2121 Laboratory 1

Unit Tests

Deveveloped by Jonathan Redmann and Brian Walters

Introduction

In this lab we will explore unit testing. The purpose of writing test classes is to automate the process of ensuring the validity of our code. In our test classes, we create instances of the class we want to test, manipulate these objects using their public methods, and programmatically determine whether the results meet our expectations. We can think of testing as a replacement for designing a TUI or GUI and tediously testing every possible manipulation of an object. They also provide a simple way of ensuring that any changes we make to our code continue to produce valid results.

Creating Tests

To write tests, we must know the desired outcome of calling a method. We must know the values that our queries should return and we must know how commands change the state of our objects. Broadly, there are three broad ways in which we come about this knowledge.

With white box testing, we have access to the code we are testing. We can use the code to determine what the expected outcomes should be. White box testing assumes that the code we are provided is correct. We often use white box testing when we expecting a need to change code and want to ensure that same behavior is acquired after the changes.

In the opposite end of the spectrum, there is black box testing. With black box testing, we rely on the documentation to determine how a method should behave and write our tests accordingly. This is a key feature in the software development paradigm called test driven development. In test driven development, we write our tests before we write any code. To write our tests, we rely only on the software documentation. As we write our code, then we can then run it against the tests to ensure that it is behaving as expected.

Normally, our testing process is somewhere in between (sometimes called gray box testing). Having access to the code we are testing helps us anticipate edge cases. However, relying too heavily on the code could bias our tests towards merely "passing" as opposed to achieving the expected functionality. It is good practice to first think generally about the problem our objects are modeling and scratchpad our test cases before we look at the code.

JUnit

In this lab, we will use a framework called JUnit to provide us with the tools we need to write unit tests for our code. Unit testing is a particular kind of automated testing in which we focus on testing discrete units of code. Usually this involves designing tests for each individual method of a class, but it can also involve testing sets of methods whose functionality is tightly coupled.

JUnit is not included with the Java SDK and may need to be downloaded separately from: http://junit.org/. For the purposes of our labs, we will use JUnit's 3.x style of testing.

- It is conventional to name the test class based on the class being tested. For instance, if we are testing the class Circle, then we name our test class CircleTest.

- Test classes extend the junit.framework.TestCase class.
- The test class should contain a set of instance variables called test fixtures that will be used to testing.
- JUnit tests use a setUp() method instead of a constructor to initialize the test fixtures. The setUp() method is *automatically called before each test method*. This allows each method to have a fresh set of objects. A tearDown() method may also be used to clean up object state after each test has been run. This may be useful for tests that require the opening and closing of files, etc.
- A JUnit test method is usually named based on the method being tested. If, for instance, we are testing the add() method, then our test method would be called testAdd().
- JUnit tests use assertions to determine whether the tested unit has passed all of its tests. Some common asserts are assertTrue(), assertFalse(), assertEquals(), assertNull(), assertNotNull(). See http://junit.sourceforge.net/javadoc/org/junit/Assert.html for a complete list.
- Note that you can only test public methods and you can only *directly* test methods that return a value.
- Methods that return Strings – especially toString() – can be difficult to test because of non-printing characters. You should still try to write tests for these methods. Just because careful with formatting and newlines!

Using JUnit at the Command Line

For JUnit 3.x style testing, we are provided with a runner class called TestRunner in the junit.textui package. TestRunner takes as a parameter the name the compiled test class we want to run. To be clear, we do not run the test class directly. Instead we are invoking the TestRunner class, which takes the name of the test class as a parameter. Be certain to include the locations of both files in the classpath switch. For instance:

java -cp /usr/share/java/junit.jar junit.textui.TestRunner [test class name]

Exercise One Testing Class String

Let's create a test class together. In this exercise we will write tests for a few methods belonging to Java's String API (http://docs.oracle.com/javase/7/docs/api/java/lang/String.html). As we begin, think about the "edge" cases that need to be considered for strings.

1. Create a StringTest class that extends TestCase. Be sure to include the TestCase class from the junit.framework package.

```
import junit.framework.TestCase;
/**
 * A test class for a few of the methods for Java's String class.
 */
```

```
public class TestString extends TestCase {}
```

2. Create stubs for the methods we need to override from TestCase. In this case, we only need a setUp() method. We can ignore the tearDown() method.

```
/**
 * Setup
 */
protected void setUp () {}
```

3. Create stubs for String's methods we want to test. In this case, we will test the methods length(), equals(), and toLowerCase().

```
/**
 * Test String class's length() method.
 */
public void testLength() {}

/**
 * Tests String class's equals() method.
 */
public void testEquals() {}

/**
 * Tests String class's toLowerCase() method.
 */
public void testToLowerCase() {}
```

4. Create some test fixtures (instance variables) for our StringTest class. For our example, let's make five.

```
// The Strings to test.
private String letters;
private String numbers;
private String empty;
private String simpleSentence;
private String complexSentence;
```

5. Initialize our test fixtures in the setUp() method. Notice the cases we have covered. We considered standard strings, but we also considered some edge cases. Does the class handle numbers properly? Does the class handle the empty string? Can it handle punctuation? What about non-printed characters like the newline character or the tab character?

```
    letters = new String("abcdefg");
    numbers = new String("1234567");
    empty = new String("");
```

```
    simpleSentence = new String("This is a sentence.\n");
    complexSentence = new String("%Punctuation!, Can cr\te8 <problems.$");
```

6. Now we can write our tests.

```
  /**
   * Test String class's length() method.
   */
  public void testLength() {
   assertTrue( letters.length() == 7 );
   assertTrue( numbers.length() == 7 );
   assertTrue( empty.length() == 0 );
   assertTrue( simpleSentence.length() == 20 );
   assertTrue( complexSentence.length() == 36 );
  }

  /**
   * Tests String class's equals() method.
   */
  public void testEquals() {
   assertTrue(letters.equals("abcdefg"));
   assertTrue(numbers.equals("1234567"));
   assertTrue(empty.equals(""));
   assertTrue(simpleSentence.equals("This is a sentence.\n"));
   assertTrue(complexSentence.equals("%Punctuation!, Can cr\te8 <problems.$"));
  }

  /**
   * Tests String class's toLowerCase() method.
   */
  public void testToLowerCase() {
   assertTrue(letters.toLowerCase().equals("abcdefg"));
   assertTrue(numbers.toLowerCase().equals("1234567"));
   assertTrue(empty.toLowerCase().equals(""));
   assertTrue(simpleSentence.toLowerCase().equals("this is a sentence.\n"));
   assertTrue(complexSentence.toLowerCase().equals("%punctuation!, can cr\te8
<problems.$"));
  }
```

Note: While we used assertTrue() in these tests, there are a number of different methods available. For instance, instead of using:
assertTrue(letters.equals("abcdefg"));

we could have used assertEquals(), such as:
assertEquals(letters, "abcdefg");

Some other common assertion are assertFalse(), assertNull(), assertNotNull(). See http://junit.sourceforge.net/javadoc/org/junit/Assert.html for a list of methods available.

7. Compile and run your test class.

Exercise Two Testing Class Rectangle

Now let's write a test class for Java's Rectangle class (http://docs.oracle.com/javase/7/docs/api/java/awt/Rectangle.html). It's important to test constructors and commands – methods that do not return a value. To test these units, we will need to rely on the query methods to inspect the object's state. For our purposes, let's assume we have written tests for the queries getX(), getY(), getHeight(), and getWidth(). Now let's write tests for Rectangle's default constructor and the constructor that takes four parameters. Let's also write a test for the contains() method that takes two parameters.

1. Identify some test fixures that we will use to test the contains() method. Think about any edge cases that will need to be tested.

2. Create a RectangleTest class that extends junit.framework.TestCase.

3. Write a setUp() method for the fixtures you have chosen.

4. Write a test for the default constructor. Since we are testing the constructor, we will not want to use the fixtures instantiated with the setUp() method. You may, however, use the instance variable names and reinstantiate the object. Use the basic queries to ensure that the object is instantiated correctly.

5. Write a test for the other constructor. Consider any edge cases that this constructor might receive.

6. Write a test for the contains() method. What general cases need to be tested? Use comments to explain in natural language where your tests are covering these cases.

7. Compile and run your test class.

Exercise Three Testing Class Fraction

Now it's your turn. In CSCI 1581, you wrote a Fraction class. Now you need to write a test class for it. Use the above discussion as your guide. You're on your own!

Summary

When you are finished, push your local respository to your GitLabs repository. Your lab1 folder should contain:

- StringTest.java and StringTest.class
- RectangleTest.java and RectangleTest.class
- Fraction.java, Fraction.class, FractionTest.java, and FractionTest.class

Please be certain that your class files are the final product. If you make last minute changes, recompile!