**Exercise 1: Control Structures**

DECLARE

CURSOR senior\_cust IS

SELECT customer\_id, interest\_rate FROM Customers WHERE age > 60;

CURSOR vip\_cust IS

SELECT customer\_id FROM Customers WHERE balance > 10000;

CURSOR due\_loans IS

SELECT loan\_id, customer\_id, due\_date

FROM Loans

WHERE due\_date <= SYSDATE + 30;

BEGIN

-- Scenario 1: Apply 1% discount for customers above 60

FOR cust IN senior\_cust LOOP

UPDATE Customers

SET interest\_rate = interest\_rate - (interest\_rate \* 0.01)

WHERE customer\_id = cust.customer\_id;

END LOOP;

-- Scenario 2: Promote to VIP based on balance

FOR cust IN vip\_cust LOOP

UPDATE Customers

SET isVIP = 'TRUE'

WHERE customer\_id = cust.customer\_id;

END LOOP;

-- Scenario 3: Print loan due reminders

FOR loan IN due\_loans LOOP

DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan ' || loan.loan\_id ||

' for Customer ' || loan.customer\_id ||

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

END LOOP;

COMMIT;

END;

**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.

BEGIN

FOR cust IN (SELECT customer\_id, age, interest\_rate FROM Customers WHERE age > 60) LOOP

UPDATE Customers

SET interest\_rate = interest\_rate - (interest\_rate \* 0.01)

WHERE customer\_id = cust.customer\_id;

END LOOP;

COMMIT;

END;

**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.

BEGIN

FOR cust IN (SELECT customer\_id, balance FROM Customers WHERE balance > 10000) LOOP

UPDATE Customers

SET isVIP = 'TRUE'

WHERE customer\_id = cust.customer\_id;

END LOOP;

COMMIT;

END;

**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.

BEGIN

FOR loan IN (

SELECT loan\_id, customer\_id, due\_date

FROM Loans

WHERE due\_date <= SYSDATE + 30

) LOOP

DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan ' || loan.loan\_id || ' for customer ' || loan.customer\_id ||

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

END LOOP;

END;

**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.

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

BEGIN

UPDATE Accounts

SET balance = balance + (balance \* 0.01)

WHERE account\_type = 'SAVINGS';

COMMIT;

END;

**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.

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(

dept\_id IN NUMBER,

bonus\_pct IN NUMBER

) IS

BEGIN

UPDATE Employees

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

WHERE department\_id = dept\_id;

COMMIT;

END;

**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.

CREATE OR REPLACE PROCEDURE TransferFunds(

from\_acc IN NUMBER,

to\_acc IN NUMBER,

amt IN NUMBER

) IS

insufficient\_balance EXCEPTION;

current\_balance NUMBER;

BEGIN

-- Check balance of source account

SELECT balance INTO current\_balance FROM Accounts WHERE account\_id = from\_acc FOR UPDATE;

IF current\_balance < amt THEN

RAISE insufficient\_balance;

END IF;

-- Deduct from source

UPDATE Accounts

SET balance = balance - amt

WHERE account\_id = from\_acc;

-- Credit to destination

UPDATE Accounts

SET balance = balance + amt

WHERE account\_id = to\_acc;

COMMIT;

EXCEPTION

WHEN insufficient\_balance THEN

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

WHEN OTHERS THEN

ROLLBACK;

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

END;

**xercise 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

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13.2</version>

<scope>test</scope>

</dependency>

public class Calculator {

public int add(int a, int b) {

return a + b;

}

}

import org.junit.Test;

import static org.junit.Assert.\*;

public class CalculatorTest {

@Test

public void testAdd() {

Calculator calc = new Calculator();

int result = calc.add(3, 4);

assertEquals(7, result);

}

}

**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.

import org.junit.Test;

import static org.junit.Assert.\*;

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());

}

}

**Exercise 4: Functions**

**Scenario 1:** Calculate the age of customers for eligibility checks.

* + **Question:** Write a function CalculateAge that takes a customer's date of birth as input and returns their age in years.

**Scenario 2:** The bank needs to compute the monthly installment for a loan.

* + **Question:** Write a function **CalculateMonthlyInstallment** that takes the loan amount, interest rate, and loan duration in years as input and returns the monthly installment amount.

**Scenario 3:** Check if a customer has sufficient balance before making a transaction.

* + **Question:** Write a function **HasSufficientBalance** that takes an account ID and an amount as input and returns a boolean indicating whether the account has at least the specified amount.

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() {

// Setup: Initialize objects before each test

calculator = new Calculator();

System.out.println("Setup complete.");

}

@After

public void tearDown() {

// Teardown: Cleanup resources after each test

calculator = null;

System.out.println("Teardown complete.");

}

@Test

public void testAddition() {

// Arrange

int a = 10;

int b = 20;

// Act

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

// Assert

assertEquals(30, result);

}

@Test

public void testSubtraction() {

// Arrange

int a = 15;

int b = 5;

// Act

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

// Assert

assertEquals(10, result);

}

}

**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

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

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

public class MyServiceTest {

@Test

public void testExternalApi() {

// Arrange: Create mock and stub method

ExternalApi mockApi = mock(ExternalApi.class);

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

// Act: Use mock in service

MyService service = new MyService(mockApi);

String result = service.fetchData();

// Assert: Validate result

assertEquals("Mock Data", result);

}

}

**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

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

public class MyServiceTest {

@Test

public void testVerifyInteraction() {

// Arrange

ExternalApi mockApi = mock(ExternalApi.class);

MyService service = new MyService(mockApi);

// Act

service.fetchData();

// Assert

verify(mockApi).getData(); // checks if getData() was called

}

}

**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");

}

}

<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>

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class LoggingExample {

// Logger instance

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

public static void main(String[] args) {

logger.error("This is an error message"); // Severe issue

logger.warn("This is a warning message"); // Something to watch for

}

}

<configuration>

<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">

<encoder>

<pattern>%d{HH:mm:ss} %-5level %logger{36} - %msg%n</pattern>

</encoder>

</appender>

<root level="warn">

<appender-ref ref="STDOUT" />

</root>

</configuration>