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.

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.

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.

→ALTER TABLE Customers ADD IsVIP BOOLEAN DEFAULT FALSE;

```
-- FIRST QUESTION
DELIMITER $$
CREATE PROCEDURE ApplySeniorLoanDiscount()
BEGIN
  DECLARE done INT DEFAULT FALSE;
  DECLARE cust_id INT;
  DECLARE dob DATE;
  DECLARE cur CURSOR FOR SELECT CustomerID, DOB FROM Customers;
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;
  OPEN cur;
  read_loop: LOOP
    FETCH cur INTO cust_id, dob;
    IF done THEN
      LEAVE read_loop;
    END IF;
    IF TIMESTAMPDIFF(YEAR, dob, CURDATE()) > 60 THEN
      UPDATE Loans
      SET InterestRate = InterestRate - 1
```

```
            WHERE CustomerID = cust_id;
        END IF;
    END LOOP;
CLOSE cur;
END$$
DELIMITER ;
CALL ApplySeniorLoanDiscount();
SELECT l.LoanID, l.CustomerID, c.Name, l.InterestRate
FROM Loans l
JOIN Customers c ON l.CustomerID = c.CustomerID
WHERE TIMESTAMPDIFF(YEAR, c.DOB, CURDATE()) > 60;
```

-- SECOND QUESTION

```
DELIMITER $$
CREATE PROCEDURE PromoteHighBalanceCustomers()
BEGIN
    UPDATE Customers
    SET IsVIP = TRUE
    WHERE Balance > 10000;
END$$
DELIMITER ;
CALL PromoteHighBalanceCustomers();
SELECT CustomerID, Name, Balance, IsVIP
FROM Customers
WHERE Balance > 10000;
```

-- THIRD QUESTION

```
DELIMITER $$
CREATE PROCEDURE SendUpcomingLoanReminders()
```

```
BEGIN
    SELECT l.LoanID, c.Name AS CustomerName, l.EndDate
    FROM Loans l
    JOIN Customers c ON l.CustomerID = c.CustomerID
    WHERE l.EndDate BETWEEN CURDATE() AND DATE_ADD(CURDATE(), INTERVAL 30 DAY);
END$$
DELIMITER ;
CALL SendUpcomingLoanReminders();
```

Output:

```
+--------+------------+----------+--------------+
| LoanID | CustomerID | Name     | InterestRate |
+--------+------------+----------+--------------+
|      1 |          1 | John Doe |         5.00 |
+--------+------------+----------+--------------+
+------------+------------+----------+-------+
| CustomerID | Name       | Balance  | IsVIP |
+------------+------------+----------+-------+
|          2 | Jane Smith | 15000.00 |     1 |
+------------+------------+----------+-------+
```

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.

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.

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.

→

--First question

```
DELIMITER $$
CREATE PROCEDURE SafeTransferFunds (
   IN fromAccountID INT,
   IN toAccountID INT,
   IN transferAmount DECIMAL(10,2)
)
BEGIN
   DECLARE insufficient_funds CONDITION FOR SQLSTATE '45000';
   DECLARE EXIT HANDLER FOR SQLEXCEPTION
   BEGIN
      -- Rollback on error
      ROLLBACK;
      SELECT 'Error occurred during transfer. Transaction rolled back.' AS Message;
   END;
   START TRANSACTION;
   -- Check balance
   IF (SELECT Balance FROM Accounts WHERE AccountID = fromAccountID) < transferAmount THEN
      SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Insufficient funds';
   END IF;
   -- Perform transfer
   UPDATE Accounts SET Balance = Balance - transferAmount WHERE AccountID = fromAccountID;
   UPDATE Accounts SET Balance = Balance + transferAmount WHERE AccountID = toAccountID;
   COMMIT;
   SELECT 'Transfer completed successfully' AS Message;
END $$
DELIMITER ;
```

--Second Question

```
DELIMITER $$
```

```
CREATE PROCEDURE UpdateSalary (
    IN empID INT,
    IN percentIncrease DECIMAL(5,2)
)
BEGIN
    DECLARE empExists INT DEFAULT 0;
    -- Check if employee exists
    SELECT COUNT(*) INTO empExists FROM Employees WHERE EmployeeID = empID;
    IF empExists = 0 THEN
        SELECT CONCAT('Error: Employee ID ', empID, ' does not exist.') AS Message;
    ELSE
        UPDATE Employees
        SET Salary = Salary + (Salary * percentIncrease / 100)
        WHERE EmployeeID = empID;
        SELECT 'Salary updated successfully' AS Message;
    END IF;
END $$
DELIMITER ;
```

--THIRD QUESTION

```
DELIMITER $$
CREATE PROCEDURE AddNewCustomer (
    IN pCustomerID INT,
    IN pName VARCHAR(100),
    IN pDOB DATE,
    IN pBalance DECIMAL(10,2)
)
BEGIN
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
    BEGIN
```

```
    SELECT CONCAT('Error: Could not add customer with ID ', pCustomerID, ' (possibly duplicate).')
AS Message;
  END;
  IF EXISTS (SELECT 1 FROM Customers WHERE CustomerID = pCustomerID) THEN
    SELECT CONCAT('Customer ID ', pCustomerID, ' already exists.') AS Message;
  ELSE
    INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)
    VALUES (pCustomerID, pName, pDOB, pBalance, NOW());
    SELECT 'Customer added successfully' AS Message;
  END IF;
END $$
DELIMITER ;
CALL SafeTransferFunds(1, 2, 500.00);
CALL UpdateSalary(2, 10);
CALL AddNewCustomer(3, 'Sarah Lee', '1988-12-05', 2000.00);
```

Output:

```
+---------------------------------+
| Message                         |
+---------------------------------+
| Transfer completed successfully |
+---------------------------------+
+-----------------------------+
| Message                     |
+-----------------------------+
| Salary updated successfully |
+-----------------------------+
+-----------------------------+
| Message                     |
+-----------------------------+
| Customer added successfully |
+-----------------------------+
```

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.

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.

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.

→--FIRST QUESTION

```
DELIMITER $$
CREATE PROCEDURE ProcessMonthlyInterest()
BEGIN
   UPDATE Accounts
   SET Balance = Balance + (Balance * 0.01)
   WHERE AccountType = 'Savings';
   SELECT 'Monthly interest processed successfully.' AS Message;
END $$
DELIMITER ;
```

--SECOND QUESTION

```
DELIMITER $$
CREATE PROCEDURE UpdateEmployeeBonus(
   IN deptName VARCHAR(50),
   IN bonusPercent DECIMAL(5,2)
)
BEGIN
   UPDATE Employees
   SET Salary = Salary + (Salary * bonusPercent / 100)
```

```
    WHERE Department = deptName;
    SELECT CONCAT('Bonus applied to department: ', deptName) AS Message;
END $$
DELIMITER ;



DELIMITER $$
CREATE PROCEDURE TransferFunds(
    IN fromAccount INT,
    IN toAccount INT,
    IN amount DECIMAL(10,2)
)
BEGIN
    DECLARE fromBalance DECIMAL(10,2);
    -- Get balance of source account
    SELECT Balance INTO fromBalance FROM Accounts WHERE AccountID = fromAccount;
    -- Check for sufficient balance
    IF fromBalance < amount THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Insufficient balance in source account.';
    ELSE
        -- Deduct from source
        UPDATE Accounts
        SET Balance = Balance - amount
        WHERE AccountID = fromAccount;
        -- Add to destination
        UPDATE Accounts
        SET Balance = Balance + amount
        WHERE AccountID = toAccount;
        SELECT 'Funds transferred successfully.' AS Message;
    END IF;
```

END $$

DELIMITER ;

CALL ProcessMonthlyInterest();

CALL UpdateEmployeeBonus('HR', 5);

CALL TransferFunds(1, 2, 100.00);

Output:

```
+----------------------------------------+
| Message                                |
+----------------------------------------+
| Monthly interest processed successfully. |
+----------------------------------------+
+---------------------------------+
| Message                         |
+---------------------------------+
| Bonus applied to department: HR |
+---------------------------------+
+---------------------------------+
| Message                         |
+---------------------------------+
| Funds transferred successfully. |
+---------------------------------+
```

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.

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.

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.

→DELIMITER $$

```
CREATE FUNCTION CalculateAge(dob DATE)
RETURNS INT
DETERMINISTIC
BEGIN
   RETURN TIMESTAMPDIFF(YEAR, dob, CURDATE());
END $$
DELIMITER ;
SELECT Name, CalculateAge(DOB) AS Age FROM Customers;
--SECOND QUESTION
DELIMITER $$
CREATE FUNCTION CalculateMonthlyInstallment(
   loanAmount DECIMAL(10,2),
   annualRate DECIMAL(5,2),
   years INT
)
RETURNS DECIMAL(10,2)
DETERMINISTIC
BEGIN
   DECLARE r DECIMAL(10,8);
   DECLARE n INT;
   DECLARE emi DECIMAL(10,2);
   SET r = annualRate / 12 / 100;
   SET n = years * 12;
   IF r = 0 THEN
      SET emi = loanAmount / n;
   ELSE
      SET emi = loanAmount * r * POW(1 + r, n) / (POW(1 + r, n) - 1);
   END IF;
   RETURN ROUND(emi, 2);
END $$
```

```
DELIMITER ;
SELECT CalculateMonthlyInstallment(10000, 5, 3) AS MonthlyInstallment;
--THIRD QUESTION
DELIMITER $$
CREATE FUNCTION HasSufficientBalance(
    accID INT,
    amt DECIMAL(10,2)
)
RETURNS BOOLEAN
DETERMINISTIC
BEGIN
    DECLARE bal DECIMAL(10,2);
    SELECT Balance INTO bal FROM Accounts WHERE AccountID = accID;
    RETURN bal >= amt;
END $$
DELIMITER ;
SELECT HasSufficientBalance(1, 500) AS IsSufficient;
```

Output:

```
+------------+------+
| Name       | Age  |
+------------+------+
| John Doe   |   61 |
| Jane Smith |   34 |
+------------+------+
+--------------------+
| MonthlyInstallment |
+--------------------+
|             299.71 |
+--------------------+
+--------------+
| IsSufficient |
+--------------+
|            1 |
+--------------+
```

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.

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.

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.

→--FIRST QUESTION

```
DELIMITER $$

CREATE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON Customers

FOR EACH ROW

BEGIN

   SET NEW.LastModified = NOW();

END $$

DELIMITER ;

UPDATE Customers SET Balance = Balance + 100 WHERE CustomerID = 1;

SELECT * FROM Customers WHERE CustomerID = 1;
```

--SECOND QUESTION

```
CREATE TABLE AuditLog (

   LogID INT AUTO_INCREMENT PRIMARY KEY,

   AccountID INT,

   TransactionDate DATETIME,

   Amount DECIMAL(10,2),

   TransactionType VARCHAR(10),

   LoggedAt DATETIME

);

DELIMITER $$
```

```
CREATE TRIGGER LogTransaction
AFTER INSERT ON Transactions
FOR EACH ROW
BEGIN
   INSERT INTO AuditLog (AccountID, TransactionDate, Amount, TransactionType, LoggedAt)
   VALUES (NEW.AccountID, NEW.TransactionDate, NEW.Amount, NEW.TransactionType, NOW());
END $$
DELIMITER ;
INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)
VALUES (3, 1, NOW(), 100, 'Deposit');
SELECT * FROM AuditLog;
--THIRD QUESTION
DELIMITER $$
CREATE TRIGGER CheckTransactionRules
BEFORE INSERT ON Transactions
FOR EACH ROW
BEGIN
   DECLARE current_balance DECIMAL(10,2);
   SELECT Balance INTO current_balance FROM Accounts WHERE AccountID = NEW.AccountID;
   -- Rule 1: Deposit must be positive
   IF NEW.TransactionType = 'Deposit' AND NEW.Amount <= 0 THEN
      SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'Deposit amount must be positive';
   END IF;
   -- Rule 2: Withdrawal must not exceed balance
   IF NEW.TransactionType = 'Withdrawal' AND NEW.Amount > current_balance THEN
      SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'Insufficient funds for withdrawal';
   END IF;
END $$
```

DELIMITER ;

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)

VALUES (6, 1, NOW(), 50, 'Withdrawal');

Output:

```
+------------+----------+------------+---------+---------------------+-------+
| CustomerID | Name     | DOB        | Balance | LastModified        | IsVIP |
+------------+----------+------------+---------+---------------------+-------+
|          1 | John Doe | 1964-05-15 | 1100.00 | 2025-06-29 16:39:32 |     0 |
+------------+----------+------------+---------+---------------------+-------+
+-------+-----------+---------------------+--------+-----------------+---------------------+
| LogID | AccountID | TransactionDate     | Amount | TransactionType | LoggedAt            |
+-------+-----------+---------------------+--------+-----------------+---------------------+
|     1 |         1 | 2025-06-29 16:39:32 | 100.00 | Deposit         | 2025-06-29 16:39:32 |
+-------+-----------+---------------------+--------+-----------------+---------------------+
```

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.

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.

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.

→--FIRST QUESTION

DELIMITER $$

CREATE PROCEDURE GenerateMonthlyStatements()

BEGIN

DECLARE done INT DEFAULT FALSE;

DECLARE cid INT;

DECLARE cname VARCHAR(100);

DECLARE tdate DATE;

```
    DECLARE amount DECIMAL(10,2);
    DECLARE ttype VARCHAR(10);
    DECLARE cur CURSOR FOR
        SELECT c.CustomerID, c.Name, 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 MONTH(t.TransactionDate) = MONTH(CURDATE())
          AND YEAR(t.TransactionDate) = YEAR(CURDATE());
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;
    OPEN cur;
    read_loop: LOOP
        FETCH cur INTO cid, cname, tdate, amount, ttype;
        IF done THEN
            LEAVE read_loop;
        END IF;
        SELECT CONCAT('Customer ', cname, ' (ID: ', cid, ') had a ', ttype,
                      ' of $', amount, ' on ', tdate) AS Statement;
    END LOOP;
  CLOSE cur;
END$$
DELIMITER ;
--SECOND QUESTION
DELIMITER $$
CREATE PROCEDURE ApplyAnnualFee()
BEGIN
    DECLARE done INT DEFAULT FALSE;
    DECLARE acc_id INT;
    DECLARE fee DECIMAL(10,2) DEFAULT 50.00;
    DECLARE cur CURSOR FOR SELECT AccountID FROM Accounts;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;
```

```
    OPEN cur;

    read_loop: LOOP
        FETCH cur INTO acc_id;
        IF done THEN
            LEAVE read_loop;
        END IF;
        UPDATE Accounts SET Balance = Balance - fee WHERE AccountID = acc_id;
    END LOOP;
    CLOSE cur;
END$$
DELIMITER ;
--THIRD QUESTION
DELIMITER $$
CREATE PROCEDURE UpdateLoanInterestRates()
BEGIN
    DECLARE done INT DEFAULT FALSE;
    DECLARE loan_id INT;
    DECLARE amount DECIMAL(10,2);
    DECLARE rate DECIMAL(5,2);
    DECLARE cur CURSOR FOR SELECT LoanID, LoanAmount, InterestRate FROM Loans;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;
    OPEN cur;
    read_loop: LOOP
        FETCH cur INTO loan_id, amount, rate;
        IF done THEN
            LEAVE read_loop;
        END IF;
        IF amount > 10000 THEN
            SET rate = rate + 0.5;
```

```
        ELSE
            SET rate = rate + 0.2;
        END IF;
        UPDATE Loans SET InterestRate = rate WHERE LoanID = loan_id;
    END LOOP;
    CLOSE cur;
END$$
DELIMITER ;
CALL GenerateMonthlyStatements();
CALL ApplyAnnualFee();
CALL UpdateLoanInterestRates();
```

Output:

```
+-----------------------------------------------------------------+
| Statement                                                       |
+-----------------------------------------------------------------+
| Customer John Doe (ID: 1) had a Deposit of $200.00 on 2025-06-29 |
+-----------------------------------------------------------------+
+-------------------------------------------------------------------+
| Statement                                                         |
+-------------------------------------------------------------------+
| Customer Jane Smith (ID: 2) had a Withdrawal of $300.00 on 2025-06-29 |
+-------------------------------------------------------------------+
```

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.

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.

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.

→DELIMITER $$

```
CREATE PROCEDURE CustomerManagement_AddCustomer(
    IN p_CustomerID INT,
    IN p_Name VARCHAR(100),
    IN p_DOB DATE,
    IN p_Balance DECIMAL(10,2)
)
BEGIN
    INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)
    VALUES (p_CustomerID, p_Name, p_DOB, p_Balance, NOW());
END$$
DELIMITER ;
DELIMITER $$
CREATE PROCEDURE CustomerManagement_UpdateCustomer(
    IN p_CustomerID INT,
    IN p_Name VARCHAR(100),
    IN p_Balance DECIMAL(10,2)
)
BEGIN
    UPDATE Customers
    SET Name = p_Name,
        Balance = p_Balance,
        LastModified = NOW()
    WHERE CustomerID = p_CustomerID;
END$$
DELIMITER ;
DELIMITER $$

CREATE FUNCTION CustomerManagement_GetBalance(p_CustomerID INT)
RETURNS DECIMAL(10,2)
```

```
DETERMINISTIC
BEGIN
    DECLARE bal DECIMAL(10,2);
    SELECT Balance INTO bal FROM Customers WHERE CustomerID = p_CustomerID;
    RETURN bal;
END$$
DELIMITER ;
DELIMITER $$
CREATE PROCEDURE EmployeeManagement_Hire(
    IN p_EmployeeID INT,
    IN p_Name VARCHAR(100),
    IN p_Position VARCHAR(50),
    IN p_Salary DECIMAL(10,2),
    IN p_Department VARCHAR(50),
    IN p_HireDate DATE
)
BEGIN
    INSERT INTO Employees(EmployeeID, Name, Position, Salary, Department, HireDate)
    VALUES (p_EmployeeID, p_Name, p_Position, p_Salary, p_Department, p_HireDate);
END$$
DELIMITER ;
DELIMITER $$
CREATE PROCEDURE EmployeeManagement_UpdateDetails(
    IN p_EmployeeID INT,
    IN p_Salary DECIMAL(10,2),
    IN p_Position VARCHAR(50)
)
BEGIN
    UPDATE Employees
    SET Salary = p_Salary,
        Position = p_Position
```

```
    WHERE EmployeeID = p_EmployeeID;
END$$
DELIMITER ;
DELIMITER $$
CREATE FUNCTION EmployeeManagement_AnnualSalary(p_EmployeeID INT)
RETURNS DECIMAL(10,2)
DETERMINISTIC
BEGIN
    DECLARE sal DECIMAL(10,2);
    SELECT Salary * 12 INTO sal FROM Employees WHERE EmployeeID = p_EmployeeID;
    RETURN sal;
END$$
DELIMITER ;
DELIMITER $$
CREATE PROCEDURE AccountOperations_OpenAccount(
    IN p_AccountID INT,
    IN p_CustomerID INT,
    IN p_AccountType VARCHAR(20),
    IN p_Balance DECIMAL(10,2)
)
BEGIN
    INSERT INTO Accounts(AccountID, CustomerID, AccountType, Balance, LastModified)
    VALUES (p_AccountID, p_CustomerID, p_AccountType, p_Balance, NOW());
END$$
DELIMITER ;
DELIMITER $$
CREATE PROCEDURE AccountOperations_CloseAccount(p_AccountID INT)
BEGIN
    DELETE FROM Accounts WHERE AccountID = p_AccountID;
END$$
DELIMITER ;
```

```
DELIMITER $$
CREATE FUNCTION AccountOperations_TotalBalance(p_CustomerID INT)
RETURNS DECIMAL(10,2)
DETERMINISTIC
BEGIN
   DECLARE total DECIMAL(10,2);
   SELECT IFNULL(SUM(Balance), 0) INTO total FROM Accounts WHERE CustomerID = p_CustomerID;
   RETURN total;
END$$
DELIMITER ;
CALL CustomerManagement_AddCustomer(10, 'Tom Hardy', '1980-04-05', 2500);
CALL EmployeeManagement_Hire(101, 'Jane Doe', 'Analyst', 50000, 'Finance', CURDATE());
SELECT AccountOperations_TotalBalance(10);
```

Output:

```
+-----------------------------------+
| AccountOperations_TotalBalance(10) |
+-----------------------------------+
|                              0.00 |
+-----------------------------------+
```