**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.

**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.

**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.

**Source Code:**

-- 1. Drop tables if they exist (optional cleanup)

BEGIN

EXECUTE IMMEDIATE 'DROP TABLE Loans';

EXCEPTION WHEN OTHERS THEN NULL;

END;

/

BEGIN

EXECUTE IMMEDIATE 'DROP TABLE Customers';

EXCEPTION WHEN OTHERS THEN NULL;

END;

/

-- 2. Create Customers table

CREATE TABLE Customers (

CustomerID NUMBER PRIMARY KEY,

Name VARCHAR2(100),

DOB DATE,

Balance NUMBER,

LastModified DATE,

IsVIP CHAR(1)

);

-- 3. Create Loans table

CREATE TABLE Loans (

LoanID NUMBER PRIMARY KEY,

CustomerID NUMBER,

LoanAmount NUMBER,

InterestRate NUMBER,

StartDate DATE,

EndDate DATE,

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

-- 4. Insert sample customers

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'), 1500, SYSDATE);

-- 5. Insert sample loans

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

VALUES (1, 1, 10000, 7.5, SYSDATE, ADD\_MONTHS(SYSDATE, 60));

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

VALUES (2, 2, 8000, 6.5, SYSDATE, ADD\_MONTHS(SYSDATE, 48));

-- 6. Scenario 1: Apply 1% interest discount for customers above 60

BEGIN

FOR c IN (SELECT CustomerID, DOB FROM Customers) LOOP

IF FLOOR(MONTHS\_BETWEEN(SYSDATE, c.DOB)/12) > 60 THEN

UPDATE Loans

SET InterestRate = InterestRate - 1

WHERE CustomerID = c.CustomerID;

END IF;

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('Interest rate updated for senior customers.');

END;

/

-- 7. Scenario 2: Promote to VIP based on balance

BEGIN

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

IF c.Balance > 10000 THEN

UPDATE Customers

SET IsVIP = 'Y'

WHERE CustomerID = c.CustomerID;

ELSE

UPDATE Customers

SET IsVIP = 'N'

WHERE CustomerID = c.CustomerID;

END IF;

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('VIP status updated.');

END;

/

-- 8. Scenario 3: Loan reminders for loans due in the next 30 days

BEGIN

FOR l IN (

SELECT LoanID, CustomerID, EndDate

FROM Loans

WHERE EndDate BETWEEN SYSDATE AND SYSDATE + 30

) LOOP

DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan ID ' || l.LoanID ||

' for Customer ID ' || l.CustomerID ||

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

END LOOP;

END;

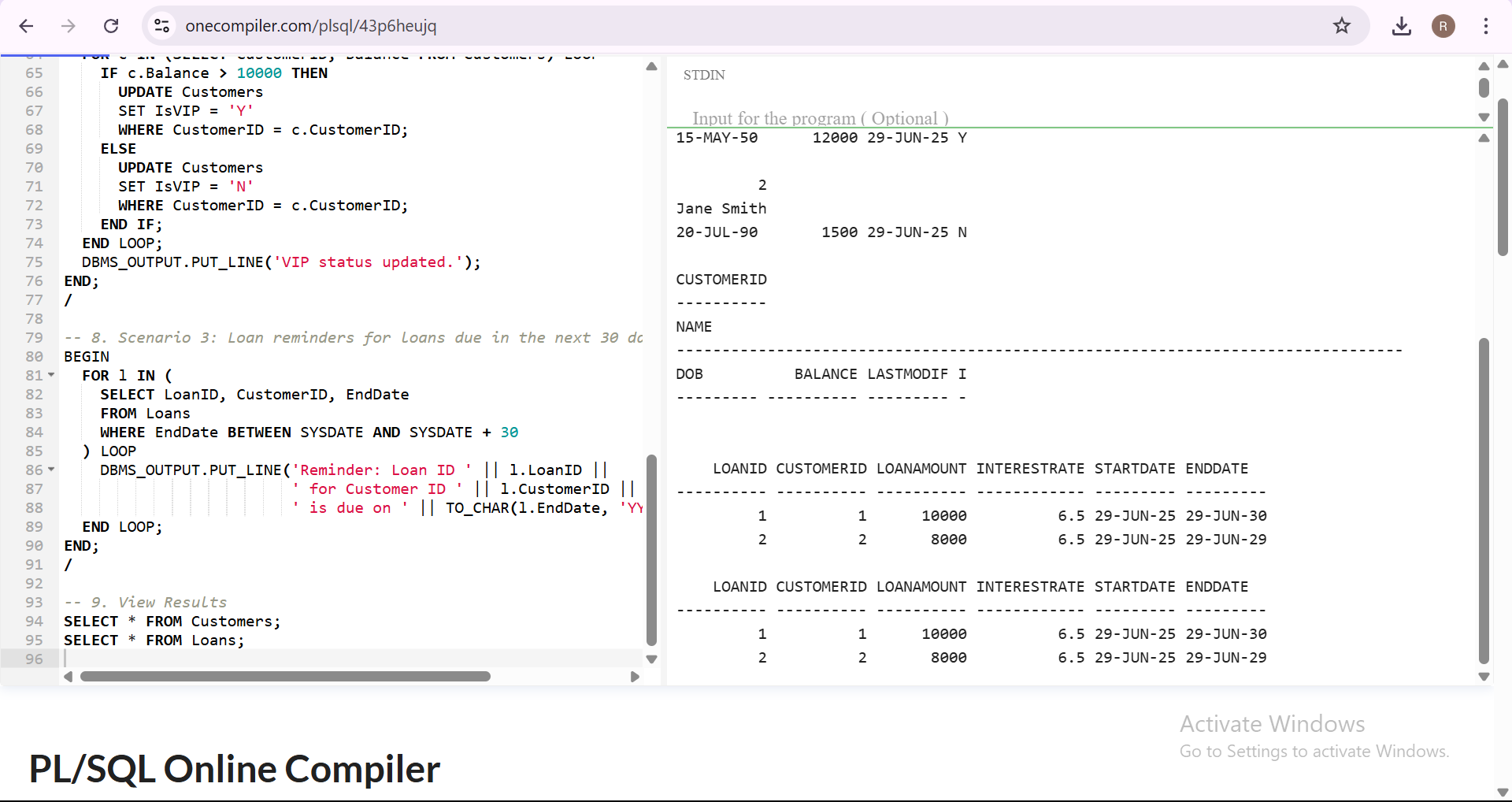
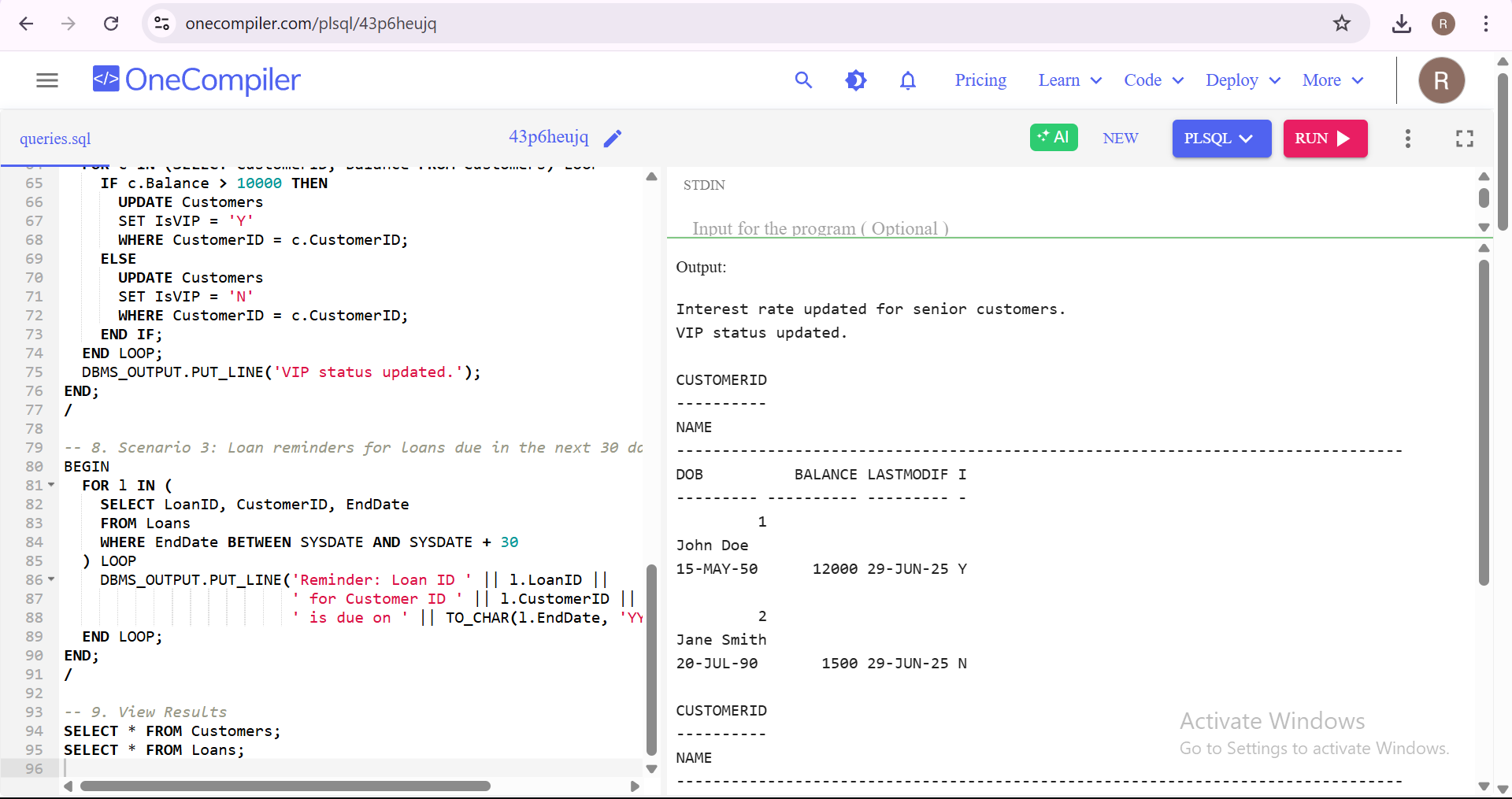
/

-- 9. View Results

SELECT \* FROM Customers;

SELECT \* FROM Loans;

**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.

**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.

**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.

**Source Code:**

-- 1. Drop tables if they already exist

BEGIN

EXECUTE IMMEDIATE 'DROP TABLE Accounts';

EXCEPTION

WHEN OTHERS THEN NULL;

END;

/

BEGIN

EXECUTE IMMEDIATE 'DROP TABLE Employees';

EXCEPTION

WHEN OTHERS THEN NULL;

END;

/

BEGIN

EXECUTE IMMEDIATE 'DROP TABLE Customers';

EXCEPTION

WHEN OTHERS THEN NULL;

END;

/

-- 2. Create necessary tables

-- Customers table

CREATE TABLE Customers (

CustomerID NUMBER PRIMARY KEY,

Name VARCHAR2(100),

DOB DATE,

Balance NUMBER,

LastModified DATE,

IsVIP CHAR(1)

);

-- Accounts table

CREATE TABLE Accounts (

AccountID NUMBER PRIMARY KEY,

CustomerID NUMBER,

AccountType VARCHAR2(20),

Balance NUMBER,

LastModified DATE,

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

-- Employees table

CREATE TABLE Employees (

EmployeeID NUMBER PRIMARY KEY,

Name VARCHAR2(100),

Position VARCHAR2(50),

Salary NUMBER,

Department VARCHAR2(50),

HireDate DATE

);

-- 3. Insert sample data

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

INSERT INTO Customers VALUES (2, 'Jane Smith', TO\_DATE('1990-07-20', 'YYYY-MM-DD'), 1500, SYSDATE, 'N');

INSERT INTO Accounts VALUES (1, 1, 'Savings', 12000, SYSDATE);

INSERT INTO Accounts VALUES (2, 2, 'Savings', 1500, SYSDATE);

INSERT INTO Employees VALUES (1, 'Alice', 'Manager', 70000, 'HR', SYSDATE);

INSERT INTO Employees VALUES (2, 'Bob', 'Developer', 50000, 'IT', SYSDATE);

COMMIT;

/

-- 4. Procedure: SafeTransferFunds

CREATE OR REPLACE PROCEDURE SafeTransferFunds (

p\_from\_account IN NUMBER,

p\_to\_account IN NUMBER,

p\_amount IN NUMBER

) IS

v\_balance NUMBER;

BEGIN

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

IF v\_balance < p\_amount THEN

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

END IF;

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

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

COMMIT;

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

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END;

/

SHOW ERRORS;

-- 5. Procedure: UpdateSalary

CREATE OR REPLACE PROCEDURE UpdateSalary (

p\_emp\_id IN NUMBER,

p\_percent IN NUMBER

) IS

BEGIN

UPDATE Employees

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

WHERE EmployeeID = p\_emp\_id;

IF SQL%ROWCOUNT = 0 THEN

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

END IF;

COMMIT;

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

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END;

/

SHOW ERRORS;

-- 6. Procedure: AddNewCustomer

CREATE OR REPLACE PROCEDURE AddNewCustomer (

p\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_dob IN DATE,

p\_balance IN NUMBER

) IS

BEGIN

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

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

COMMIT;

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

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

WHEN OTHERS THEN

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

END;

/

SHOW ERRORS;

-- 7. Call and test all three procedures

BEGIN

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

SafeTransferFunds(1, 2, 100); -- should succeed

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

UpdateSalary(1, 10); -- should succeed

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

AddNewCustomer(3, 'Sam Wilson', TO\_DATE('1995-01-01', 'YYYY-MM-DD'), 2000); -- should succeed

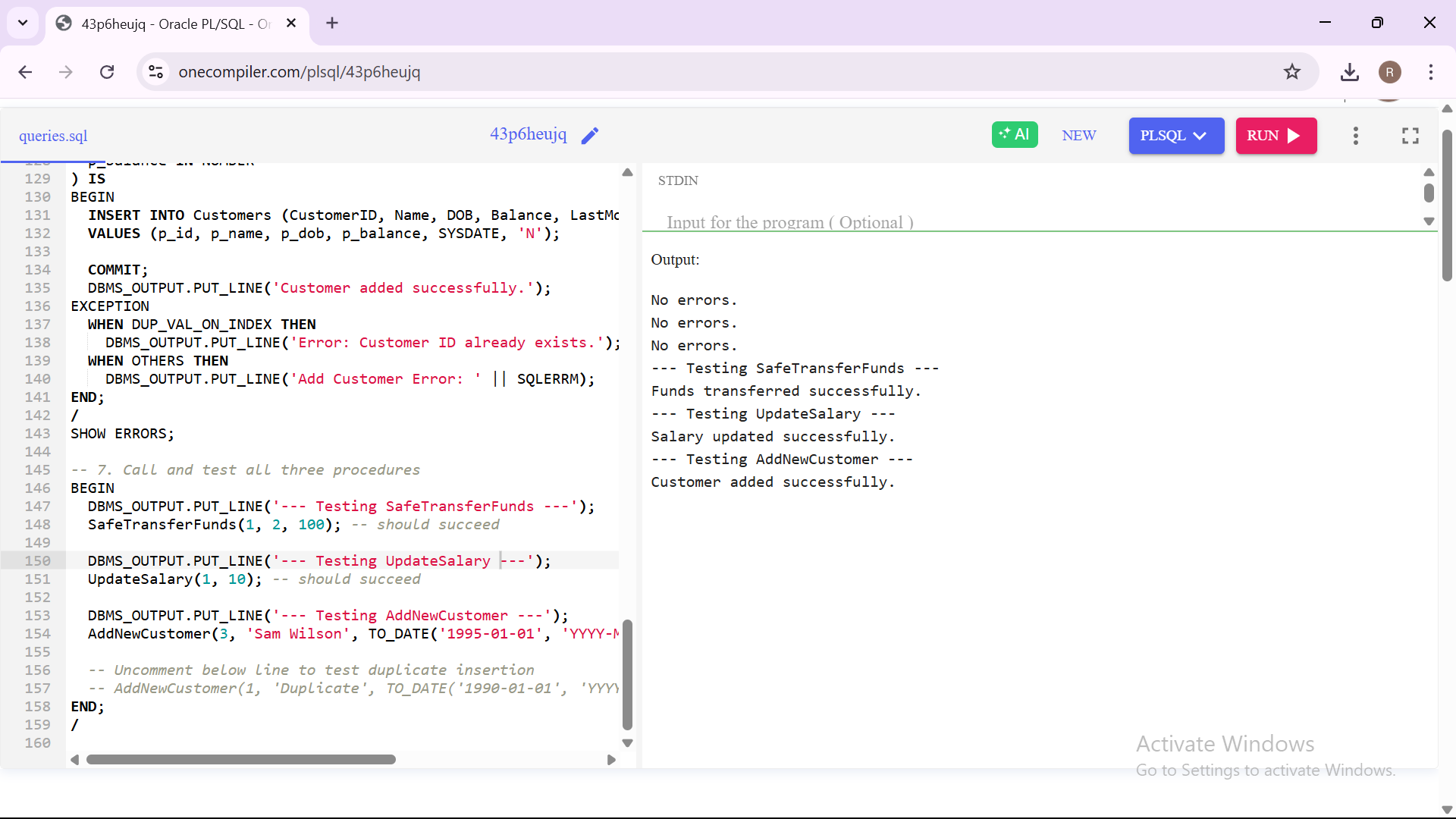
-- Uncomment below line to test duplicate insertion

-- AddNewCustomer(1, 'Duplicate', TO\_DATE('1990-01-01', 'YYYY-MM-DD'), 5000);

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.

**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.

**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.

**Source Code:**

**-- 1. Drop and recreate Accounts and Employees tables (if needed)**

**BEGIN**

**EXECUTE IMMEDIATE 'DROP TABLE Accounts';**

**EXCEPTION**

**WHEN OTHERS THEN NULL;**

**END;**

**/**

**BEGIN**

**EXECUTE IMMEDIATE 'DROP TABLE Employees';**

**EXCEPTION**

**WHEN OTHERS THEN NULL;**

**END;**

**/**

**-- 2. Recreate required tables**

**-- Accounts table**

**CREATE TABLE Accounts (**

**AccountID NUMBER PRIMARY KEY,**

**CustomerID NUMBER,**

**AccountType VARCHAR2(20),**

**Balance NUMBER,**

**LastModified DATE**

**);**

**-- Employees table**

**CREATE TABLE Employees (**

**EmployeeID NUMBER PRIMARY KEY,**

**Name VARCHAR2(100),**

**Position VARCHAR2(50),**

**Salary NUMBER,**

**Department VARCHAR2(50),**

**HireDate DATE**

**);**

**-- 3. Insert sample data**

**INSERT INTO Accounts VALUES (1, 1, 'Savings', 5000, SYSDATE);**

**INSERT INTO Accounts VALUES (2, 2, 'Savings', 3000, SYSDATE);**

**INSERT INTO Accounts VALUES (3, 1, 'Current', 1000, SYSDATE);**

**INSERT INTO Employees VALUES (1, 'Alice', 'Manager', 70000, 'HR', SYSDATE);**

**INSERT INTO Employees VALUES (2, 'Bob', 'Developer', 60000, 'IT', SYSDATE);**

**INSERT INTO Employees VALUES (3, 'Charlie', 'Tester', 50000, 'IT', SYSDATE);**

**COMMIT;**

**/**

**-- 4. Procedure: ProcessMonthlyInterest (adds 1% interest to Savings accounts)**

**CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS**

**BEGIN**

**UPDATE Accounts**

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

**LastModified = SYSDATE**

**WHERE AccountType = 'Savings';**

**COMMIT;**

**DBMS\_OUTPUT.PUT\_LINE('Monthly interest applied to savings accounts.');**

**END;**

**/**

**SHOW ERRORS;**

**-- 5. Procedure: UpdateEmployeeBonus (adds bonus % to employees in a dept)**

**CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (**

**p\_department IN VARCHAR2,**

**p\_bonus\_percent IN NUMBER**

**) IS**

**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 || ' employees in ' || p\_department || ' department.');**

**END IF;**

**COMMIT;**

**END;**

**/**

**SHOW ERRORS;**

**-- 6. Procedure: TransferFunds (from one account to another with balance check)**

**CREATE OR REPLACE PROCEDURE TransferFunds (**

**p\_from\_account IN NUMBER,**

**p\_to\_account IN NUMBER,**

**p\_amount IN NUMBER**

**) IS**

**v\_balance NUMBER;**

**BEGIN**

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

**IF v\_balance < p\_amount THEN**

**RAISE\_APPLICATION\_ERROR(-20010, 'Insufficient balance in source account.');**

**END IF;**

**UPDATE Accounts SET Balance = Balance - p\_amount, LastModified = SYSDATE WHERE AccountID = p\_from\_account;**

**UPDATE Accounts SET Balance = Balance + p\_amount, LastModified = SYSDATE WHERE AccountID = p\_to\_account;**

**COMMIT;**

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

**EXCEPTION**

**WHEN OTHERS THEN**

**ROLLBACK;**

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

**END;**

**/**

**SHOW ERRORS;**

**-- 7. Test all procedures**

**BEGIN**

**DBMS\_OUTPUT.PUT\_LINE('--- Testing ProcessMonthlyInterest ---');**

**ProcessMonthlyInterest;**

**DBMS\_OUTPUT.PUT\_LINE('--- Testing UpdateEmployeeBonus ---');**

**UpdateEmployeeBonus('IT', 10); -- 10% bonus for IT dept**

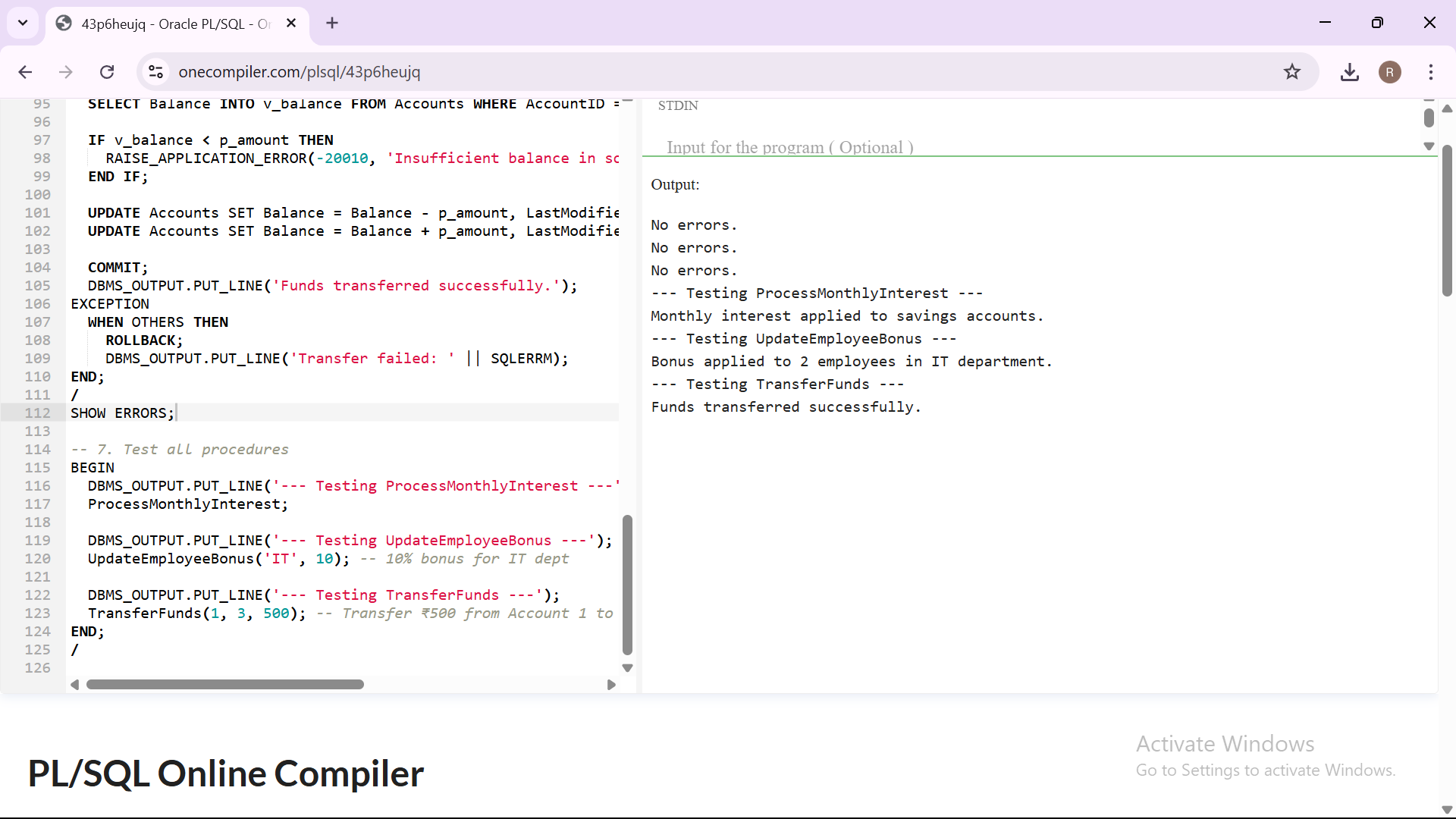
**DBMS\_OUTPUT.PUT\_LINE('--- Testing TransferFunds ---');**

**TransferFunds(1, 3, 500); -- Transfer ₹500 from Account 1 to 3**

**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.

**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.

**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.

**Source Code:**

-- Drop tables (run manually if needed)

-- DROP TABLE Accounts;

-- DROP TABLE Customers;

-- Create Customers table

CREATE TABLE Customers (

CustomerID NUMBER PRIMARY KEY,

Name VARCHAR2(100),

DOB DATE,

Balance NUMBER

);

-- Create Accounts table

CREATE TABLE Accounts (

AccountID NUMBER PRIMARY KEY,

CustomerID NUMBER,

AccountType VARCHAR2(20),

Balance NUMBER,

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

-- Insert sample data

INSERT INTO Customers VALUES (1, 'John Doe', TO\_DATE('1987-06-15','YYYY-MM-DD'), 1000);

INSERT INTO Customers VALUES (2, 'Jane Smith', TO\_DATE('1990-07-20','YYYY-MM-DD'), 1500);

INSERT INTO Accounts VALUES (1, 1, 'Savings', 2000);

INSERT INTO Accounts VALUES (2, 2, 'Checking', 800);

COMMIT;

-- Function 1: Calculate Age

CREATE OR REPLACE FUNCTION CalculateAge(p\_dob DATE) RETURN NUMBER IS

v\_age NUMBER;

BEGIN

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

RETURN v\_age;

END;

/

-- Function 2: Calculate Monthly Installment

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(

p\_amount NUMBER,

p\_rate NUMBER,

p\_years NUMBER

) RETURN NUMBER IS

v\_monthly NUMBER;

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

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

BEGIN

v\_monthly := p\_amount \* v\_monthly\_rate / (1 - POWER(1 + v\_monthly\_rate, -v\_months));

RETURN ROUND(v\_monthly, 2);

END;

/

-- Function 3: Check Sufficient Balance

CREATE OR REPLACE FUNCTION HasSufficientBalance(

p\_account\_id NUMBER,

p\_amount NUMBER

) RETURN BOOLEAN IS

v\_balance NUMBER;

BEGIN

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

RETURN v\_balance >= p\_amount;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN FALSE;

END;

/

-- Test block

SET SERVEROUTPUT ON;

BEGIN

-- Test: Calculate Age

DBMS\_OUTPUT.PUT\_LINE('Age: ' || CalculateAge(TO\_DATE('1987-06-15','YYYY-MM-DD')));

-- Test: Monthly Installment

DBMS\_OUTPUT.PUT\_LINE('Monthly Installment: ' || CalculateMonthlyInstallment(100000, 12, 5));

-- Test: Sufficient Balance

IF HasSufficientBalance(1, 1500) THEN

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

ELSE

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

END IF;

IF HasSufficientBalance(2, 1500) THEN

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

ELSE

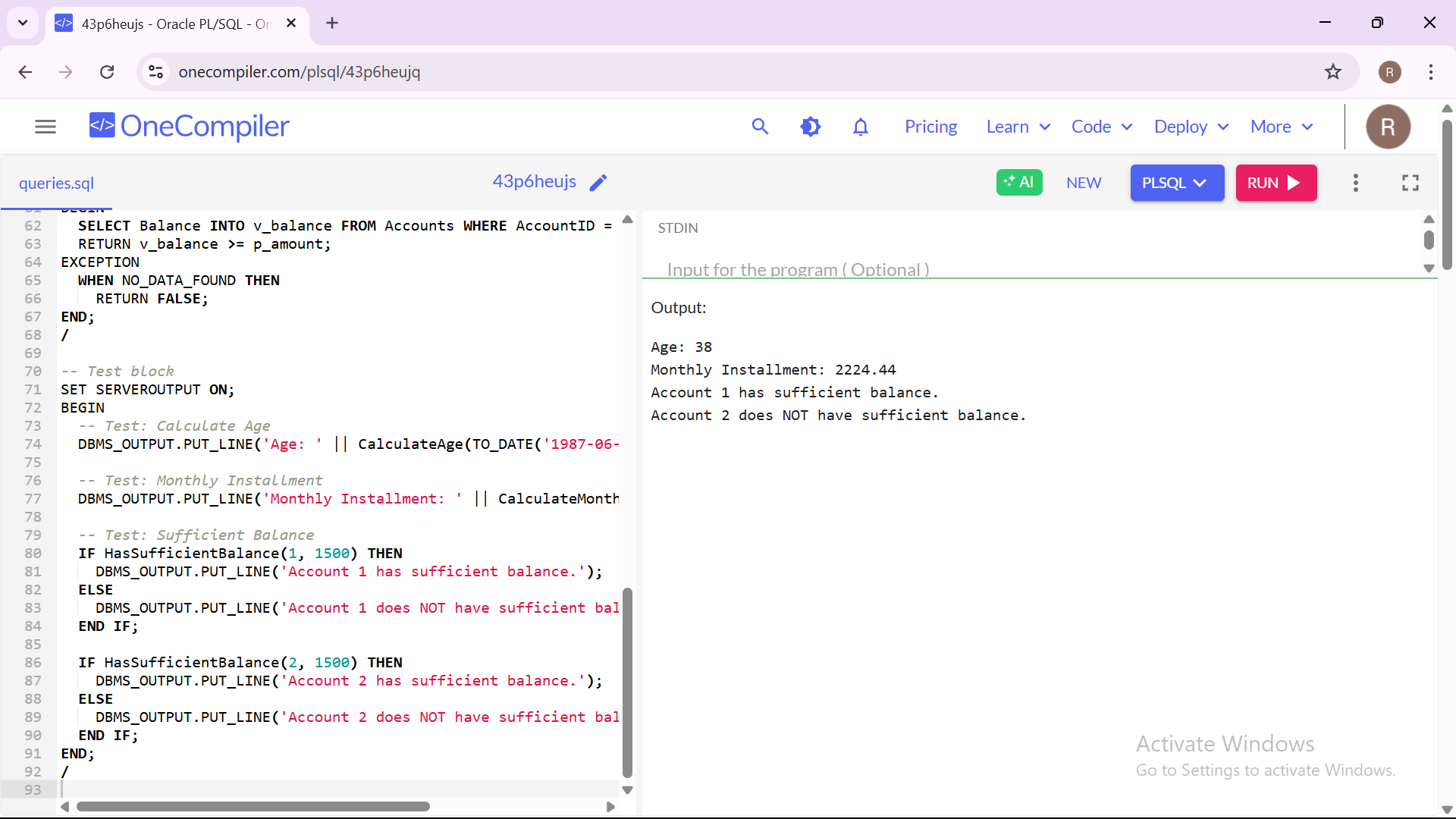
DBMS\_OUTPUT.PUT\_LINE('Account 2 does NOT have sufficient balance.');

END IF;

END;

/

**Output:**



**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.

**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.

**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.

**Source Code:**

-- Drop tables if needed (optional, remove BEGIN if your IDE doesn't support anonymous blocks)

-- DROP TABLE Customers;

-- DROP TABLE Transactions;

-- DROP TABLE AuditLog;

-- 1. Create Customers Table

CREATE TABLE Customers (

CustomerID NUMBER PRIMARY KEY,

Name VARCHAR2(100),

DOB DATE,

Balance NUMBER,

LastModified DATE

);

-- 2. Create Transactions Table

CREATE TABLE Transactions (

TransactionID NUMBER PRIMARY KEY,

AccountID NUMBER,

TransactionDate DATE,

Amount NUMBER,

TransactionType VARCHAR2(10)

);

-- 3. Create AuditLog Table

CREATE TABLE AuditLog (

LogID NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,

TransactionID NUMBER,

LogMessage VARCHAR2(255),

LogTime DATE DEFAULT SYSDATE

);

-- 4. Insert Sample Customer

INSERT INTO Customers VALUES (1, 'John Doe', TO\_DATE('1980-01-01','YYYY-MM-DD'), 5000, SYSDATE);

COMMIT;

--------------------------------------------------------------------------------

-- Scenario 1: Trigger to update LastModified column when customer is updated

--------------------------------------------------------------------------------

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON Customers

FOR EACH ROW

BEGIN

:NEW.LastModified := SYSDATE;

END;

/

--------------------------------------------------------------------------------

-- Scenario 2: Trigger to log transactions into AuditLog

--------------------------------------------------------------------------------

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog (TransactionID, LogMessage)

VALUES (:NEW.TransactionID,

'Transaction of type ' || :NEW.TransactionType ||

' for amount ' || :NEW.Amount ||

' inserted at ' || TO\_CHAR(SYSDATE, 'DD-MON-YYYY HH24:MI:SS'));

END;

/

--------------------------------------------------------------------------------

-- Scenario 3: Enforce deposit/withdrawal rules

--------------------------------------------------------------------------------

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

DECLARE

v\_balance NUMBER;

BEGIN

-- Withdrawals should not exceed balance

IF :NEW.TransactionType = 'Withdrawal' THEN

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

IF :NEW.Amount > v\_balance THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient balance for withdrawal.');

END IF;

ELSIF :NEW.TransactionType = 'Deposit' THEN

IF :NEW.Amount <= 0 THEN

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

END IF;

END IF;

END;

/

--------------------------------------------------------------------------------

-- Testing

--------------------------------------------------------------------------------

SET SERVEROUTPUT ON;

-- 1. Update customer to test trigger UpdateCustomerLastModified

UPDATE Customers SET Name = 'John D. Doe' WHERE CustomerID = 1;

-- 2. Insert valid deposit transaction

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

VALUES (101, 1, SYSDATE, 1000, 'Deposit');

-- 3. Insert valid withdrawal

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

VALUES (102, 1, SYSDATE, 500, 'Withdrawal');

-- 4. Attempt invalid deposit

-- Uncomment this to test failure

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

-- VALUES (103, 1, SYSDATE, -100, 'Deposit');

-- 5. Attempt over-withdrawal

-- Uncomment this to test failure

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

-- VALUES (104, 1, SYSDATE, 999999, 'Withdrawal');

COMMIT;

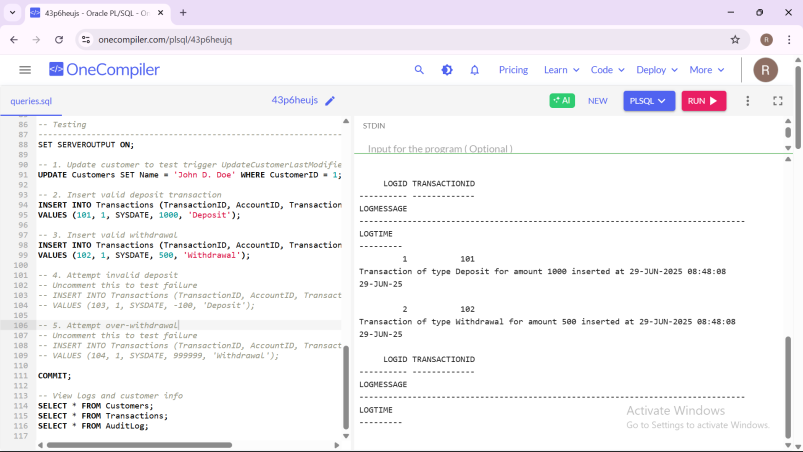
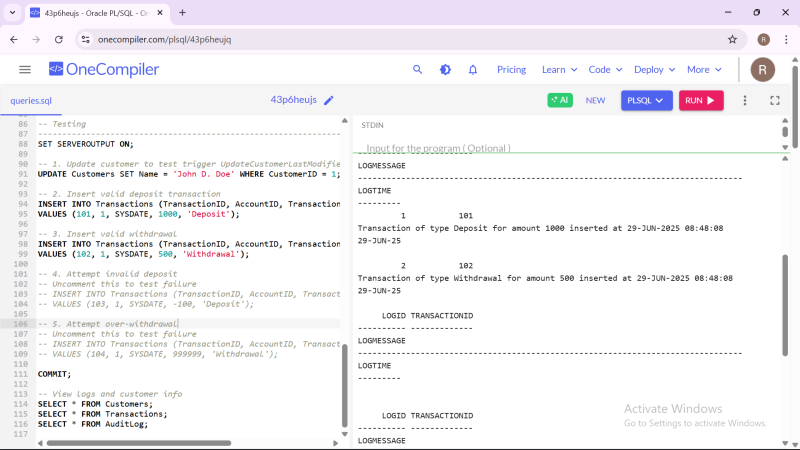
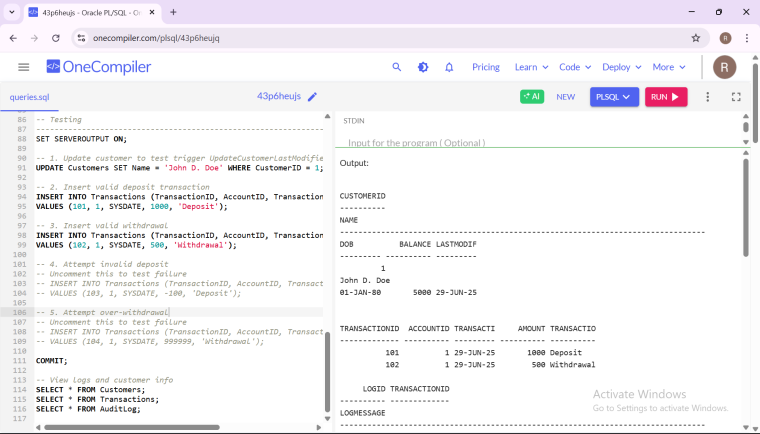
-- View logs and customer info

SELECT \* FROM Customers;

SELECT \* FROM Transactions;

SELECT \* FROM AuditLog;

**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.

**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.

**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.

**Source Code:**

-- Create necessary tables (if not already created)

CREATE TABLE Customers (

CustomerID NUMBER PRIMARY KEY,

Name VARCHAR2(100),

DOB DATE,

Balance NUMBER

);

CREATE TABLE Accounts (

AccountID NUMBER PRIMARY KEY,

CustomerID NUMBER,

AccountType VARCHAR2(20),

Balance NUMBER,

LastModified DATE,

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

CREATE TABLE Transactions (

TransactionID NUMBER PRIMARY KEY,

AccountID NUMBER,

TransactionDate DATE,

Amount NUMBER,

TransactionType VARCHAR2(10),

FOREIGN KEY (AccountID) REFERENCES Accounts(AccountID)

);

CREATE TABLE Loans (

LoanID NUMBER PRIMARY KEY,

CustomerID NUMBER,

LoanAmount NUMBER,

InterestRate NUMBER,

StartDate DATE,

EndDate DATE,

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

-- Sample data

INSERT INTO Customers VALUES (1, 'Alice', TO\_DATE('1985-06-15', 'YYYY-MM-DD'), 5000);

INSERT INTO Customers VALUES (2, 'Bob', TO\_DATE('1990-08-20', 'YYYY-MM-DD'), 8000);

INSERT INTO Accounts VALUES (1, 1, 'Savings', 5000, SYSDATE);

INSERT INTO Accounts VALUES (2, 2, 'Checking', 8000, SYSDATE);

INSERT INTO Transactions VALUES (201, 1, SYSDATE, 1000, 'Deposit');

INSERT INTO Transactions VALUES (202, 1, SYSDATE, 200, 'Withdrawal');

INSERT INTO Transactions VALUES (203, 2, SYSDATE, 500, 'Deposit');

INSERT INTO Loans VALUES (301, 1, 10000, 7.5, SYSDATE, ADD\_MONTHS(SYSDATE, 60));

INSERT INTO Loans VALUES (302, 2, 15000, 8.0, SYSDATE, ADD\_MONTHS(SYSDATE, 72));

COMMIT;

--------------------------------------------------------------------------------

-- Scenario 1: Generate monthly statements for all customers

--------------------------------------------------------------------------------

DECLARE

CURSOR txn\_cursor IS

SELECT c.CustomerID, c.Name, t.TransactionDate, t.Amount, t.TransactionType

FROM Customers c

JOIN Accounts a ON c.CustomerID = a.CustomerID

JOIN Transactions t ON a.AccountID = t.AccountID

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

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

BEGIN

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

FOR rec IN txn\_cursor LOOP

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

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

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

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

END LOOP;

END;

/

--------------------------------------------------------------------------------

-- Scenario 2: Apply annual fee to all accounts

--------------------------------------------------------------------------------

DECLARE

CURSOR acc\_cursor IS

SELECT AccountID, Balance FROM Accounts;

annual\_fee CONSTANT NUMBER := 200;

BEGIN

FOR rec IN acc\_cursor LOOP

UPDATE Accounts

SET Balance = Balance - annual\_fee

WHERE AccountID = rec.AccountID;

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('Annual fee of ' || annual\_fee || ' applied to all accounts.');

COMMIT;

END;

/

--------------------------------------------------------------------------------

-- Scenario 3: Update interest rate based on new policy

--------------------------------------------------------------------------------

DECLARE

CURSOR loan\_cursor IS

SELECT LoanID, InterestRate FROM Loans;

BEGIN

FOR rec IN loan\_cursor LOOP

UPDATE Loans

SET InterestRate = CASE

WHEN InterestRate >= 8 THEN InterestRate - 0.5

WHEN InterestRate < 8 THEN InterestRate + 0.25

END

WHERE LoanID = rec.LoanID;

END LOOP;

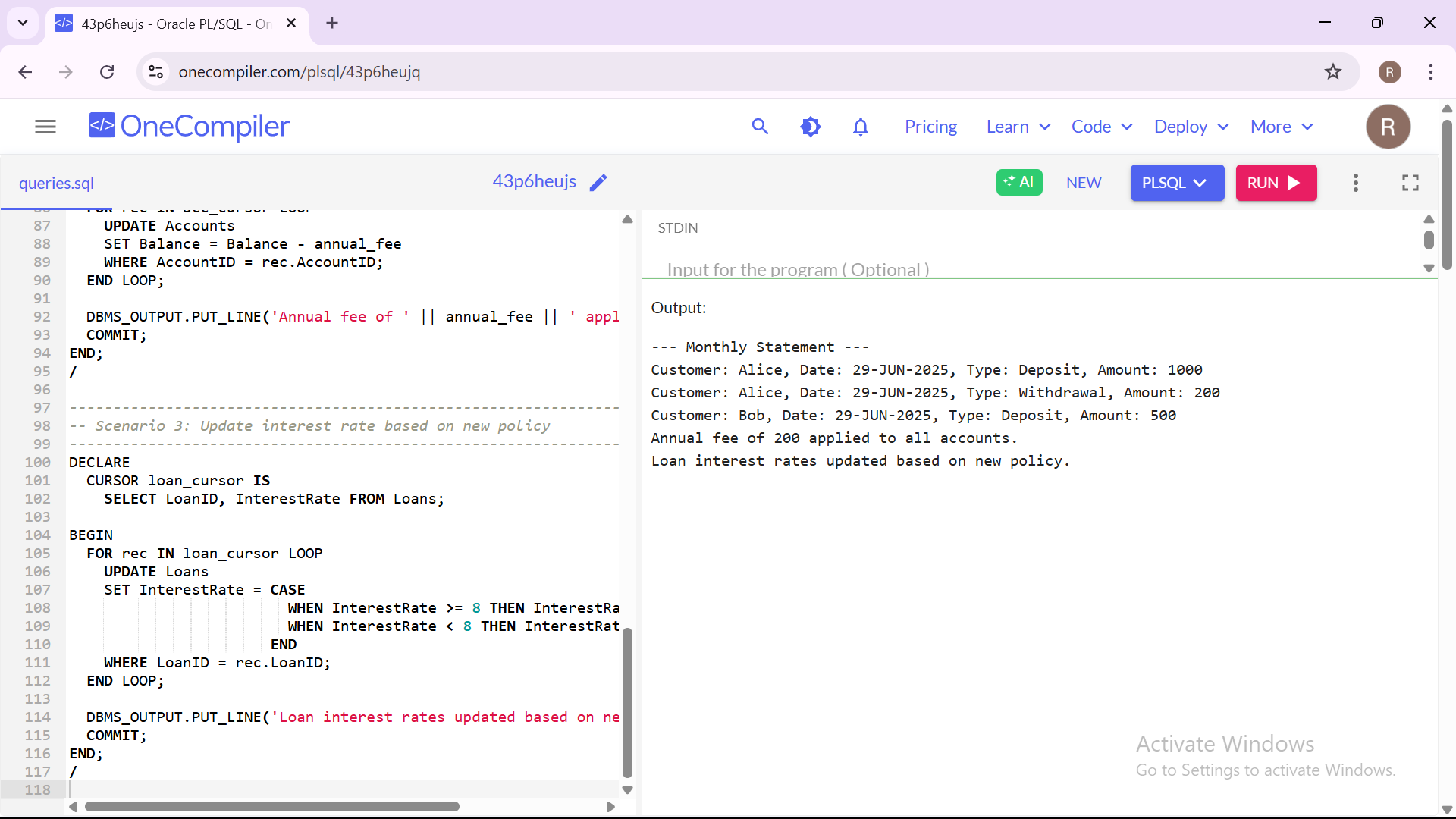
DBMS\_OUTPUT.PUT\_LINE('Loan interest rates updated based on new policy.');

COMMIT;

END;

/

**Output:**

****

**Exercise 7: Implementing the Observer Pattern**

**Scenario:**

You are developing a stock market monitoring application where multiple clients need to be notified whenever stock prices change. Use the Observer Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **ObserverPatternExample**.
2. **Define Subject Interface:**
   * Create an interface **Stock** with methods to **register**, **deregister**, and **notify** observers.
3. **Implement Concrete Subject:**
   * Create a class **StockMarket** that implements **Stock** and maintains a list of observers.
4. **Define Observer Interface:**
   * Create an interface Observer with a method **update().**
5. **Implement Concrete Observers:**
   * Create classes **MobileApp**, **WebApp** that implement Observer.
6. **Test the Observer Implementation:**
   * Create a test class to demonstrate the registration and notification of observers.

**Source Code:**

**-- Create Customers table**

**CREATE TABLE Customers\_EX7 (**

**CustomerID NUMBER PRIMARY KEY,**

**Name VARCHAR2(100),**

**DOB DATE,**

**Balance NUMBER**

**);**

**-- Create Employees table**

**CREATE TABLE Employees\_EX7 (**

**EmployeeID NUMBER PRIMARY KEY,**

**Name VARCHAR2(100),**

**Department VARCHAR2(50),**

**Salary NUMBER**

**);**

**-- Create Accounts table**

**CREATE TABLE Accounts\_EX7 (**

**AccountID NUMBER PRIMARY KEY,**

**CustomerID NUMBER,**

**Balance NUMBER**

**);**

**-- Insert sample data**

**INSERT INTO Customers\_EX7 VALUES (1, 'John Doe', TO\_DATE('1990-01-01', 'YYYY-MM-DD'), 4000);**

**INSERT INTO Employees\_EX7 VALUES (1, 'Alice Smith', 'HR', 60000);**

**INSERT INTO Accounts\_EX7 VALUES (1, 1, 4000);**

**INSERT INTO Accounts\_EX7 VALUES (2, 1, 3000);**

**COMMIT;**

**-- Package: CustomerManagement\_EX7**

**CREATE OR REPLACE PACKAGE CustomerManagement\_EX7 AS**

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

**PROCEDURE UpdateCustomerBalance(p\_id NUMBER, p\_amount NUMBER);**

**FUNCTION GetCustomerBalance(p\_id NUMBER) RETURN NUMBER;**

**END CustomerManagement\_EX7;**

**/**

**CREATE OR REPLACE PACKAGE BODY CustomerManagement\_EX7 AS**

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

**BEGIN**

**INSERT INTO Customers\_EX7 VALUES (p\_id, p\_name, p\_dob, p\_balance);**

**END;**

**PROCEDURE UpdateCustomerBalance(p\_id NUMBER, p\_amount NUMBER) IS**

**BEGIN**

**UPDATE Customers\_EX7 SET Balance = Balance + p\_amount WHERE CustomerID = p\_id;**

**END;**

**FUNCTION GetCustomerBalance(p\_id NUMBER) RETURN NUMBER IS**

**v\_balance NUMBER;**

**BEGIN**

**SELECT Balance INTO v\_balance FROM Customers\_EX7 WHERE CustomerID = p\_id;**

**RETURN v\_balance;**

**END;**

**END CustomerManagement\_EX7;**

**/**

**-- Package: EmployeeManagement\_EX7**

**CREATE OR REPLACE PACKAGE EmployeeManagement\_EX7 AS**

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

**PROCEDURE UpdateSalary(p\_id NUMBER, p\_new\_salary NUMBER);**

**FUNCTION CalculateAnnualSalary(p\_id NUMBER) RETURN NUMBER;**

**END EmployeeManagement\_EX7;**

**/**

**CREATE OR REPLACE PACKAGE BODY EmployeeManagement\_EX7 AS**

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

**BEGIN**

**INSERT INTO Employees\_EX7 VALUES (p\_id, p\_name, p\_dept, p\_salary);**

**END;**

**PROCEDURE UpdateSalary(p\_id NUMBER, p\_new\_salary NUMBER) IS**

**BEGIN**

**UPDATE Employees\_EX7 SET Salary = p\_new\_salary WHERE EmployeeID = p\_id;**

**END;**

**FUNCTION CalculateAnnualSalary(p\_id NUMBER) RETURN NUMBER IS**

**v\_salary NUMBER;**

**BEGIN**

**SELECT Salary INTO v\_salary FROM Employees\_EX7 WHERE EmployeeID = p\_id;**

**RETURN v\_salary \* 12;**

**END;**

**END EmployeeManagement\_EX7;**

**/**

**-- Package: AccountOperations\_EX7**

**CREATE OR REPLACE PACKAGE AccountOperations\_EX7 AS**

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

**PROCEDURE CloseAccount(p\_accid NUMBER);**

**FUNCTION GetTotalBalance(p\_custid NUMBER) RETURN NUMBER;**

**END AccountOperations\_EX7;**

**/**

**CREATE OR REPLACE PACKAGE BODY AccountOperations\_EX7 AS**

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

**BEGIN**

**INSERT INTO Accounts\_EX7 VALUES (p\_accid, p\_custid, p\_balance);**

**END;**

**PROCEDURE CloseAccount(p\_accid NUMBER) IS**

**BEGIN**

**DELETE FROM Accounts\_EX7 WHERE AccountID = p\_accid;**

**END;**

**FUNCTION GetTotalBalance(p\_custid NUMBER) RETURN NUMBER IS**

**v\_total NUMBER;**

**BEGIN**

**SELECT NVL(SUM(Balance), 0) INTO v\_total FROM Accounts\_EX7 WHERE CustomerID = p\_custid;**

**RETURN v\_total;**

**END;**

**END AccountOperations\_EX7;**

**/**

**-- Testing the packages**

**DECLARE**

**v\_balance NUMBER;**

**v\_salary NUMBER;**

**v\_total NUMBER;**

**BEGIN**

**v\_balance := CustomerManagement\_EX7.GetCustomerBalance(1);**

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

**v\_salary := EmployeeManagement\_EX7.CalculateAnnualSalary(1);**

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

**v\_total := AccountOperations\_EX7.GetTotalBalance(1);**

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

**END;**

**/**

**Output:**

