**WEEK-2**

**PL/SQL Exercises |Unit Testing |Mockito Exercises |SLF4J Logging exercises**

**Exercise 1: Control Structures**

**Scenario 1:** The bank wants to apply a discount to loan interest rates for customers above 60 years old.

* + **Question:** Write a PL/SQL block that loops through all customers, checks their age, and if they are above 60, apply a 1% discount to their current loan interest rates.

**Code:**

CREATE PROCEDURE ApplyDiscountToSeniors()

-> BEGIN

-> UPDATE Loans l

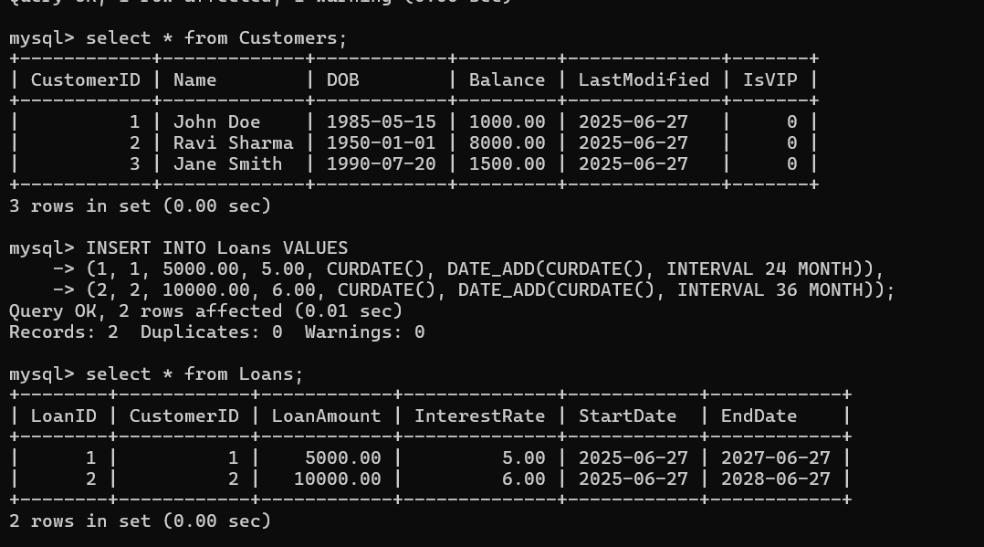
-> JOIN Customers c ON l.CustomerID = c.CustomerID

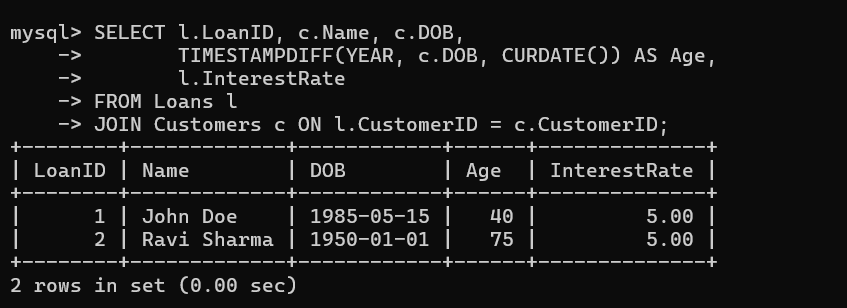
-> SET l.InterestRate = l.InterestRate - 1

-> WHERE TIMESTAMPDIFF(YEAR, c.DOB, CURDATE()) > 60;

-> END$$

**Output:**

** Before**

** After**

**Scenario 2:** A customer can be promoted to VIP status based on their balance.

**Question:** Write a PL/SQL block that iterates through all customers and sets a flag IsVIP to TRUE for those with a balance over $10,000.

**Code:**

CREATE PROCEDURE PromoteVIPs()

BEGIN

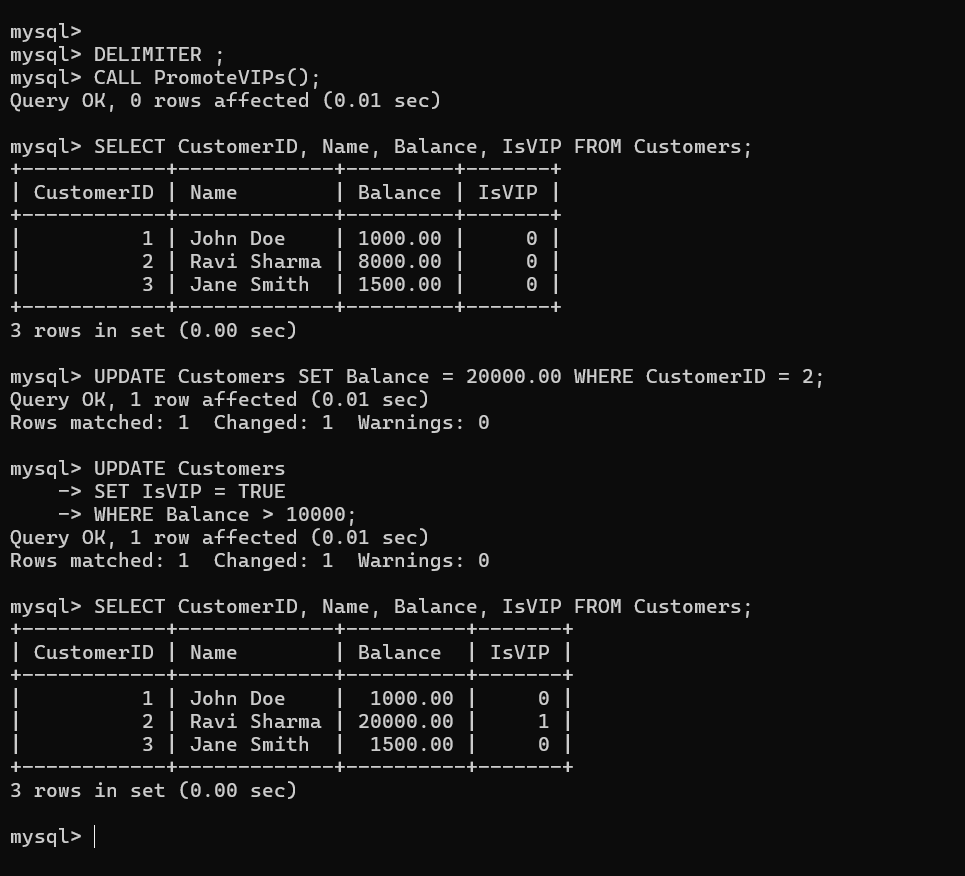
UPDATE Customers

SET IsVIP = TRUE

WHERE Balance > 10000;

END$$

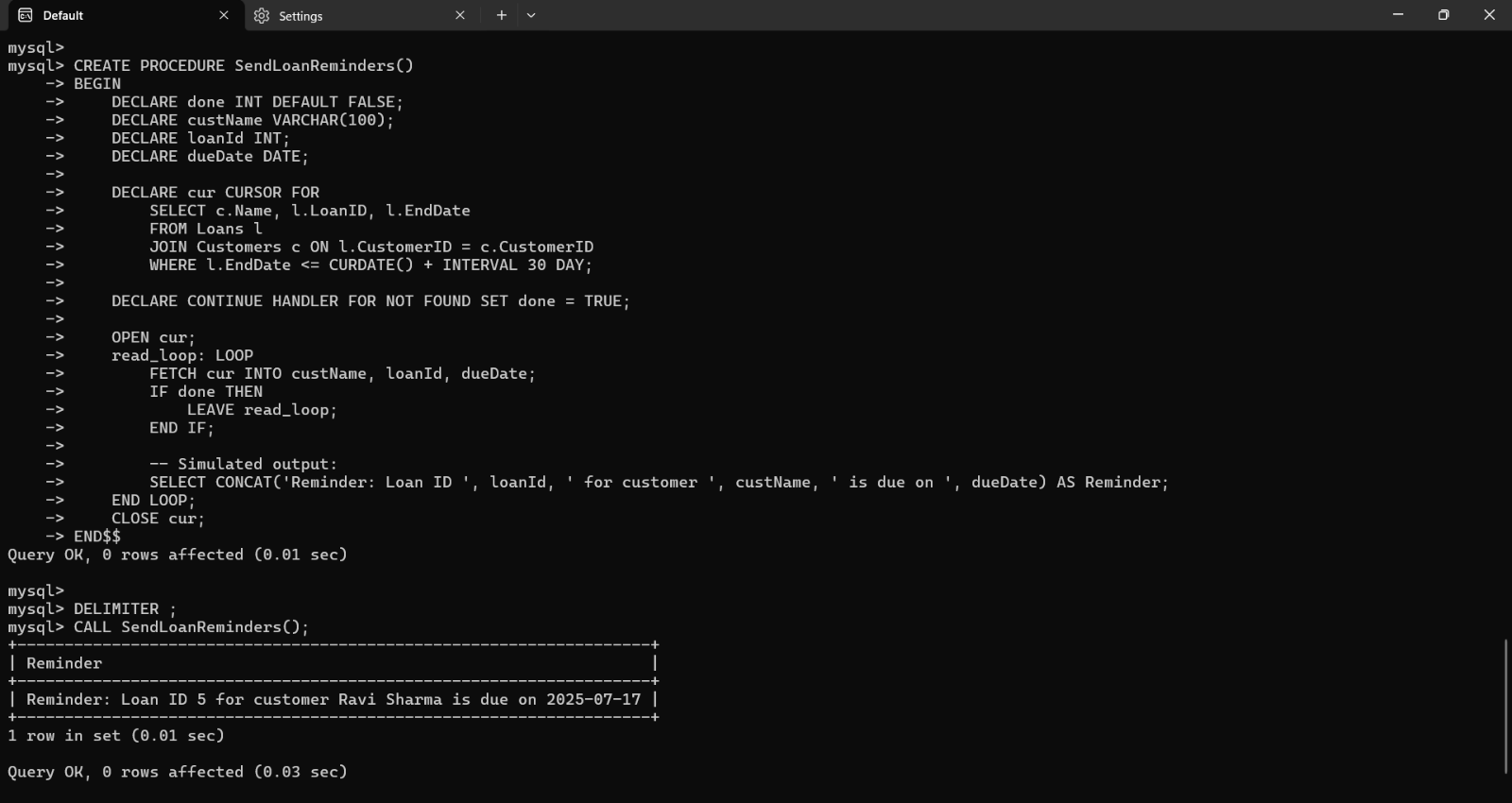
**Output:**

****

**Scenario 3:** The bank wants to send reminders to customers whose loans are due within the next 30 days.

**Question:** Write a PL/SQL block that fetches all loans due in the next 30 days and prints a reminder message for each customer.

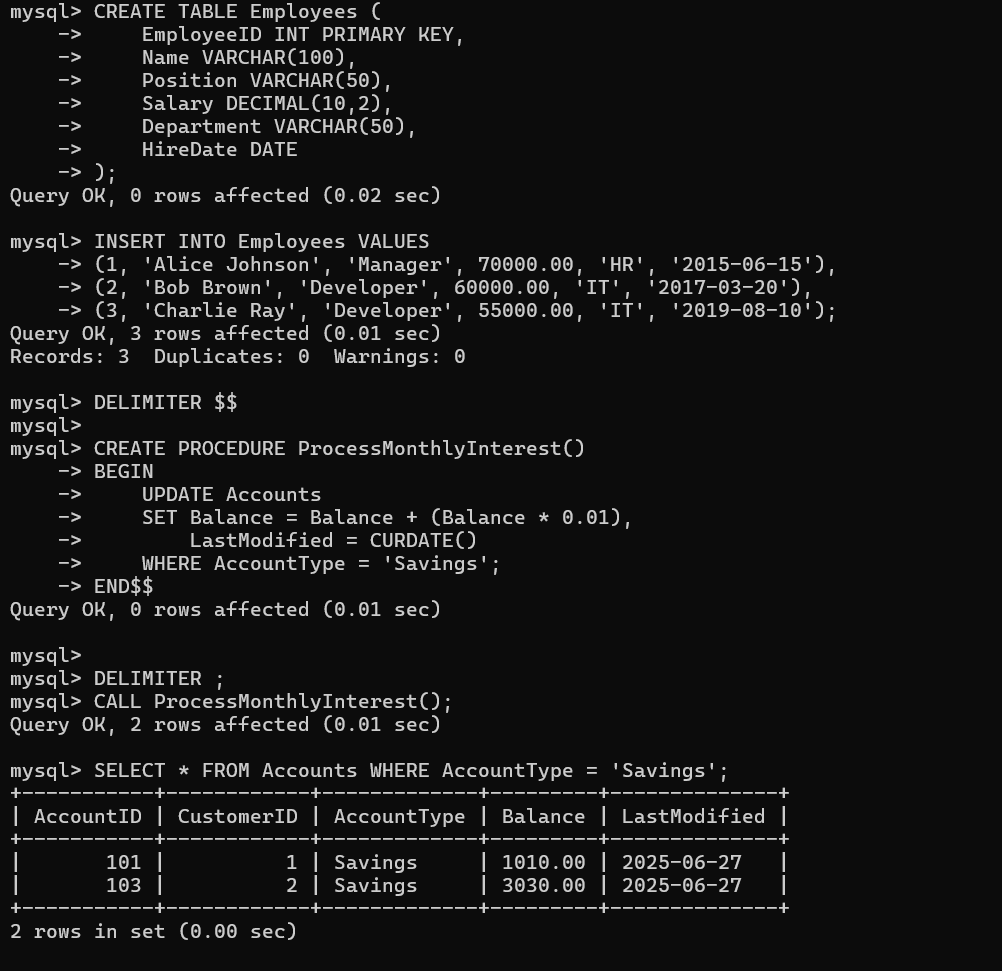
**Code:**



**Exercise 3: Stored Procedures**

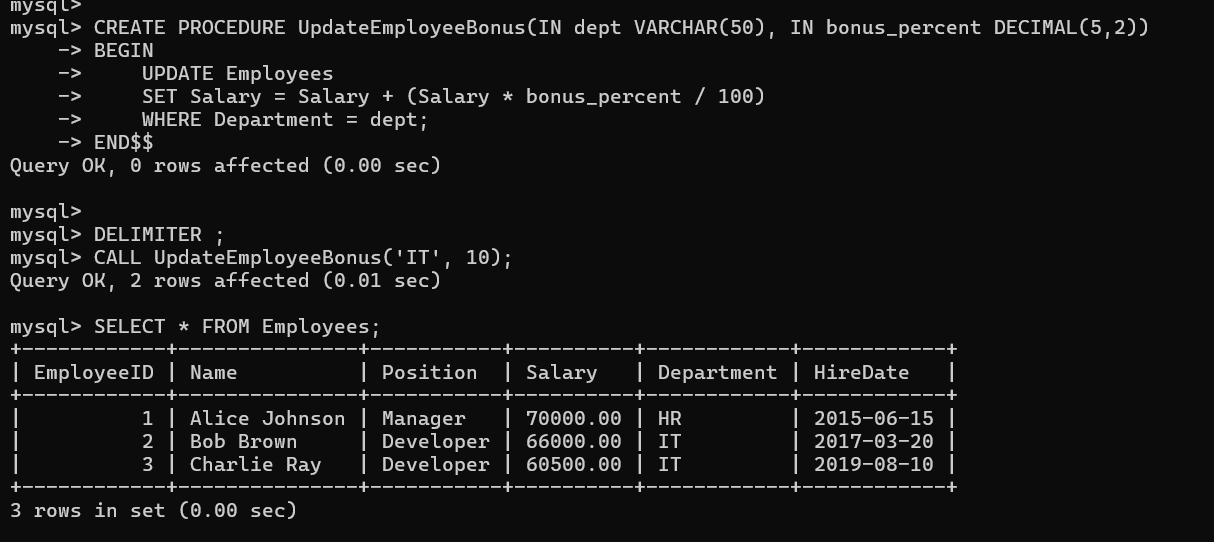
**Scenario 1:** The bank needs to process monthly interest for all savings accounts.

**Question:** Write a stored procedure **ProcessMonthlyInterest** that calculates and updates the balance of all savings accounts by applying an interest rate of 1% to the current balance.



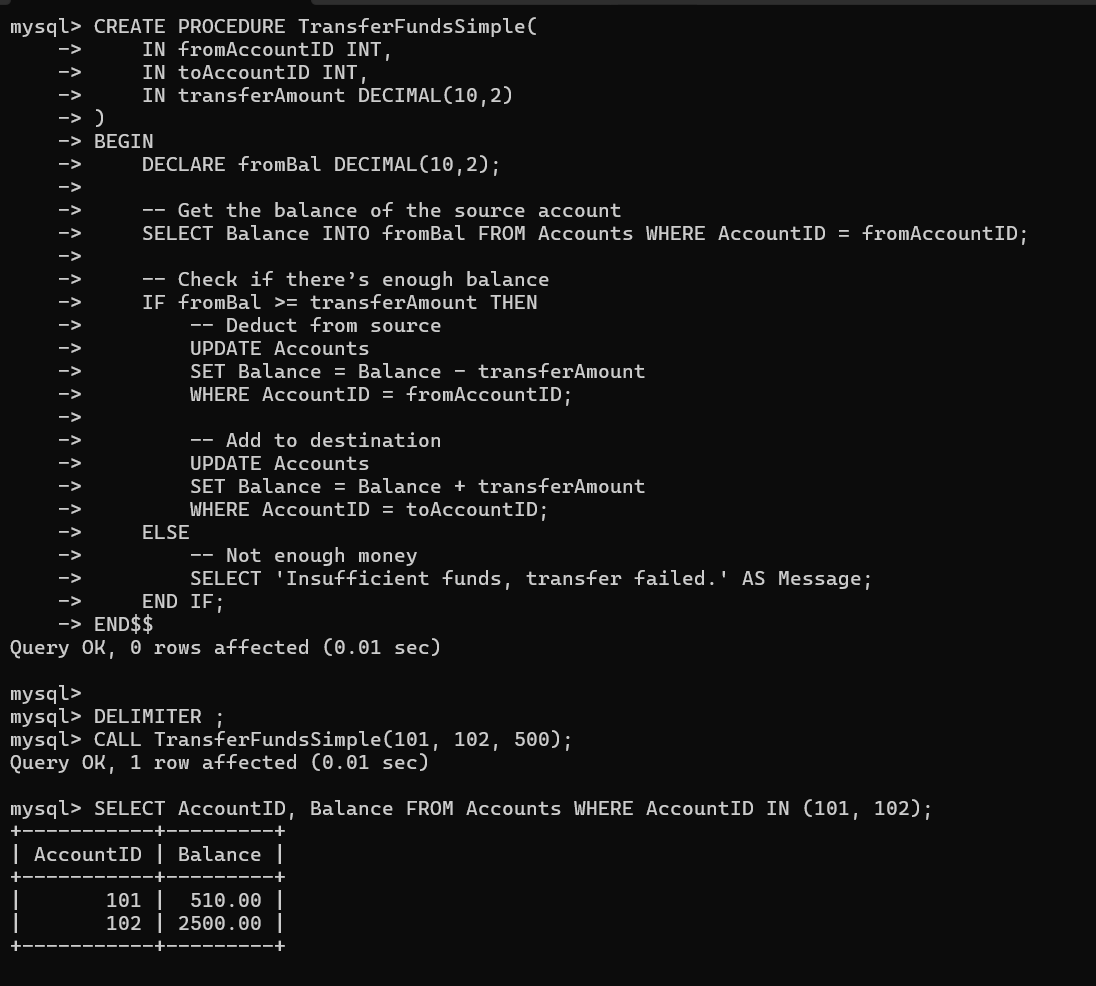
**Scenario 2:** The bank wants to implement a bonus scheme for employees based on their performance.

**Question:** Write a stored procedure **UpdateEmployeeBonus** that updates the salary of employees in a given department by adding a bonus percentage passed as a parameter.



**Scenario 3:** Customers should be able to transfer funds between their accounts.

* + **Question:** Write a stored procedure **TransferFunds** that transfers a specified amount from one account to another, checking that the source account has sufficient balance before making the transfer.



**JUnit Testing**

**Exercise 1: Setting Up JUnit**

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

**Code:**

* **Pom.xml**

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13.2</version>

<scope>test</scope>

</dependency>

* **Calculator.java**

package com.example;

public class Calculator {

public int add(int a, int b) {

return a + b; }}

* **CalculatorTest.java**

package com.example;

import static org.junit.Assert.\*;

import org.junit.Test;

public class CalculatorTest {

@Test

public void testAdd() {

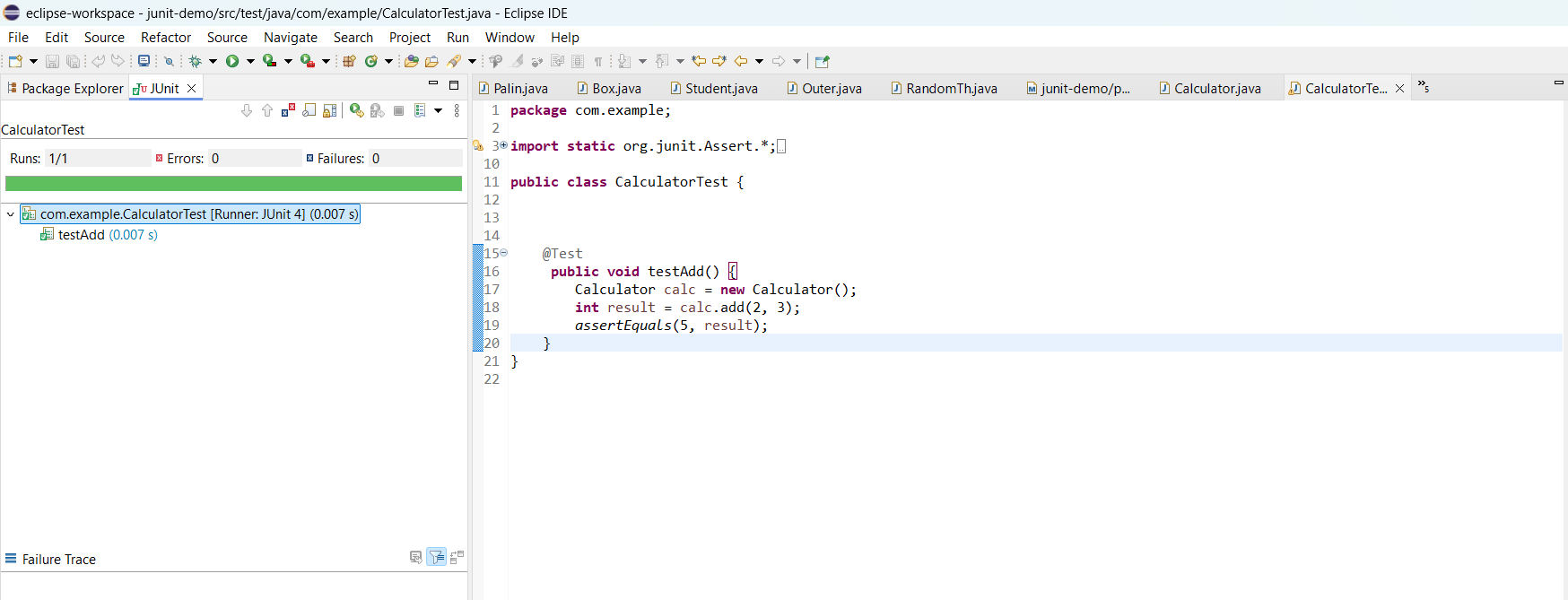
Calculator calc = new Calculator();

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

assertEquals(5, result);

}}

**Output:**

****

**Exercise 3: Assertions in JUnit**

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

**Steps:**

1. Created a new test class named `AssertionsTest` in the test folder (`src/test/java`).

2. Used different assertion methods like `assertEquals`, `assertTrue`, `assertFalse`, `assertNull`, and `assertNotNull`.

3. Ran the test using Eclipse's JUnit runner.

**Code:**

package com.example;

import static org.junit.Assert.\*;

import org.junit.Test;

public class AssertionsTest {

@Test

public void testAssertions() {

*assertEquals*(5, 2 + 3);

*assertTrue*(5 > 3);

*assertFalse*(5 < 3);

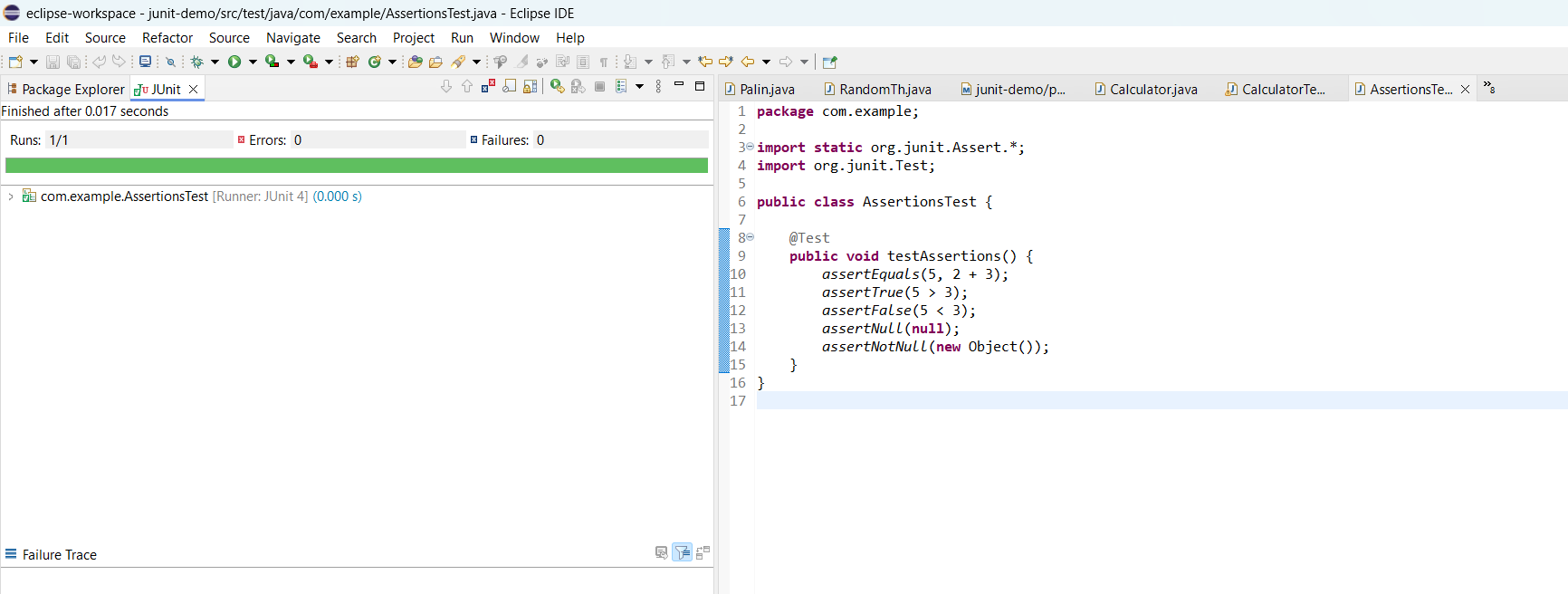
*assertNull*(null);

*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.**

**Code:**

**Calculator.java**

package com.example;

public class Calculator {

public int add(int a, int b) {

return a + b;

}

public int multiply(int a, int b) {

return a \* b;

}

}

**CalculatorTestAAA.java**

package com.example;

import static org.junit.Assert.\*;

import org.junit.After;

import org.junit.Before;

import org.junit.Test;

public class CalculatorTestAAA {

private Calculator calculator;

@Before

public void setUp() {

calculator = new Calculator(); // Arrange

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

}

@After

public void tearDown() {

calculator = null;

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

}

@Test

public void testAddition() {

int result = calculator.add(10, 5);

*assertEquals*(15, result);

}

@Test

public void testMultiplication() {

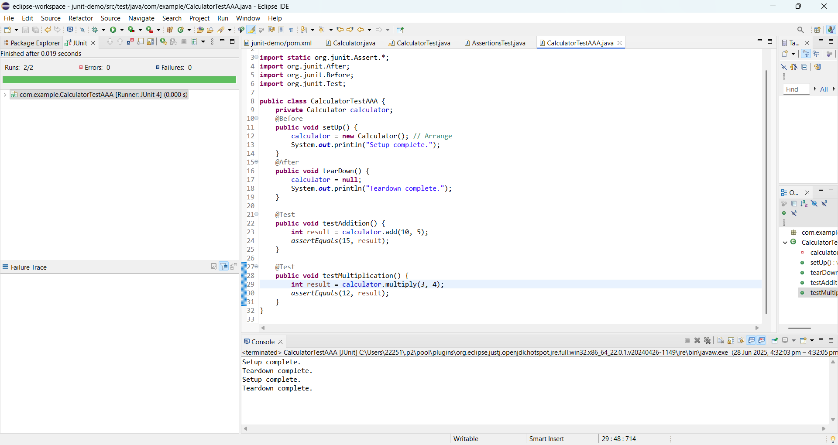
int result = calculator.multiply(3, 4);

*assertEquals*(12, result);

}

}

**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.**

**Code:**

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

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

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);

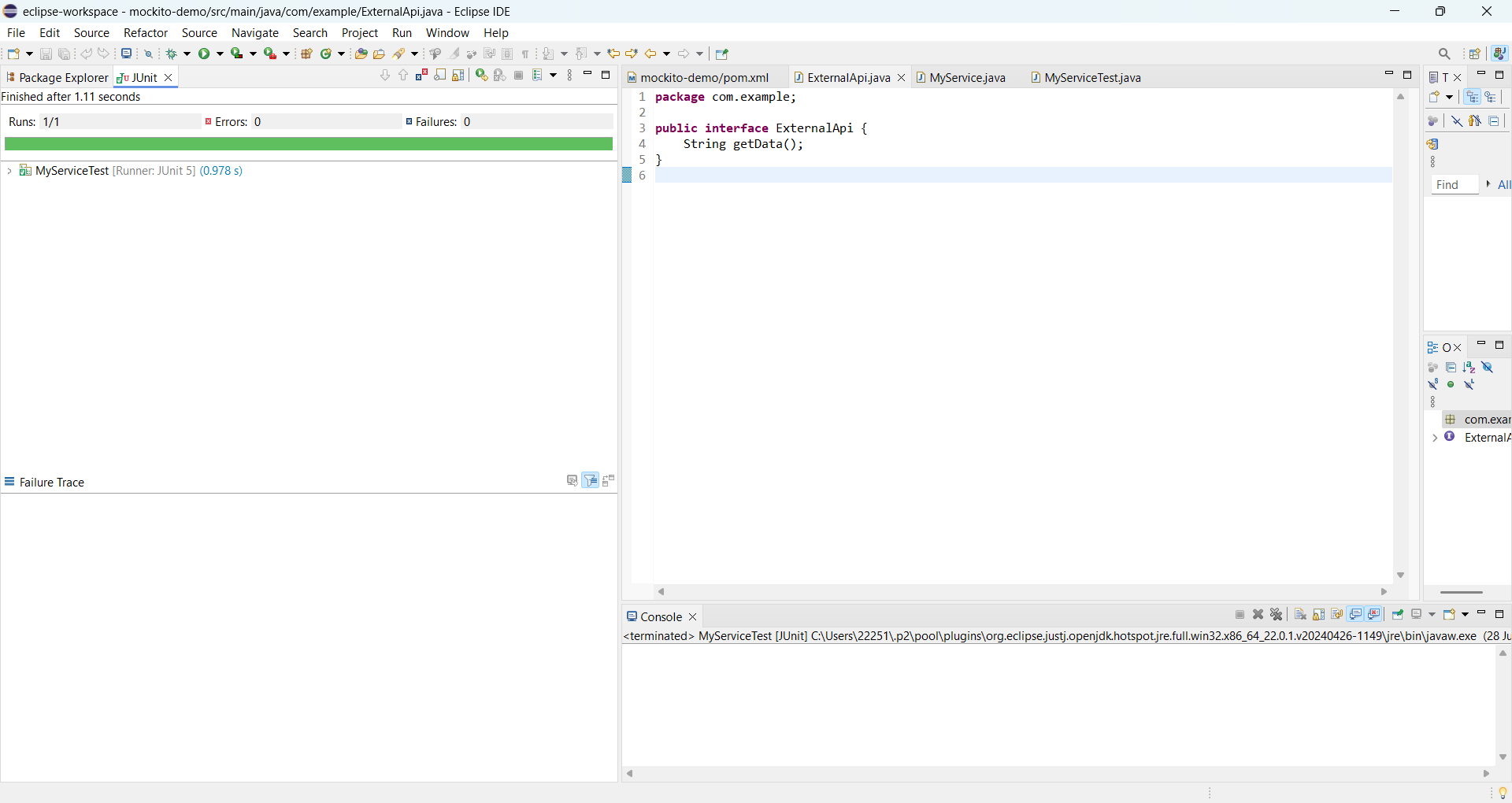
String result = service.fetchData();

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

}

}

**Output:**

****

**Exercise 2: Verifying Interactions**

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

**Code:**

**MyServiceTest.java**

package com.example;

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);

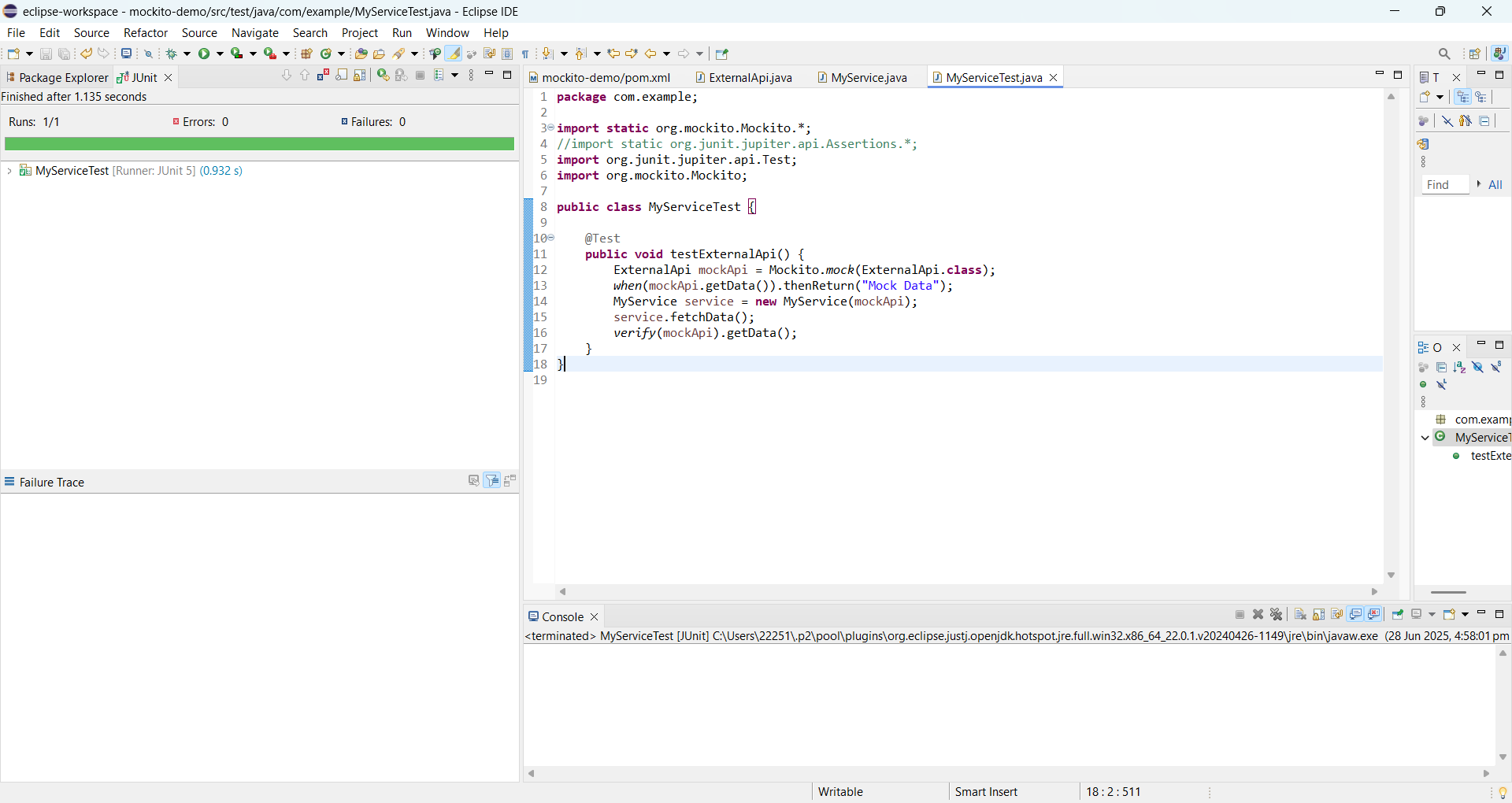
service.fetchData();

verify(mockApi).getData(); // Verifies interaction

}

}

**Output:**

****

**SLF4J 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**

**Code:**

**pom.xml**

<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>

**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) {

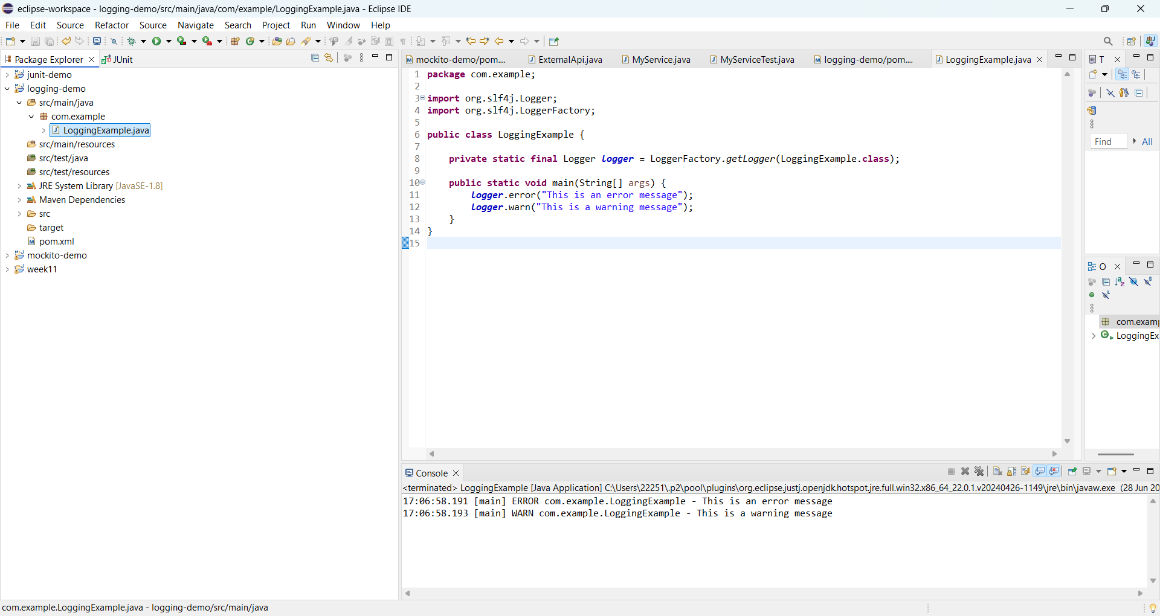
logger.error("This is an error message");

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

}

}

**Output:**

****