**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.

**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.

**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.

**Answer 1:**

DECLARE

-- Variable for age calculation

v\_age NUMBER;

BEGIN

-- Scenario 1: Apply 1% discount to loan interest rates for customers above 60

FOR rec1 IN (

SELECT l.LoanID, c.DOB

FROM Loans l

JOIN Customers c ON l.CustomerID = c.CustomerID

)

LOOP

v\_age := FLOOR(MONTHS\_BETWEEN(SYSDATE, rec1.DOB) / 12);

IF v\_age > 60 THEN

UPDATE Loans

SET InterestRate = InterestRate - 1

WHERE LoanID = rec1.LoanID;

END IF;

END LOOP;

-- Scenario 2: Set IsVIP to 'Y' for customers with balance over 10,000

FOR rec2 IN (SELECT CustomerID, Balance FROM Customers)

LOOP

IF rec2.Balance > 10000 THEN

UPDATE Customers

SET IsVIP = 'Y'

WHERE CustomerID = rec2.CustomerID;

END IF;

END LOOP;

-- Scenario 3: Print reminders for loans due within next 30 days

FOR rec3 IN (

SELECT l.LoanID, c.Name, l.EndDate

FROM Loans l

JOIN Customers c ON l.CustomerID = c.CustomerID

WHERE l.EndDate BETWEEN SYSDATE AND SYSDATE + 30

)

LOOP

DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan ' || rec3.LoanID || ' for customer ' || rec3.Name ||

' is due on ' || TO\_CHAR(rec3.EndDate, 'DD-MON-YYYY'));

END LOOP;

-- Commit all updates

COMMIT;

END;

/

**Exercise 2: Error Handling**

**Scenario 1:** Handle exceptions during fund transfers between accounts.

* + **Question:** Write a stored procedure **SafeTransferFunds** that transfers funds between two accounts. Ensure that if any error occurs (e.g., insufficient funds), an appropriate error message is logged and the transaction is rolled back.

**Scenario 2:** Manage errors when updating employee salaries.

* + **Question:** Write a stored procedure **UpdateSalary** that increases the salary of an employee by a given percentage. If the employee ID does not exist, handle the exception and log an error message.

**Scenario 3:** Ensure data integrity when adding a new customer.

* + **Question:** Write a stored procedure **AddNewCustomer** that inserts a new customer into the Customers table. If a customer with the same ID already exists, handle the exception by logging an error and preventing the insertion.

**Answer 2:**

-- Scenario 1: Handle exceptions during fund transfers between accounts

CREATE OR REPLACE PROCEDURE SafeTransferFunds (

p\_fromAccountID IN NUMBER,

p\_toAccountID IN NUMBER,

p\_amount IN NUMBER

) AS

v\_fromBalance NUMBER;

BEGIN

-- Get source account balance

SELECT Balance INTO v\_fromBalance FROM Accounts WHERE AccountID = p\_fromAccountID;

-- Check for sufficient funds

IF v\_fromBalance < p\_amount THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient funds for transfer.');

END IF;

-- Deduct from source account

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

-- Add to destination account

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

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END;

/

-- Scenario 2: Manage errors when updating employee salaries

CREATE OR REPLACE PROCEDURE UpdateSalary (

p\_employeeID IN NUMBER,

p\_percent IN NUMBER

) AS

BEGIN

-- Update salary

UPDATE Employees

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

WHERE EmployeeID = p\_employeeID;

-- Check if employee exists

IF SQL%ROWCOUNT = 0 THEN

RAISE\_APPLICATION\_ERROR(-20002, 'Employee ID not found.');

END IF;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END;

/

-- Scenario 3: Ensure 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

) AS

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

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Insertion failed: Customer with this ID already exists.');

WHEN OTHERS THEN

ROLLBACK;

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

END;

/

-- Sample calls (to test)

BEGIN

SafeTransferFunds(1, 2, 100);

UpdateSalary(1, 10);

AddNewCustomer(3, 'New Customer', TO\_DATE('1995-10-05', 'YYYY-MM-DD'), 3000);

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.

**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.

**Answer 3:**

-- Scenario 1: Process monthly interest for all savings accounts

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest AS

BEGIN

-- Apply 1% interest to all Savings accounts

UPDATE Accounts

SET Balance = Balance + (Balance \* 0.01)

WHERE AccountType = 'Savings';

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END;

/

-- Scenario 2: Implement bonus scheme for employees in a given department

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (

p\_department IN VARCHAR2,

p\_bonusPercent IN NUMBER

) AS

BEGIN

-- Update salary for employees in the given department

UPDATE Employees

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

WHERE Department = p\_department;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END;

/

-- Scenario 3: Customers can transfer funds between their accounts

CREATE OR REPLACE PROCEDURE TransferFunds (

p\_fromAccountID IN NUMBER,

p\_toAccountID IN NUMBER,

p\_amount IN NUMBER

) AS

v\_fromBalance NUMBER;

BEGIN

-- Check source account balance

SELECT Balance INTO v\_fromBalance FROM Accounts WHERE AccountID = p\_fromAccountID;

IF v\_fromBalance < p\_amount THEN

RAISE\_APPLICATION\_ERROR(-20003, 'Insufficient funds for transfer.');

END IF;

-- Deduct amount from source

UPDATE Accounts

SET Balance = Balance - p\_amount

WHERE AccountID = p\_fromAccountID;

-- Add amount to destination

UPDATE Accounts

SET Balance = Balance + p\_amount

WHERE AccountID = p\_toAccountID;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error transferring funds: ' || SQLERRM);

END;

/

-- Sample procedure calls (for testing)

BEGIN

ProcessMonthlyInterest;

UpdateEmployeeBonus('IT', 15);

TransferFunds(1, 2, 100);

END;

/

**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.

**Answer 4:**

-- Scenario 1: Function to calculate the age of a customer

CREATE OR REPLACE FUNCTION CalculateAge (

p\_dob IN DATE

) RETURN NUMBER AS

v\_age NUMBER;

BEGIN

v\_age := FLOOR(MONTHS\_BETWEEN(SYSDATE, p\_dob) / 12);

RETURN v\_age;

END;

/

-- Scenario 2: Function to compute monthly loan installment

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment (

p\_loanAmount IN NUMBER,

p\_interestRate IN NUMBER, -- Annual interest rate in %

p\_years IN NUMBER -- Loan duration in years

) RETURN NUMBER AS

v\_monthlyRate NUMBER;

v\_totalMonths NUMBER;

v\_installment NUMBER;

BEGIN

v\_monthlyRate := p\_interestRate / (12 \* 100);

v\_totalMonths := p\_years \* 12;

-- Using simple EMI formula: [P \* r \* (1 + r)^n] / [(1 + r)^n - 1]

v\_installment := (p\_loanAmount \* v\_monthlyRate \* POWER(1 + v\_monthlyRate, v\_totalMonths)) /

(POWER(1 + v\_monthlyRate, v\_totalMonths) - 1);

RETURN v\_installment;

END;

/

-- Scenario 3: Function to check sufficient balance before transaction

CREATE OR REPLACE FUNCTION HasSufficientBalance (

p\_accountID IN NUMBER,

p\_amount IN NUMBER

) RETURN BOOLEAN AS

v\_balance NUMBER;

BEGIN

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

IF v\_balance >= p\_amount THEN

RETURN TRUE;

ELSE

RETURN FALSE;

END IF;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN FALSE;

WHEN OTHERS THEN

RETURN FALSE;

END;

/

-- Example calls (in anonymous block to test functions)

DECLARE

v\_age NUMBER;

v\_installment NUMBER;

v\_hasBalance BOOLEAN;

BEGIN

-- Calculate age

SELECT CalculateAge(TO\_DATE('1990-07-20', 'YYYY-MM-DD')) INTO v\_age FROM dual;

DBMS\_OUTPUT.PUT\_LINE('Age: ' || v\_age);

-- Calculate monthly installment

v\_installment := CalculateMonthlyInstallment(50000, 8, 5);

DBMS\_OUTPUT.PUT\_LINE('Monthly Installment: ' || TO\_CHAR(ROUND(v\_installment, 2)));

-- Check sufficient balance

v\_hasBalance := HasSufficientBalance(1, 500);

IF v\_hasBalance THEN

DBMS\_OUTPUT.PUT\_LINE('Sufficient balance available.');

ELSE

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

END IF;

END;

/

**Exercise 5: Triggers**

**Scenario 1:** Automatically update the last modified date when a customer's record is updated.

* + **Question:** Write a trigger **UpdateCustomerLastModified** that updates the LastModified column of the Customers table to the current date whenever a customer's record is updated.

**Scenario 2:** Maintain an audit log for all transactions.

* + **Question:** Write a trigger **LogTransaction** that inserts a record into an AuditLog table whenever a transaction is inserted into the Transactions table.

**Scenario 3:** Enforce business rules on deposits and withdrawals.

* + **Question:** Write a trigger **CheckTransactionRules** that ensures withdrawals do not exceed the balance and deposits are positive before inserting a record into the Transactions table.

**Answer 5:**

-- Scenario 1: Automatically update LastModified 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

-- First, create the AuditLog table if it doesn't exist

BEGIN

EXECUTE IMMEDIATE 'CREATE TABLE AuditLog (

LogID NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,

TransactionID NUMBER,

LogDate DATE,

Details VARCHAR2(500)

)';

EXCEPTION

WHEN OTHERS THEN

IF SQLCODE != -955 THEN -- Ignore "table already exists" error

RAISE;

END IF;

END;

/

-- Now, create the trigger for logging transactions

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog (TransactionID, LogDate, Details)

VALUES (:NEW.TransactionID, SYSDATE, 'Transaction of type ' || :NEW.TransactionType ||

' with amount ' || :NEW.Amount || ' added.');

END;

/

-- Scenario 3: Enforce rules on deposits and withdrawals

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

DECLARE

v\_balance NUMBER;

BEGIN

-- Get current account balance

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

IF :NEW.TransactionType = 'Withdrawal' THEN

-- Check if withdrawal exceeds balance

IF :NEW.Amount > v\_balance THEN

RAISE\_APPLICATION\_ERROR(-20004, 'Withdrawal exceeds available balance.');

END IF;

ELSIF :NEW.TransactionType = 'Deposit' THEN

-- Ensure deposit is positive

IF :NEW.Amount <= 0 THEN

RAISE\_APPLICATION\_ERROR(-20005, 'Deposit amount must be positive.');

END IF;

END IF;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RAISE\_APPLICATION\_ERROR(-20006, 'Account does not exist.');

END;

/

**Exercise 6: Cursors**

**Scenario 1:** Generate monthly statements for all customers.

* + **Question:** Write a PL/SQL block using an explicit cursor **GenerateMonthlyStatements** that retrieves all transactions for the current month and prints a statement for each customer.

**Scenario 2:** Apply annual fee to all accounts.

* + **Question:** Write a PL/SQL block using an explicit cursor **ApplyAnnualFee** that deducts an annual maintenance fee from the balance of all accounts.

**Scenario 3:** Update the interest rate for all loans based on a new policy.

* + **Question:** Write a PL/SQL block using an explicit cursor **UpdateLoanInterestRates** that fetches all loans and updates their interest rates based on the new policy.

**Answer 6:**

-- Scenario 1: Generate monthly statements for all customers

DECLARE

CURSOR c\_transactions IS

SELECT t.TransactionID, a.CustomerID, t.TransactionDate, t.Amount, t.TransactionType

FROM Transactions t

JOIN Accounts a ON t.AccountID = a.AccountID

WHERE TO\_CHAR(t.TransactionDate, 'MM-YYYY') = TO\_CHAR(SYSDATE, 'MM-YYYY');

v\_rec c\_transactions%ROWTYPE;

BEGIN

OPEN c\_transactions;

LOOP

FETCH c\_transactions INTO v\_rec;

EXIT WHEN c\_transactions%NOTFOUND;

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

', Transaction ID: ' || v\_rec.TransactionID ||

', Date: ' || TO\_CHAR(v\_rec.TransactionDate, 'DD-MON-YYYY') ||

', Amount: ' || v\_rec.Amount ||

', Type: ' || v\_rec.TransactionType);

END LOOP;

CLOSE c\_transactions;

END;

/

-- Scenario 2: Apply annual fee to all accounts

DECLARE

CURSOR c\_accounts IS

SELECT AccountID, Balance FROM Accounts;

v\_acc c\_accounts%ROWTYPE;

v\_fee CONSTANT NUMBER := 200; -- Example fee

BEGIN

OPEN c\_accounts;

LOOP

FETCH c\_accounts INTO v\_acc;

EXIT WHEN c\_accounts%NOTFOUND;

-- Deduct annual fee

UPDATE Accounts

SET Balance = Balance - v\_fee

WHERE AccountID = v\_acc.AccountID;

END LOOP;

CLOSE c\_accounts;

COMMIT;

END;

/

-- Scenario 3: Update loan interest rates based on new policy

DECLARE

CURSOR c\_loans IS

SELECT LoanID, InterestRate FROM Loans;

v\_loan c\_loans%ROWTYPE;

BEGIN

OPEN c\_loans;

LOOP

FETCH c\_loans INTO v\_loan;

EXIT WHEN c\_loans%NOTFOUND;

-- Increase interest rate by 0.5%

UPDATE Loans

SET InterestRate = v\_loan.InterestRate + 0.5

WHERE LoanID = v\_loan.LoanID;

END LOOP;

CLOSE c\_loans;

COMMIT;

END;

/

**Exercise 7: Packages**

**Scenario 1:** Group all customer-related procedures and functions into a package.

* + **Question:** Create a package **CustomerManagement** with procedures for adding a new customer, updating customer details, and a function to get customer balance.

**Scenario 2:** Create a package to manage employee data.

* + **Question:** Write a package **EmployeeManagement** with procedures to hire new employees, update employee details, and a function to calculate annual salary.

**Scenario 3:** Group all account-related operations into a package.

* + **Question:** Create a package **AccountOperations** with procedures for opening a new account, closing an account, and a function to get the total balance of a customer across all accounts.

**Schema to be Created**

*CREATE TABLE Customers (*

*CustomerID NUMBER PRIMARY KEY,*

*Name VARCHAR2(100),*

*DOB DATE,*

*Balance NUMBER,*

*LastModified DATE*

*);*

*CREATE TABLE Accounts (*

*AccountID NUMBER PRIMARY KEY,*

*CustomerID NUMBER,*

*AccountType VARCHAR2(20),*

*Balance NUMBER,*

*LastModified DATE,*

*FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)*

*);*

*CREATE TABLE Transactions (*

*TransactionID NUMBER PRIMARY KEY,*

*AccountID NUMBER,*

*TransactionDate DATE,*

*Amount NUMBER,*

*TransactionType VARCHAR2(10),*

*FOREIGN KEY (AccountID) REFERENCES Accounts(AccountID)*

*);*

*CREATE TABLE Loans (*

*LoanID NUMBER PRIMARY KEY,*

*CustomerID NUMBER,*

*LoanAmount NUMBER,*

*InterestRate NUMBER,*

*StartDate DATE,*

*EndDate DATE,*

*FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)*

*);*

*CREATE TABLE Employees (*

*EmployeeID NUMBER PRIMARY KEY,*

*Name VARCHAR2(100),*

*Position VARCHAR2(50),*

*Salary NUMBER,*

*Department VARCHAR2(50),*

*HireDate DATE*

*);*

**Example Scripts for Sample Data Insertion**

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

*VALUES (1, 'John Doe', TO\_DATE('1985-05-15', 'YYYY-MM-DD'), 1000, SYSDATE);*

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

*VALUES (2, 'Jane Smith', TO\_DATE('1990-07-20', 'YYYY-MM-DD'), 1500, SYSDATE);*

*INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified)*

*VALUES (1, 1, 'Savings', 1000, SYSDATE);*

*INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified)*

*VALUES (2, 2, 'Checking', 1500, SYSDATE);*

*INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)*

*VALUES (1, 1, SYSDATE, 200, 'Deposit');*

*INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)*

*VALUES (2, 2, SYSDATE, 300, 'Withdrawal');*

*INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate)*

*VALUES (1, 1, 5000, 5, SYSDATE, ADD\_MONTHS(SYSDATE, 60));*

*INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate)*

*VALUES (1, 'Alice Johnson', 'Manager', 70000, 'HR', TO\_DATE('2015-06-15', 'YYYY-MM-DD'));*

*INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate)*

*VALUES (2, 'Bob Brown', 'Developer', 60000, 'IT', TO\_DATE('2017-03-20', 'YYYY-MM-DD'));*

**Answer 7:**

**Package 1: CustomerManagement**

-- Package Specification

CREATE OR REPLACE PACKAGE CustomerManagement AS

PROCEDURE AddNewCustomer(p\_customerID NUMBER, p\_name VARCHAR2, p\_dob DATE, p\_balance NUMBER);

PROCEDURE UpdateCustomerDetails(p\_customerID NUMBER, p\_name VARCHAR2, p\_balance NUMBER);

FUNCTION GetCustomerBalance(p\_customerID NUMBER) RETURN NUMBER;

END CustomerManagement;

/

-- Package Body

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

PROCEDURE AddNewCustomer(p\_customerID NUMBER, p\_name VARCHAR2, p\_dob DATE, p\_balance NUMBER) IS

BEGIN

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

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

COMMIT;

END;

PROCEDURE UpdateCustomerDetails(p\_customerID NUMBER, p\_name VARCHAR2, p\_balance NUMBER) IS

BEGIN

UPDATE Customers

SET Name = p\_name, Balance = p\_balance, LastModified = SYSDATE

WHERE CustomerID = p\_customerID;

COMMIT;

END;

FUNCTION GetCustomerBalance(p\_customerID NUMBER) RETURN NUMBER IS

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance FROM Customers WHERE CustomerID = p\_customerID;

RETURN v\_balance;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN 0;

END;

END CustomerManagement;

/

**Package 2: EmployeeManagement**

-- Package Specification

CREATE OR REPLACE PACKAGE EmployeeManagement AS

PROCEDURE HireEmployee(p\_employeeID NUMBER, p\_name VARCHAR2, p\_position VARCHAR2, p\_salary NUMBER, p\_department VARCHAR2, p\_hireDate DATE);

PROCEDURE UpdateEmployeeDetails(p\_employeeID NUMBER, p\_position VARCHAR2, p\_salary NUMBER);

FUNCTION CalculateAnnualSalary(p\_employeeID NUMBER) RETURN NUMBER;

END EmployeeManagement;

/

-- Package Body

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

PROCEDURE HireEmployee(p\_employeeID NUMBER, p\_name VARCHAR2, p\_position VARCHAR2, p\_salary NUMBER, p\_department VARCHAR2, p\_hireDate 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);

COMMIT;

END;

PROCEDURE UpdateEmployeeDetails(p\_employeeID NUMBER, p\_position VARCHAR2, p\_salary NUMBER) IS

BEGIN

UPDATE Employees

SET Position = p\_position, Salary = p\_salary

WHERE EmployeeID = p\_employeeID;

COMMIT;

END;

FUNCTION CalculateAnnualSalary(p\_employeeID NUMBER) RETURN NUMBER IS

v\_salary NUMBER;

BEGIN

SELECT Salary INTO v\_salary FROM Employees WHERE EmployeeID = p\_employeeID;

RETURN v\_salary \* 12;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN 0;

END;

END EmployeeManagement;

/

**Package 3: AccountOperations**

-- Package Specification

CREATE OR REPLACE PACKAGE AccountOperations AS

PROCEDURE OpenAccount(p\_accountID NUMBER, p\_customerID NUMBER, p\_accountType VARCHAR2, p\_balance NUMBER);

PROCEDURE CloseAccount(p\_accountID NUMBER);

FUNCTION GetTotalCustomerBalance(p\_customerID NUMBER) RETURN NUMBER;

END AccountOperations;

/

-- Package Body

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

PROCEDURE OpenAccount(p\_accountID NUMBER, p\_customerID NUMBER, p\_accountType VARCHAR2, p\_balance NUMBER) IS

BEGIN

INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified)

VALUES (p\_accountID, p\_customerID, p\_accountType, p\_balance, SYSDATE);

COMMIT;

END;

PROCEDURE CloseAccount(p\_accountID NUMBER) IS

BEGIN

DELETE FROM Accounts WHERE AccountID = p\_accountID;

COMMIT;

END;

FUNCTION GetTotalCustomerBalance(p\_customerID NUMBER) RETURN NUMBER IS

v\_totalBalance NUMBER;

BEGIN

SELECT NVL(SUM(Balance), 0) INTO v\_totalBalance FROM Accounts WHERE CustomerID = p\_customerID;

RETURN v\_totalBalance;

END;

END AccountOperations;

/