**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 :**

DECLARE

v\_customer\_id Customers.CustomerID%TYPE;

v\_dob Customers.DOB%TYPE;

v\_age NUMBER;

BEGIN

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

v\_customer\_id := rec.CustomerID;

v\_dob := rec.DOB;

-- Calculate age

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

IF v\_age > 60 THEN

-- Apply 1% discount on all their loans

UPDATE Loans

SET InterestRate = InterestRate - 1

WHERE CustomerID = v\_customer\_id;

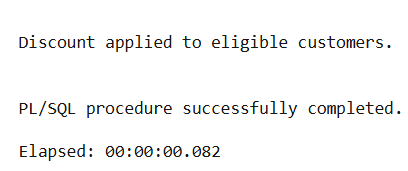
END IF;

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('Discount applied to eligible customers.');

END;

OUTPUT :

 (since in give schema there are no customers with lone and age is greatera than 60 . so I inserted couple of rows which satisfy the query)

**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.

PROGRMA :

(since in the given schema there is no IsVIP char(1) attribute so altered the table to add the attribute)

ALTER TABLE Customers ADD IsVIP CHAR(1);

BEGIN

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

IF rec.Balance > 10000 THEN

UPDATE Customers

SET IsVIP = 'Y'

WHERE CustomerID = rec.CustomerID;

ELSE

UPDATE Customers

SET IsVIP = 'N'

WHERE CustomerID = rec.CustomerID;

END IF;

END LOOP;

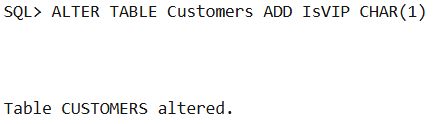
DBMS\_OUTPUT.PUT\_LINE('VIP status updated.');

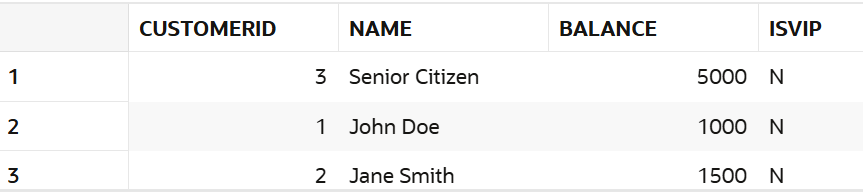
END;

**AFTER EXECUTION :**

SELECT CustomerID, 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.

BEGIN

FOR rec IN (

SELECT

L.LoanID,

L.CustomerID,

L.EndDate,

C.Name

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 ID ' || rec.LoanID ||

' for customer ' || rec.Name ||

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

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.

PROGRAM :

CREATE OR REPLACE PROCEDURE SafeTransferFunds (

    p\_FromAccountID IN NUMBER,

    p\_ToAccountID IN NUMBER,

    p\_Amount IN NUMBER

)

AS

    v\_FromBalance NUMBER;

BEGIN

    -- Start Transaction Block

    SAVEPOINT before\_transfer;

    -- Step 1: Check balance of the source account

    SELECT Balance INTO v\_FromBalance

    FROM Accounts

    WHERE AccountID = p\_FromAccountID

    FOR UPDATE;

    IF v\_FromBalance < p\_Amount THEN

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

    END IF;

    -- Step 2: Deduct amount from source account

    UPDATE Accounts

    SET Balance = Balance - p\_Amount,

        LastModified = SYSDATE

    WHERE AccountID = p\_FromAccountID;

    -- Step 3: Add amount to destination account

    UPDATE Accounts

    SET Balance = Balance + p\_Amount,

        LastModified = SYSDATE

    WHERE AccountID = p\_ToAccountID;

    -- Step 4: Record transactions

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

    VALUES (Transactions\_seq.NEXTVAL, p\_FromAccountID, SYSDATE, p\_Amount, 'Withdrawal');

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

    VALUES (Transactions\_seq.NEXTVAL, p\_ToAccountID, SYSDATE, p\_Amount, 'Deposit');

    COMMIT;

EXCEPTION

    WHEN OTHERS THEN

        -- Log error message

        INSERT INTO ErrorLog (ErrorMessage)

        VALUES ('Error during fund transfer: ' || SQLERRM);

        ROLLBACK TO before\_transfer;

        -- Optionally raise the error again

        RAISE;

END;

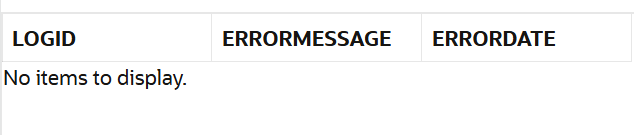
/

BEGIN

    SafeTransferFunds(1, 2, 5000);

END;

OUTPUT:



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

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

PROGAM:

CREATE OR REPLACE PROCEDURE UpdateSalary (

    p\_EmployeeID IN NUMBER,

    p\_Percent IN NUMBER

)

AS

    v\_Salary Employees.Salary%TYPE;

BEGIN

    -- Try to get current salary

    SELECT Salary INTO v\_Salary

    FROM Employees

    WHERE EmployeeID = p\_EmployeeID

    FOR UPDATE;

    -- Update the salary with the percentage increase

    UPDATE Employees

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

    WHERE EmployeeID = p\_EmployeeID;

    COMMIT;

EXCEPTION

    WHEN NO\_DATA\_FOUND THEN

        -- Log if the employee doesn't exist

        INSERT INTO ErrorLog (ErrorMessage)

        VALUES ('UpdateSalary Error: Employee ID ' || p\_EmployeeID || ' not found.');

        ROLLBACK;

    WHEN OTHERS THEN

        -- Handle all other unexpected errors

        INSERT INTO ErrorLog (ErrorMessage)

        VALUES ('UpdateSalary Error: ' || SQLERRM);

        ROLLBACK;

END;

/

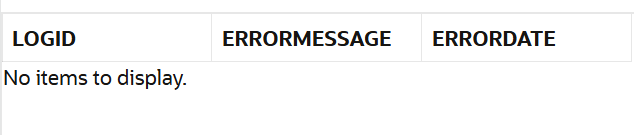
OUTPUT :Testcase 1

BEGIN

    UpdateSalary(1, 10); -- Increase salary of EmployeeID=1 by 10%

END;

NO ERROR



**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

PROGRAM :

CREATE OR REPLACE PROCEDURE AddNewCustomer (

    p\_customer\_id IN NUMBER,

    p\_name        IN VARCHAR2,

    p\_dob         IN DATE,

    p\_balance     IN NUMBER

)

IS

    v\_exists NUMBER;

BEGIN

    -- Check if the customer already exists

    SELECT COUNT(\*) INTO v\_exists

    FROM Customers

    WHERE CustomerID = p\_customer\_id;

    IF v\_exists > 0 THEN

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

    ELSE

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

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

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

    END IF;

EXCEPTION

    WHEN OTHERS THEN

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

END;

/

OUTPUT :

Test case 1:

BEGIN

    AddNewCustomer(5, 'Diana Prince', TO\_DATE('1995-08-15', 'YYYY-MM-DD'), 7000);

END;



Test case 2:

BEGIN

    AddNewCustomer(5, 'Duplicate Diana', TO\_DATE('1992-09-09', 'YYYY-MM-DD'), 8000);

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.

1. CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest
2. IS
3. v\_count NUMBER := 0;
4. BEGIN
5. -- Update savings account balances with 1% interest
6. UPDATE Accounts
7. SET
8. Balance = Balance + (Balance \* 0.01),
9. LastModified = SYSDATE
10. WHERE
11. UPPER(AccountType) = 'SAVINGS';
12. -- Get number of rows affected
13. v\_count := SQL%ROWCOUNT;
14. COMMIT;
15. DBMS\_OUTPUT.PUT\_LINE('Interest processed for ' || v\_count || ' savings account(s).');
16. EXCEPTION
17. WHEN OTHERS THEN
18. ROLLBACK;
19. DBMS\_OUTPUT.PUT\_LINE('Error processing interest: ' || SQLERRM);
20. END;
21. /

Test case:

1. BEGIN
2. ProcessMonthlyInterest;
3. 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.

PROGRAM :

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (

    p\_department IN VARCHAR2,

    p\_bonus\_percent IN NUMBER

)

IS

    v\_count NUMBER := 0;

BEGIN

    -- Update salary by adding bonus percentage

    UPDATE Employees

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

    WHERE UPPER(Department) = UPPER(p\_department);

    v\_count := SQL%ROWCOUNT;

    COMMIT;

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

EXCEPTION

    WHEN OTHERS THEN

        ROLLBACK;

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

END;

/

TESTCASE:

-- Apply 10% bonus to employees in IT department

BEGIN

    UpdateEmployeeBonus('IT', 10);

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.

PROGRAM:  
CREATE OR REPLACE PROCEDURE TransferFunds (

    p\_from\_account\_id IN NUMBER,

    p\_to\_account\_id   IN NUMBER,

    p\_amount          IN NUMBER

)

IS

    v\_from\_balance NUMBER;

BEGIN

    -- 1. Lock and check the balance of the source account

    SELECT Balance INTO v\_from\_balance

    FROM Accounts

    WHERE AccountID = p\_from\_account\_id

    FOR UPDATE;

    -- 2. Validate sufficient balance

    IF v\_from\_balance < p\_amount THEN

        DBMS\_OUTPUT.PUT\_LINE('Transfer failed: Insufficient funds in account ' || p\_from\_account\_id);

        RETURN;

    END IF;

    -- 3. Deduct from source

    UPDATE Accounts

    SET Balance = Balance - p\_amount,

        LastModified = SYSDATE

    WHERE AccountID = p\_from\_account\_id;

    -- 4. Add to destination

    UPDATE Accounts

    SET Balance = Balance + p\_amount,

        LastModified = SYSDATE

    WHERE AccountID = p\_to\_account\_id;

    COMMIT;

    DBMS\_OUTPUT.PUT\_LINE('Transfer of ' || p\_amount ||

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

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

EXCEPTION

    WHEN NO\_DATA\_FOUND THEN

        ROLLBACK;

        DBMS\_OUTPUT.PUT\_LINE('Transfer failed: One of the accounts does not exist.');

    WHEN OTHERS THEN

        ROLLBACK;

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

END;

/

Testcase:

BEGIN

    TransferFunds(101, 102, 500);

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.

PROGRAM:

CREATE OR REPLACE FUNCTION CalculateAge (

    p\_dob DATE

) RETURN NUMBER

IS

    v\_age NUMBER;

BEGIN

    -- Calculate age using months between and divide by 12

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

    RETURN v\_age;

EXCEPTION

    WHEN OTHERS THEN

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

        RETURN NULL;

END;

/

TESTCASE:

DECLARE

    v\_age NUMBER;

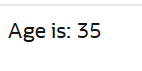
BEGIN

    v\_age := CalculateAge(TO\_DATE('1990-06-15', 'YYYY-MM-DD'));

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

END;

OUTPUT:



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

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

PROGRAM:

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment (

    p\_loan\_amount   IN NUMBER,

    p\_annual\_rate   IN NUMBER,

    p\_duration\_years IN NUMBER

) RETURN NUMBER

IS

    v\_monthly\_rate NUMBER;

    v\_total\_months NUMBER;

    v\_emi NUMBER;

BEGIN

    -- Monthly interest rate

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

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

    -- EMI Calculation using standard formula

    IF v\_monthly\_rate = 0 THEN

        -- No interest case

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

    ELSE

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

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

    END IF;

    RETURN ROUND(v\_emi, 2);

EXCEPTION

    WHEN OTHERS THEN

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

        RETURN NULL;

END;

/

TESTCASE:

DECLARE

    v\_emi NUMBER;

BEGIN

    v\_emi := CalculateMonthlyInstallment(500000, 7.5, 5);

    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.

PROGRAM:

CREATE OR REPLACE FUNCTION HasSufficientBalance (

    p\_account\_id IN NUMBER,

    p\_amount     IN NUMBER

) RETURN BOOLEAN

IS

    v\_balance NUMBER;

BEGIN

    -- Fetch account balance

    SELECT Balance INTO v\_balance

    FROM Accounts

    WHERE AccountID = p\_account\_id;

    -- Return TRUE if sufficient, FALSE otherwise

    RETURN v\_balance >= p\_amount;

EXCEPTION

    WHEN NO\_DATA\_FOUND THEN

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

        RETURN FALSE;

    WHEN OTHERS THEN

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

        RETURN FALSE;

END;

/

TESTCASE:

DECLARE

    v\_result BOOLEAN;

BEGIN

    v\_result := HasSufficientBalance(101, 500);

    IF v\_result THEN

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

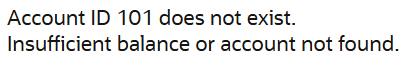
    ELSE

        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.

PROGRAM:

DECLARE

    v\_result BOOLEAN;

BEGIN

    v\_result := HasSufficientBalance(101, 500);

    IF v\_result THEN

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

    ELSE

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

    END IF;

END;

TESTCASE:

-- Update a customer's name

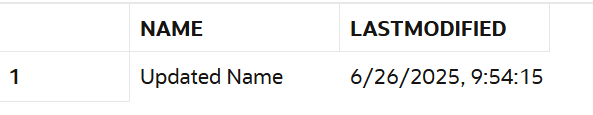
UPDATE Customers

SET Name = 'Updated Name'

WHERE CustomerID = 1;

SELECT Name, 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.

PROGRAM:

CREATE TABLE AuditLog (

    LogID NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,

    TransactionID NUMBER,

    AccountID NUMBER,

    Amount NUMBER,

    TransactionType VARCHAR2(10),

    Action VARCHAR2(10),

    LogTimestamp DATE

);

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

    INSERT INTO AuditLog (

        TransactionID,

        AccountID,

        Amount,

        TransactionType,

        Action,

        LogTimestamp

    ) VALUES (

        :NEW.TransactionID,

        :NEW.AccountID,

        :NEW.Amount,

        :NEW.TransactionType,

        'INSERT',

        SYSDATE

    );

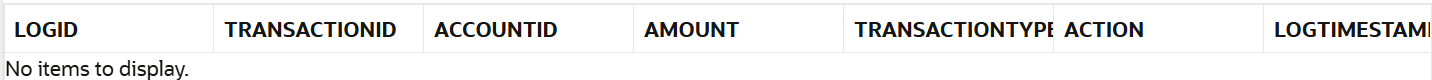
END;

/

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

VALUES (1001, 101, SYSDATE, 500, 'Deposit');

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.

PROGRAM:

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

DECLARE

    v\_balance NUMBER;

BEGIN

    -- Get the current balance of the account

    SELECT Balance INTO v\_balance

    FROM Accounts

    WHERE AccountID = :NEW.AccountID;

    -- Rule 1: Deposits must be positive

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

        RAISE\_APPLICATION\_ERROR(-20001, 'Deposit amount must be greater than 0.');

    END IF;

    -- Rule 2: Withdrawals must not exceed balance

    IF :NEW.TransactionType = 'Withdrawal' AND :NEW.Amount > v\_balance THEN

        RAISE\_APPLICATION\_ERROR(-20002, 'Insufficient balance for withdrawal.');

    END IF;

END;

/

OUTPUT:

ORA-20002: Insufficient balance for 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.

PROGRAM:

DECLARE

    -- Cursor to fetch transactions of the current month with customer details

    CURSOR monthly\_cursor IS

        SELECT c.CustomerID, c.Name, a.AccountID,

               t.TransactionID, t.TransactionDate, t.Amount, t.TransactionType

        FROM Transactions t

        JOIN Accounts a ON t.AccountID = a.AccountID

        JOIN Customers c ON a.CustomerID = c.CustomerID

        WHERE TO\_CHAR(t.TransactionDate, 'MM-YYYY') = TO\_CHAR(SYSDATE, 'MM-YYYY')

        ORDER BY c.CustomerID, t.TransactionDate;

    -- Variables to hold cursor data

    v\_customer\_id Customers.CustomerID%TYPE;

    v\_name Customers.Name%TYPE;

    v\_account\_id Accounts.AccountID%TYPE;

    v\_transaction\_id Transactions.TransactionID%TYPE;

    v\_transaction\_date Transactions.TransactionDate%TYPE;

    v\_amount Transactions.Amount%TYPE;

    v\_type Transactions.TransactionType%TYPE;

    v\_last\_customer\_id NUMBER := 0;

BEGIN

    DBMS\_OUTPUT.PUT\_LINE('--- Monthly Statements (' || TO\_CHAR(SYSDATE, 'Month YYYY') || ') ---');

    OPEN monthly\_cursor;

    LOOP

        FETCH monthly\_cursor INTO v\_customer\_id, v\_name, v\_account\_id, v\_transaction\_id, v\_transaction\_date, v\_amount, v\_type;

        EXIT WHEN monthly\_cursor%NOTFOUND;

        -- Print customer header once per customer

        IF v\_customer\_id != v\_last\_customer\_id THEN

            DBMS\_OUTPUT.PUT\_LINE(CHR(10) || 'Customer: ' || v\_name || ' (ID: ' || v\_customer\_id || ')');

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

            v\_last\_customer\_id := v\_customer\_id;

        END IF;

        -- Print transaction details

        DBMS\_OUTPUT.PUT\_LINE('TxnID: ' || v\_transaction\_id ||

                             ', Date: ' || TO\_CHAR(v\_transaction\_date, 'DD-Mon-YYYY') ||

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

                             ', Type: ' || v\_type ||

                             ', Amount: ₹' || v\_amount);

    END LOOP;

    CLOSE monthly\_cursor;

EXCEPTION

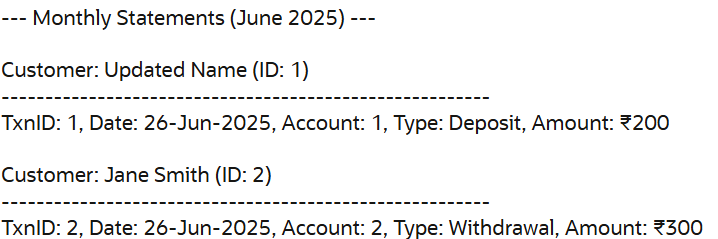
    WHEN OTHERS THEN

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

END;

/

OUTPUT:



**Scenario 2:** Apply annual fee to all accounts.

* + **Question:** Write a PL/SQL block using an explicit cursor **ApplyAnnualFee** that deducts an annual maintenance fee from the balance of all accounts.

PROGRAM:

DECLARE

    -- Annual fee to deduct

    v\_fee CONSTANT NUMBER := 250;

    -- Cursor to loop through all accounts

    CURSOR account\_cursor IS

        SELECT AccountID, Balance

        FROM Accounts

        FOR UPDATE;

    -- Variables to hold data

    v\_account\_id Accounts.AccountID%TYPE;

    v\_balance Accounts.Balance%TYPE;

BEGIN

    DBMS\_OUTPUT.PUT\_LINE('--- Applying Annual Maintenance Fee ---');

    OPEN account\_cursor;

    LOOP

        FETCH account\_cursor INTO v\_account\_id, v\_balance;

        EXIT WHEN account\_cursor%NOTFOUND;

        -- Deduct the fee from account balance

        IF v\_balance >= v\_fee THEN

            UPDATE Accounts

            SET Balance = Balance - v\_fee,

                LastModified = SYSDATE

            WHERE AccountID = v\_account\_id;

            -- Optional: Insert into Transactions table

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

            VALUES (Transactions\_seq.NEXTVAL, v\_account\_id, SYSDATE, v\_fee, 'Fee');

            DBMS\_OUTPUT.PUT\_LINE('₹' || v\_fee || ' fee deducted from Account ' || v\_account\_id);

        ELSE

            DBMS\_OUTPUT.PUT\_LINE('Skipped Account ' || v\_account\_id || ' due to insufficient balance');

        END IF;

    END LOOP;

    CLOSE account\_cursor;

    COMMIT;

    DBMS\_OUTPUT.PUT\_LINE('Annual fee application complete.');

EXCEPTION

    WHEN OTHERS THEN

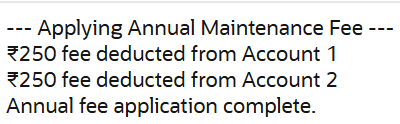
        ROLLBACK;

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

END;

/

OUTPUT:



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

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

PROGRAM:

DECLARE

    -- Annual fee to deduct

    v\_fee CONSTANT NUMBER := 250;

    -- Cursor to loop through all accounts

    CURSOR account\_cursor IS

        SELECT AccountID, Balance

        FROM Accounts

        FOR UPDATE;

    -- Variables to hold data

    v\_account\_id Accounts.AccountID%TYPE;

    v\_balance Accounts.Balance%TYPE;

BEGIN

    DBMS\_OUTPUT.PUT\_LINE('--- Applying Annual Maintenance Fee ---');

    OPEN account\_cursor;

    LOOP

        FETCH account\_cursor INTO v\_account\_id, v\_balance;

        EXIT WHEN account\_cursor%NOTFOUND;

        -- Deduct the fee from account balance

        IF v\_balance >= v\_fee THEN

            UPDATE Accounts

            SET Balance = Balance - v\_fee,

                LastModified = SYSDATE

            WHERE AccountID = v\_account\_id;

            -- Optional: Insert into Transactions table

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

            VALUES (Transactions\_seq.NEXTVAL, v\_account\_id, SYSDATE, v\_fee, 'Fee');

            DBMS\_OUTPUT.PUT\_LINE('₹' || v\_fee || ' fee deducted from Account ' || v\_account\_id);

        ELSE

            DBMS\_OUTPUT.PUT\_LINE('Skipped Account ' || v\_account\_id || ' due to insufficient balance');

        END IF;

    END LOOP;

    CLOSE account\_cursor;

    COMMIT;

    DBMS\_OUTPUT.PUT\_LINE('Annual fee application complete.');

EXCEPTION

    WHEN OTHERS THEN

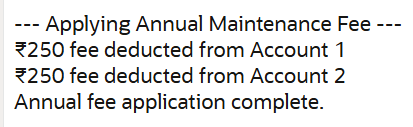
        ROLLBACK;

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

END;

/

OUTPUT:



**Exercise 7: Packages**

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

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

PROGRAM:

Package Specification

DECLARE

    v\_balance NUMBER;

BEGIN

    v\_balance := CustomerManagement.GetCustomerBalance(3);

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

END;

CREATE OR REPLACE PACKAGE CustomerManagement AS

    PROCEDURE AddCustomer (

        p\_customer\_id   IN Customers.CustomerID%TYPE,

        p\_name          IN Customers.Name%TYPE,

        p\_dob           IN Customers.DOB%TYPE,

        p\_balance       IN Customers.Balance%TYPE

    );

    PROCEDURE UpdateCustomerDetails (

        p\_customer\_id   IN Customers.CustomerID%TYPE,

        p\_name          IN Customers.Name%TYPE,

        p\_balance       IN Customers.Balance%TYPE

    );

    FUNCTION GetCustomerBalance (

        p\_customer\_id   IN Customers.CustomerID%TYPE

    ) RETURN NUMBER;

END CustomerManagement;

/

Package body:

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

    PROCEDURE AddCustomer (

        p\_customer\_id   IN Customers.CustomerID%TYPE,

        p\_name          IN Customers.Name%TYPE,

        p\_dob           IN Customers.DOB%TYPE,

        p\_balance       IN Customers.Balance%TYPE

    ) IS

    BEGIN

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

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

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

    EXCEPTION

        WHEN DUP\_VAL\_ON\_INDEX THEN

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

        WHEN OTHERS THEN

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

    END;

    PROCEDURE UpdateCustomerDetails (

        p\_customer\_id   IN Customers.CustomerID%TYPE,

        p\_name          IN Customers.Name%TYPE,

        p\_balance       IN Customers.Balance%TYPE

    ) IS

    BEGIN

        UPDATE Customers

        SET Name = p\_name,

            Balance = p\_balance,

            LastModified = SYSDATE

        WHERE CustomerID = p\_customer\_id;

        IF SQL%ROWCOUNT = 0 THEN

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

        ELSE

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

        END IF;

    END;

    FUNCTION GetCustomerBalance (

        p\_customer\_id IN Customers.CustomerID%TYPE

    ) RETURN NUMBER IS

        v\_balance NUMBER;

    BEGIN

        SELECT Balance INTO v\_balance

        FROM Customers

        WHERE CustomerID = p\_customer\_id;

        RETURN v\_balance;

    EXCEPTION

        WHEN NO\_DATA\_FOUND THEN

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

            RETURN NULL;

        WHEN OTHERS THEN

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

            RETURN NULL;

    END;

END CustomerManagement;

/



Example Usage

DECLARE

    v\_balance NUMBER;

BEGIN

    v\_balance := CustomerManagement.GetCustomerBalance(3);

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

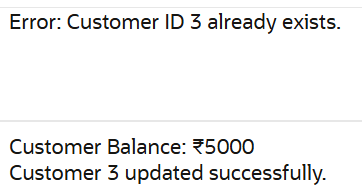
END;

BEGIN

    CustomerManagement.UpdateCustomerDetails(3, 'Alex Raymond', 8000);

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.

Package Specification

CREATE OR REPLACE PACKAGE EmployeeManagement AS

    PROCEDURE HireEmployee (

        p\_employee\_id IN Employees.EmployeeID%TYPE,

        p\_name        IN Employees.Name%TYPE,

        p\_position    IN Employees.Position%TYPE,

        p\_salary      IN Employees.Salary%TYPE,

        p\_department  IN Employees.Department%TYPE,

        p\_hire\_date   IN Employees.HireDate%TYPE

    );

    PROCEDURE UpdateEmployeeDetails (

        p\_employee\_id IN Employees.EmployeeID%TYPE,

        p\_name        IN Employees.Name%TYPE,

        p\_salary      IN Employees.Salary%TYPE

    );

    FUNCTION CalculateAnnualSalary (

        p\_employee\_id IN Employees.EmployeeID%TYPE

    ) RETURN NUMBER;

END EmployeeManagement;

/



Package Body :

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

    PROCEDURE HireEmployee (

        p\_employee\_id IN Employees.EmployeeID%TYPE,

        p\_name        IN Employees.Name%TYPE,

        p\_position    IN Employees.Position%TYPE,

        p\_salary      IN Employees.Salary%TYPE,

        p\_department  IN Employees.Department%TYPE,

        p\_hire\_date   IN Employees.HireDate%TYPE

    ) IS

    BEGIN

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

        VALUES (p\_employee\_id, p\_name, p\_position, p\_salary, p\_department, p\_hire\_date);

        DBMS\_OUTPUT.PUT\_LINE('Employee ' || p\_name || ' hired successfully.');

    EXCEPTION

        WHEN DUP\_VAL\_ON\_INDEX THEN

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

        WHEN OTHERS THEN

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

    END;

    PROCEDURE UpdateEmployeeDetails (

        p\_employee\_id IN Employees.EmployeeID%TYPE,

        p\_name        IN Employees.Name%TYPE,

        p\_salary      IN Employees.Salary%TYPE

    ) IS

    BEGIN

        UPDATE Employees

        SET Name = p\_name,

            Salary = p\_salary

        WHERE EmployeeID = p\_employee\_id;

        IF SQL%ROWCOUNT = 0 THEN

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

        ELSE

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

        END IF;

    END;

    FUNCTION CalculateAnnualSalary (

        p\_employee\_id IN Employees.EmployeeID%TYPE

    ) RETURN NUMBER IS

        v\_salary NUMBER;

    BEGIN

        SELECT Salary INTO v\_salary

        FROM Employees

        WHERE EmployeeID = p\_employee\_id;

        RETURN v\_salary \* 12;

    EXCEPTION

        WHEN NO\_DATA\_FOUND THEN

            DBMS\_OUTPUT.PUT\_LINE('Employee ID not found: ' || p\_employee\_id);

            RETURN NULL;

        WHEN OTHERS THEN

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

            RETURN NULL;

    END;

END EmployeeManagement;

/

Example Usage

BEGIN

    EmployeeManagement.HireEmployee(

        3, 'Priya Singh', 'Analyst', 40000, 'Finance', TO\_DATE('2022-01-15', 'YYYY-MM-DD')

    );

END;

BEGIN

    EmployeeManagement.UpdateEmployeeDetails(3, 'Priya S.', 45000);

END;

DECLARE

    v\_annual\_salary NUMBER;

BEGIN

    v\_annual\_salary := EmployeeManagement.CalculateAnnualSalary(3);

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

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.

PROGRAM:

Package Specification

CREATE OR REPLACE PACKAGE AccountOperations AS

    PROCEDURE OpenAccount (

        p\_account\_id   IN Accounts.AccountID%TYPE,

        p\_customer\_id  IN Accounts.CustomerID%TYPE,

        p\_account\_type IN Accounts.AccountType%TYPE,

        p\_balance      IN Accounts.Balance%TYPE

    );

    PROCEDURE CloseAccount (

        p\_account\_id IN Accounts.AccountID%TYPE

    );

    FUNCTION GetTotalCustomerBalance (

        p\_customer\_id IN Accounts.CustomerID%TYPE

    ) RETURN NUMBER;

END AccountOperations;

/



Package Body

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

    PROCEDURE OpenAccount (

        p\_account\_id   IN Accounts.AccountID%TYPE,

        p\_customer\_id  IN Accounts.CustomerID%TYPE,

        p\_account\_type IN Accounts.AccountType%TYPE,

        p\_balance      IN Accounts.Balance%TYPE

    ) IS

    BEGIN

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

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

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

    EXCEPTION

        WHEN DUP\_VAL\_ON\_INDEX THEN

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

        WHEN OTHERS THEN

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

    END;

    PROCEDURE CloseAccount (

        p\_account\_id IN Accounts.AccountID%TYPE

    ) IS

    BEGIN

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

        IF SQL%ROWCOUNT = 0 THEN

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

        ELSE

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

        END IF;

    EXCEPTION

        WHEN OTHERS THEN

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

    END;

    FUNCTION GetTotalCustomerBalance (

        p\_customer\_id IN Accounts.CustomerID%TYPE

    ) 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 getting total balance: ' || SQLERRM);

            RETURN NULL;

    END;

END AccountOperations;

/

Example Usage

BEGIN

    AccountOperations.OpenAccount(201, 1, 'Savings', 5000);

END;

BEGIN

    AccountOperations.CloseAccount(201);

END;

DECLARE

    v\_total NUMBER;

BEGIN

    v\_total := AccountOperations.GetTotalCustomerBalance(1);

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

END;

OUTPUT: