The java.io.File class

The java.io.File class *represents* a file on disk, but doesn't actually represent the *contents* of the file. What? Think of a File object as something more like a *pathname* of a file (or even a *directory*) rather than The Actual File Itself. The File class does not, for example, have methods for reading and writing. One VERY useful thing about a File object is that it offers a much safer way to represent a file than just using a String file name. For example, most classes that take a String file name in their constructor (like FileWriter or FileInputStream) can take a File object instead. You can construct a File object, verify that you've got a valid path, etc. and then give that File object to the FileWriter or FileInputStream.

A File object represents the name and path of a file or directory on disk, for example:

/Users/Kathy/Data/GameFile.txt

But it does NOT represent, or give you access to, the data *in* the file!

An address is NOT the same as the actual house! A File object is like a street address... it ~~represents~~ the name and location of a particular file, but it isn't the file itself.

Some things you can do with a File object:

① **Make a File object representing an existing file**

```
File f = new File("MyCode.txt");
```

② **Make a new directory**

```
File dir = new File("Chapter7");
dir.mkdir();
```

③ **List the contents of a directory**

```
if (dir.isDirectory()) {
      String[] dirContents = dir.list();
      for (int i = 0; i < dirContents.length; i++) {
          System.out.println(dirContents[i]);
      }
}
```

④ **Get the absolute path of a file or directory**

```
System.out.println(dir.getAbsolutePath());
```

⑤ **Delete a file or directory (returns true if successful)**

```
boolean isDeleted = f.delete();
```

A File object represents the filename "GameFile.txt"

GameFile.txt

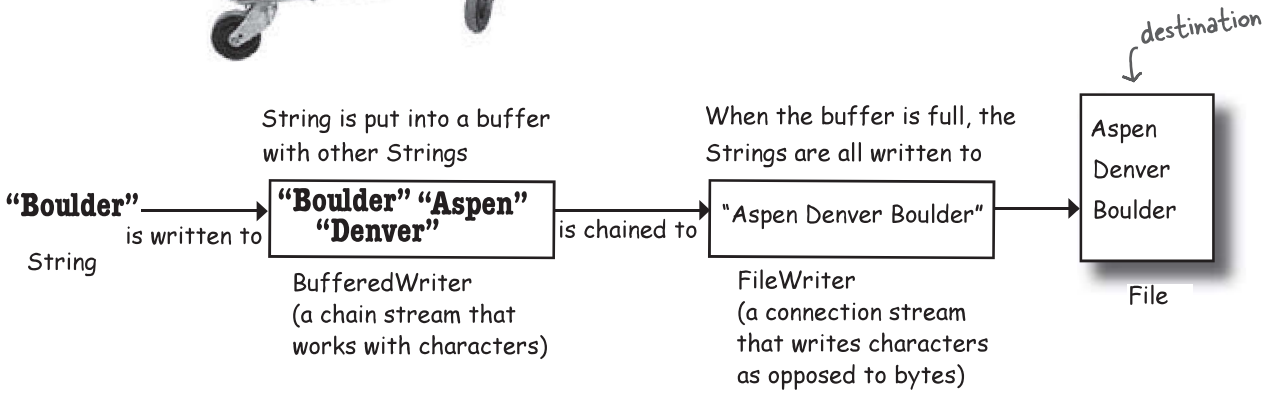
```
50,Elf,bow, sword,dust
200,Troll,bare hands,big ax
120,Magician,spells,invisibility
```

A File object does NOT represent (or give you direct access to) the data inside the file!

The beauty of buffers

If there were no buffers, it would be like shopping without a cart. You'd have to carry each thing out to your car, one soup can or toilet paper roll at a time.

buffers give you a temporary holding place to group things until the holder (like the cart) is full. You get to make far fewer trips when you use a buffer.




```
BufferedWriter writer = new BufferedWriter(new FileWriter(aFile));
```

Notice that we don't even need to keep a reference to the FileWriter object. The only thing we care about is the BufferedWriter, because that's the object we'll call methods on, and when we close the BufferedWriter, it will take care of the rest of the chain.

The cool thing about buffers is that they're *much* more efficient than working without them. You can write to a file using FileWriter alone, by calling write(someString), but FileWriter writes each and every thing you pass to the file each and every time. That's overhead you don't want or need, since every trip to the disk is a Big Deal compared to manipulating data in memory. By chaining a BufferedWriter onto a FileWriter, the BufferedWriter will hold all the stuff you write to it until it's full. *Only when the buffer is full will the FileWriter actually be told to write to the file on disk.*

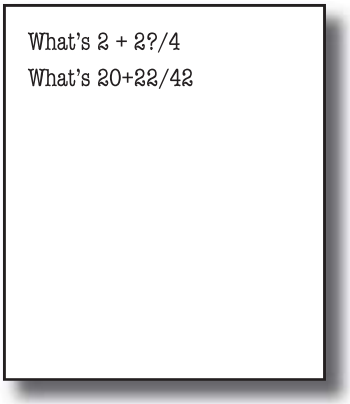
If you do want to send data *before* the buffer is full, you do have control. ***Just Flush It.*** Calls to writer.flush() say, "send whatever's in the buffer, ***now***!"

Reading from a Text File

Reading text from a file is simple, but this time we'll use a File object to represent the file, a FileReader to do the actual reading, and a BufferedReader to make the reading more efficient.

The read happens by reading lines in a *while* loop, ending the loop when the result of a readLine() is null. That's the most common style for reading data (pretty much anything that's not a Serialized object): read stuff in a while loop (actually a while loop *test*), terminating when there's nothing left to read (which we know because the result of whatever read method we're using is null).

A file with two lines of text.



MyText.txt

Don't forget the import.

```
import java.io.*;

class ReadAFile {
    public static void main (String[] args) {

       try {
          File myFile = new File("MyText.txt");
          FileReader fileReader = new FileReader(myFile);

          BufferedReader reader = new BufferedReader(fileReader);

          String line = null;

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

       } catch(Exception ex) {
            ex.printStackTrace();
       }
    }
}
```

A FileReader is a connection stream for characters, that connects to a text file

Chain the FileReader to a BufferedReader for more efficient reading. It'll go back to the file to read only when the buffer is empty (because the program has read everything in it).

Make a String variable to hold each line as the line is read

This says, "Read a line of text, and assign it to the String variable 'line'. While that variable is not null (because there WAS something to read) print out the line that was just read."

Or another way of saying it, "While there are still lines to read, read them and print them."