**WEEK-2-UNIT TESTING**

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

3. Create a new test class in your project.

**Solution:**

Main class:

**package** com.example;

**public** **class** Calculator {

**public** **int** add(**int** a, **int** b) {

**return** a + b;

}

}

Test class:

**package** com.example;

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

**import** org.junit.Test;

**public** **class** CalculatorTest {

@Test

**public** **void** testAdd() {

Calculator calc = **new** Calculator();

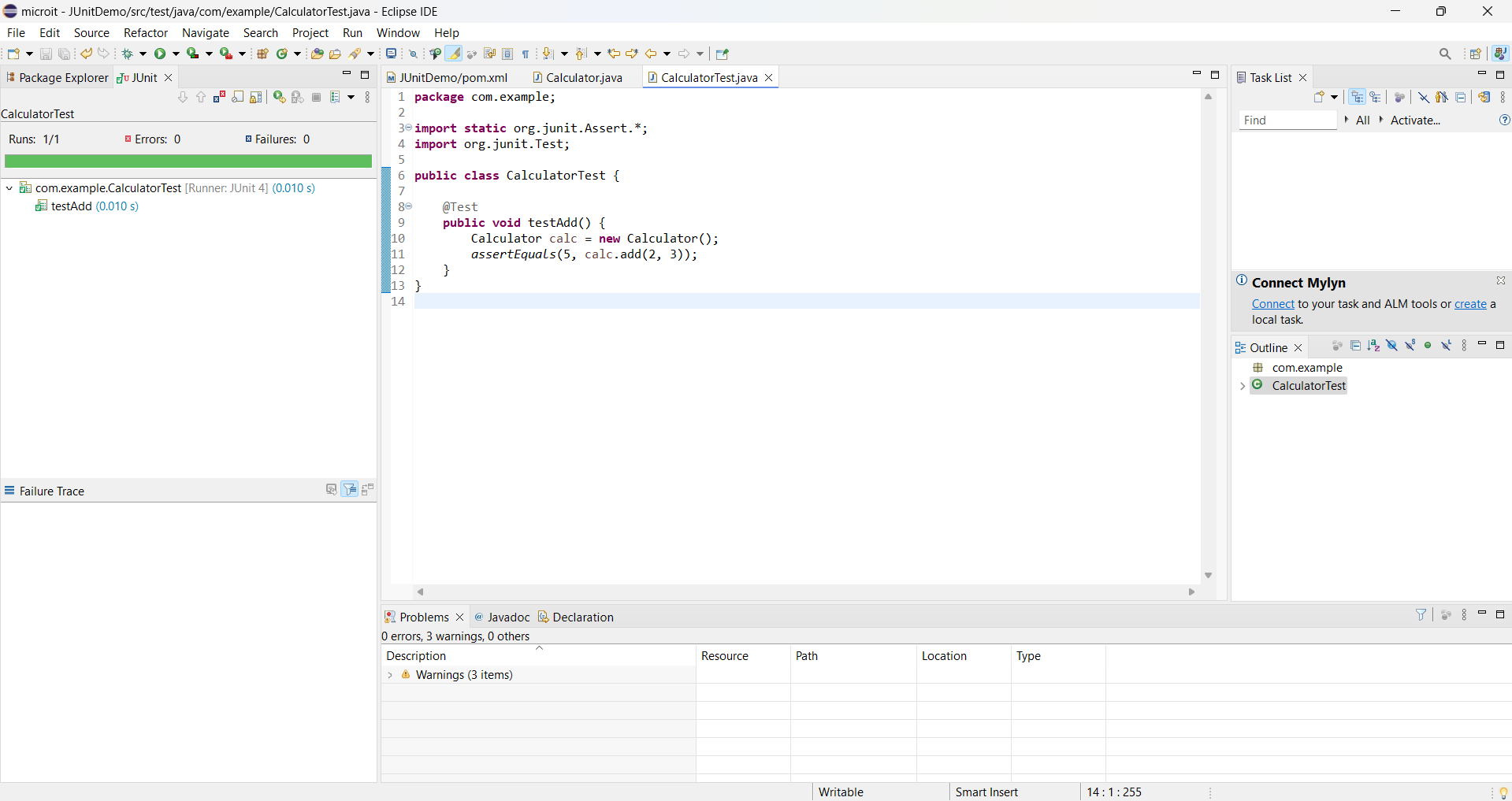
*assertEquals*(5, calc.add(2, 3));

}

}

It is a Maven Project.

**Output:**



The test has ran successfully.

**Exercise 2: Writing Basic JUnit Tests Scenario:**

You need to write basic JUnit tests for a simple Java class.

Steps:

1. Create a new Java class with some methods to test.

2. Write JUnit tests for these methods.

**Solution:**

Main class:

**package** com.example;

**public** **class** Calculator {

// Method to add two integers

**public** **int** add(**int** a, **int** b) {

**return** a + b;

}

// Method to subtract two integers

**public** **int** subtract(**int** a, **int** b) {

**return** a - b;

}

// Method to multiply two integers

**public** **int** multiply(**int** a, **int** b) {

**return** a \* b;

}

// Method to divide two integers

**public** **int** divide(**int** a, **int** b) {

**if** (b == 0) {

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

}

**return** a / b;

}

}

Test class:

**package** com.example;

**import** org.junit.Test;

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

**public** **class** CalculatorTest {

Calculator calc = **new** Calculator();

@Test

**public** **void** testAdd() {

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

}

@Test

**public** **void** testSubtract() {

*assertEquals*(1, calc.subtract(5, 4));

}

@Test

**public** **void** testMultiply() {

*assertEquals*(12, calc.multiply(3, 4));

}

@Test

**public** **void** testDivide() {

*assertEquals*(2, calc.divide(10, 5));

}

@Test(expected = IllegalArgumentException.**class**)

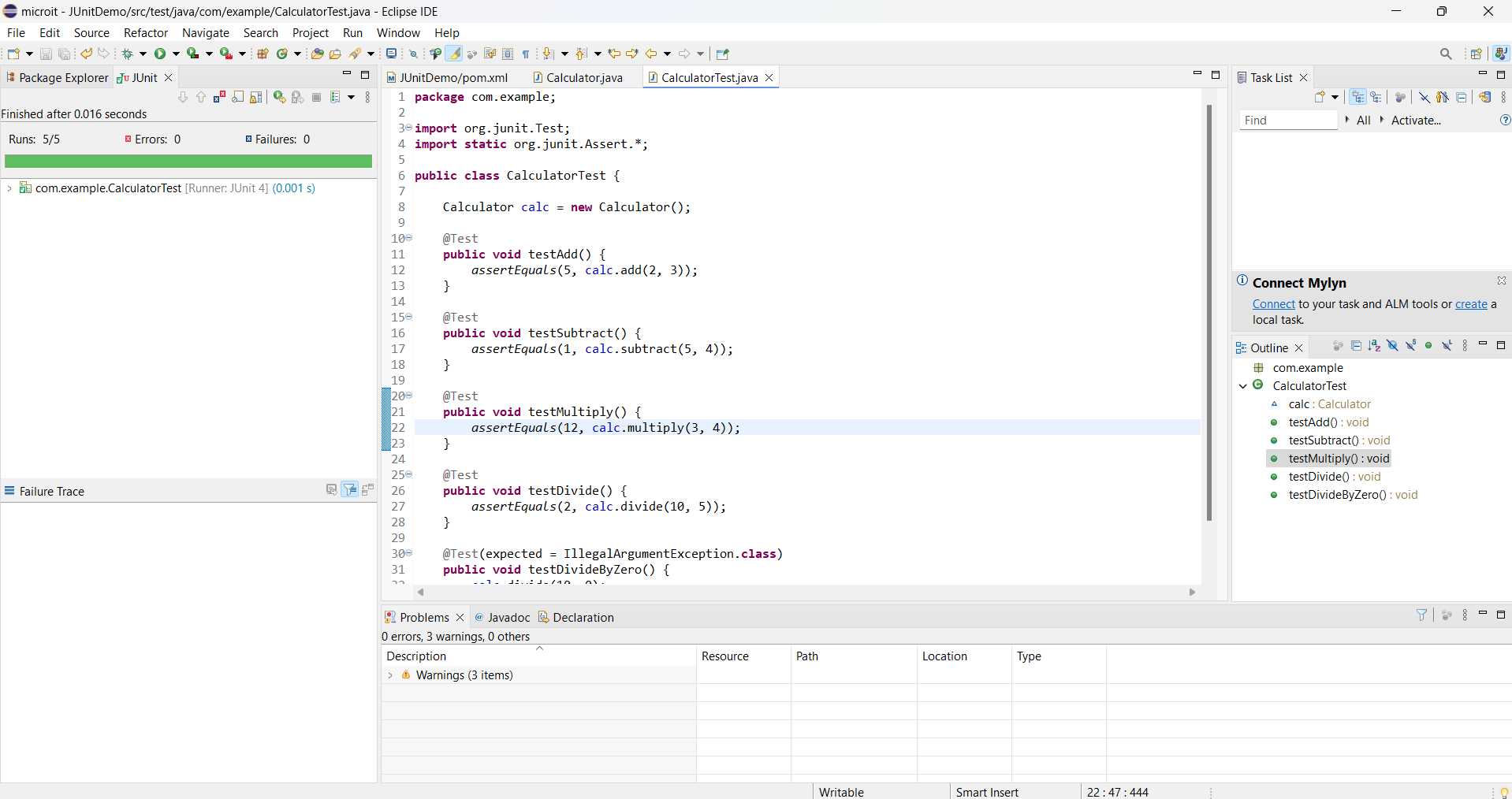
**public** **void** testDivideByZero() {

calc.divide(10, 0);

}

}

**Output:**



The test has run successfully.

**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);

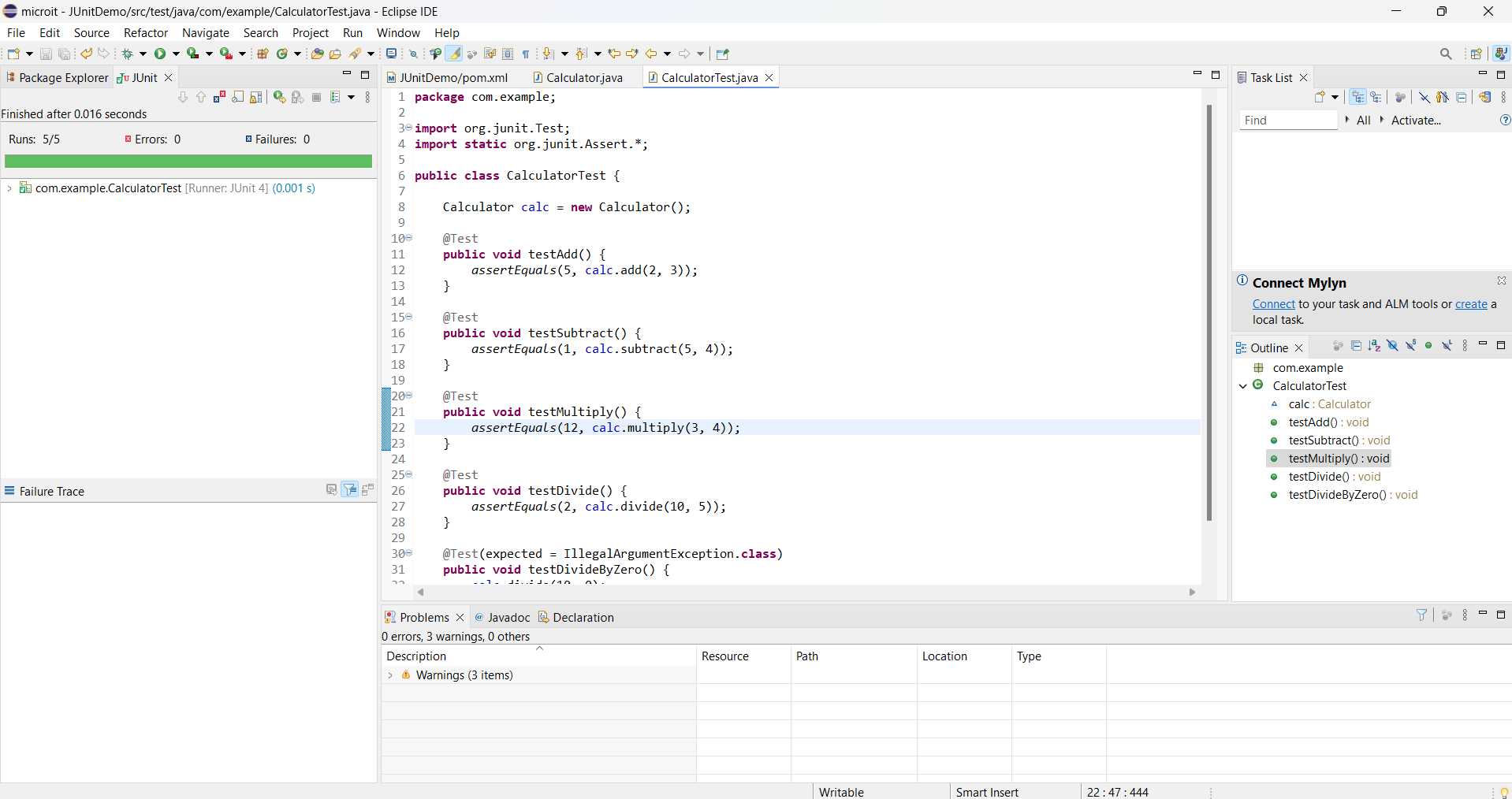
// Assert not null

assertNotNull(new Object());

}

}

**Solution:**



**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

**Solution:**

Main class:

**package** com.example;

**public** **class** Calculator {

**public** **int** add(**int** a, **int** b) {

**return** a + b;

}

**public** **int** subtract(**int** a, **int** b) {

**return** a - b;

}

}

Test class:

**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 calc;

// Setup method: runs before each test

@Before

**public** **void** setUp() {

calc = **new** Calculator(); // Arrange

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

}

// Teardown method: runs after each test

@After

**public** **void** tearDown() {

calc = **null**; // Cleanup

System.***out***.println("Cleaning up test...\n");

}

@Test

**public** **void** testAdd() {

// Act

**int** result = calc.add(5, 7);

// Assert

*assertEquals*(12, result);

}

@Test

**public** **void** testSubtract() {

// Act

**int** result = calc.subtract(10, 4);

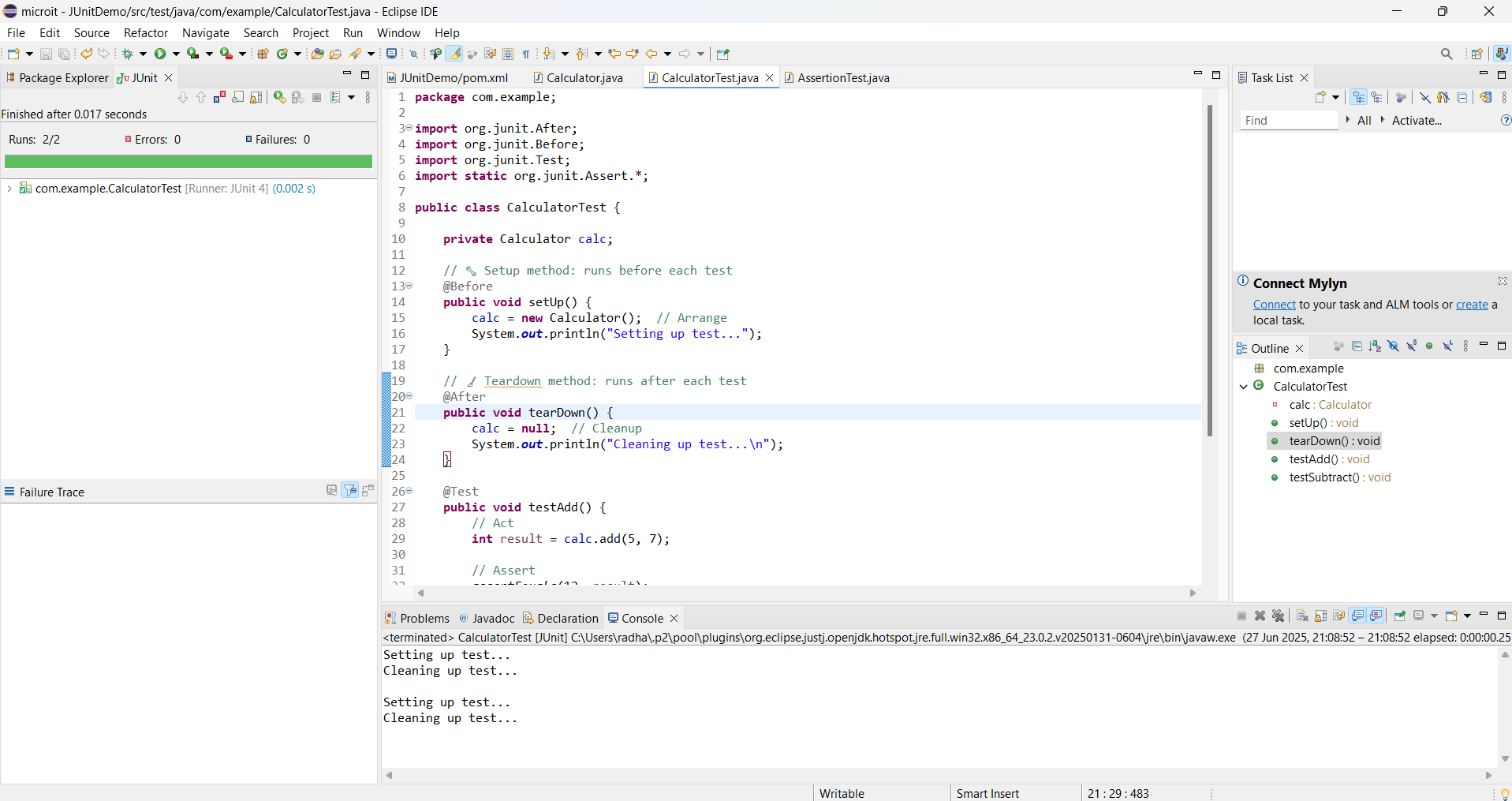
// Assert

*assertEquals*(6, result);

}

}

**Output:**



**Filename: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);

}

}

**Solution:**

package com.example.mock;

import static org.mockito.Mockito.\*;

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

import org.junit.jupiter.api.Test;

import org.mockito.Mockito;

interface ExternalApi {

String getData();

}

class MyService {

private ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public String fetchData() {

return api.getData();

}

}

public class MyServiceTest {

@Test

public void testExternalApi() {

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

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

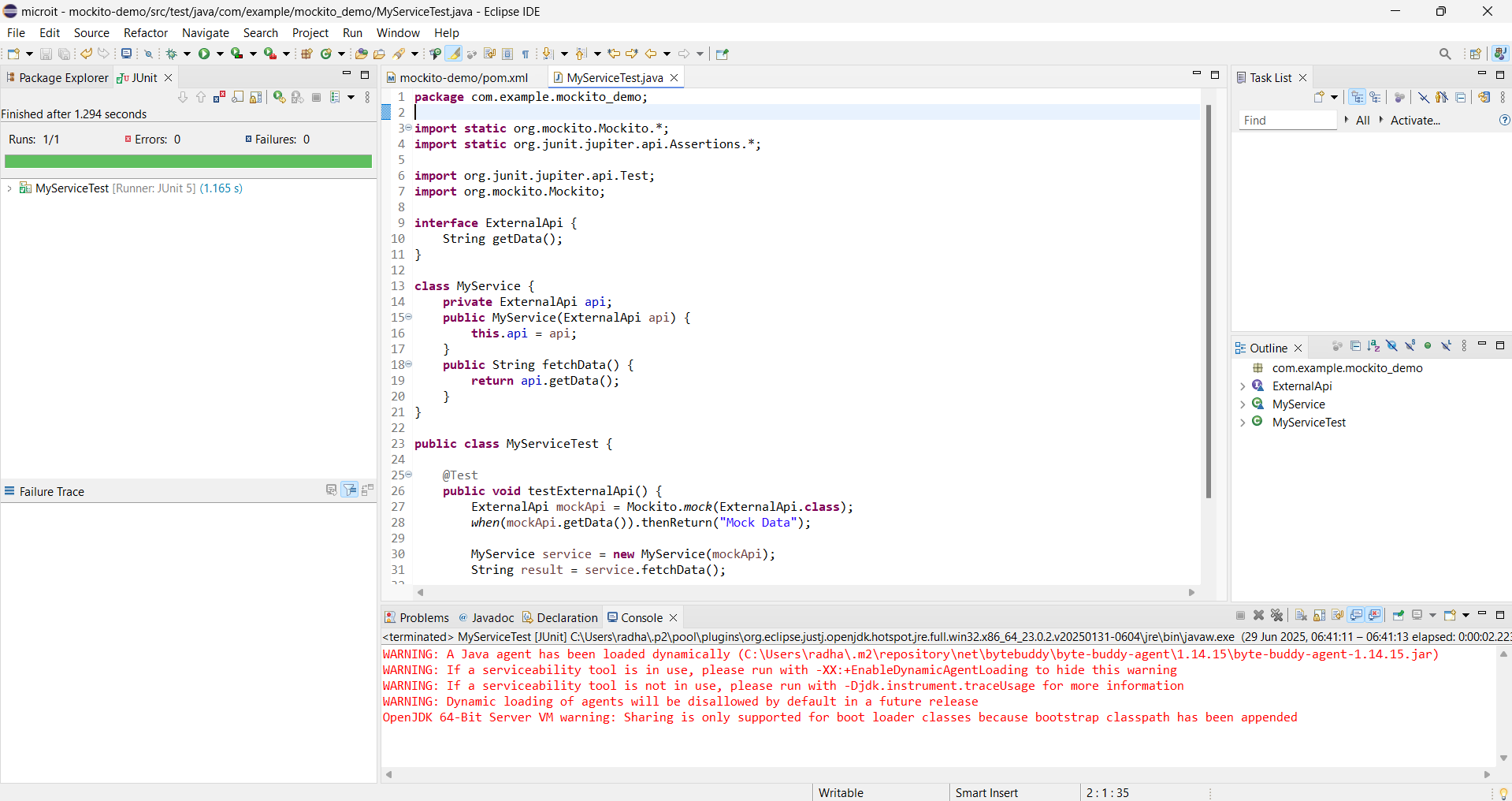
MyService service = new MyService(mockApi);

String result = service.fetchData();

assertEquals("Mock Data", result);

}

}**Output:**



**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();

}

}

**Solution:**

**package** com.example.mockito\_demo;

**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 to be mocked

**interface** ExternalApi {

String getData();

}

**class** MyService {

**private** ExternalApi api;

**public** MyService(ExternalApi api) {

**this**.api = api;

}

**public** String fetchData() {

**return** api.getData(); // method that we want to verify was called

}

}

// Unit test class

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

@Test

**public** **void** testVerifyInteraction() {

// Create mock

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

// Create service with the mock

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

service.fetchData();

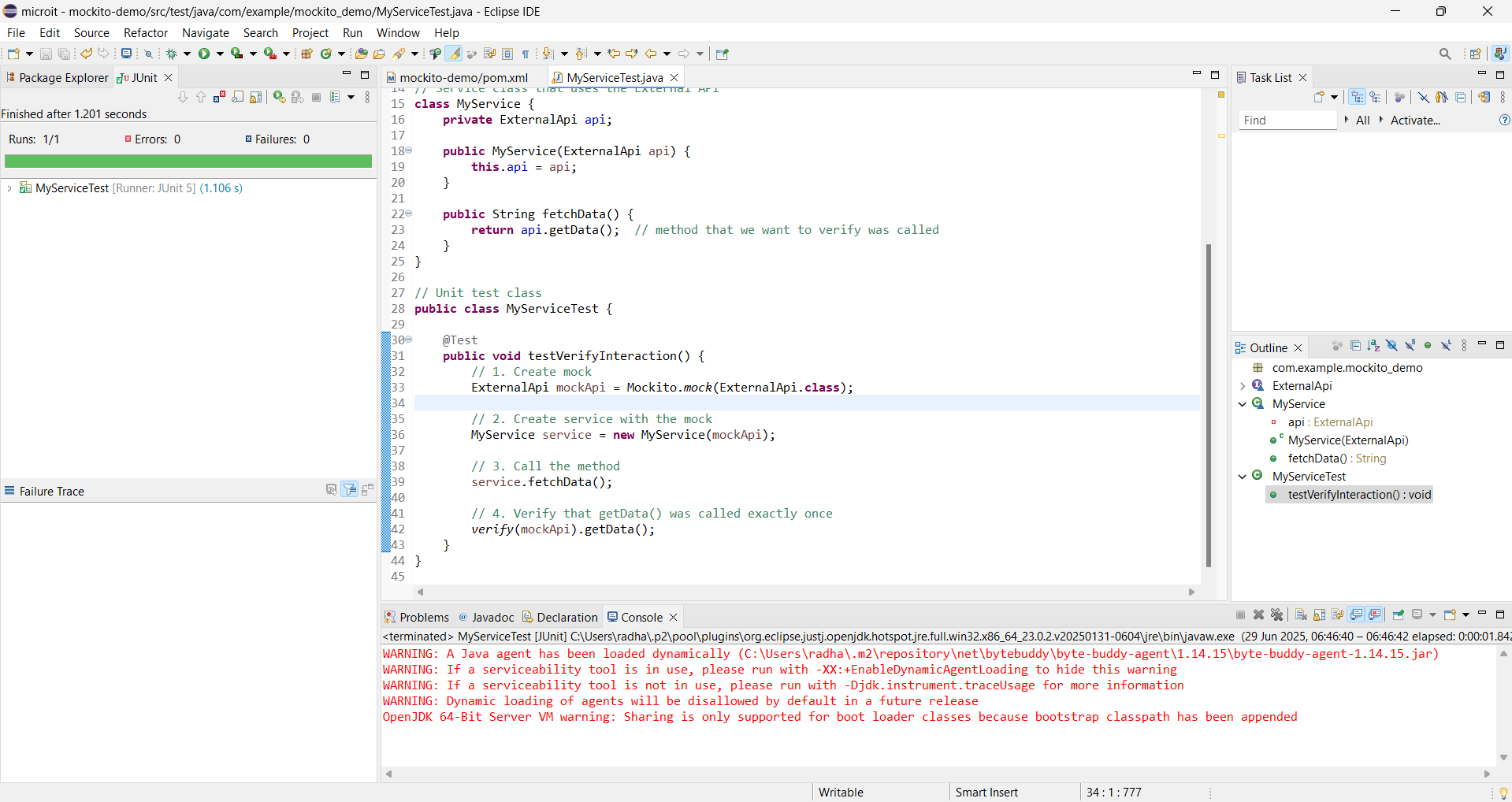
// Verify that getData() was called exactly once

*verify*(mockApi).getData();

}

}

**Output:**



**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.

**Solution:**

**package** com.example.mockito\_demo;

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

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

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

**import** org.junit.jupiter.api.Test;

**import** org.mockito.Mockito;

// External API interface

**interface** ExternalApi {

String getDataById(String id);

}

// Service class using the API

**class** MyService {

**private** ExternalApi api;

**public** MyService(ExternalApi api) {

**this**.api = api;

}

**public** String fetchDataById(String id) {

**return** api.getDataById(id);

}

}

// Test class demonstrating argument matching

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

@Test

**public** **void** testArgumentMatching() {

// 1. Create mock

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

// 2. Stub method with any string

*when*(mockApi.getDataById(*anyString*())).thenReturn("Mock Response");

// 3. Call method with specific argument

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

String result = service.fetchDataById("12345");

// 4. Assert the return value

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

// 5. Verify method was called with the exact argument

*verify*(mockApi).getDataById(*eq*("12345"));

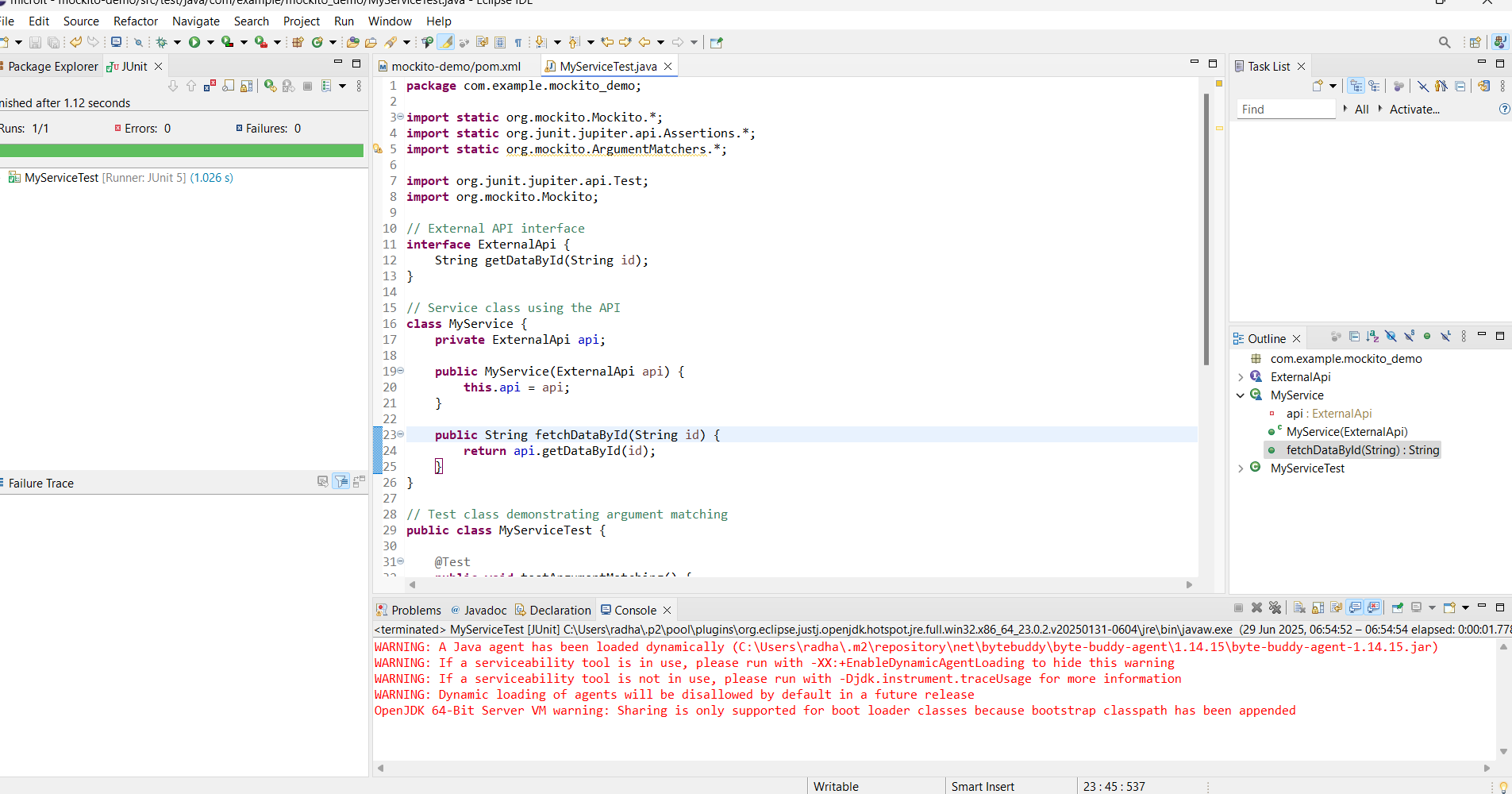
// OR: verify it was called with any string

// verify(mockApi).getDataById(anyString());

}

}

**Output:**



**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.

**Solution:**

**package** com.example.mockito\_demo;

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

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

**import** org.junit.jupiter.api.Test;

**import** org.mockito.Mockito;

// Step 1: Define an interface with a void method

**interface** Logger {

**void** log(String message);

}

// Step 2: A service class that uses the Logger

**class** MyService {

**private** Logger logger;

**public** MyService(Logger logger) {

**this**.logger = logger;

}

**public** **void** process() {

// do some work...

logger.log("Processing done");

}

}

// Step 3: Test the void method interaction

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

@Test

**public** **void** testVoidMethod() {

// 1. Create mock object

Logger mockLogger = Mockito.*mock*(Logger.**class**);

// 2. Optionally stub the void method (not required unless exception needed)

*doNothing*().when(mockLogger).log(*anyString*());

// 3. Create service with the mock

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

service.process();

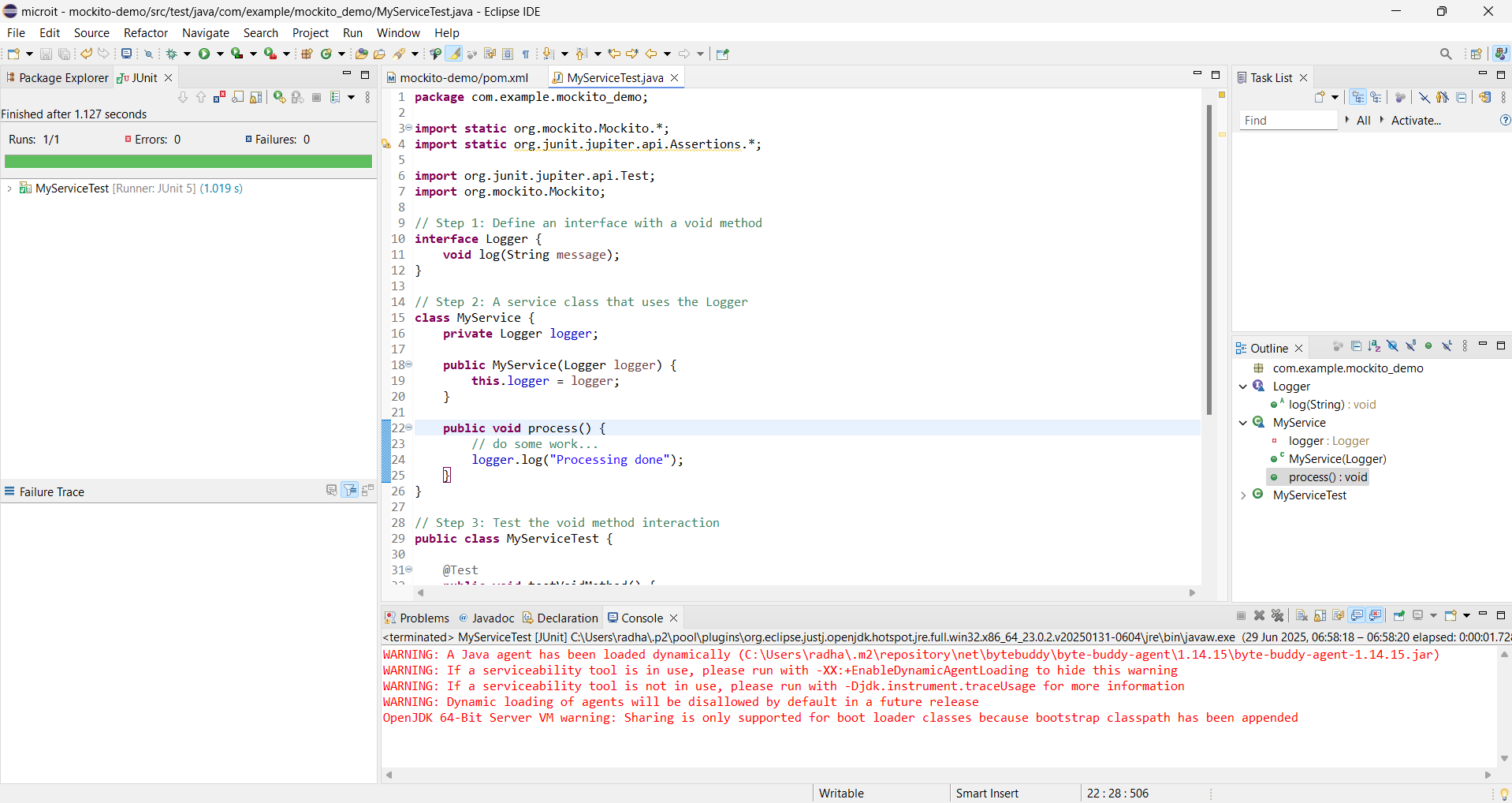
// 4. Verify the void method was called with specific argument

*verify*(mockLogger).log("Processing done");

}

}

**Output:**



**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.

**Solution:**

**package** com.example.mockito\_demo;

**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 with a method returning different values

**interface** ExternalApi {

String getStatus();

}

// Service that calls the API method multiple times

**class** MyService {

**private** ExternalApi api;

**public** MyService(ExternalApi api) {

**this**.api = api;

}

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

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

api.getStatus(),

api.getStatus(),

api.getStatus()

};

}

}

// Test class demonstrating multiple returns

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

@Test

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

// 1. Create mock object

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

// 2. Stub method with multiple return values

*when*(mockApi.getStatus())

.thenReturn("Loading")

.thenReturn("Processing")

.thenReturn("Completed");

// 3. Use service with mock

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

String[] result = service.checkStatusProgression();

// 4. Assert the sequence of return values

*assertArrayEquals*(**new** String[] {

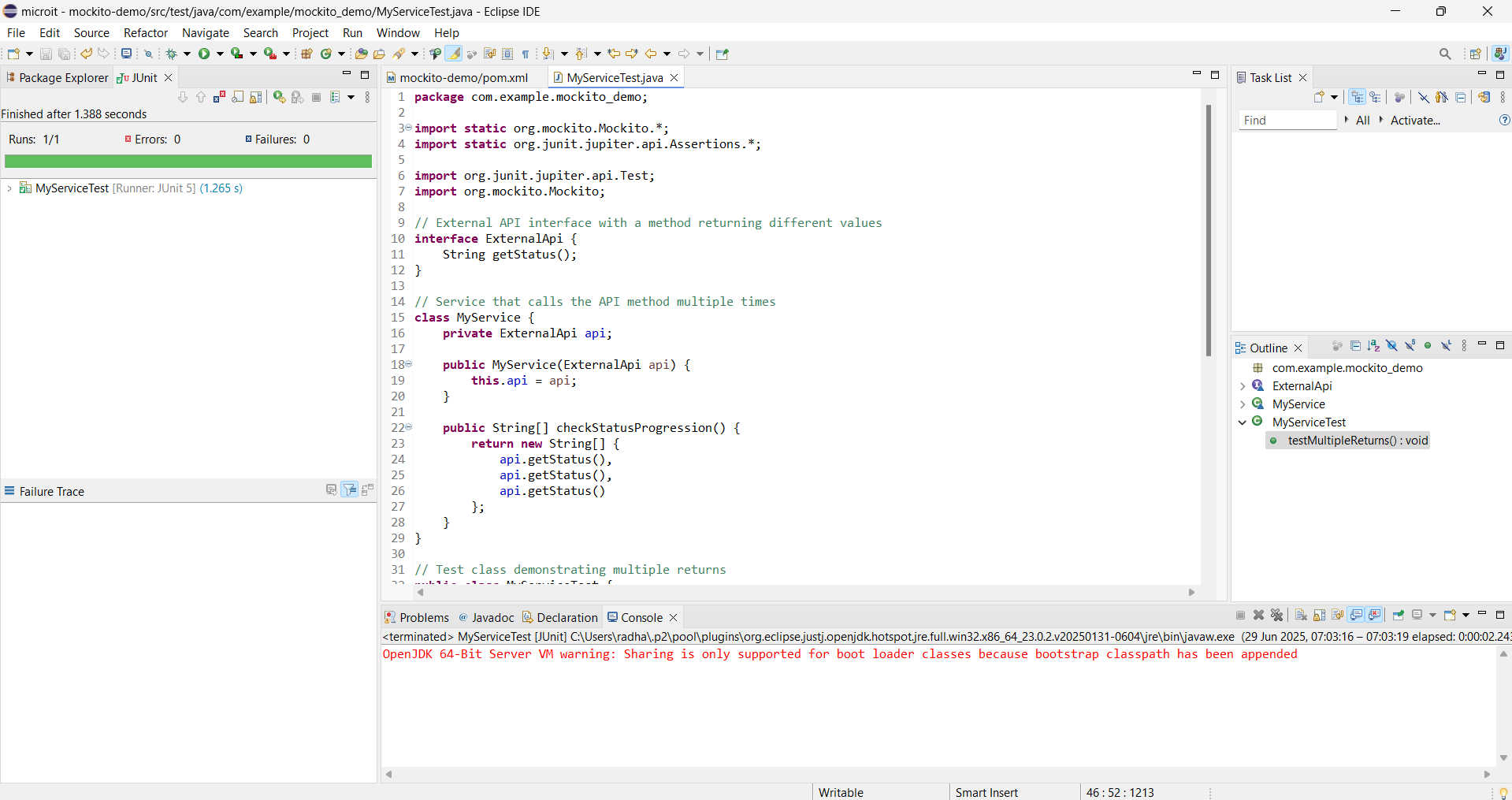
"Loading", "Processing", "Completed"

}, result);

}

}

**Output:**

****

**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.

**Solution:**

**package** com.example.mockito\_demo;

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

**import** org.junit.jupiter.api.Test;

**import** org.mockito.InOrder;

**import** org.mockito.Mockito;

// Step 1: External API interface

**interface** ExternalApi {

**void** initialize();

String fetchData();

}

// Step 2: Service that uses ExternalApi

**class** MyService {

**private** ExternalApi api;

**public** MyService(ExternalApi api) {

**this**.api = api;

}

**public** **void** process() {

api.initialize(); // should be called first

api.fetchData(); // then this

}

}

// Step 3: Test class verifying order

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

@Test

**public** **void** testInteractionOrder() {

// 1. Create a mock object

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

// 2. Call the methods in a specific order

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

service.process();

// 3. Verify the interaction order

InOrder inOrder = *inOrder*(mockApi);

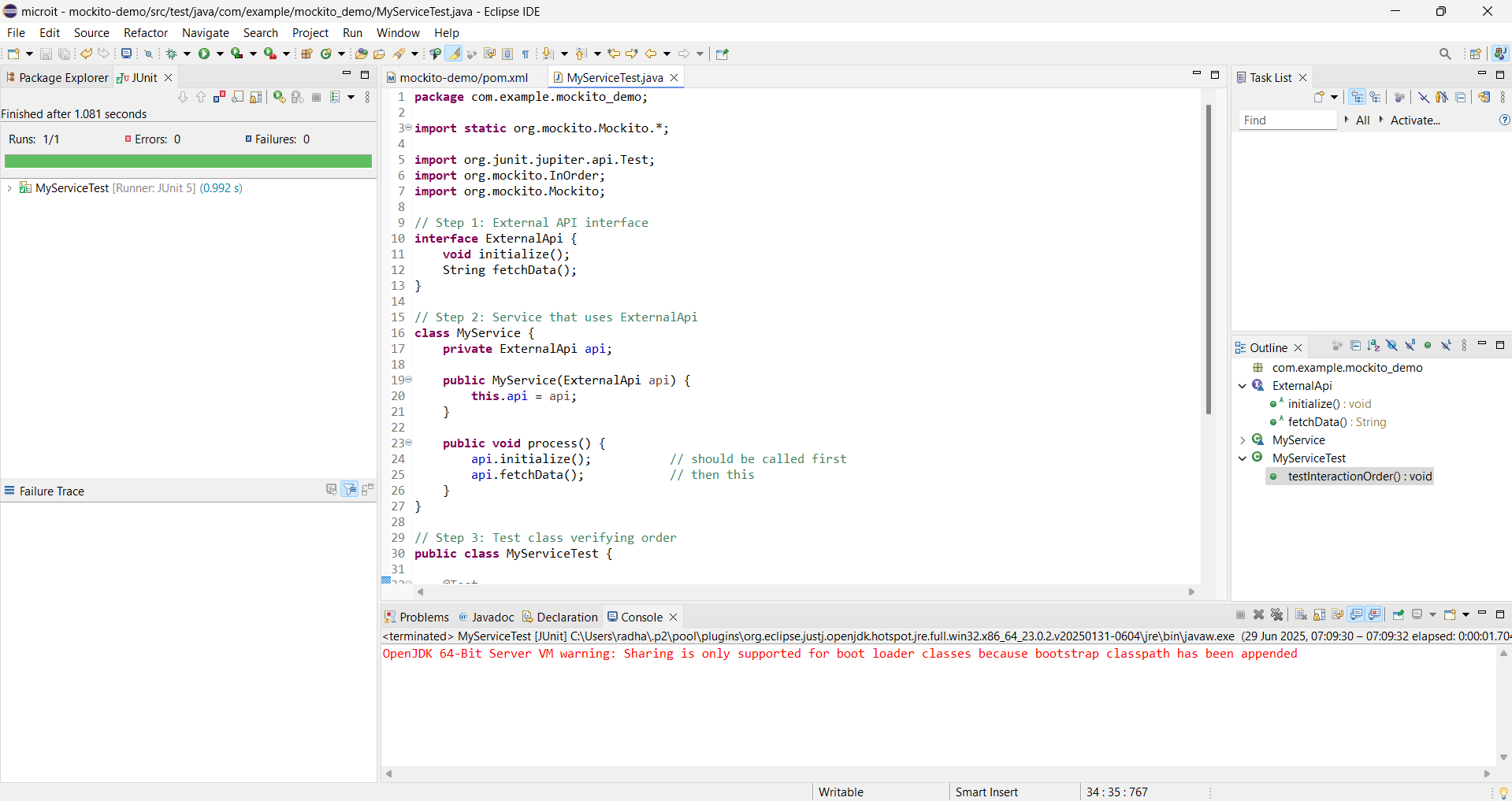
inOrder.verify(mockApi).initialize(); // Must come first

inOrder.verify(mockApi).fetchData(); // Must come after

}

}

**Output:**



**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.

**Solution:**

**package** com.example.mockito\_demo;

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

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

**import** org.junit.jupiter.api.Test;

**import** org.mockito.Mockito;

// External dependency with a void method

**interface** Auditor {

**void** audit(String event); // may throw

}

// Service under test

**class** MyService {

**private** **final** Auditor auditor;

MyService(Auditor auditor) {

**this**.auditor = auditor;

}

**void** performCriticalAction() {

// delegate to void method that might fail

auditor.audit("CRITICAL\_ACTION");

}

}

// Test ─ stub the void method to throw and verify the call

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

@Test

**void** auditFailurePropagates() {

// Create mock

Auditor mockAuditor = Mockito.*mock*(Auditor.**class**);

// Stub the void method to throw

*doThrow*(**new** RuntimeException("Audit failed"))

.when(mockAuditor).audit("CRITICAL\_ACTION");

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

// Assert that the exception is thrown

RuntimeException ex = *assertThrows*(RuntimeException.**class**,

service::performCriticalAction);

*assertEquals*("Audit failed", ex.getMessage());

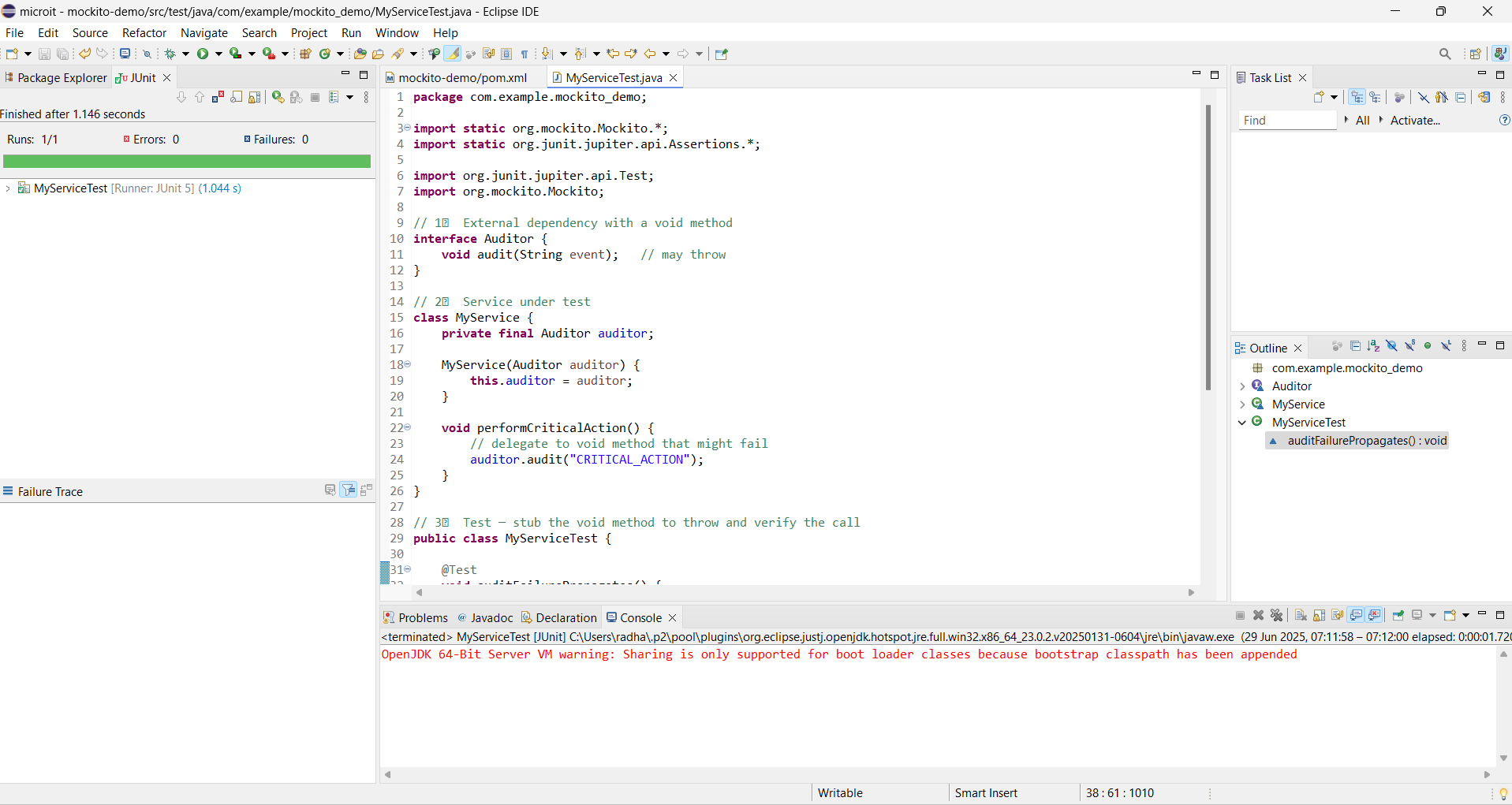
// Verify the interaction took place once with the expected arg

*verify*(mockAuditor, *times*(1)).audit("CRITICAL\_ACTION");

}

}

Output:



**Filename:SL4J Logging Exercises**

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

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

using SLF4J.

Step-by-Step Solution:

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

<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>

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

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");

}

}

**Solution:**

package com.example.logging;

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) {

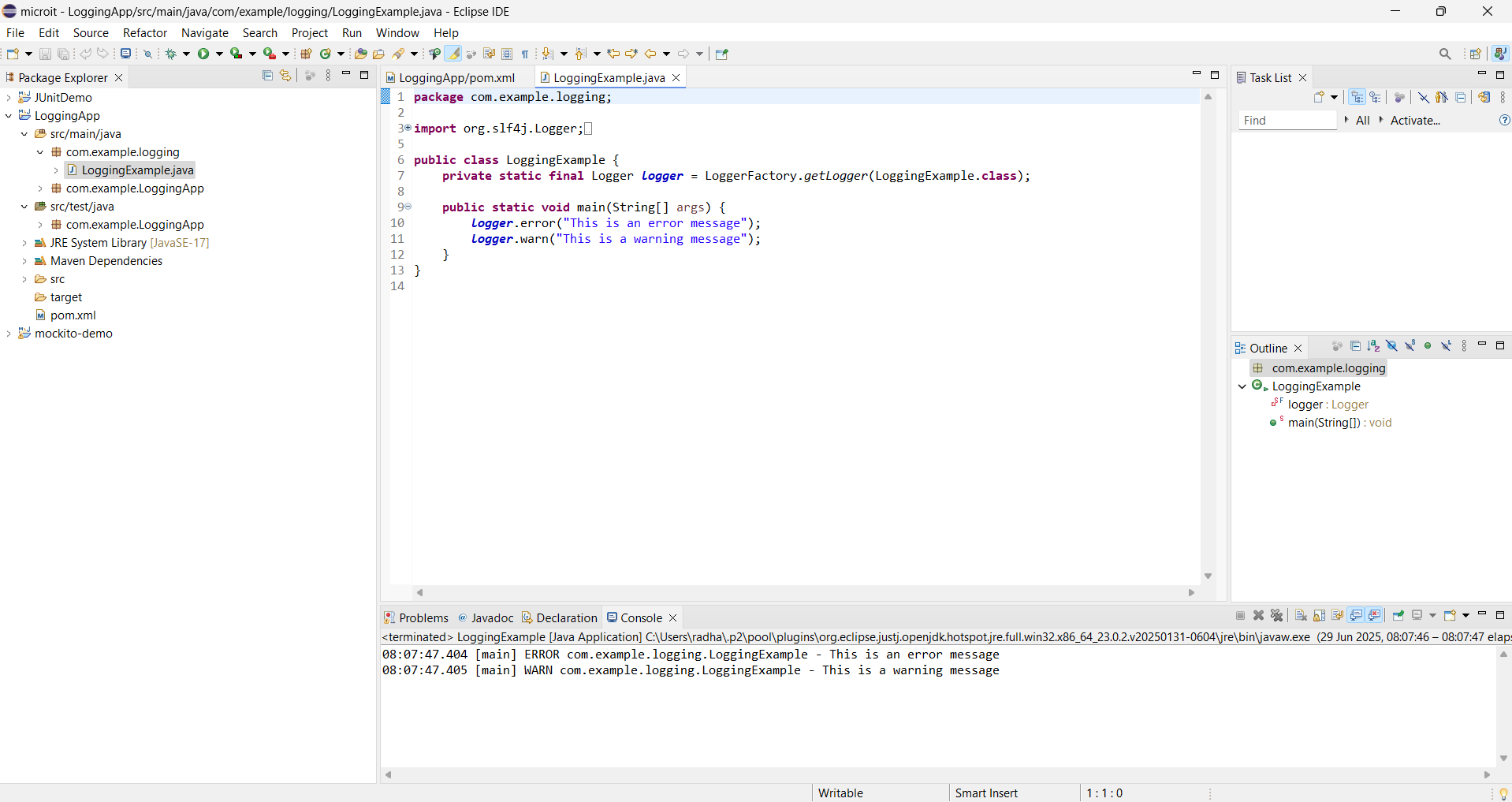
*logger*.error("This is an error message");

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

}

}

Output:



**Exercise 2: Parameterized Logging**

Task: Write a Java application that demonstrates parameterized logging using SLF4J.

Step-by-Step Solution:

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

<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>

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

Write code for this.

**Solution:**

**package** com.example.logging;

**import** org.slf4j.Logger;

**import** org.slf4j.LoggerFactory;

**public** **class** ParameterizedLoggingExample {

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

**public** **static** **void** main(String[] args) {

String user = "Likitha";

**int** age = 22;

String city = "Hyderabad";

// Parameterized logging using {} placeholders

***logger***.info("User {} is {} years old and lives in {}", user, age, city);

// Another example

**int** a = 10, b = 20;

***logger***.debug("Adding {} and {} gives {}", a, b, (a + b));

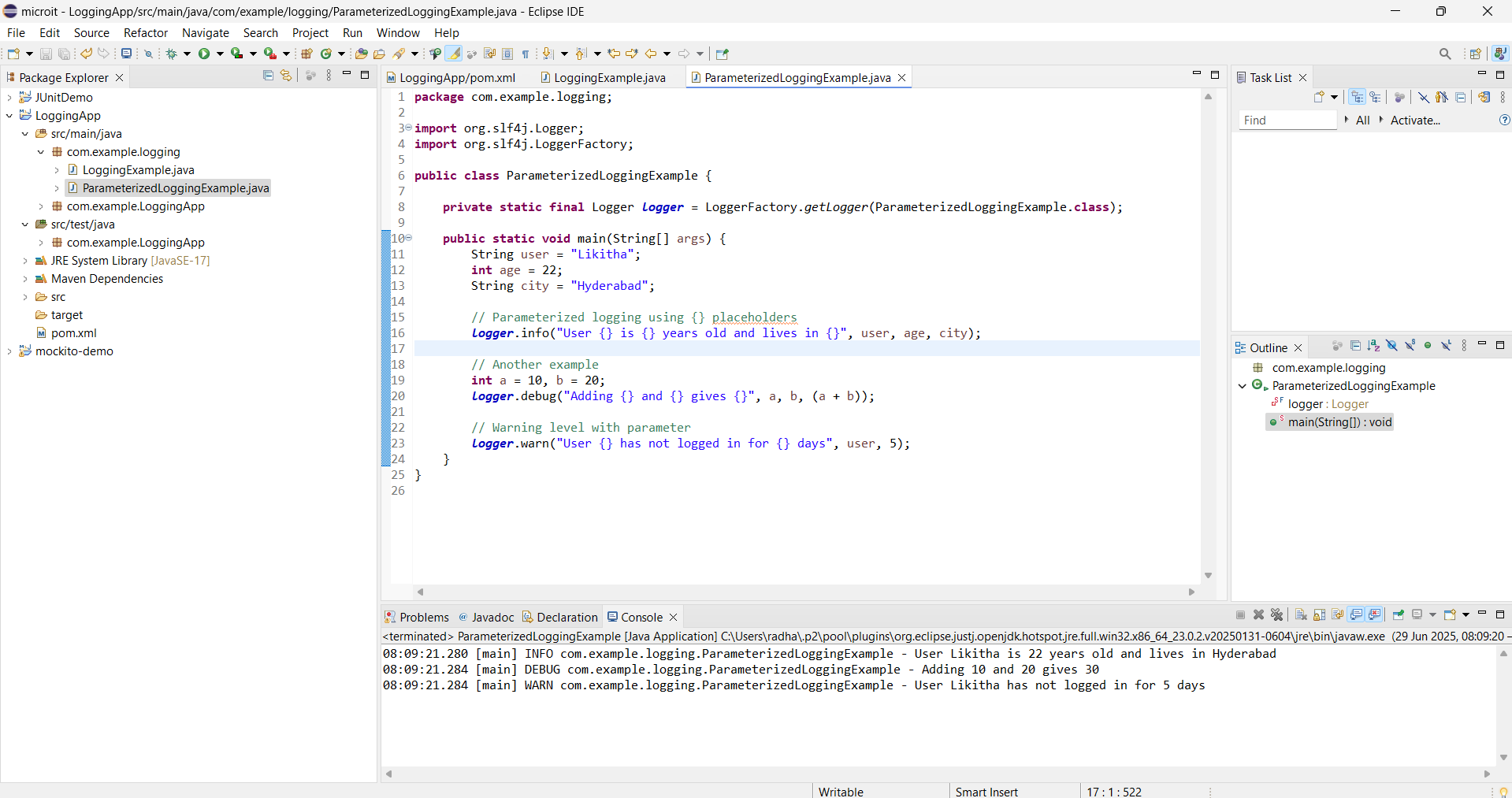
// Warning level with parameter

***logger***.warn("User {} has not logged in for {} days", user, 5);

}

}

**Output:**

****

**Exercise 3: Using Different Appenders**

Task: Write a Java application that demonstrates using different appenders with SLF4J.

Step-by-Step Solution:

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

<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>

2. Create a `logback.xml` configuration file to define different appenders:

<configuration>

<appender name="console" class="ch.qos.logback.core.ConsoleAppender">

<encoder>

<pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>

</encoder>

</appender>

<appender name="file" class="ch.qos.logback.core.FileAppender">

<file>app.log</file>

<encoder>

<pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>

</encoder>

</appender>

<root level="debug">

<appender-ref ref="console" />

<appender-ref ref="file" />

</root>

</configuration>

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

Write code for this.

**Solution:**

**package** com.example.logging;

**import** org.slf4j.Logger;

**import** org.slf4j.LoggerFactory;

**public** **class** MultiAppenderLoggingExample {

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

**public** **static** **void** main(String[] args) {

***logger***.info("Info log - visible in console and app.log");

***logger***.warn("Warning log - visible in console and app.log");

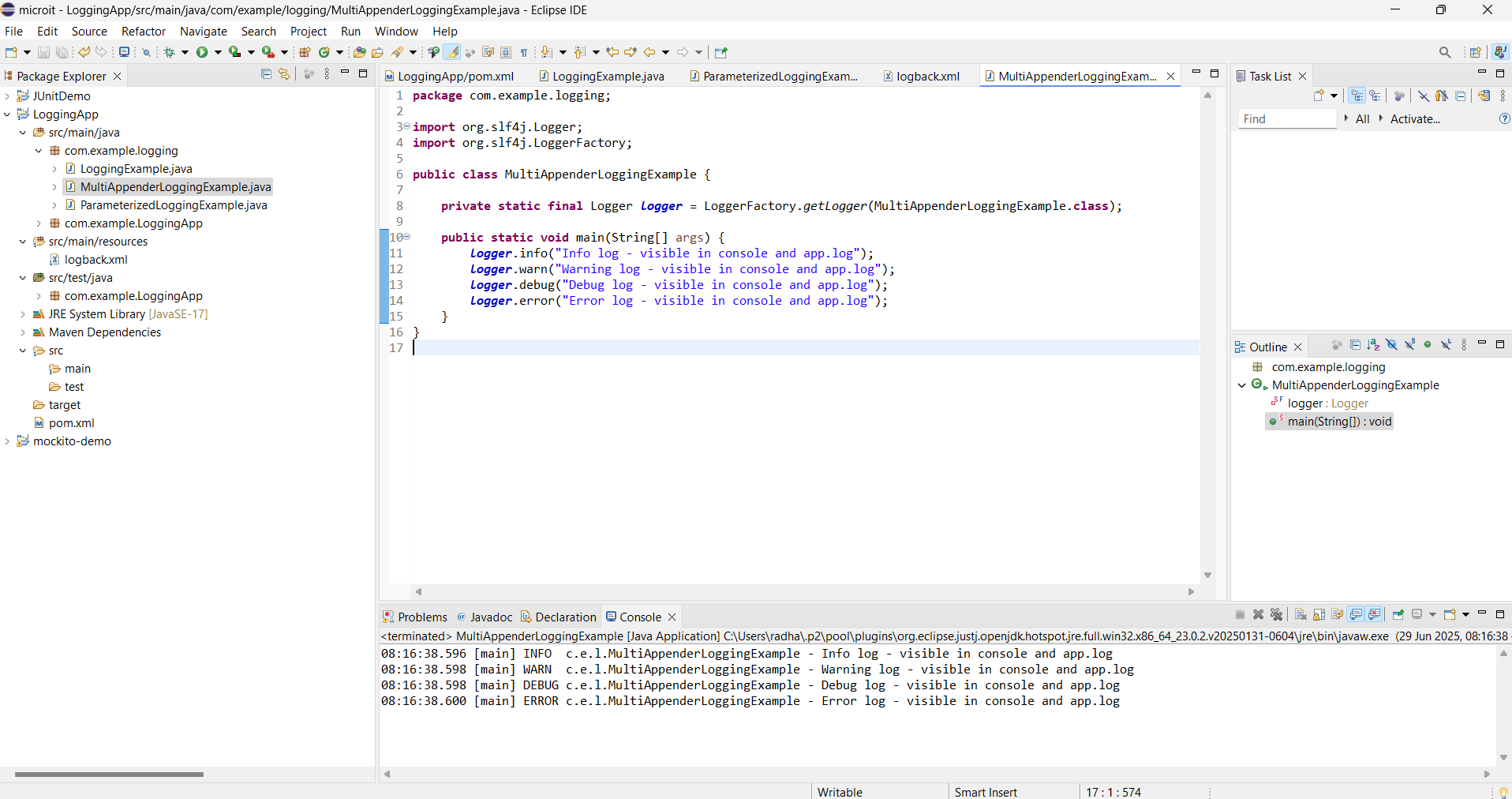
***logger***.debug("Debug log - visible in console and app.log");

***logger***.error("Error log - visible in console and app.log");

}

}

**Output:**

****