**Week-2\_ Skill - PLSQL\_HandsOn**

**SUPERSETID:6393676**

**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 TABLE customers (

customer\_id NUMBER PRIMARY KEY,

age NUMBER,

loan\_interest\_rate NUMBER

);

INSERT INTO customers (customer\_id, age, loan\_interest\_rate) VALUES (1, 55, 8.5);

INSERT INTO customers (customer\_id, age, loan\_interest\_rate) VALUES (2, 62, 9.0);

INSERT INTO customers (customer\_id, age, loan\_interest\_rate) VALUES (3, 70, 8.0);

INSERT INTO customers (customer\_id, age, loan\_interest\_rate) VALUES (4, 45, 9.5);

DECLARE

v\_discount NUMBER := 0.01; -- 1% discount

BEGIN

FOR customer IN (SELECT customer\_id, age, loan\_interest\_rate FROM customers) LOOP

IF customer.age > 60 THEN

UPDATE customers

SET loan\_interest\_rate = loan\_interest\_rate - (loan\_interest\_rate \* v\_discount)

WHERE customer\_id = customer.customer\_id;

DBMS\_OUTPUT.PUT\_LINE('Discount applied to Customer ID: ' || customer.customer\_id || ', New Interest Rate: ' || (customer.loan\_interest\_rate - (customer.loan\_interest\_rate \* v\_discount)));

ELSE

DBMS\_OUTPUT.PUT\_LINE('No discount applied to Customer ID: ' || customer.customer\_id);

END IF;

END LOOP;

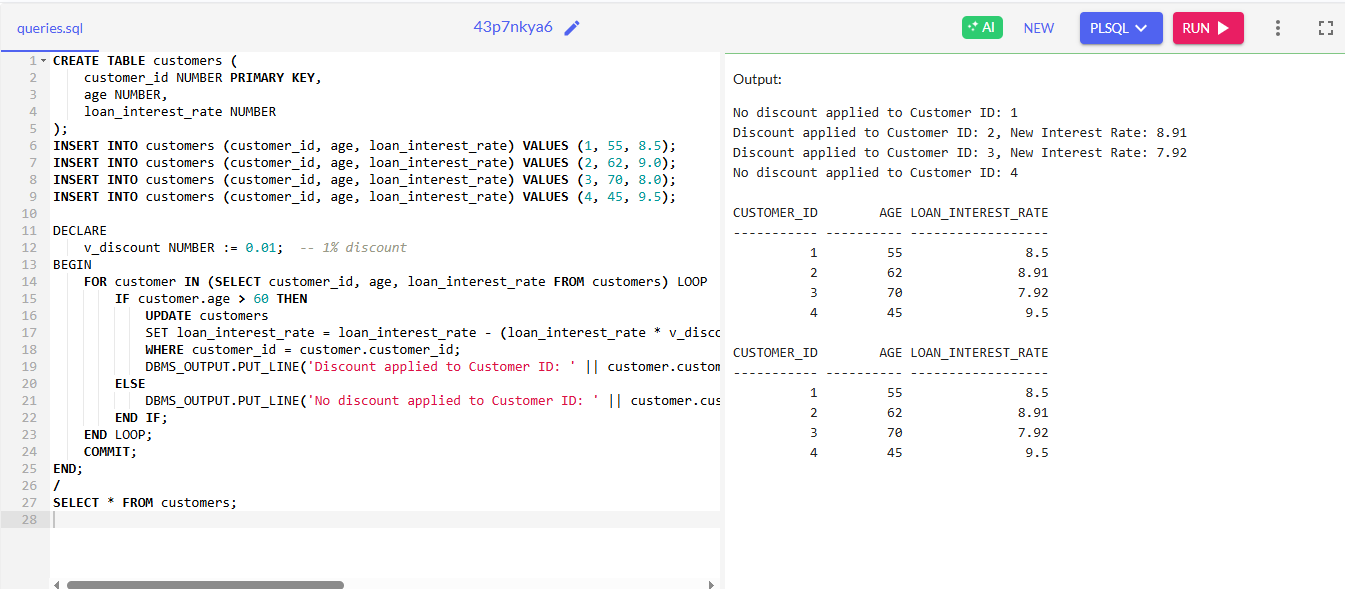
COMMIT;

END;

/

SELECT \* FROM customers;

**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:**

CREATE TABLE customers (

customer\_id NUMBER PRIMARY KEY,

balance NUMBER,

is\_vip VARCHAR2(5) DEFAULT 'FALSE'

);

INSERT INTO customers (customer\_id, balance, is\_vip) VALUES (1, 5000, 'FALSE');

INSERT INTO customers (customer\_id, balance, is\_vip) VALUES (2, 12000, 'FALSE');

INSERT INTO customers (customer\_id, balance, is\_vip) VALUES (3, 8000, 'FALSE');

INSERT INTO customers (customer\_id, balance, is\_vip) VALUES (4, 15000, 'FALSE');

DECLARE

v\_vip\_balance\_threshold NUMBER := 10000; -- Balance threshold for VIP status

BEGIN

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

IF customer.balance > v\_vip\_balance\_threshold THEN

UPDATE customers

SET is\_vip = 'TRUE'

WHERE customer\_id = customer.customer\_id;

DBMS\_OUTPUT.PUT\_LINE('Customer ID: ' || customer.customer\_id || ' has been promoted to VIP status.');

END IF;

END LOOP;

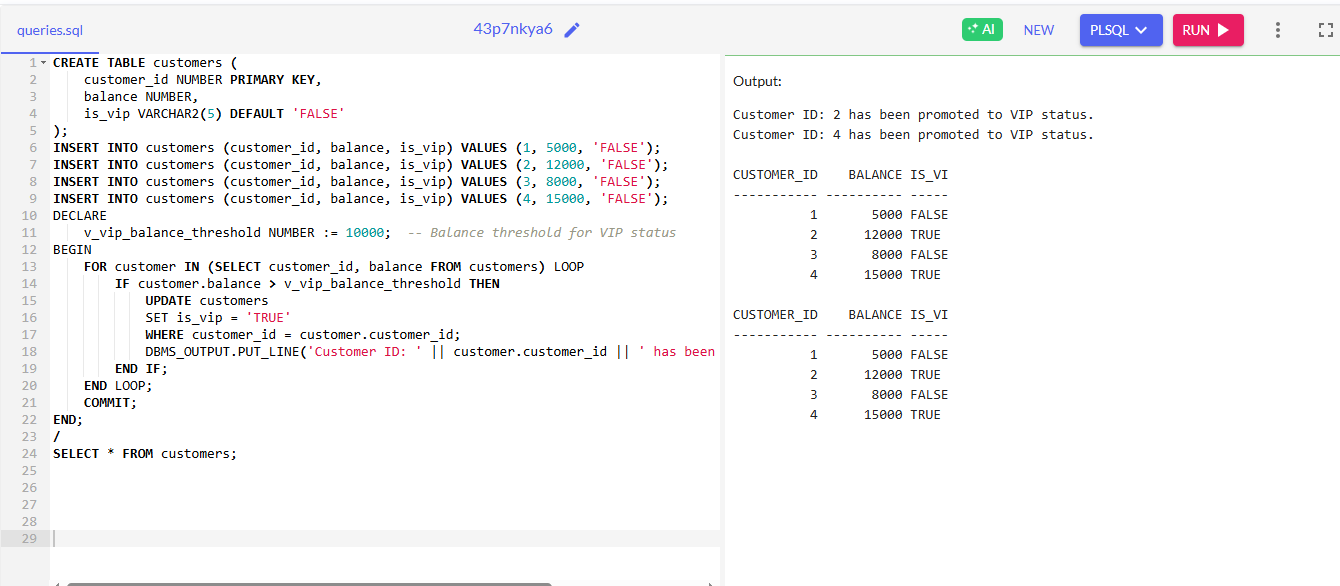
COMMIT;

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:**

CREATE TABLE loans (

loan\_id NUMBER PRIMARY KEY,

customer\_name VARCHAR2(100),

due\_date DATE,

amount NUMBER

);

INSERT INTO loans (loan\_id, customer\_name, due\_date, amount) VALUES (1, 'John Doe', SYSDATE + 20, 10000);

INSERT INTO loans (loan\_id, customer\_name, due\_date, amount) VALUES (2, 'Jane Smith', SYSDATE + 35, 20000);

INSERT INTO loans (loan\_id, customer\_name, due\_date, amount) VALUES (3, 'Bob Johnson', SYSDATE + 10, 5000);

DECLARE

v\_due\_date DATE;

v\_today DATE := SYSDATE;

v\_reminder\_days NUMBER := 30;

BEGIN

FOR loan IN (SELECT loan\_id, customer\_name, due\_date, amount

FROM loans

WHERE due\_date BETWEEN v\_today AND v\_today + v\_reminder\_days) LOOP

DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan ID ' || loan.loan\_id || ' for ' || loan.customer\_name || ' is due on ' || loan.due\_date || ' with amount ' || loan.amount);

END LOOP;

END;

/

**OUTPUT:**



**Exercise 2: Error Handling**

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

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

**SOLUTION:**

-- 1. Create table

CREATE TABLE accounts (

account\_id NUMBER PRIMARY KEY,

balance NUMBER

);

-- 2. Insert sample data

INSERT INTO accounts VALUES (1001, 5000);

INSERT INTO accounts VALUES (1002, 3000);

COMMIT;

-- 3. Create procedure

CREATE OR REPLACE PROCEDURE SafeTransferFunds(

from\_acc NUMBER,

to\_acc NUMBER,

amount NUMBER

) AS

BEGIN

UPDATE accounts SET balance = balance - amount WHERE account\_id = from\_acc;

UPDATE accounts SET balance = balance + amount WHERE account\_id = to\_acc;

COMMIT;

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

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Transfer failed: ' || SUBSTR(SQLERRM,1,70));

END;

/

-- 4. Test and display

BEGIN

-- Test transfer

DBMS\_OUTPUT.PUT\_LINE('Testing fund transfer...');

SafeTransferFunds(1001, 1002, 1000);

-- Display results

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

DBMS\_OUTPUT.PUT\_LINE('ACCOUNT\_ID BALANCE');

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

FOR r IN (SELECT account\_id, balance FROM accounts ORDER BY account\_id) LOOP

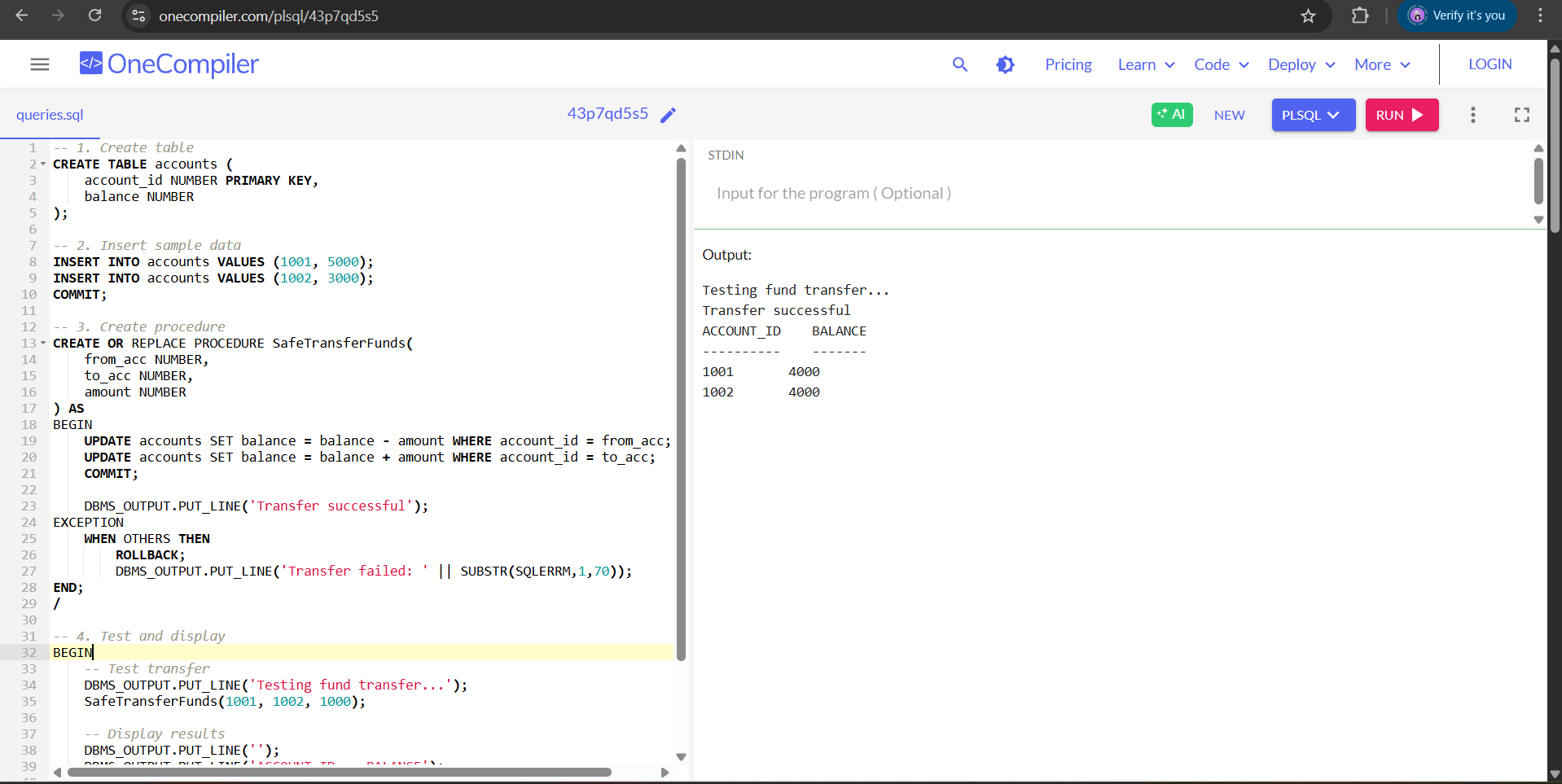
DBMS\_OUTPUT.PUT\_LINE(LPAD(r.account\_id, 10) || ' ' || LPAD(r.balance, 7));

END LOOP;

END;

/

**OUTPUT:**

****

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

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

**SOLUTION**

-- 1. Create table

CREATE TABLE employees (

emp\_id NUMBER PRIMARY KEY,

emp\_name VARCHAR2(100),

salary NUMBER

);

-- 2. Insert sample data

INSERT INTO employees VALUES (1, 'John Doe', 50000);

INSERT INTO employees VALUES (2, 'Jane Smith', 60000);

COMMIT;

-- 3. Create procedure

CREATE OR REPLACE PROCEDURE UpdateSalary(

emp\_id NUMBER,

percent NUMBER

) AS

BEGIN

UPDATE employees

SET salary = salary \* (1 + percent/100)

WHERE emp\_id = emp\_id;

COMMIT;

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

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Update failed: ' || SUBSTR(SQLERRM,1,70));

END;

/

-- 4. Test and display

BEGIN

-- Test update

DBMS\_OUTPUT.PUT\_LINE('Testing salary update...');

UpdateSalary(1, 10); -- 10% raise

-- Display results

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

DBMS\_OUTPUT.PUT\_LINE('EMP\_ID EMPLOYEE\_NAME SALARY');

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

FOR r IN (SELECT emp\_id, emp\_name, salary FROM employees ORDER BY emp\_id) LOOP

DBMS\_OUTPUT.PUT\_LINE(LPAD(r.emp\_id, 7) || ' ' ||

RPAD(r.emp\_name, 15) || ' ' ||

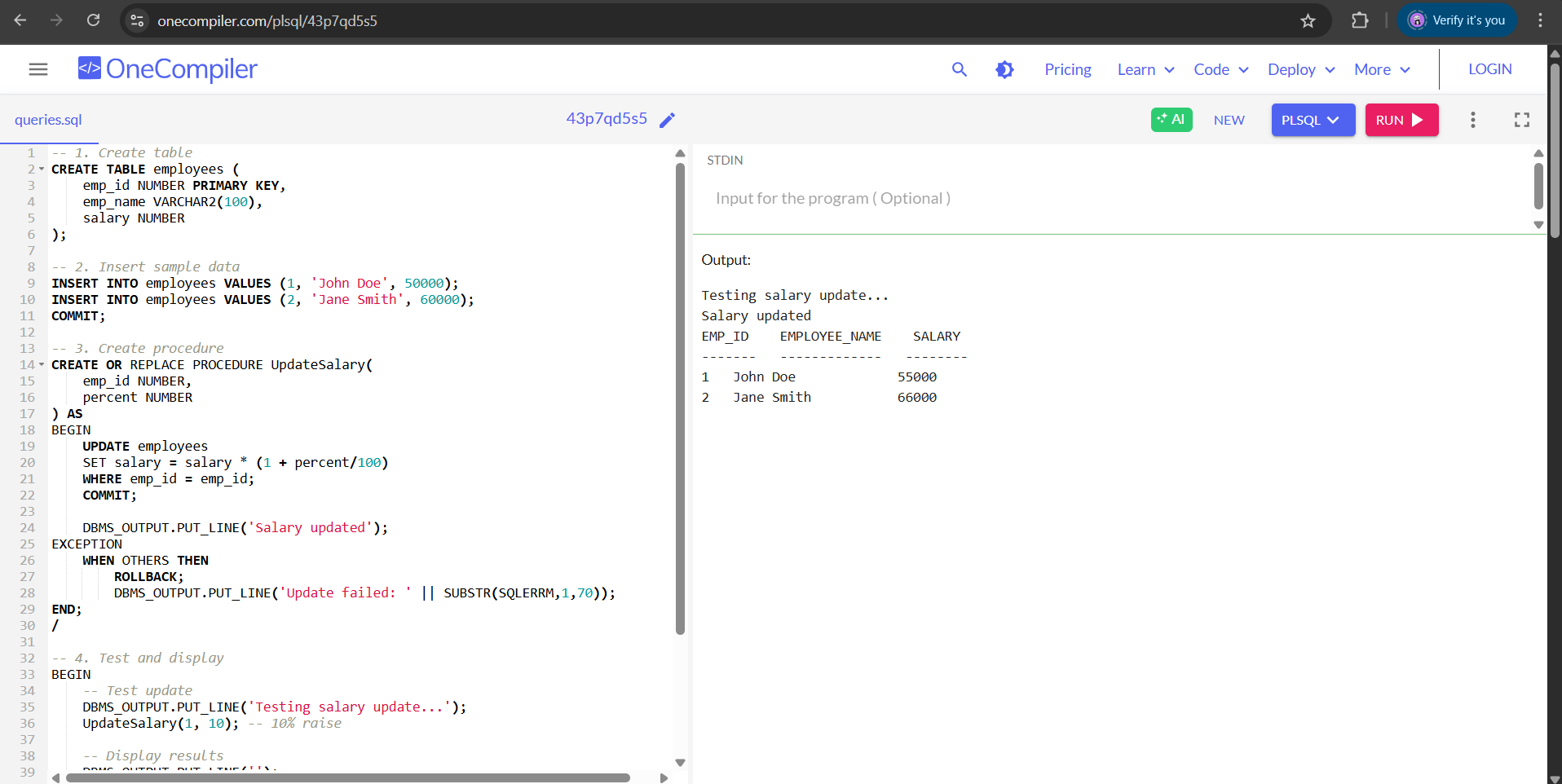
LPAD(r.salary, 8));

END LOOP;

END;

/

**OUTPUT**

****

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

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

**SOLUTION**

-- 1. Create table

CREATE TABLE customers (

cust\_id NUMBER PRIMARY KEY,

cust\_name VARCHAR2(100),

email VARCHAR2(100),

phone VARCHAR2(20)

);

-- 2. Create procedure

CREATE OR REPLACE PROCEDURE AddNewCustomer(

cust\_id NUMBER,

name VARCHAR2,

email VARCHAR2,

phone VARCHAR2

) AS

BEGIN

INSERT INTO customers VALUES (cust\_id, name, email, phone);

COMMIT;

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

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Add failed: ' || SUBSTR(SQLERRM,1,70));

END;

/

-- 3. Test and display

BEGIN

-- Test add customer

DBMS\_OUTPUT.PUT\_LINE('Testing customer add...');

AddNewCustomer(101, 'Alice Johnson', 'alice@example.com', '555-1234');

-- Display results

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

DBMS\_OUTPUT.PUT\_LINE('CUSTOMER\_ID CUSTOMER\_NAME EMAIL PHONE');

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

FOR r IN (SELECT cust\_id, cust\_name, email, phone FROM customers ORDER BY cust\_id) LOOP

DBMS\_OUTPUT.PUT\_LINE(LPAD(r.cust\_id, 11) || ' ' ||

RPAD(r.cust\_name, 15) || ' ' ||

RPAD(r.email, 18) || ' ' ||

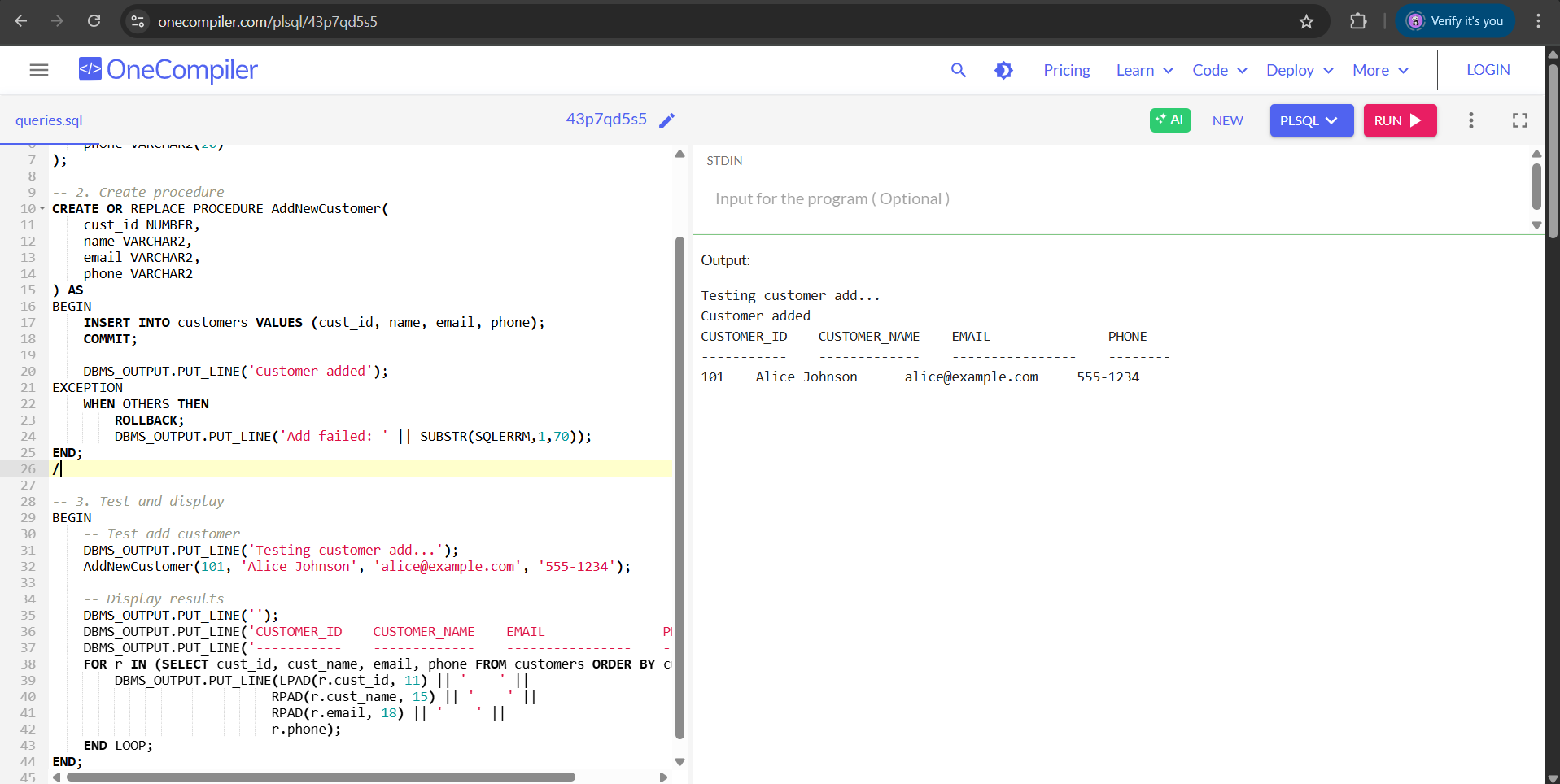
r.phone);

END LOOP;

END;

/

**OUTPUT**

****

**Exercise 3: Stored Procedures**

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

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

**SOLUTION:**

CREATE TABLE accounts (

account\_id NUMBER PRIMARY KEY,

balance NUMBER(10,2) NOT NULL,

account\_type VARCHAR2(20) DEFAULT 'SAVINGS'

);

CREATE TABLE transaction\_log (

log\_id NUMBER,

log\_date TIMESTAMP,

message VARCHAR2(500),

procedure\_name VARCHAR2(100)

);

CREATE TABLE error\_log (

error\_id NUMBER,

error\_date TIMESTAMP,

error\_message VARCHAR2(500),

procedure\_name VARCHAR2(100)

);

CREATE SEQUENCE log\_seq START WITH 1 INCREMENT BY 1;

CREATE SEQUENCE error\_seq START WITH 1 INCREMENT BY 1;

INSERT INTO accounts (account\_id, balance, account\_type) VALUES (101, 1000.00, 'SAVINGS');

INSERT INTO accounts (account\_id, balance, account\_type) VALUES (102, 500.00, 'CHECKING');

INSERT INTO accounts (account\_id, balance, account\_type) VALUES (103, 2000.00, 'SAVINGS');

INSERT INTO accounts (account\_id, balance, account\_type) VALUES (104, 5000.00, 'SAVINGS');

COMMIT;

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest AS

v\_interest\_rate NUMBER := 0.01; -- 1% monthly interest

v\_accounts\_updated NUMBER := 0;

v\_total\_interest NUMBER := 0;

v\_error\_message VARCHAR2(4000);

BEGIN

DBMS\_OUTPUT.PUT\_LINE('Starting monthly interest processing...');

FOR acc\_rec IN (SELECT account\_id, balance FROM accounts WHERE account\_type = 'SAVINGS' FOR UPDATE)

LOOP

DECLARE

v\_interest\_amount NUMBER;

BEGIN

v\_interest\_amount := acc\_rec.balance \* v\_interest\_rate;

UPDATE accounts

SET balance = balance + v\_interest\_amount

WHERE account\_id = acc\_rec.account\_id;

v\_accounts\_updated := v\_accounts\_updated + 1;

v\_total\_interest := v\_total\_interest + v\_interest\_amount;

INSERT INTO transaction\_log VALUES (

log\_seq.NEXTVAL,

SYSTIMESTAMP,

'Applied interest of ' || ROUND(v\_interest\_amount, 2) ||

' to account ' || acc\_rec.account\_id,

'ProcessMonthlyInterest'

);

EXCEPTION

WHEN OTHERS THEN

v\_error\_message := 'Error processing account ' || acc\_rec.account\_id || ': ' || SQLERRM;

INSERT INTO error\_log VALUES (

error\_seq.NEXTVAL,

SYSTIMESTAMP,

v\_error\_message,

'ProcessMonthlyInterest'

);

DBMS\_OUTPUT.PUT\_LINE(v\_error\_message);

END;

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('Monthly interest processing completed.');

DBMS\_OUTPUT.PUT\_LINE('Accounts updated: ' || v\_accounts\_updated);

DBMS\_OUTPUT.PUT\_LINE('Total interest applied: ' || ROUND(v\_total\_interest, 2));

INSERT INTO transaction\_log VALUES (

log\_seq.NEXTVAL,

SYSTIMESTAMP,

'Monthly interest processed. ' || v\_accounts\_updated ||

' accounts updated. Total interest: ' || ROUND(v\_total\_interest, 2),

'ProcessMonthlyInterest'

);

EXCEPTION

WHEN OTHERS THEN

v\_error\_message := 'Error in ProcessMonthlyInterest: ' || SQLERRM;

INSERT INTO error\_log VALUES (

error\_seq.NEXTVAL,

SYSTIMESTAMP,

v\_error\_message,

'ProcessMonthlyInterest'

);

DBMS\_OUTPUT.PUT\_LINE(v\_error\_message);

RAISE;

END ProcessMonthlyInterest;

/

BEGIN

ProcessMonthlyInterest();

DBMS\_OUTPUT.PUT\_LINE('Updated account balances:');

FOR acc IN (SELECT account\_id, balance FROM accounts WHERE account\_type = 'SAVINGS' ORDER BY account\_id)

LOOP

DBMS\_OUTPUT.PUT\_LINE('Account ' || acc.account\_id || ': ' || ROUND(acc.balance, 2));

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('Transaction Log:');

FOR tlog IN (SELECT \* FROM transaction\_log WHERE procedure\_name = 'ProcessMonthlyInterest' ORDER BY log\_date)

LOOP

DBMS\_OUTPUT.PUT\_LINE(tlog.log\_date || ' - ' || tlog.message);

END LOOP;

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

FOR elog IN (SELECT \* FROM error\_log WHERE procedure\_name = 'ProcessMonthlyInterest' ORDER BY error\_date)

LOOP

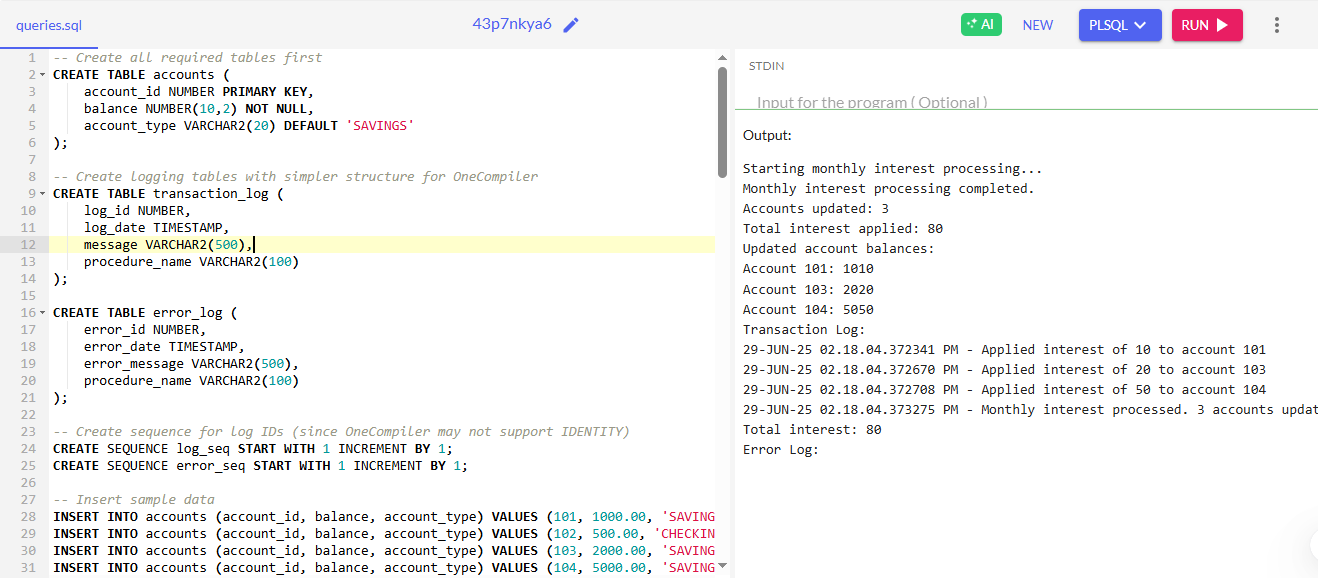
DBMS\_OUTPUT.PUT\_LINE(elog.error\_date || ' - ' || elog.error\_message);

END LOOP;

END;

/

**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 employees table if it doesn't exist

CREATE TABLE employees (

employee\_id NUMBER PRIMARY KEY,

employee\_name VARCHAR2(100) NOT NULL,

department\_id NUMBER,

salary NUMBER(10,2),

hire\_date DATE

);

-- Create departments table if it doesn't exist

CREATE TABLE departments (

department\_id NUMBER PRIMARY KEY,

department\_name VARCHAR2(100) NOT NULL

);

-- Create logging tables

CREATE TABLE transaction\_log (

log\_id NUMBER,

log\_date TIMESTAMP,

message VARCHAR2(500),

procedure\_name VARCHAR2(100)

);

CREATE TABLE error\_log (

error\_id NUMBER,

error\_date TIMESTAMP,

error\_message VARCHAR2(500),

procedure\_name VARCHAR2(100)

);

-- Create sequences for log IDs

CREATE SEQUENCE log\_seq START WITH 1 INCREMENT BY 1;

CREATE SEQUENCE error\_seq START WITH 1 INCREMENT BY 1;

-- Insert sample data

INSERT INTO departments VALUES (10, 'Accounting');

INSERT INTO departments VALUES (20, 'IT');

INSERT INTO departments VALUES (30, 'Marketing');

INSERT INTO employees VALUES (1, 'John Smith', 10, 50000, TO\_DATE('2020-01-15', 'YYYY-MM-DD'));

INSERT INTO employees VALUES (2, 'Sarah Johnson', 20, 65000, TO\_DATE('2019-05-22', 'YYYY-MM-DD'));

INSERT INTO employees VALUES (3, 'Michael Brown', 10, 55000, TO\_DATE('2021-03-10', 'YYYY-MM-DD'));

INSERT INTO employees VALUES (4, 'Emily Davis', 30, 60000, TO\_DATE('2018-11-05', 'YYYY-MM-DD'));

COMMIT;

-- Create the bonus update procedure

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(

p\_department\_id IN NUMBER,

p\_bonus\_percentage IN NUMBER

) AS

v\_employees\_updated NUMBER := 0;

v\_total\_bonus NUMBER := 0;

v\_error\_message VARCHAR2(4000);

v\_department\_name VARCHAR2(100);

BEGIN

-- Verify department exists

BEGIN

SELECT department\_name INTO v\_department\_name

FROM departments

WHERE department\_id = p\_department\_id;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

v\_error\_message := 'Department ID ' || p\_department\_id || ' does not exist';

INSERT INTO error\_log VALUES (

error\_seq.NEXTVAL,

SYSTIMESTAMP,

v\_error\_message,

'UpdateEmployeeBonus'

);

RAISE\_APPLICATION\_ERROR(-20001, v\_error\_message);

END;

-- Log procedure start

DBMS\_OUTPUT.PUT\_LINE('Starting bonus update for department ' || p\_department\_id ||

' (' || v\_department\_name || ') at ' || p\_bonus\_percentage || '%');

-- Update salaries with bonus

FOR emp\_rec IN (SELECT employee\_id, salary FROM employees

WHERE department\_id = p\_department\_id FOR UPDATE)

LOOP

DECLARE

v\_bonus\_amount NUMBER;

v\_new\_salary NUMBER;

BEGIN

-- Calculate bonus

v\_bonus\_amount := emp\_rec.salary \* (p\_bonus\_percentage/100);

v\_new\_salary := emp\_rec.salary + v\_bonus\_amount;

-- Update employee salary

UPDATE employees

SET salary = v\_new\_salary

WHERE employee\_id = emp\_rec.employee\_id;

-- Track totals

v\_employees\_updated := v\_employees\_updated + 1;

v\_total\_bonus := v\_total\_bonus + v\_bonus\_amount;

-- Log individual update

INSERT INTO transaction\_log VALUES (

log\_seq.NEXTVAL,

SYSTIMESTAMP,

'Applied ' || p\_bonus\_percentage || '% bonus (' || ROUND(v\_bonus\_amount, 2) ||

') to employee ' || emp\_rec.employee\_id || '. New salary: ' || ROUND(v\_new\_salary, 2),

'UpdateEmployeeBonus'

);

EXCEPTION

WHEN OTHERS THEN

v\_error\_message := 'Error updating employee ' || emp\_rec.employee\_id || ': ' || SQLERRM;

INSERT INTO error\_log VALUES (

error\_seq.NEXTVAL,

SYSTIMESTAMP,

v\_error\_message,

'UpdateEmployeeBonus'

);

DBMS\_OUTPUT.PUT\_LINE(v\_error\_message);

END;

END LOOP;

-- Log summary

DBMS\_OUTPUT.PUT\_LINE('Bonus update completed for department ' || p\_department\_id);

DBMS\_OUTPUT.PUT\_LINE('Employees updated: ' || v\_employees\_updated);

DBMS\_OUTPUT.PUT\_LINE('Total bonus awarded: ' || ROUND(v\_total\_bonus, 2));

-- Insert summary log

INSERT INTO transaction\_log VALUES (

log\_seq.NEXTVAL,

SYSTIMESTAMP,

'Bonus update completed for department ' || p\_department\_id ||

'. ' || v\_employees\_updated || ' employees updated. Total bonus: ' || ROUND(v\_total\_bonus, 2),

'UpdateEmployeeBonus'

);

EXCEPTION

WHEN OTHERS THEN

v\_error\_message := 'Error in UpdateEmployeeBonus: ' || SQLERRM;

INSERT INTO error\_log VALUES (

error\_seq.NEXTVAL,

SYSTIMESTAMP,

v\_error\_message,

'UpdateEmployeeBonus'

);

DBMS\_OUTPUT.PUT\_LINE(v\_error\_message);

RAISE;

END UpdateEmployeeBonus;

/

BEGIN

DBMS\_OUTPUT.PUT\_LINE('TEST 1: Valid department (Accounting - 10)');

UpdateEmployeeBonus(10, 5); -- 5% bonus for Accounting

BEGIN

DBMS\_OUTPUT.PUT\_LINE('TEST 2: Invalid department (99)');

UpdateEmployeeBonus(99, 5);

EXCEPTION

WHEN OTHERS THEN

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

END;

DBMS\_OUTPUT.PUT\_LINE('Updated employee salaries:');

FOR emp IN (SELECT e.employee\_id, e.employee\_name, d.department\_name, e.salary

FROM employees e JOIN departments d ON e.department\_id = d.department\_id

ORDER BY e.department\_id, e.employee\_id)

LOOP

DBMS\_OUTPUT.PUT\_LINE(emp.employee\_id || ' - ' || emp.employee\_name || ' (' ||

emp.department\_name || '): ' || ROUND(emp.salary, 2));

END LOOP;

-- View logs

DBMS\_OUTPUT.PUT\_LINE('Transaction Log:');

FOR tlog IN (SELECT \* FROM transaction\_log WHERE procedure\_name = 'UpdateEmployeeBonus' ORDER BY log\_date)

LOOP

DBMS\_OUTPUT.PUT\_LINE(TO\_CHAR(tlog.log\_date, 'YYYY-MM-DD HH24:MI:SS') || ' - ' || tlog.message);

END LOOP;

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

FOR elog IN (SELECT \* FROM error\_log WHERE procedure\_name = 'UpdateEmployeeBonus' ORDER BY error\_date)

LOOP

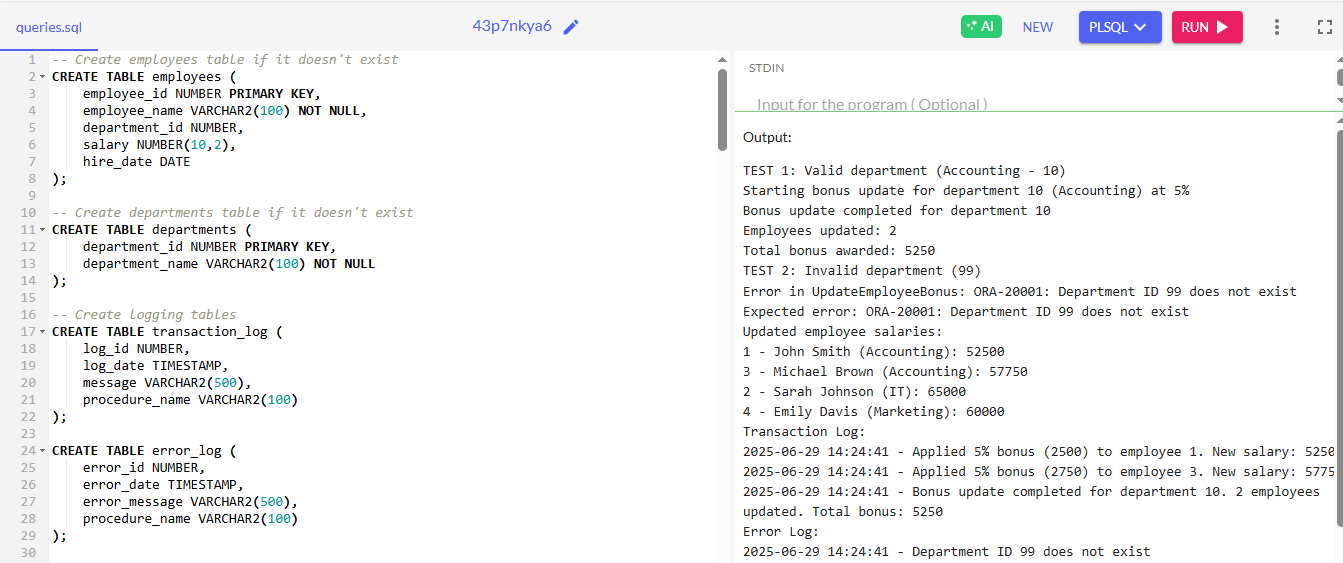
DBMS\_OUTPUT.PUT\_LINE(TO\_CHAR(elog.error\_date, 'YYYY-MM-DD HH24:MI:SS') || ' - ' || elog.error\_message);

END LOOP;

END;

/

**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**:

-- Simplified Fund Transfer Procedure with Table Output

CREATE TABLE accounts (

account\_id NUMBER PRIMARY KEY,

balance NUMBER(10,2) NOT NULL

);

CREATE TABLE transfer\_log (

log\_id NUMBER,

from\_acc NUMBER,

to\_acc NUMBER,

amount NUMBER(10,2),

status VARCHAR2(20),

log\_time TIMESTAMP

);

CREATE SEQUENCE log\_seq START WITH 1 INCREMENT BY 1;

INSERT INTO accounts VALUES (101, 5000);

INSERT INTO accounts VALUES (102, 3000);

CREATE OR REPLACE PROCEDURE transfer\_funds(

p\_from IN NUMBER,

p\_to IN NUMBER,

p\_amount IN NUMBER

) AS

v\_balance NUMBER;

BEGIN

-- Check source account balance

SELECT balance INTO v\_balance

FROM accounts

WHERE account\_id = p\_from

FOR UPDATE;

IF v\_balance < p\_amount THEN

INSERT INTO transfer\_log VALUES (log\_seq.NEXTVAL, p\_from, p\_to, p\_amount, 'FAILED', SYSTIMESTAMP);

RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient funds');

END IF;

-- Check destination account exists

BEGIN

SELECT 1 INTO v\_balance FROM accounts WHERE account\_id = p\_to;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

INSERT INTO transfer\_log VALUES (log\_seq.NEXTVAL, p\_from, p\_to, p\_amount, 'FAILED', SYSTIMESTAMP);

RAISE\_APPLICATION\_ERROR(-20002, 'Destination account not found');

END;

-- Perform transfer

UPDATE accounts SET balance = balance - p\_amount WHERE account\_id = p\_from;

UPDATE accounts SET balance = balance + p\_amount WHERE account\_id = p\_to;

-- Log success

INSERT INTO transfer\_log VALUES (log\_seq.NEXTVAL, p\_from, p\_to, p\_amount, 'SUCCESS', SYSTIMESTAMP);

COMMIT;

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

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END;

/

-- Test with improved output formatting

BEGIN

-- Display initial balances

DBMS\_OUTPUT.PUT\_LINE('Initial Balances:');

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

DBMS\_OUTPUT.PUT\_LINE('ACCOUNT\_ID BALANCE');

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

FOR acc IN (SELECT \* FROM accounts ORDER BY account\_id) LOOP

DBMS\_OUTPUT.PUT\_LINE(RPAD(acc.account\_id, 10) || ' ' || TO\_CHAR(acc.balance, '99999.99'));

END LOOP;

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

-- Run tests

DBMS\_OUTPUT.PUT\_LINE('Test 1: Valid transfer (101 → 102, 1000)');

transfer\_funds(101, 102, 1000);

DBMS\_OUTPUT.PUT\_LINE('Test 2: Insufficient funds (101 → 102, 10000)');

BEGIN

transfer\_funds(101, 102, 10000);

EXCEPTION

WHEN OTHERS THEN NULL;

END;

DBMS\_OUTPUT.PUT\_LINE('Test 3: Invalid account (101 → 999, 100)');

BEGIN

transfer\_funds(101, 999, 100);

EXCEPTION

WHEN OTHERS THEN NULL;

END;

-- Display final balances

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

DBMS\_OUTPUT.PUT\_LINE('Final Balances:');

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

DBMS\_OUTPUT.PUT\_LINE('ACCOUNT\_ID BALANCE');

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

FOR acc IN (SELECT \* FROM accounts ORDER BY account\_id) LOOP

DBMS\_OUTPUT.PUT\_LINE(RPAD(acc.account\_id, 10) || ' ' || TO\_CHAR(acc.balance, '99999.99'));

END LOOP;

-- Display transaction log

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

DBMS\_OUTPUT.PUT\_LINE('Transaction Log:');

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

DBMS\_OUTPUT.PUT\_LINE('LOG\_ID FROM\_ACC TO\_ACC AMOUNT STATUS LOG\_TIME');

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

FOR tlog IN (SELECT \* FROM transfer\_log ORDER BY log\_id) LOOP

DBMS\_OUTPUT.PUT\_LINE(

RPAD(tlog.log\_id, 7) || ' ' ||

RPAD(tlog.from\_acc, 8) || ' ' ||

RPAD(tlog.to\_acc, 7) || ' ' ||

RPAD(TO\_CHAR(tlog.amount, '9999.99'), 8) || ' ' ||

RPAD(tlog.status, 7) || ' ' ||

TO\_CHAR(tlog.log\_time, 'DD-MON-YY HH24:MI:SS')

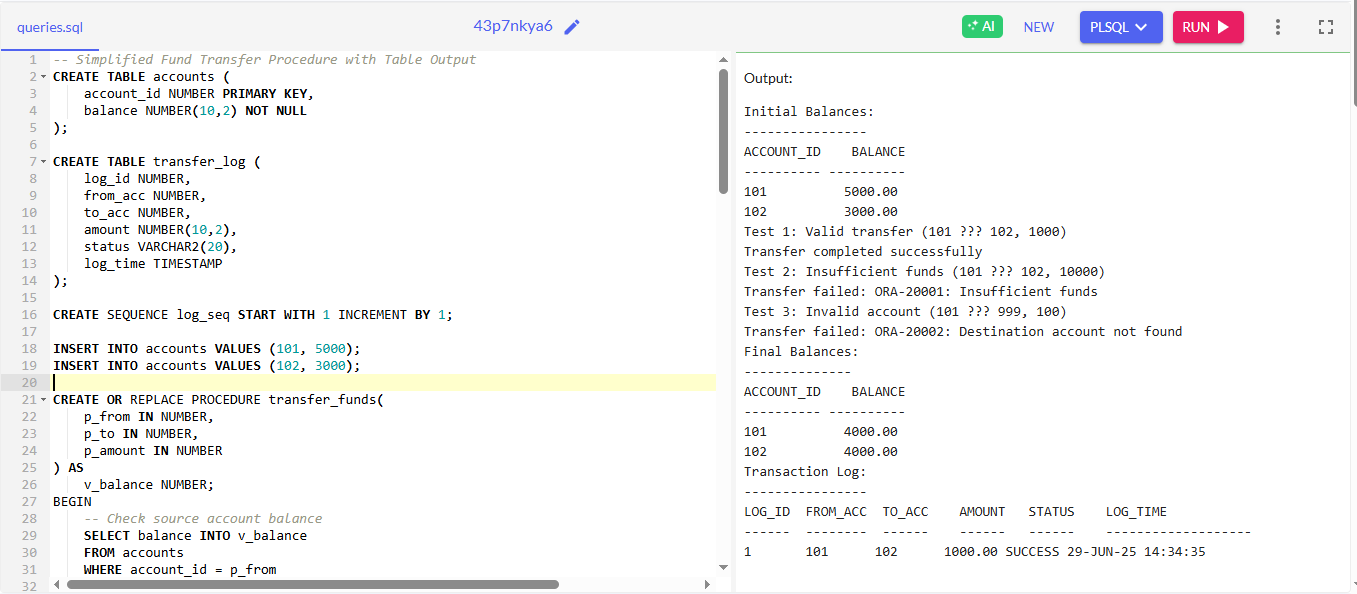
);

END LOOP;

END;

/

**OUTPUT:**



**Exercise 4: Functions**

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

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

**SOLUTION:**

CREATE OR REPLACE FUNCTION CalculateAge(p\_dob IN DATE)

RETURN NUMBER IS

v\_age NUMBER;

BEGIN

-- Calculate age precisely (considers month/day of birth)

v\_age := EXTRACT(YEAR FROM SYSDATE) - EXTRACT(YEAR FROM p\_dob);

-- Adjust if birthday hasn't occurred yet this year

IF TO\_CHAR(SYSDATE, 'MMDD') < TO\_CHAR(p\_dob, 'MMDD') THEN

v\_age := v\_age - 1;

END IF;

RETURN v\_age;

END CalculateAge;

/

-- Test cases

DECLARE

v\_dob1 DATE := TO\_DATE('1990-05-15', 'YYYY-MM-DD'); -- Birthday passed

v\_dob2 DATE := TO\_DATE('1995-12-25', 'YYYY-MM-DD'); -- Birthday coming

v\_dob3 DATE := TO\_DATE('2000-01-01', 'YYYY-MM-DD'); -- Born on Jan 1

BEGIN

DBMS\_OUTPUT.PUT\_LINE('Customer 1 (May 15): ' || CalculateAge(v\_dob1) || ' years');

DBMS\_OUTPUT.PUT\_LINE('Customer 2 (Dec 25): ' || CalculateAge(v\_dob2) || ' years');

DBMS\_OUTPUT.PUT\_LINE('Customer 3 (Jan 1): ' || CalculateAge(v\_dob3) || ' years');

END;

/

**OUTPUT:**



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

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

**SOLUTION:**

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(

p\_loan\_amount IN NUMBER,

p\_annual\_interest\_rate IN NUMBER,

p\_loan\_years IN NUMBER

) RETURN NUMBER IS

v\_monthly\_interest\_rate NUMBER;

v\_number\_of\_payments NUMBER;

v\_monthly\_payment NUMBER;

BEGIN

-- Convert annual interest rate to monthly and percentage to decimal

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

-- Calculate total number of monthly payments

v\_number\_of\_payments := p\_loan\_years \* 12;

-- Calculate monthly payment using standard loan formula

v\_monthly\_payment := p\_loan\_amount \*

(v\_monthly\_interest\_rate \* POWER(1 + v\_monthly\_interest\_rate, v\_number\_of\_payments)) /

(POWER(1 + v\_monthly\_interest\_rate, v\_number\_of\_payments) - 1);

-- Round to 2 decimal places for currency

RETURN ROUND(v\_monthly\_payment, 2);

END CalculateMonthlyInstallment;

/

-- Test the function

DECLARE

v\_loan\_amount NUMBER := 100000; -- $100,000 loan

v\_interest\_rate NUMBER := 5.5; -- 5.5% annual interest

v\_loan\_years NUMBER := 30; -- 30 year term

v\_monthly\_payment NUMBER;

BEGIN

v\_monthly\_payment := CalculateMonthlyInstallment(v\_loan\_amount, v\_interest\_rate, v\_loan\_years);

DBMS\_OUTPUT.PUT\_LINE('Loan Amount: $' || v\_loan\_amount);

DBMS\_OUTPUT.PUT\_LINE('Interest Rate: ' || v\_interest\_rate || '%');

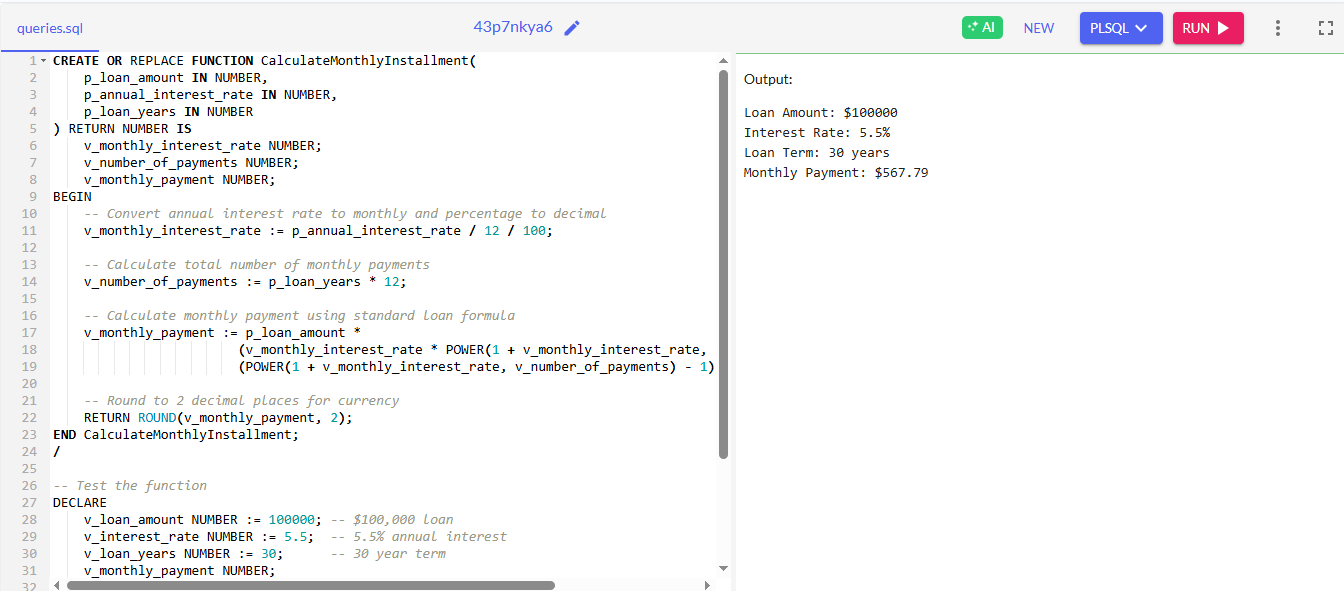
DBMS\_OUTPUT.PUT\_LINE('Loan Term: ' || v\_loan\_years || ' years');

DBMS\_OUTPUT.PUT\_LINE('Monthly Payment: $' || v\_monthly\_payment);

END;

/

**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:**

-- First create the accounts table if it doesn't exist

CREATE TABLE accounts (

account\_id NUMBER PRIMARY KEY,

balance NUMBER(10,2) NOT NULL

);

-- Insert sample data

INSERT INTO accounts VALUES (101, 5000.00);

INSERT INTO accounts VALUES (102, 3000.00);

COMMIT;

-- Create the function (corrected version)

CREATE OR REPLACE FUNCTION HasSufficientBalance(

p\_account\_id IN NUMBER,

p\_amount IN NUMBER

) RETURN VARCHAR2 IS -- Changed to VARCHAR2 for OneCompiler compatibility

v\_current\_balance NUMBER;

BEGIN

-- Get the current balance for the account

SELECT balance INTO v\_current\_balance

FROM accounts

WHERE account\_id = p\_account\_id;

-- Return 'TRUE' or 'FALSE' as strings

IF v\_current\_balance >= p\_amount THEN

RETURN 'TRUE';

ELSE

RETURN 'FALSE';

END IF;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

-- Account doesn't exist

RETURN 'FALSE';

WHEN OTHERS THEN

-- Handle any other errors

RETURN 'FALSE';

END HasSufficientBalance;

/

-- Test the function (adapted for OneCompiler)

DECLARE

v\_account\_id NUMBER := 101;

v\_amount1 NUMBER := 1000;

v\_amount2 NUMBER := 10000;

v\_result VARCHAR2(10);

BEGIN

-- Test case 1: Check for sufficient balance

v\_result := HasSufficientBalance(v\_account\_id, v\_amount1);

DBMS\_OUTPUT.PUT\_LINE('Account ' || v\_account\_id || ' has sufficient balance for $' || v\_amount1 || ': ' || v\_result);

-- Test case 2: Check for larger amount

v\_result := HasSufficientBalance(v\_account\_id, v\_amount2);

DBMS\_OUTPUT.PUT\_LINE('Account ' || v\_account\_id || ' has sufficient balance for $' || v\_amount2 || ': ' || v\_result);

-- Test case 3: Check non-existent account

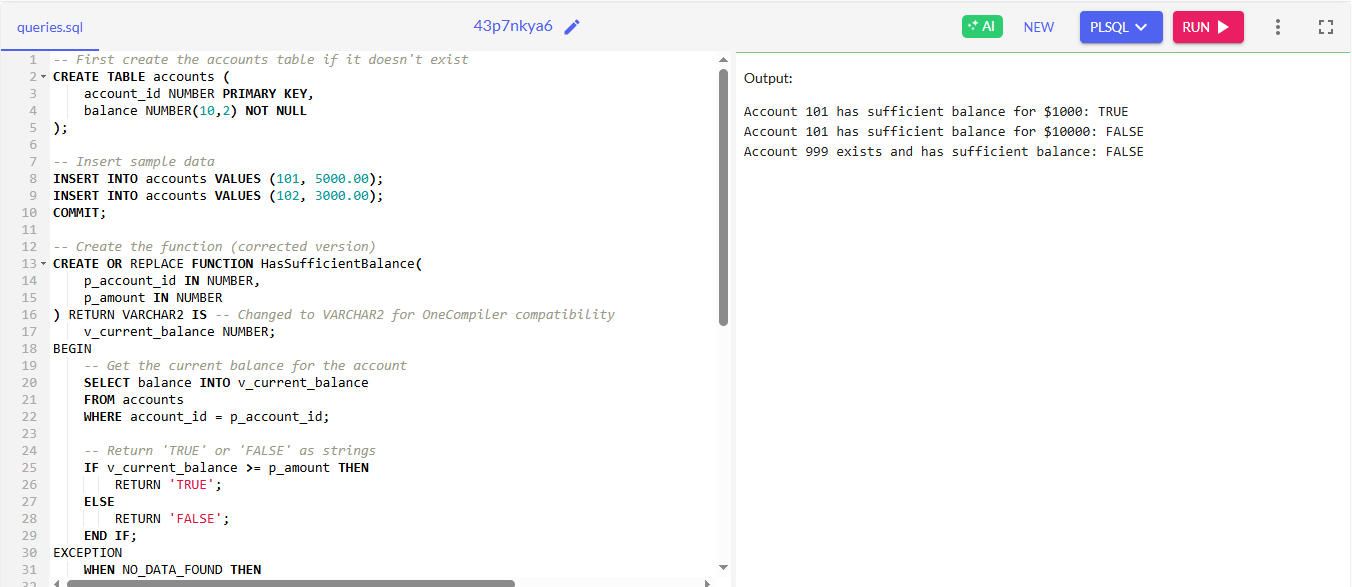
v\_result := HasSufficientBalance(999, v\_amount1);

DBMS\_OUTPUT.PUT\_LINE('Account 999 exists and has sufficient balance: ' || v\_result);

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,

CustomerName VARCHAR2(100) NOT NULL,

Email VARCHAR2(100),

Phone VARCHAR2(20),

CreatedDate DATE DEFAULT SYSDATE,

LastModified DATE

);

-- Create the trigger

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON Customers

FOR EACH ROW

BEGIN

:NEW.LastModified := SYSDATE;

-- Optional: Display update notification

DBMS\_OUTPUT.PUT\_LINE('Customer ' || :OLD.CustomerID || ' record updated. LastModified set to ' || TO\_CHAR(SYSDATE, 'DD-MON-YYYY HH24:MI:SS'));

END;

/

-- Test the trigger

BEGIN

-- Insert test data

INSERT INTO Customers (CustomerID, CustomerName, Email, Phone)

VALUES (1, 'John Smith', 'john@example.com', '555-1234');

INSERT INTO Customers (CustomerID, CustomerName, Email, Phone)

VALUES (2, 'Sarah Johnson', 'sarah@example.com', '555-5678');

COMMIT;

-- Display initial data

DBMS\_OUTPUT.PUT\_LINE('Before update:');

FOR c IN (SELECT \* FROM Customers ORDER BY CustomerID) LOOP

DBMS\_OUTPUT.PUT\_LINE(c.CustomerID || ' | ' || c.CustomerName || ' | LastModified: ' ||

NVL(TO\_CHAR(c.LastModified, 'DD-MON-YYYY HH24:MI:SS'), 'NULL'));

END LOOP;

-- Update a customer record (trigger will fire)

UPDATE Customers

SET Email = 'john.new@example.com'

WHERE CustomerID = 1;

COMMIT;

-- Display updated data

DBMS\_OUTPUT.PUT\_LINE('After update:');

FOR c IN (SELECT \* FROM Customers ORDER BY CustomerID) LOOP

DBMS\_OUTPUT.PUT\_LINE(c.CustomerID || ' | ' || c.CustomerName || ' | LastModified: ' ||

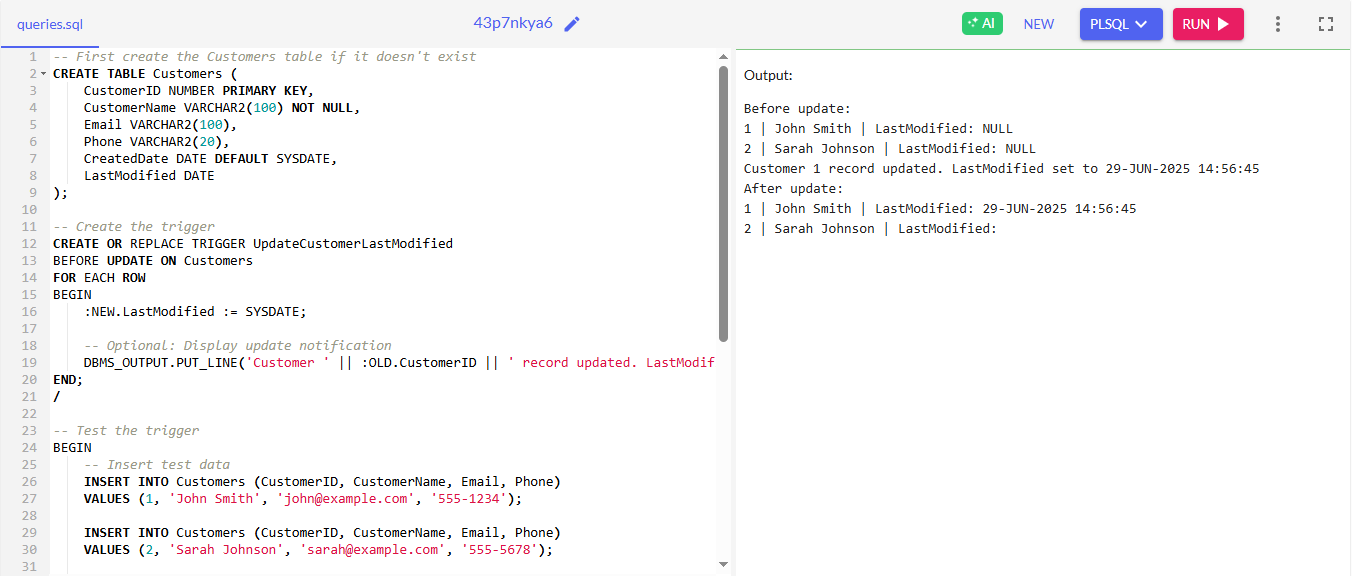
TO\_CHAR(c.LastModified, 'DD-MON-YYYY HH24:MI:SS'));

END LOOP;

END;

/

**OUTPUT:**



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

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

**SOLUTION:**

-- Create Transactions table if it doesn't exist

CREATE TABLE Transactions (

TransactionID NUMBER PRIMARY KEY,

AccountID NUMBER NOT NULL,

TransactionDate DATE DEFAULT SYSDATE,

Amount NUMBER(10,2) NOT NULL,

TransactionType VARCHAR2(20) CHECK (TransactionType IN ('DEPOSIT', 'WITHDRAWAL', 'TRANSFER'))

);

-- Create AuditLog table

CREATE TABLE AuditLog (

LogID NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,

TransactionID NUMBER,

AccountID NUMBER,

LogDate TIMESTAMP DEFAULT SYSTIMESTAMP,

Action VARCHAR2(20),

Amount NUMBER(10,2),

OldValue VARCHAR2(100),

NewValue VARCHAR2(100)

);

-- Create sequence for TransactionID (if IDENTITY not supported)

CREATE SEQUENCE trans\_seq START WITH 1 INCREMENT BY 1;

-- Create the trigger

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

-- Insert audit record for the new transaction

INSERT INTO AuditLog (

TransactionID,

AccountID,

Action,

Amount,

NewValue

) VALUES (

:NEW.TransactionID,

:NEW.AccountID,

'INSERT',

:NEW.Amount,

'Transaction Type: ' || :NEW.TransactionType ||

', Date: ' || TO\_CHAR(:NEW.TransactionDate, 'DD-MON-YYYY HH24:MI:SS')

);

DBMS\_OUTPUT.PUT\_LINE('Logged transaction #' || :NEW.TransactionID ||

' for account #' || :NEW.AccountID);

END;

/

-- Test the trigger

BEGIN

-- Insert test transactions

INSERT INTO Transactions VALUES (trans\_seq.NEXTVAL, 101, SYSDATE, 500.00, 'DEPOSIT');

INSERT INTO Transactions VALUES (trans\_seq.NEXTVAL, 102, SYSDATE, 200.00, 'WITHDRAWAL');

INSERT INTO Transactions VALUES (trans\_seq.NEXTVAL, 101, SYSDATE, 100.00, 'TRANSFER');

COMMIT;

-- Display audit log

DBMS\_OUTPUT.PUT\_LINE('Audit Log Contents:');

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

DBMS\_OUTPUT.PUT\_LINE('LOG ID | TRANS ID | ACCOUNT | ACTION | AMOUNT | DETAILS');

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

FOR log\_rec IN (SELECT \* FROM AuditLog ORDER BY LogDate) LOOP

DBMS\_OUTPUT.PUT\_LINE(

RPAD(log\_rec.LogID, 6) || ' | ' ||

RPAD(log\_rec.TransactionID, 8) || ' | ' ||

RPAD(log\_rec.AccountID, 7) || ' | ' ||

RPAD(log\_rec.Action, 7) || ' | ' ||

RPAD(log\_rec.Amount, 7) || ' | ' ||

log\_rec.NewValue

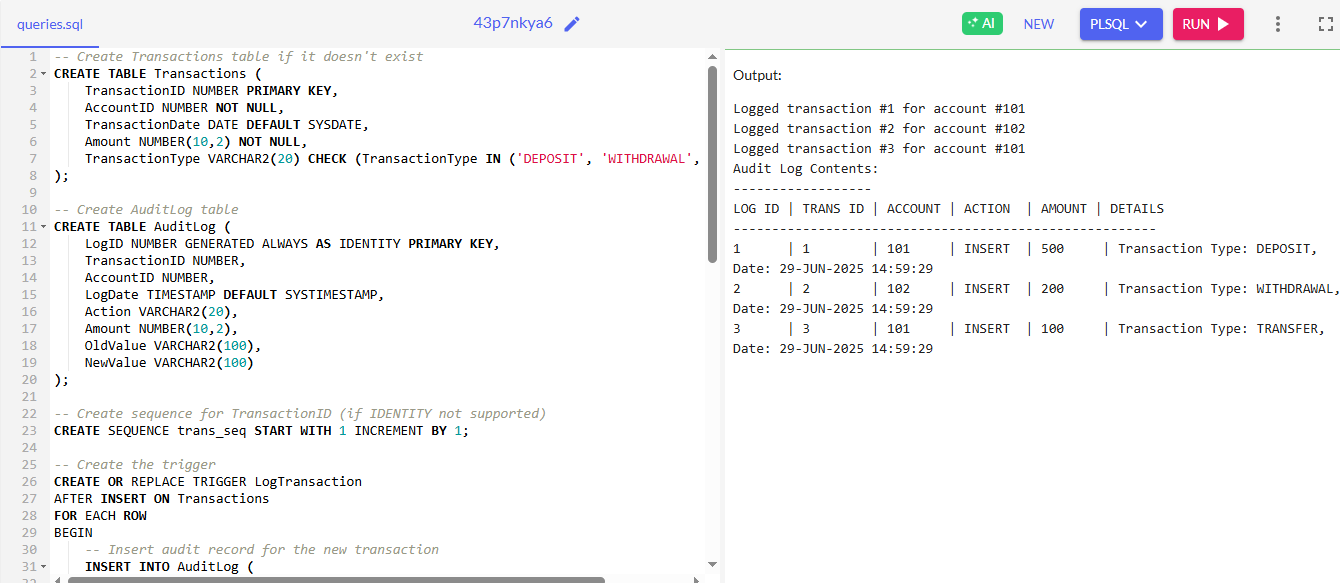
);

END LOOP;

END;

/

**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.

**SOLUTION:**

-- 1. Clean up existing objects

BEGIN

EXECUTE IMMEDIATE 'DROP TRIGGER CheckTransactionRules';

EXECUTE IMMEDIATE 'DROP PROCEDURE TestTransactionRules';

EXECUTE IMMEDIATE 'DROP TABLE Transactions';

EXECUTE IMMEDIATE 'DROP TABLE Accounts';

EXECUTE IMMEDIATE 'DROP SEQUENCE trans\_seq';

EXCEPTION

WHEN OTHERS THEN NULL;

END;

/

-- 2. Create tables with proper constraints

CREATE TABLE Accounts (

AccountID NUMBER PRIMARY KEY,

Balance NUMBER(10,2) NOT NULL

);

CREATE TABLE Transactions (

TransactionID NUMBER PRIMARY KEY,

AccountID NUMBER NOT NULL,

TransactionDate DATE DEFAULT SYSDATE,

Amount NUMBER(10,2) NOT NULL,

TransactionType VARCHAR2(20) CHECK (TransactionType IN ('DEPOSIT', 'WITHDRAWAL', 'TRANSFER')),

CONSTRAINT fk\_account FOREIGN KEY (AccountID) REFERENCES Accounts(AccountID)

);

-- 3. Create sequence for TransactionID

CREATE SEQUENCE trans\_seq START WITH 1 INCREMENT BY 1;

-- 4. Insert sample data

INSERT INTO Accounts VALUES (101, 1000.00);

INSERT INTO Accounts VALUES (102, 500.00);

COMMIT;

-- 5. Create validation trigger (silent version)

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

DECLARE

v\_balance NUMBER;

BEGIN

-- Get current balance with lock

SELECT Balance INTO v\_balance

FROM Accounts

WHERE AccountID = :NEW.AccountID

FOR UPDATE;

-- Validate deposit rules

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

RAISE\_APPLICATION\_ERROR(-20002, ' ');

END IF;

-- Validate withdrawal rules

IF :NEW.TransactionType = 'WITHDRAWAL' THEN

IF :NEW.Amount <= 0 THEN

RAISE\_APPLICATION\_ERROR(-20003, ' ');

ELSIF :NEW.Amount > v\_balance THEN

RAISE\_APPLICATION\_ERROR(-20004, ' ');

END IF;

END IF;

-- Auto-generate TransactionID if null

IF :NEW.TransactionID IS NULL THEN

:NEW.TransactionID := trans\_seq.NEXTVAL;

END IF;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RAISE\_APPLICATION\_ERROR(-20001, ' ');

END;

/

-- 6. Create balance update trigger

CREATE OR REPLACE TRIGGER UpdateAccountBalances

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

-- Update balance for deposits

IF :NEW.TransactionType = 'DEPOSIT' THEN

UPDATE Accounts

SET Balance = Balance + :NEW.Amount

WHERE AccountID = :NEW.AccountID;

END IF;

-- Update balance for withdrawals

IF :NEW.TransactionType = 'WITHDRAWAL' THEN

UPDATE Accounts

SET Balance = Balance - :NEW.Amount

WHERE AccountID = :NEW.AccountID;

END IF;

END;

/

-- 7. Create silent test procedure

CREATE OR REPLACE PROCEDURE TestTransactionRules AS

v\_bal1 NUMBER;

v\_bal2 NUMBER;

BEGIN

-- Get initial balances

SELECT Balance INTO v\_bal1 FROM Accounts WHERE AccountID = 101;

SELECT Balance INTO v\_bal2 FROM Accounts WHERE AccountID = 102;

DBMS\_OUTPUT.PUT\_LINE('=== INITIAL BALANCES ===');

DBMS\_OUTPUT.PUT\_LINE('Account 101: $'||v\_bal1);

DBMS\_OUTPUT.PUT\_LINE('Account 102: $'||v\_bal2);

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

DBMS\_OUTPUT.PUT\_LINE('=== TESTING TRANSACTIONS ===');

-- Test 1: Valid deposit

BEGIN

INSERT INTO Transactions (AccountID, Amount, TransactionType)

VALUES (101, 200.00, 'DEPOSIT');

DBMS\_OUTPUT.PUT\_LINE('1. Valid deposit: PASSED');

EXCEPTION WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('1. Valid deposit: FAILED');

END;

-- Test 2: Negative deposit

BEGIN

INSERT INTO Transactions (AccountID, Amount, TransactionType)

VALUES (101, -50.00, 'DEPOSIT');

DBMS\_OUTPUT.PUT\_LINE('2. Negative deposit: FAILED (expected)');

EXCEPTION WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('2. Negative deposit: FAILED (expected)');

END;

-- Test 3: Valid withdrawal

BEGIN

INSERT INTO Transactions (AccountID, Amount, TransactionType)

VALUES (102, 100.00, 'WITHDRAWAL');

DBMS\_OUTPUT.PUT\_LINE('3. Valid withdrawal: PASSED');

EXCEPTION WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('3. Valid withdrawal: FAILED');

END;

-- Test 4: Over-withdrawal

BEGIN

INSERT INTO Transactions (AccountID, Amount, TransactionType)

VALUES (102, 1000.00, 'WITHDRAWAL');

DBMS\_OUTPUT.PUT\_LINE('4. Over-withdrawal: FAILED (expected)');

EXCEPTION WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('4. Over-withdrawal: FAILED (expected)');

END;

-- Get final balances

SELECT Balance INTO v\_bal1 FROM Accounts WHERE AccountID = 101;

SELECT Balance INTO v\_bal2 FROM Accounts WHERE AccountID = 102;

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

DBMS\_OUTPUT.PUT\_LINE('=== FINAL BALANCES ===');

DBMS\_OUTPUT.PUT\_LINE('Account 101: $'||v\_bal1);

DBMS\_OUTPUT.PUT\_LINE('Account 102: $'||v\_bal2);

-- Show transaction log

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

DBMS\_OUTPUT.PUT\_LINE('=== TRANSACTION LOG ===');

FOR t IN (SELECT \* FROM Transactions ORDER BY TransactionID) LOOP

DBMS\_OUTPUT.PUT\_LINE(

'Txn#'||t.TransactionID||': '||

RPAD(t.TransactionType,10)||' $'||

TO\_CHAR(t.Amount,'9990.99')||

' on account '||t.AccountID

);

END LOOP;

END;

/

-- 8. Execute the test

BEGIN

-- Reset data for consistent testing

UPDATE Accounts SET Balance = 1000 WHERE AccountID = 101;

UPDATE Accounts SET Balance = 500 WHERE AccountID = 102;

DELETE FROM Transactions;

COMMIT;

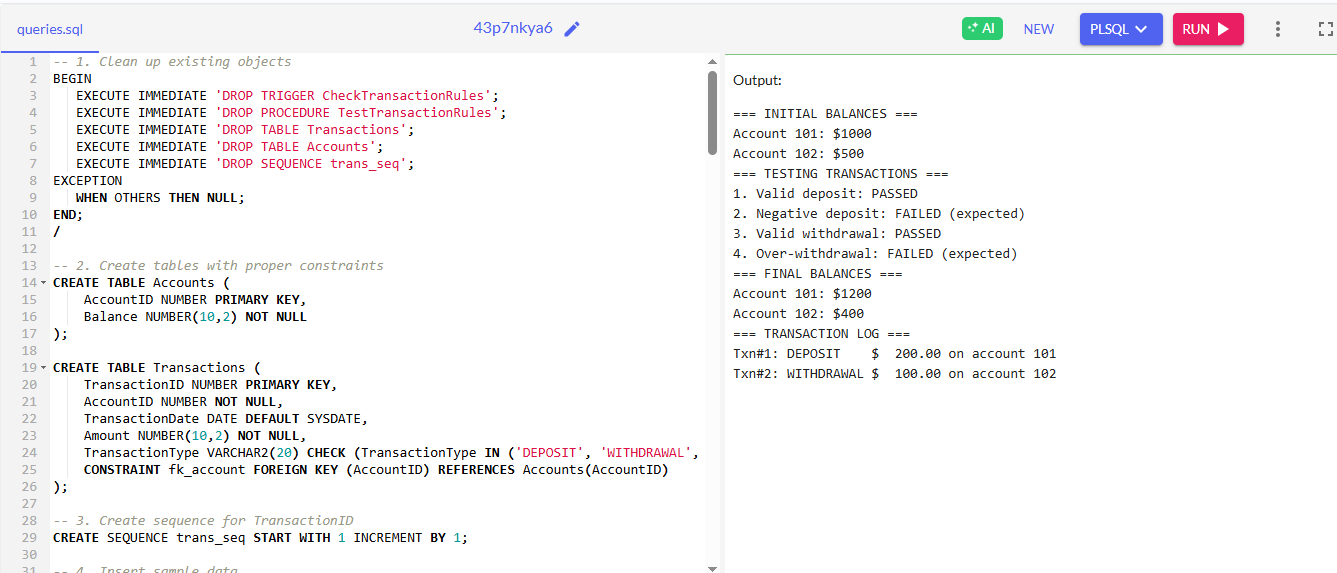
-- Run tests

TestTransactionRules;

END;

/

**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**

BEGIN

EXECUTE IMMEDIATE 'CREATE TABLE customers (

customer\_id NUMBER PRIMARY KEY,

customer\_name VARCHAR2(100),

email VARCHAR2(100),

join\_date DATE

)';

EXCEPTION

WHEN OTHERS THEN NULL; -- Table already exists

END;

/

BEGIN

EXECUTE IMMEDIATE 'CREATE TABLE transactions (

transaction\_id NUMBER PRIMARY KEY,

customer\_id NUMBER,

transaction\_date DATE,

amount NUMBER,

description VARCHAR2(200),

FOREIGN KEY (customer\_id) REFERENCES customers(customer\_id)

)';

EXCEPTION

WHEN OTHERS THEN NULL; -- Table already exists

END;

/

-- Insert sample data if needed

BEGIN

INSERT INTO customers VALUES (101, 'Alice Johnson', 'alice@example.com', TO\_DATE('2023-01-15', 'YYYY-MM-DD'));

INSERT INTO customers VALUES (102, 'Bob Smith', 'bob@example.com', TO\_DATE('2023-02-20', 'YYYY-MM-DD'));

INSERT INTO transactions VALUES (1, 101, TRUNC(SYSDATE, 'MM')+1, 150.50, 'Grocery purchase');

INSERT INTO transactions VALUES (2, 101, TRUNC(SYSDATE, 'MM')+3, 75.25, 'Online shopping');

INSERT INTO transactions VALUES (3, 102, TRUNC(SYSDATE, 'MM')+5, 200.00, 'Utility bill');

INSERT INTO transactions VALUES (4, 101, TRUNC(SYSDATE, 'MM')+7, 50.00, 'Coffee shop');

INSERT INTO transactions VALUES (5, 102, TRUNC(SYSDATE, 'MM')+10, 300.75, 'Electronics');

COMMIT;

EXCEPTION

WHEN OTHERS THEN NULL; -- Data already exists

END;

/

-- PL/SQL Block to Generate Monthly Statements

DECLARE

-- Cursor to get all active customers

CURSOR c\_customers IS

SELECT customer\_id, customer\_name, email

FROM customers

ORDER BY customer\_id;

-- Cursor to get transactions for a specific customer in current month

CURSOR c\_transactions(p\_customer\_id NUMBER) IS

SELECT transaction\_date, amount, description

FROM transactions

WHERE customer\_id = p\_customer\_id

AND transaction\_date BETWEEN TRUNC(SYSDATE, 'MM') AND LAST\_DAY(SYSDATE)

ORDER BY transaction\_date;

v\_total NUMBER := 0;

v\_month VARCHAR2(20) := TO\_CHAR(SYSDATE, 'Month YYYY');

BEGIN

DBMS\_OUTPUT.PUT\_LINE('GENERATING MONTHLY STATEMENTS FOR ' || v\_month);

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

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

-- Process each customer

FOR cust\_rec IN c\_customers LOOP

v\_total := 0;

DBMS\_OUTPUT.PUT\_LINE('Customer: ' || cust\_rec.customer\_name);

DBMS\_OUTPUT.PUT\_LINE('Customer ID: ' || cust\_rec.customer\_id);

DBMS\_OUTPUT.PUT\_LINE('Email: ' || cust\_rec.email);

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

DBMS\_OUTPUT.PUT\_LINE('TRANSACTIONS FOR ' || v\_month);

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

DBMS\_OUTPUT.PUT\_LINE('DATE AMOUNT DESCRIPTION');

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

-- Process each transaction for the current customer

FOR trans\_rec IN c\_transactions(cust\_rec.customer\_id) LOOP

DBMS\_OUTPUT.PUT\_LINE(

TO\_CHAR(trans\_rec.transaction\_date, 'DD-MON-YY') || ' ' ||

LPAD(TO\_CHAR(trans\_rec.amount, '9,999.99'), 9) || ' ' ||

trans\_rec.description

);

v\_total := v\_total + trans\_rec.amount;

END LOOP;

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

DBMS\_OUTPUT.PUT\_LINE('TOTAL FOR MONTH: ' || LPAD(TO\_CHAR(v\_total, '9,999.99'), 20));

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

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

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

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('Monthly statements generation completed.');

EXCEPTION

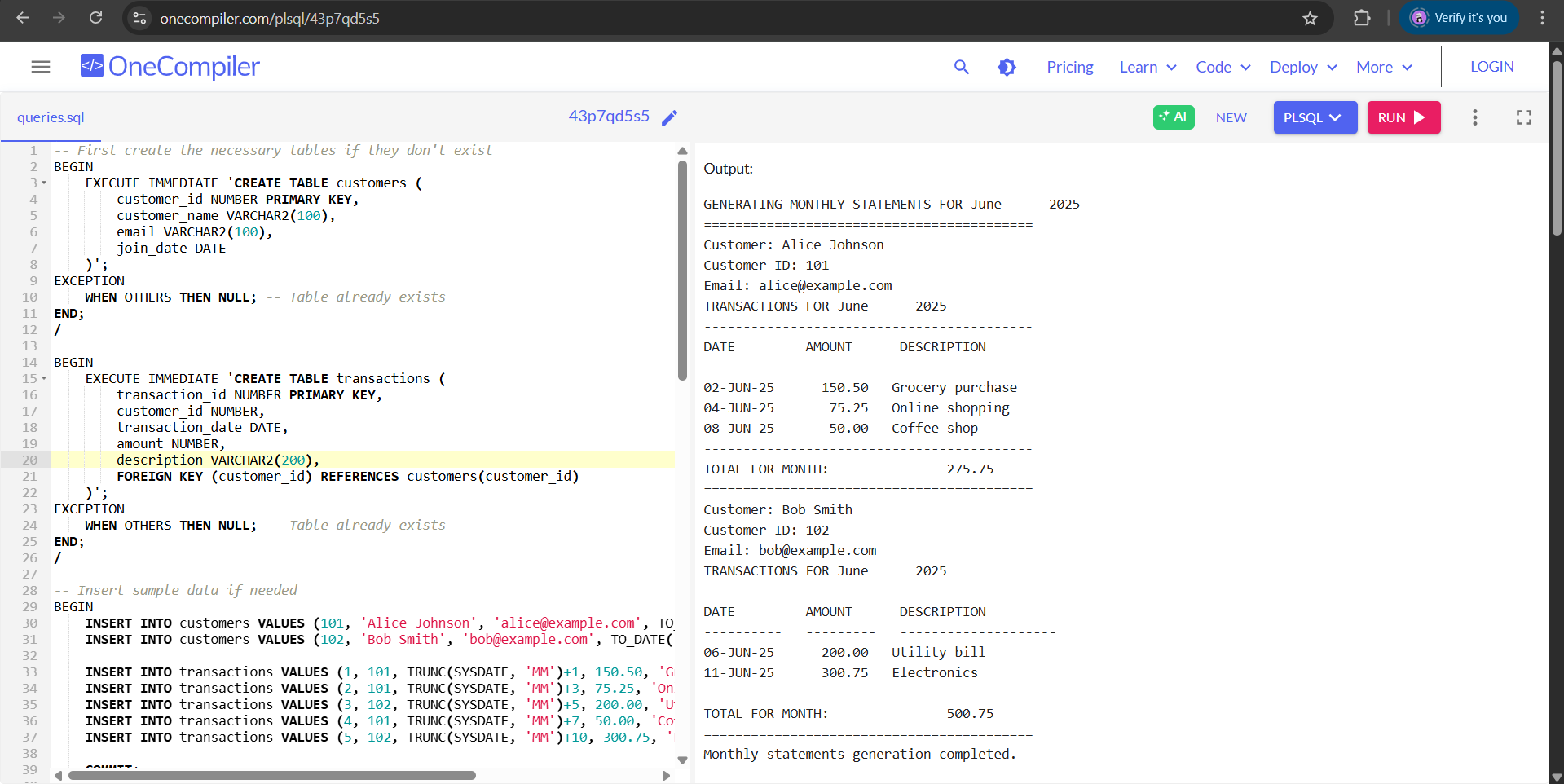
WHEN OTHERS THEN

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

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**

BEGIN

EXECUTE IMMEDIATE 'CREATE TABLE accounts (

account\_id NUMBER PRIMARY KEY,

account\_holder VARCHAR2(100),

balance NUMBER,

last\_fee\_date DATE

)';

EXCEPTION

WHEN OTHERS THEN NULL; -- Table already exists

END;

/

-- Insert sample data if needed

BEGIN

INSERT INTO accounts VALUES (1001, 'Alice Johnson', 1500.00, ADD\_MONTHS(SYSDATE, -13));

INSERT INTO accounts VALUES (1002, 'Bob Smith', 2500.50, ADD\_MONTHS(SYSDATE, -11));

INSERT INTO accounts VALUES (1003, 'Carol Williams', 500.00, ADD\_MONTHS(SYSDATE, -12));

COMMIT;

EXCEPTION

WHEN OTHERS THEN NULL; -- Data already exists

END;

/

-- PL/SQL Block to Apply Annual Fee

DECLARE

-- Constants

ANNUAL\_FEE CONSTANT NUMBER := 25.00; -- $25 annual fee

MIN\_BALANCE CONSTANT NUMBER := 100.00; -- Minimum balance to avoid penalty

-- Cursor to get all accounts that need the annual fee

CURSOR c\_accounts IS

SELECT account\_id, account\_holder, balance

FROM accounts

WHERE last\_fee\_date < ADD\_MONTHS(SYSDATE, -12) -- Fee is due

OR last\_fee\_date IS NULL -- New accounts

ORDER BY account\_id;

v\_fee\_count NUMBER := 0;

v\_skipped\_count NUMBER := 0;

BEGIN

DBMS\_OUTPUT.PUT\_LINE('ANNUAL ACCOUNT MAINTENANCE FEE PROCESSING');

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

DBMS\_OUTPUT.PUT\_LINE('Fee Amount: $' || ANNUAL\_FEE);

DBMS\_OUTPUT.PUT\_LINE('Minimum Balance: $' || MIN\_BALANCE);

DBMS\_OUTPUT.PUT\_LINE('Processing Date: ' || TO\_CHAR(SYSDATE, 'DD-MON-YYYY'));

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

-- Process each account

FOR acc\_rec IN c\_accounts LOOP

BEGIN

-- Check if account has sufficient balance

IF acc\_rec.balance >= (ANNUAL\_FEE + MIN\_BALANCE) THEN

-- Apply the fee

UPDATE accounts

SET balance = balance - ANNUAL\_FEE,

last\_fee\_date = SYSDATE

WHERE account\_id = acc\_rec.account\_id;

v\_fee\_count := v\_fee\_count + 1;

DBMS\_OUTPUT.PUT\_LINE('Account: ' || LPAD(acc\_rec.account\_id, 10) ||

' | Holder: ' || RPAD(acc\_rec.account\_holder, 20) ||

' | Fee Applied: $' || LPAD(TO\_CHAR(ANNUAL\_FEE, '999.99'), 6) ||

' | New Balance: $' || LPAD(TO\_CHAR(acc\_rec.balance - ANNUAL\_FEE, '9,999.99'), 10));

ELSE

v\_skipped\_count := v\_skipped\_count + 1;

DBMS\_OUTPUT.PUT\_LINE('Account: ' || LPAD(acc\_rec.account\_id, 10) ||

' | Holder: ' || RPAD(acc\_rec.account\_holder, 20) ||

' | SKIPPED - Insufficient funds (Balance: $' ||

LPAD(TO\_CHAR(acc\_rec.balance, '9,999.99'), 10) || ')');

END IF;

EXCEPTION

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('Error processing account ' || acc\_rec.account\_id || ': ' || SQLERRM);

END;

END LOOP;

COMMIT;

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

DBMS\_OUTPUT.PUT\_LINE('PROCESSING SUMMARY');

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

DBMS\_OUTPUT.PUT\_LINE('Accounts processed: ' || (v\_fee\_count + v\_skipped\_count));

DBMS\_OUTPUT.PUT\_LINE('Fees applied: ' || v\_fee\_count);

DBMS\_OUTPUT.PUT\_LINE('Accounts skipped: ' || v\_skipped\_count);

DBMS\_OUTPUT.PUT\_LINE('Total fees: $' || (v\_fee\_count \* ANNUAL\_FEE));

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

EXCEPTION

WHEN OTHERS THEN

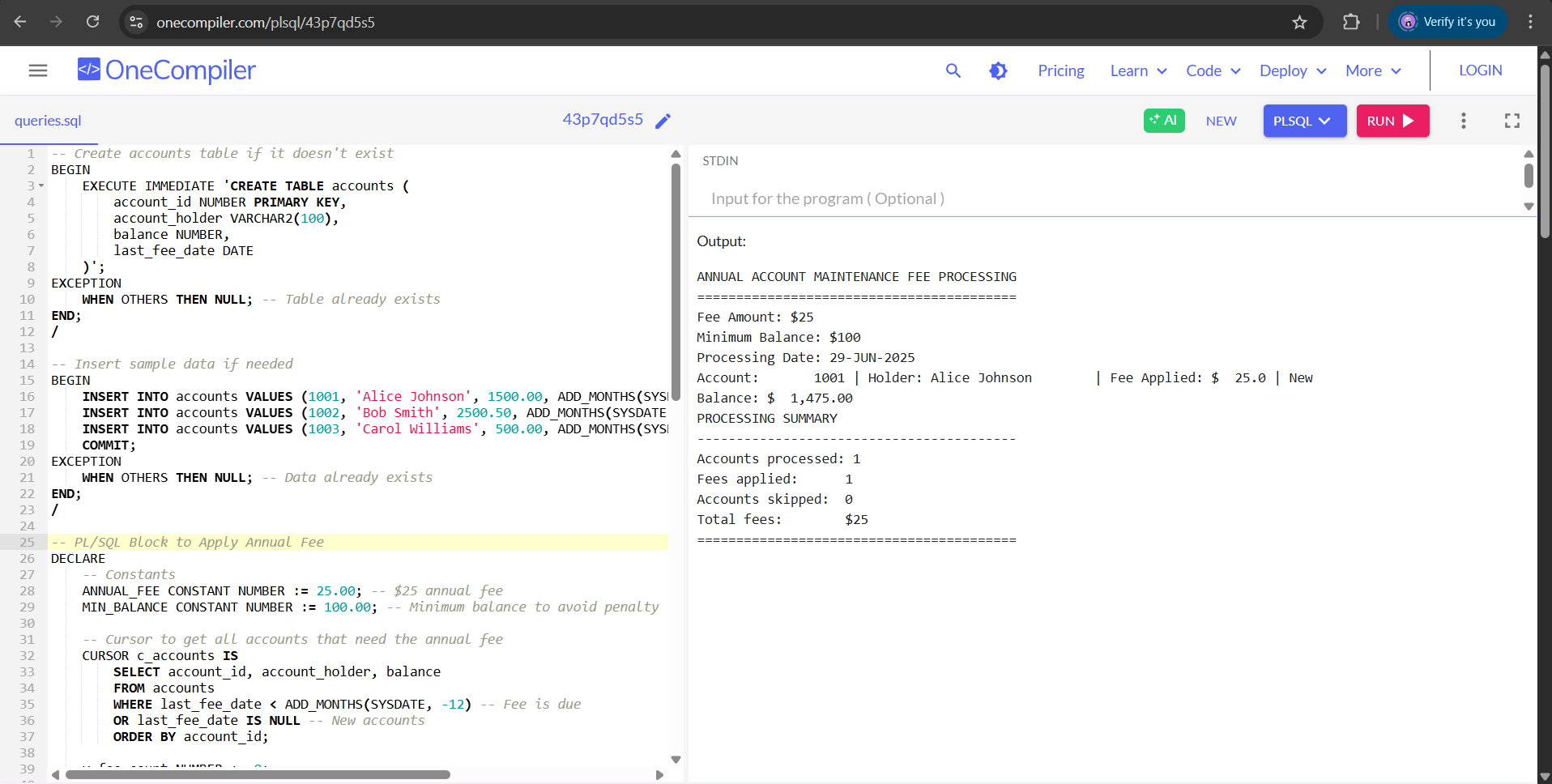
ROLLBACK;

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

END;

/

**OUTPUT**



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

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

**SOLUTION**

-- 1. First create the loans table if it doesn't exist

BEGIN

EXECUTE IMMEDIATE 'CREATE TABLE loans (

loan\_id NUMBER PRIMARY KEY,

loan\_type VARCHAR2(20),

current\_interest\_rate NUMBER,

principal\_amount NUMBER,

status VARCHAR2(10),

last\_rate\_change\_date DATE

)';

DBMS\_OUTPUT.PUT\_LINE('Table LOANS created successfully');

EXCEPTION

WHEN OTHERS THEN

IF SQLCODE = -955 THEN

DBMS\_OUTPUT.PUT\_LINE('Table LOANS already exists');

ELSE

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

END IF;

END;

/

-- 2. Insert sample data if needed

BEGIN

INSERT INTO loans VALUES (1001, 'MORTGAGE', 4.0, 250000, 'ACTIVE', TO\_DATE('2022-01-15', 'YYYY-MM-DD'));

INSERT INTO loans VALUES (1002, 'PERSONAL', 9.5, 5000, 'ACTIVE', TO\_DATE('2022-03-20', 'YYYY-MM-DD'));

INSERT INTO loans VALUES (1003, 'PERSONAL', 8.0, 15000, 'ACTIVE', TO\_DATE('2022-05-10', 'YYYY-MM-DD'));

INSERT INTO loans VALUES (1004, 'AUTO', 6.5, 20000, 'CLOSED', TO\_DATE('2022-02-28', 'YYYY-MM-DD'));

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Sample data inserted successfully');

EXCEPTION

WHEN OTHERS THEN

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

ROLLBACK;

END;

/

-- 3. PL/SQL block to update interest rates

DECLARE

CURSOR c\_loans IS

SELECT loan\_id, loan\_type, current\_interest\_rate, principal\_amount

FROM loans

WHERE status = 'ACTIVE'

FOR UPDATE;

v\_new\_rate NUMBER;

v\_count NUMBER := 0;

BEGIN

DBMS\_OUTPUT.PUT\_LINE('Starting interest rate update...');

FOR loan\_rec IN c\_loans LOOP

-- Determine new rate based on policy

IF loan\_rec.loan\_type = 'MORTGAGE' THEN

v\_new\_rate := 3.5;

ELSIF loan\_rec.loan\_type = 'PERSONAL' AND loan\_rec.principal\_amount > 10000 THEN

v\_new\_rate := 7.25;

ELSIF loan\_rec.loan\_type = 'PERSONAL' THEN

v\_new\_rate := 8.5;

ELSE

v\_new\_rate := 5.0;

END IF;

-- Update the loan

UPDATE loans

SET current\_interest\_rate = v\_new\_rate,

last\_rate\_change\_date = SYSDATE

WHERE CURRENT OF c\_loans;

DBMS\_OUTPUT.PUT\_LINE('Loan ' || loan\_rec.loan\_id || ': ' ||

loan\_rec.loan\_type || ' loan updated from ' ||

loan\_rec.current\_interest\_rate || '% to ' || v\_new\_rate || '%');

v\_count := v\_count + 1;

END LOOP;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Successfully updated ' || v\_count || ' active loans');

EXCEPTION

WHEN OTHERS THEN

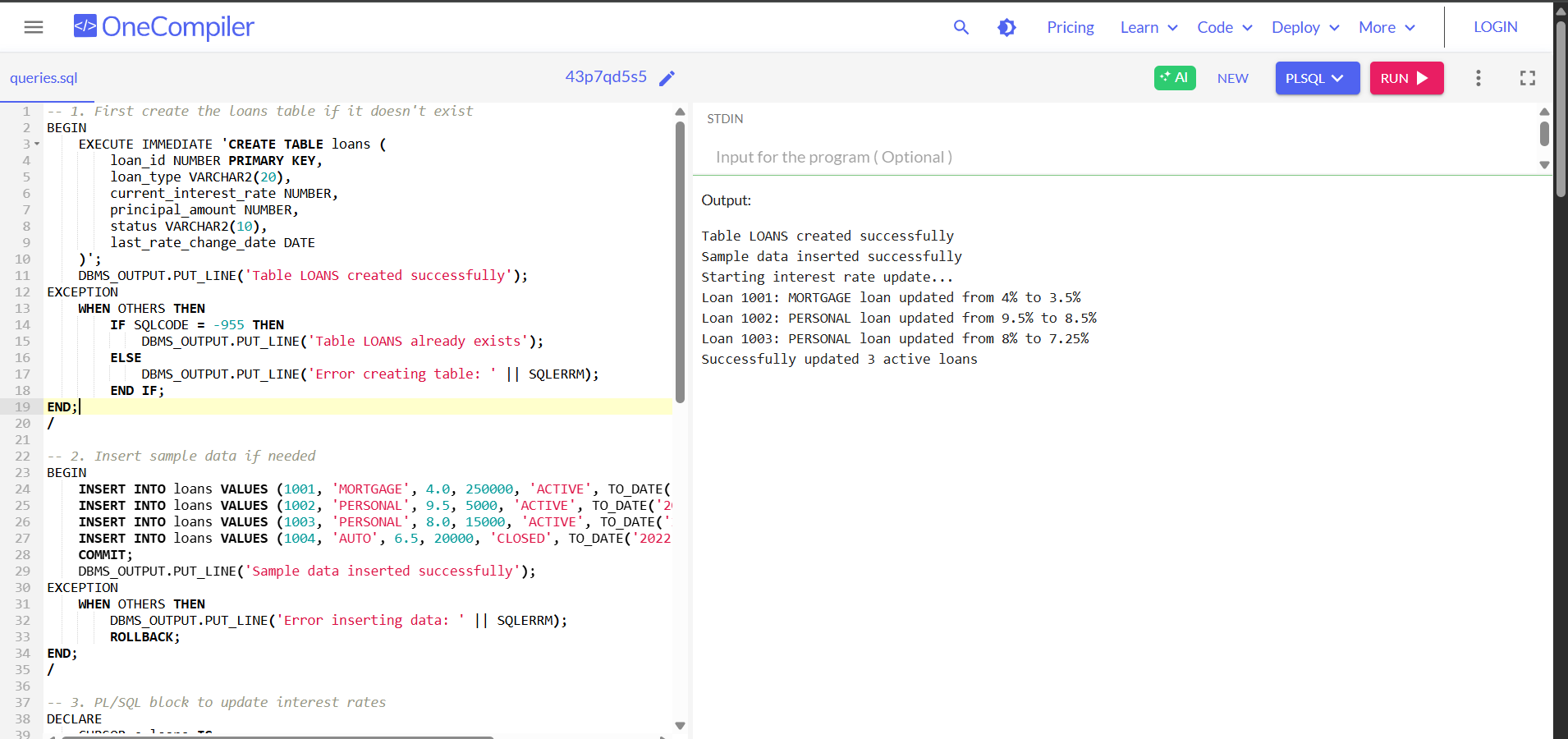
ROLLBACK;

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

END;

/

**OUTPUT**



**Exercise 7: Packages**

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

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

**SOLUTION**

BEGIN

EXECUTE IMMEDIATE 'CREATE TABLE customers (

customer\_id NUMBER PRIMARY KEY,

customer\_name VARCHAR2(100) NOT NULL,

email VARCHAR2(100) UNIQUE,

phone VARCHAR2(20),

registration\_date DATE DEFAULT SYSDATE,

status VARCHAR2(10) DEFAULT ''ACTIVE''

)';

EXCEPTION

WHEN OTHERS THEN NULL; -- Table already exists

END;

/

BEGIN

EXECUTE IMMEDIATE 'CREATE TABLE accounts (

account\_id NUMBER PRIMARY KEY,

customer\_id NUMBER,

balance NUMBER DEFAULT 0,

account\_type VARCHAR2(20),

FOREIGN KEY (customer\_id) REFERENCES customers(customer\_id)

)';

EXCEPTION

WHEN OTHERS THEN NULL; -- Table already exists

END;

/

-- Customer Management Package Specification

CREATE OR REPLACE PACKAGE CustomerManagement AS

-- Procedure to add a new customer

PROCEDURE add\_customer(

p\_customer\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_email IN VARCHAR2,

p\_phone IN VARCHAR2 DEFAULT NULL

);

-- Procedure to update customer details

PROCEDURE update\_customer(

p\_customer\_id IN NUMBER,

p\_name IN VARCHAR2 DEFAULT NULL,

p\_email IN VARCHAR2 DEFAULT NULL,

p\_phone IN VARCHAR2 DEFAULT NULL,

p\_status IN VARCHAR2 DEFAULT NULL

);

-- Function to get customer balance

FUNCTION get\_customer\_balance(

p\_customer\_id IN NUMBER

) RETURN NUMBER;

-- Procedure to display customer info

PROCEDURE display\_customer(

p\_customer\_id IN NUMBER

);

END CustomerManagement;

/

-- Customer Management Package Body

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

-- Add new customer procedure

PROCEDURE add\_customer(

p\_customer\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_email IN VARCHAR2,

p\_phone IN VARCHAR2 DEFAULT NULL

) IS

BEGIN

INSERT INTO customers (customer\_id, customer\_name, email, phone)

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

COMMIT;

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

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

ROLLBACK;

WHEN OTHERS THEN

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

ROLLBACK;

END add\_customer;

-- Update customer details procedure

PROCEDURE update\_customer(

p\_customer\_id IN NUMBER,

p\_name IN VARCHAR2 DEFAULT NULL,

p\_email IN VARCHAR2 DEFAULT NULL,

p\_phone IN VARCHAR2 DEFAULT NULL,

p\_status IN VARCHAR2 DEFAULT NULL

) IS

v\_updates NUMBER := 0;

BEGIN

IF p\_name IS NOT NULL THEN

UPDATE customers SET customer\_name = p\_name WHERE customer\_id = p\_customer\_id;

v\_updates := v\_updates + SQL%ROWCOUNT;

END IF;

IF p\_email IS NOT NULL THEN

UPDATE customers SET email = p\_email WHERE customer\_id = p\_customer\_id;

v\_updates := v\_updates + SQL%ROWCOUNT;

END IF;

IF p\_phone IS NOT NULL THEN

UPDATE customers SET phone = p\_phone WHERE customer\_id = p\_customer\_id;

v\_updates := v\_updates + SQL%ROWCOUNT;

END IF;

IF p\_status IS NOT NULL THEN

UPDATE customers SET status = p\_status WHERE customer\_id = p\_customer\_id;

v\_updates := v\_updates + SQL%ROWCOUNT;

END IF;

IF v\_updates = 0 THEN

DBMS\_OUTPUT.PUT\_LINE('No updates provided for customer ' || p\_customer\_id);

ELSE

COMMIT;

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

END IF;

EXCEPTION

WHEN OTHERS THEN

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

ROLLBACK;

END update\_customer;

-- Get customer balance function

FUNCTION get\_customer\_balance(

p\_customer\_id IN NUMBER

) RETURN NUMBER IS

v\_balance NUMBER := 0;

BEGIN

SELECT NVL(SUM(balance), 0)

INTO v\_balance

FROM accounts

WHERE customer\_id = p\_customer\_id;

RETURN v\_balance;

EXCEPTION

WHEN OTHERS THEN

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

RETURN -1; -- Return -1 to indicate error

END get\_customer\_balance;

-- Display customer info procedure

PROCEDURE display\_customer(

p\_customer\_id IN NUMBER

) IS

v\_customer customers%ROWTYPE;

v\_balance NUMBER;

BEGIN

-- Get customer details

SELECT \*

INTO v\_customer

FROM customers

WHERE customer\_id = p\_customer\_id;

-- Get customer balance

v\_balance := get\_customer\_balance(p\_customer\_id);

-- Display information

DBMS\_OUTPUT.PUT\_LINE('CUSTOMER DETAILS');

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

DBMS\_OUTPUT.PUT\_LINE('ID: ' || v\_customer.customer\_id);

DBMS\_OUTPUT.PUT\_LINE('Name: ' || v\_customer.customer\_name);

DBMS\_OUTPUT.PUT\_LINE('Email: ' || v\_customer.email);

DBMS\_OUTPUT.PUT\_LINE('Phone: ' || NVL(v\_customer.phone, 'Not provided'));

DBMS\_OUTPUT.PUT\_LINE('Status: ' || v\_customer.status);

DBMS\_OUTPUT.PUT\_LINE('Joined: ' || TO\_CHAR(v\_customer.registration\_date, 'DD-MON-YYYY'));

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

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('Customer ' || p\_customer\_id || ' not found');

WHEN OTHERS THEN

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

END display\_customer;

END CustomerManagement;

/

-- Test the package

BEGIN

-- Add some test customers

CustomerManagement.add\_customer(101, 'Alice Johnson', 'alice@example.com', '555-1234');

CustomerManagement.add\_customer(102, 'Bob Smith', 'bob@example.com', '555-5678');

-- Add some accounts for testing balance

INSERT INTO accounts VALUES (1001, 101, 1500.50, 'CHECKING');

INSERT INTO accounts VALUES (1002, 101, 2500.00, 'SAVINGS');

INSERT INTO accounts VALUES (1003, 102, 500.75, 'CHECKING');

COMMIT;

-- Test the package functions

DBMS\_OUTPUT.PUT\_LINE('Alice''s balance: $' || CustomerManagement.get\_customer\_balance(101));

DBMS\_OUTPUT.PUT\_LINE('Bob''s balance: $' || CustomerManagement.get\_customer\_balance(102));

-- Update customer details

CustomerManagement.update\_customer(

p\_customer\_id => 101,

p\_phone => '555-4321',

p\_status => 'PREMIUM'

);

-- Display customer info

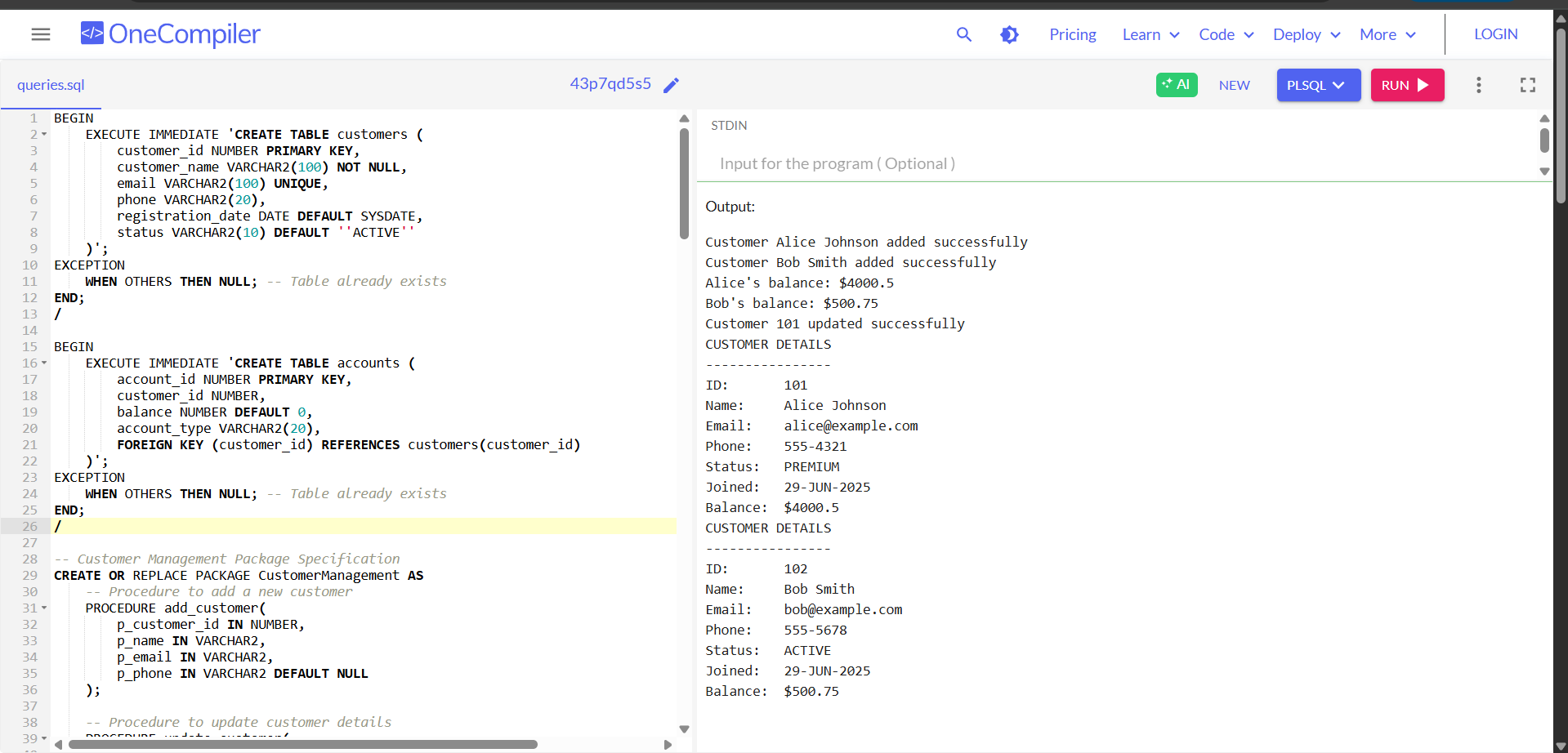
CustomerManagement.display\_customer(101);

CustomerManagement.display\_customer(102);

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.

**SOLUTION:**

-- Create employees table if not exists

CREATE TABLE employees (

emp\_id NUMBER PRIMARY KEY,

emp\_name VARCHAR2(100) NOT NULL,

job\_title VARCHAR2(50),

salary NUMBER(10,2) NOT NULL,

hire\_date DATE DEFAULT SYSDATE

);

-- Create sequence for employee IDs

CREATE SEQUENCE emp\_id\_seq START WITH 1001;

-- Package specification

CREATE OR REPLACE PACKAGE EmployeeManagement AS

-- Procedure to hire new employee

PROCEDURE hire\_employee(

p\_name IN VARCHAR2,

p\_job\_title IN VARCHAR2,

p\_salary IN NUMBER

);

-- Procedure to update employee details

PROCEDURE update\_employee(

p\_emp\_id IN NUMBER,

p\_job\_title IN VARCHAR2 DEFAULT NULL,

p\_salary IN NUMBER DEFAULT NULL

);

-- Function to calculate annual salary

FUNCTION calculate\_annual\_salary(

p\_emp\_id IN NUMBER

) RETURN NUMBER;

END EmployeeManagement;

/

-- Package body

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

-- Hire new employee

PROCEDURE hire\_employee(

p\_name IN VARCHAR2,

p\_job\_title IN VARCHAR2,

p\_salary IN NUMBER

) IS

BEGIN

INSERT INTO employees (emp\_id, emp\_name, job\_title, salary)

VALUES (emp\_id\_seq.NEXTVAL, p\_name, p\_job\_title, p\_salary);

DBMS\_OUTPUT.PUT\_LINE('Employee hired with ID: ' || emp\_id\_seq.CURRVAL);

EXCEPTION

WHEN OTHERS THEN

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

END hire\_employee;

-- Update employee details

PROCEDURE update\_employee(

p\_emp\_id IN NUMBER,

p\_job\_title IN VARCHAR2 DEFAULT NULL,

p\_salary IN NUMBER DEFAULT NULL

) IS

BEGIN

UPDATE employees

SET job\_title = NVL(p\_job\_title, job\_title),

salary = NVL(p\_salary, salary)

WHERE emp\_id = p\_emp\_id;

IF SQL%ROWCOUNT = 0 THEN

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

ELSE

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

END IF;

EXCEPTION

WHEN OTHERS THEN

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

END update\_employee;

-- Calculate annual salary

FUNCTION calculate\_annual\_salary(

p\_emp\_id IN NUMBER

) RETURN NUMBER IS

v\_monthly\_salary NUMBER;

BEGIN

SELECT salary INTO v\_monthly\_salary

FROM employees

WHERE emp\_id = p\_emp\_id;

RETURN v\_monthly\_salary \* 12;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN NULL;

END calculate\_annual\_salary;

END EmployeeManagement;

/

-- Test the package

BEGIN

-- Hire new employees

EmployeeManagement.hire\_employee('John Smith', 'Developer', 5000);

EmployeeManagement.hire\_employee('Sarah Johnson', 'Manager', 8000);

-- Update employee details

EmployeeManagement.update\_employee(1001, 'Senior Developer', 6000);

EmployeeManagement.update\_employee(9999); -- Test non-existent employee

-- Calculate annual salaries

DBMS\_OUTPUT.PUT\_LINE('Annual salary for emp 1001: $' ||

EmployeeManagement.calculate\_annual\_salary(1001));

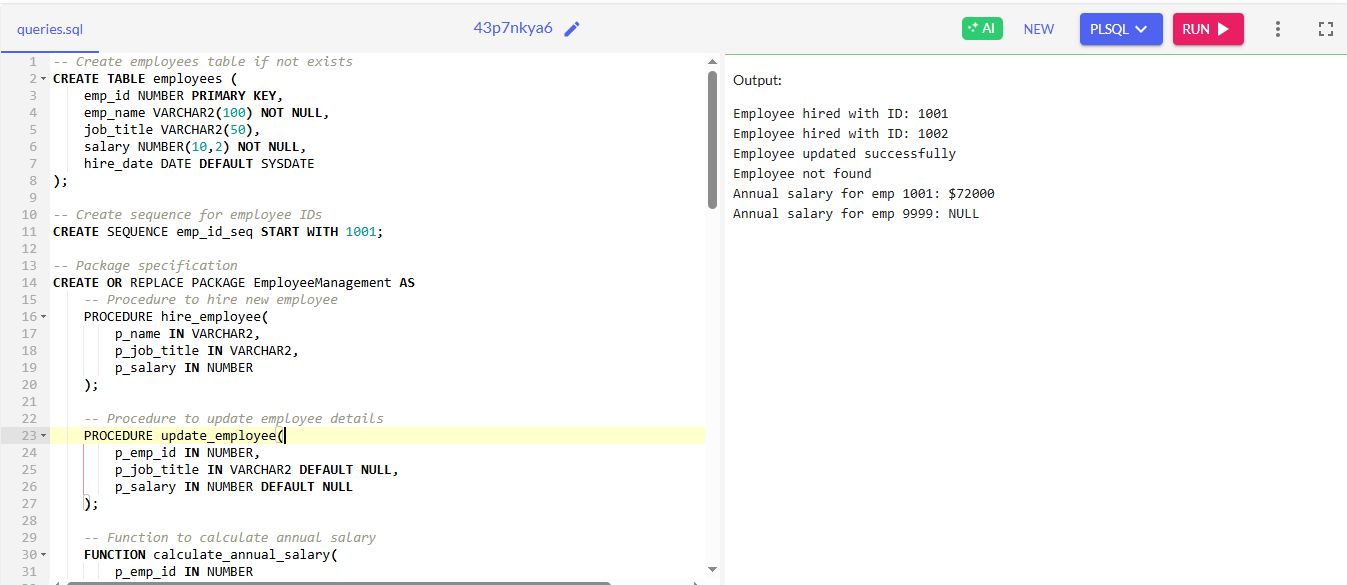
DBMS\_OUTPUT.PUT\_LINE('Annual salary for emp 9999: ' ||

NVL(TO\_CHAR(EmployeeManagement.calculate\_annual\_salary(9999)), 'NULL'));

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.

**SOLUTION**

CREATE TABLE customers (

customer\_id NUMBER PRIMARY KEY,

customer\_name VARCHAR2(100) NOT NULL,

email VARCHAR2(100) UNIQUE,

status VARCHAR2(10) DEFAULT 'ACTIVE'

);

CREATE TABLE accounts (

account\_id NUMBER PRIMARY KEY,

customer\_id NUMBER REFERENCES customers(customer\_id),

account\_type VARCHAR2(20) NOT NULL,

balance NUMBER DEFAULT 0,

status VARCHAR2(10) DEFAULT 'ACTIVE',

open\_date DATE DEFAULT SYSDATE,

close\_date DATE

);

-- 2. Insert sample data with proper values

INSERT INTO customers (customer\_id, customer\_name, email)

VALUES (101, 'Alice Johnson', 'alice@example.com');

INSERT INTO accounts (account\_id, customer\_id, account\_type, balance)

VALUES (1001, 101, 'SAVINGS', 500);

COMMIT;

-- 3. AccountOperations Package Specification

CREATE OR REPLACE PACKAGE AccountOperations AS

PROCEDURE open\_account(

p\_account\_id NUMBER,

p\_customer\_id NUMBER,

p\_account\_type VARCHAR2,

p\_initial\_deposit NUMBER DEFAULT 0

);

PROCEDURE close\_account(p\_account\_id NUMBER);

FUNCTION get\_balance(p\_customer\_id NUMBER) RETURN NUMBER;

PROCEDURE display\_account\_info(p\_account\_id NUMBER);

END AccountOperations;

/

-- 4. AccountOperations Package Body

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

PROCEDURE open\_account(

p\_account\_id NUMBER,

p\_customer\_id NUMBER,

p\_account\_type VARCHAR2,

p\_initial\_deposit NUMBER DEFAULT 0

) IS

BEGIN

-- Validate customer exists

DECLARE

v\_customer\_count NUMBER;

BEGIN

SELECT COUNT(\*) INTO v\_customer\_count

FROM customers

WHERE customer\_id = p\_customer\_id;

IF v\_customer\_count = 0 THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Customer does not exist');

END IF;

END;

-- Validate deposit amount

IF p\_initial\_deposit < 0 THEN

RAISE\_APPLICATION\_ERROR(-20002, 'Initial deposit cannot be negative');

END IF;

-- Create account

INSERT INTO accounts (

account\_id,

customer\_id,

account\_type,

balance

) VALUES (

p\_account\_id,

p\_customer\_id,

p\_account\_type,

p\_initial\_deposit

);

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Account ' || p\_account\_id || ' opened successfully');

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END open\_account;

PROCEDURE close\_account(p\_account\_id NUMBER) IS

v\_current\_balance NUMBER;

BEGIN

-- Get current balance

SELECT balance INTO v\_current\_balance

FROM accounts

WHERE account\_id = p\_account\_id

AND status = 'ACTIVE';

-- Close account

UPDATE accounts

SET status = 'CLOSED',

close\_date = SYSDATE,

balance = 0

WHERE account\_id = p\_account\_id;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Account ' || p\_account\_id || ' closed successfully');

DBMS\_OUTPUT.PUT\_LINE('Final balance refunded: ' || v\_current\_balance);

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

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

WHEN OTHERS THEN

ROLLBACK;

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

END close\_account;

FUNCTION get\_balance(p\_customer\_id NUMBER) RETURN NUMBER IS

v\_total\_balance NUMBER := 0;

BEGIN

SELECT NVL(SUM(balance), 0)

INTO v\_total\_balance

FROM accounts

WHERE customer\_id = p\_customer\_id

AND status = 'ACTIVE';

RETURN v\_total\_balance;

EXCEPTION

WHEN OTHERS THEN

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

RETURN -1; -- Error indicator

END get\_balance;

PROCEDURE display\_account\_info(p\_account\_id NUMBER) IS

v\_account\_info VARCHAR2(4000);

BEGIN

SELECT 'Account ID: ' || account\_id || CHR(10) ||

'Type: ' || account\_type || CHR(10) ||

'Balance: ' || balance || CHR(10) ||

'Status: ' || status || CHR(10) ||

'Opened: ' || TO\_CHAR(open\_date, 'DD-MON-YYYY') ||

CASE WHEN close\_date IS NOT NULL THEN

CHR(10) || 'Closed: ' || TO\_CHAR(close\_date, 'DD-MON-YYYY')

ELSE '' END

INTO v\_account\_info

FROM accounts

WHERE account\_id = p\_account\_id;

DBMS\_OUTPUT.PUT\_LINE(v\_account\_info);

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('Account ' || p\_account\_id || ' not found');

WHEN OTHERS THEN

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

END display\_account\_info;

END AccountOperations;

/

-- 5. Test the package

BEGIN

-- Open a new account

AccountOperations.open\_account(1002, 101, 'CHECKING', 1000);

-- Display account info

AccountOperations.display\_account\_info(1001);

-- Get total balance

DBMS\_OUTPUT.PUT\_LINE('Total balance: ' || AccountOperations.get\_balance(101));

-- Close an account

AccountOperations.close\_account(1001);

-- Verify balance after closing

DBMS\_OUTPUT.PUT\_LINE('Updated balance: ' || AccountOperations.get\_balance(101));

END;

/

**OUTPUT**

