CSCI 213 – Lab 3

Prof. Givens
See Canvas for due date.

Log into a Linux machine in lab.

Download the files for Lab 3 into your `lab3` folder. The lecture slides will also be useful and can be found on Canvas.

Day 1: Reading from Files

1. Counting File Data

Open a text editor (such as JEdit), create a new file called `FileCounting.java`, and save it in your lab folder. Add the class, and then add the following static methods that each require you to read from a file.

Each method has suggested tests. Run the tests in a main method as you code or edit `FileCountingTester.java` to test. When all your functions test well, finish the main method as instructed.

If you want to make sure your methods are setup correctly (name, parameters, and return types), you can use `FileCountingTester.java`. `FileCountingTester.java` does not currently test for correctness, but it will compile and run if your methods are setup correctly.

Don't forget your imports and the process to read from a file:

1. create a new File using the name of the file
2. create a new Scanner using the File object
3. read the data (*)
4. close the Scanner

(*) For some methods you may find it better to read line by line, and for some methods you make find it better to read word by word.

- `numLines(String)`: Receives a String that is a file name. Use the parameter to create a File object. Reads from the file and determines and returns the number of lines, as an int, used in the file. Include blank or whitespace-only lines in the count). If the method catches an exception, print the stack and return -1 to indicate something went wrong.

  `quote1.txt` has 3 lines. `quote2.txt` has 43 lines.

Note: Java does not allow you to write a return statement in *only* a try block. One way to avoid this is to initialize the variable you will return before the try-catch, use the variable in the try-block, and then return the variable after the try-catch is complete.

- `wordCount(String)`: Receives a String that is a file name. Use the parameter to create the File object. Reads from the file and determines and returns the number of words, as an int, used in the file. Do not include blank or whitespace-only lines in the count). If the method catches an exception, print the stack and return -1 to indicate something went wrong.

  `quote1.txt` has a word count of 21. `quote2.txt` has a word count of 153.

- `uniqueWords(String)`: Receives a String that is a file name. Use the parameter to create a File object. Reads from the file and creates and returns an ArrayList (of Strings) of all the unique words in file. The words should be without punctuation and all lowercase and should not include the empty string. If the method catches an exception, print the stack and return `null` (an empty Object) to indicate something went wrong.

  `quote1.txt` has 15 unique words. `quote2.txt` has 26 unique words. If your numbers are off, you can print the ArrayList to get an idea of which words may be incorrectly counted more than once.

- `main(String[])`: The main method should take two command line arguments: the first is the code to determine which method to run, and the second is the name of the file. The codes are `-a` for counting the number of lines, `-b` for getting the word count, and `-c` for counting the number of unique words. Check that there are enough arguments to process the request. Determine which method to run and call the method correctly. Output the result to the user.

  Note that `uniqueWords` returns an ArrayList, but you should output the number.

  Below are some sample runs of the program.

```
>java FileCounting -a quote1.txt
There are 3 lines in the file quote1.txt

>java FileCounting -a quote2.txt
There are 43 lines in the file quote2.txt

>java FileCounting -b quote1.txt
There are 21 words in the file quote1.txt

>java FileCounting -b quote2.txt
```

```
There are 153 words in the file quote2.txt

>java FileCounting -c quote1.txt
There are 15 unique words in the file quote1.txt

>java FileCounting -c quote2.txt
There are 26 unique words in the file quote1.txt

>java FileCounting
Too few arguments.
Please rerun with arguments: [code] [file]

>java FileCounting -a
Too few arguments.
Please rerun with arguments: [code] [file]

>java FileCounting -a x
java.io.FileNotFoundException: x (The system cannot find the file specified)
        at java.io.FileInputStream.open0(Native Method)
        at java.io.FileInputStream.open(Unknown Source)
        at java.io.FileInputStream.<init>(Unknown Source)
        at java.util.Scanner.<init>(Unknown Source)
        at FileCounting.numLines(FileCounting.java:11)
        at FileCounting.main(FileCounting.java:83)

>java FileCounting -d quote1.txt
Code should be -a, -b, or -c.
```

Make sure your code compiles, runs, and is correct before moving on to the next section.

Day 2: Writing to Files and CSVs

2. Grade Data

Open a text editor (such as JEdit), create a new file called `GradeProcessing.java`, and save it in your lab folder. Add the class, and then add the following static methods that require the use of CSV files.

Since this program only has one non-main method (except any helper methods you decide to use), you should code main as described below to test it. Don't forget your imports.

Don't forget your imports and the process to read from a CSV file:

1. create a new File using the name of the file
2. create a new Scanner using the File object

3. read the data line by line
4. split the data on the commas to process
5. close the Scanner

and to write to a CSV file:

1. create a new File using the name of the file
2. create a new PrintWriter using the File object
3. write data separated with commas
4. close the PrintWriter

- `gradeAverage(String, String)`: Receives two Strings, the first an input CSV file and the second an output CSV file. This function does not return anything.

  The function reads and processes each line from the input file. On each line the input file has a name followed by a sequence of grades. The method should calculate the average of the grades, determine the letter grade (A, B, C, D, or F) and output the name followed by the average (to one decimal place) and the letter grade to the output file. Use commas to separate the data and no extra spaces.

  Handling exceptions: This method should *not* use a try-catch, instead it should state that it could throw an IOException when the method is defined. Also, when processing a line, if the line has less than two pieces of data, it should throw a IllegalArgumentException with an informative message.

  Example input file:

  ```
  Sally,75,77,83,65,,
  Jane,92,85,90,77,83,
  Beth,97,86,92,85,78,81
  Emma,91,92,93,94,97,
  Liz,60,70,80,64,,
  ```

  Example output file:

  ```
  Sally,75.0,C
  Jane,85.4,B
  Beth,86.5,B
  Emma,93.4,A
  Liz,68.5,D
  ```

- `main(String[] args)`: Run `gradeAverage` on the files `class1.csv`, `class2.csv`, and `class3.csv`. Select output file names to match such as `result1.csv`. You should catch your potential I/O exceptions as well as the RuntimeException. In both cases print the stack trace and use a separate print statement to describe what likely happened. Include a finally with a print statement because you can. Note that class3.csv should cause your RuntimeException to be triggered, however, once your code is correct, no I/O exceptions should be triggered.

Expected Output

file output from `class1.csv`

```
Sally,75.0,C
```

```
Jane,85.4,B
Beth,86.5,B
Emma,93.4,A
Liz,68.5,D
```

file output from `class2.csv`

```
Ron Weasley,70.5,C
Harry Potter,85.4,B
Hermione Granger,95.5,A
Luna Lovegood,94.1,A
Ginevra Weasley,95.1,A
Lavender Brown,68.5,D
Neville Longbottom,72.8,C
Dean Thomas,91.6,A
Padma Patil,76.5,C
Pavarti Patil,75.4,C
Oliver Wood,85.1,B
Cho Chang,88.9,B
Angela Johnson,69.1,D
Draco Malfoy,83.9,B
```

If the output file is created for `class3.csv` (before the discovery of the bad line), the output file should be empty.

Submission

Submit the following files to Canvas. Do not zip the files.

- `FileCounting.java`
- `GradeProcessing.java`