Core Java 8 and Development Tools

Lesson 13 : Introduction to Junit 4

Lesson Objectives

- After completing this lesson, participants will be able to
  - Understand importance of unit testing
  - Install and use JUnit 4
  - Use JUnit Within Eclipse

Why is Testing Necessary

- To test a program implies adding value to it.
  - Testing means raising the reliability and quality of the program.
  - One should not test to show that the program works rather that it does not work.
  - Therefore testing is done with the intent of finding errors.
- Testing is a costly activity.

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.

What is Test-Driven Development (TDD)

- Test-Driven Development, also called Test-First Development, is a technique in which you write unit tests before writing the application functionality.
  - Tests are **non-production code** written in the same language as the application.
  - Tests return a simple **pass** or **fail**, giving the developer immediate feedback.

Why Unit Testing

- You can cite following reasons for doing a Unit Test:
  - Unit testing helps developers find errors in code.
  - It helps you write better code.
  - Unit testing saves time later in the production/development cycle.
  - Unit testing provides immediate feedback on the code.

Need for Testing Framework

- Testing without a framework is mostly ad hoc.
- Testing without framework is difficult to reproduce.
- Unit testing framework provides the following advantages:
  - It allows to organize and group multiple tests.
  - It allows to invoke tests in simple steps.
  - It clearly notifies if a test has passed or failed.
  - It standardizes the way tests are written.

What is JUnit

- JUnit is a free, open source, software testing framework for Java.
- It is a library put in a jar file.
- It is not an automated testing tool.
- JUnit tests are Java classes that contain one or more unit test methods.

Why JUnit

- JUnit allows you to write tests faster while increasing quality and stability.
- It is simple, elegant, and inexpensive.
- The tests check their own result and provide feedback immediately.
- JUnit tests can be put together in a hierarchy of test suites.
- The tests are written in Java.

Steps for Installing JUnit

- Following are the steps for installing and running JUnit:
  - Download JUnit from www.junit/org. You can download either the jar file or the zip file.
    - Unzip the JUnit zip file
  - Add the jar file to the CLASSPATH.
    - Set CLASSPATH=.,%CLASSPATH%;junit-4.3.1.jar

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:
  - Open a new Java project.
  - Add junit.jar in the project Build Path.

Using JUnit within Eclipse (Contd.)

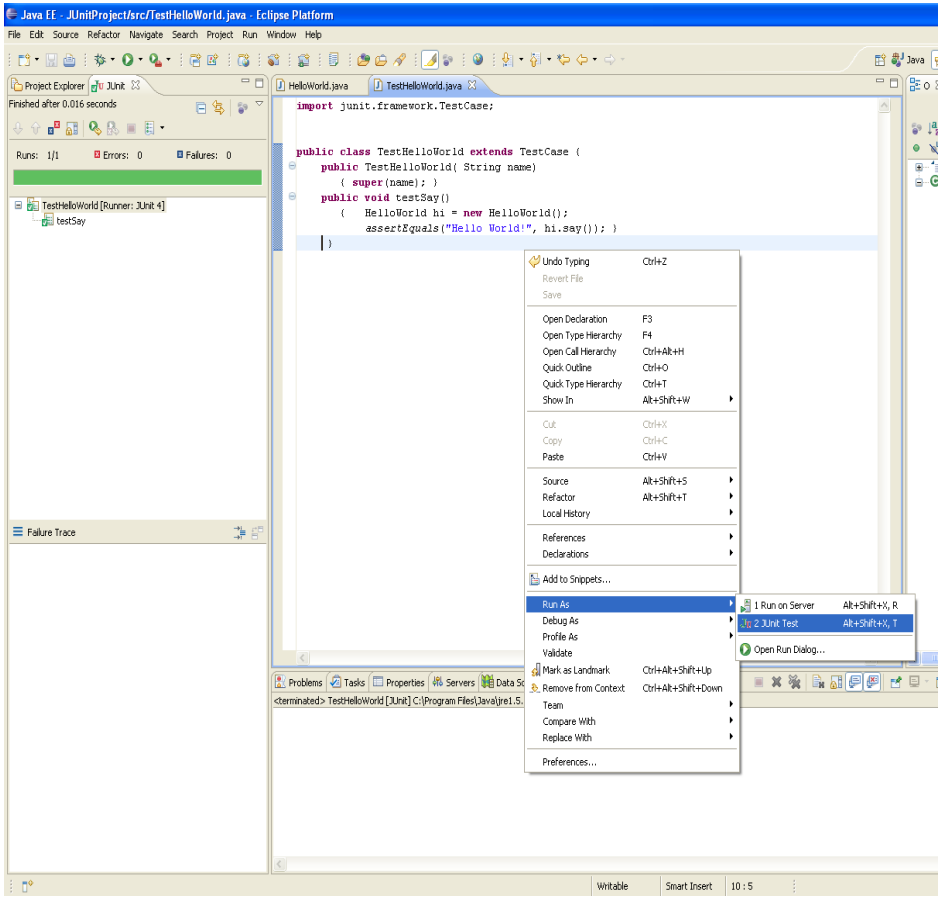
- Write the Test Case as follows:

```
import org.junit.Test;
import static org.junit.Assert.*;
public class TestHelloWorld {
    @Test
     public void testSay()
      {
          HelloWorld hi = new HelloWorld();
          assertEquals("Hello World!", hi.say());
      } }
```

```
class  HelloWorld{
     String say(){
           return “Hello World!”}
}
```

Using JUnit within Eclipse (Contd.)

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



Demo

- Demo on:
  - Using JUnit with Eclipse
    - HelloWorld.java
    - TestHelloWorld.java

Annotation Types in JUnit4.x

- JUnit4.x introduces support for the following annotations:
  - @Test – used to signify a method is a test method
  - @Before – can do initialization task before each test run
  - @After – cleanup task after each test is executed
  - @BeforeClass – execute task before start of tests
  - @AfterClass – execute cleanup task after all tests have completed
  - @Ignore – to ignore the test method

Simple Example using Junit4.x

- 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);
 } }
```

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)

Demo

- Demo on:
  - Using @Test Annotation
  - Using Assert Methods
    - Counter.java & Testcounter.java
    - Person.java & TestPerson.java

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 In Lesson-13:
  - Using the @Before and @After annotations
    - TestPersonFixture.java

Testing 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 In Lesson-13:
  - Exception Testing
    - Person.java & TestPerson2.java

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.

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; }
```

- The methods using this annotation should be public static void

Demo

- Demo In Lesson-13:
  - 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 In Lesson-13:
  - Using the @Ignore
    - Student.java & TestStudent.java

Unit Testing

- Start with writing tests for methods having the fewest dependencies and then work your way up.
- Ensure that tests are simple, preferably with no decision making.
- Use constant, expected values in the assertions instead of computed values wherever possible.

Unit Testing

- Ensure that each unit test is independent of all other tests.
- Clearly document all the tests.
- Test all methods whether public, protected, or private.
- Test the exceptions.

JUnit

- Do not use the constructor of test case to setup a test case, instead use an @Before annotated method.
- Do not assume the order in which tests within a test case should run.
- Place tests and the source code in the same location.
- Put non-parameterized tests in a separate class.

JUnit

- When writing tests consider the following questions:
  - When do I write tests?
  - Do I test everything?
  - How often the tests should be run?
  - Why use JUnit, instead why not use println() or a debugger?

Lab : Introduction to Junit

- Lab 5: Introduction to Junit

Review Question

- Question 1: Why should one do Unit Testing?
  - Option 1: Helps to write code better
  - Option 2: Provides immediate feedback on the code
  - Option 3: Because it is one of the testing methods that has to be carried out
- Question 2: JUnit is a licensed product and can be purchased with Java.
  - True / False

Review Question

- Question 3: To start working with JUnit in Eclipse, you need to add junit.jar in ___.
  - Option 1: CLASSPATH
  - Option 2: BUILDPATH
  - Option 3: Project Settings
- Question 4: You can add a number of tests using the <batchtest> element.
  - True / False