**PL/SQL PROGRAMMING**

**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.

**QUERY:**

BEGIN

FOR rec IN (

SELECT l.LoanID, c.DOB

FROM Loans l

JOIN Customers c ON l.CustomerID = c.CustomerID

) LOOP

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

UPDATE Loans SET InterestRate = InterestRate - 1

WHERE LoanID = rec.LoanID;

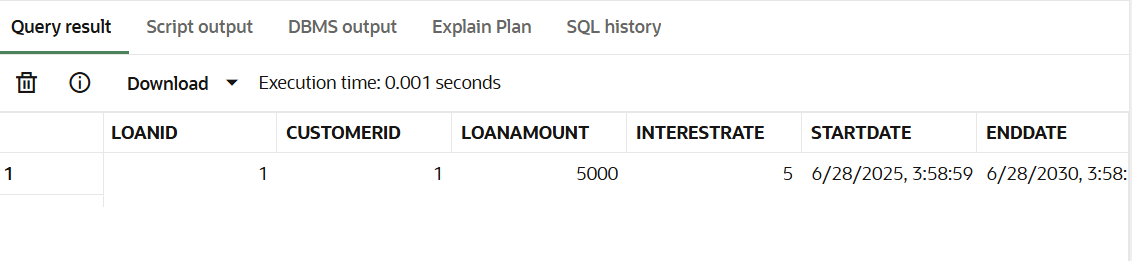
END IF;

END LOOP;

END;

/

**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.

**QUERY**

ALTER TABLE Customers ADD IsVIP CHAR(1);

BEGIN

FOR rec IN (SELECT \* FROM Customers) LOOP

IF rec.Balance > 10000 THEN

UPDATE Customers SET IsVIP = 'Y' WHERE CustomerID = rec.CustomerID;

END IF;

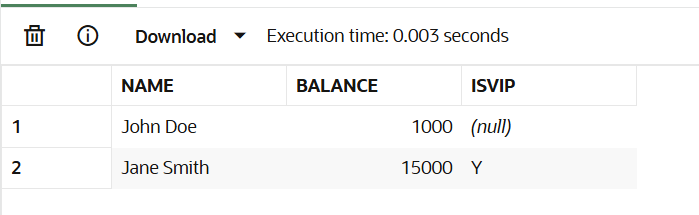
END LOOP;

END;

/

SELECT Name, Balance, IsVIP 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.

**QUERY**

INSERT INTO Loans VALUES (3, 1, 3000, 6, SYSDATE, SYSDATE + 5);

COMMIT;

BEGIN

FOR rec IN (SELECT \* FROM Loans WHERE EndDate <= SYSDATE + 30) LOOP

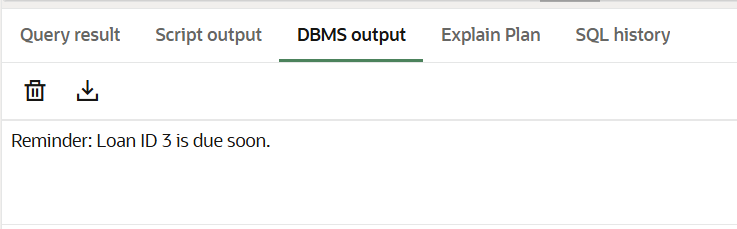
DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan ID ' || rec.LoanID || ' is due soon.');

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.

**QUERY**

CREATE OR REPLACE PROCEDURE SafeTransferFunds(

p\_from\_account\_id IN NUMBER,

p\_to\_account\_id IN NUMBER,

p\_amount IN NUMBER

) AS

v\_balance NUMBER;

BEGIN

-- Lock the source account row for update

SELECT Balance INTO v\_balance FROM Accounts

WHERE AccountID = p\_from\_account\_id FOR UPDATE;

IF v\_balance < p\_amount THEN

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

END IF;

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

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

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

COMMIT;

EXCEPTION

WHEN OTHERS THEN

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

ROLLBACK;

END;

/

BEGIN

SafeTransferFunds(1, 2, 500);

END;

/

**OUTPUT**



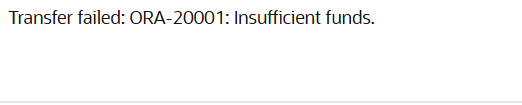
BEGIN

SafeTransferFunds(1, 2, 100000); -- should fail

END;

/

**OUTPUT** (if transfer amount is more than balance)



**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.

**OUERY**

CREATE OR REPLACE PROCEDURE UpdateSalary(

p\_emp\_id IN NUMBER,

p\_percent IN NUMBER

) AS

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 not found.');

ELSE

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

END IF;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

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

ROLLBACK;

END;

/

BEGIN

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

END;

/

**OUTPUT**



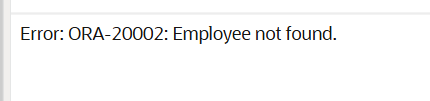
BEGIN

UpdateSalary(99, 5); --INVALID ID

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.

**QUERY**

CREATE OR REPLACE PROCEDURE AddNewCustomer(

  p\_id      IN NUMBER,

  p\_name    IN VARCHAR2,

  p\_dob     IN DATE,

  p\_balance IN NUMBER

) AS

BEGIN

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

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

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

  COMMIT;

EXCEPTION

  WHEN DUP\_VAL\_ON\_INDEX THEN

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

  WHEN OTHERS THEN

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

    ROLLBACK;

END;

/

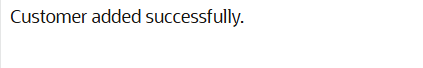
BEGIN

  AddNewCustomer(3, 'New Customer', TO\_DATE('2000-01-01','YYYY-MM-DD'), 2000);

END;

/

**OUTPUT**

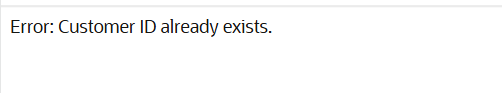


BEGIN

  AddNewCustomer(3, 'Duplicate', TO\_DATE('1999-01-01','YYYY-MM-DD'), 1000); -- Try adding same ID again

END;

/



**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.

**QUERY**

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

BEGIN

  FOR acc IN (

    SELECT AccountID, Balance

    FROM Accounts

    WHERE AccountType = 'Savings'

  ) LOOP

    UPDATE Accounts

    SET Balance = acc.Balance + (acc.Balance \* 0.01)

    WHERE AccountID = acc.AccountID;

    DBMS\_OUTPUT.PUT\_LINE('Interest added to Account ID: ' || acc.AccountID);

  END LOOP;

  COMMIT;

END;

/

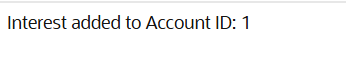
BEGIN

  ProcessMonthlyInterest;

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.

**QUERY**

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(

  p\_department IN VARCHAR2,

  p\_bonus\_pct IN NUMBER

) IS

BEGIN

  FOR emp IN (

    SELECT EmployeeID, Salary FROM Employees WHERE Department = p\_department

  ) LOOP

    UPDATE Employees

    SET Salary = emp.Salary + (emp.Salary \* p\_bonus\_pct / 100)

    WHERE EmployeeID = emp.EmployeeID;

    DBMS\_OUTPUT.PUT\_LINE('Bonus updated for Employee ID: ' || emp.EmployeeID);

  END LOOP;

  COMMIT;

END;

/

BEGIN

  UpdateEmployeeBonus('IT', 5);  -- Apply 5% bonus to IT department

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.

**QUERY**

CREATE OR REPLACE PROCEDURE TransferFunds(

  p\_from\_account\_id IN NUMBER,

  p\_to\_account\_id   IN NUMBER,

  p\_amount          IN NUMBER

) AS

  v\_balance NUMBER;

BEGIN

  -- Get balance of source account

  SELECT Balance INTO v\_balance FROM Accounts

  WHERE AccountID = p\_from\_account\_id FOR UPDATE;

  IF v\_balance < p\_amount THEN

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

    ROLLBACK;

    RETURN;

  END IF;

  -- Perform the fund transfer

  UPDATE Accounts SET Balance = Balance - p\_amount

  WHERE AccountID = p\_from\_account\_id;

  UPDATE Accounts SET Balance = Balance + p\_amount

  WHERE AccountID = p\_to\_account\_id;

  DBMS\_OUTPUT.PUT\_LINE('Transfer of ' || p\_amount || ' completed successfully.');

  COMMIT;

EXCEPTION

  WHEN NO\_DATA\_FOUND THEN

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

    ROLLBACK;

  WHEN OTHERS THEN

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

    ROLLBACK;

END;

/

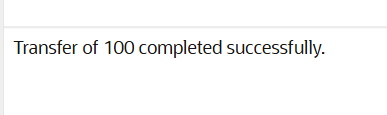
BEGIN

  TransferFunds(1, 2, 100);

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.

**QUERY**

CREATE OR REPLACE FUNCTION CalculateAge(p\_dob DATE)

RETURN NUMBER IS

  v\_age NUMBER;

BEGIN

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

  RETURN v\_age;

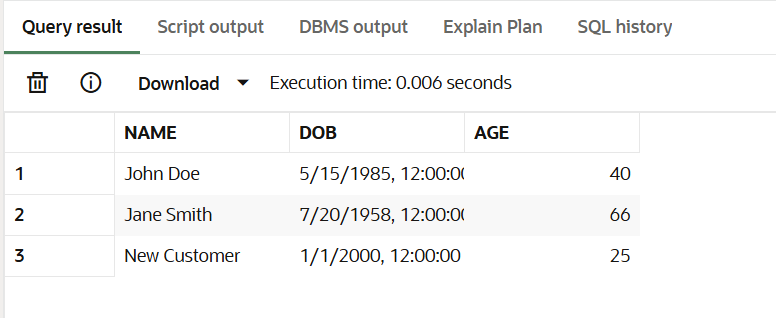
END;

/

SELECT Name, DOB, CalculateAge(DOB) AS Age

FROM Customers;

**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.

**QUERY**

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(

p\_loan\_amount NUMBER,

p\_annual\_rate NUMBER,

p\_duration\_years NUMBER

) RETURN NUMBER IS

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

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

v\_emi NUMBER;

BEGIN

IF v\_monthly\_rate = 0 THEN

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

ELSE

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

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

END IF;

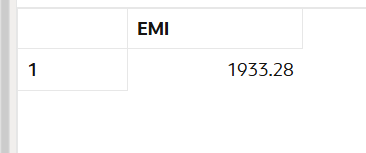
RETURN ROUND(v\_emi, 2);

END;

/

SELECT CalculateMonthlyInstallment(100000, 6, 5) AS EMI FROM dual;

**OUTPUT**



**Scenario 3:** Check if a customer has sufficient balance before making a transaction.

* + **Question:** Write a function **HasSufficientBalance** that takes an account ID and an amount as input and returns a boolean indicating whether the account has at least the specified amount.

**QUERY**

 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;

/

DECLARE

  v\_result BOOLEAN;

BEGIN

  v\_result := HasSufficientBalance(1, 500);  -- Change values as needed

  IF v\_result THEN

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

  ELSE

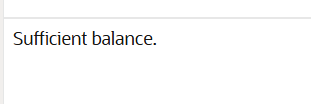
    DBMS\_OUTPUT.PUT\_LINE('Insufficient balance or account not found.');

  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.

**QUERY**

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON Customers

FOR EACH ROW

BEGIN

  :NEW.LastModified := SYSDATE;

END;

/

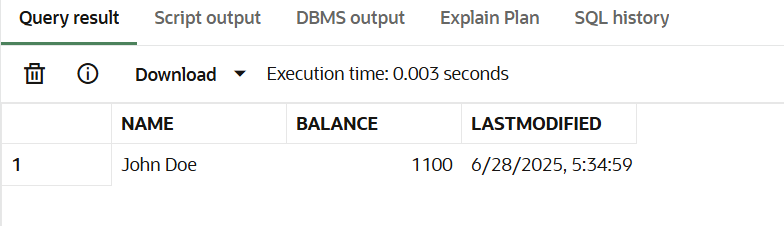
UPDATE Customers

SET Balance = Balance + 100

WHERE CustomerID = 1;

SELECT Name, Balance, LastModified FROM Customers WHERE CustomerID = 1;

**OUTPUT**



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

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

**QUERY**

CREATE TABLE AuditLog (

  LogID NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,

  TransactionID NUMBER,

  AccountID NUMBER,

  Amount NUMBER,

  TransactionType VARCHAR2(10),

  LogDate DATE

);

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

  INSERT INTO AuditLog (TransactionID, AccountID, Amount, TransactionType, LogDate)

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

END;

/

-- Insert a transaction

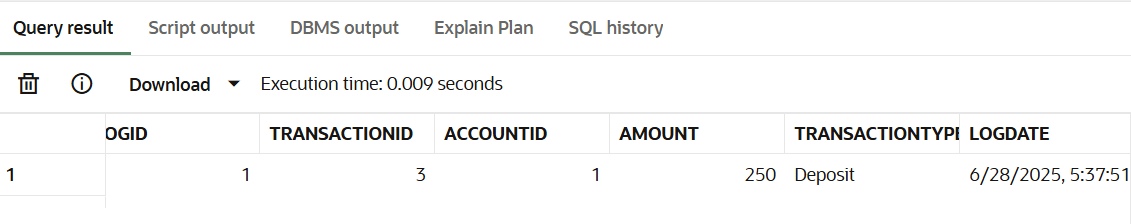
INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)

VALUES (3, 1, SYSDATE, 250, 'Deposit');

-- View audit log

SELECT \* FROM AuditLog WHERE TransactionID = 3;

**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.

**QUERY**

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

DECLARE

  v\_balance NUMBER;

BEGIN

  IF :NEW.TransactionType = 'Withdrawal' THEN

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

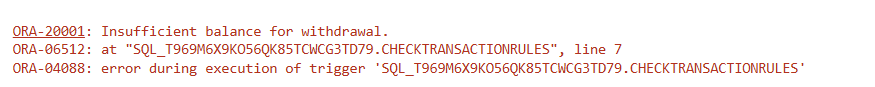
/

-- Account 2 has only 300, try withdrawing 1000

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

VALUES (4, 2, SYSDATE, 1000, 'Withdrawal');

**OUTPUT**

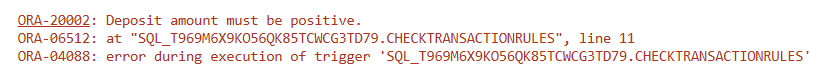


**---INVALID DEPOSIT--**

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

VALUES (5, 1, SYSDATE, -100, 'Deposit');

**OUTPUT**



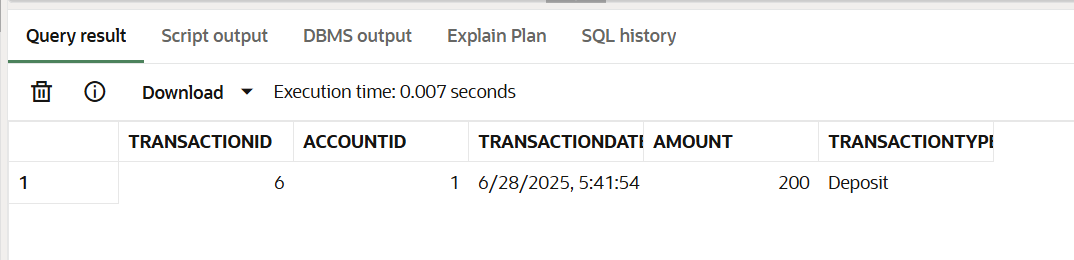
--VALID TRANSACTION—

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

VALUES (6, 1, SYSDATE, 200, 'Deposit');

SELECT \* FROM Transactions WHERE TransactionID = 6;

**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.

**QUERY**

DECLARE

  CURSOR cur\_transactions IS

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

    FROM Transactions t

    JOIN Accounts a ON t.AccountID = a.AccountID

    JOIN Customers c ON a.CustomerID = c.CustomerID

    WHERE TRUNC(t.TransactionDate, 'MM') = TRUNC(SYSDATE, 'MM');

  v\_account\_id Transactions.AccountID%TYPE;

  v\_name Customers.Name%TYPE;

  v\_amount Transactions.Amount%TYPE;

  v\_date Transactions.TransactionDate%TYPE;

  v\_type Transactions.TransactionType%TYPE;

BEGIN

  OPEN cur\_transactions;

  LOOP

    FETCH cur\_transactions INTO v\_account\_id, v\_name, v\_amount, v\_date, v\_type;

    EXIT WHEN cur\_transactions%NOTFOUND;

    DBMS\_OUTPUT.PUT\_LINE('Customer: ' || v\_name ||

                         ' | Account: ' || v\_account\_id ||

                         ' | Date: ' || TO\_CHAR(v\_date, 'DD-MON') ||

                         ' | Amount: ' || v\_amount ||

                         ' | Type: ' || v\_type);

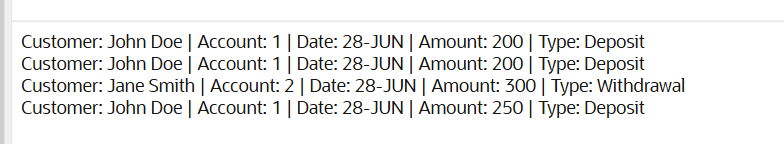
  END LOOP;

  CLOSE cur\_transactions;

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.

**QUERY**

DECLARE

CURSOR cur\_accounts IS

SELECT AccountID, Balance FROM Accounts;

v\_account\_id Accounts.AccountID%TYPE;

v\_balance Accounts.Balance%TYPE;

v\_fee CONSTANT NUMBER := 100;

BEGIN

OPEN cur\_accounts;

LOOP

FETCH cur\_accounts INTO v\_account\_id, v\_balance;

EXIT WHEN cur\_accounts%NOTFOUND;

UPDATE Accounts

SET Balance = Balance - v\_fee

WHERE AccountID = v\_account\_id;

DBMS\_OUTPUT.PUT\_LINE('Annual fee of ' || v\_fee ||

' applied to Account ID: ' || v\_account\_id);

END LOOP;

CLOSE cur\_accounts;

COMMIT;

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.

**QUERY**

DECLARE

CURSOR cur\_loans IS

SELECT LoanID, LoanAmount, InterestRate FROM Loans;

v\_loan\_id Loans.LoanID%TYPE;

v\_amount Loans.LoanAmount%TYPE;

v\_rate Loans.InterestRate%TYPE;

BEGIN

OPEN cur\_loans;

LOOP

FETCH cur\_loans INTO v\_loan\_id, v\_amount, v\_rate;

EXIT WHEN cur\_loans%NOTFOUND;

IF v\_amount > 7000 THEN

UPDATE Loans

SET InterestRate = v\_rate + 0.5

WHERE LoanID = v\_loan\_id;

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

ELSE

DBMS\_OUTPUT.PUT\_LINE('No update needed for Loan ID: ' || v\_loan\_id);

END IF;

END LOOP;

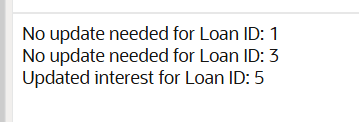
CLOSE cur\_loans;

COMMIT;

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.

**QUERY**

CREATE OR REPLACE PACKAGE CustomerManagement AS

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

  PROCEDURE UpdateCustomer(p\_id NUMBER, p\_balance NUMBER);

  FUNCTION GetCustomerBalance(p\_id NUMBER) RETURN NUMBER;

END CustomerManagement;

/

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

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

  BEGIN

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

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

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

  EXCEPTION

    WHEN DUP\_VAL\_ON\_INDEX THEN

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

  END;

  PROCEDURE UpdateCustomer(p\_id NUMBER, p\_balance NUMBER) IS

  BEGIN

    UPDATE Customers SET Balance = p\_balance, LastModified = SYSDATE WHERE CustomerID = p\_id;

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

  END;

  FUNCTION GetCustomerBalance(p\_id NUMBER) RETURN NUMBER IS

    v\_balance NUMBER;

  BEGIN

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

    RETURN v\_balance;

  EXCEPTION

    WHEN NO\_DATA\_FOUND THEN

      RETURN NULL;

  END;

END CustomerManagement;

/

BEGIN

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

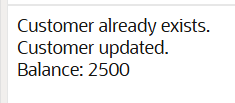
  CustomerManagement.UpdateCustomer(3, 2500);

  DBMS\_OUTPUT.PUT\_LINE('Balance: ' || CustomerManagement.GetCustomerBalance(3));

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.

**QUERY**

CREATE OR REPLACE PACKAGE EmployeeManagement AS

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

  PROCEDURE UpdateEmployee(p\_id NUMBER, p\_salary NUMBER);

  FUNCTION GetAnnualSalary(p\_id NUMBER) RETURN NUMBER;

END EmployeeManagement;

/

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

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

  BEGIN

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

    VALUES (p\_id, p\_name, p\_position, p\_salary, p\_dept, SYSDATE);

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

  END;

  PROCEDURE UpdateEmployee(p\_id NUMBER, p\_salary NUMBER) IS

  BEGIN

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

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

  END;

  FUNCTION GetAnnualSalary(p\_id NUMBER) RETURN NUMBER IS

    v\_salary NUMBER;

  BEGIN

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

    RETURN v\_salary \* 12;

  EXCEPTION

    WHEN NO\_DATA\_FOUND THEN

      RETURN NULL;

  END;

END EmployeeManagement;

/

BEGIN

  EmployeeManagement.HireEmployee(3, 'Ravi', 'Analyst', 50000, 'Finance');

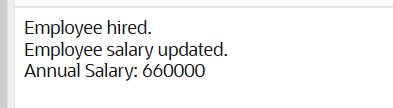
  EmployeeManagement.UpdateEmployee(3, 55000);

  DBMS\_OUTPUT.PUT\_LINE('Annual Salary: ' || EmployeeManagement.GetAnnualSalary(3));

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.

**QUERY**

CREATE OR REPLACE PACKAGE AccountOperations AS

  PROCEDURE OpenAccount(p\_acc\_id NUMBER, p\_cust\_id NUMBER, p\_type VARCHAR2, p\_balance NUMBER);

  PROCEDURE CloseAccount(p\_acc\_id NUMBER);

  FUNCTION GetTotalBalance(p\_cust\_id NUMBER) RETURN NUMBER;

END AccountOperations;

/

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

  PROCEDURE OpenAccount(p\_acc\_id NUMBER, p\_cust\_id NUMBER, p\_type VARCHAR2, p\_balance NUMBER) IS

  BEGIN

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

    VALUES (p\_acc\_id, p\_cust\_id, p\_type, p\_balance, SYSDATE);

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

  END;

  PROCEDURE CloseAccount(p\_acc\_id NUMBER) IS

  BEGIN

    DELETE FROM Accounts WHERE AccountID = p\_acc\_id;

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

  END;

  FUNCTION GetTotalBalance(p\_cust\_id NUMBER) RETURN NUMBER IS

    v\_total NUMBER;

  BEGIN

    SELECT SUM(Balance) INTO v\_total FROM Accounts WHERE CustomerID = p\_cust\_id;

    RETURN NVL(v\_total, 0);

  END;

END AccountOperations;

/

BEGIN

  AccountOperations.OpenAccount(3, 1, 'Savings', 5000);

  AccountOperations.CloseAccount(3);

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

END;

/

**OUTPUT**

