Programming Project 3

Due Sunday, March 31 at 11:59pm

The purpose of this project is to give you practice writing loops.

General Guidelines (20% of your grade will be based on using good style)

- All methods listed below must be `public` and `static`.
- You should place a comment at the top of the file that contains your name and a short description of the purpose of the class.
- You should place a *short* comment before (directly above) each method describing the method. The comment should be one sentence and describe *what* the method does, not *how* it does it. Do not simply copy the descriptions below for your comments.
- You should place a short comment directly above each field (if you have any) indicating the purpose of the variable.
- *The comments at the top of the class, above each method, and above each field are to be in JavaDoc format.* The comments above the loops and any other comments you choose to add may be in any format you feel is appropriate.
- You should place a short comment above each loop explaining how the loop works. Ideally, you should list the goal of the loop, any required precondition for the loop, and if you can, a good invariant for the loop.
- Any other complicated code such as code the has lots of if statements or variables should contain *short* comments to help the reader. The comments can either be above the code fragment or to the right, aligned in a column.
- If your code is using a loop to modify or create a string, you need to use the `StringBuilder` class from the API.
- **Important:** While it is tempting to hunt through the API to find methods of `String` and `StringBuilder` to help you, most of the methods will be too inefficient for good code. You should limit your code to use the `charAt` and `length` methods of `String` and the `append` and `toString` methods of `StringBuilder`. You may use methods from the `Character` class though they are not required.
- **Important:** You should not use the commands `break` nor `continue` in your loops.
- Remember to use good style: everything should be indented nicely, variables should have good names, there should be a blank line between each method.

To submit your project, click on the *HW 2* link, and you should find a *Submit* link. Attach your Java file and your testing report. Remember to attach the files that end in `.java`, not `.class` or `.java~`.

Part I: Programming (60% of your grade will be correctness of the following)

Definition: below you are asked to write methods dealing with *words*. For this homework, a *word* is a contiguous sequence of one or more non-whitespace characters. (A whitespace character is a space, a tab, a ``newline" character. Java has a few other whitespace special characters, but your code will not be tested on them.) So the following are all considered single *words*.

hello
hello!
fr4$,32
?

Try to keep your code as simple as possible by avoiding nested loops (loops inside loops). None of the first three problems require nested loops.

Create a class called `HW3` that contains the following methods:

1. `String capitalizeSenteces(String s)`:
   Returns a new `String` that is identical as `s` except that the first character of each sentence (if it is a letter) is capitalized. The first non-whitespace character of the `s` is the first character of the first sentence, and the first non-whitespace character that appears after a `. ? !` is the first character of a sentence. *(Note that this does not follow the word definition above.)*
   `capitalizeSentences("hello! what is my number? 341-1212! WHAT is yours?")` should return `"Hello! What is my number? 341-1212! WHAT is yours?"`
   `capitalizeSentences("look!sentence.no-spaces!")` should return `"Look!Sentence.No-spaces!"`

2. `boolean subSequence(String sequence, String string)`
   Returns `true` if `sequence` is a subsequence of the `String string`. A subsequence of a string is a sequence of characters that occur in order, but not necessarily subsequently, in the string. For example, `"abc"` is a subsequence of `"about chocolate"` but `"abc"` is not a subsequence of `"acorn bud"`.

3. `String removeExtraSpaces(String s)`
   Returns a new `String` that contains the same contents as `s`, but any spaces (including tabs) at the beginning of the string and end of the string are removed, and if more than one space (or tab) is between two non-space characters, the multiple spaces are replaced by a single space. For example, `removeExtraSpaces(" How about  that  ?  ")` should return `"How about that ?"`

4. `boolean containsWord(String s, String wordList)`
   Many programs contain some kind of search that hunts through a document for specific words, and in this method you will implement such a search. The parameter `wordList` is a `String` that will contain 0 or more "words" separated by white spaces. A white space is a space or tab, and for this method consider a word to be any sequence of 1 or more letters, numbers, or symbols.

   The method `containsWord` should return `true` if `s` contains a substring that is the same as any of the words in `wordList`. For example,
   `containsWord("My what a lovely day", "hate  love")` should return `true`,
   `containsWord("StopThat!!", "! ? *")` should return `true`,
   `containsWord("Guess who is coming to dinner?", "Who Dinner")` should return `false`, and
   `containsWord("One last example", "")` should return `false`.

5. `String wordSearch(String[] board, String wordList)`

This method is a simple version of the popular *word search* puzzle played by kids (and adults). The first parameter can hold any strings. You may assume none of the `String`s are `null`. The second parameter will contain 0 or more "words" separated by white spaces. The output should be a `String` containing each of the words of `wordList`, separated by spaces, that appears either forwards or backwards in any of the `String`s of `board`. For example:

```
String[] board = new String[5];
board[0] = "Case";
board[1] = "Western";
board[2] = "Reserve";
board[3] = "University";
board[4] = "Cleveland, OH";
```

Then `wordSearch(board, "HO CWRU west West ale stern sac")` should return `"HO West ale stern"`.

6. **Extra Credit:** `String fullWordSearch(String[] board, String wordList)`
   This function should perform the same as `wordSearch` except that instead of searching only forwards and backwards, it also searches vertically (both up and down) and each possible diagonal direction. For example:

```
String[] board = new String[5];
board[0] = "Case";
board[1] = "Western";
board[2] = "Reserve";
board[3] = "University";
board[4] = "Cleveland, OH";
```

Then `fullWordSearch(board, "CWRU lee ces lie rest dt")` should return `"CWRU lee lie dt"`.

Part 2: JUnit Test Cases and a Testing Report (20% of your project grade)

You need to write JUnit test cases for each of the methods above. Your JUnit tests cases should be thorough. Your testing report should describe why your JUnit test cases sufficiently tested your code. Your testing report *should not* give the results of the different tests. The results will be seen when the graders run your JUnit tests. If you are unable to complete a method above, you should still write JUnit tests that would test the method had it been completed.

Hints for testing loops. Your tests need to, at the minimum cover the following cases:

1. **Test 0, test 1, test many:** This means you have to test cases where the parameters, if they are integers, are 0, 1 or some value other than 1. If the parameters are strings, you have to test strings of length 0, 1, and more than 1. If the strings must contain certain data, you need to test cases where they contain 0, 1, and more than 1 of the desired data.
2. **Test first, last, and middle:** In cases where you have to search in or modify a string, you need to test cases where the item to be found or modified is the first character of the string, the last character of the string, or somewhere in the middle of the string.

In summary, your JUnit tests should cover each of the above cases for each of the methods you complete in the

homework. Your testing report should state what *test 0, test 1, test many* and *test first, last, and middle* mean for each of the homework methods and how your JUnit tests covered these cases.