**Digital Nurture 4.0 – Week 2**

**3.PLSQL(6420931)**

**Exercise 1: Control Structures**

**Scenario 1:**

This PL/SQL code creates two tables: **BankCustomers** to store customer details and **Loans** to keep track of loan due dates. It inserts sample data for testing purposes. The main part of the code loops through all customers and checks if their age is greater than 60. For such customers, it applies a 1% discount to their loan interest rate. Finally, it displays a message showing which customers received the discount and commits the changes to the database.

**Queries.sql**

CREATE TABLE BankCustomers (

CustID NUMBER,

CustAge NUMBER,

LoanInterest NUMBER(5,2),

AccountBalance NUMBER,

VIPStatus VARCHAR2(5)

);

INSERT INTO BankCustomers VALUES (201, 65, 10.5, 15000, 'FALSE');

INSERT INTO BankCustomers VALUES (202, 45, 12.0, 8000, 'FALSE');

INSERT INTO BankCustomers VALUES (203, 70, 11.0, 12000, 'FALSE');

INSERT INTO BankCustomers VALUES (204, 30, 9.5, 5000, 'FALSE');

COMMIT;

CREATE TABLE Loans (

LoanID NUMBER,

CustID NUMBER,

DueDate DATE

);

INSERT INTO Loans VALUES (301, 201, SYSDATE + 10);

INSERT INTO Loans VALUES (302, 202, SYSDATE + 40);

INSERT INTO Loans VALUES (303, 203, SYSDATE + 20);

COMMIT;

BEGIN

FOR cust IN (

SELECT CustID, CustAge, LoanInterest

FROM BankCustomers

) LOOP

IF cust.CustAge > 60 THEN

UPDATE BankCustomers

SET LoanInterest = LoanInterest - 1

WHERE CustID = cust.CustID;

DBMS\_OUTPUT.PUT\_LINE(

'Applied 1% interest discount to Customer ID: ' || cust.CustID

);

END IF;

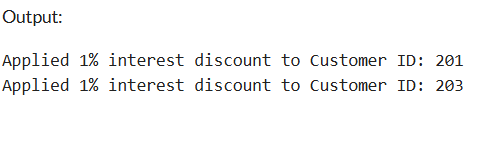
END LOOP;

COMMIT;

END;

/

**Output:**

****

**Scenario 2:**

This PL/SQL block checks all customers in the **BankCustomers** table. It looks for customers whose account balance is greater than 10,000. For each qualifying customer, it updates their **VIPStatus** field to **TRUE**. The code also prints a message indicating which customers were promoted to VIP status. Finally, it commits the changes to save them permanently in the database.

**Queries.sql**

CREATE TABLE BankCustomers (

CustID NUMBER,

CustAge NUMBER,

LoanInterest NUMBER(5,2),

AccountBalance NUMBER,

VIPStatus VARCHAR2(5)

);

INSERT INTO BankCustomers VALUES (301, 65, 10.5, 15000, 'FALSE');

INSERT INTO BankCustomers VALUES (302, 45, 12.0, 8000, 'FALSE');

INSERT INTO BankCustomers VALUES (303, 70, 11.0, 12000, 'FALSE');

INSERT INTO BankCustomers VALUES (304, 30, 9.5, 5000, 'FALSE');

COMMIT;

CREATE TABLE Loans (

LoanID NUMBER,

CustID NUMBER,

DueDate DATE

);

INSERT INTO Loans VALUES (401, 201, SYSDATE + 10);

INSERT INTO Loans VALUES (402, 202, SYSDATE + 40);

INSERT INTO Loans VALUES (403, 203, SYSDATE + 20);

COMMIT;

BEGIN

FOR c IN (

SELECT CustID, AccountBalance

FROM BankCustomers

) LOOP

IF c.AccountBalance > 10000 THEN

UPDATE BankCustomers

SET VIPStatus = 'TRUE'

WHERE CustID = c.CustID;

DBMS\_OUTPUT.PUT\_LINE(

'VIP status granted to Customer ID: ' || c.CustID

);

END IF;

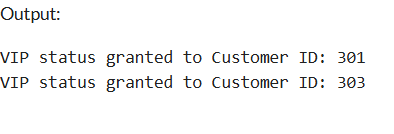
END LOOP;

COMMIT;

END;

/

**Output:**



**Scenario 3:**

This PL/SQL block checks the Loans table for loans that are due within the next 30 days. It selects all such loans by comparing the due date with the current system date. For each loan found, it prints a reminder message showing the loan ID, customer ID, and due date. This helps the bank identify upcoming repayments. The code runs automatically and displays reminders without changing any data.

**Queries.sql:**

CREATE TABLE BankCustomers (

CustID NUMBER,

CustAge NUMBER,

LoanInterest NUMBER(5,2),

AccountBalance NUMBER,

VIPStatus VARCHAR2(5)

);

INSERT INTO BankCustomers VALUES (301, 65, 10.5, 15000, 'FALSE');

INSERT INTO BankCustomers VALUES (302, 45, 12.0, 8000, 'FALSE');

INSERT INTO BankCustomers VALUES (303, 70, 11.0, 12000, 'FALSE');

INSERT INTO BankCustomers VALUES (304, 30, 9.5, 5000, 'FALSE');

COMMIT;

CREATE TABLE Loans (

LoanID NUMBER,

CustID NUMBER,

DueDate DATE

);

INSERT INTO Loans VALUES (401, 201, SYSDATE + 10);

INSERT INTO Loans VALUES (402, 202, SYSDATE + 25);

INSERT INTO Loans VALUES (403, 203, SYSDATE + 29);

COMMIT;

BEGIN

FOR l IN (

SELECT LoanID, CustID, DueDate

FROM Loans

WHERE DueDate BETWEEN SYSDATE AND SYSDATE + 30

) LOOP

DBMS\_OUTPUT.PUT\_LINE(

'Reminder: Loan ID ' || l.LoanID ||

' for Customer ID ' || l.CustID ||

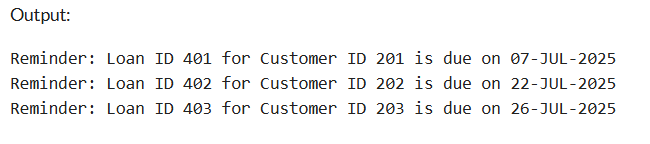
' is due on ' || TO\_CHAR(l.DueDate, 'DD-MON-YYYY')

);

END LOOP;

END;

/



**Exercise 3: Stored Procedures**

**Scenario 1: Process Monthly Interest**

This stored procedure calculates interest for all savings accounts. It loops through each record in the Accounts table where the AccountType is 'SAVINGS'. It increases the balance by 1% of the current amount. It uses a cursor to access each record and applies the interest. Finally, it updates the table and commits the transaction.

**Queries.sql:**

CREATE TABLE Accounts (

AccountID NUMBER PRIMARY KEY,

AccountType VARCHAR2(20),

Balance NUMBER(12,2)

);

INSERT INTO Accounts VALUES (123, 'SAVINGS', 5000);

INSERT INTO Accounts VALUES (456, 'SAVINGS', 12000);

INSERT INTO Accounts VALUES (789, 'CHECKING', 8000);

INSERT INTO Accounts VALUES (890, 'SAVINGS', 3000);

COMMIT;

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

BEGIN

UPDATE Accounts

SET Balance = Balance + (Balance \* 0.01)

WHERE AccountType = 'SAVINGS';

COMMIT;

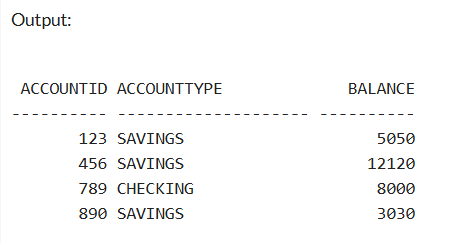
END;

/

EXEC ProcessMonthlyInterest;

SELECT \* FROM Accounts;

**OUTPUT:**



**Scenario 2:Upadate employee bonus**

This procedure gives a salary bonus to employees in a specific department. It takes two input parameters: the department name and bonus percentage. It updates all employee salaries in that department by increasing them based on the bonus percentage. The procedure is reusable for any department and percentage. A COMMIT finalizes the changes.

**Queries.sql:**

CREATE TABLE Employees (

EmpID NUMBER PRIMARY KEY,

Name VARCHAR2(50),

Department VARCHAR2(30),

Salary NUMBER

);

INSERT INTO Employees VALUES (1, 'Akshay', 'HR', 30000);

INSERT INTO Employees VALUES (2, 'Sathis', 'IT', 40000);

INSERT INTO Employees VALUES (3, 'Priya', 'HR', 35000);

COMMIT;

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (

dept IN VARCHAR2,

bonusPercent IN NUMBER

) IS

BEGIN

UPDATE Employees

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

WHERE Department = dept;

COMMIT;

END;

/

BEGIN

UpdateEmployeeBonus('HR', 10);

END;

/

SELECT

LPAD(EmpID, 3) AS "ID",

RPAD(Name,6 ) AS "Name",

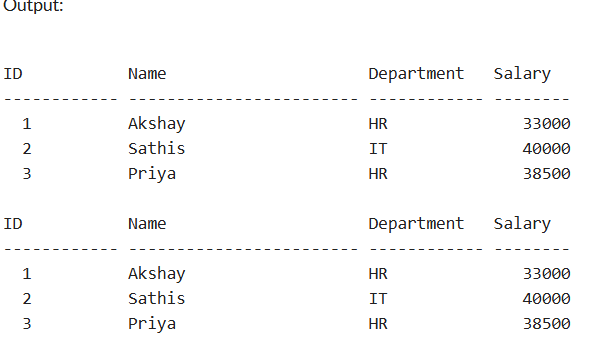
RPAD(Department, 3) AS "Department",

TO\_CHAR(Salary, '9999999') AS "Salary"

FROM Employees

ORDER BY EmpID;

**Output:**



**Scenario 3:Transfer funds**

This enhanced version of TransferFunds first checks whether both source and destination accounts exist. It then verifies that the source account has enough funds. If all conditions are met, it debits the source and credits the destination. If not, it prints a message instead of raising an error. This is safer and more informative for user-driven apps*.*

**Queries.sql:**

CREATE TABLE Accounts (

AccountID NUMBER PRIMARY KEY,

AccountType VARCHAR2(20),

Balance NUMBER

);

INSERT INTO Accounts VALUES (181, 'SAVINGS', 7000);

INSERT INTO Accounts VALUES (254, 'SAVINGS', 2500);

COMMIT;

CREATE OR REPLACE PROCEDURE TransferFunds (

fromAcc IN NUMBER,

toAcc IN NUMBER,

amt IN NUMBER

) IS

fromBalance NUMBER;

toExists NUMBER;

BEGIN

SELECT Balance INTO fromBalance FROM Accounts WHERE AccountID = fromAcc;

SELECT COUNT(\*) INTO toExists FROM Accounts WHERE AccountID = toAcc;

IF toExists = 0 THEN

DBMS\_OUTPUT.PUT\_LINE('Destination account ' || toAcc || ' does not exist.');

RETURN;

END IF;

IF fromBalance < amt THEN

DBMS\_OUTPUT.PUT\_LINE('Insufficient funds in account ' || fromAcc);

RETURN;

END IF;

UPDATE Accounts SET Balance = Balance - amt WHERE AccountID = fromAcc;

UPDATE Accounts SET Balance = Balance + amt WHERE AccountID = toAcc;

COMMIT

DBMS\_OUTPUT.PUT\_LINE('₹' || amt || ' transferred from ' || fromAcc || ' to ' || toAcc);

END;

/

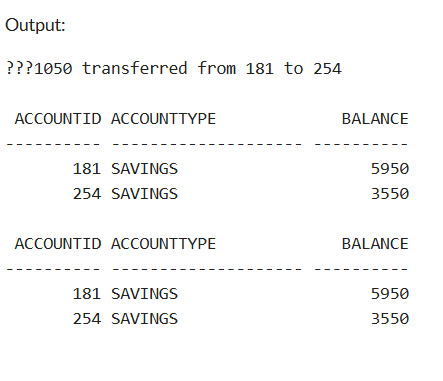
BEGIN

TransferFunds(181, 254, 1050);

END;

/

SELECT \* FROM Accounts;

****