JUnit with Mockito

Mr. Ashok

Ashok IT School

Facebook Group Name: Ashok IT School
Email: ashok.javatraining@gmail.com

JUnit is an open source framework, developed initially by Erich Gamma and Kent Beck. It is mainly used by Java developers to write unit test cases.

Before discussing JUnit in detail, it is imperative to understand Software Testing Terminology.

Software Testing: Software Testing is the process of identifying the correctness and quality of software program. The purpose is to check whether the software satisfies the specific requirements, needs and expectations of the customer/client. In other words, testing is executing a system or application in order to find software bugs, defects or errors. The job of testing is to find out the reasons of application failures so that they can be corrected according to customer/client requirements.

Example: Car manufacturer tests the car for maximum speed, fuel efficiency and safety from crash. These test results later become the part of advertising strategy for car sales.

When we are developing software projects, there can be many reasons for defects in that software. The developer can also make an error which may result in a defect or bug in the software source code. Any defect or bug in the software will produce wrong results causing a failure. When a bug or defect causes in software application, testing is done to find out the cause of defect and to remove the bug.

As part of software development if bug introduced, it has to be rectified so that it does not disrupts the whole process of software program. That rectified part is again tested to confirm that it is compatible with the rest of program.

Most common software problems are:

- Incorrect implementation of the business rules
- Weak software performance
- Incorrect results of data searches
- Incorrect matching of data
- Inadequate security controls
- Confusing or misleading data
- Incorrect file handling, etc.

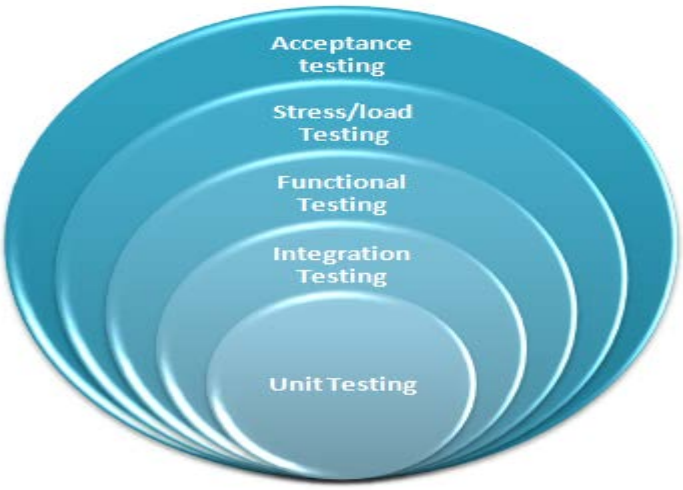
Ways of Software Testing: Software Testing can be performed in below 2 ways.

1. Manual Testing: Test Cases executed manually.
2. Automation Testing: Testing performed with the help of automation tools.

In manual testing the tester has the role of the end-user and try to find bugs or wrong behavior of the software.

Manual testing has some stages/levels as:

1) **Unit Testing:** It should examine the behavior of a distinct unit of work, like checking methods in classes by writing test cases.
2) **Integration Testing:** It should examine the interaction between components in their target environment.
3) **Functional Testing:** It should test the application use cases. Functional testing is a type of black box testing that bases its test cases on the specifications of the software component under test, and as such, should require no knowledge of the inner design of the code or logic.
4) **Stress/load Testing (Performance Testing):** It should test the performance of the application. Without performance testing, software is likely to suffer from issues such as: running slow while several users use it simultaneously, inconsistencies across different operating systems and poor usability. Performance testing will determine whether or not their software meets speed, scalability and stability requirements under expected workloads.
5) **Acceptance Testing:** It should ensure that the application has met the customer's goal.



Automation testing requires that the tester write scripts and use other software to test the developed software. The test scenarios performed manually are run more times very quickly.

As part of this material, we will understand what is Unit testing & how to perform Unit testing using Junit and EasyMock frameworks.

What is Unit Testing?

As part of application development process, Coding (or) Implementation of module/task is done by developers. After Coding is completed it will be submitted to Testing Team (QA=Quality Analysis Team) to test application functionality. QA Team will perform different types or levels of testing. But before submitting code to QA team, if Developers performs Individual tasks/modules testing then Errors (BUGs) rate will be reduced in Application.

In order to reduce the projects in project, programmer/developer cannot test the complete project, he/she can test only his/her module or task that he/she developed. This process of testing his/her module or task is called as Unit testing. Here Unit indicates a class, method, one layer, one module etc.

Unit Testing is a level of software testing where individual units/ components of a software application are tested.

The main objective of unit testing is to split code into multiple pieces and test each piece of code separately to ensure that, it works as expected.

Unit testing is used to verify a small chunk of code by creating a path, function or a method. The term "unit" exist earlier than the object-oriented era. It is basically a natural abstraction of an object oriented system i.e. a java class or object (its instantiated form).

Why to do Unit Testing? Why it is important?

- Unit testing is used to identify defects early in software development cycle.
- Unit testing will compel to read our own code. I.e. a developer starts spending more time in reading than writing.
- Defects in the design of code affect the development system. A successful code breeds the confidence of developer.
- Proper unit testing done during the development stage saves both time and money in the end.

Unit testing of software applications is done during the development (coding) of an application. Unit testing is usually performed by the developer.

Types of unit testing: There are two ways to perform unit testing

1. Manual testing
2. Automated testing.

Manual Testing: Manual Testing is used to test the program or application manually (means without using any additional tool)

Automated Testing: Automation testing is a process of testing an application by using some software tools

What is JUnit

JUnit is the most popular unit testing framework in Java. It is useful for java developers to write and run repeatable tests. It is an instance of xUnit architecture. As the name implies, it is used for unit testing of a small chunk of code.

JUnit is packaged as a single **JAR** file, it basically have a library of functionality which makes easier for a developer to create unit test in Java and verify the results of those tests.

JUnit Features

- JUnit is an open source framework which is used for writing & running tests.
- Provides Annotation to identify the test methods.
- Provides Assertions for testing expected results.
- Provides Test runners for running tests.
- JUnit tests allow you to write code faster which increasing quality
- JUnit is elegantly simple. It is less complex & takes less time.
- JUnit tests can be run automatically and they check their own results and provide immediate feedback. There's no need to manually comb through a report of test results.
- JUnit tests can be organized into test suites containing test cases and even other test suites.
- Junit shows test progress in a bar that is green if test is going fine and it turns red when a test fails.

JUnit Framework

JUnit is the most popular unit testing framework in Java. It is explicitly recommended for unit testing. JUnit does not require server for testing web application, which makes the testing process fast.

JUnit framework also allows quick and easy generation of test cases and test data. The org.Junit package consist of many interfaces and classes for JUnit testing such as Test, Assert, After, Before, etc.

URL to download JUnit Framework: http://www.junit.org/

Using JUnit with Maven

To use JUnit in Maven project, add the following dependency in pom.xml file.

```
<dependency>
   <groupId>junit</groupId>
   <artifactId>junit</artifactId>
   <version>4.12</version>
</dependency>
```

Let's move on to implement first JUnit program. Components required for developing JUnit program are

Test Fixture: A test fixture is a context where a test case runs.

```java
public class OutputFileTest {
	private File output;

	@Before
	public void createOutputFile(){
	 output = new File(...);
    }

	@After
	public void deleteOutputFile() {
		output.delete();
	}

	@Test
	public void testFile1() {
		// code for test case objective
	}

	@Test
	public void testFile2() {
		// code for test case objective
	}
}
```

OutputFileTest.java

Typically, test fixtures include:

- Objects or resources that are available for any test case.
- Activities required that makes these objects/resources available.
- These activities are
  - allocation (**setup**)
  - De-allocation (**teardown**).

Setup and Teardown

- Usually, there are some repeated tasks that must be done prior to each test case.
  Example: create a database connection.
- Likewise, at the end of each test case, there may be some repeated tasks.
  Example: to clean up once test execution is over.

JUnit provides annotations that help in setup and teardown. It ensures that resources are released, and the test system is in a ready state for next test case.

These annotations are discussed below-

Setup: @Before annotation is used on a method containing java code to run before each test case. i.e it runs before each test execution.
Teardown: @After annotation is used on a method containing java code to run after each test case. These methods will run even if any exceptions are thrown in the test case or in the case of assertion failures.

JUnit Test Suites

Test Suites are used to execute multiple tests in a specified order, it can be done by combining all the tests in one place. This place is called as the test suites

Junit Test Runner

JUnit Test Runner is used to execute our test cases.

- **JUnitCore** class is used to execute these tests.
- A method called **runClasses** provided by **org.junit.runner.JUnitCore,** is used to run one or several test classes.
- Return type of this method is the **Result** object (**org.junit.runner.Result**), which is used to access information about the tests. See following code example for more clarity.

```
public class Test {

   public static void main(String[] args) {
         Result result = JUnitCore.runClasses(CreateAndSetName.class);

         for (Failure failure : result.getFailures()) {
            System.out.println(failure.toString());
         }

         System.out.println(result.wasSuccessful());
   }
}
```

Test.java

In above code "result" object is processed to get failures and successful outcomes of test cases we are executing.

First JUnit program

Let's understand unit testing using below example. We need to create a test class with a test method annotated with **@Test** as given below:

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

public class MyFirstClassTest {

        @Test
        public void myFirstMethod() {
                String str = "JUnit is working fine";
                Assert.assertEquals("JUnit is working fine", str);
        }
}
```

MyFirstClassTest.java

To execute our test method (above) , we need to create a test runner. In the test runner we have to add test class as a parameter in JUnitCore's runclasses() method . It will return the test result, based on whether the test is passed or failed.

```
import org.JUnit.runner.JUnitCore;
import org.JUnit.runner.Result;
import org.JUnit.runner.notification.Failure;

public class TestRunner {
        public static void main(String[] args) {
                Result result = JUnitCore.runClasses(MyFirstClassTest.class);
                for (Failure failure : result.getFailures()) {
                        System.out.println(failure.toString());
                }
                System.out.println("Result==" + result.wasSuccessful());
        }
}
```

TestRunner.java

Once **TestRunner.java** executes our test methods we get output in junit console as failed or passed.

JUnit API

JUnit API includes various classes and annotations to write a test case. See below classes which are very useful while writing a test case

1. org.junit.Assert
2. org.junit.TestCase
3. org.junit.TestResult
4. org.junit.TestSuite

JUnit Assert Class

This class provides a bunch of assertion methods useful in writing a test case. If all assert statements are passed, test results are successful. If any assert statement fails, test results are failed. Below are the Assert methods and description

1. **void assertEquals(boolean expected, boolean actual) :** It checks whether two values are equals similar to equals method of Object class
2. **void assertFalse(boolean condition) :** "assertSame" functionality is to check that the two objects refer to the same object.
3. **void assertNotNull(Object object) :** "assertNotNull" functionality is to check that an object is not null.
4. **void assertNull(Object object) :** "assertNull" functionality is to check that an object is null.
5. **void assertTrue(boolean condition) :** "assertTrue" functionality is to check that a condition is true.
6. **void fail() :** If you want to throw any assertion error, you have fail() that always results in a fail verdict.
7. **void assertSame([String message] :** "assertSame" functionality is to check that the two objects refer to the same object.
8. **void assertNotSame([String message] :** "assertNotSame" functionality is to check that the two objects do not refer to the same object.

JUnit TestCase Class

To run multiple tests, TestCase class is available in org.junit.TestCase packages. Annotation @Test tells JUnit that this public void method (Test Case here) to which it is attached can be run as a test case.

Below table shows some important methods available in org.junit.TestCase class:
To run multiple test, TestCase class is available in org.junit.TestCase packages. Annotation @Test tells JUnit that this public void method (Test Case here) to which it is attached can be run as a test case

1. **int countTestCases() :** This method is used to count how many number of test cases executed by run(TestResult tr) method.
2. **TestResult createResult() :** This method is used to create a TestResult object.
3. **String getName() :** This method returns a string which is nothing but a TestCase.
4. **TestResult run() :** This method is used to execute a test which returns a TestResult object
5. **void run(TestResult result) :** This method is used to execute a test having a TestResult object which doesn't **returns anything.**
6. **void setName(String name) :** This method is used to set a name of a TestCase.
7. **void setUp() :** This method is used to write resource association code. e.g. Create a database connection.
8. **void tearDown() :** This method is used to write resource release code. e.g. Release database connection after performing transaction operation**.**
9. **@Test :** Test annotation is the creation of Test case to already developed project is known as Test.

JUnit TestResult Class

When you execute a test, it returns a result (in the form of **TestResult** object). This TestResult object can be used to analyze the resultant object. This test result can be either failure or successful. See below table for important methods used in org.junit.TestResult class:

1. **void addError(Test test, Throwable t) :** This method is used if you require add an error to the test.
2. **void addFailure (Test test, AssertionFailedError t) :** This method is used if you require add a failure to the list of failures.
3. **void endTest(Test test) :** This method is used to notify that a test is performed(completed)
4. **int errorCount() :** This method is used to get the error detected during test execution.
5. **Enumeration<TestFailure> errors() :** This method simply returns a collection (Enumeration here) of errors.
6. **int failureCount() :** This method is used to get the count of errors detected during test execution.
7. **void run(TestCase test) :** This method is used to execute a test case.
8. **int runCount() :** This method simply counts the executed test.

9. **void startTest(Test test) :** This method is used to notify that a test is started.
10. **void stop() :** This method is used to test run to be stopped.

JUnit Annotations

Annotations were introduced in Junit4, which makes java code more readable and simple. This is the big difference between Junit3 and Junit4 , that Junit4 is annotation based. With the knowledge of annotations in Junit4, one can easily learn and implement a JUnit test.

Below is the list of important and frequently used annotations:

1. **@Test** : This annotation is a replacement of org.junit.TestCase which indicates that public void method to which it is attached can be executed as a test Case.
2. **@Before:** This annotation is used if we want to execute some statement such as preconditions before each test case.
3. **@BeforeClass :** This annotation is used if we want to execute some statements before all the test cases for e.g. test connection must be executed before all the test cases.
4. **@After :** This annotation can be used if we want to execute some statements after each test case for e.g resetting variables, deleting temporary files ,variables, etc.
5. **@AfterClass :**This annotation can be used if we want to execute some statements after all test cases for e.g. Releasing resources after executing all test cases.
6. **@Ignores :** This annotation can be used if we want to ignore some statements during test execution for e.g. disabling some test cases during test execution.
7. **@Test(timeout=500) :** This annotation can be used if we want to set some timeout during test execution for e.g. if you are working under some SLA (Service level agreement), and tests need to be completed within some specified time.
8. **@Test(expected=IllegalArgumentException.class)** :This annotation can be used if we want to handle some exception during test execution. For, e.g., if you want to check whether a particular method is throwing specified exception or not.

JUnit Annotations Example

Let's create a class covering important JUnit annotations with simple print statements and execute it with a test runner class:

Step 1) Consider below java class having various methods which are attached to above-listed annotations:

```
public class MyFirstClassTest {
   private ArrayList<String> list;
   @BeforeClass
   public static void setUp() {
      System.out.println("Using @BeforeClass , executed before all test cases ");
   }
   @Before
   public void beforeTestMethod() {
      list = new ArrayList<String>();
      System.out.println("Using @Before annotations ,executed before each test cases ");
   }
   @AfterClass
   public static void tearDown() {
      System.out.println("Using @AfterClass ,executed after all test cases");
   }
   @After
   public void afterTestMethod() {
      list.clear();
      System.out.println("Using @After ,executed after each test cases");
   }
   @Test
   public void  businessMethod1() {
      list.add("test");
      assertFalse(list.isEmpty());
      assertEquals(1, list.size());
   }
}
```

MyFirstClassTest.java

Step 2) let's create a test runner class to execute above test:

```
import org.JUnit.runner.JUnitCore;
import org.JUnit.runner.Result;
import org.JUnit.runner.notification.Failure;

public class TestRunner {
        public static void main(String[] args) {
                Result result = JUnitCore.runClasses(MyFirstClassTest.class);
                for (Failure failure : result.getFailures()) {
                        System.out.println(failure.toString());
                }
                System.out.println("Result==" + result.wasSuccessful());
        }
}
```

TestRunner.java

Consider businessMethod( ) as given below : (Adding one object to list)

```
@Test
public void businessMethod() {
    list.add("test");
    assertFalse(list.isEmpty());
    assertEquals(1, list.size());
}
```

In above method as we are adding a string in the variable "list" so

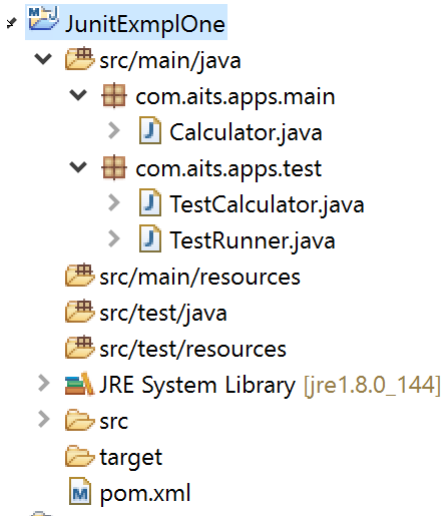
- **list.isEmpty()** will return false.
- **assertFalse(list.isEmpty())** must return true : As a result, the test case will **pass**.

As we have added only one obj in the list, so the size of the list is 1

- **list.size()** must return int value as "1" .
- So **assertEquals(1, list.size())** must return true : As a result, the test case will **pass**.

Example 1: In this example we will perform unit testing on Calculator class (Business class).

Step1: Create a maven project



Step 2: Create Calculator.java class like below

```java
package com.aits.apps.main;

public class Calculator {

	public int sum(int i, int j) {
		return i + j;
	}

	public int mul(int i, int j) {
		return i * j;
	}
}
```

Calculator.java

In the above source code, we can notice that the class has two public methods named sum() & mul(), both methods gets as inputs two integers, and performs calculation returns the result. Below, there is the code of the class named TestCalculator.java, which has the role of our test class:

Step 3: Create TestCalculator.java like below

```java
public class TestCalculator {
	private static Calculator calc = null;

	@BeforeClass
	public static void setUp() {
		calc = new Calculator();
	}
	@Test
	public void testSum() {
		int sumActualResult = calc.sum(2, 5);
		int sumExpResult = 7;
		assertEquals(sumExpResult, sumActualResult);
	}
	@Test
	public void testMul() {
		int mulActualResult = calc.mul(2, 5);
		int mulExpectedResult = 10;
		assertEquals(mulExpectedResult, mulActualResult);
	}
	@AfterClass
	public static void tearDown() {
		calc = null;
	}
 }
```

TestCalculator.java

Explanation: Firstly, we can see that there is a @Test annotation above the testSum() & testMul() methods. This annotation indicates that the public void method to which it is attached can be run as a test case. Hence, the testSum() method is the method that will test the sum() public method. We can also observe a method called assertEquals(sum, testsum). The method assertEquals ([String message], object expected, object actual) takes as inputs two objects and asserts that the two objects are equal.

Step 4: Create TestRunner class

```java
public class TestRunner {
	public static void main(String[] args) {
		Result result = JUnitCore.runClasses(TestCalculator.class);
		System.out.println("Total cases executed : " + result.getRunCount());
		System.out.println("Failure cases count : " + result.getFailureCount());
		System.out.println("Failure reasons are below...");
		List<Failure> failureList = result.getFailures();
		for (Failure f : failureList) {
			System.out.println(f.getMessage());
		}
	}
}
```

TestRunner.java

Execute above TestRunner class, by right-clicking in the TestRunner class and select Run As -> Junit Test.

Example 2: In this example, we will test Database connections using SingleConnectionProvider.java class

```java
public class SingleConnectionProvider {

	private static final String DB_URL = "jdbc:oracle:thin:@localhost:1521/XE";
	private static final String DB_UNAME = "system";
	private static final String DB_PWD = "admin";
	private static final String DB_DRIVER_CLASS = "oracle.jdbc.driver.OracleDriver";

	public Connection getConnection() throws Exception {
		Class.forName(DB_DRIVER_CLASS);
		Connection con = DriverManager.getConnection(DB_URL, DB_UNAME, DB_PWD);
		return con;
	}
}
```

SingleConnectionProvider.java

```java
public class TestSingleConnProvider {

	@Test
	public void testGetConnection() throws Exception {
		// Creating business class instance to access methods
		SingleConnectionProvider conProvider = new SingleConnectionProvider();

		// getting first connection
		Connection con1 = conProvider.getConnection();

		// getting second connection
		Connection con2 = conProvider.getConnection();

		// checking weather both connections are same or not
		assertSame(con1, con2);
	}
}
```

TestSingleConnProvider.java

```java
public class TestRunner {

	public static void main(String[] args) {

		Result result = JUnitCore.runClasses(TestSingleConnProvider.class);
		System.out.println("Total cases executed : " + result.getRunCount());
		System.out.println("Failure cases count : " + result.getFailureCount());
		System.out.println("Failure reasons are below...");
		List<Failure> failureList = result.getFailures();
		for (Failure f : failureList) {
			System.out.println(f.getMessage());
		}

	}
}
```

TestRunner.java

Explanation: When we run this TestRunner.java class, it will execute testGetConnection( ) method, in this we are checking weather it is giving same connection object or different connection object. As SingleConnectionProvider.java is not singleton class, always it will return new connection object hence our test case is going to failure. Output will be displayed like below in console.

```
<terminated> TestRunner [Java Application] C:\Program Files\Java\jre1.8.0_144\bin\javaw.exe (Oct 5, 2017, 10:37:49 PM)
Total cases executed : 1
Failure cases count : 1
Failure reasons are below...
expected same:<oracle.jdbc.driver.T4CConnection@6a5fc7f7> was not:<oracle.jdbc.driver.T4CConnection@3b6eb2ec>
```

Example 3: In this example we will test user login functionality with database.

Step1: In order to test our dao and service classes' login functionality, create a database table and insert some sample data using below queries.

```
--USER_MASTER TABLE CREATION
CREATE TABLE USER_MASTER (USER_ID NUMBER(10),USER_NAME VARCHAR2(50),USER_PWD VARCHAR2(50), primary key(user_id));

--INSERT SCRIPTS FOR STORING DATA
INSERT INTO USER_MASTER VALUES(101,'ashok','abc@123');
INSERT INTO USER_MASTER VALUES(102,'mahesh','mahi@123');
```

After executing above queries, our USER_MASTER table will have the data like below

	USER_ID	USER_NAME	USER_PWD
1	101	ashok	abc@123
2	102	mahesh	mahi@123

Step 2: Create UserDao.java class to execute database query like below

```
public class UserDao {
	private static final String DB_URL = "jdbc:oracle:thin:@localhost:1521/XE";
	private static final String DB_UNAME = "system";
	private static final String DB_PWD = "admin";
	private static final String DB_DRIVER_CLASS = "oracle.jdbc.driver.OracleDriver";
	/**
	 * This method is used to retrive User record using given
	 * username and  password
	 */
	 public UserModel loadByUsernameAndPwd(String uname, String pwd) throws SQLException {
		Connection con = null;ResultSet rs = null;
		PreparedStatement pstmt = null;UserModel model = null;
		try {
			Class.forName(DB_DRIVER_CLASS);
			con = DriverManager.getConnection(DB_URL, DB_UNAME, DB_PWD);
			String sql = "SELECT * FROM USER_MASTER WHERE USER_NAME=? AND USER_PWD=?";
			pstmt = con.prepareStatement(sql);
			rs = pstmt.executeQuery();
			if (rs.next()) {
				model = new UserModel();
				model.setUsername(rs.getString("USER_NAME"));
				model.setPassword(rs.getString("USER_PWD"));
			}
		} catch (Exception e) {
			e.printStackTrace();
		}
		return model;
	}
}
```

Note: As we are using Oracle database, need to add ojdbc14.jar/ojdbc6.jar in project build path if it is maven project add the ojdbc14 / ojdbc6 jar related dependency in pom.xml file along with junit dependency like below

```
<dependencies>
	<dependency>
		<groupId>junit</groupId>
		<artifactId>junit</artifactId>
		<version>4.12</version>
	</dependency>
	<dependency>
		<groupId>com.oracle</groupId>
		<artifactId>ojdbc14</artifactId>
		<version>1.1.1</version>
	</dependency>
</dependencies>
```

Step 3: Create LoginService.java: to call UserDao class method to load user record.

```java
public class LoginService {

	private UserDao userDao;

	public void setUserDao(UserDao userDao) {
		this.userDao = userDao;
	}

	public boolean login(String uname, String pwd) {
		UserModel model = null;
		try {
			model = userDao.loadByUsernameAndPwd(uname, pwd);
		} catch (SQLException e) {
			e.printStackTrace();
		}
		return null != model ? true : false;
	}
}
```

LoginService.java

Step 3: Create LoginServiceTest.java to test login( ) method functionality.

```java
public class TestLoginService {

	private LoginService service;

	@BeforeClass
	public void setUp() {
		service = new LoginService();
	}

	@Test
	public void testLogin() {
		boolean status = service.login("ashok", "abc@123");
		assertEquals(true, status);
	}
}
```

TestLoginService.java

Step 4 : Create TestRunner.java class like below to execute the test case and run this class.

```java
public class TestRunner {

	public static void main(String[] args) {

		Result result = JUnitCore.runClasses(TestLoginService.class);
		System.out.println("Total cases executed : " + result.getRunCount());
		System.out.println("Failure cases count : " + result.getFailureCount());
		System.out.println("Failure reasons are below...");
		List<Failure> failureList = result.getFailures();
		for (Failure f : failureList) {
			System.out.println(f.getMessage());
		}
	}
}
```

TestRunner.java

Using @ ignore annotation with Condition

Let's take the example of how to ignore a test and define the reason for ignoring along with it. As discussed above, to provide a reason you have one optional parameter in @ignore annotation where you can provide the reason statement.

```java
@Ignore("not yet ready , Please ignore.")
@Test
public void testJUnitMessage() {
   System.out.println("Junit Message is printing ");
   assertEquals(message, junitMessage.printMessage());
}
```

Mocking

As part of the application development, the code we write has a network of interdependencies, it may call into the methods of several other classes which in turn may call yet other methods; indeed this is the intent and power of object oriented programming. Usually at the same time as writing our feature code we will also write test code in the form of automated unit tests. We use these unit tests to verify the behavior of our code, to ensure that it behaves as we expect it to behave.

When we unit test our code we want to test it in isolation and we want to test it fast. For the purposes of the unit test we only care about verifying our own code, in the current class under test. Generally we also want to execute our unit tests very regularly, perhaps more than several times per hour when we are refactoring and we are working in our continuous integration environment.

This is when all our interdependencies become an issue. We might end up executing code in another class that has a bug that causes our unit test to fail. Imagine a class which we use to read user details from a database, what happens if there's no database present when we want to run our unit tests? Imagine a class which calls several remote web services, what if they're down or take a long time to respond? Our unit tests could fail due to our dependencies and not because of some issue with the behavior of our code. This is undesirable.

In addition to this, it might be very difficult to force a specific event or error condition that we want to ensure our code handles correctly. What if we want to test that some class which deserialize an object handles a possible ObjectStreamException properly? What if we want to test all boundary return values from a collaborator? What about ensuring that some calculated value is passed correctly to a collaborator? It might take a lot of coding and a long time to replicate the conditions for our tests, if it is even possible at all.

All these issues simply disappear if we use mocks. Mocks act like a substitute for the classes with which we are collaborating, they take their place and behave exactly how we tell them to behave. Mocks let us pretend that our real collaborators are there, even though they aren't. More importantly mocks can be programmed to return whatever values we want and confirm whatever values are passed to them. Mocks execute instantly and don't require any external resources. Mocks will return what we tell them to return, throw whatever exceptions we want them to throw and will do these things over and over, on demand. They let us test only the behavior of our own code, to ensure that our class works, regardless of the behavior of its collaborators.

There are several mocking frameworks available for Java, each have their own syntax, their own strengths, their own weaknesses.

Introduction to the EasyMock framework : EasyMock is a mocking framework, JAVA-based library that is used for effective unit testing of JAVA applications. EasyMock is used to mock interfaces so that a dummy functionality can be added to a mock interface that can be used in unit testing.

Before we begin, let's first understand the need behind using EasyMock. Let's say, you are building an enterprise application for maintaining user's stock portfolios. Your application would use a stock market service to retrieve stock prices from a real server (such as NASDAQ).

When it comes to testing your code, you wouldn't want to hit the real stock market server for fetching the stock prices. Instead, you would like some dummy price values. So, you need to mock the stock market service that returns dummy values without hitting the real server.

EasyMock is exactly doing the same - helps you to mock interfaces. You can pre-define the behavior of your mock objects and then use this mock object in your code for testing. Because, you are only concerned about testing

your logic and not the external services or objects. So, it makes sense mock the external services using Easy Mock.

To make it clear, have a look at the below code snippet

```
Line 1 :  StockMarket marketMock = EasyMock.createMock(StockMarket.class);
Line 2 :  EasyMock.expect(marketMock.getPrice("EBAY")).andReturn(42.00);
Line 3 :  EasyMock.replay(marketMock);
```

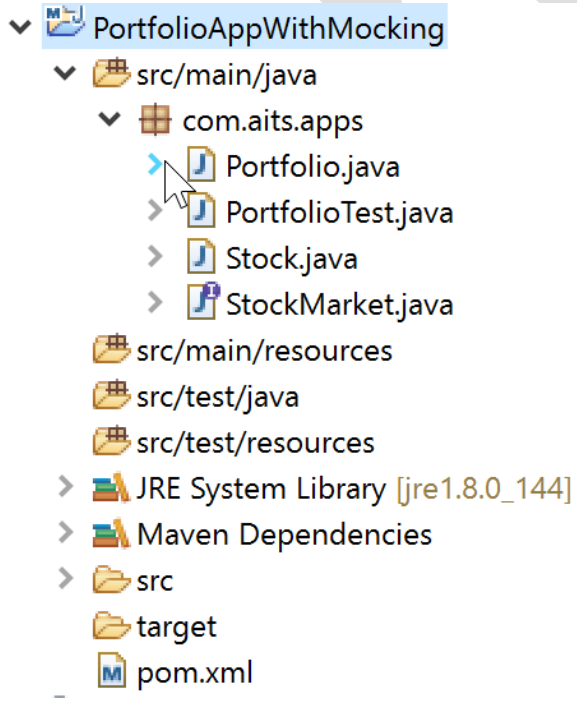
- In the first line, we ask the EasyMock to create a mock object for our StockMarket interface.
- And then in the second line, we define how this mock object should behave - i.e., when the getPrice() method is called with the parameter 'EBAY', the mock should return 42.00.
- And then, we call the replay() method, to make the mock object ready to use.

So, that pretty much set the context about the EasyMock and its usage. Let's dive into our Portfolio application.

Example: Let's develop our Portfolio application

Our Portfolio application is really simple. It has a Stock class to represent a stock name and quantity and the Portfolio class to hold a list of stocks. This Portfolio class has a method to calculate the total value of the portfolio. Our class uses a StockMarket (an interface) object to retrieve the stock prices. While testing our code, we will mock this StockMarket using EasyMock.

Step1: Create a maven project like below



Add below 2 dependencies in pom.xml file to work with Junit and EasyMock

```xml
<dependencies>
	<!-- Junit -->
	<dependency>
		<groupId>junit</groupId>
		<artifactId>junit</artifactId>
		<version>4.11</version>
	</dependency>

	<!-- Easy Mock -->
	<dependency>
		<groupId>org.easymock</groupId>
		<artifactId>easymock</artifactId>
		<version>3.1</version>
	</dependency>
</dependencies>
```

Step2: Create Stock.java: A very simple java object to represent single stock

```java
public class Stock {

	private String name;
	private int quantity;

	public Stock(String name, int quantity) {
		this.name = name;
		this.quantity = quantity;
	}

	//setters & getters
}
```

Stock.java

Step 3: create StockMarket.java : An interface to represent a stock market service. It has a method that returns the stock price of the given stock name.

```java
package com.aits.apps;

public interface StockMarket {
	public Double getPrice(String stockName);
}
```

StockMarket.java

Step 3: Create Portfolio.java : This object holds a list of Stock objects and a method to calculate the total value of the portfolio. It uses a StockMarket object to retrieve the stock prices. Since it is not a good practice to hard code the dependencies, we haven't initialized the stockMarket object. We'll inject it later using our test code.

```java
public class Portfolio {

	private String name;
	private StockMarket stockMarket;
	//setters && getters

	private List<Stock> stocks = new ArrayList<Stock>();
	/*
	 * this method gets the market value for each stock, sums it up and returns
	 * the total value of the portfolio.
	 */
	public Double getTotalValue() {
		Double value = 0.0;
		for (Stock stock : this.stocks) {
			value += (stockMarket.getPrice(stock.getName()) * stock.getQuantity());
		}
		return value;
	}
}
```

Portfolio.java

So, now we have coded the entire application. In this, we are going to test the portfolio.getTotalValue() method, because that's where our business logic is.

Testing Portfolio application using JUnit and EasyMock

```java
public class PortfolioTest  extends TestCase{

	private Portfolio portfolio;
	private StockMarket marketMock;

	@Before
	public void setUp() {
		portfolio = new Portfolio();
		portfolio.setName("Ashok's portfolio.");
		marketMock = EasyMock.createMock(StockMarket.class);
		portfolio.setStockMarket(marketMock);
	}
	@Test
	public void testGetTotalValue() {

		/* = Setup our mock object with the expected values */
		EasyMock.expect(marketMock.getPrice("EBAY")).andReturn(42.00);
		EasyMock.replay(marketMock);

		/* = Now start testing our portfolio */
		Stock ebayStock = new Stock("EBAY", 2);
		portfolio.addStock(ebayStock);
		assertEquals(84.00, portfolio.getTotalValue());
	}
}
```

PortfolioTest.java

As you can see, during setUp() we are creating new Portfolio object. Then we ask EasyMock to create a mock object for the StockMarket interface. Then we inject this mock object into our portfolio object using portfolio.setStockMarket() method.

In the @Test method, we define how our mock object should behave when called, using the below code:

EasyMock.expect(marketMock.getPrice("EBAY")).andReturn(42.00);

EasyMock.replay(marketMock);

So, here after our mock object's getPrice method would return 42.00 when called with EBAY.

Then we are creating a ebayStock with 2 quantities and add that to our portfolio. Since we setup the stock price of EBAY as 42.00, we know that the total value of our portfolio is 84.00 (i.e. 2 x 42.00). In the last line, we are asserting the same using the JUnit assertEquals() method.

The above test should run successfully if we haven't made any mistakes in the getTotalValue() code. Otherwise, the test would fail.

Conclusion

So, that's how we use the EasyMock library to mock the external services/objects and use them in our testing code.

===000===