**JAVA FSE - Week 2 - Assignment**

**Module 4 - PL/SQL Programming**

**Schema Used:**

CREATE TABLE Customers (

CustomerID INT PRIMARY KEY,

Name VARCHAR(100),

DOB DATE,

Balance DECIMAL(10, 2),

LastModified DATE

);

CREATE TABLE Accounts (

AccountID INT PRIMARY KEY,

CustomerID INT,

AccountType VARCHAR(20),

Balance DECIMAL(10, 2),

LastModified DATE,

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

CREATE TABLE Transactions (

TransactionID INT PRIMARY KEY,

AccountID INT,

TransactionDate DATE,

Amount DECIMAL(10, 2),

TransactionType VARCHAR(10),

FOREIGN KEY (AccountID) REFERENCES Accounts(AccountID)

);

CREATE TABLE Loans (

LoanID INT PRIMARY KEY,

CustomerID INT,

LoanAmount DECIMAL(10, 2),

InterestRate DECIMAL(5, 2),

StartDate DATE,

EndDate DATE,

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

CREATE TABLE Employees (

EmployeeID INT PRIMARY KEY,

Name VARCHAR(100),

Position VARCHAR(50),

Salary DECIMAL(10, 2),

Department VARCHAR(50),

HireDate DATE

);

**Data Insertions:**

**Customers Table:**

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

(1, 'Vinoth', TO\_DATE('1995-02-03', 'YYYY-MM-DD'), 1000.00, TO\_DATE('2024-08-01', 'YYYY-MM-DD'));

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

(2, 'Kannan', TO\_DATE('1991-03-04', 'YYYY-MM-DD'), 2000.00, TO\_DATE('2024-08-02', 'YYYY-MM-DD'));

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

(3, 'Vijay', TO\_DATE('1992-03-03', 'YYYY-MM-DD'), 3000.00, TO\_DATE('2024-08-03', 'YYYY-MM-DD'));

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

(4, 'Ananth', TO\_DATE('1995-04-04', 'YYYY-MM-DD'), 4000.00, TO\_DATE('2024-08-04', 'YYYY-MM-DD'));

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

(5, 'Surya', TO\_DATE('1993-05-05', 'YYYY-MM-DD'), 5000.00, TO\_DATE('2024-08-05', 'YYYY-MM-DD'));

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

(6, 'Karthi', TO\_DATE('1991-06-06', 'YYYY-MM-DD'), 6000.00, TO\_DATE('2024-08-06', 'YYYY-MM-DD'));

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

(7, 'Hari', TO\_DATE('1989-07-07', 'YYYY-MM-DD'), 7000.00, TO\_DATE('2024-08-07', 'YYYY-MM-DD'));

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

(8, 'Vishnu', TO\_DATE('1988-08-08', 'YYYY-MM-DD'), 8000.00, TO\_DATE('2024-08-08', 'YYYY-MM-DD'));

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

(9, 'Mohammed', TO\_DATE('1994-09-09', 'YYYY-MM-DD'), 9000.00, TO\_DATE('2024-08-09', 'YYYY-MM-DD'));

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

(10, 'Nawaz', TO\_DATE('1987-10-10', 'YYYY-MM-DD'), 10000.00, TO\_DATE('2024-08-10', 'YYYY-MM-DD'));

**Accounts Table:**

INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified) VALUES (1, 1, 'Savings', 1000.00, TO\_DATE('2024-08-01', 'YYYY-MM-DD'));

INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified) VALUES (2, 2, 'Current', 2000.00, TO\_DATE('2024-08-02', 'YYYY-MM-DD'));

INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified) VALUES (3, 3, 'Savings', 3000.00, TO\_DATE('2024-08-03', 'YYYY-MM-DD'));

INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified) VALUES (4, 4, 'Current', 4000.00, TO\_DATE('2024-08-04', 'YYYY-MM-DD'));

INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified) VALUES (5, 5, 'Savings', 5000.00, TO\_DATE('2024-08-05', 'YYYY-MM-DD'));

INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified) VALUES (6, 6, 'Current', 6000.00, TO\_DATE('2024-08-06', 'YYYY-MM-DD'));

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

(7, 7, 'Savings', 7000.00, TO\_DATE('2024-08-07', 'YYYY-MM-DD'));

INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified) VALUES (8, 8, 'Current', 8000.00, TO\_DATE('2024-08-08', 'YYYY-MM-DD'));

INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified) VALUES (9, 9, 'Savings', 9000.00, TO\_DATE('2024-08-09', 'YYYY-MM-DD'));

INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified) VALUES (10, 10, 'Current', 10000.00, TO\_DATE('2024-08-10', 'YYYY-MM-DD'));

**Transactions Table:**

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType) VALUES (1, 1, TO\_DATE('2024-08-01', 'YYYY-MM-DD'), 100.00, 'Credit');

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType) VALUES (2, 2, TO\_DATE('2024-08-02', 'YYYY-MM-DD'), 200.00, 'Debit');

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType) VALUES (3, 3, TO\_DATE('2024-08-03', 'YYYY-MM-DD'), 300.00, 'Credit');

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType) VALUES (4, 4, TO\_DATE('2024-08-04', 'YYYY-MM-DD'), 400.00, 'Debit');

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType) VALUES (5, 5, TO\_DATE('2024-08-05', 'YYYY-MM-DD'), 500.00, 'Credit');

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType) VALUES (6, 6, TO\_DATE('2024-08-06', 'YYYY-MM-DD'), 600.00, 'Debit');

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType) VALUES (7, 7, TO\_DATE('2024-08-07', 'YYYY-MM-DD'), 700.00, 'Credit');

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType) VALUES (8, 8, TO\_DATE('2024-08-08', 'YYYY-MM-DD'), 800.00, 'Debit');

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType) VALUES (9, 9, TO\_DATE('2024-08-09', 'YYYY-MM-DD'), 900.00, 'Credit');

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType) VALUES (10, 10, TO\_DATE('2024-08-10', 'YYYY-MM-DD'), 1000.00, 'Debit');

**Loans Table:**

INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate) VALUES (1, 1, 5000.00, 5.5, TO\_DATE('2024-01-01', 'YYYY-MM-DD'), TO\_DATE('2025-01-01', 'YYYY-MM-DD'));

INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate) VALUES (2, 2, 10000.00, 4.5, TO\_DATE('2024-02-01', 'YYYY-MM-DD'), TO\_DATE('2026-02-01', 'YYYY-MM-DD'));

INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate) VALUES (3, 3, 15000.00, 3.5, TO\_DATE('2024-03-01', 'YYYY-MM-DD'), TO\_DATE('2027-03-01', 'YYYY-MM-DD'));

INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate) VALUES (4, 4, 20000.00, 6.0, TO\_DATE('2024-04-01', 'YYYY-MM-DD'), TO\_DATE('2025-04-01', 'YYYY-MM-DD'));

INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate) VALUES (5, 5, 25000.00, 5.0, TO\_DATE('2024-05-01', 'YYYY-MM-DD'), TO\_DATE('2026-05-01', 'YYYY-MM-DD'));

INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate) VALUES (6, 6, 30000.00, 4.0, TO\_DATE('2024-06-01', 'YYYY-MM-DD'), TO\_DATE('2027-06-01', 'YYYY-MM-DD'));

INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate) VALUES (7, 7, 35000.00, 3.0, TO\_DATE('2024-07-01', 'YYYY-MM-DD'), TO\_DATE('2028-07-01', 'YYYY-MM-DD'));

INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate) VALUES (8, 8, 40000.00, 6.5, TO\_DATE('2024-08-01', 'YYYY-MM-DD'), TO\_DATE('2025-08-01', 'YYYY-MM-DD'));

INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate) VALUES (9, 9, 45000.00, 5.5, TO\_DATE('2024-09-01', 'YYYY-MM-DD'), TO\_DATE('2027-09-01', 'YYYY-MM-DD'));

INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate) VALUES (10, 10, 50000.00, 4.5, TO\_DATE('2024-10-01', 'YYYY-MM-DD'), TO\_DATE('2026-10-01', 'YYYY-MM-DD'));

**Employees Table:**

INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate) VALUES (1, 'Vinoth', 'Manager', 70000.00, 'Finance', TO\_DATE('2020-01-01', 'YYYY-MM-DD'));

INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate) VALUES (2, 'Kannan', 'Clerk', 40000.00, 'HR', TO\_DATE('2019-02-01', 'YYYY-MM-DD'));

INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate) VALUES (3, 'Vijay', 'Analyst', 60000.00, 'IT', TO\_DATE('2018-03-01', 'YYYY-MM-DD'));

INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate) VALUES (4, 'Ananth', 'Assistant', 35000.00, 'Admin', TO\_DATE('2021-04-01', 'YYYY-MM-DD'));

INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate) VALUES (5, 'Surya', 'Executive', 50000.00, 'Marketing', TO\_DATE('2022-05-01', 'YYYY-MM-DD'));

INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate) VALUES (6, 'Karthi', 'Manager', 75000.00, 'Sales', TO\_DATE('2017-06-01', 'YYYY-MM-DD'));

INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate) VALUES (7, 'Hari', 'Developer', 65000.00, 'IT', TO\_DATE('2020-07-01', 'YYYY-MM-DD'));

INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate) VALUES (8, 'Vishnu', 'Technician', 45000.00, 'Maintenance', TO\_DATE('2019-08-01', 'YYYY-MM-DD'));

INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate) VALUES (9, 'Mohammed', 'Engineer', 70000.00, 'Operations', TO\_DATE('2021-09-01', 'YYYY-MM-DD'));

INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate) VALUES (10, 'Nawaz', 'Consultant', 80000.00, 'Finance', TO\_DATE('2022-10-01', 'YYYY-MM-DD'));

**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.

**Query:**

DECLARE

v\_customer\_id Customers.CustomerID%TYPE;

v\_dob Customers.DOB%TYPE;

v\_age NUMBER;

v\_new\_rate Loans.InterestRate%TYPE;

BEGIN

FOR rec IN (SELECT CustomerID, DOB FROM Customers)

LOOP

v\_customer\_id := rec.CustomerID;

v\_dob := rec.DOB;

**-- Calculate age**

v\_age := TRUNC((SYSDATE - v\_dob) / 365.25);

**-- Check if age is above 60**

IF v\_age > 60 THEN

**-- Update the loan interest rate with a 1% discount**

UPDATE Loans

SET InterestRate = InterestRate - 1

WHERE CustomerID = v\_customer\_id;

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.

**Query:**

ALTER TABLE Customers ADD IsVIP VARCHAR2(5);

DECLARE

CURSOR cust\_cur IS

SELECT CustomerID, Balance

FROM Customers

WHERE Balance > 10000;

BEGIN

FOR rec IN cust\_cur LOOP

**-- Update the IsVIP flag to TRUE for customers with balance > $10,000**

UPDATE Customers

SET IsVIP = 'TRUE'

WHERE CustomerID = rec.CustomerID;

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.

**Query:**

DECLARE

v\_customer\_name Customers.Name%TYPE;

v\_due\_date Loans.EndDate%TYPE;

BEGIN

FOR rec IN (SELECT l.CustomerID, l.EndDate, c.Name

FROM Loans l

JOIN Customers c ON l.CustomerID = c.CustomerID

WHERE l.EndDate BETWEEN SYSDATE AND SYSDATE + 30)

LOOP

v\_customer\_name := rec.Name;

v\_due\_date := rec.EndDate;

**-- Print reminder message**

DBMS\_OUTPUT.PUT\_LINE('Reminder: Dear ' || v\_customer\_name ||

', your loan is due on ' || TO\_CHAR(v\_due\_date, '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.

**Query:**

CREATE OR REPLACE PROCEDURE SafeTransferFunds(

p\_from\_account\_id IN Accounts.AccountID%TYPE,

p\_to\_account\_id IN Accounts.AccountID%TYPE,

p\_amount IN Accounts.Balance%TYPE

)

IS

v\_from\_balance Accounts.Balance%TYPE;

insufficient\_funds EXCEPTION;

no\_account EXCEPTION;

BEGIN

BEGIN

**-- Check if from\_account exists and get balance**

SELECT Balance INTO v\_from\_balance

FROM Accounts

WHERE AccountID = p\_from\_account\_id;

EXCEPTION

WHEN no\_data\_found THEN

RAISE no\_account;

END;

IF v\_from\_balance < p\_amount THEN

RAISE insufficient\_funds;

END IF;

**-- Transfer funds**

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;

EXCEPTION

WHEN insufficient\_funds THEN

ROLLBACK;

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

WHEN no\_account THEN

ROLLBACK;

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

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: ' || 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.

**Query:**

CREATE OR REPLACE PROCEDURE UpdateSalary(

p\_employee\_id IN Employees.EmployeeID%TYPE,

p\_percentage IN NUMBER

)

IS

v\_salary Employees.Salary%TYPE;

employee\_not\_found EXCEPTION;

BEGIN

BEGIN

-- Check if employee exists and get current salary

SELECT Salary INTO v\_salary

FROM Employees

WHERE EmployeeID = p\_employee\_id;

EXCEPTION

WHEN no\_data\_found THEN

RAISE employee\_not\_found;

END;

**-- Update salary**

UPDATE Employees

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

WHERE EmployeeID = p\_employee\_id;

COMMIT;

EXCEPTION

WHEN employee\_not\_found THEN

DBMS\_OUTPUT.PUT\_LINE('Error: Employee ID ' || p\_employee\_id || ' not found.');

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: ' || 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.

**Query:**

CREATE OR REPLACE PROCEDURE AddNewCustomer(

p\_customer\_id IN Customers.CustomerID%TYPE,

p\_name IN Customers.Name%TYPE,

p\_dob IN Customers.DOB%TYPE,

p\_balance IN Customers.Balance%TYPE,

p\_last\_modified IN Customers.LastModified%TYPE

)

IS

duplicate\_customer EXCEPTION;

BEGIN

**-- Insert new customer**

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

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

COMMIT;

EXCEPTION

WHEN dup\_val\_on\_index THEN

RAISE duplicate\_customer;

WHEN duplicate\_customer THEN

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

WHEN OTHERS THEN

ROLLBACK;

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

END;

/

**To enable DBMS\_OUTPUT to see the output:**

SET SERVEROUTPUT ON;

**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.

**Query:**

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest

IS

v\_interest\_rate CONSTANT NUMBER := 0.01;

BEGIN

**-- Update the balance of all savings accounts by applying the interest rate**

UPDATE Accounts

SET Balance = Balance + (Balance \* v\_interest\_rate)

WHERE AccountType = 'Savings';

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.

**Query:**

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(

p\_department IN Employees.Department%TYPE,

p\_bonus\_percentage IN NUMBER

)

IS

BEGIN

**-- Update the salary of employees in the given department by adding the bonus percentage**

UPDATE Employees

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

WHERE Department = p\_department;

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.

**Query:**

CREATE OR REPLACE PROCEDURE TransferFunds(

p\_from\_account\_id IN Accounts.AccountID%TYPE,

p\_to\_account\_id IN Accounts.AccountID%TYPE,

p\_amount IN Accounts.Balance%TYPE

)

IS

v\_from\_balance Accounts.Balance%TYPE;

insufficient\_funds EXCEPTION;

BEGIN

**-- Check if the source account has sufficient balance**

SELECT Balance INTO v\_from\_balance

FROM Accounts

WHERE AccountID = p\_from\_account\_id;

IF v\_from\_balance < p\_amount THEN

RAISE insufficient\_funds;

END IF;

**-- Transfer funds**

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;

EXCEPTION

WHEN insufficient\_funds THEN

ROLLBACK;

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

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.

**Query:**

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

RETURN NUMBER

IS

v\_age NUMBER;

BEGIN

**-- Calculate age in years**

v\_age := TRUNC((SYSDATE - p\_dob) / 365.25);

RETURN v\_age;

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.

**Query:**

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(

p\_loan\_amount IN NUMBER,

p\_interest\_rate IN NUMBER,

p\_duration\_years IN NUMBER

)

RETURN NUMBER

IS

v\_monthly\_rate NUMBER;

v\_number\_of\_payments NUMBER;

v\_monthly\_installment NUMBER;

BEGIN

**-- Monthly interest rate**

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

**-- Total number of monthly payments**

v\_number\_of\_payments := p\_duration\_years \* 12;

**-- Calculate monthly installment using the formula for annuity payments**

IF v\_monthly\_rate > 0 THEN

v\_monthly\_installment := p\_loan\_amount \* v\_monthly\_rate /

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

ELSE

v\_monthly\_installment := p\_loan\_amount / v\_number\_of\_payments;

END IF;

RETURN v\_monthly\_installment;

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.

**Query:**

CREATE OR REPLACE FUNCTION HasSufficientBalance(

p\_account\_id IN Accounts.AccountID%TYPE,

p\_amount IN Accounts.Balance%TYPE

)

RETURN BOOLEAN

IS

v\_balance Accounts.Balance%TYPE;

BEGIN

-- Retrieve the current balance for the account

SELECT Balance INTO v\_balance

FROM Accounts

WHERE AccountID = p\_account\_id;

**-- Check if the balance is sufficient**

IF v\_balance >= p\_amount THEN

RETURN TRUE;

ELSE

RETURN FALSE;

END IF;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN FALSE;

END;

/

**Testing the functions using the following queries:**

**-- Test CalculateAge function**

SELECT CalculateAge(DATE '1995-02-03') AS age FROM DUAL;

**-- Test CalculateMonthlyInstallment function**

SELECT CalculateMonthlyInstallment(10000, 5, 2) AS monthly\_installment FROM DUAL;

**-- Test HasSufficientBalance function**

SELECT HasSufficientBalance(1, 500) AS has\_balance FROM DUAL;

**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.

**Query:**

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON Customers

FOR EACH ROW

BEGIN

**-- Update the LastModified column to the current date**

: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.

**Query:**

**AuditLog table to store the transaction audit records:**

CREATE TABLE AuditLog (

AuditID INT PRIMARY KEY,

TransactionID INT,

AccountID INT,

TransactionDate DATE,

Amount DECIMAL(10, 2),

TransactionType VARCHAR(10),

LogDate DATE DEFAULT SYSDATE

);

**A sequence AuditLog\_SEQ for generating unique AuditID values:**

CREATE SEQUENCE AuditLog\_SEQ START WITH 1 INCREMENT BY 1;

**Trigger:**

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

**-- Insert a record into the AuditLog table**

INSERT INTO AuditLog (AuditID, TransactionID, AccountID, TransactionDate, Amount, TransactionType)

VALUES (AuditLog\_SEQ.NEXTVAL, :NEW.TransactionID, :NEW.AccountID, :NEW.TransactionDate, :NEW.Amount, :NEW.TransactionType);

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.

**Query:**

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

DECLARE

v\_balance Accounts.Balance%TYPE;

BEGIN

**-- Retrieve the current balance for the account**

SELECT Balance INTO v\_balance

FROM Accounts

WHERE AccountID = :NEW.AccountID;

**-- Check if the transaction is a withdrawal and ensure it does not exceed the balance**

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

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

END IF;

**-- Check if the transaction is a deposit and ensure the amount is positive**

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

RAISE\_APPLICATION\_ERROR(-20002, '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.

**Query:**

DECLARE

CURSOR GenerateMonthlyStatements IS

SELECT t.TransactionID, t.AccountID, t.TransactionDate, t.Amount, t.TransactionType, c.Name

FROM Transactions t

JOIN Accounts a ON t.AccountID = a.AccountID

JOIN Customers c ON a.CustomerID = c.CustomerID

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

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

v\_transaction\_id Transactions.TransactionID%TYPE;

v\_account\_id Transactions.AccountID%TYPE;

v\_transaction\_date Transactions.TransactionDate%TYPE;

v\_amount Transactions.Amount%TYPE;

v\_transaction\_type Transactions.TransactionType%TYPE;

v\_customer\_name Customers.Name%TYPE;

BEGIN

FOR rec IN GenerateMonthlyStatements LOOP

v\_transaction\_id := rec.TransactionID;

v\_account\_id := rec.AccountID;

v\_transaction\_date := rec.TransactionDate;

v\_amount := rec.Amount;

v\_transaction\_type := rec.TransactionType;

v\_customer\_name := rec.Name;

**-- Print statement for each customer**

DBMS\_OUTPUT.PUT\_LINE('Customer: ' || v\_customer\_name);

DBMS\_OUTPUT.PUT\_LINE('Account ID: ' || v\_account\_id);

DBMS\_OUTPUT.PUT\_LINE('Transaction ID: ' || v\_transaction\_id);

DBMS\_OUTPUT.PUT\_LINE('Date: ' || v\_transaction\_date);

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

DBMS\_OUTPUT.PUT\_LINE('Type: ' || v\_transaction\_type);

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

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.

**Query:**

DECLARE

CURSOR ApplyAnnualFee IS

SELECT AccountID, Balance

FROM Accounts;

v\_account\_id Accounts.AccountID%TYPE;

v\_balance Accounts.Balance%TYPE;

v\_fee CONSTANT NUMBER := 50; **-- Example annual fee amount**

BEGIN

FOR rec IN ApplyAnnualFee LOOP

v\_account\_id := rec.AccountID;

v\_balance := rec.Balance;

**-- Deduct the annual fee from each account**

UPDATE Accounts

SET Balance = Balance - v\_fee

WHERE AccountID = v\_account\_id;

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.

**Query:**

DECLARE

CURSOR UpdateLoanInterestRates IS

SELECT LoanID, LoanAmount, InterestRate

FROM Loans;

v\_loan\_id Loans.LoanID%TYPE;

v\_loan\_amount Loans.LoanAmount%TYPE;

v\_old\_interest\_rate Loans.InterestRate%TYPE;

v\_new\_interest\_rate CONSTANT NUMBER := 0.07; **-- Example new interest rate**

BEGIN

FOR rec IN UpdateLoanInterestRates LOOP

v\_loan\_id := rec.LoanID;

v\_loan\_amount := rec.LoanAmount;

v\_old\_interest\_rate := rec.InterestRate;

**-- Update the interest rate for each loan based on the new policy**

UPDATE Loans

SET InterestRate = v\_new\_interest\_rate

WHERE LoanID = v\_loan\_id;

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.

**Query:**

**-- Package Specification**

CREATE OR REPLACE PACKAGE CustomerManagement AS

PROCEDURE AddNewCustomer(

p\_customer\_id IN Customers.CustomerID%TYPE,

p\_name IN Customers.Name%TYPE,

p\_dob IN Customers.DOB%TYPE,

p\_balance IN Customers.Balance%TYPE,

p\_last\_modified IN Customers.LastModified%TYPE

);

PROCEDURE UpdateCustomer(

p\_customer\_id IN Customers.CustomerID%TYPE,

p\_name IN Customers.Name%TYPE,

p\_dob IN Customers.DOB%TYPE,

p\_balance IN Customers.Balance%TYPE

);

FUNCTION GetCustomerBalance(

p\_customer\_id IN Customers.CustomerID%TYPE

) RETURN NUMBER;

END CustomerManagement;

/

**-- Package Body**

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

PROCEDURE AddNewCustomer(

p\_customer\_id IN Customers.CustomerID%TYPE,

p\_name IN Customers.Name%TYPE,

p\_dob IN Customers.DOB%TYPE,

p\_balance IN Customers.Balance%TYPE,

p\_last\_modified IN Customers.LastModified%TYPE

) IS

BEGIN

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

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

COMMIT;

END AddNewCustomer;

PROCEDURE UpdateCustomer(

p\_customer\_id IN Customers.CustomerID%TYPE,

p\_name IN Customers.Name%TYPE,

p\_dob IN Customers.DOB%TYPE,

p\_balance IN Customers.Balance%TYPE

) IS

BEGIN

UPDATE Customers

SET Name = p\_name, DOB = p\_dob, Balance = p\_balance, LastModified = SYSDATE

WHERE CustomerID = p\_customer\_id;

COMMIT;

END UpdateCustomer;

FUNCTION GetCustomerBalance(

p\_customer\_id IN Customers.CustomerID%TYPE

) RETURN NUMBER IS

v\_balance Customers.Balance%TYPE;

BEGIN

SELECT Balance INTO v\_balance

FROM Customers

WHERE CustomerID = p\_customer\_id;

RETURN v\_balance;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN 0;

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.

**Query:**

**-- Package Specification**

CREATE OR REPLACE PACKAGE EmployeeManagement AS

PROCEDURE HireNewEmployee(

p\_employee\_id IN Employees.EmployeeID%TYPE,

p\_name IN Employees.Name%TYPE,

p\_position IN Employees.Position%TYPE,

p\_salary IN Employees.Salary%TYPE,

p\_department IN Employees.Department%TYPE,

p\_hire\_date IN Employees.HireDate%TYPE

);

PROCEDURE UpdateEmployeeDetails(

p\_employee\_id IN Employees.EmployeeID%TYPE,

p\_name IN Employees.Name%TYPE,

p\_position IN Employees.Position%TYPE,

p\_salary IN Employees.Salary%TYPE,

p\_department IN Employees.Department%TYPE

);

FUNCTION CalculateAnnualSalary(

p\_employee\_id IN Employees.EmployeeID%TYPE

) RETURN NUMBER;

END EmployeeManagement;

/

**-- Package Body**

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

PROCEDURE HireNewEmployee(

p\_employee\_id IN Employees.EmployeeID%TYPE,

p\_name IN Employees.Name%TYPE,

p\_position IN Employees.Position%TYPE,

p\_salary IN Employees.Salary%TYPE,

p\_department IN Employees.Department%TYPE,

p\_hire\_date IN Employees.HireDate%TYPE

) IS

BEGIN

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

VALUES (p\_employee\_id, p\_name, p\_position, p\_salary, p\_department, p\_hire\_date);

COMMIT;

END HireNewEmployee;

PROCEDURE UpdateEmployeeDetails(

p\_employee\_id IN Employees.EmployeeID%TYPE,

p\_name IN Employees.Name%TYPE,

p\_position IN Employees.Position%TYPE,

p\_salary IN Employees.Salary%TYPE,

p\_department IN Employees.Department%TYPE

) IS

BEGIN

UPDATE Employees

SET Name = p\_name, Position = p\_position, Salary = p\_salary, Department = p\_department

WHERE EmployeeID = p\_employee\_id;

COMMIT;

END UpdateEmployeeDetails;

FUNCTION CalculateAnnualSalary(

p\_employee\_id IN Employees.EmployeeID%TYPE

) RETURN NUMBER IS

v\_salary Employees.Salary%TYPE;

BEGIN

SELECT Salary INTO v\_salary

FROM Employees

WHERE EmployeeID = p\_employee\_id;

RETURN v\_salary \* 12;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN 0;

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.

**Query:**

**-- Package Specification**

CREATE OR REPLACE PACKAGE AccountOperations AS

PROCEDURE OpenNewAccount(

p\_account\_id IN Accounts.AccountID%TYPE,

p\_customer\_id IN Accounts.CustomerID%TYPE,

p\_account\_type IN Accounts.AccountType%TYPE,

p\_balance IN Accounts.Balance%TYPE

);

PROCEDURE CloseAccount(

p\_account\_id IN Accounts.AccountID%TYPE

);

FUNCTION GetTotalBalance(

p\_customer\_id IN Accounts.CustomerID%TYPE

) RETURN NUMBER;

END AccountOperations;

/

**-- Package Body**

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

PROCEDURE OpenNewAccount(

p\_account\_id IN Accounts.AccountID%TYPE,

p\_customer\_id IN Accounts.CustomerID%TYPE,

p\_account\_type IN Accounts.AccountType%TYPE,

p\_balance IN Accounts.Balance%TYPE

) IS

BEGIN

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

VALUES (p\_account\_id, p\_customer\_id, p\_account\_type, p\_balance, SYSDATE);

COMMIT;

END OpenNewAccount;

PROCEDURE CloseAccount(

p\_account\_id IN Accounts.AccountID%TYPE

) IS

BEGIN

DELETE FROM Accounts

WHERE AccountID = p\_account\_id;

COMMIT;

END CloseAccount;

FUNCTION GetTotalBalance(

p\_customer\_id IN Accounts.CustomerID%TYPE

) RETURN NUMBER IS

v\_total\_balance NUMBER := 0;

BEGIN

SELECT SUM(Balance) INTO v\_total\_balance

FROM Accounts

WHERE CustomerID = p\_customer\_id;

RETURN NVL(v\_total\_balance, 0);

END GetTotalBalance;

END AccountOperations;

/