**JUnit Basic Testing**

**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

3. Create a new test class in your project

**Code:**

**App.java**

package com.example;

public class App {

public int add(int a, int b) {

return a + b;

}

}

**AppTest.java**

package com.example;

import org.junit.Test;

import static org.junit.Assert.\*;

public class AppTest {

@Test

public void testAdd() {

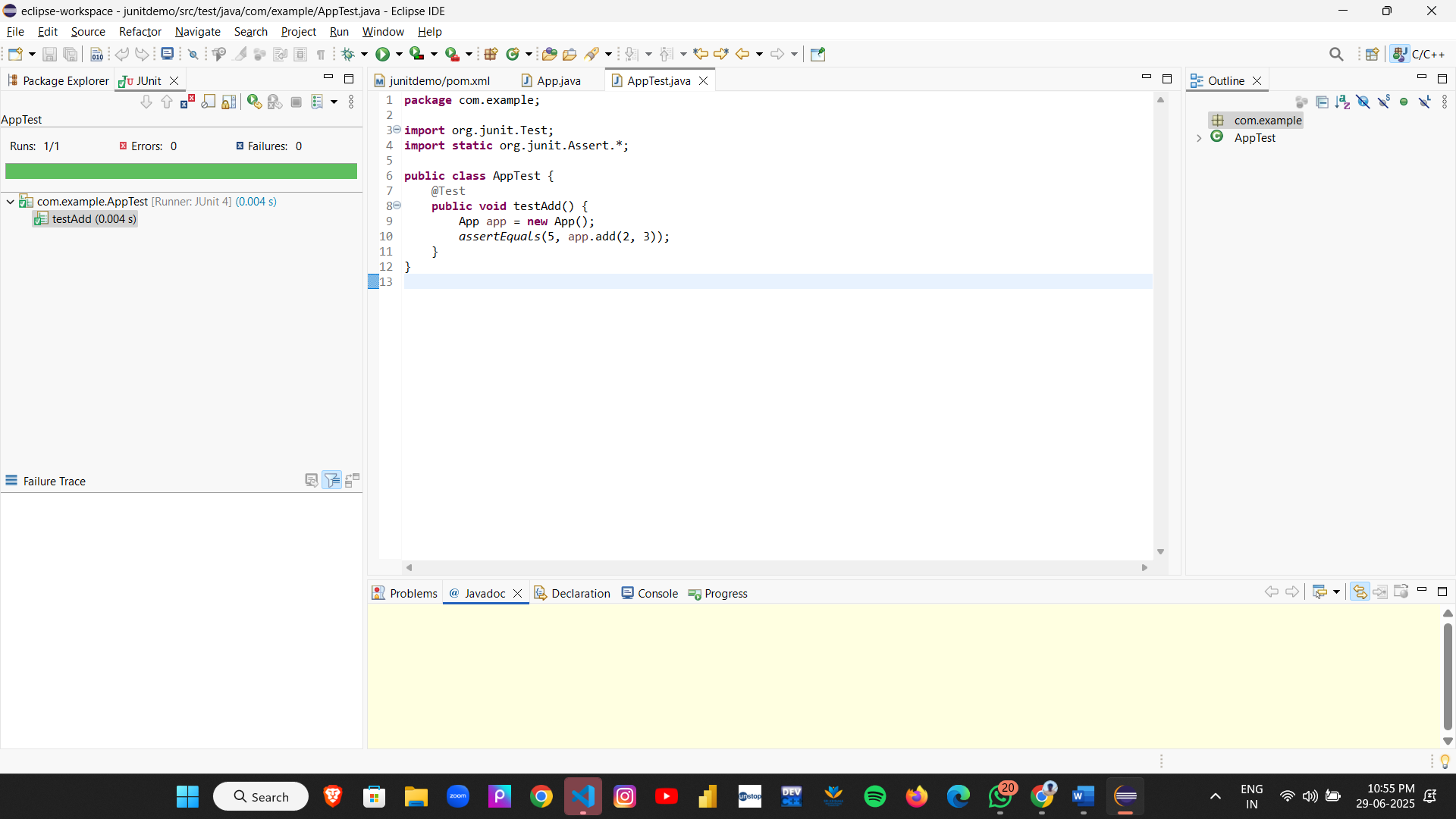
App app = new App();

*assertEquals*(5, app.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.

**Code**:

**AssertionsTest.java**

package com.example;

import org.junit.Test;

import static org.junit.Assert.\*;

public class AssertionsTest {

@Test

public void testAssertions() {

assertEquals(5, 2 + 3);

System.out.println("assertEquals passed");

assertTrue(5 > 3);

System.out.println("assertTrue passed");

assertFalse(5 < 3);

System.out.println("assertFalse passed");

assertNull(null);

System.out.println("assertNull passed");

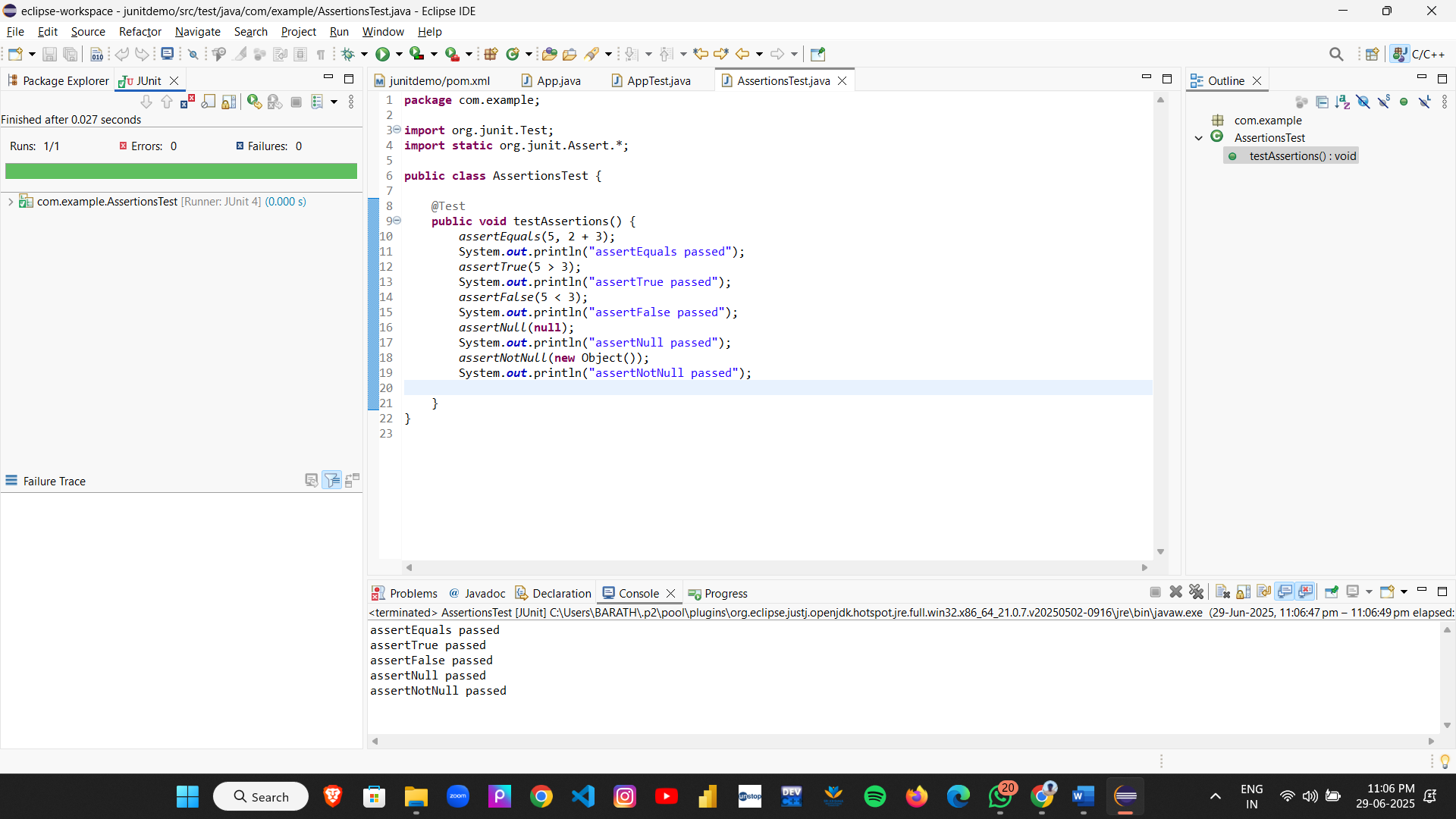
assertNotNull(new Object());

System.out.println("assertNotNull passed");

}

}

**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.example;

public class Calculator {

public int add(int a, int b) {

System.*out*.println("Calculator: Performing addition.");

return a + b;

}

public int subtract(int a, int b) {

System.*out*.println("Calculator: Performing subtraction.");

return a - b;

}

}

**CalculatorTest.java**

package com.example;

import org.junit.After;

import org.junit.Before;

import org.junit.Test;

import static org.junit.Assert.\*;

public class CalculatorTest {

private Calculator calculator;

@Before

public void setUp() {

System.*out*.println("[SETUP] Initializing calculator...");

calculator = new Calculator();

}

@After

public void tearDown() {

System.*out*.println("[TEARDOWN] Cleaning up after test...\n");

}

@Test

public void testAddition() {

System.*out*.println("[TEST] testAddition started");

int a = 10;

int b = 5;

System.*out*.println("Arrange: a = " + a + ", b = " + b);

int result = calculator.add(a, b);

System.*out*.println("Act: result = " + result);

*assertEquals*(15, result);

System.*out*.println("Assert: Passed - result == 15");

}

@Test

public void testSubtraction() {

System.*out*.println("[TEST] testSubtraction started");

int a = 10;

int b = 3;

System.*out*.println("Arrange: a = " + a + ", b = " + b);

int result = calculator.subtract(a, b);

System.*out*.println("Act: result = " + result);

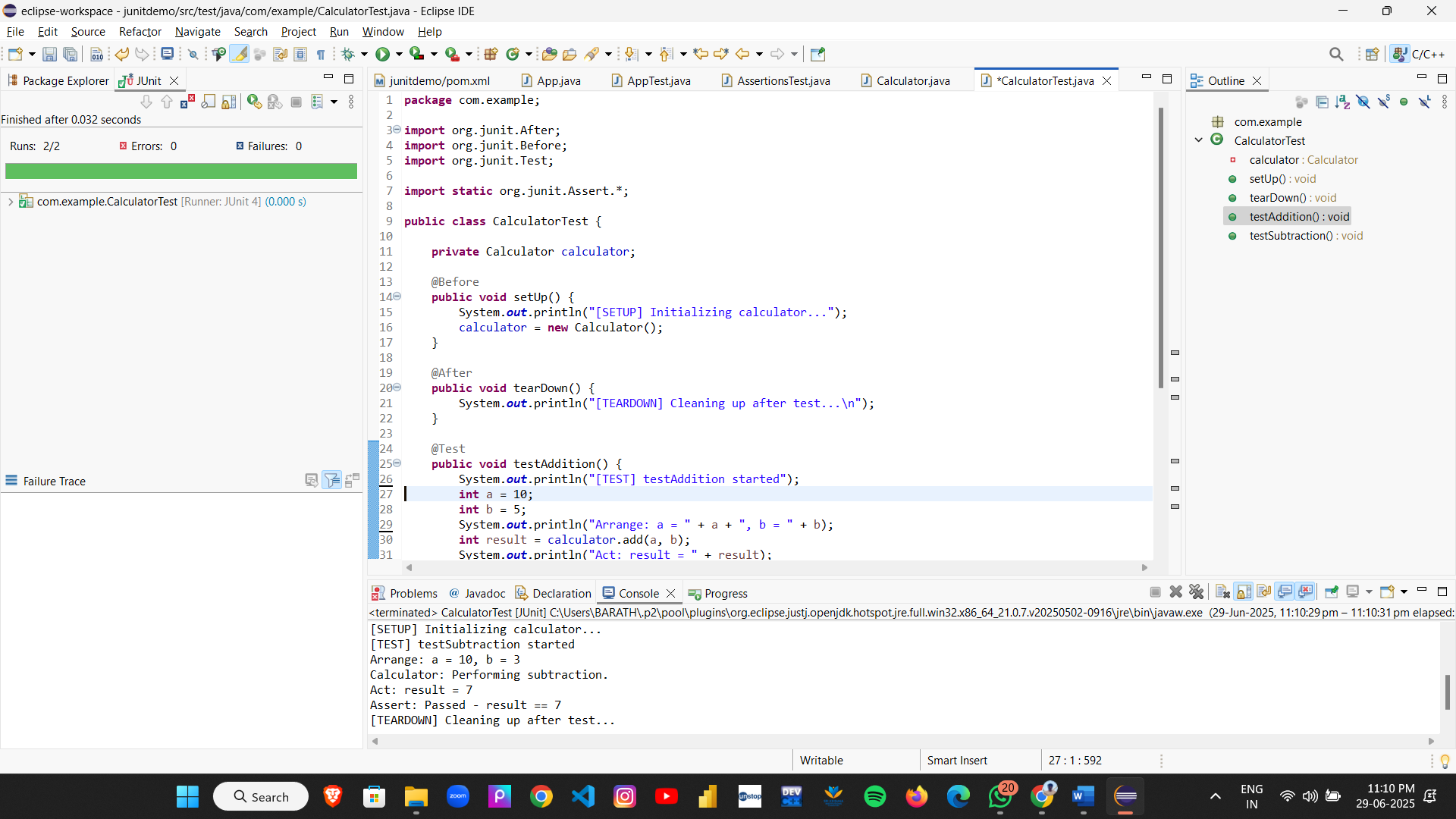
*assertEquals*(7, result);

System.*out*.println("Assert: Passed - result == 7");

}

}

**Output:**



**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.

**Code:**

**ExternalApi.java**

package com.example;

public interface ExternalApi {

String getData();

}

**MyService.java**

package com.example;

public class MyService {

private final ExternalApi api;

public MyService(ExternalApi api) {

System.*out*.println("[MyService] Constructor called with ExternalApi dependency.");

this.api = api;

}

public String fetchData() {

System.*out*.println("[MyService] Calling ExternalApi.getData()");

return api.getData();

}

}

**MyServiceTest.java**

package com.example;

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

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

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

import org.junit.jupiter.api.Test;

public class MyServiceTest {

@Test

public void testExternalApi() {

System.*out*.println("[TEST] testExternalApi started.");

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

System.*out*.println("[MOCK] Created mock for ExternalApi.");

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

System.*out*.println("[STUB] Configured mockApi.getData() to return 'Mock Data'.");

MyService service = new MyService(mockApi);

System.*out*.println("[SERVICE] MyService instantiated with mocked API.");

String result = service.fetchData();

System.*out*.println("[ACTION] Called service.fetchData(). Result: " + result);

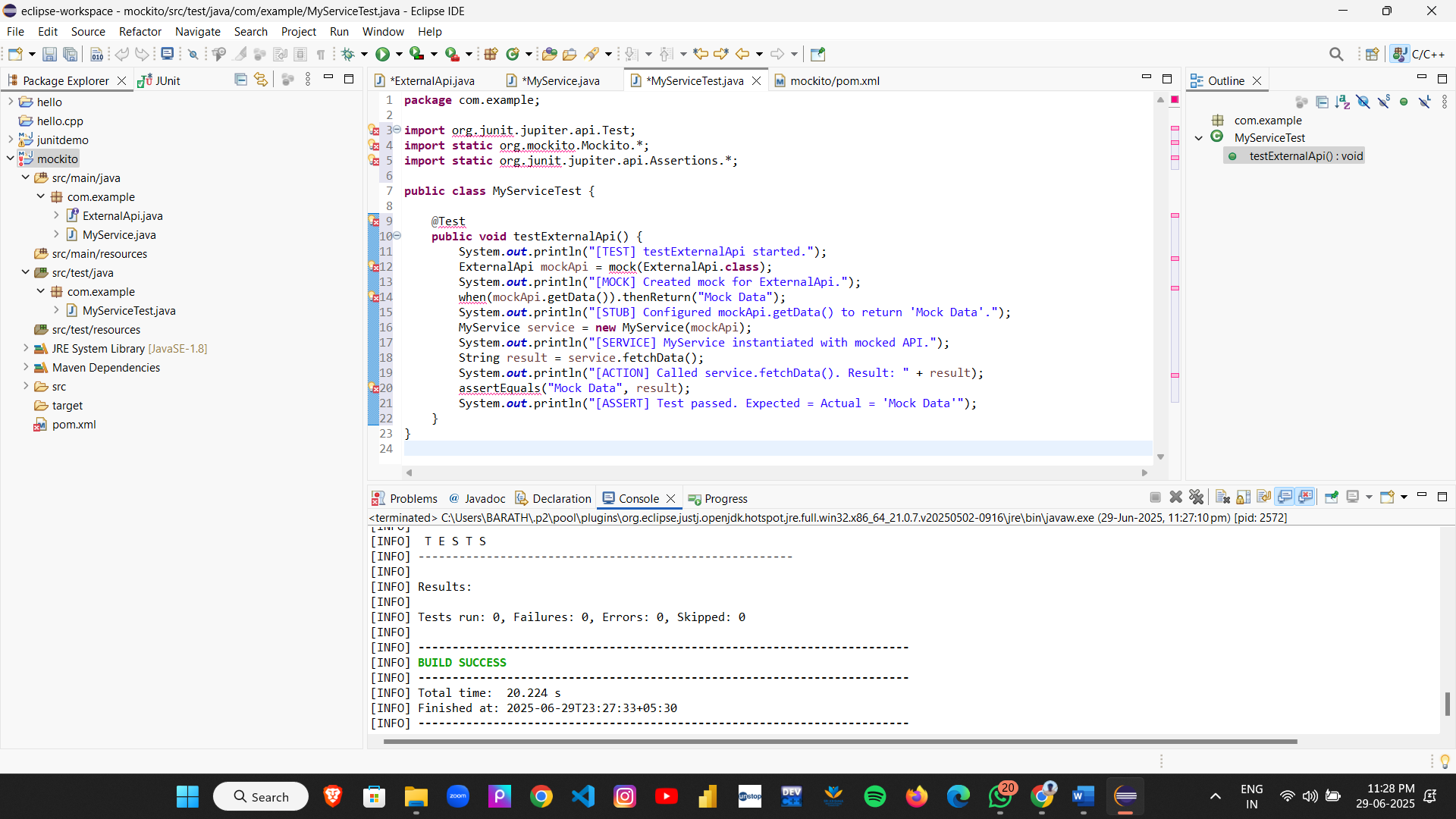
*assertEquals*("Mock Data", result);

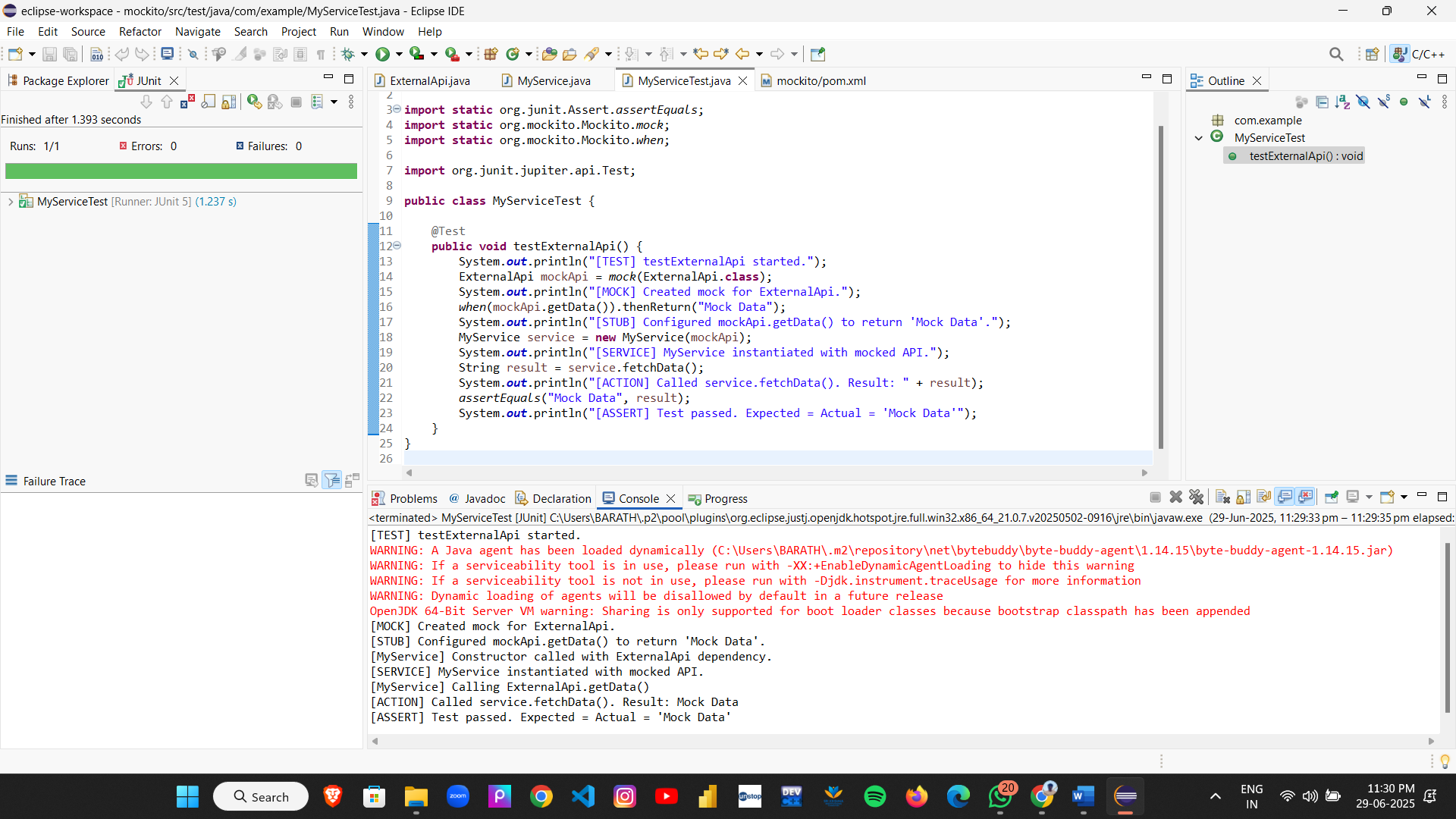
System.*out*.println("[ASSERT] Test passed. Expected = Actual = 'Mock Data'");

}

}

**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.

**Code**:

**ExternalApi.java**

package com.example;

public interface ExternalApi {

String getData();

}

**MyService.java**

package com.example;

public class MyService {

private final ExternalApi api;

public MyService(ExternalApi api) {

System.out.println("[MyService] Initialized with ExternalApi.");

this.api = api;

}

public String fetchData() {

System.out.println("[MyService] fetchData() is calling api.getData()");

return api.getData();

}

}

**MyServiceTest.java**

package com.example;

import org.junit.jupiter.api.Test;

import static org.mockito.Mockito.\*;

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

public class MyServiceTest {

@Test

public void testVerifyInteraction() {

System.out.println("[TEST] testVerifyInteraction started.");

ExternalApi mockApi = mock(ExternalApi.class);

System.out.println("[MOCK] Created mock ExternalApi.");

MyService service = new MyService(mockApi);

service.fetchData();

System.out.println("[ACTION] Called service.fetchData().");

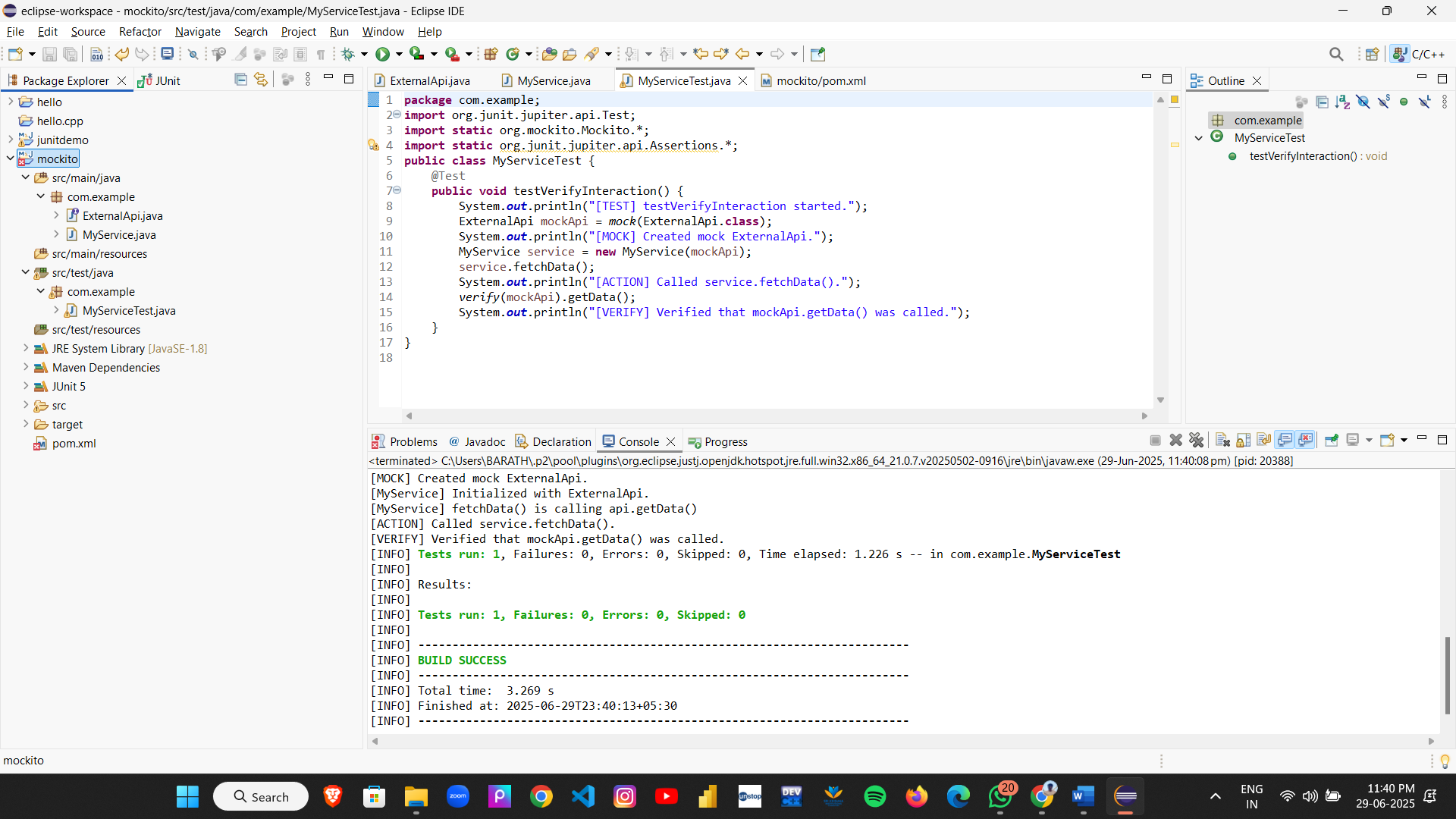
verify(mockApi).getData();

System.out.println("[VERIFY] Verified that mockApi.getData() was called.");

}

}

**Output:**



**Logging using SLF4J**

**Exercise 1: Logging Error Messages and Warning Levels**

Task: Write a Java application that demonstrates logging error messages and warning levels

using SLF4J.

1. Add SLF4J and Logback dependencies to your `pom.xml` file:

2. Create a Java class that uses SLF4J for logging:

**Code:**

**LoggingExample.java**

package com.example;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class LoggingExample {

private static final Logger logger = LoggerFactory.getLogger(LoggingExample.class);

public static void main(String[] args) {

logger.error("This is an error message");

logger.warn("This is a warning message");

logger.info("This is an info message");

logger.debug("This is a debug message");

}

}

**Output:**

