WEEK 2 HANDS ON

**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.

DECLARE

CURSOR customer\_cursor IS

SELECT c.customer\_id, c.date\_of\_birth, l.loan\_id, l.interest\_rate

FROM customers c

JOIN loans l ON c.customer\_id = l.customer\_id;

v\_age NUMBER;

BEGIN

FOR customer\_rec IN customer\_cursor LOOP

-- Calculate age based on date of birth

v\_age := FLOOR(MONTHS\_BETWEEN(SYSDATE, customer\_rec.date\_of\_birth) / 12);

IF v\_age > 60 THEN

UPDATE loans

SET interest\_rate = interest\_rate - 1

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

END IF;

END LOOP;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error in Scenario 1: ' || SQLERRM);

END;

/   
  
OUTPUT:

Discount applied for customer 1, loan 1

**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.

DECLARE

CURSOR customer\_cursor IS

SELECT customer\_id, balance

FROM customers;

BEGIN

FOR customer\_rec IN customer\_cursor LOOP

IF customer\_rec.balance > 10000 THEN

UPDATE customers

SET IsVIP = 'TRUE'

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

END IF;

END LOOP;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error in Scenario 2: ' || SQLERRM);

END;

/  
  
OUTPUT:

VIP status set for customer 1

**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.

DECLARE

CURSOR loan\_cursor IS

SELECT c.customer\_id, c.first\_name, c.last\_name, l.loan\_id, l.due\_date

FROM customers c

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

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

BEGIN

FOR loan\_rec IN loan\_cursor LOOP

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

loan\_rec.last\_name || ', your loan (ID: ' ||

loan\_rec.loan\_id || ') is due on ' ||

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

END LOOP;

EXCEPTION

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('Error in Scenario 3: ' || SQLERRM);

END;

/  
  
OUTPUT:

Reminder: Dear John Doe, your loan (ID: 1) is due on 15-JUL-2025.  
  
  
  
  
**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

CURSOR savings\_accounts IS

SELECT account\_id, balance FROM accounts WHERE account\_type = 'SAVINGS';

v\_interest\_rate NUMBER := 0.01; -- 1% interest

v\_new\_balance NUMBER;

BEGIN

FOR account\_rec IN savings\_accounts LOOP

v\_new\_balance := account\_rec.balance + (account\_rec.balance \* v\_interest\_rate);

UPDATE accounts

SET balance = v\_new\_balance

WHERE account\_id = account\_rec.account\_id;

DBMS\_OUTPUT.PUT\_LINE('Account ' || account\_rec.account\_id ||

': Old balance ' || account\_rec.balance ||

', New balance ' || v\_new\_balance);

END LOOP;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Monthly interest processing completed.');

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error processing monthly interest: ' || SQLERRM);

END ProcessMonthlyInterest;

/

OUTPUT:

Account 1001: Old balance 5000, New balance 5050

Account 1002: Old balance 10000, New balance 10100

Monthly interest processing completed.

**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(

p\_department\_id IN NUMBER,

p\_bonus\_percentage IN NUMBER

) IS

v\_updated\_count NUMBER := 0;

BEGIN

IF p\_bonus\_percentage <= 0 THEN

DBMS\_OUTPUT.PUT\_LINE('Bonus percentage must be positive.');

RETURN;

END IF;

UPDATE employees

SET salary = salary + (salary \* p\_bonus\_percentage / 100)

WHERE department\_id = p\_department\_id

RETURNING COUNT(\*) INTO v\_updated\_count;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE(v\_updated\_count || ' employees in department ' ||

p\_department\_id || ' received a ' ||

p\_bonus\_percentage || '% bonus.');

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error updating employee bonuses: ' || SQLERRM);

END UpdateEmployeeBonus;

/

OUTPUT:

8 employees in department 10 received a 5% bonus*.*

**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(

p\_from\_account IN NUMBER,

p\_to\_account IN NUMBER,

p\_amount IN NUMBER

) IS

v\_from\_balance NUMBER;

v\_to\_account\_exists NUMBER;

BEGIN

-- Check if source account has sufficient balance

SELECT balance INTO v\_from\_balance

FROM accounts

WHERE account\_id = p\_from\_account

FOR UPDATE; -- Lock the row

IF v\_from\_balance < p\_amount THEN

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

END IF;

-- Check if destination account exists

SELECT COUNT(\*) INTO v\_to\_account\_exists

FROM accounts

WHERE account\_id = p\_to\_account;

IF v\_to\_account\_exists = 0 THEN

RAISE\_APPLICATION\_ERROR(-20002, 'Destination account does not exist');

END IF;

-- Perform the transfer

UPDATE accounts

SET balance = balance - p\_amount

WHERE account\_id = p\_from\_account;

UPDATE accounts

SET balance = balance + p\_amount

WHERE account\_id = p\_to\_account;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Successfully transferred ' || p\_amount ||

' from account ' || p\_from\_account ||

' to account ' || p\_to\_account);

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END TransferFunds;

/

OUTPUT:

Successfully transferred 500 from account 1001 to account 1002

Transfer failed: ORA-20001: Insufficient funds in source account

Transfer failed: ORA-20002: Destination account does not exist

**Exercise 1: Setting Up 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;

}

public int multiply(int a, int b) {

return a \* b;

}

}

// CalculatorTest.java

import org.junit.Test;

import static org.junit.Assert.\*;

public class CalculatorTest {

@Test

public void testAddition() {

Calculator calculator = new Calculator();

assertEquals(5, calculator.add(2, 3));

}

@Test

public void testSubtraction() {

Calculator calculator = new Calculator();

assertEquals(1, calculator.subtract(4, 3));

}

@Test

public void testMultiplication() {

Calculator calculator = new Calculator();

assertEquals(6, calculator.multiply(2, 3));

}

}

**Exercise 3: Assertions in Junit**

import org.junit.Test;

import static org.junit.Assert.\*;

public class AssertionsTest {

@Test

public void testVariousAssertions() {

// Equality assertions

assertEquals("Addition should work", 5, 2 + 3);

assertNotEquals(10, 2 + 3);

// Boolean assertions

assertTrue(5 > 3);

assertFalse(5 < 3);

// Null assertions

assertNull(null);

assertNotNull(new Object());

// Array assertions

assertArrayEquals(new int[]{1, 2, 3}, new int[]{1, 2, 3});

// Reference assertions

Object obj1 = new Object();

Object obj2 = obj1;

assertSame(obj1, obj2);

assertNotSame(new Object(), new Object());

// Exception assertion

try {

int result = 5 / 0;

fail("Should have thrown ArithmeticException");

} catch (ArithmeticException e) {

assertTrue(true); // Expected exception

}

}

}

**Exercise 4: AAA Pattern with Setup/Teardown**

import org.junit.\*;

import static org.junit.Assert.\*;

public class BankAccountTest {

private BankAccount account;

@BeforeClass

public static void setUpClass() {

System.out.println("This runs once before all tests");

}

@AfterClass

public static void tearDownClass() {

System.out.println("This runs once after all tests");

}

@Before

public void setUp() {

account = new BankAccount(1000);

System.out.println("Created new account with balance 1000");

}

@After

public void tearDown() {

System.out.println("Test completed. Current balance: " + account.getBalance());

}

@Test

public void testDeposit() {

// Arrange

double amount = 500;

// Act

account.deposit(amount);

// Assert

assertEquals(1500, account.getBalance(), 0.001);

}

@Test

public void testWithdraw() {

// Arrange

double amount = 300;

// Act

account.withdraw(amount);

// Assert

assertEquals(700, account.getBalance(), 0.001);

}

@Test(expected = IllegalArgumentException.class)

public void testWithdrawMoreThanBalance() {

account.withdraw(2000);

}

}

class BankAccount {

private double balance;

public BankAccount(double initialBalance) {

this.balance = initialBalance;

}

public void deposit(double amount) {

balance += amount;

}

public void withdraw(double amount) {

if (amount > balance) {

throw new IllegalArgumentException("Insufficient funds");

}

balance -= amount;

}

public double getBalance() {

return balance;

}

}

**Mockito Hands-On Exercises**

**Exercise 1: Mocking and Stubbing**

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

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

import org.mockito.Mockito;

// Interface for the external API

interface ExternalApi {

String getData();

void processData(String data);

}

// Service class that depends on ExternalApi

class MyService {

private final ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public String fetchData() {

return api.getData();

}

public void processImportantData() {

api.processData("Important Data");

}

}

// Test class

public class MyServiceTest {

@Test

public void testExternalApi() {

// 1. Create mock object

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

// 2. Stub the method

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

// 3. Use the mock in test

MyService service = new MyService(mockApi);

String result = service.fetchData();

// Verify

assertEquals("Mock Data", result);

// Optional: Verify interaction

verify(mockApi).getData();

}

}

**Exercise 2: Verifying Interactions**

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

import org.mockito.Mockito;

public class MyServiceTest {

@Test

public void testVerifyInteraction() {

// 1. Create mock object

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

// 2. Call the method with mock

MyService service = new MyService(mockApi);

service.processImportantData();

// 3. Verify the interaction

verify(mockApi).processData("Important Data");

// Additional verification examples:

verify(mockApi, times(1)).processData(anyString()); // Verify called once with any string

verify(mockApi, never()).getData(); // Verify getData was never called

}

}

**Exercise 1: Logging Error Messages and Warning Levels**

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class SLF4JLoggingExample {

// 1. Create logger instance

private static final Logger logger = LoggerFactory.getLogger(SLF4JLoggingExample.class);

public static void main(String[] args) {

// 2. Basic logging at different levels

logger.trace("This is a TRACE level message");

logger.debug("This is a DEBUG level message");

logger.info("This is an INFO level message");

logger.warn("This is a WARN level message");

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

// 3. Logging with variables

String user = "john\_doe";

int loginAttempts = 3;

logger.info("User {} has attempted login {} times", user, loginAttempts);

// 4. Logging exceptions

try {

simulateError();

} catch (Exception e) {

logger.error("An error occurred during processing", e);

}

// 5. Conditional logging

if (logger.isDebugEnabled()) {

logger.debug("This debug message has expensive calculation: {}", expensiveCalculation());

}

}

private static void simulateError() throws Exception {

throw new Exception("Simulated error for logging demonstration");

}

private static String expensiveCalculation() {

// Simulate expensive operation

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

return "Expensive result";

}

}

/\*

Required dependencies (add to pom.xml):

<dependencies>

<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>

</dependencies>

Create src/main/resources/logback.xml with:

<configuration>

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

<encoder>

<pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>

</encoder>

</appender>

<root level="INFO">

<appender-ref ref="CONSOLE"/>

</root>

</configuration>

\*/