**PLSQL:**

**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 customer\_cursor IS

SELECT customer\_id, age, loan\_interest\_rate

FROM customers;

v\_customer\_id NUMBER;

v\_age NUMBER;

v\_loan\_interest\_rate NUMBER;

v\_discount\_rate CONSTANT NUMBER := 0.01;

BEGIN

OPEN customer\_cursor;

LOOP

FETCH customer\_cursor INTO v\_customer\_id, v\_age, v\_loan\_interest\_rate;

EXIT WHEN customer\_cursor%NOTFOUND;

IF v\_age > 60 THEN

UPDATE loans

SET interest\_rate = interest\_rate - (interest\_rate \* v\_discount\_rate)

WHERE customer\_id = v\_customer\_id;

END IF;

END LOOP;

CLOSE customer\_cursor;

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 customer\_cursor IS

SELECT customer\_id, balance

FROM customers;

v\_customer\_id NUMBER;

v\_balance NUMBER;

BEGIN

OPEN customer\_cursor;

LOOP

FETCH customer\_cursor INTO v\_customer\_id, v\_balance;

EXIT WHEN customer\_cursor%NOTFOUND;

IF v\_balance > 10000 THEN

UPDATE customers

SET is\_vip = 'TRUE'

WHERE customer\_id = v\_customer\_id;

END IF;

END LOOP;

CLOSE customer\_cursor;

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 loan\_cursor IS

SELECT l.loan\_id, c.customer\_name, l.due\_date

FROM loans l

JOIN customers c ON l.customer\_id = c.customer\_id

WHERE l.due\_date BETWEEN SYSDATE AND SYSDATE + 30;

v\_loan\_id NUMBER;

v\_customer\_name VARCHAR2(100);

v\_due\_date DATE;

BEGIN

OPEN loan\_cursor;

LOOP

FETCH loan\_cursor INTO v\_loan\_id, v\_customer\_name, v\_due\_date;

EXIT WHEN loan\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Reminder for customer ' || v\_customer\_name || ': Loan ' || v\_loan\_id || ' is due on ' || v\_due\_date);

END LOOP;

CLOSE loan\_cursor;

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\_number NUMBER,

p\_to\_account\_number NUMBER,

p\_amount NUMBER

)

IS

v\_from\_account\_balance NUMBER;

v\_to\_account\_balance NUMBER;

BEGIN

-- Check for insufficient funds

SELECT balance INTO v\_from\_account\_balance FROM accounts WHERE account\_number = p\_from\_account\_number;

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

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

END IF;

-- Begin transaction

BEGIN

UPDATE accounts SET balance = balance - p\_amount WHERE account\_number = p\_from\_account\_number;

UPDATE accounts SET balance = balance + p\_amount WHERE account\_number = p\_to\_account\_number;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

-- Log error (e.g., insert into an error log table)

INSERT INTO error\_log (error\_code, error\_message, transaction\_time)

VALUES (SQLCODE, SQLERRM, SYSDATE);

END;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 NUMBER,

p\_percentage NUMBER

)

IS

v\_current\_salary NUMBER;

BEGIN

-- Get current salary

SELECT salary INTO v\_current\_salary FROM employees WHERE employee\_id = p\_employee\_id;

-- Update salary

UPDATE employees SET salary = salary \* (1 + p\_percentage / 100) WHERE employee\_id = p\_employee\_id;

COMMIT;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

-- Handle employee not found

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

-- Log error (e.g., insert into an error log table)

WHEN OTHERS THEN

-- Handle other exceptions

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error updating salary for employee ' || p\_employee\_id);

-- Log error (e.g., insert into an error log table)

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 NUMBER,

p\_customer\_name VARCHAR2(100),

p\_address VARCHAR2(200)

)

IS

BEGIN

INSERT INTO customers (customer\_id, customer\_name, address)

VALUES (p\_customer\_id, p\_customer\_name, p\_address);

COMMIT;

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

-- Handle duplicate customer ID

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

-- Log error (e.g., insert into an error log table)

ROLLBACK;

WHEN OTHERS THEN

-- Handle other exceptions

DBMS\_OUTPUT.PUT\_LINE('Error adding new customer.');

-- Log error (e.g., insert into an error log table)

ROLLBACK;

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

BEGIN

UPDATE accounts

SET balance = balance \* 1.01

WHERE account\_type = 'SAVINGS';

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

-- Log error (e.g., insert into an error log table)

END;

/

**CREATE OR REPLACE PROCEDURE:** Creates or replaces the stored procedure ProcessMonthlyInterest.

**BEGIN:** Start the procedure body.

**UPDATE:** Updates the balance of all savings accounts by multiplying the current balance by 1.01 (which is equivalent to adding 1% interest).

**COMMIT:** Commit the transaction if successful.

**EXCEPTION:** Handles potential errors.

**WHEN OTHERS:** Catches any other exceptions.

**ROLLBACK:** Rolls back the transaction in case of an error.

**END;:** Ends the procedure.

**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 NUMBER,

p\_bonus\_percentage 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;

-- Log error (e.g., insert into an error log table)

END;

/

**CREATE OR REPLACE PROCEDURE:** Creates or replaces the stored procedure UpdateEmployeeBonus.

**DECLARE:** No explicit declaration of variables is needed in this case.

**BEGIN:** Starts the procedure body.

**UPDATE:** Updates the salary of employees in the specified department by applying the bonus percentage.

**COMMIT:** Commits the transaction if successful.

**EXCEPTION:** Handles potential errors.

**WHEN OTHERS:** Catches any other exceptions.

**ROLLBACK:** Rolls back the transaction in case of an error.

**END;:** Ends the procedure.

**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\_number NUMBER,

p\_to\_account\_number NUMBER,

p\_amount NUMBER

)

IS

v\_from\_account\_balance NUMBER;

BEGIN

-- Check for sufficient funds

SELECT balance INTO v\_from\_account\_balance FROM accounts WHERE account\_number = p\_from\_account\_number;

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

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

END IF;

-- Begin transaction

BEGIN

UPDATE accounts SET balance = balance - p\_amount WHERE account\_number = p\_from\_account\_number;

UPDATE accounts SET balance = balance + p\_amount WHERE account\_number = p\_to\_account\_number;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

-- Log error (e.g., insert into an error log table)

INSERT INTO error\_log (error\_code, error\_message, transaction\_time)

VALUES (SQLCODE, SQLERRM, SYSDATE);

END;

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\_birthdate DATE)

RETURN NUMBER

IS

v\_age NUMBER;

BEGIN

v\_age := TRUNC(SYSDATE - p\_birthdate) / 365.25;

RETURN v\_age;

EXCEPTION

WHEN OTHERS THEN

RETURN -1; -- Indicate error

END;

/

**CREATE OR REPLACE FUNCTION:** Creates or replaces the function CalculateAge.

**RETURN NUMBER:** Specifies that the function returns a number (age in years).

**IS:** Starts the function body.

**DECLARE:** Declares a variable to store the calculated age.

**BEGIN:** Starts the function logic.

**v\_age :=:** Calculates the age by subtracting the birthdate from the current date and dividing by the approximate number of days in a year (365.25).

**RETURN v\_age:** Returns the calculated age.

**EXCEPTION:** Handles potential errors (e.g., invalid birthdate).

**RETURN -1:** Returns -1 to indicate an error.

**END;:** Ends the function.

**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 NUMBER,

p\_interest\_rate NUMBER,

p\_loan\_duration\_years NUMBER

)

RETURN NUMBER

IS

v\_monthly\_interest\_rate NUMBER;

v\_total\_months NUMBER;

v\_emi NUMBER;

BEGIN

-- Convert annual interest rate to monthly interest rate

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

-- Convert loan duration from years to months

v\_total\_months := p\_loan\_duration\_years \* 12;

-- Calculate EMI using the formula

v\_emi := (p\_loan\_amount \* v\_monthly\_interest\_rate \* POWER(1 + v\_monthly\_interest\_rate, v\_total\_months)) /

(POWER(1 + v\_monthly\_interest\_rate, v\_total\_months) - 1);

RETURN v\_emi;

EXCEPTION

WHEN OTHERS THEN

RETURN -1; -- Indicate error

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 NUMBER,

p\_amount NUMBER

)

RETURN BOOLEAN

IS

v\_balance NUMBER;

BEGIN

SELECT balance INTO v\_balance FROM accounts WHERE account\_id = p\_account\_id;

RETURN v\_balance >= p\_amount;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN FALSE; -- Account not found

WHEN OTHERS THEN

RETURN FALSE; -- Error occurred

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

AFTER UPDATE ON Customers

FOR EACH ROW

BEGIN

UPDATE Customers

SET LastModified = SYSDATE

WHERE CustomerID = :NEW.CustomerID;

END;

/

**CREATE OR REPLACE TRIGGER:** Creates or replaces the trigger UpdateCustomerLastModified.

**AFTER UPDATE ON Customers:** Specifies that the trigger is fired after an update on the Customers table.

**FOR EACH ROW:** Indicates that the trigger will be fired for each row updated.

**BEGIN:** Starts the trigger body.

**UPDATE Customers:** Updates the LastModified column for the updated customer.

**WHERE CustomerID = :NEW.CustomerID:** Ensures that only the updated row is affected.

**END:** Ends the trigger body.

**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, TransactionDate, Amount, AccountFrom, AccountTo)

VALUES (:NEW.TransactionID, :NEW.TransactionDate, :NEW.Amount, :NEW.AccountFrom, :NEW.AccountTo);

END;

/

**CREATE OR REPLACE TRIGGER:** Creates or replaces the trigger LogTransaction.

**AFTER INSERT ON Transactions:** Specifies that the trigger is fired after an insert on the Transactions table.

**FOR EACH ROW:** Indicates that the trigger will be fired for each row inserted.

**BEGIN:** Starts the trigger body.

**INSERT INTO AuditLog:** Inserts a new record into the AuditLog table with the relevant transaction details.

**VALUES (:NEW.TransactionID, ...):** Populates the AuditLog record with values from the inserted transaction row.

**END:** Ends the trigger body.

**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

IF :NEW.Amount > (SELECT balance FROM accounts WHERE account\_id = :NEW.AccountFrom) THEN

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

END IF;

ELSIF :NEW.TransactionType = 'DEPOSIT' THEN

IF :NEW.Amount <= 0 THEN

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

END IF;

END IF;

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 customer\_cursor IS

SELECT DISTINCT customer\_id

FROM transactions

WHERE transaction\_date >= TRUNC(SYSDATE, 'MM');

v\_customer\_id NUMBER;

v\_transaction\_cursor REF CURSOR;

v\_transaction\_id NUMBER;

v\_transaction\_date DATE;

v\_amount NUMBER;

v\_account\_number NUMBER;

BEGIN

OPEN customer\_cursor;

LOOP

FETCH customer\_cursor INTO v\_customer\_id;

EXIT WHEN customer\_cursor%NOTFOUND;

OPEN v\_transaction\_cursor FOR

SELECT transaction\_id, transaction\_date, amount, account\_number

FROM transactions

WHERE customer\_id = v\_customer\_id

AND transaction\_date >= TRUNC(SYSDATE, 'MM');

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

LOOP

FETCH v\_transaction\_cursor INTO v\_transaction\_id, v\_transaction\_date, v\_amount, v\_account\_number;

EXIT WHEN v\_transaction\_cursor%NOTFOUND;

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

END LOOP;

CLOSE v\_transaction\_cursor;

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

END LOOP;

CLOSE customer\_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 account\_id, balance

FROM accounts;

v\_account\_id NUMBER;

v\_balance NUMBER;

v\_annual\_fee CONSTANT NUMBER := 100; -- Replace with actual fee amount

BEGIN

OPEN account\_cursor;

LOOP

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

EXIT WHEN account\_cursor%NOTFOUND;

IF v\_balance >= v\_annual\_fee THEN

UPDATE accounts

SET balance = balance - v\_annual\_fee

WHERE account\_id = v\_account\_id;

ELSE

-- Handle insufficient balance (e.g., log an error, send a notification)

END IF;

END LOOP;

CLOSE account\_cursor;

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 loan\_id, loan\_amount, interest\_rate

FROM loans;

v\_loan\_id NUMBER;

v\_loan\_amount NUMBER;

v\_old\_interest\_rate NUMBER;

v\_new\_interest\_rate NUMBER := 0.05; -- Replace with new interest rate

BEGIN

OPEN loan\_cursor;

LOOP

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

EXIT WHEN loan\_cursor%NOTFOUND;

UPDATE loans

SET interest\_rate = v\_new\_interest\_rate

WHERE loan\_id = v\_loan\_id;

END LOOP;

CLOSE loan\_cursor;

END;

/

**Exercise 7: Packages**

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

p\_customer\_id NUMBER,

p\_customer\_name VARCHAR2(100),

p\_address VARCHAR2(200)

);

PROCEDURE UpdateCustomer(

p\_customer\_id NUMBER,

p\_customer\_name VARCHAR2(100),

p\_address VARCHAR2(200)

);

FUNCTION GetCustomerBalance(

p\_customer\_id NUMBER

) RETURN NUMBER;

END CustomerManagement;

/

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

PROCEDURE AddCustomer(

p\_customer\_id NUMBER,

p\_customer\_name VARCHAR2(100),

p\_address VARCHAR2(200)

)

IS

BEGIN

-- Insert customer into Customers table

END AddCustomer;

PROCEDURE UpdateCustomer(

p\_customer\_id NUMBER,

p\_customer\_name VARCHAR2(100),

p\_address VARCHAR2(200)

)

IS

BEGIN

-- Update customer details in Customers table

END UpdateCustomer;

FUNCTION GetCustomerBalance(

p\_customer\_id NUMBER

) RETURN NUMBER

IS

v\_balance NUMBER;

BEGIN

SELECT balance INTO v\_balance FROM accounts WHERE customer\_id = p\_customer\_id;

RETURN v\_balance;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN -1; -- Indicate error

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 HireEmployee(

p\_employee\_id NUMBER,

p\_first\_name VARCHAR2(50),

p\_last\_name VARCHAR2(50),

p\_department\_id NUMBER,

p\_salary NUMBER

);

PROCEDURE UpdateEmployee(

p\_employee\_id NUMBER,

p\_first\_name VARCHAR2(50),

p\_last\_name VARCHAR2(50),

p\_department\_id NUMBER,

p\_salary NUMBER

);

FUNCTION CalculateAnnualSalary(

p\_employee\_id NUMBER

) RETURN NUMBER;

END EmployeeManagement;

/

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

PROCEDURE HireEmployee(

p\_employee\_id NUMBER,

p\_first\_name VARCHAR2(50),

p\_last\_name VARCHAR2(50),

p\_department\_id NUMBER,

p\_salary NUMBER

)

IS

BEGIN

-- Insert employee into Employees table

END HireEmployee;

PROCEDURE UpdateEmployee(

p\_employee\_id NUMBER,

p\_first\_name VARCHAR2(50),

p\_last\_name VARCHAR2(50),

p\_department\_id NUMBER,

p\_salary NUMBER

)

IS

BEGIN

-- Update employee details in Employees table

END UpdateEmployee;

FUNCTION CalculateAnnualSalary(

p\_employee\_id NUMBER

) RETURN NUMBER

IS

v\_salary NUMBER;

BEGIN

SELECT salary INTO v\_salary FROM employees WHERE employee\_id = p\_employee\_id;

RETURN v\_salary \* 12;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN -1; -- Indicate error

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 AccountOperations AS

PROCEDURE OpenAccount(

p\_customer\_id NUMBER,

p\_account\_type VARCHAR2(50),

p\_initial\_balance NUMBER

);

PROCEDURE CloseAccount(

p\_account\_number NUMBER

);

FUNCTION GetTotalCustomerBalance(

p\_customer\_id NUMBER

) RETURN NUMBER;

END AccountOperations;

/

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

PROCEDURE OpenAccount(

p\_customer\_id NUMBER,

p\_account\_type VARCHAR2(50),

p\_initial\_balance NUMBER

)

IS

BEGIN

-- Insert new account into Accounts table

END OpenAccount;

PROCEDURE CloseAccount(

p\_account\_number NUMBER

)

IS

BEGIN

-- Update account status to closed in Accounts table

END CloseAccount;

FUNCTION GetTotalCustomerBalance(

p\_customer\_id NUMBER

) RETURN NUMBER

IS

v\_total\_balance NUMBER := 0;

BEGIN

SELECT SUM(balance) INTO v\_total\_balance

FROM accounts

WHERE customer\_id = p\_customer\_id;

RETURN v\_total\_balance;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN 0; -- Customer has no accounts

END;

END AccountOperations;

/