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.

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.

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.

CODE:

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

OUTPUT:

```
1 row created.

SQL> INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate)
  2  VALUES (1, 1, 5000, 5, SYSDATE, ADD_MONTHS(SYSDATE, 60));

1 row created.

SQL> INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate)
  2  VALUES (1, 'Alice Johnson', 'Manager', 70000, 'HR', TO_DATE('2015-06-15', 'YYYY-MM-DD'));

1 row created.

SQL> INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate)
  2  VALUES (2, 'Bob Brown', 'Developer', 60000, 'IT', TO_DATE('2017-03-20', 'YYYY-MM-DD'));

1 row created.

SQL> BEGIN
  2    FOR cust IN (SELECT * FROM Customers) LOOP
  3      IF MONTHS_BETWEEN(SYSDATE, cust.DOB)/12 > 60 THEN
  4        UPDATE Loans SET InterestRate = InterestRate - 1
  5        WHERE CustomerID = cust.CustomerID;
  6      END IF;
  7    END LOOP;
  8  END;
  9  /

PL/SQL procedure successfully completed.
```

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.

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.

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.

CODE:

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

OUTPUT:

```
savepoint set sql execute commit forall merge standard pipe
purge json_object
The symbol "4" was ignored.
ORA-06550: line 5, column 3:
PLS-00103: Encountered the symbol "5" when expecting one of the following:
, * & - + ; / at mod remainder rem return returning
<an exponent (**)> where || multiset
The symbol "*" was substituted for "5" to continue.
ORA-06550: line 6, column 3:
PLS-00103: Encountered the symbol "6" when expecting one of the following:
( begin case declare else elsif end exit for goto if loop mod
null pragma raise return select update while with
<an identifier> <a double-quoted delimited-identifier>
<a bind variable> << continue close current delete fetch lock
insert open rollback savepoint set sql execute commi


SQL> BEGIN
  2    FOR cust IN (SELECT * FROM Customers) LOOP
  3      IF MONTHS_BETWEEN(SYSDATE, cust.DOB)/12 > 60 THEN
  4        UPDATE Loans SET InterestRate = InterestRate - 1
  5        WHERE CustomerID = cust.CustomerID;
  6      END IF;
  7    END LOOP;
  8  END;
  9  /

PL/SQL procedure successfully completed.

SQL>
```

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.

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.

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.

CODE:

```
BEGIN
  FOR l IN (SELECT * FROM Loans WHERE EndDate <= SYSDATE + 30) LOOP
    DBMS_OUTPUT.PUT_LINE('Reminder: Loan ' || l.LoanID || ' for Customer ' || l.CustomerID || ' is due soon.');
  END LOOP;
END;
/
```

OUTPUT:

```
SQL> BEGIN
  2    FOR l IN (SELECT * FROM Loans WHERE EndDate <= SYSDATE + 30) LOOP
  3      DBMS_OUTPUT.PUT_LINE('Reminder: Loan ' || l.LoanID || ' for Customer ' || l.CustomerID || ' is due soon.');
  4    END LOOP;
  5  END;
  6  /

PL/SQL procedure successfully completed.

SQL>
```

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.

CODE:

```
CREATE OR REPLACE PROCEDURE SafeTransferFunds(p_from NUMBER, p_to NUMBER,
p_amount NUMBER) IS
BEGIN
  UPDATE Accounts SET Balance = Balance - p_amount WHERE AccountID = p_from;
  UPDATE Accounts SET Balance = Balance + p_amount WHERE AccountID = p_to;
  COMMIT;
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
    INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount,
TransactionType)
    VALUES (Transactions_seq.NEXTVAL, p_from, SYSDATE, 0, 'Error');
    DBMS_OUTPUT.PUT_LINE('Transfer failed: ' || SQLERRM);
END;
/
```

OUTPUT:

```
Sequence created.

SQL> VALUES (Transactions_seq.NEXTVAL, p_from, SYSDATE, 0, 'Error');
SP2-0734: unknown command beginning "VALUES (Tr..." - rest of line ignored.
SQL> CREATE OR REPLACE PROCEDURE SafeTransferFunds(p_from NUMBER, p_to NUMBER, p_amount NUMBER) IS
  2  BEGIN
  3    UPDATE Accounts SET Balance = Balance - p_amount WHERE AccountID = p_from;
  4    UPDATE Accounts SET Balance = Balance + p_amount WHERE AccountID = p_to;
  5    COMMIT;
  6  EXCEPTION
  7    WHEN OTHERS THEN
  8      ROLLBACK;
  9      INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)
 10      VALUES (Transactions_seq.NEXTVAL, p_from, SYSDATE, 0, 'Error');
 11      DBMS_OUTPUT.PUT_LINE('Transfer failed: ' || SQLERRM);
 12  END;
 13  /

Procedure created.
```

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.

CODE:

```
CREATE OR REPLACE PROCEDURE UpdateSalary(p_empid NUMBER, p_percent NUMBER) IS
BEGIN
 UPDATE Employees SET Salary = Salary + (Salary * p_percent / 100) WHERE EmployeeID =
    p_empid;
 IF SQL%NOTFOUND THEN
  RAISE_APPLICATION_ERROR(-20001, 'Employee not found.');
 END IF;
EXCEPTION
 WHEN OTHERS THEN
  DBMS_OUTPUT.PUT_LINE('Error updating salary: ' || SQLERRM);
END;
 /
```

OUTPUT:

```
SQL> CREATE OR REPLACE PROCEDURE UpdateSalary(p_empid NUMBER, p_percent NUMBER) IS
  2  BEGIN
  3    UPDATE Employees SET Salary = Salary + (Salary * p_percent / 100) WHERE EmployeeID = p_empid;
  4    IF SQL%NOTFOUND THEN
  5      RAISE_APPLICATION_ERROR(-20001, 'Employee not found.');
  6    END IF;
  7  EXCEPTION
  8    WHEN OTHERS THEN
  9      DBMS_OUTPUT.PUT_LINE('Error updating salary: ' || SQLERRM);
 10  END;
 11  /

Procedure created.
```

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.

CODE:

```
CREATE OR REPLACE PROCEDURE AddNewCustomer(p_id NUMBER, p_name VARCHAR2, p_dob DATE, p_balance NUMBER) IS
BEGIN
 INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)
 VALUES (p_id, p_name, p_dob, p_balance, SYSDATE);
EXCEPTION
 WHEN DUP_VAL_ON_INDEX THEN
  DBMS_OUTPUT.PUT_LINE('Customer already exists.');
END;
/
```

OUTPUT:

```
SQL> CREATE OR REPLACE PROCEDURE AddNewCustomer(p_id NUMBER, p_name VARCHAR2, p_dob DATE, p_balance NUMBER) IS
  2  BEGIN
  3    INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)
  4    VALUES (p_id, p_name, p_dob, p_balance, SYSDATE);
  5  EXCEPTION
  6    WHEN DUP_VAL_ON_INDEX THEN
  7      DBMS_OUTPUT.PUT_LINE('Customer already exists.');
  8  END;
  9  /

Procedure created.
```

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.

CODE:

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

OUTPUT:

```
SQL> CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS
  2  BEGIN
  3    UPDATE Accounts SET Balance = Balance + (Balance * 0.01) WHERE AccountType = 'Savings';
  4  END;
  5  /

Procedure created.
```

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.

CODE:

```
CREATE OR REPLACE TRIGGER LogTransaction
AFTER INSERT ON Transactions
FOR EACH ROW
BEGIN
INSERT INTO AuditLog (TransactionID, LogDate, Details)
VALUES (:NEW.TransactionID, SYSDATE, 'Transaction Inserted');
END;
/
```

OUTPUT:

```
SQL> SHOW ERRORS TRIGGER LogTransaction;
No errors.
SQL> CREATE OR REPLACE TRIGGER LogTransaction
  2  AFTER INSERT ON Transactions
  3  FOR EACH ROW
  4  BEGIN
  5    INSERT INTO AuditLog (TransactionID, LogDate, Details)
  6    VALUES (:NEW.TransactionID, SYSDATE, 'Transaction Inserted');
  7  END;
  8  /

Trigger created.
```

Scenario 3: Enforce business rules on deposits and withdrawals.

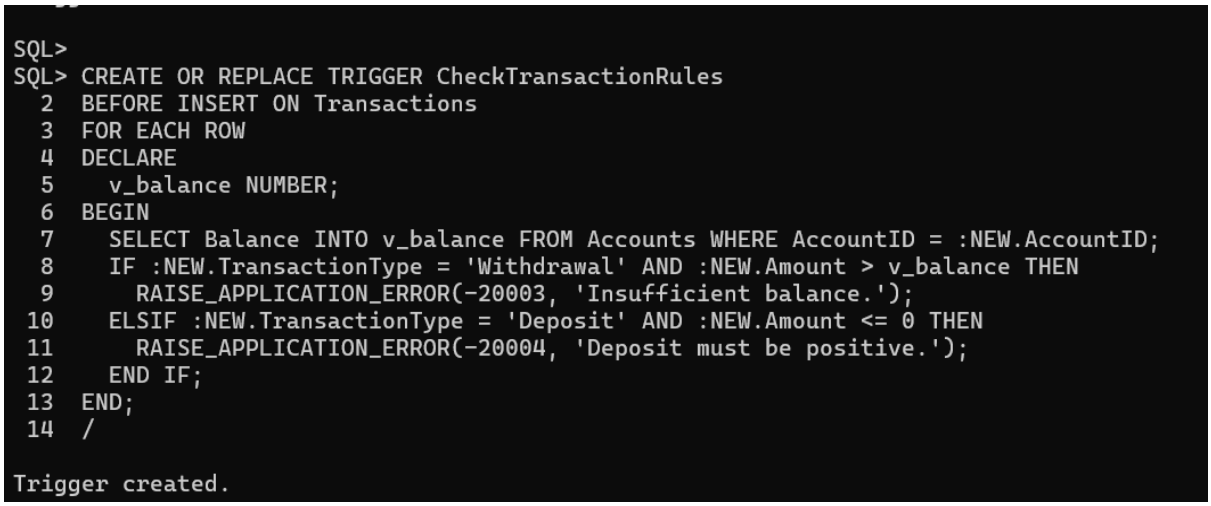
- **Question:** Write a trigger **CheckTransactionRules** that ensures withdrawals do not exceed the balance and deposits are positive before inserting a record into the Transactions table.

CODE:

```
CREATE OR REPLACE TRIGGER CheckTransactionRules
BEFORE INSERT ON Transactions
FOR EACH ROW
DECLARE
 v_balance NUMBER;
BEGIN
```

```
  SELECT Balance INTO v_balance FROM Accounts WHERE AccountID =
:NEW.AccountID;
  IF :NEW.TransactionType = 'Withdrawal' AND :NEW.Amount > v_balance THEN
    RAISE_APPLICATION_ERROR(-20003, 'Insufficient balance.');
  ELSIF :NEW.TransactionType = 'Deposit' AND :NEW.Amount <= 0 THEN
    RAISE_APPLICATION_ERROR(-20004, 'Deposit must be positive.');
  END IF;
END;
/
```

OUTPUT:



```
SQL>
SQL> CREATE OR REPLACE TRIGGER CheckTransactionRules
  2  BEFORE INSERT ON Transactions
  3  FOR EACH ROW
  4  DECLARE
  5    v_balance NUMBER;
  6  BEGIN
  7    SELECT Balance INTO v_balance FROM Accounts WHERE AccountID = :NEW.AccountID;
  8    IF :NEW.TransactionType = 'Withdrawal' AND :NEW.Amount > v_balance THEN
  9      RAISE_APPLICATION_ERROR(-20003, 'Insufficient balance.');
 10    ELSIF :NEW.TransactionType = 'Deposit' AND :NEW.Amount <= 0 THEN
 11      RAISE_APPLICATION_ERROR(-20004, 'Deposit must be positive.');
 12    END IF;
 13  END;
 14  /

Trigger created.
```

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.

CODE:

```
DECLARE
 CURSOR c_trans IS SELECT * FROM Transactions WHERE TransactionDate BETWEEN
    TRUNC(SYSDATE, 'MM') AND LAST_DAY(SYSDATE);
BEGIN
 FOR t IN c_trans LOOP
  DBMS_OUTPUT.PUT_LINE('Transaction: ' || t.TransactionID || ', Account: ' || t.AccountID);
 END LOOP;
END;
/
```

OUTPUT:

```
SQL> DECLARE
  2    CURSOR c_trans IS SELECT * FROM Transactions WHERE TransactionDate BETWEEN TRUNC(SYSDATE, 'MM') AND LAST_DAY(SYSD
ATE);
  3  BEGIN
  4    FOR t IN c_trans LOOP
  5      DBMS_OUTPUT.PUT_LINE('Transaction: ' || t.TransactionID || ', Account: ' || t.AccountID);
  6    END LOOP;
  7  END;
  8  /

PL/SQL procedure successfully completed.
```

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.

CODE:

```
DECLARE
 CURSOR c_accounts IS SELECT * FROM Accounts;
BEGIN
 FOR a IN c_accounts LOOP
  UPDATE Accounts SET Balance = Balance - 100 WHERE AccountID = a.AccountID;
 END LOOP;
END;
/
```

OUTPUT:

```
SQL> DECLARE
  2    CURSOR c_accounts IS SELECT * FROM Accounts;
  3  BEGIN
  4    FOR a IN c_accounts LOOP
  5      UPDATE Accounts SET Balance = Balance - 100 WHERE AccountID = a.AccountID;
  6    END LOOP;
  7  END;
  8  /

PL/SQL procedure successfully completed.
```

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.

CODE:

```
DECLARE
 CURSOR c_loans IS SELECT * FROM Loans;
BEGIN
 FOR l IN c_loans LOOP
  UPDATE Loans SET InterestRate = InterestRate * 1.05 WHERE LoanID = l.LoanID;
 END LOOP;
END;
/
```

OUTPUT:

```
SQL> DECLARE
  2    CURSOR c_loans IS SELECT * FROM Loans;
  3  BEGIN
  4    FOR l IN c_loans LOOP
  5      UPDATE Loans SET InterestRate = InterestRate * 1.05 WHERE LoanID = l.LoanID;
  6    END LOOP;
  7  END;
  8  /

PL/SQL procedure successfully completed.
```

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.

CODE:

```
CREATE OR REPLACE PACKAGE CustomerManagement AS
 PROCEDURE AddCustomer(p_id NUMBER, p_name VARCHAR2, p_dob DATE, p_balance
    NUMBER);
 PROCEDURE UpdateCustomerDetails(p_id NUMBER, p_name VARCHAR2);
 FUNCTION GetBalance(p_id NUMBER) RETURN NUMBER;
END CustomerManagement;
/

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS
 PROCEDURE AddCustomer(p_id NUMBER, p_name VARCHAR2, p_dob DATE, p_balance
    NUMBER) IS
 BEGIN
  INSERT INTO Customers VALUES (p_id, p_name, p_dob, p_balance, SYSDATE);
 END;

 PROCEDURE UpdateCustomerDetails(p_id NUMBER, p_name VARCHAR2) IS
 BEGIN
  UPDATE Customers SET Name = p_name WHERE CustomerID = p_id;
 END;

 FUNCTION GetBalance(p_id NUMBER) RETURN NUMBER IS
  v_balance NUMBER;
 BEGIN
  SELECT Balance INTO v_balance FROM Customers WHERE CustomerID = p_id;
  RETURN v_balance;
```

```
  END;
END CustomerManagement;
/
```

OUTPUT:

```
SQL> CREATE OR REPLACE PACKAGE BODY CustomerManagement AS
  2
  3    PROCEDURE AddCustomer(p_id NUMBER, p_name VARCHAR2, p_dob DATE, p_balance NUMBER) IS
  4    BEGIN
  5      INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)
  6      VALUES (p_id, p_name, p_dob, p_balance, SYSDATE);
  7    END;
  8
  9    PROCEDURE UpdateCustomerDetails(p_id NUMBER, p_name VARCHAR2) IS
 10    BEGIN
 11      UPDATE Customers SET Name = p_name WHERE CustomerID = p_id;
 12    END;
 13
 14    FUNCTION GetBalance(p_id NUMBER) RETURN NUMBER IS
 15      v_balance NUMBER;
 16    BEGIN
 17      SELECT Balance INTO v_balance FROM Customers WHERE CustomerID = p_id;
 18      RETURN v_balance;
 19    END;
 20
 21  END CustomerManagement;
 22  /

Package body created.
```

Scenario 2: Create a package to manage employee data.

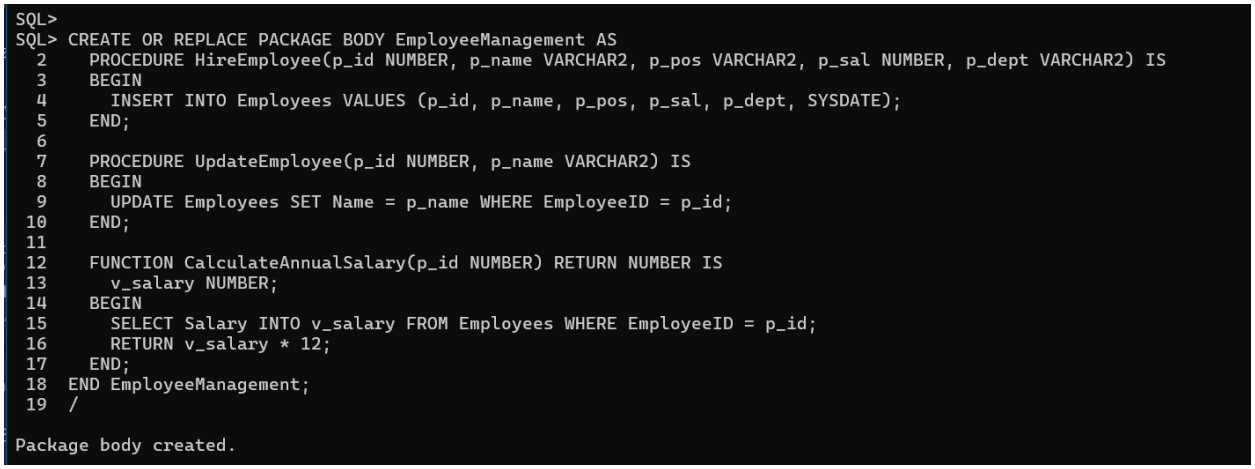
- **Question:** Write a package **EmployeeManagement** with procedures to hire new employees, update employee details, and a function to calculate annual salary.

CODE:

```
CREATE OR REPLACE PACKAGE EmployeeManagement AS
  PROCEDURE HireEmployee(p_id NUMBER, p_name VARCHAR2, p_pos VARCHAR2,
p_sal NUMBER, p_dept VARCHAR2);
  PROCEDURE UpdateEmployee(p_id NUMBER, p_name VARCHAR2);
  FUNCTION CalculateAnnualSalary(p_id NUMBER) RETURN NUMBER;
END EmployeeManagement;
/

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS
  PROCEDURE HireEmployee(p_id NUMBER, p_name VARCHAR2, p_pos VARCHAR2,
p_sal NUMBER, p_dept VARCHAR2) IS
  BEGIN
    INSERT INTO Employees VALUES (p_id, p_name, p_pos, p_sal, p_dept, SYSDATE);
  END;

  PROCEDURE UpdateEmployee(p_id NUMBER, p_name VARCHAR2) IS
  BEGIN
    UPDATE Employees SET Name = p_name WHERE EmployeeID = p_id;
```

```
 END;

 FUNCTION CalculateAnnualSalary(p_id NUMBER) RETURN NUMBER IS
  v_salary NUMBER;
 BEGIN
  SELECT Salary INTO v_salary FROM Employees WHERE EmployeeID = p_id;
  RETURN v_salary * 12;
 END;
END EmployeeManagement;
/
```

OUTPUT:



```
SQL>
SQL> CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS
  2    PROCEDURE HireEmployee(p_id NUMBER, p_name VARCHAR2, p_pos VARCHAR2, p_sal NUMBER, p_dept VARCHAR2) IS
  3    BEGIN
  4      INSERT INTO Employees VALUES (p_id, p_name, p_pos, p_sal, p_dept, SYSDATE);
  5    END;
  6
  7    PROCEDURE UpdateEmployee(p_id NUMBER, p_name VARCHAR2) IS
  8    BEGIN
  9      UPDATE Employees SET Name = p_name WHERE EmployeeID = p_id;
 10    END;
 11
 12    FUNCTION CalculateAnnualSalary(p_id NUMBER) RETURN NUMBER IS
 13      v_salary NUMBER;
 14    BEGIN
 15      SELECT Salary INTO v_salary FROM Employees WHERE EmployeeID = p_id;
 16      RETURN v_salary * 12;
 17    END;
 18  END EmployeeManagement;
 19  /

Package body created.
```

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

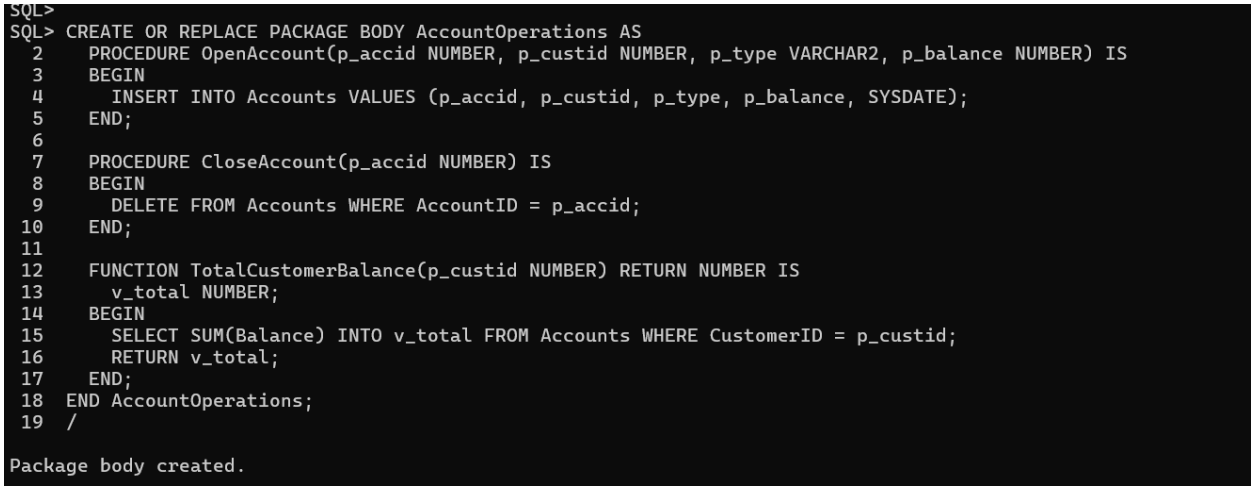
- **Question:** Create a package **AccountOperations** with procedures for opening a new account, closing an account, and a function to get the total balance of a customer across all accounts.

CODE:

```
CREATE OR REPLACE PACKAGE AccountOperations AS
 PROCEDURE OpenAccount(p_accid NUMBER, p_custid NUMBER, p_type VARCHAR2, p_balance
    NUMBER);
 PROCEDURE CloseAccount(p_accid NUMBER);
 FUNCTION TotalCustomerBalance(p_custid NUMBER) RETURN NUMBER;
END AccountOperations;
/

CREATE OR REPLACE PACKAGE BODY AccountOperations AS
 PROCEDURE OpenAccount(p_accid NUMBER, p_custid NUMBER, p_type VARCHAR2, p_balance
    NUMBER) IS
 BEGIN
  INSERT INTO Accounts VALUES (p_accid, p_custid, p_type, p_balance, SYSDATE);
 END;
```

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

  FUNCTION TotalCustomerBalance(p_custid NUMBER) RETURN NUMBER IS
    v_total NUMBER;
  BEGIN
    SELECT SUM(Balance) INTO v_total FROM Accounts WHERE CustomerID = p_custid;
    RETURN v_total;
  END;
END AccountOperations;
/
```

OUTPUT:



```
SQL>
SQL> CREATE OR REPLACE PACKAGE BODY AccountOperations AS
  2    PROCEDURE OpenAccount(p_accid NUMBER, p_custid NUMBER, p_type VARCHAR2, p_balance NUMBER) IS
  3    BEGIN
  4      INSERT INTO Accounts VALUES (p_accid, p_custid, p_type, p_balance, SYSDATE);
  5    END;
  6
  7    PROCEDURE CloseAccount(p_accid NUMBER) IS
  8    BEGIN
  9      DELETE FROM Accounts WHERE AccountID = p_accid;
 10    END;
 11
 12    FUNCTION TotalCustomerBalance(p_custid NUMBER) RETURN NUMBER IS
 13      v_total NUMBER;
 14    BEGIN
 15      SELECT SUM(Balance) INTO v_total FROM Accounts WHERE CustomerID = p_custid;
 16      RETURN v_total;
 17    END;
 18  END AccountOperations;
 19  /

Package body created.
```

Schema to be Created

```
CREATE TABLE Customers (
   CustomerID NUMBER PRIMARY KEY,
   Name VARCHAR2(100),
   DOB DATE,
   Balance NUMBER,
   LastModified DATE
);

CREATE TABLE Accounts (
   AccountID NUMBER PRIMARY KEY,
   CustomerID NUMBER,
```

```
    AccountType VARCHAR2(20),
    Balance NUMBER,
    LastModified DATE,
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);

CREATE TABLE Transactions (
    TransactionID NUMBER PRIMARY KEY,
    AccountID NUMBER,
    TransactionDate DATE,
    Amount NUMBER,
    TransactionType VARCHAR2(10),
    FOREIGN KEY (AccountID) REFERENCES Accounts(AccountID)
);

CREATE TABLE Loans (
    LoanID NUMBER PRIMARY KEY,
    CustomerID NUMBER,
    LoanAmount NUMBER,
    InterestRate NUMBER,
    StartDate DATE,
    EndDate DATE,
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);

CREATE TABLE Employees (
    EmployeeID NUMBER PRIMARY KEY,
    Name VARCHAR2(100),
    Position VARCHAR2(50),
    Salary NUMBER,
    Department VARCHAR2(50),
    HireDate DATE
);
```

Example Scripts for Sample Data Insertion

```
INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)
VALUES (1, 'John Doe', TO_DATE('1985-05-15', 'YYYY-MM-DD'), 1000, SYSDATE);

INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)
VALUES (2, 'Jane Smith', TO_DATE('1990-07-20', 'YYYY-MM-DD'), 1500, SYSDATE);
```

INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified) VALUES (1, 1, 'Savings', 1000, SYSDATE);

INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified) VALUES (2, 2, 'Checking', 1500, SYSDATE);

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType) VALUES (1, 1, SYSDATE, 200, 'Deposit');

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType) VALUES (2, 2, SYSDATE, 300, 'Withdrawal');

INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate) VALUES (1, 1, 5000, 5, SYSDATE, ADD_MONTHS(SYSDATE, 60));

INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate) VALUES (1, 'Alice Johnson', 'Manager', 70000, 'HR', TO_DATE('2015-06-15', 'YYYY-MM-DD'));

INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate) VALUES (2, 'Bob Brown', 'Developer', 60000, 'IT', TO_DATE('2017-03-20', 'YYYY-MM-DD'));