Table of Contents

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

Chapter 4. Streams

A large part of what network programs do is simple input and output: moving bytes from one system to another. Bytes are bytes; to a large extent, reading data a server sends you is not all that different from reading a file. Sending text to a client is not that different from writing a file. However, input and output (I/O) in Java is organized differently than it is in most other languages, such as Fortran, C, and C++. Consequently, we'll take a few pages to summarize Java's unique approach to I/O.

I/O in Java is built on *streams*. Input streams read data; output streams write data. Different stream classes, like `java.io.FileInputStream` and `sun.net.TelnetOutputStream`, read and write particular sources of data. However, all output streams have the same basic methods to write data and all input streams use the same basic methods to read data. After a stream is created, you can often ignore the details of exactly what it is you're reading or writing.

Filter streams can be chained to either an input stream or an output stream. Filters can modify the data as it's read or written—for instance, by encrypting or compressing it—or they can simply provide additional methods for converting the data that's read or written into other formats. For instance, the `java.io.DataOutputStream` class provides a method that converts an `int` to four bytes and writes those bytes onto its underlying output stream.

Readers and writers can be chained to input and output streams to allow programs to read and write text (that is, characters) rather than bytes. Used properly, readers and writers can handle a wide variety of character encodings, including multibyte character sets such as SJIS and UTF-8.

Streams are synchronous; that is, when a program (really, a thread) asks a stream to read or write a piece of data, it waits for the data to be read or written before it does anything else. Java 1.4 and later also support non-blocking I/O using channels and buffers. Non-blocking I/O is a little more complicated, but much faster in some high-volume applications, such as web servers. Normally, the basic stream model is all you need and all you should use for

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

clients. Since channels and buffers depend on streams, we'll start with streams and clients and later discuss non-blocking I/O for use with servers in Chapter 12.

4.1. Output Streams

Java's basic output class is `java.io.OutputStream`:

```
public abstract class OutputStream
```

This class provides the fundamental methods needed to write data. These are:

```
public abstract void write(int b) throws IOException
public void write(byte[] data) throws IOException
public void write(byte[] data, int offset, int length)
 throws IOException
public void flush( ) throws IOException
public void close( ) throws IOException
```

Subclasses of `OutputStream` use these methods to write data onto particular media. For instance, a `FileOutputStream` uses these methods to write data into a file. A `TelnetOutputStream` uses these methods to write data onto a network connection. A `ByteArrayOutputStream` uses these methods to write data into an expandable byte array. But whichever medium you're writing to, you mostly use only these same five methods. Sometimes you may not even know exactly what kind of stream you're writing onto. For instance, you won't find `TelnetOutputStream` in the Java class library documentation. It's deliberately hidden inside the `sun` packages. It's returned by various methods in various classes in `java.net`, like the `getOutputStream()` method of `java.net.Socket`. However, these methods are declared to return only `OutputStream`, not the more specific subclass `TelnetOutputStream`. That's the power of polymorphism. If you know how to use the superclass, you know how to use all the subclasses, too.

`OutputStream`'s fundamental method is `write(int b)`. This method takes an integer from 0 to 255 as an argument and writes the corresponding byte to the output stream. This method is declared abstract because subclasses need to change it to handle their particular medium. For instance, a `ByteArrayOutputStream` can implement this method with pure Java code that copies the byte into its array. However, a `FileOutputStream` will need to use native code that understands how to write data in files on the host platform.

Take note that although this method takes an `int` as an argument, it actually writes an unsigned byte. Java doesn't have an unsigned byte data type, so an `int` has to be used here instead. The only real difference between an unsigned byte and a signed byte is the

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

interpretation. They're both made up of eight bits, and when you write an `int` onto a network connection using `write(int b)`, only eight bits are placed on the wire. If an `int` outside the range 0-255 is passed to `write(int b)`, the least significant byte of the number is written and the remaining three bytes are ignored. (This is the effect of casting an `int` to a `byte`.) On rare occasions, however, you may find a buggy third-party class that does something different, such as throwing an `IllegalArgumentException` or always writing 255, so it's best not to rely on this behavior, if possible.

For example, the character generator protocol defines a server that sends out ASCII text. The most popular variation of this protocol sends 72-character lines containing printable ASCII characters. (The printable ASCII characters are those between 33 and 126 inclusive that exclude the various whitespace and control characters.) The first line contains characters 33 through 104, sorted. The second line contains characters 34 through 105. The third line contains characters 35 through 106. This continues through line 29, which contains characters 55 through 126. At that point, the characters wrap around so that line 30 contains characters 56 through 126 followed by character 33 again. Lines are terminated with a carriage return (ASCII 13) and a linefeed (ASCII 10). The output looks like this:

```
!"#$%&'( )*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefgh
"#$%&'( )*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghi
#$%&'( )*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghij
$%&'( )*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijk
%&'( )*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijkl
&'( )*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklm
'( )*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmn
```

Since ASCII is a 7-bit character set, each character is sent as a single byte. Consequently, this protocol is straightforward to implement using the basic `write( )` methods, as the next code fragment demonstrates:

```
public static void generateCharacters(OutputStream out)
  throws IOException {

   int firstPrintableCharacter     = 33;
   int numberOfPrintableCharacters = 94;
   int numberOfCharactersPerLine   = 72;

   int start = firstPrintableCharacter;
   while (true) { /* infinite loop */
     for (int i = start; i < start+numberOfCharactersPerLine; i++) {
       out.write((
        (i-firstPrintableCharacter) % numberOfPrintableCharacters)
         + firstPrintableCharacter);
     }
     out.write('\r'); // carriage return
     out.write('\n'); // linefeed
     start = ((start+1) - firstPrintableCharacter)
       % numberOfPrintableCharacters + firstPrintableCharacter;
   }
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

The character generator server class (the exact details of which will have to wait until we discuss server sockets in Chapter 10) passes an `OutputStream` named `out` to the `generateCharacters( )` method. Bytes are written onto `out` one at a time. These bytes are given as integers in a rotating sequence from 33 to 126. Most of the arithmetic here is to make the loop rotate in that range. After each 72 character chunk is written, a carriage return and a linefeed are written onto the output stream. The next start character is calculated and the loop repeats. The entire method is declared to throw `IOException`. That's important because the character generator server will terminate only when the client closes the connection. The Java code will see this as an `IOException`.

Writing a single byte at a time is often inefficient. For example, every TCP segment that goes out your Ethernet card contains at least 40 bytes of overhead for routing and error correction. If each byte is sent by itself, you may be stuffing the network with 41 times more data than you think you are! Consequently, most TCP/IP implementations buffer data to some extent. That is, they accumulate bytes in memory and send them to their eventual destination only when a certain number have accumulated or a certain amount of time has passed. However, if you have more than one byte ready to go, it's not a bad idea to send them all at once. Using `write(byte[] data)` or `write(byte[] data, int offset, int length)` is normally much faster than writing all the components of the `data` array one at a time. For instance, here's an implementation of the `generateCharacters()` method that sends a line at a time by packing a complete line into a byte array:

```
public static void generateCharacters(OutputStream out)
 throws IOException {

  int firstPrintableCharacter = 33;
  int numberOfPrintableCharacters = 94;
  int numberOfCharactersPerLine = 72;
  int start = firstPrintableCharacter;
  byte[] line = new byte[numberOfCharactersPerLine+2];
  // the +2 is for the carriage return and linefeed

  while (true) { /* infinite loop */
    for (int i = start; i < start+numberOfCharactersPerLine; i++) {
      line[i-start] = (byte) ((i-firstPrintableCharacter)
       % numberOfPrintableCharacters + firstPrintableCharacter);
    }
    line[72] = (byte) '\r'; // carriage return
    line[73] = (byte) '\n'; // line feed
    out.write(line);
    start = ((start+1)-firstPrintableCharacter)
     % numberOfPrintableCharacters + firstPrintableCharacter;
  }

}
```

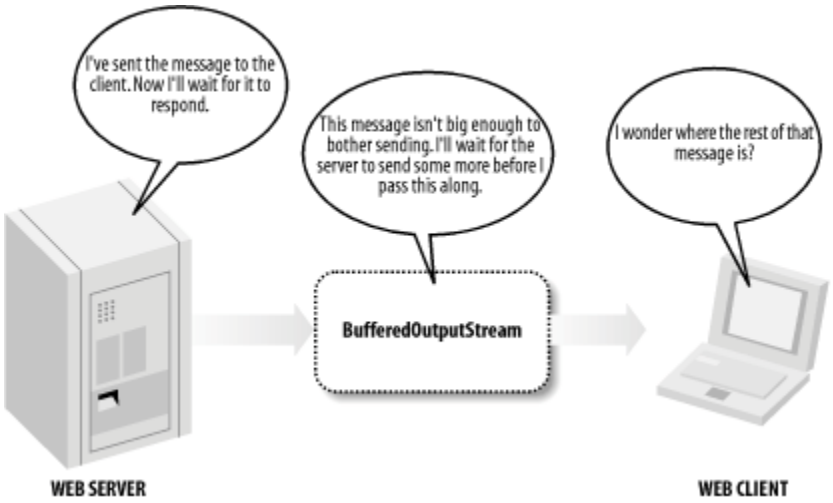
The algorithm for calculating which bytes to write when is the same as for the previous implementation. The crucial difference is that the bytes are packed into a byte array before being written onto the network. Also, notice that the `int` result of the calculation must be

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

cast to a `byte` before being stored in the array. This wasn't necessary in the previous implementation because the single byte `write()` method is declared to take an `int` as an argument.

Streams can also be buffered in software, directly in the Java code as well as in the network hardware. Typically, this is accomplished by chaining a `BufferedOutputStream` or a `BufferedWriter` to the underlying stream, a technique we'll explore shortly. Consequently, if you are done writing data, it's important to flush the output stream. For example, suppose you've written a 300-byte request to an HTTP 1.1 server that uses HTTP Keep-Alive. You generally want to wait for a response before sending any more data. However, if the output stream has a 1,024-byte buffer, the stream may be waiting for more data to arrive before it sends the data out of its buffer. No more data will be written onto the stream until the server response arrives, but the response is never going to arrive because the request hasn't been sent yet! The buffered stream won't send the data to the server until it gets more data from the underlying stream, but the underlying stream won't send more data until it gets data from the server, which won't send data until it gets the data that's stuck in the buffer! Figure 4-1 illustrates this Catch-22. The `flush()` method breaks the deadlock by forcing the buffered stream to send its data even if the buffer isn't yet full.

Figure 4-1. Data can get lost if you don't flush your streams



It's important to flush your streams whether you think you need to or not. Depending on how you got hold of a reference to the stream, you may or may not know whether it's buffered. (For instance, `System.out` is buffered whether you want it to be or not.) If flushing isn't necessary for a particular stream, it's a very low cost operation. However, if it is necessary, it's very necessary. Failing to flush when you need to can lead to unpredictable, unrepeatable program hangs that are extremely hard to diagnose if you don't have a good idea of what the problem is in the first place. As a corollary to all this, you should flush all streams

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

immediately before you close them. Otherwise, data left in the buffer when the stream is closed may get lost.

Finally, when you're done with a stream, close it by invoking its `close( )` method. This releases any resources associated with the stream, such as file handles or ports. Once an output stream has been closed, further writes to it throw `IOException`s. However, some kinds of streams may still allow you to do things with the object. For instance, a closed `ByteArrayOutputStream` can still be converted to an actual byte array and a closed `DigestOutputStream` can still return its digest.

4.2. Input Streams

Java's basic input class is `java.io.InputStream`:

```
public abstract class InputStream
```

This class provides the fundamental methods needed to read data as raw bytes. These are:

```
public abstract int read( ) throws IOException
public int read(byte[] input) throws IOException
public int read(byte[] input, int offset, int length) throws IOException
public long skip(long n) throws IOException
public int available( ) throws IOException
public void close( ) throws IOException
```

Concrete subclasses of `InputStream` use these methods to read data from particular media. For instance, a `FileInputStream` reads data from a file. A `TelnetInputStream` reads data from a network connection. A `ByteArrayInputStream` reads data from an array of bytes. But whichever source you're reading, you mostly use only these same six methods. Sometimes you don't know exactly what kind of stream you're reading from. For instance, `TelnetInputStream` is an undocumented class hidden inside the `sun.net` package. Instances of it are returned by various methods in the `java.net` package: for example, the `openStream( )` method of `java.net.URL`. However, these methods are declared to return only `InputStream`, not the more specific subclass `TelnetInputStream`. That's polymorphism at work once again. The instance of the subclass can be used transparently as an instance of its superclass. No specific knowledge of the subclass is required.

The basic method of `InputStream` is the noargs `read( )` method. This method reads a single byte of data from the input stream's source and returns it as an `int` from 0 to 255. End of stream is signified by returning -1. The `read( )` method waits and blocks execution of any code that follows it until a byte of data is available and ready to be read. Input and output

can be slow, so if your program is doing anything else of importance, try to put I/O in its own thread.

The `read( )` method is declared abstract because subclasses need to change it to handle their particular medium. For instance, a `ByteArrayInputStream` can implement this method with pure Java code that copies the byte from its array. However, a `TelnetInputStream` needs to use a native library that understands how to read data from the network interface on the host platform.

The following code fragment reads 10 bytes from the `InputStream in` and stores them in the `byte` array `input`. However, if end of stream is detected, the loop is terminated early:

```
byte[] input = new byte[10];
for (int i = 0; i < input.length; i++) {
  int b = in.read( );
  if (b  == -1) break;
  input[i] = (byte) b;
}
```

Although `read( )` only reads a byte, it returns an `int`. Thus, a cast is necessary before storing the result in the byte array. Of course, this produces a signed byte from -128 to 127 instead of the unsigned byte from 0 to 255 returned by the `read( )` method. However, as long as you're clear about which one you're working with, this is not a major problem. You can convert a signed byte to an unsigned byte like this:

```
int i = b >= 0 ? b : 256 + b;
```

Reading a byte at a time is as inefficient as writing data one byte at a time. Consequently, there are two overloaded `read()` methods that fill a specified array with multiple bytes of data read from the stream, `read(byte[] input)` and `read(byte[] input, int offset, int length)`. The first method attempts to fill the specified array `input`. The second attempts to fill the specified subarray of `input`, starting at `offset` and continuing for `length` bytes.

Notice I said these methods *attempt* to fill the array, not that they necessarily succeed. An attempt may fail in several ways. For instance, it's not unheard of that while your program is reading data from a remote web server over a PPP dialup link, a bug in a switch at a phone company central office will disconnect you and several thousand of your neighbors from the rest of the world. This would cause an `IOException`. More commonly, however, a read attempt won't completely fail but won't completely succeed, either. Some of the requested bytes may be read, but not all of them. For example, you may try to read 1,024 bytes from a network connection, when only 512 have actually arrived from the server; the rest are still in transit. They'll arrive eventually, but they aren't available at this moment. To account for this,

the multibyte read methods return the number of bytes actually read. For example, consider this code fragment:

```
byte[] input  = new byte[1024];
int bytesRead = in.read(input);
```

It attempts to read 1,024 bytes from the `InputStream in` into the array `input`. However, if only 512 bytes are available, that's all that will be read, and `bytesRead` will be set to 512. To guarantee that all the bytes you want are actually read, place the read in a loop that reads repeatedly until the array is filled. For example:

```
int bytesRead   = 0;
int bytesToRead = 1024;
byte[] input    = new byte[bytesToRead];
while (bytesRead < bytesToRead) {
  bytesRead += in.read(input, bytesRead, bytesToRead - bytesRead);
}
```

This technique is especially crucial for network streams. Chances are that if a file is available at all, all the bytes of a file are also available. However, since networks move much more slowly than CPUs, it is very easy for a program to empty a network buffer before all the data has arrived. In fact, if one of these two methods tries to read from a temporarily empty but open network buffer, it will generally return 0, indicating that no data is available but the stream is not yet closed. This is often preferable to the behavior of the single-byte `read ( )` method, which blocks the running thread in the same circumstances.

All three `read( )` methods return -1 to signal the end of the stream. If the stream ends while there's still data that hasn't been read, the multibyte `read( )` methods return the data until the buffer has been emptied. The next call to any of the `read( )` methods will return -1. The -1 is never placed in the array. The array only contains actual data. The previous code fragment had a bug because it didn't consider the possibility that all 1,024 bytes might never arrive (as opposed to not being immediately available). Fixing that bug requires testing the return value of `read( )` before adding it to `bytesRead`. For example:

```
int bytesRead=0;
int bytesToRead=1024;
byte[] input = new byte[bytesToRead];
while (bytesRead < bytesToRead) {
  int result = in.read(input, bytesRead, bytesToRead - bytesRead);
  if (result == -1) break;
  bytesRead += result;
}
```

If you do not want to wait until all the bytes you need are immediately available, you can use the `available()` method to determine how many bytes can be read without blocking. This returns the minimum number of bytes you can read. You may in fact be able to read

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

more, but you will be able to read at least as many bytes as `available()` suggests. For example:

```
int bytesAvailable = in.available( );
byte[] input = new byte[bytesAvailable];
int bytesRead = in.read(input, 0, bytesAvailable);
// continue with rest of program immediately...
```

In this case, you can assert that `bytesRead` is exactly equal to `bytesAvailable`. You cannot, however, assert that `bytesRead` is greater than zero. It is possible that no bytes were available. On end of stream, `available( )` returns 0. Generally, `read(byte[] input, int offset, int length)` returns -1 on end of stream; but if `length` is 0, then it does not notice the end of stream and returns 0 instead.

On rare occasions, you may want to skip over data without reading it. The `skip( )` method accomplishes this task. It's less useful on network connections than when reading from files. Network connections are sequential and normally quite slow, so it's not significantly more time-consuming to read data than to skip over it. Files are random access so that skipping can be implemented simply by repositioning a file pointer rather than processing each byte to be skipped.

As with output streams, once your program has finished with an input stream, it should close it by invoking its `close()` method. This releases any resources associated with the stream, such as file handles or ports. Once an input stream has been closed, further reads from it throw `IOException`s. However, some kinds of streams may still allow you to do things with the object. For instance, you generally won't want to get the message digest from a `java.security.DigestInputStream` until after the data has been read and the stream closed.

4.2.1. Marking and Resetting

The `InputStream` class also has three less commonly used methods that allow programs to back up and reread data they've already read. These are:

```
public void mark(int readAheadLimit)
public void reset( ) throws IOException
public boolean markSupported( )
```

In order to reread data, mark the current position in the stream with the `mark( )` method. At a later point, you can reset the stream to the marked position using the `reset( )` method. Subsequent reads then return data starting from the marked position. However, you may not be able to reset as far back as you like. The number of bytes you can read from the mark

and still reset is determined by the `readAheadLimit` argument to `mark( )`. If you try to reset too far back, an `IOException` is thrown. Furthermore, there can be only one mark in a stream at any given time. Marking a second location erases the first mark.

Marking and resetting are usually implemented by storing every byte read from the marked position on in an internal buffer. However, not all input streams support this. Before trying to use marking and resetting, check to see whether the `markSupported()` method returns true. If it does, the stream supports marking and resetting. Otherwise, `mark( )` will do nothing and `reset( )` will throw an `IOException`.

In my opinion, this demonstrates very poor design. In practice, more streams *don't* support marking and resetting than *do*. Attaching functionality to an abstract superclass that is not available to many, probably most, subclasses is a very poor idea. It would be better to place these three methods in a separate interface that could be implemented by those classes that provided this functionality. The disadvantage of this approach is that you couldn't then invoke these methods on an arbitrary input stream of unknown type, but in practice, you can't do that anyway because not all streams support marking and resetting. Providing a method such as `markSupported( )` to check for functionality at runtime is a more traditional, non-object-oriented solution to the problem. An object-oriented approach would embed this in the type system through interfaces and classes so that it could all be checked at compile time.

The only two input stream classes in `java.io` that always support marking are `BufferedInputStream` and `ByteArrayInputStream`. However, other input streams such as `TelnetInputStream` may support marking if they're chained to a buffered input stream first.

4.3. Filter Streams

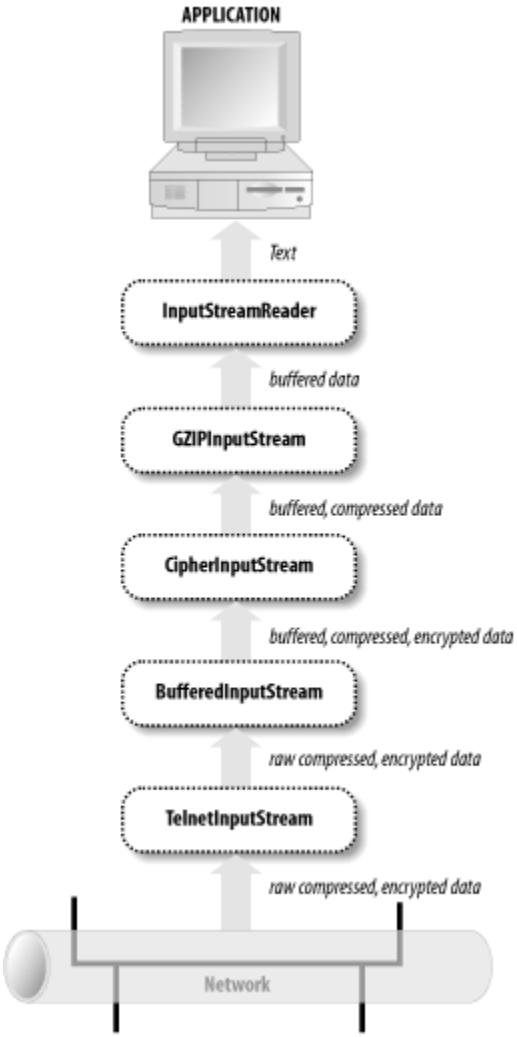
`InputStream` and `OutputStream` are fairly raw classes. They read and write bytes singly or in groups, but that's all. Deciding what those bytes mean—whether they're integers or IEEE 754 floating point numbers or Unicode text—is completely up to the programmer and the code. However, there are certain extremely common data formats that can benefit from a solid implementation in the class library. For example, many integers passed as parts of

network protocols are 32-bit big-endian integers. Much text sent over the Web is either 7-bit ASCII, 8-bit Latin-1, or multi-byte UTF-8. Many files transferred by FTP are stored in the zip format. Java provides a number of filter classes you can attach to raw streams to translate the raw bytes to and from these and other formats.

The filters come in two versions: the filter streams and the readers and writers. The filter streams still work primarily with raw data as bytes: for instance, by compressing the data or interpreting it as binary numbers. The readers and writers handle the special case of text in a variety of encodings such as UTF-8 and ISO 8859-1. Filter streams are placed on top of raw streams such as a `TelnetInputStream` or a `FileOutputStream` or other filter streams. Readers and writers can be layered on top of raw streams, filter streams, or other readers and writers. However, filter streams cannot be placed on top of a reader or a writer, so we'll start with filter streams and address readers and writers in the next section.

Filters are organized in a chain, as shown in Figure 4-2. Each link in the chain receives data from the previous filter or stream and passes the data along to the next link in the chain. In this example, a compressed, encrypted text file arrives from the local network interface, where native code presents it to the undocumented `TelnetInputStream`. A `BufferedInputStream` buffers the data to speed up the entire process. A `CipherInputStream` decrypts the data. A `GZIPInputStream` decompresses the deciphered data. An `InputStreamReader` converts the decompressed data to Unicode text. Finally, the text is read into the application and processed.

Figure 4-2. The flow of data through a chain of filters



Every filter output stream has the same `write( )`, `close( )`, and `flush( )` methods as `java.io.OutputStream`. Every filter input stream has the same `read( )`, `close( )`, and `available( )` methods as `java.io.InputStream`. In some cases, such as `BufferedInputStream` and `BufferedOutputStream`, these may be the only methods they have. The filtering is purely internal and does not expose any new public interface. However, in most cases, the filter stream adds public methods with additional purposes. Sometimes these are intended to be used in addition to the usual `read()` and `write( )` methods, like the `unread( )` method of `PushbackInputStream`. At other times, they almost completely replace the original interface. For example, it's relatively rare to use the `write()` method of `PrintStream` instead of one of its `print( )` and `println( )` methods.

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

4.3.1. Chaining Filters Together

Filters are connected to streams by their constructors. For example, the following code fragment buffers input from the file *data.txt*. First, a `FileInputStream` object `fin` is created by passing the name of the file as an argument to the `FileInputStream` constructor. Then, a `BufferedInputStream` object `bin` is created by passing `fin` as an argument to the `BufferedInputStream` constructor:

```
FileInputStream     fin = new FileInputStream("data.txt");
BufferedInputStream bin = new BufferedInputStream(fin);
```

From this point forward, it's possible to use the `read( )` methods of both `fin` and `bin` to read data from the file *data.txt*. However, intermixing calls to different streams connected to the same source may violate several implicit contracts of the filter streams. Most of the time, you should only use the last filter in the chain to do the actual reading or writing. One way to write your code so that it's at least harder to introduce this sort of bug is to deliberately lose the reference to the underlying input stream. For example:

```
InputStream in = new FileInputStream("data.txt");
in = new BufferedInputStream(in);
```

After these two lines execute, there's no longer any way to access the underlying file input stream, so you can't accidentally read from it and corrupt the buffer. This example works because it's not necessary to distinguish between the methods of `InputStream` and those of `BufferedInputStream`. `BufferedInputStream` is simply used polymorphically as an instance of `InputStream` in the first place. In cases where it is necessary to use the additional methods of the filter stream not declared in the superclass, you may be able to construct one stream directly inside another. For example:

```
DataOutputStream dout = new DataOutputStream(new BufferedOutputStream(
 new FileOutputStream("data.txt")));
```

Although these statements can get a little long, it's easy to split the statement across several lines, like this:

```
DataOutputStream dout = new DataOutputStream(
                         new BufferedOutputStream(
                          new FileOutputStream("data.txt")
                         )
                        );
```

Connection is permanent. Filters cannot be disconnected from a stream.

There are times when you may need to use the methods of multiple filters in a chain. For instance, if you're reading a Unicode text file, you may want to read the byte order mark in the first three bytes to determine whether the file is encoded as big-endian UCS-2, little-endian UCS-2, or UTF-8, and then select the matching `Reader` filter for the encoding. Or, if you're connecting to a web server, you may want to read the header the server sends to find the `Content-encoding` and then use that content encoding to pick the right `Reader` filter to read the body of the response. Or perhaps you want to send floating point numbers across a network connection using a `DataOutputStream` and then retrieve a `MessageDigest` from the `DigestOutputStream` that the `DataOutputStream` is chained to. In all these cases, you need to save and use references to each of the underlying streams. However, under no circumstances should you ever read from or write to anything other than the last filter in the chain.

4.3.2. Buffered Streams

The `BufferedOutputStream` class stores written data in a buffer (a protected byte array field named `buf`) until the buffer is full or the stream is flushed. Then it writes the data onto the underlying output stream all at once. A single write of many bytes is almost always much faster than many small writes that add up to the same thing. This is especially true of network connections because each TCP segment or UDP packet carries a finite amount of overhead, generally about 40 bytes' worth. This means that sending 1 kilobyte of data 1 byte at a time actually requires sending 40 kilobytes over the wire, whereas sending it all at once only requires sending a little more than 1K of data. Most network cards and TCP implementations provide some level of buffering themselves, so the real numbers aren't quite this dramatic. Nonetheless, buffering network output is generally a huge performance win.

The `BufferedInputStream` class also has a protected byte array named `buf` that serves as a buffer. When one of the stream's `read( )` methods is called, it first tries to get the requested data from the buffer. Only when the buffer runs out of data does the stream read from the underlying source. At this point, it reads as much data as it can from the source into the buffer, whether it needs all the data immediately or not. Data that isn't used immediately will be available for later invocations of `read()`. When reading files from a local disk, it's almost as fast to read several hundred bytes of data from the underlying stream as it is to read one byte of data. Therefore, buffering can substantially improve performance. The gain is less obvious on network connections where the bottleneck is often the speed at which the network can deliver data rather than the speed at which the network interface delivers data to the program or the speed at which the program runs. Nonetheless, buffering input rarely hurts and will become more important over time as network speeds increase.

`BufferedInputStream` has two constructors, as does `BufferedOutputStream`:

```
public BufferedInputStream(InputStream in)
public BufferedInputStream(InputStream in, int bufferSize)
public BufferedOutputStream(OutputStream out)
public BufferedOutputStream(OutputStream out, int bufferSize)
```

The first argument is the underlying stream from which unbuffered data will be read or to which buffered data will be written. The second argument, if present, specifies the number of bytes in the buffer. Otherwise, the buffer size is set to 2,048 bytes for an input stream and 512 bytes for an output stream. The ideal size for a buffer depends on what sort of stream you're buffering. For network connections, you want something a little larger than the typical packet size. However, this can be hard to predict and varies depending on local network connections and protocols. Faster, higher-bandwidth networks tend to use larger packets, although eight kilobytes is an effective maximum packet size for UDP on most networks today, and TCP segments are often no larger than a kilobyte.

`BufferedInputStream` does not declare any new methods of its own. It only overrides methods from `InputStream`. It does support marking and resetting.

```
public int read( ) throws IOException
public int read(byte[] input, int offset, int length)
 throws IOException
public long skip(long n) throws IOException
public int available( ) throws IOException
public void mark(int readLimit)
public void reset( ) throws IOException
public boolean markSupported( )
```

The two multibyte `read( )` methods attempt to completely fill the specified array or subarray of data by reading from the underlying input stream as many times as necessary. They return only when the array or subarray has been completely filled, the end of stream is reached, or the underlying stream would block on further reads. Most input streams (including buffered input streams in Java 1.1 and 1.0) do not behave like this. They read from the underlying stream or data source only once before returning.

`BufferedOutputStream` also does not declare any new methods of its own. It overrides three methods from `OutputStream`:

```
public void write(int b) throws IOException
public void write(byte[] data, int offset, int length) throws IOException
public void flush( ) throws IOException
```

You call these methods exactly as you would in any output stream. The difference is that each write places data in the buffer rather than directly on the underlying output stream. Consequently, it is essential to flush the stream when you reach a point at which the data needs to be sent.

4.3.3. PrintStream

The `PrintStream` class is the first filter output stream most programmers encounter because `System.out` is a `PrintStream`. However, other output streams can also be chained to print streams, using these two constructors:

```
public PrintStream(OutputStream out)
public PrintStream(OutputStream out, boolean autoFlush)
```

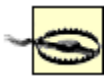
By default, print streams should be explicitly flushed. However, if the `autoFlush` argument is `true`, the stream will be flushed every time a byte array or linefeed is written or a `println( )` method is invoked.

As well as the usual `write( )`, `flush()`, and `close( )` methods, `PrintStream` has 9 overloaded `print()` methods and 10 overloaded `println( )` methods:

```
public void print(boolean b)
public void print(char c)
public void print(int i)
public void print(long l)
public void print(float f)
public void print(double d)
public void print(char[] text)
public void print(String s)
public void print(Object o)
public void println( )
public void println(boolean b)
public void println(char c)
public void println(int i)
public void println(long l)
public void println(float f)
public void println(double d)
public void println(char[] text)
public void println(String s)
public void println(Object o)
```

Each `print( )` method converts its argument to a string in a predictable fashion and writes the string onto the underlying output stream using the default encoding. The `println( )` methods do the same thing, but they also append a platform-dependent line separator character to the end of the line they write. This is a linefeed (`\n`) on Unix (including Mac OS X), a carriage return (`\r`) on Mac OS 9, and a carriage return/linefeed pair (`\r\n`) on Windows.

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004



`PrintStream` is evil and network programmers should avoid it like the plague!

The first problem is that the output from `println( )` is platform-dependent. Depending on what system runs your code, lines may sometimes be broken with a linefeed, a carriage return, or a carriage return/linefeed pair. This doesn't cause problems when writing to the console, but it's a disaster for writing network clients and servers that must follow a precise protocol. Most network protocols such as HTTP and Gnutela specify that lines should be terminated with a carriage return/linefeed pair. Using `println( )` makes it easy to write a program that works on Windows but fails on Unix and the Mac. While many servers and clients are liberal in what they accept and can handle incorrect line terminators, there are occasional exceptions. In particular, in conjunction with the bug in `readLine( )` discussed shortly, a client running on Mac OS 9 that uses `println( )` may hang both the server and the client. To some extent, this could be fixed by using only `print( )` and ignoring `println()`. However, `PrintStream` has other problems.

The second problem is that `PrintStream` assumes the default encoding of the platform on which it's running. However, this encoding may not be what the server or client expects. For example, a web browser receiving XML files will expect them to be encoded in UTF-8 or UTF-16 unless the server tells it otherwise. However, a web server that uses `PrintStream` may well send the files encoded in CP1252 from a U.S.-localized Windows system or SJIS from a Japanese-localized system, whether the client expects or understands those encodings or not. `PrintStream` doesn't provide any mechanism for changing the default encoding. This problem can be patched over by using the related `PrintWriter` class instead. But the problems continue.

The third problem is that `PrintStream` eats all exceptions. This makes `PrintStream` suitable for textbook programs such as HelloWorld, since simple console output can be taught without burdening students with first learning about exception handling and all that implies. However, network connections are much less reliable than the console. Connections routinely fail because of network congestion, phone company misfeasance, remote systems crashing, and many other reasons. Network programs must be prepared to deal with unexpected interruptions in the flow of data. The way to do this is by handling exceptions. However, `PrintStream` catches any exceptions thrown by the underlying output stream. Notice that the declaration of the standard five `OutputStream` methods in `PrintStream` does not have the usual `throws IOException` declaration:

```
public abstract void write(int b)
public void write(byte[] data)
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
public void write(byte[] data, int offset, int length)
public void flush( )
public void close( )
```

Instead, `PrintStream` relies on an outdated and inadequate error flag. If the underlying stream throws an exception, this internal error flag is set. The programmer is relied upon to check the value of the flag using the `checkError()` method:

```
public boolean checkError( )
```

If programmers are to do any error checking at all on a `PrintStream`, they must explicitly check every call. Furthermore, once an error has occurred, there is no way to unset the flag so further errors can be detected. Nor is any additional information available about the error. In short, the error notification provided by `PrintStream` is wholly inadequate for unreliable network connections. At the end of this chapter, we'll introduce a class that fixes all these shortcomings.

4.3.4. PushbackInputStream

`PushbackInputStream` is a subclass of `FilterInputStream` that provides a pushback stack so that a program can "unread" bytes onto the input stream. This lets programs add data to a running stream. For example, you could prefix a stream with a header before passing it to another process that needed that header.

The `read( )` and `available( )` methods of `PushbackInputStream` are invoked exactly as with normal input streams. However, they first attempt to read from the pushback buffer before reading from the underlying input stream. What this class adds is `unread()` methods that push bytes into the buffer:

```
public void unread(int b) throws IOException
```

This method pushes an unsigned byte given as an `int` between 0 and 255 onto the stream. Integers outside this range are truncated to this range as by a cast to `byte`. Assuming nothing else is pushed back onto this stream, the next read from the stream will return that byte. As multiple bytes are pushed onto the stream by repeated invocations of `unread( )`, they are stored in a stack and returned in a last-in, first-out order. In essence, the buffer is a stack sitting on top of an input stream. Only when the stack is empty will the underlying stream be read.

There are two more `unread( )` methods that push a specified array or subarray onto the stream:

```
public void unread(byte[] input) throws IOException
public void unread(byte[] input, int offset, int length) throws IOException
```

The arrays are stacked in last-in, first-out order. However, bytes popped from the same array will be returned in the order they appeared in the array. That is, the zeroth component of the array will be read before the first component of the array.

By default, the buffer is only one byte long, and trying to unread more than one byte throws an `IOException`. However, the buffer size can be changed by passing a second argument to the constructor:

```
public PushbackInputStream(InputStream in)
public PushbackInputStream(InputStream in, int size)
```

Although `PushbackInputStream` and `BufferedInputStream` both use buffers, `BufferedInputStream` uses them for data read from the underlying input stream, while `PushbackInputStream` uses them for arbitrary data, which may or may not have been read from the stream originally. Furthermore, `PushbackInputStream` does not allow marking and resetting. The `markSupported()` method of `PushbackInputStream` returns false.

4.3.5. Data Streams

The `DataInputStream` and `DataOutputStream` classes provide methods for reading and writing Java's primitive data types and strings in a binary format. The binary formats used are primarily intended for exchanging data between two different Java programs whether through a network connection, a datafile, a pipe, or some other intermediary. What a data output stream writes, a data input stream can read. However, it happens that the formats are the same ones used for most Internet protocols that exchange binary numbers. For instance, the time protocol uses 32-bit big-endian integers, just like Java's `int` data type. The controlled-load network element service uses 32-bit IEEE 754 floating point numbers, just like Java's `float` data type. (This is probably correlation rather than causation. Both Java and most network protocols were designed by Unix programmers, and consequently both tend to use the formats common to most Unix systems.) However, this isn't true for all network protocols, so check the details of any protocol you use. For instance, the Network Time Protocol (NTP) represents times as 64-bit unsigned fixed point numbers with the integer part in the first 32 bits and the fraction part in the last 32 bits. This doesn't match any primitive data type in any common programming language, although it is fairly straightforward to work with—at least as far as is necessary for NTP.

The `DataOutputStream` class offers these 11 methods for writing particular Java data types:

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
public final void writeBoolean(boolean b) throws IOException
public final void writeByte(int b) throws IOException
public final void writeShort(int s) throws IOException
public final void writeChar(int c) throws IOException
public final void writeInt(int i) throws IOException
public final void writeLong(long l) throws IOException
public final void writeFloat(float f) throws IOException
public final void writeDouble(double d) throws IOException
public final void writeChars(String s) throws IOException
public final void writeBytes(String s) throws IOException
public final void writeUTF(String s) throws IOException
```

All data is written in big-endian format. Integers are written in two's complement in the minimum number of bytes possible. Thus, a `byte` is written as one two's-complement byte, a `short` as two two's-complement bytes, an `int` as four two's-complement bytes, and a `long` as eight two's-complement bytes. Floats and doubles are written in IEEE 754 form in 4 and 8 bytes, respectively. Booleans are written as a single byte with the value 0 for false and 1 for true. Chars are written as two unsigned bytes.

The last three methods are a little trickier. The `writeChars( )` method simply iterates through the `String` argument, writing each character in turn as a 2-byte, big-endian Unicode character (a UTF-16 code point, to be absolutely precise). The `writeBytes( )` method iterates through the `String` argument but writes only the least significant byte of each character. Thus, information will be lost for any string with characters from outside the Latin-1 character set. This method may be useful on some network protocols that specify the ASCII encoding, but it should be avoided most of the time.

Neither `writeChars( )` nor `writeBytes()` encodes the length of the string in the output stream. As a result, you can't really distinguish between raw characters and characters that make up part of a string. The `writeUTF( )` method does include the length of the string. It encodes the string itself in a *variant* of the UTF-8 encoding of Unicode. Since this variant is subtly incompatible with most non-Java software, it should be used only for exchanging data with other Java programs that use a `DataInputStream` to read strings. For exchanging UTF-8 text with all other software, you should use an `InputStreamReader` with the appropriate encoding. (There wouldn't be any confusion if Sun had just called this method and its partner `writeString( )` and `readString( )` rather than `writeUTF( )` and `readUTF( )`.)

Along with these methods for writing binary numbers and strings, `DataOutputStream` of course has the usual `write( )`, `flush( )`, and `close( )` methods any `OutputStream` class has.

`DataInputStream` is the complementary class to `DataOutputStream`. Every format that `DataOutputStream` writes, `DataInputStream` can read. In addition,

`DataInputStream` has the usual `read( )`, `available( )`, `skip()`, and `close( )` methods, as well as methods for reading complete arrays of bytes and lines of text.

There are 9 methods to read binary data that match the 11 methods in `DataOutputStream` (there's no exact complement for `writeBytes( )` or `writeChars( )`; these are handled by reading the bytes and chars one at a time):

```
public final boolean readBoolean( ) throws IOException
public final byte readByte( ) throws IOException
public final char readChar( ) throws IOException
public final short readShort( ) throws IOException
public final int readInt( ) throws IOException
public final long readLong( ) throws IOException
public final float readFloat( ) throws IOException
public final double readDouble( ) throws IOException
public final String readUTF( ) throws IOException
```

In addition, `DataInputStream` provides two methods to read unsigned bytes and unsigned shorts and return the equivalent `int`. Java doesn't have either of these data types, but you may encounter them when reading binary data written by a C program:

```
public final int readUnsignedByte( ) throws IOException
public final int readUnsignedShort( ) throws IOException
```

`DataInputStream` has the usual two multibyte `read( )` methods that read data into an array or subarray and return the number of bytes read. It also has two `readFully( )` methods that repeatedly read data from the underlying input stream into an array until the requested number of bytes have been read. If enough data cannot be read, an `IOException` is thrown. These methods are especially useful when you know in advance exactly how many bytes you have to read. This might be the case when you've read the `Content-length` field out of an HTTP header and thus know how many bytes of data there are:

```
public final int read(byte[] input) throws IOException
public final int read(byte[] input, int offset, int length)
 throws IOException
public final void readFully(byte[] input) throws IOException
public final void readFully(byte[] input, int offset, int length)
 throws IOException
```

Finally, `DataInputStream` provides the popular `readLine()` method that reads a line of text as delimited by a line terminator and returns a string:

```
public final String readLine( ) throws IOException
```

However, this method should not be used under any circumstances, both because it is deprecated and because it is buggy. It's deprecated because it doesn't properly convert non-

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

ASCII characters to bytes in most circumstances. That task is now handled by the `readLine ( )` method of the `BufferedReader` class. However, that method and this one share the same insidious bug: they do not always recognize a single carriage return as ending a line. Rather, `readLine( )` recognizes only a linefeed or a carriage return/linefeed pair. When a carriage return is detected in the stream, `readLine( )` waits to see whether the next character is a linefeed before continuing. If it is a linefeed, the carriage return and the linefeed are thrown away and the line is returned as a `String`. If it isn't a linefeed, the carriage return is thrown away, the line is returned as a `String`, and the extra character that was read becomes part of the next line. However, if the carriage return is the last character in the stream (a very likely occurrence if the stream originates from a Macintosh or a file created on a Macintosh), then `readLine( )` hangs, waiting for the last character, which isn't forthcoming.

This problem isn't obvious when reading files because there will almost certainly be a next character: -1 for end of stream, if nothing else. However, on persistent network connections such as those used for FTP and late-model HTTP, a server or client may simply stop sending data after the last character and wait for a response without actually closing the connection. If you're lucky, the connection may eventually time out on one end or the other and you'll get an `IOException`, although this will probably take at least a couple of minutes. If you're not lucky, the program will hang indefinitely.

Note that it is not enough for your program to merely be running on Windows or Unix to avoid this bug. It must also ensure that it does not send or receive text files created on a Macintosh and that it never talks to Macintosh clients or servers. These are very strong conditions in the heterogeneous world of the Internet. It's much simpler to avoid `readLine ()` completely.

4.3.6. Compressing Streams

The `java.util.zip` package contains filter streams that compress and decompress streams in zip, gzip, and deflate formats. Along with its better-known uses with files, this package allows Java applications to easily exchange compressed data across the network. HTTP 1.1 includes support for compressed file transfer in which the server compresses and the browser decompresses files, in effect trading increasingly cheap CPU power for still-expensive network bandwidth. This process is completely transparent to the user. Of course, it's not transparent to the programmer who has to write the compression and decompression code. However, the `java.util.zip` filter streams make it a lot more transparent than it otherwise would be.

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

There are six stream classes that perform compression and decompression; the input streams decompress data and the output streams compress it:

```
public class DeflaterOutputStream extends FilterOutputStream
public class InflaterInputStream extends FilterInputStream
public class GZIPOutputStream extends FilterOutputStream
public class GZIPInputStream extends FilterInputStream
public class ZipOutputStream extends FilterOutputStream
public class ZipInputStream extends FilterInputStream
```

All of these classes use essentially the same compression algorithm. They differ only in various constants and meta-information included with the compressed data. In addition, a zip stream may contain more than one compressed file.

Compressing and decompressing data with these classes is almost trivially easy. You simply chain the filter to the underlying stream and read or write it like normal. For example, suppose you want to read the compressed file *allnames.gz*. Simply open a `FileInputStream` to the file and chain a `GZIPInputStream` to it, like this:

```
FileInputStream fin = new FileInputStream("allnames.gz");
GZIPInputStream gzin = new GZIPInputStream(fin);
```

From this point forward, you can read uncompressed data from `gzin` using the usual `read( )`, `skip( )`, and `available( )` methods. For instance, this code fragment reads and decompresses a file named *allnames.gz* in the current working directory:

```
FileInputStream fin   = new FileInputStream("allnames.gz");
GZIPInputStream gzin  = new GZIPInputStream(fin);
FileOutputStream fout = new FileOutputStream("allnames");
int b = 0;
while ((b = gzin.read( )) != -1) fout.write(b);
gzin.close( );
out.flush( );
out.close( );
```

In fact, it isn't even necessary to know that `gzin` is a `GZIPInputStream` for this to work. A simple `InputStream` type works equally well. For example:

```
InputStream in = new GZIPInputStream(new FileInputStream("allnames.gz"));
```

`DeflaterOutputStream` and `InflaterInputStream` are equally straightforward. `ZipInputStream` and `ZipOutputStream` are a little more complicated because a zip file is actually an archive that may contain multiple entries, each of which must be read separately. Each file in a zip archive is represented as a `ZipEntry` object whose `getName( )` method returns the original name of the file. For example, this code fragment decompresses the archive *shareware.zip* in the current working directory:

```
FileInputStream fin = new FileInputStream("shareware.zip");
ZipInputStream zin = new ZipInputStream(fin);
ZipEntry ze = null;
int b = 0;
while ((ze = zin.getNextEntry( )) != null) {
  FileOutputStream fout = new FileOutputStream(ze.getName( ));
  while ((b = zin.read( )) != -1) fout.write(b);
  zin.closeEntry( );
  fout.flush( );
  fout.close( );
}
zin.close( );
```

4.3.7. Digest Streams

The `java.util.security` package contains two filter streams that can calculate a message digest for a stream. They are `DigestInputStream` and `DigestOutputStream`. A message digest, represented in Java by the `java.util.security.MessageDigest` class, is a strong hash code for the stream; that is, it is a large integer (typically 20 bytes long in binary format) that can easily be calculated from a stream of any length in such a fashion that no information about the stream is available from the message digest. Message digests can be used for digital signatures and for detecting data that has been corrupted in transit across the network.

In practice, the use of message digests in digital signatures is more important. Mere data corruption can be detected with much simpler, less computationally expensive algorithms. However, the digest filter streams are so easy to use that at times it may be worth paying the computational price for the corresponding increase in programmer productivity. To calculate a digest for an output stream, you first construct a `MessageDigest` object that uses a particular algorithm, such as the Secure Hash Algorithm (SHA). Pass both the `MessageDigest` object and the stream you want to digest to the `DigestOutputStream` constructor. This chains the digest stream to the underlying output stream. Then write data onto the stream as normal, flush it, close it, and invoke the `getMessageDigest()` method to retrieve the `MessageDigest` object. Finally, invoke the `digest( )` method on the `MessageDigest` object to finish calculating the actual digest. Here's an example:

```
MessageDigest sha = MessageDigest.getInstance("SHA");
DigestOutputStream dout = new DigestOutputStream(out, sha);
byte[] buffer = new byte[128];
while (true) {
  int bytesRead = in.read(buffer);
  if (bytesRead < 0) break;
  dout.write(buffer, 0, bytesRead);
}
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
dout.flush( );
dout.close( );
byte[] result = dout.getMessageDigest( ).digest( );
```

Calculating the digest of an input stream you read is equally simple. It still isn't quite as transparent as some of the other filter streams because you do need to be at least marginally conversant with the methods of the `MessageDigest` class. Nonetheless, it's still far easier than writing your own secure hash function and manually feeding it each byte you write.

Of course, you also need a way of associating a particular message digest with a particular stream. In some circumstances, the digest may be sent over the same channel used to send the digested data. The sender calculates the digest as it sends data, while the receiver calculates the digest as it receives the data. When the sender is done, it sends a signal that the receiver recognizes as indicating the end of the stream and then sends the digest. The receiver receives the digest, checks that the digest received is the same as the one calculated locally, and closes the connection. If the digests don't match, the receiver may instead ask the sender to send the message again. Alternatively, both the digest and the files it digests may be stored in the same zip archive. And there are many other possibilities. Situations like this generally call for the design of a relatively formal custom protocol. However, while the protocol may be complicated, the calculation of the digest is straightforward, thanks to the `DigestInputStream` and `DigestOutputStream` filter classes.

4.3.8. Encrypting Streams

The `CipherInputStream` and `CipherOutputStream` classes in the `javax.crypto` package provide encryption and decryption services. They are both powered by a `Cipher` engine object that encapsulates the algorithm used to perform encryption and decryption. By changing the `Cipher` engine object, you change the algorithm that the streams use to encrypt and decrypt. Most ciphers also require a *key* to encrypt and decrypt the data. Symmetric or secret key ciphers use the same key for both encryption and decryption. Asymmetric or public key ciphers use different keys for encryption and decryption. The encryption key can be distributed as long as the decryption key is kept secret. Keys are specific to the algorithm and are represented in Java by instances of the `java.security.Key` interface. The `Cipher` object is set in the constructor. Like all filter stream constructors, these constructors also take another input stream as an argument:

```
public CipherInputStream(InputStream in, Cipher c)
public CipherOutputStream(OutputStream out, Cipher c)
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

For legal reasons `CipherInputStream` and `CipherOutputStream` are not bundled with the core API in Java 1.3 and earlier. Instead, they are part of a standard extension to Java called the Java Cryptography Extension, JCE for short. This is in the `javax.crypto` package. Sun provides an implementation of this API (available from http://java.sun.com/products/jce/) and various third parties have written independent implementations. Of particular note is the Legion of the Bouncy Castle's open source implementation, which can be downloaded from http://www.bouncycastle.org/.

To get a properly initialized `Cipher` object, use the static `Cipher.getInstance( )` factory method. This `Cipher` object must be initialized for either encryption or decryption with `init( )` before being passed into one of the previous constructors. For example, this code fragment prepares a `CipherInputStream` for decryption using the password "two and not a fnord" and the Data Encryption Standard (DES) algorithm:

```
byte[] desKeyData = "two and not a fnord".getBytes( );
DESKeySpec desKeySpec = new DESKeySpec(desKeyData);
SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("DES");
SecretKey desKey = keyFactory.generateSecret(desKeySpec);
Cipher des = Cipher.getInstance("DES");
des.init(Cipher.DECRYPT_MODE, desKey);
CipherInputStream cin = new CipherInputStream(fin, des);
```

This fragment uses classes from the `java.security`, `java.security.spec`, `javax.crypto`, and `javax.crypto.spec` packages. Different implementations of the JCE support different groups of encryption algorithms. Common algorithms include DES, RSA, and Blowfish. The construction of a key is generally algorithm-specific. Consult the documentation for your JCE implementation for more details.

`CipherInputStream` overrides most of the normal `InputStream` methods like `read( )` and `available( )`. `CipherOutputStream` overrides most of the usual `OutputStream` methods like `write()` and `flush( )`. These methods are all invoked much as they would be for any other stream. However, as the data is read or written, the stream's `Cipher` object either decrypts or encrypts the data. (Assuming your program wants to work with unencrypted data—as is commonly the case—a cipher input stream will decrypt the data and a cipher output stream will encrypt the data.) For example, this code fragment encrypts the file *secrets.txt* using the password "Mary had a little spider":

```
String infile   = "secrets.txt";
String outfile  = "secrets.des";
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
String password = "Mary had a little spider";

try {

   FileInputStream fin = new FileInputStream(infile);
   FileOutputStream fout = new FileOutputStream(outfile);

   // register the provider that implements the algorithm
   Provider sunJce = new com.sun.crypto.provider.SunJCE( );
   Security.addProvider(sunJce);

   // create a key
   char[] pbeKeyData = password.toCharArray( );
   PBEKeySpec pbeKeySpec = new PBEKeySpec(pbeKeyData);
   SecretKeyFactory keyFactory =
   SecretKeyFactory.getInstance("PBEWithMD5AndDES");
   SecretKey pbeKey = keyFactory.generateSecret(pbeKeySpec);

   // use Data Encryption Standard
   Cipher pbe = Cipher.getInstance("PBEWithMD5AndDES");
   pbe.init(Cipher.ENCRYPT_MODE, pbeKey);
   CipherOutputStream cout = new CipherOutputStream(fout, pbe);

   byte[] input = new byte[64];
   while (true) {
     int bytesRead = fin.read(input);
     if (bytesRead == -1) break;
     cout.write(input, 0, bytesRead);
   }

   cout.flush( );
   cout.close( );
   fin.close( );

}
catch (Exception ex) {
   System.err.println(ex);
}
```

I admit that this is more complicated than it needs to be. There's a lot of setup work involved in creating the `Cipher` object that actually performs the encryption. Partly, that's because key generation involves quite a bit more than a simple password. However, a large part of the complication is due to inane U.S. export laws that prevent Sun from fully integrating the JCE with the JDK and JRE. To a large extent, the complex architecture used here is driven by a need to separate the actual encrypting and decrypting code from the cipher stream classes.

4.4. Readers and Writers

Many programmers have a bad habit of writing code as if all text were ASCII or at least in the native encoding of the platform. While some older, simpler network protocols, such as daytime, quote of the day, and chargen, do specify ASCII encoding for text, this is not true of

HTTP and many other more modern protocols, which allow a wide variety of localized encodings, such as K0I8-R Cyrillic, Big-5 Chinese, and ISO 8859-2 for most Central European languages. Java's native character set is the UTF-16 encoding of Unicode. When the encoding is no longer ASCII, the assumption that bytes and chars are essentially the same things also breaks down. Consequently, Java provides an almost complete mirror of the input and output stream class hierarchy designed for working with characters instead of bytes.

In this mirror image hierarchy, two abstract superclasses define the basic API for reading and writing characters. The `java.io.Reader` class specifies the API by which characters are read. The `java.io.Writer` class specifies the API by which characters are written. Wherever input and output streams use bytes, readers and writers use Unicode characters. Concrete subclasses of `Reader` and `Writer` allow particular sources to be read and targets to be written. Filter readers and writers can be attached to other readers and writers to provide additional services or interfaces.

The most important concrete subclasses of `Reader` and `Writer` are the `InputStreamReader` and the `OutputStreamWriter` classes. An `InputStreamReader` contains an underlying input stream from which it reads raw bytes. It translates these bytes into Unicode characters according to a specified encoding. An `OutputStreamWriter` receives Unicode characters from a running program. It then translates those characters into bytes using a specified encoding and writes the bytes onto an underlying output stream.

In addition to these two classes, the `java.io` package provides several raw reader and writer classes that read characters without directly requiring an underlying input stream, including:

- `FileReader`
- `FileWriter`
- `StringReader`
- `StringWriter`
- `CharArrayReader`
- `CharArrayWriter`

The first two classes in this list work with files and the last four work inside Java, so they aren't of great use for network programming. However, aside from different constructors, these classes have pretty much the same public interface as all other reader and writer classes.

4.4.1. Writers

The `Writer` class mirrors the `java.io.OutputStream` class. It's abstract and has two protected constructors. Like `OutputStream`, the `Writer` class is never used directly;

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

instead, it is used polymorphically, through one of its subclasses. It has five `write()` methods as well as a `flush( )` and a `close( )` method:

```
protected Writer( )
protected Writer(Object lock)
public abstract void write(char[] text, int offset, int length)
 throws IOException
public void write(int c) throws IOException
public void write(char[] text) throws IOException
public void write(String s) throws IOException
public void write(String s, int offset, int length) throws IOException
public abstract void flush( ) throws IOException
public abstract void close( ) throws IOException
```

The `write(char[] text, int offset, int length)` method is the base method in terms of which the other four `write( )` methods are implemented. A subclass must override at least this method as well as `flush( )` and `close()`, although most override some of the other `write( )` methods as well in order to provide more efficient implementations. For example, given a `Writer` object `w`, you can write the string "Network" like this:

```
char[] network = {'N', 'e', 't', 'w', 'o', 'r', 'k'};
w.write(network, 0, network.length);
```

The same task can be accomplished with these other methods, as well:

```
w.write(network);
for (int i = 0;  i < network.length;  i++) w.write(network[i]);
w.write("Network");
w.write("Network", 0, 7);
```

All of these examples are different ways of expressing the same thing. Which you use in any given situation is mostly a matter of convenience and taste. However, how many and which bytes are written by these lines depends on the encoding `w` uses. If it's using big-endian UTF-16, it will write these 14 bytes (shown here in hexadecimal) in this order:

```
00 4E 00 65 00 74 00 77 00 6F 00 72 00 6B
```

On the other hand, if `w` uses little-endian UTF-16, this sequence of 14 bytes is written:

```
4E 00 65 00 74 00 77 00 6F 00 72 00 6B 00
```

If `w` uses Latin-1, UTF-8, or MacRoman, this sequence of seven bytes is written:

```
4E 65 74 77 6F 72 6B
```

Other encodings may write still different sequences of bytes. The exact output depends on the encoding.

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

Writers may be buffered, either directly by being chained to a `BufferedWriter` or indirectly because their underlying output stream is buffered. To force a write to be committed to the output medium, invoke the `flush()` method:

```
w.flush( );
```

The `close( )` method behaves similarly to the `close( )` method of `OutputStream`. `close( )` flushes the writer, then closes the underlying output stream and releases any resources associated with it:

```
public abstract void close( ) throws IOException
```

After a writer has been closed, further writes throw `IOException`s.

4.4.2. OutputStreamWriter

`OutputStreamWriter` is the most important concrete subclass of `Writer`. An `OutputStreamWriter` receives characters from a Java program. It converts these into bytes according to a specified encoding and writes them onto an underlying output stream. Its constructor specifies the output stream to write to and the encoding to use:

```
public OutputStreamWriter(OutputStream out, String encoding)
 throws UnsupportedEncodingException
public OutputStreamWriter(OutputStream out)
```

Valid encodings are listed in the documentation for Sun's native2ascii tool included with the JDK and available from http://java.sun.com/j2se/1.4.2/docs/guide/intl/encoding.doc.html. If no encoding is specified, the default encoding for the platform is used. (In the United States, the default encoding is ISO Latin-1 on Solaris and Windows, MacRoman on the Mac.) For example, this code fragment writes the string

ῆμος δ᾽ἠριγένεια φάνη ῥοδοδάκτυλος᾽Ηώς in the Cp1253 Windows Greek encoding:

```
OutputStreamWriter w = new OutputStreamWriter(
 new FileOutputStream("OdysseyB.txt"), "Cp1253");
w.write("
ῆμος δ᾽ἠριγένεια φάνη ῥοδοδάκτυλος᾽Ηώς
");
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

Other than the constructors, `OutputStreamWriter` has only the usual `Writer` methods (which are used exactly as they are for any `Writer` class) and one method to return the encoding of the object:

```
public String getEncoding( )
```

4.4.3. Readers

The `Reader` class mirrors the `java.io.InputStream` class. It's abstract with two protected constructors. Like `InputStream` and `Writer`, the `Reader` class is never used directly, only through one of its subclasses. It has three `read()` methods, as well as `skip( ),close( ),ready( ),mark( ),reset( )`, and `markSupported( )` methods:

```
protected Reader( )
protected Reader(Object lock)
public abstract int read(char[] text, int offset, int length)
 throws IOException
public int read( ) throws IOException
public int read(char[] text) throws IOException
public long skip(long n) throws IOException
public boolean ready( )
public boolean markSupported( )
public void mark(int readAheadLimit) throws IOException
public void reset( ) throws IOException
public abstract void close( ) throws IOException
```

The `read(char[] text, int offset, int length)` method is the fundamental method through which the other two `read( )` methods are implemented. A subclass must override at least this method as well as `close( )`, although most will override some of the other `read( )` methods as well in order to provide more efficient implementations.

Most of these methods are easily understood by analogy with their `InputStream` counterparts. The `read()` method returns a single Unicode character as an `int` with a value from 0 to 65,535 or -1 on end of stream. The `read(char[] text)` method tries to fill the array `text` with characters and returns the actual number of characters read or -1 on end of stream. The `read(char[] text, int offset, int length)` method attempts to read `length` characters into the subarray of `text` beginning at `offset` and continuing for `length` characters. It also returns the actual number of characters read or -1 on end of stream. The `skip(long n)` method skips `n` characters. The `mark( )` and `reset( )` methods allow some readers to reset back to a marked position in the character sequence. The `markSupported( )` method tells you whether the reader supports marking and resetting. The `close( )` method closes the reader and any underlying input stream so that further attempts to read from it throw `IOException`s.

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

The exception to the rule of similarity is `ready()`, which has the same general purpose as `available( )` but not quite the same semantics, even modulo the byte-to-char conversion. Whereas `available( )` returns an `int` specifying a minimum number of bytes that may be read without blocking, `ready( )` only returns a `boolean` indicating whether the reader may be read without blocking. The problem is that some character encodings, such as UTF-8, use different numbers of bytes for different characters. Thus, it's hard to tell how many characters are waiting in the network or filesystem buffer without actually reading them out of the buffer.

`InputStreamReader` is the most important concrete subclass of `Reader`. An `InputStreamReader` reads bytes from an underlying input stream such as a `FileInputStream` or `TelnetInputStream`. It converts these into characters according to a specified encoding and returns them. The constructor specifies the input stream to read from and the encoding to use:

```
public InputStreamReader(InputStream in)
public InputStreamReader(InputStream in, String encoding)
 throws UnsupportedEncodingException
```

If no encoding is specified, the default encoding for the platform is used. If an unknown encoding is specified, then an `UnsupportedEncodingException` is thrown.

For example, this method reads an input stream and converts it all to one Unicode string using the MacCyrillic encoding:

```
public static String getMacCyrillicString(InputStream in)
 throws IOException {

  InputStreamReader r = new InputStreamReader(in, "MacCyrillic");
  StringBuffer sb = new StringBuffer( );
  int c;
  while ((c = r.read( )) != -1) sb.append((char) c);
  r.close( );
  return sb.toString( );

}
```

4.4.4. Filter Readers and Writers

The `InputStreamReader` and `OutputStreamWriter` classes act as decorators on top of input and output streams that change the interface from a byte-oriented interface to a character-oriented interface. Once this is done, additional character-oriented filters can be

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

layered on top of the reader or writer using the `java.io.FilterReader` and `java.io.FilterWriter` classes. As with filter streams, there are a variety of subclasses that perform specific filtering, including:

- `BufferedReader`
- `BufferedWriter`
- `LineNumberReader`
- `PushbackReader`
- `PrintWriter`

4.4.4.1. Buffered readers and writers

The `BufferedReader` and `BufferedWriter` classes are the character-based equivalents of the byte-oriented `BufferedInputStream` and `BufferedOutputStream` classes. Where `BufferedInputStream` and `BufferedOutputStream` use an internal array of bytes as a buffer, `BufferedReader` and `BufferedWriter` use an internal array of chars.

When a program reads from a `BufferedReader`, text is taken from the buffer rather than directly from the underlying input stream or other text source. When the buffer empties, it is filled again with as much text as possible, even if not all of it is immediately needed, making future reads much faster. When a program writes to a `BufferedWriter`, the text is placed in the buffer. The text is moved to the underlying output stream or other target only when the buffer fills up or when the writer is explicitly flushed, which can make writes much faster than would otherwise be the case.

`BufferedReader` and `BufferedWriter` have the usual methods associated with readers and writers, like `read( )`, `ready( )`, `write( )`, and `close( )`. They each have two constructors that chain the `BufferedReader` or `BufferedWriter` to an underlying reader or writer and set the size of the buffer. If the size is not set, the default size of 8,192 characters is used:

```
public BufferedReader(Reader in, int bufferSize)
public BufferedReader(Reader in)
public BufferedWriter(Writer out)
public BufferedWriter(Writer out, int bufferSize)
```

For example, the earlier `getMacCyrillicString( )` example was less than efficient because it read characters one at a time. Since MacCyrillic is a 1-byte character set, it also read bytes one at a time. However, it's straightforward to make it run faster by chaining a `BufferedReader` to the `InputStreamReader`, like this:

```
public static String getMacCyrillicString(InputStream in)
 throws IOException {

  Reader r = new InputStreamReader(in, "MacCyrillic");
  r = new BufferedReader(r, 1024);
  StringBuffer sb = new StringBuffer( );
  int c;
  while ((c = r.read( )) != -1) sb.append((char) c);
  r.close( );
  return sb.toString( );

}
```

All that was needed to buffer this method was one additional line of code. None of the rest of the algorithm had to change, since the only `InputStreamReader` methods used were the `read( )` and `close( )` methods declared in the `Reader` superclass and shared by all `Reader` subclasses, including `BufferedReader`.

The `BufferedReader` class also has a `readLine( )` method that reads a single line of text and returns it as a string:

```
public String readLine( ) throws IOException
```

This method is supposed to replace the deprecated `readLine()` method in `DataInputStream`, and it has mostly the same behavior as that method. The big difference is that by chaining a `BufferedReader` to an `InputStreamReader`, you can correctly read lines in character sets other than the default encoding for the platform. Unfortunately, this method shares the same bugs as the `readLine( )` method in `DataInputStream`, discussed earlier in this chapter. That is, `readline( )` tends to hang its thread when reading streams where lines end in carriage returns, as is commonly the case when the streams derive from a Macintosh or a Macintosh text file. Consequently, you should scrupulously avoid this method in network programs.

It's not all that difficult, however, to write a safe version of this class that correctly implements the `readLine( )` method. Example 4-1 is such a `SafeBufferedReader` class. It has exactly the same public interface as `BufferedReader`; it just has a slightly different private implementation. I'll use this class in future chapters in situations where it's extremely convenient to have a `readLine( )` method.

Example 4-1. The SafeBufferedReader class

```
package com.macfaq.io;

import java.io.*;
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
public class SafeBufferedReader extends BufferedReader {

  public SafeBufferedReader(Reader in) {
    this(in, 1024);
  }

  public SafeBufferedReader(Reader in, int bufferSize) {
    super(in, bufferSize);
  }

  private boolean lookingForLineFeed = false;

  public String readLine( ) throws IOException {
    StringBuffer sb = new StringBuffer("");
    while (true) {
      int c = this.read( );
      if (c == -1) { // end of stream
        if (sb.equals("")) return null;
        return sb.toString( );
      }
      else if (c == '\n') {
        if (lookingForLineFeed) {
          lookingForLineFeed = false;
          continue;
        }
        else {
          return sb.toString( );
        }
      }
      else if (c == '\r') {
        lookingForLineFeed = true;
        return sb.toString( );
      }
      else {
        lookingForLineFeed = false;
        sb.append((char) c);
      }
    }
  }

}
```

The `BufferedWriter( )` class adds one new method not included in its superclass, called `newLine( )`, also geared toward writing lines:

```
public void newLine( ) throws IOException
```

This method inserts a platform-dependent line-separator string into the output. The `line.separator` system property determines exactly what the string is: probably a linefeed on Unix and Mac OS X, a carriage return on Mac OS 9, and a carriage return/linefeed pair on Windows. Since network protocols generally specify the required line-terminator, you should not use this method for network programming. Instead, explicitly write the line-terminator the protocol requires.

4.4.4.2. LineNumberReader

`LineNumberReader` is a subclass of `BufferedReader` that keeps track of the current line number. This can be retrieved at any time with the `getLineNumber( )` method:

```
public int getLineNumber( )
```

By default, the first line number is 0. However, the number of the current line and all subsequent lines can be changed with the `setLineNumber( )` method:

```
public void setLineNumber(int lineNumber)
```

This method adjusts only the line numbers that `getLineNumber( )` reports. It does not change the point at which the stream is read.

The `LineNumberReader`'s `readLine( )` method shares the same bug as `BufferedReader` and `DataInputStream`'s, and is not suitable for network programming. However, the line numbers are also tracked if you use only the regular `read( )` methods, and these do not share that bug. Besides these methods and the usual `Reader` methods, `LineNumberReader` has only these two constructors:

```
public LineNumberReader(Reader in)
public LineNumberReader(Reader in, int bufferSize)
```

Since `LineNumberReader` is a subclass of `BufferedReader`, it has an internal character buffer whose size can be set with the second constructor. The default size is 8,192 characters.

4.4.4.3. PushbackReader

The `PushbackReader` class is the mirror image of the `PushbackInputStream` class. As usual, the main difference is that it pushes back chars rather than bytes. It provides three `unread( )` methods that push characters onto the reader's input buffer:

```
public void unread(int c) throws IOException
public void unread(char[] text) throws IOException
public void unread(char[] text, int offset, int length)
 throws IOException
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

The first `unread( )` method pushes a single character onto the reader. The second pushes an array of characters. The third pushes the specified subarray of characters, starting with `text[offset]` and continuing through `text[offset+length-1]`.

By default, the size of the pushback buffer is only one character. However, the size can be adjusted in the second constructor:

```
public PushbackReader(Reader in)
public PushbackReader(Reader in, int bufferSize)
```

Trying to unread more characters than the buffer will hold throws an `IOException`.

4.4.4.4. PrintWriter

The `PrintWriter` class is a replacement for Java 1.0's `PrintStream` class that properly handles multibyte character sets and international text. Sun originally planned to deprecate `PrintStream` in favor of `PrintWriter` but backed off when it realized this step would invalidate too much existing code, especially code that depended on `System.out`. Nonetheless, new code should use `PrintWriter` instead of `PrintStream`.

Aside from the constructors, the `PrintWriter` class has an almost identical collection of methods to `PrintStream`. These include:

```
public PrintWriter(Writer out)
public PrintWriter(Writer out, boolean autoFlush)
public PrintWriter(OutputStream out)
public PrintWriter(OutputStream out, boolean autoFlush)
public void flush( )
public void close( )
public boolean checkError( )
protected void setError( )
public void write(int c)
public void write(char[] text, int offset, int length)
public void write(char[] text)
public void write(String s, int offset, int length)
public void write(String s)
public void print(boolean b)
public void print(char c)
public void print(int i)
public void print(long l)
public void print(float f)
public void print(double d)
public void print(char[] text)
public void print(String s)
public void print(Object o)
public void println( )
public void println(boolean b)
public void println(char c)
public void println(int i)
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
public void println(long l)
public void println(float f)
public void println(double d)
public void println(char[] text)
public void println(String s)
public void println(Object o)
```

Most of these methods behave the same for `PrintWriter` as they do for `PrintStream`. The exceptions are the four `write( )` methods, which write characters rather than bytes; also, if the underlying writer properly handles character set conversion, so do all the methods of the `PrintWriter`. This is an improvement over the noninternationalizable `PrintStream` class, but it's still not good enough for network programming. `PrintWriter` still has the problems of platform dependency and minimal error reporting that plague `PrintStream`.

It isn't hard to write a `PrintWriter` class that does work for network programming. You simply have to require the programmer to specify a line separator and let the `IOException`s fall where they may. Example 4-2 demonstrates. Notice that all the constructors require an explicit line-separator string to be provided.

Example 4-2. SafePrintWriter

```
/*
 * @(#)SafePrintWriter.java 1.0 04/06/28
 *
 * Placed in the public domain
 * No rights reserved.
 */
package com.macfaq.io;

import java.io.*;

/**
 * @version   1.1, 2004-06-28
 * @author  Elliotte Rusty Harold
 * @since Java Network Programming, 2nd edition
 */
public class SafePrintWriter extends Writer {

  protected Writer out;

  private boolean autoFlush = false;
  private String lineSeparator;
  private boolean closed = false;

  public SafePrintWriter(Writer out, String lineSeparator) {
    this(out, false, lineSeparator);
  }
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
    public SafePrintWriter(Writer out, char lineSeparator) {
      this(out, false, String.valueOf(lineSeparator));
    }

    public SafePrintWriter(Writer out, boolean autoFlush, String lineSeparator) {
      super(out);
      this.out = out;
      this.autoFlush = autoFlush;
      if (lineSeparator == null) {
        throw new NullPointerException("Null line separator");
      }
      this.lineSeparator = lineSeparator;
    }

    public SafePrintWriter(OutputStream out, boolean autoFlush,
     String encoding, String lineSeparator)
     throws UnsupportedEncodingException {
      this(new OutputStreamWriter(out, encoding), autoFlush, lineSeparator);
    }

    public void flush( ) throws IOException {

      synchronized (lock) {
        if (closed) throw new IOException("Stream closed");
        out.flush( );
      }

    }

    public void close( ) throws IOException {

      try {
        this.flush( );
      }
      catch (IOException ex) {
      }

      synchronized (lock) {
        out.close( );
        this.closed = true;
      }

    }

    public void write(int c) throws IOException {

      synchronized (lock) {
        if (closed) throw new IOException("Stream closed");
        out.write(c);
      }

    }

    public void write(char[] text, int offset, int length) throws IOException {
      synchronized (lock) {
        if (closed) throw new IOException("Stream closed");
        out.write(text, offset, length);
      }
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
    }

    public void write(char[] text) throws IOException {

      synchronized (lock) {
        if (closed) throw new IOException("Stream closed");
        out.write(text, 0, text.length);
      }

    }

    public void write(String s, int offset, int length) throws IOException {

      synchronized (lock) {
        if (closed) throw new IOException("Stream closed");
        out.write(s, offset, length);
      }

    }

    public void print(boolean b) throws IOException {
      if (b) this.write("true");
      else this.write("false");
    }

    public void println(boolean b) throws IOException {
      if (b) this.write("true");
      else this.write("false");
      this.write(lineSeparator);
      if (autoFlush) out.flush( );
    }

    public void print(char c) throws IOException {
      this.write(String.valueOf(c));
    }

    public void println(char c) throws IOException {
      this.write(String.valueOf(c));
      this.write(lineSeparator);
      if (autoFlush) out.flush( );
    }

    public void print(int i) throws IOException {
      this.write(String.valueOf(i));
    }

    public void println(int i) throws IOException {
      this.write(String.valueOf(i));
      this.write(lineSeparator);
      if (autoFlush) out.flush( );
    }

    public void print(long l) throws IOException {
      this.write(String.valueOf(l));
    }

    public void println(long l) throws IOException {
      this.write(String.valueOf(l));
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
    this.write(lineSeparator);
    if (autoFlush) out.flush( );
  }

  public void print(float f) throws IOException {
    this.write(String.valueOf(f));
  }

  public void println(float f) throws IOException {
    this.write(String.valueOf(f));
    this.write(lineSeparator);
    if (autoFlush) out.flush( );
  }

  public void print(double d) throws IOException {
    this.write(String.valueOf(d));
  }

  public void println(double d) throws IOException {
    this.write(String.valueOf(d));
    this.write(lineSeparator);
    if (autoFlush) out.flush( );
  }

  public void print(char[] text) throws IOException {
    this.write(text);
  }

  public void println(char[] text) throws IOException {
    this.write(text);
    this.write(lineSeparator);
    if (autoFlush) out.flush( );
  }

  public void print(String s) throws IOException {
    if (s == null) this.write("null");
    else this.write(s);
  }

  public void println(String s) throws IOException {
    if (s == null) this.write("null");
    else this.write(s);
    this.write(lineSeparator);
    if (autoFlush) out.flush( );
  }

  public void print(Object o) throws IOException {
    if (o == null) this.write("null");
    else this.write(o.toString( ));
  }

  public void println(Object o) throws IOException {
    if (o == null) this.write("null");
    else this.write(o.toString( ));
    this.write(lineSeparator);
    if (autoFlush) out.flush( );
  }
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
    public void println( ) throws IOException {
      this.write(lineSeparator);
      if (autoFlush) out.flush( );
    }

  }
```

This class actually extends `Writer` rather than `FilterWriter`, unlike `PrintWriter`. It could extend `FilterWriter` instead; however, this would save only one field and one line of code, since this class needs to override every single method in `FilterWriter` (`close( )`, `flush( )`, and all three `write( )` methods). The reason for this is twofold. First, the `PrintWriter` class has to be much more careful about synchronization than the `FilterWriter` class. Second, some of the classes that may be used as an underlying `Writer` for this class, notably `CharArrayWriter`, do not implement the proper semantics for `close( )` and allow further writes to take place even after the writer is closed. Consequently, programmers have to handle the checks for whether the stream is closed in this class rather than relying on the underlying `Writer out` to do it for them.

This chapter has been a whirlwind tour of the `java.io` package, covering the bare minimum you need to know to write network programs. For a more detailed and comprehensive look with many more examples, check out my other book in this series, *Java I/O* (O'Reilly).

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004