Defining Dependency Injection

All most all developers get fear when they hear the word dependency injection. It's a difficult pattern and it's not meant for beginners. That's what we are made to believe. The truth is that dependency injection is a fundamental pattern that is very easy to adopt.

When I first heard about dependency injection, I also thought it was a technique too advanced for my needs at the begining time. I could do without dependency injection, whatever it was.

I later learned that, if reduced to its bare essentials, dependency injection is a simple concept.

James Shore offers a succinct and straightforward definition of dependency injection.

Dependency injection means giving an object its instance variables. Really. That's it. - James Shore

What is Dependency Injection? Why it is very important?

Lets assume of computer manufacture company like Apple, Dell, HP etc, all complete parts are not manufacture by them. In one laptop we have different company manufactured parts and assembled by them. Imagine you have Dell laptop with Samsung drive. Somehow drive damaged and Samsung drive not available, need to take Toshiba drive. If mother board is not going to work with that then, problem. Than its tightly coupled with Samsung drive.

Same in our code if classes are tightly coupled with each other then we will have problem. One place small change also cause high level issue. To make it stable, need to make changes everywhere. Because code transparency not there.

For long term projects and dynamic requirement, its a big issue. Even creates problem for new joiner of the stake holders. To resolve this issue Dependency Injection concept came.

Dependency Injection (DI) is a technique which allows to populate a class with objects, rather than relying on the class to create the objects itself.

Dependency Injection(DI) helps us to make our components less coupled and more reusable in different contexts. Basically, it is one form of *Separation of Concerns*, because it separates the algorithm using the dependencies from their initialisation and configuration. To achieve that, we can apply different techniques to inject dependencies into our modules.

As mentioned above, one really important aspect of Dependency Injection is that it makes our code more testable. We can inject mock instances for the dependencies of our classes/modules we want to test. This allows us to focus our tests on the unit of code in the module and make sure that this part is working as expected, without having fuzzy side effects that lead to unclear test failures, because one of its dependencies is not behaving as expected. These dependencies should be tested on their own, to find the real mistakes easier and speed up your development workflow.

dependency injection cannot be considered without *Dependency inversion* principle. The principle states that implementation details should depend on and implement higher level abstractions, rather than the other way around. It is foundational when creating loosely-coupled applications

Dependency Injection is used ubiquitously in Cocoa too, and in the following examples, we'll see code snippets both from Cocoa and typical client-side code. Let's take a look at the following four sections to learn how to use Dependency Injection.

Let have following example:

```
13  let endPoint = "https://restcountries.eu/rest/v2/name/India"
```

We have a JSON endpoint which gives us Country details of India.

The bellow EndpointClient class to get the country details from the above endpoint.

```
//Mark: Without DI
//Mark: Endpoint Client Defination Class

class EndpointClient{
    func getCountryData(url: URL, completionHandler:@escaping (_ data:
        Data?) -> Void){
        URLSession.shared.dataTask(with: url){ (dataResponse, urlResponse,
            error) in
            completionHandler(dataResponse)

        }.resume()
    }
}
```

The below CallCountry class gets the data response using EndpointClient class.

```
class CallCountry{
    func getCountryDetails() {
        let clinet: EndpointClient = EndpointClient()
        /*
         The above line of code is hidden dependacy.
         If any new developer joins the team, developer cant know this unless
         going through the coding documents.

         */

        clinet.getCountryData(url: URL(string: endPoint)!){(responseData) in
            if responseData?.count != 0 {
                //process the response data
                print("Country Data \(String(describing:
                    responseData?.count))")

            }

        }

    }
}
```

And here initialisation of CallCountry class and get the country details implementation.

```
        let country = CallCountry() // initialized object

        country.getCountryDetails() // called details method
```

Here is the output.

```
Country Data Optional(2378)
```

This code works perfectly, but the line number 74 is hidden dependency. The end developer who is initialisation of CallCountry class and geting the country details for further implementation. To avoid this DI came into picture.

Important ways to use Dependency Injection(DI) in Swift (Beginners)

1. **Property Injection**
2. **Constructor Injection**

Property Injection

As per my knowledge that Construction Injection is the best way to do DI, so why bother finding other methods? Well, it is not always possible to define the constructor the way we want. We will see why not always later below. Let's talk about Property Injection first.

```
72  class CallCountry{
73
74      var client : EndpointClient? = nil
75
76      func getCountryDetails() {
77         // let clinet: EndpointClient = EndpointClient()
78          /*
79           The above line of code is commented hidden dependacy.
80           Added clinet object as class property
81           */
82
83          client?.getCountryData(url: URL(string: endPoint)!){(responseData) in
84              if responseData?.count != 0 {
85                  //process the response data
86                  print("Country Data \(String(describing:
                        responseData?.count))")
87
88              }
89
90          }
91
92      }
93  }
```

I commented following line

// let client: EndpointClient = EndpointClient()

And created class property called “client” of type EndpointClient class. You can see at bellow code, I have injected the client property. That way the hidden “client” property is now visible to developers.

```
28          let country = CallCountry() // initialized object
29          country.client = EndpointClient()//Property Injected
30          country.getCountryDetails() // called details method
```

Output is same as earlier, but dependency removed.

```
Country Data Optional(2378)
```

This pattern is less elegant than the previous one:

1. The CallCountry isn't in the right state until all the properties are set
2. Properties introduce mutability, and immutable classes are simpler and more efficient
3. The properties must be defined as optional, leading to add question marks everywhere
4. They are set by an external object, so they must be writeable and this could potentially permit something else to overwrite the value set at the beginning after a while
5. There is no way to enforce the validity of the setup at compile-time

This is easiest way to implement DI but if developers some how forget or missed to pass the property then it will not throw error but expected result will not come. Because the property is set to be optional. All though is human error, but need to avoid it.

Constructor Injection

The second way to do DI is to pass the collaborators in the constructor, where they are then saved in private properties. Let's have as an example on

This is the best way to do DI. After the construction, the object is fully formed and it has a consistent state. Also, by just looking at the signature of init, the dependencies of this object are clear.

In the below code snippet, added constructor with argument type EndpointClient object.

```
72 class CallCountry{
73
74     var client : EndpointClient? = nil
75
76     /*
77      Added Constructor with dependency value as argument
78      */
79     init(_client : EndpointClient){
80         client = _client
81     }
82     func getCountryDetails() {
83
84         client?.getCountryData(url: URL(string: endPoint)!){(responseData) in
85             if responseData?.count != 0 {
86                 //process the response data
87                 print("Country Data \(String(describing:
                       responseData?.count))")
88
89             }
90
91         }
92
93     }
94 }
```

So that when developers implements CallCountry, it prompts for EndpointClient object. This is basically constructor DI.

```
28          let country = CallCountry(_client: EndpointClient)// initialized
                object
```

```
28          let country = CallCountry(_client: EndpointClient())
29          // initialized object with EndpointClient object
30          country.getCountryDetails() // called details method
```

```
Country Data Optional(2378)
```

Actually, the Constructor Injection is not only the most effective, but it's also the easiest. This way if in future instead of JSON endpoint wants to point XML end just change the endpoint url and instead of JSON parse, use XML parse mechanism.

Of-course, DI increases the code lines but it decouples and removes the dependency with complete transparency.

There more DI implementations available, I will be discussing them in advance DI article. Soon you will have in next. Also next article will cover **Dependency Injection using storyboard**.

Git link: https://github.com/Nayaksb/DependencyInjection/tree/MediumAndTutorials

References: https://en.wikipedia.org/wiki/Dependency_injection