

The Byte Streams

The byte stream classes provide a rich environment for handling byte-oriented I/O. A byte stream can be used with any type of object, including binary data. This versatility makes byte streams important to many types of programs. Since the byte stream classes are topped by **InputStream** and **OutputStream**, our discussion begins with them.

InputStream

InputStream is an abstract class that defines Java's model of streaming byte input. It implements the **AutoCloseable** and **Closeable** interfaces. Most of the methods in this class will throw an **IOException** when an I/O error occurs. (The exceptions are **mark()** and **markSupported()**.) Table 20-1 shows the methods in **InputStream**.

NOTE Most of the methods described in Table 20-1 are implemented by the subclasses of **InputStream**. The **mark()** and **reset()** methods are exceptions; notice their use, or lack thereof, by each subclass in the discussions that follow.

OutputStream

OutputStream is an abstract class that defines streaming byte output. It implements the **AutoCloseable**, **Closeable**, and **Flushable** interfaces. Most of the methods defined by this class return **void** and throw an **IOException** in the case of I/O errors. Table 20-2 shows the methods in **OutputStream**.

Method	Description
<code>int available()</code>	Returns the number of bytes of input currently available for reading.
<code>void close()</code>	Closes the input source. Further read attempts will generate an IOException .
<code>void mark(int numBytes)</code>	Places a mark at the current point in the input stream that will remain valid until <i>numBytes</i> bytes are read.
<code>boolean markSupported()</code>	Returns true if mark() / reset() are supported by the invoking stream.
<code>int read()</code>	Returns an integer representation of the next available byte of input. <code>-1</code> is returned when the end of the file is encountered.
<code>int read(byte buffer[])</code>	Attempts to read up to <i>buffer.length</i> bytes into <i>buffer</i> and returns the actual number of bytes that were successfully read. <code>-1</code> is returned when the end of the file is encountered.
<code>int read(byte buffer[], int offset, int numBytes)</code>	Attempts to read up to <i>numBytes</i> bytes into <i>buffer</i> starting at <i>buffer[offset]</i> , returning the number of bytes successfully read. <code>-1</code> is returned when the end of the file is encountered.
<code>void reset()</code>	Resets the input pointer to the previously set mark.
<code>long skip(long numBytes)</code>	Ignores (that is, skips) <i>numBytes</i> bytes of input, returning the number of bytes actually ignored.

Table 20-1 The Methods Defined by **InputStream**

Method	Description
<code>void close()</code>	Closes the output stream. Further write attempts will generate an IOException .
<code>void flush()</code>	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
<code>void write(int b)</code>	Writes a single byte to an output stream. Note that the parameter is an int , which allows you to call write() with an expression without having to cast it back to byte .
<code>void write(byte buffer[])</code>	Writes a complete array of bytes to an output stream.
<code>void write(byte buffer[], int offset, int numBytes)</code>	Writes a subrange of <i>numBytes</i> bytes from the array <i>buffer</i> , beginning at <i>buffer[offset]</i> .

Table 20-2 The Methods Defined by **OutputStream**

FileInputStream

The **FileInputStream** class creates an **InputStream** that you can use to read bytes from a file. Two commonly used constructors are shown here:

```
FileInputStream(String filePath)
FileInputStream(File fileObj)
```

Either can throw a **FileNotFoundException**. Here, *filePath* is the full path name of a file, and *fileObj* is a **File** object that describes the file.

The following example creates two **FileInputStreams** that use the same disk file and each of the two constructors:

```
FileInputStream f0 = new FileInputStream("/autoexec.bat")
File f = new File("/autoexec.bat");
FileInputStream f1 = new FileInputStream(f);
```

Although the first constructor is probably more commonly used, the second allows you to closely examine the file using the **File** methods, before attaching it to an input stream. When a **FileInputStream** is created, it is also opened for reading. **FileInputStream** overrides six of the methods in the abstract class **InputStream**. The **mark()** and **reset()** methods are not overridden, and any attempt to use **reset()** on a **FileInputStream** will generate an **IOException**.

The next example shows how to read a single byte, an array of bytes, and a subrange of an array of bytes. It also illustrates how to use **available()** to determine the number of bytes remaining and how to use the **skip()** method to skip over unwanted bytes. The program reads its own source file, which must be in the current directory. Notice that it uses the **try-with-resources** statement to automatically close the file when it is no longer needed.

```
// Demonstrate FileInputStream.
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;
```

```

class FileInputStreamDemo {
    public static void main(String args[]) {
        int size;

        // Use try-with-resources to close the stream.
        try ( FileInputStream f =
            new FileInputStream("FileInputStreamDemo.java") ) {

            System.out.println("Total Available Bytes: " +
                (size = f.available()));

            int n = size/40;
            System.out.println("First " + n +
                " bytes of the file one read() at a time");
            for (int i=0; i < n; i++) {
                System.out.print((char) f.read());
            }

            System.out.println("\nStill Available: " + f.available());

            System.out.println("Reading the next " + n +
                " with one read(b[])");
            byte b[] = new byte[n];
            if (f.read(b) != n) {
                System.err.println("couldn't read " + n + " bytes.");
            }

            System.out.println(new String(b, 0, n));
            System.out.println("\nStill Available: " + (size = f.available()));
            System.out.println("Skipping half of remaining bytes with skip()");
            f.skip(size/2);
            System.out.println("Still Available: " + f.available());

            System.out.println("Reading " + n/2 + " into the end of array");
            if (f.read(b, n/2, n/2) != n/2) {
                System.err.println("couldn't read " + n/2 + " bytes.");
            }

            System.out.println(new String(b, 0, b.length));
            System.out.println("\nStill Available: " + f.available());
        } catch (IOException e) {
            System.out.println("I/O Error: " + e);
        }
    }
}

```

Here is the output produced by this program:

```

Total Available Bytes: 1785
First 44 bytes of the file one read() at a time
// Demonstrate FileInputStream.
// This pr
Still Available: 1741

```

```
Reading the next 44 with one read(b[])
ogram uses try-with-resources. It requires J
```

```
Still Available: 1697
Skipping half of remaining bytes with skip()
Still Available: 849
Reading 22 into the end of array
ogram uses try-with-rebyte[n];
    if (
```

```
Still Available: 827
```

This somewhat contrived example demonstrates how to read three ways, to skip input, and to inspect the amount of data available on a stream.

NOTE The preceding example and the other examples in this chapter handle any I/O exceptions that might occur as described in Chapter 13. See Chapter 13 for details and alternatives.

FileOutputStream

FileOutputStream creates an **OutputStream** that you can use to write bytes to a file. It implements the **AutoCloseable**, **Closeable**, and **Flushable** interfaces. Four of its constructors are shown here:

```
FileOutputStream(String filePath)
FileOutputStream(File fileObj)
FileOutputStream(String filePath, boolean append)
FileOutputStream(File fileObj, boolean append)
```

They can throw a **FileNotFoundException**. Here, *filePath* is the full path name of a file, and *fileObj* is a **File** object that describes the file. If *append* is **true**, the file is opened in append mode.

Creation of a **FileOutputStream** is not dependent on the file already existing.

FileOutputStream will create the file before opening it for output when you create the object. In the case where you attempt to open a read-only file, an exception will be thrown.

The following example creates a sample buffer of bytes by first making a **String** and then using the **getBytes()** method to extract the byte array equivalent. It then creates three files. The first, **file1.txt**, will contain every other byte from the sample. The second, **file2.txt**, will contain the entire set of bytes. The third and last, **file3.txt**, will contain only the last quarter.

```
// Demonstrate FileOutputStream.
// This program uses the traditional approach to closing a file.

import java.io.*;

class FileOutputStreamDemo {
    public static void main(String args[]) {
        String source = "Now is the time for all good men\n"
            + "to come to the aid of their country\n"
            + "and pay their due taxes.";
    }
}
```

```

byte buf[] = source.getBytes();
FileOutputStream f0 = null;
FileOutputStream f1 = null;
FileOutputStream f2 = null;

try {
    f0 = new FileOutputStream("file1.txt");
    f1 = new FileOutputStream("file2.txt");
    f2 = new FileOutputStream("file3.txt");

    // write to first file
    for (int i=0; i < buf.length; i += 2) f0.write(buf[i]);

    // write to second file
    f1.write(buf);

    // write to third file
    f2.write(buf, buf.length-buf.length/4, buf.length/4);
} catch(IOException e) {
    System.out.println("An I/O Error Occurred");
} finally {
    try {
        if(f0 != null) f0.close();
    } catch(IOException e) {
        System.out.println("Error Closing file1.txt");
    }
    try {
        if(f1 != null) f1.close();
    } catch(IOException e) {
        System.out.println("Error Closing file2.txt");
    }
    try {
        if(f2 != null) f2.close();
    } catch(IOException e) {
        System.out.println("Error Closing file3.txt");
    }
}
}
}

```

Here are the contents of each file after running this program. First, **file1.txt**:

```

Nwi h iefralgo e
t oet h i ftercuty n a hi u ae.

```

Next, **file2.txt**:

```

Now is the time for all good men
to come to the aid of their country
and pay their due taxes.

```

Finally, **file3.txt**:

```

nd pay their due taxes.

```

As the comment at the top of the program states, the preceding program shows an example that uses the traditional approach to closing a file when it is no longer needed. This approach is required by all versions of Java prior to JDK 7 and is widely used in legacy code. As you can see, quite a bit of rather awkward code is required to explicitly call `close()` because each call could generate an **IOException** if the close operation fails. This program can be substantially improved by using the new **try-with-resources** statement. For comparison, here is the revised version. Notice that it is much shorter and streamlined:

```
// Demonstrate FileOutputStream.
// This version uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

class FileOutputStreamDemo {
    public static void main(String args[]) {
        String source = "Now is the time for all good men\n"
            + "to come to the aid of their country\n"
            + "and pay their due taxes.";
        byte buf[] = source.getBytes();

        // Use try-with-resources to close the files.
        try (FileOutputStream f0 = new FileOutputStream("file1.txt");
            FileOutputStream f1 = new FileOutputStream("file2.txt");
            FileOutputStream f2 = new FileOutputStream("file3.txt") )
        {

            // write to first file
            for (int i=0; i < buf.length; i += 2) f0.write(buf[i]);

            // write to second file
            f1.write(buf);

            // write to third file
            f2.write(buf, buf.length-buf.length/4, buf.length/4);
        } catch(IOException e) {
            System.out.println("An I/O Error Occurred");
        }
    }
}
```

ByteArrayInputStream

ByteArrayInputStream is an implementation of an input stream that uses a byte array as the source. This class has two constructors, each of which requires a byte array to provide the data source:

```
ByteArrayInputStream(byte array [ ])
ByteArrayInputStream(byte array [ ], int start, int numBytes)
```

Here, *array* is the input source. The second constructor creates an **InputStream** from a subset of the byte array that begins with the character at the index specified by *start* and is *numBytes* long.

The `close()` method has no effect on a **ByteArrayInputStream**. Therefore, it is not necessary to call `close()` on a **ByteArrayInputStream**, but doing so is not an error.