**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.

BEGIN

FOR customer IN (SELECT customer\_id, age, loan\_interest\_rate

FROM customers) LOOP

IF customer.age > 60 THEN

UPDATE loans

SET loan\_interest\_rate = loan\_interest\_rate - 1

WHERE customer\_id = customer.customer\_id;

END IF;

END LOOP;

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.

BEGIN

FOR customer IN (SELECT customer\_id, balance

FROM customers) LOOP

IF customer.balance > 10000 THEN

UPDATE customers

SET IsVIP = 'TRUE'

WHERE customer\_id = customer.customer\_id;

END IF;

END LOOP;

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.

BEGIN

FOR loan IN (SELECT customer\_id, due\_date

FROM loans

WHERE due\_date BETWEEN SYSDATE AND SYSDATE + 30) LOOP

DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan for customer ' || loan.customer\_id || ' is due on ' || TO\_CHAR(loan.due\_date, '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.

CREATE OR REPLACE PROCEDURE SafeTransferFunds (

p\_from\_account\_id IN NUMBER,

p\_to\_account\_id IN NUMBER,

p\_amount IN NUMBER

) IS

insufficient\_funds EXCEPTION;

BEGIN

-- Check if the from\_account has sufficient funds

DECLARE

v\_balance NUMBER;

BEGIN

SELECT balance INTO v\_balance

FROM accounts

WHERE account\_id = p\_from\_account\_id;

IF v\_balance < p\_amount THEN

RAISE insufficient\_funds;

END IF;

END;

-- Perform the transfer

UPDATE accounts

SET balance = balance - p\_amount

WHERE account\_id = p\_from\_account\_id;

UPDATE accounts

SET balance = balance + p\_amount

WHERE account\_id = p\_to\_account\_id;

COMMIT;

EXCEPTION

WHEN insufficient\_funds THEN

ROLLBACK;

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

WHEN OTHERS THEN

ROLLBACK;

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

END SafeTransferFunds;

/

**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 IN NUMBER,

p\_percentage IN NUMBER

) IS

employee\_not\_found EXCEPTION;

BEGIN

-- Update the salary

UPDATE employees

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

WHERE employee\_id = p\_employee\_id;

-- Check if the update was successful

IF SQL%ROWCOUNT = 0 THEN

RAISE employee\_not\_found;

END IF;

COMMIT;

EXCEPTION

WHEN employee\_not\_found THEN

ROLLBACK;

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

WHEN OTHERS THEN

ROLLBACK;

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

END UpdateSalary;

/

**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 IN NUMBER,

p\_customer\_name IN VARCHAR2,

p\_customer\_age IN NUMBER,

p\_customer\_balance IN NUMBER

) IS

duplicate\_customer EXCEPTION;

BEGIN

-- Insert the new customer

INSERT INTO customers (customer\_id, name, age, balance)

VALUES (p\_customer\_id, p\_customer\_name, p\_customer\_age, p\_customer\_balance);

COMMIT;

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

ROLLBACK;

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

WHEN OTHERS THEN

ROLLBACK;

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

END AddNewCustomer;

/

**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 IS

BEGIN

UPDATE savings\_accounts

SET balance = balance \* 1.01;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

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.

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (

p\_department\_id IN NUMBER,

p\_bonus\_percentage IN NUMBER

) IS

BEGIN

UPDATE employees

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

WHERE department\_id = p\_department\_id;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

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.

CREATE OR REPLACE PROCEDURE TransferFunds (

p\_from\_account\_id IN NUMBER,

p\_to\_account\_id IN NUMBER,

p\_amount IN NUMBER

) IS

insufficient\_funds EXCEPTION;

BEGIN

-- Check if the from\_account has sufficient funds

DECLARE

v\_balance NUMBER;

BEGIN

SELECT balance INTO v\_balance

FROM accounts

WHERE account\_id = p\_from\_account\_id;

IF v\_balance < p\_amount THEN

RAISE insufficient\_funds;

END IF;

END;

-- Perform the transfer

UPDATE accounts

SET balance = balance - p\_amount

WHERE account\_id = p\_from\_account\_id;

UPDATE accounts

SET balance = balance + p\_amount

WHERE account\_id = p\_to\_account\_id;

COMMIT;

EXCEPTION

WHEN insufficient\_funds THEN

ROLLBACK;

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

WHEN OTHERS THEN

ROLLBACK;

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

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.

CREATE OR REPLACE FUNCTION CalculateAge (

p\_date\_of\_birth DATE

) RETURN NUMBER IS

v\_age NUMBER;

BEGIN

SELECT FLOOR((SYSDATE - p\_date\_of\_birth) / 365.25) INTO v\_age

FROM dual;

RETURN v\_age;

EXCEPTION

WHEN OTHERS THEN

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

RETURN NULL;

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.

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment (

p\_loan\_amount IN NUMBER,

p\_annual\_interest\_rate IN NUMBER,

p\_loan\_duration\_years IN NUMBER

) RETURN NUMBER IS

v\_monthly\_interest\_rate NUMBER;

v\_number\_of\_payments NUMBER;

v\_monthly\_installment NUMBER;

BEGIN

v\_monthly\_interest\_rate := p\_annual\_interest\_rate / 12 / 100;

v\_number\_of\_payments := p\_loan\_duration\_years \* 12;

IF v\_monthly\_interest\_rate = 0 THEN

v\_monthly\_installment := p\_loan\_amount / v\_number\_of\_payments;

ELSE

v\_monthly\_installment := p\_loan\_amount \* v\_monthly\_interest\_rate / (1 - POWER(1 + v\_monthly\_interest\_rate, -v\_number\_of\_payments));

END IF;

RETURN v\_monthly\_installment;

EXCEPTION

WHEN OTHERS THEN

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

RETURN NULL;

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.

CREATE OR REPLACE FUNCTION HasSufficientBalance (

p\_account\_id IN NUMBER,

p\_amount IN NUMBER

) RETURN BOOLEAN IS

v\_balance NUMBER;

BEGIN

SELECT balance INTO v\_balance

FROM accounts

WHERE account\_id = p\_account\_id;

IF v\_balance >= p\_amount THEN

RETURN TRUE;

ELSE

RETURN FALSE;

END IF;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

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

RETURN FALSE;

WHEN OTHERS THEN

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

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.

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON customers

FOR EACH ROW

BEGIN

:NEW.LastModified := SYSDATE;

END UpdateCustomerLastModified;

/

**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 (transaction\_id, account\_id, transaction\_date, amount, transaction\_type)

VALUES (:NEW.transaction\_id, :NEW.account\_id, :NEW.transaction\_date, :NEW.amount, :NEW.transaction\_type);

END LogTransaction;

/

**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

IF :NEW.transaction\_type = 'Withdrawal' THEN

SELECT balance INTO v\_balance

FROM accounts

WHERE account\_id = :NEW.account\_id;

IF :NEW.amount > v\_balance THEN

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

END IF;

ELSIF :NEW.transaction\_type = 'Deposit' THEN

IF :NEW.amount <= 0 THEN

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

END IF;

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.

DECLARE

CURSOR transaction\_cursor IS

SELECT c.CustomerID, c.Name, t.TransactionID, t.TransactionDate, t.Amount, t.TransactionType

FROM Customers c

JOIN Accounts a ON c.CustomerID = a.CustomerID

JOIN Transactions t ON a.AccountID = t.AccountID

WHERE TO\_CHAR(t.TransactionDate, 'YYYY-MM') = TO\_CHAR(SYSDATE, 'YYYY-MM');

v\_customer\_id Customers.CustomerID%TYPE;

v\_name Customers.Name%TYPE;

v\_transaction\_id Transactions.TransactionID%TYPE;

v\_transaction\_date Transactions.TransactionDate%TYPE;

v\_amount Transactions.Amount%TYPE;

v\_transaction\_type Transactions.TransactionType%TYPE;

BEGIN

OPEN transaction\_cursor;

LOOP

FETCH transaction\_cursor INTO v\_customer\_id, v\_name, v\_transaction\_id, v\_transaction\_date, v\_amount, v\_transaction\_type;

EXIT WHEN transaction\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Customer ID: ' || v\_customer\_id);

DBMS\_OUTPUT.PUT\_LINE('Name: ' || v\_name);

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

DBMS\_OUTPUT.PUT\_LINE('Transaction Date: ' || TO\_CHAR(v\_transaction\_date, 'YYYY-MM-DD'));

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

DBMS\_OUTPUT.PUT\_LINE('Transaction Type: ' || v\_transaction\_type);

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

END LOOP;

CLOSE transaction\_cursor;

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.

DECLARE

CURSOR account\_cursor IS

SELECT AccountID, Balance

FROM Accounts;

v\_account\_id Accounts.AccountID%TYPE;

v\_balance Accounts.Balance%TYPE;

v\_annual\_fee CONSTANT NUMBER := 50; -- Assuming a fixed annual fee of 50

BEGIN

OPEN account\_cursor;

LOOP

FETCH account\_cursor INTO v\_account\_id, v\_balance;

EXIT WHEN account\_cursor%NOTFOUND;

UPDATE Accounts

SET Balance = Balance - v\_annual\_fee,

LastModified = SYSDATE

WHERE AccountID = v\_account\_id;

DBMS\_OUTPUT.PUT\_LINE('Applied annual fee to account ID: ' || v\_account\_id);

END LOOP;

CLOSE account\_cursor;

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.

DECLARE

CURSOR loan\_cursor IS

SELECT LoanID, InterestRate

FROM Loans;

v\_loan\_id Loans.LoanID%TYPE;

v\_interest\_rate Loans.InterestRate%TYPE;

v\_new\_interest\_rate CONSTANT NUMBER := 0.05; -- Assuming a new fixed interest rate of 5%

BEGIN

OPEN loan\_cursor;

LOOP

FETCH loan\_cursor INTO v\_loan\_id, v\_interest\_rate;

EXIT WHEN loan\_cursor%NOTFOUND;

UPDATE Loans

SET InterestRate = v\_new\_interest\_rate

WHERE LoanID = v\_loan\_id;

DBMS\_OUTPUT.PUT\_LINE('Updated interest rate for loan ID: ' || v\_loan\_id);

END LOOP;

CLOSE loan\_cursor;

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.

*Specifications:*

CREATE OR REPLACE PACKAGE CustomerManagement IS

PROCEDURE AddNewCustomer(

p\_customer\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_dob IN DATE,

p\_balance IN NUMBER

);

PROCEDURE UpdateCustomerDetails(

p\_customer\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_dob IN DATE

);

FUNCTION GetCustomerBalance(

p\_customer\_id IN NUMBER

) RETURN NUMBER;

END CustomerManagement;

/

*Body:*

*CREATE OR REPLACE PACKAGE BODY CustomerManagement IS*

*PROCEDURE AddNewCustomer(*

*p\_customer\_id 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\_customer\_id, p\_name, p\_dob, p\_balance, SYSDATE);*

*COMMIT;*

*EXCEPTION*

*WHEN DUP\_VAL\_ON\_INDEX THEN*

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

*END AddNewCustomer;*

*PROCEDURE UpdateCustomerDetails(*

*p\_customer\_id IN NUMBER,*

*p\_name IN VARCHAR2,*

*p\_dob IN DATE*

*) IS*

*BEGIN*

*UPDATE Customers*

*SET Name = p\_name, DOB = p\_dob, LastModified = SYSDATE*

*WHERE CustomerID = p\_customer\_id;*

*COMMIT;*

*EXCEPTION*

*WHEN NO\_DATA\_FOUND THEN*

*DBMS\_OUTPUT.PUT\_LINE('Error: Customer ID ' || p\_customer\_id || ' not found.');*

*END UpdateCustomerDetails;*

*FUNCTION GetCustomerBalance(*

*p\_customer\_id IN 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*

*DBMS\_OUTPUT.PUT\_LINE('Error: Customer ID ' || p\_customer\_id || ' not found.');*

*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.

*Specification:*

CREATE OR REPLACE PACKAGE EmployeeManagement IS

PROCEDURE HireNewEmployee(

p\_employee\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_position IN VARCHAR2,

p\_salary IN NUMBER,

p\_department IN VARCHAR2,

p\_hire\_date IN DATE

);

PROCEDURE UpdateEmployeeDetails(

p\_employee\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_position IN VARCHAR2,

p\_salary IN NUMBER,

p\_department IN VARCHAR2

);

FUNCTION CalculateAnnualSalary(

p\_employee\_id IN NUMBER

) RETURN NUMBER;

END EmployeeManagement;

/

*Body*

CREATE OR REPLACE PACKAGE BODY EmployeeManagement IS

PROCEDURE HireNewEmployee(

p\_employee\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_position IN VARCHAR2,

p\_salary IN NUMBER,

p\_department IN VARCHAR2,

p\_hire\_date IN DATE

) IS

BEGIN

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

VALUES (p\_employee\_id, p\_name, p\_position, p\_salary, p\_department, p\_hire\_date);

COMMIT;

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

END HireNewEmployee;

PROCEDURE UpdateEmployeeDetails(

p\_employee\_id 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\_employee\_id;

COMMIT;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

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

END UpdateEmployeeDetails;

FUNCTION CalculateAnnualSalary(

p\_employee\_id IN NUMBER

) RETURN NUMBER IS

v\_salary NUMBER;

BEGIN

SELECT Salary \* 12 INTO v\_salary

FROM Employees

WHERE EmployeeID = p\_employee\_id;

RETURN v\_salary;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

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

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.

*Specification*

CREATE OR REPLACE PACKAGE AccountOperations IS

PROCEDURE OpenNewAccount(

p\_account\_id IN NUMBER,

p\_customer\_id IN NUMBER,

p\_account\_type IN VARCHAR2,

p\_balance IN NUMBER

);

PROCEDURE CloseAccount(

p\_account\_id IN NUMBER

);

FUNCTION GetTotalBalance(

p\_customer\_id IN NUMBER

) RETURN NUMBER;

END AccountOperations;

/

*Body*

CREATE OR REPLACE PACKAGE BODY AccountOperations IS

PROCEDURE OpenNewAccount(

p\_account\_id IN NUMBER,

p\_customer\_id IN NUMBER,

p\_account\_type IN VARCHAR2,

p\_balance IN NUMBER

) IS

BEGIN

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

VALUES (p\_account\_id, p\_customer\_id, p\_account\_type, p\_balance, SYSDATE);

COMMIT;

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

END OpenNewAccount;

PROCEDURE CloseAccount(

p\_account\_id IN NUMBER

) IS

BEGIN

DELETE FROM Accounts

WHERE AccountID = p\_account\_id;

COMMIT;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

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

END CloseAccount;

FUNCTION GetTotalBalance(

p\_customer\_id IN NUMBER

) RETURN NUMBER IS

v\_total\_balance NUMBER;

BEGIN

SELECT SUM(Balance) INTO v\_total\_balance

FROM Accounts

WHERE CustomerID = p\_customer\_id;

RETURN v\_total\_balance;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

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

RETURN NULL;

END GetTotalBalance;

END AccountOperations;

/