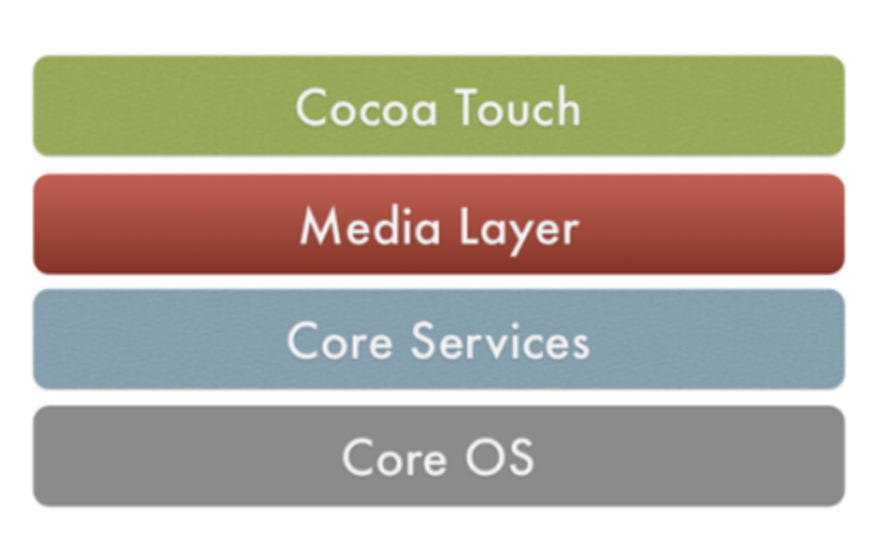
Public api: <https://github.com/public-apis/public-apis>

Ios patern designs: <https://viblo.asia/p/ios-architecture-patterns-mvc-mvp-mvvm-ban-da-hieu-ro-chua-bWrZnPyO5xw>

**I. IOS application architecture:**



**Core OS includes:**

* Core Bluetooth Framework.
* Accelerate Framework.
* External Accessory Framework.
* Security Services framework.
* Local Authentication framework

**Core Service includes:**

- ****Address book framework****– Gives programmatic access to a contacts database of user

- ****Cloud Kit framework****– Gives a medium for moving data between your app and iCloud

- ****Core data Framework –**** Technology for managing the data model of a Model View Controller app

- ****Core Foundation framework –**** Interfaces that give fundamental data management and service features for iOS apps

- ****Core Location framework –**** Gives location and heading information to apps

- ****Core Motion Framework –**** Access all motion-based data available on a device. Using this core motion framework Accelerometer based information can be accessed

- ****Foundation Framework –**** Objective C covers too many of the features found in the Core Foundation framework

- ****Healthkit framework –**** New framework for handling health-related information of the user

- ****Homekit framework –**** A new framework for talking with and controlling connected devices in a user’s home

- ****Social framework –**** Simple interface for accessing the user’s social media accounts

- ****StoreKit framework –**** Gives support for the buying of content and services from inside your iOS apps, a feature is known as In-App Purchase



3. Media layer:

*Graphics Framework:*

- UIKit graphics: use for support designing images and use for animating contents of your view.

- Core graphic framework: is drawing engine for ios apps, customs 2D vector and image-based rendering.

- Core animation: technique for optimizing animation in ios apps.

- Core image: support for video and motionless image.

- OpenGL and GLKit: manager 2D, 3D rendering by hardware-accelerated interfaces

*Audio Framework:*

- Media player framework:

- AV Foundation: handling recording and playback of video and audio.

- Open AL: technology for audio.

*Video Framework:*

- AV Kit: presenting video

- AV Foundation: playback and record video

- Core media:

1. Cocoa Touch:

***II. Memory manager:***

<https://viblo.asia/p/co-phai-khi-nao-cung-can-dung-weak-self-trong-closures-gAm5y4bLldb>

## **1. Automatic Reference Counting**

- Bộ nhớ trong Swift được quản lí theo cơ chế ARC, nó hoạt động theo cơ chế tính số lượng reference đến vùng nhớ

Nó sẽ được giải phóng khi không có đối tượng nào tham chiếu đến nó. Vì thế trong một số trường hợp bạn không giải phóng được do vẫn có strong reference đến nó.

- Một số phương pháp để kiểm tra memory leak:

* Kiểm tra trong hàm deinit() khi ViewController bị đóng.
* Quan sát mức độ bộ nhớ tăng dần
* Sử dụng tool Allocations Instruments

## **2. Unowned, weak và Strong - weak**

- Closues có thể strong capture bất cứ biến hay hằng số trong context mà nó được định nghĩa. Ví dụ bạn sử dụng self trong closures thì nó sẽ được duy trì trong suốt vòng đời của scope đó. Và nếu self vẫn giữ strong reference sẽ dẫn đến strong reference cycle => phải sử dụng [weak self] và [unowned self]

- [unowned self]: nếu object bị nil => bị crash.

## **3. Escaping và non-escaping closures**

- Non-escaping closures: thực thi code trong scope của nó một cách tức thì và không có khả năng lưu trữ hay sử dụng sau đó => không gây ra reference cycle => ko cần sử dụng unowned và weak.

- Escaping closures: có thể lưu trữ nó vào 1 biến hoặc 1 closure khác và có thể thực thi nó trong tương lai

=> nếu closues đc  lưu trữ vào 1 biến hoặc 1 closures khác

hoặc có sử dụng self => fai sử dụng [unowned self] và [weak self]



### **6.1 Grand Central Dispatch**

- Dispatch thường sẽ ko gây ra reference cycle

- dispatch được lưu trữ vào 1 biến thì sẽ gây ra leak

func leakyDispatchQueue() { let workItem = DispatchWorkItem { self.view.backgroundColor = .red } DispatchQueue.main.asyncAfter(deadline: .now() + 1.0, execute: workItem) self.workItem = workItem // stored in a property }

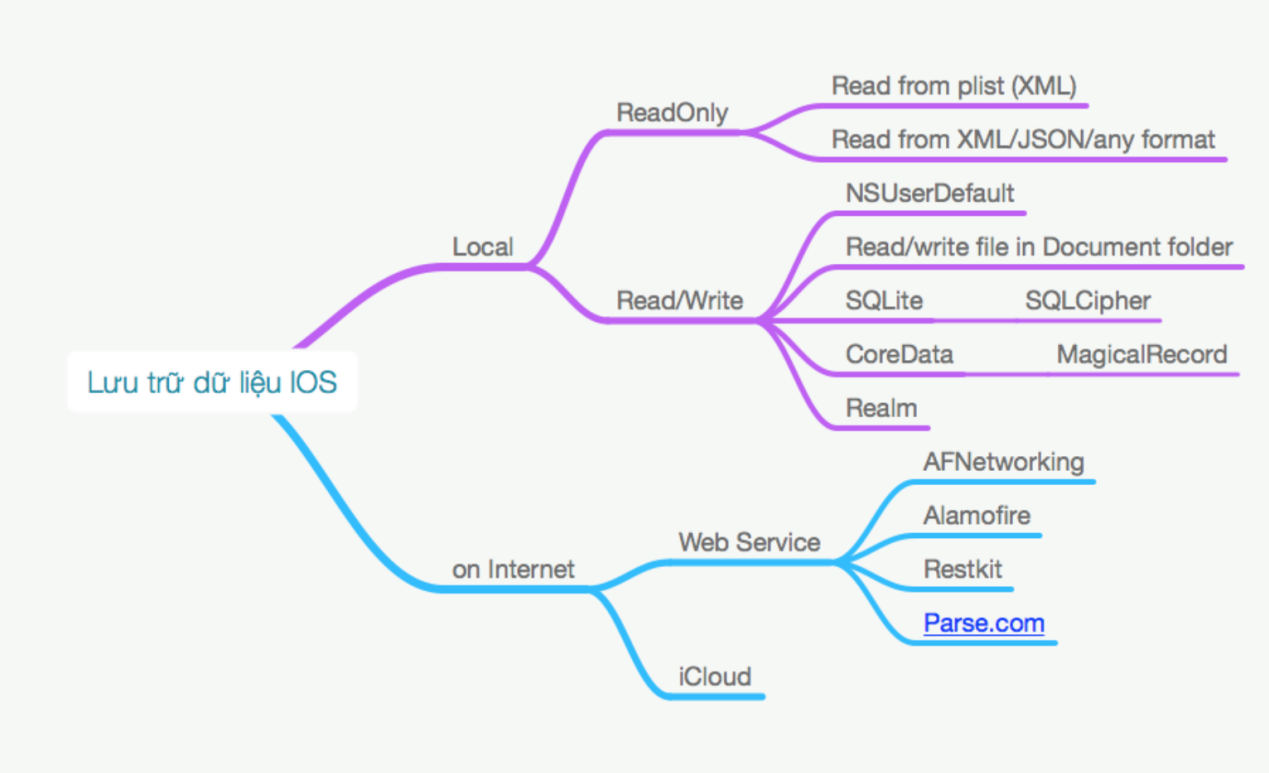
### **6.2 UIView.Animate and UIViewPropertyAnimator**

Tương tự như GCD nó cũng không gây ra memory leak nếu không lưu trữ nó vào thuộc tính.



**III. IOS Backend services:**

<https://blog.back4app.com/ios-backend-service/>



1. **RESTFull API:**

- RestFul la 1 chuẩn để thiết kế các api cho web service.

- API (Application Programing Interface): là 1 tập hợp các cơ chế để các ứng dụng tương tác với nhau. API trả về kiểu dữ liệu Json, xml.

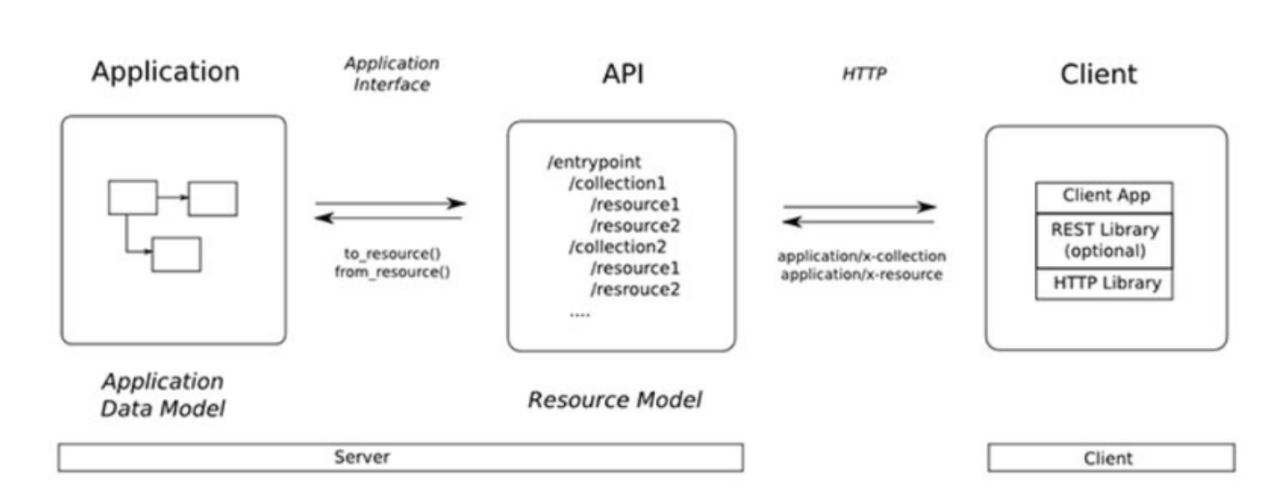
- REST: là 1 kiểu cấu trúc để viêts API

+ nó sử dụng phương thức HTTP để tạo giao tiếp giữa các => thay vì sử dụng 1 URL để sử lý thông tin người dùng, Rest gửi các yêu cầu http: get, post, delete... đến 1 URL để sử lý dữ liệu.

+ Rest quy định việc sử dụng các http method: get, post, , put...

+ Rest quy đinh cách định dạng URL cho ứng dụng web để xử lý resource.

- Nguyên lý hoạt động của restful api:



1. **WebSocket:**

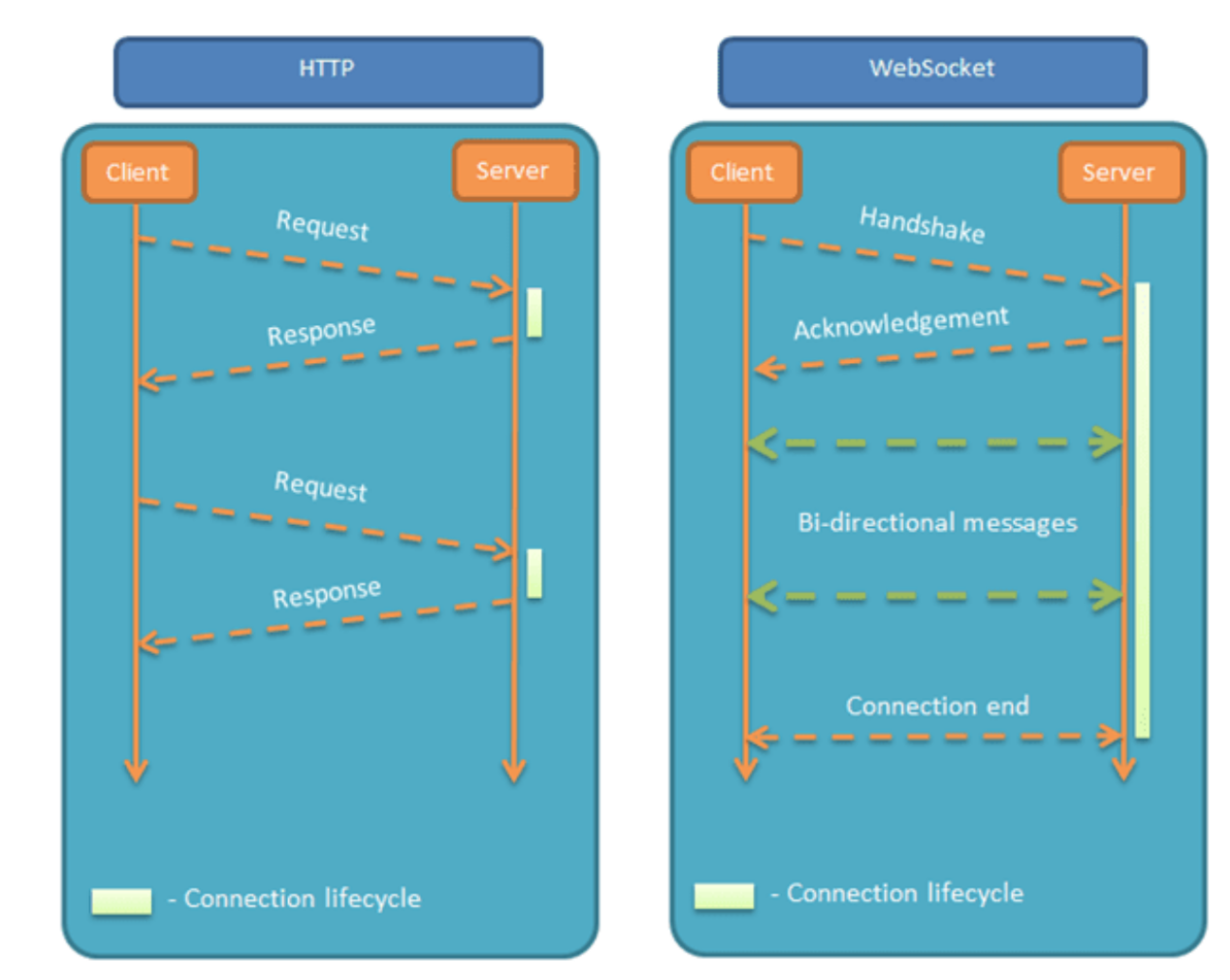
- là giao thức mạng, cho phép kết nối 2 chiều giữa client và web.

- Websocket rất hữu dụng cho các ứng dụng di động, cho phép các ứng dụng này tái sử dụng lại các công cụ, API được xây dựng cho ứng dụng web, như việc chúng ta tái sử dụng các API trên giao thức HTTP.

* WebSocket là một phương thức giúp máy trạm và máy chủ có thể giao tiếp thời gian thực 2 chiều với nhau.
* Khả năng giảm độ trễ của mạng đến mức tối đa và vượt trội hơn HTTP. Nếu bạn phát triển các ứng dụng thời gian thực như: sàn tiền ảo, Game, chat hay video call,… sử dụng WebSocket sẽ mang lại trải nghiệm tối ưu nhất cho người dùng.
* Khả năng giảm kích thước header HTTP lên đến 1000 lần! Nếu một request HTTP có header nặng 871 byte, khi sử dụng WebSocket, khối lượng header của request chỉ còn 2 byte.

- Sử dụng trong các app: theo real-time, game online, app chat.

- Thư viện cho ios app: **SocketRocket** sử dungj objective-



1. **HTTP/HTTPs:**

**- là giao thức 1 chiều**dựa theo giao thức TCP, bạn có thể tạo ra các kết nối dựa vào request HTTP, sau khi kết nối được thực hiện xong và được phản hồi lại, quá trình sẽ kết thúc và đóng lại.

- Ứng dụng: dùng trong các app Restful đơn giản, chỉ nhận thông tin 1 chiều về server để xử lý => phản hồi => đóng kết nối.

**IV. Push Notification:**

Có 2 loại :

- local notification:

- Remote notification (push notification): sử dụng các remote push như Firebase, SNS Amazon cloud service thông qua APNS.

1. **Local Notifications:**

-Registering UNUserNotificationCenter in file appdelegate.swift, UNUserNotificationCenterDelegate

- Sending a noti: using UNTimeIntervalNotificationTrigger

UNCalendarNotificationTrigger

UNLocationNotificationTrigger

1. **Remote Notifications:**

- Là dạng push được send bằng backend tức là bằng server.

1. APNS: Apple push notification service



**V. Biometrics:**

- là bảo mật bằng vân tay, Touch ID

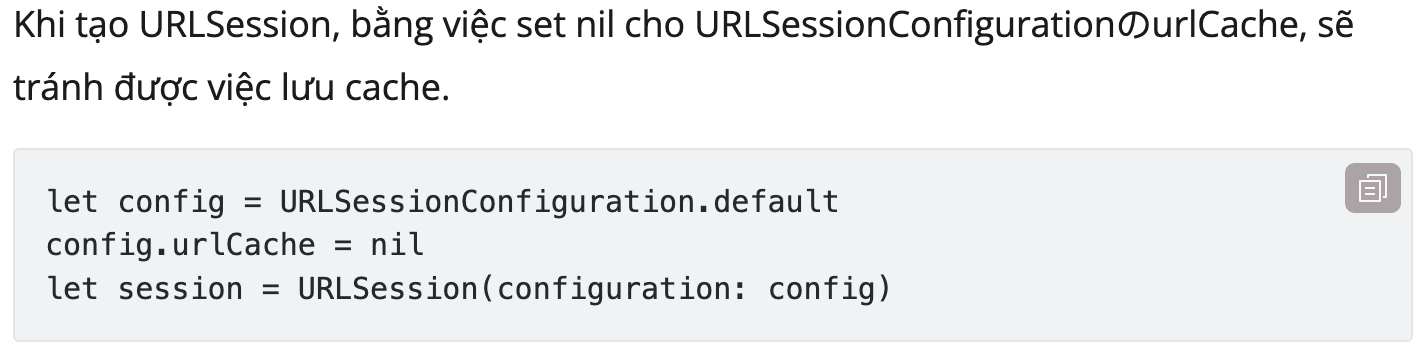
- sử dụng ***LocalAuthentication***

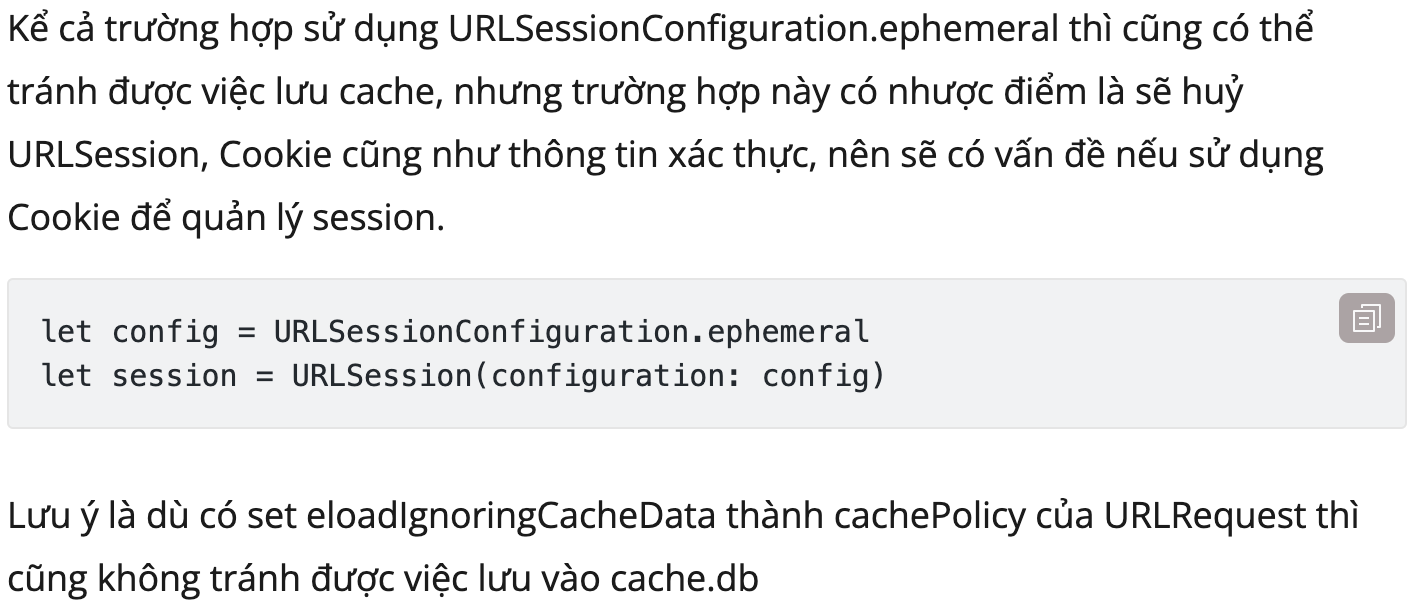
**VI. Security for ios app:**

1. **[iOS] Cache dữ liệu truyền:**

- Nếu thực hiện truyền dữ liệu bằng URLSession,..thì dữ liệu truyền bởi OS sẽ bị lưu tự động vào file (cache.db) trong thiết bị. Dữ liệu cache này do được lưu bằng plain text, kể cả truyền bằng HTTPS, nên có khả năng cung cấp thông tin có lợi cho kẻ tấn công.

- Giải phaps:





1. **[iOS] App Transport Security(ATS):**

Rủi ro

Nếu vô hiệu hoá App Transport Security (ATS) thì đồng nghĩa với việc chấp nhận truyền thông tin bằng phương thức không an toàn như truyền bằng plain text. Bên cạnh đó, có khả năng trong tương lai Apple sẽ bắt buộc hoá việc cài đặt ATS.

Giải pháp

Nếu phía server có khả năng đáp ứng thì nên cài đặt App Transport Security (ATS).

**VII. View Controller Life Cycle**

- là vòng đời của một view controller được tính từ lúc nó được nạp vào bộ nhớ(RAM) cho tới khi nó bị huỷ khỏi bộ nhớ.

**Cac function trog 1 ViewController life cycle:**

- LoadView

Hàm này sẽ được gọi đầu tiên khi khởi tạo một view controller.

Hàm này được dùng khá nhiều vào cái thời code giao diện bằng tay hay nói cách khác là thời chưa có Storyboard.

Mục đích chính là dùng để khởi tạo các UI (Label, Button, Text Field…) chứa t rong View lớn này.

- viewDidLoad

Khi view controller đã được nạp vào bộ nhớ( điều kiện là cái view controller này chưa tồn tại trong bộ nhớ), thì hàm viewDidLoad được gọi.

Chỉ được gọi một lần duy nhất trong chu kỳ sống của view đó thôi.

Thường dùng để chuẩn bị data hoặc là khởi tạo các giá trị mặc định cho các object cũng như UI trên màn hình.

- viewDidUnload

Khi app của bạn nhận được cảnh báo từ hệ thống về trạng thái bộ nhớ đang gần hết thì hàm này sẽ được gọi

Tại hàm này sẽ giải phóng bớt các property không cần dùng, gán nil chúng để giải phóng bộ nhớ.

- viewWillAppear:

Hàm sẽ được gọi trước khi một view được thêm vào hệ thống view và trước animation hiển thị một view.

Tại thời điểm animation để hiển thị view thì nếu bạn muốn tuỳ chỉnh gì thì hàm này sẽ giúp bạn làm điều đó.

- viewDidAppear:

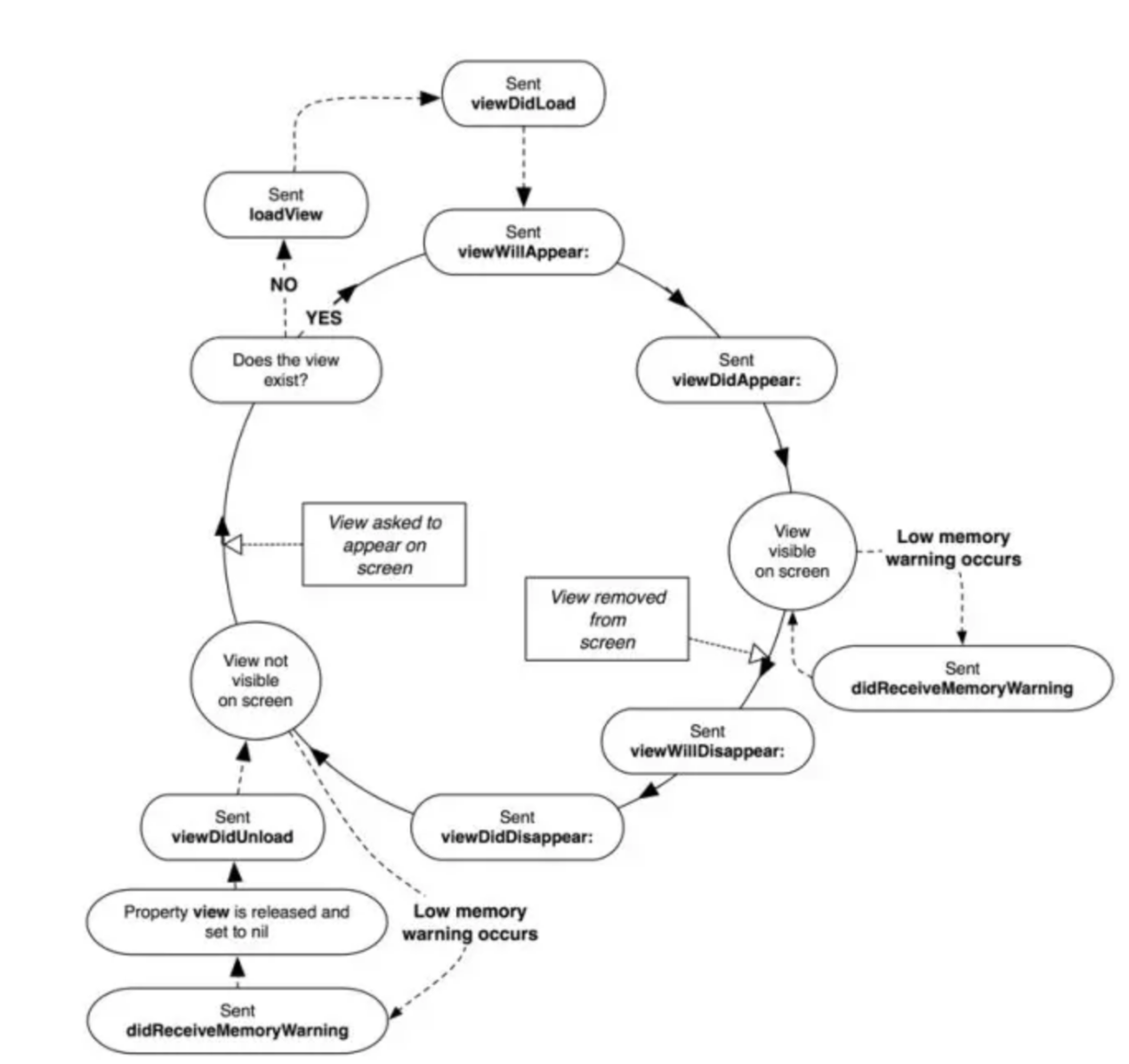
Hàm sẽ được gọi khi một view đã được thêm vài hệ thống view và đã hiển thị lên màn hình.

- viewWillDisappear:

Gọi khi một view đã được ẩn khỏi màn hình và amination khi ẩn view đó.

viewDidDisappear

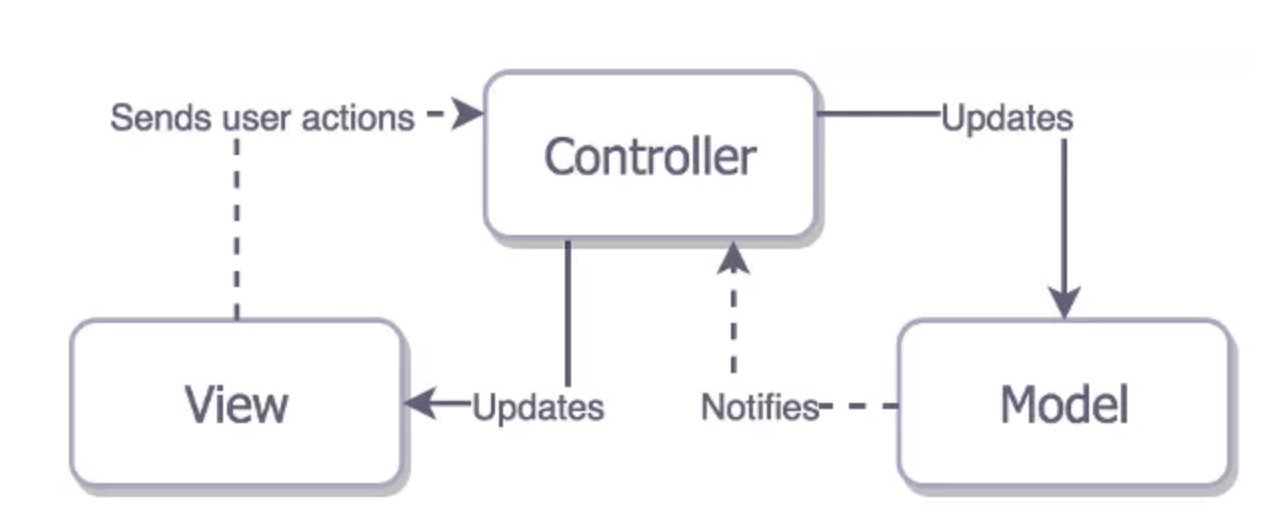
Gọi khi một view đã được ẩn khỏi màn hình.



**VIII. IOS Architecture design:**

<https://viblo.asia/p/ios-architecture-patterns-mvc-mvp-mvvm-ban-da-hieu-ro-chua-bWrZnPyO5xw>

1. **MVC:**



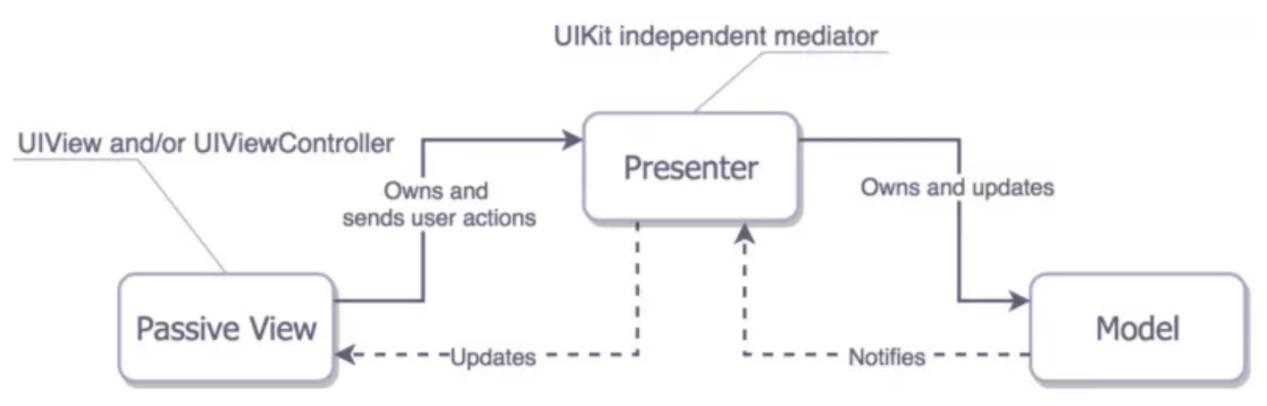
- View và Model sẽ không biết về nhau.

- Controller: Là một class trung gian giữa View và Model,

xử lý data, logic giữa các view, chuyển view, update data, sẽ được xử lý ở Controller.

=> Controller sẽ fai đảm nhận rất nhiều thứ.

1. **MVP:**

****

- View (V) ở đây gọi là Passive (Bị động) View. Chúng là UIView hoặc UIViewController.

- Present (P): ở đây sẽ là các UIKit independent mediator - Là class trung gian độc lập, không phụ thuộc vào cái nào

- Views sẽ chịu trách nhiệm nhận sự kiện (events) từ UI, sau đó gọi Presenter khi cần thiết. Trên thực tế, Presenter chịu trách nhiệm cập nhật View khi có dữ liệu mới được trả về từ Model.

- MVP ko giống MVC:

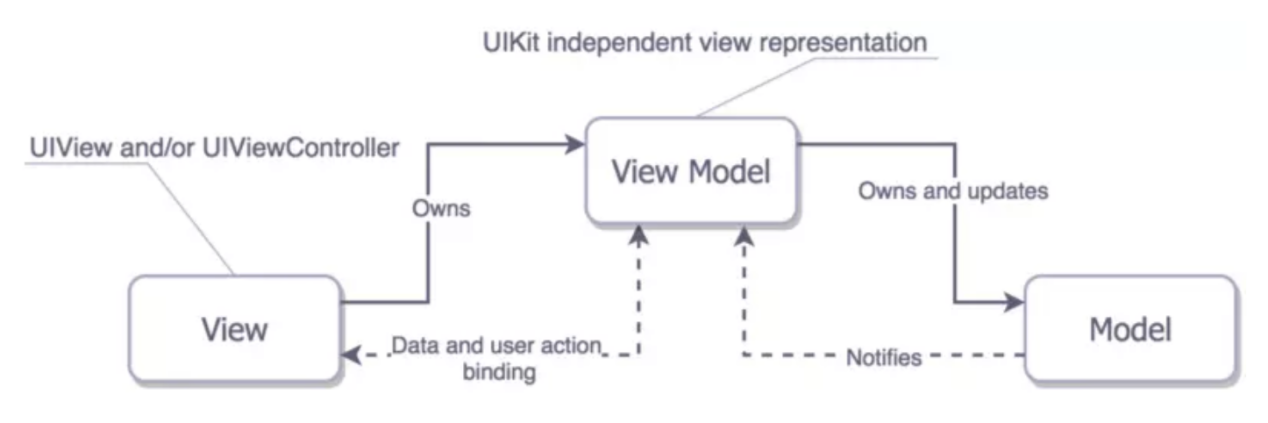
View liên kết chắc chẽ với Controller, Trong khi đó Presenter lại là class trung gian KHÔNG làm bất cứ điều gì với lifecycle của viewController.

View có thể khởi tại giả định dễ dàng do đó không có layout code trong Presenter nhưng nó có trách nhiệm cập nhật View khi có data mới hoặc trạng thái view mới.

1. **MVVM:**

<https://medium.com/swift-india/mvvm-1-a-general-discussion-764581a2d5d9>

<https://medium.com/swift-india/mvvm-2-a-deep-tour-3c8d9fa0cd53>



* Nó khả giống với MVP:

- MVVM sử dụng ViewController như là View.

- View only contains UI logic: data rendering, navigation, ...

- Không có sự liên kết chặt chẽ giữa View và Model.

* ViewModel là gì?

- Về cơ bản, nó là một class trung gian độc lập (UIKit independent) đại diện cho View và trạng thái của nó.

- ViewModel lắng nghe những thay đổi từ Model và tự cập nhật chính nó với những thay đổi từ Model. Vì chúng ta có 1 binding giữa View và ViewModel.

* Binding:

- Bindings khỗng có sẵn trong iOS toolbox. Tuy nhiên chúng ta có KVO (Key value observing) và notification nhưng chúng không thuận tiện như Bindings.

- Chúng ta có 2 lựa chọn để có thể thực hiện Bindings

- Một trong những thư viện bindings dựa vào KVO như RZDataBinding hay SwiftBond

- Functional reactive programing tốt nhất là ReactiveCocoa, RxSwift hay PromiseKit

Thường dùng ReactiveCocoa.

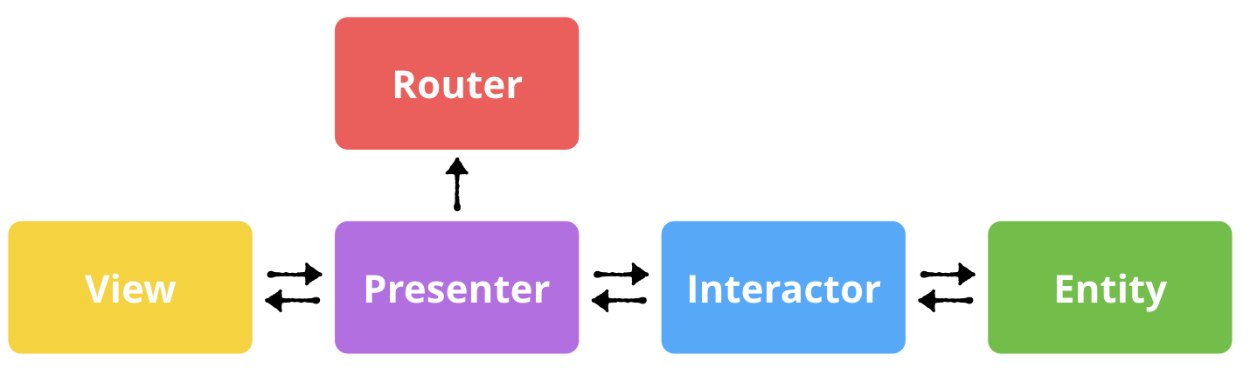
1. **VIPER:**

<https://viblo.asia/p/gioi-thieu-ve-kien-truc-viper-OEqGj6QPG9bL>

<https://www.kodeco.com/8440907-getting-started-with-the-viper-architecture-pattern>

<https://betterprogramming.pub/viper-design-pattern-step-by-step-with-example-project-247b3d101a9e>

Simple example: <https://betterprogramming.pub/viper-design-pattern-step-by-step-with-example-project-247b3d101a9e>



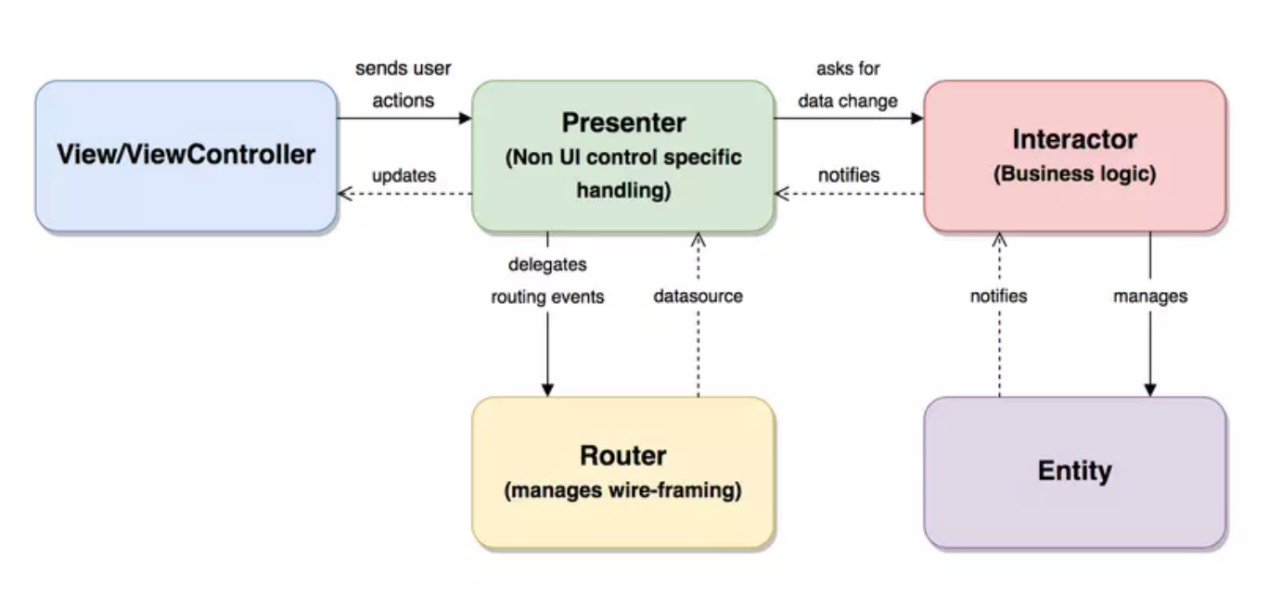
- VIEW : hiển thị giao diện dựa trên hướng dẫn của PRESENTER đồng thời tiếp nhận input và truyền tới PRESENTER. Là các ViewController, nhưng chúng bị động, chỉ chờ để nhận lệnh xử lí từ Presenter. Ko chứa logic xử lý giao diện, cái này sẽ được xử lý ở presenter.

- PRESENTOR : Chứa view logic để hiển thị ( nhận kết quả từ interactor), nhận user input và request tới interactor. Presenter nhận Input từ các interactor và update UI, đồng thời gửi request tới các interactor. It also asks the router/wireframe for navigation. We define all view operations in the presenter in the interface of the view. The purpose of this is to be able to access from the presenter.

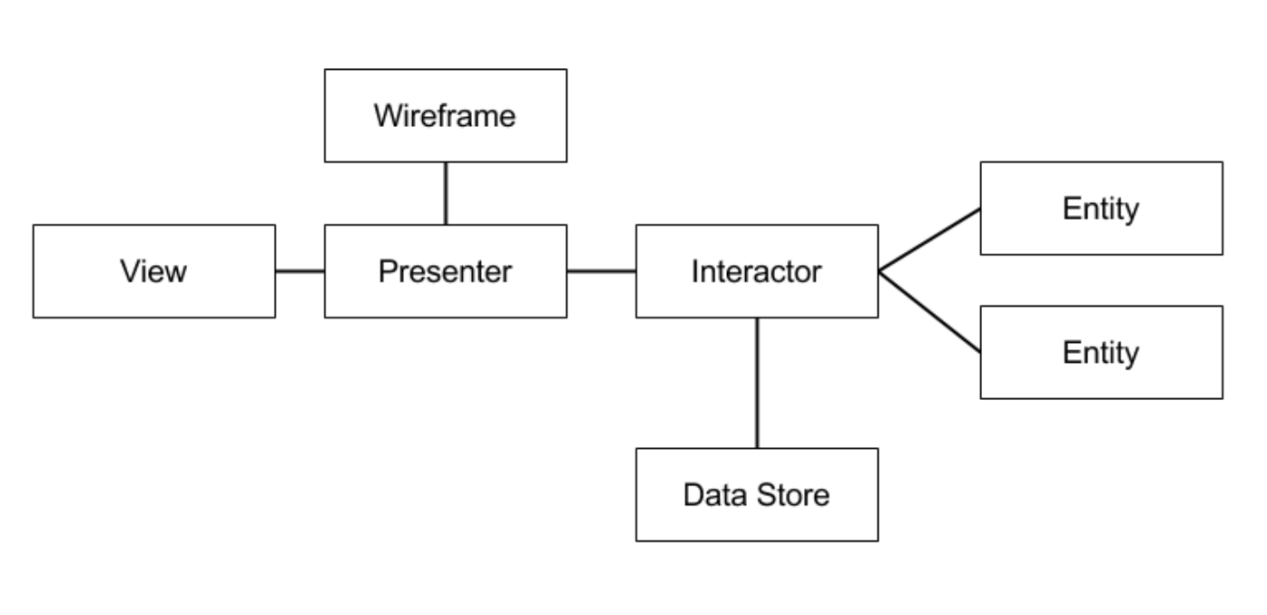
- INTERACTOR : Chứa các business logic tùy theo use case tương ứng. công việc thực hiện trên Interactor hoàn toàn không liên quan gì tới UI, và chỉ tương tác với dữ liệu.

- ENTITY : Các model object được sử dụng bởi interactor.

- ROUTING : chính là wireframe, điều khiển việc hiển thị các màn hình. Chỉ thực hiện khởi tạo các màn hình và điều khiển việc hiển thị, dịch chuyển các màn hình. It has all navigation logic for describing which screens are to be shown when. It is normally written as a wireframe.



- Chia project thành các layer tương ứng:



**IX. Data Binding techniques:**

<https://joelkingsleyr.com/2022/06/10/data-binding-techniques-in-mvvm-swift-uikit/>

<https://fitzafful.medium.com/data-binding-in-mvvm-on-ios-714eb15e3913>

*- Các cách để xử lý bất đồng bộ async trong ios: NotificationCenter, closure, Delegate pattern, Grand Cental Dispatch, RxSwift, Combine*



1. **Delegate pattern:**

<https://viblo.asia/p/delegation-pattern-trong-swift-m68Z07XAKkG>

- là một messaging design pattern trong Swift

- sử dụng trong việc giao tiếp 1-1 giữa những object, tận dụng protocol trong Swift để ủy thác cho một object.

1. **Observable:**

<https://viblo.asia/p/observable-la-gi-cach-tao-ra-observable-bWrZnrRbZxw>

<https://fitzafful.medium.com/data-binding-in-mvvm-on-ios-714eb15e3913>

- An Observable is an entity that publishes its current value whenever its value changes.

- Observable dựa trên concept của reactive

- Các thư viện dùng observable: RxSwift, Combine, tự tạo class Observable.

- Có thể dùng RxSwift.

- Apple cũng có 1 thư viện native reactive là Combine, support từ ios 13+.

- Nếu app cần support ios 12 đổ xuống thì dùng thư viện của bên thứ 3: RxSwift.

- Tự tạo 1 class Observable: see in <https://fitzafful.medium.com/data-binding-in-mvvm-on-ios-714eb15e3913>

1. **Notifications:**

- will use NotificationCenter.

1. **Completion Handler:**

<https://viblo.asia/p/cach-su-dung-atescaping-va-atnoescape-trong-closures-jvEla4zNZkw>

- is a function that calls back when a task completes. So it’s called Callback function.

- A closure is passed as an argument to a function. When this function completes performing the task, it executes the closure.

- @noescape:

+ la mặc định khi tạo closure.

+ Khi bạn gọi 1 function, việc xử lý được thực thi trong function và trả về kết quả. Khi việc thực thi kết thúc, closure sẽ được remove khỏi memory.

+ Được dùng khi bạn muốn gọi function trả về kết quả ngay lập tức sau khi gọi và không quan tâm tới việc quản lý memory.

+ Khong can su dung [weak self]

+ Lifecycle: Gọi function có closure >> Xử lý trong function >> Chạy closure >> Return về trình biên dịch.

- @escaping closures:

+ Khi bạn gọi 1 function, closure được giữ để được thực thi sau và body của function được thực thi, trả về trình biên dịch.

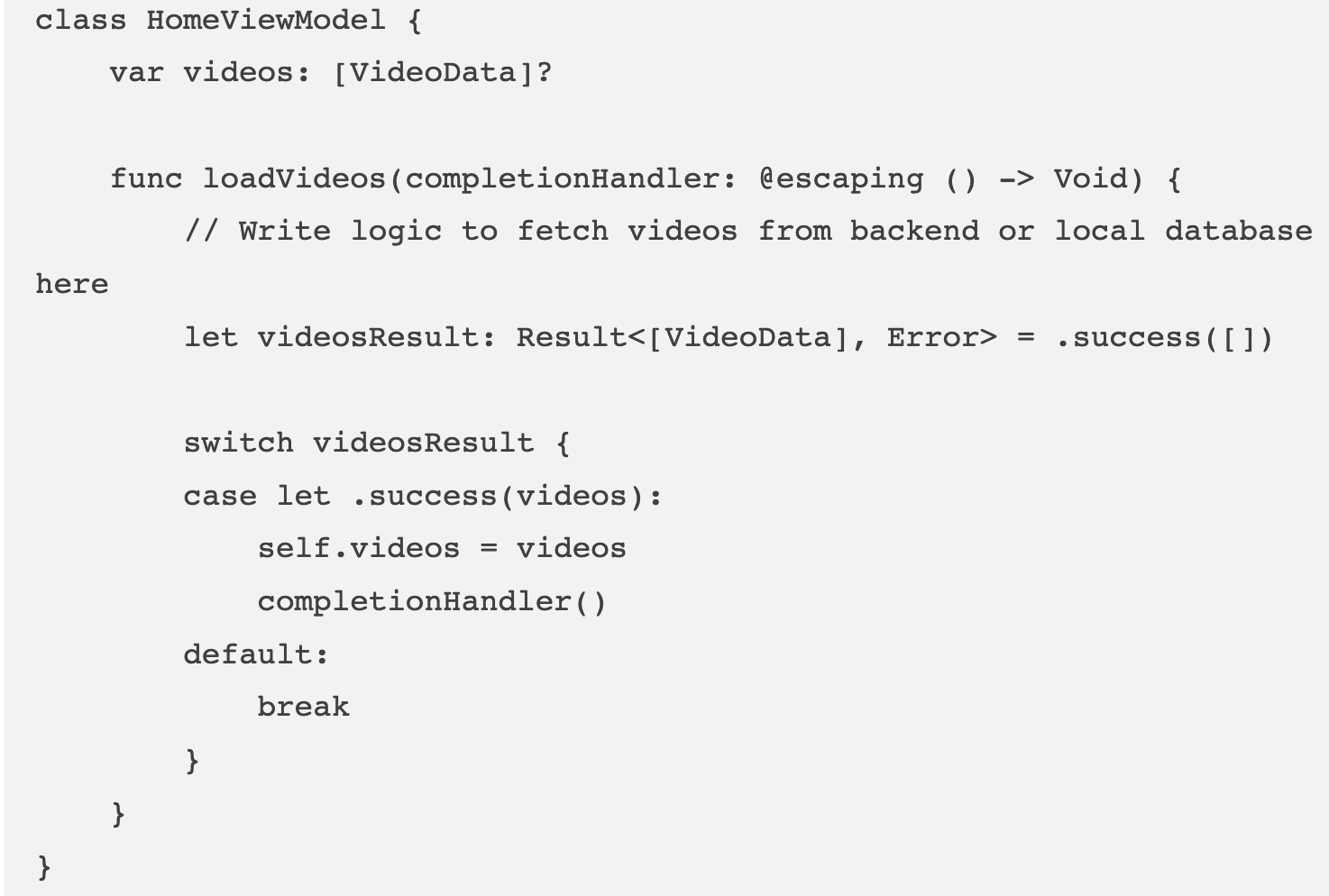
+ Khi function thực thi kết thúc, closure tồn tại trong bộ nhớ, cho đến khi closure được thực thi.

+ Storage: Khi bạn cần phải giữ closure lưu trữ tồn tại trong bộ nhớ, đợi hoàn thành việc nào đó mới thực thi closure và trả về kết quả. (Giống như chờ phản hồi API).

+ Asynchronous: Khi bạn đang thực hiện closure không đồng bộ trên hàng đợi, hàng đợi sẽ giữ closure trong memory cho bạn, để được sử dụng trong tương lai.

+ phai su dung su dung [weak self].

+ Lifecycle: Gọi function có closure >> Xử lý trong function >> Function thực thi closure asynchronous hoặc storage >> Return về trình biên dịch.



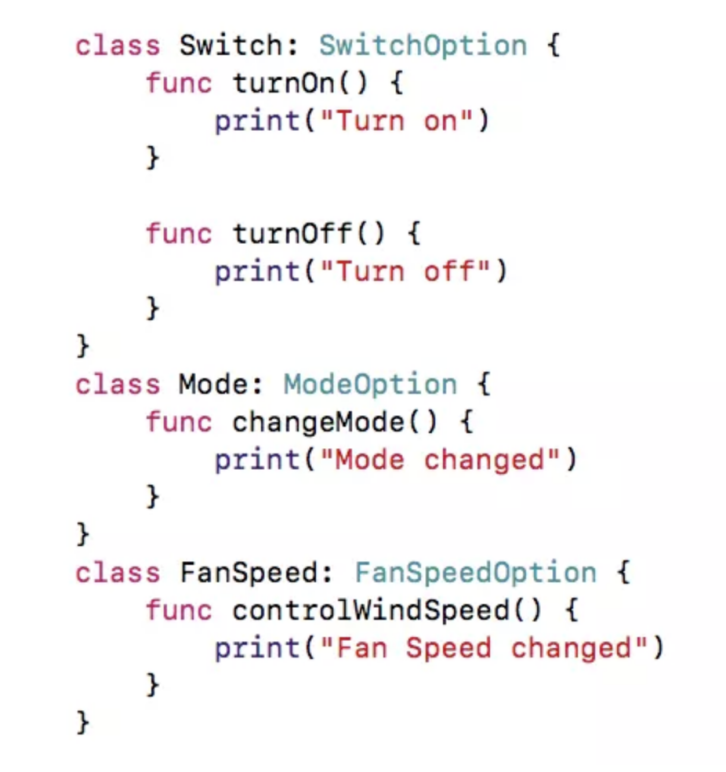
**X. SOLID in ios:**

<https://viblo.asia/p/solid-pattern-trong-swift-Az45bz7N5xY>

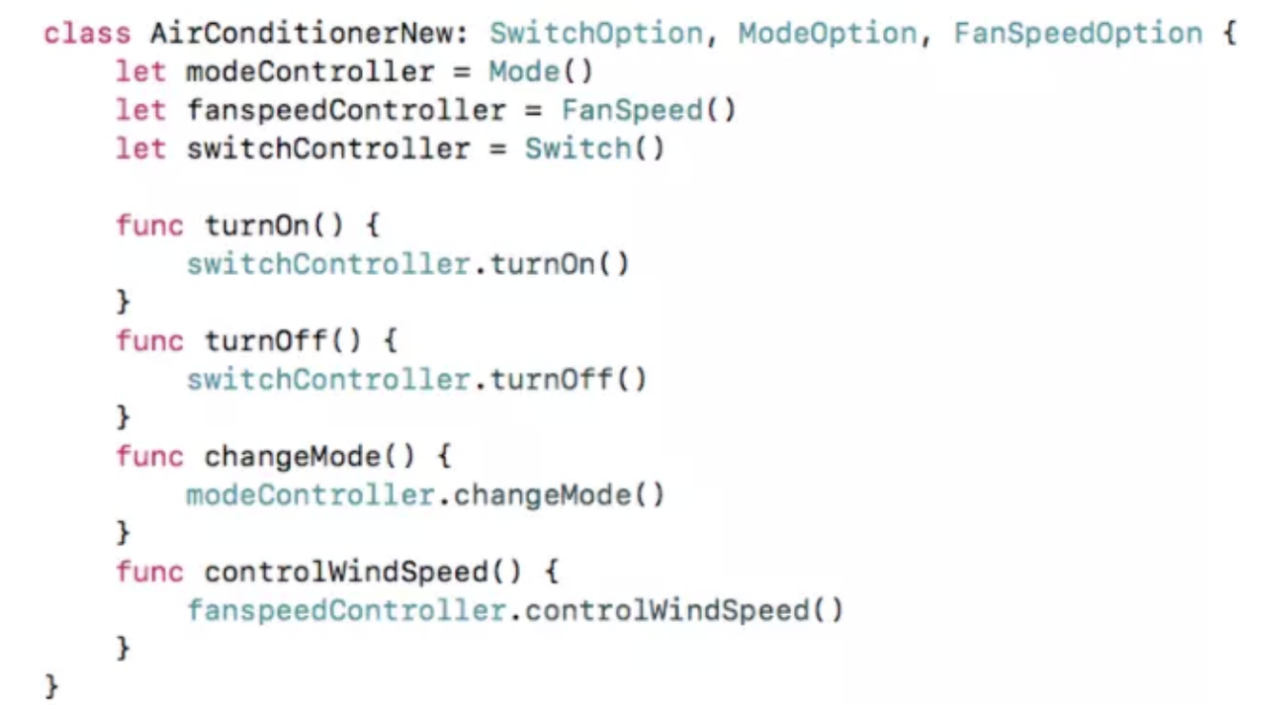
1. **Single Responsibility:**

- Mỗi class chỉ nên có 1 chức năng duy nhất.

- ta có thể tạo 3 class với 3 nhóm chức năng như sau:



- Cuối cùng trong class AirConditioner ta tạo các thực thể để sử dụng từng chức năng như sau:

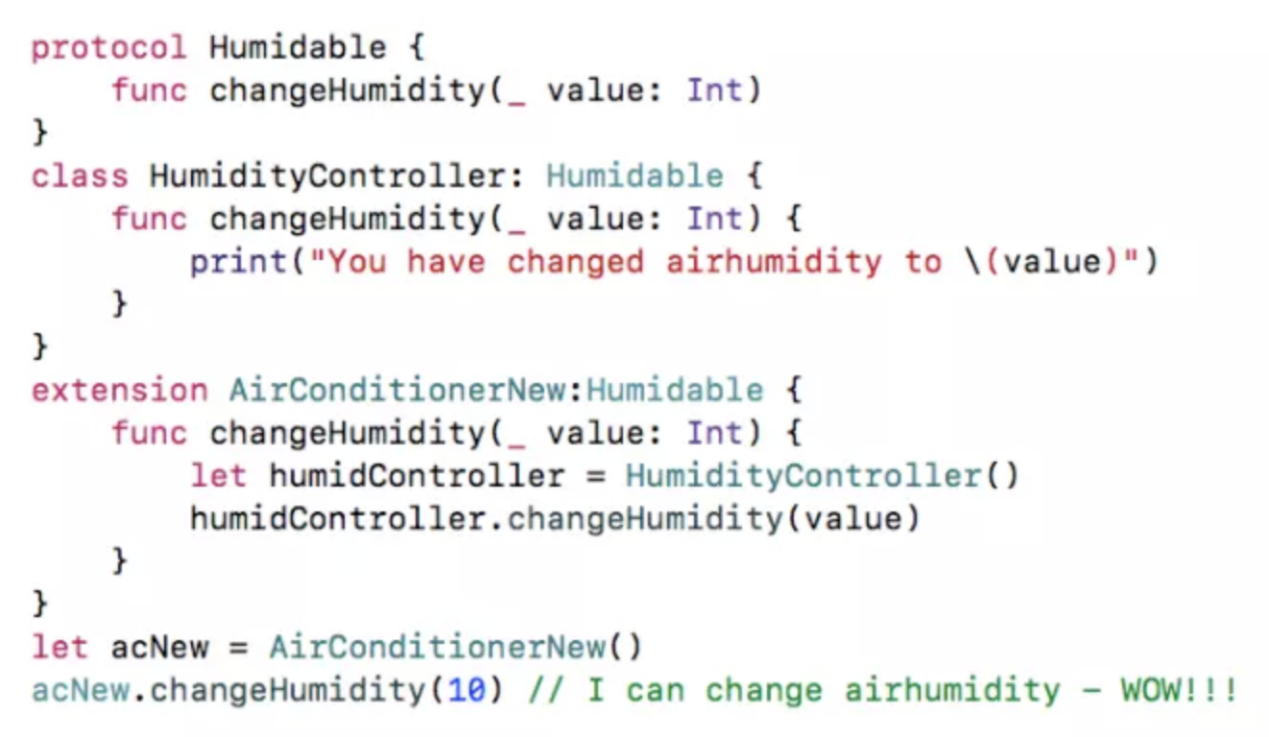


1. **Open/ close principle:**

- là các class cần phải được thiết kế phải mở đối với việc mở rộng nhưng đóng đối với các thay đổi.

- Quay trở lại ví dụ cũ. Giả sử chúng ta muốn thêm vào class AirConditionerNew một chức năng mới là hướng gió. Nếu ta sửa lại phần code trong class cũ điều này vi phạm nguyên tắc của SOLID vì vậy ta có giải pháp như sau: thêm vào 1 property nhưng ko cần phải sữa lại class AirConditionerNew bằng việc sử dụng extension.

- extensions được phép confirm protocol. Sau đó extensions implement tất cả function của protocol nếu classs không làm thế.



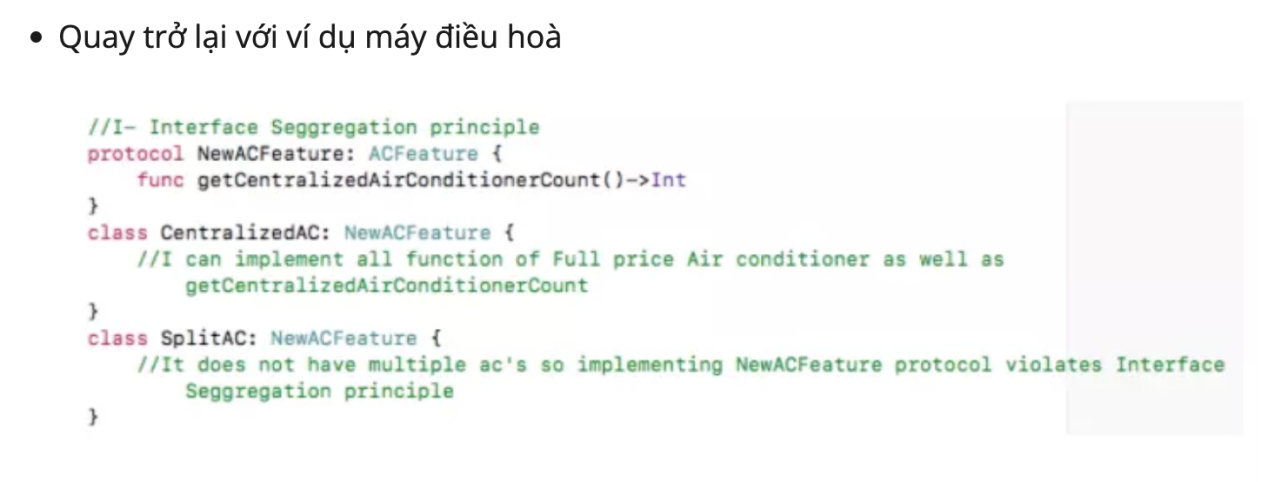
1. **Lislov substitution principle:**

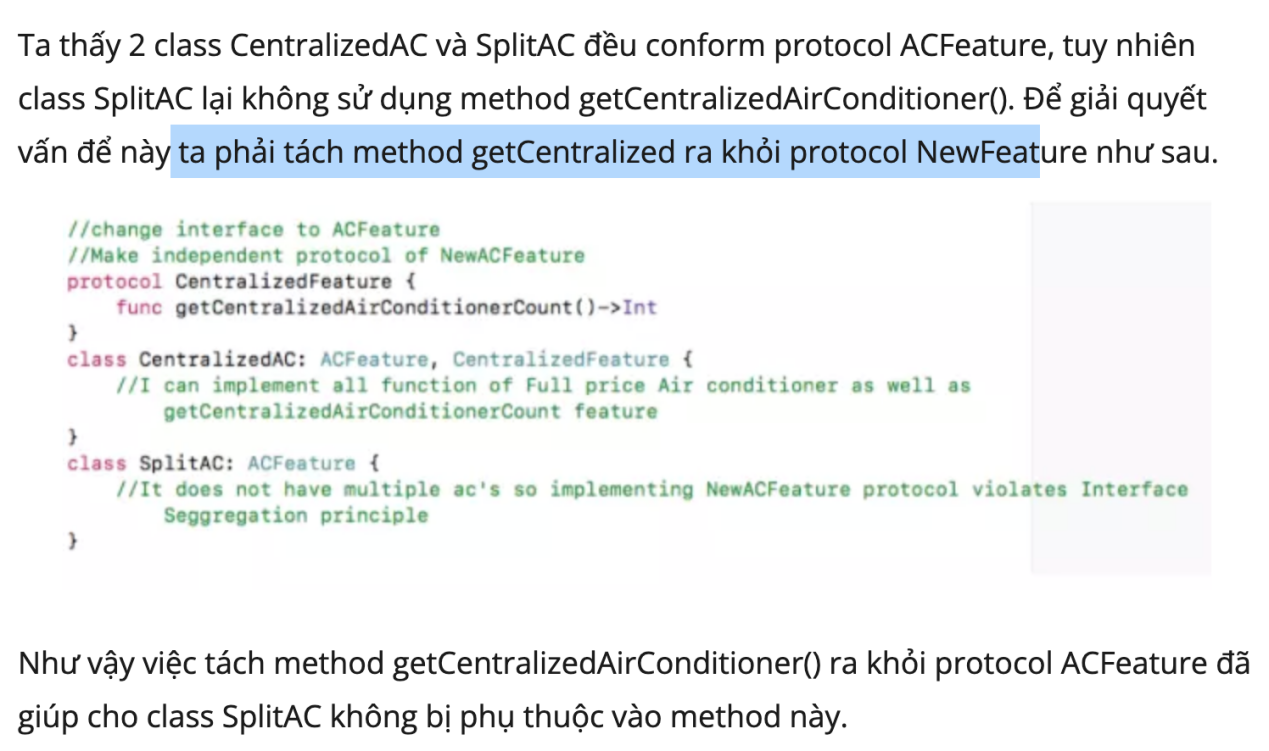
- Nếu S là một kiểu dữ liệu con của T. Thì thực thể của T có thể được thay thế bằng thực thể của S.

- Class A là con của Class B, thì A phải sở hữu hoàn các thuộc tính và chức năng của B. Như vậy A có thể thay thế cho B.

1. **Interface Seregation:**

- Nguyên tắc này phát biểu: "một client không nên bị force phụ thuộc vào một method mà nó không sử dụng".





1. **Dependency Inversion:**

- Các module bậc cao không nên phụ thuộc vào module bậc thấp. chúng nên phụ thuộc vào 1 lớp trừu tường.



Trong ví dụ trên, class CoversationData là module bậc cao, CoreDataController là module bậc thấp. Ta thấy class ConverstionData đang chưa một thực thể CoreDataController. Có thể thấy module bậc cao đã bị phụ thuộc vào module bậc thấp vì vậy sẽ gây khó khăn nếu muốn thay module CoreDataController bằng module khác. Để giải quyết ta tạo thêm một class để trừu tượng để giảm sự phụ thuộc của 2 bên.



Như vậy, thay vì chứa một thực thể CoreDataController thì class ConverstionData chỉ chứa 1 thực thể trừu tượng DataBase. Do đó sự phụ thuộc của 2 module sẽ giảm đi đáng kể. Trong trường hợp muốn thay module CoreDataController ta chỉ việc tạo một module mới và conform protocol Database.

**XII. SWift:**

<https://docs.swift.org/swift-book/documentation/the-swift-programming-language/thebasics/>

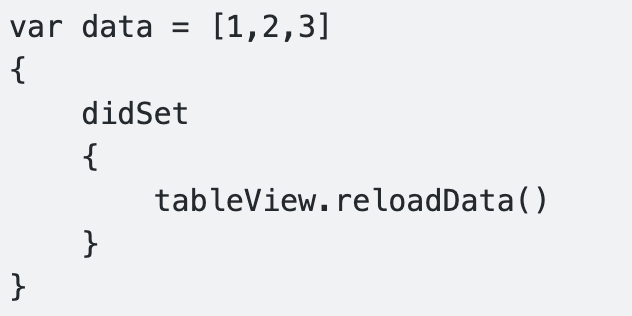
1. **Một số key trong swift:**

**<https://viblo.asia/p/ios-mot-so-key-trong-swift-Az45bW8QKxY>**

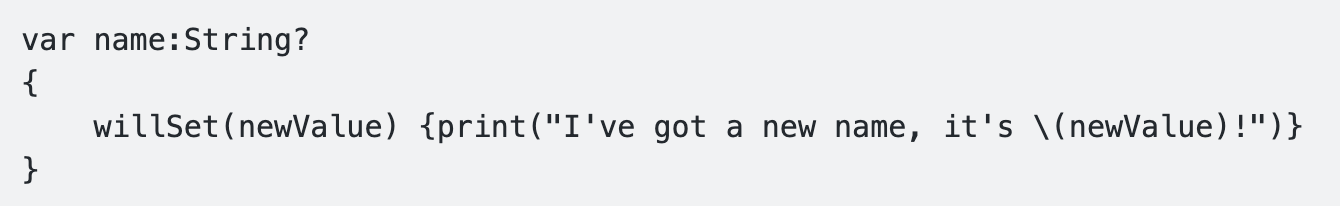
- convenience:

- dynamic: chạy bằng Objective-C runtime và Swift runtime sẽ cho ra kết quả khác nhau, thêm dynamic vào thì coi như nó chạy bằng Objective-C runtime.

- didSet : sau khi gán xong thì nó sẽ thực hiện câu lệnh bên trong didSet.



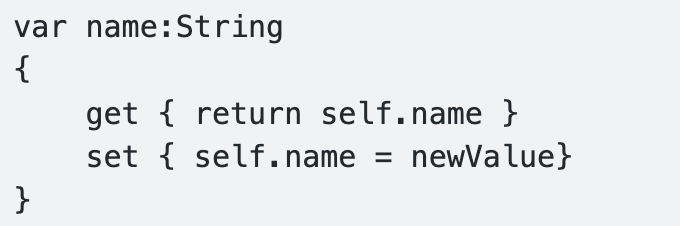
- willSet : willSet thì được gọi ngay trước khi biến được gán đâu đó. Nó có newValue là giá trị sẽ được gán đó.



- Final: Ngăn không cho kế thừa nữa.

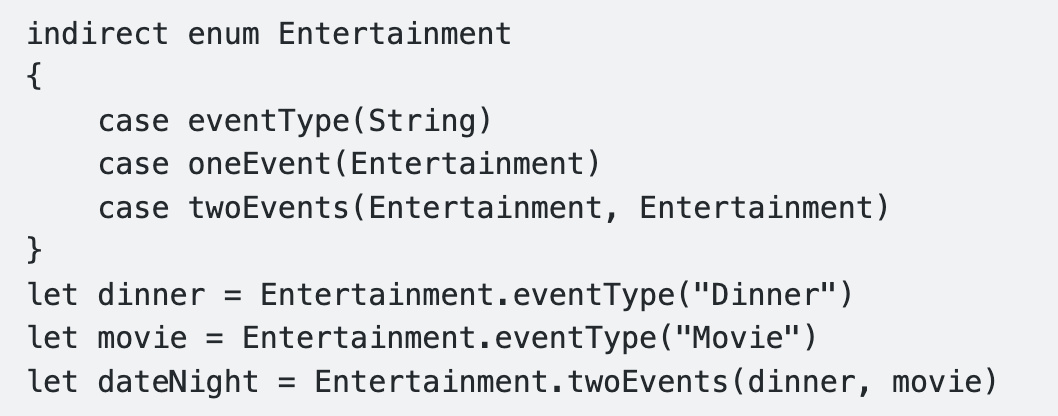


- get : dùng để trả về giá trị 1 giá trị nào đó.



- set : trong hàm set có thể làm gì thì làm, set giá trị cho chính property chứa nó hoặc property khác cũng được.

- indirect : Chỉ ra enum có 1 case khác liên quan tới giá trị của 1 hoặc nhiều case trong enum đó.



- lazy : biến nào có lazy thì nó chỉ được tính toán khi nó được gọi ra, giúp tiết kiệm bộ nhớ hơn.

- mutating : Cho phép thay đổi giá trị của thuộc tính của struct hoặc enum

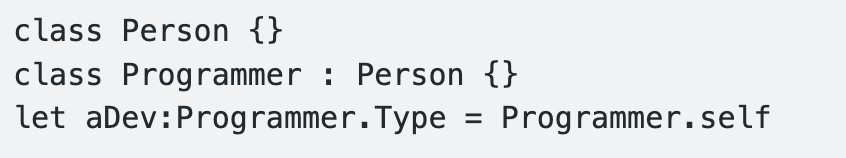
- nonmutating : Chỉ ra rằng hàm setter không thay đổi được instance chứa nó.

- optional : Sử dụng để khai báo hàm optional trong protocol. Dùng từ khoá @objc

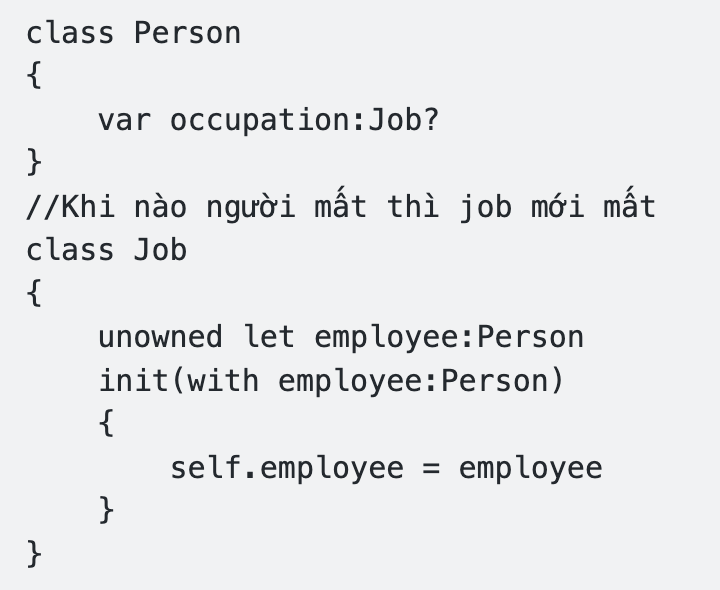
- override : Chỉ ra rằng lớp con đang ghi đè lên biến hoặc hàm của lớp cha.

- required : đảm bảo rằng mọi lớp con phải thực thi hàm khởi tạo cho trước.

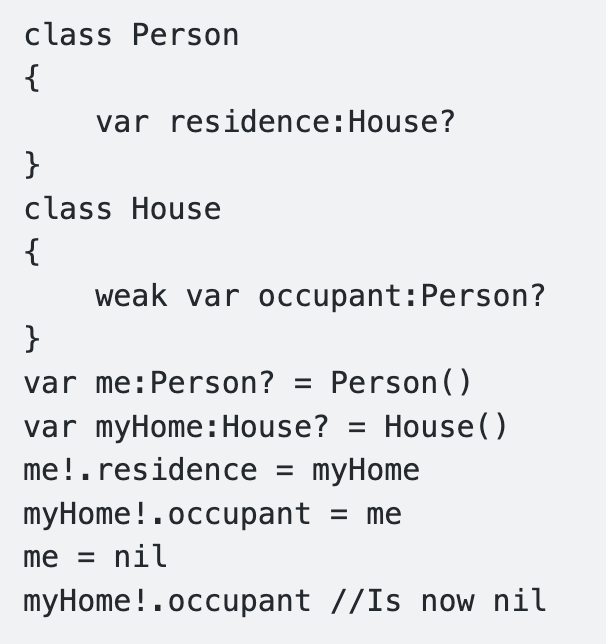
- Type : có thể là class type, struct type, enum type hoặc protocol type.



- unowned : Cho phép 1 instance tham chiếu tới 1 instance khác mà không làm tăng reference count, đảm bảo rằng instance đã tham chiếu tới có lifetime bằng hoặc lâu hơn chính nó.



- weak : Cho phép 1 instance tham chiếu tới 1 instance khác mà không làm tăng reference count, nhưng instance đã tham chiếu kia có lifetime ngắn hơn (có thể bị huỷ trước)



- Throws: Một func có thể được định nghĩa là throwing, về cơ bản có nghĩa là nếu có gì sai, nó có thể thông báo có lỗi đã xảy ra.

Để bắt lỗi này, chúng ta cần triển khai một câu lệnh gọi là do-catch.

- is, as: Type casting trong Swift:

+ is: được sử dụng để kiểm tra kiểu của giá trị.

+ as: được sử dụng để chuyển đổi một giá trị sang một kiểu khác.

+ as?, as!: Một hằng hoặc biến của một type class cụ thể có thể refer tới một subclass, trường hợp này bạn có thể downcast tới type subclass.

+ as?: trả về một optional value của kiểu mà bạn đang cố gắng downcast tới

+ as!: cố gắng downcast và force-unwrap kết quả.

1. **Memory Management:**

<https://viblo.asia/p/co-phai-khi-nao-cung-can-dung-weak-self-trong-closures-gAm5y4bLldb>

1. **Unowned, weak:**

- If you use self in closure -> self will exist in all life-cycle of the scope. Nếu self vẫn là strong reference -> dẫn đến strong reference cycle. Vì vậy để tránh strong reference cycle cần sử dụng [weak self] or [unowned self]

- [unowned self]: nếu object đó nil -> app sẽ bị crash.

1. **Escaping and non-escaping closure:**

- Non-escaping closures thực thi code trong scope của nó một cách tức thì và không có khả năng lưu trữ hay sử dụng sau đó. -> không gây ra reference cycle -> không cần dùng weak or unowned.

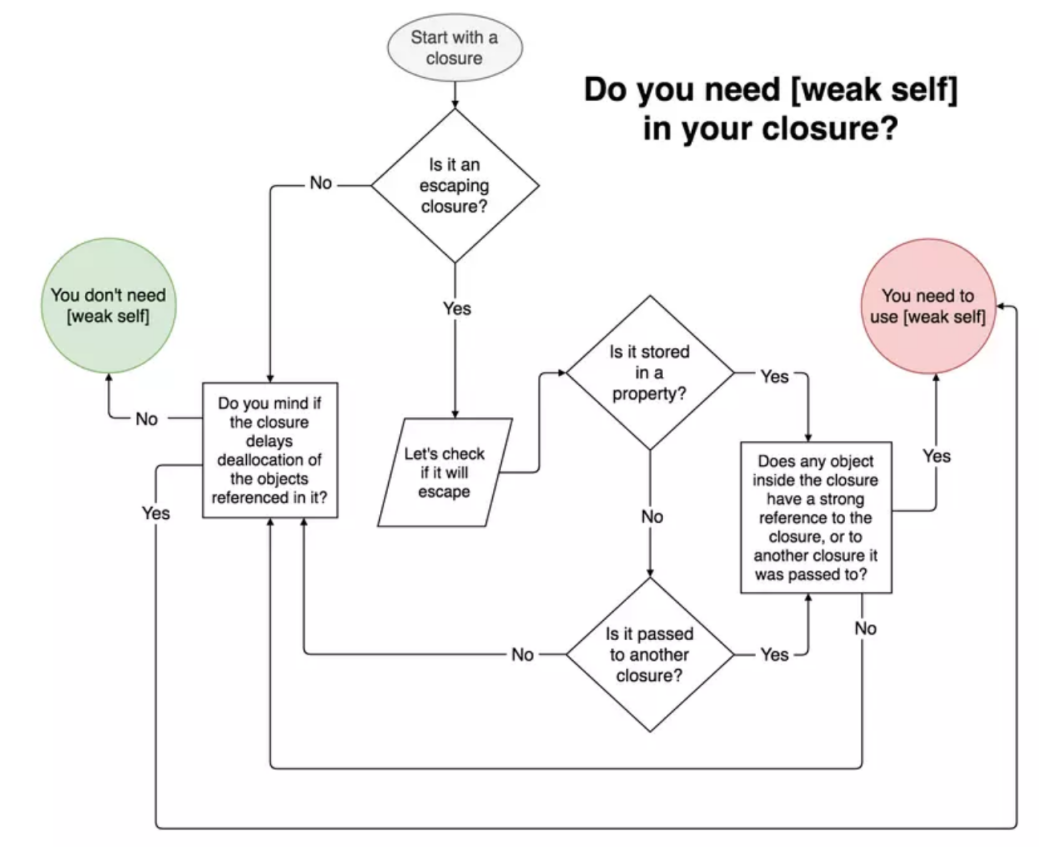
- Escaping closures thì có thể lưu trữ nó vào 1 biến hoặc 1 closure khác và có thể thực thi nó trong tương lai.

- Escaping closures gây ra reference cycle khi:

+ Closures được lưu trữ vào 1 biến hoặc 1 closures khác.

+ Có sử dụng self để tham chiếu trong closure.

=> phải dùng weak or unowned.



1. **Delay deallocation:**

- Delay deallocation không phải nguyên nhân trực tiếp dẫn đến memory leak, nhưng nó sẽ gây ra một vài điều không mong muốn: khi VC bị dismiss nhưng closure vẫn đang bị block thì VC sẽ không được deinit ngay lúc đó -> Gây ra sự delay không mong muốn -> có thể dùng weak self để tránh điều này xảy ra.

1. **‘guard let self = self’ and optional chaining:**

- Khi sử dụng [weak self] là chúng ta đã tạo 1 biến optional -> khi sử dụng chúng ta cần unwrap hoặc sử dụng optional chaining.

- unwrap: bằng guard let

- optional chaining: self?

- Nếu như trong closure có một task tốn thời gian và có thể sinh ra delay dealloc thì việc sử dụng guard let ngay từ đầu cũng không thể tránh được điều đó.

- sử dụng guard let là chúng ta so sánh self với nil, nếu không nil thì chúng ta sẽ tạo ra một strong reference để sử dụng trong scope --> nguyên nhân có thể gây ra delay deallocation.

- dùng optional chaining thì chúng ta sẽ so sánh self với nil ở từng dòng lệnh, nếu nil sẽ bỏ qua --> không tạo ra strong reference ở đây.

1. **Grand central dispatch**

- GCD thường không gây ra reference cycles nếu như nó không được lưu trữ để dùng sau đó.

- nếu bạn lưu trữ GCD vào 1 biến như sau thì sẽ gây ra leak.

- UIView.Animate and UIViewPropertyAnimator: không gây ra reference cycles nếu như nó không được lưu vào 1 biến.