**`pWordCount` — Implementing `wc` in Java**

This assignment's **repo name** is `p01-WordCount`

## Learning Outcomes

After completing this assignment, a student should be able to

- **Implement** an "instantiate-run" Java program.
- **Separate** concerns by using multiple java **`package`**s.
- **Use** the `classpath` for multiple packages.
- (Review) **Process** *command-line arguments* – checking the number, treating them as file names.
- (Review) **Read** a text file *by line*.
- (Review) **Document** how to *compile* and *run* the program as well as how it was tested.

## Overview

Linux (Unix) has a utility program called `wc`, word count, that counts the number of lines, words, and characters in any number of files named on the command-line.

Students will implement a simplified version in Java: `WordCount` will

- *Verify* that there is exactly one (1) command-line argument; if there is a different number, program should terminate with an appropriate error message and a return code of 1.
- *Open* the named file for input; if the file does not exist then the program should terminate with an appropriate error message and a return code of 2. (**Notice** the error codes would permit the operating system to tell the difference between different abnormal terminations.)
- *Read* the input file by **line**, processing each line to count the number of *words* and *characters*. The program tracks the number of lines, words, and characters (remembering to count the end-of-line characters).
- *Print* the name of the file along with the various counts. The following output is for the provided `LineReportInterface.java` file. The results of `wc` on the same file are shown afterwards for comparison.

```
$ java <class path stuff> WordCount LineReportInterface.java
  LineReportInterface.java:
    lines: 22
    words: 89
    chars: 566
$ wc LineReportInterface.java
   22  89 566 LineReportInterface.java
```

`WordFrequency`: a program that expects exactly *one* (1) file name on the command-line, opens that text file, reading it *word-by-word*, keeping a count of how many times each unique word appears in the text and then prints out words and counts in *text-file order*.

To simplify implementation the program will define "word" as any collection of whitespace separated non-whitespace characters. Further, no attempt will be made to clean the input by normalizing capitalization or stripping punctuation. That means the "That" at the beginning of this sentence would count separately from **this** that.

**More detail:** `WordFrequency`, when run, checks that there is *exactly one* (1) command-line argument, terminating with a usage message and an **error** if not. With a single file name, `WordFrequency` attempts to open the named file for *input*; if the file *cannot* be opened, the program terminates with an appropriate message and an **error**. Terminating with an **error** is to terminate *abnormally*.

After the file is opened, the program reads each *word* and processes it. Processing a word means checking if the word is already in the word count list. If it is *not* already in the list, append it to the list with a count of 1 (it has been seen once); if it *is* already in the list, increment the count associated with it.

When the file is finished, print out all the words in the list along with their count. Then terminate *normally*.

**Note** "terminate normally" means to end with the program returning the value `0` to the operating system. This is what Java does when you run off of `main` **or** call `System.exit(0)`. The other two termination conditions described above are "abnormal"; that can be signaled to the operating system by exiting with a non-zero return code. That requires using `System.exit(<non-zero>)`.

## Procedure

1. Read the **whole** assignment. This is important for *every* assignment: it puts the task into your brain so that it can begin working on answering the questions. Of particular interest when you read are the SLO (first section above); gives you the links between the assignment to the big picture (learning computer science).

2. Before writing any code: create a *project* root directory. Initialize `git` in the root, put in a `.gitignore` file, and add a beginning `README`. Something like this is the first step of every program you will turn in during the semester.

3. Set up your *packages*. **Abstraction** is an important part of computer programming: the separation of concerns into small, single-concern *functions* or *classes* means that while programming one element you are protected from the details of any other part.

   This **separation of concerns** extends beyond the **`class`**, into a **`package`** of related classes. For example, a modern video game might have six *thousand* source files. To make navigating and debugging so much code it is typically broken up into packages like `sound`, `video`, `gameplay`, `networking`, *etc.*

   This assignment breaks up into three major areas of concern: keeping track of the count of **one** word; keeping track of a list of **all** of the words; and an application that handles counting command-line arguments, opening files, and dealing with the top-level errors that could occur.

   In Java a **`package`** corresponds to a *directory*. Just as the **`class`** `Snoopy` must be in the file named `Snoopy.java`, if that class's full name is `peanuts.Snoopy`, it belongs in the file named `peanuts/Snoopy.java`.

   With three distinct areas of concern, this program will have three packages:

   ```
   <project-root>
   |-- application
   |-- list
   `-- count
   ```

   Notice that the `<project-root>` is where ever you put your code in your directory structure. The three subdirectories are to be named exactly as shown.

   As has been discussed in class, this project will be *run* from the project-root. As discussed below, this means using the `-`**`class`**`-path` command-line argument for `java`/`javac`.

You **will** document this in your `README` file.

4. **Sketch** a solution to the overall problem from the **top down**:

   (a) Possible sketch of `main`:

```
// application.WordFrequency.main
check number of command-line parameters
if ! exactly one parameter
  print usage message
  terminate with non-zero value

create a WordFrequency object with the command-line file name
call WordFrequency.run
```

   The sketch shows where one possible error is handled. It is therefore possible to write just the `main` function and test it *before* even implementing the word count object or the linked list.

   (b) Possible sketch of `WordFrequency` constructor:

```
// application.WordFrequency.WordFrequency
save file name into an object field
```

   This is a very simple function; `run` will open the file and handle any corresponding errors. Separation of concerns makes functions simple.

   (c) Possible sketch of `WordFrequency.run`:

```
// application.WordFrequency.run
try
  open named file
  initialize WordList, w

  while file has a next word
    count word into w

  for every WordCount object in w
    println the pair
catch FileNotFoundException
  print appropriate error message
  terminate with non-zero value
```

   Did you catch that? Two new classes introduced, bringing the total to three:

   **`application.WordFrequency`** The class containing `main` and `run`. It is in its own package.

   **`count.WordCount`** Associate word (`String`) paired with its count (**`int`**). It will implement the `WordCountInterface` (which is provided in the `src` folder; you will need to put it in the correct package):
   You may, of course, write any other *private* functions you wish in `WordCount`. Except when calling the *constructor* you **must** use the **`interface`** in all other classes.

Look carefully at `WordCountInterface`: there are no `set*` methods. Think, for a minute, about why they are not needed (or *wanted*).

...

Since a `WCI` (`WordCountInterface`) object represents a `<word, count>` pair, it seems obvious that after creation there is no reason to *change* the word. What would it even mean to count the number of times `snake` appears in some sequence of words and then change the word to `chicken`?

Similarly, changing the count by setting it to a smaller number or incrementing it by more than one does not seem to fit the software model we are building of the contents of the file.

So the interface permits incrementing an existing `WCI` (when we see another instance of the word), checking if the `WCI` contains a given word, and to return a `String` representation of the `WCI`.

The two `get*` methods are there for testing/debugging. They *are* used in the provided `WordCountTestHarness.java` program; they are not needed in any other programs or classes.

**`list.WordList`** A *list* of `WCI` objects. Limiting access to the *interface* means that the list is implementation **agnostic** (it uses no knowledge about the implementation outside of the constructor call). Changing to a different implementation of the interface would not require more than a new **`import`** and recompiling the list with a call to the different constructor.

The interface expects list-like *behavior* but does not specify how the list is to be *implemented*. This **assignment** requires that `WordList` be implemented as a linked-list where the data in each node is the `WCI` object for some word.

(d) Assume the given interfaces are correctly implemented. Use `WordListInterface` to expand the sketch of `WordFrequency.run`. Once you read a word, how do you use the list object to determine whether to add it to the end or increment the existing entry.

What does the list printing code look like?

5. *Incremental development* (also sometimes referred to as *spiral development*) is the practice of building a program one small feature at a time. Figure out what the next feature should do and how you would verify that it works. Compile and test until it works and check the feature into version control (`git`). Lather, rinse, and repeat.

   One path to solving this assignment would be:

   - `application.WordFrequency.main` empty and compiling.
   - `main` checks number of arguments.
   - `WordFrequency` constructor that takes a `String` (file name).
   - `run` exists.
   - `main` completed with construction and call to `run`.
   - `run` opens the file or reports error.
   - Get `WordCount` working. Before you even get `WordList` working, `WordCountTestHarnes` can run unit tests[1] on `WordCount`.

   ---

   Seven steps in and the code does almost nothing. **But**, the foundation for the program is laid. Now it is time to write the linked-list and its functions. You can decide on the order of implementation for them.

   ---

   - Get `WordList` working. Can be initially tested with `WordListTestHarness`.
   - Hook processing each word to adding it to a `WordListInterface` object. Add the printing.
   - **Test** the finished program. Make sure you see all the errors that come from the command-line in your testing (and only when you expect them).
     Make sure to *document* your testing with what you expected, what you did, and what the results were. That goes in the `README`.

6. *Testing* is important. The `data` directory in this assignment has several files that you can run the program on. An example run (with the first few lines of output) is:[2]

```
java WordFrequency data/oneFish.txt
One : 1
fish, : 7
Two : 1
Red : 1
Blue : 2
Black : 1
Old : 1
```

[1] A *unit test* is a test run on a "unit" of a program, the very smallest testable part of the code. Here it is a *class* and the public *methods* it exposes. Unit testing will be a major part of your programming this semester.

[2] The execution line is stripped down to just show the name of the Java object with `main` in it. Assume that the correct `classpath` was specified.

```
New : 1
fish. : 1
This : 2
one : 4
has : 3
a : 6
little : 2
  ...
```

Take a look at the data file and see if you believe that of the 207 words in `oneFish.txt`, there are only 117 *distinct* words.[3] Also note that the order of the list is the order in the file of the distinct words.

You may want to think about what the output should look like before you program because if you *cannot* describe the correct output, it is unlikely that you understand the problem well enough to program the correct output.

You may want to develop some even shorter test data for quick feedback.

7. This assigment's **repo name** is `p01-WordCount`

   Commit your finished work (including `README` w/ all your documentation) to your local `git` repository. Then connect the local database to an on-line database on the Gitea server and `push` the conents.

   The *departmental* `git` server, running Gitea software (hence our referring to it as the "Gitea server") is at `https://cs-devel.potsdam.edu`. Log in to your account.

   You created the `F24-205-<CCID>`[4] organization in the first lab this semester. In it you will create a new, **private** *repository* with the name `p01-WordCount`. Gitea will respond with a set of commands to connect a local repo to the newly created one on the server. Use them to connect the repos and push your program.

   If, after you do this, you change your *local* repo contents, you will just need to rerun the `push` command to copy the changes up to the server. Dr. Ladd will retrieve your code from Gitea for grading. Remember that commits are *timestamped*; don't commit things after they are due. Pushing the database is separate from the `commit` operation; if it happens to be "late", as long as the commit is before the due time, all is well.

   The Gitea repository name is **specified** (and has nothing to do with local directory names) so that Dr. Ladd can automate harvesting all the repos for grading. Thanks for your consideration.

[3] How to figure those two numbers: Linux has a utility called `wc` (for *word count*). It give the number of lines, words, and characters in a file provided on the command line. That is where the 207 came from. The other comes from the number of lines that the sample solution produces when run.

[4] `CCID` – Campus Computer ID; your campus e-mail address w/o `potsdam.edu`.