**Introduction to Mockito**

Mockito is one of the most widely used testing frameworks in the Java ecosystem. It's designed to facilitate the creation of mock objects in unit tests. Mocking is essential for isolating the components under test by simulating behavior of dependencies, which helps in writing efficient and isolated unit tests.

**What is Mockito?**

Mockito is a **mocking framework** for unit testing in Java. It allows you to mock interfaces or classes, helping you isolate the class under test by simulating the behavior of its dependencies. Mockito focuses on making tests simple and readable, reducing boilerplate code, and increasing test reliability by controlling behavior and verifying interactions.

**Mockito Purpose**

* **Mocking Dependencies:** When unit testing a class that has dependencies, Mockito allows you to mock those dependencies and specify their behavior.
* **Simulate External Interactions:** It helps simulate how external systems behave without needing actual communication (e.g., database, files, web services).
* **Verification:** It verifies the behavior of your code (e.g., if certain methods were called, or with what parameters).
* **Isolation:** Isolate the class under test by replacing its collaborators with mocked versions.
* **Avoiding Heavy Resources:** Mocking external resources like APIs or databases in tests to prevent the tests from requiring actual resources or being dependent on them.

**Core Features of Mockito**

1. **Mocking:** Creating mock objects for interfaces or classes to simulate their behavior.
2. **Stubbing:** Defining the behavior of mock objects to return specific values or perform certain actions.
3. **Verifying:** Ensuring that specific interactions with mocks occurred during the test.
4. **Argument Matching:** Using matchers to flexibly define what should be matched during method invocations.
5. **Spy:** Allows partial mocking, where real methods are invoked unless they are stubbed.

**Setting Up Mockito with JUnit 5**

Before using Mockito, make sure to include the required dependencies in your pom.xml (Maven)

**Maven**

<dependencies>

<!-- JUnit 5 dependencies -->

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-api</artifactId>

<version>5.7.0</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-engine</artifactId>

<version>5.7.0</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-params</artifactId>

<version>5.7.0</version>

<scope>test</scope>

</dependency>

<!-- Mockito 3.x dependencies -->

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>3.11.2</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-junit-jupiter</artifactId>

<version>3.11.2</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-inline</artifactId>

<version>4.8.1</version>

<scope>test</scope>

</dependency>

</dependencies>

**Create a class to Mock**

**package** com.rit.mockito;

**public** **class** UserService {

**public** String getUserName(**int** userId) {

// This would fetch data from a database

// Logic is not yet implemented

**return** **null**;

}

**public** **boolean** sendSMS() {

System.***out***.println("SMS Sent");

// Logic is not yet implemented

**return** **true**;

}

**public** **static** **boolean** sendMail() {

System.***out***.println("Email Sent");

// Logic is not yet implemented

**return** **true**;

}

}

**Basic Mockito Operations**

**1. Mocking an Object**

You can mock an object of an interface or class using Mockito.mock().

**2. Stubbing Method Calls**

You can stub methods of the mock objects to return specific values when called.

package com.rit.mockito;

import static org.mockito.Mockito.mock;

import static org.mockito.Mockito.when;

import org.junit.jupiter.api.Test;

class T1MockTest {

@Test

void testMocking() {

// Mocking the UserService

UserService userService = mock(UserService.class);

// Stubbing Defining behavior of the mocked method

when(userService.getUserName(101)).thenReturn("Anand");

// Asserting that the mocked behavior works

assert userService.getUserName(101).equals("Anand");

}

}

**3. Argument Matching**

Mockito provides argument matchers like any(), eq(), etc., which allow you to match arguments flexibly.

**Common Argument Matchers**

1. **any()**: Matches any object.
2. **anyInt()**: Matches any int value.
3. **anyString()**: Matches any String value.
4. **eq()**: Matches the exact value.
5. **isNull()**: Matches null value.
6. **notNull()**: Matches any non-null value.
7. **startsWith(String prefix)**: Matches any string that starts with the given prefix.
8. **endsWith(String suffix)**: Matches any string that ends with the given suffix.
9. **contains(String substring)**: Matches any string that contains the given substring.

@Test

**void** testArgMatching() {

UserService userService = *mock*(UserService.**class**);

// Stubbing Defining behavior of the mocked method

*when*(userService.getUserName(*anyInt*())).thenReturn("Anand");

// Asserting that the mocked behavior works

**assert** userService.getUserName(101).equals("Anand");

}

**4. Verifying Method Calls**

You can verify if certain methods were called on the mock objects.

@Test

**void** testVerifying() {

UserService userService = *mock*(UserService.**class**);

userService.sendSMS();

// Verifying if above methods was called

Mockito.*verify*(userService).sendSMS();

}

**5. Using Spy for Partial Mocking**

A spy is a real object wrapped in a mock. It allows for partial mocking, where the real methods are called unless explicitly stubbed.

@Test

**void** testVerifying() {

UserService userService = **new** UserService();

// Creating spy - Partial mocking

UserService spyService = *spy*(userService);

//Real Method calls

spyService.sendSMS();

// Stubbing mocked method

*when*(spyService.getUserName(102)).thenReturn("Gayathri");

// Asserting that the mocked behavior works

**assert** spyService.getUserName(102).equals("Gayathri");

}

**6. Mocking Static Methods**

Mockito 3 introduced a feature that allows mocking static methods using Mockito.mockStatic().

package com.rit.mockito;

import static org.junit.jupiter.api.Assertions.assertTrue;

import static org.mockito.Mockito.mockStatic;

import static org.mockito.Mockito.times;

import org.junit.jupiter.api.Test;

import org.mockito.MockedStatic;

class T5StaticTest {

@Test

void testVerifying() {

try (MockedStatic<UserService> mockedStatic = mockStatic(UserService.class)) {

// Mock the static method behavior

mockedStatic.when(UserService::sendMail).thenReturn(true);

// Assert the mock behavior

assertTrue(UserService.sendMail());

// Verify the static method was called exactly once

mockedStatic.verify(UserService::sendMail);

// You can also verify that it was called a specific number of times

mockedStatic.verify(UserService::sendMail, times(1)); // verify once

}

}

}

**7. Mocking Void Methods**

package com.rit.mockito;

import static org.mockito.Mockito.doNothing;

import static org.mockito.Mockito.doThrow;

import static org.mockito.Mockito.mock;

import org.junit.jupiter.api.Test;

import org.mockito.Mockito;

class T6VoidMethodTest {

@Test

void testVoidMethod() {

// Create a mock object

MyClass myMock = mock(MyClass.class);

// Stubbing void method to do nothing

doNothing().when(myMock).voidMethod();

myMock.voidMethod();

Mockito.verify(myMock).voidMethod();

}

static class MyClass {

public void voidMethod() {

// Actual implementation of void method

}

}

}

**8. Stubbing Methods to throw Exception**

package com.rit.mockito;

import static org.mockito.Mockito.doThrow;

import static org.mockito.Mockito.mock;

import org.junit.jupiter.api.Test;

class T7ThrowExceptionTest {

@Test

void testVoidMethod() {

MyClass myMock = mock(MyClass.class);

// Stubbing method to throw an exception

doThrow(new RuntimeException("Exception thrown")).when(myMock).voidMethod();

try {

myMock.voidMethod(); // This will throw a RuntimeException

} catch (RuntimeException e) {

System.out.println("Caught exception: " + e.getMessage());

}

}

static class MyClass {

public void voidMethod() {

// Actual implementation of void method

}

}

}

**9. Injecting Mocks with Annotations**

Mockito provides annotations to reduce boilerplate code in setting up mocks.

package com.rit.mockito;

import static org.junit.jupiter.api.Assertions.assertEquals;

import static org.mockito.Mockito.when;

import org.junit.jupiter.api.Test;

import org.junit.jupiter.api.extension.ExtendWith;

import org.mockito.InjectMocks;

import org.mockito.Mock;

import org.mockito.junit.jupiter.MockitoExtension;

//Use MockitoExtension to automatically initialize mocks

@ExtendWith(MockitoExtension.class)

class T8AnnotationTest {

@Mock

private MyRepository myRepository;

@InjectMocks

private MyService myService;

@Test

void testGetData() {

when(myRepository.getData()).thenReturn("Mocked Data");

String result = myService.getData();

assertEquals("Mocked Data", result);

}

}

class MyRepository {

public String getData() {

return "Real Data";

}

}

class MyService {

private final MyRepository myRepository;

public MyService(MyRepository myRepository) {

this.myRepository = myRepository;

}

public String getData() {

return myRepository.getData();

}

}

**Mockito Best Practices**

1. **Use Mocks for Collaboration, Not Implementation Details:** Mocks should focus on the interactions between objects, not on the internal implementation of classes.
2. **Don’t Mock Everything:** Only mock dependencies that are external or complex to test, like databases or external APIs.
3. **Keep Tests Readable and Focused:** Keep the tests focused on what matters — the behavior of the system under test.
4. **Verify Interactions When Necessary:** Only verify interactions that are part of your test scenario. Over-verify, and it will make your tests fragile.