20.2 Byte Streams: Input Streams and Output Streams

The abstract classes `InputStream` and `OutputStream` in the `java.io` package are the root of the inheritance hierarchies for handling the reading and writing of data as *bytes* (Figure 20.1). Their subclasses, implementing different kinds of input and output (I/O) streams, override the following methods from the `InputStream` and `OutputStream` classes to customize the reading and writing of bytes, respectively:

From the `InputStream` abstract class:

```
int read() throws IOException
int read(byte[] b) throws IOException
int read(byte[] b, int off, int len) throws IOException
```

Note that the first `read()` method reads a *byte*, but returns an `int` value. The byte read resides in the eight least significant bits of the `int` value, while the remaining bits in the `int` value are zeroed out. The `read()` methods return the value –1 when the end of the input stream is reached.

```
long transferTo(OutputStream out) throws IOException
```

Reads bytes from this input stream and writes to the specified output stream in the order they are read. The I/O streams are *not closed* after the operation (see below).

From the `OutputStream` abstract class:

```
void write(int b) throws IOException
void write(byte[] b) throws IOException
void write(byte[] b, int off, int len) throws IOException
```

The first `write()` method takes an `int` as an argument, but truncates it to the eight least significant bits before writing it out as a byte to the output stream.

`void close() throws IOException`	Both `InputStream` and `OutputStream`
`void flush() throws IOException`	Only for `OutputStream`

A I/O stream should be *closed* when no longer needed, to free system resources. Closing an output stream automatically *flushes* the output stream, meaning that any data in its internal buffer is written out.

Since byte streams also implement the `AutoCloseable` interface, they can be declared in a `try`-with-resources statement (§7.7, p. 407) that will ensure they are properly closed after use at runtime.

An output stream can also be manually flushed by calling the second method.

Read and write operations on streams are *blocking* operations—that is, a call to a read or write method does not return before a byte has been read or written.

Many methods in the classes contained in the `java.io` package throw the checked `IOException`. A calling method must therefore either catch the exception explicitly, or specify it in a `throws` clause.

Figure 20.1 *Partial Byte Stream Inheritance Hierarchies in the* `java.io` *Package*

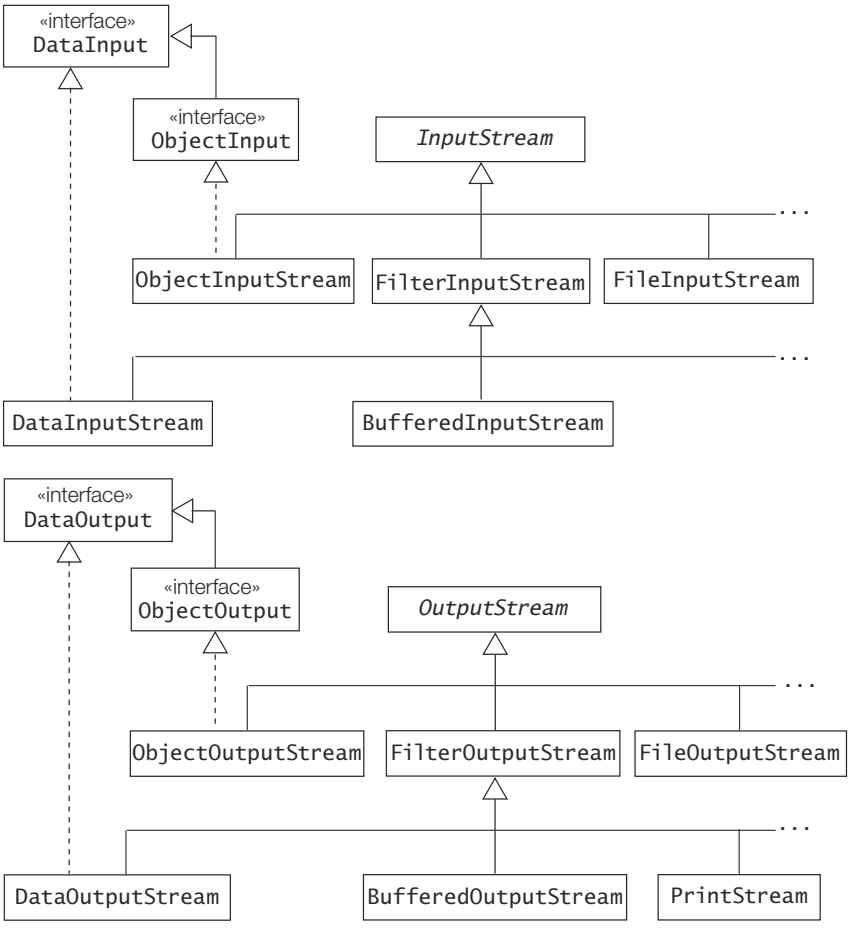


Table 20.1 and Table 20.2 give an overview of selected byte streams. Usually an output stream has a corresponding input stream of the same type.

Table 20.1 *Selected Input Streams*

`FileInputStream`	Data is read as bytes from a file. The file acting as the input stream can be specified by a `File` object, a `FileDescriptor`, or a `String` file name.
`FilterInputStream`	The superclass of all *input filter streams*. An input filter stream must be chained to an underlying input stream.
`DataInputStream`	A filter stream that allows the *binary representation of Java primitive values* to be read from an underlying input stream. The underlying input stream must be specified.
`ObjectInputStream`	A filter stream that allows *binary representations of Java objects and Java primitive values* to be read from a specified input stream.

Table 20.2 *Selected Output Streams*

`FileOutputStream`	Data is written as *bytes* to a file. The file acting as the output stream can be specified by a `File` object, a `FileDescriptor`, or a `String` file name.
`FilterOutputStream`	The superclass of all *output filter streams*. An output filter stream must be chained to an underlying output stream.
`PrintStream`	A filter output stream that converts a *text representation of Java objects and Java primitive values* to bytes before writing them to an underlying output stream, which must be specified. This is the type of `System.out` and `System.err` (p. 1255). However, the `PrintWriter` class is recommended when writing characters rather than bytes (p. 1247).
`DataOutputStream`	A filter stream that allows the *binary representation of Java primitive values* to be written to an underlying output stream. The underlying output stream must be specified.
`ObjectOutputStream`	A filter stream that allows the *binary representation of Java objects and Java primitive values* to be written to a specified underlying output stream.

File Streams

The subclasses `FileInputStream` and `FileOutputStream` represent low-level streams that define byte input and output streams that are connected to files. Data can only be read or written as a sequence of *bytes*. Such file streams are typically used for handling image data.

A `FileInputStream` for reading bytes can be created using the following constructor:

```
FileInputStream(String name) throws FileNotFoundException
```

The file designated by the file name is assigned to a new file input stream.

If the file does not exist, a `FileNotFoundException` is thrown. If it exists, it is set to be read from the beginning. A `SecurityException` is thrown if the file does not have read access.

A `FileOutputStream` for writing bytes can be created using the following constructor:

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

The file designated by the file name is assigned to a new file output stream.

If the file does not exist, it is created. If it exists, its contents are reset, unless the appropriate constructor is used to indicate that output should be appended to the file. A `SecurityException` is thrown if the file does not have write access or it cannot be created. A `FileNotFoundException` is thrown if it is not possible to open the file for any other reasons.

The `FileInputStream` class provides an implementation for the `read()` methods in its superclass `InputStream`. Similarly, the `FileOutputStream` class provides an implementation for the `write()` methods in its superclass `OutputStream`.

Example 20.1 demonstrates using a buffer to read bytes from and write bytes to file streams. The input and the output file names are specified on the command line. The streams are created at (1) and (2).

The bytes are read into a buffer by the `read()` method that returns the number of bytes read. The same number of bytes from the buffer are written to the output file by the `write()` method, regardless of whether the buffer is full or not after every read operation.

The end of file is reached when the `read()` method returns the value `-1`. The code at (3a) using a buffer can be replaced by a call to the `transferTo()` method at (3b) to do the same operation. The streams are closed by the `try`-with-resources statement. Note that most of the code consists of a `try`-with-resources statement with `catch` clauses to handle the various exceptions.

Example 20.1 *Copying a File Using a Byte Buffer*

```
/* Copy a file using a byte buffer.
   Command syntax: java CopyFile <from_file> <to_file> */
import java.io.*;

class CopyFile {
  public static void main(String[] args) {

    try (// Assign the files:
        FileInputStream fromFile = new FileInputStream(args[0]);       // (1)
        FileOutputStream toFile = new FileOutputStream(args[1]))  {    // (2)

      // Copy bytes using buffer:                                      // (3a)
      byte[] buffer = new byte[1024];
      int length = 0;
      while((length = fromFile.read(buffer)) != -1) {
        toFile.write(buffer, 0, length);
      }

      // Transfer bytes:
//    fromFile.transferTo(toFile);                                     // (3b)

    } catch(ArrayIndexOutOfBoundsException e) {
      System.err.println("Usage: java CopyFile <from_file> <to_file>");
    } catch(FileNotFoundException e) {
      System.err.println("File could not be copied: " + e);
    } catch(IOException e) {
      System.err.println("I/O error.");
    }
  }
}
```

I/O Filter Streams

An *I/O filter stream* is a high-level I/O stream that provides additional functionality to an underlying stream to which it is chained. The data from the underlying stream is manipulated in some way by the filter stream. The `FilterInputStream` and `FilterOutputStream` classes, together with their subclasses, define input and output filter streams. The subclasses `BufferedInputStream` and `BufferedOutputStream` implement filter streams that buffer input from and output to the underlying stream, respectively. The subclasses `DataInputStream` and `DataOutputStream` implement filter streams that allow binary representation of Java primitive values to be read and written, respectively, from and to an underlying stream.

Reading and Writing Binary Values

The `java.io` package contains the two interfaces `DataInput` and `DataOutput`, which streams can implement to allow reading and writing of *binary representation of Java primitive values* (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, `double`). The methods for writing binary representations of Java primitive values are named `writeX`, where *X* is any Java primitive data type. The methods for reading binary representations of Java primitive values are similarly named `readX`. Table 20.3 gives an overview of the `readX()` and `writeX()` methods found in these two interfaces. A file containing *binary values* (i.e., binary representation of Java primitive values) is usually called a *binary file*.

Note the methods provided for reading and writing strings. However, the recommended practice for reading and writing characters is to use *character streams*, called *readers* and *writers*, which are discussed in §20.3.

The filter streams `DataOutputStream` and `DataInputStream` implement the `DataOutput` and `DataInput` interfaces, respectively, and can be used to read and write binary representation of Java primitive values from and to an underlying stream. Both the `writeX()` and `readX()` methods throw an `IOException` in the event of an I/O error. In particular, the `readX()` methods throw an `EOFException` (a subclass of `IOException`) if the input stream does not contain the correct number of bytes to read. Bytes can also be skipped from a `DataInput` stream, using the `skipBytes(int n)` method which skips `n` bytes.

```
DataInputStream(InputStream in)
DataOutputStream(OutputStream out)
```

These constructors can be used to set up filter streams from an underlying stream for reading and writing Java primitive values, respectively.

Table 20.3 *The* `DataInput` *and* `DataOutput` *Interfaces*

Type	Methods in the `DataInput` interface	Methods in the `DataOutput` interface
`boolean`	`readBoolean()`	`writeBoolean(boolean b)`
`char`	`readChar()`	`writeChar(int c)`

Table 20.3 *The* `DataInput` *and* `DataOutput` *Interfaces (Continued)*

Type	Methods in the `DataInput` interface	Methods in the `DataOutput` interface
`byte`	`readByte()`	`writeByte(int b)`
`short`	`readShort()`	`writeShort(int s)`
`int`	`readInt()`	`writeInt(int i)`
`long`	`readLong()`	`writeLong(long l)`
`float`	`readFloat()`	`writeFloat(float f)`
`double`	`readDouble()`	`writeDouble(double d)`
`String`	`readLine()`	`writeChars(String str)`
`String`	`readUTF()`	`writeUTF(String str)`

Writing Binary Values to a File

To write the binary representation of Java primitive values to a *binary file*, the following procedure can be used, which is also depicted in Figure 20.2.

1. Use a `try`-with-resources statement for declaring and creating the necessary streams, which guarantees closing of the filter stream and any underlying stream.
2. Create a `FileOutputStream`:

   ```
   FileOutputStream outputFile = new FileOutputStream("primitives.data");
   ```

3. Create a `DataOutputStream` which is chained to the `FileOutputStream`:

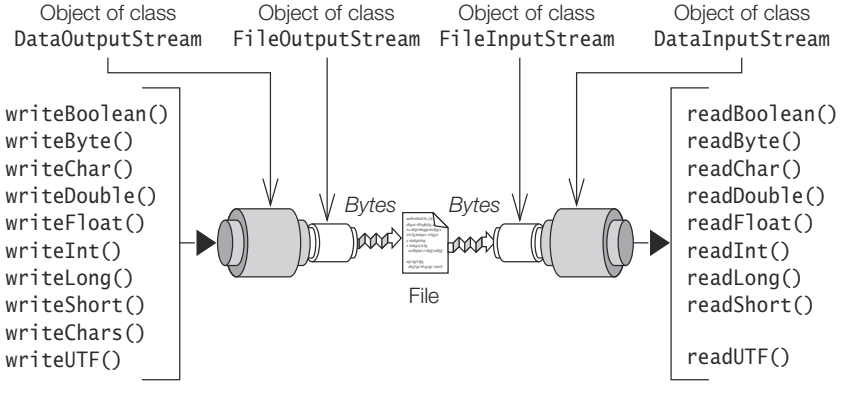
   ```
   DataOutputStream outputStream = new DataOutputStream(outputFile);
   ```

4. Write Java primitive values using relevant `write`*X*`()` methods:

   Note that in the case of `char`, `byte`, and `short` data types, the `int` argument to the `write`*X*`()` method is converted to the corresponding type, before it is written (see Table 20.3).

See also the numbered lines in Example 20.2 corresponding to the steps above.

Figure 20.2 *Stream Chaining for Reading and Writing Binary Values to a File*



Reading Binary Values from a File

To read the binary representation of Java primitive values from a *binary file*, the following procedure can be used, which is also depicted in Figure 20.2.

1. Use a `try`-with-resources statement for declaring and creating the necessary streams, which guarantees closing of the filter stream and any underlying stream.
2. Create a `FileInputStream`:
   ```
   FileInputStream inputFile = new FileInputStream("primitives.data");
   ```
3. Create a `DataInputStream` which is chained to the `FileInputStream`:
   ```
   DataInputStream inputStream = new DataInputStream(inputFile);
   ```
4. Read the (exact number of) Java primitive values in the *same order* they were written out to the file, using relevant `read`*X*`()` methods. Not doing so will unleash the wrath of the `IOException`.

See also the numbered lines in Example 20.2 corresponding to the steps above. Example 20.2 uses both procedures described above: first to write and then to read some Java primitive values to and from a file. It also checks to see if the end of the stream has been reached, signaled by an `EOFException`. The values are also written to the standard output stream.

Example 20.2 *Reading and Writing Binary Values*

```
import java.io.*;

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

    // Write binary values to a file:
    try(                                                                    // (1)
        // Create a FileOutputStream.                                       (2)
        FileOutputStream outputFile = new FileOutputStream("primitives.data");
        // Create a DataOutputStream which is chained to the FileOutputStream.(3)
        DataOutputStream outputStream = new DataOutputStream(outputFile)) {

      // Write Java primitive values in binary representation:              (4)
      outputStream.writeBoolean(true);
      outputStream.writeChar('A');              // int written as Unicode char
      outputStream.writeByte(Byte.MAX_VALUE);   // int written as 8-bits byte
      outputStream.writeShort(Short.MIN_VALUE); // int written as 16-bits short
      outputStream.writeInt(Integer.MAX_VALUE);
      outputStream.writeLong(Long.MIN_VALUE);
      outputStream.writeFloat(Float.MAX_VALUE);
      outputStream.writeDouble(Math.PI);
    }

    // Read binary values from a file:
    try (                                                                   // (1)
        // Create a FileInputStream.                                        (2)
        FileInputStream inputFile = new FileInputStream("primitives.data");
```

```
          // Create a DataInputStream which is chained to the FileInputStream.  (3)
          DataInputStream inputStream = new DataInputStream(inputFile)) {

        // Read the binary representation of Java primitive values
        // in the same order they were written out:                              (4)
        System.out.println(inputStream.readBoolean());
        System.out.println(inputStream.readChar());
        System.out.println(inputStream.readByte());
        System.out.println(inputStream.readShort());
        System.out.println(inputStream.readInt());
        System.out.println(inputStream.readLong());
        System.out.println(inputStream.readFloat());
        System.out.println(inputStream.readDouble());

        // Check for end of stream:
        int value = inputStream.readByte();
        System.out.println("More input: " + value);
      } catch (FileNotFoundException fnf) {
        System.out.println("File not found.");
      } catch (EOFException eof) {
        System.out.println("End of input stream.");
      }
    }
  }
```

Output from the program:

```
true
A
127
-32768
2147483647
-9223372036854775808
3.4028235E38
3.141592653589793
End of input stream.
```