**WEEK – 2**

**PL/SQL programming - PLSQL\_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.

**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.

**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.

**APPROACH**

* Create customers and loans tables to store customer details, loan information, and fields like age, balance, interest rate, and due date.
* Insert sample data into both tables to simulate real-world records for testing the logic.
* Use UPDATE queries with WHERE conditions to:
  + Apply a 1% interest discount for customers above 60 years old.
  + Set the IsVIP flag for customers with balance over $10,000.
* Use a SELECT query with JOIN to fetch loan records due within the next 30 days and display reminder messages for each customer.
* Execute all SQL statements in One Compiler (SQL Server) and view the output and changes directly in the result pane for validation.

**CODE**

**Create Tables**

**--**Customers

CREATE TABLE customers (

customer\_id INT,

name VARCHAR(100),

age INT,

balance DECIMAL(10, 2),

is\_vip BIT,

interest\_rate DECIMAL(4, 2)

);

--Loans

CREATE TABLE loans (

loan\_id INT,

customer\_id INT,

due\_date DATE,

loan\_amount DECIMAL(10, 2)

);

**Insert Sample Data**

INSERT INTO customers VALUES (1, 'John Doe', 65, 5000.00, 0, 6.5);

INSERT INTO customers VALUES (2, 'Jane Smith', 45, 12000.00, 0, 7.2);

INSERT INTO customers VALUES (3, 'Alex Brown', 70, 8000.00, 0, 8.0);

SELECT \* FROM customers;

INSERT INTO loans VALUES (101, 1, GETDATE() + 10, 20000.00);

INSERT INTO loans VALUES (102, 2, GETDATE() + 40, 15000.00);

INSERT INTO loans VALUES (103, 3, GETDATE() + 5, 30000.00);

**Scenario 1 - Apply 1% Discount to Interest Rate for Customers Aged Above 60**

PRINT 'Scenario 1: Interest‑rate discount for customers over 60 ';

UPDATE customers

SET interest\_rate = interest\_rate - 1

WHERE age > 60;

//View Updated Customer Data

SELECT customer\_id, name, age, interest\_rate

FROM customers

WHERE age > 60;

**Scenario 2 - Set IsVIP Flag to True for Customers with Balance > 10000**

PRINT 'Scenario 2: Customers promoted to VIP (balance>10 000) ';

UPDATE customers

SET is\_vip = 1

WHERE balance > 10000;

//View Updated Customer Data

SELECT customer\_id, name, balance, is\_vip

FROM customers

WHERE is\_vip = 1;

**Scenario 3 - Print Loan Reminders Due Within Next 30 Days**

PRINT 'Scenario 3: Loan reminders (due ≤ 30 days) ';

SELECT c.name AS Customer\_Name,

l.loan\_id AS Loan\_ID,

l.due\_date AS Due\_Date,

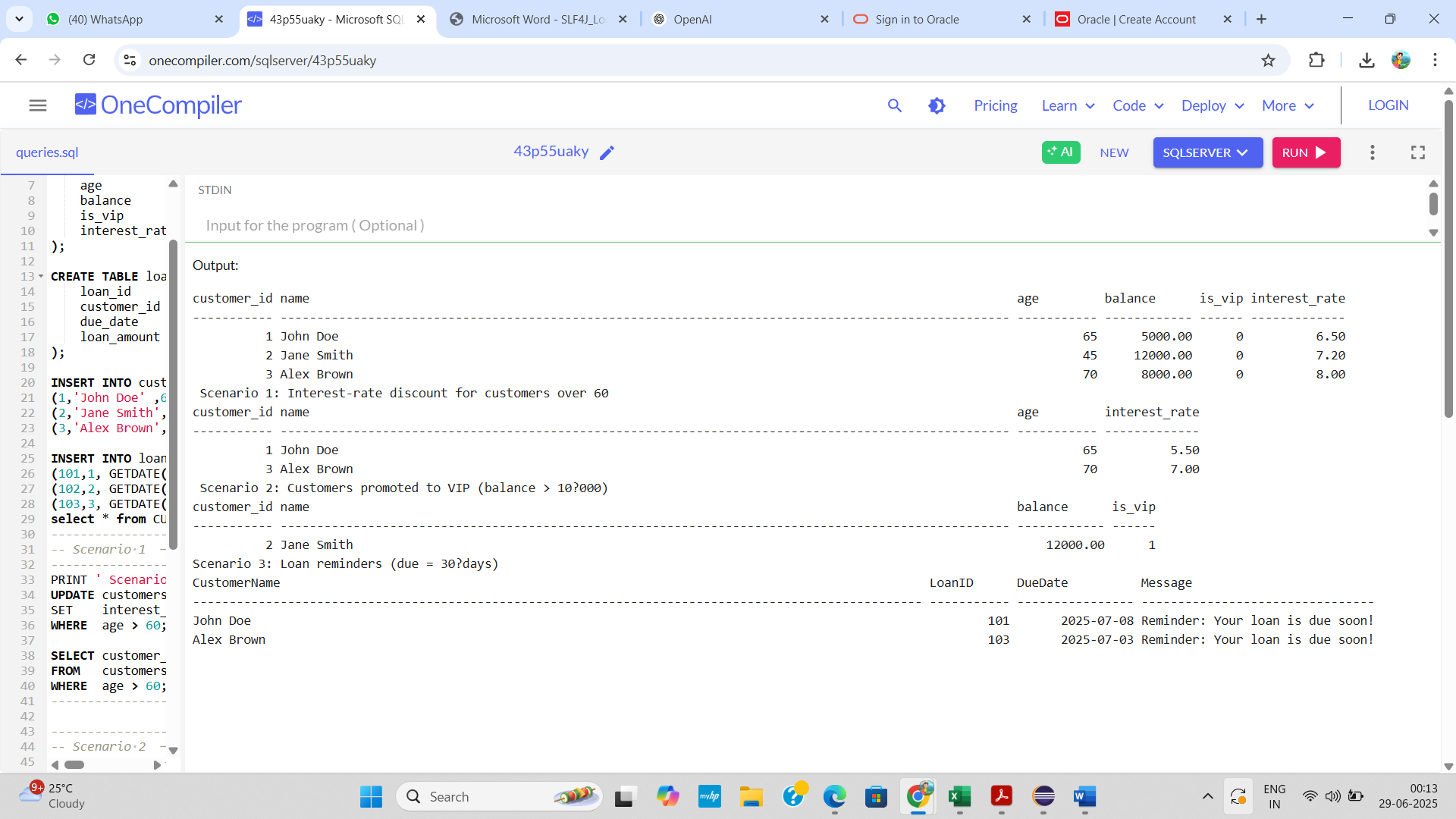
'Reminder: Your loan is due soon!' AS Message

FROM loans l

JOIN customers c ON c.customer\_id = l.customer\_id

WHERE l.due\_date <= GETDATE() + 30;

**OUTPUT**



**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.

**APPROACH**

* Create necessary tables and insert sample data for SavingsAccounts, Employees, and BankAccounts to simulate realistic banking scenarios like interest processing, bonuses, and fund transfers.
* Define ProcessMonthlyInterest stored procedure to apply a 1% monthly interest to all savings account balances by updating the SavingsAccounts table.
* Define UpdateEmployeeBonus stored procedure with parameters for department ID and bonus percentage, and update the Employees table by increasing salaries accordingly.
* Define TransferFunds stored procedure to safely move money between two bank accounts, with checks for sufficient funds and atomic operations using transactions (BEGIN TRAN, COMMIT, ROLLBACK).
* Execute each procedure and verify results using SELECT queries and PRINT statements to confirm the correctness of updates after each operation.

**CODE**

**Create tables**

-- Savings accounts

CREATE TABLE SavingsAccounts (

account\_id INT PRIMARY KEY,

customer\_id INT,

balance DECIMAL(12,2)

);

-- Employees

CREATE TABLE Employees (

emp\_id INT PRIMARY KEY,

emp\_name VARCHAR(50),

dept\_id INT,

salary DECIMAL(12,2)

);

-- Generic accounts for transfers

CREATE TABLE BankAccounts (

account\_id INT PRIMARY KEY,

customer\_id INT,

balance DECIMAL(12,2)

);

**Insert Sample Data**

INSERT INTO SavingsAccounts VALUES (101, 1, 1000.00), (102, 2, 2500.00);

INSERT INTO Employees VALUES (1,'Alice', 10, 60000.00),

(2,'Bob', 10, 55000.00),

(3,'Cara', 20, 50000.00);

INSERT INTO BankAccounts VALUES (201, 1, 5000.00), (202, 1, 2000.00);

**Scenario 1 ─ Stored procedure to process monthly interest**

GO

CREATE OR ALTER PROCEDURE ProcessMonthlyInterest

AS

BEGIN

SET NOCOUNT ON;

UPDATE SavingsAccounts

SET balance = balance \* 1.01; -- +1 %

PRINT 'Scenario-1 Monthly interest applied to all savings accounts.';

END;

GO

EXEC ProcessMonthlyInterest;

PRINT '--- Savings account balances after interest ---';

SELECT \* FROM SavingsAccounts;

**Scenario 2 ─ Stored procedure to apply bonus to a department**

GO

CREATE OR ALTER PROCEDURE UpdateEmployeeBonus

@DeptId INT,

@BonusPct DECIMAL(5,4) -- 0.0500 = 5 %

AS

BEGIN

SET NOCOUNT ON;

UPDATE Employees

SET salary = salary \* (1 + @BonusPct)

WHERE dept\_id = @DeptId;

PRINT CONCAT('Scenario - 2 Bonus of ', @BonusPct\*100, '% applied to department ', @DeptId);

END;

GO

EXEC UpdateEmployeeBonus @DeptId = 10, @BonusPct = 0.10; -- 10 % bonus

PRINT '--- Employee salaries after bonus ---';

SELECT \* FROM Employees;

**Scenario 3 ─ Stored procedure for fund transfer**

GO

CREATE OR ALTER PROCEDURE TransferFunds

@SrcAcct INT,

@DestAcct INT,

@Amount DECIMAL(12,2)

AS

BEGIN

SET NOCOUNT ON;

BEGIN TRAN;

DECLARE @SrcBal DECIMAL(12,2);

SELECT @SrcBal = balance FROM BankAccounts WHERE account\_id = @SrcAcct;

IF @SrcBal IS NULL

BEGIN

ROLLBACK;

RAISERROR ('Source account not found.', 16, 1);

RETURN;

END

IF @SrcBal < @Amount

BEGIN

ROLLBACK;

RAISERROR ('Insufficient funds in source account.', 16, 1);

RETURN;

END

UPDATE BankAccounts SET balance = balance - @Amount WHERE account\_id = @SrcAcct;

UPDATE BankAccounts SET balance = balance + @Amount WHERE account\_id = @DestAcct;

COMMIT;

PRINT CONCAT('Scenario-3 Transferred $', @Amount, ' from account ', @SrcAcct, ' to ', @DestAcct);

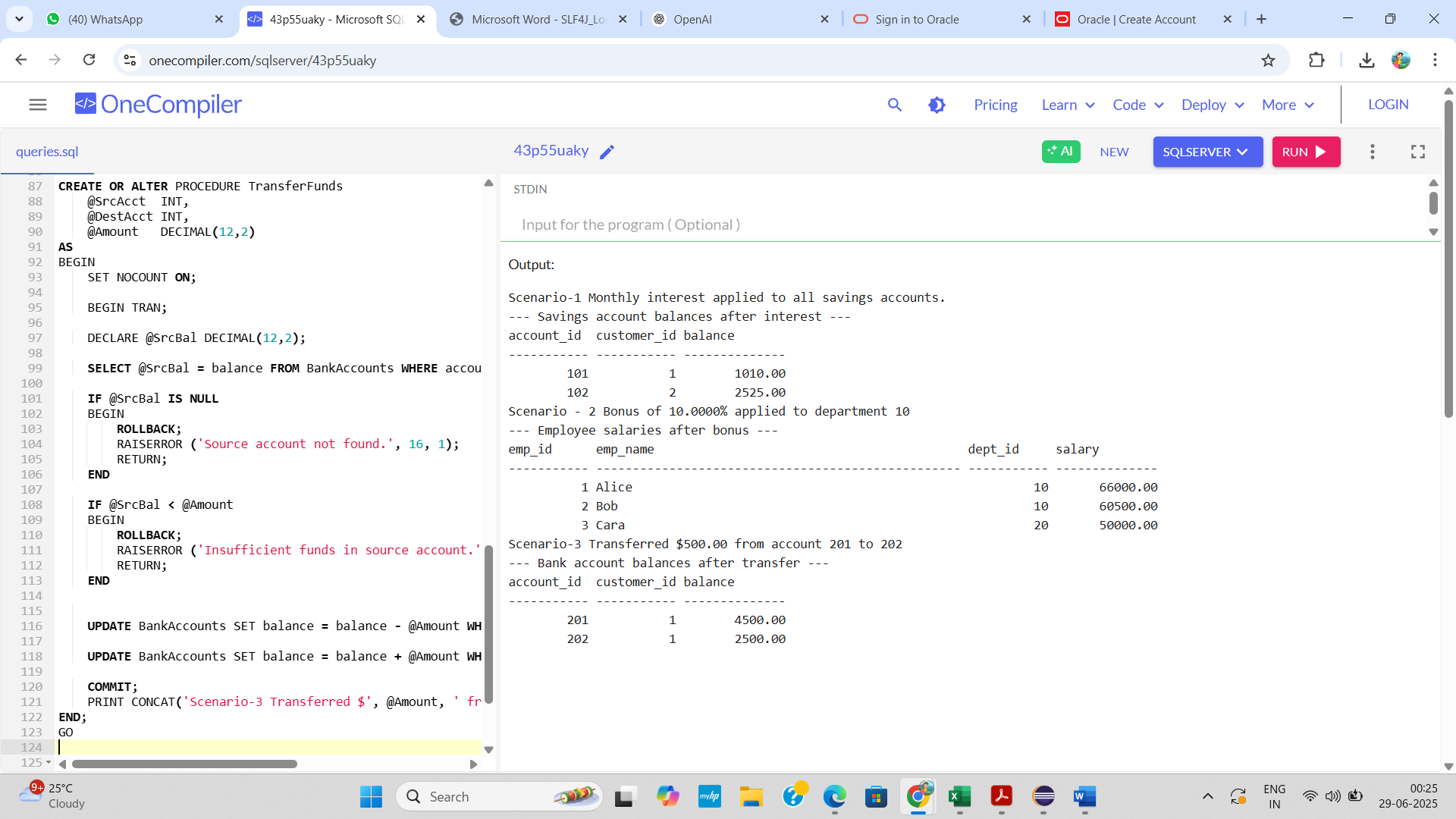
END;

GO

PRINT '--- Bank account balances after transfer ---';

SELECT \* FROM BankAccounts;

**OUTPUT**



**TDD USING JUNIT5 AND MOCKITO**

**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.

**APPROACH**

* Created a Maven project named primechecker in Eclipse with proper folder structure.
* Added PrimeUtils.java for prime-checking logic and Main.java for user input/output in the com.example package.
* Included JUnit 4.13.2 dependency in pom.xml for unit testing support.
* Wrote PrimeUtilsTest.java under src/test/java to validate different prime and non-prime cases.
* Executed both the main program and JUnit tests successfully in Eclipse with correct output.

**CODE**

**Main.java**

package com.example;

import java.util.Scanner;

public class Main {

public static void main(String[] args) {

PrimeUtils utils = new PrimeUtils();

Scanner scanner = new Scanner(System.in);

System.out.print("Enter a number to check if it is prime: ");

int number = scanner.nextInt();

boolean result = utils.isPrime(number);

if (result) {System.out.println(number + " is a prime number."); }

else {System.out.println(number + " is NOT a prime number."); }

}}

**PrimeUtils.java**

package com.example;

public class PrimeUtils {

public boolean isPrime(int n) {

if (n < 2) return false;

if (n == 2) return true;

if (n % 2 == 0) return false;

for (int i = 3; i \* i <= n; i += 2) {

if (n % i == 0) return false;

}

return true;

}

}

**PrimeUtilsTest.java**

package com.example;

import org.junit.Test;

import com.example.PrimeUtils;

import static org.junit.Assert.\*;

public class PrimeUtilsTest {

private final PrimeUtils utils = new PrimeUtils();

@Test

public void testSmallPrimes() {

assertTrue(utils.isPrime(2));

assertTrue(utils.isPrime(3));

assertTrue(utils.isPrime(5));

assertTrue(utils.isPrime(7));

}

@Test

public void testSmallComposites() {

assertFalse(utils.isPrime(0));

assertFalse(utils.isPrime(1));

assertFalse(utils.isPrime(4));

assertFalse(utils.isPrime(6));

assertFalse(utils.isPrime(9));

}

@Test

public void testLargeNumbers() {

assertTrue(utils.isPrime(97));

assertFalse(utils.isPrime(100));

}

@Test

public void testNegativeNumbers() {

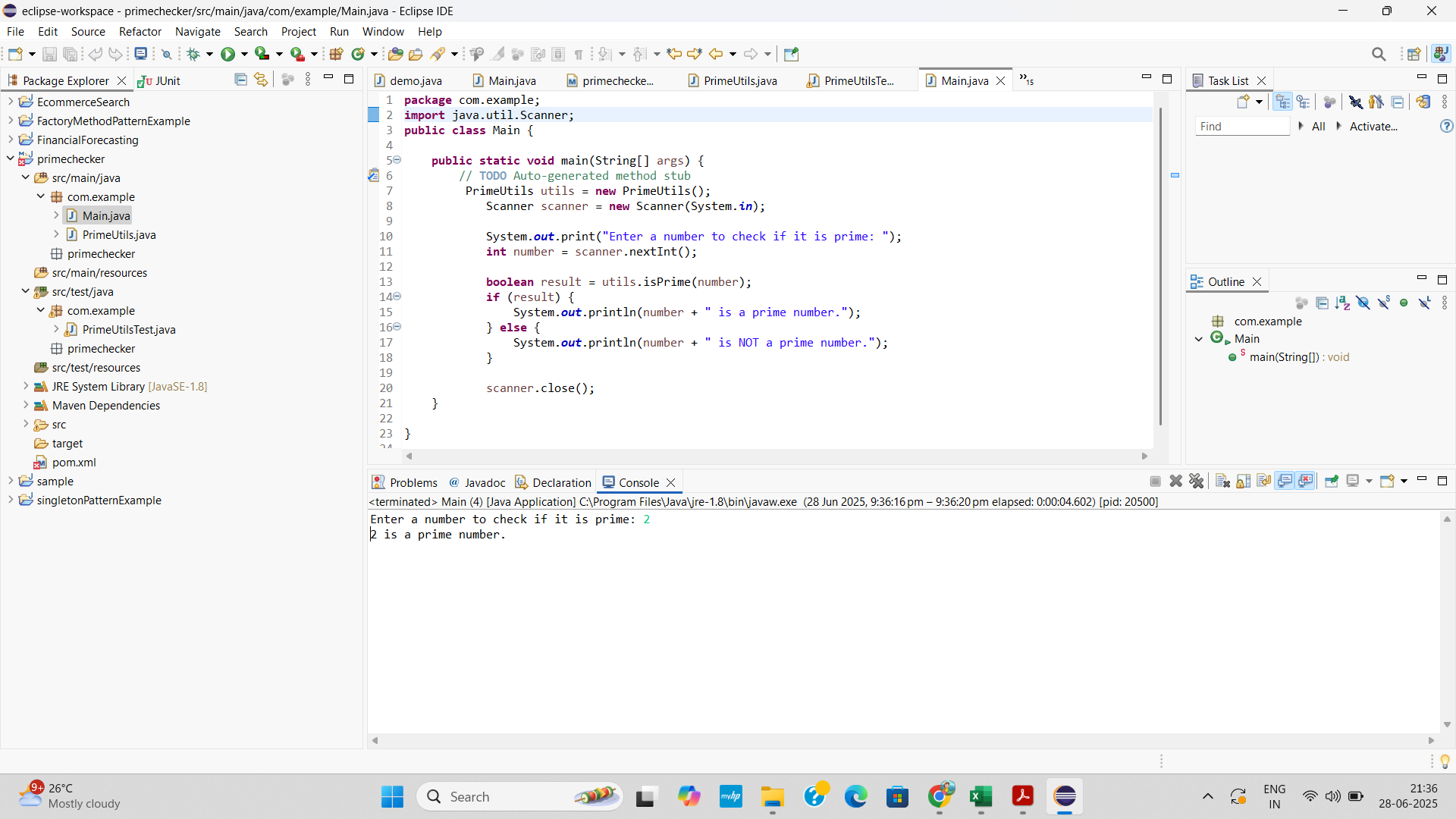
assertFalse(utils.isPrime(-11));

assertFalse(utils.isPrime(-2));

}

}

**OUTPUT**

****

**EXERCISE 3: ASSERTIONS IN JUNIT**

**SCENARIO**

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

**APPROACH**

* Write a test class (AssertionsTest) using @Test methods with assertions like assertEquals, assertTrue, etc.
* Place the test class in the src/test/java folder under the appropriate package (e.g., com.example).
* Add JUnit 4.13.2 dependency in pom.xml and update the Maven project to download necessary libraries.
* Right-click the test class and choose Run As → JUnit Test to execute the test.
* View the result in the JUnit tab (green bar = pass, red bar = fail).

**CODE**

**AssertionsTest.java**

package com.example;

import static org.junit.Assert.\*;

import org.junit.Test;

public class AssertionsTest {

public void testAssertions() {

@Test

assertEquals(5, 2 + 3);

assertTrue(5 > 3);

assertFalse(5 < 3);

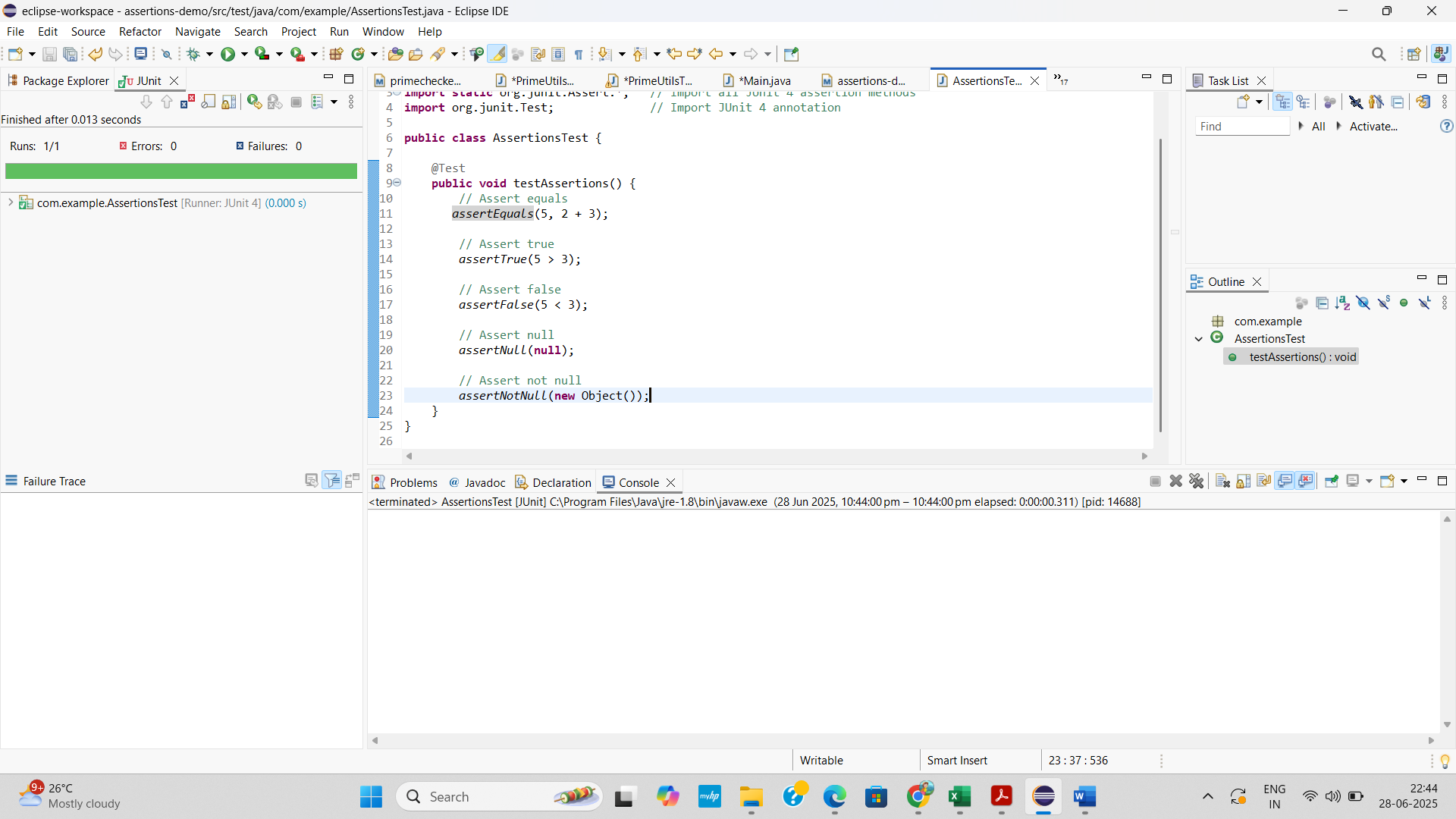
assertNull(null);

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.

**APPROACH**

* Create a class OrderProcessor with methods placeOrder(), cancelOrder(), and isOrderPlaced() to simulate order handling logic.
* Write a test class OrderProcessorTest in src/test/java using JUnit 4 and organize each test with the Arrange–Act–Assert structure for clarity.
* Use @Before annotation to initialize the OrderProcessor object before each test method runs.
* Use @After annotation to reset or destroy the test object after each test method, ensuring a clean test environment.
* Run the test class using “Run As → JUnit Test” in Eclipse and verify the results in the JUnit view (green bar for pass, red for failure) and setup/teardown messages in the Console.

**CODE**

**OrderProcessor.java**

package com.example;

public class OrderProcessor {

private boolean orderPlaced;

public boolean placeOrder() {

orderPlaced = true;

return orderPlaced;

}

public boolean cancelOrder() {

if (orderPlaced) {

orderPlaced = false;

return true;

}

else {

return false;

}

}

public boolean isOrderPlaced() {

return orderPlaced;

}

}

**OrderProcessorTest.java**

package com.example;

import static org.junit.Assert.\*;

import org.junit.Before;

import org.junit.After;

import org.junit.Test;

public class OrderProcessorTest {

private OrderProcessor orderProcessor;

@Before

public void setUp() {

orderProcessor = new OrderProcessor();

System.out.println("Setup: New OrderProcessor created.");

}

@After

public void tearDown() {

orderProcessor = null;

System.out.println("Teardown: OrderProcessor destroyed.");

}

@Test

public void testPlaceOrder() {

boolean result = orderProcessor.placeOrder();

assertTrue(result);

assertTrue(orderProcessor.isOrderPlaced());

}

@Test

public void testCancelOrder() {

orderProcessor.placeOrder();

boolean result = orderProcessor.cancelOrder();

assertTrue(result);

assertFalse(orderProcessor.isOrderPlaced());

}

@Test

public void testCancelWithoutPlacing() {

boolean result = orderProcessor.cancelOrder();

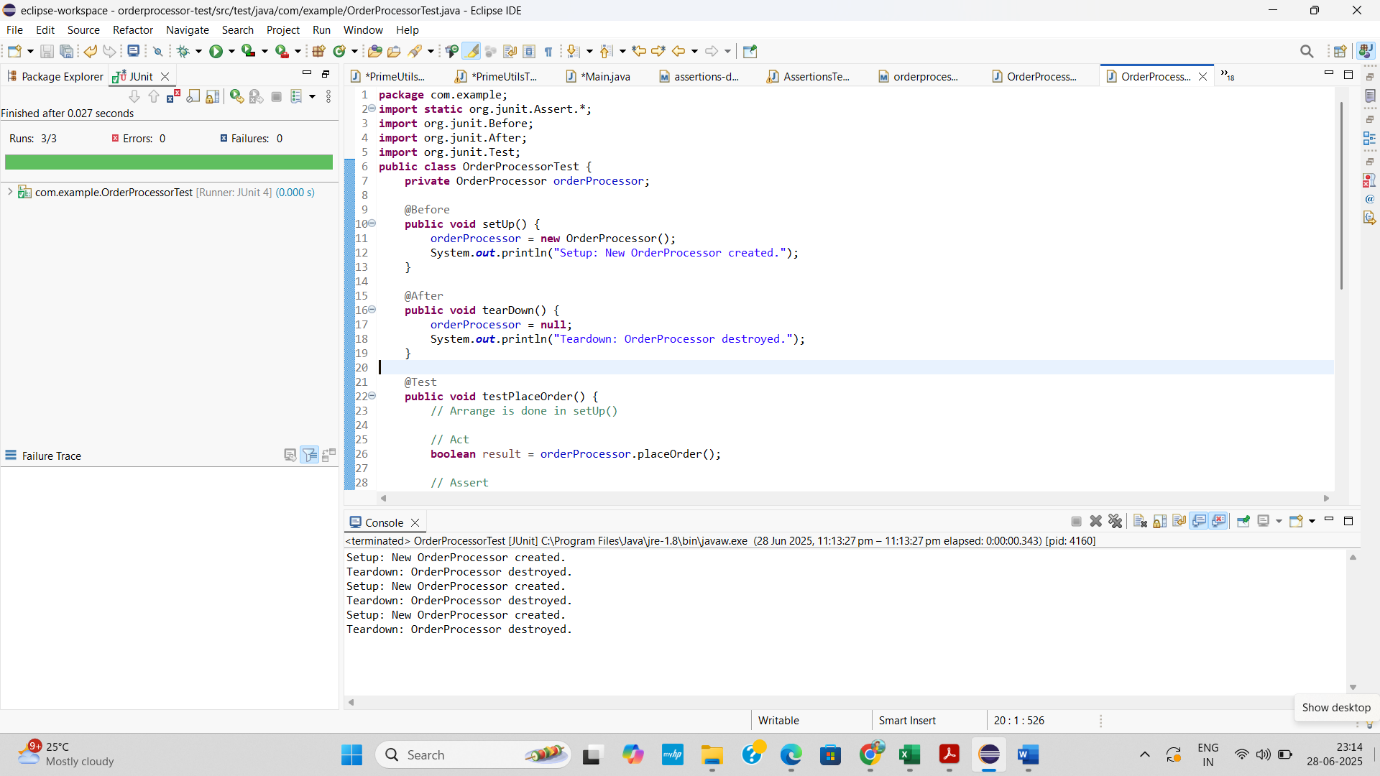
assertFalse(result);

assertFalse(orderProcessor.isOrderPlaced());

}

}

**OUTPUT**

****

**TDD USING JUNIT5 AND MOCKITO - 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.

**APPROACH**

* Create a Maven project in Eclipse and add dependencies for JUnit 5 and a Java 8–compatible version of Mockito (e.g., 3.12.4) in pom.xml.
* Define an interface ExternalApi and a service class MyService that depends on it via constructor injection, using a method like getData().
* Create a test class MyServiceTest in src/test/java using Mockito to mock ExternalApi and stub its method to return a predefined value.
* Use JUnit assertions in the test to verify that the service behaves correctly when the mocked API returns the stubbed value.
* Run the test as a JUnit test in Eclipse and ensure it passes, confirming that mocking and stubbing were successful without real API calls.

**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();

System.out.println("Fetched result from mocked API: " + result);

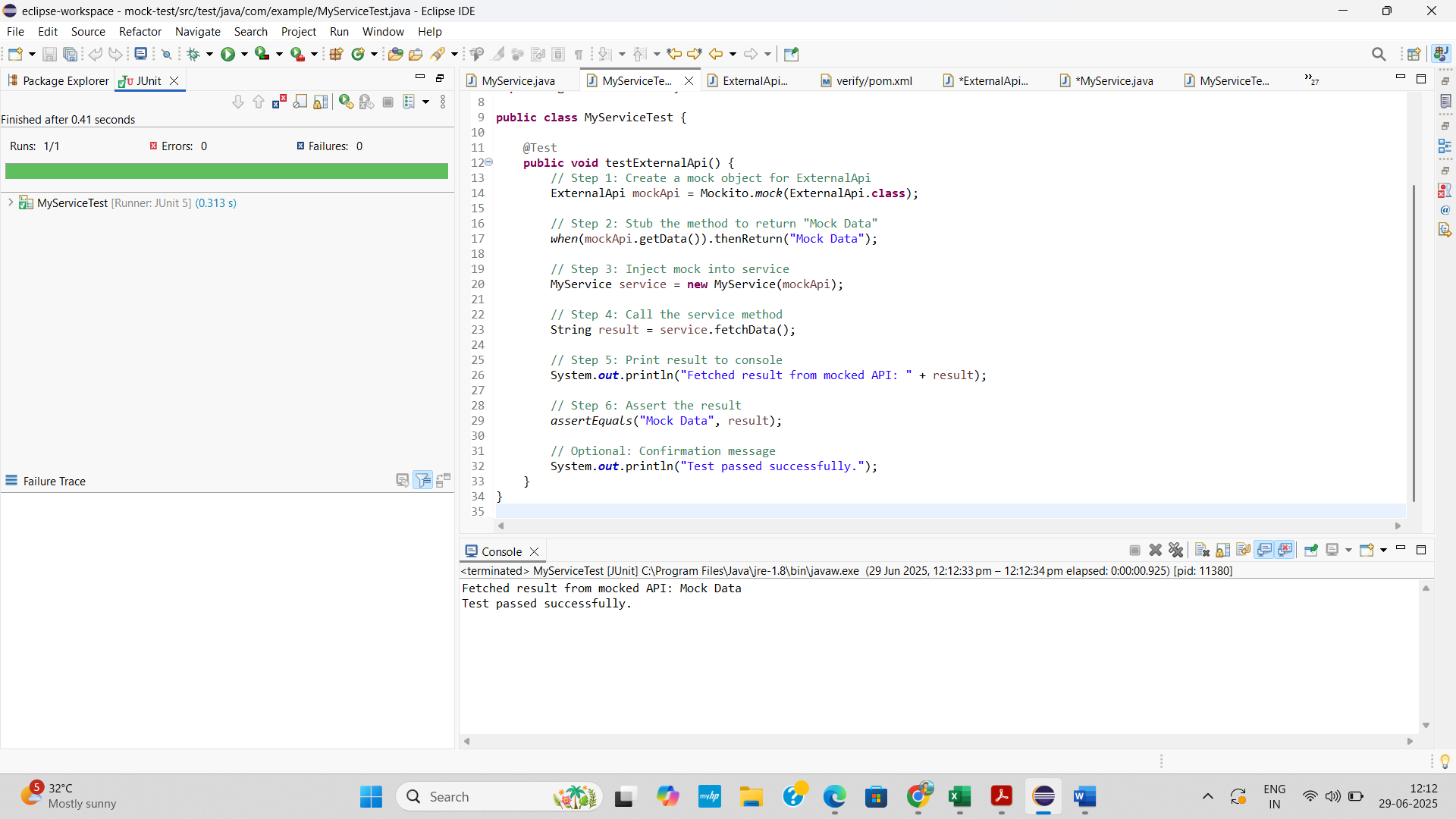
assertEquals("Mock Data", result);

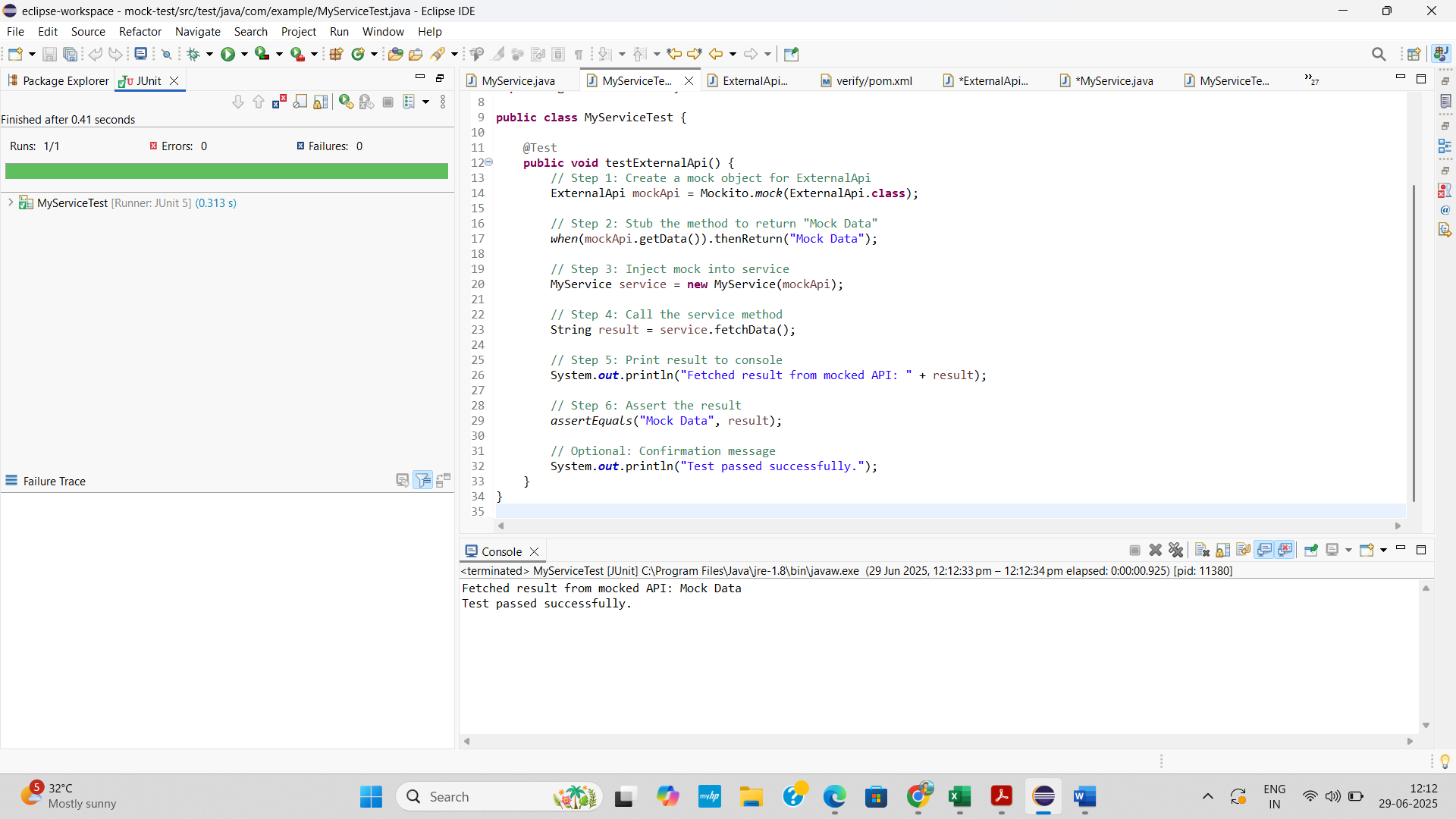
System.out.println("Test passed successfully.");

}

}

**OUTPUT**





**EXERCISE 2: VERIFYING INTERACTIONS**

**SCENARIO**

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

**APPROACH**

* Create a mock object using Mockito.mock() to simulate the external dependency (ExternalApi).
* Stub the method with when(...).thenReturn(...) so it returns a predictable value (e.g., "Mocked Response").
* Call the method using the real class (MyService) that depends on the mock.
* Print the returned value to the console using System.out.println() to confirm the method executed.
* Verify the interaction using verify(mock).method() and print a confirmation message that the method was successfully called.

**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) {

this.api = api;

}

public String fetchData() {

return api.getData();

}}

**MyServiceTest.java**

package com.example;

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

public class MyServiceTest {

@Test

public void testVerifyInteraction() {

ExternalApi mockApi = mock(ExternalApi.class);

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

MyService service = new MyService(mockApi);

String result = service.fetchData();

System.out.println("Fetched result: " + result);

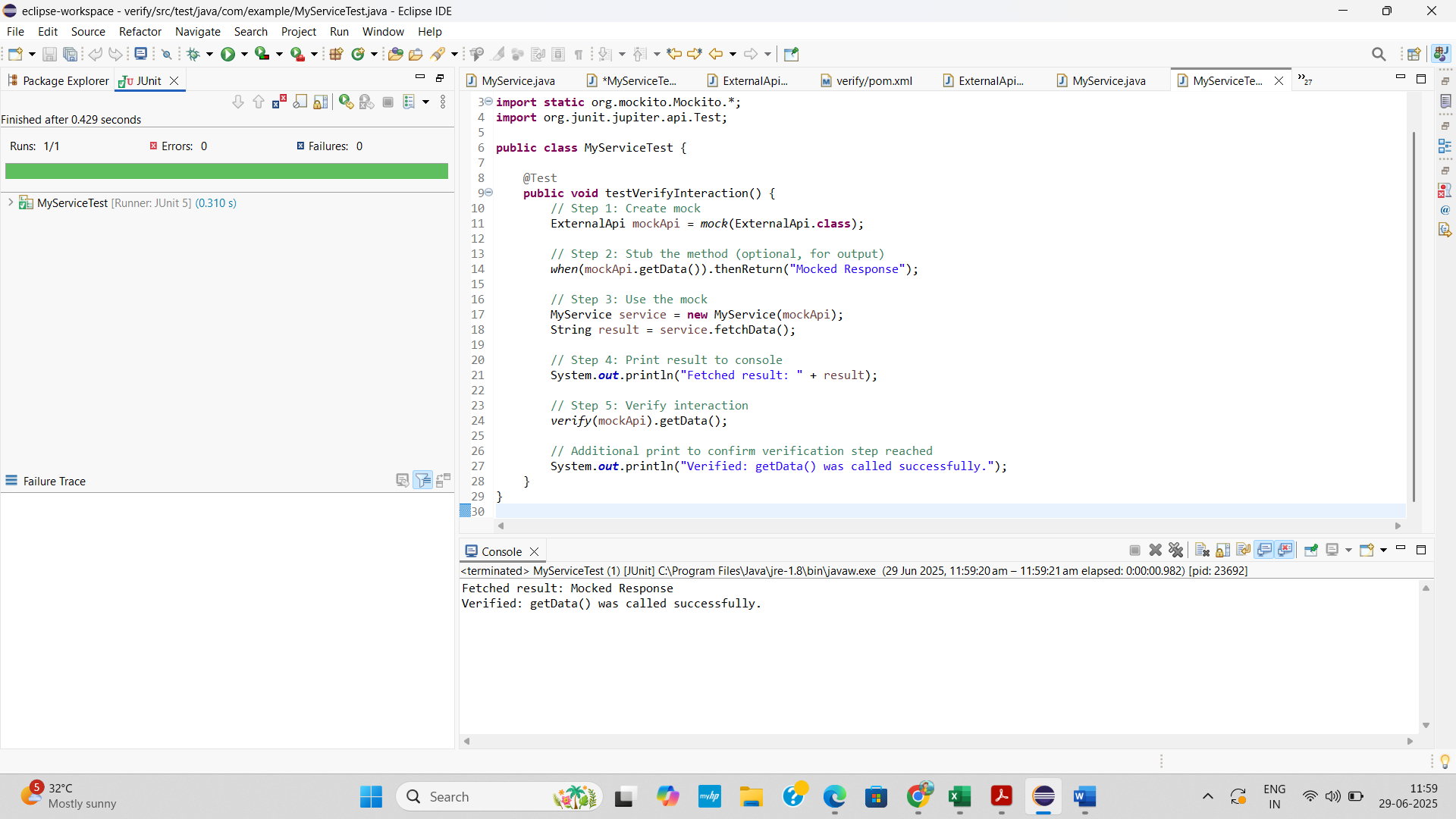
verify(mockApi).getData();

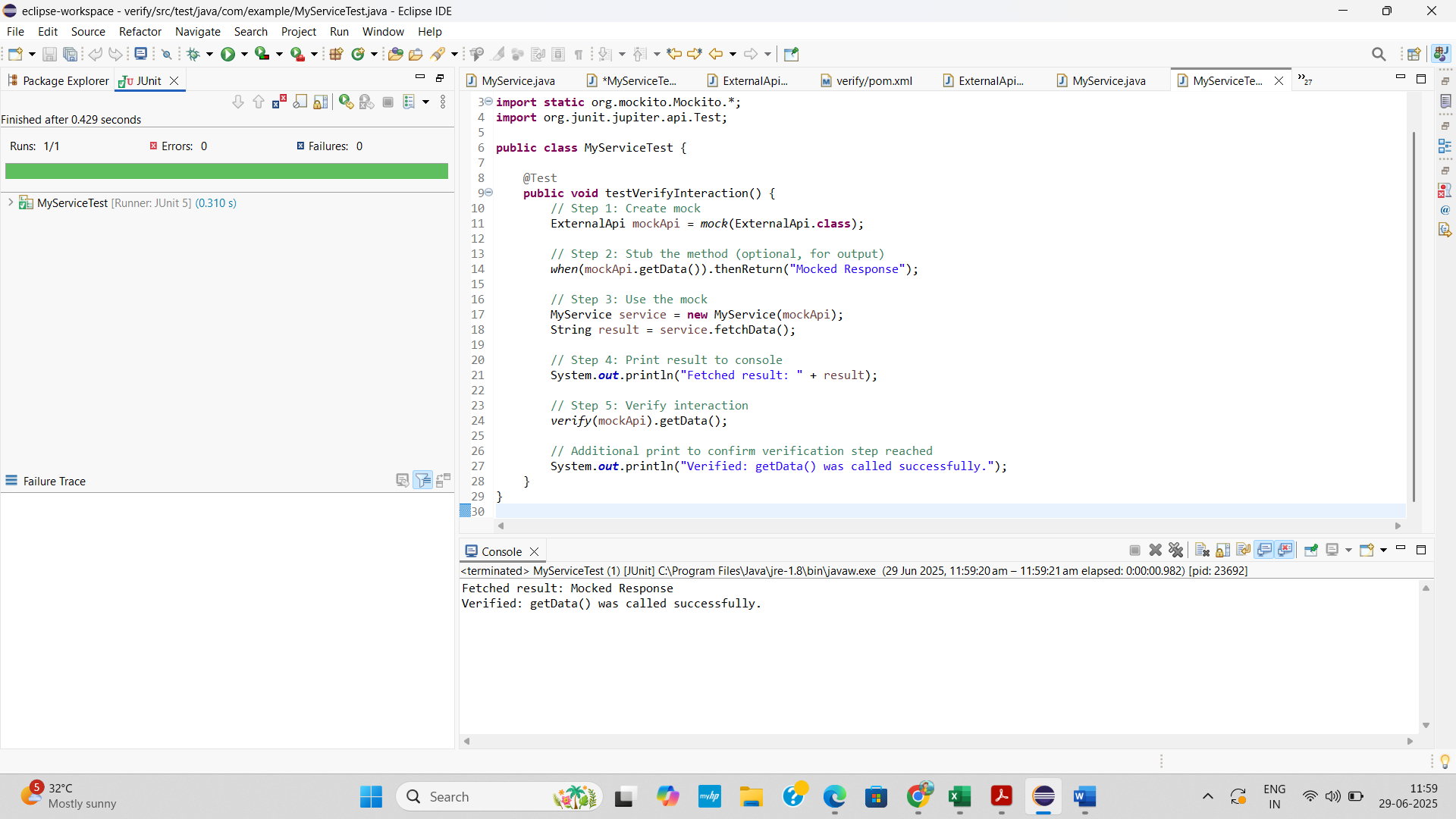
System.out.println("Verified: getData() was called successfully.");

}

}

**OUTPUT**

****

****

**SLF4J LOGGING FRAMEWORK- 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.

**APPROACH**

* Create a Maven project and add slf4j-api and logback-classic dependencies.
* Create a class and define a logger using LoggerFactory.getLogger().
* Use logger.error() and logger.warn() to log messages.
* Run the Java application and view logs in the Eclipse console.

**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) {

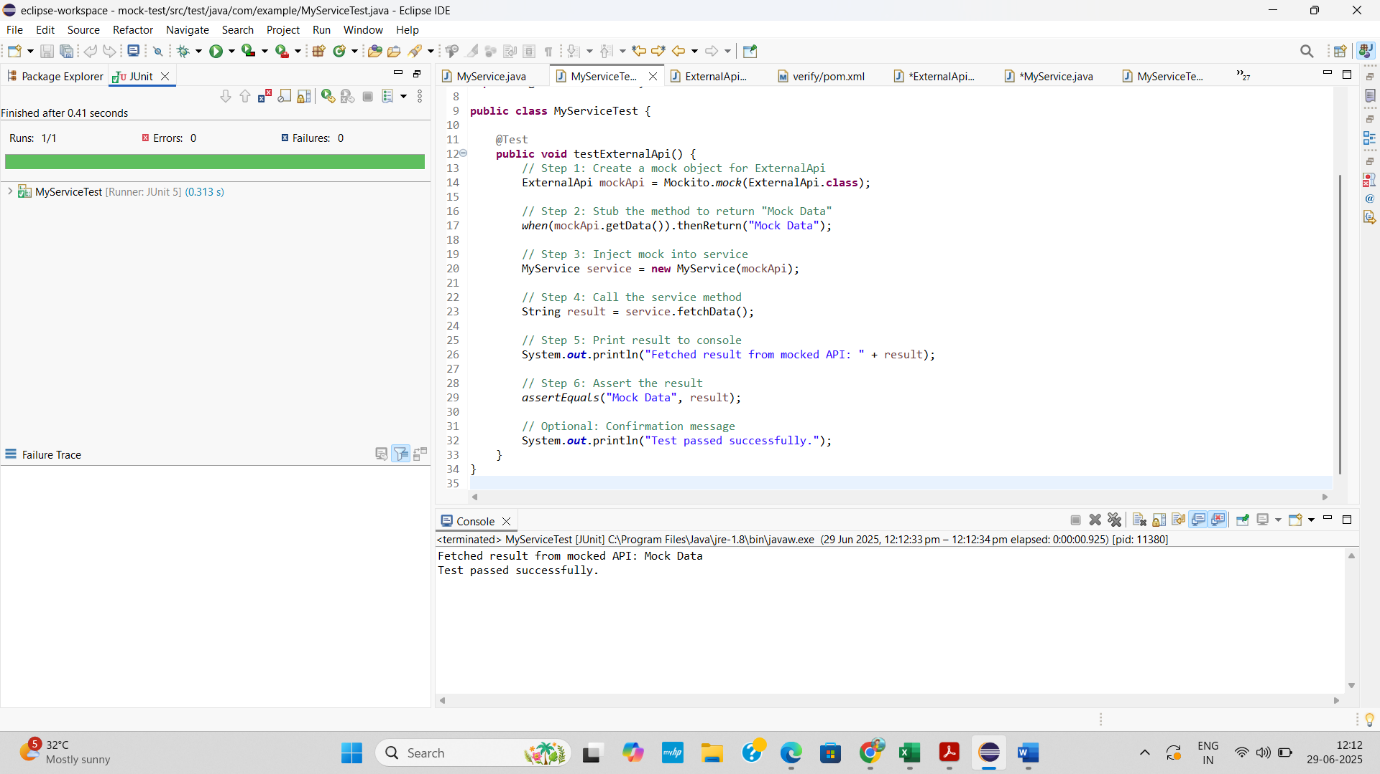
*logger*.error("This is an error message");

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

}

}

**OUTPUT**

****