PL/SQL Assignment

**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 cust\_cursor IS

SELECT l.LoanID, l.InterestRate

FROM Customers c

INNER JOIN Loans l ON c.CustomerID = l.CustomerID

WHERE EXTRACT(YEAR FROM SYSDATE) - EXTRACT(YEAR FROM c.DOB) > 60;

BEGIN

FOR rec IN cust\_cursor LOOP

UPDATE Loans

SET InterestRate = rec.InterestRate - 1

WHERE LoanID = rec.LoanID;

DBMS\_OUTPUT.PUT\_LINE('1% interest rate discount applied to Loan ID: ' || rec.LoanID);

END LOOP;

COMMIT;

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.

ALTER TABLE Customers ADD (IsVIP CHAR(1));

DECLARE

CURSOR cust\_cursor IS

SELECT CustomerID, Balance

FROM Customers;

BEGIN

FOR rec IN cust\_cursor LOOP

IF rec.Balance > 10000 THEN

UPDATE Customers

SET IsVIP = 'Y'

WHERE CustomerID = rec.CustomerID;

DBMS\_OUTPUT.PUT\_LINE('Customer ID ' || rec.CustomerID || ' has been upgraded to VIP status.');

ELSE

UPDATE Customers

SET IsVIP = 'N'

WHERE CustomerID = rec.CustomerID;

END IF;

END LOOP;

COMMIT;

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.

DECLARE

CURSOR loan\_due\_cursor IS

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

FROM Loans l

INNER JOIN Customers c ON l.CustomerID = c.CustomerID

WHERE l.EndDate <= SYSDATE + 30;

BEGIN

FOR rec IN loan\_due\_cursor LOOP

DBMS\_OUTPUT.PUT\_LINE('Loan Reminder: Dear ' || rec.Name || ' (Customer ID: ' || rec.CustomerID || '), your loan with ID ' || rec.LoanID || ' is due on ' || TO\_CHAR(rec.EndDate, 'YYYY-MM-DD') || '.');

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.  
  
CREATE OR REPLACE PROCEDURE SafeTransferFunds (

p\_from\_account IN NUMBER,

p\_to\_account IN NUMBER,

p\_amount IN NUMBER

) AS

l\_balance NUMBER;

BEGIN

SELECT Balance INTO l\_balance

FROM Accounts

WHERE AccountID = p\_from\_account;

IF l\_balance < p\_amount THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient funds in account ' || p\_from\_account);

END IF;

UPDATE Accounts

SET Balance = Balance - p\_amount

WHERE AccountID = p\_from\_account;

UPDATE Accounts

SET Balance = Balance + p\_amount

WHERE AccountID = p\_to\_account;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

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

CREATE OR REPLACE PROCEDURE UpdateSalary (

p\_employee\_id IN NUMBER,

p\_percentage IN NUMBER

) AS

v\_rows\_updated NUMBER;

BEGIN

UPDATE Employees

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

WHERE EmployeeID = p\_employee\_id;

v\_rows\_updated := SQL%ROWCOUNT;

IF v\_rows\_updated = 0 THEN

RAISE\_APPLICATION\_ERROR(-20002, 'Employee with ID ' || p\_employee\_id || ' not found');

END IF;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Failed to update salary: ' || SQLERRM);

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.

CREATE OR REPLACE PROCEDURE AddNewCustomer(

p\_customer\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_dob IN DATE,

p\_balance IN NUMBER

) AS

BEGIN

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

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

COMMIT;

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

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Insertion failed: Customer ID ' || p\_customer\_id || ' already exists.');

WHEN OTHERS THEN

ROLLBACK;

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

RAISE;

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.

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest AS

BEGIN

UPDATE Accounts

SET Balance = Balance \* 1.01

WHERE AccountType = 'Savings';

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Interest has been applied to all savings accounts for this month.');

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Failed to process monthly interest: ' || SQLERRM);

RAISE;

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.

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (

p\_department IN VARCHAR2,

p\_bonus\_percentage IN NUMBER

) AS

BEGIN

UPDATE Employees

SET Salary = Salary \* (1 + p\_bonus\_percentage / 100)

WHERE Department = p\_department;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Applied a ' || p\_bonus\_percentage || '% bonus to all employees in the ' || p\_department || ' department.');

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

RAISE;

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.

CREATE OR REPLACE PROCEDURE TransferFunds (

p\_from\_account\_id IN NUMBER,

p\_to\_account\_id IN NUMBER,

p\_amount IN NUMBER

) AS

v\_source\_balance NUMBER;

v\_target\_balance NUMBER;

BEGIN

BEGIN

SELECT Balance INTO v\_source\_balance

FROM Accounts

WHERE AccountID = p\_from\_account\_id;

IF v\_source\_balance < p\_amount THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient funds in account ' || p\_from\_account\_id || '.');

END IF;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RAISE\_APPLICATION\_ERROR(-20002, 'Source account ' || p\_from\_account\_id || ' does not exist.');

END;

BEGIN

SELECT Balance INTO v\_target\_balance

FROM Accounts

WHERE AccountID = p\_to\_account\_id;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RAISE\_APPLICATION\_ERROR(-20003, 'Destination account ' || p\_to\_account\_id || ' does not exist.');

END;

UPDATE Accounts

SET Balance = Balance - p\_amount

WHERE AccountID = p\_from\_account\_id;

UPDATE Accounts

SET Balance = Balance + p\_amount

WHERE AccountID = p\_to\_account\_id;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Successfully transferred ' || p\_amount || ' from account ' || p\_from\_account\_id || ' to account ' || p\_to\_account\_id || '.');

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

RAISE;

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.

CREATE OR REPLACE FUNCTION CalculateAge (

p\_dob IN DATE

) RETURN NUMBER IS

v\_age NUMBER;

BEGIN

SELECT TRUNC(MONTHS\_BETWEEN(SYSDATE, p\_dob) / 12) INTO v\_age

FROM dual;

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.

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment (

p\_loan\_amount IN NUMBER,

p\_annual\_interest\_rate IN NUMBER,

p\_duration\_years IN NUMBER

) RETURN NUMBER IS

v\_monthly\_rate NUMBER;

v\_num\_installments NUMBER;

v\_monthly\_installment NUMBER;

BEGIN

v\_monthly\_rate := p\_annual\_interest\_rate / 100 / 12;

v\_num\_installments := p\_duration\_years \* 12;

IF v\_monthly\_rate = 0 THEN

v\_monthly\_installment := p\_loan\_amount / v\_num\_installments;

ELSE

v\_monthly\_installment := (p\_loan\_amount \* v\_monthly\_rate \* POWER(1 + v\_monthly\_rate, v\_num\_installments)) /

(POWER(1 + v\_monthly\_rate, v\_num\_installments) - 1);

END IF;

RETURN v\_monthly\_installment;

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.

CREATE OR REPLACE FUNCTION HasSufficientBalance (

p\_account\_id IN NUMBER,

p\_amount IN NUMBER

) RETURN BOOLEAN IS

v\_balance NUMBER;

BEGIN

BEGIN

SELECT Balance INTO v\_balance

FROM Accounts

WHERE AccountID = p\_account\_id;

IF v\_balance >= p\_amount THEN

RETURN TRUE;

ELSE

RETURN FALSE;

END IF;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN FALSE;

END;

EXCEPTION

WHEN OTHERS THEN

RETURN FALSE;

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.

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.

CREATE TABLE AuditLog (

AuditID NUMBER PRIMARY KEY,

TransactionID NUMBER,

ChangeDate DATE,

ChangeType VARCHAR2(50)

);

CREATE SEQUENCE AuditLogSeq

START WITH 1

INCREMENT BY 1

NOCACHE

NOCYCLE;

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog (AuditID, TransactionID, ChangeDate, ChangeType)

VALUES (AuditLogSeq.NEXTVAL, :NEW.TransactionID, 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.

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

DECLARE

v\_balance NUMBER;

BEGIN

IF :NEW.TransactionType = 'Withdrawal' THEN

SELECT Balance INTO v\_balance

FROM Accounts

WHERE AccountID = :NEW.AccountID;

IF v\_balance< :NEW.Amount THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient funds for withdrawal on account ' || :NEW.AccountID);

END IF;

ELSIF :NEW.TransactionType = 'Deposit' THEN

IF :NEW.Amount<= 0 THEN

RAISE\_APPLICATION\_ERROR(-20002, '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/SQLblock using an explicit cursor **GenerateMonthlyStatements** that retrieves all transactions for the current month and prints a statement for each customer.

DECLARE

CURSOR c\_transactions IS

SELECT

t.TransactionID,

t.AccountID,

a.CustomerID,

t.TransactionDate,

t.Amount,

t.TransactionType

FROM Transactions t

JOIN Accounts a ON t.AccountID = a.AccountID

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

AND EXTRACT(YEAR FROM t.TransactionDate) = EXTRACT(YEAR FROM SYSDATE);

v\_customer\_idAccounts.CustomerID%TYPE;

v\_statement VARCHAR2(4000);

BEGIN

FOR r\_transaction IN c\_transactions LOOP

v\_customer\_id := r\_transaction.CustomerID;

v\_statement := 'Customer ID: ' || v\_customer\_id || CHR(10) ||

'Transaction ID: ' || r\_transaction.TransactionID || CHR(10) ||

'Account ID: ' || r\_transaction.AccountID || CHR(10) ||

'Date: ' || TO\_CHAR(r\_transaction.TransactionDate, 'YYYY-MM-DD') || CHR(10) ||

'Amount: ' || r\_transaction.Amount || CHR(10) ||

'Type: ' || r\_transaction.TransactionType || CHR(10);

DBMS\_OUTPUT.PUT\_LINE(v\_statement);

END LOOP;

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.

DECLARE

CURSOR c\_accounts IS

SELECT AccountID, Balance

FROM Accounts;

v\_account\_idAccounts.AccountID%TYPE;

v\_balanceAccounts.Balance%TYPE;

v\_annual\_fee NUMBER := 50;

BEGIN

FOR r\_account IN c\_accounts LOOP

v\_account\_id := r\_account.AccountID;

v\_balance := r\_account.Balance;

UPDATE Accounts

SET Balance = Balance - v\_annual\_fee

WHERE AccountID = v\_account\_id;

DBMS\_OUTPUT.PUT\_LINE('Account ID: ' || v\_account\_id || ' - New Balance: ' || (v\_balance - v\_annual\_fee));

END LOOP;

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.

DECLARE

CURSOR c\_loans IS

SELECT LoanID, InterestRate

FROM Loans;

v\_loan\_idLoans.LoanID%TYPE;

v\_interest\_rateLoans.InterestRate%TYPE;

v\_new\_interest\_rate NUMBER;

BEGIN

FOR r\_loan IN c\_loans LOOP

v\_loan\_id := r\_loan.LoanID;

v\_interest\_rate := r\_loan.InterestRate;

v\_new\_interest\_rate := v\_interest\_rate \* 1.05;

UPDATE Loans

SET InterestRate = v\_new\_interest\_rate

WHERE LoanID = v\_loan\_id;

DBMS\_OUTPUT.PUT\_LINE('Loan ID: ' || v\_loan\_id || ' - New Interest Rate: ' || v\_new\_interest\_rate);

END LOOP;

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.

**Package Specification**

CREATE OR REPLACE PACKAGE CustomerManagement AS

PROCEDURE AddCustomer(

p\_CustomerID NUMBER,

p\_Name VARCHAR2,

p\_DOB DATE,

p\_Balance NUMBER);

PROCEDURE UpdateCustomerDetails(

p\_CustomerID NUMBER,

p\_Name VARCHAR2,

p\_DOB DATE,

p\_Balance NUMBER);

FUNCTION GetCustomerBalance(

p\_CustomerID NUMBER) RETURN NUMBER;

END CustomerManagement;

**Package Body**

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

PROCEDURE AddCustomer(

p\_CustomerID NUMBER,

p\_Name VARCHAR2,

p\_DOB DATE,

p\_Balance NUMBER)

AS

v\_count NUMBER;

BEGIN

SELECT COUNT(\*)

INTO v\_count

FROM Customers

WHERE CustomerID = p\_CustomerID;

IF v\_count = 0 THEN

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

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

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

ELSE

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

END IF;

EXCEPTION

WHEN OTHERS THEN

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

END AddCustomer;

PROCEDURE UpdateCustomerDetails(

p\_CustomerID NUMBER,

p\_Name VARCHAR2,

p\_DOB DATE,

p\_Balance NUMBER)

AS

BEGIN

UPDATE Customers

SET Name = p\_Name,

DOB = p\_DOB,

Balance = p\_Balance,

LastModified = SYSDATE

WHERE CustomerID = p\_CustomerID;

IF SQL%ROWCOUNT = 0 THEN

DBMS\_OUTPUT.PUT\_LINE('No customer found with ID: ' || p\_CustomerID);

ELSE

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

END IF;

EXCEPTION

WHEN OTHERS THEN

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

END UpdateCustomerDetails;

FUNCTION GetCustomerBalance(

p\_CustomerID NUMBER) RETURN NUMBER

AS

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 NULL;

WHEN OTHERS THEN

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

RETURN NULL;

END GetCustomerBalance;

END CustomerManagement;

**Function Call**

BEGIN

CustomerManagement.AddCustomer(1, 'John Doe', TO\_DATE('1985-05-15', 'YYYY-MM-DD'), 1000);

CustomerManagement.UpdateCustomerDetails(1, 'John Doe', TO\_DATE('1985-05-15', 'YYYY-MM-DD'), 1500);

DBMS\_OUTPUT.PUT\_LINE('Customer Balance: ' || CustomerManagement.GetCustomerBalance(1));

END;

**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.

**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\_Name VARCHAR2,

p\_Position VARCHAR2,

p\_Salary NUMBER,

p\_Department VARCHAR2,

p\_HireDate DATE);

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)

AS

v\_count NUMBER;

BEGIN

SELECT COUNT(\*)

INTO v\_count

FROM Employees

WHERE EmployeeID = p\_EmployeeID;

IF v\_count = 0 THEN

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

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

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

ELSE

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

END IF;

EXCEPTION

WHEN OTHERS THEN

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

END HireEmployee;

PROCEDURE UpdateEmployeeDetails(

p\_EmployeeID NUMBER,

p\_Name VARCHAR2,

p\_Position VARCHAR2,

p\_Salary NUMBER,

p\_Department VARCHAR2,

p\_HireDate DATE)

AS

BEGIN

UPDATE Employees

SET Name = p\_Name,

Position = p\_Position,

Salary = p\_Salary,

Department = p\_Department,

HireDate = p\_HireDate

WHERE EmployeeID = p\_EmployeeID;

IF SQL%ROWCOUNT = 0 THEN

DBMS\_OUTPUT.PUT\_LINE('No employee found with ID: ' || p\_EmployeeID);

ELSE

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

END IF;

EXCEPTION

WHEN OTHERS THEN

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

END UpdateEmployeeDetails;

FUNCTION CalculateAnnualSalary(

p\_EmployeeID NUMBER) RETURN NUMBER

AS

v\_Salary NUMBER;

v\_AnnualSalary NUMBER;

BEGIN

SELECT Salary INTO v\_Salary

FROM Employees

WHERE EmployeeID = p\_EmployeeID;

v\_AnnualSalary := v\_Salary \* 12;

RETURN v\_AnnualSalary;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN NULL;

WHEN OTHERS THEN

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

RETURN NULL;

END CalculateAnnualSalary;

END EmployeeManagement;

**Function Call**

BEGIN

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

EmployeeManagement.UpdateEmployeeDetails(1, 'Alice Johnson', 'Senior Manager', 80000, 'HR', TO\_DATE('2015-06-15', 'YYYY-MM-DD'));

DBMS\_OUTPUT.PUT\_LINE('Annual Salary: ' || EmployeeManagement.CalculateAnnualSalary(1));

END;

**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.

**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 GetTotalBalance(

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)

AS

v\_count NUMBER;

BEGIN

SELECT COUNT(\*)

INTO v\_count

FROM Accounts

WHERE AccountID = p\_AccountID;

IF v\_count = 0 THEN

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

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

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

ELSE

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

END IF;

EXCEPTION

WHEN OTHERS THEN

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

END OpenAccount;

PROCEDURE CloseAccount(

p\_AccountID NUMBER)

AS

v\_count NUMBER;

BEGIN

DELETE FROM Accounts

WHERE AccountID = p\_AccountID;

IF SQL%ROWCOUNT = 0 THEN

DBMS\_OUTPUT.PUT\_LINE('No account found with ID: ' || p\_AccountID);

ELSE

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

END IF;

EXCEPTION

WHEN OTHERS THEN

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

END CloseAccount;

FUNCTION GetTotalBalance(

p\_CustomerID NUMBER) RETURN NUMBER

AS

v\_TotalBalance NUMBER;

BEGIN

SELECT SUM(Balance) INTO v\_TotalBalance

FROM Accounts

WHERE CustomerID = p\_CustomerID;

RETURN v\_TotalBalance;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN 0;

WHEN OTHERS THEN

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

RETURN 0;

END GetTotalBalance;

END AccountOperations;

**Function Call**

BEGIN

AccountOperations.OpenAccount(1, 100, 'Savings', 5000);

AccountOperations.CloseAccount(1);

DBMS\_OUTPUT.PUT\_LINE('Total Balance: ' || AccountOperations.GetTotalBalance(100));

END;