**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.

**Solution:**

BEGIN

    FOR cust IN (

        SELECT CustomerID, FLOOR(MONTHS\_BETWEEN(SYSDATE, DOB)/12) AS Age

        FROM Customers

    ) LOOP

        IF cust.Age > 60 THEN

            UPDATE Loans

            SET InterestRate = InterestRate - 1

            WHERE CustomerID = cust.CustomerID;

        END IF;

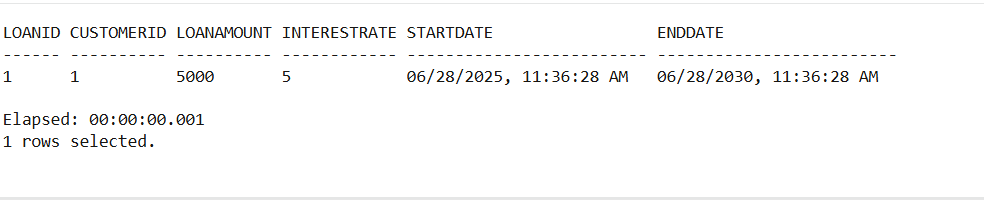
    END LOOP;

END;

/

SELECT \* FROM Loans;

**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.

**Solution:**

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;

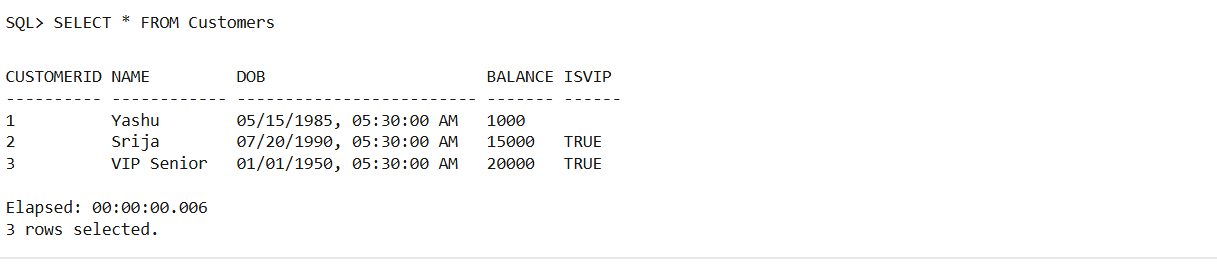
  COMMIT;

END;

/

SELECT \* FROM Customers;

**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.

**Solution:**

BEGIN

  FOR loan\_rec IN (

    SELECT L.LoanID, L.CustomerID, L.DueDate, C.Name

    FROM Loans L

    JOIN Customers C ON L.CustomerID = C.CustomerID

    WHERE L.DueDate BETWEEN SYSDATE AND SYSDATE + 30

  ) LOOP

    DBMS\_OUTPUT.PUT\_LINE(

      'Reminder: Customer ' || loan\_rec.Name ||

      ' (CustomerID: ' || loan\_rec.CustomerID ||

      ') has a loan due on ' || TO\_CHAR(loan\_rec.DueDate, 'DD-MON-YYYY')

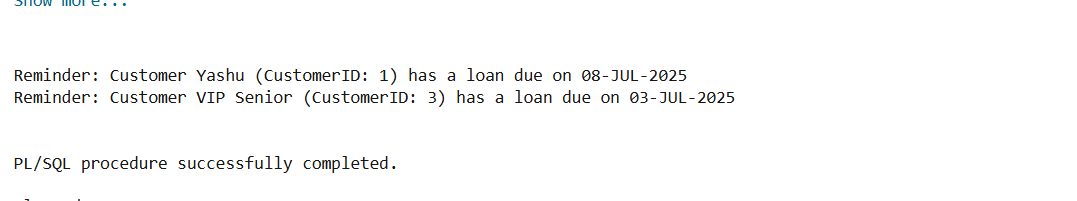
    );

  END LOOP;

END;

/

SET SERVEROUTPUT ON;

****

**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.

**Solution:**

CREATE OR REPLACE PROCEDURE SafeTransferFunds(

    FromAccID IN NUMBER,

    ToAccID IN NUMBER,

    Amount IN NUMBER

) AS

    InsufficientFunds EXCEPTION;

    v\_balance NUMBER;

BEGIN

    SELECT Balance INTO v\_balance FROM Accounts WHERE AccountID = FromAccID;

    IF v\_balance < Amount THEN

        RAISE InsufficientFunds;

    END IF;

    UPDATE Accounts SET Balance = Balance - Amount WHERE AccountID = FromAccID;

    UPDATE Accounts SET Balance = Balance + Amount WHERE AccountID = ToAccID;

    COMMIT;

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

EXCEPTION

    WHEN InsufficientFunds THEN

        ROLLBACK;

        DBMS\_OUTPUT.PUT\_LINE('Error: Insufficient funds for transfer.');

    WHEN OTHERS THEN

        ROLLBACK;

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

END;

/

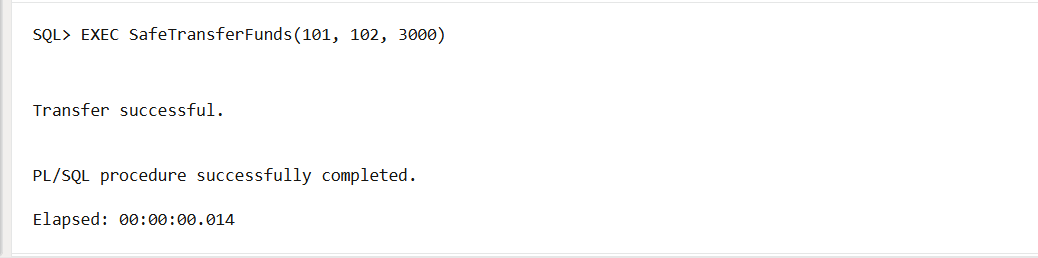
SET SERVEROUTPUT ON;

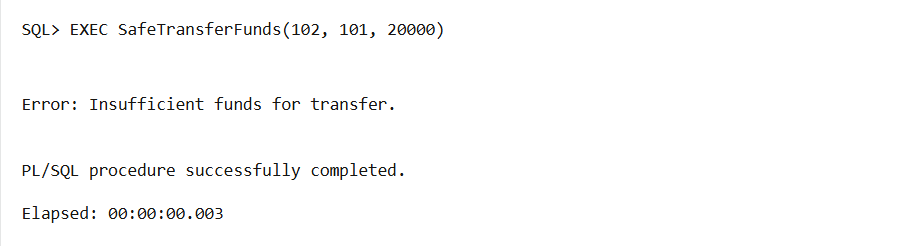
SELECT \* FROM Accounts;

EXEC SafeTransferFunds(101, 102, 3000);

EXEC SafeTransferFunds(102, 101, 20000);

**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.

**Solution:**

CREATE OR REPLACE PROCEDURE UpdateSalary(

    emp\_id IN NUMBER,

    percent IN NUMBER

) AS

BEGIN

    UPDATE Employees

    SET Salary = Salary + (Salary \* percent / 100)

    WHERE EmpID = emp\_id;

    IF SQL%ROWCOUNT = 0 THEN

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

    ELSE

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

        COMMIT;

    END IF;

EXCEPTION

    WHEN OTHERS THEN

        ROLLBACK;

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

END;

/

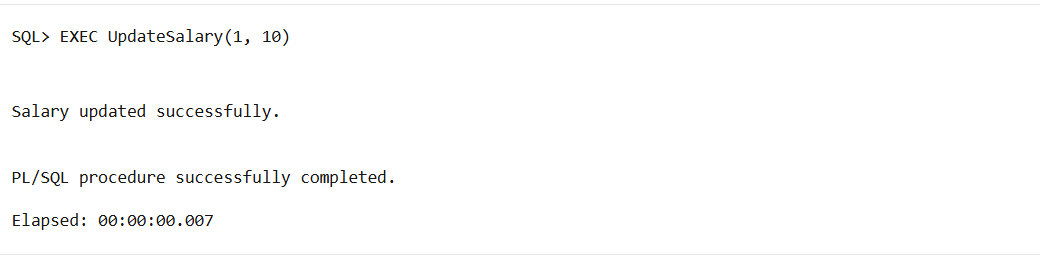
SET SERVEROUTPUT ON;

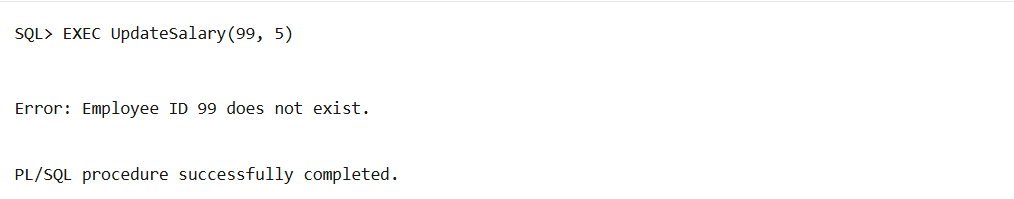
SELECT \* FROM Employees;

EXEC UpdateSalary(1, 10);

EXEC UpdateSalary(99, 5);

**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.

**Solution:**

CREATE TABLE Customers (

    CustomerID NUMBER PRIMARY KEY,

    Name VARCHAR2(100),

    DOB DATE,

    Balance NUMBER,

    IsVIP VARCHAR2(5)

);

CREATE OR REPLACE PROCEDURE AddNewCustomer(

    p\_id       IN NUMBER,

    p\_name     IN VARCHAR2,

    p\_dob      IN DATE,

    p\_balance  IN NUMBER,

    p\_isvip    IN VARCHAR2

) AS

BEGIN

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

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

    COMMIT;

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

EXCEPTION

    WHEN DUP\_VAL\_ON\_INDEX THEN

        ROLLBACK;

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

    WHEN OTHERS THEN

        ROLLBACK;

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

END;

/

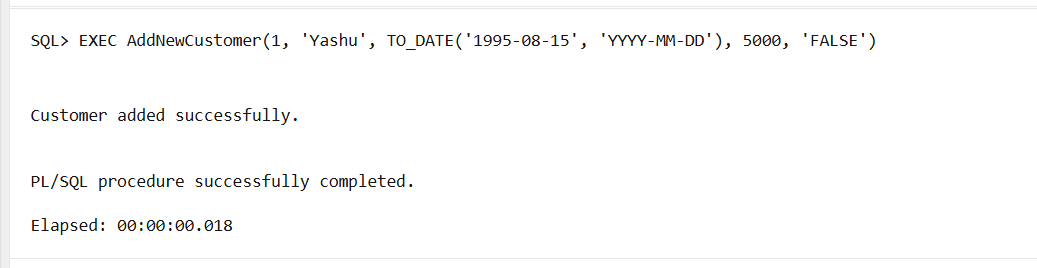
SET SERVEROUTPUT ON;

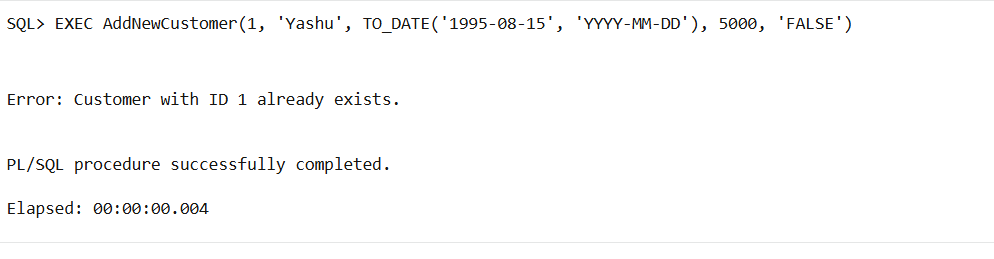
EXEC AddNewCustomer(1, 'Yashu', TO\_DATE('1995-08-15', 'YYYY-MM-DD'), 5000, 'FALSE');

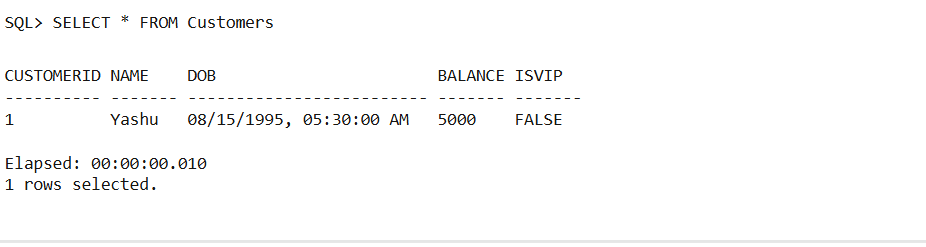
EXEC AddNewCustomer(1, 'Yashu', TO\_DATE('1995-08-15', 'YYYY-MM-DD'), 5000, 'FALSE');

SELECT \* FROM Customers;

**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.

**Solution:**

CREATE TABLE SavingsAccounts (

    AccountID NUMBER PRIMARY KEY,

    Balance NUMBER

);

INSERT INTO SavingsAccounts VALUES (201, 10000);

INSERT INTO SavingsAccounts VALUES (202, 5000);

COMMIT;

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

BEGIN

  UPDATE SavingsAccounts

  SET Balance = Balance + (Balance \* 0.01);

  COMMIT;

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

EXCEPTION

  WHEN OTHERS THEN

    ROLLBACK;

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

END;

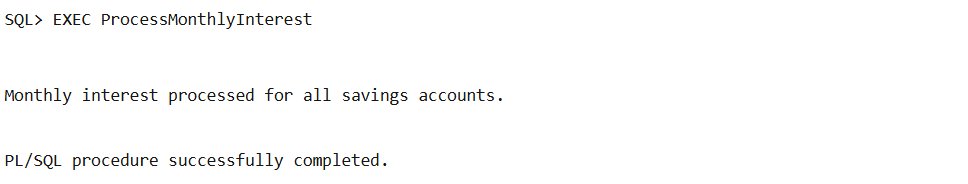
/

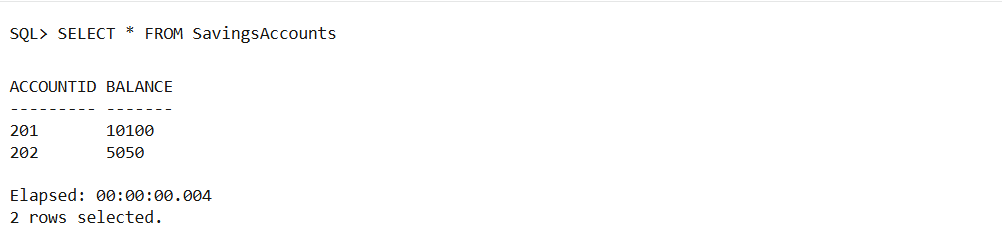
SET SERVEROUTPUT ON;

EXEC ProcessMonthlyInterest;

SELECT \* FROM SavingsAccounts;

**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.

**Solution:**

CREATE TABLE Employees (

    EmpID NUMBER PRIMARY KEY,

    Name VARCHAR2(100),

    Department VARCHAR2(50),

    Salary NUMBER

);

INSERT INTO Employees VALUES (1, 'Yashu', 'Sales', 30000);

INSERT INTO Employees VALUES (2, 'Srija', 'Sales', 40000);

INSERT INTO Employees VALUES (3, 'Anu', 'IT', 50000);

COMMIT;

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(

    p\_dept IN VARCHAR2,

    p\_bonus\_percent IN NUMBER

) AS

BEGIN

  UPDATE Employees

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

  WHERE Department = p\_dept;

  IF SQL%ROWCOUNT = 0 THEN

    DBMS\_OUTPUT.PUT\_LINE('No employees found in department: ' || p\_dept);

  ELSE

    DBMS\_OUTPUT.PUT\_LINE(SQL%ROWCOUNT || ' employee(s) received bonus in ' || p\_dept);

  END IF;

  COMMIT;

EXCEPTION

  WHEN OTHERS THEN

    ROLLBACK;

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

END;

/

SET SERVEROUTPUT ON;

EXEC UpdateEmployeeBonus('Sales', 10);

SELECT \* FROM Employees;

**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.

**Solution:**

CREATE TABLE Accounts (

    AccountID NUMBER PRIMARY KEY,

    Balance NUMBER

);

INSERT INTO Accounts VALUES (101, 10000);

INSERT INTO Accounts VALUES (102, 5000);

COMMIT;

CREATE OR REPLACE PROCEDURE TransferFunds(

    p\_from\_acc IN NUMBER,

    p\_to\_acc IN NUMBER,

    p\_amount IN NUMBER

) AS

    v\_balance NUMBER;

    InsufficientFunds EXCEPTION;

BEGIN

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

    IF v\_balance < p\_amount THEN

        RAISE InsufficientFunds;

    END IF;

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

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

    COMMIT;

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

EXCEPTION

    WHEN InsufficientFunds THEN

        ROLLBACK;

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

    WHEN OTHERS THEN

        ROLLBACK;

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

END;

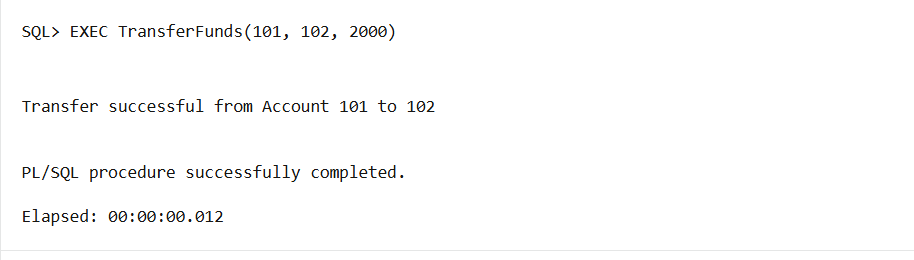
/

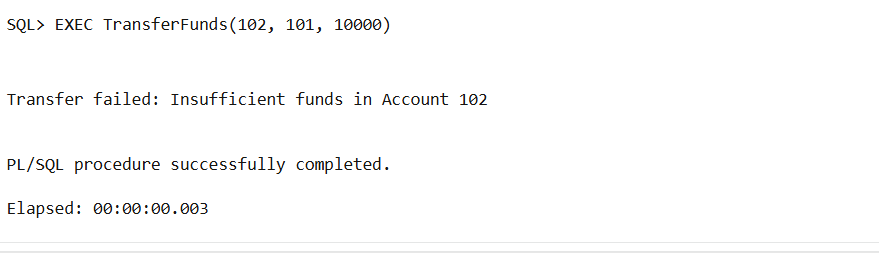
EXEC TransferFunds(101, 102, 2000);

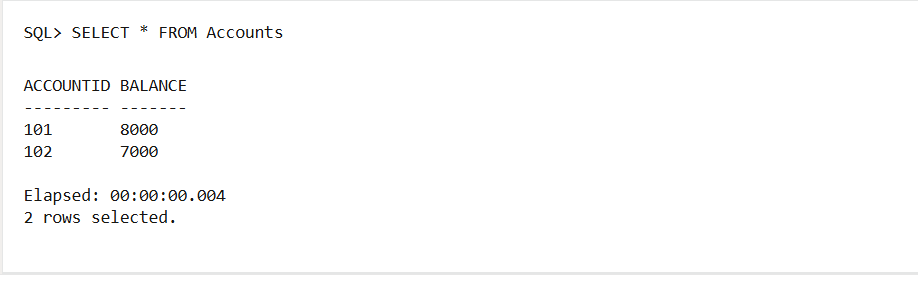
EXEC TransferFunds(102, 101, 10000);

SELECT \* FROM Accounts;

**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.

**Solution:**

CREATE TABLE Customers (

    CustomerID NUMBER PRIMARY KEY,

    Name VARCHAR2(100),

    DOB DATE

);

INSERT INTO Customers VALUES (1, 'Yashu', TO\_DATE('2000-06-28', 'YYYY-MM-DD'));

INSERT INTO Customers VALUES (2, 'Srija', TO\_DATE('1985-01-01', 'YYYY-MM-DD'));

COMMIT;

CREATE OR REPLACE FUNCTION CalculateAge(p\_dob DATE)

RETURN NUMBER IS

    v\_age NUMBER;

BEGIN

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

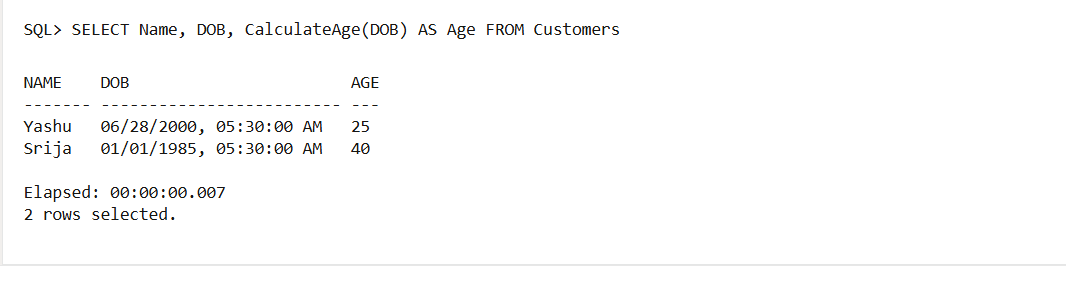
    RETURN v\_age;

END;

/

SELECT Name, DOB, CalculateAge(DOB) AS Age FROM Customers;

**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.

**Solution:**

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(

    p\_loan\_amount NUMBER,

    p\_annual\_rate NUMBER,

    p\_years NUMBER

) RETURN NUMBER IS

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

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

    v\_emi NUMBER;

BEGIN

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

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

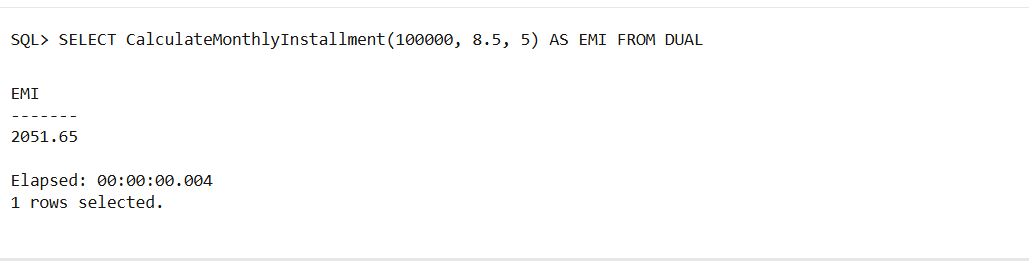
    RETURN ROUND(v\_emi, 2);

END;

/

SELECT CalculateMonthlyInstallment(100000, 8.5, 5) AS EMI 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.

**Solution:**

CREATE TABLE Accounts (

    AccountID NUMBER PRIMARY KEY,

    Balance NUMBER

);

INSERT INTO Accounts VALUES (101, 10000);

INSERT INTO Accounts VALUES (102, 3000);

COMMIT;

CREATE OR REPLACE FUNCTION HasSufficientBalance(

    p\_acc\_id NUMBER,

    p\_amount NUMBER

) RETURN BOOLEAN IS

    v\_balance NUMBER;

BEGIN

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

    RETURN v\_balance >= p\_amount;

EXCEPTION

    WHEN NO\_DATA\_FOUND THEN

        RETURN FALSE;

END;

/

DECLARE

    result BOOLEAN;

BEGIN

    result := HasSufficientBalance(101, 5000);

    IF result THEN

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

    ELSE

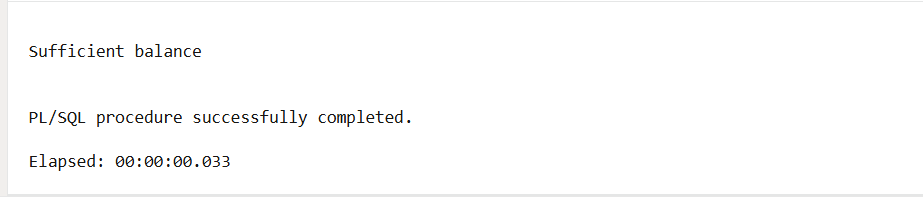
        DBMS\_OUTPUT.PUT\_LINE('Insufficient balance');

    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.

**Solution:**

CREATE TABLE Customers (

    CustomerID NUMBER PRIMARY KEY,

    Name VARCHAR2(100),

    Balance NUMBER,

    LastModified DATE

);

INSERT INTO Customers VALUES (1, 'Yashu', 5000, SYSDATE);

COMMIT;

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON Customers

FOR EACH ROW

BEGIN

    :NEW.LastModified := SYSDATE;

END;

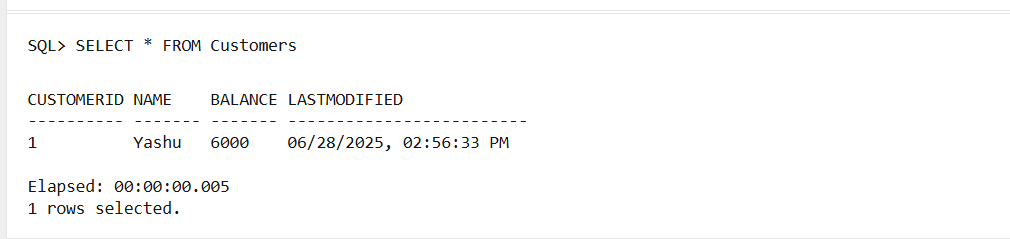
/

UPDATE Customers SET Balance = 6000 WHERE CustomerID = 1;

COMMIT;

SELECT \* FROM Customers;

**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.

**Solution:**

CREATE TABLE Transactions (

    TxnID NUMBER PRIMARY KEY,

    AccountID NUMBER,

    TxnType VARCHAR2(20), -- 'DEPOSIT' or 'WITHDRAW'

    Amount NUMBER

);

CREATE TABLE AuditLog (

    LogID NUMBER GENERATED ALWAYS AS IDENTITY,

    TxnID NUMBER,

    ActionDate DATE,

    Description VARCHAR2(200)

);

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

    INSERT INTO AuditLog (TxnID, ActionDate, Description)

    VALUES (:NEW.TxnID, SYSDATE, 'Transaction of type ' || :NEW.TxnType || ' for amount ' || :NEW.Amount);

END;

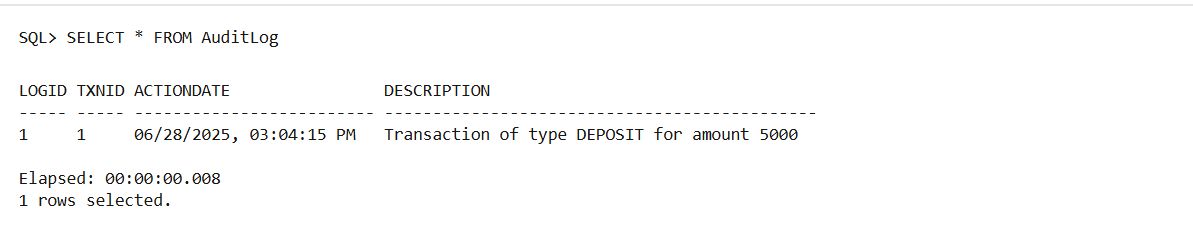
/

INSERT INTO Transactions VALUES (1, 101, 'DEPOSIT', 5000);

COMMIT;

SELECT \* FROM AuditLog;

**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.

**Solutions:**

CREATE TABLE Accounts (

    AccountID NUMBER PRIMARY KEY,

    Balance NUMBER

);

INSERT INTO Accounts VALUES (101, 10000);

COMMIT;

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

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

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

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

    END IF;

END;

/

INSERT INTO Transactions VALUES (2, 101, 'DEPOSIT', 3000);

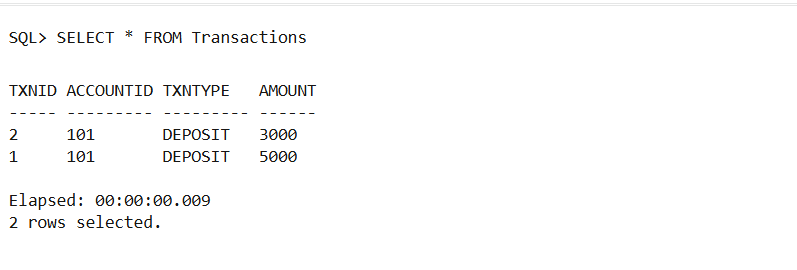
COMMIT;

INSERT INTO Transactions VALUES (3, 101, 'WITHDRAW', 20000);

INSERT INTO Transactions VALUES (4, 101, 'DEPOSIT', -500);

SELECT \* FROM Transactions;

**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.

**Solution:**

CREATE TABLE Customers (

    CustomerID NUMBER PRIMARY KEY,

    Name VARCHAR2(100)

);

CREATE TABLE Transactions (

    TxnID NUMBER PRIMARY KEY,

    CustomerID NUMBER,

    TxnDate DATE,

    TxnType VARCHAR2(20),

    Amount NUMBER

);

INSERT INTO Customers VALUES (1, 'Yashu');

INSERT INTO Customers VALUES (2, 'Srija');

INSERT INTO Transactions VALUES (101, 1, SYSDATE, 'DEPOSIT', 1000);

INSERT INTO Transactions VALUES (102, 1, SYSDATE, 'WITHDRAW', 500);

INSERT INTO Transactions VALUES (103, 2, SYSDATE, 'DEPOSIT', 1500);

COMMIT;

SET SERVEROUTPUT ON;

DECLARE

    CURSOR txn\_cursor IS

        SELECT C.Name, T.TxnType, T.Amount, T.TxnDate

        FROM Customers C

        JOIN Transactions T ON C.CustomerID = T.CustomerID

        WHERE TO\_CHAR(T.TxnDate, 'MMYYYY') = TO\_CHAR(SYSDATE, 'MMYYYY');

    v\_name Customers.Name%TYPE;

    v\_type Transactions.TxnType%TYPE;

    v\_amt  Transactions.Amount%TYPE;

    v\_date Transactions.TxnDate%TYPE;

BEGIN

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

    OPEN txn\_cursor;

    LOOP

        FETCH txn\_cursor INTO v\_name, v\_type, v\_amt, v\_date;

        EXIT WHEN txn\_cursor%NOTFOUND;

        DBMS\_OUTPUT.PUT\_LINE(v\_name || ' | ' || v\_type || ' | ₹' || v\_amt || ' on ' || TO\_CHAR(v\_date, 'DD-MON-YYYY'));

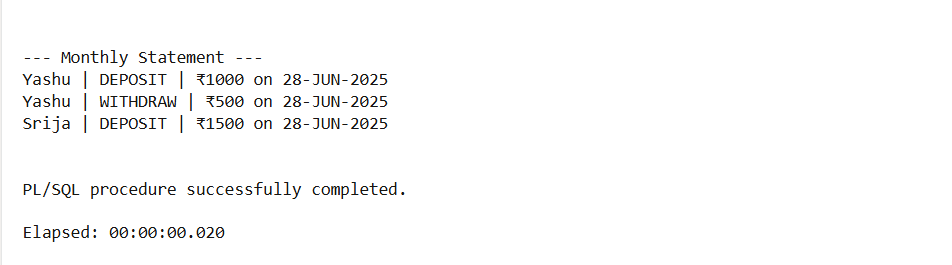
    END LOOP;

    CLOSE txn\_cursor;

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.

**Solution:**

CREATE TABLE Accounts (

    AccountID NUMBER PRIMARY KEY,

    Balance NUMBER

);

INSERT INTO Accounts VALUES (201, 10000);

INSERT INTO Accounts VALUES (202, 3000);

COMMIT;

SET SERVEROUTPUT ON;

DECLARE

    CURSOR acc\_cursor IS SELECT AccountID, Balance FROM Accounts;

    v\_id Accounts.AccountID%TYPE;

    v\_bal Accounts.Balance%TYPE;

BEGIN

    OPEN acc\_cursor;

    LOOP

        FETCH acc\_cursor INTO v\_id, v\_bal;

        EXIT WHEN acc\_cursor%NOTFOUND;

        UPDATE Accounts

        SET Balance = Balance - 500

        WHERE AccountID = v\_id;

        DBMS\_OUTPUT.PUT\_LINE('Annual fee of ₹500 applied to Account ' || v\_id);

    END LOOP;

    CLOSE acc\_cursor;

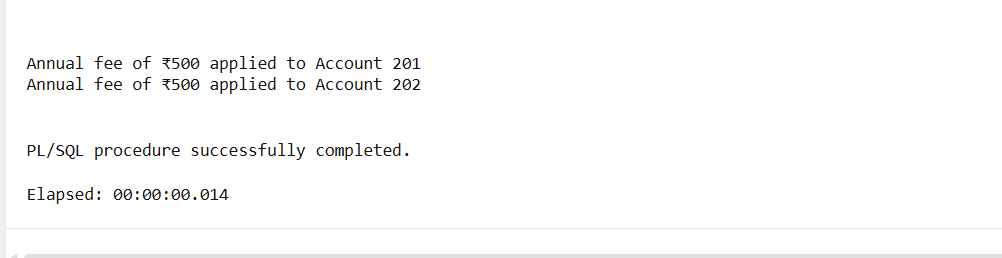
    COMMIT;

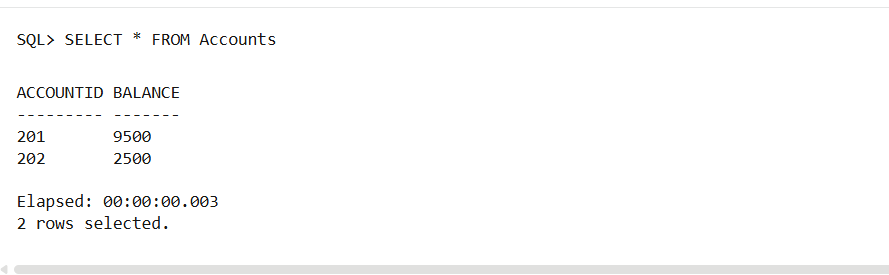
END;

/

SELECT \* FROM Accounts;

**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.

**Solution:**

CREATE TABLE Loans (

    LoanID NUMBER PRIMARY KEY,

    CustomerID NUMBER,

    InterestRate NUMBER

);

INSERT INTO Loans VALUES (301, 1, 8.0);

INSERT INTO Loans VALUES (302, 2, 9.5);

COMMIT;

SET SERVEROUTPUT ON;

DECLARE

    CURSOR loan\_cursor IS SELECT LoanID, InterestRate FROM Loans;

    v\_id Loans.LoanID%TYPE;

    v\_rate Loans.InterestRate%TYPE;

BEGIN

    OPEN loan\_cursor;

    LOOP

        FETCH loan\_cursor INTO v\_id, v\_rate;

        EXIT WHEN loan\_cursor%NOTFOUND;

        IF v\_rate < 9 THEN

            UPDATE Loans SET InterestRate = v\_rate + 0.5 WHERE LoanID = v\_id;

            DBMS\_OUTPUT.PUT\_LINE('Interest increased for Loan ' || v\_id);

        ELSE

            UPDATE Loans SET InterestRate = v\_rate - 0.25 WHERE LoanID = v\_id;

            DBMS\_OUTPUT.PUT\_LINE('Interest reduced for Loan ' || v\_id);

        END IF;

    END LOOP;

    CLOSE loan\_cursor;

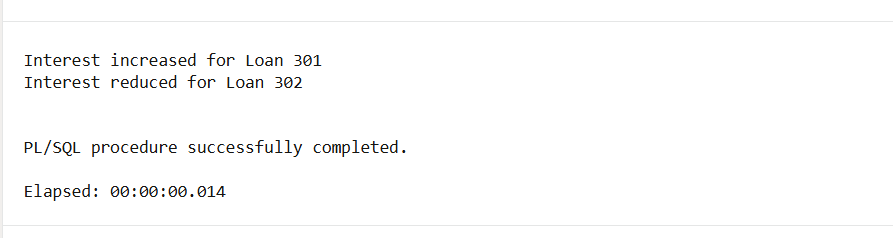
    COMMIT;

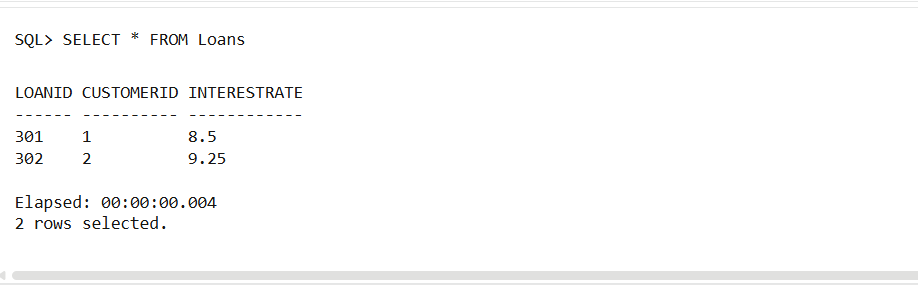
END;

/

SELECT \* FROM Loans;

**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.

**Solution:**

CREATE TABLE Customers (

    CustomerID NUMBER PRIMARY KEY,

    Name VARCHAR2(100),

    Balance NUMBER

);

CREATE OR REPLACE PACKAGE CustomerManagement AS

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

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

    FUNCTION GetBalance(p\_id NUMBER) RETURN NUMBER;

END CustomerManagement;

/

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

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

    BEGIN

        INSERT INTO Customers VALUES (p\_id, p\_name, p\_balance);

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

    END;

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

    BEGIN

        UPDATE Customers SET Name = p\_name WHERE CustomerID = p\_id;

        DBMS\_OUTPUT.PUT\_LINE('Customer updated to: ' || p\_name);

    END;

    FUNCTION GetBalance(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;

/

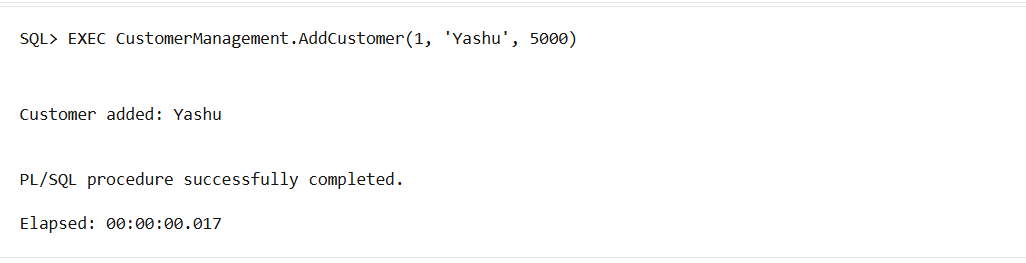
SET SERVEROUTPUT ON;

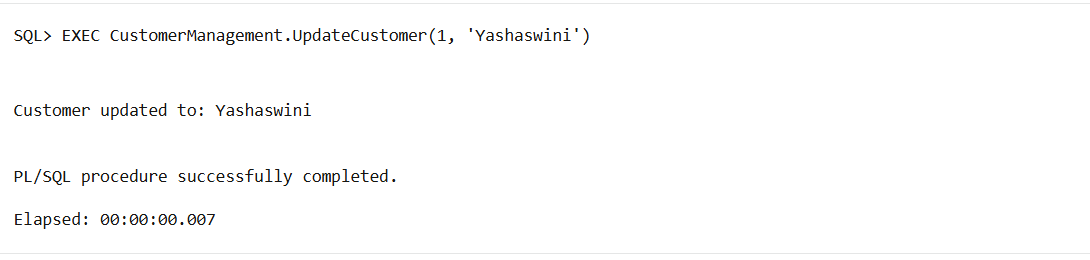
EXEC CustomerManagement.AddCustomer(1, 'Yashu', 5000);

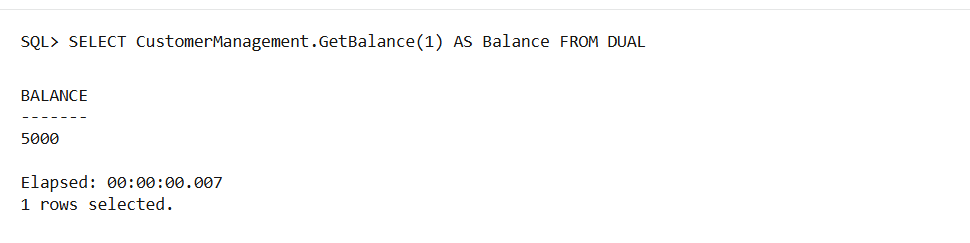
EXEC CustomerManagement.UpdateCustomer(1, 'Yashaswini');

SELECT CustomerManagement.GetBalance(1) AS Balance FROM DUAL;

**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.

**Solution:**

CREATE TABLE Employees (

    EmpID NUMBER PRIMARY KEY,

    Name VARCHAR2(100),

    Salary NUMBER

);

CREATE OR REPLACE PACKAGE EmployeeManagement AS

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

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

    FUNCTION AnnualSalary(p\_id NUMBER) RETURN NUMBER;

END EmployeeManagement;

/

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

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

    BEGIN

        INSERT INTO Employees VALUES (p\_id, p\_name, p\_salary);

        DBMS\_OUTPUT.PUT\_LINE('Employee hired: ' || p\_name);

    END;

    PROCEDURE UpdateEmployee(p\_id NUMBER, p\_name VARCHAR2) IS

    BEGIN

        UPDATE Employees SET Name = p\_name WHERE EmpID = p\_id;

        DBMS\_OUTPUT.PUT\_LINE('Employee updated to: ' || p\_name);

    END;

    FUNCTION AnnualSalary(p\_id NUMBER) RETURN NUMBER IS

        v\_salary NUMBER;

    BEGIN

        SELECT Salary INTO v\_salary FROM Employees WHERE EmpID = p\_id;

        RETURN v\_salary \* 12;

    END;

END EmployeeManagement;

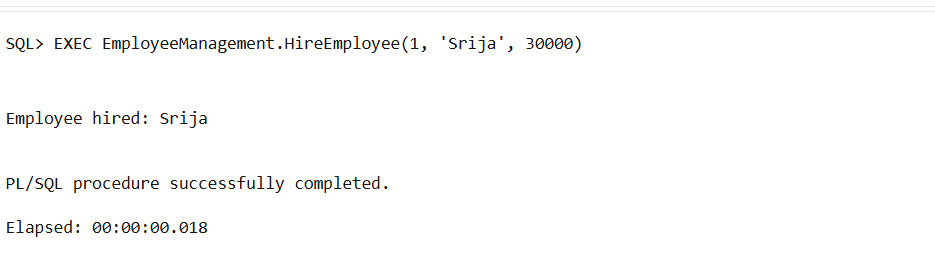
/

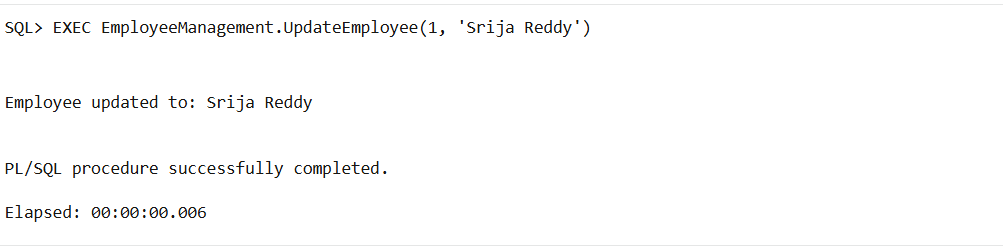
EXEC EmployeeManagement.HireEmployee(1, 'Srija', 30000);

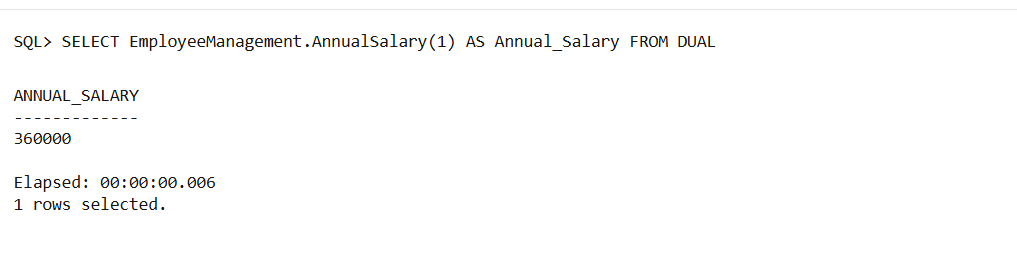
EXEC EmployeeManagement.UpdateEmployee(1, 'Srija Reddy');

SELECT EmployeeManagement.AnnualSalary(1) AS Annual\_Salary FROM DUAL;

**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.

**Solution:**

CREATE TABLE Accounts (

    AccountID NUMBER PRIMARY KEY,

    CustomerID NUMBER,

    Balance NUMBER

);

CREATE OR REPLACE PACKAGE AccountOperations AS

    PROCEDURE OpenAccount(p\_accid NUMBER, p\_custid NUMBER, p\_balance NUMBER);

    PROCEDURE CloseAccount(p\_accid NUMBER);

    FUNCTION GetTotalBalance(p\_custid NUMBER) RETURN NUMBER;

END AccountOperations;

/

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

    PROCEDURE OpenAccount(p\_accid NUMBER, p\_custid NUMBER, p\_balance NUMBER) IS

    BEGIN

        INSERT INTO Accounts VALUES (p\_accid, p\_custid, p\_balance);

        DBMS\_OUTPUT.PUT\_LINE('Account opened: ' || p\_accid);

    END;

    PROCEDURE CloseAccount(p\_accid NUMBER) IS

    BEGIN

        DELETE FROM Accounts WHERE AccountID = p\_accid;

        DBMS\_OUTPUT.PUT\_LINE('Account closed: ' || p\_accid);

    END;

    FUNCTION GetTotalBalance(p\_custid NUMBER) RETURN NUMBER IS

        v\_total NUMBER;

    BEGIN

        SELECT NVL(SUM(Balance), 0) INTO v\_total FROM Accounts WHERE CustomerID = p\_custid;

        RETURN v\_total;

    END;

END AccountOperations;

/

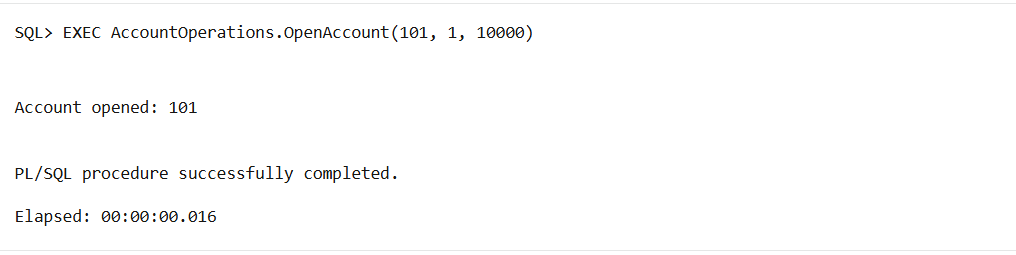
EXEC AccountOperations.OpenAccount(101, 1, 10000);

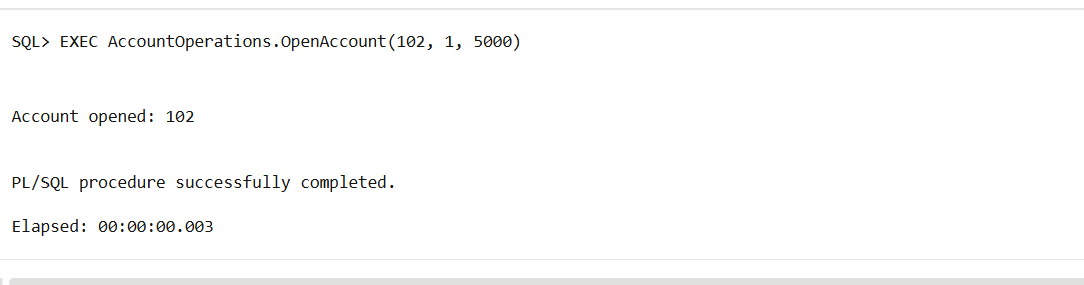
EXEC AccountOperations.OpenAccount(102, 1, 5000);

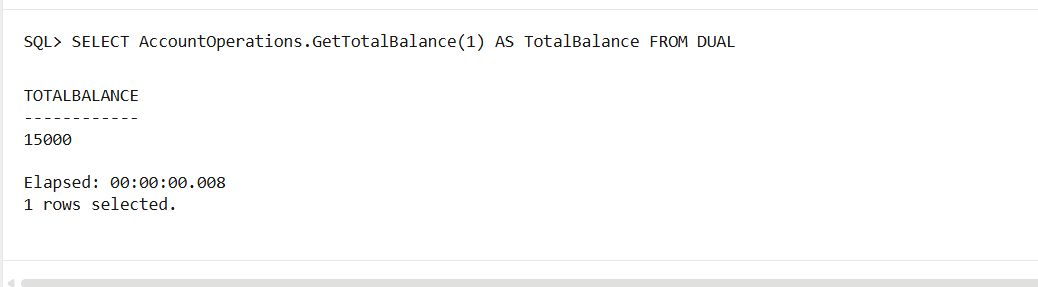
SELECT AccountOperations.GetTotalBalance(1) AS TotalBalance FROM DUAL;

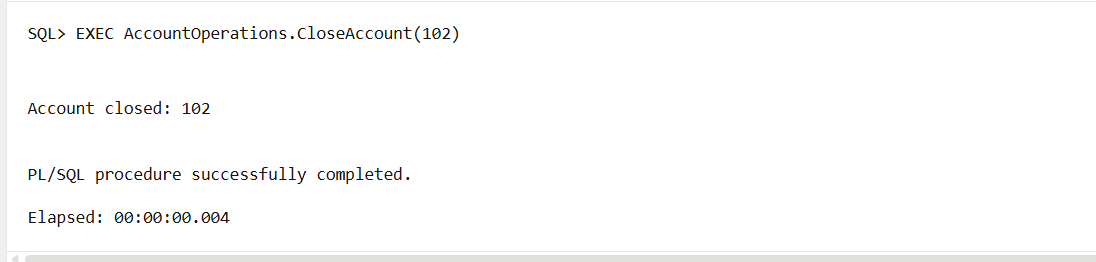
EXEC AccountOperations.CloseAccount(102);

**Output:**

****

****

****

****