**Exercise 2: Error Handling**

**Scenario 1:** Handle exceptions during fund transfers between accounts.

SET SERVEROUTPUT ON;

CREATE OR REPLACE PROCEDURE SafeTransferFunds (

acc\_from IN NUMBER,

acc\_to IN NUMBER,

amt IN NUMBER

) AS

bal NUMBER;

BEGIN

SELECT Balance INTO bal

FROM Accounts

WHERE AccountID = acc\_from

FOR UPDATE;

IF bal < amt THEN

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

END IF;

UPDATE Accounts

SET Balance = Balance - amt,

LastModified = SYSDATE

WHERE AccountID = acc\_from;

UPDATE Accounts

SET Balance = Balance + amt,

LastModified = SYSDATE

WHERE AccountID = acc\_to;

COMMIT;

END;

/

BEGIN

SafeTransferFunds(1, 2, 300);

EXCEPTION

WHEN OTHERS THEN

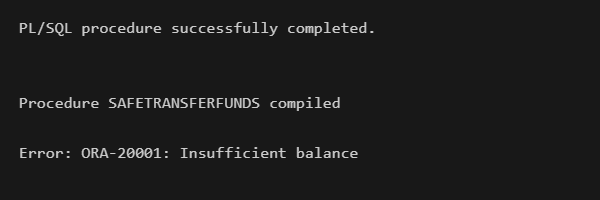
ROLLBACK;

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

END;

/

**Output:**

****

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

CREATE OR REPLACE PROCEDURE UpdateSalary (

emp\_id IN NUMBER,

pinc IN NUMBER

) AS

BEGIN

UPDATE Employees

SET Salary = Salary + (Salary \* pinc/ 100)

WHERE EmployeeID = emp\_id;

IF SQL%ROWCOUNT = 0 THEN

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

END IF;

COMMIT;

END;

/

BEGIN

UPDATESALARY(1, 15);

EXCEPTION

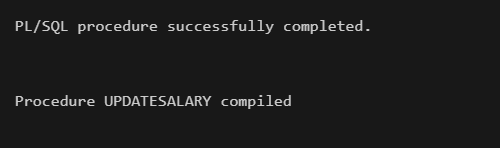
WHEN OTHERS THEN

ROLLBACK;

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

END;

/



**Scenario 3:** Ensure data integrity when adding a new customer

CREATE OR REPLACE PROCEDURE AddNewCustomer (

cid IN NUMBER,

name IN VARCHAR2,

dob IN DATE,

bal IN NUMBER

) AS

BEGIN

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

VALUES (cid, name, dob, bal, SYSDATE, 'N');

COMMIT;

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

ROLLBACK;

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

WHEN OTHERS THEN

ROLLBACK;

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

END;

/

BEGIN

AddNewCustomer(1,'Eva', TO\_DATE('1995-02-04', 'YYYY-MM-DD'), 2000);

EXCEPTION

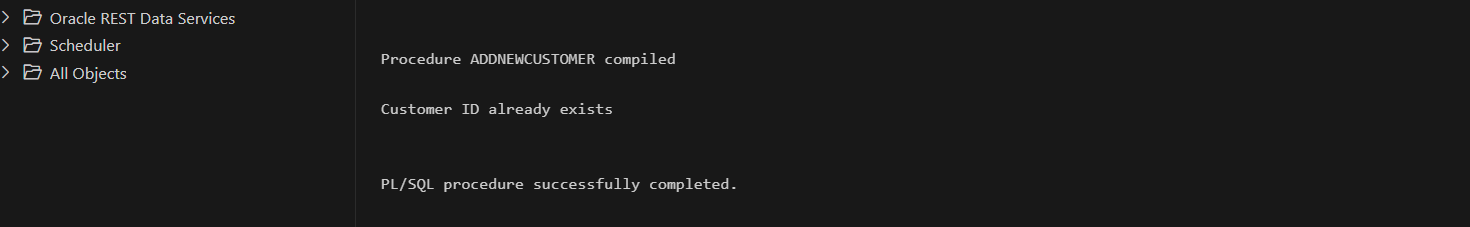
WHEN OTHERS THEN

ROLLBACK;

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

END;

/



**Exercise 4: Functions**

**Scenario 1:** Calculate the age of customers for eligibility checks.

CREATE OR REPLACE FUNCTION CalculateAge (

dob IN DATE

) RETURN NUMBER IS

years NUMBER;

BEGIN

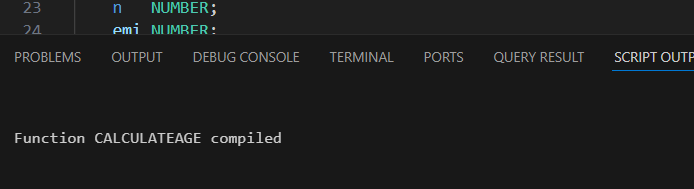
years := FLOOR(MONTHS\_BETWEEN(SYSDATE, dob) / 12);

RETURN years;

END;

/

SELECT CalculateAge(TO\_DATE('2000-06-15', 'YYYY-MM-DD')) AS age FROM customers;



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

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment (

amt IN NUMBER,

rate IN NUMBER,

time IN NUMBER

) RETURN NUMBER IS

r NUMBER;

n NUMBER;

emi NUMBER;

BEGIN

r := rate / 12 / 100;

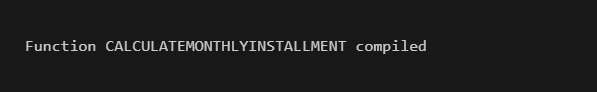
n := time \* 12;

emi := (amt \* r \* POWER(1 + r, n)) / (POWER(1 + r, n) - 1);

RETURN ROUND(emi, 2);

END;

/



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

CREATE OR REPLACE FUNCTION HasSufficientBalance (

id IN NUMBER,

amt IN NUMBER

) RETURN BOOLEAN IS

bal NUMBER;

BEGIN

SELECT Balance INTO bal

FROM Accounts

WHERE AccountID = id;

RETURN bal >= amt;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN FALSE;

WHEN OTHERS THEN

RETURN FALSE;

END;

/

DECLARE

answer BOOLEAN;

BEGIN

answer := HasSufficientBalance(2,3000);

IF answer THEN

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

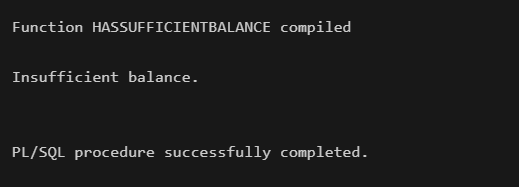
ELSE

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

END IF;

END;

/



**Exercise 5: Triggers**

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

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON Customers

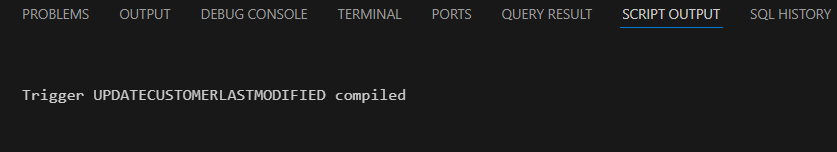
FOR EACH ROW

BEGIN

:NEW.LastModified := SYSDATE;

END;

/



**Scenario 2:** Maintain an audit log for all transaction

CREATE TABLE AuditLog (

ID NUMBER GENERATED BY DEFAULT ON NULL AS IDENTITY PRIMARY KEY,

TID NUMBER,

Action VARCHAR2(50),

LogTime DATE

);

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

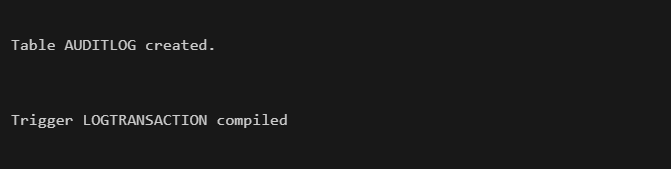
BEGIN

INSERT INTO AuditLog (TID, Action, LogTime)

VALUES (:NEW.TransactionID, 'INSERTED', SYSDATE);

END;

/



**Scenario 3:** Enforce business rules on deposits and withdrawals.

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

DECLARE

acc\_balance NUMBER;

BEGIN

-- Get account balance

SELECT Balance INTO acc\_balance

FROM Accounts

WHERE AccountID = :NEW.AccountID;

IF :NEW.TransactionType = 'Withdrawal' THEN

IF :NEW.Amount > acc\_balance THEN

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

END IF;

ELSIF :NEW.TransactionType = 'Deposit' THEN

IF :NEW.Amount <= 0 THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Amount cannot be negative');

END IF;

ELSE

RAISE\_APPLICATION\_ERROR(-20001, 'Invalid transaction type');

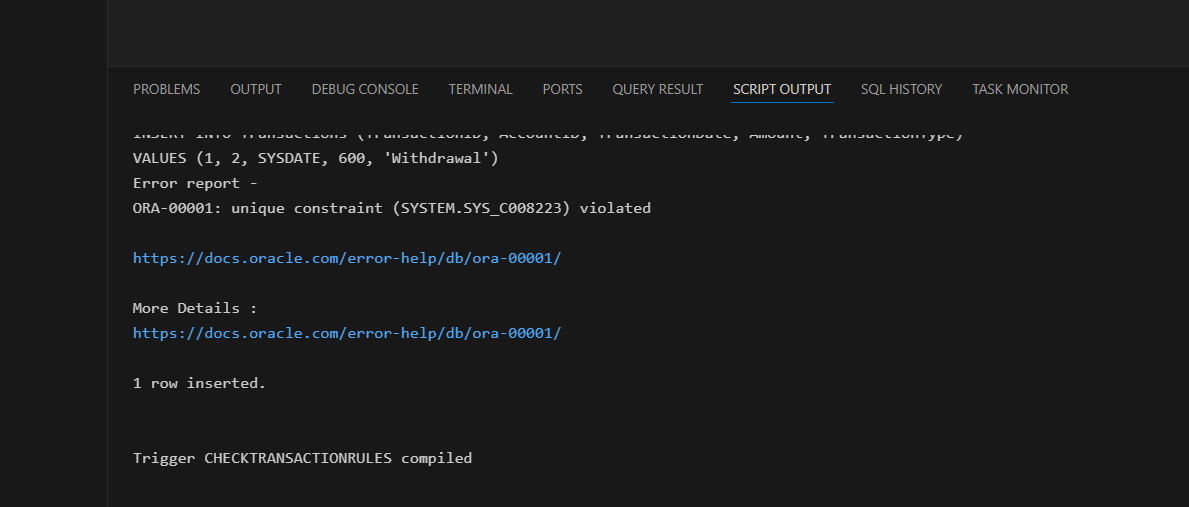
END IF;

END;

/

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

VALUES (3, 2, SYSDATE, 600, 'Withdrawal');



**Exercise 6: Cursors**

**Scenario 1:** Generate monthly statements for all customers

DECLARE

CURSOR txn\_cursor IS

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

FROM Customers c

JOIN Accounts a ON c.CustomerID = a.CustomerID

JOIN Transactions t ON a.AccountID = t.AccountID

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

ORDER BY c.CustomerID;

cid Customers.CustomerID%TYPE;

c\_name Customers.Name%TYPE;

c\_date Transactions.TransactionDate%TYPE;

amt Transactions.Amount%TYPE;

c\_type Transactions.TransactionType%TYPE;

BEGIN

OPEN txn\_cursor;

LOOP

FETCH txn\_cursor INTO cid, c\_name, c\_date, amt, c\_type;

EXIT WHEN txn\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Customer: ' || c\_name || ' | Date: ' || TO\_CHAR(c\_date, 'DD-Mon') ||

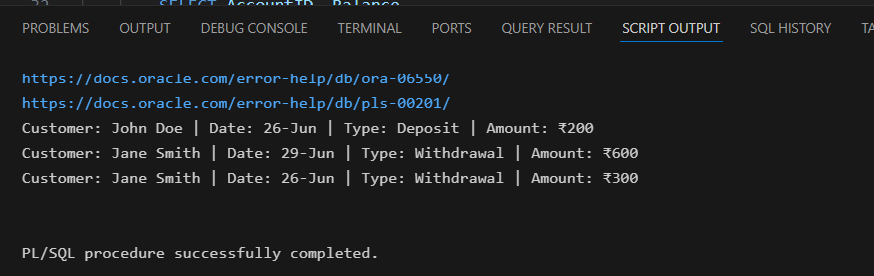
' | Type: ' || c\_type || ' | Amount: ₹' || amt);

END LOOP;

CLOSE txn\_cursor;

END;

/



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

DECLARE

CURSOR accnt\_cursor IS

SELECT AccountID, Balance

FROM Accounts;

a\_id Accounts.AccountID%TYPE;

a\_bal Accounts.Balance%TYPE;

BEGIN

OPEN accnt\_cursor;

LOOP

FETCH accnt\_cursor INTO a\_id, a\_bal;

EXIT WHEN accnt\_cursor%NOTFOUND;

UPDATE Accounts

SET Balance = Balance - 500,

LastModified = SYSDATE

WHERE AccountID = a\_id;

END LOOP;

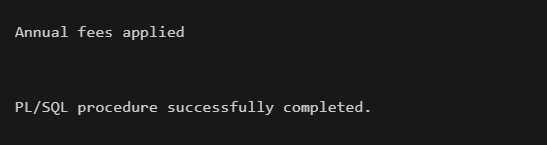
CLOSE accnt\_cursor;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Annual fees applied ');

END;

/



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

DECLARE

CURSOR loan\_cursor IS

SELECT LoanID, InterestRate, LoanAmount

FROM Loans;

l\_id Loans.LoanID%TYPE;

l\_rate Loans.InterestRate%TYPE;

l\_amount Loans.LoanAmount%TYPE;

BEGIN

OPEN loan\_cursor;

LOOP

FETCH loan\_cursor INTO l\_id, l\_rate, l\_amount;

EXIT WHEN loan\_cursor%NOTFOUND;

IF l\_amount > 100000 THEN

UPDATE Loans

SET InterestRate = l\_rate + 0.5

WHERE LoanID = l\_id;

END IF;

END LOOP;

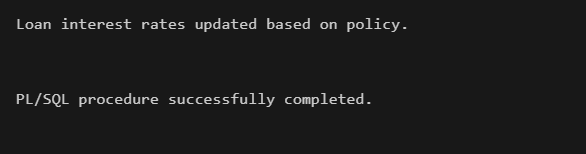
CLOSE loan\_cursor;

COMMIT;

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

END;

/



**Exercise 7: Packages**

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

CREATE OR REPLACE PACKAGE CustomerManagement AS

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

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

FUNCTION GetBalance(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\_bal NUMBER) IS

BEGIN

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

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

COMMIT;

END;

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

BEGIN

UPDATE Customers

SET Name = p\_name, DOB = p\_dob, Balance = p\_bal, LastModified = SYSDATE

WHERE CustomerID = p\_id;

COMMIT;

END;

FUNCTION GetBalance(p\_id NUMBER) RETURN NUMBER IS

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance

FROM Customers

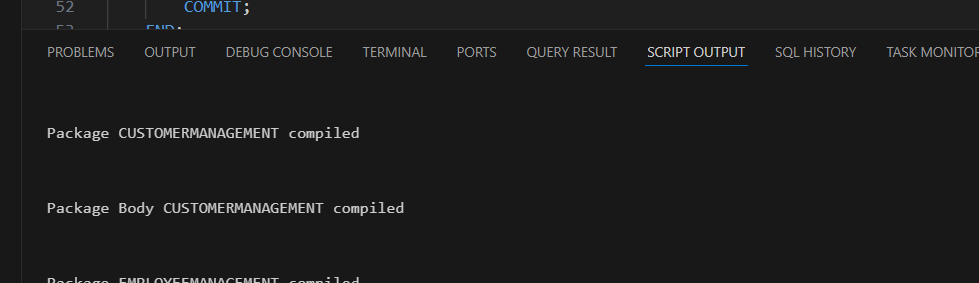
WHERE CustomerID = p\_id;

RETURN v\_balance;

END;

END CustomerManagement;

/



**Scenario 2:** Create a package to manage employee data.

CREATE OR REPLACE PACKAGE EmployeeManagement AS

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\_salary NUMBER);

FUNCTION AnnualSalary(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, p\_date DATE) IS

BEGIN

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

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

COMMIT;

END;

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

BEGIN

UPDATE Employees

SET Salary = p\_salary

WHERE EmployeeID = p\_id;

COMMIT;

END;

FUNCTION AnnualSalary(p\_id NUMBER) RETURN NUMBER IS

v\_salary NUMBER;

BEGIN

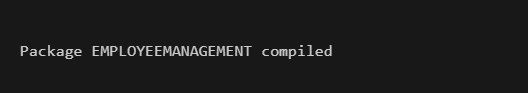
SELECT Salary INTO v\_salary FROM Employees WHERE EmployeeID = p\_id;

RETURN v\_salary \* 12;

END;

END EmployeeManagement;

/



**Scenario 3:** Group all account-related operations into a package.

CREATE OR REPLACE PACKAGE AccountOperations AS

PROCEDURE OpenAccount(p\_id NUMBER, p\_custid NUMBER, p\_type VARCHAR2, p\_bal NUMBER);

PROCEDURE CloseAccount(p\_id NUMBER);

END AccountOperations;

/

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

PROCEDURE OpenAccount(p\_id NUMBER, p\_custid NUMBER, p\_type VARCHAR2, p\_bal NUMBER) IS

BEGIN

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

VALUES (p\_id, p\_custid, p\_type, p\_bal, SYSDATE);

COMMIT;

END;

PROCEDURE CloseAccount(p\_id NUMBER) IS

BEGIN

DELETE FROM Accounts WHERE AccountID = p\_id;

COMMIT;

END;

END AccountOperations;

/

