As you may have noticed while reading the preceding 12 chapters, not much use has been made of I/O in the example programs. In fact, aside from **print( )** and **println( )**, none of the I/O methods have been used significantly. The reason is simple: most real applications of Java are not text-based, console programs. Rather, they are graphically oriented programs that rely upon Java’s Abstract Window Toolkit (AWT) or Swing for interaction with the user. Although text-based programs are excellent as teaching examples, they do not constitute an important use for Java in the real world. Also, Java’s support for console I/O is limited and somewhat awkward to use—even in simple example programs. Text-based console I/O is just not very important to Java programming. The preceding paragraph notwithstanding, Java does provide strong, flexible support for I/O as it relates to files and networks. Java’s I/O system is cohesive and consistent.

**Streams**

Java programs perform I/O through streams. A *stream* is an abstraction that either produces or consumes information. A stream is linked to a physical device by the Java I/O system. All streams behave in the same manner, even if the actual physical devices to which they are linked differ. Thus, the same I/O classes and methods can be applied to any type of device. This means that an input stream can abstract many different kinds of input: from a disk file, a keyboard, or a network socket. Likewise, an output stream may refer to the console, a disk file, or a network connection. Streams are a clean way to deal with input/output without having every part of your code understand the difference between a keyboard and a network, for example. Java implements streams within class hierarchies defined in the **java.io** package.

**Byte Streams and Character Streams**

Java defines two types of streams: byte and character. *Byte streams* provide a convenient means for handling input and output of bytes. Byte streams are used, for example, when reading or writing binary data. *Character streams* provide a convenient means for handling input and output of characters. They use Unicode and, therefore, can be internationalized.

Also, in some cases, character streams are more efficient than byte streams. The original version of Java (Java 1.0) did not include character streams and, thus, all I/O was byte-oriented. Character streams were added by Java 1.1, and certain byte-oriented classes and methods were deprecated. This is why older code that doesn’t use character streams should be updated to take advantage of them, where appropriate. One other point: at the lowest level, all I/O is still byte-oriented. The character-based streams simply provide a convenient and efficient means for handling characters. An overview of both byte-oriented streams and character-oriented streams is presented in the following sections.

**The Byte Stream Classes**

Byte streams are defined by using two class hierarchies. At the top are two abstract classes: **InputStream** and **OutputStream**. Each of these abstract classes has several concrete subclasses that handle the differences between various devices, such as disk files, network connections, and even memory buffers. The byte stream classes are shown in Table 13-1. A few of these classes are discussed later in this section. Others are described in Part II. Remember, to use the stream classes, you must import **java.io**. The abstract classes **InputStream** and **OutputStream** define several key methods that the other stream classes implement. Two of the most important are **read( )** and **write( )**, which, respectively, read and write bytes of data. Both methods are declared as abstract inside **InputStream** and **OutputStream**. They are overridden by derived stream classes.

**The Character Stream Classes**

Character streams are defined by using two class hierarchies. At the top are two abstract classes, **Reader** and **Writer**. These abstract classes handle Unicode character streams. Java has several concrete subclasses of each of these. The character stream classes are shown in Table 13-2. The abstract classes **Reader** and **Writer** define several key methods that the other stream classes implement. Two of the most important methods are **read( )** and **write( )**, which read and write characters of data, respectively. These methods are overridden by derived stream classes.



**The Byte Stream Classes**

****

****

**The Character Stream I/O Classes**

**The Predefined Streams**

As you know, all Java programs automatically import the **java.lang** package. This package defines a class called **System**, which encapsulates several aspects of the run-time environment. **System** also contains three predefined stream

variables: **in**, **out**, and **err**. These fields are declared as **public**, **static**, and **final** within **System**. This means that they can be used by any other part of your program and without reference to a specific **System** object. **System.out** refers to the standard output stream. By default, this is the console. **System.in** refers to standard input, which is the keyboard by default. **System.err** refers to the standard error stream, which also is the console by default. However, these streams may be redirected to any compatible I/O device. **System.in** is an object of type **InputStream**; **System.out** and **System.err** are objects of type **PrintStream**. These are byte streams, even though they typically are used to read and

write characters from and to the console. As you will see, you can wrap these within characterbased streams, if desired.

**Reading Console Input**

In Java 1.0, the only way to perform console input was to use a byte stream, and older code that uses this approach persists. Today, using a byte stream to read console input is still technically possible, but doing so is not recommended. The preferred method of reading console input is to use a character-oriented stream, which makes your program easier to internationalize and maintain. In Java, console input is accomplished by reading from **System.in**. To obtain a characterbased stream that is attached to the console, wrap **System.in** in a **BufferedReader** object. **BufferedReader** supports a buffered input stream. Its most commonly used constructor is shown here: BufferedReader(Reader *inputReader*) Here, *inputReader* is the stream that is linked to the instance of **BufferedReader** that is being created. **Reader** is an abstract class. One of its concrete subclasses is **InputStreamReader**, which converts bytes to characters. To obtain an **InputStreamReader** object that is linked to **System.in**, use the following constructor:

InputStreamReader(InputStream *inputStream*)

Because **System.in** refers to an object of type **InputStream**, it can be used for *inputStream.* Putting it all together, the following line of code creates a **BufferedReader** that is connected to the keyboard:

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

After this statement executes, **br** is a character-based stream that is linked to the console through **System.in**.

**Reading Characters**

To read a character from a **BufferedReader**, use **read( )**. The version of **read( )** that we will be using is int read( ) throws IOException Each time that **read( )** is called, it reads a character from the input stream and returns it as an integer value. It returns –1 when the end of the stream is encountered. As you can see, it can throw an **IOException**. The following program demonstrates **read( )** by reading characters from the console until the user types a *"*q.” Notice that any I/O exceptions that might be generated are simply thrown out of **main( )**. Such an approach is common when reading from the console, but you can handle these types of errors yourself, if you chose.

**Reading Strings**

To read a string from the keyboard, use the version of **readLine( )** that is a member of the **BufferedReader** class. Its general form is shown here:

String readLine( ) throws IOException

As you can see, it returns a **String** object.

The following program demonstrates **BufferedReader** and the **readLine( )** method;

the program reads and displays lines of text until you enter the word “stop”:

// Read a string from console using a BufferedReader.

import java.io.\*;

class BRReadLines {

public static void main(String args[])

throws IOException

{

// create a BufferedReader using System.in

BufferedReader br = new BufferedReader(new

InputStreamReader(System.in));

String str;

System.out.println("Enter lines of text.");

System.out.println("Enter 'stop' to quit.");

do {

str = br.readLine();

System.out.println(str);

} while(!str.equals("stop"));

}

}

**Writing Console Output**

Console output is most easily accomplished with **print( )** and **println( )**, described earlier, which are used in most of the examples in this book. These methods are defined by the class **PrintStream** (which is the type of object referenced by **System.out**). Even though **System.out** is a byte stream, using it for simple program output is still acceptable. However,

a character-based alternative is described in the next section. Because **PrintStream** is an output stream derived from **OutputStream**, it also implements the low-level method **write( )**. Thus, **write( )** can be used to write to the console. The simplest form of **write( )** defined by **PrintStream** is shown here: void write(int *byteval*)

This method writes to the stream the byte specified by *byteval.* Although *byteval* is declared as an integer, only the low-order eight bits are written. Here is a short example that uses **write( )** to output the character “A” followed by a newline to the screen:

// Demonstrate System.out.write().

class WriteDemo {

public static void main(String args[]) {

int b;

b = 'A';

System.out.write(b);

System.out.write('\n');

}

}

**The PrintWriter Class**

Although using **System.out** to write to the console is acceptable, its use is recommended mostly for debugging purposes or for sample programs, such as those found in this book. For real-world programs, the recommended method of writing to the console when using Java is through a **PrintWriter** stream. **PrintWriter** is one of the character-based classes.

Using a character-based class for console output makes it easier to internationalize your program.

**PrintWriter** defines several constructors. The one we will use is shown here:

PrintWriter(OutputStream *outputStream*, boolean *flushOnNewline*)

Here, *outputStream* is an object of type **OutputStream**, and *flushOnNewline* controls whether Java flushes the output stream every time a **println( )** method is called. If *flushOnNewline* is **true**, flushing automatically takes place. If **false**, flushing is not automatic. **PrintWriter** supports the **print( )** and **println( )** methods for all types including **Object**.

Thus, you can use these methods in the same way as they have been used with **System.out**. If an argument is not a simple type, the **PrintWriter** methods call the object’s **toString( )** method and then print the result.

To write to the console by using a **PrintWriter**, specify **System.out** for the output stream and flush the stream after each newline. For example, this line of code creates a **PrintWriter** that is connected to console output:

PrintWriter pw = new PrintWriter(System.out, true);

The following application illustrates using a **PrintWriter** to handle console output:

// Demonstrate PrintWriter

import java.io.\*;

public class PrintWriterDemo {

public static void main(String args[]) {

PrintWriter pw = new PrintWriter(System.out, true);

pw.println("This is a string");

int i = -7;

pw.println(i);

double d = 4.5e-7;

pw.println(d);

}

}

The output from this program is shown here:

This is a string

-7

4.5E-7

Remember, there is nothing wrong with using **System.out** to write simple text output to the console when you are learning Java or debugging your programs. However, using a **PrintWriter** will make your real-world applications easier to internationalize. Because no advantage is gained by using a **PrintWriter** in the sample programs shown in this book, we

will continue to use **System.out** to write to the console.