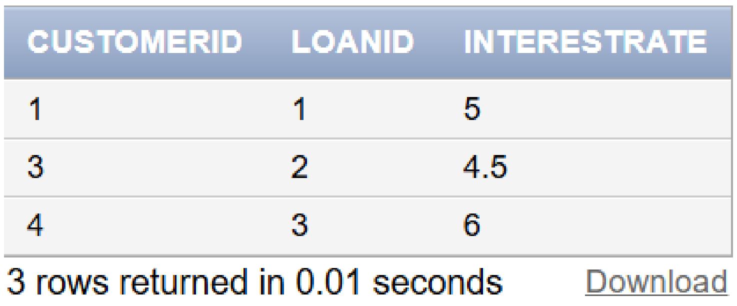
PLSQL EXERCISE

EXCERCISE 1

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

```
DECLARE
   v_current_date DATE := SYSDATE;
   v_age NUMBER;
BEGIN
   -- Loop through all customers
   FOR rec IN (SELECT CustomerID, DOB FROM Customers) LOOP
      -- Calculate the age of the customer
      v_age := TRUNC(MONTHS_BETWEEN(v_current_date, rec.DOB) / 12);

      -- Check if the customer is above 60 years old
      IF v_age > 60 THEN
         -- Update loan interest rates with a 1% discount
         UPDATE Loans
         SET InterestRate = InterestRate - 1
         WHERE CustomerID = rec.CustomerID;
      END IF;
   END LOOP;

   COMMIT;  -- Commit the changes
END;
/
```



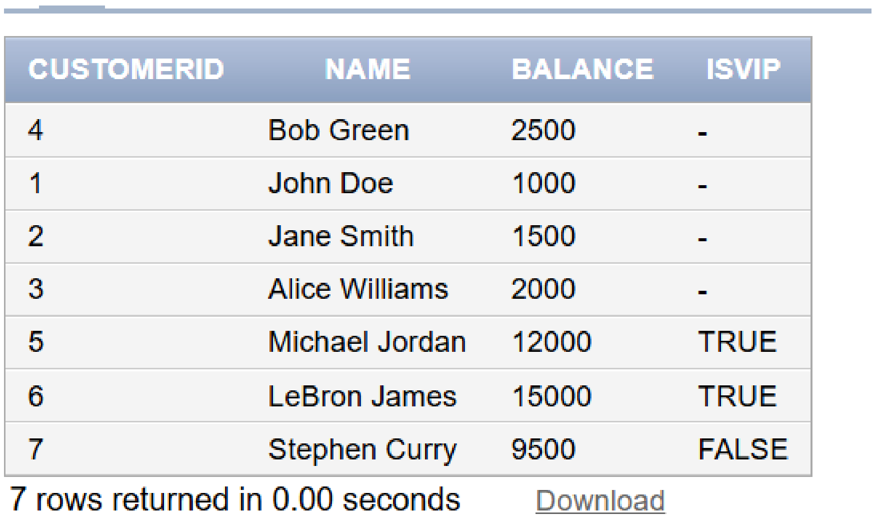
CUSTOMERID	LOANID	INTERESTRATE
1	1	5
3	2	4.5
4	3	6

3 rows returned in 0.01 seconds   Download

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
  -- Loop through all customers
  FOR rec IN (SELECT CustomerID, Balance FROM Customers) LOOP
    -- Check if the balance is over $10,000
    IF rec.Balance > 10000 THEN
      -- Update the IsVIP flag (assuming you have added an IsVIP column in the Customers table)
      UPDATE Customers
      SET IsVIP = 'TRUE'
      WHERE CustomerID = rec.CustomerID;
    END IF;
  END LOOP;

  COMMIT;  -- Commit the changes
END;
/
```



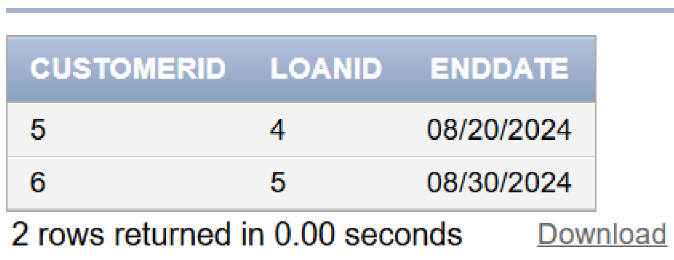
CUSTOMERID	NAME	BALANCE	ISVIP
4	Bob Green	2500	-
1	John Doe	1000	-
2	Jane Smith	1500	-
3	Alice Williams	2000	-
5	Michael Jordan	12000	TRUE
6	LeBron James	15000	TRUE
7	Stephen Curry	9500	FALSE

7 rows returned in 0.00 seconds   Download

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;
BEGIN
   -- Fetch all loans due in the next 30 days
   FOR rec IN (SELECT CustomerID, LoanID, EndDate FROM Loans
            WHERE EndDate BETWEEN SYSDATE AND SYSDATE + 30) LOOP
      -- Print a reminder message
      v_due_date := rec.EndDate;
      DBMS_OUTPUT.PUT_LINE('Reminder: Loan ID ' || rec.LoanID ||
                     ' for Customer ID ' || rec.CustomerID ||
                     ' is due on ' || TO_CHAR(v_due_date, 'YYYY-MM-DD') ||
                     '. Please take action accordingly.');
   END LOOP;
END;
/
```



CUSTOMERID	LOANID	ENDDATE
5	4	08/20/2024
6	5	08/30/2024

2 rows returned in 0.00 seconds Download

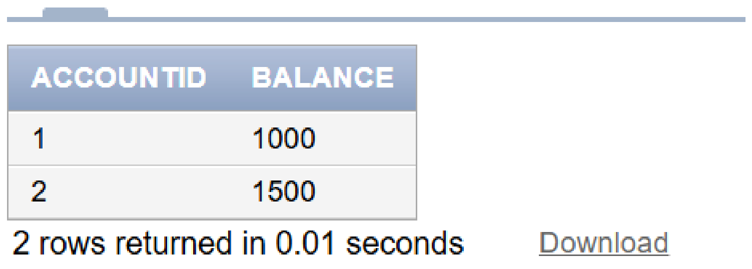
Exercise 2: Error Handling

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_account_id IN NUMBER,
   p_to_account_id IN NUMBER,
   p_amount IN NUMBER
) AS
   v_from_balance NUMBER;
   v_to_balance NUMBER;
BEGIN
   -- Start the transaction
   SAVEPOINT transfer_start;

   -- Get the current balance of the from account
   SELECT Balance INTO v_from_balance
   FROM Accounts
   WHERE AccountID = p_from_account_id;

   -- Check for sufficient funds
   IF v_from_balance < p_amount THEN
      RAISE_APPLICATION_ERROR(-20001, 'Insufficient funds in the source account.');
   END IF;

   -- Get the current balance of the to account
   SELECT Balance INTO v_to_balance
   FROM Accounts
   WHERE AccountID = p_to_account_id;

   -- Update the balances
   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 the transaction
   COMMIT;
```

```
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      ROLLBACK TO transfer_start;
      DBMS_OUTPUT.PUT_LINE('Error: One or both account IDs are invalid.');
   WHEN OTHERS THEN
      ROLLBACK TO transfer_start;
      DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
END;
/
```



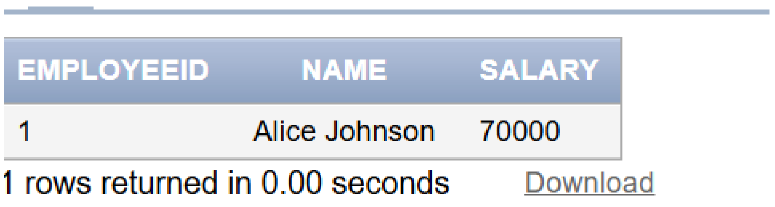
ACCOUNTID	BALANCE
1	1000
2	1500

2 rows returned in 0.01 seconds   Download

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_employee_id IN NUMBER,
   p_percentage IN NUMBER
) AS
   v_current_salary NUMBER;
BEGIN
   -- Start the transaction
   SAVEPOINT salary_update_start;

   -- Get the current salary of the employee
   SELECT Salary INTO v_current_salary
   FROM Employees
   WHERE EmployeeID = p_employee_id;

   -- Update the salary
   UPDATE Employees
   SET Salary = Salary * (1 + p_percentage / 100)
   WHERE EmployeeID = p_employee_id;

   -- Commit the transaction
   COMMIT;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      ROLLBACK TO salary_update_start;
      DBMS_OUTPUT.PUT_LINE('Error: Employee ID does not exist.');
   WHEN OTHERS THEN
      ROLLBACK TO salary_update_start;
      DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
END;
/
```



EMPLOYEEID	NAME	SALARY
1	Alice Johnson	70000

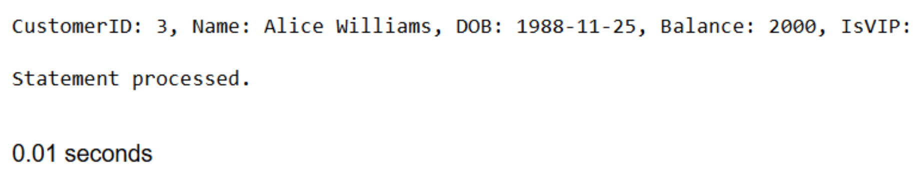
1 rows returned in 0.00 seconds   Download

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.

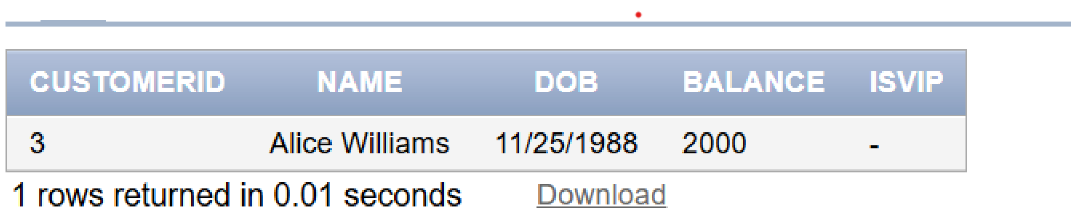
CODE

```
CREATE OR REPLACE PROCEDURE AddNewCustomer (
   p_customer_id IN NUMBER,
   p_name IN VARCHAR2,
   p_dob IN DATE,
   p_balance IN NUMBER
) AS
BEGIN
   -- Start the transaction
   SAVEPOINT customer_add_start;

   -- Try to insert a new customer
   BEGIN
      INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified, IsVIP)
      VALUES (p_customer_id, p_name, p_dob, p_balance, SYSDATE, 'FALSE');
      COMMIT;
   EXCEPTION
      WHEN DUP_VAL_ON_INDEX THEN
         ROLLBACK TO customer_add_start;
         DBMS_OUTPUT.PUT_LINE('Error: Customer with ID ' || p_customer_id || ' already exists.');
      WHEN OTHERS THEN
         ROLLBACK TO customer_add_start;
```

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



```
CustomerID: 3, Name: Alice Williams, DOB: 1988-11-25, Balance: 2000, IsVIP:

Statement processed.

0.01 seconds
```



CUSTOMERID	NAME	DOB	BALANCE	ISVIP
3	Alice Williams	11/25/1988	2000	-

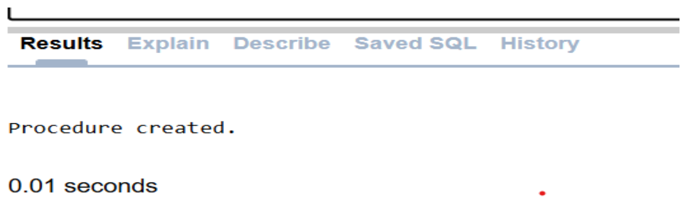
1 rows returned in 0.01 seconds   Download

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
    -- Update the balance for all savings accounts
    UPDATE Accounts
    SET Balance = Balance * 1.01  -- Applying 1% interest
    WHERE AccountType = 'Savings';

    COMMIT;  -- Commit the transaction
END;
/
```

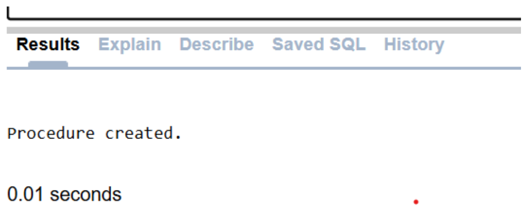


```
Procedure created.

0.01 seconds
```

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_percentage IN NUMBER
) AS
BEGIN
   -- Update the salary of employees in the specified department
   UPDATE Employees
   SET Salary = Salary * (1 + p_bonus_percentage / 100)
   WHERE Department = p_department;

   COMMIT;  -- Commit the transaction
END;
/
```

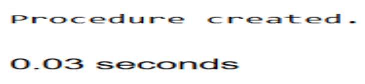


```
Procedure created.

0.01 seconds
```

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_id IN NUMBER,
   p_to_account_id IN NUMBER,
   p_amount IN NUMBER
) AS
   v_from_balance NUMBER;
   v_to_balance NUMBER;
BEGIN
   -- Start the transaction
   SAVEPOINT transfer_start;

   -- Get the current balance of the from account
   SELECT Balance INTO v_from_balance
   FROM Accounts
   WHERE AccountID = p_from_account_id;

   -- Check for sufficient funds
   IF v_from_balance < p_amount THEN
      RAISE_APPLICATION_ERROR(-20001, 'Insufficient funds in the source account.');
   END IF;

   -- Get the current balance of the to account
   SELECT Balance INTO v_to_balance
   FROM Accounts
   WHERE AccountID = p_to_account_id;

   -- Update the balances
   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 the transaction
   COMMIT;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      ROLLBACK TO transfer_start;
      DBMS_OUTPUT.PUT_LINE('Error: One or both account IDs are invalid.');
   WHEN OTHERS THEN
      ROLLBACK TO transfer_start;
      DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
END;
/
```



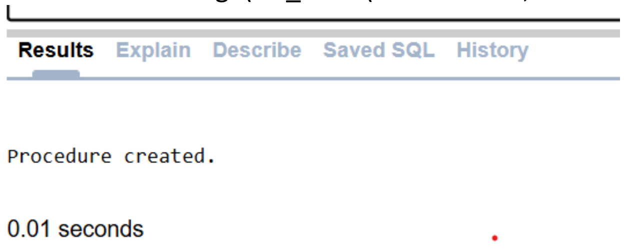
```
Procedure created.

0.03 seconds
```

Exercise 4: Functions

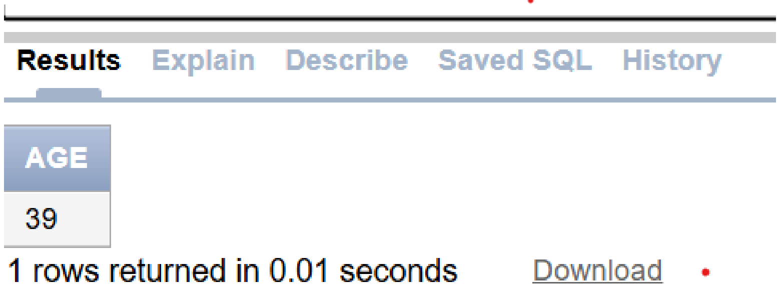
Scenario 1: Calculate the age of customers for eligibility checks.
Question: Write a function CalculateAge that takes a customer's date of birth as input and returns their age in years.

CODE

```
CREATE OR REPLACE FUNCTION CalculateAge (
   p_dob IN DATE
) RETURN NUMBER IS
   v_age NUMBER;
BEGIN
   v_age := TRUNC(MONTHS_BETWEEN(SYSDATE, p_dob) / 12);
   RETURN v_age;
END;
/

SELECT CalculateAge(TO_DATE('1985-05-15', 'YYYY-MM-DD')) AS Age FROM DUAL;
```



```
Procedure created.

0.01 seconds
```



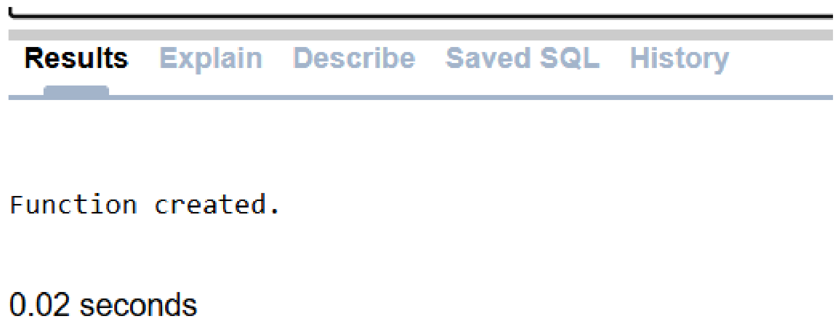
AGE
39

```
1 rows returned in 0.01 seconds
```

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.

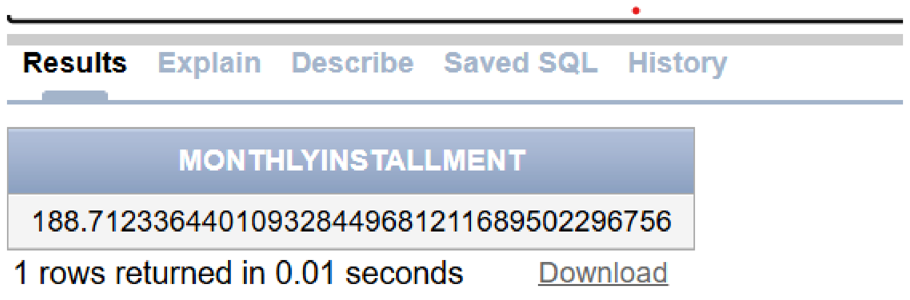
CODE

```
CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment (
   p_loan_amount IN NUMBER,
   p_interest_rate IN NUMBER,  -- Annual interest rate in percentage
   p_loan_duration_years IN NUMBER
) RETURN NUMBER IS
   v_monthly_rate NUMBER;
   v_number_of_payments NUMBER;
   v_installment_amount NUMBER;
BEGIN
   v_monthly_rate := p_interest_rate / 100 / 12;
   v_number_of_payments := p_loan_duration_years * 12;

   IF v_monthly_rate = 0 THEN
      v_installment_amount := p_loan_amount / v_number_of_payments;
   ELSE
      v_installment_amount := p_loan_amount * v_monthly_rate /
         (1 - POWER(1 + v_monthly_rate, -v_number_of_payments));
   END IF;

   RETURN v_installment_amount;
END;
/

SELECT CalculateMonthlyInstallment(10000, 5, 5) AS MonthlyInstallment FROM DUAL;
```



```
Function created.
```

0.02 seconds



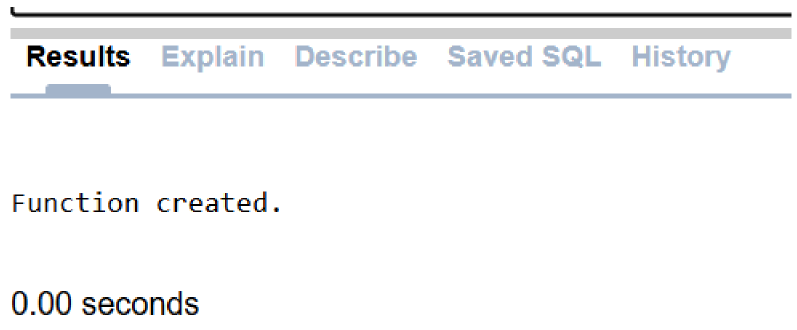
MONTHLYINSTALLMENT
188.7123364401093284496812116895022967\u0035\u0036

1 rows returned in 0.01 seconds

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_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;  -- Account not found
END;
/
```



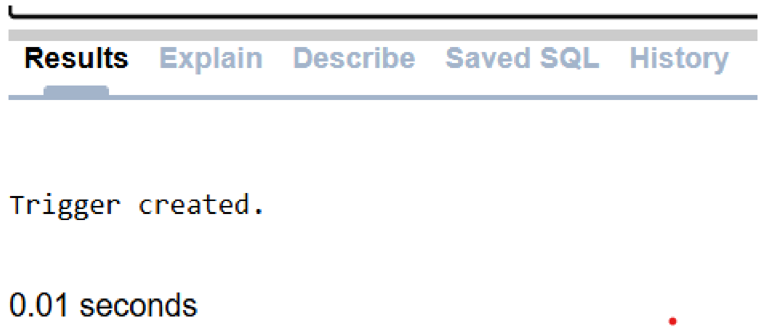
```
Function created.

0.00 seconds
```

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;
/
```

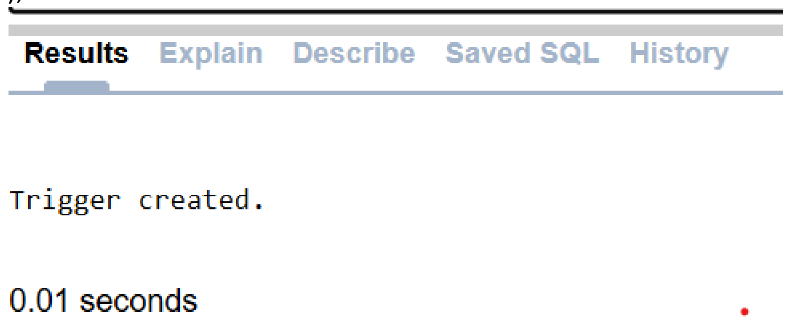


```
Trigger created.

0.01 seconds
```

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 TABLE AuditLog (
   AuditID NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
   TransactionID NUMBER,
   AuditDate DATE,
   Action VARCHAR2(50)
);
```



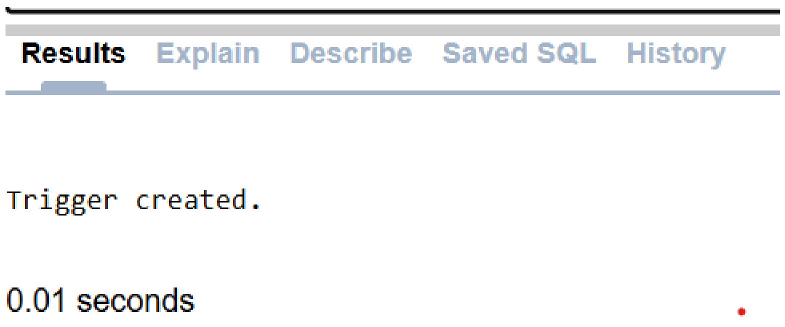
```
Trigger created.
```

0.01 seconds

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
   -- Ensure deposits are positive
   IF :NEW.TransactionType = 'Deposit' AND :NEW.Amount <= 0 THEN
      RAISE_APPLICATION_ERROR(-20002, 'Deposit amount must be positive.');
   END IF;

   -- Ensure withdrawals do not exceed balance
   IF :NEW.TransactionType = 'Withdrawal' THEN
      SELECT Balance INTO v_balance
      FROM Accounts
      WHERE AccountID = :NEW.AccountID;

      IF v_balance < :NEW.Amount THEN
         RAISE_APPLICATION_ERROR(-20003, 'Insufficient balance for withdrawal.');
      END IF;
   END IF;
END;
/
```



```
Trigger created.
```

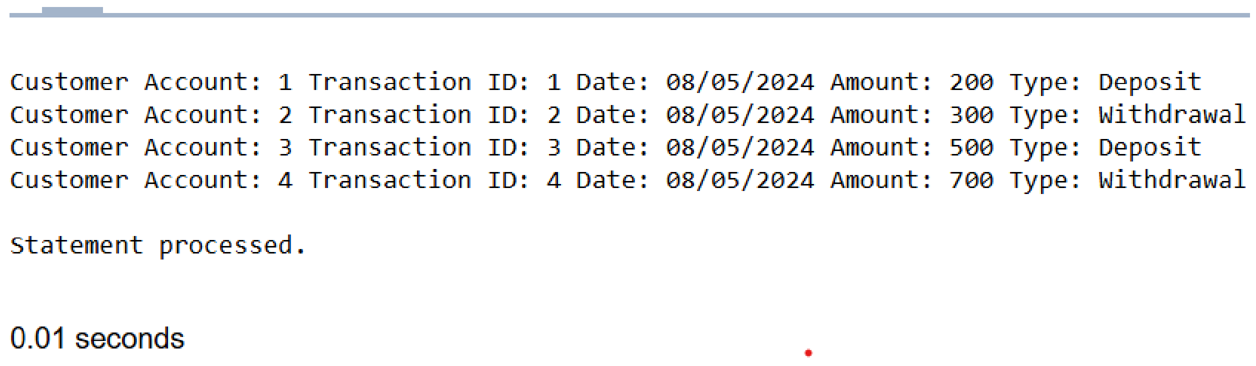
0.01 seconds

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 c_transactions IS
    SELECT t.TransactionID, t.AccountID, t.TransactionDate, t.Amount, t.TransactionType
    FROM Transactions t
    WHERE EXTRACT(MONTH FROM t.TransactionDate) = EXTRACT(MONTH FROM SYSDATE)
    AND EXTRACT(YEAR FROM t.TransactionDate) = EXTRACT(YEAR FROM SYSDATE);

  v_account_id NUMBER;
BEGIN
  FOR rec IN c_transactions LOOP
    v_account_id := rec.AccountID;
    DBMS_OUTPUT.PUT_LINE('Customer Account: ' || v_account_id ||
               ' Transaction ID: ' || rec.TransactionID ||
               ' Date: ' || rec.TransactionDate ||
               ' Amount: ' || rec.Amount ||
               ' Type: ' || rec.TransactionType);
  END LOOP;
END;
/
```



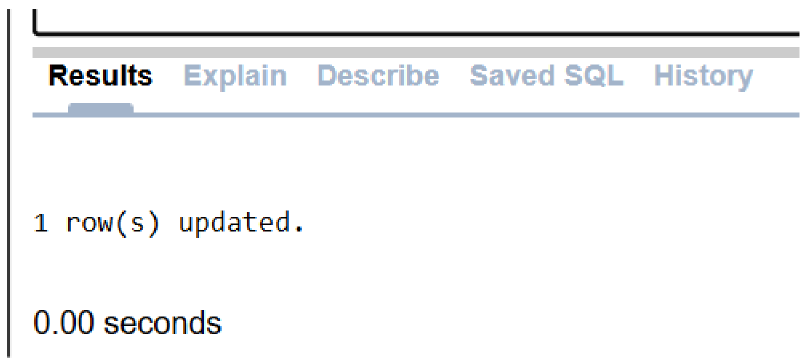
```
Customer Account: 1 Transaction ID: 1 Date: 08/05/2024 Amount: 200 Type: Deposit
Customer Account: 2 Transaction ID: 2 Date: 08/05/2024 Amount: 300 Type: Withdrawal
Customer Account: 3 Transaction ID: 3 Date: 08/05/2024 Amount: 500 Type: Deposit
Customer Account: 4 Transaction ID: 4 Date: 08/05/2024 Amount: 700 Type: Withdrawal

Statement processed.
```

0.01 seconds

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 c_accounts IS
      SELECT AccountID, Balance
      FROM Accounts;

   v_fee NUMBER := 50;  -- Example annual fee
BEGIN
   FOR rec IN c_accounts LOOP
      UPDATE Accounts
      SET Balance = Balance - v_fee
      WHERE AccountID = rec.AccountID;
   END LOOP;

   COMMIT;
END;
/
```



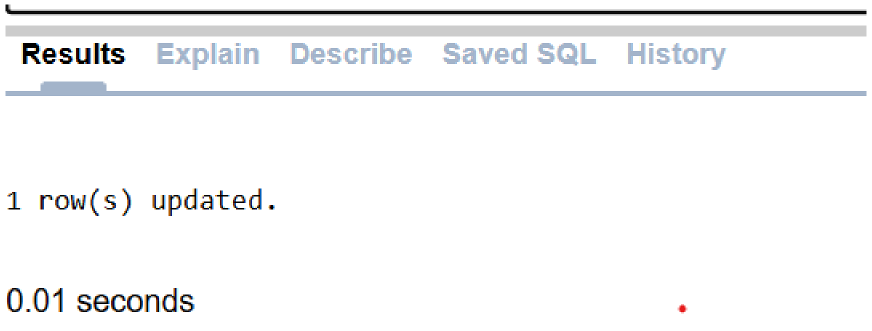
```
1 row(s) updated.
```

0.00 seconds

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 c_loans IS
      SELECT LoanID, LoanAmount, InterestRate
      FROM Loans;

   v_new_interest_rate NUMBER := 6;  -- Example new interest rate
BEGIN
   FOR rec IN c_loans LOOP
      UPDATE Loans
      SET InterestRate = v_new_interest_rate
      WHERE LoanID = rec.LoanID;
   END LOOP;

   COMMIT;
END;
/
```



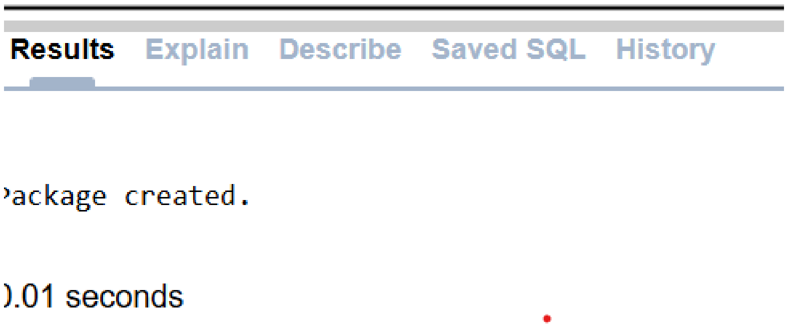
```
1 row(s) updated.

0.01 seconds
```

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 AddNewCustomer (
      p_name IN VARCHAR2,
    p_dob IN DATE,
    p_balance IN NUMBER
  );

  PROCEDURE UpdateCustomer (
    p_customer_id IN NUMBER,
    p_name IN VARCHAR2,
    p_dob IN DATE,
    p_balance IN NUMBER
  );

  FUNCTION GetCustomerBalance (
    p_customer_id IN NUMBER
  ) RETURN NUMBER;
END CustomerManagement;
/
```

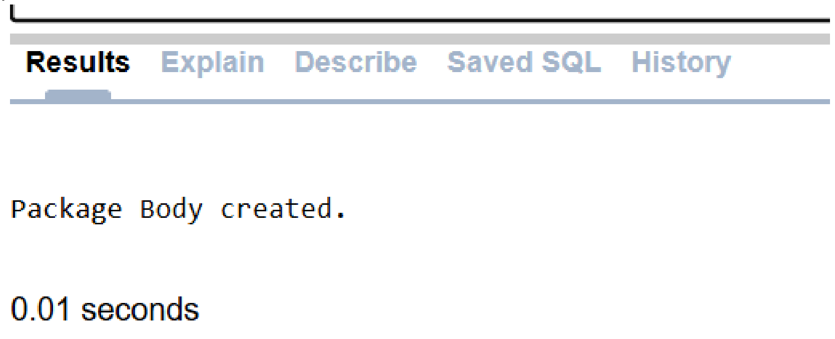


```
ackage created.
```

.01 seconds

```
CREATE OR REPLACE PACKAGE BODY CustomerManagement AS
  PROCEDURE AddNewCustomer (
    p_name IN VARCHAR2,
    p_dob IN DATE,
    p_balance IN NUMBER
  ) IS
  BEGIN
    INSERT INTO Customers (Name, DOB, Balance, LastModified)
    VALUES (p_name, p_dob, p_balance, SYSDATE);
    COMMIT;
  END;

  PROCEDURE UpdateCustomer (
    p_customer_id IN NUMBER,
    p_name IN VARCHAR2,
    p_dob IN DATE,
    p_balance IN NUMBER
  ) IS
  BEGIN
    UPDATE Customers
```

```
        SET Name = p_name,
            DOB = p_dob,
            Balance = p_balance,
            LastModified = SYSDATE
        WHERE CustomerID = p_customer_id;
        COMMIT;
    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;
    END;
END CustomerManagement;
/
```

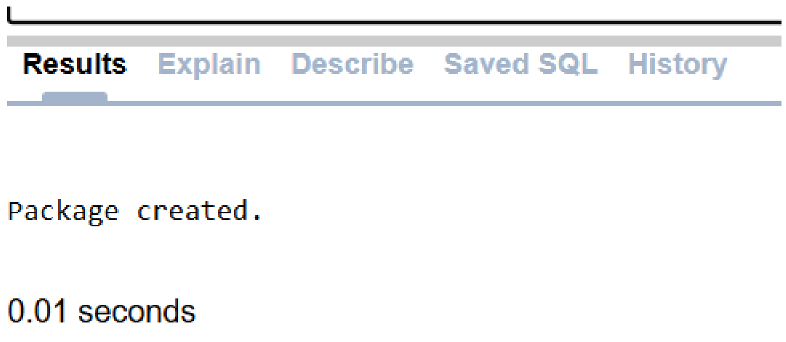


```
Package Body created.

0.01 seconds
```

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.

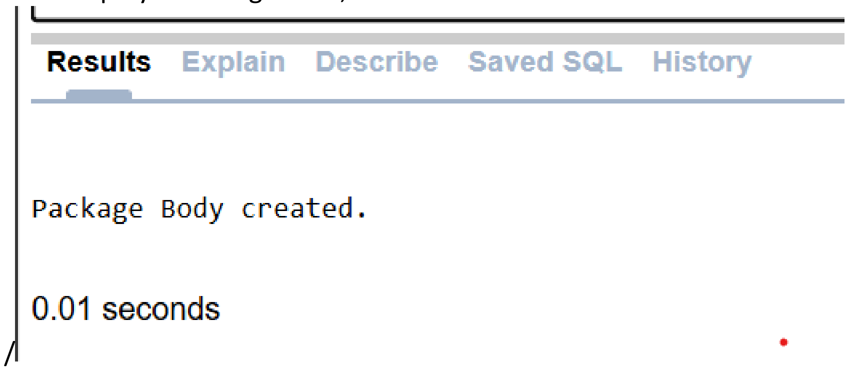
CODE

```
CREATE OR REPLACE PACKAGE EmployeeManagement AS
    PROCEDURE HireEmployee (
        p_name IN VARCHAR2,
        p_position IN VARCHAR2,
        p_salary IN NUMBER,
        p_department IN VARCHAR2,
        p_hire_date IN DATE
    );

    PROCEDURE UpdateEmployee (
        p_employee_id IN NUMBER,
        p_name IN VARCHAR2,
        p_position IN VARCHAR2,
        p_salary IN NUMBER,
        p_department IN VARCHAR2
    );

    FUNCTION CalculateAnnualSalary (
        p_salary IN NUMBER
    ) RETURN NUMBER;
END EmployeeManagement;
/
```



```
Package created.

0.01 seconds
```

```
CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS
    PROCEDURE HireEmployee (
        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 (Name, Position, Salary, Department, HireDate)
        VALUES (p_name, p_position, p_salary, p_department, p_hire_date);
        COMMIT;
    END;

    PROCEDURE UpdateEmployee (
        p_employee_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_employee_id;
      COMMIT;
   END;

   FUNCTION CalculateAnnualSalary (
      p_salary IN NUMBER
   ) RETURN NUMBER IS
   BEGIN
      RETURN p_salary * 12;
   END;
END EmployeeManagement;
```



```
Package Body created.

0.01 seconds
```

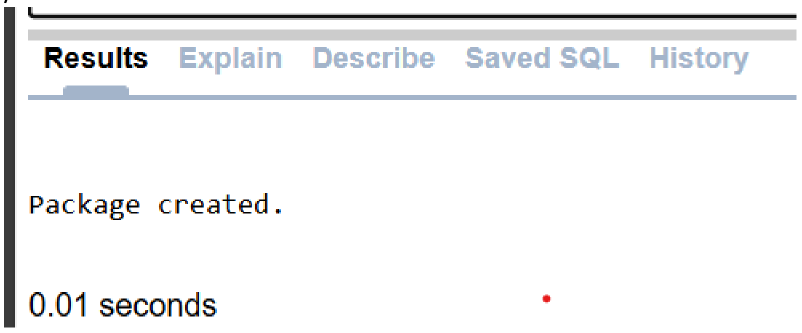
/

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

CODE

```
CREATE OR REPLACE PACKAGE AccountOperations AS
   PROCEDURE OpenNewAccount (
      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;
```

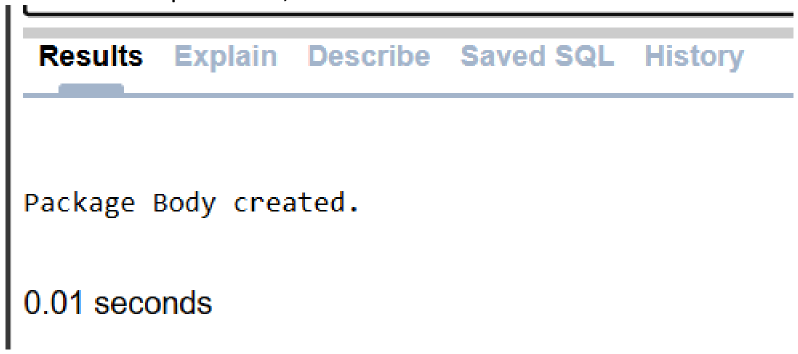
/



```
Package created.

0.01 seconds
```

```
CREATE OR REPLACE PACKAGE BODY AccountOperations AS
  PROCEDURE OpenNewAccount (
    p_customer_id IN NUMBER,
    p_account_type IN VARCHAR2,
    p_balance IN NUMBER
  ) IS
  BEGIN
    INSERT INTO Accounts (CustomerID, AccountType, Balance, LastModified)
    VALUES (p_customer_id, p_account_type, p_balance, SYSDATE);
    COMMIT;
  END;

  PROCEDURE CloseAccount (
    p_account_id IN NUMBER
  ) IS
  BEGIN
    DELETE FROM Accounts
    WHERE AccountID = p_account_id;
    COMMIT;
  END;

  FUNCTION GetTotalBalance (
    p_customer_id IN NUMBER
  ) RETURN NUMBER IS
    v_total_balance NUMBER;
  BEGIN
    SELECT SUM(Balance) INTO v_total_balance
    FROM Accounts
    WHERE CustomerID = p_customer_id;

    IF v_total_balance IS NULL THEN
      v_total_balance := 0;
    END IF;

    RETURN v_total_balance;
  END;
```

END AccountOperations;



```
Package Body created.
```

0.01 seconds

/