`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) **Use** a `Scanner` to parse a `String`.
- (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
```

Define "word" as any collection of whitespace separated non-whitespace characters. This is the same definition used by `Scanner.next()` (how convenient).

Note: "terminate normally" means to end with the program returning the value `0` to the operating system; this is what Java does whenever 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 two major areas of concern: file handling (checking number of file names, opening file, reading file by line, all the errors) and line processing (counting words and (much easier) characters).

   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 two distinct areas of concern, this program will have two packages:

   ```
   <project-root>
   |-- application
   `-- counter
   ```

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

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

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

4. **Sketch** an *instantiate-run* (all our programs from this point forward will have this "shape") solution to the overall problem from the top, down. Here, top means `main` and down means moving to simpler and simpler methods.

   Remember to keep each method focused on exactly *one* responsibility. You can even use that responsibility to write the header comment for the method.

   (a) Possible sketch of `main`:

   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 counting or line processing.

   (b) Possible sketch of `WordCount` constructor:

   ```
   // application.WordCount.WordCount
   // responsibility: capture the filename, initialize counts
   save file name into an object field
   initialize the three counts in object fields
   ```

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

construct WordCount(file name)
call WordCount.run
```

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 `WordCount.run`:

```
// application.WordCount.run
// responsibility: open file or handle errors
try
  open named file into a scanner

  process the file wrapped in the scanner

  print the results
catch FileNotFoundException
  print appropriate error message
  terminate with non-zero value
```

```
// application.WordCount.processFile
// responsibility: read file by line; process each line
while scanner has a next line
  read line
  LineReportInteface i = new LineReport(line)
  update all three counts from i
```

5. Did you catch that? `LineReport`, implementing `LineReportInterface` is another class you need to write.

   Let's look at the **`interface`**:

   ```
   package counter;

   /**
    * Interface for reporting on a line's word and character counts.
    *
    * This interface does not require a toString method as this is an
    * internal type constructed from a String (the line) and reporting
   ```

```
 * back the number of words and number of characters in the line.
 */
public interface LineReportInterface {
  /**
   * Get the number of (whitespace separated) words.
   * @return number of words
   */
  int getNumberOfWords();

  /**
   * Get number of characters.
   * @return number of chacaters.
   */
  int getNumberOfCharacters();
}
```

Your class `LineReport` needs two (three, if you count the constructor) methods: one to return the number of characters (probably just the length of the string it was constructed with, right?) and the number of *words* in the line. You will want to use a `Scanner` on the line to count them.

6. 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>`[1] 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.

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