**Superset ID:6373322**

**COGNITZANT DIGITAL NURTURE 4.0 JAVA FSE**

**WEEK-2: TDD Using Junit and Mockito**

**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. If you are using Maven, add the following to your

pom.xml:

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13.2</version>

<scope>test</scope>

</dependency>

3. Create a new test class in your project.

**CODE:**

**Pom.xml:**

<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.example1</groupId>

<artifactId>JUnitSetupExample</artifactId>

<version>0.0.1-SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13.2</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>5.12.0</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>1.7.30</version>

</dependency>

<dependency>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-classic</artifactId>

<version>1.2.3</version>

</dependency>

</dependencies>

</project>

**Calculator.java**

package myapp;

public class Calculator {

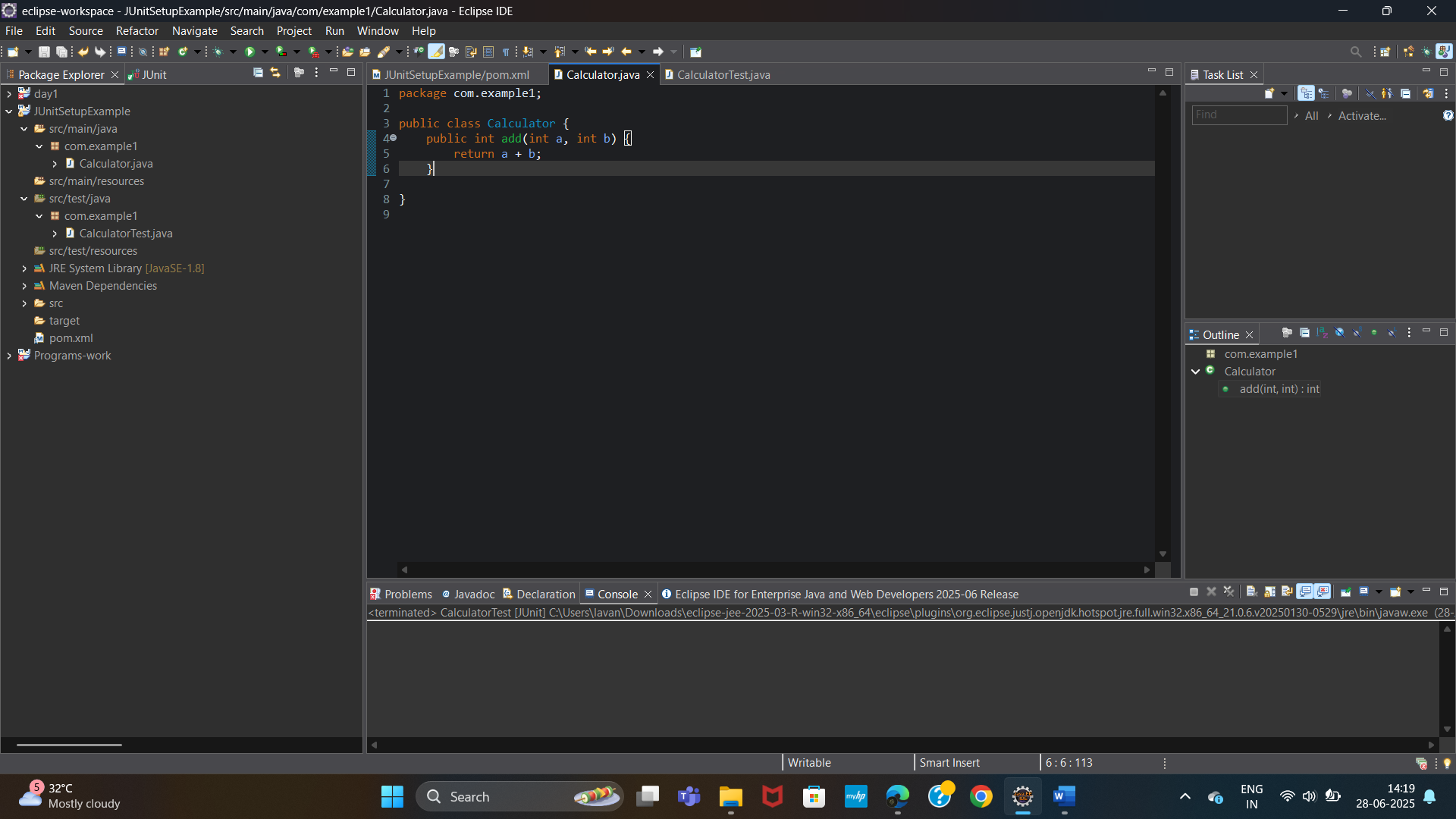
public int add(int a, int b) {

return a + b;

}

}

**OUTPUT:**



**CalculatorTest.java**

package myapp

import static org.junit.Assert.\*;

import org.junit.Test;

public class CalculatorTest {

@Test

public void testAdd() {

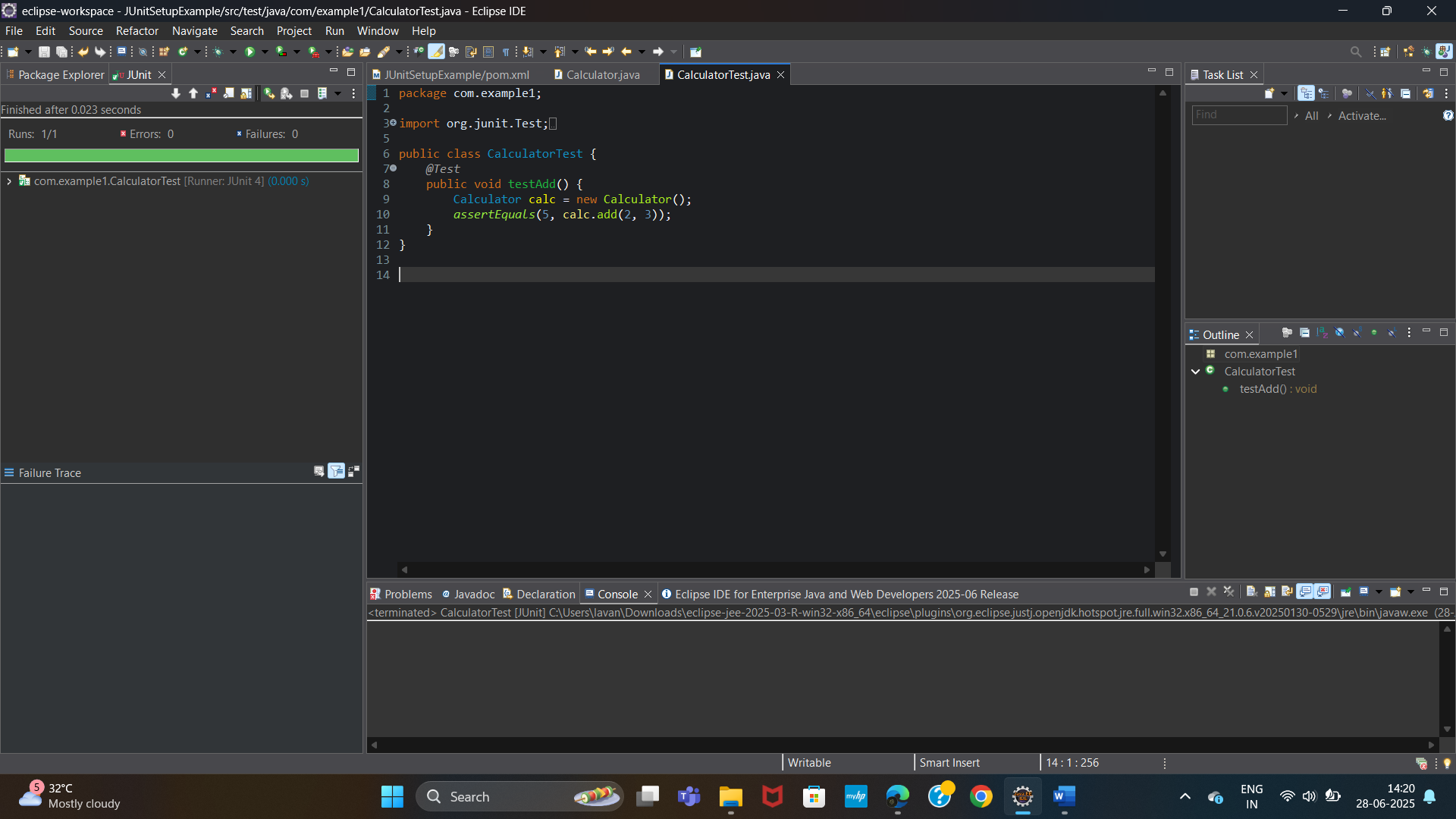
Calculator calc = new Calculator();

assertEquals(5, calc.add(2, 3));

}

}

**OUTPUT:**



**Exercise 3: Assertions in JUnit**

Scenario:

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

Steps:

1. Write tests using various JUnit assertions.

2. Solution Code:

public class AssertionsTest {

@Test

public void testAssertions() {

// Assert equals

assertEquals(5, 2 + 3);

// Assert true

assertTrue(5 > 3);

// Assert false

assertFalse(5 < 3);

// Assert null

assertNull(null);

// Assert not null

assertNotNull(new Object());

}

}

**CODE:**

**AssertionsTest.java**

package com.example1;

import org.junit.Test;

import static org.junit.Assert.\*;

public class AssertionsTest {

@Test

public void testAssertions() {

// Assert equals

assertEquals(5, 2 + 3);

// Assert true

assertTrue(5 > 3);

// Assert false

assertFalse(5 < 3);

// Assert null

assertNull(null);

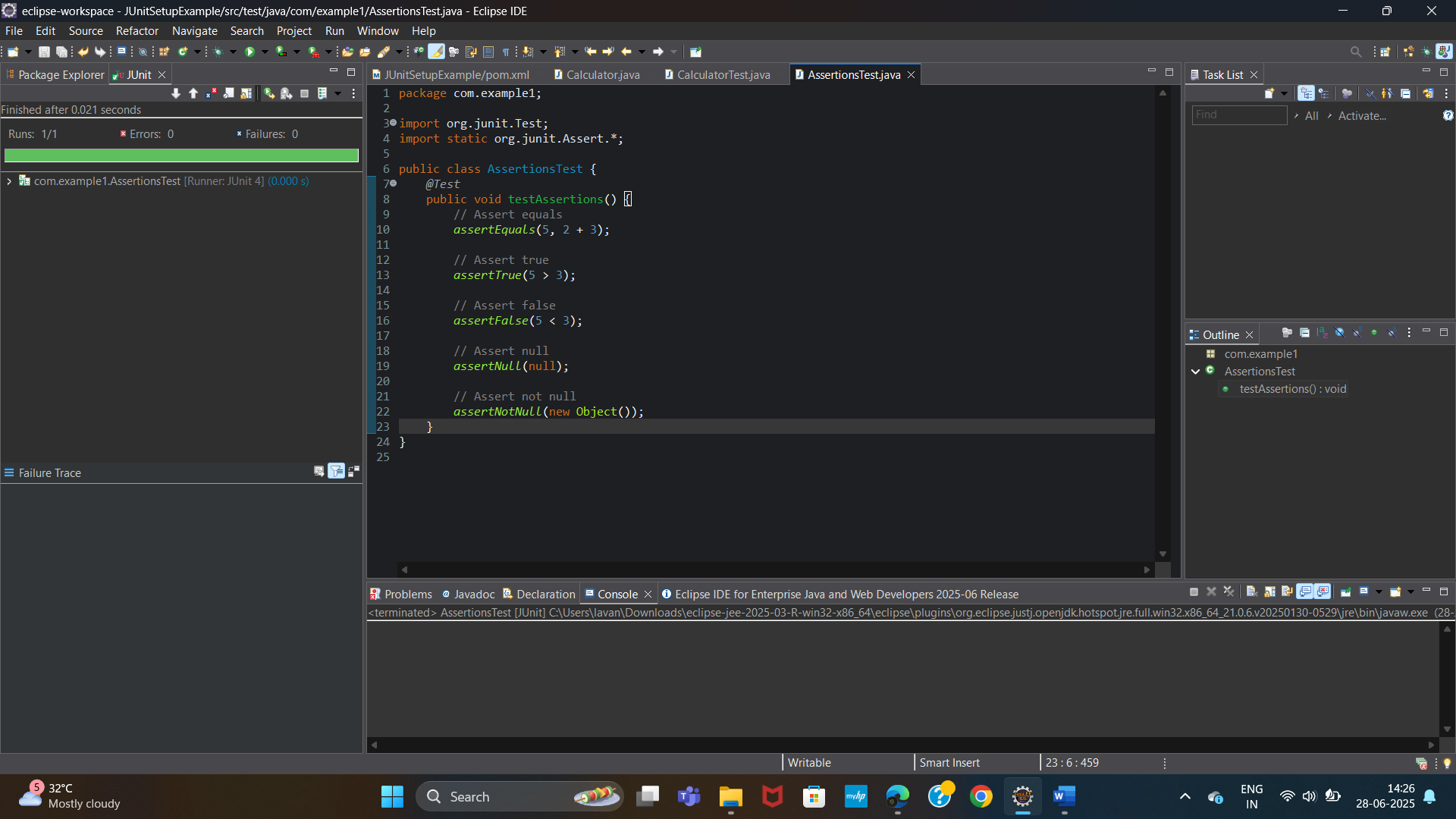
// Assert not null

assertNotNull(new Object());

}

}

**OUTPUT:**



**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.

CODE:

**Calculator.java**

package com.example1;

public class Calculator {

public int add(int a, int b) {

return a + b;

}

public int subtract(int a, int b) {

return a - b;

}

public int multiply(int a, int b) {

return a \* b;

}

public int divide(int a, int b) {

if (b == 0) {

throw new IllegalArgumentException("Cannot divide by zero");

}

return a / b;

}

}

**CalculatorTest.java**

package com.example1;

import org.junit.Before;

import org.junit.After;

import org.junit.Test;

import static org.junit.Assert.\*;

public class CalculatorTest {

private Calculator calc;

// Setup: runs before each test

*@Before*

public void setUp() {

calc = new Calculator();

System.***out***.println("Setup complete.");

}

// Teardown: runs after each test

*@After*

public void tearDown() {

calc = null;

System.***out***.println("Teardown complete.");

}

*@Test*

public void testAdd() {

// Arrange - done in setUp()

// Act

int result = calc.add(2, 3);

// Assert

*assertEquals*(5, result);

}

*@Test*

public void testSubtract() {

int result = calc.subtract(10, 5);

*assertEquals*(5, result);

}

*@Test*

public void testMultiply() {

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

*assertEquals*(20, result);

}

*@Test*

public void testDivide() {

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

*assertEquals*(5, result);

}

*@Test*(expected = IllegalArgumentException.class)

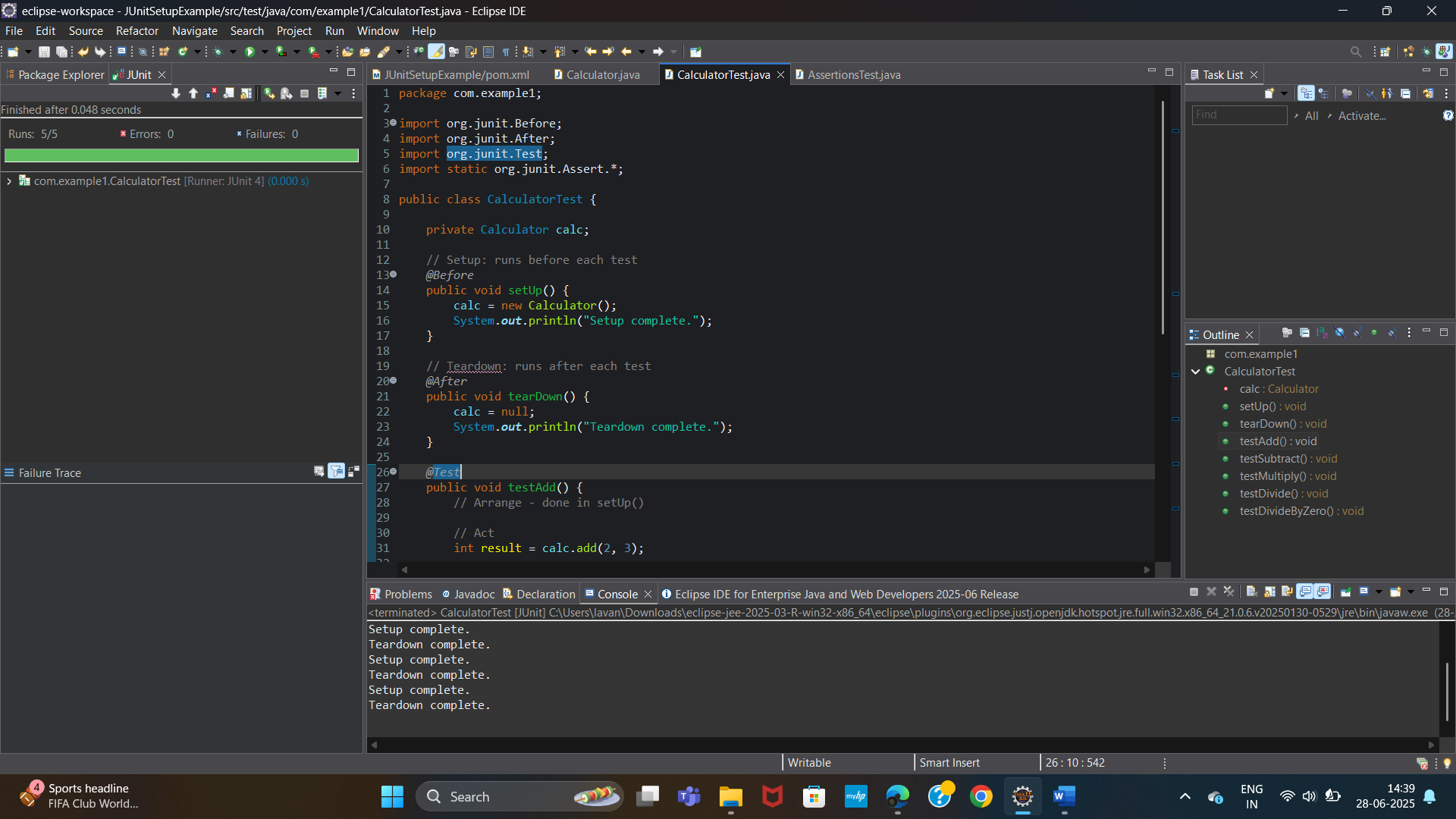
public void testDivideByZero() {

calc.divide(10, 0);

}

}

**OUTPUT:**



**MOCKITO 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.

Solution Code:

import static org.mockito.Mockito.\*;

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);

}

}

**CODE:**

**ExternalApi.java**

package com.example1;

public interface ExternalApi {

String getData();

}

**MyService.java**

package com.example1;

public class MyService {

private ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public String fetchData() {

return api.getData(); // Would call real API if not mocked

}

}

**MyServiceTest.java**

package com.example1;

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

import static org.mockito.Mockito.\*;

import org.junit.Test;

public class MyServiceTest {

*@Test*

public void testExternalApi() {

// Step 1: Create a mock of ExternalApi

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

// Step 2: Stub the getData() method

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

// Step 3: Inject mock into MyService

MyService service = new MyService(mockApi);

// Step 4: Call the method and assert

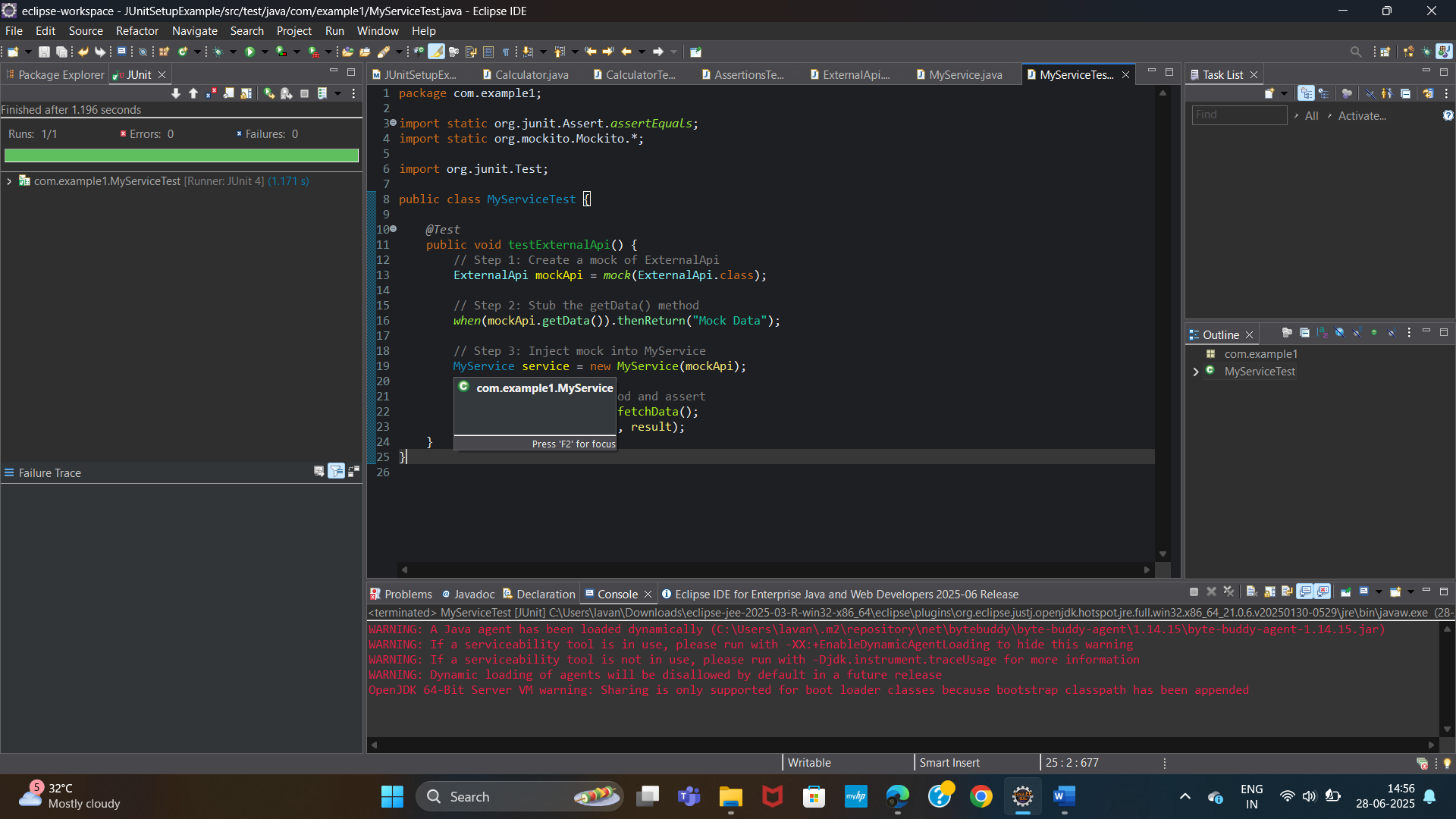
String result = service.fetchData();

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

}

}

**OUTPUT:**



**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.

Solution Code:

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

import org.mockito.Mockito;

public class MyServiceTest {

@Test

public void testVerifyInteraction() {

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

MyService service = new MyService(mockApi);

service.fetchData();

verify(mockApi).getData();

}

**CODE:**

**ExternalApi.java**

package com.example1;

public interface ExternalApi {

String getData();

}

**MyService.java**

package com.example1;

public class MyService {

private ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public String fetchData() {

return api.getData(); // Would call real API if not mocked

}

}

}

**MyServiceTest.java**

package com.example1;

import org.junit.Test;

import static org.mockito.Mockito.\*;

public class MyServiceTest {

*@Test*

public void testVerifyInteraction() {

// Step 1: Create a mock

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

// Step 2: Stub if needed (optional here)

*when*(mockApi.getData()).thenReturn("Mocked Response");

// Step 3: Use service

MyService service = new MyService(mockApi);

service.fetchData();

// Step 4: Verify the method was called

*verify*(mockApi).getData();

}

}

**OUTPUT:**

