JUnit

Lab 4

Wednesday, September 27, 2017

CSC 062: Fall, 2017

This is an important lab! Today, we're going to be learning about unit testing. A unit is the smallest testable part of a program. Often, a unit corresponds to the individual methods of a program. A unit test, then, is an automated piece of code that tests a single assumption regarding the correctness of a unit (e.g., method). Read this definition again and again until you understand what a unit test is.

Unit tests fit nicely with our discussion of pre- and post-conditions. Recall the following method from the `FreqList` class from assignment 2:

```
public String get(double p)
```

This method assumes that $0 \leq p \leq 1$. If this is not the case, the method should throw an `IllegalArgumentException`. We can write a unit test that passes in an invalid value for p (e.g., $p = -1$) and then checks that such an exception is thrown. In the future, if we change the implementation of the get method, we can run all of the associated unit tests to make sure our changes did not affect the expected behavior of the method.

As another example, consider the `add` method. This method takes a string and adds it to the frequency list. We can write a unit test that calls the `add` method and then checks that the string has, indeed, been added to the frequency list. (How could you verify that the string has indeed been added?). Again, if we change the implementation of the `add` method in the future, we can run all of the associated unit tests to make sure our changes did not affect the expected behavior of the method.

`JUnit` is a modern testing tool for creating unit tests designed by Eric Gamma and Kent Beck (promoter of "Extreme Programming") that is available from `www.junit.org`. `JUnit` is a tool that reflects the philosophy of test-driven development. Test-driven development involves writing unit tests **first**, and then writing the actual code **second**.

Unit testing is easier and more effective (at least in the early stages of programming) than writing tests of the entire project. Ultimately, the goal is that you would never integrate a class into a project until you are convinced that it works properly. Moreover, because we often modify classes, we keep the unit test so that we can rerun them after any changes are made. While anyone can do such testing, `JUnit` is a tool that helps you set up the tests and then runs them all automatically for you – a huge advantage!

Before coming to lab, it is important that you read through this lab writeup as well as reading the following links:

1. The `JUnit cookbook` linked off of the `Documentation and Handouts` page
2. The `Getting Started` page at `www.junit.org`
3. `The Exception Testing` page at `www.junit.org`
4. Skim over the types of assertions you can use in the `JUnit JavaDocs`

Creating Unit Tests in `JUnit`

`JUnit` is integrated into Eclipse, so it is especially easy to use it when using Eclipse. Here are general directions to create a unit test for an existing project. Read them through quickly to get a sense as to what is happening. We'll use these instructions for an in-lab demo that creates a unit test for your Word Frequency program.

1. In the "Package Explorer", click on the folder for "Assignment02"
2. Create a class extending `JUnit.framework.TestCase`. The best way to do this is to select "New" and then "JUnit Test Case."
   (a) When the "New JUnit Test Case" window pops up, select "New JUnit 4 test" at the top.
   (b) All of the fields should already be filled in. If not, type a name for the test class in the "Name" field. Type "wordsGeneric.FreqList" in the "Class under test" field. Click the "Next" button.

(c) You will get a window showing the methods of the class you will be testing (if you selected one). Click in the checkboxes to have method headers automatically generated for each selected method. Click on "Finish" and your class will be displayed. (The system may inform you that JUnit 4 is not on the build path. If so, select "Add JUnit 4 library to the build path" and select OK.)

3. The generated class will import `org.junit.Assert.*` and `org.junit.Test`. Also all of the methods that have stubs created will have an annotation `@Test`. The names of all of these methods begin with test, take no parameters and return `void`. They all start with a default body `fail(''Not yet implemented'')`. If you wish to add your own test methods make sure that they follow the same template.

4. You may need some instance variables to be declared and initialized to run your tests. Declare the instance variables as usual, and initialize them in a method with header public void `setUp()` (which may have been added by the wizard). That method should start with the annotation `@Before`. (If the compiler objects to that annotation, add `import org.junit.Before;`.) If you need to put away resources (e.g., close files) after a test, add a method with no parameters (typically called cleanup) that has an `@After` tag before its declaration.

5. Write the body for each method. The tests should use methods like `assertEquals` and `assertTrue`. The `setUp` method will be run before each test.

Running JUnit

To run a test case:

1. Select the test class in the Package Explorer pane on the left side of the Eclipse window.

2. In the toolbar at the top of the window, there is a green circle with a white triangle that controls program execution. Pull down the menu to the right of this green circle and select "Run As" and then select "JUnit test."

A `JUnit` panel should replace the PackageExplorer on the leftmost panel in the window. If the bar across the top is green, then all tests have succeeded. If one or more tests have failed then the bar will be red and each failed test will be listed. Select each to find out why the test failed.

The `JUnit` panel on the left side of the screen can be replaced by the usual Package Explorer panel if you just click on that tab on the top of the panel.

Continue to refine and add to your tests (just add a method whose name starts with test and code until you are convinced that all of the methods in your class work correctly).

The recommended way of working with `JUnit` is always to write your tests before writing the code that uses them. Thus when you start, all of your tests should fail. Your goal is to fix the failed tests one at a time until you get a green bar. If you had a good test suite then your class should be correct. (Though if you had a lousy test suit, you may still have errors!)

Today's Assignment

In today's lab, you will add unit tests to the test class for the `FreqList` class. Before you begin, go back to your `FreqList` class and change the instance variables to be `protected` instead of `private`. (Why is this important?) Your test class must contain the following methods:

1. A `setUp` method that creates a `FreqList` object

2. A method to test that the `add` method correctly inserts a string

3. A method that passes in $p < 0$ to the `get` method

4. A method that passes in $p > 1$ to the `get` method

5. A method that passes in correct values for `p`. For a given value of `p`, you should check that the correct string is returned

6. You can add more tests as you see fit. Note that your test class can only access the `public` or `protected` methods and instance variables of the `FreqList` class. If you are working with someone else, you may use either of your programs.

Submission instructions

Follow the submission instructions for lab/assignment 1. Remember to fill out the lab04.json file (you can rename a previous .json file or use the blank one in the /common/cs/cs062/labs/lab04 folder):

Add your CS username to the "collaborators" list.

Add your partner's username as well.

If you have anything you want to say to the graders, put that in the "notes" field.