Project 1 Inverted Index

For this project, you will write a Java program that recursively processes all text files in a directory and builds an inverted index to store the mapping from words to the documents (and position within those documents) where those words were found. For example, suppose we have the following mapping stored in our inverted index:

```
{
  "capybara": {
    "mammals.txt": [
      11
    ]
  },
  "platypus": {
    "mammals.txt": [
      3,
      8
    ],
    "venomous.txt": [
      2
    ]
  }
}
```

This indicates that the word `capybara` is found in the file `mammals.txt` in position 11. The word `platypus` is found in two files, `mammals.txt` and `venomous.txt`. In the file `mammals.txt`, the word `platypus` appears twice in positions 3 and 8. In file `venomous.txt`, the word `platypus` is in position 2 in the file.

Functionality

For this project, your code must traverse a directory and process all text files found in that directory (including all subdirectories). For each text file, you must parse each line into words. For each word, you must store a mapping of the word to the file and position that word was found in a custom inverted index data structure.

Specifically, the core functionality of your project must satisfy the following requirements:

- Create an custom **inverted index** data structure that stores a mapping from a word to the file(s) the word was found, and the position(s) in that file the word is located. *This will require nesting multiple built-in data structures.*
- Store the normalized relative path for file locations as a `String` object. Do not convert the file paths to absolute

paths!

- Positions stored in your inverted index should start at 1. For example, if a file has words "ant bat cat", then "ant" is in position 1, "bat" is in position 2, and "cat" is in position 3.
- Traverse a directory and its subdirectories and parse all of the text files found. Any file ending in the `.txt` extension (case-insensitive) should be considered a text file.
- Use the `UTF-8` character encoding for all file processing (including reading and writing).
- Replace all characters except letters and digits with an empty string. This includes the `_` underscore character as well. For example, the word `age_long` should be seen as `agelong` since the `_` underscore will be replaced with an empty string.
- Separate text into words by any whitespace character, including spaces, tabs, and newline characters.
- Words must be case-insensitive. For example, the words APPLE, Apple, apple, and aPpLe should all be seen as the same word.
- Process command-line parameters to determine the directory to parse and output to produce. See the **Execution** section below for specifics.
- Output the inverted index to a text file if the proper command-line parameters are provided. See the **Output** section below for specifics.

You must satisfy the core functionality prior to submitting your project for code review. See the Project README for specifics.

Design

In addition to the functionality requirements, you should consider the following when designing your project:

- Your program should support large text files. As a result, you should **not** read the entire file into memory at once. Instead, read a single line from the file into memory at a time.
- Your program should be object-oriented. For example, you should separate directory traversing, file parsing, and data maintenance into different classes.
- You program should protect the integrity of your inverted index, making sure all private data members are properly encapsulated. For example, do not return a reference to a private mutable data member in a public method.

These requirements are **not** necessary to be eligible for code review. You should only worry about these requirements once your core functionality is complete.

Execution

Your `main` method must be placed in a class named `Driver`. The `Driver` class should accept the following command-line arguments:

- `-input directory` where `-input` indicates the next argument is a directory, and `directory` is the input directory of text files that must be processed
- `-index filepath` where `-index` is an *optional* flag that indicates the next argument is a file path, and `filepath` is the path to the file to use for the inverted index output file. If the `filepath` argument is not provided, use `index.json` as the default output filename. If the `-index` flag is not provided, do not produce an output file.

The command-line flag/value pairs may be provided in any order. Your code should support all of the command-line arguments from the previous project as well.

⚠️ The `Driver` class should be the only file that is not generalized and specific to the project. If the proper command-line arguments are not provided, the `Driver` class should output a user-friendly error message to the console and exit gracefully.

Examples

The following are a few examples (non-comprehensive) to illustrate the usage of the command-line arguments. Consider the following example:

```
java Driver –input input/index/simple
            –index index–simple.json
```

In this case your program should build an inverted index from all of the text files in the `input/index/simple` directory. The inverted index should be output to the file `index–simple.json` in the current working directory.

Next, consider the following example:

```
java Driver –index –input input/index/simple
```

Your program should execute the same as before, except that it will output the inverted index to the default file `index.json` instead. Notice how the arguments may appear in any order.

Next, consider the following example:

```
java Driver –input input/index/simple
```

In this case, your program still executes as before but produces no output files. **Your code must still, however, build the inverted index!** This is useful later for testing the efficiency of your code.

Output

The output of your program should be a file that contains the contents of your inverted index in alphabetically sorted order as a nested JSON object using a "pretty" format using 2 spaces for indentation. For example:

```
{
  "capybara": {
    "mammals.txt": [
      11
    ]
  },
  "platypus": {
    "mammals.txt": [
      3,
      8
    ],
    "venomous.txt": [
      2
    ]
  }
}
```

Each path should be a normalized relative path. Make sure there are no trailing commas after the last element. See the JSON files in the `expected` directory for more examples.

Hints

It is important to develop the project **iteratively**. In fact, you may already have certain components complete thanks to the homework assignments. One possible breakdown of tasks are:

- Create code that handles parsing command-line arguments, so that you can easily retrieve the directory containing the text files to parse. Test your code.
- Create code that is able to traverse a directory and return a list of all the text files found within that directory. Test your code.
- Create code that handles parsing a single text file into words. Test your code.
- Create code that handles storing a word, file path, and location into an inverted index data structure. Test your code.
- Integrate your code.
    - Make your code that parses a text file into words store words in an inverted index data structure.
    - Make your code that traverses a directory work with the code that parses a text file and stores words into an inverted index data structure.
    - Make your code that parses command-line arguments work with the code that traverses a directory, and so on.

The important part will be to **test your code as you go**. The tests provided only test the entire project as a whole, not the individual parts. You are responsible for testing the individual parts themselves.

💡 These hints may or may *not* be useful depending on your approach. Do not be overly concerned if you do not find these hints helpful for your approach for this project.

Submission

💡 See the Project README for additional details on testing, submission, and code review.