Value type vs Reference Type:

A **value type** is a data type in computer programming that directly stores the data's value, rather than a reference to the data's location in memory. Value types are typically small, simple data types, and they are characterized by the following properties:

**Immutability**: Value types are often immutable, meaning their values cannot be changed once they are created. Any operation that appears to modify a value type creates a new instance with the modified value.

**Value Semantics**: Equality comparisons for value types are based on their actual values rather than memory references. Two instances of the same value type with identical values are considered equal.

**Stack Allocation**: In some programming languages, value types are allocated on the stack rather than the heap. This can lead to efficient memory management.

Examples of value types include integers, floating-point numbers, characters, and structs in C#, among others.

**Reference Type**:

A **reference type** is a data type in computer programming that stores a reference or pointer to the memory location where the actual data is stored. Reference types are typically used for more complex, large, or mutable data structures and are characterized by the following properties:

**Reference Semantics**: Equality comparisons for reference types are based on memory references, not the actual values. Two reference variables are considered equal if they point to the same memory location.

**Heap Allocation**: Reference types are often allocated on the heap, which allows for more dynamic memory management and the ability to share data among multiple parts of a program.

**Mutability**: Reference types can be mutable, meaning their data can be modified directly through references.

Examples of reference types include classes, objects, arrays, and custom user-defined classes in many programming languages.

**Immutable:**

"Immutable" refers to the property of an object or data structure that cannot be modified after it is created. In other words, once an immutable object is created, its state cannot be changed. Any operation that appears to modify the object actually creates a new object with the desired changes, leaving the original object unchanged.

**Class vs Record**

The main difference between class and record type in C# is that a record has the main purpose of storing data, while a class defines responsibility. Records are immutable, while classes are not.

**When to use a record over a class?**

Use a record when an object's only purpose is to contain public data. On other hand, use a class if your object has unique logic. Classes are mutable so even if they have the same data, doesn't mean they are the same.

For example, if we think about a class that represents a window. Two windows on a house can look the same, and have the same size and color, but they are not the same:

2. Class vs Struct

In basic terms, a **struct** is a value type while a **class** is a reference type. Value types contain their data directly on the stack, while reference types store a reference to an object containing the data on the heap.

Structs are often used to represent simple data types, such as **integers**, **strings**, and other basic data types. Classes, on the other hand, are used to represent more complex objects with **multiple properties and methods**.

Another important difference between structs and classes is that structs are value types, which means that they are **copied** when they are passed as arguments to methods or functions.

Due to their memory allocation differences, structs are generally **faster** than classes. If you’re working with a large amount of data, structs can be more efficient because they don’t require the overhead of heap memory allocation.

However, there are some cases where classes are faster than structs. For example, when copying large objects, classes can be more efficient because they only copy a reference to the object instead of the object itself.

One common mistake when using structs is to make them too large. If your struct is too large, it will be stored on the heap instead of the stack, which can cause **performance issues**.

**3. Scoped vs Transient vs Singleton Service in C#**

**Explaining the Scoped Service in C#**

Have you ever felt like you’ve entered a unique space only to find out it already exists in another context? That’s how a scoped service behaves!

Imagine your application represents a hotel building. Your request is to get a hotel room – that’s your “scope”. Each scoped service is unique per request, like each hotel room being unique for each guest. A new instance is created for each scope. It creates one instance per request. (jotobar e use hok akta instance theke serve korbe per request)

*// Registering a Scoped Service*

services.AddScoped<IMyService, MyService>();

With this code, we register a service as scoped. Whenever a new request is made by the application, a new instance is created. Different requests? Different instances!

The beauty of Scoped service is in its ability to share resource within a request. Time-save? Heck, yes!

However, be warned: misusing scoped services can result in cross-request contamination and concurrency issues. Always handle with care!

Explaining the Transient Services in C#

Transient services are like butterflies, a different one with each glance! They create a new instance each time they’re requested, providing all classes with a unique copy. Let’s take a look at the registration code. It can create multiple instance per request.(jotobar lagbe toto ta instance create korbe)

*// Registering a Transient Service*

services.AddTransient<IMyService, MyService>();

Each call to GetService<IMyService> gets a fresh instance like receiving a new gadget each time you ask!

Transient services lend themselves to lightweight, stateless services yet careful consideration is required. Overuse can lead to increased memory usage. Note to self: More isn’t always merry!

**Explaining the Singleton Service in C#**

One ring to rule them all, one ring to bind them! If services were rings, Singleton would be the One ring. It creates a single instance for the whole application. Registered once, used everywhere!

*// Registering a Singleton Service*

services.AddSingleton<IMyService, MyService>();

Order up a Singleton service, and you’re provided an instance. Make a second request and you’ll get the same instance. It’s persistent!

Singleton services are great for sharing states across requests and even connections. Exercise caution though, singletons don’t come without tripwires. Remember, because they hang around for longer than scoped or transient, you might find them hoarding resources. It creates one instance in one life cycle.

**Scoped Services**

Think of scoped services as your personal assistant during a trip. It’s there when you embark, serves all your needs during journey, and concludes its role once the journey ends. It retains information through a specific scope/request, but this doesn’t persist across different scopes.

Scoped services are beneficial when multiple objects in a single scope need to communicate or share data.

**Transient Services**

Transient services, in contrast, are more like hitchhikers on separate journeys. They join you momentarily before disembarking to join another journey. They don’t retain any memory or state from previous journeys.

If you need a service in separate classes which doesn’t retain data and doesn’t need to communicate with each other, transient services are your answer.

**Singleton Services**

Singleton services are your loyal companions. Initiated once, they stick around throughout the application lifetime.

Singletons are useful if you need to maintain stateful information that persists through different requests in the same application session.

**When to Use Each Service**

**Scoped Service**: Best suited when you want to maintain state within a single request, but not persistently. This is also ideal when you need shared communication/data-access within object instances of a single request.

**Transient Service**: Optimal choice for lightweight, stateless services that are implemented throughout your application without needing integration or communication. These instances do not remember their previous state – like state amnesia!

**Singleton Service**: Preferable for dealing with data or state that needs to be shared across multiple requests, or when the instantiation process is expensive. Keep an eye on state management and thread-safety when using singleton services.

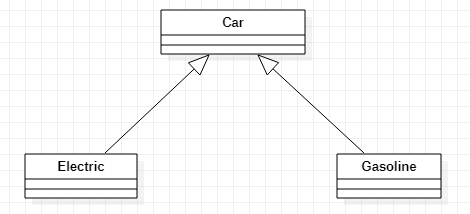
4. **Why no multiple inheritance**

C# does not support multiple inheritance of classes, meaning a class cannot directly inherit from more than one class at the same time. This limitation was introduced for several reasons, including reducing complexity, avoiding the "diamond problem" (a naming conflict that arises with multiple inheritance), and promoting better code organization through interfaces.

# 5. What is a diamond problem in Object-Oriented Programming?

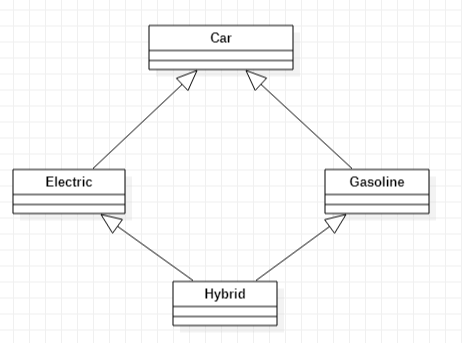
## Example

Consider the example of a car.



Both electric and gasoline cars inherit the properties of a car.

Now, there is a special case if there is another class – a Hybrid class, for example – that inherits both the Electric and Gasoline class. We can see in the diagram that it will form a diamond.



The hybrid car is both an electric car and a gasoline car. These kinds of special cases will result in a **diamond problem**.

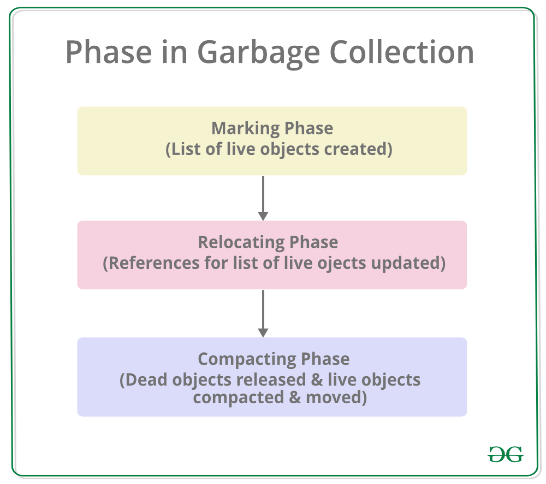
This diamond creates a problem, because now the Hybrid class has two copies of the Car class for each path.

6. **Garbage Collection in c#:**

Garbage collection is a memory management technique used in the .NET Framework and many other programming languages. In C#, the garbage collector is responsible for managing memory and automatically freeing up memory that is no longer being used by the application.

The garbage collector works by periodically scanning the application’s memory to determine which objects are still being used and which are no longer needed. Objects that are no longer being used are marked for garbage collection, and their memory is freed up automatically by the garbage collector.

There are mainly **3** phases in garbage collection. Details about these are given as follows:



𝐈𝐄𝐧𝐮𝐦𝐞𝐫𝐚𝐛𝐥𝐞 vs 𝐈𝐐𝐮𝐞𝐫𝐲𝐚𝐛𝐥𝐞

𝐈𝐄𝐧𝐮𝐦𝐞𝐫𝐚𝐛𝐥𝐞 serves as a fundamental interface suitable for in-memory collections, offering lazy loading and executing queries in-memory through LINQ extension methods. Conversely, 𝐈𝐐𝐮𝐞𝐫𝐲𝐚𝐛𝐥𝐞 is a generic interface that extends 𝐈𝐄𝐧𝐮𝐦𝐞𝐫𝐚𝐛𝐥𝐞 and is tailored for querying external data sources. It supports deferred execution, translating queries into a format understandable by the underlying data source, such as a database, thereby optimizing performance for large datasets.

The key differences lie in their execution strategies, where 𝐈𝐄𝐧𝐮𝐦𝐞𝐫𝐚𝐛𝐥𝐞 processes queries in-memory, and 𝐈𝐐𝐮𝐞𝐫𝐲𝐚𝐛𝐥𝐞 defers execution and performs server-side processing, making it more suitable for scenarios involving significant data volumes and external data sources. The choice between them hinges on whether the data resides in-memory or originates from an external source, influencing the trade-offs between flexibility and performance.

7. **The CQRS Pattern**

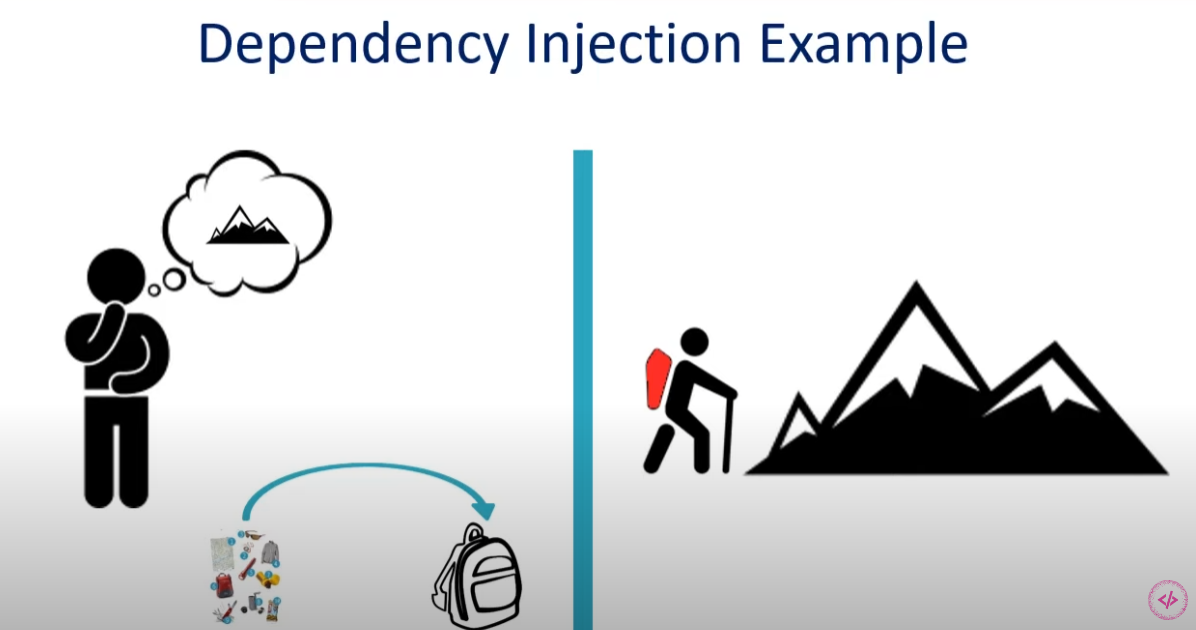
The Command and Query Responsibility Segregation (CQRS) pattern states that we must separate the operations for reading the data from the operations for writing or updating the data. This means that functions for reading and writing data are not kept in the same interface or class. The main advantages of doing this include:

1. Separate teams can work on these operations
2. Each can be made to scale according to their own needs. Write operations are mostly used much less than read operations
3. Each can have their own security as per requirements
4. Read operations can have a different architecture to support caching, conversions to data transformation objects as required by clients
5. Write operations can include data validation. lookups etc.

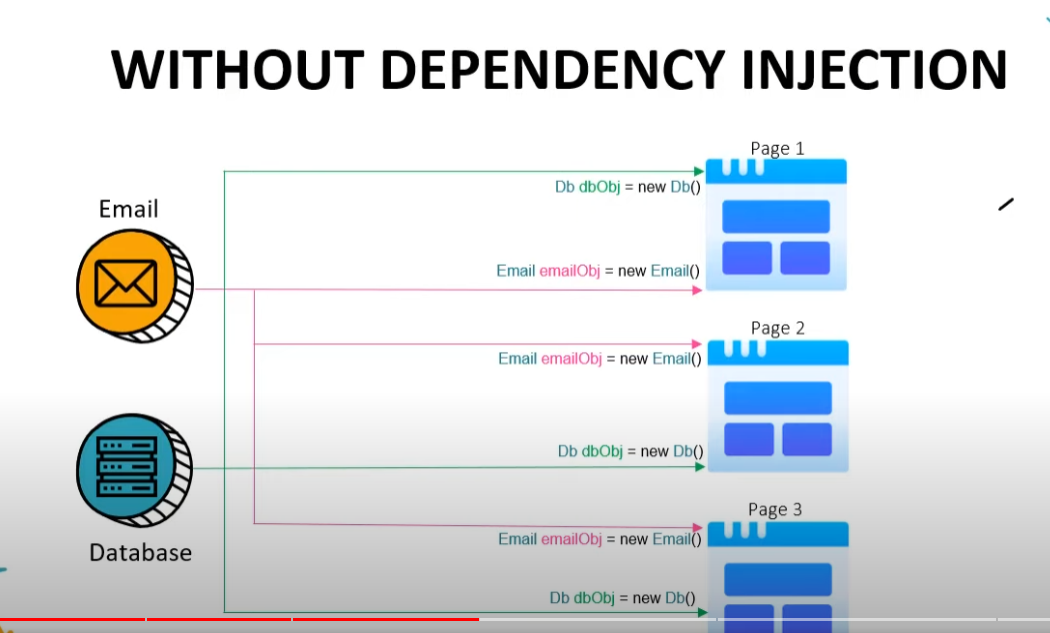
However, do keep in mind that this pattern is better suited to larger applications where the requirements and load levels between read and write operations are different. For a simple and small application, the normal CRUD pattern, often auto-generated from ORM tools, is sufficient.

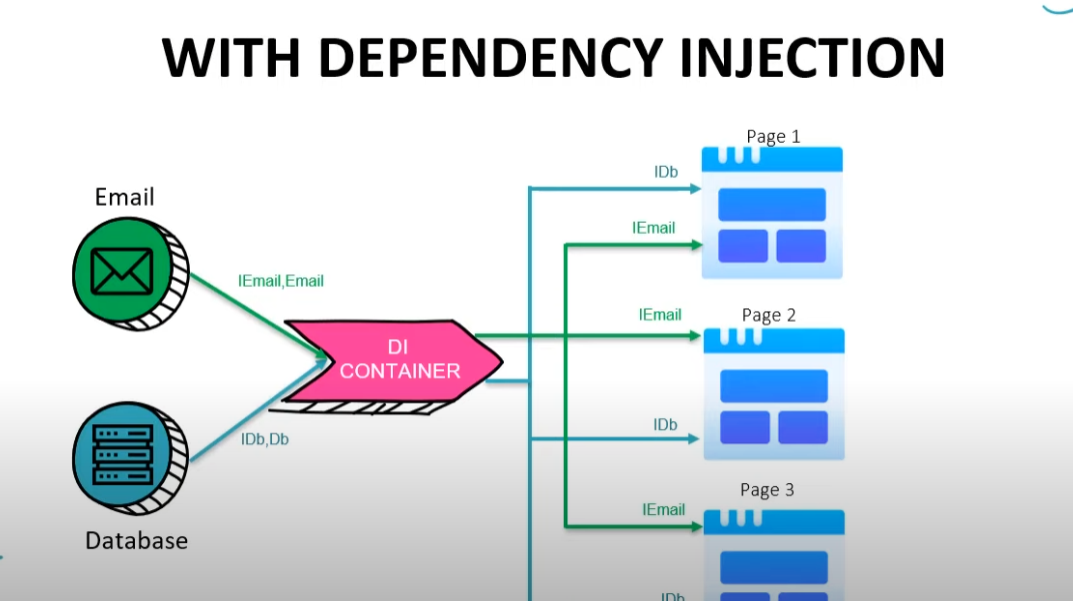
**8. DI(Dependency Injection ):**

Dependency Injection (DI) in .NET is a design pattern and a technique that facilitates the development of loosely coupled, maintainable, and testable software.

****

Suppose you want to hike a mountain. You need some tools like flash light, water, dry food etc. So, you put these things in a bag and carry the bag while hiking. Now when you need water you can get the water from the bag and then again put the water in the bag. When you are hungry you can get the dry food from the bag. You don’t need to get the things out of the bag all at a time. Here the bag is act like a container.





Here are some of the key advantages of using dependency injection:

**Loose Coupling:**

Reduced Dependency on Concrete Implementations: Dependency Injection helps in reducing the direct dependencies between components. Components depend on abstractions (interfaces or abstract classes) rather than concrete implementations, promoting loose coupling.

Testability:

Easier Code Maintenance: With DI, it's easier to maintain and extend code. Adding new features or changing existing ones often requires modifying only the necessary components, and the changes are less likely to have ripple effects throughout the application.

Flexibility and Reusability:

Swappable Implementations: DI allows you to easily swap implementations of dependencies without modifying the dependent components. This is especially useful for supporting different configurations or for using different implementations in various parts of the application.

Readability:

Improved Code Readability: Dependency Injection can improve code readability by explicitly declaring dependencies in the constructor or method parameters. It makes it clear what a component relies on and what is required for it to function properly.

Separation of Concerns:

Enhanced Separation of Concerns (SoC): DI encourages a better separation of concerns by ensuring that each component has a well-defined responsibility. This separation makes the codebase more modular and easier to understand.

Scalability:

Easier Scalability: As the application grows, DI makes it easier to manage and scale the codebase. New components can be added without significantly impacting existing ones, and the overall architecture can remain more maintainable.

Lifecycle Management:

There are 3 types of Dependency Injection.

* Constructor Injection
* Property Injection
* Method Injection

10.

**Difference between Abstract Class and Interface**

| **Abstract Class** | **Interface** |
| --- | --- |
| **It contains both declaration and implementation parts.** | **It contains only the declaration of methods, properties, events, or indexers. Since C# 8,**[default implementations](https://devblogs.microsoft.com/dotnet/default-implementations-in-interfaces/)**can also be included in interfaces.** |
| **Multiple inheritance is not achieved by abstract class.** | **Multiple inheritance is achieved by interface.** |
| **It contain**[constructor](https://www.geeksforgeeks.org/c-sharp-constructors/)**.** | **It does not contain**[constructor](https://www.geeksforgeeks.org/c-sharp-constructors/)**.** |
| **It can contain static members.** | **It does not contain static members.** |
| **It can contain different types of access modifiers like public, private, protected etc.** | **It only contains public access modifier because everything in the interface is public.** |
| **The performance of an abstract class is fast.** | **The performance of interface is slow because it requires time to search actual method in the corresponding class.** |
| **It is used to implement the core identity of class.** | **It is used to implement peripheral abilities of class.** |
| **A class can only use one abstract class.** | **A class can use multiple interface.** |
| **If many implementations are of the same kind and use common behavior, then it is superior to use abstract class.** | **If many implementations only share methods, then it is superior to use Interface.** |
| **Abstract class can contain methods, fields, constants, etc.** | **Interface can only contains methods, properties, indexers, events.** |
| **The keyword “:” can be used for implementing the Abstract class.** | **The keyword “:” and “,” can be used for implementing the Interface.** |
| **It can be fully, partially or not implemented.** | **It should be fully implemented.** |

11. **IOC(Inversion of Control)**

Suppose you are an object. And you go to a restaurant:

**Without IoC**: you ask for "apple", and you are always served apple when you ask more.

**With IoC**: You can ask for "fruit". You can get different fruits each time you get served. for example, apple, orange, or water melon.

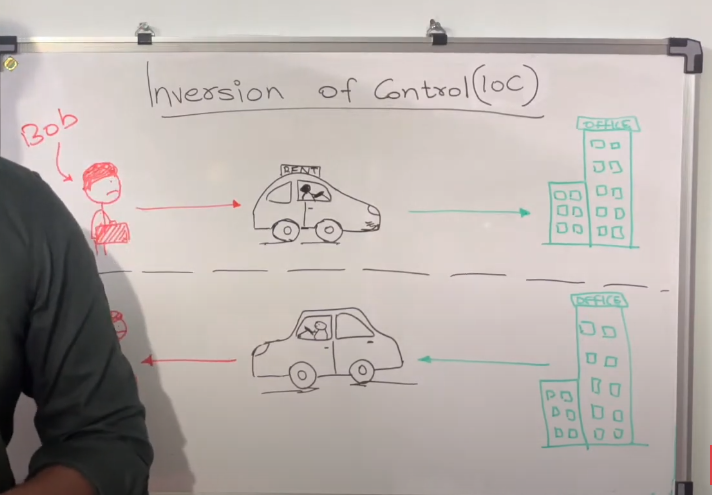
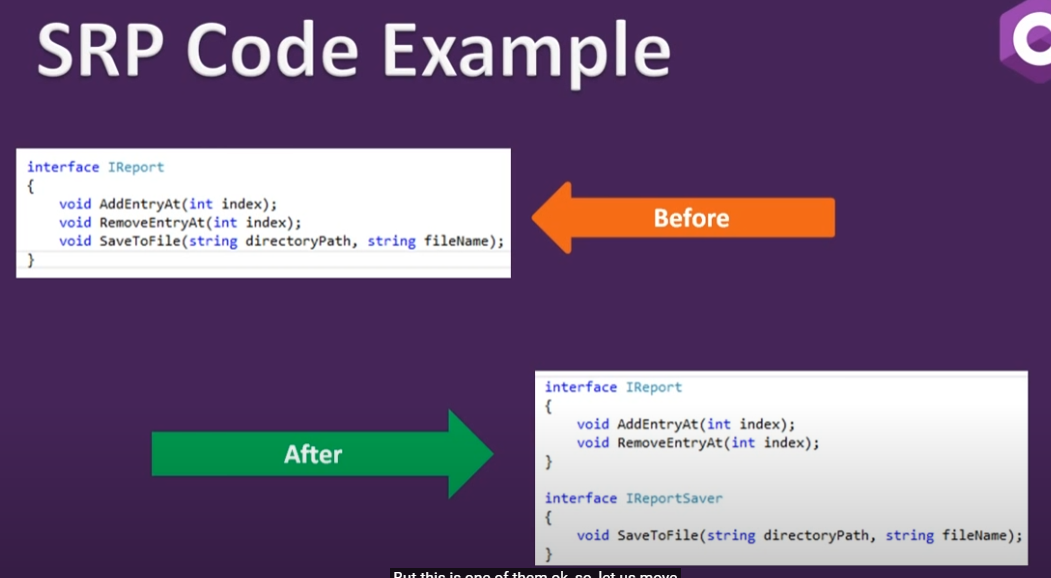
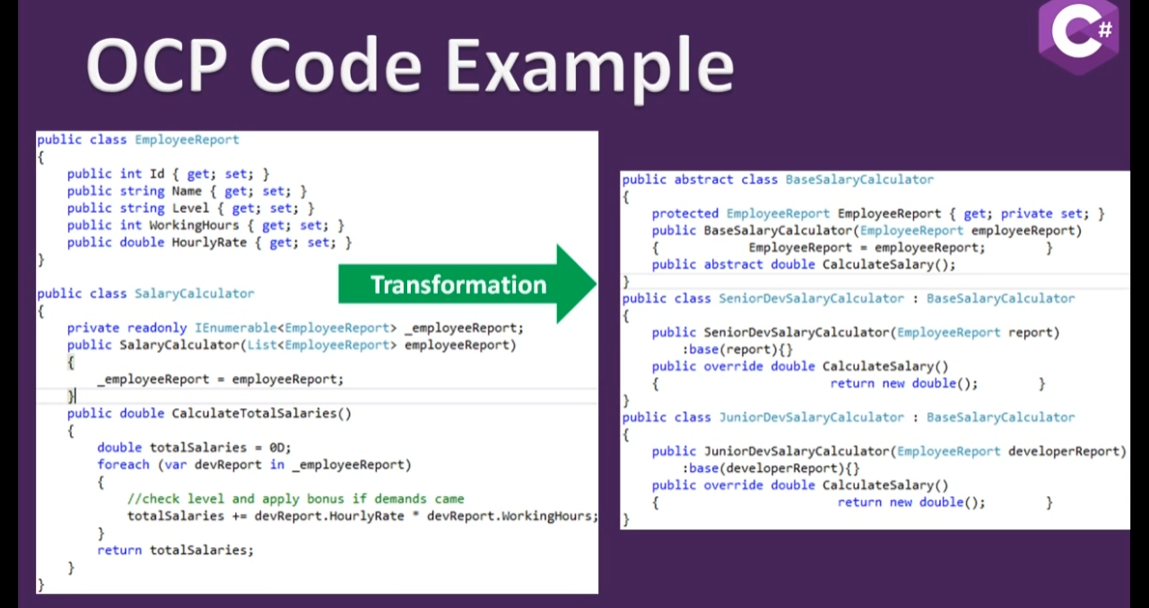


Fig: example of IOC

The **Inversion-of-Control (IoC)** pattern, is about providing any kind of callback, which "implements" and/or controls reaction, instead of acting ourselves directly (in other words, inversion and/or redirecting control to the external handler/controller). The **Dependency-Injection (DI)** pattern is a more specific version of IoC pattern, and is all about removing dependencies from your code.

SOLID:   
Single Responsibility principle:   


Open closed Principle:   
 >class or software entities should be open for extension but closed for modification.

* Use abstract/interface class as a base class for most common abstract feature and use the separate classes for different implementation.
* 

Liskov Substitution Principle:

* Object in a program should be replaceable with the instances of their subtypes without altering the correctness of the program. Example:   
  if class B is subtype of class A, then we should be able to replace object A with B without breaking any behavior of the program. Subclass should extend the capability of parent class not narrow it down
* Base class instance replaced/ substitution by its sub type instance with no change in functionality.
* Extension of open close principle.

public class Vehicle

{

public virtual int getNumberOfWheels()

{

return 2;

}

public virtual string EngineModel()

{

return "test";

}

}

public class MotorCycle: Vehicle

{

public override int getNumberOfWheels()

{

return 4;

}

public override string EngineModel()

{

return "R15";

}

}

public class BiCycle: Vehicle

{

}

var bicycle = new BiCycle();

bicycle.EngineModel();

it breaks the code because bicycle does not contain any engine. It breaks liscov substitute principle.   
To solve this we need the following implementation:

public class Vehicle

{

public virtual int getNumberOfWheels()

{

return 2;

}

}

public class EngineVehicle : Vehicle

{

public virtual string EngineModel()

{

return "test";

}

}

public class MotorCycle: EngineVehicle

{

public override int getNumberOfWheels()

{

return 4;

}

public override string EngineModel()

{

return "R15";

}

}

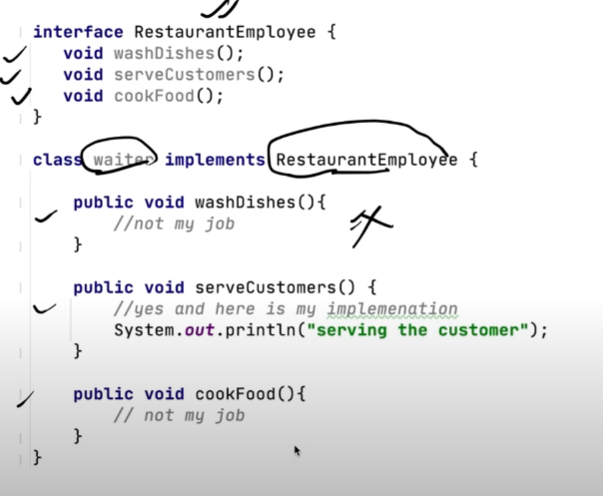
public class BiCycle: Vehicle

{

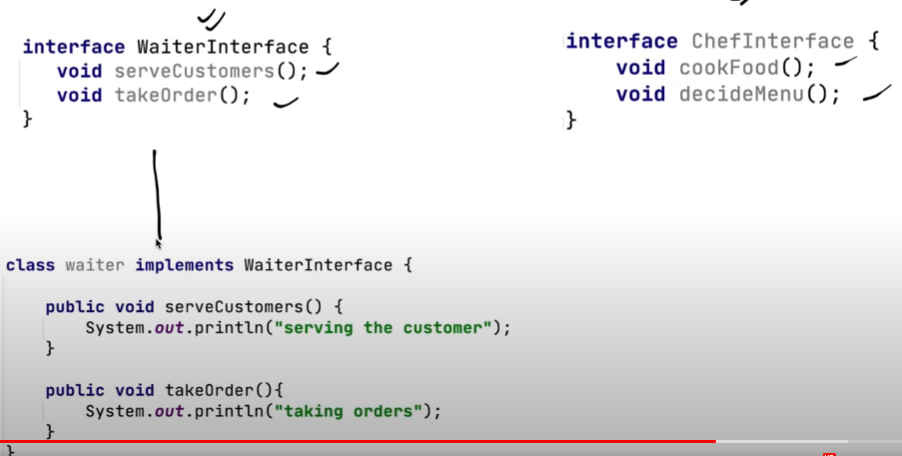
}

**I**nterface Segregation Principle:

Interface should be such, that client should not implement unnecessary functions they do not need.



The above example breaks the interface segregation principle. To solve this we can do the below things:



## Dependency Inversion Principle

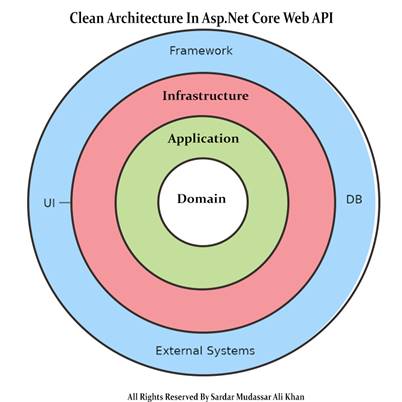
The Dependency Inversion principle states that our classes should depend on interfaces or abstract classes instead of concrete classes and functions.

Robert C. Martin’s definition of the Dependency Inversion Principle consists of two parts:

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

Clean Architecture:

Clean architecture has a domain layer, Application Layer, Infrastructure Layer, and Framework Layer. The domain and application layer are always the center of the design and are known as the core of the system. The core will be independent of the data access and infrastructure concerns. We can achieve this goal by using the Interfaces and abstraction within the core system, but implementing them outside of the core system.



## **Layer In Clean Architecture**

Clean architecture has a domain layer, Application Layer, Infrastructure Layer, and Presentation Layer. The domain and application layer are always the center of the design and are known as the core of the system.

In Clean architecture, all the dependencies of the application are Independent/Inwards and the Core system has no dependencies on any other layer of the system. So, in the future, if we want to change the UI/ OR framework of the system, we can do it easily because all the other dependencies of the system are not dependent on the core of the system.

### **Domain Layer**

The domain layer in the clean architecture contains the enterprise logic, like the entities and their specifications. This layer lies in the center of the architecture where we have application entities, which are the application model classes or database model classes, using the code first approach in the application development using Asp.net core these entities are used to create the tables in the database.

### **Application Layer**

The application layer contains the business logic. All the business logic will be written in this layer. It is in this layer that services interfaces are kept, separate from their implementation, for loose coupling and separation of concerns.

### **Infrastructure Layer**

In the infrastructure layer, we have model objects we will maintain all the database migrations and database context Objects in this layer. In this layer, we have the repositories of all the domain model objects.

### **Presentation Layer**

In the case of the API Presentation layer that presents us the object data from the database using the HTTP request in the form of JSON Object. But in the case of front-end applications, we present the data using the UI by consuming the APIS.

String Vs StringBuilder:

StringBuilder is used to represent a mutable string of characters. Mutable means the string which can be changed. So String objects are immutable but StringBuilder is the mutable string type. It will not create a new modified instance of the current string object but do the modifications in the existing string object. The complete functionality of StringBuilder is provided by StringBuilder class which is present in System.Text namespace.