Junit

=====

JUnit is an open-source Unit Testing framework for Java. it can be useful for java developers to write and run test cases.

Erich Gamma and Kent Beck have initially developed this framework.

Once you are done with your code,you should execute all test cases and it should be a pass.

Q) What is meant by Unit Testing ?

Unit Testing is used to verify/test your piece of code by creating a function or method.

-> Unit Testing is used to identifying the defects in your development

Q) Why do you need Unit testing ?

-> I find the bugs early in the code , which makes our code more reliable

-> Junit is useful for developers who are working in a test-driven environment

Q) What are the Types of Unit Testing?

1) Manula Testing -

2) Automated Testing

Q) How to Configure JUnit?

By Placing Junit maven dependency in pom.xml

Q) How to define a test in Junit?

A JUnit test is a method in a class that is only used for testing. This class is called a Test class. To mark a method as a test method we can use @Test annotation. This method will execute your Junit code.

Note : Junit Test classes should be written inside the src/test/java

What is the difference between manual testing and automated testing?

Manual testing is performed by humans, so it is time-consuming and costly. Automated testing is performed by testing tools or programs, so it is fast and less costly.

import org.junit.Assert;

import org.junit.Test;

public class TestHello {

@Test

public void helloTest(){

Assert.assertTrue(true);

}

}

import org.junit.Assert;

import org.junit.BeforeClass;

import org.junit.Test;

import com.junit.Calculator;

public class CalculatorTest {

static Calculator cal;

@BeforeClass

public static void setUp(){

cal = new Calculator();

}

@Test

public void testAddition(){

Assert.assertEquals(11, cal.addition(5, 6));

}

@Test

public void testMultiply(){

Assert.assertEquals(30, cal.multiply(5, 6));

}

@Test

public void testEvenorOdd(){

Assert.assertEquals("odd", cal.evenorodd(10));

}

}

The Method annotated with @BeforeClass runs before Test

A method annotated with @Test defines a test method

@DisplayName can be used to define name of the test which is displayed to the user

Assert.assertEquals(30, cal.multiply(5, 6));

This is an assert statement which validates that is expected and actual values is same

Q) Where should the test classes be located?

src/test/java

assertEquals(boolean expected ,boolean actualvalue) -- > checks 2 objects are equla are notr

assertTrue(boolean condition) --> checks that condition is fine

assertNull(Object obj) - checks that the object is null

assertNotNull(Object obj) - checks that the object not null

static imports

================

JUnit allows to use static imports for its assets statements to make the testis code short and easy to use

@RepeatedTest

-------------

defines that this test method will be executed multiple times

Junit lifecyle

---------------

Junit test cases can contain many test methods .Each method identified as a test will be executed within the Junit test life cycle.

The life cycle contains 1.setup 2.test and 3.teardown

1.public void setup(){} before test

2.public void testXXX(){}

3.public void tearDown() ; called to do any required post processing after test

Assertions methods ;

-------------------

assertNull(Object x) -> Validates that te=he parameter is NULL

assertNotNull(Object x) ->Validates that the parameter is NOT NULL

assertTrue(boolean x) -> Validates that the parameter is true

assertFalse(boolean x) -> Validates that the parameter is false

assertEquals(Object x,Object y) -> it will passed based on equal equlas(Object obj1,Object obj2);

fail() - > programmatically fail the test

BeforeClass and AfterClass are in Jubit 4

package com.aplolis.junit\_practice;

import static org.junit.jupiter.api.Assertions.assertEquals;

import static org.junit.jupiter.api.Assertions.assertNotNull;

import static org.junit.jupiter.api.Assertions.assertTrue;

import java.util.stream.Stream;

import org.junit.jupiter.api.AfterEach;

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.DisplayName;

import org.junit.jupiter.api.Test;

public class CalculateTest {

Calculate calculate;

@BeforeEach

void setUp() {

calculate = new Calculate();

}

@Test

@DisplayName("Simple test for Addition")

public void addTest() {

assertEquals(20, calculate.add(10, 10));

}

@Test

@DisplayName("Simple test for Multiply")

public void multiPlyTest() {

assertEquals(100, calculate.multiply(10, 10));

}

@Test

@DisplayName("Simple test for Even or Odd")

public void evenorOddTest() {

assertEquals("even", calculate.isEvenorOdd(10));

}

public void testResult() {

String p1 = "test";

assertNotNull(p1);

}

@AfterEach

public void tearDown() {

System.out.println("ALl test passed success");

}

public void testJava8() {

assertTrue(Stream.of(1,2,3).mapToInt(i->i).sum()>5,()->"sum should be greaterthan 5");

}

}

Mockito

========

prerequisities

Knowledge on JAVA programming and Junit

what is Mocking ?

================

Mocking is a way of testing the functionality of a class . Mocking does not require any database for properties files and server mocking means we are creating dummy objects instead of using real objects.

it is mock por clone of real objects

To mock an Object we need to know the three concepts stub,fake and mock

stud : stub objects provide predefined data and given it to caller methods during testing.

fake : Fake are the objects that contain working implementation.

mock : mock objects act as a dummy or clone of the real objects in testing.

Benefits of Mockito

-------------------

No Handwriting: No need to write mock objects on your own

refactoring safe: renaming an interface method names or reordering parameters will not break the test case

Return Value support: Supports return values

Exception support: supports exceptions

Annotations Support: supports annotations

Mockito mock() method

---------------------

This mock() is used to create objects of a given class or interface. Mockito contains 5 mock() methods with different arguments

when we did not assign anything to mocks, they would return default values. All five methods perform the same function of mocking the objects

1.mock() --> mock(classname)

2.mock() with Answer () -- > it is used to create mock objects of a class or interface with specific procedure .

mock(class name, Answer default Anwer)

3. mock method with MockSettings; it is used to create mock objects with non-standard settings

mock(classname,MockSettings mockSettings)

4.mock() method with ReturnValues ()

mock(class name,ReturnValues return values) --it is deprecated and replaced with Answer

5. mock() with String

mock(class name,String name);

Mockito when(); it will be used when we want to mock to return specific values when a particular method is called.

"when abc() method is called then return BBC" mostly when() used when there is some condition to execute.

syntax : when(methodcall)

when(mock.abc()).thenReturn("BBC"); thenReturn is mostly used with the when() .

Mockito Verify (): used to verify or check whether a specific method is called or not.

Mockito Spy()

===============

Mockito provides a method to partially mock an object which is known as a spy.

Mockito reset (); it will reset the mocks which means all mocked objects

instead of creating new objects, it will mock the objects

it will prepare the test data

In order to use Mockito in our project, we need to use

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-all</artifactId>

<version>2.0.1-beta</version>

<scope>test</scope>

</dependency>

along with junit

Disadvantages of stubs

======================

stubs are useful in simple projects and scenarios but in complex scenarios, we need something more dynamic than stubs.

To overcome this mocks can be useful

//@Test

public void testTutorial(){

TutorialService service = mock(TutorialService.class);

List<String> list = Arrays.asList(" Core Java ", " Spring Core ", " Hibernate ", " Spring MVC ");

when(service.getTutotials("Hibernate")).thenReturn(list);

TutorialLogic logic = new TutorialLogic(service);

List<String> alltd = logic.getTutorialSpringBoot("Hibernate");

System.out.println(alltd);

Assert.assertEquals(2, alltd.size());

}

//@Test

public void testList(){

List<String> list = mock(List.class);

when(list.size()).thenReturn(10);

Assert.assertEquals(10, list.size());

}

//@Test

public void testListGet(){

List<String> list = mock(List.class);

when(list.get(0)).thenReturn("Mockito");

Assert.assertEquals("Mockito", list.get(0));

}

//@Test

public void verifyTest(){

TutorialService service = mock(TutorialService.class);

List<String> list = Arrays.asList(" Core Java ", " Spring Core ", " Hibernate ", " Spring MVC ");

verify(service).getTutotials("Hibernate");

when(service.getTutotials("Hibernate")).thenReturn(list);

Assert.assertEquals(1, service.getTutotials("Hibernate"));

}

@Test

public void testSpy(){

List<String> list = spy(List.class);

Assert.assertEquals(0, list.size());

list.add("Mockito");

Assert.assertEquals(1, list.size());

}

Argument Matchers

=================

Argument Matchers are mainly used for performing verification in mockito. it extends the "ArgumentMatchers" class

@Test

public void argumentmatcherTest(){

List<String> list = mock(List.class);

when(list.get(anyInt())).thenReturn("Mockito");

Assert.assertEquals("Mockito", list.get(2));

}