Assignment 5: CREW (concurrent-read exclusive-write)

Concurrent Read Exclusive Write

Concurrent-read exclusive-write (CREW) is a fundamental concurrent programming paradigm. As the name suggests, it is about implementing a system that allows for multiple threads to read a shared object, but only allows one thread to write to a shared object at any given moment. In this part of the assignment, your task is to write a class called **`WordCounter`**, whose work is described below:

1. This class will read a list of plain text files, all of which are in a given folder.
2. Based on the word counts, it will generate a table that shows which word appears how many times and in which file.

Let us look at a worked out example. Suppose there are three text files:

- text-a.txt: "There are so many words in this file. Well, actually not that many if you really start counting them."
- text-b.txt: "The words are scattered everywhere, and in no order whatsoever. What. So. Ever. How will counting them help?"
- text-c.txt: "So, what can you do?"

The output *table* is shown below.

For the purpose of this assignment, you need not worry about hyphenated words (e.g., "e-mail"), apostrophes (e.g., "it's" or "Ma'am"), 'word-like' objects (e.g., a date written as "01-12-2020"), acronyms and abbreviations (e.g., "Mr.", "U.S.A."), or quotes.

Note the following features of this output:

- the words in column one are alphabetically ordered
- the columns corresponding to the files are alphabetically ordered by file name
- the first column's width is the longest word's length plus one character (so that there's a single space between the longest word and the first number)
- all the columns specific to a file have the same width, and counts the number of occurrences of a word in *that* file
- the last column counts the total number of occurrences of a word across *all* files
- all words are lowercased
- punctuation (`. , : ; ! ?`) has been completely ignored

	text-a	text-b	text-c	total
actually	1	0	0	1
and	0	1	0	1
are	1	1	0	2
can	0	0	1	1
counting	1	1	0	2
do	0	0	1	1
ever	0	1	0	1
everywhere	0	1	0	1
file	1	0	0	1
help	0	1	0	1
how	0	1	0	1
if	1	0	0	1
in	1	1	0	1
many	2	0	0	2
no	0	1	0	1
not	1	0	0	1
order	0	1	0	1
really	1	0	0	1
scattered	0	1	0	1
so	1	1	1	3
start	1	0	0	1
that	1	0	0	1
the	0	1	0	1
them	1	1	0	2
there	1	0	0	1
this	1	0	0	1
well	1	0	0	1
what	0	1	1	2
whatsoever	0	1	0	1
will	0	1	0	1
words	1	1	0	2
you	1	0	1	2

The **`WordCounter`** file must generate this output for the set of all files in a specific folder. To accomplish this, it must read each text file in a separate thread. Of course, all the words and word-counts from individual files have to consolidated to write the final table. This is where you must carefully put together the results obtained by the different threads.

First, you should test this code without multithreading (i.e., the number of threads is just one -- the main thread) and then, with multithreading, to see what kind of speed boost (if any) you are getting. This will also help you identify bugs in your code. If something is going wrong with a single thread, then of course, the multithreaded version will also have bugs (and debugging single-threaded programs are much easier).

The basic structure of WordCounter is given below. Your class must have the constants **`FOLDER_OF_TEXT_FILES`**, **`WORD_COUNT_TABLE_FILE`**, and **`NUMBER_OF_THREADS`**. These are the only values we will change to run your code, so you have to be absolutely sure that your code is modular and robust enough to not crash if these details are changed when the code is run on another machine. For grading, only these constants will be changed, and your code will then be run (the graders will not modify anything else).

```
public class WordCounter {

        // The following are the ONLY variables we will modify for grading.
        // The rest of your code must run with no changes.
        public static final Path FOLDER_OF_TEXT_FILES  = Paths.get("..."); // path to the folder where input text files are located
        public static final Path WORD_COUNT_TABLE_FILE = Paths.get("..."); // path to the output plain-text (.txt) file
        public static final int  NUMBER_OF_THREADS     = 2;                // max. number of threads to spawn

        public static void main(String... args) {
            // your implementation of how to run the WordCounter as a stand-alone multi-threaded program
        }
    }
```

NOTE:

- You are not allowed to use the **`parallelStream()`** method for this assignment
- As with the previous assignment, your language level in IntelliJ IDEA must be set to JDK 1.8.
- Please be VERY careful that you do not use hard-coded strings to specify any file or folder in your code. If you do that, you may end up in a situation where your code (i) runs on one operating system but not on another, or (ii) runs on your computer, but not on any other system.

Rubric

1. WordCounter table generation follows the format and all columns are well-aligned: 20 points
2. Words are counted after discounting uppercase/lowercase differences: 10 points
3. Punctuation is filtered out, and does not appear in the output table: 10 points
4. WordCounter results are correct in the single-threaded scenario: 10 points
5. WordCounter results are correct in multithreaded scenarios with a small number of threads (say, around 5 threads): 20 points
6. Results are correct with a large number of files (> 50) in the folder: 10 points
7. Results are correct with a large number of threads (> 10): 10 points
8. Speed improvement (even if not very significant) with multithreading: 10 points

Submission details

- The single Java file, `WordCounter.java`.
- All submissions are due on Blackboard by **11:59, Dec 08 (Tuesday)**.