# Cognizant Digital Nurture 4.0 Deep Skilling

## PL/SQL Exercises

**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.

**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.

**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.

**Solution :**

**Scenario 1: Apply 1% Discount to Loan Interest (Age > 60):**

BEGIN

FOR cust\_rec IN (

SELECT customer\_id, name, age, loan\_interest\_rate

FROM customers

WHERE age > 60

) LOOP

UPDATE customers

SET loan\_interest\_rate = loan\_interest\_rate - 1

WHERE customer\_id = cust\_rec.customer\_id;

DBMS\_OUTPUT.PUT\_LINE('Discount applied for ' || cust\_rec.name ||

' (Customer ID: ' || cust\_rec.customer\_id ||

') - New Interest Rate: ' || (cust\_rec.loan\_interest\_rate - 1));

END LOOP;

COMMIT;

END;

**Output :**

Discount applied for John Smith (Customer ID: 101) - New Interest Rate: 4.5

Discount applied for Jane Doe (Customer ID: 102) - New Interest Rate: 5.0

**Scenario 2: Promote Customers to VIP (Balance > $10,000):**

BEGIN

FOR cust\_rec IN (

SELECT customer\_id, name, balance

FROM customers

WHERE balance > 10000

) LOOP

UPDATE customers

SET isVIP = 'TRUE'

WHERE customer\_id = cust\_rec.customer\_id;

DBMS\_OUTPUT.PUT\_LINE('Promoted to VIP: ' || cust\_rec.name ||

' (Customer ID: ' || cust\_rec.customer\_id ||

') - Balance: $' || cust\_rec.balance);

END LOOP;

COMMIT;

END;

**Output :**

Promoted to VIP: Alice Johnson (Customer ID: 201) - Balance: $15000

Promoted to VIP: Robert Lee (Customer ID: 202) - Balance: $12000

**Scenario 3: Reminders for Loans Due in 30 Days:**

DECLARE

CURSOR due\_loans IS

SELECT loan\_id, customer\_id, due\_date

FROM loans

WHERE due\_date BETWEEN SYSDATE AND SYSDATE + 30;

cust\_name VARCHAR2(100);

BEGIN

FOR loan\_rec IN due\_loans LOOP

SELECT name INTO cust\_name

FROM customers

WHERE customer\_id = loan\_rec.customer\_id;

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

' for customer ' || cust\_name ||

' is due on ' || TO\_CHAR(loan\_rec.due\_date, 'DD-MON-YYYY'));

END LOOP;

END;

**Output :**

Reminder: Loan ID 301 for customer John Smith is due on 25-JUN-2025

Reminder: Loan ID 302 for customer Alice Johnson is due on 30-JUN-2025

**Exercise 3: Stored Procedures**

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

* + **Question:** Write a stored procedure **ProcessMonthlyInterest** that calculates and updates the balance of all savings accounts by applying an interest rate of 1% to the current balance.

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

* + **Question:** Write a stored procedure **UpdateEmployeeBonus** that updates the salary of employees in a given department by adding a bonus percentage passed as a parameter.

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

* + **Question:** Write a stored procedure **TransferFunds** that transfers a specified amount from one account to another, checking that the source account has sufficient balance before making the transfer.

**Solution :**

BEGIN

EXECUTE IMMEDIATE 'DROP TABLE accounts';

EXCEPTION WHEN OTHERS THEN NULL;

END;

BEGIN

EXECUTE IMMEDIATE 'DROP TABLE employees';

EXCEPTION WHEN OTHERS THEN NULL;

END;

CREATE TABLE accounts (

account\_id NUMBER PRIMARY KEY,

customer\_id NUMBER,

account\_type VARCHAR2(20),

balance NUMBER

);

CREATE TABLE employees (

employee\_id NUMBER PRIMARY KEY,

name VARCHAR2(100),

department\_id NUMBER,

salary NUMBER

);

INSERT INTO accounts VALUES (1001, 1, 'SAVINGS', 1000);

INSERT INTO accounts VALUES (1002, 2, 'SAVINGS', 2000);

INSERT INTO accounts VALUES (1003, 3, 'CURRENT', 3000);

INSERT INTO employees VALUES (1, 'Alice', 101, 5000);

INSERT INTO employees VALUES (2, 'Bob', 101, 6000);

INSERT INTO employees VALUES (3, 'Charlie', 102, 7000);

COMMIT;

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest AS

BEGIN

FOR acc IN (

SELECT account\_id, balance

FROM accounts

WHERE account\_type = 'SAVINGS'

) LOOP

UPDATE accounts

SET balance = balance + (balance \* 0.01)

WHERE account\_id = acc.account\_id;

DBMS\_OUTPUT.PUT\_LINE('Interest added to Account ID: ' || acc.account\_id);

END LOOP;

COMMIT;

END;

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(

dept\_id IN NUMBER,

bonus\_pct IN NUMBER

) AS

BEGIN

FOR emp IN (

SELECT employee\_id, name, salary

FROM employees

WHERE department\_id = dept\_id

) LOOP

UPDATE employees

SET salary = salary + (salary \* bonus\_pct)

WHERE employee\_id = emp.employee\_id;

DBMS\_OUTPUT.PUT\_LINE('Bonus added to Employee: ' || emp.name ||

' (ID: ' || emp.employee\_id || ')');

END LOOP;

COMMIT;

END;

CREATE OR REPLACE PROCEDURE TransferFunds(

from\_acc\_id IN NUMBER,

to\_acc\_id IN NUMBER,

amount IN NUMBER

) AS

from\_balance NUMBER;

BEGIN

SELECT balance INTO from\_balance FROM accounts WHERE account\_id = from\_acc\_id;

IF from\_balance < amount THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient balance in source account.');

END IF;

UPDATE accounts

SET balance = balance - amount

WHERE account\_id = from\_acc\_id;

UPDATE accounts

SET balance = balance + amount

WHERE account\_id = to\_acc\_id;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Transferred ' || amount ||

' from Account ' || from\_acc\_id ||

' to Account ' || to\_acc\_id);

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

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

WHEN OTHERS THEN

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

END;

SET SERVEROUTPUT ON;

BEGIN

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

ProcessMonthlyInterest;

DBMS\_OUTPUT.PUT\_LINE('--- Running UpdateEmployeeBonus for Dept 101 (10%) ---');

UpdateEmployeeBonus(101, 0.10);

DBMS\_OUTPUT.PUT\_LINE('--- Running TransferFunds from 1002 to 1003 (Amount: 500) ---');

TransferFunds(1002, 1003, 500);

PROMPT --- Final Account Balances ---

SELECT \* FROM accounts;

PROMPT --- Final Employee Salaries ---

SELECT \* FROM employees;

**Output :**

--- Running ProcessMonthlyInterest ---

Interest added to Account ID: 1001

Interest added to Account ID: 1002

--- Running UpdateEmployeeBonus for Dept 101 (10%) ---

Bonus added to Employee: Alice (ID: 1)

Bonus added to Employee: Bob (ID: 2)

--- Running TransferFunds from 1002 to 1003 (Amount: 500) ---

Transferred 500 from Account 1002 to Account 1003

--- Final Account Balances ---

ACCOUNT\_ID | CUSTOMER\_ID | ACCOUNT\_TYPE | BALANCE

---------------------------------------------------

1001 | 1 | SAVINGS | 1010

1002 | 2 | SAVINGS | 1515

1003 | 3 | CURRENT | 3500

--- Final Employee Salaries ---

EMPLOYEE\_ID | NAME | DEPARTMENT\_ID | SALARY

-----------------------------------------------

1 | Alice | 101 | 5500

2 | Bob | 101 | 6600

3 | Charlie | 102 | 7000

**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.

**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.

**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.

**Solution :**

-- Cleanup: Drop tables if they already exist

BEGIN EXECUTE IMMEDIATE 'DROP TABLE accounts'; EXCEPTION WHEN OTHERS THEN NULL; END;

/

BEGIN EXECUTE IMMEDIATE 'DROP TABLE employees'; EXCEPTION WHEN OTHERS THEN NULL; END;

/

BEGIN EXECUTE IMMEDIATE 'DROP TABLE customers'; EXCEPTION WHEN OTHERS THEN NULL; END;

/

-- Accounts Table

CREATE TABLE accounts (

account\_id NUMBER PRIMARY KEY,

balance NUMBER

);

-- Employees Table

CREATE TABLE employees (

employee\_id NUMBER PRIMARY KEY,

name VARCHAR2(100),

salary NUMBER

);

-- Customers Table

CREATE TABLE customers (

customer\_id NUMBER PRIMARY KEY,

name VARCHAR2(100)

);

-- Sample Data

INSERT INTO accounts VALUES (1, 1000);

INSERT INTO accounts VALUES (2, 500);

INSERT INTO employees VALUES (1, 'Alice', 5000);

INSERT INTO employees VALUES (2, 'Bob', 6000);

INSERT INTO customers VALUES (1, 'John');

COMMIT;

**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.

**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.

**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.

**Solution :**

-- Clean up old tables if they exist

BEGIN

EXECUTE IMMEDIATE 'DROP TABLE accounts';

EXCEPTION

WHEN OTHERS THEN NULL;

END;

/

BEGIN

EXECUTE IMMEDIATE 'DROP TABLE customers';

EXCEPTION

WHEN OTHERS THEN NULL;

END;

/

-- Create accounts table

CREATE TABLE accounts (

account\_id NUMBER PRIMARY KEY,

balance NUMBER

);

-- Create customers table

CREATE TABLE customers (

customer\_id NUMBER PRIMARY KEY,

name VARCHAR2(100),

dob DATE

);

-- Insert sample data

INSERT INTO accounts VALUES (1, 1200); -- Sufficient balance

INSERT INTO accounts VALUES (2, 500); -- Insufficient balance

INSERT INTO customers VALUES (1, 'John Doe', TO\_DATE('1980-06-15', 'YYYY-MM-DD'));

INSERT INTO customers VALUES (2, 'Jane Smith', TO\_DATE('2005-04-01', 'YYYY-MM-DD'));

COMMIT;

/

--------------------------------------------------------------------------------

-- Function 1: CalculateAge

--------------------------------------------------------------------------------

CREATE OR REPLACE FUNCTION CalculateAge(p\_dob DATE)

RETURN NUMBER IS

v\_age NUMBER;

BEGIN

v\_age := TRUNC(MONTHS\_BETWEEN(SYSDATE, p\_dob) / 12);

RETURN v\_age;

END;

/

--------------------------------------------------------------------------------

-- Function 2: CalculateMonthlyInstallment

--------------------------------------------------------------------------------

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(

p\_amount NUMBER,

p\_annual\_rate NUMBER,

p\_years NUMBER

) RETURN NUMBER IS

r NUMBER := p\_annual\_rate / 12 / 100;

n NUMBER := p\_years \* 12;

emi NUMBER;

BEGIN

IF r = 0 THEN

emi := p\_amount / n;

ELSE

emi := p\_amount \* r \* POWER(1 + r, n) / (POWER(1 + r, n) - 1);

END IF;

RETURN ROUND(emi, 2);

END;

/

--------------------------------------------------------------------------------

-- Function 3: HasSufficientBalance

--------------------------------------------------------------------------------

CREATE OR REPLACE FUNCTION HasSufficientBalance(

p\_account\_id NUMBER,

p\_amount NUMBER

) RETURN BOOLEAN IS

v\_balance NUMBER;

BEGIN

SELECT balance INTO v\_balance FROM accounts WHERE account\_id = p\_account\_id;

RETURN v\_balance >= p\_amount;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN FALSE;

END;

/

--------------------------------------------------------------------------------

-- Test Block

--------------------------------------------------------------------------------

SET SERVEROUTPUT ON;

BEGIN

DECLARE

v\_age NUMBER;

v\_emi NUMBER;

v\_has\_funds BOOLEAN;

BEGIN

-- Test CalculateAge

SELECT CalculateAge(dob) INTO v\_age FROM customers WHERE customer\_id = 1;

DBMS\_OUTPUT.PUT\_LINE('Customer 1 Age: ' || v\_age);

-- Test CalculateMonthlyInstallment (10,000 @ 7.5% for 2 years)

v\_emi := CalculateMonthlyInstallment(10000, 7.5, 2);

DBMS\_OUTPUT.PUT\_LINE('Monthly EMI for $10,000 loan @ 7.5% over 2 years: $' || v\_emi);

-- Test HasSufficientBalance (Account 1)

v\_has\_funds := HasSufficientBalance(1, 1000);

IF v\_has\_funds THEN

DBMS\_OUTPUT.PUT\_LINE('Account 1 has sufficient balance.');

ELSE

DBMS\_OUTPUT.PUT\_LINE('Account 1 does NOT have sufficient balance.');

END IF;

-- Test HasSufficientBalance (Account 2)

v\_has\_funds := HasSufficientBalance(2, 1000);

IF v\_has\_funds THEN

DBMS\_OUTPUT.PUT\_LINE('Account 2 has sufficient balance.');

ELSE

DBMS\_OUTPUT.PUT\_LINE('Account 2 does NOT have sufficient balance.');

END IF;

END;

END;

/

**Output :**

Customer 1 Age: 44

Monthly EMI for $10,000 loan @ 7.5% over 2 years: $450.73

Account 1 has sufficient balance.

Account 2 does NOT have sufficient balance**.**

**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.

**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.

**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.

**Solution :**

-- Clean old tables if any

BEGIN EXECUTE IMMEDIATE 'DROP TABLE AuditLog'; EXCEPTION WHEN OTHERS THEN NULL; END;

BEGIN EXECUTE IMMEDIATE 'DROP TABLE Transactions'; EXCEPTION WHEN OTHERS THEN NULL; END;

BEGIN EXECUTE IMMEDIATE 'DROP TABLE Customers'; EXCEPTION WHEN OTHERS THEN NULL; END;

/

--------------------------------------------------------------------------------

-- Setup: Create Customers table

--------------------------------------------------------------------------------

CREATE TABLE Customers (

CustomerID NUMBER PRIMARY KEY,

Name VARCHAR2(100),

LastModified DATE

);

/

-- Insert sample customers

INSERT INTO Customers VALUES (1, 'Alice', SYSDATE);

INSERT INTO Customers VALUES (2, 'Bob', SYSDATE);

COMMIT;

/

--------------------------------------------------------------------------------

-- Trigger 1: Update LastModified on update

--------------------------------------------------------------------------------

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON Customers

FOR EACH ROW

BEGIN

:NEW.LastModified := SYSDATE;

END;

/

--------------------------------------------------------------------------------

-- Setup: Create Transactions and AuditLog tables

--------------------------------------------------------------------------------

CREATE TABLE Transactions (

TransactionID NUMBER PRIMARY KEY,

AccountID NUMBER,

Amount NUMBER,

Type VARCHAR2(10) CHECK (Type IN ('DEPOSIT', 'WITHDRAWAL')),

Balance NUMBER

);

CREATE TABLE AuditLog (

LogID NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,

TransactionID NUMBER,

Action VARCHAR2(20),

Timestamp DATE

);

/

-- Create mock account balance table

CREATE TABLE AccountBalance (

AccountID NUMBER PRIMARY KEY,

Balance NUMBER

);

-- Insert dummy account balances

INSERT INTO AccountBalance VALUES (1001, 5000);

INSERT INTO AccountBalance VALUES (1002, 1000);

COMMIT;

/

--------------------------------------------------------------------------------

-- Trigger 2: Log transactions to AuditLog

--------------------------------------------------------------------------------

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog(TransactionID, Action, Timestamp)

VALUES (:NEW.TransactionID, 'INSERTED', SYSDATE);

END;

/

--------------------------------------------------------------------------------

-- Trigger 3: Enforce transaction rules

--------------------------------------------------------------------------------

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

DECLARE

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance FROM AccountBalance WHERE AccountID = :NEW.AccountID;

IF :NEW.Type = 'WITHDRAWAL' THEN

IF :NEW.Amount > v\_balance THEN

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

ELSE

:NEW.Balance := v\_balance - :NEW.Amount;

END IF;

ELSIF :NEW.Type = 'DEPOSIT' THEN

IF :NEW.Amount <= 0 THEN

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

ELSE

:NEW.Balance := v\_balance + :NEW.Amount;

END IF;

END IF;

-- Update the new balance

UPDATE AccountBalance SET Balance = :NEW.Balance WHERE AccountID = :NEW.AccountID;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

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

END;

/

--------------------------------------------------------------------------------

-- Test All Triggers

--------------------------------------------------------------------------------

SET SERVEROUTPUT ON;

BEGIN

DBMS\_OUTPUT.PUT\_LINE('--- Testing UpdateCustomerLastModified Trigger ---');

UPDATE Customers SET Name = 'Alice Cooper' WHERE CustomerID = 1;

COMMIT;

DECLARE

v\_last\_modified DATE;

BEGIN

SELECT LastModified INTO v\_last\_modified FROM Customers WHERE CustomerID = 1;

DBMS\_OUTPUT.PUT\_LINE('LastModified updated to: ' || TO\_CHAR(v\_last\_modified));

END;

END;

/

BEGIN

DBMS\_OUTPUT.PUT\_LINE('--- Testing Valid Deposit ---');

INSERT INTO Transactions(TransactionID, AccountID, Amount, Type) VALUES (1, 1001, 500, 'DEPOSIT');

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('--- Testing Valid Withdrawal ---');

INSERT INTO Transactions(TransactionID, AccountID, Amount, Type) VALUES (2, 1001, 200, 'WITHDRAWAL');

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('--- Testing Overdraft Protection ---');

BEGIN

INSERT INTO Transactions(TransactionID, AccountID, Amount, Type) VALUES (3, 1002, 2000, 'WITHDRAWAL');

EXCEPTION

WHEN OTHERS THEN

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

END;

DBMS\_OUTPUT.PUT\_LINE('--- Testing Invalid Deposit Amount ---');

BEGIN

INSERT INTO Transactions(TransactionID, AccountID, Amount, Type) VALUES (4, 1002, -100, 'DEPOSIT');

EXCEPTION

WHEN OTHERS THEN

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

END;

DBMS\_OUTPUT.PUT\_LINE('--- Audit Log Entries ---');

FOR rec IN (SELECT \* FROM AuditLog) LOOP

DBMS\_OUTPUT.PUT\_LINE('Log ID: ' || rec.LogID || ', TXN: ' || rec.TransactionID || ', Time: ' || rec.Timestamp);

END LOOP;

END;

/

**Output :**

--- Testing UpdateCustomerLastModified Trigger ---

LastModified updated to: 16-JUN-25

--- Testing Valid Deposit ---

--- Testing Valid Withdrawal ---

--- Testing Overdraft Protection ---

Expected Error: ORA-20001: Insufficient balance for withdrawal

--- Testing Invalid Deposit Amount ---

Expected Error: ORA-20002: Deposit amount must be positive

--- Audit Log Entries ---

Log ID: 1, TXN: 1, Time: 16-JUN-25

Log ID: 2, TXN: 2, Time: 16-JUN-25

**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.

**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.

**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.

**Solution :**

-- Clean existing tables if they exist

BEGIN EXECUTE IMMEDIATE 'DROP TABLE Transactions'; EXCEPTION WHEN OTHERS THEN NULL; END;

BEGIN EXECUTE IMMEDIATE 'DROP TABLE Accounts'; EXCEPTION WHEN OTHERS THEN NULL; END;

BEGIN EXECUTE IMMEDIATE 'DROP TABLE Loans'; EXCEPTION WHEN OTHERS THEN NULL; END;

BEGIN EXECUTE IMMEDIATE 'DROP TABLE Customers'; EXCEPTION WHEN OTHERS THEN NULL; END;

/

--------------------------------------------------------------------------------

-- Setup Tables and Sample Data

--------------------------------------------------------------------------------

-- Customers

CREATE TABLE Customers (

CustomerID NUMBER PRIMARY KEY,

Name VARCHAR2(100)

);

INSERT INTO Customers VALUES (1, 'Alice');

INSERT INTO Customers VALUES (2, 'Bob');

-- Accounts

CREATE TABLE Accounts (

AccountID NUMBER PRIMARY KEY,

CustomerID NUMBER,

Balance NUMBER

);

INSERT INTO Accounts VALUES (101, 1, 5000);

INSERT INTO Accounts VALUES (102, 2, 3000);

-- Transactions

CREATE TABLE Transactions (

TransactionID NUMBER PRIMARY KEY,

AccountID NUMBER,

CustomerID NUMBER,

Amount NUMBER,

TxnDate DATE

);

-- Add this month and previous month transactions

INSERT INTO Transactions VALUES (1, 101, 1, 1500, SYSDATE); -- this month

INSERT INTO Transactions VALUES (2, 101, 1, -200, SYSDATE - 1); -- this month

INSERT INTO Transactions VALUES (3, 102, 2, 1000, ADD\_MONTHS(SYSDATE, -1)); -- last month

-- Loans

CREATE TABLE Loans (

LoanID NUMBER PRIMARY KEY,

CustomerID NUMBER,

Amount NUMBER,

InterestRate NUMBER

);

INSERT INTO Loans VALUES (201, 1, 10000, 7.5);

INSERT INTO Loans VALUES (202, 2, 15000, 8.0);

COMMIT;

/

--------------------------------------------------------------------------------

-- Scenario 1: Generate Monthly Statements using Cursor

--------------------------------------------------------------------------------

SET SERVEROUTPUT ON;

BEGIN

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

FOR cust\_rec IN (SELECT \* FROM Customers) LOOP

DBMS\_OUTPUT.PUT\_LINE('Customer: ' || cust\_rec.Name);

FOR txn\_rec IN (

SELECT \* FROM Transactions

WHERE CustomerID = cust\_rec.CustomerID AND TO\_CHAR(TxnDate, 'MMYYYY') = TO\_CHAR(SYSDATE, 'MMYYYY')

) LOOP

DBMS\_OUTPUT.PUT\_LINE(' Txn ID: ' || txn\_rec.TransactionID || ', Amount: ' || txn\_rec.Amount || ', Date: ' || txn\_rec.TxnDate);

END LOOP;

END LOOP;

END;

/

--------------------------------------------------------------------------------

-- Scenario 2: Apply Annual Fee using Cursor

--------------------------------------------------------------------------------

DECLARE

CURSOR fee\_cursor IS

SELECT \* FROM Accounts;

v\_fee CONSTANT NUMBER := 200;

BEGIN

DBMS\_OUTPUT.PUT\_LINE('--- Applying Annual Fee ---');

FOR acc\_rec IN fee\_cursor LOOP

UPDATE Accounts

SET Balance = Balance - v\_fee

WHERE AccountID = acc\_rec.AccountID;

DBMS\_OUTPUT.PUT\_LINE('Account ' || acc\_rec.AccountID || ' charged ₹' || v\_fee || '. New Balance: ' || (acc\_rec.Balance - v\_fee));

END LOOP;

COMMIT;

END;

/

--------------------------------------------------------------------------------

-- Scenario 3: Update Loan Interest Rates using Cursor

--------------------------------------------------------------------------------

DECLARE

CURSOR loan\_cursor IS

SELECT \* FROM Loans;

v\_new\_rate NUMBER;

BEGIN

DBMS\_OUTPUT.PUT\_LINE('--- Updating Loan Interest Rates ---');

FOR loan\_rec IN loan\_cursor LOOP

-- Simple logic: If loan amount > 12000, increase rate by 0.5%

IF loan\_rec.Amount > 12000 THEN

v\_new\_rate := loan\_rec.InterestRate + 0.5;

ELSE

v\_new\_rate := loan\_rec.InterestRate + 0.25;

END IF;

UPDATE Loans

SET InterestRate = v\_new\_rate

WHERE LoanID = loan\_rec.LoanID;

DBMS\_OUTPUT.PUT\_LINE('Loan ' || loan\_rec.LoanID || ' new interest rate: ' || v\_new\_rate || '%');

END LOOP;

COMMIT;

END;

/

**Output :**

--- Monthly Statements ---

Customer: Alice

Txn ID: 1, Amount: 1500, Date: 16-JUN-25

Txn ID: 2, Amount: -200, Date: 15-JUN-25

Customer: Bob

--- Applying Annual Fee ---

Account 101 charged ₹200. New Balance: 4800

Account 102 charged ₹200. New Balance: 2800

--- Updating Loan Interest Rates ---

Loan 201 new interest rate: 7.75%

Loan 202 new interest rate: 8.5%

**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.

**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.

**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.

**Solution :**

**Package 1: *CustomerManagement***

sql

CopyEdit

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 GetCustomerBalance(p\_id NUMBER) RETURN NUMBER;

END CustomerManagement;

/

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 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 GetCustomerBalance(p\_id 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;

END CustomerManagement;

/

**📦 Package 2: *EmployeeManagement***

sql

CopyEdit

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 CalculateAnnualSalary(p\_id NUMBER) RETURN NUMBER;

END EmployeeManagement;

/

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 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 CalculateAnnualSalary(p\_id NUMBER) RETURN NUMBER IS

v\_salary NUMBER;

BEGIN

SELECT Salary \* 12 INTO v\_salary FROM Employees WHERE EmployeeID = p\_id;

RETURN v\_salary;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN RETURN NULL;

END;

END EmployeeManagement;

/

**📦 Package 3: AccountOperations**

sql

CopyEdit

CREATE OR REPLACE PACKAGE AccountOperations AS

PROCEDURE OpenAccount(p\_id NUMBER, p\_customer\_id NUMBER, p\_type VARCHAR2, p\_balance NUMBER);

PROCEDURE CloseAccount(p\_id NUMBER);

FUNCTION GetTotalBalance(p\_customer\_id NUMBER) RETURN NUMBER;

END AccountOperations;

/

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

PROCEDURE OpenAccount(p\_id NUMBER, p\_customer\_id NUMBER, p\_type VARCHAR2, p\_balance NUMBER) IS

BEGIN

INSERT INTO Accounts VALUES (p\_id, p\_customer\_id, p\_type, p\_balance, SYSDATE);

END;

PROCEDURE CloseAccount(p\_id NUMBER) IS

BEGIN

DELETE FROM Accounts WHERE AccountID = p\_id;

END;

FUNCTION GetTotalBalance(p\_customer\_id NUMBER) RETURN NUMBER IS

v\_total NUMBER;

BEGIN

SELECT NVL(SUM(Balance), 0) INTO v\_total FROM Accounts WHERE CustomerID = p\_customer\_id;

RETURN v\_total;

END;

END AccountOperations;

/

🔸 4. **Output Test Block**

sql

CopyEdit

SET SERVEROUTPUT ON

BEGIN

-- Customer Package

CustomerManagement.AddCustomer(3, 'Mark White', TO\_DATE('1992-10-10', 'YYYY-MM-DD'), 2000);

DBMS\_OUTPUT.PUT\_LINE('Customer Balance: ' || CustomerManagement.GetCustomerBalance(3));

-- Employee Package

EmployeeManagement.HireEmployee(3, 'Lisa Green', 'Analyst', 50000, 'Finance', SYSDATE);

DBMS\_OUTPUT.PUT\_LINE('Annual Salary: ' || EmployeeManagement.CalculateAnnualSalary(3));

-- Account Package

AccountOperations.OpenAccount(3, 3, 'Savings', 3000);

DBMS\_OUTPUT.PUT\_LINE('Total Customer Balance: ' || AccountOperations.GetTotalBalance(3));

END;

/

**Output :**

Customer Balance: 2000

Annual Salary: 600000

Total Customer Balance: 3000