Exercise : Ranking and Window Functions

```sql
SELECT *
FROM (
    SELECT *,
           ROW_NUMBER() OVER (PARTITION BY Category ORDER BY Price DESC) AS RowNum
    FROM Products
) AS Ranked
WHERE RowNum <= 3;
SELECT *
FROM (
    SELECT *,
           RANK() OVER (PARTITION BY Category ORDER BY Price DESC) AS RankVal
    FROM Products
) AS Ranked
WHERE RankVal <= 3;
SELECT *
FROM (
    SELECT *,
           DENSE_RANK() OVER (PARTITION BY Category ORDER BY Price DESC) AS DenseRankVal
    FROM Products
) AS Ranked
WHERE DenseRankVal <= 3;

```

	ProductID	ProductName	Category	Price	RowNum
1	1	Laptop	Electronics	800.00	1
2	2	Smartphone	Electronics	700.00	2
3	3	Tablet	Electronics	500.00	3
4	5	Sofa	Furniture	400.00	1
5	6	Bookshelf	Furniture	150.00	2
6	4	Desk Chair	Furniture	120.00	3
7	8	Notebook	Stationery	3.50	1
8	9	Folder	Stationery	2.75	2
9	7	Pen	Stationery	2.50	3

	ProductID	ProductName	Category	Price	RankVal
1	1	Laptop	Electronics	800.00	1
2	2	Smartphone	Electronics	700.00	2
3	3	Tablet	Electronics	500.00	3
4	5	Sofa	Furniture	400.00	1
5	6	Bookshelf	Furniture	150.00	2
6	4	Desk Chair	Furniture	120.00	3
7	8	Notebook	Stationery	3.50	1
8	9	Folder	Stationery	2.75	2
9	7	Pen	Stationery	2.50	3

	ProductID	ProductName	Category	Price	DenseRankVal
1	1	Laptop	Electronics	800.00	1
2	2	Smartphone	Electronics	700.00	2
3	3	Tablet	Electronics	500.00	3
4	5	Sofa	Furniture	400.00	1
5	6	Bookshelf	Furniture	150.00	2
6	4	Desk Chair	Furniture	120.00	3
7	8	Notebook	Stationery	3.50	1
8	9	Folder	Stationery	2.75	2
9	7	Pen	Stationery	2.50	3

Exercise : Create a Stored Procedure

```sql
CREATE PROCEDURE 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;
```

```sql
EXEC sp_InsertEmployee
    @EmployeeID = 5,
    @FirstName = 'Alice',
    @LastName = 'Brown',
    @DepartmentID = 1,
    @Salary = 5200.00,
    @JoinDate = '2023-04-01';
```

	EmployeeID	FirstName	LastName	DepartmentID	Salary	JoinDate
1	1	John	Doe	1	5000.00	2020-01-15
2	2	Jane	Smith	2	6000.00	2019-03-22
3	3	Michael	Johnson	3	7000.00	2018-07-30
4	4	Emily	Davis	4	5500.00	2021-11-05
5	5	Alice	Brown	1	5200.00	2023-04-01

```sql
EXEC sp_InsertEmployee
    @EmployeeID = 6,
    @FirstName = 'Klein',
    @LastName = 'Moretti',
    @DepartmentID = 1,
    @Salary = 6666.66,
    @JoinDate = '2023-04-01';

Select * from Employees;
```

	EmployeeID	FirstName	LastName	DepartmentID	Salary	JoinDate
1	1	John	Doe	1	5000.00	2020-01-15
2	2	Jane	Smith	2	6000.00	2019-03-22
3	3	Michael	Johnson	3	7000.00	2018-07-30
4	4	Emily	Davis	4	5500.00	2021-11-05
5	5	Alice	Brown	1	5200.00	2023-04-01
6	6	Klein	Moretti	1	6666.66	2023-04-01

Exercise : Return Data from a Stored Procedure

```sql
CREATE PROCEDURE sp_GetEmployeeCountByDepartment
    @DepartmentID INT
AS
BEGIN
    SELECT COUNT(*) AS TotalEmployees
    FROM Employees
    WHERE DepartmentID = @DepartmentID;
END;
```

```sql
EXEC sp_GetEmployeeCountByDepartment @DepartmentID = 1;
```

	TotalEmployees
1	3

1. NUnit-Handson

1. What is Unit Testing (vs Functional Testing)?

Unit testing is all about testing the smallest pieces of code in isolation — usually individual methods or classes. It's done by developers to make sure each unit works exactly as expected. For example, if I write a method to calculate tax, a unit test would check if it's returning the right result for different inputs.
On the other hand, functional testing looks at the entire application or a feature and checks whether it behaves correctly based on business requirements. It's more like testing the system from the user's perspective.

Comparison	Unit Testing	Functional Testing
What it tests	Individual methods/classes	Features or flows in the application
Who does it	Usually developers	Often done by testers/QA
Dependencies	Usually mocked	Real-world dependencies used
Focus	Code correctness	User behavior and output

In short: unit testing checks if the code is right, functional testing checks if the behavior is right.

2. Types of Testing

There are different types of testing that we come across in software development. Here are a few important ones:
Unit Testing – Testing individual parts of the code (like a method).
Functional Testing – Making sure a feature or functionality works as expected.

Automated Testing – Using tools to run tests automatically, especially useful when testing the same thing repeatedly.
Performance Testing – Checks how the system performs under load, like checking response time during high traffic.
Each of these plays an important role at different stages of the development and deployment process.

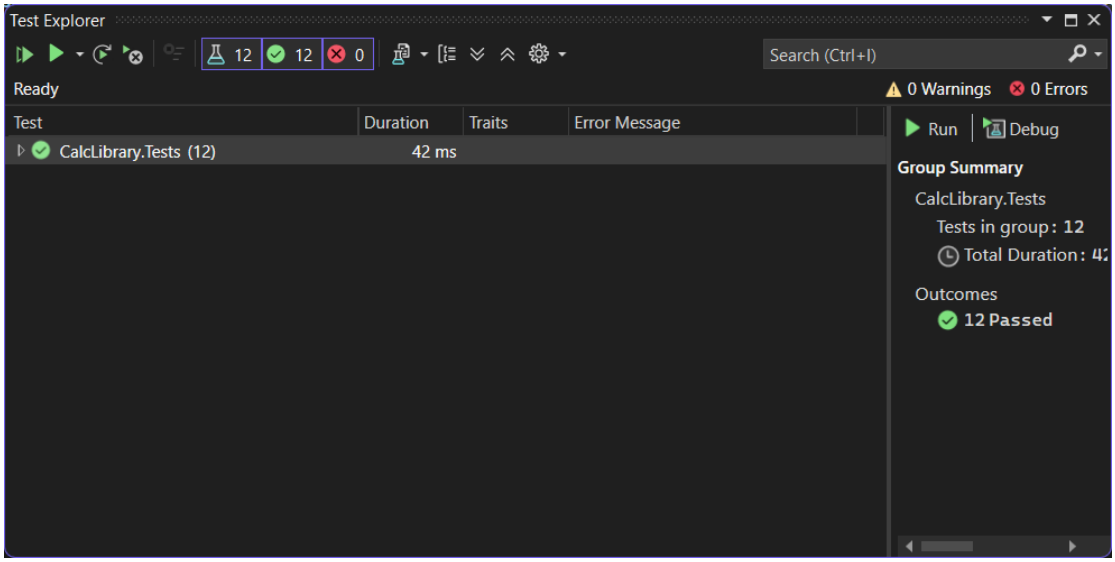
3. Why Automated Testing Matters

Automated testing can save a lot of time and effort. Once written, tests can be run repeatedly with just one click (or even automatically during builds). Some key benefits:
You can catch bugs early, even before a human sees them.
It saves time
It's great for regression testing — making sure new changes don't break old code.
Helps in writing clean and modular code, since tests encourage better design.

4. What is a Loosely Coupled and Testable Design?

A loosely coupled design is where parts of your code don't directly depend on each other. Instead, they talk through interfaces or abstractions. That way, when you test something, you can mock the other part.
Example: If your OrderService depends on PaymentProcessor, don't directly call it — instead, use an interface. Then in your unit tests, you can provide a fake/mock payment processor.
It's a small change in design, but it makes testing easier and the code more flexible.

TestFixture & Test



1. Moq-Handson

What is Mocking?
Mocking is the practice of creating simulated objects (called mocks) that mimic the behavior of real dependencies in your application during unit testing.

These mocks help isolate the unit of code being tested by replacing external systems (like databases, mail servers, file systems, etc.) with fake implementations.

Why Use Mocks in Unit Testing?

Unit tests are supposed to be:
Fast
Reliable
Independent
Using mocks allows you to:
Avoid external dependencies (e.g., network, database).
Focus on the logic of the code itself, not on the behavior of external systems.
Control the behavior of the dependency (e.g., force a method to return true).
Test edge cases easily (e.g., simulate an exception).
Mocking and Isolation in Unit Testing
Isolation means testing only one component/class without invoking its collaborators.
Mocks isolate the code by standing in for:
Web services
Databases
File systems
Email servers
Time and random generators
Example: If a class sends an email, we mock the MailSender to avoid actually sending an email during testing.

Isolating Dependencies Using Mocks and Stubs

Concept	Description	Example
Mock	A fake object that can verify interactions (e.g., was method X called?).	Moq verifying that SendMail() was called once.
Stub	A fake object that **returns predefined values** but **doesn't track interactions**.	A stubbed GetFiles() that always returns two test files.
Fake	A working, lightweight implementation (like an in-memory DB).	An in-memory repository instead of SQL Server.

```cs
using System.Net;
using System.Net.Mail;

0 references
public class MailSender : IMailSender
{
    3 references
    public bool SendMail(string toAddress, string message)
    {
        MailMessage mail = new MailMessage();
        SmtpClient smtpServer = new SmtpClient("smtp.gmail.com");

        mail.From = new MailAddress("your_email@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;

        smtpServer.Send(mail);
        return true;
    }
}
```

```cs
5 references
public interface IMailSender
{
    3 references
    bool SendMail(string toAddress, string message);
}
```

```
public class CustomerComm
{
    private IMailSender _mailSender;

    1 reference
    public CustomerComm(IMailSender mailSender)
    {
        _mailSender = mailSender;
    }

    1 reference
    public bool SendMailToCustomer()
    {
        return _mailSender.SendMail("cust123@abc.com", "Some Message");
    }
}
```

```
using NUnit.Framework;
using Moq;
using CustomerCommLib;

[TestFixture]
0 references
public class CustomerCommTests
{
    private Mock<IMailSender> _mockMailSender;
    private CustomerComm _customerComm;

    [OneTimeSetUp]
    0 references
    public void Setup()
    {
        _mockMailSender = new Mock<IMailSender>();
        _mockMailSender.Setup(x => x.SendMail(It.IsAny<string>(), It.IsAny<string>())).Returns(true);

        _customerComm = new CustomerComm(_mockMailSender.Object);
    }

    [Test]
    0 references
    public void SendMailToCustomer_ReturnsTrue()
    {
        var result = _customerComm.SendMailToCustomer();
        Assert.IsTrue(result);
    }
}
```