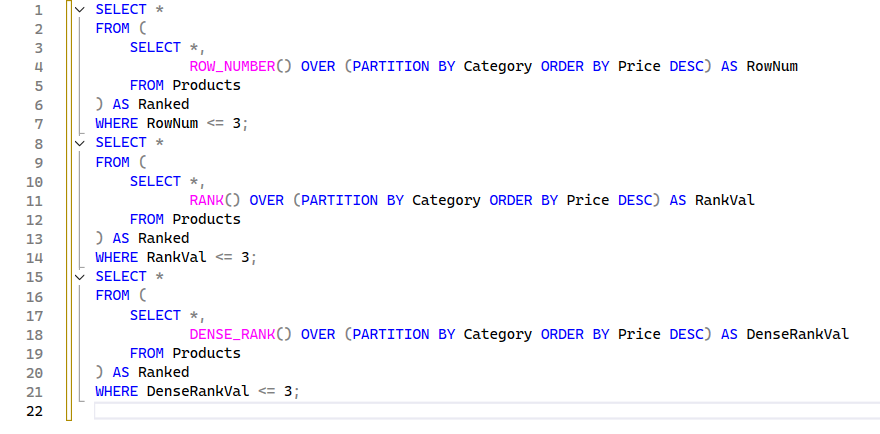
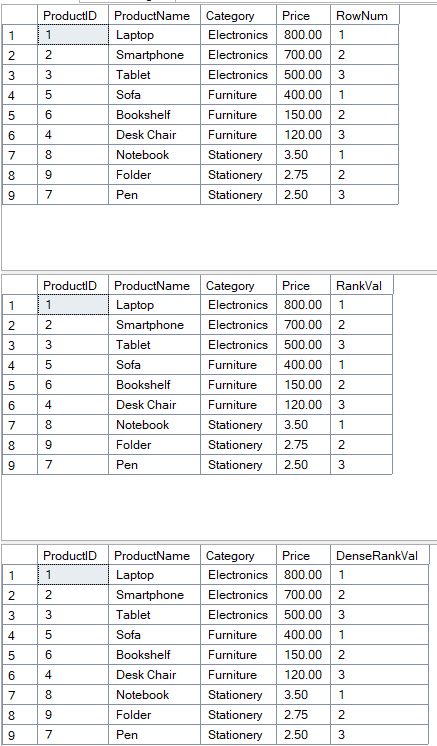
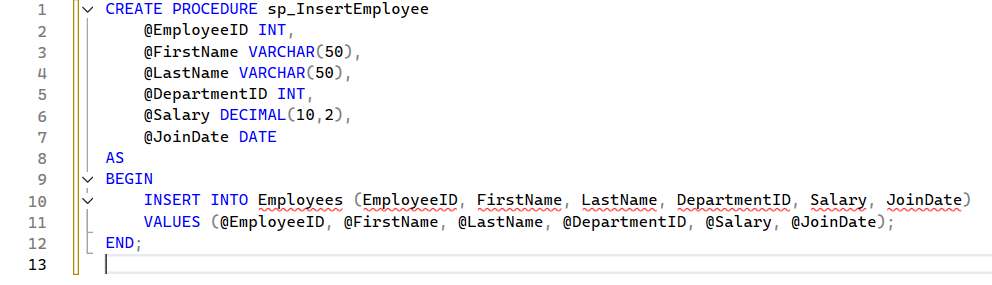
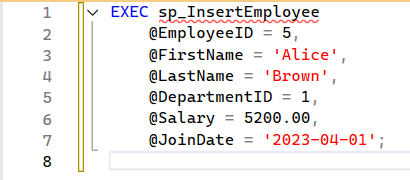
**Exercise : Ranking and Window Functions**

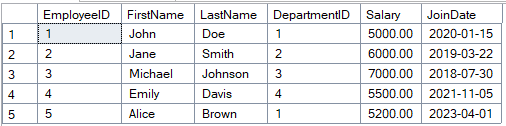
****

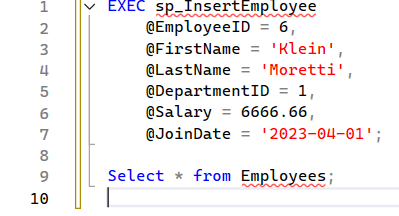
****

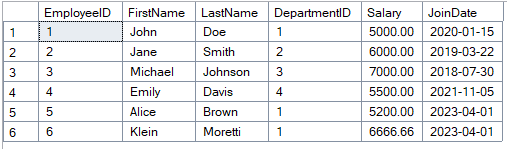
**Exercise : Create a Stored Procedure**

****

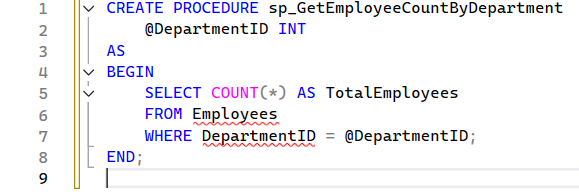
****

****

****

****

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

****

**Screenshot 2025-06-29 193751**

**Screenshot 2025-06-29 193757**

1. **NUnit-Handson**
2. *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.

1. *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.

1. *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.

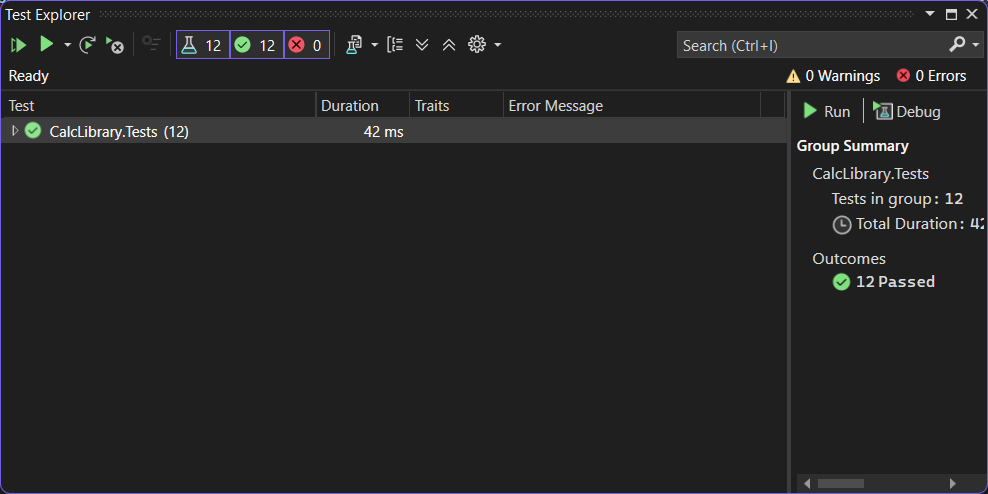
1. *What is a Loosely Coupled and Testable Design?*

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. |

