Simple Files

This handout contains the information about files needed to complete Assignment #4.

The concept of a file

In a certain sense, you have been working with text files all quarter. The programs that you have written have been stored in **`.java`** files that contain the text of your Java code. Those files, however, have been part of the program structure and not part of the data. In developing practical computer-based applications, it is often important to store data permanently in a file. Files come in many different types, but the only one that you need to be concerned with in this class is the text file, containing ASCII characters.

In many ways, a text file is similar to a string. Both are ordered sequences of characters. The two critical differences are:

- *The information stored in a file is permanent*. Data in a string variable persists only as long as the variable does. Local variables disappear when the method returns, and instance variables disappear when the object goes away, which typically does not occur until the program exits. File data exists until the file is deleted.
- *Files are usually read sequentially*. When you read data from a file, you usually start at the beginning and read the characters in order. Once a character has been read, you go on to the next character until you reach the end of the file.

Key steps in working with files

1. Declare a variable of the appropriate subclass of **`Reader`** or **`Writer`**. In Math 121, the appropriate subclass will almost always be **`BufferedReader`** for input files and **`PrintWriter`** for output files.
2. Initialize that variable by calling the appropriate set of constructors, which are almost always nested. The typical calls are

   ```
   BufferedReader rd = new BufferedReader(new FileReader(filename));
   PrintWriter wr = new PrintWriter(new FileWriter(filename));
   ```

   You will need to check for errors in each of these declarations as described later in this handout.
3. Call the appropriate methods to perform the I/O operations you need. For an input file, these methods read data from the file into your program; for an output file, the methods transfer data from the program to the file. For now, the only operations you need to know are **`readLine`**, which reads a single line of text from a file just as it does from the console, and **`println`**, which acts just like the standard version except that it writes its output to a file.

4. Indicate that the file operations are complete by calling **`close`**. This operation is called **closing** the file and lets the operating system know that the input/output operations are finished.

Reading lines from a file

For the time being, the only operation you need to use to read text data from a file is the **`readLine`** method. To understand its use, imagine for a moment that you are trying to read the file **`antony.txt`**, which has the following contents taken from Marc Antony's funeral oration from *Julius Caesar:*

```
Friends, Romans, countrymen,
Lend me your ears;
I come to bury Caesar,
Not to praise him.
```

You can the open the file by calling

```
rd = new BufferedReader(new FileReader("antony.txt"));
```

which has the effect of establishing a link between the data file and the value of the variable **`rd`**. The reader maintains an internal file pointer that begins before the first character of the file; I've indicated it here using the vertical bar in the following diagram, even though there is nothing in the actual file to mark it:

```
|Friends, Romans, countrymen,
Lend me your ears;
I come to bury Caesar,
Not to praise him.
```

If you then read a line by calling

```
line = rd.readLine();
```

the string variable **`line`** will be set to the string

```
"Friends, Romans, countrymen,"
```

and the file pointer will advance past the entire first line to the beginning of the second:

```
Friends, Romans, countrymen,
|Lend me your ears;
I come to bury Caesar,
Not to praise him.
```

Eventually, after you have read all four lines of the file, the call to **`readLine`** will return the constant **`null`** to indicate that there is no more data in the file.

Checking for errors

Opening a file is an example of an operation that can sometimes fail. For example, if you request the name of input file from the user, and the user types the name incorrectly, the **`FileReader`** constructor will be unable to find the file you requested. To signal failure of this sort, the methods in the **`java.io`** package respond by **throwing an exception,** which is the name used in Java for signalling an exceptional condition outside the normal code flow. When an exception is thrown, the program stops running the code it was executing and proceeds upward through the various stack frames until it finds a point at which that exception is "caught." If the exception is never caught, the program simply stops running and the exception is reported in the Eclipse debugging window.

Many of the exceptions that occur in Java, such as dividing by 0 and the like, are called **runtime exceptions** and can occur at any point in the code. To make sure that clients check for situations like nonexistent input files, Java *requires* you to catch exceptions that occur in the **`java.io`** package using code that looks like this:

```
BufferedReader rd = null;
try {
    rd = new BufferedReader(new FileReader("antony.txt"));
} catch (IOException ex) {
    Code to report or recover from the error.
}
```

In class we will discuss the example method

```
BufferedReader openInputFile(String prompt)
```

This method displays the prompt and asks the user for an input file name. If that file exists, **`openInputFile`** returns a **`BufferedReader`** for that file. If not, **`openInputFile`** prints out a message indicating the error and gives the user another chance.

Using files for output

Files can also be used to store data written by a program. In many respects, this operation is easier than using files for input because you can use the **`println`** method, which works exactly like the method you've been using all this time. The only difference is that you need to supply a receiver, which is a **`PrintWriter`** directed to that file. Thus, you could change the **`run`** method from **`HelloProgram`** in Chapter 2 so that the output would go to the file **`hello.txt`** as follows:

```
public void run() {
    try {
        PrintWriter wr
            = new PrintWriter(new FileWriter("hello.txt"));
        wr.println("hello, world");
        wr.close();
    } catch (IOException ex) {
        println("An I/O exception has occurred");
    }
}
```