Exercise 1: Control Structures

Scenario 1: The bank wants to apply a discount to loan interest rates for customers above 60 years old.

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 EXTRACT(YEAR FROM SYSDATE) - EXTRACT(YEAR FROM c.DOB) > 60;

BEGIN
   FOR loan_record IN customer_cursor LOOP
      UPDATE Loans
      SET InterestRate = InterestRate - 1
      WHERE LoanID = loan_record.LoanID;

      DBMS_OUTPUT.PUT_LINE('Applied 1% discount to loan ID: ' || loan_record.LoanID);
   END LOOP;
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 CHAR(1));
DECLARE
   CURSOR customer_cursor IS
      SELECT CustomerID, Balance
```

```
        FROM Customers;

BEGIN
    FOR customer_record IN customer_cursor LOOP
        IF customer_record.Balance > 10000 THEN
            UPDATE Customers
            SET IsVIP = 'Y'
            WHERE CustomerID = customer_record.CustomerID;

            DBMS_OUTPUT.PUT_LINE('Promoted to VIP status for customer ID: ' ||
customer_record.CustomerID);
        ELSE
            UPDATE Customers
            SET IsVIP = 'N'
            WHERE CustomerID = customer_record.CustomerID;
        END IF;
    END LOOP;
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 l.LoanID, l.CustomerID, l.EndDate, c.Name
        FROM Loans l
        JOIN Customers c ON l.CustomerID = c.CustomerID
        WHERE l.EndDate BETWEEN SYSDATE AND SYSDATE + 30;

BEGIN
    FOR loan_record IN loan_cursor LOOP
```

```
        DBMS_OUTPUT.PUT_LINE('Reminder: Loan ID ' || loan_record.LoanID ||
                             ' for customer ' || loan_record.Name ||
                             ' is due on ' || loan_record.EndDate);
    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.

```
CREATE OR REPLACE PROCEDURE SafeTransferFunds (
  p_from_account IN NUMBER,
  p_to_account IN NUMBER,
  p_amount IN NUMBER
) AS
BEGIN
  BEGIN
    DECLARE
      v_balance NUMBER;
    BEGIN
      SELECT Balance INTO v_balance
      FROM Accounts
      WHERE AccountID = p_from_account;

      IF v_balance < p_amount THEN
        RAISE_APPLICATION_ERROR(-20001, 'Insufficient funds in account ' || p_from_account);
      END IF;
    END;
    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;

    EXCEPTION
        WHEN OTHERS THEN
            ROLLBACK;
            DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
    END;
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_employee_id IN NUMBER,
    p_percentage IN NUMBER
) AS
BEGIN
    BEGIN
        UPDATE Employees
        SET Salary = Salary * (1 + p_percentage / 100)
        WHERE EmployeeID = p_employee_id;

        IF SQL%ROWCOUNT = 0 THEN
            RAISE_APPLICATION_ERROR(-20002, 'Employee ID ' || p_employee_id || ' does not exist');
        END IF;

        COMMIT;
```

```
  EXCEPTION
    WHEN OTHERS THEN
      ROLLBACK;
      DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
  END;
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_customer_id IN NUMBER,
  p_name IN VARCHAR2,
  p_dob IN DATE,
  p_balance IN NUMBER
) AS
BEGIN
  BEGIN
    INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)
    VALUES (p_customer_id, p_name, p_dob, p_balance, SYSDATE);

    COMMIT;

  EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
      DBMS_OUTPUT.PUT_LINE('Error: Customer ID ' || p_customer_id || ' already exists');
    WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
```

```
        ROLLBACK;
    END;
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 AS
BEGIN
   UPDATE Accounts
   SET Balance = Balance * 1.01
   WHERE AccountType = 'Savings';

   COMMIT;

   DBMS_OUTPUT.PUT_LINE('Monthly interest applied to all savings accounts.');
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 VARCHAR2,
   p_bonus_percentage IN NUMBER
) AS
BEGIN
   UPDATE Employees
   SET Salary = Salary * (1 + p_bonus_percentage / 100)
   WHERE Department = p_department;
```

```
    COMMIT;

    DBMS_OUTPUT.PUT_LINE('Bonus applied to all employees in department: ' || p_department);
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_from_account IN NUMBER,
    p_to_account IN NUMBER,
    p_amount IN NUMBER
) AS
    v_balance NUMBER;
BEGIN
    SELECT Balance INTO v_balance
    FROM Accounts
    WHERE AccountID = p_from_account;

    IF v_balance < p_amount THEN
        RAISE_APPLICATION_ERROR(-20001, 'Insufficient funds in account ' || p_from_account);
    END IF;

    BEGIN
        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;

            DBMS_OUTPUT.PUT_LINE('Transfer of ' || p_amount || ' from account ' || p_from_account
|| ' to account ' || p_to_account || ' completed successfully.');

    EXCEPTION

        WHEN OTHERS THEN

            ROLLBACK;

            DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);

    END;

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 DATE)
RETURN NUMBER
IS
   v_age NUMBER;
BEGIN
   SELECT FLOOR(MONTHS_BETWEEN(SYSDATE, p_dob) / 12) INTO v_age FROM dual;
   RETURN v_age;
EXCEPTION
   WHEN OTHERS THEN
      RETURN NULL;
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.

```
CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(
   p_loan_amount NUMBER,
   p_annual_interest_rate NUMBER,
   p_loan_duration_years NUMBER
)
RETURN NUMBER
IS
   v_monthly_interest_rate NUMBER;
```

```
    v_number_of_months NUMBER;
    v_monthly_installment NUMBER;
BEGIN
    v_monthly_interest_rate := p_annual_interest_rate / 12 / 100;
    v_number_of_months := p_loan_duration_years * 12;

    IF v_monthly_interest_rate > 0 THEN
        v_monthly_installment := (p_loan_amount * v_monthly_interest_rate) /
                                 (1 - POWER(1 + v_monthly_interest_rate, -v_number_of_months));
    ELSE
        v_monthly_installment := p_loan_amount / v_number_of_months;
    END IF;

    RETURN v_monthly_installment;
EXCEPTION
    WHEN OTHERS THEN
        RETURN NULL;
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.

```
CREATE OR REPLACE FUNCTION HasSufficientBalance(
    p_account_id NUMBER,
    p_amount NUMBER
)
RETURN BOOLEAN
IS
```

```
    v_balance NUMBER;
BEGIN
    SELECT Balance INTO v_balance
    FROM Accounts
    WHERE AccountID = p_account_id;

    RETURN v_balance >= p_amount;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RETURN FALSE;
    WHEN OTHERS THEN
        RETURN FALSE;
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.

```
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 (

   AuditID NUMBER PRIMARY KEY,

   TransactionID NUMBER,

   ChangeDate DATE,

   ChangeType VARCHAR2(50)

);


CREATE SEQUENCE AuditLogSeq

START WITH 1

INCREMENT BY 1

NOCACHE

NOCYCLE;
```

```
CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

  INSERT INTO AuditLog (AuditID, TransactionID, ChangeDate, ChangeType)

  VALUES (AuditLogSeq.NEXTVAL, :NEW.TransactionID, SYSDATE, 'INSERT');

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 NUMBER;

BEGIN

  IF :NEW.TransactionType = 'Withdrawal' THEN

    SELECT Balance INTO v_balance

    FROM Accounts

    WHERE AccountID = :NEW.AccountID;


    IF v_balance < :NEW.Amount THEN

      RAISE_APPLICATION_ERROR(-20001, 'Insufficient funds for withdrawal');

    END IF;

  END IF;


  IF :NEW.TransactionType = 'Deposit' THEN

    IF :NEW.Amount <= 0 THEN

      RAISE_APPLICATION_ERROR(-20002, 'Deposit amount must be positive');

    END IF;
    END IF;
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.

```
DECLARE
   CURSOR cur_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 t.TransactionDate BETWEEN TRUNC(SYSDATE, 'MM') AND LAST_DAY(SYSDATE);
   v_customerID Customers.CustomerID%TYPE;
   v_name Customers.Name%TYPE;
   v_transactionDate Transactions.TransactionDate%TYPE;
   v_amount Transactions.Amount%TYPE;
   v_transactionType Transactions.TransactionType%TYPE;
BEGIN
   OPEN cur_transactions;
   LOOP
      FETCH cur_transactions INTO v_customerID, v_name, v_transactionDate, v_amount,
v_transactionType;
      EXIT WHEN cur_transactions%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE('Customer: ' || v_name || ' (' || v_customerID || ')');
      DBMS_OUTPUT.PUT_LINE('Transaction Date: ' || v_transactionDate);
      DBMS_OUTPUT.PUT_LINE('Amount: ' || v_amount || ' Type: ' || v_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
   CURSOR cur_accounts IS
      SELECT AccountID, Balance
      FROM Accounts;
   v_accountID Accounts.AccountID%TYPE;
   v_balance Accounts.Balance%TYPE;
   v_annualFee CONSTANT NUMBER := 100;
BEGIN
   OPEN cur_accounts;
   LOOP
      FETCH cur_accounts INTO v_accountID, v_balance;
      EXIT WHEN cur_accounts%NOTFOUND;
      UPDATE Accounts
      SET Balance = Balance - v_annualFee
      WHERE AccountID = v_accountID;
      DBMS_OUTPUT.PUT_LINE('Account ID: ' || v_accountID || ' New Balance: ' || (v_balance -
v_annualFee));
   END LOOP;
   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
   CURSOR cur_loans IS
```

```
        SELECT LoanID, InterestRate
        FROM Loans;
    v_loanID Loans.LoanID%TYPE;
    v_interestRate Loans.InterestRate%TYPE;
    v_newInterestRate CONSTANT NUMBER := 5;
BEGIN
    OPEN cur_loans;
    LOOP
        FETCH cur_loans INTO v_loanID, v_interestRate;
        EXIT WHEN cur_loans%NOTFOUND;
        UPDATE Loans
        SET InterestRate = v_newInterestRate
        WHERE LoanID = v_loanID;
        DBMS_OUTPUT.PUT_LINE('Loan ID: ' || v_loanID || ' New Interest Rate: ' || v_newInterestRate);
    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 AddCustomer(p_CustomerID NUMBER, p_Name VARCHAR2, p_DOB DATE, p_Balance NUMBER);
  PROCEDURE UpdateCustomer(p_CustomerID NUMBER, p_Name VARCHAR2, p_DOB DATE, p_Balance NUMBER);
  FUNCTION GetCustomerBalance(p_CustomerID NUMBER) RETURN NUMBER;
END CustomerManagement;

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS
  PROCEDURE AddCustomer(p_CustomerID NUMBER, p_Name VARCHAR2, p_DOB DATE, p_Balance NUMBER) IS
  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
      DBMS_OUTPUT.PUT_LINE('Customer with this ID already exists.');
  END AddCustomer;

  PROCEDURE UpdateCustomer(p_CustomerID NUMBER, p_Name VARCHAR2, p_DOB DATE, p_Balance NUMBER) IS
  BEGIN
    UPDATE Customers
    SET Name = p_Name, DOB = p_DOB, Balance = p_Balance, LastModified = SYSDATE
    WHERE CustomerID = p_CustomerID;
    IF SQL%ROWCOUNT = 0 THEN
      DBMS_OUTPUT.PUT_LINE('Customer not found.');
      END IF;
   END UpdateCustomer;

   FUNCTION GetCustomerBalance(p_CustomerID 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
         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 NUMBER, p_Name VARCHAR2, p_Position VARCHAR2, p_Salary NUMBER, p_Department VARCHAR2, p_HireDate DATE);
   PROCEDURE UpdateEmployee(p_EmployeeID NUMBER, p_Name VARCHAR2, p_Position VARCHAR2, p_Salary NUMBER, p_Department VARCHAR2);
   FUNCTION CalculateAnnualSalary(p_EmployeeID NUMBER) RETURN NUMBER;
END EmployeeManagement;

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS
   PROCEDURE HireEmployee(p_EmployeeID NUMBER, p_Name VARCHAR2, p_Position VARCHAR2, p_Salary NUMBER, p_Department VARCHAR2, p_HireDate 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);
  EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
      DBMS_OUTPUT.PUT_LINE('Employee with this ID already exists.');
  END HireEmployee;

  PROCEDURE UpdateEmployee(p_EmployeeID NUMBER, p_Name VARCHAR2, p_Position
VARCHAR2, p_Salary NUMBER, p_Department VARCHAR2) IS
  BEGIN
    UPDATE Employees
    SET Name = p_Name, Position = p_Position, Salary = p_Salary, Department = p_Department
    WHERE EmployeeID = p_EmployeeID;
    IF SQL%ROWCOUNT = 0 THEN
      DBMS_OUTPUT.PUT_LINE('Employee not found.');
    END IF;
  END UpdateEmployee;

  FUNCTION CalculateAnnualSalary(p_EmployeeID NUMBER) RETURN NUMBER IS
    v_salary NUMBER;
  BEGIN
    SELECT Salary INTO v_salary
    FROM Employees
    WHERE EmployeeID = p_EmployeeID;
    RETURN v_salary * 12;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      RETURN NULL;
  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.

```
CREATE OR REPLACE PACKAGE AccountOperations AS

  PROCEDURE OpenAccount(p_AccountID NUMBER, p_CustomerID NUMBER, p_AccountType VARCHAR2, p_Balance NUMBER);

  PROCEDURE CloseAccount(p_AccountID NUMBER);

  FUNCTION GetTotalBalance(p_CustomerID NUMBER) RETURN NUMBER;

END AccountOperations;


CREATE OR REPLACE PACKAGE BODY AccountOperations AS

  PROCEDURE OpenAccount(p_AccountID NUMBER, p_CustomerID NUMBER, p_AccountType VARCHAR2, p_Balance NUMBER) IS

  BEGIN

    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

      DBMS_OUTPUT.PUT_LINE('Account with this ID already exists.');

  END OpenAccount;


  PROCEDURE CloseAccount(p_AccountID NUMBER) IS

  BEGIN

    DELETE FROM Accounts

    WHERE AccountID = p_AccountID;

    IF SQL%ROWCOUNT = 0 THEN

      DBMS_OUTPUT.PUT_LINE('Account not found.');

    END IF;

  END CloseAccount;


  FUNCTION GetTotalBalance(p_CustomerID 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
      RETURN 0;
  END GetTotalBalance;
END AccountOperations;
```