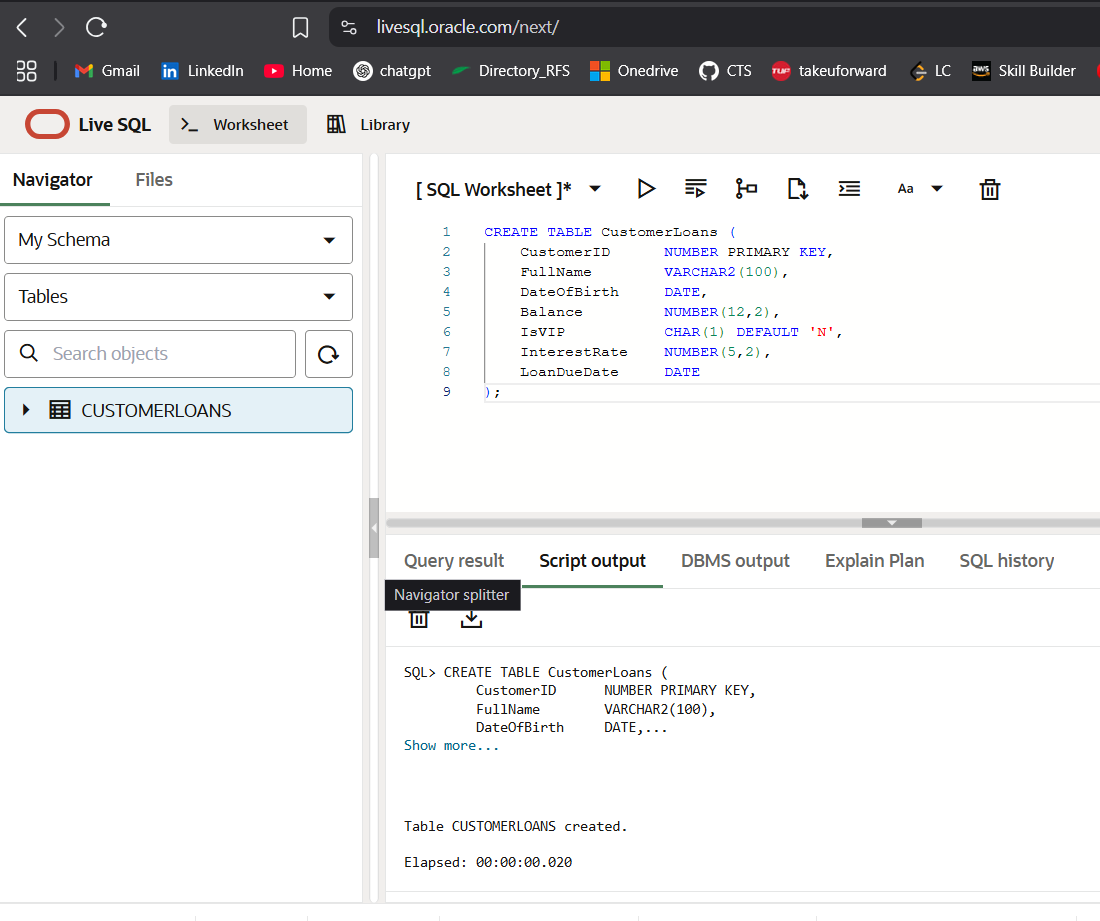
**PL/SQL EXCERCISES**

**Exercise 1: Control Structures**

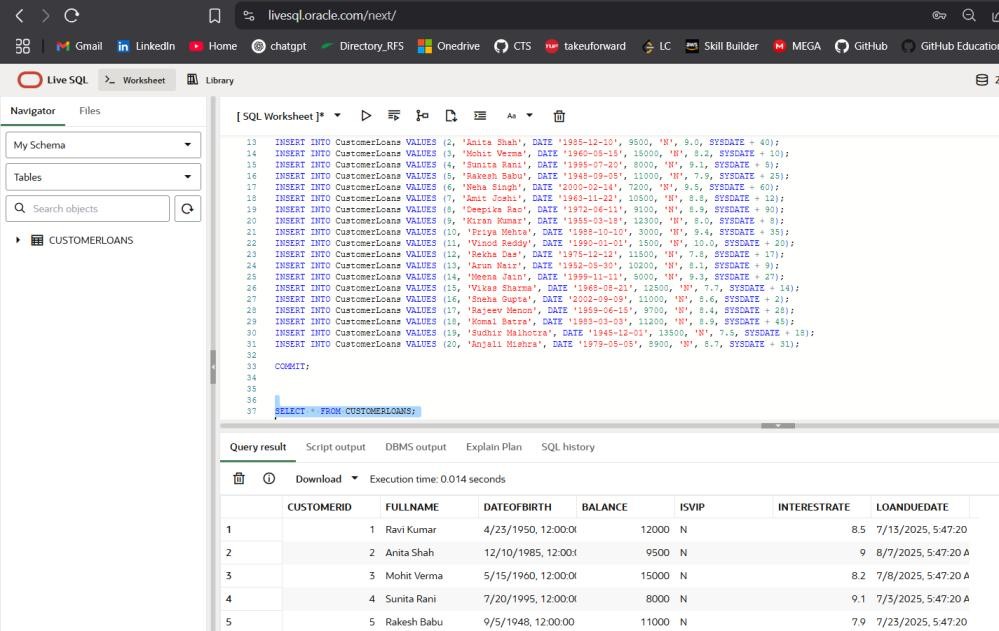
**Scenario 1:** The bank needs to process monthly interest for all savings accounts.

* **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.

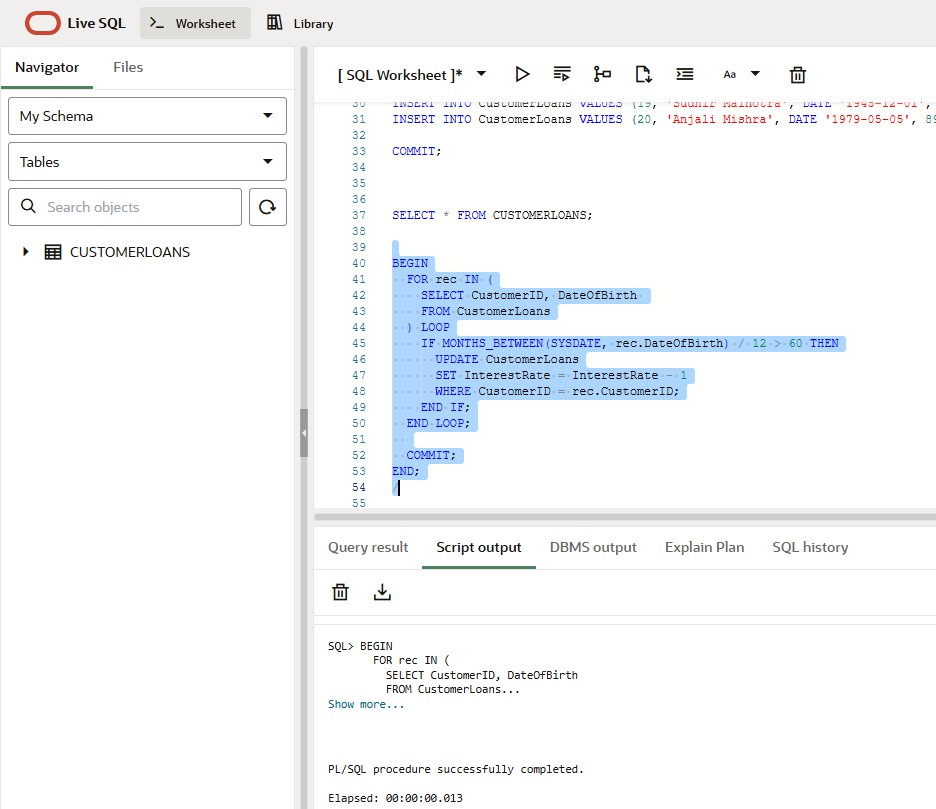
**Step-1: Create required tables**

****

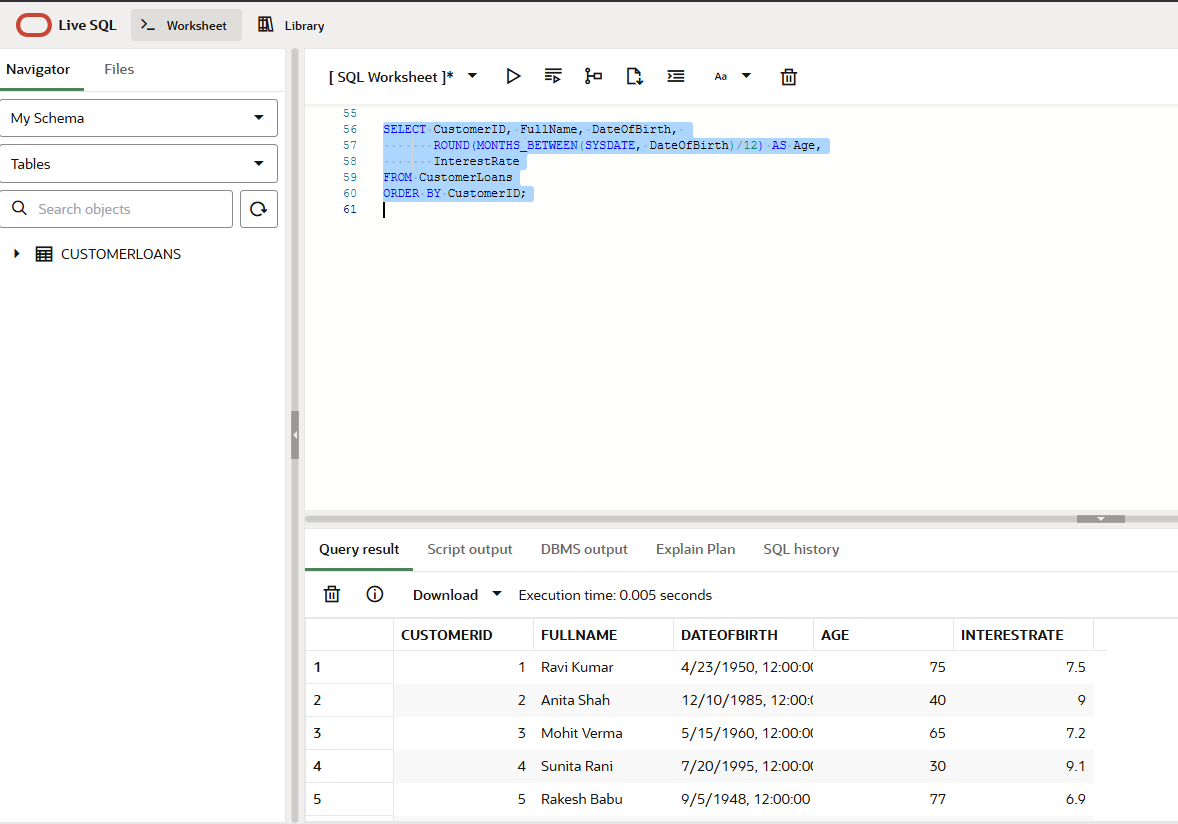
**Step-2: Insert data into table**

****

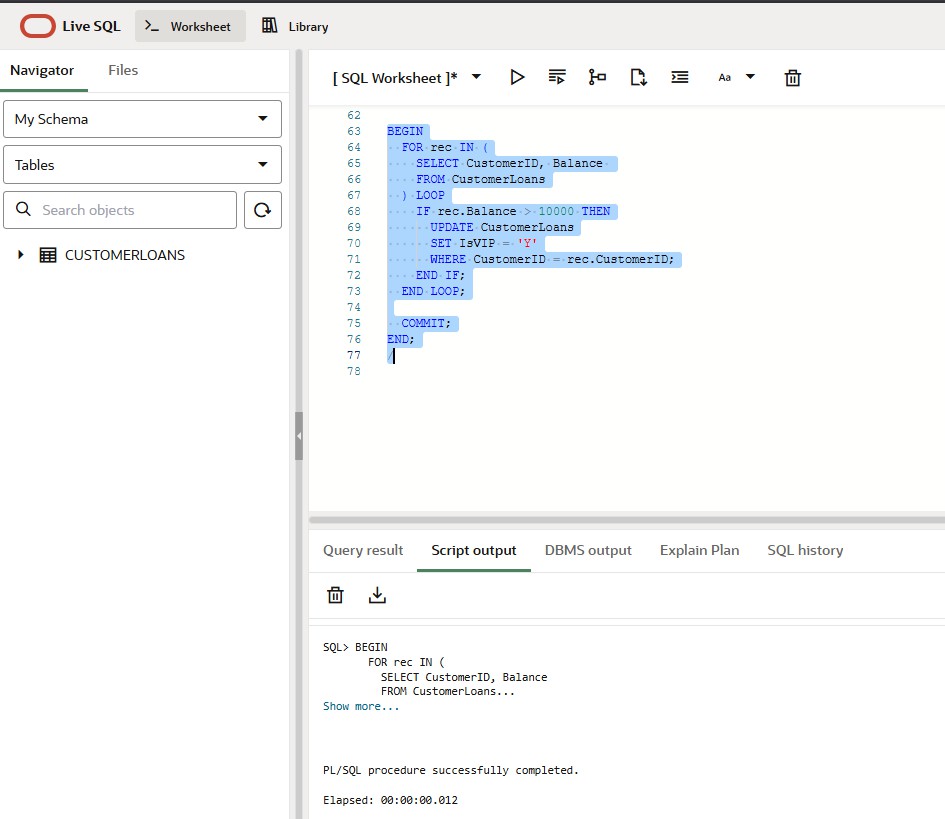
**Step-3: Scenario 1 – Apply 1% Discount for Age > 60:**

****

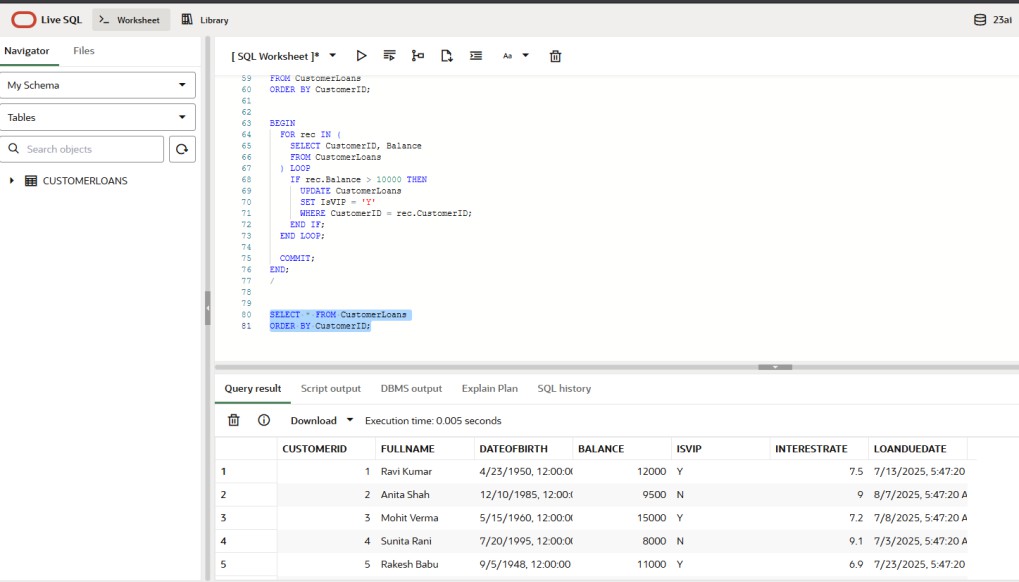
**Output**:



**Step-4: Scenario 2 – Promote to VIP for Balance > $10,000**

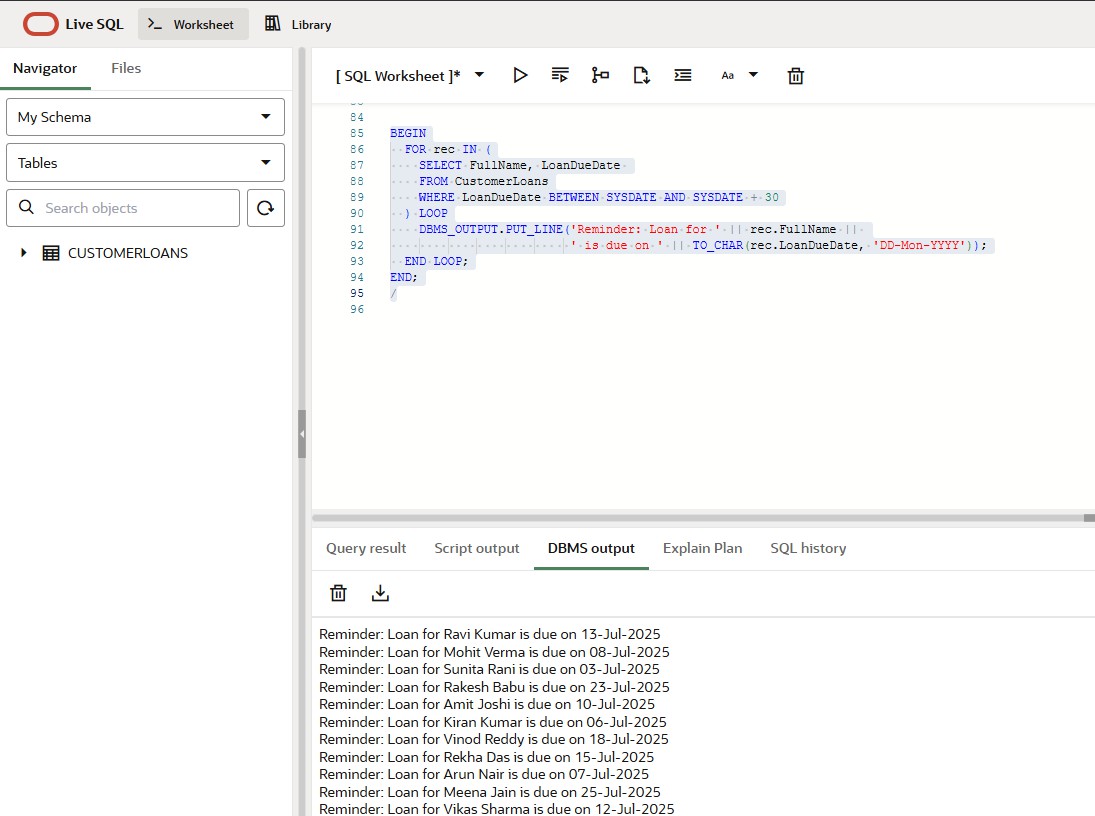
****

**Output:**

****

**Step-5: Scenario 3 – Loan Due Reminder (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**

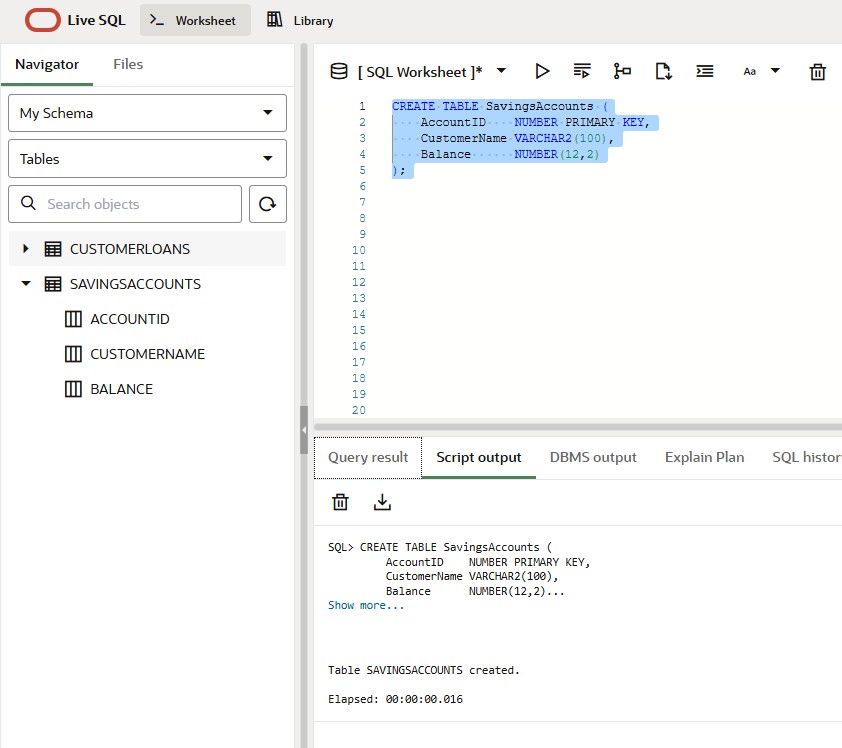
****

**Exercise 3: Stored Procedures**

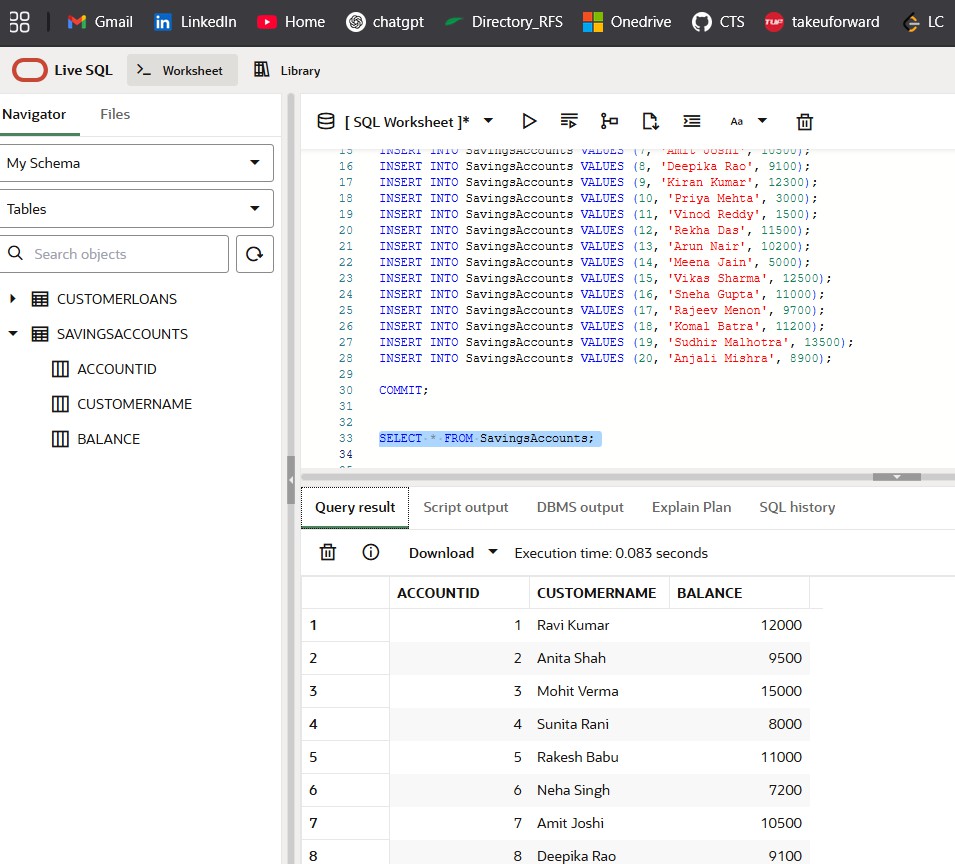
**Scenario 1: The bank needs to process monthly interest for all savings accounts.**

* **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.**

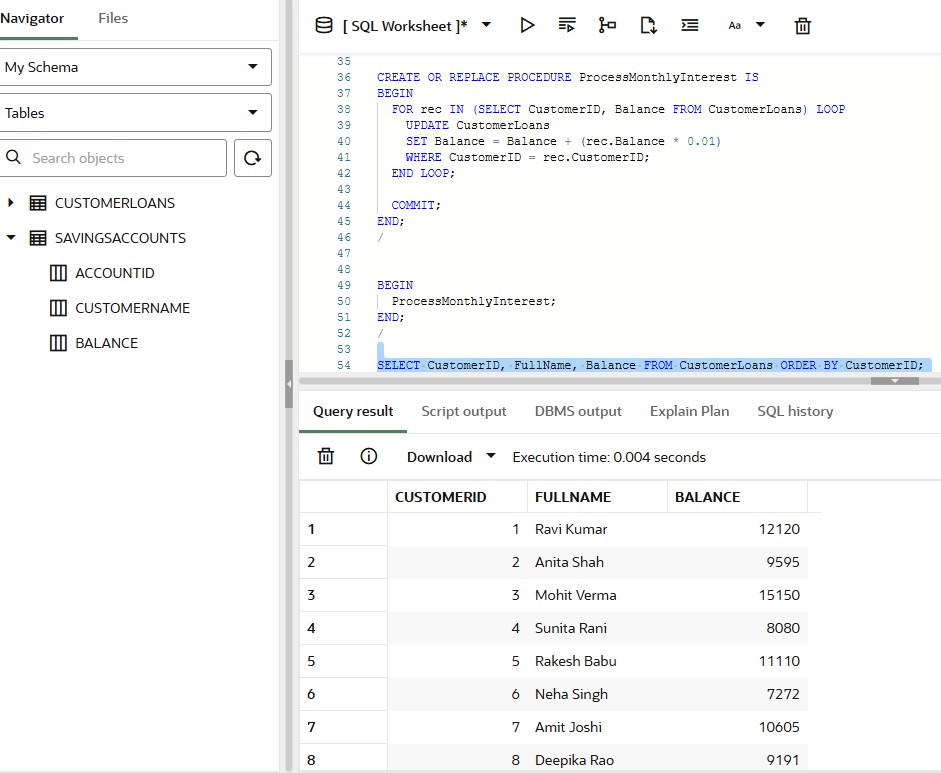
**Step 1: Create SavingsAccounts table and insert data**

****

**Step-2: Insert data into table**

****

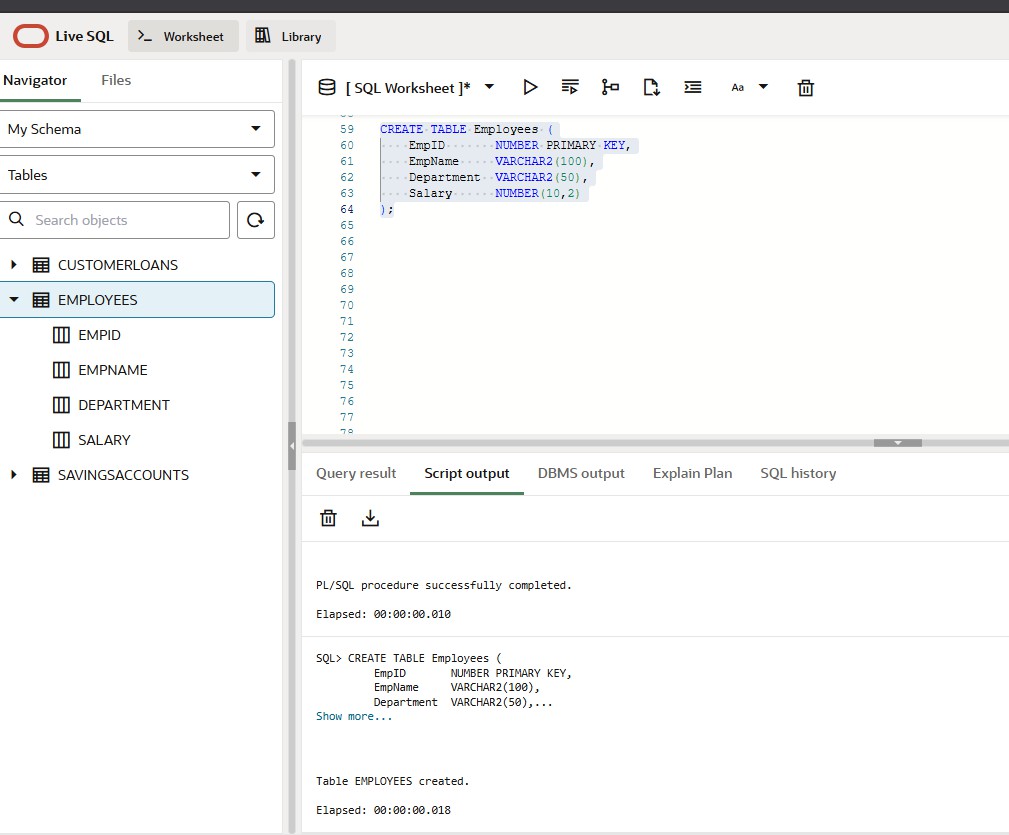
**Step 3: A stored procedure to calculate monthly interest**

****

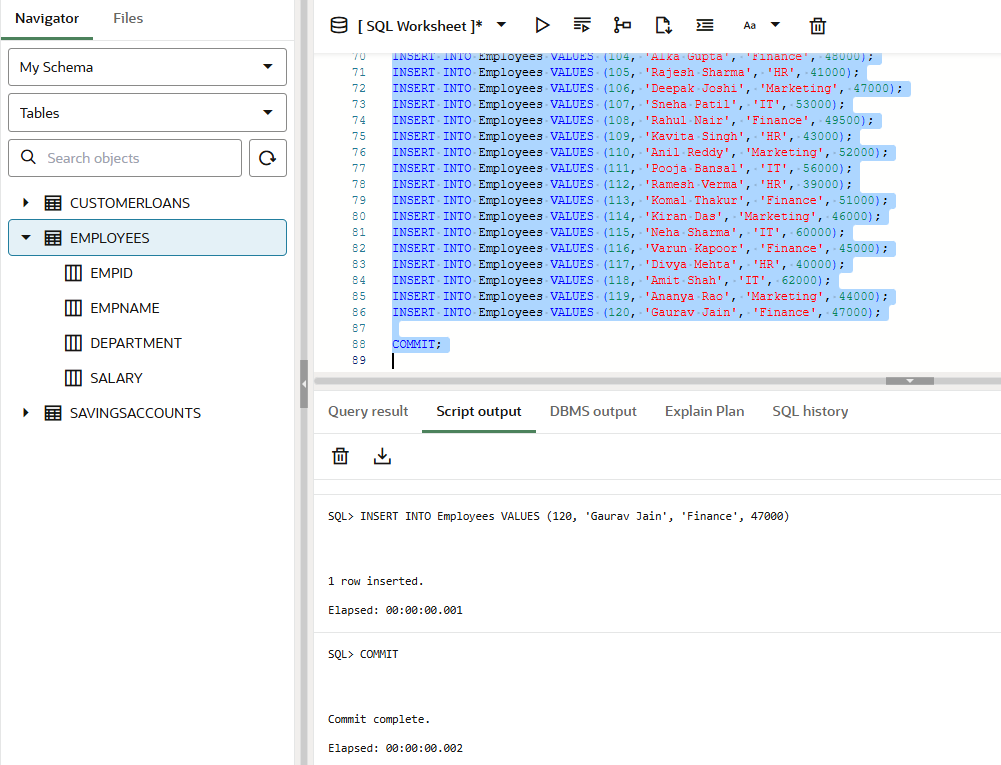
**Scenario 2: The bank wants to implement a bonus scheme for employees based on their performance.**

* **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.**

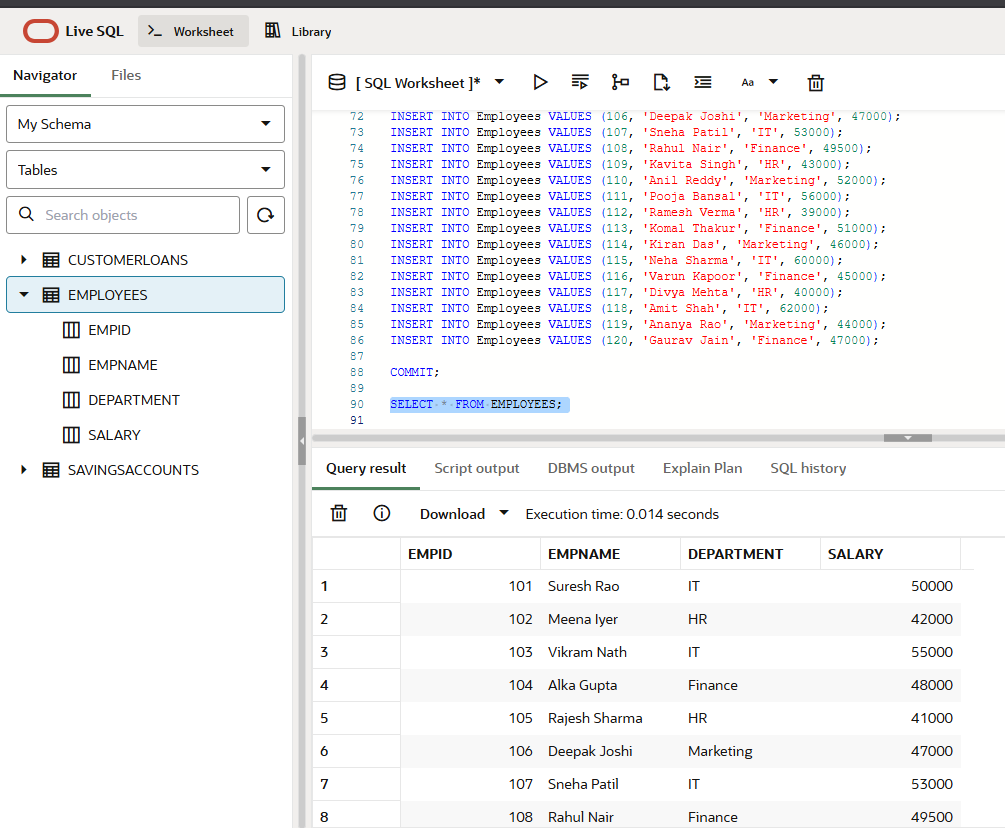
**Step 1: Create Employees table**



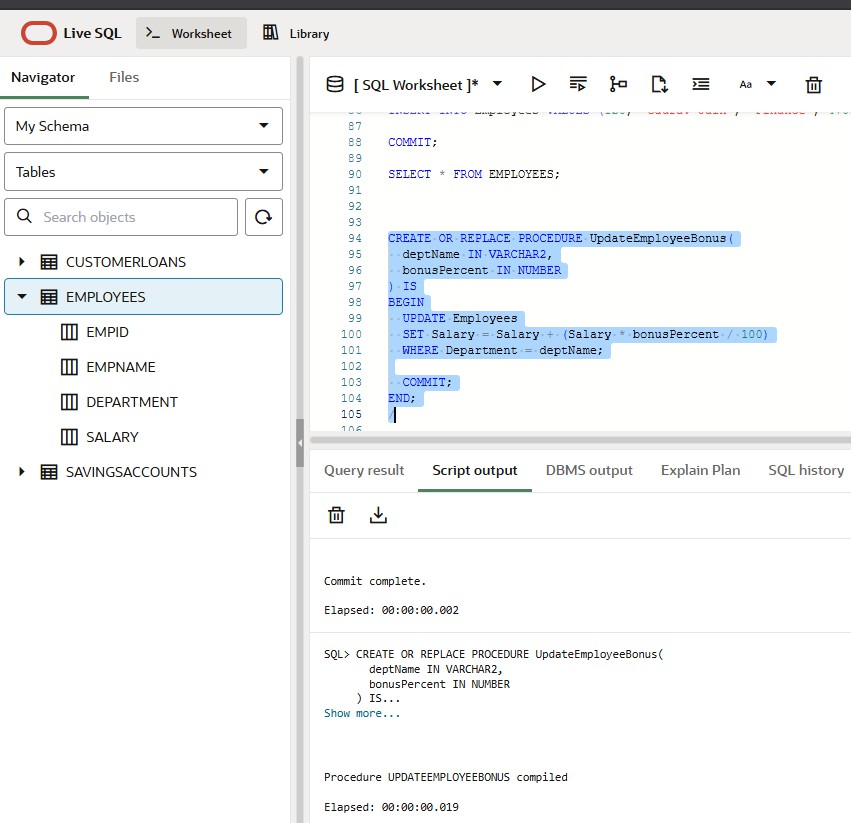
**Step-2: Insert data into table**

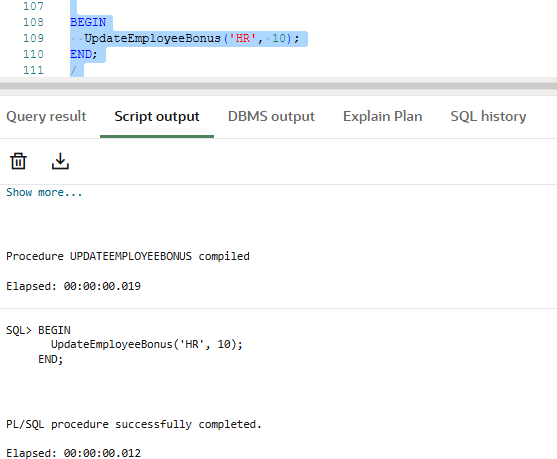
****

**Output:**

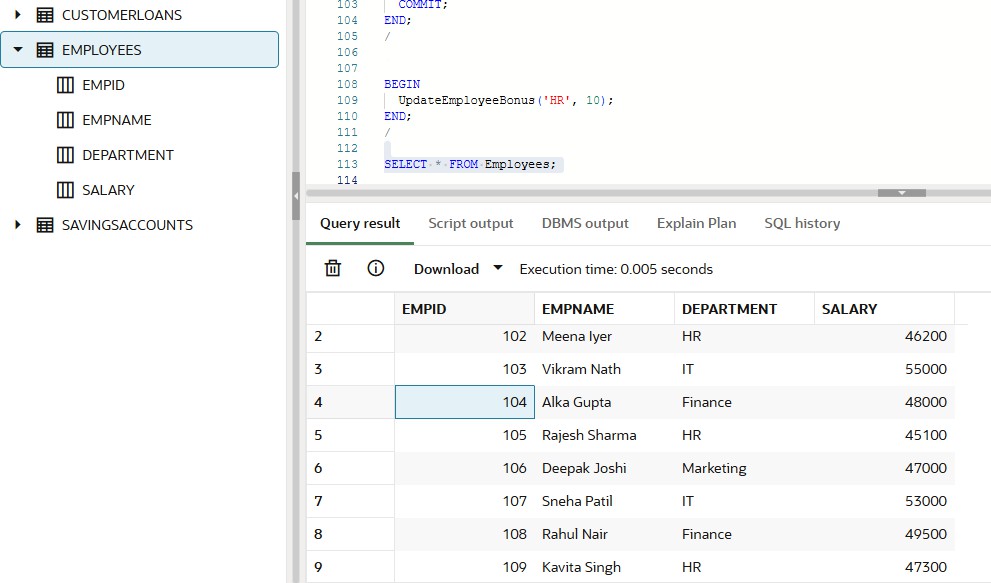
****

**Step 3: A Stored procedure to update the salary of employees in a given department**

****



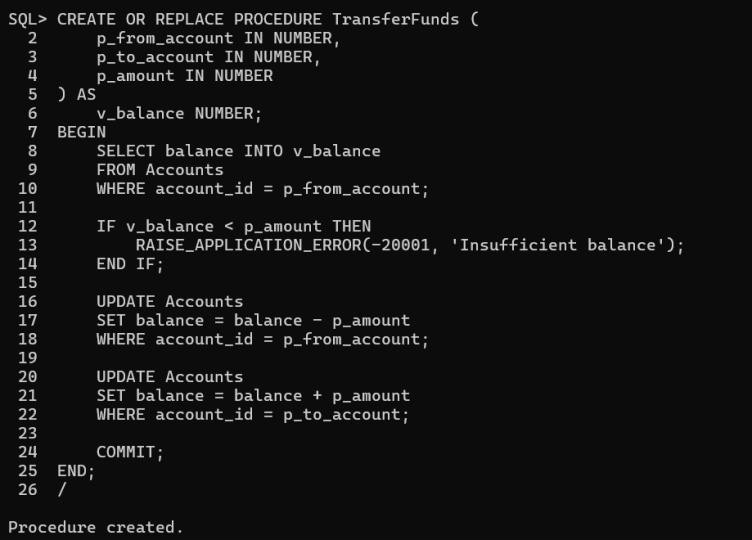
**Output:**

****

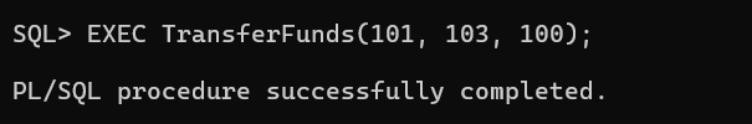
**Scenario 3: Customers should be able to transfer funds between their accounts.**

* **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.**

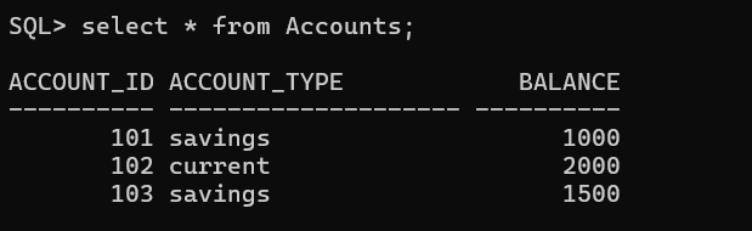
**Step 1: Used the same accounts table and created a procedure to transfer funds between accounts**

****

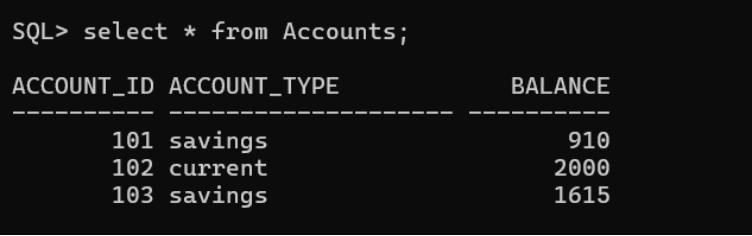
**Step 2: Transferring funds from one account to another**

****

**Step 3: Results Before Transferring**

****

**After Transfer**

****

# Exercise – 1

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

**Steps:**

#### Create a new Java project in your IDE

Created a project with Group Id: com.example & Artifact Id: calculator

#### Add JUnit dependency to your project. If you are using Maven, add the following to your pom.xml:

<dependency>

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

<artifactId>junit-jupiter</artifactId>

<version>5.10.2</version>

<scope>test</scope>

</dependency>

#### Create a new test class in your project.

* + **Create a new Java 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;

}

public int multiply(int a, int b) { return a \* b;

}

public double divide(int a, int b) {

if (b == 0) throw new ArithmeticException("Cannot divide by zero."); return (double) a / b;

}

public int modulus(int a, int b) {

if (b == 0) throw new ArithmeticException("Cannot perform modulus by zero."); return a % b;

}

public double power(double base, double exponent) { return Math.*pow*(base, exponent);

}

public double squareRoot(double value) {

if (value < 0) throw new ArithmeticException("Cannot take square root of a negative number."); return Math.*sqrt*(value);

}

}

#### Create a JUnit Test Class

package com.example;

import org.junit.jupiter.api.Test;

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

class CalculatorTest {

Calculator calc = new Calculator();

@Test

void testAddition() {

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

}

@Test

void testSubtraction() {

*assertEquals*(1, calc.subtract(4, 3));

}

@Test

void testMultiplication() {

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

}

@Test

void testDivision() {

*assertEquals*(2.0, calc.divide(6, 3));

}

@Test

void testDivisionByZero() {

*assertThrows*(ArithmeticException.class, () -> calc.divide(5, 0));

}

@Test

void testModulus() {

*assertEquals*(1, calc.modulus(10, 3));

}

@Test

void testModulusByZero() {

*assertThrows*(ArithmeticException.class, () -> calc.modulus(10, 0));

}

@Test

void testPower() {

*assertEquals*(8.0, calc.power(2, 3));

}

@Test

void testSquareRoot() {

*assertEquals*(3.0, calc.squareRoot(9));

}

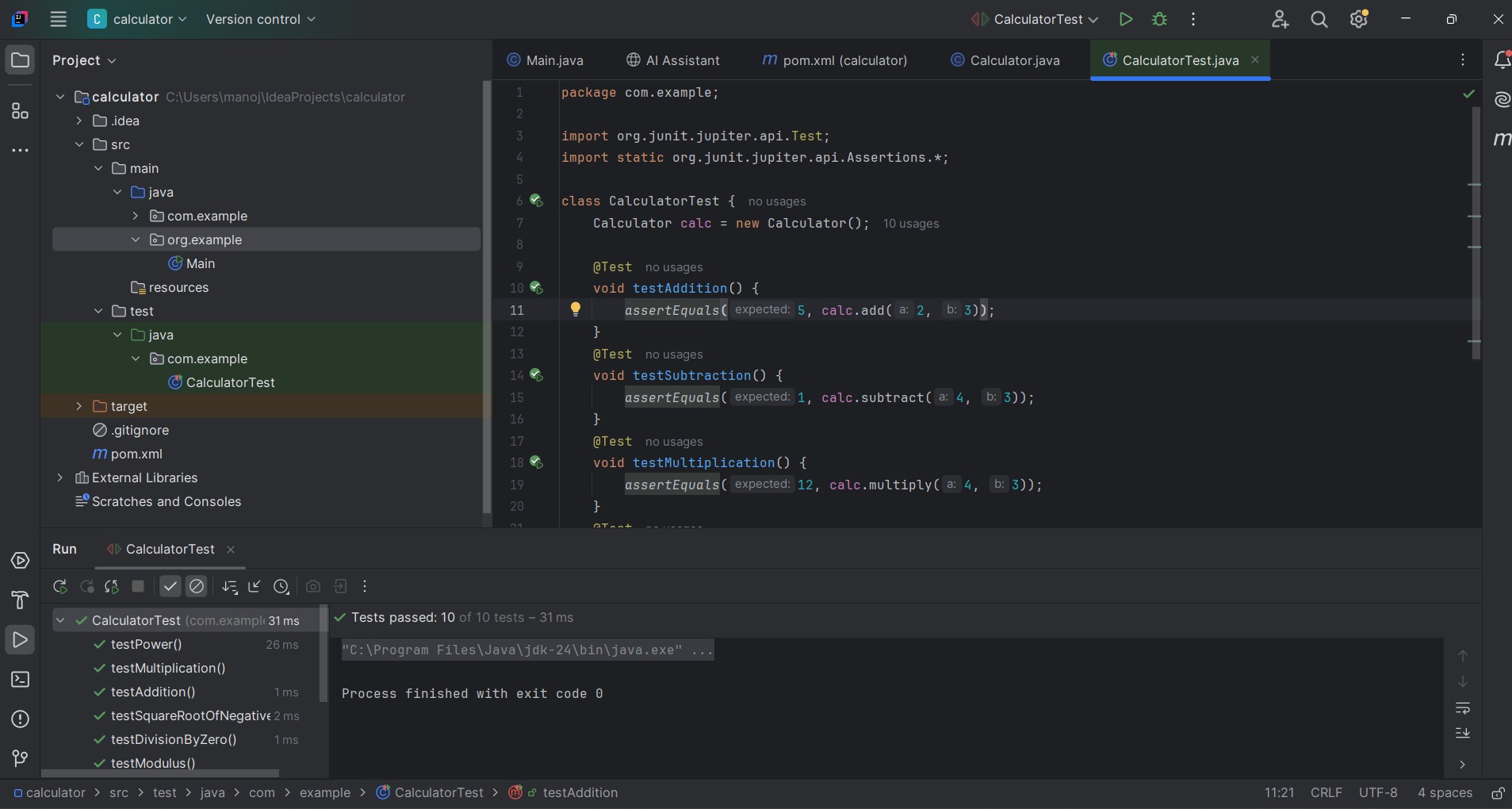
@Test

void testSquareRootOfNegative() {

*assertThrows*(ArithmeticException.class, () -> calc.squareRoot(-4));

}

}

**Output**

# Exercise – 3: Assertions in Junit

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

**Step 1:** Update the pom.xml file with Junit 5 in IntelliJ

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="<http://maven.apache.org/POM/4.0.0>"

xmlns:x[si="http://www.w3.org/2001/XMLSchema-instance"](http://www.w3.org/2001/XMLSchema-instance)

xsi:schemaLocation="<http://maven.apache.org/POM/4.0.0> [http://maven.apache.org/xsd/maven-4.0.0.xsd">](http://maven.apache.org/xsd/maven-4.0.0.xsd)

<modelVersion>4.0.0</modelVersion>

<groupId>org.example</groupId>

<artifactId>calculator</artifactId>

<version>1.0-SNAPSHOT</version>

<properties>

<maven.compiler.source>24</maven.compiler.source>

<maven.compiler.target>24</maven.compiler.target>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<!-- JUnit Jupiter -->

<dependency>

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

<artifactId>junit-jupiter</artifactId>

<version>5.10.2</version>

<scope>test</scope>

</dependency>

</dependencies>

</project>

**Step 2:** Create a test class named Assertion Test in src/test.java package com.example;

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

public class AssertionsTest { @Test

public void testAssertions() {

int a = 2;

int b = 3;

int sum = a + b;

*assertEquals*(5, sum);

int x = 5; int y = 3;

*assertTrue*(x > y);

*assertFalse*(y > x);

String nullString = null;

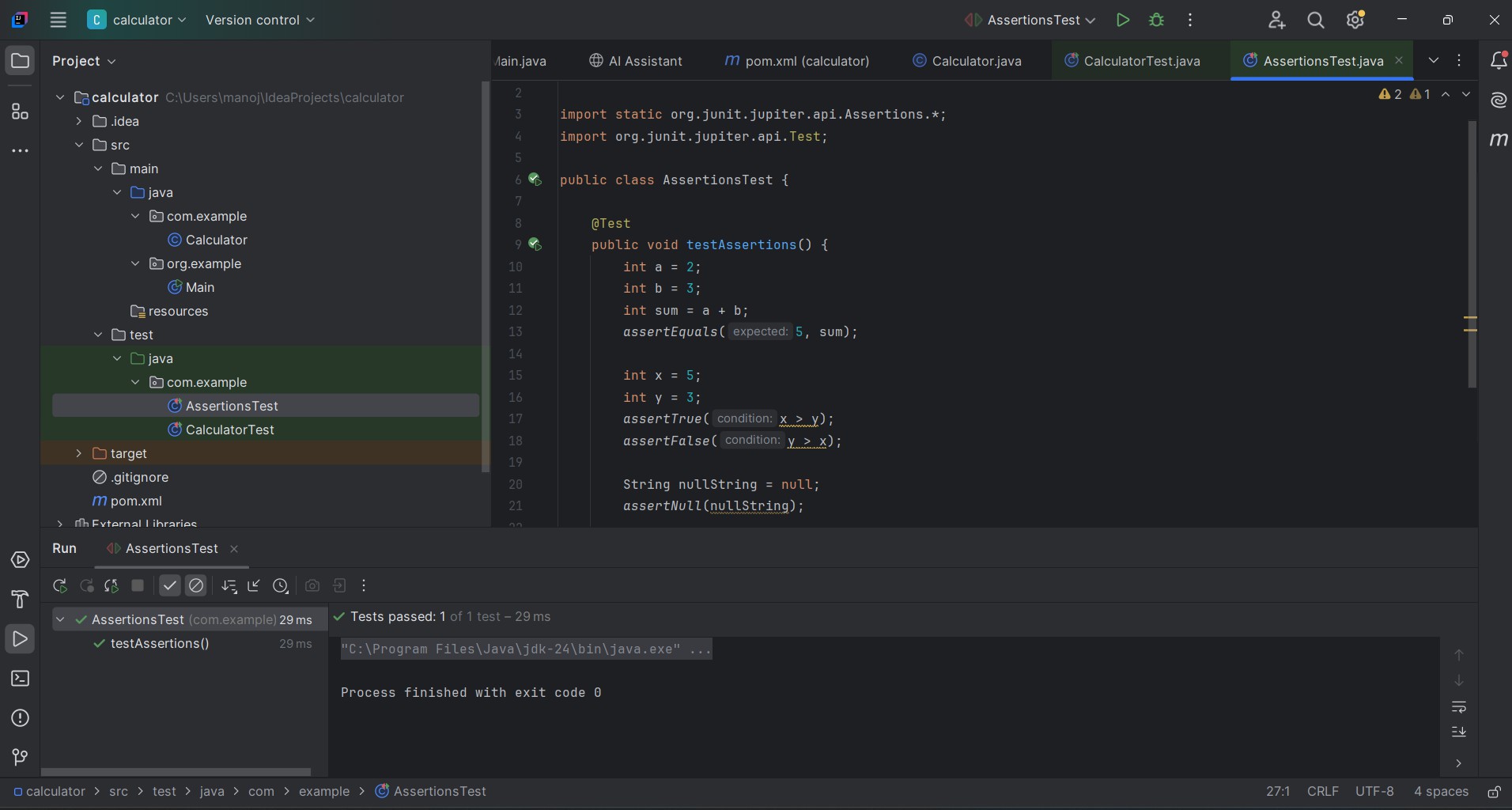
*assertNull*(nullString);

Object obj = new Object();

*assertNotNull*(obj);

}

}

**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

**Step 1. Create the class to test**

package com.example;

public class Calculator {

public int add(int a, int b) { return a + b;

}

}

**Step 2: Write test using AAA pattern, setup, and teardown**

package com.example;

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

public class CalculatorTest {

private Calculator calculator; @BeforeEach

public void setUp() {

calculator = new Calculator();

System.*out*.println("Setup: Calculator object created");

}

@AfterEach

public void tearDown() { calculator = null;

System.*out*.println("Teardown: Calculator object cleared");

}

@Test

public void testAddition() { int a = 5;

int b = 3;

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

*assertEquals*(8, result, "5 + 3 should equal 8");

}

@Test

public void testAdditionWithZero() { int a = 0;

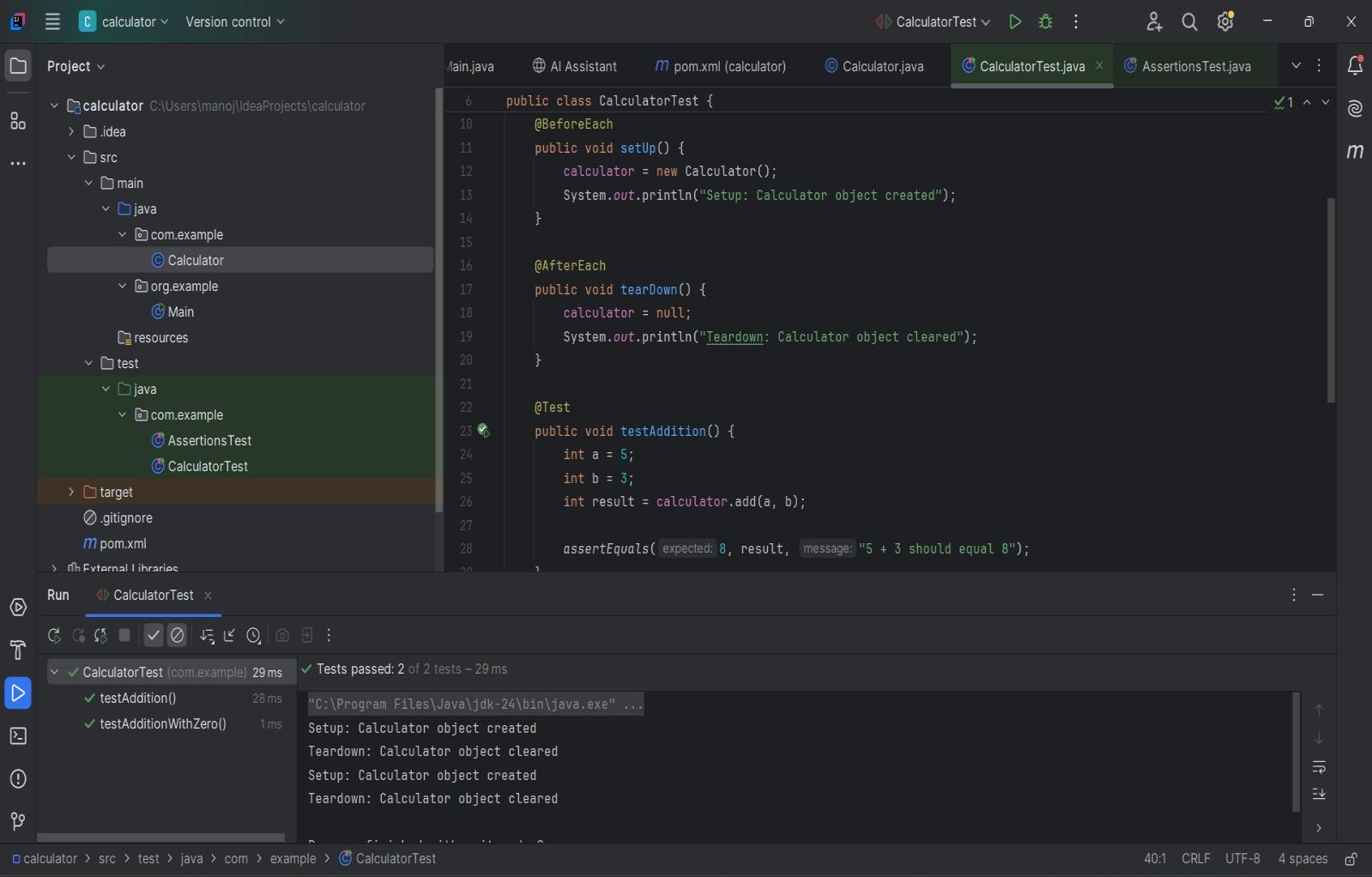
int b = 7;

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

*assertEquals*(7, result, "0 + 7 should equal 7");

}

}

**Output:**

# 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
4. **Create a Java Class for the External API and the Service**

## ExternalApi.java

package com.example;

public interface ExternalApi { String getData();

}

## MyService.java

package com.example;

public class MyService {

private final ExternalApi externalApi;

public MyService(ExternalApi externalApi) { this.externalApi = externalApi;

}

public String fetchData() { return externalApi.getData();

}

}

1. **Add Mockito and JUnit Dependencies to pom.xml**

<dependencies>

<!-- JUnit 5 -->

<dependency>

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

<artifactId>junit-jupiter</artifactId>

<version>5.10.2</version>

<scope>test</scope>

</dependency>

<!-- Mockito -->

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>5.11.0</version>

<scope>test</scope>

</dependency>

</dependencies>

1. **Create the JUnit Test Class using Mocking and Stubbing**

package com.example;

import org.junit.jupiter.api.Test; import org.mockito.Mockito;

import static org.junit.jupiter.api.Assertions.*assertEquals*; import static org.mockito.Mockito.*when*;

public class MyServiceTest { @Test

public void testExternalApi() {

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

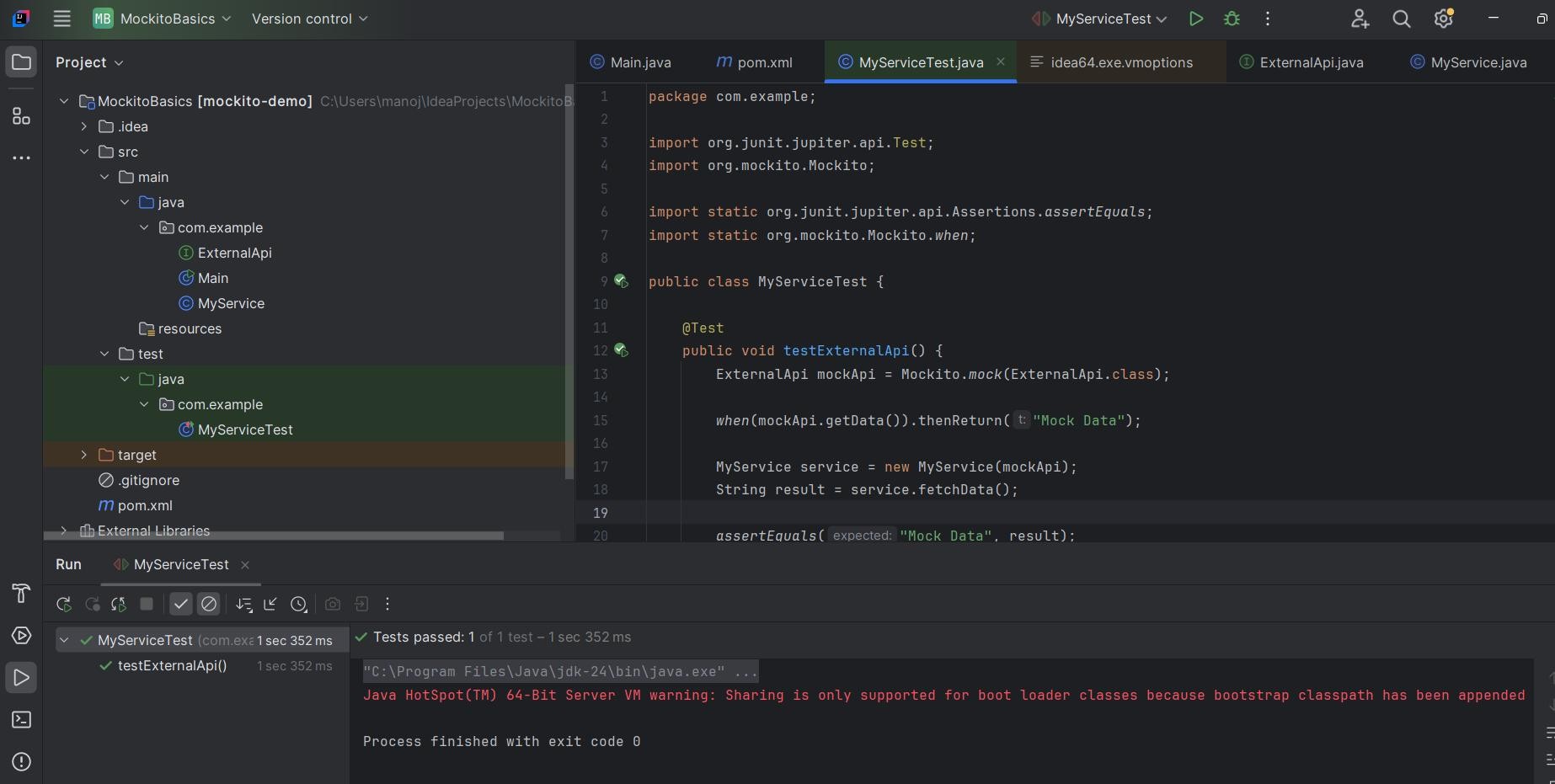
*when*(mockApi.getData()).thenReturn("Mock Data"); MyService service = new MyService(mockApi);

String result = service.fetchData();

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

}

}

**Output:**

**Scenario:**

# Exercise – 2: Verifying Interactions

You need to ensure that a method is called with specific arguments using **Mockito**.

**Steps:**

#### Create a mock object

1. **Call the method** with specific arguments
2. **Verify the interaction**
3. **Java Classes Setup**

## ExternalApi.java

package com.example;

public interface ExternalApi { String getData();

}

## MyService.java

package com.example;

public class MyService {

private final ExternalApi externalApi;

public MyService(ExternalApi externalApi) { this.externalApi = externalApi;

}

public String fetchData() { return externalApi.getData();

}

}

1. **Add Mockito and JUnit Dependencies in pom.xml**

<dependencies>

<!-- JUnit 5 -->

<dependency>

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

<artifactId>junit-jupiter</artifactId>

<version>5.10.2</version>

<scope>test</scope>

</dependency>

<!-- Mockito -->

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>5.11.0</version>

<scope>test</scope>

</dependency>

</dependencies>

1. **Test Class for Verifying Interaction**

package com.example;

import org.junit.jupiter.api.Test; import static org.mockito.Mockito.\*;

public class MyServiceInteractionTest { @Test

public void testVerifyInteraction() {

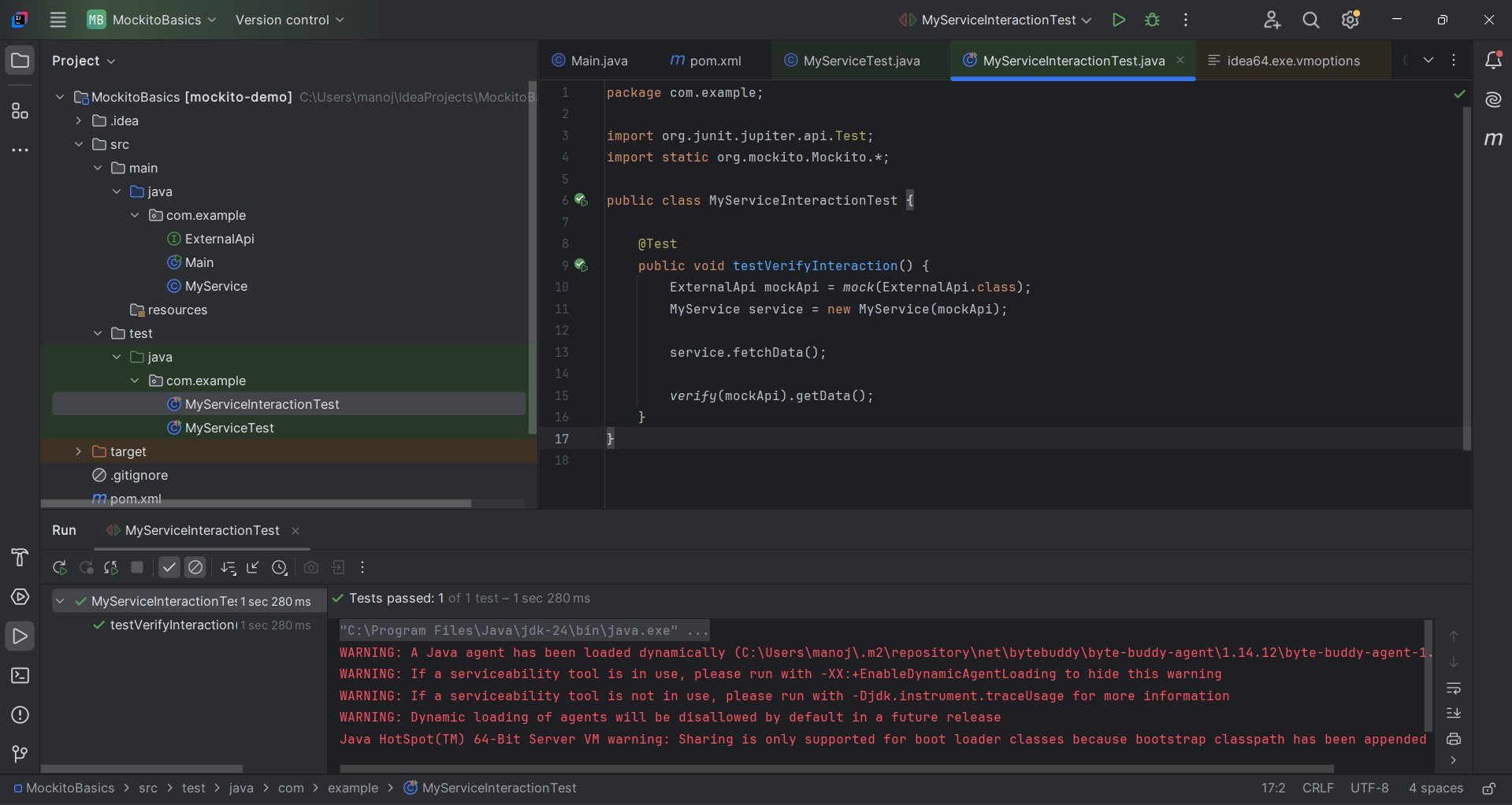
ExternalApi mockApi = *mock*(ExternalApi.class); MyService service = new MyService(mockApi);

service.fetchData();

*verify*(mockApi).getData();

}

}

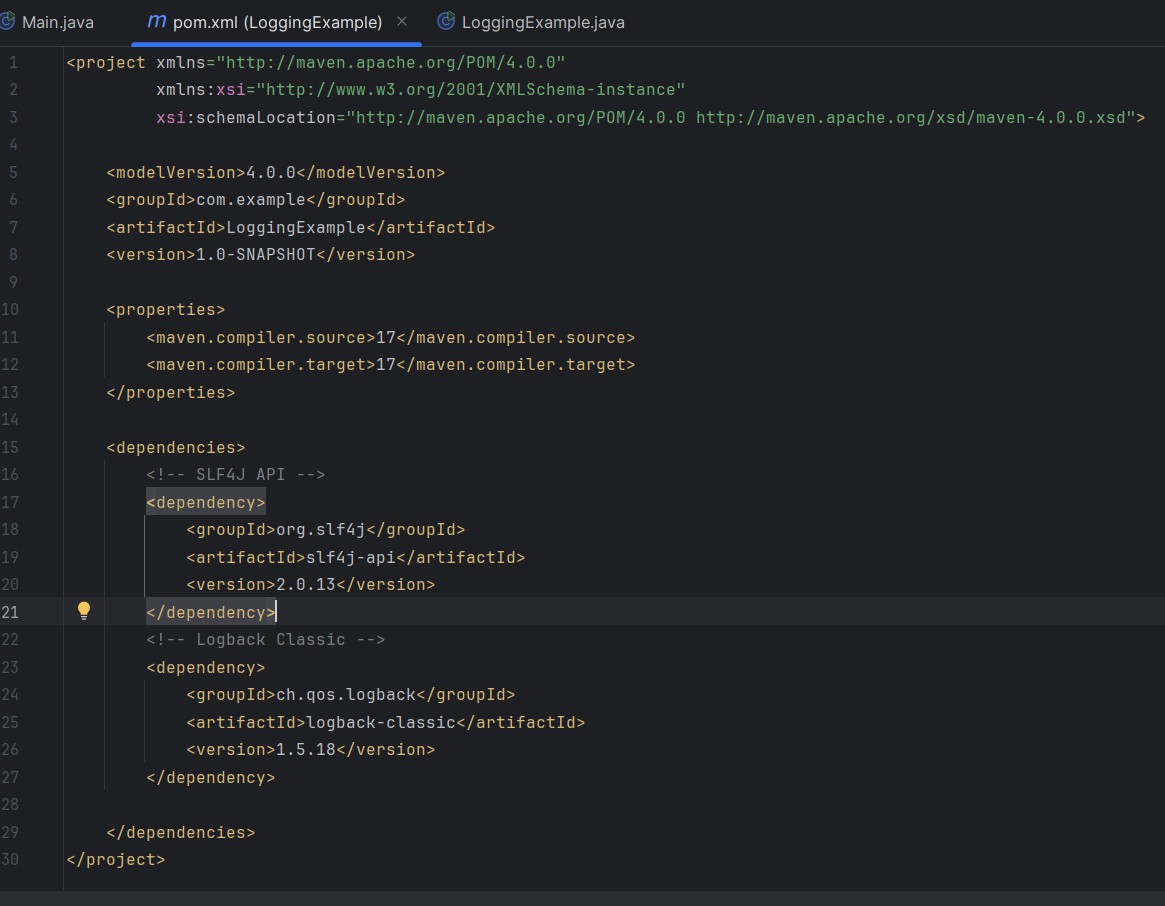
**Output:**

# Exercise – 6

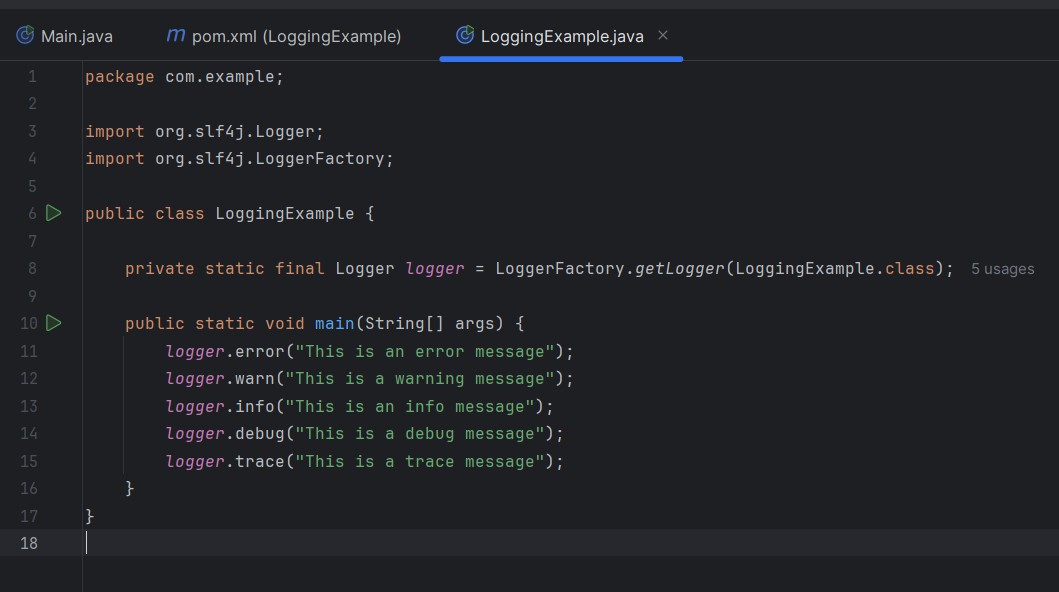
**SLF4J Logging — Logging Error Messages and Warning Levels**

**Scenario:** You need to demonstrate logging in a Java application using the **SLF4J (Simple Logging Facade for Java)** with **Logback** as the implementation backend.

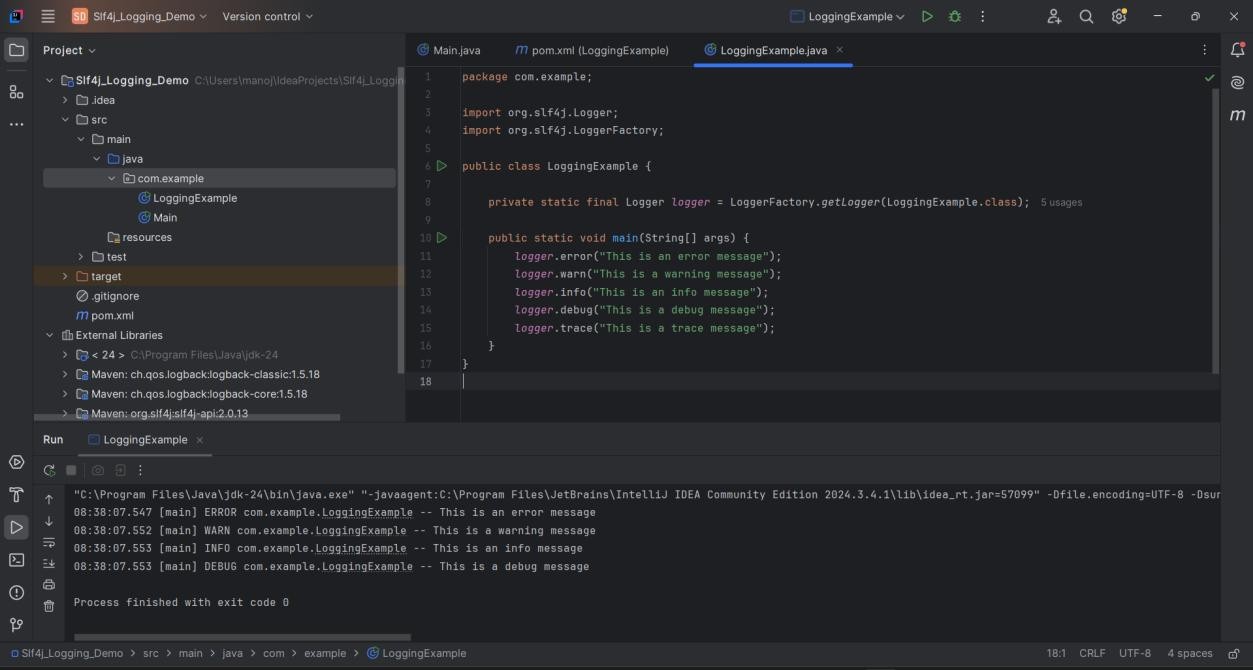
**Step 1: Add SLF4J and Logback dependencies to pom.xml**

****

**Step 2: Create the Logging Class**

****

**Output:**

****