Java File Input and Output (I/O) Notes (Substitute Chapter 12, Version 0.9)
BE-104, Spring '05, Dr. C. S. Tritt

File input and output provides a mechanism for the "persistence" of information before and after program execution. For example, a data file can be prepared beforehand (either manually or via the use of other programs) and later "read" during the execution of a particular program (often running in batch mode). Likewise, the results generated by a program can be "written" to a file and used later in other programs or viewed in a text editor (like Notepad) or word processor (like Wordpad or Word).

Java can deal with many types of files using many types of encoding. It can deal with text files using old ASCII code or modern Unicode characters. It can also deal with binary files accessed either sequentially or in a "random" (arbitrary order) fashion. Java program can also store data in databases, but this is a somewhat different topic. This handout deals only with accessing and storing information in sequentially in simple text files.

The *File* Class

While not always necessary, the *File* class (part of the *java.io* package) provides an operating system independent way to specify, access, process and store files and folders. For example,

```
File inFile = new File("sample.data");
```

would refer to the *sample.data* file in the same folder as the program in which the line occurred. An alternate *File* constructor takes two string arguments, the first specifies the folder (like *d:/javaPrograms*) and the second the file (like *newData.txt*).

Particularly useful *File* methods include *isFile()* which returns true if the File refers to a file (as opposed to a folder (also known as a directory), *canRead()* which returns true if the file can be read from, *canWrite()* which returns true if the file can be written to, *delete()* which deletes the specified file and *exists()* which returns true if the file exists.

Simple Sequential Input

The *Scanner* class in the *java.util* package was designed for simple file input. A new scanner can be constructed using a *File* object or a *String* to specify the file to be "connected" to the scanner. For example,

```
Scanner inScan = new Scanner(inFile);
```

would create a scanner through which the contents of *inFile* could be read.

A scanner connected to a file works in the same way as a scanner connected to *System.in* except that the source of the data is the file rather than the console keyboard.

Particularly useful scanner methods include *nextInt()*, *nextDouble()*, *next()*, *hasNext()* and *close()*. The *close()* method is important in that it "finalizes" access to a particular file. Open files (those associated with scanners) use system resources (including main memory). Closing a file (by way of its associated scanner's *close()* method) frees these resources.

Simple Sequential Output

The *Formatter* class in the *java.util* package was design for simple file output. A formatter connected to a file works in the same way as a scanner connected to *System.out* except that the destination of the data is the file rather than the console screen.

There are several formatter constructors. The one taking a single File argument and the one taking a single String argument are particularly useful. Particularly useful formatter methods include *format()*, *flush()* and *close()*. The format method is the "work horse" of the class. Use it to produce neatly formatted output to a file.

In order to increase speed and efficiency, data "written" to a file is often temporarily stored in the computer's main memory. Flushing or closing a file (by way of its associated formatter's *flush()* or *close()* methods) forces the writing of all data to disk. The *close()* method also frees system resources used by the formatter and the underlying file.

IOExceptions

Unexpected events can occur during file I/O. These events include unexpected data (values not formatted as expected (for example, the word "one" rather than the digit 1 when reading an integer value)), the unexpected end of the data a file, disks becoming full, disks being removed, etc. Java provides, and in the cause of files, requires the use of a mechanism to account for ("handle") these events. This mechanism is based on "throwing" and "catching" "exceptions."

The proper use of exceptions is a rather complex subject and will not be covered in this handout. It is sufficient to add the notation "throws IOException" to the declaration of any method that perform file I/O (include those that simply associates files with particular scanners or formatters). For example the declaration of a main method that includes file I/O would be:

```
public main (Sting[] arg) throws IOException { …
```

Other File I/O Issues

The JFileChooser class provides relatively easy way to specify files for input or output using GUI style dialogs.

A full discussion of file input and output would included a discussion of streams and classes like *BufferedReader* and *PrintStream* in the *java.io* package. This material is not particularly difficult but is expansive. It suffices to say that you can do just about anything imaginable with Java file I/O provided you take the time and effort to study the *java.io* package and related topics.