I/O in Java is built on streams. Input streams read data; output streams write data.

Different stream classes (FileInputStream 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.

Power of polymorphism. If you know how to use the superclass, you know how to use all the subclasses, too.

Streams are synchronous; that is, when a program 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.

**Output Streams - java.io.OutputStream:**

This class provides the fundamental methods needed to write data:

* 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

write(int b):

* This method takes an integer from 0 to 255 as an argument and writes the corresponding byte (unsigned)to the output stream.
* Java doesn’t have an unsigned byte data type, so an int has to be used here instead.
* 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.

Ex: Server sends 72-character lines containing printable ASCII characters between 33 and 126.

The first line contains characters 33 through 104, sorted.

The second line contains characters 34 through 105.

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).

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

**int** firstChar = 33;

**int** numberOfChar = 94;

**int** charPerLine = 72;

**int** start = firstChar;

**while** (**true**) { /\* infinite loop \*/

**for** (**int** i = start; i < start + charPerLine; i++) {

**out.write**(( (i - firstChar) % numberOfChar) + firstChar);

}

out.write('\r'); // carriage return

out.write('\n'); // linefeed

start = ((start + 1) - firstChar) % numberOfChar + firstChar;

}

}

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 very inefficient!!

Every TCP segment contains at least 40 bytes of overhead for routing and error correction. If each byte is sent by itself, you are stuffing the network with 41 times more data than you think you are! if you have more than one byte ready to go, it’s not a bad idea to send them all at once.

How to do this? Using

* write(byte[] data) or
* write(byte[] data, int offset, int length)

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

**int** firstChar = 33;

**int** noOfChar = 94;

**int** charPerLine = 72;

**int** start = firstChar;

**byte**[] line = **new** **byte**[charPerLine + 2];

// the +2 is for the carriage return and linefeed

**while** (**true**) { /\* infinite loop \*/

**for** (**int** i = start; i < start + charPerLine; i++) {

line[i - start] = (**byte**)((i - firstChar) % noOfChar + firstChar);

}

line[72] = (**byte**) '\r'; // carriage return

line[73] = (**byte**) '\n'; // line feed

**out.write(line);**

start = ((start + 1) - firstChar) % noOfChar + firstChar;

}

}

if you are done writing data, it’s important to flush the output stream. Why?

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 flush() method breaks the deadlock by forcing the buffered stream to send its data even if the buffer isn’t yet full.

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

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.

If the stream derives from a network connection, then closing the stream terminates the connection.

Once an output stream has been closed, further writes to it throw IOExceptions.

Failure to close a stream in a long-running program can leak file handles, network ports, and other resources. Consequently, in Java 6 and earlier, it’s wise to close the stream in a finally block.

**Input Streams - 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 only these 6 methods to read data from particular media. The instance of the subclass can be used transparently as an instance of its superclass. No specific knowledge of the subclass is required.

That’s polymorphism at work once again !!

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

**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, a signed byte from –128 to 127 instead of the unsigned byte from 0 to 255.

Reading a byte at a time is inefficient. Consequently, there are two overloaded read() methods

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.

These methods attempt to fill the array, but this attempt may fail in several ways.

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. 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.

**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**; // end of stream

bytesRead += result;

}

This technique is especially crucial for network streams. Because 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.

All three read() methods return –1 to signal the end of the stream.

Note that, –1 is never placed in the array. The array only contains actual data.

**available()**

**int** bytesAvailable = in.available();

**byte**[] input = **new** **byte**[bytesAvailable];

**int** bytesRead = in.read(input, 0, bytesAvailable);

you can use the available() method to determine how many bytes can be read without blocking. available() returns the minimum number of bytes you can read. You may in fact be able to read more, but you will be able to read at least as many bytes as available() suggests.

you may want to skip over data without reading it. The skip() method accomplishes this task.

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 IOExceptions.