**Exercise 1: Control Structures**

**Scenario 1:** The bank wants to apply a discount to loan interest rates for customers above 60 years old.

Logic:

* Loop through all loans and check customer's age using DOB.
* If age > 60, reduce loan interest rate by 1%.

**Code:**

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;

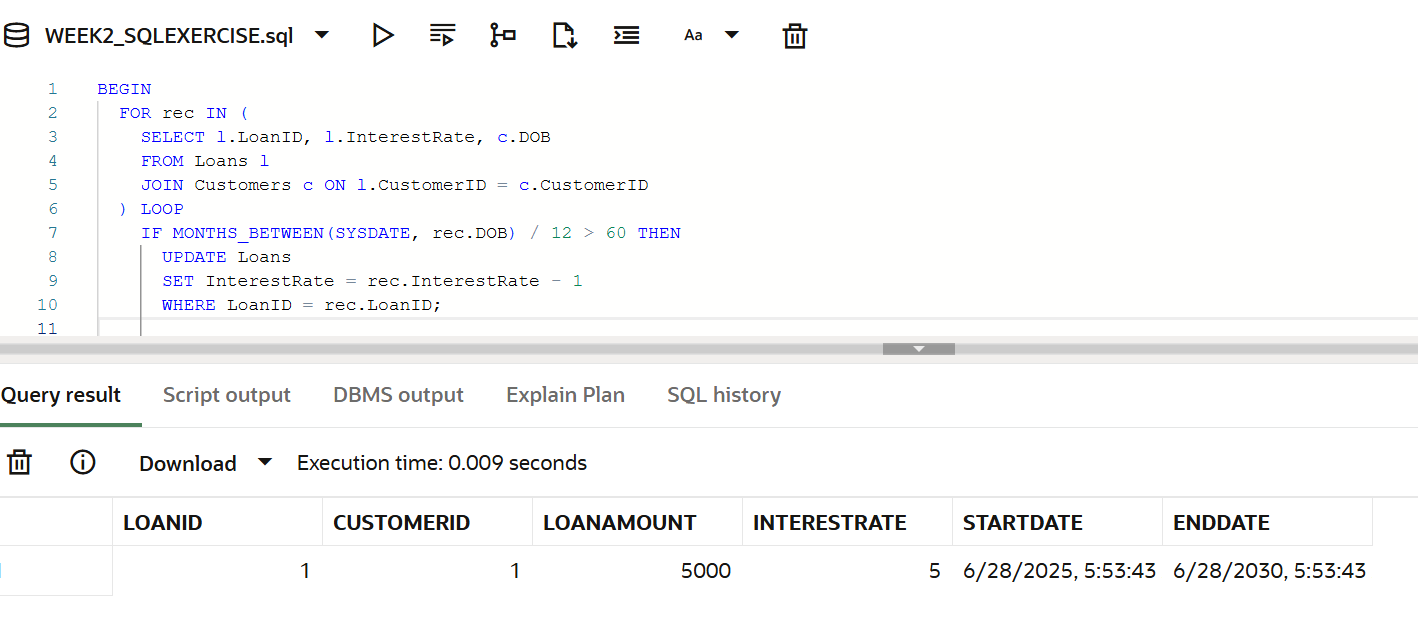
END IF;

END LOOP;

COMMIT;

END;

**Output:**



**Scenario 2:** A customer can be promoted to VIP status based on their balance.

**Logic:**

* add the IsVIP column
* If a customer's balance is greater than 10,000
* Update their IsVIP status to 'Y'

**Code:**

ALTER TABLE Customers ADD IsVIP CHAR(1) DEFAULT 'N';

BEGIN

  FOR rec IN (  SELECT CustomerID, Balance FROM Customers  )

LOOP

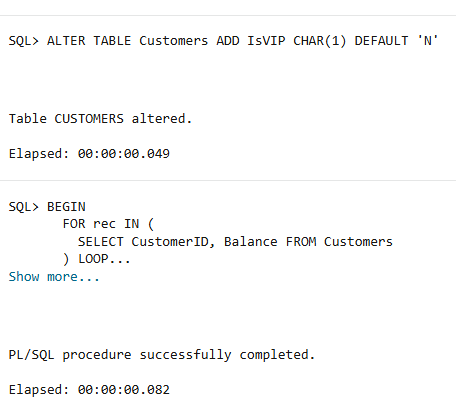
    IF rec.Balance > 10000 THEN UPDATE Customers SET IsVIP = 'Y' WHERE CustomerID = rec.CustomerID;

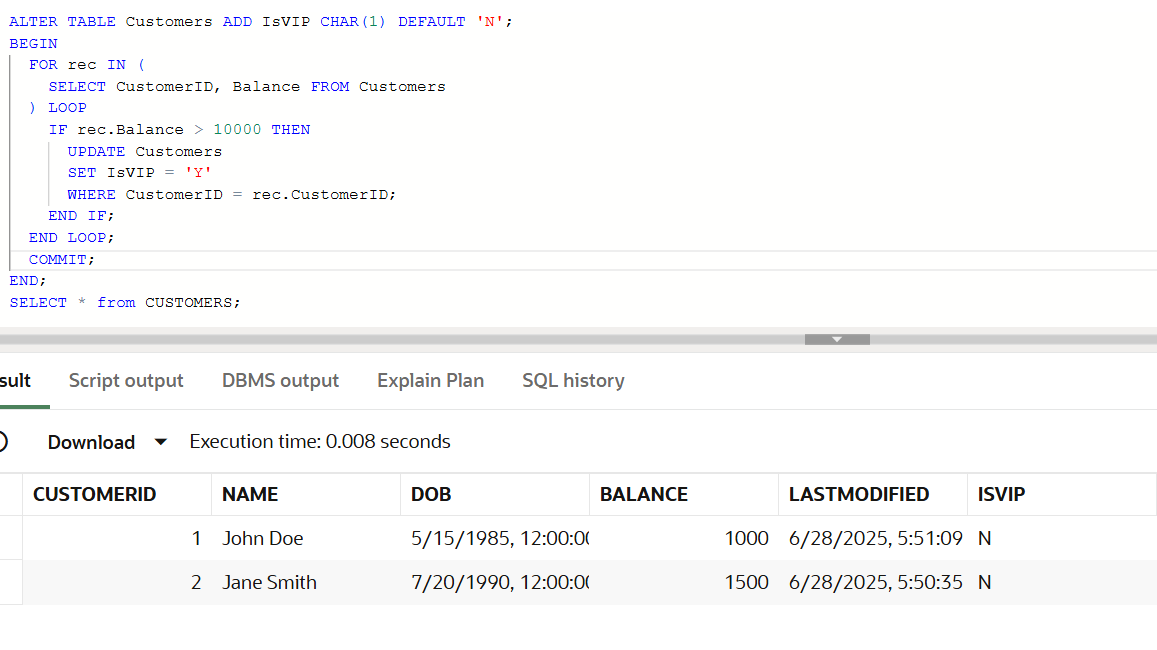
    END IF;

  END LOOP;

  COMMIT;

END;

**Output:**



**Scenario 3:** The bank wants to send reminders to customers whose loans are due within the next 30 days.

**Logic:**

* Check the loan EndDate between SYSDATE and SYSDATE + 30.
* Print a reminder for each customer.

**Code:**

BEGIN

FOR rec IN (

SELECT l.LoanID, l.EndDate, c.Name FROM Loans l

JOIN Customers c ON l.CustomerID = c.CustomerID WHERE l.EndDate BETWEEN SYSDATE AND SYSDATE + 3)

LOOP

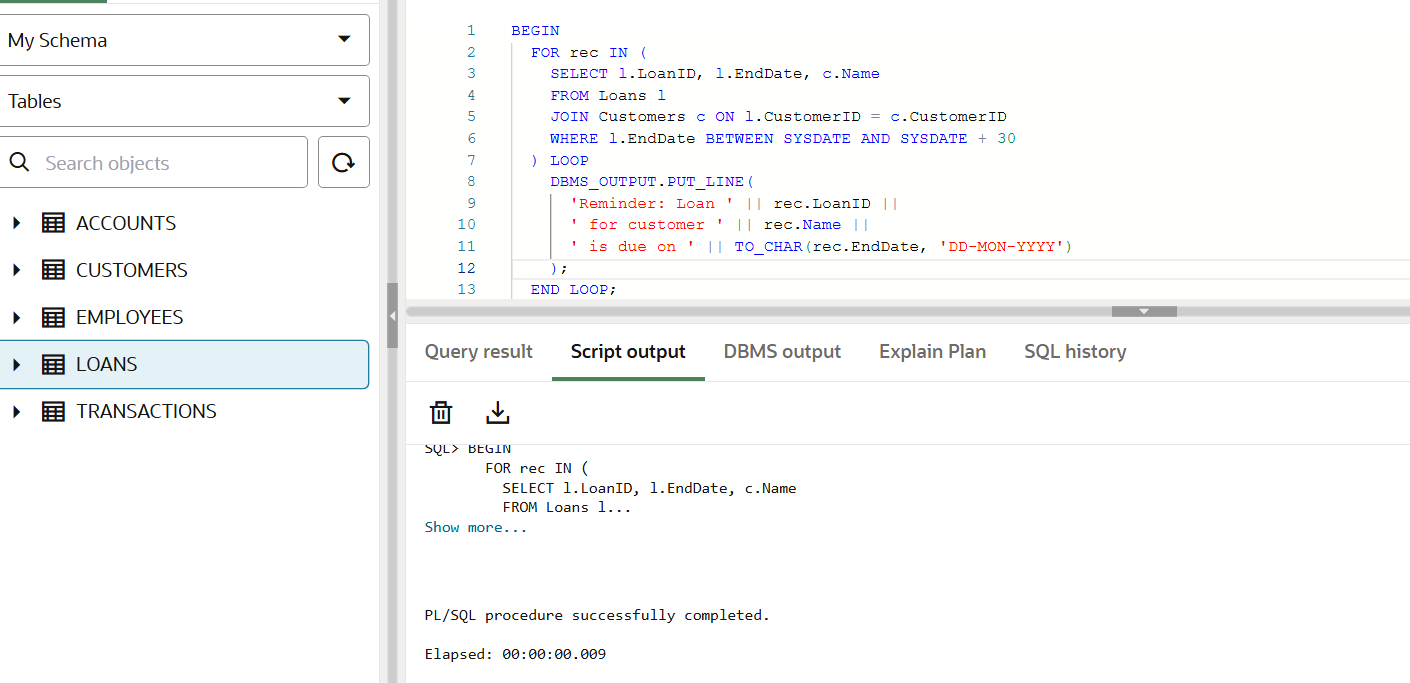
DBMS\_OUTPUT.PUT\_LINE(

'Reminder: Loan ' || rec.LoanID ||' for customer ' || rec.Name ||' is due on ' || TO\_CHAR(rec.EndDate, 'DD-MON-YYYY'));

END LOOP;

END;

**Output:**

****

**Exercise 2: Error Handling**

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

**Logic:**

* Create a procedure SafeTransferFunds(account\_from, account\_to, amount)
* Source account has **insufficient funds** => Log an error
* Any error occurs => ROLLBACK the transaction

**Code:**

CREATE OR REPLACE PROCEDURE SafeTransferFunds (p\_from\_account IN NUMBER,

p\_to\_account IN NUMBER,p\_amount IN NUMBER) IS v\_balance NUMBER;

BEGIN

-- Get balance of the source account

SELECT Balance INTO v\_balance FROM Accounts WHERE AccountID = p\_from\_account;

-- Check for sufficient balance

IF v\_balance < p\_amount THEN

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

END IF;

-- Deduct from source

UPDATE Accounts SET Balance = Balance - p\_amount WHERE AccountID = p\_from\_account;

-- Add to destination

UPDATE Accounts SET Balance = Balance + p\_amount WHERE AccountID = p\_to\_account;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Transfer successful from Account ' || p\_from\_account || ' to Account ' || p\_to\_account);

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END;

/

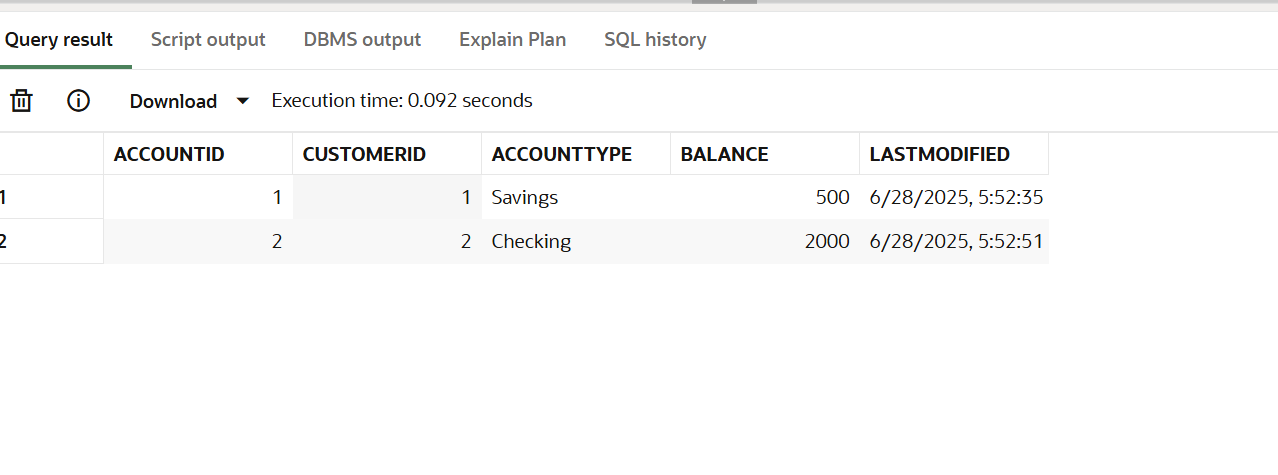
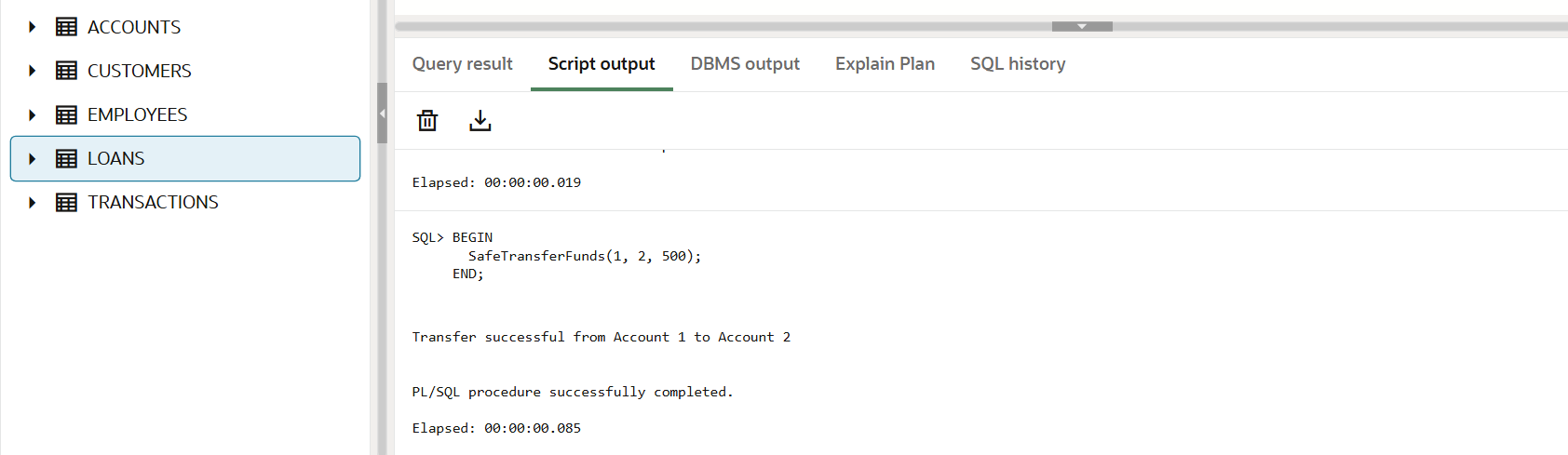
Call the procedure

BEGIN

SafeTransferFunds(1, 2, 500);

END;

**Output:**

**Scenario 2:** Manage errors when updating employee salaries.

**Logic:**

* Create a stored procedure UpdateSalary
* Inputs: EmployeeID, percentage
* If EmployeeID doesn't exist, catch the exception and **log an error message**
* Otherwise, increase the salary

**Code:**

CREATE PROCEDURE UpdateSalary (p\_emp\_id IN NUMBER,p\_percentage IN NUMBER) IS

  v\_current\_salary NUMBER;

BEGIN

  -- Try to get current salary

  SELECT Salary INTO v\_current\_salary FROM Employees WHERE EmployeeID = p\_emp\_id;

  -- Update salary

  UPDATE Employees SET Salary = Salary + (Salary \* p\_percentage / 100)  WHERE EmployeeID = p\_emp\_id;

  COMMIT;

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

EXCEPTION

  WHEN NO\_DATA\_FOUND THEN

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

  WHEN OTHERS THEN

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

END;

/

Call the procedure

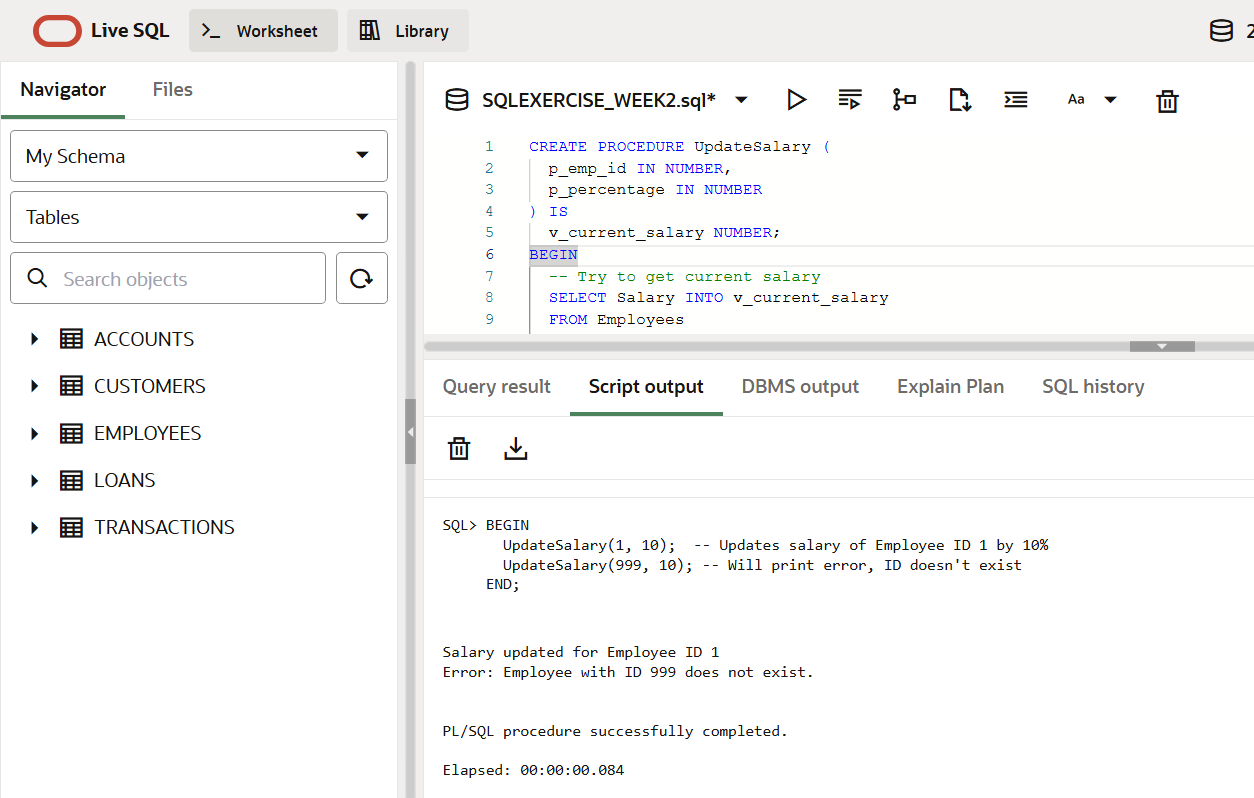
BEGIN

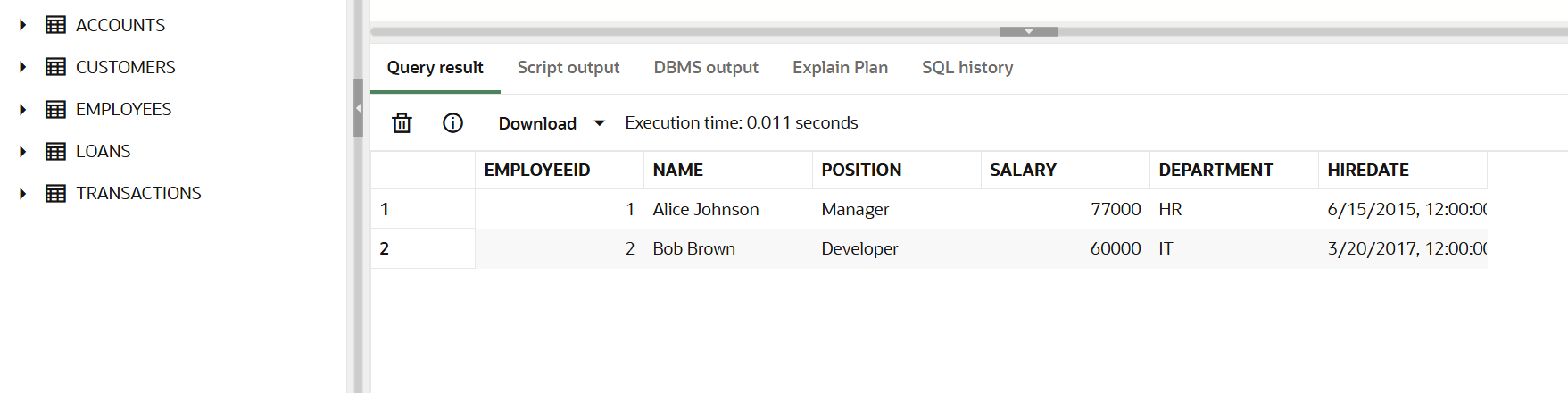
  UpdateSalary(1, 10);  -- Updates salary of Employee ID 1 by 10%

  UpdateSalary(999, 10); -- Will print error, ID doesn't exist

END;

**Output:**





**Scenario 3:** Ensure data integrity when adding a new customer.

Logic:

* Write a procedure AddNewCustomer
* Inputs: CustomerID, Name, DOB, Balance
* If CustomerID already exists:
  + **Prevent insertion**
  + **Log an error message** using DBMS\_OUTPUT

**Code:**

CREATE PROCEDURE AddNewCustomer (

  p\_cust\_id IN NUMBER,

  p\_name IN VARCHAR2,

  p\_dob IN DATE,

  p\_balance IN NUMBER

) IS

BEGIN

  -- Try inserting the new customer

  INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)

  VALUES (p\_cust\_id, p\_name, p\_dob, p\_balance, SYSDATE);

  COMMIT;

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

EXCEPTION

  WHEN DUP\_VAL\_ON\_INDEX THEN

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

  WHEN OTHERS THEN

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

END;

/

**Call the procedure:**

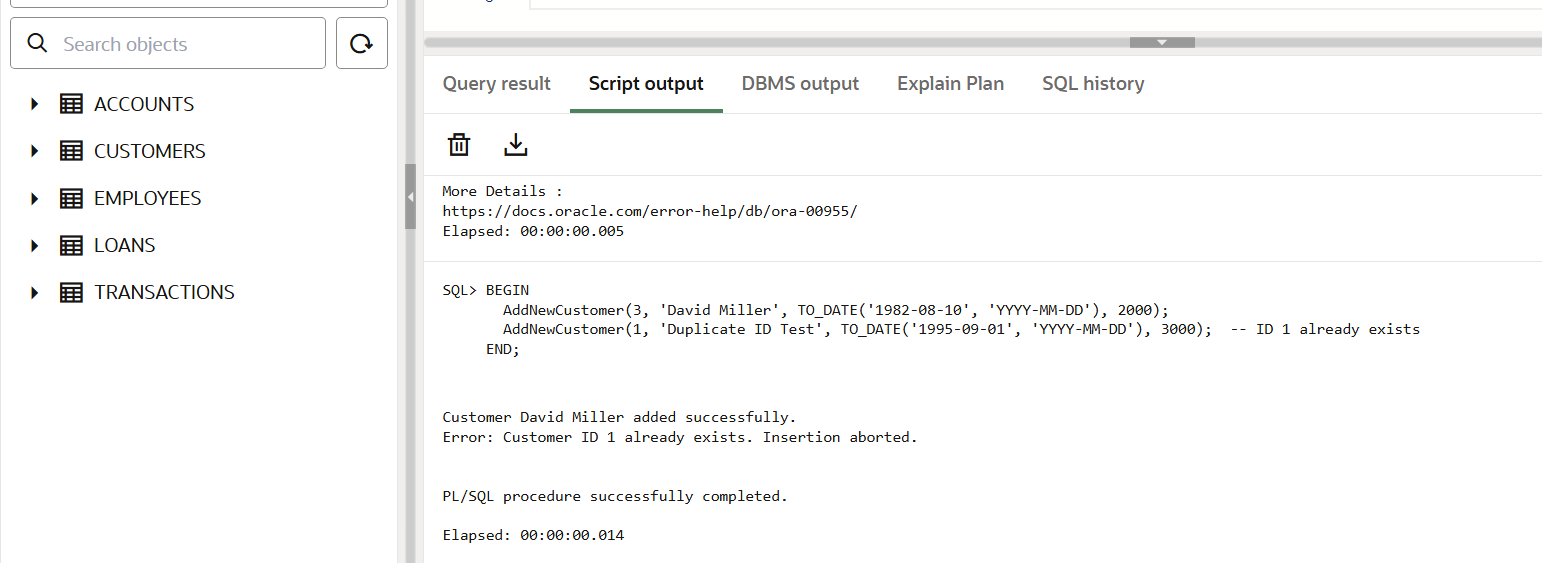
BEGIN

AddNewCustomer(3, 'David Miller', TO\_DATE('1982-08-10', 'YYYY-MM-DD'), 2000);

AddNewCustomer(1, 'Duplicate ID Test', TO\_DATE('1995-09-01', 'YYYY-MM-DD'), 3000);

END;

**Output:**

****

**Exercise 3: Stored Procedures**

**Scenario 1:** The bank needs to process monthly interest for all savings accounts.

**Logic:**

* Procedure name: ProcessMonthlyInterest
* Apply 1% interest to accounts of type 'Savings'

**Code:**

CREATE PROCEDURE ProcessMonthlyInterest IS

BEGIN

  FOR rec IN (

    SELECT AccountID, Balance FROM Accounts WHERE AccountType = 'Savings'

  ) LOOP

    UPDATE Accounts SET Balance = rec.Balance + (rec.Balance \* 0.01)

    WHERE AccountID = rec.AccountID;

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

  END LOOP;

  COMMIT;

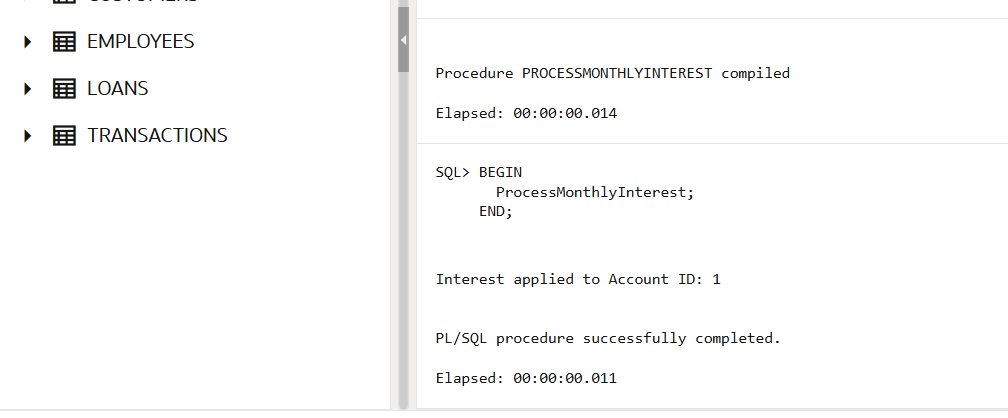
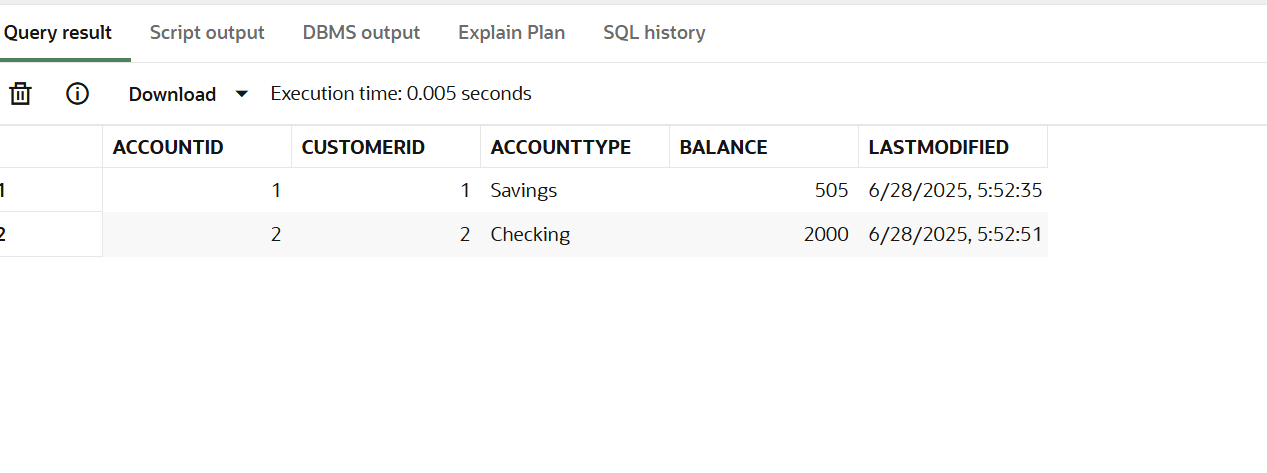
END;

/

Call Procedure:  
 BEGIN

ProcessMonthlyInterest;

END;

**Output**

**Scenario 2:** The bank wants to implement a bonus scheme for employees based on their performance.

**Logic:**

* Procedure name: UpdateEmployeeBonus
* Increases the salary of all employees in that department by the given bonus percentage

**Code:**

CREATE PROCEDURE UpdateEmployeeBonus (p\_department IN VARCHAR2,p\_bonus\_pct IN NUMBER) IS

BEGIN

  FOR rec IN (

    SELECT EmployeeID, Salary FROM Employees WHERE Department = p\_department )

LOOP

    UPDATE Employees SET Salary = rec.Salary + (rec.Salary \* p\_bonus\_pct / 100)

    WHERE EmployeeID = rec.EmployeeID;

    DBMS\_OUTPUT.PUT\_LINE('Bonus applied to Employee ID: ' || rec.EmployeeID);

  END LOOP;

  COMMIT;

END;

/

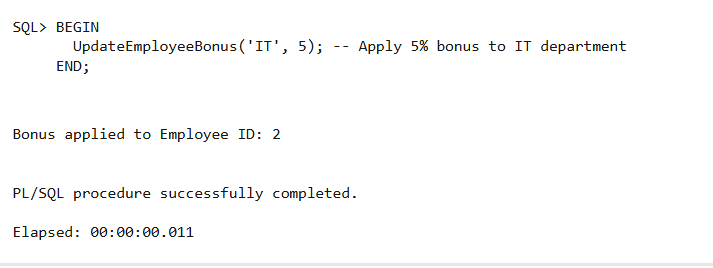
Call the Procedure:

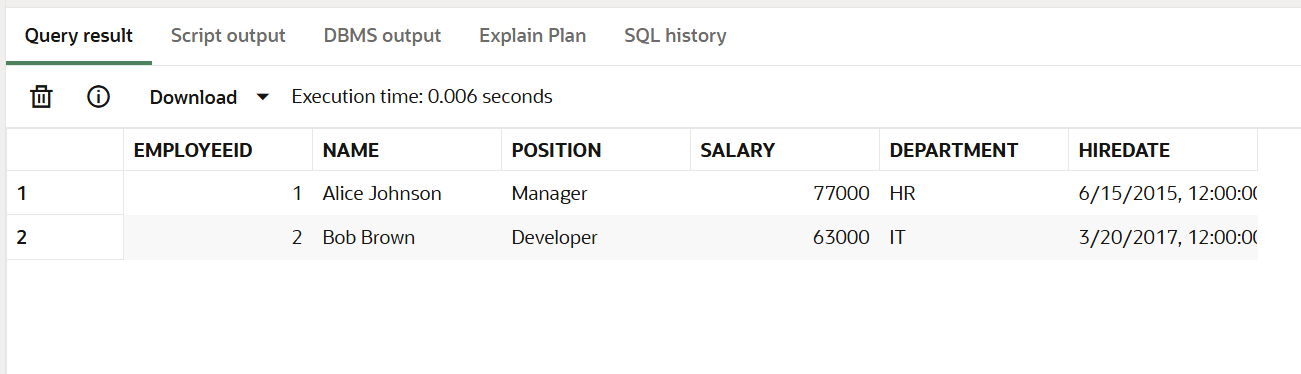
BEGIN

UpdateEmployeeBonus('IT', 5);

END;

**Output:**

****

****

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

**Logic:**

* Procedure name: TransferFunds
* Before transferring:
  + Check that from\_account has sufficient balance
  + If not, show an error
  + Else, debit from source and credit to destination

**Code:**

CREATE PROCEDURE TransferFunds (p\_from\_account IN NUMBER, p\_to\_account IN NUMBER, p\_amount IN NUMBER) IS v\_balance NUMBER;

BEGIN

  -- Get current balance of source account

  SELECT Balance INTO v\_balance FROM Accounts WHERE AccountID = p\_from\_account;

  -- Check for sufficient funds

  IF v\_balance < p\_amount THEN

    DBMS\_OUTPUT.PUT\_LINE('Transfer failed: Insufficient balance in Account ' || p\_from\_account);

  ELSE

    -- Deduct from source

  UPDATE Accounts SET Balance = Balance - p\_amount WHERE AccountID = p\_from\_account;

    -- Add to destination

    UPDATE Accounts SET Balance = Balance + p\_amount  WHERE AccountID = p\_to\_account;

    COMMIT;

    DBMS\_OUTPUT.PUT\_LINE('Successfully transferred ₹' || p\_amount || ' from Account ' || p\_from\_account ||  ' to Account ' || p\_to\_account);

  END IF;

EXCEPTION

  WHEN NO\_DATA\_FOUND THEN

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

  WHEN OTHERS THEN

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

END;

/

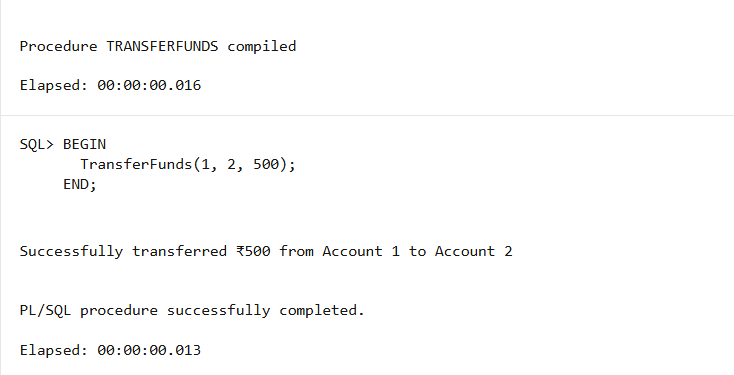
Call the Procedure:

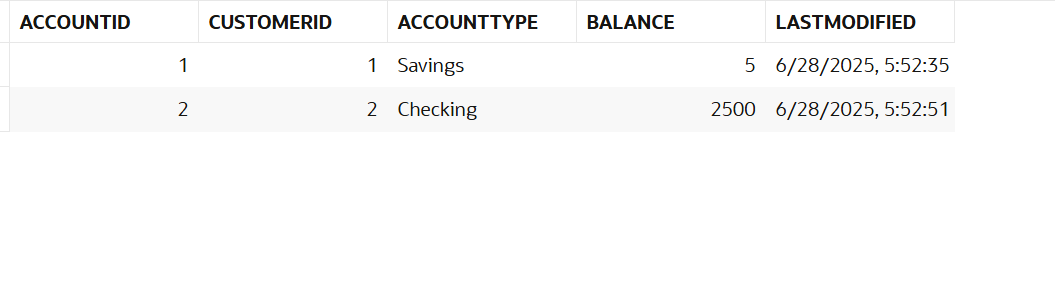
BEGIN

  TransferFunds(1, 2, 500);

END;

**Output:**



**Exercise 4: Functions**

**Scenario 1:** Calculate the age of customers for eligibility checks.

**Logic:**

* Function name: CalculateAge
* Return: Age in years

**Code:**

CREATE FUNCTION CalculateAge ( p\_dob IN DATE) RETURN NUMBER IS

v\_age NUMBER;

BEGIN

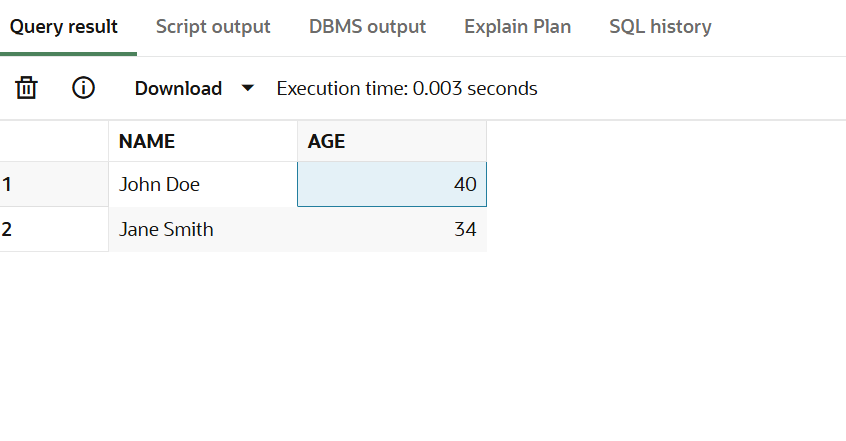
v\_age := FLOOR(MONTHS\_BETWEEN(SYSDATE, p\_dob) / 12);

RETURN v\_age;

END;

/

**Output:**



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

Logic:

* Function name: CalculateMonthlyInstallment
* Monthly EMI (as a NUMBER, rounded to 2 decimal places)

**Code:**

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment ( p\_amount IN NUMBER,p\_rate IN NUMBER, p\_years IN NUMBER)

RETURN NUMBER IS

r NUMBER := p\_rate / 12 / 100; -- monthly interest rate

n NUMBER := p\_years \* 12; -- number of months

emi NUMBER;

BEGIN

IF r = 0 THEN

emi := p\_amount / n; -- No interest case

ELSE

emi := p\_amount \* r \* POWER(1 + r, n) / (POWER(1 + r, n) - 1);

END IF;

RETURN ROUND(emi, 2);

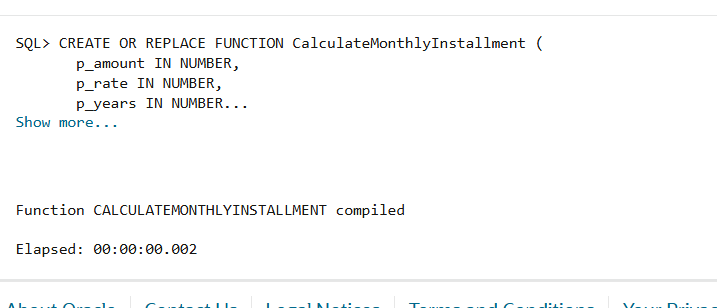
END;

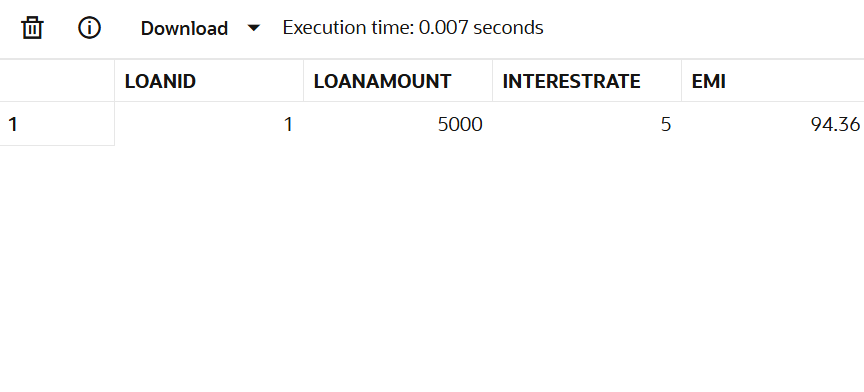
/

--Query

SELECT LoanID,LoanAmount,InterestRate,CalculateMonthlyInstallment(LoanAmount,InterestRate, 5) AS EMI FROM Loans;

**Output:**





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

**Logic:**

* Function name: HasSufficientBalance
* Output: BOOLEAN(YES or NO)

**Code:**

CREATE OR REPLACE FUNCTION HasSufficientBalance (p\_account\_id IN NUMBER,

  p\_amount IN NUMBER)

RETURN VARCHAR2 IS v\_balance NUMBER;

BEGIN

  SELECT Balance INTO v\_balance FROM Accounts WHERE AccountID = p\_account\_id;

  IF v\_balance >= p\_amount THEN

    RETURN 'YES';

  ELSE

    RETURN 'NO';

  END IF;

EXCEPTION

  WHEN NO\_DATA\_FOUND THEN

    RETURN 'NO';

  WHEN OTHERS THEN

    RETURN 'NO';

END;

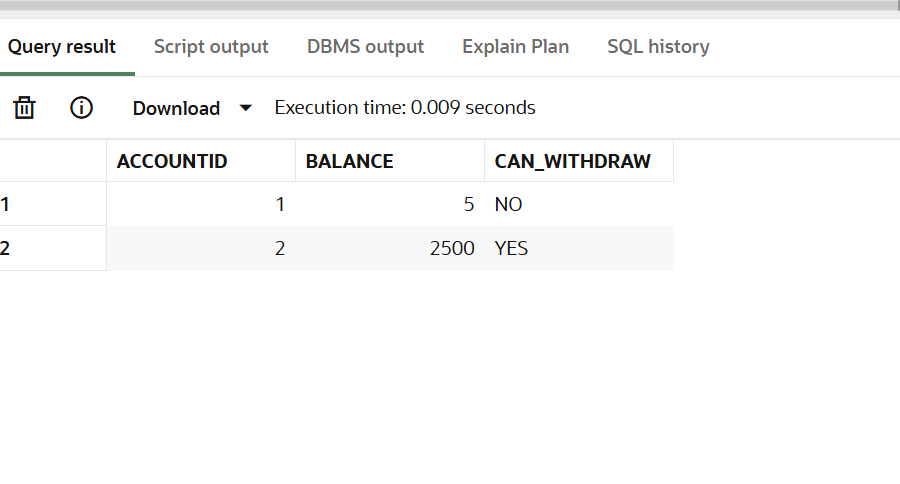
/

--Query

SELECT AccountID, Balance, HasSufficientBalance(AccountID, 1000) AS Can\_Withdraw

FROM Accounts;

Output:



**Exercise 5: Triggers**

**Scenario 1:** Automatically update the last modified date when a customer's record is updated.

**Code:**

create or replace trigger UpdateCustomerLastModified before update on Customers for each row

begin

:new.LastModified := sysdate;

end;

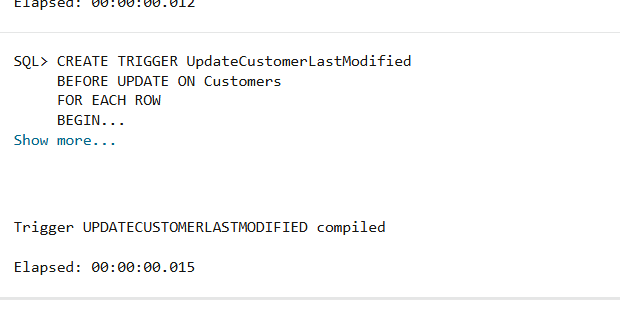
**/**

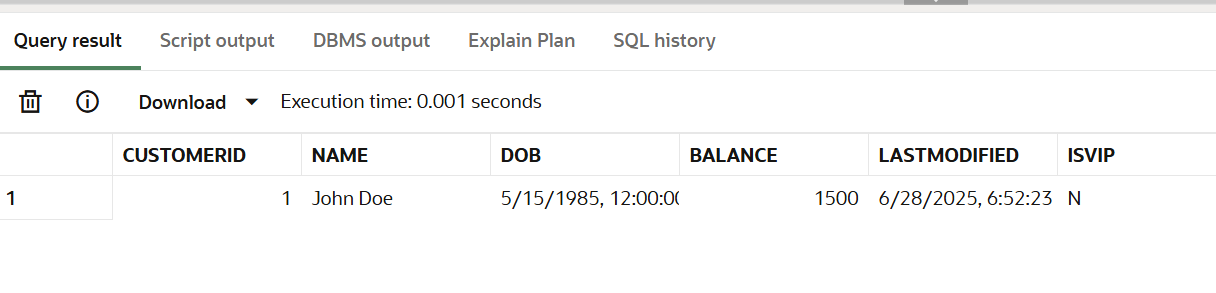
--To Test

UPDATE Customers SET Balance = Balance + 500 WHERE CustomerID = 1;

SELECT \* FROM Customers WHERE CustomerID = 1;

**Output:**

****

****

**Scenario 2: Maintain an audit log for all transactions.**

**Code:**

--create audit table

create table AuditLog (

LogID number primary key,

AccountID number,

Action varchar2(50),

LogDate date

);

create trigger LogTransaction

after insert on Transactions

for each row

begin

  insert into AuditLog(LogID, AccountID, Action, LogDate)

  values(AuditLog\_seq.nextval, :new.AccountID, 'Transaction Added', sysdate);

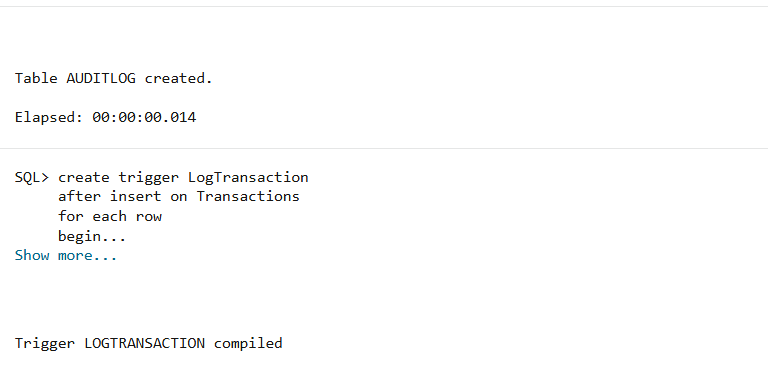
end;

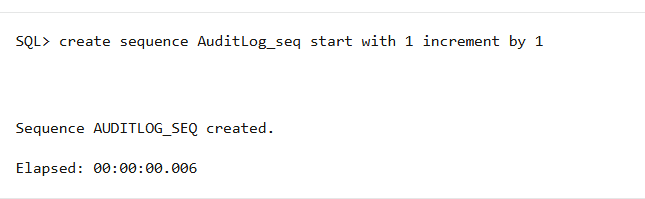
/

--to Test

create sequence AuditLog\_seq start with 1 increment by 1;

**Output:**





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

**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' and :new.Amount > v\_balance then

    raise\_application\_error(-20001, 'Not enough balance.');

  end if;

  if :new.TransactionType = 'Deposit' and :new.Amount <= 0 then

    raise\_application\_error(-20002, 'Deposit must be positive.');

  end if;

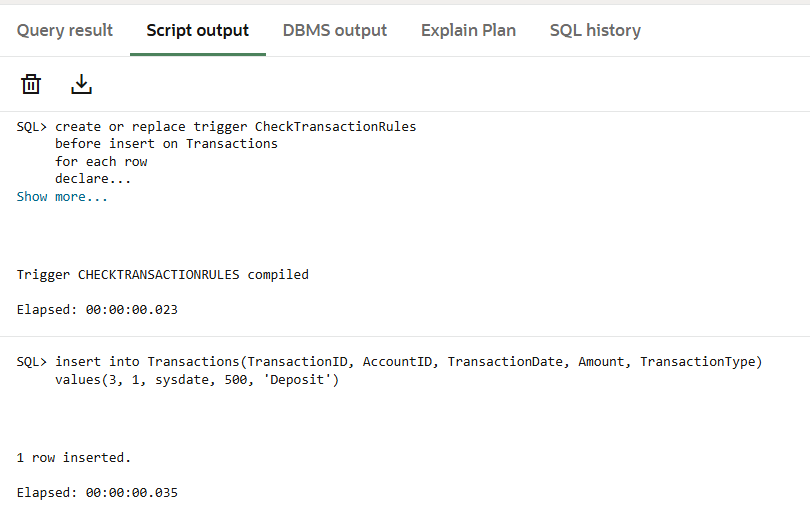
end;

/

--to Test

insert into Transactions(TransactionID, AccountID, TransactionDate, Amount, TransactionType) values(3, 1, sysdate, 500, 'Deposit');

**Output:**

****

**Exercise 6: Cursors**

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

**Code:**

declare cursor cur\_trans is

select t.TransactionID, c.Name, t.AccountID, t.Amount, t.TransactionType, t.TransactionDate

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');

v\_tid Transactions.TransactionID%type;

v\_name Customers.Name%type;

v\_accid Transactions.AccountID%type;

v\_amt Transactions.Amount%type;

v\_type Transactions.TransactionType%type;

v\_date Transactions.TransactionDate%type;

begin

open cur\_trans;

loop

fetch cur\_trans into v\_tid, v\_name, v\_accid, v\_amt, v\_type, v\_date;

exit when cur\_trans%notfound;

dbms\_output.put\_line('Customer: ' || v\_name ||

' | AccID: ' || v\_accid ||

' | Type: ' || v\_type ||

' | Amount: ' || v\_amt ||

' | Date: ' || to\_char(v\_date, 'DD-Mon-YYYY'));

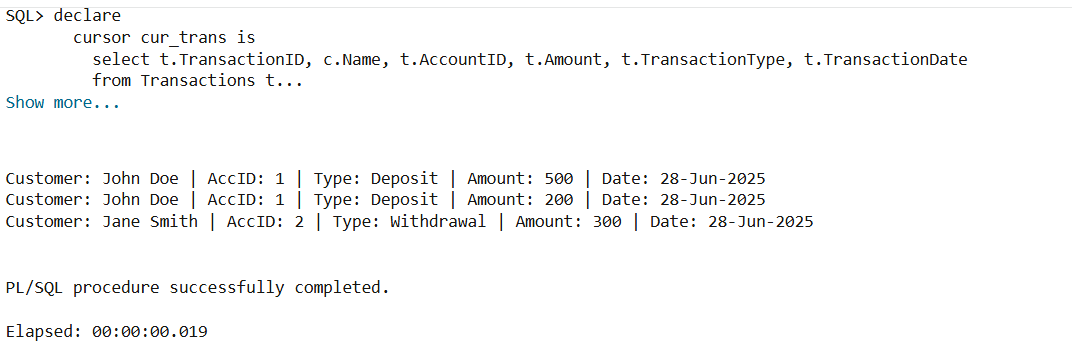
end loop;

close cur\_trans;

end;

/

**Output:**

****

**Scenario 2: Apply annual fee to all accounts.**

**Goal:**

* Use an explicit cursor
* Deduct a fixed annual fee (say ₹100) from every account's balance

**Code:**

declare

cursor cur\_acc is

select AccountID, Balance from Accounts;

v\_accid Accounts.AccountID%type;

v\_bal Accounts.Balance%type;

v\_fee number := 100;

begin

open cur\_acc;

loop

fetch cur\_acc into v\_accid, v\_bal;

exit when cur\_acc%notfound;

update Accounts set Balance = Balance - v\_fee where AccountID = v\_accid;

dbms\_output.put\_line('₹' || v\_fee || ' deducted from Account ID: ' || v\_accid);

end loop;

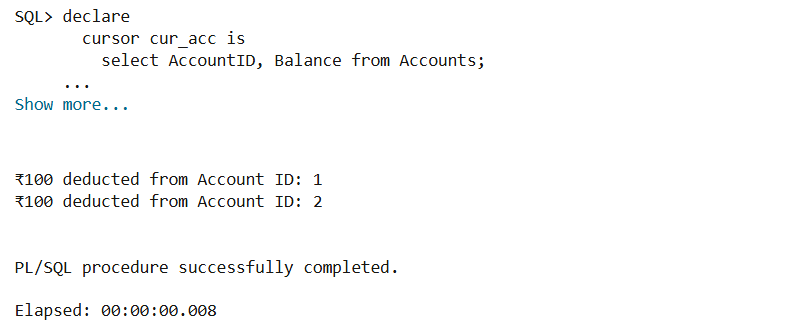
close cur\_acc;

commit;

end;

/

**Output:**

****

**Scenario 3: Update the interest rate for all loans based on a new policy.**

**Goal:**

* Use an explicit cursor
* Apply a new interest rate policy, like:
  + If LoanAmount > 50000, increase rate by 0.5%
  + Else, decrease by 0.25%

**Code:**

declare

cursor cur\_loans is

select LoanID, InterestRate, LoanAmount from Loans;

v\_loanid Loans.LoanID%type;

v\_rate Loans.InterestRate%type;

v\_amt Loans.LoanAmount%type;

begin

open cur\_loans;

loop

fetch cur\_loans into v\_loanid, v\_rate, v\_amt;

exit when cur\_loans%notfound;

if v\_amt > 50000 then

update Loans set InterestRate = v\_rate + 0.5 where LoanID = v\_loanid;

dbms\_output.put\_line('Interest rate increased for Loan ID: ' || v\_loanid);

else

update Loans set InterestRate = v\_rate - 0.25 where LoanID = v\_loanid;

dbms\_output.put\_line('Interest rate decreased for Loan ID: ' || v\_loanid);

end if;

end loop;

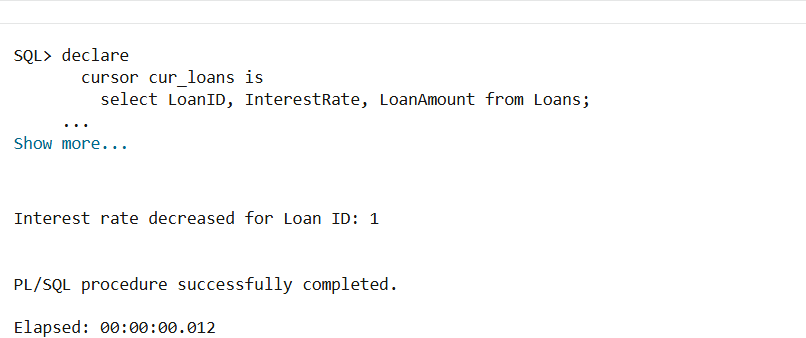
close cur\_loans;

commit;

end;

/

**Output:**



**Exercise 7: Packages**

**Scenario 1:** Group all customer-related procedures and functions into a package.  
**Code:**

--Package Specification

create or replace package CustomerManagement as

procedure AddCustomer(p\_id number, p\_name varchar2, p\_dob date, p\_balance number);

procedure UpdateCustomerDetails(p\_id number, p\_name varchar2);

function GetCustomerBalance(p\_id number) return number;

end CustomerManagement;

/

--Package Body

create 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 UpdateCustomerDetails(p\_id number, p\_name varchar2) is

begin

update Customers set Name = p\_name, 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

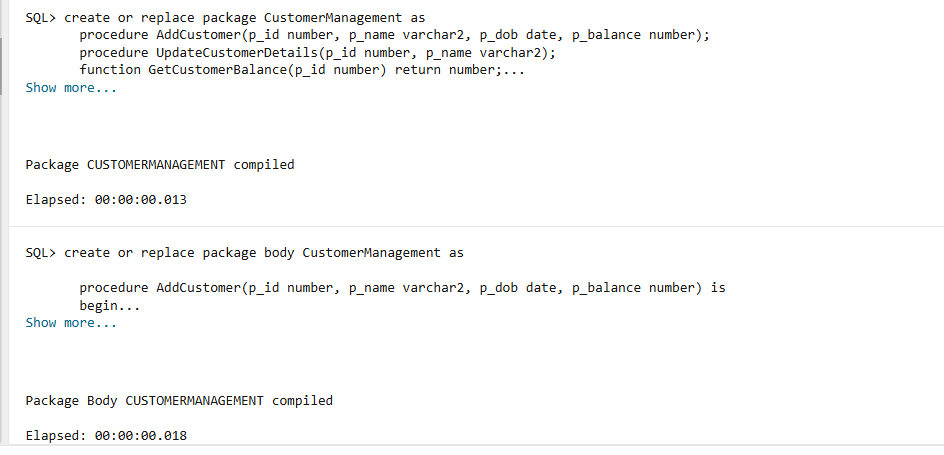
return -1;

end;

end CustomerManagement;

/

**Output:**



**Scenario 2:** Create a package to manage employee data.

**Code:**

--Package Specification

create package EmployeeManagement as

procedure HireEmployee(p\_id number, p\_name varchar2, p\_position varchar2, p\_salary number, p\_dept varchar2, p\_hiredate date);

procedure UpdateEmployeeDetails(p\_id number, p\_name varchar2, p\_position varchar2);

function CalculateAnnualSalary(p\_id number) return number;

end EmployeeManagement;

/

--Packege Body

create or replace package body EmployeeManagement as

procedure HireEmployee(p\_id number, p\_name varchar2, p\_position varchar2, p\_salary number, p\_dept varchar2, p\_hiredate date) is

begin

insert into Employees(EmployeeID, Name, Position, Salary, Department, HireDate)

values(p\_id, p\_name, p\_position, p\_salary, p\_dept, p\_hiredate);

end;

procedure UpdateEmployeeDetails(p\_id number, p\_name varchar2, p\_position varchar2) is

begin

update Employees set Name = p\_name, Position = p\_position 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

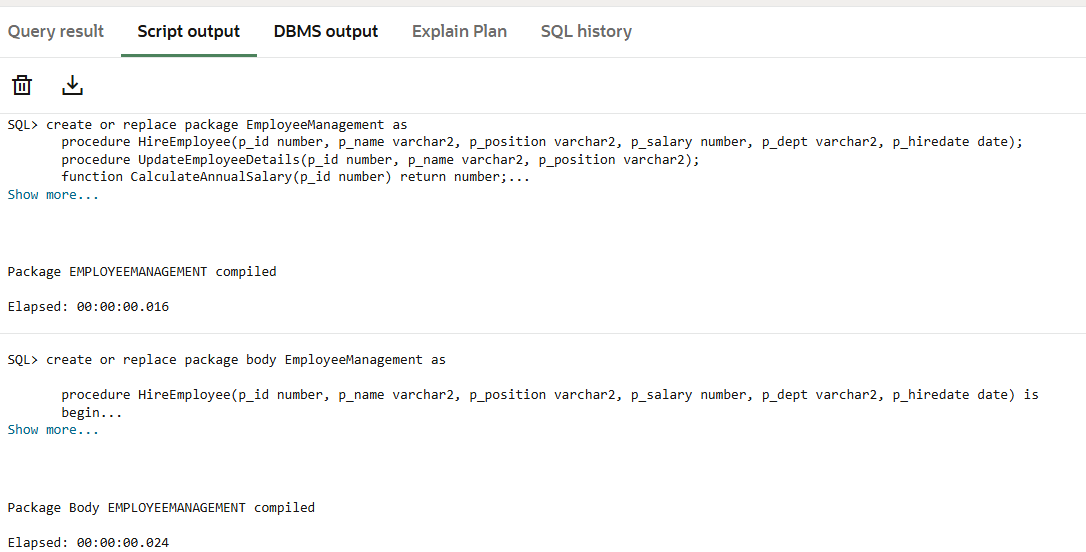
return -1;

end;

end EmployeeManagement;

/

**Output:**

****

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

**Code:**

--Package Specification

create or replace package AccountOperations as

procedure OpenAccount(p\_accid number, p\_custid number, p\_type varchar2, p\_balance number);

procedure CloseAccount(p\_accid number);

function GetTotalBalance(p\_custid number) return number;

end AccountOperations;

/

--Package Body

create or replace package body AccountOperations as

procedure OpenAccount(p\_accid number, p\_custid number, p\_type varchar2, p\_balance number) is

begin

insert into Accounts(AccountID, CustomerID, AccountType, Balance, LastModified)

values(p\_accid, p\_custid, p\_type, p\_balance, sysdate);

end;

procedure CloseAccount(p\_accid number) is

begin

delete from Accounts where AccountID = p\_accid;

end;

function GetTotalBalance(p\_custid number) return number is

v\_total number;

begin

select sum(Balance) into v\_total from Accounts where CustomerID = p\_custid;

return v\_total;

exception

when no\_data\_found then

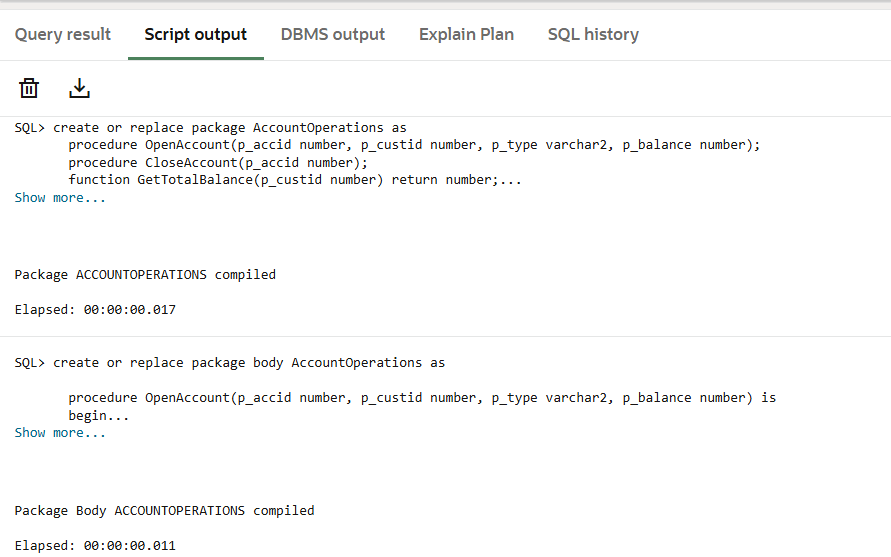
return 0;

end;

end AccountOperations;

/

**Output:**

****