**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.

**PL/SQL Block to Apply 1% Discount:**

BEGIN

FOR rec IN (

SELECT c.customer\_id, l.loan\_id, l.interest\_rate, c.age

FROM customers c JOIN loans l ON c.customer\_id = l.customer\_id

)

LOOP

IF rec.age > 60 THEN

UPDATE loans

SET interest\_rate = interest\_rate - 1

WHERE loan\_id = rec.loan\_id;

END IF;

END LOOP;

COMMIT;

END;

/

**Display Output:**

SELECT \* FROM loans;

**OUTPUT SCREENSHOTS:**

|  |
| --- |
| **VIEW OF CREATED TABLES:** |
|  |
| **PL/SQL BLOCK :** |
|  |
| **OUTPUT AFTER EXECUTION OF PL/SQL BLOCK:** |
|  |

**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.

**PL/SQL Block to Set VIP Status:**

BEGIN

FOR rec IN (SELECT customer\_id, balance FROM customers)

LOOP

IF rec.balance > 10000 THEN

UPDATE customers

SET IsVIP = 'TRUE'

WHERE customer\_id = rec.customer\_id;

END IF;

END LOOP;

COMMIT;

END;

/

**View Final Table with Updated VIP Flags:**  
SELECT \* FROM customers;

**OUTPUT SCREENSHOTS:**

|  |
| --- |
| **CREATION OF TABLE AND INSERTION OF VALUES:** |
|  |
| **PL/SQL BLOCK:** |
|  |
| **OUTPUT AFTER EXECUTION OF PL/SQL BLOCK:** |
|  |

**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.

**PL/SQL Block to Print Reminders for Loans Due in Next 30 Days:**  
BEGIN

FOR rec IN (

SELECT l.loan\_id, c.name, l.due\_date

FROM loans l

JOIN customers c ON l.customer\_id = c.customer\_id

WHERE l.due\_date BETWEEN SYSDATE AND SYSDATE + 30

)

LOOP

DBMS\_OUTPUT.PUT\_LINE(

'Reminder: Loan ' || rec.loan\_id ||

' for customer ' || rec.name ||

' is due on ' || TO\_CHAR(rec.due\_date, 'DD-Mon-YYYY')

);

END LOOP;

END;

/

**View All Loan Records:**

SELECT \* FROM loans;

**OUTPUT :**

|  |
| --- |
| **CREATION OF TABLE AND INSERTION OF VALUES:** |
|  |
| **PL/SQL BLOCK:** |
|  |
| **OUTPUT AFTER EXECUTION OF PL/SQL BLOCK:** |
|  |

**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 Customers Table

-- Recreate table

CREATE TABLE accounts (

acc\_no NUMBER PRIMARY KEY,

holder\_name VARCHAR2(100),

balance NUMBER

);

-- Insert fresh data

INSERT INTO accounts VALUES (101, 'Koushika', 5000);

INSERT INTO accounts VALUES (102, 'Madhu', 3000);

COMMIT;

PL/SQL code;

CREATE OR REPLACE PROCEDURE AddNewCustomer(

id IN NUMBER,

cname IN VARCHAR2,

cage IN NUMBER,

cbalance IN NUMBER

) IS

BEGIN

INSERT INTO customers(customer\_id, name, age, balance, IsVIP)

VALUES (id, cname, cage, cbalance, 'FALSE');

COMMIT;

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

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

WHEN OTHERS THEN

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

END;

/

**TABLE CREATION**

|  |
| --- |
|  |
|  |
| **PLSQL** |
|  |
|  |
| **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 TABLE employees (

emp\_id NUMBER PRIMARY KEY,

emp\_name VARCHAR2(100),

salary NUMBER

);  
INSERT INTO employees VALUES (201, 'Siva', 40000);

COMMIT;  
  
CREATE OR REPLACE PROCEDURE UpdateSalary(

emp\_id\_in IN NUMBER,

percent IN NUMBER

) IS

BEGIN

UPDATE employees

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

WHERE emp\_id = emp\_id\_in;

IF SQL%ROWCOUNT = 0 THEN

RAISE\_APPLICATION\_ERROR(-20003, 'Employee ID not found.');

END IF;

COMMIT;

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

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END;

/  
  
EXEC UpdateSalary(201, 10);

|  |
| --- |
| **TABLE CREATION** |
|  |
| **PLSQL** |
|  |
| **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 TABLE customers (

customer\_id NUMBER PRIMARY KEY,

name VARCHAR2(100),

age NUMBER,

balance NUMBER,

IsVIP VARCHAR2(5)

);  
  
CREATE OR REPLACE PROCEDURE AddNewCustomer(

id IN NUMBER,

cname IN VARCHAR2,

cage IN NUMBER,

cbalance IN NUMBER

) IS

BEGIN

INSERT INTO customers(customer\_id, name, age, balance, IsVIP)

VALUES (id, cname, cage, cbalance, 'FALSE');

COMMIT;

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

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

WHEN OTHERS THEN

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

END;

/  
  
EXEC AddNewCustomer(1, 'Ajay', 35, 7000);  
  
**TABLE CREATION**

|  |
| --- |
|  |
|  |
| **PLSQL** |
|  |
| **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 TABLE savings\_accounts (

acc\_no NUMBER PRIMARY KEY,

holder\_name VARCHAR2(100),

balance NUMBER

);

INSERT INTO savings\_accounts VALUES (201, 'Koushika', 10000);

INSERT INTO savings\_accounts VALUES (202, 'Madhu', 15000);

COMMIT;

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

BEGIN

UPDATE savings\_accounts

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

COMMIT;

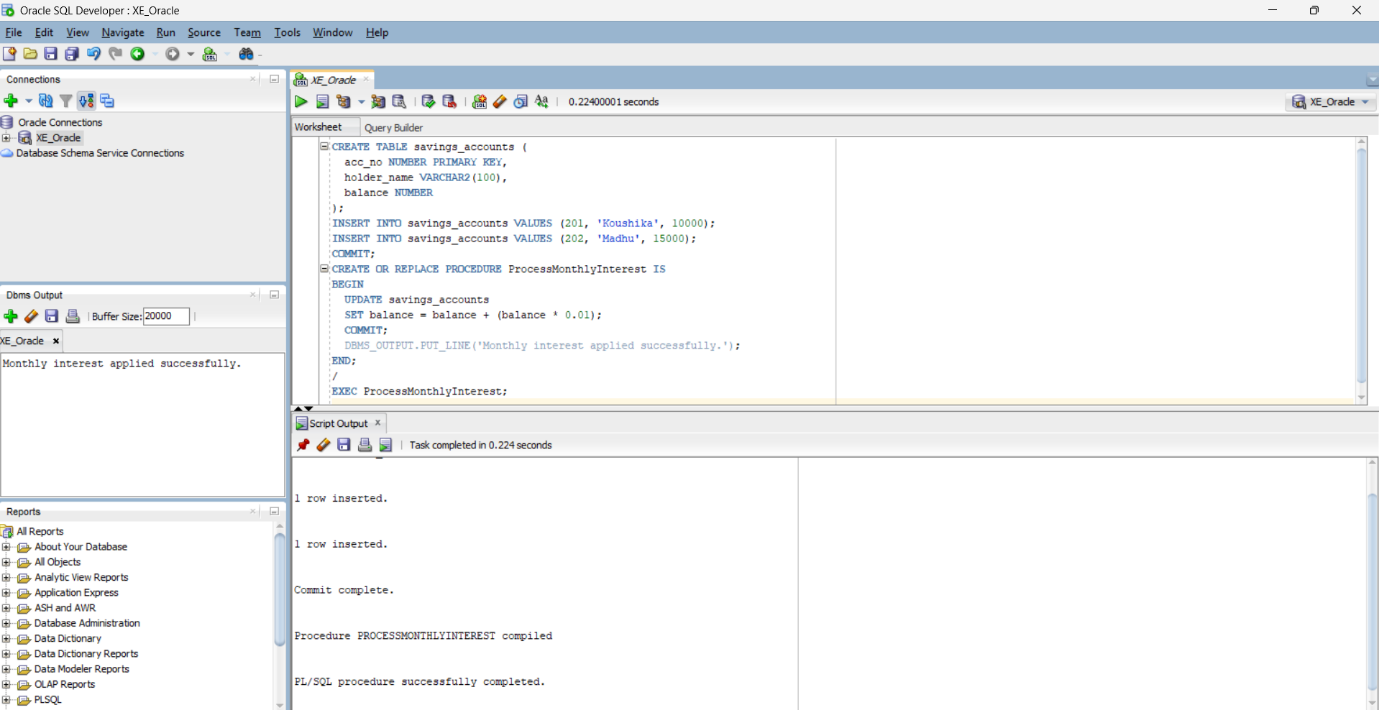
DBMS\_OUTPUT.PUT\_LINE('Monthly interest applied successfully.');

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 TABLE employees (

emp\_id NUMBER PRIMARY KEY,

emp\_name VARCHAR2(100),

department VARCHAR2(50),

salary NUMBER

);

INSERT INTO employees VALUES (301, 'Siva', 'IT', 50000);

INSERT INTO employees VALUES (302, 'Arun', 'HR', 40000);

INSERT INTO employees VALUES (303, 'Meena', 'IT', 55000);

COMMIT;

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(

dept\_name IN VARCHAR2,

bonus\_percent IN NUMBER

) IS

BEGIN

UPDATE employees

SET salary = salary + (salary \* bonus\_percent / 100)

WHERE department = dept\_name;

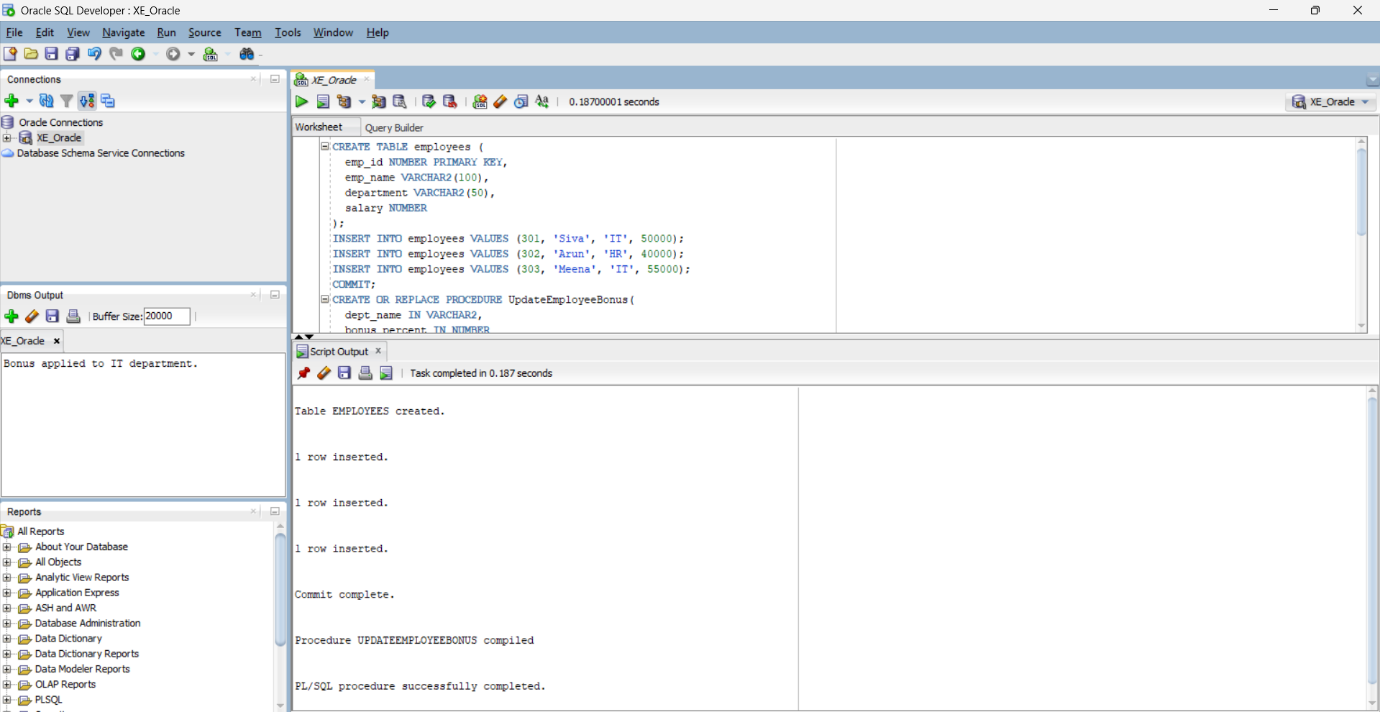
COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Bonus applied to ' || dept\_name || ' department.');

END;

/

EXEC UpdateEmployeeBonus('IT', 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 TABLE accounts (

acc\_no NUMBER PRIMARY KEY,

holder\_name VARCHAR2(100),

balance NUMBER

);

INSERT INTO accounts VALUES (401, 'Koushika', 8000);

INSERT INTO accounts VALUES (402, 'Madhu', 3000);

COMMIT;

CREATE OR REPLACE PROCEDURE TransferFunds(

from\_acc IN NUMBER,

to\_acc IN NUMBER,

amount IN NUMBER

) IS

BEGIN

UPDATE accounts

SET balance = balance - amount

WHERE acc\_no = from\_acc AND balance >= amount;

IF SQL%ROWCOUNT = 0 THEN

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

END IF;

UPDATE accounts

SET balance = balance + amount

WHERE acc\_no = to\_acc;

IF SQL%ROWCOUNT = 0 THEN

RAISE\_APPLICATION\_ERROR(-20002, 'Invalid destination account.');

END IF;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Transfer completed successfully.');

EXCEPTION

WHEN OTHERS THEN

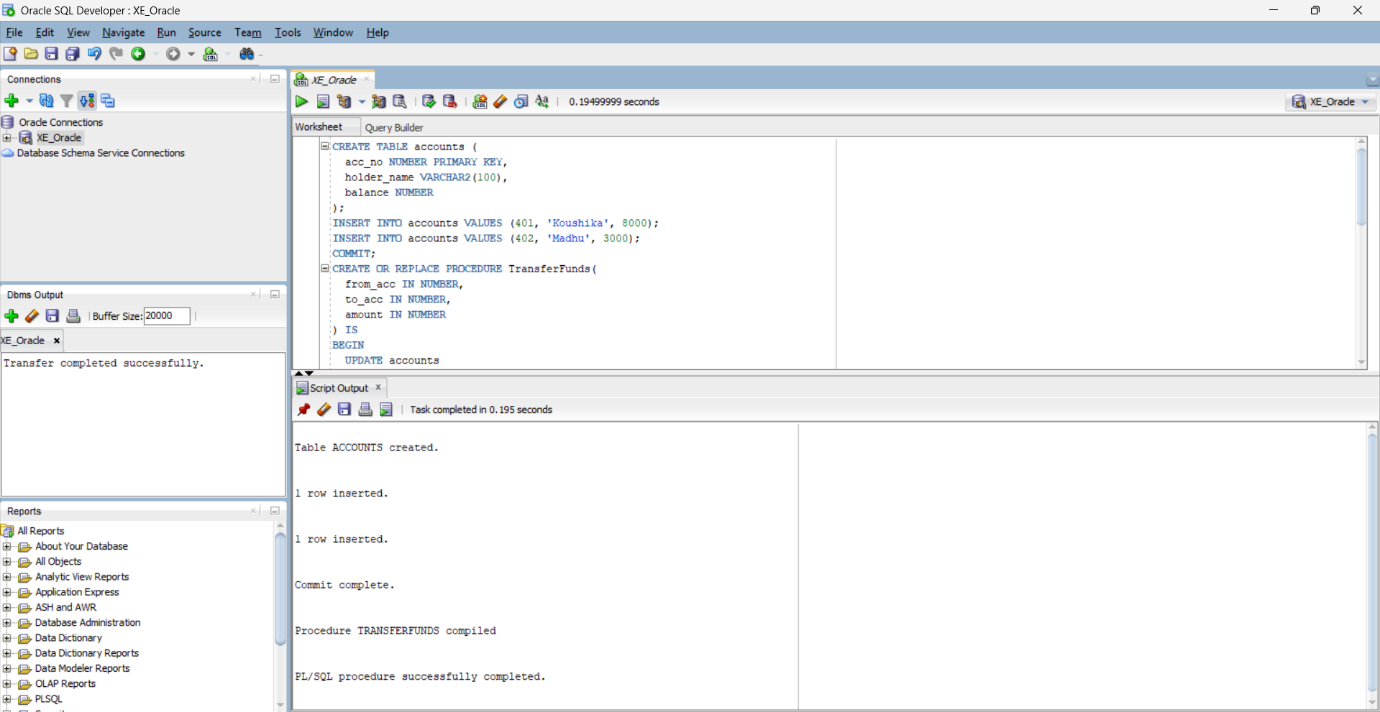
ROLLBACK;

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

END;

/

EXEC TransferFunds(401, 402, 1000);

**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(dob DATE)

RETURN NUMBER IS

age NUMBER;

BEGIN

age := FLOOR(MONTHS\_BETWEEN(SYSDATE, dob) / 12);

RETURN age;

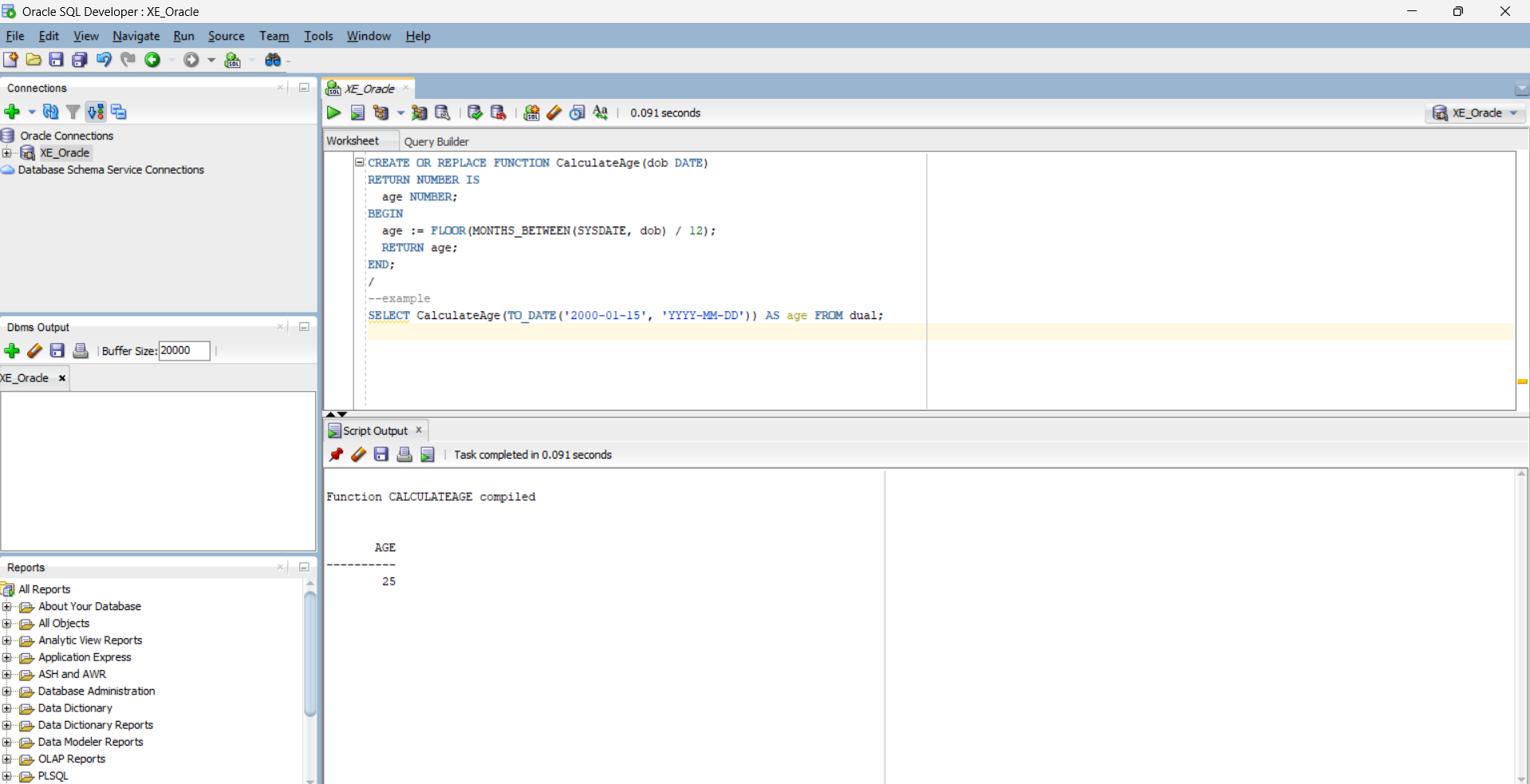
END;

/

--example

SELECT CalculateAge(TO\_DATE('2000-01-15', 'YYYY-MM-DD')) AS age FROM dual;

**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(

amount NUMBER,

annual\_rate NUMBER,

years NUMBER

)

RETURN NUMBER IS

monthly\_rate NUMBER;

n NUMBER;

emi NUMBER;

BEGIN

monthly\_rate := annual\_rate / 12 / 100;

n := years \* 12;

emi := (amount \* monthly\_rate \* POWER(1 + monthly\_rate, n)) /

(POWER(1 + monthly\_rate, n) - 1);

RETURN ROUND(emi, 2);

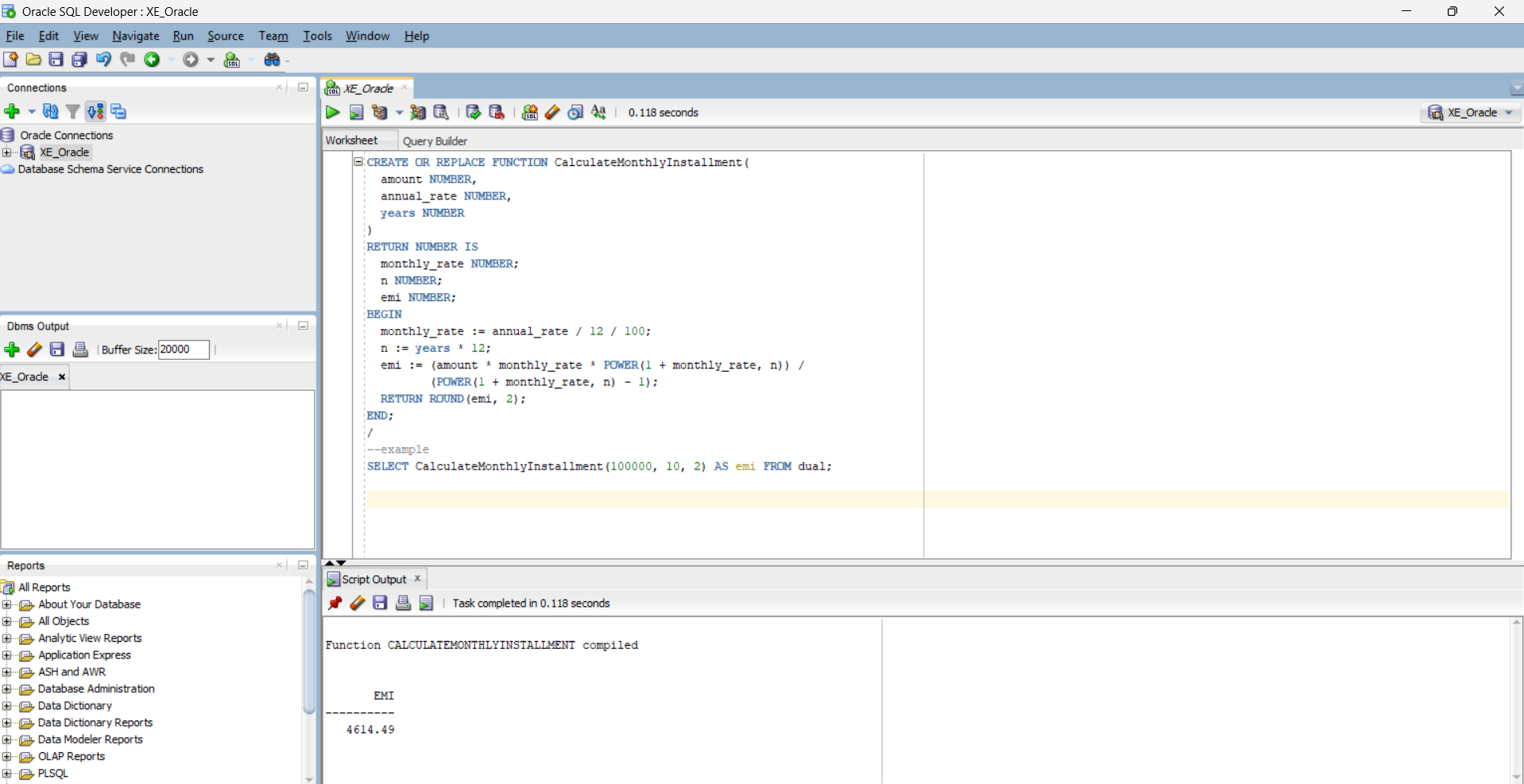
END;

/

--example

SELECT CalculateMonthlyInstallment(100000, 10, 2) 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.

CREATE TABLE accounts (

acc\_no NUMBER PRIMARY KEY,

holder\_name VARCHAR2(100),

balance NUMBER

);

CREATE OR REPLACE FUNCTION HasSufficientBalance(

acc\_id IN NUMBER,

amount IN NUMBER

)

RETURN BOOLEAN IS

acc\_balance NUMBER;

BEGIN

SELECT balance INTO acc\_balance FROM accounts WHERE acc\_no = acc\_id;

RETURN acc\_balance >= amount;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN FALSE;

END;

/

--example usage

DECLARE

result BOOLEAN;

BEGIN

result := HasSufficientBalance(401, 2000);

IF result THEN

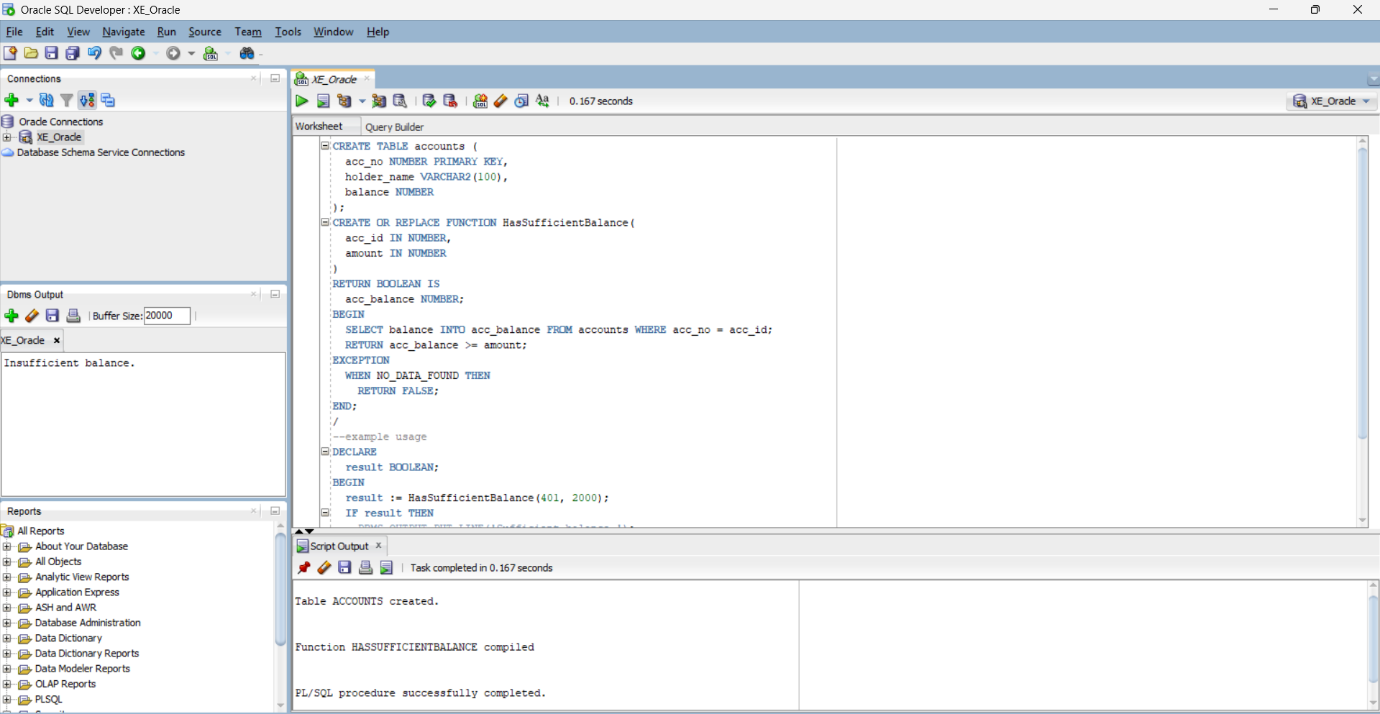
DBMS\_OUTPUT.PUT\_LINE('Sufficient balance.');

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.  
  
CREATE TABLE customers (

customer\_id NUMBER PRIMARY KEY,

name VARCHAR2(100),

age NUMBER,

balance NUMBER,

IsVIP VARCHAR2(5),

LastModified DATE

);

INSERT INTO customers VALUES (1, 'Koushika', 25, 12000, 'FALSE', SYSDATE);

COMMIT;

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON customers

FOR EACH ROW

BEGIN

:NEW.LastModified := SYSDATE;

END;

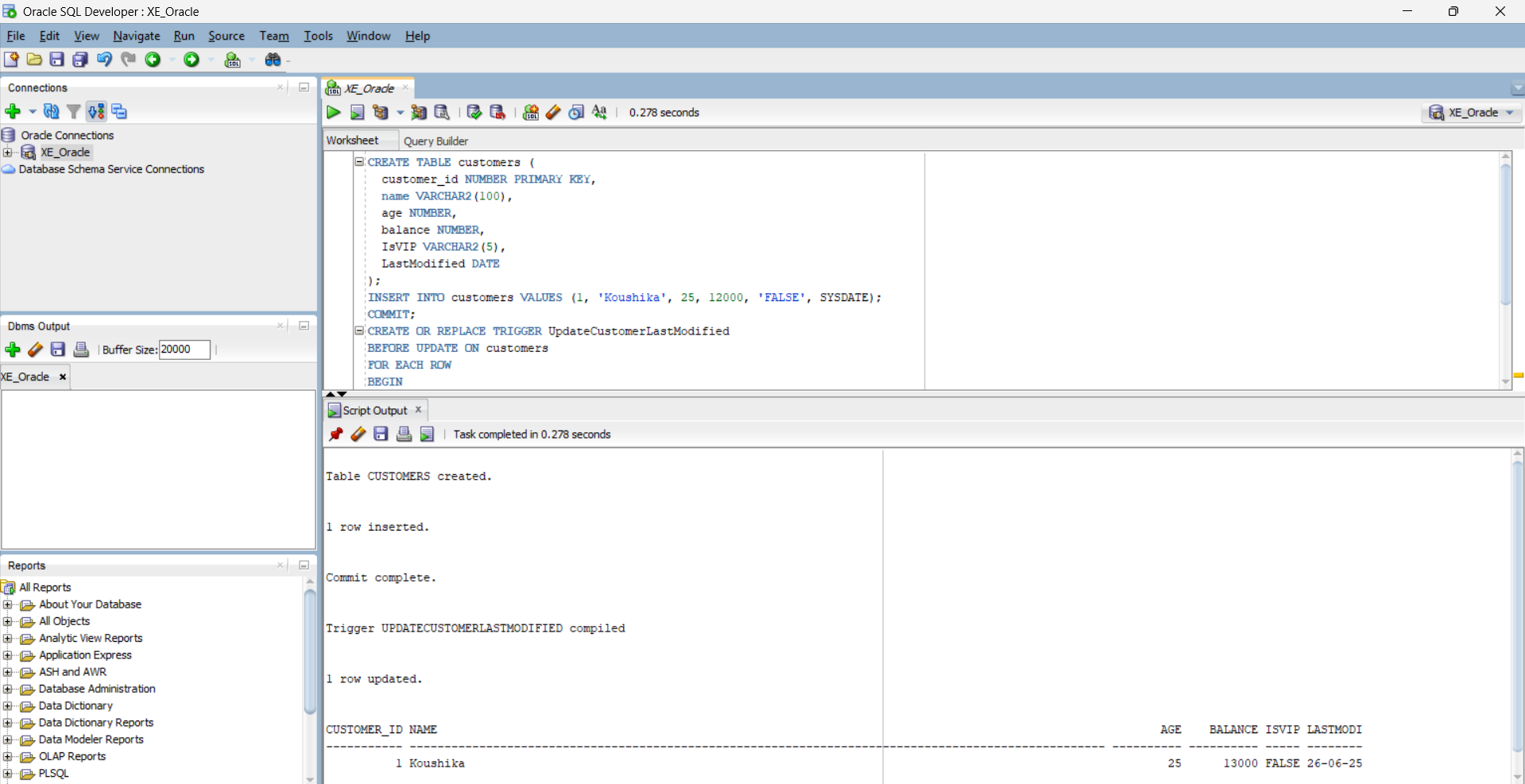
/

--example usage

UPDATE customers SET balance = balance + 1000 WHERE customer\_id = 1;

SELECT \* FROM customers WHERE customer\_id = 1;

**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 transactions (

txn\_id NUMBER PRIMARY KEY,

acc\_no NUMBER,

txn\_type VARCHAR2(20),

amount NUMBER,

txn\_date DATE

);

CREATE TABLE audit\_log (

log\_id NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,

txn\_id NUMBER,

log\_time DATE,

action VARCHAR2(50)

);

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON transactions

FOR EACH ROW

BEGIN

INSERT INTO audit\_log (txn\_id, log\_time, action)

VALUES (:NEW.txn\_id, SYSDATE, 'Transaction Inserted');

END;

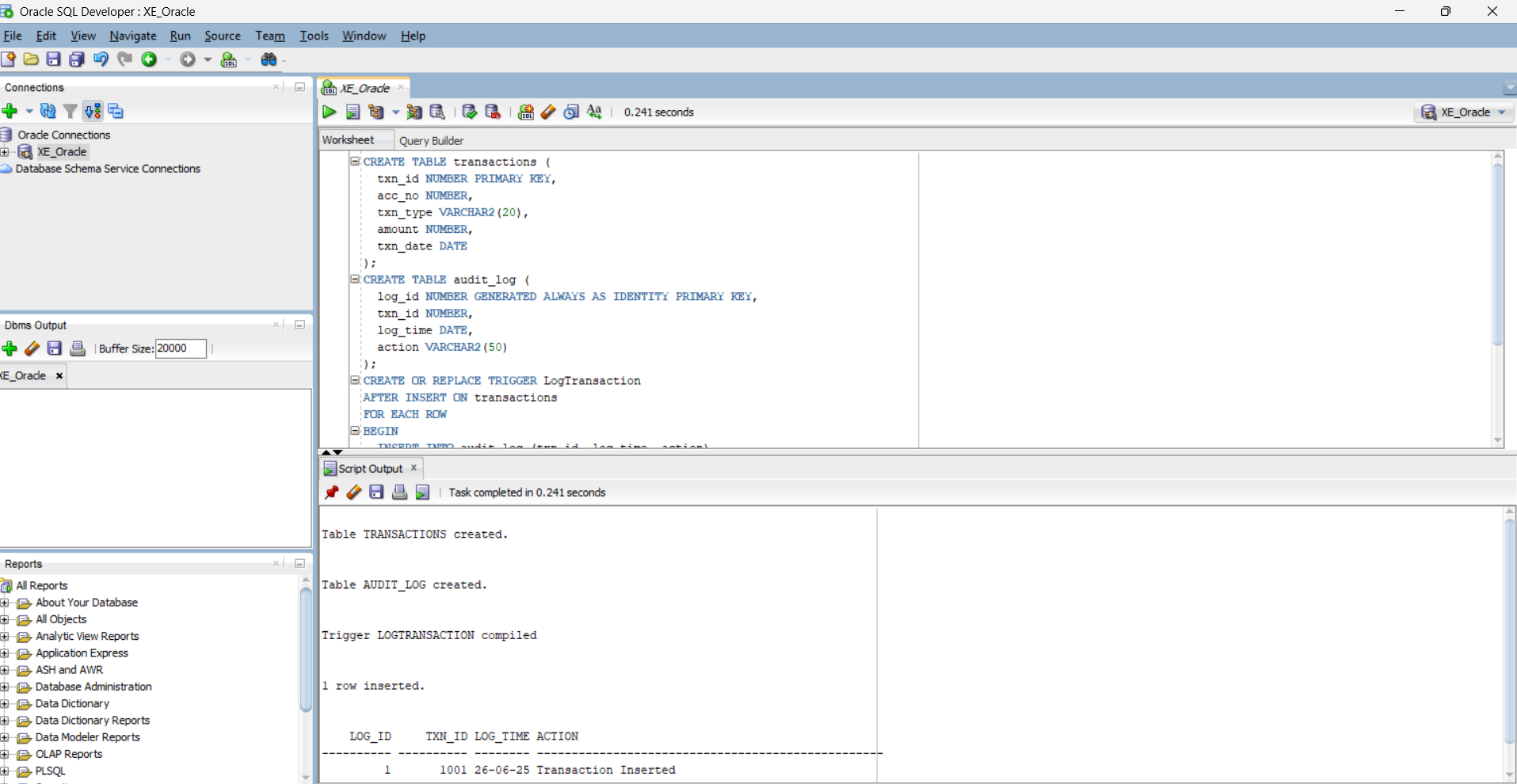
/

--example usage

INSERT INTO transactions VALUES (1001, 401, 'deposit', 2000, SYSDATE);

SELECT \* FROM audit\_log;

**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 TABLE accounts (

acc\_no NUMBER PRIMARY KEY,

holder\_name VARCHAR2(100),

balance NUMBER

);

CREATE TABLE transactions (

txn\_id NUMBER PRIMARY KEY,

acc\_no NUMBER,

txn\_type VARCHAR2(20),

amount NUMBER,

txn\_date DATE

);

INSERT INTO accounts VALUES (401, 'Koushika', 8000);

COMMIT;

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON transactions

FOR EACH ROW

DECLARE

acc\_balance NUMBER;

BEGIN

SELECT balance INTO acc\_balance

FROM accounts

WHERE acc\_no = :NEW.acc\_no;

IF :NEW.txn\_type = 'withdraw' AND :NEW.amount > acc\_balance THEN

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

END IF;

IF :NEW.txn\_type = 'deposit' AND :NEW.amount <= 0 THEN

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

END IF;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RAISE\_APPLICATION\_ERROR(-20003, 'Account not found.');

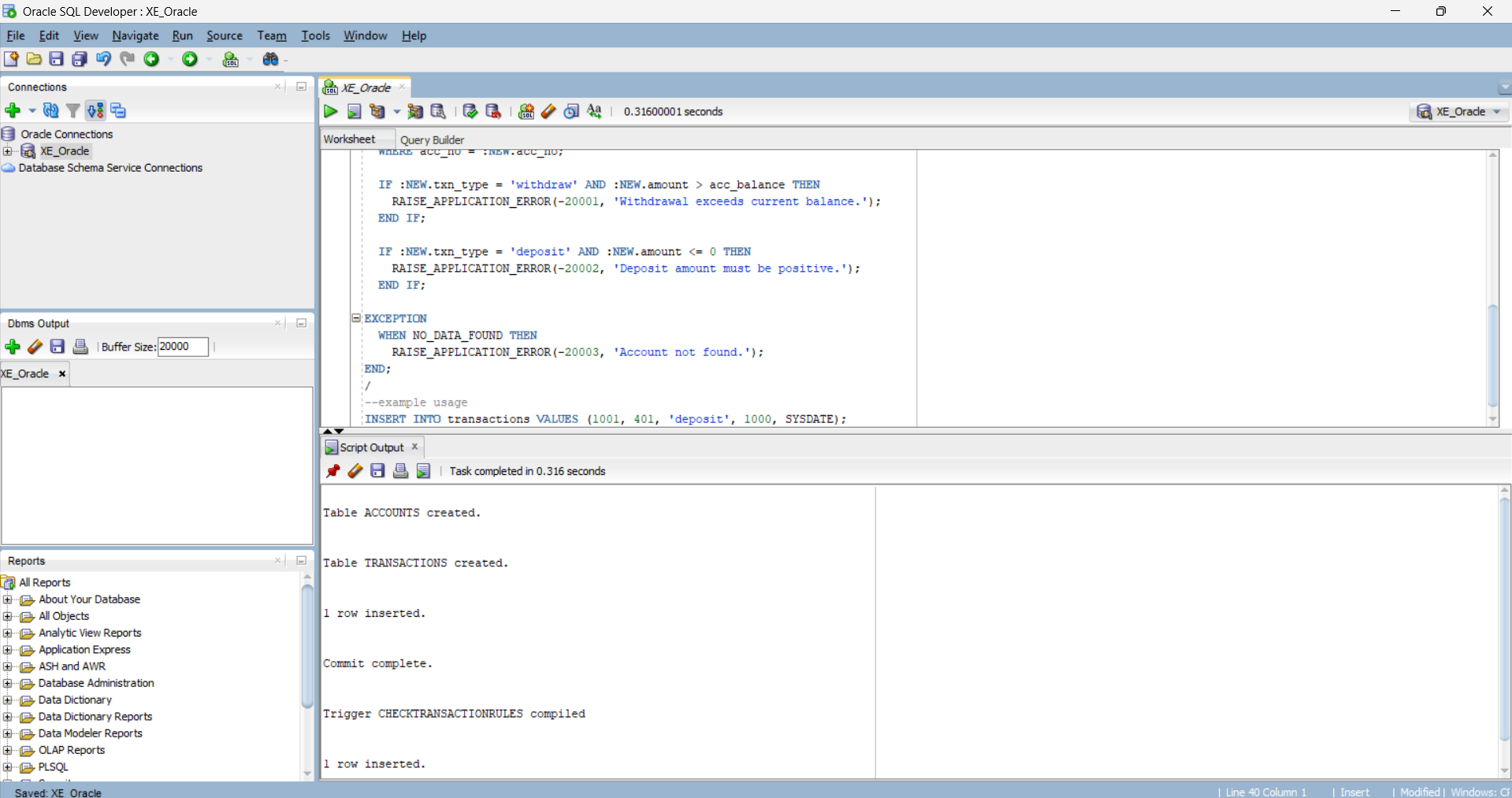
END;

/

--example usage

INSERT INTO transactions VALUES (1001, 401, 'deposit', 1000, SYSDATE);

**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.

CREATE TABLE transactions (

txn\_id NUMBER PRIMARY KEY,

acc\_no NUMBER,

txn\_type VARCHAR2(20),

amount NUMBER,

txn\_date DATE

);

INSERT INTO transactions VALUES (2001, 401, 'deposit', 1000, SYSDATE);

INSERT INTO transactions VALUES (2002, 401, 'withdraw', 500, SYSDATE - 2);

INSERT INTO transactions VALUES (2003, 402, 'deposit', 2000, SYSDATE - 40);

COMMIT;

DECLARE

CURSOR txn\_cursor IS

SELECT acc\_no, txn\_type, amount, txn\_date

FROM transactions

WHERE TO\_CHAR(txn\_date, 'MMYYYY') = TO\_CHAR(SYSDATE, 'MMYYYY');

v\_acc\_no transactions.acc\_no%TYPE;

v\_type transactions.txn\_type%TYPE;

v\_amt transactions.amount%TYPE;

v\_date transactions.txn\_date%TYPE;

BEGIN

OPEN txn\_cursor;

LOOP

FETCH txn\_cursor INTO v\_acc\_no, v\_type, v\_amt, v\_date;

EXIT WHEN txn\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Account: ' || v\_acc\_no || ' | Type: ' || v\_type || ' | Amount: ' || v\_amt || ' | Date: ' || TO\_CHAR(v\_date, 'DD-Mon-YYYY'));

END LOOP;

CLOSE txn\_cursor;

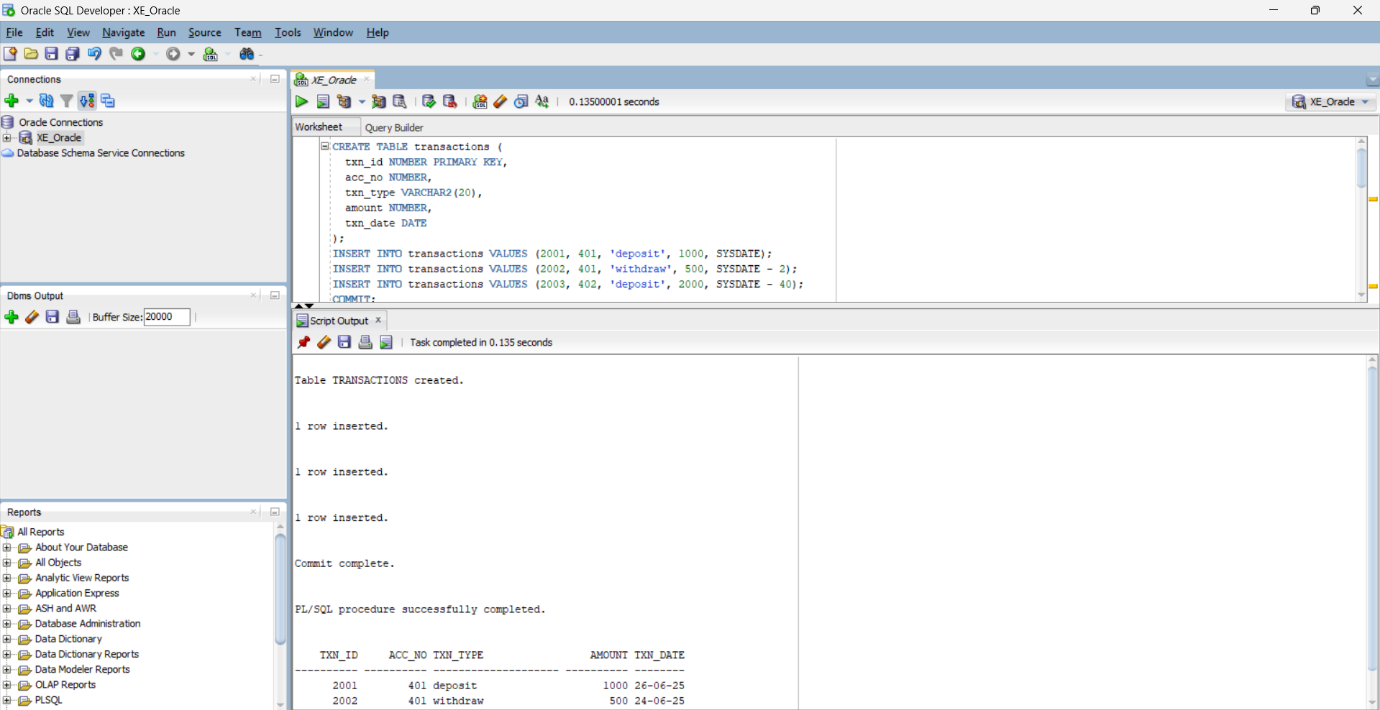
END;

/

--example usage

SELECT \* FROM transactions

WHERE TO\_CHAR(txn\_date, 'MMYYYY') = TO\_CHAR(SYSDATE, 'MMYYYY');

 **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 acc\_no, balance FROM accounts;

v\_acc\_no accounts.acc\_no%TYPE;

v\_balance accounts.balance%TYPE;

annual\_fee CONSTANT NUMBER := 500;

BEGIN

OPEN acc\_cursor;

LOOP

FETCH acc\_cursor INTO v\_acc\_no, v\_balance;

EXIT WHEN acc\_cursor%NOTFOUND;

UPDATE accounts

SET balance = balance - annual\_fee

WHERE acc\_no = v\_acc\_no;

DBMS\_OUTPUT.PUT\_LINE('Annual fee applied to Account: ' || v\_acc\_no);

END LOOP;

CLOSE acc\_cursor;

COMMIT;

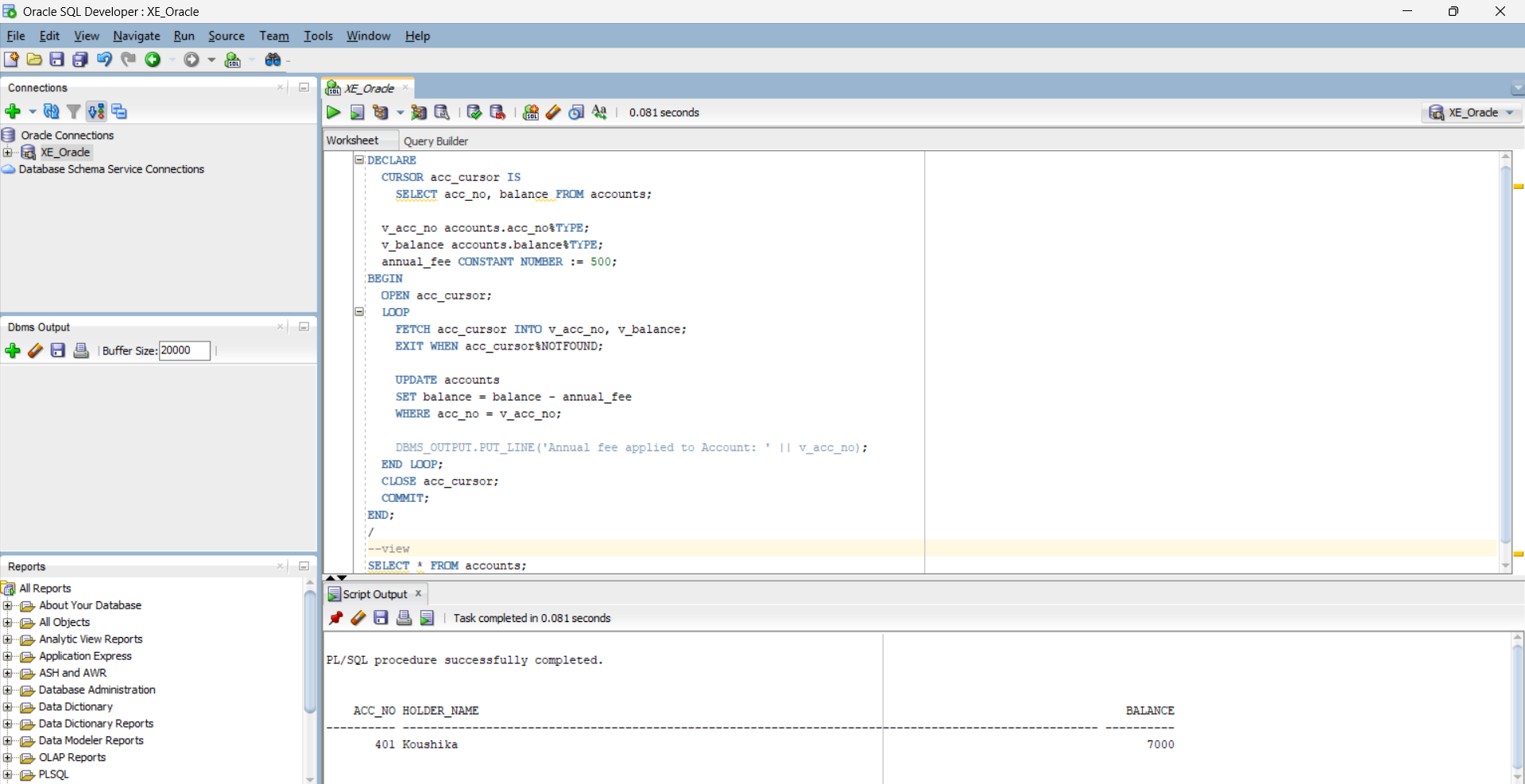
END;

/

--view

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.

CREATE TABLE loans (

loan\_id NUMBER PRIMARY KEY,

customer\_id NUMBER,

interest\_rate NUMBER,

due\_date DATE

);

INSERT INTO loans VALUES (301, 1, 9.5, SYSDATE + 180);

INSERT INTO loans VALUES (302, 2, 8.0, SYSDATE + 90);

COMMIT;

DECLARE

CURSOR loan\_cursor IS

SELECT loan\_id, interest\_rate FROM loans;

v\_loan\_id loans.loan\_id%TYPE;

v\_rate loans.interest\_rate%TYPE;

BEGIN

OPEN loan\_cursor;

LOOP

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

EXIT WHEN loan\_cursor%NOTFOUND;

UPDATE loans

SET interest\_rate = v\_rate + 0.5

WHERE loan\_id = v\_loan\_id;

DBMS\_OUTPUT.PUT\_LINE('Updated interest for Loan ID: ' || v\_loan\_id);

END LOOP;

CLOSE loan\_cursor;

COMMIT;

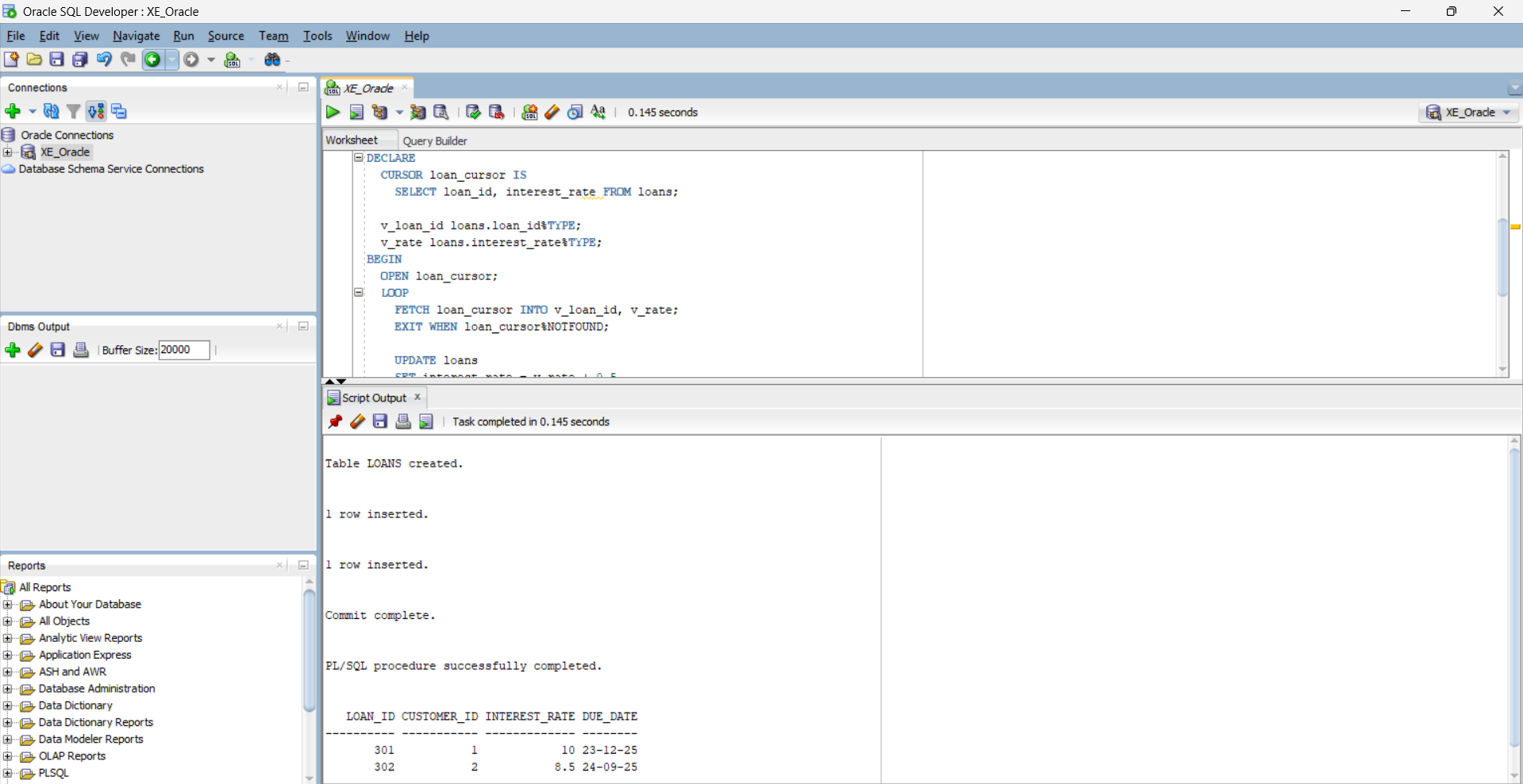
END;

/

--view

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.  
CREATE TABLE customers (

CustomerID NUMBER PRIMARY KEY,

Name VARCHAR2(100),

DOB DATE,

Balance NUMBER,

LastModified DATE

);

CREATE OR REPLACE PACKAGE CustomerManagement AS

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

PROCEDURE UpdateCustomerBalance(p\_id NUMBER, p\_new\_balance NUMBER);

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 UpdateCustomerBalance(p\_id NUMBER, p\_new\_balance NUMBER) IS

BEGIN

UPDATE customers

SET Balance = p\_new\_balance, 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;

/

--test

BEGIN

CustomerManagement.AddCustomer(3, 'Koushika', TO\_DATE('1999-01-01','YYYY-MM-DD'), 2000);

END;

/

BEGIN

CustomerManagement.UpdateCustomerBalance(3, 2500);

END;

/

DECLARE

v\_bal NUMBER;

BEGIN

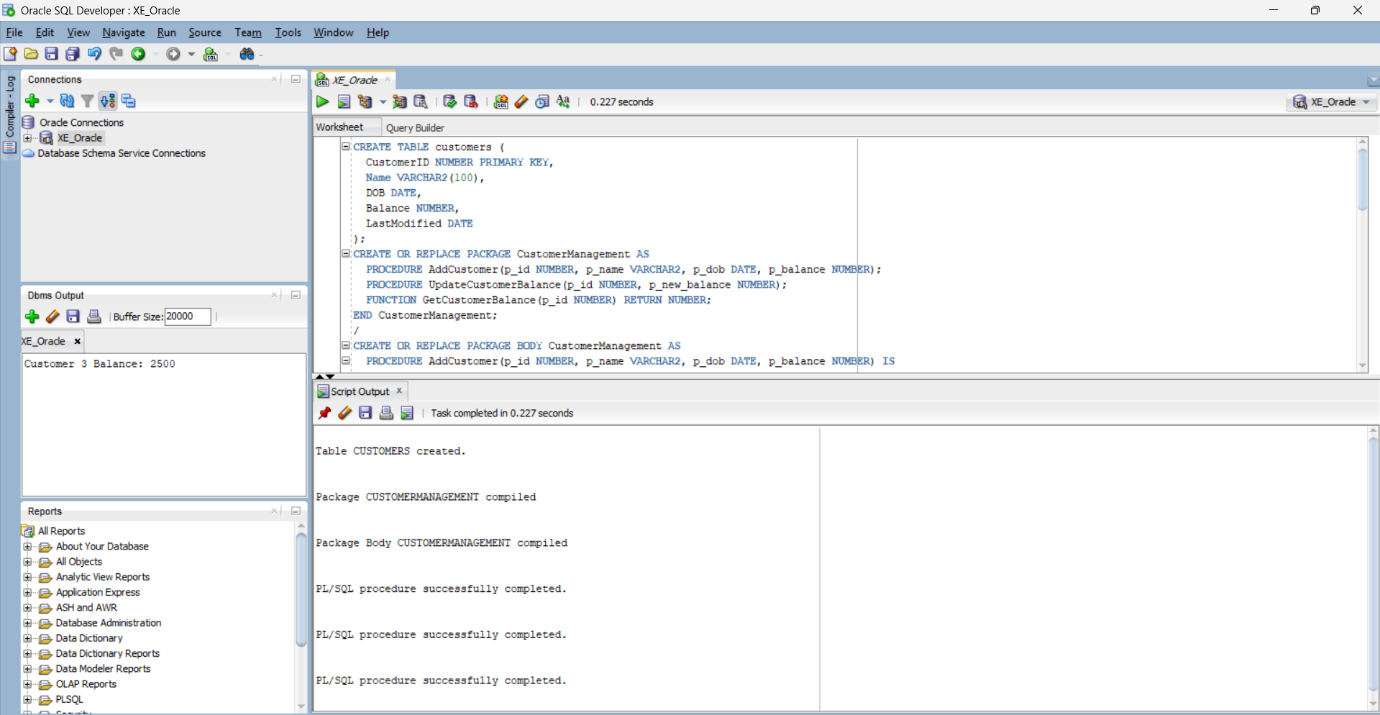
v\_bal := CustomerManagement.GetCustomerBalance(3);

DBMS\_OUTPUT.PUT\_LINE('Customer 3 Balance: ' || v\_bal);

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.

SET SERVEROUTPUT ON;

BEGIN

EXECUTE IMMEDIATE 'DROP TABLE employees CASCADE CONSTRAINTS';

EXCEPTION

WHEN OTHERS THEN

IF SQLCODE != -942 THEN

RAISE;

END IF;

END;

/

CREATE TABLE employees (

EmployeeID NUMBER PRIMARY KEY,

Name VARCHAR2(100),

Position VARCHAR2(50),

Salary NUMBER,

Department VARCHAR2(50),

HireDate DATE

);

/

CREATE OR REPLACE PACKAGE EmployeeManagement AS

PROCEDURE HireEmployee(

p\_id NUMBER, p\_name VARCHAR2, p\_position VARCHAR2,

p\_salary NUMBER, p\_department VARCHAR2, p\_hiredate DATE

);

PROCEDURE UpdateEmployeeSalary(p\_id NUMBER, p\_new\_salary NUMBER);

FUNCTION GetAnnualSalary(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, p\_hiredate DATE

) IS

BEGIN

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

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

END;

PROCEDURE UpdateEmployeeSalary(p\_id NUMBER, p\_new\_salary NUMBER) IS

BEGIN

UPDATE employees

SET Salary = p\_new\_salary

WHERE EmployeeID = p\_id;

END;

FUNCTION GetAnnualSalary(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(

3, 'Siva', 'Analyst', 55000, 'Finance', SYSDATE

);

END;

/

DECLARE

v\_annual NUMBER;

BEGIN

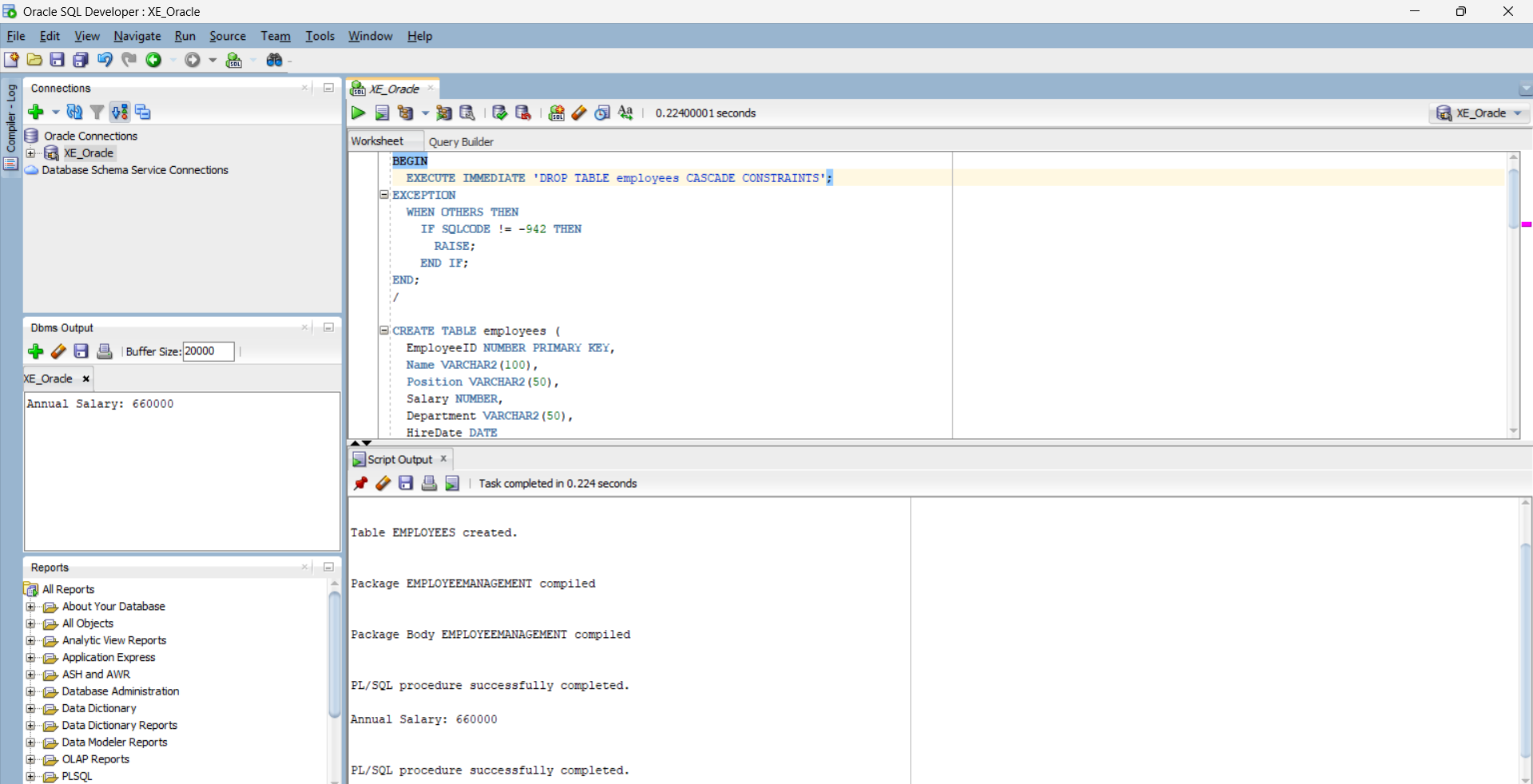
v\_annual := EmployeeManagement.GetAnnualSalary(3);

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.  
  
SET SERVEROUTPUT ON;

BEGIN

EXECUTE IMMEDIATE 'DROP TABLE accounts CASCADE CONSTRAINTS';

EXCEPTION

WHEN OTHERS THEN

IF SQLCODE != -942 THEN

RAISE;

END IF;

END;

/

CREATE TABLE accounts (

AccountID NUMBER PRIMARY KEY,

CustomerID NUMBER,

AccountType VARCHAR2(20),

Balance NUMBER,

LastModified DATE

);

/

CREATE OR REPLACE PACKAGE AccountOperations AS

PROCEDURE OpenAccount(p\_accID NUMBER, p\_custID NUMBER, p\_type VARCHAR2, p\_balance NUMBER);

PROCEDURE CloseAccount(p\_accID NUMBER);

FUNCTION GetTotalBalance(p\_custID NUMBER) RETURN NUMBER;

END AccountOperations;

/

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

PROCEDURE OpenAccount(p\_accID NUMBER, p\_custID NUMBER, p\_type VARCHAR2, p\_balance NUMBER) IS

BEGIN

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

VALUES (p\_accID, p\_custID, p\_type, p\_balance, SYSDATE);

END;

PROCEDURE CloseAccount(p\_accID NUMBER) IS

BEGIN

DELETE FROM accounts WHERE AccountID = p\_accID;

END;

FUNCTION GetTotalBalance(p\_custID NUMBER) RETURN NUMBER IS

v\_total NUMBER := 0;

BEGIN

SELECT NVL(SUM(Balance), 0) INTO v\_total

FROM accounts

WHERE CustomerID = p\_custID;

RETURN v\_total;

END;

END AccountOperations;

/

BEGIN

AccountOperations.OpenAccount(101, 1, 'Savings', 2500);

AccountOperations.OpenAccount(102, 1, 'Checking', 1500);

END;

/

DECLARE

v\_balance NUMBER;

BEGIN

v\_balance := AccountOperations.GetTotalBalance(1);

DBMS\_OUTPUT.PUT\_LINE('Total Balance for Customer 1: ' || v\_balance);

END;

/

BEGIN

AccountOperations.CloseAccount(102);

END;

/

DECLARE

v\_balance NUMBER;

BEGIN

v\_balance := AccountOperations.GetTotalBalance(1);

DBMS\_OUTPUT.PUT\_LINE('Balance after closing one account: ' || v\_balance);

END;

/

**OUTPUT:**

