INTRODUCTION TO UNIT TESTING

Installing and Using JUnit

Installing JUnit: Goto http://junit.org/ ⇒ "Download and Install Guide" ⇒ Download the "`junit.jar`" and "`hamcrest-core.jar`". You could download the API documentation as well as the source code.

Using JUnit: To use the JUnit, include JUnit jar-files "`junit-4.##.jar`" and "`hamcrest-core-1.##.jar`" in your `CLASSPATH`.

Using JUnit under Eclipse

Include JUnit Library in your Java Project: Create a new Java project ⇒ right-click on the project ⇒ Properties ⇒ Java Build Path ⇒ "Libraries" tab ⇒ Add Library ⇒ JUnit ⇒ In "JUnit library version", choose "JUnit 4" ⇒ In "current location" use the eclipse's JUnit or your own download. [Alternatively, when you create a new test case or test suite (as describe below), Eclipse will prompt you to include the JUnit library.]

Create Test case (or Test Suite): To create a new JUnit test case (or test suite, which contains many test cases) ⇒ File ⇒ Others ⇒ Java ⇒ JUnit ⇒ JUnit test case (or JUnit test suite).

Run Test case (or Test Suite): To run a test case (or test suite), right-click the file ⇒ Run As ⇒ JUnit Test.

JUnit 4

There are two versions of JUnit, version 3 and version 4, which are radically different. JUnit 4 uses the *annotation* feature (since JDK 1.5) to streamline the process and drop the strict naming conventions of test methods.

Getting Started with an Example

Suppose that we wish to carry out unit testing on the following Java program, which uses `static` methods to perform arithmetic operations on two integers. Take note that divide throws an `IllegalArgumentException` for divisor of zero.

```
1/**
2 * The Calculator class provides static methods for
3 *  arithmetic operations on two integers.
4 */
5public class Calculator {
6   public static int add(int number1, int number2) {
7      return number1 + number2;
8   }
9
10  public static int sub(int number1, int number2) {
11     return number1 - number2;
12  }
13
14  public static int mul(int number1, int number2) {
15     return number1 * number2;
16  }
```

```
17
18   // Integer divide. Return a truncated int.
19   public static int divInt(int number1, int number2) {
20      if (number2 == 0) {
21         throw new IllegalArgumentException("Cannot divide by 0!");
22      }
23      return number1 / number2;
24   }
25
26   // Real number divide. Return a double.
27   public static double divReal(int number1, int number2) {
28      if (number2 == 0) {
29         throw new IllegalArgumentException("Cannot divide by 0!");
30      }
31      return (double) number1 / number2;
32   }
33}
```

First Test Case

Let's do it under Eclipse.

1. Create a new Eclipse Java project called "`JUnitTest`".
2. Create a new class called "`Calculator`" under "`src`" folder, with the above program code.
3. Create a new folder called "`test`" for storing test scripts ⇒ Right-click on the project ⇒ New ⇒ Folder ⇒ In folder name, enter "`test`". Make "`test`" a source folder by right-click on "`test`" ⇒ Build Path ⇒ Use as source folder.
4. Create the first test case called "`AddSubTest`" ⇒ Right-click on folder "`test`" ⇒ New ⇒ Other ⇒ Java ⇒ JUnit ⇒ JUnit Test Case ⇒ New JUnit 4 test ⇒ In Name, enter "`AddSubTest`". Enter the following codes:

```
 1import static org.junit.Assert.*;
 2import org.junit.Test;
 3
 4public class AddSubTest {
 5   @Test
 6   public void testAddPass() {
 7      // assertEquals(String message, long expected, long actual)
 8      assertEquals("error in add()",  3, Calculator.add(1, 2));
 9      assertEquals("error in add()", -3, Calculator.add(-1, -2));
10      assertEquals("error in add()",  9, Calculator.add(9, 0));
11   }
12
13   @Test
14   public void testAddFail() {
15      // assertNotEquals(String message, long expected, long actual)
16      assertNotEquals("error in add()", 0, Calculator.add(1, 2));
17   }
```

```
18
19   @Test
20   public void testSubPass() {
21      assertEquals("error in sub()",  1, Calculator.sub(2, 1));
22      assertEquals("error in sub()", -1, Calculator.sub(-2, -1));
23      assertEquals("error in sub()",  0, Calculator.sub(2, 2));
24   }
25
26   @Test
27   public void testSubFail() {
28      assertNotEquals("error in sub()", 0, Calculator.sub(2, 1));
29   }
30}
```

5. To run the test case, right-click on the file ⇒ Run as ⇒ JUnit Test. The test result is shown in the JUnit panel. 4 tests were run and all succeeded. Study the test results.
6. Try modify one of the test to force a test failure and observe the test result, e.g.,

```
7. @Test
8. public void testAddPass() {
9.    assertEquals("error in add()",  0, Calculator.add(1, 2));
10.   .....

   }
```

Explanation

- A test case contains a number of tests, marked with annotation of "`@org.junit.Test`". Each of the test is run independently from the other tests.
- Inside the test method, we can use static methods `assertXxx()` (in class `org.unit.Assert`) to assert the expected and actual test outcomes, such as:

```
•  public static void assertEquals([String message,] long expected, long actual)
•        // int and long: expected == actual
•  public static void assertEquals([String message,] double expected, double actual,
   double epsilon)
•        // double: expect == actual within tolerance of epsilon
•  public static void assertEquals([String message,] Object expected, Object actual)
•        // Object: expected.equals(actual)
•  public static void assertNotEquals(.....)
•
•  public static void assertSame([String message,] Object expected, Object actual)
•        // Same Object: expected == actual
•  public static void assertNotSame(.....)
•
•  public static void assertTrue([String message,] boolean condition)
•        // boolean: condition == true
```

```
public static void assertFalse([String message,] boolean condition)
      // boolean: condition == false

public static void assertNull([String message,] Object object)
      // object == null
public static void assertNotNull(......)

public static void assertArrayEquals([String message,], int[] expecteds, int[] actuals)
public static void assertArrayEquals([String message,], byte[] expecteds, byte[] actuals)
public static void assertArrayEquals([String message,], char[] expecteds, char[] actuals)
public static void assertArrayEquals([String message,], long[] expecteds, long[] actuals)
public static void assertArrayEquals([String message,], byte[] expecteds, byte[] actuals)
public static void assertArrayEquals([String message,], short[] expecteds, short[] actuals)
public static void assertArrayEquals([String message,], Object[] expecteds, Object[] actuals)

public static <T> void assertThat([String message,], T actual, org.hamcrest.Matcher<T> matcher)
```

Run Test Standalone via Test Runner

To run your test standalone (outside Eclipse), you could write a Test Runner as follows:

```
1import org.junit.runner.JUnitCore;
2import org.junit.runner.Result;
3import org.junit.runner.notification.Failure;
4
5public class RunTestStandalone {
6   public static void main(String[] args) {
7      Result result = JUnitCore.runClasses(AddSubTest.class);
8      for (Failure failure : result.getFailures()) {
9         System.out.println(failure.toString());
10     }
11     System.out.println(result.wasSuccessful());
12  }
13}
```

You can include more than one test cases using `runClasses(test1.class, test2.class, ...)`.

Run Test in Command-line

JUnit also provides a console version of test-runner called org.junit.runner.JUnitCore for you to run the tests in command-line, with the following syntax:

```
// Need to include JUnit's jar-file in CLASSPATH.
$ java org.junit.runner.JUnitCore TestClass1 [TestClass2 ...]
```

1. Copy all your classes into one folder (for simplicity).
2. Set the `CLASSPATH` to include the JUnit jar-files:

```
// Unix/Linux/Ubuntu/Mac (for this bash session only)
$ export CLASSPATH=.:$CLASSPATH:/path/to/junit/junit-4.11.jar:/path/to/junit/hamcrest-core-1.3.jar

// Windows (for this CMD session only)
> set CLASSPATH=.;%CLASSPATH%;x:\path\to\junit\junit-4.11.jar;x:\path\to\junit\hamcrest-core-1.3.jar
```

8. Compile all the source files:

```
$ cd /path/to/source-files
```

```
$ javac Calculator.java AddSubTest.java
```

10. Run the test:

```
$ java org.junit.runner.JUnitCore AddSubTest
JUnit version 4.11
....
Time: 0.006
```

```
OK (4 tests)
```

Second Test Case

Let's write another test case to test the divide methods, which throw an Exception for divisor of zero. Furthermore, the method `divReal()` returns a double which cannot be compared with absolute precision.

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

public class DivTest {
   @Test
   public void testDivIntPass() {
      assertEquals("error in divInt()", 3, Calculator.divInt(9, 3));
      assertEquals("error in divInt()", 0, Calculator.divInt(1, 9));
   }

   @Test
   public void testDivIntFail() {
      assertNotEquals("error in divInt()", 1, Calculator.divInt(9, 3));
```

```
14   }
15
16   @Test(expected = IllegalArgumentException.class)
17   public void testDivIntByZero() {
18      Calculator.divInt(9, 0); // expect an IllegalArgumentException
19   }
20
21   @Test
22   public void testDivRealPass() {
23      assertEquals("error in divInt()", 0.333333, Calculator.divReal(1, 3), 1e-6);
24      assertEquals("error in divInt()", 0.111111, Calculator.divReal(1, 9), 1e-6);
25   }
26
27   @Test(expected = IllegalArgumentException.class)
28   public void testDivRealByZero() {
29      Calculator.divReal(9, 0); // expect an IllegalArgumentException
30   }
31}
```

Run the test and observe the test result. Change `testDivRealPass()`'s expected value from 0.333333 to 0.333330 and check the test result.

Explanation

- To test for exception, use annotation `@Test` with attribute `expected = ExceptionClassName`.
- To compare doubles, use a tolerance (epsilon) as shown.

First Test Suite

A test suite comprises many test cases.

To create a test suite under Eclipse ⇒ right-click on the `test` folder ⇒ New ⇒ other ⇒ Java ⇒ JUnit ⇒ JUnit Test Suite ⇒ In Name, enter "`AllTests`" ⇒ Select test cases to be included - AddSubTest and DivTest.

The following test script will be created:

```
1import org.junit.runner.RunWith;
2import org.junit.runners.Suite;
3import org.junit.runners.Suite.SuiteClasses;
4
5@RunWith(Suite.class)
6@SuiteClasses({ AddSubTest.class, DivTest.class })
7public class AllTests {
8
9}
```

Take note that the test suite class is marked by annotation `@RunWith` and `@SuiteClasses` with an empty class body.

To run the test suite ⇒ right-click on the file ⇒ Run as ⇒ JUnit Test. Observe the test results produced.

You can also run the test suite via Test Runner `org.junit.runner.JUnitCore`, just like running test cases (as described earlier).

3.2 Testing Java Classes By Example

Instead of testing static methods in a Java class, let's carry out unit test on a proper self-contained and encapsulated Java class with its own private variables and public operations.

Suppose that we have a class called `MyNumber` that represents a number, and capable of performing arithmetic operations.

Again, we shall work under Eclipse.

1. Create a Java project called "`JUnitTest2`"
2. Create a new Java class called "`MyNumber`", as follow:

```
 1/**
 2 * The class MyNumber represent a number, and capable
 3 * of performing arithmetic operations.
 4 */
 5public class MyNumber {
 6   int number;
 7
 8   // Constructor
 9   public MyNumber() {
10      this.number = 0;
11   }
12
13   public MyNumber(int number) {
14      this.number = number;
15   }
16
17   // Getter and setter
18   public int getNumber() {
19      return number;
20   }
21
22   public void setNumber(int number) {
23      this.number = number;
24   }
25
26   // Public methods
27   public MyNumber add(MyNumber rhs) {
28      this.number += rhs.number;
29      return this;
30   }
31
```

```
32   public MyNumber div(MyNumber rhs) {
33      if (rhs.number == 0) throw new IllegalArgumentException("Cannot divide by 0!")
34      this.number /= rhs.number;
35      return this;
36   }
37}
```

3. Create a new source folder called "`test`" for storing test scripts. Make it a source folder by right-click ⇒ Build Path ⇒ Use as source folder.
4. Create the first test case called MyNumberTest (under "`test`" folder), as follows:

```
 1import static org.junit.Assert.*;
 2import org.junit.After;
 3import org.junit.Before;
 4import org.junit.Test;
 5
 6public class MyNumberTest {
 7   private MyNumber number1, number2; // Test fixtures
 8
 9   @Before
10   public void setUp() throws Exception {
11      System.out.println("Run @Before"); // for illustration
12      number1 = new MyNumber(11);
13      number2 = new MyNumber(22);
14   }
15
16   @After
17   public void tearDown() throws Exception {
18      System.out.println("Run @After"); // for illustration
19   }
20
21   @Test
22   public void testGetterSetter() {
23      System.out.println("Run @Test testGetterSetter"); // for illustration
24      int value = 33;
25      number1.setNumber(value);
26      assertEquals("error in getter/setter", value, number1.getNumber());
27   }
28
29   @Test
30   public void testAdd() {
31      System.out.println("Run @Test testAdd"); // for illustration
32      assertEquals("error in add()", 33, number1.add(number2).getNumber());
33      assertEquals("error in add()", 55, number1.add(number2).getNumber());
34   }
35
```

```
36   @Test
37   public void testDiv() {
38      System.out.println("Run @Test testDiv"); // for illustration
39      assertEquals("error in div()", 2, number2.div(number1).getNumber());
40      assertEquals("error in div()", 0, number2.div(number1).getNumber());
41   }
42
43   @Test(expected = IllegalArgumentException.class)
44   public void testDivByZero() {
45      System.out.println("Run @Test testDivByZero"); // for illustration
46      number2.setNumber(0);
47      number1.div(number2);
48   }
49}
```

5. Run the test and observe the result. Modify some lines to make the test fails and observe the result.

   The output, used for illustrating the sequence of operations, is as follows:

```
Run @Before
Run @Test testDivByZero
Run @After
Run @Before
Run @Test testAdd
Run @After
Run @Before
Run @Test testDiv
Run @After
Run @Before
Run @Test testGetterSetter
Run @After
```

Test Fixtures, `@Before` and `@After`

A test fixtures is a fixed state of a set of objects used as a baseline for running tests. The purpose of a test fixture is to ensure that there is a well known and fixed environment in which tests are run so that results are repeatable.

In JUnit 4, fixtures are setup via the `@Before` and `@After` annotations.

- The `@Before` annotated method (known as `setup()`) will be run before EACH test method (annotated with `@Test`) to set up the fixtures.
- The `@After` annotation method (known as tearDown()) will be run after EACH test.

We typically declare text fixtures as private instance variables; initialize via `@Before` annotated method; and clean-up via `@After` annotation method. Each test method runs on its own set of text fixtures with the same initial states. This ensures isolation between the test methods.

`@BeforeClass` and `@AfterClass`

Beside the `@Before` and `@After`, there is also `@BeforeClass` and `@AfterClass`.

- The `@BeforeClass` annotated method will be run once before any test, which can be used to perform one-time initialization tasks such as setting up database connection.
- The `@AfterClass` annotated method will be run once after all tests, which can be used to perform housekeeping tasks such as closing database connection.