

iOS

inside

- Bounds 와 Frame 의 차이점을 설명하시오.

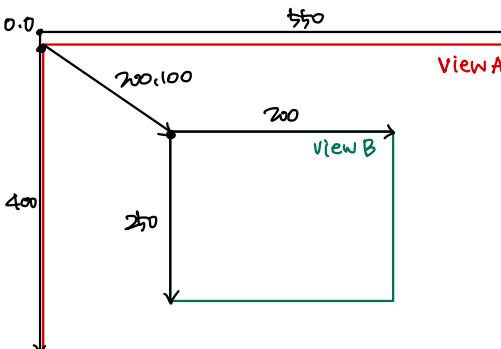
• **Bounds** of UIView is the rectangle, expressed as location (x,y) and size ($width, height$) relative to its own coordinate system.

• **Frame** of an UIView is the rectangle, expressed as a loc (x,y) and size ($width, height$) relative to the superview it is contained.

ex) size 100×100 (w,h) positioned at $25, 25$ of its superview

bounds. origin. x : 0
bounds. origin. y : 0
bounds. size. width : 100
bounds. size. height : 100

frame. origin. x : 25
frame. origin. y : 25
frame. size. width : 100
frame. size. height : 100



View A frame :
origin : (0,0)
size : 550×400

B frame
($200,100$)
 200×250

View A Bounds:
origin : (0,0)
size : 550×400

B Bounds
(0,0)
 200×250

- 실제 디바이스가 없을 경우 개발 환경에서 할 수 있는 것과 없는 것을 설명 하시오.

Hardware: 가속도 센서, 가령계 센서, 주변광 센서, GPS 센서 이용 X
마우스로 Simulator touch 하기 X / zoomIn/out 가능 X
카메라, 마이크, 전화기능 사용 불가능.

API: Apple의 push notification 받기와 보내기 불가능.
사진, 연락처, 카alendar 앱에서 초기화하기 위해 개인 정보 알림 지원 X
hand off 가능 X / message UI 가능 X

ETC: 디바이스 성능이 아이폰의 성능보다 뛰어나 CPU나 메모리 부담 확인 불가
비트워크 속도 테스트 측정 X / Face ID 인식 X

- 앱이 foreground에 있을 때와 background에 있을 때의 제약사항

Not Running

application (_ : willFinishLaunchingWithOptions) : 앱 실행 준비 method
필요한 주요 객체 생성, 앱 실행 준비가 끝나기 전에 호출

applicationDidFinishLaunching (_ :) : 앱 실행을 위한 모든 준비가 끝난 후 화면이
사용자에게 보여주기 직전 호출 (초기화 코드 작성)

applicationWillTerminate (_ :) : 앱 종료되기 전에 호출
(메모리 회수와 함께 suspended 상태에 있는 앱이
종료될 때나 background 상태에서 사용자에게 의해
종료될 때나 오류로 인해 종료 시 호출 X)

Inactive

sceneWillEnterForeground (_ :) : 앱이 백그라운드나 대리터에서 foreground로
들어가기 직전에 호출 (비활 \rightarrow 활성화)

sceneWillResignActive (_ :)

Active

sceneDidBecomeActive (_ :) : 앱이 비활 상태에서 활성상태로 진입하고 난 후 호출
실제로 사용자가 전에 준비할 수 있는 코드를 작성할 수 O

Background

sceneDidEnterBackground (_ :) : 앱이 백그라운드 상태로 들어갔을 때 호출
Suspended 상태가 되기 전 Data 저장.

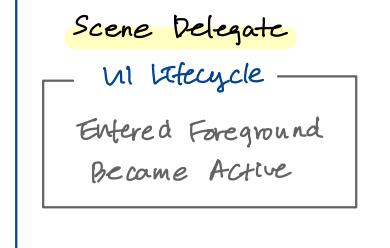
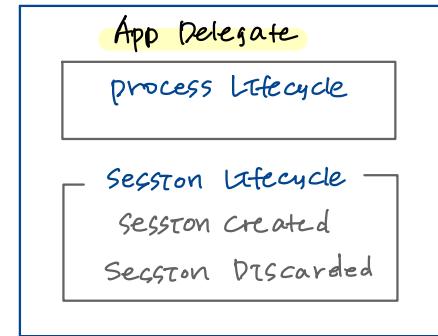
Suspended

파고 호출되는 메소드는 없으며 background 상태에서 특별한 작업이 없을 때

1. Not Running : 앱이 실행되지 않았거나 완전히 종료되었을 때
2. Inactive : 앱이 실행되면서 foreground에 진입하지만 event X
3. Active : 앱 실행중 foreground, event O
4. Background : 앱이 Background, 다른 앱으로 전환되었거나 Home Button.
5. Suspend : Background에서 특별한 작업이 없을 경우 전환 state.

ipad \rightarrow 편리

- scene delegate에 대해 설명하시오.



iOS 13부터 UI lifecycle에 대한 부분을 SceneDelegate가 함.

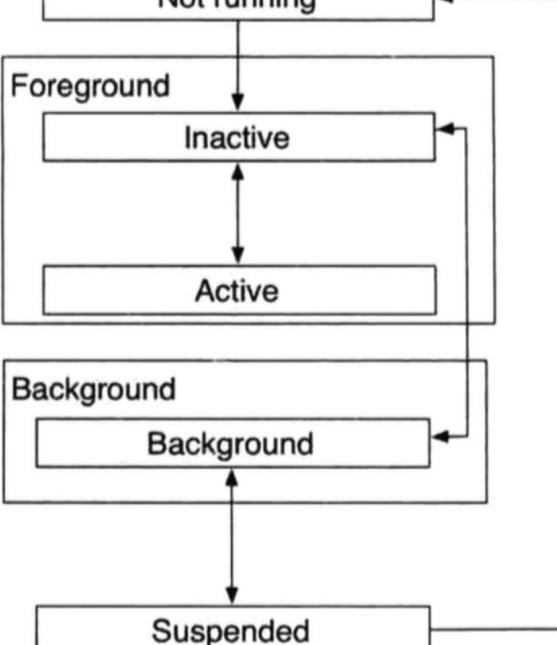
Scene?

UIKit은 UIWindowScene 객체를 사용하는 앱 UI의 각 인스턴스를 관리합니다. **Scene**에는 UI의 하나의 인스턴스를 나타내는 windows와 view controllers가 들어있습니다. 또한 각 scene에 해당하는 UIWindowSceneDelegate 객체를 가지고 있고, 이 객체는 **UIKit과 앱 간의 상호 작용을 조정**하는데 사용합니다. Scene들은 같은 메모리와 앱 프로세스 공간을 공유하면서 서로 동시에 실행됩니다. 결과적으로 하나의 앱은 여러 **scene**과 **scene delegate** 객체를 동시에 활성화할 수 있습니다.
(Scenes - Apple Developer Document 참고)

UI의 상태를 알 수 있는 **UILifeCycle**에 대한 역할을 **SceneDelegate** 가 하게 됐죠! 역할이 분리된 대신 **AppDelegate**에서 **Scene Session**을 통해서 **scene**에 대한 정보를 업데이트 받는데요! 그럼 **Scene Session**은 뭘까요??

Scene Session?

UISceneSession 객체는 scene의 고유의 런타임 인스턴스를 관리합니다. 사용자가 앱에 새로운 scene을 추가하거나 프로그래밍적으로 scene을 요청하면, 시스템은 그 **scene**을 추적하는 **session** 객체를 생성합니다. 그 **session**에는 고유한 식별자와 **scene**의 구성 세부사항(**configuration details**)가 들어있습니다. UIKit은 session 정보를 그 **scene** 자체의 생애(life time)동안 유지하고 app switcher에서 사용자가 그 **scene**을 클로징하는 것에 대응하여 그 **session**을 파괴합니다. session 객체는 직접 생성하지 않고 UIKit이 앱의 사용자 인터페이스에 대응하여 생성합니다. 또한 위 3번에서 소개한 두 메소드를 통해서 UIKit에 새로운 **scene**과 **session**을 프로그래밍적 방식으로 생성할 수 있습니다.
(UISceneSession - Apple Developer Document 참고)



Q: 앱이 Inactive 상태가 되는 시나리오를 설명하세요.

앱이 Background 상태에서 Foreground로 진입할 때 Inactive \rightarrow active 상태가 되고 앱이 foreground에 진입하면서 Inactive \rightarrow active 상태가 됨.

Method 순서는

Not running 상태에서 앱을 실행하고 foreground로 전환에 applicationDidEnterLaunching이 호출되고 foreground 진입시 sceneWillEnterForeground가 호출되어 액티브 상태.

유저가 백그라운드 전환했을 때 sceneDidEnterBackground가 호출되고 다시 foreground 들어갈 때 sceneDidBecomeActive 호출

application life cycle

- application: `willFinishLaunchingWithOptions`: → Bool
 - intended for initial application setup. Storyboards have already been loaded at this point, but state restoration hasn't occurred yet.

Launch

- application: `didFinishLaunchingWithOptions`: → Bool
 - called when app has finished launching and restored state and do final initialization such as creating UI
- `applicationWillEnterForeground`
 - called after `didFinishLaunchingWithOptions` becomes active again after receiving a phone call or other system interruption. (inactive → active)
- `applicationDidBecomeActive`
 - called after `applicationWillEnterForeground`: to finish up the transition to the foreground

Termination

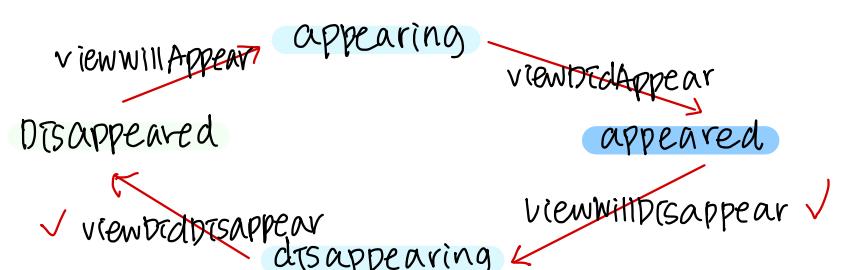
- `applicationWillResignActive`
 - app is about to become inactive (ex. receives phone call, user hits the home button)
- `applicationDidEnterBackground`
 - app enters a background state after becoming inactive. approx. have 5 seconds to run any tasks to backup incase app gets terminated later. (inactive → background)
- `applicationWillTerminate`
 - called when app is about to be purged from memory. final cleanup here

iOS Application States

- ① **not running state**: app has not been launched / or was running but was terminated by the system.
- ② **inactive state**: running foreground but not receiving events.
Stays briefly as it transitions to a different state
Stays: when the user locks the screen or the system prompts the user to respond to phone call + SMS message.
- ③ **active state**: running in foreground and is receiving event normal mode for foreground apps.
- ④ **background state**: in background and executing code.
Enter briefly on their way to be suspended. However, an app that requests extra execution time can remain in this state for some time. Also, an app being launched directly into the background enters this state instead of inactive state.
- ⑤ **suspended state**: remains in memory but does not execute any code.
When a low memory condition occurs, the system may purge suspended apps without notice to make more space for the foreground app

viewController lifecycle

- `loadView`: this is where subclasses should create their custom view hierarchy if they aren't using a nib. Should never be called directly. only override this method when you programmatically create views and assign the root view to the view property.
- `loadViewIfNecessary`: loads the view controller's view if it has not already been set.
- `viewDidLoad`: only called when the view is created and loaded into memory but the bounds for the view are not defined yet. Initialise the objs that v.c is using.
- `viewWillAppear`: view is about to made visible. Default does nothing. This event notifies the v.c whenever the view appears on the screen. The view has bounds that are defined but the orientation is not set.
- `viewWillLayoutSubviews`: first step in the lifecycle where the bounds are finalized. If you are not using constraints or auto layout, update the subviews here.
- `viewDidLayoutSubviews`: this event notifies the v.c that the subviews have been set up. Make any changes to the subviews after they have been set.
- `viewDidAppear`: fires after the view is presented on the screen. Get data from a backend service or database
- `viewWillDisappear`: fires when v.c is about to disappear, dismiss, cover, or hide behind other v.c. Restrict your network calls, invalidate timer or release objects which is bound to that v.c.
- `viewDidDisappear`: last step lifecycle, fires just after the view of presented v.c has been disappeared, dismissed, covered, or hidden



iOS13부터 AppDelegate가 하는 일?

이전에는 앱이 foreground에 들어가거나 background로 이동할 때 앱의 상태를 업데이트하는 등의 앱의 주요 생명 주기 이벤트를 관리했었지만 더 이상 하지 않습니다.

현재 하는 일은

1. 앱의 가장 중요한 데이터 구조를 초기화하는 것
2. 앱의 scene을 환경설정(Configuration)하는 것
3. 앱 밖에서 발생한 알림(배터리 부족, 다운로드 완료 등)에 대응하는 것
4. 특정한 scenes, views, view controllers에 한정되지 않고 앱 자체를 타겟하는 이벤트에 대응하는 것.
5. 애플 푸쉬 알림 서비스와 같이 실행시 요구되는 모든 서비스를 등록하는 것.

입니다.
(UIApplicationDelegate - Apple Developer Document 참고)

1. Protocol / Delegate

Protocol: 선언만 되고 구현되지 않은 method.

기능을 정의하고 method 구현해명 됨.

delegate: 대표 위임의 뜻, 하나의 객체가 모든 일을 처리하는 것이 아니라 처리해야하는 일 중 일부를 다른 객체로 넘기는 것.

textField.delegate = self

- 대리자가 누구인지 알려주는 과정 → "textField 돌아나가는 내가!"
너에게 이벤트가 발생하면 프로토콜에 따라 너에게 응답을 줘라

2. 동기 / 비동기 작업 차이

동기: 작업이 완료 될 때까지 기다린다.

비동기: 백그라운드에서 작업이 완료되면 알림을 받을 수 있다.

3. strong, weak, read only, copy의 차이는?

- strong: 참조 카운트가 증가될 것이고 reference(참조)는 그 객체의 생명주기를 통해 유지될 수 있음.
- weak: 객체에 포인터를 가르키고 있지만 RC를 올리지 X.
부모 객체 관계. 부모는 자식에게 strong ref.
자식은 부모에게 weak ref.
- read only: 초기에 속성 세팅할 수 있지만 변경할 수 X
- copy: 객체가 생성될 때 객체의 값을 복사한다는 의미(value)

4. Completion Handler

작업이 끝났을 때 API 호출하고 싶을 때 사용 (closure)

NSDate?, NSURLResponse?, NSError?로 코드를 받고 void 리턴

5. Frame vs. Bound?

- UIView의 Bound는 고유 좌표 시스템 (0,0)에 연관된 position(x,y)와 size(width, height)로 표현되는 시각형이고,
- UIView의 Frame은 그 부를 담고 있는 parent와 연관된 position(x,y)와 size(width, height)로 표기 시각형.

6. 할수장?

어떤 작업을 수행하기 위해 일련의 명령을 묶을 수 있게 해준다. 코드상에서 사용 가능하게 / 중복되는 명령을 찾는다면, 할수가 반복을 피하게 해줌.

7. Singleton Pattern

- 특정 용도로 객체를 하나만 생성하여, 공용하여 사용하고 싶을 때.
- 장점: 한번의 instance만 생성 → 메모리 방지
 - : 전역 instance로 다른 class들과资源共享
 - : 공통된 객체를 여러개 생성 상황 (cache, threadpool...)
- 단점: Singleton instance가 너무 많은 일을 하거나 데이터 공유↑,
다른 class instance간 결합도 높아져 "개발-폐쇄" 원칙 위배
 - : 수정과 테스트가 어려워짐

8. Observer 패턴이란?

원본 객체의 상태변화를 다른 obj에게 알리겠다.

9. 디자인 Pattern?

MVC : Model - 도메인 데이터의 역할이나 데이터를 조작하는 로직 어플리케이션
접근하는 데이터.

: View - 표현하는 레이어 (GUI)의 역할, UI

: Controller - Model과 View의 중개인, View에서 실행된
(Presenter 사용자 행동에 반응하여 Model을 바꾸고
ViewModel) View에 갱신하는 역할.

MVVM : View와 그 상태의 표현이 독립적인 UIKit.

ViewModel은 Model에서 바뀐 것을 부르고 정상화된 model로
그 자신을 정신.

10. JSON / PLIST의 차이?

- Obj 생성하고 디스크에 serialize
- 느림
- serialize / deserialize
- thread safe X

11. SQLite의 차이?

- 관계형 DB - table마다 schema 정의
- 손쉽 query 작성. → 오류에 mapping
- 쿼리 빠름.

12. Realm

- 오픈소스, 빠르다

13. Generic Type

특정 data type에 의존하지 않는 코드.

14. lazy var

그 변수가 이용되기 전까지는 사용 X

15. Concurrency 동시성.

- Process: 실행되고 있는 한 app의 한 instance
- thread: 코드상의 실행 경로
- multithread: 동시간에 실행되는 다른 thread 경로.
- concurrency: 한번에 여러 작업을 실행시키는 것.
- queue: FIFO 방식으로 obj 관리하는 경량의 자료구조.
- sync vs async

16. Grand Central Dispatch (GCD)

두에서 thread를 관리하면서 동시에 작업을 실행시키는 API.

- DispatchQueue: FIFO
- SerialDispatchQ: 한번에 한 작업.
- Concurrent D.Q: 기다리지 않고 작업 실행.

Main Dis. Q: 메인 thread에서 작업을 실행할 수 있는 절연 사용 가능

17. guard의 장점

- 런타임 구조의 코드 대체 수 있음 (if let)
- break, return을 사용하여 초기에 할수 빠져나가기.

18. Method Swizzling

#selector(method) mapping 변경 → 런타임 다른 것으로olini

19. nonescaping / escaping closure.

↳ 할수파 클로저 전달 → 실행 → 할수 return.

↳ 정역 → 실행 → asunc → escape stored

10. weak, unowned?
- unowned는 nil이 될수 X, optional X
 - instance가 메모리 해제되었을 때 사용되거나 않을 때 고려해 사용
21. ARC?
- 강제로 타입의 자동 메모리 관리 가능
 - strong ref가 0이 될 때만 obj를 메모리에서 빼기.
22. CollectionView | TableView
- TableView: 여러 항목들의 list를 보여주고, 수직적 방법
수직 스크롤로 제작
 - CollectionView: 여러 항목을 보여주지만 여러 커스텀 + 조우.
23. REST, HTTP, JSON?
- HTTP는 app protocol이며 규칙의 집합
Server - Client 사이에 전송에 사용.
 - REST: 일관성 설계, 규칙, Web API 등에 유래한다.
 - JSON: 직관적이고 사람이 읽기 쉬운 시스템간 데이터 전송.
24. Delegation이 해결할 수 있는 문제.
- Obj 역할을 하는 것들 막는다.
 - 동작이나 모듈 수정.
25. Down Casting
- as : upcasting
 - as? : 실패시 nil
 - as! : 날짜
26. IB outlet | IBAction? outlet
- IBoutlet: interface builder's needed to associate properties in your app with components in IB.
 - IBAction: Interface Builder Action is used to allow your methods to be associated with actions in IB which is called when a specific user interaction occurs.
27. [weak self], [unowned self]
- Both do not create a strong hold on the referred object.
They don't increase the retain count in order to prevent ARC from dealloc. the referred object.
 - weak self: possibility of nil → optional
 - unowned self: will not become nil → unowned reference
MUST be set during initialization: defined non optional
 - use weak: closure to handle a response from a network request (long lived)
 - V.C could close before the request completes so self no longer points to a valid object when the closure is called.
 - use unowned: closure that handles an event on a button
 - as soon as V.C goes away, the button and any other items it may be referencing from self goes away simultaneously, closure block also goes away too.
 - self can be nil → [weak self]
self cannot be nil → [unowned self]
- static vs. final
- global variable
can be ALWAYS accessed
- static: used to define the class member that can be used independently of any object of that class.
 - Final: declare a constant variable, a method which cannot be overridden and a class that cannot be inherited.
28. Extension (Similar to categories in obj-c)
- adds new functionality to an existing class, enum, protocol type.
 - ability to extend types for which you do not have access
 - ① add computed instance properties + type properties
 - ② define instance methods + type methods
 - ③ provide new initializers
 - ④ define new Subscripts
 - ⑤ define and use new nested types
 - ⑥ make an existing type conform to a protocol
29. if-let: unwrap an optionals in safe way, otherwise nil
30. guard: early exit and doesn't allow any code if the condition is false, guard let/var outside the block of guard, which you can't do it let/var
31. optional chaining: retrieving value from the chain of optional value. If everything after the question will only be run if everything before the question mark have the value
- ```
if let roomCount = guest.residence?.numberOfRooms {
 print("\(roomCount)")
} else {
 print("unable to return")
}
```
32. optional binding: optional values and how we unwrap values whether an optional contains a value → make value available
- ```
if let constant = someoptional { }
```

1. App life cycle

3가지의 실행모드, 5가지의 상태(state)로 구분이 가능

- ① not running

- 실행되지 않는 코드와 상태

② Foreground

- active
 - inactive

③ Background

- running
 - suspended

not running → active : app 실행 대기

active \rightarrow inactive \rightarrow running : APP 활성화 상태에서 비활성화 상태로 만든 뒤
백그라운드에서도 계속 실행중인 상태.

active → inactive → suspend : app 활성화 상태에서 비활성화 상태로 만든다
백그라운드에서도 정지되어 있는 상태.

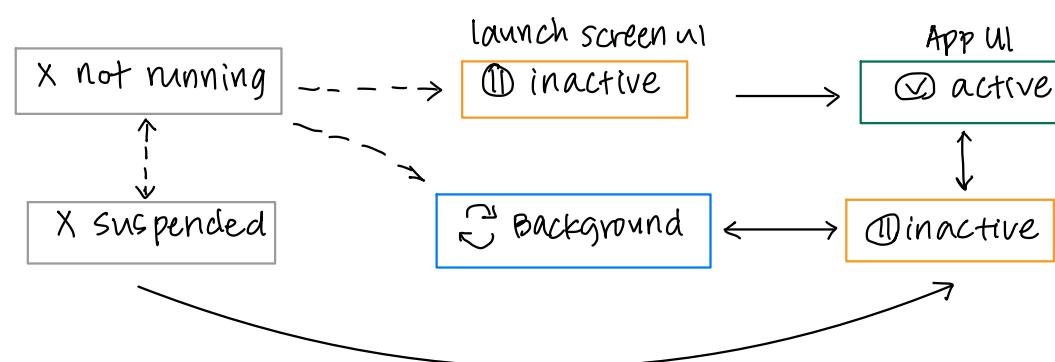
running → active : 백그라운드에서 실행 중인 app이 foreground에서 활성화.

App and Environment: manage life-cycle events and your app's UI scenes, and get info about traits and the environment in which your app runs

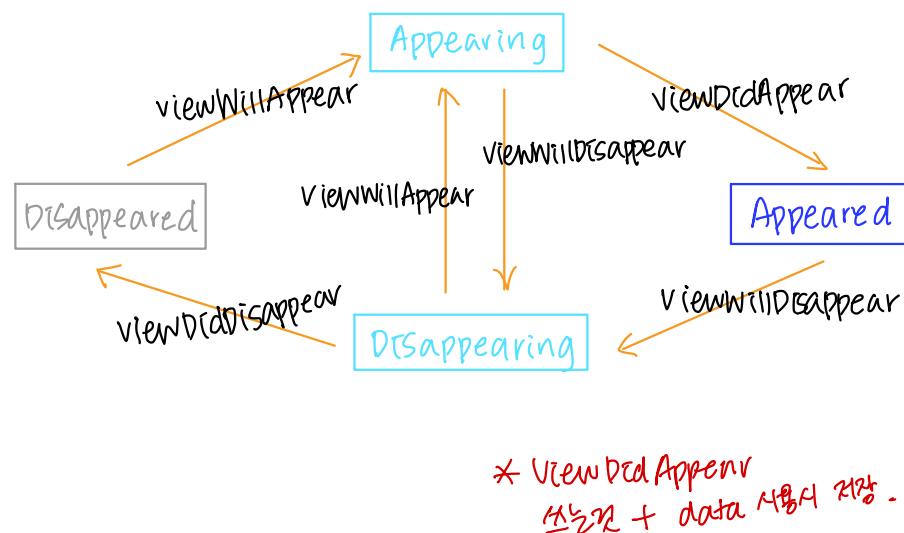
UIkit notifies by calling methods

- iOS 13 and later → UISceneDelegate: respond to life cycle events in a scene based app.

- iOS 12 and earlier → UIApplicationDelegate: respond to lifecycle events



2. View life cycle : app은 하나 이상의 view로 구성되어 있으며, 각각의 view들은 life cycle을 가지고 있음.

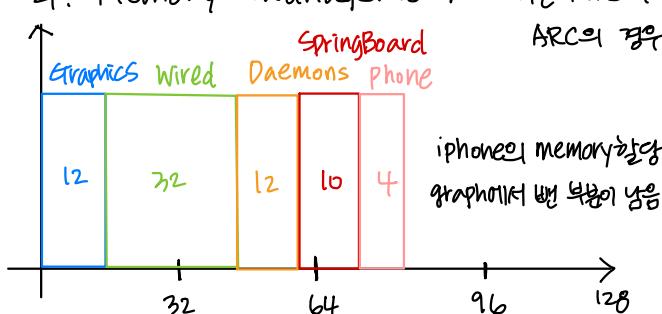


- **viewDidLoad**: viewController class가 생성될 때 가장 먼저 실행.
특별한 경우가 아니라면 딱 한번 실행되기 때문에 초기화할 때 사용.
 - **viewWillAppear**: view가 생성되기 직전에 항상 실행되기 때문에 view가 나타나기 전에 설정해야 하는 작업들을 여기서
 - **viewDidAppear**: View가 생성된 후에 실행. 데이터를 받아서 화면에 뿌려주거나, 초기화이던 작업의 로직 위치 (**viewWillAppear**의 로직을 넣었다가 안되는 경우)
 - **viewWillDisappear**: view 사라지기 직전 실행.
 - **viewDidDisappear**: ————— 고 날 뒤 실행.

3. Delegate vs. Block vs. Notification.

- Delegate는 객체간의 데이터통신을 할 경우 전달자 역할을 합니다. - event 처리: 특정 객체에서 발생한 event를 다른 객체에게通报. Delegate에게 알릴 수 있는 것은 여러 event가 있으나 class가 delegate로 부터 데이터를 가져와야 할 때 사용. ex) UITableView.
 - Block은 event가 처리될 때 사용하기 좋습니다. Completion block 사용 예 ex) NSURLConnection sendAsynchronousRequest:queue : completionHandler
 - Notification은 event를 통해 여러 Listener들을 때 사용. ex) 내가 특정 event를 기반으로 정보를 표시하는 방법을 Not.으로 broadcasting 하여 연결해나면서 이를 때 목록의 객체가 전달되는 것 확인.

4. Memory management : 기존 MRC의 대체 retain/release/autorelease (objc-)



* DDD FAQ : What is strong reference?
| why need weak / unowned?
| weak self in closure

closure + delegate → reference counting?
→ escaping / nonescaping.

5. Frame vs. Bound

- Frame: Parent view의 상대적 위치 (x,y) 및 크기 (너비, 높이)로 표현되는 시각형
- Bound: 자체 좌표계 (0,0)를 기준으로 위치 (x,y) 및 크기 (너비, 높이)로 표현되는 시각형.

FAQs

① 블록 객체는 어디에 생성되는가?
Heap : reference type (class)
Stack : value type (struct)

② 데이터 저장 방법
↓
server / cloud
property list
archive
- SQLite
- File
- CoreData

Struct VS Class

Struct

- call by value (할당 | parameter 전달시 value copy)
- stack memory 영역에 할당 (속도가 빠름)
 - Scope based lifetime: 컴파일 타임에 compiler가 실제 메모리를 할당/해제하지 정확히 알고 있음.
 - data locality: CPU Cache 를 더 높음
- 상속 불가능 (protocol은 사용 가능)
- NSData로 serialize 불가능
- Codable 프로토콜을 이용하여 순수한 JSON \leftrightarrow struct 변환 가능.
- 할당 시에는 copy가 일어나기 때문에 multi-thread 환경에서 공유 변수로 인해 문제를 일으킬 확률이 적음.

Class

- call by reference (할당 | parameter 전달시 메모리 주소 복사)
- heap memory 영역에 할당 (속도 느림)
 - runtime에 직접 allocation과 reference counting을 통해 deallocate 처리함.
 - memory fragmentation 등의 overhead 증가.
- 상속 가능
- NSData serialize 가능
- Codable 사용 불가능
- runtime의 type casting을 통해서 class instance에 따라 여러 동작이 가능

RULE

STRUCT : 상속이 필요 X, 모델의 사이즈 ↓, JSON 필드와 1:1 mapping

CLASS : serialize할 때 정답이 아닐 경우 파일로 저장, Obj-C API 사용.

GCD (Grand Central Dispatch)

`DispatchQueue`: an object that manages the execution of tasks serially / concurrently on your app's main thread or on a background thread.

Declaration

```
class DispatchQueue : DispatchObject
```

Overview

Dispatch queues are **FIFO** queues to which your application can submit tasks in the form of block objects. Dispatch queues execute tasks either **serially** or **concurrently**. Work submitted to dispatch queues executes on a pool of threads managed by the system. Except for the dispatch queue representing your app's **main thread**, the system makes no guarantees about which thread it uses to execute a task.

You schedule work items synchronously or asynchronously. When you schedule a work item **synchronously**, your code **waits until that item finishes execution**. When you schedule a work item **asynchronously**, your code **continues executing while the work item runs elsewhere**.

Important

Attempting to **synchronously** execute a work item on the **main queue** results in deadlock.

Avoiding Excessive Thread Creation

When designing tasks for concurrent execution, do not call methods that block the current thread of execution. When a task scheduled by a concurrent dispatch queue blocks a thread, the system creates additional threads to run other queued concurrent tasks. If too many tasks block, the system may run out of threads for your app.

Another way that apps consume too many threads is by creating too many **private concurrent dispatch queues**. Because each dispatch queue consumes thread resources, creating additional concurrent dispatch queues exacerbates the thread consumption problem. Instead of creating private concurrent queues, submit tasks to one of the **global concurrent dispatch queues**. For serial tasks, set the target of your serial queue to one of the global concurrent queues. That way, you can maintain the serialized behavior of the queue while minimizing the number of separate queues creating threads.

Concurrency: as soon as you add heavy tasks to your app like data loading it slows your UI work down or even freezes it. Concurrency lets you perform 2 or more tasks "simultaneously"

- thread safety - not always as easy to control.
ex) trying to change the same variable on a different threads or accessing the resources already blocked by the different threads.

- queues
- synchronous / asynchronous task performance
- priorities
- common troubles

Queues: Serial / Concurrent / Global / Private

Serial queues: tasks will be finished one by one

Concurrent queues: tasks will be performed simultaneously and will be finished on unexpected schedule

Private queues (Serial / Concurrent)

Global queues

Main queue is only Serial queue out of all of the global q.

- don't perform heavy task which are not referred UI work on the main (loading data from network) to keep UI unfrozen and responsive to user actions.
- Global queues are system queues, there are some other tasks can run by the system on them

(직렬)

- Serial: Queue에 추가된 작업들이 각 순서에 맞게 시작된다
순서에 해당되는 작업이 끝나야만 그 다음 작업 시작.
- 직렬·동기: Queue 작업들의 시작 및 종료 순서가 보장되고 동일 Queue의 작업이 끝나면 Queue 밖의 작업 실행 (일반화)
- 직렬·비동기: Queue 작업의 순서 보장, Queue 밖의 작업은 순서 X
 $0 < 10$ $0 < 1000$
`DispatchQueue.sync → temp = 1` `or 1001`
`DispatchQueue.async → temp = 1`

(동기)

- Concurrent: Queue에 추가된 작업들이 각 순서에 맞지 않아 작업이 시작되는 순서의 작업이 실행 중간에라도 새 작업이 시작되고 동시에 여러 작업이 실행중인 상태.
- `DispatchQueue.global().async → 1`
`DispatchQueue.global().sync → 991`
다른 영상이 끝나면서 다른 영상이 시작되어 누적되는 듯한 X.

`DispatchQueue.main(Queue)`: Serial queue

Sync and async do not determine serialization or currency of a queue, but instead refer to how the task is handled.

동기/비동기는 영상 / 동시간을 결정하는 것이 아니라 task가 어떤 queue로 향할지를 결정하는 것임.

deadlock

Synchronous func returns the control on the current queue only after task is finished. It blocks the queue and waits until the task is finished.

동기방법은 task가 끝날 때까지 기다려다가 실행함.

block main thread until the task has finished.

- When you need to wait for sth done on a DIFF queue and only then continue working on current queue
ex) Serial queue - Mutex: making sure only one thread is able to perform the protected piece of code s.t.

Asynchronous func returns control on the current queue right after task has been sent to be performed on the different queue. Doesn't block the queue

will happen on a background thread and update the main thread when it is thread.

Quality of Service / Priority

Queues also have different qos (Quality of Service) which sets the task performing priority

- UserInteractive - main queue
- UserInitiative - for the user initiated tasks on which user waits for some response
- Utility - for the tasks some time and doesn't require immediate response ex) working with data.
- background - for the tasks which aren't related with the visual part and which aren't strict for completion time
- default - doesn't transfer qos info. b/w .userinitiated and .utility

Common Troubles

- Race condition: caused when the app work depends on the order of the code parts execution.
- Priority Inversion: when the higher priority tasks wait for the smaller priority tasks to be finished due to some resources being blocked.
- Deadlock: when a few queues have infinite wait for the sources (variables, data) already blocked by some queues
- NEVER call sync method on the main queue
block queue - will never be finished - deadlock
- Sync: need to wait until the task is finished
ex) making sure func method is not double called.

ARC (Automatic Reference Counting)

#1

Swift uses Automatic Reference Counting (ARC) to track and manage your app's memory usage. In most cases, this means that memory management "just works" in Swift, and you don't need to think about memory management yourself. ARC automatically frees up the memory used by class instances when those instances are no longer needed.

Reference counting applies only to instances of classes. Structures and enumerations are value types, not reference types, and aren't stored and passed by reference.

How ARC Works

Every time you create a new instance of a class, ARC allocates a chunk of memory to store information about that instance. This memory holds information about the type of the instance, together with the values of any stored properties associated with that instance.

Additionally, when an instance is no longer needed, ARC frees up the memory used by that instance so that the memory can be used for other purposes instead. This ensures that class instances don't take up space in memory when they're no longer needed.

However, if ARC were to deallocate an instance that was still in use, it would no longer be possible to access that instance's properties, or call that instance's methods. Indeed, if you tried to access the instance, your app would most likely crash.

To make sure that instances don't disappear while they're still needed, ARC tracks how many properties, constants, and variables are currently referring to each class instance. ARC will not deallocate an instance as long as at least one active reference to that instance still exists.

To make this possible, whenever you assign a class instance to a property, constant, or variable, that property, constant, or variable makes a strong reference to the instance. The reference is called a "strong" reference because it keeps a firm hold on that instance, and doesn't allow it to be deallocated for as long as that strong reference remains.

```
1 class Person {  
2     let name: String  
3     init(name: String) {  
4         self.name = name  
5         print("\(name) is being initialized")  
6     }  
7     deinit {  
8         print("\(name) is being deinitialized")  
9     }  
10}
```

```
1 var reference1: Person?  
2 var reference2: Person?  
3 var reference3: Person?
```

optional
→ automatically initialized a person with a value nil
So don't reference a person with a value nil

You can now create a new Person instance and assign it to one of these three variables:

```
1 reference1 = Person(name: "John Appleseed")  
2 // Prints "John Appleseed is being initialized"
```

Note that the message "John Appleseed is being initialized" is printed at the point that you call the Person class's initializer. This confirms that initialization has taken place.

Because the new Person instance has been assigned to the reference1 variable, there's now a strong reference from reference1 to the new Person instance. Because there's at least one strong reference, ARC makes sure that this Person is kept in memory and isn't deallocated.

If you assign the same Person instance to two more variables, two more strong references to that instance are established:

```
1 reference2 = reference1  
2 reference3 = reference1
```

There are now three strong references to this single Person instance.

If you break two of these strong references (including the original reference) by assigning nil to two of the variables, a single strong reference remains, and the Person instance isn't deallocated:

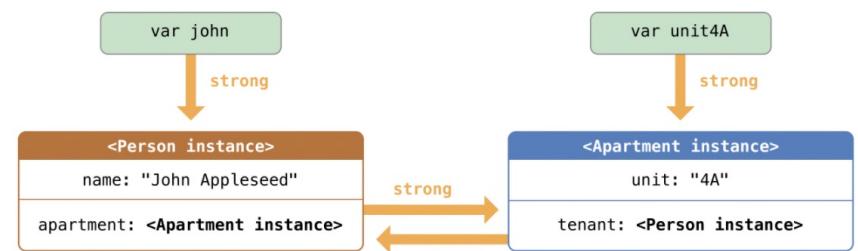
```
1 reference1 = nil  
2 reference2 = nil
```

ARC doesn't deallocate the Person instance until the third and final strong reference is broken, at which point it's clear that you are no longer using the Person instance:

```
1 reference3 = nil  
2 // Prints "John Appleseed is being deinitialized"
```

```
1 john!.apartment = unit4A  
2 unit4A!.tenant = john
```

Here's how the strong references look after you link the two instances together:

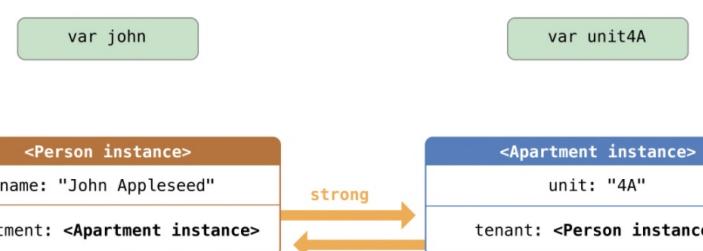


Unfortunately, linking these two instances creates a strong reference cycle between them. The Person instance now has a strong reference to the Apartment instance, and the Apartment instance has a strong reference to the Person instance. Therefore, when you break the strong references held by the john and unit4A variables, the reference counts don't drop to zero, and the instances aren't deallocated by ARC:

```
1 john = nil  
2 unit4A = nil
```

Note that neither deinitializer was called when you set these two variables to nil. The strong reference cycle prevents the Person and Apartment instances from ever being deallocated, causing a memory leak in your app.

Here's how the strong references look after you set the john and unit4A variables to nil:



The strong references between the Person instance and the Apartment instance remain and can't be broken.

You can now create a specific Person instance and Apartment instance and assign these new instances to the john and unit4A variables:

```
1 john = Person(name: "John Appleseed")  
2 unit4A = Apartment(unit: "4A")
```

Here's how the strong references look after creating and assigning these two instances. The john variable now has a strong reference to the new Person instance, and the unit4A variable has a strong reference to the new Apartment instance:

Resolving Strong Reference Cycles Between Class Instances

Swift provides two ways to resolve strong reference cycles when you work with properties of class type: weak references and unowned references.

Weak and unowned references enable one instance in a reference cycle to refer to the other instance without keeping a strong hold on it. The instances can then refer to each other without creating a strong reference cycle.

Use a weak reference when the other instance has a shorter lifetime—that is, when the other instance can be deallocated first. In the Apartment example above, it's appropriate for an apartment to be able to have no tenant at some point in its lifetime, and so a weak reference is an appropriate way to break the reference cycle in this case. In contrast, use an unowned reference when the other instance has the same lifetime or a longer lifetime.

Weak References

A weak reference is a reference that doesn't keep a strong hold on the instance it refers to, and so doesn't stop ARC from disposing of the referenced instance. This behavior prevents the reference from becoming part of a strong reference cycle.

Because a weak reference doesn't keep a strong hold on the instance it refers to, it's possible for that instance to be deallocated while the weak reference is still referring to it. Therefore, ARC automatically sets a weak reference to nil when the instance that it refers to is deallocated. And, because weak references need to allow their value to be changed to nil at runtime, they're always declared as variables, rather than constants, of an optional type.

You can check for the existence of a value in the weak reference, just like any other optional value, and you will never end up with a reference to an invalid instance that no longer exists.

```

1 class Person {
2     let name: String
3     init(name: String) { self.name = name }
4     var apartment: Apartment?
5     deinit { print("\(name) is being deinitialized") }
6 }
7
8 class Apartment {
9     let unit: String
10    init(unit: String) { self.unit = unit }
11    weak var tenant: Person?
12    deinit { print("Apartment \(unit) is being deinitialized") }
13 }
```

The strong references from the two variables (john and unit4A) and the links between the two instances are created as before:

```

1 var john: Person?
2 var unit4A: Apartment?
3
4 john = Person(name: "John Appleseed")
5 unit4A = Apartment(unit: "4A")
6
7 john!.apartment = unit4A
8 unit4A!.tenant = john
```

```

1 john = nil
2 // Prints "John Appleseed is being deinitialized"
```

```

1 unit4A = nil
2 // Prints "Apartment 4A is being deinitialized"
```

Unowned References

Like a weak reference, an unowned reference doesn't keep a strong hold on the instance it refers to. Unlike a weak reference, however, an unowned reference is used when the other instance has the same lifetime or a longer lifetime. You indicate an unowned reference by placing the unowned keyword before a property or variable declaration.

Unlike a weak reference, an unowned reference is expected to always have a value. As a result, marking a value as unowned doesn't make it optional, and ARC never sets an unowned reference's value to nil.

If you try to access the value of an unowned reference after that instance has been deallocated, you'll get a runtime error.

In this data model, a customer may or may not have a credit card, but a credit card will always be associated with a customer. A CreditCard instance never outlives the Customer that it refers to. To represent this, the Customer class has an optional card property, but the CreditCard class has an unowned (and non-optional) customer property.

Furthermore, a new CreditCard instance can only be created by passing a number value and a customer instance to a custom CreditCard initializer. This ensures that a CreditCard instance always has a customer instance associated with it when the CreditCard instance is created.

Because a credit card will always have a customer, you define its customer property as an unowned reference, to avoid a strong reference cycle:

```

1 class Customer {
2     let name: String
3     var card: CreditCard?
4     init(name: String) {
5         self.name = name
6     }
7     deinit { print("\(name) is being deinitialized") }
8 }
9
10 class CreditCard {
11     let number: UInt64
12     unowned let customer: Customer
13     init(number: UInt64, customer: Customer) {
14         self.number = number
15         self.customer = customer
16     }
17     deinit { print("Card #\(number) is being deinitialized") }
18 }
```

not optional
nor var

```

var john: Customer? 1 john = Customer(name: "John Appleseed")
2 john!.card = CreditCard(number: 1234_5678_9012_3456, customer: john!)
```

Because there are no more strong references to the Customer instance, it's deallocated. After this happens, there are no more strong references to the CreditCard instance, and it too is deallocated:

```

1 john = nil
2 // Prints "John Appleseed is being deinitialized"
3 // Prints "Card #1234567890123456 is being deinitialized"
```

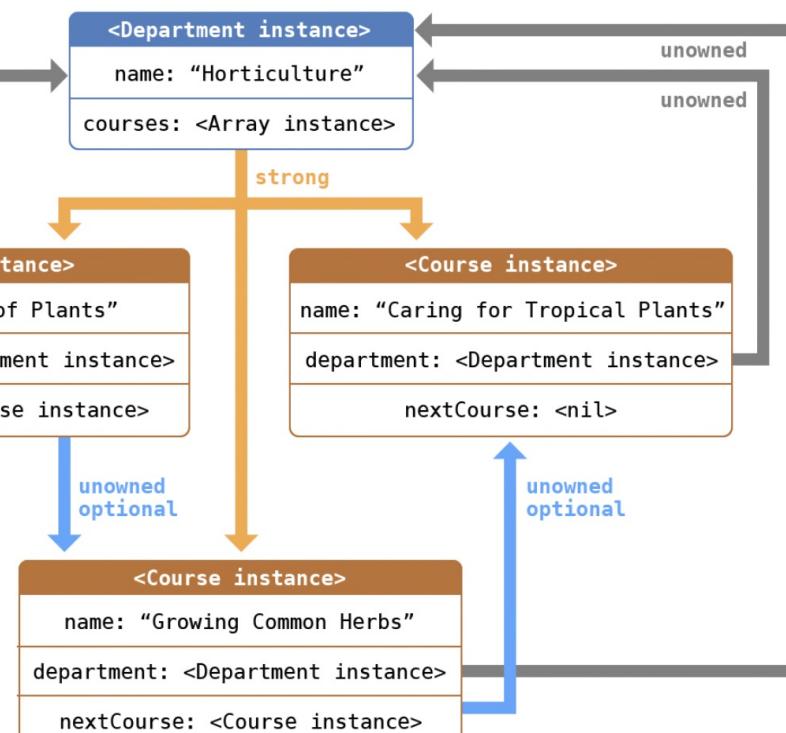
Unowned Optional References

You can mark an optional reference to a class as unowned. In terms of the ARC ownership model, an unowned optional reference and a weak reference can both be used in the same contexts. The difference is that when you use an unowned optional reference, you're responsible for making sure it always refers to a valid object or is set to nil.

```

1  class Department {
2      var name: String
3      var courses: [Course]
4      init(name: String) {
5          self.name = name
6          self.courses = []
7      }
8  }
9
10 class Course {
11     var name: String
12     unowned var department: Department
13     unowned var nextCourse: Course?
14     init(name: String, in department: Department) {
15         self.name = name
16         self.department = department
17         self.nextCourse = nil
18     }
19 }

let department = Department(name: "Horticulture")
2
3 let intro = Course(name: "Survey of Plants", in: department)
4 let intermediate = Course(name: "Growing Common Herbs", in: department)
5 let advanced = Course(name: "Caring for Tropical Plants", in: department)
6
7 intro.nextCourse = intermediate
8 intermediate.nextCourse = advanced
9 department.courses = [intro, intermediate, advanced]
```



Autolayout

1 Understanding Auto Layout

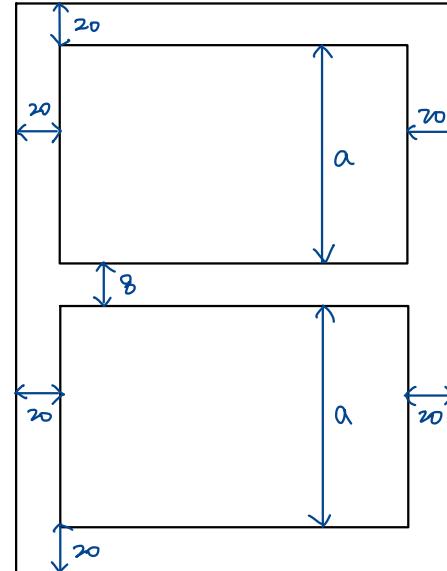
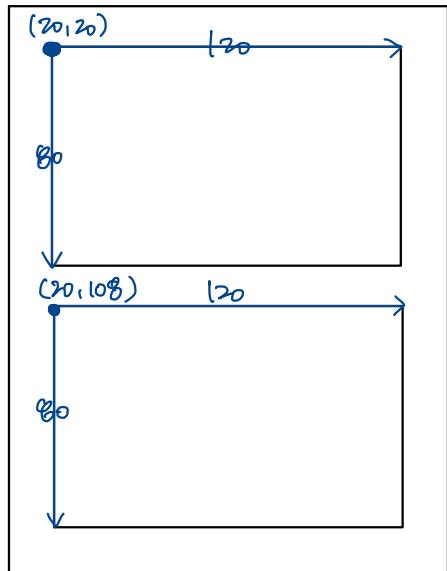
1. External Changes

- user resizes the window (os x)
- user enters/ leaves split view on an iPad (ios)
- Device rotates (ios)
- The active call and audio recording bars appear/disappear
- You want to support different size classes
Screen sizes

2. Internal Changes : when the size of the views / controls in your user interface change

- The content displayed by the app changes depends on language / region.
(201, 202, 203, 204)
- The app supports internationalization.
- The app supports Dynamic Type (ios)
ex) font sizes

3. Auto Layout vs. Frame-based layout



Programmatic Layout

- need to calculate every view hierarchy
→ changes occur → recalculate

+ (most flexibility + power)

- inefficiency

Autoresizing mask

- defines how a view's frame changes when its superview's frame changes
- + Simplifies the creation of layout (ext)
- supports small subset of layout (intx)

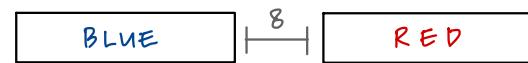
Auto Layout : Constraints
relationship between two views

- dynamically respond to both int/ext changes

2 Auto Layout without Constraints - Stackviews

- axis (NSTableView only) : stackview's orientation
- orientation (NSTableView only) : stackview's orientation
- distribution : the layout of the views along the axis
- alignment : the layout of the views perpendicular to the stackview's axis
- spacing : spacing between adjacent views.

3 Anatomy of a constraints



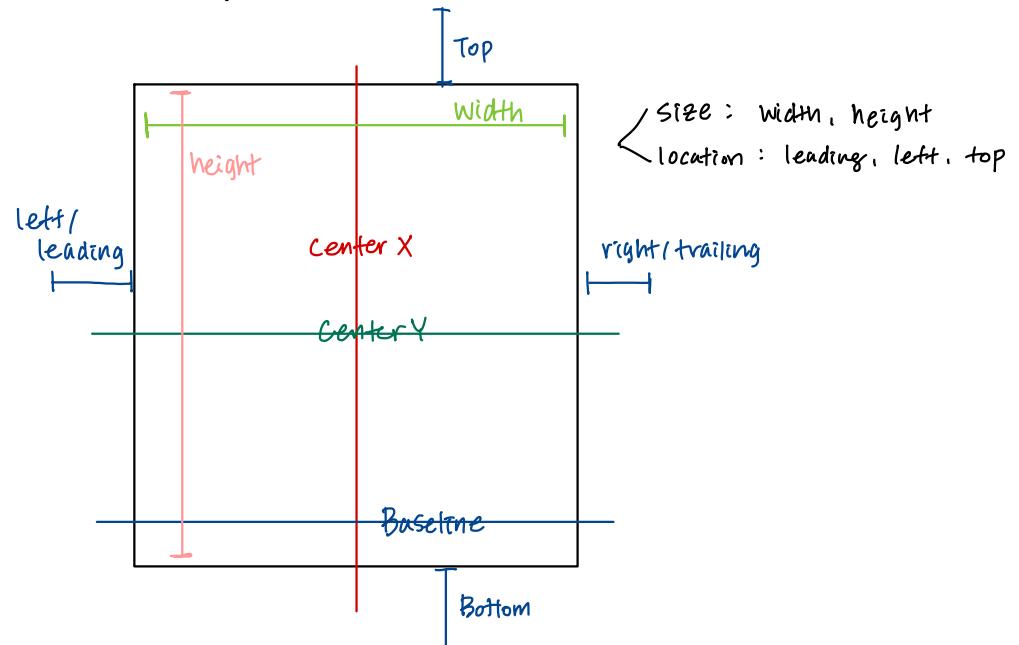
$$\text{RedView}.Leading = 1.0 \times \text{BlueView}.trailing + 8.0$$

Item1 attribute1 Item2 attribute2 Constants

↓ ↓ ↓ ↓

multiplier relationship

1. Auto layout Attributes



Rules

- Cannot contain a size attribute to a location attribute
- Cannot assign constant values to location attributes
- cannot use a nonidentity multiplier (a value other than 1.0) with location attributes
- For location attributes, you cannot constrain vertical attributes to horizontal attr.
- For location attr. You cannot constrain leading or trailing attr to left or right attr.

Height **size** - can be assigned constant values or combined with other height and width attr. (nonnegative)

Top **value increases** attr can be combined only with
Bottom **as you move down** - **Center Y, Top, Bottom, Baseline** attr.
Baseline **the screen**

Leading **value increases as you move towards the trailing edge.** can be combined only
Trailing **the values increase as you move to the right.** w/ **leading, trailing, center X** attr.

left **For right-left layout direction,**
right **the values increase as you move to left**

left **the values increase as you move to the right** can be combined only w/ l, r, CenterX
right **l, r** ~~are~~ **are** **not** **leading/trailing attr.**
due to **adaption to view's reading direction** (language setting)

Center X **Based on the other**
Center Y **attr in the equation**

- **Center X** can be combined w/
Center X, leading, trailing, r, l
Center Y can be combined w/
CenterY, top, bottom, Baseline.

Closure

* retain cycle em myself

+ how to resolve

→ combine 에이드?

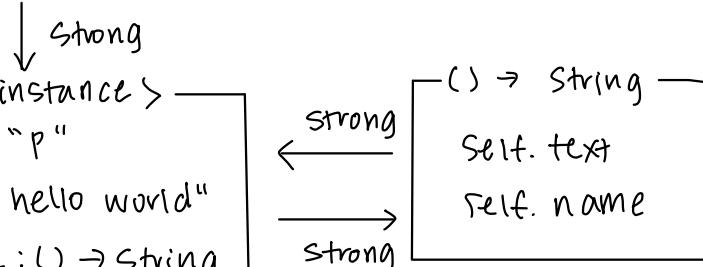
Syntax:

```
{(parameters) → return type in  
statements}
```

```
let birthday: (String) → () = { name in  
    print("Happy Birthday, \((name)!)")  
    birthday("bob")
```

Closure type: (String) → ()
expression: { name in ... }

var Paragraph



단일 paragraph variable이 nil이 된다고 하더라도
HTMLElement, closure는 꽤나 deallocate 되지 X

① escaping vs. ② nonescaping

① ② nonescaping: when passing a closure as the function arg. the closure gets execute with the func's body and returns the compiler back. As the execution ends, the passed closure goes out of scope and have no more existence in memory.

Lifecycle: 1) pass closure as func arg. during func call
2) do some additional work with func
3) func runs the closure.
4) func returns the compiler back.

② ③ escaping: when passing a closure as the func arg, the closure is being preserved to be execute later and func's body gets executed, returns the compiler back. As the execution ends, the scope of the passed closure exist and existence in memory, till the closure gets executed.

- storage: when you need to preserve the closure in storage that exist in the memory, past of the calling func. get executed and return the compiler back. (like waiting for the API response)
- asynchronous execution: when you are executing the closure asynchronously on dispatch queue, the queue will hold the closure in memory for you, to be used in future. In this case, you have no idea when the closure will get executed.

Lifecycle: 1) Pass the closure as func arg. during func call.
2) Do some additional work in func
3) Func execute the closure asynchronously or stored.
4) Func returns the compiler back.

Strong reference cycles for closures

• occurs if you assign a closure to a property of a class instance, and the body of that closure captures the instance. This capture might occur b/c the closure's body accesses a property of the instance, such as self.someProperty or b/c the closure calls a method on the instance, such as self.someMethod(). → these accesses cause the closure to "capture" self, creating a strong reference cycle

• When assigning a closure to a property → assigning a reference to that closure

Resolving strong reference cycle for closure

• capture list: defines rules to use when capturing one or more ref. cycles b/w two class instances. You declare each ref. to be a weak or unowned depends on the relationships b/w the different parts of your code.

ex) lazy var someClosure = {

```
[unowned self, weak delegate = self.delegate]  
(index: Int, stringToProcess: String) → String in  
    // body
```

Weak and Unowned Reference

- unowned self: when the closure and the instance it captures will always refer to each other, and will always deallocated at the same time.
- weak self: when the captured ref. may become nil at some point in the future. Weak ref. are always of an optional type, and automatically become nil when the instance they reference is deallocated. This enables you to check for their existence within the closure's body.

Miscell: why nonescaping default?

- performance and code optimisation by the compiler

Storage example - Storing the closure for future use

var completionHandler: ((Int) → Void)?

func getSumOf(array: [Int],

handler: ② escaping ((Int) → Void)) {

var sum: Int = 0

for val in array { sum += val }

self.completionHandler = handler

}

func doSomething() {

```
    self.getSumOf(array: [...]) { [weak self] (sum) in  
        print(sum)
```

}

Asynchronous Execution example

func getSumOf(array: [Int],

handler: ② escaping ((Int) → Void)) {

var sum: Int = 0

for val in array { sum += value }

Globals.delay(0.3, closure: { handler(sum) })

func doSomething() {

```
    self.getSumOf(array: [...]) { [weak self] (sum) in  
        print(sum)
```

}

① Escaping

: a closure is to escape a function when the closure is passed as an argument to the function, but is called after the function returns. When you declare a function that takes a closure as one of its parameters, you can write `@escaping` before the parameter's type to indicate that the closure is allowed to escape.

(함수 블록 내에서 뒤에 두어도 블리적 수 있어야 가능 (비동기 처리 유형))

아래와 같이 함수밖에 있는 브록에 (completionHandlers) 함수안의 인자값인 completionHandler를 넣어줍니다.

```
1 var completionHandlers = [() -> Void]()
2 func someFunctionWithEscapingClosure(completionHandler: @escaping () ->
3   Void) {
4   completionHandlers.append(completionHandler)
5 }
```

그 다음으로 escaping이 없는 함수도 만들어줍니다.

```
func someFunctionWithNonescapingClosure(closure: () -> Void) {
  closure()
}
```

어떤 클래스에서 값이 10인 x 변수를 만들어주고 escaping하는 함수엔 x값을 100으로 escaping이 없는 함수엔 x값을 200으로

만들어주는 doSomething 함수를 만들어줍니다.

```
class SomeClass {
  var x = 10
  func doSomething() {
    someFunctionWithEscapingClosure { self.x = 100 }
    someFunctionWithNonescapingClosure { x = 200 }
  }
}
```

이 클래스를 불러와서 doSomething을 실행한 뒤 x를 출력해보면 200이 나옵니다.

또한 아래 completionHandlers에 저장해두었던 completion을 불러와서 실행한 뒤 x를 출력해보면 100이 나옵니다.

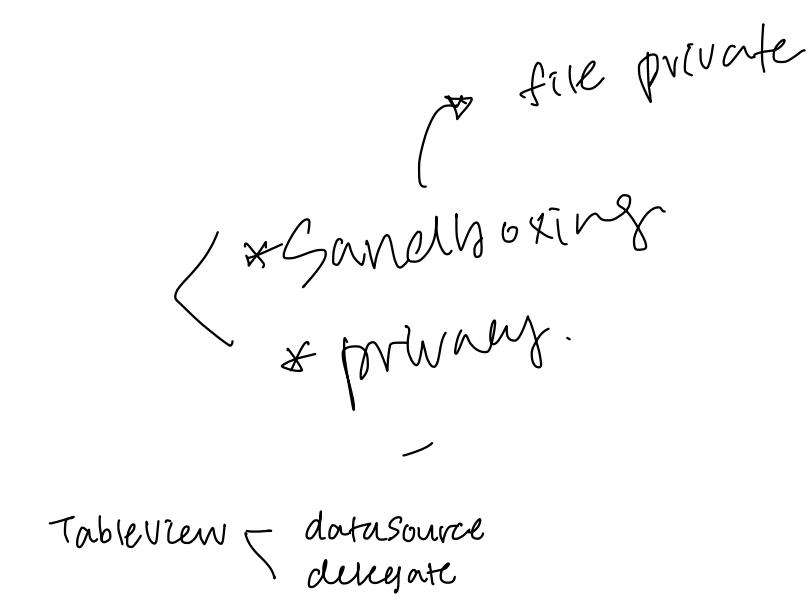
```
let instance = SomeClass()
instance.doSomething()
print(instance.x)
// Prints "200"

completionHandlers.first?()
print(instance.x)
// Prints "100"
```

이것으로 `@escaping`을 통해 completion을 탈출시켜 외부에 저장해두고 원할 때 불러와 값을 변경할 수 있습니다.

만약 noescape함수에 블로거를 외부에 저장하려고 하면 아래와 같이 에러가 납니다.

```
func someFunctionWithNonescapingClosure(closure: () -> Void) {
  closure()
}
completionHandlers.append(closure)
```



pass the data: prepare - right before any segue happens from that V.C

```
prepare(for segue: UIStoryboardSegue, sender: AnyObject)
let indexPath = tableView.indexPathForSelectedRow
let index = indexPath?.row
let destinationViewController = segue.destinationViewController
as! DetailViewController
detailViewController.index = index
```

CoreData vs Realm

#1 Database Setup and Initialization

CoreData - need to create a model file containing all the entities and their attributes required during the app dev. process

- from iOS 10 `NSPersistentContainer` obj appeared,

```
persistentContainer = NSPersistentContainer(name: "model")
```

Realm - need to add framework to our project (cocoapods)

```
let realm = try! Realm()
```

#2 Creating model

CoreData (3 options to create the class)

1. Manual / None

2. Class Definition: automatic creation, XCode가 관리

3. Category / Extension: manual and automatic class creation

Realm

add new class that inherits from `Object`, adding properties to our model specifying the name of the property / datatype

ex) `let items = List<Item>()`

```
var parentCategory = LinkingObject(fromType: Category.self,
                                    properties: "items")
```

#3 Creating and saving objects

CoreData: `PersistentContainer`

```
let appDelegate = UIApplication.shared.delegate as! AppDelegate
let context = appDelegate.persistentContainer.viewContext
```

```
let newCategory = Category(context: self.context)
```

```
self.categories.append(newCategory)
```

```
context.save()
```

Realm

```
self.save(category: newCategory)
```

```
    ↳ do {
```

```
        try realm.write {
```

```
            realm.add(category)
```

```
    } catch
```

```
    tableView.reloadData()
```

#4 Requests

CoreData:

```
let request: NSFetchedResultsController<Category> = Category.fetchRequest()
```

```
do {
```

```
    categories = try context.fetch(request)
```

OR

```
let catP = NSPredicate(format: "parentCategory.name MATCHES %@",
```

```
                        selectedCategories!.name!)
```

Realm:

```
todoItems = selectedCategories?.items.sorted(byKeyPath: "title", ascending: true)
```

Pros & Cons

#1 Database Usage

Realm이 더 사용, - 변경에 대해 자동으로 save가 되어 글로벌 솔루션 위험 ↓

coreData의 경우 API에 차이나는 점을 알아야 함

- 변경이 있을 때마다 재작업, save를 반복할 듯

#2 Creation of the scheme

coreData의 경우 XCode가 플래그를 만들 - visual representation

one-to-many, many-to-many 등 유리.

realm의 경우 Objc dynamic var

```
LinkingObject(fromType: Category.self,
              property: "items")
```

#3 The app volume

coreData의 경우 앱의 크기가 커지지 않지만

realm의 경우 13MB 정도 더 커지니 됨.

#4 Cross-Platform Support

realm은 iOS, OS X, Java, Swift, Objective-C 등 있어서
logically 통일적임 (일致性)

#5 Work Speed

realm은 자체의 engine을 사용해나 때문이 → simple and fast.

#6 Mobile database security

Class vs. Struct

Common

- Both structs and classes can define properties to store values, and they can define functions.
- Can define subscripts to provide access to values with subscript syntax.
- Can define initializers to set up their initial state `init()`
- Can be extended with extension
- Can conform to protocols
- Can work with generics to provide flexible and reusable types

Classes Capabilities

- classes can inherit from another class, like you inherit from `UIViewController` to create your own view controller subclass.
- classes can be deallocated, you can invoke a `deinit()` function before the class is destroyed.
- classes are reference types and structs are value type
 value type: each instance keeps a unique copy of the data. Change in one instance doesn't change the other.
 reference type: each instance shares the data. The reference itself is copied, but not the data it references it.

When to use structs

- Simple data types:** when they are well-defined, don't need to accommodate complex relationships b/w objects
- Thread Safety:** in multi-thread environment, ex) database connection that's opened in diff thread.
 \rightarrow deadlock
- mostly structs scenario:** when properties of a struct are mostly value type
- Inheritance X**

When to use classes

Access members of a collection

- a variable or constant, containing collection
- 2 square bracket
- a key or index inside the bracket.

default & o/x

default initializer

memberwise initializer - only available in **Struct**

Failable initializer

\hookrightarrow returns nil / value

optional & on fail.

`init? [value: ___] {`

guard condition else {
return nil
}

Designated Initializer

```
Struct Rectangle {
    var width = 0
    var height = 0
}

let square
=Rectangle(width:10,
height:10)
```

```
Class Circle {
    var radius: Double
    var circumference: Double {
        2 * .pi * radius
    }
}

init(radius: Double) {
    self.radius = radius
}
```

struct는 stack 영역에 메모리를 저장해서 주소가 빠르고 struct를 바꾸는 시도는 struct가 아니라

class는 heap 영역에 메모리를 저장해 주소가 느리고, 빠지를 한다. reference 블록 \rightarrow deep copy 상속 및 serialize는 class

SWIFT

Value type - each instance keeps a unique copy of its data.

Basic data type, struct, enum, array, tuples.

Reference Type - shares a single copy of the data,

1. ARC(Automatic Reference Counting)에 대해 설명하세요.

[DEF] 자동 리퍼런스 카운팅으로서 자동으로 메모리를 관리해주는 방식을 말한다. 참조 카운팅이 0이 될 때만 메모리에서 해제된다.

원리: class에 새로운 instance를 만들 때 ARC는 인스턴스의 정보를 저장하기 위해 메모리를 할당. ARC는 인스턴스가 더 이상 사용되지 않는다고 판단하면 메모리 해제. Reference property의 instance를 할당하면 ARC는 참조되는 property 개수를 카운팅하여 참조하는 모든 변수가 인스턴스를 해제하기 전에 ARC는 instance를 메모리에서 해제하지 X

시점: 컴파일 시점에 동작.

7/16

7/16

2. Weak, Strong, Unowned

• Strong: 객체를 소유하여 reference count가 증가하는 property. 값은 지정 시점에 retain이 되고 참조가 종료되는 시점에 release가 된다.
- RC로 메모리 해제 피하고 객체를 안전하게 사용하고자 할 때.

• Weak: 객체 소유하지 않고 주소값만을 가지고 있는 포인터 개념. 참조하지만 weak 메모리를 해제시킬 수 있는 권한은 다른 클래스에. 값 지정 시 리터럴이 nil 가능 X → release X. 언제 메모리가 해제될지 알 수 X. 해제된 경우 자동으로 reference가 nil로 초기화를 해준다. MUST BE OPTIONAL
- retain cycle에 의해 메모리가 누수되는 문제를 막기 위해 사용.
ios Framework에서 대표적으로 delegate patterns

• Unowned: nil이 될 수 X, MUST NOT OPTIONAL. 해제된 메모리 영역을 접근하지 않는다는 핵심한 경계에만 사용.
- non-optional type으로 선언. 객체가 ARC에 의해 메모리가 해제되더라도 해당 객체 값을 조작하는 것으로 인식하며, 해당 객체에 액세스 할 경우 런타임 오류.
life cycle 명시하고 개발자에 의해 제어 가능이 명확한 경우, weak optional type 대신 사용해 간결화.

3. 클로저 블록내에 [Weak self] in 의 역할과 이유는?

• [weak self]의 역할: ARC가 property 개수를 counting하지 않도록 하여 순환참조가 일어나지 않도록 만드는 역할. 해제될 때 property value = nil로 만들어 순환 참조 X.

4. Escaping Closure Closure 쓰는 데서 completion block 등

• method parameter로 전달받은 closure는 method의 life cycle 내에서 실행하여 끌어내지 않고, method scope의 외부에 전달할 때는 해당 closure를 escape. 해당 closure가 끌어온 이유에도 closure는 메모리 어딘가 저장되어야 하며, closure 안에 사용된 outer object (self)가 weaker 같은 reference type을 사용해야 할 수 있음을 주의.

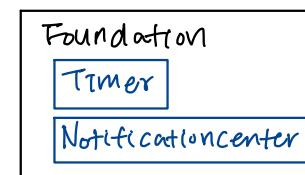
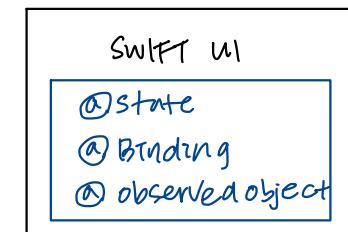
5. Casting

- as: type 변환의 성공 보장. compile time에 가능/불가능 여부 알 수 O
- as?: type 변환 실패시 nil return. —————→ ————— X
- as!: type 변환 실패하는 경우 runtime error —————→ ————— X

Foundation and UIKit / AppKit

Apple: several mechanisms you can use, on different system levels, to create and execute asynchronous code.

1. **NotificationCenter**: executes a piece of code any time an event of interest happens, such as when the user changes the orientation of the device or when the software keyboard shows or hide on the screen.
state on 따라가 code를 실행할 수 있도록 하는 장치
2. **The delegate pattern**: lets you define an object that acts on behalf of, or in coordination with, another object. For example, in your app delegate, you define what should happen when a new remote notification arrives, but you have no idea when this piece of code will be executed or how many times it will execute.
3. **Grand Central Dispatch and Operations**: Helps you abstract the execution of pieces of work. You can use them to schedule code to be executed sequentially in a serial queue or to run a multitude of tasks concurrently in different queues with different priorities.
Concurrency = synchronization.
4. **Closures**: create detached pieces of code that you can pass around in your code, so other objects can decide whether to execute it, how many times, and in what context



1. Where can SwiftUI be used?

- SwiftUI runs on iOS 13, macOS 10.15, tvOS 13, watchOS 6

2. Does SwiftUI replace UIKit?

- No. Many parts of SwiftUI directly build on top of existing UIKit component. (ex. UITableView)
- Some yes, new controls rendered by SwiftUI and not UIKit.

3. Does SwiftUI use Auto Layout?

- In the backend Yes (certainly being used - 하지만 개발자들은 알지 못함)
Frontend Box layout system (web)을 이용한 UI 보여짐.

4. Is SwiftUI fast?

- Yes. UIKit을 능가하는 속도
 - ① aggressively flatten their layer hierarchy → less drawing
 - ② many operations can bypass Core Animation entirely and go straight to Metal for extra speed.

원래 delegate, data source 같은 state management pattern → imperative framework UIKit + AppKit

SWIFT UI

State property

- to represent the internal state of a SwiftUI view, and automatically make a view update when the state was changed
∴ keep it private → ensuring only be mutated within the view's body

View에서 변수가 바뀐 것을 알고 → 바뀐 때 view에서 적용하기 바란다.

Two way Binding

- Provides a two-way connection between a given view and a state property defined outside of the view, and both state, and a binding-wrapped properties that can be passed as bindings by prefixing their property &

Concurrency: doing multiple tasks same time : Apple Hardware multi-core processor

Grand Central Dispatch FIFO

& NS Operation Queues: manages threads for us.

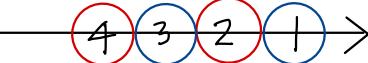
thread: think it as highway

Main thread

- Speedy
- UI

background thread

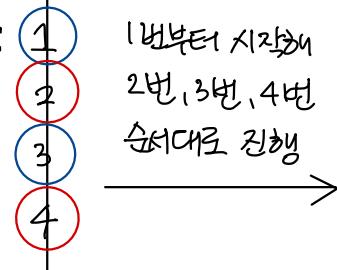
- loaded tasks (timely)

Serial: 

1번이 끝나면 2번 시작, 2번 끝나면 3번 시작. 3번 끝나면 4번 시작.

- predictable executable order
- prevents race conditions

MAIN QUEUE

Concurrent: 

1번부터 시작해
2번, 3번, 4번
순서대로 진행

- faster
- unpredictable order

"unpredictable"

BACKGROUND QUEUE

ex) Background에서 API를 가지고 와 UITableView를 populate 하는 경우 →

DispatchQueue.main.async {

self.tableView.reloadData()

Serial concurrency: destination → identifies the queue has a single thread or multiple threads

Synchronous / : source of the task → sync: will wait and block the current run loop until execution finishes before moving on to the next task.

asynchronous: will start, but return the execution to your app immediately.

Concurrency problem: race condition - multiple threads are trying to write to the same variable at the same time.

deadlock - both waiting on another task that can never complete

priority inversion - a queue with a lower quality of service is given higher system priority than a queue with higher quality of service.

tech blog of Chungkyu

10. tech

메소드 (method)

- instance method
- class method
- static (정적) method

```
def __str__(self): # 인스턴스 메소드
    return "사용자: {}, 이메일: {}".format(self.name, self.email)

④ classmethod # 클래스 메소드
def number_of_users(cls):
    print("총 유저 수는: {}".format(cls.count))

⑤ staticmethod # 정적 메소드
def is_valid_email(email_address):
    return "@" in email_address
```

instance 변수인 self.name, self.email 사용
class 변수인 cls.count 사용
→ 이전 변수 사용 X
어떤 속성을 다루지 않고, 단지 기능(행동)적인 역할만 하는 메소드를 정의하는 때.

절차적 프로그래밍과 객체 지향 프로그래밍의 차이

- ① 절차 - 필요한 데이터를 관련 있는 함수와 뭉어서 관리하기 힘들다. + ② 명령어들을 순서대로 실행하는 것.
- 객체 - 서로 관련 있는 데이터와 함수를 객체로 뭉어서 사용 가능. + 프로그램을 객체간의 소통으로 바라봄. 객체가 프로그래밍의 기본 단위가 되고 객체 속을 들여다보면 서로 관련된 속성과 행동이 있음

4가지 객체 지향 프로그래밍

- 추상화 (abstraction): extension of encapsulation, selecting data from a larger pool to show only the relevant details.
Adv: being able to apply the same info used to other applications with little or no modification.
- 간결화 (Encapsulation): can be accomplished when each object maintains a private state, inside a class
Other objects cannot access this state directly, they can only invoke a list of public functions.
- 상속 (Inheritance): ability of one object to acquire some/all properties of another object → reusability
- 다형성 (Polymorphism): Gives us a way to use a class exactly like its parent so there is no confusion with mixing types. Each child sub-class keeps its own functions/methods as they are

protocol

1. **Identifiable**: a class of types whose instances hold the value of an entity with stable identity. - use it when to provide a stable notion of identity to a class or value type.

"Identifiable" leaves the duration and scope of the identity unspecified.

- guaranteed always unique (e.g. UUID)
- persistently unique per environment
- unique for the lifetime of a process / object / current collection

2. **StateObject**: ① observed object or ② SwiftUI 2.0에서 추가됨.

① observed object의 경우 view의 reloading / refresh가 있을 경우에 data persistence X → reloading.

그러므로 ViewModel이 data를 holding하고 있기 때문에 StateObject 사용.

- 첫번째로 data instance를 initialize 할 경우 StateObject를 사용 (CREATION)

2nd view는 passing을 통해 subview에서 사용할 경우 ObservedObject 사용. (PASSING TO 2ND VIEW)

Foundation: access essential data types, collections, and operating-system services to define the base layer of functionality for your app.

- data storage, persistence, text processing, date and time calculations, sorting/filtering, networking
- classes, protocols, and data types defined by Foundation are used throughout iOS, macOS, watchOS, tvOS SDK

1. Fundamentals: ① numbers, Data, Basic Value / ② strings and Text / ③ collections / ④ dates and times
 ⑤ units and measurement / ⑥ Data Formatting / ⑦ filters and sorting

2. App support: ① Task management: manage app's work and how it interacts with system services like Handoff and shortcuts

② Resources: access assets and other data bundled with app

③ Notification: design patterns for broadcasting info and for subscribing to broadcasts

④ App Extension Support: manage the interaction between an app extension and its hosting app

⑤ Errors and Exceptions: respond to problem situations in your interactions with APIs, and fine-tune your app for better debugging

⑥ Scripting Support: Allow users to control your app with AppleScript and other automation technologies, or run scripts from within your app.

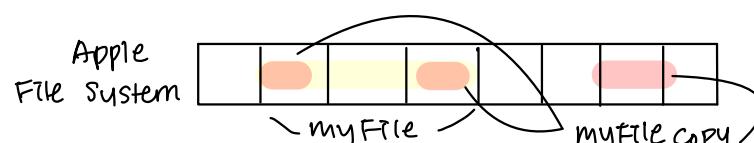
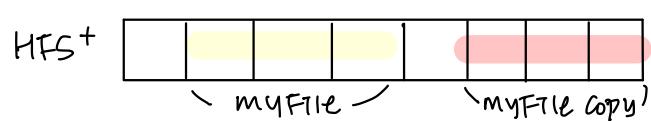
3. Files and Data Persistence: ① Filesystem: create, read, write, and examine files and folders in the filesystem.

Data Persistence

- cloning, snapshots, space sharing, fast directory sizing, atomic safe-save, sparse files (using high-level API in Foundation ex) FileManager, FileHandle

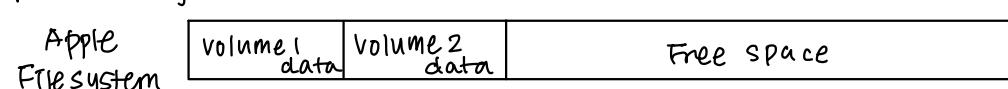
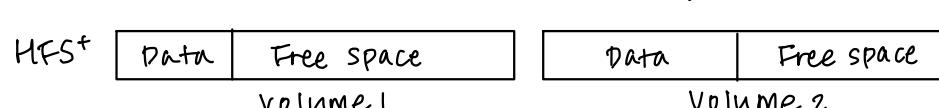
i) Clone: copy of a file or directory that occupies no additional space on disk

modification to the data are written elsewhere, both files continue to share unmodified blocks



ii) Space shared: many filesystem support only a single volume per partition because free space can't be shared across partitions, each volume's size is set when partitioning the storage device, and each volume can only grow into its available free space.

But, Apple File System supports multiple volumes within a single partition → can share free space
 All of volumes can grow + shrink independently



iii) Sparse files: on-disk blocks are allocated only when those blocks are actually written to. This behavior lets files that contain blank sections, such as disk images and database dumps, be saved on disk more efficiently.

② Archives and serialization: convert objects and values to and from Property List, JSON, and other flat binary representation.

i) encoding/decoding: Codable - a type that can convert itself into and out of an external representation.
 Custom types

- typealiases for the encodable/decodable protocols. When you use Codable as a type or a generic constraint, it matches any type that conforms to both protocols

③ Preference: persistently store domain-scoped pieces of information for configuring your app.

i) App-Specific Data: UserDefaults - an interface to the user's defaults database, where you store key-value pairs persistently across launches of your app

ex) allow users to specify their preferred unit of measurement or media playback speed. App stores these preferences by assigning values to a set of parameters in a user's default database.

- setting default value: changed synchronously within process, asynchronously to persistent storage and other processes

④ Spotlight: search for files and other items on the local device, and index your app's content for searching

⑤ iCloud: manage files and key-value data that automatically synchronize among a user's iCloud devices.

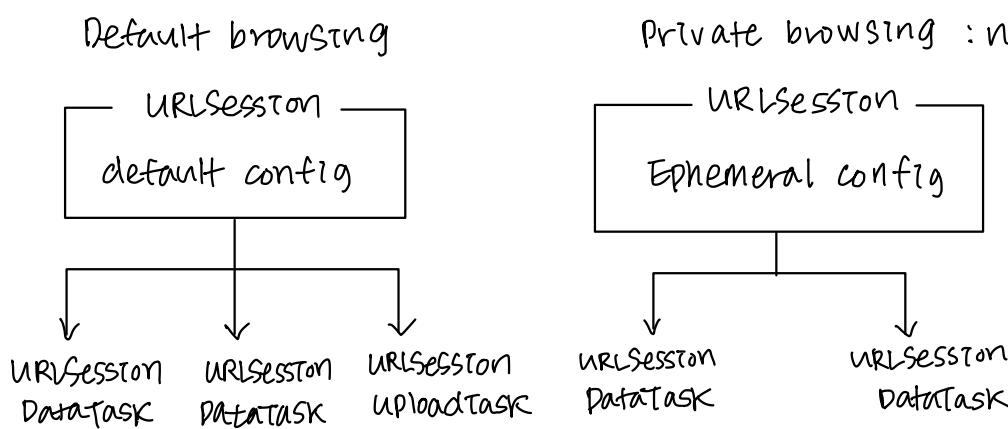
i) Synchronizing app: store preference, configuration, and app state data in the iCloud key-value store preferences using NSUbiquitousKeyValueStore class

ii) Optimizing app data for iCloud backup [store your app's data in the appropriate standard app directories mark files that iCloud doesn't need to backup]

4. Networking

→ HTTP or custom protocol

- Uploading ① URL loading system: Interact with URLs and communicate with servers using standard Internet protocols
- loading is performed asynchronously, so app can remain responsive and handle incoming data/events
 - URLSession / URLSessionTask: can fetch and return data, download files, upload data to remote
 - URLSessionConfiguration: controls behavior like how to use caches and cookies / connections on cellular network



- each session is associated with a delegate to receive periodic updates (or errors)
- the default delegate calls a completion handler block
- configure a session to run in the background so that while the app is suspended, the system can download data on its behalf and wake up the app to deliver the results

i) Fetching Website Data into Memory

- receive results with a completion handler: task delivers the server's response, data, error



dataTask:

- ① verify error is nil, otherwise handle the error and exit

- ② check response to verify that status code indicates success and expected type. otherwise handle the server error and exit.

② Uploading Data to a website: Post data from your app to servers

i) Prepare data for upload: JSONEncoder class on Encodable types like arrays and dictionaries

```
struct Order {
    let customerId: String
    let items: [String]
}
```

```
let order = Order(customId: "12345", items: ["cheese", "soda"])
guard let uploadData = try? JSONEncoder().encode(order)
else { return }
```

ii) Configure an uploading request: Set the httpMethod property of the request to "POST" or "PUT"

```
let url = URL(string: "https://example.com/post")!
var request = URLRequest(url: url)
request.httpMethod = "POST"
request.setValue("Application/json", forHTTPHeaderField: "Content-Type")
```

iii) Create and start an upload task

③ Uploading Streams of data: Send a stream of data to a server

Downloading: ①

1. modules and source files

- Module: single unit of code distribution - framework / application that's built and shipped as a single unit and can be imported using import keyword.
- Source file: a single source file can contain definitions for multiple types, functions, ... (within a module)

2. Access levels

Subclassing.

- Open access / public access: enable entities to be used within any source file from their defining module, and also in a source file from another module that imports the defining module.
- VS. - specifying the public interface to a framework → classes and class members
- Internal access: enables entities to be used within any source file from their defining module, but not in any source file outside of that module.
 - defining an app's or a framework's internal structure. → Default Access Level.
- File-private access: restricts the use of an entity to its own defining source file. Use it to hide the implementation details of a specific piece of functionality when those details are used within an entire file.
- VS. - hide the implementation details of a specific piece of functionality when those details are used only within a single declaration.
- Private access: restricts the use of an entity to the enclosing declaration, and to extensions of that declaration that are in the same file
 - hide the implementation details of a specific piece of functionality when those details are used only within a single declaration.
 - & also subclass or override outside. (target)

Syntax open: can access all data members and member functions within the same module / outside public class PublicClass {} : same as open, but can't subclass or override outside the module. internal class InternalClass {} : default, allows all data members + member funcs to be accessed within the same module only and restrict access outside the module. fileprivate class FileprivateClass {} : same as private, difference, it allows access in a subclass within the same file.

```
public var PublicVar = 0
```

```
internal let InternalConst = 0
```

```
fileprivate func FileprivateFunc() {}
```

private func PrivateFunc() {} : access data members and func within its enclosing declaration, and extension within the same file. Doesn't allow access in a subclass within the same file or in another file

① Explain difference b/w open vs. public

public - cannot subclass public entity outside the module

but in open - you can inherit a class or override a method.

NetworkFramework / NetworkManager.swift
created inside NetworkFramework with public

open

```
public class NetworkManager: NSObject {
    public func createRequest() {}
```

error: cannot inherit non-open class
outside of its defining
module

```
import NetworkFramework.
```

```
class LoginNetworkManager: NetworkManager {}  
override func createRequest() {}
```

error: overriding non-open instance
method outside of its defining module

② Explain the difference b/w internal vs. public

public - access enables data member to be used within any source file from their defining module, and also in a source file from another module that imports the defining module

internal - access enables entities to be used within any source file from their defining module, but not in any source file outside of that module.

③ Explain the difference between private vs. fileprivate

private - allow access data members and function within its enclosing declaration + extension within the same file, i) within the same source file, if property or func is declared as private - scope is by default the class and extension of that class only.

```
class NetworkManager: NSObject {
    private var urlString: String = ""
    private func createRequest() {}
```

```
extension NetworkManager {
    func get urlString() {}
```

accessing entities
inside extension.

class NetworkManagerSubclass: NetworkManager {
 func get urlString() -> String {
 createRequest()
 return urlString
 }
}

```
func get urlString() {}  
data members defined in superclass
```

```
create urlString() {}  
createRequest()
```

```
print("\( urlString)")
```

ERROR!
inaccessible due to
private.

Private: extension available but not subclassing

ii) in the different source file, if `Property(func)` is declared as `private` in one source file and access within extension / subclass in another source file - access not allowed.

in `NetworkUtility.swift`

```
class NetworkManagerSubclass : NetworkManager {  
    func getURLString() -> String {  
        createRequest()  
        return urlString  
    }  
}
```

```
extension NetworkManager {  
    func getURLString() {  
        createRequest()  
        print("\(urlString)")  
    }  
}
```

Error!
`Private Protection`
(`CTC TUTORIAL`)

`fileprivate`: allow access data members and func within the same source file either in `subclass` or `extension`

i) within the same source file, if we create an extension or subclass within the same class file and try to access `fileprivate` entity in its extension / subclass

`NetworkManager.swift`

```
class NetworkManager : NSObject {  
    fileprivate var urlString = ""  
    fileprivate func createRequest() {}  
}
```

```
class NetworkManagerSubclass : NetworkManager {  
    func getURLString() -> String {  
        createRequest()  
        return urlString  
    }  
}  
extension NetworkManager {  
    func getRequest() {  
        createRequest()  
        print("\(urlString)")  
    }  
}
```

ii) in the different source file, `fileprivate` behaves the same as the `private`

④ Where can we apply access levels?

- classes, structs, enumerations
- properties, functions, computed properties, and subscripts
- custom types, and nested types

open > private > internal > fileprivate
|
> private.
inherit
extension

⑤ How do access levels work with tuple?

Tuple type don't have a standalone definition in the way that classes, structs, enumerations, and functions do. A tuple type's access level is determined automatically from the types that make a tuple type, and cannot be specified explicitly.

ex) if you compose a tuple from 2 different types, one with `internal` access, and one with `private` access then what will be the access level of the tuple?

→ access level for that compound tuple type will be `private` because most restrictive access level is applied

Refactoring: describes the constant restructuring of code all the way to the simplest design

1. all unit tests are running
2. the code communicates all of its design concepts.
3. the code does not contain redundancy (duplicated code or logic)
4. Under the above rules, the code contains the smallest possible number of classes and methods

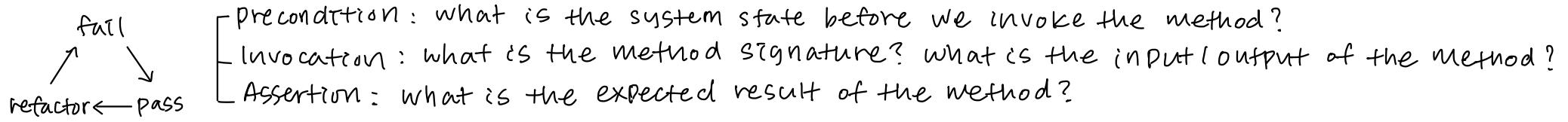
:- Process of changing a software system in such a way that it does not alter function of the code yet improves its internal structure.

functionality same, improve
internal structure.

Unit testing vs. TDD (Test Driven Development)

Unit testing: write code → write tests to verify the logic and code that you wrote.

TDD: first write tests that fail and then write code to make the tests pass, and refactor code to be simple.



HLS: 가장 널리 사용되는 video streaming protocol

- 비디오 파일을 다운로드 할 수 있는 HTTP 파일 조각으로 나누고 HTTP 프로토콜을 이용하여 전송 - Client 장치는 HTTP 파일을 로드한 후 비디오로 재생.
- 장점: 모든 인터넷 연결 장치가 HTTP를 지원하기 때문에 전용서버가 필요한 스트리밍 protocol보다 간단하게 실행 재생에 지장을 주지 않고 네트워크 상태에 따라 비디오 품질을 높이거나 낮출 수 있음.

HTTP Live Streaming

HTTP: 네트워크에 연결된 장치 사이에 정보를 전송하기 위한 application layer protocol.

일반적으로 사용자가 액세스할 수 있는 모든 웹사이트와 application은 HTTP에서 실행 (응답과 요청)

HLS의 경우 Client - server 사이의 연결은 streaming 기간동안 열려있고 server는 비디오 data를 client가 불러갈 때마다 client는 비디오 data segment마다 요청하지 않아도 됨

작동원리

(주문형 스트리밍)

(라이브 스트리밍)

① Server: 미디어 파일이 저장된 서버 / 스트리밍이 제작된 서버에서 시작. 모든 web server streaming 시작.

i) encoding: 비디오 데이터 포맷을 다시 설정하여 모든 장치가 데이터를 인식하고 해석. (H.264, 사용)

ii) segmentation: 비디오는 몇 초 길이의 segment로 나눔 (기본 길이 6초, 2016년도 16초)

(고객화)

: 비디오 segment index file을 만들어 segment의 순서 기록.

: 480p, 720p, 1080p 다양한 품질로 여러세트를 복제

② 배포: encoding된 video segment는 client 장치가 streaming을 요청하면 client 장치로 전송.

contents delivery network가 여러 지역으로 streaming을 배포하는데 도움, CDN streaming을 caching하여 client 신속히 전송 가능

③ Client 장치: streaming을 받고 비디오를 재생하는 사용자의 장치. index file을 참조하여 비디오를 순서대로 조합하고 때문에 따라 quality를 높이거나 낮춤.

Adaptive Bitrate Delivery: network의 상태변화에 따라 streaming 중 품질을 조정하는 능력.

network의 상태가 나빠져도 비디오를 계속 재생할 수 있고, 지원할 수 있는 한계까지 품질 확대화.

segment process 중에 HLS가 다양한 품질로 streaming segment를 복제하여 ABR 가능

MPEG-DASH: run over HTTP, use TCP as their transport protocol / break video into segments with an accompanying index file, and offer adaptive bitrate streaming.

HLS
VS
difference: encoding formats ↗ DASH: use of any encoding standard.
VS
HLS: H.264, H.265.

HDS (adobe)

Device support ↗ DASH: Apple product x

HLS: Apple device

segment length ↗ DASH: between 2-10 seconds in length.

HLS: default 6 seconds

standardization ↗ DASH: international standard

HLS: developed by Apple

AVKit + AV Foundation.

Design Pattern

Asynchronous Programming - weak on ARC?

CA / CP

AVKIT: Create user interfaces for media playback, complete with transport

controls, chapter navigation, picture-in-picture support, and display of subtitles and closed captions

- **AVPlayerViewController:** View controller that displays content from a player and presents a native user interface to control playback

adding media playback capabilities that match the styling/features

Class **AVPlayerViewController:** UIViewController

of native system players

① Support AirPlay: Airplay lets users stream media - need to configure

② Adopt picture in picture playback (PiP) - lets users minimize the video player to a small floating window so they can perform other activities in the primary app

③ Customize tvOS Playback Experience

↳ 이전에 배운 것처럼.

AVFoundation: work with audiovisual assets, control device cameras, process audio, and configure system audio interactions - capturing, processing, synthesizing, controlling, importing/exporting audio visual media on Apple platforms.

① Media assets and metadata: load, inspect, export media assets and metadata, and perform low-level reading and writing of media sample data

i) loading media data asynchronously: simplify async property loading using language level conc. feat.

ex) let asset = AVAsset(url: url)

asset.loadValuesAsynchronously(forKey: ["duration"]) { → loading the value of duration async.
var error: NSError?

if asset.statusOfValue(forKey: "duration", error: &error) == .loaded {

 presentDuration(asset.duration) → access property value synchronously.

}

* Beginning iOS 15, introduces API builds on the new concurrency - can write media inspection code
let duration = try await asset.load(.duration) → load asset's track property value synch.

ii) **AVAsset:** an object that models timed audiovisual media (container object) NSObject.

video, audio, supplementary tracks (captions, subtitles, timed metadata)

(load the tracks for an asset by asynchronously loading its tracks property)

Class **StreamingVideoPlayer**

playerViewController = AVPlayerViewController()

AVKIT.

avPlayer = AVPlayer() : used to manage the playback + timing of a media asset (AVFoundation)

func play(url: URL) { → AVFoundation → url을 가진 asset으로 만들.

let asset = AVAsset(url: url)

let playerItem = AVPlayerItem(asset: asset) : stores a reference to an AVAsset object, need to access info before enqueue for playback

avPlayer.replaceCurrentItem(with: playerItem)

playerViewController.player = avPlayer

playerViewController.player?.play()

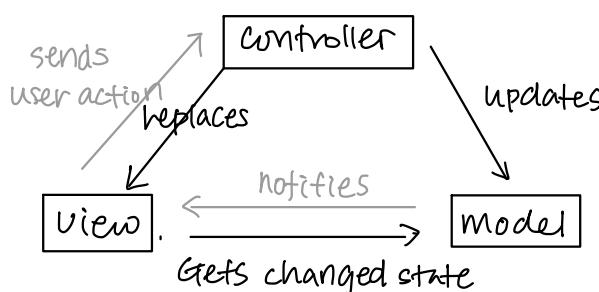
Feature of good architecture

- Balanced distribution of responsibilities among entities with strict roles
- Testability
- ease of use + low maintenance cost

Entities divided to

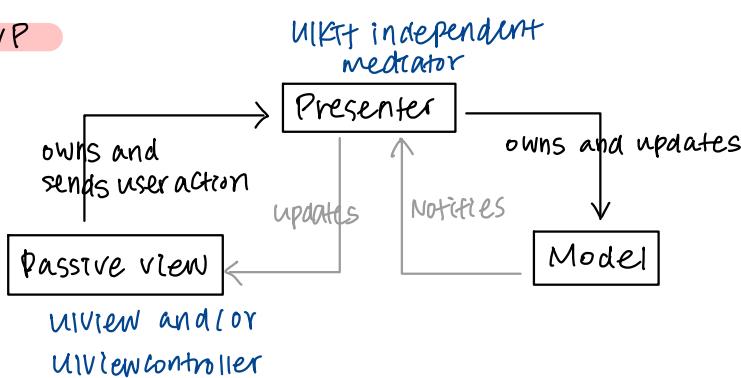
- understand them better
- reuse them (view and model)
- test them independently.

Traditional MVC

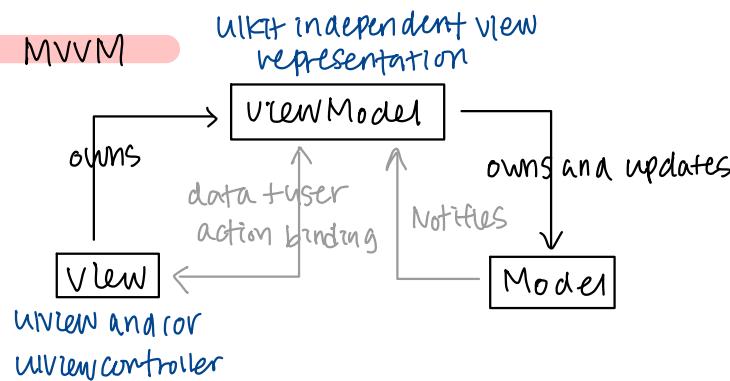


- all three entities are tightly coupled
- each entity knows another two - reusability↓

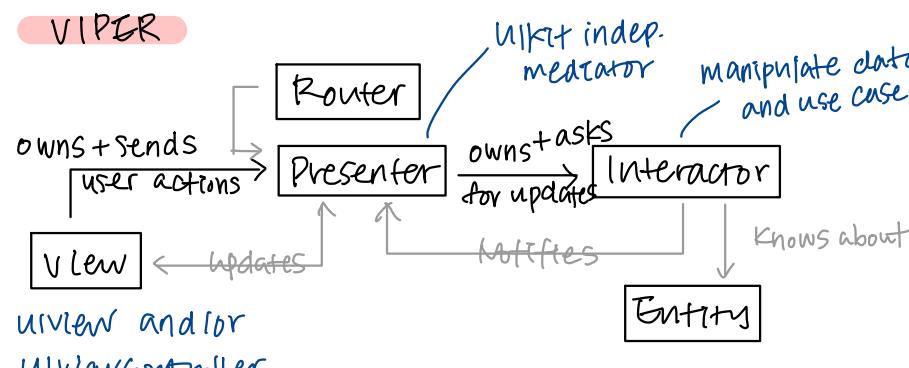
MVP



MVVM



VIPER

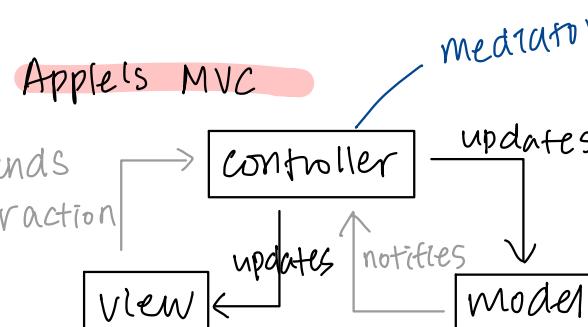


- model logic shifted into Interactor with Entities as dumb data structure.
- only UI repr. duties of controller/presenter/viewmodel moved into the presenter, not data altering cap.
- addresses navigation responsib. resolved by Router.

Distribution: good.

Testability: good.

Easy: cost of maintainability,



- Controllers are involved in view's lifecycle - separate X
- delegate + datasource + dispatching + cancelling network requests.

- : tightly coupled with the view → hard to test
- Distribution: view and model separated, view and controller coupled.
- Testability: bad distribution, only test model.
- Ease of use: least code among patterns - easily maintained + speed.

* Presenter: nothing to do with lifecycle of viewController + no layout code
→ responsible for updating the view with data and state

- Distribution: most responsibilities divided b/w Presenter, model
- Testability: can test most of business logic
- Ease of use: Code↑ (long)

View Model: UIKit independent representation of view and state.
invokes changes in the model and updates itself w/ updated model
Bindings: R2DataBinding / SwiftBond (KVO based)
ReactiveCocoa, RXSwift, PromiseKit (functional reactive programming)
Distribution: View↑ - view updates update state from VM - bindings
Testability: VM doesn't know about view → testability↑
View - UIKit dependant → eh.

Easy: skinnier than MVP by binding.

- Interactor: contains business logic related to data (entities) or networking
- Presenter: contains UI related (UIKit indep) business logic, invokes methods on the Interactor
- Entities: plain data object, not data access layer.
- Router: responsible for the segue between the VIPER module.

Tested solution, code unification. Common vocabulary

Types

- ① **creational:** deal with creation mechanisms – instantiate objects for particular situation.
 - ex) factory method, abstract factory, builder, singleton, prototype.
- ② **structural:** simplify the design by finding an easy way of realising relationship b/w class and objects
 - ex) adapter, bridge, facade, decorator, composite, flyweight, proxy.
- ③ **Behavior:** identify common communication patterns b/w entities
 - ex) chain of responsibility, template method, command, iterator, mediator, memento, observer, state, visitor.

 Builder: creational

lets construct complex objects step by step. Produce different types and representations of an object using the same construction code

Singleton

Creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.

Problem

The Singleton pattern solves two problems at the same time, violating the Single Responsibility Principle:

1. Ensure that a class has just a single instance. Why would anyone want to control how many instances a class has? The most common reason for this is to control access to some shared resource—for example, a database or a file.

Here's how it works: imagine that you created an object, but after a while decided to create a new one. Instead of receiving a fresh object, you'll get the one you already created.

Note that this behavior is impossible to implement with a regular constructor since a constructor call must always return a new object by design.

2. Provide a global access point to that instance. Remember those global variables that you (all right, me) used to store some essential objects? While they're very handy, they're also very unsafe since any code can potentially overwrite the contents of those variables and crash the app.

- Just like a global variable, the Singleton pattern lets you access some object from anywhere in the program. However, it also protects that instance from being overwritten by other code.

- There's another side to this problem: you don't want the code that solves problem #1 to be scattered all over your program. It's much better to have it within one class, especially if the rest of your code already depends on it.

Nowadays, the Singleton pattern has become so popular that people may call something a singleton even if it solves just one of the listed problems.

Solution

All implementations of the Singleton have these two steps in common:

- Make the default constructor private, to prevent other objects from using the new operator with the Singleton class.

Create a static creation method that acts as a constructor. Under the hood, this method calls the private constructor to create an object and saves it in a static field. All following calls to this method return the cached object.

- If your code has access to the Singleton class, then it's able to call the Singleton's static method. So whenever that method is called, the same object is always returned.

Applicability

Use the Singleton pattern when a class in your program should have just a single instance available to all clients; for example, a single database object shared by different parts of the program.

- The Singleton pattern disables all other means of creating objects of a class except for the special creation method. This method either creates a new object or returns an existing one if it has already been created.

Use the Singleton pattern when you need stricter control over global variables.

- Unlike global variables, the Singleton pattern guarantees that there's just one instance of a class. Nothing, except for the Singleton class itself, can replace the cached instance.

Note that you can always adjust this limitation and allow creating any number of Singleton instances. The only piece of code that needs changing is the body of the getInstance method.

How to Implement

↳ **_singleton (static)**
↳ **singleton instance** ↳ **single count**.
↳ **thread safe**.

1. Add a private static field to the class for storing the singleton instance.
2. Declare a public static creation method for getting the singleton instance.
3. Implement "lazy initialization" inside the static method. It should create a new object on its first call and put it into the static field. The method should always return that instance on all subsequent calls.
4. Make the constructor of the class private. The static method of the class will still be able to call the constructor, but not the other objects.
5. Go over the client code and replace all direct calls to the singleton's constructor with calls to its static creation method.

Pros and Cons

- + You can be sure that a class has only a single instance. → **保证只有一个实例**
 - + You gain a global access point to that instance. → **全局访问点**
 - + The singleton object is initialized only when it's requested for the first time.
-
- Violates the Single Responsibility Principle. The pattern solves two problems at the time. **instance管理→单例=多用途设计原则**
 - The Singleton pattern can mask bad design, for instance, when the components of the program know too much about each other.
 - The pattern requires special treatment in a multithreaded environment so that multiple threads won't create a singleton object several times.
 - It may be difficult to unit test the client code of the Singleton because many test frameworks rely on inheritance when producing mock objects. Since the constructor of the singleton class is private and overriding static methods is impossible in most languages, you will need to think of a creative way to mock the singleton. Or just don't write the tests. Or don't use the Singleton pattern. → **单元测试困难**

let screen = UIScreen.main

let userDefault = UserDefaults.standard

let application = UIApplication.shared

let fileManager = FileManager.default

let notification = NotificationCenter.default

```

class UserInfo {
    static let shared = UserInfo()
    var id: String?
    var password: String?
    var name: String?

    private init() {}
    ↗️ 허락도 init 함수를 통해 instance를
    또 수정방지
}

```

A VC

```
let userInfo = UserInfo.shared
userInfo.id = "hello"
```

B VC

```
let userInfo = UserInfo.shared
userInfo.password = "123"
```

C VC

```
let userInfo = UserInfo.shared
userInfo.name = "Jenny"
```

→ static property 자체 → shared instance로.

Static instance

- ① 향수 끝이 끝까지 있는 경우 - 메모리 대입. - 메모리 초기화 때 한번.
- ② 중위의 대입 - 같은 것의 메모리 주소만 바라보기 때문에.

Static method

utility method처럼 됨 - static value는 static method보다 훨씬 적음.

Create properties and methods that belongs to a type, rather than to instances of a type.
 → organizing data meaningfully by sharing shared data.

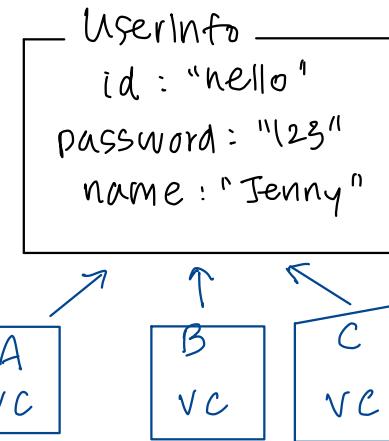
static method belongs to the struct itself, rather than to instances of that struct,
 can't use it to access any non-static properties from the struct.

function: piece of the code that is called by name. → can pass data to operate on
 can optionally return data

method: piece of the code that is called by name that is associated with an object.

diff. ① method is implicitly passed the object on which it was called.

② method is able to operate on data that is contained within the class,
 the object is an instance of the data.



approach

WHAT program should accomplish without specifying HOW the program should achieve the result.

Imperative programming : focus on how to execute, define control flow as statements that directly change a program computed state (datafields)

- direct assignments, common data structures, global variables

Commands for comp. perform.

Declarative programming : focus on what to execute, defines program logic, but not detailed control flow.

- fourth generation languages, spreadsheets, report program generators

Main programming paradigms

- Procedural / Structured prog.: specifies steps a program must take to reach a desired state
 - Structured prog.: a style of imperative prog. with more logical program structure (related to imp. prog.)
 - structograms, indentation, no or limit use of goto statement.
 - Procedural prog.: derived from structured prog., based on the concept of modular programming or the procedural call - local variables, sequence, selection, iteration, and modularization. (related to structured, imperative) - scopes, blocks
 - breaking down a prog. task into a collection of variables, DS, subroutines procedures to operate on DS
- functional prog.: treats programs as evaluating mathematical functions and avoids state + mutable data.
 - lambda calculus, compositionality, formula, recursion, referential transparency, no side effects (related to declarative)
- oop: organizes programs as object: data structures consisting of data fields and methods together with their interaction.
 - (predefined)
 - Objects, methods, message passing, information hiding, data abstraction, encapsulation, polymorphism inheritance, serialization - marshalling. (related to procedural). → expose behavior (method) and data (members / attribute)

Common types using different paradigms

Mouse clicks, key presses

- event-driven programming: program control flow is determined by events, s.a sensor input / user actions / messages from other programs or thread.
 - main loop, event handlers, asynchronous processes (related to procedural, dataflow)
- automata-based prog.: a program, or part, is treated as a model of a finite state machine or any other formal automation.
 - state enumeration, control variable, state changes, isomorphism, state transition table. (related to imperative, event-driven)
- reactive prog.: declarative prog. paradigm concerned with data streams and the propagation of change.

seq. program instructions performs a specific task, packaged as unit. (routine, subprogram, function, method, procedure).

oop do not deliberately alter program state directly, instead using mutator methods to encapsulate these state changes will, as a direct consequence, have more overhead → message passing (subroutine call) 3 added overhead.

① dynamic memory allocation. ② parameter copying ③ dynamic dispatch.

obtaining memory from heap and copying parameters for message passing may involve significant resources that far exceed those needed for the state change. Accessors (getter) that merely return the values of private member variables also depend on similar message passing subroutines, instead of using a more direct assignment / comparison, adding to total path length.

frequency

system is composed of distinguishable pieces (subroutine)

procedural programming: granularity of the code is largely determined by # of discrete procedures / modules

functional programming: frequent calls to library subroutines are common, but may be often inlined by optimizing compiler

object oriented programming: # method calls invoked is also partly determined by granularity of DS, and thus include many read-only accesses to low level objects that are encapsulated, and thus accessible in no other, more direct, way. Since increased granularity is prerequisite for greater code reuse, the tendency is actively discouraged. Constructors also add to the count as they are also subroutine calls. Performance problems caused by excessive granularity may not become apparent until scalability becomes an issue.

other paradigms, where a mix of the above paradigms may be employed, subroutine use is less predictable.

Allocation of dynamic memory for message and object storage.

OSP involves dynamic memory allocation from heap storage for creation + message passing.

→ Heap implementations tend to stay general for all platforms, and hence have heavy overhead.

dynamically dispatched message calls v. direct procedure call overhead.

Heavy use of inheritance and dynamically-bound messages is likely to make code more extensible and reusable, but it also imposes a significant performance overhead, relative to an equivalent but non-extensible program written in non-object oriented manner.

structured graphic packages - perf. cost of the extra flexibility provided by using a heavily oo style acceptable.

basic data structure libraries / numerical computing packages / rendering libraries / trace-driven simulation frameworks

→ cost of message passing ↑ → avoid.

serializing objects.

serialization imposes large overheads when passing objects from one system to other, esp. when transfer is in human readable formats such as Extensible Markup Language (XML), JavaScript Object Notation (JSON). This contrasts with compact binary formats for non-object-oriented data. Both encoding / decoding object's value and its attributes are involved in the serializing process, which also includes awareness of complex issues s.a. inheriting, encapsulating, data hiding.

Difference b/w stack and heap?

- Stack: static in memory and allocation happens only during compile time. Special region of computer's memory that stores temporary variables created by each function main(). LIFO DS. When called, all instances will be pushed on to the current stack and pop when func. returned. Once freed, region of memory becomes available for other stack variables.
 - ① Stack grows and shrinks as func. push + pop local variable.
 - ② no need to manage memory, variables are allocated and freed automatically. Space managed eff. by CPU.
 - ③ Stack has size limits, cannot be resized
 - ④ Stack variables only exist while the func that created them, is running.
 - ⑤ fast access
- Heap: region of computer's memory that is not managed automatically for user ^{by} CPU. Dynamic in memory, and alloc. happens during runtime. Values can be referenced at any time through memory address. Alloc - use malloc() / calloc(), built-in C func. and use free() to dealloc. that memory (garbage collector)
 - ① Variables can be accessed globally
 - ② no limit on memory size
 - ③ slower access
 - ④ must manage memory
 - ⑤ Variables can be resized using realloc()
 - ⑥ no guaranteed efficient use of space, memory become fragmented overtime as blocks of memory are allocated, then freed.

Model codable

BookData

```
books: [Book]
```

Book

```
title, subtitle, isbn13,  
price, image, url
```

BookDetail

```
title, subtitle, authors  
publisher, isbn13, pages,  
rating, desc, price, image  
url
```

WebService Extension

with JSON Decoding < search by keyword
detail book info by isbn

Search response

```
{"total": 0  
"pages": 1  
"books": [{"title": "Practical MongoDB",  
"subtitle": "—"  
"isbn13": "—"  
"image": "—"  
"url": "—"},  
:  
]}
```

Books response

```
{"error": "0",  
"title": "Securing DevOps",  
:  
}
```

Project #1 TodoList.

#1

UI (views)

navigation bar ← leftbar button item
rightbar button item.

scenes ← Category - TableView - Cell - ContentView
TodoList - TableView ← SearchBar
Cell - ContentView

Controller

CategoryViewController : SwipeTableViewController

```
let realm = try! Realm()
var categories = Results<Category>?
```

ViewWillAppear: called before the view is added to the windows' view hierarchy. Ideal for updating viewController's data.

```
guard let navBar = navigationController?.navigationBar else {
    fatalError("!")
}
navBar.backgroundColor = defaultColor
```

TableView DataSource : numberOfRowsInSection → Int : 편의 ITST member? 있나?
Categories? count ?? 1

cellForRowAt → UITableViewCell : Insert in the certain row

```
let cell = super.tableView(tableView, cellForRowAt indexPath)
```

if let category = categories?[indexPath.row] {

TableView Delegate Method : didSelectRowAt → performSegue(withIdentifier: "goToItems", sender: self)
※ storyboard Configured 액션으로 표기

※ 올 DM 틀리고
선택!

prepare(for segue: UIStoryboardSegue, sender: Any?)

let destinationVC = segue.destination as! TodoListController

if let indexPath = tableView.indexPathForSelectedRow {

destinationVC.selectedCategories = categories?[indexPath.row]

}

ToDoListViewController : SwipeTableViewCellController

```
@IBOutlet weak var searchBar: UISearchBar!
var todoItems: Results<Item>?
let realm = try! Realm()
```

```
var selectedCategories: Category?
didSet {
    loadItems()
}
```

SwipeTableViewCellController : UITableViewDelegate, SwipeTableViewCellDelegate {

override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) → UITableViewCell {

let cell = tableView.dequeueReusableCell(withIdentifier: "cell", for: indexPath) as! SwipeTableViewCell

cell.delegate = self

return cell

editActionsForRowAt → [SwipeActions]?

guard orientation == .right else { return nil }

let deleteAction = SwipeAction(style: .destructive, title: "Delete") { (action, indexPath) in
 self.updateModel(at: indexPath)
}

return [deleteAction]

Model

class Category: Object {

objc dynamic var name: String = ""

objc dynamic var color: String = ""

let items = List<Item>()

class Item: Object {

objc dynamic var title: String = ""

objc dynamic var done: Bool = false

objc dynamic var date: Date?

var parentCategory = LinkingObjects(fromType: Category.self,
property: "items")

CoreData

```
AppDelegate : applicationWillTerminate self. saveContext()
    lazy var persistentContainer: NSPersistentContainer = {
        let container = NSPersistentContainer(name: "DataModel")
        func saveContext() {
            let context = persistentContainer.viewContext
            if context.hasChanges {
                do {
                    try context.save() ...
                }
            }
        }
    }
```

AppDelegate TUTORIAL
data²環状 cycle of TBL
(applicationWillTerminate)

CategoryViewController : UITableViewController

```
let context = (UIApplication.shared.delegate as! AppDelegate).persistentContainer.viewContext
```

```
def saveCategories() {
    do {
        try context.save()
    } catch {
        print("error")
    }
}
```

```
def loadCategories() {
    let request: NSFetchedRequest<Category> = Category.fetchRequest()
    do {
        categories = try context.fetch(request)
    } catch {
        print("Error")
    }
}
```

ToDoListViewController : UITableViewController

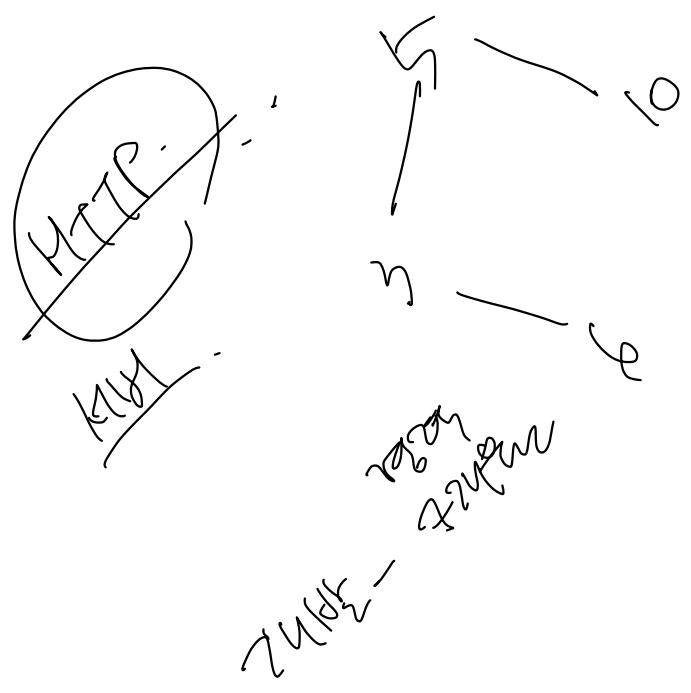
```
var itemArray = [Items]()
let context = (UIApplication.shared.delegate as! AppDelegate).persistentContainer.viewContext

func loadItems(with request: NSFetchedRequest<Items> = Items.fetchRequest(),
               predicate: NSPredicate? = nil) {
    let categoryPredicate = NSPredicate(format: "parentCategory.name MATCHES %@", selectedCategories!.name!)
    if let additionalPredicate = predicate {
        request.predicate = NSCompoundPredicate(andPredicateWithSubPredicate: [categoryPredicate, additionalPredicate])
    } else {
        request.predicate = categoryPredicate
    }
    do {
        itemArray = try context.fetch(request)
    } catch {
        print("error")
    }
}
```

extension ToDoListViewController : UISearchBarDelegate {

```
func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {
    let request: NSFetchedRequest<Items> = Items.fetchRequest()
    let predicate = NSPredicate(format: "item CONTAINS %@", searchBar.text!)
    request.predicate = predicate
    let sortDescriptor = NSSortDescriptor(key: "item", ascending: true)
    request.sortDescriptor = [sortDescriptor]
    loadItems(with: request)
}

func searchBar(_ searchBar: UISearchBar, textDidChange searchText: String) {
    if searchBar.text?.count == 0 {
        loadItems()
    }
    DispatchQueue.main.async {
        searchBar.resignFirstResponder()
    }
}
```



~~solid~~

ISX
12 min

Binary tree

Bin

return

BT
BSF

T/F

tail
recursion

tail
recursion

tail
return

G/F

Backtrace
var

tail
on? not on

data
tail
point
troubleshooting

tail
compiled
language

tail
recursion