**MOCKITO EXERCISES**

**Exercise 1: Mocking and Stubbing** scenario using **Mockito**:

### **Scenario Recap**

We are testing a service (MyService) that internally uses an external API (ExternalApi). Since calling the actual API in a unit test is not ideal (due to unpredictability, slowness, or cost), we use **mocking** to simulate the API behavior and **stubbing** to define its return values.

### **Steps Explanation with Code**

#### **1. Create a mock object for the external API**

ExternalApi mockApi = Mockito.mock(ExternalApi.class);

* This line creates a **mock** instance of ExternalApi using Mockito.
* mockApi will now simulate the behavior of the real API.

#### **2. Stub the methods to return predefined values**

when(mockApi.getData()).thenReturn("Mock Data");

* This is **stubbing**. We are telling Mockito:  
  + "When getData() is called on mockApi, return the string "Mock Data"."
* This allows us to control the output of the mock during testing.

#### **3. Write a test case that uses the mock object**

MyService service = new MyService(mockApi);

String result = service.fetchData();

assertEquals("Mock Data", result);

* MyService is the class under test. It depends on ExternalApi.
* Instead of giving it the real API, we give it our **mock**.
* service.fetchData() internally calls mockApi.getData() and should receive "Mock Data" as we stubbed it.
* assertEquals("Mock Data", result) checks if the service behaves as expected with the mocked dependency.

### **Full Refined Code**

import static org.mockito.Mockito.\*;

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

import org.junit.jupiter.api.Test;

import org.mockito.Mockito;

// External API interface (assumed)

interface ExternalApi {

String getData();

}

// Service under test

class MyService {

private final ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public String fetchData() {

return api.getData();

}

}

public class MyServiceTest {

@Test

public void testExternalApi() {

// Step 1: Create mock

ExternalApi mockApi = Mockito.mock(ExternalApi.class);

// Step 2: Stub method

when(mockApi.getData()).thenReturn("Mock Data");

// Step 3: Inject mock and test

MyService service = new MyService(mockApi);

String result = service.fetchData();

// Assert the expected behavior

assertEquals("Mock Data", result);

}

}

This is a clean and complete JUnit test using **Mockito** for mocking and stubbing.

**Exercise 2: Verifying Interactions** using **Mockito**.

### **Scenario Recap**

We want to **verify** that a certain method of a mocked object was called, and optionally, whether it was called with specific arguments. This is useful for testing **behavior** rather than just the result.

### **Steps Explanation with Code**

#### **1. Create a mock object**

ExternalApi mockApi = Mockito.mock(ExternalApi.class);

* Creates a mock of the ExternalApi interface. This object will simulate the behavior of the real API.

#### **2. Call the method with specific arguments**

MyService service = new MyService(mockApi);

service.fetchData();

* MyService is the class under test.
* It takes mockApi as a dependency.
* When fetchData() is called, it internally calls mockApi.getData().

#### **3. Verify the interaction**

verify(mockApi).getData();

* This tells Mockito: “Verify that the method getData() was called on mockApi.”
* If getData() was not called, the test will fail.

### **Full Refined Code**

import static org.mockito.Mockito.\*;

import static org.junit.jupiter.api.Assertions.\*;

import org.junit.jupiter.api.Test;

import org.mockito.Mockito;

// External API interface (assumed)

interface ExternalApi {

String getData();

}

// Service under test

class MyService {

private final ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public String fetchData() {

return api.getData(); // This method call will be verified

}

}

public class MyServiceTest {

@Test

public void testVerifyInteraction() {

// Step 1: Create mock

ExternalApi mockApi = Mockito.mock(ExternalApi.class);

// Step 2: Inject mock and invoke method

MyService service = new MyService(mockApi);

service.fetchData();

// Step 3: Verify interaction

verify(mockApi).getData(); // Asserts that getData() was called exactly once

}

}

### **Notes**

If you want to verify that a method was called with specific arguments, you can pass them like:  
  
 verify(mockApi).methodName(expectedArg1, expectedArg2);

To verify how many times a method was called:  
  
 verify(mockApi, times(2)).getData();

To ensure no interaction:  
  
 verifyNoInteractions(mockApi);

This test ensures that the service behaves correctly by interacting with its dependency exactly as expected.

**Exercise 3: Argument Matching** using **Mockito**:

### **Scenario Recap**

We need to verify that a method of a mocked object is called with the **correct arguments**. In some cases, exact values might not be known or might vary, so we can use **argument matchers** (like any(), eq(), contains(), etc.) from Mockito.

### **Steps Explanation with Code**

#### **1. Create a mock object**

ExternalApi mockApi = Mockito.mock(ExternalApi.class);

* This creates a mock of the external dependency ExternalApi.

#### **2. Call the method with specific arguments**

MyService service = new MyService(mockApi);

service.sendData("Hello", 42);

* This is the method under test. It uses the mocked mockApi to call some method with parameters.
* The arguments "Hello" and 42 will be passed to the mock’s method.

#### **3. Use argument matchers to verify the interaction**

verify(mockApi).send("Hello", 42);

* This directly verifies if send() was called with "Hello" and 42.

Or, using **Mockito matchers**:

verify(mockApi).send(eq("Hello"), anyInt());

* eq("Hello") ensures the first argument was "Hello".
* anyInt() is a matcher that allows any integer for the second argument.

### **Full Working Code**

import static org.mockito.Mockito.\*;

import static org.junit.jupiter.api.Assertions.\*;

import org.junit.jupiter.api.Test;

import org.mockito.Mockito;

// External API interface (assumed)

interface ExternalApi {

void send(String message, int code);

}

// Service under test

class MyService {

private final ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public void sendData(String message, int code) {

api.send(message, code);

}

}

public class MyServiceTest {

@Test

public void testArgumentMatching() {

// Step 1: Create mock

ExternalApi mockApi = Mockito.mock(ExternalApi.class);

// Step 2: Inject and call method

MyService service = new MyService(mockApi);

service.sendData("Hello", 42);

// Step 3: Verify using argument matchers

verify(mockApi).send(eq("Hello"), anyInt());

}

}

### **Common Mockito Matchers**

| **Matcher** | **Description** |
| --- | --- |
| any() | Matches any object of any type |
| anyString() | Matches any String |
| anyInt() | Matches any int |
| eq(value) | Matches the exact value |
| contains(str) | Matches strings containing str |
| argThat(...) | Use a lambda or custom matcher |

This test uses argument matchers to assert that the mocked method was invoked with specific or flexible arguments, which is useful when testing dynamic or variable inputs.

**Exercise 4: Handling Void Methods** using **Mockito**.

### **Scenario Recap**

When dealing with **void methods** (i.e., methods that return nothing), you can't verify the result of a return value — instead, you test their **side effects** or verify if they were **called correctly**. In some cases, you also need to **stub** them if they throw exceptions or perform actions.

### **Steps Explanation with Code**

#### **1. Create a mock object**

ExternalApi mockApi = Mockito.mock(ExternalApi.class);

* This creates a mock for the dependency with the void method.

#### **2. Stub the void method**

doNothing().when(mockApi).logMessage(anyString());

* Since logMessage() returns void, you **can't use when(...).thenReturn(...)**.
* Instead, use doNothing().when(...) or doThrow().when(...) for void methods.
* This line tells Mockito to **do nothing** when logMessage() is called.

#### **3. Verify the interaction**

verify(mockApi).logMessage("Test message");

* Verifies that the method logMessage() was called with "Test message" as the argument.

### **Full Working Code**

import static org.mockito.Mockito.\*;

import static org.junit.jupiter.api.Assertions.\*;

import org.junit.jupiter.api.Test;

import org.mockito.Mockito;

// External API interface (assumed)

interface ExternalApi {

void logMessage(String message);

}

// Service under test

class MyService {

private final ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public void process() {

api.logMessage("Test message"); // This is the void method we want to test

}

}

public class MyServiceTest {

@Test

public void testVoidMethodInteraction() {

// Step 1: Create mock

ExternalApi mockApi = Mockito.mock(ExternalApi.class);

// Step 2: Stub the void method

doNothing().when(mockApi).logMessage(anyString());

// Step 3: Inject and call the method

MyService service = new MyService(mockApi);

service.process();

// Step 4: Verify interaction

verify(mockApi).logMessage("Test message");

}

}

### **Alternative: Stubbing a void method to throw an exception**

If you want the void method to throw an exception for testing:

doThrow(new RuntimeException("Error")).when(mockApi).logMessage(anyString());

Then test that your code handles the exception correctly.

This demonstrates how to stub and verify interactions with **void methods** using doNothing() and verify() in Mockito, which is essential for testing side-effect-based methods.

**Exercise 5: Mocking and Stubbing with Multiple Returns** using **Mockito**.

### **Scenario Recap**

We want to test a service that calls an external API multiple times. Each call to the same method should return **different values**, mimicking a real-world scenario like receiving updated data on each request.

### **Steps Explanation with Code**

#### **1. Create a mock object for the external API**

ExternalApi mockApi = Mockito.mock(ExternalApi.class);

* This creates a mock for the external API that the service depends on.

#### **2. Stub the method to return different values on consecutive calls**

when(mockApi.getStatus()).thenReturn("Pending", "Processing", "Completed");

* This tells Mockito:  
  + On **first call**, return "Pending"
  + On **second call**, return "Processing"
  + On **third call**, return "Completed"
* This is useful for simulating API behavior that changes over time (e.g., tracking status updates).

#### **3. Write a test case that uses the mock object**

String status1 = service.checkStatus(); // should return "Pending"

String status2 = service.checkStatus(); // should return "Processing"

String status3 = service.checkStatus(); // should return "Completed"

* The test verifies that the service correctly receives the changing responses from the API.

### **Full Working Code**

import static org.mockito.Mockito.\*;

import static org.junit.jupiter.api.Assertions.\*;

import org.junit.jupiter.api.Test;

import org.mockito.Mockito;

// External API interface (assumed)

interface ExternalApi {

String getStatus();

}

// Service under test

class MyService {

private final ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public String checkStatus() {

return api.getStatus();

}

}

public class MyServiceTest {

@Test

public void testMultipleReturns() {

// Step 1: Create mock

ExternalApi mockApi = Mockito.mock(ExternalApi.class);

// Step 2: Stub multiple return values

when(mockApi.getStatus()).thenReturn("Pending", "Processing", "Completed");

// Step 3: Inject mock and call method multiple times

MyService service = new MyService(mockApi);

String status1 = service.checkStatus();

String status2 = service.checkStatus();

String status3 = service.checkStatus();

// Assertions

assertEquals("Pending", status1);

assertEquals("Processing", status2);

assertEquals("Completed", status3);

}

}

### **Optional: Using thenAnswer() for complex logic**

If return values depend on inputs or call count, you can use thenAnswer():

AtomicInteger counter = new AtomicInteger(0);

when(mockApi.getStatus()).thenAnswer(invocation -> {

int count = counter.incrementAndGet();

return switch (count) {

case 1 -> "Pending";

case 2 -> "Processing";

default -> "Completed";

};

});

This test ensures that the mocked method can return **different values on subsequent calls**, mimicking real-time behavior in unit tests.

**Exercise 6: Verifying Interaction Order** using **Mockito**.

### **Scenario Recap**

You want to **verify that certain methods are invoked in a specific order**. This is useful when the **sequence** of operations matters — for example, opening a connection before sending data, or validating input before saving it.

### **Steps Explanation with Code**

#### **1. Create a mock object**

ExternalApi mockApi = Mockito.mock(ExternalApi.class);

* Creates a mock of the ExternalApi interface which will be used to track method calls.

#### **2. Call the methods in a specific order**

MyService service = new MyService(mockApi);

service.performOperation();

* This service method will internally call multiple methods on mockApi in a specific order.
* Example: authenticate() → fetchData() → logout()

#### **3. Verify the interaction order**

InOrder inOrder = inOrder(mockApi);

inOrder.verify(mockApi).authenticate();

inOrder.verify(mockApi).fetchData();

inOrder.verify(mockApi).logout();

* InOrder is a Mockito utility used to **enforce** call order verification.

### **Full Working Code**

import static org.mockito.Mockito.\*;

import static org.junit.jupiter.api.Assertions.\*;

import org.junit.jupiter.api.Test;

import org.mockito.InOrder;

// External API interface (assumed)

interface ExternalApi {

void authenticate();

void fetchData();

void logout();

}

// Service under test

class MyService {

private final ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public void performOperation() {

api.authenticate();

api.fetchData();

api.logout();

}

}

public class MyServiceTest {

@Test

public void testInteractionOrder() {

// Step 1: Create mock

ExternalApi mockApi = mock(ExternalApi.class);

// Step 2: Inject and perform operations

MyService service = new MyService(mockApi);

service.performOperation();

// Step 3: Verify interaction order

InOrder inOrder = inOrder(mockApi);

inOrder.verify(mockApi).authenticate();

inOrder.verify(mockApi).fetchData();

inOrder.verify(mockApi).logout();

}

}

### **Notes**

* If methods are called out of order, the test will **fail**.

You can also verify **interactions across multiple mocks** by passing them to inOrder():  
  
 InOrder inOrder = inOrder(mock1, mock2);

This test ensures that the mocked methods are invoked in the **exact sequence** expected, which is crucial when the order of operations has functional significance.

**Exercise 7: Handling Void Methods with Exceptions** using **Mockito**.

### **Scenario Recap**

In this case, you want to **test a void method** (i.e., a method that returns nothing) which may **throw an exception**. This is common when the method performs an operation like writing to a file, logging, or deleting a resource — and it can fail.

Mockito provides the doThrow() method to simulate exceptions for void methods.

### **Steps Explanation with Code**

#### **1. Create a mock object**

ExternalApi mockApi = Mockito.mock(ExternalApi.class);

* This creates a mock of ExternalApi, which includes the void method that might throw an exception.

#### **2. Stub the void method to throw an exception**

doThrow(new RuntimeException("Logging failed")).when(mockApi).logMessage("Error occurred");

* For void methods, use doThrow(...).when(...) instead of when(...).thenThrow(...).
* This line tells Mockito to throw a RuntimeException when logMessage("Error occurred") is called.

#### **3. Verify the interaction**

verify(mockApi).logMessage("Error occurred");

* Even if the method throws an exception, you can still verify that it was called.

### **Full Working Code**

import static org.mockito.Mockito.\*;

import static org.junit.jupiter.api.Assertions.\*;

import org.junit.jupiter.api.Test;

import org.mockito.Mockito;

// External API interface (assumed)

interface ExternalApi {

void logMessage(String message);

}

// Service under test

class MyService {

private final ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public void handleError() {

api.logMessage("Error occurred");

}

}

public class MyServiceTest {

@Test

public void testVoidMethodThrowsException() {

// Step 1: Create mock

ExternalApi mockApi = Mockito.mock(ExternalApi.class);

// Step 2: Stub to throw exception

doThrow(new RuntimeException("Logging failed"))

.when(mockApi)

.logMessage("Error occurred");

// Step 3: Inject and call method

MyService service = new MyService(mockApi);

// Verify exception is thrown

RuntimeException exception = assertThrows(RuntimeException.class, () -> {

service.handleError();

});

assertEquals("Logging failed", exception.getMessage());

// Step 4: Verify interaction

verify(mockApi).logMessage("Error occurred");

}

}