**Week 2: Advanced SQL, Nunit and Moq**

**SQL Exercises: Advanced SQL Concepts**

**Exercise 1: RANKING AND WINDOW FUNCTIONS**

CREATE TABLE Products (

ProductID INT PRIMARY KEY,

ProductName NVARCHAR(100),

Category NVARCHAR(50),

Price DECIMAL(10, 2)

);

INSERT INTO Products (ProductID, ProductName, Category, Price) VALUES

(1, 'Laptop', 'Electronics', 1000.00),

(2, 'Smartphone', 'Electronics', 700.00),

(3, 'Tablet', 'Electronics', 700.00),

(12, 'Jacket', 'Fashion', 200.00),

(14, 'Watch', 'Fashion', 200.00),

(11, 'Shoes', 'Fashion', 150.00),

(6, 'Sofa', 'Furniture', 1200.00),

(7, 'Dining Table', 'Furniture', 800.00),

(8, 'Chair', 'Furniture', 800.00);

WITH RankedProducts AS (

SELECT

ProductID,

ProductName,

Category,

Price,

ROW\_NUMBER() OVER (PARTITION BY Category ORDER BY Price DESC) AS RowNum,

RANK() OVER (PARTITION BY Category ORDER BY Price DESC) AS RankNum,

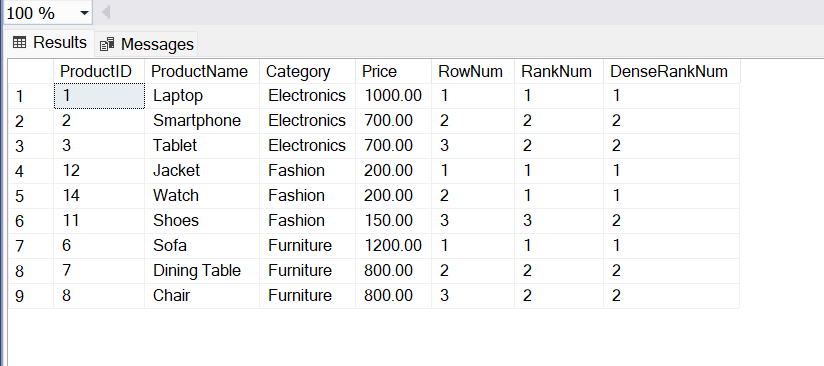
DENSE\_RANK() OVER (PARTITION BY Category ORDER BY Price DESC) AS DenseRankNum

FROM Products

)

SELECT \* FROM RankedProducts ORDER BY Category, RowNum;

**OUTPUT:**

****

**SQL Exercise: Stored Procedure**

**Exercise 1: Create a Stored Procedure**

**Create Stored Procedure to Get Employees by DepartmentID:**

CREATE PROCEDURE sp\_GetEmployeesByDepartment

@DeptID INT

AS

BEGIN

SELECT

E.EmployeeID,

E.FirstName,

E.LastName,

D.DepartmentName,

E.Salary,

E.JoinDate

FROM Employees E

INNER JOIN Departments D ON E.DepartmentID = D.DepartmentID

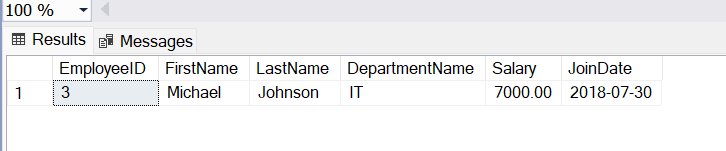
WHERE E.DepartmentID = @DeptID;

END;

**2. Select employee based on DepartmentID:**

EXEC sp\_GetEmployeesByDepartment @DeptID = 3;

**OUTPUT:**



**3. Stored procedure named ‘sp\_InsertEmployee’:**

CREATE PROCEDURE sp\_InsertEmployee

@FirstName VARCHAR(50),

@LastName VARCHAR(50),

@DepartmentID INT,

@Salary DECIMAL(10,2),

@JoinDate DATE

AS

BEGIN

INSERT INTO Employees (FirstName, LastName, DepartmentID, Salary, JoinDate)

VALUES (@FirstName, @LastName, @DepartmentID, @Salary, @JoinDate);

END;  
  
EXEC sp\_InsertEmployee

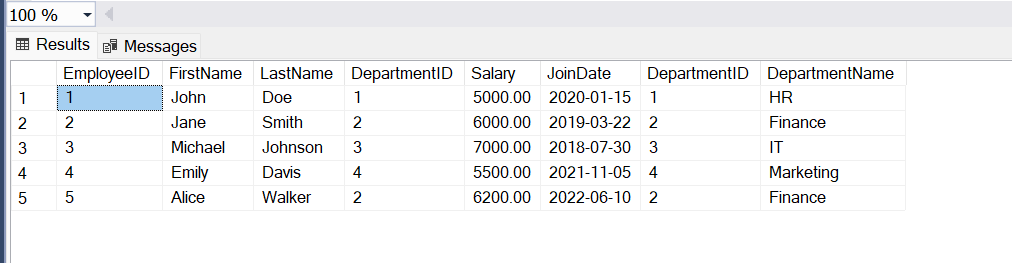
@FirstName = 'Alice',

@LastName = 'Walker',

@DepartmentID = 2,

@Salary = 6200.00,

@JoinDate = '2022-06-10';  
**OUTPUT:**



**SQL Exercise: Stored Procedure**

**Exercise 5: Return Data from a Stored Procedure**

**1.Stored Procedure with DepartmentdID:**

CREATE PROCEDURE sp\_GetEmployeeCountByDepartment

@DeptID INT

AS

BEGIN

SELECT

COUNT(\*) AS TotalEmployees

FROM Employees

WHERE DepartmentID = @DeptID;

END;

**2.Execute / Save the Procedure:**

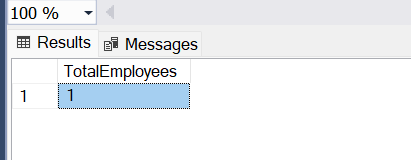
To save it, just run the above SQL block in SSMS (SQL Server Management Studio).  
Once successfully created, you will see:

"Commands completed successfully."

**3.Query for count the number of employees in the department:**

EXEC sp\_GetEmployeeCountByDepartment @DeptID = 3;

**OUTPUT:**



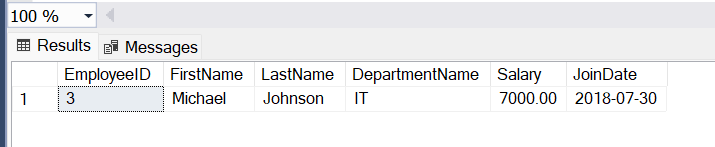
**SQL Exercise: Stored Procedure**

**Exercise 4: Execute a stored procedure**

**To Execute the stored procedure:**

EXEC sp\_GetEmployeesByDepartment @DeptID = 3;

**OUTPUT:**

****

**SQL Exercise: Functions**

**Exercise 7: Return data from Scalar function**

**1.Create the Scalar Function fn\_CalculateAnnualSalary:**

CREATE FUNCTION fn\_CalculateAnnualSalary

(

@EmployeeID INT

)

RETURNS DECIMAL(12,2)

AS

BEGIN

DECLARE @AnnualSalary DECIMAL(12,2)

SELECT @AnnualSalary = Salary \* 12

FROM Employees

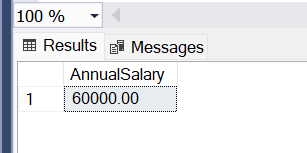
WHERE EmployeeID = @EmployeeID;

RETURN @AnnualSalary;

END;

SELECT dbo.fn\_CalculateAnnualSalary(1) AS AnnualSalary;

**OUTPUT:**

****

**2. Verify the result:**

SELECT

E.EmployeeID,

E.FirstName,

E.LastName,

D.DepartmentName,

E.Salary AS MonthlySalary,

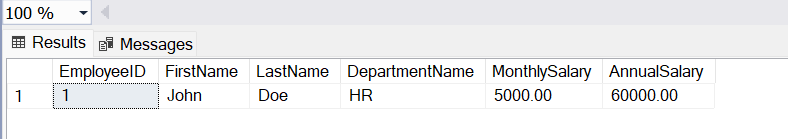
dbo.fn\_CalculateAnnualSalary(E.EmployeeID) AS AnnualSalary

FROM Employees E

JOIN Departments D ON E.DepartmentID = D.DepartmentID

WHERE E.EmployeeID = 1;

**OUTPUT:**

****

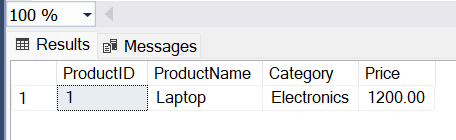
**SQL Exercise: Index.sql-** Hands-on in this document

**Exercise 1: Creating a Non-Clustered Index:**

Goal: Create a non-clustered index on the ProductName column in the Products table and compare query execution time before and after index creation.

**Step 1: Query to fetch product details before index creation**

SELECT \* FROM Products WHERE ProductName = 'Laptop';



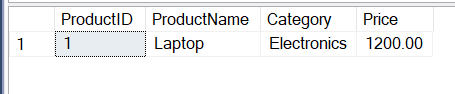
**Step 2: Create a non-clustered index on ProductName**

CREATE NONCLUSTERED INDEX idx\_ProductName

ON Products (ProductName);

**Step 3: Query to fetch product details after index creation**

SELECT \* FROM Products WHERE ProductName = 'Laptop';

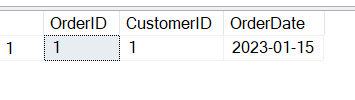


**Exercise 2: Creating a Clustered Index:**

Goal: Create a clustered index on the OrderDate column in the Orders table and compare query execution time before and after index creation.

**Step 1: Query to fetch orders before index creation**

SELECT \* FROM Orders WHERE OrderDate = '2023-01-15';



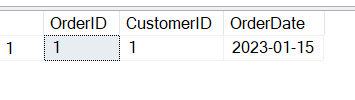
**Step 2: Create a clustered index on OrderDate**

CREATE NONCLUSTERED INDEX idx\_OrderDate

ON Orders (OrderDate);

**Step 3: Query to fetch orders after index creation**

SELECT \* FROM Orders WHERE OrderDate = '2023-01-15';

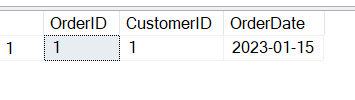


**Exercise 3: Creating a Composite Index:**

Goal: Create a composite index on the CustomerID and OrderDate columns in the Orders table and compare query execution time before and after index creation.

**Step 1: Query to fetch orders before index creation**

SELECT \* FROM Orders WHERE CustomerID = 1 AND OrderDate = '2023-01-15';



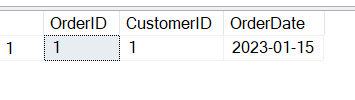
**Step 2: Create a composite index on CustomerID and OrderDate**

CREATE NONCLUSTERED INDEX idx\_CustomerID\_OrderDate

ON Orders (CustomerID, OrderDate);

**Step 3: Query to fetch orders after index creation**

SELECT \* FROM Orders WHERE CustomerID = 1 AND OrderDate = '2023-01-15';



**1.NUnit-Handson**

## **Objectives**

• Explain the meaning of Unit testing and its difference compared with Functional testing.

• List various types of testing.

• Understand the benefit of automated testing.

• Explain what is loosely coupled & testable design.

• Write your first testing program to validate a calculator addition operation.

• Understand the purpose of the [SetUp], [TearDown] & [Ignore] attributes.

• Explain the benefit of writing parameterised test cases with [TestCase].

## **1. Unit Testing vs Functional Testing**

Unit testing focuses on the smallest testable part of an application—usually a single method or class—ensuring it behaves correctly in isolation. Functional (integration) testing validates end‑to‑end flows against requirements, often exercising multiple components together.  
  
Mocking is used in unit tests to replace real dependencies with test doubles, keeping the test fast and deterministic.

## **2. Types of Testing**

• Unit Testing  
• Functional / Integration Testing  
• Automated Regression Testing  
• Performance / Load Testing  
• UI / End‑to‑End Testing  
• Security & Penetration Testing

## **3. Benefits of Automated Testing**

Automated tests run quickly, provide immediate feedback, prevent regressions, enable refactoring, and support CI/CD pipelines.

## **4. Loosely Coupled & Testable Design**

Writing code against abstractions (interfaces) rather than concrete classes eliminates hard dependencies. This makes components replaceable by mocks/stubs inside unit tests.

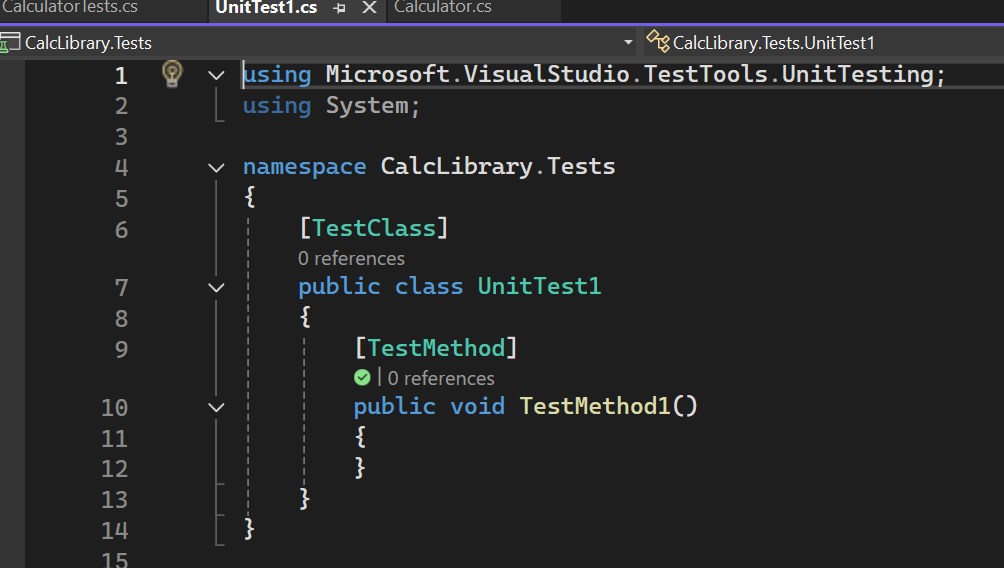
the steps listed below to write the NUnit test cases for the application.

* Create a Unit test project(.Net Framework) in the solution provided.
* Add the CalcLibrary project as reference
* Create a class “CalculatorTests” to write all the test cases for the methods in the solution
* Use the ‘TestFixture’, ‘SetUp’ and ‘TearDown’ attributes, to declare, initialize and cleanup activities respectively
* Create a Test method to check the addition functionality
* Use the ‘TestCase’ attribute to send the inputs and the expected result
* Use Assert.That to check the actual and expected result match

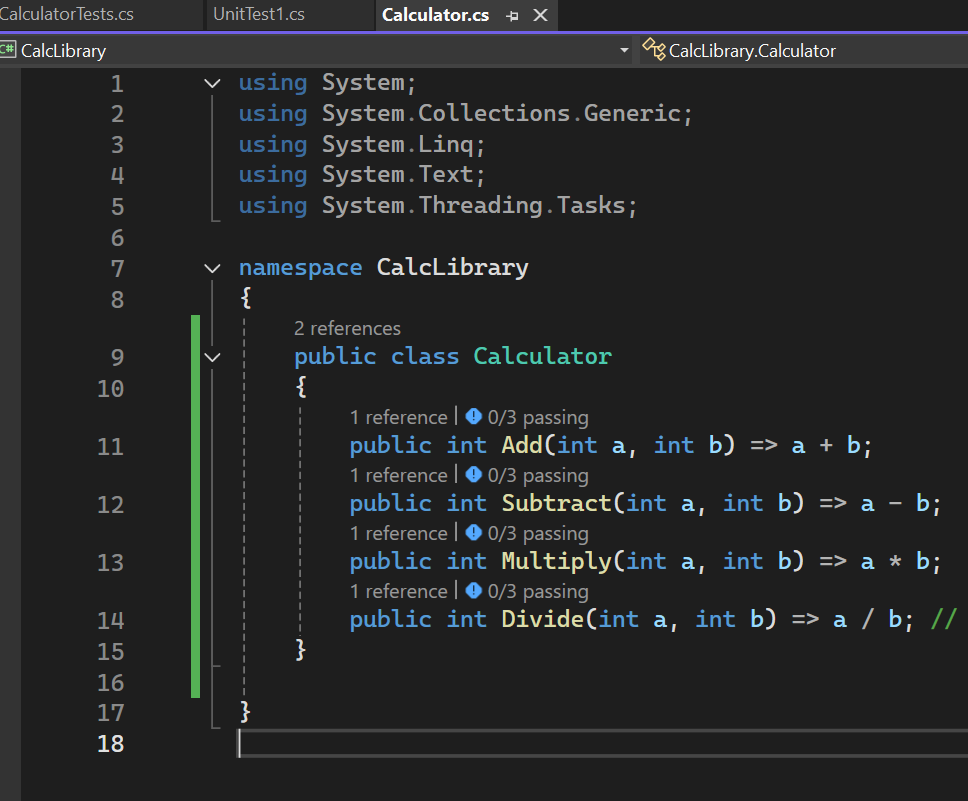
**CalculatorTests.cs:**



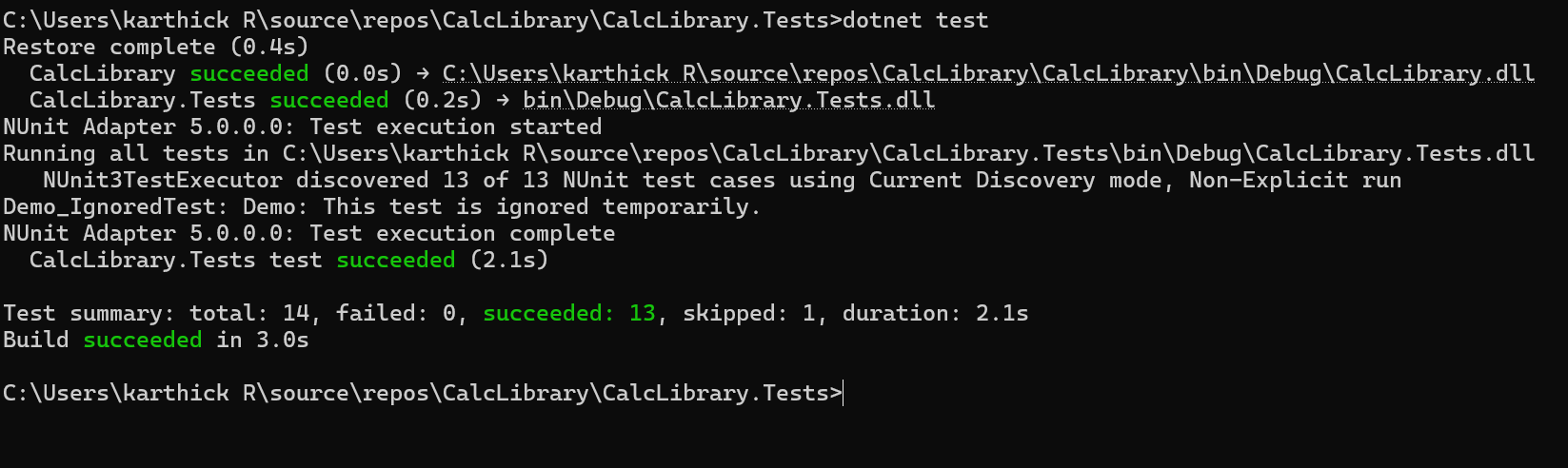
**UnitTest.cs:**



**Calculator.cs**



**OUTPUT:**

****

**1. Moq-Handson**

**1.Write Testable Code with Moq:**

Scenario:

You are tasked to write a unit test code for the below scenario.

The application in which you are teamed up with, deals with a mail server communication in which your application tries to send mail to its users upon every transaction. Your role is to write unit testing the module that contains send mail functionality. You wanted to perform testing the module without sending any email.

After investigating the problem scenario, you found a solution and that is creating mock objects of these external dependencies in the unit testing project so that you can achieve speedier test execution and loose coupling of code.

Task 1:

* Created a Class Library named CustomerCommLib.
* Defined an interface IMailSender with a SendMail() method.
* Implemented the interface in a class called MailSender using SmtpClient to send real emails.
* Created a class CustomerComm that injects IMailSender through its constructor.
* The method SendMailToCustomer() calls the injected SendMail() method.
* This structure makes the code testable by allowing mocking of the SendMail() method during unit testing.

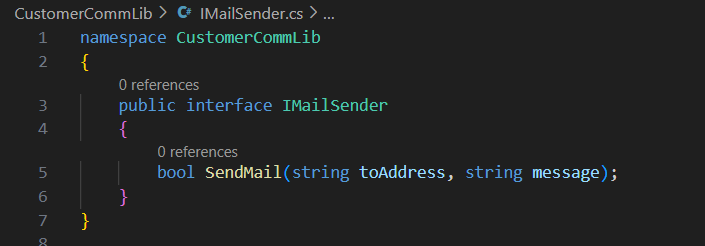
Task2 :

In this task, you will create unit test project which make use of NUnit framework and Moq.

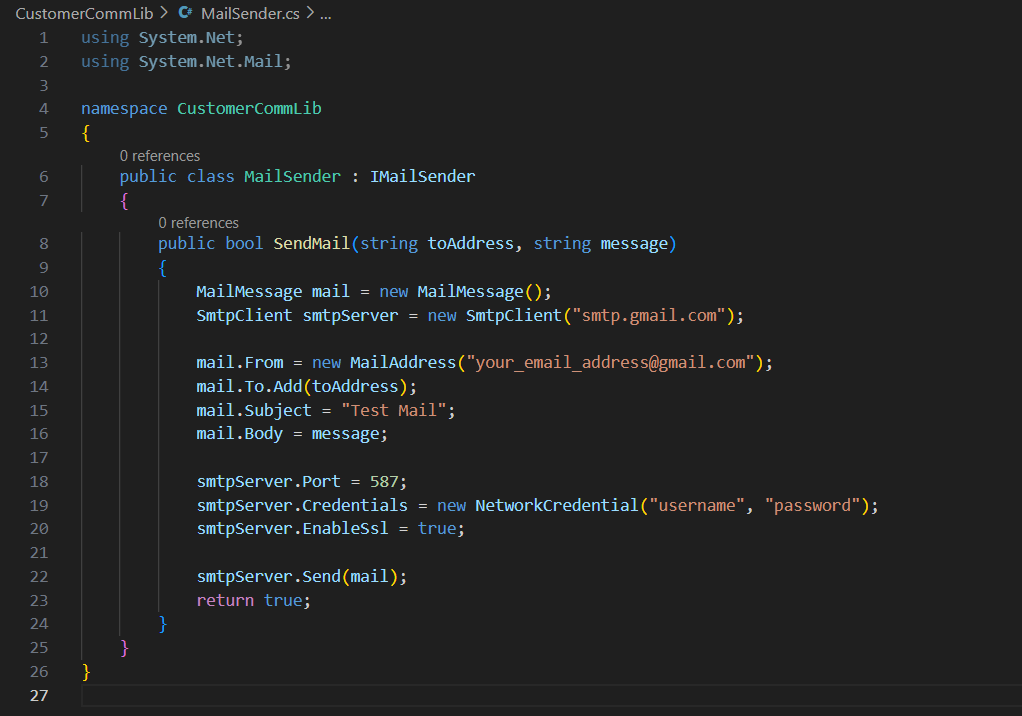
* Create a new class library project called CustomerComm.Tests and add the following external dependencies to it using NuGet Package Manager.
  + NUnit
  + NUnit Test Adapter
  + Moq
* Add the references of assemblies as appropriate including CustomerCommLib.
* Write unit test code and mock the MailSender (IMailSender) class.
* Use TestFixture, OneTimeSetUp and TestCase attribute classes on top of test class, init method and test method respectively.
* Configure the mock object in such away that SendMail() method will accept any two string arguments and always return true when SendMailToCustomer() gets invoked.
* Finally assert the return value to “true”.

**Code Implementation:**

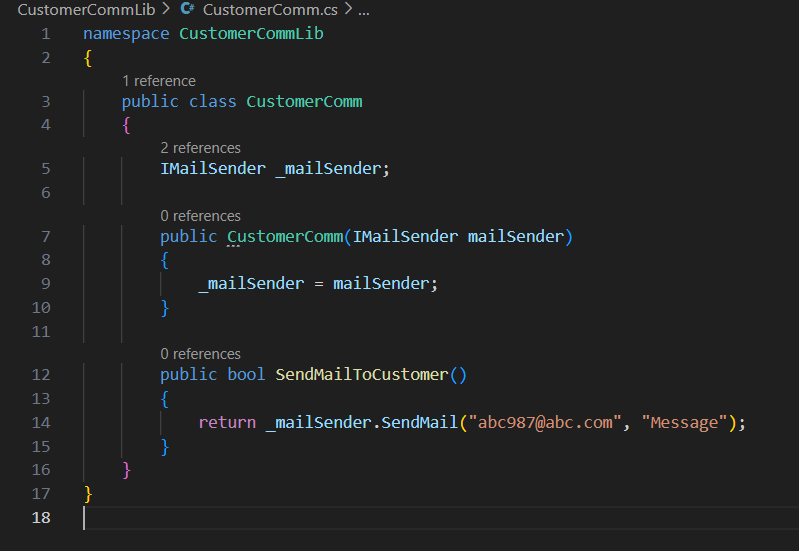
**IMailSender.cs:**



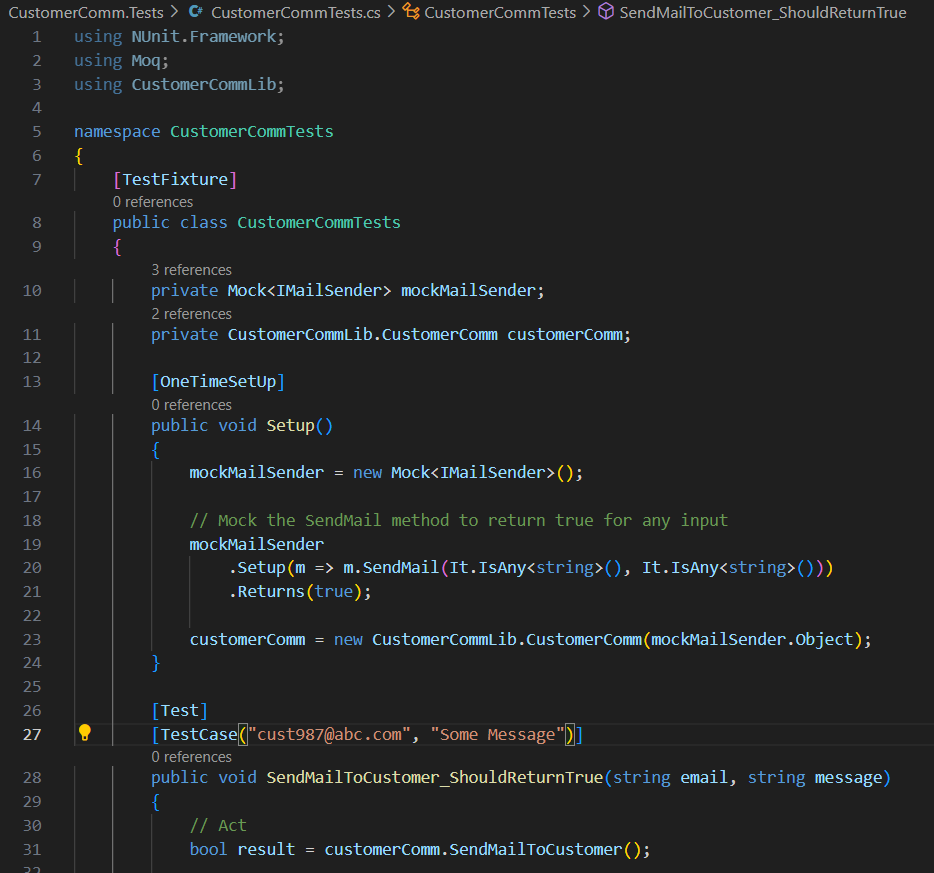
**MailSender.cs:**

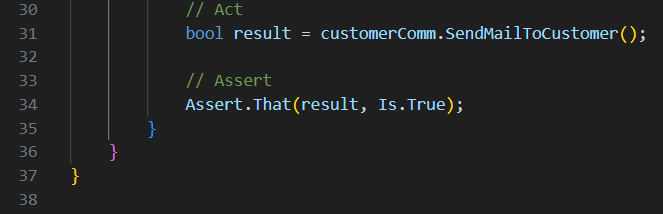


**CustomerComm.cs:**



**CustomerCommTests.cs**





**OUTPUT:**

**OUTPUT:**

