CIT 5940 - Module 4 Programming Assignment
Sentiment Analysis

Contents

Assignment Overview

In this assignment, you will write a program that analyzes the sentiment (positive or negative) of a sentence based on the words it contain by implementing methods that use the `List`, `Set`, and `Map` interfaces.

Learning Objectives

- Become familiar with the methods in the `Java.util.List`, `Java.util.Set`, and `Java.util.Map` interfaces.
- Continue working with abstract data types by using only the interface of an implementation.
- Apply what you have learned about how lists, sets, and maps work.
- Gain a better understanding of the differences between lists and sets.
- Demonstrate that you can use lists, sets, and maps, to solve real-world problems.
- Gain experience writing Java code that reads a file.

Advice

We highly encourage you to utilize the course discussion forum to discuss the project with classmates and TAs. Before asking the question, please search in the forum to see if there are any similar questions asked before. Please check *all* suggestions listed there.

For this and all other assignments in this course, we recommend that you work on your local computer using an integrated development environment (IDE) such as Eclipse, IntelliJ, Visual Studio Code, or whatever IDE you used for any prior Java courses that you are familiar with. Although you will need to upload your solution to Codio for grading (see the Submission Section below) and could develop your solution on that platform, we recommend using industry standard tools such as Eclipse or IntelliJ so that you become more used to them and can take advantage of their features.

If you have trouble setting up your IDE or cannot get the starter code to compile, please post a note in the discussion forum and a member of the instruction staff will try to help you get started.

We expect that you will create your own JUnit tests to validate your solution before submission.

1 Setup

1. Begin by downloading the `CIT594-sets-maps.zip` archive from Canvas, then extract the contents. This contains the unimplemented methods for the code that you will write in this assignment:

   (a) `Sentence.java`, which is a class representing a "sentence". You MUST NOT modify this file.

   (b) `ObservationTally.java`, which is a class representing the tally of observations. You MUST NOT modify this file.

   (c) `Analyzer.java`, which contains the unimplemented methods that you will write in this assignment. We have provided a `main` method you can use to run the analysis. This is the ONLY file you are permitted to modify.

   (d) `reviews.txt`, which contains example sentences for analysis. You can test your program with this file.

2. Compile this code before making any changes. This will ensure your local environment is configured correctly.

3. Review the current implementation. You'll want to understand the various parts of the code and how they work together before making changes!

2 Requirements

2.1 Structure and Compiling

1. You MUST use a JDK version of 17 or higher.

2. You MAY use a JDK version higher than 17, but you MUST set the Java language level to 17 for compatibility with Codio.

3. You MUST NOT change any of method signatures for any of the provided methods. This includes the parameter lists, names, and return value types, etc.

4. You MUST leave all classes in the default package.

5. You MUST NOT create additional `.java` files for your solution.

6. You SHOULD create JUnit tests to help validate your code. These tests SHOULD NOT be a part of your solution. We do expect you to test your code thoroughly with JUnit tests before submission.

7. You MUST follow our directives in the starter code comments. This means if we ask you to not change a variable's value, you MUST NOT change that variable's value.

8. You MAY create additional helper functions, as long as they meet the other requirements of the assignment (e.g. doesn't crash with invalid inputs, etc.).

9. You MUST fill out the required Academic Integrity signature in the comment block at the top of your submission files.

2.2 Functionality Specifications

2.2.1 Definitions

- **valid line**: a line starting with an optional sign character (`+` or `-`), followed by a single digit representing a valid `score` (integers from `-2` to `2` inclusive), followed by a single **whitespace** character, followed by a **statement**. The regular expression for a **valid line** is:

```
"^(?<score>[+-]?[0-2])\\s(?<text>.*)$"
```

- **statement**: a string, which may be empty, and may contain `0` or more **whitespace** separated **tokens**, each of which may be a **word**.

- **sentence**: an object of type `Sentence` containing a `text String` and an integer sentiment `score`.

- **token**: all of the non-**whitespace** characters between **whitespace** characters or at the beginning of a **sentence** or **statement**.

- **valid word**: a **token** starting with one **letter**. Any additional characters may be **letter**s or any other non-**whitespace** character.

- **letter**: any character for which the method `java.lang.Character.isLetter` returns `true`.

- **whitespace**: any character for which the method `java.lang.Character.isWhitespace` returns `true`.

- **`ObservationTally`**: an accumulator object of the `ObservationTally` class representing a **word**'s accumulated context `score`s from all of its appearances which have been analysed so far. An `ObservationTally`'s `count` is the total number of times the **word** has been seen so far, and the `total` is the sum of every occurrence of the **word**'s sentiment `score` seen so far.

2.2.2 General Requirements

1. Your methods MUST NOT crash if a user enters an invalid input. Remember you do not have control over what a user inputs as arguments. Instead, you MUST return sensible output even if the input is invalid. For example, methods that return some type of `Collection` MUST return an empty collection; methods that return numerical types MUST return `0`.

2. If the input contains invalid items, such as `null` strings or non-**word token**s in non-`null` strings MUST be ignored; the method MUST continue processing subsequent valid items.
3. **token**s and **words** MUST be converted to lowercase (this will simplify case-insensitive comparisons).
4. **words** MUST NOT be altered in any other way

2.2.3 `Analyser.readFile`

1. this method MUST take a (`nullable`) filename
2. it MUST return a non-`null` `List` of `Sentence` objects
3. You MUST select a class that implements `java.util.List` as the return object
4. the `List` MUST contain all of the **valid line**s
5. it MUST preserve the original order of the **valid line**s as they appear in the input file.
6. **invalid line**s MUST be ignored and not added to the output `List`
7. If the input filename is `null`, or if the file cannot be opened for reading, this method MUST return an empty `List`
8. The first **whitespace** character on the line is the separator between the score and the text. This character MUST NOT be considered part of either the score or the text. We ask that you test with that expression before asking us questions about what is considered valid or invalid.

   As an example, for the **valid line**

   ```
   2 I am learning a lot .
   ```

   the `score` field of the `Sentence` object should be set to `2`, and the `text` field should be

   ```
   "I am learning a lot ."
   ```
9. You MAY choose to use a regular expression for this assignmment. The expression in the **valid line** definition above is provided as a formal exact definition. You can test regular expressions on regex101.com and in Java using `jshell`.
10. it MUST NOT alter the statement text

2.2.4 `Analyser.allOccurrences`

1. this method MUST take a (`nullable`) `List` of (`nullable`) `Sentence` objects
2. it MUST output a non-`null` `List` containing every **word** encountered in the input `List`
3. You MUST select a class that implements `java.util.List` as the return object

4. the **word**s MUST be lowercase
5. it MUST preserve the order of the **words** as they are encountered in the input
6. it MUST ignore both `null Sentence` objects and invalid **words**
7. if the input parameter is `null`, it MUST return an empty `List`

2.2.5 `Analyser.uniqueWords`

This method is exactly the same as Analyser.allOccurrences, except for the following:

1. the output MUST be a `Set` instead of a `List`. This means that the output will not have duplicates, and does not need to preserve the original order

2.2.6 `Analyser.wordTallies`

1. this method MUST take a (`nullable`) `List` of (`nullable`) `Sentence` objects
2. it MUST output a non-`null` `Map`, whose keys are **word**s and whose values are the final scores, represented by an `ObservationTally` for that **word**.
3. You MUST select a class that implements `java.util.Map` as the return object
4. if the input `List` of `Sentences` is `null` or empty, this method MUST return an empty `Map`
5. if a `Sentence` object in the `List` is `null` or if the text of a `Sentence` is `null`, then this method MUST ignore it and continue processing the remaining `Sentences`
6. if a word appears in multiple `Sentences`, or multiple times in the same `Sentence`, then the corresponding `ObservationTally` MUST accrue multiple scores, one for each occurance
7. do not assume that the strings in the `Sentence` objects have already been converted to lowercase

2.2.7 `Analyser.calculateScores`

1. this method MUST take a (`nullable`) `Map`, with (non-`null`) **word** keys and (non-`null`) `ObeservationTally` values
2. it MUST output a non-`null` `Map` with the original **word** as the key and the **word**'s average sentiment score as the value
3. You MUST select a class that implements `java.util.Map` as the return object
4. if the input `Map` is `null`, it MUST return an empty `Map`

5. You MUST use the the `ObservationTally`'s `calculateScore` method to get the average sentiment score for that **word** from its previously recorded context scores
6. You MUST create a new `Map` with the appropriate keys and values, as opposed to modifying the input `Map`

2.2.8 `Analyser.calculateSentenceScore`

1. this method MUST take a (`nullable`) `Map`, with (non-`null`) **word**s keys and (non-`null`) sentiment scores, and an arbitrary statement text `String`
2. it MUST output the sentiment score for the input statement text
3. the output sentiment score MUST be the arithmetic mean score for all of the **valid word**s
4. the sentiment scores for individual words MUST come from the input `Map`
5. it MUST NOT filer duplicate words in the input statement text
6. it MUST be case insensitive
7. if the input `Map` is `null` or empty, this method MUST return `0`
8. if the input `Map` does not contain any **valid word**s, this method MUST return `0`

2.3 Writing Test Cases

Upon completion of each functional requirement, always conduct thorough testing of the corresponding functions. Consult the guidelines on composing JUnit tests and creating test cases in Module 1. You MUST develop JUnit tests to cover all corner cases you can think of for each of your implemented function in order to get full credit. This testing component carries a weight of approximately 5% towards the total assignment score.

3 Submission

3.1 Pre-submission Check

Before you submit, please double check that you followed all the instructions, especially the items in the Structure and Compiling section above.

3.2 Codio Submission

1. When you are ready to submit the assignment, go to the Module 4 Programming Assignment Submission item and click the button to go to the Codio platform.

2. Once you are logged into Codio, read the submission instructions in the `README` file. This should be the first thing that appears in the window.
3. Upload your solution and any JUnit test files to the "submit" folder.
4. In the menu bar, select Run JUnit Tests. This will run any unit test files that you have uploaded with your submission. Note that there are no pre-submission checks provided. However you can use this feature as a basic validation check to be sure your code compiles.
5. When you're ready to submit, go to Education and select Mark As Completed. Confirm at the prompt.
6. You will see quite a bit of output, even if all the tests pass. At the bottom of the output, starting at `YOUR AUTOGRADING RESULTS BELOW` you will see the number of successful and failed test cases.
7. If you want to update your code and resubmit, for example if you did not pass all of the tests, un-mark the assignment as complete. You will then be able to edit your code, run your JUnit tests, and resubmit.

4 Grading

4.1 Grading Overview

1. There is a peer review for this assignment. Other students will be reviewing your code and providing feedback. While their review will not directly impact your grade, we still expect you to use best practices (comments, indentation, etc). There is a separate assignment in Canvas for you to submit your code for peer review and review the submissions of your classmates.
2. There are no hidden tests. You will be able to see the name of each test, if it passed or failed, and the associated point value.
3. Failed tests will have brief feedback about the specifics of the failure.
4. We will not provide the exact test cases.
5. You have unlimited submissions, until the due date of the assignment as noted in the Syllabus (or your approved extension).
6. The score at the due date is final. Scores will sync to Canvas relatively immediately.

4.2 Rubric Items

This assignment will be marked on 6 rubric items.

- **`readFile` method** : 22 points

- **`allOccurrences` method** : 7 points
- **`uniqueWords` method** : 6 points
- **`wordTallies` method** : 27 points
- **`calculateScores` method** : 12 points
- **`calculateSenteneScore` method** : 22 points

Total : 96 points (+ 6 points for JUnit test cases)

5 Additional Resources

1. Java's `List` interface
   https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/List.html
2. Java's `Set` interface
   https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Set.html
3. Java's `Map` interface
   https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Map.html
4. Java `Characters`
   https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Character.html
5. How to read files in Java
   https://docs.oracle.com/javase/tutorial/essential/io/file.html
6. Java regular expression `Pattern`
   https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/regex/Pattern.html
7. regex101.com
   https://regex101.com
8. jshell reference
   https://docs.oracle.com/en/java/javase/17/jshell/introduction-jshell.html

Good Luck!