15.7 (Optional) Additional `java.io` Classes

This section overviews additional interfaces and classes (from package `java.io`).

15.7.1 Interfaces and Classes for Byte-Based Input and Output

`InputStream` and `OutputStream` are `abstract` classes that declare methods for performing byte-based input and output, respectively.

Pipe Streams

Pipes are synchronized communication channels between threads. We discuss threads in Chapter 23. Java provides `PipedOutputStream` (a subclass of `OutputStream`) and `Piped-InputStream` (a subclass of `InputStream`) to establish pipes between two threads in a program. One thread sends data to another by writing to a `PipedOutputStream`. The target thread reads information from the pipe via a `PipedInputStream`.

Filter Streams

A `FilterInputStream` filters an `InputStream`, and a `FilterOutputStream` filters an `OutputStream`. **Filtering** means simply that the filter stream provides additional functionality, such as aggregating bytes into meaningful primitive-type units. `FilterInputStream` and `FilterOutputStream` are typically used as superclasses, so some of their filtering capabilities are provided by their subclasses.

A `PrintStream` (a subclass of `FilterOutputStream`) performs text output to the specified stream. Actually, we've been using `PrintStream` output throughout the text to this point—`System.out` and `System.err` are `PrintStream` objects.

Data Streams

Reading data as raw bytes is fast, but crude. Usually, programs read data as aggregates of bytes that form `int`s, `float`s, `double`s and so on. Java programs can use several classes to input and output data in aggregate form.

Interface `DataInput` describes methods for reading primitive types from an input stream. Classes

DataInputStream and RandomAccessFile each implement this interface to read sets of bytes and view them as primitive-type values. Interface DataInput includes methods such as readBoolean, readByte, readChar, readDouble, readFloat, readFully (for byte arrays), readInt, readLong, readShort, readUnsignedByte, readUnsigned-Short, readUTF (for reading Unicode characters encoded by Java—we discuss UTF encoding in Appendix H) and skipBytes.

Interface DataOutput describes a set of methods for writing primitive types to an output stream. Classes DataOutputStream (a subclass of FilterOutputStream) and RandomAccessFile each implement this interface to write primitive-type values as bytes. Interface DataOutput includes overloaded versions of method write (for a byte or for a byte array) and methods writeBoolean, writeByte, writeBytes, writeChar, writeChars (for Unicode Strings), writeDouble, writeFloat, writeInt, writeLong, writeShort and writeUTF (to output text modified for Unicode).

Buffered Streams

Buffering is an I/O-performance-enhancement technique. With a BufferedOutput-Stream (a subclass of class FilterOutputStream), each output statement does *not* necessarily result in an actual physical transfer of data to the output device (which is a slow operation compared to processor and main memory speeds). Rather, each output operation is directed to a region in memory called a **buffer** that's large enough to hold the data of many output operations. Then, actual transfer to the output device is performed in one large **physical output operation** each time the buffer fills. The output operations directed to the output buffer in memory are often called **logical output operations**. With a `BufferedOutputStream`, a partially filled buffer can be forced out to the device at any time by invoking the stream object's `flush` method.

Using buffering can greatly increase the performance of an application. Typical I/O operations are extremely slow compared with the speed of accessing data in computer memory. Buffering reduces the number of I/O operations by first combining smaller outputs together in memory. The number of actual physical I/O operations is small compared with the number of I/O requests issued by the program. Thus, the program that's using buffering is more efficient.

Performance Tip 15.1

Buffered I/O can yield significant performance improvements over unbuffered I/O.

With a `BufferedInputStream` (a subclass of class `FilterInputStream`), many "logical" chunks of data from a file are read as one large **physical input operation** into a memory buffer. As a program requests each new chunk of data, it’s taken from the buffer. (This procedure is sometimes referred to as a **logical input operation**.) When the buffer is empty, the next actual physical input operation from the input device is performed to read in the next group of “logical” chunks of data. Thus, the number of actual physical input operations is small compared with the number of read requests issued by the program.

Memory-Based `byte` Array Steams

Java stream I/O includes capabilities for inputting from `byte` arrays in memory and outputting to `byte` arrays in memory. A `ByteArrayInputStream` (a subclass of `InputStream`) reads from a `byte` array in memory. A `ByteArrayOutputStream` (a subclass of `OutputStream`) outputs to a `byte` array in memory. One use of `byte`-array I/O is *data validation*. A program can input an entire line at a time from the input stream into a `byte` array. Then a validation routine can scrutinize the contents of the `byte` array and correct the data if necessary. Finally, the program can proceed to input from the `byte` array, “knowing” that the input data is in the proper format. Outputting to a `byte` array is a nice way to take advantage of the powerful output-formatting capabilities of Java streams. For example, data can be stored in a `byte` array, using the same formatting that will be displayed at a later time, and the `byte` array can then be output to a file to preserve the formatting.

Sequencing Input from Multiple Streams

A `SequenceInputStream` (a subclass of `InputStream`) logically concatenates several `InputStream`s—the program sees the group as one continuous `InputStream`. When the program reaches the end of one input stream, that stream closes, and the next stream in the sequence opens.

15.7.2 Interfaces and Classes for Character-Based Input and Output

In addition to the byte-based streams, Java provides the `Reader` and `Writer` `abstract` classes, which are character-based streams like those you used for text-file processing in Section 15.4. Most of the byte-based streams have corresponding character-based concrete `Reader` or `Writer` classes.

Character-Based Buffering

Readers and Writers

Classes BufferedReader (a subclass of abstract class Reader) and BufferedWriter (a subclass of abstract class Writer) enable buffering for character-based streams. Remember that character-based streams use Unicode characters—such streams can process data in any language that the Unicode character set represents.

Memory-Based char Array Readers and Writers

Classes CharArrayReader and CharArrayWriter read and write, respectively, a stream of characters to a char array. A LineNumberReader (a subclass of BufferedReader) is a buffered character stream that keeps track of the number of lines read—newlines, returns and carriage-return–line-feed combinations increment the line count. Keeping track of line numbers can be useful if the program needs to inform the reader of an error on a specific line.

Character-Based File, Pipe and String Readers and Writers

An `InputStream` can be converted to a `Reader` via class `InputStreamReader`. Similarly, an `OutputStream` can be converted to a `Writer` via class `OutputStreamWriter`. Class `FileReader` (a subclass of `InputStreamReader`) and class `FileWriter` (a subclass of `OutputStreamWriter`) read characters from and write characters to a file, respectively. Class `PipedReader` and class `PipedWriter` implement piped-character streams for transferring data between threads. Class `StringReader` and `StringWriter` read characters from and write characters to `String`s, respectively. A `PrintWriter` writes characters to a stream.