Chapter 2

Stream and File I/O

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 they are linked to differ. Thus, the same I/O classes and methods can be applied to different types of devices. For example, the same methods that you use to write to the console can also be used to write to a disk file. Java implements I/O streams within class hierarchies defined in the java.io package.

Byte Streams and Character Streams

Modern versions of Java define two types of I/O streams: byte and character. Byte streams provide a convenient means for handling input and output of bytes. They are used, for example, when reading or writing binary data. They are especially helpful when working with files. Character streams are designed for handling the 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 fact that Java defines two different types of streams makes the I/O system quite large because two separate sets of class hierarchies (one for bytes, one for characters) are needed. 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.

The Byte Stream Classes

Byte streams are defined by using two class hierarchies. At the top of these are two abstract classes: InputStream and OutputStream. InputStream defines the characteristics common to byte input streams and OutputStream describes the behavior of byte output streams.

From InputStream and OutputStream are created several concrete subclasses that offer varying functionality and handle the details of reading and writing to various devices, such as disk files. The byte stream classes are shown in the following table.

Table 1 The Byte Stream Classes

Byte Stream Class Name	Meaning
BufferedInputStream	Buffered Input Stream
BufferedOutputStream	Buffered Output Stream
ByteArrayInputStream	Input Stream that reads from a byte array
ByteArrayOutputStream	Output Stream that write to a byte array
DataInputStream	An Input Stream that contains methods for reading the java standard data types
DataOutputStream	An Output Stream that contains methods for writing the java standard data types
FileInputStream	Input Stream that reads from a file
FileOutputStream	Output Stream that writes to a file
FilterInputStream	Implements Input stream
FilterOutputStream	Implements Output stream
InputStream	Abstract class that describes Stream Input
ObjectInputStream	Input Stream for object
ObjectOutputStream	Output Stream for object
OutputStream	Abstract class that describes Stream Output
PipedInputStream	Input Pipe
PipedOutputStream	Output Pipe
PrintStream	Output Stream that contain print() and println()
PushbackInputStream	Input Stream that allows bytes to be returned to the stream
SequenceInputStream	Input Stream that is a combination of two or more Input Stream that will be read sequentially, one after another.

The Character Stream Classes

Character streams are defined by using two class hierarchies topped by these two abstract classes: Reader and Writer. Reader is used for input, and Writer is used for output. Concrete classes derived from Reader and Writer operates on Unicode character streams.

The Predefined Streams

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. Among other things, it contains three predefined stream variables, called in, out, and err. These fields are declared as public, final, and static 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 by default the keyboard. System.err refers to the standard error stream, which is also the console by default. However, these streams can 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 are typically used to read and write characters from and to the console. The reason they are byte and not character streams is that the predefined streams were part of the original specification for Java, which did not include the character streams. As you will see, it is possible to wrap these within character-based streams if desired.

Using the Byte Streams

At the top of the byte stream hierarchy are the InputStream and OutputStream classes. In general, the methods in InputStream and OutputStream can throw an IOException on error. The methods defined by these two abstract classes are available to all of their subclasses. Thus, they form a minimal set of I/O functions that all byte streams will have.

Reading Console Input

Originally, the only way to perform console input was to use a byte stream, and much Java code still uses the byte streams exclusively. Today, you can use byte or character streams. For commercial code, the preferred method of reading console input is to use a character-oriented stream. Doing so makes your program easier to internationalize and easier to maintain. It is also more convenient to operate directly on characters rather than converting back and forth between characters and bytes. However, for sample programs, simple utility programs for your own use, and applications that deal with raw keyboard input, using the byte streams is acceptable. For this reason, a console I/O using byte stream is examined here.

Because System.in is an instance of InputStream, you automatically have access to the methods defined by InputStream. Unfortunately, InputStream defines only one input method, read( ), which reads bytes. There are three versions of read( ), which are shown here:

- int read( ) throws IOException
- int read(byte data[ ]) throws IOException
- int read(byte data[ ], int start, int max) throws IOException

The first version of read( ) reads a single character from the keyboard (from System.in). It returns –1 when the end of the stream is encountered. The second version reads bytes from the input stream and puts them into data until the array is full, the end of stream is reached, or an error occurs. It returns the number of bytes read, or –1 when the end of the stream is encountered. The third version reads input into data beginning at the location specified by start. Up to max bytes are stored. It returns the number of bytes read, or –1 when the end of the stream is reached. All throw an IOException when an error occurs. When reading from System.in, pressing ENTER generates an end-of-stream condition.

Here is a program that demonstrates reading an array of bytes from System.in. Notice that any I/O exceptions that might be generated are simply thrown out of main( ).

```
import java.io.*;
        public class Echo {
              public static void main(String[] args)throws IOException{
                        byte a [] = new byte[10];
                        System.out.println("Enter some character: ");
                            System.in.read(a);
                        for (int i = 0; i < a.length; i++){
                            System.out.println((char)a[i]);
                        }                          }                }
```

Writing Console Output

As is the case with console input, Java originally provided only byte streams for console output. Java 1.1 added character streams. For the most portable code, character streams are recommended. Because System.out is a byte stream, however, byte-based console output is still widely used.

Console output is most easily accomplished with print( ) and println(). These methods are defined by the class PrintStream (which is the type of the object referenced by System.out). Even though System.out is a byte stream, it is still acceptable to use this stream for simple console output.

Since PrintStream is an output stream derived from OutputStream, it also implements the low-level method write( ). Thus, it is possible to write to the console by using write( ). The simplest form of write( ) defined by PrintStream is shown here:
void write(int byteval)

Here is a short example that uses write( ) to output the character X followed by a new line:

```
class writeDemo{
pubic static void main(Strin[] args) throws IOExceptio{
int b;
b = ‘X’;
Systm.out.write(b);
Systm.out.write(”\n”);
}     }
```

You will not often use write( ) to perform console output (although it might be useful in some situations), since print( ) and println( ) are substantially easier to use.

Reading and Writing Files Using Byte Streams

Java provides a number of classes and methods that allow you to read and write files. Of course, the most common types of files are disk files. In Java, all files are byte-oriented, and Java provides methods to read and write bytes from and to a file. Thus, reading and writing files using byte streams is very common. However, Java allows you to wrap a byte-oriented file stream within a character-based object, which is shown later in this chapter.

To create a byte stream linked to a file, use FileInputStream or FileOutputStream. To open a file, simply create an object of one of these classes, specifying the name of the file as an argument to the constructor. Once the file is open, you can read from or write to it.

Inputting from a File

A file is opened for input by creating a FileInputStream object. Here is a commonly used constructor:

```
FileInputStream(String fileName) throws FileNotFoundException
```

Here, fileName specifies the name of the file you want to open. If the file does not exist, then FileNotFoundException is thrown. FileNotFoundException is a subclass of IOException.

To read from a file, you can use read( ). The version that we will use is shown here:

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

Each time it is called, read( ) reads a single byte from the file and returns it as an integer value. It returns –1 when the end of the file is encountered. It throws an IOException when an error occurs. Thus, this version of read( ) is the same as the one used to read from the console.

When you are done with a file, you must close it by calling close( ). Its general form is shown here:

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

Closing a file releases the system resources allocated to the file, allowing them to be used by another file. Failure to close a file can result in “memory leaks” because of unused resources remaining allocated.

The following program uses read( ) to input and display the contents of a text file

```
import java.io.*;
public class ShowFile {
    public static void main(String[] args){
        int i;
        FileInputStream fin;
        try {
            //the file test.txt must exist in the specified disrectory
            fin = new FileInputStream("C:\\Users\\hp\\Desktop\\test.txt");
        }catch (FileNotFoundException e){
            System.out.println("File not found");
            return;
        }
        try{
            do{
```

```
            i = fin.read();
            if (i != -1) System.out.println((char)i);
        }while(i != -1);
    }catch(IOException e){
        System.out.println("Error reading file");
    }
    try{
        fin.close();
    }catch(IOException e){
        System.out.println("Error closing file");
    }    }    }
```

Writing to a File

To open a file for output, create a FileOutputStream object. Here are two commonly used constructors:

```
FileOutputStream(String fileName) throws FileNotFoundException
FileOutputStream(String fileName, boolean append)
throws FileNotFoundException
```

If the file cannot be created, then FileNotFoundException is thrown. In the first form, when an output file is opened, any preexisting file by the same name is destroyed. In the second form, if append is true, then output is appended to the end of the file. Otherwise, the file is overwritten.

To write to a file, you will use the write( ) method. Its simplest form is shown here:

```
void write(int byteval) throws IOException
```

This method writes the byte specified by byteval to the file. Although byteval is declared as an integer, only the low-order 8 bits are written to the file. If an error occurs during writing, an IOException is thrown.

Once you are done with an output file, you must close it using close( ), shown here:

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

The following example copies a text file.

```
package file;
import java.io.*;
public class CopyFile {
    public static void main(String[] args)throws IOException{
        int i;
        FileInputStream fin = null;
        FileOutputStream fout = null;
        try{
            fin = new FileInputStream("in.txt");
            fout = new FileOutputStream("out.txt");
            do{
                i = fin.read();
                if(i != -1) fout.write(i);
            }while(i != -1);
        }catch (IOException exc){
            System.out.println("I/O exception " + exc);
        }finally{
            try{
                if(fin != null) fin.close();
            }catch(IOException e ){
                System.out.println("error closing input");
            }
            try {
                if (fout != null)fout.close();
            }catch(IOException e){
                System.out.println("Error closing output file");
            }       }    }    }
```

Using Java's Character-Based Streams

Java's byte streams are both powerful and flexible. However, they are not the ideal way to handle character-based I/O. For this purpose, Java defines the character stream classes. At the top of the character stream hierarchy are the abstract classes **Reader** and **Writer**. The following tables show the character stream I/O classes, and the methods in Reader and Writer. Most of the methods can throw an IOException on error. The methods defined by these two abstract classes are available to all of their subclasses. Thus, they form a minimal set of I/O functions that all character streams will have.

Character Stream class	Description
BufferedReader	Buffered input Character Stream
BufferedWriter	Buffered output Character Stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
FilterReader	Filter Reader
FilterWriter	Filter Writer
InputStreamReader	Input stream that translates byte to character
LineNumberReader	Input stream that counts lines
OutputStreamWriter	Output stream that translates byte to character
PipedReader	Input Pipe
PipedWriter	Output Pipe
PushbackReader	Input Stream that allows character to be returned to the input stream
Reader	Abstract class that describes character stream input
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream Output

Table 2: - The Character Stream I/O Classes

Method	Description
abstract void close()	Closes the input source. Further read attempts will generate an IOException.
Void mark(int numChars)	Places mark at the current point in the input stream that will remain valid until numChars characters are read.
boolean markSupprted	Returns true if mark() /reset are supported on this stream
int read()	Returns an integer representation of the next available character from the invoking input stream. -1 is returned when the end of stream is

	reached.
int read(char buffer[])	Attempts to read up to buffer.length characters into buffer and returns the actual number of characters that were successfully read. -1 is returned when the end of stream is reached.
abstract int read(char buffer[], int offset, int numChars)	Attempts to read up to numChars characters into buffer starting starting at buffer[offset], returning the actual number of characters that were successfully read. -1 is returned when the end of stream is reached.
int read(charBuffer buffer)	Attempts to fill the buffer specified by buffer, returning the actual number of characters that were successfully read. -1 is returned when the end of stream is reached.
Boolean ready()	Returns true if the next input request will not wait. Otherwise, it returns false.
Void reset()	Resets the input pointer to the previously set mark.
Long skip (long numChars)	Skips the numChar characters of input, returning the number of characters actually skipped.

Table 2: - The Methods Defined by Reader

Method	Description
Writer append(char ch)	Appends ch to the end of the invoking output stream. returns a reference to the invoking stream.
Writer append (charSequence chars)	Appends chars to the end of the invoking output stream. returns a reference to the invoking stream. charSequence an interface that defines read-only operations on a sequence of characters.
Writer append (charSequence chars, int begin, int end)	Appends the sequence of chars starting at begin and stopping with end to the end of the invoking output stream. charSequence is an interface that defines reada-only operation in a sequence of characters.
abstract void close()	Closes the output stream. further write attempt will generate in IOException.
abstract void flush()	Causes any output that has been buffered to be sent to its destination. i.e., it flushes the output buffer.
Void write (int ch)	Write a single character to the invoking utput stream. The Parameter is an int, which allows you to call write () with expression without having to cast them back to char.
Void write (char buffer[])	Writes a complete array of characters to the invoking output stream.
abstract void write (char buffer[], int offset, int numChars)	Writes a sub range of numChars characters from the array buffer characters from the array buffer, beginning at buffer [offset], to the invoking output stream.
Void write(String str)	Writes str to the invoking output stream.
Void write(String str, int offset, int numChars)	Writes a subrange of numChars characters from the array str, beginning at the specified offset.

Table 3: - The methods defined by writer

Console Input Using Character Streams

For code that will be internationalized, inputting from the console using Java's character-based streams is a better, more convenient way to read characters from the keyboard than is using the byte streams. However, since System.in is a byte stream, we will need to wrap System.in inside some type of Reader. The best class for reading console input is BufferedReader, which supports a buffered input stream. However, you cannot construct a BufferedReader directly from System.in. Instead, you must first convert it into a character stream. To do this, you will use InputStreamReader, which converts bytes to characters. To obtain an InputStreamReader object that is linked to System.in, use the constructor shown next:

```
InputStreamReader(InputStream inputStream)
```

Since System.in refers to an object of type InputStream, it can be used for inputStream.

Next, using the object produced by InputStreamReader, construct a BufferedReader using the constructor shown here:

```
BufferedReader(Reader inputReader)
```

Here, inputReader is the stream that is linked to the instance of BufferedReader being created. 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 will be a character-based stream that is linked to the console through System.in.

Reading Characters

Characters can be read from System.in using the read( ) method defined by BufferedReader in much the same way as they were read using byte streams. Here are three versions of read( ) supported by BufferedReader.

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

```
int read(char data[ ]) throws IOException
int read(char data[ ], int start, int max) throws IOException
```

The first version of read ( ) reads a single Unicode character. It returns –1 when the end of the stream is reached. The second version reads characters from the input stream and puts them into data until the array is full, the end of stream is reached, or an error occurs. It returns the number of characters read or –1 at the end of the stream. The third version reads input into data beginning at the location specified by start. Up to max characters are stored. It returns the number of characters read or –1 when the end of the stream is encountered. All throw an IOException on error. When reading from System.in, pressing ENTER generates an end-of-stream condition.

The following program demonstrates read ( ) by reading characters from the console until the user types a period. Notice that any I/O exceptions that might be generated are simply thrown out of main ( ). As mentioned earlier in this chapter, such an approach is common when reading from the console. Of course, you can handle these types of errors under program control, if you choose.

```
import java.io.*;
public class BufferedReaderDemo {
  public static void main(String [] args)  throws IOException{
      String d;
      BufferedReader        bf        =        new        BufferedReader(new
InputStreamReader(System.in));
System.out.println("Enter some character, (.) to exit");
      do{
          d =(char) br.read();
System.out.printnl (C);
      }while(C != '.'));
  }   }
```

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

It returns a String object that contains the characters read. It returns null if an attempt is made to read when at the end of the stream. The following program demonstrates BufferedReader and the readLine( ) method. The program reads and displays lines of text until you enter the word “stop”.

```
import java.io.*;
public class BufferedReaderDemo {
public static vod main(String [] args) throws IOException {
BufferedReader bf = new BufferedReader(new InputStreamReader(System.in));
String st;
System.out.printnl (“Enter some line of text, stop to exit”);
do{
st = bf.readLine();
System.out.printnl (st);
}while(st.equal(“stop”));
}     }
```

Console Output Using Character Streams

While it is still permissible to use System.out to write to the console under Java, its use is recommended mostly for debugging purposes or for sample programs. For real-world programs, the preferred method of writing to the console when using Java is through a PrintWriter stream. PrintWriter is one of the character-based classes. As explained, 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 flushingOn)
```

Here, outputStream is an object of type OutputStream and flushingOn controls whether Java flushes the output stream every time a println( ) method (among others) is called. If flushingOn is true, flushing automatically takes place. If false, flushing is not automatic.

To write to the console using a PrintWriter, specify System.out for the output stream and flush the stream after each call to println( ). For example, this line of code creates a PrintWriter that is connected to console output.

The following program illustrates using a PrintWriter to handle console output.

```
import java.io.*;
public class PrintWriterDemo{
```

```
public static void main(String [] args) throws IOException {
PrintWriter pw = new PrintWriter(System.out, true);
int a;
double b = 23.54;
pw.println(a);
pw.println(b);
pw.println(a + " + " + b + " = " + (a+b));
}     }
```

Remember that 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.

File I/O Using Character Streams

Although byte-oriented file handling is the most common, it is possible to use character-based streams for this purpose. The advantage to the character streams is that they operate directly on Unicode characters. Thus, if you want to store Unicode text, the character streams are certainly your best option. In general, to perform character-based file I/O, you will use the FileReader and FileWriter classes.

Using a FileWriter

FileWriter creates a Writer that you can use to write to a file. Two commonly used constructors are:

```
FileWriter(String fileName) throws IOException
FileWriter(String fileName, boolean append) throws IOException
```

Here, fileName is the full path name of a file. If append is true, then output is appended to the end of the file. Otherwise, the file is overwritten. Either throws an IOException on failure. FileWriter is derived from OutputStreamWriter and Writer. Thus, it has access to the methods defined by these classes.

Here is a simple key-to-disk program that reads lines of text entered at the keyboard and writes them to a file called "test.txt". Text is read until the user enters the word "stop". It uses a FileWriter to output to the file.

```
import java.io.*;
public class charOp {
    public static void main(String[] args){
        String s;
        BufferedReader        br        =        new        BufferedReader(new
InputStreamReader(System.in));
        System.out.println("enter some text");
        try        (FileWriter        fw        =        new        FileWriter
("C:\\Users\\hp\\Desktop\\test.txt", true);) {
            do{
            s = br.readLine();
            if(s.compareTo("stop") == 0)break;
            fw.write(s);
            }while (s.compareTo("stop") != 0);
            }catch(IOException e){
                System.out.println("Error" + e);}
       }        }
```

Using a FileReader

The FileReader class creates a Reader that you can use to read the contents of a file. A commonly used constructor is shown here:

FileReader(String fileName) throws FileNotFoundException

Here, fileName is the full path name of a file. It throws a FileNotFoundException if the file does not exist. FileReader is derived from InputStreamReader and Reader. Thus, it has access to the methods defined by these classes. The following program creates a simple disk-to-screen utility that reads a text file called "test.txt" and displays its contents on the screen.

```
import java.io.*;
public class charOp {
```

```
    public static void main(String[] args){
String s;
try(BufferedReader br = new BufferedReader(new FileReader(“test.txt”)))
      {
while((s = br.readLine()) != null)  {
System.out.println(s);
}
}catch(IOException exc){System.out.println(“Error: “ + exc);}     }     }
```

In this example, the FileReader is wrapped in a BufferedReader. This gives it access to readLine( ). Also, closing the BufferedReader, br in this case, automatically closes the file.