Academia Xideral Java Monterrey

Daniel Ivan Anaya Alvarez

Mockito Activity

Introduction

In this activity we are going to use Mockito, Mockito is used to create mock objects that simulate the behavior of real dependencies, allowing us to focus on testing the core functionality of a class without interference from external factors. This approach simplifies testing and makes it easier to identify and fix issues in the code.

In this project, we have a simple Calculator class that relies on an AdditionService to perform addition operations. The Calculator class itself does not contain the logic to add numbers; instead, it delegates this responsibility to the AdditionService. To test the Calculator class independently, we use Mockito to create a mock of AdditionService. Creating a Mock Object:

1.The first step is to create a mock object of the AdditionService interface using Mockito.mock(). This mock object acts as a stand-in for the real AdditionService during testing, allowing us to simulate its behavior without needing an actual implementation.

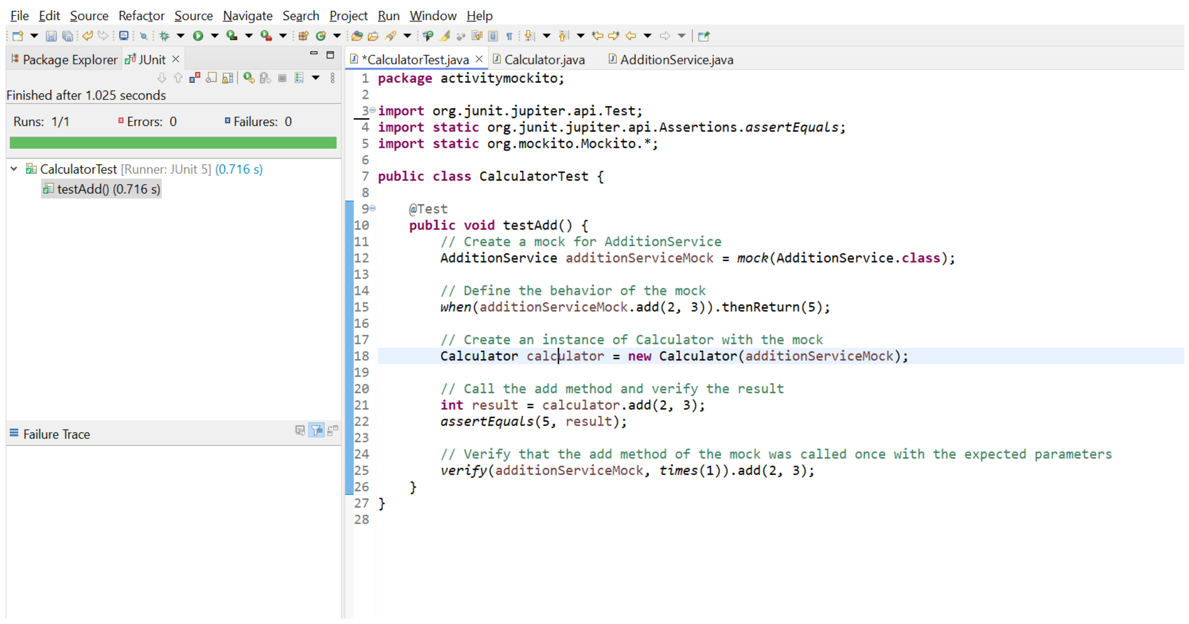
2.Defining Mock Behavior: Next, we use Mockito.when(...).thenReturn(...) to define the behavior of the mock object. In this example, we specify that when the add method is called with the parameters 2 and 3, the mock should return 5. This allows us to control the output of the mock and test the Calculator class under specific conditions.

3.Injecting the Mock: The mock object is then injected into the Calculator class by passing it to the constructor. This step replaces the real AdditionService dependency with the mock, ensuring that all calls to the AdditionService during the test are handled by the mock object.

4.Executing the Test: With the mock in place, we call the add method on the Calculator instance. The test verifies that the method returns the expected result (5), which confirms that the Calculator is correctly interacting with its dependency.

5.Verifying Interactions: Finally, we use Mockito.verify(...) to check that the mock's add method was called exactly once with the correct parameters (2 and 3). This verification step ensures that the Calculator class behaves as expected and interacts properly with its dependencies.

Print



Finished after 1.025 seconds

Runs: 1/1 Errors: 0 Failures: 0

CalculatorTest [Runner: JUnit 5] (0.716 s)
testAdd() (0.716 s)

```java
package activitymockito;

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

public class CalculatorTest {

    @Test
    public void testAdd() {
        // Create a mock for AdditionService
        AdditionService additionServiceMock = mock(AdditionService.class);

        // Define the behavior of the mock
        when(additionServiceMock.add(2, 3)).thenReturn(5);

        // Create an instance of Calculator with the mock
        Calculator calculator = new Calculator(additionServiceMock);

        // Call the add method and verify the result
        int result = calculator.add(2, 3);
        assertEquals(5, result);

        // Verify that the add method of the mock was called once with the expected parameters
        verify(additionServiceMock, times(1)).add(2, 3);
    }
}
```

Conclusion:

Mockito is a powerful tool for unit testing in Java, enabling developers to create mock objects that simulate the behavior of real dependencies.

Appendix (Code)

Calculator.java

```java
package activitymockito;

public class Calculator {
    private AdditionService additionService;

    public Calculator(AdditionService additionService) {
        this.additionService = additionService;
    }

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

AdditionService.java

```java
package activitymockito;
public interface AdditionService {
    int add(int a, int b);
}
```

CalculatorTest.java

```java
package activitymockito;

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

public class CalculatorTest {

    @Test
    public void testAdd() {
        // Create a mock for AdditionService
        AdditionService additionServiceMock = mock(AdditionService.class);

        // Define the behavior of the mock
        when(additionServiceMock.add(2, 3)).thenReturn(5);

        // Create an instance of Calculator with the mock
        Calculator calculator = new Calculator(additionServiceMock);

        // Call the add method and verify the result
        int result = calculator.add(2, 3);
        assertEquals(5, result);

        // Verify that the add method of the mock was called once with the
expected parameters
        verify(additionServiceMock, times(1)).add(2, 3);
    }
}
```