**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.

**Code:**

CREATE TABLE CUSTOMERS (

CustomerID NUMBER PRIMARY KEY,

Name VARCHAR2(100),

Age NUMBER,

Balance NUMBER(10,2),

IsVIP VARCHAR2(5), -- Will use 'TRUE' or 'FALSE'

InterestRate NUMBER(5,2)

);

CREATE TABLE LOANS (

LoanID NUMBER PRIMARY KEY,

CustomerID NUMBER,

DueDate DATE,

Amount NUMBER(10,2),

FOREIGN KEY (CustomerID) REFERENCES CUSTOMERS(CustomerID)

);

INSERT INTO CUSTOMERS VALUES (1, 'John Doe', 65, 12000, 'FALSE', 7.5);

INSERT INTO CUSTOMERS VALUES (2, 'Jane Smith', 45, 9000, 'FALSE', 6.8);

INSERT INTO CUSTOMERS VALUES (3, 'Robert Brown', 70, 15000, 'FALSE', 8.2);

INSERT INTO CUSTOMERS VALUES (4, 'Emily White', 30, 11000, 'FALSE', 7.0);

INSERT INTO LOANS VALUES (101, 1, SYSDATE + 10, 5000);

INSERT INTO LOANS VALUES (102, 2, SYSDATE + 40, 6000);

INSERT INTO LOANS VALUES (103, 3, SYSDATE + 5, 7000);

INSERT INTO LOANS VALUES (104, 4, SYSDATE + 25, 3000);

COMMIT;

**--Scenario 1: Apply 1% Discount to Interest Rate if Age > 60**

BEGIN

FOR cust IN (SELECT \* FROM CUSTOMERS) LOOP

IF cust.Age > 60 THEN

UPDATE CUSTOMERS

SET InterestRate = InterestRate - 1

WHERE CustomerID = cust.CustomerID;

DBMS\_OUTPUT.PUT\_LINE('Discount applied to ' || cust.Name || '. New Interest Rate: ' || (cust.InterestRate - 1));

END IF;

END LOOP;

END;

/

**Output:**

****

**-- Scenario 2: Promote Customers to VIP if Balance > 10,000**

BEGIN

FOR cust IN (SELECT \* FROM CUSTOMERS) LOOP

IF cust.Balance > 10000 THEN

UPDATE CUSTOMERS

SET IsVIP = 'TRUE'

WHERE CustomerID = cust.CustomerID;

DBMS\_OUTPUT.PUT\_LINE(cust.Name || ' has been promoted to VIP.');

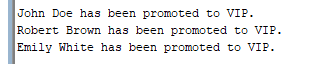
END IF;

END LOOP;

END;

/

**Output:**

****

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

BEGIN

FOR loan\_rec IN (

SELECT L.LoanID, C.Name, L.DueDate

FROM LOANS L

JOIN CUSTOMERS C ON L.CustomerID = C.CustomerID

WHERE L.DueDate BETWEEN SYSDATE AND SYSDATE + 30

) LOOP

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

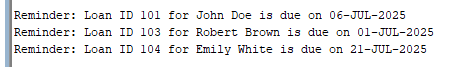
loan\_rec.Name || ' is due on ' || TO\_CHAR(loan\_rec.DueDate, 'DD-MON-YYYY'));

END LOOP;

END;

/

**Output:**

****

**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.

**Code:**

DROP TABLE ErrorLog;

-- Table for storing customers

CREATE TABLE Customer (

CustomerID NUMBER PRIMARY KEY,

Name VARCHAR2(100),

DOB DATE,

Balance NUMBER,

LastModified DATE

);

-- Table for accounts

CREATE TABLE Accounts (

AccountID NUMBER PRIMARY KEY,

CustomerID NUMBER,

Balance NUMBER,

LastModified DATE,

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

-- Table for employees

CREATE TABLE Employees (

EmployeeID NUMBER PRIMARY KEY,

Name VARCHAR2(100),

Position VARCHAR2(50),

Salary NUMBER,

Department VARCHAR2(50),

HireDate DATE

);

-- Error log table

CREATE TABLE ErrorLog (

LogID NUMBER GENERATED BY DEFAULT ON NULL AS IDENTITY PRIMARY KEY,

ErrorMessage VARCHAR2(4000),

LogDate DATE DEFAULT SYSDATE

);

**--Scenario 1: SafeTransferFunds**

CREATE OR REPLACE PROCEDURE SafeTransferFunds(

p\_from\_acc\_id IN NUMBER,

p\_to\_acc\_id IN NUMBER,

p\_amount IN NUMBER

) IS

v\_from\_balance NUMBER;

v\_dummy NUMBER;

BEGIN

BEGIN

SELECT 1 INTO v\_dummy FROM Accounts WHERE AccountID = p\_from\_acc\_id;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RAISE\_APPLICATION\_ERROR(-20010, 'Source account does not exist.');

END;

BEGIN

SELECT 1 INTO v\_dummy FROM Accounts WHERE AccountID = p\_to\_acc\_id;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RAISE\_APPLICATION\_ERROR(-20011, 'Target account does not exist.');

END;

-- Get current balance

SELECT Balance INTO v\_from\_balance FROM Accounts WHERE AccountID = p\_from\_acc\_id;

IF v\_from\_balance < p\_amount THEN

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

END IF;

-- Start transfer

UPDATE Accounts SET Balance = Balance - p\_amount WHERE AccountID = p\_from\_acc\_id;

UPDATE Accounts SET Balance = Balance + p\_amount WHERE AccountID = p\_to\_acc\_id;

COMMIT;

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

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END;

/

BEGIN

SafeTransferFunds(1, 2, 500);

END;

/

**Output:**

****

**--Scenario 2: UpdateSalary**

CREATE OR REPLACE PROCEDURE UpdateSalary(

p\_emp\_id IN NUMBER,

p\_percent IN NUMBER

) IS

BEGIN

UPDATE Employees

SET Salary = Salary + (Salary \* p\_percent / 100)

WHERE EmployeeID = p\_emp\_id;

IF SQL%ROWCOUNT = 0 THEN

RAISE\_APPLICATION\_ERROR(-20002, 'Employee not found.');

END IF;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END;

/

BEGIN

UpdateSalary(1, 10); -- Give 10% raise to employee 1

END;

/

**Output:**

****

**--Scenario 3: AddNewCustomer**

CREATE OR REPLACE PROCEDURE AddNewCustomer(

p\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_dob IN DATE,

p\_balance IN NUMBER

) IS

BEGIN

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

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

COMMIT;

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

DBMS\_OUTPUT.PUT\_LINE('Customer with this ID already exists.');

WHEN OTHERS THEN

ROLLBACK;

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

END;

/

BEGIN

AddNewCustomer(3, 'Alice Cooper', TO\_DATE('1992-12-01', 'YYYY-MM-DD'), 1200);

END;

/

**Output:**

****

**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.

**Code:**

CREATE TABLE ACCOUNTS (

AccountID NUMBER PRIMARY KEY,

HolderName VARCHAR2(100),

AccountType VARCHAR2(20),

Balance NUMBER(10,2)

);

CREATE TABLE EMPLOYEES (

EmpID NUMBER PRIMARY KEY,

EmpName VARCHAR2(100),

Department VARCHAR2(50),

Salary NUMBER(10,2)

);

INSERT INTO ACCOUNTS VALUES (101, 'Alice', 'SAVINGS', 10000);

INSERT INTO ACCOUNTS VALUES (102, 'Bob', 'CURRENT', 7000);

INSERT INTO ACCOUNTS VALUES (103, 'Charlie', 'SAVINGS', 15000);

INSERT INTO EMPLOYEES VALUES (1, 'John', 'HR', 40000);

INSERT INTO EMPLOYEES VALUES (2, 'Jane', 'SALES', 45000);

INSERT INTO EMPLOYEES VALUES (3, 'Smith', 'SALES', 42000);

COMMIT;

**--Scenario 1: Process Monthly Interest for All SAVINGS Accounts**

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

BEGIN

FOR acc IN (SELECT AccountID, Balance FROM ACCOUNTS WHERE AccountType = 'SAVINGS') LOOP

UPDATE ACCOUNTS

SET Balance = Balance + (Balance \* 0.01)

WHERE AccountID = acc.AccountID;

DBMS\_OUTPUT.PUT\_LINE('Interest added to Account ID ' || acc.AccountID ||

'. New Balance will be updated on commit.');

END LOOP;

COMMIT;

END;

/

BEGIN

ProcessMonthlyInterest;

END;

/

**Output:**

****

**--Scenario 2: Add Bonus to Employees in a Department**

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(

p\_Department IN VARCHAR2,

p\_BonusPct IN NUMBER

) IS

BEGIN

UPDATE EMPLOYEES

SET Salary = Salary + (Salary \* p\_BonusPct / 100)

WHERE Department = p\_Department;

IF SQL%ROWCOUNT = 0 THEN

DBMS\_OUTPUT.PUT\_LINE('No employees found in department: ' || p\_Department);

ELSE

DBMS\_OUTPUT.PUT\_LINE(SQL%ROWCOUNT || ' employees updated in department: ' || p\_Department);

END IF;

COMMIT;

END;

/

BEGIN

UpdateEmployeeBonus('SALES', 10); -- Gives 10% bonus to SALES dept

UpdateEmployeeBonus('IT', 5); -- No effect, no such department

END;

/

**Output:**

****

**--Scenario 3: Transfer Funds Between Accounts (with Check)**

CREATE OR REPLACE PROCEDURE TransferFunds(

p\_FromAccID IN NUMBER,

p\_ToAccID IN NUMBER,

p\_Amount IN NUMBER

) IS

v\_Balance ACCOUNTS.Balance%TYPE;

BEGIN

-- Check balance of sender

SELECT Balance INTO v\_Balance FROM ACCOUNTS WHERE AccountID = p\_FromAccID;

IF v\_Balance < p\_Amount THEN

RAISE\_APPLICATION\_ERROR(-20010, 'Insufficient balance in source account.');

END IF;

-- Transfer funds

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

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

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Transfer successful from Account ' || p\_FromAccID ||

' to Account ' || p\_ToAccID);

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END;

/

BEGIN

TransferFunds(101, 102, 3000); -- Valid

TransferFunds(102, 103, 9000); -- Invalid (insufficient funds)

END;

/

**Output:**

****

**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.

**Code:**

**--Scenario 1: Calculate Age of Customers**

CREATE OR REPLACE FUNCTION CalculateAge(

p\_DOB IN DATE

) RETURN NUMBER IS

v\_Age NUMBER;

BEGIN

v\_Age := FLOOR(MONTHS\_BETWEEN(SYSDATE, p\_DOB) / 12);

RETURN v\_Age;

END;

/

DECLARE

age NUMBER;

BEGIN

age := CalculateAge(TO\_DATE('1990-06-25', 'YYYY-MM-DD'));

DBMS\_OUTPUT.PUT\_LINE('Age: ' || age);

END;

/

**Output:**

****

**--Scenario 2: Compute Monthly Installment for a Loan**

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(

p\_LoanAmount IN NUMBER,

p\_AnnualRate IN NUMBER,

p\_Years IN NUMBER

) RETURN NUMBER IS

v\_EMI NUMBER;

r NUMBER; -- Monthly rate

n NUMBER; -- Total months

BEGIN

r := p\_AnnualRate / 12 / 100;

n := p\_Years \* 12;

v\_EMI := (p\_LoanAmount \* r \* POWER(1 + r, n)) / (POWER(1 + r, n) - 1);

RETURN ROUND(v\_EMI, 2);

END;

/

DECLARE

emi NUMBER;

BEGIN

emi := CalculateMonthlyInstallment(100000, 10, 5);

DBMS\_OUTPUT.PUT\_LINE('Monthly Installment: ' || emi);

END;

/

**Output:**

****

**--Scenario 3: Check if Account Has Sufficient Balance**

CREATE OR REPLACE FUNCTION HasSufficientBalance(

p\_AccID IN NUMBER,

p\_Amount IN NUMBER

) RETURN BOOLEAN IS

v\_Balance ACCOUNTS.Balance%TYPE;

BEGIN

SELECT Balance INTO v\_Balance FROM ACCOUNTS WHERE AccountID = p\_AccID;

RETURN v\_Balance >= p\_Amount;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN FALSE;

WHEN OTHERS THEN

RETURN FALSE;

END;

/

DECLARE

result BOOLEAN;

BEGIN

result := HasSufficientBalance(101, 2000);

IF result THEN

DBMS\_OUTPUT.PUT\_LINE('Sufficient balance.');

ELSE

DBMS\_OUTPUT.PUT\_LINE('Insufficient balance.');

END IF;

END;

/

**Output:**

****

**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.

**Code:**

CREATE TABLE CUSTOMER (

CustomerID NUMBER PRIMARY KEY,

Name VARCHAR2(100),

Age NUMBER,

Balance NUMBER(10,2),

LastModified DATE

);

CREATE TABLE TRANSACTIONS (

TransactionID NUMBER PRIMARY KEY,

AccountID NUMBER,

TransType VARCHAR2(20), -- 'DEPOSIT' or 'WITHDRAWAL'

Amount NUMBER(10,2),

TransDate DATE DEFAULT SYSDATE

);

CREATE TABLE AUDITLOG (

AuditID NUMBER GENERATED BY DEFAULT AS IDENTITY,

TransactionID NUMBER,

AccountID NUMBER,

Action VARCHAR2(50),

ActionTime TIMESTAMP DEFAULT SYSTIMESTAMP

);

INSERT INTO CUSTOMER VALUES (101, 'Alice', 30, 10000, SYSDATE);

INSERT INTO CUSTOMER VALUES (102, 'Bob', 40, 3000, SYSDATE);

COMMIT;

**--Scenario 1: Automatically Update LastModified Column**

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON CUSTOMERS

FOR EACH ROW

BEGIN

:NEW.LastModified := SYSDATE;

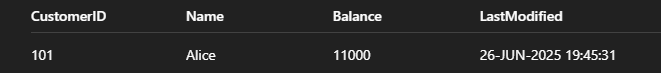
END;

/

UPDATE CUSTOMERS SET Balance = Balance + 500 WHERE CustomerID = 101;

SELECT CustomerID, Balance, LastModified FROM CUSTOMERS;

**Output:**

****

**--Scenario 2: Maintain Audit Log on Transactions**

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON TRANSACTIONS

FOR EACH ROW

BEGIN

INSERT INTO AUDITLOG(TransactionID, AccountID, Action)

VALUES (:NEW.TransactionID, :NEW.AccountID, 'Transaction Inserted');

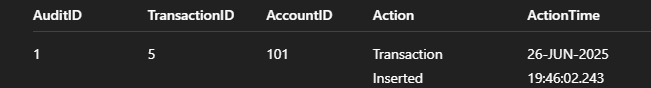
END;

/

INSERT INTO TRANSACTIONS VALUES (1, 101, 'DEPOSIT', 2000, SYSDATE);

SELECT \* FROM AUDITLOG;

**Output:**

****

**-- Scenario 3: Enforce Business Rules on Deposits & Withdrawals**

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON TRANSACTIONS

FOR EACH ROW

DECLARE

v\_Balance CUSTOMERS.Balance%TYPE;

BEGIN

-- Get customer balance

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

-- Rule 1: Withdrawal should not exceed balance

IF :NEW.TransType = 'WITHDRAWAL' AND :NEW.Amount > v\_Balance THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Withdrawal exceeds current balance.');

END IF;

-- Rule 2: Deposit amount must be positive

IF :NEW.TransType = 'DEPOSIT' AND :NEW.Amount <= 0 THEN

RAISE\_APPLICATION\_ERROR(-20002, 'Deposit amount must be greater than zero.');

END IF;

END;

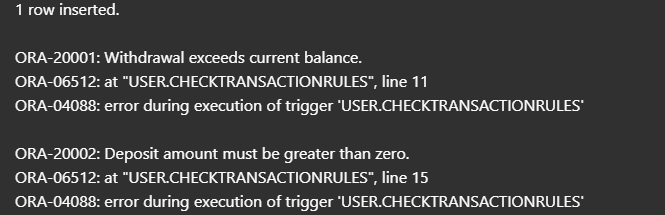
/

INSERT INTO TRANSACTIONS VALUES (2, 101, 'WITHDRAWAL', 1000, SYSDATE);

INSERT INTO TRANSACTIONS VALUES (3, 102, 'WITHDRAWAL', 5000, SYSDATE);

INSERT INTO TRANSACTIONS VALUES (4, 101, 'DEPOSIT', -100, SYSDATE);

**Output:**

****

**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.

**Code:**

CREATE TABLE ACCOUNTS (

AccountID NUMBER PRIMARY KEY,

HolderName VARCHAR2(100),

Balance NUMBER(10,2),

AccountType VARCHAR2(20)

);

CREATE TABLE TRANSACTIONS (

TransactionID NUMBER PRIMARY KEY,

AccountID NUMBER,

TransType VARCHAR2(20), -- DEPOSIT/WITHDRAWAL

Amount NUMBER(10,2),

TransDate DATE

);

CREATE TABLE LOANS (

LoanID NUMBER PRIMARY KEY,

CustomerID NUMBER,

LoanAmount NUMBER(10,2),

InterestRate NUMBER(5,2)

);

INSERT INTO ACCOUNTS VALUES (101, 'Alice', 5000, 'SAVINGS');

INSERT INTO ACCOUNTS VALUES (102, 'Bob', 3000, 'CURRENT');

INSERT INTO ACCOUNTS VALUES (103, 'Charlie', 7000, 'SAVINGS');

INSERT INTO TRANSACTIONS VALUES (1, 101, 'DEPOSIT', 1000, SYSDATE);

INSERT INTO TRANSACTIONS VALUES (2, 101, 'WITHDRAWAL', 200, ADD\_MONTHS(SYSDATE, -1));

INSERT INTO TRANSACTIONS VALUES (3, 102, 'DEPOSIT', 1500, SYSDATE);

INSERT INTO TRANSACTIONS VALUES (4, 103, 'DEPOSIT', 800, SYSDATE);

INSERT INTO LOANS VALUES (1, 101, 100000, 10.0);

INSERT INTO LOANS VALUES (2, 102, 50000, 11.5);

COMMIT;

**--Scenario 1: Generate Monthly Statements**

DECLARE

CURSOR cur\_month\_txns IS

SELECT AccountID, TransType, Amount, TransDate

FROM TRANSACTIONS

WHERE TO\_CHAR(TransDate, 'MMYYYY') = TO\_CHAR(SYSDATE, 'MMYYYY')

ORDER BY AccountID;

v\_AccountID TRANSACTIONS.AccountID%TYPE;

v\_TransType TRANSACTIONS.TransType%TYPE;

v\_Amount TRANSACTIONS.Amount%TYPE;

v\_TransDate TRANSACTIONS.TransDate%TYPE;

v\_CurrentAcct NUMBER := NULL;

BEGIN

DBMS\_OUTPUT.PUT\_LINE('Monthly Statements - ' || TO\_CHAR(SYSDATE, 'Month YYYY'));

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

OPEN cur\_month\_txns;

LOOP

FETCH cur\_month\_txns INTO v\_AccountID, v\_TransType, v\_Amount, v\_TransDate;

EXIT WHEN cur\_month\_txns%NOTFOUND;

IF v\_CurrentAcct IS NULL OR v\_CurrentAcct != v\_AccountID THEN

DBMS\_OUTPUT.PUT\_LINE(CHR(10) || 'Account ID: ' || v\_AccountID);

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

v\_CurrentAcct := v\_AccountID;

END IF;

DBMS\_OUTPUT.PUT\_LINE(TO\_CHAR(v\_TransDate, 'DD-MON') || ' - ' || v\_TransType || ' - ₹' || v\_Amount);

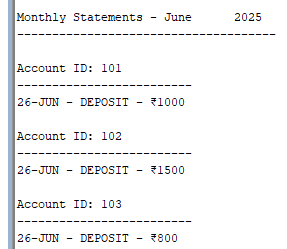
END LOOP;

CLOSE cur\_month\_txns;

END;

/

**Output:**

****

**-- Scenario 2: Apply Annual Fee to All Accounts**

DECLARE

CURSOR cur\_accounts IS

SELECT AccountID, Balance FROM ACCOUNTS;

v\_AccID ACCOUNTS.AccountID%TYPE;

v\_Balance ACCOUNTS.Balance%TYPE;

v\_Fee CONSTANT NUMBER := 500;

BEGIN

OPEN cur\_accounts;

LOOP

FETCH cur\_accounts INTO v\_AccID, v\_Balance;

EXIT WHEN cur\_accounts%NOTFOUND;

IF v\_Balance >= v\_Fee THEN

UPDATE ACCOUNTS

SET Balance = Balance - v\_Fee

WHERE AccountID = v\_AccID;

DBMS\_OUTPUT.PUT\_LINE('Annual fee of ₹' || v\_Fee || ' deducted from Account ' || v\_AccID);

ELSE

DBMS\_OUTPUT.PUT\_LINE('Account ' || v\_AccID || ' has insufficient balance to apply fee.');

END IF;

END LOOP;

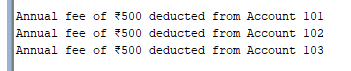
CLOSE cur\_accounts;

COMMIT;

END;

/

**Output:**

****

**--Scenario 3: Update Interest Rate for All Loans**

DECLARE

CURSOR cur\_loans IS

SELECT LoanID, InterestRate FROM LOANS;

v\_LoanID LOANS.LoanID%TYPE;

v\_Rate LOANS.InterestRate%TYPE;

v\_NewRate LOANS.InterestRate%TYPE;

BEGIN

OPEN cur\_loans;

LOOP

FETCH cur\_loans INTO v\_LoanID, v\_Rate;

EXIT WHEN cur\_loans%NOTFOUND;

IF v\_Rate > 10 THEN

v\_NewRate := v\_Rate - 1;

ELSE

v\_NewRate := v\_Rate + 0.5;

END IF;

UPDATE LOANS

SET InterestRate = v\_NewRate

WHERE LoanID = v\_LoanID;

DBMS\_OUTPUT.PUT\_LINE('Loan ID ' || v\_LoanID || ' interest updated from ' || v\_Rate || '% to ' || v\_NewRate || '%');

END LOOP;

CLOSE cur\_loans;

COMMIT;

END;

/

**Output:**

****

**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.

**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'));*

**Code:**

CREATE TABLE Customers (

CustomerID NUMBER PRIMARY KEY,

Name VARCHAR2(100),

DOB DATE,

Balance NUMBER,

LastModified DATE

);

CREATE TABLE Employees (

EmployeeID NUMBER PRIMARY KEY,

Name VARCHAR2(100),

Position VARCHAR2(50),

Salary NUMBER,

Department VARCHAR2(50),

HireDate DATE

);

CREATE TABLE Accounts (

AccountID NUMBER PRIMARY KEY,

CustomerID NUMBER,

AccountType VARCHAR2(20),

Balance NUMBER,

LastModified DATE,

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

**--CustomerManagement Package**

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 (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 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;

/

**--EmployeeManagement Package**

CREATE OR REPLACE PACKAGE EmployeeManagement AS

PROCEDURE HireEmployee(p\_id NUMBER, p\_name VARCHAR2, p\_position VARCHAR2, p\_salary NUMBER, p\_dept VARCHAR2);

PROCEDURE UpdateEmployee(p\_id NUMBER, p\_position VARCHAR2, p\_salary NUMBER);

FUNCTION GetAnnualSalary(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\_dept VARCHAR2) IS

BEGIN

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

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

END;

PROCEDURE UpdateEmployee(p\_id NUMBER, p\_position VARCHAR2, p\_salary NUMBER) IS

BEGIN

UPDATE Employees

SET Position = p\_position,

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;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN NULL;

END;

END EmployeeManagement;

/

**--AccountOperations Package**

CREATE OR REPLACE PACKAGE AccountOperations AS

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

PROCEDURE CloseAccount(p\_acc\_id NUMBER);

FUNCTION GetTotalCustomerBalance(p\_cust\_id NUMBER) RETURN NUMBER;

END AccountOperations;

/

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

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

BEGIN

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

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

END;

PROCEDURE CloseAccount(p\_acc\_id NUMBER) IS

BEGIN

DELETE FROM Accounts WHERE AccountID = p\_acc\_id;

END;

FUNCTION GetTotalCustomerBalance(p\_cust\_id NUMBER) RETURN NUMBER IS

v\_total NUMBER;

BEGIN

SELECT SUM(Balance) INTO v\_total FROM Accounts WHERE CustomerID = p\_cust\_id;

RETURN NVL(v\_total, 0);

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN 0;

END;

END AccountOperations;

/

BEGIN

**-- Customer operations**

CustomerManagement.AddCustomer(3, 'Robert Downey', TO\_DATE('1970-04-04', 'YYYY-MM-DD'), 2000);

CustomerManagement.UpdateCustomer(3, 'Robert D. Jr.', 2500);

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

**-- Employee operations**

EmployeeManagement.HireEmployee(201, 'Emma Watson', 'Manager', 60000, 'HR');

EmployeeManagement.UpdateEmployee(201, 'Senior Manager', 75000);

DBMS\_OUTPUT.PUT\_LINE('Employee 201 Annual Salary: ₹' || EmployeeManagement.GetAnnualSalary(201));

**-- Account operations**

AccountOperations.OpenAccount(3, 3, 'Savings', 1500);

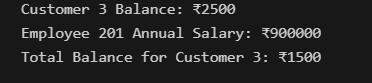
DBMS\_OUTPUT.PUT\_LINE('Total Balance for Customer 3: ₹' || AccountOperations.GetTotalCustomerBalance(3));

AccountOperations.CloseAccount(3);

END;

/

**Output:**

****