Chapter 45: Dependency Injection

Section 45.1: Dependency Injection with View Controllers

Dependenct Injection Intro

An application is composed of many objects that collaborate with each other. Objects usually depend on other objects to perform some task. When an object is responsible for referencing its own dependencies it leads to a highly coupled, hard-to-test and hard-to-change code.

Dependency injection is a software design pattern that implements inversion of control for resolving dependencies. An injection is passing of dependency to a dependent object that would use it. This allows a separation of client's dependencies from the client's behaviour, which allows the application to be loosely coupled.

Not to be confused with the above definition - a dependency injection simply means giving an object its instance variables.

It's that simple, but it provides a lot of benefits:

- easier to test your code (using automated tests like unit and UI tests)
- when used in tandem with protocol-oriented programming it makes it easy to change the implementation of a certain class - easier to refactor
- it makes the code more modular and reusable

There are three most commonly used ways Dependency Injection (DI) can be implemented in an application:

1. Initializer injection
2. Property injection
3. Using third party DI frameworks (like Swinject, Cleanse, Dip or Typhoon)

There is an interesting article with links to more articles about Dependency Injection so check it out if you want to dig deeper into DI and Inversion of Control principle.

Let's show how to use DI with View Controllers - an every day task for an average iOS developer.

Example Without DI

We'll have two View Controllers: **LoginViewController** and **TimelineViewController**. LoginViewController is used to login and upon successful loign, it will switch to the TimelineViewController. Both view controllers are dependent on the **FirebaseNetworkService**.

LoginViewController

```
class LoginViewController: UIViewController {

    var networkService = FirebaseNetworkService()

    override func viewDidLoad() {
        super.viewDidLoad()
    }
}
```

TimelineViewController

```
class TimelineViewController: UIViewController {
```

```
    var networkService = FirebaseNetworkService()

    override func viewDidLoad() {
        super.viewDidLoad()
    }

    @IBAction func logoutButtonPressed(_ sender: UIButton) {
        networkService.logutCurrentUser()
    }
}
```

FirebaseNetworkService

```
class FirebaseNetworkService {

    func loginUser(username: String, passwordHash: String) {
        // Implementation not important for this example
    }

    func logutCurrentUser() {
        // Implementation not important for this example
    }
}
```

This example is very simple, but let's assume you have 10 or 15 different view controller and some of them are also dependent on the FirebaseNetworkService. At some moment you want to change Firebase as your backend service with your company's in-house backend service. To do that you'll have to go through every view controller and change FirebaseNetworkService with CompanyNetworkService. And if some of the methods in the CompanyNetworkService have changed, you'll have a lot of work to do.

Unit and UI testing is not the scope of this example, but if you wanted to unit test view controllers with tightly coupled dependencies, you would have a really hard time doing so.

Let's rewrite this example and inject Network Service to our view controllers.

Example with Dependancy Injection

To make the best out of the Dependency Injection, let's define the functionality of the Network Service in a protocol. This way, view controllers dependent on a network service won't even have to know about the real implementation of it.

```
protocol NetworkService {
    func loginUser(username: String, passwordHash: String)
    func logutCurrentUser()
}
```

Add an implementation of the NetworkService protocol:

```
class FirebaseNetworkServiceImpl: NetworkService {
    func loginUser(username: String, passwordHash: String) {
        // Firebase implementation
    }

    func logutCurrentUser() {
        // Firebase implementation
    }
}
```

Let's change LoginViewController and TimelineViewController to use new NetworkService protocol instead of FirebaseNetworkService.

LoginViewController

```
class LoginViewController: UIViewController {

    // No need to initialize it here since an implementation
    // of the NetworkService protocol will be injected
    var networkService: NetworkService?

    override func viewDidLoad() {
        super.viewDidLoad()
    }
}
```

TimelineViewController

```
class TimelineViewController: UIViewController {

    var networkService: NetworkService?

    override func viewDidLoad() {
        super.viewDidLoad()
    }

    @IBAction func logoutButtonPressed(_ sender: UIButton) {
        networkService?.logutCurrentUser()
    }
}
```

Now, the question is: How do we inject the correct NetworkService implementation in the LoginViewController and TimelineViewController?

Since LoginViewController is the starting view controller and will show every time the application starts, we can inject all dependencies in the **AppDelegate**.

```
func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions:
[UIApplicationLaunchOptionsKey: Any]?) -> Bool {
    // This logic will be different based on your project's structure or whether
    // you have a navigation controller or tab bar controller for your starting view controller
    if let loginVC = window?.rootViewController as? LoginViewController {
        loginVC.networkService = FirebaseNetworkServiceImpl()
    }
    return true
}
```

In the AppDelegate we are simply taking the reference to the first view controller (LoginViewController) and injecting the NetworkService implementation using the property injection method.

Now, the next task is to inject the NetworkService implementation in the TimelineViewController. The easiest way is to do that when LoginViewController is transitioning to the TimlineViewController.

We'll add the injection code in the prepareForSegue method in the LoginViewController (if you are using a different approach to navigate through view controllers, place the injection code there).

Our LoginViewController class looks like this now:

```
class LoginViewController: UIViewController {
    // No need to initialize it here since an implementation
    // of the NetworkService protocol will be injected
    var networkService: NetworkService?

    override func viewDidLoad() {
        super.viewDidLoad()
    }

    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        if segue.identifier == "TimelineViewController" {
            if let timelineVC = segue.destination as? TimelineViewController {
                // Injecting the NetworkService implementation
                timelineVC.networkService = networkService
            }
        }
    }
}
```

We are done and it's that easy.

Now imagine we want to switch our NetworkService implementation from Firebase to our custom company's backend implementation. All we would have to do is:

Add new NetworkService implementation class:

```
class CompanyNetworkServiceImpl: NetworkService {
    func loginUser(username: String, passwordHash: String) {
        // Company API implementation
    }

    func logutCurrentUser() {
        // Company API implementation
    }
}
```

Switch the FirebaseNetworkServiceImpl with the new implementation in the AppDelegate:

```
func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions:
[UIApplicationLaunchOptionsKey: Any]?) -> Bool {
        // This logic will be different based on your project's structure or whether
        // you have a navigation controller or tab bar controller for your starting view controller
        if let loginVC = window?.rootViewController as? LoginViewController {
            loginVC.networkService = CompanyNetworkServiceImpl()
        }
        return true
    }
```

That's it, we have switched the whole underlaying implementation of the NetworkService protocol without even touching LoginViewController or TimelineViewController.

As this is a simple example, you might not see all the benefits right now, but if you try to use DI in your projects, you'll see the benefits and will always use Dependency Injection.

Section 45.2: Dependency Injection Types

This example will demonstrate how to use Dependency Injection (**DI**) design pattern in Swift using these methods:

1. **Initializer Injection** (the proper term is Constructor Injection, but since Swift has initializers it's called initializer injection)
2. **Property Injection**
3. **Method Injection**

Example Setup without DI

```
    protocol Engine {
        func startEngine()
        func stopEngine()
    }

    class TrainEngine: Engine {
        func startEngine() {
            print("Engine started")
        }

        func stopEngine() {
            print("Engine stopped")
        }
    }

    protocol TrainCar {
    var numberOfSeats: Int { get }
    func attachCar(attach: Bool)
}

class RestaurantCar: TrainCar {
    var numberOfSeats: Int {
        get {
            return 30
        }
    }
    func attachCar(attach: Bool) {
        print("Attach car")
    }
}

class PassengerCar: TrainCar {
    var numberOfSeats: Int {
        get {
            return 50
        }
    }
    func attachCar(attach: Bool) {
        print("Attach car")
    }
}

class Train {
    let engine: Engine?
    var mainCar: TrainCar?
}
```

Initializer Dependency Injection

As the name says, all dependencies are injected through the class initializer. To inject dependencies through the initializer, we'll add the initializer to the `Train` class.

Train class now looks like this:

```
class Train {
```

```
    let engine: Engine?
    var mainCar: TrainCar?

    init(engine: Engine) {
        self.engine = engine
    }
}
```

When we want to create an instance of the Train class we'll use initializer to inject a specific Engine implementation:

```
let train = Train(engine: TrainEngine())
```

NOTE: The main advantage of the initializer injection versus the property injection is that we can set the variable as private variable or even make it a constant with the `let` keyword (as we did in our example). This way we can make sure that no one can access it or change it.

Properties Dependency Injection

DI using properties is even simpler that using an initializer. Let's inject a PassengerCar dependency to the train object we already created using the properties DI:

```
train.mainCar = PassengerCar()
```

That's it. Our train's `mainCar` is now a `PassengerCar` instance.

Method Dependency Injection

This type of dependency injection is a little different that the previous two because it won't affect the whole object, but it will only inject a dependency to be used in the scope of one specific method. When a dependency is only used in a single method, it's usually not good to make the whole object dependent on it. Let's add a new method to the Train class:

```
func reparkCar(trainCar: TrainCar) {
    trainCar.attachCar(attach: true)
    engine?.startEngine()
    engine?.stopEngine()
    trainCar.attachCar(attach: false)
}
```

Now, if we call the new Train's class method, we'll inject the `TrainCar` using the method dependency injection.

```
train.reparkCar(trainCar: RestaurantCar())
```