CHAPTER

17

BINARY I/O

Objectives

- To discover how I/O is processed in Java (§17.2).
- To distinguish between text I/O and binary I/O (§17.3).
- To read and write bytes using `FileInputStream` and `FileOutputStream` (§17.4.1).
- To filter data using the base classes `FilterInputStream` and `FilterOutputStream` (§17.4.2).
- To read and write primitive values and strings using `DataInputStream` and `DataOutputStream` (§17.4.3).
- To improve I/O performance by using `BufferedInputStream` and `BufferedOutputStream` (§17.4.4).
- To write a program that copies a file (§17.5).
- To store and restore objects using `ObjectOutputStream` and `ObjectInputStream` (§17.6).
- To implement the `Serializable` interface to make objects serializable (§17.6.1).
- To serialize arrays (§17.6.2).
- To read and write files using the `RandomAccessFile` class (§17.7).

17.1 Introduction

Key Point

Java provides many classes for performing text I/O and binary I/O.

text file
binary file

Files can be classified as either text or binary. A file that can be processed (read, created, or modified) using a text editor such as Notepad on Windows or vi on UNIX is called a *text file*. All other files are called *binary files*. You cannot read binary files using a text editor—they are designed to be read by programs. For example, Java source programs are text files and can be read by a text editor, but Java class files are binary files and are read by the JVM.

why binary I/O?

Although it is not technically precise and correct, you can envision a text file as consisting of a sequence of characters, and a binary file as consisting of a sequence of bits. Characters in a text file are encoded using a character-encoding scheme such as ASCII or Unicode. For example, the decimal integer `199` is stored as a sequence of three characters `199` in a text file, and the same integer is stored as a byte-type value `C7` in a binary file, because decimal `199` equals hex `C7` ($199 = 12 \times 16^1 + 7$). The advantage of binary files is that they are more efficient to process than text files.

text I/O
binary I/O

Java offers many classes for performing file input and output. These can be categorized as *text I/O classes* and *binary I/O classes*. In Section 12.11, File Input and Output, you learned how to read and write strings and numeric values from/to a text file using `Scanner` and `PrintWriter`. This chapter introduces the classes for performing binary I/O.

17.2 How Is Text I/O Handled in Java?

Key Point

Text data are read using the `Scanner` class and written using the `PrintWriter` class.

Recall that a `File` object encapsulates the properties of a file or a path but does not contain the methods for reading/writing data from/to a file. In order to perform I/O, you need to create objects using appropriate Java I/O classes. The objects contain the methods for reading/writing data from/to a file. For example, to write text to a file named **temp.txt**, you can create an object using the `PrintWriter` class as follows:

```
PrintWriter output = new PrintWriter("temp.txt");
```

You can now invoke the `print` method on the object to write a string to the file. For example, the following statement writes `Java 101` to the file:

```
output.print("Java 101");
```

The following statement closes the file:

```
output.close();
```

There are many I/O classes for various purposes. In general, these can be classified as input classes and output classes. An *input class* contains the methods to read data, and an *output class* contains the methods to write data. `PrintWriter` is an example of an output class, and `Scanner` is an example of an input class. The following code creates an input object for the file **temp.txt** and reads data from the file:

```
Scanner input = new Scanner(new File("temp.txt"));
System.out.println(input.nextLine());
```

If **temp.txt** contains the text `Java 101`, `input.nextLine()` returns the string `"Java 101"`.

stream
input stream
output stream

Figure 17.1 illustrates Java I/O programming. An input object reads a *stream* of data from a file, and an output object writes a stream of data to a file. An input object is also called an *input stream* and an output object an *output stream*.

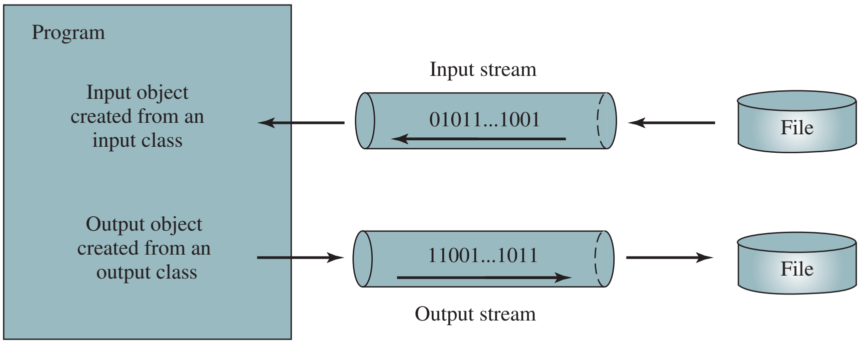


FIGURE 17.1 The program receives data through an input object and sends data through an output object.

Check Point

17.2.1 What is a text file and what is a binary file? Can you view a text file or a binary file using a text editor?

17.2.2 How do you read or write text data in Java? What is a stream?

17.3 Text I/O vs. Binary I/O

Key Point

Binary I/O does not involve encoding or decoding and thus is more efficient than text I/O.

Computers do not differentiate between binary files and text files. All files are stored in binary format, and thus all files are essentially binary files. Text I/O is built upon binary I/O to provide a level of abstraction for character encoding and decoding, as shown in Figure 17.2a. Encoding and decoding are automatically performed for text I/O. The JVM converts Unicode to a file-specific encoding when writing a character, and converts a file-specific encoding to Unicode when reading a character. For example, suppose you write the string `"199"` using text I/O to a file, each character is written to the file. Since the Unicode for character `1` is `0x0031`, the Unicode `0x0031` is converted to a code that depends on the encoding scheme for the file. (Note the prefix `0x` denotes a hex number.) In the United States, the default encoding for text files on Windows is ASCII. The ASCII code for character `1` is `49` (`0x31` in hex) and for

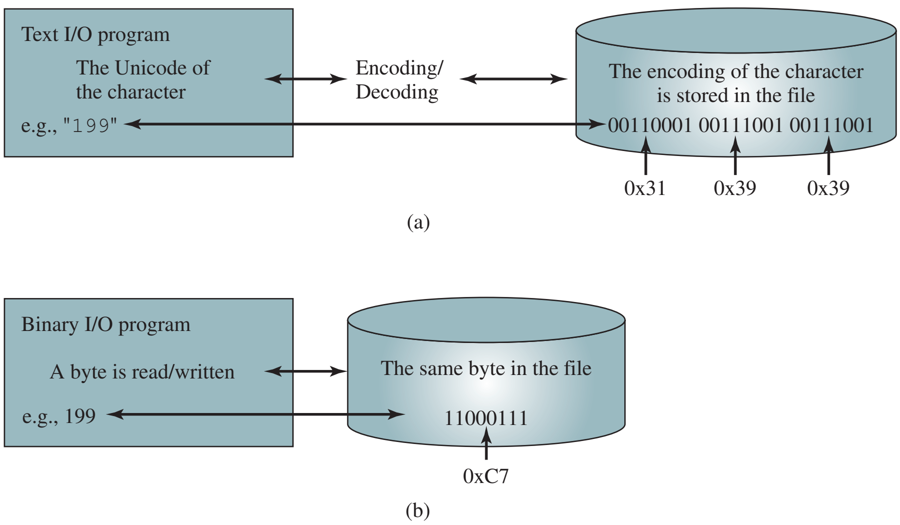


FIGURE 17.2 Text I/O requires encoding and decoding, whereas binary I/O does not.

character **9** is **57** (**0x39** in hex). Thus, to write the characters **199**, three bytes—**0x31**, **0x39**, and **0x39**—are sent to the output, as shown in Figure 17.2a.

Binary I/O does not require conversions. If you write a numeric value to a file using binary I/O, the exact value in the memory is copied into the file. For example, a byte-type value **199** is represented as **0xC7** ($199 = 12 \times 16^1 + 7$) in the memory and appears exactly as **0xC7** in the file, as shown in Figure 17.2b. When you read a byte using binary I/O, one byte value is read from the input.

In general, you should use text input to read a file created by a text editor or a text output program, and use binary input to read a file created by a Java binary output program.

Binary I/O is more efficient than text I/O because binary I/O does not require encoding and decoding. Binary files are independent of the encoding scheme on the host machine and thus are portable. Java programs on any machine can read a binary file created by a Java program. This is why Java class files are binary files. Java class files can run on a JVM on any machine.

.txt and .dat

Note

For consistency, this book uses the extension **.txt** to name text files and **.dat** to name binary files.

Check Point

17.3.1 What are the differences between text I/O and binary I/O?

17.3.2 How is a Java character represented in the memory, and how is a character represented in a text file?

17.3.3 If you write the string **"ABC"** to an ASCII text file, what values are stored in the file?

17.3.4 If you write the string **"100"** to an ASCII text file, what values are stored in the file? If you write a numeric byte-type value **100** using binary I/O, what values are stored in the file?

17.3.5 What is the encoding scheme for representing a character in a Java program? By default, what is the encoding scheme for a text file on Windows?

17.4 Binary I/O Classes

Key Point

*The abstract **InputStream** is the root class for reading binary data, and the abstract **OutputStream** is the root class for writing binary data.*

The design of the Java I/O classes is a good example of applying inheritance, where common operations are generalized in superclasses, and subclasses provide specialized operations. Figure 17.3 lists some of the classes for performing binary I/O. **InputStream** is the root for

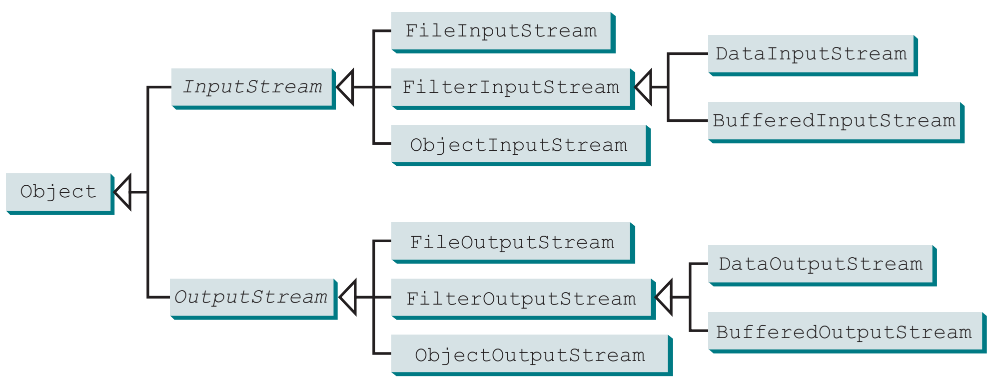


Figure 17.3 **InputStream**, **OutputStream**, and their subclasses are for performing binary I/O.

java.io.InputStream	
+read(): int	Reads the next byte of data from the input stream. The value byte is returned as an `int` value in the range `0–255`. If no byte is available because the end of the stream has been reached, the value –1 is returned.
+read(b: byte[]): int	Reads up to `b.length` bytes into array `b` from the input stream and returns the actual number of bytes read. Returns –1 at the end of the stream.
+read(b: byte[], off: int, len: int): int	Reads bytes from the input stream and stores them in `b[off], b[off+1], ..., b[off+len-1]`. The actual number of bytes read is returned. Returns `-1` at the end of the stream.
+close(): void	Closes this input stream and releases any system resources occupied by it.
+skip(n: long): long	Skips over and discards `n` bytes of data from this input stream. The actual number of bytes skipped is returned.

FIGURE 17.4 The abstract **`InputStream`** class defines the methods for the input stream of bytes.

binary input classes, and **`OutputStream`** is the root for binary output classes. Figures 17.4 and 17.5 list all the methods in the classes **`InputStream`** and **`OutputStream`**.

Note

All the methods in the binary I/O classes are declared to throw **`java.io.IOException`** or a subclass of **`java.io.IOException`**. throws IOException

java.io.OutputStream	
+write(int b): void	Writes the specified byte to this output stream. The parameter `b` is an `int` value. `(byte)b` is written to the output stream.
+write(b: byte[]): void	Writes all the bytes in array `b` to the output stream.
+write(b: byte[], off: int, len: int): void	Writes `b[off], b[off+1], ..., b[off+len-1]` into the output stream.
+close(): void	Closes this output stream and releases any system resources occupied by it.
+flush(): void	Flushes this output stream and forces any buffered output bytes to be written out.

FIGURE 17.5 The abstract **`OutputStream`** class defines the methods for the output stream of bytes.

17.4.1 FileInputStream/FileOutputStream

`FileInputStream`/**`FileOutputStream`** are for reading/writing bytes from/to files. All the methods in these classes are inherited from **`InputStream`** and **`OutputStream`**. **`FileInputStream`**/**`FileOutputStream`** do not introduce new methods. To construct a **`FileInputStream`**, use the constructors shown in Figure 17.6.

A **`java.io.FileNotFoundException`** will occur if you attempt to create a **`FileInputStream`** with a nonexistent file. FileNotFoundException

To construct a **`FileOutputStream`**, use the constructors shown in Figure 17.7.

If the file does not exist, a new file will be created. If the file already exists, the first two constructors will delete the current content of the file. To retain the current content and append new data into the file, use the last two constructors and pass **`true`** to the **`append`** parameter.

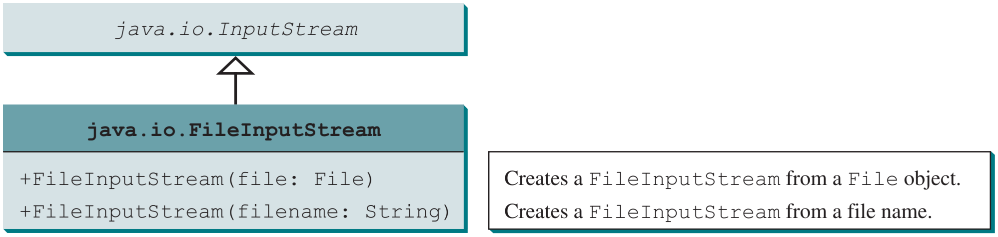


FIGURE 17.6 `FileInputStream` inputs a stream of bytes from a file.

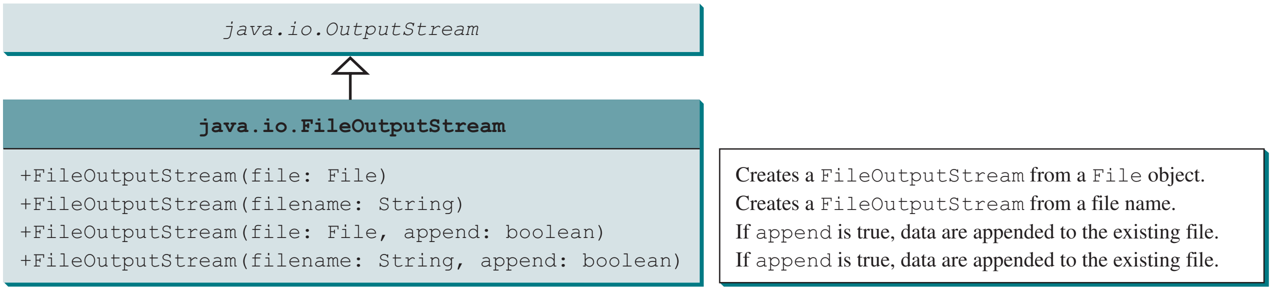


FIGURE 17.7 `FileOutputStream` outputs a stream of bytes to a file.

IOException

Almost all the methods in the I/O classes throw `java.io.IOException`. Therefore, you have to declare to throw `java.io.IOException` in the method in (a) or place the code in a try-catch block in (b), as shown below:

(a)
Declaring exception in the method

```
public static void main(String[] args)
    throws IOException {
  // Perform I/O operations
}
```

(b)
Using try-catch block

```
public static void main(String[] args) {
  try {
    // Perform I/O operations
  }
  catch (IOException ex) {
    ex.printStackTrace();
  }
}
```

Listing 17.1 uses binary I/O to write 10 byte values from `1` to `10` to a file named **temp.dat** and reads them back from the file.

LISTING 17.1 TestFileStream.java

import

output stream

output

```
import java.io.*;

public class TestFileStream {
  public static void main(String[] args) throws IOException {
    try (
      // Create an output stream to the file
      FileOutputStream output = new FileOutputStream("temp.dat");
    ) {
      // Output values to the file
      for (int i = 1; i <= 10; i++)
        output.write(i);
    }

    try (
```

```
15          // Create an input stream for the file
16          FileInputStream input = new FileInputStream("temp.dat");
17        ) {
18          // Read values from the file
19          int value;
20          while ((value = input.read()) != -1)
21            System.out.print(value + " ");
22        }
23      }
24    }
```

input stream

input

```
1 2 3 4 5 6 7 8 9 10
```

The program uses the try-with-resources to declare and create input and output streams so they will be automatically closed after they are used. The `java.io.InputStream` and `java.io.OutputStream` classes implement the `AutoClosable` interface. The `AutoClosable` interface defines the `close()` method that closes a resource. Any object of the `AutoClosable` type can be used with the try-with-resources syntax for automatic closing.

AutoClosable

A `FileOutputStream` is created for the file **temp.dat** in line 7. The `for` loop writes 10 byte values into the file (lines 10 and 11). Invoking `write(i)` is the same as invoking `write((byte)i)`. Line 16 creates a `FileInputStream` for the file **temp.dat**. Values are read from the file and displayed on the console in lines 19–21. The expression `((value = input.read()) != -1)` (line 20) reads a byte from `input.read()`, assigns it to `value`, and checks whether it is `-1`. The input value of `-1` signifies the end of a file.

end of a file

The file **temp.dat** created in this example is a binary file. It can be read from a Java program but not from a text editor, as shown in Figure 17.8.

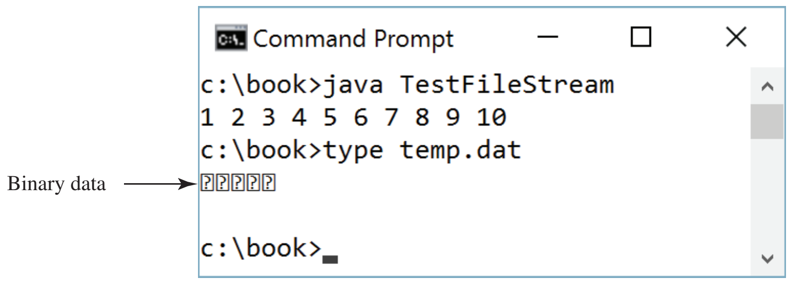


Figure 17.8 A binary file cannot be displayed in text mode. *Source*: Copyright

Tip

When a stream is no longer needed, always close it using the `close()` method or automatically close it using a try-with-resource statement. Not closing streams may cause data corruption in the output file or other programming errors.

close stream

Note

The root directory for the file is the classpath directory. For the example in this book, the root directory is **c:\book**, so the file **temp.dat** is located at **c:\book**. If you wish to place **temp.dat** in a specific directory, replace line 6 with

where is the file?

```
FileOutputStream output =
  new FileOutputStream ("directory/temp.dat");
```

Note

An instance of `FileInputStream` can be used as an argument to construct a `Scanner`, and an instance of `FileOutputStream` can be used as an argument to construct a `PrintWriter`. You can create a `PrintWriter` to append text into a file using

appending to text file

```
new PrintWriter(new FileOutputStream("temp.txt", true));
```

If **temp.txt** does not exist, it is created. If **temp.txt** already exists, new data are appended to the file. See Programming Exercise 17.1.

17.4.2 FilterInputStream/FilterOutputStream

Filter streams are streams that filter bytes for some purpose. The basic byte input stream provides a **read** method that can be used only for reading bytes. If you want to read integers, doubles, or strings, you need a filter class to wrap the byte input stream. Using a filter class enables you to read integers, doubles, and strings instead of bytes and characters. **FilterInputStream** and **FilterOutputStream** are the base classes for filtering data. When you need to process primitive numeric types, use **DataInputStream** and **DataOutputStream** to filter bytes.

17.4.3 DataInputStream/DataOutputStream

DataInputStream reads bytes from the stream and converts them into appropriate primitive-type values or strings. **DataOutputStream** converts primitive-type values or strings into bytes and outputs the bytes to the stream.

DataInputStream extends **FilterInputStream** and implements the **DataInput** interface, as shown in Figure 17.9. **DataOutputStream** extends **FilterOutputStream** and implements the **DataOutput** interface, as shown in Figure 17.10.

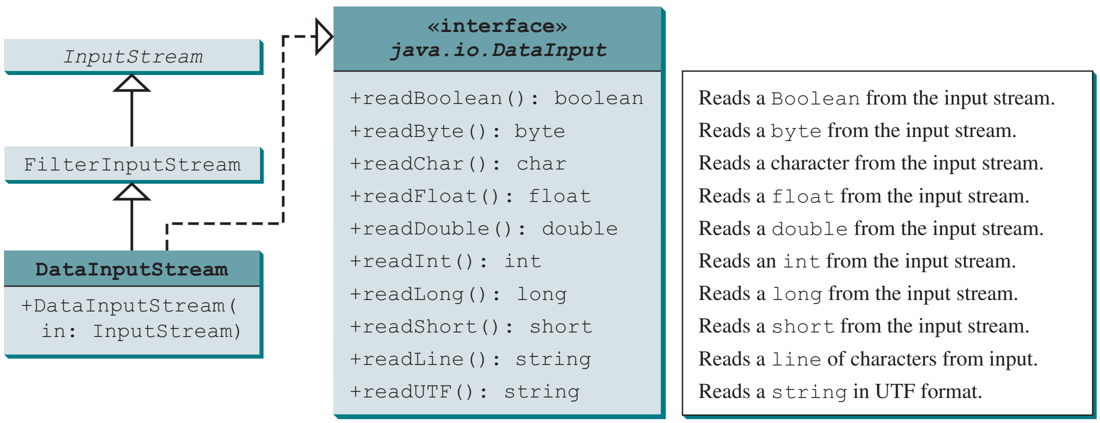


FIGURE 17.9 **DataInputStream** filters an input stream of bytes into primitive data-type values and strings.

DataInputStream implements the methods defined in the **DataInput** interface to read primitive data-type values and strings. **DataOutputStream** implements the methods defined in the **DataOutput** interface to write primitive data-type values and strings. Primitive values are copied from memory to the output without any conversions. Characters in a string may be written in several ways, as discussed in the next section.

Characters and Strings in Binary I/O

A Unicode character consists of two bytes. The **writeChar(char c)** method writes the Unicode of character **c** to the output. The **writeChars(String s)** method writes the Unicode for each character in the string **s** to the output. The **writeBytes(String s)** method writes the lower byte of the Unicode for each character in the string **s** to the output. The high byte of the Unicode is discarded. The **writeBytes** method is suitable for strings that consist of

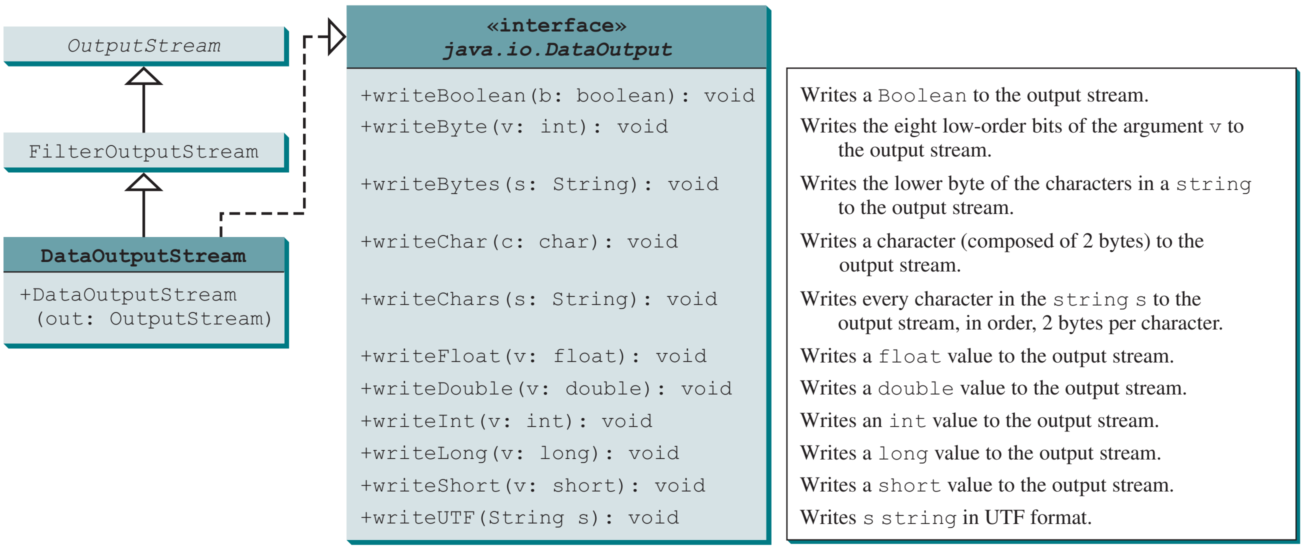


Figure 17.10 `DataOutputStream` enables you to write primitive data-type values and strings into an output stream.

ASCII characters, since an ASCII code is stored only in the lower byte of a Unicode. If a string consists of non-ASCII characters, you have to use the `writeChars` method to write the string.

The `writeUTF(String s)` method writes a string using the UTF coding scheme. UTF is efficient for compressing a string with Unicode characters. For more information on UTF, see Supplement III.Z, UTF in Java. The `readUTF()` method reads a string that has been written using the `writeUTF` method.

Creating **DataInputStream/DataOutputStream**

`DataInputStream`/`DataOutputStream` are created using the following constructors (see Figures 17.9 and 17.10):

```
public DataInputStream(InputStream instream)
public DataOutputStream(OutputStream outstream)
```

The following statements create data streams. The first statement creates an input stream for the file **in.dat**; the second statement creates an output stream for the file **out.dat**.

```
DataInputStream input =
  new DataInputStream(new FileInputStream("in.dat"));
DataOutputStream output =
  new DataOutputStream(new FileOutputStream("out.dat"));
```

Listing 17.2 writes student names and scores to a file named **temp.dat** and reads the data back from the file.

Listing 17.2 TestDataStream.java

```
import java.io.*;

public class TestDataStream {
  public static void main(String[] args) throws IOException {
    try ( // Create an output stream for file temp.dat
      DataOutputStream output =
        new DataOutputStream(new FileOutputStream("temp.dat"));
    ) {
```

output stream

output

input stream

input

```
9          // Write student test scores to the file
10         output.writeUTF("John");
11         output.writeDouble(85.5);
12         output.writeUTF("Susan");
13         output.writeDouble(185.5);
14         output.writeUTF("Kim");
15         output.writeDouble(105.25);
16       }
17
18       try ( // Create an input stream for file temp.dat
19         DataInputStream input =
20           new DataInputStream(new FileInputStream("temp.dat"));
21       ) {
22         // Read student test scores from the file
23         System.out.println(input.readUTF() + " " + input.readDouble());
24         System.out.println(input.readUTF() + " " + input.readDouble());
25         System.out.println(input.readUTF() + " " + input.readDouble());
26       }
27     }
28   }
```

```
John 85.5
Susan 185.5
Kim 105.25
```

A **DataOutputStream** is created for file **temp.dat** in lines 6 and 7. Student names and scores are written to the file in lines 10–15. A **DataInputStream** is created for the same file in lines 19 and 20. Student names and scores are read back from the file and displayed on the console in lines 23–25.

DataInputStream and **DataOutputStream** read and write Java primitive-type values and strings in a machine-independent fashion, thereby enabling you to write a data file on one machine and read it on another machine that has a different operating system or file structure. An application uses a data output stream to write data that can later be read by a program using a data input stream.

DataInputStream filters data from an input stream into appropriate primitive-type values or strings. **DataOutputStream** converts primitive-type values or strings into bytes and outputs the bytes to an output stream. You can view **DataInputStream**/**FileInputStream** and **DataOutputStream**/**FileOutputStream** working in a pipe line as shown in Figure 17.11.

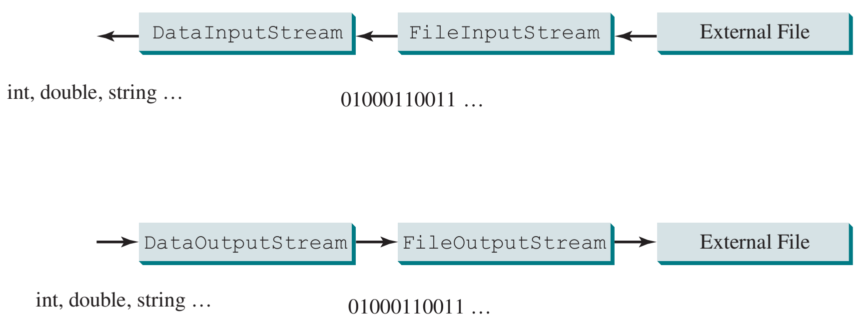


FIGURE 17.11 **DataInputStream** filters an input stream of byte to data and **DataOutputStream** converts data into a stream of bytes.

Caution

You have to read data in the same order and format in which they are stored. For example, since names are written in UTF using `writeUTF`, you must read names using `readUTF`.

Detecting the End of a File

If you keep reading data at the end of an `InputStream`, an `EOFException` will occur. This exception can be used to detect the end of a file, as shown in Listing 17.3.

`EOFException`

Listing 17.3 DetectEndOfFile.java

```
1  import java.io.*;
2
3  public class DetectEndOfFile {
4    public static void main(String[] args) {
5      try {
6        try (DataOutputStream output =
7          new DataOutputStream(new FileOutputStream("test.dat"))) {
8          output.writeDouble(4.5);
9          output.writeDouble(43.25);
10         output.writeDouble(3.2);
11       }
12
13       try (DataInputStream input =
14         new DataInputStream(new FileInputStream("test.dat"))) {
15         while (true)
16           System.out.println(input.readDouble());
17       }
18     }
19     catch (EOFException ex) {
20       System.out.println("All data were read");
21     }
22     catch (IOException ex) {
23       ex.printStackTrace();
24     }
25   }
26 }
```

output stream (line 6)
output (line 8)
input stream (line 13)
input (line 16)
`EOFException` (line 19)

```
4.5
43.25
3.2
All data were read
```

The program writes three double values to the file using `DataOutputStream` (lines 6–11) and reads the data using `DataInputStream` (lines 13–17). When reading past the end of the file, an `EOFException` is thrown. The exception is caught in line 19.

17.4.4 BufferedInputStream/BufferedOutputStream

`BufferedInputStream`/`BufferedOutputStream` can be used to speed up input and output by reducing the number of disk reads and writes. Using `BufferedInputStream`, the whole block of data on the disk is read into the buffer in the memory once. The individual data are then loaded to your program from the buffer, as shown in Figure 17.12a. Using `BufferedOutputStream`, the individual data are first written to the buffer in the memory. When the buffer is full, all data in the buffer are written to the disk once, as shown in Figure 17.12b.

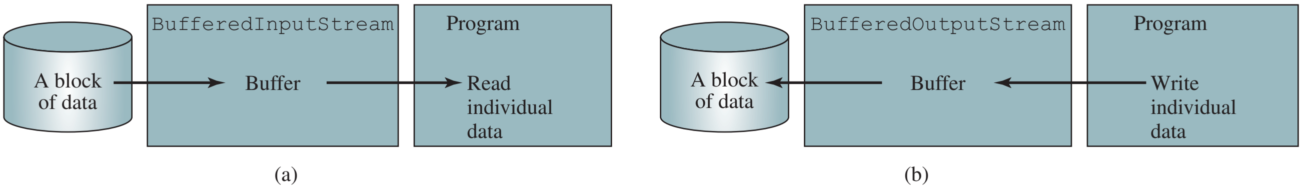


FIGURE 17.12 Buffer I/O places data in a buffer for fast processing.

BufferedInputStream/**BufferedOutputStream** does not contain new methods. All the methods in **BufferedInputStream**/**BufferedOutputStream** are inherited from the **InputStream**/**OutputStream** classes. **BufferedInputStream**/**BufferedOutputStream** manages a buffer behind the scene and automatically reads/writes data from/to disk on demand.

You can wrap a **BufferedInputStream**/**BufferedOutputStream** on any **InputStream**/**OutputStream** using the constructors shown in Figures 17.13 and 17.14.

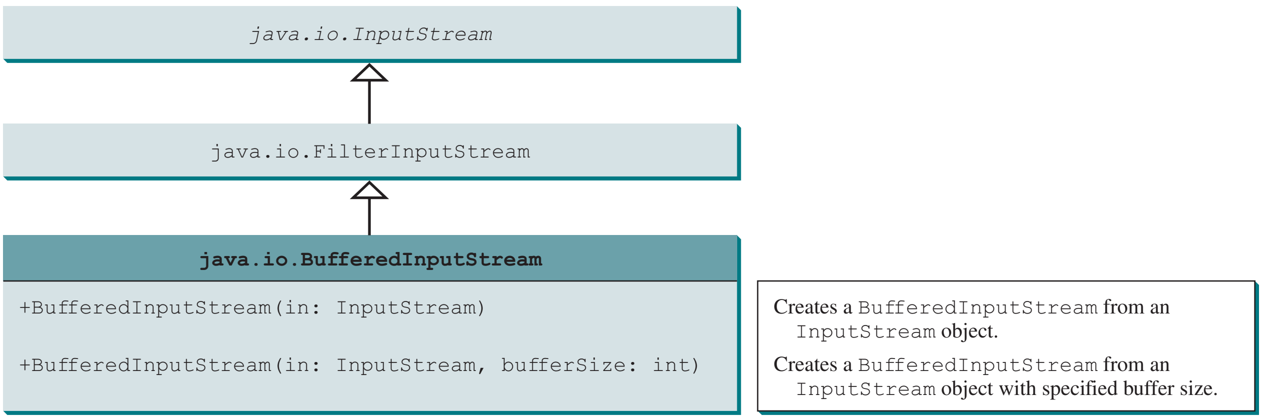


FIGURE 17.13 **BufferedInputStream** buffers an input stream.

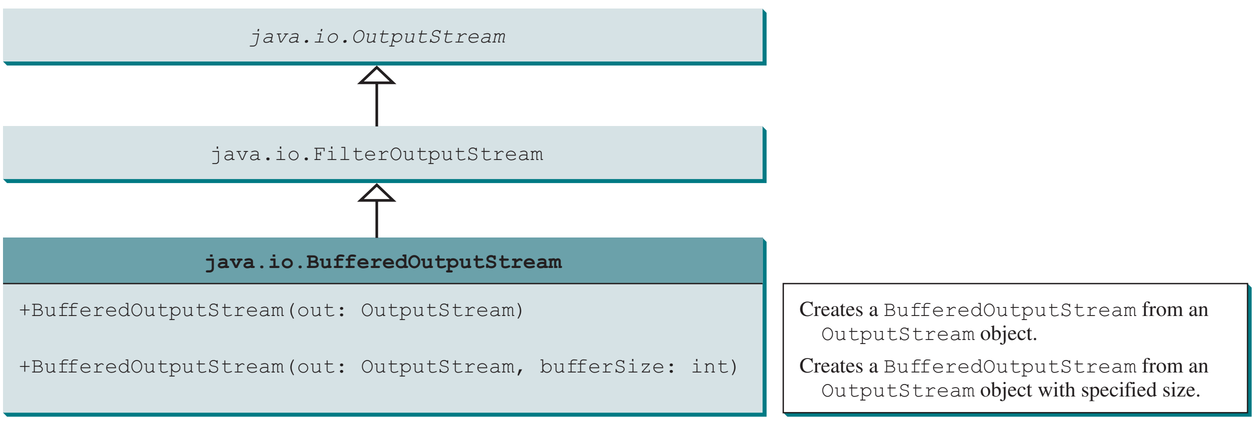


FIGURE 17.14 **BufferedOutputStream** buffers an output stream.

If no buffer size is specified, the default size is 512 bytes. You can improve the performance of the TestDataStream program in Listing 17.2 by adding buffers in the stream in lines 6–9 and 19–20, as follows:

```
DataOutputStream output = new DataOutputStream(
  new BufferedOutputStream(new FileOutputStream("temp.dat")));
DataInputStream input = new DataInputStream(
  new BufferedInputStream(new FileInputStream("temp.dat")));
```

Tip

You should always use buffered I/O to speed up input and output. For small files, you may not notice performance improvements. However, for large files—over 100 MB—you will see substantial improvements using buffered I/O.

Check Point

17.4.1 The `read()` method in `InputStream` reads a byte. Why does it return an `int` instead of a `byte`? Find the abstract methods in `InputStream` and `OutputStream`.

17.4.2 Why do you have to declare to throw `IOException` in the method or use a try-catch block to handle `IOException` for Java I/O programs?

17.4.3 Why should you always close streams? How do you close streams?

17.4.4 Does `FileInputStream`/`FileOutputStream` introduce any new methods beyond the methods inherited from `InputStream`/`OutputStream`? How do you create a `FileInputStream`/`FileOutputStream`?

17.4.5 What will happen if you attempt to create an input stream on a nonexistent file? What will happen if you attempt to create an output stream on an existing file? Can you append data to an existing file?

17.4.6 How do you append data to an existing text file using `java.io.PrintWriter`?

17.4.7 What is written to a file using `writeByte(91)` on a `FileOutputStream`?

17.4.8 What is wrong in the following code?

```
import java.io.*;

public class Test {
  public static void main(String[] args) {
    try (
      FileInputStream fis = new FileInputStream("test.dat"); ) {
    }
    catch (IOException ex) {
      ex.printStackTrace();
    }
    catch (FileNotFoundException ex) {
      ex.printStackTrace();
    }
  }
}
```

17.4.9 Suppose a file contains an unspecified number of `double` values that were written to the file using the `writeDouble` method using a `DataOutputStream`. How do you write a program to read all these values? How do you detect the end of a file?

17.4.10 How do you check the end of a file in an input stream (`FileInputStream`, `DataInputStream`)?

17.4.11 Suppose you run the following program on Windows using the default ASCII encoding after the program is finished. How many bytes are there in the file **t.txt**? Show the contents of each byte.

```
public class Test {
  public static void main(String[] args)
      throws java.io.IOException {
    try (java.io.PrintWriter output =
        new java.io.PrintWriter("t.txt"); ) {
      output.printf("%s", "1234");
      output.printf("%s", "5678");
      output.close();
    }
  }
}
```

17.4.12 After the following program is finished, how many bytes are there in the file **t.dat**? Show the contents of each byte.

```
import java.io.*;

public class Test {
  public static void main(String[] args) throws IOException {
    try (DataOutputStream output = new DataOutputStream(
        new FileOutputStream("t.dat")); ) {
      output.writeInt(1234);
      output.writeInt(5678);
      output.close();
    }
  }
}
```

17.4.13 For each of the following statements on a `DataOutputStream output`, how many bytes are sent to the output?

```
output.writeChar('A');
output.writeChars("BC");
output.writeUTF("DEF");
```

17.4.14 What are the advantages of using buffered streams? Are the following statements correct?

```
BufferedInputStream input1 =
  new BufferedInputStream(new FileInputStream("t.dat"));

DataInputStream input2 = new DataInputStream(
  new BufferedInputStream(new FileInputStream("t.dat")));

DataOutputStream output = new DataOutputStream(
  new BufferedOutputStream(new FileOutnputStream("t.dat")));
```

17.5 Case Study: Copying Files

Key Point

This section develops a useful utility for copying files.

VideoNote
Copy file

In this section, you will learn how to write a program that lets users copy files. The user needs to provide a source file and a target file as command-line arguments using the command

```
java Copy source target
```

The program copies the source file to the target file and displays the number of bytes in the file. The program should alert the user if the source file does not exist or if the target file already exists. A sample run of the program is shown in Figure 17.15.

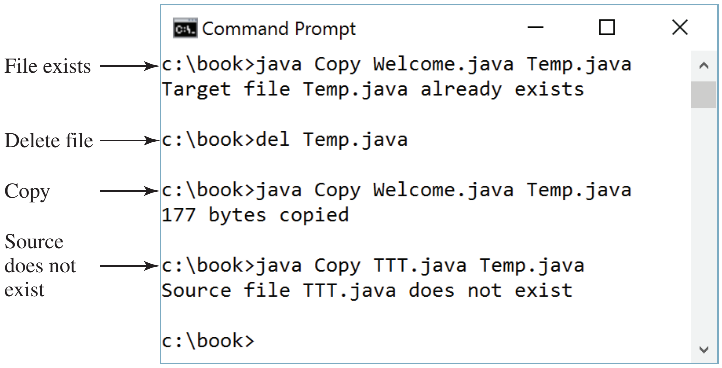


FIGURE 17.15 The program copies a file. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

To copy the contents from a source file to a target file, it is appropriate to use an input stream to read bytes from the source file, and an output stream to send bytes to the target file, regardless of the file's contents. The source file and the target file are specified from the command line. Create an `InputFileStream` for the source file, and an `OutputFileStream` for the target file. Use the `read()` method to read a byte from the input stream and then use the `write(b)` method to write the byte to the output stream. Use `BufferedInputStream` and `BufferedOutputStream` to improve the performance. Listing 17.4 gives the solution to the problem.

LISTING 17.4 Copy.java

```java
import java.io.*;

public class Copy {
  /** Main method
     @param args[0] for sourcefile
     @param args[1] for target file
   */
  public static void main(String[] args) throws IOException {
    // Check command-line parameter usage
    if (args.length != 2) {
      System.out.println(
        "Usage: java Copy sourceFile targetfile");
      System.exit(1);
    }

    // Check if source file exists
    File sourceFile = new File(args[0]);
    if (!sourceFile.exists()) {
       System.out.println("Source file " + args[0]
         + " does not exist");
       System.exit(2);
    }

    // Check if target file exists
    File targetFile = new File(args[1]);
    if (targetFile.exists()) {
      System.out.println("Target file " + args[1]
        + " already exists");
      System.exit(3);
    }

```

check usage

source file

target file

```
32       try (
33         // Create an input stream
34         BufferedInputStream input =
35           new BufferedInputStream(new FileInputStream(sourceFile));
36 
37         // Create an output stream
38         BufferedOutputStream output =
39           new BufferedOutputStream(new FileOutputStream(targetFile));
40       ) {
41         // Continuously read a byte from input and write it to output
42         int r, numberOfBytesCopied = 0;
43         while ((r = input.read()) != -1) {
44           output.write((byte)r);
45           numberOfBytesCopied++;
46         }
47 
48         // Display the file size
49         System.out.println(numberOfBytesCopied + " bytes copied");
50       }
51     }
52   }
```

input stream

output stream

read

write

The program first checks whether the user has passed the two required arguments from the command line in lines 10–14.

The program uses the `File` class to check whether the source file and target file exist. If the source file does not exist (lines 18–22), or if the target file already exists (lines 25–30), the program ends.

An input stream is created using `BufferedInputStream` wrapped on `FileInputStream` in lines 34–35, and an output stream is created using `BufferedOutputStream` wrapped on `FileOutputStream` in lines 38–39.

The expression `((r = input.read()) != -1)` (line 43) reads a byte from `input.read()`, assigns it to `r`, and checks whether it is `-1`. The input value of `-1` signifies the end of a file. The program continuously reads bytes from the input stream and sends them to the output stream until all of the bytes have been read.

Check Point

17.5.1 How does the program check if a file already exists?

17.5.2 How does the program detect the end of the file while reading data?

17.5.3 How does the program count the number of bytes read from the file?

17.6 Object I/O

Key Point

`ObjectInputStream`/`ObjectOutputStream` classes can be used to read/write serializable objects.

VideoNote

Object I/O

`DataInputStream`/`DataOutputStream` enables you to perform I/O for primitive-type values and strings. `ObjectInputStream`/`ObjectOutputStream` enables you to perform I/O for objects in addition to primitive-type values and strings. Since `ObjectInputStream`/`ObjectOutputStream` contains all the functions of `DataInputStream`/`DataOutputStream`, you can replace `DataInputStream`/`DataOutputStream` completely with `ObjectInputStream`/`ObjectOutputStream`.

`ObjectInputStream` extends `InputStream` and implements `ObjectInput` and `ObjectStreamConstants`, as shown in Figure 17.16. `ObjectInput` is a subinterface of `DataInput` (`DataInput` is shown in Figure 17.9). `ObjectStreamConstants` contains the constants to support `ObjectInputStream`/`ObjectOutputStream`.

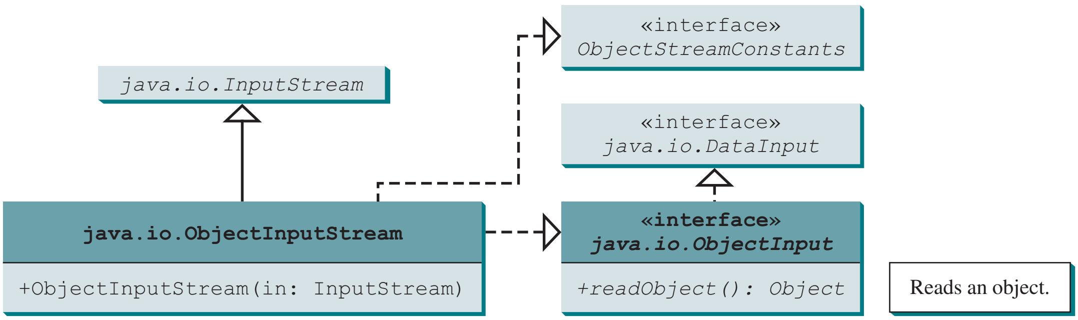


FIGURE 17.16 `ObjectInputStream` can read objects, primitive-type values, and strings.

`ObjectOutputStream` extends `OutputStream` and implements `ObjectOutput` and `ObjectStreamConstants`, as shown in Figure 17.17. `ObjectOutput` is a subinterface of `DataOutput` (`DataOutput` is shown in Figure 17.10).

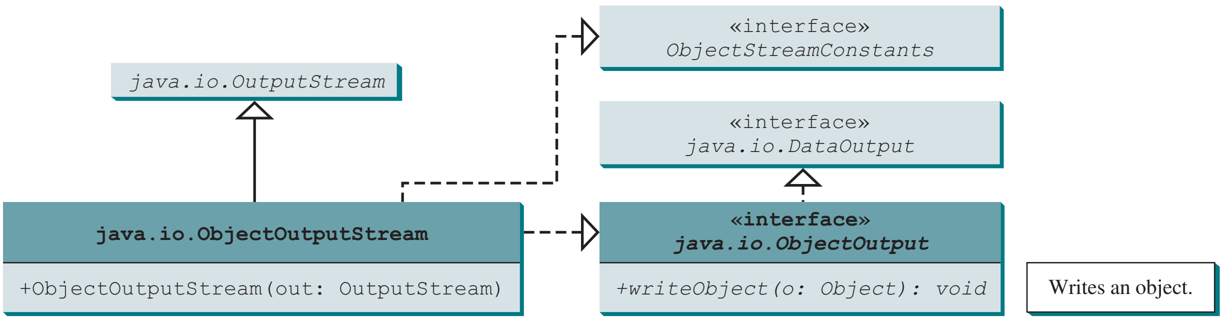


FIGURE 17.17 `ObjectOutputStream` can write objects, primitive-type values, and strings.

You can wrap an `ObjectInputStream`/`ObjectOutputStream` on any `InputStream`/`OutputStream` using the following constructors:

```
// Create an ObjectInputStream
public ObjectInputStream(InputStream in)

// Create an ObjectOutputStream
public ObjectOutputStream(OutputStream out)
```

Listing 17.5 writes students' names, scores, and the current date to a file named **object.dat**.

LISTING 17.5 TestObjectOutputStream.java

```
import java.io.*;

public class TestObjectOutputStream {
  public static void main(String[] args) throws IOException {
    try ( // Create an output stream for file object.dat
      ObjectOutputStream output =
        new ObjectOutputStream(new FileOutputStream("object.dat"));
    ) {
      // Write a string, double value, and object to the file
      output.writeUTF("John");
      output.writeDouble(85.5);
```

output stream

output string

output object

```
12         output.writeObject(new java.util.Date());
13       }
14     }
15   }
```

An **`ObjectOutputStream`** is created to write data into the file **object.dat** in lines 6 and 7. A string, a double value, and an object are written to the file in lines 10–12. To improve performance, you may add a buffer in the stream using the following statement to replace lines 6 and 7:

```
ObjectOutputStream output = new ObjectOutputStream(
  new BufferedOutputStream(new FileOutputStream("object.dat")));
```

Multiple objects or primitives can be written to the stream. The objects must be read back from the corresponding **`ObjectInputStream`** with the same types and in the same order as they were written. Java's safe casting should be used to get the desired type. Listing 17.6 reads data from **object.dat**.

LISTING 17.6 TestObjectInputStream.java

input stream

input string

input object

```
 1  import java.io.*;
 2
 3  public class TestObjectInputStream {
 4    public static void main(String[] args)
 5      throws ClassNotFoundException, IOException {
 6      try ( // Create an input stream for file object.dat
 7        ObjectInputStream input =
 8          new ObjectInputStream(new FileInputStream("object.dat"));
 9      ) {
10        // Read a string, double value, and object from the file
11        String name = input.readUTF();
12        double score = input.readDouble();
13        java.util.Date date = (java.util.Date)(input.readObject());
14        System.out.println(name + " " + score + " " + date);
15      }
16    }
17  }
```

```
John 85.5 Sun Dec 04 10:35:31 EST 2011
```

`ClassNotFoundException`

The **`readObject()`** method may throw **`java.lang.ClassNotFoundException`** because when the JVM restores an object, it first loads the class for the object if the class has not been loaded. Since **`ClassNotFoundException`** is a checked exception, the **`main`** method declares to throw it in line 5. An **`ObjectInputStream`** is created to read input from **object.dat** in lines 7–8. You have to read the data from the file in the same order and format as they were written to the file. A string, a double value, and an object are read in lines 11–13. Since **`readObject()`** returns an **`Object`**, it is cast into **`Date`** and assigned to a **`Date`** variable in line 13.

17.6.1 The `Serializable` Interface

serializable

Not every object can be written to an output stream. Objects that can be so written are said to be *serializable*. A serializable object is an instance of the **`java.io.Serializable`** interface, so the object's class must implement **`Serializable`**.

The `Serializable` interface is a marker interface. Since it has no methods, you don't need to add additional code in your class that implements `Serializable`. Implementing this interface enables the Java serialization mechanism to automate the process of storing objects and arrays.

To appreciate this automation feature, consider what you otherwise need to do in order to store an object. Suppose that you wish to store an `ArrayList` object. To do this, you need to store all the elements in the list. Each element is an object that may contain other objects. As you can see, this would be a very tedious process. Fortunately, you don't have to go through it manually. Java provides a built-in mechanism to automate the process of writing objects. This process is referred as *object serialization*, which is implemented in `ObjectOutputStream`. In contrast, the process of reading objects is referred as *object deserialization*, which is implemented in `ObjectInputStream`.

serialization

deserialization

Many classes in the Java API implement `Serializable`. All the wrapper classes for primitive-type values: `java.math.BigInteger`, `java.math.BigDecimal`, `java.lang.String`, `java.lang.StringBuilder`, `java.lang.StringBuffer`, `java.util.Date`, and `java.util.ArrayList` implement `java.io.Serializable`. Attempting to store an object that does not support the `Serializable` interface would cause a `NotSerializableException`.

NotSerializableException

When a serializable object is stored, the class of the object is encoded; this includes the class name and the signature of the class, the values of the object's instance variables, and the closure of any other objects referenced by the object. The values of the object's static variables are not stored.

Note

Nonserializable fields

If an object is an instance of `Serializable` but contains nonserializable instance data fields, can it be serialized? The answer is no. To enable the object to be serialized, mark these data fields with the `transient` keyword to tell the JVM to ignore them when writing the object to an object stream. Consider the following class:

```
public class C implements java.io.Serializable {
  private int v1;
  private static double v2;
  private transient A v3 = new A();
}

class A { } // A is not serializable
```

When an object of the `C` class is serialized, only variable `v1` is serialized. Variable `v2` is not serialized because it is a static variable, and variable `v3` is not serialized because it is marked `transient`. If `v3` were not marked `transient`, a `java.io.NotSerializableException` would occur.

transient

Note

Duplicate objects

If an object is written to an object stream more than once, will it be stored in multiple copies? No, it will not. When an object is written for the first time, a serial number is created for it. The JVM writes the complete contents of the object along with the serial number into the object stream. After the first time, only the serial number is stored if the same object is written again. When the objects are read back, their references are the same since only one object is actually created in the memory.

17.6.2 Serializing Arrays

An array is serializable if all its elements are serializable. An entire array can be saved into a file using `writeObject` and later can be restored using `readObject`. Listing 17.7 stores an array of five `int` values and an array of three strings, and reads them back to display on the console.

LISTING 17.7 TestObjectStreamForArray.java

output stream
store array
input stream
restore array

```
1  import java.io.*;
2
3  public class TestObjectStreamForArray {
4    public static void main(String[] args)
5        throws ClassNotFoundException, IOException {
6      int[] numbers = {1, 2, 3, 4, 5};
7      String[] strings = {"John", "Susan", "Kim"};
8
9      try ( // Create an output stream for file array.dat
10       ObjectOutputStream output = new ObjectOutputStream(new
11         FileOutputStream("array.dat", true));
12     ) {
13       // Write arrays to the object output stream
14       output.writeObject(numbers);
15       output.writeObject(strings);
16     }
17
18     try ( // Create an input stream for file array.dat
19       ObjectInputStream input =
20         new ObjectInputStream(new FileInputStream("array.dat"));
21     ) {
22       int[] newNumbers = (int[])(input.readObject());
23       String[] newStrings = (String[])(input.readObject());
24
25       // Display arrays
26       for (int i = 0; i < newNumbers.length; i++)
27         System.out.print(newNumbers[i] + " ");
28       System.out.println();
29
30       for (int i = 0; i < newStrings.length; i++)
31         System.out.print(newStrings[i] + " ");
32     }
33   }
34 }
```

```
1 2 3 4 5
John Susan Kim
```

Lines 14–15 write two arrays into file **array.dat**. Lines 22–23 read two arrays back in the same order they were written. Since `readObject()` returns `Object`, casting is used to cast the objects into `int[]` and `String[]`.

Check Point

17.6.1 Is it true that `DataInputStream`/`DataOutputStream` can always be replaced by `ObjectInputStream`/`ObjectOutputStream`?

17.6.2 What types of objects can be stored using the `ObjectOutputStream`? What is the method for writing an object? What is the method for reading an object? What is the return type of the method that reads an object from `ObjectInputStream`?

17.6.3 If you serialize two objects of the same type, will they take the same amount of space? If not, give an example.

17.6.4 Is it true that any instance of `java.io.Serializable` can be successfully serialized? Are the static variables in an object serialized? How do you mark an instance variable not to be serialized?

17.6.5 What will happen when you attempt to run the following code?

```
import java.io.*;

public class Test {
  public static void main(String[] args) throws IOException {
    try ( ObjectOutputStream output =
        new ObjectOutputStream(new FileOutputStream("object.dat")); ) {
      output.writeObject(new A());
    }
  }
}

class A implements Serializable {
  B b = new B();
}

class B {
}
```

17.6.6 Can you write an array to an `ObjectOutputStream`?

17.7 Random-Access Files

Java provides the `RandomAccessFile` *class to allow data to be read from and written to at any locations in the file.*

Key Point

All of the streams you have used so far are known as *read-only* or *write-only* streams. These streams are called *sequential streams*. A file that is opened using a sequential stream is called a *sequential-access file*. The contents of a sequential-access file cannot be updated. However, it is often necessary to modify files. Java provides the `RandomAccessFile` class to allow data to be read from and written to at any locations in the file. A file that is opened using the `RandomAccessFile` class is known as a *random-access file*.

read-only
write-only
sequential-access file
random-access file

The `RandomAccessFile` class implements the `DataInput` and `DataOutput` interfaces, as shown in Figure 17.18. The `DataInput` interface (see Figure 17.9) defines the methods for reading primitive-type values and strings (e.g., `readInt`, `readDouble`, `readChar`, `readBoolean`, and `readUTF`) and the `DataOutput` interface (see Figure 17.10) defines the methods for writing primitive-type values and strings (e.g., `writeInt`, `writeDouble`, `writeChar`, `writeBoolean`, and `writeUTF`).

When creating a `RandomAccessFile`, you can specify one of the two modes: `r` or `rw`. Mode `r` means that the stream is read-only, and mode `rw` indicates that the stream allows both read and write. For example, the following statement creates a new stream, `raf`, that allows the program to read from and write to the file **test.dat**:

```
RandomAccessFile raf = new RandomAccessFile("test.dat", "rw");
```

If **test.dat** already exists, `raf` is created to access it; if **test.dat** does not exist, a new file named **test.dat** is created and `raf` is created to access the new file. The method `raf.length()` returns the number of bytes in **test.dat** at any given time. If you append new data into the file, `raf.length()` increases.

Tip

If the file is not intended to be modified, open it with the `r` mode. This prevents unintentional modification of the file.

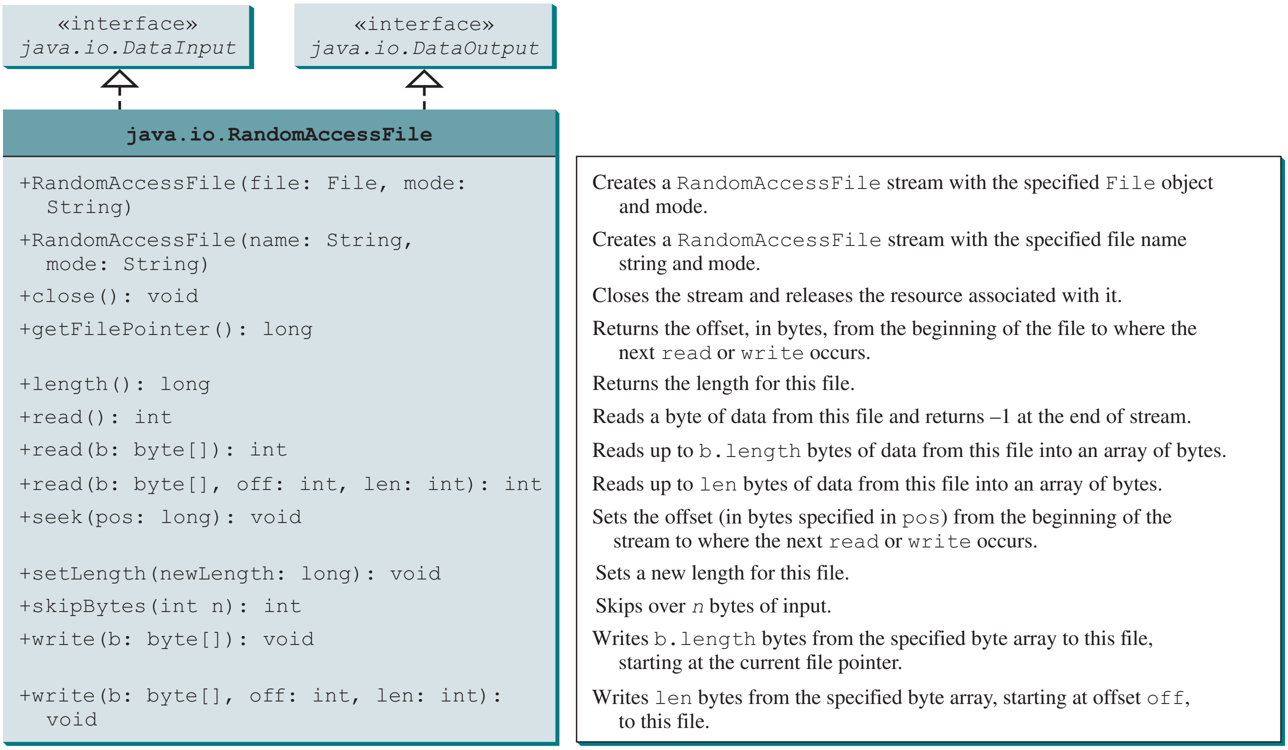


FIGURE 17.18 `RandomAccessFile` implements the `DataInput` and `DataOutput` interfaces with additional methods to support random access.

file pointer

A random-access file consists of a sequence of bytes. A special marker called a *file pointer* is positioned at one of these bytes. A read or write operation takes place at the location of the file pointer. When a file is opened, the file pointer is set at the beginning of the file. When you read from or write data to the file, the file pointer moves forward to the next data item. For example, if you read an `int` value using `readInt()`, the JVM reads `4` bytes from the file pointer and now the file pointer is `4` bytes ahead of the previous location, as shown in Figure 17.19.

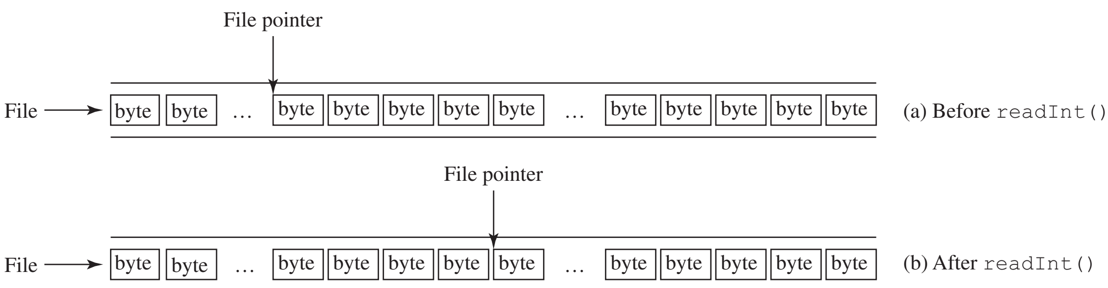


FIGURE 17.19 After an `int` value is read, the file pointer is moved 4 bytes ahead.

For a `RandomAccessFile raf`, you can use the `raf.seek(position)` method to move the file pointer to a specified position. `raf.seek(0)` moves it to the beginning of the file and `raf.seek(raf.length())` moves it to the end of the file. Listing 17.8 demonstrates `RandomAccessFile`. A large case study of using `RandomAccessFile` to organize an address book is given in Supplement VI.D.

LISTING 17.8 TestRandomAccessFile.java

```java
1  import java.io.*;
2
3  public class TestRandomAccessFile {
4    public static void main(String[] args) throws IOException {
5      try ( // Create a random access file
6        RandomAccessFile inout = new RandomAccessFile("inout.dat", "rw");
7      ) {
8        // Clear the file to destroy the old contents if exists
9        inout.setLength(0);
10
11       // Write new integers to the file
12       for (int i = 0; i < 200; i++)
13         inout.writeInt(i);
14
15       // Display the current length of the file
16       System.out.println("Current file length is " + inout.length());
17
18       // Retrieve the first number
19       inout.seek(0); // Move the file pointer to the beginning
20       System.out.println("The first number is " + inout.readInt());
21
22       // Retrieve the second number
23       inout.seek(1 * 4); // Move the file pointer to the second number
24       System.out.println("The second number is " + inout.readInt());
25
26       // Retrieve the tenth number
27       inout.seek(9 * 4); // Move the file pointer to the tenth number
28       System.out.println("The tenth number is " + inout.readInt());
29
30       // Modify the eleventh number
31       inout.writeInt(555);
32
33       // Append a new number
34       inout.seek(inout.length()); // Move the file pointer to the end
35       inout.writeInt(999);
36
37       // Display the new length
38       System.out.println("The new length is " + inout.length());
39
40       // Retrieve the new eleventh number
41       inout.seek(10 * 4); // Move the file pointer to the eleventh number
42       System.out.println("The eleventh number is " + inout.readInt());
43     }
44   }
45 }
```

RandomAccessFile

empty file

write

move pointer

read

```
Current file length is 800
The first number is 0
The second number is 1
The tenth number is 9
The new length is 804
The eleventh number is 555
```

A **RandomAccessFile** is created for the file named **inout.dat** with mode **rw** to allow both read and write operations in line 6.

`inout.setLength(0)` sets the length to `0` in line 9. This, in effect, deletes the old contents of the file.

The `for` loop writes `200` `int` values from `0` to `199` into the file in lines 12–13. Since each `int` value takes `4` bytes, the total length of the file returned from `inout.length()` is now `800` (line 16), as shown in the sample output.

Invoking `inout.seek(0)` in line 19 sets the file pointer to the beginning of the file. `inout.readInt()` reads the first value in line 20 and moves the file pointer to the next number. The second number is read in line 24.

`inout.seek(9 * 4)` (line 27) moves the file pointer to the tenth number. `inout.readInt()` reads the tenth number and moves the file pointer to the eleventh number in line 28. `inout.write(555)` writes a new eleventh number at the current position (line 31). The previous eleventh number is deleted.

`inout.seek(inout.length())` moves the file pointer to the end of the file (line 34). `inout.writeInt(999)` writes a `999` to the file (line 35). Now the length of the file is increased by `4`, so `inout.length()` returns `804` (line 38).

`inout.seek(10 * 4)` moves the file pointer to the eleventh number in line 41. The new eleventh number, `555`, is displayed in line 42.

Check Point

17.7.1 Can `RandomAccessFile` streams read and write a data file created by `DataOutputStream`? Can `RandomAccessFile` streams read and write objects?

17.7.2 Create a `RandomAccessFile` stream for the file **address.dat** to allow the updating of student information in the file. Create a `DataOutputStream` for the file **address.dat**. Explain the differences between these two statements.

17.7.3 What happens if the file **test.dat** does not exist when you attempt to compile and run the following code?

```
import java.io.*;

public class Test {
  public static void main(String[] args) {
    try ( RandomAccessFile raf =
        new RandomAccessFile("test.dat", "r"); ) {
      int i = raf.readInt();
    }
    catch (IOException ex) {
      System.out.println("IO exception");
    }
  }
}
```

Key Terms

binary I/O 714
deserialization 731
file pointer 734
random-access file 733
sequential-access file 733
serialization 731
stream 714
text I/O 714

Chapter Summary

1. I/O can be classified into *text I/O* and *binary I/O*. Text I/O interprets data in sequences of characters. Binary I/O interprets data as raw binary values. How text is stored in a file depends on the encoding scheme for the file. Java automatically performs encoding and decoding for text I/O.

2. The `InputStream` and `OutputStream` classes are the roots of all binary I/O classes. `FileInputStream`/`FileOutputStream` associates a file for input/output. `BufferedInputStream`/`BufferedOutputStream` can be used to wrap any binary I/O stream to improve performance. `DataInputStream`/`DataOutputStream` can be used to read/write primitive values and strings.

3. `ObjectInputStream`/`ObjectOutputStream` can be used to read/write objects in addition to primitive values and strings. To enable object *serialization*, the object's defining class must implement the `java.io.Serializable` marker interface.

4. The `RandomAccessFile` class enables you to read and write data to a file. You can open a file with the `r` mode to indicate that it is read-only, or with the `rw` mode to indicate that it is updateable. Since the `RandomAccessFile` class implements `DataInput` and `DataOutput` interfaces, many methods in `RandomAccessFile` are the same as those in `DataInputStream` and `DataOutputStream`.

Quiz

Answer the quiz for this chapter online at the book Companion Website.

Programming Exercises

MyProgrammingLab™

Section 17.3

***17.1** (*Create a text file*) Write a program to create a file named **Exercise17_01.txt** if it does not exist. Append new data to it if it already exists. Write 150 integers created randomly into the file using text I/O. Integers are separated by a space.

Section 17.4

***17.2** (*Create a binary data file*) Write a program to create a file named **Exercise17_02.dat** if it does not exist. Append new data to it if it already exists. Write 150 integers created randomly into the file using binary I/O.

***17.3** (*Sum all the floating points in a binary data file*) Suppose a binary data file named **Exercise17_03.dat** has been created and its data are created using `writeDouble(double)` in `DataOutputStream`. The file contains an unspecified number of floating points. Write a program to find the sum of the floating points.

***17.4** (*Writing combination of data types into binary file*) Write a program that writes information about a student in an examination – student ID (int), student name (string), and exam score (double) – into a file **Exercise17_04.dat**, using `DataOutputStream`. Then create another program that reads the file **Exercise17_04.dat** using `DataInputStream` and prints the three attributes. Use the following commands to run the programs:

```
java Exercise17_04_write
java Exercise17_04_read
```

Section 17.6

***17.5** (*Store objects and arrays in a file*) Write a program that stores an array of the six `int` values `1`, `2`, `3`, `4`, `5`, and `6`, a `Date` object for the current time, and the `double` value `10.5` into the file named **Exercise17_05.dat**.

***17.6** (*Store* `StudentExamScore` *objects*) Create a `StudentExamScore` class – consisting of student ID (int), student name (string), and examination score (double) – that implements `Serializable`. Write a program that creates five `StudentExamScore` objects and stores them in a file named **Exercise17_06.dat**.

***17.7** (*Restore objects from a file*) Using the same `StudentExamScore` class from Exercise 17.6, write a program that reads `StudentExamScore` from **Exercise17_06.dat** and then outputs the average examination score, the highest examination score, and the name and ID of the student who achieved the highest exam score. Suppose that you don't know how many `StudentExamScore` are there in the file, use `EOFException` to end the loop.

Section 17.7

***17.8** (*Update count*) Suppose that you wish to track how many times a program has been executed. You can store an `int` to count the file. Increase the count by `1` each time this program is executed. Let the program be **Exercise17_08.txt** and store the count in **Exercise17_08.dat**.

*****17.9** (*Address book*) Write a program that stores, retrieves, adds, and updates addresses as shown in Figure 17.20. Use a fixed-length string for storing each attribute in the address. Use random-access file for reading and writing an address. Assume the sizes of the name, street, city, state, and zip are 32, 32, 20, 2, and 5 bytes, respectively.

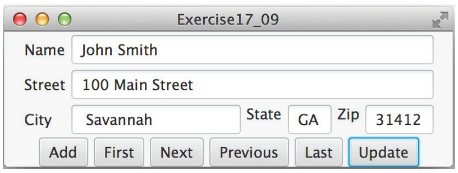


FIGURE 17.20 The application can store, retrieve, and update addresses from a file. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

Comprehensive

VideoNote
Split a large file

***17.10** (*Split files*) Suppose you want to back up a huge file (e.g., a 10-GB AVI file) to a CD-R. You can achieve it by splitting the file into smaller pieces and backing up these pieces separately. Write a utility program that splits a large file into smaller ones using the following command:

```
java Exercise17_10 SourceFile numberOfPieces
```

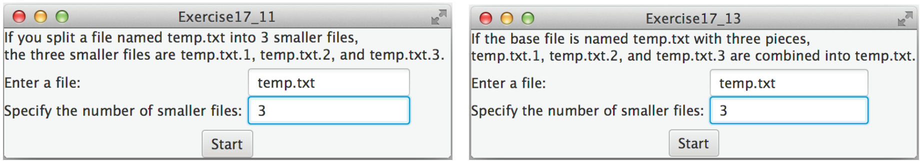
The command creates the files **SourceFile.1**, **SourceFile.2**, . . . , **SourceFile.n**, where **n** is `numberOfPieces` and the output files are about the same size.

****17.11** (*Split files GUI*) Rewrite Exercise 17.10 with a GUI, as shown in Figure 17.21a.

***17.12** (*Combine files*) Write a utility program that combines the files together into a new file using the following command:

```
java Exercise17_12 SourceFile1 . . . SourceFilen TargetFile
```

The command combines SourceFile1, . . . , and SourceFilen into TargetFile.



(a) (b)

FIGURE 17.21 (a) The program splits a file. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission. (b) The program combines files into a new file.

***17.13** (*Combine files GUI*) Rewrite Exercise 17.12 with a GUI, as shown in Figure 17.21b.

17.14 (*Encrypt files*) Encode the file by adding `5` to every byte in the file. Write a program that prompts the user to enter an input file name and an output file name and saves the encrypted version of the input file to the output file.

17.15 (*Decrypt files*) Suppose a file is encrypted using the scheme in Programming Exercise 17.14. Write a program to decode an encrypted file. Your program should prompt the user to enter an input file name for the encrypted file and an output file name for the unencrypted version of the input file.

17.16 (*Frequency of characters*) Write a program that prompts the user to enter the name of an ASCII text file and displays the frequency of the characters in the file.

****17.17** (*`BitOutputStream`*) Implement a class named `BitOutputStream`, as shown in Figure 17.22, for writing bits to an output stream. The `writeBit(char bit)` method stores the bit in a byte variable. When you create a `BitOutputStream`, the byte is empty. After invoking `writeBit('1')`, the byte becomes `00000001`. After invoking `writeBit("0101")`, the byte becomes `00010101`. The first three bits are not filled yet. When a byte is full, it is sent to the output stream. Now the byte is reset to empty. You must close the stream by invoking the `close()` method. If the byte is neither empty nor full, the `close()` method first fills the zeros to make a full `8` bits in the byte and then outputs the byte and closes the stream. For a hint, see Programming Exercise 5.44. Write a test program that sends the bits `010000100100001001101` to the file named **Exercise17_17.dat**.

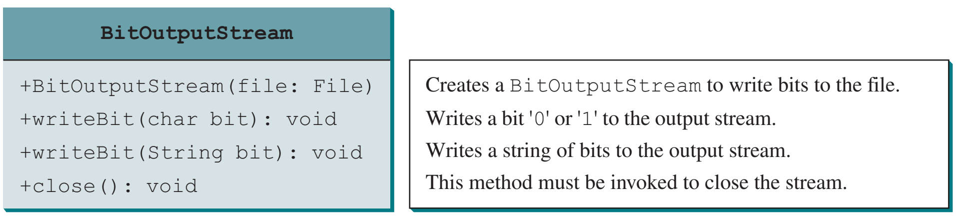


FIGURE 17.22 `BitOutputStream` outputs a stream of bits to a file.

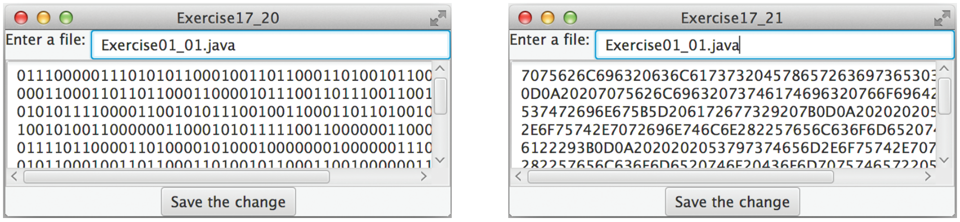
***17.18** (*View bits*) Write the following method that displays the bit representation for the last byte in an integer:

```java
public static String getBits(int value)
```

For a hint, see Programming Exercise 5.44. Write a program that prompts the user to enter a file name, reads bytes from the file, and displays each byte's binary representation.

***17.19** (*View hex*) Write a program that prompts the user to enter a file name, reads bytes from the file, and displays each byte's hex representation. (*Hint*: You can first convert the byte value into an 8-bit string, then convert the bit string into a two-digit hex string.)

****17.20** (*Binary editor*) Write a GUI application that lets the user to enter a file name in the text field and press the *Enter* key to display its binary representation in a text area. The user can also modify the binary code and save it back to the file, as shown in Figure 17.23a.



(a) (b)

FIGURE 17.23 The programs enable the user to manipulate the contents of the file in (a) binary (b) hex. *Source*:

****17.21** (*Hex editor*) Write a GUI application that lets the user to enter a file name in the text field and press the *Enter* key to display its hex representation in a text area. The user can also modify the hex code and save it back to the file, as shown in Figure 17.23b.