Project 1: TextFilter

Release date: 6/23

Due date: 6/30

Goals:

- Critical Thinking
- Problem Solving
- Parsing and modifying complex Strings

Prerequisites:

- Primitives
- Strings
- Scanners
- Conditionals
- Loops

Description:

Text manipulation is one of the most commonly needed routines in computer science. Consider a company like Twitter. They see millions of tweets every day, of all shapes and sizes. It may be incumbent upon them to censor out certain (vulgar) words in certain circumstances, or to provide text editing features to make composing tweets easier (such as find and replace). Consider a game like World of Warcraft, which may want to censor out personal information (such as email addresses) from its in-game chat window in real-time, in order to prevent breaches in privacy.

Several years after graduating from Purdue, you and a group of your old buddies from college decide to make a startup social networking company. Looking back at what you learned, you think fondly of your days in CS180, and decide to implement the new platform using Java :)

After launching your network, your users have been outspoken at their delight for a new and unique experience. However, users would like to share information privately between other users - or to the world - and have tools available to filter that information as they see fit. Your company has decided to implement some of the most requested features as an addition to your social network, which are explained in more detail below.

This project is separated into 4 parts that you have to complete:

1. **Part 1: Censoring Words**
2. **Part 2: Replacing Words**
3. **Part 3: Censoring Personal Information**
4. **Part 4: Filtering multiple times**

Part 1 - Censoring Words

In this part, you will be taking a **single, non-empty line** from the user, as well as a keyword (or key-phrase) from the user. You will then be replacing any instance of the keyword (or key-phrase) in the sentence with X's. Consider the following examples:

Example 1:

An Uncensored Passage:

- I scream, you scream, **we all scream for ice cream**!

The same passage, but Censored with the keyword "we all scream for ice cream" :

- I scream, you scream, **XXXXXXXXXXXXXXXXXXXXXXXXXXX**!

In this case, we are censoring the key-phrase, spaces and all, with X's.

Example 2:

An Uncensored Passage:

- How much **wood** would a woodchuck chuck if a woodchuck could chuck **wood**? He would chuck, he would, as much as he could, and chuck as much **wood** as a woodchuck would if a woodchuck could chuck **wood**.

The same passage, but Censored with the keyword "wood" :

- How much **XXXX** would a woodchuck chuck if a woodchuck could chuck **XXXX**? He would chuck, he would, as much as he could, and chuck as much **XXXX** as a woodchuck would if a woodchuck could chuck **XXXX**.

In this phrase, we are searching for the word "wood" within the passage and replacing each instance of that word with X's.

Note:

- Words may be censored if they are surrounded by whitespace
- Words may be at the beginning and end of the String
- Words may be followed by a period, comma, question mark or exclamation point
  - (. , ? !)
  - This is only true if these characters come **after** the Word!
- Words that are part of a larger word should **not** be censored.
  - (i.e. "wood" and "woodchuck" from the example above)

Part 2 - Replacing Words

In a similar fashion to Part 1, in Part 2 you will be finding keywords or key-phrases in a sentence or passage. The only difference is that this time, you will also ask what the user wants to put in place of this keyword/key-phrase.

Consider the following example, which simply replaces "chickens" with "turtles" :

An Uncensored Passage:

- Don't count your **chickens** before the eggs have hatched!

The same passage, but Censored with keyword "chickens" and filter "turtles"

- Don't count your **turtles** before the eggs have hatched!

And a more complex example:

An Uncensored Passage:

- I recently upgraded my desktop computer with a new **i7 processor. Now I can render my recording projects even faster!**

The same passage, but Censored:

- I recently upgraded my desktop computer with a new **GTX 1080 graphics card. Now I can play my favorite games in 4K Ultra HD!**

Can you spot the difference?

Key-phrase → **"i7 processor. Now I can render my recording projects even faster!"**
Filter → **"GTX 1080 graphics card. Now I can play my favorite games in 4K Ultra HD!"**

Note: The same rules apply for spacing, location and special characters as in **Part 1**!

Part 3 - Censoring Personal Information

In the world today, online privacy is a serious concern. Your users have outcried at the leaks of personal information that hackers have posted for the world to see. Your mission in this task is to find that personal information and censor it before it even gets posted to your social network. Your users will be happy, leakers will be foiled, and privacy advocates will rejoice for your efforts!

This task requires careful attention to detail and formatting. Users will be entering lines that may contain personal information related to Names, Emails and Telephone Numbers. Your job is to parse through their input and censor that information accordingly.

In the skeleton code we have given you below, included the way that input strings should be formatted.

1. Users will enter a line of information or text, and hit enter to add another line to the input.
    - After each time the User hits the enter key, we will add the new text to a string that will store all of the input.
    - After doing this, we will also add a newline character to separate the lines of input.
2. Once the user enters an empty line, we will stop reading input.

For example, if the user were to type in:
Name: Purdue Pete
Email: pete@purdue.edu

The String we would store, **including** newline characters, would be:
Name: Purdue Pete\nEmail: pete@purdue.edu\n

After obtaining input, it is your job to censor any personal information. First we will look at an example, and then outline what you explicitly need to censor.

An Uncensored Input:
School Mascot:
Name: Purdue Pete
Email: pete@purdue.edu
Phone: 456-832-7180

The newly Censored input:
School Mascot:
Name: P***** ***e
Email: p***@p*****.edu
Phone: ***-***-7180

When censoring information, we will be using the following format:

- "Type: Information_here\n"
    - Type we want to filter can be "Name", "Email" or "Phone"

    - There will always be a colon and a space between the “Type” and “Information_here”
    - Each line will end with a newline character (\n)
- When Censoring **Names**:
    - Only the first letter of the first name and last letter of the last name should remain uncensored
    - All other characters should be censored
    - There will only be first and last names, no middle names
    - The space between the first and last name should **NOT** be censored!
- When Censoring **Email Addresses**:
    - In the format <identifier>@<domain>**.com** :
        - Only the first letter of the **identifier** should remain uncensored
        - Only the first letter of the **domain** should remain uncensored
        - The extension, I.e. .com, .edu, .net, etc should remain uncensored
- When Censoring **Telephone Numbers**:
    - Only the last 4 digits should remain uncensored
- **All characters that are censored should be replaced with an asterisk (*)**

Note:

- It is safe to assume that during testing, inputs will be always be given in this format and that they will not be malformed.
- There might be lines with different things in between the actual ones you need to censor. Make sure you keep those lines intact in your output.

Part 4 - Filtering multiple times

- For this part, you will simply need to update the value of the boolean keepFiltering toward the bottom of the skeleton code.

Coding:

This project is split up into four parts: Censoring and Replacing Words, Censoring Personal Information and Filtering multiple times. Each one of these parts was explained in detail above, and will require you to think about the reading, parsing and changing of Strings using the programming tools you have learned thus far.

If you have not read or skipped the instructions for Parts 1, 2, 3 and 4 above, please go back and read them before continuing!

You are being given a skeleton code that outlines what parts you need to implement and suggestions on where you should implement them.

- Please read over this file as well as the rest of the handout before you begin coding.
- Areas where you will need to implement your code have been marked with TODO comments.
  - **DO NOT alter the given skeleton code outside of the TODOs. Doing so may result in the failure of test cases during grading!**
- If you skipped to here from the top, please go back and read the entire handout.

Notes:

- We suggest that you work in chronological order, that is, starting with Part 1 and ending with Part 4.
- If you are curious, or would like to use the powerful Java API String functions , you can view the String API here:
  - https://docs.oracle.com/javase/8/docs/api/java/lang/String.html

Appendix

Error messages from Assert Class:

When looking at the error messages displayed by Vocareum when you fail certain test cases, you might get something like this:

```
tester(Test): Ensure that you can replace words at the
beginning and/or end of a passage.
     expected:<...sage you would like [filtered:
    Please enter the word you would like to replace:
    Please enter word you would like to insert:
    Uncensored: ]
    Hello there I would...> but was:<...sage you would like [to
be filtered:
    Please enter the word you would like to replace:
    Please enter word you would like to insert:
    Uncensored :]
    Hello there I would...>
```

Following the example above, let's look at the first line:

```
tester(Test): Ensure that you can replace words at the
beginning and/or end of a passage.
```

What this line is saying is that a certain test case named **`tester`** in the **`Test`** class failed and this was the rest is the helpful hint or note written for that particular test case.
Next, let's look at the next few lines:

```
 expected:<...sage you would like [filtered:
    Please enter the word you would like to replace:
    Please enter word you would like to insert:
    Uncensored: ]
    Hello there I would...>
```

Expected is essentially what the word means. The test case expected to see the String in between < and >
The asd **`<...`** and **`...>`** is the method's way of truncating long Strings. Meaning, there are characters before the string you see and it is not important as it is not different.
What you should pay attention to is everything in between the square brackets, namely:

```
[filtered:
    Please enter the word you would like to replace:
    Please enter word you would like to insert:
    Uncensored: ]
```

This means that there is something in here that is different. Since this is the expected output, naturally we need to see our actual output in order to see what the difference is. Conveniently it is also shown here:

```
but was:<...sage you would like [to be filtered:
    Please enter the word you would like to replace:
    Please enter word you would like to insert:
    Uncensored :]
    Hello there I would...>
```

Imagine reading this, expected x but was y. That's exactly how the error message should be interpreted.
Can you find the error that's causing this test case to fail?

`filtered:` vs **`to be filtered:`**

Please make sure you understand these error messages as reading debug messages give insight as to why your code is failing. If you still do not understand, please ask a TA for help!

Turning in Your Work

- Log into Blackboard
- Select PJ 01
- Upload your code
- Hit Submit
- Done!

Submission

Submit the following:

- **`TextFilter.java`**

Rubric:

- Part 1 - Censoring Words -- 30%
- Part 2 - Replacing Words -- 30%
- Part 3 - Censoring Personal Information -- 30%
- Part 4 - Filtering multiple times -- 10%

Good Luck