Unit Testing in Java Using the JUnit Framework

The JUnit Framework can be easily integrated with Eclipse. Unit testing accelerates programming speed.

While software testing is generally performed by the professional software tester, unit testing is often performed by the software developers working on the project, at that point of time. Unit testing ensures that the specific function is working perfectly. It also reduces software development risks, cost and time. Unit testing is performed by the software developer during the construction phase of the software development life cycle. The major benefit of unit testing is that it reduces the construction errors during software development, thus improving the quality of the product. Unit testing is about testing classes and methods.

What is JUnit?

JUnit is a testing tool for the Java programming language. It is very helpful when you want to test each unit of the project during the software development process.

How to perform unit testing using the JUnit testing tool

To perform JUnit testing in Java, first of all, you have to install the Eclipse editor. Installation of the latest version is recommended. You can download the Eclipse IDE from the following link: *http://eclipse.org/downloads/*.

In the Eclipse editor, you can write any code. For example, let's suppose that I want to test the following code:

Code-1

```
package com;

public class Junit {

   public String concatenate(String firstName, String
lastName) {

     return firstName + lastName;
   }

   public int multiply(int number1, int number2) {

     return number1 * number2;
   }

}
```

After writing Code-1, let's write two test cases—one for the *concatenate* method and the other for the *multiply* method of the JUnit class defined in this code. To create JUnit test cases, you need to click on the Eclipse editor: *File→New→JUnit Test Case*

Defining the test case for the *concatenate()* method of the JUnit class (Code-1)

Code-2

```
package com;
import static org.junit.Assert.*;
import org.junit.Test;

public class ConcatTest {

```

```
7    @Test
8    public void testConcatnate() {
9
10     Junit test = new Junit();
11
12     String result = test.concatenate("Vikas","Kumar");
13
14     assertEquals("VikasKumar", result);
15
16   }
17
18}
```

Code-2 is the test case for the *concatenate()* method defined inside the JUnit class in Code-1. The annotation *@Test* at Line 7 is supported by JUnit version 4. To add JUnit version 4, you can click on *Project directory* in the Eclipse IDE and go to *Java Build Path*, before clicking on *Add Library* and then on *JUnit*, where you select *Junit 4*.

The *assertEquals()* method is a predefined method, and it takes two parameters. The first parameter is called *expected output* and the second is *original output*. If the expected output doesn't match the original output, then the test case fails. To run the test cases, right click the Eclipse code and then click on *Run as JUnit Test*.

Defining the test case for the *multiply()* method of JUnit class (Code-1)

Code-3

```
package com;

import static org.junit.Assert.*;
import org.junit.Test;

public class MultiplyTest {

   @Test
   public void testMultiply() {

     Junit test = new Junit();

     int result = test.multiply(5, 5);

     assertEquals(25, result);
   }

}
```

Code-3 is the test case for the *multiply()* method of the JUnit class defined above.

Creating a test suite

A test suite is a combination of multiple test cases. To create a JUnit test suite, you need to click on the following in Eclipse: *File→Other→Java→JUnit→JUnit Test Suite*

After creating the JUnit test suite, the code will look like what is shown in the Code-4 snippet.

Code-4

```
package com;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({ ConcatTest.class, MultiplyTest.class })
public class AllTests {

}
```

Understanding the *@Before* annotation

The *@Before* annotation is used to annotate the method that has to be executed before the actual test method gets executed. To understand this, let's look at Code-5.

Code-5

```
package com;

public class Calculator {

   public int add(int x, int y) {

     return x + y;
   }

   public int sub(int x, int y) {

     return x - y;

   }
}
```

Now let's create the test case for Code-5. The following code is the JUnit test case for the *Calculator* class defined in this code.

Code-6

```
package com;

import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;

public class CaculatorTest {

```

```
9    Calculator cal;
10
11   @Before
12   /*
13     the init() method will be called for each test, such
14     testAdd() as well as testSub()
15    */
16   public void init() {
17
18     cal = new Calculator();
19
20   }
21
22   @Test
23   public void testAdd() {
24
25     int x = 10;
26     int y = 20;
27     assertEquals(30, cal.add(x, y));
28
29   }
30
31   @Test
32   public void testSub() {
33     int x = 10;
34     int y = 20;
35     assertEquals(-10, cal.sub(x, y));
36   }
37
38 }
```

Parameterised unit test cases using JUnit

If you want to test any method with multiple input values, you would normally have to write multiple test cases for the same method. But if you use the parameterised unit testing technique, you don't need to write multiple test cases for the same method.

Let's look at the example of the *Calculator* class defined in Code-5. If you have to create parameterised test cases for the *add()* method of the Calculator class with multiple inputs, then consider the following code for that requirement.

Code-7

```
1     package com.emertxe;
2
3  import static org.junit.Assert.*;
4  import java.util.Arrays;
5  import java.util.Collection;
6  import org.junit.Assert;
7  import org.junit.Before;
8  import org.junit.Test;
9  import org.junit.runner.RunWith;
10 import org.junit.runners.Parameterized;
11 import org.junit.runners.Parameterized.Parameters;
12
13 @RunWith(Parameterized.class)
14 public class AddParamTest {
15
16   private int expectedResult;
17   private int firstVal;
18   private int secondVal;
19   Calculator cal;
20
21   @Before
22   public void init() {
23
24     cal = new Calculator();
25   }
26
27   public AddParamTest(int expectedResult, int firstVal, int
secondVal) {
28     this.expectedResult = expectedResult;
29     this.firstVal = firstVal;
30     this.secondVal = secondVal;
31   }
32
33   @Parameters
34   public static Collection<Object[]> testData() {
35
36     Object[][] data = new Object[][] { { 6, 2, 4 }, { 7, 4, 3 },
37             { 8, 2, 6 } };
38
39     return Arrays.asList(data);
40   }
41
42   @Test
43   public void testAdd() {
44  Assert.assertEquals(expectedResult, cal.add(firstVal,
secondVal));
45   }
46 }
```

When the test case written in Code-7 is executed, then an execution occurs in the following order:

1. Parameterised class at Line 11 is executed.
2. Static method at Line 32 is executed.
3. Instance of *AddParamTest* class at Line 12 is executed.
4. The data {6,2,4}, {7,4,3} and {8,2,6} at Lines 34-35 is passed to the constructor at Line 24.
5. *testAdd()* method at Line 41 is executed. END

By: Vikas Kumar Gautam

The author is a mentor at Emertxe Information Technology (P) Ltd. His main areas of expertise include application development using Java/J2EE and Android for both Web and mobile devices. A Sun Certified Java Professional (SCJP), his interests include acquiring greater expertise in the application space by learning from the latest happenings in the industry. He can be reached at *vikash_kumar@emertxe.com*