What is Dependency injection?

Dependency injection (DI) is a software design pattern where an object receives other objects that it depends upon *known as dependencies* rather than those dependencies needing to be instantiated by that object. It is a technique for achieving Inversion of Control (IoC) between classes and their dependencies.

💡 Dependency injection in .NET is a built-in part of the framework, along with configuration, logging, and the options pattern.

Taking an example:

💡 Suppose your ***Client*** class needs to use two service classes, ***Service 1*** and ***Service 2.***

Dependency injection requires your ***Client*** class use an abstraction i.e. ***IService*** interface rather than implementation of these two classes. In this way, you can change the implementation of the ***IService*** interface at any time (and for how many times you want) without changing the client class code.

Why the need?

- It helps us separate concerns making it easy to solve any issues that might arise.
- It allows different parts of the code base to evolve independently from others making development agile.
- Improves code maintainability
- Makes unit testing possible and easier.

Variations of dependency injection

- **Constructor Injection.** This is a widely used way to implement DI. Dependency Injection under this approach is done by supplying the ***DEPENDENCY*** through the class's constructor when creating the instance of that class.
  The injected class can be used anywhere within the class.

> 💡 It is mostly recommended to use when the injected dependency will be used across the class methods and scenarios where a class requires more than one dependency.

- **Property/Setter Injection.** This approach allows you to assign the instance of your dependency to a specific property of the dependent class.

> 💡 Generally recommended when a class has optional dependencies, or where the implementations may need to be swapped. For example: different logger implementations

- **Method Injection**: in this case, you create an instance of the dependency and pass it into a specific method of the dependent class.

> 💡 It could be useful, where the whole class does not need the dependency, only one method utilizing that dependency. This is rarely used.

How to use dependency injection.

You can implement Dependency Injection on your own by creating instances of the lower-level components and passing them to the higher-level ones. You can do it using three common approaches:

- **Constructor Injection**: with this approach, you create an instance of your dependency and pass it as an argument to the constructor of the dependent class.
- **Method Injection**: in this case, you create an instance of your dependency and pass it to a specific method of the dependent class.
- **Property Injection**: this approach allows you to assign the instance of your dependency to a specific property of the dependent class.

.NET Core specifically comes with a built-in Inversion of Control (IoC) Container that simplifies Dependency Injection management.

This container is responsible for supporting automatic Dependency Injection and its basic features:

- **Registration**: the container needs to know which type of object to create for a specific dependency; so, it provides a way to map a type to a class so that it can create the correct dependency instance.
- **Resolution**: this feature allows the IoC container to resolve a dependency by creating an object and injecting it into the requesting class. Thanks to this feature, you don't have to instantiate objects manually to manage dependencies.
- **Disposition**: the IoC Container manages the lifetime of the dependencies following specific criteria.

> 💡 In .NET Core, the dependencies managed by the container are called S**ervices.**

More about Services

All dependencies (services) need to be registered in the IoC container. The IoC container allows you to control the lifetime of a registered service. When you register a service specifying a lifetime, the container will automatically dispose of it accordingly. You have three service lifetimes:

- **Singleton**: this lifetime creates one instance of the service. The service instance may be created at the registration time by using the AddSingleton() method.
- **Transient**: by using this lifetime, your service will be created each time it is requested. This means, for example, that a service injected in the constructor of a class will last as long as that class instance exists. To create a service with the transient lifetime, we use the AddTransient() method.
- **Scoped**: the scoped lifetime allows you to create an instance of a service for each client request. This is particularly useful in the **ASP.NET** context since it allows you to share the same service instance for the duration of an HTTP request processing. To enable the scoped lifetime, you need to use the AddScoped() method.

💡 Choosing the right lifetime for the service you want to use is crucial both for the correct behavior of your application and for better resource management.