**Exercise 1: Control Structures**

**Scenario 1:**Discount for senior citizens (age > 60)

BEGIN

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

FROM Customers c JOIN Loans l ON c.CustomerID = l.CustomerID) LOOP

IF MONTHS\_BETWEEN(SYSDATE, cust.DOB) / 12 > 60 THEN

UPDATE Loans

SET InterestRate = InterestRate - 1

WHERE LoanID = cust.LoanID;

END IF;

END LOOP;

COMMIT;

END;

**Scenario 2:** Promote to VIP based on balance

BEGIN

FOR cust IN (SELECT CustomerID, Balance FROM Customers) LOOP

IF cust.Balance > 10000 THEN

UPDATE Customers SET IsVIP = 'TRUE' WHERE CustomerID = cust.CustomerID;

END IF;

END LOOP;

COMMIT;

END;

**Scenario 3:** Send loan reminders

BEGIN

FOR loan\_rec IN (SELECT LoanID, CustomerID, EndDate

FROM Loans

WHERE EndDate BETWEEN SYSDATE AND SYSDATE + 30) LOOP

DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan ID ' || loan\_rec.LoanID ||

' for Customer ID ' || loan\_rec.CustomerID ||

' is due on ' || TO\_CHAR(loan\_rec.EndDate, 'DD-Mon-YYYY'));

END LOOP;

END;

**Exercise 2: Error Handling**

**Scenario 1: Safe fund transfer**

CREATE OR REPLACE PROCEDURE SafeTransferFunds(p\_from IN NUMBER, p\_to IN NUMBER, p\_amount IN NUMBER) IS

insufficient\_funds EXCEPTION;

BEGIN

DECLARE

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance FROM Accounts WHERE AccountID = p\_from;

IF v\_balance < p\_amount THEN

RAISE insufficient\_funds;

END IF;

UPDATE Accounts SET Balance = Balance - p\_amount WHERE AccountID = p\_from;

UPDATE Accounts SET Balance = Balance + p\_amount WHERE AccountID = p\_to;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Transfer successful.');

EXCEPTION

WHEN insufficient\_funds THEN

DBMS\_OUTPUT.PUT\_LINE('Error: Insufficient funds.');

ROLLBACK;

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('Unknown error: ' || SQLERRM);

ROLLBACK;

END;

END;

**Scenario 3: Add customer safely**

CREATE OR REPLACE PROCEDURE AddNewCustomer(p\_id IN NUMBER, p\_name IN VARCHAR2, p\_dob IN DATE, p\_balance IN NUMBER) IS

BEGIN

INSERT INTO Customers(CustomerID, Name, DOB, Balance, LastModified)

VALUES (p\_id, p\_name, p\_dob, p\_balance, SYSDATE);

COMMIT;

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

DBMS\_OUTPUT.PUT\_LINE('Customer already exists.');

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('Something went wrong: ' || SQLERRM);

ROLLBACK;

END;

**Exercise 3: Stored Procedures**

**Scenario 1: Process interest for savings**

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

BEGIN

UPDATE Accounts

SET Balance = Balance + (Balance \* 0.01)

WHERE AccountType = 'Savings';

COMMIT;

END;

**Scenario 2:** Add bonus to department

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(p\_dept IN VARCHAR2, p\_bonus IN NUMBER) IS

BEGIN

UPDATE Employees

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

WHERE Department = p\_dept;

COMMIT;

END;

**Scenario 3: Transfer funds between customer’s accounts**

CREATE OR REPLACE PROCEDURE TransferFunds(p\_from IN NUMBER, p\_to IN NUMBER, p\_amt IN NUMBER) IS

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance FROM Accounts WHERE AccountID = p\_from;

IF v\_balance >= p\_amt THEN

UPDATE Accounts SET Balance = Balance - p\_amt WHERE AccountID = p\_from;

UPDATE Accounts SET Balance = Balance + p\_amt WHERE AccountID = p\_to;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Transfer Done');

ELSE

DBMS\_OUTPUT.PUT\_LINE('Insufficient Balance');

END IF;

EXCEPTION

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('Error during transfer: ' || SQLERRM);

ROLLBACK;

END;

**Exercise 4: Functions**

**Scenario 1:** Calculate age

CREATE OR REPLACE FUNCTION CalculateAge(p\_dob IN DATE) RETURN NUMBER IS

BEGIN

RETURN TRUNC(MONTHS\_BETWEEN(SYSDATE, p\_dob) / 12);

END;

**Scenario 2:** EMI calculator

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(p\_loan IN NUMBER, p\_rate IN NUMBER, p\_years IN NUMBER)

RETURN NUMBER IS

v\_monthly NUMBER;

r NUMBER := p\_rate / (12 \* 100); -- monthly rate

n NUMBER := p\_years \* 12;

BEGIN

v\_monthly := (p\_loan \* r \* POWER(1 + r, n)) / (POWER(1 + r, n) - 1);

RETURN v\_monthly;

END;

**Scenario 3:** Check balance

CREATE OR REPLACE FUNCTION HasSufficientBalance(p\_acc IN NUMBER, p\_amt IN NUMBER) RETURN BOOLEAN IS

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance FROM Accounts WHERE AccountID = p\_acc;

RETURN v\_balance >= p\_amt;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN FALSE;

END;

**Exercise 5: Triggers**

**Scenario 1: Update LastModified**

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON Customers

FOR EACH ROW

BEGIN

:NEW.LastModified := SYSDATE;

END;

**Scenario 2**: Log transaction

CREATE TABLE AuditLog (

LogID NUMBER PRIMARY KEY,

AccountID NUMBER,

LogDate DATE,

Amount NUMBER,

Action VARCHAR2(20)

);

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog (LogID, AccountID, LogDate, Amount, Action)

VALUES (AuditLog\_seq.NEXTVAL, :NEW.AccountID, SYSDATE, :NEW.Amount, :NEW.TransactionType);

END;

**Scenario 3**: Validate transaction rules

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

DECLARE

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance FROM Accounts WHERE AccountID = :NEW.AccountID;

IF :NEW.TransactionType = 'Withdrawal' AND :NEW.Amount > v\_balance THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Not enough balance.');

ELSIF :NEW.TransactionType = 'Deposit' AND :NEW.Amount <= 0 THEN

RAISE\_APPLICATION\_ERROR(-20002, 'Invalid deposit amount.');

END IF;

END;

**Exercise 6: Cursors**

**Scenario 1: Monthly statement**

DECLARE

CURSOR c IS SELECT \* FROM Transactions WHERE TransactionDate BETWEEN TRUNC(SYSDATE, 'MM') AND LAST\_DAY(SYSDATE);

BEGIN

FOR tx IN c LOOP

DBMS\_OUTPUT.PUT\_LINE('Account: ' || tx.AccountID || ', Type: ' || tx.TransactionType || ' Amount: ' || tx.Amount);

END LOOP;

END;

**Scenario 2:** Apply annual fee

DECLARE

CURSOR acc\_cursor IS SELECT AccountID, Balance FROM Accounts;

BEGIN

FOR acc IN acc\_cursor LOOP

UPDATE Accounts SET Balance = Balance - 100 WHERE AccountID = acc.AccountID;

END LOOP;

COMMIT;

END;

**Scenario 3:** Update loan interest

DECLARE

CURSOR loan\_cursor IS SELECT LoanID, InterestRate FROM Loans;

BEGIN

FOR loan IN loan\_cursor LOOP

UPDATE Loans SET InterestRate = loan.InterestRate + 0.5 WHERE LoanID = loan.LoanID;

END LOOP;

COMMIT;

END;

**Exercise 7: Packages**

**Scenario 1**: Customer package

CREATE OR REPLACE PACKAGE CustomerManagement AS

PROCEDURE AddCustomer(...);

PROCEDURE UpdateCustomer(...);

FUNCTION GetBalance(p\_id NUMBER) RETURN NUMBER;

END;

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

-- Procedure and function bodies here (reuse above ones)

END;

**Scenario 2: Employee package**

CREATE OR REPLACE PACKAGE EmployeeManagement AS

PROCEDURE HireEmployee(...);

PROCEDURE UpdateEmployee(...);

FUNCTION GetAnnualSalary(p\_id NUMBER) RETURN NUMBER;

END;

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

-- Implementations here

END;

**Unit Testing**

**Exercise 1: Setting Up Junit**

**Scenario: Set up JUnit in a Java project**

import org.junit.Test;

public class HelloTest {

@Test

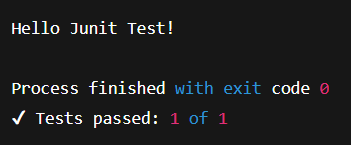
public void sayHelloTest() {

System.out.println("Hello Junit Test!");

}

}

Output:



**Exercise 2: Writing Basic JUnit Tests**

Calculator.java

public class Calculator {

public int add(int a, int b) {

return a + b;

}

public int multiply(int a, int b) {

return a \* b;

}

public int divide(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 res = calc.add(2, 3);

assertEquals(5, res);

}

@Test

public void testMultiply() {

Calculator calc = new Calculator();

int result = calc.multiply(3, 3);

assertEquals(9, result);

}

@Test

public void testDivide() {

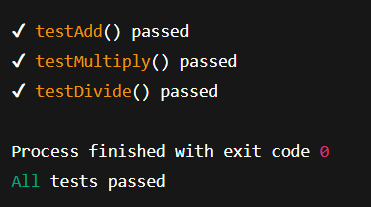
Calculator calc = new Calculator();

int div = calc.divide(10, 5);

assertEquals(2, div);

}

}

}

**Exercise 3: Assertions in Junit**

AssertionsTest.java

import org.junit.Test;

import static org.junit.Assert.\*;

public class AssertionsTest {

@Test

public void testAssertions() {

assertEquals(5, 2 + 3);

assertTrue(5 > 3);

assertFalse(3 > 5);

String name = null;

assertNull(name);

Object obj = new Object();

assertNotNull(obj);

}

}

Output:

**testAssertions() passed**

**Process finished with exit code 0**

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

**BankAccount.java**

public class BankAccount {

private int bal = 0;

public void deposit(int amt) {

bal += amt; // add money

}

public void withdraw(int amt) {

bal -= amt; // subtract

}

public int getBalance() {

return bal;

}

}

**BankTest.java**

import org.junit.Before;

import org.junit.After;

import org.junit.Test;

import static org.junit.Assert.\*;

public class BankTest {

BankAccount acc;

@Before

public void setUp() {

acc = new BankAccount();

acc.deposit(100);

System.out.println("acc created");

}

@After

public void tearDown() {

acc = null;

System.out.println("acc cleared");

}

@Test

public void testWithdraw() {

acc.withdraw(40);

assertEquals(60, acc.getBalance());

}

@Test

public void testDeposit() {

acc.deposit(50);

assertEquals(150, acc.getBalance());

}

}

Ooutput:

