**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 c IN (SELECT CustomerID, DOB FROM Customers) LOOP

IF MONTHS\_BETWEEN(SYSDATE, c.DOB) / 12 > 60 THEN

UPDATE Loans

SET InterestRate = InterestRate - 1

WHERE CustomerID = c.CustomerID;

DBMS\_OUTPUT.PUT\_LINE('Applied 1% discount to customer ID: ' || c.CustomerID);

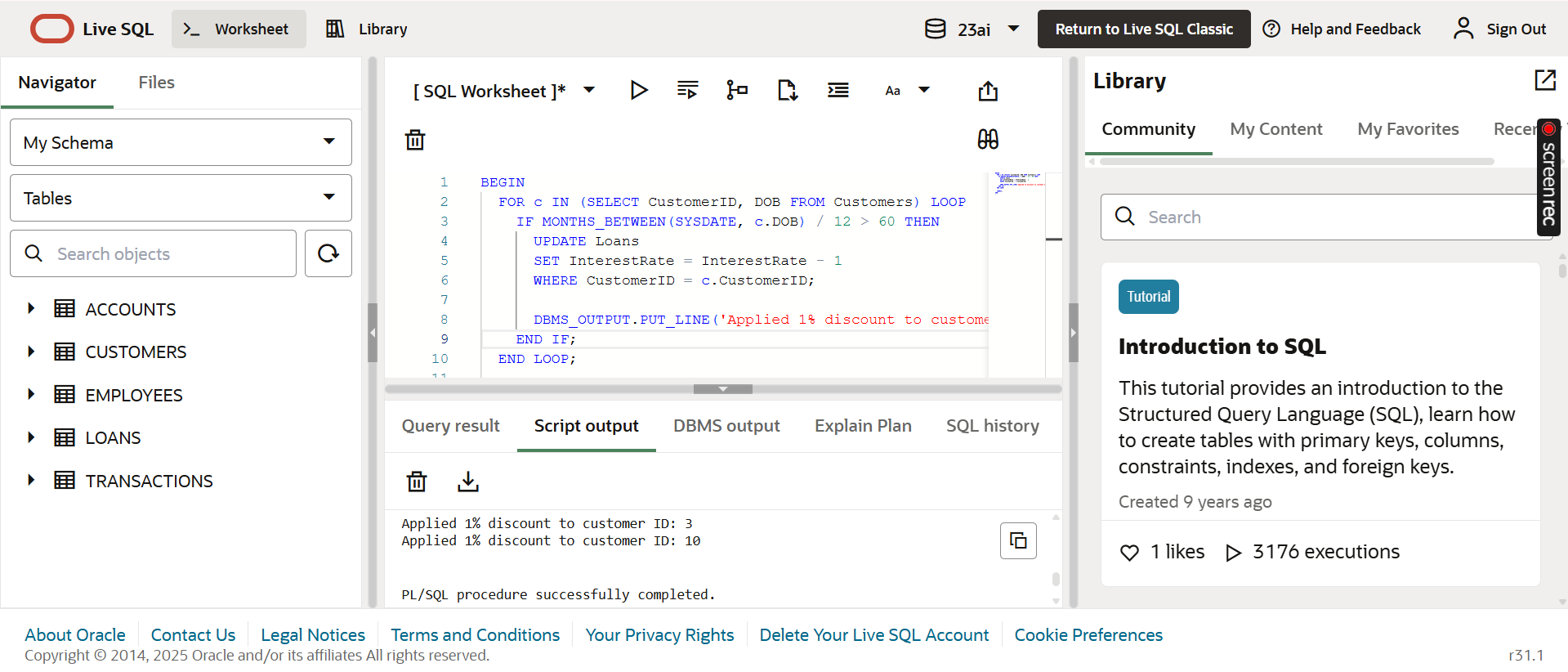
END IF;

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));

begin

    for c in (select CustomerID,Balance from CUSTOMERS) loop

     IF c.Balance>10000 then

        update Customers

        set IsVIP='Y'

        where CustomerID=c.CustomerID;

     else

        update Customers

        set IsVIP='N'

        where CustomerID=c.CustomerID;

     end if;

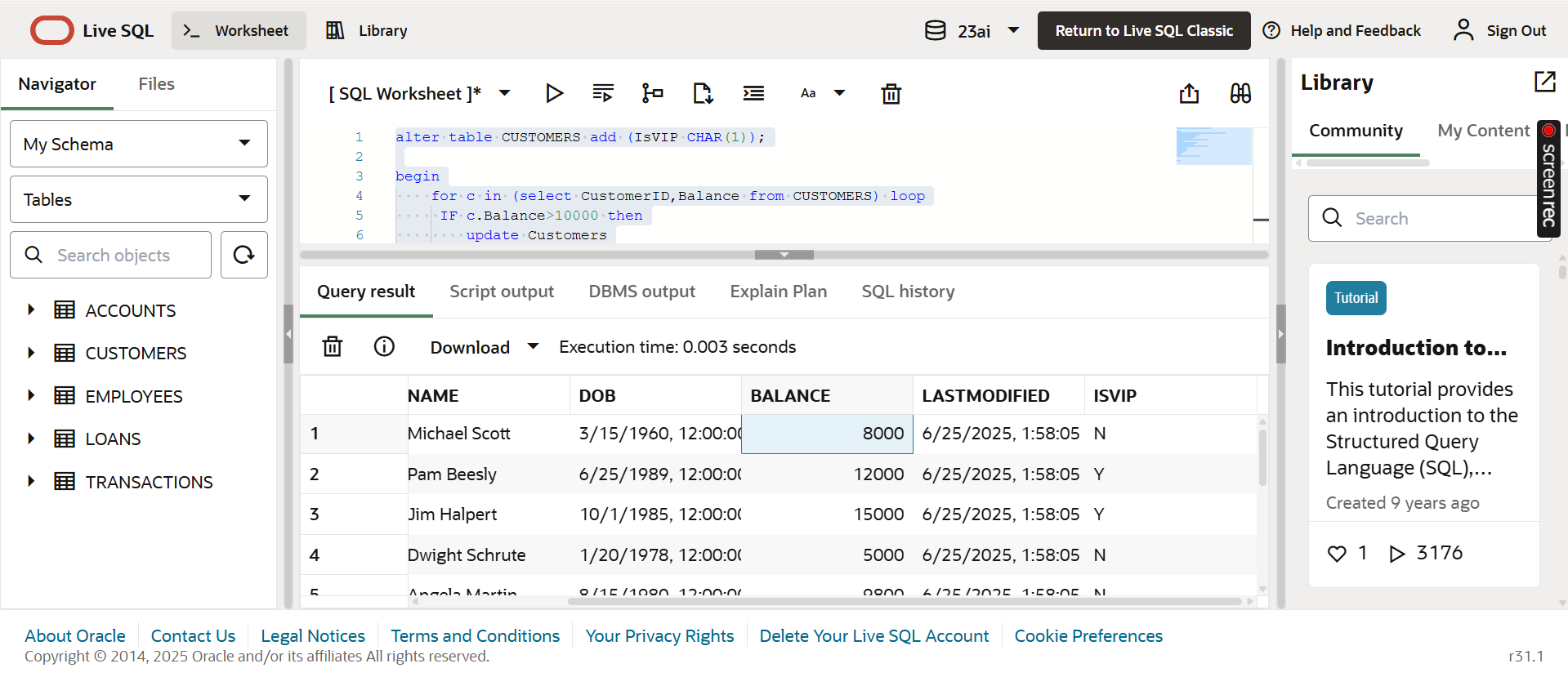
    end loop;

    COMMIT;

    end;

    /

    select \* from CUSTOMERS;



**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 l IN (

    SELECT l.LoanID, l.CustomerID, c.Name, l.ENDDATE

    FROM Loans l

    JOIN Customers c ON c.CustomerID = l.CustomerID

    WHERE l.ENDDATE BETWEEN SYSDATE AND SYSDATE + 30

  ) LOOP

    DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan ID ' || l.LoanID ||

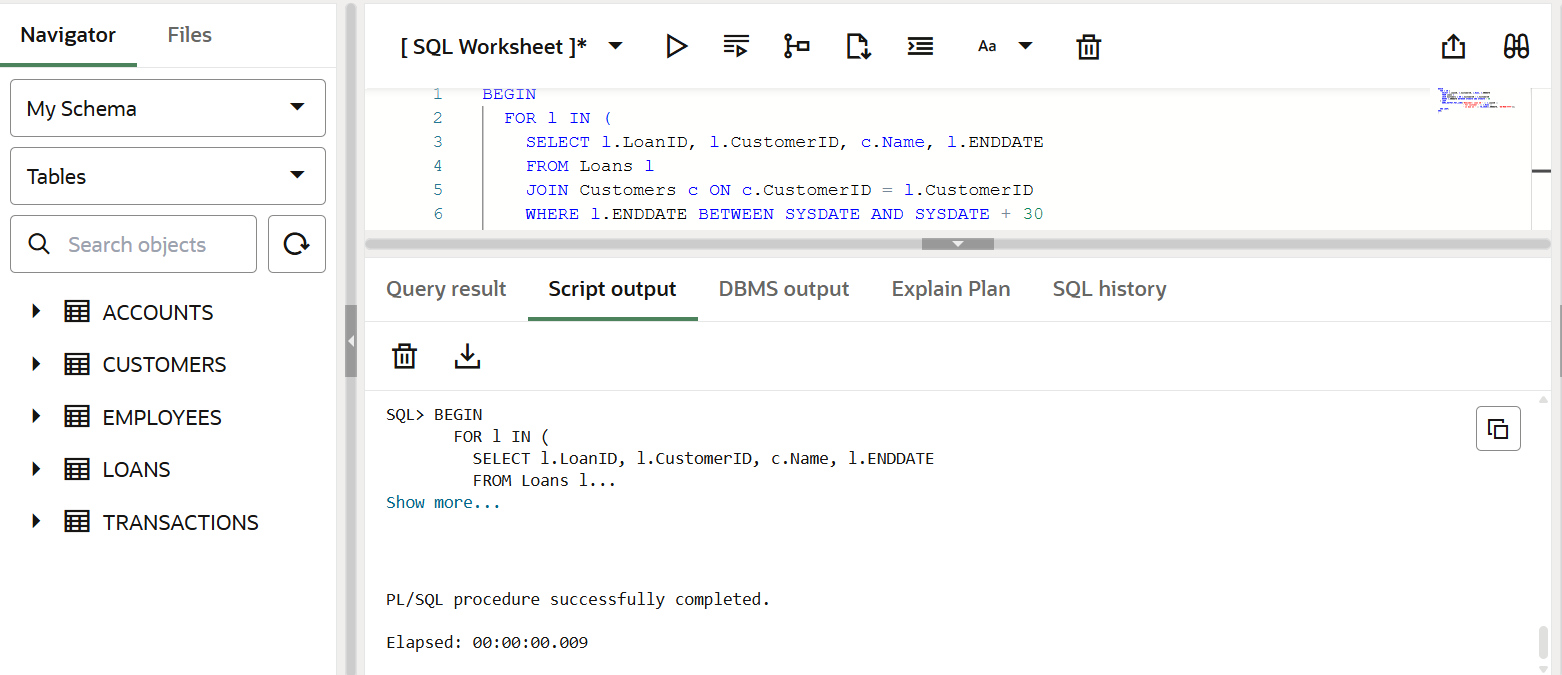
                         ' for customer ' || l.Name ||

                         ' is due on ' || TO\_CHAR(l.ENDDATE, 'DD-MON-YYYY'));

  END LOOP;

END;

/



**Exercise 2: Error Handling**

**Scenario 1:** Handle exceptions during fund transfers between accounts.

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

CREATE OR REPLACE PROCEDURE SafeTransferFunds (

  p\_from\_account\_id IN NUMBER,

  p\_to\_account\_id   IN NUMBER,

  p\_amount          IN NUMBER

) AS

  v\_from\_balance NUMBER;

BEGIN

  SELECT Balance INTO v\_from\_balance

  FROM Accounts

  WHERE AccountID = p\_from\_account\_id

  FOR UPDATE;

  IF v\_from\_balance < p\_amount THEN

    RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient funds in source account.');

  END IF;

  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;

  DBMS\_OUTPUT.PUT\_LINE('Transfer successful: ' || p\_amount || ' transferred from Account ' ||

                        p\_from\_account\_id || ' to Account ' || p\_to\_account\_id);

EXCEPTION

  WHEN OTHERS THEN

    -- Rollback the whole transaction in case of any error

    ROLLBACK;

    DBMS\_OUTPUT.PUT\_LINE('Transfer failed: ' || SQLERRM);

END;

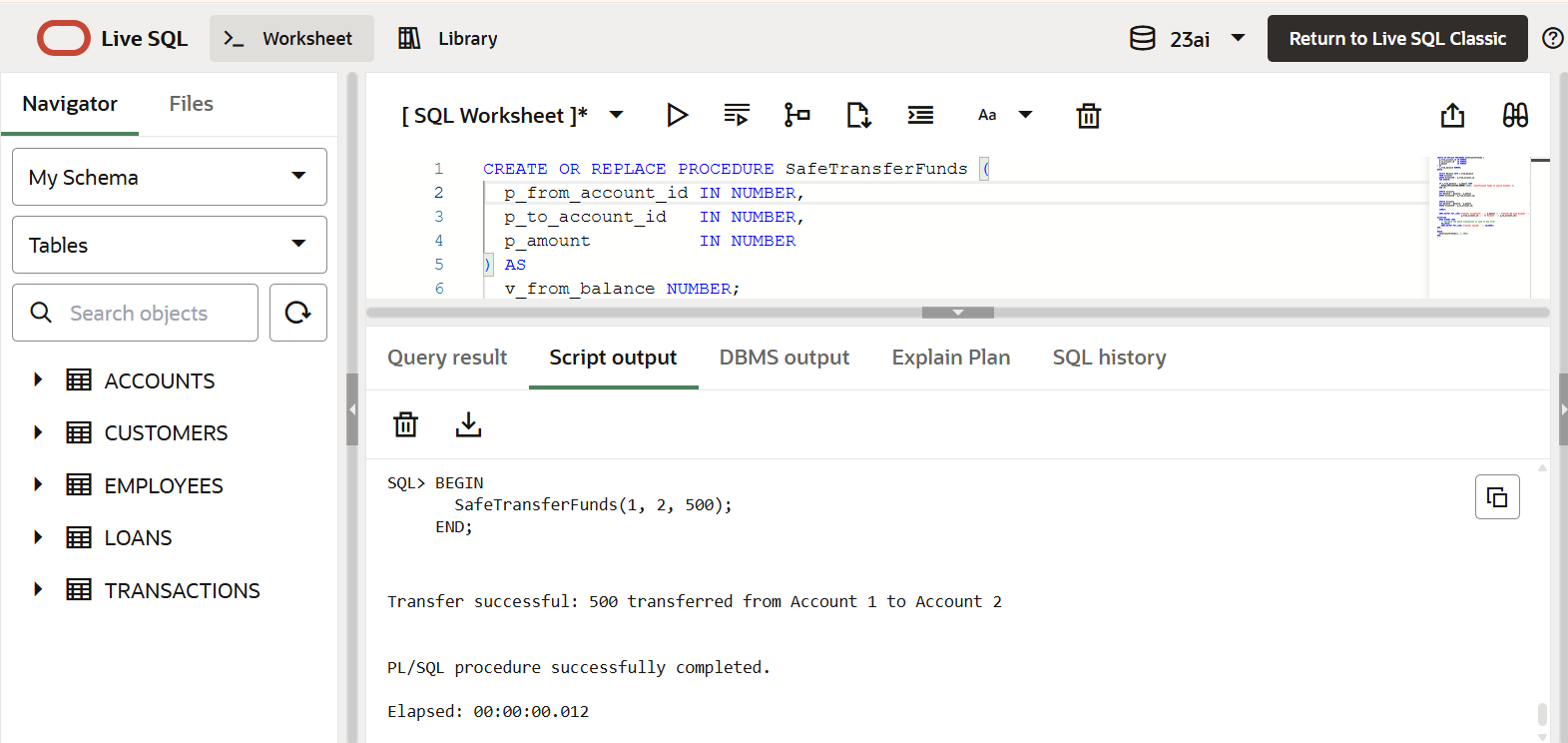
/

BEGIN

  SafeTransferFunds(1, 2, 500);

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\_incr IN NUMBER

) AS

  v\_current\_salary NUMBER;

BEGIN

  SELECT Salary INTO v\_current\_salary

  FROM Employees

  WHERE EmployeeID = p\_employee\_id

  FOR UPDATE;

  UPDATE Employees

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

  WHERE EmployeeID = p\_employee\_id;

  COMMIT;

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

EXCEPTION

  WHEN NO\_DATA\_FOUND THEN

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

  WHEN OTHERS THEN

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

    ROLLBACK;

END;

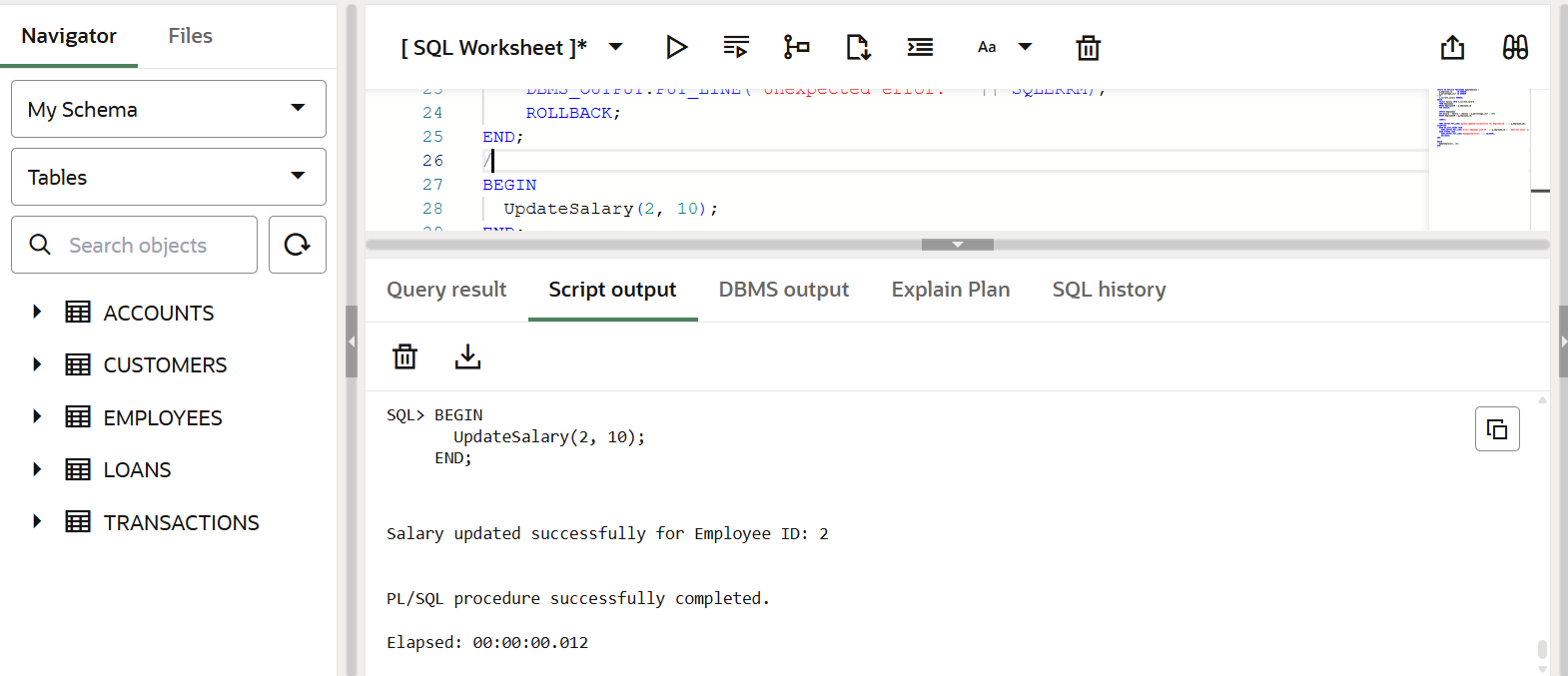
/

BEGIN

  UpdateSalary(2, 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: ' || p\_customer\_id || ' - ' || p\_name);

EXCEPTION

  WHEN DUP\_VAL\_ON\_INDEX THEN

    DBMS\_OUTPUT.PUT\_LINE('Error: Customer with ID ' || p\_customer\_id || ' already exists.');

  WHEN OTHERS THEN

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

    ROLLBACK;

END;

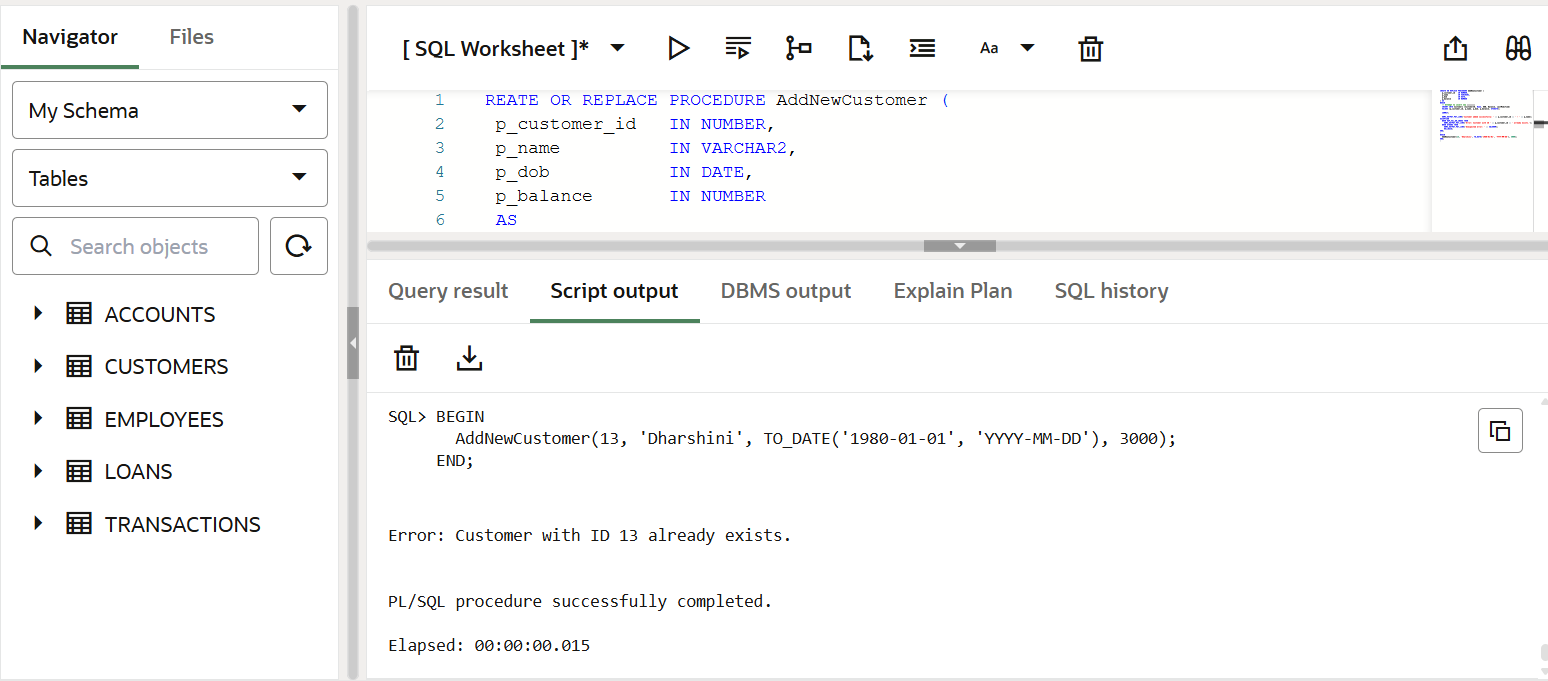
/

BEGIN

  AddNewCustomer(13, 'Dharshini', TO\_DATE('1980-01-01', 'YYYY-MM-DD'), 3000);

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

  FOR acc IN (

    SELECT AccountID, Balance

    FROM Accounts

    WHERE AccountType = 'Savings'

    FOR UPDATE

  ) LOOP

    UPDATE Accounts

    SET Balance = Balance + (acc.Balance \* 0.01)

    WHERE AccountID = acc.AccountID;

    DBMS\_OUTPUT.PUT\_LINE('Interest applied to Account ID: ' || acc.AccountID ||

                         ' | New Balance approx: ' || TO\_CHAR(acc.Balance \* 1.01, '999999.99'));

  END LOOP;

  COMMIT;

END;

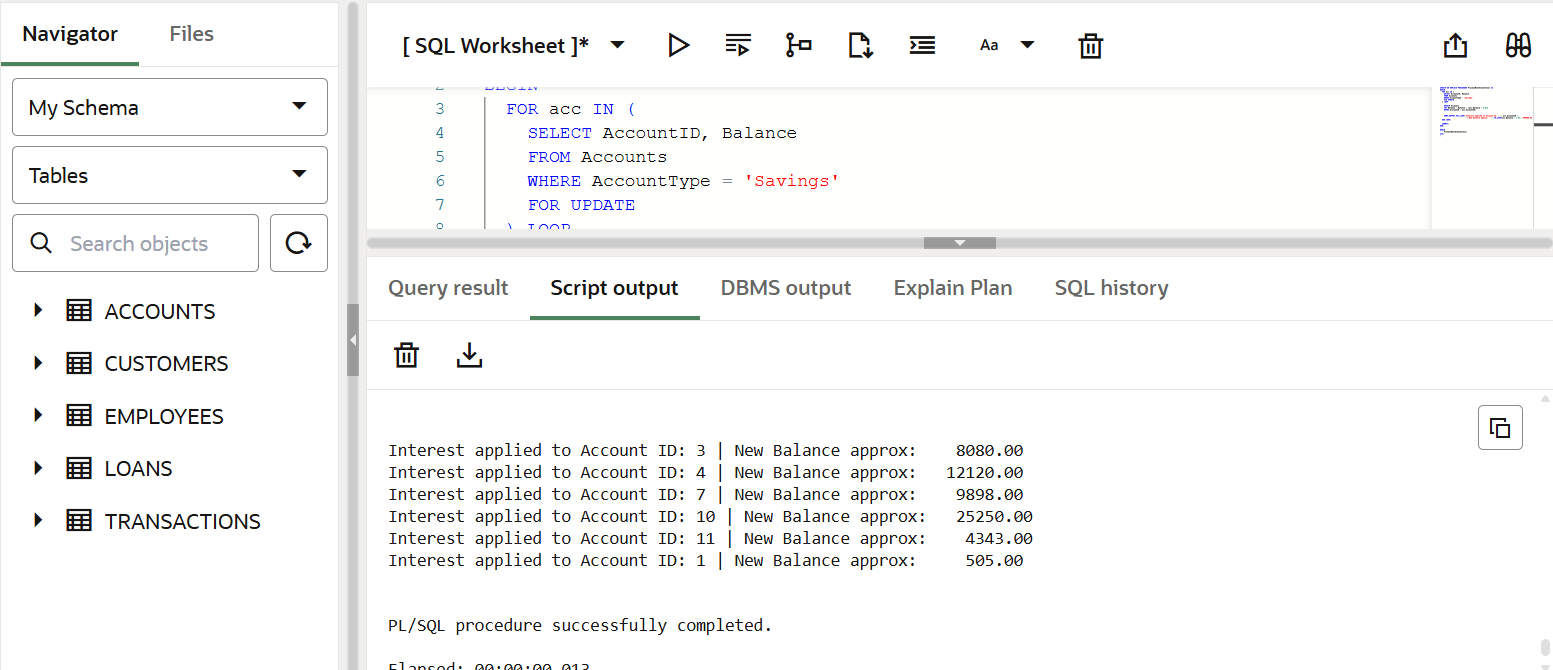
/

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\_percent  IN NUMBER

) AS

BEGIN

  FOR emp IN (

    SELECT EmployeeID, Name, Salary

    FROM Employees

    WHERE Department = p\_department

    FOR UPDATE

  ) LOOP

    UPDATE Employees

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

    WHERE EmployeeID = emp.EmployeeID;

    DBMS\_OUTPUT.PUT\_LINE('Bonus applied to ' || emp.Name ||

                         ' | New Salary approx: ' || TO\_CHAR(emp.Salary \* (1 + p\_bonus\_percent / 100), '999999.99'));

  END LOOP;

  COMMIT;

END;

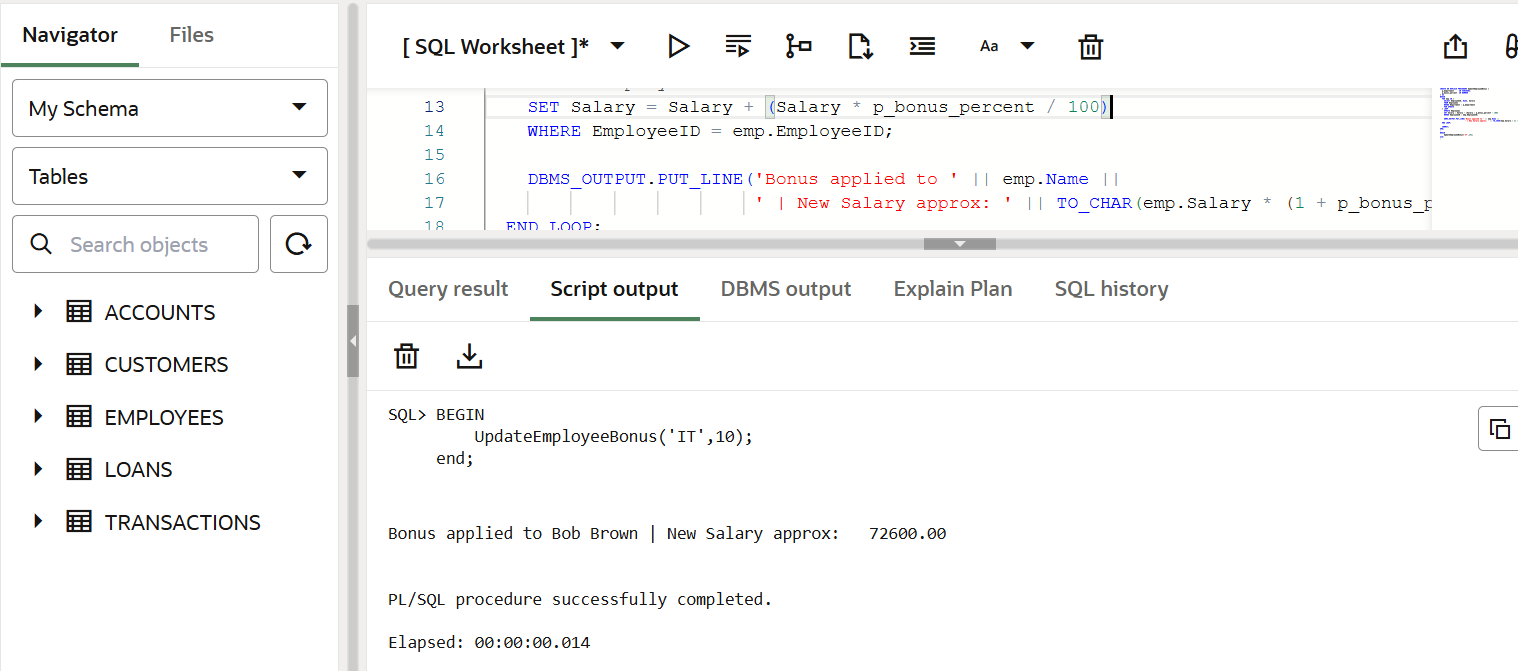
/

BEGIN

    UpdateEmployeeBonus('IT',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\_from\_balance NUMBER;

BEGIN

  SELECT Balance INTO v\_from\_balance

  FROM Accounts

  WHERE AccountID = p\_from\_account\_id

  FOR UPDATE;

  IF v\_from\_balance < p\_amount THEN

    DBMS\_OUTPUT.PUT\_LINE('Transfer failed: Insufficient funds in source account.');

    RETURN;

  END IF;

  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;

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

                       ' transferred from Account ' || p\_from\_account\_id ||

                       ' to Account ' || p\_to\_account\_id);

EXCEPTION

  WHEN NO\_DATA\_FOUND THEN

    DBMS\_OUTPUT.PUT\_LINE('Transfer failed: One of the accounts does not exist.');

  WHEN OTHERS THEN

    ROLLBACK;

    DBMS\_OUTPUT.PUT\_LINE('Transfer failed: ' || SQLERRM);

END;

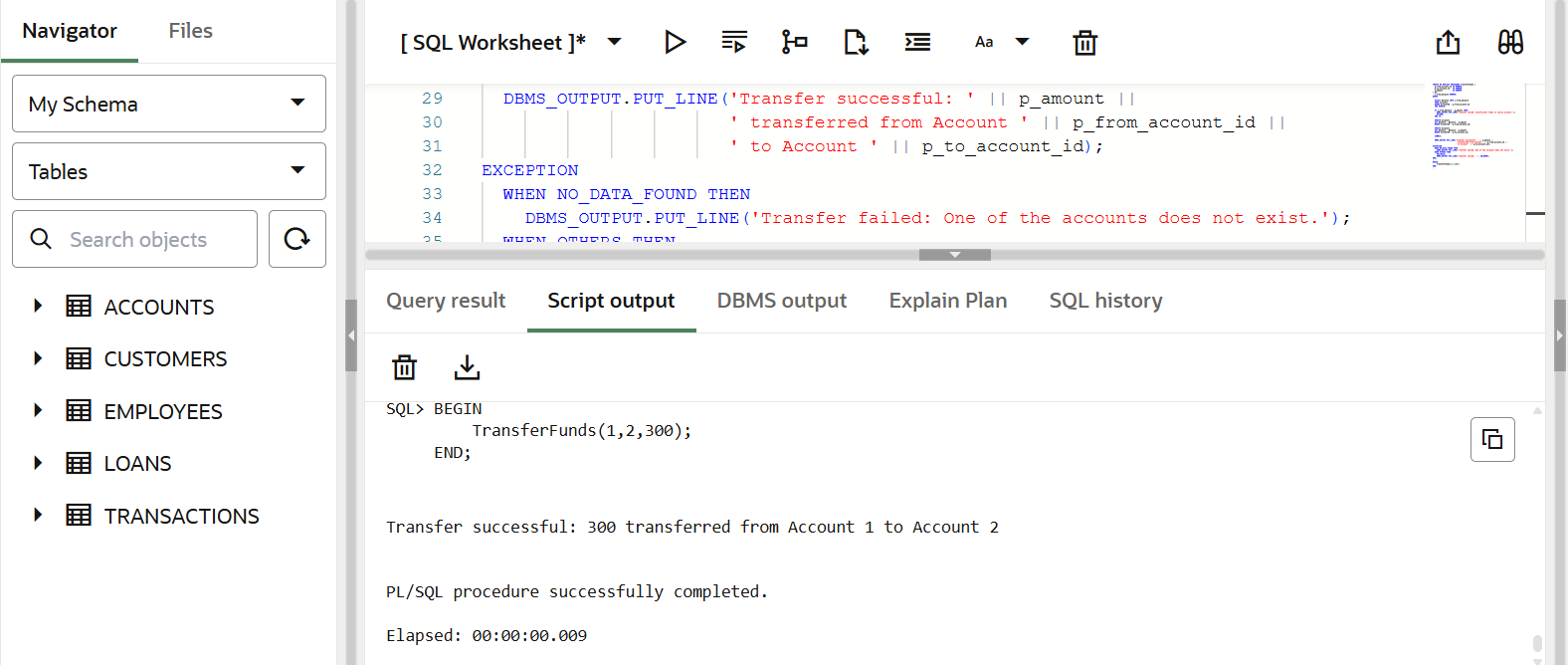
/

BEGIN

    TransferFunds(1,2,300);

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

  RETURN v\_age;

END;

/

DECLARE

  v\_age NUMBER;

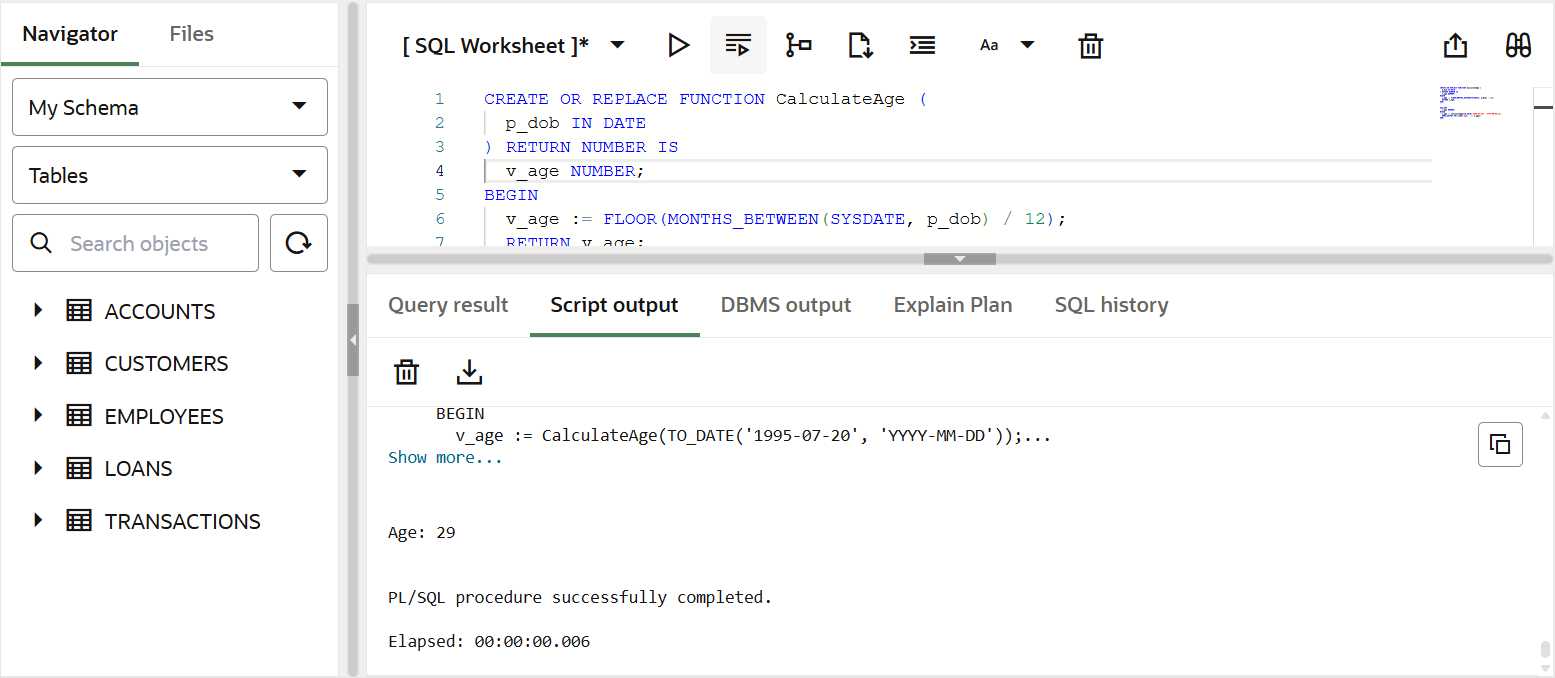
BEGIN

  v\_age := CalculateAge(TO\_DATE('1995-07-20', 'YYYY-MM-DD'));

  DBMS\_OUTPUT.PUT\_LINE('Age: ' || v\_age);

END;

/



**Scenario 2:** The bank needs to compute the monthly installment for a loan.

* + **Question:** Write a function **CalculateMonthlyInstallment** that takes the loan amount, interest rate, and loan duration in years as input and returns the monthly installment amount.

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;

END;

/

DECLARE

  v\_age NUMBER;

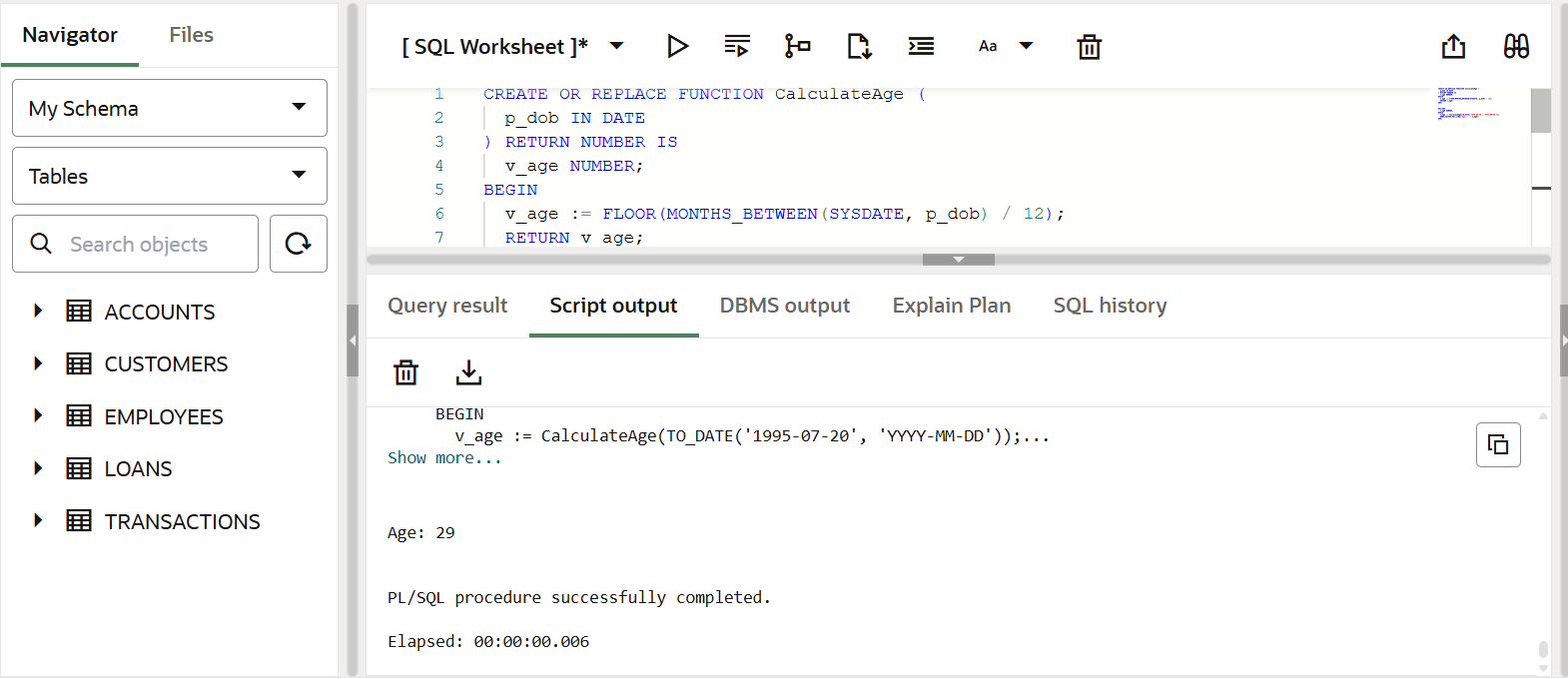
BEGIN

  v\_age := CalculateAge(TO\_DATE('1995-07-20', 'YYYY-MM-DD'));

  DBMS\_OUTPUT.PUT\_LINE('Age: ' || v\_age);

END;

/



**Scenario 3:** Check if a customer has sufficient balance before making a transaction.

* + **Question:** Write a function **HasSufficientBalance** that takes an account ID and an amount as input and returns a boolean indicating whether the account has at least the specified amount.

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');

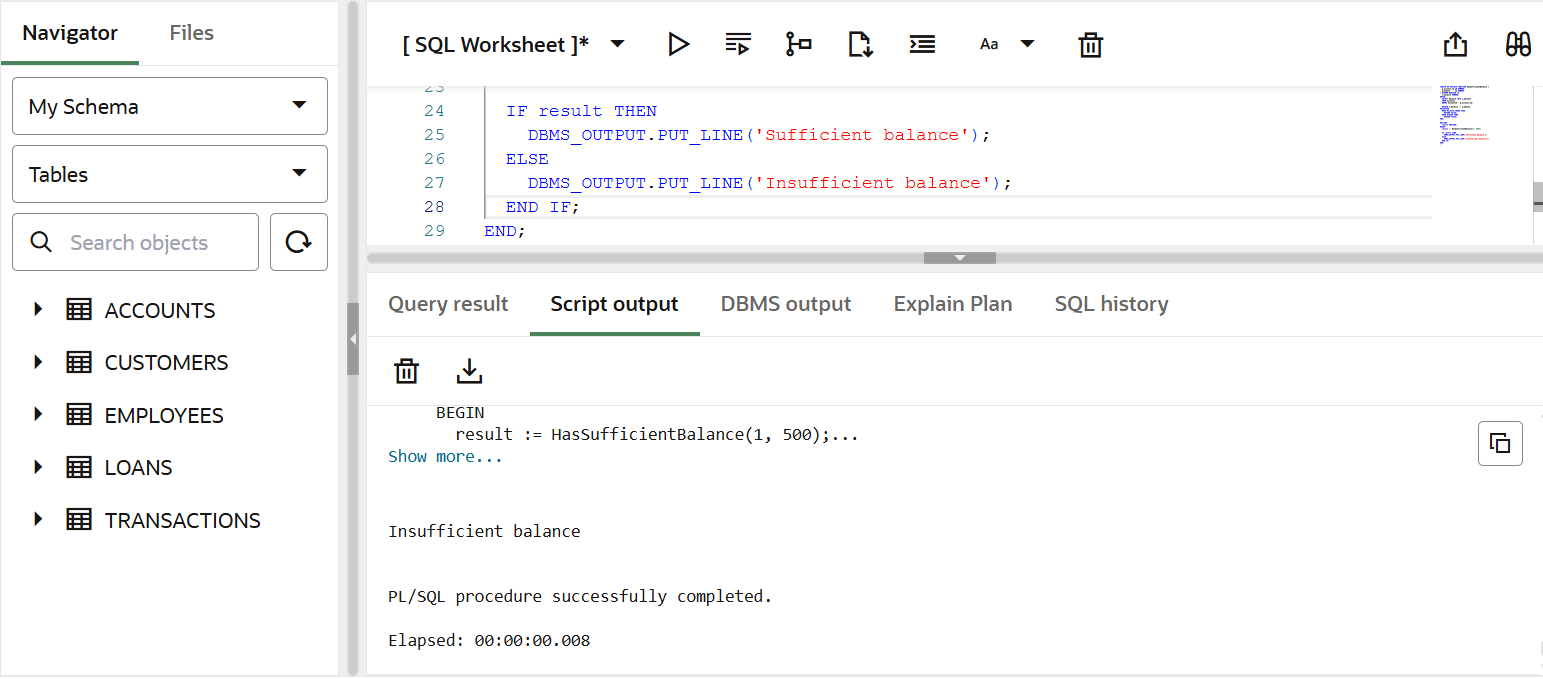
  ELSE

    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;

/

UPDATE Customers

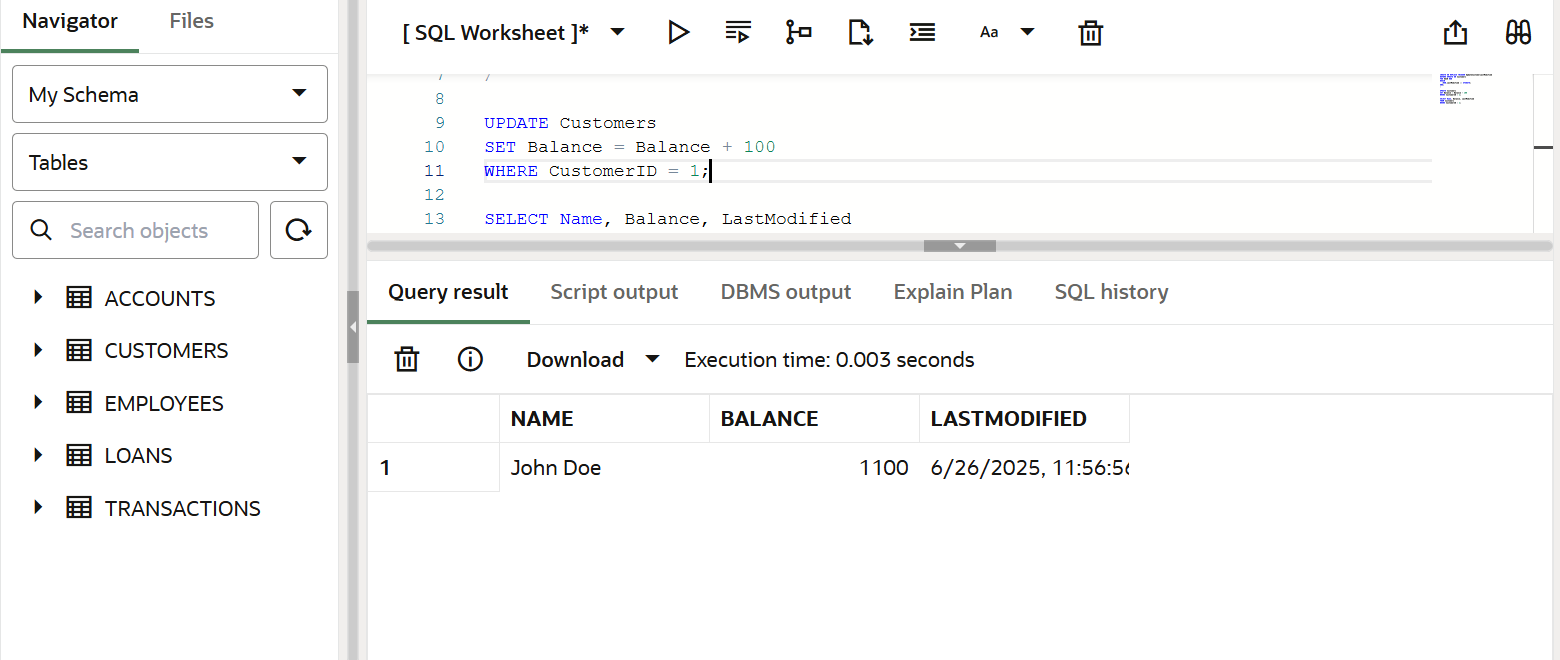
SET Balance = Balance + 100

WHERE CustomerID = 1;

SELECT Name, Balance, LastModified

FROM Customers

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 BY DEFAULT AS IDENTITY PRIMARY KEY,

TransactionID NUMBER,

AccountID NUMBER,

Amount NUMBER,

TransactionType VARCHAR2(10),

LoggedAt DATE

);

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog (

TransactionID,

AccountID,

Amount,

TransactionType,

LoggedAt

) VALUES (

:NEW.TransactionID,

:NEW.AccountID,

:NEW.Amount,

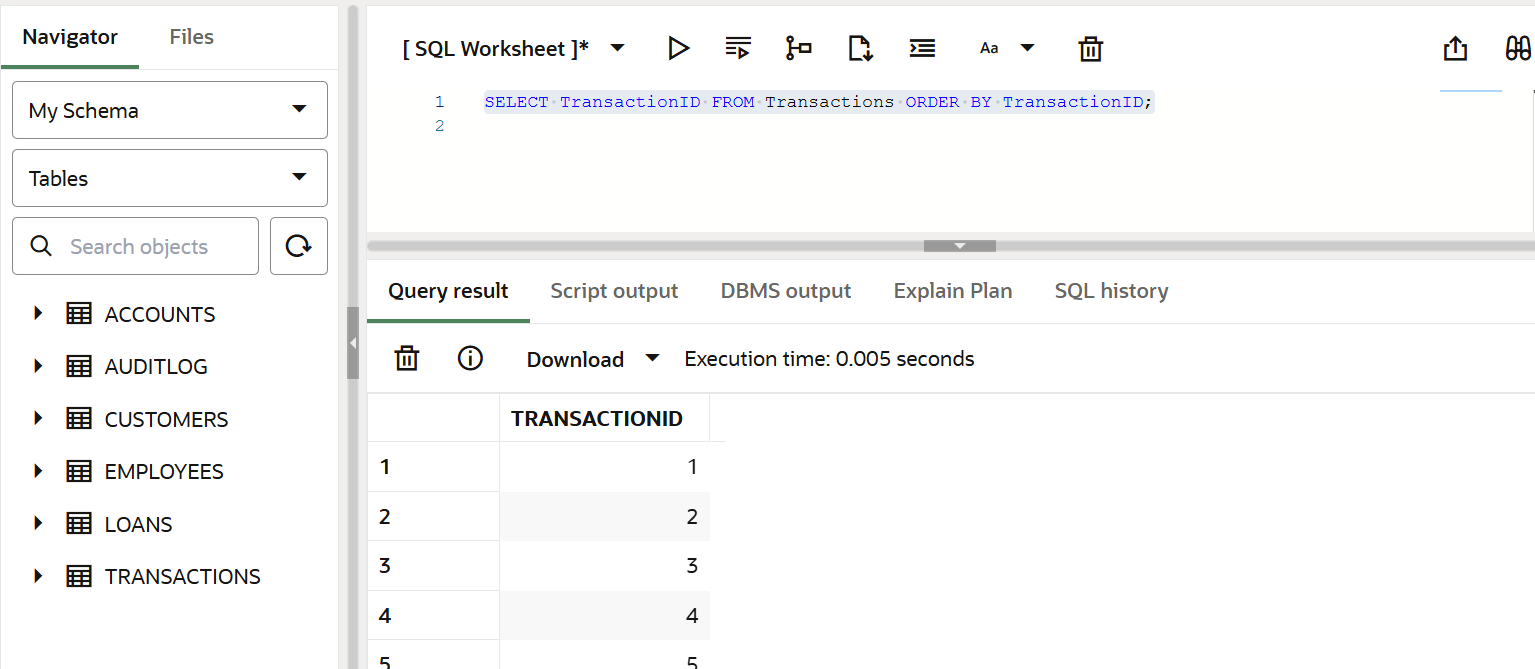
:NEW.TransactionType,

SYSDATE

);

END;

/



**Scenario 3:** Enforce business rules on deposits and withdrawals.

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

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

DECLARE

  v\_balance NUMBER;

BEGIN

  IF :NEW.TransactionType = 'Deposit' AND :NEW.Amount <= 0 THEN

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

  END IF;

   IF :NEW.TransactionType = 'Withdrawal' THEN

    SELECT Balance INTO v\_balance

    FROM Accounts

    WHERE AccountID = :NEW.AccountID;

    IF :NEW.Amount > v\_balance THEN

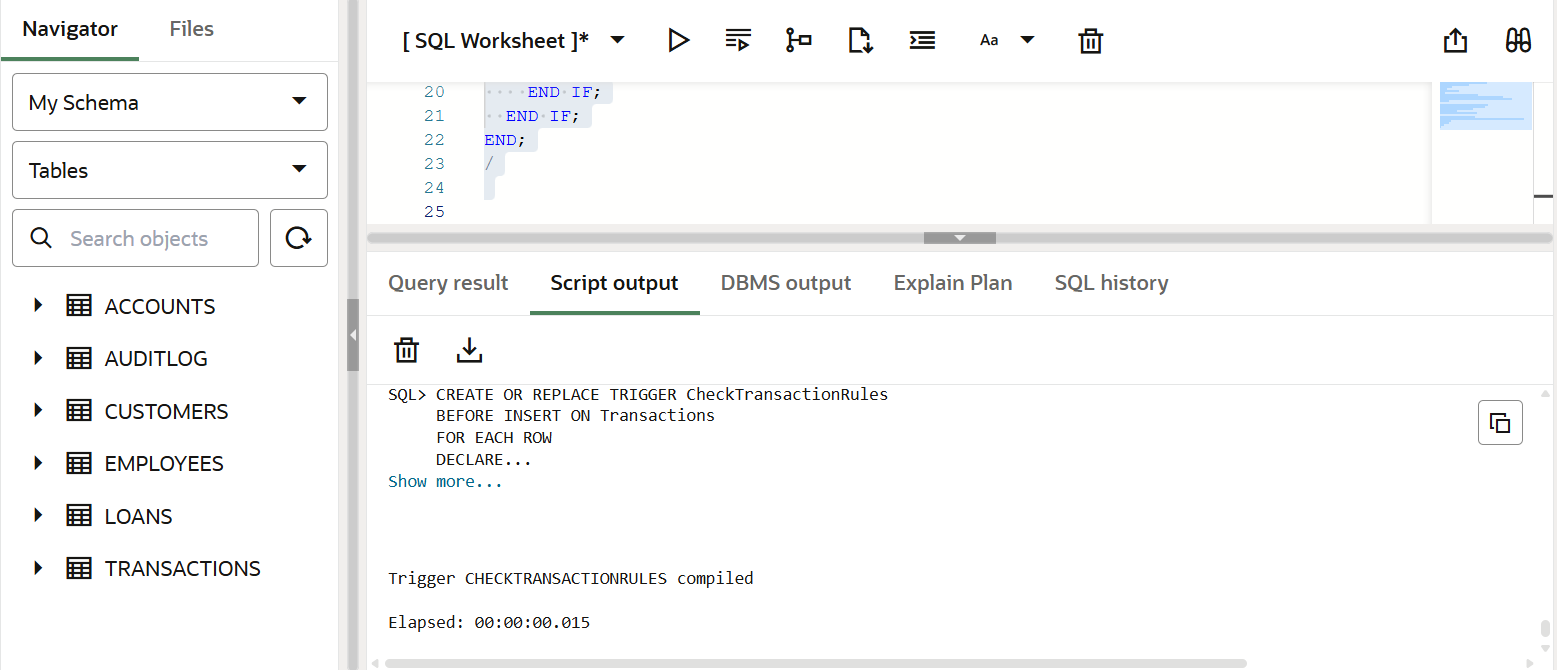
      RAISE\_APPLICATION\_ERROR(-20002, 'Withdrawal amount exceeds account balance.');

    END IF;

  END IF;

END;

/



**Exercise 6: Cursors**

**Scenario 1:** Generate monthly statements for all customers.

* + **Question:** Write a PL/SQL block using an explicit cursor **GenerateMonthlyStatements** that retrieves all transactions for the current month and prints a statement for each customer.

DECLARE

  CURSOR trans\_cursor IS

    SELECT c.CustomerID, c.Name, t.TransactionID, 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 TO\_CHAR(t.TransactionDate, 'MMYYYY') = TO\_CHAR(SYSDATE, 'MMYYYY')

    ORDER BY c.CustomerID, t.TransactionDate;

  v\_cust\_id     Customers.CustomerID%TYPE;

  v\_name        Customers.Name%TYPE;

  v\_trans\_id    Transactions.TransactionID%TYPE;

  v\_trans\_date  Transactions.TransactionDate%TYPE;

  v\_amount      Transactions.Amount%TYPE;

  v\_type        Transactions.TransactionType%TYPE;

  v\_last\_cust   NUMBER := -1;

BEGIN

  DBMS\_OUTPUT.PUT\_LINE('--- Monthly Statements for ' || TO\_CHAR(SYSDATE, 'Month YYYY') || ' ---');

  OPEN trans\_cursor;

  LOOP

    FETCH trans\_cursor INTO v\_cust\_id, v\_name, v\_trans\_id, v\_trans\_date, v\_amount, v\_type;

    EXIT WHEN trans\_cursor%NOTFOUND;

    IF v\_cust\_id != v\_last\_cust THEN

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

      DBMS\_OUTPUT.PUT\_LINE('Date       | Type       | Amount');

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

      v\_last\_cust := v\_cust\_id;

    END IF;

    DBMS\_OUTPUT.PUT\_LINE(

      TO\_CHAR(v\_trans\_date, 'DD-Mon-YYYY') || ' | ' ||

      RPAD(v\_type, 10) || ' | ' ||

      TO\_CHAR(v\_amount, '99999.99')

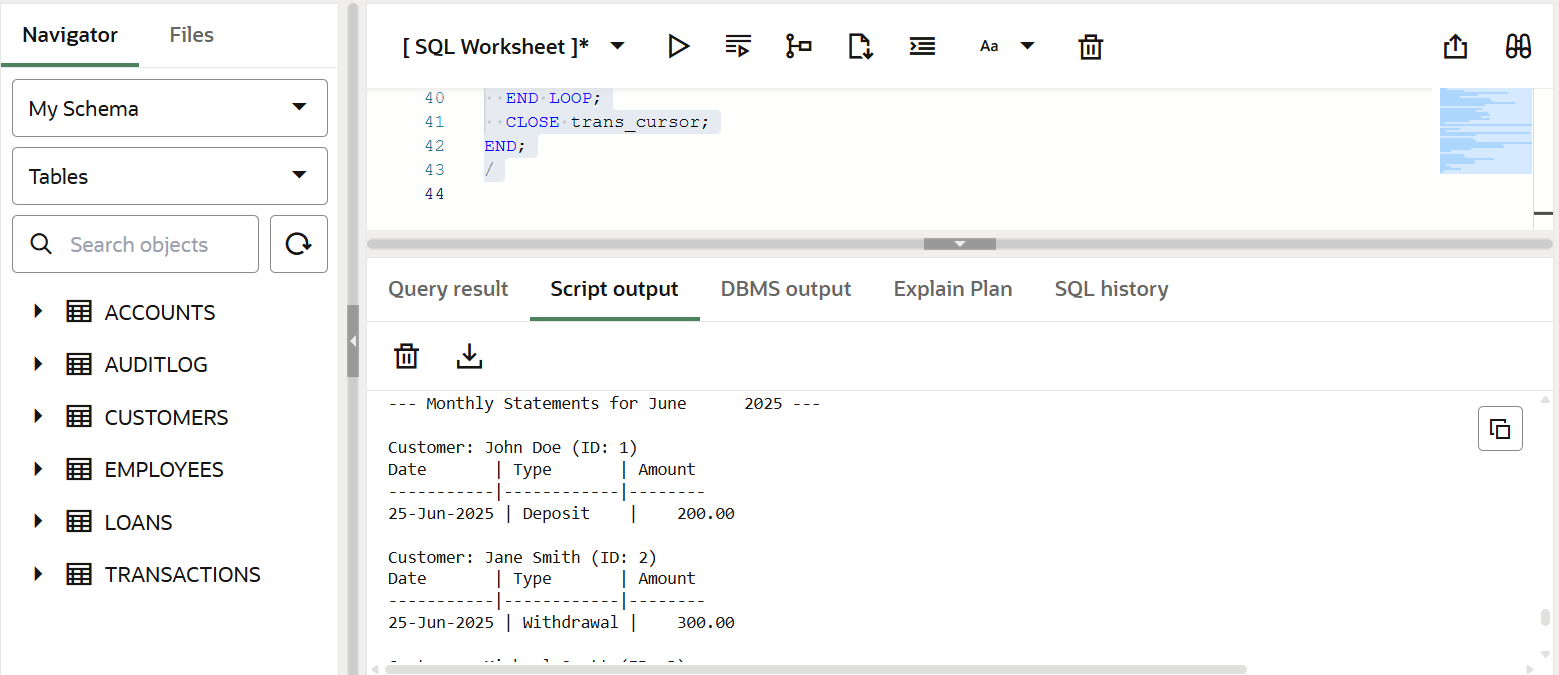
    );

  END LOOP;

  CLOSE trans\_cursor;

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

    FOR UPDATE;

  v\_account\_id Accounts.AccountID%TYPE;

  v\_balance    Accounts.Balance%TYPE;

  v\_fee CONSTANT NUMBER := 100; -- Annual fee amount

BEGIN

  OPEN acc\_cursor;

  LOOP

    FETCH acc\_cursor INTO v\_account\_id, v\_balance;

    EXIT WHEN acc\_cursor%NOTFOUND;

    -- Only deduct if balance is enough

    IF v\_balance >= v\_fee THEN

      UPDATE Accounts

      SET Balance = Balance - v\_fee

      WHERE AccountID = v\_account\_id;

      DBMS\_OUTPUT.PUT\_LINE('Annual fee of ₹' || v\_fee || ' deducted from Account ID: ' || v\_account\_id ||

                           ' | New Balance: ₹' || TO\_CHAR(v\_balance - v\_fee, '99999.99'));

    ELSE

      DBMS\_OUTPUT.PUT\_LINE('Skipping Account ID: ' || v\_account\_id || ' due to insufficient balance.');

    END IF;

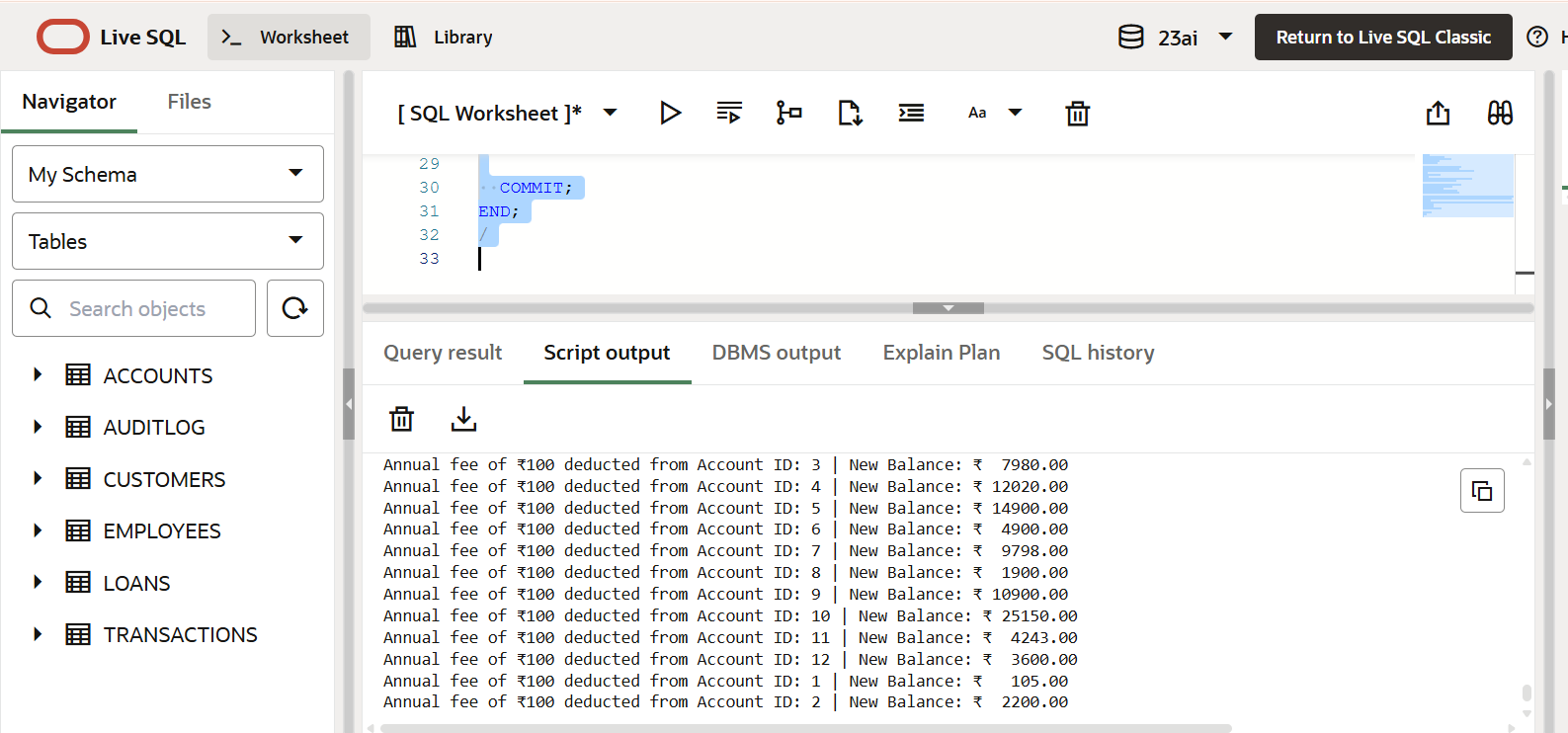
  END LOOP;

  CLOSE acc\_cursor;

  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 loan\_cursor IS

    SELECT LoanID, LoanAmount, InterestRate, StartDate

    FROM Loans

    FOR UPDATE;

  v\_loan\_id      Loans.LoanID%TYPE;

  v\_loan\_amt     Loans.LoanAmount%TYPE;

  v\_rate         Loans.InterestRate%TYPE;

  v\_start\_date   Loans.StartDate%TYPE;

  v\_new\_rate     NUMBER;

BEGIN

  OPEN loan\_cursor;

  LOOP

    FETCH loan\_cursor INTO v\_loan\_id, v\_loan\_amt, v\_rate, v\_start\_date;

    EXIT WHEN loan\_cursor%NOTFOUND;

    v\_new\_rate := v\_rate;

    -- Apply new policy

    IF v\_loan\_amt > 50000 THEN

      v\_new\_rate := v\_new\_rate - 1;

    END IF;

    IF MONTHS\_BETWEEN(SYSDATE, v\_start\_date) > 36 THEN

      v\_new\_rate := v\_new\_rate - 0.5;

    END IF;

    -- Update only if the rate changed

    IF v\_new\_rate != v\_rate THEN

      UPDATE Loans

      SET InterestRate = v\_new\_rate

      WHERE LoanID = v\_loan\_id;

      DBMS\_OUTPUT.PUT\_LINE('Loan ID ' || v\_loan\_id || ' updated | Old Rate: ' || v\_rate ||

                           ' -> New Rate: ' || v\_new\_rate);

    ELSE

      DBMS\_OUTPUT.PUT\_LINE('Loan ID ' || v\_loan\_id || ' has no change in interest rate.');

    END IF;

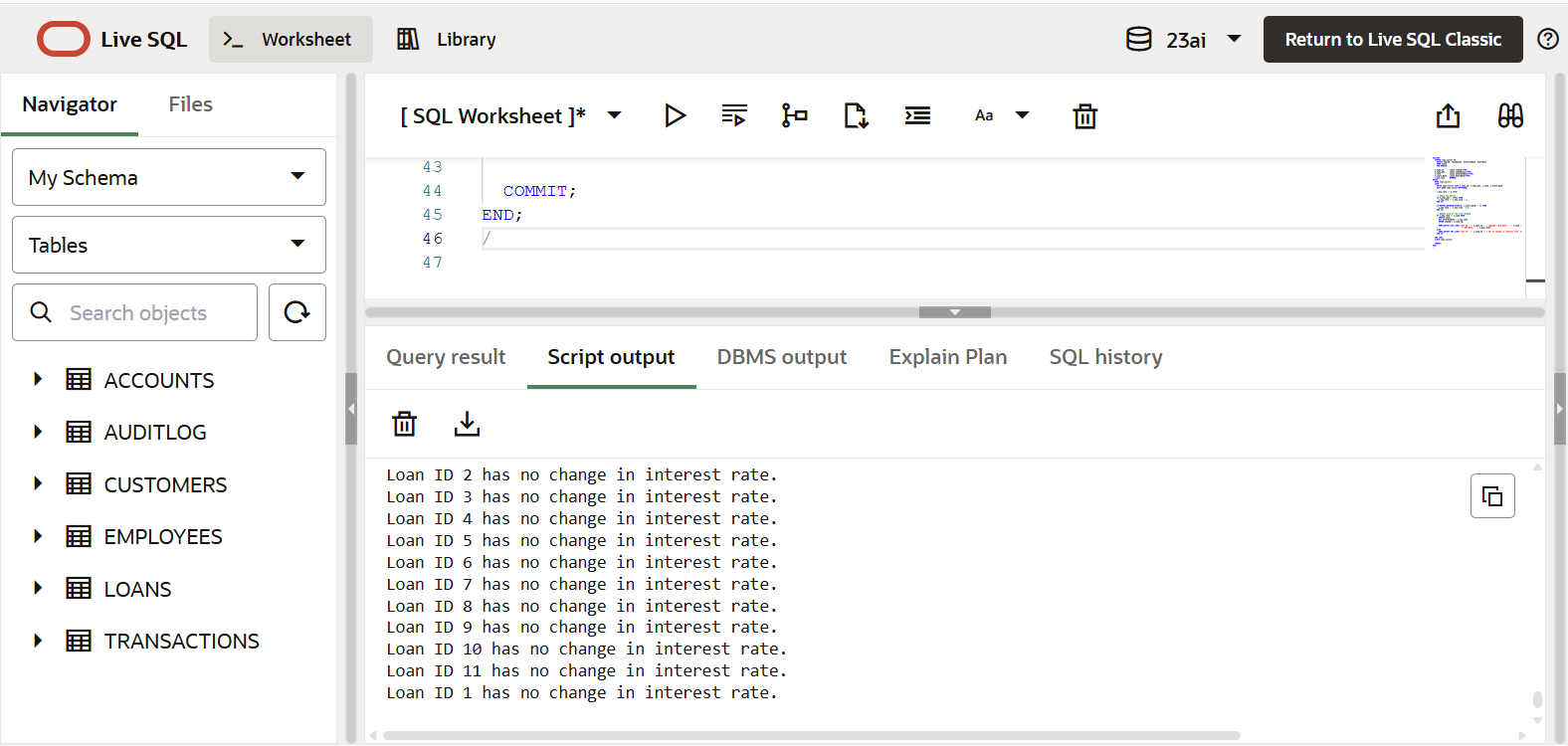
  END LOOP;

  CLOSE loan\_cursor;

  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 AddCustomer(

    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\_name        IN VARCHAR2,

    p\_dob         IN DATE

  );

  FUNCTION GetCustomerBalance(

    p\_customer\_id IN NUMBER

  ) RETURN NUMBER;

END CustomerManagement;

/

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

  PROCEDURE AddCustomer(

    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);

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

  EXCEPTION

    WHEN DUP\_VAL\_ON\_INDEX THEN

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

    WHEN OTHERS THEN

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

  END;

  PROCEDURE UpdateCustomerDetails(

    p\_customer\_id 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\_customer\_id;

    IF SQL%ROWCOUNT = 0 THEN

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

    ELSE

      DBMS\_OUTPUT.PUT\_LINE('Customer details updated.');

    END IF;

  END;

  FUNCTION GetCustomerBalance(

    p\_customer\_id IN NUMBER

  ) RETURN NUMBER IS

    v\_balance NUMBER;

  BEGIN

    SELECT NVL(SUM(Balance), 0)

    INTO v\_balance

    FROM Accounts

    WHERE CustomerID = p\_customer\_id;

    RETURN v\_balance;

  EXCEPTION

    WHEN NO\_DATA\_FOUND THEN

      RETURN 0;

  END;

END CustomerManagement;

/

BEGIN

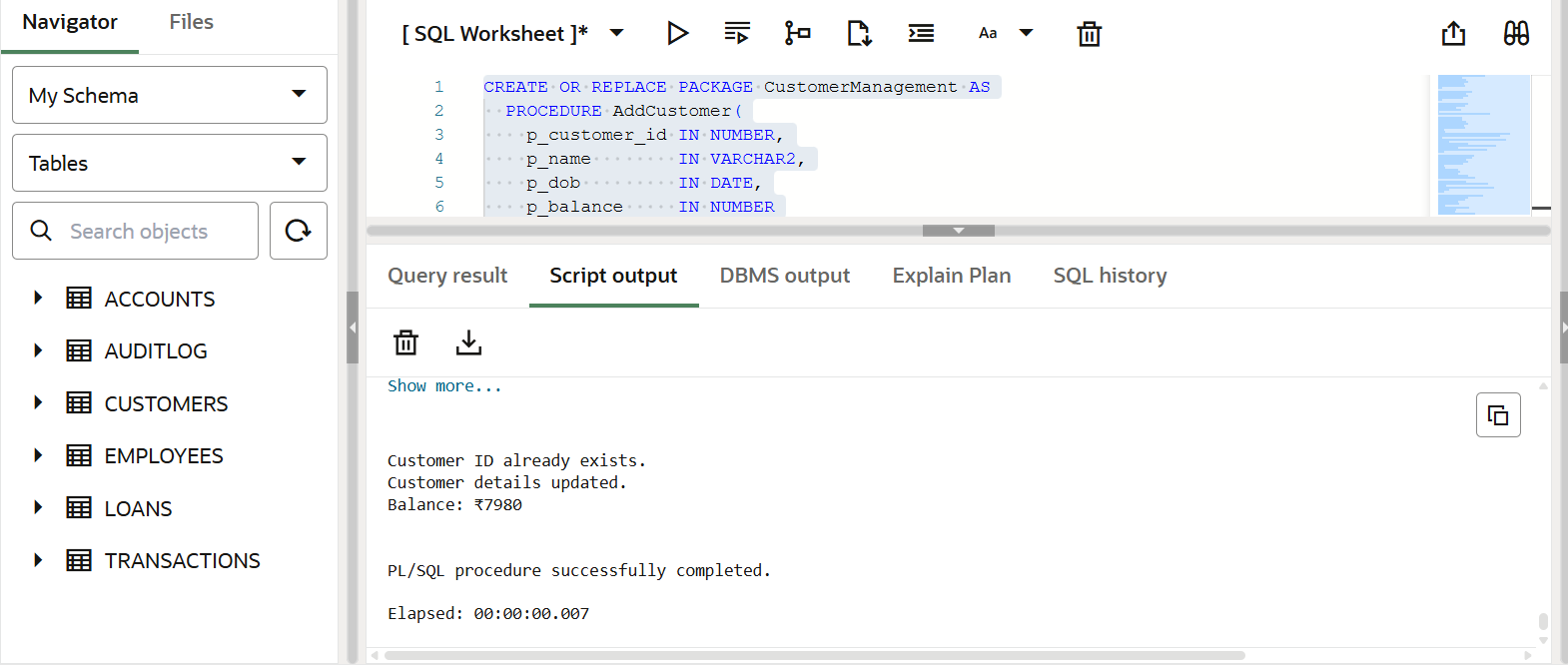
  CustomerManagement.AddCustomer(3, 'Kavi Kumar', TO\_DATE('1995-10-10','YYYY-MM-DD'), 2000);

  CustomerManagement.UpdateCustomerDetails(3, 'Kavi K.', TO\_DATE('1995-10-10','YYYY-MM-DD'));

  DBMS\_OUTPUT.PUT\_LINE('Balance: ₹' || CustomerManagement.GetCustomerBalance(3));

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\_department IN VARCHAR2,

    p\_hire\_date  IN DATE

  );

  PROCEDURE UpdateEmployeeDetails(

    p\_emp\_id     IN NUMBER,

    p\_name       IN VARCHAR2,

    p\_position   IN VARCHAR2,

    p\_department IN VARCHAR2

  );

  FUNCTION GetAnnualSalary(

    p\_emp\_id IN NUMBER

  ) RETURN NUMBER;

END EmployeeManagement;

/

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

  -- 1. Hire new employee

  PROCEDURE HireEmployee(

    p\_emp\_id     IN NUMBER,

    p\_name       IN VARCHAR2,

    p\_position   IN VARCHAR2,

    p\_salary     IN NUMBER,

    p\_department IN VARCHAR2,

    p\_hire\_date  IN DATE

  ) IS

  BEGIN

    INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate)

    VALUES (p\_emp\_id, p\_name, p\_position, p\_salary, p\_department, p\_hire\_date);

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

  EXCEPTION

    WHEN DUP\_VAL\_ON\_INDEX THEN

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

    WHEN OTHERS THEN

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

  END;

  -- 2. Update employee details

  PROCEDURE UpdateEmployeeDetails(

    p\_emp\_id     IN NUMBER,

    p\_name       IN VARCHAR2,

    p\_position   IN VARCHAR2,

    p\_department IN VARCHAR2

  ) IS

  BEGIN

    UPDATE Employees

    SET Name = p\_name,

        Position = p\_position,

        Department = p\_department

    WHERE EmployeeID = p\_emp\_id;

    IF SQL%ROWCOUNT = 0 THEN

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

    ELSE

      DBMS\_OUTPUT.PUT\_LINE('Employee details updated.');

    END IF;

  END;

  -- 3. Get annual salary

  FUNCTION GetAnnualSalary(

    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 0;

  END;

END EmployeeManagement;

/

BEGIN

  EmployeeManagement.HireEmployee(

    3, 'Sahana Raj', 'Analyst', 40000, 'Finance', TO\_DATE('2024-09-01', 'YYYY-MM-DD')

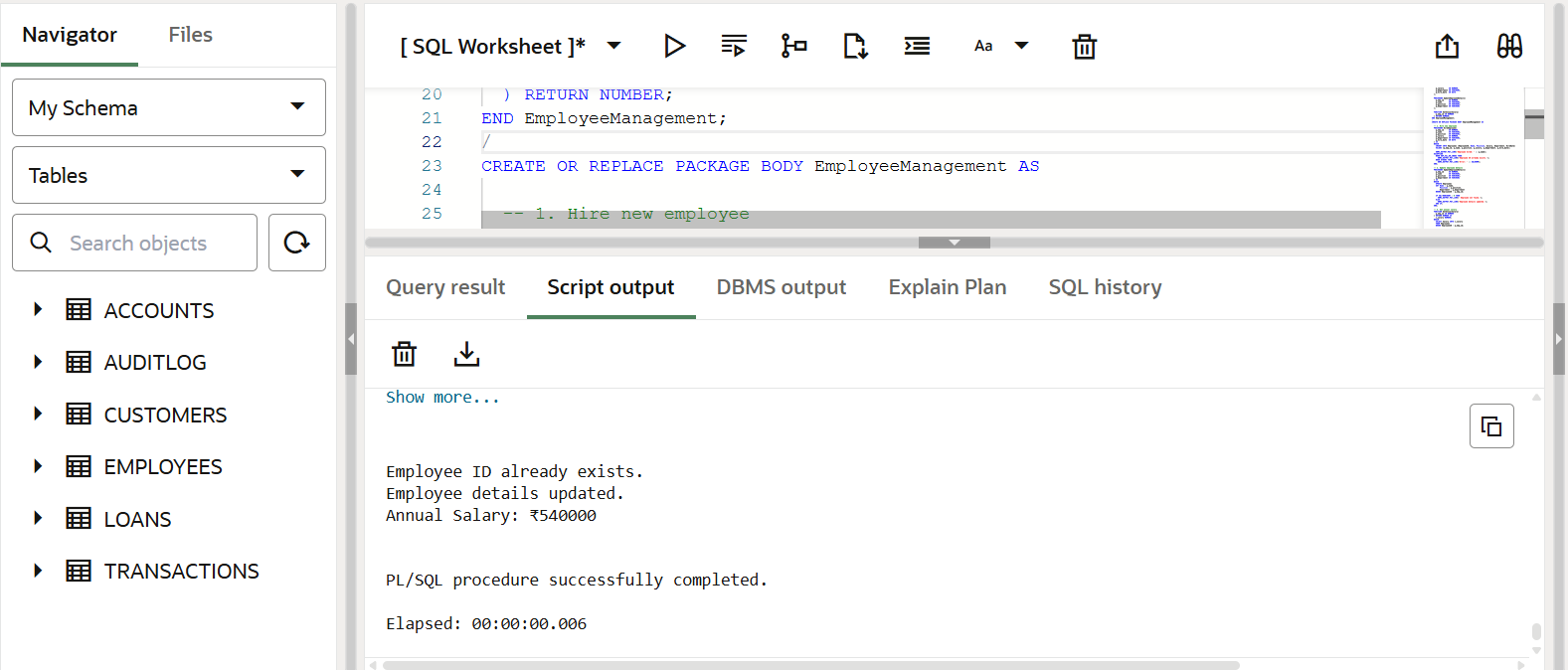
  );

  EmployeeManagement.UpdateEmployeeDetails(3, 'Sahana R.', 'Senior Analyst', 'Finance');

  DBMS\_OUTPUT.PUT\_LINE('Annual Salary: ₹' || EmployeeManagement.GetAnnualSalary(3));

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.

CREATE OR REPLACE PACKAGE AccountOperations AS

  PROCEDURE OpenAccount(

    p\_account\_id   IN NUMBER,

    p\_customer\_id  IN NUMBER,

    p\_account\_type IN VARCHAR2,

    p\_balance      IN NUMBER

  );

  PROCEDURE CloseAccount(

    p\_account\_id IN NUMBER

  );

  FUNCTION GetTotalBalance(

    p\_customer\_id IN NUMBER

  ) RETURN NUMBER;

END AccountOperations;

/

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

  PROCEDURE OpenAccount(

    p\_account\_id   IN NUMBER,

    p\_customer\_id  IN NUMBER,

    p\_account\_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\_account\_type, p\_balance, SYSDATE

    );

    DBMS\_OUTPUT.PUT\_LINE('Account opened successfully: ID ' || p\_account\_id);

  EXCEPTION

    WHEN DUP\_VAL\_ON\_INDEX THEN

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

    WHEN OTHERS THEN

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

  END;

  PROCEDURE CloseAccount(

    p\_account\_id IN NUMBER

  ) IS

  BEGIN

    DELETE FROM Accounts

    WHERE AccountID = p\_account\_id;

    IF SQL%ROWCOUNT = 0 THEN

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

    ELSE

      DBMS\_OUTPUT.PUT\_LINE('Account closed: ID ' || p\_account\_id);

    END IF;

  END;

  FUNCTION GetTotalBalance(

    p\_customer\_id IN NUMBER

  ) RETURN NUMBER IS

    v\_total\_balance NUMBER;

  BEGIN

    SELECT NVL(SUM(Balance), 0)

    INTO v\_total\_balance

    FROM Accounts

    WHERE CustomerID = p\_customer\_id;

    RETURN v\_total\_balance;

  EXCEPTION

    WHEN NO\_DATA\_FOUND THEN

      RETURN 0;

  END;

END AccountOperations;

/

BEGIN

  AccountOperations.OpenAccount(3, 1, 'Savings', 2500);

  AccountOperations.CloseAccount(2);

  DBMS\_OUTPUT.PUT\_LINE('Total balance for Customer 1: ₹' ||

                       AccountOperations.GetTotalBalance(1));

END;

/

