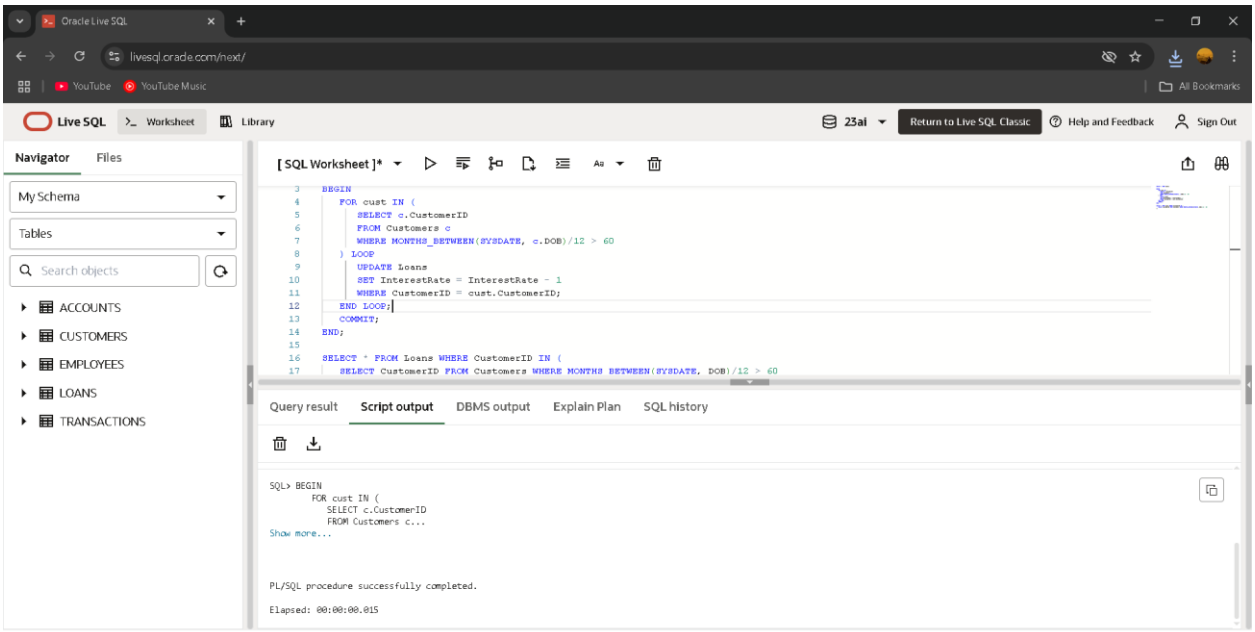
Exercise 1: Control Structures

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

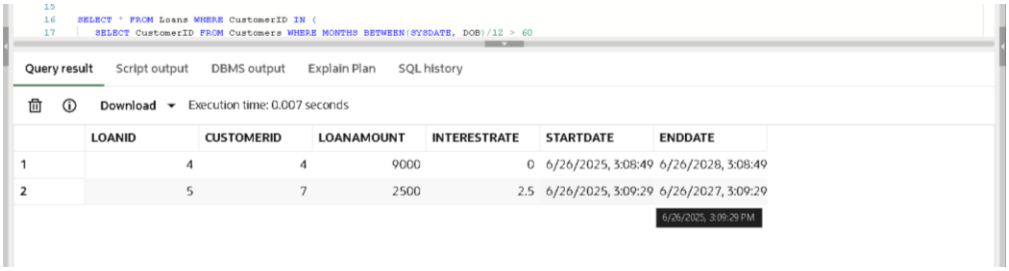
- **Question:** Write a PL/SQL block that loops through all customers, checks their age, and if they are above 60, apply a 1% discount to their current loan interest rates.

Procedure:

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



After applying discount:

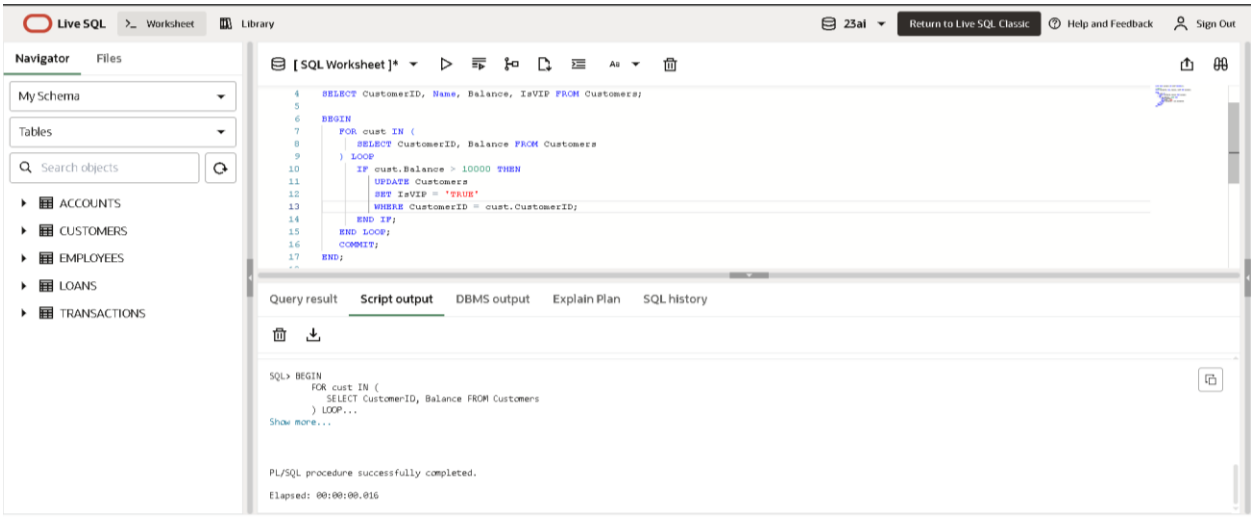


Scenario 2: A customer can be promoted to VIP status based on their balance.

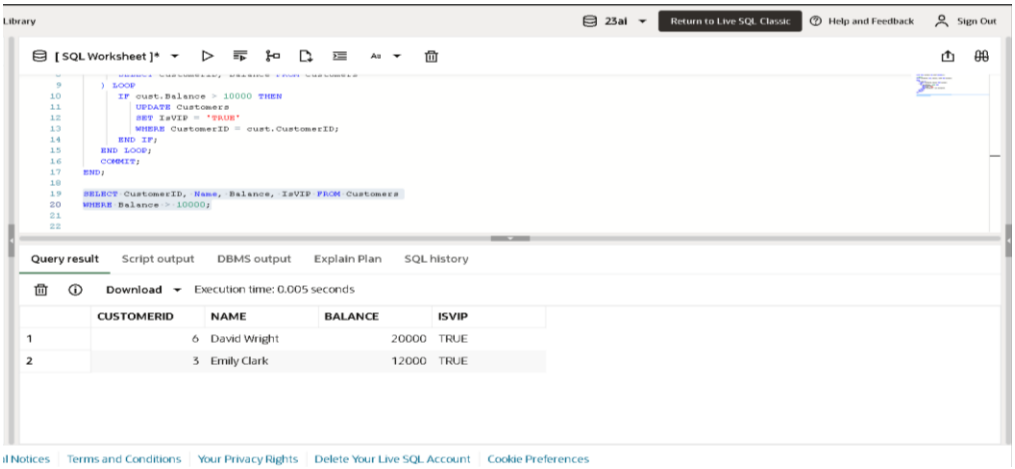
- **Question:** Write a PL/SQL block that iterates through all customers and sets a flag IsVIP to TRUE for those with a balance over $10,000.

Procedure:

```
ALTER TABLE Customers ADD IsVIP VARCHAR2(5);
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;
```



After procedure:

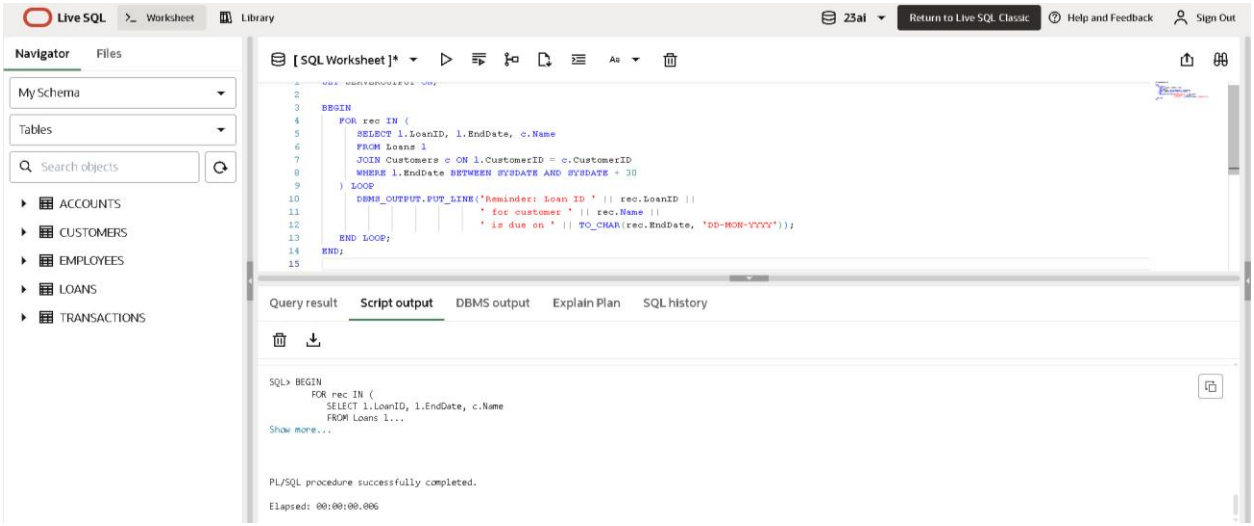


Scenario 3: The bank wants to send reminders to customers whose loans are due within the next 30 days.

- **Question:** Write a PL/SQL block that fetches all loans due in the next 30 days and prints a reminder message for each customer.

Procedure:

```
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 + 30
  ) LOOP
    DBMS_OUTPUT.PUT_LINE('Reminder: Loan ID ' || rec.LoanID ||
                         ' for customer ' || rec.Name ||
                         ' is due on ' || TO_CHAR(rec.EndDate, 'DD-MON-YYYY'));
  END LOOP;
END
```



Exercise 2: Error Handling

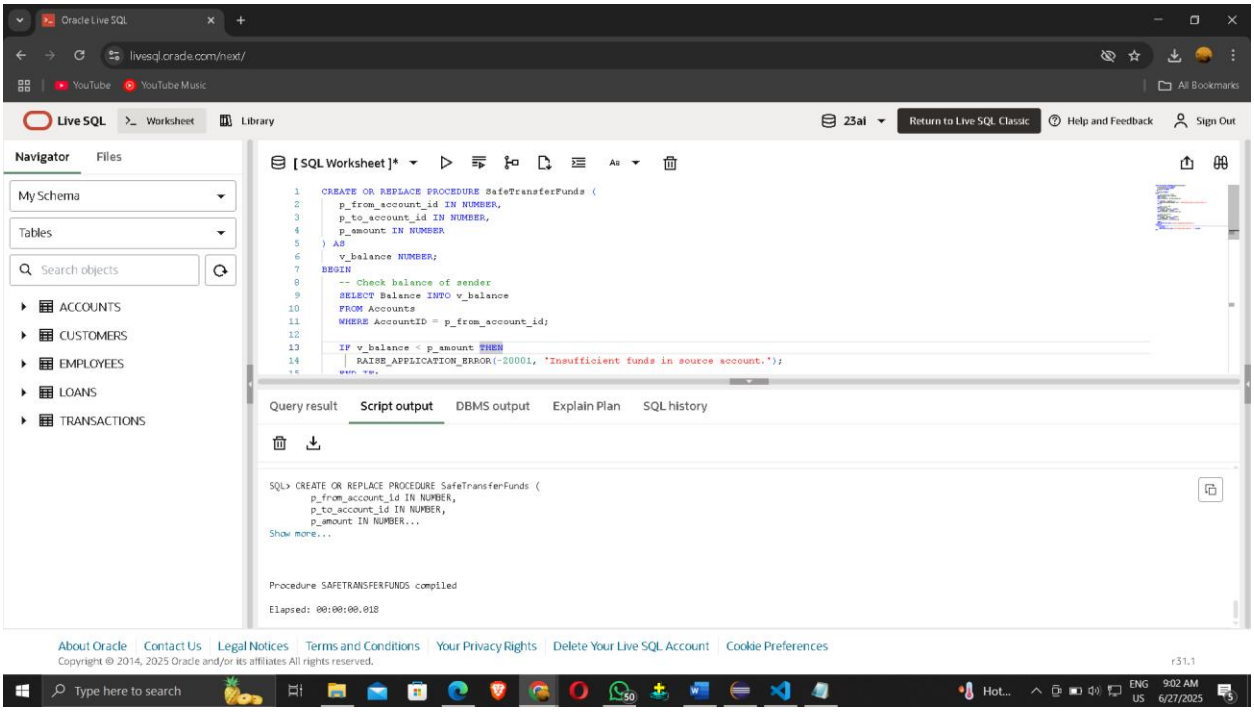
Scenario 1: Handle exceptions during fund transfers between accounts.

- **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.

Procedure:

```
CREATE OR REPLACE PROCEDURE SafeTransferFunds (
  p_from_account_id IN NUMBER,
  p_to_account_id IN NUMBER,
  p_amount IN NUMBER
) AS
```

```
  v_balance NUMBER;
BEGIN
  -- Check balance of sender
  SELECT Balance INTO v_balance
  FROM Accounts
  WHERE AccountID = p_from_account_id;
  IF v_balance < p_amount THEN
    RAISE_APPLICATION_ERROR(-20001, 'Insufficient funds in source account.');
  END IF;
  -- Deduct from sender
  UPDATE Accounts
  SET Balance = Balance - p_amount,
     LastModified = SYSDATE
  WHERE AccountID = p_from_account_id;
  -- Credit to receiver
  UPDATE Accounts
  SET Balance = Balance + p_amount,
     LastModified = SYSDATE
  WHERE AccountID = p_to_account_id;
  COMMIT;
  DBMS_OUTPUT.PUT_LINE('Transfer completed successfully.');
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
    DBMS_OUTPUT.PUT_LINE('Error during fund transfer: ' || SQLERRM);
END;
```

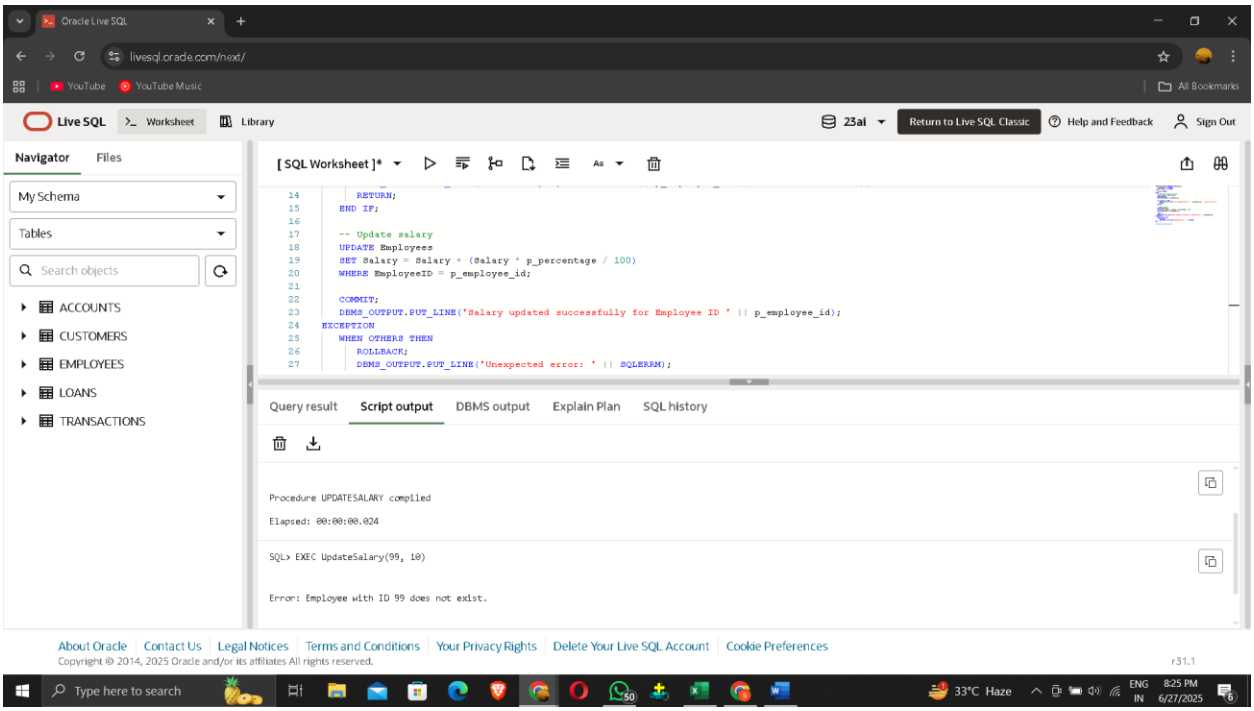


Scenario 2: Manage errors when updating employee salaries.

- **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.

Procedure

```
CREATE OR REPLACE PROCEDURE UpdateSalary (
  p_employee_id IN NUMBER,
  p_percentage   IN NUMBER
) AS
  v_count NUMBER;
BEGIN
  -- Check if the employee exists
  SELECT COUNT(*) INTO v_count
  FROM Employees
  WHERE EmployeeID = p_employee_id;

  IF v_count = 0 THEN
    DBMS_OUTPUT.PUT_LINE('Error: Employee with ID ' || p_employee_id || ' does not exist.');
    RETURN;
  END IF;

  -- Update salary
  UPDATE Employees
  SET Salary = Salary + (Salary * p_percentage / 100)
  WHERE EmployeeID = p_employee_id;

  COMMIT;
  DBMS_OUTPUT.PUT_LINE('Salary updated successfully for Employee ID ' || p_employee_id);
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
    DBMS_OUTPUT.PUT_LINE('Unexpected error: ' || SQLERRM);
END;

EXEC UpdateSalary(99, 10);
```



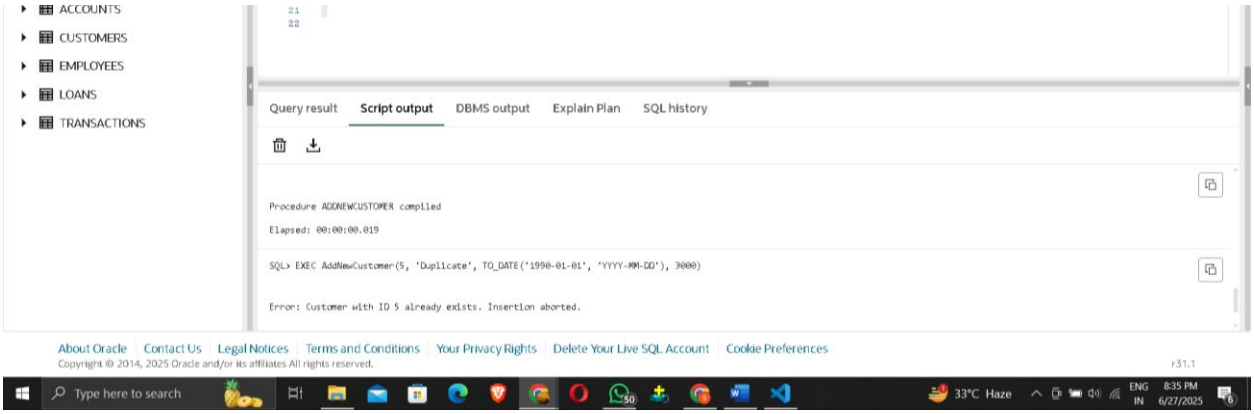
Scenario 3: Ensure data integrity when adding a new customer.

- **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.

Procedure:

```
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 inserted successfully. ID: ' || p_customer_id);
EXCEPTION
  WHEN DUP_VAL_ON_INDEX THEN
    DBMS_OUTPUT.PUT_LINE('Error: Customer with ID ' || p_customer_id || ' already exists. Insertion
aborted.');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Unexpected error occurred: ' || SQLERRM);
END;
```

EXEC AddNewCustomer(5, 'Duplicate', TO_DATE('1990-01-01', 'YYYY-MM-DD'), 3000);



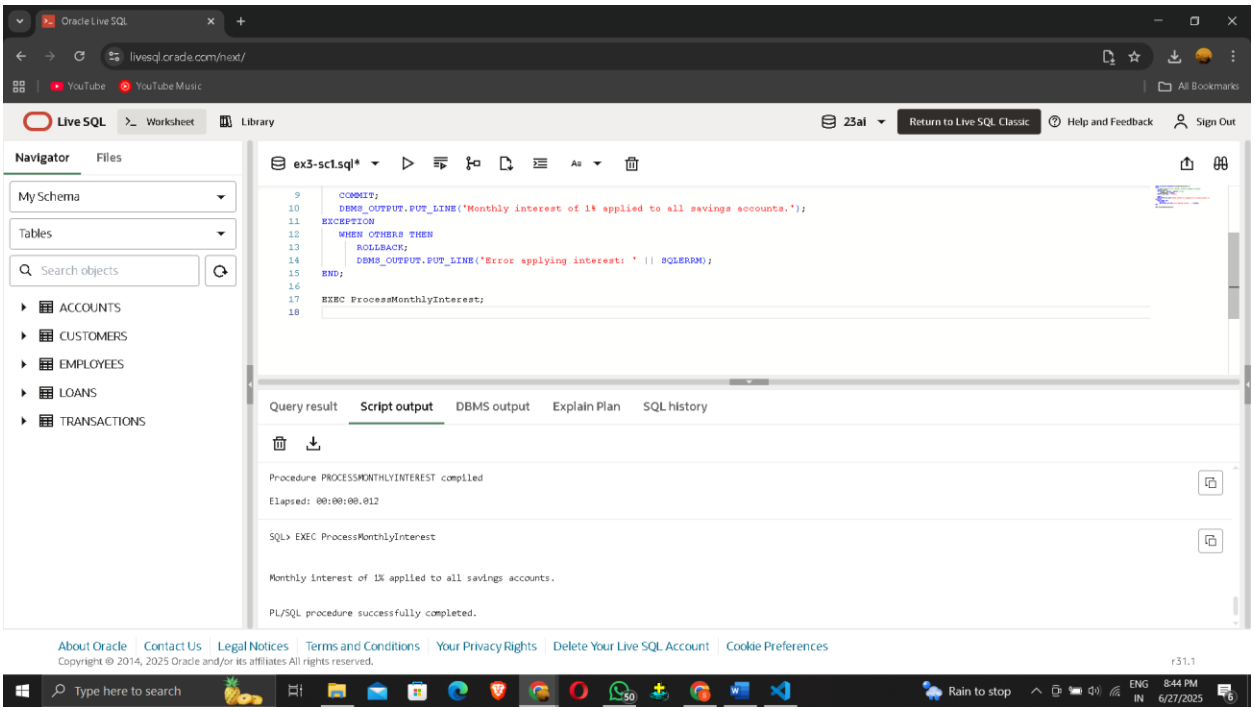
Exercise 3: Stored Procedures

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

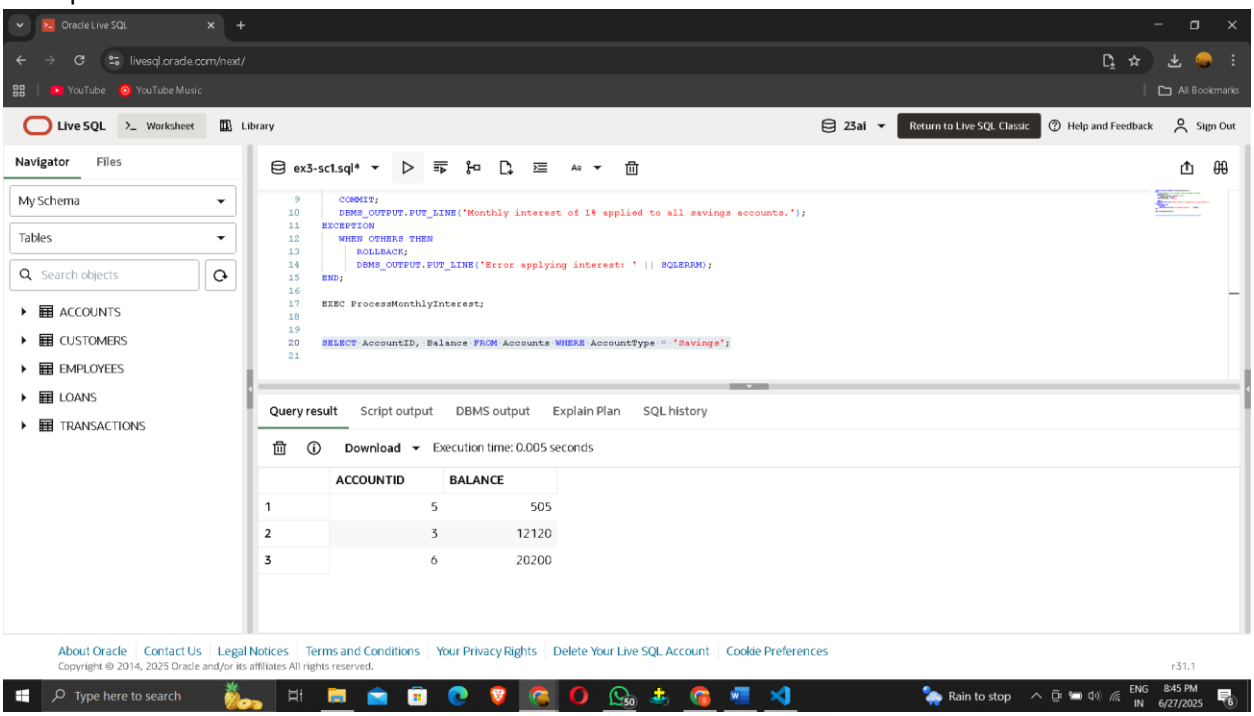
- **Question:** Write a stored procedure **ProcessMonthlyInterest** that calculates and updates the balance of all savings accounts by applying an interest rate of 1% to the current balance.

Procedure

```
CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS
BEGIN
  -- Update balance for all 'Savings' accounts by adding 1% interest
  UPDATE Accounts
  SET Balance = Balance + (Balance * 0.01),
     LastModified = SYSDATE
  WHERE AccountType = 'Savings';

  COMMIT;
  DBMS_OUTPUT.PUT_LINE('Monthly interest of 1% applied to all savings accounts.');
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
    DBMS_OUTPUT.PUT_LINE('Error applying interest: ' || SQLERRM);
END;

EXEC ProcessMonthlyInterest;

--to see whether it has been applied
SELECT AccountID, Balance FROM Accounts WHERE AccountType = 'Savings';
```



After procedure schema looks like



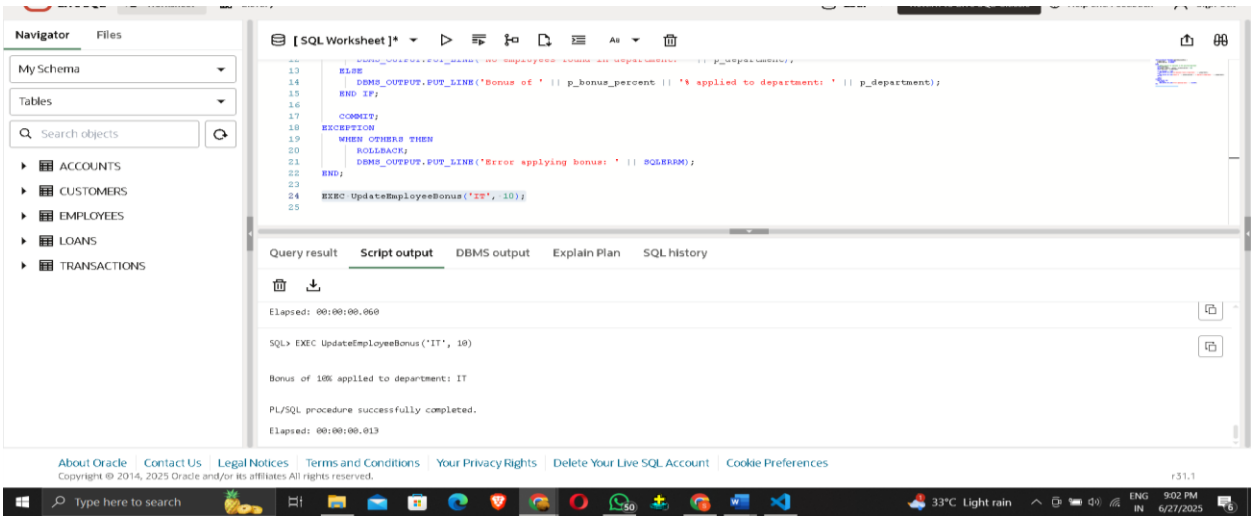
Scenario 2: The bank wants to implement a bonus scheme for employees based on their performance.

- **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.

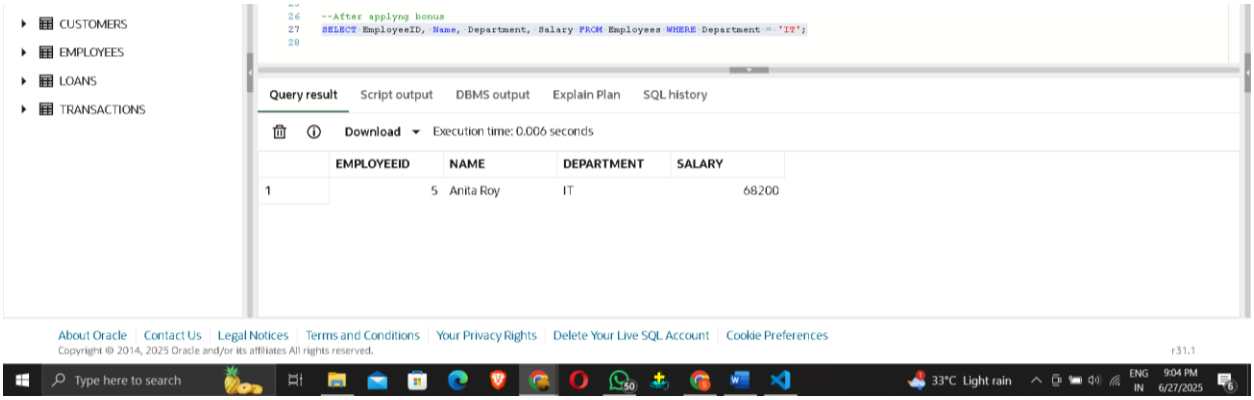
Procedure:

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (

```
  p_department     IN VARCHAR2,
  p_bonus_percent  IN NUMBER
) AS
BEGIN
  -- Update salaries for employees in the specified department
  UPDATE Employees
  SET Salary = Salary + (Salary * p_bonus_percent / 100)
  WHERE Department = p_department;
  IF SQL%ROWCOUNT = 0 THEN
    DBMS_OUTPUT.PUT_LINE('No employees found in department: ' || p_department);
  ELSE
    DBMS_OUTPUT.PUT_LINE('Bonus of ' || p_bonus_percent || '% applied to department: ' ||
p_department);
  END IF;

  COMMIT;
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
    DBMS_OUTPUT.PUT_LINE('Error applying bonus: ' || SQLERRM);
END;
EXEC UpdateEmployeeBonus('IT', 10);
--After applyng bonus
SELECT EmployeeID, Name, Department, Salary FROM Employees WHERE Department = 'IT';
```



Schema after execution looks like



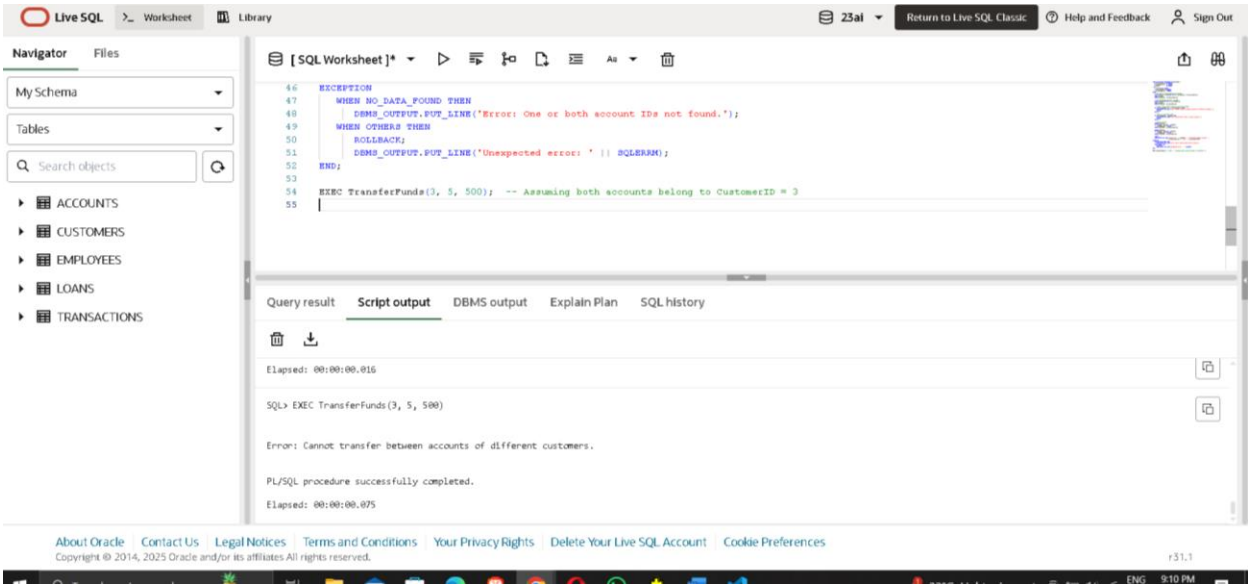
Scenario 3: Customers should be able to transfer funds between their accounts.

- **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.

Procedure:

```
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;
  v_from_customer_id NUMBER;
  v_to_customer_id   NUMBER;
BEGIN
  -- Get source account balance and customer
  SELECT Balance, CustomerID INTO v_from_balance, v_from_customer_id
  FROM Accounts
  WHERE AccountID = p_from_account_id;
  -- Get destination account's customer
  SELECT CustomerID INTO v_to_customer_id
  FROM Accounts
  WHERE AccountID = p_to_account_id;
  -- Check if both accounts belong to the same customer
  IF v_from_customer_id != v_to_customer_id THEN
    DBMS_OUTPUT.PUT_LINE('Error: Cannot transfer between accounts of different customers.');
    RETURN;
  END IF;
  -- Check for sufficient funds
  IF v_from_balance < p_amount THEN
    DBMS_OUTPUT.PUT_LINE('Error: Insufficient funds in source account.');
    RETURN;
  END IF;
  -- Perform transfer
```

```
  UPDATE Accounts
  SET Balance = Balance - p_amount,
      LastModified = SYSDATE
  WHERE AccountID = p_from_account_id;
  UPDATE Accounts
  SET Balance = Balance + p_amount,
      LastModified = SYSDATE
  WHERE AccountID = p_to_account_id;
  COMMIT;
  DBMS_OUTPUT.PUT_LINE('Transfer of ' || p_amount || ' successful between accounts ' ||
                       p_from_account_id || ' and ' || p_to_account_id);
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('Error: One or both account IDs not found.');
  WHEN OTHERS THEN
    ROLLBACK;
    DBMS_OUTPUT.PUT_LINE('Unexpected error: ' || SQLERRM);
END;

EXEC TransferFunds(3, 5, 500);  -- Assuming both accounts belong to CustomerID = 3
```



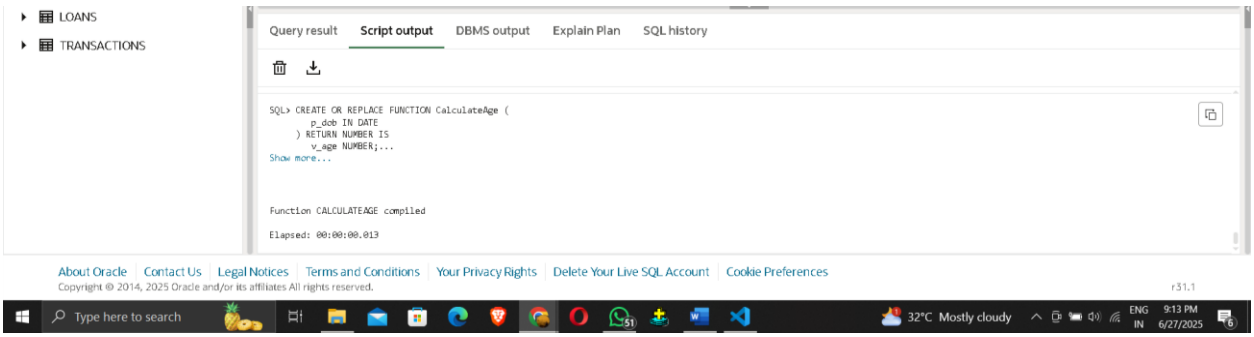
Exercise 4: Functions

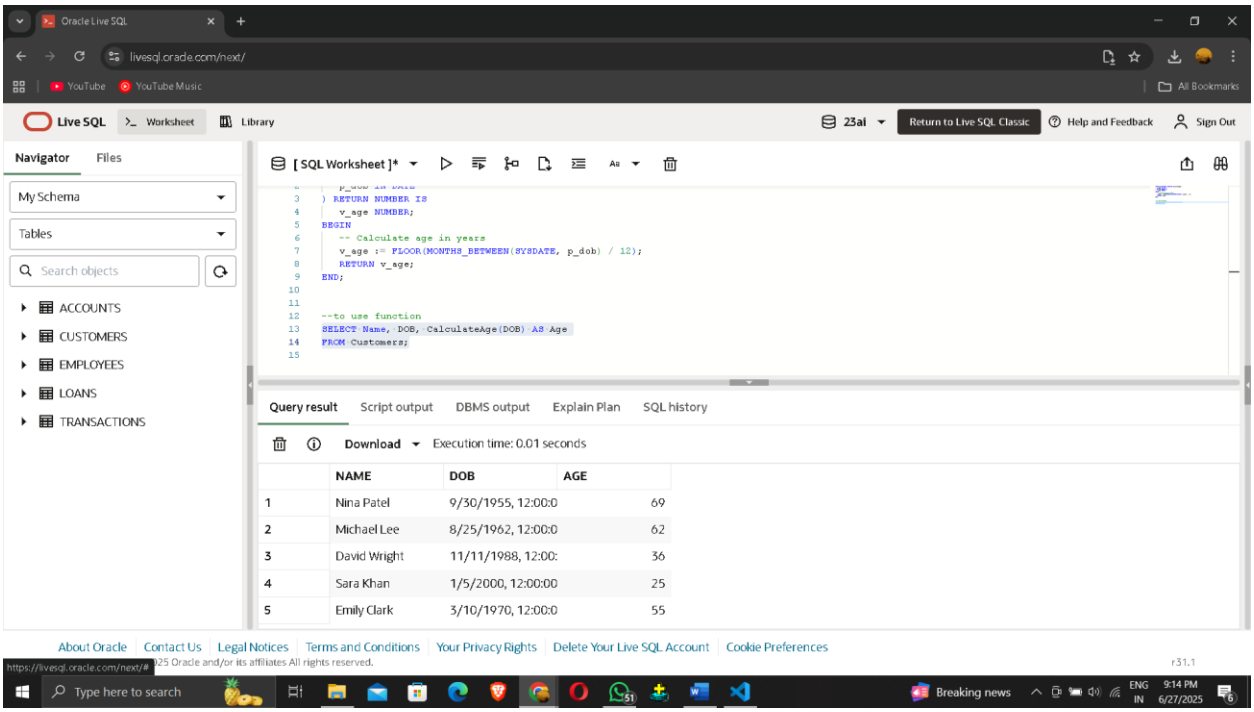
Scenario 1: Calculate the age of customers for eligibility checks.

- **Question:** Write a function CalculateAge that takes a customer's date of birth as input and returns their age in years

Procedure

```
CREATE OR REPLACE FUNCTION CalculateAge (
  p_dob IN DATE
) RETURN NUMBER IS
  v_age NUMBER;
BEGIN
  -- Calculate age in years
  v_age := FLOOR(MONTHS_BETWEEN(SYSDATE, p_dob) / 12);
  RETURN v_age;
END;

--to use function
SELECT Name, DOB, CalculateAge(DOB) AS Age
FROM Customers;
```



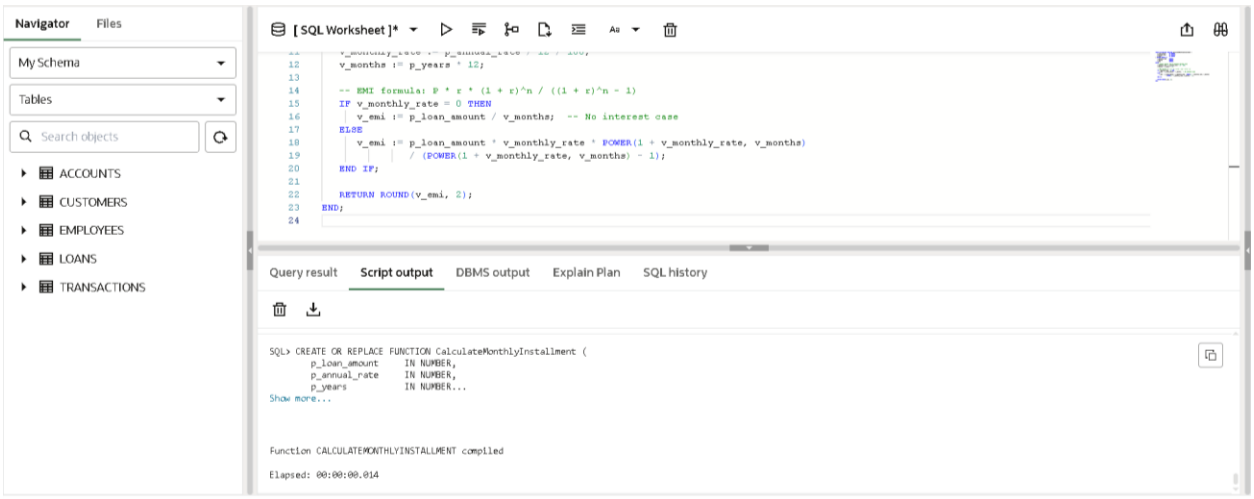


Scenario 2: The bank needs to compute the monthly installment for a loan.

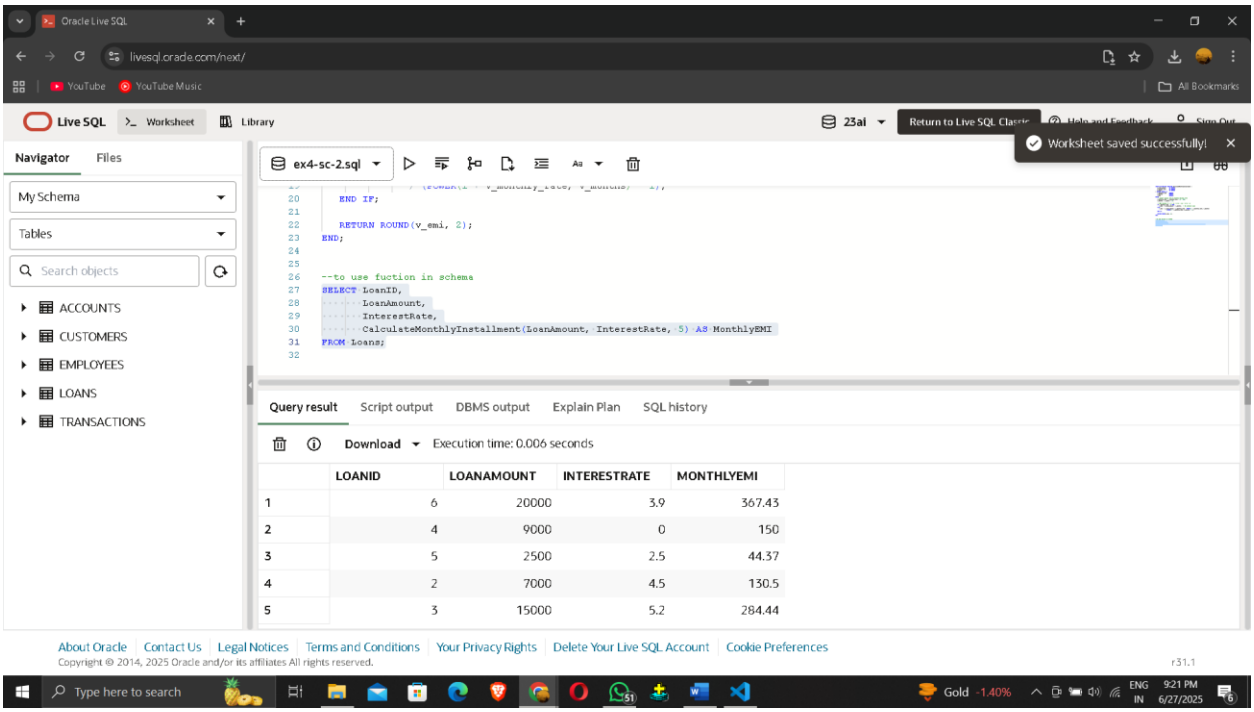
- **Question:** Write a function **CalculateMonthlyInstallment** that takes the loan amount, interest rate, and loan duration in years as input and returns the monthly installment amount

Procedure:

```
CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment (
  p_loan_amount     IN NUMBER,
  p_annual_rate     IN NUMBER,
  p_years           IN NUMBER
) RETURN NUMBER IS
  v_monthly_rate    NUMBER;
  v_months          NUMBER;
  v_emi             NUMBER;
BEGIN
  -- Convert annual rate to monthly decimal rate
  v_monthly_rate := p_annual_rate / 12 / 100;
  v_months := p_years * 12;

  -- EMI formula: P * r * (1 + r)^n / ((1 + r)^n - 1)
  IF v_monthly_rate = 0 THEN
    v_emi := p_loan_amount / v_months;  -- No interest case
  ELSE
    v_emi := p_loan_amount * v_monthly_rate * POWER(1 + v_monthly_rate, v_months)
          / (POWER(1 + v_monthly_rate, v_months) - 1);
  END IF;

  RETURN ROUND(v_emi, 2);
END;

--to use fuction in schema
SELECT LoanID,
     LoanAmount,
     InterestRate,
     CalculateMonthlyInstallment(LoanAmount, InterestRate, 5) AS MonthlyEMI
FROM Loans;
```



Schema:



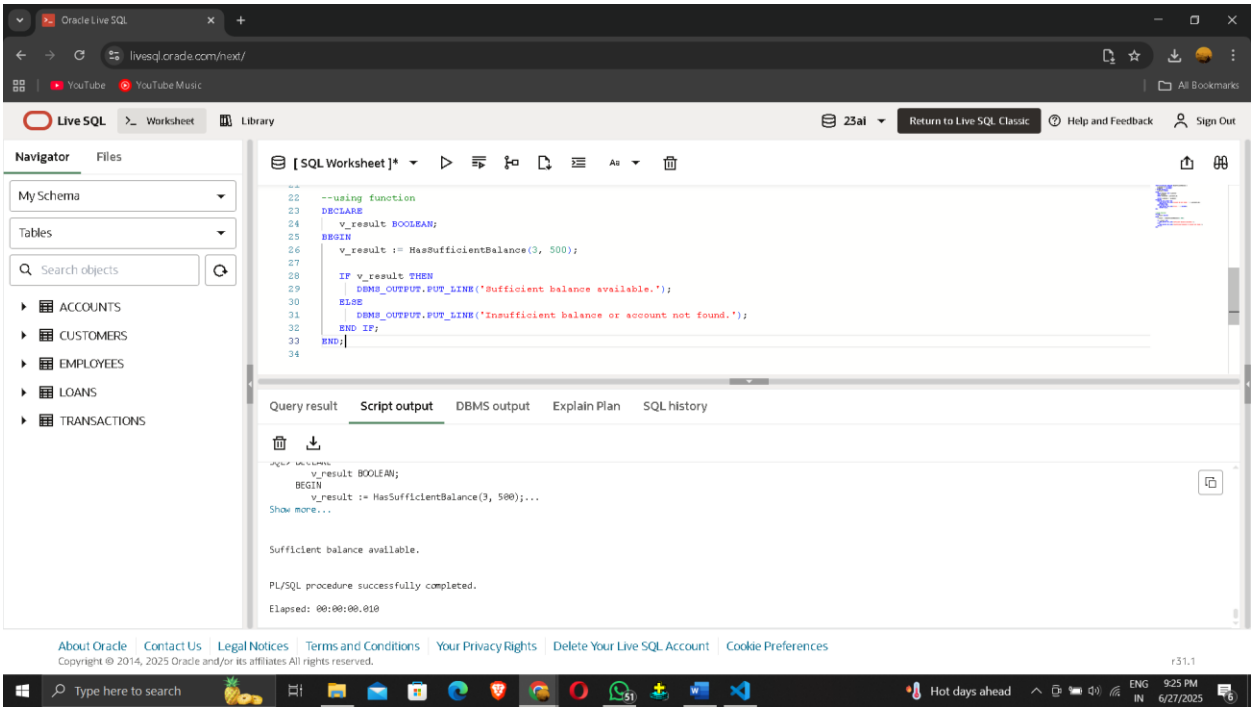
Scenario 3: Check if a customer has sufficient balance before making a transaction.

- **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.

Procedure:

```
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
    DBMS_OUTPUT.PUT_LINE('Account ID not found: ' || p_account_id);
    RETURN FALSE;
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
    RETURN FALSE;
END;

--using this func
DECLARE
  v_result BOOLEAN;
BEGIN
  v_result := HasSufficientBalance(3, 500);

  IF v_result THEN
    DBMS_OUTPUT.PUT_LINE('Sufficient balance available.');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Insufficient balance or account not found.');
  END IF;
END;
```



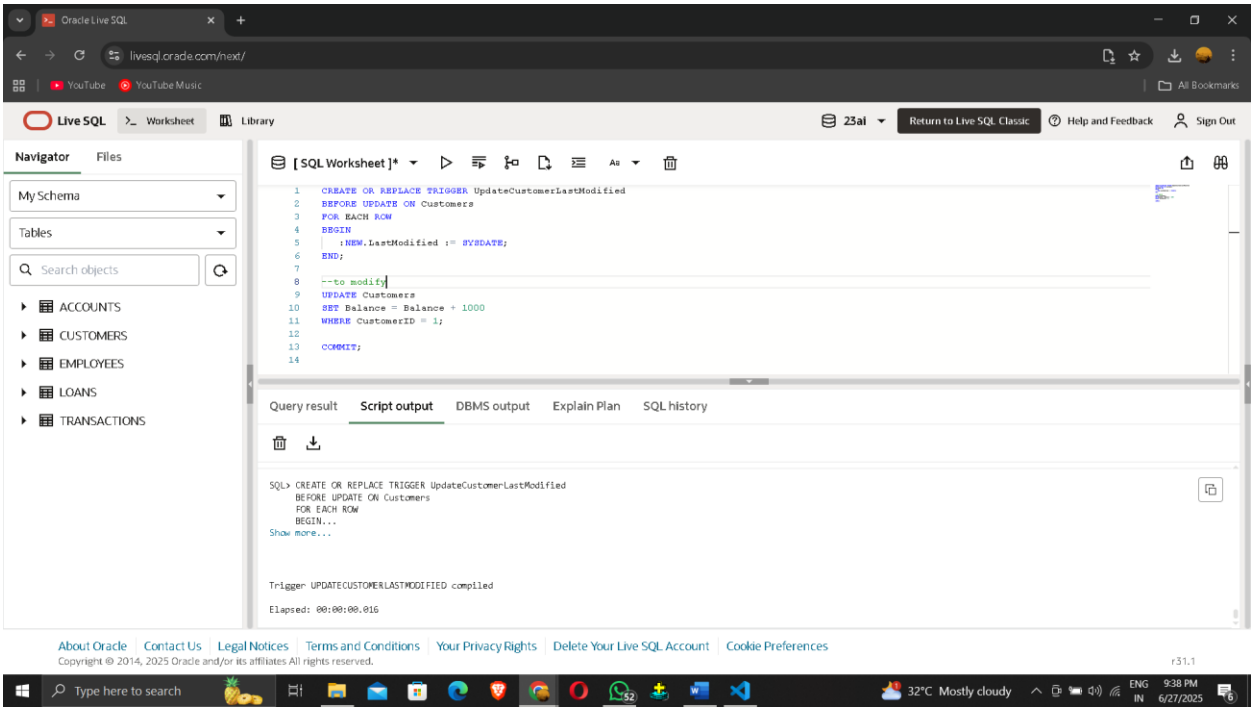
Exercise 5: Triggers

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

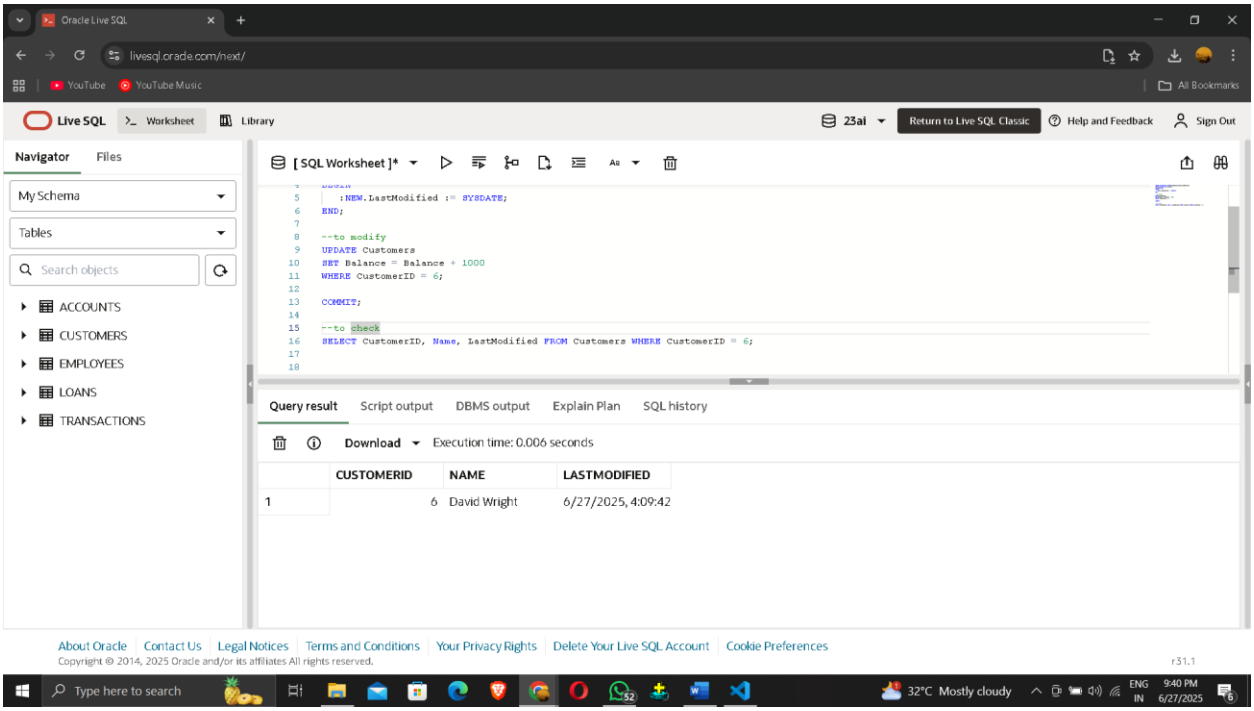
- **Question:** Write a trigger **UpdateCustomerLastModified** that updates the LastModified column of the Customers table to the current date whenever a customer's record is updated.

Procedure:

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

--to modify
UPDATE Customers
SET Balance = Balance + 1000
WHERE CustomerID = 6;

COMMIT;

--to check
SELECT CustomerID, Name, LastModified FROM Customers WHERE CustomerID = 6;
```



After modification



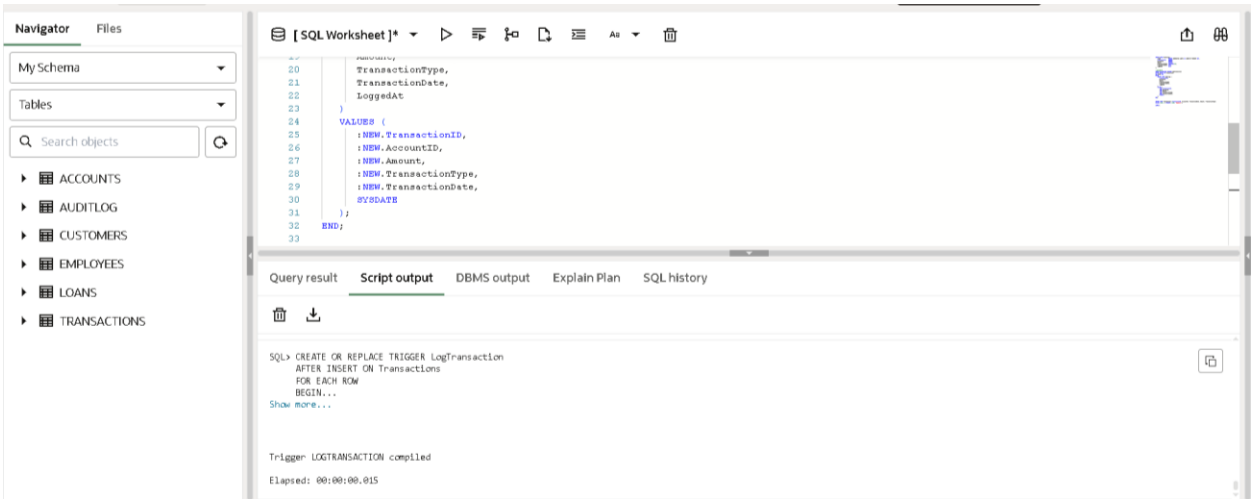
Scenario 2: Maintain an audit log for all transactions.

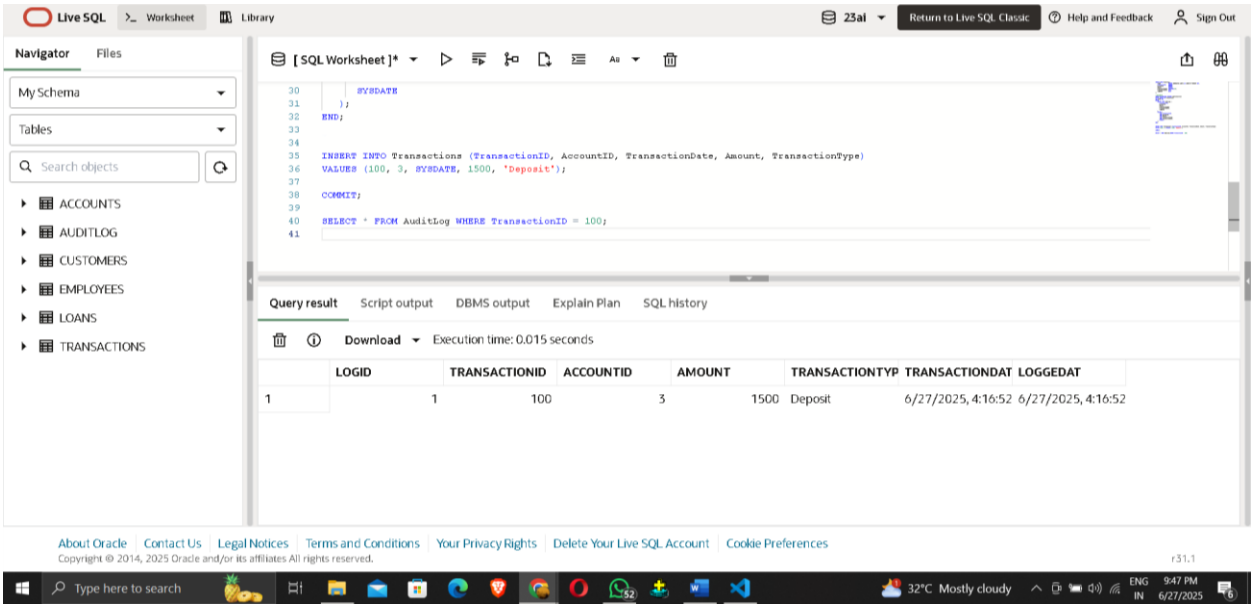
- **Question:** Write a trigger **LogTransaction** that inserts a record into an AuditLog table whenever a transaction is inserted into the Transactions table.

Procedure

```
--to create audit table
CREATE TABLE AuditLog (
  LogID          NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  TransactionID    NUMBER,
  AccountID       NUMBER,
  Amount          NUMBER,
  TransactionType  VARCHAR2(10),
  TransactionDate  DATE,
  LoggedAt        DATE
);

CREATE OR REPLACE TRIGGER LogTransaction
AFTER INSERT ON Transactions
FOR EACH ROW
BEGIN
  INSERT INTO AuditLog (
    TransactionID,
    AccountID,
    Amount,
    TransactionType,
```

```
    TransactionDate,
    LoggedAt
  )
  VALUES (
    :NEW.TransactionID,
    :NEW.AccountID,
    :NEW.Amount,
    :NEW.TransactionType,
    :NEW.TransactionDate,
    SYSDATE
  );
END;

--insert values

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)
VALUES (100, 3, SYSDATE, 1500, 'Deposit');

COMMIT;


--display audit
SELECT * FROM AuditLog WHERE TransactionID = 100;
```





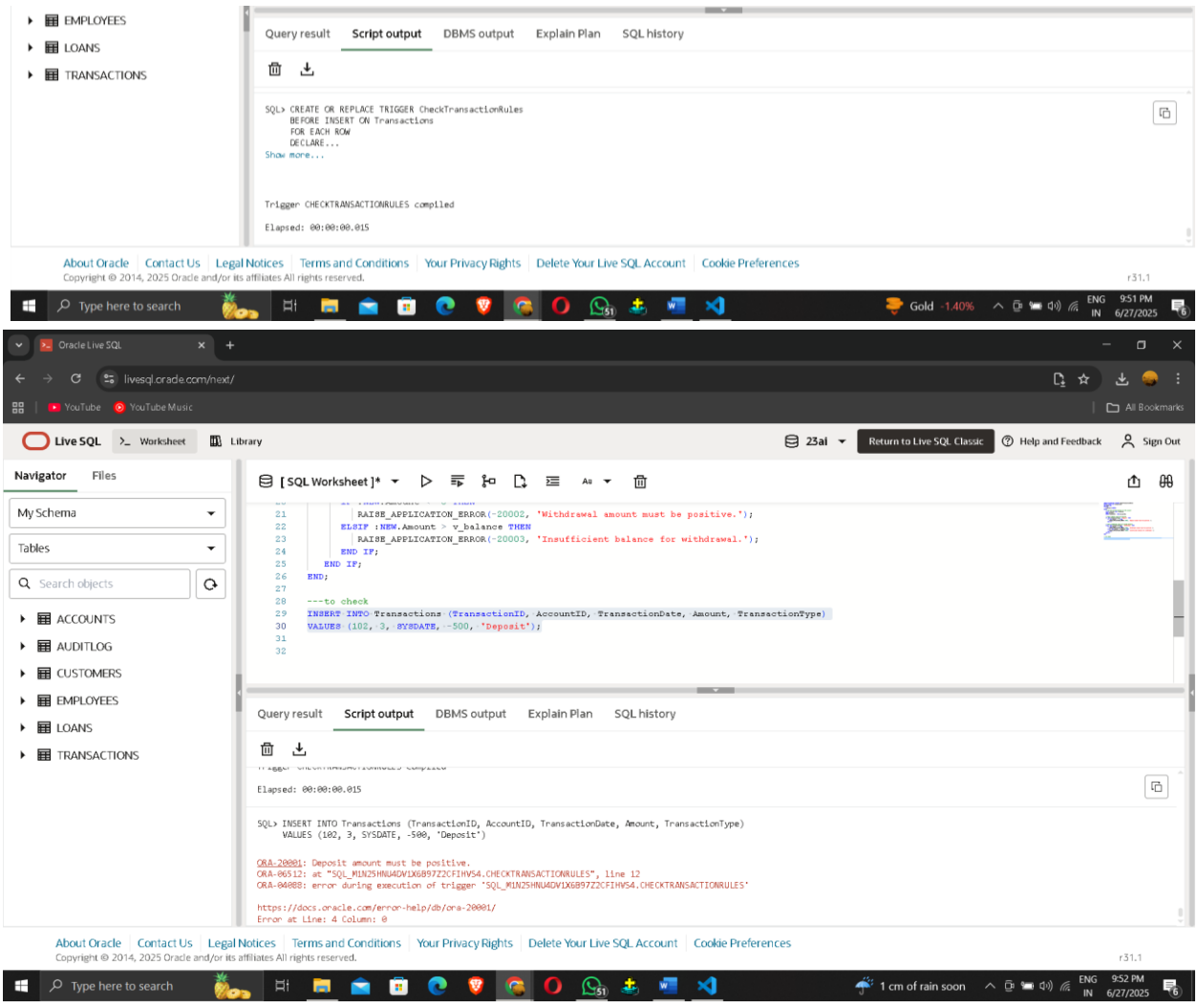
Scenario 3: Enforce business rules on deposits and withdrawals.

- **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.

Procedure

```
CREATE OR REPLACE TRIGGER CheckTransactionRules
BEFORE INSERT ON Transactions
FOR EACH ROW
DECLARE
  v_balance NUMBER;
BEGIN
  -- Get the current balance of the account
  SELECT Balance INTO v_balance
  FROM Accounts
  WHERE AccountID = :NEW.AccountID;

  -- Rule: Deposit must be positive
  IF :NEW.TransactionType = 'Deposit' THEN
    IF :NEW.Amount <= 0 THEN
      RAISE_APPLICATION_ERROR(-20001, 'Deposit amount must be positive.');
    END IF;

  -- Rule: Withdrawal must not exceed balance
  ELSIF :NEW.TransactionType = 'Withdrawal' THEN
    IF :NEW.Amount <= 0 THEN
      RAISE_APPLICATION_ERROR(-20002, 'Withdrawal amount must be positive.');
    ELSIF :NEW.Amount > v_balance THEN
      RAISE_APPLICATION_ERROR(-20003, 'Insufficient balance for withdrawal.');
```

```
    END IF;
  END IF;
END;

---to check
INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)
VALUES (102, 3, SYSDATE, -500, 'Deposit');
```



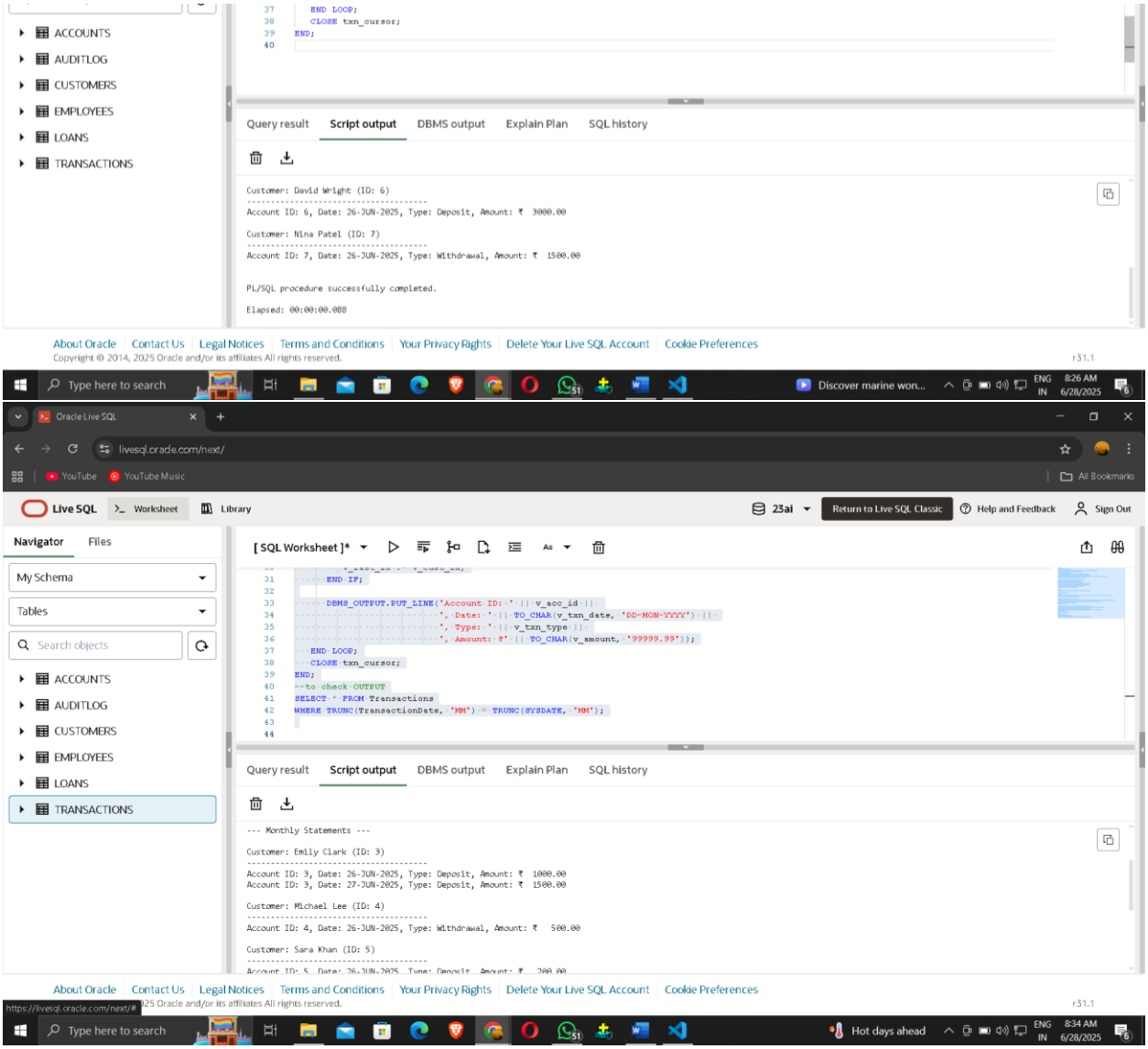
Exercise 6: Cursors

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.

procedure.

```
DECLARE
  CURSOR txn_cursor IS
    SELECT c.CustomerID, c.Name, a.AccountID, t.TransactionDate,
         t.TransactionType, t.Amount
    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')  -- Current month
    ORDER BY c.CustomerID, t.TransactionDate;

  v_cust_id       Customers.CustomerID%TYPE;
  v_name          Customers.Name%TYPE;
  v_acc_id        Accounts.AccountID%TYPE;
  v_txn_date      Transactions.TransactionDate%TYPE;
  v_txn_type      Transactions.TransactionType%TYPE;
  v_amount        Transactions.Amount%TYPE;
  v_last_id       NUMBER := -1;

BEGIN
  DBMS_OUTPUT.PUT_LINE('--- Monthly Statements ---');

  OPEN txn_cursor;
  LOOP
    FETCH txn_cursor INTO v_cust_id, v_name, v_acc_id, v_txn_date, v_txn_type, v_amount;
    EXIT WHEN txn_cursor%NOTFOUND;

    IF v_cust_id != v_last_id THEN
      DBMS_OUTPUT.PUT_LINE(CHR(10) || 'Customer: ' || v_name || ' (ID: ' || v_cust_id || ')');
      DBMS_OUTPUT.PUT_LINE('---------------------------------------');
      v_last_id := v_cust_id;
    END IF;

    DBMS_OUTPUT.PUT_LINE('Account ID: ' || v_acc_id ||
                  ', Date: ' || TO_CHAR(v_txn_date, 'DD-MON-YYYY') ||
                  ', Type: ' || v_txn_type ||
                  ', Amount: ₹' || TO_CHAR(v_amount, '99999.99'));
  END LOOP;
  CLOSE txn_cursor;
END;
--to check OUTPUT
SELECT * FROM Transactions
WHERE TRUNC(TransactionDate, 'MM') = TRUNC(SYSDATE, 'MM');
```



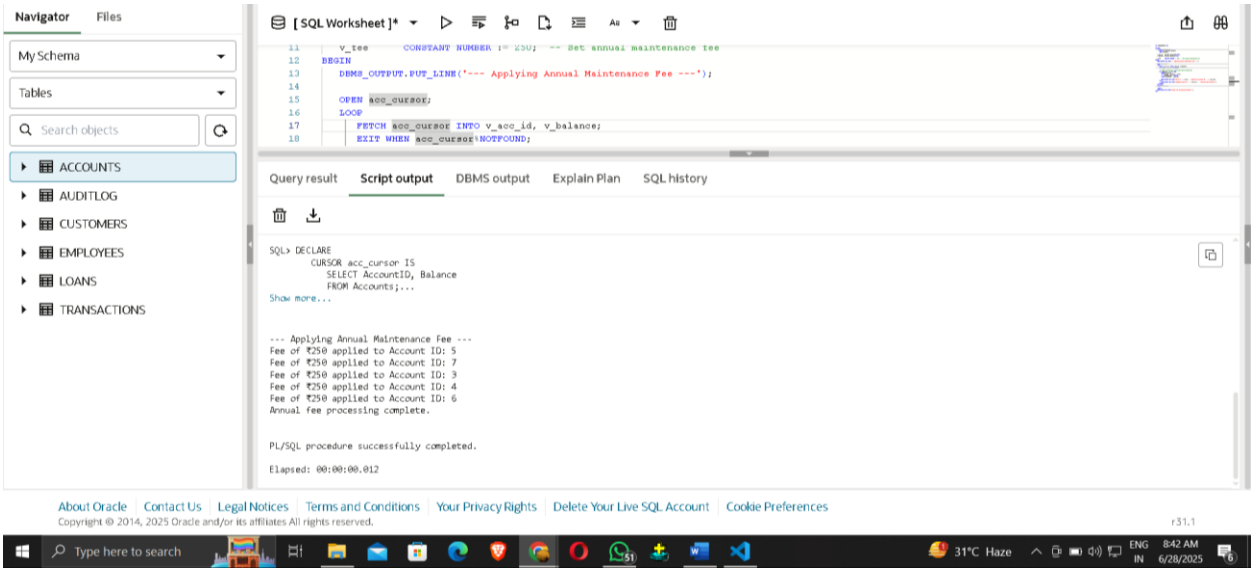
Scenario 2: Apply annual fee to all accounts.

- **Question:** Write a PL/SQL block using an explicit cursor **ApplyAnnualFee** that deducts an annual maintenance fee from the balance of all accounts

Procedure

```
SET SERVEROUTPUT ON;

DECLARE
  CURSOR acc_cursor IS
    SELECT AccountID, Balance
    FROM Accounts;

  v_acc_id   Accounts.AccountID%TYPE;
  v_balance  Accounts.Balance%TYPE;
```

```
  v_fee      CONSTANT NUMBER := 250;  -- Set annual maintenance fee
BEGIN
  DBMS_OUTPUT.PUT_LINE('--- Applying Annual Maintenance Fee ---');

  OPEN acc_cursor;
  LOOP
    FETCH acc_cursor INTO v_acc_id, v_balance;
    EXIT WHEN acc_cursor%NOTFOUND;

    -- Check if balance is sufficient before deducting
    IF v_balance >= v_fee THEN
      UPDATE Accounts
      SET Balance = Balance - v_fee,
          LastModified = SYSDATE
      WHERE AccountID = v_acc_id;

      DBMS_OUTPUT.PUT_LINE('Fee of ₹' || v_fee || ' applied to Account ID: ' || v_acc_id);
    ELSE
      DBMS_OUTPUT.PUT_LINE('Skipped Account ID ' || v_acc_id || ' (Insufficient balance)');
    END IF;
  END LOOP;
  CLOSE acc_cursor;

  COMMIT;
  DBMS_OUTPUT.PUT_LINE('Annual fee processing complete.');
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.

Procedure:

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

  v_loan_id     Loans.LoanID%TYPE;
  v_amount      Loans.LoanAmount%TYPE;
  v_old_rate    Loans.InterestRate%TYPE;
  v_new_rate    NUMBER;
BEGIN
  DBMS_OUTPUT.PUT_LINE('--- Updating Loan Interest Rates Based on New Policy ---');

  OPEN loan_cursor;
  LOOP
    FETCH loan_cursor INTO v_loan_id, v_amount, v_old_rate;
    EXIT WHEN loan_cursor%NOTFOUND;

    -- Apply new interest rate policy
    IF v_amount < 10000 THEN
      v_new_rate := 6;
    ELSIF v_amount BETWEEN 10000 AND 50000 THEN
      v_new_rate := 5;
    ELSE
      v_new_rate := 4.5;
    END IF;

    -- Update only if rate has changed
    IF v_new_rate != v_old_rate THEN
      UPDATE Loans
      SET InterestRate = v_new_rate
      WHERE LoanID = v_loan_id;

      DBMS_OUTPUT.PUT_LINE('Loan ID: ' || v_loan_id ||
                           ' | Old Rate: ' || v_old_rate || '%' ||
                           ' → New Rate: ' || v_new_rate || '%');
    END IF;
  END LOOP;
  CLOSE loan_cursor;

  COMMIT;
```

```
   DBMS_OUTPUT.PUT_LINE('Interest rate update complete.');
END;

--test procdure and display updated loan
SELECT LoanID, LoanAmount, InterestRate FROM Loans;
```

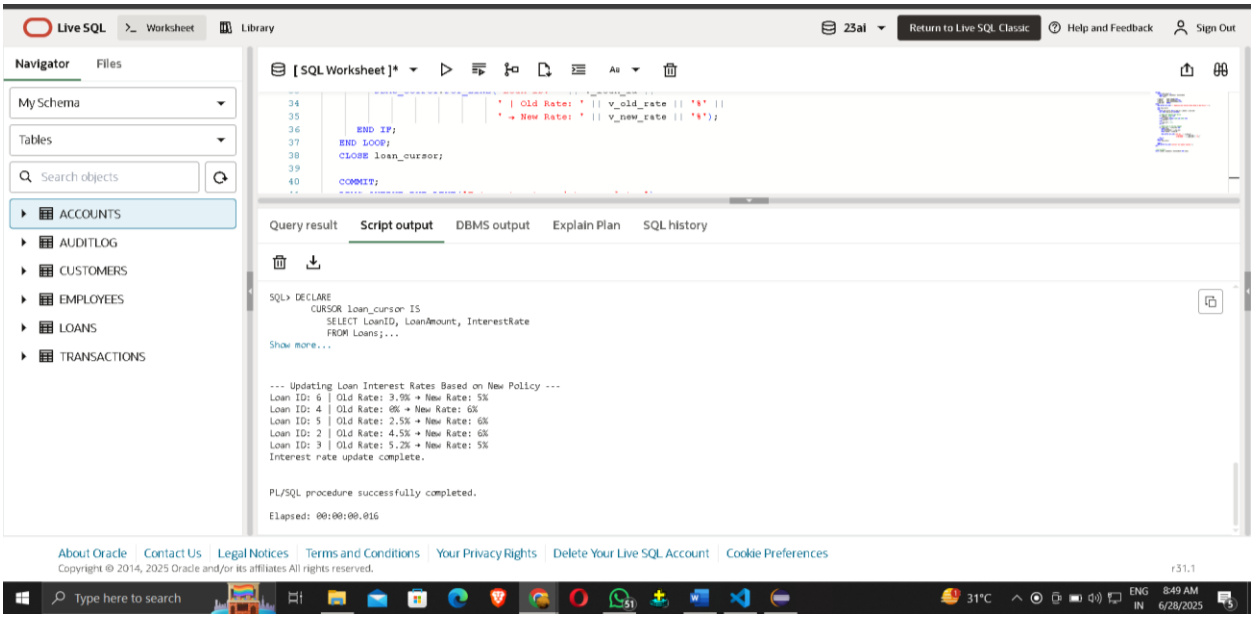
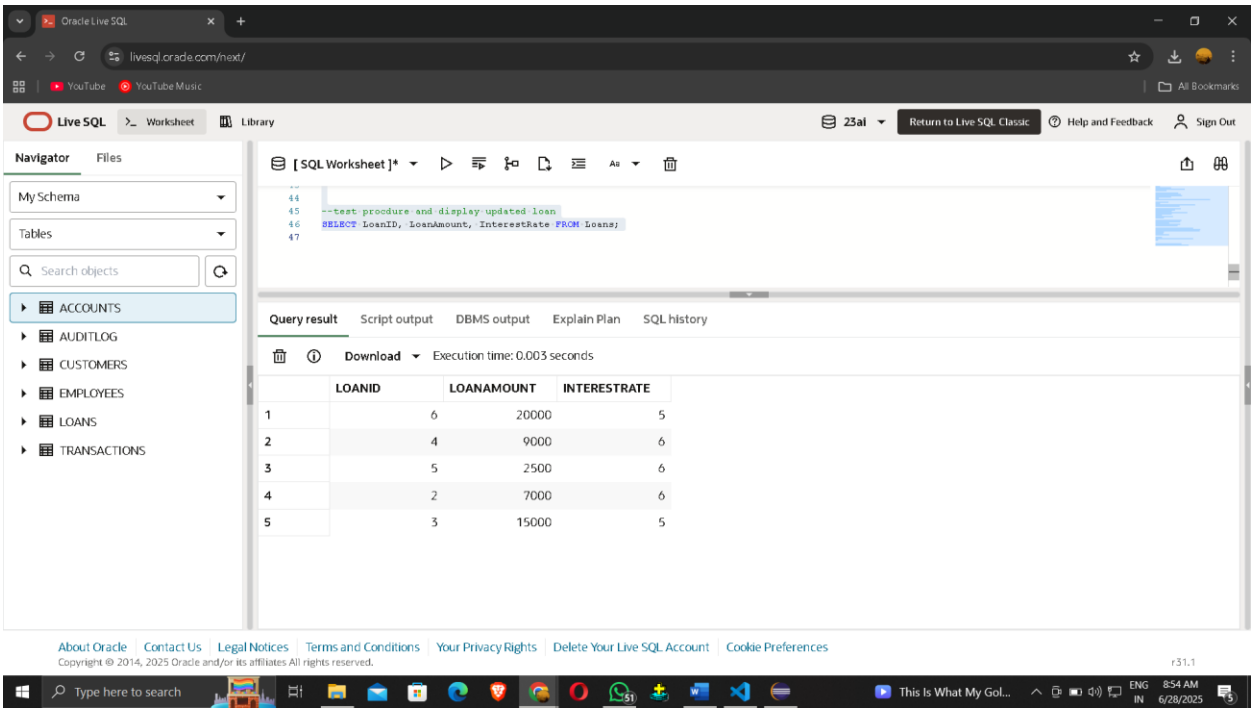


Table:



Exercise 7: Packages

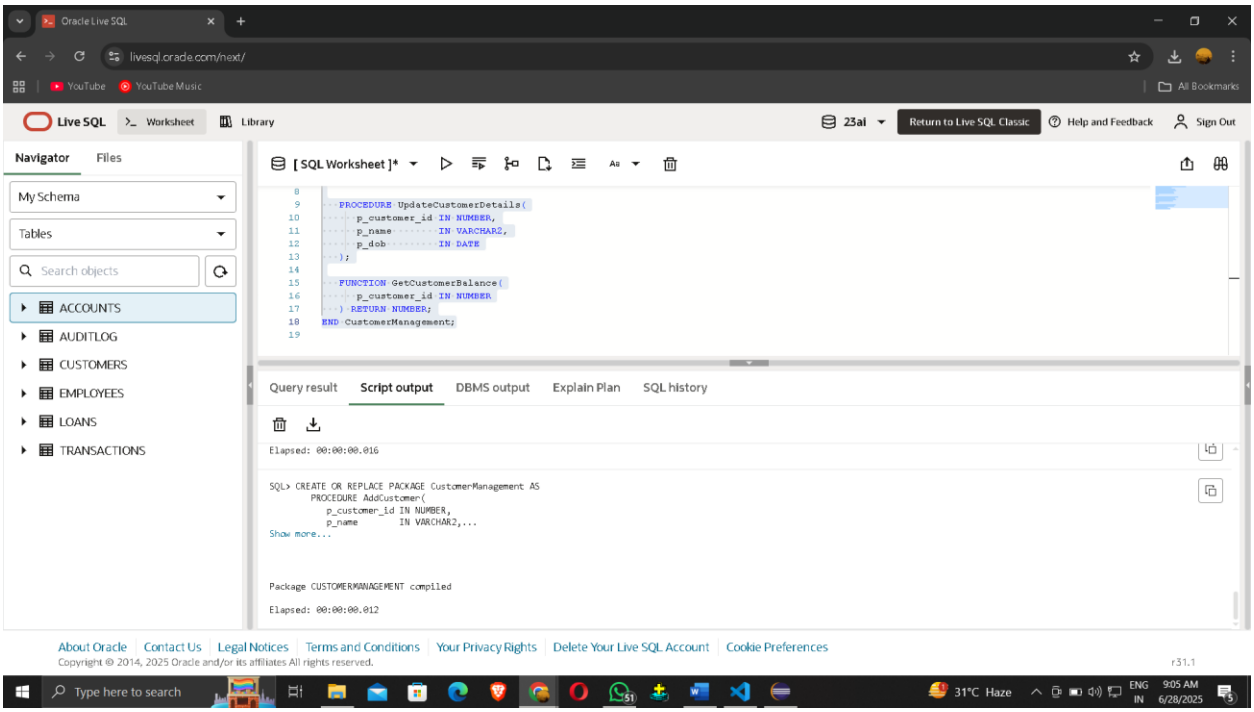
Scenario 1: Group all customer-related procedures and functions into a package.

- **Question:** Create a package **CustomerManagement** with procedures for adding a new customer, updating customer details, and a function to get customer balance.

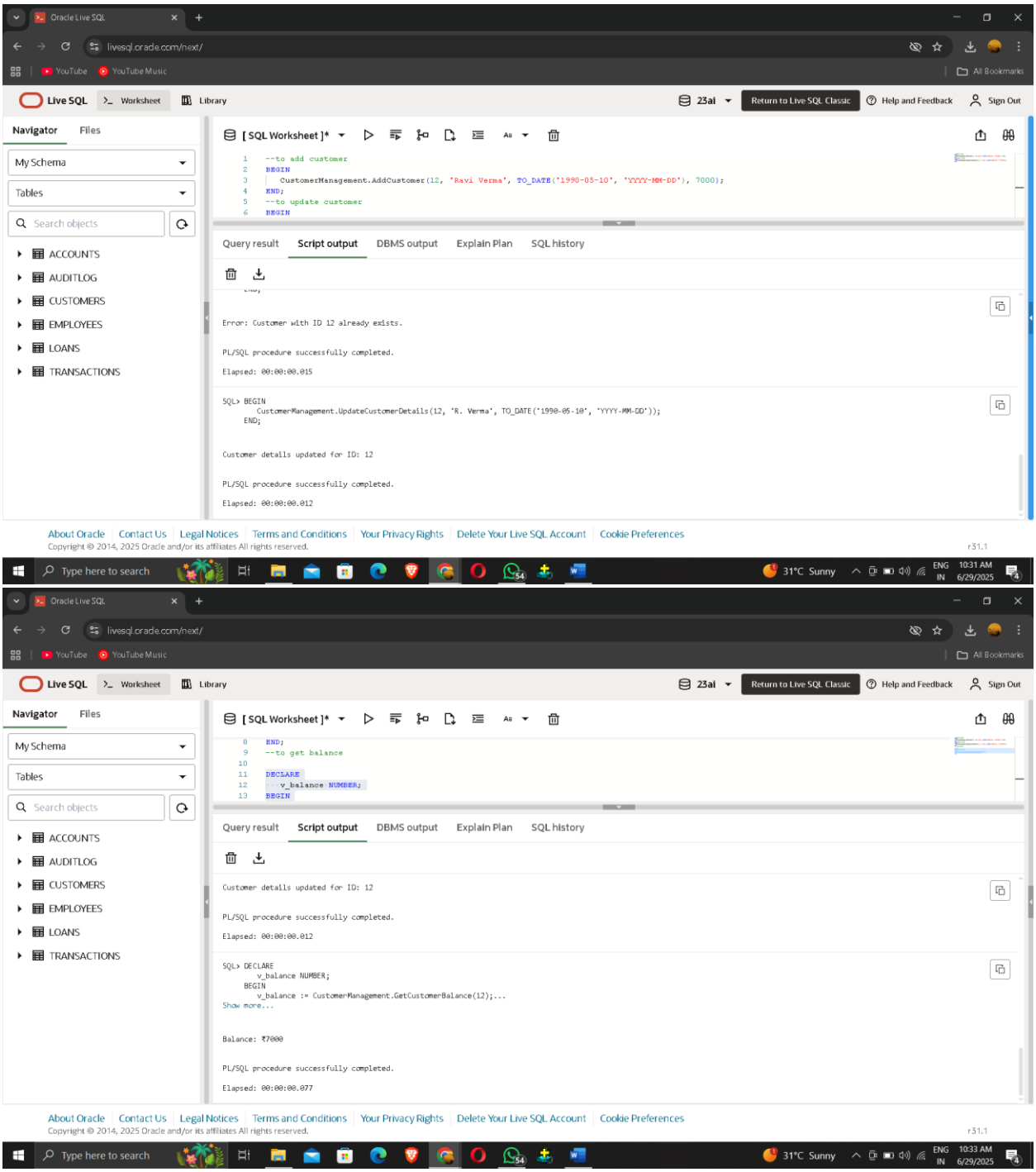
Procedure:

```
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);

    COMMIT;
    DBMS_OUTPUT.PUT_LINE('Customer added successfully. ID: ' || p_customer_id);
  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);
  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 ID ' || p_customer_id || ' not found.');
    ELSE
      COMMIT;
      DBMS_OUTPUT.PUT_LINE('Customer details updated for ID: ' || p_customer_id);
    END IF;
  END;

  FUNCTION GetCustomerBalance (
    p_customer_id IN NUMBER
  ) RETURN NUMBER IS
    v_balance NUMBER;
  BEGIN
    SELECT Balance INTO v_balance
    FROM Customers
    WHERE CustomerID = p_customer_id;

    RETURN v_balance;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('Customer not found.');
      RETURN NULL;
    WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
      RETURN NULL;
  END;

END CustomerManagement;
```




```
--to add customer
BEGIN
  CustomerManagement.AddCustomer(12, 'Ravi Verma', TO_DATE('1990-05-10', 'YYYY-MM-DD'), 7000);
END;
--to update customer
BEGIN
  CustomerManagement.UpdateCustomerDetails(12, 'R. Verma', TO_DATE('1990-05-10', 'YYYY-MM-DD'));
END;
--to get balance

DECLARE
  v_balance NUMBER;
BEGIN
  v_balance := CustomerManagement.GetCustomerBalance(12);
  DBMS_OUTPUT.PUT_LINE('Balance: ₹' || v_balance);
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.

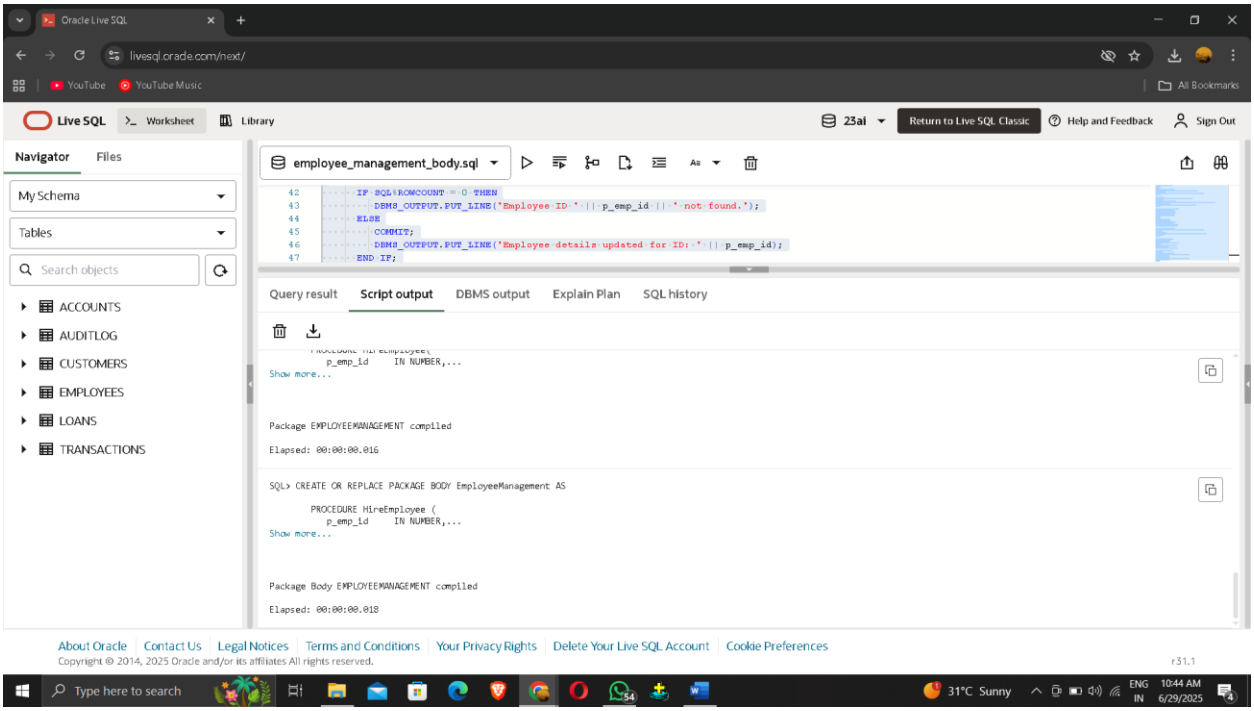
Procedure:

```
CREATE OR REPLACE PACKAGE EmployeeManagement AS
  -- Procedure to hire a 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
  );

  -- Procedure to update existing employee details
  PROCEDURE UpdateEmployeeDetails(
    p_emp_id     IN NUMBER,
    p_name       IN VARCHAR2,
    p_position   IN VARCHAR2,
    p_salary     IN NUMBER,
    p_department IN VARCHAR2
  );

  -- Function to calculate annual salary
  FUNCTION GetAnnualSalary(
    p_emp_id IN NUMBER
  ) RETURN NUMBER;
END EmployeeManagement;


CREATE OR REPLACE PACKAGE BODY 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
  ) 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
    );

    COMMIT;
    DBMS_OUTPUT.PUT_LINE('Hired employee: ' || p_name);
  EXCEPTION
```

```
  WHEN DUP_VAL_ON_INDEX THEN
    DBMS_OUTPUT.PUT_LINE('Error: Employee with ID ' || p_emp_id || ' already exists.');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Unexpected error: ' || SQLERRM);
END;

PROCEDURE UpdateEmployeeDetails (
  p_emp_id     IN NUMBER,
  p_name       IN VARCHAR2,
  p_position   IN VARCHAR2,
  p_salary     IN NUMBER,
  p_department IN VARCHAR2
) IS
BEGIN
  UPDATE Employees
  SET Name = p_name,
     Position = p_position,
     Salary = p_salary,
     Department = p_department
  WHERE EmployeeID = p_emp_id;

  IF SQL%ROWCOUNT = 0 THEN
    DBMS_OUTPUT.PUT_LINE('Employee ID ' || p_emp_id || ' not found.');
  ELSE
    COMMIT;
    DBMS_OUTPUT.PUT_LINE('Employee details updated for ID: ' || p_emp_id);
  END IF;
END;

FUNCTION GetAnnualSalary (
  p_emp_id IN NUMBER
) RETURN NUMBER IS
  v_salary Employees.Salary%TYPE;
BEGIN
  SELECT Salary INTO v_salary
  FROM Employees
  WHERE EmployeeID = p_emp_id;

  RETURN v_salary * 12;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('Employee not found.');
    RETURN NULL;
  WHEN OTHERS THEN
```

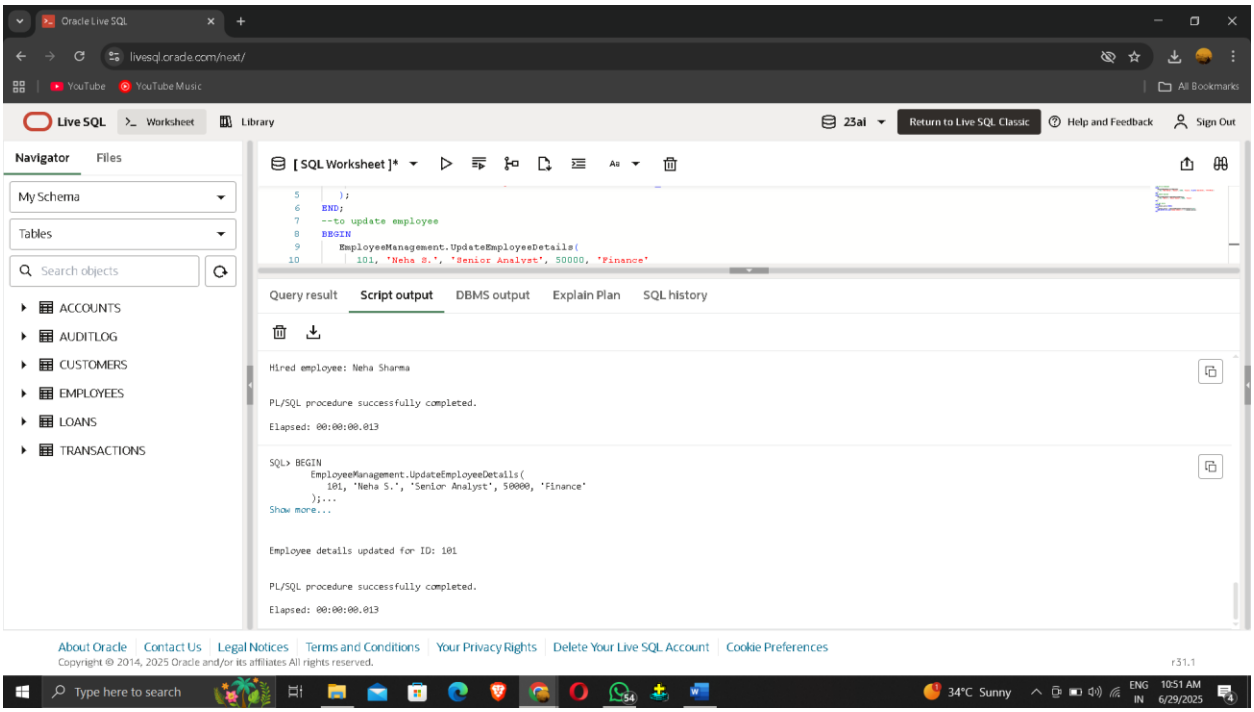
```
      DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
      RETURN NULL;
  END;

END EmployeeManagement;
```

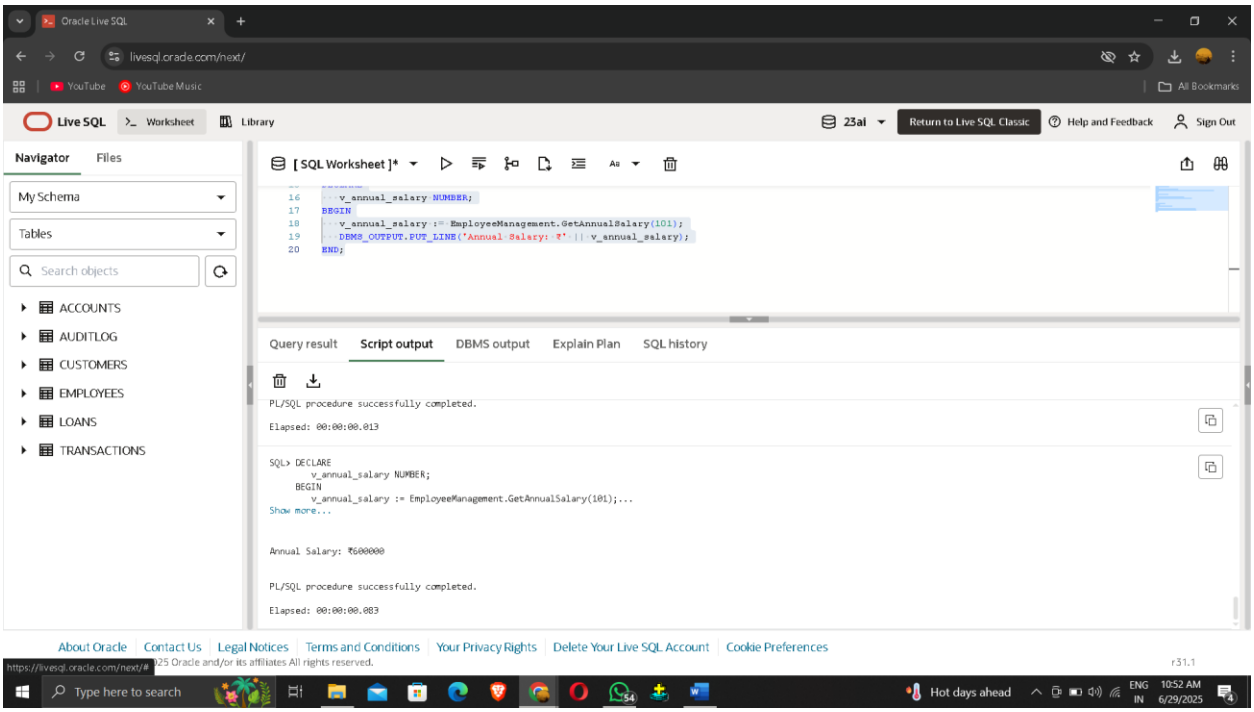


Testing package

```
--to hire an employee
BEGIN
  EmployeeManagement.HireEmployee(
    101, 'Neha Sharma', 'Analyst', 45000, 'Finance', TO_DATE('2022-08-01', 'YYYY-MM-DD')
  );
END;
--to update employee
BEGIN
  EmployeeManagement.UpdateEmployeeDetails(
    101, 'Neha S.', 'Senior Analyst', 50000, 'Finance'
  );
END;

--to get salary
DECLARE
  v_annual_salary NUMBER;
```

```
BEGIN
  v_annual_salary := EmployeeManagement.GetAnnualSalary(101);
  DBMS_OUTPUT.PUT_LINE('Annual Salary: ₹' || v_annual_salary);
END;
```





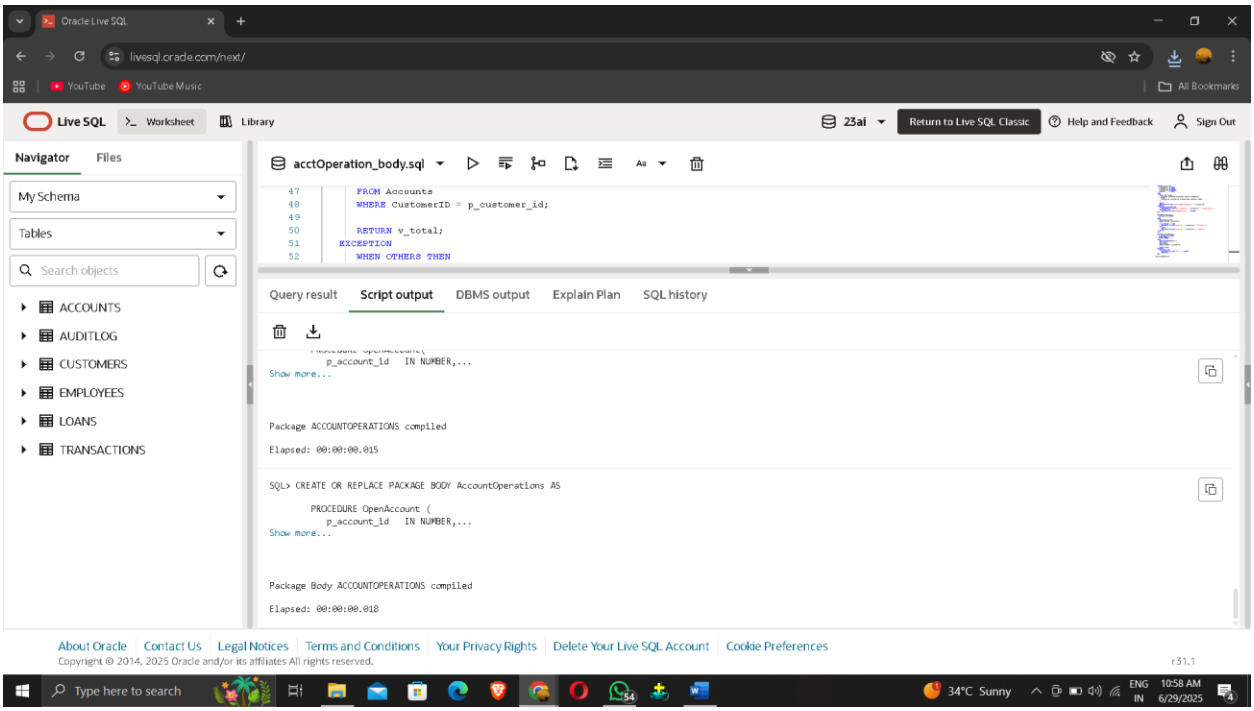
Scenario 3: Group all account-related operations into a package.

- **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.

Procedure:

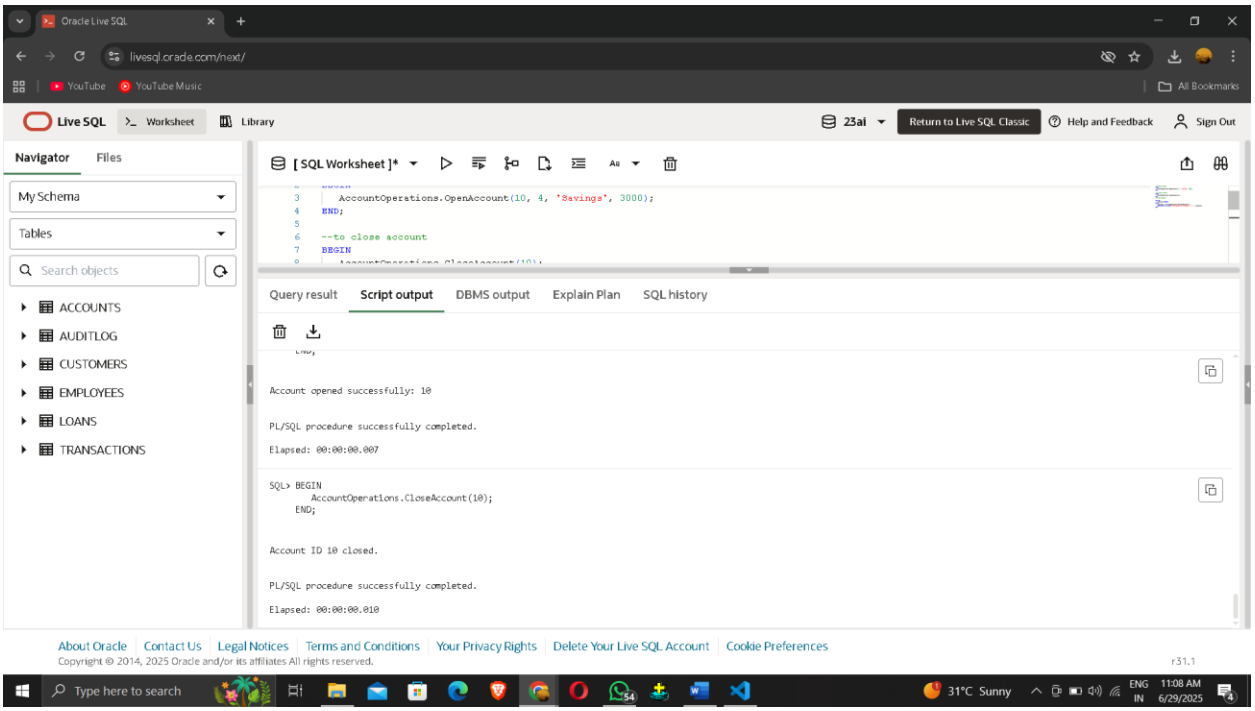
```
CREATE OR REPLACE PACKAGE AccountOperations AS
  -- Procedure to open a new account
  PROCEDURE OpenAccount(
    p_account_id   IN NUMBER,
    p_customer_id  IN NUMBER,
    p_account_type IN VARCHAR2,
    p_balance      IN NUMBER
  );

  -- Procedure to close an account
  PROCEDURE CloseAccount(
    p_account_id IN NUMBER
  );

  -- Function to get total balance for a customer across all accounts
  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
    );

    COMMIT;
    DBMS_OUTPUT.PUT_LINE('Account opened successfully: ' || p_account_id);
```

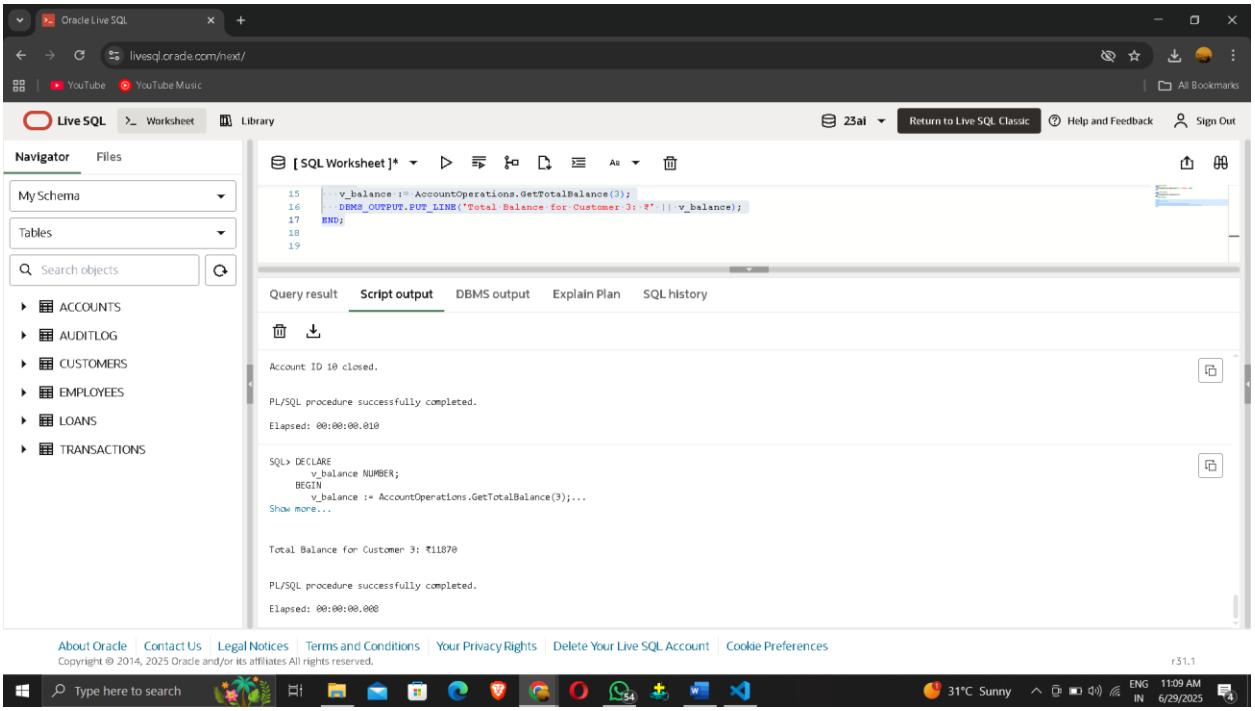
```
  EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
      DBMS_OUTPUT.PUT_LINE('Error: Account ID ' || p_account_id || ' already exists.');
    WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('Unexpected error: ' || SQLERRM);
  END;

  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 ID ' || p_account_id || ' not found.');
    ELSE
      COMMIT;
      DBMS_OUTPUT.PUT_LINE('Account ID ' || p_account_id || ' closed.');
    END IF;
  END;

  FUNCTION GetTotalBalance (
    p_customer_id IN 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;
  EXCEPTION
    WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
      RETURN NULL;
  END;

END AccountOperations;
```



Testing Account Operation

```
--to open account
BEGIN
  AccountOperations.OpenAccount(10, 4, 'Savings', 3000);
END;

--to close account
BEGIN
  AccountOperations.CloseAccount(10);
END;
--to get balance

DECLARE
  v_balance NUMBER;
BEGIN
  v_balance := AccountOperations.GetTotalBalance(3);
  DBMS_OUTPUT.PUT_LINE('Total Balance for Customer 3: ₹' || v_balance);
END;
```


Oracle Live SQL
livesql.oracle.com/next/
Live SQL
Worksheet
Library
23ai
Return to Live SQL Classic
Help and Feedback
Sign Out
Navigator
Files
My Schema
Tables
Search objects
ACCOUNTS
AUDITLOG
CUSTOMERS
EMPLOYEES
LOANS
TRANSACTIONS
[SQL Worksheet]*
AccountOperations.OpenAccount(10, 4, 'Savings', 3000);
END;
--to close account
BEGIN
Query result
Script output
DBMS output
Explain Plan
SQL history
Account opened successfully: 10
PL/SQL procedure successfully completed.
Elapsed: 00:00:00.007
SQL> BEGIN
AccountOperations.CloseAccount(10);
END;
Account ID 10 closed.
PL/SQL procedure successfully completed.
Elapsed: 00:00:00.010
About Oracle
Contact Us
Legal Notices
Terms and Conditions
Your Privacy Rights
Delete Your Live SQL Account
Cookie Preferences
Copyright © 2014, 2025 Oracle and/or its affiliates All rights reserved.
r31.1
Type here to search
31°C Sunny
ENG
IN
11:08 AM
6/29/2025


Oracle Live SQL
livesql.oracle.com/next/
Live SQL
Worksheet
Library
23ai
Return to Live SQL Classic
Help and Feedback
Sign Out
Navigator
Files
My Schema
Tables
Search objects
ACCOUNTS
AUDITLOG
CUSTOMERS
EMPLOYEES
LOANS
TRANSACTIONS
[SQL Worksheet]*
v_balance := AccountOperations.GetTotalBalance(3);
DBMS_OUTPUT.PUT_LINE('Total Balance for Customer 3: ₹' || v_balance);
END;
Query result
Script output
DBMS output
Explain Plan
SQL history
Account ID 10 closed.
PL/SQL procedure successfully completed.
Elapsed: 00:00:00.010
SQL> DECLARE
v_balance NUMBER;
BEGIN
v_balance := AccountOperations.GetTotalBalance(3);...
Show more...
Total Balance for Customer 3: ₹11870
PL/SQL procedure successfully completed.
Elapsed: 00:00:00.008
About Oracle
Contact Us
Legal Notices
Terms and Conditions
Your Privacy Rights
Delete Your Live SQL Account
Cookie Preferences
Copyright © 2014, 2025 Oracle and/or its affiliates All rights reserved.
r31.1
Type here to search
31°C Sunny
ENG
IN
11:09 AM
6/29/2025

Schema to be Created

```
CREATE TABLE Customers (
    CustomerID NUMBER PRIMARY KEY,
    Name VARCHAR2(100),
    DOB DATE,
    Balance NUMBER,
    LastModified DATE
);

CREATE TABLE Accounts (
    AccountID NUMBER PRIMARY KEY,
    CustomerID NUMBER,
    AccountType VARCHAR2(20),
    Balance NUMBER,
    LastModified DATE,
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);

CREATE TABLE Transactions (
    TransactionID NUMBER PRIMARY KEY,
    AccountID NUMBER,
    TransactionDate DATE,
    Amount NUMBER,
    TransactionType VARCHAR2(10),
    FOREIGN KEY (AccountID) REFERENCES Accounts(AccountID)
);

CREATE TABLE Loans (
    LoanID NUMBER PRIMARY KEY,
    CustomerID NUMBER,
    LoanAmount NUMBER,
    InterestRate NUMBER,
    StartDate DATE,
    EndDate DATE,
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);

CREATE TABLE Employees (
    EmployeeID NUMBER PRIMARY KEY,
    Name VARCHAR2(100),
    Position VARCHAR2(50),
    Salary NUMBER,
```

```
    Department VARCHAR2(50),
    HireDate DATE
);
```

Example Scripts for Sample Data Insertion

```
INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)
VALUES (1, 'John Doe', TO_DATE('1985-05-15', 'YYYY-MM-DD'), 1000, SYSDATE);

INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)
VALUES (2, 'Jane Smith', TO_DATE('1990-07-20', 'YYYY-MM-DD'), 1500, SYSDATE);

INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified)
VALUES (1, 1, 'Savings', 1000, SYSDATE);

INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified)
VALUES (2, 2, 'Checking', 1500, SYSDATE);

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)
VALUES (1, 1, SYSDATE, 200, 'Deposit');

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)
VALUES (2, 2, SYSDATE, 300, 'Withdrawal');

INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate)
VALUES (1, 1, 5000, 5, SYSDATE, ADD_MONTHS(SYSDATE, 60));

INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate)
VALUES (1, 'Alice Johnson', 'Manager', 70000, 'HR', TO_DATE('2015-06-15', 'YYYY-MM-DD'));

INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate)
VALUES (2, 'Bob Brown', 'Developer', 60000, 'IT', TO_DATE('2017-03-20', 'YYYY-MM-DD'));
```