41-Mockito

Author: Vincent Lau

Learning Objectives

- Understand the purpose of using Mockito
- Understand the syntax to include Mockito framework to JUnit5
- Understand the ways to mock a dependency

Introduction

Mockito is a popular Java framework used for creating and working with mock objects in unit testing. It helps developers isolate and test specific units of code (usually classes or methods) in isolation, even when they have dependencies on other classes or components.

Mockito allows you to **replace real dependencies with mock objects** that simulate the behavior of those dependencies, making it easier to test the target code in isolation.

Mockito Framework

Mockito

Mockito is a Java library that provides an API for creating and working with mock objects. It's used primarily for unit testing and is widely used in the Java development community.

Mock Objects

In the context of Mockito, mock objects are simulated objects that mimic the behavior of real objects or components. Mockito allows you to create mock objects for classes or interfaces, define their behavior, and verify interactions with them during testing.

Maven Dependency

```xml
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>5.5.0</version>
    <scope>test</scope>
</dependency>
```

Key Concepts & Usages

Here are some key concepts and how Mockito is typically used in unit testing.

Mock Creation

You can create mock objects for classes or interfaces using Mockito's `mock()` method. For example: `MyDependency mockedDependency = mock(MyDependency.class);`

Stubbing

You can define the behavior of mock objects using `when(...).thenReturn(...)` to specify what a method should return when called on the mock. For example:
`when(mockedDependency.someMethod()).thenReturn("Mocked Result");`

Verification

You can verify interactions with mock objects using `verify(...)` to ensure that specific methods were called with expected arguments and a specified number of times. For example:

```
verify(mockedDependency, times(1)).someMethod();
```

Mock

- **Purpose**: A mock object is primarily used to replace a real object's behavior with predefined behavior. It allows you to set up expectations and specify how methods should be invoked during a test.
- **Behavior Control**: When you create a mock, **it starts with no real behavior**. You define the behavior you want to simulate by specifying how methods should respond to specific inputs.
- **Verification**: You typically use mocks to verify interactions, such as whether certain methods were called with specific arguments or how many times they were called.
- **Examples**: Mocks are often used when you want to isolate the code under test and ensure that it interacts correctly with its dependencies without invoking their actual implementations.

```
1 // Creating a mock object
2 List<String> mockList = Mockito.mock(List.class);
3
4 // Defining behavior for the mock
5 when(mockList.size()).thenReturn(5);
```

Spy

- **Purpose**: A spy, on the other hand, is used to wrap a real object while preserving its real behavior. You can use spies to partially mock an object, allowing you to override specific methods while still using the real implementations of others.
- **Behavior Control**: **Spies retain the real behavior of the object by default**. You can choose to override specific methods with custom behavior while leaving others untouched.
- **Verification**: Spies can also be used for verification, similar to mocks, to check whether specific methods were called with specific arguments.
- **Examples**: Spies are often used when you want to test parts of a real object's behavior while keeping the rest intact. This can be useful for unit testing classes with complex dependencies.

```
1 // Creating a spy object
2 List<String> spyList = Mockito.spy(new ArrayList<>());
3
4 // Overriding behavior for specific method
5 doReturn(5).when(spyList).size();
```

Mock vs Spy

- Mocks: Mocks are primarily used to replace an object's behavior completely and define how methods should behave during a test. They are typically used when you want to isolate the code under test from its dependencies.
- Spies: Spies are used to wrap real objects while preserving their real behavior. They allow you to selectively override specific methods while using the real implementations of others. Spies are useful when you want to test parts of a real object's behavior while keeping the rest intact.

Unit Test by Mockito

Unit Test

Unit testing is a software testing practice where individual units (components or functions) of a software application are tested in isolation. The goal is to verify that each unit of code works as expected and produces the correct output for a given input.

Dependencies in Unit Test

Many units of code have dependencies on other classes, services, or components. These dependencies can make unit testing challenging because you want to test the target unit in isolation, without involving the real dependencies.

Mocking Dependencies

This is where Mockito comes into play. Mockito allows you to replace real dependencies with mock objects. These mock objects can be configured to simulate the behavior of real dependencies without executing their actual code. This way, you can isolate the unit under test and focus on its behavior without worrying about the behavior of its dependencies.

@Mock Example

main code: MyDependency

```
1 // Example class MyDependency
2 class MyDependency {
3     String someMethod() {
4         // This could be a real implementation, but we are mocking it for
  testing
```

```
        return "Real Result";
    }
}
```

main code: MyDependency

```
// Example class MyService
class MyService {
    private final MyDependency dependency;

    MyService(MyDependency dependency) {
        this.dependency = dependency;
    }

    String someMethod() {
        // This method may use the MyDependency instance
        return dependency.someMethod();
    }
}
```

test code: MyServiceTest

- `MyDependency` is a simple class with a `someMethod` method that returns a string. In a real-world scenario, this class might represent a dependency that `MyService` interacts with.
- `MyService` is another class that takes `MyDependency` as a dependency and has a `someMethod` method that delegates to `MyDependency`.
- `MyServiceTest` is the test class where we use Mockito to mock `MyDependency` and verify the behavior of `MyService` when interacting with this mock.

```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

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

// Use @ExtendWith to enable the MockitoExtension in Junit5
@ExtendWith(MockitoExtension.class)
public class MyServiceTest {
    // Create a mock object for the dependency
```

```
13      @Mock
14      private MyDependency mockedDependency;
15
16      @Test
17      void testSomeMethod() {
18          // Set up the behavior of the mock
19          when(mockedDependency.someMethod()).thenReturn("Mocked Result");
20
21          // Create an instance of the class you want to test, passing in the
    mock
22          MyService myService = new MyService(mockedDependency);
23
24          // Call the method being tested
25          String result = myService.someMethod();
26
27          // Verify that the method under test behaved as expected
28          assertEquals("Mocked Result", result);
29
30          // Verify that the mock was called with specific arguments (if
    applicable)
31          verify(mockedDependency, times(1)).someMethod();
32      }
33  }
```

- We use the `@ExtendWith` annotation to enable the Mockito extension (`MockitoExtension`) for our test class. This extension initializes and manages mock objects for our tests.
- We declare a mock object `mockedDependency` using the `@Mock` annotation. This mock object represents a dependency of the class we want to test (`MyService`). We mark it as private because it's only used within this test class.
- **Stubbing Behavior:** We configure the behavior of the `mockedDependency` using `when(mockedDependency.someMethod()).thenReturn("Mocked Result")` . This means that when the `someMethod` of `mockedDependency` is called, it should return the string `"Mocked Result"` .
- We create an instance of the class we want to test, `MyService` , and pass in the `mockedDependency` as a dependency.
- We call the method `someMethod` on the `myService` instance and store the result in the `result` variable.
- **Assert Unit Test Result:** We use `assertEquals("Mocked Result", result)` to verify that the method under test (`someMethod`) produced the expected result, which is `"Mocked Result"` .

- **Verify Dependency:** We use `verify(mockedDependency, times(1)).someMethod()` to verify that the `someMethod` of the `mockedDependency` was called exactly once with specific arguments. This step ensures that our code under test correctly interacts with its dependencies.

@InjectMock & @Mock Example

main code: PaymentService

```
public interface PaymentService {
    boolean processPayment();
}

public class PaymentServiceImpl implements PaymentService {

    @Override
    public boolean processPayment() {
        // Simulate payment processing logic
        return true; // For simplicity, always assume payment succeeds
    }
}
```

main code: ShippingService

```
public interface ShippingService {
    boolean shipOrder();
}

public class ShippingServiceImpl implements ShippingService {
    @Override
    public boolean shipOrder() {
        // Simulate shipping logic
        return true; // For simplicity, always assume shipping succeeds
    }
}
```

main code: OrderService

```
public class OrderService {
    private PaymentService paymentService;
```

```
    private ShippingService shippingService;

    public OrderService(PaymentService paymentService, ShippingService shippingService) {
        this.paymentService = paymentService;
        this.shippingService = shippingService;
    }

    public boolean processOrder() {
        // Some complex logic involving paymentService and shippingService
        boolean paymentSuccess = paymentService.processPayment();
        if (!paymentSuccess)
            return false;
        boolean shippingSuccess = shippingService.shipOrder();
        return shippingSuccess;
    }
}
```

test code: OrderServiceTest

- `@Mock` is used to create mock instances of `PaymentService` and `ShippingService`.
- `@InjectMocks` is applied to the `orderService` field, indicating that mock instances of the dependencies should be injected into `orderService`.
- In the `setUp` method, we initialize the mocks using `MockitoAnnotations.initMocks(this)`.

With `@InjectMocks`, Mockito will automatically inject the mocked `PaymentService` and `ShippingService` into the `OrderService` instance, allowing you to test the behavior of `OrderService` while controlling the behavior of its dependencies.

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;

// We can use initMocks to replace @ExtendWith
// @ExtendWith(MockitoExtension.class)
public class OrderServiceTest {

    @Mock
    private PaymentService paymentService;

    @Mock
```

```
15      private ShippingService shippingService;
16
17      @InjectMocks
18      private OrderService orderService;
19
20      @BeforeEach
21      void setUp() {
22          MockitoAnnotations.initMocks(this); // Initialize the mocks
23      }
24
25      @Test
26      void testOrderProcessing() {
27          // Configure behavior of paymentService and shippingService mocks
28          when(paymentService.processPayment()).thenReturn(true);
29          when(shippingService.shipOrder()).thenReturn(true);
30
31          // Test the order processing logic in orderService
32          boolean result = orderService.processOrder();
33
34          // Assertions and test logic
35          assertTrue(result);
36      }
37  }
```

Benefits of Using Mockito

Isolation

Mockito helps you isolate the unit you're testing from its dependencies, making it easier to focus on the unit's behavior.

Repeatable Tests

Mockito allows you to simulate different scenarios and behaviors of dependencies, making your tests more comprehensive and repeatable.

Reduced Test Setup

You can avoid setting up complex real dependencies, which can be time-consuming and error-prone.

Improved Test Coverage

By mocking dependencies, you can simulate edge cases and error conditions that might be hard to reproduce with real dependencies.

In summary, Mockito is a powerful tool for unit testing in Java. It helps you create mock objects to isolate and test units of code independently. This allows you to thoroughly test your code's behavior in various scenarios and ensure that it works as expected, even when interacting with external dependencies.