# Files and Streams

## File

The File class represents a file or directory in the file system. It provides platform independent ways to test file attributes, list directories, etc.

You do not open the File object directly, instead use input and output streams...

## Stream Abstraction

Java presents traditional streamed file and network input/output through InputStream and OutputStream objects

For example, to read a file, we have an in memory input stream object connected to the file. We send a series of read() messages to the input stream object to gradually see the bytes of the file.

## InputStream / OutputStream

These are the base class streams -- they have very few features beyond simple read() and write().

These deal with plain bytes -- usually you interact with them through an intermediate class or subclass that adds features (see below).

## Stream Variants

FileInputStream and FileOutputStream are subclasses specialized to connect to files in the file system. Their constructors take File objects or filenames.

ByteArrayInputStream and ByteArrayOutputStream read and write to an in-memory array of bytes for their storage. You could use them, for example, to re-use your file-writing code to create an in-memory backup for undo.

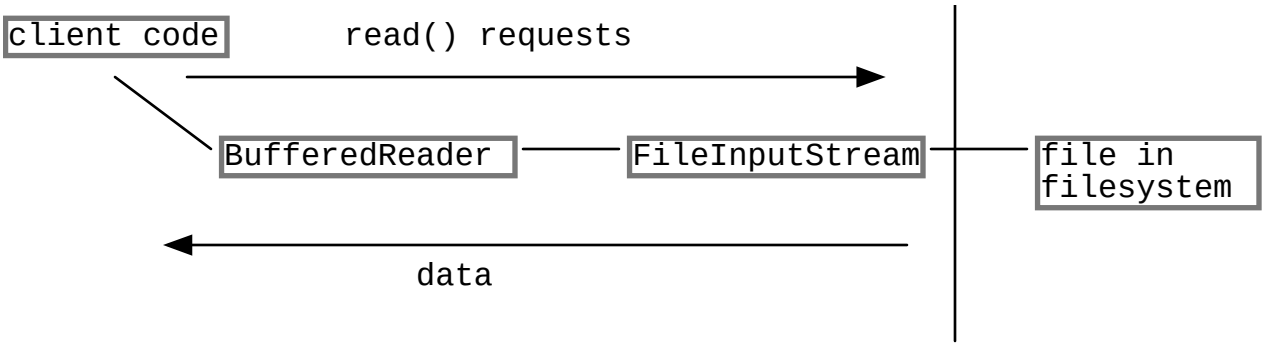
In particular, the default InputStream and OutputStream deal with 1 byte at a time. You almost always want to go through a BufferedInputStream or BufferedOutputStream which buffer data internally and so are much more efficient. In retrospect, it is probabbly a design error that buffering is an option that must be added in this way -- it should have been on by default.

## Reader / Writer

The Reader/Writer classes take the **raw bytes** of plain stream, and convert them to **unicode Char and String** data.

To read or write text, we create a reader or writer layer connected to the basic stream.

The InputStream and OutputStream classes are quite minimal -- you usually interact with them through one or more other classes to get more features. For example. to read text data, you create a BufferedReader object on the InputStream. The client sends read() messages to the BufferedReader, which relays them to the InputStream.




# Streams vs. Threads

The stream classes have a simple but effective 1-thread per stream design.

When a thread sends read() to a stream, if the data is not ready, the thread blocks in the call to read(). When the data is there, the thread unblocks and the call to read() returns. The reading or writing code does not need to do anything special to get this behavior -- the blocking is down in the guts of the read/write machinery.

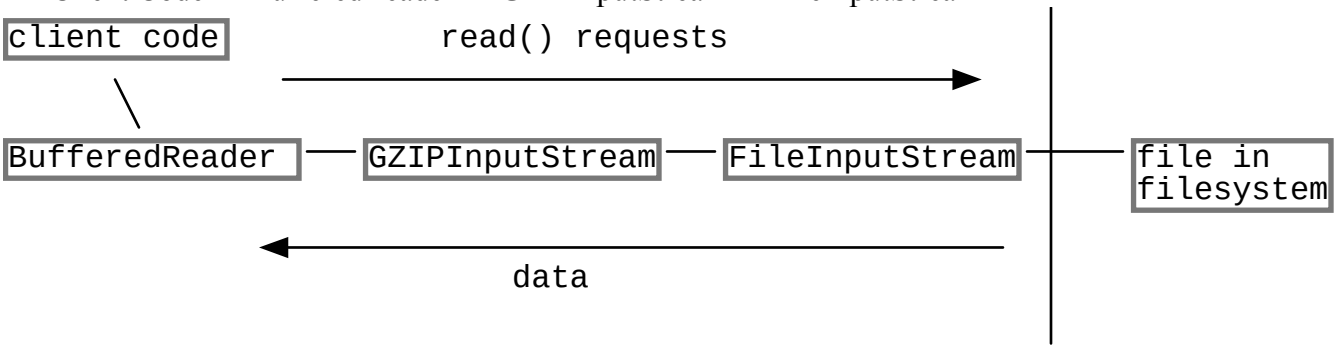
In java, if you want to read from 10 network sockets at once, the easiest design will be to create 10 threads, and send them all off to read -- they will block as needed.

In java 1.4 some "non-blocking" io options were added, but they are not appropriate for most code.

# Layering Trick

It is possible to layer stream object, so each layer accepts read() or write() messages, and then passes them on to the next layer, which could be another stream of some sort.

1. So for ordinary reading the client code sends read() to the input stream which reads from the file
   Client Code -> BufferedReader -> FileInputStream
2. Layered -- suppose the file is GZIP compressed, then we can use a GZIPInputStream which takes in compressed data and provides it in its plain, uncompressed form. We insert the GZIPInputStream as an additional layer, and the client code is unaware that the data is being decompressed
   Client Code -> BufferedReader -> GZIPInputStream -> FileInputStream



# IMHO -- Problems With The Stream Classes

I think, ultimately, the design of the stream classes is not that great. If you really understand them, they are capable of some cute tricks, such as the GZIP trick above. However, for a novice trying to do very common operations, they are not that easy to use. The stream classes are elegant, but they violate the "easy things should be easy, hard things should be possible" rule.

I think too much energy was put into the overall elegance of the stream architecture and its capacity to handle many cases, and too little emphasis was put into a practical study of the common client use cases, and how to make them easy and obvious. This is perhaps a risk for software architects who are immersed in an interesting area, and loose sight of the common, and perhaps not that interesting, needs of the general population.

# Text Reading

Below is the standard incantation to read a text file.

We construct a FileReader object, that takes either a File object or a String filename.

We wrap that reader in a BufferedReader, which responds to a readLine() message which returns a String, or null if there is no more data.

readLine() recognizes the many different end-line conventions, and strips all the end-line chars before returning the string.

It's polite to close() the reader when done. This may help free up resources in the VM a little more quickly. However, code that forgets to close() will still work.

# Text Reading Code

```
public void readLines(String fname) {
   try {
      // Build a reader on the fname, (also works with File object)
      BufferedReader in = new BufferedReader(new FileReader(fname));

      String line;
      while ((line = in.readLine()) != null) {
         // do something with 'line'
         System.out.println(line);
      }

      in.close(); // polite
   }
   catch (IOException e) {
      e.printStackTrace();
   }
}
```

# Text Writing

Writing is pretty similar.

We construct a BufferedWriter on a FileWriter

The writer responds to print() and println() messages to write strings and chars.

# Text Writing Code

```
public void writeLines(String fname) {
   try {
      // Build a writer on the fname (also works on File objects)
      BufferedWriter out = new BufferedWriter(new FileWriter(fname));

      // Send out.print(), out.println() to write chars
      for (int i=0; i<data.size(); i++) {
         out.println( ... ith data string ... );
      }

      out.close();      // polite
   }
   catch (IOException e) {
      e.printStackTrace();
   }
}
```

# HTTP

Java has built-in URL and HttpURLConnection objects to support connections based on a URL.

These are good examples of modern OOP programming with a library of standard code to pull "off the shelf" to solve common problems.

The HttpURLConnection object exposes an InputStream object that the client uses to see all the HTTP data.

# URL / HTTP Code

```
/*
 Given a url string, like "http://cslibrary.stanford.edu/104/",
 Attempts to connect to the given URL and print out
 the data it sends back.
*/
public static void dumpURL(String urlString) {
   try {
      URL url = new URL(urlString);
      URLConnection conn = url.openConnection();

      InputStream stream = conn.getInputStream();
      BufferedReader in = new BufferedReader( new InputStreamReader(stream));

      String line;
      while ( (line = in.readLine()) != null) {
         System.out.println(line);
      }
      in.close();

   }
   catch (MalformedURLException e) {
      e.printStackTrace();
   }
   catch (IOException e) {
      e.printStackTrace();
   }
}
/*
 could test
```

```
  if (conn instanceof HttpURLConnection)

HttpURLConnection responds to
-getResponseCode()
-getContentType();

 Most but not all URLs upport opening a connection -- try it
 and catch the exception (e.g. mailto: won't work).
*/
```

# Reading in larger chunks

Given a reader, can get chars one char or one one line at a time.
For better performance, read chars in larger, 1000 or 4000 byte chunks.
In this example, we read into a char array, and then append the chars into a StringBuffer.
A similar strategy could be used to read binary (not char) data in larger chunks for each call to read().

```
/*
 Given a reader, reads all its chars into a StringBuffer.
*/
public static StringBuffer readIntoBuffer(Reader in) throws IOException {
   // char array for temporary storage
   char[] chars = new char[512];
   int len;

   StringBuffer buff = new StringBuffer();

   // call read() to put chars into the array
   // read() returns -1 on EOF
   while ((len = in.read(chars, 0, chars.length)) >= 0) {
      // append the chars into the String Buffer
      buff.append(chars, 0, len);
   }
   return(buff);
}
```

For performance, it would be best if the char array and the StringBuffer did not need to be allocated anew each time the method runs. Also, we could call read() in a way to append into the one char array, instead of copying the data over to the StringBuffer on every iteration, but in that case, we would need to manage the size of the car array to ensure it was big enough as we went.