MOCKITO

MIGUEL ANGEL RAMIREZ JUAREZ

JAVA ACADEMY

SEP/ 06/ 2024

What is Mockito in Java?

Mockito is a Java library that simplifies the creation of mock objects for unit testing. Its primary purpose is to allow developers to isolate and test a specific unit of code without relying on its external dependencies. By using Mockito, you can mock and verify interactions with dependencies, ensuring that the code behaves correctly under various conditions.

Key Features:

Mock Objects: Allows the creation of mock objects to test interactions with other components of the system.

Interaction Verification: Enables verification of method calls, including the correctness of parameters and the expected number of invocations.

Behavior Configuration: Facilitates defining predefined responses for method calls on mock objects.

JUnit Integration: Easily integrates with testing frameworks like JUnit, simplifying the creation and execution of unit tests.

Mockito is widely used to enhance software quality by enabling more accurate tests focused on business logic, without depending on real implementations of dependencies.

Project Overview

In this project, I utilized Mockito to create a simple example demonstrating its application in unit testing. The project involves a basic Java setup with a service that relies on a calculator class for its operations.

Project Structure:

Calculator Class: Represents a basic model with an add method for summing two numbers.

CalculatorService Class: Uses the Calculator class to perform operations and provides a method performAddition to execute addition.

Unit Test: Uses Mockito to create a mock of the Calculator class, allowing for the simulation of its behavior and verification of interactions with the CalculatorService.

Objectives:

Demonstrate Mockito Usage: Show how Mockito can be used to create mock objects and configure their behavior for testing purposes.

Verify Interactions: Ensure that the CalculatorService correctly interacts with the Calculator mock, and verify that the addition operation behaves as expected.

Outcome:

The project successfully demonstrates the use of Mockito by providing a clear example of mocking and verifying interactions. The unit tests confirm that the CalculatorService correctly calls the add method on the Calculator mock and handles the addition operation as intended.

By isolating the Calculator dependency, Mockito allows for a focused test of the CalculatorService logic, ensuring reliable and maintainable code.

code explanation:

1. **Calculator.java**

```java
package com.example.mockitoexample.model;

public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
}
```

- **Purpose**: This is a simple model class with a single method `add` that takes two integers and returns their sum.
- **Usage**: The `Calculator` class represents a basic component whose behavior we want to simulate and verify in our tests.

2- **CalculatorService.java**

```java
import com.example.mockitoexample.model.Calculator;
```

```
public class CalculatorService {
    private Calculator calculator;

    public CalculatorService(Calculator calculator) {
        this.calculator = calculator;
    }

    public int performAddition(int a, int b) {
        return calculator.add(a, b);
    }
}
```

- **Purpose**: This class uses an instance of `Calculator` to perform operations. It has a method `performAddition` that calls the `add` method of `Calculator`.
- **Usage**: The `CalculatorService` class is the component under test. It depends on `Calculator`, which we will mock in our tests.

3. **CalculatorServiceTest.java**

```
import com.example.mockitoexample.model.Calculator;

public class CalculatorService {
    private Calculator calculator;

    public CalculatorService(Calculator calculator) {
        this.calculator = calculator;
    }

    public int performAddition(int a, int b) {
        return calculator.add(a, b);
    }
}
```

- **Setup Method**: The `@BeforeEach` method initializes the mock `Calculator` and injects it into the `CalculatorService` before each test.
- **Test Method**:

- **Mock Configuration**: `when(calculator.add(5, 5)).thenReturn(10);` sets up the mock to return `10` when the `add` method is called with arguments `5` and `5`.
- **Method Under Test**: `calculatorService.performAddition(5, 5);` calls the method that uses the mocked `Calculator`.
- **Assertions**: `assertEquals(10, result);` checks that the result is as expected.
- **Verification**: `verify(calculator).add(5, 5);` ensures that the `add` method was called with the correct parameters.

TEST WITH MOCKITO IN VISUAL STUDIO CODE:

```
import org.junit.jupiter.api.Test;
// import org.springframework.boot.test.context.SpringBootTest;


import com.example.mockitoexample.model.Calculator;
import com.example.mockitoexample.service.CalculatorService;

import org.junit.jupiter.api.BeforeEach;
import org.mockito.Mockito;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.*;

public class CalculatorServiceTest {

    private Calculator calculator;
    private CalculatorService calculatorService;

    @BeforeEach
    public void setup() {
        calculator = Mockito.mock(Calculator.class);
        calculatorService = new CalculatorService(calculator);
    }

    @Test
    public void testPerformAddition() {
        when(calculator.add(5, 5)).thenReturn(10);

        int result = calculatorService.performAddition(5, 5);

        assertEquals(10, result);
```

```
        verify(calculator).add(5, 5);
    }
}
```

COVERAGE from jacoco:

com.example.mockitoexample.service

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
CalculatorService		100 %		n/a	0	2	0	4	0	2	0	1
Total	0 of 12	100 %	0 of 0	n/a	0	2	0	4	0	2	0	1

Summary of Results

In this Mockito project, the primary objective was to demonstrate the application of Mockito for unit testing by creating mock objects and verifying their interactions. The following results were achieved:

Successful Mocking:

The Calculator class was successfully mocked using Mockito, allowing the CalculatorService to interact with a simulated Calculator object instead of a real one.

Verified Method Calls:

The unit tests confirmed that the CalculatorService correctly invokes the add method on the mocked Calculator with the expected parameters. This was verified through Mockito's verify method.

Accurate Behavior Simulation:

Mockito's when method was used to configure the mock to return a predefined value (10) when specific inputs (5, 5) were provided. The CalculatorService correctly handled this mocked behavior, as evidenced by the test results.

Assertions and Validation:

The test assertions confirmed that the performAddition method in CalculatorService returned the correct result (10), validating that the service operates correctly with the mocked Calculator.

Overall, the project effectively demonstrated how Mockito can be used to create mock objects, configure their behavior, and verify interactions in unit tests. This ensures that the code under test behaves as expected and interacts correctly with its dependencies.