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

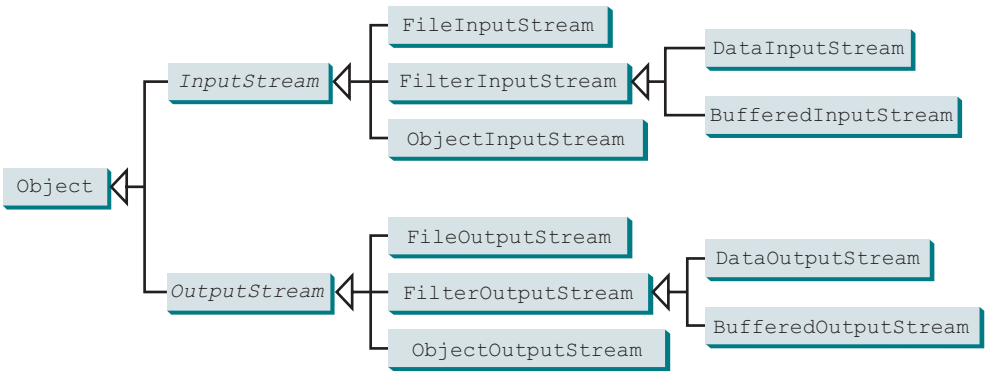

FIGURE 17.3 `InputStream`, `OutputStream`, and their subclasses are for performing binary I/O.

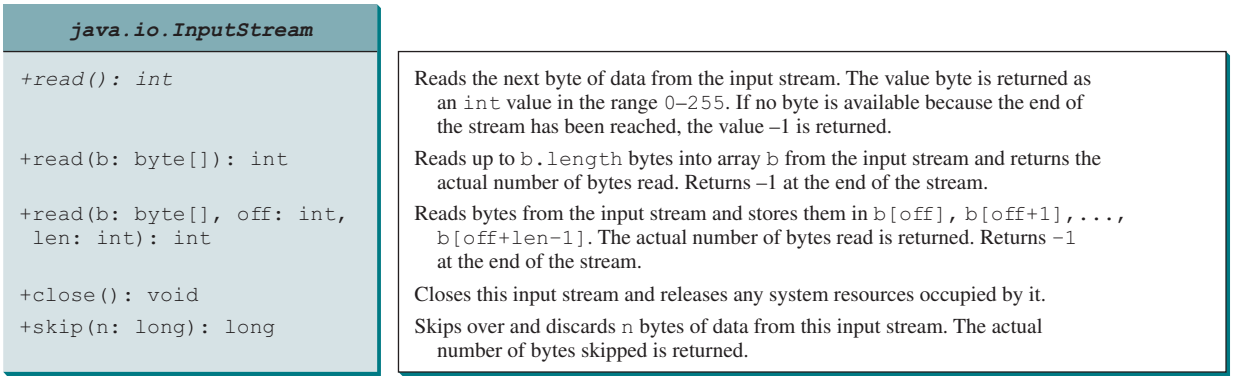

FIGURE 17.4 The abstract `InputStream` class defines the methods for the input stream of bytes.

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`

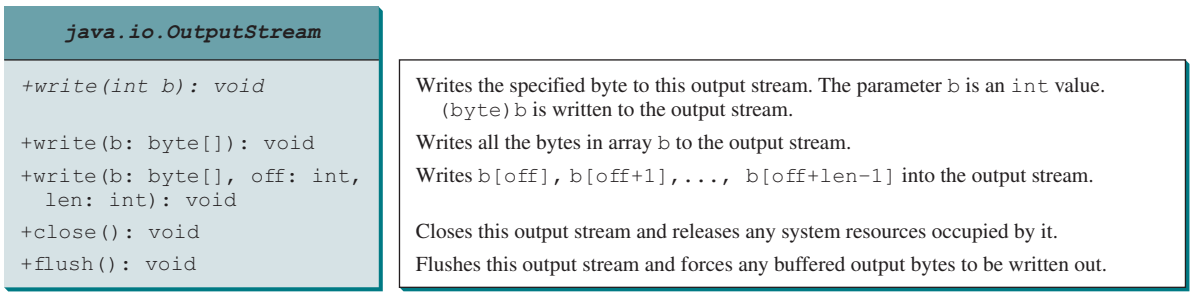

FIGURE 17.5 The abstract `OutputStream` class defines the methods for the output stream of bytes.

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.

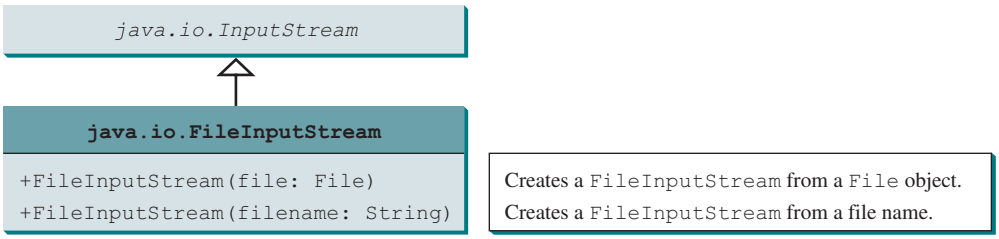

FIGURE 17.6 `FileInputStream` inputs a stream of bytes from a file.

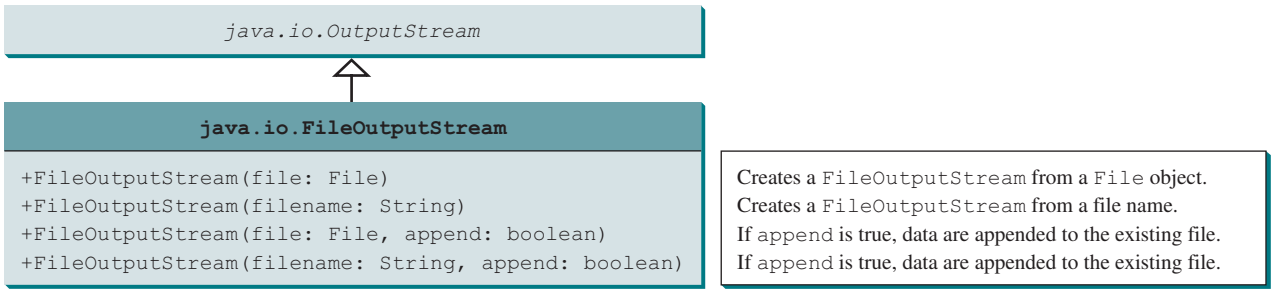

FIGURE 17.7 `FileOutputStream` outputs a stream of bytes to a file.

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

```
1  import java.io.*;
2
3  public class TestFileStream {
4    public static void main(String[] args) throws IOException {
5      try (
6        // Create an output stream to the file
7        FileOutputStream output = new FileOutputStream("temp.dat");
8      ) {
9        // Output values to the file
10       for (int i = 1; i <= 10; i++)
11         output.write(i);
12     }
13
14     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 © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

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.

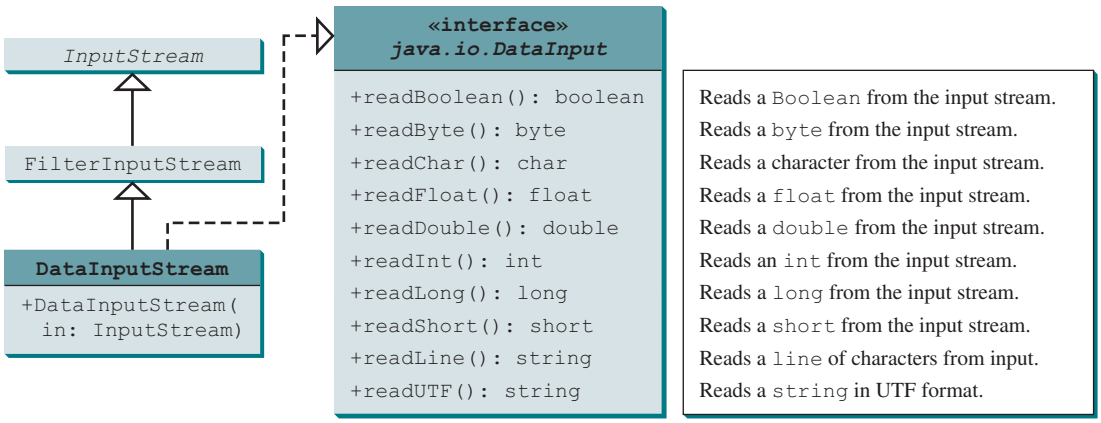

FIGURE 17.9 `DataInputStream` filters an input stream of bytes into primitive data-type values and strings.

`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

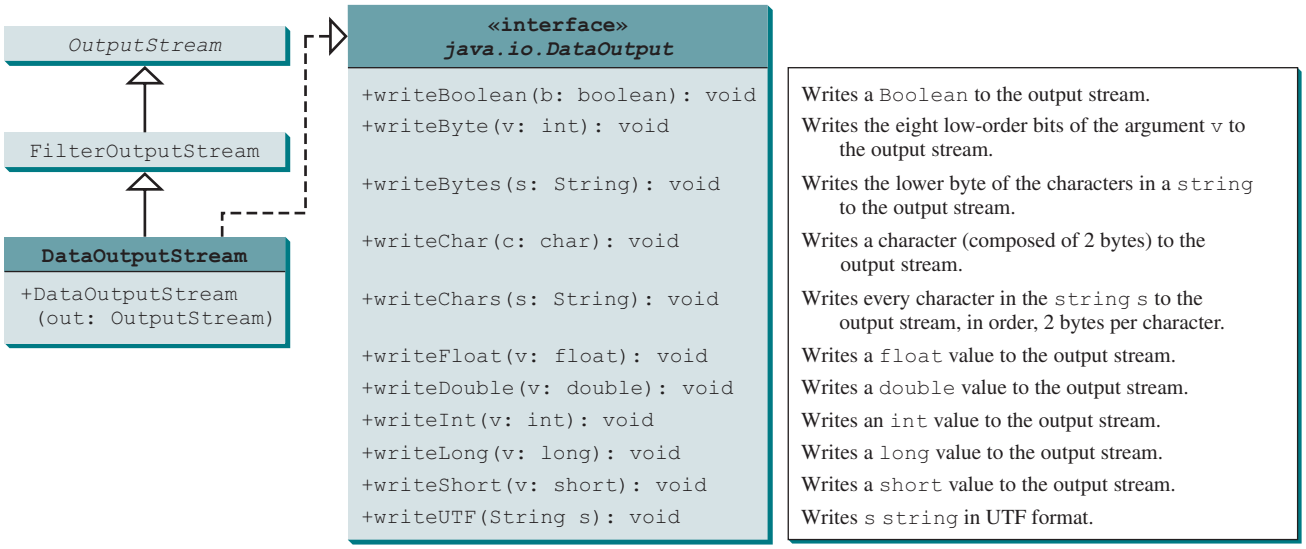

FIGURE 17.10 `DataOutputStream` enables you to write primitive data-type values and strings into an output stream.

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

```
1  import java.io.*;
2
3  public class TestDataStream {
4    public static void main(String[] args) throws IOException {
5      try ( // Create an output stream for file temp.dat
6        DataOutputStream output =                                    output stream
7          new DataOutputStream(new FileOutputStream("temp.dat"));
8      ) {
```

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.

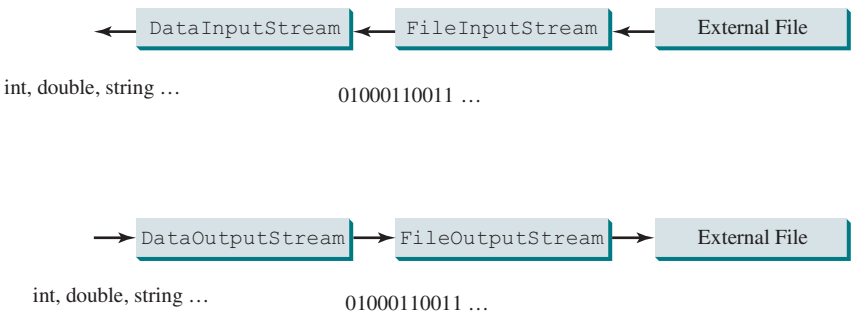

FIGURE 17.11 **DataInputStream** filters an input stream of byte to data and **DataOutputStream** converts data into a stream of bytes.

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 =                                    output stream
7          new DataOutputStream(new FileOutputStream("test.dat"))) {
8          output.writeDouble(4.5);                                       output
9          output.writeDouble(43.25);
10         output.writeDouble(3.2);
11       }
12
13       try (DataInputStream input =                                     input stream
14         new DataInputStream(new FileInputStream("test.dat"))) {
15         while (true)
16           System.out.println(input.readDouble());                      input
17       }
18     }
19     catch (EOFException ex) {                                          EOFException
20       System.out.println("All data were read");
21     }
22     catch (IOException ex) {
23       ex.printStackTrace();
24     }
25   }
26 }
```

```
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.

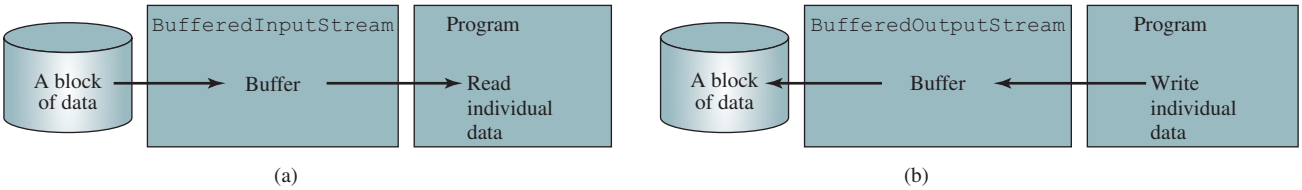

FIGURE 17.12 Buffer I/O places data in a buffer for fast processing.

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.

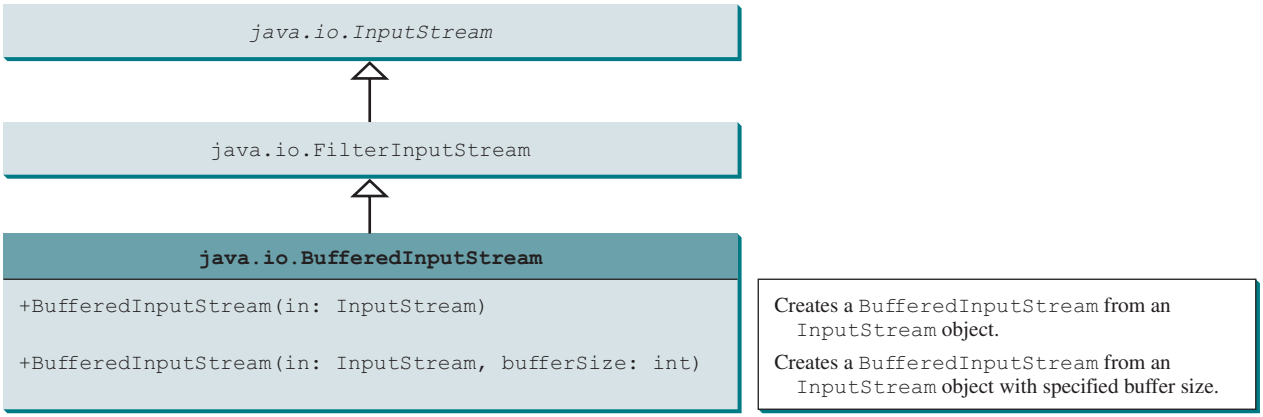

FIGURE 17.13 **BufferedInputStream** buffers an input stream.

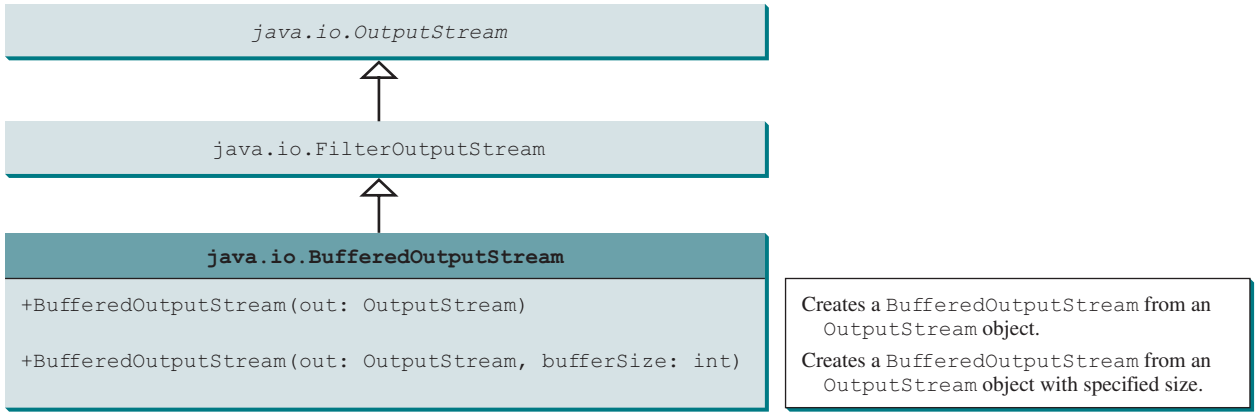

FIGURE 17.14 **BufferedOutputStream** buffers an output stream.

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.

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

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.