.in));
```

These two streams are automatically available to your program. You can define other streams that come from or go to files. Once you have defined them, you can use them in your program in ways that are similar to how you use `System.out` and `System.in`.

Streams

A **stream** is a flow of data. If the data flows *into your program,* then the stream is called an **input stream**. If the data flows *out of your program,* the stream is called an **output stream**.

Streams are used for both console I/O, which you have been using already, and file I/O.

input stream
output stream

TEXT FILES AND BINARY FILES

Text files are files that appear to contain sequences of characters when viewed in a text editor or read by a program. For example, the files that contain your Java programs are text files. Text files are sometimes also called ASCII files because they contain data encoded using a scheme known as ASCII coding. Files whose contents must be handled as sequences of binary digits are called **binary files**.

text file
ASCII file
binary file

Although it is not strictly speaking correct, you can safely think of a text file as containing a sequence of characters, and think of a binary file as containing a sequence of binary digits. Another way to distinguish between binary files and text files is to note that text files are designed to be read by human beings, whereas binary files are designed to be read only by programs.

One advantage of text files is that they are usually the same on all computers, so you can move your text files from one computer to another with few or no problems. The implementation of binary files usually differs from one computer to another, so your binary data files ordinarily must be read only on the same type of computer, and with the same programming language, as the computer that created that file.

The advantage of binary files is that they are more efficient to process than text files. Unlike other programming languages, Java also gives its binary files some of the advantages of text files. In particular, 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. This combines the portability of text files with the efficiency of binary files.

The one big advantage of text files is that you can read and write to them using a text editor. With binary files, all the reading and writing must normally be done by a program.

Text Files Versus Binary Files

Files that you write and read using an editor are called *text files*. Binary files represent data in a way that is not convenient to read with a text editor, but that can be written to and read from a program very efficiently.

Self-Test Exercises

1. A stream is a flow of data. From where and to where does the data flow in an input stream? From where and to where does the data flow in an output stream?
2. What is the difference between a binary file and a text file?

10.2 Text Files

Polonius: What do you read, my lord?
Hamlet: Words, words, words.

William Shakespeare, *Hamlet*

In this section, we describe the most common ways to do text file I/O in Java.

WRITING TO A TEXT FILE

PrintWriter

The class `PrintWriter` is the preferred stream class for writing to a text file. An object of the class `PrintWriter` has the methods `print` and `println`, which are like the methods `System.out.print` and `System.out.println` that you have been using for screen output, but with an object of the class `PrintWriter`, the output goes to a text file. Display 10.1 contains a simple program that uses `PrintWriter` to send output to a text file. Let's look at the details of that program.

java.io

All the file I/O related classes we introduce in this chapter are in the package `java.io`, so all our program files begin with import statements similar to the ones in Display 10.1.

Display 10.1 Sending Output to a Text File

```
1  import java.io.PrintWriter;
2  import java.io.FileOutputStream;
3  import java.io.FileNotFoundException;

4  public class TextFileOutputDemo
5  {
6      public static void main(String[] args)
7      {
8          PrintWriter outputStream = null;
9          try
10         {
11             outputStream =
12                 new PrintWriter(new FileOutputStream("stuff.txt"));
13         }
14         catch(FileNotFoundException e)
15         {
16             System.out.println("Error opening the file stuff.txt.");
17             System.exit(0);
18         }

19         System.out.println("Writing to file.");

20         outputStream.println("The quick brown fox");
21         outputStream.println("jumped over the lazy dog.");

22         outputStream.close();

23         System.out.println("End of program.");
24     }
25 }
```

SAMPLE DIALOGUE

```
Writing to file.
End of program.
```

FILE `stuff.txt` (after the program is run.)

```
The quick brown fox
jumped over the lazy dog.
```

You can read this file using a text editor.

The program in Display 10.1 creates a text file named `stuff.txt` that a person can read using an editor or that another Java program can read. The program creates an object of the class `PrintWriter` as follows:

```
outputStream =
          new PrintWriter(new FileOutputStream("stuff.txt"));
```

The variable `outputStream` is of type `PrintWriter` and is declared outside the `try` block. The preceding two lines of code connect the stream named `outputStream` to the file named `stuff.txt`. This is called **opening the file**. When you connect a file to a stream in this way, your program always starts with an empty file. If the file `stuff.txt` already exists, the old contents of `stuff.txt` will be lost. If the file `stuff.txt` does not exist, then a new, empty file named `stuff.txt` will be created.

opening a file

We want to associate the output stream `outputStream` with the file named `stuff.txt`. However, the class `PrintWriter` has no constructor that takes a file name as its argument. So we use the class `FileOutputStream` to create a stream that can be used as an argument to a `PrintWriter` constructor. The expression

FileOutputStream

```
new FileOutputStream("stuff.txt")
```

takes a file name as an argument and creates an anonymous object of the class `FileOutputStream`, which is then used as an argument to a constructor for the class `PrintWriter` as follows:

```
new PrintWriter(new FileOutputStream("stuff.txt"))
```

This produces an object of the class `PrintWriter` that is connected to the file `stuff.txt`. Note that the name of the file, in this case `stuff.txt`, is given as a `String` value and so is given in quotes.

file name

If you want to read the file name from the keyboard, you could read the name to a variable of type `String` and use the `String` variable as the argument to the `FileOutputStream` constructor.

reading the file name

When you open a text file in the way just discussed, a `FileNotFoundException` can be thrown, and any such possible exception should be caught in a `catch` block. (Actually, it is the `FileOutputStream` constructor that might throw the `FileNotFoundException`, but the net effect is the same.)

FileNotFoundException

Notice that the `try` block in Display 10.1 encloses only the opening of the file. That is the only place that an exception might be thrown. Also note that the variable `outputStream` is declared outside of the `try` block. This is so that the variable `outputStream` can be used outside of the `try` block. Remember, anything declared in a block (even in a `try` block) is local to the block.

We said that when you open a text file for writing output to the file, the constructor might throw a `FileNotFoundException`. But in this situation you want to create a new file for output, so why would you care that the file was not found? The answer is simply that the exception is poorly named. A `FileNotFoundException` does not mean that the

Opening a Text File for Writing Output

You create a stream of the class `PrintWriter` and connect it to a text file for writing as follows.

Syntax:

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

Example:

```
PrintWriter outputStream = null;
outputStream =
          new PrintWriter(new FileOutputStream("stuff.txt"));
```

After this, you can use the methods `println` and `print` to write to the file.

When used in this way, the `FileOutputStream` constructor, and thus the `PrintWriter` constructor invocation, can throw a `FileNotFoundException`, which is a kind of `IOException`.

File Names

The rules for how you spell file names depend on your operating system, not on Java. When you give 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. A suffix, such as `.txt` in `stuff.txt`, has no special meaning to a Java program. We are using the suffix `.txt` 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.

A File Has Two Names

Every input file and every output file used by your program has two names: (1) the real file name that is used by the operating system and (2) the name of the stream that is connected to the file.

The stream name serves as a temporary name for the file and is the name that is primarily used within your program. After you connect the file to the stream, your program always refers to the file by using the stream name.

file was not found. In this case, it actually means that the file could not be created. A `FileNotFoundException` is thrown if it is impossible to create the file—for example, because the file name is already used for a directory (folder) name.

A `FileNotFoundException` is a kind of `IOException`, so a `catch` block for an `IOException` would also work and would look more sensible. However, it is best to catch the most specific exception that you can, since that can give more information.

`println`
`print`

As illustrated in Display 10.1, the method `println` of the class `PrintWriter` works the same for writing to a text file as the method `System.out.println` works for writing to the screen. The class `PrintWriter` also has the method `print`, which behaves just like `System.out.print` except that the output goes to a text file. Display 10.2 describes some of the methods in the class `PrintWriter`.

Display 10.2 Some Methods of the Class `PrintWriter` *(Part 1 of 2)*

`PrintWriter` and `FileOutputStream` are in the `java.io` package.

```
public PrintWriter(OutputStream streamObject)
```

This is the only constructor you are likely to need. There is no constructor that accepts a file name as an argument. If you want to create a stream using a file name, you use

```
new PrintWriter(new FileOutputStream(File_Name))
```

When the constructor is used in this way, a blank file is created. If there already was a file named *File_Name*, then the old contents of the file are lost. If you want instead to append new text to the end of the old file contents, use

```
new PrintWriter(new FileOutputStream(File_Name, true))
```

(For an explanation of the argument `true`, read the subsection "Appending to a Text File.")

When used in either of these ways, the `FileOutputStream` constructor, and so the `PrintWriter` constructor invocation, can throw a `FileNotFoundException`, which is a kind of `IOException`.

If you want to create a stream using an object of the class `File`, you can use a `File` object in place of the *File_Name*. (The `File` class will be covered in Section 10.3. We discuss it here so that you will have a more complete reference in this display, but you can ignore the reference to the class `File` until after you've read that section.)

```
public final void println(Argument)[a]
```

The *Argument* can be a string, character, integer, floating-point number, `boolean` value, or any combination of these, connected with + signs. The *Argument* can also be any object, although it will not work as desired unless the object has a properly defined `toString()` method. The *Argument* is output to the file connected to the stream. After the *Argument* has been output, the line ends, and so the next output is sent to the next line.

[a] The modifier `final` is discussed in Chapter 7, which covers inheritance. If you have not yet read Chapter 7, you can safely ignore any reference to `final`.

Display 10.2 Some Methods of the Class `PrintWriter` *(Part 2 of 2)*

`public final void print(`*Argument*`)`

This is the same as `println`, except that this method does not end the line, so the next output will be on the same line.

`public void close()`

Closes the stream's connection to a file. This method calls `flush` before closing the file.

`public void flush()`

Flushes the output stream. This forces an actual physical write to the file of any data that has been buffered and not yet physically written to the file. Normally, you should not need to invoke `flush`.

When your program is finished writing to a file, it should **close** the stream connected to that file. In Display 10.1, the stream connected to the file `stuff.txt` is closed with the statement

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

The class `PrintWriter`, and every other class for file output or file input streams, has a method named `close`. When this method is invoked, the system releases any resources used to connect the stream to the file and does any other housekeeping that is needed. If your program does not close a file before the program ends, Java will close it for you when the program ends, but it is safest to close the file with an explicit call to `close`.

Output streams connected to files are often **buffered**, which means that, rather than physically writing every instance of output data as soon as possible, the data is saved in

buffered

Closing a Text File

When your program is finished writing to a file or reading from a file, it should close the stream connected to that file by invoking the method named `close`.

Syntax:

```
Stream_Object.close();
```

Examples:

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

buffer

a temporary location, known as a **buffer**; when enough data is accumulated in this temporary location, it is physically written to the file. This can add to efficiency, since physical writes to a file can be slow. The method `flush` causes a physical write to the file of any buffered data. The method `close` includes an invocation of the method `flush`.

It may seem like there is no reason to use the method `close` to close a file. If your program ends normally but without closing a file, the system will automatically close it for you. So why should you bother to close files with an explicit call to the method `close`? There are at least two reasons. First, if your program ends abnormally, then Java may not be able to close the file for you. This could damage the file. In particular, if it is an output file, any buffered output will not have been physically written to the file. So the file will be incomplete. The sooner you close a file, the less likely it is that this will happen. Second, if your program writes to a file and later reads from the same file, it must close the file 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 opened for both reading and writing, which we will discuss in Section 10.5.)

Pitfall

A `try` Block Is a Block

Notice that in Display 10.1 we declare the variable `outputStream` outside of the `try` block. If you were to move that declaration inside the `try` block, you would get a compiler error message. Let's see why.

Suppose you replace

```
PrintWriter outputStream = null;
try
{
    outputStream =
       new PrintWriter(new FileOutputStream("stuff.txt"));
}
```

in Display 10.1 with the following:

```
try
{
    PrintWriter outputStream =
       new PrintWriter(new FileOutputStream("stuff.txt"));
}
```

This replacement looks innocent enough, but it makes the variable `outputStream` a local variable for the `try` block, which would mean that you could not use `outputStream` outside of the `try` block. If you make this change and try to compile the changed program, you will get an error message saying that `outputStream` when used outside the `try` block is an undefined identifier.

Pitfall

Overwriting an Output File

When you connect a stream to a text file for writing to the text file, as illustrated by what follows, you always produce an empty file:

```
outputStream =
       new PrintWriter(new FileOutputStream("stuff.txt"));
```

If there is no file named `stuff.txt`, this will create an empty file named `stuff.txt`. If a file named `stuff.txt` already exists, then this will eliminate that file and create a new, empty file named `stuff.txt`. So if there is a file named `stuff.txt` before this file opening, then all the data in that file will be lost. The section "The File Class" tells you how to test to see whether a file already exists so that you can avoid accidentally overwriting a file. The following subsection shows you how to add data to a text file without losing the data that is already in the file.

APPENDING TO A TEXT FILE

When you open a text file for writing in the way we did it in Display 10.1 and a file with the given name already exists, the old contents are lost. However, sometimes you instead want to add the program output to the end of the file. This is called **appending to a file**. If you wanted to append program output to the file `stuff.txt`, you would connect the file to the stream `outputStream` in the following manner:

appending

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

If the file `stuff.txt` does not already exist, Java will create an empty file of that name and append the output to the end of this empty file. So if there is no file named `stuff.txt`, the effect of opening the file is the same as in Display 10.1. However, if the file `stuff.txt` already exists, then the old contents will remain, and the program's output will be placed after the old contents of the file.

When appending to a text file in this way, you would still use the same `try` and `catch` blocks as in Display 10.1.

That second argument of `true` deserves a bit of explanation. Why did the designers use `true` to signal appending? Why not something like the string `"append"`? The reason is that this version of the constructor for the class `FileOutputStream` was designed to also allow you to use a Boolean variable (or expression) to decide whether you append to an existing file or create a new file. For example, the following might be used:

```
System.out.println(
            "Enter A for append or N for a new file:");
char answer;
<Use your favorite way to read a single character into the variable answer.>
```

```
boolean append = (answer == 'A' || answer == 'a');
outputStream =
        new PrintWriter(new FileOutputStream("stuff.txt", append));
```

From this point on, your program writes to the file in exactly the same way that the program in Display 10.1 does. If the user answered with upper- or lowercase A, then any input will be added after the old file contents. If the user answered with upper- or lowercase N (or with anything other than an A), then any old contents of the file are lost.

Opening a Text File for Appending

To create an object of the class `PrintWriter` and connect it to a text file for appending to the end of the text already in the file, proceed as follows.

Syntax:

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

Example:

```
PrintWriter outputStream;
outputStream =
   new PrintWriter(new FileOutputStream("stuff.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.

(If you want to create a stream using an object of the class `File`, you can use a `File` object in place of the *File_Name*. The `File` class is discussed in the section entitled "The File Class.")

When used in this way, the `FileOutputStream` constructor, and so the `PrintWriter` constructor invocation, can throw a `FileNotFoundException`, which is a kind of `IOException`.

Tip

`toString` Helps with Text File Output

In Chapter 4 we noted that if a class has a suitable `toString()` method and `anObject` is an object of that class, then `anObject` can be used as an argument to `System.out.println` and that will produce sensible output.[1] The same thing applies to the methods `println` and `print` of

[1] There is a more detailed discussion of this in Chapter 8, but you need not read Chapter 8 to use this fact.

the class PrintWriter. Both println and print of the class PrintWriter can take any object as an argument and will produce reasonable output so long as the object has a sensible toString() method.

Self-Test Exercises

3. What kind of exception might be thrown by the following, and what would it indicate if this exception were thrown?

```
PrintWriter outputStream =
     new PrintWriter(new FileOutputStream("stuff.txt"));
```

4. Does the class PrintWriter have a constructor that accepts a string (for a file name) as an argument, so that the following code would be legal?

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

5. Write some code that will create a stream named outStream that is a member of the class PrintWriter, and that connects this stream to a text file named sam so that your program can send output to the file. Do this in a way such that the file sam always starts out empty. So, if there already is a file named sam, the old contents of sam are lost.

6. As in exercise 5, write some code that will create a stream named outStream that is a member of the class PrintWriter, and that connects this stream to a text file named sam so that your program can send output to the file. This time, however, do it in such a way that, if the file sam already exists, the old contents of sam will not be lost and the program output will be written after the old contents of the file.

7. The class Person was defined in Display 5.11 of Chapter 5. Suppose mary is an object of the class Person and suppose outputStream is connected to a text file as in Display 10.1. Will the following send sensible output to the file connected to outputStream?

```
outputStream.println(mary);
```

READING FROM A TEXT FILE

BufferedReader

The class BufferedReader is the preferred stream class to use for reading from a text file. The use of BufferedReader is illustrated in Display 10.3, which contains a program that reads two lines from a text file named morestuff.txt and writes them back to the screen. The file morestuff.txt is a text file that a person could have created with a text editor or that a Java program could have created using PrintWriter.

Display 10.3 Reading Input from a Text File *(Part 1 of 2)*

```
1  import java.io.BufferedReader;
2  import java.io.FileReader;
3  import java.io.FileNotFoundException;
4  import java.io.IOException;

5  public class TextFileInputDemo
6  {
7      public static void main(String[] args)
8      {
9         try
10        {
11            BufferedReader inputStream =
12               new BufferedReader(new FileReader("morestuff.txt"));

13            String line = inputStream.readLine();
14            System.out.println(
15                            "The first line read from the file is:");
16            System.out.println(line);
17
18            line = inputStream.readLine();
19            System.out.println(
20                           "The second line read from the file is:");
21            System.out.println(line);
22            inputStream.close();
23        }
24        catch(FileNotFoundException e)
25        {
26            System.out.println("File morestuff.txt was not found");
27            System.out.println("or could not be opened.");
28        }
29        catch(IOException e)
30        {
31            System.out.println("Error reading from morestuff.txt.");
32        }
33     }
34 }
```

FILE morestuff.txt

```
1 2 3
Jack jump over
the candle stick.
```

This file could have been made with a text editor or by another Java program.

Display 10.3 Reading Input from a Text File *(Part 2 of 2)*

SCREEN OUTPUT

```
The first line read from the file is:
1 2 3
The second line read from the file is:
Jack jump over
```

The program opens the text file `morestuff.txt` as follows: opening a file

```
BufferedReader inputStream =
        new BufferedReader(new FileReader("morestuff.txt"));
```

The class `BufferedReader`, like the class `PrintWriter`, has no constructor that takes a file name as its argument, so we need to use another class—in this case, the class `FileReader`—to convert the file name to an object that can be an argument to `BufferedReader`.

OPENING A TEXT FILE FOR READING

You create a stream of the class `BufferedReader` and connect it to a text file for reading as follows:

SYNTAX:

```
BufferedReader Stream_Object =
            new BufferedReader(new FileReader(File_Name));
```

EXAMPLE:

```
BufferedReader inputStream =
        new BufferedReader(new FileReader("morestuff.txt"));
```

After this statement, you can use the methods `readLine` and `read` to read from the file.

When used in this way, the `FileReader` constructor, and hence the `BufferedReader` constructor invocation, can throw a `FileNotFoundException`, which is a kind of `IOException`.

We have already used objects of the class `BufferedReader`, created as follows, for reading from the keyboard: `readLine`

```
BufferedReader keyboard =
                  new BufferedReader(new InputStreamReader(System.in));
```

An object of the class `BufferedReader` that is connected to a text file, as in Display 10.3, has the same method `readLine` that we used to read from the keyboard, but it

reads from a text file rather than the keyboard. This use of `readLine` to read from a text file is illustrated in Display 10.3.

Display 10.4 describes some of the methods in the class `BufferedReader`. Notice that there are only two methods for reading from a text file, `readLine` and `read`. We have already discussed `readLine`.

Display 10.4 Some Methods of the Class `BufferedReader`

`BufferedReader` and `FileReader` are in the `java.io` package.

```
public BufferedReader(Reader readerObject)
```

This is the only constructor you are likely to need. There is no constructor that accepts a file name as an argument. If you want to create a stream using a file name, you use

```
new BufferedReader(new FileReader(File_Name))
```

When used in this way, the `FileReader` constructor, and thus the `BufferedReader` constructor invocation, can throw a `FileNotFoundException`, which is a kind of `IOException`.

The `File` class will be covered in the section entitled "The File Class." We discuss it here so that you will have a more complete reference in this display, but you can ignore the following reference to the class `File` until after you've read that section.

If you want to create a stream using an object of the class `File`, you use

```
new BufferedReader(new FileReader(File_Object))
```

When used in this way, the `FileReader` constructor, and thus the `BufferedReader` constructor invocation, can throw a `FileNotFoundException`, which is a kind of `IOException`.

```
public String readLine() throws IOException
```

Reads a line of input from the input stream and returns that line. If the read goes beyond the end of the file, `null` is returned. (Note that an `EOFException` is not thrown at the end of a file. The end of a file is signaled by returning `null`.)

```
public int read() throws IOException
```

Reads a single character from the input stream and returns that character as an `int` value. If the read goes beyond the end of the file, then `–1` is returned. Note that the value is returned as an `int`. To obtain a `char`, you must perform a type cast on the value returned. The end of a file is signaled by returning `–1`. (All of the "real" characters return a positive integer.)

```
public long skip(long n) throws IOException
```

Skips `n` characters.

```
public void close() throws IOException
```

Closes the stream's connection to a file.

The method `read` reads a single character. But, note that `read` returns a value of type `int` that corresponds to the character read; it does not return the character itself. Thus, to get the character, you must use a type cast, as in

read method

```
char next = (char)(inputStream.read());
```

If `inputStream` is in the class `BufferedReader` and is connected to a text file, this will set `next` equal to the first character in the file that has not yet been read.

Notice that the program in Display 10.3 catches two kinds of exceptions: `FileNotFoundException` and `IOException`. An attempt to open the file may throw a `FileNotFoundException`, and any of the invocations of `inputStream.readLine()` may throw an `IOException`. Because `FileNotFoundException` is a kind of `IOException`, you could use only the `catch` block for `IOException`. However, if you were to do this, then you would get less information if an exception were thrown. If you use only one `catch` block and an exception is thrown, you will not know if the problem occurred when opening the file or when reading from the file after it was opened.

FileNotFoundException

If your program attempts to open a file for reading and there is no such file, then a `FileNotFoundException` is thrown. As you saw earlier in this chapter, a `FileNotFoundException` is also thrown in some other situations. A `FileNotFoundException` is a kind of `IOException`.

Self-Test Exercises

8. Write some code that will create a stream named `fileIn` that is a member of the class `BufferedReader` and that connects the stream to a text file named `joe` so that your program can read input from the text file `joe`.
9. What is the type of a value returned by the method `readLine` in the class `BufferedReader`? What is the type of a value returned by the method `read` in the class `BufferedReader`?
10. Might the methods `read` and `readLine` in the class `BufferedReader` throw an exception? If so, what type of exception?
11. One difference between the `try` blocks in Display 10.1 and Display 10.3 is that the `try` block in Display 10.1 encloses only the opening of the file, while the `try` block in Display 10.3 encloses most of the action in the program. Why is the `try` block in Display 10.3 larger than the one in Display 10.1?
12. Might the following throw an exception that needs to be caught or declared in a `throws` clause?

```
BufferedReader inputStream =
     new BufferedReader(new FileReader("morestuff.txt"));
```

(The stream `inputStream` would be used to read from the text file `morestuff.txt`.)

13. Might the following throw an exception that needs to be caught or declared in a `throws` clause?

```
BufferedReader keyboard =
                    new BufferedReader(new InputStreamReader(System.in));
```

(The stream `keyboard` would be used to read from the keyboard.)

14. In both exercises 12 and 13 the code invokes a constructor for the class `BufferedReader` but one can throw an exception while the other does not throw any exception. How can this be?

Tip

Reading Numbers

The class `BufferedReader` has no methods to read a number from a text file. You can read numbers from a text file using `BufferedReader` in the same way that you read numbers from the keyboard using `BufferedReader`. To read a single number on a line by itself, read it using the method `readLine` and then use `Integer.parseInt`, `Double.parseDouble`, or some similar method to convert the string read to a number. If there are multiple numbers on a single line, read the line using `readLine` and then use `StringTokenizer` to decompose the string into tokens. Then, use `Integer.parseInt` or a similar method to convert each token to a number.

TESTING FOR THE END OF A TEXT FILE

When using the class `BufferedReader`, if your program tries to read beyond the end of the file with either of the methods `readLine` or `read`, then the method returns a special value to signal that the end of the file has been reached. When `readLine` tries to read beyond the end of a file, it returns the value `null`. Thus, your program can test for the end of the file by testing to see if `readLine` returns `null`. This technique is illustrated in Display 10.5. When the method `read` tries to read beyond the end of a file, it returns the value –1. Because the `int` value corresponding to each ordinary character is positive, this can be used to test for the end of a file.

(As you will see when we discuss binary file input, when your program tries to read beyond the end of a binary file, it throws an `EOFException`. When reading from a text file using `BufferedReader`, an `EOFException` is never thrown.)

Display 10.5 Checking for the End of a Text File *(Part 1 of 2)*

```
1  import java.io.BufferedReader;
2  import java.io.FileReader;
3  import java.io.PrintWriter;
4  import java.io.FileOutputStream;
5  import java.io.FileNotFoundException;
6  import java.io.IOException;

7  /**
8   Makes numbered.txt the same as original.txt, but with each line numbered.
9  */
10 public class TextEOFDemo
11 {
12    public static void main(String[] args)
13    {
14       try
15       {
16          BufferedReader inputStream =
17                new BufferedReader(new FileReader("original.txt"));
18          PrintWriter outputStream =
19                new PrintWriter(new FileOutputStream("numbered.txt"));

20          int count = 0;
21          String line = inputStream.readLine();
22          while (line != null)
23          {
24             count++;
25             outputStream.println(count + " " + line);
26             line = inputStream.readLine();
27           }
28          inputStream.close();
29          outputStream.close();
30       }
31       catch(FileNotFoundException e)
32       {
33          System.out.println("Problem opening files.");
34       }
35       catch(IOException e)
36       {
37          System.out.println("Error reading from original.txt.");
38       }
39    }
40 }
```

Display 10.5 Checking for the End of a Text File *(Part 2 of 2)*

FILE original.txt

```
Little Miss Muffet
sat on a tuffet
eating her curves away.
Along came a spider
who sat down beside her
and said "Will you marry me?"
```

FILE numbered.txt (after the program is run)

```
1 Little Miss Muffet
2 sat on a tuffet
3 eating her curves away.
4 Along came a spider
5 who sat down beside her
6 and said "Will you marry me?"
```

If your version of `numbered.txt` has numbered blank lines after 6, that means you had blank lines at the end of `original.txt`.

Checking for the End of a Text File

The method `readLine` of the class `BufferedReader` returns `null` when it tries to read beyond the end of a text file. The method `read` of the class `BufferedReader` returns –1 when it tries to read beyond the end of a text file.

Self-Test Exercises

15. Does the class `BufferedReader` have a method to read an `int` value from a text file?
16. What happens when the method `readLine` in the class `BufferedReader` attempts to read beyond the end of a file? How can you use this to test for the end of a file?
17. What is the type of the value returned by the method `read` in the class `BufferedReader`?
18. What happens when the method `read` in the class `BufferedReader` attempts to read beyond the end of a file? How can you use this to test for the end of a file?
19. Does the program in Display 10.5 work correctly if `original.txt` is an empty file?

PATH NAMES

When giving a file name as an argument to a constructor for opening a file in any of the ways we have discussed, you may use a simple file name, in which case it is assumed that the file is in the same directory (folder) as the one in which the program is run. You can also use a full or relative path name.

path names

A **path name** not only gives the name of the file, but also tells what directory (folder) the file is in. 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 that your program is in. The way that you specify path names depends on your particular operating system.

A typical UNIX path name is

```
/user/sallyz/data/data.txt
```

To create an input stream connected to this file, you use

```
BufferedReader inputStream =
       new BufferedReader(new FileReader("/user/sallyz/data/data.txt"));
```

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

```
C:\dataFiles\goodData\data.txt
```

To create an input stream connected to this file, you use

```
BufferedReader inputStream =
    new BufferedReader(
       new FileReader("C:\\dataFiles\\goodData\\data.txt"));
```

using \, \\, or /

Note that you need to use \\ in place of \, since otherwise Java will interpret a backslash paired with a character, such as \d, as an escape character. Although you must worry about using a backslash (\) in a quoted string, this problem does not occur with path names read in from the keyboard.

One way to avoid these escape-character problems altogether is to always 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 with an operating system that does not match the syntax. Thus, an alternate way to create an input stream connected to the Windows file

```
C:\dataFiles\goodData\data.txt
```

is the following:

```
BufferedReader inputStream =
       new BufferedReader(
          new FileReader("C:/dataFiles/goodData/data.txt"));
```

NESTED CONSTRUCTOR INVOCATIONS

Expressions with two constructors, such as the following, are common when dealing with Java's library of I/O classes:

```
new BufferedReader(new FileReader("original.txt"))
```

This is a manifestation of the general approach to how Java I/O libraries work. Each I/O class serves one or a small number of functions. To obtain full functionality you normally need to combine two (or more) class constructors. For example, in the above example, the object `new FileReader("original.txt")` establishes a connection with the file `original.txt` but provides only very primitive methods for input. The constructor for `BufferedReader` takes this file reader object and adds a richer collection of input methods. In these cases the inner object, such as `new FileReader("original.txt")`, is transformed into an instance variable of the outer object, such as `BufferedReader`. (The instance variable might be literally the object produced by the inner constructor or it might be derived from that object in some way.)

The situation is similar to that of what we did with the class `ConsoleIn` (Display 9.12). The class `ConsoleIn` has an instance variable of type `BufferedReader` that is connected to the keyboard (by `System.in`), and all the input is ultimately done by this `BufferedReader` object, but the methods of the class `ConsoleIn` add a user-friendly interface for this simple input facility.

Self-Test Exercises

20. Of the classes `PrintWriter`, `BufferedReader`, `FileReader`, and `FileOutputStream`, which have a constructor that accepts a file name as an argument?

21. Is the following legal?

```
FileReader readerObject =
           new FileReader("myFile.txt");

BufferedReader inputStream =
           new BufferedReader(readerObject);
```

System.in, System.out, AND System.err

The streams `System.in`, `System.out`, and `System.err` are three streams that are automatically available to your Java code. You have already been using `System.in` and `System.out`. `System.err` is just like `System.out`, except that it has a different name. For example, both of the following statements will send the string `"Hello"` to the screen so the screen receives two lines each containing `"Hello"`:

```
System.out.println("Hello");
System.err.println("Hello");
```

The output stream `System.out` is intended to be used for normal output from code that is not in trouble. `System.err` is meant to be used for error messages.

redirecting output

Having two different standard output streams can be handy when you redirect output. For example, you can redirect the regular output to one file and redirect the error messages to a different file. Java allows you to redirect any of these three standard streams to or from a file (or other I/O device). This is done with the static methods `setIn`, `setOut`, and `setErr` of the class `System`.

For example, suppose your code connects the output stream `errStream` (of a type to be specified later) to a text file. You can then redirect the stream `System.err` to this text file as follows:

```
System.setErr(errStream);
```

If the following appears later in your code:

```
System.out.println("Hello from System.out.");
System.err.println("Hello from System.err.");
```

then `"Hello from System.out."` will be written to the screen, but `"Hello from System.err."` will be written to the file connected to the output stream `errStream`. A simple program illustrating this is given in Display 10.6.

Note that the arguments to the redirecting methods must be of the types shown in the following headings, and these are classes we do not discuss in this book:

```
public static void setIn(InputStream inStream)
public static void setOut(PrintStream outStream)
public static void setErr(PrintStream outStream)
```

None of the input or output streams we constructed in our previous programs are of a type suitable to be an argument to any of these redirection methods. Space constraints keep us from giving any more details on the stream classes that are suitable for producing arguments for these redirection methods. However, you can use Display 10.6 as a model to allow you to redirect either `System.err` or `System.out` to a text file of your choice.

Self-Test Exercises

22. Suppose you want the program in Display 10.6 to send an error message to the screen and regular (`System.out`) output to the file `errormessages.txt`. (Just the reverse of what the program in Display 10.6 does.) How would you change the program in Display 10.6?

23. Suppose you want the program in Display 10.6 to send all output (both `System.out` and `System.err`) to the file `errormessages.txt`. How would you change the program in Display 10.6?

Display 10.6 Redirecting Error Messages

```
1  import java.io.PrintStream;
2  import java.io.FileOutputStream;
3  import java.io.FileNotFoundException;

4  public class RedirectionDemo
5  {
6     public static void main(String[] args)
7     {
8        PrintStream errStream = null;
9        try
10       {
11          errStream =
12               new PrintStream(
13                       new FileOutputStream("errormessages.txt"));
14       }
15       catch(FileNotFoundException e)
16       {
17          System.out.println(
18                   "Error opening file with FileOutputStream.");
19       }

20       System.setErr(errStream);

21       System.err.println("Hello from System.err.");
22       System.out.println("Hello from System.out.");
23       System.err.println("Hello again from System.err.");

24       errStream.close();
25    }
26 }
```

Note the stream classes used.

None of `System.in`, `System.out`, or `System.err` needs to be closed, but the streams you create should be explicitly closed.

FILE errormessages.txt

```
Hello from System.err.
Hello again from System.err.
```

SCREEN OUTPUT

```
Hello from System.out.
```

10.3 The File Class

The scars of others should teach us caution.

Saint Jerome

In this section we describe the class `File`, which is not really an I/O stream class but is often used in conjunction with file I/O. The class `File` is so important to file I/O programming that it was even placed in the `java.io` package.

PROGRAMMING WITH THE `File` CLASS

The `File` class contains methods that allow you to check various properties of a file, such whether there is a file with a specified name, whether the file can be written to, and so forth. Display 10.7 gives a sample program that uses the class `File` with text files. (The class `File` works the same with binary files as it does with text files.)

Note that the `File` class constructor takes a name, known as the **abstract name**, as an (string) argument. So the `File` class really checks properties of names. For example, the method `exists` tests whether there is a file with the abstract name. Moreover, the abstract name may be a potential directory (folder) name, not necessarily a potential file name. For example, the method `isDirectory` tests whether the abstract name is really the name of a directory (folder). The abstract name may be either a relative path name (which includes the case of a simple file name) or a full path name.

Display 10.8 lists some of the methods in the class `File`.

THE `File` CLASS

The `File` class is like a wrapper class for 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: Does the file exist? Does your program have permission to read the file? Does your program have permission to write to the file? Display 10.10 has a summary of some of the methods for the class `File`.

EXAMPLE:

```
File fileObject = new File("data.txt");
if ( ! fileObject.canRead())
    System.out.println("File data.txt is not readable.");
```

Display 10.7 Using the `File` Class *(Part 1 of 2)*

```
1  import java.io.BufferedReader;
2  import java.io.InputStreamReader;
3  import java.io.IOException;
4  import java.io.File;
1  import java.io.PrintWriter;
2  import java.io.FileOutputStream;
3  import java.io.FileNotFoundException;

4  public class FileClassDemo
5  {
6      public static void main(String[] args)
7      {
8          BufferedReader keyboard =
9                    new BufferedReader(new InputStreamReader(System.in));
10         String line = null;
11         String fileName = null;

12         try
13         {
14             System.out.println("I will store a line of text for you.");
15             System.out.println("Enter the line of text:");
16             line = keyboard.readLine();

17             System.out.println("Enter a file name to hold the line:");
18             fileName = keyboard.readLine();
19             fileName = fileName.trim();
20             File fileObject = new File(fileName);
21             while (fileObject.exists())
22             {
23                 System.out.println("There already is a file named "
24                                          + fileName);
25                 System.out.println("Enter a different file name:");
26                 fileName = keyboard.readLine();
27                 fileName = fileName.trim();
28                 fileObject = new File(fileName);
29             }
30         }

31         catch(IOException e)
32         {
33             System.out.println("Error reading from the keyboard. ");
34             System.exit(0);
35         }
```

Display 10.7 **Using the File Class *(Part 2 of 2)***

```
36          PrintWriter outputStream = null;
37          try
38          {
39              outputStream =
40                   new PrintWriter(new FileOutputStream(fileName));
41          }
42          catch(FileNotFoundException e)
43          {
44              System.out.println("Error opening the file " + fileName);
45              System.exit(0);
46          }

47          System.out.println("Writing \""  + line + "\"");
48          System.out.println("to the file "  + fileName);
49          outputStream.println(line);

50          outputStream.close();
51          System.out.println("Writing completed.");
52      }
53  }
```

If you wish, you can use `fileObject` instead of `fileName` as the argument to `FileOutputStream`.

The dialog assumes that there already is a file named myLine.txt but that there is no file named mySaying.txt.

Sample Dialogue

```
I will store a line of text for you.
Enter the line of text:
May the hair on your toes grow long and curly.
Enter a file name to hold the line:
myLine.txt
There already is a file named myLine.txt
Enter a different file name:
mySaying.txt
Writing "May the hair on your toes grow long and curly."
to the file mySaying.txt
Writing completed.
```

Display 10.8 **Some Methods in the Class `File` *(Part 1 of 3)***

`File` is in the `java.io` package.

```
public File(String File_Name)
```

Constructor. *File_Name* can be either a full or a relative path name (which includes the case of a simple file name). *File_Name* is referred to as the **abstract path name**.

Display 10.8 Some Methods in the Class `File` *(Part 2 of 3)*

`public boolean exists()`

Tests whether there is a file with the abstract path name.

`public boolean canRead()`

Tests whether the program can read from the file. Returns `true` if the file named by the abstract path name exists and is readable by the program; otherwise returns `false`.

`public boolean setReadOnly()`

Sets the file represented by the abstract path name to be read only. Returns `true` if successful; otherwise returns `false`.

`public boolean canWrite()`

Tests whether the program can write to the file. Returns `true` if the file named by the abstract path name exists and is writable by the program; otherwise returns `false`.

`public boolean delete()`

Tries to delete the file or directory named by the abstract path name. A directory must be empty to be removed. Returns `true` if it was able to delete the file or directory. Returns `false` if it was unable to delete the file or directory.

`public boolean createNewFile() throws IOException`

Creates a new empty file named by the abstract path name, provided that a file of that name does not already exist. Returns `true` if successful, and returns `false` otherwise.

`public String getName()`

Returns the last name in the abstract path name (that is, the simple file name). Returns the empty string if the abstract path name is the empty string.

`public String getPath()`

Returns the abstract path name as a `String` value.

`public boolean renameTo(File` *New_Name*`)`

Renames the file represented by the abstract path name to *New_Name*. Returns `true` if successful; otherwise returns `false`. *New_Name* can be a relative or absolute path name. This may require moving the file. Whether or not the file can be moved is system dependent.

`public boolean isFile()`

Returns `true` if a file exists that is named by the abstract path name and the file is a normal file; otherwise returns `false`. The meaning of *normal* is system dependent. Any file created by a Java program is guaranteed to be normal.

Display 10.8 Some Methods in the Class `File` *(Part 3 of 3)*

`public boolean isDirectory()`

Returns `true` if a directory (folder) exists that is named by the abstract path name; otherwise returns `false`.

`public boolean mkdir()`

Makes a directory named by the abstract path name. Will not create parent directories. See `mkdirs`. Returns `true` if successful; otherwise returns `false`.

`public boolean mkdirs()`

Makes a directory named by the abstract path name. Will create any necessary but nonexistent parent directories. Returns `true` if successful; otherwise returns `false`. Note that if it fails, then some of the parent directories may have been created.

`public long length()`

Returns the length in bytes of the file named by the abstract path name. If the file does not exist or the abstract path name names a directory, then the value returned is not specified and may be anything.

Self-Test Exercises

24. Write a complete (although simple) Java program that tests whether or not the directory (folder) containing the program also contains a file named `Sally.txt`. The program has no input and the only output tells whether or not there is a file named `Sally.txt`.
25. Write a complete Java program that asks the user for a file name, tests whether the file exists, and, if the file exists, asks the user whether or not it should be deleted. It then either deletes or does not delete the file as the user requests.

10.4 Binary Files

A little more than kin, and less than kind.

William Shakespeare, *Hamlet*

Binary files store data in the same format that is used in the computer's memory to store the values of variables. For example, a value of type `int` is stored as the same sequence of bytes (same sequence of zeros and ones) whether it is stored in an `int` variable in memory or in a binary file. So, no conversion of any kind needs to be performed when you store or retrieve a value in a binary file. This is why binary files can be handled more efficiently than text files.

Java binary files are unlike binary files in other programming languages in that they are portable. A binary file created by a Java program can be moved from one computer to another and still be read by a Java program—but only by a Java program. They cannot normally be read with a text editor or with a program written in any programming language other than Java.

The preferred stream classes for processing binary files are `ObjectInputStream` and `ObjectOutputStream`. Each has methods to read or write data one byte at a time. These streams can also automatically 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 not just bytes but were strings or items of any of Java's primitive data types, such as `int`, `char`, and `double`, or even objects of classes you define. If you do not need to access your files using an editor, then the easiest and most efficient way to read data from and write data to files is to use binary files in the way we describe here.

We conclude this section with a discussion of how you can use `ObjectOutputStream` and `ObjectInputStream` to write and later read objects of any class you define. This will let your code store objects of the classes you define in binary files and later read them back, all with the same convenience and efficiency that you get when storing strings and primitive type data in binary files.

WRITING SIMPLE DATA TO A BINARY FILE

The class `ObjectOutputStream` is the preferred stream class for writing to a binary file.[2] An object of the class `ObjectOutputStream` has methods to write strings and values of any of the primitive types to a binary file. Display 10.9 shows a sample program that writes values of type `int` to a binary file. Display 10.10 describes the methods used for writing data of other types to a binary file.

Notice that almost all the code in the sample program in Display 10.9 is in a `try` block. Any part of the code that does binary file I/O in the ways we are describing can throw an `IOException`.

The output stream for writing to the binary file `numbers.dat` is created and named with the following:

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

opening a file

As with text files, this is called **opening the 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

[2] `DataOutputStream` is also widely used and behaves exactly as we describe for `ObjectOutputStream` in this section. However, the techniques given in the subsections "Binary I/O of Objects" and "Array Objects in Binary Files" only work for `ObjectOutputStream`; they do not work for `DataOutputStream`.

Display 10.9 Writing to a Binary File

```
1  import java.io.ObjectOutputStream;
2  import java.io.FileOutputStream;
3  import java.io.IOException;

4  public class BinaryOutputDemo
5  {
6     public static void main(String[] args)
7     {
8        try
9        {
10          ObjectOutputStream outputStream =
11              new ObjectOutputStream(new FileOutputStream("numbers.dat"));

12          int i;
13          for (i = 0; i < 5; i++)
14              outputStream.writeInt(i);

15          System.out.println("Numbers written to the file numbers.dat.");
16          outputStream.close();
17       }
18       catch(IOException e)
19       {
20          System.out.println("Problem with file output.");
21       }
22    }
23 }
```

FILE REPRESENTATION (after program is run)

```
0
1
2
3
4
```

This is a binary file. It really contains representations of each number as bytes, that is, zeros and ones, and is read as bytes. You cannot read this file with your text editor.

starts out empty. The situation is basically the same as what you learned for text files, except that we're using a different class.

As is typical of Java I/O classes, the constructor for the class `ObjectOutputStream` takes another I/O class object as an argument, in this case an anonymous argument of the class `FileOutputStream`.

The class `ObjectOutputStream` does not have a method named `println`, as we had with text file output and screen output. However, as shown in Display 10.9, an object of

Opening a Binary File for Output

You create a stream of the class `ObjectOutputStream` and connect it to a binary file as follows:

Syntax:

```
ObjectOutputStream Output_Stream_Name =
    new ObjectOutputStream(new FileOutputStream(File_Name));
```

The constructor for `FileOutputStream` may throw a `FileNotFoundException`, which is a kind of `IOException`. If the `FileOutputStream` constructor succeeds, then the constructor for `ObjectOutputStream` may throw a different `IOException`. A single `catch` block for `IOException` would cover all cases.

Examples:

```
ObjectOutputStream myOutputStream =
           new ObjectOutputStream(new FileOutputStream("mydata.dat"));
```

After opening the file, you can use the methods of the class `ObjectOutputStream` (Display 10.10) to write to the file.

writeInt

the class `ObjectOutputStream` does have a method named `writeInt` that can write a single `int` value to a file, and it also has the other output methods described in Display 10.10.

In Display 10.9, we made it look as though the numbers in the file `numbers.dat` were written one per line in a human-readable form. That is not what happens, however. There are no lines or other separators between the numbers. Instead, the numbers are written in the file one immediately after the other, and they are encoded as a sequence of bytes in the same way that the numbers would be encoded in the computer's main memory. These coded `int` values cannot be read using your editor. Realistically, they can be read only by another Java program.

writeChar

You can use a stream from the class `ObjectOutputStream` to output values of any primitive type and also to write data of the type `String`. Each primitive data type has a corresponding write method in the class `ObjectOutputStream`. We have already mentioned the write methods for outputting `int` values. The methods for the other primitive types are completely analogous to `writeInt`. For example, the following would write a `double` value, a `boolean` value, and a `char` value to the binary file connected to the `ObjectOutputStream` object `outputStream`:

```
outputStream.writeDouble(9.99);
outputStream.writeBoolean(false);
outputStream.writeChar((int)'A');
```

Display 10.10 Some Methods in the Class `ObjectOutputStream` *(Part 1 of 2)*

`ObjectOutputStream` and `FileOutputStream` are in the `java.io` package.

```
public ObjectOutputStream(OutputStream streamObject)
```

There is no constructor that takes a file name as an argument. If you want to create a stream using a file name, you use

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

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

If you want to create a stream using an object of the class `File`, you use

```
new ObjectOutputStream(new FileOutputStream(File_Object))
```

The constructor for `FileOutputStream` may throw a `FileNotFoundException`, which is a kind of `IOException`. If the `FileOutputStream` constructor succeeds, then the constructor for `ObjectOutputStream` may throw a different `IOException`.

```
public void writeInt(int n) throws IOException
```

Writes the `int` value `n` to the output stream.

```
public void writeShort(short n) throws IOException
```

Writes the `short` 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 n) throws IOException
```

Writes the `char` value `n` to the output stream. Note that it expects its argument to be an `int` value. However, if you simply use the `char` value, then Java will automatically type cast it to an `int` value. The following are equivalent:

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

and

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

```
public void writeBoolean(boolean b) throws IOException
```

Writes the `boolean` value `b` to the output stream.

Display 10.10 Some Methods in the Class `ObjectOutputStream` *(Part 2 of 2)*

`public void writeUTF(String aString) throws IOException` Writes the `String` value `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`.
`public void writeObject(Object anObject) throws IOException` Writes its argument to the output stream. The object argument should be an object of a serializable class, a concept discussed later in this chapter. Throws various `IOExceptions`.
`public void close() throws IOException` Closes the stream's connection to a file. This method calls `flush` before closing the file.
`public void flush() throws IOException` Flushes the output stream. This forces an actual physical write to the file of any data that has been buffered and not yet physically written to the file. Normally, you should not need to invoke `flush`.

The method `writeChar` has one possibly surprising property: It expects its argument to be of type `int`. So if you start with a value of type `char`, the `char` value can be type cast to an `int` before it is given to the method `writeChar`. For example, to output the contents of a `char` variable named `symbol`, you can use

```
outputStream.writeChar((int)symbol);
```

In actual fact, you do not need to write in the type cast to an `int`, because Java automatically performs a type cast from a `char` value to an `int` value for you. So, the following is equivalent to the above invocation of `writeChar`:

```
outputStream.writeChar(symbol);
```

writeUTF for strings

To output a value of type `String`, you use the method `writeUTF`. For example, if `outputStream` is a stream of type `ObjectOutputStream`, the following will write the string `"Hello friend."` to the file connected to that stream:

```
outputStream.writeUTF("Hello friend.");
```

You may write output of different types to the same file. So, you may write a combination of, for example, `int`, `double`, and `String` values. However, mixing types in a file does require special care to make it possible to read them back out of the file. To read them back, you need to know the order in which the various types appear in the file, because, as you will see, a program that reads from the file will use a different method to read data of each different type.

closing a binary file

Note that, as illustrated in Display 10.9 and as you will see shortly, you close a binary output or input stream in the same way that you close a stream connected to a text file.

UTF AND `writeUTF`

Recall that Java uses the Unicode character set, a set of characters that includes many characters used in languages whose character sets are different from English. Readers of this book are undoubtedly using editors and operating systems that use the ASCII character set, which is the character set normally used for English and for our Java programs. The ASCII character set is a subset of the Unicode character set, so the Unicode character set has a lot of characters you probably do not need. There is a standard way of encoding all the Unicode characters, but for English-speaking countries, it is not a very efficient coding scheme. The UTF coding scheme is an alternative scheme that still codes all Unicode characters but that favors the ASCII character set. The UTF coding method gives short, efficient codes for the ASCII characters. The price is that it gives long, inefficient codes to the other Unicode characters. However, because you probably do not use the other Unicode characters, this is a very favorable trade-off. The method `writeUTF` uses the UTF coding method to write strings to a binary file.

The method `writeInt` writes integers into a file using the same number of bytes—that is, the same number of zeros and ones—to store any integer. Similarly, the method `writeLong` uses the same number of bytes to store each value of type `long`. (But the methods `writeInt` and `writeLong` use a different number of bytes from each other.) The situation is the same for all the other write methods that write primitive types to binary files. However, the method `writeUTF` uses differing numbers of bytes to store different strings in a file. Longer strings require more bytes than shorter strings. This can present a problem to Java, because there are no separators between data items in a binary file. The way that Java manages to make this work is 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 `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 that was written using the UTF coding method.)

The situation with `writeUTF` is even a little more complicated than what we discussed in the previous paragraph. 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, and you are undoubtedly using only ASCII characters, so this difference is more theoretical than real to you now.

Self-Test Exercises

26. How do you open the binary file `bindata.dat` so that it is connected to an output stream of type `ObjectOutputStream` that is named `outputThisWay`?

27. Give two statements that will write the values of the two `double` variables `v1` and `v2` to the file `bindata.dat`. Use the stream `outputThisWay` that you created as the answer to exercise 26.

28. Give a statement that will write the string value "Hello" to the file bindata.dat. Use the stream outputThisWay that you created as the answer to exercise 26.
29. Give a statement that will close the stream outputThisWay created as the answer to exercise 26.

READING SIMPLE DATA FROM A BINARY FILE

The stream class ObjectInputStream is used to read from a file that has been written to using ObjectOutputStream. Display 10.11 gives some of the most commonly used methods for this class. If you compare that table with the methods for ObjectOutputStream given in Display 10.10, you will see that each output method in ObjectOutputStream has a corresponding input method in ObjectInputStream. For example, if you write an integer to a file using the method writeInt of ObjectOutputStream, then you can read that integer back with the method readInt of ObjectInputStream. If you write a number to a file using the method writeDouble of ObjectOutputStream, then you can read that number back with the method readDouble of ObjectInputStream, and so forth. Display 10.12 gives an example of using readInt in this way.

Display 10.11 Some Methods in the Class ObjectInputStream *(Part 1 of 3)*

The classes ObjectInputStream and FileInputStream are in the java.io package.

```
public ObjectInputStream(InputStream streamObject)
```

There is no constructor that takes a file name as an argument. If you want to create a stream using a file name, you use

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

Alternatively, you can use an object of the class File in place of the *File_Name*, as follows:

```
new ObjectInputStream(new FileInputStream(File_Object))
```

The constructor for FileInputStream may throw a FileNotFoundException, which is a kind of IOException. If the FileInputStream constructor succeeds, then the constructor for ObjectInputStream may throw a different IOException.

```
public int readInt() throws IOException
```

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

```
public int readShort() throws IOException
```

Reads a short value from the input stream and returns that short value. If readShort tries to read a value from the file and that value was not written using the method writeShort of the class ObjectOutputStream (or written in some equivalent way), then problems will occur. If an attempt is made to read beyond the end of the file, an EOFException is thrown.

Display 10.11 Some Methods in the Class `ObjectInputStream` *(Part 2 of 3)*

`public long readLong() throws IOException`

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

`public double readDouble() throws IOException`

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

`public float readFloat() throws IOException`

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

`public char readChar() throws IOException`

Reads a `char` value from the input stream and returns that `char` value. If `readChar` tries to read a value from the file and that value was not written using the method `writeChar` of the class `ObjectOutputStream` (or written in some equivalent way), then problems will occur. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.

`public boolean readBoolean() throws IOException`

Reads a `boolean` value from the input stream and returns that `boolean` value. If `readBoolean` tries to read a value from the file and that value was not written using the method `writeBoolean` of the class `ObjectOutputStream` (or written in some equivalent way), then problems will occur. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.

`public String readUTF() throws IOException`

Reads a `String` value from the input stream and returns that `String` value. If `readUTF` tries to read a value from the file and that value was not written using the method `writeUTF` of the class `ObjectOutputStream` (or written in some equivalent way), then problems will occur. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.

`Object readObject( ) throws ClassNotFoundException, IOException`

Reads an object from the input stream. The object read should have been written using `writeObject` of the class `ObjectOutputStream`. Throws a `ClassNotFoundException` if the class of a serialized object cannot be found. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown. May throw various other `IOExceptions`.

`public int skipBytes(int n) throws IOException`

Skips `n` bytes.

Display 10.11 Some Methods in the Class `ObjectInputStream` *(Part 3 of 3)*

```
public void close() throws IOException
```

Closes the stream's connection to a file.

The input stream for reading from the binary file `numbers.dat` is opened as follows:

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

Note that this is identical to how we opened a file using `ObjectOutputStream` in Display 10.9, except that here we've used the classes `ObjectInputStream` and `FileInputStream` instead of `ObjectOutputStream` and `FileOutputStream`.

Opening a Binary File for Reading

You create a stream of the class `ObjectInputStream` and connect it to a binary file as follows:

Syntax:

```
ObjectInputStream Input_Stream_Name =
      new ObjectInputStream(new FileInputStream(File_Name));
```

The constructor for `FileInputStream` may throw a `FileNotFoundException`, which is a kind of `IOException`. If the `FileInputStream` constructor succeeds, then the constructor for `ObjectInputStream` may throw a different `IOException`.

Examples:

```
ObjectInputStream inputFile =
          new ObjectInputStream(new FileInputStream("somefile.dat"));
```

After this, you can use the methods in Display 10.11 to read from the file.

reading multiple types

`ObjectInputStream` allows you to read input values 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 a mess. For example, if your program writes an integer using `writeInt`, then any program that reads that integer should read it using `readInt`. If you instead use `readLong` or `readDouble`, your program will misbehave.

closing a binary file

Note that, as illustrated in Display 10.12, you close a binary input stream in the same way that you close all the other I/O streams we have seen.

Display 10.12 Reading from a Binary File

```
1  import java.io.ObjectInputStream;
2  import java.io.FileInputStream;
3  import java.io.IOException;
4  import java.io.FileNotFoundException;

5  public class BinaryInputDemo
6  {
7    public static void main(String[] args)
8    {
9       try
10      {
11          ObjectInputStream inputStream =
12            new ObjectInputStream(new FileInputStream("numbers.dat"));

13          System.out.println("Reading the file numbers.dat.");
14          int n1 = inputStream.readInt();
15          int n2 = inputStream.readInt();

16          System.out.println("Numbers read from file:");
17          System.out.println(n1);
18          System.out.println(n2);
19          inputStream.close();
20      }
21      catch(FileNotFoundException e)
22      {
23          System.out.println("Cannot find file numbers.dat.");
24      }
25      catch(IOException e)
26      {
27          System.out.println("Problems with input from numbers.dat.");
28      }
29      System.out.println("End of program.");
30   }
31 }
```

Sample Dialogue

```
Reading the file numbers.dat.
Numbers read from file:
0
1
End of program.
```

Assumes the program in Display 10.9 was run to create the file numbers.dat.

Self-Test Exercises

30. Write code to open the binary file named `someStuff` and connect it to a `ObjectInputStream` object named `inputThing` so it is ready for reading.
31. Give a statement that will read a number of type `double` from the file `someStuff` and place the value in a variable named `number`. Use the stream `inputThing` that you created in exercise 30. (Assume the first thing written to the file was written using the method `writeDouble` of the class `ObjectOutputStream` and assume `number` is of type `double`.)
32. Give a statement that will close the stream `inputThing` created in exercise 30.
33. Can one program write a number to a file using `writeInt` and then have another program read that number using `readLong`? Can a program read that number using `readDouble`?
34. Can you use `readUTF` to read a string from a text file?

CHECKING FOR THE END OF A BINARY FILE

EOF-Exception

All of the `ObjectInputStream` methods that read from a binary file will throw an `EOFException` when they try to read beyond the end of a file. So, your code can test for the end of a binary file by catching an `EOFException` as illustrated in Display 10.13.

In Display 10.13 the reading is placed in an "infinite loop" through the use of `true` as the Boolean expression in the `while` loop. The loop is not really infinite, because when the end of the file is reached, an exception is thrown, and that ends the entire `try` block and passes control to the `catch` block.

EOFException

If your program is reading from a binary file using any of the methods listed in Display 10.11 for the class `ObjectInputStream`, and your program attempts to read beyond the end of the file, then an `EOFException` is thrown. This can be used to end a loop that reads all the data in a file.

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

Pitfall

Checking for the End of a File in the Wrong Way

Different file-reading methods check for the end of a file in different ways. If you test for the end of a file in the wrong way, one of two things will probably happen: Your program will either go into an unintended infinite loop or terminate abnormally.

Display 10.13 Using EOFException *(Part 1 of 2)*

```
1  import java.io.ObjectInputStream;
2  import java.io.FileInputStream;
3  import java.io.EOFException;
4  import java.io.IOException;
5  import java.io.FileNotFoundException;

6  public class EOFDemo
7  {
8      public static void main(String[] args)
9      {
10         try
11         {
12             ObjectInputStream inputStream =
13               new ObjectInputStream(new FileInputStream("numbers.dat"));
14             int number;
15             System.out.println("Reading numbers in numbers.dat");
16             try
17             {
18                 while (true)
19                 {
20                     number = inputStream.readInt();
21                     System.out.println(number);
22                 }
23             }
24             catch(EOFException e)
25             {
26                 System.out.println("No more numbers in the file.");
27             }
28             inputStream.close();
29         }
30         catch(FileNotFoundException e)
31         {
32             System.out.println("Cannot find file numbers.dat.");
33         }
34         catch(IOException e)
35         {
36             System.out.println("Problem with input from file numbers.dat.");
37         }
38     }
39 }
```

Display 10.13 Using EOFException *(Part 2 of 2)*

Sample Dialogue

```
Reading numbers in numbers.dat
0
1
2
3
4
No more numbers in the file.
```

Assumes the program in Display 10.9 was run to create the file numbers.dat.

For the classes discussed in this book, the following rules apply: If your program is reading from a binary file, then an `EOFException` will be thrown when the reading goes beyond the end of the file. If your program is reading from a text file, then some special value, such as `null`, will be returned when your program attempts to read beyond the end of the file, and no `EOFException` will be thrown.

Self-Test Exercises

35. When opening a binary file for output in the ways discussed in this chapter, might an exception be thrown? What kind of exception? When opening a binary file for input in the ways discussed in this chapter, might an exception be thrown? What kind of exception?

36. Suppose a binary file contains three numbers written to the file with the method `writeDouble` of the class `ObjectOutputStream`. Suppose further that your program reads all three numbers with three invocations of the method `readDouble` of the class `ObjectInputStream`. When will an `EOFException` be thrown? Right after reading the third number? When your program tries to read a fourth number? Some other time?

37. The following appears in the program in Display 10.13:

```
try
{
    while (true)
    {
        number = inputStream.readInt();
        System.out.println(number);
    }
}
catch(EOFException e)
{
```

```
            System.out.println("No more numbers in the file.");
        }
```

Why isn't this an infinite loop?

BINARY I/O OF OBJECTS

You can output objects of classes you define as easily as you output `int` values using `writeInt`, and you can later read the objects back into your program as easily as you read `int` values with the method `readInt`. For you to be able to do this, the class of objects that your code is writing and reading must implement the `Serializable` interface.

We will discuss interfaces in general in Chapter 13. However, the `Serializable` interface is particularly easy to use and requires no knowledge of interfaces. All you need to do to make a class implement the `Serializable` interface is add the two words `implements Serializable` to the heading of the class definition, as in the following example:

`Serializable` interface

```
public class Person implements Serializable
{
```

The `Serializable` interface is in the same `java.io` package that contains all the I/O classes we have discussed in this chapter. For example, in Display 10.14 we define a toy class named `SomeClass` that implements the `Serializable` interface. We will explain the effect of the `Serializable` interface a bit later in this chapter, but first let's see how you do binary file I/O with a serializable class such as this class `SomeClass` in Display 10.14.

Display 10.15 illustrates how class objects can be written to and read from a binary file. To write an object of a class such as `SomeClass` to a binary file, you simply use the method `writeObject` of the class `ObjectOutputStream`. You use `writeObject` in the same way that you use the other methods of the class `ObjectOutputStream`, such as `writeInt`, but you use an object as the argument.

`writeObject`

If an object is written to a file with `writeObject`, then it can be read back out of the file with `readObject` of the stream class `ObjectInputStream`, as also illustrated in Display 10.14. The method `readObject` returns its value as an object of type `Object`. If you want to use the values returned by `readObject` as an object of a class like `SomeClass`, you must do a type cast, as shown in Display 10.15.

`readObject`

THE `Serializable` INTERFACE

A class that implements the `Serializable` interface is said to be a **serializable** class. To use objects of a class with `writeObject` and `readObject`, that class must be serializable. But, to make the class serializable, we change nothing in the class. All we do is add the phrase `implements Serializable`. This phrase tells the run-time system that it is okay to treat objects of the class in a particular way when doing file I/O. If a class is serializable,

serializable

Display 10.14 A Serializable Class

```
1  import java.io.Serializable;

2  public class SomeClass implements Serializable
3  {
4      private int number;
5      private char letter;

6      public SomeClass()
7      {
8          number = 0;
9          letter = 'A';
10     }

11     public SomeClass(int theNumber, char theLetter)
12     {
13         number = theNumber;
14         letter = theLetter;
15     }

16     public String toString()
17     {
18         return "Number = " + number
19                            + " Letter = " + letter;
20     }

21 }
```

Display 10.15 Binary File I/O of Objects *(Part 1 of 3)*

```
1  import java.io.ObjectOutputStream;
2  import java.io.FileOutputStream;
1  import java.io.ObjectInputStream;
2  import java.io.FileInputStream;
3  import java.io.IOException;
4  import java.io.FileNotFoundException;

5  /**
6   Demonstrates binary file I/O of serializable class objects.
7  */
8  public class ObjectIODemo
9  {
```

Display 10.15 Binary File I/O of Objects *(Part 2 of 3)*

```
10      public static void main(String[] args)
11      {

12         try
13         {
14             ObjectOutputStream outputStream =
15                new ObjectOutputStream(new FileOutputStream("datafile"));

16             SomeClass oneObject = new SomeClass(1, 'A');
17             SomeClass anotherObject = new SomeClass(42, 'Z');

18             outputStream.writeObject(oneObject);
19             outputStream.writeObject(anotherObject);

20             outputStream.close();

21             System.out.println("Data sent to file.");
22         }
23         catch(IOException e)
24         {
25            System.out.println("Problem with file output.");
26         }

27         System.out.println(
28                 "Now let's reopen the file and display the data.");

29         try
30         {
31             ObjectInputStream inputStream =
32               new ObjectInputStream(new FileInputStream("datafile"));
                                                  Notice the type casts.

33             SomeClass readOne = (SomeClass)inputStream.readObject();
34             SomeClass readTwo = (SomeClass)inputStream.readObject();

35             System.out.println("The following were read from the file:");
36             System.out.println(readOne);
37             System.out.println(readTwo);
38         }
39         catch(FileNotFoundException e)
40         {
41             System.out.println("Cannot find datafile.");
42         }
43         catch(ClassNotFoundException e)
44         {
```

Display 10.15 Binary File I/O of Objects *(Part 3 of 3)*

```
45              System.out.println("Problems with file input.");
46          }
47          catch(IOException e)
48          {
49              System.out.println("Problems with file input.");
50          }

51          System.out.println("End of program.");
52      }
53  }
```

SAMPLE DIALOGUE

```
Data sent to file.
Now let's reopen the file and display the data.
The following were read from the file:
Number = 1 Letter = A
Number = 42 Letter = Z
End of program.
```

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, then after the first time, Java writes only the serial number for the object and not a full description of the object's data. This makes file I/O more efficient and makes the files smaller. When read back out with a stream of type `ObjectInputStream`, duplicate serial numbers are returned as references to the same object. Note that this 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, then the two objects that are read will again 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 back.

class instance variables

When a serializable class has instance variables of a class type, then the classes for the instance variables must also be serializable, and so on for all levels of class instance variables within classes. So, a class is not serializable unless the classes for all instance variables are also serializable.

Why aren't all classes made serializable? For security reasons. The serial number system makes it easier for programmers to get access to the object data written to secondary storage. Also, for some classes it may not make sense to write objects to secondary storage, since they would be meaningless when read out again later. For example, if the object contains system-dependent data, the data may be meaningless when later read out.

Pitfall

Mixing Class Types in the Same File

The best way to write and read objects using `ObjectOutputStream` and `ObjectInputStream` is to store only data of one class type in any one file. If you store objects of multiple class types or even objects of only one class type mixed in with primitive type data, it has been our experience that the system can get confused and you could lose data.

ARRAY OBJECTS IN BINARY FILES

An array is an object and hence a suitable argument for `writeObject`. An entire array can be saved to a binary file using `writeObject` and later read using `readObject`. When doing so, if the array has a base type that is a class, then the class must be serializable. This means that if you store all your data for one serializable class in a single array, then you can output all your data to a binary file with one invocation of `writeObject`.

This way of storing an array in a binary file is illustrated in Display 10.16. Note that the base class type, `SomeClass`, is serializable. Also, notice the type cast that uses the array type `SomeClass[]`. Since `readObject` returns its value as an object of type `Object`, it must be type cast to the correct array type.

Display 10.16 File I/O of an Array Object *(Part 1 of 3)*

```
1  import java.io.ObjectOutputStream;
2  import java.io.FileOutputStream;
3  import java.io.ObjectInputStream;
4  import java.io.FileInputStream;
5  import java.io.IOException;
6  import java.io.FileNotFoundException;

7  public class ArrayIODemo
8  {

9      public static void main(String[] args)
10     {
11         SomeClass[] a = new SomeClass[2];
12         a[0] = new SomeClass(1, 'A');
13         a[1] = new SomeClass(2, 'B');

14         try
15         {
16             ObjectOutputStream outputStream =
17                 new ObjectOutputStream(new FileOutputStream("arrayfile"));
```

Display 10.16 File I/O of an Array Object *(Part 2 of 3)*

```
18              outputStream.writeObject(a);
19              outputStream.close();
20          }
21          catch(IOException e)
22          {
23              System.out.println("Error writing to file.");
24              System.exit(0);
25          }

26           System.out.println(
27                     "Array written to file arrayfile.");

28          System.out.println(
29                "Now let's reopen the file and display the array.");

30          SomeClass[] b = null;

31          try
32          {
33              ObjectInputStream inputStream =
34                   new ObjectInputStream(new FileInputStream("arrayfile"));
35              b = (SomeClass[])inputStream.readObject();
36              inputStream.close();
37          }
38          catch(FileNotFoundException e)
39          {
40              System.out.println("Cannot find file arrayfile.");
41              System.exit(0);
42          }
43          catch(ClassNotFoundException e)
44          {
45              System.out.println("Problems with file input.");
46              System.exit(0);
47          }
48          catch(IOException e)
49          {
50              System.out.println("Problems with file input.");
51              System.exit(0);
52          }

53          System.out.println(
54                "The following array elements were read from the file:");
55          int i;
56          for (i = 0; i < b.length; i++)
57              System.out.println(b[i]);
```

Notice the type cast.

Display 10.16 **File I/O of an Array Object *(Part 3 of 3)***

```
58            System.out.println("End of program.");
59        }
60    }
```

SAMPLE DIALOGUE

```
Array written to file arrayfile.
Now let's reopen the file and display the array.
The following array elements were read from the file:
Number = 1 Letter = A
Number = 2 Letter = B
End of program.
```

Self-Test Exercises

38. How do you make a class implement the `Serializable` interface?
39. What import statement do you need to be able to use the `Serializable` interface?
40. What is the return type for the method `readObject` of the class `ObjectInputStream`?
41. Is an array of type `Object`?

10.5 Random Access to Binary Files

Any time, any where.

Common response to a challenge for a confrontation

The streams for sequential access to files, which we discussed in the previous sections of this chapter, are the ones most often used for file access in Java. However, some applications that require very rapid access to records in very large databases require some sort of random access to particular parts of a file. Such applications might best be done with specialized database software. But, perhaps you are given the job of writing such a package in Java, or perhaps you are just curious about how such things are done in Java. Java does provide for random access to files so that your program can both read from and write to random locations in a file. In this section we will describe simple uses of random access to files.

READING AND WRITING TO THE SAME FILE

If you want random access to both read and write to a file in Java, you use the stream class `RandomAccessFile`, which is in the `java.io` package like all other file I/O classes.

A random access file consists of a sequence of numbered bytes. There is a kind of marker called the **file pointer** that is always positioned at one of these bytes. All reads and writes take place starting at the location of the file pointer. You can move the file pointer to a new location with the method `seek`.

Although a random access file is byte oriented, there are methods to allow for reading or writing values of the primitive types and of string values to a random access file. In fact, these are the same methods as those we already used for sequential access files, as previously discussed. A `RandomAccessFile` stream has methods `writeInt`, `writeDouble`, `writeUTF`, and so forth as well as methods `readInt`, `readDouble`, `readUTF`, and so forth. However, the class `RandomAccessFile` does not have the methods `writeObject` or `readObject`. The most important methods of the class `RandomAccessFile` are given in Display 10.17. A demonstration program for random access files is given in Display 10.18.

Display 10.17 Some Methods of the Class `RandomAccessFile` *(Part 1 of 3)*

The class `RandomAccessFile` is in the `java.io` package.

```
public RandomAccessFile(String fileName, String mode)
public RandomAccessFile(File fileObject, String mode)
```

Opens the file, does not delete data already in the file, but does position the file pointer at the first (zeroth) location.

The mode must be one of the following:

`"r"` Open for reading only.

`"rw"` Open for reading and writing.

`"rws"` Same as `"rw"`, and also requires that every update to the file's content or metadata be written synchronously to the underlying storage device.

`"rwd"` Same as `"rw"`, and also requires that every update to the file's content be written synchronously to the underlying storage device.

`"rws"` and `"rwd"` are not covered in this book.

```
public long getFilePointer() throws IOException
```

Returns the current location of the file pointer. Locations are numbered starting with `0`.

```
public void seek(long location) throws IOException
```

Moves the file pointer to the specified `location`.

```
public long length() throws IOException
```

Returns the length of the file.

Display 10.17 Some Methods of the Class RandomAccessFile *(Part 2 of 3)*

`public void setLength(long newLength) throws IOException`

Sets the length of this file.

If the present length of the file as returned by the `length` method is greater than the `newLength` argument, then the file will be truncated. In this case, if the file pointer location as returned by the `getFilePointer` method is greater than `newLength`, then after this method returns, the file pointer location will be equal to `newLength`.

If the present length of the file as returned by the `length` method is smaller than `newLength`, then the file will be extended. In this case, the contents of the extended portion of the file are not defined.

`public void close() throws IOException`

Closes the stream's connection to a file.

`public void write(int b) throws IOException`

Writes the specified byte to the file.

`public void write(byte[] a) throws IOException`

Writes `a.length` bytes from the specified byte array to the file.

`public final void writeByte(byte b) throws IOException`

Writes the `byte` `b` to the file.

`public final void writeShort(short n) throws IOException`

Writes the `short` `n` to the file.

`public final void writeInt(int n) throws IOException`

Writes the `int` `n` to the file.

`public final void writeLong(long n) throws IOException`

Writes the `long` `n` to the file.

`public final void writeDouble(double d) throws IOException`

Writes the `double` `d` to the file.

`public final void writeFloat(float f) throws IOException`

Writes the `float` `f` to the file.

`public final void writeChar(char c) throws IOException`

Writes the `char` `c` to the file.

`public final void writeBoolean(boolean b) throws IOException`

Writes the `boolean` `b` to the file.

Display 10.17 Some Methods of the RandomAccessFile *(Part 3 of 3)*

`public final void writeUTF(String s) throws IOException` Writes the `String` `s` to the file.
`public int read() throws IOException` Reads a byte of data from the file and returns it as an integer in the range 0 to 255.
`public int read(byte[] a) throws IOException` Reads `a.length` bytes of data from the file into the array of bytes `a`. Returns the number of bytes read or -1 if the end of the file is encountered.
`public final byte readByte() throws IOException` Reads a `byte` value from the file and returns that value. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.
`public final short readShort() throws IOException` Reads a `short` value from the file and returns that value. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.
`public final int readInt() throws IOException` Reads an `int` value from the file and returns that value. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.
`public final long readLong() throws IOException` Reads a `long` value from the file and returns that value. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.
`public final double readDouble() throws IOException` Reads a `double` value from the file and returns that value. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.
`public final float readFloat() throws IOException` Reads a `float` value from the file and returns that value. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.
`public final char readChar() throws IOException` Reads a `char` value from the file and returns that value. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.
`public final boolean readBoolean() throws IOException` Reads a `boolean` value from the file and returns that value. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.
`public final String readUTF() throws IOException` Reads a `String` value from the file and returns that value. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.

Display 10.18 Random Access to a File *(Part 1 of 2)*

```
1  import java.io.RandomAccessFile;
2  import java.io.IOException;
3  import java.io.FileNotFoundException;

4  public class RandomAccessDemo
5  {
6    public static void main(String[] args)
7    {
8       try
9       {
10          RandomAccessFile ioStream =
11            new RandomAccessFile("bytedata", "rw");

12          System.out.println("Writing 3 bytes to the file bytedata.");
13          ioStream.writeByte(1);
14          ioStream.writeByte(2);
15          ioStream.writeByte(3);
16          System.out.println("The length of the file is now = "
17                             + ioStream.length());
18          System.out.println("The file pointer location is "
19                             + ioStream.getFilePointer());

20          System.out.println("Moving the file pointer to location 1.");
21          ioStream.seek(1);
22          byte oneByte = ioStream.readByte();
23          System.out.println("The value at location 1 is " + oneByte);
24          oneByte = ioStream.readByte();
25          System.out.println("The value at the next location is "
26                             + oneByte);

27          System.out.println("Now we move the file pointer back to");
28          System.out.println("location 1, and change the byte.");
29          ioStream.seek(1);
30          ioStream.writeByte(9);
31          ioStream.seek(1);
32          oneByte = ioStream.readByte();
33          System.out.println("The value at location 1 is now " + oneByte);

34          System.out.println("Now we go to the end of the file");
35          System.out.println("and write a double.");
36          ioStream.seek(ioStream.length());
37          ioStream.writeDouble(41.99);
38          System.out.println("The length of the file is now = "
39                             + ioStream.length());
```

Display 10.18 **Random Access to a File** ***(Part 2 of 2)***

```
40            System.out.println("Returning to location 3,");
41            System.out.println("where we wrote the double.");
42            ioStream.seek(3);
43            double oneDouble = ioStream.readDouble();
44            System.out.println("The double value at location 3 is "
45                                       + oneDouble);
46            ioStream.close();
47        }
48        catch(FileNotFoundException e)
49        {
50            System.out.println("Problem opening file.");
51        }
52        catch(IOException e)
53        {
54            System.out.println("Problems with file I/O.");
55        }
56        System.out.println("End of program.");
57    }
58  }
```

The location of `readDouble` must be a location where `writeDouble` wrote to the file.

The dialog assumes the file `bytedata` did not exist before the program was run.

SAMPLE DIALOGUE

```
Writing 3 bytes to the file bytedata.
The length of the file is now = 3
The file pointer location is 3
Moving the file pointer to location 1.
The value at location 1 is 2
The value at the next location is 3
Now we move the file pointer back to
location 1, and change the byte.
The value at location 1 is now 9
Now we go to the end of the file
and write a double.
The length of the file is now = 11
Returning to location 3,
where we wrote the double.
The double value at location 3 is 41.99
End of program.
```

Byte locations are numbered starting with zero.

Three 1-byte values and one `double` value that uses 8 bytes = 11 bytes total.

opening a file

The constructor for `RandomAccessFile` takes either a string name for the file or an object of the class `File` as its first argument. The second argument must be one of the four strings `"rw"`, `"r"`, and two modes we will not discuss, `"rws"` and `"rwd"`. The string `"rw"` means your code can both read and write to the file after it is open. The string `"r"` means your code can read from the file but cannot write to the file.

If the file already exists, then when it is opened, the length is not reset to `0`, but the file pointer will be positioned at the start of the file, which is what you would expect at least for `"r"`. If the length of the file is not what you want, you can change it with the method `setLength`. In particular, you can use `setLength` to empty the file.

Pitfall

A RandomAccessFile Need Not Start Empty

If a file already exists, then when it is opened with `RandomAccessFile`, the length is not reset to `0`, but the file pointer will be positioned at the start of the file. So, old data in the file is not lost and the file pointer is set for the most likely position for reading, not the most likely position for writing.

Self-Test Exercises

42. If you run the program in Display 10.18 a second time, will the output be the same?
43. How can you modify the program in Display 10.18 so the file always starts out empty?

Chapter Summary

- Files that are considered to be strings of characters and that look like characters to your program and your editor are called text files. Files whose contents must be handled as strings of binary digits are called binary files.
- You can use the class `PrintWriter` to write to a text file and can use the class `BufferedReader` to read from a text file.
- The class `File` can be used to check whether there is a file with a given name. It can also check whether your program is allowed to read the file and/or allowed to 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 to a binary file. The objects can be read back with the method `readObject` of the class `ObjectInputStream`.
- To use the method `writeObject` of the class `ObjectOutputStream` or the method `readObject` of the class `ObjectInputStream`, the class whose objects are written to a file must implement the `Serializable` interface.
- The way that 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 use the class `RandomAccessFile` to create a stream that gives random access to the bytes in a file.

Answers to Self-Test Exercises

1. With an input stream, data flows from a file or input device to your program. With an output stream, data flows from your program to a file or output device.
2. A binary file contains data that is processed as binary data. A text file allows your program and editor to view the file as if it contained a sequence of characters. A text file can be viewed with an editor, whereas a binary file cannot.
3. A `FileNotFoundException` would be thrown if the file cannot be opened because, for example, there is already a directory (folder) named `stuff.txt`. Note that if the file does not exist but can be created, then 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`.
4. No. That is why we use an object of the class `FileOutputStream` as an argument. The correct way to express the code displayed in the question is as follows:

```
PrintWriter outputStream =
     new PrintWriter(new FileOutputStream("stuff.txt"));
```

5.
```
PrintWriter outputStream =
     new PrintWriter(new FileOutputStream("sam");
```

6.
```
PrintWriter outStream =
     new PrintWriter(new FileOutputStream("sam", true));
```

7. Yes, it will send suitable output to the text file because the class `Person` has a well-defined `toString()` method.
8.
```
BufferedReader fileIn =
              new BufferedReader(new FileReader("joe"));
```

9. The method `readLine` returns a value of type `String`. The method `read` reads a single *character*, but it returns it as a value of type `int`. To get the value to be of type `char`, you need to do a type cast.
10. Both `read` and `readLine` in the class `BufferedReader` might throw an `IOException`.
11. The `try` block in Display 10.3 is larger so that it includes the invocations of the method `readLine`, which might throw an `IOException`. The method `println` in Display 10.1 does not throw any exceptions that must be caught.
12. Yes
13. No
14. In the following code from exercise 12:

```
BufferedReader inputStream =
     new BufferedReader(new FileReader("morestuff.txt"));
```

The possible exception is thrown by the `FileReader` constructor not by the `BufferedReader` constructor. The code in exercise 13 has no invocation of a `FileReader` constructor.

15. No, you must read the number as a string and then convert the string to a number with `Integer.parseInt` (or in some other way).
16. When the method `readLine` tries to read beyond the end of a file, it returns the value `null`. Thus, you can test for the end of a file by testing for `null`.
17. The method `read` reads a single *character*, but it returns it as a value of type `int`. To get the value to be of type `char`, you need to do a type cast.
18. When the method `read` tries to read beyond the end of a file, it returns the value –1. Thus, you can test for the end of a file by testing for the value –1. This works because all "real" characters return a positive `int` value.
19. Yes, if `original.txt` is an empty file, then the file `numbered.txt` produced by the program will also be empty.
20. Only the classes `FileReader` and `FileOutputStream` have a constructor that accepts a file name as an argument.
21. Yes, it is legal.
22. Replace

    ```
    System.setErr(errStream);
    ```

    with

    ```
    System.setOut(errStream);
    ```

23. Add

    ```
    System.setOut(errStream);
    ```

    to get

    ```
    System.setErr(errStream);
    System.setOut(errStream);
    ```

24. ```
    import java.io.File;

    public class FileExercise
    {
        public static void main(String[] args)
        {
            File fileObject = new File("Sally.txt");
                if (fileObject.exists())
                System.out.println(
                            "There is a file named Sally.txt.");
    ```
    ```

```
        else
            System.out.println(
                        "There is no file named Sally.txt.");
    }
}
```

25.
```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
import java.io.File;

public class FileExercise2
{
    public static void main(String[] args)
    {
        BufferedReader keyboard = new BufferedReader(
                         new InputStreamReader(System.in));
        String fileName = null;
        File fileObject = null;

        try
        {
            System.out.print("Enter a file name and I will");
            System.out.println("tell you if it exists.");
            fileName = keyboard.readLine();
            fileName = fileName.trim();
            fileObject = new File(fileName);


            if (fileObject.exists())
            {
                System.out.println("There is a file named  "
                                              + fileName);
                System.out.println("Delete the file? (y/n)");
                char answer = (char)System.in.read();
                //Alternatively you can use
                //ConsoleIn.readLineNonwhiteChar
                //which does not need try-catch.

                if ((answer == 'y') || (answer == 'Y'))
                {
                    if (fileObject.delete())
                        System.out.println("File deleted.");
                    else
                        System.out.println(
                                    "Cannot delete file.");
                }
            }
```

```
            else
                System.out.println(
                                "No file named " + fileName);
        }
        catch(IOException e)
        {
            System.out.println(
                            "Error reading from keyboard.");
        }
    }
}
```

26.
```
ObjectOutputStream outputThisWay =
                    new ObjectOutputStream(
                       new FileOutputStream("bindata.dat"));
```

27.
```
 outputThisWay.writeDouble(v1);
outputThisWay.writeDouble(v2);
```

28.
```
outputThisWay.writeUTF("Hello");
```

29.
```
outputThisWay.close();
```

30.
```
ObjectInputStream inputThing =
                  new ObjectInputStream(
                       new FileInputStream("someStuff"));
```

31.
```
number = inputThing.readDouble();
```

32.
```
inputThing.close();
```

33. If a number is written to a file with `writeInt`, it should be read only with `readInt`. If you use `readLong` or `readDouble` to read the number, something will go wrong.

34. 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`.

35. When opening a binary file for either output or input in the ways discussed in this chapter, a `FileNotFoundException` might be thrown and other `IOException`s may be thrown.

36. An `EOFException` is thrown when your program tries to read the (nonexisting) fourth number.

37. It is not an infinite loop because when the end of the file is reached, an exception will be thrown, and that will end the entire `try` block.

38. You add the two words `implements Serializable` to the beginning of the class definition. You also must do this for the classes for the instance variables and so on for all levels of class instance variables within classes.

39. `import java.io.Serializable;` or `import java.io.*;`

40. The return type is `Object`, which means the returned value usually needs to be type cast.
41. Yes. That is why it is a legitimate argument for `writeObject`.
42. No. Each time the program is run, the file will get longer.
43. Add the following near the start of `main`:

```
ioStream.setLength(0);
```

Programming Projects

Projects Involving Only Text Files

1. Write a program that will search a text file of strings representing numbers of type `int` and will write the largest and the smallest numbers to the screen. The file contains nothing but strings representing numbers of type `int`, one per line.
2. Write a program that takes its input from a text file of strings representing numbers of type `double` and outputs the average of the numbers in the file to the screen. The file contains nothing but strings representing numbers of type `double`, one per line.
3. Write a program that takes its input from a text file of strings representing numbers of type `double`. The program outputs to the screen the average and standard deviation of the numbers in the file. The file contains nothing but strings representing numbers of type `double`, one per line. The standard deviation of a list of numbers n_1, n_2, n_3, and so forth is defined as the square root of the average of the following numbers:

   $(n_1 - a)^2$, $(n_2 - a)^2$, $(n_3 - a)^2$, and so forth.

   The number a is the average of the numbers n_1, n_2, n_3, and so forth. Hint: Write your program so that it first reads the entire file and computes the average of all the numbers, then closes the file, then reopens the file and computes the standard deviation. You will find it helpful to first do Programming Project 2 and then modify that program to obtain the program for this project.
4. Write a program to edit text files for extra blanks. The program will replace any string of two or more blanks with a single blank. Your program should work as follows: Create a temporary file. Copy from the file to the temporary file but do not copy extra blanks. Copy the contents of the temporary file back into the original file. Use a method (or methods) in the class `File` to remove the temporary file. You will also want to use the class `File` for other things in your program. The temporary file should have a name different from all existing files so that the existing files are not affected (except for the file being edited). Your program will ask the user for the name of the file to be edited. However, it will not ask the user for the name of the temporary file but instead will generate the name within the program. You can generate the name any way that is clear and efficient. One possible way to

generate the temporary file is to start with an unlikely name, such as "TempX", and to append a character, such as 'X', until a name is found that does not name an existing file.

5. Write a program that gives and takes advice on program writing. The program starts by writing a piece of advice to the screen and asking the user to type in a different piece of advice. The program then ends. The next person to run the program receives the advice given by the person who last ran the program. The advice is kept in a text file and the content of the file changes after each run of the program. You can use your editor to enter the initial piece of advice in the file so that the first person who runs the program receives some advice. Allow the user to type in advice of any length so that it can be any number of lines long. The user is told to end his or her advice by pressing the Return key two times. Your program can then test to see that it has reached the end of the input by checking to see when it reads two consecutive occurrences of the character '\n'. Alternatively, your program can simply test for an empty line marking the end of the file.

Projects Involving Binary Files

6. Write a program that will search a binary file of numbers of type int and write the largest and the smallest numbers to the screen. The file contains nothing but numbers of type int written to the file with writeInt.

7. Write a program that takes its input from a binary file of numbers of type double and outputs the average of the numbers in the file to the screen. The file contains nothing but numbers of type double written to the file with writeDouble.

8. Write a program that takes its input from a binary file of numbers of type double. The file contains nothing but numbers of type double written to the file with writeDouble. The program outputs to the screen the average and standard deviation of the numbers in the file. The standard deviation of a list of numbers n_1, n_2, n_3, and so forth is defined as the square root of the average of the following numbers:

   $(n_1 - a)^2$, $(n_2 - a)^2$, $(n_3 - a)^2$, and so forth.

   The number a is the average of the numbers n_1, n_2, n_3, and so forth. Hint: Write your program so that it first reads the entire file and computes the average of all the numbers, then closes the file, then reopens the file and computes the standard deviation. You will find it helpful to first do Programming Project 7 and then modify that program to obtain the program for this project.

9. Change the definition of the class Person in Display 5.11 to be serializable. Note that this requires that you also change the class Date. Then, write a program to maintain a binary file of records of people (records of type Person). Allow commands to delete a record specified by the person's name, to add a record, to retrieve and display a record, and to obtain all records of people within a specified age range. To obtain the age of a person, you need the current date. Your program will ask the user for the current date when the program begins. You can do this with random access files, but for this exercise do not use random access files. Use a file or files that record records with the method writeObject of the class ObjectOutputStream.