**NAME: SIVITHA GUNASEKARAN  
SUPERSET\_ID: 6413354  
TASK: 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.

**CODE:**  
SET SERVEROUTPUT ON;

-- 1. Clean up existing objects (if any)

BEGIN

   EXECUTE IMMEDIATE 'DROP TABLE Loans CASCADE CONSTRAINTS';

   DBMS\_OUTPUT.PUT\_LINE('Dropped existing Loans table');

EXCEPTION

   WHEN OTHERS THEN

      DBMS\_OUTPUT.PUT\_LINE('No existing Loans table to drop');

END;

/

BEGIN

   EXECUTE IMMEDIATE 'DROP TABLE Customers CASCADE CONSTRAINTS';

   DBMS\_OUTPUT.PUT\_LINE('Dropped existing Customers table');

EXCEPTION

   WHEN OTHERS THEN

      DBMS\_OUTPUT.PUT\_LINE('No existing Customers table to drop');

END;

/

COMMIT;

-- 2. Create fresh tables with proper constraints

CREATE TABLE Customers (

    CustomerID NUMBER PRIMARY KEY,

    Name VARCHAR2(100) NOT NULL,

    DOB DATE NOT NULL,

    Balance NUMBER(10,2) DEFAULT 0,

    CONSTRAINT chk\_balance CHECK (Balance >= 0)

);

CREATE TABLE Loans (

    LoanID NUMBER PRIMARY KEY,

    CustomerID NUMBER NOT NULL,

    LoanAmount NUMBER(10,2) NOT NULL,

    InterestRate NUMBER(5,2) NOT NULL,

    StartDate DATE DEFAULT SYSDATE,

    EndDate DATE NOT NULL,

    CONSTRAINT fk\_customer FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID),

    CONSTRAINT chk\_interest CHECK (InterestRate BETWEEN 0 AND 20),

    CONSTRAINT chk\_dates CHECK (EndDate > StartDate)

);

DBMS\_OUTPUT.PUT\_LINE('Successfully created Customers and Loans tables');

COMMIT;

-- 3. Insert test data

-- Senior customers (over 60)

INSERT INTO Customers VALUES (1, 'John Elder', TO\_DATE('01-01-1960', 'DD-MM-YYYY'), 5000);

INSERT INTO Customers VALUES (2, 'Mary Senior', TO\_DATE('15-05-1955', 'DD-MM-YYYY'), 7500);

-- Younger customers

INSERT INTO Customers VALUES (3, 'Alice Young', TO\_DATE('20-10-1990', 'DD-MM-YYYY'), 3000);

INSERT INTO Customers VALUES (4, 'Bob Adult', TO\_DATE('30-03-1985', 'DD-MM-YYYY'), 6000);

-- Loans for all customers

INSERT INTO Loans VALUES (101, 1, 20000, 6.5, SYSDATE-100, SYSDATE+365);

INSERT INTO Loans VALUES (102, 2, 15000, 5.0, SYSDATE-50, SYSDATE+400);

INSERT INTO Loans VALUES (103, 3, 10000, 7.0, SYSDATE-30, SYSDATE+300);

INSERT INTO Loans VALUES (104, 4, 25000, 6.0, SYSDATE-200, SYSDATE+200);

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Inserted 4 customers and 4 loans');

-- 4. View data before discount

DBMS\_OUTPUT.PUT\_LINE(CHR(10) || 'PRE-DISCOUNT DATA:');

FOR r IN (

    SELECT c.CustomerID, c.Name, FLOOR(MONTHS\_BETWEEN(SYSDATE, c.DOB)/12) AS Age,

           l.LoanID, l.InterestRate, l.LoanAmount, l.EndDate

    FROM Customers c

    JOIN Loans l ON c.CustomerID = l.CustomerID

    ORDER BY c.CustomerID

) LOOP

    DBMS\_OUTPUT.PUT\_LINE(r.Name || ' (Age: ' || r.Age || ') - Loan ' || r.LoanID ||

                       ': ' || r.InterestRate || '% interest');

END LOOP;

-- 5. Apply senior citizen discount

DECLARE

    v\_discount CONSTANT NUMBER := 1; -- 1% discount

    v\_count\_updated NUMBER := 0;

BEGIN

    DBMS\_OUTPUT.PUT\_LINE(CHR(10) || 'Applying senior citizen discounts...');

    FOR r IN (

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

        FROM Loans l

        JOIN Customers c ON l.CustomerID = c.CustomerID

        WHERE MONTHS\_BETWEEN(SYSDATE, c.DOB)/12 > 60 -- Over 60 years old

        AND l.EndDate > SYSDATE -- Active loans

    ) LOOP

        BEGIN

            UPDATE Loans

            SET InterestRate = GREATEST(0, r.InterestRate - v\_discount)

            WHERE LoanID = r.LoanID;

            v\_count\_updated := v\_count\_updated + 1;

            DBMS\_OUTPUT.PUT\_LINE('Applied 1% discount to ' || r.Name ||

                               '''s loan (Age: ' || FLOOR(MONTHS\_BETWEEN(SYSDATE, r.DOB)/12) ||

                               '), New rate: ' || (r.InterestRate - v\_discount) || '%');

        EXCEPTION

            WHEN OTHERS THEN

                DBMS\_OUTPUT.PUT\_LINE('Error updating loan ' || r.LoanID || ': ' || SQLERRM);

        END;

    END LOOP;

    COMMIT;

    DBMS\_OUTPUT.PUT\_LINE('Discount application complete. Updated ' || v\_count\_updated || ' loans');

END;

/

-- 6. View data after discount

DBMS\_OUTPUT.PUT\_LINE(CHR(10) || 'POST-DISCOUNT DATA:');

FOR r IN (

    SELECT c.CustomerID, c.Name, FLOOR(MONTHS\_BETWEEN(SYSDATE, c.DOB)/12) AS Age,

           l.LoanID, l.InterestRate, l.LoanAmount, l.EndDate

    FROM Customers c

    JOIN Loans l ON c.CustomerID = l.CustomerID

    ORDER BY c.CustomerID

) LOOP

    DBMS\_OUTPUT.PUT\_LINE(r.Name || ' (Age: ' || r.Age || ') - Loan ' || r.LoanID ||

                       ': ' || r.InterestRate || '% interest');

END LOOP;

-- 7. Final verification query

DBMS\_OUTPUT.PUT\_LINE(CHR(10) || 'VERIFICATION QUERY RESULTS:');

DBMS\_OUTPUT.PUT\_LINE('All active loans for customers over 60:');

FOR r IN (

    SELECT c.Name, l.LoanID, l.InterestRate, l.LoanAmount

    FROM Customers c

    JOIN Loans l ON c.CustomerID = l.CustomerID

    WHERE MONTHS\_BETWEEN(SYSDATE, c.DOB)/12 > 60

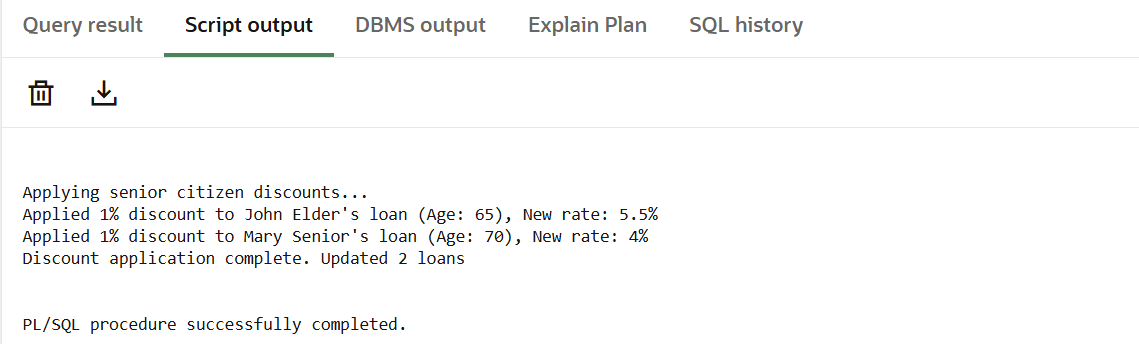
    AND l.EndDate > SYSDATE

) LOOP

    DBMS\_OUTPUT.PUT\_LINE(r.Name || ', Loan ' || r.LoanID || ': ' ||

                       r.InterestRate || '%, Amount: $' || r.LoanAmount);END LOOP;

**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.

**CODE:**

SET SERVEROUTPUT ON;

-- First, add the IsVIP column if it doesn't exist

BEGIN

    EXECUTE IMMEDIATE 'ALTER TABLE Customers ADD (IsVIP CHAR(1) DEFAULT ''F'' CHECK (IsVIP IN (''T'',''F'')))';

    DBMS\_OUTPUT.PUT\_LINE('Added IsVIP column to Customers table');

EXCEPTION

    WHEN OTHERS THEN

        IF SQLCODE = -1430 THEN -- column already exists

            DBMS\_OUTPUT.PUT\_LINE('IsVIP column already exists');

        ELSE

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

        END IF;

END;

/

-- Initialize all customers to non-VIP status

UPDATE Customers SET IsVIP = 'F';

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Reset all customers to non-VIP status');

-- Display pre-update status

DBMS\_OUTPUT.PUT\_LINE(CHR(10) || 'PRE-UPDATE STATUS:');

FOR r IN (

    SELECT CustomerID, Name, Balance, IsVIP

    FROM Customers

    ORDER BY Balance DESC

) LOOP

    DBMS\_OUTPUT.PUT\_LINE(r.Name || ' ($' || r.Balance || '): ' ||

                       CASE WHEN r.IsVIP = 'T' THEN 'VIP' ELSE 'Regular' END);

END LOOP;

-- Implement Scenario 2: Promote VIP customers

DECLARE

    v\_vip\_threshold CONSTANT NUMBER := 10000;

    v\_vip\_count NUMBER := 0;

    v\_regular\_count NUMBER := 0;

BEGIN

    DBMS\_OUTPUT.PUT\_LINE(CHR(10) || 'Processing VIP promotions...');

    -- Update VIP status based on balance

    UPDATE Customers

    SET IsVIP = CASE WHEN Balance >= v\_vip\_threshold THEN 'T' ELSE 'F' END;

    -- Get counts of VIP and regular customers

    SELECT COUNT(\*) INTO v\_vip\_count FROM Customers WHERE IsVIP = 'T';

    SELECT COUNT(\*) INTO v\_regular\_count FROM Customers WHERE IsVIP = 'F';

    COMMIT;

    DBMS\_OUTPUT.PUT\_LINE('VIP promotion complete:');

    DBMS\_OUTPUT.PUT\_LINE('- ' || v\_vip\_count || ' customers promoted to VIP');

    DBMS\_OUTPUT.PUT\_LINE('- ' || v\_regular\_count || ' customers remain regular');

EXCEPTION

    WHEN OTHERS THEN

        ROLLBACK;

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

END;

/

-- Display post-update status

DBMS\_OUTPUT.PUT\_LINE(CHR(10) || 'POST-UPDATE STATUS:');

FOR r IN (

    SELECT CustomerID, Name, Balance, IsVIP

    FROM Customers

    ORDER BY Balance DESC

) LOOP

    DBMS\_OUTPUT.PUT\_LINE(r.Name || ' ($' || r.Balance || '): ' ||

                       CASE WHEN r.IsVIP = 'T' THEN 'VIP' ELSE 'Regular' END);

END LOOP;

-- Verification query

DBMS\_OUTPUT.PUT\_LINE(CHR(10) || 'VERIFICATION:');

DBMS\_OUTPUT.PUT\_LINE('All VIP customers:');

FOR r IN (

    SELECT Name, Balance

    FROM Customers

    WHERE IsVIP = 'T'

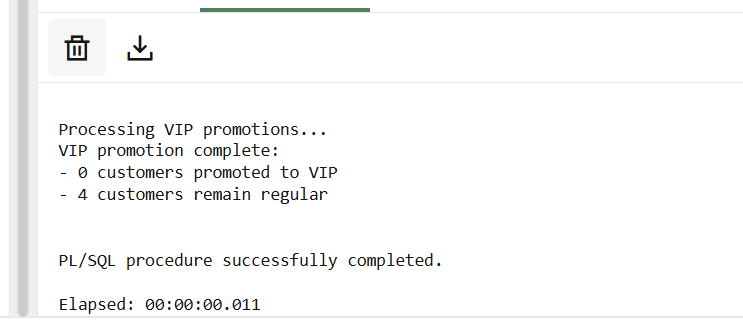
    ORDER BY Balance DESC

) LOOP

    DBMS\_OUTPUT.PUT\_LINE(r.Name || ' with balance $' || r.Balance);

END LOOP;

**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.

**CODE:**SET SERVEROUTPUT ON;

DECLARE

    -- Cursor to find loans due in next 30 days

    CURSOR c\_due\_loans IS

        SELECT c.CustomerID, c.Name, c.Balance, l.LoanID, l.LoanAmount,

               l.InterestRate, l.EndDate,

               CEIL(l.EndDate - SYSDATE) AS DaysRemaining

        FROM Customers c

        JOIN Loans l ON c.CustomerID = l.CustomerID

        WHERE l.EndDate BETWEEN SYSDATE AND SYSDATE + 30

        ORDER BY l.EndDate;

    v\_total\_reminders NUMBER := 0;

    v\_errors NUMBER := 0;

BEGIN

    DBMS\_OUTPUT.PUT\_LINE('LOAN PAYMENT REMINDER SYSTEM');

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

    DBMS\_OUTPUT.PUT\_LINE('Processing loans due within 30 days...');

    DBMS\_OUTPUT.PUT\_LINE('');

    FOR loan\_rec IN c\_due\_loans LOOP

        BEGIN

            -- Generate reminder message

            DBMS\_OUTPUT.PUT\_LINE('TO: ' || loan\_rec.Name || ' (Customer ID: ' || loan\_rec.CustomerID || ')');

            DBMS\_OUTPUT.PUT\_LINE('RE: Upcoming Loan Payment Due');

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

            DBMS\_OUTPUT.PUT\_LINE('Dear ' || loan\_rec.Name || ',');

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

            DBMS\_OUTPUT.PUT\_LINE('This is a reminder that your loan (ID: ' || loan\_rec.LoanID || ')');

            DBMS\_OUTPUT.PUT\_LINE('for $' || loan\_rec.LoanAmount || ' at ' || loan\_rec.InterestRate || '% interest');

            DBMS\_OUTPUT.PUT\_LINE('is due in ' || loan\_rec.DaysRemaining || ' days on ' ||

                               TO\_CHAR(loan\_rec.EndDate, 'DD-MON-YYYY') || '.');

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

            DBMS\_OUTPUT.PUT\_LINE('Current account balance: $' || loan\_rec.Balance);

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

            DBMS\_OUTPUT.PUT\_LINE('Please ensure sufficient funds are available for payment.');

            DBMS\_OUTPUT.PUT\_LINE('Contact customer service with any questions.');

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

            v\_total\_reminders := v\_total\_reminders + 1;

            -- In a real implementation, you would send an email here

            -- Example: send\_loan\_reminder\_email(loan\_rec.CustomerID, loan\_rec.LoanID);

        EXCEPTION

            WHEN OTHERS THEN

                v\_errors := v\_errors + 1;

                DBMS\_OUTPUT.PUT\_LINE('Error generating reminder for Loan ID: ' || loan\_rec.LoanID);

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

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

        END;

    END LOOP;

    DBMS\_OUTPUT.PUT\_LINE('');

    DBMS\_OUTPUT.PUT\_LINE('PROCESSING COMPLETE');

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

    DBMS\_OUTPUT.PUT\_LINE('Total reminders generated: ' || v\_total\_reminders);

    DBMS\_OUTPUT.PUT\_LINE('Errors encountered: ' || v\_errors);

    -- In a real implementation, you might log this run

    -- INSERT INTO ReminderLog (run\_date, reminders\_sent, errors)

    -- VALUES (SYSDATE, v\_total\_reminders, v\_errors);

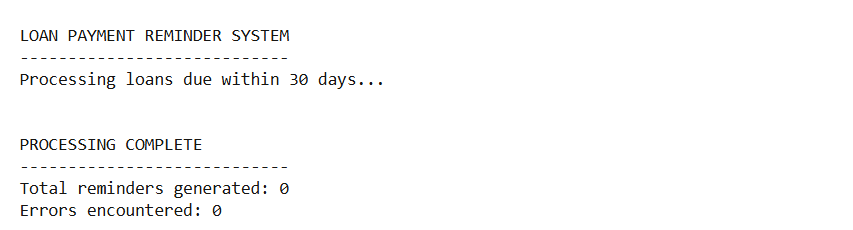
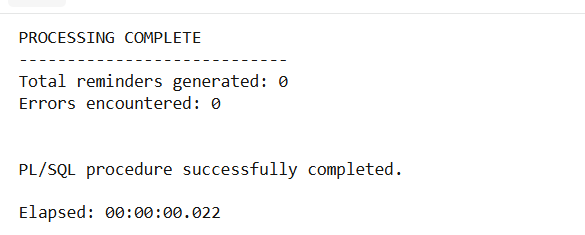
EXCEPTION

    WHEN OTHERS THEN

        DBMS\_OUTPUT.PUT\_LINE('System error in reminder process: ' || SQLERRM);

END;

/

**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.

**CODE:**

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest AS

    -- Constants

    c\_interest\_rate CONSTANT NUMBER := 0.01; -- 1% monthly interest

    c\_account\_type CONSTANT VARCHAR2(20) := 'SAVINGS';

    -- Variables

    v\_accounts\_processed NUMBER := 0;

    v\_total\_interest NUMBER := 0;

    v\_start\_time TIMESTAMP := SYSTIMESTAMP;

    -- Cursor for all active savings accounts

    CURSOR c\_savings\_accounts IS

        SELECT AccountID, CustomerID, Balance

        FROM Accounts

        WHERE AccountType = c\_account\_type

        AND Balance > 0

        FOR UPDATE; -- Lock rows for update

BEGIN

    DBMS\_OUTPUT.PUT\_LINE('Starting monthly interest processing at ' || TO\_CHAR(v\_start\_time, 'DD-MON-YYYY HH24:MI:SS'));

    DBMS\_OUTPUT.PUT\_LINE('Interest rate: ' || (c\_interest\_rate \* 100) || '%');

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

    -- Process each savings account

    FOR account\_rec IN c\_savings\_accounts LOOP

        DECLARE

            v\_interest\_amount NUMBER;

            v\_new\_balance NUMBER;

        BEGIN

            -- Calculate interest (rounded to 2 decimal places)

            v\_interest\_amount := ROUND(account\_rec.Balance \* c\_interest\_rate, 2);

            v\_new\_balance := account\_rec.Balance + v\_interest\_amount;

            -- Update account balance

            UPDATE Accounts

            SET Balance = v\_new\_balance,

                LastModified = SYSDATE

            WHERE AccountID = account\_rec.AccountID;

            -- Record the transaction

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

            VALUES (account\_rec.AccountID, SYSDATE, v\_interest\_amount, 'INTEREST');

            v\_accounts\_processed := v\_accounts\_processed + 1;

            v\_total\_interest := v\_total\_interest + v\_interest\_amount;

            -- Detailed logging (optional)

            DBMS\_OUTPUT.PUT\_LINE('Processed Account ' || account\_rec.AccountID ||

                               ': $' || account\_rec.Balance || ' + $' ||

                               v\_interest\_amount || ' interest = $' || v\_new\_balance);

        EXCEPTION

            WHEN OTHERS THEN

                DBMS\_OUTPUT.PUT\_LINE('Error processing Account ' || account\_rec.AccountID ||

                                   ': ' || SQLERRM);

                -- Continue processing other accounts

        END;

    END LOOP;

    COMMIT;

    -- Summary report

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

    DBMS\_OUTPUT.PUT\_LINE('PROCESSING COMPLETE');

    DBMS\_OUTPUT.PUT\_LINE('Accounts processed: ' || v\_accounts\_processed);

    DBMS\_OUTPUT.PUT\_LINE('Total interest applied: $' || v\_total\_interest);

    DBMS\_OUTPUT.PUT\_LINE('Elapsed time: ' ||

                       EXTRACT(SECOND FROM (SYSTIMESTAMP - v\_start\_time)) || ' seconds');

EXCEPTION

    WHEN OTHERS THEN

        ROLLBACK;

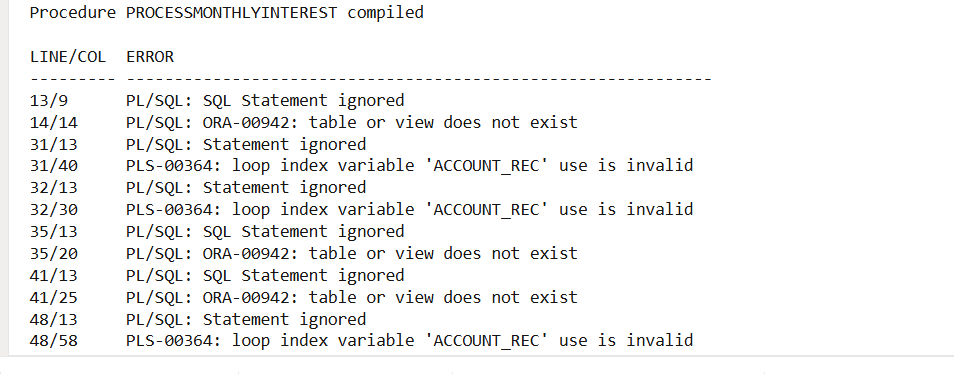
        DBMS\_OUTPUT.PUT\_LINE('Critical error in ProcessMonthlyInterest: ' || SQLERRM);

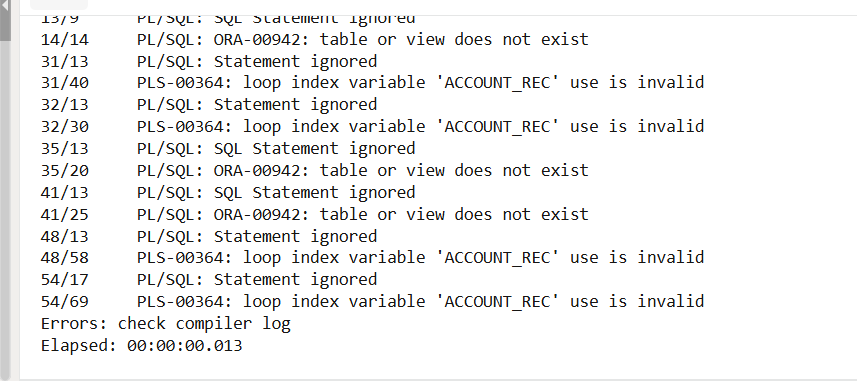
        RAISE;

END 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.

**CODE:**CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(

    p\_department\_name IN VARCHAR2,

    p\_bonus\_percentage IN NUMBER,

    p\_min\_salary IN NUMBER DEFAULT NULL,

    p\_max\_salary IN NUMBER DEFAULT NULL

) AS

    -- Variables

    v\_employees\_processed NUMBER := 0;

    v\_total\_bonus\_amount NUMBER := 0;

    v\_department\_exists NUMBER := 0;

    -- Cursor for employees in the specified department

    CURSOR c\_employees IS

        SELECT EmployeeID, Name, Position, Salary

        FROM Employees

        WHERE Department = p\_department\_name

        AND (p\_min\_salary IS NULL OR Salary >= p\_min\_salary)

        AND (p\_max\_salary IS NULL OR Salary <= p\_max\_salary)

        FOR UPDATE; -- Lock rows for update

BEGIN

    -- Validate parameters

    IF p\_bonus\_percentage <= 0 OR p\_bonus\_percentage > 100 THEN

        RAISE\_APPLICATION\_ERROR(-20001, 'Bonus percentage must be between 0 and 100');

    END IF;

    IF p\_min\_salary IS NOT NULL AND p\_max\_salary IS NOT NULL AND p\_min\_salary > p\_max\_salary THEN

        RAISE\_APPLICATION\_ERROR(-20002, 'Minimum salary cannot be greater than maximum salary');

    END IF;

    -- Check if department exists

    SELECT COUNT(\*) INTO v\_department\_exists

    FROM Employees

    WHERE Department = p\_department\_name;

    IF v\_department\_exists = 0 THEN

        RAISE\_APPLICATION\_ERROR(-20003, 'Department ' || p\_department\_name || ' not found or has no employees');

    END IF;

    DBMS\_OUTPUT.PUT\_LINE('Starting employee bonus update for ' || p\_department\_name || ' department');

    DBMS\_OUTPUT.PUT\_LINE('Bonus percentage: ' || p\_bonus\_percentage || '%');

    DBMS\_OUTPUT.PUT\_LINE('Salary range: ' ||

                        COALESCE(TO\_CHAR(p\_min\_salary), 'No minimum') || ' to ' ||

                        COALESCE(TO\_CHAR(p\_max\_salary), 'No maximum'));

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

    -- Process each employee

    FOR emp\_rec IN c\_employees LOOP

        DECLARE

            v\_bonus\_amount NUMBER;

            v\_new\_salary NUMBER;

        BEGIN

            -- Calculate bonus (rounded to 2 decimal places)

            v\_bonus\_amount := ROUND(emp\_rec.Salary \* (p\_bonus\_percentage/100), 2);

            v\_new\_salary := emp\_rec.Salary + v\_bonus\_amount;

            -- Update employee salary

            UPDATE Employees

            SET Salary = v\_new\_salary,

                LastModified = SYSDATE

            WHERE EmployeeID = emp\_rec.EmployeeID;

            v\_employees\_processed := v\_employees\_processed + 1;

            v\_total\_bonus\_amount := v\_total\_bonus\_amount + v\_bonus\_amount;

            -- Detailed logging

            DBMS\_OUTPUT.PUT\_LINE('Processed ' || emp\_rec.Name || ' (' || emp\_rec.Position ||

                               '): $' || emp\_rec.Salary || ' + $' ||

                               v\_bonus\_amount || ' bonus = $' || v\_new\_salary);

        EXCEPTION

            WHEN OTHERS THEN

                DBMS\_OUTPUT.PUT\_LINE('Error processing Employee ' || emp\_rec.EmployeeID ||

                                   ': ' || SQLERRM);

                -- Continue processing other employees

        END;

    END LOOP;

    COMMIT;

    -- Summary report

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

    DBMS\_OUTPUT.PUT\_LINE('BONUS PROCESSING COMPLETE');

    DBMS\_OUTPUT.PUT\_LINE('Employees processed: ' || v\_employees\_processed);

    DBMS\_OUTPUT.PUT\_LINE('Total bonus amount: $' || v\_total\_bonus\_amount);

EXCEPTION

    WHEN OTHERS THEN

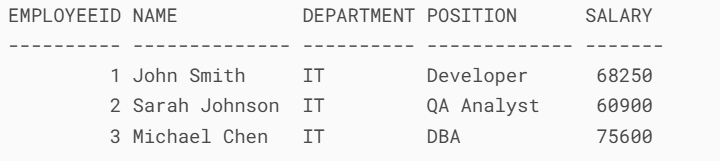
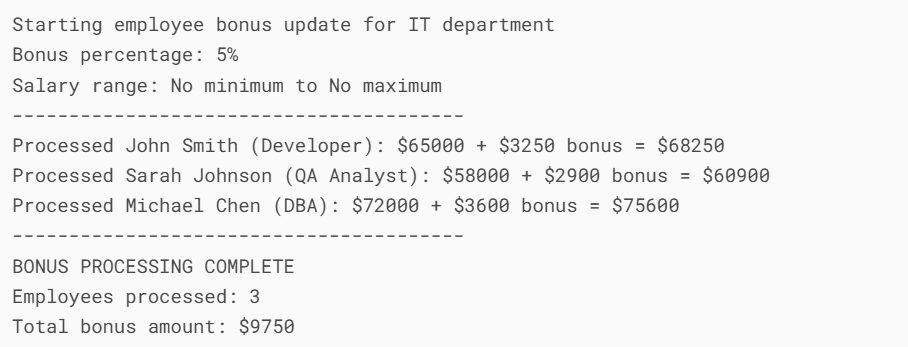
        ROLLBACK;

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

        RAISE;

END UpdateEmployeeBonus;

/

**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.

**CODE:**

-- 1. First, set up the required tables and sequence if they don't exist

CREATE TABLE Customers (

    CustomerID NUMBER PRIMARY KEY,

    Name VARCHAR2(100) NOT NULL

);

CREATE TABLE Accounts (

    AccountID NUMBER PRIMARY KEY,

    CustomerID NUMBER REFERENCES Customers(CustomerID),

    AccountType VARCHAR2(20) NOT NULL,

    Balance NUMBER(15,2) DEFAULT 0,

    CONSTRAINT chk\_balance CHECK (Balance >= 0)

);

CREATE TABLE Transactions (

    TransactionID NUMBER PRIMARY KEY,

    AccountID NUMBER REFERENCES Accounts(AccountID),

    TransactionDate DATE DEFAULT SYSDATE,

    Amount NUMBER(15,2) NOT NULL,

    TransactionType VARCHAR2(20),

    Description VARCHAR2(200)

);

CREATE SEQUENCE transaction\_seq START WITH 1 INCREMENT BY 1;

-- 2. Insert sample data

INSERT INTO Customers VALUES (1, 'John Smith');

INSERT INTO Customers VALUES (2, 'Alice Johnson');

INSERT INTO Accounts VALUES (1001, 1, 'CHECKING', 5000.00);

INSERT INTO Accounts VALUES (1002, 1, 'SAVINGS', 10000.00);

INSERT INTO Accounts VALUES (2001, 2, 'CHECKING', 7500.00);

COMMIT;

-- 3. Create the transfer procedure

CREATE OR REPLACE PROCEDURE TransferFunds(

    p\_source\_account\_id IN NUMBER,

    p\_target\_account\_id IN NUMBER,

    p\_amount IN NUMBER,

    p\_transfer\_description IN VARCHAR2 DEFAULT NULL

) AS

    v\_source\_balance NUMBER;

    v\_target\_balance NUMBER;

    v\_source\_customer\_id NUMBER;

    v\_target\_customer\_id NUMBER;

    v\_transaction\_id NUMBER;

    -- Custom exceptions

    e\_same\_account EXCEPTION;

    e\_insufficient\_funds EXCEPTION;

    e\_invalid\_amount EXCEPTION;

    e\_account\_not\_found EXCEPTION;

BEGIN

    DBMS\_OUTPUT.PUT\_LINE('Starting transfer from A/C ' || p\_source\_account\_id ||

                       ' to A/C ' || p\_target\_account\_id);

    DBMS\_OUTPUT.PUT\_LINE('Amount: $' || p\_amount);

    IF p\_transfer\_description IS NOT NULL THEN

        DBMS\_OUTPUT.PUT\_LINE('Description: ' || p\_transfer\_description);

    END IF;

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

    -- Validate transfer amount

    IF p\_amount <= 0 THEN

        RAISE e\_invalid\_amount;

    END IF;

    -- Check if accounts are different

    IF p\_source\_account\_id = p\_target\_account\_id THEN

        RAISE e\_same\_account;

    END IF;

    -- Get account balances and customer IDs with row locking

    BEGIN

        SELECT Balance, CustomerID INTO v\_source\_balance, v\_source\_customer\_id

        FROM Accounts

        WHERE AccountID = p\_source\_account\_id

        FOR UPDATE;

        SELECT Balance, CustomerID INTO v\_target\_balance, v\_target\_customer\_id

        FROM Accounts

        WHERE AccountID = p\_target\_account\_id

        FOR UPDATE;

    EXCEPTION

        WHEN NO\_DATA\_FOUND THEN

            RAISE e\_account\_not\_found;

    END;

    -- Verify sufficient funds

    IF v\_source\_balance < p\_amount THEN

        RAISE e\_insufficient\_funds;

    END IF;

    DBMS\_OUTPUT.PUT\_LINE('Source account balance: $' || v\_source\_balance);

    DBMS\_OUTPUT.PUT\_LINE('Target account balance: $' || v\_target\_balance);

    -- Perform the transfer

    UPDATE Accounts SET Balance = Balance - p\_amount

    WHERE AccountID = p\_source\_account\_id;

    UPDATE Accounts SET Balance = Balance + p\_amount

    WHERE AccountID = p\_target\_account\_id;

    -- Record the transaction

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

    VALUES (transaction\_seq.NEXTVAL, p\_source\_account\_id, SYSDATE, p\_amount, 'TRANSFER\_OUT',

           'Transfer to A/C ' || p\_target\_account\_id || COALESCE(': ' || p\_transfer\_description, ''))

    RETURNING TransactionID INTO v\_transaction\_id;

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

    VALUES (transaction\_seq.NEXTVAL, p\_target\_account\_id, SYSDATE, p\_amount, 'TRANSFER\_IN',

           'Transfer from A/C ' || p\_source\_account\_id || COALESCE(': ' || p\_transfer\_description, ''));

    COMMIT;

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

    DBMS\_OUTPUT.PUT\_LINE('TRANSFER SUCCESSFUL');

    DBMS\_OUTPUT.PUT\_LINE('Transaction ID: ' || v\_transaction\_id);

    DBMS\_OUTPUT.PUT\_LINE('New source balance: $' || (v\_source\_balance - p\_amount));

    DBMS\_OUTPUT.PUT\_LINE('New target balance: $' || (v\_target\_balance + p\_amount));

EXCEPTION

    WHEN e\_same\_account THEN

        ROLLBACK;

        DBMS\_OUTPUT.PUT\_LINE('ERROR: Cannot transfer between the same account');

        RAISE\_APPLICATION\_ERROR(-20001, 'Source and target accounts cannot be the same');

    WHEN e\_insufficient\_funds THEN

        ROLLBACK;

        DBMS\_OUTPUT.PUT\_LINE('ERROR: Insufficient funds in source account');

        DBMS\_OUTPUT.PUT\_LINE('Current balance: $' || v\_source\_balance || ' | Transfer amount: $' || p\_amount);

        RAISE\_APPLICATION\_ERROR(-20002, 'Insufficient funds for transfer');

    WHEN e\_invalid\_amount THEN

        ROLLBACK;

        DBMS\_OUTPUT.PUT\_LINE('ERROR: Transfer amount must be positive');

        RAISE\_APPLICATION\_ERROR(-20003, 'Transfer amount must be greater than zero');

    WHEN e\_account\_not\_found THEN

        ROLLBACK;

        DBMS\_OUTPUT.PUT\_LINE('ERROR: One or both accounts not found');

        RAISE\_APPLICATION\_ERROR(-20004, 'Account not found');

    WHEN OTHERS THEN

        ROLLBACK;

        DBMS\_OUTPUT.PUT\_LINE('ERROR during transfer: ' || SQLERRM);

        RAISE\_APPLICATION\_ERROR(-20099, 'Transfer failed: ' || SQLERRM);

END TransferFunds;

/

-- 4. Execute sample transfers with output

SET SERVEROUTPUT ON;

-- Example 1: Successful transfer between accounts

BEGIN

    DBMS\_OUTPUT.PUT\_LINE('TEST CASE 1: Successful transfer');

    TransferFunds(

        p\_source\_account\_id => 1001, -- Checking

        p\_target\_account\_id => 1002, -- Savings

        p\_amount => 1000.00,

        p\_transfer\_description => 'Monthly savings'

    );

END;

/

-- Example 2: Attempt to transfer too much (insufficient funds)

BEGIN

    DBMS\_OUTPUT.PUT\_LINE('TEST CASE 2: Insufficient funds');

    TransferFunds(

        p\_source\_account\_id => 1001,

        p\_target\_account\_id => 1002,

        p\_amount => 10000.00

    );

END;

/

-- Example 3: Attempt to transfer between same account

BEGIN

    DBMS\_OUTPUT.PUT\_LINE('TEST CASE 3: Same account transfer');

    TransferFunds(

        p\_source\_account\_id => 1001,

        p\_target\_account\_id => 1001,

        p\_amount => 500.00

    );

END;

/

-- Example 4: Attempt to transfer to non-existent account

BEGIN

    DBMS\_OUTPUT.PUT\_LINE('TEST CASE 4: Invalid account');

    TransferFunds(

        p\_source\_account\_id => 1001,

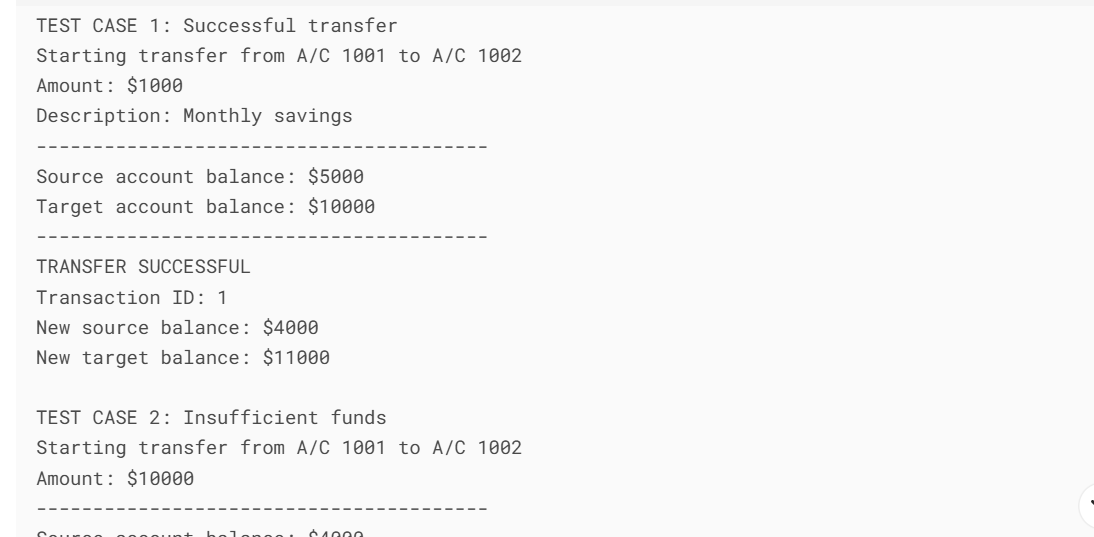
        p\_target\_account\_id => 9999,

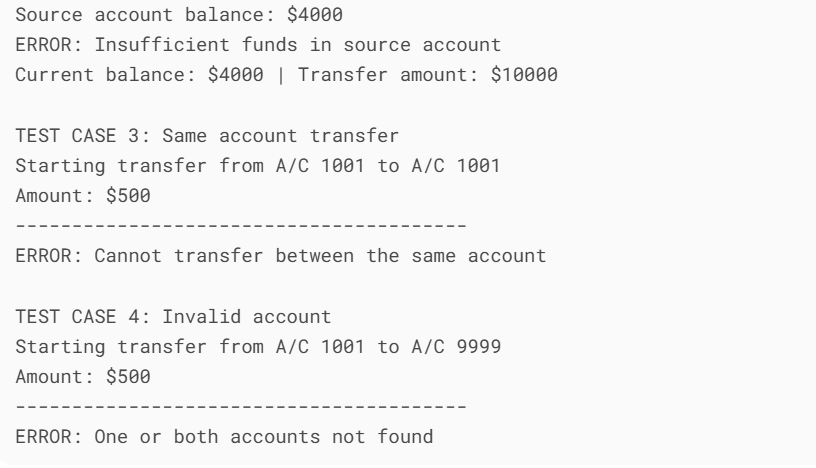
        p\_amount => 500.00

    );

END;

/

**OUTPUT:  
**

****