**WEEK2 – PL/SQL**

**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:**

BEGIN

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

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

UPDATE Loans

SET InterestRate = InterestRate - 1

WHERE CustomerID = person.CustomerID;

END IF;

END LOOP;

COMMIT;

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.**

**Code:**

BEGIN

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

IF customer.Balance > 10000 THEN

UPDATE Customers

SET IsVIP = TRUE

WHERE CustomerID = customer.CustomerID;

END IF;

END LOOP;

COMMIT;

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.**

**Code:**

BEGIN

FOR loan 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.LoanID ||

' for customer ' || loan.Name ||

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

);

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.**

**Code:**

CREATE OR REPLACE PROCEDURE SafeTransferFunds (

p\_FromAccountID IN NUMBER,

p\_ToAccountID IN NUMBER,

p\_Amount IN NUMBER

) IS

v\_FromBalance NUMBER;

v\_ToBalance NUMBER;

BEGIN

SAVEPOINT start\_transaction;

SELECT Balance INTO v\_FromBalance

FROM Accounts

WHERE AccountID = p\_FromAccountID;

SELECT Balance INTO v\_ToBalance

FROM Accounts

WHERE AccountID = p\_ToAccountID;

IF v\_FromBalance < p\_Amount THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient funds in account ' || p\_FromAccountID);

END IF;

UPDATE Accounts

SET Balance = Balance - p\_Amount

WHERE AccountID = p\_FromAccountID;

UPDATE Accounts

SET Balance = Balance + p\_Amount

WHERE AccountID = p\_ToAccountID;

COMMIT;

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

EXCEPTION

WHEN OTHERS THEN

ROLLBACK TO start\_transaction;

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

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.**

**Code:**

CREATE OR REPLACE PROCEDURE AdjustEmployeeSalary (

emp\_id NUMBER,

pct\_increase NUMBER

) AS

not\_found EXCEPTION;

PRAGMA EXCEPTION\_INIT(not\_found, -20002); -- Custom error code for not found

current\_salary NUMBER;

BEGIN

-- Check if the employee exists and get the current salary

SELECT Salary INTO current\_salary

FROM Employees

WHERE EmployeeID = emp\_id;

-- Update the salary by the given percentage

UPDATE Employees

SET Salary = Salary + (Salary \* pct\_increase / 100)

WHERE EmployeeID = emp\_id;

-- Commit the transaction

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Salary updated successfully for EmployeeID ' || emp\_id);

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RAISE not\_found;

WHEN not\_found THEN

DBMS\_OUTPUT.PUT\_LINE('Error: Employee not found.');

ROLLBACK;

WHEN OTHERS THEN

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

ROLLBACK;

END AdjustEmployeeSalary;

**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 RegisterCustomer (

new\_customer\_id NUMBER,

new\_name VARCHAR2,

new\_dob DATE,

initial\_balance NUMBER

) AS

already\_exists EXCEPTION;

PRAGMA EXCEPTION\_INIT(already\_exists,-1);

customer\_count NUMBER;

BEGIN

SELECT COUNT(\*) INTO customer\_count

FROM Customers

WHERE CustomerID = new\_customer\_id;

IF customer\_count > 0 THEN

RAISE already\_exists;

END IF;

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

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

COMMIT;

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

EXCEPTION

WHEN already\_exists THEN

DBMS\_OUTPUT.PUT\_LINE('Error: Customer with ID ' || new\_customer\_id

|| ' already exists.');

ROLLBACK;

WHEN OTHERS THEN

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

ROLLBACK;

END RegisterCustomer;

**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 IS

v\_InterestRate NUMBER := 0.01; -- 1% interest rate

BEGIN

UPDATE Accounts

SET Balance = Balance + (Balance \* v\_InterestRate)

WHERE AccountType = 'Savings';

COMMIT;

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

END ProcessMonthlyInterest;

**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 VARCHAR2,

p\_BonusPercentage NUMBER

) IS

BEGIN

UPDATE Employees

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

WHERE Department = p\_Department;

COMMIT;

END UpdateEmployeeBonus;

**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\_FromAccountID NUMBER,

p\_ToAccountID NUMBER,

p\_Amount NUMBER

) AS

v\_FromBalance NUMBER;

v\_ToBalance NUMBER;

BEGIN

SELECT Balance INTO v\_FromBalance

FROM Accounts

WHERE AccountID = p\_FromAccountID;

IF v\_FromBalance < p\_Amount THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient funds in account ' || p\_FromAccountID);

END IF;

UPDATE Accounts

SET Balance = Balance - p\_Amount

WHERE AccountID = p\_FromAccountID;

UPDATE Accounts

SET Balance = Balance + p\_Amount

WHERE AccountID = p\_ToAccountID;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Funds transferred successfully from account ' || p\_FromAccountID || ' to account ' || p\_ToAccountID || '.');

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

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

ROLLBACK;

WHEN OTHERS THEN

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

ROLLBACK;

END TransferFunds;

**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 DATE

) RETURN NUMBER

IS

v\_Age NUMBER;

BEGIN

v\_Age := FLOOR(MONTHS\_BETWEEN(SYSDATE, p\_DOB) / 12);

RETURN v\_Age;

END CalculateAge;

**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 NUMBER,

p\_InterestRate NUMBER,

p\_LoanDurationYears NUMBER

) RETURN NUMBER

IS

v\_MonthlyRate NUMBER;

v\_NumPayments NUMBER;

v\_MonthlyInstallment NUMBER;

BEGIN

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

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

IF v\_MonthlyRate > 0 THEN

v\_MonthlyInstallment := (p\_LoanAmount \* v\_MonthlyRate \* POWER(1 + v\_MonthlyRate, v\_NumPayments)) /(POWER(1 + v\_MonthlyRate, v\_NumPayments) - 1);

ELSE

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

END IF;

RETURN v\_MonthlyInstallment;

END CalculateMonthlyInstallment;

**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 NUMBER,

p\_Amount NUMBER

) RETURN BOOLEAN

IS

v\_Balance NUMBER;

BEGIN

SELECT Balance INTO v\_Balance

FROM Accounts

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;

END HasSufficientBalance;

**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;

**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 OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog ( TransactionID, AccountID,TransactionDate,

Amount,TransactionType,Action)

VALUES (:NEW.TransactionID,:NEW.AccountID,:NEW.TransactionDate,

:NEW.Amount,:NEW.TransactionType,'INSERT' );

END;

**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\_AccountBalance NUMBER;

BEGIN

SELECT Balance INTO v\_AccountBalance

FROM Accounts

WHERE AccountID = :NEW.AccountID;

IF :NEW.TransactionType = 'Withdrawal' AND

:NEW.Amount >v\_AccountBalance THEN

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

END IF;

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

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

END IF;

END CheckTransactionRules;

**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:**

BEGIN

FOR mon\_cust IN (SELECT CustomerID FROM Customers) LOOP

DBMS\_OUTPUT.PUT\_LINE('Statement for Customer ID: ' || mon\_cust.CustomerID);

FOR txn IN (

SELECT \*

FROM Transactions

WHERE AccountID IN (

SELECT AccountID

FROM Accounts

WHERE CustomerID = mon\_cust.CustomerID

)

AND TransactionDate BETWEEN TRUNC(SYSDATE, 'MM') AND LAST\_DAY(SYSDATE)

) LOOP

DBMS\_OUTPUT.PUT\_LINE(

'Transaction ID: ' || txn.TransactionID ||

', Amount: ' || txn.Amount ||

', Type: ' || txn.TransactionType

);

END LOOP;

END LOOP;

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.**

**Code:**

BEGIN

FOR acc IN (SELECT AccountID FROM Accounts) LOOP

UPDATE Accounts

SET Balance = Balance - 100

WHERE AccountID = acc.AccountID;

END LOOP;

COMMIT;

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.**

**Code:**

BEGIN

FOR loan IN (SELECT LoanID FROM Loans) LOOP

UPDATE Loans

SET InterestRate = InterestRate+0.5

WHERE LoanID = loan.LoanID;

END LOOP;

COMMIT;

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.**

**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\_DOB IN DATE,

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

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('Error: Customer ID already exists.');

ROLLBACK;

WHEN OTHERS THEN

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

ROLLBACK;

END AddNewCustomer;

PROCEDURE UpdateCustomerDetails(

p\_CustomerID IN NUMBER,

p\_Name IN VARCHAR2,

p\_DOB IN DATE,

p\_Balance IN NUMBER

) IS

BEGIN

UPDATE Customers

SET Name = p\_Name,

DOB = p\_DOB,

Balance = p\_Balance,

LastModified = SYSDATE

WHERE CustomerID = p\_CustomerID;

COMMIT;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('Error: Customer ID not found.');

ROLLBACK;

WHEN OTHERS THEN

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

ROLLBACK;

END UpdateCustomerDetails;

FUNCTION GetCustomerBalance(

p\_CustomerID IN NUMBER

) RETURN NUMBER IS

v\_Balance NUMBER;

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('Error: Customer ID not found.');

RETURN NULL;

WHEN OTHERS THEN

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

RETURN NULL;

END GetCustomerBalance;

END CustomerManagement;

**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 HireEmployee(

p\_EmployeeID IN NUMBER,

p\_Name IN VARCHAR2,

p\_Position IN VARCHAR2,

p\_Salary IN NUMBER,

p\_Department IN VARCHAR2,

p\_HireDate IN DATE

);

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 HireEmployee(

p\_EmployeeID IN NUMBER,

p\_Name IN VARCHAR2,

p\_Position IN VARCHAR2,

p\_Salary IN NUMBER,

p\_Department IN VARCHAR2,

p\_HireDate IN DATE

) IS

BEGIN

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

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

COMMIT;

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

DBMS\_OUTPUT.PUT\_LINE('Error: Employee ID already exists.');

ROLLBACK;

WHEN OTHERS THEN

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

ROLLBACK;

END HireEmployee;

PROCEDURE UpdateEmployeeDetails(

p\_EmployeeID IN NUMBER,

p\_Name IN VARCHAR2,

p\_Position IN VARCHAR2,

p\_Salary IN NUMBER,

p\_Department IN VARCHAR2

) IS

BEGIN

UPDATE Employees

SET Name = p\_Name,

Position = p\_Position,

Salary = p\_Salary,

Department = p\_Department

WHERE EmployeeID = p\_EmployeeID;

COMMIT;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

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

ROLLBACK;

WHEN OTHERS THEN

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

ROLLBACK;

END UpdateEmployeeDetails;

FUNCTION CalculateAnnualSalary(

p\_EmployeeID IN NUMBER

) RETURN NUMBER IS

v\_Salary NUMBER;

v\_AnnualSalary NUMBER;

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('Error: Employee ID not found.');

RETURN NULL;

WHEN OTHERS THEN

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

RETURN NULL;

END CalculateAnnualSalary;

END EmployeeManagement;

**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 OpenAccount(

p\_AccountID IN NUMBER,

p\_CustomerID IN NUMBER,

p\_AccountType IN VARCHAR2,

p\_Balance IN NUMBER

);

PROCEDURE CloseAccount(

p\_AccountID IN NUMBER

);

FUNCTION GetTotalBalance(

p\_CustomerID IN NUMBER

) RETURN NUMBER;

END AccountOperations;

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

PROCEDURE OpenAccount(

p\_AccountID IN NUMBER,

p\_CustomerID IN NUMBER,

p\_AccountType IN VARCHAR2,

p\_Balance IN NUMBER

) IS

BEGIN

INSERT INTO Accounts (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('Error: Account ID already exists.');

ROLLBACK;

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('Error: Customer ID not found.');

ROLLBACK;

WHEN OTHERS THEN

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

ROLLBACK;

END OpenAccount;

PROCEDURE CloseAccount(

p\_AccountID IN NUMBER

) IS

BEGIN

DELETE FROM Accounts

WHERE AccountID = p\_AccountID;

COMMIT;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('Error: Account ID not found.');

ROLLBACK;

WHEN OTHERS THEN

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

ROLLBACK;

END CloseAccount;

FUNCTION GetTotalBalance(

p\_CustomerID IN NUMBER

) RETURN NUMBER IS

v\_TotalBalance NUMBER;

BEGIN

SELECT SUM(Balance) INTO v\_TotalBalance

FROM Accounts

WHERE CustomerID = p\_CustomerID;

RETURN v\_TotalBalance;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('Error: Customer ID not found.');

RETURN 0;

WHEN OTHERS THEN

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

RETURN 0;

END GetTotalBalance;

END AccountOperations;