**WEEK-2**

**PL SQL HANDS ON**

**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 IN (

    SELECT c.customerid, TRUNC(MONTHS\_BETWEEN(SYSDATE, c.dob)/12) AS age,

           l.loanid, l.interestrate

    FROM customers c

    JOIN loans l ON c.customerid = l.customerid

  ) LOOP

    IF cust.age > 60 THEN

      UPDATE loans

      SET interestrate = interestrate - 1

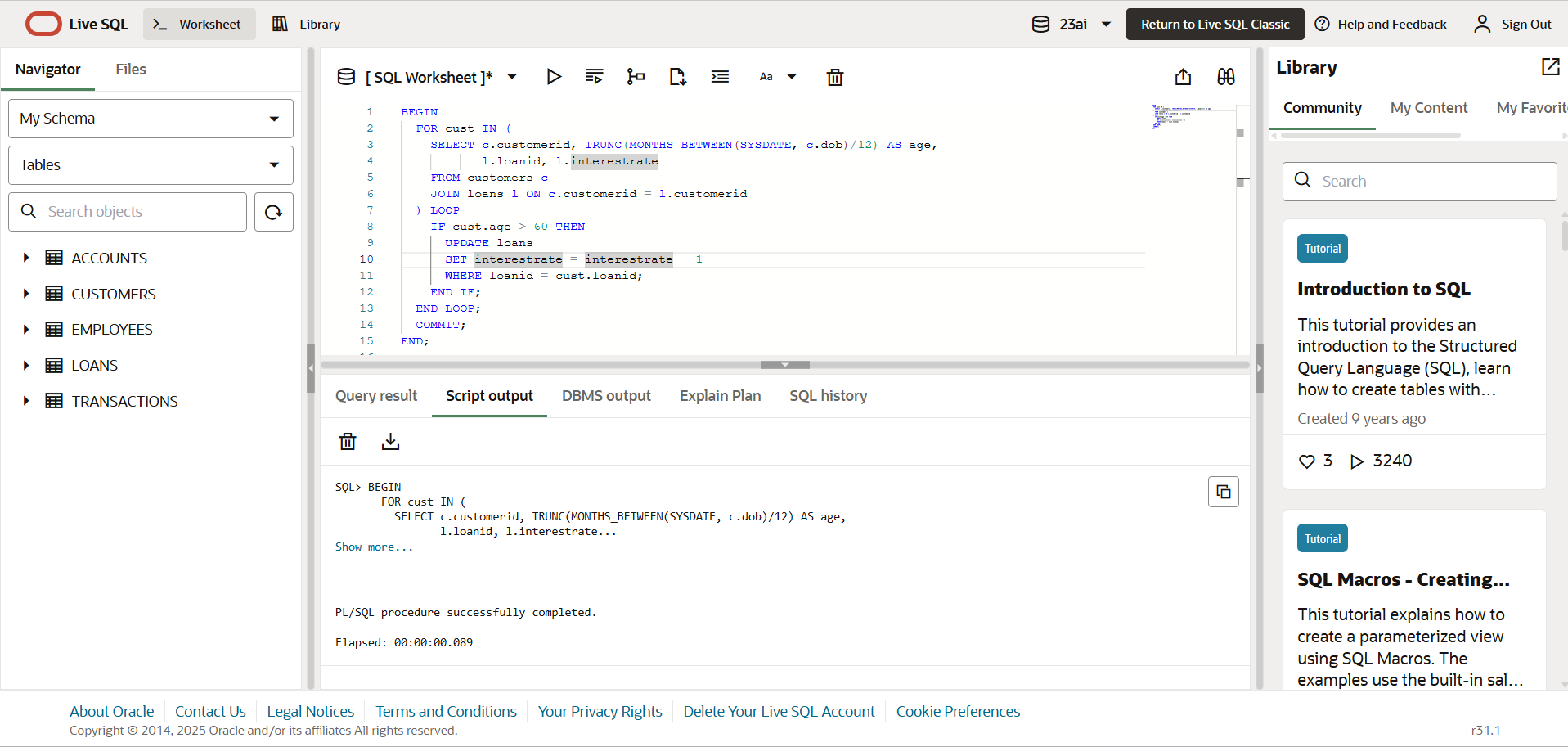
      WHERE loanid = cust.loanid;

    END IF;

  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 IN (

    SELECT customerid, balance

    FROM customers

  ) LOOP

    IF cust.balance > 10000 THEN

      UPDATE customers

      SET isvip = 'TRUE'

      WHERE customerid = cust.customerid;

    END IF;

  END LOOP;

  COMMIT;

END;

**OUTPUT:  
A screenshot of a computer

AI-generated content may be incorrect.**

**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:**DECLARE

  v\_due\_date DATE := SYSDATE + 30;

BEGIN

  FOR loan\_rec IN (

    SELECT l.loanid, c.name AS customer\_name, l.enddate AS due\_date

    FROM loans l

    JOIN customers c ON l.customerid = c.customerid

    WHERE l.enddate BETWEEN SYSDATE AND v\_due\_date

  ) LOOP

    DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan ' || loan\_rec.loanid ||

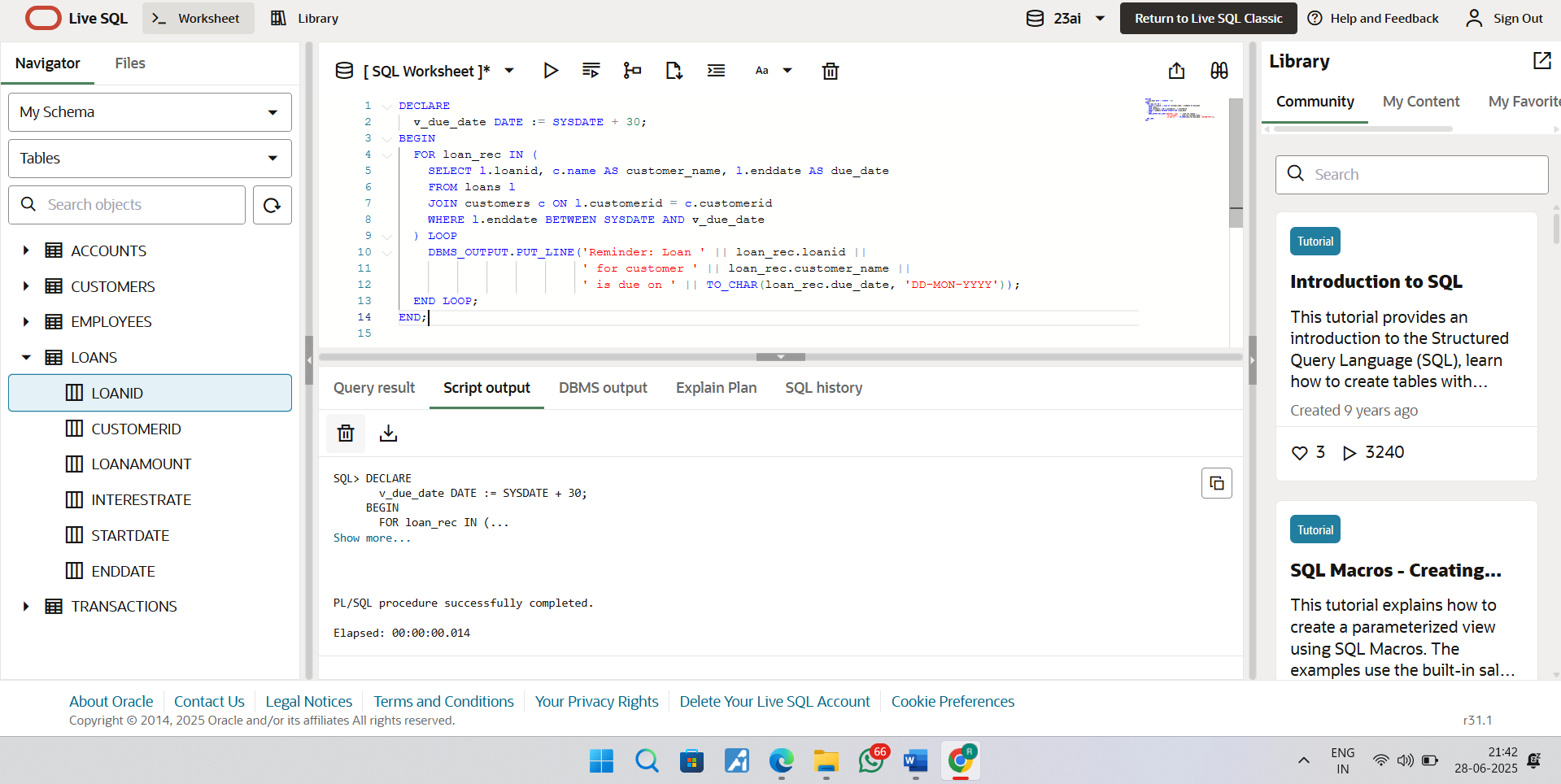
                         ' for customer ' || loan\_rec.customer\_name ||

                         ' is due on ' || TO\_CHAR(loan\_rec.due\_date, '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\_from\_acc\_id IN NUMBER,

  p\_to\_acc\_id IN NUMBER,

  p\_amount IN NUMBER

) AS

  v\_balance NUMBER;

BEGIN

  SELECT balance INTO v\_balance

  FROM accounts

  WHERE accountid = p\_from\_acc\_id;

  IF v\_balance < p\_amount THEN

    RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient funds');

  END IF;

  UPDATE accounts

  SET balance = balance - p\_amount

  WHERE accountid = p\_from\_acc\_id;

  UPDATE accounts

  SET balance = balance + p\_amount

  WHERE accountid = p\_to\_acc\_id;

  COMMIT;

  DBMS\_OUTPUT.PUT\_LINE('Transfer successful');

EXCEPTION

  WHEN OTHERS THEN

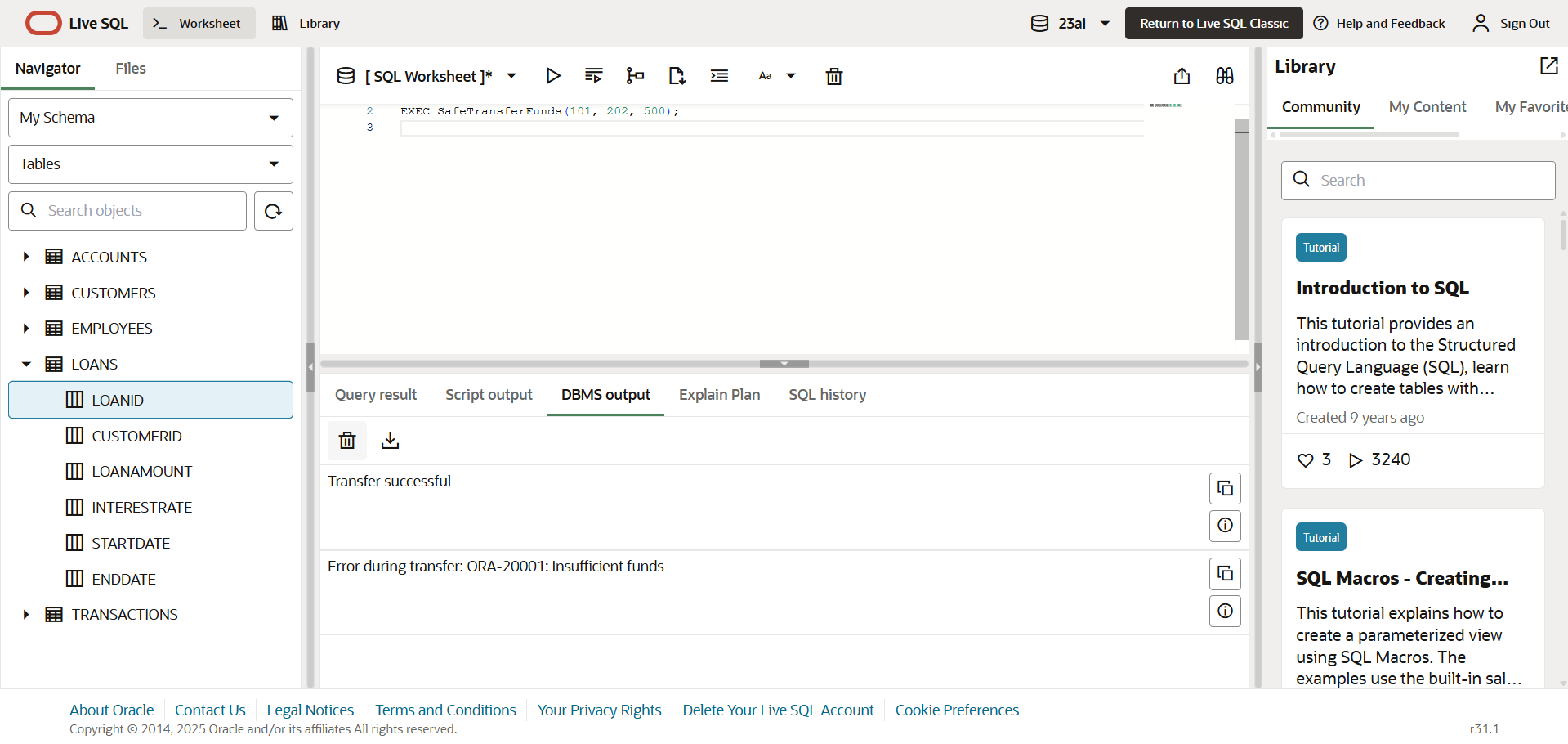
    ROLLBACK;

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

END;

EXEC SafeTransferFunds(101, 202, 500);

**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\_emp\_id IN NUMBER,

  p\_percent IN NUMBER

) AS

BEGIN

  UPDATE employees

  SET salary = salary + (salary \* p\_percent / 100)

  WHERE employeeid = p\_emp\_id;

  IF SQL%ROWCOUNT = 0 THEN

    RAISE\_APPLICATION\_ERROR(-20002, 'Employee ID not found');

  END IF;

  COMMIT;

  DBMS\_OUTPUT.PUT\_LINE('Salary updated successfully');

EXCEPTION

  WHEN OTHERS THEN

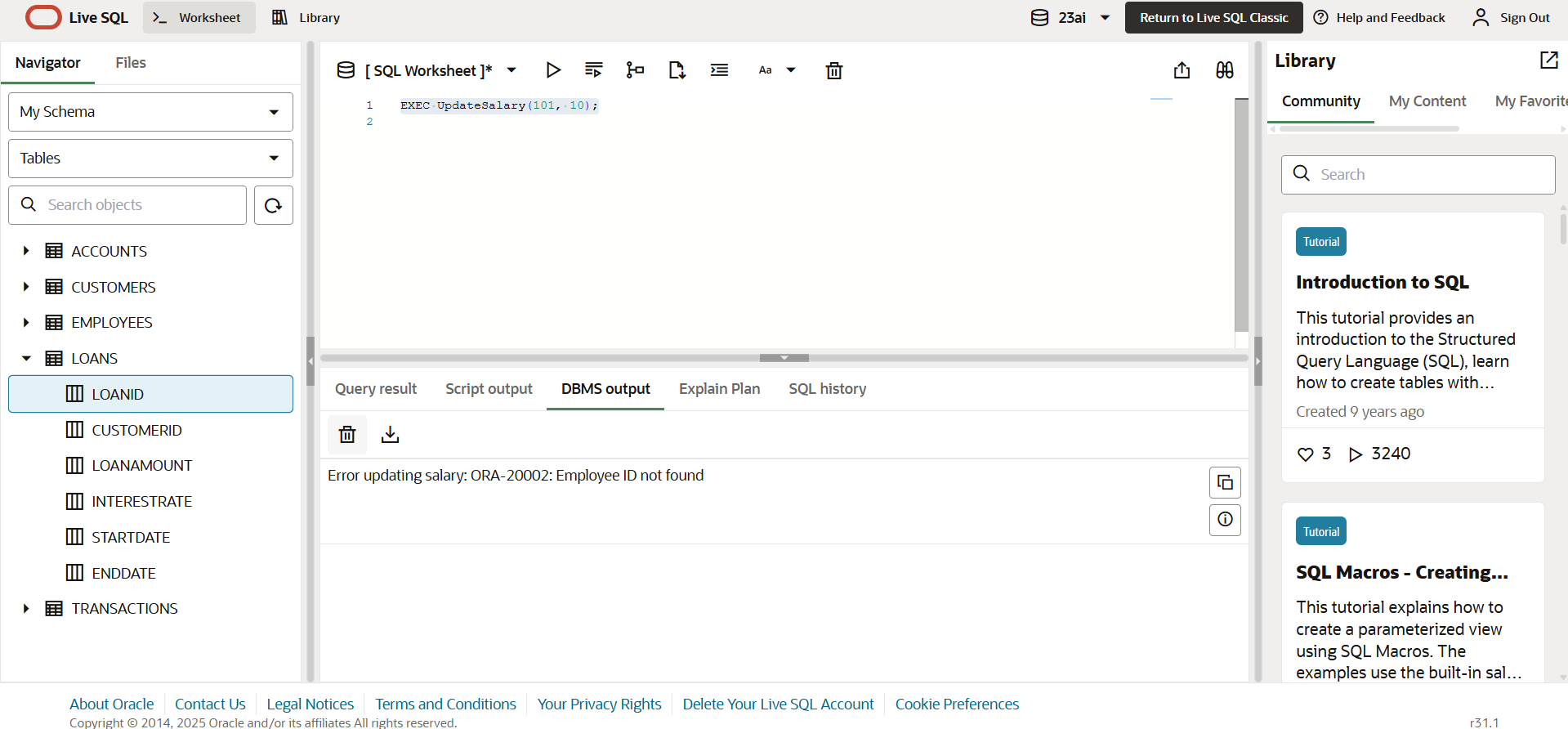
    ROLLBACK;

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

END;

EXEC UpdateSalary(101, 10);

**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\_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)

  VALUES (p\_customer\_id, p\_name, p\_dob, p\_balance);

  COMMIT;

  DBMS\_OUTPUT.PUT\_LINE('Customer added successfully');

EXCEPTION

  WHEN DUP\_VAL\_ON\_INDEX THEN

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

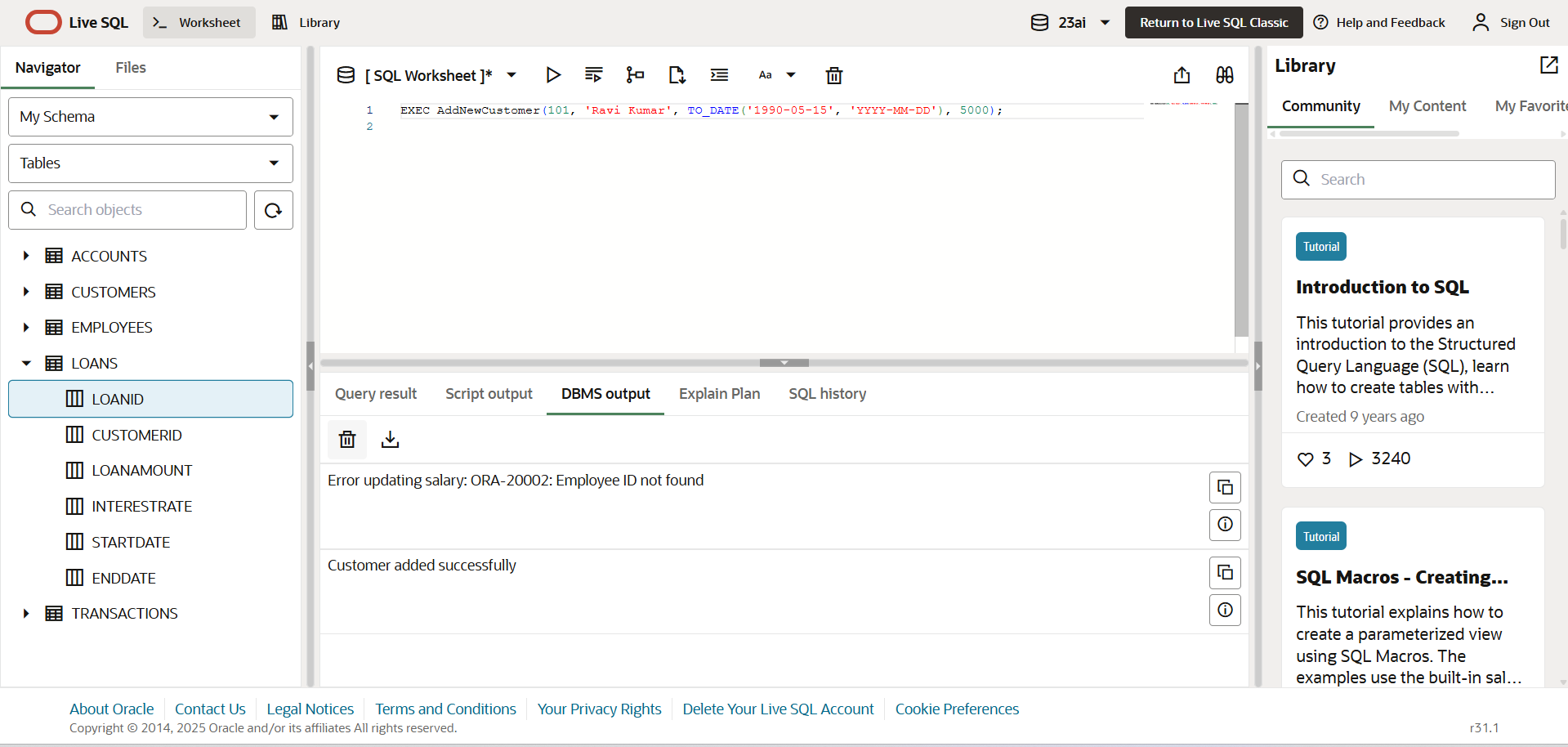
  WHEN OTHERS THEN

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

END;

SET SERVEROUTPUT ON;

EXEC AddNewCustomer(101, 'Ravi Kumar', TO\_DATE('1990-05-15', 'YYYY-MM-DD'), 5000);

**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:**CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest AS

BEGIN

  FOR acc IN (

    SELECT AccountID, Balance

    FROM Accounts

    WHERE AccountType = 'Savings'

  ) LOOP

    UPDATE Accounts

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

        LastModified = SYSDATE

    WHERE AccountID = acc.AccountID;

  END LOOP;

  COMMIT;

  DBMS\_OUTPUT.PUT\_LINE('Monthly interest processed for all savings accounts.');

END;

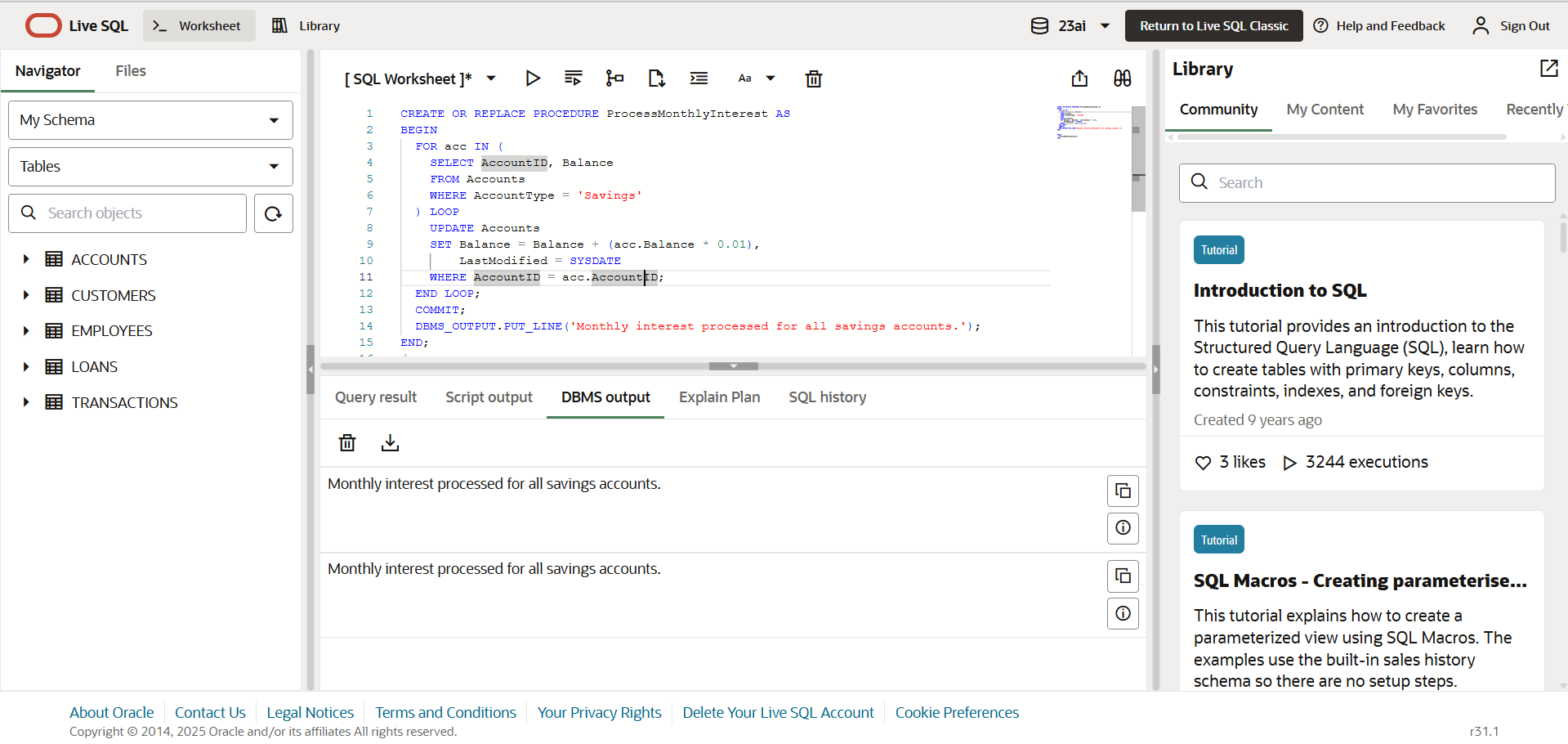
/

BEGIN

  ProcessMonthlyInterest;

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

) AS

BEGIN

  UPDATE Employees

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

  WHERE Department = p\_department;

  COMMIT;

  DBMS\_OUTPUT.PUT\_LINE('Bonus updated for department: ' || p\_department);

END;

/

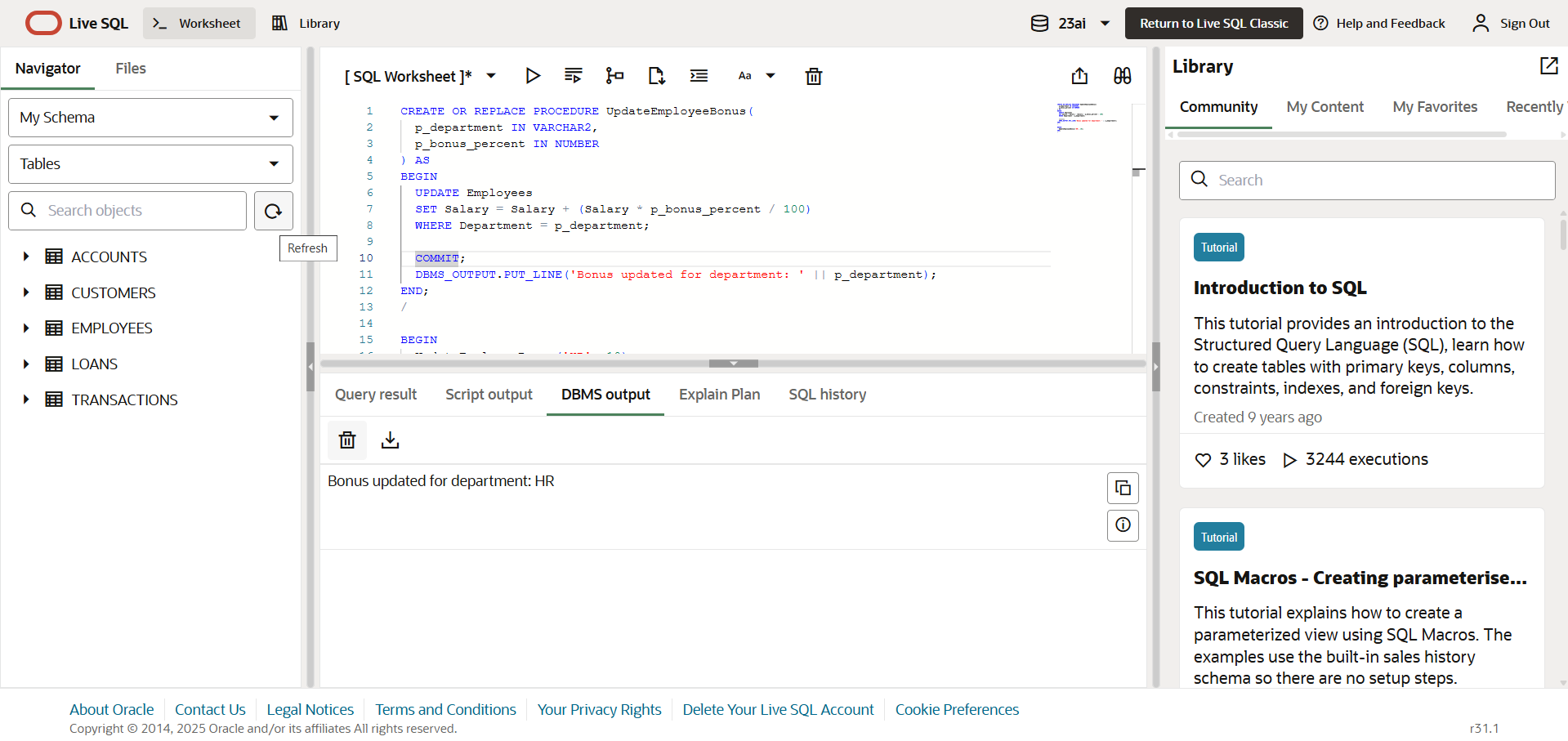
BEGIN

  UpdateEmployeeBonus('HR', 10);

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\_from\_account IN NUMBER,

  p\_to\_account IN NUMBER,

  p\_amount IN NUMBER

) AS

  v\_balance NUMBER;

BEGIN

  SELECT Balance INTO v\_balance

  FROM Accounts

  WHERE AccountID = p\_from\_account

  FOR UPDATE;

  IF v\_balance < p\_amount THEN

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

  END IF;

  UPDATE Accounts

  SET Balance = Balance - p\_amount,

      LastModified = SYSDATE

  WHERE AccountID = p\_from\_account;

  UPDATE Accounts

  SET Balance = Balance + p\_amount,

      LastModified = SYSDATE

  WHERE AccountID = p\_to\_account;

  INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)

  VALUES (Transactions\_seq.NEXTVAL, p\_from\_account, SYSDATE, p\_amount, 'Debit');

  INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)

  VALUES (Transactions\_seq.NEXTVAL, p\_to\_account, SYSDATE, p\_amount, 'Credit');

  COMMIT;

  DBMS\_OUTPUT.PUT\_LINE('Funds transferred successfully.');

EXCEPTION

  WHEN NO\_DATA\_FOUND THEN

    RAISE\_APPLICATION\_ERROR(-20002, 'One of the account IDs does not exist.');

  WHEN OTHERS THEN

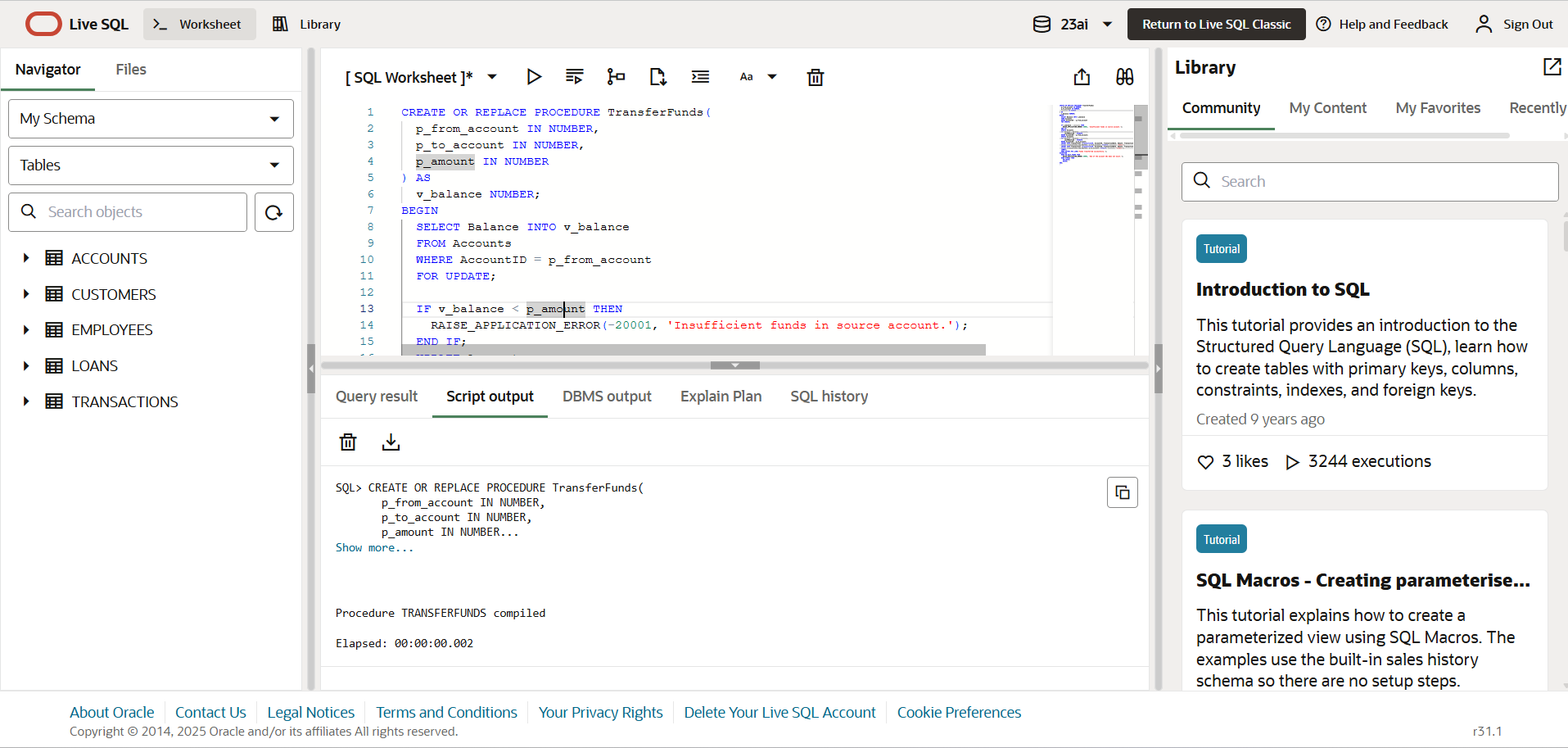
    ROLLBACK;

    RAISE;

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 DATE)

RETURN NUMBER IS

  v\_age NUMBER;

BEGIN

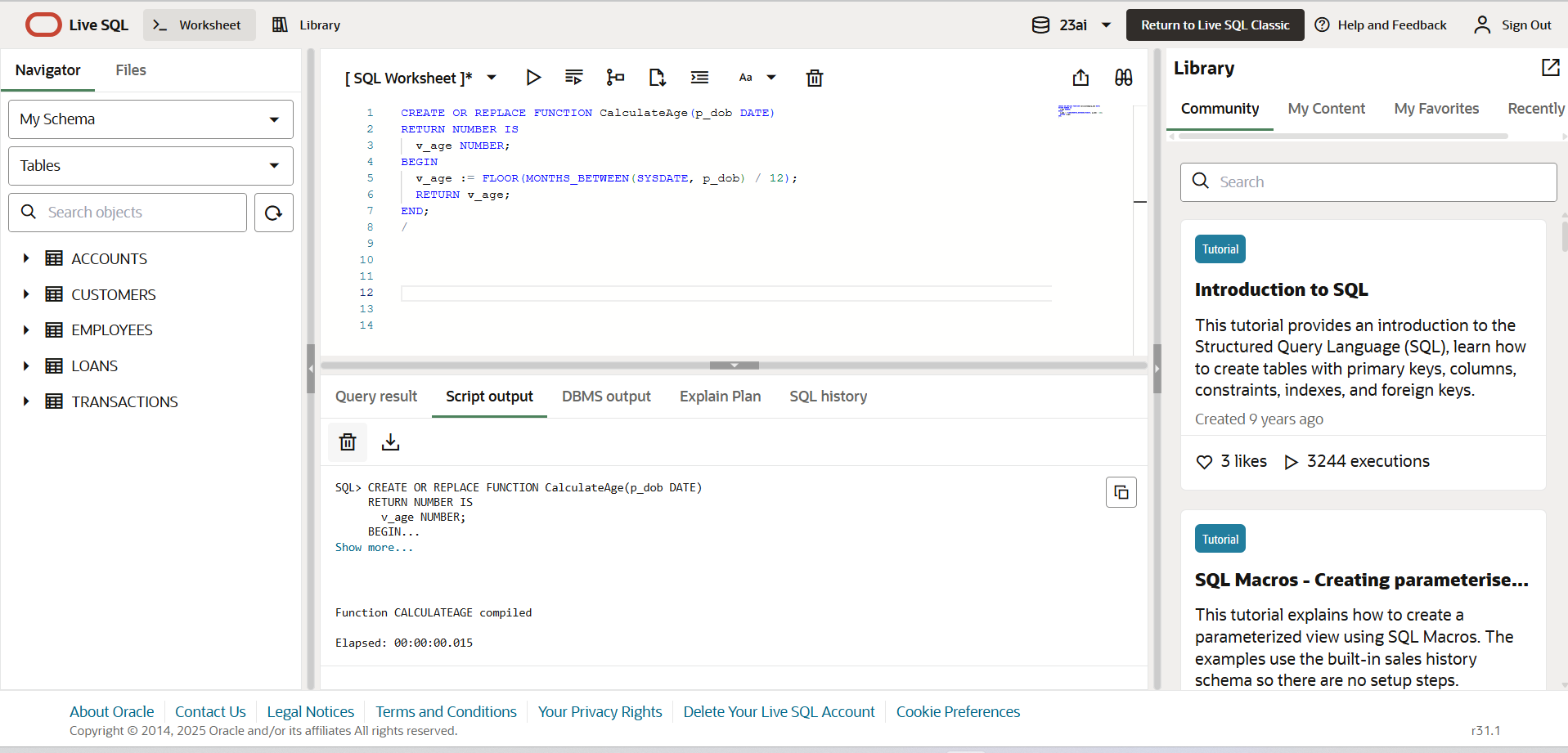
  v\_age := FLOOR(MONTHS\_BETWEEN(SYSDATE, p\_dob) / 12);

  RETURN v\_age;

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\_loan\_amount IN NUMBER,

p\_annual\_rate IN NUMBER,

p\_duration\_years IN NUMBER

)

RETURN NUMBER IS

v\_monthly\_rate NUMBER := p\_annual\_rate / 12 / 100;

v\_total\_months NUMBER := p\_duration\_years \* 12;

v\_emi NUMBER;

BEGIN

IF v\_monthly\_rate = 0 THEN

v\_emi := p\_loan\_amount / v\_total\_months;

ELSE

v\_emi := p\_loan\_amount \* v\_monthly\_rate \* POWER(1 + v\_monthly\_rate, v\_total\_months)

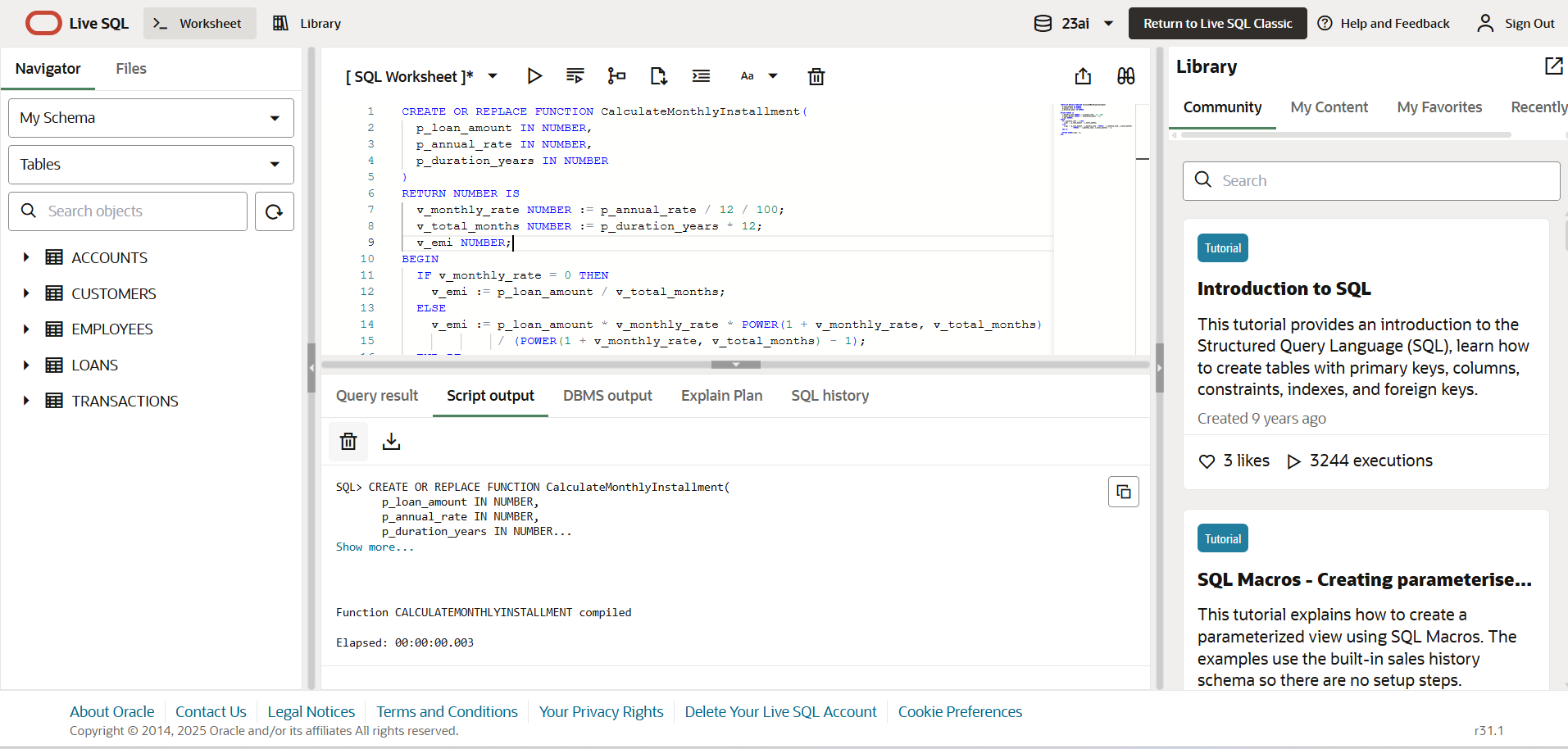
/ (POWER(1 + v\_monthly\_rate, v\_total\_months) - 1);

END IF;

RETURN ROUND(v\_emi, 2);

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\_account\_id IN NUMBER,

  p\_required\_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\_required\_amount;

EXCEPTION

  WHEN NO\_DATA\_FOUND THEN

    RETURN FALSE;

END;

**OUTPUT:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**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

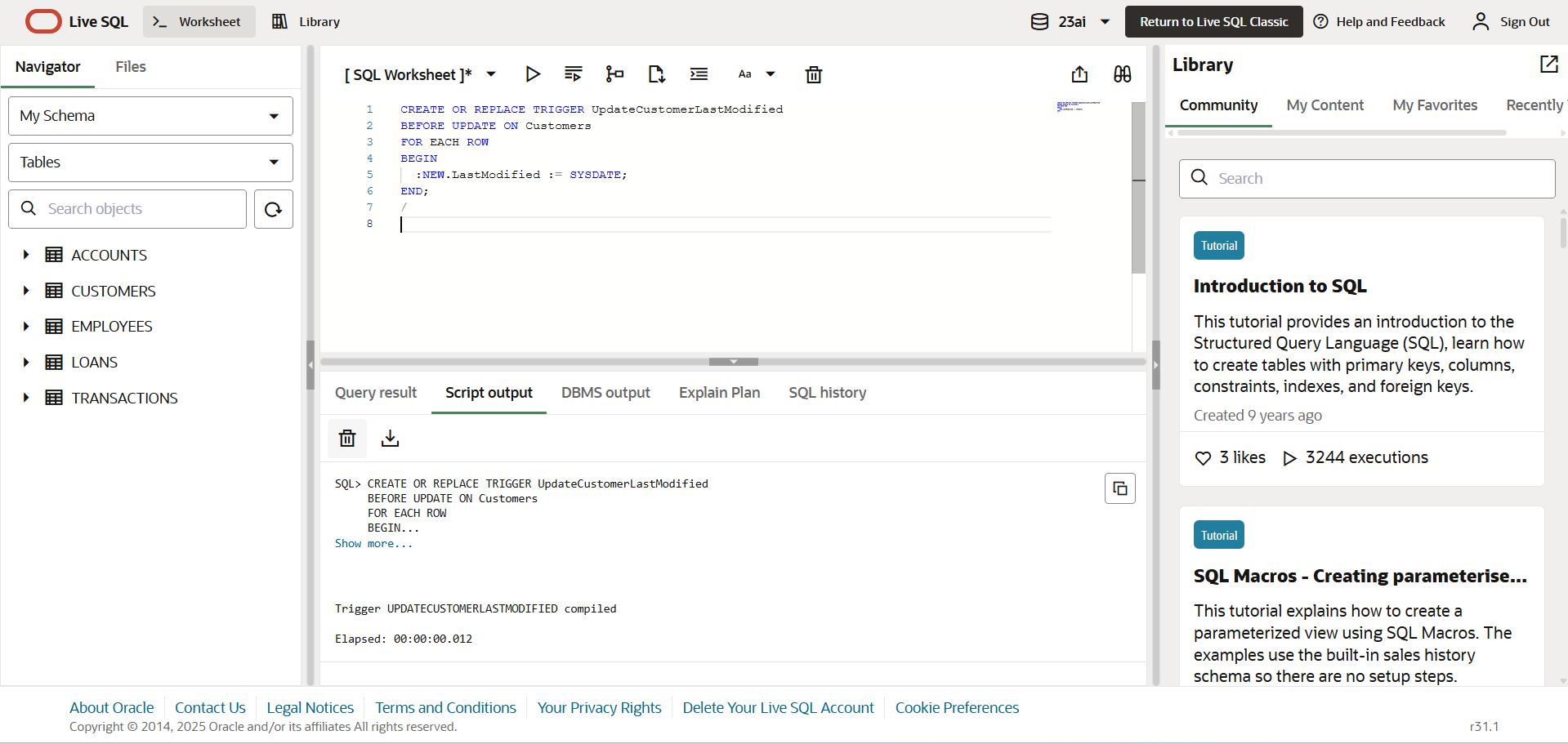
BEGIN

  :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 OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

  INSERT INTO AuditLog (

    AuditID, AccountID, TransactionDate, Amount, TransactionType, LoggedAt

  )

  VALUES (

    AuditLog\_seq.NEXTVAL,

    :NEW.AccountID,

    :NEW.TransactionDate,

    :NEW.Amount,

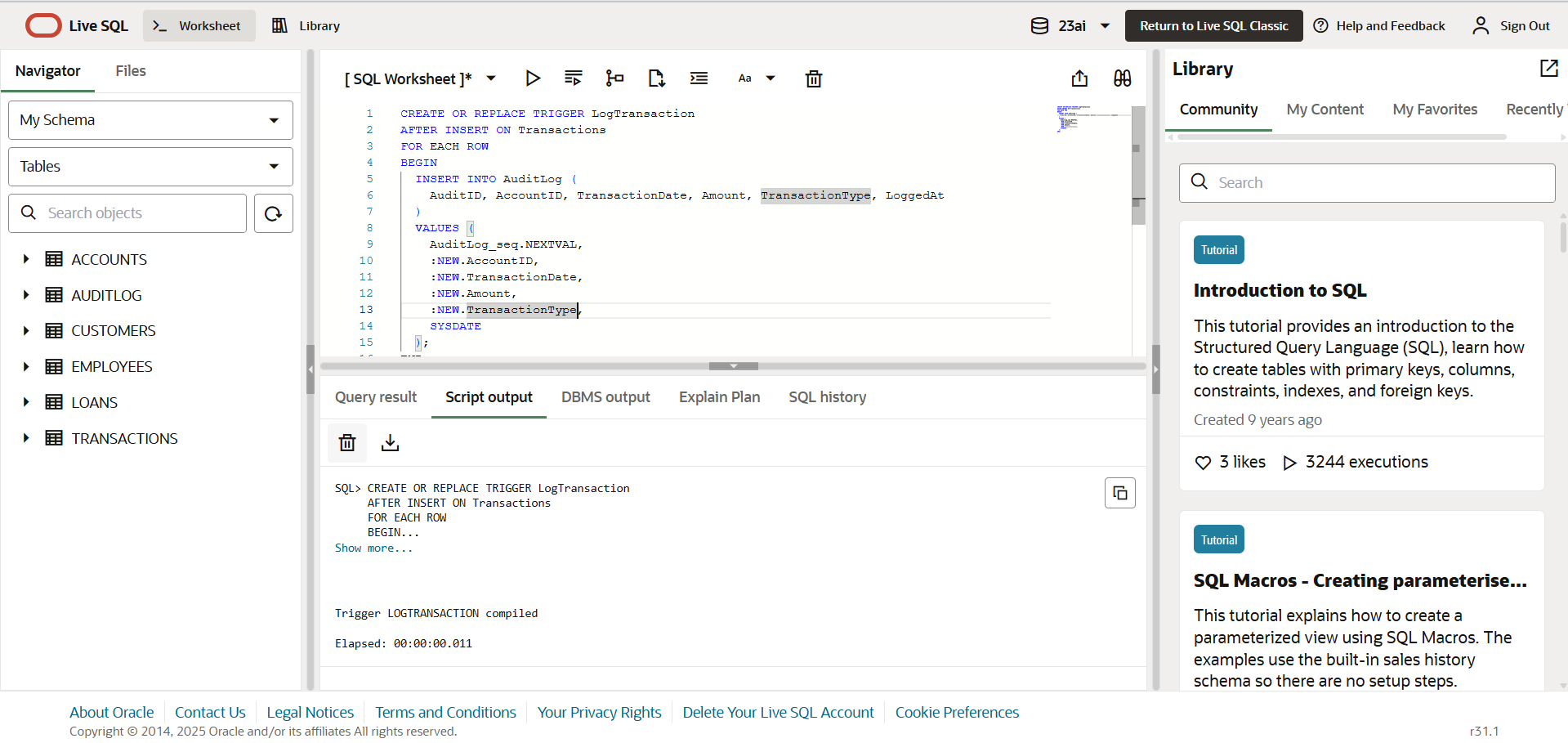
    :NEW.TransactionType,

    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

  SELECT Balance INTO v\_balance

  FROM Accounts

  WHERE AccountID = :NEW.AccountID;

  IF :NEW.TransactionType = 'Withdrawal' THEN

    IF :NEW.Amount <= 0 THEN

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

    ELSIF v\_balance < :NEW.Amount THEN

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

    END IF;

  ELSIF :NEW.TransactionType = 'Deposit' THEN

    IF :NEW.Amount <= 0 THEN

      RAISE\_APPLICATION\_ERROR(-20003, 'Deposit amount must be positive.');

    END IF;

  ELSE

    RAISE\_APPLICATION\_ERROR(-20004, 'Invalid transaction type.');

  END IF;

END;

/

**OUTPUT:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**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 cur\_monthly\_txns IS

    SELECT c.CustomerID, 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 TRUNC(t.TransactionDate, 'MM') = TRUNC(SYSDATE, 'MM')

    ORDER BY c.CustomerID, t.TransactionDate;

  v\_customer\_id Customers.CustomerID%TYPE;

  v\_name Customers.Name%TYPE;

  v\_date Transactions.TransactionDate%TYPE;

  v\_amount Transactions.Amount%TYPE;

  v\_type Transactions.TransactionType%TYPE;

BEGIN

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

  OPEN cur\_monthly\_txns;

  LOOP

    FETCH cur\_monthly\_txns INTO v\_customer\_id, v\_name, v\_date, v\_amount, v\_type;

    EXIT WHEN cur\_monthly\_txns%NOTFOUND;

    DBMS\_OUTPUT.PUT\_LINE('Customer: ' || v\_name || ' | Date: ' || v\_date ||

                         ' | Amount: ' || v\_amount || ' | Type: ' || v\_type);

  END LOOP;

  CLOSE cur\_monthly\_txns;

END;

/

**OUTPUT:  
A screenshot of a computer

AI-generated content may be incorrect.**

**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 cur\_accounts IS

    SELECT AccountID, Balance

    FROM Accounts;

  v\_account\_id Accounts.AccountID%TYPE;

  v\_balance Accounts.Balance%TYPE;

  v\_fee CONSTANT NUMBER := 500;

BEGIN

  OPEN cur\_accounts;

  LOOP

    FETCH cur\_accounts INTO v\_account\_id, v\_balance;

    EXIT WHEN cur\_accounts%NOTFOUND;

    UPDATE Accounts

    SET Balance = Balance - v\_fee,

        LastModified = SYSDATE

    WHERE AccountID = v\_account\_id;

    DBMS\_OUTPUT.PUT\_LINE('Annual fee applied to AccountID: ' || v\_account\_id);

  END LOOP;

  CLOSE cur\_accounts;

  COMMIT;

END;

/

**OUTPUT:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**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 cur\_loans IS

    SELECT LoanID, InterestRate

    FROM Loans;

  v\_loan\_id Loans.LoanID%TYPE;

  v\_rate Loans.InterestRate%TYPE;

BEGIN

  OPEN cur\_loans;

  LOOP

    FETCH cur\_loans INTO v\_loan\_id, v\_rate;

    EXIT WHEN cur\_loans%NOTFOUND;

    IF v\_rate < 6 THEN

      UPDATE Loans

      SET InterestRate = InterestRate + 0.5

      WHERE LoanID = v\_loan\_id;

    ELSE

      UPDATE Loans

      SET InterestRate = InterestRate + 0.25

      WHERE LoanID = v\_loan\_id;

    END IF;

    DBMS\_OUTPUT.PUT\_LINE('Updated interest for LoanID ' || v\_loan\_id);

  END LOOP;

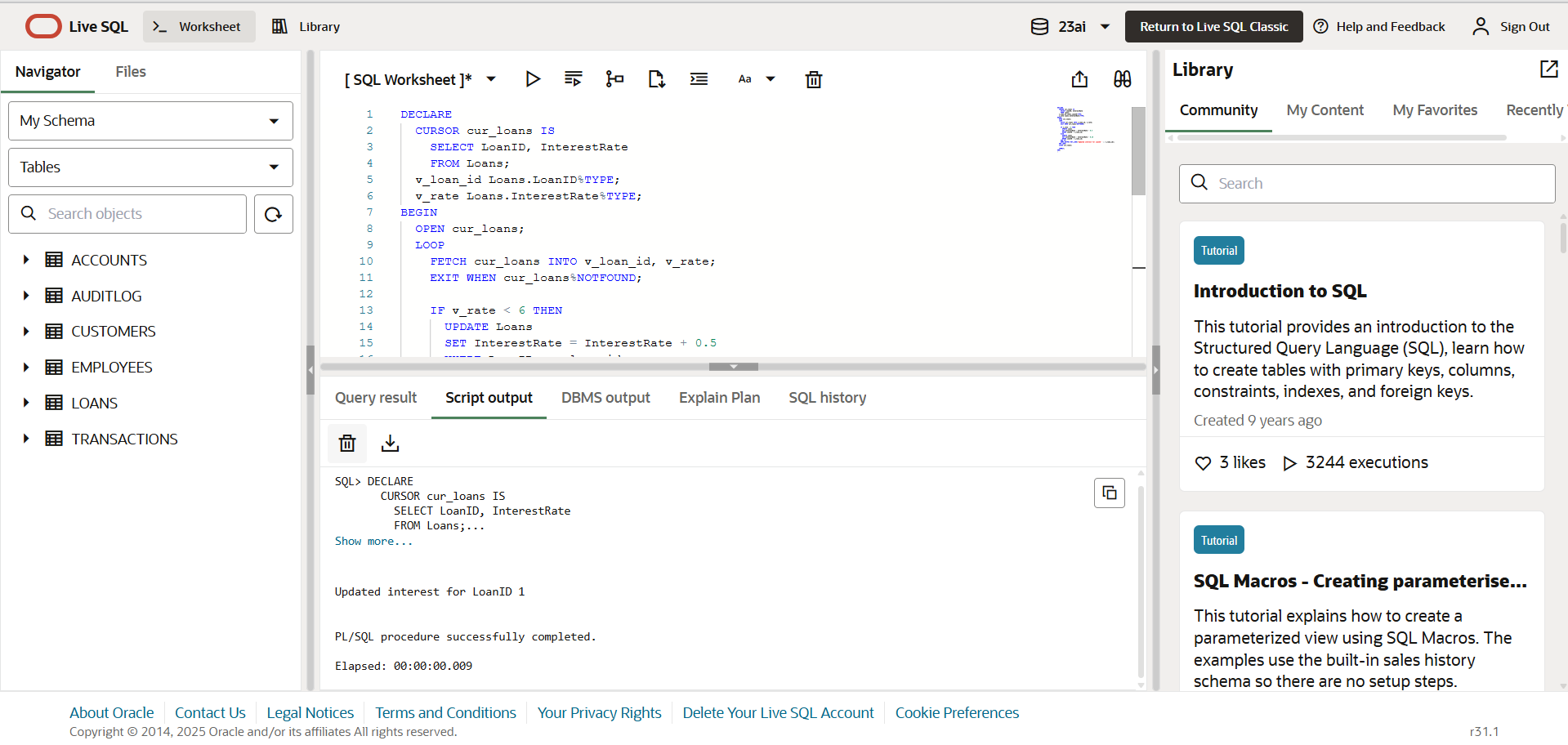
  CLOSE cur\_loans;

  COMMIT;

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 AS

  PROCEDURE AddCustomer(p\_id NUMBER, p\_name VARCHAR2, p\_dob DATE, p\_balance NUMBER);

  PROCEDURE UpdateCustomer(p\_id NUMBER, p\_name VARCHAR2, p\_dob DATE);

  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);

    COMMIT;

  END;

  PROCEDURE UpdateCustomer(p\_id NUMBER, p\_name VARCHAR2, p\_dob DATE) IS

  BEGIN

    UPDATE Customers

    SET Name = p\_name, DOB = p\_dob, LastModified = SYSDATE

    WHERE CustomerID = p\_id;

    COMMIT;

  END;

  FUNCTION GetCustomerBalance(p\_id NUMBER) RETURN NUMBER IS

    v\_balance NUMBER;

  BEGIN

    SELECT Balance INTO v\_balance

    FROM Customers

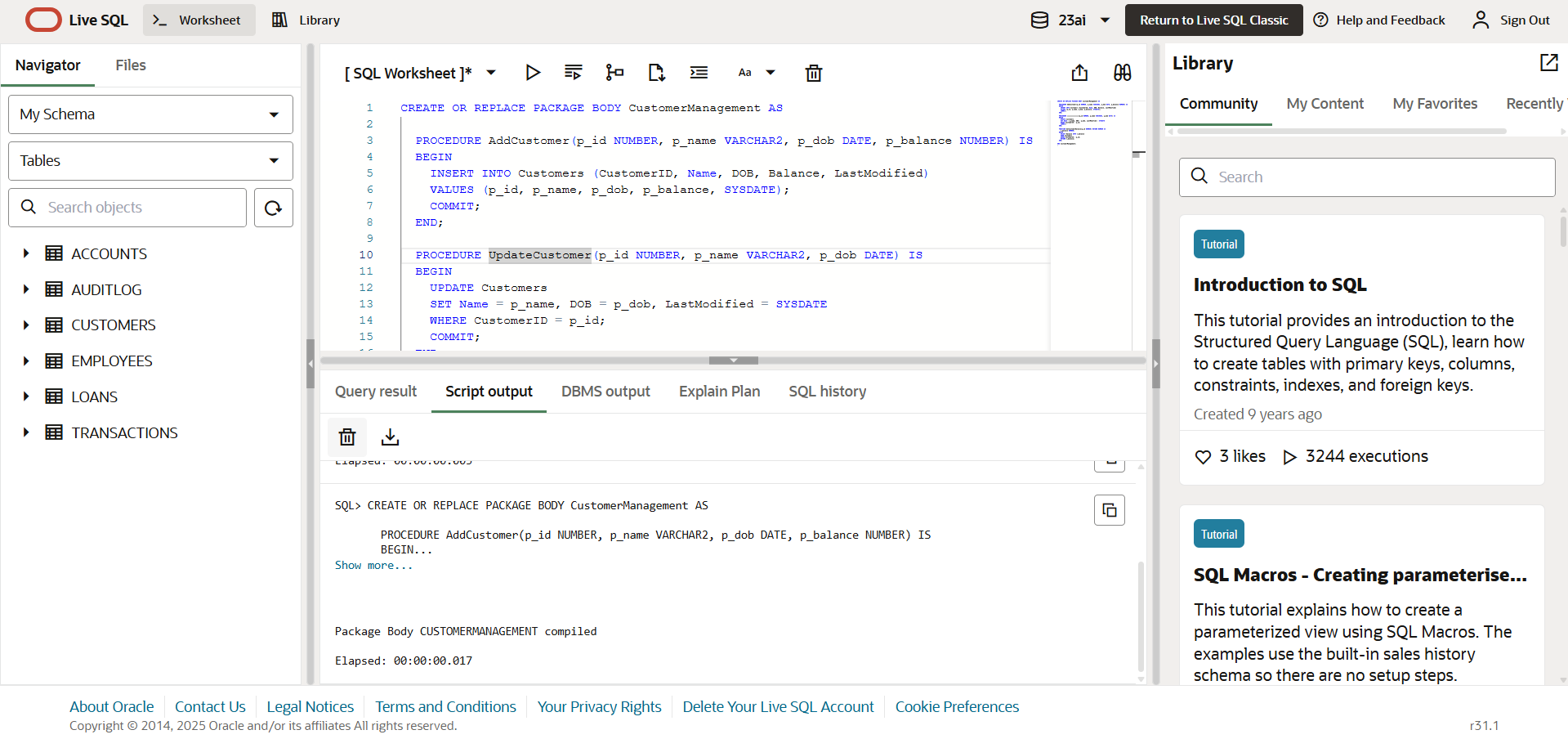
    WHERE CustomerID = p\_id;

    RETURN v\_balance;

  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 AS

PROCEDURE HireEmployee(p\_id NUMBER, p\_name VARCHAR2, p\_position VARCHAR2,

p\_salary NUMBER, p\_department VARCHAR2, p\_hire\_date DATE);

PROCEDURE UpdateEmployee(p\_id NUMBER, p\_salary NUMBER, p\_position VARCHAR2);

FUNCTION GetAnnualSalary(p\_id NUMBER) RETURN NUMBER;

END EmployeeManagement;

/

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

PROCEDURE HireEmployee(p\_id NUMBER, p\_name VARCHAR2, p\_position VARCHAR2,

p\_salary NUMBER, p\_department VARCHAR2, p\_hire\_date DATE) IS

BEGIN

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

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

COMMIT;

END;

PROCEDURE UpdateEmployee(p\_id NUMBER, p\_salary NUMBER, p\_position VARCHAR2) IS

BEGIN

UPDATE Employees

SET Salary = p\_salary, Position = p\_position

WHERE EmployeeID = p\_id;

COMMIT;

END;

FUNCTION GetAnnualSalary(p\_id NUMBER) RETURN NUMBER IS

v\_salary NUMBER;

BEGIN

SELECT Salary INTO v\_salary

FROM Employees

WHERE EmployeeID = p\_id;

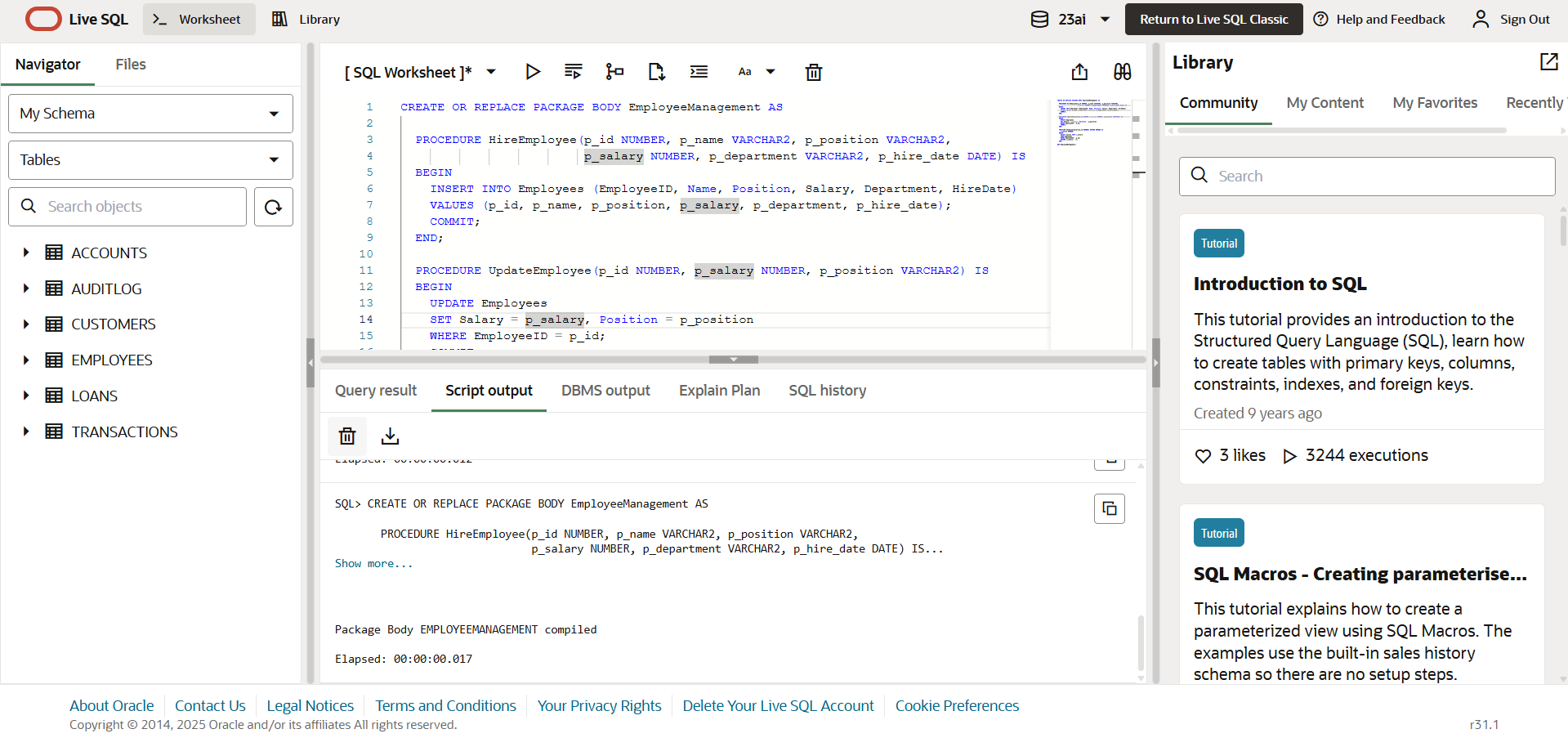
RETURN v\_salary \* 12;

END;

END EmployeeManagement;

/

**OUTPUT:**

****

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

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

**CODE:**

CREATE OR REPLACE PACKAGE AccountOperations AS

  PROCEDURE OpenAccount(p\_account\_id NUMBER, p\_customer\_id NUMBER, p\_type VARCHAR2, p\_balance NUMBER);

  PROCEDURE CloseAccount(p\_account\_id NUMBER);

  FUNCTION GetTotalBalance(p\_customer\_id NUMBER) RETURN NUMBER;

END AccountOperations;

/

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

PROCEDURE OpenAccount(p\_account\_id NUMBER, p\_customer\_id NUMBER, p\_type VARCHAR2, p\_balance NUMBER) IS

BEGIN

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

VALUES (p\_account\_id, p\_customer\_id, p\_type, p\_balance, SYSDATE);

COMMIT;

END;

PROCEDURE CloseAccount(p\_account\_id NUMBER) IS

BEGIN

DELETE FROM Accounts

WHERE AccountID = p\_account\_id;

COMMIT;

END;

FUNCTION GetTotalBalance(p\_customer\_id NUMBER) RETURN NUMBER IS

v\_total NUMBER := 0;

BEGIN

SELECT NVL(SUM(Balance), 0) INTO v\_total

FROM Accounts

WHERE CustomerID = p\_customer\_id;

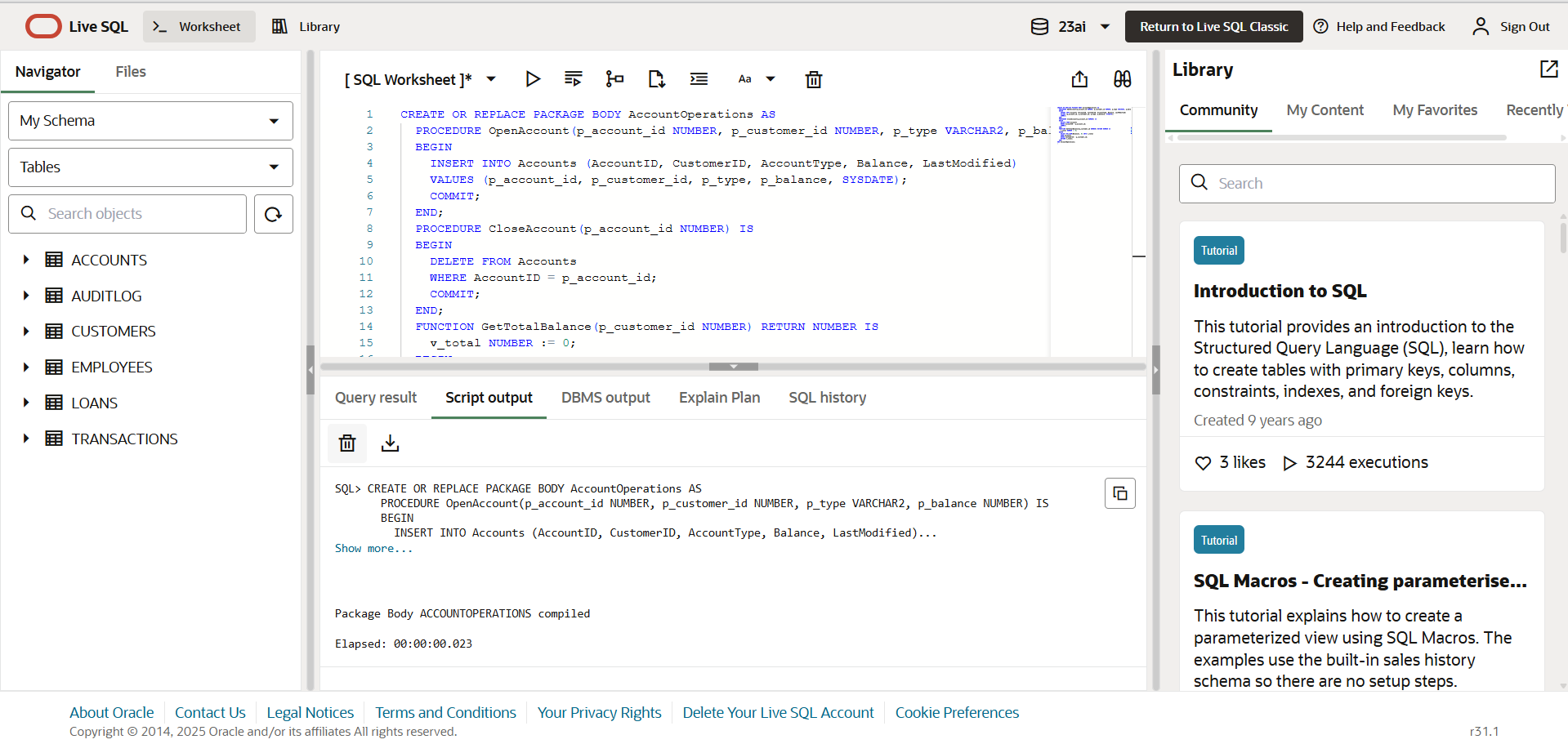
RETURN v\_total;

END;

END AccountOperations;

/

**OUTPUT:**

****