Chapter 2

What do network programs do?

* Simple input & output
  + Moving bytes from one system to another

Input and Output (I/O) in Java is different than most other languages

* Built on streams
* Input streams read data
  + All input streams use same basic to read data
* Output streams write data
  + All output streams use same basic to write data
* Different stream classes write different sources of data
* Filter streams can be chained to either input stream or output stream
* Filters modify data as its being read / written
  + ex: encryption, compression, etc.
* Readers and writers can be chained to I/O streams to allow programs to read & write text, rather than bytes
* Streams are synchronous (meaning: when a program, or thread, asks a stream to read or write a piece of data, it waits for the data to be read or written before doing anything)

Output Streams

* Java’s basic output class is java.io.OutputStream, and the class provides fundamental methods needed to write data
* Subclasses of OutputStream use the same methods to write data onto a unique source
  + FileOutputStream uses the methods to write data to a file
  + TelnetOutputStream uses the methods to write data onto a network connection
  + ByteArrayOutputStream uses the methods to write data into an expandable byte array
  + More methods exist for different purposes
* Knowing how to use the superclass will also transfer when attempting to use subclasses (thanks polymorphism!)
* OutputStream’s fundamental method is write(int b)
  + Takes an integer (0-255) as an argument & writes corresponding byte to the output stream
  + Method is declared abstract because subclasses use the method differently, depending on the purpose of the subclass
  + Writes an unsigned byte, despite taking int as argument
    - Java doesn’t have an unsigned byte data type, so int is used instead
  + If an int outside of 0-255 is passed, the least significant byte of the number is written and the remaining three bytes are ignored (due to casting an int to a byte)
  + Writing a single byte at a time is inefficient; TCP segments contain at least 40 bytes of overhead for routing & error connections
    - Better to send data more than one byte at a time
  + Streams can also be buffered in software (both directly in Java code and network hardware)
    - Done typically by chaining a BufferedOutputStream or a BufferedWriter to the underlying stream
  + When done writing data, it’s important to flush the output stream
    - Helps prevent the stream from waiting an indefinite amount of time for data to be done written from the buffer limit size
  + Failing to flush when needed can lead to unpredictable, unrepeatable program hangs, which tend to be tough to diagnose
  + Streams should be flushed before they are closed
  + When done with a stream, close it by invoking close()
    - Releases any resources associated with stream, such as file handlers or ports
    - Closing a stream deriving from a network connection will terminate the connection
  + Further writes after closing the stream will throw IOExceptions
  + Some streams may still allow you to continue working with an object
    - A closed ByteArrayOutputStream can still be converted to an actual byte array
    - A closed DigestOutputStream can still return its digest
  + Failure to close a stream in a long-running program can leak file handles, network ports, and other resources
  + Closing streams had to be previously done in finally blocks of code, but is no longer needed in Java 7 and up. Java automatically invokes close() on any AutoCloseable objects declared inside the argument list of the try block

Input Streams

* Java’s basic input class is java.io.InputStream, and the class provides fundamental methods needed to read data as raw bytes
* Subclasses of InputStream use the same methods to read data from a unique source
  + TelnetInputStream reads data from a network connection
  + ByteArrayInputStream reads data from an array of bytes
* Knowing how to use the superclass will also transfer when attempting to use subclasses (thanks polymorphism!)
* The basic method of InputStream is the no arguments read() method
  + Reads a single byte of data from input stream’s source and returns it as an int from 0-255
  + End of stream is indicated by returning a -1
  + Waits and blocks execution of any code that follows it until a byte of data is available and ready to be read
  + Declared abstract because subclasses use the method differently, depending on the purpose of the subclass
* I/O can be slow, so if your program needs to do anything else that’s more important, try putting I/O in its own thread
* Reading a byte at a time is as inefficient as writing data one byte at a time
* Two overloaded read() methods that fill a specified array with multiple bytes of data read from the stream
  + End of streams are noted by the methods returning -1
  + First method read(byte[] input) attempts to fill the specified array input
  + Second method read(byte[] input, int offset, int length) attempts to fill the specified subarray of input, starting at offset and continuing for length bytes
  + Attempts from either method may fail in several ways:
    - Disconnection from the network due to an outage of some sort (causing an IOException)
    - Attempt of reading 1,024 bytes from a network connection, but only having 512 arrive from server while the rest are in transit
* If a file is available at all, then it’s likely the bytes of a file area also available
* Since networks move more slowly than CPUs, it is easy for a program to empty a network buffer before all data arrives
* The methods returning -1 to signify end of streams don’t actually place -1 in the array
  + Array only contains actual data
* The available() method can be called to determine how many bytes can be read without blocking, if it is undesirable to wait until are bytes are immediately available
* You would end up reading as many bytes as available() suggests

Marking and Resetting

* The InputStream class has three commonly used methods that allow programs to back up and reread data that’s been previously read
  + mark(int readAheadLimit)
    - Marks the current position in the stream in order to reread data
    - Can only be one mark in the stream at any given time; making a second mark overrides the first one
  + reset() throws IOException
    - Resets the stream for the marked position
  + markSupported()
    - Checks to ensure that the stream supports marking and resetting functions
* The only java.io input streams that always support marking are BufferedInputStream and ByteArrayInputStream; TelnetInputStream can support marking if chained to a buffered input stream first

Filter Streams

* InputStream and OutputStream are raw classes; ready and write bytes singly or in groups
* Java provides a number of filter classes that can be attached to raw streams to translate raw bytes to and from other formats, such as ZIPs, text, and integers
* Filters come in two versions:
  + Filter streams
    - Works primarily with raw data as bytes, by compressing data or interpreting it as binary numbers
  + Readers and writers
    - Handle the special case of text in a variety of encodings such as UTF-8 and ISO 8859-1
* Filters are organized in a chain. Each link in the chain receives data from the previous filter/stream and passes data along to the next link in the chain. (image of figure 2-2)
* 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
* Certain stream classes, such as BufferedInputStream and BufferedOutputStream, only have these methods
* Filter streams may add public methods with additional purposes, sometimes intended to be used in addition to the typical read() and write() methods, and sometimes even replacing the original interface

Chaining Filters Together

* Filters are connected to streams by their constructors (put code on page 37 below)