**CSC 101 – Intro to Computer Programming**

**LAB #13**

**File I/O**

**NAME:** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Objectives**

1. Create File Objects.
2. Create Input/Output Streams.
3. Read from a text file.
4. Create Scanner Object linked to a file.
5. Write to a text file.

**Getting Started**

Reading and writing data to files increases the usefulness of our programs tenfold. Using java programs to store data and/or read data from files allows us to come back to problems when we have new information. Imagine having to re-input all of our inventory every time that we run our program. Instead we can store our inventory to a file and open that file each time we need to update or check our inventory.

The first step to reading or writing to files is to understand how computers store information. Computers think in zeros and ones, the number 23 for them would be 00010111 in memory. Any text would be stored as zeros and ones that represent the ASCII code for those letters. Which means storing the number 23 in memory would be the equivalent to storing the character ‘2’ followed by the character ’3’. This would be 00110010 00110011 in ASCII code converted into binary. Other things need to be considered also, are we using international standard which would add 8 more bits per character, and how many bits are used to store an integer, I only used 8 but most likely at least 32 are used.

So we must decide if we want to store our data in a text format, or in straight binary. Since text format allows us to open and look at our data files with any text editor, I like to use these more often.

**File Objects**

First step to linking to files outside of our java program is creating what is known as a file object. The File class is created for you and is located in the java.io package.

import java.io.\*; //contains Files and I/O Streams

Creating a File object is as simple as creating any other object, all you need to know is the name of the file to connect to.

File infile = new File("test.txt");

This can be a String literal as shown above or a String object. What this does is create the necessary link to a file by that name. Whether the file exists or not at this point does not matter. Any use of this File object will be associated with a file by this name.

We can check if this file exists by testing the object as follows:

if (infile.exists() )   
 System.out.println("File exists already - we can read it");

If it does not exist – we can create it:

infile.createNewFile();

Although any time we tried to link to the file to write to it, it would create it automatically.

**IOException**

Many methods that read or write to files, including creating a file, throw an exception that needs to be handled. During debugging this can be a pain to add lots of try and catch blocks. A simple solution to allow us to continue working is to pass on the exception to the next higher method. In a sense we are tricking the computer into thinking that if an exception happens it will be handled at the next level. All we need to do is add this code at the end of the method signature that we are currently in (including the main). Here is an example:

public static void main(String args[]) throws IOException{

This tells the compiler that we will handle the exception at a higher level.

**Input/Output Streams**

A stream is an object created to read or write information to a file. Java has many built in streams that allow for different kinds of file access information. We are going to use a couple of basic ones and then you can try the others when you need them. *FileInputStream* and *FileOutputStream* are classes that contain methods for reading and writing to a linked File object.

As we create each object, we use the constructor and give it the name of a File object directly:

File text\_file = new File("test.txt");

FileOutputStream outstream = new FileOutputStream(text\_file);

OR

File another\_file = new File("test2.txt");

FileInputStream instream = new FileInputStream(another\_file);

The stream objects now contain the methods *read* and *write* which allow us to send to or read from a file one byte at a time. See text for details. Reading and writing by sending bytes of data is handy, but most of the time we are interested in moving integers, doubles, or text. Additional handy streams are created for our use.

Let’s look at outputting to a file first. The only method available to us with *FileOutputStream* is the *write* method, which expects one byte of data or an array of bytes. There exists another level of output streams which use this write method within more convenient methods for us. The classes *DataOutputStream* and *PrintWriter* are two of them. *DataOutputStream* has methods such as *writeInt( )* and *writeDouble( ),* which makes sending data of known types very convienent, however it converts them to binary and stores them in the file as machine language. *PrintWriter* has the methods *print( )* and *println( )*, which converts everything to text first and sends the information as ASCII characters. This is used exactly like *System.out* has been throughout the semester for output. See example below:

File text = new File("textfile.txt");  
FileOutputStream outstream = new FileOutputStream(text);  
PrintWriter outtext = new PrintWriter(outstream);  
   
// methods print() and println() in PrintWriter  
   
outtext.println(89123);//Items converted to text   
outtext.println(4.567); //and sent to text file  
outtext.println("Hello");

Reading from files is done in a similar fashion. The class *DataInputStream* is available to read from binary files. The methods *readInt( )*, *readDouble( )* and others are available to directly read these data types from files that have them stored in binary. Note: You have to know in what format the data is stored to read it correctly. If you used *PrintWriter* to write a bunch of integers to a file, you will get incorrect data if you try to read it back using methods from *DataInputStream*.

So to read information from a textfile, we need to use a different stream. Two very useful streams to read are *BufferedReader* and *Scanner*. *BufferedReader* has a method called *readLine( )* which will pull a single string from the file and give it to you in text format. If it is a list of integers, you will then need to convert it back into number format by using *Integer.parseInt( )*.

File text = new File("readfrom.txt");  
FileReader instream = new FileReader(text);//not FileInputStream   
BufferedReader in = new BufferedReader(instream);  
   
// method readLine() in BufferedReader  
   
String temp = in.readLine();//read first line   
int number= Integer.parseInt(temp); //string to integer

Using *Scanner* is even easier, and potentially more useful. We have used *Scanner* all semester, but we always inputted *System.in* when we constructed it. We simple tell it to use a file instead, and we can use all of the methods that we have been using. We don’t even have to use a *FileInputStream* with *Scanner*.

File text = new File("readfrom.txt");

Scanner fromfile = new Scanner(text); //Instead of System.in   
   
int num = fromfile.nextInt(); //read next int in file

**Closing Streams**

When we have finished using any input or output stream it is important to close the stream. The stream creates a link to an external file. If the program quits before all information has been transferred to or from the file, we can get bad data or a corrupted file.

fromfile.close(); //very simple and important

We don’t always control when disk operations happen on a computer, and this forces a completion of activities before moving on in the program.

**Lab Tasks – complete these and show instructor before leaving**

Use the information provided here and in chapter 8 of our text to do the following exercises before you leave.

1. Write a program that will ask for the user to input a filename of a text file that contains an unknown number of integers. And also an output filename to display results. You will read all of the integers from the input file, and store them in an array. (You may need to read all the values in the file once just to get the total count) Using this array you will find the max number, min number, average value, and standard deviation. These results will be reported to both the screen and placed into the output file that the user choose.

Output to screen ***and*** file could look like this:

Read from file: 12 values

Maximum value = 20

Minimum value = 3

Average value = 13.34

Standard Deviation = 2.15

You can create a simple NotePad file to store a bunch of integers, just make sure that there is at least a space or a line between each number.

1. Write a program that does basic encrypting of text. You will ask the user the filename of a text file which contains a few sentences of text. You will read this text file one character at a time, and change each letter to another one and write out to an output file. The conversion should be done a -> b, b->c , … z->a, A->B, B->C, …. Z->A. This means just shift each letter by one, but Z goes back to A.

Example: Hello converts to Ifmmp

All other punctuation and spaces or symbols can stay as they are.

If you have time you could write the reverse program that would take your encrypted file as the input and get back the original message

Make sure to show these working programs to your instructor before you leave lab to get full credit. If you can’t not finish all problems assigned during lab time and you were present and working the full time, you will have a chance to show me what you completed at the beginning of the next lab. You cannot just leave lab early and get full credit later.