WEEK -2

PL/SQL PROGRAMMING

**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.

**CODE:**

BEGIN

FOR cust\_rec IN (

SELECT CustomerID

FROM Customers

WHERE MONTHS\_BETWEEN(SYSDATE, DOB) / 12 > 60

) LOOP

UPDATE Loans

SET InterestRate = InterestRate - 1

WHERE CustomerID = cust\_rec.CustomerID;

DBMS\_OUTPUT.PUT\_LINE('1% discount applied for Customer ID: ' || cust\_rec.CustomerID);

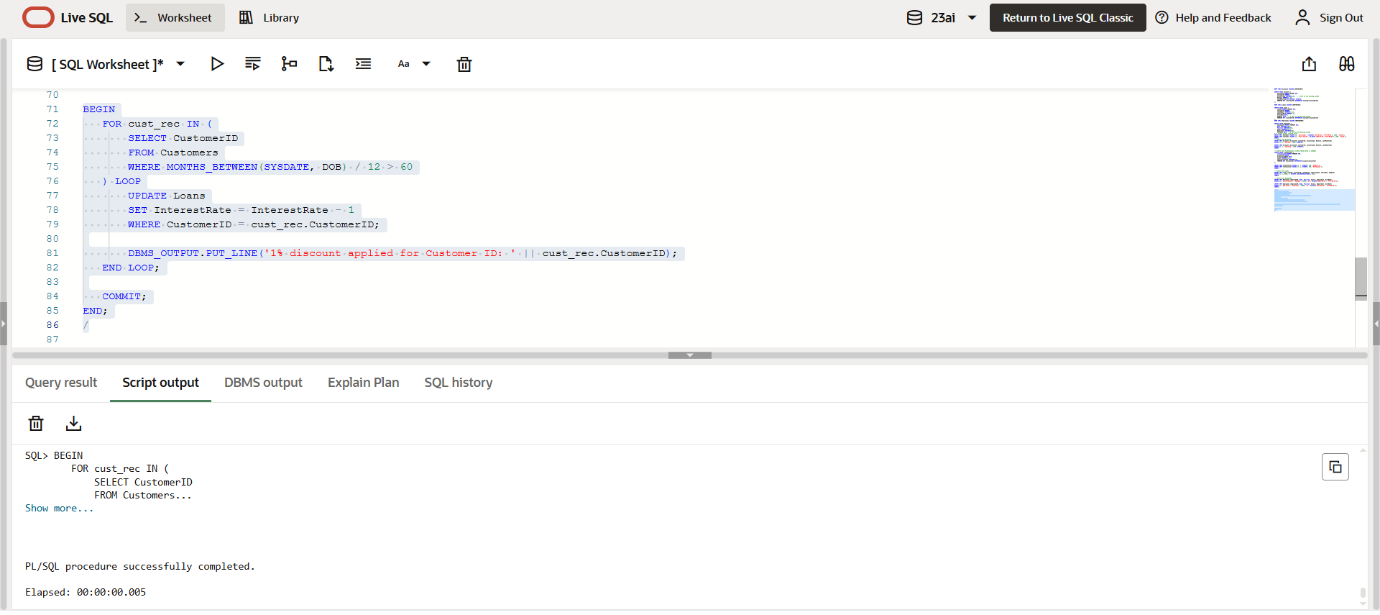
END LOOP;

COMMIT;

END;

/

**OUTPUT:**

****

**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.

**CODE:**

BEGIN

FOR cust\_rec IN (

SELECT CustomerID FROM Customers WHERE Balance > 10000

) LOOP

UPDATE Customers

SET IsVIP = 'TRUE'

WHERE CustomerID = cust\_rec.CustomerID;

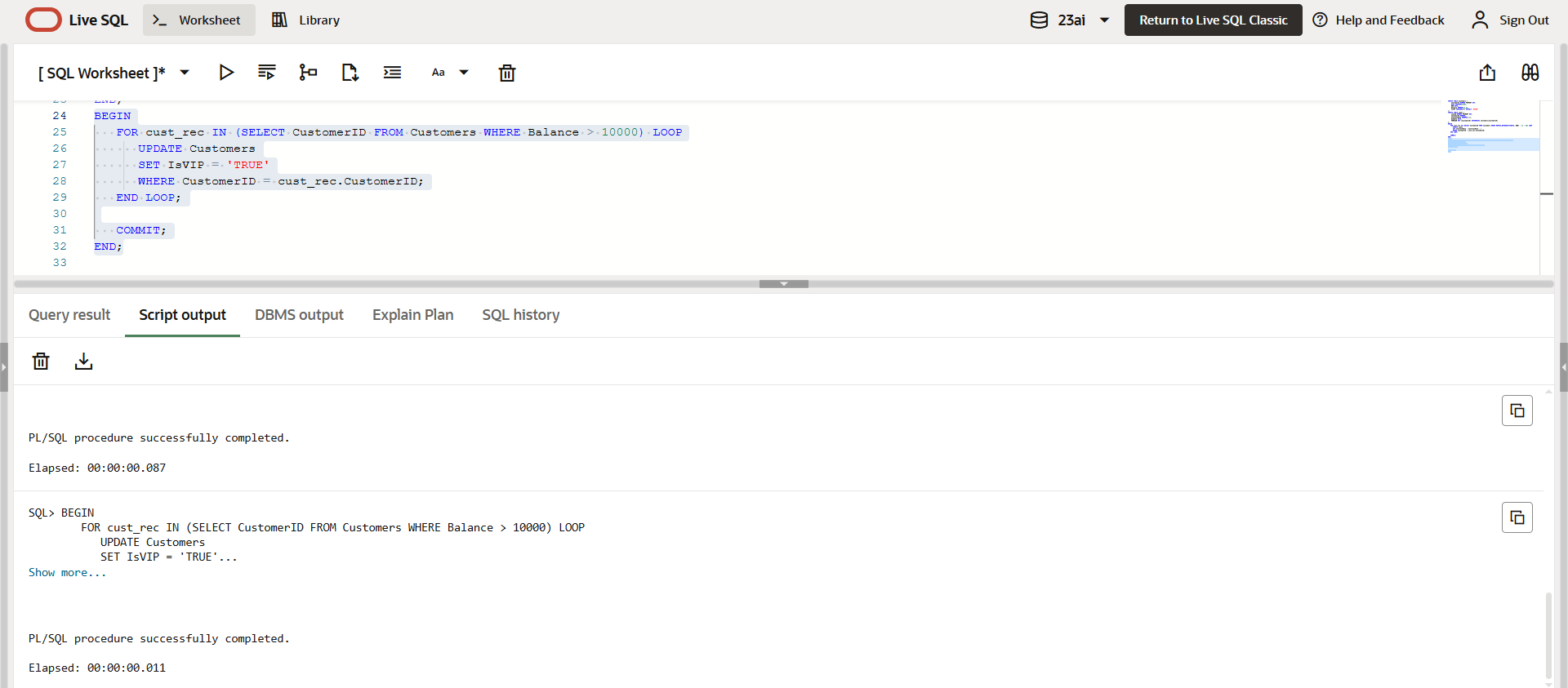
DBMS\_OUTPUT.PUT\_LINE('Customer ID ' || cust\_rec.CustomerID || ' promoted to VIP.');

END LOOP;

COMMIT;

END;

**OUTPUT:**

****

**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 loan\_rec IN (

SELECT L.LoanID, L.CustomerID, C.Name, L.DueDate

FROM Loans L

JOIN Customers C ON L.CustomerID = C.CustomerID

WHERE L.DueDate BETWEEN SYSDATE AND SYSDATE + 30

) LOOP

DBMS\_OUTPUT.PUT\_LINE('Reminder: Dear ' || loan\_rec.Name ||

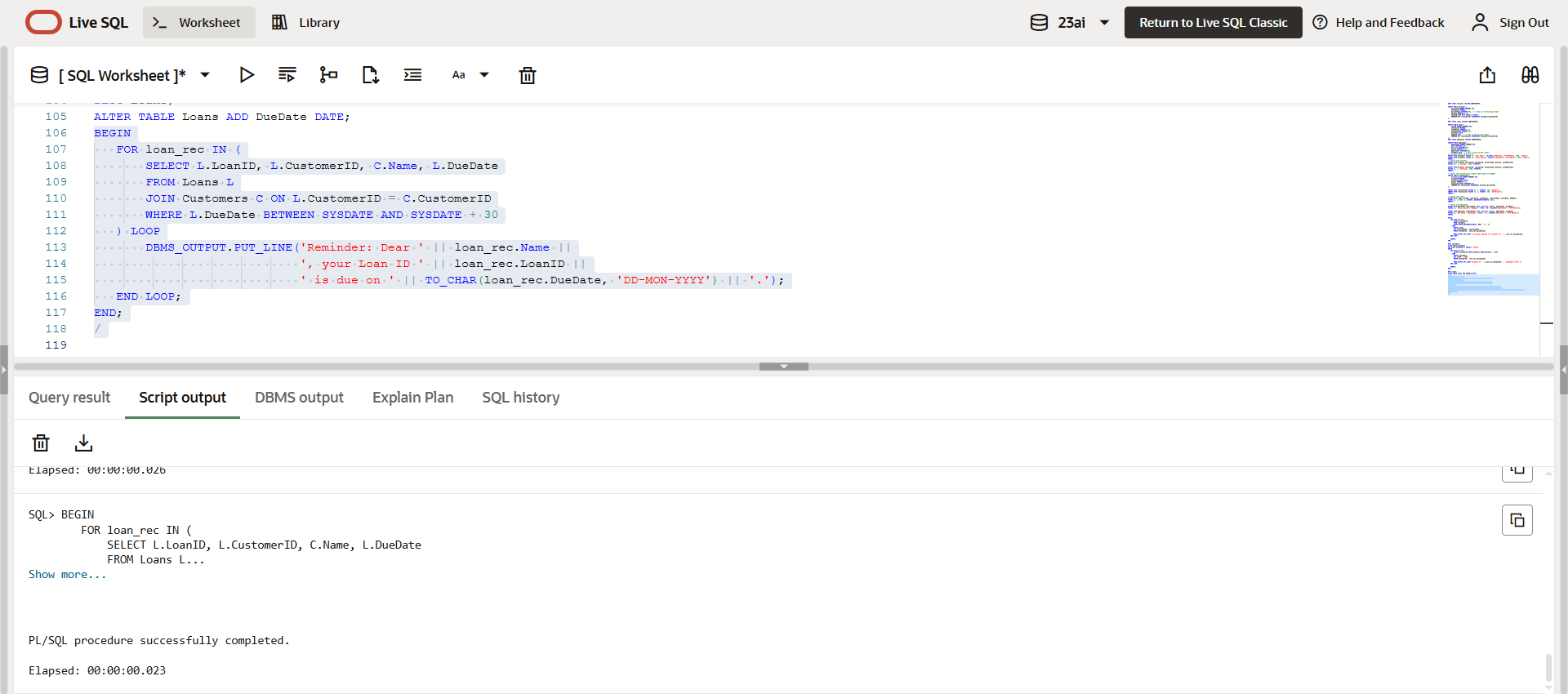
', your Loan ID ' || loan\_rec.LoanID ||

' is due on ' || TO\_CHAR(loan\_rec.DueDate, 'DD-MON-YYYY') || '.');

END LOOP;

END;

**OUTPUT:**



**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.

**CODE:**

CREATE OR REPLACE PROCEDURE SafeTransferFunds (

p\_fromAccountID IN NUMBER,

p\_toAccountID IN NUMBER,

p\_amount IN NUMBER

)

IS

v\_balance NUMBER;

insufficient\_funds EXCEPTION;

BEGIN

SELECT Balance INTO v\_balance

FROM Accounts

WHERE AccountID = p\_fromAccountID;

IF v\_balance < p\_amount THEN

RAISE insufficient\_funds;

END IF;

UPDATE Accounts

SET Balance = Balance - p\_amount,

LastModified = SYSDATE

WHERE AccountID = p\_fromAccountID;

UPDATE Accounts

SET Balance = Balance + p\_amount,

LastModified = SYSDATE

WHERE AccountID = p\_toAccountID;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Transfer successful: ₹' || p\_amount ||

' from Account ' || p\_fromAccountID ||

' to Account ' || p\_toAccountID);

EXCEPTION

WHEN insufficient\_funds THEN

DBMS\_OUTPUT.PUT\_LINE('Error: Insufficient funds in Account ' || p\_fromAccountID);

ROLLBACK;

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('Error: One or both account IDs do not exist.');

ROLLBACK;

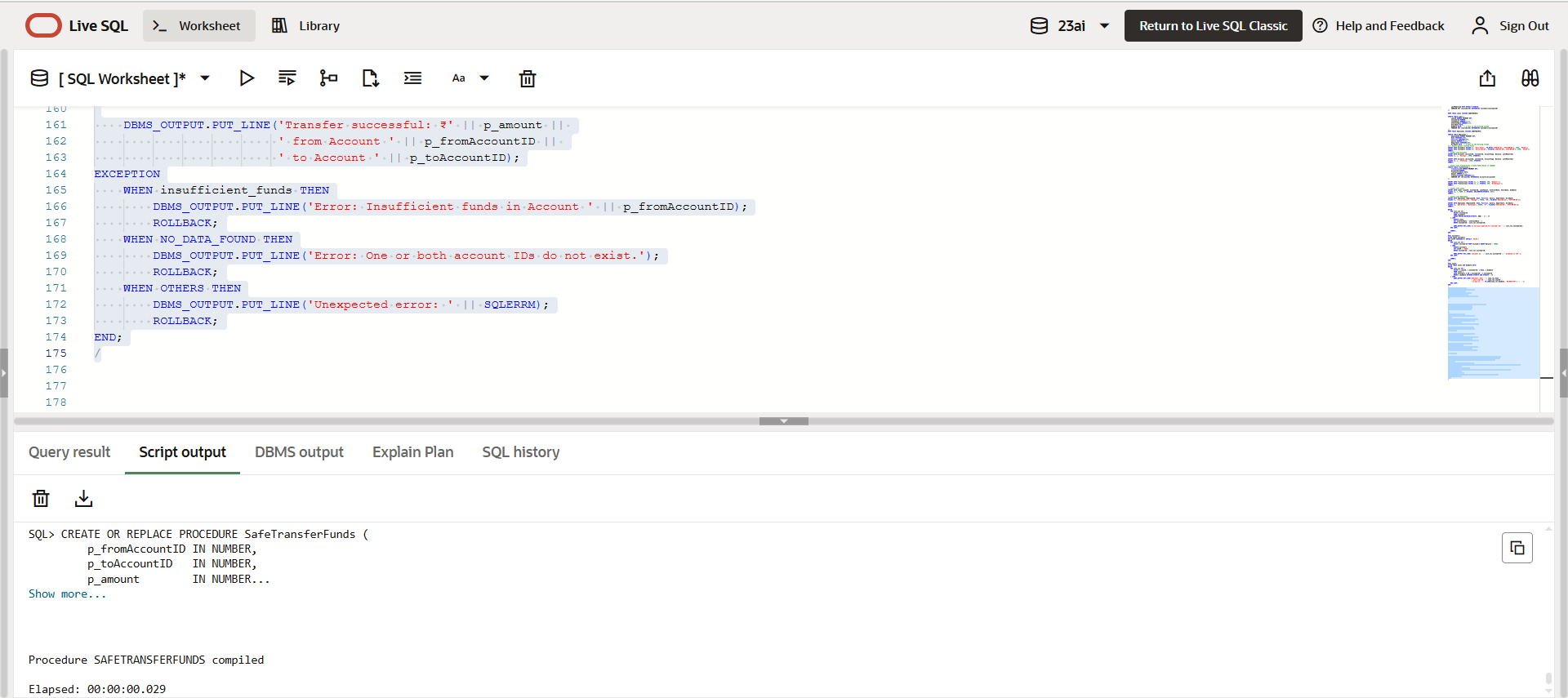
WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('Unexpected error: ' || SQLERRM);

ROLLBACK;

END;

**OUTPUT:**



**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.

**CODE:**

CREATE OR REPLACE PROCEDURE UpdateSalary (

    p\_empID      IN NUMBER,

    p\_percentage IN NUMBER

)

IS

    v\_salary NUMBER;

BEGIN

    SELECT Salary INTO v\_salary

    FROM Employees

    WHERE EmployeeID = p\_empID;

    UPDATE Employees

    SET Salary = Salary + (Salary \* p\_percentage / 100)

    WHERE EmployeeID = p\_empID;

    COMMIT;

    DBMS\_OUTPUT.PUT\_LINE('Salary updated successfully for Employee ID ' || p\_empID);

EXCEPTION

    WHEN NO\_DATA\_FOUND THEN

        DBMS\_OUTPUT.PUT\_LINE('Error: Employee ID ' || p\_empID || ' does not exist.');

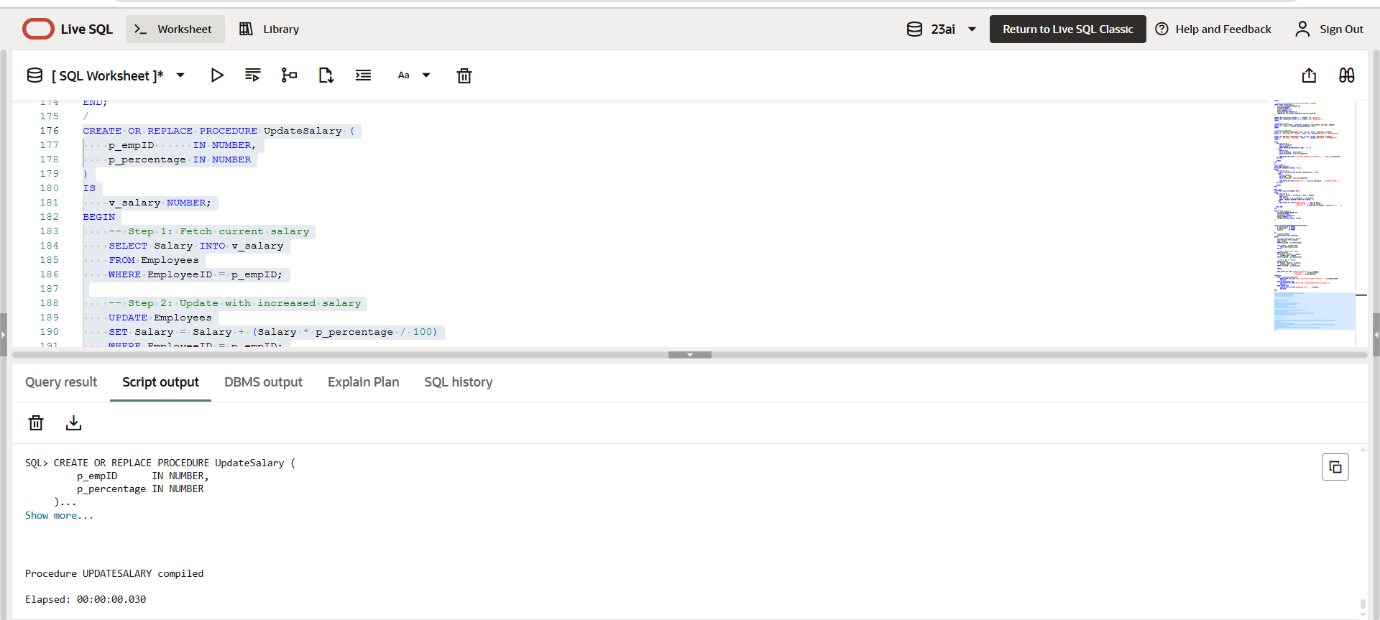
    WHEN OTHERS THEN

        DBMS\_OUTPUT.PUT\_LINE('Unexpected error: ' || SQLERRM);

END;

/

**OUTPUT:**



**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:**

CREATE OR REPLACE PROCEDURE AddNewCustomer (

    p\_customerID 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\_customerID, p\_name, p\_dob, p\_balance, SYSDATE);

    COMMIT;

    DBMS\_OUTPUT.PUT\_LINE('Customer added successfully. ID: ' || p\_customerID);

EXCEPTION

    WHEN DUP\_VAL\_ON\_INDEX THEN

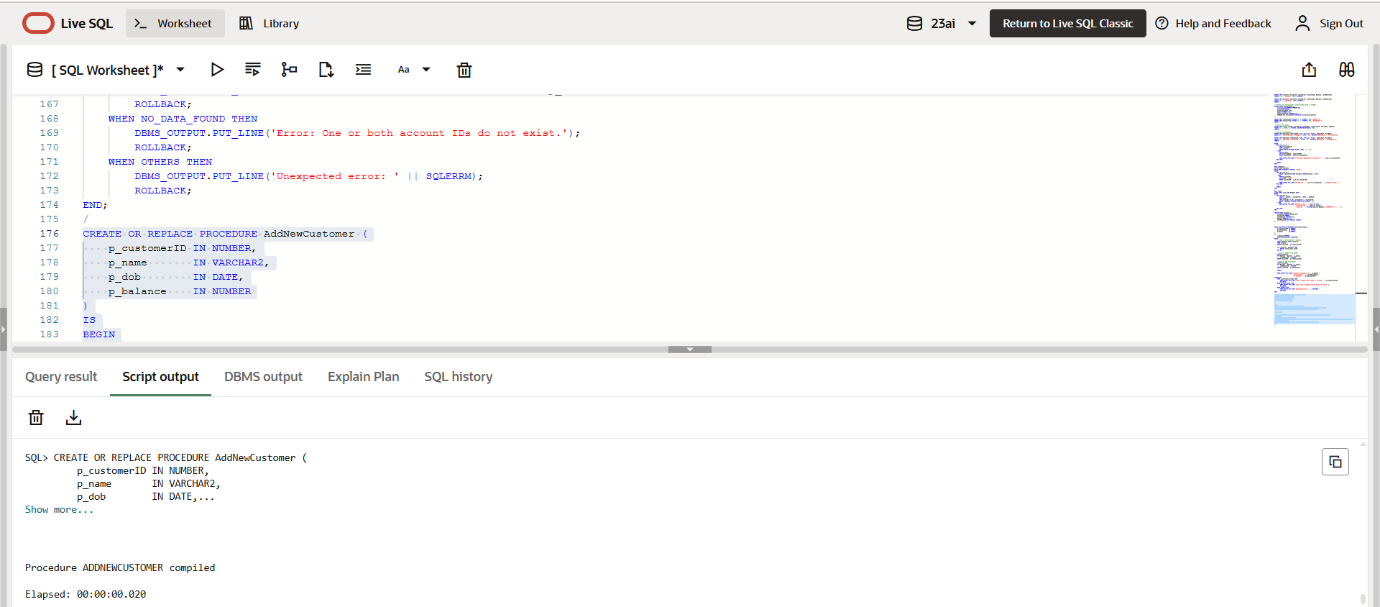
        DBMS\_OUTPUT.PUT\_LINE('Error: Customer ID ' || p\_customerID || ' already exists. Insertion aborted.');

    WHEN OTHERS THEN

        DBMS\_OUTPUT.PUT\_LINE('Unexpected error: ' || SQLERRM);

END;

/

**OUTPUT:**

**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.

**CODE:**

BEGIN

    UPDATE Accounts

    SET Balance = Balance + (Balance \* 0.01),

        LastModified = SYSDATE

    WHERE AccountType = 'Savings';

    COMMIT;

    DBMS\_OUTPUT.PUT\_LINE('Monthly interest applied to all savings accounts.');

EXCEPTION

    WHEN OTHERS THEN

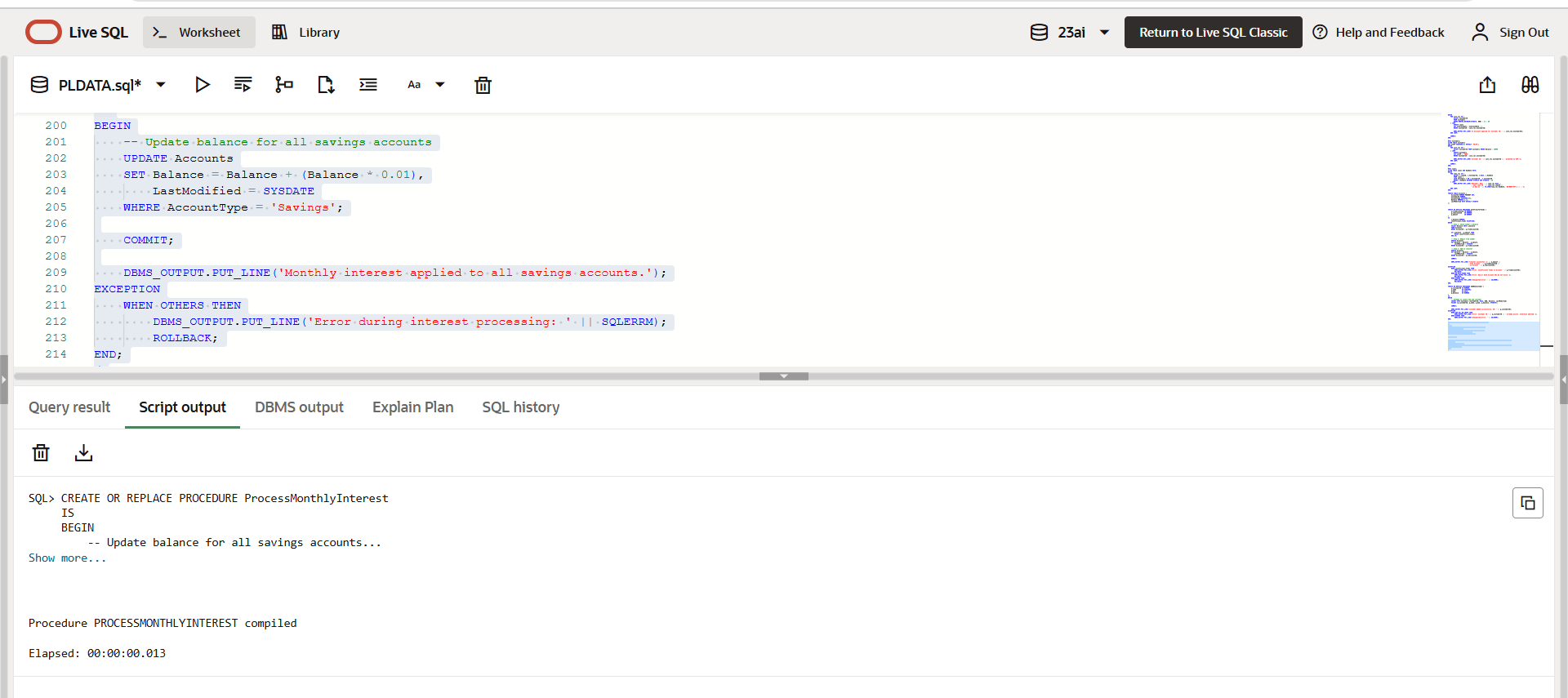
        DBMS\_OUTPUT.PUT\_LINE('Error during interest processing: ' || SQLERRM);

        ROLLBACK;

END;

/

**OUTPUT:**



**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.

**CODE:**

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (

    p\_department     IN VARCHAR2,

    p\_bonus\_percent  IN NUMBER

)

IS

    v\_count NUMBER := 0;

BEGIN

    UPDATE Employees

    SET Salary = Salary + (Salary \* p\_bonus\_percent / 100)

    WHERE Department = p\_department;

    v\_count := SQL%ROWCOUNT;

    COMMIT;

    DBMS\_OUTPUT.PUT\_LINE('Bonus applied to ' || v\_count || ' employee(s) in department: ' || p\_department);

EXCEPTION

    WHEN OTHERS THEN

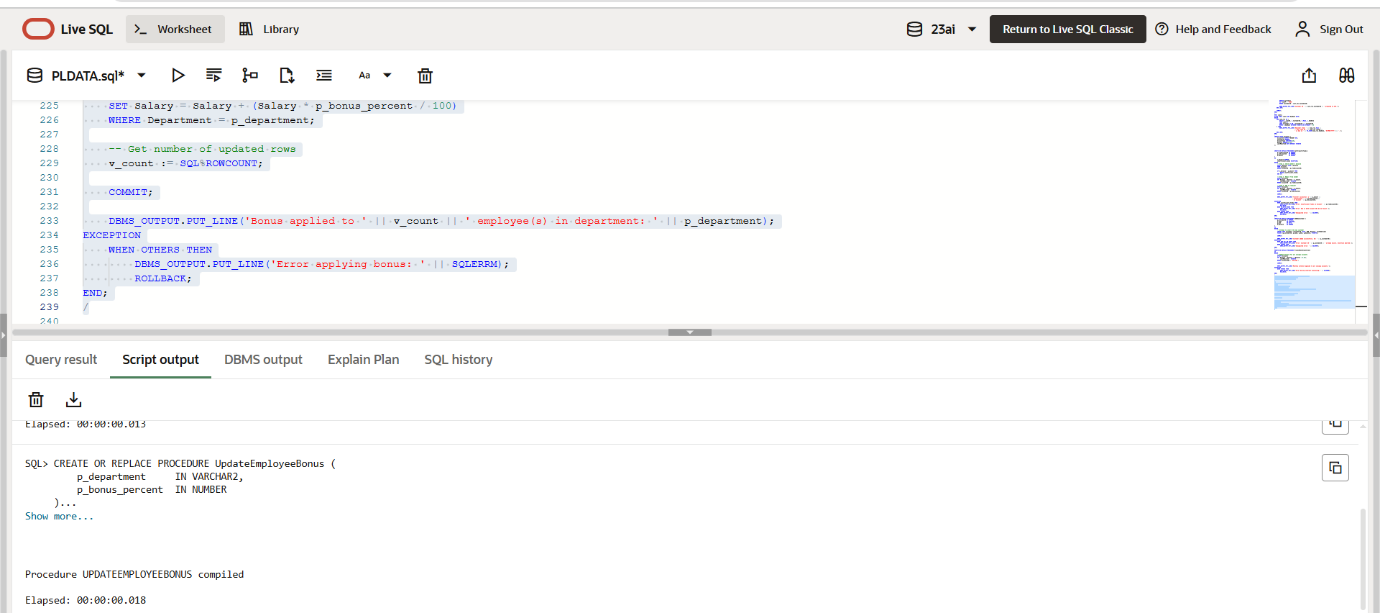
        DBMS\_OUTPUT.PUT\_LINE('Error applying bonus: ' || SQLERRM);

        ROLLBACK;

END;

/

**OUTPUT:**



**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:**

CREATE OR REPLACE PROCEDURE TransferFunds (

    p\_fromAccountID IN NUMBER,

    p\_toAccountID   IN NUMBER,

    p\_amount        IN NUMBER

)

IS

    v\_balance NUMBER;

    insufficient\_funds EXCEPTION;

BEGIN

    SELECT Balance INTO v\_balance

    FROM Accounts

    WHERE AccountID = p\_fromAccountID;

    IF v\_balance < p\_amount THEN

        RAISE insufficient\_funds;

    END IF;

    UPDATE Accounts

    SET Balance = Balance - p\_amount,

        LastModified = SYSDATE

    WHERE AccountID = p\_fromAccountID;

    UPDATE Accounts

    SET Balance = Balance + p\_amount,

        LastModified = SYSDATE

    WHERE AccountID = p\_toAccountID;

    COMMIT;

    DBMS\_OUTPUT.PUT\_LINE('₹' || p\_amount || ' transferred from Account ' || p\_fromAccountID || ' to Account ' || p\_toAccountID);

EXCEPTION

    WHEN insufficient\_funds THEN

        DBMS\_OUTPUT.PUT\_LINE('Error: Insufficient funds in Account ' || p\_fromAccountID);

        ROLLBACK;

    WHEN NO\_DATA\_FOUND THEN

        DBMS\_OUTPUT.PUT\_LINE('Error: One or both account IDs not found.');

        ROLLBACK;

    WHEN OTHERS THEN

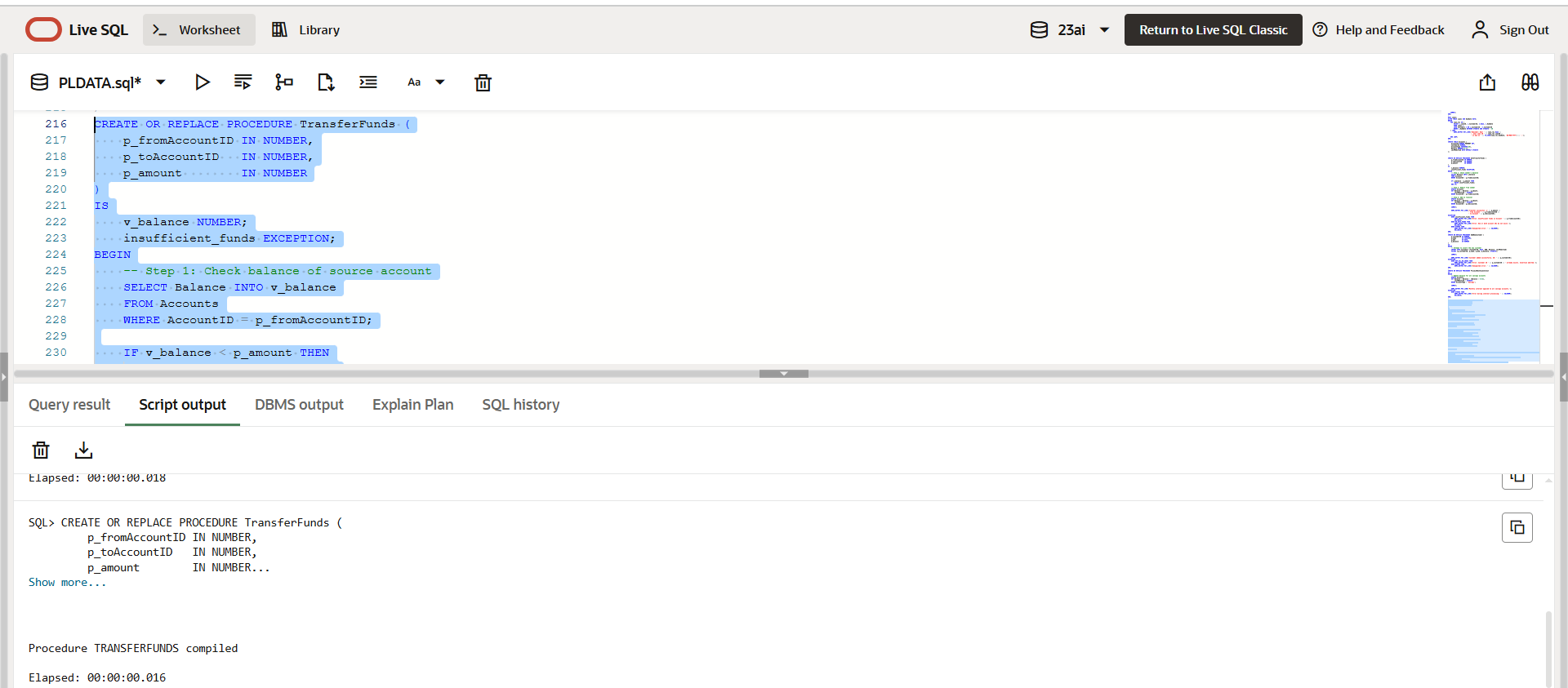
        DBMS\_OUTPUT.PUT\_LINE('Unexpected error: ' || SQLERRM);

        ROLLBACK;

END;

/

**OUTPUT:**



**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 FUNCTION CalculateAge (

    p\_dob IN DATE

) RETURN NUMBER

IS

    v\_age NUMBER;

BEGIN

    v\_age := FLOOR(MONTHS\_BETWEEN(SYSDATE, p\_dob) / 12);

    RETURN v\_age;

EXCEPTION

    WHEN OTHERS THEN

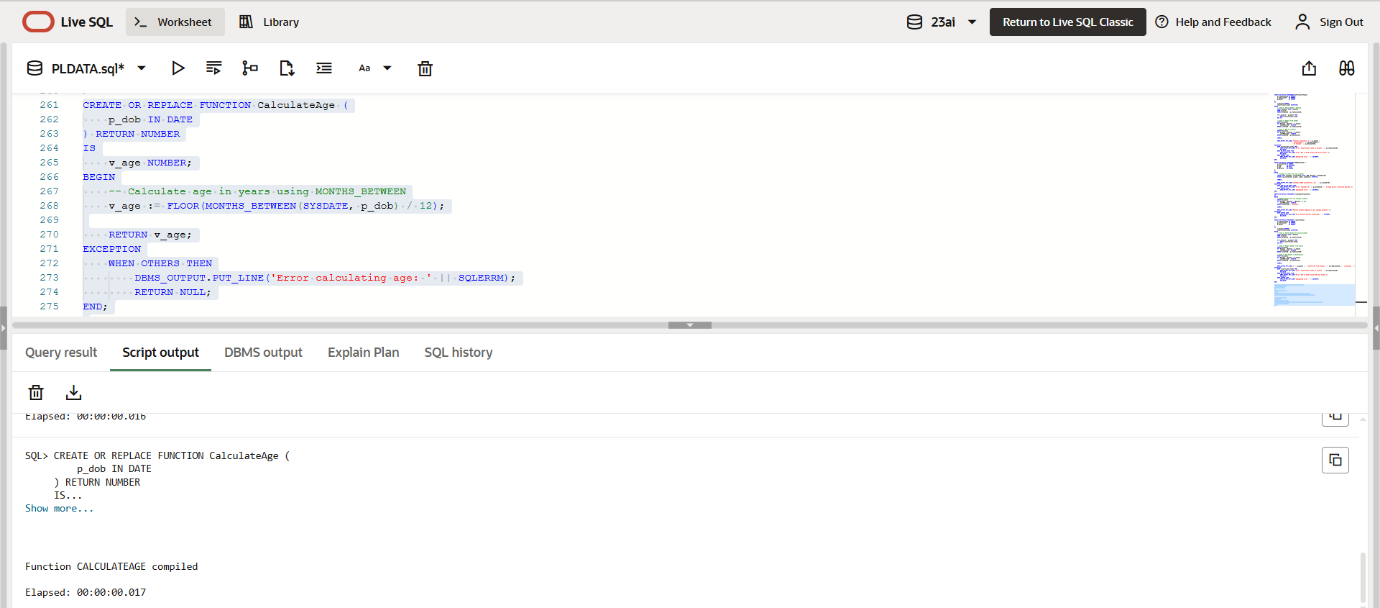
        DBMS\_OUTPUT.PUT\_LINE('Error calculating age: ' || SQLERRM);

        RETURN NULL;

END;

/

**OUTPUT:**



**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 FUNCTION CalculateMonthlyInstallment (

    p\_loanAmount   IN NUMBER,

    p\_interestRate IN NUMBER,

    p\_years        IN NUMBER

) RETURN NUMBER

IS

    v\_monthlyRate NUMBER;

    v\_totalMonths NUMBER;

    v\_emi         NUMBER;

BEGIN

    v\_monthlyRate := p\_interestRate / (12 \* 100);

    v\_totalMonths := p\_years \* 12;

    v\_emi := (p\_loanAmount \* v\_monthlyRate \* POWER(1 + v\_monthlyRate, v\_totalMonths)) /

             (POWER(1 + v\_monthlyRate, v\_totalMonths) - 1);

    RETURN ROUND(v\_emi, 2);

EXCEPTION

    WHEN OTHERS THEN

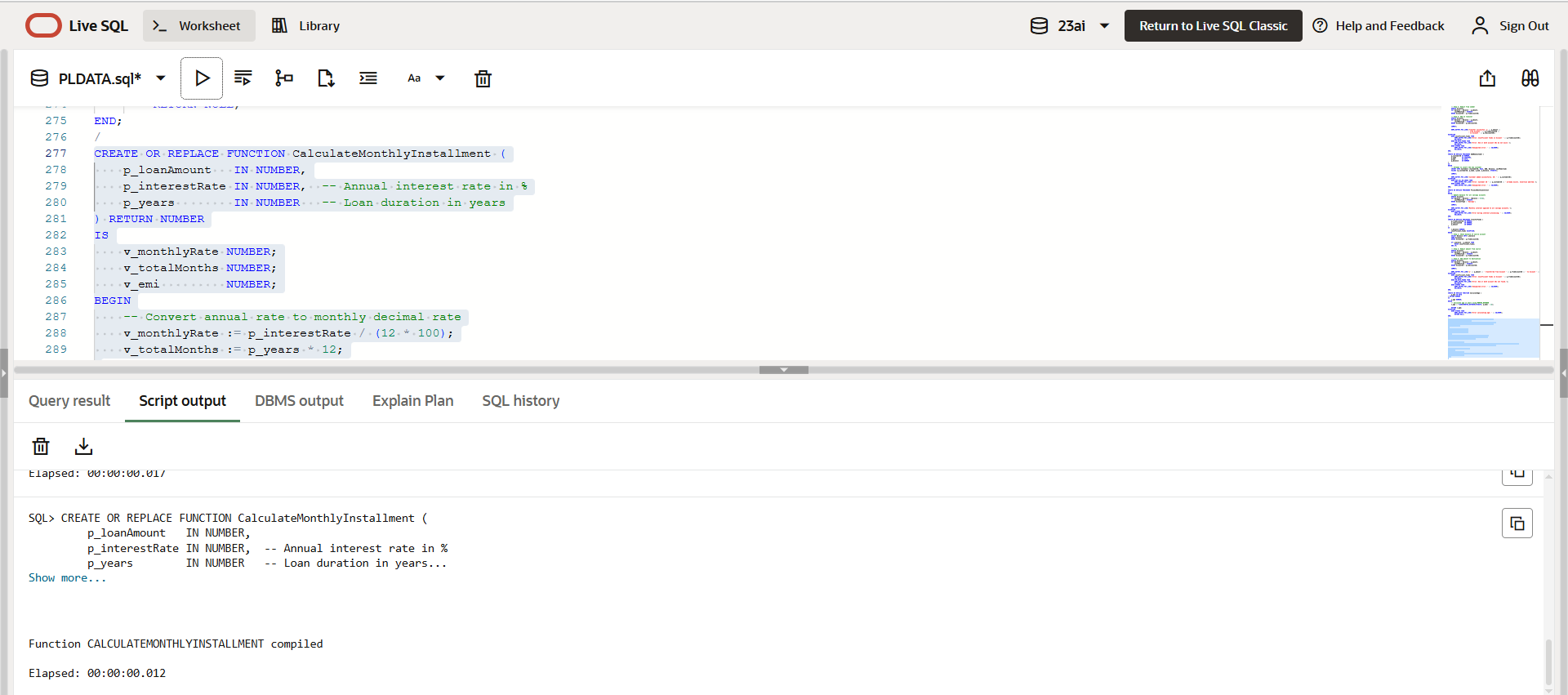
        DBMS\_OUTPUT.PUT\_LINE('Error calculating EMI: ' || SQLERRM);

        RETURN NULL;

END;

/

**OUTPUT:**



**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 FUNCTION HasSufficientBalance (

p\_accountID IN NUMBER,

p\_amount IN NUMBER

) RETURN BOOLEAN

IS

v\_balance NUMBER;

BEGIN

-- Get account balance

SELECT Balance INTO v\_balance

FROM Accounts

WHERE AccountID = p\_accountID;

RETURN v\_balance >= p\_amount;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('Error: Account ID ' || p\_accountID || ' not found.');

RETURN FALSE;

WHEN OTHERS THEN

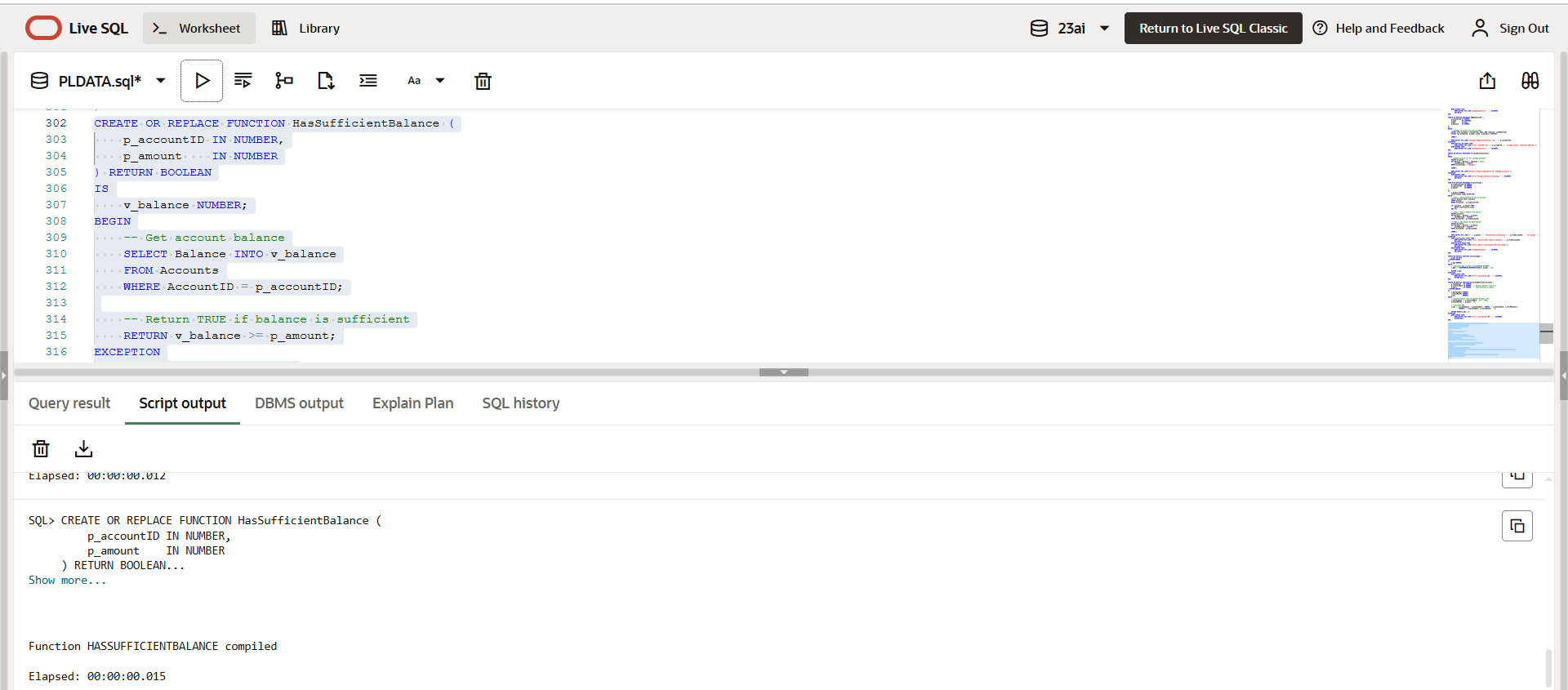
DBMS\_OUTPUT.PUT\_LINE('Unexpected error: ' || SQLERRM);

RETURN FALSE;

END;

/

**OUTPUT:**



**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 TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON Customers

FOR EACH ROW

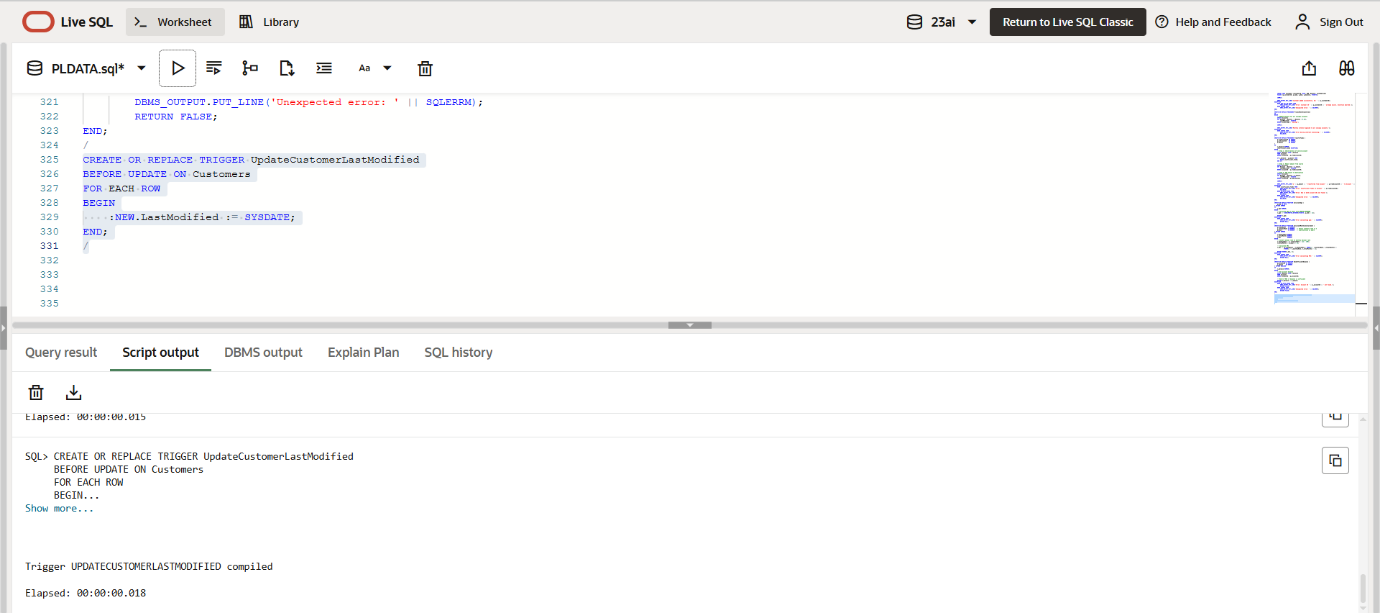
BEGIN

:NEW.LastModified := SYSDATE;

END;

/

**OUTPUT:**



**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 TABLE AuditLog (

    LogID NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,

    TransactionID NUMBER,

    AccountID NUMBER,

    Amount NUMBER,

    TransactionType VARCHAR2(10),

    ActionDate DATE

);

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

    INSERT INTO AuditLog (

        TransactionID,

        AccountID,

        Amount,

        TransactionType,

        ActionDate

    ) VALUES (

        :NEW.TransactionID,

        :NEW.AccountID,

        :NEW.Amount,

        :NEW.TransactionType,

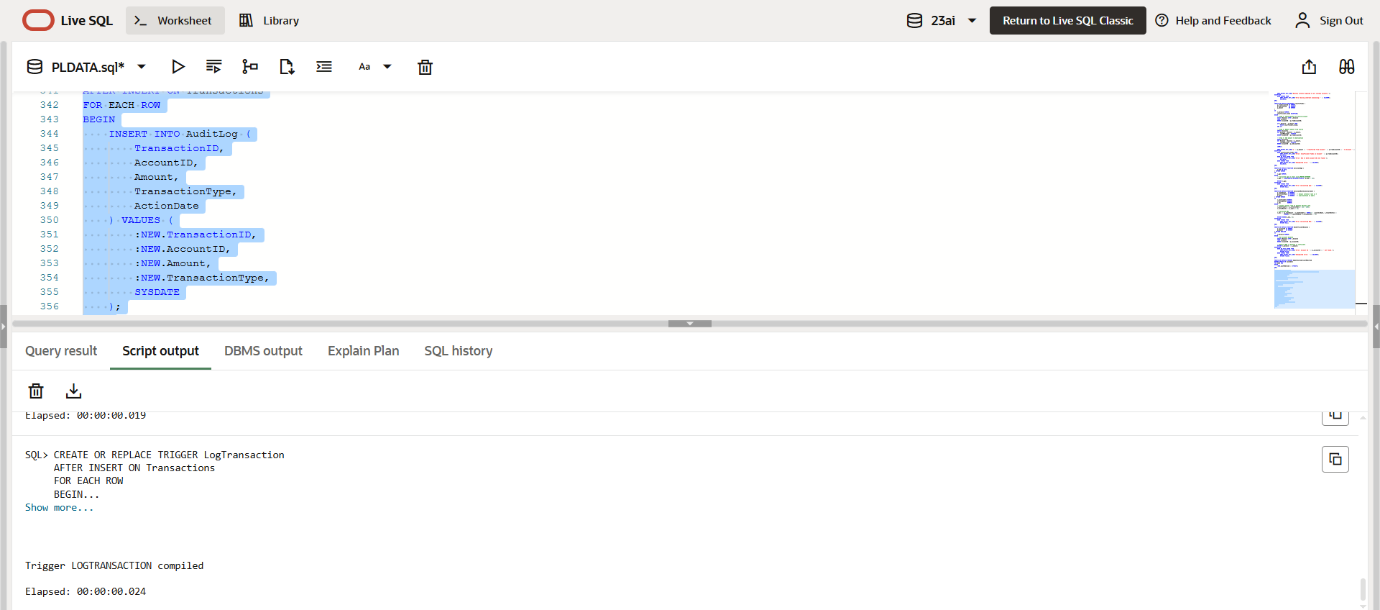
        SYSDATE

    );

END;

/

**OUTPUT:**



**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

    IF :NEW.TransactionType = 'Deposit' THEN

        IF :NEW.Amount <= 0 THEN

            RAISE\_APPLICATION\_ERROR(-20001, 'Deposit amount must be positive.');

        END IF;

    ELSIF :NEW.TransactionType = 'Withdrawal' THEN

        SELECT Balance INTO v\_balance

        FROM Accounts

        WHERE AccountID = :NEW.AccountID;

        IF :NEW.Amount > v\_balance THEN

            RAISE\_APPLICATION\_ERROR(-20002, 'Insufficient balance for withdrawal.');

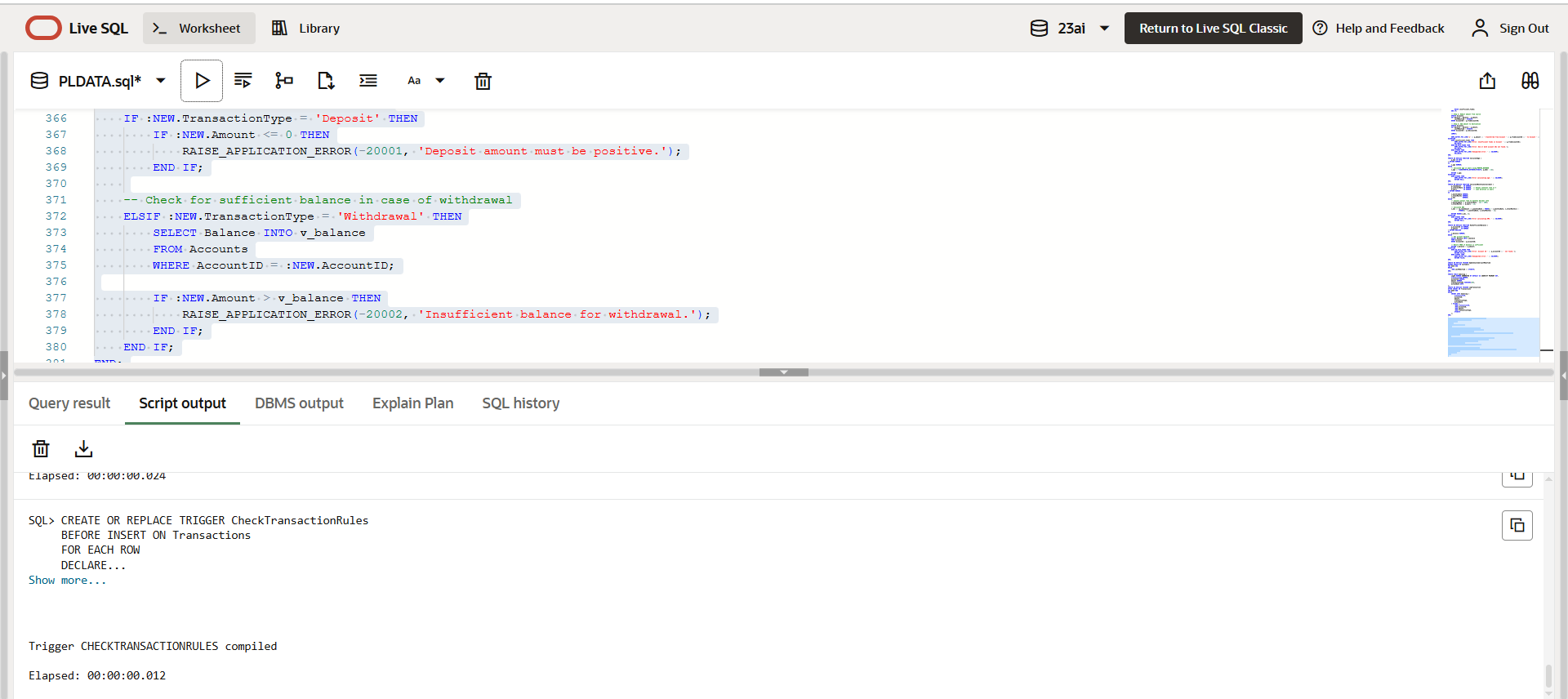
        END IF;

    END IF;

END;

/

**OUTPUT:**



**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 txn\_cursor IS

        SELECT

            c.CustomerID,

            c.Name,

            t.TransactionID,

            t.AccountID,

            t.TransactionDate,

            t.Amount,

            t.TransactionType

        FROM Transactions t

        JOIN Accounts a ON t.AccountID = a.AccountID

        JOIN Customers c ON a.CustomerID = c.CustomerID

        WHERE TO\_CHAR(t.TransactionDate, 'MMYYYY') = TO\_CHAR(SYSDATE, 'MMYYYY')

        ORDER BY c.CustomerID, t.TransactionDate;

    v\_custID      Customers.CustomerID%TYPE;

    v\_name        Customers.Name%TYPE;

    v\_txnID       Transactions.TransactionID%TYPE;

    v\_accID       Transactions.AccountID%TYPE;

    v\_date        Transactions.TransactionDate%TYPE;

    v\_amt         Transactions.Amount%TYPE;

    v\_type        Transactions.TransactionType%TYPE;

    v\_lastCustID  NUMBER := NULL;

BEGIN

    DBMS\_OUTPUT.PUT\_LINE('--- Monthly Transaction Statements ---');

    OPEN txn\_cursor;

    LOOP

        FETCH txn\_cursor INTO v\_custID, v\_name, v\_txnID, v\_accID, v\_date, v\_amt, v\_type;

        EXIT WHEN txn\_cursor%NOTFOUND;

        IF v\_lastCustID IS NULL OR v\_lastCustID != v\_custID THEN

            DBMS\_OUTPUT.PUT\_LINE(CHR(10) || 'Customer ID: ' || v\_custID || ' | Name: ' || v\_name);

            DBMS\_OUTPUT.PUT\_LINE('-------------------------------------------');

            v\_lastCustID := v\_custID;

        END IF;

        DBMS\_OUTPUT.PUT\_LINE(

            'TxnID: ' || v\_txnID ||

            ', AccID: ' || v\_accID ||

            ', Date: ' || TO\_CHAR(v\_date, 'DD-MON-YYYY') ||

            ', Type: ' || v\_type ||

            ', Amount: ₹' || v\_amt

        );

    END LOOP;

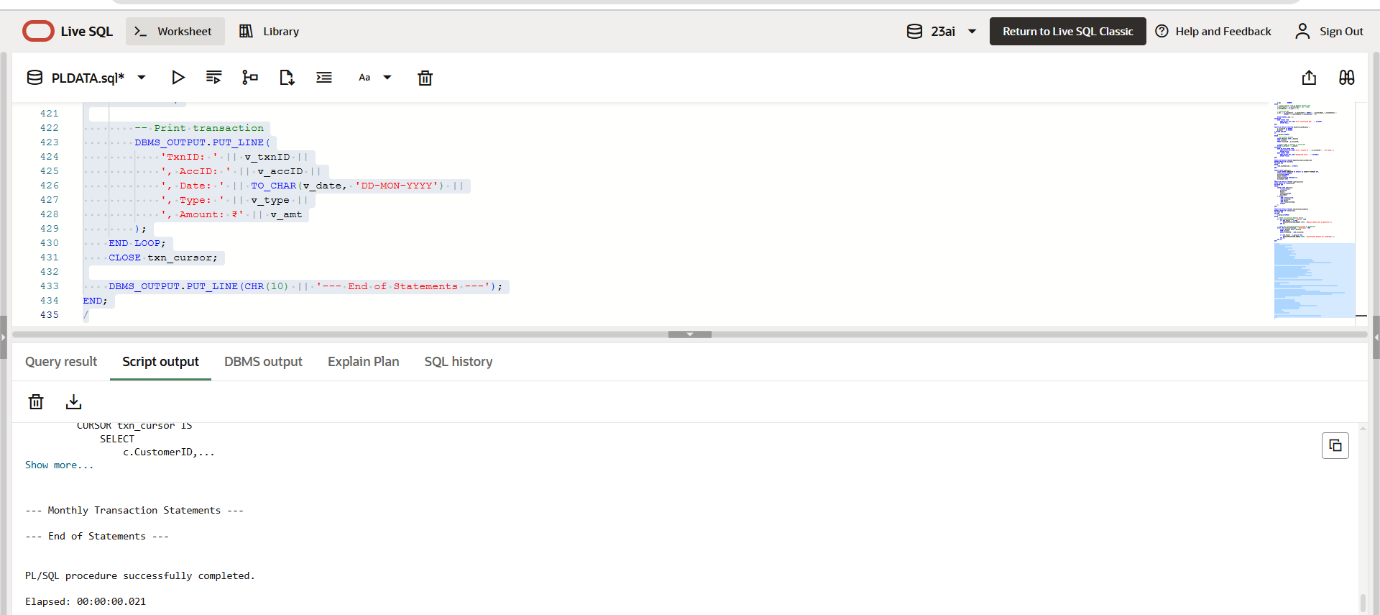
    CLOSE txn\_cursor;

    DBMS\_OUTPUT.PUT\_LINE(CHR(10) || '--- End of Statements ---');

END;

/

**OUTPUT:**



**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 acc\_cursor IS

        SELECT AccountID, Balance

        FROM Accounts;

    v\_accountID Accounts.AccountID%TYPE;

    v\_balance   Accounts.Balance%TYPE;

    v\_fee       CONSTANT NUMBER := 500;  -- Fixed annual fee

BEGIN

    OPEN acc\_cursor;

    LOOP

        FETCH acc\_cursor INTO v\_accountID, v\_balance;

        EXIT WHEN acc\_cursor%NOTFOUND;

        IF v\_balance >= v\_fee THEN

            UPDATE Accounts

            SET Balance = Balance - v\_fee,

                LastModified = SYSDATE

            WHERE AccountID = v\_accountID;

            DBMS\_OUTPUT.PUT\_LINE('Annual fee of ₹' || v\_fee || ' applied to Account ID: ' || v\_accountID);

        ELSE

            DBMS\_OUTPUT.PUT\_LINE('Account ID: ' || v\_accountID || ' has insufficient balance for fee deduction.');

        END IF;

    END LOOP;

    CLOSE acc\_cursor;

    COMMIT;

    DBMS\_OUTPUT.PUT\_LINE(CHR(10) || 'Annual fee application complete.');

EXCEPTION

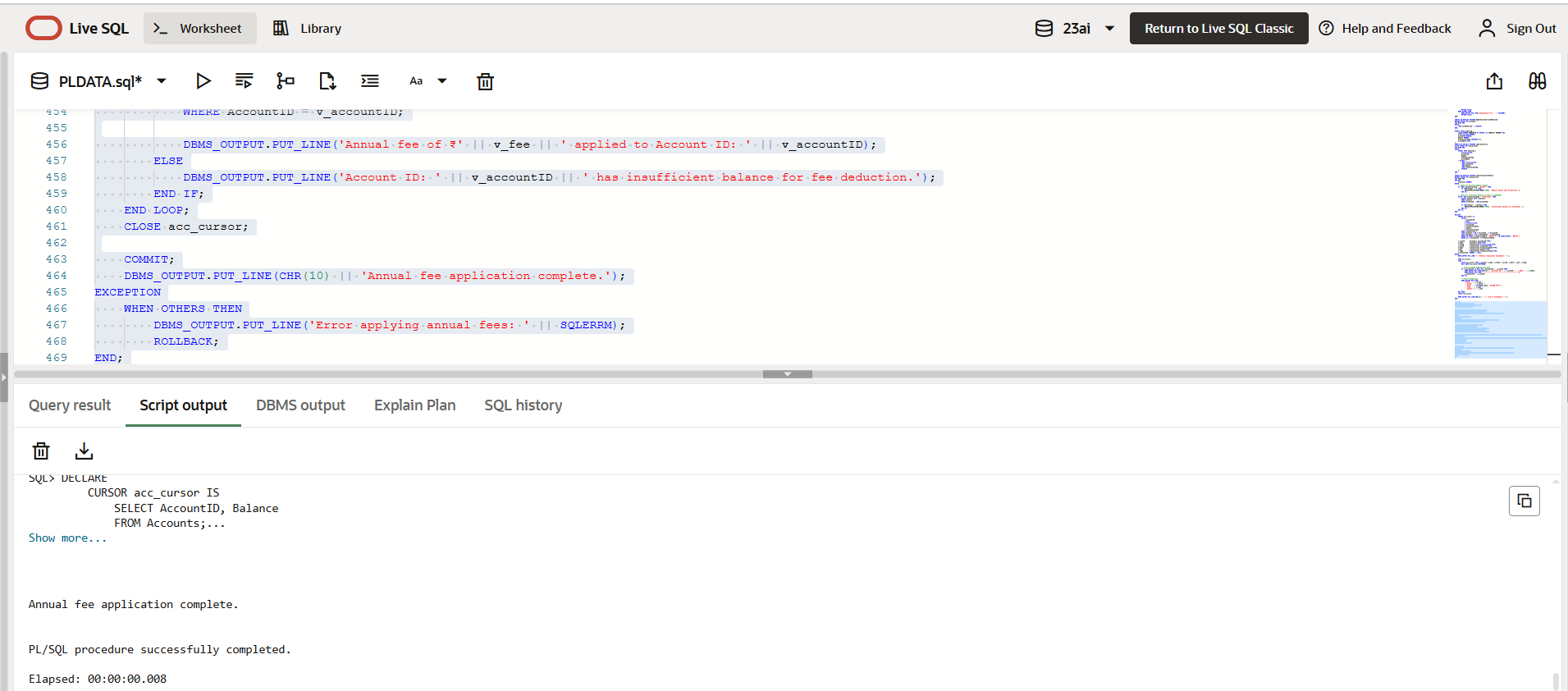
    WHEN OTHERS THEN

        DBMS\_OUTPUT.PUT\_LINE('Error applying annual fees: ' || SQLERRM);

        ROLLBACK;

END;

/

**OUTPUT:**

**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 loan\_cursor IS

SELECT LoanID, InterestRate, LoanAmount

FROM Loans;

v\_loanID Loans.LoanID%TYPE;

v\_interestRate Loans.InterestRate%TYPE;

v\_loanAmount Loans.LoanAmount%TYPE;

v\_newRate NUMBER;

BEGIN

OPEN loan\_cursor;

LOOP

FETCH loan\_cursor INTO v\_loanID, v\_interestRate, v\_loanAmount;

EXIT WHEN loan\_cursor%NOTFOUND;

IF v\_loanAmount > 100000 THEN

v\_newRate := v\_interestRate + 0.5;

ELSE

v\_newRate := v\_interestRate + 0.25;

END IF;

UPDATE Loans

SET InterestRate = v\_newRate

WHERE LoanID = v\_loanID;

DBMS\_OUTPUT.PUT\_LINE('Loan ID ' || v\_loanID || ' updated: New Rate = ' || v\_newRate || '%');

END LOOP;

CLOSE loan\_cursor;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE(CHR(10) || 'Interest rates updated based on new policy.');

EXCEPTION

WHEN OTHERS THEN

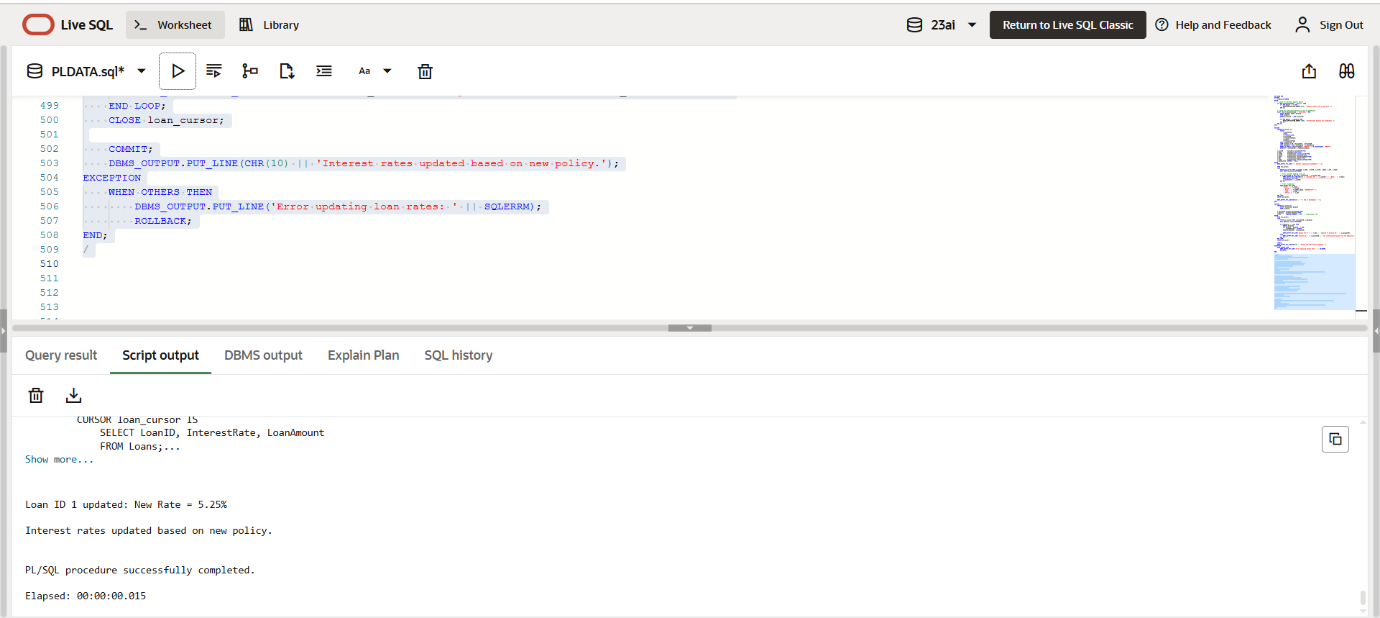
DBMS\_OUTPUT.PUT\_LINE('Error updating loan rates: ' || SQLERRM);

ROLLBACK;

END;

/

**OUTPUT:**



**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 IS

    PROCEDURE AddCustomer (

        p\_customerID IN NUMBER,

        p\_name       IN VARCHAR2,

        p\_dob        IN DATE,

        p\_balance    IN NUMBER

    );

    PROCEDURE UpdateCustomer (

        p\_customerID IN NUMBER,

        p\_name       IN VARCHAR2,

        p\_dob        IN DATE

    );

    FUNCTION GetCustomerBalance (

        p\_customerID IN NUMBER

    ) RETURN NUMBER;

END CustomerManagement;

/

CREATE OR REPLACE PACKAGE BODY CustomerManagement IS

    PROCEDURE AddCustomer (

        p\_customerID 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\_customerID, p\_name, p\_dob, p\_balance, SYSDATE);

        DBMS\_OUTPUT.PUT\_LINE('Customer added: ' || p\_name);

    EXCEPTION

        WHEN DUP\_VAL\_ON\_INDEX THEN

            DBMS\_OUTPUT.PUT\_LINE('Error: Customer ID already exists.');

        WHEN OTHERS THEN

            DBMS\_OUTPUT.PUT\_LINE('Unexpected error: ' || SQLERRM);

    END;

    PROCEDURE UpdateCustomer (

        p\_customerID IN NUMBER,

        p\_name       IN VARCHAR2,

        p\_dob        IN DATE

    ) IS

    BEGIN

        UPDATE Customers

        SET Name = p\_name,

            DOB = p\_dob,

            LastModified = SYSDATE

        WHERE CustomerID = p\_customerID;

        IF SQL%ROWCOUNT = 0 THEN

            DBMS\_OUTPUT.PUT\_LINE('No customer found with ID: ' || p\_customerID);

        ELSE

            DBMS\_OUTPUT.PUT\_LINE('Customer updated: ' || p\_name);

        END IF;

    EXCEPTION

        WHEN OTHERS THEN

            DBMS\_OUTPUT.PUT\_LINE('Error updating customer: ' || SQLERRM);

    END;

    FUNCTION GetCustomerBalance (

        p\_customerID IN NUMBER

    ) RETURN NUMBER IS

        v\_balance NUMBER;

    BEGIN

        SELECT Balance INTO v\_balance

        FROM Customers

        WHERE CustomerID = p\_customerID;

        RETURN v\_balance;

    EXCEPTION

        WHEN NO\_DATA\_FOUND THEN

            DBMS\_OUTPUT.PUT\_LINE('Customer not found.');

            RETURN NULL;

        WHEN OTHERS THEN

            DBMS\_OUTPUT.PUT\_LINE('Error retrieving balance: ' || SQLERRM);

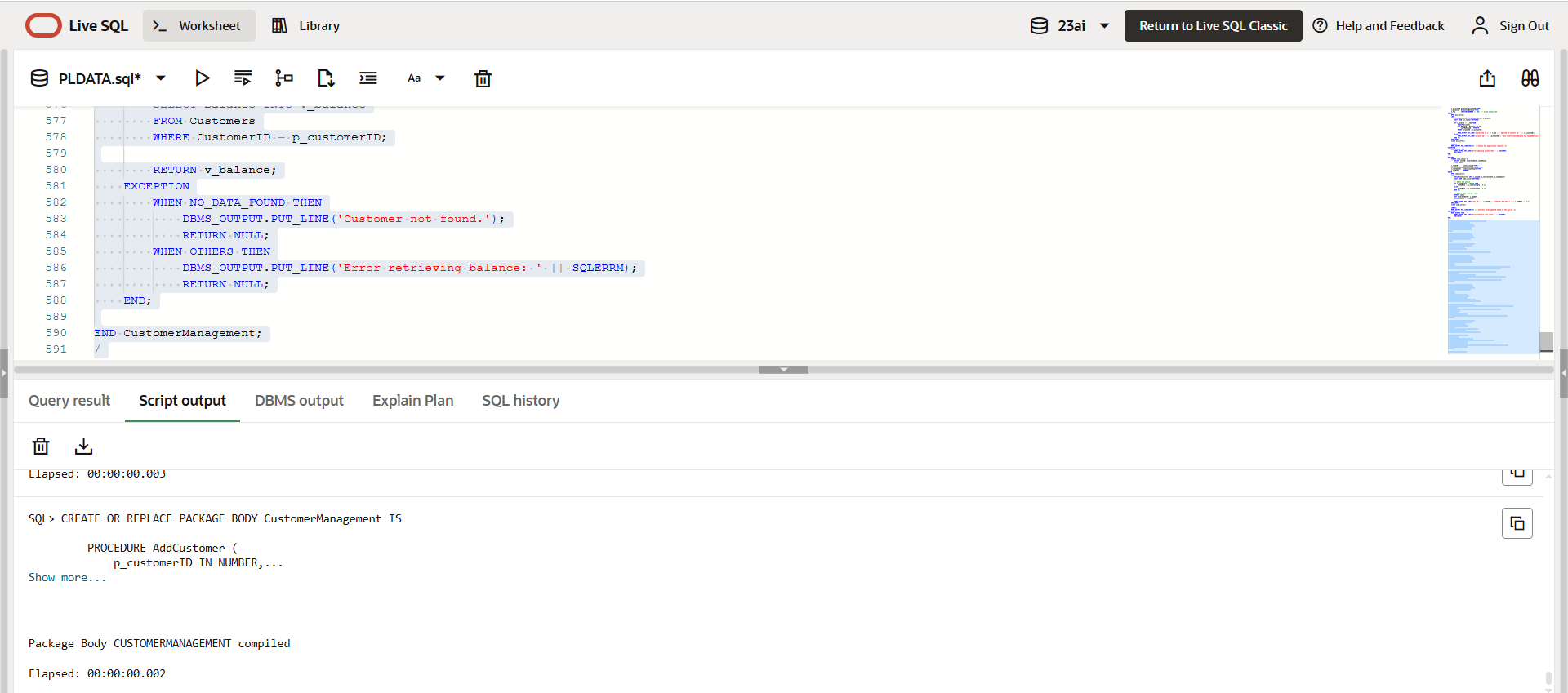
            RETURN NULL;

    END;

END CustomerManagement;

/

**OUTPUT:**



**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 IS

    PROCEDURE HireEmployee (

        p\_employeeID IN NUMBER,

        p\_name       IN VARCHAR2,

        p\_position   IN VARCHAR2,

        p\_salary     IN NUMBER,

        p\_department IN VARCHAR2,

        p\_hireDate   IN DATE

    );

    PROCEDURE UpdateEmployee (

        p\_employeeID IN NUMBER,

        p\_name       IN VARCHAR2,

        p\_position   IN VARCHAR2,

        p\_salary     IN NUMBER

    );

    FUNCTION CalculateAnnualSalary (

        p\_employeeID IN NUMBER

    ) RETURN NUMBER;

END EmployeeManagement;

/

CREATE OR REPLACE PACKAGE BODY EmployeeManagement IS

    PROCEDURE HireEmployee (

        p\_employeeID IN NUMBER,

        p\_name       IN VARCHAR2,

        p\_position   IN VARCHAR2,

        p\_salary     IN NUMBER,

        p\_department IN VARCHAR2,

        p\_hireDate   IN 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);

        DBMS\_OUTPUT.PUT\_LINE('Employee hired: ' || p\_name);

    EXCEPTION

        WHEN DUP\_VAL\_ON\_INDEX THEN

            DBMS\_OUTPUT.PUT\_LINE('Error: Employee ID already exists.');

        WHEN OTHERS THEN

            DBMS\_OUTPUT.PUT\_LINE('Unexpected error: ' || SQLERRM);

    END;

    PROCEDURE UpdateEmployee (

        p\_employeeID IN NUMBER,

        p\_name       IN VARCHAR2,

        p\_position   IN VARCHAR2,

        p\_salary     IN NUMBER

    ) IS

    BEGIN

        UPDATE Employees

        SET Name = p\_name,

            Position = p\_position,

            Salary = p\_salary

        WHERE EmployeeID = p\_employeeID;

        IF SQL%ROWCOUNT = 0 THEN

            DBMS\_OUTPUT.PUT\_LINE('No employee found with ID: ' || p\_employeeID);

        ELSE

            DBMS\_OUTPUT.PUT\_LINE('Employee updated: ' || p\_name);

        END IF;

    EXCEPTION

        WHEN OTHERS THEN

            DBMS\_OUTPUT.PUT\_LINE('Error updating employee: ' || SQLERRM);

    END;

    FUNCTION CalculateAnnualSalary (

        p\_employeeID IN NUMBER

    ) RETURN NUMBER IS

        v\_salary NUMBER;

    BEGIN

        SELECT Salary INTO v\_salary

        FROM Employees

        WHERE EmployeeID = p\_employeeID;

        RETURN v\_salary \* 12;

    EXCEPTION

        WHEN NO\_DATA\_FOUND THEN

            DBMS\_OUTPUT.PUT\_LINE('Employee not found.');

            RETURN NULL;

        WHEN OTHERS THEN

            DBMS\_OUTPUT.PUT\_LINE('Error calculating salary: ' || SQLERRM);

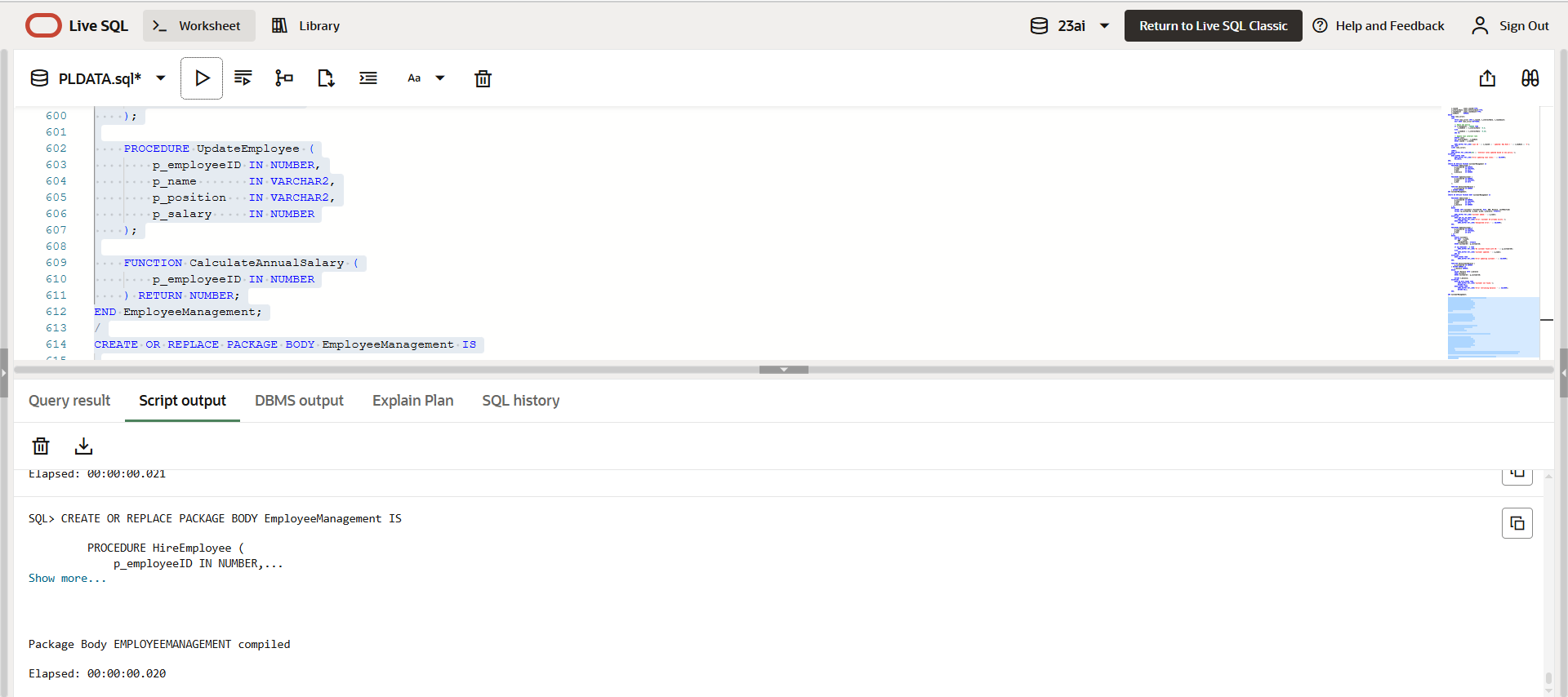
            RETURN NULL;

    END;

END EmployeeManagement;

/

**OUTPUT:**



**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 IS

    PROCEDURE OpenAccount (

        p\_accountID   IN NUMBER,

        p\_customerID  IN NUMBER,

        p\_accountType IN VARCHAR2,

        p\_balance     IN NUMBER

    );

    PROCEDURE CloseAccount (

        p\_accountID IN NUMBER

    );

    FUNCTION GetTotalBalance (

        p\_customerID IN NUMBER

    ) RETURN NUMBER;

END AccountOperations;

/

CREATE OR REPLACE PACKAGE BODY AccountOperations IS

    PROCEDURE OpenAccount (

        p\_accountID   IN NUMBER,

        p\_customerID  IN NUMBER,

        p\_accountType IN VARCHAR2,

        p\_balance     IN NUMBER

    ) IS

    BEGIN

        INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified)

        VALUES (p\_accountID, p\_customerID, p\_accountType, p\_balance, SYSDATE);

        DBMS\_OUTPUT.PUT\_LINE('Account opened with ID: ' || p\_accountID);

    EXCEPTION

        WHEN DUP\_VAL\_ON\_INDEX THEN

            DBMS\_OUTPUT.PUT\_LINE('Error: Account ID already exists.');

        WHEN OTHERS THEN

            DBMS\_OUTPUT.PUT\_LINE('Error opening account: ' || SQLERRM);

    END;

    PROCEDURE CloseAccount (

        p\_accountID IN NUMBER

    ) IS

    BEGIN

        DELETE FROM Accounts

        WHERE AccountID = p\_accountID;

        IF SQL%ROWCOUNT = 0 THEN

            DBMS\_OUTPUT.PUT\_LINE('Account not found: ' || p\_accountID);

        ELSE

            DBMS\_OUTPUT.PUT\_LINE('Account closed: ' || p\_accountID);

        END IF;

    EXCEPTION

        WHEN OTHERS THEN

            DBMS\_OUTPUT.PUT\_LINE('Error closing account: ' || SQLERRM);

    END;

    FUNCTION GetTotalBalance (

        p\_customerID IN NUMBER

    ) RETURN NUMBER IS

        v\_total NUMBER := 0;

    BEGIN

        SELECT NVL(SUM(Balance), 0)

        INTO v\_total

        FROM Accounts

        WHERE CustomerID = p\_customerID;

        RETURN v\_total;

    EXCEPTION

        WHEN OTHERS THEN

            DBMS\_OUTPUT.PUT\_LINE('Error retrieving total balance: ' || SQLERRM);

            RETURN NULL;

    END;

END AccountOperations;

/

**OUTPUT:**

