**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 –**

UPDATE Loans

SET InterestRate = InterestRate - 0.01

WHERE CustomerID IN (

SELECT CustomerID

FROM Customers

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

);

--- SELECT EXTRACT(YEAR FROM SYSDATE) - EXTRACT(YEAR FROM DOB) AS age

--- FROM Customers;

UPDATE Loans

SET InterestRate = InterestRate - 0.01

WHERE CustomerID IN (

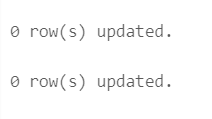
SELECT CustomerID

FROM Customers

WHERE EXTRACT(YEAR FROM SYSDATE) - EXTRACT(YEAR FROM DOB) > 60

);

**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-**

ALTER TABLE Customers

ADD IsVIP NUMBER(1) DEFAULT 0;

BEGIN

FOR customer\_record IN (

SELECT CustomerID, Balance

FROM Customers

) LOOP

IF customer\_record.Balance > 10000 THEN

UPDATE Customers

SET IsVIP = 1

WHERE CustomerID = customer\_record.CustomerID;

END IF;

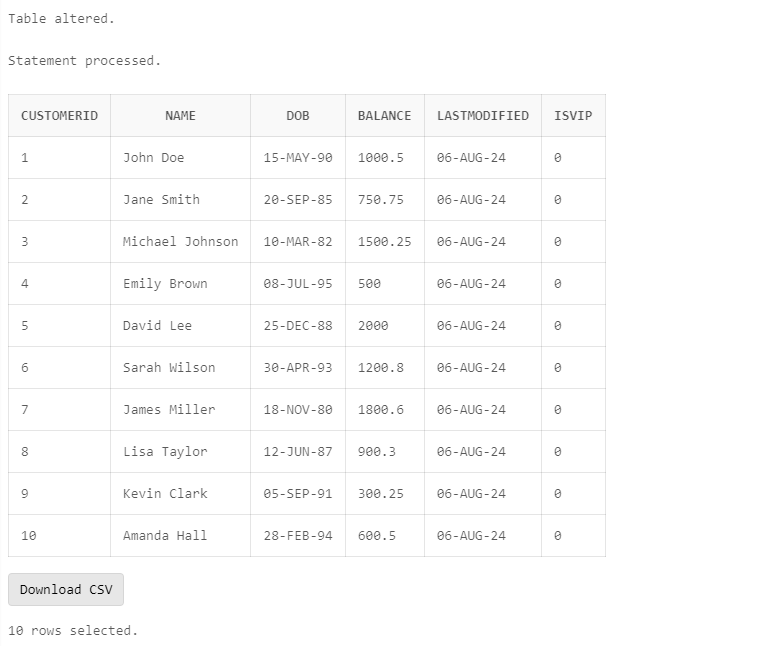
END LOOP;

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.

**Code –**

SET SERVEROUTPUT ON;

BEGIN

DBMS\_OUTPUT.PUT\_LINE('Starting to fetch loan reminders...');

FOR loan\_record IN (

SELECT l.LoanID, l.CustomerID, 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 ' || loan\_record.LoanID ||

' for customer ' || loan\_record.Name ||

' (Customer ID: ' || loan\_record.CustomerID || ') is due on ' ||

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

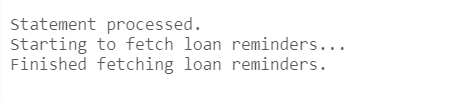
END LOOP;

DBMS\_OUTPUT.PUT\_LINE('Finished fetching loan reminders.');

END;

/

**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.

**Code-**

SET SERVEROUTPUT ON;

CREATE OR REPLACE PROCEDURE SafeTransferFunds (

p\_from\_account\_id NUMBER,

p\_to\_account\_id NUMBER,

p\_amount NUMBER

) IS

insufficient\_funds EXCEPTION;

v\_from\_account\_balance NUMBER;

v\_to\_account\_balance NUMBER;

v\_error\_message VARCHAR2(4000);

BEGIN

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

SELECT balance INTO v\_from\_account\_balance

FROM Accounts

WHERE AccountID = p\_from\_account\_id

FOR UPDATE;

DBMS\_OUTPUT.PUT\_LINE('From Account Balance: ' || v\_from\_account\_balance);

IF v\_from\_account\_balance < p\_amount THEN

RAISE insufficient\_funds;

END IF;

SELECT balance INTO v\_to\_account\_balance

FROM Accounts

WHERE AccountID = p\_to\_account\_id

FOR UPDATE;

DBMS\_OUTPUT.PUT\_LINE('To Account Balance: ' || v\_to\_account\_balance);

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;

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

EXCEPTION

WHEN insufficient\_funds THEN

DBMS\_OUTPUT.PUT\_LINE('Insufficient Funds Exception');

v\_error\_message := 'Insufficient funds in account ' || p\_from\_account\_id;

---DBMS\_OUTPUT.PUT\_LINE('Logging error: ' || v\_error\_message);

---INSERT INTO error\_log (error\_id, error\_message, log\_date)

---VALUES (error\_log\_seq.NEXTVAL, v\_error\_message, SYSDATE);

--DBMS\_OUTPUT.PUT\_LINE('Error logged successfully');

ROLLBACK;

WHEN OTHERS THEN

v\_error\_message := SQLERRM;

DBMS\_OUTPUT.PUT\_LINE('Other Exception: ' || v\_error\_message);

---DBMS\_OUTPUT.PUT\_LINE('Logging error: ' || v\_error\_message);

---INSERT INTO error\_log (error\_id, error\_message, log\_date)

---VALUES (error\_log\_seq.NEXTVAL, v\_error\_message, SYSDATE);

--DBMS\_OUTPUT.PUT\_LINE('Error logged successfully');

ROLLBACK;

END SafeTransferFunds;

/

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

VALUES (20, 'John Doe', TO\_DATE('1990-01-01', 'YYYY-MM-DD'), 20000, SYSDATE);

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

VALUES (21, 'Jane Smith', TO\_DATE('1995-05-15', 'YYYY-MM-DD'), 30000, SYSDATE);

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

VALUES (20, 20, 'Checking', 10000, SYSDATE);

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

VALUES (21, 21, 'Savings', 5000, SYSDATE);

COMMIT;

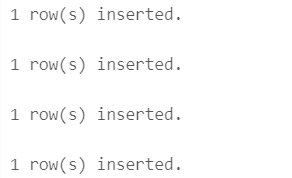
SELECT \* FROM ACCOUNTS;

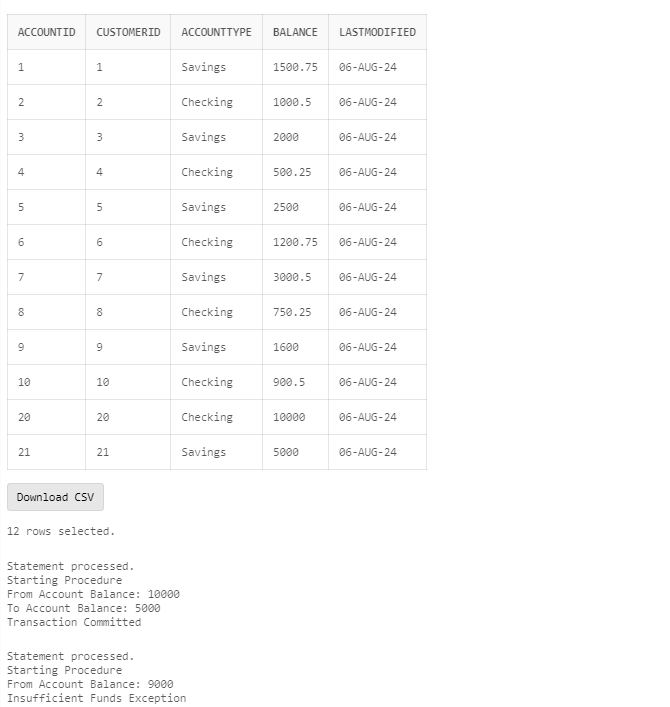
EXEC SafeTransferFunds(20, 21, 1000);

EXEC SafeTransferFunds(20, 21, 20000);

SELECT \* FROM error\_log;

**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.

**Code –**

CREATE OR REPLACE PROCEDURE UpdateSalary (

p\_EmployeeID IN Employees.EmployeeID%TYPE,

p\_Percentage IN NUMBER

) IS

v\_OldSalary Employees.Salary%TYPE;

BEGIN

SELECT Salary

INTO v\_OldSalary

FROM Employees

WHERE EmployeeID = p\_EmployeeID;

UPDATE Employees

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

WHERE EmployeeID = p\_EmployeeID;

COMMIT;

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

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('Error: Employee with ID ' || p\_EmployeeID || ' does not exist.');

WHEN OTHERS THEN

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

END UpdateSalary;

/

BEGIN

UpdateSalary(p\_EmployeeID => 1, p\_Percentage => 10);

END;

/

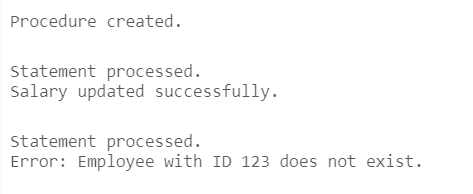
BEGIN

UpdateSalary(123,10);

END;

/

**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.

**Code –**

CREATE OR REPLACE PROCEDURE AddNewCustomer (

p\_CustomerID IN Customers.CustomerID%TYPE,

p\_Name IN Customers.Name%TYPE,

p\_DOB IN Customers.DOB%TYPE,

p\_Balance IN Customers.Balance%TYPE,

p\_LastModified IN Customers.LastModified%TYPE

) IS

BEGIN

BEGIN

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

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

COMMIT;

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

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

WHEN OTHERS THEN

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

END;

END AddNewCustomer;

/

BEGIN

AddNewCustomer(

p\_CustomerID => 1001,

p\_Name => 'John Doe',

p\_DOB => TO\_DATE('1980-01-15', 'YYYY-MM-DD'),

p\_Balance => 5000,

p\_LastModified => SYSDATE

);

END;

/

BEGIN

AddNewCustomer(

p\_CustomerID => 1002,

p\_Name => 'Johnnnnn Doe',

p\_DOB => TO\_DATE('1980-01-15', 'YYYY-MM-DD'),

p\_Balance => 500053,

p\_LastModified => SYSDATE

);

END;

/

BEGIN

AddNewCustomer(

p\_CustomerID => 1001,

p\_Name => 'Johnyyy Doe',

p\_DOB => TO\_DATE('1980-01-15', 'YYYY-MM-DD'),

p\_Balance => 503300,

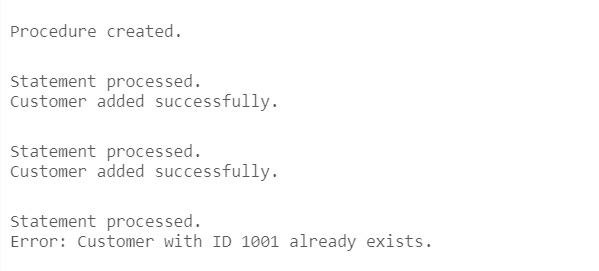
p\_LastModified => SYSDATE

);

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 (

p\_InterestRate IN NUMBER

) IS

BEGIN

UPDATE Account

SET Balance = Balance \* (1 + p\_InterestRate / 100)

WHERE AccountType = 'Savings';

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Monthly interest applied to all savings Account with an interest rate of ' || p\_InterestRate || '%.');

EXCEPTION

WHEN OTHERS THEN

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

ROLLBACK;

END ProcessMonthlyInterest;

/

BEGIN

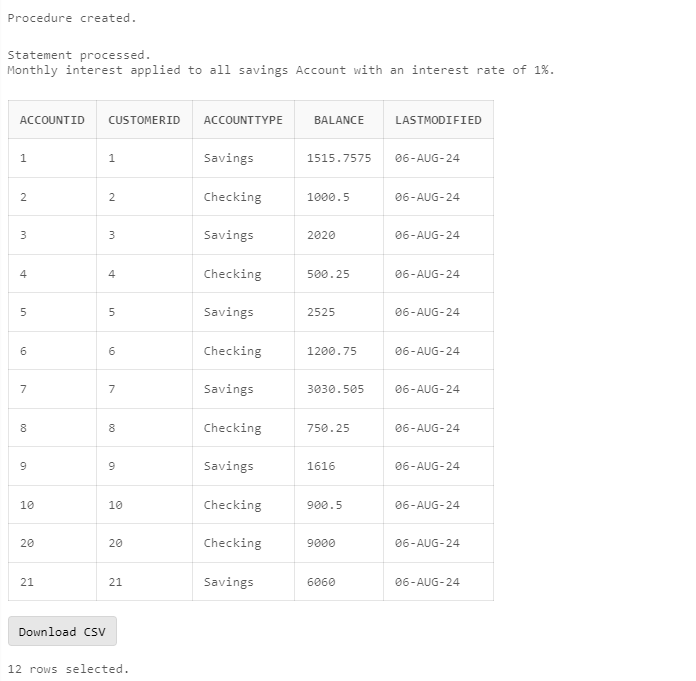
ProcessMonthlyInterest(p\_InterestRate => 1);

END;

/

SELECT \* FROM Account;

**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 IN Employees.Department%TYPE,

p\_BonusPercentage IN NUMBER

) IS

BEGIN

UPDATE Employees

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

WHERE Department = p\_Department;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Salaries updated with a bonus of ' || p\_BonusPercentage || '% for department ' || p\_Department || '.');

EXCEPTION

WHEN OTHERS THEN

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

ROLLBACK;

END UpdateEmployeeBonus;

/

SELECT \* FROM EMPLOYEES;

BEGIN

UpdateEmployeeBonus(p\_Department => 'Sales', p\_BonusPercentage => 10);

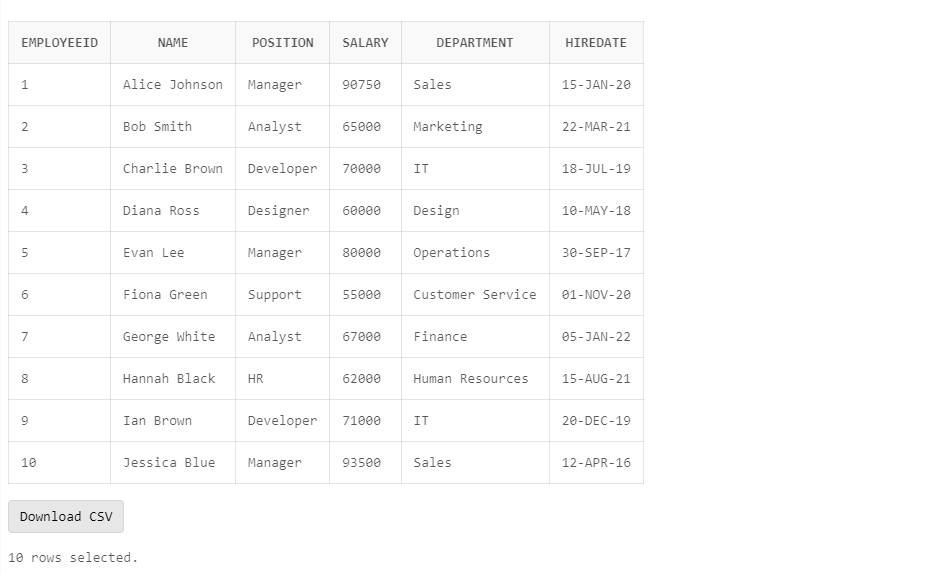
END;

/

SELECT \* 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.

**Code –**

CREATE OR REPLACE PROCEDURE TransferFunds (

p\_SourceAccountID IN Account.AccountID%TYPE,

p\_DestinationAccountID IN Account.AccountID%TYPE,

p\_Amount IN NUMBER

) IS

v\_SourceBalance Account.Balance%TYPE;

BEGIN

SELECT Balance INTO v\_SourceBalance

FROM Account

WHERE AccountID = p\_SourceAccountID;

IF v\_SourceBalance < p\_Amount THEN

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

END IF;

BEGIN

UPDATE Account

SET Balance = Balance - p\_Amount

WHERE AccountID = p\_SourceAccountID;

UPDATE Account

SET Balance = Balance + p\_Amount

WHERE AccountID = p\_DestinationAccountID;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Funds transferred successfully.');

EXCEPTION

WHEN OTHERS THEN

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

ROLLBACK; -- Rollback the transaction in case of an error

END;

END TransferFunds;

/

BEGIN

TransferFunds(p\_SourceAccountID => 1, p\_DestinationAccountID => 2, p\_Amount => 500);

END;

/

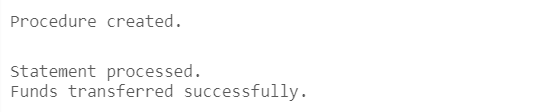
BEGIN

TransferFunds(p\_SourceAccountID => 1, p\_DestinationAccountID => 2, p\_Amount => 50000000000);

END;

/

**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.

**Code –**

CREATE OR REPLACE FUNCTION CalculateAge (

p\_DOB IN DATE

) RETURN NUMBER IS

v\_Age NUMBER;

BEGIN

SELECT FLOOR(MONTHS\_BETWEEN(SYSDATE, p\_DOB) / 12)

INTO v\_Age

FROM dual;

RETURN v\_Age;

EXCEPTION

WHEN OTHERS THEN

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

RETURN NULL; -- Return NULL in case of an error

END CalculateAge;

/

DECLARE

v\_Age NUMBER;

BEGIN

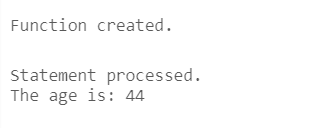
v\_Age := CalculateAge(TO\_DATE('1980-01-15', 'YYYY-MM-DD'));

DBMS\_OUTPUT.PUT\_LINE('The age is: ' || v\_Age);

END;

/

**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.

**Code –**

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment (

p\_LoanAmount IN NUMBER,

p\_InterestRate IN NUMBER,

p\_LoanDurationYears IN NUMBER

) RETURN NUMBER IS

v\_MonthlyInstallment NUMBER;

v\_MonthlyInterestRate NUMBER;

v\_NumPayments NUMBER;

BEGIN

v\_MonthlyInterestRate := p\_InterestRate / 12 / 100;

v\_NumPayments := p\_LoanDurationYears \* 12;

IF v\_MonthlyInterestRate > 0 THEN

v\_MonthlyInstallment := (p\_LoanAmount \* v\_MonthlyInterestRate) /

(1 - POWER(1 + v\_MonthlyInterestRate, -v\_NumPayments));

ELSE

-- If interest rate is 0, simply divide loan amount by number of payments

v\_MonthlyInstallment := p\_LoanAmount / v\_NumPayments;

END IF;

RETURN v\_MonthlyInstallment;

END CalculateMonthlyInstallment;

/

DECLARE

v\_MonthlyInstallment NUMBER;

BEGIN

v\_MonthlyInstallment := CalculateMonthlyInstallment(

p\_LoanAmount => 10000,

p\_InterestRate => 5,

p\_LoanDurationYears => 10

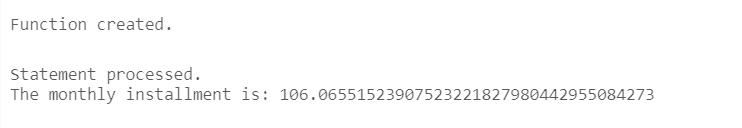
);

DBMS\_OUTPUT.PUT\_LINE('The monthly installment is: ' || v\_MonthlyInstallment);

END;

/

**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.

**Code –**

CREATE OR REPLACE FUNCTION HasSufficientBalance (

p\_AccountID IN Account.AccountID%TYPE,

p\_Amount IN NUMBER

) RETURN BOOLEAN IS

v\_Balance NUMBER;

BEGIN

SELECT Balance INTO v\_Balance

FROM Account

WHERE AccountID = p\_AccountID;

IF v\_Balance >= p\_Amount THEN

RETURN TRUE;

ELSE

RETURN FALSE;

END IF;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN FALSE;

WHEN OTHERS THEN

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

RETURN FALSE;

END HasSufficientBalance;

/

DECLARE

v\_HasSufficient BOOLEAN;

BEGIN

v\_HasSufficient := HasSufficientBalance(p\_AccountID => 222, p\_Amount => 500);

IF v\_HasSufficient THEN

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

ELSE

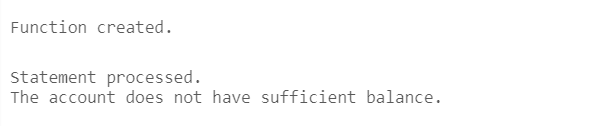
DBMS\_OUTPUT.PUT\_LINE('The account does not have sufficient balance.');

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.

**Code –**

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON Customers

FOR EACH ROW

BEGIN

:NEW.LastModified := SYSDATE;

END;

/

UPDATE Customers

SET Name = 'John Doe'

WHERE CustomerID = 1;

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.

**Code –**

CREATE TABLE AuditLog (

TransactionID NUMBER PRIMARY KEY,

AuditDate DATE,

Action VARCHAR2(50),

FOREIGN KEY (TransactionID) REFERENCES Transactions(TransactionID)

);

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog (TransactionID, AuditDate, Action)

VALUES (

:NEW.TransactionID,

SYSDATE,

'INSERT'

);

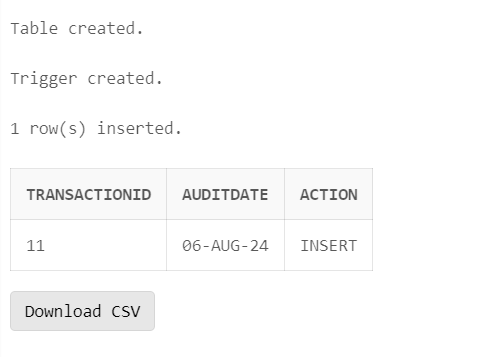
END;

/

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType) VALUES (11, 8, TO\_DATE('2024-07-28', 'YYYY-MM-DD'), 500.00, 'Withdrawal');

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.

**Code –**

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

DECLARE

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance

FROM Account

WHERE AccountID = :NEW.AccountID;

IF :NEW.TransactionType = 'Withdrawal' THEN

IF :NEW.Amount > v\_balance THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Withdrawal amount exceeds the current balance.');

END IF;

ELSIF :NEW.TransactionType = 'Deposit' THEN

IF :NEW.Amount <= 0 THEN

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

END IF;

ELSE

RAISE\_APPLICATION\_ERROR(-20003, 'Invalid transaction type. Must be DEPOSIT or WITHDRAWAL.');

END IF;

END;

/

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType) VALUES (11, 5, TO\_DATE('2024-07-20', 'YYYY-MM-DD'), 500999999.00, 'WITHDRAWAL')

**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.

**Code –**

DECLARE

CURSOR GenerateMonthlyStatements IS

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

FROM Transactions t

JOIN Account a ON t.AccountID = a.AccountID

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

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

v\_Transaction GenerateMonthlyStatements%ROWTYPE;

--v\_Transaction TransactionRec;

BEGIN

OPEN GenerateMonthlyStatements;

LOOP

FETCH GenerateMonthlyStatements INTO v\_Transaction;

EXIT WHEN GenerateMonthlyStatements%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Transaction ID: ' || v\_Transaction.TransactionID);

DBMS\_OUTPUT.PUT\_LINE('Account ID: ' || v\_Transaction.AccountID);

DBMS\_OUTPUT.PUT\_LINE('Transaction Date: ' || v\_Transaction.TransactionDate);

DBMS\_OUTPUT.PUT\_LINE('Amount: ' || v\_Transaction.Amount);

DBMS\_OUTPUT.PUT\_LINE('Transaction Type: ' || v\_Transaction.TransactionType);

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

END LOOP;

CLOSE GenerateMonthlyStatements;

EXCEPTION

WHEN OTHERS THEN

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

END;

/

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

VALUES (15, 1, TO\_DATE('2024-08-01', 'YYYY-MM-DD'), 1000, 'DEPOSIT');

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

VALUES (16, 2, TO\_DATE('2024-08-15', 'YYYY-MM-DD'), 500, 'WITHDRAWAL');

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

VALUES (17, 3, TO\_DATE('2024-08-20', 'YYYY-MM-DD'), 1500, 'DEPOSIT');

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

VALUES (18, 4, TO\_DATE('2024-07-25', 'YYYY-MM-DD'), 2000, 'DEPOSIT'); -- Older transaction

COMMIT;

**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.

**Code –**

DECLARE

CURSOR ApplyAnnualFee IS

SELECT AccountID, Balance

FROM Account;

v\_Account ApplyAnnualFee%ROWTYPE;

v\_AnnualFee NUMBER := 100;

BEGIN

OPEN ApplyAnnualFee;

LOOP

FETCH ApplyAnnualFee INTO v\_Account;

EXIT WHEN ApplyAnnualFee%NOTFOUND;

IF v\_Account.Balance >= v\_AnnualFee THEN

UPDATE Account

SET Balance = Balance - v\_AnnualFee

WHERE AccountID = v\_Account.AccountID;

ELSE

DBMS\_OUTPUT.PUT\_LINE('Insufficient balance for Account ID: ' || v\_Account.AccountID);

END IF;

END LOOP;

CLOSE ApplyAnnualFee;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

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

END;

/

SELECT \* FROM Account;

**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.

**Code –**

DECLARE

CURSOR UpdateLoanInterestRates IS

SELECT LoanID, InterestRate

FROM Loans;

v\_Loan UpdateLoanInterestRates%ROWTYPE;

v\_NewInterestRate NUMBER := 0.5;

BEGIN

OPEN UpdateLoanInterestRates;

LOOP

FETCH UpdateLoanInterestRates INTO v\_Loan;

EXIT WHEN UpdateLoanInterestRates%NOTFOUND;

UPDATE Loans

SET InterestRate = InterestRate + v\_NewInterestRate

WHERE LoanID = v\_Loan.LoanID;

DBMS\_OUTPUT.PUT\_LINE('Updated Loan ID: ' || v\_Loan.LoanID ||

', New Interest Rate: ' || (v\_Loan.InterestRate + v\_NewInterestRate));

END LOOP;

CLOSE UpdateLoanInterestRates;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

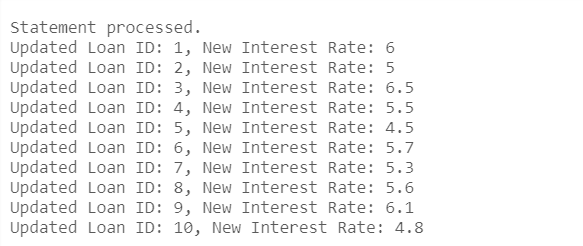
-- Handle any exceptions that occur

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

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.

**Code –**

CREATE OR REPLACE PACKAGE CustomerManagement AS

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

PROCEDURE UpdateCustomerDetails(p\_CustomerID IN NUMBER, p\_Name IN VARCHAR2, p\_Balance IN NUMBER);

FUNCTION GetCustomerBalance(p\_CustomerID IN NUMBER) RETURN NUMBER;

END CustomerManagement;

/

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

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

BEGIN

BEGIN

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

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

COMMIT;

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

WHEN OTHERS THEN

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

END;

END AddNewCustomer;

PROCEDURE UpdateCustomerDetails(p\_CustomerID IN NUMBER, p\_Name IN VARCHAR2, p\_Balance IN NUMBER) IS

BEGIN

BEGIN

UPDATE Customers

SET Name = p\_Name, Balance = p\_Balance, LastModified = SYSDATE

WHERE CustomerID = p\_CustomerID;

IF SQL%ROWCOUNT = 0 THEN

DBMS\_OUTPUT.PUT\_LINE('Customer with ID ' || p\_CustomerID || ' does not exist.');

ELSE

COMMIT;

END IF;

EXCEPTION

WHEN OTHERS THEN

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

END;

END UpdateCustomerDetails;

FUNCTION GetCustomerBalance(p\_CustomerID IN NUMBER) RETURN NUMBER IS

v\_Balance NUMBER;

BEGIN

BEGIN

SELECT Balance INTO v\_Balance

FROM Customers

WHERE CustomerID = p\_CustomerID;

RETURN v\_Balance;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('Customer with ID ' || p\_CustomerID || ' does not exist.');

RETURN NULL;

WHEN OTHERS THEN

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

RETURN NULL;

END;

END GetCustomerBalance;

END CustomerManagement;

/

BEGIN

CustomerManagement.AddNewCustomer(

p\_CustomerID => 123,

p\_Name => 'Johnyyyy Doeeeeee',

p\_DOB => TO\_DATE('1990-01-01', 'YYYY-MM-DD'),

p\_Balance => 10002

);

END;

/

SELECT \* FROM CUSTOMERS;

BEGIN

CustomerManagement.UpdateCustomerDetails(

p\_CustomerID => 123,

p\_Name => 'John Doe Updated',

p\_Balance => 15002

);

END;

/

SELECT \* FROM CUSTOMERS;

DECLARE

v\_Balance NUMBER;

BEGIN

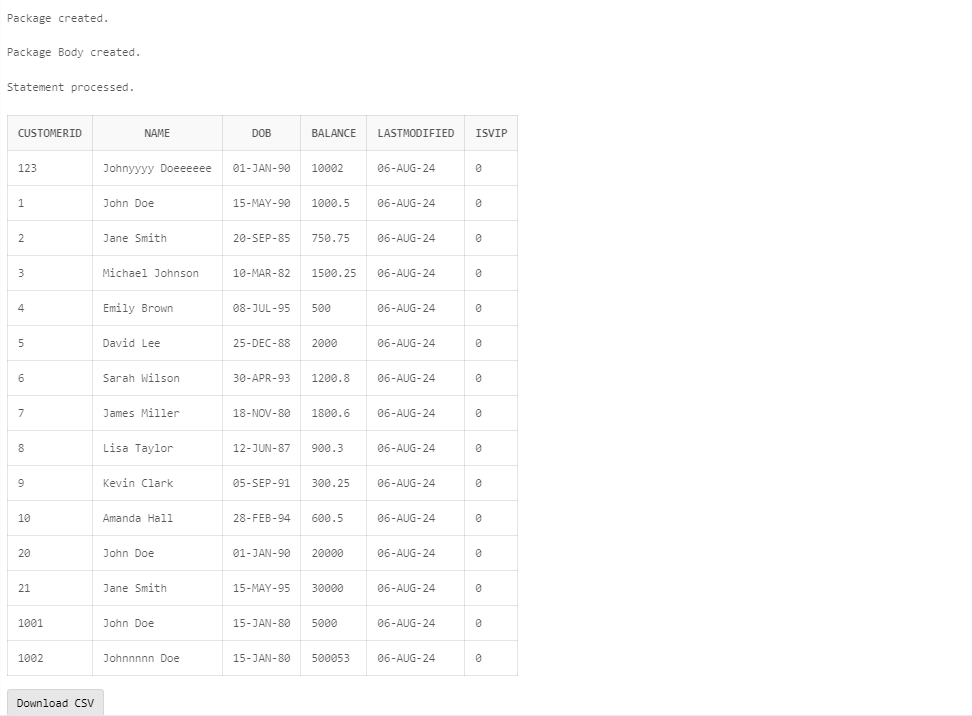
v\_Balance := CustomerManagement.GetCustomerBalance(p\_CustomerID => 123);

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

END;

/

**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.

**Code –**

CREATE OR REPLACE PACKAGE EmployeeManagement AS

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

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

FUNCTION CalculateAnnualSalary(p\_EmployeeID IN NUMBER) RETURN NUMBER;

END EmployeeManagement;

/

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

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

BEGIN

BEGIN

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

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

COMMIT;

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

WHEN OTHERS THEN

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

END;

END HireNewEmployee;

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

BEGIN

BEGIN

UPDATE Employees

SET Name = p\_Name, Position = p\_Position, Salary = p\_Salary, Department = p\_Department

WHERE EmployeeID = p\_EmployeeID;

IF SQL%ROWCOUNT = 0 THEN

DBMS\_OUTPUT.PUT\_LINE('Employee with ID ' || p\_EmployeeID || ' does not exist.');

ELSE

COMMIT;

END IF;

EXCEPTION

WHEN OTHERS THEN

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

END;

END UpdateEmployeeDetails;

FUNCTION CalculateAnnualSalary(p\_EmployeeID IN NUMBER) RETURN NUMBER IS

v\_Salary NUMBER;

v\_AnnualSalary NUMBER;

BEGIN

BEGIN

SELECT Salary INTO v\_Salary

FROM Employees

WHERE EmployeeID = p\_EmployeeID;

v\_AnnualSalary := v\_Salary \* 12;

RETURN v\_AnnualSalary;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('Employee with ID ' || p\_EmployeeID || ' does not exist.');

RETURN NULL;

WHEN OTHERS THEN

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

RETURN NULL;

END;

END CalculateAnnualSalary;

END EmployeeManagement;

/

BEGIN

EmployeeManagement.HireNewEmployee(

p\_EmployeeID => 101,

p\_Name => 'Alice Smith',

p\_Position => 'Software Engineer',

p\_Salary => 5000,

p\_Department => 'IT'

);

END;

/

SELECT \* FROM EMPLOYEES;

BEGIN

EmployeeManagement.UpdateEmployeeDetails(

p\_EmployeeID => 101,

p\_Name => 'Alice Johnson',

p\_Position => 'Senior Software Engineer',

p\_Salary => 6000,

p\_Department => 'IT'

);

END;

/

SELECT \* FROM EMPLOYEES;

DECLARE

v\_AnnualSalary NUMBER;

BEGIN

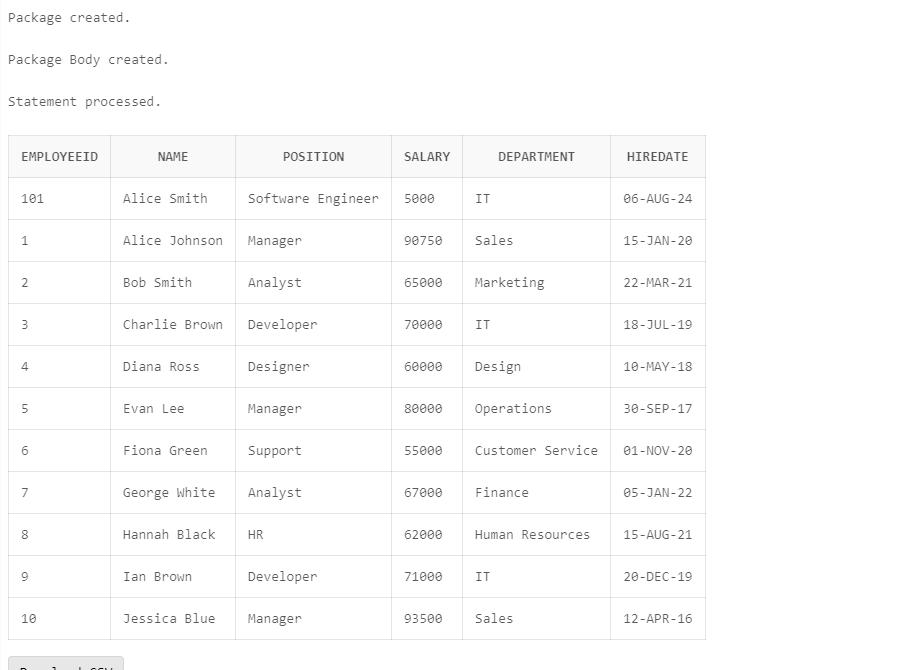
v\_AnnualSalary := EmployeeManagement.CalculateAnnualSalary(p\_EmployeeID => 101);

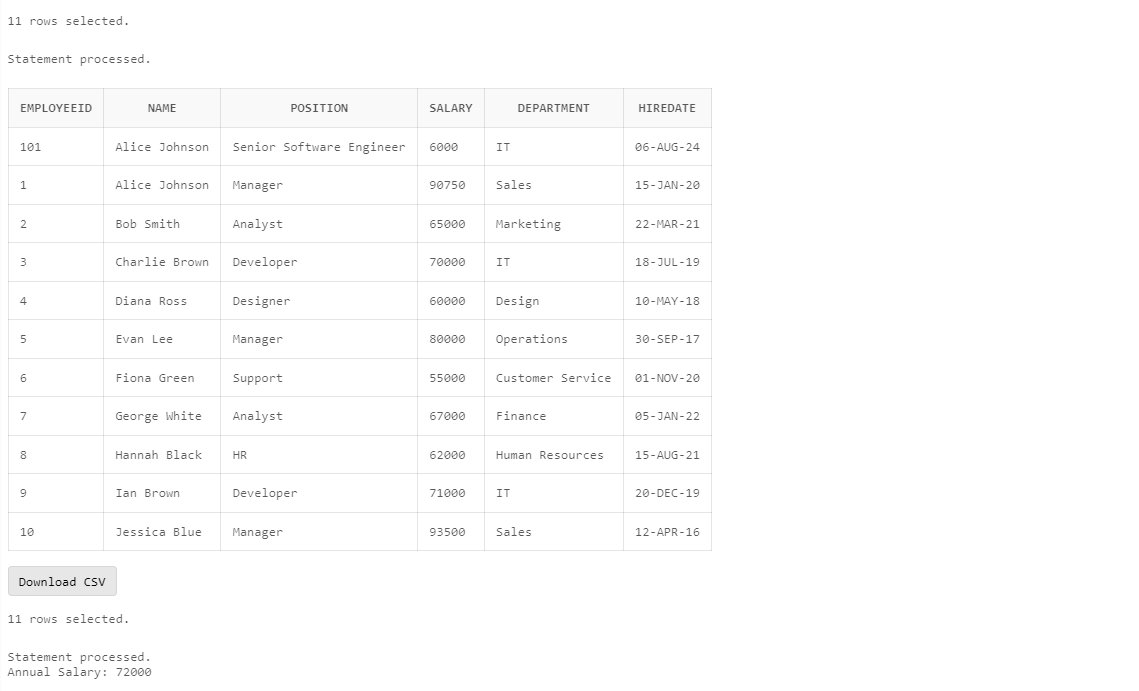
DBMS\_OUTPUT.PUT\_LINE('Annual Salary: ' || v\_AnnualSalary);

END;

/

**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.

**Code –**

CREATE OR REPLACE PACKAGE AccountOperations AS

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

PROCEDURE CloseAccount(p\_AccountID IN NUMBER);

FUNCTION GetTotalCustomerBalance(p\_CustomerID IN NUMBER) RETURN NUMBER;

END AccountOperations;

/

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

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

BEGIN

BEGIN

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

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

COMMIT;

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

WHEN OTHERS THEN

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

END;

END OpenNewAccount;

PROCEDURE CloseAccount(p\_AccountID IN NUMBER) IS

BEGIN

BEGIN

DELETE FROM Account

WHERE AccountID = p\_AccountID;

IF SQL%ROWCOUNT = 0 THEN

DBMS\_OUTPUT.PUT\_LINE('Account with ID ' || p\_AccountID || ' does not exist.');

ELSE

COMMIT;

END IF;

EXCEPTION

WHEN OTHERS THEN

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

END;

END CloseAccount;

FUNCTION GetTotalCustomerBalance(p\_CustomerID IN NUMBER) RETURN NUMBER IS

v\_TotalBalance NUMBER;

BEGIN

BEGIN

SELECT SUM(Balance) INTO v\_TotalBalance

FROM Account

WHERE CustomerID = p\_CustomerID;

IF v\_TotalBalance IS NULL THEN

v\_TotalBalance := 0;

END IF;

RETURN v\_TotalBalance;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN 0;

WHEN OTHERS THEN

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

RETURN NULL;

END;

END GetTotalCustomerBalance;

END AccountOperations;

/

BEGIN

AccountOperations.OpenNewAccount(

p\_AccountID => 2001,

p\_CustomerID => 1,

p\_AccountType => 'Savings',

p\_Balance => 5000

);

END;

/

SELECT \* FROM Account;

BEGIN

AccountOperations.CloseAccount(p\_AccountID => 21);

END;

/

SELECT \* FROM Account;

DECLARE

v\_TotalBalance NUMBER;

BEGIN

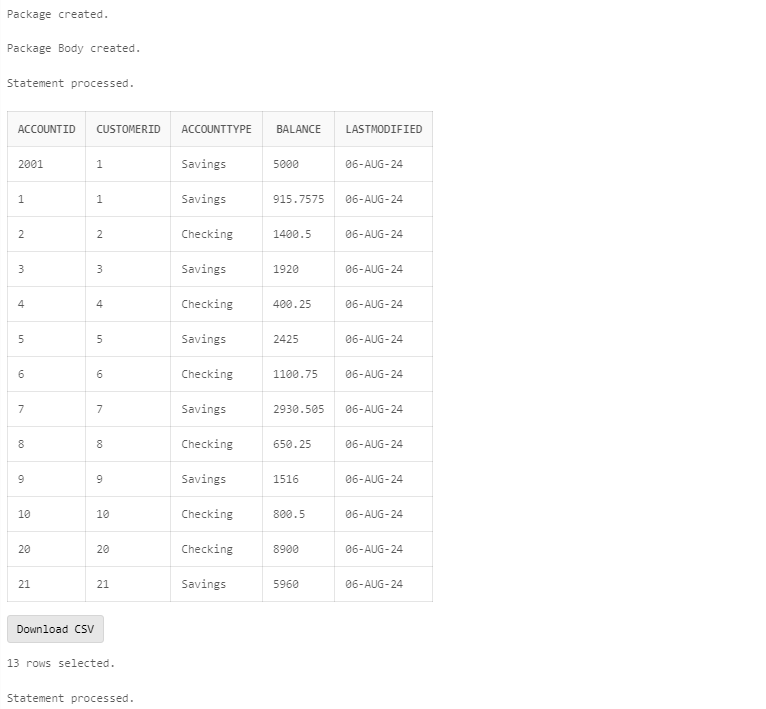
v\_TotalBalance := AccountOperations.GetTotalCustomerBalance(p\_CustomerID => 1);

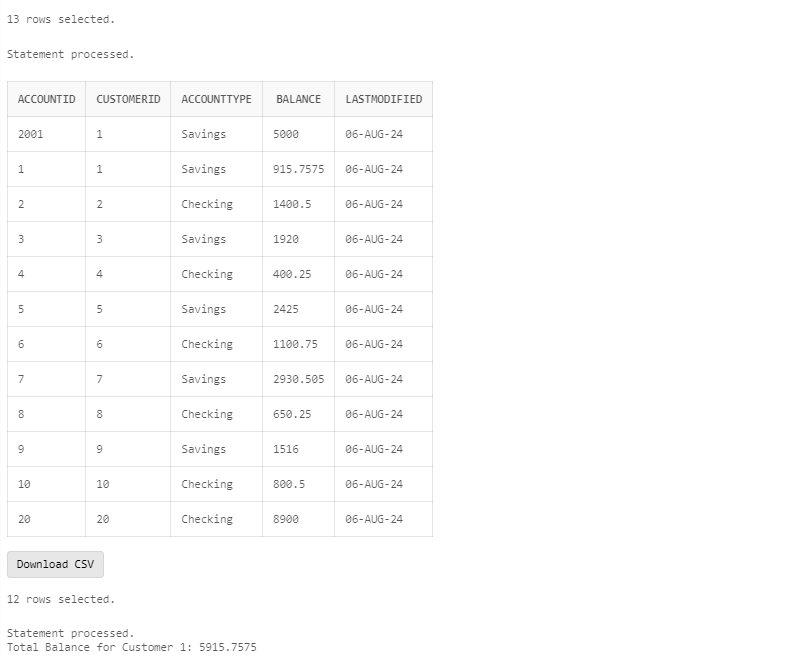
DBMS\_OUTPUT.PUT\_LINE('Total Balance for Customer 1: ' || v\_TotalBalance);

END;

/

**OUTPUT –**





**Schema to be Created**

*CREATE TABLE Customers (*

*CustomerID NUMBER PRIMARY KEY,*

*Name VARCHAR2(100),*

*DOB DATE,*

*Balance NUMBER,*

*LastModified DATE*

*);*

*CREATE TABLE Account (*

*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*

*);*

**Example Scripts for Sample 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'));*