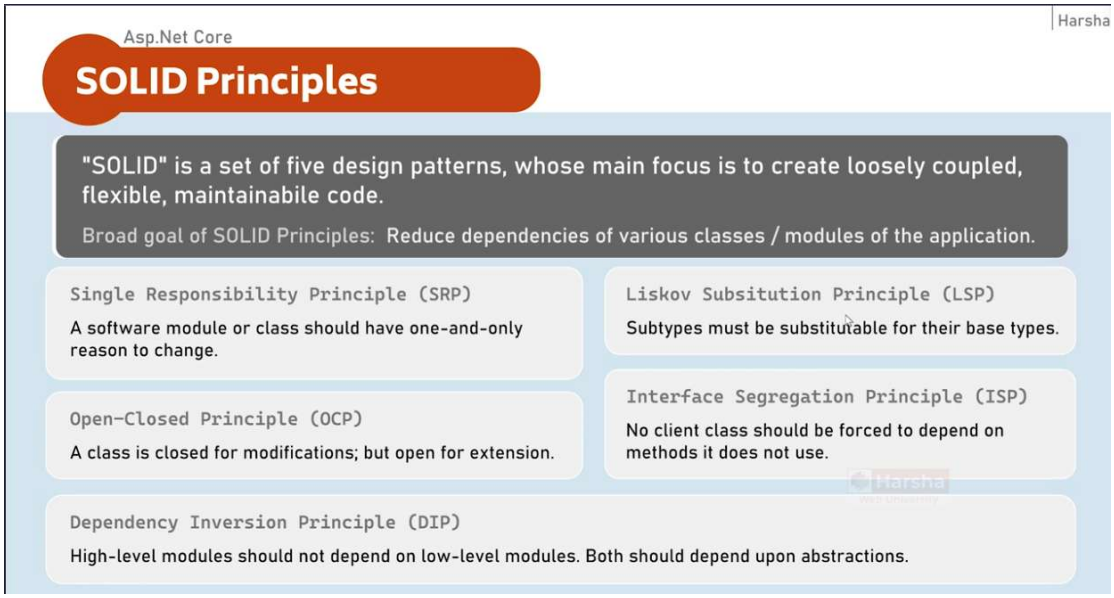
Section 23: SOLID Principles [MVC and Web API]

Monday, February 24, 2025 9:56 PM



SOLID Principles for Both Beginners and Experienced Developers

Learning **SOLID principles** is crucial for every developer. It is one of the most important topics, especially in **technical interviews**, where you may encounter at least one or two questions related to SOLID principles—both **theoretical and practical**.
In this section, we will **deep dive** into SOLID principles, covering their **concepts, benefits, and practical implementation**.

Introduction to SOLID Principles

SOLID principles are not specific to **ASP.NET Core**; rather, they are **general-purpose design principles** applicable to almost all popular programming languages, including **.NET, Java, and Python**.
The main goal of SOLID principles is to make classes **as independent as possible**.
For example, consider **Class 1 and Class 2**—there may be a scenario where **Class 1** needs to call **Class 2**.
However, both should be developed **independently** while maintaining **flexibility** so that changes in one do not negatively impact the functionality of the application.
By following **SOLID principles**, we ensure:

- ☑ **Maintainability** – Making changes in the codebase should not introduce new bugs.
- ☑ **Scalability** – New features can be added **without modifying** existing code.

☑ **Testability** – The system remains **unit-test friendly**, ensuring test cases stay relevant.

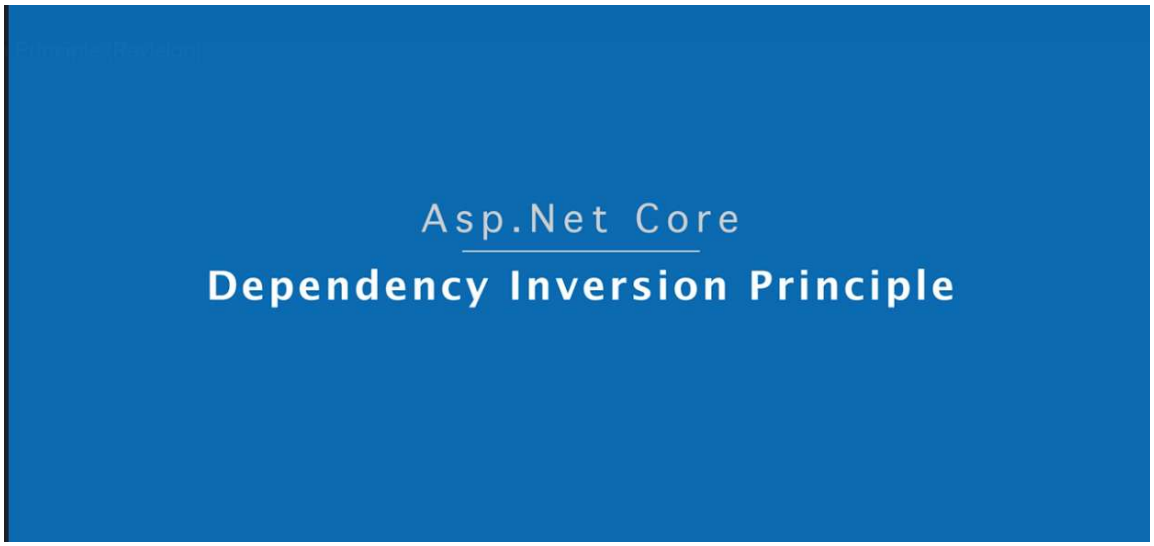
What Does SOLID Stand For?
SOLID is not just a word—it is an **acronym** for five core design principles:

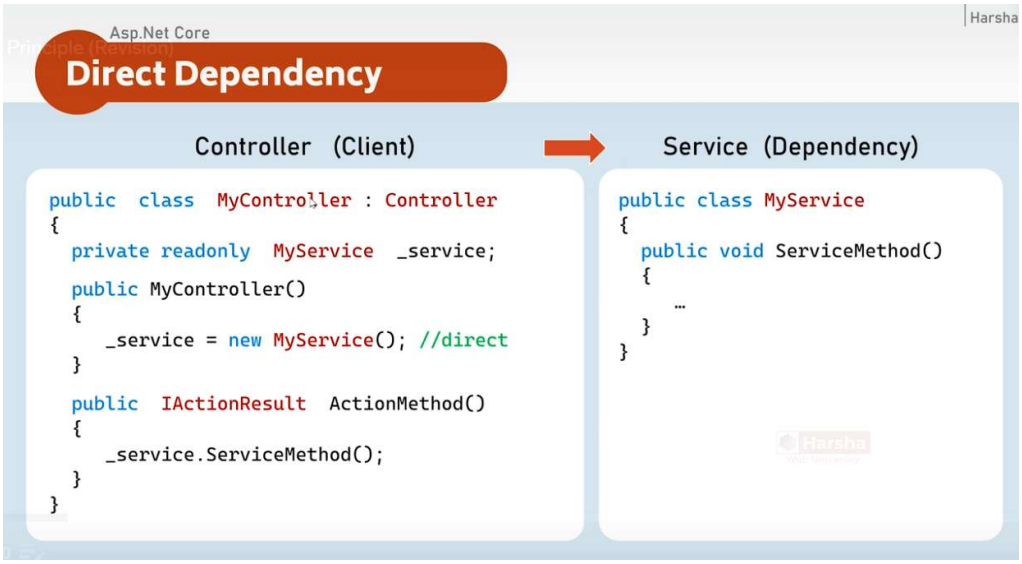
1. **S – Single Responsibility Principle (SRP)**
   - A class should have **only one reason to change**.
   - Each class should be responsible for **only one task**, avoiding multiple responsibilities.
2. **O – Open-Closed Principle (OCP)**
   - A class should be **open for extension** but **closed for modification**.
   - Instead of modifying existing code to add new functionality, you should **create a new class** and reuse the code using **inheritance or interfaces**.
3. **L – Liskov Substitution Principle (LSP)**
   - Subtypes should be **substitutable** for their base types **without altering** the expected behavior.
   - When overriding methods, the **method signature, return type, and exceptions** should match the base class.
4. **I – Interface Segregation Principle (ISP)**
   - Instead of creating **one large interface** with multiple methods, break it down into **smaller, specific interfaces**.
   - Example: Instead of one interface for **CRUD operations**, create separate interfaces for **Insert, Update, Delete**, etc.
5. **D – Dependency Inversion Principle (DIP)**
   - **High-level modules** should not depend on **low-level modules** directly.
   - Instead, both should depend on **abstractions** (interfaces) to ensure **loosely coupled** code.

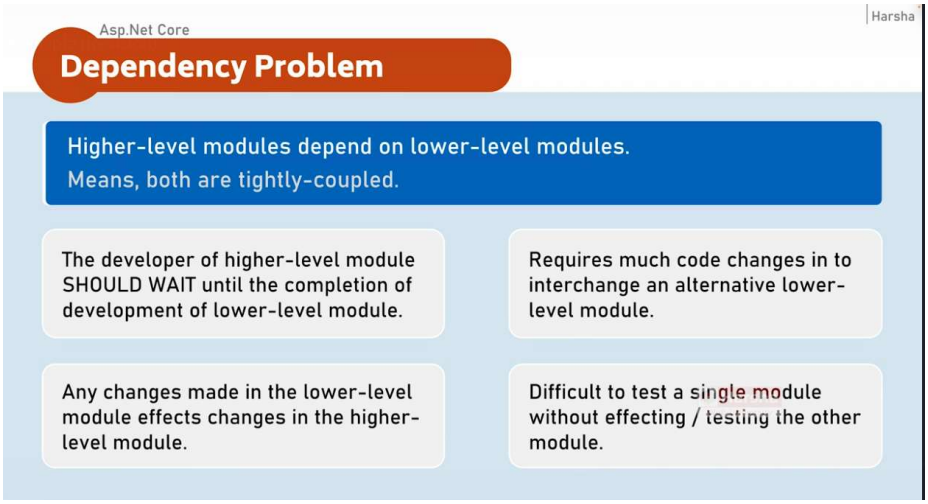
Practical Implementation
While this is just an **overview**, you might have questions like:

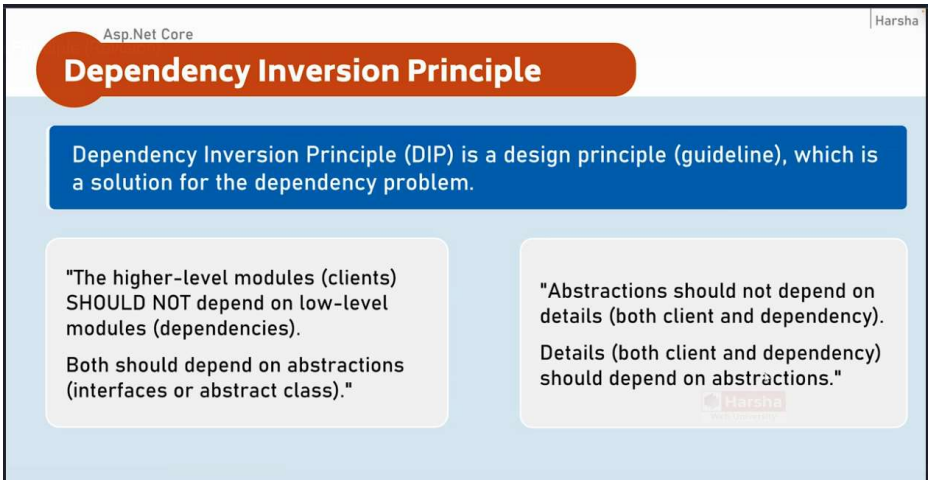
- What is the **real purpose** of each principle?
- **Where** should they be applied?
- **How** do we implement them in code?

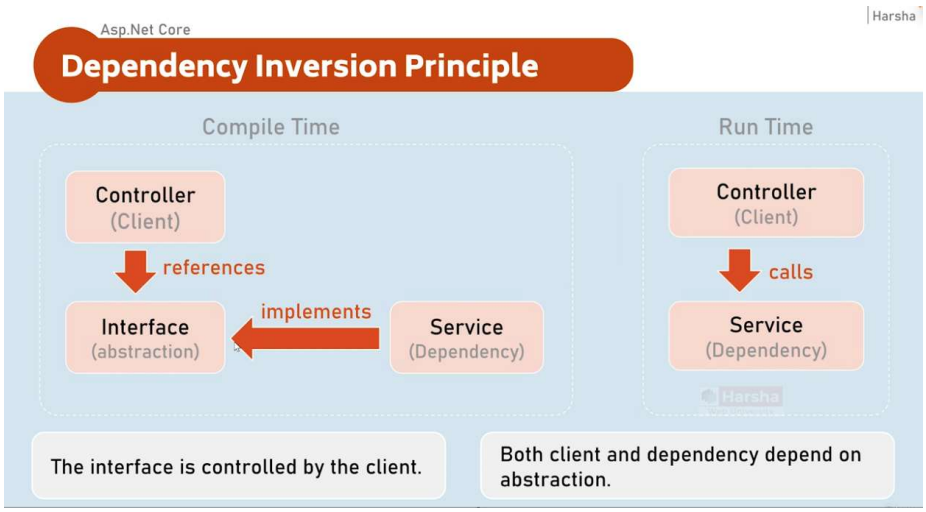
The answers to these questions will become clear as we **practically implement** each principle in the upcoming sections.
So, let's dive in and explore each **SOLID principle in detail!** 🚀

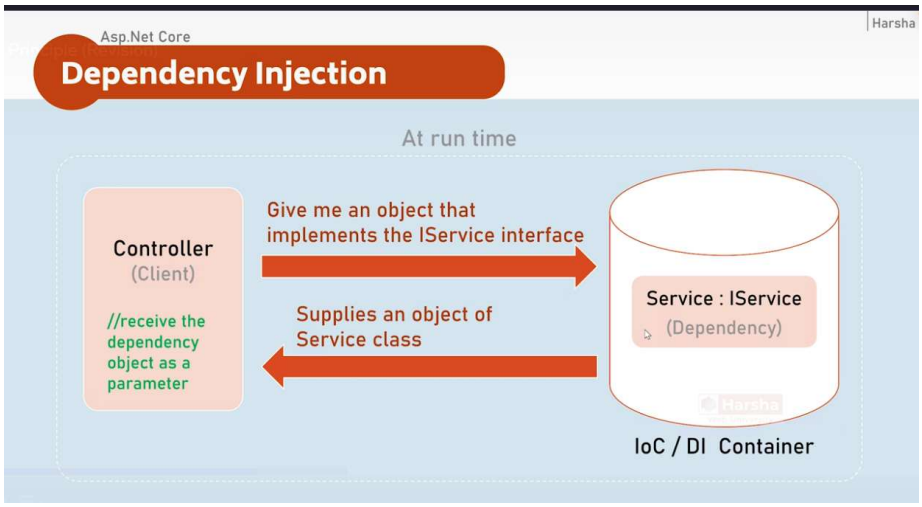


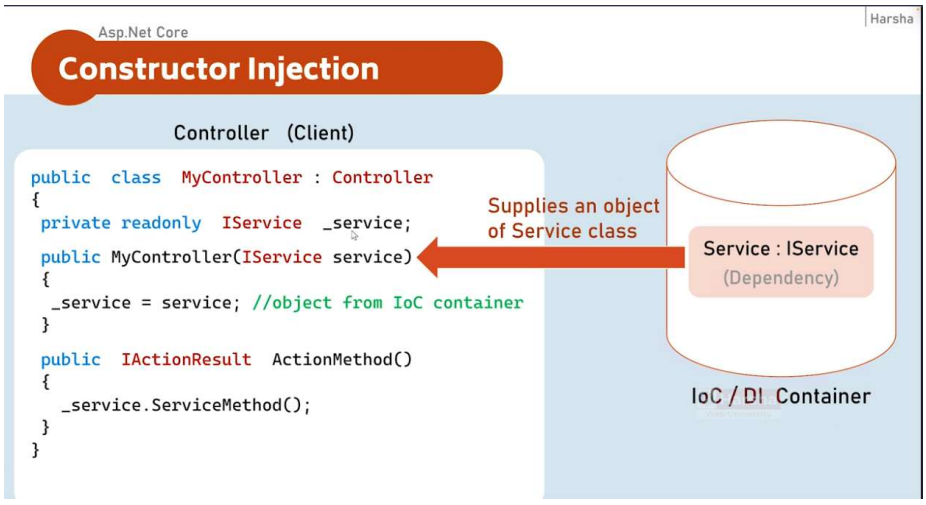

Harsha
Asp.Net Core
Direct Dependency
Controller (Client)
Service (Dependency)
public class MyController : Controller
{
private readonly MyService _service;
public MyController()
{
_service = new MyService(); //direct
}
public IActionResult ActionMethod()
{
_service.ServiceMethod();
}
}
public class MyService
{
public void ServiceMethod()
{
…
}
}

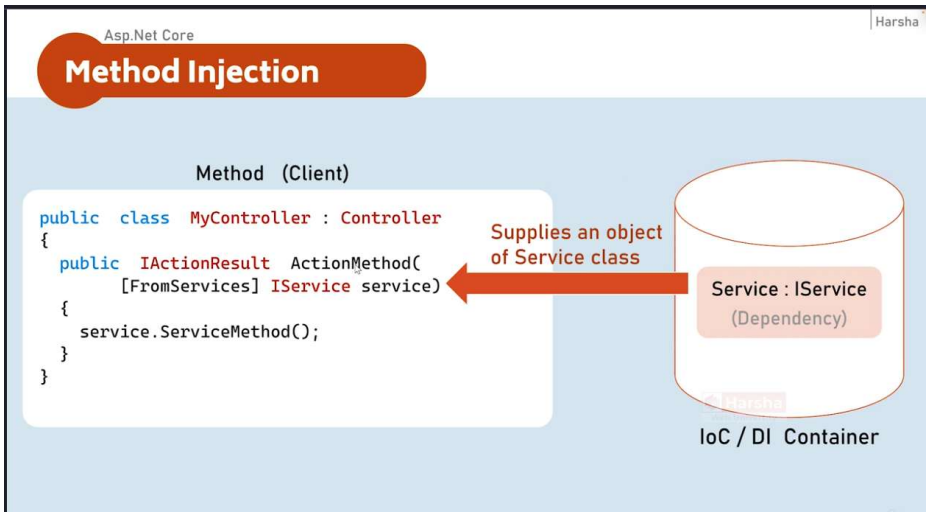

Harsha
Asp.Net Core
Dependency Problem
Higher-level modules depend on lower-level modules.
Means, both are tightly-coupled.
The developer of higher-level module SHOULD WAIT until the completion of development of lower-level module.
Requires much code changes in to interchange an alternative lower-level module.
Any changes made in the lower-level module effects changes in the higher-level module.
Difficult to test a single module without effecting / testing the other module.


Harsha
Asp.Net Core
Dependency Inversion Principle
Dependency Inversion Principle (DIP) is a design principle (guideline), which is a solution for the dependency problem.
"The higher-level modules (clients) SHOULD NOT depend on low-level modules (dependencies).
Both should depend on abstractions (interfaces or abstract class)."
"Abstractions should not depend on details (both client and dependency).
Details (both client and dependency) should depend on abstractions."


Harsha
Asp.Net Core
Dependency Inversion Principle
Compile Time
Run Time
Controller
(Client)
references
Interface
(abstraction)
implements
Service
(Dependency)
Controller
(Client)
calls
Service
(Dependency)
Harsha
The interface is controlled by the client.
Both client and dependency depend on abstraction.


Harsha
Asp.Net Core
Dependency Injection
At run time
Controller
(Client)
//receive the dependency object as a parameter
Give me an object that implements the IService interface
Supplies an object of Service class
Service : IService
(Dependency)
Harsha
IoC / DI Container


Harsha
Asp.Net Core
Constructor Injection
Controller (Client)
public class MyController : Controller
{
private readonly IService _service;
public MyController(IService service)
{
_service = service; //object from IoC container
}
public IActionResult ActionMethod()
{
_service.ServiceMethod();
}
}
Supplies an object of Service class
Service : IService
(Dependency)
IoC / DI Container



Recap: Dependency Inversion Principle (DIP)

We've already covered **Dependency Inversion Principle (DIP)** in detail in **Section 12**. This is a quick recap to reinforce the key concepts.

🔴 What Problem Does DIP Solve?

When a **class directly creates an object** of another class, it becomes **dependent** on that class. This creates a problem because:

1. **High-level modules depend on low-level modules**
   - If you modify a low-level module, the high-level module may **break**.
   - Developers working on high-level modules must **wait** for low-level module completion.
2. **Tightly coupled code**
   - Changes in one class **affect** others, making the system hard to maintain.

🛠 How Does DIP Solve This?

DIP states that:

- **High-level modules should not depend on low-level modules.**
- **Both should depend on abstractions (interfaces).**
- **Dependencies should be injected** rather than created inside classes.

◇ **Before DIP (Tightly Coupled)**

```
class Service {
   Repository repo = new Repository(); // Direct dependency
}
```

◇ **After DIP (Loosely Coupled)**

```
class Service {
   private readonly IRepository _repo;
public Service(IRepository repo) {
      _repo = repo;  // Dependency Injection
   }
}
```

🔄 Dependency Injection (DI) Techniques

DIP is often implemented using **Dependency Injection (DI)**. We covered:

✓ **Constructor Injection** – Injecting dependencies via the constructor.
✓ **Method Injection** – Passing dependencies as method parameters.

🚀 Benefits of DIP

☑ **Decouples modules** → Client and dependency can be developed **independently**.
☑ **Flexible & Scalable** → Easy to add new dependencies without modifying existing code.
☑ **Easier Testing** → Mocking dependencies in unit tests becomes simpler.

📌 Key Takeaway

DIP ensures that both **clients and dependencies depend on abstractions**, promoting **loose coupling** and **better maintainability**.
We've already implemented DIP practically in **Section 12**, so this recap should reinforce your understanding. 🎯🔥

```
});

//add services into IoC container
services.AddScoped<ICountriesRepository,
 CountriesRepository>();
services.AddScoped<IPersonsRepository,
 PersonsRepository>();

services.AddScoped<ICountriesService,
 CountriesService>();
services.AddScoped<IPersonsService,
 PersonsService>();
```

Asp.Net Core

Single Responsibility Principle

Harsha

Asp.Net Core

Single Responsibility Principle (SRP)

Class1

```
public class Class1
{
  //functionality1 & functionality2
}
```

should be re-written as

Class1

```
public class Class1
{
 //functionality1
}
```

Class2

```
public class Class2
{
 //functionality1
}
```

Asp.Net Core

Single Responsibility Principle (SRP)

A class should have one-and-only reason to change.

A class should implement only one functionality.

Avoid multiple / tightly coupled functionalities in a single class.

Benefit: Makes the class independent of other classes, in terms of its purpose / functionality.

So that, the classes become easier to design, write, debug, maintain and test.

Eg:

A class that performs validation should only involve in validation.

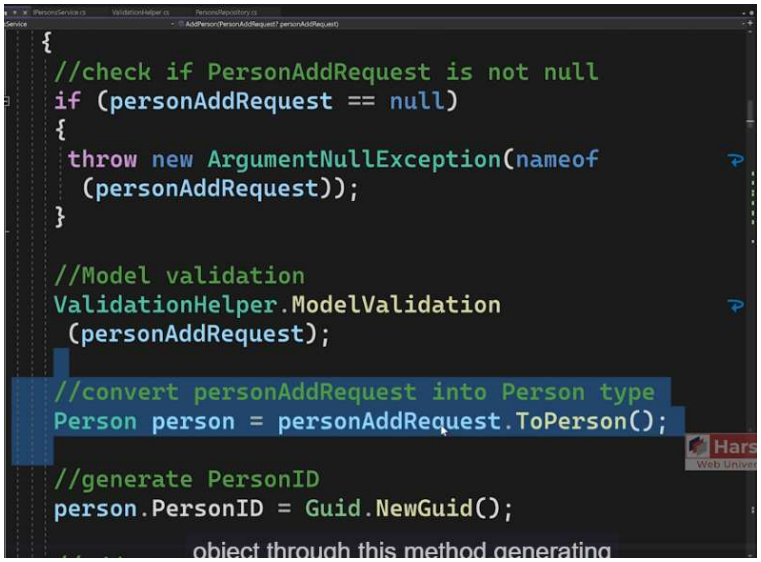
But it should not read configuration settings from a configuration file.

But instead, it call a method of another class that reads configuration settings.

(only validations)

(only reading configuration)

```
_personsRepository = personsRepository;
_logger = logger;
_diagnosticContext = diagnosticContext;
}

6 references
public async Task<PersonResponse> AddPerson
 (PersonAddRequest? personAddRequest)
{
 //check if PersonAddRequest is not null
 if (personAddRequest == null)
 {
  throw new ArgumentNullException(nameof
   (personAddRequest));
 }

 //Model validation
 ValidationHelper.ModelValidation
```



```
{
 //check if PersonAddRequest is not null
 if (personAddRequest == null)
 {
  throw new ArgumentNullException(nameof
   (personAddRequest));
 }

 //Model validation
 ValidationHelper.ModelValidation
  (personAddRequest);

 //convert personAddRequest into Person type
 Person person = personAddRequest.ToPerson();

 //generate PersonID
 person.PersonID = Guid.NewGuid();

 object through this method generating
```

```
43      }
44
45      //Model validation
46      ValidationHelper.ModelValidation
            (personAddRequest);
47
```

```
3   using System.ComponentModel.DataAnnotations;
4
5   namespace Services.Helpers
6   {
        2 references
7       public class ValidationHelper
8       {
            2 references
9           internal static void ModelValidation(object
                obj)
10          {
11              //Model validations
12              ValidationContext validationContext = new
                    ValidationContext(obj);
13              List<ValidationResult> validationResults =
                    new List<ValidationResult>();
14
15              bool isValid = Validator.TryValidateObject
                    (obj, validationContext, validationResults,
                    true);
```

class in the validation helper class

```
person.PersonID = Guid.NewGuid();

//add person object to persons list
await _personsRepository.AddPerson(person);
```

```
        7 references
73      public async Task<PersonResponse?>
            GetPersonByPersonID(Guid? personID)...
85
86
        5 references
87      public async Task<List<PersonResponse>>
            GetFilteredPersons(string searchBy, string?
            searchString)
88      {
89          _logger.LogInformation("GetFilteredPersons of
                PersonsService");
90
91          List<Person> persons;
92
93          using (Operation.Time("Time for Filtered Persons
                from Database"))
94          {
```

The Single Responsibility Principle says that one class should provide only a single functionality, not more than one.

For example, if a class implements both validation, retrieving data from the configuration, and data access from the database, it is implementing multiple functionalities, which is a bad practice. Instead, each individual functionality should be implemented as a separate class.

For example:
- One class for data access
- Another class for validation
- Another class for manipulating the cache
- Another class for manipulating cookies
- Another class for handling user requests

Each functionality should be separated into multiple classes or modules.

Why is implementing multiple functionalities in a single class a bad practice?

If we do so, not only does the code become cumbersome and difficult to debug, but modifying existing functionality requires multiple changes in the same class. As a result, we have to rewrite unit test cases, leading to regression testing. This means the existing unit test cases may or may not be relevant to the updated code.
Maintaining such code becomes difficult—bug identification, fixing, and distributing work across multiple teams all become problematic. To overcome this, the Single Responsibility Principle is recommended.

Formal Definition
A class should have only one reason to change. That means a class should implement only one functionality, not multiple tightly coupled functionalities.

Benefits

- The class becomes independent.
- Easier to understand, modify, design, write, maintain, and test.

Interview Explanation
If asked about the Single Responsibility Principle in an interview, you can explain it like this:
For example, if a class implements both reading configuration settings from a config file and model validation, doing both in a single class makes design, debugging, and bug fixing harder. To overcome this, separate the code into two classes:
- One class for validation
- Another class for configuration reading

Additional Consideration
One class can call another class's method when needed. However, the complete code of Class 2 should not be embedded in Class 1—just calling a method is acceptable.

For example, in an existing PersonService, before adding a new person to the database, we:
- Ensure the incoming argument is not null
- Convert it into a Person object
- Generate a new PersonID

This is business functionality. However, instead of writing data annotation validation directly in PersonService, we handle it separately in a ValidationHelper class. The validation method returns either success or an exception, which is handled by PersonService.
Similarly, instead of accessing the database directly, we use a separate repository class and call its method from the service. This is another implementation of the Single Responsibility Principle.

Another Example: Logging
In the GetFilteredPersons method, we call the LogInformation method of ILogger instead of writing logs directly to a file, database, or other log sources. Instead, logging is configured globally (e.g., with Serilog).

If logging were handled directly within the service, it would violate the Single Responsibility Principle. However, simply calling the appropriate logging methods is acceptable.

Conclusion
This is the essence of the Single Responsibility Principle. No code modifications are needed in this case, as our code already follows SRP.

Asp.Net Core

Interface Segregation Principle

Harsha

Asp.Net Core

Interfae Segregation Principle (ISP)

Interface1

```
public interface Interface1
{
  void Method1(); //performs one task
  void Method2(); //performs a different task
}
```

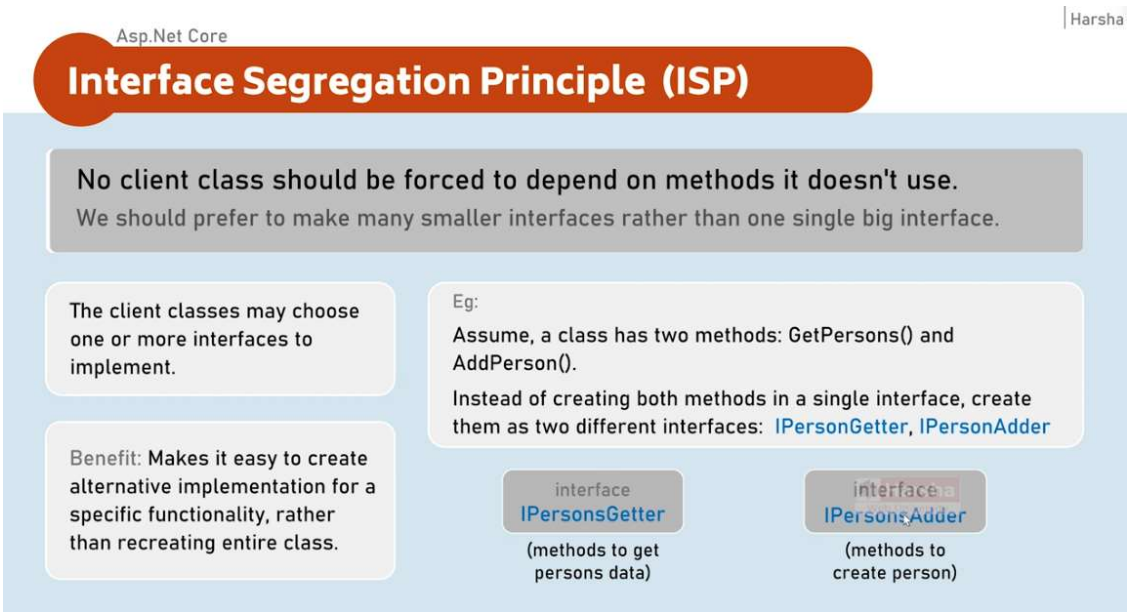
should be re-written as

Interface1

```
public interface Interface1
{
  void Method1();
}
```

Interface2

```
public interface Interface2
{
  void Method2();
}
```



According to the **Interface Segregation Principle**, instead of creating one large interface with many methods, you should group them into multiple interfaces with fewer methods.

Example

Consider an interface for a service that contains methods for all CRUD operations—Insert, Update, Delete, and other related methods. Instead of having a single interface for all operations, you should create:

- One interface for retrieving data
- Another interface for inserting data
- And so on...

Multiple smaller interfaces are better than one large, combined interface.

Formal Definition

No client should be forced to depend on methods it does not use.
This means that if a client class (e.g., a service class) implements an interface, it should not be required to implement unnecessary methods.

Problem with Large Interfaces

If an interface contains multiple methods, a client class must implement all of them, even if some are irrelevant. This leads to unnecessary code implementation.

Benefits of Smaller Interfaces

Breaking down a large interface into smaller interfaces allows for alternative implementations of specific methods.

Example Scenario
Let's say we need to implement two methods:

1. GetPersons()
2. AddPerson()

Instead of a single interface, we create two:

- IPersonsGetter (contains only GetPersons())
- IPersonsAdder (contains only AddPerson())

Now, consider an existing service implementing IPersonsGetter that returns data in JSON format. If the client later requests data in XML format with a different set of fields, we can create an **alternative service** that also implements IPersonsGetter but not IPersonsAdder.

Key Point

The new service **only implements IPersonsGetter**, not IPersonsAdder, because we only need an alternative for retrieval, not for adding.
This aligns with the Interface Segregation Principle: **The client class should not be forced to implement methods it doesn't need.**
For example, our service class does **not** need to implement AddPerson().

Dependency Injection Example

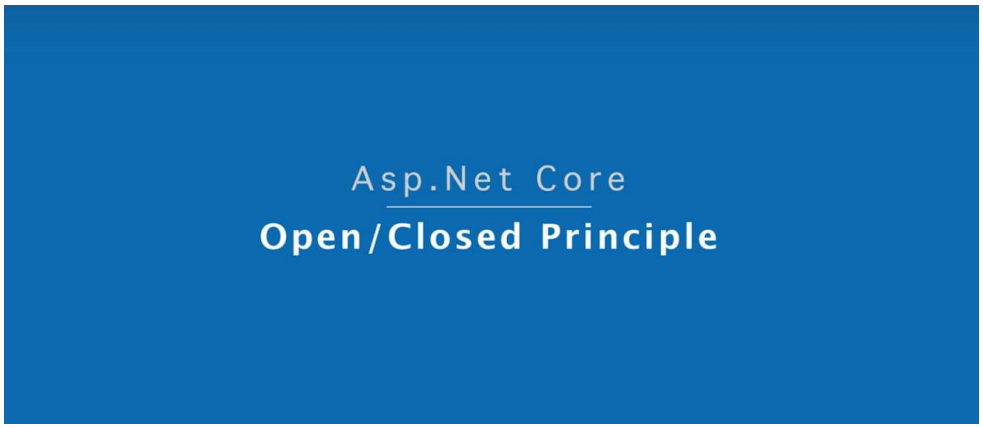
In a controller class, we inject only the necessary service:

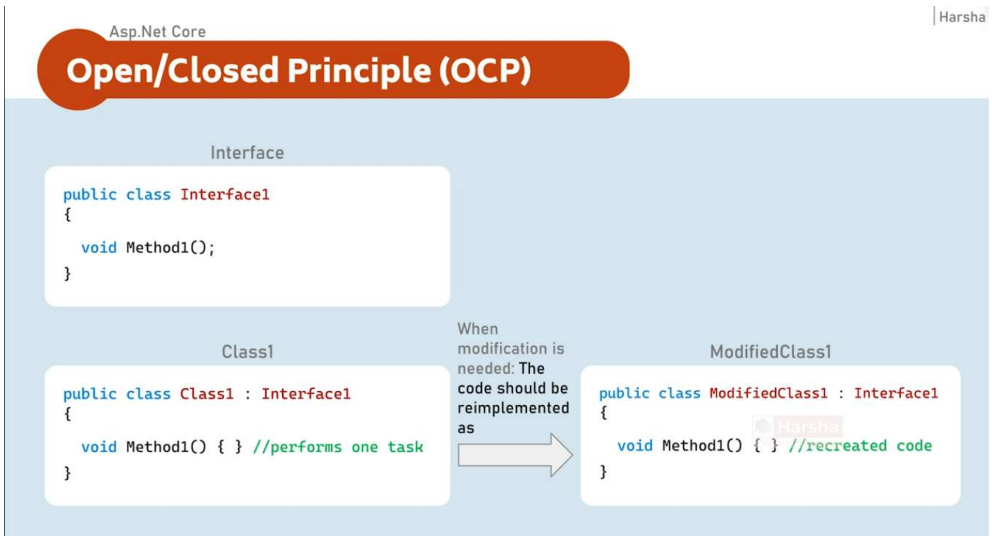
- If the controller **only retrieves data**, it injects a service that implements IPersonsGetter.
- If the controller **only adds data**, it injects a service that implements IPersonsAdder.
- If the controller **needs both**, it injects both interfaces.

This provides flexibility, allowing each client class to use only the methods it requires.

Issue in Our Existing Code

Unfortunately, in our current codebase, we **are not following** this principle.
Previously, we created a single interface, IPersonsService, containing all CRUD operation methods for the Persons table. Our approach was **one table, one service**.
However, **this violates the Interface Segregation Principle**. Instead of grouping methods by table, we should group them **by purpose or operation**.
To follow the principle correctly, we need to split our large interface into smaller, purpose-driven interfaces





Here's your explanation with improved clarity and structure:

OCP - Open/Closed Principle

The **Open/Closed Principle (OCP)** states that:

> **"Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification."**

This means that **you should not modify existing code** whenever there is a new requirement or change. Instead, **extend** the existing functionality by creating new implementations.

Why Follow OCP?

Modifying existing code can:

- Introduce **new bugs**
- Break **other dependent functionalities**
- Increase **maintenance complexity**

Instead of altering existing code, you should **create a separate implementation** that extends or overrides the behavior.

Example Scenario

Before Applying OCP

Suppose we have an **interface IInsertionService** with a method Insert(), which performs a database insertion:

```
public interface IInsertionService
{
   void Insert();
}
public class StandardInsertionService : IInsertionService
{
   public void Insert()
   {
      Console.WriteLine("Inserting data in the standard way...");
   }
}
```

Now, due to a **business requirement change**, we need to modify the way insertion happens. A **bad approach** would be **modifying the Insert() method** in StandardInsertionService.

Applying OCP (Good Approach)

Instead of changing the existing implementation, we create a **new class** with the modified behavior:

```
public class ModifiedInsertionService : IInsertionService
{
   public void Insert()
   {
      Console.WriteLine("Inserting data in a modified way...");
   }
}
```

Now, the **existing code remains untouched**, and we can use the new implementation where needed.

Practical Use Case: Dependency Injection

With **dependency injection**, we can dynamically use the required implementation:

```
public class DataProcessor
{
   private readonly IInsertionService _insertionService;
public DataProcessor(IInsertionService insertionService)
   {
      _insertionService = insertionService;
   }
public void ProcessData()
   {
      _insertionService.Insert();
   }
}
```

Now, we can decide **which insertion method to use** at runtime:

```
// Using standard insertion
var standardProcessor = new DataProcessor(new StandardInsertionService());
standardProcessor.ProcessData();
// Using modified insertion
var modifiedProcessor = new DataProcessor(new ModifiedInsertionService());
modifiedProcessor.ProcessData();
```

Key Takeaways

- ☑ **Existing code is not modified**, reducing risks of breaking functionality.
- ☑ **New requirements are handled through extension**, making code easier to maintain.
- ☑ **Flexible and scalable**, since we can introduce multiple variations without touching old code.

This is exactly what **OCP - Open/Closed Principle** suggests:

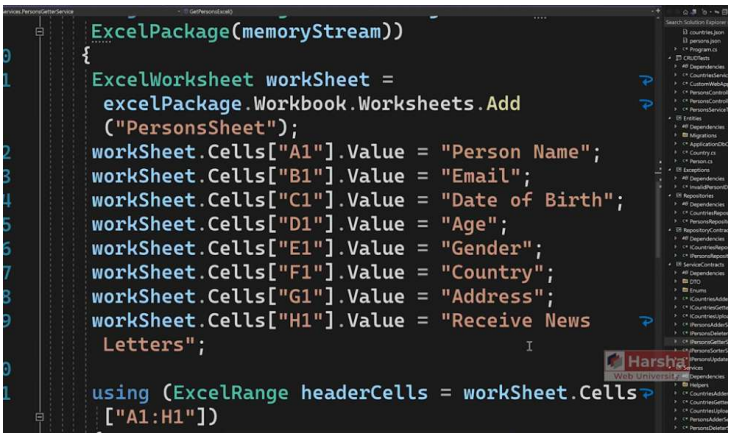
- ➔ **Open for extension** (you can create new implementations).
- ➔ **Closed for modification** (existing code remains unchanged).

Suppose, we are at 'IPersonsGetterService' interface.

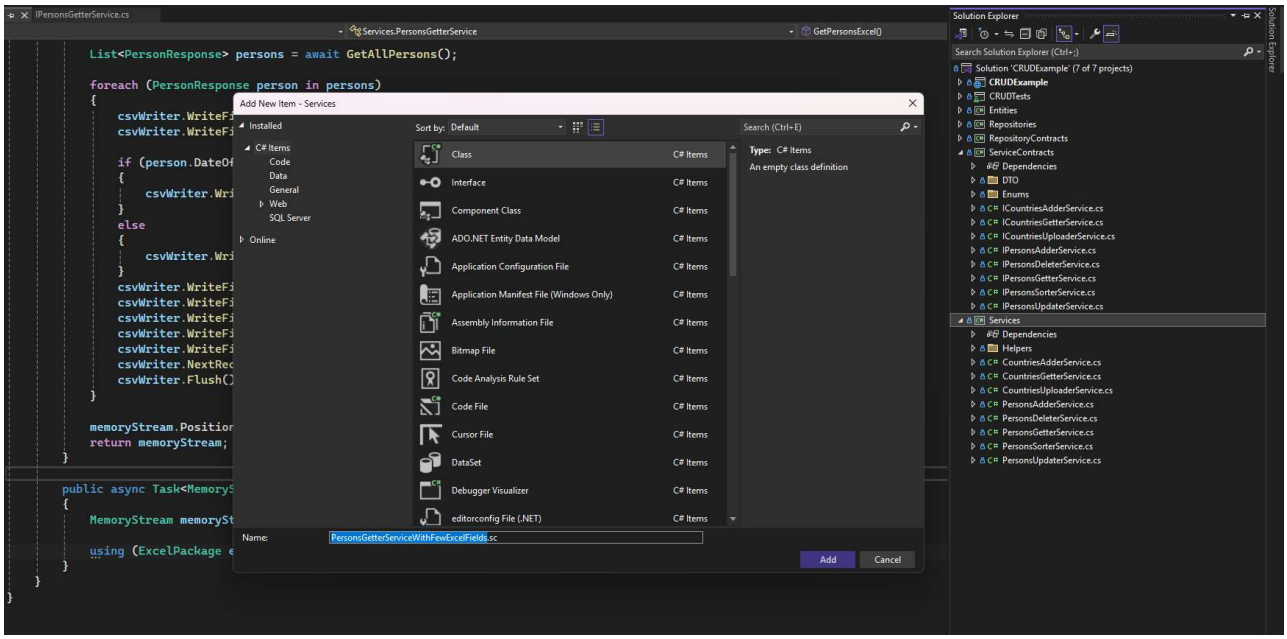
```
using ServiceContracts.DTO;
using ServiceContracts.Enums;
using System;

namespace ServiceContracts
{
    /// <summary>
    /// Represents business logic for manipulating Person entity
    /// </summary>
    public interface IPersonsGetterService
    {
        /// <summary>
        /// Returns all persons
        /// </summary>
        /// <returns>Returns a list of objects of PersonResponse type</returns>
        Task<List<PersonResponse>> GetAllPersons();

        /// <summary>
        /// Returns the person object based on the give person id
        /// </summary>
        /// <param name="guid">Person id to search</param>
        /// <returns>Returns maching person object</returns>
        Task<PersonResponse?> GetPersonByPersonId(Guid? personID);

        /// <summary>
        /// Returns all person objects that matches with the given search field and search string
        /// </summary>
        /// <param name="searchBy">Saerch field to search</param>
        /// <param name="searchString">Search string to search</param>
        /// <returns>Returns all matching persons based on the given search field and search string</returns>
        Task<List<PersonResponse>> GetFilteredPersons(string searchBy, string? searchString);

        /// <summary>
        /// Returns persons as CSV
        /// </summary>
        /// <returns>Returns the memory stream with CSV data</returns>
        Task<MemoryStream> GetPersonsCSV();
```

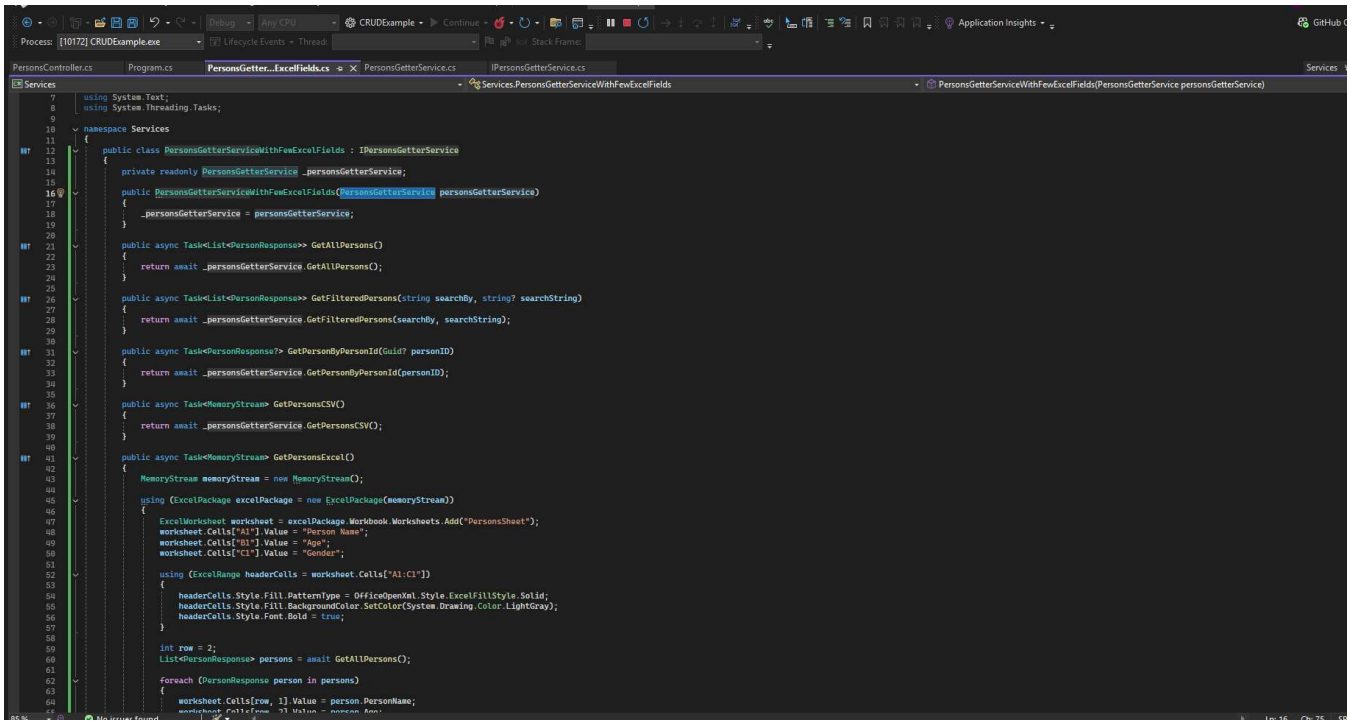
Suppose, there is something changes in the excel file generation.

Now the business requirement is that, client want to see few field. Not all the field. (PersonName, Age, Gender). But the probem is that if you made small changes in this existing code, it will lead to regression testing, some of the existing unit test case might not be relevant and it may introduce new bug.

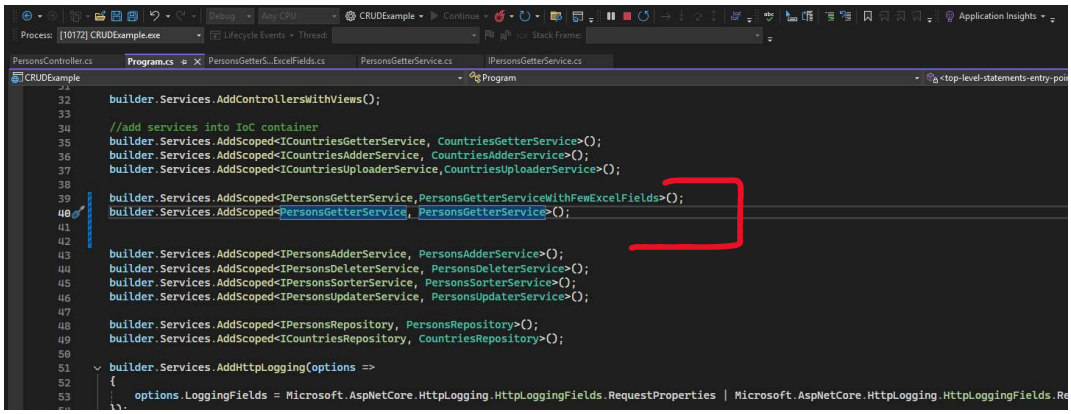


So making changes to the exiting code, leads to new bug.





```
using System.Text;
using System.Threading.Tasks;

namespace Services
{
    public class PersonsGetterServiceWithFewExcelFields : IPersonsGetterService
    {
        private readonly PersonsGetterService _personsGetterService;

        public PersonsGetterServiceWithFewExcelFields(PersonsGetterService personsGetterService)
        {
            _personsGetterService = personsGetterService;
        }

        public async Task<List<PersonResponse>> GetAllPersons()
        {
            return await _personsGetterService.GetAllPersons();
        }

        public async Task<List<PersonResponse>> GetFilteredPersons(string searchBy, string? searchString)
        {
            return await _personsGetterService.GetFilteredPersons(searchBy, searchString);
        }

        public async Task<PersonResponse?> GetPersonByPersonId(Guid? personID)
        {
            return await _personsGetterService.GetPersonByPersonId(personID);
        }

        public async Task<MemoryStream> GetPersonsCSV()
        {
            return await _personsGetterService.GetPersonsCSV();
        }

        public async Task<MemoryStream> GetPersonsExcel()
        {
            MemoryStream memoryStream = new MemoryStream();

            using (ExcelPackage excelPackage = new ExcelPackage(memoryStream))
            {
                ExcelWorksheet worksheet = excelPackage.Workbook.Worksheets.Add("PersonsSheet");
                worksheet.Cells["A1"].Value = "Person Name";
                worksheet.Cells["B1"].Value = "Age";
                worksheet.Cells["C1"].Value = "Gender";

                using (ExcelRange headerCells = worksheet.Cells["A1:C1"])
                {
                    headerCells.Style.Fill.PatternType = OfficeOpenXml.Style.ExcelFillStyle.Solid;
                    headerCells.Style.Fill.BackgroundColor.SetColor(System.Drawing.Color.LightGray);
                    headerCells.Style.Font.Bold = true;
                }

                int row = 2;
                List<PersonResponse> persons = await GetAllPersons();

                foreach (PersonResponse person in persons)
                {
                    worksheet.Cells[row, 1].Value = person.PersonName;
```

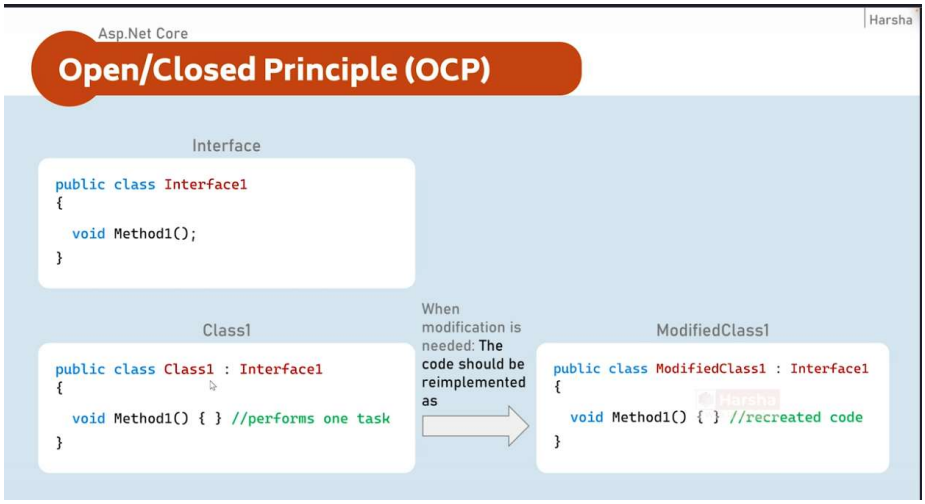


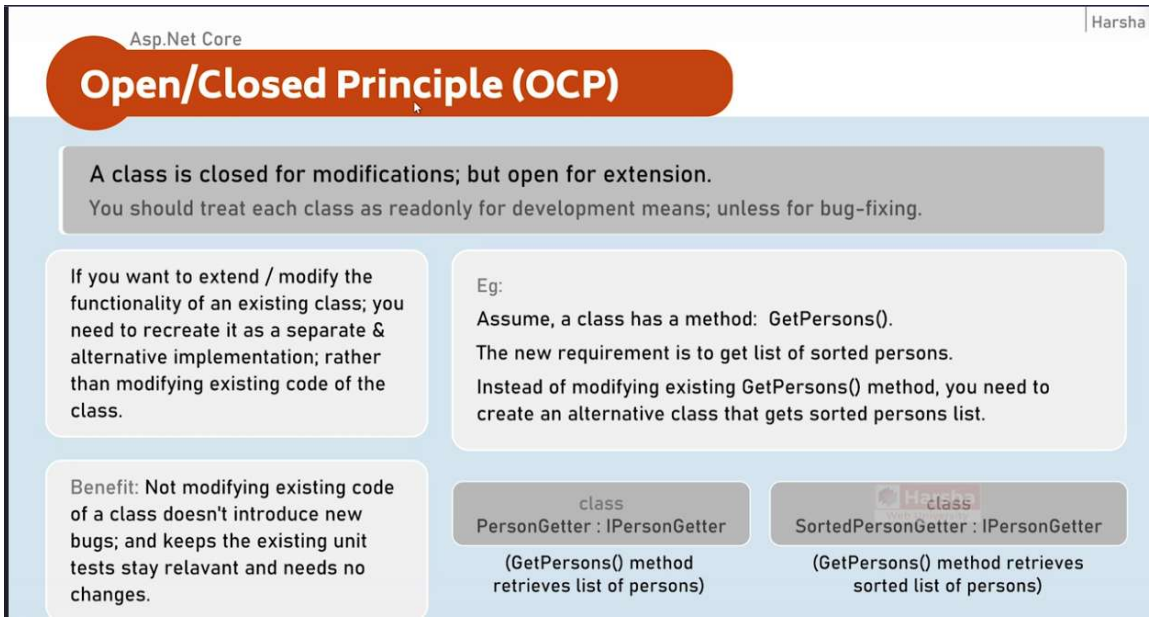
```
builder.Services.AddControllersWithViews();

//add services into IoC container
builder.Services.AddScoped<ICountriesGetterService, CountriesGetterService>();
builder.Services.AddScoped<ICountriesAdderService, CountriesAdderService>();
builder.Services.AddScoped<ICountriesUploaderService,CountriesUploaderService>();

builder.Services.AddScoped<IPersonsGetterService,PersonsGetterServiceWithFewExcelFields>();
builder.Services.AddScoped<PersonsGetterService, PersonsGetterService>();


builder.Services.AddScoped<IPersonsAdderService, PersonsAdderService>();
builder.Services.AddScoped<IPersonsDeleterService, PersonsDeleterService>();
builder.Services.AddScoped<IPersonsSorterService, PersonsSorterService>();
builder.Services.AddScoped<IPersonsUpdaterService, PersonsUpdaterService>();

builder.Services.AddScoped<IPersonsRepository, PersonsRepository>();
builder.Services.AddScoped<ICountriesRepository, CountriesRepository>();

builder.Services.AddHttpLogging(options =>
{
    options.LoggingFields = Microsoft.AspNetCore.HttpLogging.HttpLoggingFields.RequestProperties | Microsoft.AspNetCore.HttpLogging.HttpLoggingFields.Re
```

So what's the exact benefit of not modifying the exiting code?

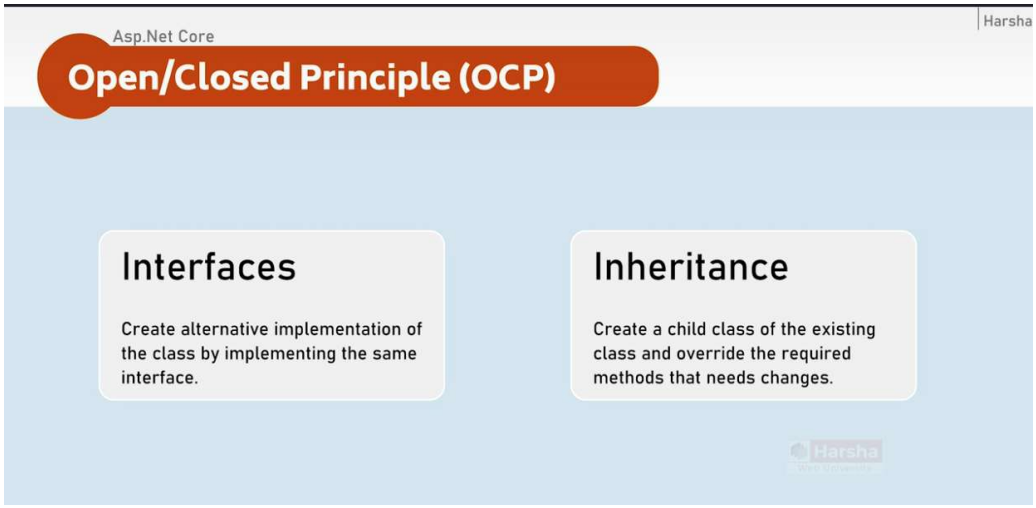
If you don't modify anything in the exiting code, all the unit test cases that are already written on this class 1 is still be relevant and valid. (needs no review or changes). Only the thing is you will create few more additional unit test cases on the newer code. It will save large amount of time in large scale production.

Also any other client classes such as any other controllers that are injecting exiting services still can consume the older existing functionality of older service.


Harsha
Asp.Net Core
Open/Closed Principle (OCP)
Interface
public class Interface1
{
void Method1();
}
Class1
public class Class1 : Interface1
{
void Method1() { } //performs one task
}
When modification is needed: The code should be reimplemented as
ModifiedClass1
public class ModifiedClass1 : Interface1
{
void Method1() { } //recreated code
}


Harsha
Asp.Net Core
Open/Closed Principle (OCP)
A class is closed for modifications; but open for extension.
You should treat each class as readonly for development means; unless for bug-fixing.
If you want to extend / modify the functionality of an existing class; you need to recreate it as a separate & alternative implementation; rather than modifying existing code of the class.
Eg:
Assume, a class has a method: GetPersons().
The new requirement is to get list of sorted persons.
Instead of modifying existing GetPersons() method, you need to create an alternative class that gets sorted persons list.
Benefit: Not modifying existing code of a class doesn't introduce new bugs; and keeps the existing unit tests stay relavant and needs no changes.
class PersonGetter : IPersonGetter
(GetPersons() method retrieves list of persons)
class SortedPersonGetter : IPersonGetter
(GetPersons() method retrieves sorted list of persons)


Asp.Net Core
OCP with Inheritance



OCP Implementation Using Inheritance
While **interfaces** are the preferred way to follow the **Open/Closed Principle (OCP)**, **inheritance** can also be used in some cases.

Method 2: Using Inheritance
Example Scenario
Let's say we have a **base class** InsertionService that provides a default Insert() method:

```
public class InsertionService
{
    public virtual void Insert()
    {
        Console.WriteLine("Inserting data in the standard way...");
    }
}
```

Now, due to a **business requirement change**, we need a modified insertion process.
Instead of **modifying the existing Insert() method**, we **extend** the base class and override the method in a derived class:

```
public class ModifiedInsertionService : InsertionService
{
    public override void Insert()
    {
        Console.WriteLine("Inserting data in a modified way...");
    }
}
```

Using Inheritance in Practice
Now, we can use **polymorphism** to decide which insertion method to use dynamically:

```
public class DataProcessor
{
    private readonly InsertionService _insertionService;
public DataProcessor(InsertionService insertionService)
    {
        _insertionService = insertionService;
    }
public void ProcessData()
    {
        _insertionService.Insert();
    }
}
```

Now, we can **choose the implementation at runtime**:

```
// Using standard insertion
var standardProcessor = new DataProcessor(new InsertionService());
standardProcessor.ProcessData();
// Using modified insertion
var modifiedProcessor = new DataProcessor(new ModifiedInsertionService());
modifiedProcessor.ProcessData();
```

Key Takeaways
☑ **Base class remains unchanged** → No risk of breaking existing functionality.
☑ **New behavior is implemented via subclassing** → No need to modify the original class.

☑ **Polymorphism allows runtime flexibility** → The correct implementation can be selected dynamically.

Comparison: Inheritance vs. Interfaces in OCP

Approach	Pros	Cons
Interfaces	☑ More flexible and loosely coupled	❌ Requires all implementations to be rewritten
Inheritance	☑ Simpler for similar behaviors	❌ More tightly coupled to the base class

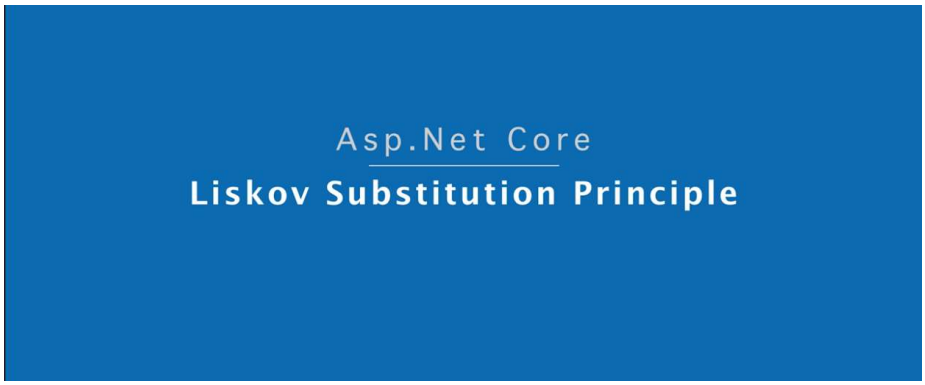
Which One to Use?

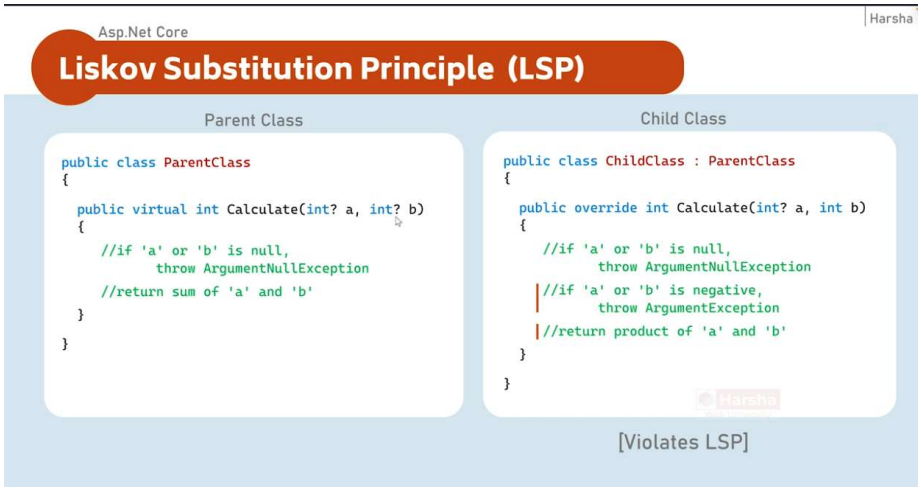
- **Prefer interfaces** in most cases (more flexibility, better for unit testing, and avoids tight coupling).
- **Use inheritance** if all implementations share a common base behavior and only need slight modifications.

Final Thought

Even though **inheritance can be used to follow OCP**, **interfaces** are still the better and more scalable option in modern applications.

Our Project Implementation:





In our project, we have created this class. If arguments are same, child class's method return different result which violated LSP.

```
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Services
{
    public class PersonsGetterServiceChild : PersonsGetterService
    {
        public PersonsGetterServiceChild(IPersonsRepository personsRepository, ILogger<PersonsGetterService> _logger, IDiagnosticContext _diagnosticCont
            _diagnosticContext)
        {

        }

        public async override Task<MemoryStream> GetPersonsExcel()
        {
            MemoryStream memoryStream = new MemoryStream();

            using (ExcelPackage excelPackage = new ExcelPackage(memoryStream))
            {
                ExcelWorksheet worksheet = excelPackage.Workbook.Worksheets.Add("PersonsSheet");
                worksheet.Cells["A1"].Value = "Person Name";
                worksheet.Cells["B1"].Value = "Age";
                worksheet.Cells["C1"].Val

                using (ExcelRange headerC
                {
                    headerCells.Style.Fill.PatternType = OfficeOpenXml.Style.ExcelFillStyle.Solid;
                    headerCells.Style.Fill.BackgroundColor.SetColor(System.Drawing.Color.LightGray);
                    headerCells.Style.Font.Bold = true;
                }

                int row = 2;
                List<PersonResponse> persons = await GetAllPersons();

                foreach (PersonResponse person in persons)
```

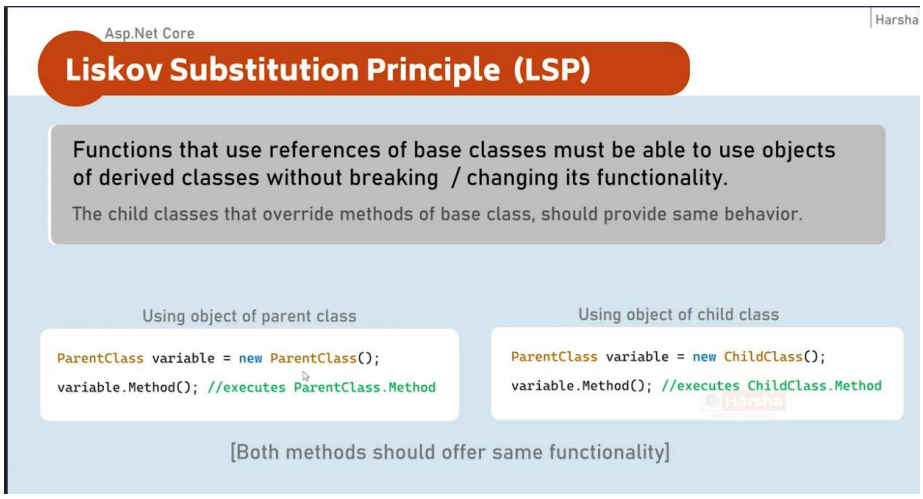
object ExcelRangeBase.Value { get; set; }
Set the range to a specific value

'Value' is not null here.

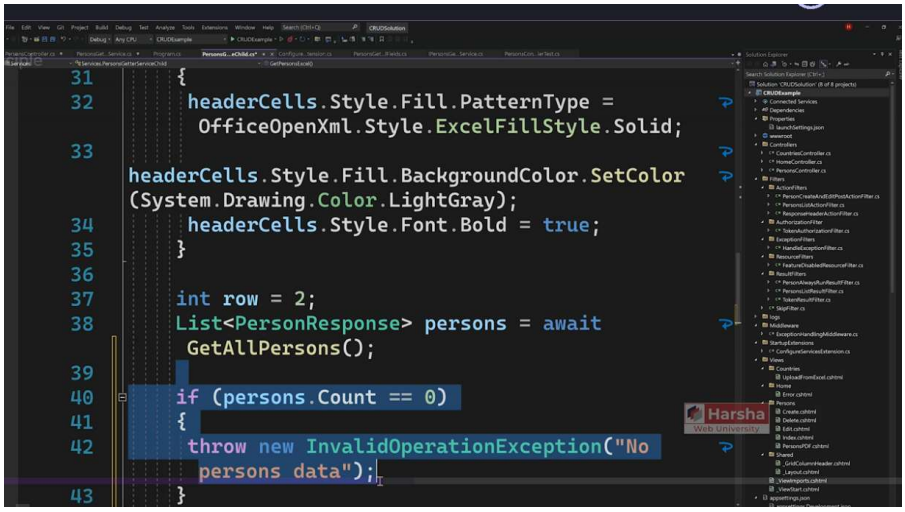
No issues found

```
        return memoryStream;
    }

    public virtual async Task<MemoryStream> GetPersonsExcel()
    {
        MemoryStream memoryStream = new MemoryStream();

        using (ExcelPackage excelPackage = new ExcelPackage(memoryStream))
        {
            ExcelWorksheet worksheet = excelPackage.Workbook.Worksheets.Add("PersonsSheet");
            worksheet.Cells["A1"].Value = "Person Name";
            worksheet.Cells["B1"].Value = "Email";
            worksheet.Cells["C1"].Value = "Date of Birth";
            worksheet.Cells["D1"].Value = "Age";
            worksheet.Cells["E1"].Value = "Gender";
            worksheet.Cells["F1"].Value = "Country";
            worksheet.Cells["G1"].Value = "Address";
            worksheet.Cells["H1"].Value = "Receive News Letters";

            using (ExcelRange headerCells = worksheet.Cells["A1:H1"])
            {
                headerCells.Style.Fill.PatternType = OfficeOpenXml.Style.ExcelFillStyle.Solid;
                headerCells.Style.Fill.BackgroundColor.SetColor(System.Drawing.Color.LightGray);
                headerCells.Style.Font.Bold = true;
            }

            int row = 2;
            List<PersonResponse> persons = await GetAllPersons();

            foreach (PersonResponse person in persons)
            {
                worksheet.Cells[row, 1].Value = person.PersonName;
                worksheet.Cells[row, 2].Value = person.Email;

                if (person.DateOfBirth.HasValue)
                {
                    worksheet.Cells[row, 3].Value = person.DateOfBirth.Value.ToString("yyyy-MM-dd");
```

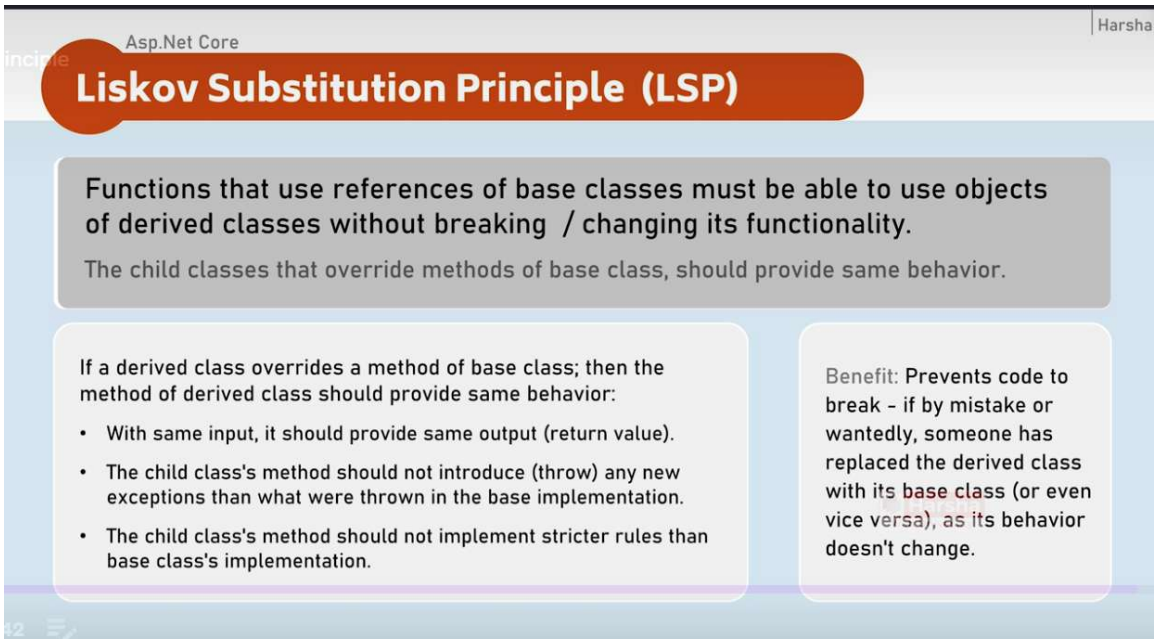
If you create object of Parent class, it will execute parent class method.
If you create child class, it will execute child class method.



Another way of violating LSP is that, in the child class method, we have introduced completely new exception.



You can identify if you are violating LSP or not in either of these three cases.



Liskov Substitution Principle (LSP) in SOLID

LSP states that **a subclass should be able to replace its parent class without affecting the correctness of the program**.
If a child class **modifies the behavior** of a parent class **in a way that breaks expectations**, it **violates LSP**.

Example of LSP Violation

Let's take an example where a **base class** defines a method for adding two numbers, but a **child class overrides it with a different operation (multiplication instead of addition)**.

```
public class Calculator
{
   public virtual int Calculate(int a, int b)
   {
      if (a == 0 || b == 0)
      {
         throw new ArgumentNullException("Inputs cannot be zero.");
      }
      return a + b; // Parent class performs addition
   }
}
public class AdvancedCalculator : Calculator
{
   public override int Calculate(int a, int b)
   {
      if (a < 0 || b < 0)  // New rule: No negative numbers
      {
         throw new ArgumentException("Inputs cannot be negative.");
      }
      return a * b; // Child class changes behavior to multiplication
   }
}
```

Why This Violates LSP?

1. **Different Behavior**
   - The **parent class** method performs **addition**.
   - The **child class** method performs **multiplication** instead.
   - If the system expects an addition operation, but gets multiplication, it leads to **unexpected behavior**.
2. **New Exception Handling**
   - The **parent class** throws ArgumentNullException for zero inputs.
   - The **child class** introduces a **new exception** (ArgumentException) for negative inputs.
   - This can cause unexpected crashes in existing code that was not designed to handle this exception.

Correct Implementation (Following LSP)

To **follow LSP**, the child class should **extend** behavior without modifying expected outcomes.

☑ **Solution: Use Composition Instead of Inheritance** If we need a modified behavior, we can use **composition** rather than **overriding methods incorrectly**.

```
public interface ICalculator
{
   int Calculate(int a, int b);
}
public class AdditionCalculator : ICalculator
{
   public int Calculate(int a, int b) => a + b;
}
public class MultiplicationCalculator : ICalculator
{
   public int Calculate(int a, int b) => a * b;
}
```

Now, instead of **violating LSP** through inheritance, we allow the **caller to choose the correct implementation**.

```
ICalculator calculator = new AdditionCalculator();
Console.WriteLine(calculator.Calculate(5, 10)); // Output: 15
calculator = new MultiplicationCalculator();
Console.WriteLine(calculator.Calculate(5, 10)); // Output: 50
```

Key Takeaways

☑ **Child classes should not alter expected behavior** → Maintain the same logic and expected return values.
☑ **Avoid introducing new exceptions** → A subclass should not introduce new failure conditions.
☑ **Prefer composition over inheritance** → Instead of forcing different behavior in subclasses, create separate implementations.

Final Thought

💡 **LSP ensures that code remains predictable and stable when using polymorphism.** If overriding a method **changes behavior in a way that existing code cannot handle**, it's better to **refactor the design** using **interfaces or composition**. 🚀