JUnit4.x

Introduction to JUnit

Lesson Objectives

- **In this lesson, you will learn:**
  - Installing and running JUnit
  - Using JUnit within Eclipse

What is Unit Testing?

- **The process of testing the individual subprograms, subroutines, or procedures to compare the function of the module to its specifications is called Unit Testing.**
  - Unit Testing is relatively inexpensive and an easy way to produce better code.
  - Unit testing is done with the intent that a piece of code does what it is supposed to do.

Steps for Installing JUnit, Running JUnit

- **Following are the steps for installing and running JUnit:**
  1. Download JUnit from **www.junit/org.** You can download either the jar file or the zip file.
  2. Add the jar file to the CLASSPATH.
  3. Test the installation by running the following:
     java org.junit.runner.JUnitCore org.junit.tests.Alltests.

Using JUnit within Eclipse

- **JUnit can be easily plugged in with Eclipse.**
- **Let us understand how JUnit can be used within Eclipse.**
  - Consider a simple "Hello World" program.
  - The code is tested using JUnit and Eclipse IDE.
- **Steps for using JUnit within JUnit:**
  1. Open a new Java project.
  2. Add **junit.jar** in the project Build Path.

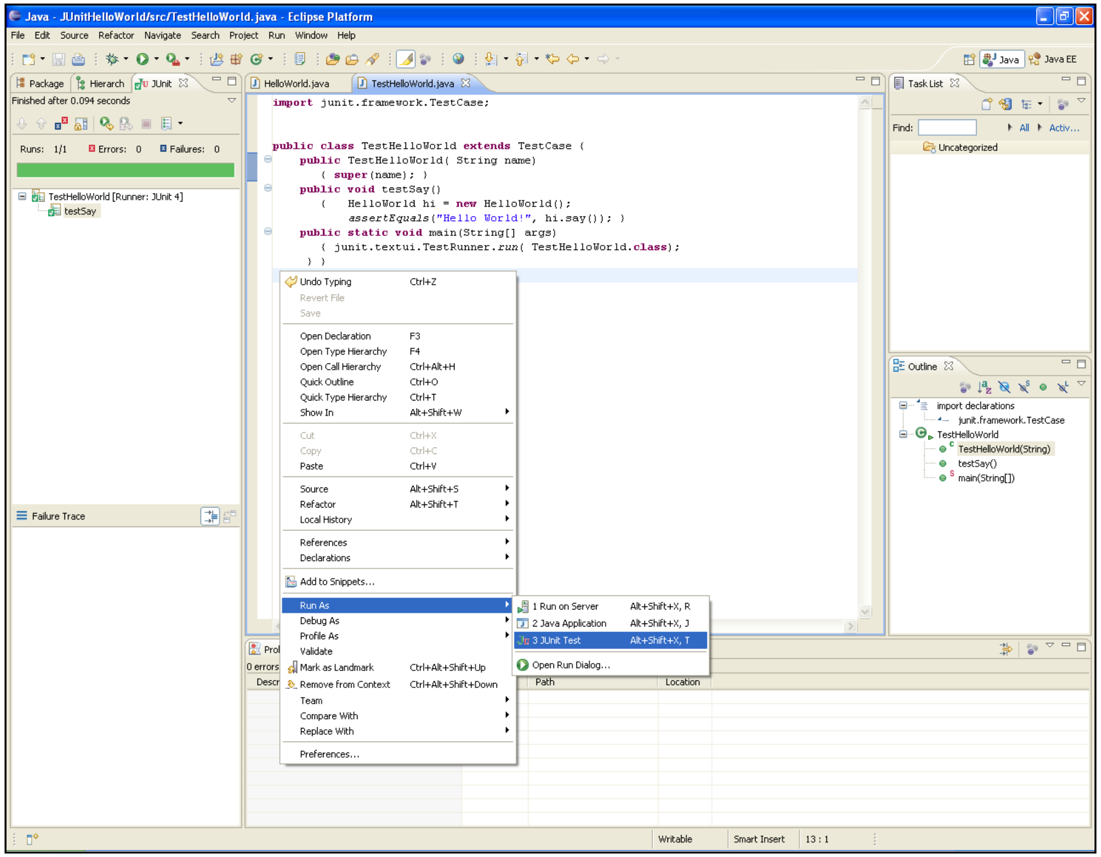
Using JUnit within Eclipse

– Write the Test Case as follows:

```
import junit.framework.TestCase;
 public class TestHelloWorld extends TestCase {
   public TestHelloWorld( String name)
     { super(name); }
   public void testSay()
     {   HelloWorld hi = new HelloWorld();
          assertEquals("Hello World!", hi.say()); }
   public static void main(String[] args)
     { junit.textui.TestRunner.run( TestHelloWorld.class);
    } }
```

Using JUnit within Eclipse

- **Run the Test Case.**
  - Right-click the Project → Run As → JUnit Test
- **The output of the test case is seen in Eclipse.**



```
Finished after 0.094 seconds
Runs: 1/1    Errors: 0    Failures: 0
TestHelloWorld [Runner: JUnit 4]
    testSay
```

```java
import junit.framework.TestCase;

public class TestHelloWorld extends TestCase {
    public TestHelloWorld( String name)
       { super(name); }
    public void testSay()
       {   HelloWorld hi = new HelloWorld();
           assertEquals("Hello World!", hi.say()); }
    public static void main(String[] args)
       { junit.textui.TestRunner.run( TestHelloWorld.class);
     } }
```

JUnit4.x

Testing with JUnit

Comparing JUnit4.x with JUnit3.x

JUnit4.x	JUnit3.x
1. The TestCase class need not be extended for writing tests.	1. All classes must derive from TestCase.
2. Annotations are available for initialization and cleanup: Before, @After	2. Setup() and tearDown() methods are used for initialization and cleanup.

Comparing JUnit4.x with JUnit3.x

JUnit4.x	JUnit3.x
3. Test methods use the @Test annotation.	3. Test methods must be called testXXX().
4. In JUnit 4.x, you will import org.junit.*	4. In JUnit 3.x, you will import junit.framework.*
5. This has failures, successes, and ignored tests.	5. This has failures, errors, and successes.

Simple Example

- **Consider the following code snippet:**

```
import static org.junit.Assert.*;
import org.junit.Test;
 public class FirstJUnitTest {
  @Test
      public void simpleAdd() {
     int result = 1;
     int expected = 1;
     assertEquals(expected, actual);
 } }
```

Annotation Types in JUnit4.x

- **JUnit4.x introduces support for the following annotations:**
  - @Test
  - @Before
  - @After
  - @BeforeClass
  - @AfterClass
  - @Ignore

Assert Statements in JUnit

- **Following are the methods in Assert class :**
  - Fail(String)
  - assertTrue(boolean)
  - assertEquals([String message],expected,actual)
  - assertNull([message],object)
  - assertNotNull([message],object)
  - assertSame([String],expected,actual)
  - assertNotSame([String],expected,actual)
  - assertThat(String,T actual, Matcher<T> matcher)

- assertThat(String,T actual, Matcher<T> matcher)

examples

- **assertThat(x,is(5));**
- **assertThat(sourcestring ,either(contains(“smith”)or contains(“smithe));**
- **assertThat(values,hasItem(anyOf(equalTo(“one”),equalTo (“Two”),equalTo(“Three”))));**

Demo

- **Demo on:**
  - Using @Test Annotation
  - Using Assert Methods

Testing using the Exceptions

- **It is ideal to check that exceptions are thrown correctly by methods.**
- **Use the expected parameter in @Test annotation to test the exception that should be thrown.**
- **For example:**

```
@Test(expected = ArithmeticException.class)
public void divideByZeroTest() {
calobj.divide(15,0);
}
```

Demo

- **Demo on:**
  - Exception Testing

Using @Before and @After

- **Test fixtures help in avoiding redundant code when several methods share the same initialization and cleanup code.**
- **Methods can be annotated with @Before and @After.**
  - **@Before:** This method executes before every test.
  - **@After:** This method executes after every test.
- **Any number of @Before and @After methods can exist.**
- **They can inherit the methods annotated with @Before and @After.**

Using @Before and @After

- **Example of @Before:**

```
@Before
public void beforeEachTest() {
Calculator cal=new Calculator();
Calculator cal1=new Calculator("5", "2"); }
```

- Example of @After:

```
@After
public void afterEachTest() {
Calculator cal=null;
Calculator cal1=null; }
```

Demo

- **Demo on:**
  - Using the @Before and @After annotations

Using @BeforeClass and @AfterClass

- **Suppose some initialization has to be done and several tests have to be executed before the cleanup.**
- **Then methods can be annotated by using the @BeforeClass and @AfterClass.**
  - **@BeforeClass:** It is executed once before the test methods.
  - **@AfterClass:** It is executed once after all the tests have executed.
- **Only one set of @BeforeClass and @Afterclass methods are allowed.**

Using @BeforeClass and @AfterClass

- Example of @BeforeClass:

```
@BeforeClass
public static void beforeAllTests() {
Connection
conn=DriverManager.getConnection(….);}
```

- Example of @AfterClass:

```
@AfterClass
public static void afterAllTests() {
conn.close; }
```

Demo

- **Demo on:**
  - Using the @BeforeClass and @AfterClass annotations

Using @Ignore

- **The @Ignore annotation notifies the runner to ignore a test.**
- **The runner reports that the test was not run.**
- **Optionally, a message can be included to indicate why the test should be ignored.**
- **This annotation should be added either in before or after the @Test annotation.**

Using @Ignore

- Example of @Ignore for a method:

```
@Ignore (“The network resource is not currently
available”)
@Test
  public void multiplyTest() { ……}
```

- Example of @Ignore for a class:

```
@Ignore public class TestCal {
@Test public void addTest(){ …. }
@Test public void subtractTest(){…..}
}
```

Demo

- **Demo on:**
  - Using the @Ignore

Composing Test into Test Suites

- **A testsuite comprises of multiple tests and is a convenient way to group the tests, which are related.**
- **It also helps in specifying the order for executing the tests.**
- **JUnit provides the following:**
  - **org.junit.runners.Suite class** : It runs a group of test cases.
  - **@RunWith :** It specifies runner class to run the annotated class.
  - **@Suite.SuiteClasses :** It specifies an array of test classes for the Suite.Class to run.
    - The annotated class should be an empty class.

Composing Test into Test Suites

- Example:

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
@RunWith(Suite.class)
@Suite.SuiteClasses({ TestCalAdd.class,
TestCalSubtract.class,
TestCalMultiply.class, TestCalDivide.class })
public class CalSuite {
// the class remains completely empty,
// being used only as a holder for the above
annotations
}
```

Demo

- **Demo on:**
  - Composing tests into Test Suites

Summary

- **In this lesson, you have learnt:**
  - Comparison between JUnit3.8 and JUnit4.x
  - The concept of JUnit Framework
  - The concepts of Annotations and Assertions
  - Composing Test Cases into Test Suites
  - Testing Isolation using Mock Objects