SWIFT BEST PRACTICES

DEPENDENCY INJECTION – DIFFICULTY: MEDIUM

INTRODUCTION

Dependency injection is designed to solve the problem of hidden dependencies: external data that a method uses that wasn't explicitly handed to it. They are called “hidden” because you can't see them when you look at a function signature – it might accept a string and an array of integers, but internally might draw on other system state that is outside your control. The dependency injection solves this by making us explicitly send all data into a function that the function needs to operate.

THE PROBLEM

If you already completed the Singletons document you saw how one way of dealing with common singletons is to make them an implementation detail, which is particularly useful when dealing with singletons you own that you use everywhere.

However, dependencies that you use only sometimes are more tricky. This is particularly common with system classes such as **UIApplication**, **UIDevice**, **NotificationCenter**, and more. It's common to see code like this:

```
if UIDevice.current.userInterfaceIdiom == .pad {
  // run iPad code
}
```

This makes the code harder to write tests for, because we might want to be able to run tests for behavior on specific devices.

THE SOLUTION

Dependency injection makes hidden dependencies explicit by passing them in to the type or function that needs to use it. This is likely to be annoying with certainly dependencies that might need to be passed from object to object, such as loggers or analytics, but for it's extremely useful for system singletons such as **UIApplication** and **UIDevice**.

For example, working with images requires knowledge of the screen scale: is it a 2x device or a 3x device? You can read this by using **UIScreen.main.scale**, but doing so won't let you write unit tests easily because you don't have control over that value. So, instead you might write code like this:

```
func render(using scale: CGFloat) {
   // ...
}
```

Even better, you can specify the original value you had as a default parameter, allowing you to inject test values when necessary but not care otherwise:

```
func render(using scale: CGFloat = UIScreen.main.scale) { ... }
```

Things are harder when using something like **UIApplication**. You might want to test that **UIApplication.shared.open()** was called inside a method, but you can't subclass **UIApplication** for testing – creating an instance of **UIApplication** just crashes your code.

In this instance, we must first wrap our type in a protocol, like this:

```
protocol URLOpening {
    func open(_ url: URL, options:
    [UIApplication.OpenExternalURLOptionsKey: Any], completionHandler
    completion: ((Bool) -> Void)?)
}
```

UIApplication already conforms because the only required method was taken from that class:

```
extension UIApplication: URLOpening { }
```

But now can create a testing data type that conforms to the same protocol, whose job it is just to track how many times the **open()** method was called:

```
class ApplicationMock: URLOpening {
    var urlOpenCount = 0

    func open(_ url: URL, options:
    [UIApplication.OpenExternalURLOptionsKey: Any], completionHandler
    completion: ((Bool) -> Void)?) {
       urlOpenCount += 1
    }
}
```

With that in place we can write methods like this:

```
func openURL(_ url: URL, using: URLOpening = UIApplication.shared) {
   // ...
}
```

THE CHALLENGE

Our sample app uses system singletons such as **UIApplication**, **UIDevice**, and **FileManager**, which makes it hard to write tests for.

Can you try migrating one of them over to dependency injection? You can choose any one you want, but you'll probably find **UIApplication** easiest – look for where **UIApplication.shared.open** is used in the project and start there.