WEEK 2 PL/SQL

ABHISHEK GUHA

# TABLE CREATION

## SCHEMA DEFINATION

*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)*

*);*

*CREATE TABLE Loans (*

*LoanID NUMBER PRIMARY KEY,*

*CustomerID NUMBER,*

*LoanAmount NUMBER,*

*InterestRate NUMBER,*

*StartDate DATE,*

*EndDate DATE,*

*FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)*

*);*

*CREATE TABLE Employees (*

*EmployeeID NUMBER PRIMARY KEY,*

*Name VARCHAR2(100),*

*Position VARCHAR2(50),*

*Salary NUMBER,*

*Department VARCHAR2(50),*

*HireDate DATE*

*);*

## DATA INSERTION

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

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

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

*VALUES (2, 'Jane Smith', TO\_DATE('1990-07-20', 'YYYY-MM-DD'), 1500, SYSDATE);*

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

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

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

*VALUES (2, 2, 'Checking', 1500, SYSDATE);*

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

*VALUES (1, 1, SYSDATE, 200, 'Deposit');*

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

*VALUES (2, 2, SYSDATE, 300, 'Withdrawal');*

*INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate)*

*VALUES (1, 1, 5000, 5, SYSDATE, ADD\_MONTHS(SYSDATE, 60));*

*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'));*

### INSERT MORE STATEMENTS

*-- More Customers*

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

*VALUES (3, 'Emily White', TO\_DATE('1955-12-10', 'YYYY-MM-DD'), 15000, SYSDATE);*

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

*VALUES (4, 'Michael Brown', TO\_DATE('1978-08-22', 'YYYY-MM-DD'), 500, SYSDATE);*

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

*VALUES (5, 'Sarah Johnson', TO\_DATE('1992-03-15', 'YYYY-MM-DD'), 12000, SYSDATE);*

*-- More Accounts*

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

*VALUES (3, 3, 'Savings', 15000, SYSDATE);*

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

*VALUES (4, 4, 'Checking', 500, SYSDATE);*

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

*VALUES (5, 5, 'Savings', 12000, SYSDATE);*

*-- More Transactions*

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

*VALUES (3, 3, SYSDATE - 5, 5000, 'Deposit');*

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

*VALUES (4, 4, SYSDATE - 2, 100, 'Withdrawal');*

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

*VALUES (5, 5, SYSDATE - 1, 2000, 'Deposit');*

*-- More Loans*

*INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate)*

*VALUES (2, 2, 10000, 6.5, SYSDATE - 365, ADD\_MONTHS(SYSDATE, 24));*

*INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate)*

*VALUES (3, 3, 20000, 4.5, SYSDATE - 730, ADD\_MONTHS(SYSDATE, 1));*

*INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate)*

*VALUES (4, 4, 7500, 7, SYSDATE - 90, ADD\_MONTHS(SYSDATE, 36));*

*-- More Employees*

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

*VALUES (3, 'Carol Davis', 'Teller', 45000, 'Operations', TO\_DATE('2019-09-01', 'YYYY-MM-DD'));*

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

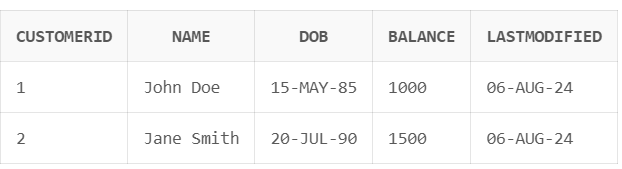
*VALUES (4, 'David Wilson', 'Loan Officer', 55000, 'Loans', TO\_DATE('2016-11-15', 'YYYY-MM-DD'));*

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

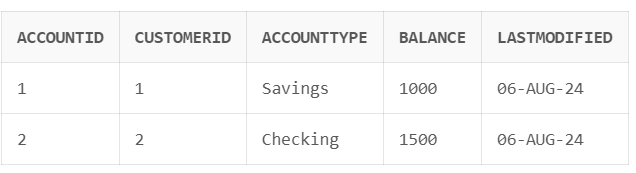
*VALUES (5, 'Eva Martinez', 'IT Specialist', 65000, 'IT', TO\_DATE('2018-07-01', 'YYYY-MM-DD'));*

## TABLE APPEARANCES (AS GIVEN)

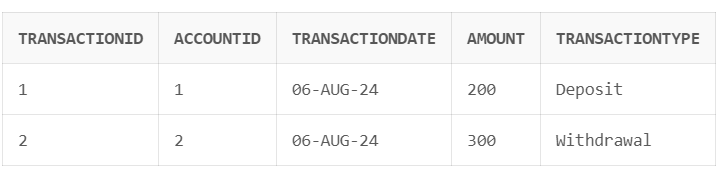
CUSTOMERS TABLE



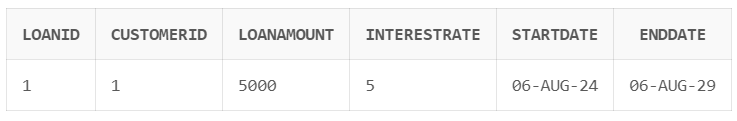
ACCOUNTS TABLE



TRANSACTIONS TABLE



LOANS TABLE



EMPLOYEE TABLE



# 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.

DECLARE

v\_age NUMBER;

v\_discount NUMBER := 0.01; -- 1% discount

BEGIN

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

v\_age := TRUNC(MONTHS\_BETWEEN(SYSDATE, customer.DOB) / 12);

IF v\_age > 60 THEN

UPDATE Loans

SET InterestRate = InterestRate - v\_discount

WHERE CustomerID = customer.CustomerID;

END IF;

END LOOP;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Interest rate discounts applied for customers over 60.');

END;

/



**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.

ALTER TABLE Customers ADD IsVIP VARCHAR2(5);

DECLARE

v\_vip\_threshold NUMBER := 10000;

BEGIN

UPDATE Customers

SET IsVIP = CASE

WHEN Balance > v\_vip\_threshold THEN 'TRUE'

ELSE 'FALSE'

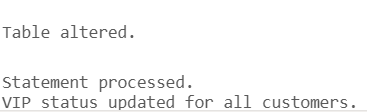
END;

COMMIT;

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

END;

/



**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.

DECLARE

v\_reminder\_days NUMBER := 30;

BEGIN

FOR loan IN (

SELECT c.Name, l.LoanID, l.EndDate

FROM Loans l

JOIN Customers c ON l.CustomerID = c.CustomerID

WHERE l.EndDate BETWEEN SYSDATE AND SYSDATE + v\_reminder\_days

) LOOP

DBMS\_OUTPUT.PUT\_LINE('Reminder: Dear ' || loan.Name ||

', your loan (ID: ' || loan.LoanID ||

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

'. Please arrange for repayment.');

END LOOP;

END;

/



# 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.

CREATE OR REPLACE PROCEDURE SafeTransferFunds(

p\_from\_account NUMBER,

p\_to\_account NUMBER,

p\_amount NUMBER

)

AS

v\_from\_balance NUMBER;

insufficient\_funds EXCEPTION;

PRAGMA EXCEPTION\_INIT(insufficient\_funds, -20001);

BEGIN

-- Start transaction

SAVEPOINT start\_transfer;

-- Check if source account has sufficient funds

SELECT Balance INTO v\_from\_balance

FROM Accounts

WHERE AccountID = p\_from\_account

FOR UPDATE;

IF v\_from\_balance < p\_amount THEN

RAISE insufficient\_funds;

END IF;

-- Update balances

UPDATE Accounts

SET Balance = Balance - p\_amount

WHERE AccountID = p\_from\_account;

UPDATE Accounts

SET Balance = Balance + p\_amount

WHERE AccountID = p\_to\_account;

-- Commit transaction

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Transfer completed successfully.');

EXCEPTION

WHEN insufficient\_funds THEN

ROLLBACK TO start\_transfer;

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

WHEN NO\_DATA\_FOUND THEN

ROLLBACK TO start\_transfer;

DBMS\_OUTPUT.PUT\_LINE('Error: One or both account IDs are invalid.');

WHEN OTHERS THEN

ROLLBACK TO start\_transfer;

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

END SafeTransferFunds;

/

-- Successful transfer

BEGIN

SafeTransferFunds(1, 2, 500);

END;

/

-- Insufficient funds

BEGIN

SafeTransferFunds(1, 2, 100000);

END;

/

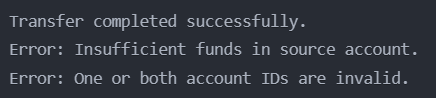
-- Invalid account

BEGIN

SafeTransferFunds(1, 999, 500);

END;

/

****

**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.

CREATE OR REPLACE PROCEDURE UpdateSalary(

p\_employee\_id NUMBER,

p\_increase\_percentage NUMBER

)

AS

v\_current\_salary NUMBER;

employee\_not\_found EXCEPTION;

PRAGMA EXCEPTION\_INIT(employee\_not\_found, -20002);

BEGIN

-- Check if employee exists

SELECT Salary INTO v\_current\_salary

FROM Employees

WHERE EmployeeID = p\_employee\_id

FOR UPDATE;

-- Update salary

UPDATE Employees

SET Salary = Salary \* (1 + p\_increase\_percentage / 100)

WHERE EmployeeID = p\_employee\_id;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Salary updated successfully.');

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RAISE employee\_not\_found;

WHEN employee\_not\_found THEN

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

WHEN OTHERS THEN

ROLLBACK;

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

END UpdateSalary;

/

-- Successful salary update

BEGIN

UpdateSalary(1, 10);

END;

/

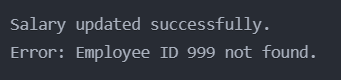
-- Employee not found

BEGIN

UpdateSalary(999, 10);

END;

/

****

**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.

CREATE OR REPLACE PROCEDURE AddNewCustomer(

p\_customer\_id NUMBER,

p\_name VARCHAR2,

p\_dob DATE,

p\_balance NUMBER

)

AS

duplicate\_customer EXCEPTION;

PRAGMA EXCEPTION\_INIT(duplicate\_customer, -20003);

BEGIN

-- Check if customer already exists

IF EXISTS (SELECT 1 FROM Customers WHERE CustomerID = p\_customer\_id) THEN

RAISE duplicate\_customer;

END IF;

-- Insert new customer

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

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

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('New customer added successfully.');

EXCEPTION

WHEN duplicate\_customer THEN

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

WHEN OTHERS THEN

ROLLBACK;

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

END AddNewCustomer;

/

-- Successful customer addition

BEGIN

AddNewCustomer(6, 'Alex Green', TO\_DATE('1988-09-25', 'YYYY-MM-DD'), 3000);

END;

/

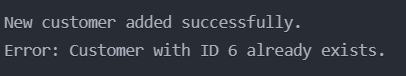
-- Attempt to add duplicate customer

BEGIN

AddNewCustomer(6, 'Alex Green', TO\_DATE('1988-09-25', 'YYYY-MM-DD'), 3000);

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.

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest AS

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

BEGIN

UPDATE Accounts a

SET Balance = Balance + (Balance \* v\_interest\_rate)

WHERE AccountType = 'Savings';

DBMS\_OUTPUT.PUT\_LINE('Monthly interest processed for all savings accounts.');

EXCEPTION

WHEN OTHERS THEN

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

ROLLBACK;

END ProcessMonthlyInterest;

/

-- Example usage:

BEGIN

ProcessMonthlyInterest;

END;

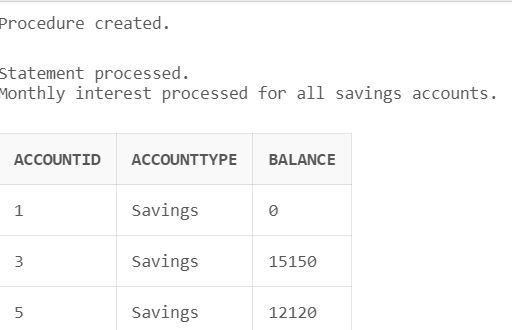
/

-- Verify results:

SELECT AccountID, AccountType, Balance

FROM Accounts

WHERE AccountType = 'Savings';



**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.

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(

p\_department VARCHAR2,

p\_bonus\_percentage NUMBER

) AS

v\_affected\_rows NUMBER;

BEGIN

UPDATE Employees

SET Salary = Salary \* (1 + p\_bonus\_percentage / 100)

WHERE Department = p\_department;

v\_affected\_rows := SQL%ROWCOUNT;

IF v\_affected\_rows > 0 THEN

DBMS\_OUTPUT.PUT\_LINE('Bonus applied to ' || v\_affected\_rows || ' employees in ' || p\_department || ' department.');

ELSE

DBMS\_OUTPUT.PUT\_LINE('No employees found in ' || p\_department || ' department.');

END IF;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

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

ROLLBACK;

END UpdateEmployeeBonus;

/

-- Example usage:

BEGIN

UpdateEmployeeBonus('IT', 5);

END;

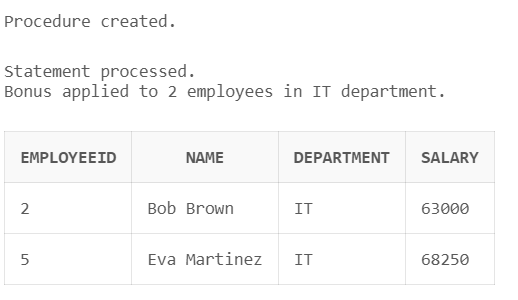
/

-- Verify results:

SELECT EmployeeID, Name, Department, Salary

FROM Employees

WHERE Department = 'IT';



**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.

CREATE OR REPLACE PROCEDURE TransferFunds(

p\_from\_account NUMBER,

p\_to\_account NUMBER,

p\_amount NUMBER

) AS

v\_from\_balance NUMBER;

insufficient\_funds EXCEPTION;

PRAGMA EXCEPTION\_INIT(insufficient\_funds, -20001);

BEGIN

-- Check if source account has sufficient funds

SELECT Balance INTO v\_from\_balance

FROM Accounts

WHERE AccountID = p\_from\_account

FOR UPDATE;

IF v\_from\_balance < p\_amount THEN

RAISE insufficient\_funds;

END IF;

-- Perform the transfer

UPDATE Accounts

SET Balance = Balance - p\_amount

WHERE AccountID = p\_from\_account;

UPDATE Accounts

SET Balance = Balance + p\_amount

WHERE AccountID = p\_to\_account;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Transfer of ' || p\_amount || ' completed successfully.');

EXCEPTION

WHEN insufficient\_funds THEN

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

ROLLBACK;

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('Error: One or both account IDs are invalid.');

ROLLBACK;

WHEN OTHERS THEN

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

ROLLBACK;

END TransferFunds;

/

-- Example usage:

BEGIN

TransferFunds(1, 2, 500);

END;

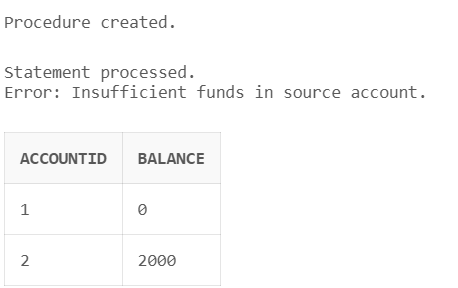
/

-- Verify results:

SELECT AccountID, Balance

FROM Accounts

WHERE AccountID IN (1, 2);



# 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.

CREATE OR REPLACE FUNCTION CalculateAge(p\_dob DATE)

RETURN NUMBER IS

BEGIN

RETURN TRUNC(MONTHS\_BETWEEN(SYSDATE, p\_dob) / 12);

END CalculateAge;

/

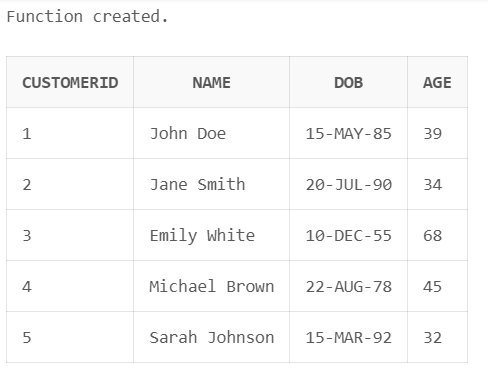
-- Example usage:

SELECT CustomerID, Name, DOB, CalculateAge(DOB) AS Age

FROM Customers;

-- Test with a specific date:

SELECT CalculateAge(TO\_DATE('1990-01-01', 'YYYY-MM-DD')) AS Age FROM DUAL;



**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.

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(

p\_loan\_amount NUMBER,

p\_interest\_rate NUMBER,

p\_duration\_years NUMBER

) RETURN NUMBER IS

v\_monthly\_rate NUMBER;

v\_num\_payments NUMBER;

BEGIN

v\_monthly\_rate := p\_interest\_rate / (12 \* 100);

v\_num\_payments := p\_duration\_years \* 12;

RETURN (p\_loan\_amount \* v\_monthly\_rate \* POWER(1 + v\_monthly\_rate, v\_num\_payments)) /

(POWER(1 + v\_monthly\_rate, v\_num\_payments) - 1);

END CalculateMonthlyInstallment;

/

-- Example usage:

SELECT LoanID, LoanAmount, InterestRate,

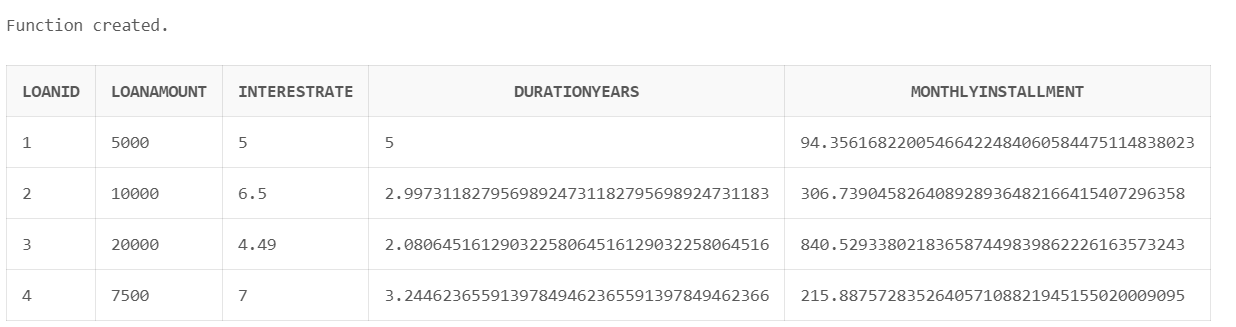
MONTHS\_BETWEEN(EndDate, StartDate) / 12 AS DurationYears,

CalculateMonthlyInstallment(LoanAmount, InterestRate, MONTHS\_BETWEEN(EndDate, StartDate) / 12) AS MonthlyInstallment

FROM Loans;

-- Test with specific values:

SELECT CalculateMonthlyInstallment(100000, 5, 10) AS MonthlyInstallment FROM DUAL;



**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.

CREATE OR REPLACE FUNCTION HasSufficientBalance(

p\_account\_id NUMBER,

p\_amount NUMBER

) RETURN BOOLEAN IS

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance

FROM Accounts

WHERE AccountID = p\_account\_id;

RETURN v\_balance >= p\_amount;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN FALSE;

END HasSufficientBalance;

/

-- Example usage:

SELECT AccountID, Balance,

CASE WHEN HasSufficientBalance(AccountID, 1000) THEN 'Yes' ELSE 'No' END AS HasSufficientBalance

FROM Accounts;

-- Test with specific values:

DECLARE

v\_result BOOLEAN;

BEGIN

v\_result := HasSufficientBalance(1, 500);

IF v\_result THEN

DBMS\_OUTPUT.PUT\_LINE('Account has sufficient balance.');

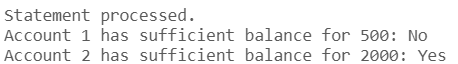
ELSE

DBMS\_OUTPUT.PUT\_LINE('Account does not have sufficient balance.');

END IF;

END;

/



# Exercise 5: Triggers

## CREATE AUDIT TABLE

-- Create AuditLog table if it doesn't exist

CREATE TABLE AuditLog (

LogID NUMBER PRIMARY KEY,

TableName VARCHAR2(50),

OperationType VARCHAR2(10),

RecordID NUMBER,

LogDate DATE

);

-- Create a sequence for AuditLog

CREATE SEQUENCE AuditLog\_Seq START WITH 1 INCREMENT BY 1;

**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.

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON Customers

FOR EACH ROW

BEGIN

:NEW.LastModified := SYSDATE;

END;

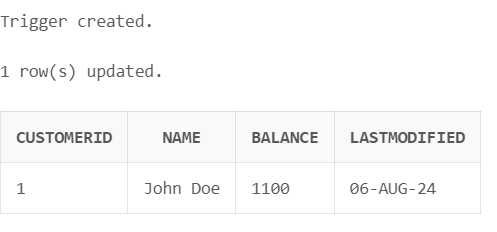
/

-- Test the trigger

UPDATE Customers SET Balance = Balance + 100 WHERE CustomerID = 1;

-- Verify the result

SELECT CustomerID, Name, Balance, LastModified FROM Customers WHERE CustomerID = 1;



**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.

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog (LogID, TableName, OperationType, RecordID, LogDate)

VALUES (AuditLog\_Seq.NEXTVAL, 'Transactions', 'INSERT', :NEW.TransactionID, SYSDATE);

END;

/

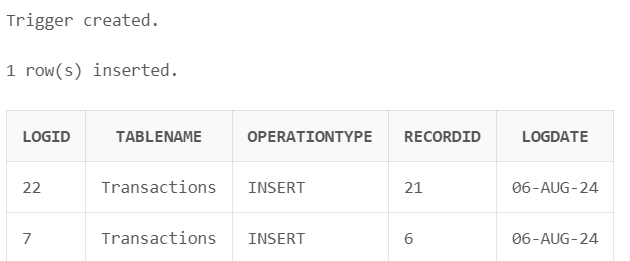
-- Test the trigger

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

VALUES (AuditLog\_Seq.NEXTVAL, 1, SYSDATE, 1000, 'Deposit');

-- Verify the result

SELECT \* FROM AuditLog;



**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.

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, 'Insufficient funds for withdrawal');

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

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

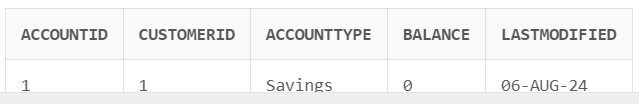
END IF;

END;

/

-- Verify the results

SELECT \* FROM Accounts WHERE AccountID = 1;



# 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.

CREATE OR REPLACE PROCEDURE GenerateMonthlyStatements IS

CURSOR cust\_cur IS

SELECT DISTINCT c.CustomerID, c.Name

FROM Customers c

JOIN Accounts a ON c.CustomerID = a.CustomerID

JOIN Transactions t ON a.AccountID = t.AccountID

WHERE EXTRACT(MONTH FROM t.TransactionDate) = EXTRACT(MONTH FROM SYSDATE)

AND EXTRACT(YEAR FROM t.TransactionDate) = EXTRACT(YEAR FROM SYSDATE);

CURSOR trans\_cur(p\_customer\_id NUMBER) IS

SELECT a.AccountID, t.TransactionDate, t.Amount, t.TransactionType

FROM Accounts a

JOIN Transactions t ON a.AccountID = t.AccountID

WHERE a.CustomerID = p\_customer\_id

AND EXTRACT(MONTH FROM t.TransactionDate) = EXTRACT(MONTH FROM SYSDATE)

AND EXTRACT(YEAR FROM t.TransactionDate) = EXTRACT(YEAR FROM SYSDATE);

v\_total\_balance NUMBER;

BEGIN

FOR cust\_rec IN cust\_cur LOOP

DBMS\_OUTPUT.PUT\_LINE('Monthly Statement for ' || cust\_rec.Name);

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

v\_total\_balance := 0;

FOR trans\_rec IN trans\_cur(cust\_rec.CustomerID) LOOP

DBMS\_OUTPUT.PUT\_LINE('Account: ' || trans\_rec.AccountID ||

', Date: ' || TO\_CHAR(trans\_rec.TransactionDate, 'DD-MON-YYYY') ||

', Amount: ' || trans\_rec.Amount ||

', Type: ' || trans\_rec.TransactionType);

IF trans\_rec.TransactionType = 'Deposit' THEN

v\_total\_balance := v\_total\_balance + trans\_rec.Amount;

ELSIF trans\_rec.TransactionType = 'Withdrawal' THEN

v\_total\_balance := v\_total\_balance - trans\_rec.Amount;

END IF;

END LOOP;

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

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

END LOOP;

END GenerateMonthlyStatements;

/

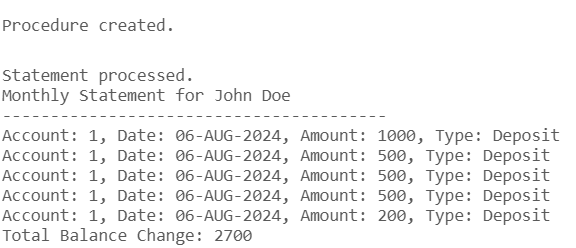
-- Execute the procedure

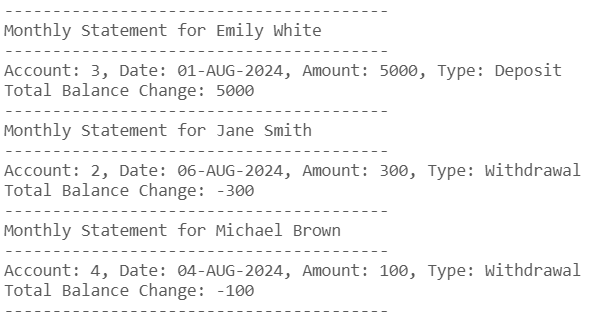
BEGIN

GenerateMonthlyStatements;

END;

/





**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.

CREATE OR REPLACE PROCEDURE ApplyAnnualFee IS

CURSOR acc\_cur IS

SELECT AccountID, Balance

FROM Accounts

FOR UPDATE OF Balance;

v\_annual\_fee NUMBER := 50; -- Set your annual fee amount here

BEGIN

FOR acc\_rec IN acc\_cur LOOP

IF acc\_rec.Balance >= v\_annual\_fee THEN

UPDATE Accounts

SET Balance = Balance - v\_annual\_fee

WHERE CURRENT OF acc\_cur;

DBMS\_OUTPUT.PUT\_LINE('Annual fee applied to Account ' || acc\_rec.AccountID);

ELSE

DBMS\_OUTPUT.PUT\_LINE('Insufficient balance in Account ' || acc\_rec.AccountID || ' to apply annual fee');

END IF;

END LOOP;

COMMIT;

END ApplyAnnualFee;

/

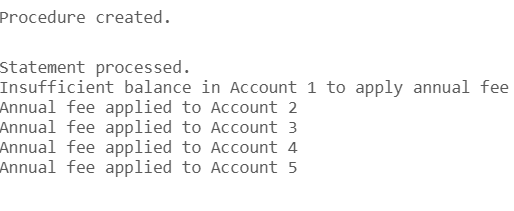
-- Execute the procedure

BEGIN

ApplyAnnualFee;

END;

/



**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.

CREATE OR REPLACE PROCEDURE UpdateLoanInterestRates IS

CURSOR loan\_cur IS

SELECT LoanID, LoanAmount, InterestRate

FROM Loans

FOR UPDATE OF InterestRate;

v\_new\_rate NUMBER;

BEGIN

FOR loan\_rec IN loan\_cur LOOP

-- Example policy: Increase rate by 0.5% for loans over 10000, decrease by 0.25% for others

IF loan\_rec.LoanAmount > 10000 THEN

v\_new\_rate := loan\_rec.InterestRate + 0.5;

ELSE

v\_new\_rate := loan\_rec.InterestRate - 0.25;

END IF;

UPDATE Loans

SET InterestRate = v\_new\_rate

WHERE CURRENT OF loan\_cur;

DBMS\_OUTPUT.PUT\_LINE('Updated interest rate for Loan ' || loan\_rec.LoanID || ' from ' ||

loan\_rec.InterestRate || '% to ' || v\_new\_rate || '%');

END LOOP;

COMMIT;

END UpdateLoanInterestRates;

/

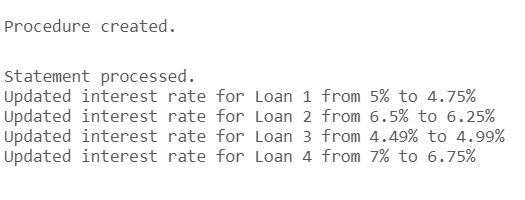
-- Execute the procedure

BEGIN

UpdateLoanInterestRates;

END;

/



# 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.

CREATE SEQUENCE Customers\_Seq

START WITH 1

INCREMENT BY 1

NOCACHE

NOCYCLE;

/

CREATE OR REPLACE PACKAGE CustomerManagement AS

PROCEDURE AddNewCustomer(

p\_name VARCHAR2,

p\_dob DATE,

p\_initial\_balance NUMBER

);

PROCEDURE UpdateCustomerDetails(

p\_customer\_id NUMBER,

p\_name VARCHAR2,

p\_dob DATE

);

FUNCTION GetCustomerBalance(p\_customer\_id NUMBER) RETURN NUMBER;

END CustomerManagement;

/

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

PROCEDURE AddNewCustomer(

p\_name VARCHAR2,

p\_dob DATE,

p\_initial\_balance NUMBER

) IS

v\_customer\_id NUMBER;

BEGIN

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

VALUES (Customers\_Seq.NEXTVAL, p\_name, p\_dob, p\_initial\_balance, SYSDATE)

RETURNING CustomerID INTO v\_customer\_id;

DBMS\_OUTPUT.PUT\_LINE('New customer added with ID: ' || v\_customer\_id);

END AddNewCustomer;

PROCEDURE UpdateCustomerDetails(

p\_customer\_id NUMBER,

p\_name VARCHAR2,

p\_dob DATE

) IS

BEGIN

UPDATE Customers

SET Name = p\_name,

DOB = p\_dob,

LastModified = SYSDATE

WHERE CustomerID = p\_customer\_id;

IF SQL%ROWCOUNT = 0 THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Customer not found');

ELSE

DBMS\_OUTPUT.PUT\_LINE('Customer details updated successfully');

END IF;

END UpdateCustomerDetails;

FUNCTION GetCustomerBalance(p\_customer\_id NUMBER) 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

RAISE\_APPLICATION\_ERROR(-20002, 'Customer not found');

END GetCustomerBalance;

END CustomerManagement;

/

-- Example usage:

BEGIN

CustomerManagement.AddNewCustomer('John Doe', TO\_DATE('1990-01-01', 'YYYY-MM-DD'), 1000);

CustomerManagement.UpdateCustomerDetails(1, 'John Smith', TO\_DATE('1990-01-01', 'YYYY-MM-DD'));

DBMS\_OUTPUT.PUT\_LINE('Customer balance: ' || CustomerManagement.GetCustomerBalance(1));

END;

/

-- Example usage:

BEGIN

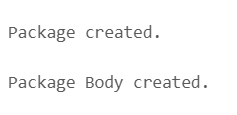
CustomerManagement.AddNewCustomer('John Doe', TO\_DATE('1990-01-01', 'YYYY-MM-DD'), 1000);

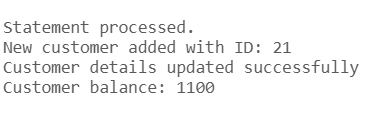
CustomerManagement.UpdateCustomerDetails(1, 'John Smith', TO\_DATE('1990-01-01', 'YYYY-MM-DD'));

DBMS\_OUTPUT.PUT\_LINE('Customer balance: ' || CustomerManagement.GetCustomerBalance(1));

END;

/





**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.

CREATE SEQUENCE Employees\_Seq

START WITH 1

INCREMENT BY 1

NOCACHE

NOCYCLE;

CREATE OR REPLACE PACKAGE EmployeeManagement AS

PROCEDURE HireNewEmployee(

p\_name VARCHAR2,

p\_position VARCHAR2,

p\_salary NUMBER,

p\_department VARCHAR2

);

PROCEDURE UpdateEmployeeDetails(

p\_employee\_id NUMBER,

p\_position VARCHAR2,

p\_salary NUMBER,

p\_department VARCHAR2

);

FUNCTION CalculateAnnualSalary(p\_employee\_id NUMBER) RETURN NUMBER;

END EmployeeManagement;

/

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

PROCEDURE HireNewEmployee(

p\_name VARCHAR2,

p\_position VARCHAR2,

p\_salary NUMBER,

p\_department VARCHAR2

) IS

v\_employee\_id NUMBER;

BEGIN

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

VALUES (Employees\_Seq.NEXTVAL, p\_name, p\_position, p\_salary, p\_department, SYSDATE)

RETURNING EmployeeID INTO v\_employee\_id;

DBMS\_OUTPUT.PUT\_LINE('New employee hired with ID: ' || v\_employee\_id);

END HireNewEmployee;

PROCEDURE UpdateEmployeeDetails(

p\_employee\_id NUMBER,

p\_position VARCHAR2,

p\_salary NUMBER,

p\_department VARCHAR2

) IS

BEGIN

UPDATE Employees

SET Position = p\_position,

Salary = p\_salary,

Department = p\_department

WHERE EmployeeID = p\_employee\_id;

IF SQL%ROWCOUNT = 0 THEN

RAISE\_APPLICATION\_ERROR(-20003, 'Employee not found');

ELSE

DBMS\_OUTPUT.PUT\_LINE('Employee details updated successfully');

END IF;

END UpdateEmployeeDetails;

FUNCTION CalculateAnnualSalary(p\_employee\_id NUMBER) RETURN NUMBER IS

v\_monthly\_salary NUMBER;

BEGIN

SELECT Salary INTO v\_monthly\_salary

FROM Employees

WHERE EmployeeID = p\_employee\_id;

RETURN v\_monthly\_salary \* 12;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

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

END CalculateAnnualSalary;

END EmployeeManagement;

/

-- Example usage:

BEGIN

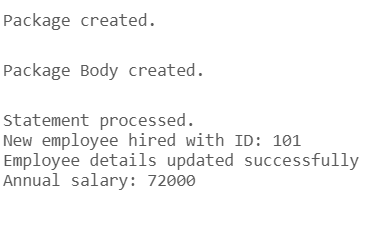
EmployeeManagement.HireNewEmployee('Jane Doe', 'Manager', 5000, 'Sales');

EmployeeManagement.UpdateAmployeeDetails(1, 'Senior Manager', 6000, 'Sales');

DBMS\_OUTPUT.PUT\_LINE('Annual salary: ' || EmployeeManagement.CalculateAnnualSalary(1));

END;

/



**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.

CREATE SEQUENCE Accounts\_Seq

START WITH 1

INCREMENT BY 1

NOCACHE

NOCYCLE;

CREATE OR REPLACE PACKAGE AccountOperations AS

PROCEDURE OpenNewAccount(

p\_customer\_id NUMBER,

p\_account\_type VARCHAR2,

p\_initial\_balance NUMBER

);

PROCEDURE CloseAccount(p\_account\_id NUMBER);

FUNCTION GetTotalCustomerBalance(p\_customer\_id NUMBER) RETURN NUMBER;

END AccountOperations;

/

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

PROCEDURE OpenNewAccount(

p\_customer\_id NUMBER,

p\_account\_type VARCHAR2,

p\_initial\_balance NUMBER

) IS

v\_account\_id NUMBER;

BEGIN

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

VALUES (Accounts\_Seq.NEXTVAL, p\_customer\_id, p\_account\_type, p\_initial\_balance, SYSDATE)

RETURNING AccountID INTO v\_account\_id;

DBMS\_OUTPUT.PUT\_LINE('New account opened with ID: ' || v\_account\_id);

END OpenNewAccount;

PROCEDURE CloseAccount(p\_account\_id NUMBER) IS

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance

FROM Accounts

WHERE AccountID = p\_account\_id;

IF v\_balance <> 0 THEN

RAISE\_APPLICATION\_ERROR(-20005, 'Account balance must be zero to close');

ELSE

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

IF SQL%ROWCOUNT = 0 THEN

RAISE\_APPLICATION\_ERROR(-20006, 'Account not found');

ELSE

DBMS\_OUTPUT.PUT\_LINE('Account closed successfully');

END IF;

END IF;

END CloseAccount;

FUNCTION GetTotalCustomerBalance(p\_customer\_id NUMBER) RETURN NUMBER IS

v\_total\_balance NUMBER;

BEGIN

SELECT SUM(Balance) INTO v\_total\_balance

FROM Accounts

WHERE CustomerID = p\_customer\_id;

RETURN NVL(v\_total\_balance, 0);

END GetTotalCustomerBalance;

END AccountOperations;

/

BEGIN

-- Open new accounts

AccountOperations.OpenNewAccount(1, 'Savings', 1000);

AccountOperations.OpenNewAccount(1, 'Checking', 500);

-- Get total balance for customer ID 1

DBMS\_OUTPUT.PUT\_LINE('Total balance: ' || AccountOperations.GetTotalCustomerBalance(1));

END;

/

