**Exercise 1: Control Structures**

# Scenario 1: Applying a Discount to Loan Interest Rates for Customers Above 60 Years Old

## DECLARE

CURSOR customer\_cursor IS

SELECT CustomerID, DOB FROM Customers;

v\_CustomerID Customers.CustomerID%TYPE; v\_DOB Customers.DOB%TYPE;

v\_Age NUMBER;

## BEGIN

FOR customer\_record IN customer\_cursor LOOP v\_CustomerID := customer\_record.CustomerID; v\_DOB := customer\_record.DOB;

-- Calculate age

v\_Age := FLOOR((SYSDATE - v\_DOB) / 365.25);

IF v\_Age > 60 THEN

-- Apply 1% discount to loan interest rates UPDATE Loans

SET InterestRate = InterestRate \* 0.99 WHERE CustomerID = v\_CustomerID;

END IF;

END LOOP;

COMMIT; END;

# Scenario 2: Promoting Customers to VIP Status Based on Balance

ALTER TABLE Customers ADD (IsVIP VARCHAR2(3)); DECLARE

CURSOR customer\_cursor IS

SELECT CustomerID, Balance FROM Customers;

v\_CustomerID Customers.CustomerID%TYPE; v\_Balance Customers.Balance%TYPE;

## BEGIN

FOR customer\_record IN customer\_cursor LOOP v\_CustomerID := customer\_record.CustomerID;

v\_Balance := customer\_record.Balance;

IF v\_Balance > 10000 THEN

-- Set IsVIP flag to TRUE UPDATE Customers

SET IsVIP = 'TRUE'

WHERE CustomerID = v\_CustomerID; END IF;

END LOOP;

COMMIT; END;

# Scenario 3: Sending Reminders for Loans Due Within the Next 30 Days

## DECLARE

CURSOR loan\_cursor IS

SELECT LoanID, CustomerID, EndDate FROM Loans WHERE EndDate BETWEEN SYSDATE AND SYSDATE + 30;

v\_LoanID Loans.LoanID%TYPE; v\_CustomerID Loans.CustomerID%TYPE; v\_EndDate Loans.EndDate%TYPE;

v\_CustomerName Customers.Name%TYPE; BEGIN

FOR loan\_record IN loan\_cursor LOOP v\_LoanID := loan\_record.LoanID;

v\_CustomerID := loan\_record.CustomerID; v\_EndDate := loan\_record.EndDate;

-- Fetch customer's name

SELECT Name INTO v\_CustomerName FROM Customers

WHERE CustomerID = v\_CustomerID;

-- Print reminder message

DBMS\_OUTPUT.PUT\_LINE('Reminder: Customer ' || v\_CustomerName || ' (CustomerID: ' || v\_CustomerID ||

') has a loan (LoanID: ' || v\_LoanID ||

') due on ' || TO\_CHAR(v\_EndDate, 'DD-MON-YYYY') || '.');

## END LOOP;

END;

**Exercise 2: Error Handling**

# Scenario 1: Handling Exceptions During Fund Transfers Between Accounts

CREATE OR REPLACE PROCEDURE SafeTransferFunds ( p\_FromAccountID IN NUMBER,

p\_ToAccountID IN NUMBER, p\_Amount IN NUMBER

## ) IS

v\_FromBalance NUMBER; v\_ToBalance NUMBER;

## BEGIN

-- Check balance of from account SELECT Balance INTO v\_FromBalance FROM Accounts

WHERE AccountID = p\_FromAccountID FOR UPDATE;

IF v\_FromBalance < p\_Amount THEN

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

-- Deduct amount from source account UPDATE Accounts

SET Balance = Balance - p\_Amount, LastModified = SYSDATE

WHERE AccountID = p\_FromAccountID;

-- Add amount to destination account UPDATE Accounts

SET Balance = Balance + p\_Amount, LastModified = SYSDATE

WHERE AccountID = p\_ToAccountID; COMMIT;

## EXCEPTION

WHEN OTHERS THEN ROLLBACK;

-- Log the error message

INSERT INTO ErrorLog (ErrorMessage, ErrorDate) VALUES (SQLERRM, SYSDATE);

RAISE;

END;

# Scenario 2: Managing Errors When Updating Employee Salaries

CREATE OR REPLACE PROCEDURE UpdateSalary (

p\_EmployeeID IN NUMBER, p\_Percentage IN NUMBER

## ) IS

v\_Salary NUMBER;

## BEGIN

BEGIN

-- Fetch current salary SELECT Salary INTO v\_Salary FROM Employees

WHERE EmployeeID = p\_EmployeeID FOR UPDATE;

-- Update salary UPDATE Employees

SET Salary = Salary \* (1 + p\_Percentage / 100), LastModified = SYSDATE

WHERE EmployeeID = p\_EmployeeID;

## COMMIT; EXCEPTION

WHEN NO\_DATA\_FOUND THEN

-- Log the error message

INSERT INTO ErrorLog (ErrorMessage, ErrorDate)

VALUES ('Employee ID ' || p\_EmployeeID || ' not found.', SYSDATE); WHEN OTHERS THEN

## ROLLBACK;

-- Log the error message

INSERT INTO ErrorLog (ErrorMessage, ErrorDate) VALUES (SQLERRM, SYSDATE);

RAISE;

END;

END;

/

# Scenario 3: Ensuring Data Integrity When Adding a New Customer

CREATE OR REPLACE PROCEDURE AddNewCustomer (

p\_CustomerID IN NUMBER, p\_Name IN VARCHAR2, p\_DOB IN DATE,

p\_Balance IN NUMBER

## ) IS BEGIN

BEGIN

-- Insert new customer

INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified) VALUES (p\_CustomerID, p\_Name, p\_DOB, p\_Balance, SYSDATE);

## COMMIT; EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

-- Log the error message

INSERT INTO ErrorLog (ErrorMessage, ErrorDate)

VALUES ('Customer ID ' || p\_CustomerID || ' already exists.', SYSDATE); WHEN OTHERS THEN

## ROLLBACK;

-- Log the error message

INSERT INTO ErrorLog (ErrorMessage, ErrorDate) VALUES (SQLERRM, SYSDATE);

RAISE;

END;

END;

/

# ErrorLog Table

CREATE TABLE ErrorLog (

ErrorID NUMBER PRIMARY KEY,

ErrorMessage VARCHAR2(4000), ErrorDate DATE

);

**Exercise 3: Stored Procedures**

# Scenario 1: Processing Monthly Interest for All Savings Accounts

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS CURSOR savings\_accounts\_cursor IS

SELECT AccountID, Balance FROM Accounts

WHERE AccountType = 'Savings' FOR UPDATE;

v\_AccountID Accounts.AccountID%TYPE; v\_Balance Accounts.Balance%TYPE;

v\_InterestRate CONSTANT NUMBER := 0.01; BEGIN

FOR account\_record IN savings\_accounts\_cursor LOOP v\_AccountID := account\_record.AccountID;

v\_Balance := account\_record.Balance;

-- Calculate new balance with interest

v\_Balance := v\_Balance \* (1 + v\_InterestRate);

-- Update account balance UPDATE Accounts

SET Balance = v\_Balance, LastModified = SYSDATE

WHERE AccountID = v\_AccountID; END LOOP;

## COMMIT; END;

/

# Scenario 2: Implementing a Bonus Scheme for Employees

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus ( p\_Department IN VARCHAR2,

p\_BonusPercentage IN NUMBER

## ) IS BEGIN

-- Update salary by adding bonus percentage UPDATE Employees

SET Salary = Salary \* (1 + p\_BonusPercentage / 100), LastModified = SYSDATE

WHERE Department = p\_Department;

## COMMIT; END;

/

# Scenario 3: Transferring Funds Between Customer Accounts

CREATE OR REPLACE PROCEDURE TransferFunds (

p\_FromAccountID IN NUMBER, p\_ToAccountID IN NUMBER, p\_Amount IN NUMBER

## ) IS

v\_FromBalance NUMBER; v\_ToBalance NUMBER;

## BEGIN

-- Check balance of the source account SELECT Balance INTO v\_FromBalance

FROM Accounts

WHERE AccountID = p\_FromAccountID FOR UPDATE;

IF v\_FromBalance < p\_Amount THEN

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

-- Deduct amount from source account UPDATE Accounts

SET Balance = Balance - p\_Amount, LastModified = SYSDATE

WHERE AccountID = p\_FromAccountID;

-- Add amount to destination account UPDATE Accounts

SET Balance = Balance + p\_Amount, LastModified = SYSDATE

WHERE AccountID = p\_ToAccountID; COMMIT;

## EXCEPTION

WHEN OTHERS THEN ROLLBACK;

-- Log the error message

INSERT INTO ErrorLog (ErrorMessage, ErrorDate) VALUES (SQLERRM, SYSDATE);

## RAISE;

END;

/

**Exercise 4: Functions**

# Scenario 1: Calculate the Age of Customers for Eligibility Checks

CREATE OR REPLACE FUNCTION CalculateAge ( p\_DOB DATE

## ) RETURN NUMBER IS

v\_Age NUMBER;

## BEGIN

-- Calculate age

v\_Age := FLOOR((SYSDATE - p\_DOB) / 365.25);

RETURN v\_Age;

## END;

/

# Scenario 2: Compute the Monthly Installment for a Loan

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment ( p\_LoanAmount NUMBER,

p\_InterestRate NUMBER, p\_LoanDurationYears NUMBER

## ) RETURN NUMBER IS

v\_MonthlyRate NUMBER;

v\_NumberOfPayments NUMBER; v\_MonthlyInstallment NUMBER;

## BEGIN

-- Convert annual interest rate to monthly and loan duration to number of months v\_MonthlyRate := p\_InterestRate / 12 / 100;

v\_NumberOfPayments := p\_LoanDurationYears \* 12;

-- Calculate monthly installment using the formula for an annuity

v\_MonthlyInstallment := p\_LoanAmount \* v\_MonthlyRate / (1 - POWER(1 + v\_MonthlyRate, -v\_NumberOfPayments));

RETURN v\_MonthlyInstallment; END;

/

# Scenario 3: Check if a Customer Has Sufficient Balance Before Making a Transaction

CREATE OR REPLACE FUNCTION HasSufficientBalance ( p\_AccountID NUMBER,

p\_Amount NUMBER

## ) RETURN BOOLEAN IS

v\_Balance NUMBER;

## BEGIN

-- Fetch account balance SELECT Balance INTO v\_Balance FROM Accounts

WHERE AccountID = p\_AccountID;

-- Check if balance is sufficient

IF v\_Balance >= p\_Amount THEN RETURN TRUE;

ELSE

RETURN FALSE; END IF;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN RETURN FALSE;

END;

/

**Exercise 5: Triggers**

# Scenario 1: Automatically Update the Last Modified Date When a Customer's Record is Updated

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified BEFORE UPDATE ON Customers

## FOR EACH ROW BEGIN

:NEW.LastModified := SYSDATE; END;

/

# Scenario 2: Maintain an Audit Log for All Transactions

CREATE TABLE AuditLog (

AuditID NUMBER PRIMARY KEY,

TransactionID NUMBER, AccountID NUMBER, TransactionDate DATE, Amount NUMBER,

TransactionType VARCHAR2(10), LogDate DATE

);

Now, create the trigger:

CREATE OR REPLACE TRIGGER LogTransaction AFTER INSERT ON Transactions

## FOR EACH ROW BEGIN

INSERT INTO AuditLog (

AuditID, TransactionID, AccountID, TransactionDate, Amount, TransactionType, LogDate

## ) VALUES (

AuditLog\_seq.NEXTVAL, -- Assuming a sequence named AuditLog\_seq exists

:NEW.TransactionID,

:NEW.AccountID,

:NEW.TransactionDate,

:NEW.Amount,

:NEW.TransactionType, SYSDATE

## ); END;

/

# Scenario 3: Enforce Business Rules on Deposits and Withdrawals

CREATE OR REPLACE TRIGGER CheckTransactionRules BEFORE INSERT ON Transactions

## FOR EACH ROW DECLARE

v\_Balance NUMBER;

## BEGIN

-- Fetch the current balance of the account SELECT Balance INTO v\_Balance

FROM Accounts

WHERE AccountID = :NEW.AccountID

## FOR UPDATE;

-- Check for deposit

IF :NEW.TransactionType = 'Deposit' THEN IF :NEW.Amount <= 0 THEN

RAISE\_APPLICATION\_ERROR(-20002, 'Deposit amount must be positive.'); END IF;

ELSIF :NEW.TransactionType = 'Withdrawal' THEN

-- Check for withdrawal

IF :NEW.Amount <= 0 THEN

RAISE\_APPLICATION\_ERROR(-20003, 'Withdrawal amount must be positive.'); ELSIF v\_Balance < :NEW.Amount THEN

RAISE\_APPLICATION\_ERROR(-20004, 'Insufficient funds for withdrawal.'); END IF;

## ELSE

RAISE\_APPLICATION\_ERROR(-20005, 'Invalid transaction type.'); END IF;

## END;

/

**Exercise 6: Cursors**

# Scenario 1: Generate Monthly Statements for All Customers

## DECLARE

CURSOR transactions\_cursor IS

SELECT CustomerID, AccountID, TransactionDate, Amount, TransactionType FROM Transactions

WHERE TransactionDate BETWEEN TRUNC(SYSDATE, 'MM') AND LAST\_DAY(SYSDATE);

v\_CustomerID Transactions.CustomerID%TYPE; v\_AccountID Transactions.AccountID%TYPE;

v\_TransactionDate Transactions.TransactionDate%TYPE; v\_Amount Transactions.Amount%TYPE;

v\_TransactionType Transactions.TransactionType%TYPE; BEGIN

OPEN transactions\_cursor; LOOP

FETCH transactions\_cursor INTO v\_CustomerID, v\_AccountID, v\_TransactionDate, v\_Amount, v\_TransactionType;

EXIT WHEN transactions\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Customer ID: ' || v\_CustomerID ||

## END LOOP;

', Account ID: ' || v\_AccountID ||

', Date: ' || TO\_CHAR(v\_TransactionDate, 'DD-MON-YYYY') || ', Amount: ' || v\_Amount ||

', Type: ' || v\_TransactionType);

CLOSE transactions\_cursor; END;

/

# Scenario 2: Apply Annual Fee to All Accounts

## DECLARE

CURSOR accounts\_cursor IS SELECT AccountID, Balance FROM Accounts

## FOR UPDATE;

v\_AccountID Accounts.AccountID%TYPE; v\_Balance Accounts.Balance%TYPE;

v\_AnnualFee CONSTANT NUMBER := 50; -- Annual fee amount BEGIN

OPEN accounts\_cursor; LOOP

FETCH accounts\_cursor INTO v\_AccountID, v\_Balance; EXIT WHEN accounts\_cursor%NOTFOUND;

-- Deduct annual fee

v\_Balance := v\_Balance - v\_AnnualFee;

-- Update account balance UPDATE Accounts

SET Balance = v\_Balance, LastModified = SYSDATE

WHERE CURRENT OF accounts\_cursor; END LOOP;

CLOSE accounts\_cursor;

## COMMIT; END;

/

# Scenario 3: Update the Interest Rate for All Loans Based on a New Policy

## DECLARE

CURSOR loans\_cursor IS SELECT LoanID, InterestRate FROM Loans

## FOR UPDATE;

v\_LoanID Loans.LoanID%TYPE;

v\_InterestRate Loans.InterestRate%TYPE;

v\_NewRateAdjustment CONSTANT NUMBER := 0.5; -- Example adjustment, could be an increase by 0.5%

## BEGIN

OPEN loans\_cursor; LOOP

FETCH loans\_cursor INTO v\_LoanID, v\_InterestRate; EXIT WHEN loans\_cursor%NOTFOUND;

-- Update interest rate based on the new policy v\_InterestRate := v\_InterestRate + v\_NewRateAdjustment;

-- Update loan interest rate UPDATE Loans

SET InterestRate = v\_InterestRate, LastModified = SYSDATE

WHERE CURRENT OF loans\_cursor; END LOOP;

CLOSE loans\_cursor;

## COMMIT; END;

/

**Exercise 7: Packages**

# Scenario 1: Grouping All Customer-Related Procedures and Functions into a Package

-- Package Specification

CREATE OR REPLACE PACKAGE CustomerManagement IS PROCEDURE AddNewCustomer(

p\_CustomerID IN NUMBER, p\_Name IN VARCHAR2,

p\_DOB IN DATE,

p\_Balance IN NUMBER

);

PROCEDURE UpdateCustomerDetails( p\_CustomerID IN NUMBER,

p\_Name IN VARCHAR2, p\_DOB IN DATE

);

FUNCTION GetCustomerBalance( p\_CustomerID IN NUMBER

## ) RETURN NUMBER;

END CustomerManagement;

/

-- Package Body

CREATE OR REPLACE PACKAGE BODY CustomerManagement IS PROCEDURE AddNewCustomer(

p\_CustomerID 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\_CustomerID, p\_Name, p\_DOB, p\_Balance, SYSDATE);

END AddNewCustomer;

PROCEDURE UpdateCustomerDetails( p\_CustomerID IN NUMBER,

p\_Name IN VARCHAR2, p\_DOB IN DATE

## ) IS BEGIN

UPDATE Customers SET Name = p\_Name,

DOB = p\_DOB,

LastModified = SYSDATE

WHERE CustomerID = p\_CustomerID; END UpdateCustomerDetails;

FUNCTION GetCustomerBalance( p\_CustomerID IN NUMBER

## ) RETURN NUMBER IS

v\_Balance NUMBER; BEGIN

SELECT Balance INTO v\_Balance FROM Customers

WHERE CustomerID = p\_CustomerID;

RETURN v\_Balance;

END GetCustomerBalance; END CustomerManagement;

/

# Scenario 2: Creating a Package to Manage Employee Data

-- Package Specification

CREATE OR REPLACE PACKAGE EmployeeManagement IS PROCEDURE HireEmployee(

p\_EmployeeID IN NUMBER, p\_Name IN VARCHAR2,

p\_Position IN VARCHAR2, p\_Salary IN NUMBER, p\_Department IN VARCHAR2, p\_HireDate IN DATE

);

PROCEDURE UpdateEmployeeDetails( p\_EmployeeID IN NUMBER,

p\_Name IN VARCHAR2,

p\_Position IN VARCHAR2, p\_Salary IN NUMBER, p\_Department IN VARCHAR2

);

FUNCTION CalculateAnnualSalary( p\_EmployeeID IN NUMBER

## ) RETURN NUMBER;

END EmployeeManagement;

/

-- Package Body

CREATE OR REPLACE PACKAGE BODY EmployeeManagement IS PROCEDURE HireEmployee(

p\_EmployeeID IN NUMBER,

p\_Name IN VARCHAR2,

p\_Position IN VARCHAR2, p\_Salary IN NUMBER, p\_Department IN VARCHAR2, p\_HireDate IN DATE

## ) IS BEGIN

INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate) VALUES (p\_EmployeeID, p\_Name, p\_Position, p\_Salary, p\_Department, p\_HireDate);

END HireEmployee;

PROCEDURE UpdateEmployeeDetails( p\_EmployeeID IN NUMBER,

p\_Name IN VARCHAR2,

p\_Position IN VARCHAR2, p\_Salary IN NUMBER, p\_Department IN VARCHAR2

## ) IS BEGIN

UPDATE Employees SET Name = p\_Name,

Position = p\_Position, Salary = p\_Salary,

Department = p\_Department, LastModified = SYSDATE

WHERE EmployeeID = p\_EmployeeID; END UpdateEmployeeDetails;

FUNCTION CalculateAnnualSalary( p\_EmployeeID IN NUMBER

## ) RETURN NUMBER IS

v\_Salary NUMBER;

## BEGIN

SELECT Salary \* 12 INTO v\_Salary FROM Employees

WHERE EmployeeID = p\_EmployeeID;

RETURN v\_Salary;

END CalculateAnnualSalary; END EmployeeManagement;

/

# Scenario 3: Grouping All Account-Related Operations into a Package

-- Package Specification

CREATE OR REPLACE PACKAGE AccountOperations IS PROCEDURE OpenAccount(

p\_AccountID IN NUMBER, p\_CustomerID IN NUMBER, p\_AccountType IN VARCHAR2, p\_Balance IN NUMBER

);

PROCEDURE CloseAccount( p\_AccountID IN NUMBER

);

FUNCTION GetTotalBalance( p\_CustomerID IN NUMBER

## ) RETURN NUMBER;

END AccountOperations;

/

-- Package Body

CREATE OR REPLACE PACKAGE BODY AccountOperations IS PROCEDURE OpenAccount(

p\_AccountID IN NUMBER, p\_CustomerID IN NUMBER, p\_AccountType IN VARCHAR2, p\_Balance IN NUMBER

## ) IS BEGIN

INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified) VALUES (p\_AccountID, p\_CustomerID, p\_AccountType, p\_Balance, SYSDATE);

END OpenAccount;

PROCEDURE CloseAccount( p\_AccountID IN NUMBER

## ) IS BEGIN

DELETE FROM Accounts

WHERE AccountID = p\_AccountID; END CloseAccount;

FUNCTION GetTotalBalance(

p\_CustomerID IN NUMBER

## ) RETURN NUMBER IS

v\_TotalBalance NUMBER; BEGIN

SELECT SUM(Balance) INTO v\_TotalBalance FROM Accounts

WHERE CustomerID = p\_CustomerID;

RETURN v\_TotalBalance; END GetTotalBalance;

END AccountOperations;

/