**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:**

**Create the table**

CREATE TABLE Customers (

CustomerID NUMBER PRIMARY KEY,

Name VARCHAR2(50),

Age NUMBER,

LoanInterestRate NUMBER

);

**Insert sample data**

INSERT INTO Customers VALUES (1, 'Ravi', 45, 8.5);

INSERT INTO Customers VALUES (2, 'Meena', 62, 9.0);

INSERT INTO Customers VALUES (3, 'Arun', 70, 7.5);

INSERT INTO Customers VALUES (4, 'Latha', 59, 10.0);

INSERT INTO Customers VALUES (5, 'Kumar', 65, 8.0);

COMMIT;

**PL/SQL Block to apply 1% discount to customers above 60**

BEGIN

FOR rec IN (SELECT CustomerID, Age, LoanInterestRate FROM Customers) LOOP

IF rec.Age > 60 THEN

UPDATE Customers

SET LoanInterestRate = LoanInterestRate - (LoanInterestRate \* 0.01)

WHERE CustomerID = rec.CustomerID;

END IF;

END LOOP;

COMMIT;

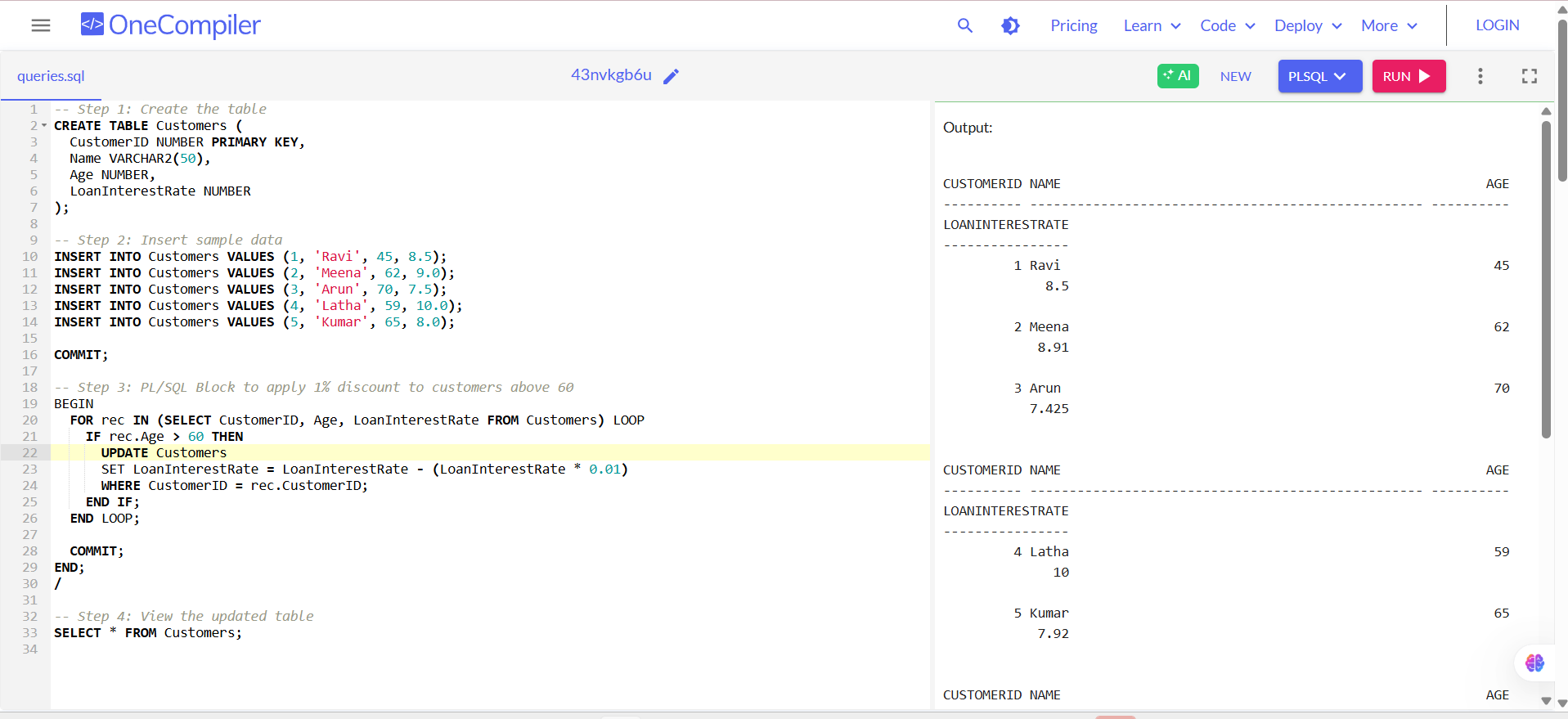
END;

/

**View the updated table**

SELECT \* FROM Customers;

**OUTPUT SCREENSHOT:**

****

**EXPLANATION:**

1. Creates a Customers table and inserts sample data with age and loan interest rate.
2. Loops through each customer and checks if their age is above 60.
3. Applies a 1% discount on their loan interest rate and updates the table.

**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:**

**Create the Customers table with IsVIP column**

CREATE TABLE Customers (

CustomerID NUMBER PRIMARY KEY,

Name VARCHAR2(50),

Balance NUMBER,

IsVIP VARCHAR2(5) -- This will store 'TRUE' or 'FALSE'

);

**Insert sample data**

INSERT INTO Customers VALUES (1, 'Ravi', 9500, 'FALSE');

INSERT INTO Customers VALUES (2, 'Meena', 15000, 'FALSE');

INSERT INTO Customers VALUES (3, 'Arun', 10500, 'FALSE');

INSERT INTO Customers VALUES (4, 'Latha', 8200, 'FALSE');

INSERT INTO Customers VALUES (5, 'Kumar', 25000, 'FALSE');

COMMIT;

**PL/SQL block to set IsVIP = 'TRUE' for Balance > 10000**

BEGIN

FOR rec IN (SELECT CustomerID, Balance FROM Customers) LOOP

IF rec.Balance > 10000 THEN

UPDATE Customers

SET IsVIP = 'TRUE'

WHERE CustomerID = rec.CustomerID;

END IF;

END LOOP;

COMMIT;

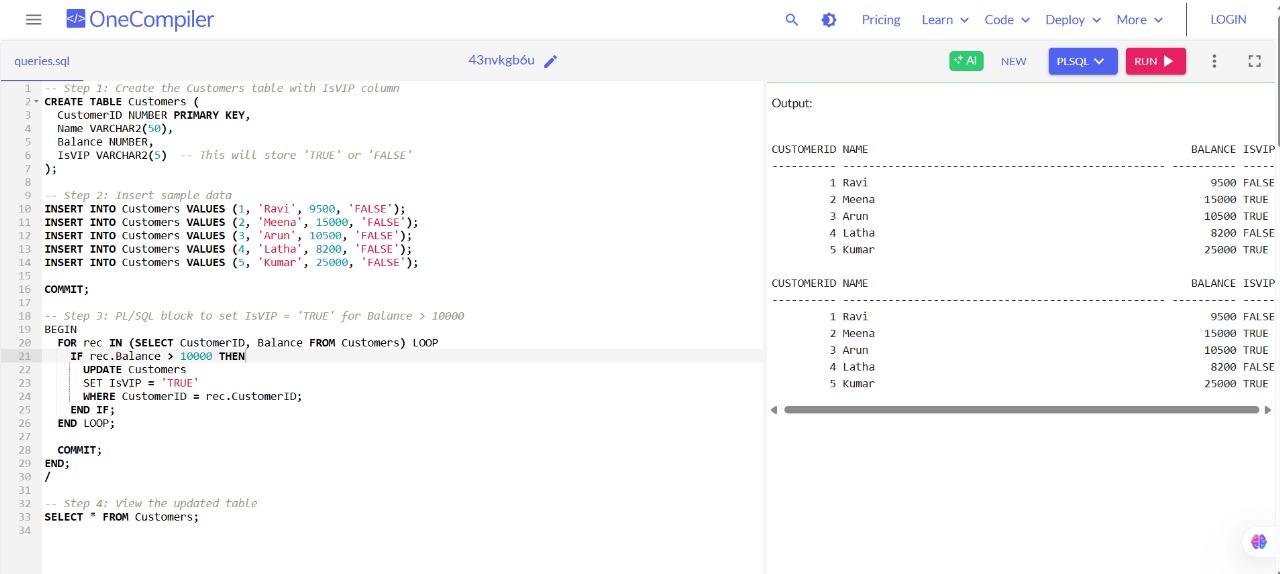
END;

/

**View the updated table**

SELECT \* FROM Customers;

**OUTPUT SCREENSHOT:**



EXPLANATION:

1. Creates a Customers table and adds a flag column IsVIP to store VIP status.
2. Loops through all customers and checks if their balance is over $10,000.
3. Sets IsVIP to 'TRUE' for those customers and updates the table.

**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:**

CREATE TABLE Customers (

CustomerID NUMBER PRIMARY KEY,

Name VARCHAR2(100),

DOB DATE,

Balance NUMBER,

LastModified DATE

);

CREATE TABLE Loans (

LoanID NUMBER PRIMARY KEY,

CustomerID NUMBER,

LoanAmount NUMBER,

InterestRate NUMBER,

StartDate DATE,

EndDate DATE,

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

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

VALUES (1, 'John Doe', TO\_DATE('1950-05-15', 'YYYY-MM-DD'), 12000, SYSDATE);

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

VALUES (2, 'Jane Smith', TO\_DATE('1990-07-20', 'YYYY-MM-DD'), 8000, SYSDATE);

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

VALUES (3, 'Mike Lee', TO\_DATE('1982-10-10', 'YYYY-MM-DD'), 15000, SYSDATE);

INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate)

VALUES (1, 1, 5000, 5.5, SYSDATE, SYSDATE + 20);

INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate)

VALUES (2, 2, 7000, 6.0, SYSDATE, SYSDATE + 45);

INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate) VALUES (3, 3, 3000, 4.5, SYSDATE, SYSDATE + 10);

SET SERVEROUTPUT ON;

BEGIN

FOR loan\_exm IN (

SELECT l.LoanID, c.Name, l.EndDate

FROM Loans l

JOIN Customers c ON c.CustomerID = l.CustomerID

WHERE l.EndDate BETWEEN SYSDATE AND SYSDATE + 30

)

LOOP

DBMS\_OUTPUT.PUT\_LINE(

'Reminder: Loan ID ' || loan\_exm.LoanID ||

' for customer ' || loan\_exm.Name ||

' is due on ' || TO\_CHAR(loan\_exm.EndDate, 'YYYY-MM-DD')

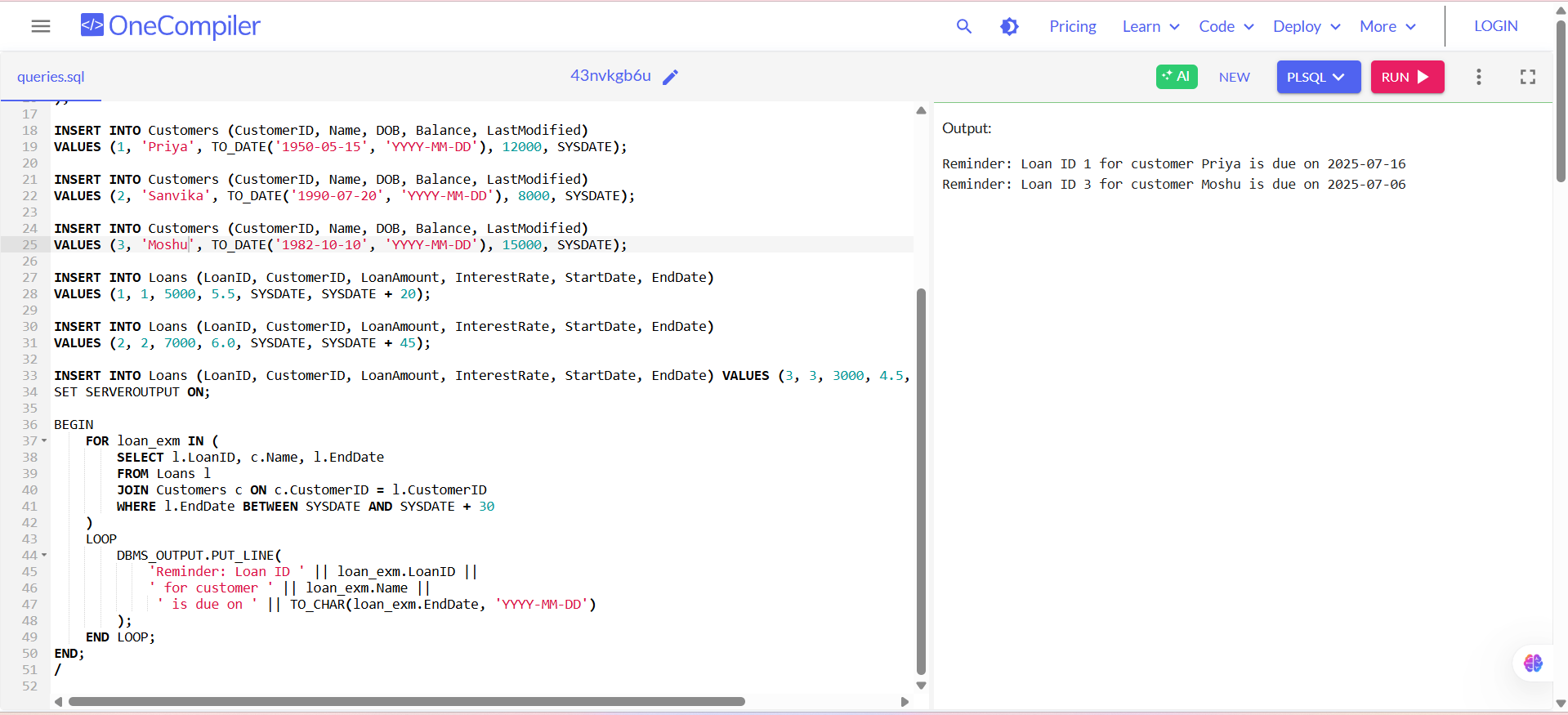
);

END LOOP;

END;

/

**OUTPUT SCREENSHOT:**



**EXPLANATION:**

1. **Loan Due Date Check:**The PL/SQL block filters loans with EndDate within the next 30 days using BETWEEN SYSDATE AND SYSDATE + 30.
2. **Customer Info Join:**  
   It joins the Customers table to show each customer’s name along with the loan info.
3. **Message Output:**  
   For each matching record, it prints a formatted reminder using DBMS\_OUTPUT.PUT\_LINE.

**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 Accounts table**

CREATE TABLE Accounts (

AccountID NUMBER PRIMARY KEY,

AccountHolder VARCHAR2(50),

Balance NUMBER

);

**-- Insert sample data**

INSERT INTO Accounts VALUES (1, 'Ravi', 5000);

INSERT INTO Accounts VALUES (2, 'Meena', 3000);

COMMIT;

**-- Create procedure SafeTransferFunds**

CREATE OR REPLACE PROCEDURE SafeTransferFunds (

p\_from\_account IN NUMBER,

p\_to\_account IN NUMBER,

p\_amount IN NUMBER

) AS

insufficient\_funds EXCEPTION;

pragma EXCEPTION\_INIT(insufficient\_funds, -20001);

v\_balance NUMBER;

BEGIN

-- Check balance of source account

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

IF v\_balance < p\_amount THEN

RAISE insufficient\_funds;

END IF;

-- Deduct from source

UPDATE Accounts

SET Balance = Balance - p\_amount

WHERE AccountID = p\_from\_account;

-- Add to destination

UPDATE Accounts

SET Balance = Balance + p\_amount

WHERE AccountID = p\_to\_account;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Transfer of ' || p\_amount || ' from Account ' || p\_from\_account ||

' to Account ' || p\_to\_account || ' completed successfully.');

EXCEPTION

WHEN insufficient\_funds THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: Insufficient funds in Account ' || p\_from\_account || '. Transfer aborted.');

WHEN NO\_DATA\_FOUND THEN

ROLLBACK;

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

WHEN OTHERS THEN

ROLLBACK;

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

END;

/

-- Enable DBMS output

BEGIN

DBMS\_OUTPUT.ENABLE;

END;

/

**-- Test the procedure: one success, one failure**

BEGIN

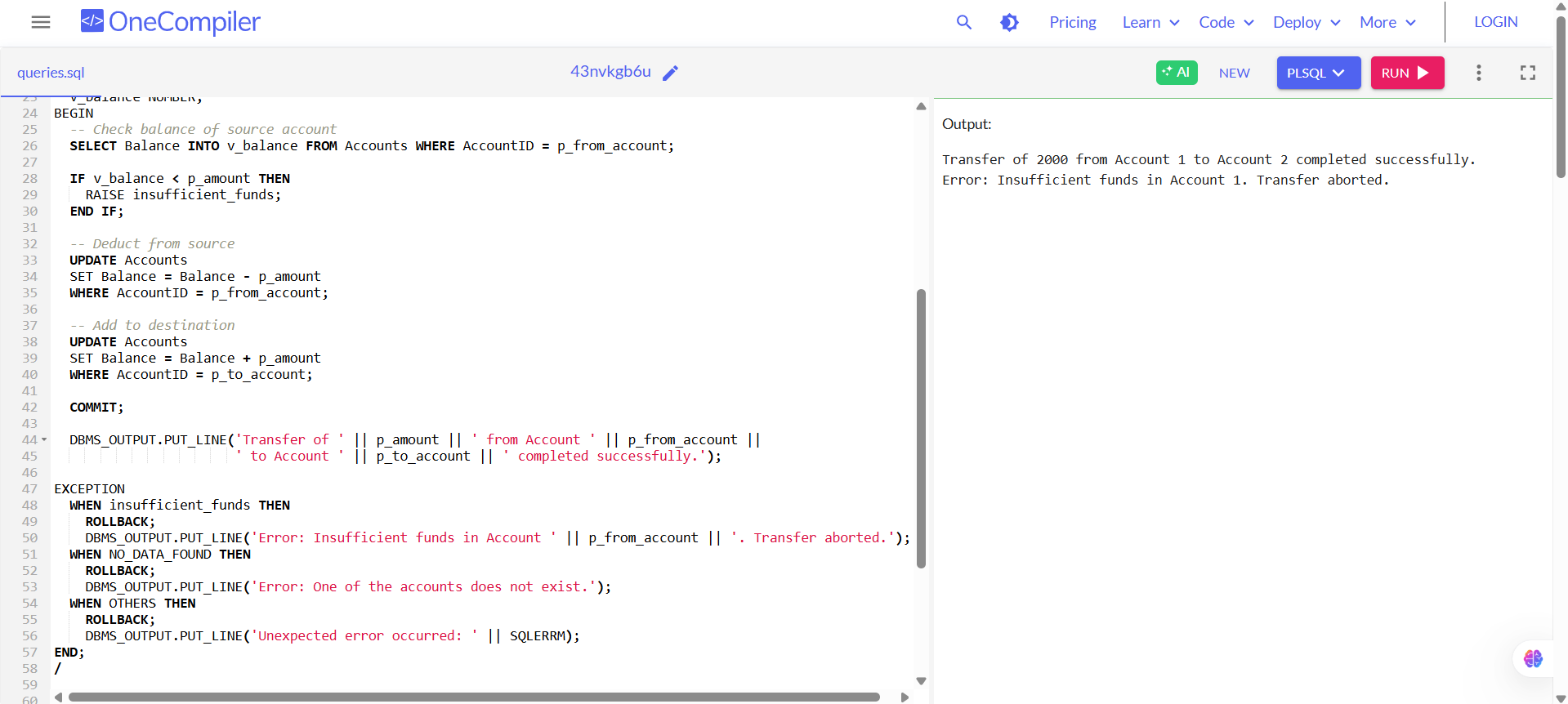
SafeTransferFunds(1, 2, 2000); -- Should succeed

SafeTransferFunds(1, 2, 4000); -- Should fail due to insufficient funds

END;

/

**OUTPUT SCREENSHOT:**

****

**EXPLANATION:**

1. The procedure transfers money between two accounts only if the source has enough balance.
2. If there’s an error like insufficient funds or invalid account, it rolls back the transaction to keep data safe.
3. It prints clear success or error messages using DBMS\_OUTPUT.PUT\_LINE so you know what happened.

**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 Employees table**

CREATE TABLE Employees (

EmployeeID NUMBER PRIMARY KEY,

EmployeeName VARCHAR2(50),

Salary NUMBER

);

**-- Insert sample data**

INSERT INTO Employees VALUES (101, 'Ravi', 50000);

INSERT INTO Employees VALUES (102, 'Meena', 60000);

COMMIT;

-- Create UpdateSalary procedure

CREATE OR REPLACE PROCEDURE UpdateSalary (

p\_employee\_id IN NUMBER,

p\_percentage IN NUMBER

) AS

BEGIN

UPDATE Employees

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

WHERE EmployeeID = p\_employee\_id;

IF SQL%ROWCOUNT = 0 THEN

RAISE NO\_DATA\_FOUND;

END IF;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Salary updated successfully for Employee ID: ' || p\_employee\_id);

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

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

WHEN OTHERS THEN

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

END;

/

-- Enable DBMS output

BEGIN

DBMS\_OUTPUT.ENABLE;

END;

/

-- **Test the procedure**

BEGIN

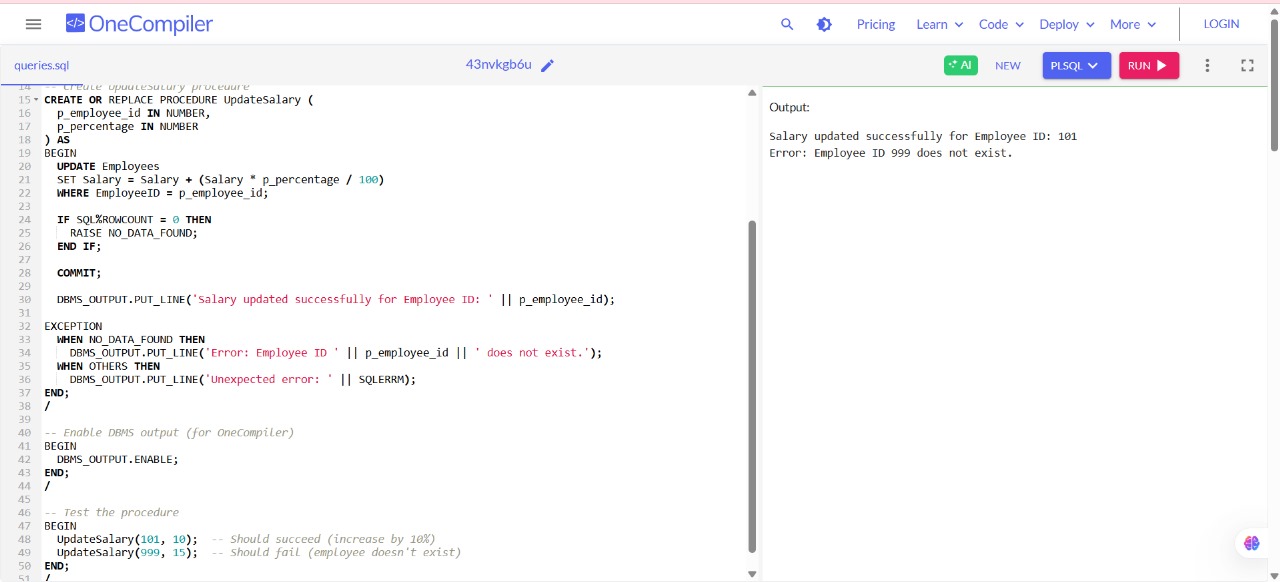
UpdateSalary(101, 10); -- Should succeed (increase by 10%)

UpdateSalary(999, 15); -- Should fail (employee doesn't exist)

END;

/

**OUTPUT SCREENSHOT:**



**EXPLANATION:**

1. The procedure updates an employee’s salary by a given percentage if the employee exists in the table.

2. If the employee ID is not found, it raises an exception and prints a clear error message.

3. Any unexpected errors are caught and logged with the error details using DBMS\_OUTPUT.PUT\_LINE.

**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 Customers table**

CREATE TABLE Customers (

CustomerID NUMBER PRIMARY KEY,

CustomerName VARCHAR2(50),

Email VARCHAR2(100)

);

**Insert sample data**

INSERT INTO Customers VALUES (1, 'Ravi', 'ravi@example.com');

INSERT INTO Customers VALUES (2, 'Meena', 'meena@example.com');

COMMIT;

**Create procedure AddNewCustomer**

CREATE OR REPLACE PROCEDURE AddNewCustomer (

p\_customer\_id IN NUMBER,

p\_customer\_name IN VARCHAR2,

p\_email IN VARCHAR2

) AS

BEGIN

INSERT INTO Customers (CustomerID, CustomerName, Email)

VALUES (p\_customer\_id, p\_customer\_name, p\_email);

COMMIT;

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

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

WHEN OTHERS THEN

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

END;

/

-- Enable DBMS output

BEGIN

DBMS\_OUTPUT.ENABLE;

END;

/

**Test the procedure**

BEGIN

AddNewCustomer(3, 'Arun', 'arun@example.com'); -- Should succeed

AddNewCustomer(2, 'Kumar', 'kumar@example.com'); -- Should fail (duplicate ID)

END;

/

**OUTPUT SCREENSHOT:**



**EXPLANATION:**

1. The procedure inserts a new customer into the Customers table if the ID is unique.

2. If a customer with the same ID already exists, it catches the duplicate key error and prints an error message.

3. Any unexpected errors are handled gracefully and logged using DBMS\_OUTPUT.PUT\_LINE

**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 SavingsAccounts table**

CREATE TABLE SavingsAccounts (

AccountID NUMBER PRIMARY KEY,

AccountHolder VARCHAR2(50),

Balance NUMBER

);

**Insert sample data**

INSERT INTO SavingsAccounts VALUES (1, 'Ravi', 10000);

INSERT INTO SavingsAccounts VALUES (2, 'Meena', 20000);

INSERT INTO SavingsAccounts VALUES (3, 'Arun', 15000);

COMMIT;

**Create the stored procedure**

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest AS

BEGIN

UPDATE SavingsAccounts

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

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Monthly interest of 1% applied to all savings accounts.');

END;

/

-- Enable DBMS Output

BEGIN

DBMS\_OUTPUT.ENABLE;

END;

/

**Call the procedure to process interest**

BEGIN

ProcessMonthlyInterest;

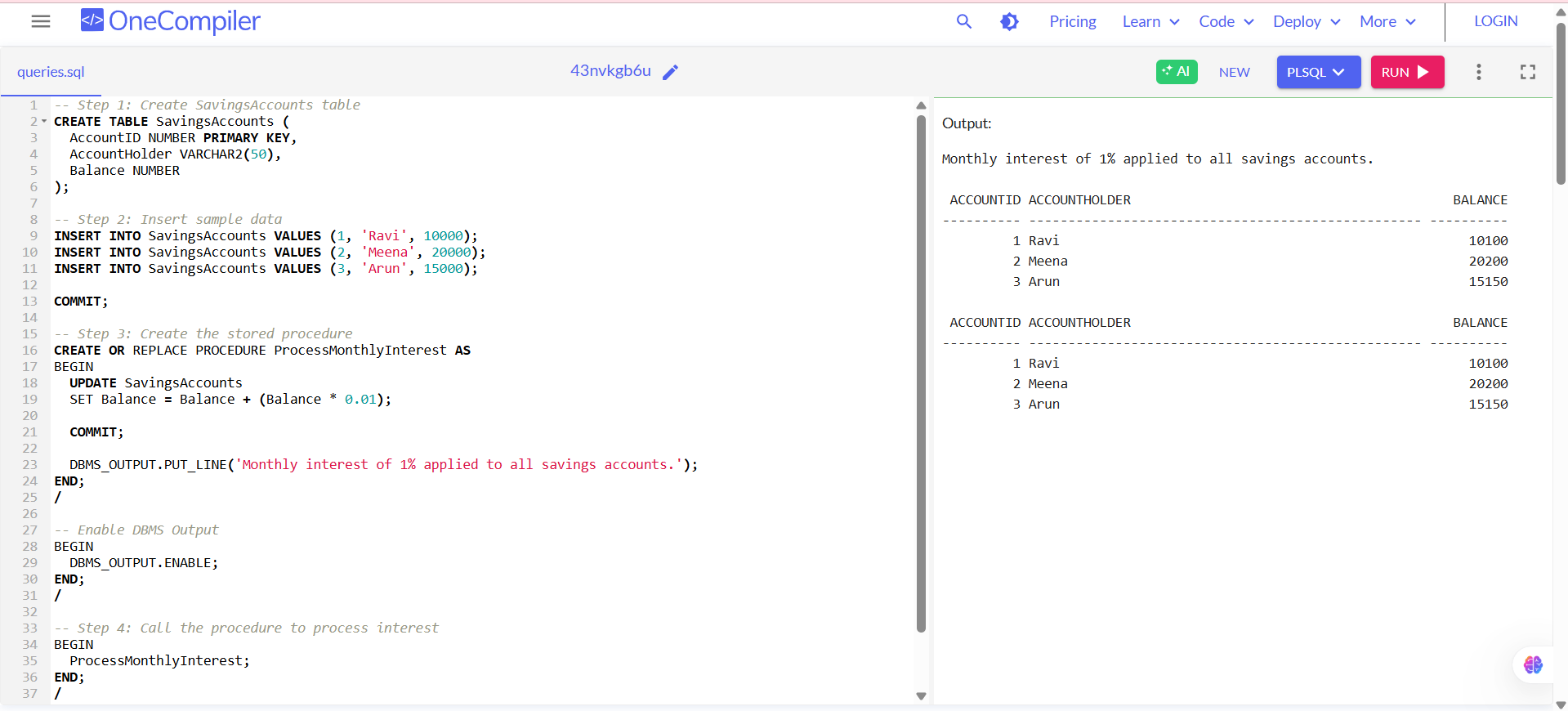
END;

/

**Check updated balances**

SELECT \* FROM SavingsAccounts;

**OUTPUT SCREENSHOT:**



**EXPLANATION:**

1.The procedure updates every savings account by adding 1% interest to their current balance.

2. It commits the changes to save updated balances permanently in the database.

3. A confirmation message is printed to let you know the process completed successfully.

**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 Employees table**

CREATE TABLE Employees (

EmployeeID NUMBER PRIMARY KEY,

EmployeeName VARCHAR2(50),

Department VARCHAR2(50),

Salary NUMBER

);

**Insert sample data**

INSERT INTO Employees VALUES (1, 'Ravi', 'HR', 50000);

INSERT INTO Employees VALUES (2, 'Meena', 'Finance', 60000);

INSERT INTO Employees VALUES (3, 'Arun', 'HR', 55000);

INSERT INTO Employees VALUES (4, 'Kumar', 'IT', 70000);

COMMIT;

**Create stored procedure UpdateEmployeeBonus**

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (

p\_department IN VARCHAR2,

p\_bonus\_percent IN NUMBER

) AS

BEGIN

UPDATE Employees

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

WHERE Department = p\_department;

IF SQL%ROWCOUNT = 0 THEN

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

ELSE

DBMS\_OUTPUT.PUT\_LINE('Bonus applied to ' || SQL%ROWCOUNT || ' employee(s) in department: ' || p\_department);

END IF;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

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

END;

/

**Enable DBMS Output**

BEGIN

DBMS\_OUTPUT.ENABLE;

END;

/

**Test the procedure**

BEGIN

UpdateEmployeeBonus('HR', 10); -- Should apply 10% bonus to HR

UpdateEmployeeBonus('Legal', 5); -- Should show "No employees found"

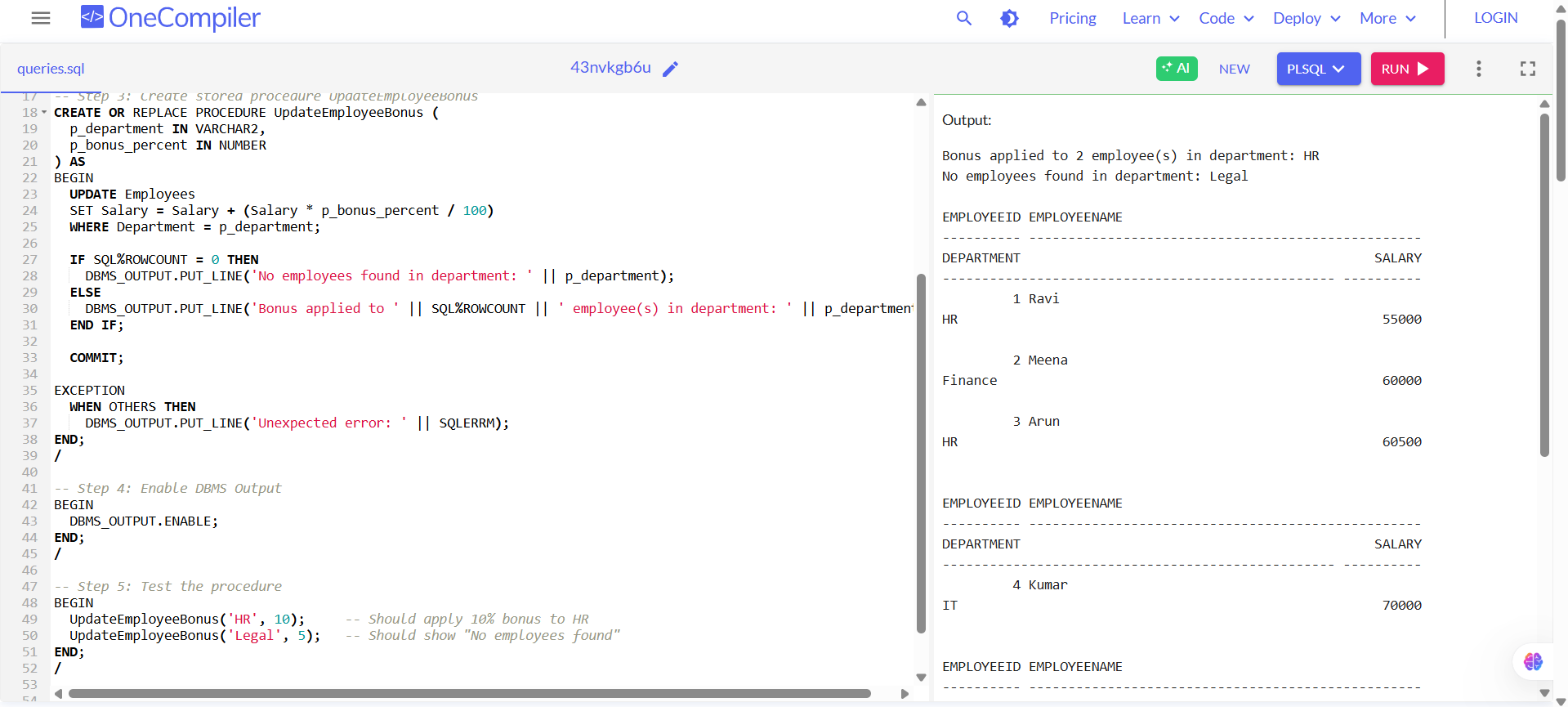
END;

/

**Check updated salaries**

SELECT \* FROM Employees;

**OUTPUT SCREENSHOT:**



**EXPLANATION:**

1. The procedure updates employee salaries by adding a bonus percentage for all employees in a given department.
2. It checks if any rows were updated — if none, it logs a message saying no employees were found.
3. It uses DBMS\_OUTPUT.PUT\_LINE to print messages and commits the updated salaries to the database.

**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:**

**-- Step 1: Create Accounts table**

CREATE TABLE Accounts (

AccountID NUMBER PRIMARY KEY,

AccountHolder VARCHAR2(50),

Balance NUMBER

);

**-- Step 2: Insert sample data**

INSERT INTO Accounts VALUES (1, 'Ravi', 5000);

INSERT INTO Accounts VALUES (2, 'Meena', 3000);

INSERT INTO Accounts VALUES (3, 'Arun', 1000);

COMMIT;

-- Step 3: Create stored procedure TransferFunds

CREATE OR REPLACE PROCEDURE TransferFunds (

p\_from\_account IN NUMBER,

p\_to\_account IN NUMBER,

p\_amount IN NUMBER

) AS

v\_balance NUMBER;

BEGIN

-- Get balance of source account

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

-- Check for sufficient funds

IF v\_balance < p\_amount THEN

DBMS\_OUTPUT.PUT\_LINE('Error: Insufficient balance in Account ID ' || p\_from\_account);

RETURN;

END IF;

-- Deduct from source

UPDATE Accounts

SET Balance = Balance - p\_amount

WHERE AccountID = p\_from\_account;

-- Add to destination

UPDATE Accounts

SET Balance = Balance + p\_amount

WHERE AccountID = p\_to\_account;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Transfer of Rs. ' || p\_amount || ' from Account ' || p\_from\_account ||

' to Account ' || p\_to\_account || ' completed successfully.');

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('Error: One or both account IDs are invalid.');

WHEN OTHERS THEN

ROLLBACK;

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

END;

/

**Enable DBMS Output**

BEGIN

DBMS\_OUTPUT.ENABLE;

END;

/

**Test the procedure**

BEGIN

TransferFunds(1, 2, 2000); -- Should succeed

TransferFunds(3, 2, 5000); -- Should fail (insufficient funds)

TransferFunds(99, 2, 1000); -- Should fail (invalid account)

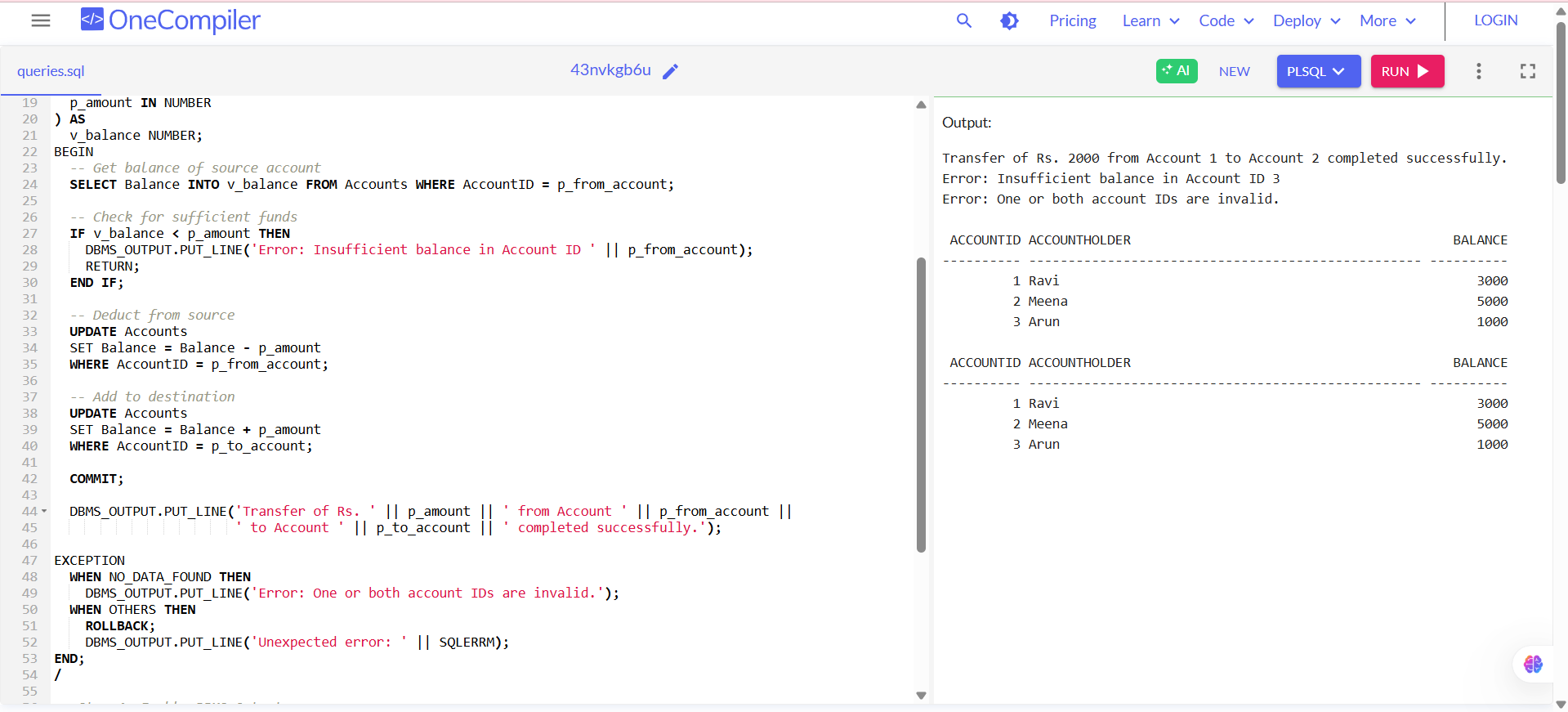
END;

/

**View updated balances**

SELECT \* FROM Accounts;

**OUTPUT SCREENSHOT:**



**EXPLANATION:**

1. The procedure transfers money from one account to another **only if the source has enough balance**.

2. It **checks for valid account IDs** and handles insufficient funds or invalid input using exceptions.

3. It uses DBMS\_OUTPUT.PUT\_LINE to log success or error messages and ensures **data integrity** with COMMIT or ROLLBACK.

**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 a function to calculate age**

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;

/

**Use the function in a test block**

BEGIN

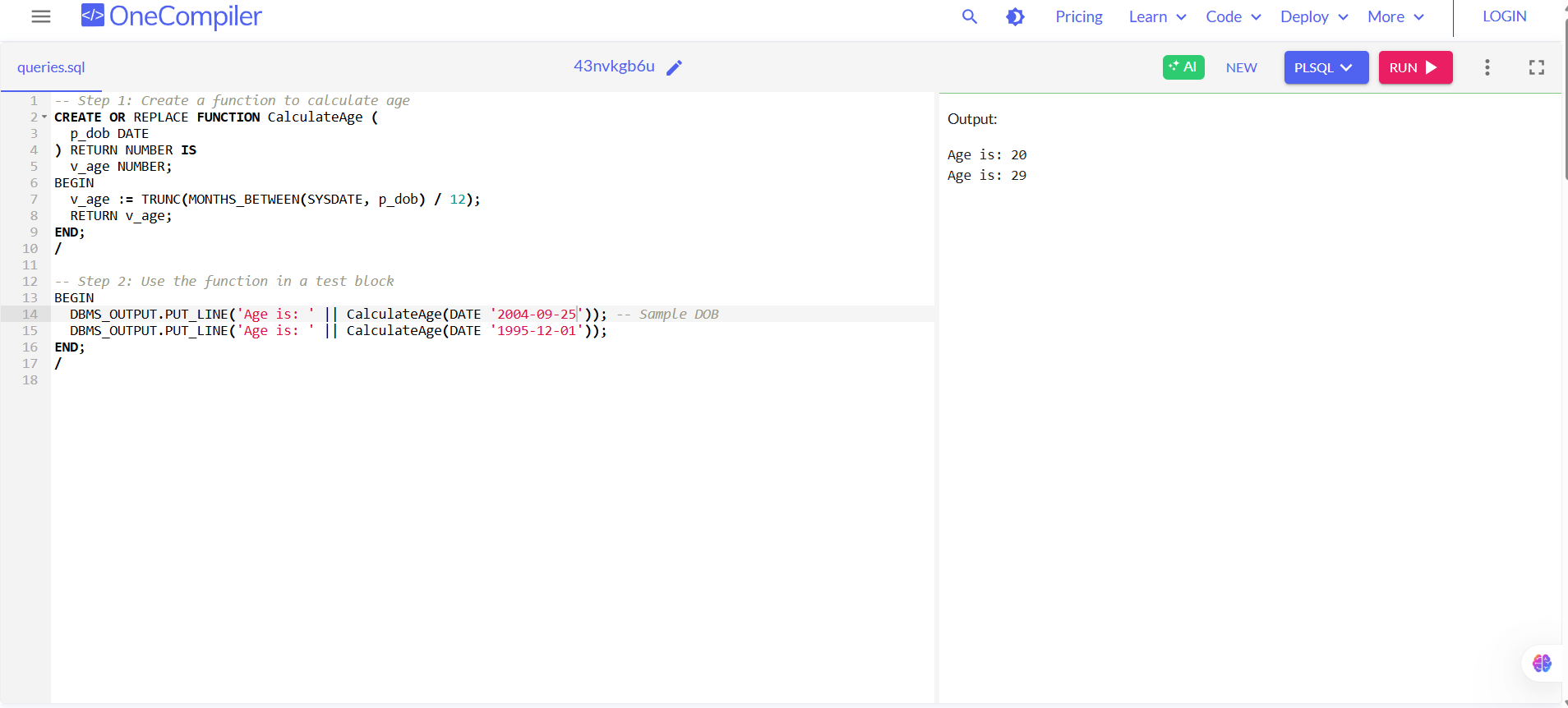
DBMS\_OUTPUT.PUT\_LINE('Age is: ' || CalculateAge(DATE '2000-06-15')); -- Sample DOB

DBMS\_OUTPUT.PUT\_LINE('Age is: ' || CalculateAge(DATE '1995-12-01'));

END;

/

**OUTPUT SCREENSHOT:**



**EXPLANATION:**

1. The function CalculateAge calculates the number of full years between today and the given date of birth.

2. It uses MONTHS\_BETWEEN and divides by 12, then truncates to get age in years.

3.You can use it in other procedures, queries, or eligibility checks easily.

**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 the function**

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment (

p\_loan\_amount IN NUMBER,

p\_annual\_interest\_rate IN NUMBER,

p\_loan\_duration\_years IN NUMBER

) RETURN NUMBER IS

v\_monthly\_rate NUMBER;

v\_total\_months NUMBER;

v\_emi NUMBER;

BEGIN

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

v\_total\_months := p\_loan\_duration\_years \* 12;

IF v\_monthly\_rate = 0 THEN

-- Simple case: no interest

v\_emi := p\_loan\_amount / v\_total\_months;

ELSE

-- EMI formula

v\_emi := (p\_loan\_amount \* v\_monthly\_rate \* POWER(1 + v\_monthly\_rate, v\_total\_months)) /

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

END IF;

RETURN ROUND(v\_emi, 2); -- Round to 2 decimal places

END;

/

**Test the function**

BEGIN

DBMS\_OUTPUT.PUT\_LINE('EMI: ₹' || CalculateMonthlyInstallment(500000, 8.5, 5)); -- 5 lakh, 8.5% interest, 5 years

DBMS\_OUTPUT.PUT\_LINE('EMI: ₹' || CalculateMonthlyInstallment(300000, 0, 3)); -- No interest case

END;

/

**OUTPUT SCREENSHOT:**



**EXPLANATION:**

1. The function calculates EMI using the standard formula based on interest rate and loan duration.

2. If interest is 0%, it handles the case with simple equal monthly division.

3. It returns the EMI rounded to 2 decimal places, ideal for billing.

**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 Accounts table**

CREATE TABLE Accounts (

AccountID NUMBER PRIMARY KEY,

AccountHolder VARCHAR2(50),

Balance NUMBER

);

**Insert sample data**

INSERT INTO Accounts VALUES (1, 'Ravi', 5000);

INSERT INTO Accounts VALUES (2, 'Meena', 2000);

INSERT INTO Accounts VALUES (3, 'Arun', 8000);

COMMIT;

**Create the function HasSufficientBalance**

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

RETURN TRUE;

ELSE

RETURN FALSE;

END IF;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('Error: Account ID ' || p\_account\_id || ' does not exist.');

RETURN FALSE;

WHEN OTHERS THEN

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

RETURN FALSE;

END;

/

**Test the function**

DECLARE

result BOOLEAN;

BEGIN

result := HasSufficientBalance(1, 3000); -- Should return TRUE

IF result THEN

DBMS\_OUTPUT.PUT\_LINE('Account 1 has sufficient balance.');

ELSE

DBMS\_OUTPUT.PUT\_LINE('Account 1 does NOT have sufficient balance.');

END IF;

result := HasSufficientBalance(2, 3000); -- Should return FALSE

IF result THEN

DBMS\_OUTPUT.PUT\_LINE('Account 2 has sufficient balance.');

ELSE

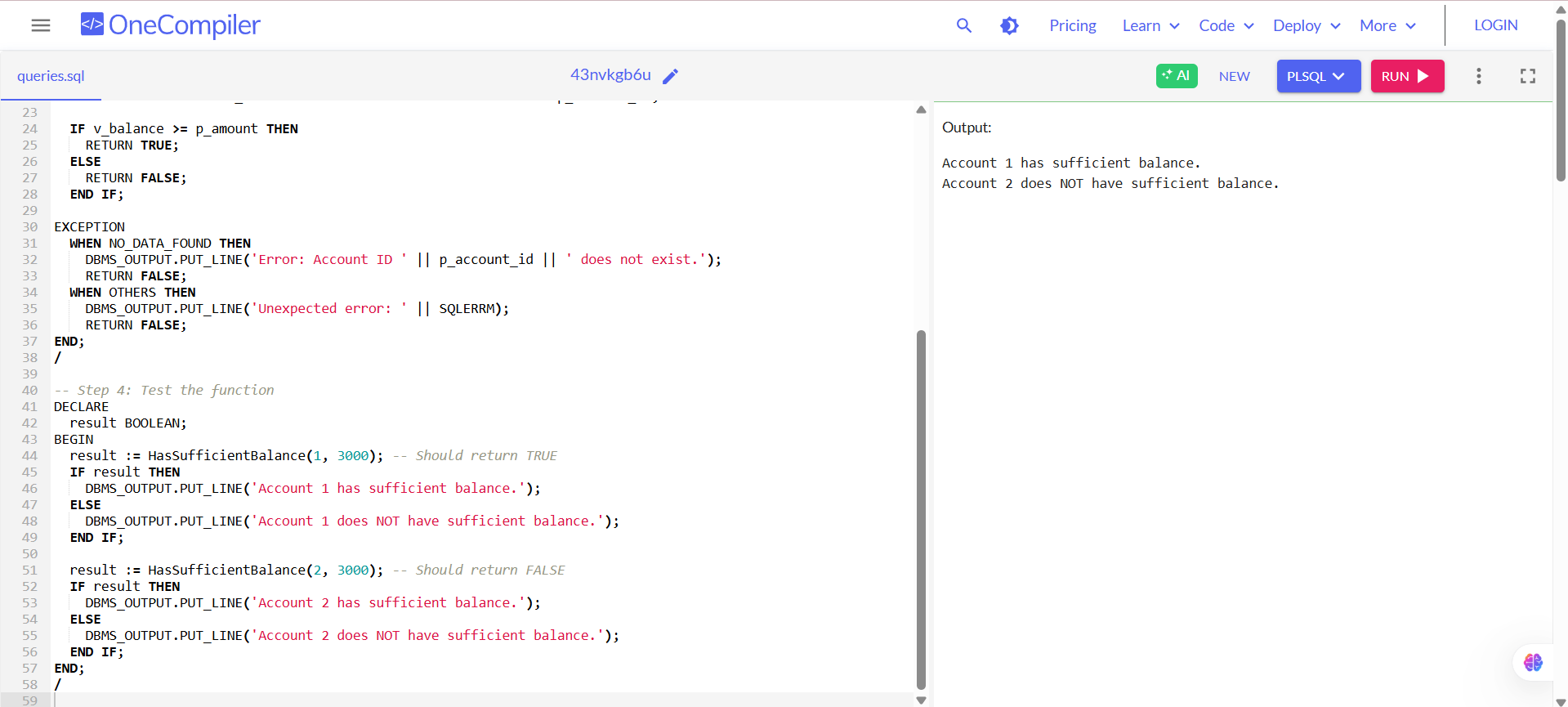
DBMS\_OUTPUT.PUT\_LINE('Account 2 does NOT have sufficient balance.');

END IF;

END;

/

**OUTPUT SCREENSHOT:**



**EXPLANATION:**

1. The function checks if the account has a balance greater than or equal to the specified amount.

2. It returns TRUE if the condition is met, otherwise FALSE.

3. It handles missing account IDs and unexpected errors using exception handling.

**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 the Customers table**

CREATE TABLE Customers (

CustomerID NUMBER PRIMARY KEY,

CustomerName VARCHAR2(50),

Email VARCHAR2(100),

LastModified DATE

);

**Insert sample data**

INSERT INTO Customers VALUES (1, 'Ravi', 'ravi@example.com', SYSDATE);

INSERT INTO Customers VALUES (2, 'Meena', 'meena@example.com', SYSDATE);

COMMIT;

**Create the trigger**

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON Customers

FOR EACH ROW

BEGIN

:NEW.LastModified := SYSDATE;

END;

/

**Test by updating a customer**

UPDATE Customers

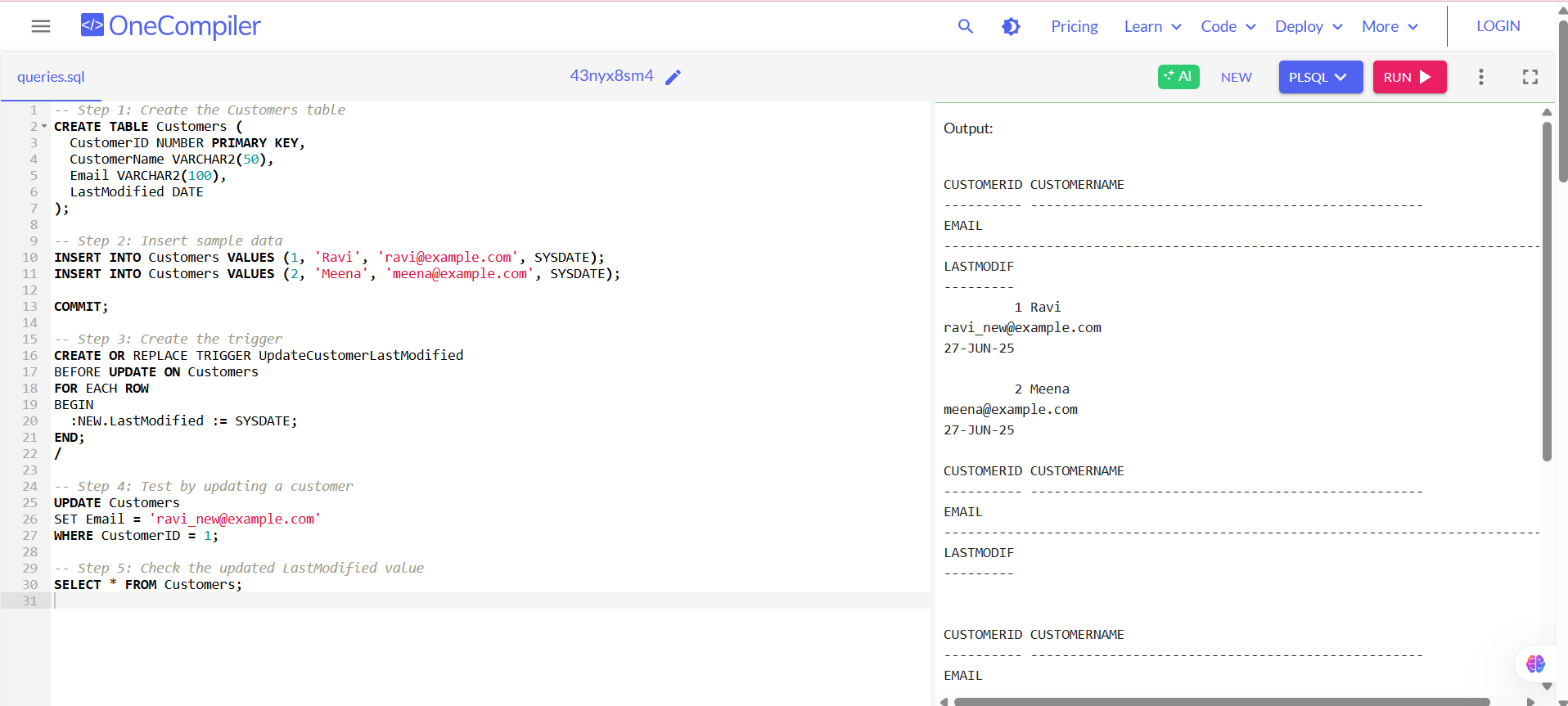
SET Email = 'ravi\_new@example.com'

WHERE CustomerID = 1;

**Check the updated LastModified value**

SELECT \* FROM Customers;

**OUTPUT SCREENSHOT:**



**EXPLANATION:**

1. The trigger fires **automatically before any update** on the Customers table.2

2. It updates the LastModified field to the **current date and time (SYSDATE)**.

3. This ensures accurate tracking of the **last time the record was changed**, without needing manual updates.

**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 the Transactions table**

CREATE TABLE Transactions (

TransactionID NUMBER PRIMARY KEY,

AccountID NUMBER,

Amount NUMBER,

TransactionDate DATE

);

**Create the AuditLog table**

CREATE TABLE AuditLog (

LogID NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,

TransactionID NUMBER,

AccountID NUMBER,

Amount NUMBER,

Action VARCHAR2(20),

LoggedAt DATE

);

**Create the trigger**

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog (TransactionID, AccountID, Amount, Action, LoggedAt)

VALUES (:NEW.TransactionID, :NEW.AccountID, :NEW.Amount, 'INSERT', SYSDATE);

END;

/

**Insert test data into Transactions table**

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

VALUES (1, 101, 5000, SYSDATE);

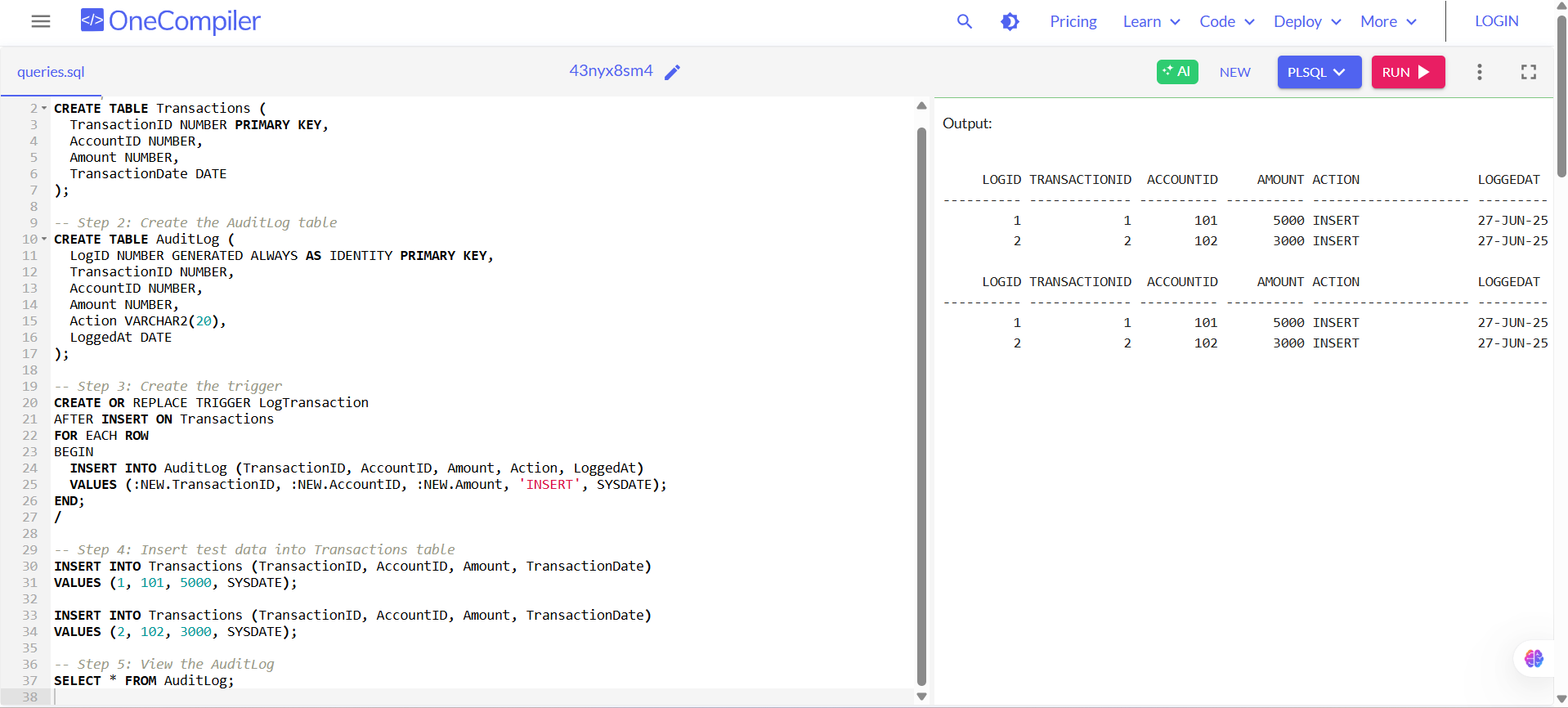
INSERT INTO Transactions (TransactionID, AccountID, Amount, TransactionDate)

VALUES (2, 102, 3000, SYSDATE);

**View the AuditLog**

SELECT \* FROM AuditLog;

**OUTPUT SCREENSHOT:**



**EXPLANATION:**

 The trigger **fires after a new transaction is inserted** into the Transactions table.

 It automatically inserts a log record into the AuditLog table with transaction details.

 This ensures a **secure and automatic record of all transaction activity** for auditing purposes.

**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.

**SOLUTION:**

**Create Accounts table**

CREATE TABLE Accounts (

AccountID NUMBER PRIMARY KEY,

AccountHolder VARCHAR2(50),

Balance NUMBER

);

**Create Transactions table**

CREATE TABLE Transactions (

TransactionID NUMBER PRIMARY KEY,

AccountID NUMBER,

Amount NUMBER, -- Positive for deposit, negative for withdrawal

TransactionDate DATE

);

**Insert sample account data**

INSERT INTO Accounts VALUES (101, 'Ravi', 5000);

INSERT INTO Accounts VALUES (102, 'Meena', 3000);

COMMIT;

**Create the trigger to enforce transaction rules**

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

DECLARE

v\_balance NUMBER;

BEGIN

-- Get current account balance

SELECT Balance INTO v\_balance FROM Accounts WHERE AccountID = :NEW.AccountID;

-- Rule 1: Deposit must be positive

IF :NEW.Amount > 0 THEN

NULL; -- deposit is allowed

-- Rule 2: Withdrawal must not exceed balance

ELSIF :NEW.Amount < 0 THEN

IF ABS(:NEW.Amount) > v\_balance THEN

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

END IF;

-- Rule 3: Amount cannot be zero

ELSE

RAISE\_APPLICATION\_ERROR(-20002, 'Transaction amount must not be zero.');

END IF;

END;

/

**Test Cases**

-- ✅ Valid deposit

INSERT INTO Transactions VALUES (1, 101, 2000, SYSDATE);

-- ❌ Invalid withdrawal (exceeds balance)

INSERT INTO Transactions VALUES (2, 101, -10000, SYSDATE); -- Should raise error

-- ❌ Invalid zero amount

INSERT INTO Transactions VALUES (3, 101, 0, SYSDATE); -- Should raise error

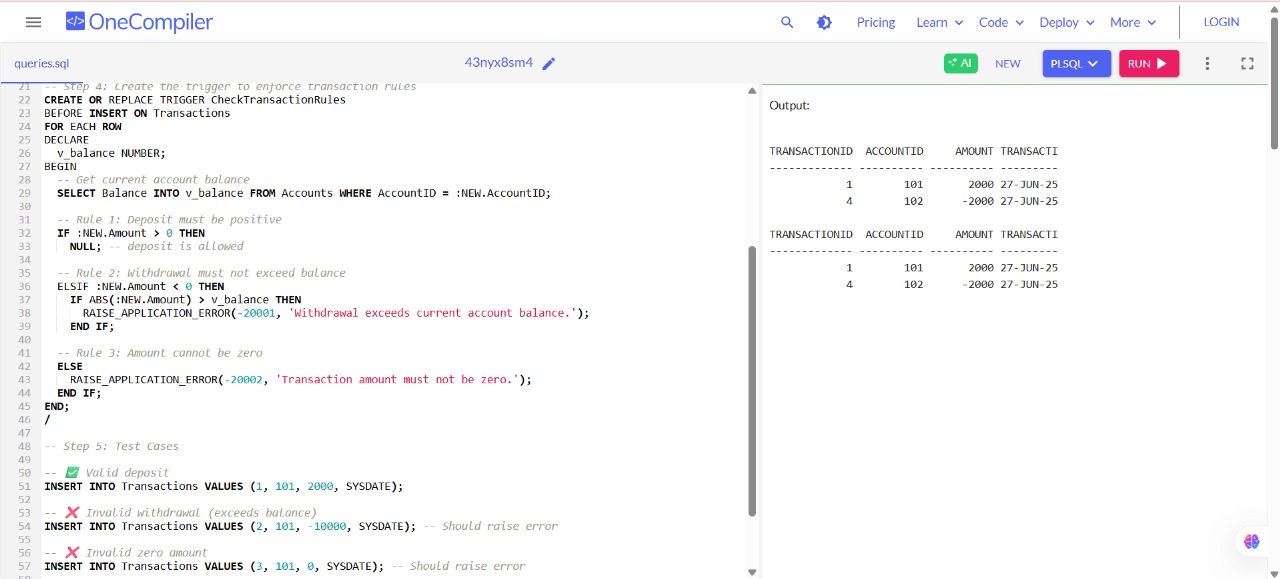
-- ✅ Valid withdrawal

INSERT INTO Transactions VALUES (4, 102, -2000, SYSDATE);

**View valid transactions**

SELECT \* FROM Transactions;

**OUTPUT SCREENSHOT:**



**EXPLANATION:**

 This **BEFORE INSERT trigger** checks the type of transaction and the current balance of the account.

 It **prevents invalid deposits** (zero/negative) and **blocks withdrawals** that would overdraw the account.

 It uses RAISE\_APPLICATION\_ERROR to stop the transaction with a meaningful error message.

**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 Customers table

CREATE TABLE Customers (

CustomerID NUMBER PRIMARY KEY,

CustomerName VARCHAR2(100)

);

-- Create Transactions table

CREATE TABLE Transactions (

TransactionID NUMBER PRIMARY KEY,

CustomerID NUMBER,

TransactionDate DATE,

Amount NUMBER(10,2),

Description VARCHAR2(200),

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

-- Insert sample data into Customers

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

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

-- Insert sample data into Transactions

-- Assume SYSDATE is June 2025 when this runs

INSERT INTO Transactions VALUES (101, 1, TO\_DATE('2025-06-01', 'YYYY-MM-DD'), 500.00, 'Deposit');

INSERT INTO Transactions VALUES (102, 1, TO\_DATE('2025-06-10', 'YYYY-MM-DD'), -200.00, 'Withdrawal');

INSERT INTO Transactions VALUES (103, 2, TO\_DATE('2025-06-05', 'YYYY-MM-DD'), 1000.00, 'Salary Credit');

INSERT INTO Transactions VALUES (104, 2, TO\_DATE('2025-05-25', 'YYYY-MM-DD'), -100.00, 'Old Purchase'); -- Should be skipped

-- Commit the data

COMMIT;

-- Enable output

SET SERVEROUTPUT ON;

-- PL/SQL block using explicit cursor to generate monthly statements

DECLARE

CURSOR GenerateMonthlyStatements IS

SELECT

c.CustomerID,

c.CustomerName,

t.TransactionDate,

t.Amount,

t.Description

FROM

Customers c

JOIN

Transactions t ON c.CustomerID = t.CustomerID

WHERE

EXTRACT(MONTH FROM t.TransactionDate) = EXTRACT(MONTH FROM SYSDATE)

AND EXTRACT(YEAR FROM t.TransactionDate) = EXTRACT(YEAR FROM SYSDATE)

ORDER BY

c.CustomerID, t.TransactionDate;

rec GenerateMonthlyStatements%ROWTYPE;

prev\_customer\_id Customers.CustomerID%TYPE := NULL;

BEGIN

OPEN GenerateMonthlyStatements;

LOOP

FETCH GenerateMonthlyStatements INTO rec;

EXIT WHEN GenerateMonthlyStatements%NOTFOUND;

IF prev\_customer\_id IS NULL OR prev\_customer\_id != rec.CustomerID THEN

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

DBMS\_OUTPUT.PUT\_LINE('Monthly Statement for Customer: ' || rec.CustomerName || ' (ID: ' || rec.CustomerID || ')');

DBMS\_OUTPUT.PUT\_LINE('Date | Amount | Description');

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

prev\_customer\_id := rec.CustomerID;

END IF;

DBMS\_OUTPUT.PUT\_LINE(TO\_CHAR(rec.TransactionDate, 'YYYY-MM-DD') || ' | ' ||

TO\_CHAR(rec.Amount, '99999.99') || ' | ' || rec.Description);

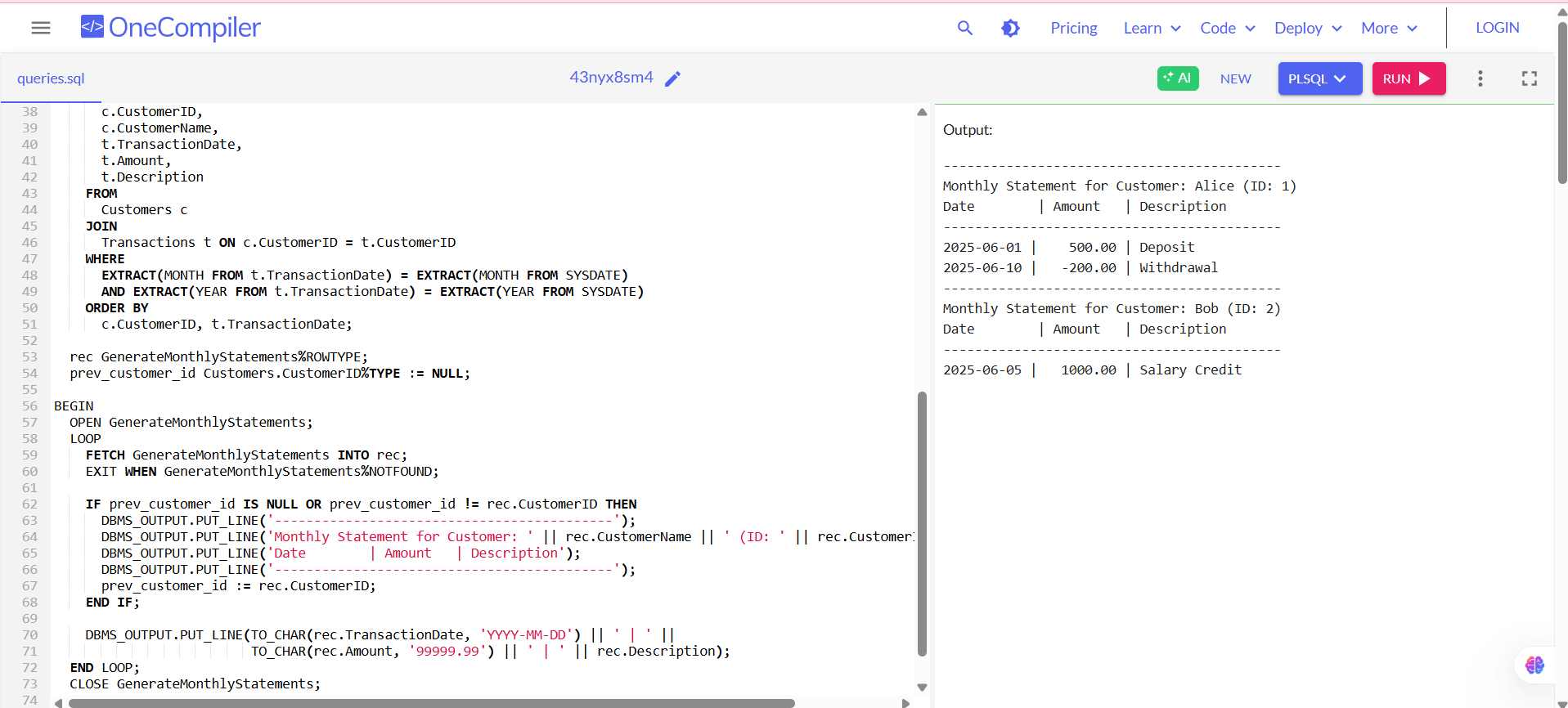
END LOOP;

CLOSE GenerateMonthlyStatements;

END;

/

**OUTPUT SCREENSHOT:**



**EXPLANATION:**

 This PL/SQL program uses an **explicit cursor** to fetch all **customer transactions** from the current month.

 It prints a **monthly statement** for each customer, showing date, amount, and description.

 The output is displayed using DBMS\_OUTPUT.PUT\_LINE in a formatted manner.

**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 the Accounts table**

CREATE TABLE Accounts (

AccountID NUMBER PRIMARY KEY,

AccountHolder VARCHAR2(50),

Balance NUMBER

);

**Insert sample data**

INSERT INTO Accounts VALUES (101, 'Ravi', 5000);

INSERT INTO Accounts VALUES (102, 'Meena', 3000);

INSERT INTO Accounts VALUES (103, 'Arun', 1000);

COMMIT;

**PL/SQL block to apply annual fee using an explicit cursor**

DECLARE

CURSOR account\_cursor IS

SELECT AccountID, Balance FROM Accounts;

v\_account\_id Accounts.AccountID%TYPE;

v\_balance Accounts.Balance%TYPE;

v\_fee CONSTANT NUMBER := 200; -- Annual maintenance fee

BEGIN

DBMS\_OUTPUT.PUT\_LINE('--- Applying Annual Maintenance Fee ---');

OPEN account\_cursor;

LOOP

FETCH account\_cursor INTO v\_account\_id, v\_balance;

EXIT WHEN account\_cursor%NOTFOUND;

-- Deduct fee and update balance

UPDATE Accounts

SET Balance = Balance - v\_fee

WHERE AccountID = v\_account\_id;

DBMS\_OUTPUT.PUT\_LINE('Rs. ' || v\_fee || ' deducted from Account ID: ' || v\_account\_id);

END LOOP;

CLOSE account\_cursor;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('--- Annual Fees Applied Successfully ---');

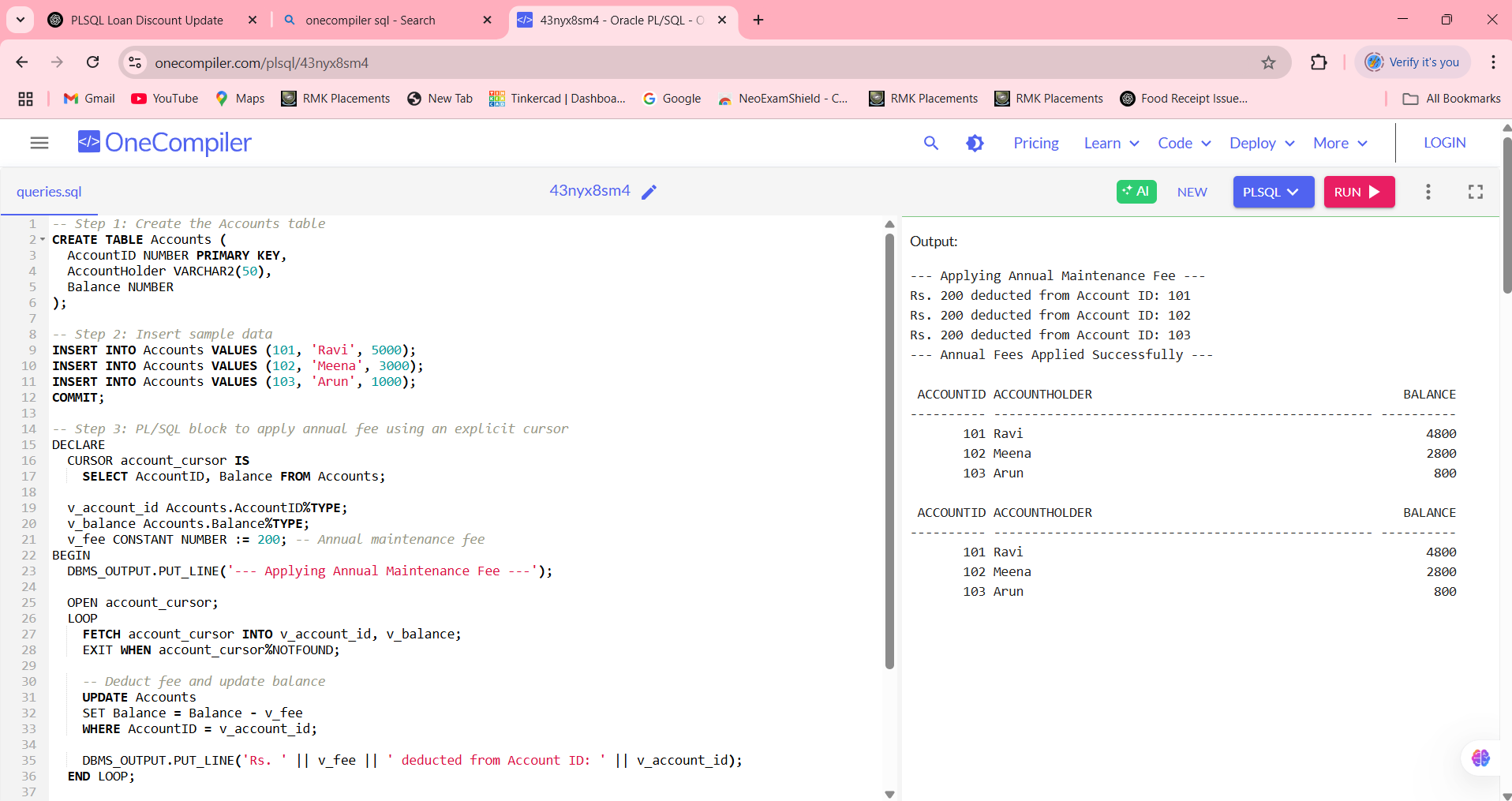
END;

/

-- Step 4: View updated balances

SELECT \* FROM Accounts;

**OUTPUT SCREENSHOT:**



**EXPLANATION:**

 A **cursor** is used to loop through each account and apply the annual fee.

 The balance is **updated** by subtracting Rs. 200 from each account.

 DBMS\_OUTPUT gives a confirmation for each deduction and commits all updates.

**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 Loans table**

CREATE TABLE Loans (

LoanID NUMBER PRIMARY KEY,

CustomerName VARCHAR2(50),

LoanAmount NUMBER,

InterestRate NUMBER

);

**Insert sample data**

INSERT INTO Loans VALUES (1, 'Ravi', 80000, 7.5);

INSERT INTO Loans VALUES (2, 'Meena', 40000, 6.5);

INSERT INTO Loans VALUES (3, 'Arun', 60000, 7.0);

COMMIT;

**PL/SQL block to update interest rates using explicit cursor**

DECLARE

CURSOR loan\_cursor IS

SELECT LoanID, LoanAmount, InterestRate FROM Loans;

v\_loan\_id Loans.LoanID%TYPE;

v\_loan\_amount Loans.LoanAmount%TYPE;

v\_interest\_rate Loans.InterestRate%TYPE;

BEGIN

DBMS\_OUTPUT.PUT\_LINE('--- Updating Loan Interest Rates Based on New Policy ---');

OPEN loan\_cursor;

LOOP

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

EXIT WHEN loan\_cursor%NOTFOUND;

-- Apply new policy:

-- If loan > ₹50,000, increase rate by 0.5%

-- Else, increase rate by 0.25%

IF v\_loan\_amount > 50000 THEN

v\_interest\_rate := v\_interest\_rate + 0.5;

ELSE

v\_interest\_rate := v\_interest\_rate + 0.25;

END IF;

-- Update the loan record

UPDATE Loans

SET InterestRate = v\_interest\_rate

WHERE LoanID = v\_loan\_id;

DBMS\_OUTPUT.PUT\_LINE('Loan ID ' || v\_loan\_id || ': New Interest Rate = ' || v\_interest\_rate || '%');

END LOOP;

CLOSE loan\_cursor;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('--- Interest Rates Updated Successfully ---');

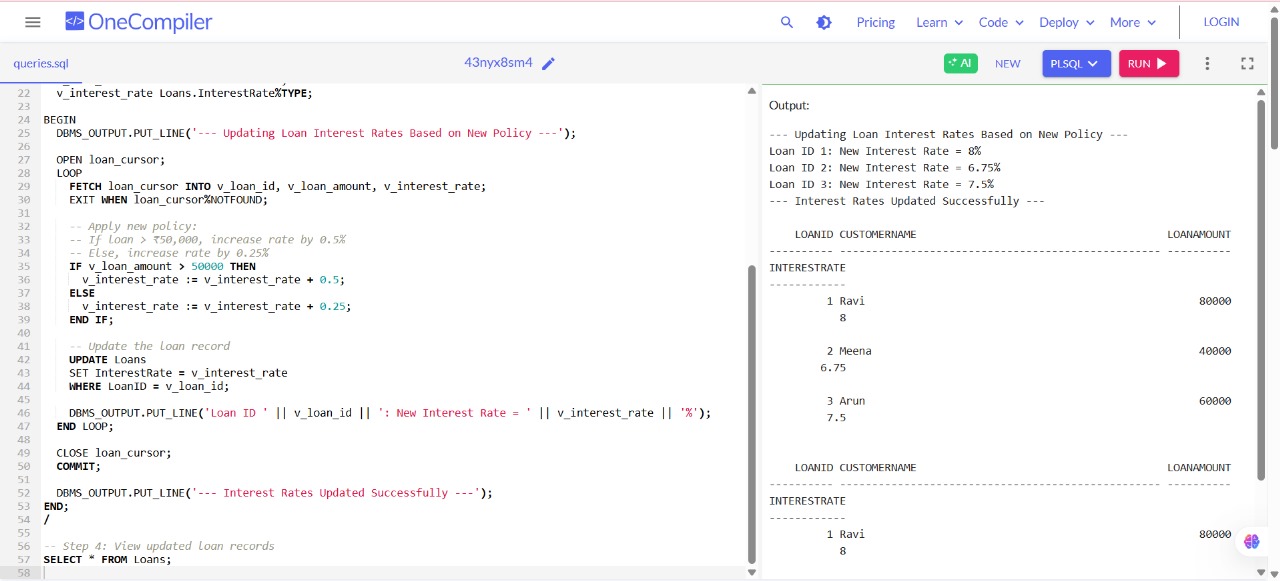
END;

/

-- Step 4: View updated loan records

SELECT \* FROM Loans;

**OUTPUT SCREENSHOT:**



**EXPLANATION:**

 The cursor **iterates over each loan**, checking the loan amount to decide how much to increase the interest rate.

 Loans above ₹50,000 get a **0.5% hike**, while others get a **0.25% hike**.

 The updated rates are **committed to the database**, and the result is displayed using DBMS\_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,

CustomerName VARCHAR2(50),

Email VARCHAR2(100),

Balance NUMBER

);

CREATE OR REPLACE PACKAGE CustomerManagement AS

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

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

FUNCTION GetBalance(p\_id NUMBER) RETURN NUMBER;

END CustomerManagement;

/

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

-- Procedure to add a new customer

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

BEGIN

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

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

EXCEPTION

WHEN OTHERS THEN

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

END;

-- Procedure to update name and email

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

BEGIN

UPDATE Customers

SET CustomerName = p\_name, Email = p\_email

WHERE CustomerID = p\_id;

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

EXCEPTION

WHEN OTHERS THEN

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

END;

-- Function to return customer balance

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;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

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

RETURN NULL;

WHEN OTHERS THEN

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

RETURN NULL;

END;

END CustomerManagement;

/

BEGIN

-- Add customer

CustomerManagement.AddCustomer(1, 'Ravi', 'ravi@example.com', 5000);

CustomerManagement.AddCustomer(2, 'Meena', 'meena@example.com', 3000);

-- Update customer

CustomerManagement.UpdateCustomer(1, 'Ravi Kumar', 'ravi.kumar@example.com');

-- Get balance

DBMS\_OUTPUT.PUT\_LINE('Balance for Customer 1: ₹' || CustomerManagement.GetBalance(1));

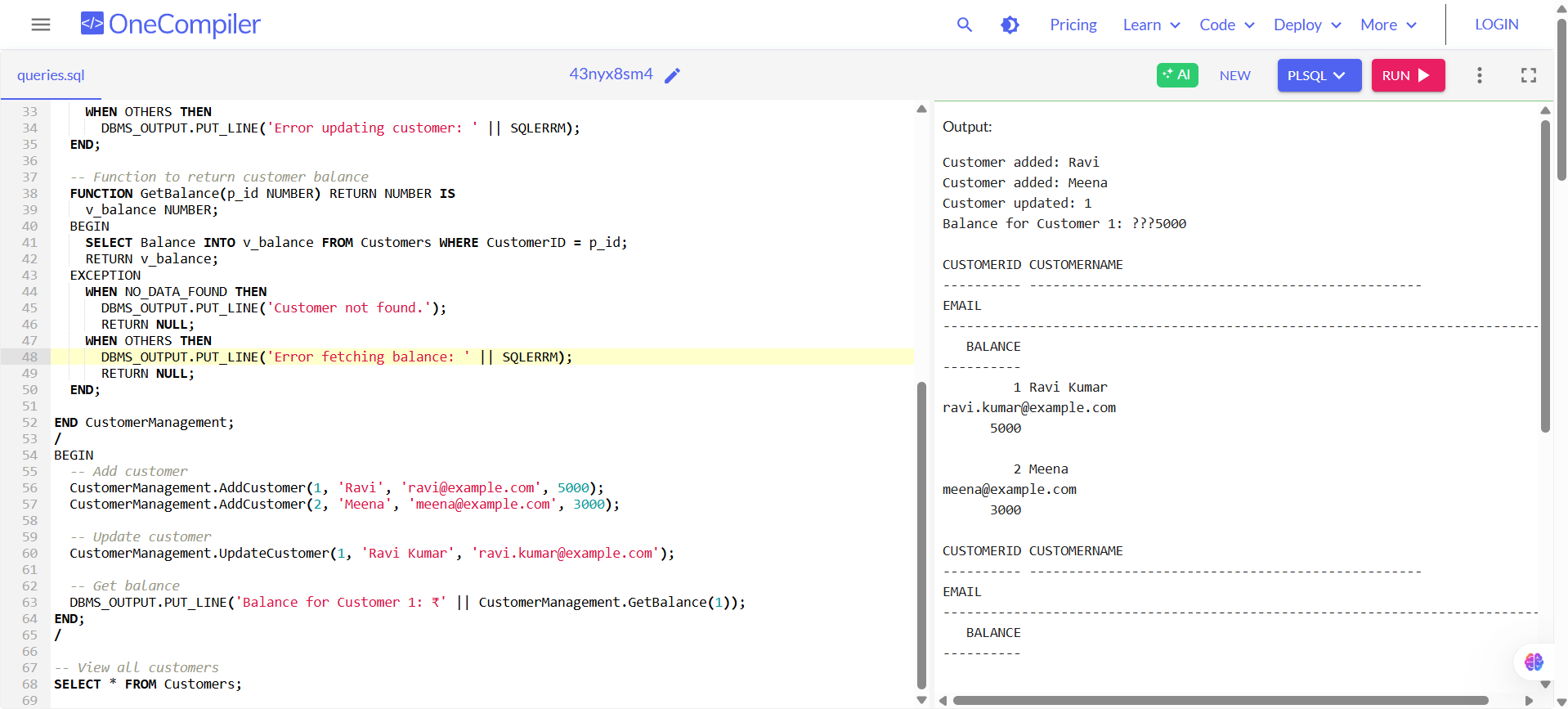
END;

/

-- View all customers

SELECT \* FROM Customers;

**OUTPUT SCREENSHOT:**



**EXPLANATION:**

 The **package groups customer operations** into one logical unit for easy reuse and maintenance.

 Procedures handle **data entry and updates**, while the function allows balance checking.

 It improves structure, code reuse, and **modularity** in large banking systems.

**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,

EmpName VARCHAR2(50),

Department VARCHAR2(50),

MonthlySalary NUMBER

);

CREATE OR REPLACE PACKAGE EmployeeManagement AS

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

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

FUNCTION CalculateAnnualSalary(p\_id NUMBER) RETURN NUMBER;

END EmployeeManagement;

/

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

-- Procedure to hire a new employee

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

BEGIN

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

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

EXCEPTION

WHEN OTHERS THEN

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

END;

-- Procedure to update employee details

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

BEGIN

UPDATE Employees

SET EmpName = p\_name, Department = p\_dept

WHERE EmpID = p\_id;

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

EXCEPTION

WHEN OTHERS THEN

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

END;

-- Function to calculate annual salary

FUNCTION CalculateAnnualSalary(p\_id NUMBER) RETURN NUMBER IS

v\_salary NUMBER;

BEGIN

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

RETURN v\_salary \* 12;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

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

RETURN NULL;

WHEN OTHERS THEN

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

RETURN NULL;

END;

END EmployeeManagement;

/

BEGIN

-- Hire employees

EmployeeManagement.HireEmployee(101, 'Anjali', 'HR', 40000);

EmployeeManagement.HireEmployee(102, 'Rahul', 'IT', 55000);

-- Update an employee

EmployeeManagement.UpdateEmployee(101, 'Anjali Sharma', 'Admin');

-- Get annual salary

DBMS\_OUTPUT.PUT\_LINE('Annual Salary of Emp 102: ₹' || EmployeeManagement.CalculateAnnualSalary(102));

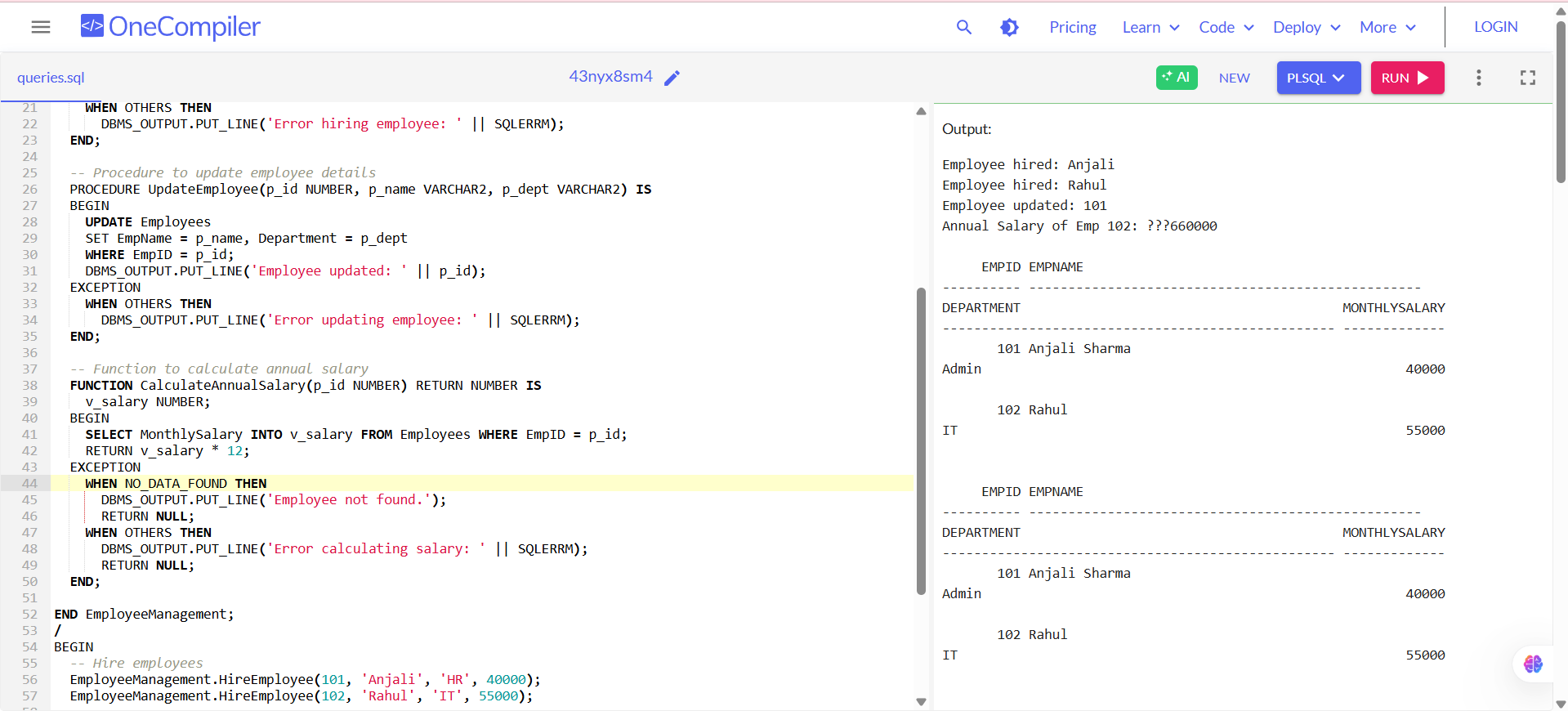
END;

/

-- View employees

SELECT \* FROM Employees;

**OUTPUT SCREENSHOT:**



**EXPLANATION:**

 The **EmployeeManagement** package organizes all employee operations in one place.

 It simplifies employee **hiring, updating, and salary calculations**.

 Great for large applications needing **modular, reusable, and secure code**!

**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 Accounts table**

CREATE TABLE Accounts (

AccountID NUMBER PRIMARY KEY,

CustomerID NUMBER,

Balance NUMBER(10,2),

Status VARCHAR2(20) -- 'ACTIVE' or 'CLOSED'

);

-- Insert sample data

INSERT INTO Accounts VALUES (101, 1, 5000, 'ACTIVE');

INSERT INTO Accounts VALUES (102, 1, 2500, 'ACTIVE');

INSERT INTO Accounts VALUES (103, 2, 7000, 'ACTIVE');

COMMIT;

**Create Package Specification**

CREATE OR REPLACE PACKAGE AccountOperations AS

PROCEDURE OpenAccount(p\_AccountID NUMBER, p\_CustomerID NUMBER, p\_InitialBalance NUMBER);

PROCEDURE CloseAccount(p\_AccountID NUMBER);

FUNCTION GetTotalBalance(p\_CustomerID NUMBER) RETURN NUMBER;

END AccountOperations;

/

**Create Package Body**

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

PROCEDURE OpenAccount(p\_AccountID NUMBER, p\_CustomerID NUMBER, p\_InitialBalance NUMBER) IS

BEGIN

INSERT INTO Accounts (AccountID, CustomerID, Balance, Status)

VALUES (p\_AccountID, p\_CustomerID, p\_InitialBalance, 'ACTIVE');

DBMS\_OUTPUT.PUT\_LINE('Account ' || p\_AccountID || ' opened for Customer ' || p\_CustomerID);

END;

PROCEDURE CloseAccount(p\_AccountID NUMBER) IS

BEGIN

UPDATE Accounts

SET Status = 'CLOSED'

WHERE AccountID = p\_AccountID;

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

END;

FUNCTION GetTotalBalance(p\_CustomerID NUMBER) RETURN NUMBER IS

v\_total NUMBER := 0;

BEGIN

SELECT SUM(Balance)

INTO v\_total

FROM Accounts

WHERE CustomerID = p\_CustomerID AND Status = 'ACTIVE';

RETURN NVL(v\_total, 0);

END;

END AccountOperations;

/

Enable output

SET SERVEROUTPUT ON;

**Test the package**

BEGIN

-- Open a new account

AccountOperations.OpenAccount(104, 1, 1000);

-- Close an existing account

AccountOperations.CloseAccount(102);

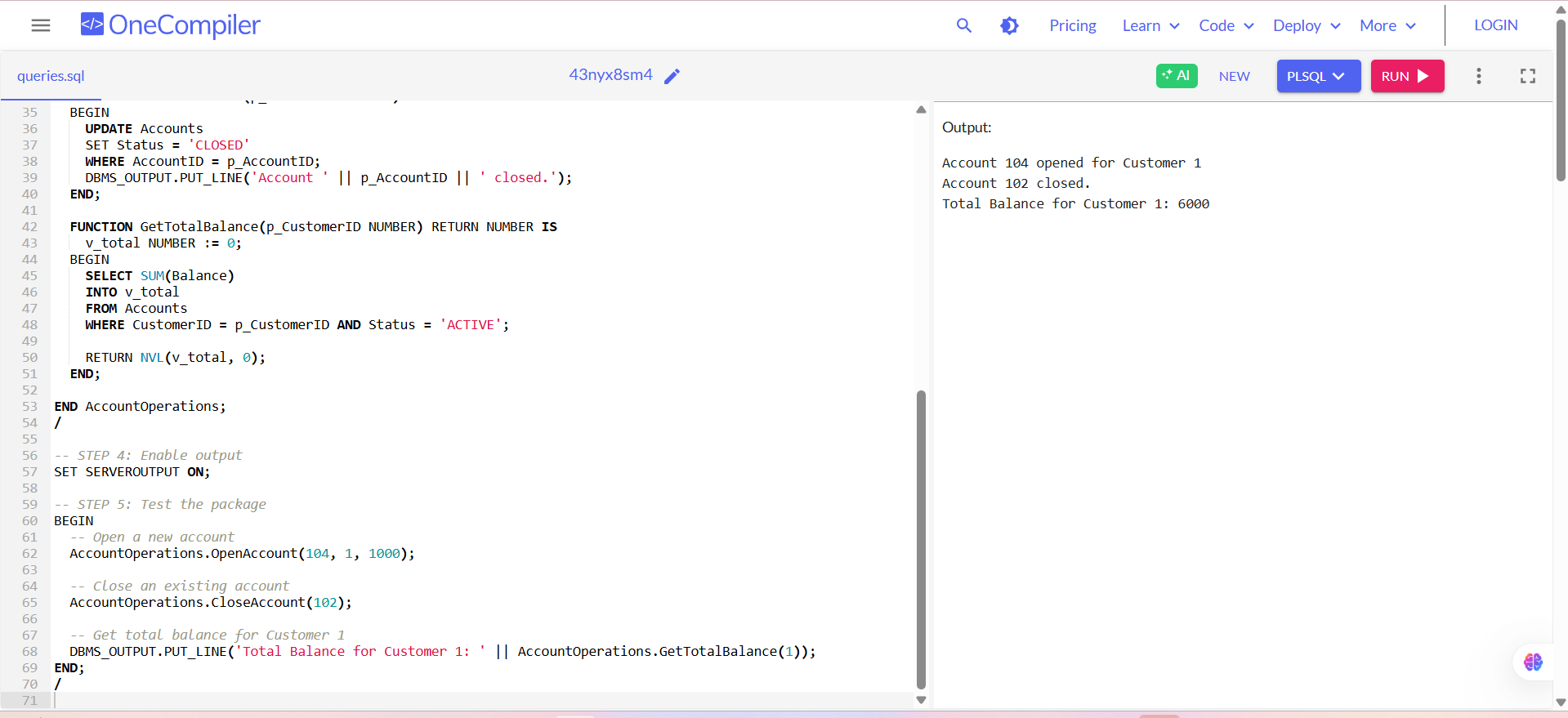
-- Get total balance for Customer 1

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

END;

/

**OUTPUT SCREENSHOT:**



**EXPLANATION:**

1. The AccountOperations package groups account-related logic: **opening**, **closing**, and **checking balances**.

2. Procedures use INSERT and UPDATE, while the function uses SUM to calculate total balance.

3. Package improves **modularity and reusability** of database operations.