Schemas created

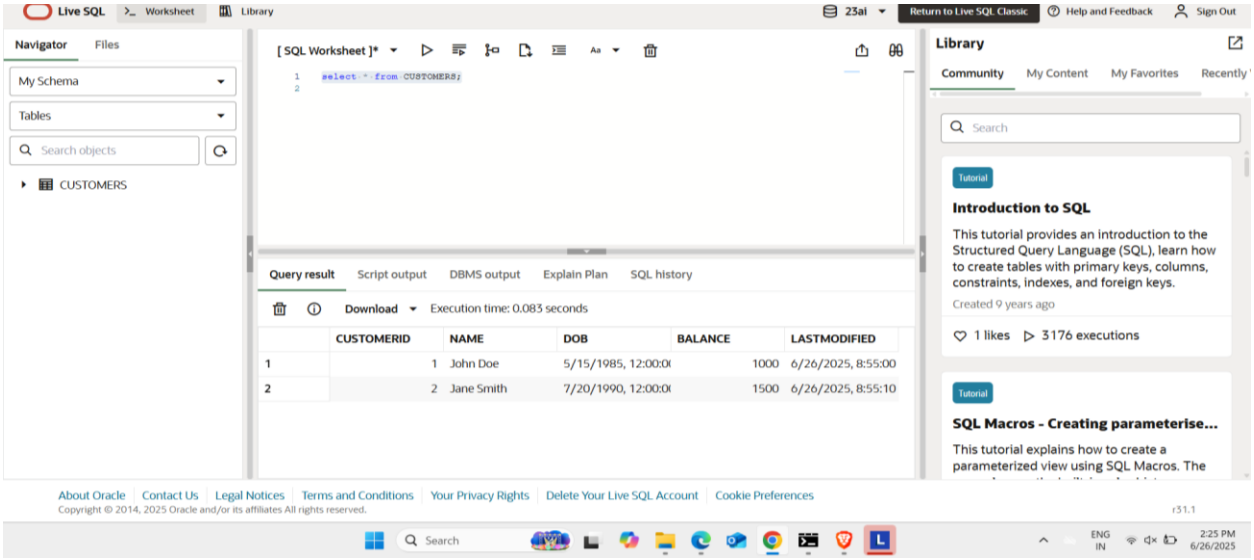
Customers Table

```
CREATE TABLE Customers (

    CustomerID      NUMBER PRIMARY KEY,

    Name            VARCHAR2(100),

    DOB             DATE,

    Balance         NUMBER,

    LastModified    DATE

);
```

```
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);
```

OUTPUT

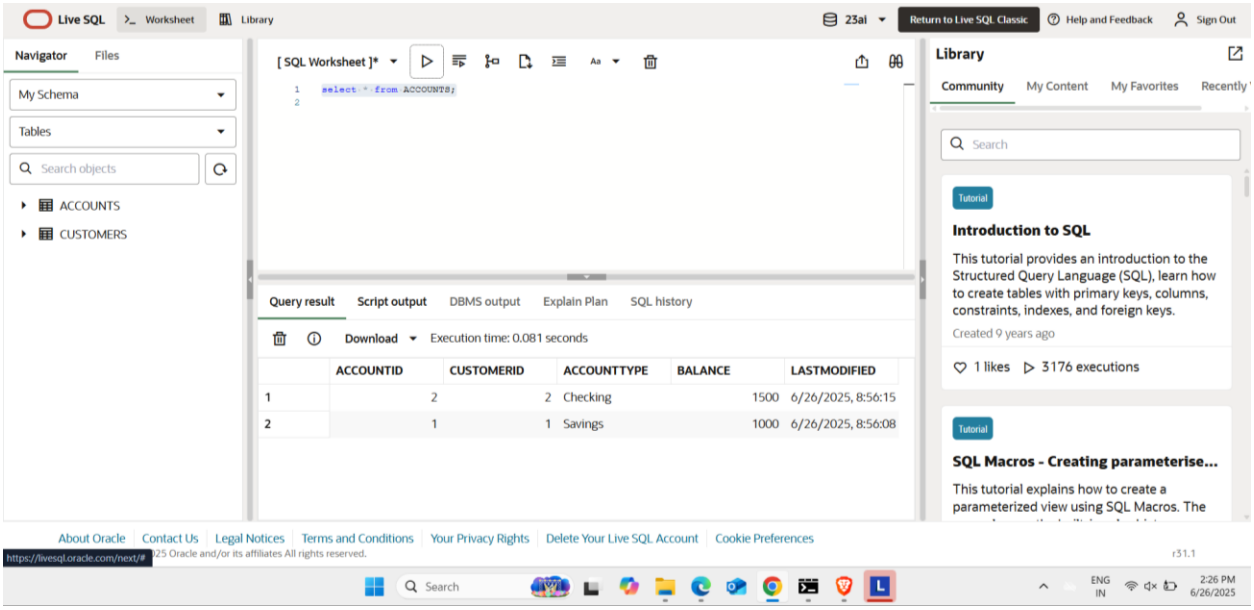


Accounts Table

```
CREATE TABLE Accounts (

    AccountID       NUMBER PRIMARY KEY,

    CustomerID      NUMBER,
```

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

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);
```

OUTPUT

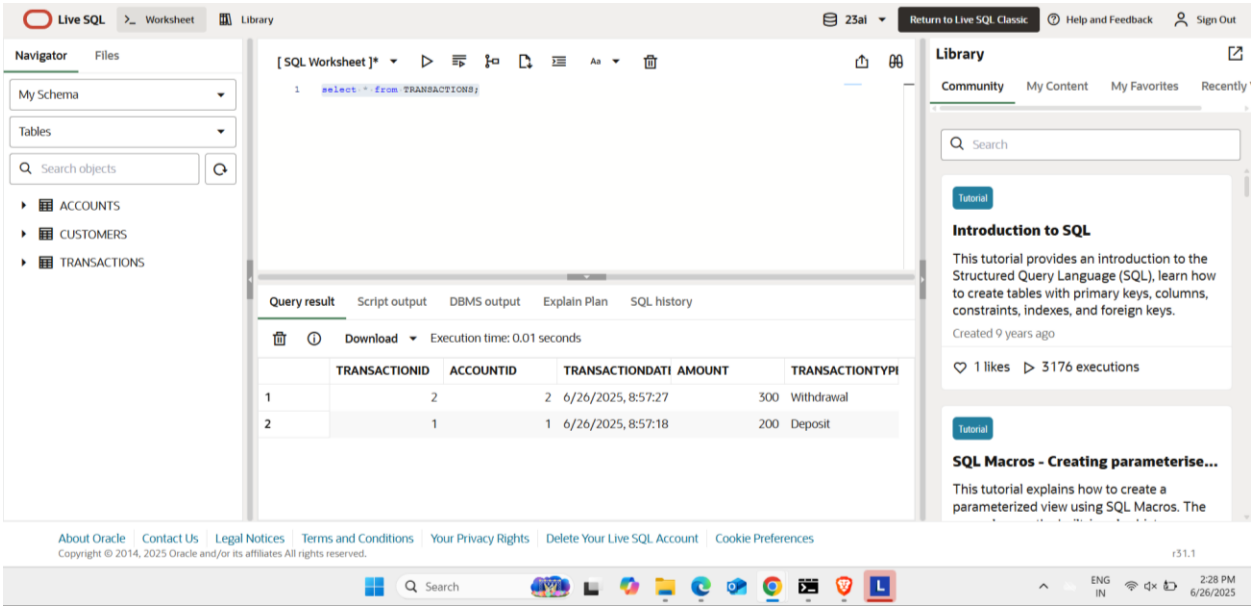


Transactions Table

```
CREATE TABLE Transactions (
    TransactionID     NUMBER PRIMARY KEY,
    AccountID         NUMBER,
    TransactionDate   DATE,
    Amount            NUMBER,
    TransactionType   VARCHAR2(10),
```

```
    FOREIGN KEY (AccountID) REFERENCES Accounts(AccountID)

);

NSERT 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');
```

OUTPUT

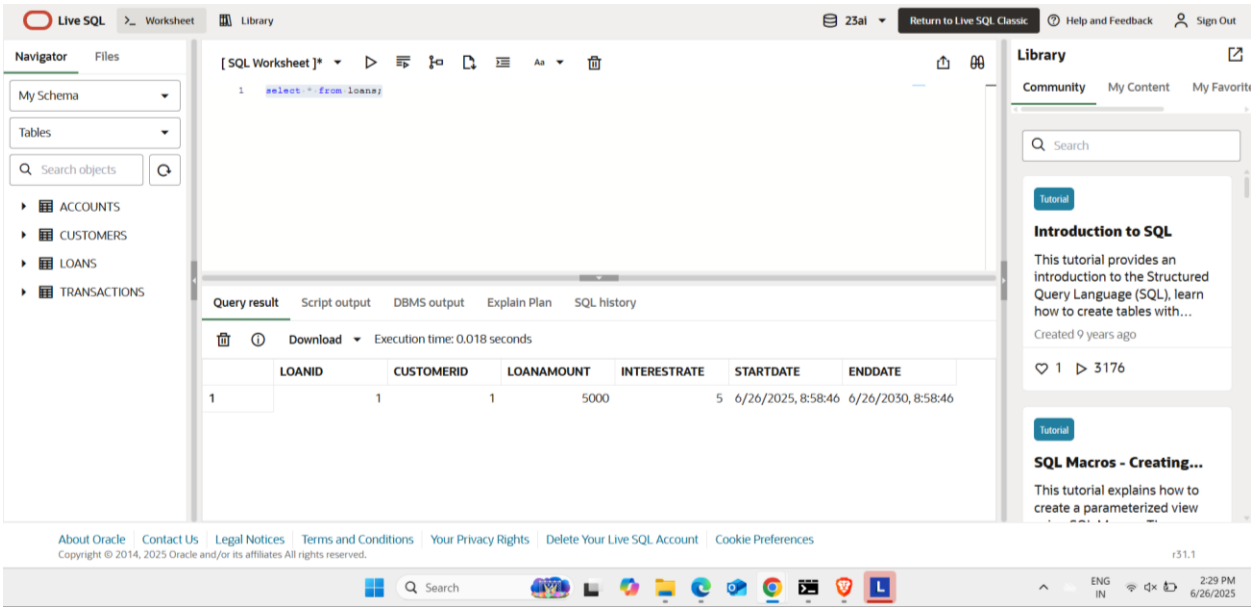


Loans Table

```
CREATE TABLE Loans (

    LoanID        NUMBER PRIMARY KEY,

    CustomerID    NUMBER,

    LoanAmount    NUMBER,

    InterestRate  NUMBER,

    StartDate     DATE,

    EndDate       DATE,

    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);
```

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

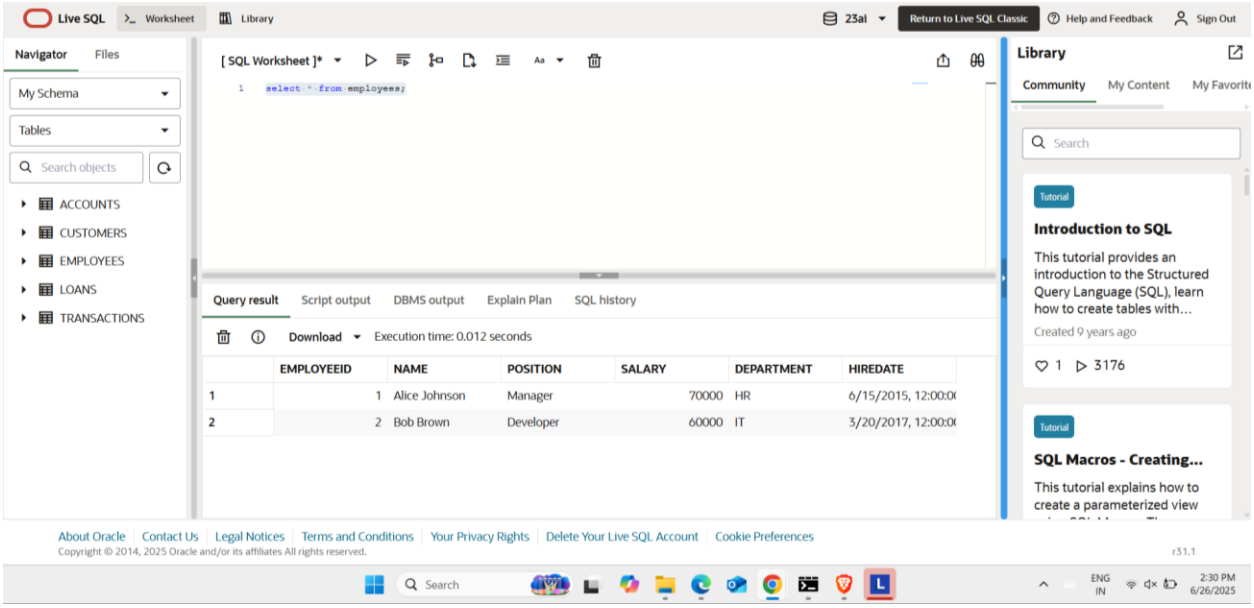
OUTPUT



Employees Table

```
CREATE TABLE Employees (

    EmployeeID   NUMBER PRIMARY KEY,

    Name         VARCHAR2(100),

    Position     VARCHAR2(50),

    Salary       NUMBER,

    Department   VARCHAR2(50),

    HireDate     DATE

);

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'));
```

OUTPUT



EXERCISE 1 – CONTROL STRUCTURES

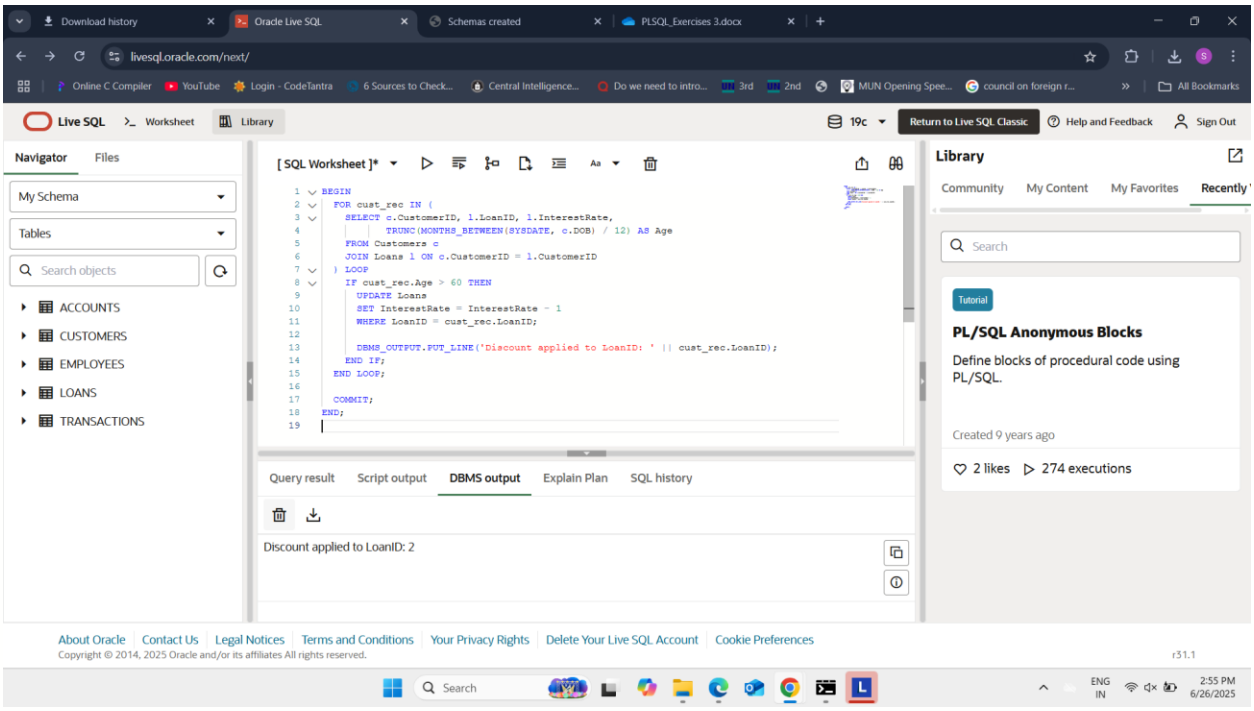
Scenario 1: The bank wants to apply a discount to loan interest rates for customers above 60 years old.
Question: Write a PL/SQL block that loops through all customers, checks their age, and if they are above 60, apply a 1% discount to their current loan interest rates.

Customer over 60

```
INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified) VALUES (3, 'Elder Johnson', TO_DATE('1950-03-10', 'YYYY-MM-DD'), 8000, SYSDATE);
```

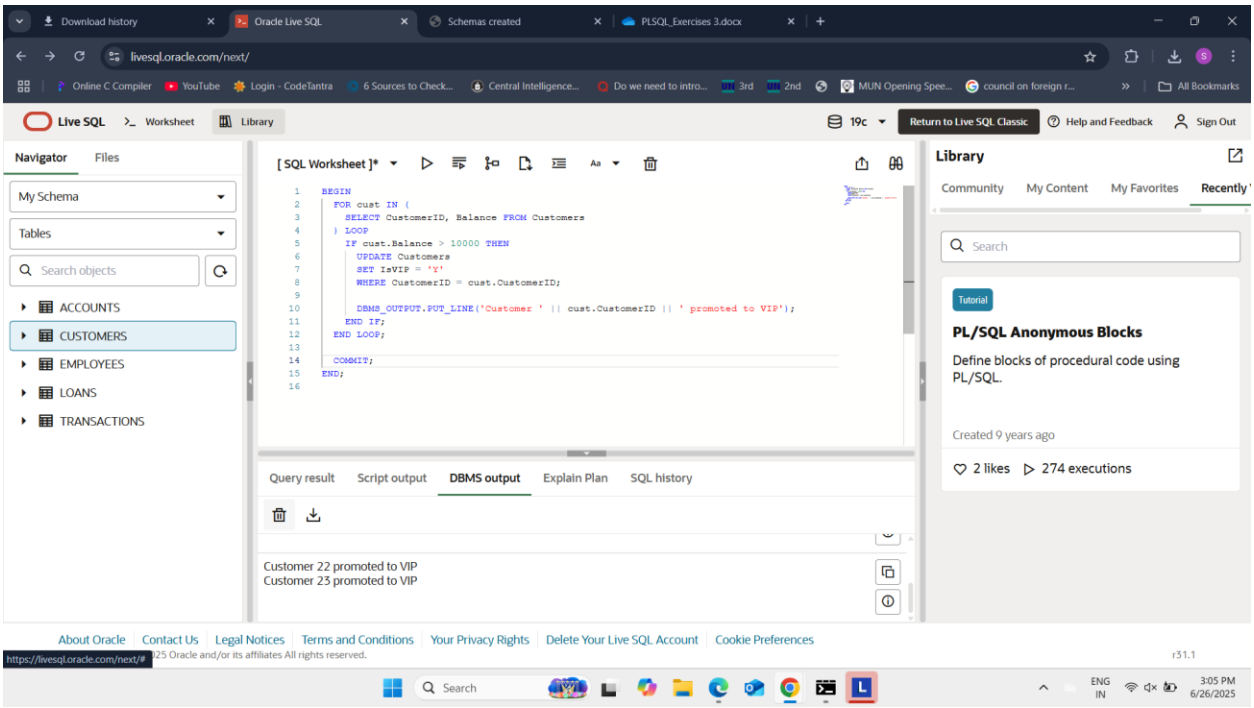
Loan for Elder Johnson

```
INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate)
VALUES (2, 3, 10000, 6, SYSDATE, ADD_MONTHS(SYSDATE, 60));

BEGIN
  FOR cust_rec IN (
    SELECT c.CustomerID, l.LoanID, l.InterestRate,
           TRUNC(MONTHS_BETWEEN(SYSDATE, c.DOB) / 12) AS Age
    FROM Customers c
    JOIN Loans l ON c.CustomerID = l.CustomerID
  ) LOOP
    IF cust_rec.Age > 60 THEN
      UPDATE Loans
      SET InterestRate = InterestRate - 1
      WHERE LoanID = cust_rec.LoanID;

      DBMS_OUTPUT.PUT_LINE('Discount applied to LoanID: ' || cust_rec.LoanID);
    END IF;
  END LOOP;

  COMMIT;
END;
```



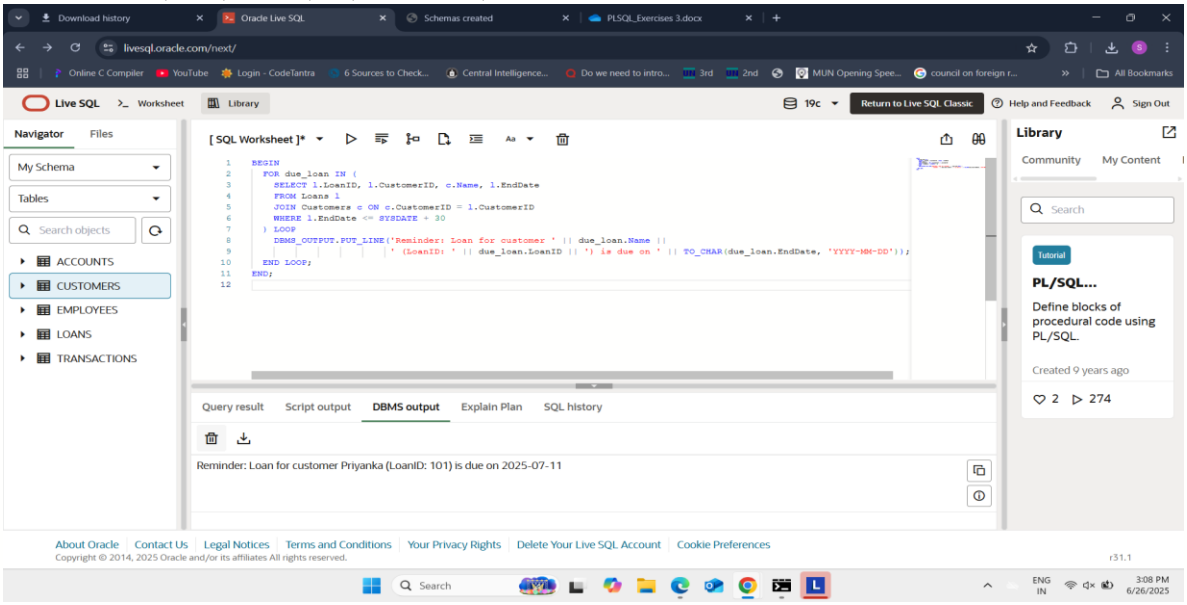
Scenario 2: A customer can be promoted to VIP status based on their balance.
Question: Write a PL/SQL block that iterates through all customers and sets a flag IsVIP to TRUE for those with a balance over $10,000.

```
ALTER TABLE Customers ADD IsVIP CHAR(1);

INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)
VALUES (21, 'Lasya', TO_DATE('1992-05-10', 'YYYY-MM-DD'), 7000, SYSDATE);

INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)
VALUES (22, 'Sree', TO_DATE('1985-11-25', 'YYYY-MM-DD'), 15000, SYSDATE);

INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)
VALUES (23, 'Priyanka', TO_DATE('1990-03-18', 'YYYY-MM-DD'), 12000, SYSDATE);

BEGIN
  FOR cust IN (
    SELECT CustomerID, Balance FROM Customers
  ) LOOP
    IF cust.Balance > 10000 THEN
      UPDATE Customers
      SET IsVIP = 'Y'
      WHERE CustomerID = cust.CustomerID;

      DBMS_OUTPUT.PUT_LINE('Customer ' || cust.CustomerID || ' promoted to VIP');
    END IF;
  END LOOP;

  COMMIT;
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.

```
INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate)
VALUES (101, 23, 8000, 5, SYSDATE, SYSDATE + 15);
```



EXERCISE 2: ERROR HANDLING

Scenario 1: Handle exceptions during fund transfers between accounts.

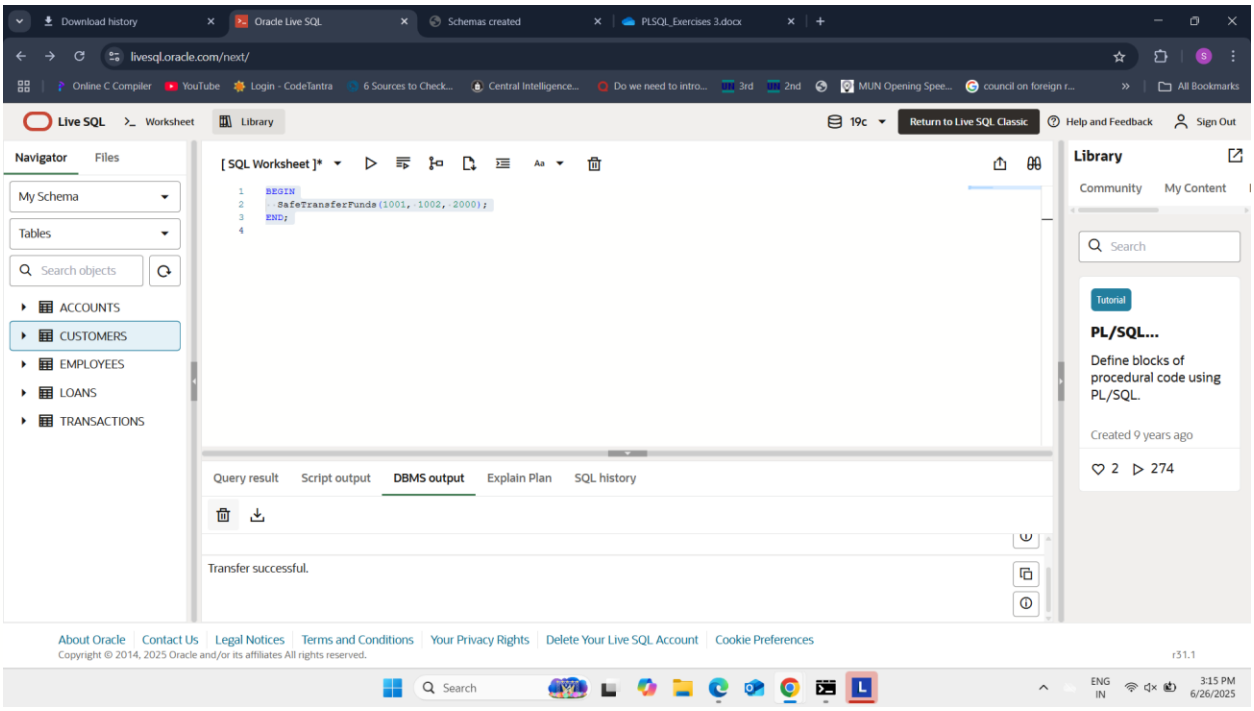
Question: Write a stored procedure SafeTransferFunds that transfers funds between two accounts. Ensure that if any error occurs (e.g., insufficient funds), an appropriate error message is logged and the transaction is rolled back.

-- Source account with enough balance

```
INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified)
VALUES (1001, 1, 'Savings', 5000, SYSDATE);
```

-- Destination account INSERT INTO Accounts

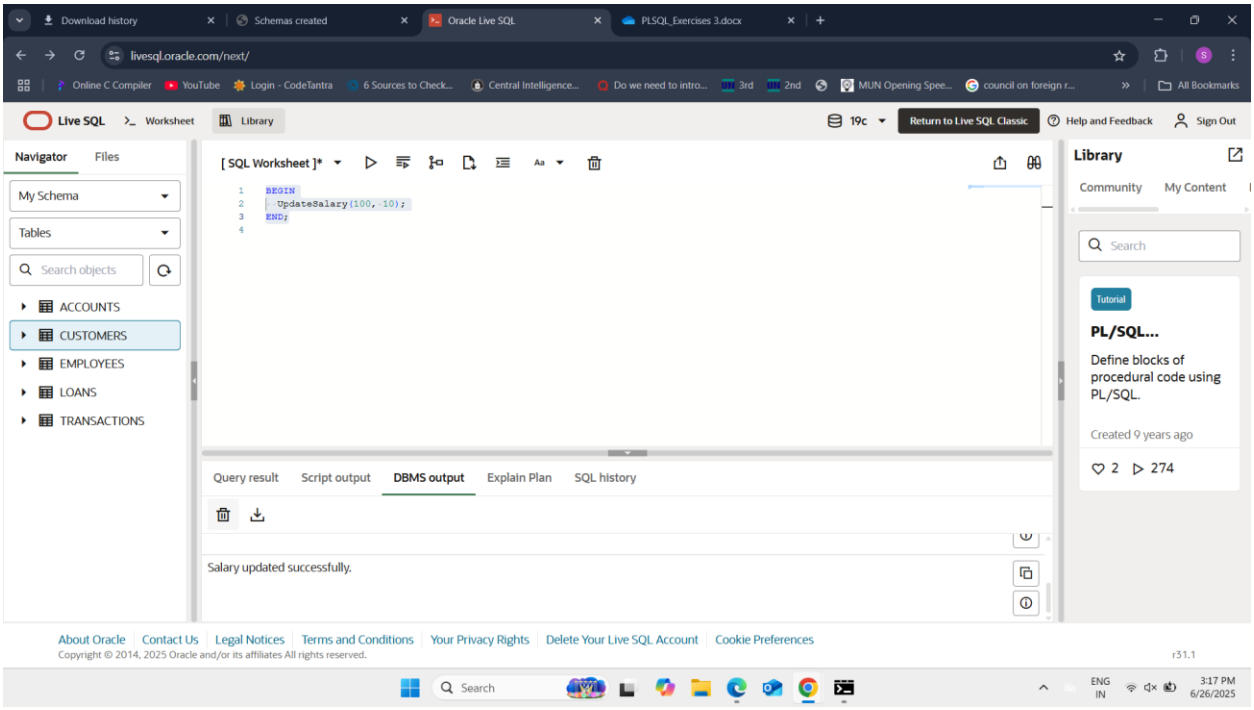
```
INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified)
VALUES (1002, 2, 'Savings', 3000, SYSDATE);

CREATE OR REPLACE PROCEDURE SafeTransferFunds (
  p_from_account_id IN NUMBER,
  p_to_account_id   IN NUMBER,
  p_amount          IN NUMBER
) AS
  v_balance NUMBER;
BEGIN
  -- Check balance
  SELECT Balance INTO v_balance
  FROM Accounts
  WHERE AccountID = p_from_account_id;

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

  -- Debit source account
  UPDATE Accounts
  SET Balance = Balance - p_amount,
      LastModified = SYSDATE
  WHERE AccountID = p_from_account_id;

  -- Credit destination account
  UPDATE Accounts
  SET Balance = Balance + p_amount,
      LastModified = SYSDATE
  WHERE AccountID = p_to_account_id;

  COMMIT;

  DBMS_OUTPUT.PUT_LINE('Transfer successful.');
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
    DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
END;
BEGIN
  SafeTransferFunds(1001, 1002, 2000);
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.

```
INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate)
VALUES (100, 'Lasya', 'Analyst', 40000, 'Finance', TO_DATE('2020-01-01', 'YYYY-MM-
DD'));

CREATE OR REPLACE PROCEDURE UpdateSalary (
  p_emp_id      IN NUMBER,
  p_percentage  IN NUMBER
) AS
  v_exists NUMBER;
BEGIN
  -- Check if employee exists
  SELECT COUNT(*) INTO v_exists
  FROM Employees
  WHERE EmployeeID = p_emp_id;

  IF v_exists = 0 THEN
    RAISE_APPLICATION_ERROR(-20002, 'Employee ID not found.');
  END IF;

  -- Update salary
  UPDATE Employees
  SET Salary = Salary + (Salary * p_percentage / 100)
  WHERE EmployeeID = p_emp_id;

  COMMIT;

  DBMS_OUTPUT.PUT_LINE('Salary updated successfully.');
EXCEPTION
```

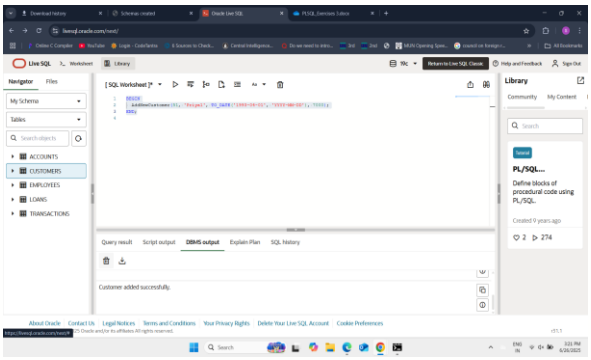
```
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
    ROLLBACK;
END;

BEGIN
  UpdateSalary(100, 10);
END;
```

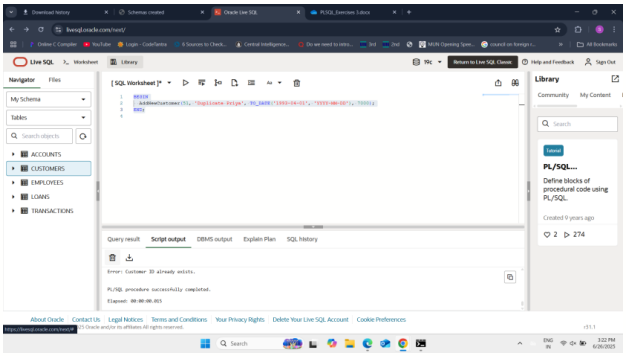


Scenario 3: Ensure data integrity when adding a new customer.
Question: Write a stored procedure AddNewCustomer that inserts a new customer into the Customers table. If a customer with the same ID already exists, handle the exception by logging an error and preventing the insertion.

```
CREATE OR REPLACE PROCEDURE AddNewCustomer (
  p_customer_id   IN NUMBER,
  p_name          IN VARCHAR2,
  p_dob           IN DATE,
  p_balance       IN NUMBER
) AS
BEGIN
  INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)
  VALUES (p_customer_id, p_name, p_dob, p_balance, SYSDATE);

  COMMIT;

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

```
BEGIN
  AddNewCustomer(50, 'Priya', TO_DATE('1993-04-01', 'YYYY-MM-DD'), 7000);
END;

BEGIN
  AddNewCustomer(50, 'Duplicate Priya', TO_DATE('1990-01-01', 'YYYY-MM-DD'), 9000);
END;
```

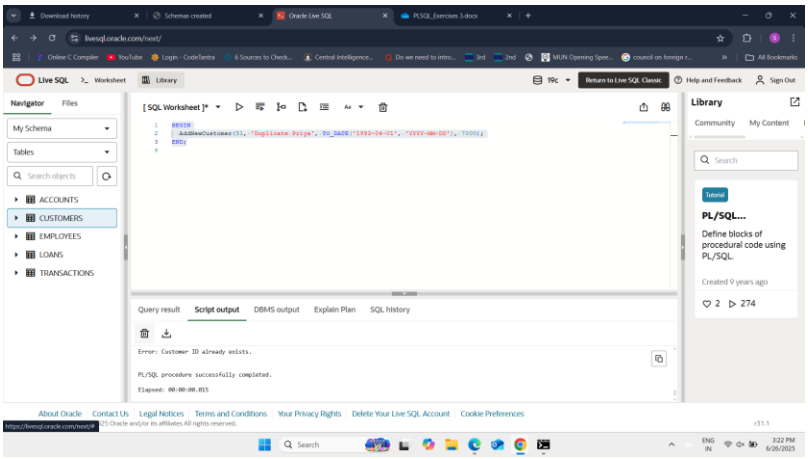




EXERCISE 3: STORED PROCEDURES

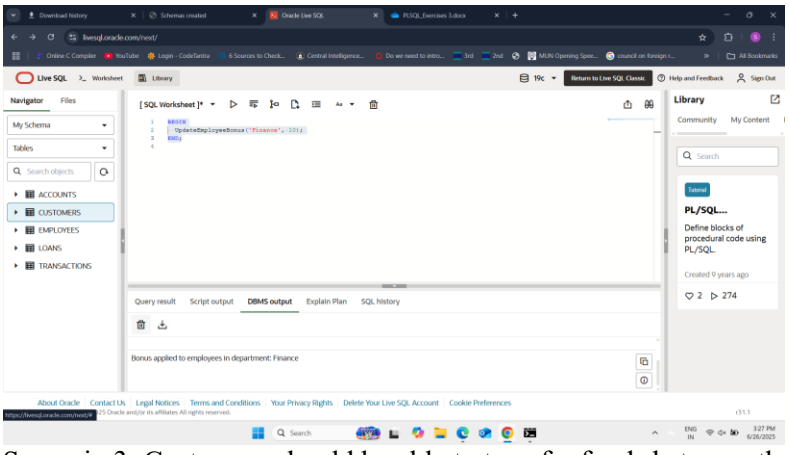
Scenario 1: The bank needs to process monthly interest for all savings accounts.
Question: Write a stored procedure ProcessMonthlyInterest that calculates and updates the balance of all savings accounts by applying an interest rate of 1% to the current balance.

```
BEGIN
  ProcessMonthlyInterest;
END;
CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest AS
BEGIN
  FOR acc IN (
    SELECT AccountID, Balance
    FROM Accounts
    WHERE AccountType = 'Savings'
  ) LOOP
    UPDATE Accounts
    SET Balance = Balance + (Balance * 0.01),
        LastModified = SYSDATE
    WHERE AccountID = acc.AccountID;

    DBMS_OUTPUT.PUT_LINE('Interest added to AccountID: ' || acc.AccountID);
  END LOOP;

  COMMIT;
END;


BEGIN
  ProcessMonthlyInterest;
END;
```



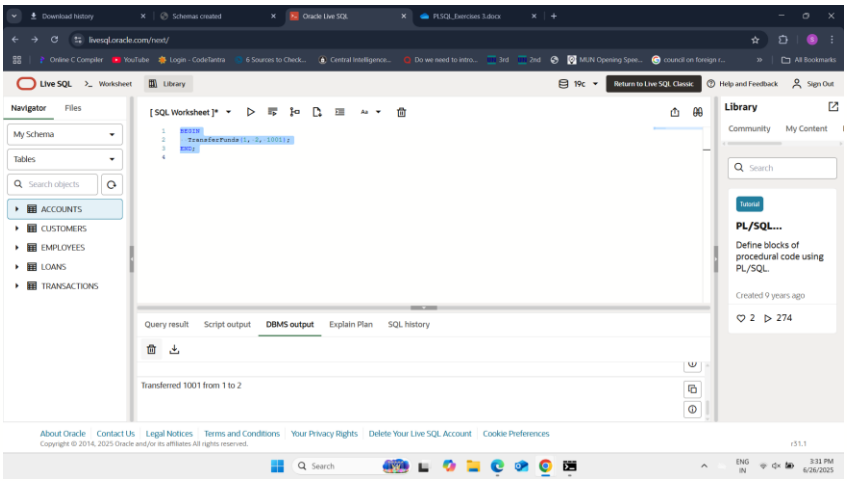
Scenario 2: The bank wants to implement a bonus scheme for employees based on their performance.
Question: Write a stored procedure UpdateEmployeeBonus that updates the salary of employees in a given department by adding a bonus percentage passed as a parameter.

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

  DBMS_OUTPUT.PUT_LINE('Bonus applied to employees in department: ' || p_department);

  COMMIT;
END;

BEGIN
  UpdateEmployeeBonus('Finance', 10);
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.

```
CREATE OR REPLACE PROCEDURE TransferFunds (
  p_from_account_id IN NUMBER,
  p_to_account_id   IN NUMBER,
  p_amount          IN NUMBER
) AS
  v_balance NUMBER;
BEGIN
  SELECT Balance INTO v_balance
  FROM Accounts
  WHERE AccountID = p_from_account_id;
  IF v_balance < p_amount THEN
    RAISE_APPLICATION_ERROR(-20001, 'Insufficient balance in source account.');
  END IF;
  UPDATE Accounts
  SET Balance = Balance - p_amount,
      LastModified = SYSDATE
  WHERE AccountID = p_from_account_id;
  UPDATE Accounts
  SET Balance = Balance + p_amount,
      LastModified = SYSDATE
  WHERE AccountID = p_to_account_id;
  COMMIT;
  DBMS_OUTPUT.PUT_LINE('Transferred ' || p_amount || ' from ' || p_from_account_id ||
' to ' || p_to_account_id);
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
    DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
END;

BEGIN
  TransferFunds(1, 2, 1001);
END;
```



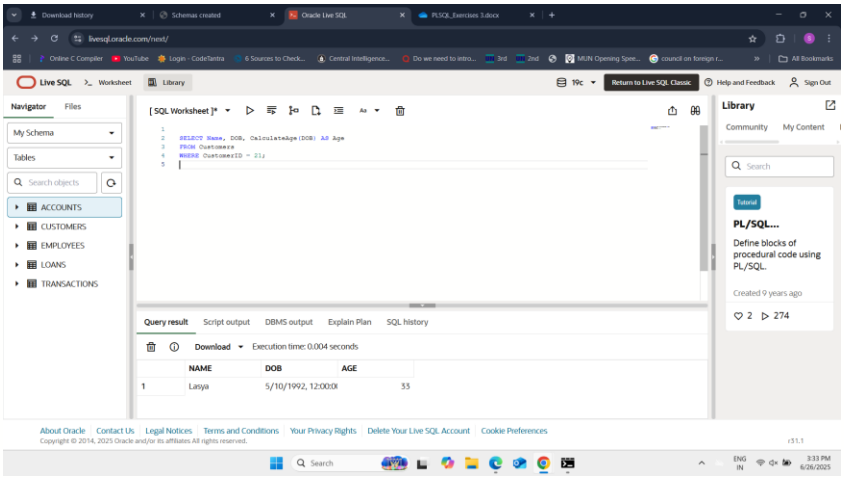
EXERCISE 4: FUNCTIONS

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.

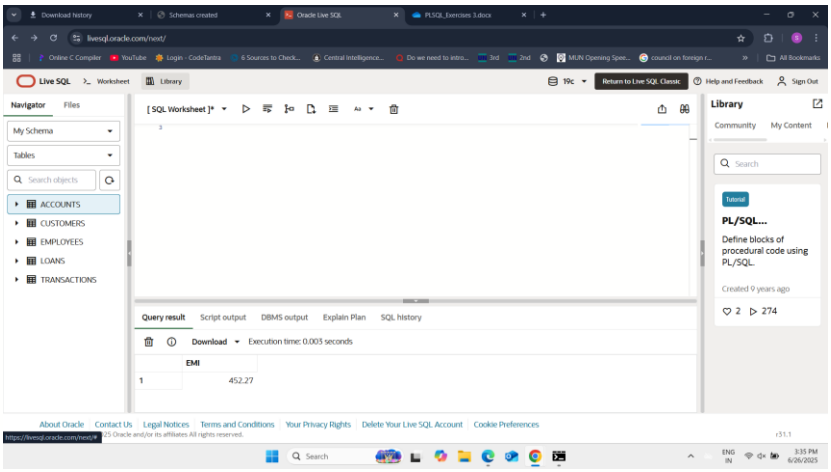
```
CREATE OR REPLACE FUNCTION CalculateAge (
  p_dob IN DATE
) RETURN NUMBER IS
  v_age NUMBER;
```

```
BEGIN
  v_age := TRUNC(MONTHS_BETWEEN(SYSDATE, p_dob) / 12);
  RETURN v_age;
END;


SELECT Name, DOB, CalculateAge(DOB) AS Age
FROM Customers
WHERE CustomerID = 21;
```



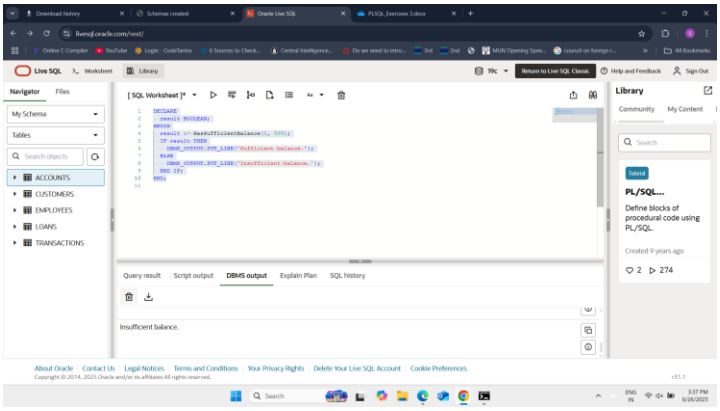
Scenario 2: The bank needs to compute the monthly installment for a loan.
Question: Write a function CalculateMonthlyInstallment that takes the loan amount, interest rate, and loan duration in years as input and returns the monthly installment amount.

```
CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment (
  p_loan_amount   IN NUMBER,
  p_annual_rate   IN NUMBER,
  p_years         IN NUMBER
) RETURN NUMBER IS
  v_r   NUMBER := p_annual_rate / 12 / 100;
  v_n   NUMBER := p_years * 12;
  v_emi NUMBER;
BEGIN
  v_emi := (p_loan_amount * v_r * POWER(1 + v_r, v_n)) /
           (POWER(1 + v_r, v_n) - 1);
  RETURN ROUND(v_emi, 2);
END;


SELECT CalculateMonthlyInstallment(10000, 8, 2) AS EMI FROM DUAL;
```



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. give everything for this as well

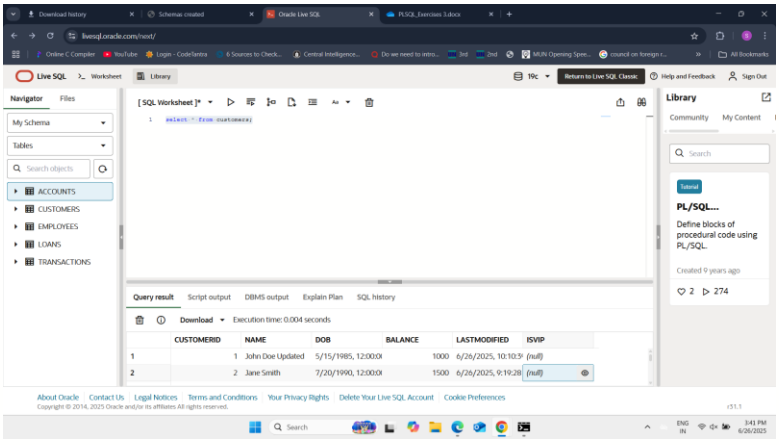
```
CREATE OR REPLACE FUNCTION HasSufficientBalance (
  p_account_id IN NUMBER,
  p_amount     IN NUMBER
) RETURN BOOLEAN IS
  v_balance NUMBER;
BEGIN
  SELECT Balance INTO v_balance
  FROM Accounts
  WHERE AccountID = p_account_id;

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

DECLARE
  result BOOLEAN;
BEGIN
  result := HasSufficientBalance(1, 500);
  IF result THEN
    DBMS_OUTPUT.PUT_LINE('Sufficient balance.');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Insufficient balance.');
  END IF;
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.

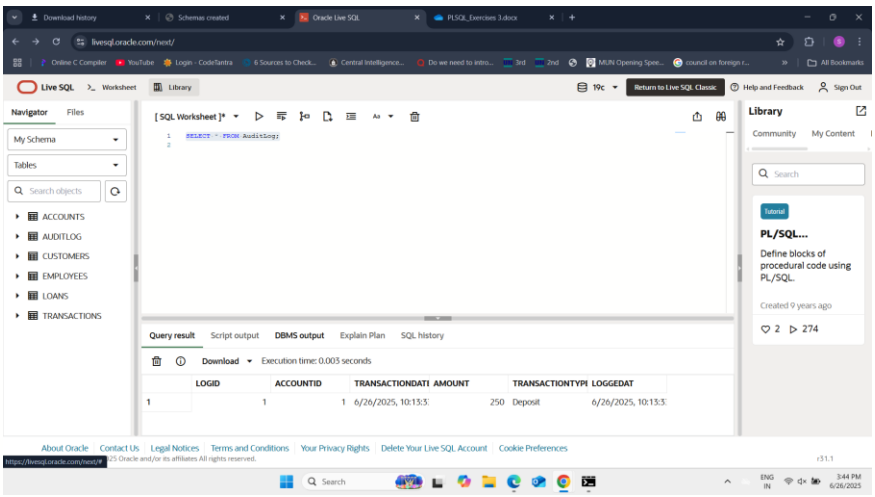
```
CREATE OR REPLACE TRIGGER UpdateCustomerLastModified
BEFORE UPDATE ON Customers
FOR EACH ROW
BEGIN
  :NEW.LastModified := SYSDATE;
END;

BEGIN
  DBMS_LOCK.SLEEP(2);
END;

UPDATE Customers
SET Name = 'John Doe Updated'
WHERE CustomerID = 1;
```



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 (
  LogID NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
```

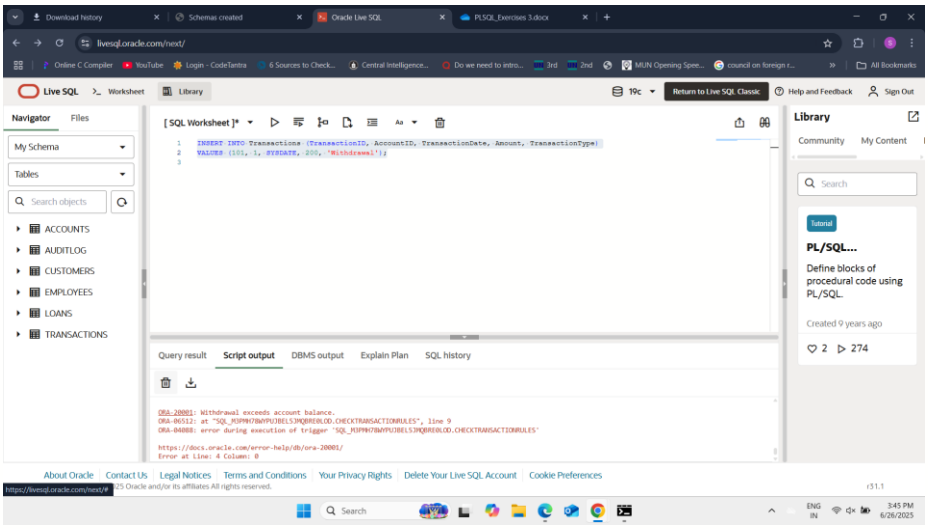
```
  AccountID NUMBER,
  TransactionDate DATE,
  Amount NUMBER,
  TransactionType VARCHAR2(10),
  LoggedAt DATE
);

CREATE OR REPLACE TRIGGER LogTransaction
AFTER INSERT ON Transactions
FOR EACH ROW
BEGIN
  INSERT INTO AuditLog (AccountID, TransactionDate, Amount, TransactionType, LoggedAt)
  VALUES (:NEW.AccountID, :NEW.TransactionDate, :NEW.Amount, :NEW.TransactionType,
SYSDATE);
END;

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount,
TransactionType)
VALUES (100, 1, SYSDATE, 250, 'Deposit');
```



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

  IF :NEW.TransactionType = 'Withdrawal' AND :NEW.Amount > v_balance THEN
    RAISE_APPLICATION_ERROR(-20001, 'Withdrawal exceeds account balance.');
  ELSIF :NEW.TransactionType = 'Deposit' AND :NEW.Amount <= 0 THEN
    RAISE_APPLICATION_ERROR(-20002, 'Deposit amount must be positive.');
  END IF;
END;
```

```
INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount,
TransactionType)
VALUES (101, 1, SYSDATE, 200, 'Withdrawal');
```

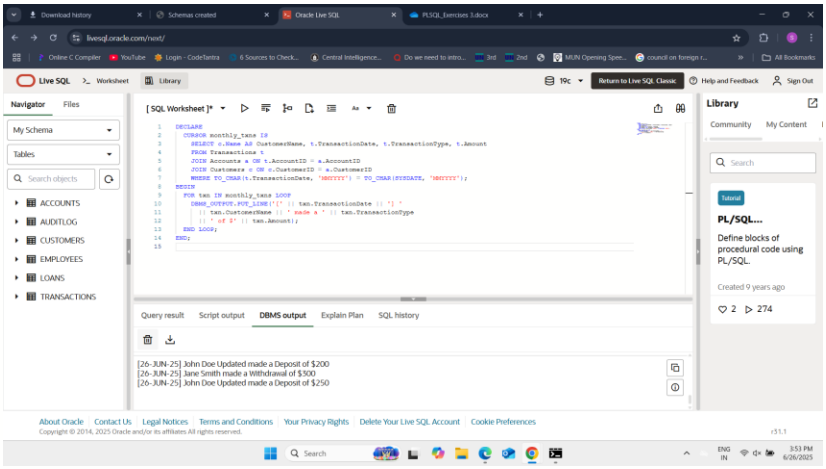


EXERCISE 6: CURSORS

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.

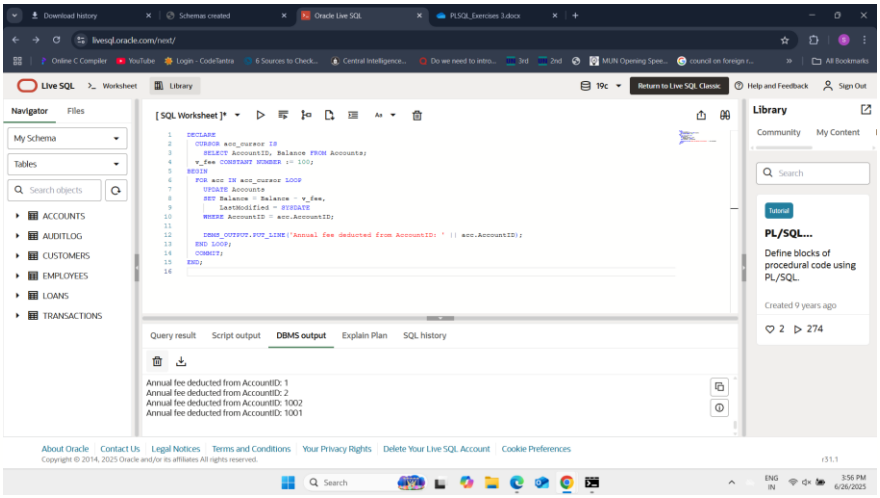
```
DECLARE
  CURSOR monthly_txns IS
    SELECT c.Name AS CustomerName, t.TransactionDate, t.TransactionType, t.Amount
    FROM Transactions t
    JOIN Accounts a ON t.AccountID = a.AccountID
    JOIN Customers c ON c.CustomerID = a.CustomerID
    WHERE TO_CHAR(t.TransactionDate, 'MMYYYY') = TO_CHAR(SYSDATE, 'MMYYYY');
BEGIN
  FOR txn IN monthly_txns LOOP
    DBMS_OUTPUT.PUT_LINE('[' || txn.TransactionDate || '] '
      || txn.CustomerName || ' made a ' || txn.TransactionType
      || ' of $' || txn.Amount);
  END LOOP;
END;
```



Scenario 2: Apply annual fee to all accounts.

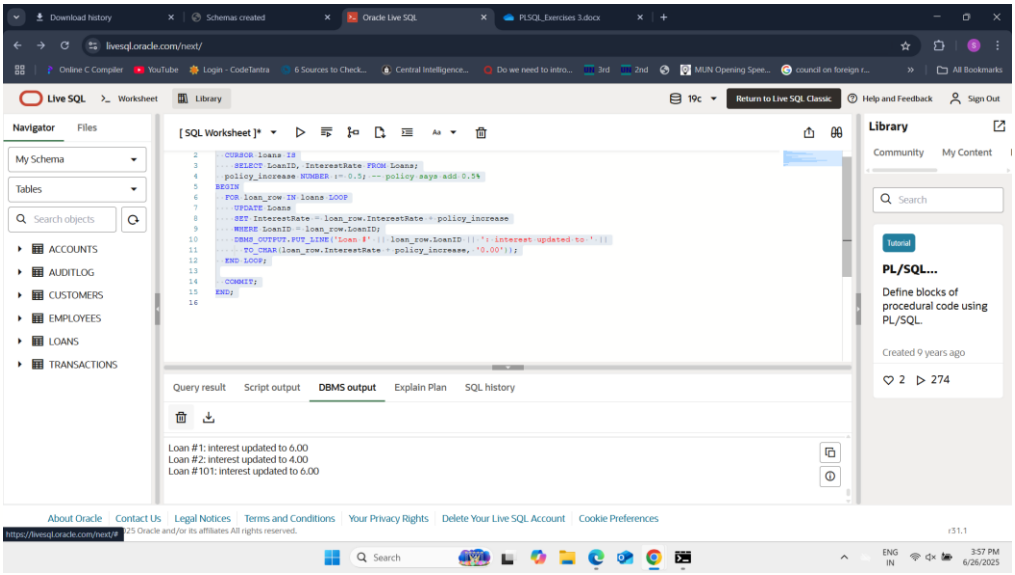
Question: Write a PL/SQL block using an explicit cursor ApplyAnnualFee that deducts an annual maintenance fee from the balance of all accounts.

```
DECLARE
  CURSOR acc_cursor IS
    SELECT AccountID, Balance FROM Accounts;
  v_fee CONSTANT NUMBER := 100;
BEGIN
  FOR acc IN acc_cursor LOOP
    UPDATE Accounts
    SET Balance = Balance - v_fee,
        LastModified = SYSDATE
    WHERE AccountID = acc.AccountID;

    DBMS_OUTPUT.PUT_LINE('Annual fee deducted from AccountID: ' || acc.AccountID);
  END LOOP;
  COMMIT;
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 loans IS
    SELECT LoanID, InterestRate FROM Loans;
  policy_increase NUMBER := 0.5; -- policy says add 0.5%
BEGIN
  FOR loan_row IN loans LOOP
    UPDATE Loans
    SET InterestRate = loan_row.InterestRate + policy_increase
    WHERE LoanID = loan_row.LoanID;
    DBMS_OUTPUT.PUT_LINE('Loan #' || loan_row.LoanID || ': interest updated to ' ||
      TO_CHAR(loan_row.InterestRate + policy_increase, '0.00'));
  END LOOP;

  COMMIT;
END;
```



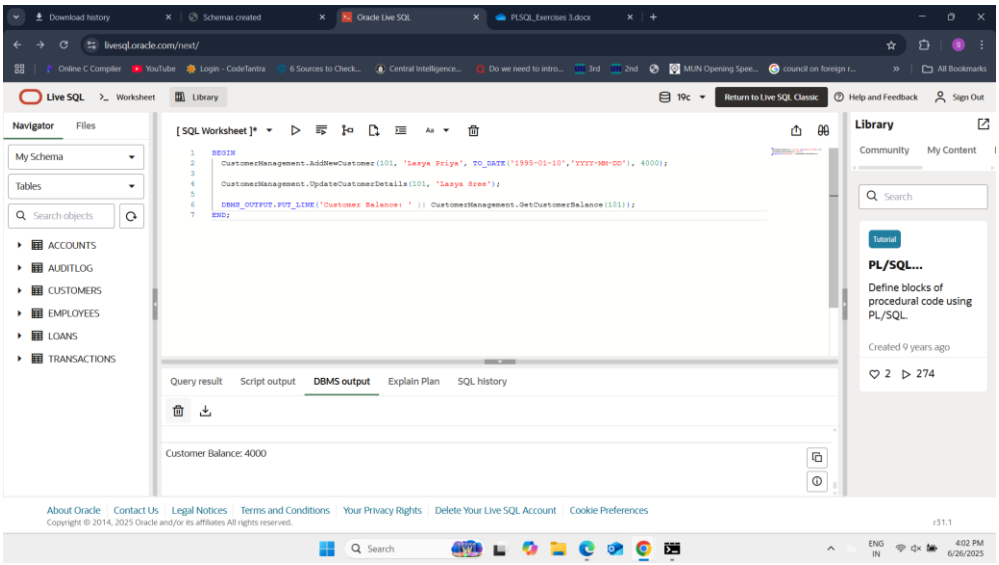
EXERCISE 7: PACKAGES

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 AddNewCustomer(
    p_customer_id IN NUMBER,
    p_name        IN VARCHAR2,
    p_dob         IN DATE,
    p_balance     IN NUMBER
  );

  PROCEDURE UpdateCustomerDetails(
    p_customer_id IN NUMBER,
    p_new_name    IN VARCHAR2
  );

  FUNCTION GetCustomerBalance(
    p_customer_id IN NUMBER
  ) RETURN NUMBER;
END CustomerManagement;
```

```
CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

  PROCEDURE AddNewCustomer(
    p_customer_id 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_customer_id, p_name, p_dob, p_balance, SYSDATE);
```

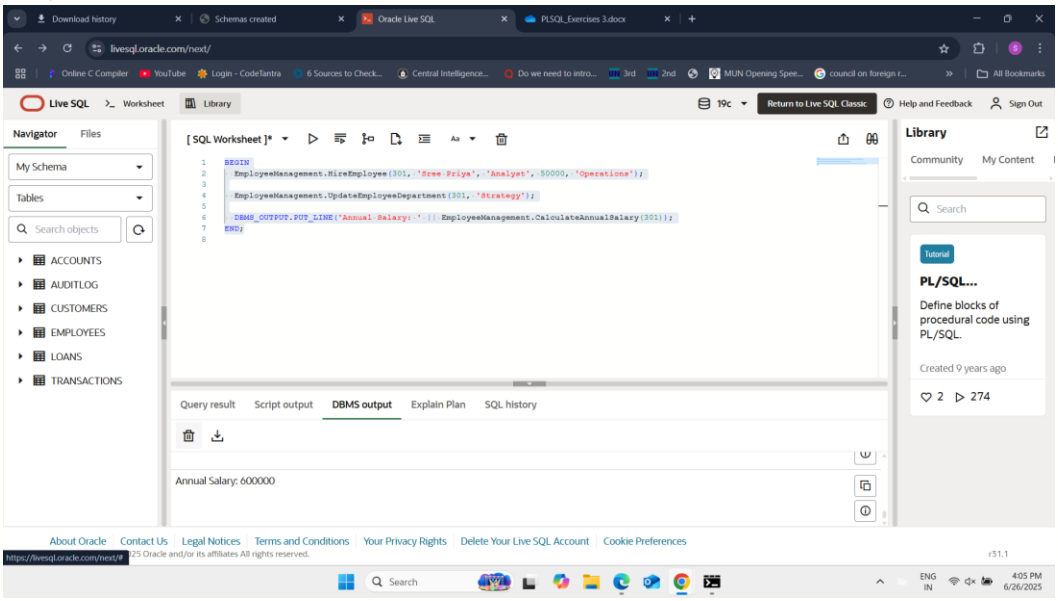
```
  EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
      DBMS_OUTPUT.PUT_LINE('Customer already exists with ID: ' || p_customer_id);
  END;

  PROCEDURE UpdateCustomerDetails(
    p_customer_id IN NUMBER,
    p_new_name    IN VARCHAR2
  ) IS
  BEGIN
    UPDATE Customers
    SET Name = p_new_name,
        LastModified = SYSDATE
    WHERE CustomerID = p_customer_id;
  END;

  FUNCTION GetCustomerBalance(
    p_customer_id IN NUMBER
  ) RETURN NUMBER IS
    v_balance NUMBER;
  BEGIN
    SELECT Balance INTO v_balance
    FROM Customers
    WHERE CustomerID = p_customer_id;
    RETURN v_balance;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      RETURN NULL;
  END;

END CustomerManagement;



BEGIN
  CustomerManagement.AddNewCustomer(101, 'Lasya Priya', TO_DATE('1995-01-10','YYYY-MM-
DD'), 4000);
  CustomerManagement.UpdateCustomerDetails(101, 'Lasya Sree');
  DBMS_OUTPUT.PUT_LINE('Customer Balance: ' ||
CustomerManagement.GetCustomerBalance(101));
END;
```



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_emp_id    IN NUMBER,
    p_name      IN VARCHAR2,
    p_position  IN VARCHAR2,
    p_salary    IN NUMBER,
    p_dept      IN VARCHAR2
  );

  PROCEDURE UpdateEmployeeDepartment(
    p_emp_id IN NUMBER,
    p_dept   IN VARCHAR2
  );

  FUNCTION CalculateAnnualSalary(
    p_emp_id IN NUMBER
  ) RETURN NUMBER;
END EmployeeManagement;


CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

  PROCEDURE HireEmployee(
    p_emp_id    IN NUMBER,
    p_name      IN VARCHAR2,
    p_position  IN VARCHAR2,
    p_salary    IN NUMBER,
    p_dept      IN VARCHAR2
  ) IS
  BEGIN
    INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate)
    VALUES (p_emp_id, p_name, p_position, p_salary, p_dept, SYSDATE);
  END;

  PROCEDURE UpdateEmployeeDepartment(
    p_emp_id IN NUMBER,
```

```
    p_dept   IN VARCHAR2
  ) IS
  BEGIN
    UPDATE Employees
    SET Department = p_dept
    WHERE EmployeeID = p_emp_id;
  END;

  FUNCTION CalculateAnnualSalary(
    p_emp_id IN NUMBER
  ) RETURN NUMBER IS
    v_salary NUMBER;
  BEGIN
    SELECT Salary INTO v_salary
    FROM Employees
    WHERE EmployeeID = p_emp_id;

    RETURN v_salary * 12;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      RETURN NULL;
  END;

END EmployeeManagement;


BEGIN
  EmployeeManagement.HireEmployee(301, 'Sree Priya', 'Analyst', 50000, 'Operations');

  EmployeeManagement.UpdateEmployeeDepartment(301, 'Strategy');

  DBMS_OUTPUT.PUT_LINE('Annual Salary: ' ||
EmployeeManagement.CalculateAnnualSalary(301));
END;
```



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. human like

```
CREATE OR REPLACE PACKAGE AccountOperations AS
  PROCEDURE OpenNewAccount(
    p_account_id  IN NUMBER,
    p_customer_id IN NUMBER,
    p_type        IN VARCHAR2,
    p_balance     IN NUMBER
  );

  PROCEDURE CloseAccount(
    p_account_id IN NUMBER
  );
  FUNCTION GetCustomerTotalBalance(
    p_customer_id IN NUMBER
  ) RETURN NUMBER;
END AccountOperations;



CREATE OR REPLACE PACKAGE BODY AccountOperations AS

  PROCEDURE OpenNewAccount(
    p_account_id  IN NUMBER,
    p_customer_id IN NUMBER,
    p_type        IN VARCHAR2,
    p_balance     IN NUMBER
  ) IS
  BEGIN
    INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified)
    VALUES (p_account_id, p_customer_id, p_type, p_balance, SYSDATE);
  END;

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

  FUNCTION GetCustomerTotalBalance(
    p_customer_id IN NUMBER
  ) RETURN NUMBER IS
    total_balance NUMBER;
  BEGIN
    SELECT SUM(Balance) INTO total_balance
    FROM Accounts
    WHERE CustomerID = p_customer_id;

    RETURN NVL(total_balance, 0);
  END;

END AccountOperations;



BEGIN
  AccountOperations.OpenNewAccount(3, 1, 'Fixed Deposit', 5000);
```

```
  DBMS_OUTPUT.PUT_LINE('Total Balance for Customer 1: ' ||
AccountOperations.GetCustomerTotalBalance(1));
  AccountOperations.CloseAccount(3);
  DBMS_OUTPUT.PUT_LINE('Balance after closing Fixed Deposit: ' ||
AccountOperations.GetCustomerTotalBalance(1));
END;
```