MANDATORY EXERCISES 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.

o 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.

o 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.

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

**TABLE**-- Sample structure

CREATE TABLE Customers (

CustomerID NUMBER,

Name VARCHAR2(50),

Age NUMBER,

Balance NUMBER,

IsVIP CHAR(1)

);

CREATE TABLE Loans (

LoanID NUMBER,

CustomerID NUMBER,

InterestRate NUMBER,

DueDate DATE

);

-- Sample Data

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

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

INSERT INTO Customers VALUES (3, 'Carol', 70, 15000, 'N');

INSERT INTO Loans VALUES (101, 1, 8.5, SYSDATE + 20);

INSERT INTO Loans VALUES (102, 2, 9.0, SYSDATE + 45);

INSERT INTO Loans VALUES (103, 3, 7.5, SYSDATE + 10);

COMMIT;

**CODE**

BEGIN

FOR cust IN (SELECT c.CustomerID, l.LoanID, l.InterestRate

FROM Customers c

JOIN Loans l ON c.CustomerID = l.CustomerID

WHERE c.Age > 60) LOOP

UPDATE Loans

SET InterestRate = InterestRate - 1

WHERE LoanID = cust.LoanID;

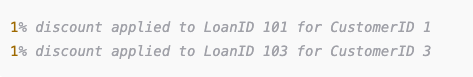
DBMS\_OUTPUT.PUT\_LINE('1% discount applied to LoanID ' || cust.LoanID || ' for CustomerID ' || cust.CustomerID);

END LOOP;

COMMIT;

END;

**OUTPUT**



**CODE**

BEGIN

FOR cust IN (SELECT CustomerID FROM Customers WHERE Balance > 10000) LOOP

UPDATE Customers

SET IsVIP = 'Y'

WHERE CustomerID = cust.CustomerID;

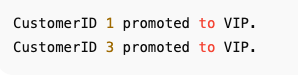
DBMS\_OUTPUT.PUT\_LINE('CustomerID ' || cust.CustomerID || ' promoted to VIP.');

END LOOP;

COMMIT;

END;

**OUTPUT**



**CODE**

BEGIN

FOR dueLoan IN (SELECT l.LoanID, c.Name, l.DueDate

FROM Loans l

JOIN Customers c ON l.CustomerID = c.CustomerID

WHERE l.DueDate <= SYSDATE + 30) LOOP

DBMS\_OUTPUT.PUT\_LINE('Reminder: LoanID ' || dueLoan.LoanID ||

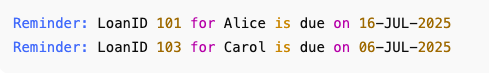
' for ' || dueLoan.Name ||

' is due on ' || TO\_CHAR(dueLoan.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.

o 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.

o 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.

o 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.

**TABLE**

-- Table for Savings Accounts

CREATE TABLE SavingsAccounts (

AccountID NUMBER PRIMARY KEY,

CustomerID NUMBER,

Balance NUMBER

);

-- Table for Employees

CREATE TABLE Employees (

EmployeeID NUMBER PRIMARY KEY,

Name VARCHAR2(50),

DepartmentID NUMBER,

Salary NUMBER

);

-- Table for Bank Accounts (for fund transfer)

CREATE TABLE BankAccounts (

AccountID NUMBER PRIMARY KEY,

CustomerID NUMBER,

Balance NUMBER

);

**CODE**

-- SavingsAccounts

INSERT INTO SavingsAccounts VALUES (1, 101, 10000);

INSERT INTO SavingsAccounts VALUES (2, 102, 20000);

-- Employees

INSERT INTO Employees VALUES (1, 'Alice', 10, 50000);

INSERT INTO Employees VALUES (2, 'Bob', 20, 45000);

INSERT INTO Employees VALUES (3, 'Carol', 10, 48000);

-- BankAccounts

INSERT INTO BankAccounts VALUES (1, 101, 15000);

INSERT INTO BankAccounts VALUES (2, 101, 5000);

COMMIT;

**CODE**

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest AS

BEGIN

FOR acc IN (SELECT AccountID, Balance FROM SavingsAccounts) LOOP

UPDATE SavingsAccounts

SET Balance = Balance + (Balance \* 0.01)

WHERE AccountID = acc.AccountID;

DBMS\_OUTPUT.PUT\_LINE('Updated AccountID ' || acc.AccountID ||

' with new balance: ' || TO\_CHAR(acc.Balance \* 1.01));

END LOOP;

COMMIT;

END;

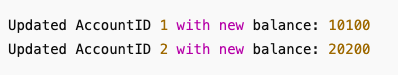
**CODE**

BEGIN

ProcessMonthlyInterest;

END;

**OUTPUT**



**CODE**

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (

dept\_id IN NUMBER,

bonus\_percent IN NUMBER

) AS

BEGIN

FOR emp IN (SELECT EmployeeID, Salary FROM Employees WHERE DepartmentID = dept\_id) LOOP

UPDATE Employees

SET Salary = Salary + (Salary \* bonus\_percent / 100)

WHERE EmployeeID = emp.EmployeeID;

DBMS\_OUTPUT.PUT\_LINE('Updated Salary for EmployeeID ' || emp.EmployeeID ||

' to: ' || TO\_CHAR(emp.Salary \* (1 + bonus\_percent / 100)));

END LOOP;

COMMIT;

END;

**CODE**

BEGIN

UpdateEmployeeBonus(10, 10); -- 10% bonus to department 10

END;

**OUTPUT**



**CODE**

CREATE OR REPLACE PROCEDURE TransferFunds (

from\_acct IN NUMBER,

to\_acct IN NUMBER,

amount IN NUMBER

) AS

insufficient\_balance EXCEPTION;

BEGIN

-- Check balance

DECLARE

source\_balance NUMBER;

BEGIN

SELECT Balance INTO source\_balance FROM BankAccounts WHERE AccountID = from\_acct;

IF source\_balance < amount THEN

RAISE insufficient\_balance;

ELSE

-- Deduct from source

UPDATE BankAccounts

SET Balance = Balance - amount

WHERE AccountID = from\_acct;

-- Add to destination

UPDATE BankAccounts

SET Balance = Balance + amount

WHERE AccountID = to\_acct;

DBMS\_OUTPUT.PUT\_LINE('Transfer of ' || amount ||

' from Account ' || from\_acct ||

' to Account ' || to\_acct || ' successful.');

END IF;

COMMIT;

EXCEPTION

WHEN insufficient\_balance THEN

DBMS\_OUTPUT.PUT\_LINE('Transfer failed: Insufficient balance in Account ' || from\_acct);

END;

END;

**CODE**

BEGIN

TransferFunds(1, 2, 3000);

END;

**OUTPUT**



**CODE**

BEGIN

TransferFunds(2, 1, 6000);

END;

**OUTPUT**



**TDD USING JUNIT TESTING AND MOCKITO**

**JUnit Testing Exercises**

**Exercise 1: Setting Up JUnit**

Scenario:

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

Steps:

1. Create a new Java project in your IDE (e.g., IntelliJ IDEA, Eclipse).

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

<dependency> <groupId>junit</groupId> <artifactId>junit</artifactId> <version>4.13.2</version> <scope>test</scope>

</dependency>

3. Create a new test class in your project.

**CODE**

<dependencies>

<!-- JUnit dependency -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13.2</version>

<scope>test</scope>

</dependency>

</dependencies>

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;

}

}

package com.example;

import org.junit.Test;

import static org.junit.Assert.\*;

public class CalculatorTest {

@Test

public void testAddition() {

Calculator calc = new Calculator();

assertEquals(10, calc.add(5, 5));

}

@Test

public void testSubtraction() {

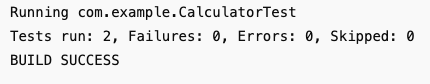
Calculator calc = new Calculator();

assertEquals(0, calc.subtract(5, 5));

}

}

**OUTPUT**



**Exercise 3: Assertions in JUnit**

Scenario:

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

1. Write tests using various JUnit assertions. Solution Code:

public class AssertionsTest { @Test

public void testAssertions() { // Assert equals assertEquals(5, 2 + 3);

// Assert true assertTrue(5 > 3);

// Assert false assertFalse(5 < 3);

// Assert null assertNull(null);

// Assert not null

assertNotNull(new Object()); }

}

**CODE**

import org.junit.Test;

import static org.junit.Assert.\*;

public class AssertionsTest {

@Test

public void testAssertions() {

// Assert equals

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

// Assert true

assertTrue("5 should be greater than 3", 5 > 3);

// Assert false

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

// Assert null

Object obj1 = null;

assertNull("Object should be null", obj1);

// Assert not null

Object obj2 = new Object();

assertNotNull("Object should not be null", obj2);

}

}

<dependency>

<groupId>junit</groupId>

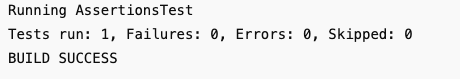
<artifactId>junit</artifactId>

<version>4.13.2</version>

<scope>test</scope>

</dependency>

**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.

Steps:

1. Write tests using the AAA pattern.

2. Use @Before and @After annotations for setup and teardown methods.

**CODE**

import org.junit.Before;

import org.junit.After;

import org.junit.Test;

import static org.junit.Assert.\*;

public class CalculatorTest {

private Calculator calculator;

// Setup method: runs before each test

@Before

public void setUp() {

calculator = new Calculator();

System.out.println("Setup: Calculator initialized.");

}

// Teardown method: runs after each test

@After

public void tearDown() {

calculator = null;

System.out.println("Teardown: Calculator cleaned up.\n");

}

@Test

public void testAddition() {

// Arrange

int a = 10;

int b = 20;

// Act

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

// Assert

assertEquals("Addition should return 30", 30, result);

}

@Test

public void testSubtraction() {

// Arrange

int a = 15;

int b = 5;

// Act

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

// Assert

assertEquals("Subtraction should return 10", 10, result);

}

}

public class Calculator {

public int add(int x, int y) {

return x + y;

}

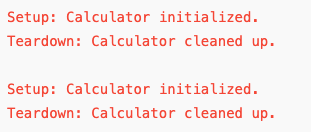
public int subtract(int x, int y) {

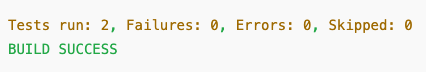
return x - y;

}

}

**OUTPUT**





**Mockito Hands-On 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.

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.

Solution Code:

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

} }

**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(); // Calls external API (mocked during test)

}

}

import org.junit.jupiter.api.Test;

import org.mockito.Mockito;

import static org.mockito.Mockito.\*;

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

public class MyServiceTest {

@Test

public void testExternalApi() {

// Step 1: Create mock of ExternalApi

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

// Step 2: Stub the method getData()

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

// Step 3: Inject mock into service

MyService service = new MyService(mockApi);

// Act

String result = service.fetchData();

// Assert

assertEquals("Mock Data", result);

}

}

**CODE**

<dependencies>

<!-- JUnit 5 -->

<dependency>

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

<artifactId>junit-jupiter</artifactId>

<version>5.10.0</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>

**OUTPUT**



**Exercise 2: Verifying Interactions**

Scenario:

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

Steps:

1. Create a mock object.

2. Call the method with specific arguments. 3. Verify the interaction.

Solution Code:

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();

} }

**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(); // Should trigger interaction

}

}

import org.junit.jupiter.api.Test;

import static org.mockito.Mockito.\*;

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

import org.mockito.Mockito;

public class MyServiceTest {

@Test

public void testVerifyInteraction() {

// Step 1: Create mock object

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

// Step 2: Create service with mocked API

MyService service = new MyService(mockApi);

// Step 3: Call method on the service

service.fetchData();

// Step 4: Verify that mockApi.getData() was called

verify(mockApi).getData(); // ✅ Will fail if not called

}

}

verify(mockApi, times(1)).getData(); // Verify called exactly once

verify(mockApi, never()).toString(); // Verify another method was NOT called

**CODE**

<dependencies>

<dependency>

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

<artifactId>junit-jupiter</artifactId>

<version>5.10.0</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

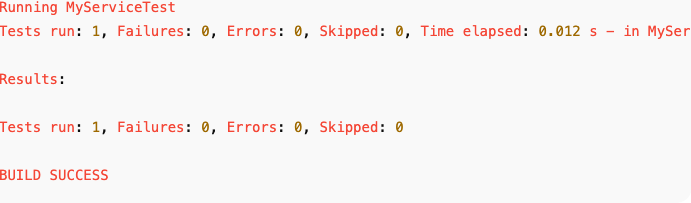
<version>5.11.0</version>

<scope>test</scope>

</dependency>

</dependencies>

**OUTPUT**



**Logging using SLF4J**

Exercise 1: Logging Error Messages and Warning Levels

Task: Write a Java application that demonstrates logging error messages and warning levels using SLF4J.

Step-by-Step Solution:

1. Add SLF4J and Logback dependencies to your `pom.xml` file:

<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>

2. Create a Java class that uses SLF4J for logging:

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");

} }

**CODE**

<dependencies>

<!-- SLF4J API -->

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>1.7.30</version>

</dependency>

<!-- Logback (implementation of SLF4J) -->

<dependency>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-classic</artifactId>

<version>1.2.3</version>

</dependency>

</dependencies>

**CODE**

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class LoggingExample {

// Create logger for this class

private static final Logger logger = LoggerFactory.getLogger(LoggingExample.class);

public static void main(String[] args) {

// Log an error-level message

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

// Log a warning-level message

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

}

}

**OUTPUT**

