Project 1: Building an Index

DS 730

Overview

In this project, you will be working with input, output, classes in the Java Collections Framework and threads. You are expected to investigate the Java library and use the classes and methods in the Collections library as much as possible. You will also need to look into the File class to see how to use folders and files. I would encourage you to use the Scanner class to read in data from the files as it is very easy to use. However, a BufferedReader will provide you with better runtimes if you use it correctly.

In short, you are creating a word index. An index helps you find information in your files faster. An index can also be very useful when comparing how similar two documents are to one another. One way of determining how similar two documents are is to compare the number of uncommon words they share. This might be useful in a recommender type system. For example, if I really like a book that often contains the words *California, surfing, sunshine* and *beach*, then odds are good I will like another book that contains a similar number of those keywords.

Project Tasks

You must implement the following steps.

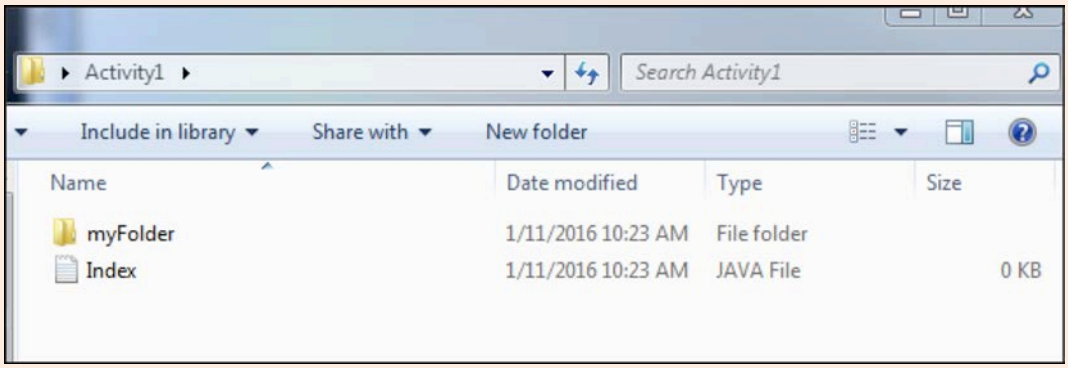
1. You will write a program that goes through a text file and creates a word index of every word in the file. The index will be the "page" that a particular word is found on. Since a text file only contains text and does not contain any metadata, "pages" will be created depending on the number of characters read in so far (not including delimiters). The number of characters will be specified at runtime.
2. The user will specify 2 command-line arguments. The first argument is the folder where all of the text files are saved and the second argument is the number of characters that represent a page.

Example:

Assuming the Java file is called **Index.java**, the following command:

```
java Index myFolder 100
```

indicates that all input files are stored in a folder called **myFolder** that is in the same folder as your Java class file. See the following screen shot for the file hierarchy. All text files are in **myFolder**.



The number of characters on each page goes up to but doesn't exceed 100 actual characters. For instance, if you have read in 98 characters so far and the next word is **badger**, the word **badger** would be the first word on the next page. To make this a bit easier, you can ignore delimiter characters.

3. Create a word index of each file and store that index into an output file. If your input file is called **a.txt**, then your output file must be called **a_output.txt**. You will create an output file for each input file. Your word index should be created in the following way:

    a) Read in all words. A word is any consecutive sequence of letters, numbers, apostrophes, special symbols, etc. The only things that delimit words are a space, tab and new line (i.e. whitespace). In my sample files, I am using the default delimiters specified by the Scanner class. If you use something different, you may get different results. You should store the words into one of the following Collections: TreeSet, HashSet, TreeMap or HashMap. If you don't, your code will likely run very slow.

    b) Along with reading in all of the words, remember which "page" the word was on.

    Example: Going back to the example in step 2, where pages contain 100 characters, a word that is one of the first 100 characters to be read in is considered to be on page 1. A word that is one of the second 100 characters is considered to be on page 2. As said before, if a word goes over the 100-character limit for that page, it is "moved" to the next page. You can assume all words are less than 100 characters.

c) For each file, after you have read in all of the words, you should write out your word index to that files output file. You should write out each word that appears in the file and for each word that you write out, you must also write what page(s) that word appears on. You must write out the words in alphabetical order. Your output should be the word, followed by a space, followed by the page(s) that word appeared on where each page is separated by a comma (see sample input/output file posted online).

> **Note:** Words that appear multiple times on the same page should *not* show up in the final output. For example, if the word **cat** appears on page 3 a total of 4 times, page 3 should only show up once in the output.

4. You must solve this project in *2* different ways. The first way is *without* using threads. You must time your code and determine how long it took to solve without using threads.
5. Once you have your solution to 4, *modify it so that it uses threads* in some fashion. The most natural way to use threads is to create a new thread for each file you read in. If you are testing this on a machine with multiple cores, you should notice a significant increase in time (assuming you are using large enough files).
6. Record your running times in a separate document. You can simply create a table and tell me what your input was and what your timings were for each input and for each method used. You are allowed to use any machine you want to test your code but the timings must be done on the virtual desktop machines to ensure we have similar results.

> **Note:** I have included a .zip file containing several sample input and output files. You can find this file in the online course. If your program creates a word index that is identical to mine, you likely wrote your program correctly.

Submitting Your Work

When you are finished, submit *a .zip file* to the **Project 1 dropbox** that includes:

- Your code.
- The document that includes your running time results (see step 6). You can simply create a table and tell me what your input was and what your timings were for each input and for each method used.