10.3. Java input and output basics in java.io package

I/O Basics

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 either graphically oriented programs that rely on Java's Abstract Window Toolkit (AWT) or Swing for user interaction, or they are Web applications. Although text-based, console 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 that useful in real-world 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. In fact, once you understand its fundamentals, the rest of the I/O system is easy to master. A general overview of I/O is presented here.

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.

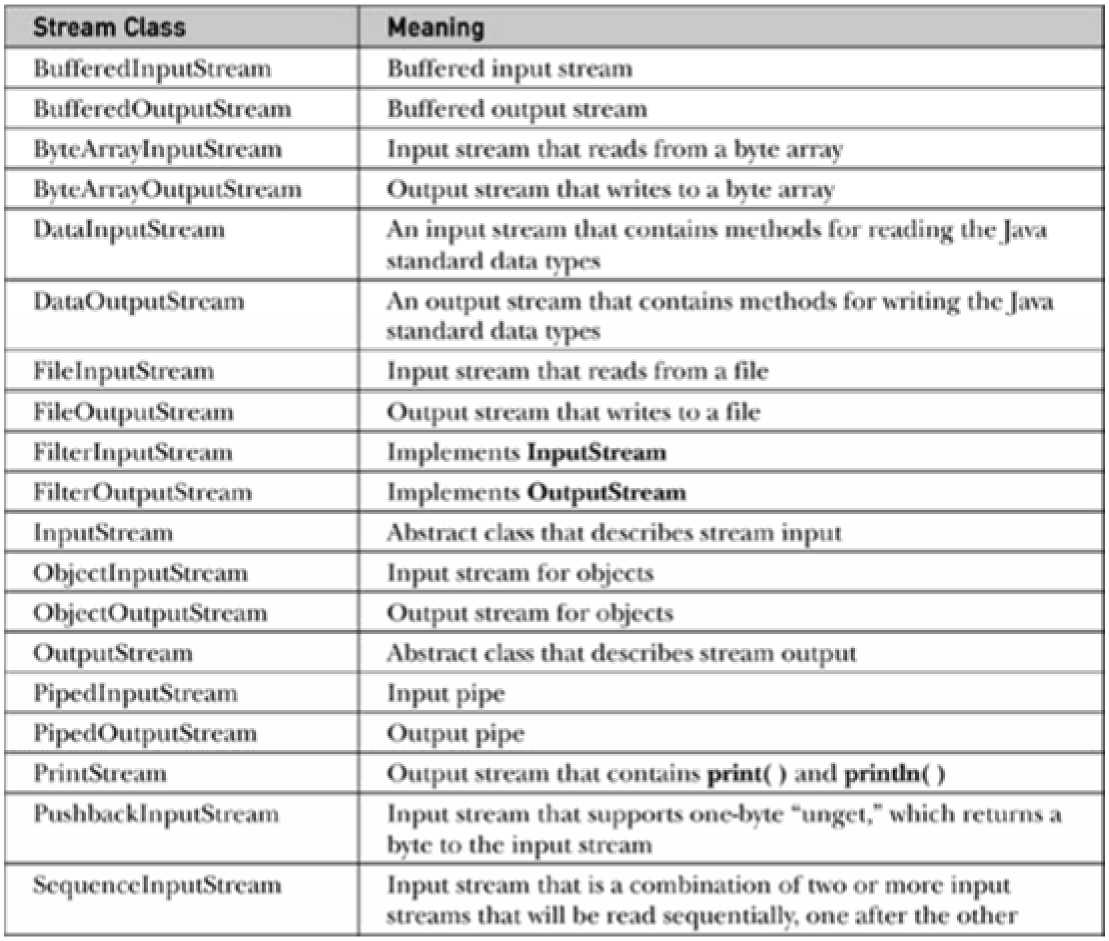
The original version of Java (Java 1.0) did not include character streams and, thus, all I/O was byte-oriented. Character streams were added by Java 1.1, and certain byte-oriented classes and methods were deprecated. Although old code that doesn't use character streams is becoming increasingly rare, it may still be encountered from time to time. As a general rule, old code should be updated to take advantage of character streams 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 among various devices, such as disk files, network connections, and even memory buffers. The byte stream classes in **java.io** are shown in Table 10-3-1. A few of these classes are discussed later in this section



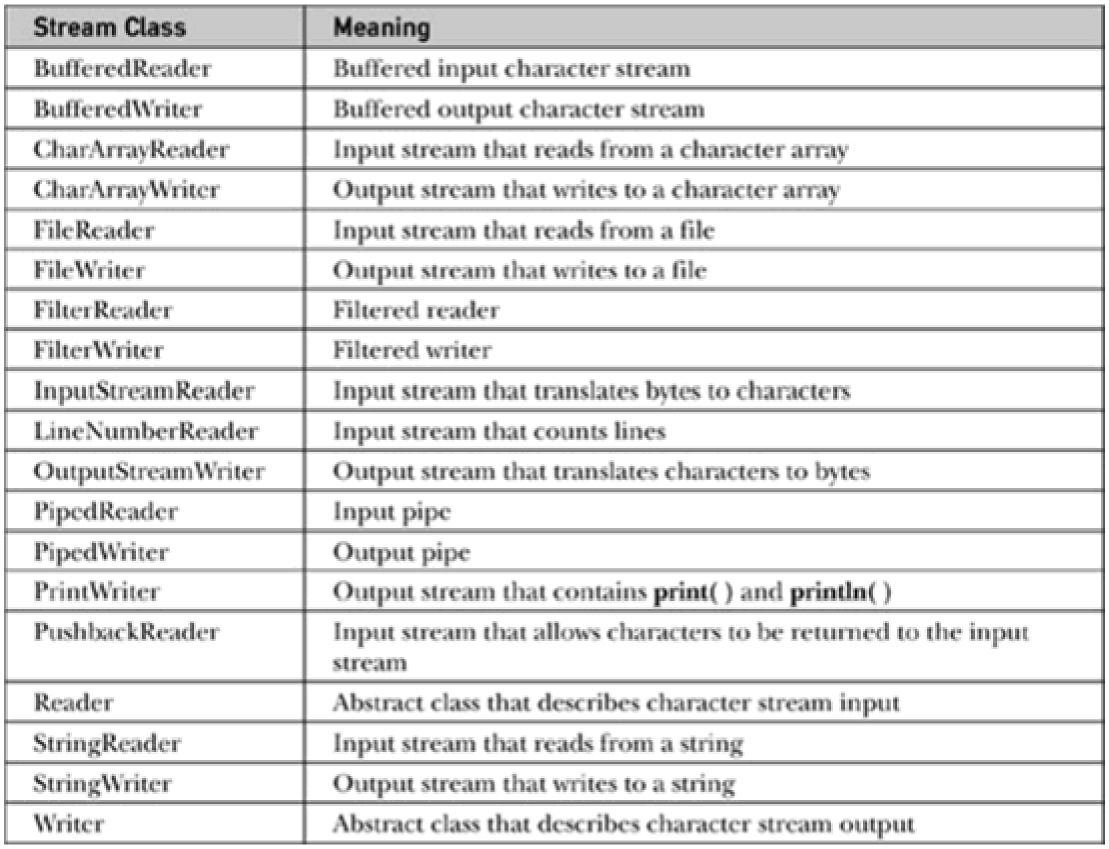
Stream Class	Meaning
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An input stream that contains methods for reading the 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 **InputStream**
FilterOutputStream	Implements **OutputStream**
InputStream	Abstract class that describes stream input
ObjectInputStream	Input stream for objects
ObjectOutputStream	Output stream for objects
OutputStream	Abstract class that describes stream output
PipedInputStream	Input pipe
PipedOutputStream	Output pipe
PrintStream	Output stream that contains **print( )** and **println( )**
PushbackInputStream	Input stream that supports one-byte "unget," which returns a byte to the input stream
SequenceInputStream	Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

Table 10-3-1 The Byte Stream Classes in **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. Each has forms that are abstract and must be 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 in **java.io** are shown in Table 10-3-2.



Stream Class	Meaning
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	Filtered reader
FilterWriter	Filtered writer
InputStreamReader	Input stream that translates bytes to characters
LineNumberReader	Input stream that counts lines
OutputStreamWriter	Output stream that translates characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output stream that contains **print( )** and **println( )**
PushbackReader	Input stream that allows characters 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 10-3-2 The Character Stream I/O Classes in **java.io**

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. Each has forms that are abstract and must be overridden by derived stream 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. For example, using some of its methods, you can obtain the current time and the settings of various properties associated with the system. **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 are typically used to read and write characters from and to the console. As you will see, you can wrap these within character-based streams, if desired.

The preceding chapters have been using **System.out** in their examples. You can use **System.err** in much the same way. As explained in the next section, use of **System.in** is a little more complicated.

Reading Console Input

In Java 1.0, the only way to perform console input was to use a byte stream. Today, using a byte stream to read console input is still acceptable. However, for commercial applications, the preferred method of reading console input is to use a character-oriented stream. This makes your program easier to internationalize and maintain.

In Java, console input is accomplished by reading from **System.in**. To obtain a character-based stream that is attached to the console, wrap **System.in** in a **BufferedReader** object. **BufferedReader** supports a buffered input stream. A 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 in simple example programs such as those shown in this book, but in more sophisticated applications, you can handle the exceptions explicitly.

```
// Use a BufferedReader to read characters from the console.
import java.io.*;

class BRRead {
  public static void main(String args[]) throws IOException
  {
    char c;
    BufferedReader br = new
            BufferedReader(new InputStreamReader(System.in));
    System.out.println("Enter characters, 'q' to quit.");
    // read characters
    do  {
       c = (char) br.read();
       System.out.println(c);
     } while(c != 'q');
  }
}
```

Here is a sample run:

```
Enter characters, 'q' to quit.
123abcq
1
2
3
a
b
c
q
```

This output may look a little different from what you expected because **System.in** is line buffered, by default. This means that no input is actually passed to the program until you press

ENTER. As you can guess, this does not make **read( )** particularly valuable for interactive console input.

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"));

  }

}
```

The next example creates a tiny text editor. It creates an array of **String** objects and then reads in lines of text, storing each line in the array. It will read up to 100 lines or until you enter “stop.” It uses a **BufferedReader** to read from the console.

```
// A tiny editor.

import java.io.*;
```

```
class TinyEdit {
  public static void main(String args[]) throws IOException
  {
    // create a BufferedReader using System.in
    BufferedReader br = new BufferedReader(new
                             InputStreamReader(System.in));
    String str[] = new String[100];
    System.out.println("Enter lines of text.");
    System.out.println("Enter 'stop' to quit.");
    for(int i=0; i<100; i++) {
       str[i] = br.readLine();
       if(str[i].equals("stop")) break;
    }
    System.out.println("\nHere is your file:");
    // display the lines
    for(int i=0; i<100; i++) {
      if(str[i].equals("stop")) break;
      System.out.println(str[i]);
    }
  }
}
```

Here is a sample run:

```
Enter lines of text.
Enter 'stop' to quit.
This is line one.
This is line two.
Java makes working with strings easy.
Just create String objects.
```

```
stop

Here is your file:

This is line one.

This is line two.

Java makes working with strings easy.

Just create String objects.
```

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 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');

  }

}
```

You will not often use **write( )** to perform console output (although doing so might be useful in some situations) because **print( )** and **println( )** are substantially easier to use.

The PrintWriter Class

Although using **System.out** to write to the console is acceptable, its use is probably best 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 internationalizing your program easier.

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. Thus, you can use these methods in the same way as you used them 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** makes 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.

Reading and Writing Files

Java provides a number of classes and methods that allow you to read and write files. Before we begin, it is important to state that the topic of file I/O is quite large and file I/O is examined in detail in Part II. The purpose of this section is to introduce the basic techniques that read from and write to a file. Although bytes streams are used, these techniques can be adapted to the character-based streams.

Two of the most often-used stream classes are **FileInputStream** and **FileOutputStream**, which create byte streams linked to files. To open a file, you simply create an object of one of these classes, specifying the name of the file as an argument to the constructor. Although both classes support additional constructors, the following are the forms that we will be using:

FileInputStream(String *fileName*) throws FileNotFoundException
FileOutputStream(String *fileName*) throws FileNotFoundException

Here, *fileName* specifies the name of the file that you want to open. When you create an input stream, if the file does not exist, then **FileNotFoundException** is thrown. For output streams, if the file cannot be opened or created, then **FileNotFoundException** is thrown. **FileNotFoundException** is a subclass of **IOException**. When an output file is opened, any preexisting file by the same name is destroyed.

NOTE

In situations in which a security manager is present, several of the file classes, including **FileInputStream** and **FileOutputStream**, will throw a **SecurityException** if a security violation occurs when attempting to open a file. By default, applications run via **java** do not use a security manager. For that reason, the I/O examples in this book do not need to watch for a possible **SecurityException**. However, other types of applications (such as applets) will use the security manager, and file I/O performed by such an application could generate a **SecurityException**. In that case, you will need to appropriately handle this exception.

When you are done with a file, you must close it. This is done by calling the **close( )** method, which is implemented by both **FileInputStream** and **FileOutputStream**. It 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.

NOTE

Beginning with JDK 7, the **close( )** method is specified by the **AutoCloseable** interface in **java.lang**. **AutoCloseable** is inherited by the **Closeable** interface in **java.io**. Both interfaces are implemented by the stream classes, including **FileInputStream** and **FileOutputStream**.

Before moving on, it is important to point out that there are two basic approaches that you can use to close a file when you are done with it. The first is the traditional approach, in which **close( )** is called explicitly when the file is no longer needed. This is the approach used by all versions of Java prior to JDK 7 and is, therefore, found in all legacy code. The second is to use the new **try**-with-resources statement added by JDK 7, which automatically closes a file when it is no longer needed. In this approach, no explicit call to **close( )** is executed. Since there are millions of lines of pre-JDK 7 legacy code that are still being used and maintained, it is important that you know and understand the traditional approach. Therefore, we will begin with it. The new automated approach is described in the following section.

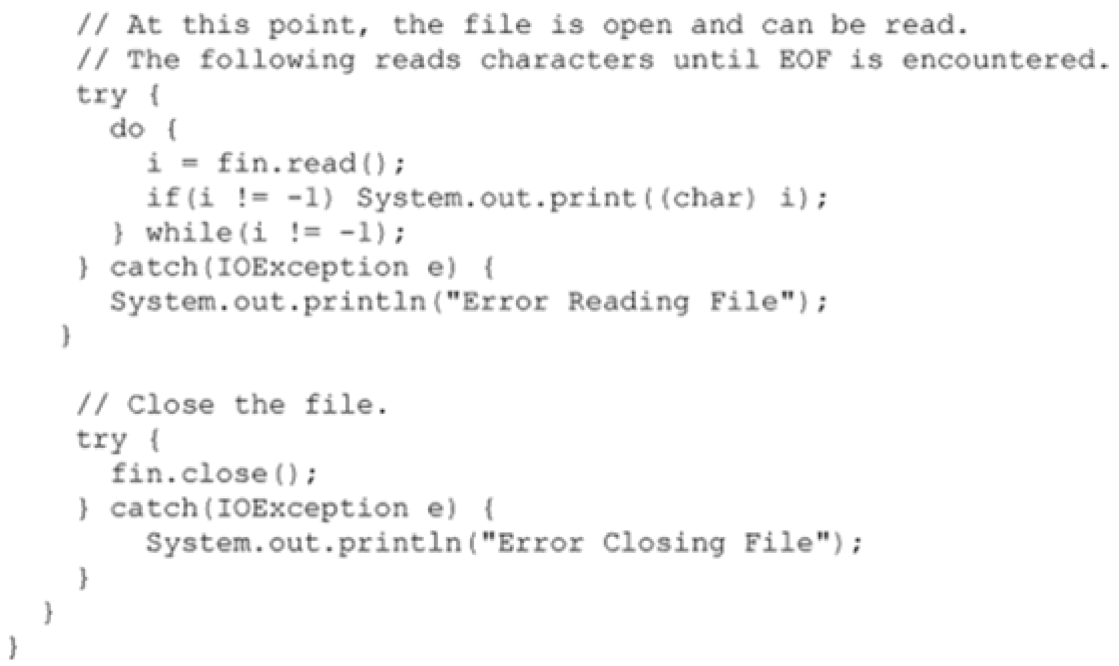
To read from a file, you can use a version of **read( )** that is defined within **FileInputStream**. The one that we will use is shown here:

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

Each time that it is called, it reads a single byte from the file and returns the byte as an integer value. **read( )** returns –1 when the end of the file is encountered. It can throw an **IOException**.

The following program uses **read( )** to input and display the contents of a file that contains ASCII text. The name of the file is specified as a command-line argument.

```
/* Display a text file.
   To use this program, specify the name
   of the file that you want to see.
   For example, to see a file called TEST.TXT,
   use the following command line.

   java ShowFile TEST.TXT
*/

import java.io.*;

class ShowFile {
  public static void main(String args[])
  {
    int i;
    FileInputStream fin;

    // First, confirm that a filename has been specified.
    if(args.length != 1) {
      System.out.println("Usage: ShowFile filename");
      return;
    }

    // Attempt to open the file.
    try {
      fin = new FileInputStream(args[0]);
    } catch(FileNotFoundException e) {
      System.out.println("Cannot Open File");
      return;
    }
```



```
    // At this point, the file is open and can be read.
    // The following reads characters until EOF is encountered.
    try {
      do {
        i = fin.read();
        if(i != -1) System.out.print((char) i);
      } while(i != -1);
    } catch(IOException e) {
      System.out.println("Error Reading File");
   }

    // Close the file.
    try {
      fin.close();
    } catch(IOException e) {
        System.out.println("Error Closing File");
    }
  }
}
```

In the program, notice the **try/catch** blocks that handle the I/O errors that might occur. Each I/O operation is monitored for exceptions, and if an exception occurs, it is handled. Be aware that in simple programs or example code, it is common to see I/O exceptions simply thrown out of **main( )**, as was done in the earlier console I/O examples. Also, in some real-world code, it can be helpful to let an exception propagate to a calling routine to let the caller know that an I/O operation failed. However, most of the file I/O examples in this book handle all I/O exceptions explicitly, as shown, for the sake of illustration.

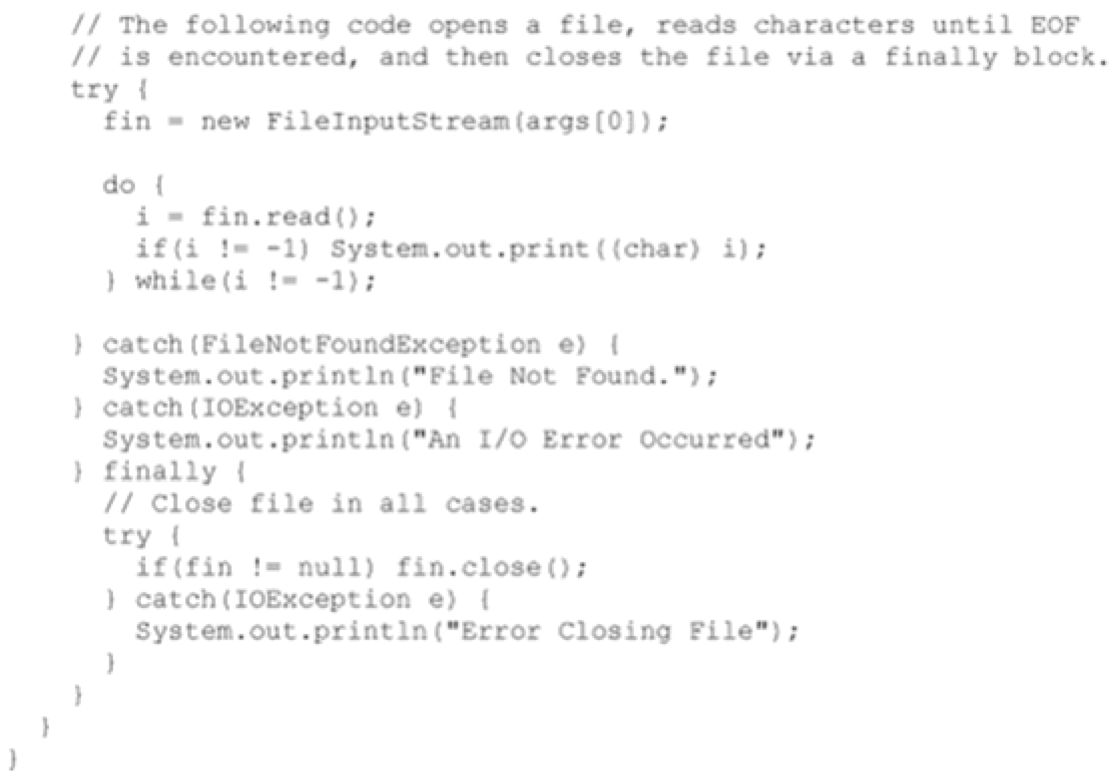
Although the preceding example closes the file stream after the file is read, there is a variation that is often useful. The variation is to call **close( )** within a **finally** block. In this approach, all of the methods that access the file are contained within a **try** block, and the **finally** block is used to close the file. This way, no matter how the **try** block terminates, the file is closed. Assuming the preceding example, here is how the **try** block that reads the file can be recoded:

```
try {
  do {
    i = fin.read();
    if(i != -1) System.out.print((char) i);
  } while(i != -1);
} catch(IOException e) {
  System.out.println("Error Reading File");
} finally {
  // Close file on the way out of the try block.
  try {
    fin.close();

  } catch(IOException e) {

    System.out.println("Error Closing File");

  }

}
```

Although not an issue in this case, one advantage to this approach in general is that if the code that accesses a file terminates because of some non-I/O related exception, the file is still closed by the **finally** block.

Sometimes it’s easier to wrap the portions of a program that open the file and access the file within a single **try** block (rather than separating the two) and then use a **finally** block to close the file. For example, here is another way to write the **ShowFile** program:

```
/* Display a text file.
   To use this program, specify the name
   of the file that you want to see.
   For example, to see a file called TEST.TXT,
   use the following command line.

   java ShowFile TEST.TXT

   This variation wraps the code that opens and
   accesses the file within a single try block.
   The file is closed by the finally block.
*/

import java.io.*;
class ShowFile {
  public static void main(String args[])
  {
    int i;
    FileInputStream fin = null;

    // First, confirm that a filename has been specified.
    if(args.length != 1) {
      System.out.println("Usage: ShowFile filename");
      return;
    }
```



```
    // The following code opens a file, reads characters until EOF
    // is encountered, and then closes the file via a finally block.
    try {
      fin = new FileInputStream(args[0]);

      do {
        i = fin.read();
        if(i != -1) System.out.print((char) i);
      } while(i != -1);

    } catch(FileNotFoundException e) {
      System.out.println("File Not Found.");
    } catch(IOException e) {
      System.out.println("An I/O Error Occurred");
    } finally {
      // Close file in all cases.
      try {
        if(fin != null) fin.close();
      } catch(IOException e) {
        System.out.println("Error Closing File");
      }
    }
  }
}
```

In this approach, notice that **fin** is initialized to **null**. Then, in the **finally** block, the file is closed only if **fin** is not **null**. This works because **fin** will be non-**null** only if the file is successfully opened. Thus, **close( )** is not called if an exception occurs while opening the file.

It is possible to make the **try/catch** sequence in the preceding example a bit more compact. Because **FileNotFoundException** is a subclass of **IOException**, it need not be caught separately. For example, here is the sequence recoded to eliminate catching **FileNotFoundException**. In this case, the standard exception message, which describes the error, is displayed.

```
try {
  fin = new FileInputStream(args[0]);

  do {
    i = fin.read();
    if(i != -1) System.out.print((char) i);
  } while(i != -1);

} catch(IOException e) {
  System.out.println("I/O Error: " + e);
} finally {

  // Close file in all cases.

  try {

    if(fin != null) fin.close();

  } catch(IOException e) {

    System.out.println("Error Closing File");

  }

}
```

In this approach, any error, including an error opening the file, is simply handled by the single **catch** statement. Because of its compactness, this approach is used by many of the I/O examples in this book. Be aware, however, that this approach is not appropriate in cases in which you want to deal separately with a failure to open a file, such as might be caused if a user mistypes a filename. In such a situation, you might want to prompt for the correct name, for example, before entering a **try** block that accesses the file.

To write to a file, you can use the **write( )** method defined by **FileOutputStream**. 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 eight bits are written to the file. If an error occurs during writing, an **IOException** is thrown. The next example uses **write( )** to copy a file:

```
/* Copy a file.
   To use this program, specify the name
   of the source file and the destination file.
   For example, to copy a file called FIRST.TXT
   to a file called SECOND.TXT, use the following
   command line.

   java CopyFile FIRST.TXT SECOND.TXT
*/

import java.io.*;

class CopyFile {
  public static void main(String args[]) throws IOException
  {
    int i;
    FileInputStream fin = null;
    FileOutputStream fout = null;

    // First, confirm that both files have been specified.
    if(args.length != 2) {
      System.out.println("Usage: CopyFile from to");
      return;
    }
```

```
      // Copy a File.
      try {
        // Attempt to open the files.
        fin = new FileInputStream(args[0]);
        fout = new FileOutputStream(args[1]);

        do {
          i = fin.read();
          if(i != -1) fout.write(i);
        } while(i != -1);

      } catch(IOException e) {
        System.out.println("I/O Error: " + e);
      } finally {
        try {
          if(fin != null) fin.close();
        } catch(IOException e2) {
          System.out.println("Error Closing Input File");
        }
        try {
          if(fout != null) fout.close();
        } catch(IOException e2) {
          System.out.println("Error Closing Output File");
        }
      }
    }
  }
```

In the program, notice that two separate **try** blocks are used when closing the files. This ensures that both files are closed, even if the call to **fin.close( )** throws an exception.

In general, notice that all potential I/O errors are handled in the preceding two programs by the use of exceptions. This differs from some computer languages that use error codes to report file errors. Not only do exceptions make file handling cleaner, but they also enable Java to easily differentiate the end-of-file condition from file errors when input is being performed. In C/C++, many input functions return the same value when an error occurs and when the end of the file is reached. (That is, in C/C++, an EOF condition often is mapped to the same value as an input error.) This usually means that the programmer must include extra program statements to determine which event actually occurred. In Java, input errors are passed to your program via exceptions, not by values returned by **read( )**. Thus, when **read( )** returns –1, it means only one thing: the end of the file has been encountered.

Automatically Closing a File

In the preceding section, the example programs have made explicit calls to **close( )** to close a file once it is no longer needed. As mentioned, this is the way files were closed when using versions of Java prior to JDK 7. Although this approach is still valid and useful, JDK 7 adds a new feature that offers another way to manage resources, such as file streams, by automating the closing process. This feature, sometimes referred to as *automatic resource management*, or *ARM* for short, is based on an expanded version of the **try** statement. The principal advantage of automatic resource management is that it prevents situations in which a file (or other resource) is inadvertently not released after it is no longer needed. As explained, forgetting to close a file can result in memory leaks, and could lead to other problems.

Automatic resource management is based on an expanded form of the **try** statement. Here is its general form:

```
try (resource-specification) {
  // use the resource
}
```

Here, *resource-specification* is a statement that declares and initializes a resource, such as a file stream. It consists of a variable declaration in which the variable is initialized with a reference to the object being managed. When the **try** block ends, the resource is automatically released. In the case of a file, this means that the file is automatically closed. (Thus, there is no need to call **close( )** explicitly.) Of course, this form of **try** can also include **catch** and **finally** clauses. This new form of **try** is called the ***try**-with-resources* statement.

The **try**-with-resources statement can be used only with those resources that implement the **AutoCloseable** interface defined by **java.lang**. This interface defines the **close( )** method. **AutoCloseable** is inherited by the **Closeable** interface in **java.io**. Both interfaces are implemented by the stream classes. Thus, **try**-with-resources can be used when working with streams, including file streams.

As a first example of automatically closing a file, here is a reworked version of the **ShowFile** program that uses it:

```
/* This version of the ShowFile program uses a try-with-resources

   statement to automatically close a file after it is no longer needed.



   Note: This code requires JDK 7 or later.

*/



import java.io.*;



class ShowFile {

  public static void main(String args[])

  {

    int i;
```

```
      // First, confirm that a filename has been specified.

      if(args.length != 1) {

        System.out.println("Usage: ShowFile filename");

        return;

      }


      // The following code uses a try-with-resources statement to open

      // a file and then automatically close it when the try block is left.

      try(FileInputStream fin = new FileInputStream(args[0])) {


        do {

          i = fin.read();

          if(i != -1) System.out.print((char) i);

        } while(i != -1);


      }   catch(FileNotFoundException e) {

          System.out.println("File Not Found.");

      }   catch(IOException e) {

          System.out.println("An I/O Error Occurred");

      }


    }

  }
```

In the program, pay special attention to how the file is opened within the **try** statement:

```
try(FileInputStream fin = new FileInputStream(args[0])) {
```

Notice how the resource-specification portion of the **try** declares a **FileInputStream** called **fin**, which is then assigned a reference to the file opened by its constructor. Thus, in this version of the program, the variable **fin** is local to the **try** block, being created when the **try** is entered. When the **try** is left, the stream associated with **fin** is automatically closed by an implicit call to **close( )**. You don’t need to call **close( )** explicitly, which means that you can’t forget to close the file. This is a key advantage of using **try**-with-resources.

It is important to understand that the resource declared in the **try** statement is implicitly **final**. This means that you can’t assign to the resource after it has been created. Also, the scope of the resource is limited to the **try**-with-resources statement.

You can manage more than one resource within a single **try** statement. To do so, simply separate each resource specification with a semicolon. The following program shows an example. It reworks the **CopyFile** program shown earlier so that it uses a single **try**-with-resources statement to manage both **fin** and **fout**.

```
/* A version of CopyFile that uses try-with-resources.

   It demonstrates two resources (in this case files) being

   managed by a single try statement.

*/


import java.io.*;


class CopyFile {

  public static void main(String args[]) throws IOException

  {

    int i;


    // First, confirm that both files have been specified.

    if(args.length != 2) {

      System.out.println("Usage: CopyFile from to");

      return;

    }


    // Open and manage two files via the try statement.

    try (FileInputStream fin = new FileInputStream(args[0]);

         FileOutputStream fout = new FileOutputStream(args[1]))

    {


      do  {
         i = fin.read();

         if(i != -1) fout.write(i);

      }  while(i != -1);


    } catch(IOException e) {

      System.out.println("I/O Error: " + e);

    }

  }

}
```

In this program, notice how the input and output files are opened within the **try** block:

```
try (FileInputStream fin = new FileInputStream(args[0]);

     FileOutputStream fout = new FileOutputStream(args[1]))

{

  // …
```

After this **try** block ends, both **fin** and **fout** will have been closed. If you compare this version of the program to the previous version, you will see that it is much shorter. The ability to streamline source code is a side-benefit of automatic resource management.

There is one other aspect to **try**-with-resources that needs to be mentioned. In general, when a **try** block executes, it is possible that an exception inside the **try** block will lead to another exception that occurs when the resource is closed in a **finally** clause. In the case of a "normal" **try** statement, the original exception is lost, being preempted by the second exception. However, when using **try**-with-resources, the second exception is *suppressed*. It is not, however, lost. Instead, it is added to the list of suppressed exceptions associated with the first exception. The list of suppressed exceptions can be obtained by using the **getSuppressed( )** method defined by **Throwable**.

Because of the benefits that the **try**-with-resources statement offers, it will be used by many, but not all, of the example programs in this edition of this book. Some of the examples will still use the traditional approach to closing a resource. There are several reasons for this. First, there are millions of lines of legacy code in widespread use that rely on the traditional approach. It is important that all Java programmers be fully versed in, and comfortable with, the traditional approach when maintaining this older code. Second, because not all project development will immediately switch to a new version of the JDK, it is likely that some programmers will continue to work in a pre-JDK 7 environment for a period of time. In such situations, the expanded form of **try** is not available. Finally, there may be cases in which explicitly closing a resource is more appropriate than the automated approach. For these reasons, some of the examples in this book will continue to use the traditional approach, explicitly calling **close( )**. In addition to illustrating the traditional technique, these examples can also be compiled and run by all readers in all environments.