**WEEK-2 (PLSQL)**

**Exercise 1: Control Structures**

**Scenario 1:** Apply a discount to loan interest rates for customers above 60 years old

DECLARE

CURSOR c\_customers IS

SELECT customer\_id, loan\_interest\_rate

FROM customers

WHERE age > 60;

v\_customer\_id customers.customer\_id%TYPE;

v\_loan\_interest\_rate customers.loan\_interest\_rate%TYPE;

BEGIN

OPEN c\_customers;

LOOP

FETCH c\_customers INTO v\_customer\_id, v\_loan\_interest\_rate;

EXIT WHEN c\_customers%NOTFOUND;

-- Apply a 1% discount

UPDATE customers

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

WHERE customer\_id = v\_customer\_id;

END LOOP;

CLOSE c\_customers;

COMMIT;

END;

**Scenario 2:** Set IsVIP flag for customers with a balance over $10,000

DECLARE

CURSOR c\_customers IS

SELECT customer\_id, balance

FROM customers

WHERE balance > 10000;

v\_customer\_id customers.customer\_id%TYPE;

BEGIN

OPEN c\_customers;

LOOP

FETCH c\_customers INTO v\_customer\_id;

EXIT WHEN c\_customers%NOTFOUND;

-- Set IsVIP flag to TRUE

UPDATE customers

SET IsVIP = TRUE

WHERE customer\_id = v\_customer\_id;

END LOOP;

CLOSE c\_customers;

COMMIT;

END;

**Scenario 3:** Send reminders for loans due within the next 30 days

DECLARE

CURSOR c\_loans IS

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

FROM loans

WHERE due\_date BETWEEN SYSDATE AND SYSDATE + 30;

v\_loan\_id loans.loan\_id%TYPE;

v\_customer\_id loans.customer\_id%TYPE;

v\_due\_date loans.due\_date%TYPE;

BEGIN

OPEN c\_loans;

LOOP

FETCH c\_loans INTO v\_loan\_id, v\_customer\_id, v\_due\_date;

EXIT WHEN c\_loans%NOTFOUND;

-- Print reminder message

DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan ' || v\_loan\_id || ' for customer ' || v\_customer\_id || ' is due on ' || v\_due\_date);

END LOOP;

CLOSE c\_loans;

END;

**Exercise 2: Error Handling**

**Scenario 1:** SafeTransferFunds procedure

CREATE OR REPLACE PROCEDURE SafeTransferFunds(p\_from\_account IN NUMBER, p\_to\_account IN NUMBER, p\_amount IN NUMBER) IS

insufficient\_funds EXCEPTION;

BEGIN

-- Start the transaction

SAVEPOINT start\_transaction;

-- Check if the from\_account has sufficient funds

DECLARE

v\_balance NUMBER;

BEGIN

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

IF v\_balance < p\_amount THEN

RAISE insufficient\_funds;

END IF;

END;

-- Deduct from from\_account

UPDATE accounts

SET balance = balance - p\_amount

WHERE account\_id = p\_from\_account;

-- Add to to\_account

UPDATE accounts

SET balance = balance + p\_amount

WHERE account\_id = p\_to\_account;

-- Commit the transaction

COMMIT;

EXCEPTION

WHEN insufficient\_funds THEN

ROLLBACK TO start\_transaction;

DBMS\_OUTPUT.PUT\_LINE('Error: Insufficient funds in the source account.');

WHEN OTHERS THEN

ROLLBACK TO start\_transaction;

DBMS\_OUTPUT.PUT\_LINE('Error: An unexpected error occurred during the transfer.');

END SafeTransferFunds;

**Scenario 2:** UpdateSalary procedure

CREATE OR REPLACE PROCEDURE UpdateSalary(p\_employee\_id IN NUMBER, p\_percentage IN NUMBER) IS

employee\_not\_found EXCEPTION;

BEGIN

-- Attempt to update the employee's salary

UPDATE employees

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

WHERE employee\_id = p\_employee\_id;

IF SQL%NOTFOUND THEN

RAISE employee\_not\_found;

END IF;

COMMIT;

EXCEPTION

WHEN employee\_not\_found THEN

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

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('Error: An unexpected error occurred while updating the salary.');

END UpdateSalary;

**Scenario 3:**AddNewCustomer procedure

CREATE OR REPLACE PROCEDURE AddNewCustomer(p\_customer\_id IN NUMBER, p\_name IN VARCHAR2, p\_age IN NUMBER, p\_balance IN NUMBER) IS

customer\_exists EXCEPTION;

BEGIN

-- Attempt to insert the new customer

INSERT INTO customers (customer\_id, name, age, balance)

VALUES (p\_customer\_id, p\_name, p\_age, p\_balance);

COMMIT;

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

RAISE customer\_exists;

WHEN customer\_exists THEN

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

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('Error: An unexpected error occurred while adding the new customer.');

END AddNewCustomer;

These PL/SQL blocks and stored procedures handle the specified scenarios for control structures and error handling, ensuring robust and maintainable code for the library management system.

**Exercise 3:** Stored Procedures

**Scenario 1:** Process Monthly Interest for Savings Accounts

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

CURSOR c\_savings\_accounts IS

SELECT account\_id, balance

FROM accounts

WHERE account\_type = 'SAVINGS';

v\_account\_id accounts.account\_id%TYPE;

v\_balance accounts.balance%TYPE;

v\_interest\_rate CONSTANT NUMBER := 0.01;

BEGIN

OPEN c\_savings\_accounts;

LOOP

FETCH c\_savings\_accounts INTO v\_account\_id, v\_balance;

EXIT WHEN c\_savings\_accounts%NOTFOUND;

-- Calculate and update the balance with interest

UPDATE accounts

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

WHERE account\_id = v\_account\_id;

END LOOP;

CLOSE c\_savings\_accounts;

COMMIT;

END ProcessMonthlyInterest;

**Scenario 2:** Update Employee Bonus

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(p\_department\_id IN NUMBER, p\_bonus\_percentage IN NUMBER) IS

BEGIN

-- Update the salary of employees in the given department

UPDATE employees

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

WHERE department\_id = p\_department\_id;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('Error: An unexpected error occurred while updating the employee bonus.');

END UpdateEmployeeBonus;

**Scenario 3:** Transfer Funds Between Accounts

CREATE OR REPLACE PROCEDURE TransferFunds(p\_from\_account IN NUMBER, p\_to\_account IN NUMBER, p\_amount IN NUMBER) IS

insufficient\_funds EXCEPTION;

v\_balance NUMBER;

BEGIN

-- Check if the from\_account has sufficient funds

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

IF v\_balance < p\_amount THEN

RAISE insufficient\_funds;

END IF;

-- Deduct from from\_account

UPDATE accounts

SET balance = balance - p\_amount

WHERE account\_id = p\_from\_account;

-- Add to to\_account

UPDATE accounts

SET balance = balance + p\_amount

WHERE account\_id = p\_to\_account;

COMMIT;

EXCEPTION

WHEN insufficient\_funds THEN

DBMS\_OUTPUT.PUT\_LINE('Error: Insufficient funds in the source account.');

ROLLBACK;

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('Error: An unexpected error occurred during the transfer.');

ROLLBACK;

END TransferFunds;

**Exercise 4:** Functions

**Scenario 1:** Calculate Age of Customers

CREATE OR REPLACE FUNCTION CalculateAge(p\_dob DATE) RETURN NUMBER IS

v\_age NUMBER;

BEGIN

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

RETURN v\_age;

END CalculateAge;

**Scenario 2:** Calculate Monthly Installment for a Loan

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(p\_loan\_amount NUMBER, p\_interest\_rate NUMBER, p\_duration\_years NUMBER) RETURN NUMBER IS

v\_monthly\_rate NUMBER;

v\_number\_of\_payments NUMBER;

v\_monthly\_installment NUMBER;

BEGIN

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

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

v\_monthly\_installment := p\_loan\_amount \* v\_monthly\_rate / (1 - POWER(1 + v\_monthly\_rate, -v\_number\_of\_payments));

RETURN v\_monthly\_installment;

END CalculateMonthlyInstallment;

**Scenario 3:** Check Sufficient Balance Before Transaction

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;

IF v\_balance >= p\_amount THEN

RETURN TRUE;

ELSE

RETURN FALSE;

END IF;

END HasSufficientBalance;

These stored procedures and functions implement the required scenarios, ensuring robust and efficient handling of banking operations in the library management system.

**Exercise 5:** Triggers

**Scenario 1:** Automatically update the last modified date when a customer's record is updated

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON customers

FOR EACH ROW

BEGIN

:NEW.LastModified := SYSDATE;

END UpdateCustomerLastModified;

**Scenario 2:** Maintain an audit log for all transactions

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog (transaction\_id, account\_id, transaction\_date, amount, transaction\_type)

VALUES (:NEW.transaction\_id, :NEW.account\_id, :NEW.transaction\_date, :NEW.amount, :NEW.transaction\_type);

END LogTransaction;

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

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON transactions

FOR EACH ROW

BEGIN

IF :NEW.transaction\_type = 'WITHDRAWAL' THEN

DECLARE

v\_balance NUMBER;

BEGIN

SELECT balance INTO v\_balance FROM accounts WHERE account\_id = :NEW.account\_id FOR UPDATE;

IF v\_balance < :NEW.amount THEN

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

END IF;

END;

ELSIF :NEW.transaction\_type = 'DEPOSIT' THEN

IF :NEW.amount <= 0 THEN

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

END IF;

END IF;

END CheckTransactionRules;

**Exercise 6:** Cursors

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

DECLARE

CURSOR c\_transactions IS

SELECT customer\_id, transaction\_date, amount, transaction\_type

FROM transactions

WHERE transaction\_date BETWEEN TRUNC(SYSDATE, 'MM') AND LAST\_DAY(SYSDATE);

v\_customer\_id transactions.customer\_id%TYPE;

v\_transaction\_date transactions.transaction\_date%TYPE;

v\_amount transactions.amount%TYPE;

v\_transaction\_type transactions.transaction\_type%TYPE;

BEGIN

OPEN c\_transactions;

LOOP

FETCH c\_transactions INTO v\_customer\_id, v\_transaction\_date, v\_amount, v\_transaction\_type;

EXIT WHEN c\_transactions%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Customer ID: ' || v\_customer\_id || ', Date: ' || v\_transaction\_date || ', Amount: ' || v\_amount || ', Type: ' || v\_transaction\_type);

END LOOP;

CLOSE c\_transactions;

END;

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

DECLARE

CURSOR c\_accounts IS

SELECT account\_id, balance

FROM accounts;

v\_account\_id accounts.account\_id%TYPE;

v\_balance accounts.balance%TYPE;

v\_annual\_fee CONSTANT NUMBER := 100;

BEGIN

OPEN c\_accounts;

LOOP

FETCH c\_accounts INTO v\_account\_id, v\_balance;

EXIT WHEN c\_accounts%NOTFOUND;

UPDATE accounts

SET balance = balance - v\_annual\_fee

WHERE account\_id = v\_account\_id;

END LOOP;

CLOSE c\_accounts;

COMMIT;

END;

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

DECLARE

CURSOR c\_loans IS

SELECT loan\_id, interest\_rate

FROM loans;

v\_loan\_id loans.loan\_id%TYPE;

v\_interest\_rate loans.interest\_rate%TYPE;

v\_new\_interest\_rate CONSTANT NUMBER := 5; -- Example new interest rate

BEGIN

OPEN c\_loans;

LOOP

FETCH c\_loans INTO v\_loan\_id, v\_interest\_rate;

EXIT WHEN c\_loans%NOTFOUND;

UPDATE loans

SET interest\_rate = v\_new\_interest\_rate

WHERE loan\_id = v\_loan\_id;

END LOOP;

CLOSE c\_loans;

COMMIT;

END;

These triggers and PL/SQL blocks ensure that the bank's operations adhere to the defined business rules and requirements while maintaining data integrity and consistency.

**Exercise 7:** Packages

**Scenario 1:** Customer Management Package

CREATE OR REPLACE PACKAGE CustomerManagement AS

PROCEDURE AddNewCustomer(p\_customer\_id IN NUMBER, p\_name IN VARCHAR2, p\_dob IN DATE, p\_balance IN NUMBER);

PROCEDURE UpdateCustomerDetails(p\_customer\_id IN NUMBER, p\_name IN VARCHAR2, p\_dob IN DATE, p\_balance IN NUMBER);

FUNCTION GetCustomerBalance(p\_customer\_id IN NUMBER) RETURN NUMBER;

END CustomerManagement;

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

PROCEDURE AddNewCustomer(p\_customer\_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\_customer\_id, p\_name, p\_dob, p\_balance, SYSDATE);

COMMIT;

END AddNewCustomer;

PROCEDURE UpdateCustomerDetails(p\_customer\_id IN NUMBER, p\_name IN VARCHAR2, p\_dob IN DATE, p\_balance IN NUMBER) IS

BEGIN

UPDATE Customers

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

WHERE CustomerID = p\_customer\_id;

COMMIT;

END UpdateCustomerDetails;

FUNCTION GetCustomerBalance(p\_customer\_id IN NUMBER) RETURN NUMBER IS

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance

FROM Customers

WHERE CustomerID = p\_customer\_id;

RETURN v\_balance;

END GetCustomerBalance;

END CustomerManagement;

**Scenario 2:** Employee Management Package

CREATE OR REPLACE PACKAGE EmployeeManagement AS

PROCEDURE HireNewEmployee(p\_employee\_id IN NUMBER, p\_name IN VARCHAR2, p\_position IN VARCHAR2, p\_salary IN NUMBER, p\_department IN VARCHAR2, p\_hire\_date IN DATE);

PROCEDURE UpdateEmployeeDetails(p\_employee\_id IN NUMBER, p\_name IN VARCHAR2, p\_position IN VARCHAR2, p\_salary IN NUMBER, p\_department IN VARCHAR2);

FUNCTION CalculateAnnualSalary(p\_employee\_id IN NUMBER) RETURN NUMBER;

END EmployeeManagement;

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

PROCEDURE HireNewEmployee(p\_employee\_id IN NUMBER, p\_name IN VARCHAR2, p\_position IN VARCHAR2, p\_salary IN NUMBER, p\_department IN VARCHAR2, p\_hire\_date IN DATE) 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 NUMBER, p\_name IN VARCHAR2, p\_position IN VARCHAR2, p\_salary IN NUMBER, p\_department IN VARCHAR2) 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 NUMBER) RETURN NUMBER IS

v\_salary NUMBER;

BEGIN

SELECT Salary INTO v\_salary

FROM Employees

WHERE EmployeeID = p\_employee\_id;

RETURN v\_salary \* 12;

END CalculateAnnualSalary;

END EmployeeManagement;

**Scenario 3:** Account Operations Package

CREATE OR REPLACE PACKAGE AccountOperations AS

PROCEDURE OpenNewAccount(p\_account\_id IN NUMBER, p\_customer\_id IN NUMBER, p\_account\_type IN VARCHAR2, p\_balance IN NUMBER);

PROCEDURE CloseAccount(p\_account\_id IN NUMBER);

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

END AccountOperations;

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

PROCEDURE OpenNewAccount(p\_account\_id IN NUMBER, p\_customer\_id IN NUMBER, p\_account\_type IN VARCHAR2, p\_balance IN NUMBER) 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 NUMBER) IS

BEGIN

DELETE FROM Accounts

WHERE AccountID = p\_account\_id;

COMMIT;

END CloseAccount;

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

v\_total\_balance NUMBER;

BEGIN

SELECT SUM(Balance) INTO v\_total\_balance

FROM Accounts

WHERE CustomerID = p\_customer\_id;

RETURN v\_total\_balance;

END GetTotalBalance;

END AccountOperations;

These packages provide a structured way to manage customer, employee, and account-related operations, encapsulating the functionality into reusable components.