Exercise 1: Control Structures

Scenario 1: Apply discount to loan interest rates for customers above 60 years old

```
BEGIN
  FOR customer IN (SELECT CustomerID, EXTRACT(YEAR FROM SYSDATE) -
EXTRACT(YEAR FROM DOB) AS Age
                   FROM Customers)
  LOOP
    IF customer.Age > 60 THEN
      UPDATE Loans
      SET InterestRate = InterestRate - 1
      WHERE CustomerID = customer.CustomerID;
    END IF;
  END LOOP;
END;
```

Scenario 2: Promote customer to VIP status based on balance

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

Scenario 3: Send reminders to customers whose loans are due within the next 30 days

```
BEGIN
  FOR loan IN (SELECT LoanID, CustomerID, EndDate
               FROM Loans
               WHERE EndDate BETWEEN SYSDATE AND SYSDATE + 30)
```

```
  LOOP
    DBMS_OUTPUT.PUT_LINE('Reminder: Your loan with ID ' || loan.LoanID
|| ' is due on ' || loan.EndDate);
  END LOOP;
END;
```

Exercise 2: Error Handling

Scenario 1: Handle exceptions during fund transfers between accounts

```
CREATE OR REPLACE PROCEDURE SafeTransferFunds (
  p_fromAccountID NUMBER,
  p_toAccountID NUMBER,
  p_amount NUMBER
) IS
  insufficient_funds EXCEPTION;
  insufficient_balance BOOLEAN;
BEGIN
  SELECT (Balance >= p_amount) INTO insufficient_balance FROM Accounts
WHERE AccountID = p_fromAccountID;

  IF NOT insufficient_balance THEN
    RAISE insufficient_funds;
  END IF;

  UPDATE Accounts SET Balance = Balance - p_amount WHERE AccountID =
p_fromAccountID;
  UPDATE Accounts SET Balance = Balance + p_amount WHERE AccountID =
p_toAccountID;

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

```
    ROLLBACK;
END;
```

Scenario 2: Manage errors when updating employee salaries

```
CREATE OR REPLACE PROCEDURE UpdateSalary (
  p_employeeID NUMBER,
  p_percentage NUMBER
) IS
  employee_not_found EXCEPTION;
  v_count NUMBER;
BEGIN
  SELECT COUNT(*) INTO v_count FROM Employees WHERE EmployeeID =
p_employeeID;

  IF v_count = 0 THEN
    RAISE employee_not_found;
  END IF;

  UPDATE Employees
  SET Salary = Salary * (1 + p_percentage / 100)
  WHERE EmployeeID = p_employeeID;

  COMMIT;
EXCEPTION
  WHEN employee_not_found THEN
    DBMS_OUTPUT.PUT_LINE('Error: Employee ID not found.');
    ROLLBACK;
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
    ROLLBACK;
END;
```

Scenario 3: Ensure data integrity when adding a new customer

```
CREATE OR REPLACE PROCEDURE AddNewCustomer (
  p_customerID NUMBER,
```

```
  p_name VARCHAR2,
  p_dob DATE,
  p_balance NUMBER
) IS
  customer_exists EXCEPTION;
  v_count NUMBER;
BEGIN
  SELECT COUNT(*) INTO v_count FROM Customers WHERE CustomerID =
p_customerID;

  IF v_count > 0 THEN
    RAISE customer_exists;
  END IF;

  INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)
  VALUES (p_customerID, p_name, p_dob, p_balance, SYSDATE);

  COMMIT;
EXCEPTION
  WHEN customer_exists THEN
    DBMS_OUTPUT.PUT_LINE('Error: Customer with the same ID already
exists.');
    ROLLBACK;
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
    ROLLBACK;
END;
```

Exercise 3: Stored Procedures

Scenario 1: Process monthly interest for all savings accounts

```
CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS
BEGIN
  FOR account IN (SELECT AccountID, Balance FROM Accounts WHERE
AccountType = 'Savings')
  LOOP
    UPDATE Accounts
```

```
    SET Balance = Balance * 1.01
    WHERE AccountID = account.AccountID;
  END LOOP;
  COMMIT;
END;
```

Scenario 2: Implement a bonus scheme for employees based on their performance

```
CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (
  p_department VARCHAR2,
  p_bonus_percentage NUMBER
) IS
BEGIN
  UPDATE Employees
  SET Salary = Salary * (1 + p_bonus_percentage / 100)
  WHERE Department = p_department;

  COMMIT;
END;
```

Scenario 3: Customers transfer funds between their accounts

```
CREATE OR REPLACE PROCEDURE TransferFunds (
  p_fromAccountID NUMBER,
  p_toAccountID NUMBER,
  p_amount NUMBER
) IS
  insufficient_funds EXCEPTION;
  insufficient_balance BOOLEAN;
BEGIN
  SELECT (Balance >= p_amount) INTO insufficient_balance FROM Accounts
WHERE AccountID = p_fromAccountID;

  IF NOT insufficient_balance THEN
    RAISE insufficient_funds;
  END IF;
```

```
  UPDATE Accounts SET Balance = Balance - p_amount WHERE AccountID =
p_fromAccountID;
  UPDATE Accounts SET Balance = Balance + p_amount WHERE AccountID =
p_toAccountID;

  COMMIT;
EXCEPTION
  WHEN insufficient_funds THEN
    DBMS_OUTPUT.PUT_LINE('Error: Insufficient funds in the source
account.');
    ROLLBACK;
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
    ROLLBACK;
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
  v_age := FLOOR(MONTHS_BETWEEN(SYSDATE, p_dob) / 12);
  RETURN v_age;
END;
```

Scenario 2: Compute the monthly installment for a loan

```
CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment (
  p_loan_amount NUMBER,
  p_interest_rate NUMBER,
  p_duration_years NUMBER
) RETURN NUMBER IS
  v_monthly_rate NUMBER;
```

```
  v_months NUMBER;
  v_installment NUMBER;
BEGIN
  v_monthly_rate := p_interest_rate / 100 / 12;
  v_months := p_duration_years * 12;
  v_installment := p_loan_amount * v_monthly_rate / (1 - POWER(1 +
v_monthly_rate, -v_months));
  RETURN v_installment;
END;
```

Scenario 3: Check if a customer has sufficient balance before making a transaction

```
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;
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 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.Amount
, :NEW.TransactionType);
END;
```

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, 'Withdrawal amount exceeds
available balance.');
  ELSIF :NEW.TransactionType = 'Deposit' AND :NEW.Amount <= 0 THEN
    RAISE_APPLICATION_ERROR(-20002, 'Deposit amount must be
positive.');
  END IF;
END;
```

Exercise 6: Cursors

Scenario 1: Generate monthly statements for all customers

```
BEGIN
  FOR customer IN (SELECT DISTINCT CustomerID FROM Accounts)
  LOOP
    DBMS_OUTPUT.PUT_LINE('Monthly Statement for Customer ID: ' ||
```

```
customer.CustomerID);
    FOR transaction IN (SELECT TransactionDate, Amount,
TransactionType
                        FROM Transactions
                        WHERE AccountID IN (SELECT AccountID FROM
Accounts WHERE CustomerID = customer.CustomerID)
                          AND TransactionDate BETWEEN TRUNC(SYSDATE,
'MM') AND LAST_DAY(SYSDATE))
    LOOP
      DBMS_OUTPUT.PUT_LINE(transaction.TransactionDate || ' - ' ||
transaction.TransactionType || ' - ' || transaction.Amount);
    END LOOP;
  END LOOP;
END;
```

Scenario 2: Apply annual fee to all accounts

```
BEGIN
  FOR account IN (SELECT AccountID, Balance FROM Accounts)
  LOOP
    UPDATE Accounts
    SET Balance = Balance - 50
    WHERE AccountID = account.AccountID;
  END LOOP;
  COMMIT;
END;
```

Scenario 3: Update the interest rate for all loans based on a new policy

```
BEGIN
  FOR loan IN (SELECT LoanID, InterestRate FROM Loans)
  LOOP
    UPDATE Loans
    SET InterestRate = InterestRate + 0.5
    WHERE LoanID = loan.LoanID;
  END LOOP;
  COMMIT;
```

```
END;
```

Exercise 7: Packages

Scenario 1: Group all customer-related procedures and functions into a package

```
CREATE OR REPLACE PACKAGE CustomerManagement IS
  PROCEDURE AddNewCustomer(p_customerID NUMBER, p_name VARCHAR2, p_dob
DATE, p_balance NUMBER);
  PROCEDURE UpdateCustomerDetails(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 IS
  PROCEDURE AddNewCustomer (
    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);
  END AddNewCustomer;

  PROCEDURE UpdateCustomerDetails (
    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;
  END UpdateCustomerDetails;

  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;
  END GetCustomerBalance;
END CustomerManagement;
```

Scenario 2: Create a package to manage employee data

```
CREATE OR REPLACE PACKAGE EmployeeManagement IS
  PROCEDURE HireNewEmployee(p_employeeID NUMBER, p_name VARCHAR2,
p_position VARCHAR2, p_salary NUMBER, p_department VARCHAR2,
p_hiredate DATE);
  PROCEDURE UpdateEmployeeDetails(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 IS
  PROCEDURE HireNewEmployee (
    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);
  END HireNewEmployee;

  PROCEDURE UpdateEmployeeDetails (
    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;
  END UpdateEmployeeDetails;

  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;
  END CalculateAnnualSalary;
END EmployeeManagement;
```

Scenario 3: Group all account-related operations into a package

```
CREATE OR REPLACE PACKAGE AccountOperations IS
  PROCEDURE OpenNewAccount(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 IS
  PROCEDURE OpenNewAccount (
    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);
  END OpenNewAccount;

  PROCEDURE CloseAccount (
    p_accountID NUMBER
  ) IS
  BEGIN
    DELETE FROM Accounts WHERE AccountID = p_accountID;
  END CloseAccount;

  FUNCTION GetTotalBalance (
    p_customerID NUMBER
  ) RETURN NUMBER IS
    v_total_balance NUMBER;
  BEGIN
    SELECT SUM(Balance) INTO v_total_balance FROM Accounts WHERE
CustomerID = p_customerID;
    RETURN v_total_balance;
  END GetTotalBalance;
END AccountOperations;
```