**2. 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.

Inserting values:

Insert into Customers (CustomerID, Name, DOB, Balance, LastModified)

values (99, 'Test Senior', TO\_DATE('1950-01-01','YYYY-MM-DD'), 1000, SYSDATE);

insert into Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate)

values (99, 99, 5000, 5, SYSDATE, ADD\_MONTHS(SYSDATE, 60));

COMMIT;

**Code:**

SET SERVEROUTPUT ON;

BEGIN

FOR rec IN (

SELECT c.CustomerID, c.DOB, l.LoanID, l.InterestRate

FROM Customers c

JOIN Loans l ON c.CustomerID = l.CustomerID

) LOOP

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

UPDATE Loans

SET InterestRate = rec.InterestRate - 1

WHERE LoanID = rec.LoanID;

DBMS\_OUTPUT.PUT\_LINE('Discount applied to LoanID: ' || rec.LoanID);

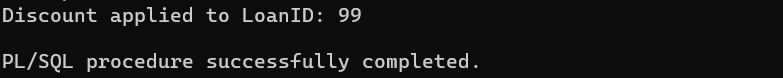
END IF;

END LOOP;

END;

/

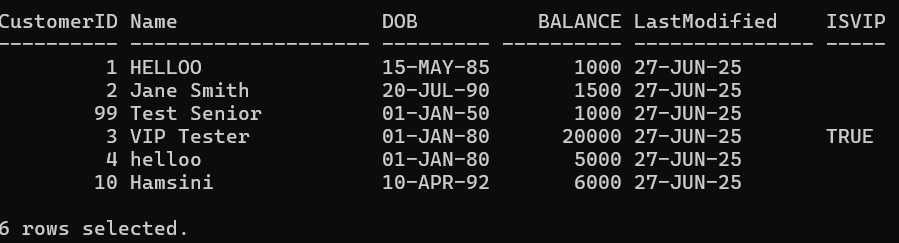
OUTPUT:



**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 Is VIP to TRUE for those with a balance over $10,000.

ALTER TABLE Customers ADD IsVIP VARCHAR2(5);



SET SERVEROUTPUT ON;

BEGIN

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

IF rec.Balance > 10000 THEN

UPDATE Customers

SET IsVIP = 'TRUE'

WHERE CustomerID = rec.CustomerID;

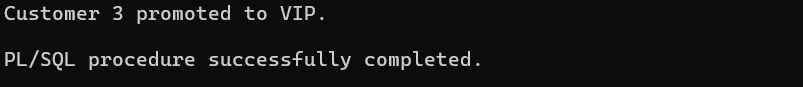
DBMS\_OUTPUT.PUT\_LINE('Customer ' || rec.CustomerID || ' promoted to VIP.');

END IF;

END LOOP;

END;

/



**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.

BEGIN

FOR rec IN (

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

FROM Loans l

JOIN Customers c ON l.CustomerID = c.CustomerID

WHERE l.EndDate BETWEEN SYSDATE AND SYSDATE + 30

) LOOP

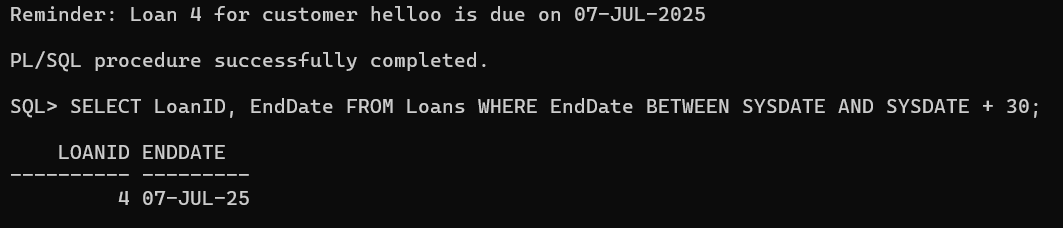
DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan ' || rec.LoanID ||' for customer ' || rec.Name ||' is due on ' || TO\_CHAR(rec.EndDate, 'DD-MON-YYYY'));

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.

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('Transfer of ' || p\_amount || ' from Account 'p\_from\_account || ' to Account ' || p\_to\_account || ' completed.');

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

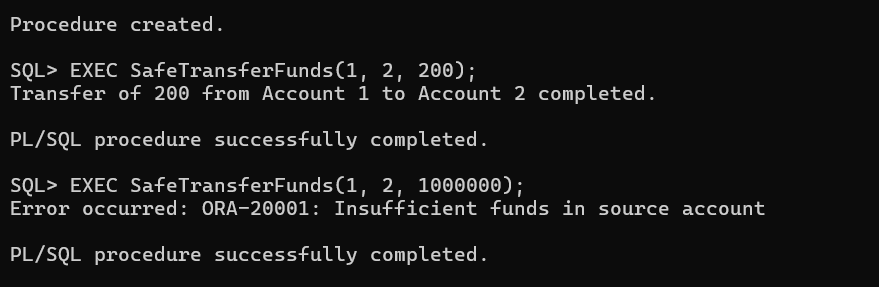
DBMS\_OUTPUT.PUT\_LINE('Error occurred: ' || SQLERRM);

END;

/

EXEC SafeTransferFunds(1, 2, 200);

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.

CREATE OR REPLACE PROCEDURE UpdateSalary(p\_emp\_id IN NUMBER,

p\_percent IN NUMBER

) IS

v\_rows NUMBER;

BEGIN

UPDATE Employees

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

WHERE EmployeeID = p\_emp\_id;

v\_rows := SQL%ROWCOUNT;

IF v\_rows = 0 THEN

RAISE\_APPLICATION\_ERROR(-20002, 'Employee with ID ' || p\_emp\_id || ' does not exist.');

ELSE

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

END IF;

EXCEPTION

WHEN OTHERS THEN

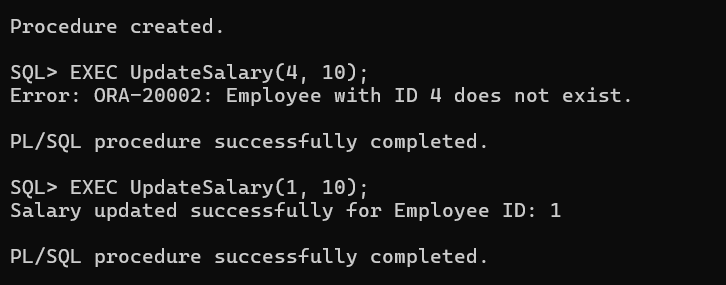
DBMS\_OUTPUT.PUT\_LINE('Error: ' || SQLERRM);

END;

/

EXEC UpdateSalary(1, 10);

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.

CREATE OR REPLACE PROCEDURE AddNewCustomer (

p\_customer\_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)

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

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

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

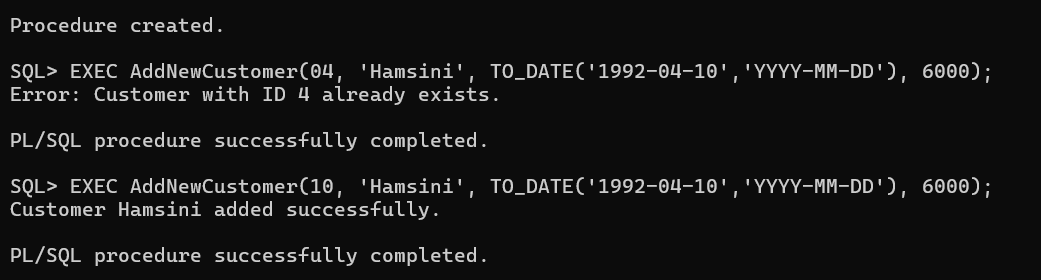
DBMS\_OUTPUT.PUT\_LINE('Error: Customer with ID ' || p\_customer\_id || ' already exists.');

END;

/

EXEC AddNewCustomer(10, 'Hamsini', TO\_DATE('1992-04-10','YYYY-MM-DD'), 6000);

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

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

BEGIN

UPDATE Accounts

SET Balance = Balance + (Balance \* 0.01)

WHERE AccountType = 'Savings';

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

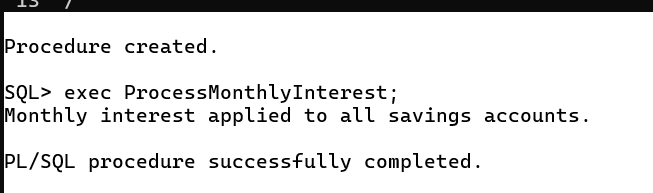
EXCEPTION

END;

/

EXEC ProcessMonthlyInterest;

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.

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (p\_department IN VARCHAR2,

p\_bonus\_percent IN NUMBER

) IS

v\_count NUMBER;

BEGIN

UPDATE Employees

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

WHERE Department = p\_department;

v\_count := SQL%ROWCOUNT;

IF v\_count = 0 THEN

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

ELSE

DBMS\_OUTPUT.PUT\_LINE('Bonus applied to ' || v\_count || ' employee(s) in department ' || p\_department);

END IF;

EXCEPTION

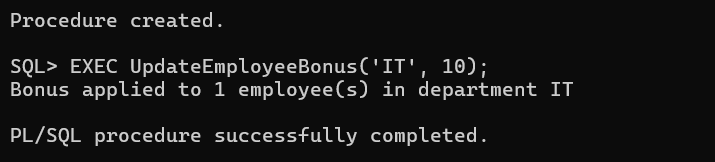
WHEN OTHERS THEN

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

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.

CREATE OR REPLACE PROCEDURE TransferFunds (

p\_from\_account IN NUMBER,

p\_to\_account IN NUMBER,

p\_amount IN NUMBER

) IS

v\_balance NUMBER;

BEGIN

-- Check balance in source account

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;

-- Debit source account

UPDATE Accounts

SET Balance = Balance - p\_amount

WHERE AccountID = p\_from\_account;

-- Credit destination account

UPDATE Accounts

SET Balance = Balance + p\_amount

WHERE AccountID = p\_to\_account;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Transferred ' || p\_amount ||

' from Account ' || p\_from\_account ||

' to Account ' || p\_to\_account);

EXCEPTION

WHEN OTHERS THEN

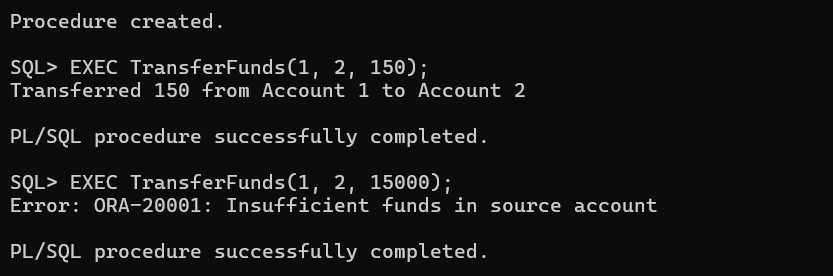
ROLLBACK;

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

END;

/

EXEC TransferFunds(1, 2, 150);



**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.

CREATE OR REPLACE FUNCTION CalculateAge (

p\_dob IN DATE

) RETURN NUMBER IS

v\_age NUMBER;

BEGIN

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

RETURN v\_age;

END;

/

OUTPUT: Function created.

DECLARE

v\_age NUMBER;

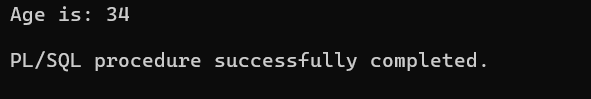
BEGIN

v\_age := CalculateAge(TO\_DATE('1990-07-20','YYYY-MM-DD'));

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

END;

/



**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.

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment (

p\_loan\_amount IN NUMBER,

p\_annual\_rate IN NUMBER,

p\_years IN NUMBER

) RETURN NUMBER IS

v\_monthly\_rate NUMBER := p\_annual\_rate / 1200;

v\_months NUMBER := p\_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 /

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

END IF;

RETURN ROUND(v\_emi, 2);

END;

/



DECLARE

v\_emi NUMBER;

BEGIN

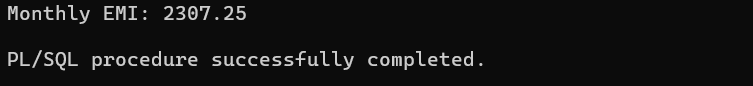
v\_emi := CalculateMonthlyInstallment(50000, 10, 2);

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

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.

SET SERVEROUTPUT ON;

DECLARE

v\_result BOOLEAN;

BEGIN

v\_result := HasSufficientBalance(1, 300);

IF v\_result THEN

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

ELSE

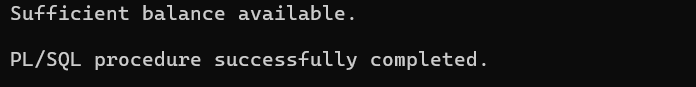
DBMS\_OUTPUT.PUT\_LINE('Insufficient balance.');

END IF;

END;

/

OUTPUT:

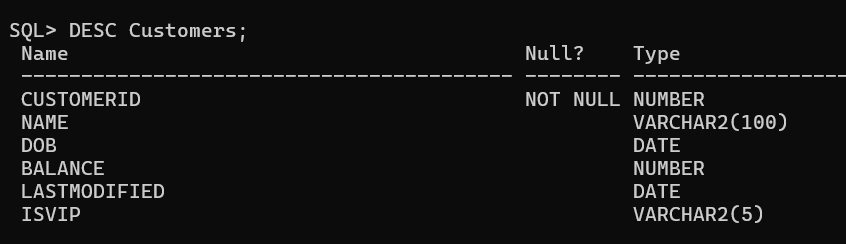


**Exercise 5: Triggers**

**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.

DESC Customers; (lastModified exists)



CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON Customers

FOR EACH ROW

BEGIN

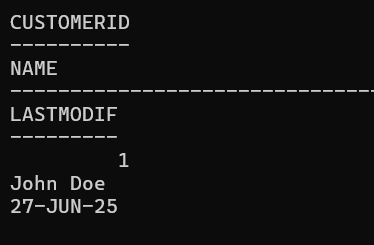
:NEW.LastModified := SYSDATE;

END;

/

OUTPUT: Trigger created.

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



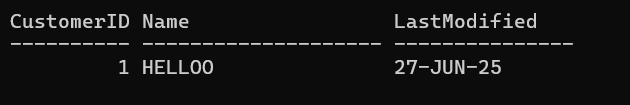
UPDATE Customers

SET Name = 'HELLOO'

WHERE CustomerID = 1;

COMMIT;

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



**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.

CREATE TABLE AuditLog (

LogID NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,

AccountID NUMBER, ActionDate DATE, Amount NUMBER, Type VARCHAR2(10)

);

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog (AccountID, ActionDate, Amount, Type)

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

END;

/

OUTPUT: Trigger Created.

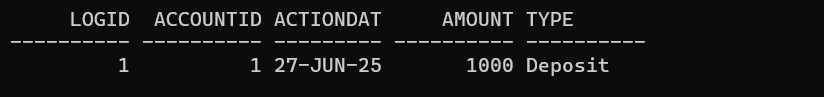
INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)

VALUES (10, 1, SYSDATE, 1000, 'Deposit');

COMMIT;

OUTPUT: 1 row created.

SELECT \* FROM AuditLog ORDER BY LogID DESC;



**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.

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' THEN

IF :NEW.Amount > v\_balance THEN

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

ELSIF :NEW.Amount <= 0 THEN

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

END IF;

END IF;

IF :NEW.TransactionType = 'Deposit' THEN

IF :NEW.Amount <= 0 THEN

RAISE\_APPLICATION\_ERROR(-20003, 'Deposit amount must be positive.');

END IF;

END IF;

END;

/

OUTPUT: Trigger created.

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

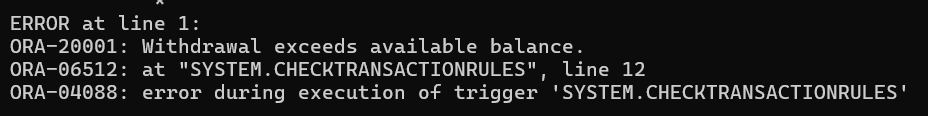
VALUES (201, 1, SYSDATE, 1000, 'Deposit');

COMMIT;

OUTPUT: 1 row created.

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

VALUES (202, 1, SYSDATE, 999999, 'Withdrawal');



**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.

DECLARE

CURSOR cur\_monthly\_statements IS

SELECT c.CustomerID, c.Name, a.AccountID, 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 TRUNC(t.TransactionDate, 'MM') = TRUNC(SYSDATE, 'MM')

ORDER BY c.CustomerID, t.TransactionDate;

v\_customer\_id Customers.CustomerID%TYPE;

v\_name Customers.Name%TYPE;

v\_account\_id Accounts.AccountID%TYPE;

v\_date Transactions.TransactionDate%TYPE;

v\_amount Transactions.Amount%TYPE;

v\_type Transactions.TransactionType%TYPE;

BEGIN

OPEN cur\_monthly\_statements;

LOOP

FETCH cur\_monthly\_statements INTO v\_customer\_id, v\_name, v\_account\_id, v\_date, v\_amount, v\_type;

EXIT WHEN cur\_monthly\_statements%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE(

'Customer: ' || v\_name || ' (ID: ' || v\_customer\_id || ') | ' ||

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

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

v\_type || ': $' || v\_amount

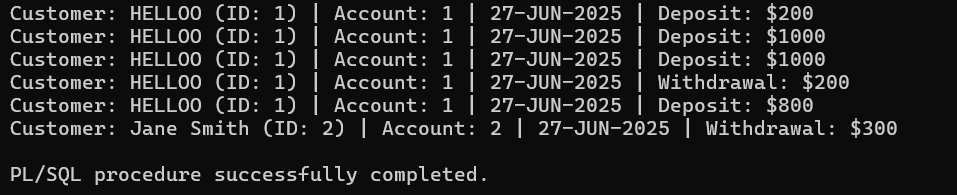
);

END LOOP;

CLOSE cur\_monthly\_statements;

END;

/



**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.

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('RS.' || v\_fee || ' annual fee applied to Account ' || v\_account\_id);

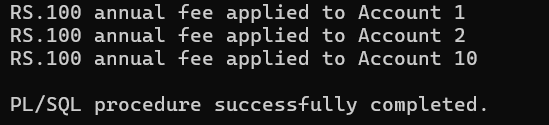
END LOOP;

CLOSE cur\_accounts;

COMMIT;

END;

/



**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.

DECLARE

CURSOR cur\_loans IS

SELECT LoanID, InterestRate FROM Loans;

v\_loan\_id Loans.LoanID%TYPE;

v\_rate Loans.InterestRate%TYPE;

v\_new\_rate NUMBER;

BEGIN

OPEN cur\_loans;

LOOP

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

EXIT WHEN cur\_loans%NOTFOUND;

IF v\_rate < 6 THEN

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

ELSIF v\_rate < 8 THEN

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

ELSE

v\_new\_rate := v\_rate;

END IF;

IF v\_new\_rate != v\_rate THEN

UPDATE Loans

SET InterestRate = v\_new\_rate

WHERE LoanID = v\_loan\_id;

DBMS\_OUTPUT.PUT\_LINE('Updated LoanID ' || v\_loan\_id ||': Interest rate changed from ' || v\_rate ||' to ' || v\_new\_rate);

ELSE

DBMS\_OUTPUT.PUT\_LINE('LoanID ' || v\_loan\_id || ' already meets policy. No change.');

END IF;

END LOOP;

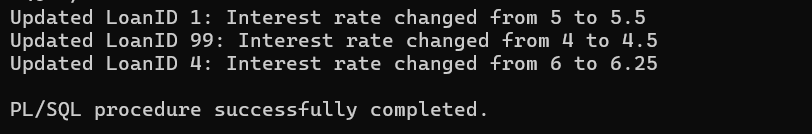
CLOSE cur\_loans;

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.

CREATE OR REPLACE PACKAGE CustomerManagement IS

PROCEDURE AddCustomer(

p\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_dob IN DATE,

p\_balance IN NUMBER

);

PROCEDURE UpdateCustomer(

p\_id IN NUMBER,

p\_name IN VARCHAR2

);

FUNCTION GetBalance(

p\_id IN NUMBER

) RETURN NUMBER;

END CustomerManagement;

/

OUTPUT: Package created.

CREATE OR REPLACE PACKAGE BODY CustomerManagement IS

PROCEDURE AddCustomer(

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)

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

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

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);

END AddCustomer;

PROCEDURE UpdateCustomer(

p\_id IN NUMBER,

p\_name IN VARCHAR2

) IS

BEGIN

UPDATE Customers

SET Name = p\_name

WHERE CustomerID = p\_id;

IF SQL%ROWCOUNT = 0 THEN

DBMS\_OUTPUT.PUT\_LINE('No customer found with ID ' || p\_id);

ELSE

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

END IF;

END UpdateCustomer;

FUNCTION GetBalance(

p\_id IN NUMBER

) RETURN NUMBER IS

v\_balance NUMBER;

BEGIN

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

RETURN v\_balance;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

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

RETURN NULL;

WHEN OTHERS THEN

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

RETURN NULL;

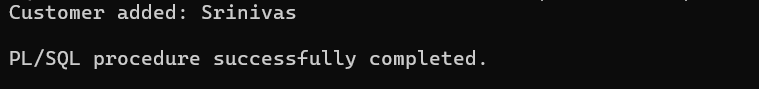
END GetBalance;

END CustomerManagement;

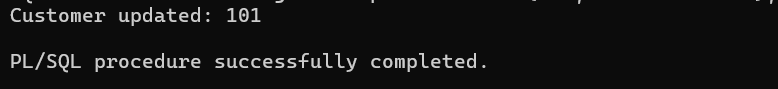
/

OUTPUT: Package body created.

EXEC CustomerManagement.AddCustomer(101, 'Srinivas', TO\_DATE('1900-12-28', 'YYYY-MM-DD'), 10000);



EXEC CustomerManagement.UpdateCustomer(101, 'Srinivas E.');



VARIABLE v\_bal NUMBER;

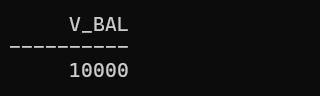
BEGIN

:v\_bal := CustomerManagement.GetBalance(101);

END;

/

PRINT v\_bal;



**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.

CREATE OR REPLACE PACKAGE EmployeeManagement IS

PROCEDURE HireEmployee(

p\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_position IN VARCHAR2,

p\_salary IN NUMBER,

p\_dept IN VARCHAR2

);

PROCEDURE UpdateEmployeeDetails(

p\_id IN NUMBER,

p\_position IN VARCHAR2,

p\_salary IN NUMBER

);

FUNCTION AnnualSalary(

p\_id IN NUMBER

) RETURN NUMBER;

END EmployeeManagement;

/

OUTPUT: Package created.

CREATE OR REPLACE PACKAGE BODY EmployeeManagement IS

PROCEDURE HireEmployee(

p\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_position IN VARCHAR2,

p\_salary IN NUMBER,

p\_dept IN 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 ' || p\_name || ' hired.');

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

WHEN OTHERS THEN

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

END HireEmployee;

PROCEDURE UpdateEmployeeDetails(

p\_id IN NUMBER,

p\_position IN VARCHAR2,

p\_salary IN NUMBER

) IS

BEGIN

UPDATE Employees

SET Position = p\_position,

Salary = p\_salary

WHERE EmployeeID = p\_id;

IF SQL%ROWCOUNT = 0 THEN

DBMS\_OUTPUT.PUT\_LINE('No employee found with ID ' || p\_id);

ELSE

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

END IF;

END UpdateEmployeeDetails;

FUNCTION AnnualSalary(

p\_id IN 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

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

RETURN NULL;

WHEN OTHERS THEN

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

RETURN NULL;

END AnnualSalary;

END EmployeeManagement;

/

OUTPUT: Package body created.

EXEC EmployeeManagement.HireEmployee(201, 'Harika', 'Analyst', 40000, 'Finance');



EXEC EmployeeManagement.UpdateEmployeeDetails(201, 'Senior Analyst', 50000);



VARIABLE v\_salary NUMBER;

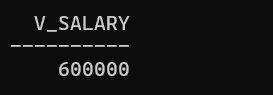
BEGIN

:v\_salary := EmployeeManagement.AnnualSalary(201);

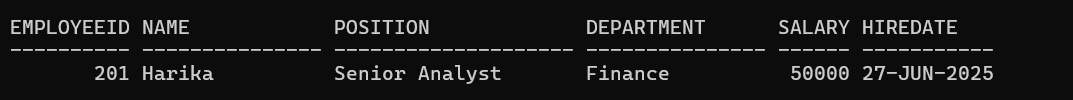
END;

/

PRINT v\_salary;



SELECT \* FROM Employees WHERE EmployeeID = 201;



**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.

CREATE OR REPLACE PACKAGE AccountOperations IS

PROCEDURE OpenAccount(

p\_account\_id IN NUMBER,

p\_customer\_id IN NUMBER,

p\_type IN VARCHAR2,

p\_balance IN NUMBER

);

PROCEDURE CloseAccount(

p\_account\_id IN NUMBER

);

FUNCTION TotalBalance(

p\_customer\_id IN NUMBER

) RETURN NUMBER;

END AccountOperations;

/

OUTPUT: Package created.

CREATE OR REPLACE PACKAGE BODY AccountOperations IS

PROCEDURE OpenAccount(

p\_account\_id IN NUMBER,

p\_customer\_id IN NUMBER,

p\_type IN VARCHAR2,

p\_balance IN NUMBER

) IS

BEGIN

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

VALUES (p\_account\_id, p\_customer\_id, p\_type, p\_balance, SYSDATE);

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

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

WHEN OTHERS THEN

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

END OpenAccount;

PROCEDURE CloseAccount(

p\_account\_id IN NUMBER

) IS

v\_count NUMBER;

BEGIN

DELETE FROM Accounts

WHERE AccountID = p\_account\_id;

v\_count := SQL%ROWCOUNT;

IF v\_count = 0 THEN

DBMS\_OUTPUT.PUT\_LINE('No account found with ID ' || p\_account\_id);

ELSE

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

END IF;

END CloseAccount;

FUNCTION TotalBalance(

p\_customer\_id IN NUMBER

) RETURN NUMBER IS

v\_total NUMBER;

BEGIN

SELECT NVL(SUM(Balance), 0)

INTO v\_total

FROM Accounts

WHERE CustomerID = p\_customer\_id;

RETURN v\_total;

EXCEPTION

WHEN OTHERS THEN

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

RETURN NULL;

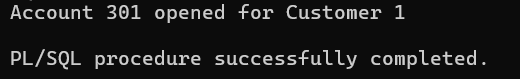
END TotalBalance;

END AccountOperations;

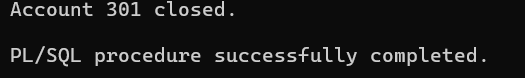
/

OUTPUT: Package body created.

EXEC AccountOperations.OpenAccount(301, 1, 'Savings', 5000);



EXEC AccountOperations.CloseAccount(301);



VARIABLE v\_total\_balance NUMBER;

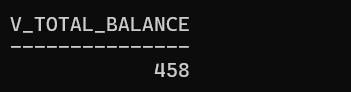
BEGIN

:v\_total\_balance := AccountOperations.TotalBalance(1);

END;

/

PRINT v\_total\_balance;



SELECT \* FROM Accounts WHERE CustomerID = 1;

