**PLSQL Assignment**

**Exercise 1: Control Structures**  
  
**Scenario 1:**  
  
DECLARE  
  -- Cursor to loop through customers  
  CURSOR customer\_cur IS  
    SELECT customer\_id, age, loan\_interest\_rate  
    FROM customers  
    WHERE loan\_interest\_rate IS NOT NULL;  
  
  -- Variable to hold the current customer's data  
  customer\_rec customer\_cur%ROWTYPE;  
BEGIN  
  -- Open the cursor  
  OPEN customer\_cur;  
  
  -- Loop through each customer  
  LOOP  
    FETCH customer\_cur INTO customer\_rec;  
    EXIT WHEN customer\_cur%NOTFOUND;  
  
    -- Check if the customer is above 60  
    IF customer\_rec.age > 60 THEN  
      -- Apply a 1% discount to their current loan interest rate  
      UPDATE customers  
      SET loan\_interest\_rate = loan\_interest\_rate \* 0.99  
      WHERE customer\_id = customer\_rec.customer\_id;  
    END IF;  
  END LOOP;  
  
  -- Close the cursor  
  CLOSE customer\_cur;  
  
  -- Commit the changes  
  COMMIT;  
END;  
  
  
**Scenario 2:**  
  
DECLARE  
  -- Cursor to loop through customers  
  CURSOR customer\_cur IS  
    SELECT customer\_id, balance  
    FROM customers;  
  
  -- Variable to hold the current customer's data  
  customer\_rec customer\_cur%ROWTYPE;  
BEGIN  
  -- Open the cursor  
  OPEN customer\_cur;  
  
  -- Loop through each customer  
  LOOP  
    FETCH customer\_cur INTO customer\_rec;  
    EXIT WHEN customer\_cur%NOTFOUND;  
  
    -- Check if the customer's balance is over $10,000  
    IF customer\_rec.balance > 10000 THEN  
      -- Set the customer as a VIP  
      UPDATE customers  
      SET IsVIP = TRUE  
      WHERE customer\_id = customer\_rec.customer\_id;  
    END IF;  
  END LOOP;  
  
  -- Close the cursor  
  CLOSE customer\_cur;  
  
  -- Commit the changes  
  COMMIT;  
END;  
  
  
**Scenario 3:**  
  
DECLARE  
--Cursor to fetch loans due in the next 30 days  
  CURSOR loan\_cur IS  
    SELECT customer\_name, loan\_id, due\_date  
    FROM loans  
    WHERE due\_date BETWEEN SYSTIMESTAMP AND SYSTIMESTAMP + INTERVAL ‘30’ DAY;  
  
--Variable to hold the current loan’s data  
  Loan\_rec loan\_cur%ROWTYPE;  
BEGIN  
--    Open the cursor  
  OPEN loan\_cur;  
  
--    Loop through each loan  
  LOOP  
    FETCH loan\_cur INTO loan\_rec;  
    EXIT WHEN loan\_cur%NOTFOUND;  
  
--    Print a reminder message for each customer  
    DBMS\_OUTPUT.PUT\_LINE(‘Reminder: ‘ || loan\_rec.customer\_name || ‘, Loan ID ‘ || loan\_rec.loan\_id || ‘ is due on ‘ || loan\_rec.due\_date);  
  END LOOP;  
  
--    Close the cursor  
  CLOSE loan\_cur;  
END;  
  
  
  
**Exercise 2 : Error Handling**  
  
**Scenario 1:**  
  
CREATE OR REPLACE PROCEDURE SafeTransferFunds(  
  p\_from\_account\_id NUMBER,  
  p\_to\_account\_id NUMBER,  
  p\_amount NUMBER  
) AS  
BEGIN  
  -- Start a new transaction  
  BEGIN TRANSACTION;  
  
  -- Check if the from\_account has sufficient funds  
  IF (SELECT balance FROM accounts WHERE account\_id = p\_from\_account\_id) < p\_amount THEN  
    -- Log error and roll back transaction  
    RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient funds in from\_account');  
  END IF;  
  
  -- Deduct funds from from\_account  
  UPDATE accounts  
  SET balance = balance - p\_amount  
  WHERE account\_id = p\_from\_account\_id;  
  
  -- Add funds to to\_account  
  UPDATE accounts  
  SET balance = balance + p\_amount  
  WHERE account\_id = p\_to\_account\_id;  
  
  -- Commit the transaction  
  COMMIT;  
EXCEPTION  
  WHEN OTHERS THEN  
    -- Roll back transaction on error  
    ROLLBACK;  
    -- Log error message  
    DBMS\_OUTPUT.PUT\_LINE('Error transferring funds: ' || SQLERRM);  
    -- Re-raise the error  
    RAISE;  
END;  
  
  
**Scenario 2:**  
  
CREATE OR REPLACE PROCEDURE UpdateSalary(  
  P\_employee\_id NUMBER,  
  P\_percentage NUMBER  
) AS  
BEGIN  
--    Update the salary  
  UPDATE employees  
  SET salary = salary + (salary \* (p\_percentage / 100))  
  WHERE employee\_id = p\_employee\_id;  
  
--    Check if the update was successful  
  IF SQL%ROWCOUNT = 0 THEN  
--    Log error and raise exception  
    DBMS\_OUTPUT.PUT\_LINE(‘Error: Employee ID ‘ || p\_employee\_id || ‘ does not exist’);  
    RAISE\_APPLICATION\_ERROR(-20002, ‘Employee ID does not exist’);  
  END IF;  
EXCEPTION  
  WHEN OTHERS THEN  
--    Log error message  
    DBMS\_OUTPUT.PUT\_LINE(‘Error updating salary: ‘ || SQLERRM);  
--    Re-raise the error  
    RAISE;  
END;  
  
  
**Scenario 3:**  
  
CREATE OR REPLACE PROCEDURE AddNewCustomer(  
  p\_customer\_id NUMBER,  
  p\_name VARCHAR2,  
  p\_email VARCHAR2,  
  p\_phone VARCHAR2  
) AS  
BEGIN  
  -- Insert the new customer  
  INSERT INTO Customers (customer\_id, name, email, phone)  
  VALUES (p\_customer\_id, p\_name, p\_email, p\_phone);  
    
EXCEPTION  
  WHEN DUP\_VAL\_ON\_INDEX THEN  
    -- Log error and prevent insertion  
    DBMS\_OUTPUT.PUT\_LINE('Error: Customer ID ' || p\_customer\_id || ' already exists');  
    RAISE\_APPLICATION\_ERROR(-20003, 'Customer ID already exists');  
  WHEN OTHERS THEN  
    -- Log error message  
    DBMS\_OUTPUT.PUT\_LINE('Error adding customer: ' || SQLERRM);  
    -- Re-raise the error  
    RAISE;  
END;  
  
  
  
**Exercise 3: Stored Procedures**  
      
**Scenario 1:**  
  
CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest AS  
BEGIN  
  -- Update the balance of all savings accounts  
  UPDATE Savings\_Accounts  
  SET balance = balance + (balance \* 0.01);  
    
  -- Commit the changes  
  COMMIT;  
EXCEPTION  
  WHEN OTHERS THEN  
    -- Log error message  
    DBMS\_OUTPUT.PUT\_LINE('Error processing monthly interest: ' || SQLERRM);  
    -- Roll back the changes  
    ROLLBACK;  
    -- Re-raise the error  
    RAISE;  
END;  
  
  
**Scenario 2:**  
  
CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(  
  P\_department\_id NUMBER,  
  P\_bonus\_percentage NUMBER  
) AS  
BEGIN  
--    Update the salary of employees in the given department  
  UPDATE Employees  
  SET salary = salary + (salary \* (p\_bonus\_percentage / 100))  
  WHERE department\_id = p\_department\_id;  
    
--    Commit the changes  
  COMMIT;  
EXCEPTION  
  WHEN OTHERS THEN  
--    Log error message  
    DBMS\_OUTPUT.PUT\_LINE(‘Error updating employee bonus: ‘ || SQLERRM);  
--    Roll back the changes  
    ROLLBACK;  
--    Re-raise the error  
    RAISE;  
END;  
  
  
**Scenario 3:**  
  
CREATE OR REPLACE PROCEDURE TransferFunds(  
  P\_source\_account\_id NUMBER,  
  P\_target\_account\_id NUMBER,  
  P\_amount NUMBER  
) AS  
  V\_source\_balance NUMBER;  
BEGIN  
--    Check the source account balance  
  SELECT balance INTO v\_source\_balance  
  FROM Accounts  
  WHERE account\_id = p\_source\_account\_id;  
    
--    Check if the source account has sufficient balance  
  IF v\_source\_balance < p\_amount THEN  
    RAISE\_APPLICATION\_ERROR(-20004, ‘Insufficient balance in source account’);  
  END IF;  
    
--    Deduct the amount from the source account  
  UPDATE Accounts  
  SET balance = balance – p\_amount  
  WHERE account\_id = p\_source\_account\_id;  
    
--    Add the amount to the target account  
  UPDATE Accounts  
  SET balance = balance + p\_amount  
  WHERE account\_id = p\_target\_account\_id;  
    
--    Commit the changes  
  COMMIT;  
EXCEPTION  
  WHEN OTHERS THEN  
--    Log error message  
    DBMS\_OUTPUT.PUT\_LINE(‘Error transferring funds: ‘ || SQLERRM);  
--    Roll back the changes  
    ROLLBACK;  
--    Re-raise the error  
    RAISE;  
  
  
  
**Exercise 4: Functions**  
  
**Scenario 1:**  
CREATE OR REPLACE FUNCTION CalculateAge(p\_date\_of\_birth DATE) RETURN NUMBER AS  
  v\_age NUMBER;  
BEGIN  
  v\_age := TRUNC(MONTHS\_BETWEEN(SYSDATE, p\_date\_of\_birth) / 12);  
  RETURN v\_age;  
END;  
  
SELECT CalculateAge('1990-01-01') FROM dual;  
  
  
**Scenario 2:**  
  
CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(  
  p\_loan\_amount NUMBER,  
  p\_interest\_rate NUMBER,  
  p\_loan\_duration NUMBER  
) RETURN NUMBER AS  
  v\_monthly\_interest\_rate NUMBER;  
  v\_number\_of\_payments NUMBER;  
  v\_monthly\_installment NUMBER;  
BEGIN  
  -- Convert annual interest rate to monthly interest rate  
  v\_monthly\_interest\_rate := p\_interest\_rate / 1200;  
    
  -- Calculate the number of payments  
  v\_number\_of\_payments := p\_loan\_duration \* 12;  
    
  -- Calculate the monthly installment  
  v\_monthly\_installment := p\_loan\_amount \* v\_monthly\_interest\_rate \*  
    POWER(1 + v\_monthly\_interest\_rate, v\_number\_of\_payments) /  
    (POWER(1 + v\_monthly\_interest\_rate, v\_number\_of\_payments) - 1);  
    
  RETURN v\_monthly\_installment;  
END;  
  
  
**Scenario 3:**  
CREATE OR REPLACE FUNCTION HasSufficientBalance(  
  p\_account\_id NUMBER,  
  p\_amount NUMBER  
) RETURN BOOLEAN AS  
  v\_balance NUMBER;  
BEGIN  
  -- Get the current balance of the account  
  SELECT balance INTO v\_balance  
  FROM Accounts  
  WHERE account\_id = p\_account\_id;  
    
  -- Check if the balance is sufficient  
  IF v\_balance >= p\_amount THEN  
    RETURN TRUE;  
  ELSE  
    RETURN FALSE;  
  END IF;  
EXCEPTION  
  WHEN NO\_DATA\_FOUND THEN  
    -- If the account does not exist, return FALSE  
    RETURN FALSE;  
END;  
SELECT HasSufficientBalance(123, 1000) FROM dual;  
  
  
  
**Exercise 5: Triggers**  
  
**Scenario 1:**  
  
CREATE OR REPLACE TRIGGER UpdateCustomerLastModified  
BEFORE UPDATE ON Customers  
FOR EACH ROW  
BEGIN  
  :NEW.LastModified := SYSTIMESTAMP;  
END;  
  
  
**Scenario 2:**  
CREATE OR REPLACE TRIGGER LogTransaction  
AFTER INSERT ON Transactions  
FOR EACH ROW  
BEGIN  
  INSERT INTO AuditLog (  
    TransactionID,  
    TransactionType,  
    TransactionDate,  
    LogDate  
  ) VALUES (  
    :NEW.TransactionID,  
    'INSERT',  
    :NEW.TransactionDate,  
    SYSTIMESTAMP  
  );  
END;  
  
  
**Scenario 3:**  
  
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 AccountID = :NEW.AccountID) THEN  
      RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient balance 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;  
  END IF;  
END;  
  
  
  
**Exerrcise 6: Cursors**      
**Scenario 1:**  
  
DECLARE  
  -- Explicit cursor to retrieve transactions for the current month  
  CURSOR GenerateMonthlyStatements IS  
    SELECT t.AccountID, c.Name, t.TransactionDate, t.Amount, t.TransactionType  
    FROM Transactions t  
    JOIN Customers c ON t.AccountID = c.AccountID  
    WHERE EXTRACT(MONTH FROM t.TransactionDate) = EXTRACT(MONTH FROM SYSTIMESTAMP)  
    AND EXTRACT(YEAR FROM t.TransactionDate) = EXTRACT(YEAR FROM SYSTIMESTAMP)  
    ORDER BY t.AccountID, t.TransactionDate;  
  
  -- Variables to store cursor data  
  v\_AccountID NUMBER;  
  v\_Name VARCHAR2(100);  
  v\_TransactionDate DATE;  
  v\_Amount NUMBER;  
  v\_TransactionType VARCHAR2(10);  
  
  -- Variable to store the previous account ID  
  v\_PreviousAccountID NUMBER;  
  
BEGIN  
  -- Open the cursor  
  OPEN GenerateMonthlyStatements;  
  
  -- Loop through the cursor  
  LOOP  
    FETCH GenerateMonthlyStatements INTO v\_AccountID, v\_Name, v\_TransactionDate, v\_Amount, v\_TransactionType;  
  
    -- Exit the loop when no more data is found  
    EXIT WHEN GenerateMonthlyStatements%NOTFOUND;  
  
    -- Print a header for each customer  
    IF v\_AccountID != v\_PreviousAccountID THEN  
      DBMS\_OUTPUT.PUT\_LINE('Statement for ' || v\_Name || ':');  
      DBMS\_OUTPUT.PUT\_LINE('-------------------------------');  
    END IF;  
  
    -- Print transaction details  
    DBMS\_OUTPUT.PUT\_LINE(v\_TransactionDate || ' ' || v\_TransactionType || ': ' || v\_Amount);  
  
    -- Update the previous account ID  
    v\_PreviousAccountID := v\_AccountID;  
  END LOOP;  
  
  -- Close the cursor  
  CLOSE GenerateMonthlyStatements;  
END;  
  
  
**Scenario 2:**  
  
DECLARE  
  -- Explicit cursor to retrieve accounts  
  CURSOR ApplyAnnualFee IS  
    SELECT AccountID, Balance  
    FROM Accounts;  
  
  -- Variables to store cursor data  
  v\_AccountID NUMBER;  
  v\_Balance NUMBER;  
  
  -- Annual fee amount  
  v\_AnnualFee NUMBER := 20;  -- Replace with the actual fee amount  
  
BEGIN  
  -- Open the cursor  
  OPEN ApplyAnnualFee;  
  
  -- Loop through the cursor  
  LOOP  
    FETCH ApplyAnnualFee INTO v\_AccountID, v\_Balance;  
  
    -- Exit the loop when no more data is found  
    EXIT WHEN ApplyAnnualFee%NOTFOUND;  
  
    -- Check if the balance is sufficient to deduct the fee  
    IF v\_Balance >= v\_AnnualFee THEN  
      -- Deduct the annual fee from the balance  
      UPDATE Accounts  
      SET Balance = Balance - v\_AnnualFee  
      WHERE AccountID = v\_AccountID;  
  
      -- Commit the change  
      COMMIT;  
  
      -- Print a message indicating the fee was deducted  
      DBMS\_OUTPUT.PUT\_LINE('Annual fee deducted from account ' || v\_AccountID);  
    ELSE  
      -- Print a message indicating insufficient balance  
      DBMS\_OUTPUT.PUT\_LINE('Insufficient balance in account ' || v\_AccountID);  
    END IF;  
  END LOOP;  
  
  -- Close the cursor  
  CLOSE ApplyAnnualFee;  
END;  
  
  
**Scenario 3:**  
  
DECLARE  
  -- Explicit cursor to retrieve loans  
  CURSOR UpdateLoanInterestRates IS  
    SELECT LoanID, InterestRate, LoanType  
    FROM Loans;  
  
  -- Variables to store cursor data  
  v\_LoanID NUMBER;  
  v\_InterestRate NUMBER;  
  v\_LoanType VARCHAR2(20);  
  
  -- New interest rates based on loan type  
  v\_NewInterestRate NUMBER;  
  
BEGIN  
  -- Open the cursor  
  OPEN UpdateLoanInterestRates;  
  
  -- Loop through the cursor  
  LOOP  
    FETCH UpdateLoanInterestRates INTO v\_LoanID, v\_InterestRate, v\_LoanType;  
  
    -- Exit the loop when no more data is found  
    EXIT WHEN UpdateLoanInterestRates%NOTFOUND;  
  
    -- Determine the new interest rate based on loan type  
    CASE v\_LoanType  
      WHEN 'Personal' THEN v\_NewInterestRate := 6.5;  
      WHEN 'Mortgage' THEN v\_NewInterestRate := 4.2;  
      WHEN 'Auto' THEN v\_NewInterestRate := 5.8;  
      ELSE v\_NewInterestRate := 0;  -- Default rate for other loan types  
    END CASE;  
  
    -- Update the interest rate  
    UPDATE Loans  
    SET InterestRate = v\_NewInterestRate  
    WHERE LoanID = v\_LoanID;  
  
    -- Commit the change  
    COMMIT;  
  
    -- Print a message indicating the loan ID and new interest rate  
    DBMS\_OUTPUT.PUT\_LINE('Updated loan ' || v\_LoanID || ' with new interest rate: ' || v\_NewInterestRate);  
  END LOOP;  
  
  -- Close the cursor  
  CLOSE UpdateLoanInterestRates;  
END;  
  
  
  
**Exercise 7: Packages**  
  
**Scenario 1:**  
  
  
CREATE OR REPLACE PACKAGE CustomerManagement AS  
  -- Procedure to add a new customer  
  PROCEDURE AddCustomer(  
    p\_Name VARCHAR2,  
    p\_Address VARCHAR2,  
    p\_PhoneNumber VARCHAR2,  
    p\_Email VARCHAR2  
  );  
  
  -- Procedure to update customer details  
  PROCEDURE UpdateCustomer(  
    p\_CustomerID NUMBER,  
    p\_Name VARCHAR2,  
    p\_Address VARCHAR2,  
    p\_PhoneNumber VARCHAR2,  
    p\_Email VARCHAR2  
  );  
  
  -- Function to get customer balance  
  FUNCTION GetCustomerBalance(  
    p\_CustomerID NUMBER  
  ) RETURN NUMBER;  
END CustomerManagement;  
  
CREATE OR REPLACE PACKAGE BODY CustomerManagement AS  
  PROCEDURE AddCustomer(  
    p\_Name VARCHAR2,  
    p\_Address VARCHAR2,  
    p\_PhoneNumber VARCHAR2,  
    p\_Email VARCHAR2  
  ) AS  
  BEGIN  
    INSERT INTO Customers (Name, Address, PhoneNumber, Email)  
    VALUES (p\_Name, p\_Address, p\_PhoneNumber, p\_Email);  
  END AddCustomer;  
  
  PROCEDURE UpdateCustomer(  
    p\_CustomerID NUMBER,  
    p\_Name VARCHAR2,  
    p\_Address VARCHAR2,  
    p\_PhoneNumber VARCHAR2,  
    p\_Email VARCHAR2  
  ) AS  
  BEGIN  
    UPDATE Customers  
    SET Name = p\_Name, Address = p\_Address, PhoneNumber = p\_PhoneNumber, Email = p\_Email  
    WHERE CustomerID = p\_CustomerID;  
  END UpdateCustomer;  
  
  FUNCTION GetCustomerBalance(  
    p\_CustomerID NUMBER  
  ) RETURN NUMBER AS  
    v\_Balance NUMBER;  
  BEGIN  
    SELECT Balance INTO v\_Balance  
    FROM Accounts  
    WHERE CustomerID = p\_CustomerID;  
    RETURN v\_Balance;  
  END GetCustomerBalance;  
END CustomerManagement;  
  
  
**Scenario 2:**  
CREATE OR REPLACE PACKAGE EmployeeManagement AS  
  -- Procedure to hire a new employee  
  PROCEDURE HireEmployee(  
    p\_FirstName VARCHAR2,  
    p\_LastName VARCHAR2,  
    p\_JobTitle VARCHAR2,  
    p\_HireDate DATE,  
    p\_Salary NUMBER  
  );  
  
  -- Procedure to update employee details  
  PROCEDURE UpdateEmployee(  
    p\_EmployeeID NUMBER,  
    p\_FirstName VARCHAR2,  
    p\_LastName VARCHAR2,  
    p\_JobTitle VARCHAR2,  
    p\_Salary NUMBER  
  );  
  
  -- Function to calculate annual salary  
  FUNCTION CalculateAnnualSalary(  
    p\_Salary NUMBER  
  ) RETURN NUMBER;  
END EmployeeManagement;  
  
CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS  
  PROCEDURE HireEmployee(  
    p\_FirstName VARCHAR2,  
    p\_LastName VARCHAR2,  
    p\_JobTitle VARCHAR2,  
    p\_HireDate DATE,  
    p\_Salary NUMBER  
  ) AS  
  BEGIN  
    INSERT INTO Employees (FirstName, LastName, JobTitle, HireDate, Salary)  
    VALUES (p\_FirstName, p\_LastName, p\_JobTitle, p\_HireDate, p\_Salary);  
  END HireEmployee;  
  
  PROCEDURE UpdateEmployee(  
    p\_EmployeeID NUMBER,  
    p\_FirstName VARCHAR2,  
    p\_LastName VARCHAR2,  
    p\_JobTitle VARCHAR2,  
    p\_Salary NUMBER  
  ) AS  
  BEGIN  
    UPDATE Employees  
    SET FirstName = p\_FirstName, LastName = p\_LastName, JobTitle = p\_JobTitle, Salary = p\_Salary  
    WHERE EmployeeID = p\_EmployeeID;  
  END UpdateEmployee;  
  
  FUNCTION CalculateAnnualSalary(  
    p\_Salary NUMBER  
  ) RETURN NUMBER AS  
  BEGIN  
    RETURN p\_Salary \* 12;  
  END CalculateAnnualSalary;  
END EmployeeManagement;  
  
  
**Scenario 3:**  
  
CREATE OR REPLACE PACKAGE AccountOperations AS  
  -- Procedure to open a new account  
  PROCEDURE OpenAccount(  
    p\_CustomerID NUMBER,  
    p\_AccountType VARCHAR2,  
    p\_InitialBalance NUMBER  
  );  
  
  -- Procedure to close an account  
  PROCEDURE CloseAccount(  
    p\_AccountID NUMBER  
  );  
  
  -- Function to get the total balance of a customer across all accounts  
  FUNCTION GetTotalBalance(  
    p\_CustomerID NUMBER  
  ) RETURN NUMBER;  
END AccountOperations;  
  
CREATE OR REPLACE PACKAGE BODY AccountOperations AS  
  PROCEDURE OpenAccount(  
    p\_CustomerID NUMBER,  
    p\_AccountType VARCHAR2,  
    p\_InitialBalance NUMBER  
  ) AS  
  BEGIN  
    INSERT INTO Accounts (CustomerID, AccountType, Balance)  
    VALUES (p\_CustomerID, p\_AccountType, p\_InitialBalance);  
  END OpenAccount;  
  
  PROCEDURE CloseAccount(  
    p\_AccountID NUMBER  
  ) AS  
  BEGIN  
    UPDATE Accounts  
    SET Status = 'CLOSED'  
    WHERE AccountID = p\_AccountID;  
  END CloseAccount;  
  
  FUNCTION GetTotalBalance(  
    p\_CustomerID NUMBER  
  ) RETURN NUMBER AS  
    v\_TotalBalance NUMBER;  
  BEGIN  
    SELECT SUM(Balance) INTO v\_TotalBalance  
    FROM Accounts  
    WHERE CustomerID = p\_CustomerID AND Status = 'ACTIVE';  
    RETURN v\_TotalBalance;  
  END GetTotalBalance;  
END AccountOperations;  
  
  
  
  
  
​