**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.

Solution:

DECLARE

    DISCOUNT NUMBER := 1;

BEGIN

    UPDATE LOANS

    SET INTERESTRATE=INTERESTRATE-DISCOUNT

    WHERE CUSTOMERID IN (

        SELECT CUSTOMERID FROM CUSTOMERS WHERE FLOOR(MONTHS\_BETWEEN(SYSDATE,DOB)/12)> 60

    );

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.

Solution:

ALTER TABLE CUSTOMERS ADD IsVIP VARCHAR(5) DEFAULT 'FALSE';

BEGIN

    FOR C\_RECORDS IN (SELECT CUSTOMERID,BALANCE FROM CUSTOMERS) LOOP

        IF C\_RECORDS.BALANCE>10000 THEN

            UPDATE CUSTOMERS

            SET IsVIP='TRUE'

            WHERE CUSTOMERID=C\_RECORDS.CUSTOMERID;

        ELSE

            UPDATE CUSTOMERS

            SET IsVIP='FALSE'

            WHERE CUSTOMERID=C\_RECORDS.CUSTOMERID;

        END IF;

    END LOOP;

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.

Solution:

BEGIN

    FOR REQ\_DATA IN (

        SELECT L.LOANID,L.CUSTOMERID,L.ENDDATE,C.NAME

        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 for Customer ' ||REQ\_DATA.NAME|| 'with ID: ' ||REQ\_DATA.CUSTOMERID|| ' has a loan (ID: ' ||REQ\_DATA.LOANID|| ') due on ' ||TO\_CHAR(REQ\_DATA.ENDDATE, 'DD-MON-YYYY')|| '.');

    END LOOP;

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.

Solution:

CREATE OR REPLACE PROCEDURE SafeTransferFunds (

    p\_FromAccountID IN NUMBER,

    p\_ToAccountID IN NUMBER,

    p\_Amount IN NUMBER

) AS

    v\_FromAccountBalance NUMBER;

    insufficient\_funds EXCEPTION;

BEGIN

    SELECT Balance INTO v\_FromAccountBalance

    FROM Accounts

    WHERE AccountID = p\_FromAccountID

    FOR UPDATE;

    IF v\_FromAccountBalance < p\_Amount THEN

        RAISE insufficient\_funds;

    END IF;

    UPDATE Accounts

    SET Balance = Balance - p\_Amount,

        LastModified = SYSDATE

    WHERE AccountID = p\_FromAccountID;

    UPDATE Accounts

    SET Balance = Balance + p\_Amount,

        LastModified = SYSDATE

    WHERE AccountID = p\_ToAccountID;

    COMMIT;

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

EXCEPTION

    WHEN insufficient\_funds THEN

        DBMS\_OUTPUT.PUT\_LINE('Error: Insufficient funds in account ' || p\_FromAccountID);

        ROLLBACK;

    WHEN OTHERS THEN

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

        ROLLBACK;

END SafeTransferFunds;

/

**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.

Solution:

CREATE OR REPLACE PROCEDURE UpdateSalary (

    p\_EmployeeID IN NUMBER,

    p\_Percentage IN NUMBER

) AS

    v\_CurrentSalary Employees.Salary%TYPE;

    v\_NewSalary Employees.Salary%TYPE;

    emp\_not\_found EXCEPTION;

BEGIN

    SELECT Salary INTO v\_CurrentSalary

    FROM Employees

    WHERE EmployeeID = p\_EmployeeID

    FOR UPDATE;

    v\_NewSalary := v\_CurrentSalary \* (1 + p\_Percentage / 100);

    UPDATE Employees

    SET Salary = v\_NewSalary,

        LastModified = SYSDATE

    WHERE EmployeeID = p\_EmployeeID;

    COMMIT;

    DBMS\_OUTPUT.PUT\_LINE('Salary updated successfully for Employee ID ' || p\_EmployeeID || '.');

EXCEPTION

    WHEN emp\_not\_found THEN

        DBMS\_OUTPUT.PUT\_LINE('Error: Employee ID ' || p\_EmployeeID || ' does not exist.');

        ROLLBACK;

END UpdateSalary;

/

**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.

Solution:

CREATE OR REPLACE PROCEDURE AddNewCustomer (

    p\_CustomerID IN NUMBER,

    p\_Name IN VARCHAR2,

    p\_DOB IN DATE,

    p\_Balance IN NUMBER

) AS

    v\_ExistingCustomer NUMBER;

    customer\_exists EXCEPTION;

BEGIN

    BEGIN

        SELECT CustomerID INTO v\_ExistingCustomer

        FROM Customers

        WHERE CustomerID = p\_CustomerID;

        RAISE customer\_exists;

    EXCEPTION

        WHEN NO\_DATA\_FOUND THEN

            NULL;

    END;

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

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

    COMMIT;

    DBMS\_OUTPUT.PUT\_LINE('Customer added successfully.');

EXCEPTION

    WHEN customer\_exists THEN

        DBMS\_OUTPUT.PUT\_LINE('Error: Customer ID ' || p\_CustomerID || ' already exists.');

        ROLLBACK;

END AddNewCustomer;

/

**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.

Solution:

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

BEGIN

    UPDATE Accounts

    SET Balance = Balance \* 1.01,

        LastModified = SYSDATE

    WHERE AccountType = 'Savings';

    COMMIT;

END ProcessMonthlyInterest;

/

**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.

Solution:

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (

    p\_Department IN VARCHAR2,

    p\_BonusPercentage IN NUMBER

) AS

    emp\_not\_found EXCEPTION;

BEGIN

    UPDATE Employees

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

        LastModified = SYSDATE

    WHERE Department = p\_Department;

    COMMIT;

    DBMS\_OUTPUT.PUT\_LINE('Employee Bonus Updated in department ' || p\_Department || ' successfully.');

END UpdateEmployeeBonus;

/

**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.

Solution:

CREATE OR REPLACE PROCEDURE TransferFunds (

    p\_FromAccountID IN NUMBER,

    p\_ToAccountID IN NUMBER,

    p\_Amount IN NUMBER

) AS

    v\_FromAccountBalance NUMBER;

    v\_ToAccountBalance NUMBER;

    insufficient\_funds EXCEPTION;

BEGIN

    BEGIN

        SELECT Balance INTO v\_FromAccountBalance

        FROM Accounts

        WHERE AccountID = p\_FromAccountID

        FOR UPDATE;

        IF v\_FromAccountBalance < p\_Amount THEN

            RAISE insufficient\_funds;

        END IF;

    END;

    BEGIN

        SELECT Balance INTO v\_ToAccountBalance

        FROM Accounts

        WHERE AccountID = p\_ToAccountID

        FOR UPDATE;

    END;

    UPDATE Accounts

    SET Balance = Balance - p\_Amount,

        LastModified = SYSDATE

    WHERE AccountID = p\_FromAccountID;

    UPDATE Accounts

    SET Balance = Balance + p\_Amount,

        LastModified = SYSDATE

    WHERE AccountID = p\_ToAccountID;

    COMMIT;

    DBMS\_OUTPUT.PUT\_LINE('Transfer of ' || p\_Amount || ' successful from account ' || p\_FromAccountID || ' to account ' || p\_ToAccountID || '.');

EXCEPTION

    WHEN insufficient\_funds THEN

        DBMS\_OUTPUT.PUT\_LINE('Error: Insufficient funds in account ' || p\_FromAccountID);

        ROLLBACK;

    WHEN OTHERS THEN

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

        ROLLBACK;

END TransferFunds;

/

**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.

Solution:

CREATE OR REPLACE FUNCTION CalculateAge (

    p\_DOB IN DATE

) RETURN NUMBER IS

    v\_Age NUMBER;

    v\_CurrentDate DATE := SYSDATE;

BEGIN

    v\_Age := FLOOR(MONTHS\_BETWEEN(v\_CurrentDate, p\_DOB) / 12);

    RETURN v\_Age;

END CalculateAge;

/

**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.

Solution:

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment (

    p\_LoanAmount IN NUMBER,

    p\_AnnualInterestRate IN NUMBER,

    p\_LoanDurationYears IN NUMBER

) RETURN NUMBER IS

    v\_MonthlyInterestRate NUMBER;

    v\_TotalPayments NUMBER;

    v\_MonthlyInstallment NUMBER;

BEGIN

    v\_MonthlyInterestRate := p\_AnnualInterestRate / 100 / 12;

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

    IF v\_MonthlyInterestRate > 0 THEN

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

    ELSE

        v\_MonthlyInstallment := p\_LoanAmount / v\_TotalPayments;

    END IF;

    RETURN v\_MonthlyInstallment;

END CalculateMonthlyInstallment;

/

**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.

Solution:

CREATE OR REPLACE FUNCTION HasSufficientBalance (

    p\_AccountID IN NUMBER,

    p\_Amount IN NUMBER

) RETURN BOOLEAN IS

    v\_Balance NUMBER;

BEGIN

    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;

END HasSufficientBalance;

/

**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.

Solution:

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.

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

Solution:

CREATE TABLE AuditLog (

    AuditID NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,

    TransactionID NUMBER,

    AccountID NUMBER,

    TransactionDate DATE,

    Amount NUMBER,

    TransactionType VARCHAR2(10),

    ActionDate DATE,

    ActionType VARCHAR2(10)

);

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

    INSERT INTO AuditLog (

        TransactionID,

        AccountID,

        TransactionDate,

        Amount,

        TransactionType,

        ActionDate,

        ActionType

    ) VALUES (

        :NEW.TransactionID,

        :NEW.AccountID,

        :NEW.TransactionDate,

        :NEW.Amount,

        :NEW.TransactionType,

        SYSDATE,

        'INSERT'

    );

END;

/

**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.

Solution:

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

DECLARE

    v\_AccountBalance NUMBER;

BEGIN

    SELECT Balance INTO v\_AccountBalance

    FROM Accounts

    WHERE AccountID = :NEW.AccountID

    FOR UPDATE;

    IF :NEW.TransactionType = 'Withdrawal' THEN

        IF :NEW.Amount > v\_AccountBalance THEN

            RAISE\_APPLICATION\_ERROR(-20001, 'Error: Withdrawal amount exceeds account balance.');

        END IF;

    END IF;

    IF :NEW.TransactionType = 'Deposit' THEN

        IF :NEW.Amount <= 0 THEN

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

        END IF;

    END IF;

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.

Solution:

DECLARE

    CURSOR GenerateMonthlyStatements IS

        SELECT t.TransactionID,

               t.AccountID,

               t.TransactionDate,

               t.Amount,

               t.TransactionType,

               a.CustomerID,

               c.Name

        FROM Transactions t

        JOIN Accounts a ON t.AccountID = a.AccountID

        JOIN Customers c ON a.CustomerID = c.CustomerID

        WHERE EXTRACT(YEAR FROM t.TransactionDate) = EXTRACT(YEAR FROM SYSDATE)

          AND EXTRACT(MONTH FROM t.TransactionDate) = EXTRACT(MONTH FROM SYSDATE)

        ORDER BY c.CustomerID, t.TransactionDate;

    r\_Transaction GenerateMonthlyStatements%ROWTYPE;

    v\_CurrentCustomerID NUMBER := NULL;

BEGIN

    OPEN GenerateMonthlyStatements;

    FETCH GenerateMonthlyStatements INTO r\_Transaction;

    WHILE GenerateMonthlyStatements%FOUND LOOP

        IF v\_CurrentCustomerID IS NULL OR v\_CurrentCustomerID != r\_Transaction.CustomerID THEN

            DBMS\_OUTPUT.PUT\_LINE('----------------------------------------------------');

            DBMS\_OUTPUT.PUT\_LINE('Statement for Customer: ' || r\_Transaction.Name);

            DBMS\_OUTPUT.PUT\_LINE('Customer ID: ' || r\_Transaction.CustomerID);

            DBMS\_OUTPUT.PUT\_LINE('Account ID: ' || r\_Transaction.AccountID);

            DBMS\_OUTPUT.PUT\_LINE('Month: ' || TO\_CHAR(SYSDATE, 'Month YYYY'));

            DBMS\_OUTPUT.PUT\_LINE('----------------------------------------------------');

            v\_CurrentCustomerID := r\_Transaction.CustomerID;

        END IF;

        DBMS\_OUTPUT.PUT\_LINE('Transaction ID: ' || r\_Transaction.TransactionID);

        DBMS\_OUTPUT.PUT\_LINE('Date: ' || TO\_CHAR(r\_Transaction.TransactionDate, 'DD-MON-YYYY'));

        DBMS\_OUTPUT.PUT\_LINE('Amount: ' || r\_Transaction.Amount);

        DBMS\_OUTPUT.PUT\_LINE('Type: ' || r\_Transaction.TransactionType);

        DBMS\_OUTPUT.PUT\_LINE('----------------------------------------------------');

        FETCH GenerateMonthlyStatements INTO r\_Transaction;

    END LOOP;

    CLOSE GenerateMonthlyStatements;

END;

/

**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.

Solution:

DECLARE

    v\_AnnualFee NUMBER := 50;

    CURSOR ApplyAnnualFee IS

        SELECT AccountID, Balance

        FROM Accounts

        FOR UPDATE;

    r\_Account ApplyAnnualFee%ROWTYPE;

BEGIN

    OPEN ApplyAnnualFee;

    FETCH ApplyAnnualFee INTO r\_Account;

    WHILE ApplyAnnualFee%FOUND LOOP

        IF r\_Account.Balance >= v\_AnnualFee THEN

            UPDATE Accounts

            SET Balance = Balance - v\_AnnualFee

            WHERE AccountID = r\_Account.AccountID;

        ELSE

            DBMS\_OUTPUT.PUT\_LINE('Account ID ' || r\_Account.AccountID || ' has insufficient balance for the annual fee.');

        END IF;

        FETCH ApplyAnnualFee INTO r\_Account;

    END LOOP;

    CLOSE ApplyAnnualFee;

    COMMIT;

END;

/

**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.

Solution:

DECLARE

    v\_InterestRateAdjustment NUMBER := 0.07;

    CURSOR UpdateLoanInterestRates IS

        SELECT LoanID, InterestRate

        FROM Loans

        FOR UPDATE;

    r\_Loan UpdateLoanInterestRates%ROWTYPE;

BEGIN

    OPEN UpdateLoanInterestRates;

    FETCH UpdateLoanInterestRates INTO r\_Loan;

    WHILE UpdateLoanInterestRates%FOUND LOOP

        UPDATE Loans

        SET InterestRate = InterestRate + v\_InterestRateAdjustment

        WHERE LoanID = r\_Loan.LoanID;

        FETCH UpdateLoanInterestRates INTO r\_Loan;

    END LOOP;

    CLOSE UpdateLoanInterestRates;

    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.

Solution:

CREATE OR REPLACE PACKAGE CustomerManagement AS

    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,

        p\_Balance IN NUMBER

    );

    FUNCTION GetCustomerBalance (

        p\_CustomerID IN NUMBER

    ) RETURN NUMBER;

END CustomerManagement;

/

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

    PROCEDURE AddNewCustomer (

        p\_CustomerID IN NUMBER,

        p\_Name IN VARCHAR2,

        p\_DOB IN DATE,

        p\_Balance IN NUMBER

    ) IS

    BEGIN

        BEGIN

            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

                DBMS\_OUTPUT.PUT\_LINE('Error: Customer with ID ' || p\_CustomerID || ' already exists.');

            WHEN OTHERS THEN

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

                ROLLBACK;

        END;

    END AddNewCustomer;

    PROCEDURE UpdateCustomerDetails (

        p\_CustomerID IN NUMBER,

        p\_Name IN VARCHAR2,

        p\_DOB IN DATE,

        p\_Balance IN NUMBER

    ) IS

    BEGIN

        BEGIN

            UPDATE Customers

            SET Name = p\_Name,

                DOB = p\_DOB,

                Balance = p\_Balance,

                LastModified = SYSDATE

            WHERE CustomerID = p\_CustomerID;

            COMMIT;

        EXCEPTION

            WHEN NO\_DATA\_FOUND THEN

                DBMS\_OUTPUT.PUT\_LINE('Error: Customer with ID ' || p\_CustomerID || ' does not exist.');

            WHEN OTHERS THEN

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

                ROLLBACK;

        END;

    END UpdateCustomerDetails;

    FUNCTION GetCustomerBalance (

        p\_CustomerID IN NUMBER

    ) RETURN NUMBER IS

        v\_Balance NUMBER;

    BEGIN

        BEGIN

            SELECT Balance INTO v\_Balance

            FROM Customers

            WHERE CustomerID = p\_CustomerID;

            RETURN v\_Balance;

        EXCEPTION

            WHEN NO\_DATA\_FOUND THEN

                DBMS\_OUTPUT.PUT\_LINE('Error: Customer with ID ' || p\_CustomerID || ' does not exist.');

                RETURN NULL;

            WHEN OTHERS THEN

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

                RETURN NULL;

        END;

    END GetCustomerBalance;

END CustomerManagement;

/

**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.

Solution:

CREATE OR REPLACE PACKAGE EmployeeManagement AS

    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;

/

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

    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

        BEGIN

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

            VALUES (p\_EmployeeID, p\_Name, p\_Position, p\_Salary, p\_Department, p\_HireDate);

            COMMIT;

        EXCEPTION

            WHEN DUP\_VAL\_ON\_INDEX THEN

                DBMS\_OUTPUT.PUT\_LINE('Error: Employee with ID ' || p\_EmployeeID || ' already exists.');

            WHEN OTHERS THEN

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

                ROLLBACK;

        END;

    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

        BEGIN

            UPDATE Employees

            SET Name = p\_Name,

                Position = p\_Position,

                Salary = p\_Salary,

                Department = p\_Department

            WHERE EmployeeID = p\_EmployeeID;

            COMMIT;

        EXCEPTION

            WHEN NO\_DATA\_FOUND THEN

                DBMS\_OUTPUT.PUT\_LINE('Error: Employee with ID ' || p\_EmployeeID || ' does not exist.');

            WHEN OTHERS THEN

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

                ROLLBACK;

        END;

    END UpdateEmployeeDetails;

    FUNCTION CalculateAnnualSalary (

        p\_EmployeeID IN NUMBER

    ) RETURN NUMBER IS

        v\_Salary NUMBER;

    BEGIN

        BEGIN

            SELECT Salary INTO v\_Salary

            FROM Employees

            WHERE EmployeeID = p\_EmployeeID;

            RETURN v\_Salary \* 12;

        EXCEPTION

            WHEN NO\_DATA\_FOUND THEN

                DBMS\_OUTPUT.PUT\_LINE('Error: Employee with ID ' || p\_EmployeeID || ' does not exist.');

                RETURN NULL;

            WHEN OTHERS THEN

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

                RETURN NULL;

        END;

    END CalculateAnnualSalary;

END EmployeeManagement;

/

**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.

Solution:

CREATE OR REPLACE PACKAGE AccountOperations AS

    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;

/

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

    PROCEDURE OpenAccount (

        p\_AccountID IN NUMBER,

        p\_CustomerID IN NUMBER,

        p\_AccountType IN VARCHAR2,

        p\_Balance IN NUMBER

    ) IS

    BEGIN

        BEGIN

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

            VALUES (p\_AccountID, p\_CustomerID, p\_AccountType, p\_Balance, SYSDATE);

            COMMIT;

        EXCEPTION

            WHEN DUP\_VAL\_ON\_INDEX THEN

                DBMS\_OUTPUT.PUT\_LINE('Error: Account with ID ' || p\_AccountID || ' already exists.');

            WHEN NO\_DATA\_FOUND THEN

                DBMS\_OUTPUT.PUT\_LINE('Error: Customer with ID ' || p\_CustomerID || ' does not exist.');

            WHEN OTHERS THEN

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

                ROLLBACK;

        END;

    END OpenAccount;

    PROCEDURE CloseAccount (

        p\_AccountID IN NUMBER

    ) IS

    BEGIN

        BEGIN

            DELETE FROM Accounts

            WHERE AccountID = p\_AccountID;

            COMMIT;

        EXCEPTION

            WHEN NO\_DATA\_FOUND THEN

                DBMS\_OUTPUT.PUT\_LINE('Error: Account with ID ' || p\_AccountID || ' does not exist.');

            WHEN OTHERS THEN

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

                ROLLBACK;

        END;

    END CloseAccount;

    FUNCTION GetTotalBalance (

        p\_CustomerID IN NUMBER

    ) RETURN NUMBER IS

        v\_TotalBalance NUMBER;

    BEGIN

        BEGIN

            SELECT SUM(Balance) INTO v\_TotalBalance

            FROM Accounts

            WHERE CustomerID = p\_CustomerID;

            RETURN NVL(v\_TotalBalance, 0);

        EXCEPTION

            WHEN NO\_DATA\_FOUND THEN

                DBMS\_OUTPUT.PUT\_LINE('Error: No accounts found for Customer ID ' || p\_CustomerID || '.');

                RETURN 0;

            WHEN OTHERS THEN

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

                RETURN 0;

        END;

    END GetTotalBalance;

END AccountOperations;

/