Assignment 3: Basic Java coding and JUnit

This assignment assesses your basic knowledge of Java and JUnit, which you will need for future assignments and projects.

To complete the assignment you must complete the following tasks:

- Clone your individual GitHub repository in your local workspace. This is the same repository that you used for the previous assignment and will use for all future individual assignments: `https://github.gatech.edu/gt-omscs-se-2019fall/6300Fall19<GT username>.git`
- Download the archive assignment3.tar.gz
- Extract the archive in the root directory of the repository, which should create a directory called `Assignment3` and several subdirectories. Hereafter, we will refer to the directory `Assignment3` in your local repo as `<dir>`.
  (If you do not know how to extract a "tar.gz" archive, you should be able to find plenty of resources on how to do that online--ask on Piazza otherwise.)
- Directory `<dir>/src` contains, in a suitable directory, Java interface `edu.gatech.seclass.MyCustomStringInterface`. It also contains exception `edu.gatech.seclass.MyIndexOutOfBoundsException`, which is used by the interface.
- Your **first task** is to develop a Java class called `MyCustomString` that suitably **implements** the `MyCustomStringInterface` that we provided. (The semantics of the methods in the interface should be obvious from their name, the JavaDoc comments in the code, **and the test examples in class `MyCustomStringTest`**. If not, please ask on Piazza.) Class `MyCustomString` should be in the same package as the interface and should also be saved under `<dir>/src/edu/gatech/seclass`.
- Your **second task** is to develop a set of **JUnit 4** test cases for class `MyCustomString`.
  - You should create test cases for each method that implements a method in the interface, except for getters and setters (i.e., the first two methods). **Make sure that every test method has a suitable oracle** (i.e., either an assertion or an expected exception) and that the tests are not trivial (i.e., have a specific and unique purpose). In other words, each test should (1) test a specific piece of functionality and (2) check that such piece of functionality behaves as expected.
  - In addition, for each exception that can be thrown by a method M and is explicitly mentioned in M's documentation, there should be at least one test for M that results in that expected exception. For example, at least two of the tests for method `filterLetters` should result in expected exceptions:
    - One for `IllegalArgumentException`
    - One for `MyIndexOutOfBoundsException`
  - **When testing for an expected exception, make sure to use the @Test(expected = <exception class>) notation.** For example:

    ```
    @Test(expected = NullPointerException.class)
    public char mostCommonChar () {
      ...
    }
    ```

    Note that, as an example and to help you make sure that your code suitably handles exceptions, we already provide in class `MyCustomStringTest` (discussed next) one such test for one of the methods.
  - To make your job a little easier, the archive also contains, in directory `<dir>/test`, a test class `edu.gatech.seclass.MyCustomStringTest`, which provides a skeleton for the test cases that you need to implement and a complete implementation for a few of them. Your job is to write the body of all of the test cases that are not implemented (i.e., the ones that simply fail with a "Not yet implemented" message). Make also sure that the test cases you write are not just a trivial variation of the ones we provide. To this end, make sure to add a concise comment before each test that you implement to clarify its rationale (e.g., "//This test checks whether method

`mostCommonChar` suitably throws a `NullPointerException` if the current string is null"). This comment should provide a **unique purpose** for the test, and not simply list its input. Finally, creating more than the required test cases will not be rewarded nor penalized; in other words, feel free to write more tests if you are so inclined, but **you will have to provide at least as many tests as there are skeletons** and we will grade the tests in the provided skeletons.

- Submit your solution by:
  - Pushing `<dir>` and all the files and directories underneath it (except for the generated files excluded by .gitignore, if you have one, such as all class files) to the individual remote GitHub repository we assigned to you. **If you use any external libraries other than junit 4, they must also be pushed to your repository.** You can decide whether to commit IDEA's project related files, as this does not make a difference for us, but the repo **must necessarily** contain the two files:
    - `Assignment3/src/edu/gatech/seclass/MyCustomString.java`
    - `Assignment3/test/edu/gatech/seclass/MyCustomStringTest.java`
  - Submitting **only the final commit ID** for your submission on Canvas. You can get your commit ID by running "`git log -1`" or by viewing the commit ID in your assigned repository on github.gatech.edu. Viewing the commit on the site will also verify that you have pushed the files you intended.

Notes:

- **You cannot modify the provided interface (`MyCustomStringInterface`), nor the already provided test cases (except for those that you are supposed to implement, obviously, whose body is simply "`fail("Not yet implemented")`"), nor the names of the test cases, nor the test class name, nor the declaration of mycustomstring in the test class.** (We understand that more meaningful names could be used for the test cases, which is in general advisable, but having numbered tests considerably helps our grading.)
- **You should use Java version 11 or 12 to solve the assignment.**
- You may assume that we have a junit 4 library, as well as the associated hamcrest library. **Any other external library used must be included in your repository.** Internal java libraries are the packages as documented in the Java Platform Standard Edition API Specification. Do not assume that all libraries available on your computer are standard libraries.
- With the term "*digits*", we mean characters '0' through '9'. "*Letters*" refers to letters in the English alphabet, 'a' through 'z' and 'A' through 'Z'.
- We will also run your code against our set of test cases to make sure that you implemented the functionality of the required methods correctly. Note that our test cases are fairly simple and are not trying to exercise arcane corner cases, so as long as you implement the interface as described, you will be fine.
- Although it is not mandatory, we recommend that you use IntelliJ IDEA to complete the assignment, so that you can also get familiar with this IDE (in case you aren't already). To do so, you should open IDEA and do the following (notice the initial, alternative steps):
  - **If you are in the initial IntelliJ IDEA screen (see image below), do the following:**
    - Click "Import Project"
    - Select <dir> as the directory to import
  - **Else, if you are already in a project window, do the following**
    - Click menu "File" => "New" => "Project from Existing Sources…"
    - Select `<dir>` as the project directory
    - Select "Create project from existing sources"
  - Click "Next" accepting the default choices (just make sure that either Java SDK 1.7 or 1.8 (recommended) are selected) and then "Finish"

(The following instructions add JUnit to your project)

- On the project window that opens after clicking "Finish", right-click on your project name and choose "Open Module Settings"

    - Click on "Libraries" (left pane, under “Project Settings”
    - Click the plus (“+”) sign at the top of the pane that appears to the right of “Libraries” and choose "From Maven...".
    - Search for "junit" (click the magnifying glass to get the search started)
    - When the search is done, select “junit:junit:4.*” in the drop-down menu under the sea search term (where “*” indicates any number, such as “junit:junit:4.12”).
    - Make sure that the box “Transitive dependencies” is selected
    - Click “OK” until you get back to the original screen

- Please note that using an IDE should also make it easier to create skeleton code for the class that implements the interface, create and run the JUnit test cases, and so on.
- Before submitting, make sure to compile and **run your test cases** as a group (not only individually) and to check that they all pass--something else that you can do at the push of a button within most IDEs.
- **Verify that your final commit ID contains what you intend, and is pushed to the repository, by cloning it in a separate location or viewing it on github.gatech.edu.**
- You can perform multiple commits as you produce your solution. This is not only fine, but actually very much encouraged. You should only submit the final ID that you want graded.
- We also encourage you to use a “development” branch and merge your stable version(s) into the “master” branch. If interested, see this site for an example of a possible branching model of this kind. (There are many others.)
- From this point on in the course, you should **NOT** run the git repository reset commands given in Assignment 2, any version of *force --push*, or any commands deleting branches, tags, or prior commits on your repository.