tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com

About the Tutorial

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.

This tutorial should help you learn how to create unit tests with Mockito as well as how to use its APIs in a simple and intuitive way.

Audience

This tutorial is meant for Java developers, from novice to expert level, who would like to improve the quality of their software through unit testing and test-driven development.

After completing this tutorial, you should gain sufficient exposure to Mockito from where you can take yourself to next levels of expertise.

Prerequisites

Readers must have a working knowledge of JAVA programming language in order to make the most of this tutorial. Knowledge of JUnit is an added advantage.

Copyright & Disclaimer

Table of Contents

1. Mockito – Overview

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.

Consider the following code snippet.

```
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);
   }
}
```

Let's understand the important concepts of the above program. The complete code is available in the chapter ***First Application***.

- **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.

2. Mockito – Environment Setup

Mockito is a framework for Java, so the very first requirement is to have JDK installed in your machine.

System Requirement

JDK	1.5 or above.
Memory	no minimum requirement.
Disk Space	no minimum requirement.
Operating System	no minimum requirement.

Step 1: Verify Java Installation on Your Machine

Open the console and execute the following **java** command.

OS	Task	Command
Windows	Open Command Console	c:\> java -version
Linux	Open Command Terminal	$ java -version
Mac	Open Terminal	machine:> joseph$ java -version

Let's verify the output for all the operating systems:

OS	Output
Windows	java version "1.6.0_21" Java(TM) SE Runtime Environment (build 1.6.0_21-b07) Java HotSpot(TM) Client VM (build 17.0-b17, mixed mode, sharing)
Linux	java version "1.6.0_21" Java(TM) SE Runtime Environment (build 1.6.0_21-b07) Java HotSpot(TM) Client VM (build 17.0-b17, mixed mode, sharing)
Mac	java version "1.6.0_21" Java(TM) SE Runtime Environment (build 1.6.0_21-b07)

	Java HotSpot(TM)64-Bit Server VM (build 17.0-b17, mixed mode, sharing)

If you do not have Java installed, To install the Java Software Development Kit (SDK) click here.

We assume you have Java 1.6.0_21 installed on your system for this tutorial.

Step 2: Set JAVA Environment

Set the **JAVA_HOME** environment variable to point to the base directory location where Java is installed on your machine. For example,

OS	Output
Windows	Set the environment variable JAVA_HOME to C:\Program Files\Java\jdk1.6.0_21
Linux	export JAVA_HOME=/usr/local/java-current
Mac	export JAVA_HOME=/Library/Java/Home

Append the location of the Java compiler to your System Path.

OS	Output
Windows	Append the string ;C:\Program Files\Java\jdk1.6.0_21\bin to the end of the system variable, Path.
Linux	export PATH=$PATH:$JAVA_HOME/bin/
Mac	not required

Verify Java Installation using the command **java -version** as explained above.

Step 3: Download Mockito-All Archive

To download the latest version of Mockito from Maven Repository click here.

Save the jar file on your C drive, let's say, C:\>Mockito.

OS	Archive name
Windows	mockito-all-2.0.2-beta.jar
Linux	mockito-all-2.0.2-beta.jar
Mac	mockito-all-2.0.2-beta.jar

Step 4: Set Mockito Environment

Set the **Mockito_HOME** environment variable to point to the base directory location where Mockito and dependency jars are stored on your machine.

The following table shows how to set the environment variable on different operating systems, assuming we've extracted mockito-all-2.0.2-beta.jar onto C:\>Mockito folder.

OS	Output
Windows	Set the environment variable Mockito_HOME to C:\Mockito
Linux	export Mockito_HOME=/usr/local/Mockito
Mac	export Mockito_HOME=/Library/Mockito

Step 5: Set CLASSPATH Variable

Set the **CLASSPATH** environment variable to point to the location where Mockito jar is stored. The following table shows how to set the CLASSPATH variable on different operating systems.

OS	Output
Windows	Set the environment variable CLASSPATH to %CLASSPATH%;%Mockito_HOME%\mockito-all-2.0.2-beta.jar;.;
Linux	export CLASSPATH=$CLASSPATH:$Mockito_HOME/mockito-all-2.0.2-beta.jar:.
Mac	export CLASSPATH=$CLASSPATH:$Mockito_HOME/mockito-all-2.0.2-beta.jar:.

Step 6: Download JUnit Archive

Download the latest version of JUnit jar file from Github. Save the folder at the location C:\>Junit.

OS	Archive name
Windows	junit4.11.jar, hamcrest-core-1.2.1.jar
Linux	junit4.11.jar, hamcrest-core-1.2.1.jar
Mac	junit4.11.jar, hamcrest-core-1.2.1.jar

Step 7: Set JUnit Environment

Set the **JUNIT_HOME** environment variable to point to the base directory location where JUnit jars are stored on your machine.

The following table shows how to set this environment variable on different operating systems, assuming we've stored junit4.11.jar and hamcrest-core-1.2.1.jar at C:\>Junit.

OS	Output
Windows	Set the environment variable JUNIT_HOME to C:\JUNIT
Linux	export JUNIT_HOME=/usr/local/JUNIT
Mac	export JUNIT_HOME=/Library/JUNIT

Step 8: Set CLASSPATH Variable

Set the CLASSPATH environment variable to point to the JUNIT jar location. The following table shows how it is done on different operating systems.

OS	Output
Windows	Set the environment variable CLASSPATH to %CLASSPATH%;%JUNIT_HOME%\junit4.11.jar; %JUNIT_HOME%\hamcrest-core-1.2.1.jar;.;
Linux	export CLASSPATH=$CLASSPATH:$JUNIT_HOME/junit4.11.jar:$JUNIT_HOME/hamcrest-core-1.2.1.jar:.
Mac	export CLASSPATH=$CLASSPATH:$JUNIT_HOME/junit4.11.jar:$JUNIT_HOME/hamcrest-core-1.2.1.jar:.

3. Mockito - First Application

Before going into the details of the Mockito Framework, let's see an application in action. In this example, we've created a mock of Stock Service to get the dummy price of some stocks and unit tested a java class named Portfolio.

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

Step 1: Create a JAVA class to represent the Stock

File: Stock.java

```
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;
   }
}
```

Step 2: Create an interface StockService to get the price of a stock

File: StockService.java

```
public interface StockService {
   public double getPrice(Stock stock);
}
```

Step 3: Create a class Portfolio to represent the portfolio of any client

File: Portfolio.java

```
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;
   }
}
```

Step 4: Test the Portfolio class

Let's test the Portfolio class, by injecting in it a mock of stockservice. Mock will be created by Mockito.

File: PortfolioTester.java

```
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;
   }
}
```


Step 5: Verify the result

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

```
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
```


4. Mockito - JUnit Integration

In this chapter, we'll learn how to integrate JUnit and Mockito together. Here we will create a Math Application which uses CalculatorService to perform basic mathematical operations such as addition, subtraction, multiply, and division.

We'll use Mockito to mock the dummy implementation of CalculatorService. In addition, we've made extensive use of annotations to showcase their compatibility with both JUnit and Mockito.

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

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

File: CalculatorService.java

```
public interface CalculatorService {
   public double add(double input1, double input2);
   public double subtract(double input1, double input2);
   public double multiply(double input1, double input2);
   public double divide(double input1, double input2);
}
```

Step 2: Create a JAVA class to represent MathApplication

File: MathApplication.java

```
public class MathApplication {
   private CalculatorService calcService;

   public void setCalculatorService(CalculatorService calcService){
      this.calcService = calcService;
   }

   public double add(double input1, double input2){
      return calcService.add(input1, input2);
   }
```


```
   public double subtract(double input1, double input2){
      return calcService.subtract(input1, input2);
   }

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

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

Step 3: Test the MathApplication class

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

File: MathApplicationTester.java

```
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);
   }
}
```

Step 4: Create a class to execute to 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
```

To learn more about JUnit, please refer to JUnit Tutorial at Tutorials Point.

Mockito adds a functionality to a mock object using the method **when()**. Take a look at the following code snippet.

```
//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.

At this point of time, Mock recorded the behavior and is a working mock object.

```
//add the behavior of calc service to add two numbers
when(calcService.add(10.0,20.0)).thenReturn(30.00);
```

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.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);
   }
}
```

Step 4: Execute test cases

Create a java class file named TestRunner in **C:\>Mockito_WORKSPACE** to execute the 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 can ensure whether a mock method is being called with reequired arguments or not. It is done using the **verify()** method. Take a look at the following code snippet.

```
//test the add functionality
Assert.assertEquals(mathApplication.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

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);
      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);
   }
}
```

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 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);

      //verify the behavior
      verify(calcService).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
```

Example - verify() with different arguments

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);
      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);
   }
}
```

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 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);

      //verify the behavior
      verify(calcService).add(20.0, 30.0);
   }
}
```

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):
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
```


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);
```

Create CalculatorService interface as follows.

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);
}
```

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.verify;
import static org.mockito.Mockito.when;
import static org.mockito.Mockito.times;
import static org.mockito.Mockito.never;
```


```
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);
```


```
      //default call count is 1
      verify(calcService).subtract(20.0, 10.0);

      //check if add function is called three times
      verify(calcService, times(3)).add(10.0, 20.0);

      //verify that method was never called on a mock
      verify(calcService, never()).multiply(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 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 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
```


10. Mockito – Create Mock

So far, we've used annotations to create mocks. Mockito provides various methods to create mock objects. mock() creates mocks without bothering about the order of method calls that the mock is going to make in due course of its action.

Syntax

```
calcService = mock(CalculatorService.class);
```

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 create(), the order of execution of the method does not matter.

File: MathApplicationTester.java

```
package com.tutorialspoint.mock;

import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.when;

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

      //test the add functionality
      Assert.assertEquals(mathApplication.add(20.0, 10.0),30.0,0);

      //verify call to calcService is made or not
      verify(calcService).add(20.0,10.0);
      verify(calcService).subtract(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.

```
true
```


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 provides an Answer interface which allows stubbing with generic interface.

Syntax

```
//add the behavior to add numbers
when(calcService.add(20.0,10.0)).thenAnswer(new Answer() {
   @Override
   public Double answer(InvocationOnMock invocation) throws Throwable {
      //get the arguments passed to mock
      Object[] args = invocation.getArguments();
      //get the mock
      Object mock = invocation.getMock();
      //return the result
      return 30.0;
   }
});
```

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.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 testAdd(){

      //add the behavior to add numbers
      when(calcService.add(20.0,10.0)).thenAnswer(new Answer() {

      @Override
      public Double answer(InvocationOnMock invocation) throws Throwable {
            //get the arguments passed to mock
            Object[] args = invocation.getArguments();
```


```
            //get the mock
            Object mock = invocation.getMock();
            //return the result
            return 30.0;
         }
      });

      //test the add functionality
      Assert.assertEquals(mathApplication.add(20.0, 10.0),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.

```
true
```


Mockito provides an option to create spy on real objects. When a spy is called, then the actual method of the 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
```


14. Mockito - Resetting a Mock

Mockito provides the capability to reset a mock, so that it can be reused later. Take a look at the following code snippet.

```
//reset mock
reset(calcService);
```

Here we've reset a mock object. MathApplication makes use of calcService. Once you reset the mock, the mocked method will fail the test.

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.Mockito.mock;
import static org.mockito.Mockito.when;
import static org.mockito.Mockito.reset;

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 testAddAndSubtract(){

      //add the behavior to add numbers
      when(calcService.add(20.0,10.0)).thenReturn(30.0);

      //test the add functionality
      Assert.assertEquals(mathApplication.add(20.0, 10.0),30.0,0);

      //reset the mock
      reset(calcService);

      //test the add functionality after resetting the mock
      Assert.assertEquals(mathApplication.add(20.0, 10.0),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.

```
testAddAndSubtract(MathApplicationTester): expected:<0.0> but was:<30.0>
false
```


Behavior Driven Development is a style of writing tests using **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 the **given** method of BDDMockito class instead of **when** method.

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
```


Mockito provides a special Timeout option to test if a method is called within its stipulated time frame.

Syntax

```
//passes when add() is called within 100 ms.
verify(calcService,timeout(100)).add(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.

File: MathApplicationTester.java

```
package com.tutorialspoint.mock;

import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.when;

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

      //test the add functionality
      Assert.assertEquals(mathApplication.add(20.0, 10.0),30.0,0);

      //verify call to add method to be completed within 100 ms
      verify(calcService, timeout(100)).add(20.0,10.0);

      //invocation count can be added to ensure multiplication invocations
      //can be checked within given timeframe
      verify(calcService, timeout(100).times(1)).subtract(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.

```
true
```