COSC222 LAB 2 - Unit Testing with JUnit

Unit testing is an important part of quality assurance. The purpose of unit tests is to confirm that a class/method does what you claim it does and does not break under certain circumstances (i.e. wrong input, corner cases etc.). It essentially provides a way to detect and track logical errors.

In this lab you will learn how to create a test cases for yourself and how to execute these tests in the Eclipse IDE. Create a Java project in your IDE, right-click it in the Package Explorer and choose

`Build Path > Add Libraries… > JUnit > Next > JUnit5`

To check if it worked, right-click your project and select **`New > JUnit Test Case.`** Then attempt to run the file using "Run as -> JUnit Test". If this causes error, try making a new project, add a new JUnit test file, and Eclipse will offer to adjust your Run Configurations for you to add JUnit. If again this fails for you, you may attempt to use JUnit4 or JUnit3 to see if an earlier version is more compatible with your current setup.

All your files should contain your name and student ID number in a comment at the top of the file.

You are given 3 files: **`BadFunctions.java, BadSet.java, TestBadFiles.java.`**

1) Begin by looking at the class **`BadFunctions.java`**. This class contains some static methods which are all flawed in some way. Also look at **`BadSet.java.`** This attempts to provide a generic implementation of a Set data type, but it is flawed. Recall that a Set is a collection which contains no repeated elements. So, for example, adding 3,3,8,8 to a set will make the set of size two equivalent to {3, 8}. Removing 3 from the set should remove 3 from {3, 8}, no matter how many times a user has added 3 to the set.

2) Next, look at **`TestBadFiles.java`**. This is a JUnit test file. The annotation @Test before a method header signifies that the method is a Testcase and will create a checkmark or an 'x' in the Test window after running the test file. JUnit testfiles can contain other, un-annotated static methods. There are many other JUnit annotations that you will see in future labs.

   `@Before` methods are done before tests. This is useful for initializing variables and objects that will be used in the test cases to ensure they are created fresh before each test runs. (Not used in this lab)

   `@Test` methods are where you test the functionality of your classes/methods. This example has one test per method but this does not always have to be the case.

A test is an `Assert` statement. `Asserts` are used to validate facts. There are several types of assert statements, and here you will see `AssertEquals, AssertTrue, AssertFalse, AssertArrayEquals. AssertEquals(correct, actual)` is used to test that the "actual" computed result matches the "correct" value we expect. For example, in testing the average(x,y) function, the correct answer we expect from average(3,9) is 6. So to test if average(x,y) passes this test, we can create an assert statement `AssertEquals(6, average(3,9))`. For us to write a proper assert statement, we need to know what result we expect from a particular input. Similarly, `AssertTrue` and `AssertFalse` check if an argument evaluates to True or False, respectively. So if isPalindrome(s) is supposed to return 'true' when s is a palindrome and false otherwise, we can create tests `AssertTrue(isPalindrome("otto"))` and `AssertFalse(isPalindrome("abcdefg"))`. There are many different type of assert statements. See the documentation here:

http://junit.sourceforge.net/javadoc/org/junit/Assert.html

Here are some additional reference material for creating tests:
http://www.tutorialspoint.com/junit/
https://dzone.com/articles/junit-tutorial-beginners

3) Your task is to write tests in the testfile that will expose errors in the BadFunctions and BadSet file. There are //TODO items in the testfile for you to follow. As the file is currently given, the BadFunctions and BadSet files appear to be tested and appear to pass all the tests, but there are cases on which these methods will fail, and it is your job to write tests that will expose the faults of these methods. You may write multiple new tests for a function, but you must find at least one that exposes an error.
4) The main goal of this assignment is for you to learn how to set up JUnit and write unit test cases, so the majority of points are devoted to the testing. But to get full marks on this assignment, also create a copy of BadFunctions.java called GoodFunctions.java, and correct every flawed function. Also make a copy of BadSet.java and call it GoodSet.java and make it work correctly. All the above corrections can be done by changing or adding a single instruction in each method. Don't forget to add your name and student ID at the top of your new files.
5) Finally, create a copy of the TestBadFiles.java and call it TestGoodFiles.java. Change all the tests so that they are testing GoodFunctions.java and GoodSet.java instead of BadFunctions.java and BadSet.java. (These changes can be made quickly with your IDE's Find/Replace).

Submit all the .java files you edited. Submit the files as .java files – not in a compressed folder or zip or archive file. You do not need to submit BadFunctions.java and BadSet.java as you do not need to edit these files, but you may submit them if you choose (just remember to still include your name and ID at the top of the files!)

Grading:
+1 for including your name and ID in all your files and submitting only .java files
+6 for completing the test cases in TestBadFiles.java. This requires correctly finding at least one testcase that will cause each BadFunction fail, and at least one example that shows that BadSet fails.
+1 for a corrected GoodFunctions.java file
+1 for a corrected GoodSet.java file
+1 for TestGoodFiles.java correctly testing and passing on GoodFunctions and GoodSet