**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.

**Solution:**

BEGIN

FOR customer IN (SELECT customer\_id, age, loan\_interest\_rate

FROM customers) LOOP

IF customer.age > 60 THEN

UPDATE customers

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

WHERE customer\_id = customer.customer\_id;

END IF;

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.

**Solution:**

BEGIN

FOR customer IN (SELECT customer\_id, balance

FROM customers) LOOP

IF customer.balance > 10000 THEN

UPDATE customers

SET IsVIP = TRUE

WHERE customer\_id = customer.customer\_id;

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.

**Solution:**

DECLARE

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

v\_customer\_name customers.name%TYPE;

BEGIN

FOR loan IN (SELECT l.loan\_id, l.due\_date, c.name

FROM loans l

JOIN customers c ON l.customer\_id = c.customer\_id

WHERE l.due\_date BETWEEN SYSDATE AND SYSDATE + 30) LOOP

v\_due\_date := loan.due\_date;

v\_customer\_name := loan.name;

DBMS\_OUTPUT.PUT\_LINE('Reminder: Dear ' || v\_customer\_name || ', your loan with ID ' || loan.loan\_id || ' is due on ' || TO\_CHAR(v\_due\_date, 'DD-MON-YYYY') || '.');

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.

**Solution:**

CREATE PROCEDURE SafeTransferFunds

@SourceAccountID INT,

@TargetAccountID INT,

@Amount DECIMAL(18, 2)

AS

BEGIN

-- Begin transaction

BEGIN TRANSACTION;

BEGIN TRY

-- Check if the source account has sufficient funds

DECLARE @SourceBalance DECIMAL(18, 2);

SELECT @SourceBalance = Balance

FROM Accounts

WHERE AccountID = @SourceAccountID;

IF @SourceBalance < @Amount

BEGIN

-- If not enough funds, raise an error

RAISERROR('Insufficient funds in the source account.', 16, 1);

RETURN;

END

-- Deduct the amount from the source account

UPDATE Accounts

SET Balance = Balance - @Amount

WHERE AccountID = @SourceAccountID;

-- Add the amount to the target account

UPDATE Accounts

SET Balance = Balance + @Amount

WHERE AccountID = @TargetAccountID;

-- Commit transaction if everything is fine

COMMIT TRANSACTION;

END TRY

BEGIN CATCH

-- Rollback transaction in case of error

ROLLBACK TRANSACTION;

-- Log the error message

DECLARE @ErrorMessage NVARCHAR(4000);

DECLARE @ErrorSeverity INT;

DECLARE @ErrorState INT;

SELECT

@ErrorMessage = ERROR\_MESSAGE(),

@ErrorSeverity = ERROR\_SEVERITY(),

@ErrorState = ERROR\_STATE();

-- Example of logging the error (could be inserted into an ErrorLog table)

INSERT INTO ErrorLog (ErrorMessage, ErrorSeverity, ErrorState, ErrorTime)

VALUES (@ErrorMessage, @ErrorSeverity, @ErrorState, GETDATE());

-- Rethrow the error

RAISERROR(@ErrorMessage, @ErrorSeverity, @ErrorState);

END CATCH;

END;

**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.

**Solution:**

CREATE PROCEDURE UpdateSalary

@EmployeeID INT,

@PercentageIncrease DECIMAL(5, 2)

AS

BEGIN

-- Begin transaction

BEGIN TRANSACTION;

BEGIN TRY

-- Check if the employee exists

IF NOT EXISTS (SELECT 1 FROM Employees WHERE EmployeeID = @EmployeeID)

BEGIN

-- If the employee does not exist, raise an error

RAISERROR('Employee ID %d does not exist.', 16, 1, @EmployeeID);

RETURN;

END

-- Update the salary by the given percentage

UPDATE Employees

SET Salary = Salary + (Salary \* @PercentageIncrease / 100)

WHERE EmployeeID = @EmployeeID;

-- Commit transaction if everything is fine

COMMIT TRANSACTION;

END TRY

BEGIN CATCH

-- Rollback transaction in case of error

ROLLBACK TRANSACTION;

-- Log the error message

DECLARE @ErrorMessage NVARCHAR(4000);

DECLARE @ErrorSeverity INT;

DECLARE @ErrorState INT;

SELECT

@ErrorMessage = ERROR\_MESSAGE(),

@ErrorSeverity = ERROR\_SEVERITY(),

@ErrorState = ERROR\_STATE();

-- Example of logging the error (could be inserted into an ErrorLog table)

INSERT INTO ErrorLog (ErrorMessage, ErrorSeverity, ErrorState, ErrorTime)

VALUES (@ErrorMessage, @ErrorSeverity, @ErrorState, GETDATE());

-- Rethrow the error

RAISERROR(@ErrorMessage, @ErrorSeverity, @ErrorState);

END CATCH;

END;

**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.

**Solution:**

CREATE PROCEDURE AddNewCustomer

@CustomerID INT,

@CustomerName NVARCHAR(100),

@ContactInfo NVARCHAR(100)

AS

BEGIN

-- Begin transaction

BEGIN TRANSACTION;

BEGIN TRY

-- Check if a customer with the same ID already exists

IF EXISTS (SELECT 1 FROM Customers WHERE CustomerID = @CustomerID)

BEGIN

-- If the customer already exists, raise an error

RAISERROR('Customer with ID %d already exists.', 16, 1, @CustomerID);

RETURN;

END

-- Insert the new customer into the Customers table

INSERT INTO Customers (CustomerID, CustomerName, ContactInfo)

VALUES (@CustomerID, @CustomerName, @ContactInfo);

-- Commit transaction if everything is fine

COMMIT TRANSACTION;

END TRY

BEGIN CATCH

-- Rollback transaction in case of error

ROLLBACK TRANSACTION;

-- Log the error message

DECLARE @ErrorMessage NVARCHAR(4000);

DECLARE @ErrorSeverity INT;

DECLARE @ErrorState INT;

SELECT

@ErrorMessage = ERROR\_MESSAGE(),

@ErrorSeverity = ERROR\_SEVERITY(),

@ErrorState = ERROR\_STATE();

-- Example of logging the error (could be inserted into an ErrorLog table)

INSERT INTO ErrorLog (ErrorMessage, ErrorSeverity, ErrorState, ErrorTime)

VALUES (@ErrorMessage, @ErrorSeverity, @ErrorState, GETDATE());

-- Rethrow the error

RAISERROR(@ErrorMessage, @ErrorSeverity, @ErrorState);

END CATCH;

END;

**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.

**Solution:**

CREATE PROCEDURE ProcessMonthlyInterest

AS

BEGIN

-- Begin transaction

BEGIN TRANSACTION;

BEGIN TRY

-- Update the balance of all savings accounts by applying 1% interest

UPDATE SavingsAccounts

SET Balance = Balance + (Balance \* 0.01);

-- Commit transaction if everything is fine

COMMIT TRANSACTION;

END TRY

BEGIN CATCH

-- Rollback transaction in case of error

ROLLBACK TRANSACTION;

-- Log the error message

DECLARE @ErrorMessage NVARCHAR(4000);

DECLARE @ErrorSeverity INT;

DECLARE @ErrorState INT;

SELECT

@ErrorMessage = ERROR\_MESSAGE(),

@ErrorSeverity = ERROR\_SEVERITY(),

@ErrorState = ERROR\_STATE();

-- Example of logging the error (could be inserted into an ErrorLog table)

INSERT INTO ErrorLog (ErrorMessage, ErrorSeverity, ErrorState, ErrorTime)

VALUES (@ErrorMessage, @ErrorSeverity, @ErrorState, GETDATE());

-- Rethrow the error

RAISERROR(@ErrorMessage, @ErrorSeverity, @ErrorState);

END CATCH;

END;

**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.

**Solution:**

CREATE PROCEDURE UpdateEmployeeBonus

@DepartmentID INT,

@BonusPercentage DECIMAL(5, 2)

AS

BEGIN

BEGIN TRANSACTION;

BEGIN TRY

-- Update the salary of employees in the specified department by adding the bonus percentage

UPDATE Employees

SET Salary = Salary + (Salary \* @BonusPercentage / 100)

WHERE DepartmentID = @DepartmentID;

-- Commit transaction if everything is fine

COMMIT TRANSACTION;

END TRY

BEGIN CATCH

-- Rollback transaction in case of error

ROLLBACK TRANSACTION;

-- Log the error message

DECLARE @ErrorMessage NVARCHAR(4000);

DECLARE @ErrorSeverity INT;

DECLARE @ErrorState INT;

SELECT

@ErrorMessage = ERROR\_MESSAGE(),

@ErrorSeverity = ERROR\_SEVERITY(),

@ErrorState = ERROR\_STATE();

-- Example of logging the error (could be inserted into an ErrorLog table)

INSERT INTO ErrorLog (ErrorMessage, ErrorSeverity, ErrorState, ErrorTime)

VALUES (@ErrorMessage, @ErrorSeverity, @ErrorState, GETDATE());

-- Rethrow the error

RAISERROR(@ErrorMessage, @ErrorSeverity, @ErrorState);

END CATCH;

END;

**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.

**Solution:**

CREATE PROCEDURE TransferFunds

@SourceAccountID INT,

@TargetAccountID INT,

@Amount DECIMAL(18, 2)

AS

BEGIN

-- Begin transaction

BEGIN TRANSACTION;

BEGIN TRY

-- Check if the source account has sufficient balance

DECLARE @SourceBalance DECIMAL(18, 2);

SELECT @SourceBalance = Balance

FROM Accounts

WHERE AccountID = @SourceAccountID;

IF @SourceBalance < @Amount

BEGIN

-- If not enough funds, raise an error

RAISERROR('Insufficient funds in the source account.', 16, 1);

RETURN;

END

-- Deduct the amount from the source account

UPDATE Accounts

SET Balance = Balance - @Amount

WHERE AccountID = @SourceAccountID;

-- Add the amount to the target account

UPDATE Accounts

SET Balance = Balance + @Amount

WHERE AccountID = @TargetAccountID;

-- Commit transaction if everything is fine

COMMIT TRANSACTION;

END TRY

BEGIN CATCH

-- Rollback transaction in case of error

ROLLBACK TRANSACTION;

-- Log the error message

DECLARE @ErrorMessage NVARCHAR(4000);

DECLARE @ErrorSeverity INT;

DECLARE @ErrorState INT;

SELECT

@ErrorMessage = ERROR\_MESSAGE(),

@ErrorSeverity = ERROR\_SEVERITY(),

@ErrorState = ERROR\_STATE();

-- Example of logging the error (could be inserted into an ErrorLog table)

INSERT INTO ErrorLog (ErrorMessage, ErrorSeverity, ErrorState, ErrorTime)

VALUES (@ErrorMessage, @ErrorSeverity, @ErrorState, GETDATE());

-- Rethrow the error

RAISERROR(@ErrorMessage, @ErrorSeverity, @ErrorState);

END CATCH;

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.

**Solution:**

CREATE FUNCTION CalculateAge (@DateOfBirth DATE)

RETURNS INT

AS

BEGIN

DECLARE @CurrentDate DATE = GETDATE();

DECLARE @Age INT;

SET @Age = DATEDIFF(YEAR, @DateOfBirth, @CurrentDate);

-- Adjust the age if the current date is before the birthday in the current year

IF (MONTH(@CurrentDate) < MONTH(@DateOfBirth))

OR (MONTH(@CurrentDate) = MONTH(@DateOfBirth) AND DAY(@CurrentDate) < DAY(@DateOfBirth))

BEGIN

SET @Age = @Age - 1;

END

RETURN @Age;

END;

**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.

**Solution:**

CREATE FUNCTION CalculateMonthlyInstallment (

@LoanAmount DECIMAL(18, 2),

@AnnualInterestRate DECIMAL(5, 2),

@LoanDurationYears INT

)

RETURNS DECIMAL(18, 2)

AS

BEGIN

DECLARE @MonthlyInterestRate DECIMAL(18, 4) = @AnnualInterestRate / 1200.0;

DECLARE @NumberOfPayments INT = @LoanDurationYears \* 12;

DECLARE @MonthlyInstallment DECIMAL(18, 2);

IF @MonthlyInterestRate = 0

BEGIN

-- If interest rate is 0, the installment is simply the loan amount divided by the number of payments

SET @MonthlyInstallment = @LoanAmount / @NumberOfPayments;

END

ELSE

BEGIN

-- Calculate monthly installment using the amortization formula

SET @MonthlyInstallment = @LoanAmount \* @MonthlyInterestRate /

(1 - POWER(1 + @MonthlyInterestRate, -@NumberOfPayments));

END

RETURN @MonthlyInstallment;

END;

**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.

**Solution:**

CREATE FUNCTION HasSufficientBalance (

@AccountID INT,

@Amount DECIMAL(18, 2)

)

RETURNS BIT

AS

BEGIN

DECLARE @Balance DECIMAL(18, 2);

-- Retrieve the current balance of the specified account

SELECT @Balance = Balance

FROM Accounts

WHERE AccountID = @AccountID;

-- Return 1 (TRUE) if the balance is sufficient, otherwise 0 (FALSE)

IF @Balance >= @Amount

BEGIN

RETURN 1;

END

ELSE

BEGIN

RETURN 0;

END

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.

**Solution:**

CREATE TRIGGER UpdateCustomerLastModified

ON Customers

AFTER UPDATE

AS

BEGIN

SET NOCOUNT ON;

-- Update the LastModified column to the current date

UPDATE Customers

SET LastModified = GETDATE()

FROM Customers

INNER JOIN inserted ON Customers.CustomerID = inserted.CustomerID;

-- Optionally, you can log the update or take other actions if necessary

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.

**Solution:**

CREATE TRIGGER LogTransaction

ON Transactions

AFTER INSERT

AS

BEGIN

SET NOCOUNT ON;

-- Insert a record into the AuditLog table for each new transaction

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

SELECT

inserted.TransactionID,

inserted.AccountID,

inserted.Amount,

inserted.TransactionType,

inserted.TransactionDate,

GETDATE() AS AuditDate

FROM inserted;

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.

**Solution:**

CREATE TRIGGER CheckTransactionRules

ON Transactions

INSTEAD OF INSERT

AS

BEGIN

SET NOCOUNT ON;

DECLARE @AccountID INT;

DECLARE @Amount DECIMAL(18, 2);

DECLARE @TransactionType NVARCHAR(50);

-- Get values from the inserted row

SELECT

@AccountID = inserted.AccountID,

@Amount = inserted.Amount,

@TransactionType = inserted.TransactionType

FROM inserted;

-- Check if it's a withdrawal and if the balance is sufficient

IF @TransactionType = 'Withdrawal'

BEGIN

DECLARE @Balance DECIMAL(18, 2);

-- Get the current balance of the account

SELECT @Balance = Balance FROM Accounts WHERE AccountID = @AccountID;

IF @Balance < @Amount

BEGIN

-- Raise an error if the balance is insufficient

RAISERROR('Insufficient balance for the withdrawal.', 16, 1);

ROLLBACK TRANSACTION;

RETURN;

END

END

-- Check if it's a deposit and the amount is positive

IF @TransactionType = 'Deposit'

BEGIN

IF @Amount <= 0

BEGIN

-- Raise an error if the deposit amount is not positive

RAISERROR('Deposit amount must be positive.', 16, 1);

ROLLBACK TRANSACTION;

RETURN;

END

END

-- If all checks pass, insert the transaction

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

SELECT AccountID, Amount, TransactionType, TransactionDate

FROM inserted;

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.

**Solution:**

DECLARE

-- Define cursor to fetch all transactions for the current month

CURSOR cur\_transactions IS

SELECT CustomerID, TransactionDate, TransactionType, Amount

FROM Transactions

WHERE EXTRACT(MONTH FROM TransactionDate) = EXTRACT(MONTH FROM SYSDATE)

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

-- Define variables to hold cursor data

v\_CustomerID Transactions.CustomerID%TYPE;

v\_TransactionDate Transactions.TransactionDate%TYPE;

v\_TransactionType Transactions.TransactionType%TYPE;

v\_Amount Transactions.Amount%TYPE;

BEGIN

-- Open the cursor

OPEN cur\_transactions;

-- Loop through the cursor and print statements

LOOP

FETCH cur\_transactions INTO v\_CustomerID, v\_TransactionDate, v\_TransactionType, v\_Amount;

EXIT WHEN cur\_transactions%NOTFOUND;

-- Print statement for the customer

DBMS\_OUTPUT.PUT\_LINE('Customer ID: ' || v\_CustomerID);

DBMS\_OUTPUT.PUT\_LINE('Date: ' || TO\_CHAR(v\_TransactionDate, 'YYYY-MM-DD'));

DBMS\_OUTPUT.PUT\_LINE('Transaction Type: ' || v\_TransactionType);

DBMS\_OUTPUT.PUT\_LINE('Amount: ' || v\_Amount);

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

END LOOP;

-- Close the cursor

CLOSE cur\_transactions;

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.

**Solution:**

DECLARE

-- Define cursor to fetch all accounts

CURSOR cur\_accounts IS

SELECT AccountID, Balance

FROM Accounts;

-- Define variables to hold cursor data

v\_AccountID Accounts.AccountID%TYPE;

v\_Balance Accounts.Balance%TYPE;

-- Define the annual fee

v\_AnnualFee CONSTANT NUMBER := 50.00;

BEGIN

-- Open the cursor

OPEN cur\_accounts;

-- Loop through the cursor and apply the annual fee

LOOP

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

EXIT WHEN cur\_accounts%NOTFOUND;

-- Deduct the annual fee from the account balance

UPDATE Accounts

SET Balance = Balance - v\_AnnualFee

WHERE AccountID = v\_AccountID;

-- Print the result for verification

DBMS\_OUTPUT.PUT\_LINE('Account ID: ' || v\_AccountID || ' - Fee Applied: ' || v\_AnnualFee);

END LOOP;

-- Close the cursor

CLOSE cur\_accounts;

-- Commit the transaction

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.

**Solution:**

DECLARE

-- Define cursor to fetch all loans

CURSOR cur\_loans IS

SELECT LoanID, InterestRate

FROM Loans;

-- Define variables to hold cursor data

v\_LoanID Loans.LoanID%TYPE;

v\_InterestRate Loans.InterestRate%TYPE;

-- Define the new interest rate policy (example: increase by 0.5%)

v\_NewInterestRate CONSTANT NUMBER := 0.50;

BEGIN

-- Open the cursor

OPEN cur\_loans;

-- Loop through the cursor and update the interest rates

LOOP

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

EXIT WHEN cur\_loans%NOTFOUND;

-- Update the interest rate based on the new policy

UPDATE Loans

SET InterestRate = v\_InterestRate + v\_NewInterestRate

WHERE LoanID = v\_LoanID;

-- Print the result for verification

DBMS\_OUTPUT.PUT\_LINE('Loan ID: ' || v\_LoanID || ' - New Interest Rate: ' || (v\_InterestRate + v\_NewInterestRate));

END LOOP;

-- Close the cursor

CLOSE cur\_loans;

-- Commit the transaction

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.

**Solution:**

CREATE OR REPLACE PACKAGE CustomerManagement AS

-- Procedure to add a new customer

PROCEDURE AddNewCustomer(

p\_CustomerID IN NUMBER,

p\_CustomerName IN VARCHAR2,

p\_DateOfBirth IN DATE,

p\_Address IN VARCHAR2

);

-- Procedure to update customer details

PROCEDURE UpdateCustomerDetails(

p\_CustomerID IN NUMBER,

p\_CustomerName IN VARCHAR2,

p\_DateOfBirth IN DATE,

p\_Address IN VARCHAR2

);

-- Function to get the customer's balance

FUNCTION GetCustomerBalance(

p\_CustomerID IN NUMBER

) RETURN NUMBER;

END CustomerManagement;

**Package Body:**

**CustomerManagement**

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

PROCEDURE AddNewCustomer(

p\_CustomerID IN NUMBER,

p\_CustomerName IN VARCHAR2,

p\_DateOfBirth IN DATE,

p\_Address IN VARCHAR2

) IS

BEGIN

INSERT INTO Customers (CustomerID, CustomerName, DateOfBirth, Address, LastModified)

VALUES (p\_CustomerID, p\_CustomerName, p\_DateOfBirth, p\_Address, SYSDATE);

END AddNewCustomer;

PROCEDURE UpdateCustomerDetails(

p\_CustomerID IN NUMBER,

p\_CustomerName IN VARCHAR2,

p\_DateOfBirth IN DATE,

p\_Address IN VARCHAR2

) IS

BEGIN

UPDATE Customers

SET CustomerName = p\_CustomerName,

DateOfBirth = p\_DateOfBirth,

Address = p\_Address,

LastModified = SYSDATE

WHERE CustomerID = p\_CustomerID;

END UpdateCustomerDetails;

FUNCTION GetCustomerBalance(

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);

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.

**Solurion:**

CREATE OR REPLACE PACKAGE EmployeeManagement AS

-- Procedure to hire a new employee

PROCEDURE HireEmployee(

p\_EmployeeID IN NUMBER,

p\_EmployeeName IN VARCHAR2,

p\_DepartmentID IN NUMBER,

p\_Salary IN NUMBER,

p\_HireDate IN DATE

);

-- Procedure to update employee details

PROCEDURE UpdateEmployeeDetails(

p\_EmployeeID IN NUMBER,

p\_EmployeeName IN VARCHAR2,

p\_DepartmentID IN NUMBER,

p\_Salary IN NUMBER

);

-- Function to calculate the annual salary of an employee

FUNCTION CalculateAnnualSalary(

p\_EmployeeID IN NUMBER

) RETURN NUMBER;

END EmployeeManagement;

**Package body :**

**EmployeeManagement**

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

PROCEDURE HireEmployee(

p\_EmployeeID IN NUMBER,

p\_EmployeeName IN VARCHAR2,

p\_DepartmentID IN NUMBER,

p\_Salary IN NUMBER,

p\_HireDate IN DATE

) IS

BEGIN

INSERT INTO Employees (EmployeeID, EmployeeName, DepartmentID, Salary, HireDate)

VALUES (p\_EmployeeID, p\_EmployeeName, p\_DepartmentID, p\_Salary, p\_HireDate);

END HireEmployee;

PROCEDURE UpdateEmployeeDetails(

p\_EmployeeID IN NUMBER,

p\_EmployeeName IN VARCHAR2,

p\_DepartmentID IN NUMBER,

p\_Salary IN NUMBER

) IS

BEGIN

UPDATE Employees

SET EmployeeName = p\_EmployeeName,

DepartmentID = p\_DepartmentID,

Salary = p\_Salary

WHERE EmployeeID = p\_EmployeeID;

END UpdateEmployeeDetails;

FUNCTION CalculateAnnualSalary(

p\_EmployeeID IN NUMBER

) RETURN NUMBER IS

v\_Salary NUMBER;

BEGIN

SELECT Salary \* 12

INTO v\_Salary

FROM Employees

WHERE EmployeeID = p\_EmployeeID;

RETURN v\_Salary;

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.

**Solution:**

CREATE OR REPLACE PACKAGE AccountOperations AS

-- Procedure to open a new account

PROCEDURE OpenAccount(

p\_AccountID IN NUMBER,

p\_CustomerID IN NUMBER,

p\_AccountType IN VARCHAR2,

p\_InitialDeposit IN NUMBER

);

-- Procedure to close an account

PROCEDURE CloseAccount(

p\_AccountID IN NUMBER

);

-- Function to get the total balance of a customer across all accounts

FUNCTION GetTotalBalance(

p\_CustomerID IN NUMBER

) RETURN NUMBER;

END AccountOperations;

**Package body:**

**AccountOperations**

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

PROCEDURE OpenAccount(

p\_AccountID IN NUMBER,

p\_CustomerID IN NUMBER,

p\_AccountType IN VARCHAR2,

p\_InitialDeposit IN NUMBER

) IS

BEGIN

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

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

END OpenAccount;

PROCEDURE CloseAccount(

p\_AccountID IN NUMBER

) IS

BEGIN

DELETE FROM Accounts

WHERE AccountID = p\_AccountID;

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);

END GetTotalBalance;

END AccountOperations;

**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'));*