**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

v\_CustomerAge NUMBER;

v\_CurrentDate DATE := SYSDATE;

BEGIN

FOR customer IN (SELECT CustomerID, DOB FROM Customers) LOOP

-- Calculate age

v\_CustomerAge := TRUNC(MONTHS\_BETWEEN(v\_CurrentDate, customer.DOB) / 12);

-- Check if customer is above 60 years old

IF v\_CustomerAge > 60 THEN

-- Update interest rate for all loans of this customer

UPDATE Loans

SET InterestRate = InterestRate - 1

WHERE CustomerID = customer.CustomerID;

END IF;

END LOOP;

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 VARCHAR2(3);

DECLARE

v\_VIPBalanceLimit NUMBER := 10000;

BEGIN

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

-- Check if customer's balance is over $10,000

IF customer.Balance > v\_VIPBalanceLimit THEN

-- Set IsVIP to 'TRUE'

UPDATE Customers

SET IsVIP = 'TRUE'

WHERE CustomerID = customer.CustomerID;

ELSE

-- Set IsVIP to 'FALSE'

UPDATE Customers

SET IsVIP = 'FALSE'

WHERE CustomerID = customer.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.

**DECLARE**

**v\_DueDate DATE := SYSDATE + 30;**

**CURSOR loans\_cursor IS**

**SELECT L.LoanID, L.CustomerID, L.EndDate, C.Name**

**FROM Loans L**

**JOIN Customers C ON L.CustomerID = C.CustomerID**

**WHERE L.EndDate <= v\_DueDate;**

**BEGIN**

**FOR loan\_rec IN loans\_cursor LOOP**

**-- Print reminder message**

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

**' for customer ' || loan\_rec.Name ||**

**' (Customer ID: ' || loan\_rec.CustomerID ||**

**') is due on ' || TO\_CHAR(loan\_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\_FromAccountID NUMBER,**

**p\_ToAccountID NUMBER,**

**p\_Amount NUMBER**

**) IS**

**v\_FromAccountBalance NUMBER;**

**v\_ToAccountBalance NUMBER;**

**ex\_InsufficientFunds EXCEPTION;**

**BEGIN**

**-- Start transaction**

**SAVEPOINT sp\_before\_transfer;**

**-- Get the balance of the from-account**

**SELECT Balance INTO v\_FromAccountBalance**

**FROM Accounts**

**WHERE AccountID = p\_FromAccountID**

**FOR UPDATE;**

**-- Check for sufficient funds**

**IF v\_FromAccountBalance < p\_Amount THEN**

**RAISE ex\_InsufficientFunds;**

**ELSE**

**-- Deduct from from-account**

**UPDATE Accounts**

**SET Balance = Balance - p\_Amount,**

**LastModified = SYSDATE**

**WHERE AccountID = p\_FromAccountID;**

**-- Add to to-account**

**UPDATE Accounts**

**SET Balance = Balance + p\_Amount,**

**LastModified = SYSDATE**

**WHERE AccountID = p\_ToAccountID;**

**COMMIT;**

**END IF;**

**EXCEPTION**

**WHEN ex\_InsufficientFunds THEN**

**ROLLBACK TO sp\_before\_transfer;**

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

**WHEN OTHERS THEN**

**ROLLBACK TO sp\_before\_transfer;**

**DBMS\_OUTPUT.PUT\_LINE('Error: An unexpected error occurred during the transfer. Transaction aborted.');**

**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\_EmployeeID NUMBER,**

**p\_PercentageIncrease NUMBER**

**) IS**

**v\_CurrentSalary NUMBER;**

**BEGIN**

**-- Get the current salary of the employee**

**SELECT Salary INTO v\_CurrentSalary**

**FROM Employees**

**WHERE EmployeeID = p\_EmployeeID**

**FOR UPDATE;**

**-- Update the salary**

**UPDATE Employees**

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

**LastModified = SYSDATE**

**WHERE EmployeeID = p\_EmployeeID;**

**COMMIT;**

**EXCEPTION**

**WHEN NO\_DATA\_FOUND THEN**

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

**WHEN OTHERS THEN**

**DBMS\_OUTPUT.PUT\_LINE('Error: An unexpected error occurred while updating salary.');**

**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\_CustomerID NUMBER,**

**p\_Name VARCHAR2,**

**p\_DOB DATE,**

**p\_Balance NUMBER**

**) IS**

**BEGIN**

**-- Attempt to insert the new customer**

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

**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.**

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

**COMMIT;**

**EXCEPTION**

**WHEN DUP\_VAL\_ON\_INDEX THEN**

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

**WHEN OTHERS THEN**

**DBMS\_OUTPUT.PUT\_LINE('Error: An unexpected error occurred while adding the new customer.');**

**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 IS

v\_InterestRate NUMBER := 0.01; -- 1% interest rate

BEGIN

FOR account IN (SELECT AccountID, Balance FROM Accounts WHERE AccountType = 'Savings') LOOP

-- Calculate new balance with interest

UPDATE Accounts

SET Balance = Balance + (Balance \* v\_InterestRate),

LastModified = SYSDATE

WHERE AccountID = account.AccountID;

END LOOP;

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.

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(

p\_Department VARCHAR2,

p\_BonusPercentage NUMBER

) IS

BEGIN

-- Update the salary of employees in the given department

UPDATE Employees

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

LastModified = SYSDATE

WHERE Department = p\_Department;

COMMIT;

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\_FromAccountID NUMBER,

p\_ToAccountID NUMBER,

p\_Amount NUMBER

) IS

v\_FromAccountBalance NUMBER;

BEGIN

-- Get the balance of the source account

SELECT Balance INTO v\_FromAccountBalance

FROM Accounts

WHERE AccountID = p\_FromAccountID

FOR UPDATE;

-- Check if the source account has sufficient balance

IF v\_FromAccountBalance < p\_Amount THEN

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

ELSE

-- Deduct from the source account

UPDATE Accounts

SET Balance = Balance - p\_Amount,

LastModified = SYSDATE

WHERE AccountID = p\_FromAccountID;

-- Add to the destination account

UPDATE Accounts

SET Balance = Balance + p\_Amount,

LastModified = SYSDATE

WHERE AccountID = p\_ToAccountID;

COMMIT;

END IF;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('Error: One of the accounts does not exist.');

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('Error: An unexpected error occurred during the fund transfer.');

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 DATE) RETURN NUMBER IS

v\_Age NUMBER;

BEGIN

v\_Age := TRUNC(MONTHS\_BETWEEN(SYSDATE, 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.

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(

p\_LoanAmount NUMBER,

p\_AnnualInterestRate NUMBER,

p\_LoanDurationYears NUMBER

) RETURN NUMBER IS

v\_MonthlyInstallment NUMBER;

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

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

BEGIN

IF v\_MonthlyInterestRate = 0 THEN

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

ELSE

v\_MonthlyInstallment := p\_LoanAmount \* v\_MonthlyInterestRate /

(1 - POWER(1 + v\_MonthlyInterestRate, -v\_NumberOfPayments));

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.

CREATE OR REPLACE FUNCTION HasSufficientBalance(

p\_AccountID NUMBER,

p\_Amount NUMBER

) RETURN BOOLEAN IS

v\_Balance NUMBER;

BEGIN

-- Retrieve the balance of the specified account

SELECT Balance INTO v\_Balance

FROM Accounts

WHERE AccountID = p\_AccountID;

-- Check if the account has sufficient balance

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 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 UpdateCustomerLastModified;

/

**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,

AccountID NUMBER,

TransactionDate DATE,

Amount NUMBER,

TransactionType VARCHAR2(10),

LogDate DATE

);

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, :NEW.TransactionID, :NEW.AccountID, :NEW.TransactionDate, :NEW.Amount, :NEW.TransactionType, SYSDATE);

END LogTransaction;

/

**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**

**-- Fetch the current balance of the account**

**SELECT Balance INTO v\_Balance**

**FROM Accounts**

**WHERE AccountID = :NEW.AccountID**

**FOR UPDATE;**

**-- Check if the transaction type is 'Withdrawal' and ensure it does not exceed the balance**

**IF :NEW.TransactionType = 'Withdrawal' THEN**

**IF v\_Balance < :NEW.Amount THEN**

**RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient balance for withdrawal.');**

**END IF;**

**ELSIF :NEW.TransactionType = 'Deposit' THEN**

**-- Ensure the deposit amount is positive**

**IF :NEW.Amount <= 0 THEN**

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

**END IF;**

**END IF;**

**END CheckTransactionRules;**

**/**

**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.**

**DECLARE**

**CURSOR c\_Transactions IS**

**SELECT c.CustomerID, c.Name, t.TransactionID, t.TransactionDate, t.Amount, t.TransactionType**

**FROM Customers c**

**JOIN Accounts a ON c.CustomerID = a.CustomerID**

**JOIN Transactions t ON a.AccountID = t.AccountID**

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

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

**v\_Transaction c\_Transactions%ROWTYPE;**

**BEGIN**

**OPEN c\_Transactions;**

**LOOP**

**FETCH c\_Transactions INTO v\_Transaction;**

**EXIT WHEN c\_Transactions%NOTFOUND;**

**-- Print or process the transaction details**

**DBMS\_OUTPUT.PUT\_LINE('Customer: ' || v\_Transaction.Name);**

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

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

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

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

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

**END LOOP;**

**CLOSE c\_Transactions;**

**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 c\_Accounts%ROWTYPE;**

**v\_AnnualFee NUMBER := 50; -- Set the annual fee amount**

**BEGIN**

**OPEN c\_Accounts;**

**LOOP**

**FETCH c\_Accounts INTO v\_Account;**

**EXIT WHEN c\_Accounts%NOTFOUND;**

**-- Apply the annual fee**

**UPDATE Accounts**

**SET Balance = Balance - v\_AnnualFee,**

**LastModified = SYSDATE**

**WHERE AccountID = v\_Account.AccountID;**

**END LOOP;**

**CLOSE c\_Accounts;**

**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 c\_Loans%ROWTYPE;**

**v\_NewInterestRate NUMBER := 4.5; -- Set the new interest rate based on the policy**

**BEGIN**

**OPEN c\_Loans;**

**LOOP**

**FETCH c\_Loans INTO v\_Loan;**

**EXIT WHEN c\_Loans%NOTFOUND;**

**-- Update the interest rate**

**UPDATE Loans**

**SET InterestRate = v\_NewInterestRate,**

**LastModified = SYSDATE**

**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.**

**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\_DOB DATE, p\_Balance NUMBER);**

**FUNCTION GetCustomerBalance(p\_CustomerID NUMBER) RETURN NUMBER;**

**END CustomerManagement;**

**/**

**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);**

**EXCEPTION**

**WHEN DUP\_VAL\_ON\_INDEX THEN**

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

**END AddNewCustomer;**

**PROCEDURE UpdateCustomerDetails(p\_CustomerID NUMBER, p\_Name VARCHAR2, p\_DOB DATE, p\_Balance NUMBER) IS**

**BEGIN**

**UPDATE Customers**

**SET Name = p\_Name, DOB = p\_DOB, Balance = p\_Balance, LastModified = SYSDATE**

**WHERE CustomerID = p\_CustomerID;**

**EXCEPTION**

**WHEN NO\_DATA\_FOUND THEN**

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

**END UpdateCustomerDetails;**

**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 NULL;**

**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.**

**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);**

**FUNCTION CalculateAnnualSalary(p\_EmployeeID NUMBER) RETURN NUMBER;**

**END EmployeeManagement;**

**/**

**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);**

**EXCEPTION**

**WHEN DUP\_VAL\_ON\_INDEX THEN**

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

**END HireEmployee;**

**PROCEDURE UpdateEmployeeDetails(p\_EmployeeID NUMBER, p\_Name VARCHAR2, p\_Position VARCHAR2, p\_Salary NUMBER, p\_Department VARCHAR2) IS**

**BEGIN**

**UPDATE Employees**

**SET Name = p\_Name, Position = p\_Position, Salary = p\_Salary, Department = p\_Department**

**WHERE EmployeeID = p\_EmployeeID;**

**EXCEPTION**

**WHEN NO\_DATA\_FOUND THEN**

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

**END UpdateEmployeeDetails;**

**FUNCTION CalculateAnnualSalary(p\_EmployeeID NUMBER) RETURN NUMBER IS**

**v\_Salary NUMBER;**

**BEGIN**

**SELECT Salary \* 12 INTO v\_Salary FROM Employees WHERE EmployeeID = p\_EmployeeID;**

**RETURN v\_Salary;**

**EXCEPTION**

**WHEN NO\_DATA\_FOUND THEN**

**RETURN NULL;**

**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.**

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;

/

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);

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

END OpenAccount;

PROCEDURE CloseAccount(p\_AccountID NUMBER) IS

BEGIN

DELETE FROM Accounts WHERE AccountID = p\_AccountID;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

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

END CloseAccount;

FUNCTION GetTotalCustomerBalance(p\_CustomerID NUMBER) RETURN NUMBER IS

v\_TotalBalance NUMBER := 0;

BEGIN

SELECT SUM(Balance) INTO v\_TotalBalance

FROM Accounts

WHERE CustomerID = p\_CustomerID;

RETURN NVL(v\_TotalBalance, 0);

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN 0;

END GetTotalCustomerBalance;

END AccountOperations;

/