# JUnit

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

JUnit is a unit testing framework for Java programming language.

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."

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.

JUnit is a **Regression Testing Framework** used by developers to implement unit testing in Java

Download the latest version of JUnit jar file from [http://www.junit.org](https://github.com/downloads/junit-team/junit/junit-4.10.jar).

Set the **JUNIT\_HOME** environment variable to point to the base directory location where JUNIT jar is stored on your machine.

Set the **CLASSPATH** environment variable to point to the JUNIT jar location.

import org.junit.Test;

import static org.junit.Assert.assertEquals;

public class TestJunit {

@Test

public void testAdd() {

String str = "Junit is working fine";

assertEquals("Junit is working fine",str);

}

}

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());

}

}

C:\JUNIT\_WORKSPACE>javac TestJunit.java TestRunner.java

C:\JUNIT\_WORKSPACE>java TestRunner

Verify the output.

true

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;

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.

Example :-

/\*

\* 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;

}

}

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());

}

}

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());

}

}

C:\JUNIT\_WORKSPACE>javac MessageUtil.java TestJunit.java TestRunner.java

C:\JUNIT\_WORKSPACE>java TestRunner

Verify the output.

Hello World

true

Now update TestJunit in **C:\>JUNIT\_WORKSPACE** so that the test fails. Change the message string.

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() {

message = "New Word";

assertEquals(message,messageUtil.printMessage());

}

}

Now run the test runner

C:\JUNIT\_WORKSPACE>java TestRunner

Verify the output.

Hello World

testPrintMessage(TestJunit): expected:<[New Wor]d> but was:<[Hello Worl]d>

false

The most important package in JUnit is **junit.framework**, which contains all the core classes. Some of the important classes are as follows −

|  |  |  |
| --- | --- | --- |
| **Sr.No.** | **Class Name** | **Functionality** |
| 1 | Assert | A set of assert methods. |
| 2 | TestCase | A test case defines the fixture to run multiple tests. |
| 3 | TestResult | A TestResult collects the results of executing a test case. |
| 4 | TestSuite | A TestSuite is a composite of tests. |

## **Assert Class**

Following is the declaration for **org.junit.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 assertFalse(boolean condition)**  Checks that a condition is false. |
| 3 | **void assertNotNull(Object object)**  Checks that an object isn't null. |
| 4 | **void assertNull(Object object)**  Checks that an object is null. |
| 5 | **void assertTrue(boolean condition)**  Checks that a condition is true. |
| 6 | **void fail()**  Fails a test with no message. |
| 7 | **void assertSame(object1, object2)**  The assertSame() method tests if two object references point to the same object. |
| 8 | **void assertNotSame(object1, object2)**  The assertNotSame() method tests if two object references do not point to the same object. |
| 9 | **void assertArrayEquals(expectedArray, resultArray);**  The assertArrayEquals() method will test whether two arrays are equal to each other. |

import org.junit.Test;

import static org.junit.Assert.\*;

public class TestJunit1 {

@Test

public void testAdd() {

//test data

int num = 5;

String temp = null;

String str = "Junit is working fine";

//check for equality

assertEquals("Junit is working fine", str);

//check for false condition

assertFalse(num > 6);

//check for not null value

assertNotNull(temp);

}

}

## **TestCase Class**

Following is the declaration for **org.junit.TestCase** class −

public abstract class TestCase extends Assert implements Test

A test case defines the fixture to run multiple tests. Some of the important methods of **TestCase** class are as follows −

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **int countTestCases()**  Counts the number of test cases executed by run(TestResult result). |
| 2 | **TestResult createResult()**  Creates a default TestResult object. |
| 3 | **String getName()**  Gets the name of a TestCase. |
| 4 | **TestResult run()**  A convenience method to run this test, collecting the results with a default TestResult object. |
| 5 | **void run(TestResult result)**  Runs the test case and collects the results in TestResult. |
| 6 | **void setName(String name)**  Sets the name of a TestCase. |
| 7 | **void setUp()**  Sets up the fixture, for example, open a network connection. |
| 8 | **void tearDown()**  Tears down the fixture, for example, close a network connection. |
| 9 | **String toString()**  Returns a string representation of the test case. |

import junit.framework.TestCase;

import org.junit.Before;

import org.junit.Test;

public class TestJunit2 extends TestCase {

protected double fValue1;

protected double fValue2;

@Before

public void setUp() {

fValue1 = 2.0;

fValue2 = 3.0;

}

@Test

public void testAdd() {

//count the number of test cases

System.out.println("No of Test Case = "+ this.countTestCases());

//test getName

String name = this.getName();

System.out.println("Test Case Name = "+ name);

//test setName

this.setName("testNewAdd");

String newName = this.getName();

System.out.println("Updated Test Case Name = "+ newName);

}

//tearDown used to close the connection or clean up activities

public void tearDown( ) {

}

}

Write the TestRunner as above,

Compile the test case and Test Runner classes using javac.

C:\JUNIT\_WORKSPACE>javac TestJunit2.java TestRunner2.java

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

C:\JUNIT\_WORKSPACE>java TestRunner2

Verify the output.

No of Test Case = 1

Test Case Name = testAdd

Updated Test Case Name = testNewAdd

true

## **TestResult Class**

Following is the declaration for **org.junit.TestResult** class −

public class TestResult extends Object

A TestResult collects the results of executing a test case. It is an instance of the Collecting Parameter pattern. The test framework distinguishes between failures and errors. A failure is anticipated and checked for with assertions. Errors are unanticipated problems like an ArrayIndexOutOfBoundsException. Some of the important methods of **TestResult** class are as follows −

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **void addError(Test test, Throwable t)**  Adds an error to the list of errors. |
| 2 | **void addFailure(Test test, AssertionFailedError t)**  Adds a failure to the list of failures. |
| 3 | **void endTest(Test test)**  Informs the result that a test was completed. |
| 4 | **int errorCount()**  Gets the number of detected errors. |
| 5 | **Enumeration<TestFailure> errors()**  Returns an Enumeration for the errors. |
| 6 | **int failureCount()**  Gets the number of detected failures. |
| 7 | **void run(TestCase test)**  Runs a TestCase. |
| 8 | **int runCount()**  Gets the number of run tests. |
| 9 | **void startTest(Test test)**  Informs the result that a test will be started. |
| 10 | **void stop()**  Marks that the test run should stop. |

import org.junit.Test;

import junit.framework.AssertionFailedError;

import junit.framework.TestResult;

public class TestJunit3 extends TestResult {

// add the error

public synchronized void addError(Test test, Throwable t) {

super.addError((junit.framework.Test) test, t);

}

// add the failure

public synchronized void addFailure(Test test, AssertionFailedError t) {

super.addFailure((junit.framework.Test) test, t);

}

@Test

public void testAdd() {

// add any test

}

// Marks that the test run should stop.

public synchronized void stop() {

//stop the test here

}

}

C:\JUNIT\_WORKSPACE>java TestRunner3

Verify the output.

true

## **TestSuite Class**

Following is the declaration for **org.junit.TestSuite** class:

public class TestSuite extends Object implements Test

A TestSuite is a Composite of tests. It runs a collection of test cases. Some of the important methods of **TestSuite** class are as follows −

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **void addTest(Test test)**  Adds a test to the suite. |
| 2 | **void addTestSuite(Class<? extends TestCase> testClass)**  Adds the tests from the given class to the suite. |
| 3 | **int countTestCases()**  Counts the number of test cases that will be run by this test. |
| 4 | **String getName()**  Returns the name of the suite. |
| 5 | **void run(TestResult result)**  Runs the tests and collects their result in a TestResult. |
| 6 | **void setName(String name)**  Sets the name of the suite. |
| 7 | **Test testAt(int index)**  Returns the test at the given index. |
| 8 | **int testCount()**  Returns the number of tests in this suite. |
| 9 | **static Test warning(String message)**  Returns a test which will fail and log a warning message. |

import junit.framework.\*;

public class JunitTestSuite {

public static void main(String[] a) {

// add the test's in the suite

TestSuite suite = new TestSuite(TestJunit1.class, TestJunit2.class, TestJunit3.class );

TestResult result = new TestResult();

suite.run(result);

System.out.println("Number of test cases = " + result.runCount());

}

}

Compile the Test suite classes using javac.

C:\JUNIT\_WORKSPACE>javac JunitTestSuite.java

Now run the Test Suite.

C:\JUNIT\_WORKSPACE>java JunitTestSuite

Verify the output.

No of Test Case = 1

Test Case Name = testAdd

Updated Test Case Name = testNewAdd

Number of test cases = 3

Example :-

public class EmployeeDetails {

private String name;

private double monthlySalary;

private int age;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public double getMonthlySalary() {

return monthlySalary;

}

public void setMonthlySalary(double monthlySalary) {

this.monthlySalary = monthlySalary;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

}

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;

}

}

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);

}

}

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());

}

}

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

## **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. |

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");

}

}

C:\JUNIT\_WORKSPACE>java TestRunner

Verify the output.

in before class

in before

in test

in after

in after class

true

* First of all, the beforeClass() method executes only once.
* The afterClass() method executes only once.
* The before() method executes for each test case, but before executing the test case.
* The after() method executes for each test case, but after the execution of test case.
* In between before() and after(), each test case executes.

The test cases are executed using **JUnitCore** class. JUnitCore is a facade for running tests.

* A test method annotated with @Ignore will not be executed.
* If a test class is annotated with @Ignore, then none of its test methods will be executed.

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.

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());

}

}

Inside testPrintMessage()

Robert

Inside testSalutationMessage()

Hi!Robert

testPrintMessage(TestJunit): test timed out after 1000 milliseconds

false

JUnit provides an option of tracing the exception handling of code. You can test whether the code throws a desired exception or not. The **expected**parameter is used along with @Test annotation. Let us see @Test(expected) in action.

/\*

\* 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);

int a = 0;

int b = 1/a;

}

// add "Hi!" to the message

public String salutationMessage(){

message = "Hi!" + message;

System.out.println(message);

return message;

}

}

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(expected = ArithmeticException.class)

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());

}

}

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.

* 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.

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;

}

}

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));

}

}

Following are the JUnit extensions −

* Cactus
* JWebUnit
* XMLUnit
* MockObject

# Mockito

Mockito is a mocking framework, JAVA-based library that is used for effective unit testing of JAVA applications.

Mockito is used to mock interfaces so that a dummy functionality can be added to a mock interface that can be used in unit testing.

## **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.

## **Mockito**

Mockito 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. Mockito can do the same very easily, as its name suggests.

## **Benefits of Mockito**

* **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.
* package com.tutorialspoint.mock;
* import java.util.ArrayList;
* import java.util.List;
* import static org.mockito.Mockito.\*;
* public class PortfolioTester {
* public static void main(String[] args){
* //Create a portfolio object which is to be tested
* Portfolio portfolio = new Portfolio();
* //Creates a list of stocks to be added to the portfolio
* List<Stock> stocks = new ArrayList<Stock>();
* Stock googleStock = new Stock("1","Google", 10);
* Stock microsoftStock = new Stock("2","Microsoft",100);
* stocks.add(googleStock);
* stocks.add(microsoftStock);
* //Create the mock object of stock service
* StockService stockServiceMock = mock(StockService.class);
* // mock the behavior of stock service to return the value of various stocks
* when(stockServiceMock.getPrice(googleStock)).thenReturn(50.00);
* when(stockServiceMock.getPrice(microsoftStock)).thenReturn(1000.00);
* //add stocks to the portfolio
* portfolio.setStocks(stocks);
* //set the stockService to the portfolio
* portfolio.setStockService(stockServiceMock);
* double marketValue = portfolio.getMarketValue();
* //verify the market value to be
* //10\*50.00 + 100\* 1000.00 = 500.00 + 100000.00 = 100500
* System.out.println("Market value of the portfolio: "+ marketValue);
* }
* }
* **Portfolio** − An object to carry a list of stocks and to get the market value computed using stock prices and stock quantity.
* **Stock** − An object to carry the details of a stock such as its id, name, quantity, etc.
* **StockService** − A stock service returns the current price of a stock.
* **mock(...)** − Mockito created a mock of stock service.
* **when(...).thenReturn(...)** − Mock implementation of getPrice method of stockService interface. For googleStock, return 50.00 as price.
* **portfolio.setStocks(...)** − The portfolio now contains a list of two stocks.
* **portfolio.setStockService(...)** − Assigns the stockService Mock object to the portfolio.
* **portfolio.getMarketValue()** − The portfolio returns the market value based on its stocks using the mock stock service.

An Example :-

* public class Stock {
* private String stockId;
* private String name;
* private int quantity;
* public Stock(String stockId, String name, int quantity){
* this.stockId = stockId;
* this.name = name;
* this.quantity = quantity;
* }
* public String getStockId() {
* return stockId;
* }
* public void setStockId(String stockId) {
* this.stockId = stockId;
* }
* public int getQuantity() {
* return quantity;
* }
* public String getTicker() {
* return name;
* }
* }

public interface StockService {

public double getPrice(Stock stock);

}

import java.util.List;

public class Portfolio {

private StockService stockService;

private List<Stock> stocks;

public StockService getStockService() {

return stockService;

}

public void setStockService(StockService stockService) {

this.stockService = stockService;

}

public List<Stock> getStocks() {

return stocks;

}

public void setStocks(List<Stock> stocks) {

this.stocks = stocks;

}

public double getMarketValue(){

double marketValue = 0.0;

for(Stock stock:stocks){

marketValue += stockService.getPrice(stock) \* stock.getQuantity();

}

return marketValue;

}

}

package com.tutorialspoint.mock;

import java.util.ArrayList;

import java.util.List;

import static org.mockito.Mockito.\*;

public class PortfolioTester {

Portfolio portfolio;

StockService stockService;

public static void main(String[] args){

PortfolioTester tester = new PortfolioTester();

tester.setUp();

System.out.println(tester.testMarketValue()?"pass":"fail");

}

public void setUp(){

//Create a portfolio object which is to be tested

portfolio = new Portfolio();

//Create the mock object of stock service

stockService = mock(StockService.class);

//set the stockService to the portfolio

portfolio.setStockService(stockService);

}

public boolean testMarketValue(){

//Creates a list of stocks to be added to the portfolio

List<Stock> stocks = new ArrayList<Stock>();

Stock googleStock = new Stock("1","Google", 10);

Stock microsoftStock = new Stock("2","Microsoft",100);

stocks.add(googleStock);

stocks.add(microsoftStock);

//add stocks to the portfolio

portfolio.setStocks(stocks);

//mock the behavior of stock service to return the value of various stocks

when(stockService.getPrice(googleStock)).thenReturn(50.00);

when(stockService.getPrice(microsoftStock)).thenReturn(1000.00);

double marketValue = portfolio.getMarketValue();

return marketValue == 100500.0;

}

}

C:\Mockito\_WORKSPACE>javac Stock.java StockService.java Portfolio.java PortfolioTester.java

Now run the PortfolioTester to see the result −

C:\Mockito\_WORKSPACE>java PortfolioTester

Verify the Output

pass

**Mockito – Junit Integration :-**

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);

}

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);

}

}

import static org.mockito.Mockito.when;

import org.junit.Assert;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.mockito.InjectMocks;

import org.mockito.Mock;

import org.mockito.runners.MockitoJUnitRunner;

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

@RunWith(MockitoJUnitRunner.class)

public class MathApplicationTester {

//@InjectMocks annotation is used to create and inject the mock object

@InjectMocks

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

when(calcService.add(10.0,20.0)).thenReturn(30.00);

//test the add functionality

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

}

}

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());

}

}

C:\Mockito\_WORKSPACE>javac CalculatorService.java MathApplication.

java MathApplicationTester.java TestRunner.java

Now run the Test Runner to see the result −

C:\Mockito\_WORKSPACE>java TestRunner

Verify the output.

true

**Adding Behavior:-**

Mockito adds a functionality to a mock object using the methods **when()**.

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

when(calcService.add(10.0,20.0)).thenReturn(30.00);

Here we've instructed Mockito to give a behavior of adding 10 and 20 to the **add** method of **calcService** and as a result, to return the value of 30.00.

Example :-

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);

}

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);

}

}

import static org.mockito.Mockito.when;

import org.junit.Assert;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.mockito.InjectMocks;

import org.mockito.Mock;

import org.mockito.runners.MockitoJUnitRunner;

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

@RunWith(MockitoJUnitRunner.class)

public class MathApplicationTester {

//@InjectMocks annotation is used to create and inject the mock object

@InjectMocks

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

when(calcService.add(10.0,20.0)).thenReturn(30.00);

//test the add functionality

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

}

}

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());

}

}

C:\Mockito\_WORKSPACE>javac CalculatorService.java MathApplication.

java MathApplicationTester.java TestRunner.java

Now run the Test Runner to see the result −

C:\Mockito\_WORKSPACE>java TestRunner

Verify the output.

true

**Mockito – Verifying Behavior**

Mockito can ensure whether a mock method is being called with required arguments or not.

It is done using the **verify()** method.

//test the add functionality

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

//verify call to calcService is made or not with same arguments.

verify(calcService).add(10.0, 20.0);

## **Example - verify() with same arguments**

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);

}

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);

return 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);

}

}

import static org.mockito.Mockito.verify;

import static org.mockito.Mockito.when;

import org.junit.Assert;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.mockito.InjectMocks;

import org.mockito.Mock;

import org.mockito.runners.MockitoJUnitRunner;

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

@RunWith(MockitoJUnitRunner.class)

public class MathApplicationTester {

//@InjectMocks annotation is used to create and inject the mock object

@InjectMocks

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

when(calcService.add(10.0,20.0)).thenReturn(30.00);

//test the add functionality

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

//verify the behavior

verify(calcService).add(10.0, 20.0);

}

}

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());

}

}

C:\Mockito\_WORKSPACE>javac CalculatorService.java MathApplication.

java MathApplicationTester.java TestRunner.java

Now run the Test Runner to see the result

C:\Mockito\_WORKSPACE>java TestRunner

Verify the output.

true

## **Example - verify() with different arguments**

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);

}

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);

return 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);

}

}

import static org.mockito.Mockito.verify;

import static org.mockito.Mockito.when;

import org.junit.Assert;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.mockito.InjectMocks;

import org.mockito.Mock;

import org.mockito.runners.MockitoJUnitRunner;

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

@RunWith(MockitoJUnitRunner.class)

public class MathApplicationTester {

//@InjectMocks annotation is used to create and inject the mock object

@InjectMocks

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

when(calcService.add(10.0,20.0)).thenReturn(30.00);

//test the add functionality

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

//verify the behavior

verify(calcService).add(20.0, 30.0);

}

}

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());

}

}

C:\Mockito\_WORKSPACE>javac CalculatorService.java MathApplication.

java MathApplicationTester.java TestRunner.java

Now run the Test Runner to see the result −

C:\Mockito\_WORKSPACE>java TestRunner

Verify the output.

testAdd(MathApplicationTester):

Argument(s) are different! Wanted:

calcService.add(20.0, 30.0);

-> at MathApplicationTester.testAdd(MathApplicationTester.java:32)

Actual invocation has different arguments:

calcService.add(10.0, 20.0);

-> at MathApplication.add(MathApplication.java:10)

false

**Expecting Calls**

Mockito provides a special check on the number of calls that can be made on a particular method.

Suppose MathApplication should call the CalculatorService.serviceUsed() method only once, then it should not be able to call CalculatorService.serviceUsed() more than once.

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

when(calcService.add(10.0,20.0)).thenReturn(30.00);

//limit the method call to 1, no less and no more calls are allowed

verify(calcService, times(1)).add(10.0, 20.0);

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);

}

,,,,,,,,,,

……………………………… There is much more in Mockito

* + Varying Calls
  + Exception Handling
  + Create Mock
  + Ordered Verification
  + Callbacks
  + Spying
  + Resetting Mock
  + Behavior Driven Development
  + Timeouts

………………..