# Exercise 1: Control Structures

**Scenario 1:** The bank wants to apply a discount to loan interest rates for customers above 60 years old.

o **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, l.LoanID, l.InterestRate FROM Customers c

JOIN Loans l ON c.CustomerID = l.CustomerID

WHERE TRUNC(MONTHS\_BETWEEN(SYSDATE, c.DOB) / 12) > 60;

v\_CustomerID Customers.CustomerID%TYPE; v\_LoanID Loans.LoanID%TYPE;

v\_InterestRate Loans.InterestRate%TYPE; BEGIN

OPEN customer\_cursor; LOOP

FETCH customer\_cursor INTO v\_CustomerID, v\_LoanID, v\_InterestRate; EXIT WHEN customer\_cursor%NOTFOUND;

UPDATE Loans

SET InterestRate = v\_InterestRate \* 0.99 WHERE LoanID = v\_LoanID;

END LOOP;

CLOSE customer\_cursor; COMMIT;

END;

**Scenario 2:** A customer can be promoted to VIP status based on their balance.

o **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));

UPDATE Customers SET IsVIP = 'NO'; COMMIT;

DECLARE

CURSOR customer\_cursor IS SELECT CustomerID, Balance FROM Customers;

v\_CustomerID Customers.CustomerID%TYPE; v\_Balance Customers.Balance%TYPE;

BEGIN

OPEN customer\_cursor;

LOOP

FETCH customer\_cursor INTO v\_CustomerID, v\_Balance; EXIT WHEN customer\_cursor%NOTFOUND;

IF v\_Balance > 10000 THEN UPDATE Customers

SET IsVIP = 'YES'

WHERE CustomerID = v\_CustomerID; ELSE

UPDATE Customers SET IsVIP = 'NO'

WHERE CustomerID = v\_CustomerID; END IF;

END LOOP;

CLOSE customer\_cursor; COMMIT;

END;

**Scenario 3:** The bank wants to send reminders to customers whose loans are due within the next 30 days.

o **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 c.CustomerID, c.Name, c.Email, l.LoanID, l.EndDate FROM Customers c

JOIN Loans l ON c.CustomerID = l.CustomerID

WHERE l.EndDate BETWEEN SYSDATE AND SYSDATE + 30;

v\_CustomerID Customers.CustomerID%TYPE; v\_Name Customers.Name%TYPE;

v\_Email Customers.Email%TYPE; v\_LoanID Loans.LoanID%TYPE; v\_EndDate Loans.EndDate%TYPE;

BEGIN

OPEN loan\_cursor;

LOOP

FETCH loan\_cursor INTO v\_CustomerID, v\_Name, v\_Email, v\_LoanID, v\_EndDate; EXIT WHEN loan\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Reminder: Customer ' || v\_Name || ' (ID: ' || v\_CustomerID || ')');

DBMS\_OUTPUT.PUT\_LINE('Your loan with Loan ID: ' || v\_LoanID || ' is due on ' || TO\_CHAR(v\_EndDate, 'YYYY-MM-DD') || '.');

DBMS\_OUTPUT.PUT\_LINE('Please ensure payment is made by the due date.'); DBMS\_OUTPUT.PUT\_LINE(''); -- For a blank line between messages

END LOOP;

CLOSE loan\_cursor;

END;

# Exercise 2: Error Handling

**Scenario 1:** Handle exceptions during fund transfers between accounts.

o **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\_SourceAccountID IN

Accounts.AccountID%TYPE,p\_DestinationAccountID IN Accounts.AccountID%TYPE, p\_Amount IN NUMBER)

IS

v\_SourceBalance Accounts.Balance%TYPE; v\_DestinationBalance Accounts.Balance%TYPE; insufficient\_funds EXCEPTION;

account\_not\_found EXCEPTION; BEGIN

SELECT Balance INTO v\_SourceBalance FROM Accounts

WHERE AccountID = p\_SourceAccountID FOR UPDATE;

SELECT Balance INTO v\_DestinationBalance FROM Accounts

WHERE AccountID = p\_DestinationAccountID FOR UPDATE;

IF v\_SourceBalance < p\_Amount THEN RAISE insufficient\_funds;

END IF;

UPDATE Accounts

SET Balance = Balance - p\_Amount

WHERE AccountID = p\_SourceAccountID; UPDATE Accounts

SET Balance = Balance + p\_Amount

WHERE AccountID = p\_DestinationAccountID; COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Transfer of ' || p\_Amount || ' from Account ' || p\_SourceAccountID || ' to Account ' || p\_DestinationAccountID || ' completed successfully.');

EXCEPTION

WHEN insufficient\_funds THEN ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: Insufficient funds in Account ' || p\_SourceAccountID || '. Transfer aborted.');

WHEN NO\_DATA\_FOUND THEN ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: One or both accounts not found. Transfer aborted.'); WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: ' || SQLERRM || '. Transfer aborted.'); END SafeTransferFunds;

**Scenario 2:** Manage errors when updating employee salaries.

o **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 IN Employees.EmployeeID%TYPE,p\_Percentage IN NUMBER)

IS

v\_Salary Employees.Salary%TYPE; employee\_not\_found EXCEPTION; BEGIN

SELECT Salary INTO v\_Salary FROM Employees

WHERE EmployeeID = p\_EmployeeID;

v\_Salary := v\_Salary \* (1 + p\_Percentage / 100); UPDATE Employees

SET Salary = v\_Salary

WHERE EmployeeID = p\_EmployeeID; COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Salary of Employee ID ' || p\_EmployeeID || ' increased by '

|| p\_Percentage || '%. New Salary: ' || v\_Salary);

EXCEPTION

WHEN NO\_DATA\_FOUND THEN ROLLBACK;

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

Salary update aborted.'); WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: ' || SQLERRM || '. Salary update aborted.'); END UpdateSalary;

**Scenario 3:** Ensure data integrity when adding a new customer.

o **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 IN Customers.CustomerID%TYPE, p\_Name IN Customers.Name%TYPE,

p\_DOB IN Customers.DOB%TYPE, p\_Balance IN Customers.Balance%TYPE

) IS

BEGIN

INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified) VALUES (p\_CustomerID, p\_Name, p\_DOB, p\_Balance, SYSDATE);

COMMIT;

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

WHEN DUP\_VAL\_ON\_INDEX THEN ROLLBACK;

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

Insertion aborted.'); WHEN OTHERS THEN ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: ' || SQLERRM || '. Insertion aborted.'); END AddNewCustomer;

# Exercise 3: Stored Procedures

**Scenario 1:** The bank needs to process monthly interest for all savings accounts.

o **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

CURSOR savings\_accounts\_cursor IS SELECT AccountID, Balance FROM Accounts

WHERE AccountType = 'Savings' FOR UPDATE OF Balance;

v\_AccountID Accounts.AccountID%TYPE;

v\_Balance Accounts.Balance%TYPE; v\_Interest NUMBER;

BEGIN

OPEN savings\_accounts\_cursor; LOOP

FETCH savings\_accounts\_cursor INTO v\_AccountID, v\_Balance; EXIT WHEN savings\_accounts\_cursor%NOTFOUND;

v\_Interest := v\_Balance \* 0.01; UPDATE Accounts

SET Balance = Balance + v\_Interest, LastModified = SYSDATE

WHERE CURRENT OF savings\_accounts\_cursor; END LOOP;

CLOSE savings\_accounts\_cursor; COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Monthly interest has been processed for all savings accounts.');

EXCEPTION

WHEN OTHERS THEN ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: ' || SQLERRM || '. Monthly interest processing aborted.');

END ProcessMonthlyInterest;

**Scenario 2:** The bank wants to implement a bonus scheme for employees based on their performance.

o **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 Employees.Department%TYPE,p\_BonusPercentage IN NUMBER)

IS

CURSOR dept\_employees\_cursor IS SELECT EmployeeID, Salary FROM Employees

WHERE Department = p\_Department FOR UPDATE OF Salary;

v\_EmployeeID Employees.EmployeeID%TYPE; v\_Salary Employees.Salary%TYPE;

v\_Bonus NUMBER; BEGIN

OPEN dept\_employees\_cursor;

LOOP

FETCH dept\_employees\_cursor INTO v\_EmployeeID, v\_Salary; EXIT WHEN dept\_employees\_cursor%NOTFOUND;

v\_Bonus := v\_Salary \* (p\_BonusPercentage / 100); UPDATE Employees

SET Salary = Salary + v\_Bonus

WHERE CURRENT OF dept\_employees\_cursor;

END LOOP;

CLOSE dept\_employees\_cursor; COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Bonus of ' || p\_BonusPercentage || '% has been applied to all employees in the ' || p\_Department || ' department.');

EXCEPTION

WHEN OTHERS THEN ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: ' || SQLERRM || '. Bonus update aborted.'); END UpdateEmployeeBonus;

**Scenario 3:** Customers should be able to transfer funds between their accounts.

o **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\_SourceAccountID

IN

Accounts.AccountID%TYPE, p\_DestinationAccountID IN Accounts.AccountID%TYPE, p\_Amount IN NUMBER

)

IS

v\_SourceBalance Accounts.Balance%TYPE; v\_DestinationBalance Accounts.Balance%TYPE; insufficient\_funds EXCEPTION;

account\_not\_found EXCEPTION;

BEGIN

SELECT Balance INTO v\_SourceBalance FROM Accounts

WHERE AccountID = p\_SourceAccountID FOR UPDATE;

SELECT Balance INTO v\_DestinationBalance FROM Accounts

WHERE AccountID = p\_DestinationAccountID FOR UPDATE;

IF v\_SourceBalance < p\_Amount THEN RAISE insufficient\_funds;

END IF;

UPDATE Accounts

SET Balance = Balance - p\_Amount

WHERE AccountID = p\_SourceAccountID; UPDATE Accounts

SET Balance = Balance + p\_Amount

WHERE AccountID = p\_DestinationAccountID; COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Transfer of ' || p\_Amount || ' from Account ' || p\_SourceAccountID

|| ' to Account ' || p\_DestinationAccountID || ' completed successfully.'); EXCEPTION

WHEN insufficient\_funds THEN ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: Insufficient funds in Account ' || p\_SourceAccountID || '.

Transfer aborted.');

WHEN NO\_DATA\_FOUND THEN ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: One or both accounts not found. Transfer aborted.'); WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: ' || SQLERRM || '. Transfer aborted.'); END TransferFunds;

# Exercise 4: Functions

**Scenario 1:** Calculate the age of customers for eligibility checks.

o **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

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.

o **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 IN NUMBER,

p\_AnnualInterestRate IN NUMBER, p\_LoanDurationYears IN NUMBER

) RETURN NUMBER IS

v\_MonthlyInterestRate NUMBER; v\_NumberOfPayments NUMBER; v\_MonthlyInstallment NUMBER;

BEGIN

v\_MonthlyInterestRate := p\_AnnualInterestRate / 12 / 100; v\_NumberOfPayments := p\_LoanDurationYears \* 12;

IF v\_MonthlyInterestRate > 0 THEN

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

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

ELSE

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

RETURN v\_MonthlyInstallment; END CalculateMonthlyInstallment;

/

**Scenario 3:** Check if a customer has sufficient balance before making a transaction.

o **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 IN Accounts.AccountID%TYPE, p\_Amount IN NUMBER

) RETURN BOOLEAN IS

v\_Balance Accounts.Balance%TYPE; 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;

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.

o **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.

o **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, AccountID NUMBER, TransactionDate DATE, Amount NUMBER,

TransactionType VARCHAR2(10), LogTimestamp DATE

);

CREATE OR REPLACE TRIGGER LogTransaction AFTER INSERT ON Transactions

FOR EACH ROW BEGIN

INSERT INTO AuditLog (

LogID, TransactionID, AccountID,

TransactionDate, Amount, TransactionType, LogTimestamp

) VALUES (

AuditLog\_seq.NEXTVAL,

:NEW.TransactionID,

:NEW.AccountID,

:NEW.TransactionDate,

:NEW.Amount,

:NEW.TransactionType, SYSDATE

); END;

**Scenario 3:** Enforce business rules on deposits and withdrawals.

o **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 Accounts.Balance%TYPE; insufficient\_funds EXCEPTION;

negative\_deposit EXCEPTION; BEGIN

SELECT Balance INTO v\_Balance FROM Accounts

WHERE AccountID = :NEW.AccountID

FOR UPDATE;

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

RAISE insufficient\_funds; END IF;

ELSIF :NEW.TransactionType = 'Deposit' THEN

-- Check if the deposit amount is positive IF :NEW.Amount <= 0 THEN

RAISE negative\_deposit; END IF;

END IF;

EXCEPTION

WHEN insufficient\_funds THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient funds for withdrawal.'); WHEN negative\_deposit THEN

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

RAISE\_APPLICATION\_ERROR(-20003, 'An unexpected error occurred: ' || SQLERRM); END CheckTransactionRules;

/

# Exercise 6: Cursors

**Scenario 1:** Generate monthly statements for all customers.

o **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

-- Declare a record type for transactions CURSOR cur\_Transactions 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 t.TransactionDate BETWEEN TRUNC(SYSDATE, 'MM') AND LAST\_DAY(SYSDATE);

-- Record variable to hold each row fetched by the cursor r\_Transaction cur\_Transactions%ROWTYPE;

BEGIN

OPEN cur\_Transactions; LOOP

FETCH cur\_Transactions INTO r\_Transaction; EXIT WHEN cur\_Transactions%NOTFOUND;

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

TO\_CHAR(r\_Transaction.TransactionDate, 'YYYY-MM-DD'));

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

END LOOP;

CLOSE cur\_Transactions; END;

/

**Scenario 2:** Apply annual fee to all accounts.

o **Question:** Write a PL/SQL block using an explicit cursor **ApplyAnnualFee** that deducts an annual maintenance fee from the balance of all accounts.

DECLARE

annual\_fee CONSTANT NUMBER := 50; CURSOR cur\_Accounts IS

SELECT AccountID, Balance FROM Accounts

FOR UPDATE OF Balance;

r\_Account cur\_Accounts%ROWTYPE; BEGIN

OPEN cur\_Accounts;

LOOP

FETCH cur\_Accounts INTO r\_Account; EXIT WHEN cur\_Accounts%NOTFOUND;

IF r\_Account.Balance >= annual\_fee THEN

-- Deduct the annual fee from the account balance UPDATE Accounts

SET Balance = Balance - annual\_fee WHERE CURRENT OF cur\_Accounts;

-- Print the updated balance for verification

DBMS\_OUTPUT.PUT\_LINE('Account ID: ' || r\_Account.AccountID || ' - New Balance: ' || (r\_Account.Balance - annual\_fee));

ELSE

-- Print a message if there are insufficient funds

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

END IF;

END LOOP;

-- Close the cursor CLOSE cur\_Accounts;

END;

/

**Scenario 3:** Update the interest rate for all loans based on a new policy.

o **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

interest\_rate\_increment CONSTANT NUMBER := 1;

CURSOR cur\_Loans IS

SELECT LoanID, InterestRate FROM Loans

FOR UPDATE OF InterestRate; r\_Loan cur\_Loans%ROWTYPE;

BEGIN

OPEN cur\_Loans;

LOOP

FETCH cur\_Loans INTO r\_Loan;

EXIT WHEN cur\_Loans%NOTFOUND;

UPDATE Loans

SET InterestRate = r\_Loan.InterestRate + interest\_rate\_increment WHERE CURRENT OF cur\_Loans;

DBMS\_OUTPUT.PUT\_LINE('Loan ID: ' || r\_Loan.LoanID || ' - New Interest Rate: ' || (r\_Loan.InterestRate + interest\_rate\_increment));

END LOOP;

CLOSE cur\_Loans;

END;

# Exercise 7: Packages

**Scenario 1:** Group all customer-related procedures and functions into a package.

o **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 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

e\_DuplicateCustomerID EXCEPTION; 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

RAISE e\_DuplicateCustomerID; WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('An error occurred: ' || SQLERRM); END AddNewCustomer;

PROCEDURE UpdateCustomerDetails( p\_CustomerID IN NUMBER,

p\_Name IN VARCHAR2, p\_DOB IN DATE,

p\_Balance IN NUMBER

) IS

e\_CustomerNotFound EXCEPTION; BEGIN

UPDATE Customers SET Name = p\_Name,

DOB = p\_DOB,

Balance = p\_Balance, LastModified = SYSDATE

WHERE CustomerID = p\_CustomerID; IF SQL%ROWCOUNT = 0 THEN

RAISE e\_CustomerNotFound; END IF;

EXCEPTION

WHEN e\_CustomerNotFound THEN

DBMS\_OUTPUT.PUT\_LINE('Customer ID ' || p\_CustomerID || ' not found.'); WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('An error occurred: ' || SQLERRM); END UpdateCustomerDetails;

FUNCTION GetCustomerBalance( p\_CustomerID IN 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

DBMS\_OUTPUT.PUT\_LINE('Customer ID ' || p\_CustomerID || ' not found.'); RETURN NULL;

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('An error occurred: ' || SQLERRM); RETURN NULL;

END GetCustomerBalance; END CustomerManagement;

**Scenario 2:** Create a package to manage employee data.

o **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 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, p\_HireDate IN DATE

);

FUNCTION CalculateAnnualSalary( p\_EmployeeID IN NUMBER

) RETURN NUMBER;

END EmployeeManagement;

**Scenario 3:** Group all account-related operations into a package.

o **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 BODY AccountOperations AS

-- Procedure to open a new account PROCEDURE OpenAccount(

p\_AccountID IN NUMBER, p\_CustomerID IN NUMBER, p\_AccountType IN VARCHAR2, p\_Balance IN NUMBER

) IS

-- Exception for duplicate account ID e\_DuplicateAccountID EXCEPTION;

BEGIN

-- Attempt to insert the new account

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

RAISE e\_DuplicateAccountID; WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('An error occurred: ' || SQLERRM); END OpenAccount;

PROCEDURE CloseAccount( p\_AccountID IN NUMBER

) IS

e\_AccountNotFound EXCEPTION; BEGIN

DELETE FROM Accounts

WHERE AccountID = p\_AccountID; IF SQL%ROWCOUNT = 0 THEN

RAISE e\_AccountNotFound; END IF;

EXCEPTION

WHEN e\_AccountNotFound THEN

DBMS\_OUTPUT.PUT\_LINE('Account ID ' || p\_AccountID || ' not found.'); WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('An error occurred: ' || SQLERRM); END CloseAccount;

FUNCTION GetTotalBalance( p\_CustomerID IN NUMBER

) RETURN NUMBER IS

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

DBMS\_OUTPUT.PUT\_LINE('Customer ID ' || p\_CustomerID || ' not found.'); RETURN NULL;

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('An error occurred: ' || SQLERRM); RETURN NULL;

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'));*