CIS 121 Tutorial

JUnit Testing

Learning how to test your Java code on Eclipse

Edited by *Ziad Ben Hadj-Alouane*

Unit Testing is widely used in the industry, so it is important you take it seriously and get used to it from now on. It will help you understand your code better and generally be a better programmer. Since you will be using Unit testing a lot in this course (and most likely in other courses), we thought that defining what a good testing structure looks like is important.

In-depth video tutorial

If you want an in-depth explanation of JUnit Testing, watch the Tutorial video we made right here. If you want to have somewhere you can come back to quickly to check the guidelines, refer to the details below

General guidelines

- We emphasize on the concept of **bottom up testing**. This means if a method **X uses a method Y**, test method **Y before X.** We want to see that order in your test suites.
- We expect you to **test each method individually**. We also expect you to isolate cases individually. This means a test for each method and for each case you can think of.
- We expect you to **cover as many edge cases as you can**. Think of null elements, of singletons, of empty data structures, of complicated cases... really anything that might cause problems to your implemented methods. It is important that your code is bug-free!

Alternative Guide

1. **Setting up JUnit in Eclipse:**
    a. Before testing, you need to make sure you included the JUnit library in your project. To do so, right click on your project name → **Build Path** → **Add Libraries...** → **JUnit** → **Next**. Select JUnit 4. Click Finish. You're now ready to use the library!
2. **Defining your test class:**
    a. Create a Unit Test class by clicking on your package → **New** → **JUnit Test Case**
    b. If you are testing a class named *Example*, make sure to name your test class *ExampleTest*. This helps you refer to your test suite quickly and is good code practice
    c. At the top of your class (outside of it), import the following: **org.junit.Assert.***, **org.junit.Test,** and **org.junit.Before**
    d. Make sure to specify that the class you are testing belongs to your homework package by declaring it at the top: **package NameOfYourPackage.** Alternatively,

you could import the specific class you are testing so you can create instance objects.

e. Inside your class, create private fields that you intend to test (i.e deques, schedulers...)

f. Inside your class, make package private helper methods (package private means not public, not private, no modifier at all) that you think might help you (i.e printing out results, adding elements to a list if you need to, or any set of operations you think you will be repeating a lot in your test suite)

3. **Initializing your Data Structures/Test subjects:**

   a. Before making any tests, you want to think about the data structures/objects common to your test and initialize them in a *@Before* method, which will run automatically before each test case. This makes your testing less redundant and will earn you style points. For example, if you are testing a class called *Foo*, think about making the following method:

```
//This is an initialization that will affect all Unit tests.
//You add a@Before attribute before the test definition
@Before

public void initialize() {
        //Put test code here
        Foo fooObj = new Foo();
}
```

4. **Making test methods & running tests:**

   a. There are two types of test methods you should focus on:

```
//This is a normal test that tests whether you got an expected result out an
//operation. Most of your tests will have this format.
//You add a @Test attribute before the test definition
@Test
public void myNormalTest() {
        //Put test code here

}
```

```
//This is an exception testing function.
//You add the @Test (expected = SomeExpectedException.class) attribute
//and you replace SomeExpectedException with the exception you expect will be
//thrown inside your test
//i.e IllegalArgumentException, IndexOutOfBoundsException....

@Test(expected = SomeExpectedException.class)

public void myExceptionTest() {
```

```
        //Put test code here
}
```

b. Name each test in a clever way. This means you should include the name of the method you are testing in the name of test (example: **pollFirstTest**). In addition, make sure to include a hint on what case you are testing (example: **pollFirstSingletonTest**, **pollFirstNullTest**...)
c. It is generally good practice to include a short comment on what feature/method you are testing (i.e: **//Tests adding null element to linked list**)
d. Simulate the expected behavior of the test you are running. Create variables that hold the expected output, then use the assert methods to see if the outputs matches or not (meaning your test is successful or not):
    i. Use **assertEquals(expected, actual)** to test if the values of actual are the same as the values of expected. This test outputs True or False depending on the result. This tests Object equality, so make sure to cast your values to Objects for this to work (i.e avoid primitive types)
    ii. Use **assertTrue(expected)** to test if the boolean variable expected is True or not
    iii. Use **assertFalse(expected)** to test if the boolean variable expected is False or not
    iv. Use **assertArrayEquals(expectedArray, actualArray)** to test if two arrays are equal
e. Finally, run your tests. You can see the results of the tests you ran on the JUnit widget that appears on your screen. Green means pass, blue means your assert statements did not get results that matched your expectations, red means something went wrong (compilation errors)

Here is an example unit test. Remember that you have to have a @Before method that will be called automatically before all tests!

```
//Put all initializations here
@Before
public void initialize() {
        //start with empty deque for all tests
        d = new ResizingDeque<Integer>();
}


//Testing polling first such that the polled element is null
@Test
public void pollFirstNullTest(){
        d.offerLast(null); //add null (I tested offerLast() previously!)
        assertEquals(d.pollFirst(), null); //Expect polled element to be null
}
```