Unit tests should be small tests (atomic), lightweight, and fast. However, an object under test might have dependencies on other objects. It might need to interact with a database, communicate with a mail server, or talk to a web service or a message queue. All these services might not be available during unit testing. Even if they are available, unit testing the *object under test* along with its dependencies can take unacceptable amount of time. What if?

**Mocking In Unit Tests with Mockito**

* *The web service is not reachable.*
* *The database is down for maintenance.*
* *The message queue is heavy and slow.*

These all defeat the whole purpose of unit tests being atomic, lightweight, and fast. We want unit tests to execute in a few milliseconds. If the unit tests are slow, your builds become slow, which affects the productivity of your development team. The solution is to use mocking, a way to provide test doubles for your classes being tested.

If you’ve been following the [SOLID Principles of Object Oriented Programming](https://springframework.guru/solid-principles-object-oriented-programming/), and using the Spring Framework for [Dependency Injection](https://springframework.guru/dependency-injection-example-using-spring/), mocking becomes a natural solution for unit testing. You don’t really need a database connection. You just need an object that returns the expected result. If you’ve written tightly coupled code, you will have a difficult time using mocks. I’ve seen plenty of legacy code which could not be unit tested because it was so tightly coupled to other dependent objects. This untestable code did not follow the SOLID Principles of Object Oriented Programming, nor did it utilize Dependency Injection.

Mock Objects: Introduction

In unit test, a test double is a replacement of a dependent component (collaborator) of the object under test. A test double provides the same interface as of the collaborator. It may not be the complete interface, but for the functionality required for the test. Also, the test double does not have to behave exactly as the collaborator. The purpose is to mimic the collaborator to make the object under test think that it is actually using the collaborator.

Based on the role played during testing, there can be different types of test doubles, and mock object is one of them. Some other types are dummy object, fake object, and stub.

What makes a mock object different from the others is that it uses behavior verification. It means that the mock object verifies that *it (the mock object) is being used correctly by the object under test*. If the verification succeeds, it can be considered that the object under test will correctly use the real collaborator.

**The Test Scenario**

For the test scenario, consider a product ordering service. A client interacts with a DAO to fulfill a product ordering process.

We will start with the Product domain object and the DAO interface, ProductDao.

**Product.java**

package guru.springframework.unittest.mockito;

public class Product {

}

package guru.springframework.unittest.mockito;

public class Product {

}

**ProductDao.java**

package guru.springframework.unittest.mockito;

public interface ProductDao {

int getAvailableProducts(Product product);

int orderProduct(Product product, int orderedQuantity);

}

package guru.springframework.unittest.mockito;

public interface ProductDao {

int getAvailableProducts(Product product);

int orderProduct(Product product, int orderedQuantity);

}

For the purpose of the example, I kept the Product class empty. But in real applications, it will typically be an entity with states having corresponding getter and setter methods, along with any implemented behaviors.

In the ProductDao interface, we declared two methods:

The getAvailableProducts() method returns the number of available quantity of a Product passed to it.

The orderProduct() places an order for a product.

The ProductService class that we will write next is what we are interested on – the object under test.

ProductService.java

Java

package guru.springframework.unittest.mockito;

public class ProductService {

private ProductDao productDao;

public void setProductDao(ProductDao productDao) {

this.productDao = productDao;

}

public boolean buy(Product product, int orderedQuantity) throws InsufficientProductsException {

boolean transactionStatus=false;

int availableQuantity = productDao.getAvailableProducts(product);

if (orderedQuantity > availableQuantity) {

throw new InsufficientProductsException();

}

productDao.orderProduct(product, orderedQuantity);

transactionStatus=true;

return transactionStatus;

}

}

package guru.springframework.unittest.mockito;

public class ProductService {

private ProductDao productDao;

public void setProductDao(ProductDao productDao) {

this.productDao = productDao;

}

public boolean buy(Product product, int orderedQuantity) throws InsufficientProductsException {

boolean transactionStatus=false;

int availableQuantity = productDao.getAvailableProducts(product);

if (orderedQuantity > availableQuantity) {

throw new InsufficientProductsException();

}

productDao.orderProduct(product, orderedQuantity);

transactionStatus=true;

return transactionStatus;

}

}

The ProductService class above is composed of ProductDao, which is initialized through a setter method. In the buy() method, we called getAvailableProducts() of ProductDao to check if sufficient quantity of the specified product is available. If not, an exception of type InsufficientProductsException is thrown. If sufficient quantity is available, we called the orderProduct() method of ProductDao.

What we now need is to unit test ProductService. But as you can see, ProductService is composed of ProductDao, whose implementations we don’t have yet. It can be a Spring Data JPA implementation retrieving data from a remote database, or an implementation that communicates with a Web service hosting a cloud-based repository – We don’t know. Even if we have an implementation, we will use it later during integration testing, one of the software testing type I wrote earlier on. But now, we are not interested on any external implementations in this unit test.

In unit tests, we should not to be bothered what the implementation is doing. What we want is to test that our ProductService is behaving as expected and that it is able to correctly use its collaborators. For that, we will mock ProductDao and Product using Mockito.

The ProductService class also throws a custom exception, InsufficientProductsException. The code of the exception class is this.

InsufficientProductsException.java

Java

package guru.springframework.unittest.mockito;

public class InsufficientProductsException extends Exception {

private static final long serialVersionUID = 1L;

private String message = null;

public InsufficientProductsException() { super(); }

public InsufficientProductsException(String message) {

super(message);

this.message = message;

}

public InsufficientProductsException(Throwable cause)

{

super(cause);

}

@Override

public String toString() {

return message;

}

}

package guru.springframework.unittest.mockito;

public class InsufficientProductsException extends Exception {

private static final long serialVersionUID = 1L;

private String message = null;

public InsufficientProductsException() { super(); }

public InsufficientProductsException(String message) {

super(message);

this.message = message;

}

public InsufficientProductsException(Throwable cause)

{

super(cause);

}

@Override

public String toString() {

return message;

}

}

Using Mockito

Mockito is a mocking framework for unit tests written in Java. It is an open source framework available at github. You can use Mockito with JUnit to create and use mock objects during unit testing. To start using Mockito, download the JAR file and place it in your project class. If you are using Maven, you need to add its dependency in the pom.xml file, as shown below.

pom.xml

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>guru.springframework.unittest.quickstart</groupId>

<artifactId>unittest</artifactId>

<packaging>jar</packaging>

<version>1.0-SNAPSHOT</version>

<name>unittest</name>

<url>http://maven.apache.org</url>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.hamcrest</groupId>

<artifactId>hamcrest-library</artifactId>

<version>1.3</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-all</artifactId>

<version>1.9.5</version>

</dependency>

</dependencies>

</project>

Once you have set up the required dependencies, you can start using Mockito. But, before we start any unit tests with mocks, let’s have a quick overview of the key mocking concepts.

Mock Object Creation

For our example, it’s apparent that we need to mock ProductDao and Product. The simplest way is through calls to the mock() method of the Mockito class. The nice thing about Mockito is that it allows creating mock objects of both interfaces and classes without forcing any explicit declarations.

**MockCreationTest.java**

package guru.springframework.unittest.mockito;

import org.junit.Before;

import org.junit.Test;

import static org.junit.Assert.\*;

import static org.mockito.Mockito.\*;

public class MockCreationTest {

private ProductDao productDao;

private Product product;

@Before

public void setupMock() {

product = mock(Product.class);

productDao = mock(ProductDao.class);

}

@Test

public void testMockCreation(){

assertNotNull(product);

assertNotNull(productDao);

}

}

package guru.springframework.unittest.mockito;

import org.junit.Before;

import org.junit.Test;

import static org.junit.Assert.\*;

import static org.mockito.Mockito.\*;

public class MockCreationTest {

private ProductDao productDao;

private Product product;

@Before

public void setupMock() {

product = mock(Product.class);

productDao = mock(ProductDao.class);

}

@Test

public void testMockCreation(){

assertNotNull(product);

assertNotNull(productDao);

}

}

An alternative way is to use the @Mock annotation. When you use it, you will need to initialize the mocks with a call to MockitoAnnotations.initMocks(this) or specify MockitoJUnitRunner as the JUnit test runner as @RunWith(MockitoJUnitRunner.class).

**MockCreationAnnotationTest.java**

package guru.springframework.unittest.mockito;

import org.junit.Before;

import org.junit.Test;

import static org.junit.Assert.\*;

import org.mockito.Mock;

import org.mockito.MockitoAnnotations;

public class MockCreationAnnotationTest {

@Mock

private ProductDao productDao;

@Mock

private Product product;

@Before

public void setupMock() {

MockitoAnnotations.initMocks(this);

}

@Test

public void testMockCreation(){

assertNotNull(product);

assertNotNull(productDao);

}

}

package guru.springframework.unittest.mockito;

import org.junit.Before;

import org.junit.Test;

import static org.junit.Assert.\*;

import org.mockito.Mock;

import org.mockito.MockitoAnnotations;

public class MockCreationAnnotationTest {

@Mock

private ProductDao productDao;

@Mock

private Product product;

@Before

public void setupMock() {

MockitoAnnotations.initMocks(this);

}

@Test

public void testMockCreation(){

assertNotNull(product);

assertNotNull(productDao);

}

}

Stubbing

Stubbing means simulating the behavior of a mock object’s method. We can stub a method on a mock object by setting up an expectation on the method invocation. For example, we can stub the getAvailableProducts() method of the ProductDao mock to return a specific value when the method is called.

@Test

public void testBuy() throws InsufficientProductsException {

when(productDao.getAvailableProducts(product)).thenReturn(30);

assertEquals(30,productDao.getAvailableProducts(product));

}

@Test

public void testBuy() throws InsufficientProductsException {

when(productDao.getAvailableProducts(product)).thenReturn(30);

assertEquals(30,productDao.getAvailableProducts(product));

}

In Line 4 of the code above, we are stubbing getAvailableProducts(product) of ProductDao to return 30. The when() method represents the trigger to start the stubbing and thenReturn() represents the action of the trigger – which in the example code is to return the value 30. In Line 5 with an assertion, we confirmed that the stubbing performed as expected.

**Verifying**

Our objective is to test ProductService, and unitl now we only mocked Product and ProductDao and stubbed getAvailableProducts() of ProductDao.

We now want to verify the behavior of the buy() method of ProductService. First, we want to verify whether it’s calling the orderProduct() of ProductDao with the required set of parameters.

@Test

public void testBuy() throws InsufficientProductsException {

when(productDao.getAvailableProducts(product)).thenReturn(30);

assertEquals(30,productDao.getAvailableProducts(product));

productService.buy(product, 5);

verify(productDao).orderProduct(product, 5);

}

@Test

public void testBuy() throws InsufficientProductsException {

when(productDao.getAvailableProducts(product)).thenReturn(30);

assertEquals(30,productDao.getAvailableProducts(product));

productService.buy(product, 5);

verify(productDao).orderProduct(product, 5);

}

In Line 6 we called the buy() method of ProductService that is under test. In Line 7, we verified that the orderProduct() method of the ProductDao mock get’s invoked with the expected set of parameters (that we passed to buy()).

Our test passed. But, not complete yet. We also want to verify:

Number of invocations done on a method: The buy() method invokes getAvailableProduct() at least once.

Sequence of Invocation: The buy() method first invokes getAvailableProduct(), and then orderProduct().

Exception verification: The buy() method fails with InsufficientProductsException if order quantity passed to it is more than the available quantity returned by getAvailableProduct().

Behavior during exception: The buy() method doesn’t invokes orderProduct() when InsufficientProductsException is thrown.

Here is the complete test code.

**ProductServiceTest.java**

package guru.springframework.unittest.mockito;

import org.junit.Before;

import org.junit.Test;

import org.mockito.InOrder;

import static org.mockito.Mockito.\*;

import org.mockito.Mock;

public class ProductServiceTest {

private ProductService productService;

private ProductDao productDao;

private Product product;

private int purchaseQuantity = 15;

@Before

public void setupMock() {

productService = new ProductService();

product = mock(Product.class);

productDao = mock(ProductDao.class);

productService.setProductDao(productDao);

}

@Test

public void testBuy() throws InsufficientProductsException {

int availableQuantity = 30;

System.out.println("Stubbing getAvailableProducts(product) to return " + availableQuantity);

when(productDao.getAvailableProducts(product)).thenReturn(availableQuantity);

System.out.println("Calling ProductService.buy(product," + purchaseQuantity + ")");

productService.buy(product, purchaseQuantity);

System.out.println("Verifying ProductDao(product, " + purchaseQuantity + ") is called");

verify(productDao).orderProduct(product, purchaseQuantity);

System.out.println("Verifying getAvailableProducts(product) is called at least once");

verify(productDao, atLeastOnce()).getAvailableProducts(product);

System.out.println("Verifying order of method calls on ProductDao: First call getAvailableProducts() followed by orderProduct()");

InOrder order = inOrder(productDao);

order.verify(productDao).getAvailableProducts(product);

order.verify(productDao).orderProduct(product, purchaseQuantity);

}

@Test(expected = InsufficientProductsException.class)

public void purchaseWithInsufficientAvailableQuantity() throws InsufficientProductsException {

int availableQuantity = 3;

System.out.println("Stubbing getAvailableProducts(product) to return " + availableQuantity);

when(productDao.getAvailableProducts(product)).thenReturn(availableQuantity);

try {

System.out.println("productService.buy(product" + purchaseQuantity + ") should throw InsufficientProductsException");

productService.buy(product, purchaseQuantity);

} catch (InsufficientProductsException e) {

System.out.println("InsufficientProductsException has been thrown");

verify(productDao, times(0)).orderProduct(product, purchaseQuantity);

System.out.println("Verified orderProduct(product, " + purchaseQuantity + ") is not called");

throw e;

}

}

}

package guru.springframework.unittest.mockito;

import org.junit.Before;

import org.junit.Test;

import org.mockito.InOrder;

import static org.mockito.Mockito.\*;

import org.mockito.Mock;

public class ProductServiceTest {

private ProductService productService;

private ProductDao productDao;

private Product product;

private int purchaseQuantity = 15;

@Before

public void setupMock() {

productService = new ProductService();

product = mock(Product.class);

productDao = mock(ProductDao.class);

productService.setProductDao(productDao);

}

@Test

public void testBuy() throws InsufficientProductsException {

int availableQuantity = 30;

System.out.println("Stubbing getAvailableProducts(product) to return " + availableQuantity);

when(productDao.getAvailableProducts(product)).thenReturn(availableQuantity);

System.out.println("Calling ProductService.buy(product," + purchaseQuantity + ")");

productService.buy(product, purchaseQuantity);

System.out.println("Verifying ProductDao(product, " + purchaseQuantity + ") is called");

verify(productDao).orderProduct(product, purchaseQuantity);

System.out.println("Verifying getAvailableProducts(product) is called at least once");

verify(productDao, atLeastOnce()).getAvailableProducts(product);

System.out.println("Verifying order of method calls on ProductDao: First call getAvailableProducts() followed by orderProduct()");

InOrder order = inOrder(productDao);

order.verify(productDao).getAvailableProducts(product);

order.verify(productDao).orderProduct(product, purchaseQuantity);

}

@Test(expected = InsufficientProductsException.class)

public void purchaseWithInsufficientAvailableQuantity() throws InsufficientProductsException {

int availableQuantity = 3;

System.out.println("Stubbing getAvailableProducts(product) to return " + availableQuantity);

when(productDao.getAvailableProducts(product)).thenReturn(availableQuantity);

try {

System.out.println("productService.buy(product" + purchaseQuantity + ") should throw InsufficientProductsException");

productService.buy(product, purchaseQuantity);

} catch (InsufficientProductsException e) {

System.out.println("InsufficientProductsException has been thrown");

verify(productDao, times(0)).orderProduct(product, purchaseQuantity);

System.out.println("Verified orderProduct(product, " + purchaseQuantity + ") is not called");

throw e;

}

}

}

I have already explained the initial code of the test class above. So we will start with Line 36 – Line 38 where we used the inOrder() method to verify the order of method invocation that the buy() method makes on ProductDao.

Then we wrote a purchaseWithInsufficientAvailableQuantity() test method to check whether an InsufficientProductsException gets thrown, as expected, when an order with quantity more than the available quantity is made. We also verified in Line 54 that if InsufficientProductsException gets thrown, the orderProduct() method is not invoked.

The output of the test is this.

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

T E S T S

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

Running guru.springframework.unittest.mockito.ProductServiceTest

Stubbing getAvailableProducts(product) to return 30

Calling ProductService.buy(product,15)

Verifying ProductDao(product, 15) is called

Verifying getAvailableProducts(product) is called at least once

Verifying order of method calls on ProductDao: First call getAvailableProducts() followed by orderProduct()

Stubbing getAvailableProducts(product) to return 3

productService.buy(product15) should throw InsufficientProductsException

InsufficientProductsException has been thrown

Verified orderProduct(product, 15) is not called

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.077 sec

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

T E S T S

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

Running guru.springframework.unittest.mockito.ProductServiceTest

Stubbing getAvailableProducts(product) to return 30

Calling ProductService.buy(product,15)

Verifying ProductDao(product, 15) is called

Verifying getAvailableProducts(product) is called at least once

Verifying order of method calls on ProductDao: First call getAvailableProducts() followed by orderProduct()

Stubbing getAvailableProducts(product) to return 3

productService.buy(product15) should throw InsufficientProductsException

InsufficientProductsException has been thrown

Verified orderProduct(product, 15) is not called

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.077 sec

# Mockito - Varying Calls

Mockito provides the following additional methods to vary the expected call counts.

* **atLeast (int min)** − expects min calls.
* **atLeastOnce ()** − expects at least one call.
* **atMost (int max)** − expects max calls.

## Example

**Step 1 − Create an interface 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 Mockito.

***File: MathApplicationTester.java***

import static org.mockito.Mockito.verify;

import static org.mockito.Mockito.when;

import static org.mockito.Mockito.atLeastOnce;

import static org.mockito.Mockito.atLeast;

import static org.mockito.Mockito.atMost;

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);

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

when(calcService.subtract(20.0,10.0)).thenReturn(10.00);

//test the add functionality

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

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

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

//test the subtract functionality

Assert.assertEquals(mathApplication.subtract(20.0, 10.0),10.0,0.0);

//check a minimum 1 call count

verify(calcService, atLeastOnce()).subtract(20.0, 10.0);

//check if add function is called minimum 2 times

verify(calcService, atLeast(2)).add(10.0, 20.0);

//check if add function is called maximum 3 times

verify(calcService, atMost(3)).add(10.0,20.0);

}

}

**Step 4 − Execute test cases**

Create a java class file named TestRunner in **C:\> Mockito\_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());

}

}

**Step 5 − Verify the Result**

Compile the classes using **javac** compiler as follows −

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 - Exception Handling

Mockito provides the capability to a mock to throw exceptions, so exception handling can be tested. Take a look at the following code snippet.

//add the behavior to throw exception

doThrow(new Runtime Exception("divide operation not implemented"))

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

Here we've added an exception clause to a mock object. MathApplication makes use of calcService using its add method and the mock throws a RuntimeException whenever calcService.add() method is invoked.

## Example

**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 Mockito.

***File: MathApplicationTester.java***

import static org.mockito.Mockito.doThrow;

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(MockitoRunner.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(expected = RuntimeException.class)

public void testAdd(){

//add the behavior to throw exception

doThrow(new RuntimeException("Add operation not implemented"))

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

//test the add functionality

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

}

}

**Step 4 − Execute test cases**

Create a java class file named TestRunner in **C:\> Mockito\_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());

}

}

**Step 5 − Verify the Result**

Compile the classes using **javac** compiler as follows −

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): Add operation not implemented

false

# Mockito - Ordered Verification

Mockito provides Inorder class which takes care of the order of method calls that the mock is going to make in due course of its action.

## Syntax

//create an inOrder verifier for a single mock

InOrder inOrder = inOrder(calcService);

//following will make sure that add is first called then subtract is called.

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

inOrder.verify(calcService).subtract(20.0,10.0);

## Example

**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 Mockito.

Here we've added two mock method calls, add() and subtract(), to the mock object via when(). However during testing, we've called subtract() before calling add(). When we create a mock object using Mockito, the order of execution of the method does not matter. Using InOrder class, we can ensure call order.

***File: MathApplicationTester.java***

import static org.mockito.Mockito.mock;

import static org.mockito.Mockito.verify;

import static org.mockito.Mockito.when;

import static org.mockito.Mockito.inOrder;

import org.junit.Assert;

import org.junit.Before;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.mockito.InOrder;

import org.mockito.runners.MockitoJUnitRunner;

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

@RunWith(MockitoJUnitRunner.class)

public class MathApplicationTester {

private MathApplication mathApplication;

private CalculatorService calcService;

@Before

public void setUp(){

mathApplication = new MathApplication();

calcService = mock(CalculatorService.class);

mathApplication.setCalculatorService(calcService);

}

@Test

public void testAddAndSubtract(){

//add the behavior to add numbers

when(calcService.add(20.0,10.0)).thenReturn(30.0);

//subtract the behavior to subtract numbers

when(calcService.subtract(20.0,10.0)).thenReturn(10.0);

//test the add functionality

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

//test the subtract functionality

Assert.assertEquals(mathApplication.subtract(20.0, 10.0),10.0,0);

//create an inOrder verifier for a single mock

InOrder inOrder = inOrder(calcService);

//following will make sure that add is first called then subtract is called.

inOrder.verify(calcService).subtract(20.0,10.0);

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

}

}

**Step 4 − Execute test cases**

Create a java class file named TestRunner in **C:\> Mockito\_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());

}

}

**Step 5 − Verify the Result**

Compile the classes using **javac** compiler as follows −

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.

testAddAndSubtract(MathApplicationTester):

Verification in order failure

Wanted but not invoked:

calculatorService.add(20.0, 10.0);

-> at MathApplicationTester.testAddAndSubtract(MathApplicationTester.java:48)

Wanted anywhere AFTER following interaction:

calculatorService.subtract(20.0, 10.0);

-> at MathApplication.subtract(MathApplication.java:13)

false

# Mockito - Spying

Mockito provides option to create spy on real objects. When spy is called, then actual method of real object is called.

## Syntax

//create a spy on actual object

calcService = spy(calculator);

//perform operation on real object

//test the add functionality

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

## Example

**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 Mockito.

Here we've added one mock method calls, add() to the mock object via when(). However during testing, we've called subtract() before calling add(). When we create a mock object using Mockito.createStrictMock(), the order of execution of the method does matter.

***File: MathApplicationTester.java***

import static org.mockito.Mockito.spy;

import org.junit.Assert;

import org.junit.Before;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.mockito.runners.MockitoJUnitRunner;

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

@RunWith(MockitoJUnitRunner.class)

public class MathApplicationTester {

private MathApplication mathApplication;

private CalculatorService calcService;

@Before

public void setUp(){

mathApplication = new MathApplication();

Calculator calculator = new Calculator();

calcService = spy(calculator);

mathApplication.setCalculatorService(calcService);

}

@Test

public void testAdd(){

//perform operation on real object

//test the add functionality

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

}

class Calculator implements CalculatorService {

@Override

public double add(double input1, double input2) {

return input1 + input2;

}

@Override

public double subtract(double input1, double input2) {

throw new UnsupportedOperationException("Method not implemented yet!");

}

@Override

public double multiply(double input1, double input2) {

throw new UnsupportedOperationException("Method not implemented yet!");

}

@Override

public double divide(double input1, double input2) {

throw new UnsupportedOperationException("Method not implemented yet!");

}

}

}

**Step 4 − Execute test cases**

Create a java class file named TestRunner in **C:\> Mockito\_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());

}

}

**Step 5 − Verify the Result**

Compile the classes using **javac** compiler as follows −

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 - Behavior Driven Development

Behavior Driven Development is a style of writing tests uses **given**, **when** and **then** format as test methods. Mockito provides special methods to do so. Take a look at the following code snippet.

//Given

given(calcService.add(20.0,10.0)).willReturn(30.0);

//when

double result = calcService.add(20.0,10.0);

//then

Assert.assertEquals(result,30.0,0);

Here we're using **given** method of BDDMockito class instead of **when** method of .

## Example

**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 Mockito.

***File: MathApplicationTester.java***

package com.tutorialspoint.mock;

import static org.mockito.BDDMockito.\*;

import org.junit.Assert;

import org.junit.Before;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.mockito.runners.MockitoJUnitRunner;

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

@RunWith(MockitoJUnitRunner.class)

public class MathApplicationTester {

private MathApplication mathApplication;

private CalculatorService calcService;

@Before

public void setUp(){

mathApplication = new MathApplication();

calcService = mock(CalculatorService.class);

mathApplication.setCalculatorService(calcService);

}

@Test

public void testAdd(){

//Given

given(calcService.add(20.0,10.0)).willReturn(30.0);

//when

double result = calcService.add(20.0,10.0);

//then

Assert.assertEquals(result,30.0,0);

}

}

**Step 4 − Execute test cases**

Create a java class file named TestRunner in **C:\> Mockito\_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());

}

}

**Step 5 − Verify the Result**

Compile the classes using **javac** compiler as follows −

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