3. PLSQL Programming Exercises

# Exercise: 1: Conditional Statements

**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.

SET SERVEROUT ON;

BEGIN

    FOR customer\_rec IN (

        SELECT C.CustomerID, C.Name, C.DOB, L.LoanID, L.InterestRate

        FROM Customers C

        JOIN Loans L ON C.CustomerID = L.CustomerID

    ) LOOP

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

            UPDATE Loans

            SET InterestRate = InterestRate - 1

            WHERE LoanID = customer\_rec.LoanID;

            DBMS\_OUTPUT.PUT\_LINE('1% discount applied to ' || customer\_rec.Name);

        END IF;

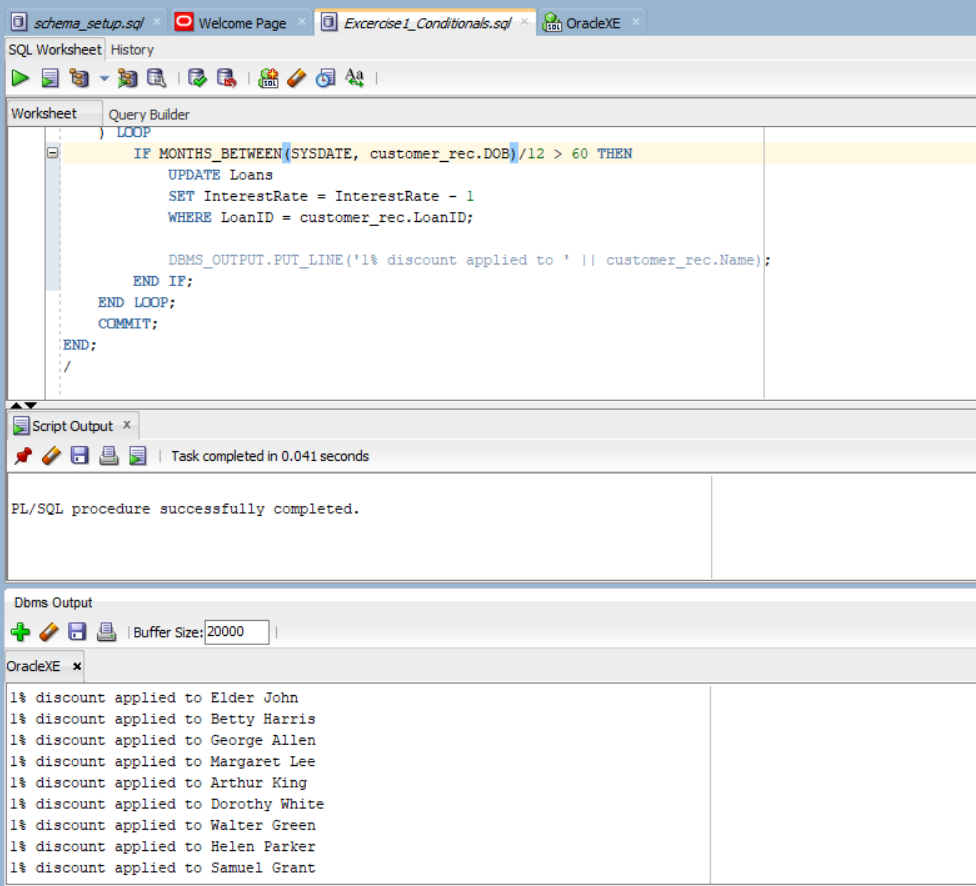
    END LOOP;

    COMMIT;

END;

/

Output



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

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

ALTER TABLE Customers ADD IsVIP VARCHAR2(5);

BEGIN

    FOR Cust IN (

    SELECT CustomerID, Name, Balance From Customers ) LOOP

    IF cust.balance>1000 THEN

         UPDATE Customers

         SET IsVIP='True'

         WHERE CustomerID=cust.CustomerID;

         DBMS\_OUTPUT.PUT\_LINE(cust.Name || 'promoted to VIP');

    ELSE

        DBMS\_OUTPUT.PUT\_LINE(cust.Name || ' not elligible to VIP');

    END IF;

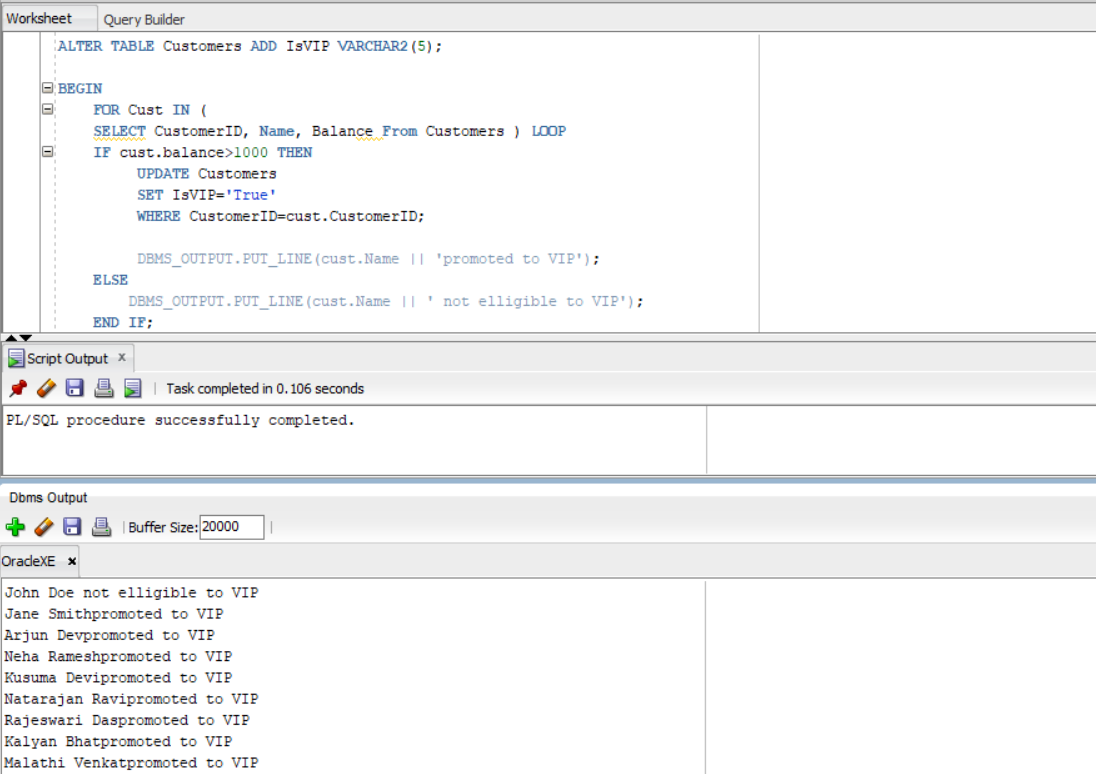
   END LOOP;

   COMMIT;

END;

/

Output:



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

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

BEGIN

    For rec in (SELECT  L.LoanID, L.CustomerID, C.Name, L.EndDate

    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 ||

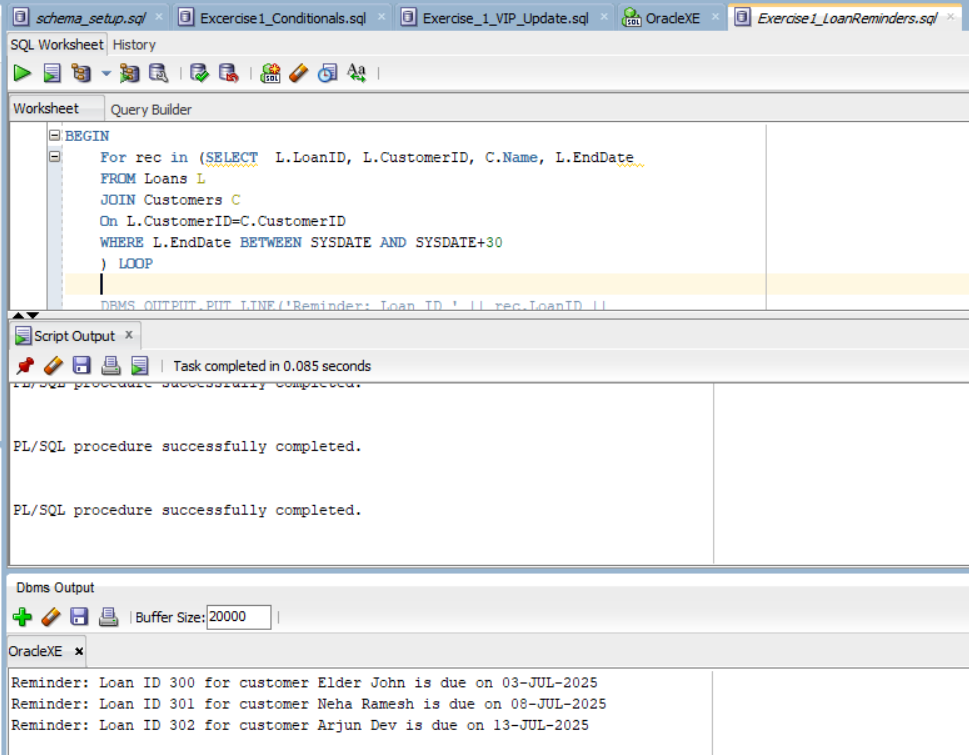
                             ' is due on ' || TO\_CHAR(rec.EndDate, 'DD-MON-YYYY'));

     END LOOP;

    END;

/

Output



# Exercise: 2: Error Handling

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

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

CREATE OR REPLACE PROCEDURE SafeTransferFunds (

    p\_from\_account IN NUMBER,

    p\_to\_account IN NUMBER,

    p\_amount IN NUMBER

) IS

    v\_balance NUMBER;

BEGIN

    -- Check balance of source account

    SELECT Balance INTO v\_balance

    FROM Accounts

    WHERE AccountID = p\_from\_account;

    -- Raise error if balance is insufficient

    IF v\_balance < p\_amount THEN

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

    END IF;

    -- Deduct from source account

    UPDATE Accounts

    SET Balance = Balance - p\_amount

    WHERE AccountID = p\_from\_account;

    -- Add to destination account

    UPDATE Accounts

    SET Balance = Balance + p\_amount

    WHERE AccountID = p\_to\_account;

    COMMIT;

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

EXCEPTION

    WHEN NO\_DATA\_FOUND THEN

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

        ROLLBACK;

    WHEN OTHERS THEN

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

        ROLLBACK;

END;

/

BEGIN

    SafeFundsTransfer(1, 2, 1000);

END;

/

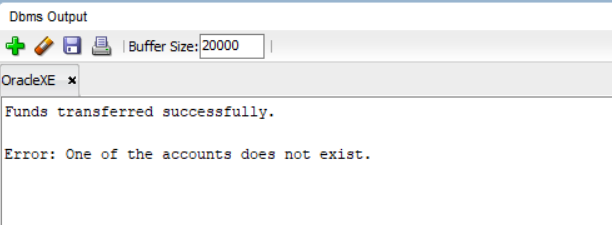
BEGIN

    SafeFundsTransfer(3, 2, 1000);

END;

/

Output:



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

**Question:** Write a stored procedure **UpdateSalary** that increases the salary of an employee by a given percentage. If the employee ID does not exist, handle the exception and log an error message.

CREATE OR REPLACE PROCEDURE UpdateSalary (

    p\_emp\_id IN NUMBER,

    p\_percent IN NUMBER

) IS

BEGIN

    -- Update salary with percentage

    UPDATE Employees

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

    WHERE EmployeeID = p\_emp\_id;

    -- Raise error if no employee found

    IF SQL%ROWCOUNT = 0 THEN

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

    END IF;

    COMMIT;

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

EXCEPTION

    WHEN OTHERS THEN

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

        ROLLBACK;

END;

/

BEGIN

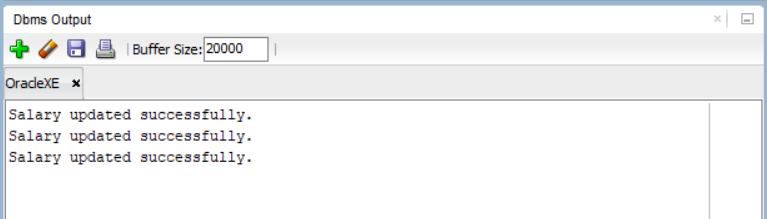
    UpdateSalary(4, 15);  -- Meera

    UpdateSalary(5, 5);   -- Rahul

    UpdateSalary(6, 0);   -- Divya

END;

/

Output: 

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

**Question:** Write a stored procedure **AddNewCustomer** that inserts a new customer into the Customers table. If a customer with the same ID already exists, handle the exception by logging an error and preventing the insertion.

CREATE OR REPLACE PROCEDURE AddNewCustomer(

    p\_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\_id,p\_name,p\_dob,p\_balance,SYSDATE);

    COMMIT;

    DBMS\_OUTPUT.PUT\_LINE('Customer Added Successfully');

EXCEPTION

    WHEN DUP\_VAL\_ON\_INDEX THEN

    DBMS\_OUTPUT.PUT\_LINE('Error: A customer with the same ID already exists.');

    ROLLBACK;

    WHEN OTHERS THEN

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

    ROLLBACK;

END;

/

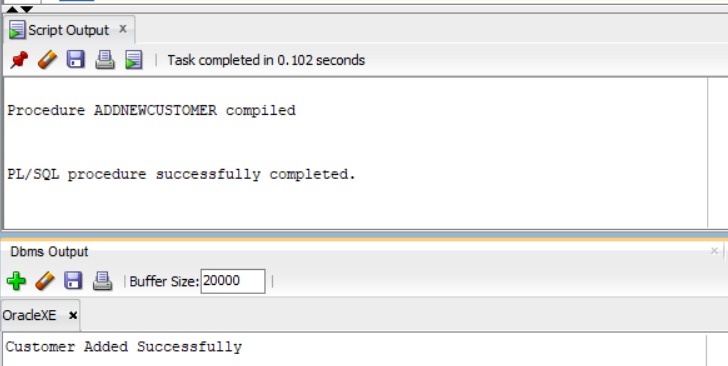
BEGIN

    AddNewCustomer(121, 'Deepa Menon', TO\_DATE('1993-07-24', 'YYYY-MM-DD'), 9200);

END;

/

Output:



# Exercise: 3: Stored Procedures

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

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

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

BEGIN

     UPDATE Accounts

     SET BALANCE = BALANCE+(BALANCE\*0.01)

     WHERE AccountType='Savings';

     COMMIT;

     DBMS\_OUTPUT.PUT\_LINE('Monthly Interest Applied to all Savings Account');

EXCEPTION

     WHEN OTHERS THEN

           DBMS\_OUTPUT.PUT\_LINE('Error applying monthly Interest'||SQLERRM);

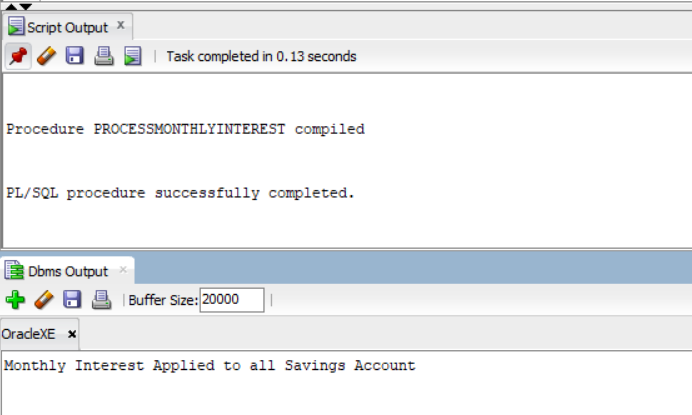
           ROLLBACK;

END;

/

EXEC ProcessMonthlyInterest;

Output:



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

**Question:** Write a stored procedure **UpdateEmployeeBonus** that updates the salary of employees in a given department by adding a bonus percentage passed as a parameter.

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (

    p\_emp\_id IN NUMBER,

    p\_bonus\_percent IN NUMBER

) IS

BEGIN

    UPDATE Employees

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

    WHERE EmployeeID = p\_emp\_id;

    IF SQL%ROWCOUNT = 0 THEN

        DBMS\_OUTPUT.PUT\_LINE('No employee found with ID: ' || p\_emp\_id);

    ELSE

        DBMS\_OUTPUT.PUT\_LINE('Bonus applied to employee ID: ' || p\_emp\_id);

    END IF;

    COMMIT;

EXCEPTION

    WHEN OTHERS THEN

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

        ROLLBACK;

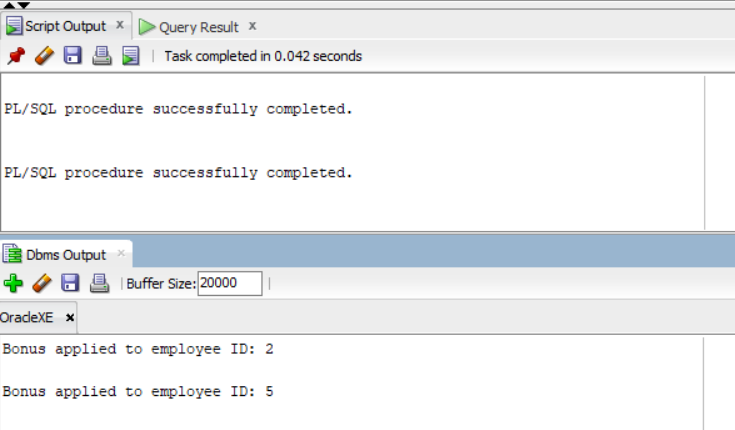
END;

/

EXEC UpdateEmployeeBonus(2, 10);

EXEC UpdateEmployeeBonus(5, 10);

Output:



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

**Question:** Write a stored procedure **TransferFunds** that transfers a specified amount from one account to another, checking that the source account has sufficient balance before making the transfer.

CREATE OR REPLACE PROCEDURE TransferFunds (

    p\_from\_account IN NUMBER,

    p\_to\_account IN NUMBER,

    p\_amount IN NUMBER

) IS

    v\_balance NUMBER;

    v\_trans\_id NUMBER;

BEGIN

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

    IF v\_balance < p\_amount THEN

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

    END IF;

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

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

    SELECT NVL(MAX(TransactionID), 0) + 1 INTO v\_trans\_id FROM Transactions;

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

    VALUES (v\_trans\_id, p\_from\_account, SYSDATE, p\_amount, 'DEBIT');

    SELECT v\_trans\_id + 1 INTO v\_trans\_id FROM DUAL;

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

    VALUES (v\_trans\_id, p\_to\_account, SYSDATE, p\_amount, 'CREDIT');

    COMMIT;

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

EXCEPTION

    WHEN NO\_DATA\_FOUND THEN

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

        ROLLBACK;

    WHEN OTHERS THEN

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

        ROLLBACK;

END;

/

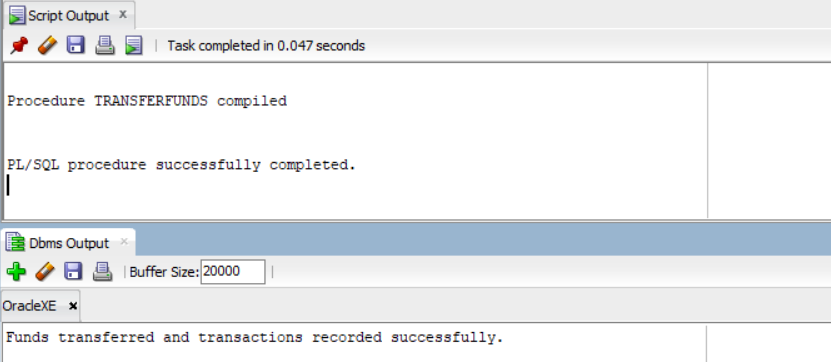
BEGIN

    TransferFunds(2, 1, 10);

END;

/

Output:



# Exercise: 4: Functions

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

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

CREATE OR REPLACE FUNCTION CalculateAge(p\_dob DATE)

RETURN NUMBER IS

    v\_age NUMBER;

BEGIN

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

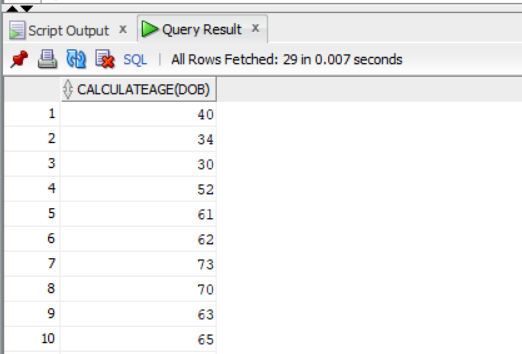
    DBMS\_OUTPUT.PUT\_LINE('All Customer Ages are computed and Returned');

    RETURN v\_age;

END;

/

Output:



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

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

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(

         p\_loan\_amount NUMBER,

         p\_annual\_rate NUMBER,

         p\_years NUMBER

) RETURN NUMBER IS

     v\_monthly\_rate NUMBER;

     v\_months NUMBER;

     v\_installment NUMBER;

BEGIN

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

    v\_months:=p\_years\*12;

    IF v\_monthly\_rate = 0 THEN

        v\_installment:=p\_loan\_amount/v\_months;

    ELSE

        v\_installment := (p\_loan\_amount \* v\_monthly\_rate) /

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

    END IF;

    RETURN ROUND(v\_installment,2);

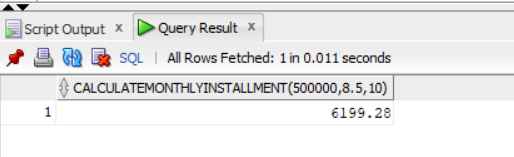
   END;

/

SELECT CalculateMonthlyInstallment(500000, 8.5, 10) FROM DUAL;

SELECT CalculateMonthlyInstallment(500000, 8.5, 6) FROM DUAL;

Output:



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

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

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;

    IF v\_balance >= p\_amount THEN

        DBMS\_OUTPUT.PUT\_LINE('Sufficient balance available.');

        RETURN TRUE;

    ELSE

        DBMS\_OUTPUT.PUT\_LINE('Insufficient balance.');

        RETURN FALSE;

    END IF;

EXCEPTION

    WHEN NO\_DATA\_FOUND THEN

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

        RETURN FALSE;

    WHEN OTHERS THEN

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

        RETURN FALSE;

END;

/

DECLARE

    v\_result BOOLEAN;

BEGIN

    v\_result := HasSufficientBalance(1, 5);

    IF v\_result THEN

        DBMS\_OUTPUT.PUT\_LINE(' Function returned: TRUE');

    ELSE

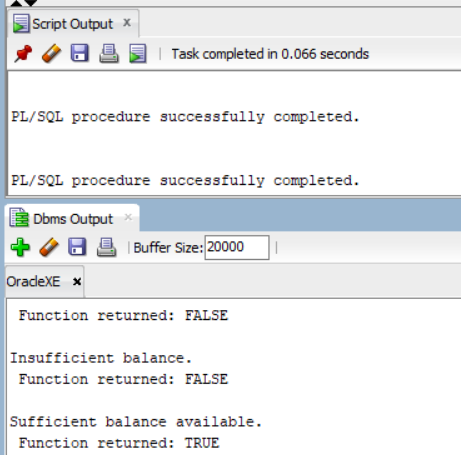
        DBMS\_OUTPUT.PUT\_LINE(' Function returned: FALSE');

    END IF;

END;

/

Output:



# Exercise: 5: Triggers

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

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

CREATE OR REPLACE TRIGGER UpdateCustomerModified

BEFORE UPDATE ON Customers

FOR EACH ROW

BEGIN

     :NEW.LastMODIFIED := SYSDATE;

END;

/

UPDATE Customers

SET Name = 'Updated Name'

WHERE CustomerID = 1;

SELECT Name,LastModified FROM Customers WHERE CustomerID = 1;

DECLARE

    v\_result BOOLEAN;

BEGIN

    v\_result := HasSufficientBalance(1, 5);

    IF v\_result THEN

        DBMS\_OUTPUT.PUT\_LINE(' Function returned: TRUE');

    ELSE

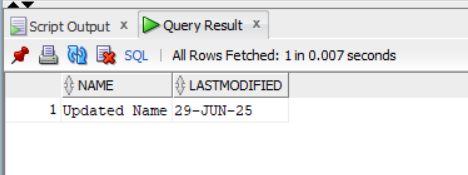
        DBMS\_OUTPUT.PUT\_LINE(' Function returned: FALSE');

    END IF;

END;

/

Output:



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

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

CREATE TABLE AuditLogs (

    LogID NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,

    TransactionID NUMBER,

    AccountID NUMBER,

    TransactionDate DATE,

    Amount NUMBER,

    TransactionType VARCHAR2(10),

    LoggedAt DATE DEFAULT SYSDATE

);

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

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

    VALUES (:NEW.TransactionID, :NEW.AccountID, :NEW.TransactionDate, :NEW.Amount, :NEW.TransactionType);

END;

/

SELECT \* FROM Accounts WHERE AccountID = 410;

-- Create customer first if not exists

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

VALUES (101, 'John Tester', TO\_DATE('1980-05-10', 'YYYY-MM-DD'), 5000, SYSDATE);

-- Now create the account

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

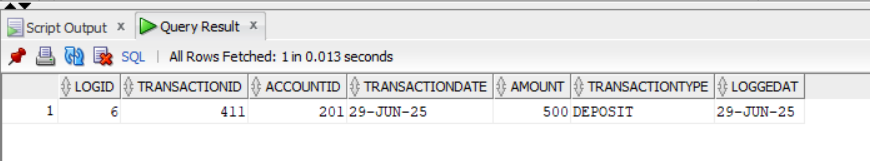
VALUES (201, 101, 'Savings', 10000, SYSDATE);

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

VALUES (411, 201, SYSDATE, 500, 'DEPOSIT');

SELECT \* FROM AuditLogs;

Output:



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

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

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 = 'WITHDRAW' AND :NEW.Amount > v\_balance THEN

        RAISE\_APPLICATION\_ERROR(-20001, 'Withdrawal exceeds available balance.');

    ELSIF :NEW.TransactionType = 'DEPOSIT' AND :NEW.Amount <= 0 THEN

        RAISE\_APPLICATION\_ERROR(-20002, 'Deposit amount must be positive.');

    END IF;

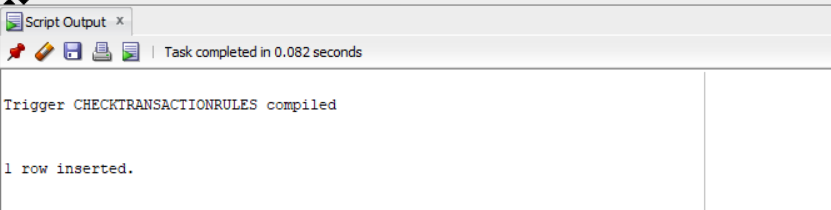
END;

/

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

VALUES (103, 201, SYSDATE, 1000, 'DEPOSIT');

Output:



# Exercise: 6: Cursors

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

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

DECLARE

      CURSOR trans\_cursor IS

      SELECT C.CustomerID, C.Name, T.TransactionDate, T.Amount, T.TransactionType

      FROM Customers C

      JOIN Accounts A ON C.CustomerID=A.CustomerID

      JOIN Transactions T ON A.AccountID=T.AccountID

      WHERE TRUNC(T.TransactionDate,'MM') = TRUNC(SYSDATE,'MM');

BEGIN

FOR rec IN trans\_cursor LOOP

       DBMS\_OUTPUT.PUT\_LINE('Customer: ' || rec.Name ||

                             ' | Date: ' || TO\_CHAR(rec.TransactionDate, 'DD-MON-YYYY') ||

                             ' | Type: ' || rec.TransactionType ||

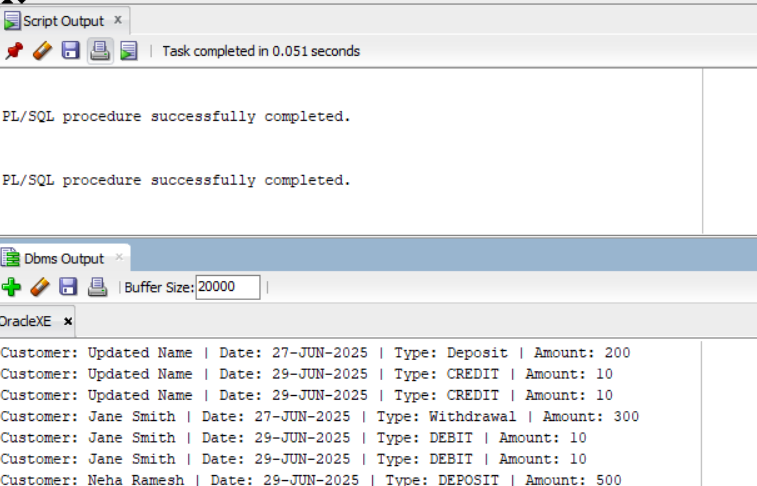
                             ' | Amount: ' || rec.Amount);

    END LOOP;

END;

/

Output:



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

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

DECLARE

     CURSOR acc\_cursor IS

        SELECT AccountID,Balance From Accounts;

BEGIN

    For rec in acc\_cursor LOOP

        UPDATE Accounts

        SET BALANCE=BALANCE-250

        WHERE AccountID=rec.AccountID;

        DBMS\_OUTPUT.PUT\_LINE('Annual Fee of 250 applied to the Account ID:'||rec.AccountID);

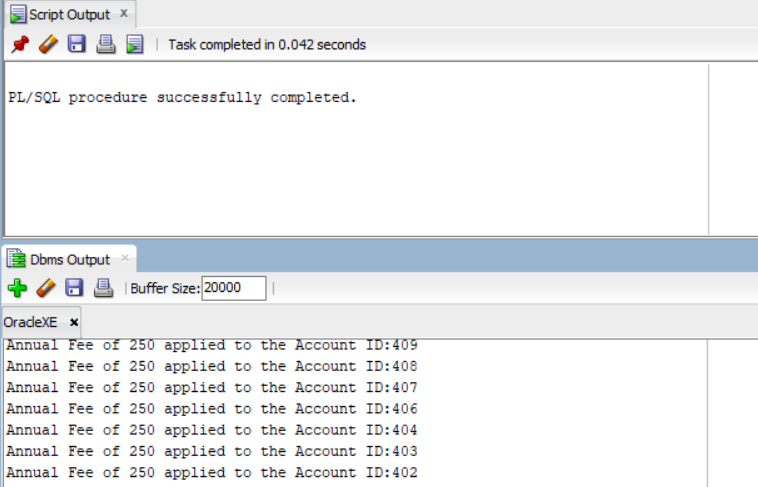
        END LOOP;

        COMMIT;

    END;

/

Output:



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

**Question:** Write a PL/SQL block using an explicit cursor **UpdateLoanInterestRates** that fetches all loans and updates their interest rates based on the new policy.

DECLARE

      CURSOR loan\_cursor IS

            SELECT LoanID,InterestRate From Loans;

BEGIN

     For rec in loan\_cursor LOOP

        UPDATE Loans

        SET InterestRate=rec.InterestRate+0.05

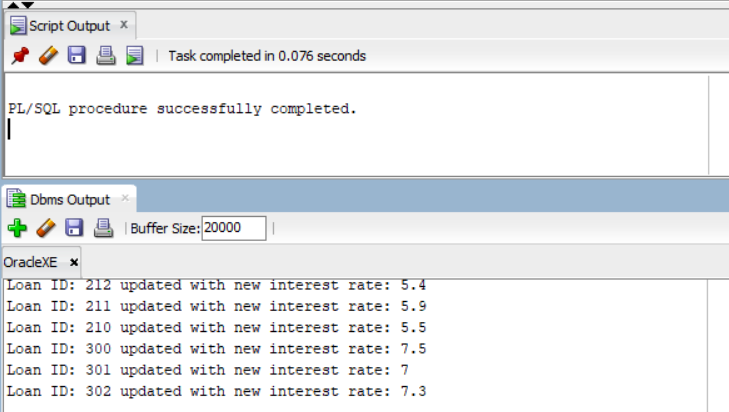
        WHERE LoanID=rec.LoanID;

        DBMS\_OUTPUT.PUT\_LINE('Loan ID: ' || rec.LoanID || ' updated with new interest rate: ' || (rec.InterestRate + 0.5));

    END LOOP;

END; /

Output:



# Exercise: 7: Packages

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

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

CREATE OR REPLACE PACKAGE CustomerManagement AS

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

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

    FUNCTION GetCustomerBalance(p\_id NUMBER) RETURN NUMBER;

END CustomerManagement;

/

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

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

    BEGIN

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

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

    END;

    PROCEDURE UpdateCustomer(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;

    END;

END CustomerManagement;

/

BEGIN

    CustomerManagement.AddCustomer(1002, 'Abi Ayshu', TO\_DATE('2000-05-10','YYYY-MM-DD'), 12000);

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

END;

/

BEGIN

    CustomerManagement.UpdateCustomer(1001, 'Ayshu Updated');

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

END;

/

DECLARE

    v\_balance NUMBER;

BEGIN

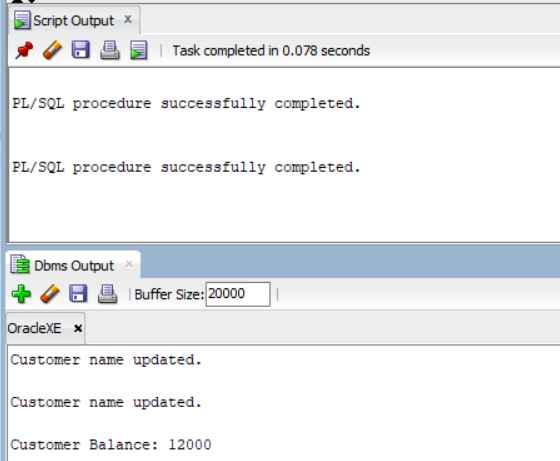
    v\_balance := CustomerManagement.GetCustomerBalance(101);

    DBMS\_OUTPUT.PUT\_LINE('Customer Balance: ' || v\_balance);

END;

/

Output:



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

**Question:** Write a package **EmployeeManagement** with procedures to hire new employees, update employee details, and a function to calculate annual salary.

CREATE OR REPLACE PACKAGE EmployeeManagement AS

    PROCEDURE HireEmployee(p\_id NUMBER, p\_name VARCHAR2, p\_position VARCHAR2, p\_salary NUMBER, p\_department VARCHAR2);

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

    FUNCTION CalculateAnnualSalary(p\_id NUMBER) RETURN NUMBER;

END EmployeeManagement;

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

    PROCEDURE HireEmployee(p\_id NUMBER, p\_name VARCHAR2, p\_position VARCHAR2, p\_salary NUMBER, p\_department VARCHAR2) IS

    BEGIN

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

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

    END;

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

    BEGIN

        UPDATE Employees

        SET Salary = p\_salary

        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;

    END;

END EmployeeManagement;

/

BEGIN

    EmployeeManagement.HireEmployee(

        p\_id         => 201,

        p\_name       => 'Abi Ayshu',

        p\_position   => 'Analyst',

        p\_salary     => 40000,

        p\_department => 'Tech'

    );

    DBMS\_OUTPUT.PUT\_LINE('Employee hired successfully.');

END;

/

BEGIN

    EmployeeManagement.UpdateEmployee(201, 45000);

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

END;

/

DECLARE

    v\_annual NUMBER;

BEGIN

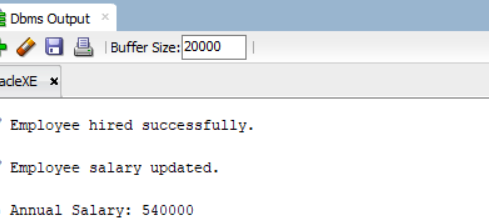
    v\_annual := EmployeeManagement.CalculateAnnualSalary(201);

    DBMS\_OUTPUT.PUT\_LINE('Annual Salary: ' || v\_annual);

END;

/

Output:



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

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

CREATE OR REPLACE PACKAGE AccountOperations AS

    PROCEDURE OpenAccount(p\_acc\_id NUMBER, p\_cust\_id NUMBER, p\_type VARCHAR2, p\_balance NUMBER);

    PROCEDURE CloseAccount(p\_acc\_id NUMBER);

    FUNCTION GetTotalCustomerBalance(p\_cust\_id NUMBER) RETURN NUMBER;

END AccountOperations;

/

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

    PROCEDURE OpenAccount(p\_acc\_id NUMBER, p\_cust\_id NUMBER, p\_type VARCHAR2, p\_balance NUMBER) IS

    BEGIN

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

        VALUES (p\_acc\_id, p\_cust\_id, p\_type, p\_balance, SYSDATE);

    END;

    PROCEDURE CloseAccount(p\_acc\_id NUMBER) IS

    BEGIN

        DELETE FROM Accounts

        WHERE AccountID = p\_acc\_id;

    END;

    FUNCTION GetTotalCustomerBalance(p\_cust\_id NUMBER) RETURN NUMBER IS

        v\_total NUMBER;

    BEGIN

        SELECT SUM(Balance) INTO v\_total FROM Accounts WHERE CustomerID = p\_cust\_id;

        RETURN NVL(v\_total, 0);

    END;

END AccountOperations;

/

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

VALUES (210, 'Riya Kapoor', TO\_DATE('1995-07-12', 'YYYY-MM-DD'), 0, SYSDATE);

BEGIN

    AccountOperations.OpenAccount(

        p\_acc\_id   => 302,

        p\_cust\_id  => 210,

        p\_type     => 'Current',

        p\_balance  => 25000

    );

    DBMS\_OUTPUT.PUT\_LINE(' Account 302 opened for customer 210.');

END;

/

DECLARE

    v\_total NUMBER;

BEGIN

    v\_total := AccountOperations.GetTotalCustomerBalance(202);

    DBMS\_OUTPUT.PUT\_LINE('Total Balance for Customer 202: ' || v\_total);

END;

/

BEGIN

    AccountOperations.CloseAccount(302);

    DBMS\_OUTPUT.PUT\_LINE('️ Account 302 closed successfully.');

END;

/

Output:

