**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.

**Program**

CREATE TABLE Customers (

CustomerID NUMBER PRIMARY KEY,

Name VARCHAR2(100),

DOB DATE,

Balance NUMBER,

LastModified DATE

);

CREATE TABLE Loans (

LoanID NUMBER PRIMARY KEY,

CustomerID NUMBER,

LoanAmount NUMBER,

InterestRate NUMBER,

StartDate DATE,

EndDate DATE,

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

INSERT INTO Customers VALUES (1, 'John Doe', TO\_DATE('1950-05-15', 'YYYY-MM-DD'), 2000, SYSDATE);

INSERT INTO Loans VALUES (101, 1, 5000, 8, SYSDATE, ADD\_MONTHS(SYSDATE, 60));

BEGIN

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

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

UPDATE Loans

SET InterestRate = InterestRate - 1

WHERE CustomerID = cust.CustomerID;

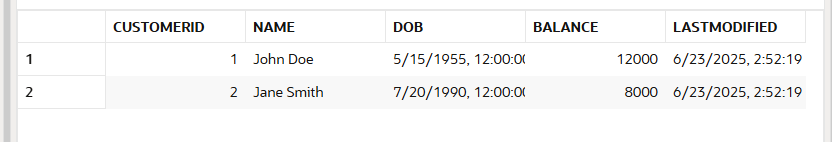
END IF;

END LOOP;

END;

/

SELECT \* FROM Customers;

**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.

**Program**

CREATE TABLE Customers (

CustomerID NUMBER PRIMARY KEY,

Name VARCHAR2(100),

DOB DATE,

Balance NUMBER,

LastModified DATE,

IsVIP VARCHAR2(5)

);

INSERT INTO Customers VALUES (1, 'John Doe', TO\_DATE('1950-05-15', 'YYYY-MM-DD'), 12000, SYSDATE, NULL);

INSERT INTO Customers VALUES (2, 'Jane Smith', TO\_DATE('1985-03-22', 'YYYY-MM-DD'), 8000, SYSDATE, NULL);

BEGIN

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

IF cust.Balance > 10000 THEN

UPDATE Customers

SET IsVIP = 'TRUE'

WHERE CustomerID = cust.CustomerID;

END IF;

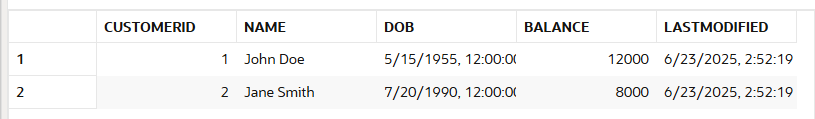
END LOOP;

END;

/

SELECT \* 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.

**Program**

SET SERVEROUTPUT ON;

CREATE TABLE Customers (

CustomerID NUMBER PRIMARY KEY,

Name VARCHAR2(100),

DOB DATE,

Balance NUMBER,

LastModified DATE

);

CREATE TABLE Loans (

LoanID NUMBER PRIMARY KEY,

CustomerID NUMBER,

LoanAmount NUMBER,

InterestRate NUMBER,

StartDate DATE,

EndDate DATE,

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

INSERT INTO Customers VALUES (1, 'John Doe', TO\_DATE('1950-05-15', 'YYYY-MM-DD'), 12000, SYSDATE);

INSERT INTO Customers VALUES (2, 'Jane Smith', TO\_DATE('1985-03-22', 'YYYY-MM-DD'), 8000, SYSDATE);

INSERT INTO Customers VALUES (3, 'Mark Ray', TO\_DATE('1990-10-10', 'YYYY-MM-DD'), 15000, SYSDATE);

INSERT INTO Loans VALUES (101, 1, 5000, 7, SYSDATE, SYSDATE + 10);

INSERT INTO Loans VALUES (102, 2, 3000, 8, SYSDATE, SYSDATE + 40);

INSERT INTO Loans VALUES (103, 3, 7000, 6, SYSDATE, SYSDATE + 5);

COMMIT;

BEGIN

FOR dueLoan IN (

SELECT L.LoanID, 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 ' || dueLoan.LoanID || ' for ' || dueLoan.Name || ' is due on ' || TO\_CHAR(dueLoan.EndDate, 'DD-Mon-YYYY'));

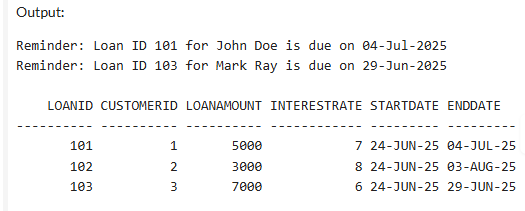
END LOOP;

END;

/

SELECT \* FROM Loans;

**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 TABLE Customers (

CustomerID NUMBER PRIMARY KEY,

Name VARCHAR2(100),

DOB DATE,

Balance NUMBER,

LastModified DATE

);

CREATE TABLE Accounts (

AccountID NUMBER PRIMARY KEY,

CustomerID NUMBER,

AccountType VARCHAR2(20),

Balance NUMBER,

LastModified DATE,

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

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

VALUES (101, 'Alice', TO\_DATE('1990-01-01', 'YYYY-MM-DD'), 0, SYSDATE);

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

VALUES (102, 'Bob', TO\_DATE('1985-05-20', 'YYYY-MM-DD'), 0, SYSDATE);

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

VALUES (1, 101, 'Savings', 1000, SYSDATE);

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

VALUES (2, 102, 'Savings', 500, SYSDATE);

COMMIT;

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

-- Check balance

SELECT Balance INTO v\_balance

FROM Accounts

WHERE AccountID = p\_from\_account\_id;

IF v\_balance < p\_amount THEN

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

END IF;

UPDATE Accounts

SET Balance = Balance - p\_amount,

LastModified = SYSDATE

WHERE AccountID = p\_from\_account\_id;

UPDATE Accounts

SET Balance = Balance + p\_amount,

LastModified = SYSDATE

WHERE AccountID = p\_to\_account\_id;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END;

/

BEGIN

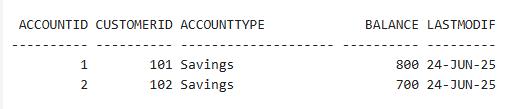
SafeTransferFunds(1, 2, 200);

END;

/

SELECT \* FROM Accounts;

**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.

**Program**

BEGIN

EXECUTE IMMEDIATE 'DROP TABLE Employees';

EXCEPTION

WHEN OTHERS THEN NULL;

END;

/

CREATE TABLE Employees (

EmployeeID NUMBER PRIMARY KEY,

Name VARCHAR2(100),

Position VARCHAR2(50),

Salary NUMBER,

Department VARCHAR2(50),

HireDate DATE

);

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

VALUES (1, 'Alice Johnson', 'Manager', 70000, 'HR', TO\_DATE('2015-06-15', 'YYYY-MM-DD'));

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

VALUES (2, 'Bob Brown', 'Developer', 60000, 'IT', TO\_DATE('2017-03-20', 'YYYY-MM-DD'));

COMMIT;

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 ID not found.');

END IF;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END;

/

BEGIN

UpdateSalary(2, 10);

END;

/

SELECT \* FROM Employees;

**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.

**Program**

CREATE TABLE Customers (

CustomerID NUMBER PRIMARY KEY,

Name VARCHAR2(100),

DOB DATE,

Balance NUMBER,

LastModified DATE

);

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

VALUES (101, 'Alice', TO\_DATE('2002-06-18', 'YYYY-MM-DD'), 7000, SYSDATE);

COMMIT;

CREATE OR REPLACE PROCEDURE AddNewCustomer (

p\_customer\_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\_customer\_id, p\_name, p\_dob, p\_balance, SYSDATE);

COMMIT;

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

ROLLBACK;

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

WHEN OTHERS THEN

ROLLBACK;

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

END;

/

BEGIN

AddNewCustomer(102, 'Prabha', TO\_DATE('2008-05-26', 'YYYY-MM-DD'), 3070);

END;

/

BEGIN

AddNewCustomer(103, 'Mughil', TO\_DATE('2000-04-26', 'YYYY-MM-DD'), 2700);

END;

/

SELECT \* FROM Customers;

**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.

**Program**

CREATE TABLE Accounts (

AccountID NUMBER PRIMARY KEY,

CustomerID NUMBER,

AccountType VARCHAR2(20),

Balance NUMBER,

LastModified DATE

);

INSERT INTO Accounts VALUES (1, 101, 'Savings', 1000, SYSDATE);

INSERT INTO Accounts VALUES (2, 102, 'Checking', 5000, SYSDATE);

INSERT INTO Accounts VALUES (3, 103, 'Savings', 2000, SYSDATE);

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

BEGIN

UPDATE Accounts

SET Balance = Balance + (Balance \* 0.01)

WHERE AccountType = 'Savings';

COMMIT;

END ProcessMonthlyInterest;

/

BEGIN

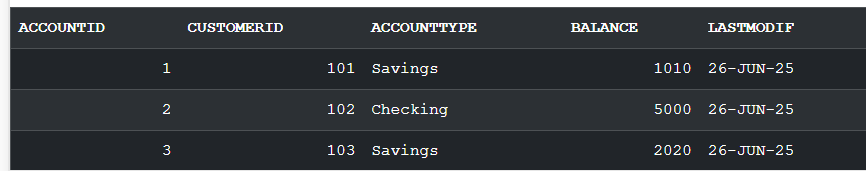
ProcessMonthlyInterest;

END;

/

SELECT \* FROM Accounts;

**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 TABLE Employees (

EmployeeID NUMBER PRIMARY KEY,

Name VARCHAR2(100),

Position VARCHAR2(50),

Salary NUMBER,

Department VARCHAR2(50),

HireDate DATE

);

/

INSERT INTO Employees VALUES (1, 'Alice Johnson', 'Manager', 70000, 'HR', TO\_DATE('2015-06-15', 'YYYY-MM-DD'));

INSERT INTO Employees VALUES (2, 'Bob Brown', 'Developer', 60000, 'IT', TO\_DATE('2017-03-20', 'YYYY-MM-DD'));

INSERT INTO Employees VALUES (3, 'Clara Lee', 'Analyst', 50000, 'HR', TO\_DATE('2020-01-10', 'YYYY-MM-DD'));

COMMIT;

/

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (

p\_Department IN VARCHAR2,

p\_BonusPercent IN NUMBER

) IS

BEGIN

UPDATE Employees

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

WHERE Department = p\_Department;

COMMIT;

END UpdateEmployeeBonus;

/

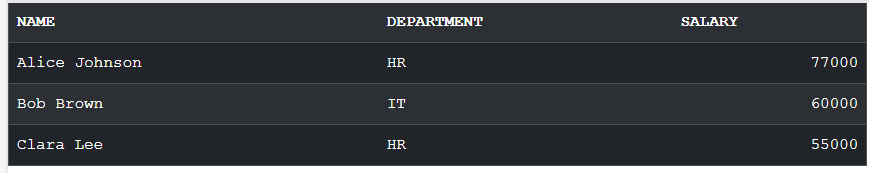
BEGIN

UpdateEmployeeBonus('HR', 10);

END;

/

SELECT Name, Department, Salary FROM Employees;

**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 TABLE Accounts (

AccountID NUMBER PRIMARY KEY,

CustomerID NUMBER,

AccountType VARCHAR2(20),

Balance NUMBER,

LastModified DATE

);

INSERT INTO Accounts VALUES (1, 101, 'Savings', 1000, SYSDATE);

INSERT INTO Accounts VALUES (2, 101, 'Checking', 500, SYSDATE);

COMMIT;

/

CREATE OR REPLACE PROCEDURE TransferFunds (

p\_FromAccountID IN NUMBER,

p\_ToAccountID IN NUMBER,

p\_Amount IN NUMBER

) IS

v\_Balance NUMBER;

BEGIN

SELECT Balance INTO v\_Balance

FROM Accounts

WHERE AccountID = p\_FromAccountID

FOR UPDATE;

IF v\_Balance < p\_Amount THEN

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

END IF;

UPDATE Accounts

SET Balance = Balance - p\_Amount,

LastModified = SYSDATE

WHERE AccountID = p\_FromAccountID;

UPDATE Accounts

SET Balance = Balance + p\_Amount,

LastModified = SYSDATE

WHERE AccountID = p\_ToAccountID;

COMMIT;

END TransferFunds;

/

BEGIN

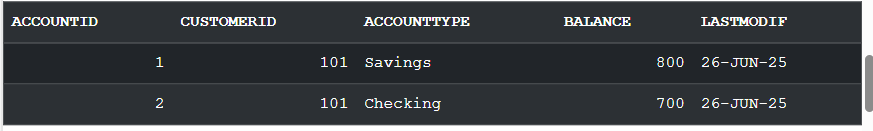
TransferFunds(1, 2, 200);

END;

/

SELECT \* FROM Accounts;

**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 TABLE Accounts (

AccountID NUMBER PRIMARY KEY,

CustomerID NUMBER,

AccountType VARCHAR2(20),

Balance NUMBER,

LastModified DATE

);

CREATE OR REPLACE FUNCTION CalculateAge(p\_DOB DATE)

RETURN NUMBER IS

v\_Age NUMBER;

BEGIN

v\_Age := FLOOR(MONTHS\_BETWEEN(SYSDATE, p\_DOB) / 12);

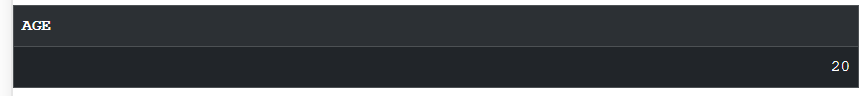
RETURN v\_Age;

END CalculateAge;

/

SELECT CalculateAge(TO\_DATE('2005-02-26', 'YYYY-MM-DD')) AS Age FROM DUAL;

**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 TABLE Accounts (

AccountID NUMBER PRIMARY KEY,

CustomerID NUMBER,

AccountType VARCHAR2(20),

Balance NUMBER,

LastModified DATE

);

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment (

p\_LoanAmount NUMBER,

p\_InterestRate NUMBER,

p\_Years NUMBER

)

RETURN NUMBER IS

v\_MonthlyRate NUMBER := p\_InterestRate / (12 \* 100);

v\_NumPayments NUMBER := p\_Years \* 12;

v\_Installment NUMBER;

BEGIN

v\_Installment := p\_LoanAmount \* v\_MonthlyRate / (1 - POWER(1 + v\_MonthlyRate, -v\_NumPayments));

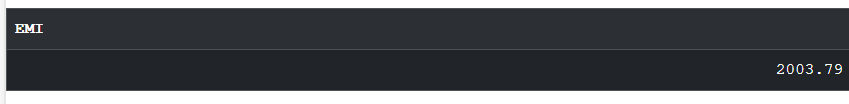
RETURN ROUND(v\_Installment, 2);

END CalculateMonthlyInstallment;

/

SELECT CalculateMonthlyInstallment(100000, 7.5, 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.

**Program**

CREATE TABLE Accounts (

AccountID NUMBER PRIMARY KEY,

CustomerID NUMBER,

AccountType VARCHAR2(20),

Balance NUMBER,

LastModified DATE

);

INSERT INTO Accounts VALUES (1, 101, 'Savings', 1000, SYSDATE);

COMMIT;

CREATE OR REPLACE FUNCTION HasSufficientBalance (

p\_AccountID IN NUMBER,

p\_Amount IN NUMBER

) RETURN BOOLEAN IS

v\_Balance NUMBER;

BEGIN

SELECT Balance INTO v\_Balance

FROM Accounts

WHERE AccountID = p\_AccountID;

RETURN v\_Balance >= p\_Amount;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN FALSE;

END HasSufficientBalance;

/

SET SERVEROUTPUT ON;

DECLARE

v\_Result BOOLEAN;

BEGIN

v\_Result := HasSufficientBalance(1, 2500);

IF v\_Result THEN

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

ELSE

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

END IF;

END;

/

**Output**



**Exercise 5: Triggers**

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

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

**Program**

CREATE TABLE Customers (

CustomerID NUMBER PRIMARY KEY,

Name VARCHAR2(100),

DOB DATE,

Balance NUMBER,

LastModified DATE

);

INSERT INTO Customers VALUES (101, 'Mughil', TO\_DATE('1985-05-15', 'YYYY-MM-DD'), 5000, SYSDATE);

COMMIT;

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON Customers

FOR EACH ROW

BEGIN

:NEW.LastModified := SYSDATE;

END;

/

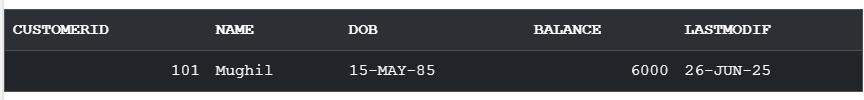
UPDATE Customers

SET Balance = Balance + 1000

WHERE CustomerID = 101;

SELECT \* FROM Customers;

**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 Transactions (

TransactionID NUMBER PRIMARY KEY,

AccountID NUMBER,

TransactionDate DATE,

Amount NUMBER,

TransactionType VARCHAR2(10)

);

CREATE TABLE AuditLog (

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

TransactionID NUMBER,

AccountID NUMBER,

Amount NUMBER,

ActionType VARCHAR2(10),

ActionTime DATE

);

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog (

TransactionID,

AccountID,

Amount,

ActionType,

ActionTime

) VALUES (

:NEW.TransactionID,

:NEW.AccountID,

:NEW.Amount,

:NEW.TransactionType,

SYSDATE

);

END;

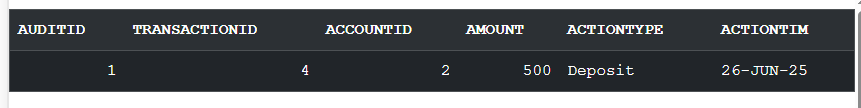
/

INSERT INTO Transactions VALUES (4, 2, SYSDATE, 500, 'Deposit');

COMMIT;

SELECT \* FROM AuditLog;

**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 TABLE Accounts (

AccountID NUMBER PRIMARY KEY,

CustomerID NUMBER,

AccountType VARCHAR2(20),

Balance NUMBER,

LastModified DATE

);

CREATE TABLE Transactions (

TransactionID NUMBER PRIMARY KEY,

AccountID NUMBER,

TransactionDate DATE,

Amount NUMBER,

TransactionType VARCHAR2(10)

);

INSERT INTO Accounts VALUES (1, 101, 'Savings', 1000, SYSDATE);

COMMIT;

INSERT INTO Transactions VALUES (1, 1, SYSDATE, 500, 'Deposit');

INSERT INTO Transactions VALUES (2, 1, SYSDATE, 2000, 'Withdrawal');

INSERT INTO Transactions VALUES (3, 1, SYSDATE, 0, 'Deposit');

COMMIT;

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

DECLARE

v\_Balance NUMBER;

BEGIN

SELECT Balance INTO v\_Balance

FROM Accounts

WHERE AccountID = :NEW.AccountID;

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

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

END IF;

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

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

END IF;

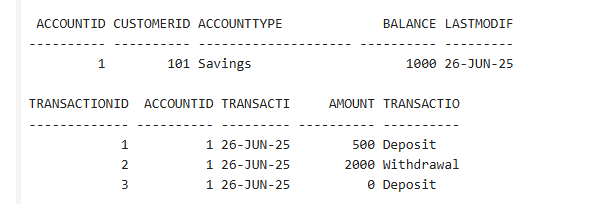
END;

/

SELECT \* FROM Accounts;

SELECT \* FROM Transactions;

**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.

**Program**

CREATE TABLE Customers (

CustomerID NUMBER PRIMARY KEY,

Name VARCHAR2(100),

DOB DATE,

Balance NUMBER,

LastModified DATE

);

CREATE TABLE Accounts (

AccountID NUMBER PRIMARY KEY,

CustomerID NUMBER,

AccountType VARCHAR2(20),

Balance NUMBER,

LastModified DATE,

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

CREATE TABLE Transactions (

TransactionID NUMBER PRIMARY KEY,

AccountID NUMBER,

TransactionDate DATE,

Amount NUMBER,

TransactionType VARCHAR2(10),

FOREIGN KEY (AccountID) REFERENCES Accounts(AccountID)

);

INSERT INTO Customers VALUES (101, 'John Doe', TO\_DATE('1985-06-25', 'YYYY-MM-DD'), 5000, SYSDATE);

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

INSERT INTO Transactions VALUES (1, 1, SYSDATE, 1000, 'Deposit');

INSERT INTO Transactions VALUES (2, 1, SYSDATE, 500, 'Withdrawal');

COMMIT;

DECLARE

CURSOR txn\_cursor IS

SELECT c.Name, t.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 TO\_CHAR(t.TransactionDate, 'MMYYYY') = TO\_CHAR(SYSDATE, 'MMYYYY')

ORDER BY c.Name;

v\_Name Customers.Name%TYPE;

v\_AccountID Accounts.AccountID%TYPE;

v\_Date Transactions.TransactionDate%TYPE;

v\_Amount Transactions.Amount%TYPE;

v\_Type Transactions.TransactionType%TYPE;

BEGIN

FOR rec IN txn\_cursor LOOP

DBMS\_OUTPUT.PUT\_LINE('Customer: ' || rec.Name ||

' | AccID: ' || rec.AccountID ||

' | Date: ' || TO\_CHAR(rec.TransactionDate, 'DD-Mon-YYYY') ||

' | Amount: ' || rec.Amount ||

' | Type: ' || rec.TransactionType);

END LOOP;

END;

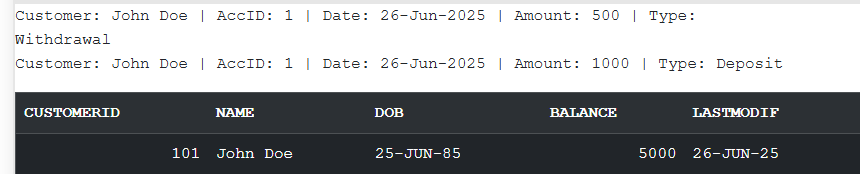
/

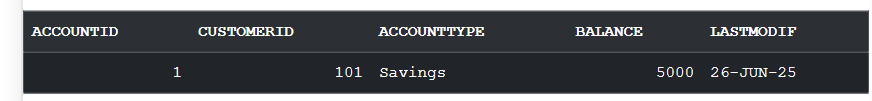
SELECT \* FROM Customers;

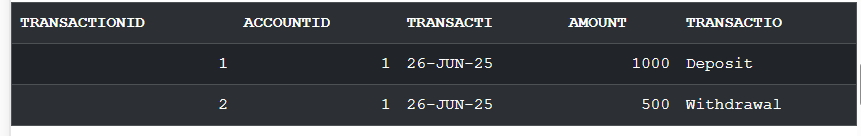
SELECT \* FROM Accounts;

SELECT \* FROM Transactions;

**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**

CREATE TABLE Accounts (

AccountID NUMBER PRIMARY KEY,

CustomerID NUMBER,

AccountType VARCHAR2(20),

Balance NUMBER,

LastModified DATE

);

INSERT INTO Accounts VALUES (2, 101, 'Savings', 5000, SYSDATE);

INSERT INTO Accounts VALUES (1, 102, 'Checking', 3000, SYSDATE);

INSERT INTO Accounts VALUES (3, 103, 'Withdraw', 800, SYSDATE);

COMMIT;

SET SERVEROUTPUT ON;

DECLARE

CURSOR acc\_cursor IS

SELECT AccountID FROM Accounts;

BEGIN

FOR rec IN acc\_cursor LOOP

UPDATE Accounts

SET Balance = Balance - 100,

LastModified = SYSDATE

WHERE AccountID = rec.AccountID;

DBMS\_OUTPUT.PUT\_LINE('Annual fee applied to Account ID: ' || rec.AccountID);

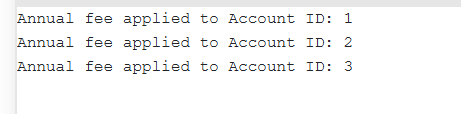
END LOOP;

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.

**Program**

CREATE TABLE Loans (

LoanID NUMBER PRIMARY KEY,

CustomerID NUMBER,

LoanAmount NUMBER,

InterestRate NUMBER,

StartDate DATE,

EndDate DATE

);

INSERT INTO Loans VALUES (1, 101, 50000, 12, SYSDATE, ADD\_MONTHS(SYSDATE, 60));

INSERT INTO Loans VALUES (2, 102, 30000, 6, SYSDATE, ADD\_MONTHS(SYSDATE, 36));

INSERT INTO Loans VALUES (3, 103, 100000, 4, SYSDATE, ADD\_MONTHS(SYSDATE, 120));

COMMIT;

SET SERVEROUTPUT ON;

DECLARE

CURSOR loan\_cursor IS

SELECT LoanID, InterestRate FROM Loans;

v\_NewRate NUMBER;

BEGIN

FOR rec IN loan\_cursor LOOP

IF rec.InterestRate > 10 THEN

v\_NewRate := rec.InterestRate - 1;

ELSIF rec.InterestRate < 5 THEN

v\_NewRate := rec.InterestRate + 0.5;

ELSE

v\_NewRate := rec.InterestRate;

END IF;

UPDATE Loans

SET InterestRate = v\_NewRate

WHERE LoanID = rec.LoanID;

DBMS\_OUTPUT.PUT\_LINE('Loan ID: ' || rec.LoanID ||

' | Old Rate: ' || rec.InterestRate ||

' | New Rate: ' || v\_NewRate);

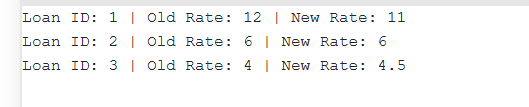
END LOOP;

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.

**Program**

CREATE TABLE Customers (

CustomerID NUMBER PRIMARY KEY,

Name VARCHAR2(100),

DOB DATE,

Balance NUMBER,

LastModified DATE

);

/

CREATE OR REPLACE PACKAGE CustomerManagement AS

PROCEDURE AddNewCustomer(p\_CustomerID NUMBER, p\_Name VARCHAR2, p\_DOB DATE, p\_Balance NUMBER);

PROCEDURE UpdateCustomerDetails(p\_CustomerID NUMBER, p\_Name VARCHAR2, p\_DOB DATE);

FUNCTION GetCustomerBalance(p\_CustomerID NUMBER) RETURN NUMBER;

END CustomerManagement;

/

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

PROCEDURE AddNewCustomer(p\_CustomerID NUMBER, p\_Name VARCHAR2, p\_DOB DATE, p\_Balance NUMBER) IS

BEGIN

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

VALUES (p\_CustomerID, p\_Name, p\_DOB, p\_Balance, SYSDATE);

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

END AddNewCustomer;

PROCEDURE UpdateCustomerDetails(p\_CustomerID NUMBER, p\_Name VARCHAR2, p\_DOB DATE) IS

BEGIN

UPDATE Customers

SET Name = p\_Name, DOB = p\_DOB, LastModified = SYSDATE

WHERE CustomerID = p\_CustomerID;

END UpdateCustomerDetails;

FUNCTION GetCustomerBalance(p\_CustomerID NUMBER) RETURN NUMBER IS

v\_Balance NUMBER;

BEGIN

SELECT Balance INTO v\_Balance FROM Customers WHERE CustomerID = p\_CustomerID;

RETURN v\_Balance;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN NULL;

END GetCustomerBalance;

END CustomerManagement;

/

BEGIN

CustomerManagement.AddNewCustomer(1, 'Prabha', TO\_DATE('1985-05-15', 'YYYY-MM-DD'), 1000);

END;

/

SET SERVEROUTPUT ON;

DECLARE

v\_balance NUMBER;

BEGIN

v\_balance := CustomerManagement.GetCustomerBalance(1);

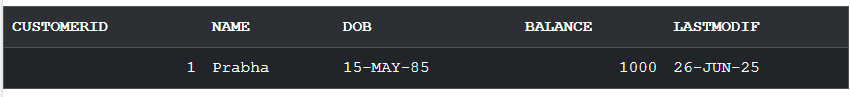
DBMS\_OUTPUT.PUT\_LINE('Customer Balance: ' || v\_balance);

END;

/

SELECT \* FROM Customers;

**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.

**Program**

CREATE TABLE Customers (

CustomerID NUMBER PRIMARY KEY,

Name VARCHAR2(100),

DOB DATE,

Balance NUMBER,

LastModified DATE

);

CREATE OR REPLACE PACKAGE EmployeeManagement AS

PROCEDURE HireEmployee(p\_EmployeeID NUMBER, p\_Name VARCHAR2, p\_Position VARCHAR2, p\_Salary NUMBER, p\_Department VARCHAR2);

PROCEDURE UpdateEmployeeDetails(p\_EmployeeID NUMBER, p\_Position VARCHAR2, p\_Salary NUMBER);

FUNCTION CalculateAnnualSalary(p\_EmployeeID NUMBER) RETURN NUMBER;

END EmployeeManagement;

/

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

PROCEDURE HireEmployee(p\_EmployeeID NUMBER, p\_Name VARCHAR2, p\_Position VARCHAR2, p\_Salary NUMBER, p\_Department VARCHAR2) IS

BEGIN

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

VALUES (p\_EmployeeID, p\_Name, p\_Position, p\_Salary, p\_Department, SYSDATE);

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

DBMS\_OUTPUT.PUT\_LINE('Employee with ID ' || p\_EmployeeID || ' already exists.');

END HireEmployee;

PROCEDURE UpdateEmployeeDetails(p\_EmployeeID NUMBER, p\_Position VARCHAR2, p\_Salary NUMBER) IS

BEGIN

UPDATE Employees

SET Position = p\_Position, Salary = p\_Salary

WHERE EmployeeID = p\_EmployeeID;

END UpdateEmployeeDetails;

FUNCTION CalculateAnnualSalary(p\_EmployeeID NUMBER) RETURN NUMBER IS

v\_Salary NUMBER;

BEGIN

SELECT Salary INTO v\_Salary FROM Employees WHERE EmployeeID = p\_EmployeeID;

RETURN v\_Salary \* 12;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN NULL;

END CalculateAnnualSalary;

END EmployeeManagement;

/

CREATE TABLE Employees (

EmployeeID NUMBER PRIMARY KEY,

Name VARCHAR2(100),

Position VARCHAR2(50),

Salary NUMBER,

Department VARCHAR2(50),

HireDate DATE

);

/

BEGIN

EmployeeManagement.HireEmployee(1, 'Alice', 'Manager', 70000, 'HR');

END;

/

BEGIN

EmployeeManagement.UpdateEmployeeDetails(1, 'Senior Manager', 80000);

END;

/

SET SERVEROUTPUT ON;

DECLARE

v\_annual\_salary NUMBER;

BEGIN

v\_annual\_salary := EmployeeManagement.CalculateAnnualSalary(1);

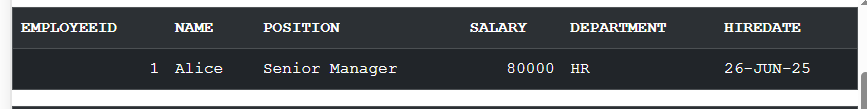
DBMS\_OUTPUT.PUT\_LINE('Annual Salary: ' || v\_annual\_salary);

END;

/

SELECT \* FROM Employees;

**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**

CREATE TABLE Customers (

CustomerID NUMBER PRIMARY KEY,

Name VARCHAR2(100),

DOB DATE,

Balance NUMBER,

LastModified DATE

);

/

-- Accounts table

CREATE TABLE Accounts (

AccountID NUMBER PRIMARY KEY,

CustomerID NUMBER,

AccountType VARCHAR2(20),

Balance NUMBER,

LastModified DATE,

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

/

CREATE OR REPLACE PACKAGE AccountOperations AS

PROCEDURE OpenAccount(p\_AccountID NUMBER, p\_CustomerID NUMBER, p\_AccountType VARCHAR2, p\_Balance NUMBER);

PROCEDURE CloseAccount(p\_AccountID NUMBER);

FUNCTION GetTotalBalance(p\_CustomerID NUMBER) RETURN NUMBER;

END AccountOperations;

/

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

PROCEDURE OpenAccount(p\_AccountID NUMBER, p\_CustomerID NUMBER, p\_AccountType VARCHAR2, p\_Balance NUMBER) IS

BEGIN

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

VALUES (p\_AccountID, p\_CustomerID, p\_AccountType, p\_Balance, SYSDATE);

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

DBMS\_OUTPUT.PUT\_LINE('Account with ID ' || p\_AccountID || ' already exists.');

END OpenAccount;

PROCEDURE CloseAccount(p\_AccountID NUMBER) IS

BEGIN

DELETE FROM Accounts WHERE AccountID = p\_AccountID;

END CloseAccount;

FUNCTION GetTotalBalance(p\_CustomerID NUMBER) RETURN NUMBER IS

v\_TotalBalance NUMBER;

BEGIN

SELECT SUM(Balance) INTO v\_TotalBalance FROM Accounts WHERE CustomerID = p\_CustomerID;

RETURN NVL(v\_TotalBalance, 0);

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN 0;

END GetTotalBalance;

END AccountOperations;

/

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

VALUES (1, 'John Doe', TO\_DATE('1985-05-15', 'YYYY-MM-DD'), 1000, SYSDATE);

/

BEGIN

AccountOperations.OpenAccount(1, 1, 'Savings', 500);

AccountOperations.OpenAccount(2, 1, 'Checking', 700);

END;

/

SET SERVEROUTPUT ON;

DECLARE

v\_total\_balance NUMBER;

BEGIN

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

DBMS\_OUTPUT.PUT\_LINE('Total Balance: ' || v\_total\_balance);

END;

/

SELECT \* FROM Accounts;

**Output**

