Exercise 1: Control Structures

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

```
DECLARE
  CURSOR customer_cursor IS
    SELECT CustomerID, DOB FROM Customers;

  v_CustomerID Customers.CustomerID%TYPE;
  v_DOB Customers.DOB%TYPE;
  v_Age NUMBER;
BEGIN
  FOR customer_record IN customer_cursor LOOP
    v_CustomerID := customer_record.CustomerID;
    v_DOB := customer_record.DOB;

    -- Calculate age
    v_Age := FLOOR((SYSDATE - v_DOB) / 365.25);

    IF v_Age > 60 THEN
      -- Apply 1% discount to loan interest rates
      UPDATE Loans
      SET InterestRate = InterestRate * 0.99
      WHERE CustomerID = v_CustomerID;
    END IF;
  END LOOP;

  COMMIT;
END;
```

Scenario 2: Promoting Customers to VIP Status Based on Balance

```
ALTER TABLE Customers ADD (IsVIP VARCHAR2(3));

DECLARE
  CURSOR customer_cursor IS
    SELECT CustomerID, Balance FROM Customers;

  v_CustomerID Customers.CustomerID%TYPE;
  v_Balance Customers.Balance%TYPE;
BEGIN
  FOR customer_record IN customer_cursor LOOP
    v_CustomerID := customer_record.CustomerID;
```

```
        v_Balance := customer_record.Balance;

        IF v_Balance > 10000 THEN
            -- Set IsVIP flag to TRUE
            UPDATE Customers
            SET IsVIP = 'TRUE'
            WHERE CustomerID = v_CustomerID;
        END IF;
    END LOOP;

    COMMIT;
END;
```

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

```
DECLARE
    CURSOR loan_cursor IS
        SELECT LoanID, CustomerID, EndDate FROM Loans
        WHERE EndDate BETWEEN SYSDATE AND SYSDATE + 30;

    v_LoanID Loans.LoanID%TYPE;
    v_CustomerID Loans.CustomerID%TYPE;
    v_EndDate Loans.EndDate%TYPE;
    v_CustomerName Customers.Name%TYPE;
BEGIN
    FOR loan_record IN loan_cursor LOOP
        v_LoanID := loan_record.LoanID;
        v_CustomerID := loan_record.CustomerID;
        v_EndDate := loan_record.EndDate;

        -- Fetch customer's name
        SELECT Name INTO v_CustomerName
        FROM Customers
        WHERE CustomerID = v_CustomerID;

        -- Print reminder message
        DBMS_OUTPUT.PUT_LINE('Reminder: Customer ' || v_CustomerName ||
                             ' (CustomerID: ' || v_CustomerID ||
                             ') has a loan (LoanID: ' || v_LoanID ||
                             ') due on ' || TO_CHAR(v_EndDate, 'DD-MON-YYYY') || '.');
    END LOOP;
END;
```

Exercise 2: Error Handling

Scenario 1: Handling Exceptions During Fund Transfers Between Accounts

```
CREATE OR REPLACE PROCEDURE SafeTransferFunds (
   p_FromAccountID IN NUMBER,
   p_ToAccountID IN NUMBER,
   p_Amount IN NUMBER
) IS
   v_FromBalance NUMBER;
   v_ToBalance NUMBER;
BEGIN
   -- Check balance of from account
   SELECT Balance INTO v_FromBalance
   FROM Accounts
   WHERE AccountID = p_FromAccountID
   FOR UPDATE;

   IF v_FromBalance < p_Amount THEN
      RAISE_APPLICATION_ERROR(-20001, 'Insufficient funds in the source account.');
   END IF;

   -- Deduct amount from source account
   UPDATE Accounts
   SET Balance = Balance - p_Amount,
      LastModified = SYSDATE
   WHERE AccountID = p_FromAccountID;

   -- Add amount to destination account
   UPDATE Accounts
   SET Balance = Balance + p_Amount,
      LastModified = SYSDATE
   WHERE AccountID = p_ToAccountID;

   COMMIT;

   EXCEPTION
      WHEN OTHERS THEN
         ROLLBACK;
         -- Log the error message
         INSERT INTO ErrorLog (ErrorMessage, ErrorDate)
         VALUES (SQLERRM, SYSDATE);
         RAISE;
END;
```

Scenario 2: Managing Errors When Updating Employee Salaries

```
CREATE OR REPLACE PROCEDURE UpdateSalary (
   p_EmployeeID IN NUMBER,
   p_Percentage IN NUMBER
) IS
   v_Salary NUMBER;
BEGIN
   BEGIN
      -- Fetch current salary
      SELECT Salary INTO v_Salary
      FROM Employees
      WHERE EmployeeID = p_EmployeeID
      FOR UPDATE;

      -- Update salary
      UPDATE Employees
      SET Salary = Salary * (1 + p_Percentage / 100),
         LastModified = SYSDATE
      WHERE EmployeeID = p_EmployeeID;

      COMMIT;
   EXCEPTION
      WHEN NO_DATA_FOUND THEN
         -- Log the error message
         INSERT INTO ErrorLog (ErrorMessage, ErrorDate)
         VALUES ('Employee ID ' || p_EmployeeID || ' not found.', SYSDATE);
      WHEN OTHERS THEN
         ROLLBACK;
         -- Log the error message
         INSERT INTO ErrorLog (ErrorMessage, ErrorDate)
         VALUES (SQLERRM, SYSDATE);
         RAISE;
   END;
END;
/
```

Scenario 3: Ensuring Data Integrity When Adding a New Customer

```
CREATE OR REPLACE PROCEDURE AddNewCustomer (
   p_CustomerID IN NUMBER,
   p_Name IN VARCHAR2,
   p_DOB IN DATE,
    p_Balance IN NUMBER
) IS
BEGIN
    BEGIN
        -- Insert new customer
        INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)
        VALUES (p_CustomerID, p_Name, p_DOB, p_Balance, SYSDATE);

        COMMIT;
    EXCEPTION
        WHEN DUP_VAL_ON_INDEX THEN
            -- Log the error message
            INSERT INTO ErrorLog (ErrorMessage, ErrorDate)
            VALUES ('Customer ID ' || p_CustomerID || ' already exists.', SYSDATE);
        WHEN OTHERS THEN
            ROLLBACK;
            -- Log the error message
            INSERT INTO ErrorLog (ErrorMessage, ErrorDate)
            VALUES (SQLERRM, SYSDATE);
            RAISE;
    END;
END;
/
```

ErrorLog Table

```
CREATE TABLE ErrorLog (
    ErrorID NUMBER PRIMARY KEY,
    ErrorMessage VARCHAR2(4000),
    ErrorDate DATE
);
```

Exercise 3: Stored Procedures

Scenario 1: Processing Monthly Interest for All Savings Accounts

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

```
   v_AccountID Accounts.AccountID%TYPE;
   v_Balance Accounts.Balance%TYPE;
   v_InterestRate CONSTANT NUMBER := 0.01;
BEGIN
   FOR account_record IN savings_accounts_cursor LOOP
      v_AccountID := account_record.AccountID;
      v_Balance := account_record.Balance;

      -- Calculate new balance with interest
      v_Balance := v_Balance * (1 + v_InterestRate);

      -- Update account balance
      UPDATE Accounts
      SET Balance = v_Balance,
         LastModified = SYSDATE
      WHERE AccountID = v_AccountID;
   END LOOP;

   COMMIT;
END;
/
```

<u>**Scenario 2: Implementing a Bonus Scheme for Employees**</u>

```
CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (
   p_Department IN VARCHAR2,
   p_BonusPercentage IN NUMBER
) IS
BEGIN
   -- Update salary by adding bonus percentage
   UPDATE Employees
   SET Salary = Salary * (1 + p_BonusPercentage / 100),
      LastModified = SYSDATE
   WHERE Department = p_Department;

   COMMIT;
END;
/
```

Scenario 3: Transferring Funds Between Customer Accounts

```
CREATE OR REPLACE PROCEDURE TransferFunds (
   p_FromAccountID IN NUMBER,
   p_ToAccountID IN NUMBER,
   p_Amount IN NUMBER
) IS
   v_FromBalance NUMBER;
   v_ToBalance NUMBER;
BEGIN
   -- Check balance of the source account
   SELECT Balance INTO v_FromBalance
   FROM Accounts
   WHERE AccountID = p_FromAccountID
   FOR UPDATE;

   IF v_FromBalance < p_Amount THEN
      RAISE_APPLICATION_ERROR(-20001, 'Insufficient funds in the source account.');
   END IF;

   -- Deduct amount from source account
   UPDATE Accounts
   SET Balance = Balance - p_Amount,
      LastModified = SYSDATE
   WHERE AccountID = p_FromAccountID;

   -- Add amount to destination account
   UPDATE Accounts
   SET Balance = Balance + p_Amount,
      LastModified = SYSDATE
   WHERE AccountID = p_ToAccountID;

   COMMIT;

EXCEPTION
   WHEN OTHERS THEN
      ROLLBACK;
      -- Log the error message
      INSERT INTO ErrorLog (ErrorMessage, ErrorDate)
      VALUES (SQLERRM, SYSDATE);
      RAISE;
END;
/
```

Exercise 4: Functions

Scenario 1: Calculate the Age of Customers for Eligibility Checks

```
CREATE OR REPLACE FUNCTION CalculateAge (
   p_DOB DATE
) RETURN NUMBER IS
   v_Age NUMBER;
BEGIN
   -- Calculate age
   v_Age := FLOOR((SYSDATE - p_DOB) / 365.25);
   RETURN v_Age;
END;
/
```

Scenario 2: Compute the Monthly Installment for a Loan

```
CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment (
   p_LoanAmount NUMBER,
   p_InterestRate NUMBER,
   p_LoanDurationYears NUMBER
) RETURN NUMBER IS
   v_MonthlyRate NUMBER;
   v_NumberOfPayments NUMBER;
   v_MonthlyInstallment NUMBER;
BEGIN
   -- Convert annual interest rate to monthly and loan duration to number of months
   v_MonthlyRate := p_InterestRate / 12 / 100;
   v_NumberOfPayments := p_LoanDurationYears * 12;

   -- Calculate monthly installment using the formula for an annuity
   v_MonthlyInstallment := p_LoanAmount * v_MonthlyRate / (1 - POWER(1 +
v_MonthlyRate, -v_NumberOfPayments));

   RETURN v_MonthlyInstallment;
END;
/
```

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

```
CREATE OR REPLACE FUNCTION HasSufficientBalance (
   p_AccountID NUMBER,
   p_Amount NUMBER
) RETURN BOOLEAN IS
   v_Balance NUMBER;
BEGIN
   -- Fetch account balance
   SELECT Balance INTO v_Balance
   FROM Accounts
   WHERE AccountID = p_AccountID;

   -- Check if balance is sufficient
   IF v_Balance >= p_Amount THEN
      RETURN TRUE;
   ELSE
      RETURN FALSE;
   END IF;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      RETURN FALSE;
END;
/
```

Exercise 5: Triggers

Scenario 1: Automatically Update 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;
/
```

Scenario 2: Maintain an Audit Log for All Transactions

```
CREATE TABLE AuditLog (
   AuditID NUMBER PRIMARY KEY,
   TransactionID NUMBER,
   AccountID NUMBER,
   TransactionDate DATE,
   Amount NUMBER,
   TransactionType VARCHAR2(10),
   LogDate DATE
);
```

Now, create the trigger:

```
CREATE OR REPLACE TRIGGER LogTransaction
AFTER INSERT ON Transactions
FOR EACH ROW
BEGIN
   INSERT INTO AuditLog (
      AuditID, TransactionID, AccountID, TransactionDate, Amount, TransactionType, LogDate
   ) VALUES (
      AuditLog_seq.NEXTVAL, -- Assuming a sequence named AuditLog_seq exists
      :NEW.TransactionID,
      :NEW.AccountID,
      :NEW.TransactionDate,
      :NEW.Amount,
      :NEW.TransactionType,
      SYSDATE
   );
END;
/
```

Scenario 3: Enforce Business Rules on Deposits and Withdrawals

```
CREATE OR REPLACE TRIGGER CheckTransactionRules
BEFORE INSERT ON Transactions
FOR EACH ROW
DECLARE
   v_Balance NUMBER;
BEGIN
   -- Fetch the current balance of the account
   SELECT Balance INTO v_Balance
   FROM Accounts
   WHERE AccountID = :NEW.AccountID
```

```
    FOR UPDATE;

    -- Check for deposit
    IF :NEW.TransactionType = 'Deposit' THEN
        IF :NEW.Amount <= 0 THEN
            RAISE_APPLICATION_ERROR(-20002, 'Deposit amount must be positive.');
        END IF;
    ELSIF :NEW.TransactionType = 'Withdrawal' THEN
        -- Check for withdrawal
        IF :NEW.Amount <= 0 THEN
            RAISE_APPLICATION_ERROR(-20003, 'Withdrawal amount must be positive.');
        ELSIF v_Balance < :NEW.Amount THEN
            RAISE_APPLICATION_ERROR(-20004, 'Insufficient funds for withdrawal.');
        END IF;
    ELSE
        RAISE_APPLICATION_ERROR(-20005, 'Invalid transaction type.');
    END IF;
END;
/
```

Exercise 6: Cursors

Scenario 1: Generate Monthly Statements for All Customers

```
DECLARE
    CURSOR transactions_cursor IS
        SELECT CustomerID, AccountID, TransactionDate, Amount, TransactionType
        FROM Transactions
        WHERE TransactionDate BETWEEN TRUNC(SYSDATE, 'MM') AND LAST_DAY(SYSDATE);

    v_CustomerID Transactions.CustomerID%TYPE;
    v_AccountID Transactions.AccountID%TYPE;
    v_TransactionDate Transactions.TransactionDate%TYPE;
    v_Amount Transactions.Amount%TYPE;
    v_TransactionType Transactions.TransactionType%TYPE;
BEGIN
    OPEN transactions_cursor;
    LOOP
        FETCH transactions_cursor INTO v_CustomerID, v_AccountID, v_TransactionDate,
v_Amount, v_TransactionType;
        EXIT WHEN transactions_cursor%NOTFOUND;

        DBMS_OUTPUT.PUT_LINE('Customer ID: ' || v_CustomerID ||
                        ', Account ID: ' || v_AccountID ||
                        ', Date: ' || TO_CHAR(v_TransactionDate, 'DD-MON-YYYY') ||
                        ', Amount: ' || v_Amount ||
                        ', Type: ' || v_TransactionType);
    END LOOP;
    CLOSE transactions_cursor;
END;
/
```

Scenario 2: Apply Annual Fee to All Accounts

```
DECLARE
    CURSOR accounts_cursor IS
        SELECT AccountID, Balance
        FROM Accounts
        FOR UPDATE;

    v_AccountID Accounts.AccountID%TYPE;
    v_Balance Accounts.Balance%TYPE;
    v_AnnualFee CONSTANT NUMBER := 50; -- Annual fee amount
BEGIN
    OPEN accounts_cursor;
    LOOP
        FETCH accounts_cursor INTO v_AccountID, v_Balance;
        EXIT WHEN accounts_cursor%NOTFOUND;

        -- Deduct annual fee
        v_Balance := v_Balance - v_AnnualFee;

        -- Update account balance
        UPDATE Accounts
        SET Balance = v_Balance,
            LastModified = SYSDATE
        WHERE CURRENT OF accounts_cursor;
    END LOOP;
    CLOSE accounts_cursor;

    COMMIT;
END;
/
```

Scenario 3: Update the Interest Rate for All Loans Based on a New Policy

```
DECLARE
   CURSOR loans_cursor IS
      SELECT LoanID, InterestRate
      FROM Loans
      FOR UPDATE;

   v_LoanID Loans.LoanID%TYPE;
   v_InterestRate Loans.InterestRate%TYPE;
   v_NewRateAdjustment CONSTANT NUMBER := 0.5; -- Example adjustment, could be an
increase by 0.5%
BEGIN
   OPEN loans_cursor;
   LOOP
      FETCH loans_cursor INTO v_LoanID, v_InterestRate;
      EXIT WHEN loans_cursor%NOTFOUND;

      -- Update interest rate based on the new policy
      v_InterestRate := v_InterestRate + v_NewRateAdjustment;

      -- Update loan interest rate
      UPDATE Loans
      SET InterestRate = v_InterestRate,
         LastModified = SYSDATE
      WHERE CURRENT OF loans_cursor;
   END LOOP;
   CLOSE loans_cursor;

   COMMIT;
END;
/
```

Exercise 7: Packages

Scenario 1: Grouping All Customer-Related Procedures and Functions into a Package

```
-- Package Specification
CREATE OR REPLACE PACKAGE CustomerManagement IS
   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
    );

    FUNCTION GetCustomerBalance(
        p_CustomerID IN NUMBER
    ) RETURN NUMBER;
END CustomerManagement;
/

-- Package Body
CREATE OR REPLACE PACKAGE BODY CustomerManagement IS
    PROCEDURE AddNewCustomer(
        p_CustomerID 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_CustomerID, p_Name, p_DOB, p_Balance, SYSDATE);
    END AddNewCustomer;

    PROCEDURE UpdateCustomerDetails(
        p_CustomerID 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_CustomerID;
    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;
    END GetCustomerBalance;
END CustomerManagement;
/
```

<u>**Scenario 2: Creating a Package to Manage Employee Data**</u>

```
-- Package Specification
CREATE OR REPLACE PACKAGE EmployeeManagement IS
    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
    );

    FUNCTION CalculateAnnualSalary(
        p_EmployeeID IN NUMBER
    ) RETURN NUMBER;
END EmployeeManagement;
/

-- Package Body
CREATE OR REPLACE PACKAGE BODY EmployeeManagement IS
    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
    ) 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 UpdateEmployeeDetails(
        p_EmployeeID 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,
           LastModified = SYSDATE
        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: Grouping All Account-Related Operations into a Package

```
-- Package Specification
CREATE OR REPLACE PACKAGE AccountOperations IS
   PROCEDURE OpenAccount(
      p_AccountID IN NUMBER,
      p_CustomerID IN NUMBER,
      p_AccountType IN VARCHAR2,
      p_Balance IN NUMBER
   );

   PROCEDURE CloseAccount(
      p_AccountID IN NUMBER
   );

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

-- Package Body
CREATE OR REPLACE PACKAGE BODY AccountOperations IS
   PROCEDURE OpenAccount(
      p_AccountID IN NUMBER,
      p_CustomerID IN NUMBER,
      p_AccountType IN VARCHAR2,
      p_Balance IN 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 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 v_TotalBalance;
    END GetTotalBalance;
END AccountOperations;
/
```