AP Computer Science A

Project 3
Text Statistics

Project Overview

In this program, you are going to implement a program that analyzes text files to generate some useful statistics. Your program will provide basic statistics like the number of characters, words, and lines, the average word length, and the number of each letter and word length. However, the same technique could be used to collect more statistics and create a program to ascertain characteristics of books and classify them for various purposes (this concept was the basis of a startup company initiated by BSU students which later was purchased by Apple for $15 million). Plain text versions of several books and articles (e.g. Alice in Wonderland, The Gettysburg Address, etc.) will be made available for you to analyze using your program.

Objectives

- Use arrays.
- Use command-line arguments.
- Read from text files using the Scanner class.
- Parse and analyze text input.
- Implement an interface.

Specifications

For this project you are going to create two classes.

- *TextStatistics* will be the class that reads a text file, parses it, and stores the information about the words and characters in the file.
- *ProcessText* is the driver class that gets a list of one or more filenames from the command line and collects statistics on each of the files using an instance of the TextStatistics object.

For this project you will implement two interfaces

- *TextStatisticsInterface.java,*
- *TextStatisticsTest.java*

You should be able to develop this program incrementally in such a way that you can turn in a program that runs even if you don't succeed in implementing all the specified functionality. Specifications for the *TextStatistics* and *ProcessText* classes are discussed below,

ProcessText

The *ProcessText* class will serve as the driver class with a main method which processes one or more files to determine some interesting statistics about them. The *ProcessText* driver class should meet the following criteria,

1. **Command-line validation**. The names of the files to process will be given as command line arguments. Your driver class must,
    - Validate the number of command line arguments. There should be at least one file name given.
    - If no files are given on the command line, your program must print a usage message and exit the program immediately. The message should read as follows. Usage: java ProcessText file1 [file2 ...] This lets the user know how they should run the program without having to go look up the documentation.
2. **Process command-line arguments**. If valid filenames are given on the command line, your program will,
    - Will process each command line argument by creating a File object from it and

checking to see that the file actually exists. Recall, the args parameter of the main method is an array of String objects that contains the command line arguments to the program. For your program, the array should contain the names of the files to be processed.

- If a file does exist, your program will create a *TextStatistics* object for that file and will print out the statistics for the file to the console.
- If a file does not exist, a meaningful error message needs to be printed to the user and your program will continue processing the next file. (An invalid file in the list should not result in the program crashing or exiting before all files have been processed.)

TextStatistics

TextStatistics is an instantiable class that reads a given text file, parses it, and stores the generated statistics. The *TextStatistics* class should do the following,

- **Implement the given TextStatisticsInterface** (don't modify the interface, it just provides a list of methods that your class must include). To implement an interface, you must modify your class header as follows,

```
public class TextStatistics implements TextStatisticsInterface{
   //Your class goes here
}
```

- **Include instance variables**. Include a reference to the processed File. Include variables for all of the statistics that are computed for the file. Look at the list of accessor methods in the TextStatisticsInterface to determine which statistics will be stored (accessor methods typically are named with the word "get" following by whatever information they are accessing)
- **Constructor.** The constructor takes a File object as a parameter. The constructor should open the file and read the entire file line-by-line, processing each line as it reads it.

  Note: Your implementation must read through each file once. By the end of the constructor, the `TextStatistics` object should have collected all of its statistics and calls to its accessor methods will simply return the stored values.

  – Your constructor needs to handle the FileNotFoundException that can occur when the File is opened in a Scanner. Use a try-catch statement to do this. Don't just throw the exception.
  – As each line is read, collect the following statistics:
    1. The number of characters and lines in the file. The number of characters should include all whitespace characters, punctuation, etc. The number of lines should include any blank lines in the file.
    2. The number of words in the file. You must use the StringTokenizer from the java.util package to count the number of words in each line of the text file.

  Note: To ensure everyone's results are consistent, you must use the exact delimiter given below rather than making up your own.

```
StringTokenizer tokenizer =
```

```
new StringTokenizer(inputLine, " ,.;:'\"&!?-_\n\t12345678910{}()@#$%^*/+-");
```

The second parameter to the StringTokenizer constructor is a string of delimiters that the tokenizer will use for separating words in the file. Note that the first character in the second argument to StringTokenizer is a blank space. The tokenizer will not return any of the delimiter characters. For example, using tokenizer.nextToken() on the string:

scheme, and the "plan" (for us)

will give the following tokens,

scheme
and
the
lan
for
us

Note: Using the above tokenizer might lead to strange words occasionally. For example, the contraction "we've" results in two words "we" and "ve." This is okay and your program does not have to do anything to fix this.

3. The number of words of each length that appears in the file. Assume that the maximum word length is 23. You do not need to print lengths that have a count of zero.
4. The average word length for the file.
5. The number of each letter that appears in the file - do not separate upper and lower case, just convert all characters to lower case before counting.

- **Getter (accessor) methods**. Implement the accessor methods for the number of characters, number of words, number of lines, average word length and for the arrays that contain the number of words of each length and the number of times each letter occurs in the file.
- **toString() method**. Write a toString() method that generates and returns a String that can be printed to summarize the statistics for the file as shown in the sample output shown below.

Extra Credit (10 points)

Find and report all the line numbers at which the longest word in a file appears. If there are two or more different words of the same longest length, then track the first one only. An ArrayList works well for tracking the line numbers since we don't know how many occurrences of the longest word will we find ahead of time.

Getting Started

- Create a new NetBeans project for this assignment.
- The easiest way to import all the files into your project is to download and unzip the starter files directly into your project workspace directory. The starter files are available here,

https://drive.google.com/drive/folders/0B6lKrC3UayiJWEdkS2tSR2ZKTWM?usp=sharing

- After you unzip the files into your workspace, go back to NetBeans and refresh your project.
- Create ProcessText and TextStatistics classes in your project.
- Start by implementing ProcessText. You can ignore the command-line arguments to start. Just hard-code a file name so you can test your TextStatistics class as you write it. Create a File object and check to see that the file actually exists.
  - If the file does exist, your program will create a TextStatistics object for that file and print out the statistics for the file to the console.
  - If the file does not exist, a meaningful error message needs to be printed to the user.
- Next you can start implementing TextStatistics according to the specifications described above
- At this point, you should go back and add command-line argument processing to ProcessText as described in the specifications below. To make sure it correctly handles command line arguments, run it from the command line with no arguments, files that don't exist, and files that do exist.
- Make sure to test your program thoroughly. I am giving you the test program and scripts that I will use to grade your program. Take advantage of this and make sure they all pass!

Testing

You must test your program thoroughly before submitting it. We will be using similar testing strategies when we grade your program, so you should have a good idea whether or not your code will pass our tests before submitting.

TextStatisticsTest.java: Automated testing based on the interface.

I have provided a test program that tests your TextStatistics class using three sample text files. This test program will not compile unless you have properly implemented the required interface. This is available in your starter files.

autograde.sh: Testing based on program output.

autograde.sh. A shell script (a program made up of shell commands) that you can run to see if your program is going to work with the shell script used for grading the programs.
testfile.txt and etex. Sample text files used by autograde.sh
testresults. The expected output of autograde.sh. Your output should match the contents of this file.

To use these files to test your program, copy them into your program directory. Comment out the package name in all related files (*TextStatisticsTest.java*, T *TextStatisticsInterface.java, ProcessText, TextStatistics).* Now you run the test by typing the following at the command line prompt,

./autograde.sh

If your program does not compile and run, you need to fix it if you want any points for the program. Make sure all the files have the names specified.

Sample Session

Sample output for bad arguments/non-existing files,

```
[you@onyx p4]$ java ProcessText
Usage: java ProcessText file1 [file2 ...]

[marissa@onyx p4]$ java ProcessText not-a-file.txt
Invalid file path: not-a-file.txt

[marissa@onyx p4]$ java ProcessText not-a-file.txt testfile.txt
Invalid file path: not-a-file.txt
Statistics for testfile.txt
 ============================================================
 11 lines
 79 words
 465 characters
 ------------------------------
  a = 27          n = 25
  b = 1           o = 26
  c = 11          p = 5
  d = 10          q = 0
  e = 33          r = 21
  f = 9           s = 30
  g = 7           t = 35
  h = 24          u = 7
  i = 25          v = 1
  j = 0           w = 10
  k = 2           x = 1
  l = 18          y = 2
  m = 5           z = 0
------------------------------
 length  frequency
 ------  ---------
      1          3
      2         13
      3         24
      4         13
      5         10
      6          2
      7          5
      8          3
      9          1
     10          3
     11          2

Average word length = 4.24
```

Sample output for the input file testfile.txt

```
[you@onyx p4]$ java TextStatisticsTest

 Testing on data file:testfile.txt

 Passed! getCharCount()
 Passed! getWordCount()
 Passed! getLineCount()
 Passed! getAverageWordLength()
 Passed! Arrays frequencies
 Passed! Letter frequencies


 Testing on data file:etext/Gettysburg-Address.txt

 Passed! getCharCount()
 Passed! getWordCount()
 Passed! getLineCount()
 Passed! getAverageWordLength()
 Passed! Arrays frequencies
 Passed! Letter frequencies


 Testing on data file:etext/Alice-in-Wonderland.txt

 Passed! getCharCount()
 Passed! getWordCount()
 Passed! getLineCount()
 Passed! getAverageWordLength()
 Passed! Arrays frequencies
 Passed! Letter frequencies
```

Documentation

- All code must be properly formatted with proper indention and naming conventions. Class files should begin with capitals (EX: ProcessText). Variables and methods should begin with lower case and be descriptive (EX: wordCount, avgWordLength). All names should use "camel case" convention, that is the first letter of each word is capitalized. The only exception is the first letter of variables and methods.
- Include *Javadoc* Comments
  - Each class should include a javadoc comment before the class. This comment is a brief description of the class.
  - Each class comment must include the @author tag at the end of the comment. This will list you as the author of your software when you create your documentation.
- Have javadoc comments before every method that you wrote. Comments must include @param and @return tags as appropriate. To build and view your comments, run the following commands,

  javadoc -author -d doc *.java

google-chrome doc/index.html • Include a plain-text file called README that describes your program and how to use it. Expected formatting and content are described in the README template, https://drive.google.com/open?id=0B6lKrC3UayiJbHNZUE9xTFM1ZXc • An example is available here, https://drive.google.com/open?id=0B6lKrC3UayiJX29SNWZORVdFOWc
Grading Rubric
Coming soon
Submitting Your Project
See the project 3 submission link for details on how to submit your project