Day 3	**JUnit** • JUnit overview • Types of testing **JUnit Framework** • JUnit Test Framework **Creating and Executing Unit Test** • Writing and Executing unit tests • Assertions • Exception Tests • Suite • Mock Objects	480 mins

JUnit Overview

Testing is the process of checking the functionality of an application to ensure it runs as per requirements. Unit testing comes into picture at the developers' level; it is the testing of single entity (class or method). Unit testing plays a critical role in helping a software company deliver quality products to its customers.

What is JUnit ?

JUnit is a unit testing framework for Java programming language. It plays a crucial role test-driven development, and is a family of unit testing frameworks collectively known as xUnit.

JUnit promotes the idea of "first testing then coding", which emphasizes on setting up the test data for a piece of code that can be tested first and then implemented. This approach is like "test a little, code a little, test a little, code a little." It increases the productivity of the programmer and the stability of program code, which in turn reduces the stress on the programmer and the time spent on debugging.

Features of JUnit

- JUnit is an open source framework, which is used for writing and running tests.
- Provides annotations to identify test methods.
- Provides assertions for testing expected results.
- Provides test runners for running tests.

- JUnit tests allow you to write codes faster, which increases quality.
- JUnit is elegantly simple. It is less complex and 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 the test is running smoothly, and it turns red when a test fails.

What is a Unit Test Case ?

A Unit Test Case is a part of code, which ensures that another part of code (method) works as expected. To achieve the desired results quickly, a test framework is required. JUnit is a perfect unit test framework for Java programming language.

A formal written unit test case is characterized by a known input and an expected output, which is worked out before the test is executed. The known input should test a precondition and the expected output should test a post-condition.

There must be at least two unit test cases for each requirement − one positive test and one negative test. If a requirement has sub-requirements, each sub-requirement must have at least two test cases as positive and negative.

Types of testing

Unit testing can be done in two ways − manual testing and automated testing.

Manual Testing	Automated Testing
Executing a test cases manually without any tool support is known as manual testing.	Taking tool support and executing the test cases by using an automation tool is known as automation testing.

Time-consuming and tedious − Since test cases are executed by human resources, it is very slow and tedious.	**Fast** − Automation runs test cases significantly faster than human resources.
Huge investment in human resources − As test cases need to be executed manually, more testers are required in manual testing.	**Less investment in human resources** − Test cases are executed using automation tools, so less number of testers are required in automation testing.
Less reliable − Manual testing is less reliable, as it has to account for human errors.	**More reliable** − Automation tests are precise and reliable.
Non-programmable − No programming can be done to write sophisticated tests to fetch hidden information.	**Programmable** − Testers can program sophisticated tests to bring out hidden information.

JUnit - Test Framework

JUnit is a **Regression Testing Framework** used by developers to implement unit testing in Java, and accelerate programming speed and increase the quality of code. JUnit Framework can be easily integrated with either of the following −

- Eclipse
- Ant
- Maven

Features of JUnit Test Framework

JUnit test framework provides the following important features −

- Fixtures
- Test suites
- Test runners
- JUnit classes

Fixtures

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. It includes −

- setUp() method, which runs before every test invocation.
- tearDown() method, which runs after every test method.

Let's check one example −

```
import junit.framework.*;


public class JavaTest extends TestCase {

   protected int value1, value2;


   // assigning the values

   protected void setUp(){

      value1 = 3;

      value2 = 3;

   }


   // test method to add two values

   public void testAdd(){

      double result = value1 + value2;

      assertTrue(result == 6);

   }

}
```

Test Suites

A test suite bundles a few unit test cases and runs them together. In JUnit, both @RunWith and @Suite annotation are used to run the suite test. Given below is an example that uses TestJunit1 & TestJunit2 test classes.

```
import org.junit.runner.RunWith;
```

```java
import org.junit.runners.Suite;


//JUnit Suite Test

@RunWith(Suite.class)


@Suite.SuiteClasses({

   TestJunit1.class ,TestJunit2.class

})


public class JunitTestSuite {

}

import org.junit.Test;

import org.junit.Ignore;

import static org.junit.Assert.assertEquals;


public class TestJunit1 {


   String message = "Robert";

   MessageUtil messageUtil = new MessageUtil(message);


   @Test

   public void testPrintMessage() {

      System.out.println("Inside testPrintMessage()");

      assertEquals(message, messageUtil.printMessage());

   }

}

import org.junit.Test;

import org.junit.Ignore;

import static org.junit.Assert.assertEquals;
```

```
public class TestJunit2 {


   String message = "Robert";
   MessageUtil messageUtil = new MessageUtil(message);


   @Test
   public void testSalutationMessage() {
      System.out.println("Inside testSalutationMessage()");
      message = "Hi!" + "Robert";
      assertEquals(message,messageUtil.salutationMessage());
   }
}
```

Test Runners

Test runner is used for executing the test cases. Here is an example that assumes the test class **TestJunit** already exists.

```
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(TestJunit.class);


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


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

JUnit Classes

JUnit classes are important classes, used in writing and testing JUnits. Some of the important classes are −

- **Assert** − Contains a set of assert methods.
- **TestCase** − Contains a test case that defines the fixture to run multiple tests.
- **TestResult** − Contains methods to collect the results of executing a test case.

Writing and Executing unit tests

Here we will see one complete example of JUnit testing using POJO class, Business logic class, and a test class, which will be run by the test runner.

Create **EmployeeDetails.java** in C:\>JUNIT_WORKSPACE, which is a POJO class.

```
public class EmployeeDetails {


   private String name;

   private double monthlySalary;

   private int age;


   /**

   * @return the name

   */


   public String getName() {

      return name;

   }

```

```
/**
* @param name the name to set
*/

public void setName(String name) {
   this.name = name;
}

/**
* @return the monthlySalary
*/

public double getMonthlySalary() {
   return monthlySalary;
}

/**
* @param monthlySalary the monthlySalary to set
*/

public void setMonthlySalary(double monthlySalary) {
   this.monthlySalary = monthlySalary;
}

/**
* @return the age
*/
public int getAge() {
   return age;
}
```

```
   /**
   * @param age the age to set
   */
   public void setAge(int age) {
      this.age = age;
   }
}
```

EmployeeDetails class is used to −

- get/set the value of employee's name.
- get/set the value of employee's monthly salary.
- get/set the value of employee's age.

Create a file called **EmpBusinessLogic.java** in C:\>JUNIT_WORKSPACE, which contains the business logic.

```
public class EmpBusinessLogic {
   // Calculate the yearly salary of employee
   public double calculateYearlySalary(EmployeeDetails employeeDetails) {
      double yearlySalary = 0;
      yearlySalary = employeeDetails.getMonthlySalary() * 12;
      return yearlySalary;
   }

   // Calculate the appraisal amount of employee
   public double calculateAppraisal(EmployeeDetails employeeDetails) {
      double appraisal = 0;

      if(employeeDetails.getMonthlySalary() < 10000){
         appraisal = 500;
      }else{
         appraisal = 1000;
      }

      return appraisal;
   }
}
```

EmpBusinessLogic class is used for calculating –

- the yearly salary of an employee.
- the appraisal amount of an employee.

Create a file called **TestEmployeeDetails.java** in C:\>JUNIT_WORKSPACE, which contains the test cases to be tested.

```
import org.junit.Test;

import static org.junit.Assert.assertEquals;


public class TestEmployeeDetails {

   EmpBusinessLogic empBusinessLogic = new EmpBusinessLogic();

   EmployeeDetails employee = new EmployeeDetails();


   //test to check appraisal

   @Test

   public void testCalculateAppriasal() {

      employee.setName("Rajeev");

      employee.setAge(25);

      employee.setMonthlySalary(8000);


      double appraisal = empBusinessLogic.calculateAppraisal(employee);

      assertEquals(500, appraisal, 0.0);

   }


   // test to check yearly salary
   @Test
   public void testCalculateYearlySalary() {
      employee.setName("Rajeev");
      employee.setAge(25);
      employee.setMonthlySalary(8000);

      double salary = empBusinessLogic.calculateYearlySalary(employee);
      assertEquals(96000, salary, 0.0);
   }
}
```

TestEmployeeDetails class is used for testing the methods of **EmpBusinessLogic** class. It

- tests the yearly salary of the employee.
- tests the appraisal amount of the employee.

Next, create a java class filed named **TestRunner.java** in C:\>JUNIT_WORKSPACE to execute test case(s).

```
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(TestEmployeeDetails.class);

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

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

(we can use eclipse to directly run junit test case)

Compile the test case and Test Runner classes using javac.

```
C:\JUNIT_WORKSPACE>javac EmployeeDetails.java
EmpBusinessLogic.java TestEmployeeDetails.java TestRunner.java
```

Now run the Test Runner, which will run the test case defined in the provided Test Case class.

```
C:\JUNIT_WORKSPACE>java TestRunner
```

Verify the output.

```
true
```

Assertions

All the assertions are in the Assert class.

```
public class Assert extends java.lang.Object
```

This class provides a set of assertion methods, useful for writing tests. Only failed assertions are recorded. Some of the important methods of Assert class are as follows −

Sr.No.	Methods & Description
1	**void assertEquals(boolean expected, boolean actual)** Checks that two primitives/objects are equal.
2	**void assertTrue(boolean condition)** Checks that a condition is true.
3	**void assertFalse(boolean condition)** Checks that a condition is false.
4	**void assertNotNull(Object object)** Checks that an object isn't null.

5	**void assertNull(Object object)** Checks that an object is null.
6	**void assertSame(object1, object2)** The assertSame() method tests if two object references point to the same object.
7	**void assertNotSame(object1, object2)** The assertNotSame() method tests if two object references do not point to the same object.
8	**void assertArrayEquals(expectedArray, resultArray);** The assertArrayEquals() method will test whether two arrays are equal to each other.

Let's use some of the above-mentioned methods in an example. Create a java class file named **TestAssertions.java** in C:\>JUNIT_WORKSPACE.

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

public class TestAssertions {

   @Test
   public void testAssertions() {
      //test data
      String str1 = new String ("abc");
      String str2 = new String ("abc");
      String str3 = null;
      String str4 = "abc";
      String str5 = "abc";

      int val1 = 5;
```

```java
        int val2 = 6;

        String[] expectedArray = {"one", "two", "three"};
        String[] resultArray =  {"one", "two", "three"};

        //Check that two objects are equal
        assertEquals(str1, str2);

        //Check that a condition is true
        assertTrue (val1 < val2);

        //Check that a condition is false
        assertFalse(val1 > val2);

        //Check that an object isn't null
        assertNotNull(str1);

        //Check that an object is null
        assertNull(str3);

        //Check if two object references point to the same object
        assertSame(str4,str5);

        //Check if two object references not point to the same object
        assertNotSame(str1,str3);

        //Check whether two arrays are equal to each other.
        assertArrayEquals(expectedArray, resultArray);
    }
}
```

Annotation

Annotations are like meta-tags that you can add to your code, and apply them to methods or in class. These annotations in JUnit provide the following information about test methods −

- which methods are going to run before and after test methods.
- which methods run before and after all the methods, and.
- which methods or classes will be ignored during the execution.

The following table provides a list of annotations and their meaning in JUnit −

Sr.No.	Annotation & Description
1	**@Test** The Test annotation tells JUnit that the public void method to which it is attached can be run as a test case.
2	**@Before** Several tests need similar objects created before they can run. Annotating a public void method with @Before causes that method to be run before each Test method.
3	**@After** If you allocate external resources in a Before method, you need to release them after the test runs. Annotating a public void method with @After causes that method to be run after the Test method.
4	**@BeforeClass** Annotating a public static void method with @BeforeClass causes it to be run once before any of the test methods in the class.
5	**@AfterClass** This will perform the method after all tests have finished. This can be used to perform clean-up activities.

6	**@Ignore** The Ignore annotation is used to ignore the test and that test will not be executed.

Create a java class file named **JunitAnnotation.java** in C:\>JUNIT_WORKSPACE to test annotation.

```
import org.junit.After;

import org.junit.AfterClass;


import org.junit.Before;

import org.junit.BeforeClass;


import org.junit.Ignore;

import org.junit.Test;


public class JunitAnnotation {


   //execute before class

   @BeforeClass

   public static void beforeClass() {

      System.out.println("in before class");

   }


   //execute after class

   @AfterClass

   public static void  afterClass() {

      System.out.println("in after class");

   }


   //execute before test
   @Before

   public void before() {

      System.out.println("in before");

   }


   //execute after test

   @After

   public void after() {

      System.out.println("in after");

   }


   //test case

   @Test

   public void test() {

      System.out.println("in test");

   }


   //test case ignore and will not execute

   @Ignore

   public void ignoreTest() {

      System.out.println("in ignore test");

   }

}
```

Next, create a java class file named **TestRunner.java** in C:\>JUNIT_WORKSPACE to execute annotations.

```
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(JunitAnnotation.class);

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

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

JUnit - Parameterized Test

JUnit 4 has introduced a new feature called **parameterized tests**. Parameterized tests allow a developer to run the same test over and over again using different values. There are five steps that you need to follow to create a parameterized test.

- Annotate test class with @RunWith(Parameterized.class).
- Create a public static method annotated with @Parameters that returns a Collection of Objects (as Array) as test data set.
- Create a public constructor that takes in what is equivalent to one "row" of test data.
- Create an instance variable for each "column" of test data.
- Create your test case(s) using the instance variables as the source of the test data.

The test case will be invoked once for each row of data. Let us see parameterized tests in action.

Create a Class

Create a java class to be tested, say, **PrimeNumberChecker.java** in C:\>JUNIT_WORKSPACE.

```
public class PrimeNumberChecker {
   public Boolean validate(final Integer primeNumber) {
      for (int i = 2; i < (primeNumber / 2); i++) {
         if (primeNumber % i == 0) {
            return false;
         }
      }
      return true;
   }
}
```

Create Parameterized Test Case Class

Create a java test class, say, **PrimeNumberCheckerTest.java**. Create a java class file named **PrimeNumberCheckerTest.java** in C:\>JUNIT_WORKSPACE.

```
import java.util.Arrays;
import java.util.Collection;


import org.junit.Test;
import org.junit.Before;


import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;
import org.junit.runner.RunWith;
import static org.junit.Assert.assertEquals;


@RunWith(Parameterized.class)
public class PrimeNumberCheckerTest {
   private Integer inputNumber;
   private Boolean expectedResult;
   private PrimeNumberChecker primeNumberChecker;


   @Before
```

```
public void initialize() {
   primeNumberChecker = new PrimeNumberChecker();
}

// Each parameter should be placed as an argument here
// Every time runner triggers, it will pass the arguments
// from parameters we defined in primeNumbers() method

public PrimeNumberCheckerTest(Integer inputNumber, Boolean expectedResult) {
   this.inputNumber = inputNumber;
   this.expectedResult = expectedResult;
}

@Parameterized.Parameters
public static Collection primeNumbers() {
   return Arrays.asList(new Object[][] {
      { 2, true },
      { 6, false },
      { 19, true },
      { 22, false },
      { 23, true }
   });
}

// This test will run 4 times since we have 5 parameters defined
@Test
public void testPrimeNumberChecker() {
   System.out.println("Parameterized Number is : " + inputNumber);
   assertEquals(expectedResult,
   primeNumberChecker.validate(inputNumber));
   }
}
```

Create Test Runner Class

Create a java class file named **TestRunner.java** in C:\>JUNIT_WORKSPACE to execute test case(s).

```
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(PrimeNumberCheckerTest.class);


      for (Failure failure : result.getFailures()) {

         System.out.println(failure.toString());

      }


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

   }

}
```

Compile the PrimeNumberChecker, PrimeNumberCheckerTest and Test Runner classes using javac.

```
C:\JUNIT_WORKSPACE>javac PrimeNumberChecker.java PrimeNumberCheckerTest.java
TestRunner.java
```

Now run the Test Runner, which will run the test cases defined in the provided Test Case class.

```
C:\JUNIT_WORKSPACE>java TestRunner
```

Verify the output.

```
Parameterized Number is : 2
Parameterized Number is : 6
Parameterized Number is : 19
Parameterized Number is : 22
Parameterized Number is : 23
```

```
true
```

JUnit - Time Test

JUnit provides a handy option of Timeout. If a test case takes more time than the specified number of milliseconds, then JUnit will automatically mark it as failed. The **timeout** parameter is used along with @Test annotation. Let us see the @Test(timeout) in action.

Create a Class

Create a java class to be tested, say, **MessageUtil.java** in C:\>JUNIT_WORKSPACE.

Add an infinite while loop inside the printMessage() method.

```
/*

* This class prints the given message on console.

*/


public class MessageUtil {


   private String message;


   //Constructor

   //@param message to be printed

   public MessageUtil(String message){

      this.message = message;

   }


   // prints the message

   public void printMessage(){

      System.out.println(message);

      while(true);

   }
   // add "Hi!" to the message
   public String salutationMessage(){
      message = "Hi!" + message;
      System.out.println(message);
      return message;
   }
}
```

Create Test Case Class

Create a java test class, say, **TestJunit.java**. Add a timeout of 1000 to testPrintMessage() test case.

Create a java class file named **TestJunit.java** in C:\>JUNIT_WORKSPACE.

```
import org.junit.Test;
import org.junit.Ignore;
import static org.junit.Assert.assertEquals;

public class TestJunit {

   String message = "Robert";
   MessageUtil messageUtil = new MessageUtil(message);

   @Test(timeout = 1000)
   public void testPrintMessage() {
      System.out.println("Inside testPrintMessage()");
      messageUtil.printMessage();
   }

   @Test
   public void testSalutationMessage() {
      System.out.println("Inside testSalutationMessage()");
      message = "Hi!" + "Robert";

      assertEquals(message,messageUtil.salutationMessage());

   }

}
```

Create Test Runner Class

Create a java class file named **TestRunner.java** in C:\>JUNIT_WORKSPACE to execute test case(s).

```
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(TestJunit.class);


      for (Failure failure : result.getFailures()) {

         System.out.println(failure.toString());

      }


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

   }

}
```

Compile the MessageUtil, Test case and Test Runner classes using javac.

```
C:\JUNIT_WORKSPACE>javac MessageUtil.java TestJunit.java TestRunner.java
```

Now run the Test Runner, which will run the test cases defined in the provided Test Case class.

```
C:\JUNIT_WORKSPACE>java TestRunner
```

Verify the output. testPrintMessage() test case will mark the unit testing failed.

```
Inside testPrintMessage()
```

```
Robert
Inside testSalutationMessage()
Hi!Robert
testPrintMessage(TestJunit): test timed out after 1000 milliseconds
false
```

Exception Tests

The test cases are executed using **JUnitCore** class. JUnitCore is a facade for running tests. It supports running JUnit 4 tests, JUnit 3.8.x tests, and mixtures. To run tests from the command line, run java org.junit.runner.JUnitCore <TestClass>. For one-shot test runs, use the static method runClasses(Class[]).

Following is the declaration for **org.junit.runner.JUnitCore** class:

```
public class JUnitCore extends java.lang.Object
```

Here we will see how to execute the tests with the help of JUnitCore.

Create a Class

Create a java class to be tested, say, **MessageUtil.java**, in C:\>JUNIT_WORKSPACE.

```
/*

* This class prints the given message on console.

*/


public class MessageUtil {


   private String message;


   //Constructor

   //@param message to be printed

   public MessageUtil(String message){

      this.message = message;

   }
   // prints the message

   public String printMessage(){

      System.out.println(message);

      return message;

   }


}
```

Create Test Case Class

- Create a java test class, say, TestJunit.java.
- Add a test method testPrintMessage() to your test class.
- Add an Annotaion @Test to the method testPrintMessage().
- Implement the test condition and check the condition using assertEquals API of JUnit.

Create a java class file named **TestJunit.java** in C:\>JUNIT_WORKSPACE.

```
import org.junit.Test;

import static org.junit.Assert.assertEquals;


public class TestJunit {


   String message = "Hello World";

   MessageUtil messageUtil = new MessageUtil(message);


   @Test

   public void testPrintMessage() {

      assertEquals(message,messageUtil.printMessage());

   }

}
```

Create Test Runner Class

Now create a java class file named **TestRunner.java** in C:\>JUNIT_WORKSPACE to execute test case(s). It imports the JUnitCore class and uses the runClasses() method that takes the test class name as its parameter.

```
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(TestJunit.class);


      for (Failure failure : result.getFailures()) {

         System.out.println(failure.toString());

      }


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

   }

}
```

Suite

Create JUnit Test Suite with Example: @RunWith @SuiteClasses

In Junit, test suite allows us to aggregate all test cases from multiple classes in one place and run it together.

To run the suite test, you need to annotate a class using below-mentioned annotations:

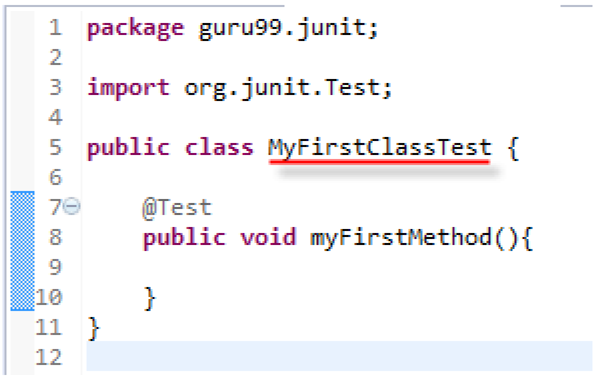
1. @Runwith(Suite.class)
2. @SuiteClasses(test1.class,test2.class……) or

   @Suite.SuiteClasses ({test1.class, test2.class……})

With above annotations, all the test classes in the suite will start executing one by one.

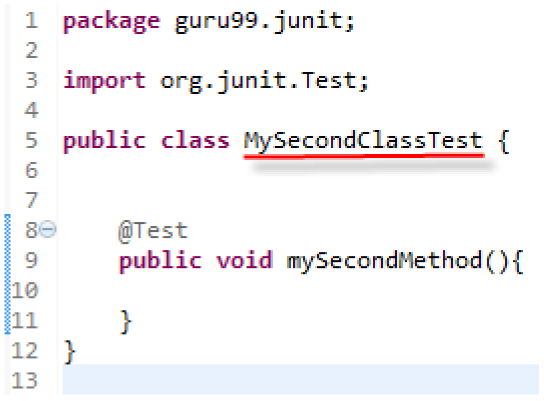
Steps to create Test Suite and Test Runner

Step 1) Create a simple test class (e.g. MyFirstClassTest) and add a method annotated with @test.



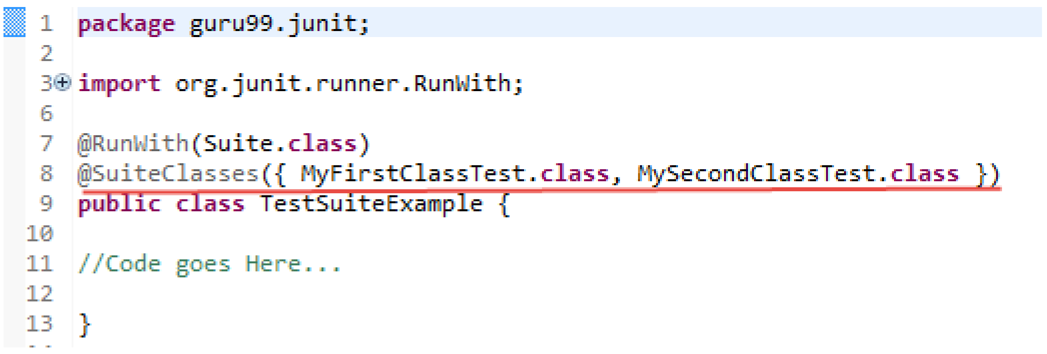
```java
package guru99.junit;

import org.junit.Test;

public class MyFirstClassTest {

    @Test
    public void myFirstMethod(){

    }
}

```

Step 2) Create another test class to add (e.g. MySecondClassTest) and create a method annotated with @test.



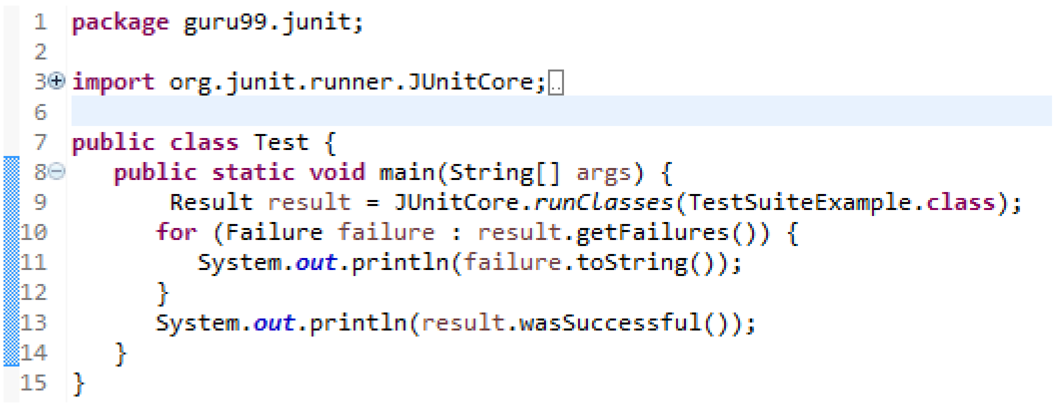
```java
package guru99.junit;

import org.junit.Test;

public class MySecondClassTest {


    @Test
    public void mySecondMethod(){

    }
}

```

Step 3) To create a testSuite you need to first annotate the class with @RunWith(Suite.class) and @SuiteClasses(class1.class2…..).



```java
package guru99.junit;

import org.junit.runner.RunWith;

@RunWith(Suite.class)
@SuiteClasses({ MyFirstClassTest.class, MySecondClassTest.class })
public class TestSuiteExample {

//Code goes Here...

}
```

Step 4) Create a Test Runner class to run our test suite as given below;



```java
 1  package guru99.junit;
 2
 3  import org.junit.runner.JUnitCore;
 6
 7  public class Test {
 8      public static void main(String[] args) {
 9          Result result = JUnitCore.runClasses(TestSuiteExample.class);
10          for (Failure failure : result.getFailures()) {
11              System.out.println(failure.toString());
12          }
13          System.out.println(result.wasSuccessful());
14      }
15  }
```

Code Explanation:

- **Code Line 8:** Declaring the main method of the class test which will run our JUnit test.
- **Code Line 9:** Executing test cases using JunitCore.runclasses which takes the test class name as a parameter (In the example above, you are using TestSuiteExample.class shown in step 3).
- **Code Line 11:** Processing the result using for loop and printing out failed result.
- **Code Line 13:** Printing out the successful result.

JUnit Test Suite Example

Consider a more complex example

JunitTest.java

JunitTest.java is a simple class annotated with **@RunWith** and **@Suite** annotations. You can list out number of .classes in the suite as parameters as given below:

```java
package com.smita.junit;
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
```

```
@RunWith(Suite.class)
@Suite.SuiteClasses({
  SuiteTest1.class,
  SuiteTest2.class,
})

public class JunitTest {
                         // This class remains empty, it is used only as a holder
for the above annotations
}
```

SuiteTest1.java

SuiteTest1.java is a test class having a test method to print out a message as given below. You will use this class as a suite in above mentioned class.

```
package com.smita.junit;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

public class SuiteTest1 {

    public String message = "Saurabh";


    JUnitMessage junitMessage = new JUnitMessage(message);


    @Test(expected = ArithmeticException.class)

    public void testJUnitMessage() {

        System.out.println("Junit Message is printing ");

        junitMessage.printMessage();

    }

    @Test
    public void testJUnitHiMessage() {
        message = "Hi!" + message;
        System.out.println("Junit Hi Message is printing ");

        assertEquals(message, junitMessage.printHiMessage());

        System.out.println("Suite Test 2 is successful " + message);

    }
}
```

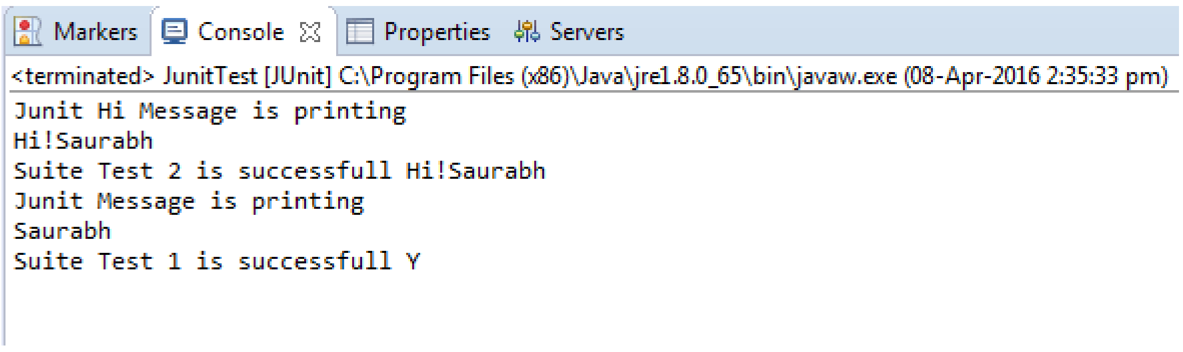
SuiteTest2.java

SuiteTest2.java is another test class similar to **SuiteTest1.java** having a test method to print out a message as given below. You will use this class as suite in **JunitTest.java**.

```
package com.smita.junit;

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

public class SuiteTest2 {


    @Test
    public void createAndSetName() {


        String expected = "Y";
        String actual = "Y";

        Assert.assertEquals(expected, actual);

        System.out.println("Suite Test 1 is successful " + actual);

    }

}
```

Output

After executing **JunitTest**.java which contains a suite having **test1.java** and **test2.java**, you will get below output:



```
<terminated> JunitTest [JUnit] C:\Program Files (x86)\Java\jre1.8.0_65\bin\javaw.exe (08-Apr-2016 2:35:33 pm)
Junit Hi Message is printing
Hi!Saurabh
Suite Test 2 is successfull Hi!Saurabh
Junit Message is printing
Saurabh
Suite Test 1 is successfull Y
```

Mock Objects- **Unit tests with Mockito**

1. Junit Project

A unit test should test functionality in isolation. Side effects from other classes or the system should be eliminated for a unit test, if possible.

This can be done via using test replacements (*test doubles*) for the real dependencies. Test doubles can be classified like the following:

- A *dummy object* is passed around but never used, i.e., its methods are never called. Such an object can for example be used to fill the parameter list of a method.
- *Fake* objects have working implementations, but are usually simplified. For example, they use an in memory database and not a real database.
- A *stub* class is an partial implementation for an interface or class with the purpose of using an instance of this stub class during testing. Stubs usually don't respond to anything outside what's programmed in for the test. Stubs may also record information about calls.
- A *mock object* is a dummy implementation for an interface or a class in which you define the output of certain method calls. Mock objects are configured to perform a certain behavior during a test. They typically record the interaction with the system and tests can validate that.

Test doubles can be passed to other objects which are tested. Your tests can validate that the class reacts correctly during tests. For example, you can validate if certain methods on the mock object were called. This helps to ensure that you only test the class while running tests and that your tests are not affected by any side effects.

Mock objects typically require less code to configure and should therefore be preferred.

2. Mock object generation

You can create mock objects manually (via code) or use a mock framework to simulate these classes. Mock frameworks allow you to create mock objects at runtime and define their behavior.

The classical example for a mock object is a data provider. In production an implementation to connect to the real data source is used. But for testing a mock object simulates the data source and ensures that the test conditions are always the same.

These mock objects can be provided to the class which is tested. Therefore, the class to be tested should avoid any hard dependency on external data.

Mocking or mock frameworks allows testing the expected interaction with the mock object. You can, for example, validate that only certain methods have been called on the mock object.

3. Using Mockito for mocking objects

Mockito is a popular mock framework which can be used in conjunction with JUnit. Mockito allows you to create and configure mock objects. Using Mockito simplifies the development of tests for classes with external dependencies significantly.

If you use Mockito in tests you typically:

- Mock away external dependencies and insert the mocks into the code under test
- Execute the code under test
- Validate that the code executed correctly

4. Adding Mockito as dependencies to a project

In simple, mocking is creating objects that mimic the behavior of real objects.
Many mocking tools are available
We will be using EasyMock.

What is Mocking?

Mocking is a way to test the functionality of a class in isolation. Mocking does not require a database connection or properties file read or file server read to test a functionality. Mock objects do the mocking of the real service. A mock object returns a dummy data corresponding to some dummy input passed to it.

EasyMock

EasyMock facilitates creating mock objects seamlessly. It uses Java Reflection in order to create mock objects for a given interface. Mock objects are nothing but proxy for actual implementations. Consider a case of Stock Service which returns the price details of a stock. During development, the actual stock service cannot be used to get real-time data. So we need a dummy implementation of the stock service. EasyMock can do the same very easily as its name suggests.

Benefits of EasyMock

- **No Handwriting** – No need to write mock objects on your own.
- **Refactoring Safe** – Renaming interface method names or reordering parameters will not break the test code as Mocks are created at runtime.
- **Return value support** – Supports return values.
- **Exception support** – Supports exceptions.
- **Order check support** – Supports check on order of method calls.
- **Annotation support** – Supports creating mocks using annotation.
- **Download EasyMock Archive**

To download the latest version of EasyMock click here.

Or Maven Dependencies

Before we dive in, let's add the following dependency to our *pom.xml*:

```
<dependency>
    <groupId>org.easymock</groupId>
    <artifactId>easymock</artifactId>
    <version>3.5.1</version>
    <scope>test</scope>
</dependency>
```

The latest version can always be found here.

Core Concepts

When generating a mock, **we can simulate the target object, specify its behavior, and finally verify whether it's used as expected.**

Working with EasyMock's mocks involves four steps:

1. creating a mock of the target class
2. recording its expected behavior, including the action, result, exceptions, etc.
3. using mocks in tests
4. verifying if it's behaving as expected

After our recording finishes, we switch it to "replay" mode, so that the mock behaves as recorded when collaborating with any object that will be using it.

Eventually, we verify if everything goes as expected.

The four steps mentioned above relate to methods in *org.easymock.EasyMock*:

1. ***mock(…)***: generates a mock of the target class, be it a concrete class or an interface. Once created, a mock is in "recording" mode, meaning that EasyMock will record any action the Mock Object takes, and replay them in the "replay" mode
2. ***expect(…)***: with this method, we can set expectations, including calls, results, and exceptions, for associated recording actions
3. ***replay(…)***: switches a given mock to "replay" mode. Then, any action triggering previously recorded method calls will replay "recorded results"
4. ***verify(…)***: verifies that all expectations were met and that no unexpected call was performed on a mock.

Lets Integrate JUnit and EasyMock together. Here we will create a Math Application which uses CalculatorService to perform basic mathematical operations such as addition, subtraction, multiply, and division. We'll use EasyMock to mock the dummy implementation of CalculatorService. In addition, we've made extensive use of annotations to showcase their compatibility with both JUnit and EasyMock.

The process is discussed below in a step-by-step manner.

Step 1: Create an interface called CalculatorService to provide mathematical functions

File: CalculatorService.java

```
public interface CalculatorService {

   public double add(double input1, double input2);

   public double subtract(double input1, double input2);

   public double multiply(double input1, double input2);

   public double divide(double input1, double input2);

}
```

Step 2: Create a JAVA class to represent MathApplication

File: MathApplication.java

```
public class MathApplication {

   private CalculatorService calcService;


   public void setCalculatorService(CalculatorService calcService){

      this.calcService = calcService;

   }


   public double add(double input1, double input2){

      return calcService.add(input1, input2);

   }


   public double subtract(double input1, double input2){

      return calcService.subtract(input1, input2);

   }


   public double multiply(double input1, double input2){

      return calcService.multiply(input1, input2);

   }


   public double divide(double input1, double input2){
      return calcService.divide(input1, input2);

   }

}
```

Step 3: Test the MathApplication class

Let's test the MathApplication class, by injecting in it a mock of calculatorService. Mock will be created by EasyMock.

File: MathApplicationTester.java

```
import org.easymock.EasyMock;

import org.easymock.EasyMockRunner;

import org.easymock.Mock;

import org.easymock.TestSubject;


import org.junit.Assert;

import org.junit.Before;

import org.junit.Test;

import org.junit.runner.RunWith;


// @RunWith attaches a runner with the test class to initialize the test data

@RunWith(EasyMockRunner.class)

public class MathApplicationTester {


   // @TestSubject annotation is used to identify class which is going to use
the mock object

   @TestSubject

   MathApplication mathApplication = new MathApplication();


   //@Mock annotation is used to create the mock object to be injected

   @Mock

   CalculatorService calcService;

   @Test

   public void testAdd(){

      //add the behavior of calc service to add two numbers

      EasyMock.expect(calcService.add(10.0,20.0)).andReturn(30.00);


      //activate the mock

      EasyMock.replay(calcService);


      //test the add functionality

      Assert.assertEquals(mathApplication.add(10.0, 20.0),30.0,0);

   }

}
```

Step 4: Create a class to execute to test cases

Create a java class file named TestRunner in **C:\> EasyMock_WORKSPACE**to execute Test case(s).

File: TestRunner.java

```
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(MathApplicationTester.class);


      for (Failure failure : result.getFailures()) {

         System.out.println(failure.toString());

      }


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

   }
}
```

Another **Java Examples**

A simple Author and Book example.

1.1 `BookService` to return a list of books by author name.

BookService.java

```java
package com.smita.examples.mock;

import java.util.List;

public interface BookService {

    List<Book> findBookByAuthor(String author);

}
```

Copy

BookServiceImpl.java

```java
package com.smita.examples.mock;

import java.util.List;

public class BookServiceImpl implements BookService {

    private BookDao bookDao;

    public BookDao getBookDao() {
        return bookDao;
    }

    public void setBookDao(BookDao bookDao) {
        this.bookDao = bookDao;
    }

    @Override
    public List<Book> findBookByAuthor(String name) {
        return bookDao.findBookByAuthor(name);
    }

}
```

Copy

BookDao.java

```java
package com.smita.examples.mock;

import java.util.List;

public interface BookDao {

    List<Book> findBookByAuthor(String author);

}
```

Copy

BookDaoImpl.java

```java
package com.smita.examples.mock;
```

```java
import java.util.List;

public class BookDaoImpl implements BookDao {

    @Override
    public List<Book> findBookByAuthor(String name) {
                 // init database
        // Connect to DB for data
        // return data
    }

}
```
Copy

1.2 A book validator.

BookValidatorService.java

```java
package com.smita.examples.mock;

public interface BookValidatorService {

    boolean isValid(Book book);

}
```
Copy

FakeBookValidatorService.java

```java
package com.smita.examples.mock;

public class FakeBookValidatorService implements BookValidatorService {

    @Override
    public boolean isValid(Book book) {
        if (book == null)
            return false;

        if ("bot".equals(book.getName())) {
            return false;
        } else {
            return true;
        }

    }
}
```
Copy

1.3 Reviews the `AuthorServiceImpl`, it has dependencies on `BookService` (depends on `BookDao`) and `BookValidatorService`, it makes the unit test a bit hard to write.

AuthorService.java

```java
package com.smita.examples.mock;

public interface AuthorService {

    int getTotalBooks(String author);

}
```
Copy

AuthorServiceImpl.java

```
package com.smita.examples.mock;

import java.util.List;
import java.util.stream.Collectors;

public class AuthorServiceImpl implements AuthorService {

    private BookService bookService;
    private BookValidatorService bookValidatorService;

    public BookValidatorService getBookValidatorService() {
        return bookValidatorService;
    }

    public void setBookValidatorService(BookValidatorService bookValidatorService)
{
        this.bookValidatorService = bookValidatorService;
    }

    public BookService getBookService() {
        return bookService;
    }

    public void setBookService(BookService bookService) {
        this.bookService = bookService;
    }

        //How to test this method ???
    @Override
    public int getTotalBooks(String author) {

        List<Book> books = bookService.findBookByAuthor(author);

        //filters some bot writers
        List<Book> filtered = books.stream().filter(
                x -> bookValidatorService.isValid(x))
                .collect(Collectors.toList());

        //other business logic

        return filtered.size();


    }
}
```
Copy

2. Unit Test

Create a unit test for `AuthorServiceImpl.getTotalBooks()`

2.1 The `AuthorServiceImpl` has two dependencies, you need to make sure both are configured properly.

AuthorServiceTest.java

```
package com.smita.mock;
```

```
import com.smita.examples.mock.AuthorServiceImpl;
import com.smita.examples.mock.BookDaoImpl;
import com.smita.examples.mock.BookServiceImpl;
import com.smita.examples.mock.FakeBookValidatorService;
import org.junit.Test;

import static org.hamcrest.CoreMatchers.is;
import static org.junit.Assert.assertThat;

public class AuthorServiceTest {

    @Test
    public void test_total_book_by_mock() {

                //1. Setup
        AuthorServiceImpl obj = new AuthorServiceImpl();
        BookServiceImpl bookService = new BookServiceImpl();
        bookService.setBookDao(new BookDaoImpl()); //Where Dao connect to?
        obj.setBookService(bookService);
        obj.setBookValidatorService(new FakeBookValidatorService());

                //2. Test method
        int qty = obj.getTotalBooks("smita");

                //3. Verify result
        assertThat(qty, is(2));

    }

}
```
Copy

To pass above unit test, you need to setup a database in DAO layer, else `bookService` will return nothing.

2.3 Some disadvantages to perform tests like above :

1. This unit test is slow, because you need to start a database in order to get data from DAO.
2. This unit test is not isolated, it always depends on external resources like database.
3. This unit test can't ensures the test condition is always the same, the data in the database may vary in time.
4. It's too much work to test a simple method, cause developers skipping the test.

2.4 **Solution**
The solution is obvious, you need a modified version of the `BookServiceImpl` class – which will always return the same data for testing, a **mock object**!

What is mocking?
Again, mocking is creating objects that mimic the behavior of real objects.

3. Unit Test – Mock Object

3.1 Create a new `MockBookServiceImpl` class and always return the same collection of books for author "smita".

MockBookServiceImpl.java

```
package com.smita.mock;

import com.smita.examples.mock.Book;
import com.smita.examples.mock.BookService;

import java.util.ArrayList;
import java.util.List;

//I am a mock object!
public class MockBookServiceImpl implements BookService {

    @Override
    public List<Book> findBookByAuthor(String author) {
        List<Book> books = new ArrayList<>();

        if ("smita".equals(author)) {
            books.add(new Book("smita in action"));
            books.add(new Book("abc in action"));
            books.add(new Book("bot"));
        }

        return books;
    }

    //implements other methods...

}
```

Copy

3.2 Update the unit test again.

AuthorServiceTest.java

```
package com.smita.mock;

import com.smita.examples.mock.AuthorServiceImpl;
import com.smita.examples.mock.FakeBookValidatorService;
import org.junit.Test;

import static org.hamcrest.CoreMatchers.is;
import static org.junit.Assert.assertThat;

public class AuthorServiceTest {

    @Test
    public void test_total_book_by_mock() {

                //1. Setup
        AuthorServiceImpl obj = new AuthorServiceImpl();

        /*BookServiceImpl bookService = new BookServiceImpl();
        bookService.setBookDao(new BookDaoImpl());
        obj.setBookService(bookService);*/
        obj.setBookService(new MockBookServiceImpl());
        obj.setBookValidatorService(new FakeBookValidatorService());

                //2. Test method
        int qty = obj.getTotalBooks("smita");

                //3. Verify result
        assertThat(qty, is(2));

    }

}
```
Copy

The above unit test is much better, fast, isolated (no more database) and the test condition (data) is always same.

3.3 But, there are some disadvantages to create mock object manually like above :

1. At the end, you may create many mock objects (classes), just for the unit test purpose.
2. If the interface contains many methods, you need to override each of them.
3. It's still too much work, and messy!

3.4 **Solution**
Try Mockito, a simple and powerful mocking framework.

4. Unit Test – Mockito

4.1 Update the unit test again, this time, create the mock object via Mockito framework.

pom.xml
```
        <dependency>
            <groupId>org.mockito</groupId>
            <artifactId>mockito-core</artifactId>
            <version>2.0.73-beta</version>
        </dependency>
```
Copy

AuthorServiceTest.java
```
package com.smita.mock;

import com.smita.examples.mock.AuthorServiceImpl;
import com.smita.examples.mock.Book;
import com.smita.examples.mock.BookServiceImpl;
import com.smita.examples.mock.FakeBookValidatorService;
import org.junit.Test;

import java.util.Arrays;
import java.util.List;

import static org.hamcrest.CoreMatchers.is;
import static org.junit.Assert.assertThat;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;

public class AuthorServiceTest {

            @Test
            public void test_total_book_by_mockito() {

                        //1. Setup
                List<Book> books = Arrays.asList(
                        new Book("smita in action"),
                        new Book("abc in action"),
                        new Book("bot"));

                BookServiceImpl mockito = mock(BookServiceImpl.class);

                //if the author is "smita", then return a 'books' object.
                when(mockito.findBookByAuthor("smita")).thenReturn(books);

                AuthorServiceImpl obj = new AuthorServiceImpl();
                obj.setBookService(mockito);
                obj.setBookValidatorService(new FakeBookValidatorService());

                        //2. Test method
                int qty = obj.getTotalBooks("smita");

                        //3. Verify result
                assertThat(qty, is(2));

            }

}
```