Exercise 1: Control Structures

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

```
BEGIN
  FOR customer_rec IN (SELECT CustomerID, DOB FROM Customers) LOOP
    IF TRUNC(MONTHS_BETWEEN(SYSDATE, customer_rec.DOB) / 12) > 60 THEN
      UPDATE Loans
      SET InterestRate = InterestRate - 1
      WHERE CustomerID = customer_rec.CustomerID;
    END IF;
  END LOOP;
  COMMIT;
END;
```

Scenario 2: Promote Customers to VIP Status Based on Balance

```
BEGIN
    FOR customer_rec IN (SELECT CustomerID, Balance FROM Customers)
LOOP
        IF customer_rec.Balance > 10000 THEN
            UPDATE Customers
            SET IsVIP = TRUE
            WHERE CustomerID = customer_rec.CustomerID;
        END IF;
    END LOOP;
    COMMIT;
END;
```

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

```
BEGIN
    FOR loan_rec IN (SELECT LoanID, CustomerID FROM Loans
                     WHERE EndDate BETWEEN SYSDATE AND SYSDATE + 30)
LOOP
        DBMS_OUTPUT.PUT_LINE('Reminder: Loan ' || loan_rec.LoanID ||
                            ' for Customer ' || loan_rec.CustomerID
|| ' is due within 30 days.');
    END LOOP;
END;
```

Exercise 2: Error Handling

Scenario 1: Handle Exceptions During Fund Transfers Between Accounts

```
CREATE OR REPLACE PROCEDURE SafeTransferFunds(p_SourceAccountID IN
NUMBER, p_TargetAccountID IN NUMBER, p_Amount IN NUMBER) IS
    insufficient_funds EXCEPTION;
    BEGIN
        UPDATE Accounts
        SET Balance = Balance - p_Amount
        WHERE AccountID = p_SourceAccountID AND Balance >= p_Amount;

        IF SQL%ROWCOUNT = 0 THEN
            RAISE insufficient_funds;
        END IF;

        UPDATE Accounts
        SET Balance = Balance + p_Amount
        WHERE AccountID = p_TargetAccountID;

        COMMIT;

    EXCEPTION
        WHEN insufficient_funds THEN
            ROLLBACK;
            DBMS_OUTPUT.PUT_LINE('Error: Insufficient funds in
source account.');
        WHEN OTHERS THEN
            ROLLBACK;
            DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
END SafeTransferFunds;
```

Scenario 2: Manage Errors When Updating Employee Salaries

```
CREATE OR REPLACE PROCEDURE UpdateSalary(p_EmployeeID IN NUMBER,
p_PercentIncrease IN NUMBER) IS
    employee_not_found EXCEPTION;
BEGIN
    UPDATE Employees
    SET Salary = Salary + (Salary * p_PercentIncrease / 100)
    WHERE EmployeeID = p_EmployeeID;

    IF SQL%ROWCOUNT = 0 THEN
        RAISE employee_not_found;
    END IF;

    COMMIT;

EXCEPTION
    WHEN employee_not_found THEN
        DBMS_OUTPUT.PUT_LINE('Error: Employee ID ' || p_EmployeeID
|| ' does not exist.');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
END UpdateSalary;
```

Scenario 3: Ensure 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
    customer_exists EXCEPTION;
BEGIN
    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
        RAISE customer_exists;
    WHEN customer_exists THEN
        DBMS_OUTPUT.PUT_LINE('Error: Customer with ID ' ||
p_CustomerID || ' already exists.');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
END AddNewCustomer;
```

Exercise 3: Stored Procedures

Scenario 1: Process Monthly Interest for All Savings Accounts

```
CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS
BEGIN
    UPDATE Accounts
    SET Balance = Balance + (Balance * 0.01)
    WHERE AccountType = 'Savings';

    COMMIT;
END ProcessMonthlyInterest;
```

Scenario 2: Implement a Bonus Scheme for Employees

```
CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(p_Department IN
VARCHAR2, p_BonusPercent IN NUMBER) IS
BEGIN
    UPDATE Employees
    SET Salary = Salary + (Salary * p_BonusPercent / 100)
    WHERE Department = p_Department;

    COMMIT;
END UpdateEmployeeBonus;
```

Scenario 3: Transfer Funds Between Accounts

```
CREATE OR REPLACE PROCEDURE TransferFunds(p_SourceAccountID IN
NUMBER, p_TargetAccountID IN NUMBER, p_Amount IN NUMBER) IS
    insufficient_funds EXCEPTION;
BEGIN
    UPDATE Accounts
    SET Balance = Balance - p_Amount
    WHERE AccountID = p_SourceAccountID AND Balance >= p_Amount;
    IF SQL%ROWCOUNT = 0 THEN
        RAISE insufficient_funds;
    END IF;

    UPDATE Accounts
    SET Balance = Balance + p_Amount
    WHERE AccountID = p_TargetAccountID;

    COMMIT;

EXCEPTION
    WHEN insufficient_funds THEN
        DBMS_OUTPUT.PUT_LINE('Error: Insufficient funds in source
account.');
        ROLLBACK;
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
        ROLLBACK;
END TransferFunds;
```

Exercise 4: Functions

Scenario 1: Calculate the Age of Customers

```
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: Compute the Monthly Installment for a Loan

```
CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(p_LoanAmount
IN NUMBER, p_InterestRate IN NUMBER, p_LoanDurationYears IN NUMBER)
RETURN NUMBER IS
    v_MonthlyInstallment NUMBER;
BEGIN
    v_MonthlyInstallment := p_LoanAmount * (p_InterestRate / 100) /
```

```
12 * POWER(1 + (p_InterestRate / 100) / 12, p_LoanDurationYears *
12) /
                            (POWER(1 + (p_InterestRate / 100) / 12,
p_LoanDurationYears * 12) - 1);
    RETURN v_MonthlyInstallment;
END CalculateMonthlyInstallment;
```

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

```
CREATE OR REPLACE FUNCTION HasSufficientBalance(p_AccountID IN
NUMBER, p_Amount IN NUMBER) RETURN BOOLEAN IS
    v_Balance NUMBER;
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;
END HasSufficientBalance;
```

Exercise 5: Triggers

Scenario 1: Update Last Modified Date When a Customer's Record is Updated

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

Scenario 2: Maintain an Audit Log for All Transactions

```
CREATE OR REPLACE TRIGGER LogTransaction
AFTER INSERT ON Transactions
```

```
FOR EACH ROW
BEGIN
    INSERT INTO AuditLog (TransactionID, AccountID, TransactionDate,
Amount, TransactionType)
    VALUES
(:NEW.TransactionID, :NEW.AccountID, :NEW.TransactionDate, :NEW.Amou
nt, :NEW.TransactionType);
END LogTransaction;
```

Scenario 3: Enforce Business Rules on Deposits and Withdrawals

```
CREATE OR REPLACE TRIGGER CheckTransactionRules
BEFORE INSERT ON Transactions
FOR EACH ROW
BEGIN
    IF :NEW.TransactionType = 'Withdrawal' AND :NEW.Amount > (SELECT
Balance FROM Accounts WHERE AccountID = :NEW.AccountID) THEN
        RAISE_APPLICATION_ERROR(-20001, 'Insufficient funds for
withdrawal.');
    ELSIF :NEW.TransactionType = 'Deposit' AND :NEW.Amount <= 0 THEN
        RAISE_APPLICATION_ERROR(-20002, 'Deposit amount must be
positive.');
    END IF;
END CheckTransactionRules;
```

Exercise 6: Cursors

Scenario 1: Generate Monthly Statements for All Customers

```
DECLARE
    CURSOR trans_cur IS
        SELECT t.AccountID, t.Amount, t.TransactionType,
t.TransactionDate
        FROM Transactions t
        WHERE t.TransactionDate BETWEEN TRUNC(SYSDATE, 'MM') AND
LAST_DAY(SYSDATE);
    trans_rec trans_cur%ROWTYPE;
BEGIN
    OPEN trans_cur;
    LOOP
        FETCH trans_cur INTO trans_rec;
        EXIT WHEN trans_cur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Account ID: ' || trans_rec.AccountID
|| ', Transaction: ' || trans_rec.TransactionType ||
                             ', Amount: ' || trans_rec.Amount || ',
Date: ' || trans_rec.TransactionDate);
    END LOOP;
    CLOSE trans_cur;
END;
```

Scenario 2: Apply Annual Fee to All Accounts

```
DECLARE
    CURSOR account_cur IS
        SELECT AccountID, Balance FROM Accounts;
    account_rec account_cur%ROWTYPE;
    v_AnnualFee CONSTANT NUMBER := 50;
BEGIN
    OPEN account_cur;
    LOOP
        FETCH account_cur INTO account_rec;
        EXIT WHEN account_cur%NOTFOUND;
        UPDATE Accounts
        SET Balance = Balance - v_AnnualFee
        WHERE AccountID = account_rec.AccountID;
    END LOOP;
    CLOSE account_cur;
    COMMIT;
END;
```

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

```
DECLARE
    CURSOR loan_cur IS
        SELECT LoanID, InterestRate FROM Loans;
    loan_rec loan_cur%ROWTYPE;
BEGIN
    OPEN loan_cur;
    LOOP
        FETCH loan_cur INTO loan_rec;
        EXIT WHEN loan_cur%NOTFOUND;
        UPDATE Loans
        SET InterestRate = loan_rec.InterestRate + 0.5
        WHERE LoanID = loan_rec.LoanID;
    END LOOP;
    CLOSE loan_cur;
    COMMIT;
END;
```

Exercise 7: Packages

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

```
CREATE OR REPLACE PACKAGE CustomerManagement AS
    PROCEDURE AddCustomer(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 AddCustomer(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);
        COMMIT;
    END AddCustomer;

    PROCEDURE UpdateCustomerDetails(p_CustomerID IN NUMBER, p_Name
IN VARCHAR2, p_DOB IN DATE, p_Balance IN NUMBER) IS
    BEGIN
        UPDATE Customers
        SET Name = p_Name, DOB = p_DOB, Balance = p_Balance,
LastModified = SYSDATE
        WHERE CustomerID = p_CustomerID;
        COMMIT;
    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;
```

Scenario 2: Manage Employee Data

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

CREATE OR REPLACE PACKAGE BODY 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) IS
    BEGIN
        INSERT INTO Employees (EmployeeID, Name, Position, Salary,
Department, HireDate)
        VALUES (p_EmployeeID, p_Name, p_Position, p_Salary,
p_Department, p_HireDate);
        COMMIT;
    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
        WHERE EmployeeID = p_EmployeeID;
        COMMIT;
    END UpdateEmployeeDetails;

    FUNCTION CalculateAnnualSalary(p_EmployeeID IN NUMBER) RETURN
NUMBER IS
        v_Salary NUMBER;
    BEGIN
        SELECT Salary INTO v_Salary FROM Employees WHERE EmployeeID
= p_EmployeeID;
        RETURN v_Salary * 12;
    END CalculateAnnualSalary;
END EmployeeManagement;
```

Scenario 3: Group All Account-Related Operations into a Package

```
CREATE OR REPLACE PACKAGE AccountOperations AS
    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;

CREATE OR REPLACE PACKAGE BODY AccountOperations AS
    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);
        COMMIT;
    END OpenAccount;

    PROCEDURE CloseAccount(p_AccountID IN NUMBER) IS
    BEGIN
        DELETE FROM Accounts WHERE AccountID = p_AccountID;
        COMMIT;
    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;
```