**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.

**Scenario 1:**

DECLARE

v\_customer\_id NUMBER;

v\_dob DATE;

v\_age NUMBER;

BEGIN

FOR rec IN (SELECT CustomerID, DOB FROM Customers) LOOP

v\_customer\_id := rec.CustomerID;

v\_dob := rec.DOB;

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

IF v\_age > 60 THEN

UPDATE Loans

SET InterestRate = InterestRate - 1

WHERE CustomerID = v\_customer\_id;

END IF;

END LOOP;

END;

**Scenario 2:**

BEGIN

FOR rec IN (SELECT CustomerID, Balance FROM Customers) LOOP

IF rec.Balance > 10000 THEN

UPDATE Customers

SET IsVIP = 'TRUE'

WHERE CustomerID = rec.CustomerID;

END IF;

END LOOP;

END;

**Scenario 3:**

BEGIN

FOR rec IN (SELECT CustomerID, LoanID, EndDate FROM Loans WHERE EndDate <= SYSDATE + 30) LOOP

DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan ' || rec.LoanID || ' for customer ' || rec.CustomerID || ' is due within 30 days.');

  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.

**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.

**Scenario-1:**

CREATE PROCEDURE SafeTransferFunds (

IN p\_from\_account\_id INT,

IN p\_to\_account\_id INT,

IN p\_amount DECIMAL(10,2)

)

BEGIN

DECLARE insufficient\_funds EXCEPTION FOR SQLSTATE '45000';

DECLARE CONTINUE HANDLER FOR insufficient\_funds

BEGIN

ROLLBACK;

INSERT INTO ErrorLog (error\_message, error\_time)

VALUES ('Insufficient funds for transfer', NOW());

END;

START TRANSACTION;

DECLARE v\_balance DECIMAL(10,2);

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

IF v\_balance < p\_amount THEN

SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = 'Insufficient funds';

ELSE

UPDATE Accounts SET balance = balance - p\_amount WHERE account\_id = p\_from\_account\_id;

UPDATE Accounts SET balance = balance + p\_amount WHERE account\_id = p\_to\_account\_id;

END IF;

COMMIT;

END //

**Scenario-2:**

DELIMITER //

CREATE PROCEDURE UpdateSalary (

IN p\_employee\_id INT,

IN p\_percentage DECIMAL(5,2)

)

BEGIN

DECLARE employee\_not\_found EXCEPTION FOR SQLSTATE '45000';

DECLARE CONTINUE HANDLER FOR employee\_not\_found

BEGIN

INSERT INTO ErrorLog (error\_message, error\_time)

VALUES (CONCAT('Employee ID ', p\_employee\_id, ' not found'), NOW());

END;

DECLARE v\_current\_salary DECIMAL(10,2);

SELECT salary INTO v\_current\_salary FROM Employees WHERE employee\_id = p\_employee\_id;

IF v\_current\_salary IS NULL THEN

SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = 'Employee not found';

ELSE

UPDATE Employees

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

WHERE employee\_id = p\_employee\_id;

END IF;

END //

**Scenario-3:**

DELIMITER //

CREATE PROCEDURE AddNewCustomer (

IN p\_customer\_id INT,

IN p\_customer\_name VARCHAR(255),

IN p\_customer\_email VARCHAR(255)

)

BEGIN

DECLARE customer\_exists EXCEPTION FOR SQLSTATE '45000';

DECLARE CONTINUE HANDLER FOR customer\_exists

BEGIN

INSERT INTO ErrorLog (error\_message, error\_time)

VALUES (CONCAT('Customer ID ', p\_customer\_id, ' already exists'), NOW());

END;

DECLARE v\_existing\_customer\_id INT;

SELECT customer\_id INTO v\_existing\_customer\_id FROM Customers WHERE customer\_id = p\_customer\_id;

IF v\_existing\_customer\_id IS NOT NULL THEN

SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = 'Customer already exists';

ELSE

INSERT INTO Customers (customer\_id, customer\_name, customer\_email)

VALUES (p\_customer\_id, p\_customer\_name, p\_customer\_email);

END IF;

END //

DELIMITER ;

**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.

**Scenario 1: Process Monthly Interest**

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest AS

BEGIN

UPDATE Accounts

SET Balance = Balance \* 1.01,

LastModified = SYSDATE

WHERE AccountType = 'Savings';

END;

**Scenario 2: Update Employee Bonus**

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (

p\_Department VARCHAR2,

p\_BonusPercentage NUMBER

) AS

BEGIN

UPDATE Employees

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

WHERE Department = p\_Department;

END;

**Scenario 3: Transfer Funds**

CREATE OR REPLACE PROCEDURE TransferFunds (

p\_FromAccountID NUMBER,

p\_ToAccountID NUMBER,

p\_Amount NUMBER

) AS

v\_FromAccountBalance NUMBER;

BEGIN

SELECT Balance INTO v\_FromAccountBalance

FROM Accounts

WHERE AccountID = p\_FromAccountID;

IF v\_FromAccountBalance >= p\_Amount THEN

-- Deduct the amount from the source account

UPDATE Accounts

SET Balance = Balance - p\_Amount,

LastModified = SYSDATE

WHERE AccountID = p\_FromAccountID;

UPDATE Accounts

SET Balance = Balance + p\_Amount,

LastModified = SYSDATE

WHERE AccountID = p\_ToAccountID;

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)

VALUES (TRANSACTIONS\_SEQ.NEXTVAL, p\_FromAccountID, SYSDATE, p\_Amount, 'Transfer Out');

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)

VALUES (TRANSACTIONS\_SEQ.NEXTVAL, p\_ToAccountID, SYSDATE, p\_Amount, 'Transfer In');

ELSE

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

END IF;

END;

**Exercise 4: Functions**

**Scenario 1:** Calculate the age of customers for eligibility checks.

* + **Question:** Write a function CalculateAge that takes a customer's date of birth as input and returns their age in years.

**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.

**Scenario 1: CalculateAge**

CREATE DEFINER=`root`@`localhost` FUNCTION `CalculateAge`(dob DATE) RETURNS int

DETERMINISTIC

BEGIN

DECLARE age INT;

SET age = TIMESTAMPDIFF(YEAR,dob,CURDATE());

RETURN age;

RETURN 1;

END

**Scenario 2: CalculateMonthlyInstallment**

CREATE DEFINER=`root`@`localhost` FUNCTION `CalculateMonthlyInstallment`(loan\_amount DECIMAL(10,2),interest\_rate DECIMAL(5,2), loan\_duration INT) RETURNS decimal(10,2)

DETERMINISTIC

BEGIN

DECLARE monthly\_rate DECIMAL(10,7);

DECLARE number\_of\_payments INT;

DECLARE installment DECIMAL(10,2);

SET monthly\_rate = interest\_rate / 12 / 100;

SET number\_of\_payments = loan\_duration \* 12;

set installment = loan\_amount \*( monthly\_rate \* POWER(1+monthly\_rate, number\_of\_payments)) / (POWER(1+monthly\_rate, number\_of\_payments)-1);

RETURN installment;

END **Scenario 3: HasSufficientBalance**  
CREATE DEFINER=`root`@`localhost` FUNCTION `HasSufficientBalance`(account\_id INT,amount DECIMAL(10,2)) RETURNS int

DETERMINISTIC

BEGIN

DECLARE account\_balance DECIMAL(10,2);

select Balance INTO account\_balance

FROM Accounts Where ACCOUNTID = account\_id;

RETURN account\_balance >= amount;

END

**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.

**Scenario 1: UpdateCustomerLastModified**

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON Customers

FOR EACH ROW

BEGIN

:NEW.LastModified := SYSDATE;

END;

**Scenario 2: LogTransaction**

CREATE TABLE AuditLog (

LogID NUMBER PRIMARY KEY,

TransactionID NUMBER,

AccountID NUMBER,

TransactionDate DATE,

Amount NUMBER,

TransactionType VARCHAR2(10),

LogDate DATE

);

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog (LogID, TransactionID, AccountID, TransactionDate, Amount, TransactionType, LogDate)

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

END;

**Scenario 3: CheckTransactionRules**

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 balance.');

END IF;

END IF;

IF :NEW.TransactionType = 'Deposit' THEN

IF :NEW.Amount <= 0 THEN

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

END IF;

END IF;

END;

**Exercise 6: Cursors**

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

* + **Question:** Write a PL/SQL block using an explicit cursor **GenerateMonthlyStatements** that retrieves all transactions for the current month and prints a statement for each customer.

**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.

**Scenario 1:**

DECLARE

CURSOR GenerateMonthlyStatements IS

SELECT C.CustomerID, C.Name, 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)

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

v\_CustomerID Customers.CustomerID%TYPE;

v\_Name Customers.Name%TYPE;

v\_TransactionDate Transactions.TransactionDate%TYPE;

v\_Amount Transactions.Amount%TYPE;

v\_TransactionType Transactions.TransactionType%TYPE;

BEGIN

OPEN GenerateMonthlyStatements;

LOOP

FETCH GenerateMonthlyStatements INTO v\_CustomerID, v\_Name, v\_TransactionDate, v\_Amount, v\_TransactionType;

EXIT WHEN GenerateMonthlyStatements%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Customer: ' || v\_Name || ' | Date: ' || v\_TransactionDate || ' | Amount: ' || v\_Amount || ' | Type: ' || v\_TransactionType);

END LOOP;

CLOSE GenerateMonthlyStatements;

END;

/

**Scenario 2:**

DECLARE

CURSOR ApplyAnnualFee IS

SELECT AccountID, Balance

FROM Accounts;

v\_AccountID Accounts.AccountID%TYPE;

v\_Balance Accounts.Balance%TYPE;

v\_AnnualFee NUMBER := 50;

BEGIN

OPEN ApplyAnnualFee;

LOOP

FETCH ApplyAnnualFee INTO v\_AccountID, v\_Balance;

EXIT WHEN ApplyAnnualFee%NOTFOUND;

UPDATE Accounts

SET Balance = v\_Balance - v\_AnnualFee,

LastModified = SYSDATE

WHERE AccountID = v\_AccountID;

END LOOP;

CLOSE ApplyAnnualFee;

END;

**Scenario 3:**

DECLARE

CURSOR UpdateLoanInterestRates IS

SELECT LoanID, InterestRate

FROM Loans;

v\_LoanID Loans.LoanID%TYPE;

v\_InterestRate Loans.InterestRate%TYPE;

v\_NewInterestRate NUMBER;

BEGIN

OPEN UpdateLoanInterestRates;

LOOP

FETCH UpdateLoanInterestRates INTO v\_LoanID, v\_InterestRate;

EXIT WHEN UpdateLoanInterestRates%NOTFOUND;

v\_NewInterestRate := v\_InterestRate + 1;

UPDATE Loans

SET InterestRate = v\_NewInterestRate,

StartDate = SYSDATE

WHERE LoanID = v\_LoanID;

END LOOP;

CLOSE UpdateLoanInterestRates;

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.

**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.

**Scenario 1:**

CREATE OR REPLACE PACKAGE CustomerManagement AS

PROCEDURE AddCustomer(p\_CustomerID NUMBER, p\_Name VARCHAR2, p\_DOB DATE, p\_Balance NUMBER);

PROCEDURE UpdateCustomer(p\_CustomerID NUMBER, p\_Name VARCHAR2, p\_DOB DATE, p\_Balance NUMBER);

FUNCTION GetCustomerBalance(p\_CustomerID NUMBER) RETURN NUMBER;

END CustomerManagement;

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

PROCEDURE AddCustomer(p\_CustomerID NUMBER, p\_Name VARCHAR2, p\_DOB DATE, p\_Balance NUMBER) IS

BEGIN

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

VALUES (p\_CustomerID, p\_Name, p\_DOB, p\_Balance, SYSDATE);

END AddCustomer;

PROCEDURE UpdateCustomer(p\_CustomerID NUMBER, p\_Name VARCHAR2, p\_DOB DATE, p\_Balance NUMBER) IS

BEGIN

UPDATE Customers

SET Name = p\_Name, DOB = p\_DOB, Balance = p\_Balance, LastModified = SYSDATE

WHERE CustomerID = p\_CustomerID;

END UpdateCustomer;

FUNCTION GetCustomerBalance(p\_CustomerID NUMBER) RETURN NUMBER IS

v\_Balance NUMBER;

BEGIN

SELECT Balance INTO v\_Balance

FROM Customers

WHERE CustomerID = p\_CustomerID;

RETURN v\_Balance;

END GetCustomerBalance;

END CustomerManagement;

**Scenario 2:**

CREATE OR REPLACE PACKAGE EmployeeManagement AS

PROCEDURE HireEmployee(p\_EmployeeID NUMBER, p\_Name VARCHAR2, p\_Position VARCHAR2, p\_Salary NUMBER, p\_Department VARCHAR2, p\_HireDate DATE);

PROCEDURE UpdateEmployee(p\_EmployeeID NUMBER, p\_Name VARCHAR2, p\_Position VARCHAR2, p\_Salary NUMBER, p\_Department VARCHAR2);

FUNCTION CalculateAnnualSalary(p\_EmployeeID NUMBER) RETURN NUMBER;

END EmployeeManagement;

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

PROCEDURE HireEmployee(p\_EmployeeID NUMBER, p\_Name VARCHAR2, p\_Position VARCHAR2, p\_Salary NUMBER, p\_Department VARCHAR2, p\_HireDate DATE) IS

BEGIN

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

VALUES (p\_EmployeeID, p\_Name, p\_Position, p\_Salary, p\_Department, p\_HireDate);

END HireEmployee;

PROCEDURE UpdateEmployee(p\_EmployeeID NUMBER, p\_Name VARCHAR2, p\_Position VARCHAR2, p\_Salary NUMBER, p\_Department VARCHAR2) IS

BEGIN

UPDATE Employees

SET Name = p\_Name, Position = p\_Position, Salary = p\_Salary, Department = p\_Department

WHERE EmployeeID = p\_EmployeeID;

END UpdateEmployee;

FUNCTION CalculateAnnualSalary(p\_EmployeeID NUMBER) RETURN NUMBER IS

v\_Salary NUMBER;

BEGIN

SELECT Salary INTO v\_Salary

FROM Employees

WHERE EmployeeID = p\_EmployeeID;

RETURN v\_Salary \* 12;

END CalculateAnnualSalary;

END EmployeeManagement;

/

**Scenario 3:**

CREATE OR REPLACE PACKAGE AccountOperations AS

PROCEDURE OpenAccount(p\_AccountID NUMBER, p\_CustomerID NUMBER, p\_AccountType VARCHAR2, p\_Balance NUMBER);

PROCEDURE CloseAccount(p\_AccountID NUMBER);

FUNCTION GetTotalCustomerBalance(p\_CustomerID NUMBER) RETURN NUMBER;

END AccountOperations;

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

PROCEDURE OpenAccount(p\_AccountID NUMBER, p\_CustomerID NUMBER, p\_AccountType VARCHAR2, p\_Balance NUMBER) IS

BEGIN

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

VALUES (p\_AccountID, p\_CustomerID, p\_AccountType, p\_Balance, SYSDATE);

END OpenAccount;

PROCEDURE CloseAccount(p\_AccountID NUMBER) IS

BEGIN

DELETE FROM Accounts WHERE AccountID = p\_AccountID;

END CloseAccount;

FUNCTION GetTotalCustomerBalance(p\_CustomerID NUMBER) RETURN NUMBER IS

v\_TotalBalance NUMBER;

BEGIN

SELECT SUM(Balance) INTO v\_TotalBalance

FROM Accounts

WHERE CustomerID = p\_CustomerID;

RETURN v\_TotalBalance;

END GetTotalCustomerBalance;

END AccountOperations;

/