**NAME-MASHEERA AFRIN**

**SUPERSET ID-6362071**

**WEEK2:**

**ADVANCED SQL EXERCERCISE**

1. **RANKING WINDOW EXERCISE**

**1. Use ROW\_NUMBER() to assign a unique rank within each category.**

SELECT \*

FROM (

SELECT

ProductName,

Category,

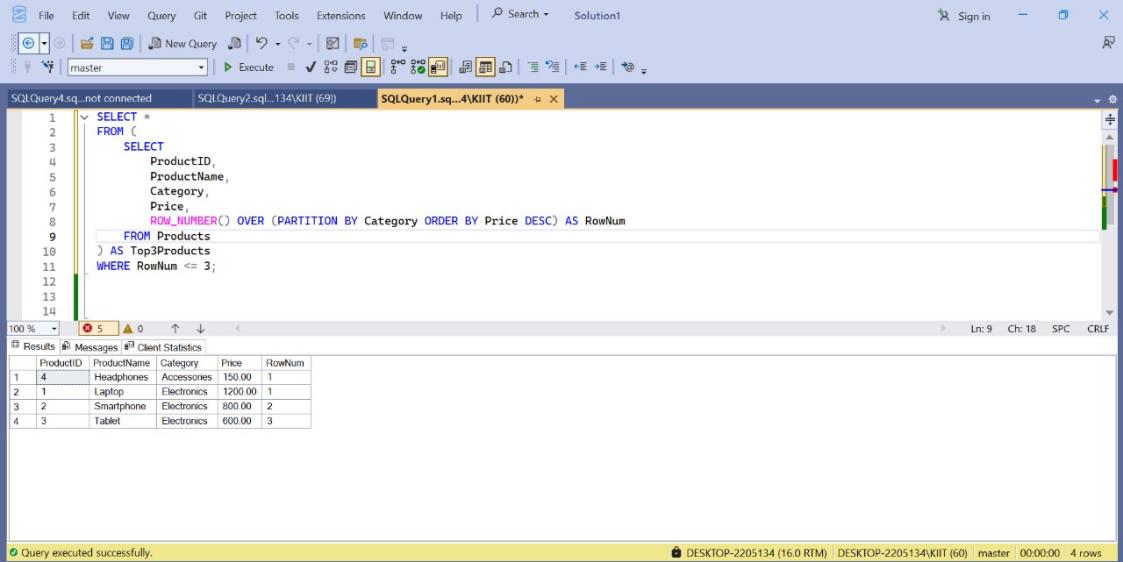
Price,

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

FROM Products

) AS Top3Products

WHERE RowNum <= 3;



1. **Use RANK() and DENSE\_RANK() to compare how ties are handled.**(**NOTE:**rank skips the number if there is tie whereas dense rank doesn’t .But the database which was provided there was no tie so the output of both are same)

SELECT

ProductID,

ProductName,

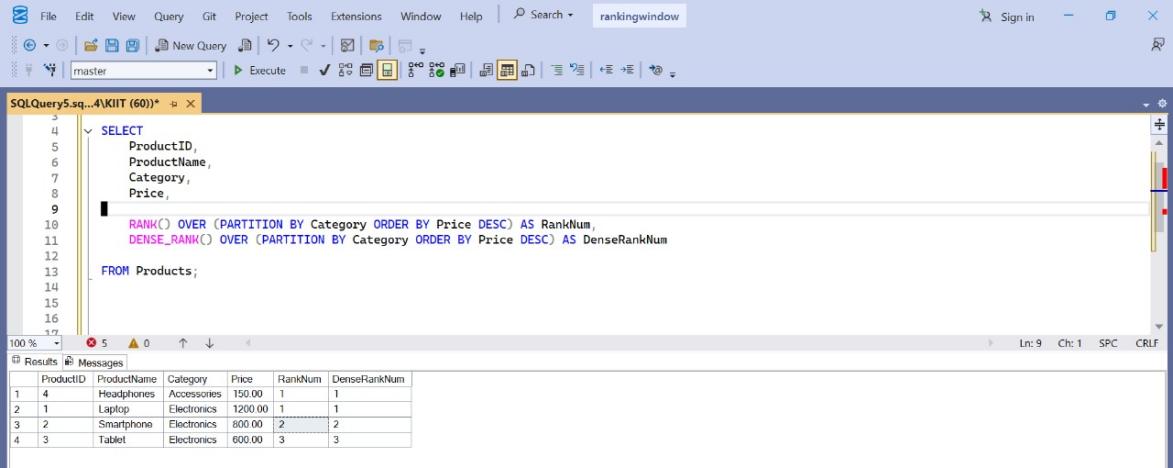
Category,

Price,

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;



1. **Use PARTITION BY Category and ORDER BY Price DESC**

SELECT

ProductID,

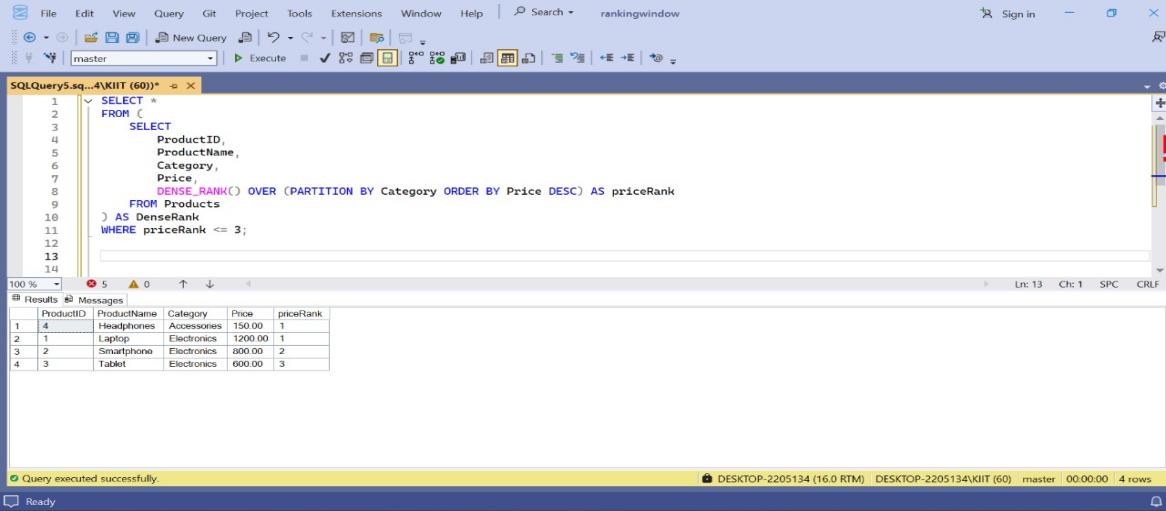
ProductName,

Category,

Price,

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

FROM Products;



**Exercise 1: Create a Stored Procedure**

1. **Define the stored procedure with a parameter for DepartmentID.**

CREATE PROCEDURE EmployeesByDepartment

@DepartmentID INT

AS

BEGIN

SELECT

EmployeeID,

FirstName,

LastName,

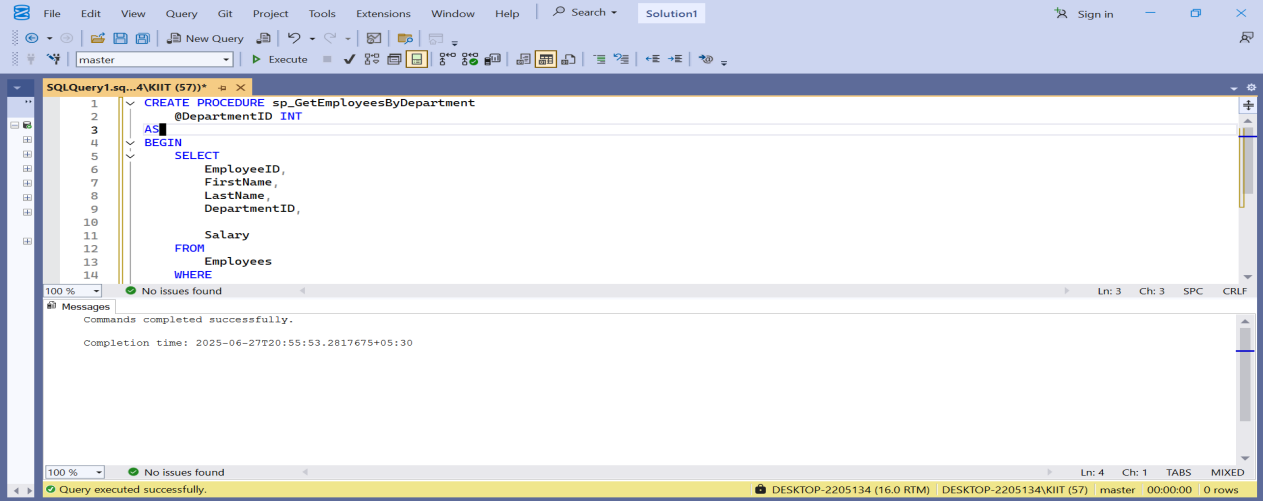
DepartmentID,

Salary

FROM

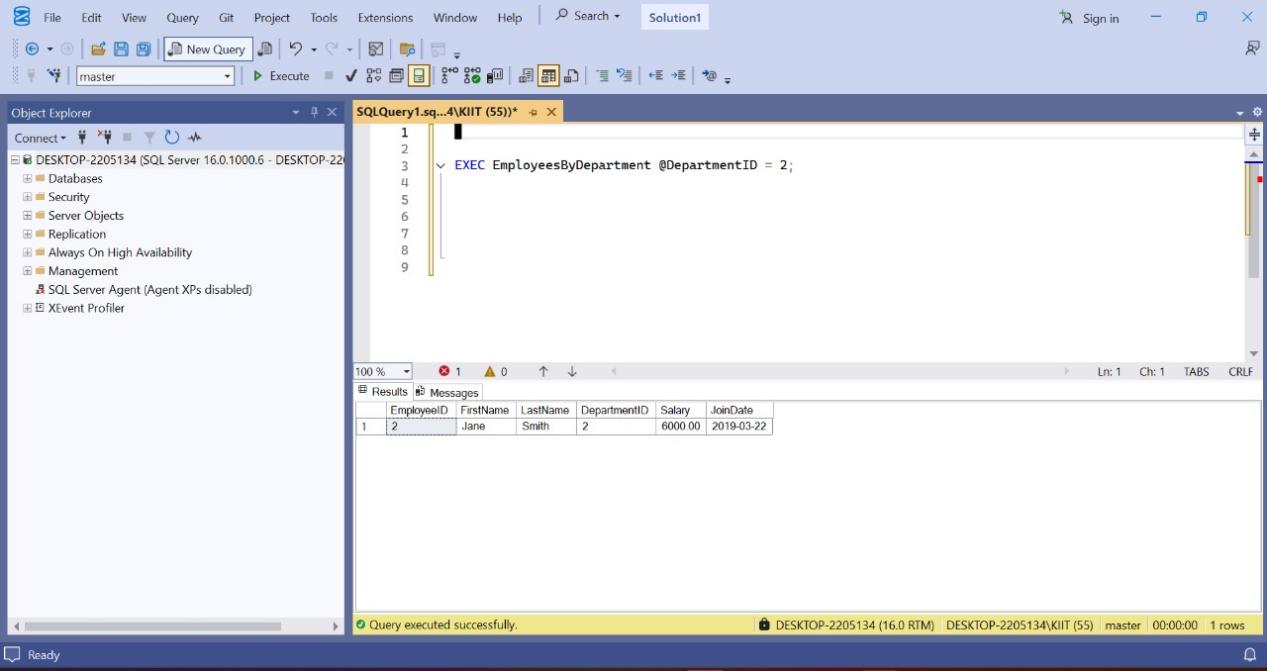
Employees

WHERE

DepartmentID = @DepartmentID END;

1. **Write the SQL query to select employee details based on the DepartmentID.**

EXEC EmployeesByDepartment @DepartmentID = 2;



1. **Create a stored procedure named 'sp\_InsertEmployee`.**

CREATEPROCEDURE sp\_InsertEmployee

@EmployeeID INT,

@FirstName VARCHAR(50),

@LastName VARCHAR(50),

@DepartmentID INT,

@Salary DECIMAL(10,2),

@JoinDate DATE

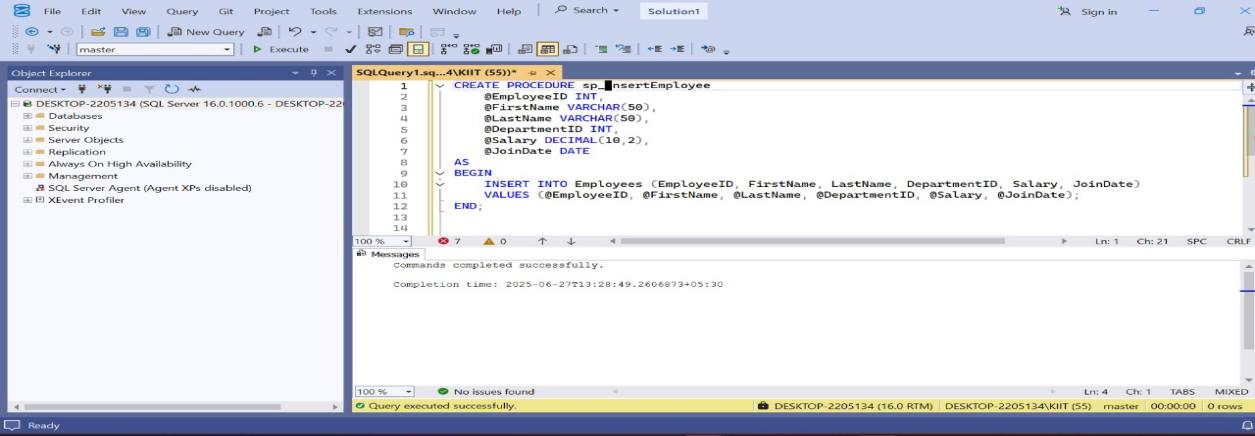
AS

BEGIN

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

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

END;



**Exercise 5: Return data from Stored Procedure**

1.CREATE PROCEDURE sp\_CountEmployees

@DepartmentID INT

AS

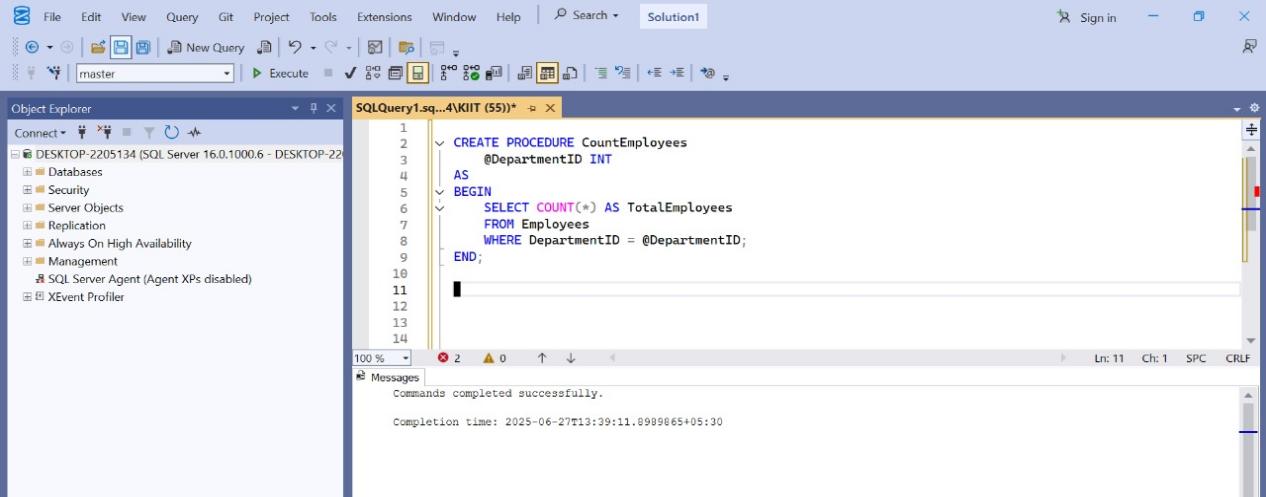
BEGIN

SELECT COUNT(\*) AS TotalEmployees

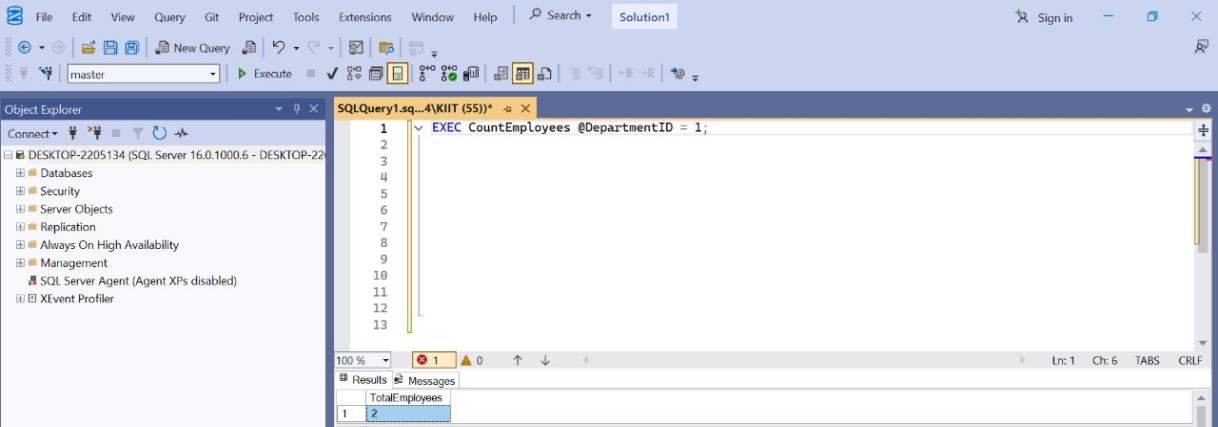
FROM Employees

WHERE DepartmentID = @DepartmentID;

END;



2.EXEC CountEmployees @Department=1;



**NUnit :**

1. **Explain the meaning of Unit testing and its difference on comparison with Functional testing Smallest unit to test mocking dependencies.**

Unit testing is the process of testing the smallest parts of your code, like it is a method in which we verify the code's correctness by running one by one. It's a key part of software development that improves code quality by testing each unit in isolation.In Unit Testing, we test the **smallest code unit**, like a method or class, **by mocking the external dependencies** it relies on, so we can focus only on the logic inside that unit.

|  |  |
| --- | --- |
| **UNIT TESTING** | **FUNCTIONAL TESTING** |
| It tests the flow, usability, and accessibility of the application. | It tests the functionality of the individual components of the application. |
| Unit Testing is a part of Functional Testing. | Functional Testing has multiple types of Testing, like Unit Testing, System Testing, etc., under its belt. |
| Unit test is fast and specific to individual components. | Functional testing is slow and tests the overall functionality of the application. |

1. **List various types of testing**

Unit testing, Functional testing, Automated testing, Performance testing

### ****Unit Testing****

### ****Purpose****: Tests individual components or functions of code.

**Example Tools**: JUnit (Java), NUnit (.NET), PyTest (Python).

**Who Performs It**: Developers.

**Key Focus**: Logic correctness of small code units (like functions, methods).

### ****2. Functional Testing****

**Purpose**: Verifies that the software functions according to specified requirements.

**Types Included**:Smoke Testing ,Sanity Testing ,Regression Testing ,Integration Testing ,System Testing

**Who Performs It**: Testers/QA Engineers.

**Example Tools**: Selenium, QTP, Postman (for API testing).

### ****3. Performance Testing****

**Purpose**: Checks how the system behaves under load.

**Types**:Load Testing (handles expected load) ,Stress Testing (under extreme load) ,Spike Testing (sudden changes in load) ,Endurance Testing (long duration)

**Example Tools**: JMeter, LoadRunner.

**Key Focus**: Speed, scalability, and stability.

**4. Automated Testing**

**Purpose**: Uses tools to automatically execute test cases.

**Common Areas**:

Regression Testing ,Unit Testing ,UI Testing ,API Testing

**Example Tools**: Selenium, Cypress, TestNG, Appium

**Benefits**: Fast feedback, reusable scripts, supports CI/CD.

1. **Understand the benefit automated testing.**
2. Saving Costs:

Automated tests can be run concurrently on multiple devices and environments without the need for manual intervention. This reduces the need for human testers and saves on labor costs. Automation can perform tests much faster than manual testing, saving time and reducing the overall cost of testing cycles.

2. Faster Feedback Loop

Automated tests provide instant feedback on the application's performance, allowing developers to identify and address issues quickly.

3. Guarantees Higher Accuracy

Automated tests perform the same steps and checks consistently, reducing the risk of human error associated with manual testing.

4. Detects Bugs Earlier

Automation testing promotes a shift-left approach, where testing is integrated early in the development cycle. This helps catch bugs at an early stage, reducing rework and cost.

5. Maximizes ROI

By saving time and resources, automation testing improves testing efficiency and productivity, leading to a higher return on investment (ROI). While initial setup and scripting may require investment, automated tests are easier to maintain and update, reducing long-term maintenance costs.

**Explain what is loosly coupled & testable design .Write code that is NOT dependent on the class for data**

A ****loosely coupled**** design means that components (like classes or modules) are independent of each other. This makes code:

**Easier to test**

**Easier to maintain**

**More flexible and reusable**

In **tight coupling**, one class directly depends on another, making testing and changes difficult.

### ****Testable Design****

A design is testable when:

You can **test components independently**

You can **inject mock or stub dependencies**

Business logic is **not hard-coded** to external systems (e.g., databases, APIs)

**Write code that is NOT dependent on the class for data.**

using System;

namespace MyApp{

public interface IDataProvider

{

string GetData();

}

public class DatabaseService : IDataProvider

{

public string GetData()

{

return "Data from Database";

}

}

public class MockDataProvider : IDataProvider

{

public string GetData()

{

return "Mock data for testing";

}

}

public class ReportGenerator

{

private readonly IDataProvider \_dataProvider;

public ReportGenerator(IDataProvider dataProvider)

{

\_dataProvider = dataProvider;

}

public void GenerateReport()

{

string data = \_dataProvider.GetData();

Console.WriteLine("Report Generated using: " + data);

}

}

public class Program

{

public static void Main(string[] args){

IDataProvider provider = new MockDataProvider();

ReportGenerator generator = new ReportGenerator(provider);

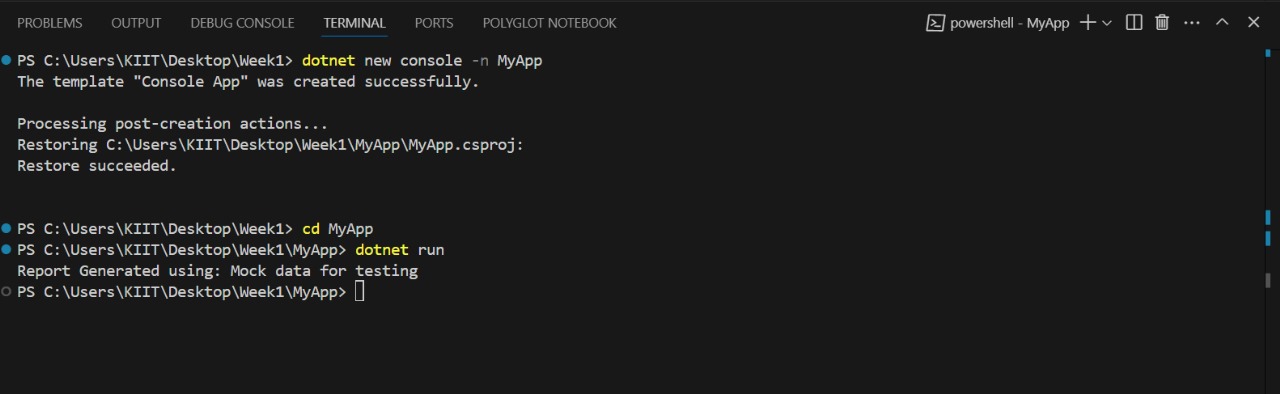
generator.GenerateReport();

}

}

}

**OUTPUT:**



1. **Write your first testing program to validate a calculator addition operation (TestFixture, Test).**

**Class1.cs**

namespace CalcLibrary

{

public class Calculator

{

public int Add(int a, int b)

{

return a + b;

}

}

} **UnitTest1.cs**

using NUnit.Framework;

using CalcLibrary;

namespace CalcLibrary.Tests

{

[TestFixture]

public class CalculatorTests

{

private Calculator calculator;

[SetUp]

public void Setup()

{

calculator = new Calculator();

}

[TearDown]

public void Cleanup()

{

calculator = null;

}

[TestCase(1, 2, 3)]

[TestCase(-1, -1, -2)]

[TestCase(0, 0, 0)]

public void Add\_ReturnsExpectedResult(int a, int b, int expected)

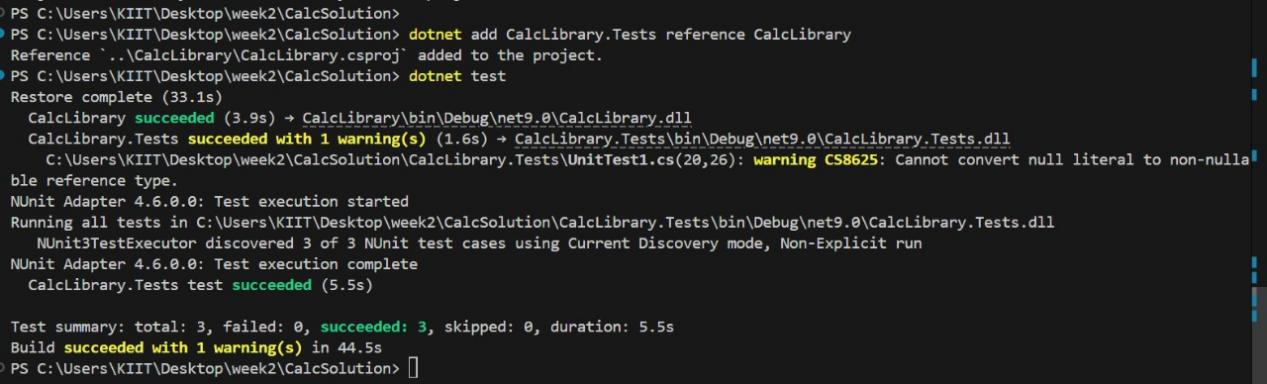
{ var result = calculator.Add(a, b);

Assert.That(result, Is.EqualTo(expected));

}

}

}

**Output:**

1. **Understand the need of [SetUp], [TearDown] & [Ignore] attributes.**

**SetUp:**Runs **before each test method** in the test class.  
To initialize common resources or setup code before every test — for example: opening a DB connection, setting default data, etc.

**TearDown**:Runs **after each test method**.  
To clean up or release resources after every test — e.g., closing connections, clearing temporary data, etc.

### [Ignore] :Skips the test case from being executed. You might want to **temporarily disable** a test that's not relevant, flaky, or under construction.

1. **Explain the benefit of writing parameterised test cases.TestCase**

Instead of writing **multiple similar tests** with different inputs and expected outputs, you can write **one test method** and pass values using [TestCase].

**Benefits of [TestCase]:**

Avoids **repetition**

Improves **readability**

Easy to **add new inputs** for testing

Keeps **test method short and focused.**

**Moq Unit**

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

**Task1:**

**Program.cs**

using System;

using CustomerCommLib;

class Program

{

static void Main(string[] args)

{

IMailSender sender = new MailSender();

CustomerComm comm = new CustomerComm(sender);

bool result = comm.SendMailToCustomer();

Console.WriteLine("Was mail sent? " + result);

}

}

**CustomerComm.cs**

namespace CustomerCommLib

{

public class CustomerComm

{

private readonly IMailSender \_mailSender;

public CustomerComm(IMailSender mailSender)

{

\_mailSender = mailSender;

}

public bool SendMailToCustomer()

{

return \_mailSender.SendMail("cust123@abc.com", "Some Message");

}

}

}

**MailSender.cs**

using System.Net;

using System.Net.Mail;

namespace CustomerCommLib

{

public interface IMailSender

{

bool SendMail(string toAddress, string message);

}

public class MailSender : IMailSender

{

public bool SendMail(string toAddress, string message)

{

MailMessage mail = new MailMessage();

SmtpClient SmtpServer = new SmtpClient("smtp.gmail.com");

mail.From = new MailAddress("your\_email\_address@gmail.com");

mail.To.Add(toAddress);

mail.Subject = "Test Mail";

mail.Body = message;

SmtpServer.Port = 587;

SmtpServer.Credentials = new NetworkCredential("username", "password");

SmtpServer.EnableSsl = true;

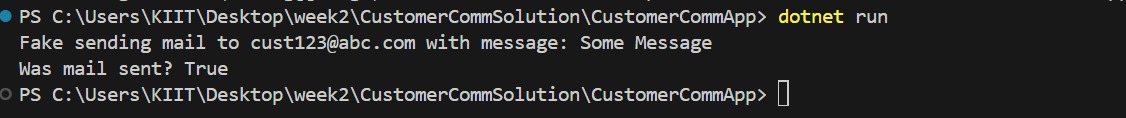
Console.WriteLine($"Fake sending mail to {toAddress} with message: {message}");

return true;

}

}

}

**OUTPUT:**

**TASK2:**

using NUnit.Framework;

using Moq;

using CustomerCommLib;

namespace CustomerComm.Tests

{

[TestFixture]

public class CustomerCommTests

{

private Mock<IMailSender> \_mockMailSender = null!;

private CustomerCommLib.CustomerComm \_customerComm = null!;

[OneTimeSetUp]

public void Setup()

{

\_mockMailSender = new Mock<IMailSender>();

\_customerComm = new CustomerCommLib.CustomerComm(\_mockMailSender.Object);}

[TestCase("abc@example.com", "Hello!")]

[TestCase("xyz@test.com", "Welcome back!")]

[TestCase("test@domain.com", "Test Message")]

public void SendMailToCustomer\_TestCases\_ReturnsTrue(string toAddress, string message)

{

\_mockMailSender

.Setup(m => m.SendMail(toAddress, message))

.Returns(true);

bool result = \_customerComm.SendMailToCustomer(toAddress, message);

Assert.That(result, Is.True);

}

}

}**OUTPUT**

