## 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.
- **Answer:**

```
BEGIN
  FOR cust IN (SELECT CustomerID, (FLOOR(MONTHS_BETWEEN(SYSDATE, DOB) /
12)) AS Age
           FROM Customers)
  LOOP
    IF cust.Age > 60 THEN
      UPDATE Loans
      SET InterestRate = InterestRate - 1
      WHERE CustomerID = cust.CustomerID;
    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.
- **Answer:**

```
ALTER TABLE Customers
ADD IsVIP BOOLEAN;



BEGIN
  FOR cust IN (SELECT CustomerID, Balance
           FROM Customers)
  LOOP
    IF cust.Balance > 10000 THEN
      UPDATE Customers
      SET IsVIP = TRUE
      WHERE CustomerID = cust.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.
- **Answer:**

```
BEGIN
  FOR loan IN (SELECT LoanID, CustomerID, EndDate
           FROM Loans
           WHERE EndDate BETWEEN SYSDATE AND SYSDATE + 30)
  LOOP
    DECLARE
      v_customer_name VARCHAR2(100);
    BEGIN
      SELECT Name
      INTO v_customer_name
      FROM Customers
      WHERE CustomerID = loan.CustomerID;

      DBMS_OUTPUT.PUT_LINE('Reminder: Dear ' || v_customer_name || ', your loan (Loan ID:
' || loan.LoanID || ') is due on ' || TO_CHAR(loan.EndDate, 'YYYY-MM-DD'));
    END;
  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.
- **Answer:**

```
CREATE OR REPLACE PROCEDURE SafeTransferFunds(
    p_from_account IN NUMBER,
    p_to_account IN NUMBER,
    p_amount IN NUMBER
) IS
    v_from_balance NUMBER;
    v_to_balance NUMBER;
BEGIN
    -- Check current balance of the from account
    SELECT Balance INTO v_from_balance
    FROM Accounts
    WHERE AccountID = p_from_account;

    -- Check current balance of the to account
    SELECT Balance INTO v_to_balance
    FROM Accounts
    WHERE AccountID = p_to_account;

    -- Ensure sufficient funds
    IF v_from_balance < p_amount THEN
        RAISE_APPLICATION_ERROR(-20001, 'Insufficient funds in the source account.');
    END IF;

    -- Perform the transfer
    UPDATE Accounts
    SET Balance = Balance - p_amount
    WHERE AccountID = p_from_account;

    UPDATE Accounts
    SET Balance = Balance + p_amount
    WHERE AccountID = p_to_account;

    -- Commit the transaction
    COMMIT;

EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('One of the account IDs does not exist.');
        ROLLBACK;
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
        ROLLBACK;
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.
- **Answer:**

```
CREATE OR REPLACE PROCEDURE UpdateSalary(
  p_employee_id IN NUMBER,
  p_percentage IN NUMBER
) IS
  v_current_salary NUMBER;
BEGIN
  -- Fetch current salary of the employee
  SELECT Salary INTO v_current_salary
  FROM Employees
  WHERE EmployeeID = p_employee_id;

  -- Update the salary
  UPDATE Employees
  SET Salary = Salary * (1 + p_percentage / 100)
  WHERE EmployeeID = p_employee_id;

  -- Commit the transaction
  COMMIT;

EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('Employee ID does not exist.');
    ROLLBACK;
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
    ROLLBACK;
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.
- **Answer:**

```
CREATE OR REPLACE PROCEDURE AddNewCustomer(
  p_customer_id IN NUMBER,
  p_name IN VARCHAR2,
  p_dob IN DATE,
  p_balance IN NUMBER
) IS
BEGIN
  -- Attempt to insert a new customer
  INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)
  VALUES (p_customer_id, p_name, p_dob, p_balance, SYSDATE);

  -- Commit the transaction
  COMMIT;

EXCEPTION
  WHEN DUP_VAL_ON_INDEX THEN
    DBMS_OUTPUT.PUT_LINE('Customer ID already exists.');
    ROLLBACK;
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
    ROLLBACK;
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.
- **Answer:**

```
CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS
BEGIN
  -- Update the balance of all savings accounts by applying 1% interest
  UPDATE Accounts
  SET Balance = Balance * 1.01
  WHERE AccountType = 'Savings';

  -- Commit the transaction
  COMMIT;
```

```
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.
- **Answer:**

```
CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(
   p_department IN VARCHAR2,
   p_bonus_percentage IN NUMBER
) IS
BEGIN
   -- Update the salary of employees in the specified department
   UPDATE Employees
   SET Salary = Salary * (1 + p_bonus_percentage / 100)
   WHERE Department = p_department;

   -- Commit the transaction
   COMMIT;
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.
- **Answer:**

```
CREATE OR REPLACE PROCEDURE TransferFunds(
   p_from_account IN NUMBER,
   p_to_account IN NUMBER,
   p_amount IN NUMBER
) IS
   v_from_balance NUMBER;
BEGIN
   -- Check current balance of the from account
   SELECT Balance INTO v_from_balance
   FROM Accounts
   WHERE AccountID = p_from_account;

   -- Ensure sufficient funds
   IF v_from_balance < p_amount THEN
      RAISE_APPLICATION_ERROR(-20001, 'Insufficient funds in the source account.');
   END IF;

   -- Perform the transfer
   UPDATE Accounts
   SET Balance = Balance - p_amount
   WHERE AccountID = p_from_account;

   UPDATE Accounts
   SET Balance = Balance + p_amount
   WHERE AccountID = p_to_account;

   -- Commit the transaction
   COMMIT;

EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('One of the account IDs does not exist.');
      ROLLBACK;
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
      ROLLBACK;
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.
- **Answer:**

```
CREATE OR REPLACE FUNCTION CalculateAge(
   p_dob IN DATE
) RETURN NUMBER IS
   v_age NUMBER;
BEGIN
   -- Calculate the age based on the date of birth
   v_age := FLOOR(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.
- **Answer:**

```
CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(
   p_loan_amount IN NUMBER,
   p_interest_rate IN NUMBER,  -- Annual interest rate (e.g., 5 for 5%)
   p_duration_years IN NUMBER
) RETURN NUMBER IS
   v_monthly_installment NUMBER;
   v_monthly_rate NUMBER;
   v_num_payments NUMBER;
BEGIN
   -- Convert annual interest rate to monthly and calculate number of payments
   v_monthly_rate := p_interest_rate / 100 / 12;
   v_num_payments := p_duration_years * 12;

   -- Calculate monthly installment using the formula
   v_monthly_installment := (p_loan_amount * v_monthly_rate) /
                    (1 - POWER(1 + v_monthly_rate, -v_num_payments));

   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.
- **Answer:**

```
CREATE OR REPLACE FUNCTION HasSufficientBalance(
   p_account_id IN NUMBER,
   p_amount IN NUMBER
) RETURN BOOLEAN IS
   v_balance NUMBER;
BEGIN
   -- Fetch the balance of the account
   SELECT Balance INTO v_balance
   FROM Accounts
   WHERE AccountID = p_account_id;

   -- Return true if balance is sufficient, false otherwise
   RETURN v_balance >= p_amount;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      RETURN FALSE;  -- Account not found, consider insufficient balance
   WHEN OTHERS THEN
      RETURN FALSE;  -- Handle any other exceptions and consider insufficient balance
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.
- **Answer:**

```
CREATE OR REPLACE TRIGGER UpdateCustomerLastModified
BEFORE UPDATE ON Customers
FOR EACH ROW
BEGIN
   :NEW.LastModified := SYSDATE;
END UpdateCustomerLastModified;
/
```

**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.
- **Answer:**

```
CREATE TABLE AuditLog (
    AuditID NUMBER PRIMARY KEY,
    TransactionID NUMBER,
    ChangeDate DATE,
    ActionType VARCHAR2(50),
    OldAmount NUMBER,
    NewAmount NUMBER
);



CREATE OR REPLACE TRIGGER LogTransaction
AFTER INSERT ON Transactions
FOR EACH ROW
DECLARE
    v_audit_id NUMBER;
BEGIN
    -- Generate a unique ID for the audit log
    SELECT AuditLog_SEQ.NEXTVAL INTO v_audit_id FROM dual;

    -- Insert record into AuditLog table
    INSERT INTO AuditLog (
        AuditID, TransactionID, ChangeDate, ActionType, OldAmount, NewAmount
    ) VALUES (
        v_audit_id, :NEW.TransactionID, SYSDATE, 'INSERT', NULL, :NEW.Amount
    );
END LogTransaction;
/
```

**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.
- **Answer:**

```
CREATE OR REPLACE TRIGGER CheckTransactionRules
BEFORE INSERT ON Transactions
FOR EACH ROW
DECLARE
    v_balance NUMBER;
BEGIN
    -- Check if the transaction is a withdrawal
    IF :NEW.TransactionType = 'Withdrawal' THEN
        -- Fetch the current balance of the account
        SELECT Balance INTO v_balance
        FROM Accounts
        WHERE AccountID = :NEW.AccountID;

        -- Ensure withdrawal does not exceed the balance
        IF :NEW.Amount > v_balance THEN
          RAISE_APPLICATION_ERROR(-20002, 'Insufficient funds for withdrawal.');
        END IF;

    -- Check if the transaction is a deposit
    ELSIF :NEW.TransactionType = 'Deposit' THEN
        -- Ensure deposit amount is positive
        IF :NEW.Amount <= 0 THEN
          RAISE_APPLICATION_ERROR(-20003, 'Deposit amount must be positive.');
        END IF;
    END IF;
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.
- **Answer;**

```
DECLARE
  CURSOR c_transactions 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_customer_name Customers.Name%TYPE;
  v_transaction_date Transactions.TransactionDate%TYPE;
  v_amount Transactions.Amount%TYPE;
  v_transaction_type Transactions.TransactionType%TYPE;
BEGIN
  FOR rec IN c_transactions LOOP
    v_customer_name := rec.Name;
```

```
        v_transaction_date := rec.TransactionDate;
        v_amount := rec.Amount;
        v_transaction_type := rec.TransactionType;

        -- Print statement for each customer
        DBMS_OUTPUT.PUT_LINE('Customer: ' || v_customer_name);
        DBMS_OUTPUT.PUT_LINE('Date: ' || TO_CHAR(v_transaction_date, 'YYYY-MM-DD'));
        DBMS_OUTPUT.PUT_LINE('Amount: ' || v_amount);
        DBMS_OUTPUT.PUT_LINE('Transaction Type: ' || v_transaction_type);
        DBMS_OUTPUT.PUT_LINE('-----------------------------');
    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.
- **Answer:**

```
DECLARE
    CURSOR c_accounts IS
        SELECT AccountID, Balance
        FROM Accounts;

    v_fee NUMBER := 50;  -- Example annual fee amount
    v_balance Accounts.Balance%TYPE;
BEGIN
    FOR rec IN c_accounts LOOP
        v_balance := rec.Balance;

        -- Deduct the annual fee
        UPDATE Accounts
        SET Balance = v_balance - v_fee
        WHERE AccountID = rec.AccountID;

        DBMS_OUTPUT.PUT_LINE('Account ID: ' || rec.AccountID);
        DBMS_OUTPUT.PUT_LINE('Balance after fee: ' || (v_balance - v_fee));
    END LOOP;

    -- 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.
- **Answer:**

```
DECLARE
  CURSOR c_loans IS
    SELECT LoanID, InterestRate
    FROM Loans;

  v_new_interest_rate NUMBER := 6;  -- Example new interest rate (e.g., 6%)
BEGIN
  FOR rec IN c_loans LOOP
    -- Update the interest rate for each loan
    UPDATE Loans
    SET InterestRate = v_new_interest_rate
    WHERE LoanID = rec.LoanID;

    DBMS_OUTPUT.PUT_LINE('Loan ID: ' || rec.LoanID);
    DBMS_OUTPUT.PUT_LINE('New Interest Rate: ' || v_new_interest_rate);
  END LOOP;

  -- 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.
- **Answer:**

```
CREATE OR REPLACE PACKAGE CustomerManagement AS
  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 AS
    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 is not found
    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.
- **Answer:**

```
CREATE OR REPLACE PACKAGE EmployeeManagement AS
    PROCEDURE HireNewEmployee(
        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 AS
  PROCEDURE HireNewEmployee(
    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 HireNewEmployee;

  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;  -- Annual salary
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      RETURN NULL;  -- Handle case where employee is not found
  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.
- **Answer:**

```
CREATE OR REPLACE PACKAGE AccountOperations AS
   PROCEDURE OpenNewAccount(
      p_account_id IN NUMBER,
      p_customer_id IN NUMBER,
      p_account_type IN VARCHAR2,
      p_balance IN NUMBER
   );

   PROCEDURE CloseAccount(
      p_account_id IN NUMBER
   );

   FUNCTION GetTotalBalance(
      p_customer_id IN NUMBER
   ) RETURN NUMBER;
END AccountOperations;
/


CREATE OR REPLACE PACKAGE BODY AccountOperations AS
   PROCEDURE OpenNewAccount(
      p_account_id IN NUMBER,
      p_customer_id IN NUMBER,
      p_account_type IN VARCHAR2,
      p_balance IN NUMBER
   ) IS
   BEGIN
      INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified)
      VALUES (p_account_id, p_customer_id, p_account_type, p_balance, SYSDATE);
      COMMIT;
   END OpenNewAccount;

   PROCEDURE CloseAccount(
      p_account_id IN NUMBER
   ) IS
```

```
    BEGIN
        DELETE FROM Accounts
        WHERE AccountID = p_account_id;
        COMMIT;
    END CloseAccount;

    FUNCTION GetTotalBalance(
        p_customer_id IN NUMBER
    ) RETURN NUMBER IS
        v_total_balance NUMBER;
    BEGIN
        SELECT SUM(Balance) INTO v_total_balance
        FROM Accounts
        WHERE CustomerID = p_customer_id;
        RETURN v_total_balance;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            RETURN 0;  -- Handle case where no accounts are found
    END GetTotalBalance;
END AccountOperations;
/
```