**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 customer\_cursor IS

SELECT c.CustomerID, c.DOB, l.LoanID, l.InterestRate

FROM Customers c

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

v\_age NUMBER;

v\_new\_rate NUMBER;

BEGIN

FOR cust\_rec IN customer\_cursor LOOP

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

IF v\_age > 60 THEN

v\_new\_rate := cust\_rec.InterestRate - 1;

UPDATE Loans SET InterestRate = v\_new\_rate WHERE LoanID = cust\_rec.LoanID;

DBMS\_OUTPUT.PUT\_LINE('Applied 1% discount to customer ' || cust\_rec.CustomerID);

END IF;

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.

DECLARE

CURSOR customer\_cursor IS

SELECT CustomerID, Balance FROM Customers;

BEGIN

FOR cust\_rec IN customer\_cursor LOOP

IF cust\_rec.Balance > 10000 THEN

UPDATE Customers SET IsVIP = 'TRUE' WHERE CustomerID = cust\_rec.CustomerID;

DBMS\_OUTPUT.PUT\_LINE('Promoted customer ' || cust\_rec.CustomerID || ' to VIP');

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\_cursor IS

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

FROM Loans l

JOIN Customers c ON l.CustomerID = c.CustomerID

WHERE l.EndDate BETWEEN SYSDATE AND SYSDATE + 30;

BEGIN

FOR loan\_rec IN loan\_cursor LOOP

DBMS\_OUTPUT.PUT\_LINE('Reminder: Customer ' || loan\_rec.Name ||

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

') has a loan 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\_from\_account IN NUMBER,

p\_to\_account IN NUMBER,

p\_amount IN NUMBER

) AS

v\_from\_balance NUMBER;

v\_to\_balance NUMBER;

BEGIN

IF p\_amount <= 0 THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Transfer amount must be positive');

END IF;

SELECT Balance INTO v\_from\_balance FROM Accounts

WHERE AccountID = p\_from\_account FOR UPDATE;

SELECT Balance INTO v\_to\_balance FROM Accounts

WHERE AccountID = p\_to\_account FOR UPDATE;

IF v\_from\_balance < p\_amount THEN

RAISE\_APPLICATION\_ERROR(-20002, 'Insufficient funds in source 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;

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

VALUES (TransactionSeq.NEXTVAL, p\_from\_account, SYSDATE, p\_amount, 'Transfer Out');

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

VALUES (TransactionSeq.NEXTVAL, p\_to\_account, SYSDATE, p\_amount, 'Transfer In');

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Transfer completed successfully');

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: One or both accounts not found');

WHEN OTHERS THEN

ROLLBACK;

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

END;

/

**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\_count NUMBER;

BEGIN

SELECT COUNT(\*) INTO v\_count FROM Employees WHERE EmployeeID = p\_employee\_id;

IF v\_count = 0 THEN

RAISE\_APPLICATION\_ERROR(-20003, 'Employee ID ' || p\_employee\_id || ' not found');

END IF;

UPDATE Employees

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

WHERE EmployeeID = p\_employee\_id;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Salary updated successfully');

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END;

/

**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

v\_count NUMBER;

BEGIN

SELECT COUNT(\*) INTO v\_count FROM Customers WHERE CustomerID = p\_customer\_id;

IF v\_count > 0 THEN

RAISE\_APPLICATION\_ERROR(-20004, 'Customer ID ' || p\_customer\_id || ' already exists');

END IF;

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 OTHERS THEN

ROLLBACK;

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

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.

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest AS

CURSOR account\_cursor IS

SELECT AccountID, Balance FROM Accounts

WHERE AccountType = 'Savings' FOR UPDATE;

v\_interest NUMBER;

BEGIN

FOR acc\_rec IN account\_cursor LOOP

v\_interest := acc\_rec.Balance \* 0.01;

UPDATE Accounts

SET Balance = Balance + v\_interest,

LastModified = SYSDATE

WHERE AccountID = acc\_rec.AccountID;

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

VALUES (TransactionSeq.NEXTVAL, acc\_rec.AccountID, SYSDATE, v\_interest, 'Interest');

END LOOP;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Monthly interest processed');

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END;

/

**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('Bonus applied to ' || SQL%ROWCOUNT || ' employees');

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END;

/

**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 IN NUMBER,

p\_to\_account IN NUMBER,

p\_amount IN NUMBER

) AS

v\_from\_balance NUMBER;

BEGIN

IF p\_amount <= 0 THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Amount must be positive');

END IF;

SELECT Balance INTO v\_from\_balance FROM Accounts

WHERE AccountID = p\_from\_account FOR UPDATE;

IF v\_from\_balance < p\_amount THEN

RAISE\_APPLICATION\_ERROR(-20002, 'Insufficient funds');

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;

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

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Account not found');

WHEN OTHERS THEN

ROLLBACK;

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

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.

CREATE OR REPLACE FUNCTION CalculateAge(p\_dob IN DATE)

RETURN NUMBER IS

BEGIN

RETURN FLOOR(MONTHS\_BETWEEN(SYSDATE, p\_dob)/12);

EXCEPTION

WHEN OTHERS THEN

RETURN NULL;

END;

/

**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\_amount IN NUMBER,

p\_rate IN NUMBER,

p\_years IN NUMBER

) RETURN NUMBER IS

v\_monthly\_rate NUMBER;

v\_payments NUMBER;

BEGIN

v\_monthly\_rate := p\_rate / 1200;

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

RETURN p\_amount \* v\_monthly\_rate \* POWER(1 + v\_monthly\_rate, v\_payments) /

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

EXCEPTION

WHEN OTHERS THEN

RETURN NULL;

END;

/

**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

SELECT Balance INTO v\_balance FROM Accounts

WHERE AccountID = p\_account\_id;

RETURN v\_balance >= p\_amount;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN FALSE;

WHEN OTHERS THEN

RETURN FALSE;

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.

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 OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog(LogID, TransactionID, ActionDate, ActionType, Amount)

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

:NEW.TransactionType, :NEW.Amount);

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\_current\_balance NUMBER;

BEGIN

-- Validate deposit amount

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

RAISE\_APPLICATION\_ERROR(-20001, 'Deposit amount must be positive.');

END IF;

-- Validate withdrawal amount

IF :NEW.TransactionType = 'Withdrawal' THEN

-- Lock and get current balance

SELECT Balance INTO v\_current\_balance

FROM Accounts

WHERE AccountID = :NEW.AccountID

FOR UPDATE;

IF v\_current\_balance < :NEW.Amount THEN

RAISE\_APPLICATION\_ERROR(-20002, 'Insufficient funds for withdrawal.');

END IF;

END IF;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RAISE\_APPLICATION\_ERROR(-20003, 'Account not found.');

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.

CREATE OR REPLACE PROCEDURE GenerateMonthlyStatements AS

CURSOR c\_customers IS

SELECT CustomerID, Name FROM Customers;

CURSOR c\_transactions(p\_cust\_id NUMBER) IS

SELECT t.TransactionDate, a.AccountType, t.TransactionType, t.Amount

FROM Transactions t

JOIN Accounts a ON t.AccountID = a.AccountID

WHERE a.CustomerID = p\_cust\_id

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

ORDER BY t.TransactionDate;

BEGIN

FOR cust IN c\_customers LOOP

DBMS\_OUTPUT.PUT\_LINE('----- Statement for ' || cust.Name || ' -----');

DBMS\_OUTPUT.PUT\_LINE('Date | Account Type | Type | Amount');

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

FOR txn IN c\_transactions(cust.CustomerID) LOOP

DBMS\_OUTPUT.PUT\_LINE(

TO\_CHAR(txn.TransactionDate, 'DD-MON-YY') || ' | ' ||

RPAD(txn.AccountType, 12) || ' | ' ||

RPAD(txn.TransactionType, 8) || ' | ' ||

TO\_CHAR(txn.Amount, '$9990.99')

);

END LOOP;

DBMS\_OUTPUT.PUT\_LINE(CHR(10)); -- Blank line between customers

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.

CREATE OR REPLACE PROCEDURE ApplyAnnualFee AS

CURSOR c\_accounts IS

SELECT AccountID, Balance FROM Accounts FOR UPDATE;

v\_fee CONSTANT NUMBER := 25; -- $25 annual fee

BEGIN

FOR acc IN c\_accounts LOOP

-- Deduct fee (balance cannot go negative)

UPDATE Accounts

SET Balance = GREATEST(0, Balance - v\_fee),

LastModified = SYSDATE

WHERE CURRENT OF c\_accounts;

-- Log fee transaction

INSERT INTO Transactions

VALUES (TransactionSeq.NEXTVAL, acc.AccountID, SYSDATE, v\_fee, 'Fee');

END LOOP;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Applied annual fee to ' || SQL%ROWCOUNT || ' accounts.');

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

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.

CREATE OR REPLACE PROCEDURE UpdateLoanInterestRates AS

CURSOR c\_loans IS

SELECT LoanID, InterestRate, CustomerID FROM Loans FOR UPDATE;

v\_new\_rate NUMBER;

BEGIN

FOR loan IN c\_loans LOOP

-- Policy: Customers with balance > $10k get 0.5% reduction, others get 0.25% increase

SELECT

CASE WHEN Balance > 10000 THEN loan.InterestRate - 0.5

ELSE loan.InterestRate + 0.25

END INTO v\_new\_rate

FROM Customers

WHERE CustomerID = loan.CustomerID;

UPDATE Loans

SET InterestRate = v\_new\_rate

WHERE CURRENT OF c\_loans;

END LOOP;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Updated interest rates for ' || SQL%ROWCOUNT || ' loans.');

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

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

-- Add new customer

PROCEDURE AddCustomer(

p\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_dob IN DATE,

p\_balance IN NUMBER DEFAULT 0

);

-- Update customer details

PROCEDURE UpdateCustomer(

p\_id IN NUMBER,

p\_name IN VARCHAR2 DEFAULT NULL,

p\_dob IN DATE DEFAULT NULL,

p\_balance IN NUMBER DEFAULT NULL

);

-- Get customer balance

FUNCTION GetBalance(p\_id IN NUMBER) RETURN NUMBER;

END CustomerManagement;

/

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

PROCEDURE AddCustomer(

p\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_dob IN DATE,

p\_balance IN NUMBER DEFAULT 0

) IS

BEGIN

INSERT INTO Customers

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

COMMIT;

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

RAISE\_APPLICATION\_ERROR(-20010, 'Customer ID already exists');

WHEN OTHERS THEN

RAISE;

END AddCustomer;

PROCEDURE UpdateCustomer(

p\_id IN NUMBER,

p\_name IN VARCHAR2 DEFAULT NULL,

p\_dob IN DATE DEFAULT NULL,

p\_balance IN NUMBER DEFAULT NULL

) IS

BEGIN

UPDATE Customers

SET

Name = NVL(p\_name, Name),

DOB = NVL(p\_dob, DOB),

Balance = NVL(p\_balance, Balance),

LastModified = SYSDATE

WHERE CustomerID = p\_id;

IF SQL%ROWCOUNT = 0 THEN

RAISE\_APPLICATION\_ERROR(-20011, 'Customer not found');

END IF;

COMMIT;

END UpdateCustomer;

FUNCTION GetBalance(p\_id IN NUMBER) RETURN NUMBER IS

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance

FROM Customers

WHERE CustomerID = p\_id;

RETURN v\_balance;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN NULL;

END GetBalance;

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

-- Hire new employee

PROCEDURE HireEmployee(

p\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_position IN VARCHAR2,

p\_salary IN NUMBER,

p\_dept IN VARCHAR2

);

-- Update employee details

PROCEDURE UpdateEmployee(

p\_id IN NUMBER,

p\_salary IN NUMBER DEFAULT NULL,

p\_dept IN VARCHAR2 DEFAULT NULL

);

-- Calculate annual salary (with optional bonus %)

FUNCTION CalculateAnnualSalary(

p\_id IN NUMBER,

p\_bonus\_pct IN NUMBER DEFAULT 0

) RETURN NUMBER;

END EmployeeManagement;

/

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

PROCEDURE HireEmployee(

p\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_position IN VARCHAR2,

p\_salary IN NUMBER,

p\_dept IN VARCHAR2

) IS

BEGIN

INSERT INTO Employees

VALUES (p\_id, p\_name, p\_position, p\_salary, p\_dept, SYSDATE);

COMMIT;

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

RAISE\_APPLICATION\_ERROR(-20020, 'Employee ID exists');

END HireEmployee;

PROCEDURE UpdateEmployee(

p\_id IN NUMBER,

p\_salary IN NUMBER DEFAULT NULL,

p\_dept IN VARCHAR2 DEFAULT NULL

) IS

BEGIN

UPDATE Employees

SET

Salary = NVL(p\_salary, Salary),

Department = NVL(p\_dept, Department)

WHERE EmployeeID = p\_id;

IF SQL%ROWCOUNT = 0 THEN

RAISE\_APPLICATION\_ERROR(-20021, 'Employee not found');

END IF;

COMMIT;

END UpdateEmployee;

FUNCTION CalculateAnnualSalary(

p\_id IN NUMBER,

p\_bonus\_pct IN NUMBER DEFAULT 0

) RETURN NUMBER IS

v\_salary NUMBER;

BEGIN

SELECT Salary INTO v\_salary

FROM Employees

WHERE EmployeeID = p\_id;

RETURN v\_salary \* 12 \* (1 + p\_bonus\_pct/100);

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

-- Open new account

PROCEDURE OpenAccount(

p\_id IN NUMBER,

p\_cust\_id IN NUMBER,

p\_type IN VARCHAR2,

p\_balance IN NUMBER DEFAULT 0

);

-- Close account (balance must be zero)

PROCEDURE CloseAccount(p\_id IN NUMBER);

-- Get total balance across all accounts

FUNCTION GetTotalBalance(p\_cust\_id IN NUMBER) RETURN NUMBER;

END AccountOperations;

/

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

PROCEDURE OpenAccount(

p\_id IN NUMBER,

p\_cust\_id IN NUMBER,

p\_type IN VARCHAR2,

p\_balance IN NUMBER DEFAULT 0

) IS

v\_cust\_exists NUMBER;

BEGIN

-- Verify customer exists

SELECT COUNT(\*) INTO v\_cust\_exists

FROM Customers

WHERE CustomerID = p\_cust\_id;

IF v\_cust\_exists = 0 THEN

RAISE\_APPLICATION\_ERROR(-20030, 'Customer does not exist');

END IF;

INSERT INTO Accounts

VALUES (p\_id, p\_cust\_id, p\_type, p\_balance, SYSDATE);

COMMIT;

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

RAISE\_APPLICATION\_ERROR(-20031, 'Account ID exists');

END OpenAccount;

PROCEDURE CloseAccount(p\_id IN NUMBER) IS

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance

FROM Accounts

WHERE AccountID = p\_id

FOR UPDATE;

IF v\_balance != 0 THEN

RAISE\_APPLICATION\_ERROR(-20032, 'Account balance must be zero to close');

END IF;

DELETE FROM Accounts WHERE AccountID = p\_id;

COMMIT;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RAISE\_APPLICATION\_ERROR(-20033, 'Account not found');

END CloseAccount;

FUNCTION GetTotalBalance(p\_cust\_id IN NUMBER) RETURN NUMBER IS

v\_total NUMBER;

BEGIN

SELECT NVL(SUM(Balance), 0) INTO v\_total

FROM Accounts

WHERE CustomerID = p\_cust\_id;

RETURN v\_total;

END GetTotalBalance;

END AccountOperations;

/

**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'));*