**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.

BEGIN

FOR rec IN (SELECT c.CustomerID, l.LoanID, l.InterestRate, c.DOB FROM Customers c JOIN Loans l ON c.CustomerID = l.CustomerID)

LOOP

IF MONTHS\_BETWEEN(SYSDATE, rec.DOB) / 12 > 60 THEN

UPDATE Loans SET InterestRate = rec.InterestRate - 1 WHERE LoanID = rec.LoanID;

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.

BEGIN

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

IF rec.Balance > 10000 THEN

UPDATE Customers SET IsVIP = 'TRUE' WHERE CustomerID = rec.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.

BEGIN

FOR rec IN (SELECT LoanID, CustomerID, EndDate FROM Loans

WHERE EndDate BETWEEN SYSDATE AND SYSDATE + 30)

LOOP

DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan ' || rec.LoanID || ' for Customer ' || rec.CustomerID || ' is due on ' || TO\_CHAR(rec.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.

CREATE OR REPLACE PROCEDURE SafeTransferFunds(p\_FromAccID NUMBER, p\_ToAccID NUMBER, p\_Amount NUMBER) AS

v\_FromBal NUMBER;

BEGIN

SELECT Balance INTO v\_FromBal FROM Accounts WHERE AccountID = p\_FromAccID FOR UPDATE;

IF v\_FromBal < p\_Amount THEN

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

END IF;

UPDATE Accounts SET Balance = Balance - p\_Amount WHERE AccountID = p\_FromAccID;

UPDATE Accounts SET Balance = Balance + p\_Amount WHERE AccountID = p\_ToAccID;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

VALUES (Transactions\_SEQ.NEXTVAL, p\_FromAccID, SYSDATE, p\_Amount, 'FailedTransfer');

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\_EmpID NUMBER, p\_Percent NUMBER) AS

v\_Salary NUMBER;

BEGIN

SELECT Salary INTO v\_Salary FROM Employees WHERE EmployeeID = p\_EmpID FOR UPDATE;

UPDATE Employees SET Salary = v\_Salary + (v\_Salary \* p\_Percent / 100) WHERE EmployeeID = p\_EmpID;

COMMIT;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

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

VALUES (Transactions\_SEQ.NEXTVAL, NULL, SYSDATE, NULL, 'SalaryUpdateError');

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\_CustomerID NUMBER, p\_Name VARCHAR2, p\_DOB DATE, p\_Balance NUMBER) AS

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

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

VALUES (Transactions\_SEQ.NEXTVAL, NULL, SYSDATE, NULL, 'CustomerInsertError');

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

BEGIN

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

UPDATE Accounts SET Balance = rec.Balance + (rec.Balance \* 0.01) WHERE AccountID = rec.AccountID;

END LOOP;

COMMIT;

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 VARCHAR2, p\_BonusPercent NUMBER) AS

BEGIN

FOR rec IN (SELECT EmployeeID, Salary FROM Employees WHERE Department = p\_Department FOR UPDATE) LOOP

UPDATE Employees SET Salary = rec.Salary + (rec.Salary \* p\_BonusPercent / 100) WHERE EmployeeID = rec.EmployeeID;

END LOOP;

COMMIT;

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\_FromAccID NUMBER, p\_ToAccID NUMBER, p\_Amount NUMBER) AS

v\_Balance NUMBER;

BEGIN

SELECT Balance INTO v\_Balance FROM Accounts WHERE AccountID = p\_FromAccID FOR UPDATE;

IF v\_Balance < p\_Amount THEN

RAISE\_APPLICATION\_ERROR(-20002, 'Insufficient balance in source account.');

END IF;

UPDATE Accounts SET Balance = Balance - p\_Amount WHERE AccountID = p\_FromAccID;

UPDATE Accounts SET Balance = Balance + p\_Amount WHERE AccountID = p\_ToAccID;

COMMIT;

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 DATE) RETURN NUMBER IS

BEGIN

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

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\_LoanAmount NUMBER, p\_InterestRate NUMBER, p\_Years NUMBER) RETURN NUMBER IS

r NUMBER;

n NUMBER;

BEGIN

r := p\_InterestRate / 1200;

n := p\_Years \* 12;

RETURN ROUND(p\_LoanAmount \* r \* POWER(1 + r, n) / (POWER(1 + r, n) - 1), 2);

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\_AccountID NUMBER, p\_Amount NUMBER) RETURN BOOLEAN IS

v\_Balance NUMBER;

BEGIN

SELECT Balance INTO v\_Balance FROM Accounts WHERE AccountID = p\_AccountID;

RETURN v\_Balance >= p\_Amount;

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 TABLE AuditLog (

LogID NUMBER PRIMARY KEY,

TransactionID NUMBER,

ActionDate DATE,

ActionType VARCHAR2(20)

);

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

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

VALUES (AuditLog\_SEQ.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

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

IF :NEW.TransactionType = 'Withdrawal' AND :NEW.Amount > v\_Balance THEN

RAISE\_APPLICATION\_ERROR(-20003, 'Withdrawal amount exceeds available balance.');

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

RAISE\_APPLICATION\_ERROR(-20004, 'Deposit amount must be positive.');

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.

DECLARE

CURSOR trans\_cur IS

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

FROM Transactions t

JOIN Accounts a ON t.AccountID = a.AccountID

WHERE TO\_CHAR(t.TransactionDate, 'MMYYYY') = TO\_CHAR(SYSDATE, 'MMYYYY');

v\_trans trans\_cur%ROWTYPE;

BEGIN

OPEN trans\_cur;

LOOP

FETCH trans\_cur INTO v\_trans;

EXIT WHEN trans\_cur%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Customer ' || v\_trans.CustomerID ||

' - Account ' || v\_trans.AccountID ||

' - ' || v\_trans.TransactionType ||

' - Amount: ' || v\_trans.Amount ||

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

END LOOP;

CLOSE trans\_cur;

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 acc\_cur IS

SELECT AccountID, Balance FROM Accounts FOR UPDATE;

v\_acc acc\_cur%ROWTYPE;

v\_fee NUMBER := 100;

BEGIN

OPEN acc\_cur;

LOOP

FETCH acc\_cur INTO v\_acc;

EXIT WHEN acc\_cur%NOTFOUND;

UPDATE Accounts SET Balance = v\_acc.Balance - v\_fee WHERE AccountID = v\_acc.AccountID;

END LOOP;

CLOSE acc\_cur;

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 loan\_cur IS

SELECT LoanID, InterestRate FROM Loans FOR UPDATE;

v\_loan loan\_cur%ROWTYPE;

BEGIN

OPEN loan\_cur;

LOOP

FETCH loan\_cur INTO v\_loan;

EXIT WHEN loan\_cur%NOTFOUND;

IF v\_loan.InterestRate > 6 THEN

UPDATE Loans SET InterestRate = v\_loan.InterestRate - 0.5 WHERE LoanID = v\_loan.LoanID;

END IF;

END LOOP;

CLOSE loan\_cur;

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 AddCustomer(p\_ID NUMBER, p\_Name VARCHAR2, p\_DOB DATE, p\_Balance NUMBER);

PROCEDURE UpdateCustomer(p\_ID NUMBER, p\_Name VARCHAR2, p\_Balance NUMBER);

FUNCTION GetBalance(p\_ID NUMBER) RETURN NUMBER;

END;

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

PROCEDURE AddCustomer(p\_ID NUMBER, p\_Name VARCHAR2, p\_DOB DATE, p\_Balance NUMBER) IS

BEGIN

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

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

END;

PROCEDURE UpdateCustomer(p\_ID NUMBER, p\_Name VARCHAR2, p\_Balance NUMBER) IS

BEGIN

UPDATE Customers SET Name = p\_Name, Balance = p\_Balance, LastModified = SYSDATE

WHERE CustomerID = p\_ID;

END;

FUNCTION GetBalance(p\_ID NUMBER) RETURN NUMBER IS

v\_Balance NUMBER;

BEGIN

SELECT Balance INTO v\_Balance FROM Customers WHERE CustomerID = p\_ID;

RETURN v\_Balance;

END;

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.

CREATE OR REPLACE PACKAGE EmployeeManagement AS

PROCEDURE HireEmployee(p\_ID NUMBER, p\_Name VARCHAR2, p\_Position VARCHAR2, p\_Salary NUMBER, p\_Department VARCHAR2, p\_HireDate DATE);

PROCEDURE UpdateEmployee(p\_ID NUMBER, p\_Salary NUMBER);

FUNCTION GetAnnualSalary(p\_ID NUMBER) RETURN NUMBER;

END;

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

PROCEDURE HireEmployee(p\_ID 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\_ID, p\_Name, p\_Position, p\_Salary, p\_Department, p\_HireDate);

END;

PROCEDURE UpdateEmployee(p\_ID NUMBER, p\_Salary NUMBER) IS

BEGIN

UPDATE Employees SET Salary = p\_Salary WHERE EmployeeID = p\_ID;

END;

FUNCTION GetAnnualSalary(p\_ID NUMBER) RETURN NUMBER IS

v\_Salary NUMBER;

BEGIN

SELECT Salary INTO v\_Salary FROM Employees WHERE EmployeeID = p\_ID;

RETURN v\_Salary \* 12;

END;

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.

CREATE OR REPLACE PACKAGE AccountOperations AS

PROCEDURE OpenAccount(p\_AccID NUMBER, p\_CustID NUMBER, p\_Type VARCHAR2, p\_Balance NUMBER);

PROCEDURE CloseAccount(p\_AccID NUMBER);

FUNCTION GetTotalBalance(p\_CustID NUMBER) RETURN NUMBER;

END;

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

PROCEDURE OpenAccount(p\_AccID NUMBER, p\_CustID NUMBER, p\_Type VARCHAR2, p\_Balance NUMBER) IS

BEGIN

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

VALUES (p\_AccID, p\_CustID, p\_Type, p\_Balance, SYSDATE);

END;

PROCEDURE CloseAccount(p\_AccID NUMBER) IS

BEGIN

DELETE FROM Accounts WHERE AccountID = p\_AccID;

END;

FUNCTION GetTotalBalance(p\_CustID NUMBER) RETURN NUMBER IS

v\_Total NUMBER;

BEGIN

SELECT SUM(Balance) INTO v\_Total FROM Accounts WHERE CustomerID = p\_CustID;

RETURN NVL(v\_Total, 0);

END;

END;