**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.

**Answer:**

BEGIN

FOR customer\_rec IN (

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

FROM customers

JOIN loans ON customers.customer\_id = loans.customer\_id

) LOOP

IF customer\_rec.age > 60 THEN

UPDATE loans

SET current\_interest\_rate = current\_interest\_rate - 0.01

WHERE loan\_id = customer\_rec.loan\_id;

DBMS\_OUTPUT.PUT\_LINE('Applied 1% discount for loan ID: ' || customer\_rec.loan\_id || ' for customer ID: ' || customer\_rec.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.

**Answer:**

BEGIN

FOR customer\_rec IN (

SELECT customer\_id, balance

FROM customers

) LOOP

IF customer\_rec.balance > 10000 THEN

UPDATE customers

SET is\_vip = TRUE

WHERE customer\_id = customer\_rec.customer\_id;

DBMS\_OUTPUT.PUT\_LINE('Customer ID: ' || customer\_rec.customer\_id || ' promoted to VIP status.');

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.

**Answer:**

BEGIN

FOR loan\_rec IN (

SELECT loan\_id, customer\_id, due\_date

FROM loans

WHERE due\_date BETWEEN SYSDATE AND SYSDATE + 30

) LOOP

DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan ID ' || loan\_rec.loan\_id || ' for customer ID ' || loan\_rec.customer\_id || ' is due on ' || loan\_rec.due\_date);

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.

**Answer:**

CREATE OR REPLACE PROCEDURE SafeTransferFunds (

p\_from\_account\_id IN NUMBER,

p\_to\_account\_id IN NUMBER,

p\_amount IN NUMBER

) AS

v\_from\_balance NUMBER;

v\_to\_balance NUMBER;

BEGIN

SAVEPOINT before\_transfer;

SELECT balance INTO v\_from\_balance

FROM accounts

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

SELECT balance INTO v\_to\_balance

FROM accounts

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

IF v\_from\_balance < p\_amount THEN

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

END IF;

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;

DBMS\_OUTPUT.PUT\_LINE('Funds transferred successfully from account ' || p\_from\_account\_id || ' to account ' || p\_to\_account\_id);

EXCEPTION

WHEN OTHERS THEN

ROLLBACK TO before\_transfer;

**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.

**Answer:**

CREATE OR REPLACE PROCEDURE UpdateSalary (

p\_employee\_id IN NUMBER,

p\_percentage IN NUMBER

) AS

v\_current\_salary NUMBER;

BEGIN

SELECT salary INTO v\_current\_salary

FROM employees

WHERE employee\_id = p\_employee\_id;

UPDATE employees

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

WHERE employee\_id = p\_employee\_id;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Salary updated successfully for employee ID: ' || p\_employee\_id);

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

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

WHEN OTHERS THEN

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

END UpdateSalary;

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

END SafeTransferFunds;

**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.

**Answer:**

CREATE OR REPLACE PROCEDURE AddNewCustomer (

p\_customer\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_balance IN NUMBER

) AS

BEGIN

BEGIN

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

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

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Customer added successfully with ID: ' || p\_customer\_id);

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

ROLLBACK;

WHEN OTHERS THEN

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

ROLLBACK;

END;

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.

**Answer:**

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest AS

v\_account\_id NUMBER;

v\_current\_balance NUMBER;

BEGIN

FOR account\_rec IN (

SELECT account\_id, balance

FROM accounts

WHERE account\_type = 'Savings'

) LOOP

v\_account\_id := account\_rec.account\_id;

v\_current\_balance := account\_rec.balance;

UPDATE accounts

SET balance = v\_current\_balance \* 1.01

WHERE account\_id = v\_account\_id;

DBMS\_OUTPUT.PUT\_LINE('Updated balance for savings account ID: ' || v\_account\_id);

END LOOP;

COMMIT;

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

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error processing monthly interest: ' || 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.

**Answer:**

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (

p\_department\_id IN NUMBER,

p\_bonus\_percentage IN NUMBER

) AS

BEGIN

UPDATE employees

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

WHERE department\_id = p\_department\_id;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Salaries updated with a bonus of ' || p\_bonus\_percentage || '% for department ID: ' || p\_department\_id);

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error updating employee bonus: ' || 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.

**Answer:**

CREATE OR REPLACE PROCEDURE TransferFunds (

p\_from\_account\_id IN NUMBER,

p\_to\_account\_id IN NUMBER,

p\_amount IN NUMBER

) AS

v\_from\_balance NUMBER;

BEGIN

SELECT balance INTO v\_from\_balance

FROM accounts

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

IF v\_from\_balance < p\_amount THEN

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

END IF;

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;

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

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

**Answer:**

CREATE OR REPLACE FUNCTION CalculateAge (

p\_date\_of\_birth DATE

) RETURN NUMBER IS

v\_current\_date DATE := SYSDATE;

v\_age NUMBER;

BEGIN

v\_age := FLOOR(MONTHS\_BETWEEN(v\_current\_date, p\_date\_of\_birth) / 12);

RETURN v\_age;

EXCEPTION

WHEN OTHERS THEN

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

**Answer:**

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment (

p\_loan\_amount NUMBER,

p\_annual\_interest\_rate NUMBER,

p\_loan\_duration\_years NUMBER

) RETURN NUMBER IS

v\_monthly\_interest\_rate NUMBER;

v\_total\_months NUMBER;

v\_monthly\_installment NUMBER;

BEGIN

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

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

IF v\_monthly\_interest\_rate = 0 THEN

v\_monthly\_installment := p\_loan\_amount / v\_total\_months;

ELSE

v\_monthly\_installment := 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);

END IF;

RETURN v\_monthly\_installment;

EXCEPTION

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('Error calculating monthly installment: ' || 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.

**Answer:**

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;

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 || ' does not exist.');

RETURN FALSE;

WHEN OTHERS THEN

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

**Answer:**

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.

**Answer:**

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog (TransactionID, TransactionDate, Amount, Action)

VALUES (:NEW.TransactionID, :NEW.TransactionDate, :NEW.Amount, 'INSERT');

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.

**Answer:**

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

DECLARE

v\_balance NUMBER;

BEGIN

IF :NEW.TransactionType = 'WITHDRAWAL' THEN

SELECT balance INTO v\_balance

FROM accounts

WHERE account\_id = :NEW.AccountID;

IF :NEW.Amount > v\_balance THEN

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

END IF;

ELSIF :NEW.TransactionType = 'DEPOSIT' THEN

IF :NEW.Amount <= 0 THEN

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

END IF;

ELSE

RAISE\_APPLICATION\_ERROR(-20003, 'Invalid transaction type.');

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.

**Answer:**

DECLARE

CURSOR transaction\_cursor IS

SELECT customer\_id, transaction\_date, amount, transaction\_type

FROM transactions

WHERE EXTRACT(MONTH FROM transaction\_date) = EXTRACT(MONTH FROM SYSDATE)

AND EXTRACT(YEAR FROM transaction\_date) = EXTRACT(YEAR FROM SYSDATE);

v\_customer\_id transactions.customer\_id%TYPE;

v\_transaction\_date transactions.transaction\_date%TYPE;

v\_amount transactions.amount%TYPE;

v\_transaction\_type transactions.transaction\_type%TYPE;

BEGIN

OPEN transaction\_cursor;

LOOP

FETCH transaction\_cursor INTO v\_customer\_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('Date: ' || v\_transaction\_date);

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

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

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

END LOOP;

CLOSE transaction\_cursor;

DBMS\_OUTPUT.PUT\_LINE('Monthly statements generated for all customers.');

EXCEPTION

WHEN OTHERS THEN

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

END GenerateMonthlyStatements;

**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.

**Answer:**

DECLARE

CURSOR account\_cursor IS

SELECT account\_id, balance

FROM accounts;

v\_account\_id accounts.account\_id%TYPE;

v\_balance accounts.balance%TYPE;

v\_annual\_fee NUMBER := 50; -- Example fee amount

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

WHERE account\_id = v\_account\_id;

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

END LOOP;

CLOSE account\_cursor;

DBMS\_OUTPUT.PUT\_LINE('Annual fee applied to all accounts.');

EXCEPTION

WHEN OTHERS THEN

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

END ApplyAnnualFee;

**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.

**Answer:**

DECLARE

CURSOR loan\_cursor IS

SELECT loan\_id, current\_interest\_rate

FROM loans;

v\_loan\_id loans.loan\_id%TYPE;

v\_current\_interest\_rate loans.current\_interest\_rate%TYPE;

v\_new\_interest\_rate NUMBER;

BEGIN

OPEN loan\_cursor;

LOOP

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

EXIT WHEN loan\_cursor%NOTFOUND;

v\_new\_interest\_rate := v\_current\_interest\_rate \* 1.05; -- Example policy: increase by 5%

UPDATE loans

SET current\_interest\_rate = v\_new\_interest\_rate

WHERE loan\_id = v\_loan\_id;

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

END LOOP;

CLOSE loan\_cursor;

DBMS\_OUTPUT.PUT\_LINE('Interest rates updated for all loans.');

EXCEPTION

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('Error updating loan interest rates: ' || SQLERRM);

END UpdateLoanInterestRates;

**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.

**Answer:**

CREATE OR REPLACE PACKAGE CustomerManagement AS

PROCEDURE AddCustomer(p\_customer\_id IN NUMBER, p\_name IN VARCHAR2, p\_balance IN NUMBER);

PROCEDURE UpdateCustomer(p\_customer\_id IN NUMBER, p\_name IN VARCHAR2, p\_balance IN NUMBER);

FUNCTION GetCustomerBalance(p\_customer\_id IN NUMBER) RETURN NUMBER;

END CustomerManagement;

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

PROCEDURE AddCustomer(p\_customer\_id IN NUMBER, p\_name IN VARCHAR2, p\_balance IN NUMBER) IS

BEGIN

INSERT INTO Customers (customer\_id, name, balance)

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

DBMS\_OUTPUT.PUT\_LINE('Customer added: ' || p\_name);

EXCEPTION

WHEN OTHERS THEN

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

END AddCustomer;

PROCEDURE UpdateCustomer(p\_customer\_id IN NUMBER, p\_name IN VARCHAR2, p\_balance IN NUMBER) IS

BEGIN

UPDATE Customers

SET name = p\_name, balance = p\_balance

WHERE customer\_id = p\_customer\_id;

DBMS\_OUTPUT.PUT\_LINE('Customer updated: ' || p\_name);

EXCEPTION

WHEN OTHERS THEN

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

END UpdateCustomer;

FUNCTION GetCustomerBalance(p\_customer\_id IN NUMBER) RETURN NUMBER IS

v\_balance NUMBER;

BEGIN

SELECT balance INTO v\_balance

FROM Customers

WHERE customer\_id = p\_customer\_id;

RETURN v\_balance;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN 0;

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('Error fetching customer balance: ' || 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.

**Answer:**

CREATE OR REPLACE PACKAGE EmployeeManagement AS

PROCEDURE HireEmployee(p\_employee\_id IN NUMBER, p\_name IN VARCHAR2, p\_salary IN NUMBER);

PROCEDURE UpdateEmployee(p\_employee\_id IN NUMBER, p\_name IN VARCHAR2, p\_salary IN NUMBER);

FUNCTION CalculateAnnualSalary(p\_salary IN NUMBER) RETURN NUMBER;

END EmployeeManagement;

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

PROCEDURE HireEmployee(p\_employee\_id IN NUMBER, p\_name IN VARCHAR2, p\_salary IN NUMBER) IS

BEGIN

INSERT INTO Employees (employee\_id, name, salary)

VALUES (p\_employee\_id, p\_name, p\_salary);

DBMS\_OUTPUT.PUT\_LINE('Employee hired: ' || p\_name);

EXCEPTION

WHEN OTHERS THEN

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

END HireEmployee;

PROCEDURE UpdateEmployee(p\_employee\_id IN NUMBER, p\_name IN VARCHAR2, p\_salary IN NUMBER) IS

BEGIN

UPDATE Employees

SET name = p\_name, salary = p\_salary

WHERE employee\_id = p\_employee\_id;

DBMS\_OUTPUT.PUT\_LINE('Employee updated: ' || p\_name);

EXCEPTION

WHEN OTHERS THEN

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

END UpdateEmployee;

FUNCTION CalculateAnnualSalary(p\_salary IN NUMBER) RETURN NUMBER IS

BEGIN

RETURN p\_salary \* 12;

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.

**Answer:**

CREATE OR REPLACE PACKAGE AccountOperations AS

PROCEDURE OpenAccount(p\_account\_id IN NUMBER, p\_customer\_id IN NUMBER, p\_initial\_balance IN NUMBER);

PROCEDURE CloseAccount(p\_account\_id IN NUMBER);

FUNCTION GetTotalBalance(p\_customer\_id IN NUMBER) RETURN NUMBER;

END AccountOperations;

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

PROCEDURE OpenAccount(p\_account\_id IN NUMBER, p\_customer\_id IN NUMBER, p\_initial\_balance IN NUMBER) IS

BEGIN

INSERT INTO Accounts (account\_id, customer\_id, balance)

VALUES (p\_account\_id, p\_customer\_id, p\_initial\_balance);

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

EXCEPTION

WHEN OTHERS THEN

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

END OpenAccount;

PROCEDURE CloseAccount(p\_account\_id IN NUMBER) IS

BEGIN

DELETE FROM Accounts

WHERE account\_id = p\_account\_id;

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

EXCEPTION

WHEN OTHERS THEN

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

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 customer\_id = p\_customer\_id;

RETURN NVL(v\_total\_balance, 0);

EXCEPTION

WHEN OTHERS THEN

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

RETURN NULL;

END GetTotalBalance;

END AccountOperations;