JUnit Testing Utility Tutorial

Ashley J.S Mills

<ug55axm@cs.bham.ac.uk>

Table of Contents

1. Introduction

JUnit is a framework for implementing testing in Java. It provides a simple way to explicitly test specific areas of a Java program, it is extensible and can be employed to test a hierarchy of program code either singularly or as multiple units.

Why use a testing framework? Using a testing framework is beneficial because it forces you to explicitly declare the expected results of specific program execution routes. When debugging it is possible to write a test which expresses the result you are trying to achieve and then debug until the test comes out positive. By having a set of tests that test all the core components of the project it is possible to modify specific areas of the project and immediately see the effect the modifications have on the other areas by the results of the test, hence, side-effects can be quickly realized.

JUnit promotes the idea of first testing then coding, in that it is possible to setup test data for a unit which defines what the expected output is and then code until the tests pass. It is believed by some that this practice of "test a little, code a little, test a little, code a little..." increases programmer productivity and stability of program code whilst reducing programmer stress and the time spent debugging. It also integrates with Ant http://jakarta.apache.org/ant/.

2. Installation

It is assumed that you know how to setup environment variables and install software on your operating system, for a comprehensive guide to doing so for Windows see *Configuring A Windows Working Environment* [../winenvars/winenvarshome.html], for Unix see *Configuring A Unix Working Environment* [../unixenvars/unixenvarshome.html] Follow these instructions to install JUnit:

1. Download the latest version of JUnit from http://download.sourceforge.net/junit/
2. Uncompress the `zip` to a directory of your choice.
3. Add `/path/to/whereyouextractedjunit/junit/junit.jar` to your Java `CLASSPATH` environment variable.

3. JUnit Basics

This section will begin with a simple example which will illustrate the basic concepts involved in testing with JUnit.

Example 1. Simple Example

```
import junit.framework.*; ❶
public class SimpleTest extends TestCase { ❷
   public SimpleTest(String name) { ❸
      super(name);
   }
   public void testSimpleTest() {❹
```

```
      int answer = 2;
      assertEquals((1+1), answer); ❺
   }
}
```

❶

```
import junit.framework.*;
```

Since we are using some constructs created by the makers of JUnit we must import any of the classes we desire to use, most of these reside in the framework subdirectory, hence the import statement.

❷

```
public class SimpleTest extends TestCase {
```

Our simple class needs to define its own test method(s) to actually be of any use so it extends TestCase which provides us with the ability to define our own test methods.

❸

```
public SimpleTest(String name) {
   super(name);
}
```

Every test is given a name so that when viewing the output results of the overall test we can see which test an error was generated if the test fails. The constructor provides this functionality by passing the parameter passed upon construction to the parent class.

❹

```
public void testSimpleTest() {
   int answer = 2;
   assertEquals((1+1), answer);
}
```

This is one of the tests included within this TestCase, at the moment it is the only test but you are allowed to add as many as you like. You can see that we can define variables and perform arithmetic like any other Java program.

❺

```
public void testSimpleTest() {
   int answer = 2;
   assertEquals((1+1), answer);
}
```

This simple test tests whether 1+1 is equal to the value of *answer*. This is just an extremely simple example, more usually one may want to test the result of some function, perhaps a function that is supposed to strip all the 'a' characters from a string, you might test this like this:

```
public void testStringStripFunction() {
   String expected = "bb"
   StringStripper stripper = new StringStripper();
   assertEquals(expected, stripper.stringStrip("aabaaaba"));
}
```

Which would cause assertEquals to fail if the *stringStrip* method did not perform as expected, one could then try to fix the code and run the test again until it passed.

Copy the example into your favorite text editor and save it as `SimpleTest.java`. Now we want to try this test out, there are three possible user interfaces for the JUnit TestRunner:

- *TextUI*: Provides text-based output to stdout.
- *AwtUI*: Provides GUI-based output using the AWT (Abstract Window Toolkit) from Java.
- *SwingUI*: Provides GUI-based output using the Swing graphical user interface component kit from Java.

To select an interface execute the following command sequence.

java junit.*USERINTERFACE*.TextRunner *classfile*

For example if we wanted text based output for our TestCase we would issue the command sequence:

java junit.textui.TestRunner SimpleTest

Compile the file and then try this, you should see some output similar to this:

```
.
Time: 0

OK (1 tests)
```

Indicating that our quite obviously correct test has been passed, well more correctly that no failures have occurred. Change the value of *answer* in the example to something other than 2, recompile the file and run the test again, you should see output similar to this.

```
.F
Time: 0.01
There was 1 failure:
1) testSimpleText(SimpleTest)junit.framework.AssertionFailedError: expected:<2> but was:<3>
        at SimpleTest.testSimpleTest(SimpleTest.java:9)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)

FAILURES!!!
Tests run: 1, Failure: 1, Errors: 0
```

If this was a real program we would now go and try and fix the error and then recompile and retest and so on until we eventually fixed the error. You should now begin to have an idea of how to go about setting up a simple test.

4. Intermediate Example

The last section was just a quick taster of and a gentle introduction to JUnit, this section will explore a more advanced example.

```
// A Class that adds up a string based on the ASCII values of its
// characters and then returns the binary representation of the sum.
public class BinString {
  public  BinString() {}

   public String convert(String s) {
      return binarise(sum(s));
   }

   public int sum(String s) {
      if(s=="") return 0;
      if(s.length()==1) return ((int)(s.charAt(0)));
      return ((int)(s.charAt(0)))+sum(s.substring(1));
   }

   public String binarise(int x) {
      if(x==0) return "";
      if(x%2==1) return "1"+binarise(x/2);
      return "0"+binarise(x/2);
   }
}
```

Copy the above code into your favorite text editor and save as `BinString.java`, all this program does is convert an ASCII string to the sum of it's characters represented as a Binary string, the reason I chose those type of functions were arbitrary but I wanted to illustrate the testing of a program with auxiliary methods. Copy the test code below into your favorite editor and save as `BinStringTest.java`.

```
import junit.framework.*;
public class BinStringTest extends TestCase {
   private BinString binString;

   public BinStringTest(String name) {
      super(name);
   }

   protected void setUp() { ❶
```

```
      binString = new BinString();
   }

   public void testSumFunction() { ❷
      int expected = 0;
      assertEquals(expected, binString.sum(""));
      expected = 100;
      assertEquals(expected, binString.sum("d"));
      expected = 265;
      assertEquals(expected, binString.sum("Add"));
   }

   public void testBinariseFunction() { ❸
      String expected = "101";
      assertEquals(expected, binString.binarise(5));
      expected = "11111100";
      assertEquals(expected, binString.binarise(252));
   }

   public void testTotalConversion() { ❹
     String expected = "1000001";
     assertEquals(expected, binString.convert("A"));
   }
}
```

❶

```
    protected void setUp() {
       binString = new BinString();
    }
```

TestCase allows us to use the method *setUp* to set up any necessary variables or objects. Note that *binString* was already declared at the top of the file like usual with *private Binstring binstring;*. *setUp* is called before the evaluation of each test, *setUp* has a brother called *tearDown* which is called after the evaluation of each test and can be used to dereference variables or whatever so that each test may be performed without issuing side-effects that may affect the outcome of the other tests.

❷

```
    public void testSumFunction() {
       int expected = 0;
       assertEquals(expected, binString.sum(""));
       expected = 100;
       assertEquals(expected, binString.sum("d"));
       expected = 265;
       assertEquals(expected, binString.sum("Add"));
    }
```

Here we are testing the auxiliary function *sum* from our example file.

❸

```
    public void testBinariseFunction() {
       String expected = "101";
       assertEquals(expected, binString.binarise(5));
       expected = "11111100";
       assertEquals(expected, binString.binarise(252));
    }
```

Here we are testing the auxiliary function *binarise* which converts an *int* to a binary String.

❹

```
    public void testTotalConversion() {
      String expected = "1000001";
      assertEquals(expected, binString.convert("A"));
    }
```

Here we are testing the *convert* function which calls the auxiliary methods to perform the conversion from String to integer to binary string.

Compile both these files and then run the tester on the TestCase by executing the following command sequence:

`java junit.textui.TestRunner BinStringTest`

This will produce the following errors:

```
..F.
```

```
Time: 0.01
There was 1 failure:
1) testBinariseFunction(BinStringTest)junit.framework.AssertionFailedError: expected:<11111100> but was
 at BinStringTest.testBinariseFunction(BinStringTest.java:26)
 at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
 at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
 at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)

FAILURES!!!
Tests run: 3,  Failures: 1,  Errors: 0
```

This tells us that there was an error in the Binarise function and it even tells us the actual result compared to the expected result, a quick glance shows that the binary string has been output in reverse so we fix this by changing the *binarise* method in the file `BinString.java` to:

```
public String binarise(int x) {
   if(x==0) return "";
   if(x%2==1) return binarise(x/2)+"1";
   return binarise(x/2)+"0";
}
```

Recompile this and execute the test again and you will see:

```
...
Time: 0.01

OK (3 tests)
```

Which indicates that everything went fine. Try this, comment out this line in the `BinString.java` file:

```
if(s.length()==1) return ((int)(s.charAt(0)));
```

Recompile the file and execute the test again, you will see:

```
.E..E
Time: 0.01
There were 2 errors:
1) testSumFunction(BinStringTest)
java.lang.StringIndexOutOfBoundsException: String index out of range: 0
 at java.lang.String.charAt(String.java:455)
 at BinString.sum(BinString.java:13)
 at BinString.sum(BinString.java:13)
 at BinStringTest.testSumFunction(BinStringTest.java:17)
 at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
 at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
 at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
2) testTotalConversion(BinStringTest)
java.lang.StringIndexOutOfBoundsException: String index out of range: 0
 at java.lang.String.charAt(String.java:455)
 at BinString.sum(BinString.java:13)
 at BinString.sum(BinString.java:13)
 at BinString.convert(BinString.java:7)
 at BinStringTest.testTotalConversion(BinStringTest.java:31)
 at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
 at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
 at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)

FAILURES!!!
Tests run: 3,  Failures: 0,  Errors: 2
```

Notice that there were no failures but the test still failed, this is because there was errors, errors are caused by completely unexpected occurrences, such as exceptions as indicated by the error messages displayed. Whenever an assertion such as an *assertEquals* fails it will throw a AssertionFailedError which is caught by the JUnit framework. Sometimes you may need to test that a certain input produces an exception, for example, we may have a String manipulation method which should throw some defined exception if passed an empty string. We could use the following to test that it does so:

```
try {
   stringmanipulationmethod("");
   fail("Should raise a someDefinedException exception here");
} catch(somedefinedException e) {
   // successful test
```

```
}
```

If the string manipulation method did not throw the exception then program control would be directed to the fail statement and we would get an informative message upon executing the test unit.

5. References

- http://members.pingnet.ch/gamma/junit.htm

  Test Infected:
  Programmers Love Writing Tests

  Kent Beck, CSLife
  Erich Gamma, OTI Zurich

- http://junit.sourceforge.net/doc/cookbook/cookbook.htm

  JUnit Cookbook
  Kent Beck, Erich Gamma

- http://junit.sourceforge.net/doc/cookstour/cookstour.htm

  JUnit A Cook's Tour
  Erich Gamma, Kent Beck