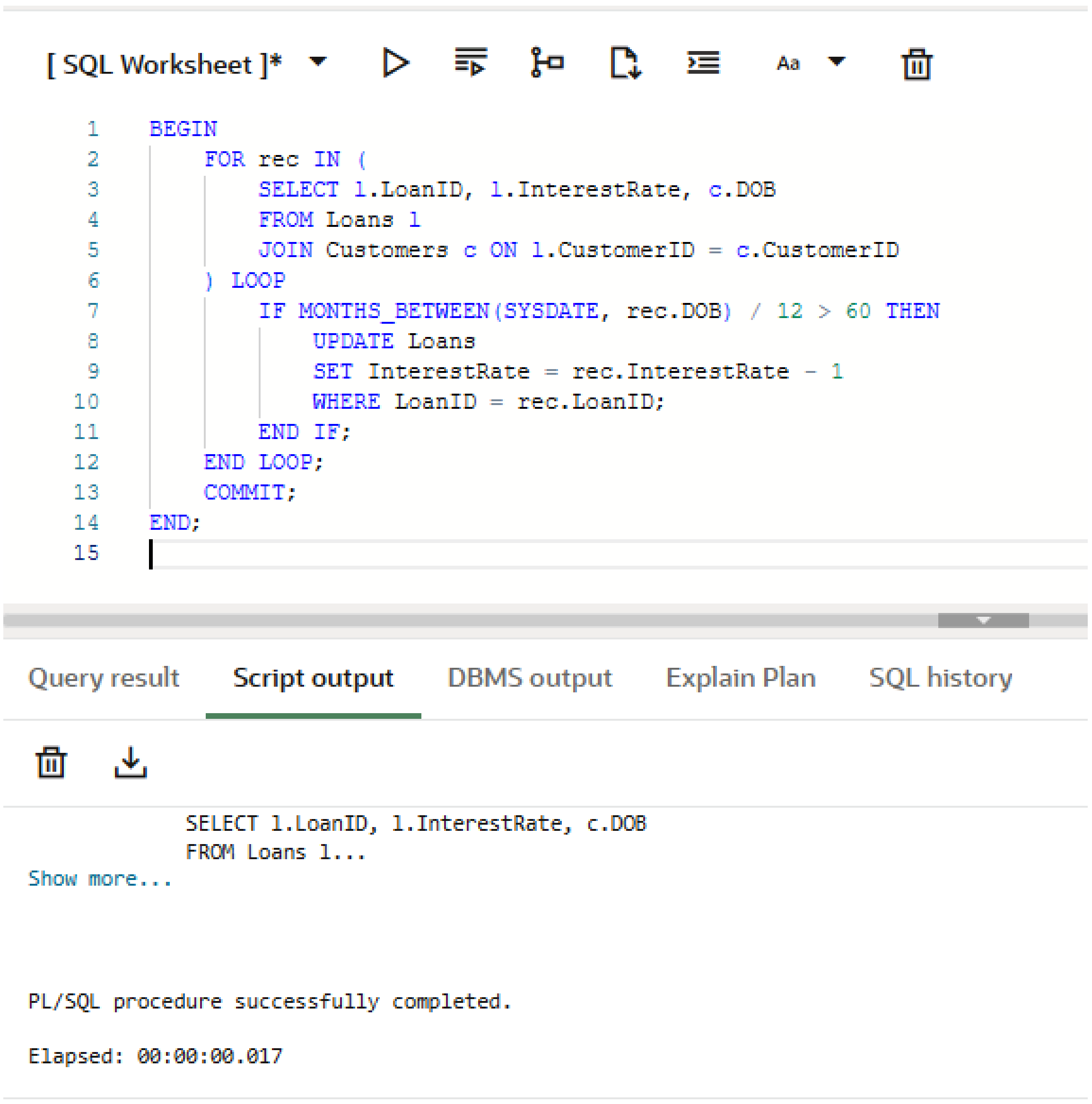
PL/SQL EXERCISES

Exercise 1: Control Structures

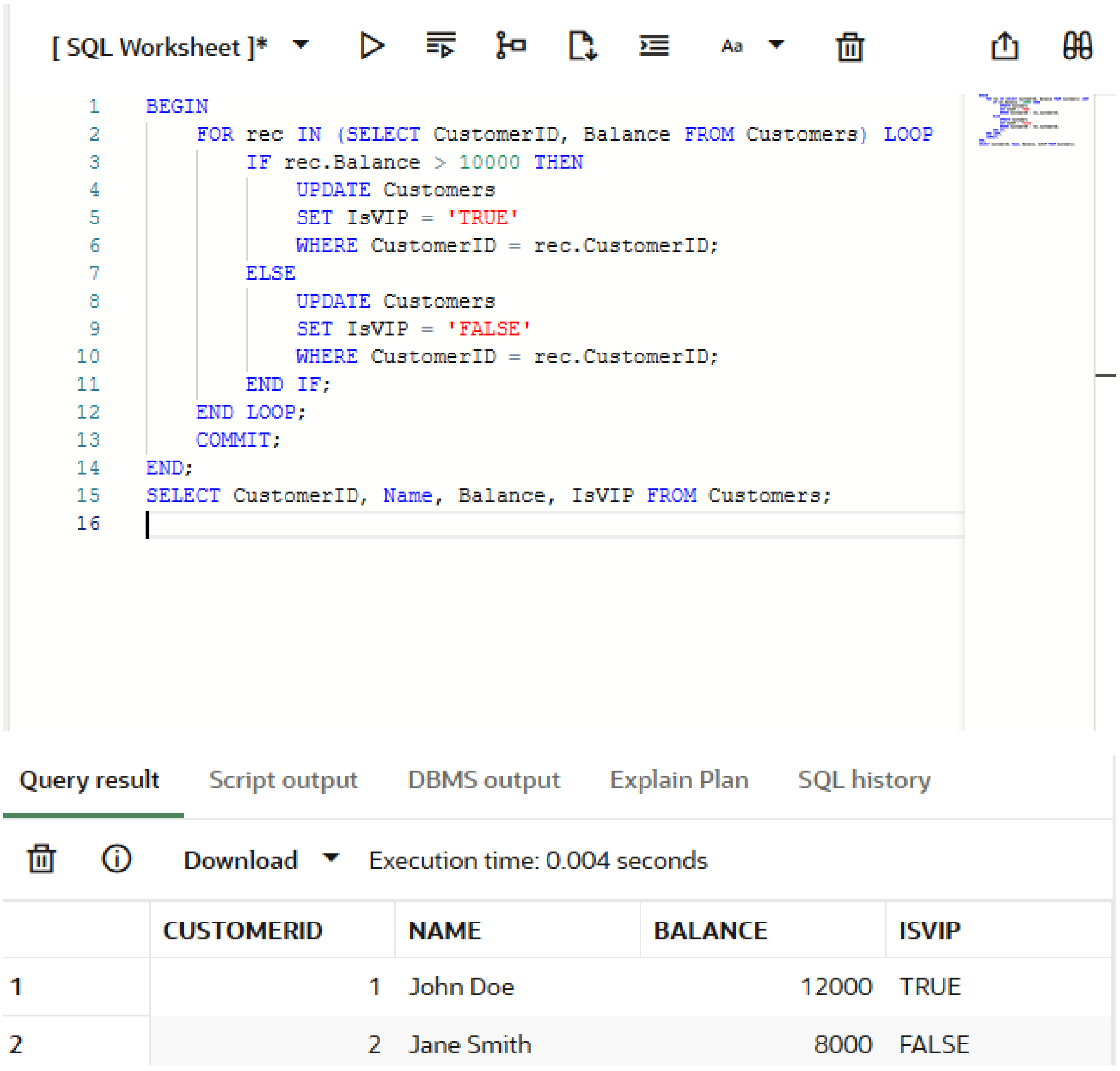
Scenario 1: The bank wants to apply a discount to loan interest rates for customers above 60 years old.
Question: Write a PL/SQL block that loops through all customers, checks their age, and if they are above 60, apply a 1% discount to their current loan interest rates.



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

```
             SELECT l.LoanID, l.InterestRate, c.DOB
             FROM Loans l...
Show more...


PL/SQL procedure successfully completed.

Elapsed: 00:00:00.017
```

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.

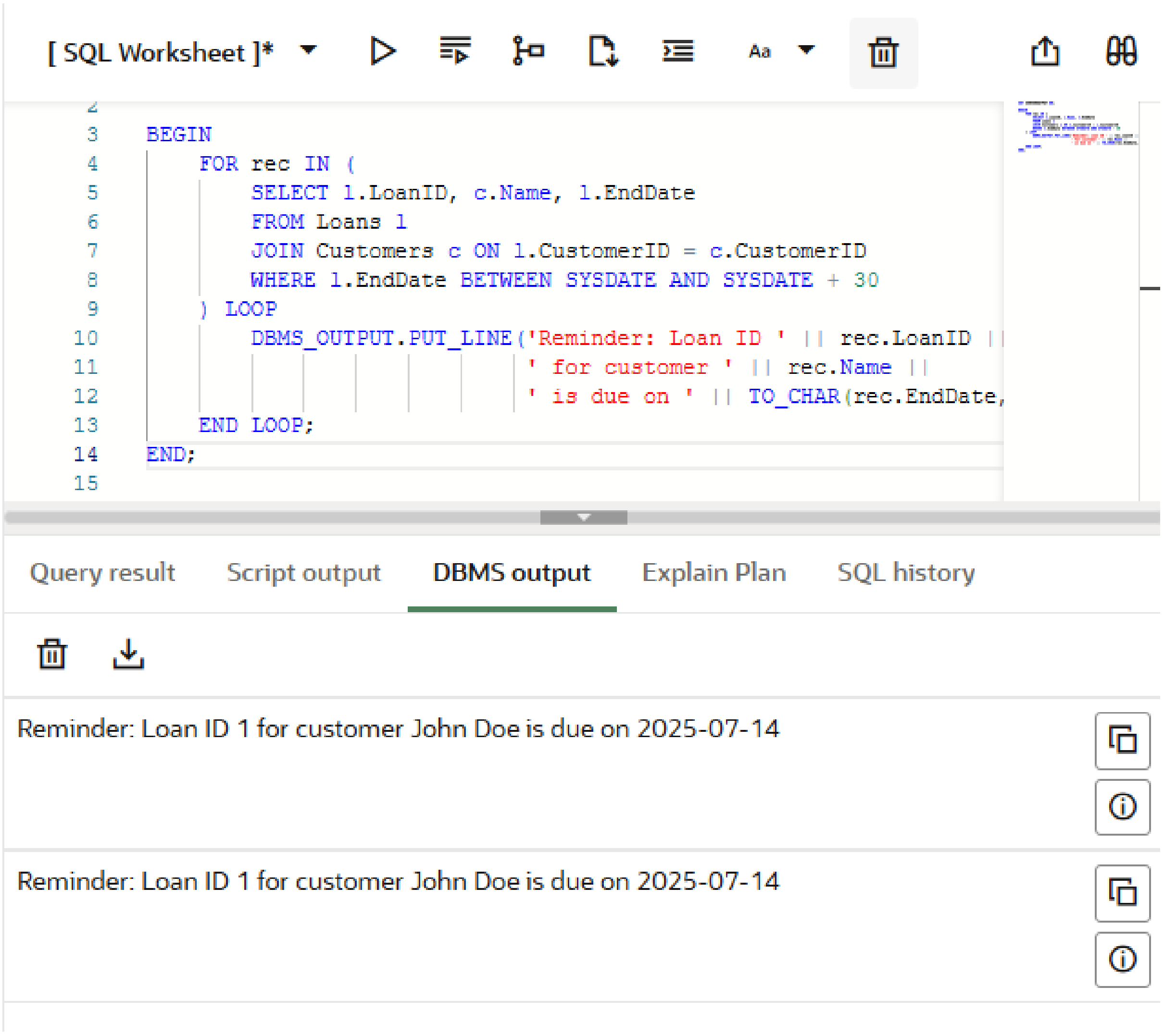


```
BEGIN
    FOR rec IN (SELECT CustomerID, Balance FROM Customers) LOOP
        IF rec.Balance > 10000 THEN
            UPDATE Customers
            SET IsVIP = 'TRUE'
            WHERE CustomerID = rec.CustomerID;
        ELSE
            UPDATE Customers
            SET IsVIP = 'FALSE'
            WHERE CustomerID = rec.CustomerID;
        END IF;
    END LOOP;
    COMMIT;
END;
SELECT CustomerID, Name, Balance, IsVIP FROM Customers;
```

Execution time: 0.004 seconds

	CUSTOMERID	NAME	BALANCE	ISVIP
1	1	John Doe	12000	TRUE
2	2	Jane Smith	8000	FALSE

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.



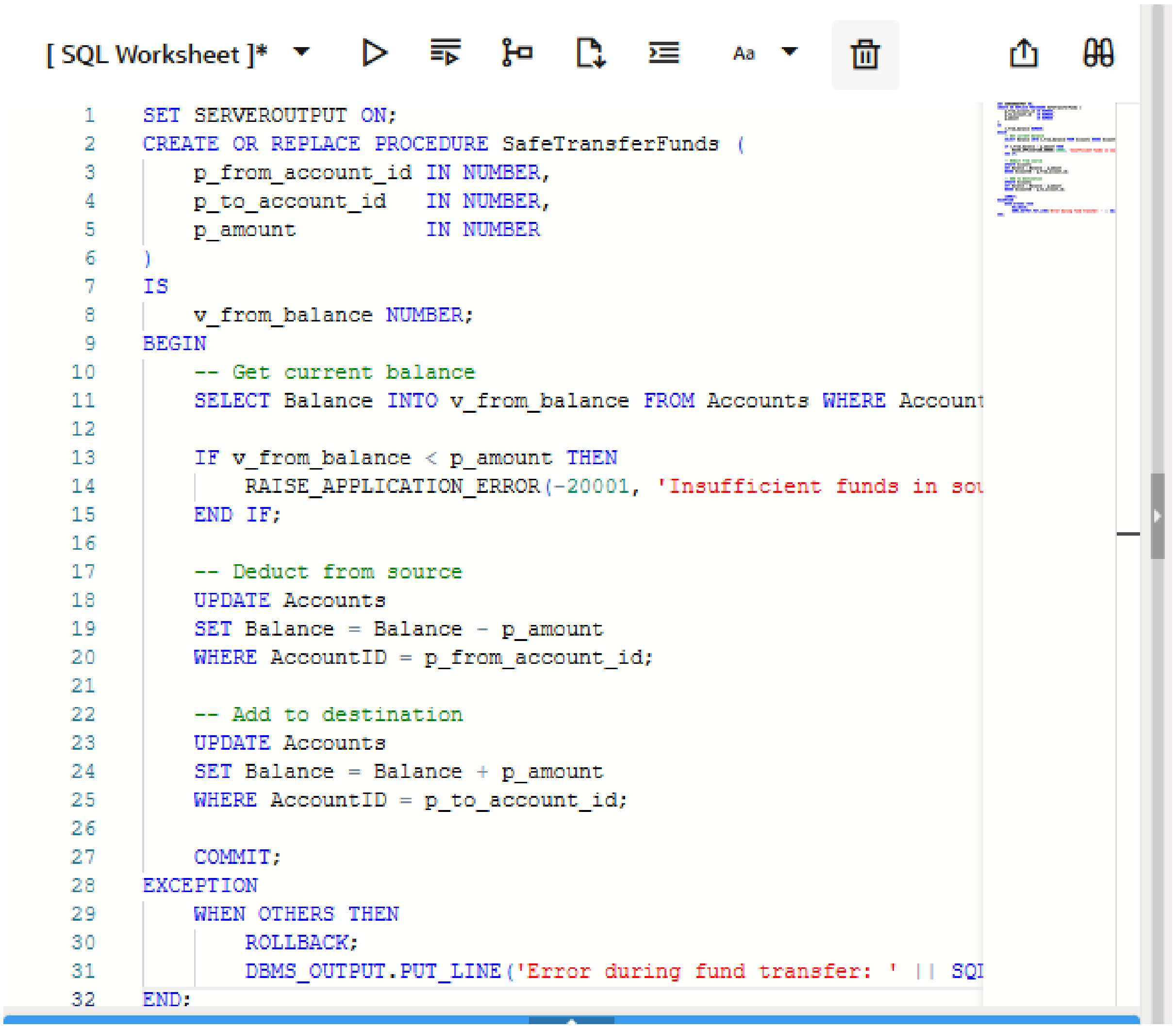
```
BEGIN
    FOR rec IN (
        SELECT l.LoanID, c.Name, l.EndDate
        FROM Loans l
        JOIN Customers c ON l.CustomerID = c.CustomerID
        WHERE l.EndDate BETWEEN SYSDATE AND SYSDATE + 30
    ) LOOP
        DBMS_OUTPUT.PUT_LINE('Reminder: Loan ID ' || rec.LoanID ||
                             ' for customer ' || rec.Name ||
                             ' is due on ' || TO_CHAR(rec.EndDate,
    END LOOP;
END;
```

```
Reminder: Loan ID 1 for customer John Doe is due on 2025-07-14
```

```
Reminder: Loan ID 1 for customer John Doe is due on 2025-07-14
```

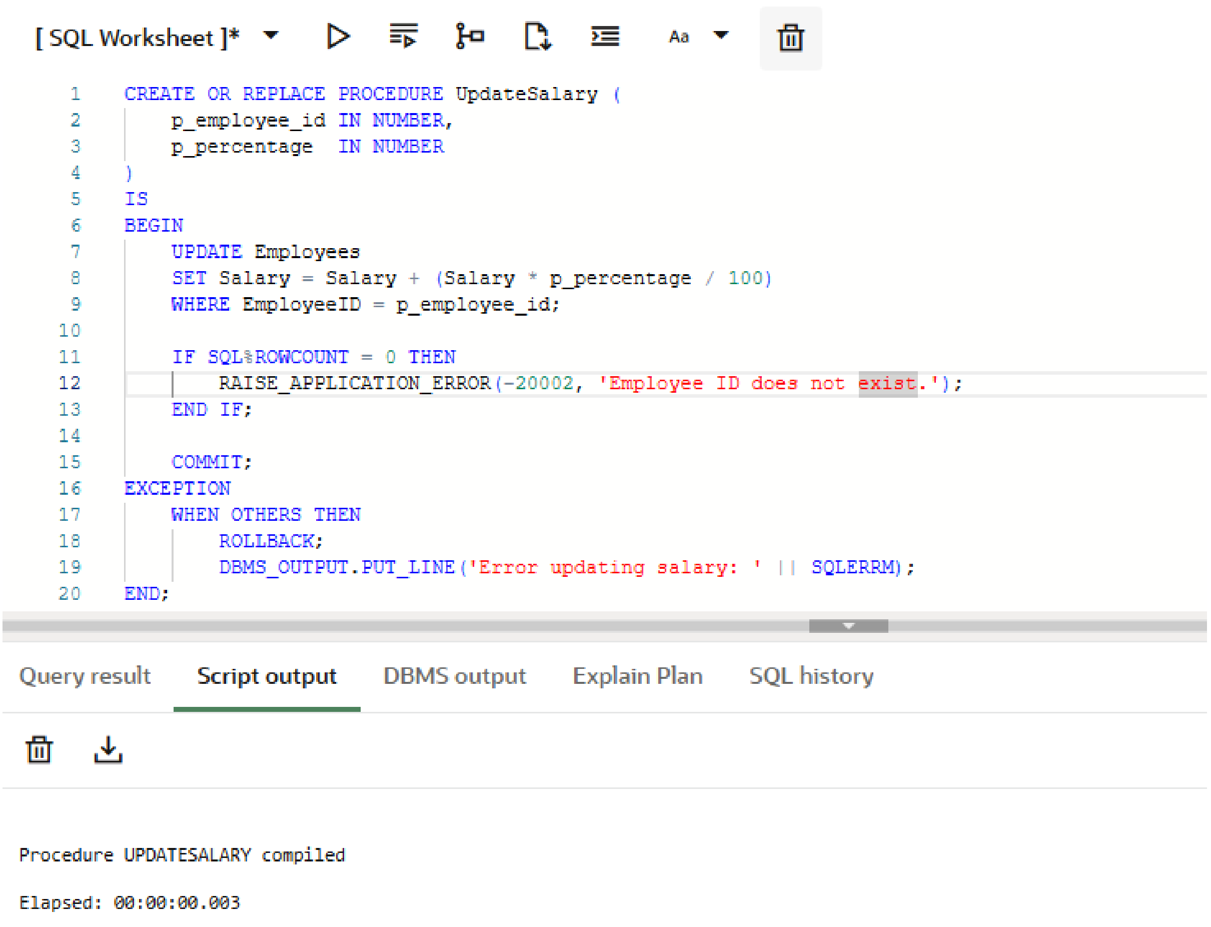
Exercise 2: Error Handling

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.



```
SET SERVEROUTPUT ON;
CREATE OR REPLACE PROCEDURE SafeTransferFunds (
    p_from_account_id IN NUMBER,
    p_to_account_id   IN NUMBER,
    p_amount          IN NUMBER
)
IS
    v_from_balance NUMBER;
BEGIN
    -- Get current balance
    SELECT Balance INTO v_from_balance FROM Accounts WHERE Account

    IF v_from_balance < p_amount THEN
        RAISE_APPLICATION_ERROR(-20001, 'Insufficient funds in sou
    END IF;

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

    -- Add to destination
    UPDATE Accounts
    SET Balance = Balance + p_amount
    WHERE AccountID = p_to_account_id;

    COMMIT;
EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
        DBMS_OUTPUT.PUT_LINE('Error during fund transfer: ' || SQI
END;
```

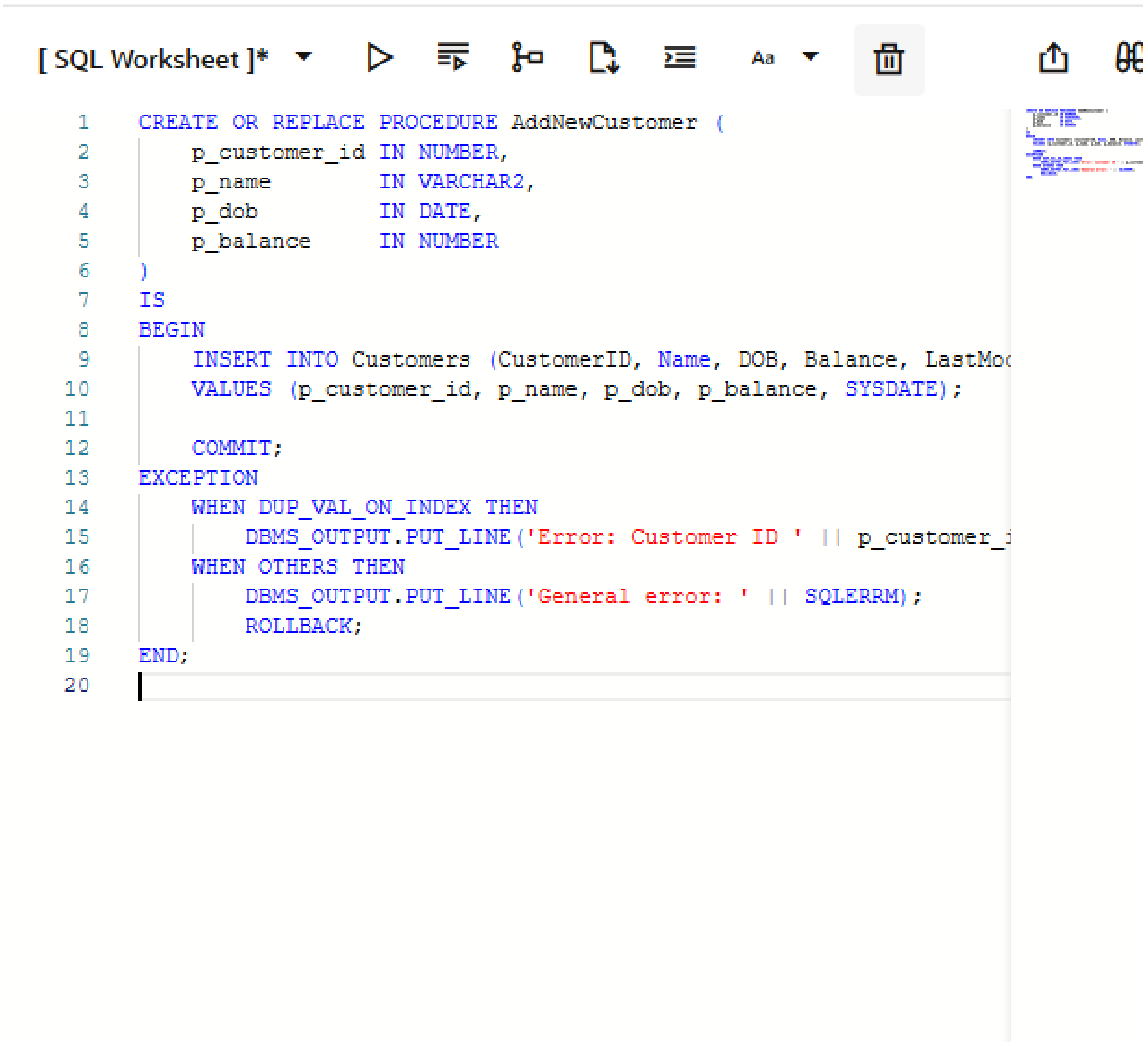
Scenario 2: Manage errors when updating employee salaries.
Question: Write a stored procedure **UpdateSalary** that increases the salary of an employee by a given percentage. If the employee ID does not exist, handle the exception and log an error message.



```
CREATE OR REPLACE PROCEDURE UpdateSalary (
    p_employee_id IN NUMBER,
    p_percentage  IN NUMBER
)
IS
BEGIN
    UPDATE Employees
    SET Salary = Salary + (Salary * p_percentage / 100)
    WHERE EmployeeID = p_employee_id;

    IF SQL%ROWCOUNT = 0 THEN
        RAISE_APPLICATION_ERROR(-20002, 'Employee ID does not exist.');
    END IF;

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

```
Procedure UPDATESALARY compiled

Elapsed: 00:00:00.003
```

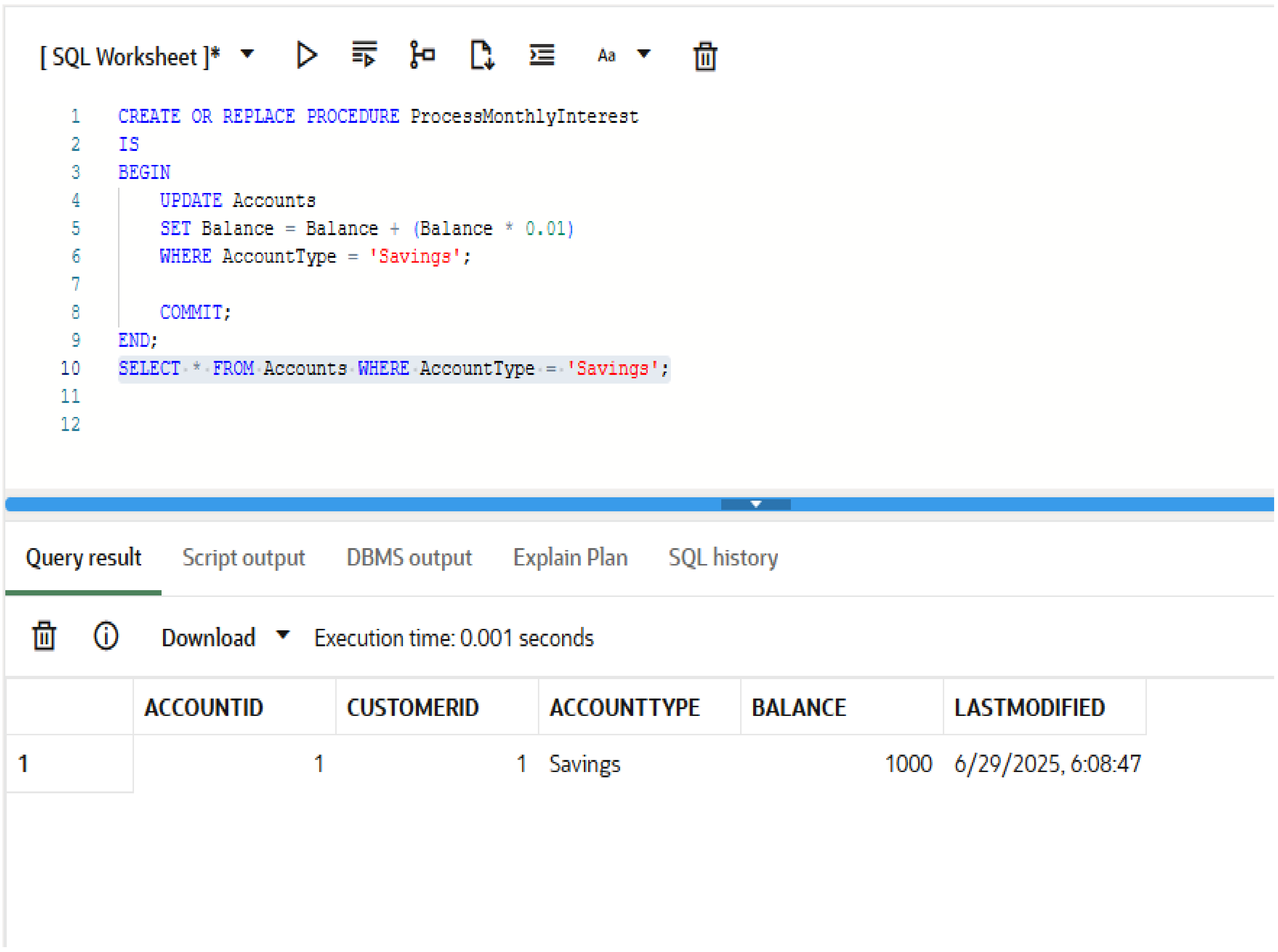
Scenario 3: Ensure data integrity when adding a new customer.
Question: Write a stored procedure **AddNewCustomer** that inserts a new customer into the Customers table. If a customer with the same ID already exists, handle the exception by logging an error and preventing the insertion.



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

    COMMIT;
EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
        DBMS_OUTPUT.PUT_LINE('Error: Customer ID ' || p_customer_i
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('General error: ' || SQLERRM);
        ROLLBACK;
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.

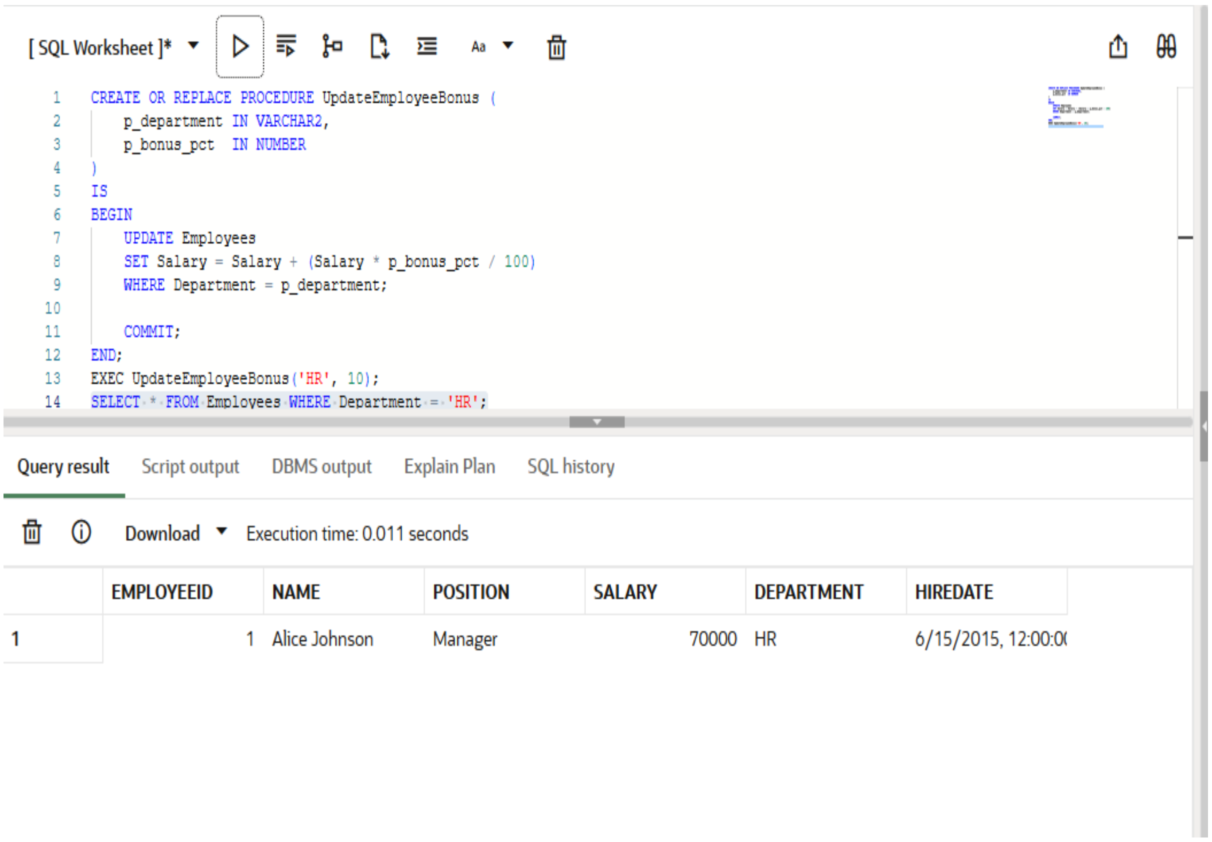


```
CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest
IS
BEGIN
    UPDATE Accounts
    SET Balance = Balance + (Balance * 0.01)
    WHERE AccountType = 'Savings';

    COMMIT;
END;
SELECT * FROM Accounts WHERE AccountType = 'Savings';
```

Execution time: 0.001 seconds

	ACCOUNTID	CUSTOMERID	ACCOUNTTYPE	BALANCE	LASTMODIFIED
1	1	1	Savings	1000	6/29/2025, 6:08:47

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.



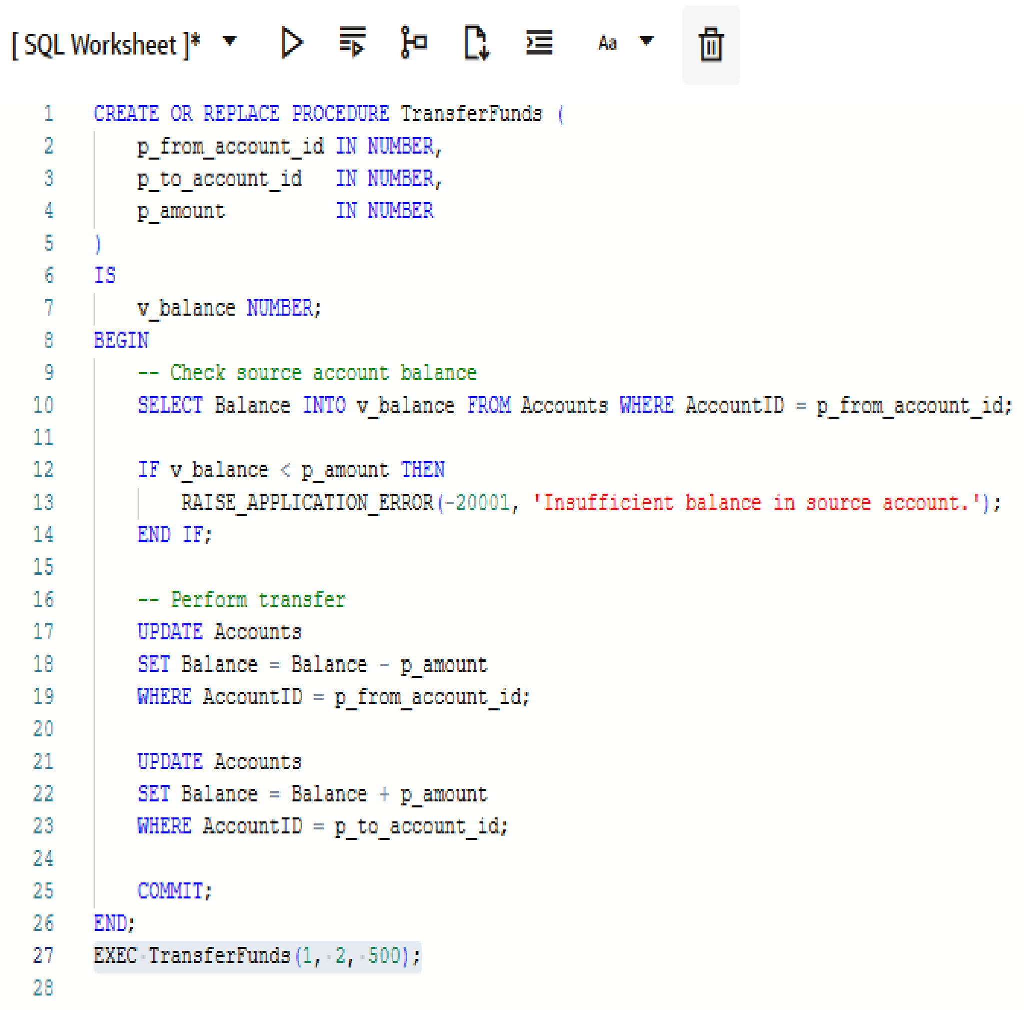
```
CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (
    p_department  IN VARCHAR2,
    p_bonus_pct   IN NUMBER
)
IS
BEGIN
    UPDATE Employees
    SET Salary = Salary + (Salary * p_bonus_pct / 100)
    WHERE Department = p_department;

    COMMIT;
END;
EXEC UpdateEmployeeBonus('HR', 10);
SELECT * FROM Employees WHERE Department = 'HR';
```

Execution time: 0.011 seconds

	EMPLOYEEID	NAME	POSITION	SALARY	DEPARTMENT	HIREDATE
1	1	Alice Johnson	Manager	70000	HR	6/15/2015, 12:00:0

Scenario 3: Customers should be able to transfer funds between their accounts.

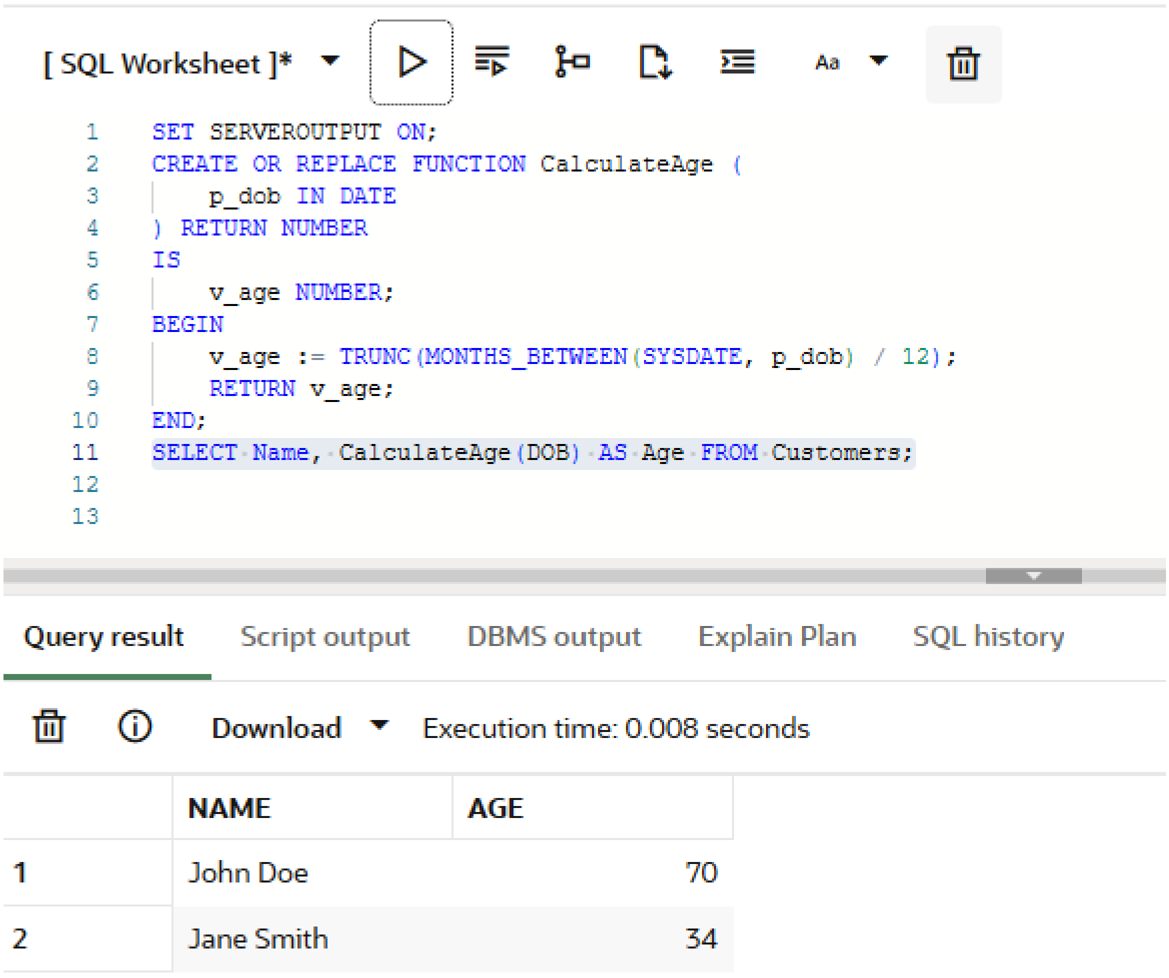
Question: Write a stored procedure **TransferFunds** that transfers a specified amount from one account to another, checking that the source account has sufficient balance before making the transfer.



```sql
CREATE OR REPLACE PROCEDURE TransferFunds (
    p_from_account_id IN NUMBER,
    p_to_account_id   IN NUMBER,
    p_amount          IN NUMBER
)
IS
    v_balance NUMBER;
BEGIN
    -- Check source account 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 balance in source account.');
    END IF;

    -- Perform transfer
    UPDATE Accounts
    SET Balance = Balance - p_amount
    WHERE AccountID = p_from_account_id;

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

    COMMIT;
END;
EXEC TransferFunds(1, 2, 500);
```

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.

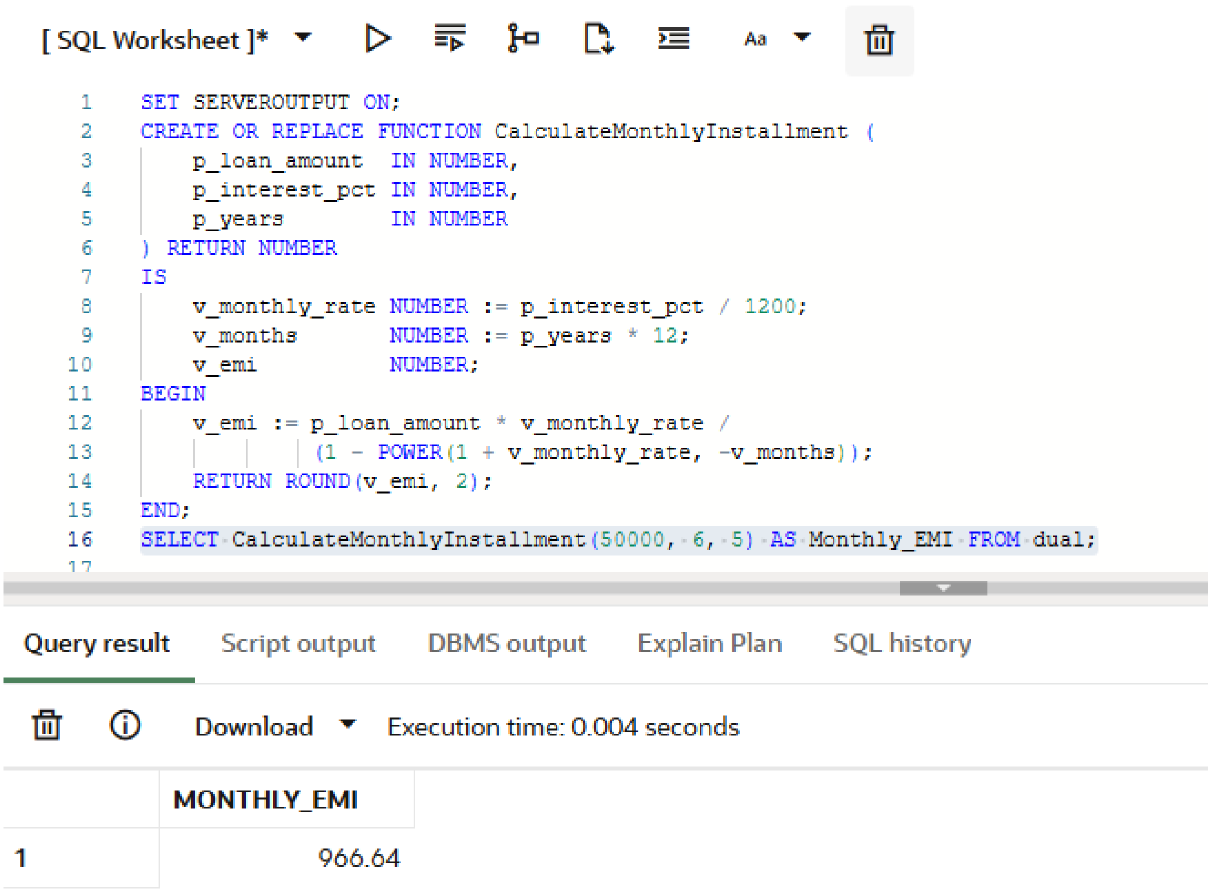


```
SET SERVEROUTPUT ON;
CREATE OR REPLACE FUNCTION CalculateAge (
    p_dob IN DATE
) RETURN NUMBER
IS
    v_age NUMBER;
BEGIN
    v_age := TRUNC(MONTHS_BETWEEN(SYSDATE, p_dob) / 12);
    RETURN v_age;
END;
SELECT Name, CalculateAge(DOB) AS Age FROM Customers;
```

Execution time: 0.008 seconds

	NAME	AGE
1	John Doe	70
2	Jane Smith	34

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.

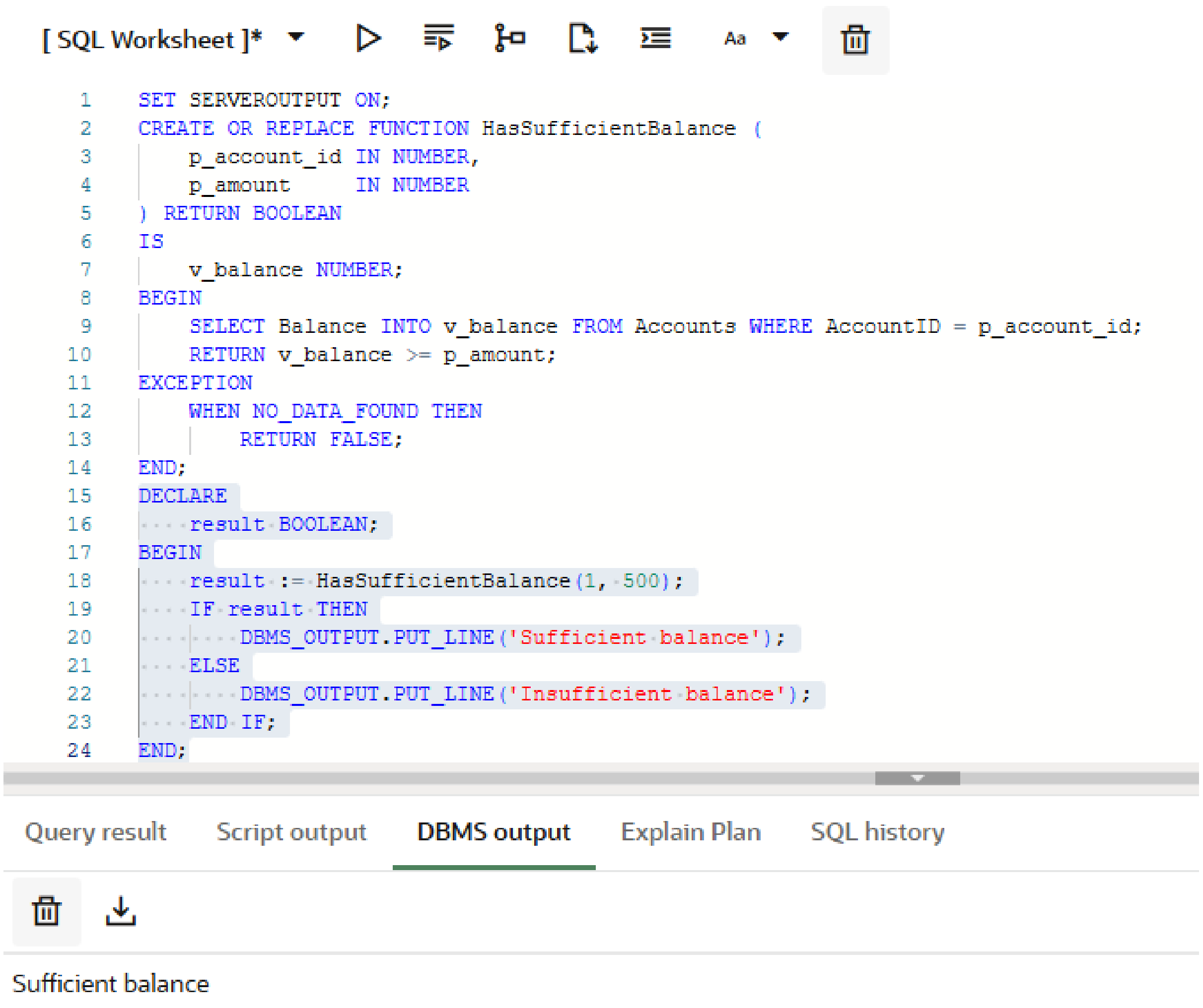


```sql
SET SERVEROUTPUT ON;
CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment (
    p_loan_amount  IN NUMBER,
    p_interest_pct IN NUMBER,
    p_years        IN NUMBER
) RETURN NUMBER
IS
    v_monthly_rate NUMBER := p_interest_pct / 1200;
    v_months       NUMBER := p_years * 12;
    v_emi          NUMBER;
BEGIN
    v_emi := p_loan_amount * v_monthly_rate /
             (1 - POWER(1 + v_monthly_rate, -v_months));
    RETURN ROUND(v_emi, 2);
END;
SELECT CalculateMonthlyInstallment(50000, 6, 5) AS Monthly_EMI FROM dual;
```

Execution time: 0.004 seconds

	MONTHLY_EMI
1	966.64

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.

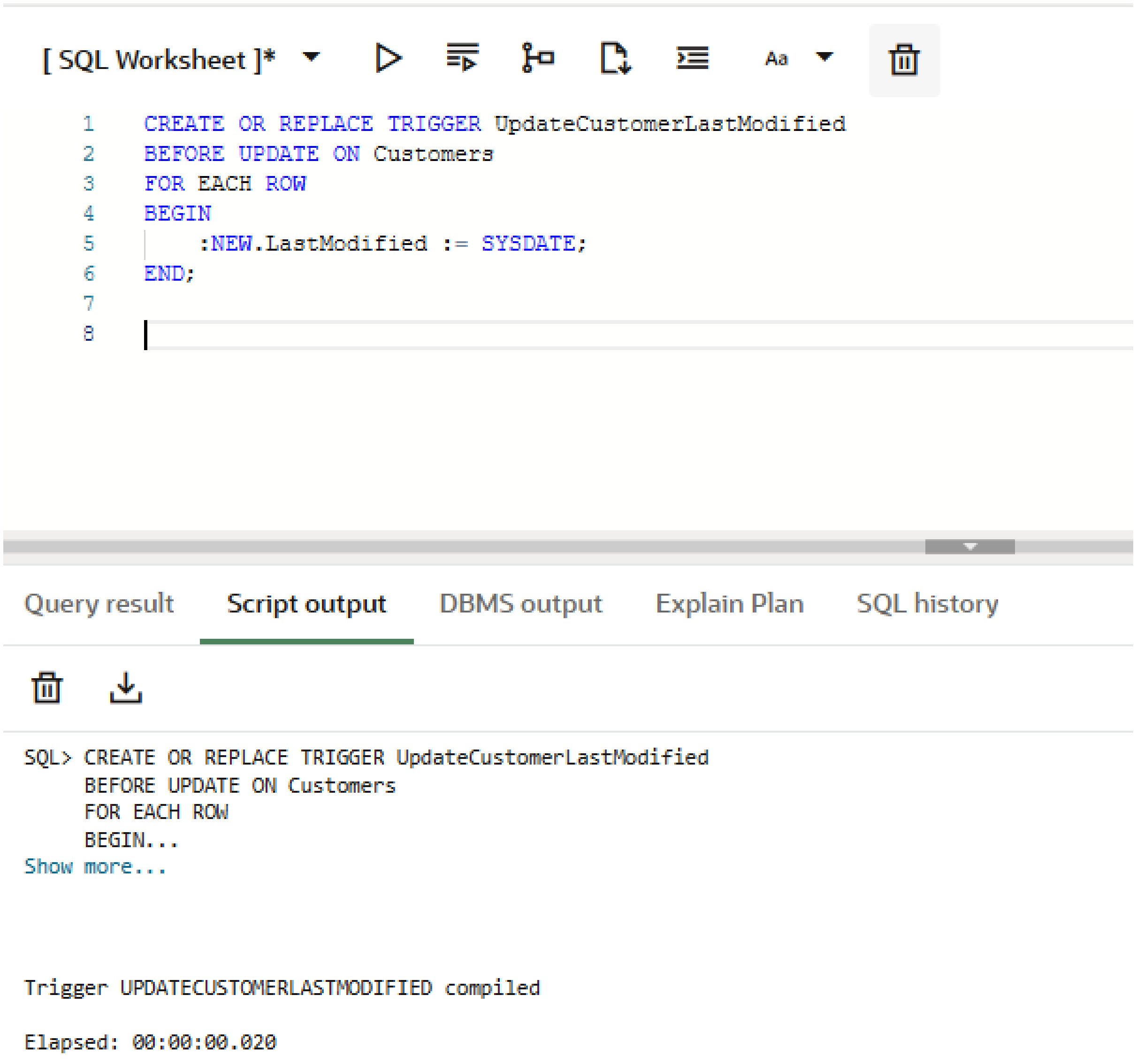


```sql
SET SERVEROUTPUT ON;
CREATE OR REPLACE FUNCTION HasSufficientBalance (
    p_account_id IN NUMBER,
    p_amount     IN NUMBER
) RETURN BOOLEAN
IS
    v_balance NUMBER;
BEGIN
    SELECT Balance INTO v_balance FROM Accounts WHERE AccountID = p_account_id;
    RETURN v_balance >= p_amount;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RETURN FALSE;
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;
```

```
Sufficient balance
```

Exercise 5: Triggers

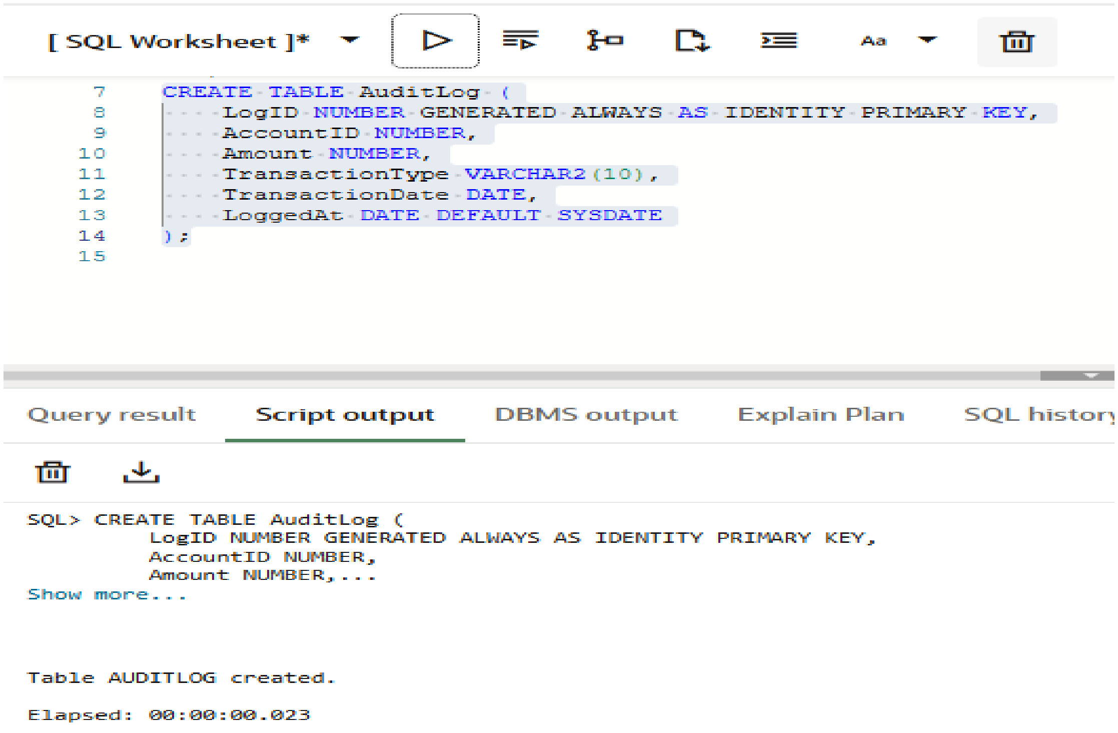
Scenario 1: Automatically update the last modified date when a customer's record is updated.
Question: Write a trigger **UpdateCustomerLastModified** that updates the LastModified column of the Customers table to the current date whenever a customer's record is updated.



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

```

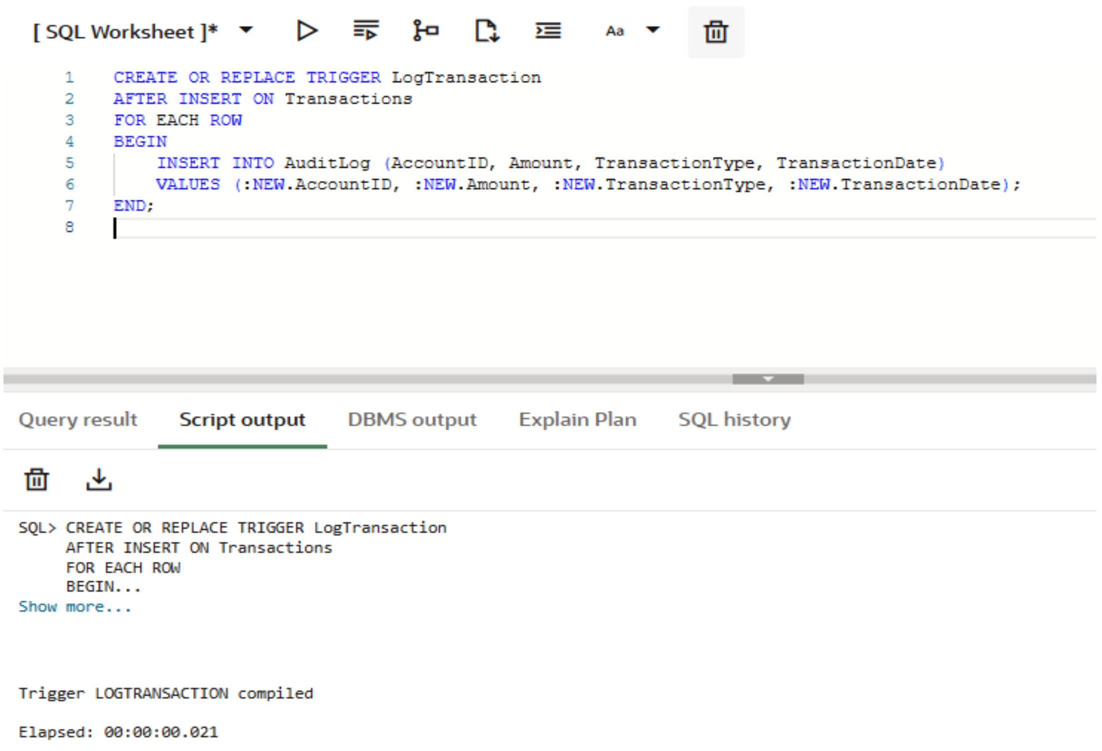
```
SQL> CREATE OR REPLACE TRIGGER UpdateCustomerLastModified
     BEFORE UPDATE ON Customers
     FOR EACH ROW
     BEGIN...
Show more...


Trigger UPDATECUSTOMERLASTMODIFIED compiled

Elapsed: 00:00:00.020
```

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

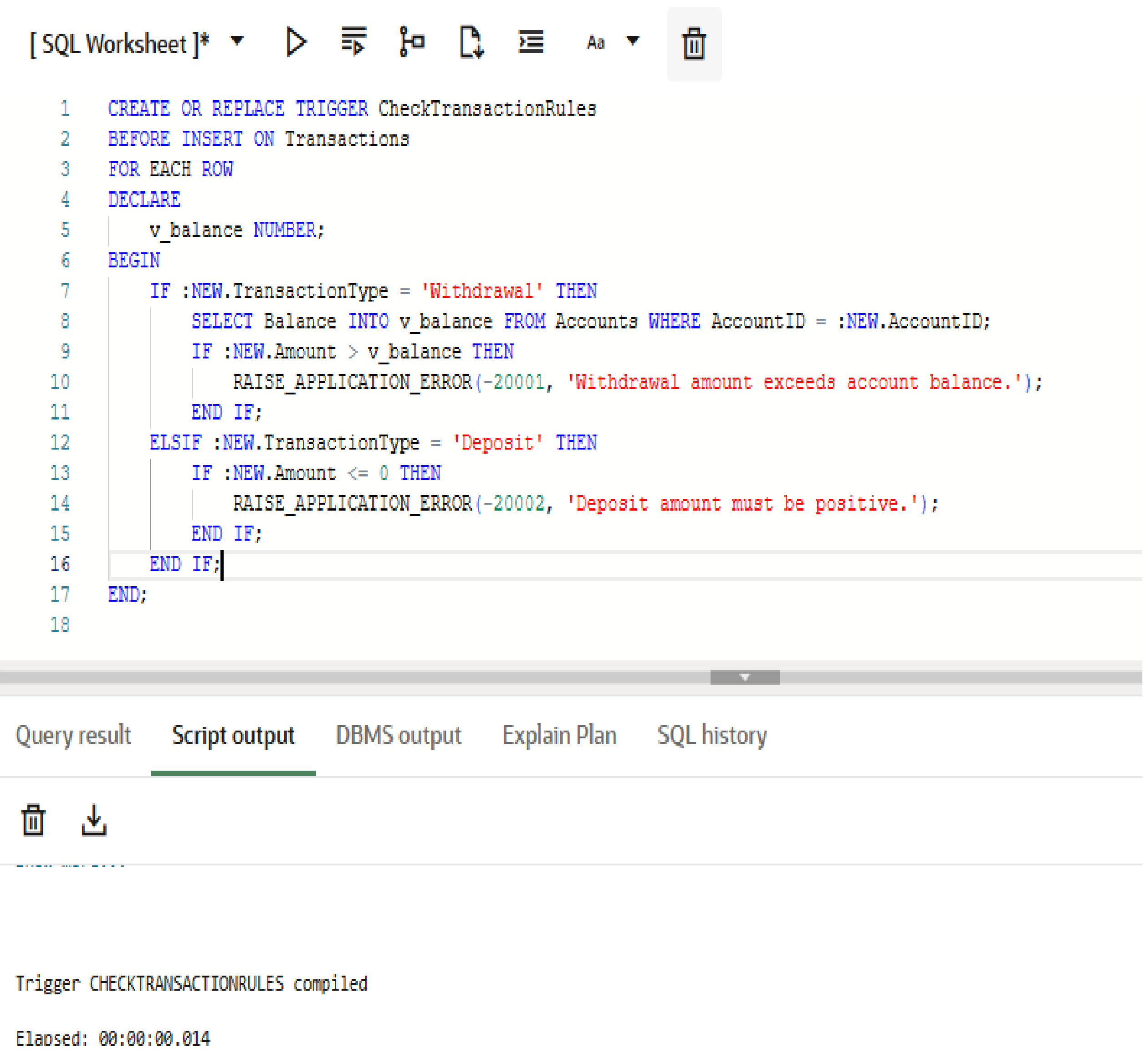
```
SQL> CREATE TABLE AuditLog (
        LogID NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
        AccountID NUMBER,
        Amount NUMBER,...
Show more...


Table AUDITLOG created.

Elapsed: 00:00:00.023
```



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

```
SQL> CREATE OR REPLACE TRIGGER LogTransaction
     AFTER INSERT ON Transactions
     FOR EACH ROW
     BEGIN...
Show more...


Trigger LOGTRANSACTION compiled

Elapsed: 00:00:00.021
```

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.

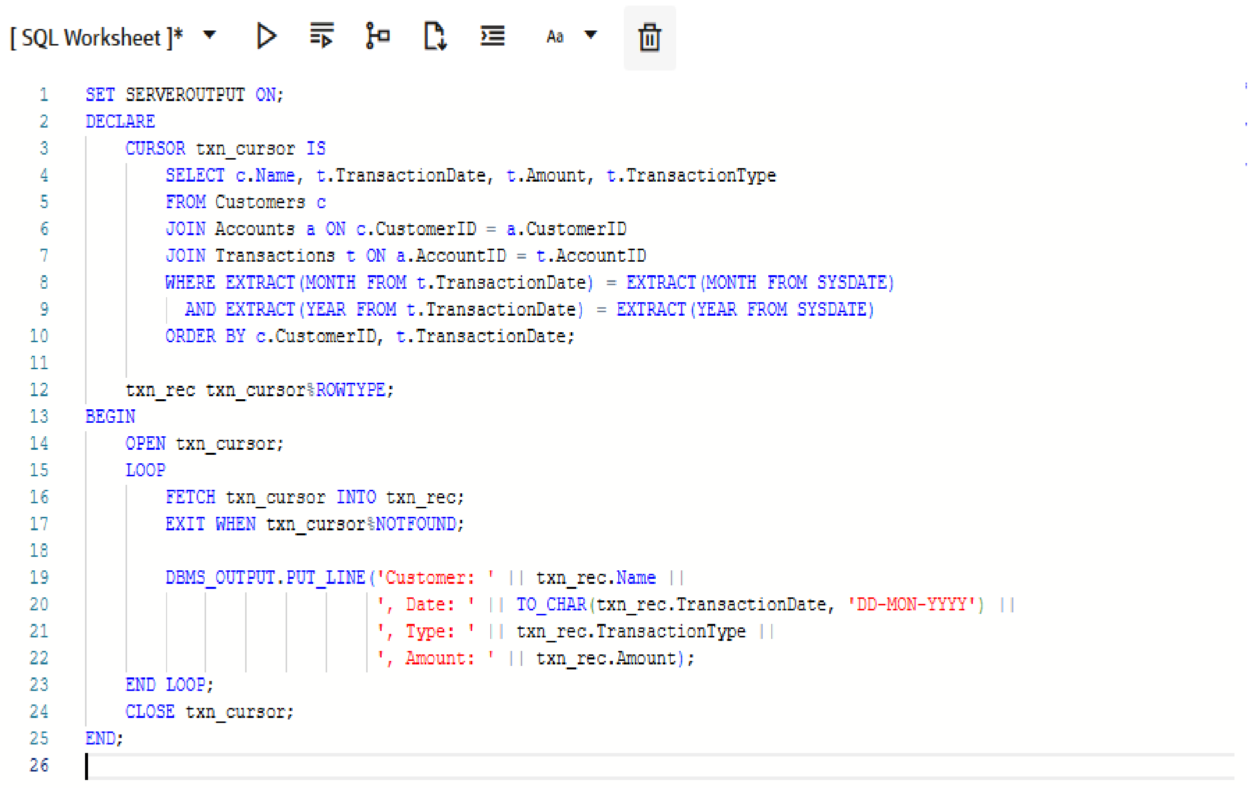


```sql
CREATE OR REPLACE TRIGGER CheckTransactionRules
BEFORE INSERT ON Transactions
FOR EACH ROW
DECLARE
    v_balance NUMBER;
BEGIN
    IF :NEW.TransactionType = 'Withdrawal' THEN
        SELECT Balance INTO v_balance FROM Accounts WHERE AccountID = :NEW.AccountID;
        IF :NEW.Amount > v_balance THEN
            RAISE_APPLICATION_ERROR(-20001, 'Withdrawal amount exceeds account balance.');
        END IF;
    ELSIF :NEW.TransactionType = 'Deposit' THEN
        IF :NEW.Amount <= 0 THEN
            RAISE_APPLICATION_ERROR(-20002, 'Deposit amount must be positive.');
        END IF;
    END IF;
END;
```

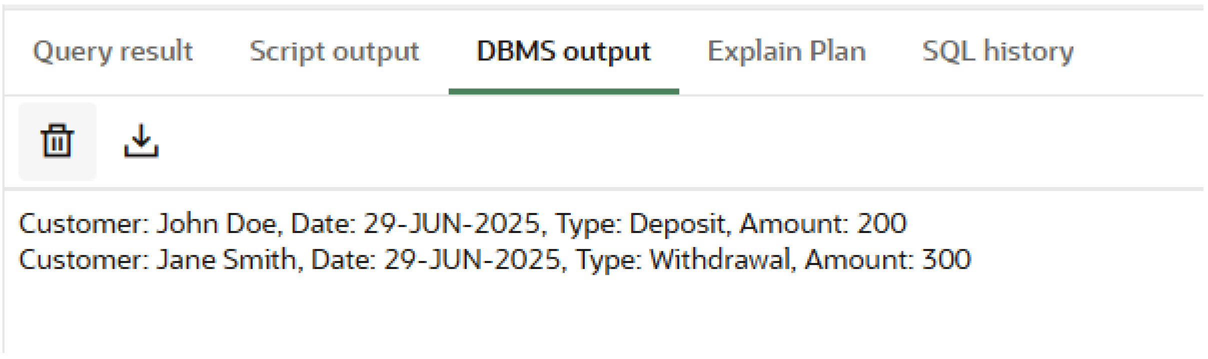
```
Trigger CHECKTRANSACTIONRULES compiled

Elapsed: 00:00:00.014
```

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.

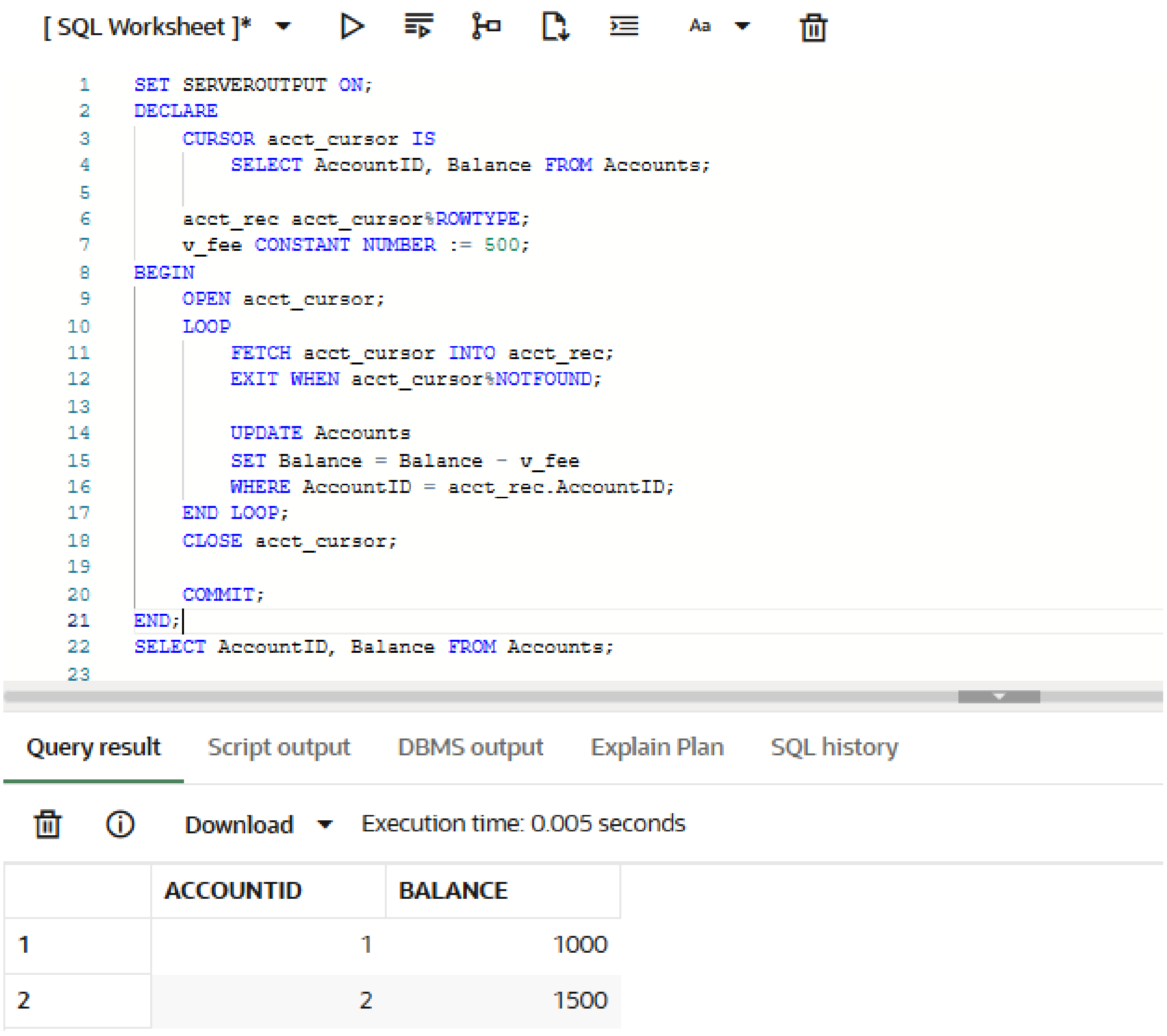


```
SET SERVEROUTPUT ON;
DECLARE
    CURSOR txn_cursor IS
        SELECT c.Name, t.TransactionDate, t.Amount, t.TransactionType
        FROM Customers c
        JOIN Accounts a ON c.CustomerID = a.CustomerID
        JOIN Transactions t ON a.AccountID = t.AccountID
        WHERE EXTRACT(MONTH FROM t.TransactionDate) = EXTRACT(MONTH FROM SYSDATE)
          AND EXTRACT(YEAR FROM t.TransactionDate) = EXTRACT(YEAR FROM SYSDATE)
        ORDER BY c.CustomerID, t.TransactionDate;

    txn_rec txn_cursor%ROWTYPE;
BEGIN
    OPEN txn_cursor;
    LOOP
        FETCH txn_cursor INTO txn_rec;
        EXIT WHEN txn_cursor%NOTFOUND;

        DBMS_OUTPUT.PUT_LINE('Customer: ' || txn_rec.Name ||
                             ', Date: ' || TO_CHAR(txn_rec.TransactionDate, 'DD-MON-YYYY') ||
                             ', Type: ' || txn_rec.TransactionType ||
                             ', Amount: ' || txn_rec.Amount);
    END LOOP;
    CLOSE txn_cursor;
END;
```



```
Customer: John Doe, Date: 29-JUN-2025, Type: Deposit, Amount: 200
Customer: Jane Smith, Date: 29-JUN-2025, Type: Withdrawal, Amount: 300
```

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.

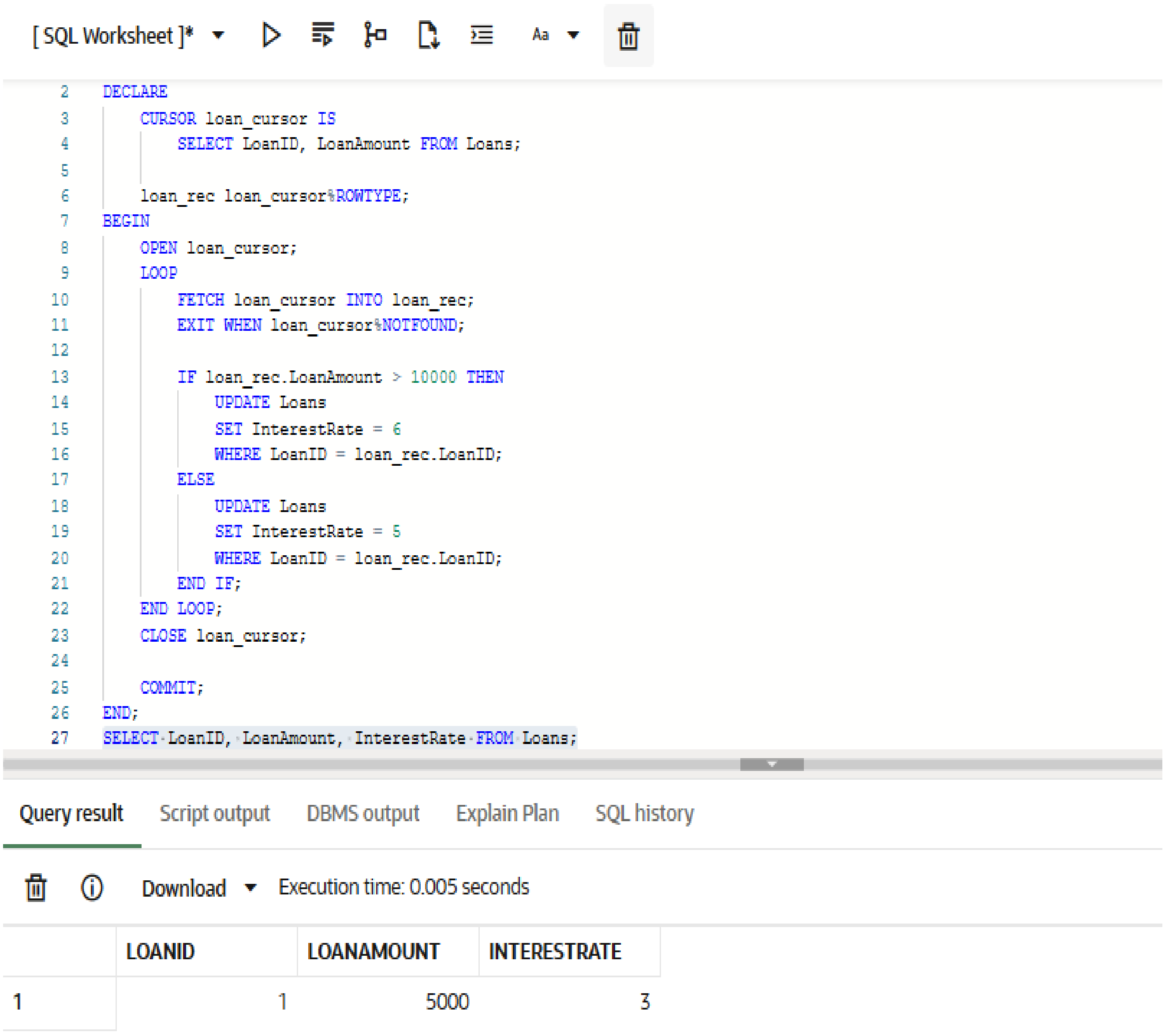


```
SET SERVEROUTPUT ON;
DECLARE
    CURSOR acct_cursor IS
        SELECT AccountID, Balance FROM Accounts;

    acct_rec acct_cursor%ROWTYPE;
    v_fee CONSTANT NUMBER := 500;
BEGIN
    OPEN acct_cursor;
    LOOP
        FETCH acct_cursor INTO acct_rec;
        EXIT WHEN acct_cursor%NOTFOUND;

        UPDATE Accounts
        SET Balance = Balance - v_fee
        WHERE AccountID = acct_rec.AccountID;
    END LOOP;
    CLOSE acct_cursor;

    COMMIT;
END;
SELECT AccountID, Balance FROM Accounts;
```

Execution time: 0.005 seconds

	ACCOUNTID	BALANCE
1	1	1000
2	2	1500

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.

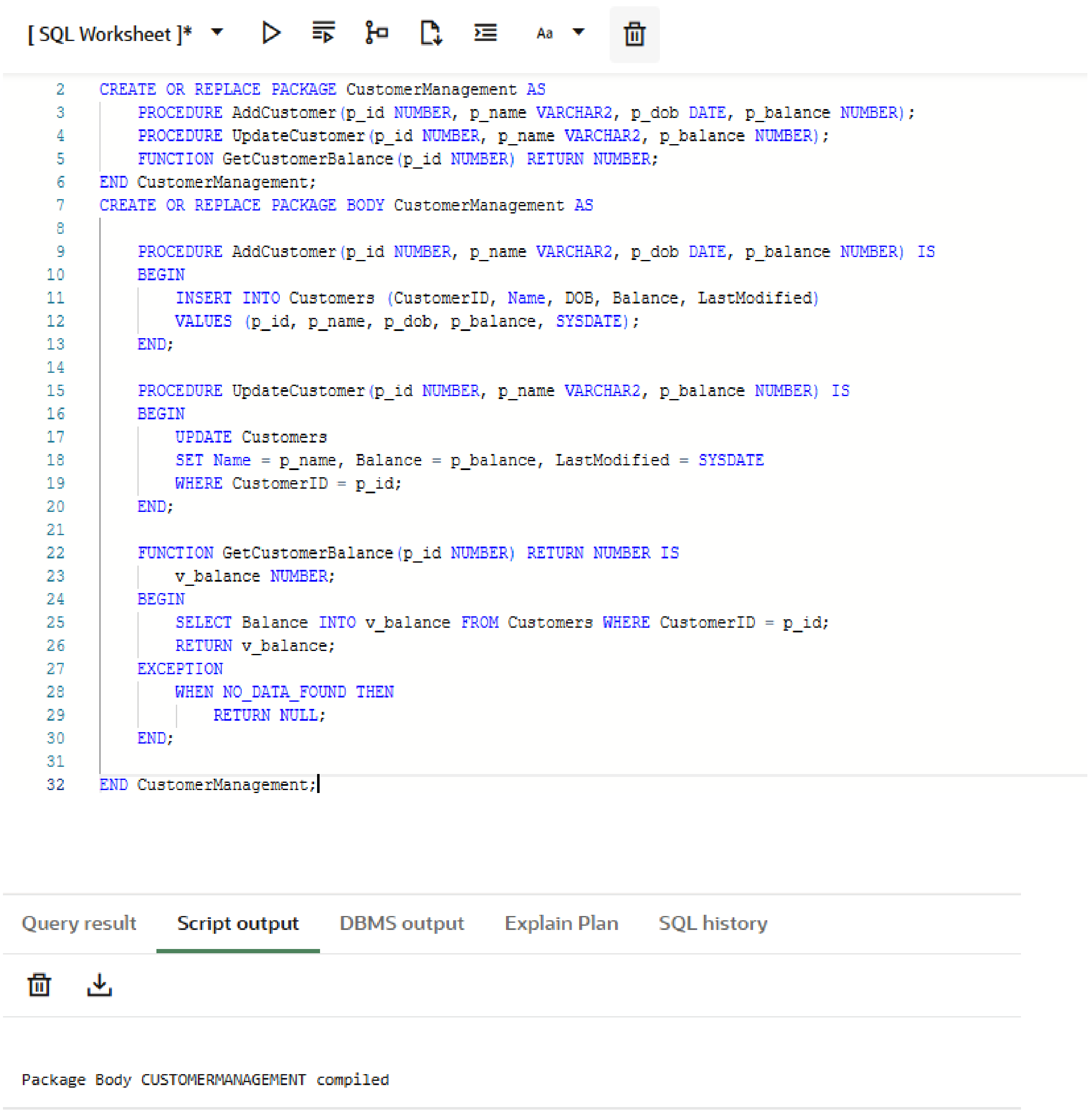


```
DECLARE
    CURSOR loan_cursor IS
        SELECT LoanID, LoanAmount FROM Loans;

    loan_rec loan_cursor%ROWTYPE;
BEGIN
    OPEN loan_cursor;
    LOOP
        FETCH loan_cursor INTO loan_rec;
        EXIT WHEN loan_cursor%NOTFOUND;

        IF loan_rec.LoanAmount > 10000 THEN
            UPDATE Loans
            SET InterestRate = 6
            WHERE LoanID = loan_rec.LoanID;
        ELSE
            UPDATE Loans
            SET InterestRate = 5
            WHERE LoanID = loan_rec.LoanID;
        END IF;
    END LOOP;
    CLOSE loan_cursor;

    COMMIT;
END;
SELECT LoanID, LoanAmount, InterestRate FROM Loans;
```

	LOANID	LOANAMOUNT	INTERESTRATE
1	1	5000	3

Exercise 7: Packages

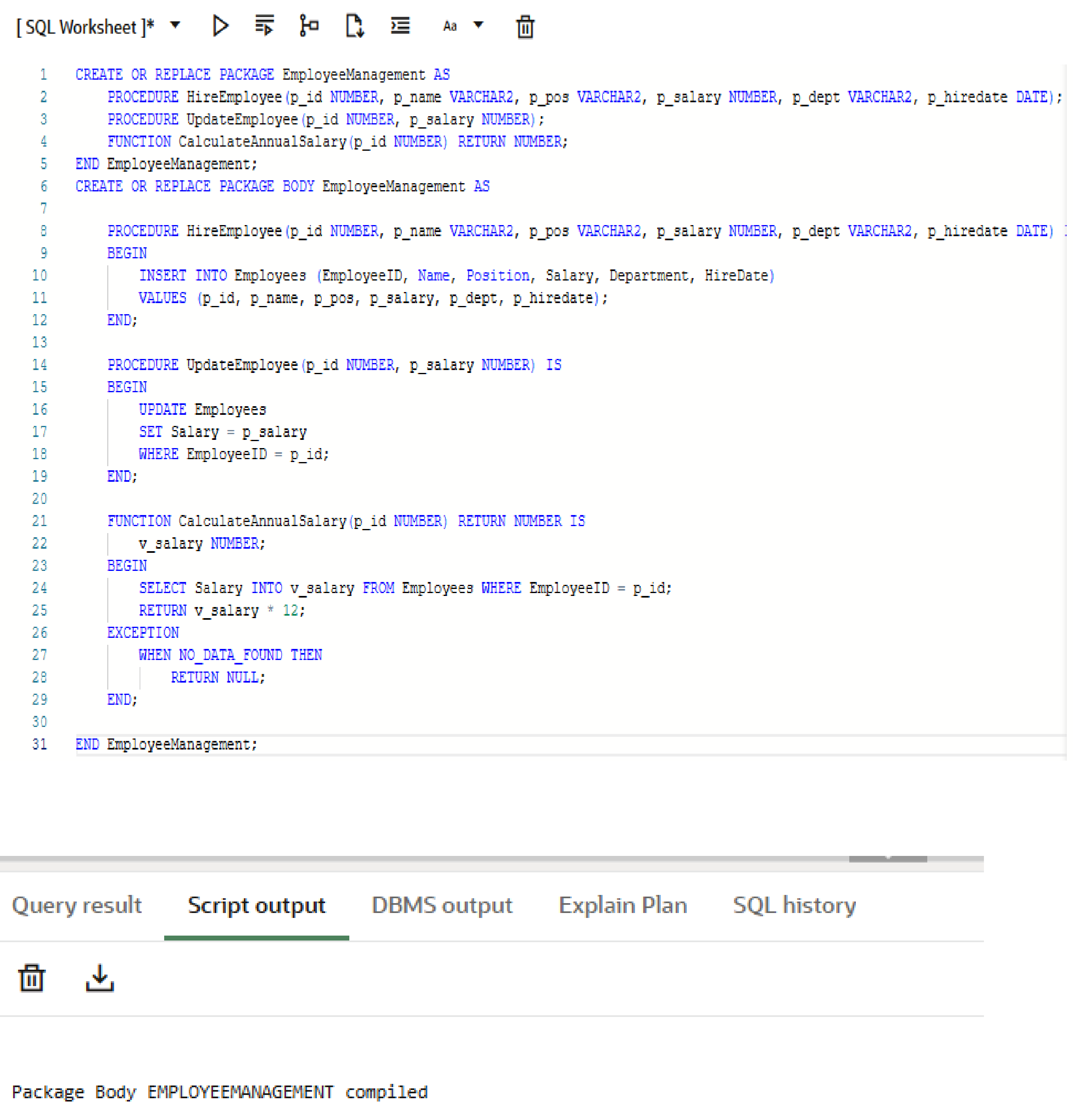
Scenario 1: Group all customer-related procedures and functions into a package.
Question: Create a package **CustomerManagement** with procedures for adding a new customer, updating customer details, and a function to get customer balance.



```
CREATE OR REPLACE PACKAGE CustomerManagement AS
    PROCEDURE AddCustomer(p_id NUMBER, p_name VARCHAR2, p_dob DATE, p_balance NUMBER);
    PROCEDURE UpdateCustomer(p_id NUMBER, p_name VARCHAR2, p_balance NUMBER);
    FUNCTION GetCustomerBalance(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 (CustomerID, Name, DOB, Balance, LastModified)
        VALUES (p_id, p_name, p_dob, p_balance, SYSDATE);
    END;

    PROCEDURE UpdateCustomer(p_id NUMBER, p_name VARCHAR2, p_balance NUMBER) IS
    BEGIN
        UPDATE Customers
        SET Name = p_name, Balance = p_balance, LastModified = SYSDATE
        WHERE CustomerID = p_id;
    END;

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

END CustomerManagement;
```

```
Package Body CUSTOMERMANAGEMENT compiled
```

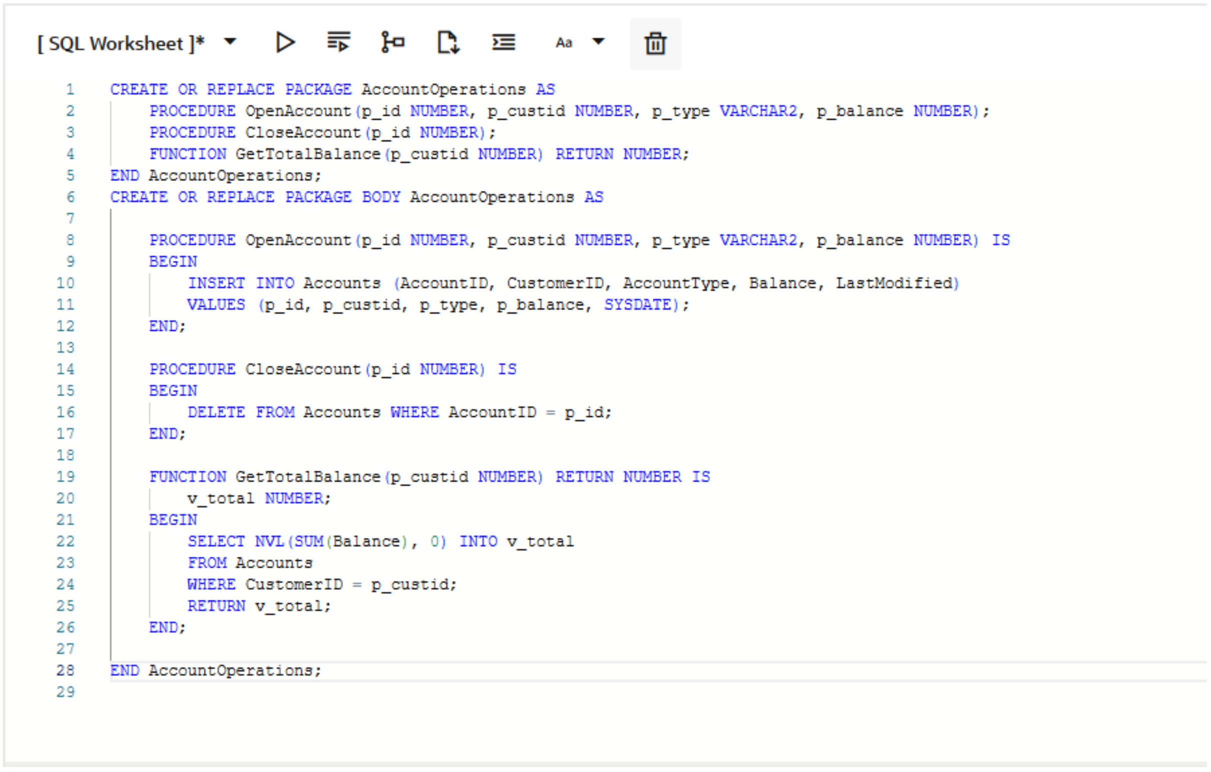
Scenario 2: Create a package to manage employee data.
Question: Write a package **EmployeeManagement** with procedures to hire new employees, update employee details, and a function to calculate annual salary.



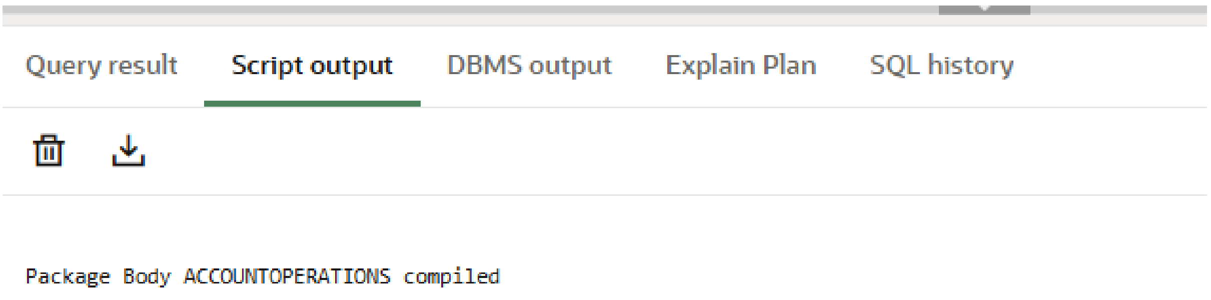
```sql
CREATE OR REPLACE PACKAGE EmployeeManagement AS
    PROCEDURE HireEmployee(p_id NUMBER, p_name VARCHAR2, p_pos VARCHAR2, p_salary NUMBER, p_dept VARCHAR2, p_hiredate DATE);
    PROCEDURE UpdateEmployee(p_id NUMBER, p_salary NUMBER);
    FUNCTION CalculateAnnualSalary(p_id NUMBER) RETURN NUMBER;
END EmployeeManagement;
CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

    PROCEDURE HireEmployee(p_id NUMBER, p_name VARCHAR2, p_pos VARCHAR2, p_salary NUMBER, p_dept VARCHAR2, p_hiredate DATE) I
    BEGIN
        INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate)
        VALUES (p_id, p_name, p_pos, p_salary, p_dept, p_hiredate);
    END;

    PROCEDURE UpdateEmployee(p_id NUMBER, p_salary NUMBER) IS
    BEGIN
        UPDATE Employees
        SET Salary = p_salary
        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;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            RETURN NULL;
    END;

END EmployeeManagement;
```

```
Package Body EMPLOYEEMANAGEMENT compiled
```

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.


[SQL Worksheet]*

```
CREATE OR REPLACE PACKAGE AccountOperations AS
    PROCEDURE OpenAccount(p_id NUMBER, p_custid NUMBER, p_type VARCHAR2, p_balance NUMBER);
    PROCEDURE CloseAccount(p_id NUMBER);
    FUNCTION GetTotalBalance(p_custid NUMBER) RETURN NUMBER;
END AccountOperations;
CREATE OR REPLACE PACKAGE BODY AccountOperations AS

    PROCEDURE OpenAccount(p_id NUMBER, p_custid NUMBER, p_type VARCHAR2, p_balance NUMBER) IS
    BEGIN
        INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified)
        VALUES (p_id, p_custid, p_type, p_balance, SYSDATE);
    END;

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

    FUNCTION GetTotalBalance(p_custid NUMBER) RETURN NUMBER IS
        v_total NUMBER;
    BEGIN
        SELECT NVL(SUM(Balance), 0) INTO v_total
        FROM Accounts
        WHERE CustomerID = p_custid;
        RETURN v_total;
    END;

END AccountOperations;
```



```
Package Body ACCOUNTOPERATIONS compiled
```