**SUPERSET ID:6426263**

**WEEK 2**

**PL/SQL programming:**

**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 (

CustomerID NUMBER PRIMARY KEY,

CustomerName VARCHAR2(50),

Age NUMBER,

Balance NUMBER,

IsVIP VARCHAR2(5)

);

CREATE TABLE Loans (

LoanID NUMBER PRIMARY KEY,

CustomerID NUMBER,

InterestRate NUMBER,

DueDate DATE,

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

INSERT INTO Customers VALUES (1, 'Alice', 65, 12000, 'FALSE');

INSERT INTO Customers VALUES (2, 'Bob', 45, 8000, 'FALSE');

INSERT INTO Customers VALUES (3, 'Charlie', 70, 15000, 'FALSE');

INSERT INTO Loans VALUES (101, 1, 9.5, SYSDATE + 10);

INSERT INTO Loans VALUES (102, 2, 10.0, SYSDATE + 40);

INSERT INTO Loans VALUES (103, 3, 8.5, SYSDATE + 20);

COMMIT; DECLARE

CURSOR customer\_cursor IS

SELECT c.CustomerID, l.LoanID, l.InterestRate

FROM Customers c

JOIN Loans l ON c.CustomerID = l.CustomerID

WHERE c.Age > 60;

BEGIN

FOR rec IN customer\_cursor LOOP

UPDATE Loans

SET InterestRate = InterestRate - 1

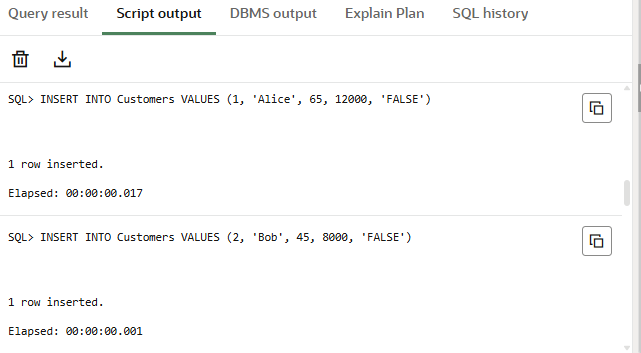
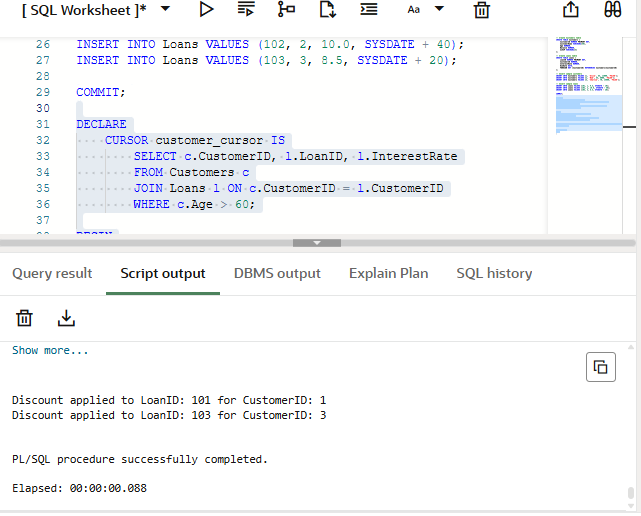
WHERE LoanID = rec.LoanID;

DBMS\_OUTPUT.PUT\_LINE('Discount applied to LoanID: ' || rec.LoanID || ' for CustomerID: ' || rec.CustomerID);

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:**

DECLARE

CURSOR vip\_cursor IS

SELECT CustomerID, Balance

FROM Customers

WHERE Balance > 10000;

BEGIN

FOR rec IN vip\_cursor LOOP

UPDATE Customers

SET IsVIP = 'TRUE'

WHERE CustomerID = rec.CustomerID;

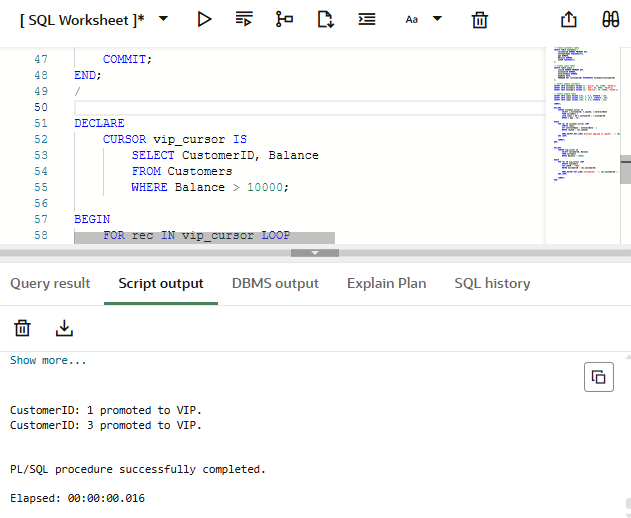
DBMS\_OUTPUT.PUT\_LINE('CustomerID: ' || rec.CustomerID || ' promoted to VIP.');

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:**

DECLARE

CURSOR due\_loan\_cursor IS

SELECT l.LoanID, l.DueDate, c.CustomerName

FROM Loans l

JOIN Customers c ON l.CustomerID = c.CustomerID

WHERE l.DueDate BETWEEN SYSDATE AND SYSDATE + 30;

BEGIN

FOR rec IN due\_loan\_cursor LOOP

DBMS\_OUTPUT.PUT\_LINE('Reminder: LoanID ' || rec.LoanID || ' for Customer ' || rec.CustomerName ||

' is due on ' || TO\_CHAR(rec.DueDate, 'DD-MON-YYYY'));

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:**

CREATE TABLE Accounts (

AccountID NUMBER PRIMARY KEY,

CustomerID NUMBER,

AccountType VARCHAR2(20),

Balance NUMBER

);

INSERT INTO Accounts VALUES (1, 101, 'SAVINGS', 1000);

INSERT INTO Accounts VALUES (2, 102, 'CURRENT', 5000);

INSERT INTO Accounts VALUES (3, 103, 'SAVINGS', 2000);

COMMIT;

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest AS

BEGIN

UPDATE Accounts

SET Balance = Balance + (Balance \* 0.01)

WHERE AccountType = 'SAVINGS';

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

END;

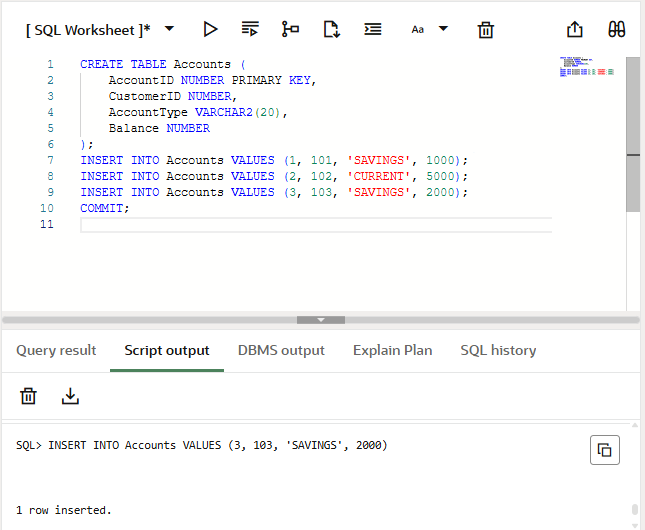
SET SERVEROUTPUT ON;

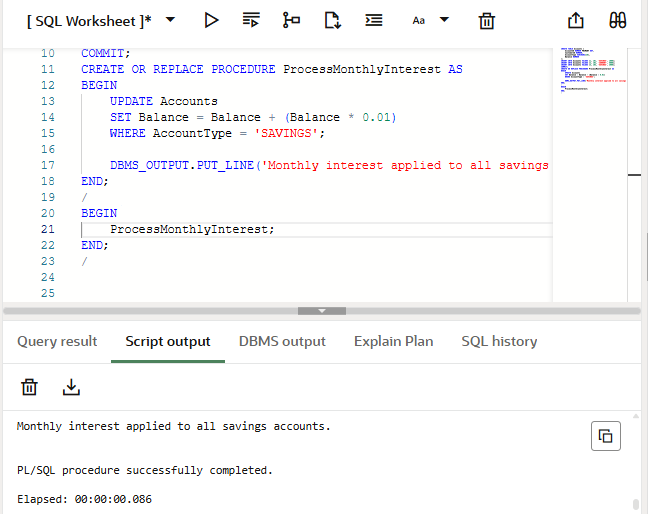
BEGIN

ProcessMonthlyInterest;

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:**

CREATE TABLE Employees (

EmpID NUMBER PRIMARY KEY,

EmpName VARCHAR2(50),

DepartmentID NUMBER,

Salary NUMBER

);

INSERT INTO Employees VALUES (1, 'Alice', 101, 50000);

INSERT INTO Employees VALUES (2, 'Bob', 101, 55000);

INSERT INTO Employees VALUES (3, 'Charlie', 102, 60000);

COMMIT;

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (

p\_DepartmentID IN NUMBER,

p\_BonusPercent IN NUMBER

) AS

BEGIN

UPDATE Employees

SET Salary = Salary + (Salary \* p\_BonusPercent / 100)

WHERE DepartmentID = p\_DepartmentID;

DBMS\_OUTPUT.PUT\_LINE('Bonus of ' || p\_BonusPercent || '% applied to Department ID: ' || p\_DepartmentID);

END;

/

SET SERVEROUTPUT ON;

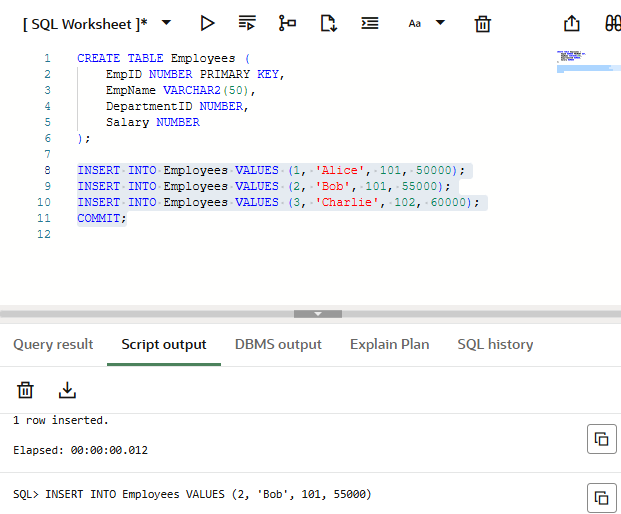
BEGIN

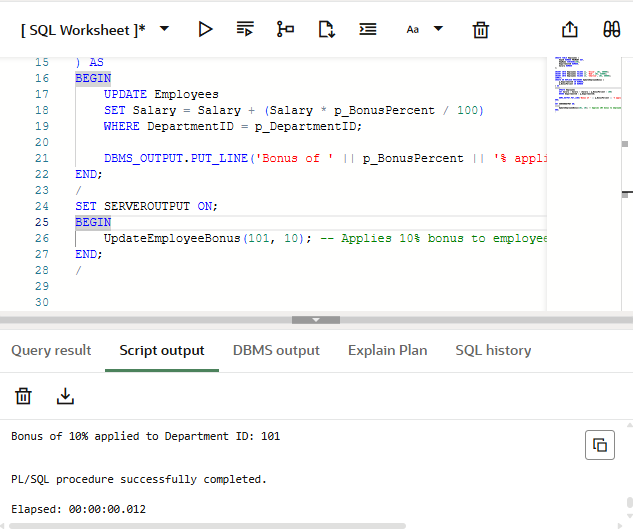
UpdateEmployeeBonus(101, 10); -- Applies 10% bonus to employees in department 101

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:**

CREATE OR REPLACE PROCEDURE TransferFunds (

p\_FromAccountID IN NUMBER,

p\_ToAccountID IN NUMBER,

p\_Amount IN NUMBER

) AS

v\_FromBalance NUMBER;

BEGIN

SELECT Balance INTO v\_FromBalance FROM Accounts WHERE AccountID = p\_FromAccountID;

IF v\_FromBalance < p\_Amount THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient balance in source account.');

END IF;

UPDATE Accounts

SET Balance = Balance - p\_Amount

WHERE AccountID = p\_FromAccountID;

UPDATE Accounts

SET Balance = Balance + p\_Amount

WHERE AccountID = p\_ToAccountID;

DBMS\_OUTPUT.PUT\_LINE('Transferred ' || p\_Amount || ' from Account ' || p\_FromAccountID || ' to ' || p\_ToAccountID);

END;

/

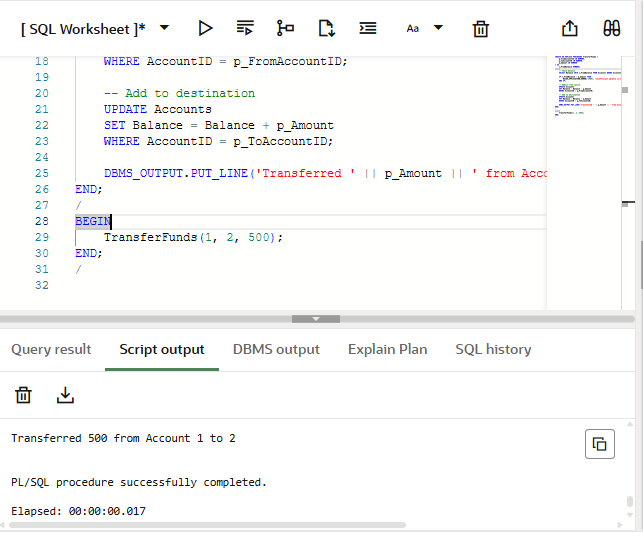
BEGIN

TransferFunds(1, 2, 500);

END;

/

**OUTPUT:**



**TDD using JUnit5 and Mockito**

**Exercise 1: Setting Up Junit**

**Code:**

public class Calculator {

public int add(int a, int b) {

return a + b;

}

}

CalculatorTest.java

import org.junit.Test;

import static org.junit.Assert.\*;

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**

**AssertionsTest.java**

import org.junit.Test;

import static org.junit.Assert.\*;

public class AssertionsTest {

@Test

public void testAssertions() {

assertEquals("Sum should be 5", 5, 2 + 3);

assertTrue("5 is greater than 3", 5 > 3);

assertFalse("5 is not less than 3", 5 < 3);

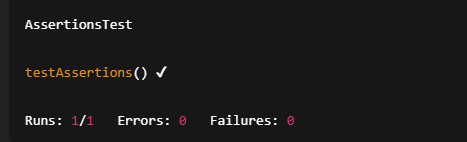
assertNull("Object should be null", null);

assertNotNull("Object should not be null", new Object());

}

}

**OUTPUT:**

****

**Exercise 4: Arrange-Act-Assert (AAA) Pattern, Test Fixtures, Setup and Teardown Methods in Junit**

**Calculator.java**

public class Calculator {

public int add(int a, int b) {

return a + b;

}

public int subtract(int a, int b) {

return a - b;

}

}

**CalculatorTest.java**

import org.junit.Before;

import org.junit.After;

import org.junit.Test;

import static org.junit.Assert.\*;

public class CalculatorTest {

private Calculator calculator;

@Before

public void setUp() {

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

calculator = new Calculator();

}

@After

public void tearDown() {

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

calculator = null;

}

@Test

public void testAdd() {

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

assertEquals("10 + 5 should be 15", 15, result);

}

@Test

public void testSubtract() {

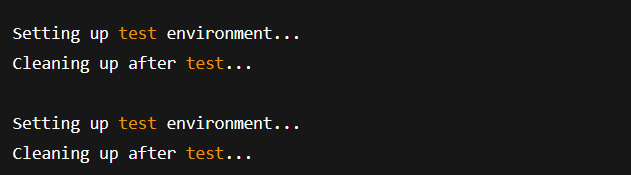
int result = calculator.subtract(10, 5);

assertEquals("10 - 5 should be 5", 5, result);

}

}

**OUTPUT:**

****

**3. MOCKITO EXERCISES**

**EXERCISE 1: MOCKING AND STUBBING**

public interface ExternalApi {

String getData();

}

**Service class**

public class MyService {

private ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public String fetchData() {

return api.getData();

}

}

**Test class**

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() {

ExternalApi mockApi = 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**

public interface ExternalApi {

String getData();

}

**Service class**

public class MyService {

private ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public String fetchData() {

return api.getData();

}

}

**Test class**

import org.junit.jupiter.api.Test;

import static org.mockito.Mockito.\*;

public class MyServiceTest {

@Test

public void testVerifyInteraction() {

ExternalApi mockApi = mock(ExternalApi.class);

MyService service = new MyService(mockApi);

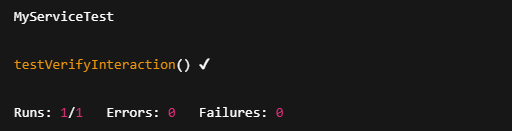
service.fetchData();

verify(mockApi).getData();

}

}

**OUTPUT:**

****

**6.SL4J LOGGING EXERCISES**

**EXERCISE 1: LOGGING ERROR MESSAGES AND WARNING LEVELS TASK**

**LoggingExample.java**

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");

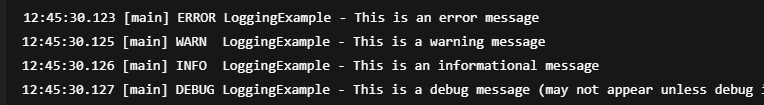
logger.info("This is an informational message");

logger.debug("This is a debug message (may not appear unless debug is enabled)");

}

}

**OUTPUT:**

****