**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.

DECLARE

CURSOR cur\_customers IS

SELECT CustomerID, LoanID, InterestRate

FROM Loans

WHERE CustomerID IN (SELECT CustomerID FROM Customers WHERE TRUNC(MONTHS\_BETWEEN(SYSDATE, DOB) / 12) > 60);

v\_customer\_id Loans.CustomerID%TYPE;

v\_loan\_id Loans.LoanID%TYPE;

v\_interest\_rate Loans.InterestRate%TYPE;

BEGIN

OPEN cur\_customers;

LOOP

FETCH cur\_customers INTO v\_customer\_id, v\_loan\_id, v\_interest\_rate;

EXIT WHEN cur\_customers%NOTFOUND;

UPDATE Loans

SET InterestRate = InterestRate - 1

WHERE LoanID = v\_loan\_id;

DBMS\_OUTPUT.PUT\_LINE('Applied 1% discount to loan ID: ' || v\_loan\_id);

END LOOP;

CLOSE cur\_customers;

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.

DECLARE

CURSOR cur\_customers IS

SELECT CustomerID

FROM Customers

WHERE Balance > 10000;

v\_customer\_id Customers.CustomerID%TYPE;

BEGIN

OPEN cur\_customers;

LOOP

FETCH cur\_customers INTO v\_customer\_id;

EXIT WHEN cur\_customers%NOTFOUND;

UPDATE Customers

SET IsVIP = TRUE

WHERE CustomerID = v\_customer\_id;

DBMS\_OUTPUT.PUT\_LINE('Customer ID ' || v\_customer\_id || ' has been promoted to VIP status.');

END LOOP;

CLOSE cur\_customers;

END;

**Scenario 3:** The bank wants to send reminders to customers whose loans are due within the next 30 days.

* **Question:** Write a PL/SQL block that fetches all loans due in the next 30 days and prints a reminder message for each customer.

DECLARE

CURSOR cur\_loans IS

SELECT CustomerID, LoanID, EndDate

FROM Loans

WHERE EndDate BETWEEN SYSDATE AND SYSDATE + 30;

v\_customer\_id Loans.CustomerID%TYPE;

v\_loan\_id Loans.LoanID%TYPE;

v\_end\_date Loans.EndDate%TYPE;

BEGIN

OPEN cur\_loans;

LOOP

FETCH cur\_loans INTO v\_customer\_id, v\_loan\_id, v\_end\_date;

EXIT WHEN cur\_loans%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Reminder: Customer ID ' || v\_customer\_id || ', Loan ID ' || v\_loan\_id || ' is due on ' || TO\_CHAR(v\_end\_date, 'YYYY-MM-DD'));

END LOOP;

CLOSE cur\_loans;

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 Accounts.AccountID%TYPE,

p\_to\_account\_id IN Accounts.AccountID%TYPE,

p\_amount IN NUMBER

) IS

e\_insufficient\_funds EXCEPTION;

v\_balance Accounts.Balance%TYPE;

BEGIN

-- Check balance of the source account

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

IF v\_balance < p\_amount THEN

RAISE e\_insufficient\_funds;

END IF;

-- Deduct amount from source account

UPDATE Accounts

SET Balance = Balance - p\_amount

WHERE AccountID = p\_from\_account\_id;

-- Add amount to destination account

UPDATE Accounts

SET Balance = Balance + p\_amount

WHERE AccountID = p\_to\_account\_id;

EXCEPTION

WHEN e\_insufficient\_funds THEN

DBMS\_OUTPUT.PUT\_LINE('Error: Insufficient funds in account ' || p\_from\_account\_id);

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('Error: ' || 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.

CREATE OR REPLACE PROCEDURE UpdateSalary (

p\_employee\_id IN Employees.EmployeeID%TYPE,

p\_percentage IN NUMBER

) IS

v\_new\_salary Employees.Salary%TYPE;

e\_employee\_not\_found EXCEPTION;

BEGIN

BEGIN

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

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RAISE e\_employee\_not\_found;

END;

v\_new\_salary := v\_new\_salary + (v\_new\_salary \* p\_percentage / 100);

UPDATE Employees

SET Salary = v\_new\_salary

WHERE EmployeeID = p\_employee\_id;

EXCEPTION

WHEN e\_employee\_not\_found THEN

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

WHEN OTHERS THEN

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

END;

**Scenario 3:** Ensure data integrity when adding a new customer.

* **Question:** Write a stored procedure **AddNewCustomer** that inserts a new customer into the Customers table. If a customer with the same ID already exists, handle the exception by logging an error and preventing the insertion.

CREATE OR REPLACE PROCEDURE AddNewCustomer (

p\_customer\_id IN Customers.CustomerID%TYPE,

p\_name IN Customers.Name%TYPE,

p\_dob IN Customers.DOB%TYPE,

p\_balance IN Customers.Balance%TYPE

) IS

e\_customer\_exists EXCEPTION;

BEGIN

BEGIN

SELECT CustomerID FROM Customers WHERE CustomerID = p\_customer\_id;

RAISE e\_customer\_exists;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

NULL;

END;

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

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

EXCEPTION

WHEN e\_customer\_exists THEN

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

WHEN OTHERS THEN

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

END;

**Exercise 3: Stored Procedures**

**Scenario 1:** The bank needs to process monthly interest for all savings accounts.

* **Question:** Write a stored procedure **ProcessMonthlyInterest** that calculates and updates the balance of all savings accounts by applying an interest rate of 1% to the current balance.

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

CURSOR cur\_savings\_accounts IS

SELECT AccountID, Balance

FROM Accounts

WHERE AccountType = 'Savings';

v\_account\_id Accounts.AccountID%TYPE;

v\_balance Accounts.Balance%TYPE;

v\_interest\_rate CONSTANT NUMBER := 0.01;

BEGIN

OPEN cur\_savings\_accounts;

LOOP

FETCH cur\_savings\_accounts INTO v\_account\_id, v\_balance;

EXIT WHEN cur\_savings\_accounts%NOTFOUND;

UPDATE Accounts

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

WHERE AccountID = v\_account\_id;

END LOOP;

CLOSE cur\_savings\_accounts;

END;

**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 IN Employees.Department%TYPE,

p\_bonus\_percentage IN NUMBER

) IS

BEGIN

UPDATE Employees

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

WHERE Department = p\_department;

END;

**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 Accounts.AccountID%TYPE,

p\_to\_account\_id IN Accounts.AccountID%TYPE,

p\_amount IN NUMBER

) IS

e\_insufficient\_funds EXCEPTION;

v\_balance Accounts.Balance%TYPE;

BEGIN

-- Check the balance of the source account

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

IF v\_balance < p\_amount THEN

RAISE e\_insufficient\_funds;

END IF;

-- Deduct the amount from the source account

UPDATE Accounts

SET Balance = Balance - p\_amount

WHERE AccountID = p\_from\_account\_id;

-- Add the amount to the destination account

UPDATE Accounts

SET Balance = Balance + p\_amount

WHERE AccountID = p\_to\_account\_id;

DBMS\_OUTPUT.PUT\_LINE('Transferred ' || p\_amount || ' from account ' || p\_from\_account\_id || ' to account ' || p\_to\_account\_id);

EXCEPTION

WHEN e\_insufficient\_funds THEN

DBMS\_OUTPUT.PUT\_LINE('Error: Insufficient funds in account ' || p\_from\_account\_id);

WHEN OTHERS THEN

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

END;

**Exercise 4: Functions**

**Scenario 1:** Calculate the age of customers for eligibility checks.

* **Question:** Write a function CalculateAge that takes a customer's date of birth as input and returns their age in years.

CREATE OR REPLACE FUNCTION CalculateAge (

p\_dob IN DATE

) RETURN NUMBER IS

v\_age NUMBER;

BEGIN

v\_age := TRUNC(MONTHS\_BETWEEN(SYSDATE, p\_dob) / 12);

DBMS\_OUTPUT.PUT\_LINE('Age: ' || v\_age);

RETURN v\_age;

END;

**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\_interest\_rate IN NUMBER,

p\_duration\_years IN NUMBER

) RETURN NUMBER IS

v\_monthly\_installment NUMBER;

v\_monthly\_rate NUMBER;

v\_num\_payments NUMBER;

BEGIN

v\_monthly\_rate := p\_interest\_rate / 1200;

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

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

DBMS\_OUTPUT.PUT\_LINE('Monthly Installment: ' || v\_monthly\_installment);

RETURN v\_monthly\_installment;

END;

**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 Accounts.AccountID%TYPE,

p\_amount IN NUMBER

) RETURN BOOLEAN IS

v\_balance Accounts.Balance%TYPE;

BEGIN

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

IF v\_balance >= p\_amount THEN

DBMS\_OUTPUT.PUT\_LINE('Sufficient balance available in account ' || p\_account\_id);

RETURN TRUE;

ELSE

DBMS\_OUTPUT.PUT\_LINE('Insufficient balance in account ' || p\_account\_id);

RETURN FALSE;

END IF;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

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

RETURN FALSE;

END;

**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 OF Name, Balance ON Customers

FOR EACH ROW

BEGIN

:NEW.LastModified := SYSDATE;

DBMS\_OUTPUT.PUT\_LINE('Customer record updated: ID = ' || :NEW.CustomerID || ', LastModified set to: ' || :NEW.LastModified);

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.

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog (TransactionID, Action, ActionDate)

VALUES (:NEW.TransactionID, 'INSERT', SYSDATE);

DBMS\_OUTPUT.PUT\_LINE('Transaction logged: ' || :NEW.TransactionID || ' with Action: 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.

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

BEGIN

IF :NEW.TransactionType = 'Withdrawal' THEN

DECLARE

v\_balance Accounts.Balance%TYPE;

BEGIN

SELECT Balance INTO v\_balance FROM Accounts WHERE AccountID = :NEW.AccountID;

IF v\_balance < :NEW.Amount THEN

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

END IF;

END;

ELSIF :NEW.TransactionType = 'Deposit' THEN

IF :NEW.Amount <= 0 THEN

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

END IF;

END IF;

DBMS\_OUTPUT.PUT\_LINE('Transaction validated: ' || :NEW.TransactionID || ', Type: ' || :NEW.TransactionType);

END;

**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 cur\_transactions IS

SELECT AccountID, TransactionDate, Amount, TransactionType

FROM Transactions

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

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

v\_account\_id Transactions.AccountID%TYPE;

v\_transaction\_date Transactions.TransactionDate%TYPE;

v\_amount Transactions.Amount%TYPE;

v\_transaction\_type Transactions.TransactionType%TYPE;

BEGIN

OPEN cur\_transactions;

LOOP

FETCH cur\_transactions INTO v\_account\_id, v\_transaction\_date, v\_amount,

V\_transaction\_type;

EXIT WHEN cur\_transactions%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Account ID: ' || v\_account\_id || ', Date: ' || v\_transaction\_date || ', Amount: ' || v\_amount || ', Type: ' || v\_transaction\_type);

END LOOP;

CLOSE cur\_transactions;

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 cur\_accounts IS

SELECT AccountID, Balance

FROM Accounts;

v\_account\_id Accounts.AccountID%TYPE;

v\_balance Accounts.Balance%TYPE;

BEGIN

OPEN cur\_accounts;

LOOP

FETCH cur\_accounts INTO v\_account\_id, v\_balance;

EXIT WHEN cur\_accounts%NOTFOUND;

UPDATE Accounts

SET Balance = Balance - 50 -- Example fee amount

WHERE AccountID = v\_account\_id;

DBMS\_OUTPUT.PUT\_LINE('Annual fee applied to Account ID: ' || v\_account\_id || ', New Balance: ' || (v\_balance - 50));

END LOOP;

CLOSE cur\_accounts;

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 cur\_loans IS

SELECT LoanID, InterestRate

FROM Loans;

v\_loan\_id Loans.LoanID%TYPE;

v\_interest\_rate Loans.InterestRate%TYPE;

BEGIN

OPEN cur\_loans;

LOOP

FETCH cur\_loans INTO v\_loan\_id, v\_interest\_rate;

EXIT WHEN cur\_loans%NOTFOUND;

UPDATE Loans

SET InterestRate = InterestRate + 0.5 -- Example rate adjustment

WHERE LoanID = v\_loan\_id;

DBMS\_OUTPUT.PUT\_LINE('Interest rate updated for Loan ID: ' || v\_loan\_id || ', New Rate: ' || (v\_interest\_rate + 0.5));

END LOOP;

CLOSE cur\_loans;

END;

**Exercise 7: Packages**

**Scenario 1:** Group all customer-related procedures and functions into a package.

* **Question:** Create a package **CustomerManagement** with procedures for adding a new customer, updating customer details, and a function to get customer balance.

CREATE OR REPLACE PACKAGE CustomerManagement AS

PROCEDURE AddNewCustomer (

p\_customer\_id IN Customers.CustomerID%TYPE,

p\_name IN Customers.Name%TYPE,

p\_dob IN Customers.DOB%TYPE,

p\_balance IN Customers.Balance%TYPE

);

PROCEDURE UpdateCustomer (

p\_customer\_id IN Customers.CustomerID%TYPE,

p\_name IN Customers.Name%TYPE,

p\_balance IN Customers.Balance%TYPE

);

FUNCTION GetCustomerBalance (

p\_customer\_id IN Customers.CustomerID%TYPE

) RETURN NUMBER;

END CustomerManagement;

/

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

PROCEDURE AddNewCustomer (

p\_customer\_id IN Customers.CustomerID%TYPE,

p\_name IN Customers.Name%TYPE,

p\_dob IN Customers.DOB%TYPE,

p\_balance IN Customers.Balance%TYPE

) IS

BEGIN

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

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

DBMS\_OUTPUT.PUT\_LINE('New customer added: ' || p\_customer\_id || ', Name: ' || p\_name);

END;

PROCEDURE UpdateCustomer (

p\_customer\_id IN Customers.CustomerID%TYPE,

p\_name IN Customers.Name%TYPE,

p\_balance IN Customers.Balance%TYPE

) IS

BEGIN

UPDATE Customers

SET Name = p\_name, Balance = p\_balance, LastModified = SYSDATE

WHERE CustomerID = p\_customer\_id;

DBMS\_OUTPUT.PUT\_LINE('Customer updated: ' || p\_customer\_id || ', New Name: ' || p\_name);

END;

FUNCTION GetCustomerBalance (

p\_customer\_id IN Customers.CustomerID%TYPE

) RETURN NUMBER IS

v\_balance Customers.Balance%TYPE;

BEGIN

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

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

RETURN v\_balance;

END;

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.

CREATE OR REPLACE PACKAGE EmployeeManagement AS

PROCEDURE HireNewEmployee (

p\_employee\_id IN Employees.EmployeeID%TYPE,

p\_name IN Employees.Name%TYPE,

p\_position IN Employees.Position%TYPE,

p\_salary IN Employees.Salary%TYPE,

p\_department IN Employees.Department%TYPE,

p\_hire\_date IN Employees.HireDate%TYPE

);

PROCEDURE UpdateEmployee (

p\_employee\_id IN Employees.EmployeeID%TYPE,

p\_name IN Employees.Name%TYPE,

p\_position IN Employees.Position%TYPE,

p\_salary IN Employees.Salary%TYPE

);

FUNCTION CalculateAnnualSalary (

p\_employee\_id IN Employees.EmployeeID%TYPE

) RETURN NUMBER;

END EmployeeManagement;

/

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

PROCEDURE HireNewEmployee (

p\_employee\_id IN Employees.EmployeeID%TYPE,

p\_name IN Employees.Name%TYPE,

p\_position IN Employees.Position%TYPE,

p\_salary IN Employees.Salary%TYPE,

p\_department IN Employees.Department%TYPE,

p\_hire\_date IN Employees.HireDate%TYPE

) 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);

DBMS\_OUTPUT.PUT\_LINE('New employee hired: ' || p\_employee\_id || ', Name: ' || p\_name);

END;

PROCEDURE UpdateEmployee (

p\_employee\_id IN Employees.EmployeeID%TYPE,

p\_name IN Employees.Name%TYPE,

p\_position IN Employees.Position%TYPE,

p\_salary IN Employees.Salary%TYPE

) IS

BEGIN

UPDATE Employees

SET Name = p\_name, Position = p\_position, Salary = p\_salary

WHERE EmployeeID = p\_employee\_id;

DBMS\_OUTPUT.PUT\_LINE('Employee updated: ' || p\_employee\_id || ', New Position: ' || p\_position);

END;

FUNCTION CalculateAnnualSalary (

p\_employee\_id IN Employees.EmployeeID%TYPE

) RETURN NUMBER IS

v\_salary Employees.Salary%TYPE;

BEGIN

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

DBMS\_OUTPUT.PUT\_LINE('Annual salary for Employee ID ' || p\_employee\_id || ': ' || (v\_salary \* 12));

RETURN v\_salary \* 12;

END;

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.

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

PROCEDURE OpenNewAccount (

p\_account\_id IN Accounts.AccountID%TYPE,

p\_customer\_id IN Accounts.CustomerID%TYPE,

p\_account\_type IN Accounts.AccountType%TYPE,

p\_balance IN Accounts.Balance%TYPE

) IS

BEGIN

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

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

DBMS\_OUTPUT.PUT\_LINE('New account opened: ID = ' || p\_account\_id || ', Type = ' || p\_account\_type);

END;

PROCEDURE CloseAccount (

p\_account\_id IN Accounts.AccountID%TYPE

) IS

BEGIN

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

DBMS\_OUTPUT.PUT\_LINE('Account closed: ID = ' || p\_account\_id);

END;

FUNCTION GetTotalBalance (

p\_customer\_id IN Accounts.CustomerID%TYPE

) RETURN NUMBER IS

v\_total\_balance NUMBER := 0;

BEGIN

SELECT SUM(Balance) INTO v\_total\_balance

FROM Accounts

WHERE CustomerID = p\_customer\_id;

DBMS\_OUTPUT.PUT\_LINE('Total balance for Customer ID ' || p\_customer\_id || ': ' || v\_total\_balance);

RETURN v\_total\_balance;

END;

END AccountOperations;