## 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.

```
DECLARE
  CURSOR customer_cursor IS
    SELECT c.CustomerID, l.LoanID, l.InterestRate
    FROM Customers c
    JOIN Loans l ON c.CustomerID = l.CustomerID
    WHERE TRUNC(MONTHS_BETWEEN(SYSDATE, c.DOB) / 12) > 60;
  v_CustomerID Customers.CustomerID%TYPE;
  v_LoanID Loans.LoanID%TYPE;
  v_InterestRate Loans.InterestRate%TYPE;
BEGIN
  OPEN customer_cursor;
  LOOP
    FETCH customer_cursor INTO v_CustomerID, v_LoanID, v_InterestRate;
    EXIT WHEN customer_cursor%NOTFOUND;
    UPDATE Loans
    SET InterestRate = v_InterestRate * 0.99
    WHERE LoanID = v_LoanID;
  END LOOP;
  CLOSE customer_cursor;
  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.

```
ALTER TABLE Customers ADD (IsVIP VARCHAR2(3));
UPDATE Customers SET IsVIP = 'NO';
COMMIT;
DECLARE
  CURSOR customer_cursor IS
    SELECT CustomerID, Balance
    FROM Customers;
  v_CustomerID Customers.CustomerID%TYPE;
  v_Balance Customers.Balance%TYPE;
BEGIN
  OPEN customer_cursor;
```

```
  LOOP
    FETCH customer_cursor INTO v_CustomerID, v_Balance;
    EXIT WHEN customer_cursor%NOTFOUND;
    IF v_Balance > 10000 THEN
      UPDATE Customers
      SET IsVIP = 'YES'
      WHERE CustomerID = v_CustomerID;
    ELSE
      UPDATE Customers
      SET IsVIP = 'NO'
      WHERE CustomerID = v_CustomerID;
    END IF;
  END LOOP;
  CLOSE customer_cursor;
  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.

```
DECLARE
  CURSOR loan_cursor IS
    SELECT c.CustomerID, c.Name, c.Email, l.LoanID, l.EndDate
    FROM Customers c
    JOIN Loans l ON c.CustomerID = l.CustomerID
    WHERE l.EndDate BETWEEN SYSDATE AND SYSDATE + 30;
  v_CustomerID Customers.CustomerID%TYPE;
  v_Name Customers.Name%TYPE;
  v_Email Customers.Email%TYPE;
  v_LoanID Loans.LoanID%TYPE;
  v_EndDate Loans.EndDate%TYPE;
BEGIN
  OPEN loan_cursor;
  LOOP
    FETCH loan_cursor INTO v_CustomerID, v_Name, v_Email, v_LoanID, v_EndDate;
    EXIT WHEN loan_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Reminder: Customer ' || v_Name || ' (ID: ' ||
v_CustomerID || ')');
    DBMS_OUTPUT.PUT_LINE('Your loan with Loan ID: ' || v_LoanID || ' is due on ' ||
TO_CHAR(v_EndDate, 'YYYY-MM-DD') || '.');
    DBMS_OUTPUT.PUT_LINE('Please ensure payment is made by the due date.');
    DBMS_OUTPUT.PUT_LINE('');  -- For a blank line between messages
```

```
    END LOOP;
    CLOSE loan_cursor;
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.

```
CREATE OR REPLACE PROCEDURE SafeTransferFunds( p_SourceAccountID IN
Accounts.AccountID%TYPE,p_DestinationAccountID IN Accounts.AccountID%TYPE,
p_Amount IN NUMBER)
IS
   v_SourceBalance Accounts.Balance%TYPE;
   v_DestinationBalance Accounts.Balance%TYPE;
   insufficient_funds EXCEPTION;
   account_not_found EXCEPTION;
BEGIN
   SELECT Balance INTO v_SourceBalance
   FROM Accounts
   WHERE AccountID = p_SourceAccountID
   FOR UPDATE;
   SELECT Balance INTO v_DestinationBalance
   FROM Accounts
   WHERE AccountID = p_DestinationAccountID
   FOR UPDATE;
   IF v_SourceBalance < p_Amount THEN
      RAISE insufficient_funds;
   END IF;
   UPDATE Accounts
   SET Balance = Balance - p_Amount
   WHERE AccountID = p_SourceAccountID;
   UPDATE Accounts
   SET Balance = Balance + p_Amount
   WHERE AccountID = p_DestinationAccountID;
   COMMIT;

DBMS_OUTPUT.PUT_LINE('Transfer of ' || p_Amount || ' from Account ' ||
p_SourceAccountID || ' to Account ' || p_DestinationAccountID || ' completed
successfully.');
```

```
EXCEPTION
   WHEN insufficient_funds THEN
      ROLLBACK;
      DBMS_OUTPUT.PUT_LINE('Error: Insufficient funds in Account ' ||
p_SourceAccountID || '. Transfer aborted.');
WHEN NO_DATA_FOUND THEN
  ROLLBACK;
  DBMS_OUTPUT.PUT_LINE('Error: One or both accounts not found. Transfer aborted.');
WHEN OTHERS THEN
      ROLLBACK;
      DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM || '. Transfer aborted.');
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.

```
CREATE OR REPLACE PROCEDURE UpdateSalary( p_EmployeeID IN
Employees.EmployeeID%TYPE,p_Percentage IN NUMBER)
IS
v_Salary Employees.Salary%TYPE;
employee_not_found EXCEPTION;
BEGIN
   SELECT Salary INTO v_Salary
   FROM Employees
   WHERE EmployeeID = p_EmployeeID;
   v_Salary := v_Salary * (1 + p_Percentage / 100);
   UPDATE Employees
   SET Salary = v_Salary
   WHERE EmployeeID = p_EmployeeID;
   COMMIT;
   DBMS_OUTPUT.PUT_LINE('Salary of Employee ID ' || p_EmployeeID || ' increased by '
|| p_Percentage || '%. New Salary: ' || v_Salary);

EXCEPTION
WHEN NO_DATA_FOUND THEN
      ROLLBACK;
      DBMS_OUTPUT.PUT_LINE('Error: Employee ID ' || p_EmployeeID || ' does not exist.
Salary update aborted.');
WHEN OTHERS THEN
      ROLLBACK;
      DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM || '. Salary update aborted.');
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.

```
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
BEGIN
   INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)
   VALUES (p_CustomerID, p_Name, p_DOB, p_Balance, SYSDATE);
   COMMIT;

   DBMS_OUTPUT.PUT_LINE('Customer ID ' || p_CustomerID || ' added successfully.');
EXCEPTION
WHEN DUP_VAL_ON_INDEX THEN
      ROLLBACK;
      DBMS_OUTPUT.PUT_LINE('Error: Customer ID ' || p_CustomerID || ' already exists.
Insertion aborted.');
WHEN OTHERS THEN
      ROLLBACK;
      DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM || '. Insertion aborted.');
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.

```
CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest
IS
CURSOR savings_accounts_cursor IS
      SELECT AccountID, Balance
      FROM Accounts
      WHERE AccountType = 'Savings'
      FOR UPDATE OF Balance;
   v_AccountID Accounts.AccountID%TYPE;
```

```
    v_Balance Accounts.Balance%TYPE;
    v_Interest NUMBER;
  BEGIN
    OPEN savings_accounts_cursor;
    LOOP
      FETCH savings_accounts_cursor INTO v_AccountID, v_Balance;
      EXIT WHEN savings_accounts_cursor%NOTFOUND;
      v_Interest := v_Balance * 0.01;
      UPDATE Accounts
      SET Balance = Balance + v_Interest,
      LastModified = SYSDATE
      WHERE CURRENT OF savings_accounts_cursor;
    END LOOP;
 CLOSE savings_accounts_cursor;
 COMMIT;

   DBMS_OUTPUT.PUT_LINE('Monthly interest has been processed for all savings
 accounts.');
 EXCEPTION
    WHEN OTHERS THEN
      ROLLBACK;
      DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM || '. Monthly interest processing
 aborted.');
 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.

```
CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(  p_Department IN
Employees.Department%TYPE,p_BonusPercentage IN NUMBER)
IS
CURSOR dept_employees_cursor IS
      SELECT EmployeeID, Salary
      FROM Employees
      WHERE Department = p_Department
      FOR UPDATE OF Salary;
   v_EmployeeID Employees.EmployeeID%TYPE;
   v_Salary Employees.Salary%TYPE;
   v_Bonus NUMBER;
BEGIN
   OPEN dept_employees_cursor;
```

```
        LOOP
            FETCH dept_employees_cursor INTO v_EmployeeID, v_Salary;
            EXIT WHEN dept_employees_cursor%NOTFOUND;
            v_Bonus := v_Salary * (p_BonusPercentage / 100);
            UPDATE Employees
            SET Salary = Salary + v_Bonus
            WHERE CURRENT OF dept_employees_cursor;
        END LOOP;

        CLOSE dept_employees_cursor;
        COMMIT;
        DBMS_OUTPUT.PUT_LINE('Bonus of ' || p_BonusPercentage || '% has been applied to all
employees in the ' || p_Department || ' department.');
        EXCEPTION
            WHEN OTHERS THEN
            ROLLBACK;
            DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM || '. Bonus update aborted.');
        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.

```
CREATE OR REPLACE PROCEDURE TransferFunds(p_SourceAccountID
IN

Accounts.AccountID%TYPE, p_DestinationAccountID IN Accounts.AccountID%TYPE, p_Amount
IN NUMBER
)

IS
   v_SourceBalance Accounts.Balance%TYPE;
   v_DestinationBalance Accounts.Balance%TYPE;
   insufficient_funds EXCEPTION;
   account_not_found EXCEPTION;
BEGIN
   SELECT Balance INTO v_SourceBalance
   FROM Accounts
   WHERE AccountID = p_SourceAccountID
   FOR UPDATE;
```

```
    SELECT Balance INTO v_DestinationBalance
    FROM Accounts
    WHERE AccountID = p_DestinationAccountID
    FOR UPDATE;
    IF v_SourceBalance < p_Amount THEN
      RAISE insufficient_funds;
    END IF;
    UPDATE Accounts
    SET Balance = Balance - p_Amount
    WHERE AccountID = p_SourceAccountID;
    UPDATE Accounts
    SET Balance = Balance + p_Amount
    WHERE AccountID = p_DestinationAccountID;
    COMMIT;

  DBMS_OUTPUT.PUT_LINE('Transfer of ' || p_Amount || ' from Account ' || p_SourceAccountID
|| ' to Account ' || p_DestinationAccountID || ' completed successfully.');
  EXCEPTION
    WHEN insufficient_funds THEN
      ROLLBACK;
      DBMS_OUTPUT.PUT_LINE('Error: Insufficient funds in Account ' || p_SourceAccountID || '.
  Transfer aborted.');
    WHEN NO_DATA_FOUND THEN
      ROLLBACK;
      DBMS_OUTPUT.PUT_LINE('Error: One or both accounts not found. Transfer aborted.');
    WHEN OTHERS THEN
      ROLLBACK;
      DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM || '. Transfer aborted.');
  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 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:** 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.

```
CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(
    p_LoanAmount IN NUMBER,
    p_AnnualInterestRate IN NUMBER,
    p_LoanDurationYears IN NUMBER
) RETURN NUMBER
IS
    v_MonthlyInterestRate NUMBER;
    v_NumberOfPayments NUMBER;
    v_MonthlyInstallment NUMBER;
BEGIN
    v_MonthlyInterestRate := p_AnnualInterestRate / 12 / 100;
    v_NumberOfPayments := p_LoanDurationYears * 12;
    IF v_MonthlyInterestRate > 0 THEN
        v_MonthlyInstallment := p_LoanAmount * v_MonthlyInterestRate * POWER(1 +
v_MonthlyInterestRate, v_NumberOfPayments) /
                            (POWER(1 + v_MonthlyInterestRate, v_NumberOfPayments) - 1);
    ELSE
        v_MonthlyInstallment := p_LoanAmount / v_NumberOfPayments;
    END IF;
    RETURN v_MonthlyInstallment;
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.

```
CREATE OR REPLACE FUNCTION HasSufficientBalance(
    p_AccountID IN Accounts.AccountID%TYPE,
    p_Amount IN NUMBER
) RETURN BOOLEAN
IS
    v_Balance Accounts.Balance%TYPE;
BEGIN
```

```
    SELECT Balance INTO v_Balance
    FROM Accounts
    WHERE AccountID = p_AccountID;
    IF v_Balance >= p_Amount THEN
        RETURN TRUE;
    ELSE
        RETURN FALSE;
    END IF;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RETURN FALSE;
    WHEN OTHERS THEN
        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.

```
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.

```
CREATE TABLE AuditLog (
    LogID NUMBER PRIMARY KEY,
    TransactionID NUMBER,
    AccountID NUMBER,
    TransactionDate DATE,
    Amount NUMBER,
```

```
    TransactionType VARCHAR2(10),
    LogTimestamp DATE
);

 CREATE OR REPLACE TRIGGER LogTransaction
AFTER INSERT ON Transactions
FOR EACH ROW
BEGIN
  INSERT INTO AuditLog (
            LogID,
            TransactionID,
            AccountID,
            TransactionDate,
            Amount,
            TransactionType,
            LogTimestamp
            ) VALUES (
              AuditLog_seq.NEXTVAL,
                :NEW.TransactionID,
                :NEW.AccountID,
                :NEW.TransactionDate,
                :NEW.Amount,
                :NEW.TransactionType,
                SYSDATE
                );
           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.

```
CREATE OR REPLACE TRIGGER CheckTransactionRules
BEFORE INSERT ON Transactions
FOR EACH ROW
DECLARE
   v_Balance Accounts.Balance%TYPE;
   insufficient_funds EXCEPTION;
   negative_deposit EXCEPTION;
BEGIN

   SELECT Balance INTO v_Balance
   FROM Accounts
   WHERE AccountID = :NEW.AccountID
```

```
    FOR UPDATE;
    IF :NEW.TransactionType = 'Withdrawal' THEN
        IF :NEW.Amount > v_Balance THEN
            RAISE insufficient_funds;
        END IF;
    ELSIF :NEW.TransactionType = 'Deposit' THEN
        -- Check if the deposit amount is positive
        IF :NEW.Amount <= 0 THEN
            RAISE negative_deposit;
        END IF;
    END IF;

EXCEPTION
    WHEN insufficient_funds THEN
        RAISE_APPLICATION_ERROR(-20001, 'Insufficient funds for withdrawal.');
    WHEN negative_deposit THEN
        RAISE_APPLICATION_ERROR(-20002, 'Deposit amount must be positive.');
    WHEN OTHERS THEN
        RAISE_APPLICATION_ERROR(-20003, 'An unexpected error occurred: ' || SQLERRM);
END CheckTransactionRules;
/
```

## 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.

```
DECLARE
    -- Declare a record type for transactions
    CURSOR cur_Transactions IS
        SELECT t.TransactionID, t.AccountID, t.TransactionDate, t.Amount, t.TransactionType,
a.CustomerID, c.Name
        FROM Transactions t
        JOIN Accounts a ON t.AccountID = a.AccountID
        JOIN Customers c ON a.CustomerID = c.CustomerID
        WHERE t.TransactionDate BETWEEN TRUNC(SYSDATE, 'MM') AND
LAST_DAY(SYSDATE);

    -- Record variable to hold each row fetched by the cursor
    r_Transaction cur_Transactions%ROWTYPE;
```

```
BEGIN
  OPEN cur_Transactions;
  LOOP
    FETCH cur_Transactions INTO r_Transaction;
    EXIT WHEN cur_Transactions%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Customer Name: ' || r_Transaction.Name);
    DBMS_OUTPUT.PUT_LINE('Customer ID: ' || r_Transaction.CustomerID);
    DBMS_OUTPUT.PUT_LINE('Account ID: ' || r_Transaction.AccountID);
    DBMS_OUTPUT.PUT_LINE('Transaction ID: ' || r_Transaction.TransactionID);
    DBMS_OUTPUT.PUT_LINE('Transaction Date: ' ||
TO_CHAR(r_Transaction.TransactionDate, 'YYYY-MM-DD'));
    DBMS_OUTPUT.PUT_LINE('Transaction Amount: ' || r_Transaction.Amount);
    DBMS_OUTPUT.PUT_LINE('Transaction Type: ' || r_Transaction.TransactionType);
    DBMS_OUTPUT.PUT_LINE('----------------------------------------');
  END LOOP;
  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.

```
DECLARE
  annual_fee CONSTANT NUMBER := 50;
    CURSOR cur_Accounts IS
     SELECT AccountID, Balance
     FROM Accounts
     FOR UPDATE OF Balance;

  r_Account cur_Accounts%ROWTYPE;

BEGIN

  OPEN cur_Accounts;
  LOOP
    FETCH cur_Accounts INTO r_Account;
    EXIT WHEN cur_Accounts%NOTFOUND;
    IF r_Account.Balance >= annual_fee THEN
      -- Deduct the annual fee from the account balance
      UPDATE Accounts
      SET Balance = Balance - annual_fee
      WHERE CURRENT OF cur_Accounts;
```

```
            -- Print the updated balance for verification
            DBMS_OUTPUT.PUT_LINE('Account ID: ' || r_Account.AccountID || ' - New
Balance: ' || (r_Account.Balance - annual_fee));
        ELSE
            -- Print a message if there are insufficient funds
            DBMS_OUTPUT.PUT_LINE('Account ID: ' || r_Account.AccountID || ' has
insufficient funds for the annual fee.');
        END IF;
    END LOOP;

    -- Close the cursor
    CLOSE cur_Accounts;
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.

```
DECLARE
    interest_rate_increment CONSTANT NUMBER := 1;

    CURSOR cur_Loans IS
        SELECT LoanID, InterestRate
        FROM Loans
        FOR UPDATE OF InterestRate;
    r_Loan cur_Loans%ROWTYPE;

BEGIN
    OPEN cur_Loans;
    LOOP
        FETCH cur_Loans INTO r_Loan;
        EXIT WHEN cur_Loans%NOTFOUND;
        UPDATE Loans
        SET InterestRate = r_Loan.InterestRate + interest_rate_increment
        WHERE CURRENT OF cur_Loans;
        DBMS_OUTPUT.PUT_LINE('Loan ID: ' || r_Loan.LoanID || ' - New Interest Rate: ' ||
(r_Loan.InterestRate + interest_rate_increment));

    END LOOP;
    CLOSE cur_Loans;
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.

```
CREATE OR REPLACE PACKAGE CustomerManagement AS
  PROCEDURE AddNewCustomer(
    p_CustomerID IN NUMBER,
    p_Name IN VARCHAR2,
    p_DOB IN DATE,
    p_Balance IN NUMBER
  );
  PROCEDURE UpdateCustomerDetails(
    p_CustomerID IN NUMBER,
    p_Name IN VARCHAR2,
    p_DOB IN DATE,
    p_Balance IN NUMBER
  );
  FUNCTION GetCustomerBalance(
    p_CustomerID IN NUMBER
  ) RETURN NUMBER;
END CustomerManagement;


CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

  PROCEDURE AddNewCustomer(
    p_CustomerID IN NUMBER,
    p_Name IN VARCHAR2,
    p_DOB IN DATE,
    p_Balance IN NUMBER
  ) IS
    e_DuplicateCustomerID EXCEPTION;
  BEGIN
    INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)
    VALUES (p_CustomerID, p_Name, p_DOB, p_Balance, SYSDATE);
  EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
      RAISE e_DuplicateCustomerID;
    WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('An error occurred: ' || SQLERRM);
  END AddNewCustomer;
  PROCEDURE UpdateCustomerDetails(
    p_CustomerID IN NUMBER,
```

```
    p_Name IN VARCHAR2,
    p_DOB IN DATE,
    p_Balance IN NUMBER
  ) IS
    e_CustomerNotFound EXCEPTION;
  BEGIN
    UPDATE Customers
    SET Name = p_Name,
      DOB = p_DOB,
      Balance = p_Balance,
      LastModified = SYSDATE
    WHERE CustomerID = p_CustomerID;
    IF SQL%ROWCOUNT = 0 THEN
      RAISE e_CustomerNotFound;
    END IF;
  EXCEPTION
    WHEN e_CustomerNotFound THEN
      DBMS_OUTPUT.PUT_LINE('Customer ID ' || p_CustomerID || ' not found.');
    WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('An error occurred: ' || SQLERRM);
  END UpdateCustomerDetails;
  FUNCTION GetCustomerBalance(
    p_CustomerID IN NUMBER
  ) RETURN NUMBER IS
    v_Balance NUMBER;
  BEGIN
    SELECT Balance INTO v_Balance
    FROM Customers
    WHERE CustomerID = p_CustomerID;
    RETURN v_Balance;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('Customer ID ' || p_CustomerID || ' not found.');
      RETURN NULL;
    WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('An error occurred: ' || SQLERRM);
      RETURN NULL;
  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.

```
CREATE OR REPLACE PACKAGE EmployeeManagement AS
  PROCEDURE HireEmployee(
    p_EmployeeID IN NUMBER,
    p_Name IN VARCHAR2,
    p_Position IN VARCHAR2,
    p_Salary IN NUMBER,
    p_Department IN VARCHAR2,
    p_HireDate IN DATE
  );
  PROCEDURE UpdateEmployeeDetails(
    p_EmployeeID IN NUMBER,
    p_Name IN VARCHAR2,
    p_Position IN VARCHAR2,
    p_Salary IN NUMBER,
    p_Department IN VARCHAR2,
    p_HireDate IN DATE
  );
  FUNCTION CalculateAnnualSalary(
    p_EmployeeID IN NUMBER
  ) RETURN NUMBER;
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.

```
CREATE OR REPLACE PACKAGE BODY AccountOperations AS

  -- Procedure to open a new account
  PROCEDURE OpenAccount(
    p_AccountID IN NUMBER,
    p_CustomerID IN NUMBER,
    p_AccountType IN VARCHAR2,
    p_Balance IN NUMBER
  ) IS
    -- Exception for duplicate account ID
    e_DuplicateAccountID EXCEPTION;
  BEGIN
    -- Attempt to insert the new account
    INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance,
LastModified)
    VALUES (p_AccountID, p_CustomerID, p_AccountType, p_Balance, SYSDATE);
```

```
  EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
      RAISE e_DuplicateAccountID;
    WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('An error occurred: ' || SQLERRM);
  END OpenAccount;
  PROCEDURE CloseAccount(
    p_AccountID IN NUMBER
  ) IS
    e_AccountNotFound EXCEPTION;
  BEGIN
    DELETE FROM Accounts
    WHERE AccountID = p_AccountID;
    IF SQL%ROWCOUNT = 0 THEN
      RAISE e_AccountNotFound;
    END IF;
  EXCEPTION
    WHEN e_AccountNotFound THEN
      DBMS_OUTPUT.PUT_LINE('Account ID ' || p_AccountID || ' not found.');
    WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('An error occurred: ' || 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 v_TotalBalance;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('Customer ID ' || p_CustomerID || ' not found.');
      RETURN NULL;
    WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('An error occurred: ' || SQLERRM);
      RETURN NULL;
  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'));
```