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 - This PL/SQL code block updates the interest rates on loans for customers who are over 60 years old by applying a 1% discount.

```
DECLARE
   CURSOR cur_customers IS
      SELECT CustomerID, LoanID, InterestRate
      FROM Loans
      WHERE CustomerID IN (SELECT CustomerID FROM Customers WHERE
TRUNC(MONTHS_BETWEEN(SYSDATE, DOB) / 12) > 60);

   v_customer_id Loans.CustomerID%TYPE;
   v_loan_id Loans.LoanID%TYPE;
   v_interest_rate Loans.InterestRate%TYPE;
BEGIN
   OPEN cur_customers;
   LOOP
      FETCH cur_customers INTO v_customer_id, v_loan_id, v_interest_rate;
      EXIT WHEN cur_customers%NOTFOUND;

      UPDATE Loans
      SET InterestRate = InterestRate - 1
      WHERE LoanID = v_loan_id;

      DBMS_OUTPUT.PUT_LINE('Applied 1% discount to loan ID: ' || v_loan_id);
   END LOOP;
   CLOSE cur_customers;
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 - The code block promotes customers with a balance greater than $10,000 to VIP status.

```
 DECLARE
  CURSOR cur_customers IS
    SELECT CustomerID
    FROM Customers
    WHERE Balance > 10000;

  v_customer_id Customers.CustomerID%TYPE;
BEGIN
  OPEN cur_customers;
  LOOP
    FETCH cur_customers INTO v_customer_id;
    EXIT WHEN cur_customers%NOTFOUND;

    UPDATE Customers
    SET IsVIP = TRUE
    WHERE CustomerID = v_customer_id;

    DBMS_OUTPUT.PUT_LINE('Customer ID ' || v_customer_id || ' has been promoted to VIP status.');
  END LOOP;
  CLOSE cur_customers;
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 - This PL/SQL code block sends reminders for loans that are due within the next 30 days.

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

   v_customer_id Loans.CustomerID%TYPE;
   v_loan_id Loans.LoanID%TYPE;
   v_end_date Loans.EndDate%TYPE;
BEGIN
   OPEN cur_loans;
   LOOP
      FETCH cur_loans INTO v_customer_id, v_loan_id, v_end_date;
      EXIT WHEN cur_loans%NOTFOUND;

      DBMS_OUTPUT.PUT_LINE('Reminder: Customer ID ' || v_customer_id || ', Loan ID ' || v_loan_id || ' is due on ' || TO_CHAR(v_end_date, 'YYYY-MM-DD'));
   END LOOP;
   CLOSE cur_loans;
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 - This PL/SQL procedure, SafeTransferFunds, safely transfers funds from one account to another while handling potential errors such as insufficient funds.

```
CREATE OR REPLACE PROCEDURE SafeTransferFunds (
    p_from_account_id IN Accounts.AccountID%TYPE,
    p_to_account_id IN Accounts.AccountID%TYPE,
    p_amount IN NUMBER
) IS
    e_insufficient_funds EXCEPTION;
    v_balance Accounts.Balance%TYPE;
BEGIN
    -- Check balance of the source account
    SELECT Balance INTO v_balance FROM Accounts WHERE AccountID = p_from_account_id;

    IF v_balance < p_amount THEN
        RAISE e_insufficient_funds;
    END IF;

    -- Deduct amount from source account
    UPDATE Accounts
    SET Balance = Balance - p_amount
    WHERE AccountID = p_from_account_id;

    -- Add amount to destination account
    UPDATE Accounts
    SET Balance = Balance + p_amount
    WHERE AccountID = p_to_account_id;
EXCEPTION
    WHEN e_insufficient_funds THEN
        DBMS_OUTPUT.PUT_LINE('Error: Insufficient funds in account ' || p_from_account_id);
```

```
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
END;
```

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 - This PL/SQL procedure, UpdateSalary, updates the salary of an employee based on a given percentage increase.

```
CREATE OR REPLACE PROCEDURE UpdateSalary (
    p_employee_id IN Employees.EmployeeID%TYPE,
    p_percentage IN NUMBER
) IS
    v_new_salary Employees.Salary%TYPE;
    e_employee_not_found EXCEPTION;
BEGIN
    BEGIN
        SELECT Salary INTO v_new_salary FROM Employees WHERE EmployeeID = p_employee_id;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            RAISE e_employee_not_found;
    END;
    v_new_salary := v_new_salary + (v_new_salary * p_percentage / 100);


    UPDATE Employees
    SET Salary = v_new_salary
    WHERE EmployeeID = p_employee_id;
EXCEPTION
    WHEN e_employee_not_found THEN
        DBMS_OUTPUT.PUT_LINE('Error: Employee ID ' || p_employee_id || ' not found');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
END;
```

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 - This PL/SQL procedure, AddNewCustomer, adds a new customer to the Customers table, ensuring that the customer ID is unique.

```
CREATE OR REPLACE PROCEDURE AddNewCustomer (
    p_customer_id IN Customers.CustomerID%TYPE,
    p_name IN Customers.Name%TYPE,
    p_dob IN Customers.DOB%TYPE,
    p_balance IN Customers.Balance%TYPE
) IS
    e_customer_exists EXCEPTION;
BEGIN
    BEGIN
        SELECT CustomerID FROM Customers WHERE CustomerID = p_customer_id;
        RAISE e_customer_exists;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            NULL;
    END;

    INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)
    VALUES (p_customer_id, p_name, p_dob, p_balance, SYSDATE);
EXCEPTION
    WHEN e_customer_exists THEN
        DBMS_OUTPUT.PUT_LINE('Error: Customer ID ' || p_customer_id || ' already exists');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
END;
```

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 - This PL/SQL procedure, ProcessMonthlyInterest, calculates and updates the monthly interest for all savings accounts

```
CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

  CURSOR cur_savings_accounts IS

    SELECT AccountID, Balance

    FROM Accounts

    WHERE AccountType = 'Savings';

  v_account_id Accounts.AccountID%TYPE;

  v_balance Accounts.Balance%TYPE;

  v_interest_rate CONSTANT NUMBER := 0.01;

BEGIN
  OPEN cur_savings_accounts;

  LOOP

    FETCH cur_savings_accounts INTO v_account_id, v_balance;

    EXIT WHEN cur_savings_accounts%NOTFOUND;

    UPDATE Accounts

    SET Balance = Balance + (Balance * v_interest_rate)
```

```
        WHERE AccountID = v_account_id;

    END LOOP;

    CLOSE cur_savings_accounts;
END;
```

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 - This PL/SQL procedure, UpdateEmployeeBonus, updates the salary of employees in a specified department by adding a bonus percentage.

```
CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (

    p_department IN Employees.Department%TYPE,

    p_bonus_percentage IN NUMBER

) IS

BEGIN

    UPDATE Employees

    SET Salary = Salary + (Salary * p_bonus_percentage / 100)

    WHERE Department = p_department;
END;
```

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 - This PL/SQL procedure, TransferFunds, transfers a specified amount of money from one account to another while ensuring there are sufficient funds in the source account

```
CREATE OR REPLACE PROCEDURE TransferFunds (

    p_from_account_id IN Accounts.AccountID%TYPE,

    p_to_account_id IN Accounts.AccountID%TYPE,

    p_amount IN NUMBER

) IS

    e_insufficient_funds EXCEPTION;

    v_balance Accounts.Balance%TYPE;

BEGIN
    -- Check the balance of the source account

    SELECT Balance INTO v_balance FROM Accounts WHERE AccountID = p_from_account_id;

    IF v_balance < p_amount THEN

        RAISE e_insufficient_funds;

    END IF;



    -- Deduct the amount from the source account
```

```
    UPDATE Accounts

    SET Balance = Balance - p_amount

    WHERE AccountID = p_from_account_id;

    -- Add the amount to the destination account

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

    DBMS_OUTPUT.PUT_LINE('Transferred ' || p_amount || ' from account ' || p_from_account_id || ' to
account ' || p_to_account_id);

EXCEPTION

    WHEN e_insufficient_funds THEN
        DBMS_OUTPUT.PUT_LINE('Error: Insufficient funds in account ' || p_from_account_id);
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
END;
```

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 - This PL/SQL function, CalculateAge, calculates the age of a person based on their date of birth and the current date.

```
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);
   DBMS_OUTPUT.PUT_LINE('Age: ' || v_age);
   RETURN v_age;
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.

Answer - This PL/SQL function, CalculateMonthlyInstallment, calculates the monthly installment for a loan based on the loan amount, interest rate, and duration.

```
CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment (
    p_loan_amount IN NUMBER,
    p_interest_rate IN NUMBER,
    p_duration_years IN NUMBER
) RETURN NUMBER IS
    v_monthly_installment NUMBER;
    v_monthly_rate NUMBER;
    v_num_payments NUMBER;
BEGIN
    v_monthly_rate := p_interest_rate / 1200;
    v_num_payments := p_duration_years * 12;

    v_monthly_installment := (p_loan_amount * v_monthly_rate) / (1 - POWER(1 + v_monthly_rate,
-v_num_payments));

    DBMS_OUTPUT.PUT_LINE('Monthly Installment: ' || v_monthly_installment);
    RETURN v_monthly_installment;
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.

Answer - This PL/SQL function, HasSufficientBalance, checks if an account has enough balance to cover a specified amount.

```
CREATE OR REPLACE FUNCTION HasSufficientBalance (
   p_account_id IN Accounts.AccountID%TYPE,
   p_amount IN NUMBER
) RETURN BOOLEAN IS
   v_balance Accounts.Balance%TYPE;
BEGIN
   SELECT Balance INTO v_balance FROM Accounts WHERE AccountID = p_account_id;

   IF v_balance >= p_amount THEN
      DBMS_OUTPUT.PUT_LINE('Sufficient balance available in account ' || p_account_id);
      RETURN TRUE;
   ELSE
      DBMS_OUTPUT.PUT_LINE('Insufficient balance in account ' || p_account_id);
      RETURN FALSE;
   END IF;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('Account ' || p_account_id || ' not found');
      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.

Answer - This PL/SQL trigger, UpdateCustomerLastModified, automatically updates the LastModified timestamp whenever the Name or Balance fields of a customer record are updated.

```
CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE OF Name, Balance ON Customers

FOR EACH ROW

BEGIN

   :NEW.LastModified := SYSDATE;

   DBMS_OUTPUT.PUT_LINE('Customer record updated: ID = ' || :NEW.CustomerID || ', LastModified set to: ' || :NEW.LastModified);

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.

Answer - This PL/SQL trigger, LogTransaction, records a log entry in the AuditLog table each time a new row is inserted into the Transactions table.

```
CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

   INSERT INTO AuditLog (TransactionID, Action, ActionDate)

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


   DBMS_OUTPUT.PUT_LINE('Transaction logged: ' || :NEW.TransactionID || ' with Action: 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.

Answer - This PL/SQL trigger, CheckTransactionRules, validates transaction rules before inserting a new record into the Transactions table.

```
CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

BEGIN

  IF :NEW.TransactionType = 'Withdrawal' THEN

    DECLARE

      v_balance Accounts.Balance%TYPE;

    BEGIN

      SELECT Balance INTO v_balance FROM Accounts WHERE AccountID = :NEW.AccountID;

      IF v_balance < :NEW.Amount THEN

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

      END IF;

    END;

  ELSIF :NEW.TransactionType = 'Deposit' THEN

    IF :NEW.Amount <= 0 THEN

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

    END IF;

  END IF;


  DBMS_OUTPUT.PUT_LINE('Transaction validated: ' || :NEW.TransactionID || ', Type: '
|| :NEW.TransactionType);

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.

Answer - This PL/SQL block retrieves and displays transactions from the Transactions table that occurred in the current month and year.

```
DECLARE

  CURSOR cur_transactions IS

    SELECT AccountID, TransactionDate, Amount, TransactionType

    FROM Transactions

    WHERE EXTRACT(MONTH FROM TransactionDate) = EXTRACT(MONTH FROM SYSDATE)

    AND EXTRACT(YEAR FROM TransactionDate) = EXTRACT(YEAR FROM SYSDATE);

  v_account_id Transactions.AccountID%TYPE;

  v_transaction_date Transactions.TransactionDate%TYPE;

  v_amount Transactions.Amount%TYPE;

  v_transaction_type Transactions.TransactionType%TYPE;

BEGIN

  OPEN cur_transactions;

  LOOP

    FETCH cur_transactions INTO v_account_id, v_transaction_date, v_amount,
```

```
V_transaction_type;

        EXIT WHEN cur_transactions%NOTFOUND;

        DBMS_OUTPUT.PUT_LINE('Account ID: ' || v_account_id || ', Date: ' || v_transaction_date || ',
Amount: ' || v_amount || ', Type: ' || v_transaction_type);

    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.

Answer - This PL/SQL block applies a fee to all accounts in the Accounts table and updates their balances accordingly.

```
DECLARE

   CURSOR cur_accounts IS

      SELECT AccountID, Balance

      FROM Accounts;

   v_account_id Accounts.AccountID%TYPE;

   v_balance Accounts.Balance%TYPE;

BEGIN

   OPEN cur_accounts;

   LOOP
      FETCH cur_accounts INTO v_account_id, v_balance;
      EXIT WHEN cur_accounts%NOTFOUND;

      UPDATE Accounts
      SET Balance = Balance - 50  -- Example fee amount
      WHERE AccountID = v_account_id;

      DBMS_OUTPUT.PUT_LINE('Annual fee applied to Account ID: ' || v_account_id || ', New Balance: ' ||
(v_balance - 50));

   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.

Answer - This PL/SQL block updates the interest rates for all loans in the table by increasing each rate by 0.5 units.

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

   v_loan_id Loans.LoanID%TYPE;
   v_interest_rate Loans.InterestRate%TYPE;
BEGIN
   OPEN cur_loans;
   LOOP
      FETCH cur_loans INTO v_loan_id, v_interest_rate;
      EXIT WHEN cur_loans%NOTFOUND;

      UPDATE Loans
      SET InterestRate = InterestRate + 0.5  -- Example rate adjustment
      WHERE LoanID = v_loan_id;

      DBMS_OUTPUT.PUT_LINE('Interest rate updated for Loan ID: ' || v_loan_id || ', New Rate: ' ||
(v_interest_rate + 0.5));
   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.

Answer - This PL/SQL package, CustomerManagement, provides procedures and functions for managing customer records.

```
CREATE OR REPLACE PACKAGE CustomerManagement AS
   PROCEDURE AddNewCustomer (
      p_customer_id IN Customers.CustomerID%TYPE,
      p_name IN Customers.Name%TYPE,
      p_dob IN Customers.DOB%TYPE,
      p_balance IN Customers.Balance%TYPE
   );

   PROCEDURE UpdateCustomer (
      p_customer_id IN Customers.CustomerID%TYPE,
      p_name IN Customers.Name%TYPE,
      p_balance IN Customers.Balance%TYPE
   );

   FUNCTION GetCustomerBalance (
      p_customer_id IN Customers.CustomerID%TYPE
   ) RETURN NUMBER;
END CustomerManagement;
/

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS
   PROCEDURE AddNewCustomer (
      p_customer_id IN Customers.CustomerID%TYPE,
      p_name IN Customers.Name%TYPE,
      p_dob IN Customers.DOB%TYPE,
      p_balance IN Customers.Balance%TYPE
   ) IS
   BEGIN
```

```
    INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)
    VALUES (p_customer_id, p_name, p_dob, p_balance, SYSDATE);
    DBMS_OUTPUT.PUT_LINE('New customer added: ' || p_customer_id || ', Name: ' || p_name);
  END;

  PROCEDURE UpdateCustomer (
    p_customer_id IN Customers.CustomerID%TYPE,
    p_name IN Customers.Name%TYPE,
    p_balance IN Customers.Balance%TYPE
  ) IS
  BEGIN
    UPDATE Customers
    SET Name = p_name, Balance = p_balance, LastModified = SYSDATE
    WHERE CustomerID = p_customer_id;
    DBMS_OUTPUT.PUT_LINE('Customer updated: ' || p_customer_id || ', New Name: ' || p_name);
  END;

  FUNCTION GetCustomerBalance (
    p_customer_id IN Customers.CustomerID%TYPE
  ) RETURN NUMBER IS
    v_balance Customers.Balance%TYPE;
  BEGIN
    SELECT Balance INTO v_balance FROM Customers WHERE CustomerID = p_customer_id;
    DBMS_OUTPUT.PUT_LINE('Customer balance for ID ' || p_customer_id || ': ' || v_balance);
    RETURN v_balance;
  END;
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 - This PL/SQL package, EmployeeManagement, provides procedures and a function for managing employee records.

```
CREATE OR REPLACE PACKAGE EmployeeManagement AS
  PROCEDURE HireNewEmployee (
    p_employee_id IN Employees.EmployeeID%TYPE,
    p_name IN Employees.Name%TYPE,
    p_position IN Employees.Position%TYPE,
    p_salary IN Employees.Salary%TYPE,
    p_department IN Employees.Department%TYPE,
    p_hire_date IN Employees.HireDate%TYPE
  );

  PROCEDURE UpdateEmployee (
    p_employee_id IN Employees.EmployeeID%TYPE,
    p_name IN Employees.Name%TYPE,
    p_position IN Employees.Position%TYPE,
    p_salary IN Employees.Salary%TYPE
  );

  FUNCTION CalculateAnnualSalary (
    p_employee_id IN Employees.EmployeeID%TYPE
  ) RETURN NUMBER;
END EmployeeManagement;
/

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS
  PROCEDURE HireNewEmployee (
    p_employee_id IN Employees.EmployeeID%TYPE,
    p_name IN Employees.Name%TYPE,
    p_position IN Employees.Position%TYPE,
    p_salary IN Employees.Salary%TYPE,
```

```
    p_department IN Employees.Department%TYPE,
    p_hire_date IN Employees.HireDate%TYPE
  ) 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);
    DBMS_OUTPUT.PUT_LINE('New employee hired: ' || p_employee_id || ', Name: ' || p_name);
  END;

  PROCEDURE UpdateEmployee (
    p_employee_id IN Employees.EmployeeID%TYPE,
    p_name IN Employees.Name%TYPE,
    p_position IN Employees.Position%TYPE,
    p_salary IN Employees.Salary%TYPE
  ) IS
  BEGIN
    UPDATE Employees
    SET Name = p_name, Position = p_position, Salary = p_salary
    WHERE EmployeeID = p_employee_id;
    DBMS_OUTPUT.PUT_LINE('Employee updated: ' || p_employee_id || ', New Position: ' || p_position);
  END;

  FUNCTION CalculateAnnualSalary (
    p_employee_id IN Employees.EmployeeID%TYPE
  ) RETURN NUMBER IS
    v_salary Employees.Salary%TYPE;
  BEGIN
    SELECT Salary INTO v_salary FROM Employees WHERE EmployeeID = p_employee_id;
    DBMS_OUTPUT.PUT_LINE('Annual salary for Employee ID ' || p_employee_id || ': ' || (v_salary * 12));
    RETURN v_salary * 12;
  END;
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 - This PL/SQL package body, AccountOperations, provides implementations for managing bank accounts.

```
CREATE OR REPLACE PACKAGE BODY AccountOperations AS
  PROCEDURE OpenNewAccount (
    p_account_id IN Accounts.AccountID%TYPE,
    p_customer_id IN Accounts.CustomerID%TYPE,
    p_account_type IN Accounts.AccountType%TYPE,
    p_balance IN Accounts.Balance%TYPE
  ) IS
  BEGIN
    INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified)
    VALUES (p_account_id, p_customer_id, p_account_type, p_balance, SYSDATE);
    DBMS_OUTPUT.PUT_LINE('New account opened: ID = ' || p_account_id || ', Type = ' ||
p_account_type);
  END;

  PROCEDURE CloseAccount (
    p_account_id IN Accounts.AccountID%TYPE
  ) IS
  BEGIN
    DELETE FROM Accounts WHERE AccountID = p_account_id;
    DBMS_OUTPUT.PUT_LINE('Account closed: ID = ' || p_account_id);
  END;

  FUNCTION GetTotalBalance (
    p_customer_id IN Accounts.CustomerID%TYPE
  ) RETURN NUMBER IS
    v_total_balance NUMBER := 0;
  BEGIN
    SELECT SUM(Balance) INTO v_total_balance
```

```
        FROM Accounts
        WHERE CustomerID = p_customer_id;

        DBMS_OUTPUT.PUT_LINE('Total balance for Customer ID ' || p_customer_id || ': ' || v_total_balance);
        RETURN v_total_balance;
    END;
END AccountOperations;
```