**Mis on Dependencye Injection?**

Dependency Injection (DI) is a design pattern used to implement IoC. It allows the creation of dependent objects outside of a class and provides those objects to a class through different ways. Using DI, we move the creation and binding of the dependent objects outside of the class that depends on them.

The Dependency Injection pattern involves 3 types of classes.

1. Client Class: The client class (dependent class) is a class which depends on the service class
2. Service Class: The service class (dependency) is a class that provides service to the client class.
3. Injector Class: The injector class injects the service class object into the client class.

the three types of Dependency Injections:

1. Constructor Injection

The basic premise here is that the object has no defaults or a single constructor. What is required are specified values at the time of creation to instantiate the object. In a nutshell, Constructor Injection uses parameters to inject dependencies. This is the most common DI, which is executed by supplying the dependency through the class’s constructor when instantiating that class.

In addition, an injected component can be used anywhere within the class. Although, it should be used when the injected dependency is required for the class to function. Furthermore, the Constructor Injection is used within the most common scenario when a class requires one or more dependencies.

1. Setter Injection

This is also called the [Property Injection](https://stackoverflow.com/questions/18779894/dependency-injection-when-to-use-property-injection). The Setter Injection lets us create costly resources and services only as required and as late as possible. Plus, it does not require up-front wiring of the entire dependency graph. The only issue is it can be difficult to identify which dependencies are required. Although, it does not require adding or modifying constructors. Furthermore, you will need to check for null before using it.

1. Method Injection

This is the least common and only used for edge cases. As the name states, [Method Injection](https://dotnetfreakblog.wordpress.com/2014/05/11/dependency-injection-using-methods-in-c-part-3/) injects the dependency into a single method to be utilized by that method. As a result, it is useful when the whole class only needs the one method and not the dependency.

With DI, you can inject additional code between the dependencies. To illustrate, you can use the Constructor Injection to give an object its dependencies. If you have a class with 10 methods that have no dependencies, but you want to add a new method with a dependency–you can change the constructor to use Constructor Injection.

Dependency Injection is mainly for injecting the concrete implementation into a class that is using abstraction i.e. interface inside. The main idea of dependency injection is to reduce the coupling between classes and move the binding of abstraction and concrete implementation out of the dependent class.

Dependency injection can be done in three ways.

1. Constructor injection
2. Method injection
3. Property injection

#### **Constructor Injection**

In this approach we pass the object of the concrete class into the constructor of the dependent class. So what we need to do to implement this is to have a constructor in the dependent class that will take the concrete class object and assign it to the interface handle this class is using.

**Kuhu classi kirjutad dependency Injectioni koodi ja mis meetodi alla?**

The dependencies are provided through a client's class constructor.

**Dependency Injectioni koodinäide tuua välja.**

using System.Threading.Tasks;

using Microsoft.Extensions.DependencyInjection;

using Microsoft.Extensions.Hosting;

namespace DependencyInjection.Example

{

class Program

{

static Task Main(string[] args) =>

CreateHostBuilder(args).Build().RunAsync();

static IHostBuilder CreateHostBuilder(string[] args) =>

Host.CreateDefaultBuilder(args)

.ConfigureServices((\_, services) =>

services.AddHostedService<Worker>()

.AddScoped<IMessageWriter, MessageWriter>());

}

}

**Mis tähendab AddSingelton?**

**Mis tähendab AddTransient?**

**Mis tähendab AddScoped?**

## Add Singleton

When we register a type as singleton, only one instance is available throughout the application and for every request.

It is similar to having a static object.

The instance is created for the first request and the same is available throughout the application and for each subsequent requests.

public void ConfigureServices(IServiceCollection services){

services.AddSingleton<ILog,Logger>()

}

## Add Scoped

When we register a type as Scoped, one instance is available throughout the application per request. When a new request comes in, the new instance is created. Add scoped specifies that a single object is available per request.

public void ConfigureServices(IServiceCollection services){

services.AddScoped<ILog,Logger>()

}

## Add Transient

When we register a type as Transient, every time a new instance is created. Transient creates new instance for every service/ controller as well as for every request and every user.

public void ConfigureServices(IServiceCollection services){

services.AddTransient<ILog,Logger>()

}

