**WEEK 2 PL SQL**

-- Customers

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 Customers (CustomerID, Name, DOB, Balance, LastModified) VALUES (3, 'GDC', TO\_DATE('1905-05-15', 'YYYY-MM-DD'), 2000, SYSDATE);

INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified) VALUES (99, 'Senior Citizen', TO\_DATE('1950-01-01','YYYY-MM-DD'), 225000, SYSDATE);

INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified) VALUES (199, 'Virat Kohli', TO\_DATE('1950-01-01','YYYY-MM-DD'), 225000, SYSDATE);

INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified) VALUES (777, 'Ravi Teja', TO\_DATE('1970-10-15', 'YYYY-MM-DD'), 7000, SYSDATE);

-- Accounts

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);

-- Transactions

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');

-- Loans

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

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

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

VALUES (99, 99, 10000, 7, SYSDATE, ADD\_MONTHS(SYSDATE, 60));

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

VALUES (101, 199, 12000, 6.5, SYSDATE, SYSDATE + 10);

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

VALUES (777, 777, 15000, 8, SYSDATE, SYSDATE + 10);

-- Employees

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

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

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

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

**Exercise 1: Control Structures**

***--Scenario 1: Senior Citizens (>60 yrs) Get 1% Reduced Loan Interest***

SET SERVEROUTPUT ON;

BEGIN

FOR c IN (SELECT \* FROM Customers c JOIN Loans l ON c.CustomerID = l.CustomerID) LOOP

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

UPDATE Loans

SET InterestRate = InterestRate - 1

WHERE LoanID = c.LoanID;

END IF;

END LOOP;

END;

/

***-- View affected records***

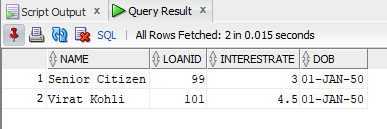
SELECT c.Name, l.LoanID, l.InterestRate, c.DOB

FROM Customers c

JOIN Loans l ON c.CustomerID = l.CustomerID

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

***OUTPUT***



***-- Scenario 2: Mark VIP Customers***

BEGIN

FOR c IN (SELECT \* FROM Customers) LOOP

IF c.Balance > 10000 THEN

UPDATE Customers SET IsVIP = 'TRUE' WHERE CustomerID = c.CustomerID;

ELSE

UPDATE Customers SET IsVIP = 'FALSE' WHERE CustomerID = c.CustomerID;

END IF;

END LOOP;

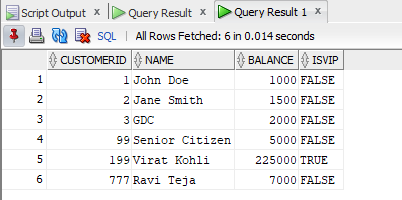
END;

/

***-- Check result***

SELECT CustomerID, Name, Balance, IsVIP FROM Customers;

***OUTPUT***



***--Scenario 3: Loan Reminders for Upcoming Due Dates***

SET SERVEROUTPUT ON;

BEGIN

FOR r IN (

SELECT c.Name, l.EndDate

FROM Loans l

JOIN Customers c ON l.CustomerID = c.CustomerID

WHERE l.EndDate <= SYSDATE + 30

) LOOP

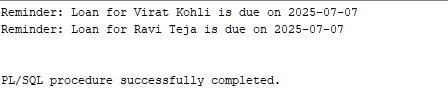
DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan for ' || r.Name || ' is due on ' || TO\_CHAR(r.EndDate, 'YYYY-MM-DD'));

END LOOP;

END;

/

***OUTPUT***



**Exercise 2: Error Handling**

***-- Scenario 1: Safe transfer funds***

CREATE OR REPLACE PROCEDURE SafeTransferFunds(p\_from NUMBER, p\_to NUMBER, p\_amount NUMBER) IS

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance FROM Accounts WHERE AccountID = p\_from;

IF v\_balance < p\_amount THEN

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

END IF;

UPDATE Accounts SET Balance = Balance - p\_amount WHERE AccountID = p\_from;

UPDATE Accounts SET Balance = Balance + p\_amount WHERE AccountID = p\_to;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END;

/

SET SERVEROUTPUT ON;

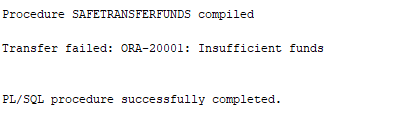
BEGIN

SafeTransferFunds(1, 2, 100000); -- Try large amount to force "Insufficient funds"

END;

/

***OUTPUT***



***-- Scenario 2: Update salary***

CREATE OR REPLACE PROCEDURE UpdateSalary(p\_emp\_id NUMBER, p\_percent NUMBER) IS

BEGIN

UPDATE Employees SET Salary = Salary + Salary \* (p\_percent / 100)

WHERE EmployeeID = p\_emp\_id;

IF SQL%ROWCOUNT = 0 THEN

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

END IF;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

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

END;

/

***--When I try to add invalid data***

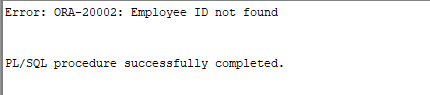
BEGIN

UpdateSalary(999, 10); -- Invalid ID to trigger error

END;

/

***OUTPUT***

******

***-- Scenario 3: Add new customer***

CREATE OR REPLACE PROCEDURE AddNewCustomer(p\_id NUMBER, p\_name VARCHAR2, p\_dob DATE, p\_balance NUMBER) IS

BEGIN

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

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

COMMIT;

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

END;

/

***--When i add existing data***

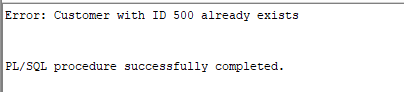
BEGIN

AddNewCustomer(500, 'MSD', TO\_DATE('1981-07-07', 'YYYY-MM-DD'), 18000);

END;

/

***OUTPUT***



**Exercise 3: Stored Procedures**

***-- Scenario 1: Process monthly interest***

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

BEGIN

UPDATE Accounts

SET Balance = Balance + (Balance \* 0.01)

WHERE AccountType = 'Savings';

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

COMMIT;

END;

/

***-- updates balances of Savings accounts by adding 1% interest.***

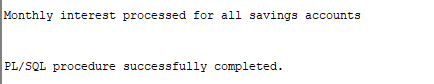
BEGIN

ProcessMonthlyInterest;

END;

/

***OUTPUT***



***-- Scenario 2: Update employee bonus***

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(p\_dept VARCHAR2, p\_bonus NUMBER) IS

BEGIN

UPDATE Employees

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

WHERE Department = p\_dept;

DBMS\_OUTPUT.PUT\_LINE('Bonus applied to department ' || p\_dept);

COMMIT;

END;

/

***--updates salaries of employees in a department.***

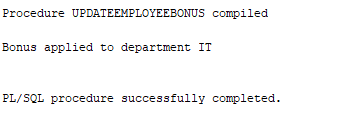
BEGIN

UpdateEmployeeBonus('IT', 15); -- Add 15% bonus to IT department

END;

/

***OUTPUT***

****

***-- Scenario 3: Transfer funds***

CREATE OR REPLACE PROCEDURE TransferFunds(p\_from NUMBER, p\_to NUMBER, p\_amount NUMBER) IS

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance FROM Accounts WHERE AccountID = p\_from;

IF v\_balance >= p\_amount THEN

UPDATE Accounts SET Balance = Balance - p\_amount WHERE AccountID = p\_from;

UPDATE Accounts SET Balance = Balance + p\_amount WHERE AccountID = p\_to;

DBMS\_OUTPUT.PUT\_LINE('Transfer successful!');

COMMIT;

ELSE

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

END IF;

END;

/

***--if enough balance is available.***

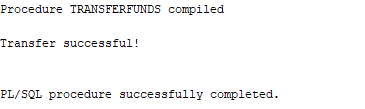
BEGIN

TransferFunds(1, 2, 50); -- Transfers ₹50 from account 1 to 2

END;

/

***OUTPUT***



***--Fails (shows output)***

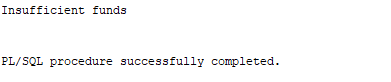
BEGIN

TransferFunds(1, 2, 100000); -- Too much, should fail

END;

/

***OUTPUT***



**Exercise 4: Functions**

***-- Scenario 1: Calculate age***

CREATE OR REPLACE FUNCTION CalculateAge(p\_dob DATE) RETURN NUMBER IS

BEGIN

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

END;

/

***-- Returns: Number (age in years)***

SELECT CalculateAge(TO\_DATE('1990-05-01', 'YYYY-MM-DD')) AS Age FROM dual;

***OUTPUT***



***-- Scenario 2: Calculate monthly installment***

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(p\_amount NUMBER, p\_rate NUMBER, p\_years NUMBER) RETURN NUMBER IS

v\_months NUMBER := p\_years \* 12;

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

BEGIN

RETURN (p\_amount \* v\_monthly\_rate) / (1 - POWER(1 + v\_monthly\_rate, -v\_months));

END;

/

***--Number (monthly EMI)***

SELECT CalculateMonthlyInstallment(100000, 10, 5) AS EMI FROM dual;

***OUTPUT***



***-- Scenario 3: Check sufficient balance***

CREATE OR REPLACE FUNCTION HasSufficientBalance(p\_acc\_id NUMBER, p\_amount NUMBER) RETURN BOOLEAN IS

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance FROM Accounts WHERE AccountID = p\_acc\_id;

RETURN v\_balance >= p\_amount;

END;

/

**-*-Boolean (TRUE or FALSE)***

DECLARE

result BOOLEAN;

BEGIN

result := HasSufficientBalance(1, 100);

IF result THEN

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

ELSE

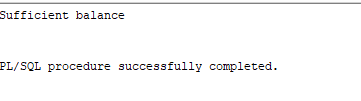
DBMS\_OUTPUT.PUT\_LINE('Insufficient balance');

END IF;

END;

/

***OUTPUT***



**Exercise 5: Triggers**

***-- Scenario 1: Update LastModified***

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON Customers

FOR EACH ROW

BEGIN

:NEW.LastModified := SYSDATE;

END;

/

***--Automatically updates the LastModified column when a customer record is updated.***

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

***-- Then check the updated LastModified***

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

***OUTPUT***



***-- Scenario 2: Audit log (Assume AuditLog table exists)***

CREATE TABLE AuditLog (

LogID NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,

ActionDate DATE,

AccountID NUMBER,

Amount NUMBER,

ActionType VARCHAR2(10)

);

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog (ActionDate, AccountID, Amount, ActionType)

VALUES (SYSDATE, :NEW.AccountID, :NEW.Amount, :NEW.TransactionType);

END;

/

***--Inserts a record into the AuditLog table whenever a new transaction is inserted into Transactions.***

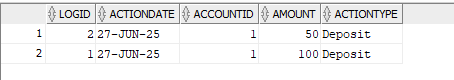
INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)

VALUES (3, 1, SYSDATE, 100, 'Deposit');

***-- Then check the audit log***

SELECT \* FROM AuditLog ORDER BY LogID DESC;

***OUTPUT***



***-- Scenario 3: Enforce rules***

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(-20003, 'Withdrawal exceeds balance');

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

RAISE\_APPLICATION\_ERROR(-20004, 'Deposit must be positive');

END IF;

END;

/

***-- should work***

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

VALUES (4, 1, SYSDATE, 50, 'Deposit');

***OUTPUT***

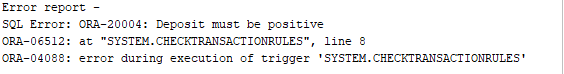


***-- should raise error***

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

VALUES (5, 1, SYSDATE, -100, 'Deposit');

***OUTPUT***

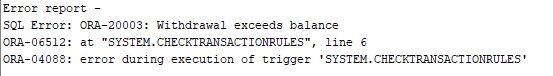


***-- too much***

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

VALUES (6, 1, SYSDATE, 1000000, 'Withdrawal');

***OUTPUT***



**Exercise 6: Cursors**

***-- Scenario 1: Monthly statements***

DECLARE

CURSOR c\_statements IS

SELECT c.Name, t.\* FROM Transactions t

JOIN Accounts a ON t.AccountID = a.AccountID

JOIN Customers c ON a.CustomerID = c.CustomerID

WHERE TO\_CHAR(t.TransactionDate, 'MMYYYY') = TO\_CHAR(SYSDATE, 'MMYYYY');

BEGIN

FOR r IN c\_statements LOOP

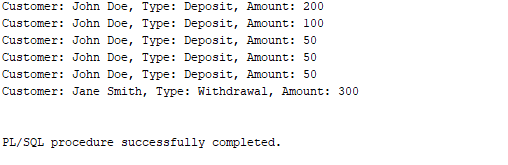
DBMS\_OUTPUT.PUT\_LINE('Customer: ' || r.Name || ', Type: ' || r.TransactionType || ', Amount: ' || r.Amount);

END LOOP;

END;

/

***OUTPUT***



***-- Scenario 2: Apply annual fee***

SET SERVEROUTPUT ON;

DECLARE

CURSOR c\_accounts IS SELECT AccountID, Balance FROM Accounts;

BEGIN

FOR acc IN c\_accounts LOOP

UPDATE Accounts

SET Balance = Balance - 100

WHERE AccountID = acc.AccountID;

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

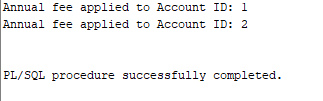
END LOOP;

COMMIT;

END;

/

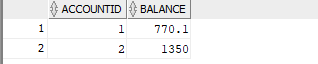
***OUTPUT***



***--Check updated balances:***

SELECT AccountID, Balance FROM Accounts;

***OUTPUT***



***-- Scenario 3: Update loan interest***

DECLARE

CURSOR c\_loans IS SELECT LoanID, InterestRate FROM Loans;

BEGIN

FOR l IN c\_loans LOOP

UPDATE Loans SET InterestRate = InterestRate + 0.5 WHERE LoanID = l.LoanID;

END LOOP;

COMMIT;

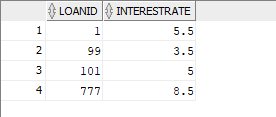
END;

/

***--Check updated interest rates:***

SELECT LoanID, InterestRate FROM Loans;

***OUTPUT***



**Exercise 7: Packages**

***-- Scenario 1: Customer package Package specification for customer operations***

CREATE OR REPLACE PACKAGE CustomerManagement AS

PROCEDURE AddCustomer(

p\_id NUMBER,

p\_name VARCHAR2,

p\_dob DATE,

p\_balance NUMBER

);

PROCEDURE UpdateCustomer(

p\_id NUMBER,

p\_balance NUMBER

);

FUNCTION GetBalance(p\_id NUMBER) RETURN NUMBER;

END;

/

**-*- Package body implementing customer operations***

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

PROCEDURE AddCustomer(

p\_id NUMBER,

p\_name VARCHAR2,

p\_dob DATE,

p\_balance NUMBER

) IS

BEGIN

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

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

END;

PROCEDURE UpdateCustomer(

p\_id NUMBER,

p\_balance NUMBER

) IS

BEGIN

UPDATE Customers

SET Balance = p\_balance,

LastModified = SYSDATE

WHERE CustomerID = p\_id;

END;

FUNCTION GetBalance(p\_id NUMBER) RETURN NUMBER IS

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance FROM Customers WHERE CustomerID = p\_id;

RETURN v\_balance;

END;

END;

/

***-- Add a customer***

BEGIN

CustomerManagement.AddCustomer(101, 'Rahul Dravid', TO\_DATE('1973-01-11', 'YYYY-MM-DD'), 5000);

END;

/

**-*- Update the balance of an existing customer***

BEGIN

CustomerManagement.UpdateCustomer(101, 7000);

END;

/

***-- Display the balance of a specific customer***

DECLARE

v\_balance NUMBER;

BEGIN

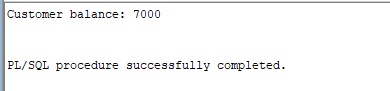
v\_balance := CustomerManagement.GetBalance(101);

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

END;

/

***OUTPUT***



***-- Scenario 2: Employee package***

***-- Package specification for employee operations***

CREATE OR REPLACE PACKAGE EmployeeManagement AS

PROCEDURE HireEmployee(p\_id NUMBER, p\_name VARCHAR2, p\_pos VARCHAR2, p\_salary NUMBER, p\_dept VARCHAR2);

PROCEDURE UpdateEmployee(p\_id NUMBER, p\_salary NUMBER);

FUNCTION AnnualSalary(p\_id NUMBER) RETURN NUMBER;

END;

/

***-- Package body implementing employee operations***

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

***-- Adds a new employee with the given details***

PROCEDURE HireEmployee(p\_id NUMBER, p\_name VARCHAR2, p\_pos VARCHAR2, p\_salary NUMBER, p\_dept VARCHAR2) IS

BEGIN

INSERT INTO Employees VALUES (p\_id, p\_name, p\_pos, p\_salary, p\_dept, SYSDATE);

END;

***-- Updates the salary of an existing employee***

PROCEDURE UpdateEmployee(p\_id NUMBER, p\_salary NUMBER) IS

BEGIN

UPDATE Employees SET Salary = p\_salary WHERE EmployeeID = p\_id;

END;

***-- Calculates and returns the annual salary of an employee***

FUNCTION AnnualSalary(p\_id NUMBER) RETURN NUMBER IS

v\_salary NUMBER;

BEGIN

SELECT Salary INTO v\_salary FROM Employees WHERE EmployeeID = p\_id;

RETURN v\_salary \* 12;

END;

END;

/

***-- Hire a new employee***

BEGIN

EmployeeManagement.HireEmployee(201, 'Anil Kumble', 'Coach', 85000, 'Sports');

END;

/

***-- Update the salary of an employee***

BEGIN

EmployeeManagement.UpdateEmployee(201, 90000);

END;

/

***-- Display annual salary of an employee***

DECLARE

v\_annual\_salary NUMBER;

BEGIN

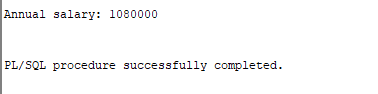
v\_annual\_salary := EmployeeManagement.AnnualSalary(201);

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

END;

/

***OUTPUT***



***-- Scenario 3: Account operations package***

***-- Package specification for account operations***

CREATE OR REPLACE PACKAGE AccountOperations AS

PROCEDURE OpenAccount(p\_id NUMBER, p\_cust NUMBER, p\_type VARCHAR2, p\_balance NUMBER);

PROCEDURE CloseAccount(p\_id NUMBER);

FUNCTION GetTotalBalance(p\_cust\_id NUMBER) RETURN NUMBER;

END;

/

***-- Package body implementing account operations***

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

PROCEDURE OpenAccount(p\_id NUMBER, p\_cust NUMBER, p\_type VARCHAR2, p\_balance NUMBER) IS

BEGIN

INSERT INTO Accounts VALUES (p\_id, p\_cust, p\_type, p\_balance, SYSDATE);

END;

PROCEDURE CloseAccount(p\_id NUMBER) IS

BEGIN

DELETE FROM Accounts WHERE AccountID = p\_id;

END;

FUNCTION GetTotalBalance(p\_cust\_id NUMBER) RETURN NUMBER IS

v\_total NUMBER := 0;

BEGIN

SELECT SUM(Balance) INTO v\_total FROM Accounts WHERE CustomerID = p\_cust\_id;

RETURN NVL(v\_total, 0);

END;

END;

/

***-- Open a new account for a customer***

BEGIN

AccountOperations.OpenAccount(302, 101, 'Savings', 18000);

END;

/

***-- Close the account by account ID***

BEGIN

AccountOperations.CloseAccount(301);

END;

/

***-- Display total balance of all accounts for a customer***

DECLARE

v\_total\_balance NUMBER;

BEGIN

v\_total\_balance := AccountOperations.GetTotalBalance(101);

DBMS\_OUTPUT.PUT\_LINE('Total balance across all accounts: ' || v\_total\_balance);

END;

/

***OUTPUT***

