**Module 4 – Test driven development and Logging framework**

**Junit Basic Testing Exercises**

**Exercise 1: Setting Up Junit**

**Scenario:**

**You need to set up JUnit in your Java project to start writing unit tests.**

**Steps:**

1. Create a new Java project in your IDE (e.g., IntelliJ IDEA, Eclipse).

2. Add JUnit dependency to your project.

3. Create a new test class in your project.

**Project Setup**

* I used **IntelliJ IDEA with Maven**
* JUnit 4.13.2 was added via pom.xml
* File structure:

src/main/java/NumberChecker.java

src/test/java/NumberCheckerTest.java

POM.XML: (Added dependency code snippet to enable Junit)

<dependencies>  
 <dependency>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 <version>4.13.2</version>  
 <scope>test</scope>  
 </dependency>  
</dependencies>

NumberChecker.java

public class NumberChecker {

public boolean isEven(int number) {

return number % 2 == 0;

}

}

NumberCheckerTest.java

import static org.junit.Assert.assertTrue;

import static org.junit.Assert.assertFalse;

import org.junit.Test;

public class NumberCheckerTest {

NumberChecker checker = new NumberChecker();

@Test

public void testIsEven() {

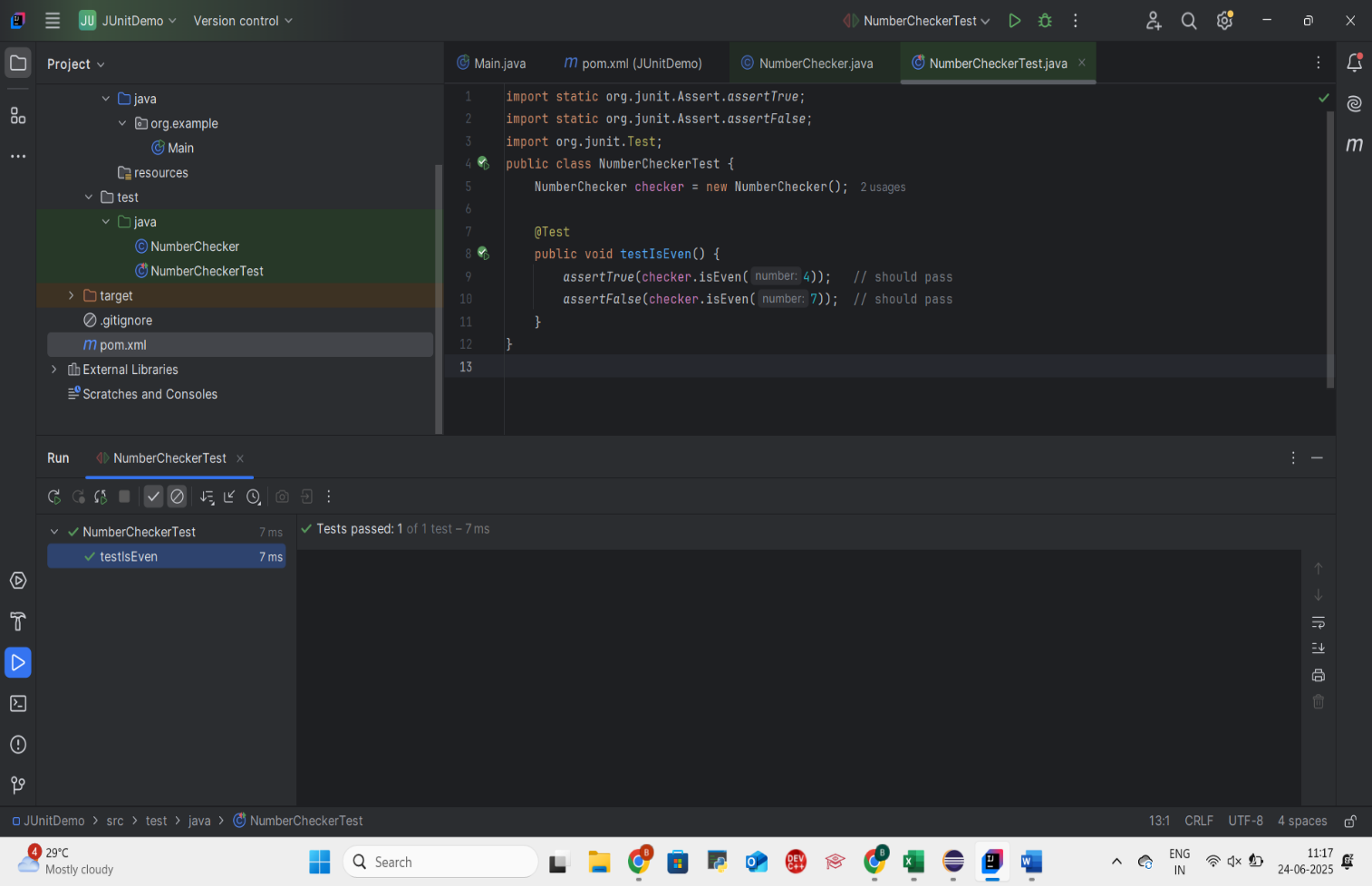
assertTrue(checker.isEven(4)); // Passes

assertFalse(checker.isEven(7)); // Passes

}

}

**OUTPUT SCREENSHOT:**



**JUnit was successfully set up using Maven in IntelliJ IDEA.**

**The test class was executed with no failures.**

**Exercise 3: Assertions in Junit**

**Scenario:**

**You need to use different assertions in JUnit to validate your test results. Steps:**

**Task:**

Write a test class that uses the following JUnit assertions:

- assertEquals

- assertTrue

- assertFalse

- assertNull

- assertNotNull

**Implementation:**

CODE:

import static org.junit.Assert.\*;

import org.junit.Test;

public class AssertionsTest {

@Test

public void testAssertions() {

assertEquals(5, 2 + 3); // Checks sum

assertTrue(5 > 3); // True condition

assertFalse(5 < 3); // False condition

assertNull(null); // Checks null

assertNotNull(new Object()); // Checks not null

}

}

EXPLANATION:

**assertEquals(expected, actual)**  
Verifies that two values are equal.

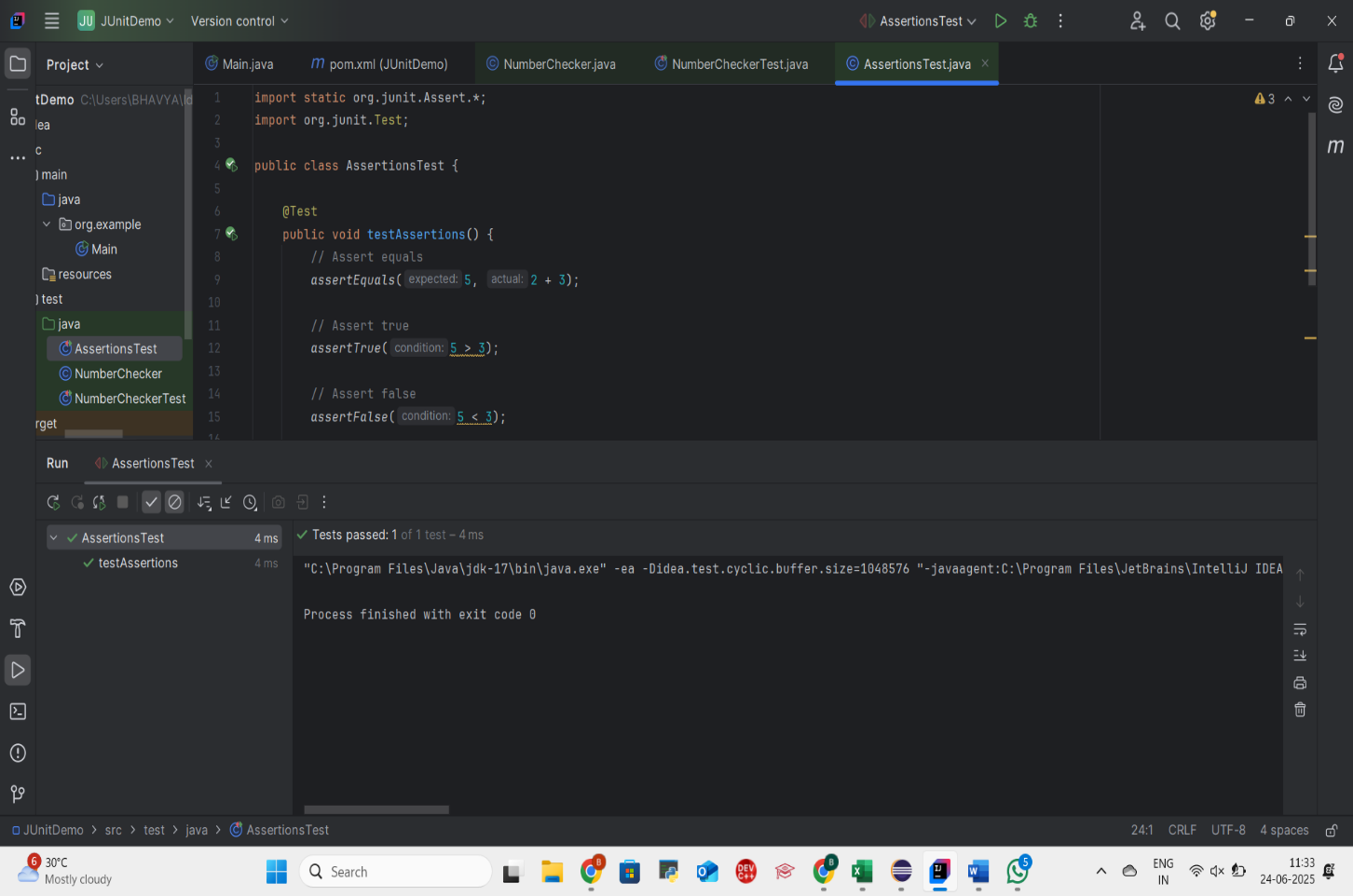
**assertTrue(condition)**  
Asserts that the given condition is true.

**assertFalse(condition)**  
Asserts that the given condition is false.

**assertNull(object)**  
Checks that the object is null.

**assertNotNull(object)**  
Checks that the object is **not** null.

**OUTPUT SCREENSHOT:**



**Assertions are crucial for validating test outcomes.**

**All assertions passed successfully, confirming correct implementation.**

**Exercise 4: Arrange-Act-Assert (AAA) Pattern, Test Fixtures, Setup and**

**Teardown Methods in JUnit**

**Scenario:**

**You need to organize your tests using the Arrange-Act-Assert (AAA) pattern and use setup and teardown methods.**

**Steps:**

1. Write tests using the AAA pattern.

2. Use @Before and @After annotations for setup and teardown methods.

**IMPLEMENTATION:**

**CODE:**

**Class Under Test – Calculator.java**

public class Calculator {

public int multiply(int a, int b) {

return a \* b;

}

public int divide(int a, int b) {

return a / b;

}

}

**JUnit Test Class – CalculatorTest.java**

import org.junit.After;

import org.junit.Before;

import org.junit.Test;

import static org.junit.Assert.assertEquals;

public class CalculatorTest {

Calculator calc;

@Before

public void setUp() {

// Arrange: Initialize resources

calc = new Calculator();

System.out.println("Setting up test environment");

}

@After

public void tearDown() {

// Clean up

System.out.println("Cleaning up after test");

}

@Test

public void testMultiply() {

// Act

int result = calc.multiply(4, 5);

// Assert

assertEquals(20, result);

}

@Test

public void testDivide() {

// Act

int result = calc.divide(10, 2);

// Assert

assertEquals(5, result);

}

}

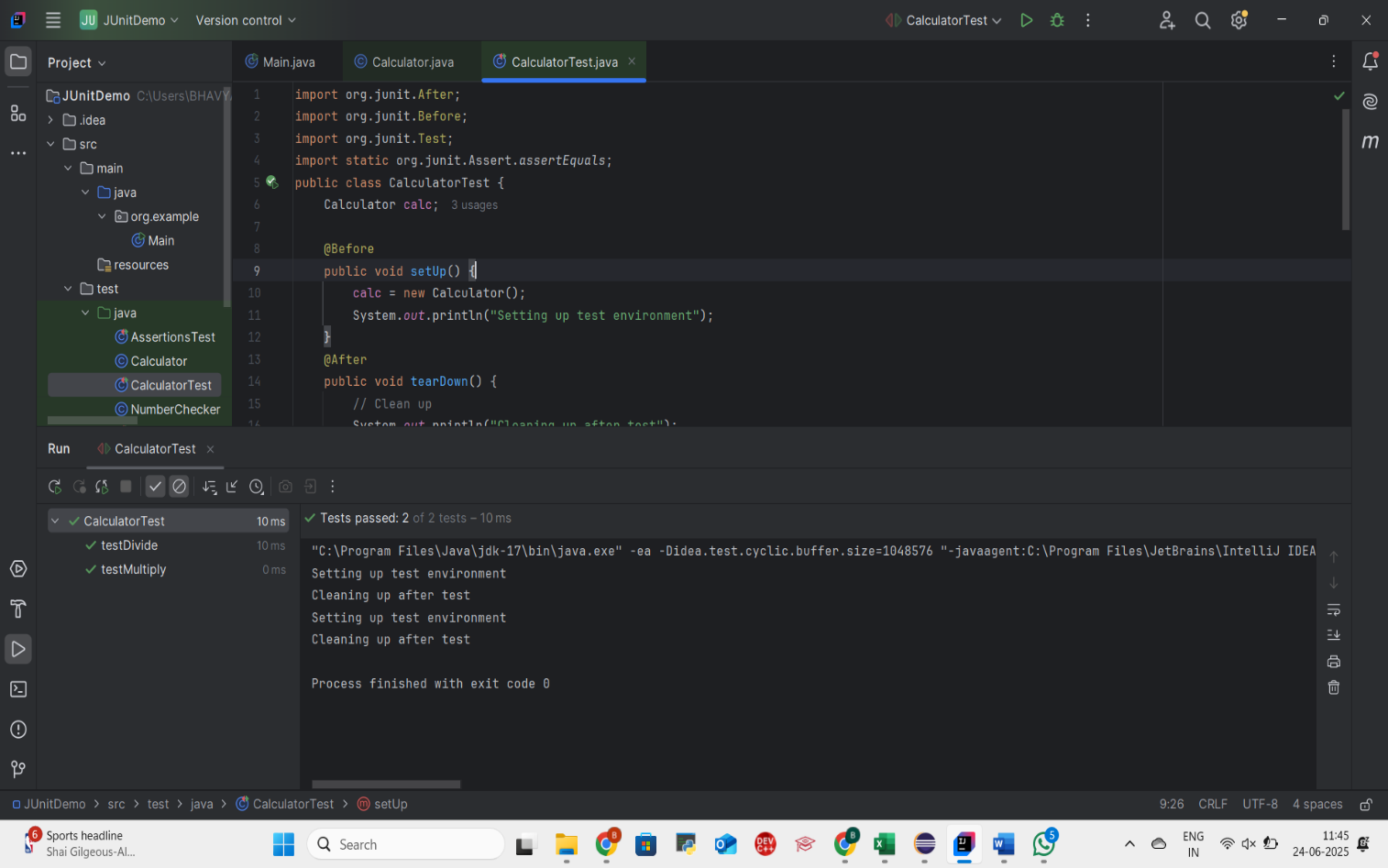
**JUnit Concepts Used**

* @Before: runs before each test method
* @After: runs after each test method
* AAA Pattern:
  + **Arrange**: Setup (@Before)
  + **Act**: Call method to test
  + **Assert**: Check result

**Explanation:**

* This test class follows the Arrange-Act-Assert (AAA) pattern for organizing test logic.
* - @Before method (`setUp`) is used to prepare the test environment before each test.
* - @After method (`tearDown`) cleans up after each test.
* - Each test method follows a clean structure:
* - Arrange (object setup in @Before)
* - Act (method call)
* - Assert (validate result)
* This improves test readability and reusability.

**OUTPUT SCREENSHOT:**

****

**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.

**Mockito Setup in IntelliJ: (Added in POM.XML file)**

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>4.8.0</version>

<scope>test</scope>

</dependency>

<dependency>

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

<artifactId>junit-jupiter</artifactId>

<version>5.9.2</version>

<scope>test</scope>

</dependency>

**IMPLEMENTATION:**

We'll work with three classes:

1. **ExternalApi.java – The interface to be mocked**

public interface ExternalApi {

String getData();

}

1. **MyService.java – The class that depends on ExternalApi**

public class MyService {

private ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public String fetchData() {

return api.getData();

}

}

1. **MyServiceTest.java – The JUnit + Mockito test class**

import static org.mockito.Mockito.\*;

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

import org.junit.jupiter.api.Test;

import org.mockito.Mockito;

public class MyServiceTest {

@Test

public void testExternalApi() {

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

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

MyService service = new MyService(mockApi);

String result = service.fetchData();

assertEquals("Mock Data", result);

}

}

**OUTPUT SCREENSHOT:**

A screenshot of a computer

AI-generated content may be incorrect.

**Successful unit test with mocked and stubbed external API using Mockito.Mockito was used to create a mock of an external API and stub its method `getData()` to return a predefined value. This allowed unit testing the `MyService` class in isolation. The test passed successfully, validating both mocking and stubbing concepts.**

**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.

**IMPLEMENTATION:**

**ExternalApi.java**

public interface ExternalApi {

String getData();

}

**MyService.java**

public class MyService {

private ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public String fetchData() {

return api.getData();

}

}

**MyServiceVerifyTest.java**

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

import org.mockito.Mockito;

public class MyServiceVerifyTest {

@Test

public void testVerifyInteraction() {

// Step 1: Mock the dependency

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

// Step 2: Call method on service

MyService service = new MyService(mockApi);

service.fetchData();

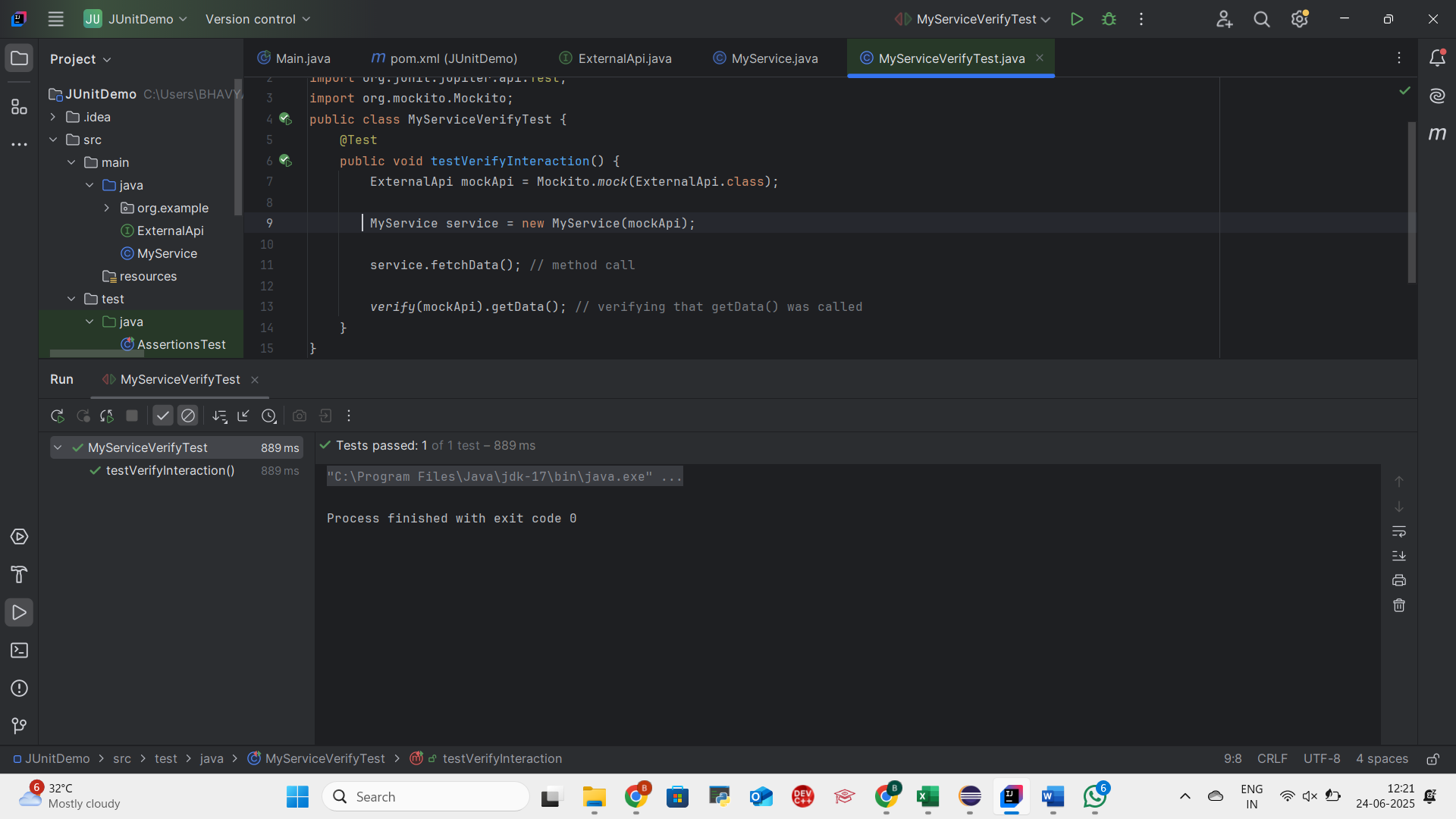
// Step 3: Verify interaction

verify(mockApi).getData();

}

}

**OUTPUT SCREENSHOT:**

****

**This test demonstrates how Mockito can be used to verify that specific method calls occurred.**

**The `verify(mockApi).getData()` statement confirms that `fetchData()` correctly interacted with the dependency.**