COGNIZANT

Digital Nurture 4.0

Deep Skilling - Java FSE

WEEK-2 HANDS ON

By Kaviya P

**PL/SQL**

**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.

**QUERY:**

**->FOR TABLE CREATION**

-- Step 1: Create Customers table

CREATE TABLE Customers (

    CustomerID NUMBER PRIMARY KEY,

    Name VARCHAR2(100),

    DOB DATE,

    Balance NUMBER,

    LastModified DATE

);

-- Step 2: 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)

);

-- Step 3: Insert sample customers

-- Customer 3: Age 72

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

VALUES (3, 'Robert King', TO\_DATE('1952-04-10', 'YYYY-MM-DD'), 6000, SYSDATE);

-- Customer 4: Age 68

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

VALUES (4, 'Susan Lee', TO\_DATE('1956-09-25', 'YYYY-MM-DD'), 8500, SYSDATE);

-- Customer 5: Age 58

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

VALUES (5, 'Tom Hardy', TO\_DATE('1966-02-14', 'YYYY-MM-DD'), 9200, SYSDATE);

-- Customer 6: Age 43

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

VALUES (6, 'Priya Sharma', TO\_DATE('1981-11-30', 'YYYY-MM-DD'), 4800, SYSDATE);

-- Customer 7: Age 61

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

VALUES (7, 'Naveen Reddy', TO\_DATE('1963-06-05', 'YYYY-MM-DD'), 10400, SYSDATE);

-- Step 4: Insert sample loans

-- Loan for Customer 3

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

VALUES (3, 3, 8000, 6.5, SYSDATE, ADD\_MONTHS(SYSDATE, 48));

-- Loan for Customer 4

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

VALUES (4, 4, 12000, 7.2, SYSDATE, ADD\_MONTHS(SYSDATE, 60));

-- Loan for Customer 5

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

VALUES (5, 5, 15000, 8, SYSDATE, ADD\_MONTHS(SYSDATE, 60));

-- Loan for Customer 6

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

VALUES (6, 6, 9500, 7.8, SYSDATE, ADD\_MONTHS(SYSDATE, 36));

-- Loan for Customer 7

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

VALUES (7, 7, 11000, 6.9, SYSDATE, ADD\_MONTHS(SYSDATE, 48));

**QUERY :**

-- Scenario 1: Apply 1% discount to loan interest for customers above 60

BEGIN

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

    IF TRUNC(MONTHS\_BETWEEN(SYSDATE, i.DOB)/12) > 60 THEN

      UPDATE Loans

      SET InterestRate = InterestRate - 1

      WHERE CustomerID = i.CustomerID;

      DBMS\_OUTPUT.PUT\_LINE('1% discount applied to CustomerID: ' || i.CustomerID);

    END IF;

  END LOOP;

END;

/

-- Scenario 2: Set IsVIP flag

alter table Customers add IsVIP BOOLEAN;

begin

  for i in (select \* from Customers) loop

    if i.Balance > 10000 then

      update Customers set IsVIP = TRUE where CustomerID = i.CustomerID;

      DBMS\_OUTPUT.PUT\_LINE('BALANCE GREATER THAN 10000: FOR CustomerID ' || i.CustomerID);

    end if;

  end loop;

end;

/

-- Scenario 3: Loan reminders

begin

  for rec in (select \* from Loans where EndDate <= SYSDATE + 30) loop

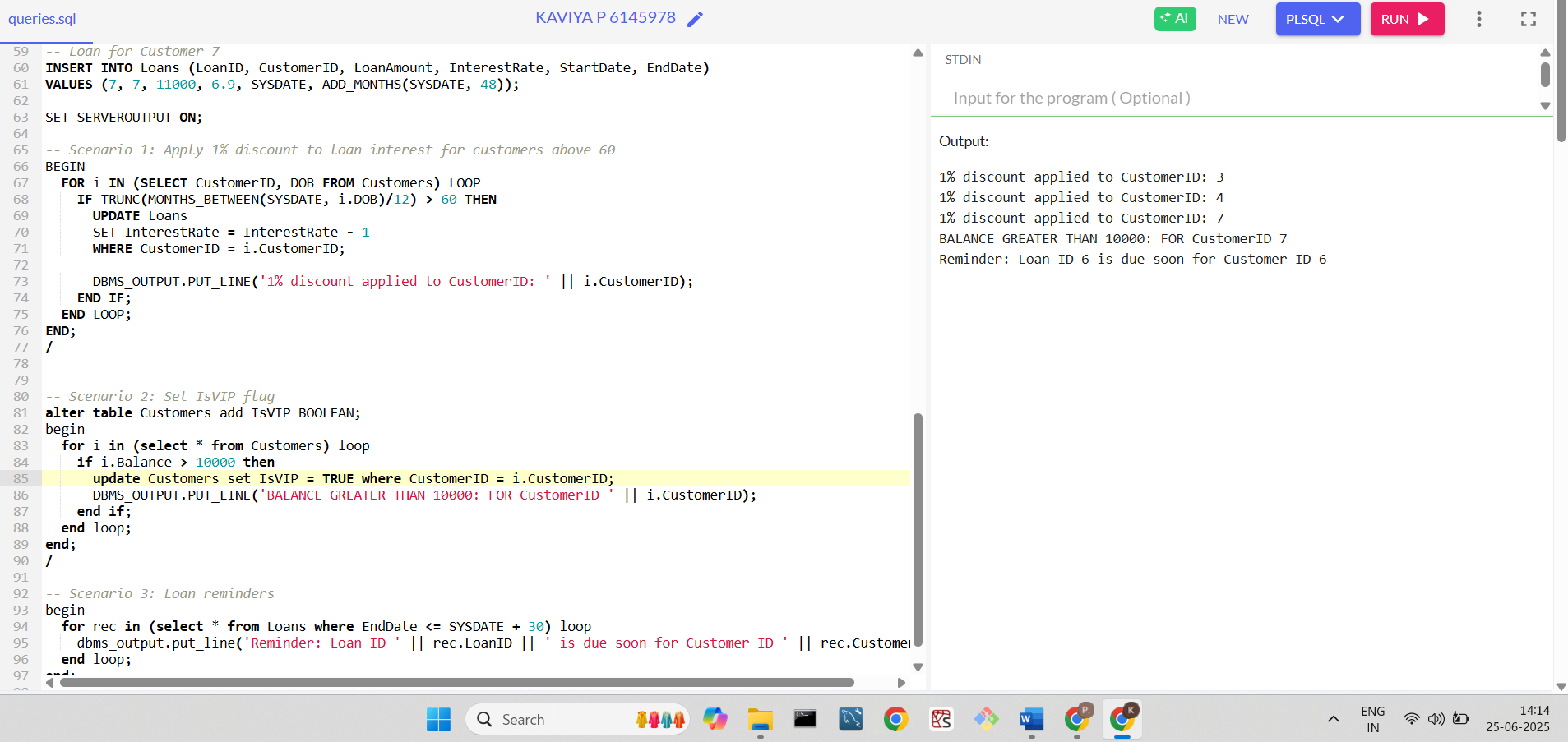
    dbms\_output.put\_line('Reminder: Loan ID ' || rec.LoanID || ' is due soon for Customer ID ' || rec.CustomerID);

  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.

**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.

**->FOR TABLE CREATION**

CREATE TABLE Accounts (

    AccountID NUMBER PRIMARY KEY,

    CustomerID NUMBER,

    AccountType VARCHAR2(20),

    Balance NUMBER,

    LastModified DATE

);

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

INSERT INTO Accounts VALUES (102, 2, 'Checking', 3000, SYSDATE);

INSERT INTO Accounts VALUES (103, 3, 'Savings', 1000, SYSDATE);

CREATE TABLE Employees (

    EmployeeID NUMBER PRIMARY KEY,

    Name VARCHAR2(100),

    Position VARCHAR2(50),

    Salary NUMBER,

    Department VARCHAR2(50),

    HireDate DATE

);

CREATE TABLE Customers (

    CustomerID NUMBER PRIMARY KEY,

    Name VARCHAR2(100),

    DOB DATE,

    Balance NUMBER,

    LastModified DATE

);

INSERT INTO Employees VALUES (1, 'Alice', 'Manager', 70000, 'HR', TO\_DATE('2015-06-10', 'YYYY-MM-DD'));

INSERT INTO Employees VALUES (2, 'Bob', 'Developer', 60000, 'IT', TO\_DATE('2016-03-20', 'YYYY-MM-DD'));

INSERT INTO Customers VALUES (1, 'John Doe', TO\_DATE('1980-04-12', 'YYYY-MM-DD'), 6000, SYSDATE);

INSERT INTO Customers VALUES (2, 'Jane Smith', TO\_DATE('1975-06-25', 'YYYY-MM-DD'), 8000, SYSDATE);

**QUERY :**

--SCENARIO 1

CREATE OR REPLACE PROCEDURE SafeTransferFunds(p\_from NUMBER, p\_to NUMBER, p\_amount NUMBER) IS

  v\_balance NUMBER;

BEGIN

  -- Get source account balance

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

  -- Check if balance is sufficient

  IF v\_balance < p\_amount THEN

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

  END IF;

  -- Perform transfer

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

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

  COMMIT;

EXCEPTION

  WHEN OTHERS THEN

    ROLLBACK;

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

END;

/

--SCENARIO 2

CREATE OR REPLACE PROCEDURE UpdateSalary(p\_emp\_id NUMBER, p\_percent NUMBER) IS

  v\_count NUMBER;

BEGIN

  -- Check if employee exists

  SELECT COUNT(\*) INTO v\_count FROM Employees WHERE EmployeeID = p\_emp\_id;

  IF v\_count = 0 THEN

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

  END IF;

  -- Update salary

  UPDATE Employees

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

  WHERE EmployeeID = p\_emp\_id;

  COMMIT;

EXCEPTION

  WHEN OTHERS THEN

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

END;

/

--SCENARIO 3

CREATE OR REPLACE PROCEDURE AddNewCustomer(

  p\_id NUMBER,

  p\_name VARCHAR2,

  p\_dob DATE,

  p\_balance NUMBER

) IS

BEGIN

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

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

  COMMIT;

EXCEPTION

  WHEN DUP\_VAL\_ON\_INDEX THEN

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

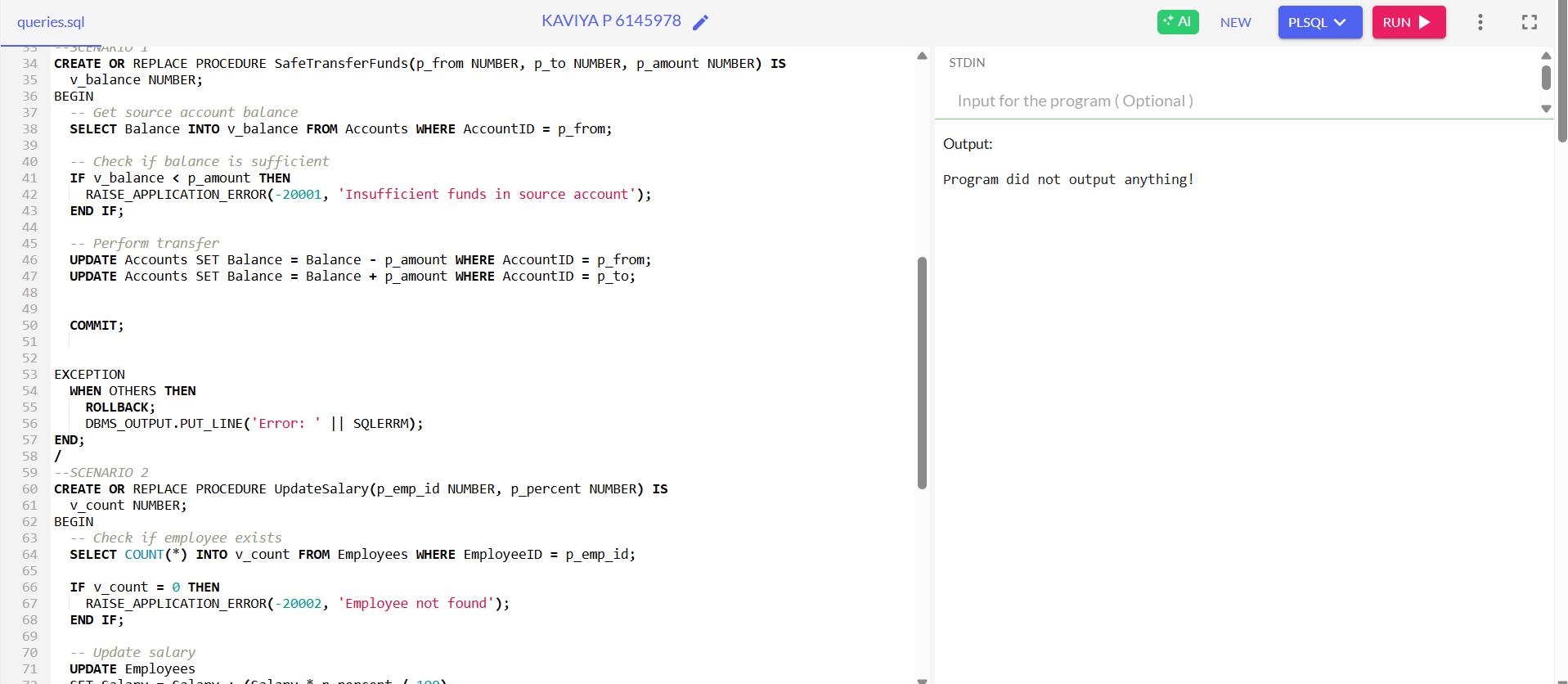
  WHEN OTHERS THEN

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

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.

-- Scenario 1: Monthly interest

create or replace procedure ProcessMonthlyInterest is

begin

  update Accounts set Balance = Balance + (Balance \* 0.01) where AccountType = 'Savings';

end;

/

-- Scenario 2: Bonus for employees

create or replace procedure UpdateEmployeeBonus(p\_dept varchar2, p\_bonus\_percent number) is

begin

  update Employees set Salary = Salary + (Salary \* p\_bonus\_percent / 100) where Department = p\_dept;

end;

/

-- Scenario 3: Transfer funds between accounts

create or replace procedure TransferFunds(p\_from number, p\_to number, p\_amount number) is

  v\_balance number;

begin

  select Balance into v\_balance from Accounts where AccountID = p\_from;

  if v\_balance < p\_amount then

    raise\_application\_error(-20003, 'Insufficient funds');

  end if;

  update Accounts set Balance = Balance - p\_amount where AccountID = p\_from;

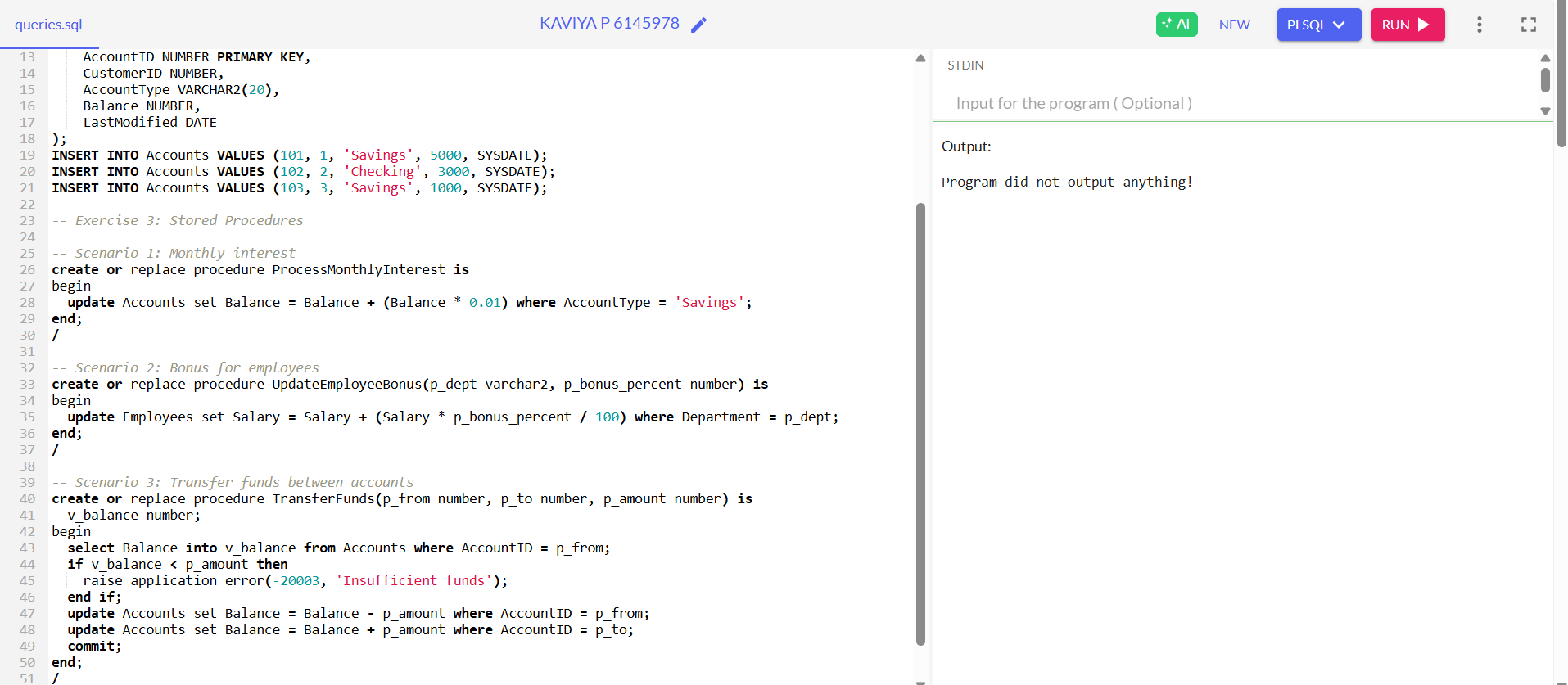
  update Accounts set Balance = Balance + p\_amount where AccountID = p\_to;

  commit;

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.

-- Scenario 1: Calculate age

create or replace function CalculateAge(p\_dob date) return number is

begin

  return trunc(months\_between(sysdate, p\_dob)/12);

end;

/

-- Scenario 2: Monthly installment

create or replace function CalculateMonthlyInstallment(p\_amount number, p\_rate number, p\_years number) return number is

  v\_months number := p\_years \* 12;

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

begin

  return (p\_amount \* v\_monthly\_rate) / (1 - power(1 + v\_monthly\_rate, -v\_months));

end;

/

-- Scenario 3: Check sufficient balance

create or replace function HasSufficientBalance(p\_accID number, p\_amount number) return boolean is

  v\_balance number;

begin

  select Balance into v\_balance from Accounts where AccountID = p\_accID;

  return v\_balance >= p\_amount;

end;

-- to call function CalculateAge

DECLARE

  v\_age NUMBER;

BEGIN

  v\_age := CalculateAge(DATE '2000-06-25'); -- Pass your DOB here

  DBMS\_OUTPUT.PUT\_LINE('Age is: ' || v\_age);

END;

/

-- call function CalculateMonthlyInstallment

DECLARE

  v\_emi NUMBER;

BEGIN

  v\_emi := CalculateMonthlyInstallment(100000, 7.5, 5); -- 1 lakh loan, 7.5% interest, 5 years

  DBMS\_OUTPUT.PUT\_LINE('Monthly EMI is: ' || ROUND(v\_emi, 2));

END;

/

-- calucate function HasSufficientBalance

DECLARE

  v\_result BOOLEAN;

BEGIN

  v\_result := HasSufficientBalance(101, 2000); -- Account 101 has ₹5000

  IF v\_result THEN

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

  ELSE

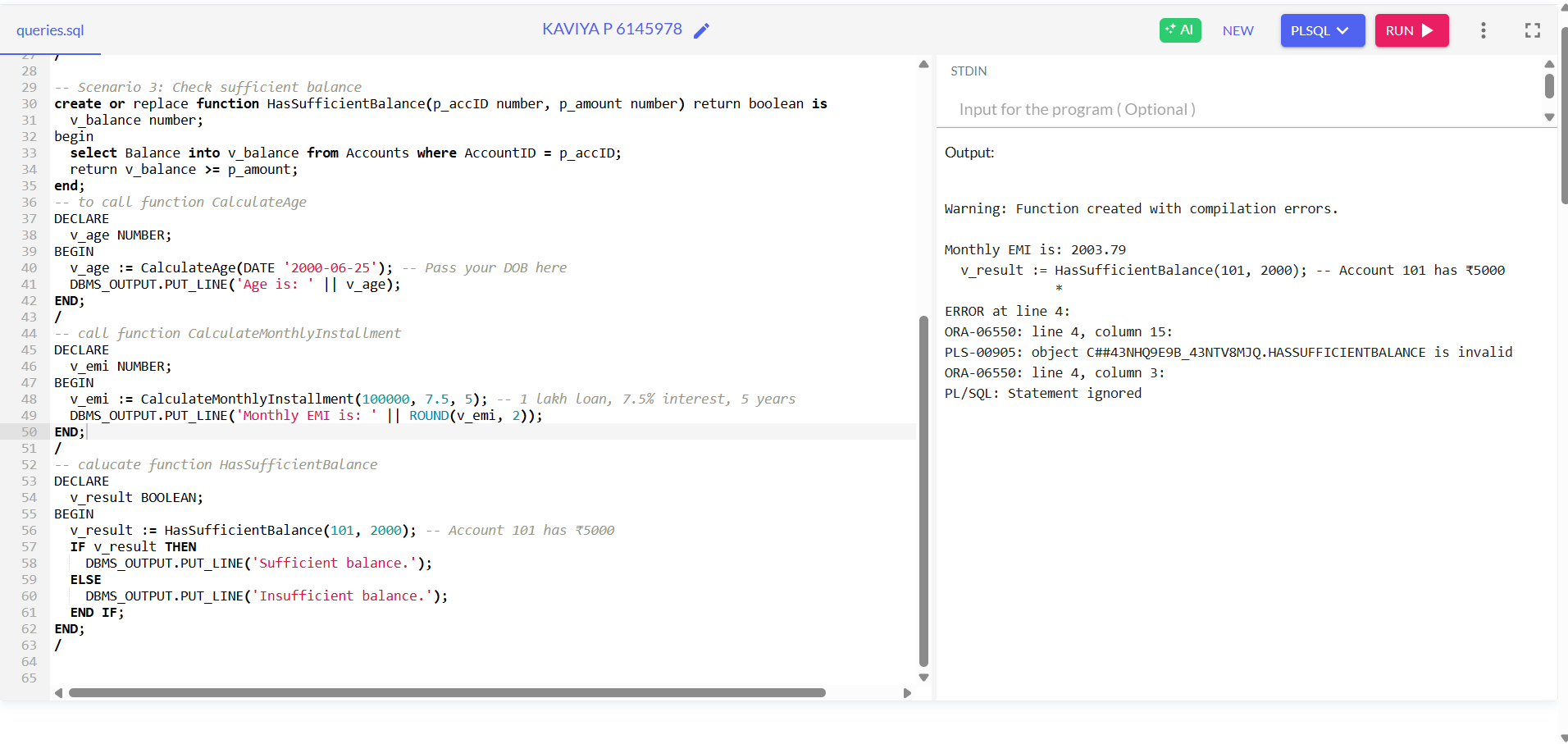
    DBMS\_OUTPUT.PUT\_LINE('Insufficient balance.');

  END IF;

END;

/

**OUTPUT:**

****

**Exercise 5: Triggers**

**Scenario 1:** Automatically update the last modified date when a customer's record is updated.

* + **Question:** Write a trigger **UpdateCustomerLastModified** that updates the LastModified column of the Customers table to the current date whenever a customer's record is updated.

**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.

- Exercise 5: Triggers

-- Scenario 1: Update LastModified on update

create or replace trigger UpdateCustomerLastModified

before update on Customers

for each row

begin

  :new.LastModified := sysdate;

end;

/

-- Scenario 2: Log transactions

create table AuditLog (

  LogID number primary key,

  TransactionID number,

  LogDate date default sysdate,

  Action varchar2(100)

);

create sequence AuditLog\_seq start with 1 increment by 1;

create or replace trigger LogTransaction

after insert on Transactions

for each row

begin

  insert into AuditLog (LogID, TransactionID, Action)

  values (AuditLog\_seq.nextval, :new.TransactionID, 'Transaction inserted');

end;

/

-- Scenario 3: Transaction rules

create or replace trigger CheckTransactionRules

before insert on Transactions

for each row

declare

  v\_balance number;

begin

  select Balance into v\_balance from Accounts where AccountID = :new.AccountID;

  if :new.TransactionType = 'Withdrawal' and :new.Amount > v\_balance then

    raise\_application\_error(-20004, 'Withdrawal exceeds balance');

  elsif :new.TransactionType = 'Deposit' and :new.Amount <= 0 then

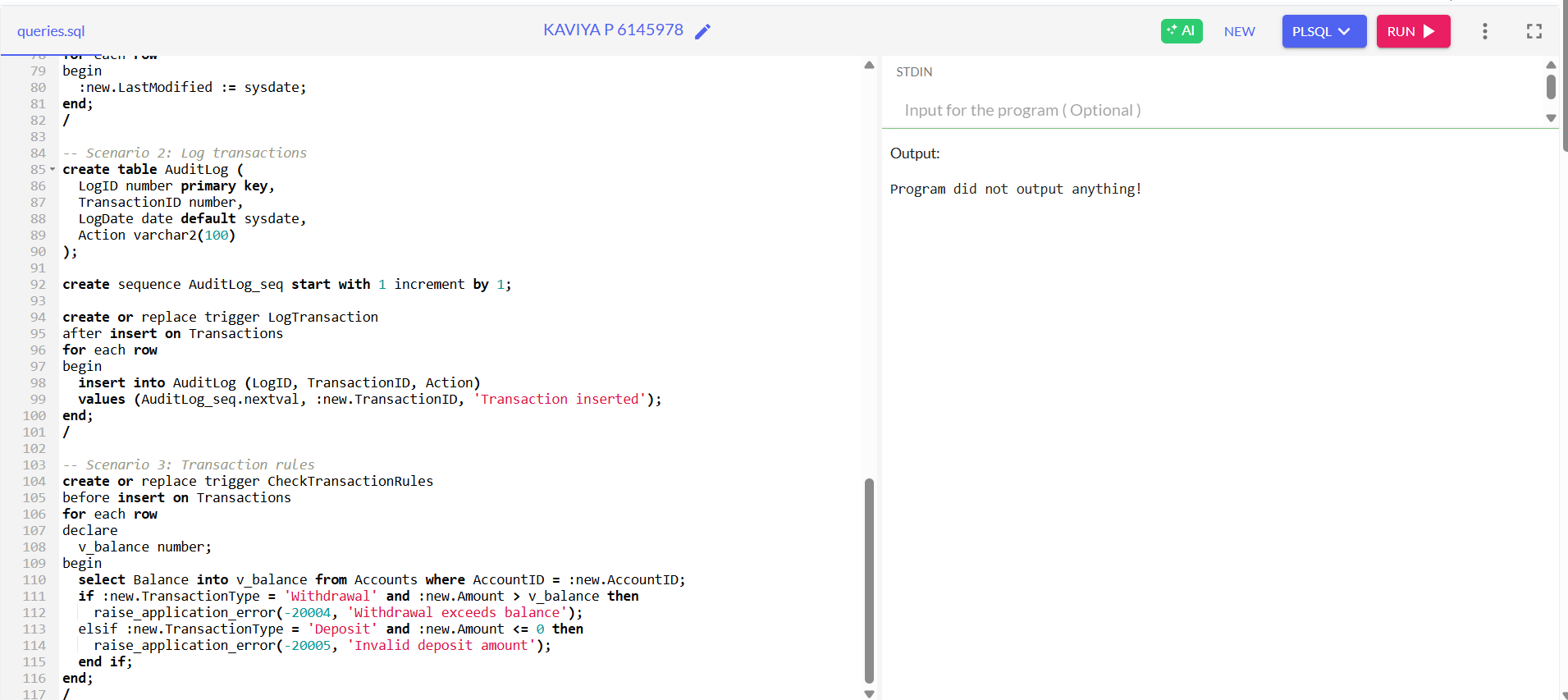
    raise\_application\_error(-20005, 'Invalid deposit amount');

  end if;

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.

**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.

-- Create Transactions table

CREATE TABLE Transactions (

    TransactionID NUMBER PRIMARY KEY,

    AccountID NUMBER REFERENCES Accounts(AccountID),

    TransactionDate DATE,

    Amount NUMBER,

    TransactionType VARCHAR2(10)

);

-- Sample transactions

INSERT INTO Transactions VALUES (1, 101, SYSDATE, 200, 'Deposit');

INSERT INTO Transactions VALUES (2, 102, SYSDATE, 300, 'Withdrawal');

-- Scenario 1: Generate statements

declare

  cursor c\_statements is

    select \* from Transactions where trunc(TransactionDate, 'MM') = trunc(sysdate, 'MM');

begin

  for rec in c\_statements loop

    dbms\_output.put\_line('Account: ' || rec.AccountID || ', Amount: ' || rec.Amount || ', Type: ' || rec.TransactionType);

  end loop;

end;

/

-- Scenario 2: Apply annual fee

declare

  cursor c\_accounts is select \* from Accounts;

begin

  for acc in c\_accounts loop

    update Accounts set Balance = Balance - 100 where AccountID = acc.AccountID;

  end loop;

end;

/

-- Scenario 3: Update loan interest

declare

  cursor c\_loans is select \* from Loans;

begin

  for l in c\_loans loop

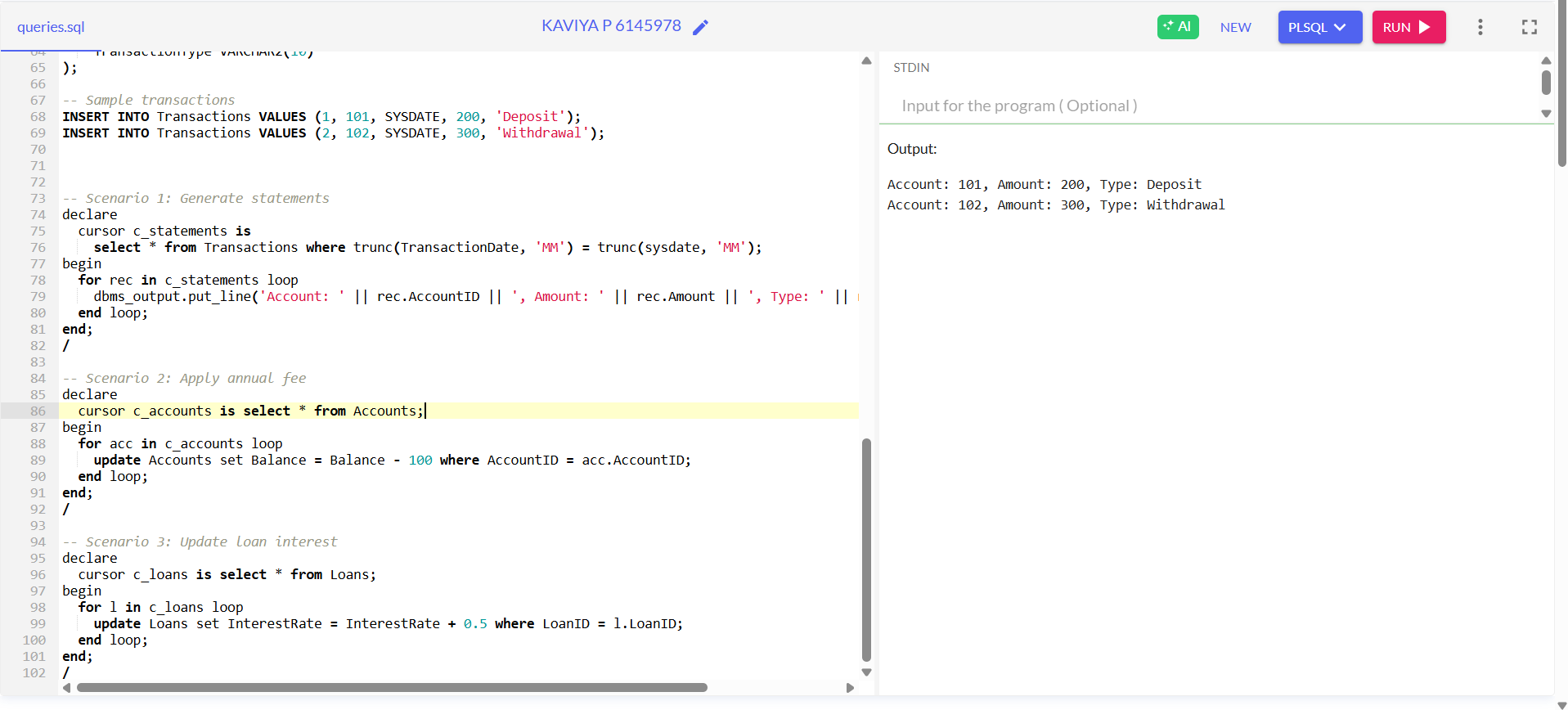
    update Loans set InterestRate = InterestRate + 0.5 where LoanID = l.LoanID;

  end loop;

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.

**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.

**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.

- Scenario 1: CustomerManagement package

CREATE OR REPLACE PACKAGE CustomerManagement IS

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

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

  FUNCTION GetCustomerBalance(p\_id NUMBER) RETURN NUMBER;

END;

/

CREATE OR REPLACE PACKAGE BODY CustomerManagement IS

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

  BEGIN

    INSERT INTO Customers (CustomerID, Name, LastModified)

    VALUES (p\_id, p\_name, SYSDATE);

  END;

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

  BEGIN

    UPDATE Customers

    SET Name = p\_name,

        LastModified = SYSDATE

    WHERE CustomerID = p\_id;

  END;

  FUNCTION GetCustomerBalance(p\_id NUMBER) RETURN NUMBER IS

    v\_balance NUMBER;

  BEGIN

    SELECT SUM(Balance)

    INTO v\_balance

    FROM Accounts

    WHERE CustomerID = p\_id;

    RETURN NVL(v\_balance, 0);

  END;

END;

/

-- Scenario 2: EmployeeManagement package

CREATE OR REPLACE PACKAGE EmployeeManagement IS

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

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

  FUNCTION CalculateAnnualSalary(p\_id NUMBER) RETURN NUMBER;

END;

/

CREATE OR REPLACE PACKAGE BODY EmployeeManagement IS

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

  BEGIN

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

  END;

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

  BEGIN

    UPDATE Employees

    SET Name = p\_name

    WHERE EmployeeID = p\_id;

  END;

  FUNCTION CalculateAnnualSalary(p\_id NUMBER) RETURN NUMBER IS

    v\_salary NUMBER;

  BEGIN

    SELECT Salary

    INTO v\_salary

    FROM Employees

    WHERE EmployeeID = p\_id;

    RETURN v\_salary \* 12;

  END;

END;

/

-- Scenario 3: AccountOperations package

CREATE OR REPLACE PACKAGE EmployeeManagement IS

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

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

  FUNCTION CalculateAnnualSalary(p\_id NUMBER) RETURN NUMBER;

END;

/

CREATE OR REPLACE PACKAGE BODY EmployeeManagement IS

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

  BEGIN

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

  END;

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

  BEGIN

    UPDATE Employees

    SET Name = p\_name

    WHERE EmployeeID = p\_id;

  END;

  FUNCTION CalculateAnnualSalary(p\_id NUMBER) RETURN NUMBER IS

    v\_salary NUMBER;

  BEGIN

    SELECT Salary

    INTO v\_salary

    FROM Employees

    WHERE EmployeeID = p\_id;

    RETURN v\_salary \* 12;

  END;

END;

/

**OUTPUT:**

