**WEEK-2 Mockito Hands-On Exercises**

**Exercise 1: Mocking and Stubbing**

Scenario:

You need to test a service that depends on an external API. Use Mockito to mock the

external API and stub its methods.

Steps:

1. Create a mock object for the external API.

2. Stub the methods to return predefined values.

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

**ExternalApi.java:**

package com.example;

public interface ExternalApi {

String getData();

}

**MyService.java:**

**package** com.example;

**public** **class** MyService {

**private** ExternalApi api;

**public** MyService(ExternalApi api) {

**this**.api = api;

}

**public** String fetchData() {

**return** api.getData();

}

}

**MyServiceTest.java:**

**package** com.example;

**import** org.junit.Test;

**import** **static** org.junit.Assert.\*;

**import** **static** org.mockito.Mockito.\*;

**public** **class** MyServiceTest {

@Test

**public** **void** testExternalApi() {

// Step 1: Create mock

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

// Step 2: Stub the method

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

// Step 3: Inject mock and test

MyService service = **new** MyService(mockApi);

String result = service.fetchData();

// Step 4: Assert the result

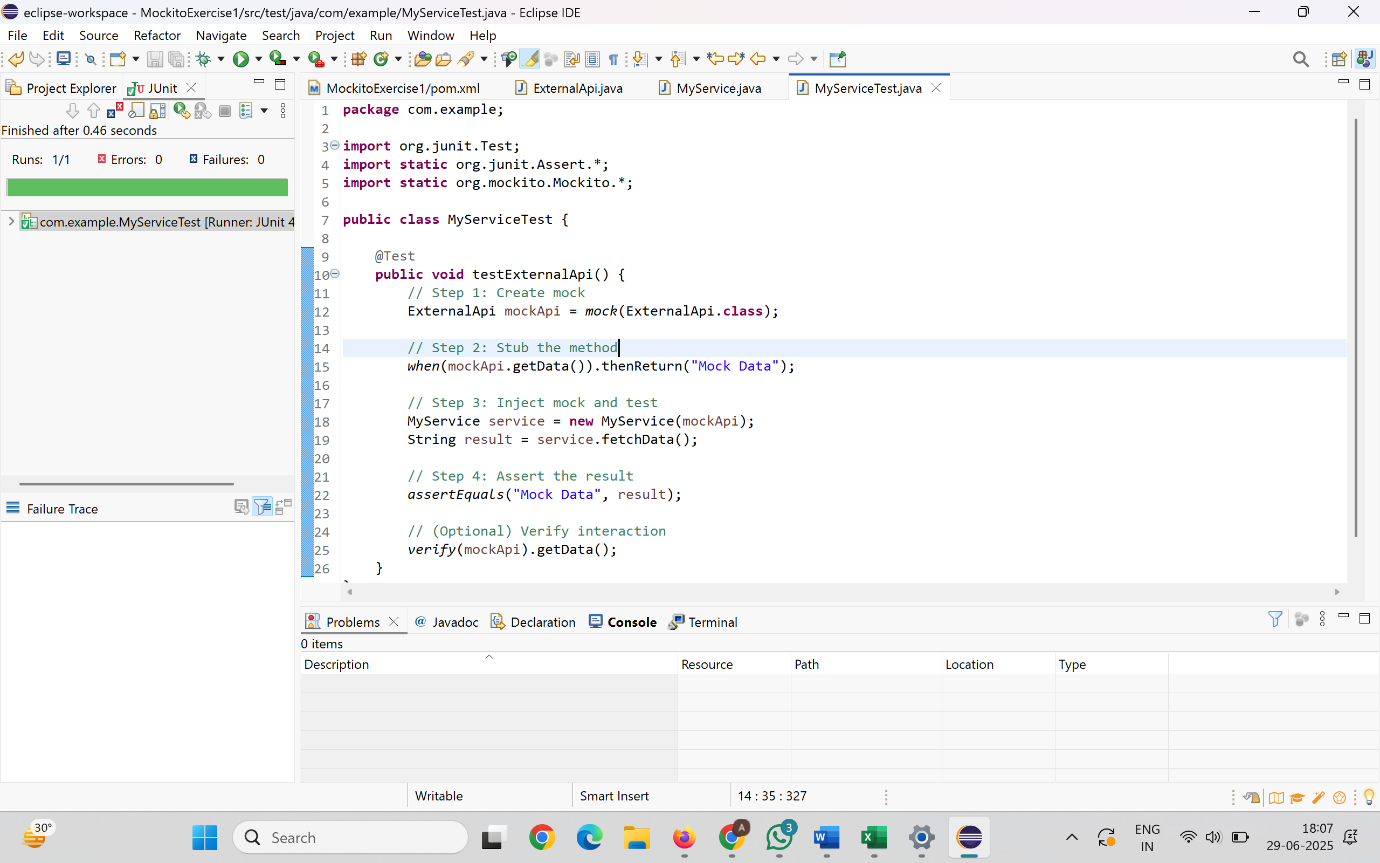
*assertEquals*("Mock Data", result);

// (Optional) Verify interaction

*verify*(mockApi).getData();

}

}



**Exercise 2: Verifying Interactions**

Scenario:

You need to ensure that a method is called with specific arguments.

Steps:

1. Create a mock object.

2. Call the method with specific arguments.

3. Verify the interaction.

**MyServiceTest.java:**

package com.example;

import org.junit.Test;

import static org.mockito.Mockito.\*;

public class MyServiceTest {

@Test

public void testVerifyInteraction() {

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

MyService service = new MyService(mockApi);

String data = service.fetchData();

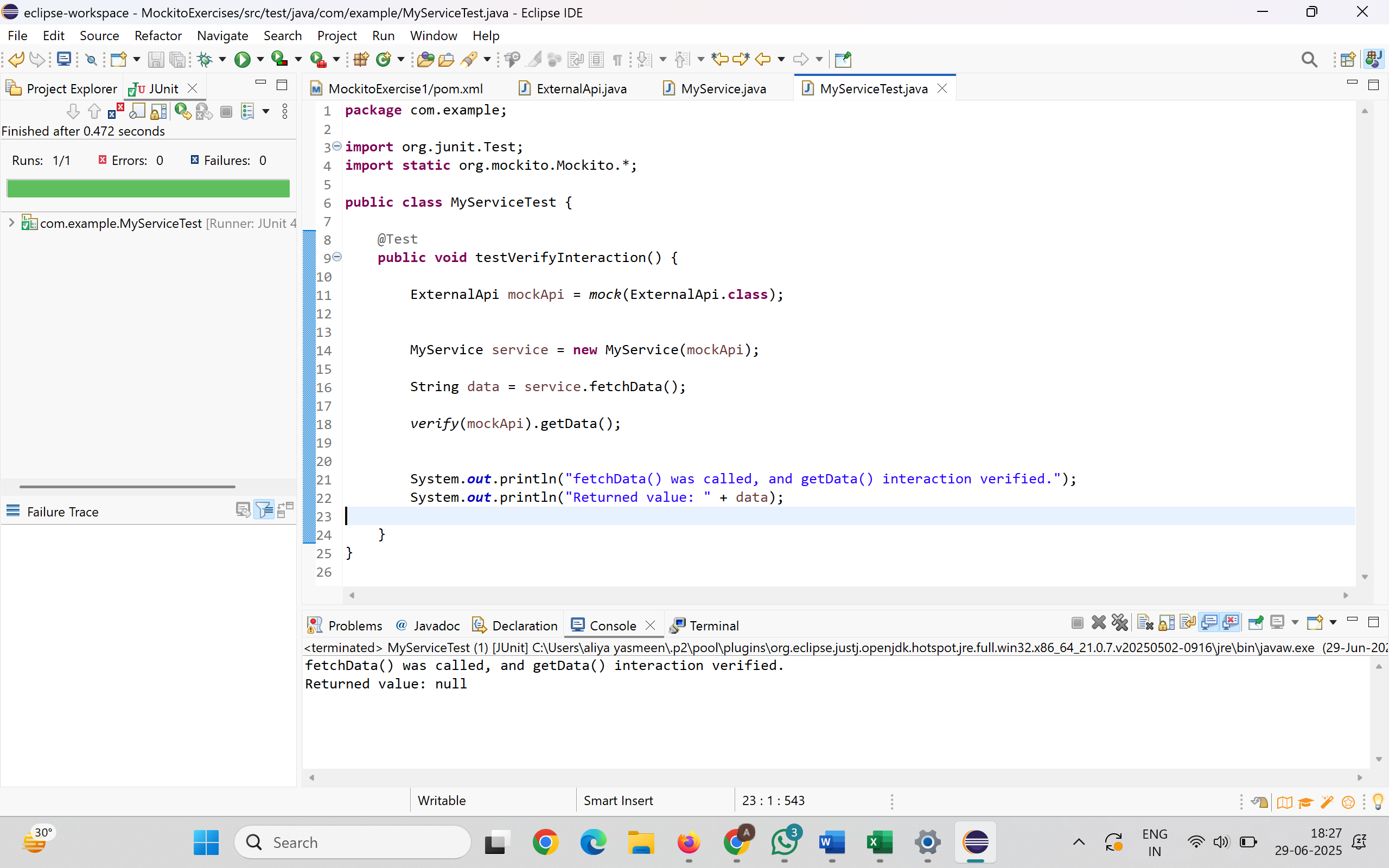
*verify*(mockApi).getData();

System.*out*.println("fetchData() was called, and getData() interaction verified.");

System.*out*.println("Returned value: " + data);

}

}



**Exercise 3: Argument Matching**

Scenario:

You need to verify that a method is called with specific arguments.

Steps:

1. Create a mock object.

2. Call the method with specific arguments.

3. Use argument matchers to verify the interaction.

UserNotifier.java:

**package** com.example;

**public** **interface** UserNotifier {

**void** sendNotification(String userId, String message);

}

**NotificationService.java:**

**package** com.example;

**public** **class** NotificationService {

**private** UserNotifier notifier;

**public** NotificationService(UserNotifier notifier) {

**this**.notifier = notifier;

}

**public** **void** notifyUser(String userId) {

String message = "Hello " + userId + ", you have a new message!";

notifier.sendNotification(userId, message);

}

}

**NotificationServiceTest.java:**

**package** com.example;

**import** org.junit.Test;

**import** **static** org.mockito.Mockito.\*;

**import** **static** org.mockito.ArgumentMatchers.\*;

**public** **class** NotificationServiceTest {

@Test

**public** **void** testSendNotificationWithCorrectArguments() {

UserNotifier mockNotifier = *mock*(UserNotifier.**class**);

NotificationService service = **new** NotificationService(mockNotifier);

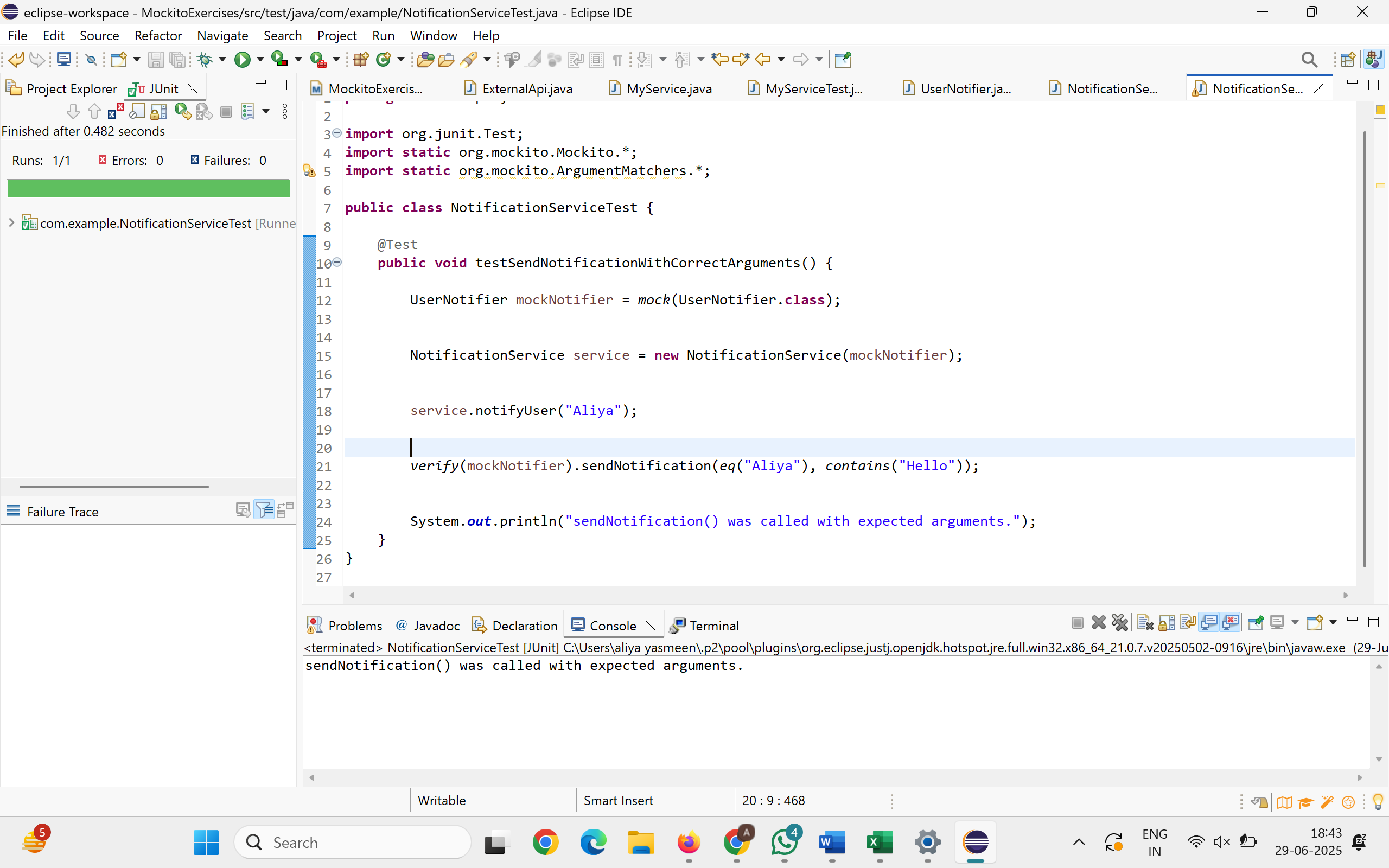
service.notifyUser("Aliya");

*verify*(mockNotifier).sendNotification(*eq*("Aliya"), *contains*("Hello"));

System.***out***.println("sendNotification() was called with expected arguments.");

}

}



**Exercise 4: Handling Void Methods**

Scenario:

You need to test a void method that performs some action.

Steps:

1. Create a mock object.

2. Stub the void method.

3. Verify the interaction.

**LoggerService.java:**

**package** com.example;

**public** **interface** LoggerService {

**void** log(String message);

}

**AuditManager.java:**

**package** com.example;

**public** **class** AuditManager {

**private** LoggerService logger;

**public** AuditManager(LoggerService logger) {

**this**.logger = logger;

}

**public** **void** recordAction(String action) {

// Action + log it

logger.log("Action recorded: " + action);

}

}

**AuditManagerTest.java:**

**package** com.example;

**import** org.junit.Test;

**import** **static** org.mockito.Mockito.\*;

**public** **class** AuditManagerTest {

@Test

**public** **void** testVoidMethodLogging() {

LoggerService mockLogger = *mock*(LoggerService.**class**);

AuditManager manager = **new** AuditManager(mockLogger);

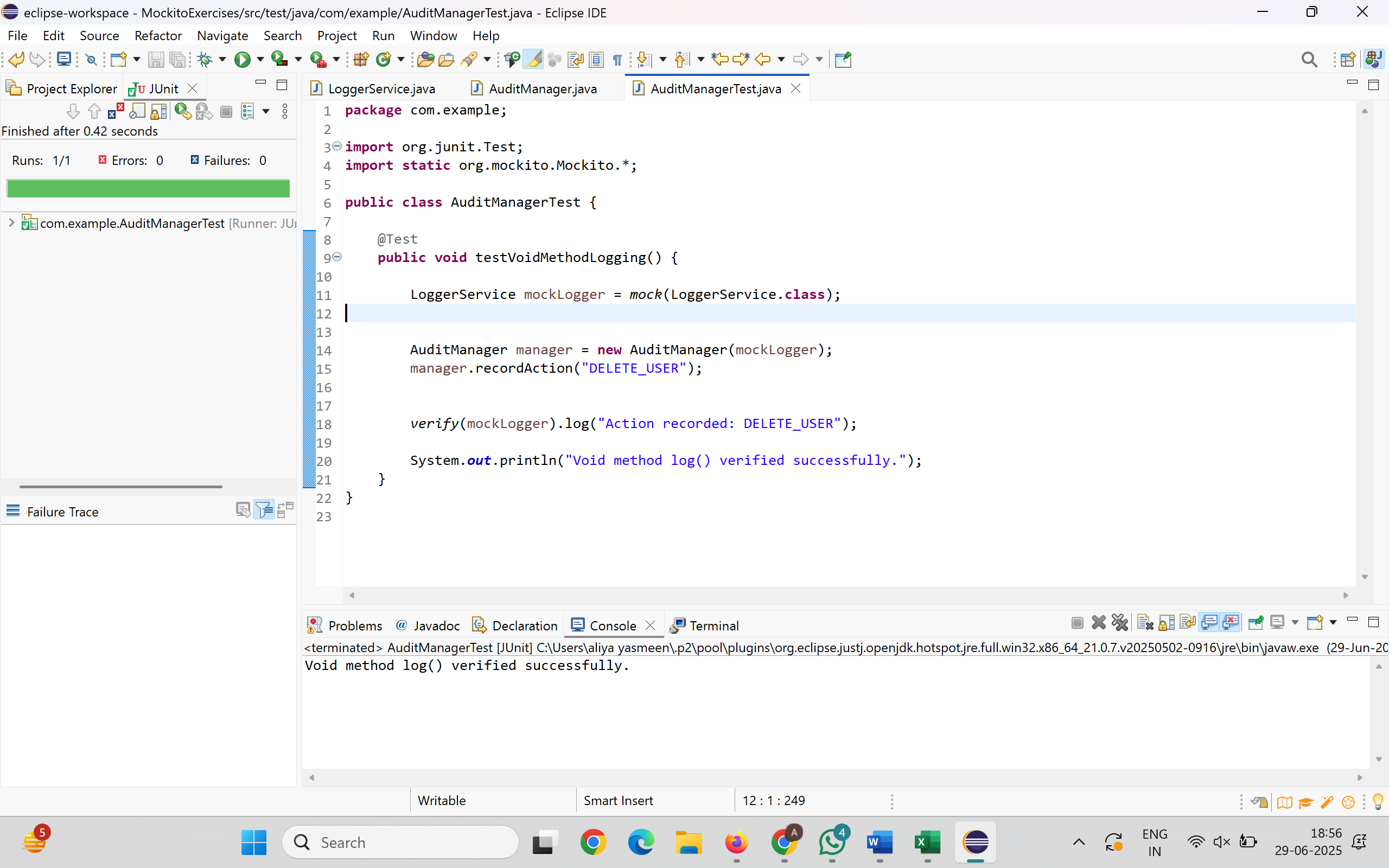
manager.recordAction("DELETE\_USER");

*verify*(mockLogger).log("Action recorded: DELETE\_USER");

System.***out***.println("Void method log() verified successfully.");

}

}



**Exercise 5: Mocking and Stubbing with Multiple Returns**

Scenario:

You need to test a service that depends on an external API with multiple return values.

Steps:

1. Create a mock object for the external API.

2. Stub the methods to return different values on consecutive calls.

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

**ExternalApi.java:**

**package** com.example;

**public** **interface** ExternalApi {

String getNextMessage();

}

**MessageService.java:**

**package** com.example;

**public** **class** MessageService {

**private** ExternalApi api;

**public** MessageService(ExternalApi api) {

**this**.api = api;

}

**public** String[] fetchMessages() {

**return** **new** String[] {

api.getNextMessage(),

api.getNextMessage(),

api.getNextMessage()

};

}

}

**MessageServiceTest.java:**

**package** com.example;

**import** org.junit.Test;

**import** **static** org.junit.Assert.\*;

**import** **static** org.mockito.Mockito.\*;

**public** **class** MessageServiceTest {

@Test

**public** **void** testMultipleReturns() {

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

*when*(mockApi.getNextMessage())

.thenReturn("Message 1")

.thenReturn("Message 2")

.thenReturn("Message 3");

MessageService service = **new** MessageService(mockApi);

String[] results = service.fetchMessages();

*assertArrayEquals*(

**new** String[] { "Message 1", "Message 2", "Message 3" },

results

);

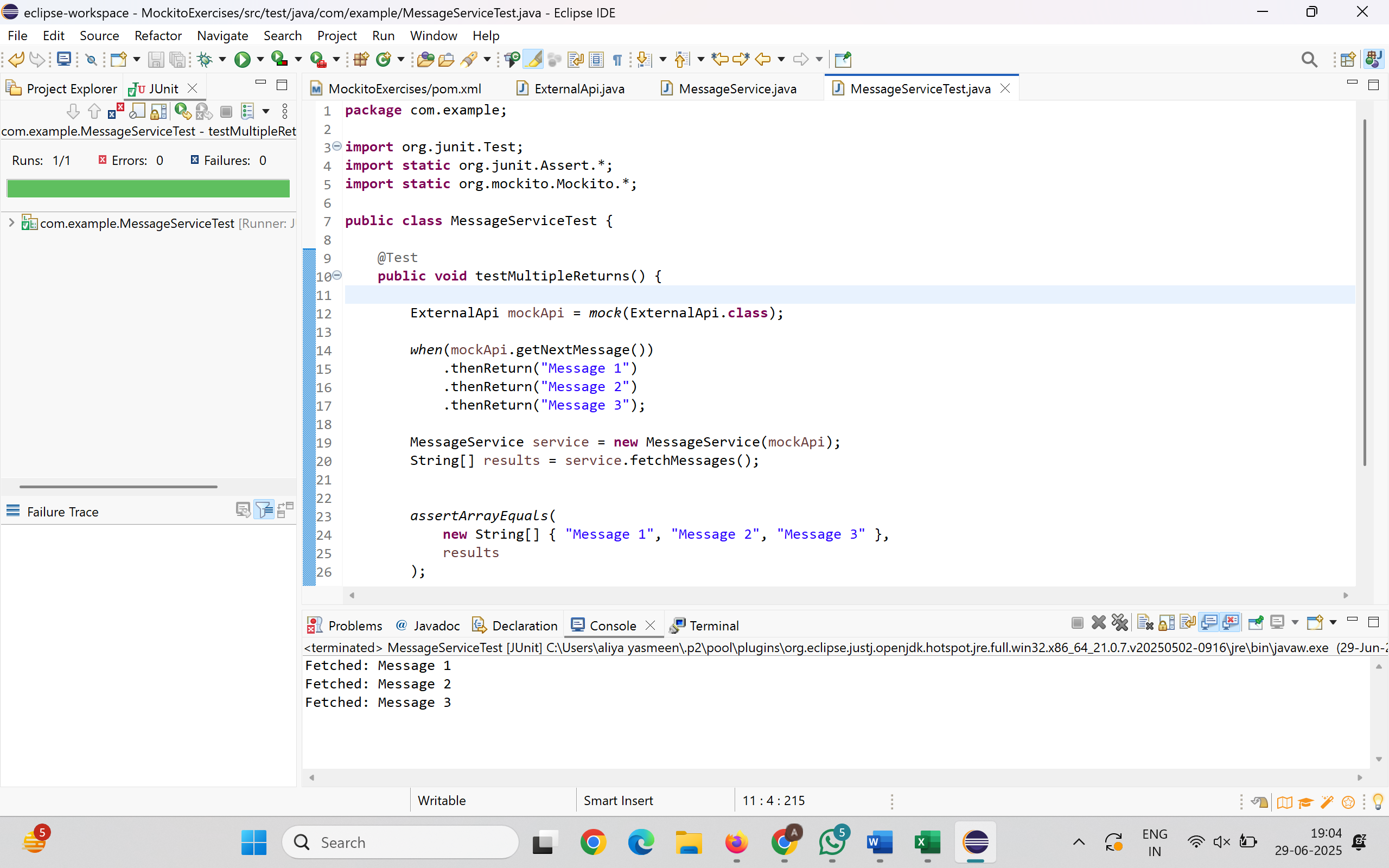
**for** (String msg : results) {

System.***out***.println("Fetched: " + msg);

}

}

}



**Exercise 6: Verifying Interaction Order**

Scenario:

You need to ensure that methods are called in a specific order.

Steps:

1. Create a mock object.

2. Call the methods in a specific order.

3. Verify the interaction order.

**TaskManager .java:**

**package** com.example;

**public** **interface** TaskManager {

**void** start();

**void** execute();

**void** finish();

}

**WorkflowService .java:**

**package** com.example;

**public** **class** WorkflowService {

**private** TaskManager manager;

**public** WorkflowService(TaskManager manager) {

**this**.manager = manager;

}

**public** **void** runWorkflow() {

manager.start();

manager.execute();

manager.finish();

}

}

**WorkflowServiceTest .java:**

**package** com.example;

**import** org.junit.Test;

**import** **static** org.mockito.Mockito.\*;

**import** org.mockito.InOrder;

**public** **class** WorkflowServiceTest {

@Test

**public** **void** testMethodCallOrder() {

TaskManager mockManager = *mock*(TaskManager.**class**);

WorkflowService service = **new** WorkflowService(mockManager);

service.runWorkflow();

InOrder inOrder = *inOrder*(mockManager);

inOrder.verify(mockManager).start();

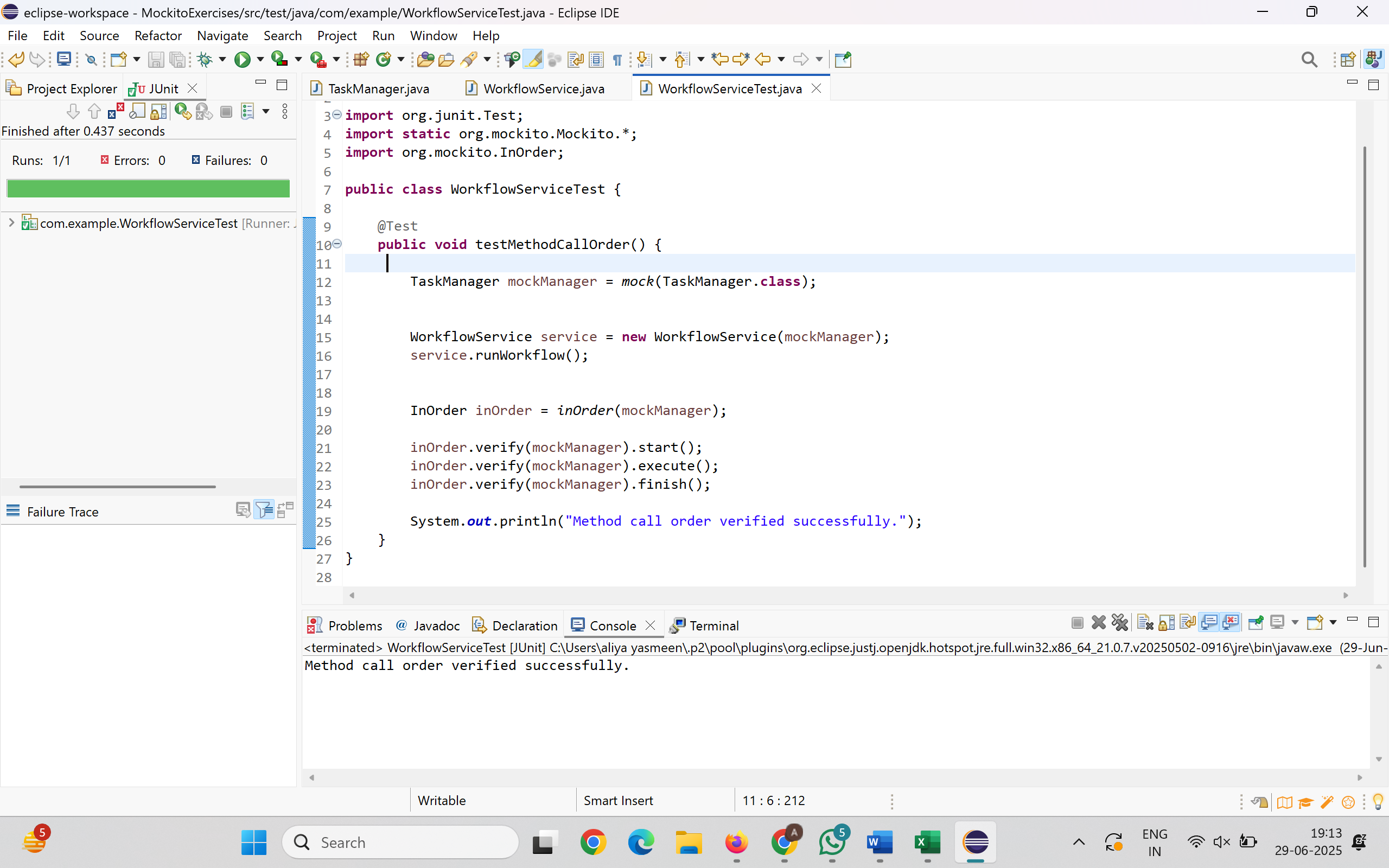
inOrder.verify(mockManager).execute();

inOrder.verify(mockManager).finish();

System.***out***.println("Method call order verified successfully.");

}

}



**Exercise 7: Handling Void Methods with Exceptions**

Scenario:

You need to test a void method that throws an exception.

Steps:

1. Create a mock object.

2. Stub the void method to throw an exception.

3. Verify the interaction.

**AlertService.java:**

**package** com.example;

**public** **interface** AlertService {

**void** sendAlert(String message);

}

**CriticalSystem.java:**

**package** com.example;

**public** **class** CriticalSystem {

**private** AlertService alertService;

**public** CriticalSystem(AlertService alertService) {

**this**.alertService = alertService;

}

**public** **void** performCriticalOperation() {

alertService.sendAlert("CRITICAL: System failure!");

}

}

**CriticalSystemTest .java:**

**package** com.example;

**import** org.junit.Test;

**import** **static** org.mockito.Mockito.\*;

**import** **static** org.junit.Assert.\*;

**public** **class** CriticalSystemTest {

@Test

**public** **void** testSendAlertThrowsException() {

AlertService mockAlert = *mock*(AlertService.**class**);

*doThrow*(**new** RuntimeException("Alert service failed"))

.when(mockAlert).sendAlert("CRITICAL: System failure!");

CriticalSystem system = **new** CriticalSystem(mockAlert);

**try** {

system.performCriticalOperation();

*fail*("Expected RuntimeException was not thrown");

} **catch** (RuntimeException ex) {

*assertEquals*("Alert service failed", ex.getMessage());

System.***out***.println("Exception caught and verified successfully.");

}

*verify*(mockAlert).sendAlert("CRITICAL: System failure!");

}

}

