**WEEK 2: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.

**CODE:**

CREATE TABLE customers (

customer\_id NUMBER PRIMARY KEY,

age NUMBER,

loan\_interest\_rate NUMBER

);

INSERT INTO customers VALUES (1, 65, 10.5);

INSERT INTO customers VALUES (2, 45, 12.0);

INSERT INTO customers VALUES (3, 70, 11.0);

INSERT INTO customers VALUES (4, 60, 9.5);

COMMIT;

BEGIN

-- Enable output if needed (some online compilers do this automatically)

FOR cust\_rec IN (

SELECT customer\_id, age, loan\_interest\_rate

FROM customers

) LOOP

IF cust\_rec.age > 60 THEN

UPDATE customers

SET loan\_interest\_rate = loan\_interest\_rate - 1

WHERE customer\_id = cust\_rec.customer\_id;

-- Show what was updated

DBMS\_OUTPUT.PUT\_LINE('Discount applied for customer ID: ' || cust\_rec.customer\_id);

END IF;

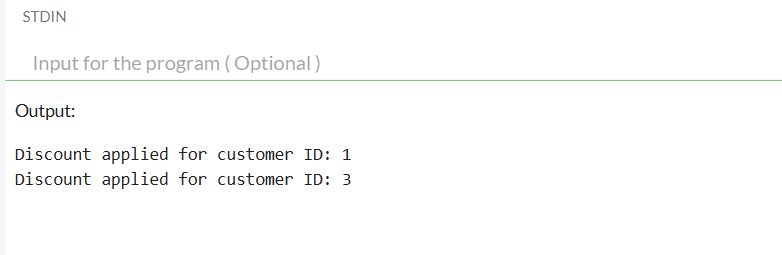
END LOOP;

COMMIT;

END;

/

**OUTPUT:**



**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 TABLE customers (

customer\_id NUMBER PRIMARY KEY,

balance NUMBER,

isvip CHAR(1) DEFAULT 'N'

);

INSERT INTO customers VALUES (1, 15000, 'N');

INSERT INTO customers VALUES (2, 5000, 'N');

INSERT INTO customers VALUES (3, 12000, 'N');

COMMIT;

BEGIN

FOR cust\_rec IN (

SELECT customer\_id, balance

FROM customers

) LOOP

IF cust\_rec.balance > 10000 THEN

UPDATE customers

SET isvip = 'Y'

WHERE customer\_id = cust\_rec.customer\_id;

DBMS\_OUTPUT.PUT\_LINE('Customer ID ' || cust\_rec.customer\_id || ' promoted to VIP.');

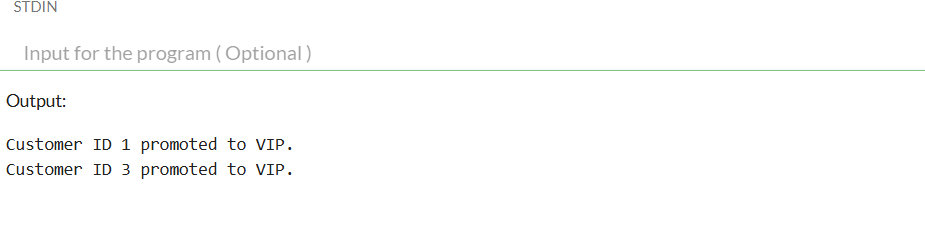
END IF;

END LOOP;

COMMIT;

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:**

BEGIN

EXECUTE IMMEDIATE 'DROP TABLE loans';

EXECUTE IMMEDIATE 'DROP TABLE customers';

EXCEPTION

WHEN OTHERS THEN NULL;

END;

CREATE TABLE customers (

customer\_id NUMBER PRIMARY KEY,

name VARCHAR2(100)

);

-- Create loans table

CREATE TABLE loans (

loan\_id NUMBER PRIMARY KEY,

customer\_id NUMBER,

due\_date DATE,

FOREIGN KEY (customer\_id) REFERENCES customers(customer\_id)

);

INSERT INTO customers VALUES (1, 'Alice');

INSERT INTO customers VALUES (2, 'Bob');

INSERT INTO customers VALUES (3, 'Charlie');

INSERT INTO loans VALUES (101, 1, SYSDATE + 15);

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

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

COMMIT;

BEGIN

FOR loan\_rec IN (

SELECT l.loan\_id, l.due\_date, c.name

FROM loans l

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

WHERE l.due\_date BETWEEN SYSDATE AND SYSDATE + 30

) LOOP

DBMS\_OUTPUT.PUT\_LINE('Reminder: Dear ' || loan\_rec.name ||

', your loan (ID: ' || loan\_rec.loan\_id ||

') is due on ' || TO\_CHAR(loan\_rec.due\_date, 'DD-MON-YYYY') ||

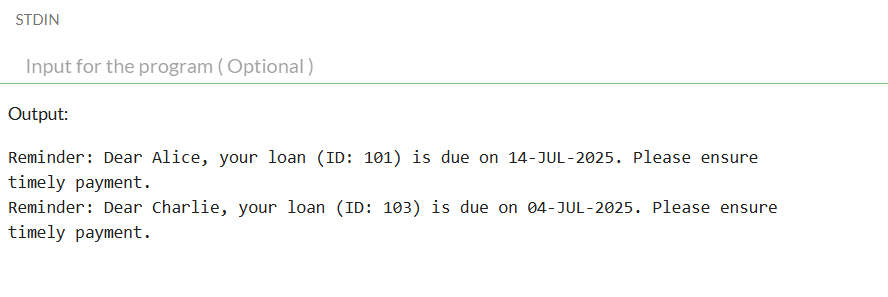
'. Please ensure timely payment.');

END LOOP;

END;

/

**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.

**CODE:**

-- Optional: Drop table if it exists (safe for reruns)

BEGIN

EXECUTE IMMEDIATE 'DROP TABLE accounts';

EXCEPTION

WHEN OTHERS THEN NULL;

END;

/

-- Step 1: Create the accounts table

CREATE TABLE accounts (

account\_id NUMBER PRIMARY KEY,

account\_type VARCHAR2(20),

balance NUMBER

);

-- Step 2: Insert sample data

INSERT INTO accounts VALUES (101, 'savings', 10000);

INSERT INTO accounts VALUES (102, 'current', 5000);

INSERT INTO accounts VALUES (103, 'savings', 20000);

COMMIT;

-- Step 3: Create the stored procedure

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

BEGIN

UPDATE accounts

SET balance = balance + (balance \* 0.01)

WHERE LOWER(account\_type) = 'savings';

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Monthly interest applied to all savings accounts.');

END;

/

-- Step 4: Execute the procedure

BEGIN

ProcessMonthlyInterest;

END;

/

-- Step 5: Display results

BEGIN

FOR acc IN (SELECT \* FROM accounts) LOOP

DBMS\_OUTPUT.PUT\_LINE('Account ID: ' || acc.account\_id ||

', Type: ' || acc.account\_type ||

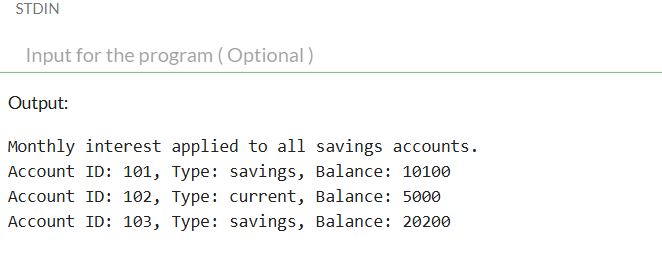
', Balance: ' || acc.balance);

END LOOP;

END;

/

**OUTPUT:**

****

**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.

**CODE:**

-- Enable output

SET SERVEROUTPUT ON;

-- Drop existing table (optional)

BEGIN

EXECUTE IMMEDIATE 'DROP TABLE employees';

EXCEPTION

WHEN OTHERS THEN NULL;

END;

/

-- Step 1: Create the employees table

CREATE TABLE employees (

employee\_id NUMBER PRIMARY KEY,

name VARCHAR2(100),

department VARCHAR2(50),

salary NUMBER

);

-- Step 2: Insert sample data

INSERT INTO employees VALUES (1, 'Alice', 'Finance', 50000);

INSERT INTO employees VALUES (2, 'Bob', 'HR', 40000);

INSERT INTO employees VALUES (3, 'Charlie', 'Finance', 55000);

INSERT INTO employees VALUES (4, 'Diana', 'IT', 60000);

COMMIT;

-- Step 3: Create the stored procedure

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (

dept\_name IN VARCHAR2,

bonus\_pct IN NUMBER -- e.g., pass 10 for 10%

) IS

BEGIN

UPDATE employees

SET salary = salary + (salary \* bonus\_pct / 100)

WHERE LOWER(department) = LOWER(dept\_name);

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Bonus of ' || bonus\_pct || '% applied to department: ' || dept\_name);

END;

/

-- Step 4: Call the procedure (e.g., give 10% bonus to Finance)

BEGIN

UpdateEmployeeBonus('Finance', 10);

END;

/

-- Step 5: Show updated employee salaries

BEGIN

FOR emp IN (SELECT \* FROM employees) LOOP

DBMS\_OUTPUT.PUT\_LINE('ID: ' || emp.employee\_id ||

', Name: ' || emp.name ||

', Dept: ' || emp.department ||

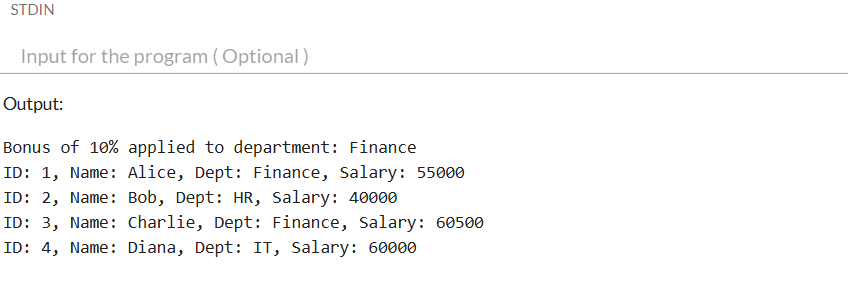
', Salary: ' || emp.salary);

END LOOP;

END;

/

**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.

**CODE:**

-- Enable output

SET SERVEROUTPUT ON;

-- Step 0: Drop existing table (optional for clean reruns)

BEGIN

EXECUTE IMMEDIATE 'DROP TABLE accounts';

EXCEPTION

WHEN OTHERS THEN NULL;

END;

/

-- Step 1: Create accounts table

CREATE TABLE accounts (

account\_id NUMBER PRIMARY KEY,

customer\_name VARCHAR2(100),

balance NUMBER

);

-- Step 2: Insert sample data

INSERT INTO accounts VALUES (1, 'Alice', 10000);

INSERT INTO accounts VALUES (2, 'Bob', 5000);

COMMIT;

-- Step 3: Create TransferFunds procedure

CREATE OR REPLACE PROCEDURE TransferFunds (

source\_account\_id IN NUMBER,

destination\_account\_id IN NUMBER,

amount IN NUMBER

) IS

insufficient\_balance EXCEPTION;

BEGIN

-- Check if source account has enough balance

DECLARE

src\_balance NUMBER;

BEGIN

SELECT balance INTO src\_balance

FROM accounts

WHERE account\_id = source\_account\_id;

IF src\_balance < amount THEN

RAISE insufficient\_balance;

END IF;

END;

-- Deduct from source

UPDATE accounts

SET balance = balance - amount

WHERE account\_id = source\_account\_id;

-- Add to destination

UPDATE accounts

SET balance = balance + amount

WHERE account\_id = destination\_account\_id;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Transfer of ' || amount || ' from Account ' || source\_account\_id ||

' to Account ' || destination\_account\_id || ' completed.');

EXCEPTION

WHEN insufficient\_balance THEN

DBMS\_OUTPUT.PUT\_LINE('Transfer failed: Insufficient balance in source account.');

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('Transfer failed: One or both account IDs not found.');

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('Transfer failed due to error: ' || SQLERRM);

ROLLBACK;

END;

/

-- Step 4: Call the procedure

BEGIN

TransferFunds(1, 2, 2000); -- should succeed

TransferFunds(2, 1, 10000); -- should fail (insufficient funds)

END;

/

-- Step 5: Display final balances

BEGIN

FOR acc IN (SELECT \* FROM accounts) LOOP

DBMS\_OUTPUT.PUT\_LINE('Account ID: ' || acc.account\_id ||

', Name: ' || acc.customer\_name ||

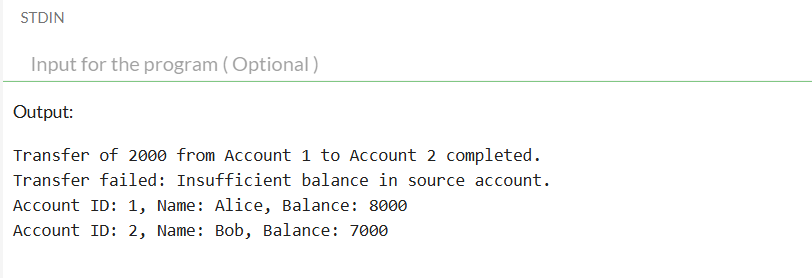
', Balance: ' || acc.balance);

END LOOP;

END;

/

**OUTPUT:**

****

**Junit\_basic testing Exercise**

**Exercise 1: Setting Up JUnit**

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

**CODE:**

package com.example;

import org.junit.jupiter.api.Test;

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

public class MainTest {

@Test

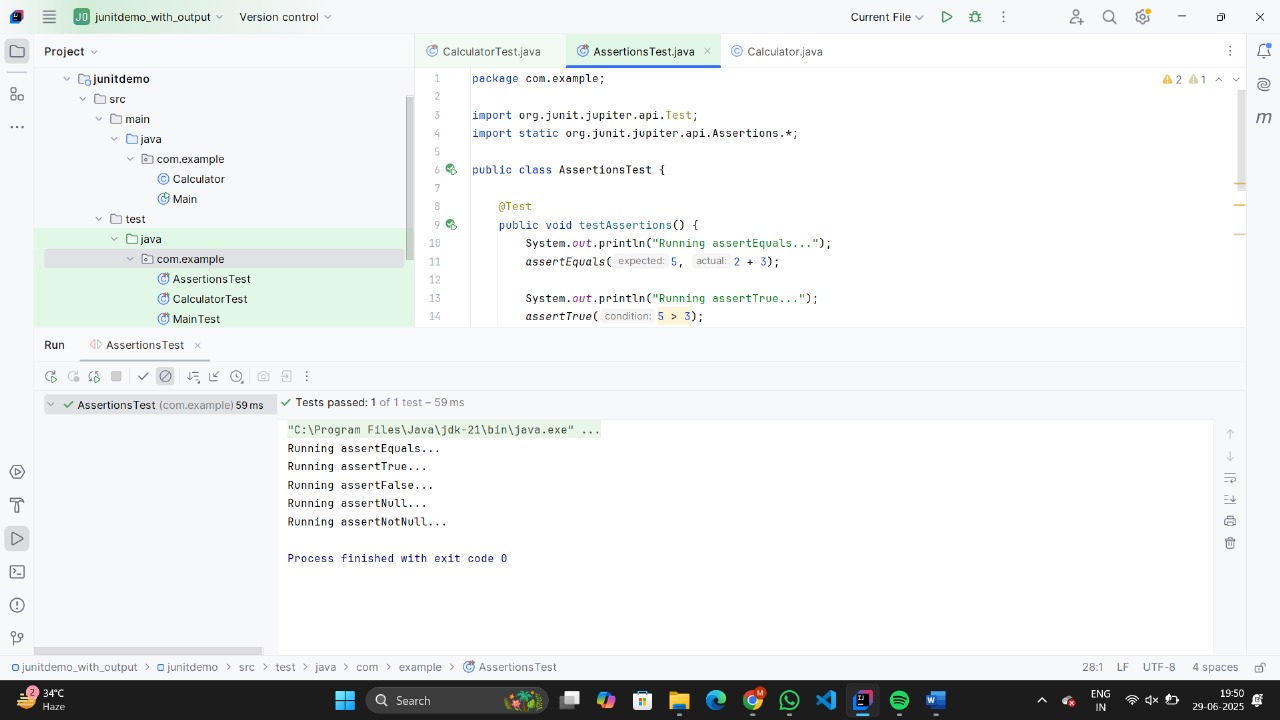
void testAddition() {

assertEquals(5, 2 + 3);

}

}

**OUTPUT:**



**Exercise 3: Assertions in Junit**

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

**CODE:**

package com.example;

import org.junit.jupiter.api.Test;

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

public class AssertionsTest {

@Test

public void testAssertions() {

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

assertEquals(5, 2 + 3);

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

assertTrue(5 > 3);

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

assertFalse(5 < 3);

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

Object obj = null;

assertNull(obj);

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

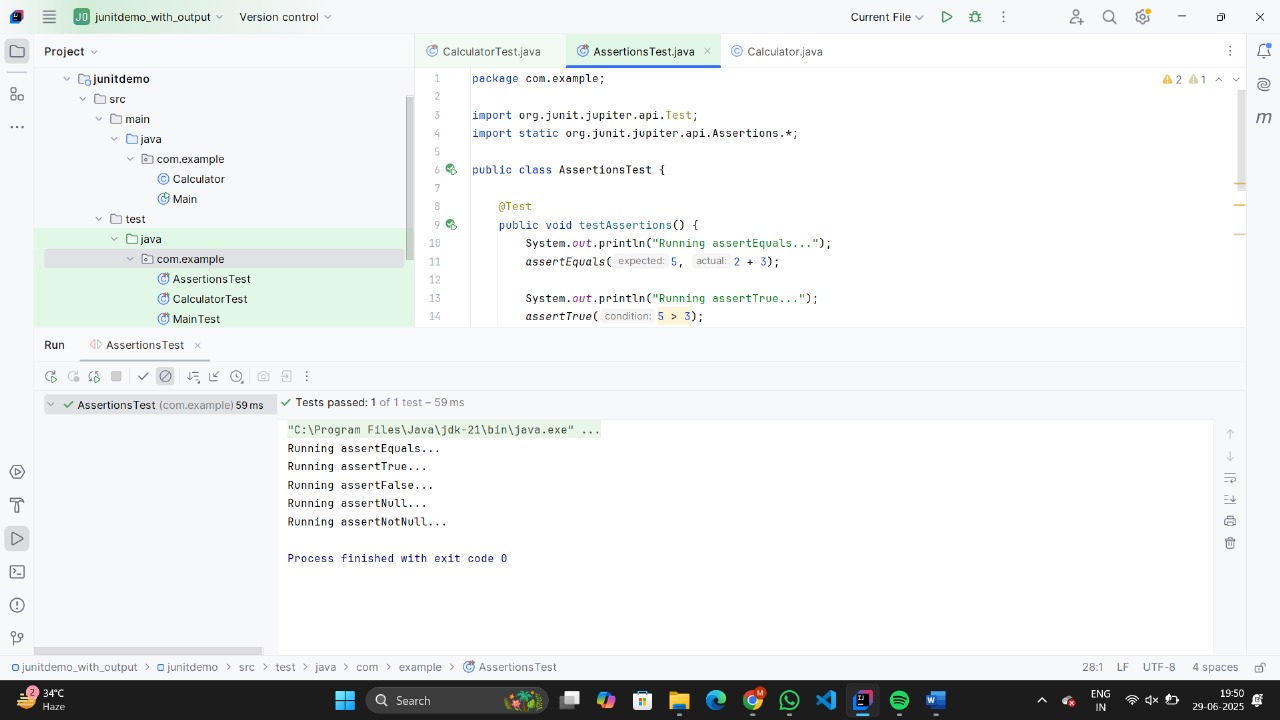
Object anotherObj = new Object();

assertNotNull(anotherObj);

}

}

**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:**

package com.example;

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

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

public class CalculatorTest {

private Calculator calculator;

@BeforeEach

void setUp() {

System.out.println("Setting up Calculator instance...");

calculator = new Calculator();

}

@AfterEach

void tearDown() {

System.out.println("Tearing down Calculator instance...");

calculator = null;

}

@Test

void testAdd() {

System.out.println("Running testAdd...");

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

assertEquals(15, result);

}

@Test

void testSubtract() {

System.out.println("Running testSubtract...");

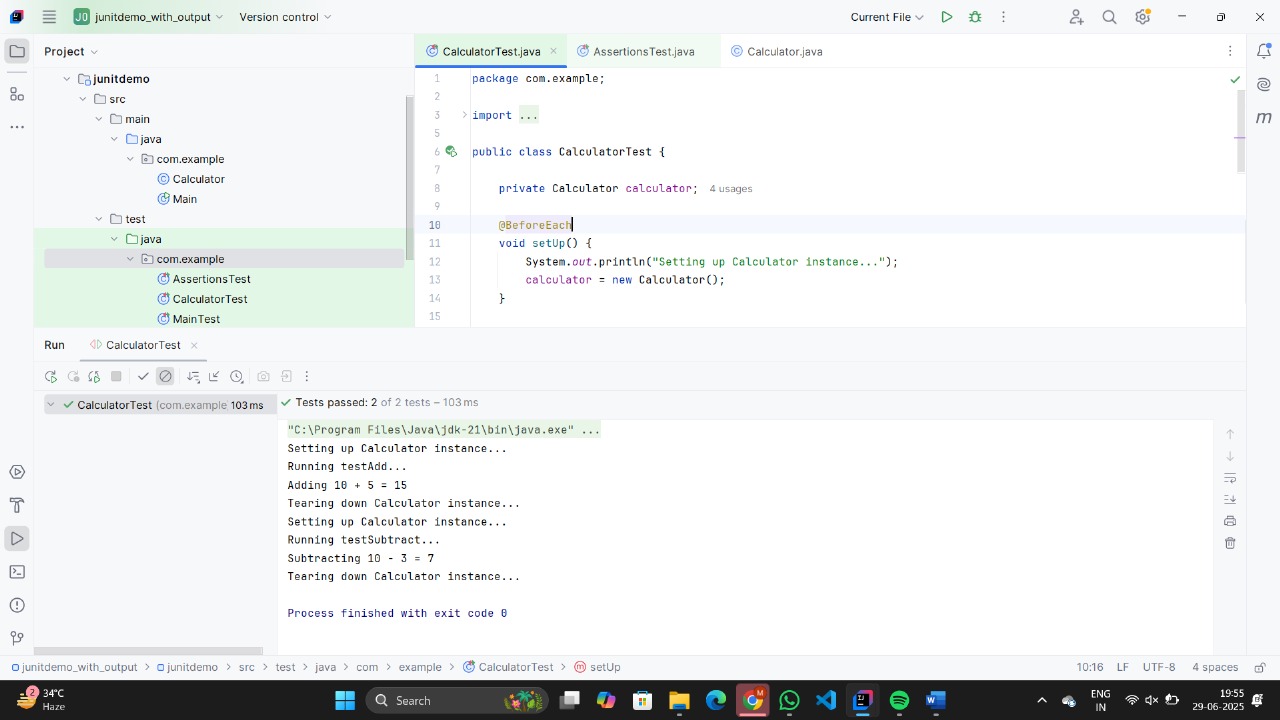
int result = calculator.subtract(10, 3);

assertEquals(7, 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:**

public interface ExternalApi {

String getData();

}

public class MyService {

private ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public String fetchData() {

return api.getData();

}

}

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;

import static org.mockito.Mockito.\*;

public class MyServiceTest {

@Test

public void testExternalApi() {

// Step 1: Create a mock object

ExternalApi mockApi = mock(ExternalApi.class);

// Step 2: Stub the method

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

// Step 3: Use mock in service and test

MyService service = new MyService(mockApi);

String result = service.fetchData();

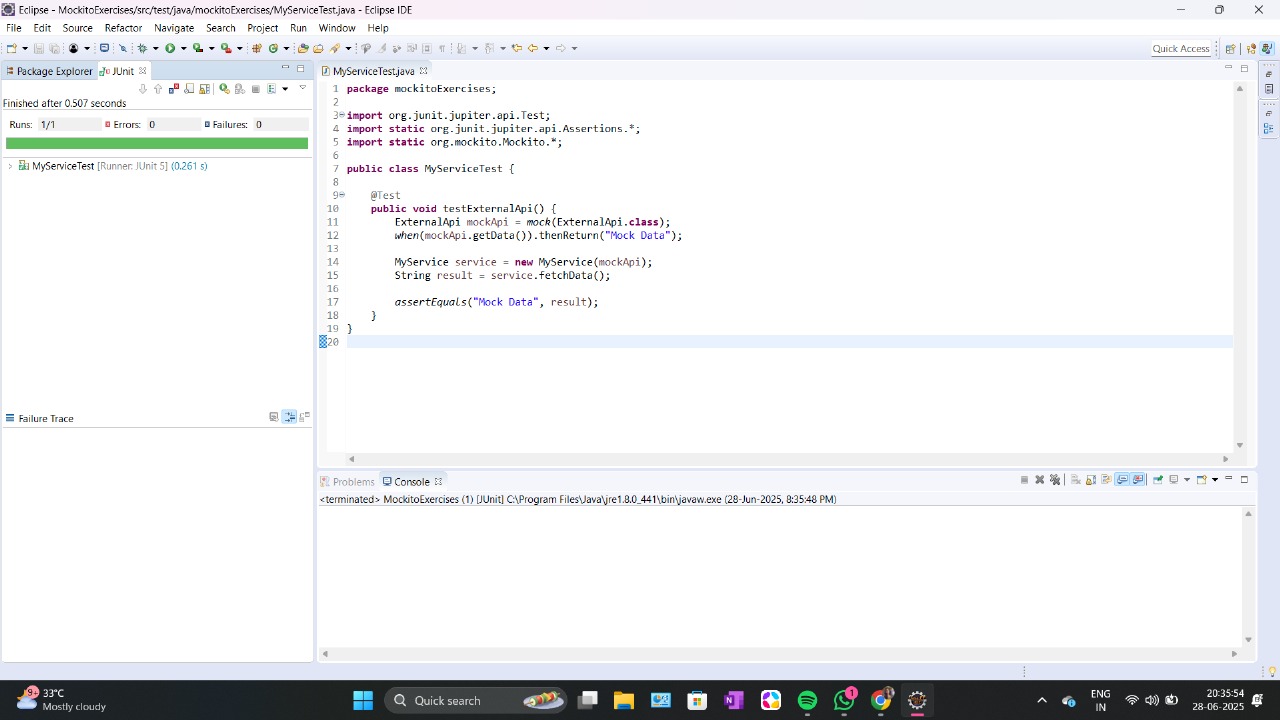
// Step 4: Verify result

assertEquals("Mock Data", result);

}

}

**OUTPUT:**



**Exercise 2: Verifying Interactions**

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

**CODE:**

public interface ExternalApi {

String getData();

}

public class MyService {

private ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public String fetchData() {

return api.getData();

}

}

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

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

public class MyServiceTest {

@Test

public void testVerifyInteraction() {

// Step 1: Create mock

ExternalApi mockApi = mock(ExternalApi.class);

// Step 2: Use the mock in the service

MyService service = new MyService(mockApi);

// Stub the method to avoid null return

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

// Step 3: Call the method

service.fetchData();

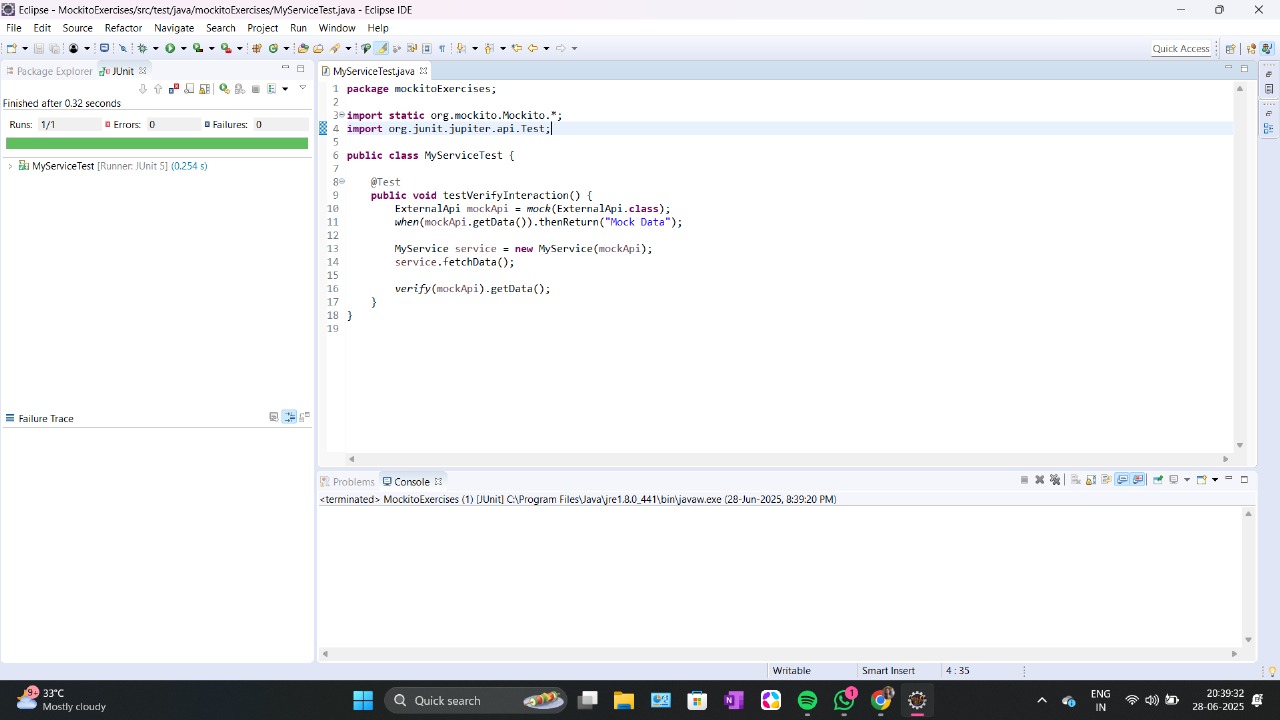
// Step 4: Verify interaction

verify(mockApi).getData(); // Verifies that getData() was called once

}

}

**OUTPUT:**



**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.

**CODE:**

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"); // Will not appear unless debug is enabled

}

}

**OUTPUT:**

