**Exercise 1: Control Structures**

**SOLUTION :**

**Scenario 1: Applying Discount to Loan Interest Rates for Customers Above 60 Years Old**

DECLARE

CURSOR cust\_cursor IS

SELECT Loans.CustomerID, Loans.InterestRate

FROM Loans

JOIN Customers ON Loans.CustomerID = Customers.CustomerID

WHERE EXTRACT(YEAR FROM SYSDATE) - EXTRACT(YEAR FROM Customers.DOB) > 60;

v\_customerID Loans.CustomerID%TYPE;

v\_interestRate Loans.InterestRate%TYPE;

BEGIN

OPEN cust\_cursor;

LOOP

FETCH cust\_cursor INTO v\_customerID, v\_interestRate;

EXIT WHEN cust\_cursor%NOTFOUND;

UPDATE Loans

SET InterestRate = InterestRate - 1

WHERE CustomerID = v\_customerID;

END LOOP;

CLOSE cust\_cursor;

COMMIT;

END;

/

**Scenario 2: Promoting Customers to VIP Status Based on Balance**

BEGIN

FOR rec IN (SELECT CustomerID FROM Customers WHERE Balance > 10000) LOOP

UPDATE Customers

SET IsVIP = TRUE

WHERE CustomerID = rec.CustomerID;

END LOOP;

COMMIT;

END;

/

**Scenario 3: Sending Reminders for Loans Due Within the Next 30 Days**

DECLARE

CURSOR loan\_cursor IS

SELECT Loans.CustomerID, Loans.EndDate

FROM Loans

WHERE Loans.EndDate BETWEEN SYSDATE AND SYSDATE + 30;

v\_customerID Loans.CustomerID%TYPE;

v\_endDate Loans.EndDate%TYPE;

BEGIN

OPEN loan\_cursor;

LOOP

FETCH loan\_cursor INTO v\_customerID, v\_endDate;

EXIT WHEN loan\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan for customer ID ' || v\_customerID || ' is due on ' || v\_endDate);

END LOOP;

CLOSE loan\_cursor;

END;

/

**Exercise 2: Error Handling**

**Scenario 1: Handling Exceptions During Fund Transfers Between Accounts**

CREATE OR REPLACE PROCEDURE SafeTransferFunds (

p\_fromAccountID IN Accounts.AccountID%TYPE,

p\_toAccountID IN Accounts.AccountID%TYPE,

p\_amount IN NUMBER

) IS

insufficient\_funds EXCEPTION;

PRAGMA EXCEPTION\_INIT(insufficient\_funds, -20101);

BEGIN

-- Check if the source account has sufficient balance

DECLARE

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance

FROM Accounts

WHERE AccountID = p\_fromAccountID;

IF v\_balance < p\_amount THEN

RAISE insufficient\_funds;

END IF;

END;

-- Transfer funds

UPDATE Accounts

SET Balance = Balance - p\_amount

WHERE AccountID = p\_fromAccountID;

UPDATE Accounts

SET Balance = Balance + p\_amount

WHERE AccountID = p\_toAccountID;

COMMIT;

EXCEPTION

WHEN insufficient\_funds THEN

ROLLBACK;

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

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('An unexpected error occurred: ' || SQLERRM);

END SafeTransferFunds;

/

**Scenario 2: Managing Errors When Updating Employee Salaries**

CREATE OR REPLACE PROCEDURE UpdateSalary (

p\_employeeID IN Employees.EmployeeID%TYPE,

p\_percentage IN NUMBER

) IS

employee\_not\_found EXCEPTION;

PRAGMA EXCEPTION\_INIT(employee\_not\_found, -20001);

BEGIN

-- Check if the employee exists

DECLARE

v\_count NUMBER;

BEGIN

SELECT COUNT(\*) INTO v\_count

FROM Employees

WHERE EmployeeID = p\_employeeID;

IF v\_count = 0 THEN

RAISE employee\_not\_found;

END IF;

END;

-- Update the salary

UPDATE Employees

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

WHERE EmployeeID = p\_employeeID;

COMMIT;

EXCEPTION

WHEN employee\_not\_found THEN

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

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('An unexpected error occurred: ' || SQLERRM);

END UpdateSalary;

/

**Scenario 3: Ensuring Data Integrity When Adding a New Customer**

CREATE OR REPLACE PROCEDURE AddNewCustomer (

p\_customerID IN Customers.CustomerID%TYPE,

p\_name IN Customers.Name%TYPE,

p\_dob IN Customers.DOB%TYPE,

p\_balance IN Customers.Balance%TYPE

) IS

customer\_exists EXCEPTION;

PRAGMA EXCEPTION\_INIT(customer\_exists, -20002);

BEGIN

-- Check if the customer already exists

DECLARE

v\_count NUMBER;

BEGIN

SELECT COUNT(\*) INTO v\_count

FROM Customers

WHERE CustomerID = p\_customerID;

IF v\_count > 0 THEN

RAISE customer\_exists;

END IF;

END;

-- Insert the new customer

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

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

COMMIT;

EXCEPTION

WHEN customer\_exists THEN

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

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('An unexpected error occurred: ' || SQLERRM);

END AddNewCustomer;

/

**Exercise 3: Stored Procedures**

Scenario 1: Processing Monthly Interest for All Savings Accounts

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

BEGIN

UPDATE Accounts

SET Balance = Balance \* 1.01

WHERE AccountType = 'Savings';

COMMIT;

END ProcessMonthlyInterest;

/

**Scenario 2: Implementing a Bonus Scheme for Employees Based on Performance**

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (

p\_department IN Employees.Department%TYPE,

p\_bonus\_percentage IN NUMBER

) IS

BEGIN

UPDATE Employees

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

WHERE Department = p\_department;

COMMIT;

END UpdateEmployeeBonus;

/

**Scenario 3: Transferring Funds Between Accounts**

CREATE OR REPLACE PROCEDURE TransferFunds (

p\_fromAccountID IN Accounts.AccountID%TYPE,

p\_toAccountID IN Accounts.AccountID%TYPE,

p\_amount IN NUMBER

) IS

BEGIN

-- Check if the source account has sufficient balance

DECLARE

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance

FROM Accounts

WHERE AccountID = p\_fromAccountID;

IF v\_balance < p\_amount THEN

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

END IF;

END;

-- Transfer funds

UPDATE Accounts

SET Balance = Balance - p\_amount

WHERE AccountID = p\_fromAccountID;

UPDATE Accounts

SET Balance = Balance + p\_amount

WHERE AccountID = p\_toAccountID;

COMMIT;

END TransferFunds;

/

**Exercise 4: Functions**

**Scenario 1: Calculating the Age of Customers for Eligibility Checks**

CREATE OR REPLACE FUNCTION CalculateAge (

p\_dob IN DATE

) RETURN NUMBER IS

v\_age NUMBER;

BEGIN

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

RETURN v\_age;

END CalculateAge;

/

**Scenario 2: Computing the Monthly Installment for a Loan**

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment (

p\_loanAmount IN NUMBER,

p\_interestRate IN NUMBER,

p\_loanDurationYears IN NUMBER

) RETURN NUMBER IS

v\_monthlyInstallment NUMBER;

BEGIN

v\_monthlyInstallment := (p\_loanAmount \* (1 + (p\_interestRate / 100) \* p\_loanDurationYears)) / (p\_loanDurationYears \* 12);

RETURN v\_monthlyInstallment;

END CalculateMonthlyInstallment;

/

**Scenario 3: Checking if a Customer Has Sufficient Balance Before Making a** **Transaction**

CREATE OR REPLACE FUNCTION HasSufficientBalance (

p\_accountID IN Accounts.AccountID%TYPE,

p\_amount IN NUMBER

) RETURN BOOLEAN IS

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance

FROM Accounts

WHERE AccountID = p\_accountID;

RETURN v\_balance >= p\_amount;

END HasSufficientBalance;

/

**Exercise 5: Triggers**

**Scenario 1: Automatically Updating 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: Maintaining an Audit Log for All Transactions**

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

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

VALUES (:NEW.TransactionID, :NEW.AccountID, :NEW.TransactionDate, :NEW.Amount, :NEW.Transaction

**Scenario 3: Preventing Deletion of Customer Records**

CREATE OR REPLACE TRIGGER PreventCustomerDeletion

BEFORE DELETE ON Customers

FOR EACH ROW

BEGIN

RAISE\_APPLICATION\_ERROR(-20002, 'Deletion of customer records is not allowed.');

END PreventCustomerDeletion;

/

**Exercise 6: Cursors**

**Scenario 1: Fetching and Displaying Customer Information**

DECLARE

CURSOR customer\_cursor IS

SELECT CustomerID, Name, DOB, Balance

FROM Customers;

v\_customerID Customers.CustomerID%TYPE;

v\_name Customers.Name%TYPE;

v\_dob Customers.DOB%TYPE;

v\_balance Customers.Balance%TYPE;

BEGIN

OPEN customer\_cursor;

LOOP

FETCH customer\_cursor INTO v\_customerID, v\_name, v\_dob, v\_balance;

EXIT WHEN customer\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Customer ID: ' || v\_customerID || ', Name: ' || v\_name || ', DOB: ' || v\_dob || ', Balance: ' || v\_balance);

END LOOP;

CLOSE customer\_cursor;

END;

/

**Scenario 2: Updating Interest Rates Using Cursors**

DECLARE

CURSOR loan\_cursor IS

SELECT LoanID, InterestRate

FROM Loans

WHERE InterestRate > 5;

v\_loanID Loans.LoanID%TYPE;

v\_interestRate Loans.InterestRate%TYPE;

BEGIN

OPEN loan\_cursor;

LOOP

FETCH loan\_cursor INTO v\_loanID, v\_interestRate;

EXIT WHEN loan\_cursor%NOTFOUND;

UPDATE Loans

SET InterestRate = InterestRate - 0.5

WHERE LoanID = v\_loanID;

END LOOP;

CLOSE loan\_cursor;

COMMIT;

END;

/

**Exercise 7: Packages**

**Scenario 1: Finding Customers with the Highest Loan Amount**

SELECT c.CustomerID, c.Name, c.Balance, l.LoanAmount

FROM Customers c

JOIN Loans l ON c.CustomerID = l.CustomerID

WHERE l.LoanAmount = (SELECT MAX(LoanAmount) FROM Loans);

**Scenario 2: Identifying Customers with Multiple Loans**

SELECT c.CustomerID, c.Name, COUNT(l.LoanID) AS LoanCount

FROM Customers c

JOIN Loans l ON c.CustomerID = l.CustomerID

GROUP BY c.CustomerID, c.Name

HAVING COUNT(l.LoanID) > 1;

**Scenario 3: Calculating the Total Balance of All Customers**

SELECT SUM(Balance) AS TotalBalance

FROM Customers;