**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.

Mocking is very essential testing tool. Whenever we developing the web application, we develop the web, business, data and integration layer separately.

Web 
Business 
Dat 

Whenever we write the test cases for web layer which is dependent on business layer, so will be mock it out the business layer while writing the test cases for web layer. So that web layer test cases will not be dependent on business layer. Same rule applies for all layer as well.

**Example 1:**

**Writing a Simple Business Service:**

**SomeBusinessImpl.java**

package com.aman.unittesting;

public class SomeBusinessImpl {

public int calculateSum(int[] data) {

int sum = 0;

for(int value:data) {

sum += value;

}

return sum;

}

}

**Writing Unit Test for a Simple Business Service:**

**SomeBusinessTest.java**

package com.aman.unittesting.test;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

import com.aman.unittesting.SomeBusinessImpl;

public class SomeBusinessTest {

@Test

public void calculateSome\_Basic() {

SomeBusinessImpl businessImpl = new SomeBusinessImpl();

int actualResult = businessImpl.calculateSum(new int[] {1,2,3});

int expectedResult = 6;

assertEquals(expectedResult, actualResult);

}

@Test

public void calculateSome\_EmptyArray() {

SomeBusinessImpl businessImpl = new SomeBusinessImpl();

int actualResult = businessImpl.calculateSum(new int[] {});

int expectedResult = 0;

assertEquals(expectedResult, actualResult);

}

@Test

public void calculateSome\_OneValue() {

SomeBusinessImpl businessImpl = new SomeBusinessImpl();

int actualResult = businessImpl.calculateSum(new int[] {1});

int expectedResult = 1;

assertEquals(expectedResult, actualResult);

}

}

}

**Run This JUNIT Test: all success**

**Example 2:**

In above example, we are directly passing the data to test the our business API 'calculateSum()' but in real world we should not pass directly data like this. We should have some Data service from where you get the data generally.

**Writing a Simple Business Service:**

**SomeBusinessImpl.java :**

package com.aman.unittesting.business;

import com.aman.unittesting.data.SomeDataService;

public class SomeBusinessImpl {

SomeDataService someDataService;

public void setSomeDataService(SomeDataService someDataService) {

this.someDataService = someDataService;

}

public int calculateSum(int[] data) {

int sum = 0;

for(int value:data) {

sum += value;

}

return sum;

}

public int calculateSumUsingDataService() {

int sum = 0;

int[] data = someDataService.retriveAllData();

for(int value:data) {

sum += value;

}

return sum;

}

}

**Writing a Simple Data Layer As interface:**

**SomeDataService.java:**

package com.aman.unittesting.data;

public interface SomeDataService {

public int[] retriveAllData();

}

**Points:**

**\*** We need some data to test. so, we need to set the data.

\* Typically, we get the data from DB, but I don't want my test should not be dependent on DB or outside the test.

\* So, to set the data, we can create the STUB implementation of this.

**Note: Stub - an object with hardcoded behavior suitable for a given test (or a group of tests)**

**STUB1: Lets create the STUB like below:**

package com.aman.unittesting.test;

import com.aman.unittesting.data.SomeDataService;

public class SomeDataServiceStub implements SomeDataService{

@Override

public int[] retriveAllData() {

return new int[] {1,2,3};

}

}

**Writing Unit Test for a Simple Business Service:**

**SomeBusinessTest.java:**

package com.aman.unittesting.test;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

import com.aman.unittesting.business.SomeBusinessImpl;

public class SomeBusinessStubTest {

@Test

public void calculateSumUsingDataService\_Basic() {

SomeBusinessImpl businessImpl = new SomeBusinessImpl();

**/\*\*\*\*\* We need some data to test. so we need to set the data.**

**\* Typically we get the data from DB, but I don't want my test should not be dependent on DB or outside the test.**

**\* So to set the data, we can create the STUB implementation of this.**

**\*\*\*\*\*/**

**businessImpl.setSomeDataService(new SomeDataServiceStub());**

int actualResult = businessImpl.calculateSumUsingDataService();

int expectedResult = 6;

assertEquals(expectedResult, actualResult);

}

}

**Run This JUNIT Test: success**

Remember, in first program we have 3 test scenarios to test, one is basic, 2nd one is empty array and 3rd one is with one value in array. The problem is above STUB is, we would be able to test only one scenario test which is 'Basic', other two test we, won’t be able to test.

So, to test other two test, we need to create two more STUBS.

**STUB2:**

package com.aman.unittesting.test;

import com.aman.unittesting.data.SomeDataService;

public class SomeDataServiceStub2 implements SomeDataService{

@Override

public int[] retriveAllData() {

return new int[] {};

}

}

**STUB3:**

package com.aman.unittesting.test;

import com.aman.unittesting.data.SomeDataService;

public class SomeDataServiceStub3 implements SomeDataService{

@Override

public int[] retriveAllData() {

return new int[] {1};

}

}

**Again, Writing Unit Test for Simple Business Service:**

**SomeBusinessTest.java:**

package com.aman.unittesting.test;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

import com.aman.unittesting.business.SomeBusinessImpl;

public class SomeBusinessStubTest {

@Test

public void calculateSumUsingDataService\_Basic() {

SomeBusinessImpl businessImpl = new SomeBusinessImpl();

businessImpl.setSomeDataService(new SomeDataServiceStub());

int actualResult = businessImpl.calculateSumUsingDataService();

int expectedResult = 6;

assertEquals(expectedResult, actualResult);

}

@Test

public void calculateSumUsingDataService\_EmptyArray() {

SomeBusinessImpl businessImpl = new SomeBusinessImpl();

businessImpl.setSomeDataService(new SomeDataServiceStub2());

int actualResult = businessImpl.calculateSumUsingDataService();

int expectedResult = 0;

assertEquals(expectedResult, actualResult);

}

@Test

public void calculateSumUsingDataService\_OneValue() {

SomeBusinessImpl businessImpl = new SomeBusinessImpl();

businessImpl.setSomeDataService(new SomeDataServiceStub3());

int actualResult = businessImpl.calculateSumUsingDataService();

int expectedResult = 1;

assertEquals(expectedResult, actualResult);

}

}

**Run This JUNIT Test: all success**

**Problem 1:** We need to write as many as STUB based on no. of test cases uses. In future, test cases increase and comes with new requirement, we will have to again create one more STUB.

**Problem 2:** Maintaining the STUBS is more difficult also. Let’s say, what if I need to add one more method in our interface 'SomeDataService'… What would happen?

**Ans.:** All our STUB would get compilation error, because now all STUB needs to implements the new method as well. That’s not really good at all. Maintenance of STUBS is very difficult

**Problem 3:** When you use STUB, then it tough to understand what kind of data will be returning by STUB. SO, you may need to do little bit analysis by open STUB class and see to know exact.

**These are problems of using STUB and that’s why MOCKITO comes into the picture.**

**Note: Mock - an object with the ability to a) have a programmed expected behavior, and b) verify the interactions occurring in its lifetime (this object is usually created with the help of mocking framework)**

**SomeBusinessMockTest.java:**

package com.aman.unittesting.test;

import static org.junit.Assert.assertEquals;

**import static org.mockito.Mockito.mock;**

**import static org.mockito.Mockito.when;**

import org.junit.Test;

import com.aman.unittesting.business.SomeBusinessImpl;

import com.aman.unittesting.data.SomeDataService;

public class SomeBusinessMockTest {

@Test

public void calculateSumUsingDataService\_Basic() {

SomeBusinessImpl businessImpl = new SomeBusinessImpl();

**/\*\*\*\*\* Creating the mock of 'SomeDataService' class \*\*\*\*\*/**

**SomeDataService someDataService = mock(SomeDataService.class);**

**/\*\*\*\* When 'retriveAllData()' of 'SomeDataService' is called then return "new int[] {1,2,3}" \*\*\*\*/**

when(someDataService.retriveAllData()).thenReturn(new int[] {1,2,3});

**businessImpl.setSomeDataService(someDataService);**

int actualResult = businessImpl.calculateSumUsingDataService();

int expectedResult = 6;

assertEquals(expectedResult, actualResult);

}

@Test

public void calculateSumUsingDataService\_EmptyArray() {

SomeBusinessImpl businessImpl = new SomeBusinessImpl();

SomeDataService someDataService = mock(SomeDataService.class);

when(someDataService.retriveAllData()).thenReturn(new int[] {});

businessImpl.setSomeDataService(someDataService);

int actualResult = businessImpl.calculateSumUsingDataService();

int expectedResult = 0;

assertEquals(expectedResult, actualResult);

}

@Test

public void calculateSumUsingDataService\_OneValue() {

SomeBusinessImpl businessImpl = new SomeBusinessImpl();

SomeDataService someDataService = mock(SomeDataService.class);

when(someDataService.retriveAllData()).thenReturn(new int[] {1});

businessImpl.setSomeDataService(someDataService);

int actualResult = businessImpl.calculateSumUsingDataService();

int expectedResult = 1;

assertEquals(expectedResult, actualResult);

}

}

**Run This JUNIT Test: all success**

**SomeBusinessMockTest.java: Reduced the above code**

package com.aman.unittesting.test;

import static org.junit.Assert.assertEquals;

import static org.mockito.Mockito.mock;

import static org.mockito.Mockito.when;

import org.junit.Before;

import org.junit.Test;

import com.aman.unittesting.business.SomeBusinessImpl;

import com.aman.unittesting.data.SomeDataService;

public class SomeBusinessMockTest {

SomeBusinessImpl businessImpl = new SomeBusinessImpl();

**/\*\*\*\*\* Creating the mock of 'SomeDataService' class \*\*\*\*\*/**

SomeDataService someDataService = mock(SomeDataService.class);

@Before

public void before() {

businessImpl.setSomeDataService(someDataService);

}

@Test

public void calculateSumUsingDataService\_Basic() {

**/\*\*\*\* When 'retriveAllData()' of 'SomeDataService' is called then return "new int[] {1,2,3}" \*\*\*\*/**

when(someDataService.retriveAllData()).thenReturn(new int[] {1,2,3});

int actualResult = businessImpl.calculateSumUsingDataService();

assertEquals(6, actualResult);

}

@Test

public void calculateSumUsingDataService\_EmptyArray() {

when(someDataService.retriveAllData()).thenReturn(new int[] {});

int actualResult = businessImpl.calculateSumUsingDataService();

assertEquals(0, actualResult);

}

@Test

public void calculateSumUsingDataService\_OneValue() {

when(someDataService.retriveAllData()).thenReturn(new int[] {1});

int actualResult = businessImpl.calculateSumUsingDataService();

assertEquals(1, actualResult);

}

}

**Run This JUNIT Test: all success**

**SomeBusinessMockTest.java:** Again, Reduced the above code: In previous

SomeBusinessMockTest.java, we are creating below code:

SomeBusinessImpl businessImpl = new SomeBusinessImpl();

**/\*\*\*\*\* Creating the mock of 'SomeDataService' class \*\*\*\*\*/**

SomeDataService someDataService = mock(SomeDataService.class);

@Before

public void before() {

businessImpl.setSomeDataService(someDataService);

}

In above code, we are creating object of 'SomeBusinessImpl' and creating mock of 'SomeDataService' using mock(SomeDataService.class) and then setting the 'someDataService' in @Before method for each test.

**Above code can be replaced by below code:**

/**/ Using @InjectMocks, we don't need create the object of SomeBusinessImpl.**

**@InjectMocks**

SomeBusinessImpl businessImpl;

/\*\*\*\*\* Creating the mock of 'SomeDataService' class \*\*\*\*\*/

**@Mock**

SomeDataService someDataService;

And also add the annotation **"@RunWith(MockitoJUnitRunner.class)"** at the class level.

So here, we are doing the mock of some data service which is defined in some data service implementation without setting through setter. So, our test is totally independent of DB or out-side of test.

In future we can do mock more data service if needed without any business logic or DB layer changes.

**SomeBusinessMockTest.java: Again Modified.**

package com.aman.unittesting.test;

import static org.junit.Assert.assertEquals;

import static org.mockito.Mockito.mock;

import static org.mockito.Mockito.when;

import org.junit.Before;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.mockito.InjectMocks;

import org.mockito.Mock;

import org.mockito.Mockito;

import org.mockito.junit.MockitoJUnitRunner;

import com.aman.unittesting.business.SomeBusinessImpl;

import com.aman.unittesting.data.SomeDataService;

**@RunWith(MockitoJUnitRunner.class)**

public class SomeBusinessMockTest {

// Using @InjectMocks, we don't need create the object of SomeBusinessImpl.

@InjectMocks

SomeBusinessImpl businessImpl;

/\*\*\*\*\* Creating the mock of 'SomeDataService' class \*\*\*\*\*/

@Mock

SomeDataService someDataService;

@Test

public void calculateSumUsingDataService\_Basic() {

/\*\*\*\* When 'retriveAllData()' of 'SomeDataService' is called then return "new int[] {1,2,3}" \*\*\*\*/

when(someDataService.retriveAllData()).thenReturn(new int[] {1,2,3});

int actualResult = businessImpl.calculateSumUsingDataService();

assertEquals(6, actualResult);

}

@Test

public void calculateSumUsingDataService\_EmptyArray() {

when(someDataService.retriveAllData()).thenReturn(new int[] {});

int actualResult = businessImpl.calculateSumUsingDataService();

assertEquals(0, actualResult);

}

@Test

public void calculateSumUsingDataService\_OneValue() {

when(someDataService.retriveAllData()).thenReturn(new int[] {1});

int actualResult = businessImpl.calculateSumUsingDataService();

assertEquals(1, actualResult);

}

}

**Run This JUNIT Test : all success**

**Concept:**

**ListMockTestjava: How to mock list class of Java.**

package com.aman.unittesting.test;

import static org.junit.Assert.assertEquals;

import static org.mockito.Mockito.mock;

import static org.mockito.Mockito.never;

import static org.mockito.Mockito.spy;

import static org.mockito.Mockito.times;

import static org.mockito.Mockito.verify;

import static org.mockito.Mockito.when;

import static org.mockito.Mockito.anyInt;

import static org.mockito.Mockito.atLeast;

import static org.mockito.Mockito.atLeastOnce;

import static org.mockito.Mockito.atMost;

import java.util.ArrayList;

import java.util.List;

import org.junit.Test;

import org.mockito.ArgumentCaptor;

public class ListMockTest {

List listMock = mock(List.class);

@Test

public void sizeListTest() {

//List listMock = mock(List.class);

when(listMock.size()).thenReturn(5);

assertEquals(5, listMock.size());

}

@Test

public void returnMultipleValues() {

//List listMock = mock(List.class);

**/\*\*\*\***

**\* When first time 'listMock.size()' gets called, it will return 5 and**

**\* when next time 'listMock.size()' gets called, it will return 10**

**\*/**

when(listMock.size()).thenReturn(5).thenReturn(10);

assertEquals(5, listMock.size());

assertEquals(10, listMock.size());

}

@Test

public void returnWithParameter() {

//List listMock = mock(List.class);

**/\*\* When we get 0th index from list it will return "Mocked Indexed Value"**

**\* when we try to get another index except 0 like 4 from list then we will get default value i.e. null in this test**

**\*\*/**

when(listMock.get(0)).thenReturn("Mocked Indexed Value");

assertEquals("Mocked Indexed Value", listMock.get(0));

assertEquals(null, listMock.get(4));

}

@Test

public void returnWithGenericParameter() {

//List listMock = mock(List.class);

**/\*\*  We can use 'anyInt()' method means when we try to get any integer index from list,**

**\* we will get "Mocked Indexed Value".**

**\* Similarly like anyInt()' method, we have a lot of methods , like anyLong(), anyDouble() etc.**

**\*\*/**

when(listMock.get(anyInt())).thenReturn("Mocked Indexed Value");

assertEquals("Mocked Indexed Value", listMock.get(0));

assertEquals("Mocked Indexed Value", listMock.get(4));

}

**/\*\*\*\***

**\* In this test, we will learn, how to verify how many whether specific method is called with the specific value on a mock.**

**\* This is imp in those scenarios where value is not returned back. Let’s say we are calling another method (which does not return anything)** **from some mocked method.**

**\***

**\* In this example, we are taking List class and in real time application it can be our application class and we would be checking verify how many whether our specific method is called with the specific value**

**\*/**

@Test

public void verificationBasics() {

//Here we are calling get method of mocked class List

String value1= (String) listMock.get(0);

String value2= (String) listMock.get(1);

//This will verify whether get() method is ever called with data 0

verify(listMock).get(0);

//This will verify  get() method is ever called twice with any integer data

verify(listMock, times(2)).get(anyInt());

//This will verify  get() method is at-least called once with any integer data

verify(listMock, atLeast(1)).get(anyInt());

//This will verify  get() method is at -east once called once with any integer data

verify(listMock, atLeastOnce()).get(anyInt());

//This will verify  get() method is at-most called twice with any integer data

verify(listMock, atMost(2)).get(anyInt());

//This will verify whether get() method is ever called with data 2

verify(listMock, never()).get(2);

}

**/\*\*\*\***

**\*How to capture a single argument that is passed to a method call.**

**\*/**

@Test

public void argumentCapturing() {

//Here we are calling get method of mocked class List and with argument 'SomeString'

listMock.add("SomeString");

//verification

//This is String because we are verifying argument is 'String' type.

ArgumentCaptor<String> argumentCaptor = ArgumentCaptor.forClass(String.class);

//We are trying to capture the argument by "argumentCaptor.capture()" which is passed to call add method on mocked List object.

verify(listMock).add(argumentCaptor.capture());

assertEquals("SomeString", argumentCaptor.getValue());

}

**/\*\*\*\***

**\*How to capture an multiple argument that is passed to a method call .**

**\*/**

@Test

public void multipleArgumentCapturing() {

listMock.add("SomeString1");

listMock.add("SomeString2");

//verification

//This is String because we are verifying agrument is 'String' type.

ArgumentCaptor<String> argumentCaptor = ArgumentCaptor.forClass(String.class);

//We are trying to capture the argument by "argumentCaptor.capture()" which is passed to call add method on mocked List object.

verify(listMock, times(2)).add(argumentCaptor.capture());

List<String> allValues = argumentCaptor.getAllValues();

assertEquals("SomeString1", allValues.get(0));

assertEquals("SomeString2", allValues.get(1));

}

@Test

public void mocking() {

//By Mocking ArrayList means every method in ArrayList is mocked , not the real method.

ArrayList arrayListMock = mock(ArrayList.class);

System.out.println(arrayListMock.get(0));// It will return by default value which is null

System.out.println(arrayListMock.size());// It will return by default value which is  0

arrayListMock.add("Test");

arrayListMock.add("Test2");

**/\*\*\*\***

**\* After adding two values in array list, still It will return by default value which is**

**\*  because we have added value in mocked ArrayList, not the real ArrayList here.**

**\*  So this mocked ArrayList method will return what we ask to ArrayList's method to return.**

**\*/**

System.out.println(arrayListMock.size());

when(arrayListMock.size()).thenReturn(5);

System.out.println(arrayListMock.size());//Now it will return 5

}

**Spy - a mock created as a proxy to an existing real object; some methods can be stubbed, while the un- stubbed ones are forwarded to the covered object:**

**/\*\*\*\***

**\* A spy, by default, retains behavior (code) of the original class!**

**\* means in this example, original behavior of ArrayList will be retained and**

**\* you can stub specific methods if you want to change the behavior of some specific method as well.**

**\*/**

@Test

public void spying() {

ArrayList arrayListSpy = spy(ArrayList.class);

/\*\*\* In this case, below line would throw an exception because ArrayList does not have any value as of now. \*\*/

//System.out.println(arrayListMock.get(0));

arrayListSpy.add("Test0");

System.out.println(arrayListSpy.get(0));//returns Test0

System.out.println(arrayListSpy.size());//returns 1

arrayListSpy.add("Test");

arrayListSpy.add("Test2");

System.out.println(arrayListSpy.size());//returns 3

//Here, we are changing the behavior if size method of ArrayList.

when(arrayListSpy.size()).thenReturn(5);

System.out.println(arrayListSpy.size());//Now it will return 5

**/\*\*\* We can also verify on spy as well.**

**\* When we don't have access to specific class to get the data from it but**

**\* you want to check what is going underneath for that specific class and**

**\* what method is getting called. Tn this kind of scenario, we can use 'Spy'**

**\*/**

verify(arrayListSpy).add("Test");

}

}

**Run This JUNIT Test: all success**

**Concept: HamcrestMatcherTest:**

[**1. Purpose of the Hamcrest matcher framework**](http://www.vogella.com/tutorials/Hamcrest/article.html#hamcrestoverview):

*Hamcrest* is a framework for software tests. Hamcrest allows checking for conditions in your code via existing matchers classes. It also allows you to define your custom matcher implementations.

To use Hamcrest matchers in JUnit you use the assertThat statement followed by one or several matchers.

Hamcrest is typically viewed as a third generation matcher framework. The first generation used assert(logical statement) but such tests were not easily readable. The second generation introduced special methods for assertions, e.g., assertEquals(). This approach leads to lots of assert methods. Hamcrest uses assertThat method with a matcher expression to determine if the test was succesful. See [Wiki on Hamcrest](http://en.wikipedia.org/wiki/Hamcrest) for more details.

Hamcrest has the target to make tests as readable as possible. For example, the is method is a thin wrapper for equalTo(value).

**import** static org.hamcrest.MatcherAssert.assertThat;  
**import** static org.hamcrest.Matchers.is;  
**import** static org.hamcrest.Matchers.equalTo;

**boolean** a;  
**boolean** b;

*// all statements test the same*  
assertThat(a, equalTo(b));  
assertThat(a, is(equalTo(b)));  
assertThat(a, is(b));

The following snippets compare pure JUnit 4 assert statements with Hamcrest matchers.

*// JUnit 4 for equals check*  
assertEquals(expected, actual);  
*// Hamcrest for equals check*  
assertThat(actual, is(equalTo(expected)));

*// JUnit 4 for not equals check*  
assertNotEquals(expected, actual)  
*// Hamcrest for not equals check*  
assertThat(actual, is(not(equalTo(expected))));

It is also possible to chain matchers, via the anyOf of allOf method.

assertThat("test", anyOf(is("testing"), containsString("est")));

In general the Hamcrest error messages are also much easier to read.

assertTrue(result **instanceof** String);

*// error message:*  
java.lang.AssertionError  
 at org.junit.Assert.fail(Assert.java:86)  
 at org.junit.Assert.assertTrue(Assert.java:41)  
 at org.junit.Assert.assertTrue(Assert.java:52)  
*// ...*

assertEquals(String.class, result.getClass());  
*// error message:*  
java.lang.NullPointerException  
 at com.vogella.hamcrest.HamcrestTest.test(HamcrestTest.java:30)  
*// ....*

assertThat(result, instanceOf(String.class));  
*// error message:*  
java.lang.AssertionError:  
Expected: an instance of java.lang.String  
 but: null  
 at org.hamcrest.MatcherAssert.assertThat(MatcherAssert.java:20)  
 at org.hamcrest.MatcherAssert.assertThat(MatcherAssert.java:8)  
*// …*

Using Hamcrest matchers also provides more type safety as these matchers use generics.

[**2. Making the Hamcrest library available**](http://www.vogella.com/tutorials/Hamcrest/article.html#hamcrest_installation)

[**2.1. Defining a Hamcrest dependency for Gradle**](http://www.vogella.com/tutorials/Hamcrest/article.html#hamcrest_gradle)

To use Hamcrest matchers for a project based on the Gradle build system, add the following dependencies to it.

dependencies {  
 *// Unit testing dependencies*  
 testImplementation 'junit:junit:4.12'  
 *// Set this dependency if you want to use Hamcrest matching*  
 testImplementation 'org.hamcrest:hamcrest-library:1.3'  
}

[**2.2. Defining a Hamcrest dependency for Maven**](http://www.vogella.com/tutorials/Hamcrest/article.html#hamcrest_maven)

To use the library for a Maven based project, the following dependency to your pom file.

<dependency>  
<groupId>org.hamcrest</groupId>  
<artifactId>hamcrest-library</artifactId>  
<version>1.3</version>  
<scope>test</scope>  
</dependency>

**HamcrestMatcherTest.java:**

package com.aman.unittesting.test;

import static org.hamcrest.CoreMatchers.hasItems;

import static org.hamcrest.MatcherAssert.assertThat;

import static org.hamcrest.Matchers.arrayContainingInAnyOrder;

import static org.hamcrest.Matchers.arrayWithSize;

import static org.hamcrest.Matchers.greaterThan;

import static org.hamcrest.Matchers.hasSize;

import static org.hamcrest.Matchers.isEmptyOrNullString;

import static org.hamcrest.Matchers.isEmptyString;

import static org.hamcrest.Matchers.lessThan;

import static org.hamcrest.core.Every.everyItem;

import java.util.Arrays;

import java.util.List;

import org.junit.Test;

public class HamcrestMatcherTest {

@Test

public void basicHamcrestMatchers() {

List<Integer> scores = Arrays.asList(99, 100, 101, 105);

assertThat(scores, hasSize(4));

assertThat(scores, hasItems(100, 101));

assertThat(scores, everyItem(greaterThan(90)));

assertThat(scores, everyItem(lessThan(200)));

// String

assertThat("", isEmptyString());

assertThat(null, isEmptyOrNullString());

// Array

Integer[] marks = { 1, 2, 3 };

assertThat(marks, arrayWithSize(3));

assertThat(marks, arrayContainingInAnyOrder(2, 3, 1));

}

}

**Concept: AssertJTest.java:**

package com.aman.unittesting.test;

import static org.assertj.core.api.Assertions.assertThat;

import java.util.Arrays;

import java.util.List;

import org.junit.Test;

public class AssertJTest {

@Test

public void learning() {

List<Integer> numbers = Arrays.asList(12,15,45);

assertThat(numbers).hasSize(3).contains(12,15)

.allMatch(x-> x > 10)

.allMatch(x-> x < 100)

.noneMatch(x -> x < 0);

assertThat("").isEmpty();

assertThat("ABCDE").contains("BCD").startsWith("ABC").endsWith("CDE");

}

}

**Concept: JsonPathTest.java:**

package com.aman.unittesting.test;

import static org.assertj.core.api.Assertions.assertThat;

import java.util.List;

import org.junit.Test;

import com.jayway.jsonpath.DocumentContext;

import com.jayway.jsonpath.JsonPath;

public class JsonPathTest {

@Test

public void learning() {

String responseFromService = "[" +

"{\"id\":10000,\"name\":\"Pencil\", \"quantity\":5}," +

"{\"id\":10001,\"name\":\"Pen\", \"quantity\":15}," +

"{\"id\":10002,\"name\":\"Eraser\", \"quantity\":10}" +

"]";

DocumentContext context =JsonPath.parse(responseFromService);

int length = context.read("$.length()");

assertThat(length).isEqualTo(3);

List<Integer> ids = context.read("$..id");

assertThat(ids).containsExactly(10000,10001,10002);

System.out.println(context.read("$.[1]").toString());

System.out.println(context.read("$.[0:2]").toString());

System.out.println(context.read("$.[?(@.name=='Eraser')]").toString());

System.out.println(context.read("$.[?(@.quantity==5)]").toString());

} }

**Mocking Exception Throwing using Mockito:**

**1. Overview**

In this quick tutorial – we’ll focus on how to configure a method call to throw an exception with Mockito.

For more information on the library, also check out our [Mockito series](https://www.baeldung.com/mockito-final).

Here’s a simple dictionary class we’ll use in these examples:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | class MyDictionary {      private Map<String, String> wordMap = new HashMap<>();        public void add(String word, String meaning) {          wordMap.put(word, meaning);      }        public String getMeaning(String word) {          return wordMap.get(word); }} |

**2. Non-*Void* Return Type**

First, if our method return type is not *void* we can use *when().thenThrow()*:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @Test(expected = NullPointerException.class)  public void whenConfigNonVoidRetunMethodToThrowEx\_thenExIsThrown() {      MyDictionary dictMock = mock(MyDictionary.class);      when(dictMock.getMeaning(anyString()))        .thenThrow(NullPointerException.class);        dictMock.getMeaning("word");  } |

Notice, we configured the *getMeaning()* method – which returns a value of type *String* – to throw a *NullPointerException* when called.

**3. *Void* Return Type**

Now, if our method returns *void,* we’ll use *doThrow()*:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @Test(expected = IllegalStateException.class)  public void whenConfigVoidRetunMethodToThrowEx\_thenExIsThrown() {      MyDictionary dictMock = mock(MyDictionary.class);      doThrow(IllegalStateException.class)        .when(dictMock)        .add(anyString(), anyString());        dictMock.add("word", "meaning");  } |

Here, we configured an *add()* method – which returns *void* – to throw *IllegalStateException* when called.

We can’t use *when().thenThrow()* with *void* return type as the compiler doesn’t allow *void* methods inside brackets.

**4. Exception as an Object**

About configuring the exception itself, we can pass the exception’s class as in our previous examples or as an object:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @Test(expected = NullPointerException.class)  public void whenConfigNonVoidRetunMethodToThrowExWithNewExObj\_thenExIsThrown() {      MyDictionary dictMock = mock(MyDictionary.class);      when(dictMock.getMeaning(anyString()))        .thenThrow(new NullPointerException("Error occurred"));        dictMock.getMeaning("word");  } |

And we can do the same with*doThrow()* as well:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @Test(expected = IllegalStateException.class)  public void whenConfigVoidRetunMethodToThrowExWithNewExObj\_thenExIsThrown() {      MyDictionary dictMock = mock(MyDictionary.class);      doThrow(new IllegalStateException("Error occurred"))        .when(dictMock)        .add(anyString(), anyString());        dictMock.add("word", "meaning");  } |

**5. Spy**

We can also configure *Spy* to throw an exception the same way we did with the mock:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1  2  3  4  5  6  7  8  9 | @Test(expected = NullPointerException.class)  public void givenSpy\_whenConfigNonVoidRetunMethodToThrowEx\_thenExIsThrown() {      MyDictionary dict = new MyDictionary();      MyDictionary spy = Mockito.spy(dict);      when(spy.getMeaning(anyString()))        .thenThrow(NullPointerException.class);        spy.getMeaning("word");  }            **Mocking Void Methods with Mockito:**  **1. Overview**  In this short tutorial, we focus on mocking *void* methods with Mockito.  As with other articles focused on the Mockito framework (like [Mockito Verify](https://www.baeldung.com/mockito-verify), [Mockito When/Then](https://www.baeldung.com/mockito-behavior), and [Mockito’s Mock Methods](https://www.baeldung.com/mockito-mock-methods)) the *MyList* class shown below will be used as the collaborator in test cases. We’ll add a new method for this tutorial:   |  |  | | --- | --- | | 1  2  3  4  5  6  7 | public class MyList extends AbstractList<String> {        @Override      public void add(int index, String element) {          // no-op      }  } |   **2. Simple Mocking and Verifying**  *Void* methods can be used with Mockito’s *doNothing(), doThrow(), and doAnswer()* methods, making mocking and verifying intuitive:   |  |  | | --- | --- | | 1  2  3  4  5  6  7  8 | @Test  public void whenAddCalledVerfied() {      MyList myList = mock(MyList.class);      doNothing().when(myList).add(isA(Integer.class), isA(String.class));      myList.add(0, "");        verify(myList, times(1)).add(0, "");  } |   **However, *doNothing()* is Mockito’s default behavior for void methods.**  This version of *whenAddCalledVerified()* accomplishes the same thing as the one above:   |  |  | | --- | --- | | 1  2  3  4  5  6  7 | @Test  public void whenAddCalledVerfied() {      MyList myList = mock(MyList.class);      myList(0, "");        verify(myList, times(1)).add(0, "");  } |   *DoThrow()* generates an exception:   |  |  | | --- | --- | | 1  2  3  4  5  6  7 | @Test(expected = Exception.class)  public void givenNull\_AddThrows() {      MyList myList = mock(MyList.class);      doThrow().when(myList).add(isA(Integer.class), isNull());        myList.add(0, null);  } |   We’ll cover *doAnswer()* below.  **3. Argument Capture**  **One reason to override the default behavior with *doNothing()*is to capture arguments.**  In the example above *verify()* is used to check the arguments passed to *add()*.  However, we may need to capture the arguments and do something more with them. In these cases, we use *doNothing()* just as we did above, but with an *ArgumentCaptor*:   |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9 | @Test  public void whenAddCalledValueCaptured() {      MyList myList = mock(MyList.class);      ArgumentCaptor valueCapture = ArgumentCaptor.forClass(String.class);      doNothing().when(myList).add(any(Integer.class), valueCapture.capture());      myList.add(0, "captured");        assertEquals("captured", valueCapture.getValue());  } |   **4. Answering a Call to *Void***  A method may perform more complex behavior than merely adding or setting value. For these situations we can use Mockito’s *Answer* to add the behavior we need:   |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12  13 | @Test  public void whenAddCalledAnswered() {      MyList myList = mock(MyList.class);      doAnswer((Answer) invocation -> {          Object arg0 = invocation.getArgument(0);          Object arg1 = invocation.getArgument(1);            assertEquals(3, arg0);          assertEquals("answer me", arg1);          return null;      }).when(myList).add(any(Integer.class), any(String.class));      myList.add(3, "answer me");  } |   As explained in [Mockito’s Java 8 Features](https://www.baeldung.com/mockito-2-java-8) we use a lambda with *Answer* to define custom behavior for *add()*.  **5. Partial Mocking**  Partial mocks are an option, too. Mockito’s *doCallRealMethod()* can be used for *void* methods:   |  |  | | --- | --- | | 1  2  3  4  5  6  7  8 | @Test  public void whenAddCalledRealMethodCalled() {      MyList myList = mock(MyList.class);      doCallRealMethod().when(myList).add(any(Integer.class), any(String.class));      myList.add(1, "real");        verify(myList, times(1)).add(1, "real");  } |   **This allows us to call the actual method is called and verify it at the same time.** |