**Dependency injection**

**Written by: Omri Zloof**

Intro - what is a dependency

All programs contain components that pass information between one another.

This is especially prominent when using an Object Oriented Programming language (such as Java on Android), where objects will call methods on other objects that they have references to.

Or put more simply:

***A dependency is when one object uses the API of another object.***

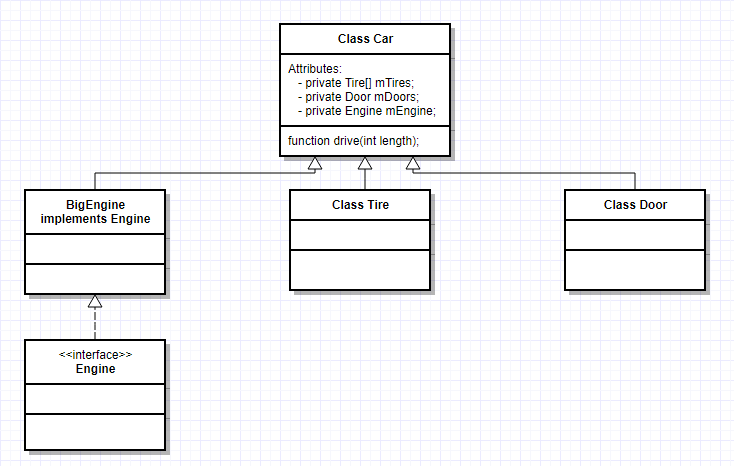
From a practical perspective in Java code, you can identify a dependency in your code whenever you use the **new** keyword to instantiate an object. In such a case, you are fully responsible for creating and properly configuring the object that is being created.

In the following document, we will explore the meaning of a dependency, discuss the problems presented by poor management of code dependencies, and understand the solutions to those problems.

Case study 1: constructor initialization.

In the following example, the class Car has a constructor in which we set up everything we need.

class Car {  
 constructor() {  
 this.engine = new BigEngine();  
 this.tires = Tires.getInstance();  
 this.doors = app.get('doors');  
 }  
}



**So, what’s the problem with this code?**

The code has two main issues:

1. The constructor assigns needed dependencies to internal properties.
2. The constructor knows how each of its dependencies is created.

These problems manifest in the way the engine object is created using the Engine constructor, Tires seems to be a singleton interface and the doors are requested via a global object (or worse, a static function \*\*ask me why static initializers are evil\*\*)

**But why are these the listed issues bad?**

Letting the constructor know how to create all of its dependencies leads to code that is hard to maintain for the following reasons:

* What happens if the Tires class evolves and its getInstance() requires a parameter? (the car class will break)
* What happens if you want to use a different class of Engine for your car?
* When you write tests for Car you're at the mercy of its hidden dependencies. Is it even possible to create a new Engine in a test environment? What does Door depend upon? What does that dependency depend on? Will getting a new instance of Door make an asynchronous call to the server? We don't want that when reusing the code, or during tests.

The issues presented above lead us to the conclusion: let’s take the dependencies from being created in the constructor, and pass them as arguments.

Case study 2: constructor DI

In this example, we moved the dependency creation out of the constructor, and changed the constructor function to expect all needed dependencies as arguments. There are no concrete implementations anymore in this code, as we moved the responsibility of creating those dependencies to a higher level.

If we want to create a car object now, all we have to do is to pass all needed dependencies to the constructor:

class Car {  
 constructor(engine, tires, doors) {  
 this.engine = engine;  
 this.tires = tires;  
 this.doors = doors;  
 }  
 }

The dependencies are now decoupled from our class.

This allows us to pass in other classes that inherit the ones declared, or classes that implement the same interface if one of the argument is an interface, In case we want to reuse the code in a different module, or in case we’re writing test and want to pass in mock objects.

**So, what’s the problem with this code?**

As stated above, the code now moved the responsibility of dependency creation to a higher level. And this is exactly what our new problem is. Who takes care of assembling all those dependencies for us? It’s us. We now need to create a “getCar” function that takes care of instantiating the car dependencies and then passing them to the car constructor - which leads to the same issues in the first example.

**But why are these the listed issues bad?**

Having the dependency injected into the constructor using the constructor arguments is nice, but we still encounter the same problems as we did in “Case study 1”, only now we encounter them when we want to call “new Car(engine, tires, doors)” instead of inside the constructor.

So once again we need to learn from our error, and solve it using

**dependency injection frameworks**

A DI framework class, can analyze the dependencies of a class. With this analysis it is able to create an instance of the class and inject the objects into the defined dependencies.

This way the class has no hard dependencies, which means it does not rely on an instance of a certain class. This allows us to change all instances of Engine in our code to use BigEngine instead, without having to change all classes who depend on it (such as Car).

This will make our code look the same as in “Case study 2”, only that now we don’t have to make a function that prepares the Car for us, the injector takes care of it for us.

Using DI is important, it lets you make reusable, testable code.

**Using di frameworks lets you achieve those goal, in an easier to code way**

Sources:

[*https://blog.thoughtram.io/angular/2015/05/18/dependency-injection-in-angular-2.html*](https://blog.thoughtram.io/angular/2015/05/18/dependency-injection-in-angular-2.html)

[*https://angular.io/guide/dependency-injection*](https://angular.io/guide/dependency-injection)

[*https://he.wikipedia.org/wiki/Dependency\_injection*](https://he.wikipedia.org/wiki/Dependency_injection)

[*http://www.vogella.com/tutorials/Dagger/article.html*](http://www.vogella.com/tutorials/Dagger/article.html)

[*https://www.raywenderlich.com/146804/dependency-injection-dagger-2*](https://www.raywenderlich.com/146804/dependency-injection-dagger-2)

[*https://android.jlelse.eu/android-mvp-architecture-with-dependency-injection-dee43fe47af0*](https://android.jlelse.eu/android-mvp-architecture-with-dependency-injection-dee43fe47af0)

[*https://github.com/google/guice/wiki/Motivation*](https://github.com/google/guice/wiki/Motivation)