Testing Synopsis

Unit Tests in Java with Framework:

JUnit

&

mockito

By Miguel Ángel Herranz Marcos 2018 EASV

INDEX

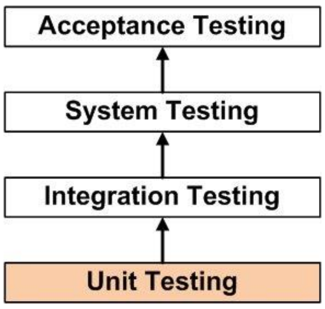
WHAT IS UNIT TESTING?

❖ INTRODUCTION

The work to be done consists of creating a series of unit tests in the java language, for this I have done the work by supporting myself in the JUnit and Mockito frameworks.

From my point of view, the first thing is to make clear that they are the unit tests and why they are carried out.

❖ UNIT TESTING



It's a level of software testing where individual units/ components of a software are tested. The purpose is to validate that each unit of the software performs as designed. A unit is the smallest testable part of any software. It usually has one or a few inputs and usually a single output. In procedural programming, a unit may be an individual program, function, procedure, etc. In object-oriented programming, the smallest unit is a method, which may belong to a base/ super class, abstract class or derived/ child class. (Some treat a module of an application as a unit. This is to be discouraged as there will probably be many individual units within that module.) Unit testing frameworks, drivers, stubs, and mock/ fake objects are used to assist in unit testing. The Unit Testing Method is performed by using the White Box Testing method.

❖ WHITE BOX TESTING

It's a software testing method in which the internal structure/design/implementation of the item being tested is known to the tester. Programming know-how and the implementation knowledge is essential.

❖ BENEFITS

Unit testing increases confidence in changing/ maintaining code. If good unit tests are written and if they are run every time any code is changed, we will be able to promptly catch any defects introduced due to the change. Codes are more reusable. In order to make unit testing possible, codes need to be modular. This means that codes are easier to reuse, so the development is faster. Writing tests takes time but the time is compensated by the less amount of time it takes to run the tests.

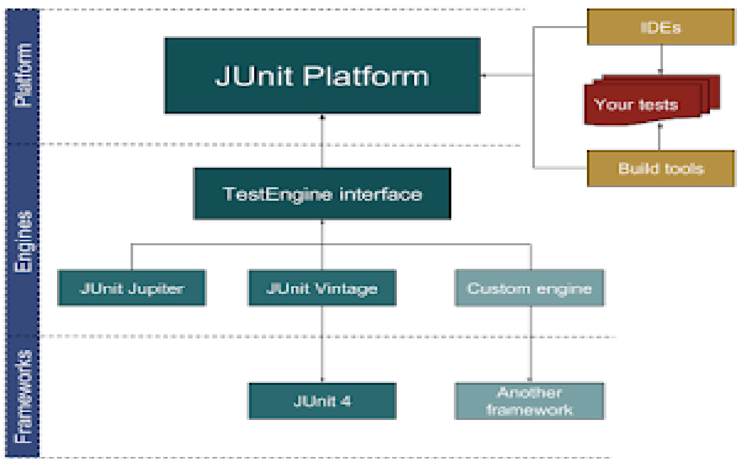
❖ FRAMEWORKS

Testing is a very important part, and for that reason, for its application to be as comfortable as possible, several Frameworks and Tools for developers have been developed.

These are a list of the most common Frameworks:

REST-Assured, Selenium, TestNG, Spock, Cucumber, Spring.

❖ WHAT FRAMEWORK HAVE I USED?



JUnit

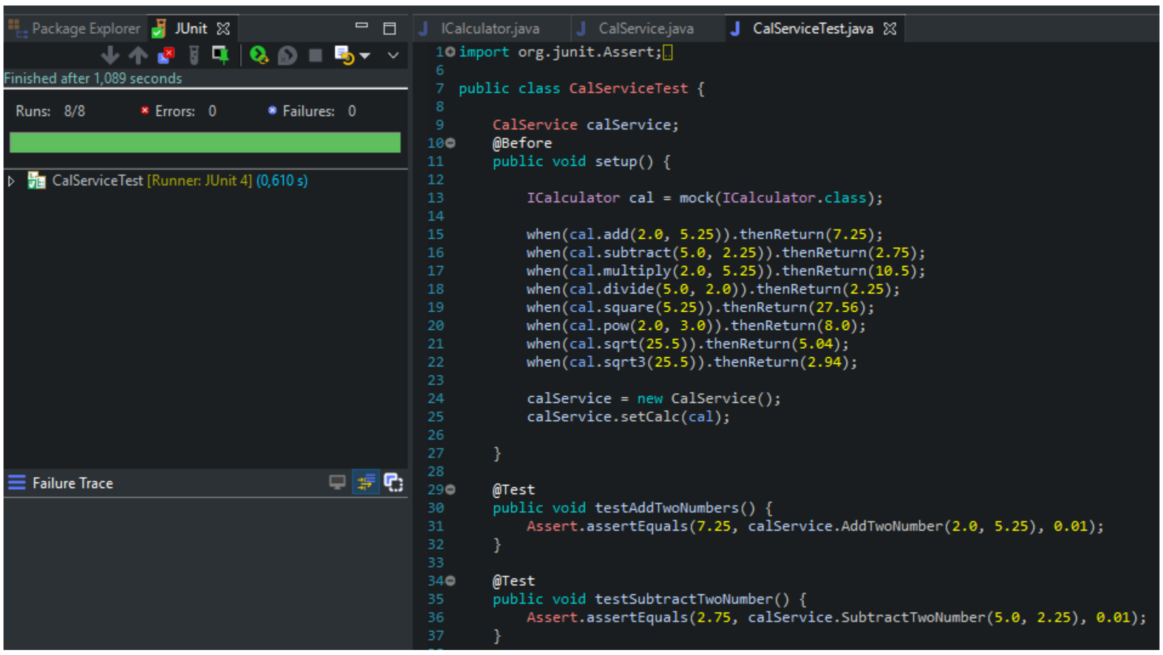
JUnit is an open source framework designed for the purpose of writing and running tests in the Java programming language. JUnit, originally written by Erich Gamma and Kent Beck, has been important in the evolution of test-driven development <method of implementing software programming that interlaces unit testing, programming and refactoring on source code>, which is part of a larger software design paradigm known as Extreme Programming (XP).

JUnit has a graphical user interface (GUI), making it possible to write and test source code quickly and easily. JUnit allows the developer to incrementally build test suites to measure progress and detect unintended side effects. Tests can be run continuously. Results are provided immediately. JUnit shows test progress in a bar that is normally green but turns red when a test fails. Multiple tests can be run concurrently.

Although JUnit was originally written for Java, spinoffs have developed for several other programming languages. The entire family of related testing frameworks is called xUnit.

mockito

Mockito is a mocking framework, JAVA-based library that is used for effective unit testing of JAVA applications. Mockito is used to mock interfaces so that a dummy functionality can be added to a mock interface that can be used in unit testing. 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.



```
Finished after 1,089 seconds
Runs:  8/8     Errors:  0     Failures:  0
CalServiceTest [Runner: JUnit 4] (0,610 s)
```

```java
import org.junit.Assert;

public class CalServiceTest {

    CalService calService;
    @Before
    public void setup() {

        ICalculator cal = mock(ICalculator.class);

        when(cal.add(2.0, 5.25)).thenReturn(7.25);
        when(cal.subtract(5.0, 2.25)).thenReturn(2.75);
        when(cal.multiply(2.0, 5.25)).thenReturn(10.5);
        when(cal.divide(5.0, 2.0)).thenReturn(2.25);
        when(cal.square(5.25)).thenReturn(27.56);
        when(cal.pow(2.0, 3.0)).thenReturn(8.0);
        when(cal.sqrt(25.5)).thenReturn(5.04);
        when(cal.sqrt3(25.5)).thenReturn(2.94);

        calService = new CalService();
        calService.setCalc(cal);

    }

    @Test
    public void testAddTwoNumbers() {
        Assert.assertEquals(7.25, calService.AddTwoNumber(2.0, 5.25), 0.01);
    }

    @Test
    public void testSubtractTwoNumber() {
        Assert.assertEquals(2.75, calService.SubtractTwoNumber(5.0, 2.25), 0.01);
    }
```

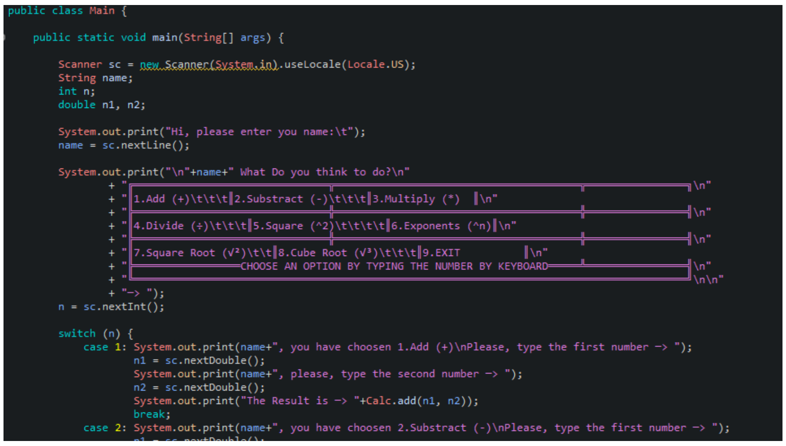
Example of JUnit and Mockito GUI in a Java Project

WHAT I DID?

Well, my synopsis, consists of three parts: What are unit tests, and why are they used? How would JUnit be implemented in a Java Project? And how would does It with Mockito?

The first part has just been explained, so I will proceed to explain the second.

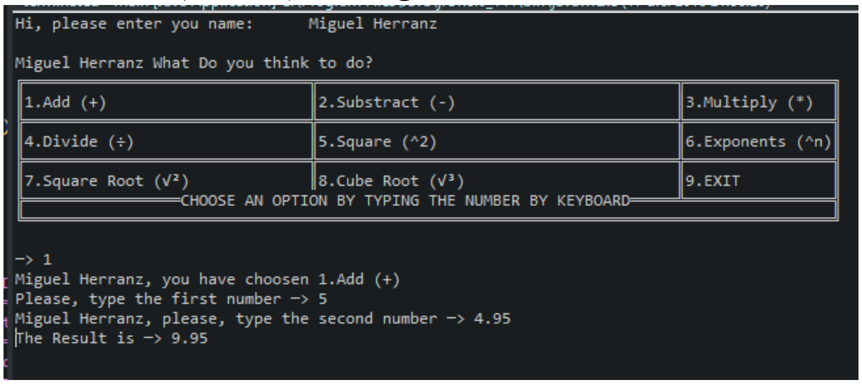
❖ JAVA PROJECT WITH JUNIT



```java
public class Main {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in).useLocale(Locale.US);
        String name;
        int n;
        double n1, n2;

        System.out.print("Hi, please enter you name:\t");
        name = sc.nextLine();

        System.out.print("\n"+name+" What Do you think to do?\n"
                + "╔════════════════╦════════════════════╦════════╗\n"
                + "║1.Add (+)\t\t\t║2.Substract (-)\t\t\t║3.Multiply (*)  ║\n"
                + "╠════════════════╬════════════════════╬════════╣\n"
                + "║4.Divide (÷)\t\t\t║5.Square (^2)\t\t\t\t║6.Exponents (^n)║\n"
                + "╠════════════════╬════════════════════╬════════╣\n"
                + "║7.Square Root (√²)\t\t║8.Cube Root (√³)\t\t\t║9.EXIT          ║\n"
                + "╠═══════CHOOSE AN OPTION BY TYPING THE NUMBER BY KEYBOARD═══╩════════╣\n"
                + "╚═════════════════════════════════════════════════╝\n\n"
                + "-> ");
        n = sc.nextInt();

        switch (n) {
            case 1: System.out.print(name+", you have choosen 1.Add (+)\nPlease, type the first number -> ");
                    n1 = sc.nextDouble();
                    System.out.print(name+", please, type the second number -> ");
                    n2 = sc.nextDouble();
                    System.out.print("The Result is -> "+Calc.add(n1, n2));
                    break;
            case 2: System.out.print(name+", you have choosen 2.Substract (-)\nPlease, type the first number -> ");
```

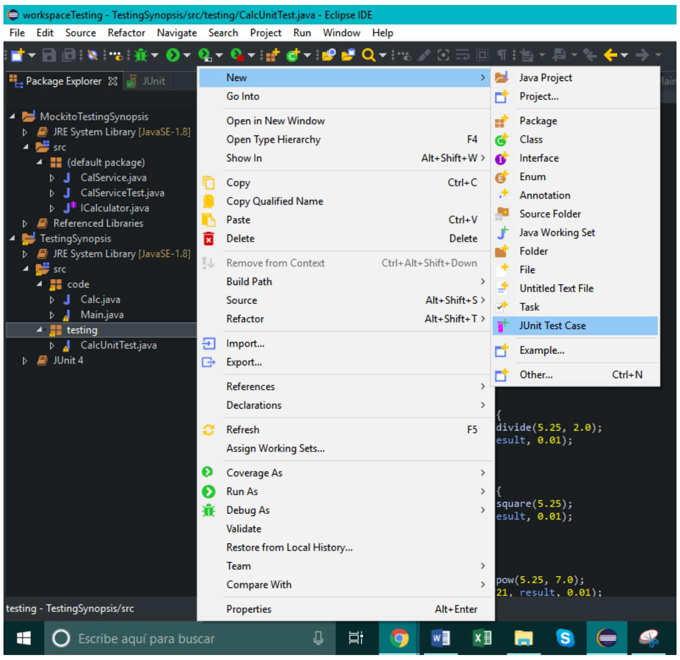
You start by creating a normal Java project, in this case my project is a calculator. My calculator is a console application, so it has no window, everything is on the console screen.

The design architecture is simple, I created a static class (so as not to have to create an object) where all the calculator's methods were found (add, subtract, multiply, divide, raise to a power, to the square, square root and root) cubic), then generate a Main class that was responsible for both the graphic design of this and the logic when choosing the functions.



```
Hi, please enter you name:      Miguel Herranz

Miguel Herranz What Do you think to do?
╔══════════════════════╦═════════════════════════╦═════════════════╗
║1.Add (+)             ║2.Substract (-)          ║3.Multiply (*)   ║
╠══════════════════════╬═════════════════════════╬═════════════════╣
║4.Divide (÷)          ║5.Square (^2)            ║6.Exponents (^n) ║
╠══════════════════════╬═════════════════════════╬═════════════════╣
║7.Square Root (√²)    ║8.Cube Root (√³)         ║9.EXIT           ║
╠════════════CHOOSE AN OPTION BY TYPING THE NUMBER BY KEYBOARD═════╣
╚══════════════════════════════════════════════════════════════════╝

-> 1
Miguel Herranz, you have choosen 1.Add (+)
Please, type the first number -> 5
Miguel Herranz, please, type the second number -> 4.95
The Result is -> 9.95
```

Then, I decided to make a new package which had the JUnit tests, to make the tests I used my usual IDE that I use with Java (Eclipse), JUnit is in the vast majority of IDEs.

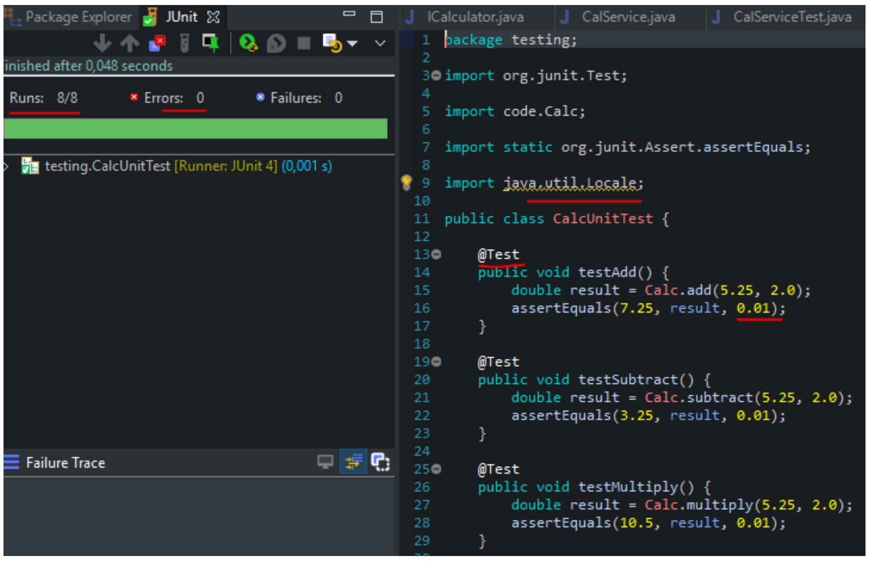


Generating the tests in this way is really simple, when you create the test it asks you what name you want to give it and with what class it will be related. When you create it, it asks you if you want to implement the corresponding JUnit library, in this case JUnit 4, automatically.

Then you declare the Test that you want with @Test on top of each

one so that the library detects them, a variable is generated that is the method to be checked and then an Assert is used, depending on the moment one or the other will be used, and depending on each what is introduced within the parentheses varies; however, it is usual to follow the order of (EXPECTED VALUE, VARIABLE CREATED FROM THE FUNCTION)

We can see how they have been executed, 8 tests of which 0 have been erroneous. Also how the @Test makes that recognize that it is a test. Later I entered a "0.01" in all the asserts because when working with double instead of int the test I did wrong, showing 8 errors. Also in the whole project implement a Java.util.Locale, so that when entering the decimals by console in the main, recognize the points as commas and there were no misunderstandings. It is logical that if the console shows you the decimals with points, you enter them in the same way.



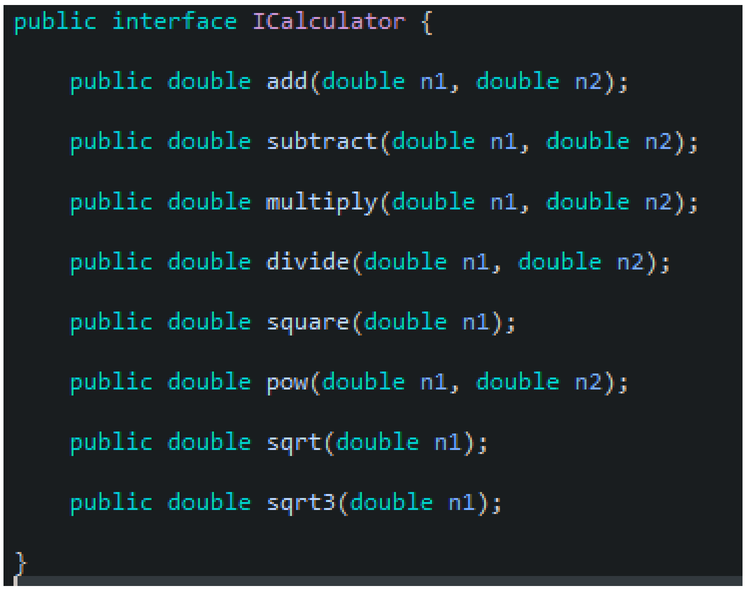
```
inished after 0,048 seconds
Runs: 8/8     Errors: 0     Failures: 0
testing.CalcUnitTest [Runner: JUnit 4] (0,001 s)
```

```java
package testing;

import org.junit.Test;

import code.Calc;

import static org.junit.Assert.assertEquals;

import java.util.Locale;

public class CalcUnitTest {

    @Test
    public void testAdd() {
        double result = Calc.add(5.25, 2.0);
        assertEquals(7.25, result, 0.01);
    }

    @Test
    public void testSubtract() {
        double result = Calc.subtract(5.25, 2.0);
        assertEquals(3.25, result, 0.01);
    }

    @Test
    public void testMultiply() {
        double result = Calc.multiply(5.25, 2.0);
        assertEquals(10.5, result, 0.01);
    }
```

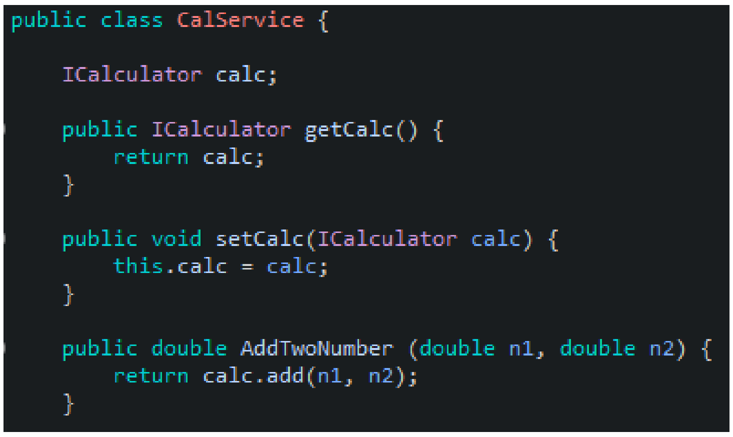
❖ JAVA PROJECT WITH MOCKITO

Mocking is a way of producing dummy objects, operations, and results as if they were real scenarios. This means that it deals with no real database connections and no real server up and running. However, it mimics them so that the lines are also covered and expect the actual result. Thus, it can be compared with the expected result and asserted.

We will use JUnit with the Mockito framework for our unit tests.



```java
public interface ICalculator {

    public double add(double n1, double n2);

    public double subtract(double n1, double n2);

    public double multiply(double n1, double n2);

    public double divide(double n1, double n2);

    public double square(double n1);

    public double pow(double n1, double n2);

    public double sqrt(double n1);

    public double sqrt3(double n1);

}
```



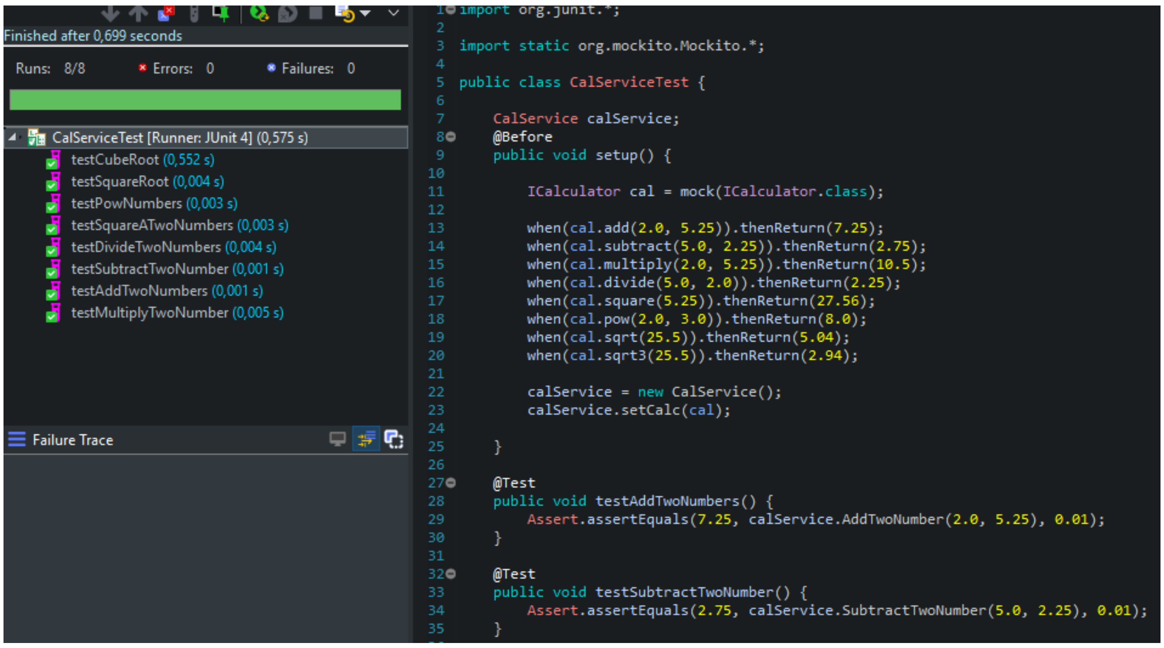
```java
public class CalService {

    ICalculator calc;

    public ICalculator getCalc() {
        return calc;
    }

    public void setCalc(ICalculator calc) {
        this.calc = calc;
    }

    public double AddTwoNumber (double n1, double n2) {
        return calc.add(n1, n2);
    }
```

Well, as in the previous example I had already created a series of unit tests, directly oriented to JUnit, the composition of the project varies as Mockito is now implemented. The first step was to create an interface "ICalculator" which had a series of defined functions.

The second step was to create a class (CalcService) which had as an attribute the "ICalculator" interface, and declare in it the methods which called the functions of the interface. Within these methods was the logical part of each of them.

The last step was to create (in the same way as before) a Test using JUnit and import the "mockito-all-1.9.5.jar" library. Then an attribute of the Test class was declared (CalServiceTest) and adding an @Before created an instance in the setUp function. In which instances in the Interface referring to the "mock" of the class. And with a series of sentences with the keywords "when" and ".thenReturn" and the expected values by entering the function and the result.

Then the calService object is assigned the cal interface (ICalculator) already filled with the mock data, and the tests are done in a normal way. With the only difference that in this case instead of referring to a Class with the functions and already, you refer to a class that previously has already been assigned to an interface filled with mock data. And that's why reference is made to "calService".



```
Finished after 0,699 seconds

Runs:  8/8      Errors:  0      Failures:  0

CalServiceTest [Runner: JUnit 4] (0,575 s)
    testCubeRoot (0,552 s)
    testSquareRoot (0,004 s)
    testPowNumbers (0,003 s)
    testSquareATwoNumbers (0,003 s)
    testDivideTwoNumbers (0,004 s)
    testSubtractTwoNumber (0,001 s)
    testAddTwoNumbers (0,001 s)
    testMultiplyTwoNumber (0,005 s)
```

```java
import org.junit.*;

import static org.mockito.Mockito.*;

public class CalServiceTest {

    CalService calService;
    @Before
    public void setup() {

        ICalculator cal = mock(ICalculator.class);

        when(cal.add(2.0, 5.25)).thenReturn(7.25);
        when(cal.subtract(5.0, 2.25)).thenReturn(2.75);
        when(cal.multiply(2.0, 5.25)).thenReturn(10.5);
        when(cal.divide(5.0, 2.0)).thenReturn(2.25);
        when(cal.square(5.25)).thenReturn(27.56);
        when(cal.pow(2.0, 3.0)).thenReturn(8.0);
        when(cal.sqrt(25.5)).thenReturn(5.04);
        when(cal.sqrt3(25.5)).thenReturn(2.94);

        calService = new CalService();
        calService.setCalc(cal);

    }

    @Test
    public void testAddTwoNumbers() {
        Assert.assertEquals(7.25, calService.AddTwoNumber(2.0, 5.25), 0.01);
    }

    @Test
    public void testSubtractTwoNumber() {
        Assert.assertEquals(2.75, calService.SubtractTwoNumber(5.0, 2.25), 0.01);
    }
```