File Navigation and I/O

I/O is a huge topic in general, and the Java APIs that deal with I/O in one fashion or another are correspondingly huge. A general discussion of I/O could include topics such as file I/O, console I/O, thread I/O, high-performance I/O, byte-oriented I/O, character-oriented I/O, I/O filtering and wrapping, serialization, and more.
Here's a summary of the I/O classes you'll need to understand:

File The API says that the class File is "An abstract representation of file and directory pathnames." The File class isn't used to actually read or write data; it's used to work at a higher level, making new empty files, searching for files, deleting files, making directories, and working with paths.

FileReader This class is used to read character files. Its `read()` methods are fairly low-level, allowing you to read single characters, the whole stream of characters, or a fixed number of characters. FileReaders are usually *wrapped* by higher-level objects such as BufferedReaders, which improve performance and provide more convenient ways to work with the data.

BufferedReader This class is used to make lower-level Reader classes like FileReader more efficient and easier to use. Compared to FileReaders, BufferedReaders read relatively large chunks of data from a file at once, and keep this data in a buffer. When you ask for the next character or line of data, it is retrieved from the buffer, which minimizes the number of times that time-intensive, file read operations are performed. In addition, BufferedReader provides more convenient methods such as `readLine()`, that allow you to get the next line of characters from a file.

FileWriter This class is used to write to character files. Its `write()` methods allow you to write character(s) or Strings to a file. FileWriters are usually *wrapped* by higher-level Writer objects such as BufferedWriters or PrintWriters, which provide better performance and higher-level, more flexible methods to write data.

BufferedWriter This class is used to make lower-level classes like FileWriters more efficient and easier to use. Compared to FileWriters, BufferedWriters write relatively large chunks of data to a file at once, minimizing the number of times that slow, file writing operations are performed. In addition, the BufferedWriter class provides a `newLine()` method that makes it easy to create platform-specific line separators automatically.

PrintWriter This class has been enhanced significantly in Java 5. Because of newly created methods and constructors (like building a PrintWriter with a File or a String), you might find that you can use PrintWriter in places where you previously needed a Writer to be wrapped with a FileWriter and/or a BufferedWriter. New methods like `format()`, `printf()`, and `append()` make PrintWriters very flexible and powerful.

Creating Files Using Class File

Objects of type File are used to represent the actual files (but not the data in the files) or directories that exist on a computer's physical disk. Just to make sure we're clear, when we talk about an object of type File, we'll say File, with a capital F.

When we're talking about what exists on a hard drive, we'll call it a file with a lowercase f (unless it's a variable name in some code). Let's start with a few basic examples of creating files, writing to them, and reading from them. First, let's create a new file and write a few lines of data to it:

```
import java.io.*;
class Writer1 {
public static void main(String [] args) {
        File file = new File("fileWrite1.txt"); // There's no
        // file yet!
}
}
```

If you compile and run this program, when you look at the contents of your current directory, you'll discover absolutely no indication of a file called `fileWrite1.txt`. When you make a new instance of the class File, *you're not yet making an actual file, you're just creating a filename*. Once you have a File *object*, there are several ways to make an actual file. Let's see what we can do with the File object
we just made:

```
import java.io.*;

class Writer1
{
public static void main(String [] args) {
try { // warning: exceptions possible
        boolean newFile = false;
        File file = new File("fileWrite1.txt"); // it's only an object
        System.out.println(file.exists()); // look for a real file
        newFile = file.createNewFile(); // maybe create a file!
        System.out.println(newFile); // already there?
        System.out.println(file.exists()); // look again
} catch(IOException e) { }
}
}
```

This produces the output

```
false
true
true
```

And also produces an empty file in your current directory. If you run the code a *second* time you get the output

```
true
false
true
```

Let's examine these sets of output:

First execution The first call to `exists()` returned `false`, which we expected…remember `new File()` doesn't create a file on the disk! The `createNewFile()` method created an actual file, and returned `true`, indicating that a new file was created, and that one didn't already exist. Finally, we called `exists()` again, and this time it returned `true`, indicating that the file existed on the disk.

Second execution The first call to `exists()` returns `true` because we built the file during the first run. Then the call to `createNewFile()` returns `false` since the method didn't create a file this time through. Of course, the last call to `exists()` returns `true`. A couple of other new things happened in this code. First, notice that we had to put our file creation code in a try/catch. This is true for almost all of the file I/O code you'll ever write. I/O is one of those inherently risky things. We're keeping it simple for now, and ignoring the exceptions, but we still need to follow the handleor- declare rule since most I/O methods declare checked exceptions. We'll talk more about I/O exceptions later. We used a couple of File's methods in this code:

boolean exists() This method returns `true` if it can find the actual file.
boolean createNewFile() This method creates a new file if it doesn't already exist.

Using FileWriter and FileReader

In practice, you probably won't use the FileWriter and FileReader classes without wrapping them. That said, let's go ahead and do a little "naked" file I/O:

```
import java.io.*;
class Writer2 {
public static void main(String [] args) {
char[] in = new char[50]; // to store input
int size = 0;
try {
        File file = new File("fileWrite2.txt"); // just an object
        FileWriter fw = new FileWriter(file); // create an actual file
// & a FileWriter obj
        fw.write("howdy\nfolks\n"); // write characters to the file
        fw.flush(); // flush before closing
        fw.close(); // close file when done

        FileReader fr = new FileReader(file); // create a FileReader object
        size = fr.read(in); // read the whole file!
        System.out.print(size + " "); // how many bytes read
        for(char c : in) // print the array
                System.out.print(c);
        fr.close(); // again, always close
} catch(IOException e) { }
}
}
```

which produces the output:

```
        12 howdy
        folks
```

Here's what just happened:

1. `FileWriter fw = new FileWriter(file)` did three things:

a. It created a FileWriter reference variable, `fw`.

b. It created a FileWriter object, and assigned it to `fw`.

c. It created an actual empty file out on the disk (and you can prove it).

2. We wrote 12 characters to the file with the `write()` method, and we did a `flush()` and a `close()`.

3. We made a new FileReader object, which also opened the file on disk for reading.

4. The `read()` method read the whole file, a character at a time, and put it into the `char[] in`.

5. We printed out the number of characters we read size, and we looped through the `in` array printing out each character we read, then we closed the file. Before we go any further let's talk about `flush()` and `close()`. When you write data out to a stream, some amount of buffering will occur, and you never know for sure exactly when the last of the data will actually be sent. You might perform many

write operations on a stream before closing it, and invoking the `flush()` method guarantees that the last of the data you thought you had already written actually gets out to the file. Whenever you're done using a file, either reading it or writing to it, you should invoke the `close()` method. When you are doing file I/O you're using expensive and limited operating system resources, and so when you're done, invoking `close()` will free up those resources. Now, back to our last example. This program certainly works, but it's painful in a couple of different ways:

1. When we were writing data to the file, we manually inserted line separators (in this case `\n`), into our data.

2. When we were reading data back in, we put it into a character array. It being an array and all, we had to declare its size beforehand, so we'd have been in trouble if we hadn't made it big enough! We could have read the data in one character at a time, looking for the end of file after each `read`(), but that's pretty painful too. Because of these limitations, we'll typically want to use higher-level I/O classes like BufferedWriter or BufferedReader in combination with FileWriter or FileReader.

Combining I/O classes

Java's entire I/O system was designed around the idea of using several classes in combination. Combining I/O classes is sometimes called *wrapping* and sometimes called *chaining*. The java.io package contains about 50 classes, 10 interfaces, and 15 exceptions. Each class in the package has a very specific purpose (creating high cohesion), and the classes are designed to be combined with each other in countless ways, to handle a wide variety of situations. When it's time to do some I/O in real life, you'll undoubtedly find yourself pouring over the java.io API, trying to figure out which classes you'll need, and how to hook them together.

Now let's say that we want to find a less painful way to write data to a file and read the file's contents back into memory. Starting with the task of writing data to a file, here's a process for determining what classes we'll need, and how we'll hook them together:

Table: java.io mini API

Java.io Class	**Extends from**	**Key methods**
File	Object	createNewFile(), delete(), exists(), isDirectory(), isFile(), list(), mkdir(), renameTo()
FileWriter	Writer	Close(), flush(), write()
BufferedWriter	Writer	Close(), flush(), newLine(), write()
PrintWriter	Writer	Close(), flush(), format(), printf(), print(), println(), write()
FileReader	Reader	Read()
BufferedReader	Readeer	Read(), readLine()

1. We know that ultimately we want to hook to a File object. So whatever other class or classes we use, one of them must have a constructor that takes an object of type File.
2. Find a method that sounds like the most powerful, easiest way to accomplish the task. When we look at BufferedWriter class it has a `newLine()` method. That sounds a little better than having to manually embed a separator after each line, but if we look further we see that PrintWriter has a method called `println()`. That sounds like the easiest approach of all, so we'll go with it.
3. When we look at PrintWriter's constructors, we see that we can build a PrintWriter object if we have an object of type file, so all we need to do to create a PrintWriter object is the following:

```
File file = new File("fileWrite2.txt"); // create a File
PrintWriter pw = new PrintWriter(file); // pass file to
// the PrintWriter constructor
```

Okay, time for a pop quiz. Prior to Java 5, PrintWriter did not have constructors that took either a String or a File. If you were writing some I/O code in Java 1.4, how would you get a PrintWriter to write data to a File

Here's one way to go about solving this puzzle: First, we know that we'll create a File object on one end of the chain, and that we want a `PrintWriter` object on the other end. PrintWriter can also be built using a Writer object. Although Writer isn't a *class* we see in the table, we can see that several other classes extend Writer, which for our purposes is just as good; any class that extends Writer is a candidate. Looking further, we can see that FileWriter has the two attributes we're looking for:

1. It can be constructed using a File.
2. It extends Writer.

Given all of this information, we can put together the following code (remember, this is a Java 1.4 example):

```
File file = new File("fileWrite2.txt"); // create a File object
FileWriter fw = new FileWriter(file); // create a FileWriter
// that will send its output to a File
PrintWriter pw = new PrintWriter(fw); // create a PrintWriter
// that will send its output to a Writer
pw.println("howdy"); // write the data
pw.println("folks");
```

At this point it should be fairly easy to put together the code to more easily read data from the file back into memory. Again, looking through the table, we see a method called `readLine()` that sounds like a much better way to read data. Going through a similar process we get the following code:

```
File file = new File("fileWrite2.txt"); // create a File object AND
// open "fileWrite2.txt"
FileReader fr = new FileReader(file); // create a FileReader to get
// data from 'file'
BufferedReader br = new BufferedReader(fr); // create a BufferReader to
// get its data from a Reader
String data = br.readLine(); // read some data
```