# EASYMOCK

object-oriented programming

tutorialspoint

SIMPLYEASYLEARNING

www.tutorialspoint.com


https://www.facebook.com/tutorialspointindia


https://twitter.com/tutorialspoint

# About the Tutorial

EasyMock is a mocking framework, JAVA-based library that is used for effective unit testing of JAVA applications. EasyMock 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 will help you learn how to create unit tests with EasyMock 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 will gain sufficient exposure to EasyMock 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 best of this tutorial. Knowledge of JUnit is an added advantage.

# Copyright & Disclaimer

# Table of Contents

# ① 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.

## EasyMock

EasyMock 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. EasyMock can do the same very easily as its name suggests.

### Benefits of EasyMock

- **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 org.easymock.EasyMock;

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 =
      EasyMock.createMock(StockService.class);

      // mock the behavior of stock service to return
      // the value of various stocks
      EasyMock.expect(stockServiceMock.getPrice(googleStock))
      .andReturn(50.00);
      EasyMock.expect(stockServiceMock.getPrice(microsoftStock))
      .andReturn(1000.00);
```

```
      EasyMock.replay(stockServiceMock);


      //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.
- **EasyMock.createMock(...)** – EasyMock created a mock of stock service.
- **EasyMock.expect(...).andReturn(...)** – Mock implementation of getPrice method of stockService interface. For googleStock, return 50.00 as price.
- **EasyMock.replay(...)** – EasyMock prepares the Mock object to be ready so that it can be used for testing.
- **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 ENVIRONMENT SETUP

EasyMock 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"<br>Java(TM) SE Runtime Environment (build 1.6.0_21-b07)<br>Java HotSpot(TM) Client VM (build 17.0-b17, mixed mode, sharing) |
| Linux | java version "1.6.0_21"<br>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"<br>Java(TM) SE Runtime Environment (build 1.6.0_21-b07)<br>Java HotSpot(TM)64-Bit Server VM (build 17.0-b17, mixed mode, sharing) |

If you do not have Java installed, install the Java Software Development Kit (SDK) from http://www.oracle.com/technetwork/java/javase/downloads/index.html.

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 EasyMock Archive

Download the latest version of EasyMock from http://sourceforge.net/projects/easymock/files/EasyMock/3.2/easymock-3.2.zip/download. Save the zip folder on your C drive, let's say, C:\>EasyMock.

| OS | Archive name |
|---|---|
| Windows | easymock-3.2.zip |
| Linux | easymock-3.2.zip |
| Mac | easymock-3.2.zip |

## Step 4: Download EasyMock Dependencies

Download the latest version of cglib jar file from https://github.com/cglib/cglib/releases and copy it onto C:\>EasyMock folder. At the time of writing this tutorial, the latest version was 3.1.

Download the latest version of objenesis zip file from http://objenesis.org/download.html and copy it onto C:\>EasyMock folder. At the time of writing this tutorial, the latest version was 2.1. Extract objenesis-2.1.jar to C:\>EasyMock folder

## Step 5: Set EasyMock Environment

Set the **EasyMock_HOME** environment variable to point to the base directory location where EasyMock and dependency jars are stored on your machine. The following table shows how to set the environment variable on differnet operating systems, assuming we've extracted easymock-3.2.jar, cglib-3.1.jar, and objenesis-2.1.jar onto C:\>EasyMock folder.

| OS | Output |
|---|---|
| Windows | Set the environment variable EasyMock_HOME to C:\EasyMock |
| Linux | export EasyMock_HOME=/usr/local/EasyMock |
| Mac | export EasyMock_HOME=/Library/EasyMock |

## Step 6: Set CLASSPATH Variable

Set the **CLASSPATH** environment variable to point to the location where EasyMock and dependency jars are 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%;%EasyMock_HOME%\easymock-3.2.jar;%EasyMock_HOME%\cglib-3.1.jar;%EasyMock_HOME%\objenesis-2.1.jar;.; |
| Linux | export CLASSPATH=$CLASSPATH:$EasyMock_HOME/easymock-3.2.jar:$EasyMock_HOME/cglib-3.1.jar:$EasyMock_HOME/objenesis-2.1.jar:. |
| Mac | export CLASSPATH=$CLASSPATH:$EasyMock_HOME/easymock-3.2.jar:$EasyMock_HOME/cglib-3.1.jar:$EasyMock_HOME/objenesis-2.1.jar:. |

## Step 7: Download JUnit Archive

Download the latest version of JUnit jar file from https://github.com/junit-team/junit/wiki/Download-and-Install. 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 8: 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 9: 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 FIRST APPLICATION

Before going into the details of the EasyMock 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**

**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**

**StockService.java**

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

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

**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 EasyMock.

### PortfolioTester.java

```
import java.util.ArrayList;
import java.util.List;
```

```
import org.easymock.EasyMock;
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 = EasyMock.createMock(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
      EasyMock.expect(stockService.getPrice(googleStock))
      .andReturn(50.00);
      EasyMock.expect(stockService.getPrice(microsoftStock))
      .andReturn(1000.00);

      // activate the mock
      EasyMock.replay(stockService);

      double marketValue = portfolio.getMarketValue();
      return marketValue == 100500.0;
   }
}
```

### Step 5: Verify the result

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

```
C:\EasyMock_WORKSPACE>javac Stock.java StockService.java Portfolio.java
PortfolioTester.java
```

Now run the PortfolioTester to see the result:

```
C:\EasyMock_WORKSPACE>java PortfolioTester

pass
```

# ④ JUNIT INTEGRATION

In this chapter, we'll learn how to integrate JUnit and EasyMock together. For JUnit tutorial, please refer to JUnit. 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 EasyMock to mock the dummy implementation of CalculatorService. In addition, we've made extensive use of annotations to showcase their compatibility with both JUnit and EasyMock.

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

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

**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**

**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 EasyMock.

### MathApplicationTester.java

```
import org.easymock.EasyMock;
import org.easymock.EasyMockRunner;
import org.easymock.Mock;
import org.easymock.TestSubject;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;

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

```
@RunWith(EasyMockRunner.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
   public void testAdd(){
      //add the behavior of calc service to add two numbers
      EasyMock.expect(calcService.add(10.0,20.0)).andReturn(30.00);

      //activate the mock
      EasyMock.replay(calcService);

      //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:\ > EasyMock_WORKSPACE** to execute Test case(s).

**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:\EasyMock_WORKSPACE>javac CalculatorService.java MathApplication.java
MathApplicationTester.java TestRunner.java
```

Now run the Test Runner to see the result:

```
C:\EasyMock_WORKSPACE>java TestRunner
```

Verify the output.

```
true
```

# 5 ADDING BEHAVIOR

EasyMock adds a functionality to a mock object using the methods **expect()** and **expectLassCall()**. Take a look at the following code snippet.

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

EasyMock.expect(calcService.add(10.0,20.0)).andReturn(30.00);
```

Here we've instructed EasyMock 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 simply recorded the behavior but it is not working as a mock object. After calling replay, it works as expected.

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

EasyMock.expect(calcService.add(10.0,20.0)).andReturn(30.00);



//activate the mock

//EasyMock.replay(calcService);
```

## Example without EasyMock.Replay()

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

**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**

**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 EasyMock.

**MathApplicationTester.java**

```
import org.easymock.EasyMock;
import org.easymock.EasyMockRunner;
import org.easymock.Mock;
import org.easymock.TestSubject;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;

//@RunWith attaches a runner with the test class to
//initialize the test data
@RunWith(EasyMockRunner.class)
public class MathApplicationTester {

   // @TestSubject annotation is used to identify the 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
   public void testAdd(){
      //add the behavior of calc service to add two numbers
```

```
      EasyMock.expect(calcService.add(10.0,20.0)).andReturn(30.00);


      //activate the mock

      //EasyMock.replay(calcService);


      //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:\>EasyMock_WORKSPACE** to execute the test case(s).

### 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:\EasyMock_WORKSPACE>javac CalculatorService.java MathApplication.java
MathApplicationTester.java TestRunner.java
```

Now run the Test Runner to see the result:

```
C:\EasyMock_WORKSPACE>java TestRunner
```

Verify the output.

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

false
```

# Example with EasyMock.Replay()

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

**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.**

**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 EasyMock.

### MathApplicationTester.java

```
import org.easymock.EasyMock;
import org.easymock.EasyMockRunner;
import org.easymock.Mock;
import org.easymock.TestSubject;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;
```

```
import org.junit.runner.RunWith;

// @RunWith attaches a runner with the test class to
// initialize the test data
@RunWith(EasyMockRunner.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
   public void testAdd(){
      // add the behavior of calc service to add two numbers
      EasyMock.expect(calcService.add(10.0,20.0)).andReturn(30.00);

      //activate the mock
      EasyMock.replay(calcService);

      // 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:\>EasyMock_WORKSPACE** to execute Test case(s).

### 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:\EasyMock_WORKSPACE>javac CalculatorService.java MathApplication.java
MathApplicationTester.java TestRunner.java
```

Now run the Test Runner to see the result.

```
C:\EasyMock_WORKSPACE>java TestRunner
```

Verify the output.

```
true
```

# 6 VERIFYING BEHAVIOR

EasyMock can ensure whether a mock is being used or not. It is done using the verify() method. Take a look at the following code snippet.

```
//activate the mock

EasyMock.replay(calcService);


//test the add functionality

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


//verify call to calcService is made or not

EasyMock.verify(calcService);
```

## Example without EasyMock.Verify()

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

**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

### 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 EasyMock.

**MathApplicationTester.java**

```
import org.easymock.EasyMock;
import org.easymock.EasyMockRunner;
import org.easymock.Mock;
import org.easymock.TestSubject;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;

// @RunWith attaches a runner with the test class to initialize
// the test data
@RunWith(EasyMockRunner.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
   public void testAdd(){
      //add the behavior of calc service to add two numbers
```

```
      EasyMock.expect(calcService.add(10.0,20.0)).andReturn(30.00);

      //activate the mock
      EasyMock.replay(calcService);

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

      //verify call to calcService is made or not
      //EasyMock.verify(calcService);
   }
}
```

### Step 4: Execute test cases

Create a java class file named TestRunner in **C:\> EasyMock_WORKSPACE** to execute Test case(s).

### 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:\EasyMock_WORKSPACE>javac CalculatorService.java MathApplication.java
MathApplicationTester.java TestRunner.java
```

Now run the Test Runner to see the result

```
C:\EasyMock_WORKSPACE>java TestRunner
```

Verify the output.

```
true
```

# Example with EasyMock.Verify()

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

**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**

**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 EasyMock.

### MathApplicationTester.java

```
import org.easymock.EasyMock;
import org.easymock.EasyMockRunner;
import org.easymock.Mock;
import org.easymock.TestSubject;
import org.junit.Assert;
```

```
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;

// @RunWith attaches a runner with the test class to initialize
// the test data
@RunWith(EasyMockRunner.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
   public void testAdd(){
      //add the behavior of calc service to add two numbers
      EasyMock.expect(calcService.add(10.0,20.0)).andReturn(30.00);

      //activate the mock
      EasyMock.replay(calcService);

      //test the add functionality
```

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


      //verify call to calcService is made or not
      EasyMock.verify(calcService);
   }
}
```

### Step 4: Execute test cases

Create a java class file named TestRunner in **C:\> EasyMock_WORKSPACE** to execute Test case(s).

### 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:\EasyMock_WORKSPACE>javac CalculatorService.java MathApplication.java
MathApplicationTester.java TestRunner.java
```

Now run the Test Runner to see the result:

```
C:\EasyMock_WORKSPACE>java TestRunner
```

Verify the output.

```
testAdd(MathApplicationTester):

  Expectation failure on verify:

    CalculatorService.add(10.0, 20.0): expected: 1, actual: 0

false
```

# 7 EXPECTING CALLS

EasyMock 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 and serviceUsed.

EasyMock.expect(calcService.add(10.0,20.0)).andReturn(30.00);

calcService.serviceUsed();

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

EasyMock.expectLastCall().times(1);
```

Create CalculatorService interface as follows.

**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);

   public void serviceUsed();

}
```

## Example with calcService.serviceUsed() called once

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

**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);
   public void serviceUsed();
}
```

**Step 2: Create a JAVA class to represent MathApplication**

**MathApplication.java**

```
public class MathApplication {
   private CalculatorService calcService;

   public void setCalculatorService(CalculatorService calcService){
      this.calcService = calcService;
   }
   public double add(double input1, double input2){
      calcService.serviceUsed();
      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 EasyMock.

### MathApplicationTester.java MathApplicationTester.java

```
import org.easymock.EasyMock;
import org.easymock.EasyMockRunner;
import org.easymock.Mock;
import org.easymock.TestSubject;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;

// @RunWith attaches a runner with the test class to initialize
// the test data
@RunWith(EasyMockRunner.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
   public void testAdd(){
      //add the behavior of calc service to add two numbers
      EasyMock.expect(calcService.add(10.0,20.0)).andReturn(30.00);
      calcService.serviceUsed();
      EasyMock.expectLastCall().times(1);

      //activate the mock
      EasyMock.replay(calcService);

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

      //verify call to calcService is made or not
      EasyMock.verify(calcService);
   }
}
```

### Step 4: Execute test cases

Create a java class file named TestRunner in **C:\> EasyMock_WORKSPACE** to execute Test case(s).

### 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:\EasyMock_WORKSPACE>javac CalculatorService.java MathApplication.java
MathApplicationTester.java TestRunner.java
```

Now run the Test Runner to see the result:

```
C:\EasyMock_WORKSPACE>java TestRunner
```

Verify the output.

```
true
```

# Example with calcService.serviceUsed() Called Twice

**Step 1: Create an interface CalculatorService to provide mathematical functions.**

**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);
   public void serviceUsed();
}
```

**Step 2: Create a JAVA class to represent MathApplication.**

**MathApplication.java**

```
public class MathApplication {
   private CalculatorService calcService;

   public void setCalculatorService(CalculatorService calcService){
      this.calcService = calcService;
   }
   public double add(double input1, double input2){
      calcService.serviceUsed();
      calcService.serviceUsed();
      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 EasyMock.

### MathApplicationTester.java MathApplicationTester.java

```
import org.easymock.EasyMock;
import org.easymock.EasyMockRunner;
import org.easymock.Mock;
import org.easymock.TestSubject;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;

// @RunWith attaches a runner with the test class to initialize
// the test data
@RunWith(EasyMockRunner.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
   public void testAdd(){
      //add the behavior of calc service to add two numbers
      EasyMock.expect(calcService.add(10.0,20.0)).andReturn(30.00);
      calcService.serviceUsed();
      EasyMock.expectLastCall().times(1);

      //activate the mock
      EasyMock.replay(calcService);

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

      //verify call to calcService is made or not
      EasyMock.verify(calcService);
   }
}
```

### Step 4: Execute test cases

Create a java class file named TestRunner in **C:\> EasyMock_WORKSPACE** to execute Test case(s).

### 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:\EasyMock_WORKSPACE>javac CalculatorService.java MathApplication.java
MathApplicationTester.java TestRunner.java
```

Now run the Test Runner to see the result:

```
C:\EasyMock_WORKSPACE>java TestRunner
```

Verify the output.

```
testAdd(com.tutorialspoint.mock.MathApplicationTester):

  Unexpected method call CalculatorService.serviceUsed():

    CalculatorService.add(10.0, 20.0): expected: 1, actual: 0

    CalculatorService.serviceUsed(): expected: 1, actual: 2

false
```

# Example without Calling calcService.serviceUsed()

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

**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);
   public void serviceUsed();
}
```

**Step 2: Create a JAVA class to represent MathApplication**

**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 EasyMock.

### MathApplicationTester.java MathApplicationTester.java

```
import org.easymock.EasyMock;
import org.easymock.EasyMockRunner;
import org.easymock.Mock;
import org.easymock.TestSubject;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;

// @RunWith attaches a runner with the test class to initialize
// the test data
@RunWith(EasyMockRunner.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
   public void testAdd(){
      //add the behavior of calc service to add two numbers
      EasyMock.expect(calcService.add(10.0,20.0)).andReturn(30.00);
      calcService.serviceUsed();
      EasyMock.expectLastCall().times(1);

      //activate the mock
      EasyMock.replay(calcService);

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

      //verify call to calcService is made or not
      EasyMock.verify(calcService);
   }
}
```

### Step 4: Execute test cases

Create a java class file named TestRunner in **C:\> EasyMock_WORKSPACE** to execute Test case(s).

### 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:\EasyMock_WORKSPACE>javac CalculatorService.java MathApplication.java
MathApplicationTester.java TestRunner.java
```

Now run the Test Runner to see the result:

```
C:\EasyMock_WORKSPACE>java TestRunner
```

Verify the output.

```
testAdd(com.tutorialspoint.mock.MathApplicationTester):

  Expectation failure on verify:

    CalculatorService.serviceUsed(): expected: 1, actual: 0

false
```

# 8 VARYING CALLS

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

- **times (int min, int max)** – expects between min and max calls.
- **atLeastOnce ()** – expects at least one call.
- **anyTimes ()** – expects an unrestricted number of calls.

## Example with times (min,max)

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

**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);

   public void serviceUsed();

}
```

**Step 2: Create a JAVA class to represent MathApplication**

**MathApplication.java**

```
public class MathApplication {

   private CalculatorService calcService;


   public void setCalculatorService(CalculatorService calcService){

      this.calcService = calcService;
```

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

### MathApplicationTester.java MathApplicationTester.java

```
import org.easymock.EasyMock;
import org.easymock.EasyMockRunner;
import org.easymock.Mock;
import org.easymock.TestSubject;
import org.junit.Assert;
import org.junit.Before;
```



```
import org.junit.Test;
import org.junit.runner.RunWith;

// @RunWith attaches a runner with the test class to initialize
// the test data
@RunWith(EasyMockRunner.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
   public void testAdd(){
      //add the behavior of calc service to add two numbers
      EasyMock.expect(calcService.add(10.0,20.0)).andReturn(30.00);
      calcService.serviceUsed();
      EasyMock.expectLastCall().times(1,3);

      //activate the mock
      EasyMock.replay(calcService);
```

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

      //verify call to calcService is made or not
      EasyMock.verify(calcService);
   }
}
```

**Step 4: Execute test cases**

Create a java class file named TestRunner in **C:\> EasyMock_WORKSPACE** to execute Test case(s)

**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:\EasyMock_WORKSPACE>javac CalculatorService.java MathApplication.java
MathApplicationTester.java TestRunner.java
```

Now run the Test Runner to see the result:

```
C:\EasyMock_WORKSPACE>java TestRunner
```

Verify the output.

```
true
```

# Example with atLeastOnce

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

**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);

   public void serviceUsed();

}
```

**Step 2: Create a JAVA class to represent MathApplication**

**MathApplication.java**

```
public class MathApplication {

   private CalculatorService calcService;


   public void setCalculatorService(CalculatorService calcService){
```

```
      this.calcService = calcService;
   }
   public double add(double input1, double input2){
      calcService.serviceUsed();
      calcService.serviceUsed();
      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 EasyMock.

### MathApplicationTester.java MathApplicationTester.java

```
import org.easymock.EasyMock;
import org.easymock.EasyMockRunner;
import org.easymock.Mock;
import org.easymock.TestSubject;
import org.junit.Assert;
import org.junit.Before;
```

```
import org.junit.Test;
import org.junit.runner.RunWith;

// @RunWith attaches a runner with the test class to initialize
// the test data
@RunWith(EasyMockRunner.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
   public void testAdd(){
      //add the behavior of calc service to add two numbers
      EasyMock.expect(calcService.add(10.0,20.0)).andReturn(30.00);
      calcService.serviceUsed();
      EasyMock.expectLastCall().atLeastOnce();

      //activate the mock
      EasyMock.replay(calcService);
```

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

      //verify call to calcService is made or not
      EasyMock.verify(calcService);
   }
}
```

### Step 4: Execute test cases

Create a java class file named TestRunner in **C:\> EasyMock_WORKSPACE** to execute Test case(s).

### 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:\EasyMock_WORKSPACE>javac CalculatorService.java MathApplication.java
MathApplicationTester.java TestRunner.java
```

Now run the Test Runner to see the result:

```
C:\EasyMock_WORKSPACE>java TestRunner
```

Verify the output.

```
true
```

# Example with anyTimes

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

**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);

   public void serviceUsed();

}
```

**Step 2: Create a JAVA class to represent MathApplication**

**MathApplication.java**

```
public class MathApplication {

   private CalculatorService calcService;


   public void setCalculatorService(CalculatorService calcService){

      this.calcService = calcService;
```

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

### MathApplicationTester.java

```
import org.easymock.EasyMock;
import org.easymock.EasyMockRunner;
import org.easymock.Mock;
import org.easymock.TestSubject;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;
```

```
import org.junit.runner.RunWith;

// @RunWith attaches a runner with the test class to initialize
// the test data
@RunWith(EasyMockRunner.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
   public void testAdd(){
      //add the behavior of calc service to add two numbers
      EasyMock.expect(calcService.add(10.0,20.0)).andReturn(30.00);
      calcService.serviceUsed();
      EasyMock.expectLastCall().anyTimes();

      //activate the mock
      EasyMock.replay(calcService);

      //test the add functionality
```

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


      //verify call to calcService is made or not
      EasyMock.verify(calcService);
   }
}
```

### Step 4: Execute test cases

Create a java class file named TestRunner in **C:\> EasyMock_WORKSPACE** to execute Test case(s).

### 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:\EasyMock_WORKSPACE>javac CalculatorService.java MathApplication.java
MathApplicationTester.java TestRunner.java
```

Now run the Test Runner to see the result:

```
C:\EasyMock_WORKSPACE>java TestRunner
```

Verify the output.

```
true
```

# 9 EXCEPTION HANDLING

EasyMock 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

EasyMock.expect(calcService.add(10.0,20.0)).andThrow(new
RuntimeException("Add operation not implemented"));
```

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**

**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**

**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 EasyMock.

### MathApplicationTester.java

```
import org.easymock.EasyMock;
import org.easymock.EasyMockRunner;
import org.easymock.Mock;
import org.easymock.TestSubject;
import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;
```

```
// @RunWith attaches a runner with the test class to initialize
// the test data
@RunWith(EasyMockRunner.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
      EasyMock.expect(calcService.add(10.0,20.0)).andThrow(new
      RuntimeException("Add operation not implemented"));

      //activate the mock
      EasyMock.replay(calcService);

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

```
      //verify call to calcService is made or not

      EasyMock.verify(calcService);

   }

}
```

### Step 4: Execute test cases

Create a java class file named TestRunner in **C:\> EasyMock_WORKSPACE** to execute Test case(s).

### 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:\EasyMock_WORKSPACE>javac MathApplicationTester.java
```

Now run the Test Runner to see the result:

```
C:\EasyMock_WORKSPACE>java TestRunner
```

Verify the output.

```
true
```

# ⑩ CREATEMOCK

So far, we've used annotations to create mocks. EasyMock provides various methods to create mock objects. EasyMock.createMock() 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 = EasyMock.createMock(CalculatorService.class);
```

## Example

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

**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**

**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 EasyMock.

Here we've added two mock method calls, add() and subtract(), to the mock object via expect(). However during testing, we've called subtract() before calling add(). When we create a mock object using EasyMock.createMock(), the order of execution of the method does not matter.

### MathApplicationTester.java

```
import org.easymock.EasyMock;

import org.easymock.EasyMockRunner;

import org.junit.Assert;

import org.junit.Before;

import org.junit.Test;

import org.junit.runner.RunWith;
```

```
@RunWith(EasyMockRunner.class)
public class MathApplicationTester {

   private MathApplication mathApplication;
   private CalculatorService calcService;

   @Before
   public void setUp(){
      mathApplication = new MathApplication();
      calcService = EasyMock.createMock(CalculatorService.class);
      mathApplication.setCalculatorService(calcService);
   }

   @Test
   public void testAddAndSubtract(){

      //add the behavior to add numbers
      EasyMock.expect(calcService.add(20.0,10.0)).andReturn(30.0);

      //subtract the behavior to subtract numbers
      EasyMock.expect(calcService.subtract(20.0,10.0)).andReturn(10.0);

      //activate the mock
      EasyMock.replay(calcService);

      //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
      EasyMock.verify(calcService);
   }
}
```

### Step 4: Execute test cases

Create a java class file named TestRunner in **C:\> EasyMock_WORKSPACE** to execute Test case(s).

### 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:\EasyMock_WORKSPACE>javac MathApplicationTester.java
```

Now run the Test Runner to see the result:

```
C:\EasyMock_WORKSPACE>java TestRunner
```

Verify the output.

```
true
```

# 11 CREATESTRICTMOCK

EasyMock.createStrictMock() creates a mock and also takes care of the order of method calls that the mock is going to make in due course of its action.

**Syntax**

```
calcService = EasyMock.createStrictMock(CalculatorService.class);
```

## Example

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

**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**

**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 EasyMock.

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

### MathApplicationTester.java

```
import org.easymock.EasyMock;
import org.easymock.EasyMockRunner;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
```

```
@RunWith(EasyMockRunner.class)
public class MathApplicationTester {

   private MathApplication mathApplication;
   private CalculatorService calcService;

   @Before
   public void setUp(){
      mathApplication = new MathApplication();
      calcService = EasyMock.createStrictMock(CalculatorService.class);
      mathApplication.setCalculatorService(calcService);
   }

   @Test
   public void testAddAndSubtract(){

      //add the behavior to add numbers
      EasyMock.expect(calcService.add(20.0,10.0)).andReturn(30.0);

      //subtract the behavior to subtract numbers
      EasyMock.expect(calcService.subtract(20.0,10.0)).andReturn(10.0);

      //activate the mock
      EasyMock.replay(calcService);

      //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
      EasyMock.verify(calcService);
   }
}
```

### Step 4: Execute test cases

Create a java class file named TestRunner in **C:\> EasyMock_WORKSPACE** to execute Test case(s).

### 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:\EasyMock_WORKSPACE>javac MathApplicationTester.java
```

Now run the Test Runner to see the result:

```
C:\EasyMock_WORKSPACE>java TestRunner
```

Verify the output.

```
testAddAndSubtract(com.tutorialspoint.mock.MathApplicationTester):
  Unexpected method call CalculatorService.subtract(20.0, 10.0):
    CalculatorService.add(20.0, 10.0): expected: 1, actual: 0
false
```

# ⑫ CREATENICEMOCK

EasyMock.createNiceMock() creates a mock and sets the default implementation of each method of the mock. If EasyMock.createMock() is used, then invoking the mock method throws assertion error.

**Syntax**

```
calcService = EasyMock.createNiceMock(CalculatorService.class);
```

## Example

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

**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**

**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 EasyMock.

Here we've added one mock method call, add(), via expect(). However during testing, we've called subtract() and other methods as well. When we create a mock object using EasyMock.createNiceMock(), the default implementation with default values are available.

### MathApplicationTester.java

```
import org.easymock.EasyMock;
import org.easymock.EasyMockRunner;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
```

```
@RunWith(EasyMockRunner.class)
public class MathApplicationTester {

   private MathApplication mathApplication;

   private CalculatorService calcService;

   @Before
   public void setUp(){
      mathApplication = new MathApplication();
      calcService = EasyMock.createNiceMock(CalculatorService.class);
      mathApplication.setCalculatorService(calcService);
   }

   @Test
      public void testCalcService(){

      //add the behavior to add numbers
      EasyMock.expect(calcService.add(20.0,10.0)).andReturn(30.0);

      //activate the mock
      EasyMock.replay(calcService);

      //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),0.0,0);

      //test the multiply functionality
      Assert.assertEquals(mathApplication.divide(20.0, 10.0),0.0,0);

      //test the divide functionality
      Assert.assertEquals(mathApplication.multiply(20.0, 10.0),0.0,0);

      //verify call to calcService is made or not
      EasyMock.verify(calcService);
   }
}
```

### Step 4: Execute test cases

Create a java class file named TestRunner in **C:\> EasyMock_WORKSPACE** to execute Test case(s).

### 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:\EasyMock_WORKSPACE>javac MathApplicationTester.java
```

Now run the Test Runner to see the result:

```
C:\EasyMock_WORKSPACE>java TestRunner
```

Verify the output.

```
true
```

# 13 EASYMOCKSUPPORT

EasyMockSupport is a utility or helper class for test classes. It provides the following functionalities:

- **replayAll()** – Registers all the created mocks in one batch.
- **verifyAll()** – Verifies all the mock operations in one batch.
- **resetAll()** – Resets all the mock operations in one batch.

## Example

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

**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**

**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 EasyMock.

### MathApplicationTester.java

```
import org.easymock.EasyMockRunner;
import org.easymock.EasyMockSupport;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;

@RunWith(EasyMockRunner.class)
public class MathApplicationTester extends EasyMockSupport {
```

```
private MathApplication mathApplication1;
private MathApplication mathApplication2;

private CalculatorService calcService1;
private CalculatorService calcService2;

@Before
public void setUp(){
   mathApplication1 = new MathApplication();
   mathApplication2 = new MathApplication();
   calcService1 = createNiceMock(CalculatorService.class);
   calcService2 = createNiceMock(CalculatorService.class);
   mathApplication1.setCalculatorService(calcService1);
   mathApplication2.setCalculatorService(calcService2);
}

@Test
public void testCalcService(){

   //activate all mocks
   replayAll();

   //test the add functionality
   Assert.assertEquals(mathApplication1.add(20.0, 10.0),0.0,0);

   //test the subtract functionality
   Assert.assertEquals(mathApplication1.subtract(20.0, 10.0),0.0,0);
```

```
      //test the multiply functionality
      Assert.assertEquals(mathApplication1.divide(20.0, 10.0),0.0,0);

      //test the divide functionality
      Assert.assertEquals(mathApplication1.multiply(20.0, 10.0),0.0,0);

      //test the add functionality
      Assert.assertEquals(mathApplication2.add(20.0, 10.0),0.0,0);

      //test the subtract functionality
      Assert.assertEquals(mathApplication2.subtract(20.0, 10.0),0.0,0);

      //test the multiply functionality
      Assert.assertEquals(mathApplication2.divide(20.0, 10.0),0.0,0);

      //test the divide functionality
      Assert.assertEquals(mathApplication2.multiply(20.0, 10.0),0.0,0);

      //verify all the mocks
      verifyAll();
}
}
```

### Step 4: Execute test cases

Create a java class file named TestRunner in **C:\> EasyMock_WORKSPACE** to execute Test case(s).

### 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:\EasyMock_WORKSPACE>javac MathApplicationTester.java
```

Now run the Test Runner to see the result:

```
C:\EasyMock_WORKSPACE>java TestRunner
```

Verify the output.

```
true
```