# **Dependency injection in C# - a simple introduction**

So, you have heard of Dependency injection (DI) and Inversion of Control (IoC) but are having a hard time grasping the concept? Well you're not alone, DI/IoC can seem quite complex at first! Fortunately they are actually pretty easy to learn and understand, and once you start practising it chances are that you never want to go back to do things the "old bad" way.

**How you might be doing today**

Let's say that you have a class called **Car** and Car need to call a method in the **Engine**class. Today you might either have to provide the instance to Engine manually when you create a Car:

var generator = **new** Generator();  
var engine = **new** Engine(generator);  
var car = **new** Car(engine);

Or perhaps create Engine in the Car's constructor:

**public** **void** **Car**()  
{  
    \_engine = **new** Engine();  
}

The problem with both these examples are that Car is directly dependant on one exact implementation of Engine. Is this bad? Not always, but it can be much better! 

**The Dependency Injection / Inversion of Control way**

Let's say that you'd like to implement the following using Dependency Injection and Inversion of Control. This is how you can do it:  
  
1) Extract your classes method definitions to **Interfaces**:

**public** **interface** **IEngine**  
{  
    **void** **Start**();  
}  
  
**public** **class** **Engine** : **IEngine**  
{  
    **public** **void** **Start**();  
}

2) Create your classes so that all their dependencies are fed to them **as Interfaces through the constructor** and store them in private variables:

**private** **readonly** IEngine \_engine;  
  
**public** **void** **Car**(IEngine engine)  
{  
    \_engine = engine;  
}

3) In the entry point of your application, **register** what instance of what class that should be provided for each interface **with an IoC container** (I'm using Autofac in this example):

var builder = new ContainerBuilder();  
builder.RegisterType<**Generator**>().As<**IGenerator**>();  
builder.RegisterType<**Engine**>().As<**IEngine**>();  
builder.RegisterType<**Car**>().As<**ICar**>();  
var container = builder.Build();

This means that whenever a constructor asks for an instance of type IGenerator the IoC will provide it with an instance of Generator and so on.  
  
4) Start the top-level instance (and all underlying instances will be created automatically for you):

var car = resolver.Resolve<ICar>();  
car.Start();

The following will happen:  
 \* The IoC-container will try to create an instance of ICar using the class Car  
 \* Doing this it will notice that Car needs an instance of IEngine in order to be constructable  
 \* The IoC-container will then try to create an instance of IEngine using the class Engine  
 \* Doing this it will notice that Engine needs an instance of IGenerator in order to be constructable  
 \* The IoC-container will then try to create an instance of IGenerator using the class Generator  
 \* The IoC-container can now create an instance of Engine as all it's dependencies have been met  
 \* The IoC-container can now create an instance of Car as all it's dependencies have been met  
 \* The instance is returned to you  
 \* You can invoke the method Start

**Why is this good?**

This has several benefits. The most important is that it automatically makes your code **testable**. As you are using interfaces everywhere, you can easily provide another implementation in your unit tests. This means that your tests will be much easier to set up as well as being restricted to test a specific unit - not a whole chain of code.  
  
Another important benefit is that it helps you write code that are **loosely coupled** as well as **separating concerns** - e.g. Engine should not have to know more about Generator than necessary and each class should do one thing and do it as isolated as possible from other classes.  
  
DI / IoC can indeed be much more complex that this but hopefully this article has given you enough understanding of the subject so you can continue to explore it. Most people need to try this for them selves in order to really see the benefits. Do it. You will not regret it.

Ref:  
<https://codeaddiction.net/articles/10/dependency-injection-in-c---a-simple-introduction>  
<https://www.codeproject.com/Articles/592372/Dependency-Injection-DI-vs-Inversion-of-Control-IO>