Programming Exercise 13:

MultiDimensional Arrays

Purpose: Using multidimensional arrays
Background readings from textbook:
Liang, Chapter 8.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{pmatrix}$$

where $c_{ij} = a_{i1} \times b_{1j} + a_{i2} \times b_{2j} + a_{i3} \times b_{3j}$.

Due date for section 001: Monday, April 18 by 10 am
Due date for section 002: Wednesday, April 20 by 10 am

Overview

The arrays we have used are 1-dimensional arrays meaning that we use a single index to access the array element. Conceptually, we think of a 1-dimensional array as a *list* (either left-to-right or top-down) although formally, we will call 1-D arrays *vectors*. The 1-D array is adequate for many applications but not all. Images have both rows and columns and a chess board also has rows and columns. Data that contains 2 dimensions should be represented using a 2-D array, or what we might call a *matrix* (or table). To declare a 2-D array, you use two sets of brackets and to access a 2-D array element, you use two indices, the row number and the column number. In this assignment, you will learn two important programming skills, working with 2-D arrays and also performing input and output to disk file (rather than keyboard). In the next two sections, these ideas are discussed. Then, we turn to your assignment.

Part 1: Declaring and Using 2-D Arrays

The 2-D array is declared like the 1-D array except that you have a second []. To access a 2-D array, you use two indices instead of one, for instance like array[i][j] where i is the row number and j is the column number (remember that array indices start at 0). Notice that the two indexes will be of different variables. Using array[i][i] means that you are always going to access a value on the diagonal of the matrix. The following is an example of creating an 8x8 array of Strings which you might use for a chess board.

```
String[][] board=new String[8][8];
```

We usually access all elements of an array using a for loop to control access. With 2-D arrays, we will use two nested for loops. Here we see a comparison where n is the number of 1-D array elements and the number of rows in the 2-D array, and m is the number of columns in the 2-D array.

```
for(int i=0;i<n;i++)
     if(array[i]==x) …;

for(int i=0;i<n;i++)
     for(int j=0;j<m;j++)
          array[i][j]++;
```

Part 2: Inputting and Outputting Disk Files

Thus far, we have used a Scanner for input and System.out for output. We can also obtain input from a disk file and output to a disk file. Although the approach is similar, it is necessarily more complex. We focus on input first. To input, we will continue to use a Scanner but when we instantiate the Scanner, we do not use the parameter System.in. Instead, we need to specify a parameter of type File. The File class is part of the java.io package so we would import java.io.File (or java.io.*). To instantiate a File, we supply it the name of the file as a String. For instance, `File f1=new File("myfile.txt");` Note that the file must be located in the same directory as your project in Eclipse. For instance, if your Java path is set up to place your files in Users > foxr > Java and you are working on project Lab13, then you would place myfile.txt in Users > foxr > Java > Lab13. This directory contains subdirectories called .settings, bin and src. Do not put your file in the src directory itself (which is where your java file(s) would be stored). Alternatively, you can specify the file with a full path such as `"C:\\Users\\foxr\\Java\\Lab13\\src\\myfile.txt"` Notice the use of \\ because \ is used to denote an escape character in Java, so to actually specify "\" you use "\\" (and in Windows, "\" is also used indicate a subdirectory). What follows is the code needed to input from the file "filename.txt".

```
File file = new File("filename.txt");
Scanner in=new Scanner(file);
```

We can combine the two instructions into one as in:

```
Scanner in=new Scanner(new File("filename.txt"));
```

The Scanner is used much like we have used it for keyboard input by specifying `in.next()`, `in.nextInt()`, etc. There is an additional method though that will be useful, `hasNext()`. This method returns true if there is still content in the file to access and false if you have reached the end of the file (known as EOF). We can use this to control a while loop to input everything from the file, such as:

```
while(in.hasNext()) {
     array[i]=in.next();
     i++;
}
```

Make sure you close your Scanner when you are done. It's critical to close disk files (more so than closing a keyboard Scanner). Now, what if you want to input from both keyboard and disk file? Use two different Scanners. Below, we create two Scanners. We use in1 for keyboard input and in2 for disk file input. Notice how we are using in1 to get the file name for in2!

```
String filename;
Scanner in1=new Scanner(System.in);
System.out.print("Enter the filename you want to use:  ");
filename=in1.next();
Scanner in2=new Scanner(new File(filename));
```

When done, close both in1 and in2.

To send output to a disk file, we need to use something other than System.out, and we do not use a Scanner. Instead, we use a class called `PrintWriter`, also part of java.io. To set up a PrintWriter, use the following instruction. Notice the use of File in creating the PrintWriter.

```
PrintWriter pw=new PrintWriter(new File("filename.txt"));
```

To perform output, pass your PrintWriter messages print or println as in:

```
pw.print(…);
pw.println(…);
```

Make sure that the file you are writing to is not currently open by a Scanner as you cannot have a file open for reading and writing. If needed, you could read from a file with a Scanner, close the file, open it with a PrintWriter, output to it then close the PrintWriter.

There is a significant catch in doing input or output to a file. To use any of the classes in java.io (such as File), you have to handle a particular type of Exception called an `IOException`. Exceptions are covered in CSC 360, so we won't look at them here other than to say that an Exception is a run-time error that, if not handled, causes your program to terminate. In Java though, you can write your own code to handle exceptions to keep the program from terminating. To handle an IOException, we will use the simplest approach which is to *throw* any raised IOException somewhere else. We do this by adding `throws IOException` to the method header of any method that uses an io class (as well as any method that calls that method). In your program, you will add this to main and all of your methods since they will all deal with io classes. For instance, you would have:

```
public static void main(String[] args) throws IOException
public static void getInput(…) throws IOException
```

Part 3: Your Assignment

In this assignment, you will create a single class with a main method and some auxiliary methods to input a 2-D array from a disk file, input some "transactions" to change the 2-D array and output the changed 2-D array to another file. Your main method will be minimal (see below). Most of the work will go on in your methods. In main, declare (but do not instantiate) your 2-D array. Then call the three methods. The first method will input into the 2-D array and return it. The other two methods are void methods. Here is the primary set of code for main.

```
data=input("input1.txt");
process("input2.txt",data);
output("output.txt", data);
```

The first method receives the name of the file to open (which contains the original 2-D array data) and the 2-D array. In this method, open the file. The first two entries of the file will be int values which specify the rows and columns for your 2-D array. Now, instantiate your array. For instance, if your parameter is called table, you would do `table=new String[rows][columns];` where rows and columns are int variables whose values were input from the file. Now, using two nested for loops, input each next String from the file and insert it into the appropriate location. If your loop indexes are i and j, this instruction might look like: `table[i][j]=in.next();` Assume that the two int values (rows, columns) are accurate. Close the Scanner and return your 2-D array. In main, you will assign your 2-D array to what is returned (see above).

The second method will receive a second filename and the 2-D array. In this method, open this file for input via a Scanner. This file will consist of rows that contain four items, two ints and two Strings. For instance, a row might be

6 3 First Second

Use a while loop to iterate while the file still has items using `hasNext()`. After each input, increment a variable that is counting the number of rows input. Now, test to see if the value in the 2-D array at the location specified by the two ints equals the first of the two Strings. If so, replace it with the second String. For instance, if data[6][3] is "First" then change it to "Second". If the item in the array does not match the first String, add 1 to a variable that is counting the number of errors. To input the four data items on this row, use four assignment statements. The first two will use `nextInt()` and the last two will use `next()`. After the loop at the end of this method, output (using System.out) a brief report that lists the file accessed and the number of transactions and how many resulted in errors.

The third method will receive a filename and the 2-D array. This method will output the elements of the 2-D array to this file. Use a different filename than either of your input files (or else you will erase those files). You will need a PrintWriter in this method (instead of a Scanner). You will use 2 nested for loops like you did in the input method but here, you will use print to output each String. If your PrintWriter is called out, the instruction might look like this:

```
out.print(table[i][j] + "\t");
```

The \t inserts a tab between this entry and the next. We want to end this row with a \n, so we also need to do `out.println();` Your outer for loop will consist of two instructions, the inner for loop (which has the print statement) and this println statement. Make sure the println only happens after the inner loop completes. Close the output file when done.

Notice that you were given the dimensions of the array by reading the first two int values from the first input file and those variables were part of the input method. How do you know the proper values for rows and columns for your output method? We will determine this by looking at the array. If the array is called table, then we can obtain these values using these two instructions:

```
rows = table.length;
columns = table[0].length;
```

The `.length` operator is not a method but an operation that returns the number of items found in the array. For a 2-D array, `.length` returns the number of rows. To determine the number of columns, use `table[0].length`.

Once you have written your program, run it on the 2 text files called test1.txt and test2.txt (they are posted next to the link of this assignment). This will generate an output file and an output of the number of items processed and the number that had errors. An example of the output file and output are also placed on the website for you to compare against your results. If you did not get properly formatted output or the right answers, debug your program. Once it is correct, run your program again on the input files input1.txt and input2.txt. Obtain the output file and copy the output you're your System.out statement(s) at the bottom of your source code in comments. Submit your source code and your output file, either by hardcopy or email.