**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.

Ans)

BEGIN

FOR rec IN (

SELECT CustomerID, DOB

FROM Customers

) LOOP

DECLARE

v\_age NUMBER;

BEGIN

v\_age := FLOOR(MONTHS\_BETWEEN(SYSDATE, rec.DOB) / 12);

IF v\_age > 60 THEN

UPDATE Loans

SET InterestRate = InterestRate - 1

WHERE CustomerID = rec.CustomerID;

END IF;

END;

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.

Ans)

ALTER TABLE Customers ADD (IsVIP CHAR(1) DEFAULT 'N');

BEGIN

FOR rec IN (

SELECT CustomerID, Balance

FROM Customers

) LOOP

IF rec.Balance > 10000 THEN

UPDATE Customers

SET IsVIP = 'Y'

WHERE CustomerID = rec.CustomerID;

ELSE

UPDATE Customers

SET IsVIP = 'N'

WHERE CustomerID = rec.CustomerID;

END IF;

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.

Ans)

BEGIN

FOR rec IN (

SELECT l.LoanID, 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

DBMS\_OUTPUT.PUT\_LINE(

'Reminder: Customer ' || rec.Name || ' has a loan with ID ' ||

rec.LoanID || ' that is due on ' || TO\_CHAR(rec.EndDate, 'YYYY-MM-DD') ||

'. Please remind the customer.'

);

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.

Ans)

CREATE OR REPLACE PROCEDURE SafeTransferFunds (

p\_from\_account\_id IN NUMBER,

p\_to\_account\_id IN NUMBER,

p\_amount IN NUMBER

) IS

v\_from\_balance NUMBER;

v\_to\_balance NUMBER;

BEGIN

IF p\_amount <= 0 THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Transfer amount must be positive.');

END IF;

SELECT Balance INTO v\_from\_balance

FROM Accounts

WHERE AccountID = p\_from\_account\_id;

SELECT Balance INTO v\_to\_balance

FROM Accounts

WHERE AccountID = p\_to\_account\_id;

IF v\_from\_balance < p\_amount THEN

RAISE\_APPLICATION\_ERROR(-20002, 'Insufficient funds in the source account.');

END IF;

BEGIN

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 OTHERS THEN

ROLLBACK;

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

RAISE;

END;

END SafeTransferFunds;

/

**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.

Ans)

CREATE OR REPLACE PROCEDURE UpdateSalary (

p\_employee\_id IN NUMBER,

p\_percentage IN NUMBER

) IS

BEGIN

IF p\_percentage <= 0 THEN

RAISE\_APPLICATION\_ERROR(-20003, 'Percentage must be positive.');

END IF;

BEGIN

UPDATE Employees

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

WHERE EmployeeID = p\_employee\_id;

IF SQL%ROWCOUNT = 0 THEN

RAISE\_APPLICATION\_ERROR(-20004, 'Employee ID does not exist.');

END IF;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

RAISE; -

END;

END UpdateSalary;

/

**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.

Ans)

CREATE OR REPLACE PROCEDURE AddNewCustomer (

p\_customer\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_dob IN DATE,

p\_balance IN NUMBER

) IS

BEGIN

BEGIN

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

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

COMMIT;

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

ROLLBACK;

WHEN OTHERS THEN

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

ROLLBACK;

END;

END AddNewCustomer;

/

**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.

Ans)

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

BEGIN

BEGIN

UPDATE Accounts

SET Balance = Balance \* 1.01 -- Applying 1% interest

WHERE AccountType = 'Savings';

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error occurred while processing monthly interest: ' || SQLERRM);

RAISE;

END;

END ProcessMonthlyInterest;

/

**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.

Ans)

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (

p\_department IN VARCHAR2,

p\_bonus\_percentage IN NUMBER

) IS

BEGIN

IF p\_bonus\_percentage <= 0 THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Bonus percentage must be positive.');

END IF;

BEGIN

UPDATE Employees

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

WHERE Department = p\_department;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error occurred while updating employee bonuses: ' || SQLERRM);

RAISE; -

END;

END UpdateEmployeeBonus;

/

**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.

Ans)

CREATE OR REPLACE PROCEDURE TransferFunds (

p\_from\_account\_id IN NUMBER,

p\_to\_account\_id IN NUMBER,

p\_amount IN NUMBER

) IS

v\_from\_balance NUMBER;

v\_to\_balance NUMBER;

BEGIN

IF p\_amount <= 0 THEN

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

END IF;

SELECT Balance INTO v\_from\_balance

FROM Accounts

WHERE AccountID = p\_from\_account\_id;

SELECT Balance INTO v\_to\_balance

FROM Accounts

WHERE AccountID = p\_to\_account\_id;

IF v\_from\_balance < p\_amount THEN

RAISE\_APPLICATION\_ERROR(-20003, 'Insufficient funds in the source account.');

END IF;

BEGIN

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 OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error occurred while transferring funds: ' || SQLERRM);

RAISE; -

END;

END TransferFunds;

/

**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.

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;

/

Ans)

**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.

Ans)

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment (

p\_loan\_amount NUMBER,

p\_interest\_rate NUMBER,

p\_loan\_duration\_years NUMBER

) RETURN NUMBER IS

v\_monthly\_installment NUMBER;

v\_monthly\_rate NUMBER;

v\_num\_payments NUMBER;

BEGIN

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

v\_num\_payments := p\_loan\_duration\_years \* 12;

IF v\_monthly\_rate > 0 THEN

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

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

ELSE

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

END IF;

RETURN v\_monthly\_installment;

END CalculateMonthlyInstallment;

/

**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.

Ans)

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 AccountID = p\_account\_id;

IF v\_balance >= p\_amount THEN

RETURN TRUE;

ELSE

RETURN FALSE;

END IF;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN FALSE;

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('Error occurred while checking balance: ' || SQLERRM);

RETURN FALSE;

END HasSufficientBalance;

/

**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.

Ans)

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON Customers

FOR EACH ROW

BEGIN

:NEW.LastModified := SYSDATE;

END;

/

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

* + **Question:** Write a trigger **LogTransaction** that inserts a record into an AuditLog table whenever a transaction is inserted into the Transactions table.

Ans)

CREATE TABLE AuditLog (

LogID NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,

TransactionID NUMBER,

LogDate DATE DEFAULT SYSDATE,

LogMessage VARCHAR2(100)

);

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog (TransactionID, LogMessage)

VALUES (:NEW.TransactionID, 'Transaction inserted on ' || TO\_CHAR(SYSDATE, 'YYYY-MM-DD HH24:MI:SS'));

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.

Ans)

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

DECLARE

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance

FROM Accounts

WHERE AccountID = :NEW.AccountID;

IF :NEW.TransactionType = 'Withdrawal' THEN

IF (:NEW.Amount > v\_balance) THEN

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

END IF;

ELSIF :NEW.TransactionType = 'Deposit' THEN

IF (:NEW.Amount <= 0) THEN

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

END IF;

ELSE

NULL;

END IF;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

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

WHEN OTHERS THEN

RAISE\_APPLICATION\_ERROR(-20004, 'Error occurred: ' || SQLERRM);

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.

Ans)

SET SERVEROUTPUT ON;

DECLARE

CURSOR monthly\_transactions\_cur IS

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

FROM Customers c

JOIN Accounts a ON c.CustomerID = a.CustomerID

JOIN Transactions t ON a.AccountID = t.AccountID

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

v\_statement VARCHAR2(4000);

BEGIN

FOR rec IN monthly\_transactions\_cur LOOP

v\_statement := 'Customer: ' || rec.Name || ', Transaction ID: ' || rec.TransactionID ||

', Date: ' || TO\_CHAR(rec.TransactionDate, 'YYYY-MM-DD') ||

', Amount: ' || rec.Amount || ', Type: ' || rec.TransactionType;

DBMS\_OUTPUT.PUT\_LINE(v\_statement);

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.

Ans)

DECLARE

CURSOR accounts\_cur IS

SELECT AccountID, Balance

FROM Accounts;

v\_annual\_fee NUMBER := 50;

BEGIN

FOR rec IN accounts\_cur LOOP

UPDATE Accounts

SET Balance = Balance - v\_annual\_fee

WHERE AccountID = rec.AccountID;

DBMS\_OUTPUT.PUT\_LINE('Annual fee deducted from Account ID ' || rec.AccountID);

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.

Ans)

DECLARE

CURSOR loans\_cur IS

SELECT LoanID, InterestRate

FROM Loans;

v\_new\_policy\_interest\_rate NUMBER := 4.5;

BEGIN

FOR rec IN loans\_cur LOOP

UPDATE Loans

SET InterestRate = v\_new\_policy\_interest\_rate

WHERE LoanID = rec.LoanID;

DBMS\_OUTPUT.PUT\_LINE('Updated interest rate for Loan ID ' || rec.LoanID);

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.

Ans)

CREATE OR REPLACE PACKAGE CustomerManagement IS

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

);

FUNCTION GetCustomerBalance (

p\_customer\_id IN NUMBER

) RETURN NUMBER;

END CustomerManagement;

/

CREATE OR REPLACE PACKAGE BODY CustomerManagement IS

-- Procedure to add a new customer

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

) IS

BEGIN

UPDATE Customers

SET Name = p\_name,

DOB = p\_dob,

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;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN NULL; -- Handle case where customer does not exist

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.

Ans)

CREATE OR REPLACE PACKAGE EmployeeManagement IS

PROCEDURE HireEmployee (

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 IS

PROCEDURE HireEmployee (

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 HireEmployee;

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; -- Assuming salary is monthly, calculate annual salary

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN NULL; -- Handle case where employee does not exist

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.

Ans)

CREATE OR REPLACE PACKAGE AccountOperations AS

PROCEDURE OpenAccount (p\_CustomerID IN NUMBER, p\_AccountType IN VARCHAR2, p\_InitialBalance IN NUMBER);

PROCEDURE CloseAccount (p\_AccountID IN NUMBER);

FUNCTION GetTotalBalance (p\_CustomerID IN NUMBER) RETURN NUMBER;

END AccountOperations;

/

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

PROCEDURE OpenAccount (p\_CustomerID IN NUMBER, p\_AccountType IN VARCHAR2, p\_InitialBalance IN NUMBER) IS

BEGIN

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

VALUES (Accounts\_SEQ.NEXTVAL, p\_CustomerID, p\_AccountType, p\_InitialBalance, SYSDATE);

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END OpenAccount;

PROCEDURE CloseAccount (p\_AccountID IN NUMBER) IS

BEGIN

DELETE FROM Accounts WHERE AccountID = p\_AccountID;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END CloseAccount;

FUNCTION GetTotalBalance (p\_CustomerID IN NUMBER) RETURN NUMBER IS

v\_TotalBalance NUMBER;

BEGIN

SELECT SUM(Balance) INTO v\_TotalBalance

FROM Accounts

WHERE CustomerID = p\_CustomerID;

RETURN NVL(v\_TotalBalance, 0);

EXCEPTION

WHEN OTHERS THEN

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

RETURN 0;

END GetTotalBalance;

END AccountOperations;

/