Homework #14

File input & output; line-based file processing; more string processing; simple array processing.

1. Aim

The purpose of this lab is to:

a. Introduce you to line-based file processing.

b. Learn how to creating output files.

c. Get additional practice with advanced string processing.

d. Introduce you to very simple processing with arrays.

It is recommended that you read and understand all the instructions below before starting this exercise.

2. Files Needed

You will not be provided with any starter files for this homework. Create a project in Eclipse for this assignment. For each exercise below, create a Java class with the appropriate name. When you create each program, be sure to have Eclipse create the public static void main method for you. Be sure to include the standard header comments at the top of each file.

3. To be Handed In

The files **MadLib2.java** and **TriangleCount.java** should be sent to me when you have them completed. Be sure to name the files/classes as specified.

4. Exercises

Part I: MadLib2.java

Complete the following exercise by writing appropriate Java code in the file **MadLib2.java.**

"Mad Libs" are short stories that have blanks called placeholders to be filled in. In the non-computerized version of this game, one person asks a second person to fill in each of the placeholders without the second person knowing the overall story. Once all placeholders are filled in, the second person is shown the resulting story, often with humorous results.

The program begins by prompting the user for input and output file names. Then the program reads the input file, prompting the user to fill in any placeholders that are found without showing the user the rest of the story. As the user fills in each placeholder, the program is writing the resulting text to the output file. After the input file is processed, the program asks the user if they want to see the resulting output file, in which case the output file is opened for input and its contents are echoed to the screen. The user is then given the option of processing another MadLib file.

Notice that if an input file is not found, the user is re-prompted. No re-prompting occurs for the output file. If the output file does not already exist, it is created. If it already exists, overwrite its contents. (These are the default behaviors in Java.) You may assume that the output file is not the same file as the input file.

Example:
This is a sample execution to show you how **MadLib2.java** should behave:

```
Welcome to the game of Mad Libs.
I will ask you to provide several words
```

```
and phrases to fill in a mad lib story.
The result will be written to an output file.

Input file name: oops.txt
File not found. Try again: tarzan.txt
Output file name: out1.txt

Please type an adjective: silly
Please type a plural noun: apples
Please type a noun: frisbee
Please type an adjective: hungry
Please type a place: Nashville, TN
Please type a plural noun: bees
Please type a noun: umbrella
Please type a funny noise: burp
Please type an adjective: shiny
Please type a noun: jelly donut
Please type an adjective: beautiful
Please type a plural noun: spoons
Please type a person's name: Keanu Reeves

Your MadLib story has been created.
Do you want to see the resulting story? (Y|N) y

Here is the resulting MadLib:

One of the most silly characters in fiction is named
"Tarzan of the apples ." Tarzan was raised by a/an
frisbee and lives in the hungry jungle in the
heart of darkest Nashville, TN . He spends most of his time
eating bees and swinging from tree to umbrella .
Whenever he gets angry, he beats on his chest and says,
" burp !" This is his war cry. Tarzan always dresses in
shiny shorts made from the skin of a/an jelly donut
and his best friend is a/an beautiful chimpanzee named
Cheetah. He is supposed to be able to speak to elephants and
spoons . In the movies, Tarzan is played by Keanu Reeves .

Do you want to process another Mad Lib? (Y|N) n
```

Input Data Files:
Download the example input files from the assignment page in Oak and save them to the project folder created by Eclipse for this assignment. Mad lib input files are mostly just plain text, but they may also contain placeholders. Placeholders are represented as input tokens that begin with `"<"` and end with `">"`. For example, the file `tarzan.txt` used in the sample execution above contains:

```
One of the most <adjective> characters in fiction is named
"Tarzan of the <plural-noun> ." Tarzan was raised by a/an
<noun> and lives in the <adjective> jungle in the
heart of darkest <place> . He spends most of his time
eating <plural-noun> and swinging from tree to <noun> .
Whenever he gets angry, he beats on his chest and says,
" <funny-noise> !" This is his war cry. Tarzan always dresses in
<adjective> shorts made from the skin of a/an <noun>
and his best friend is a/an <adjective> chimpanzee named
Cheetah. He is supposed to be able to speak to elephants and
<plural-noun> . In the movies, Tarzan is played by <person's-name> .
```

Your program should break the input into lines and break the lines into tokens using a Scanner so that you can look for all its placeholders. Normal non-placeholder tokens can be written directly to the output file as-is, but placeholder tokens cause the user to be prompted. The user's response to the prompt is written to the output file, rather than the placeholder itself. You should accept whatever response the user gives, even a multi-word answer or a blank answer.

You may assume that each word/token from the input file is separated from neighboring words by a single space. In other words, when you are writing the output, you may place a single space after each token to separate them. You do not need to worry about blank spaces at the end of lines. It's okay to place a space after each line's last token in the file.

Sometimes a placeholder has multiple words in it, separated by a hyphen (`-`), such as `<proper-noun>`. As your program discovers a placeholder, it should convert any hyphens into spaces. Any hyphens that appear outside of a placeholder, such as in the other text of the story, should be retained and not converted into spaces.

When prompting the user to fill in a placeholder, convert the placeholder to lowercase. If the placeholder begins with a vowel (a, e, i, o, or u), prompt for a response using **`"an"`**. If not, use **`"a"`**. For example:

Placeholder	**Resulting Prompt**
`<noun>`	`Please type a noun:`
`<aDjEcTiVe>`	`Please type an adjective:`
`<plURAl-noun>`	`Please type a plural noun:`
`<Emotional-Actor's-NAME>`	`Please type an emotional actor's name:`

Note: Many students lose points because they don't properly find all placeholders or properly handle the "a/an" prompt. Make sure to test your program using all of the sample input files on the assignment page as well as your own additional testing.

Do not make unnecessary assumptions about the input. For example, an input file could have < and > characters in it that are not part of placeholders, and these should be retained and included in the output. You may assume that a placeholder token will contain at least one character between its < and > (in other words, no file will contain the token <>). You may assume that a placeholder will appear entirely on a single line; no placeholder will ever span across multiple lines. Your output mad lib story must retain the original placement of the line breaks from the input story.

"To echo the resulting story" simply means to open for input the text file that was just created and echo its contents to the console. You do not need to do any processing on the file or any kind of testing on its contents; just output the file's entire contents to the console, line by line. Hint: If you want help on this part of the assignment, see the answer to Chapter 6 Self-Check problem 12, `printEntireFile`.

Implementation and Development Strategy:

Read/write files using `Scanners` and `PrintStreams`, passing them as parameters. Remember to `import java.io.*;`
Read all console user input using the `Scanner's nextLine` method (not the `next` method, which permits only one-word answers). When reading files, you may need a mixture of line-based and token-based processing as shown in Chapter 6.

To re-prompt for input file names, you need to know whether a file with a given name exists. You can do this by using methods from `File` objects as shown in class. The textbook also shows an alternative technique for solving the file-not-found problem called a `try/catch` statement, but we do not recommend using this on your assignment.

You will need to use several `String` methods to search for and replace characters. See textbook sections 3.3 and 4.3-4.4. In particular you may use the `replace` method to replace occurrences of one character with another. For example:

```
String str = "mississippi";
str = str.replace("s", "*"); // str = "mi**i**ippi"
```

Incorrect file input often leads to exceptions. If you get an `InputMismatchException`, you are trying to read the wrong type of value from a `Scanner`. If you get a `NoSuchElementException`, you are trying to read past the end of a file or line. If you get an exception, look at the exception's text to find the relevant line number in your file. For example:

```
Exception in thread "main" java.util.NoSuchElementException: No line found
        at java.util.Scanner.nextLine(Scanner.java:1516)
        at MadLibs.myMethodName(MadLibs2.java:73)
        at MadLibs.main(MadLibs2.java:20)
```

After finding the line number, use `println` statements or the debugger to see the values of each variable as it is read. A common bug is calling `next`, `nextLine`, etc. on the wrong `Scanner` object, so examine those calls carefully.

As you are developing the program, you may want to initially "hard-code" the input and output filenames; in other words, you may want to just use fixed file names in your code rather than prompting the user to enter the file names. You may also want to temporarily print extra "debug" text to the console while developing your program, such as printing each token or placeholder's text as you read it from the input file.

It is easier to debug this program when using a smaller input file with fewer placeholders. On the assignment page we have posted an input file `simple.txt` with a much shorter mad lib story. You may want to use this as your test file at first.

Style Guidelines:

Structure your solution using static methods that accept parameters and return values as appropriate. It is okay to have methods that return objects, such as a `File` or `Scanner` or `PrintStream`. Objects can also be passed as parameters.

For full credit, you must have at least **4 non-trivial methods** other than `main` in your program. It is okay for some `println` statements and code to be in `main`, as long as you use good structure and `main` is a concise summary. Each method should have a single coherent purpose, and no one method should do too large a share of the overall task. The `main` method should not directly perform a large share of the work itself, such as actually reading the mad lib input file and producing the output. It is okay for `main` to have some `println` statements and other code, but the majority of the work of your program should come from `main` making calls to other methods that each handle a clear and coherent task.

Avoid "chaining" many calls together without ever returning to `main`. We will be especially picky about redundancy and structure on this assignment. If you have repeated code, put it into a method or loop so that it needs to be written only once. If a method is too long or incoherent, split it into smaller pieces.

For this program you are limited to the language features in Chapters 1 through 6 of the textbook. In particular, you are **not allowed to use arrays on this program**. Use whitespace and indentation properly. Limit lines to 100 characters. Give meaningful names to methods/variables, and follow Java's naming standards. Localize variables. Put descriptive comments at the start of your program, at the start of each method (use JavaDoc comments), and inside methods on complex sections of code.

Acknowledgement:

This assignment is based on an assignment given at the University of Washington, where the idea for the assignment is credited to Stuart Reges.

Part II: TriangleCount.java

Complete the following exercise by writing appropriate Java code in the file **TriangleCount.java.**

This program explores patterns of digits for what are called the ***triangle numbers***. Triangle numbers are described on this Wikipedia page: http://en.wikipedia.org/wiki/Triangular_number . The triangle numbers are the number of dots in an equilateral triangle uniformly filled with dots. From the Wikipedia page, we see that the nth triangle number can be directly computed by the formula:

$$\frac{n * (n + 1)}{2}$$

You are to write a program that generates the first 10,000 triangle numbers and prints them out to the file “`TriNums.txt`”, one number per line. Additionally, while generating the numbers, you are to count how many times each digit 0-9 appears as the last digit (the ones digit) of those numbers. To do this, simply create an array of 10 integers that will be used as counters keeping track of how many times the corresponding digit appeared as the last digit (this is similar to the `mostFrequentDigit` example seen in lecture #25). After printing all 10,000 triangle numbers to the file and examining their ones digit, print out the resulting counts at the end of the file.

Note: Do not worry about breaking this program into multiple methods. We have not yet learned about passing arrays as parameters or returning arrays from methods, and this program is short enough that you can simply put it all in the main method.

Example:
This is a sample of the `TriNums.txt` file that your **TriangleCount.java** program should create:

```
1
3
6
10
...   ← your program should generate all 10000 numbers
49995000
50005000

count for 0 = xx   ← your program should print the actual counts
count for 1 = xx
count for 2 = xx
count for 3 = xx
count for 4 = xx
count for 5 = xx
count for 6 = xx
count for 7 = xx
count for 8 = xx
count for 9 = xx
```

5. Additional requirements

A. You must start your program with header comments that provide your name, VUnetID, email address, the date the program was last modified, an honor statement ("I have neither given nor received unauthorized help on this assignment"), and a short description of the program (see the examples distributed with homework #2).
B. Each method that you write (except main), should be preceded by a block of Javadoc comments which describe what the method does, what the parameters are, and what it returns. Any required preconditions should also be clearly stated.
C. You should use a consistent programming style. This should include the following.
   a. Meaningful variable & method names
   b. Consistent indenting
   c. Use of "white-space" and blank lines to make the code more readable
   d. Use of comments to explain pieces of tricky code
   e. A descriptive JavaDoc comment before each method (other than main, which already has a short description of the program). Note that we will be using JavaDoc comments from here on out to document our methods.
   f. Lines that do not extend beyond column 100 (or column 80 is even better)

See the code examples in the class text for a good formatting style.

6. Submission for grading

Once you have completed the exercise, the files **MadLib2.java** and **TriangleCount.java** should be sent to me

7. Grading

This lab is worth 40 points, 30 points for **MadLib2.java** and 10 points **TriangleCount.java**. Your grade will be based on whether your solution is correct or not, and on how closely you followed the directions above. Remember that programming style will now be a part of your grade (on this and all subsequent assignments).