iOS

- Bounds 와 Frame 의 차이점을 설명하시오.

inside

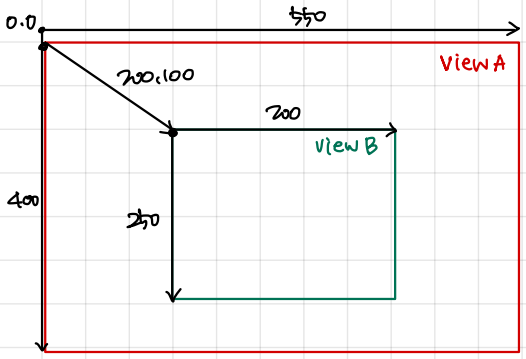
- Bounds of UIView is the rectangle, expressed as location (x,y) and size (width, height) relative to its own coordinate system (0,0).

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

outside

ex) size 100 x 100 (w,h) positioned at (x, y) 25, 25 of its superview

bounds	frame
bounds.origin.x : 0	frame.origin.x : 25
bounds.origin.y : 0	frame.origin.y : 25
bounds.size.width : 100	frame.size.width : 100
bounds.size.height : 100	frame.size.height : 100



ViewA frame :
origin : (0,0)
size : 550 x 400

B frame
(200,100)
200 x 250

ViewA Bounds:
origin : (0,0)
size : 550x400

B Bounds
(0,0)
200x250

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

Hardware : 가속도 센서, 가깝게 센서, 주변광 센서, GPS 센서 이용 X
마우스로 simulator touch 하기에 zoom in/out 기능 X
카메라, 마이크, 전화기능 사용 불가능.

API : Apple의 push notification 받기와 보내기 불가능.
사전, 연락처, 캘린더 액세스하기 위해 개인 정보 알림 지원X
hand off 기능 X / message ui 기능 X

ETC : 맥의 성능이 아이폰의 성능보다 뛰어나 cpu나 메모리 부담 확인 불가
네트워크 속도 테스트 측정 X / Face ID 얼굴 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 끝난후 화면이 사용자에게 보여주기 직전 호출 (초기화 코드 작성)

applicationWillTerminate(_:) : 앱 종료되기 전에 호출
(메모리 확보 위해 suspended 상태에 있는 앱이 종료될 때나 background 상태에서 사용자에 의해 종료될때나 업뎃로 인해 종료시 호출 X)

Inactive

sceneWillEnterForeground(_:) : 앱이 백그라운드나 non-running에서 foreground로 들어가기 직전에 호출 (비활 -> 활성화)

sceneWillResignActive(_:)

Active

sceneDidBecomeActive(_:) : 앱이 비활 상태에서 활성상태로 전환하고 난후 호출
실제로 사용자가 앱에 접근할 수 있는 코드를 작성할 수 O

Background

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

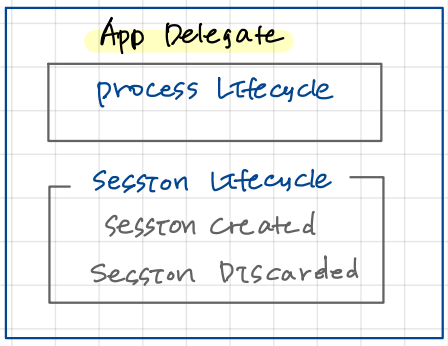
Suspended

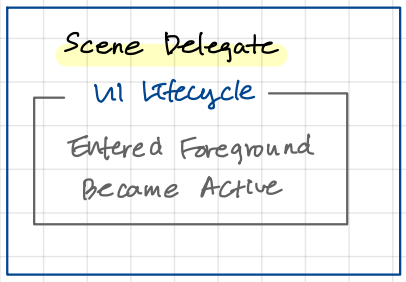
따로 호출되는 메소드는 없으며 background 상태에서 특별한 작업이 없을 때

1. Not Running : 앱이 실행되지 않았거나 완전히 종료되었을 때
2. Inactive : 앱이 실행되었지만 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 → 멀티

- scene delegate에 대해 설명하시오.





- iOS 13 부터 UILifeCycle에 대한 부분을 SceneDelegate가 함.

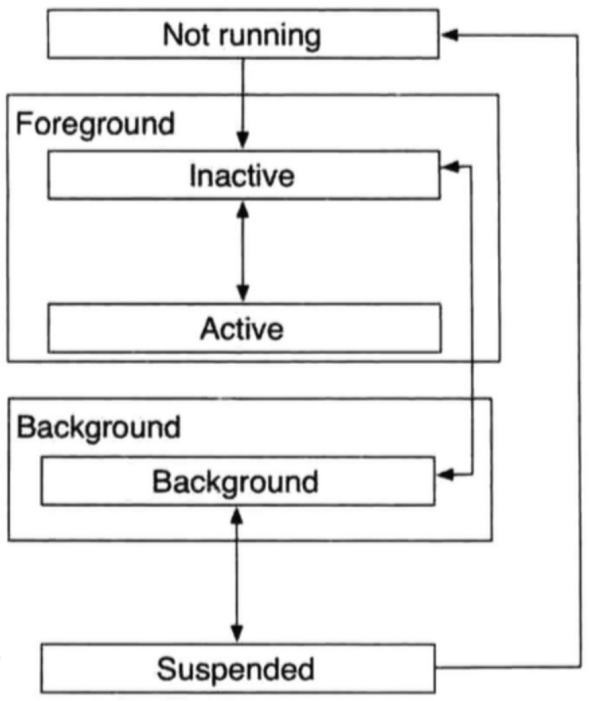
Scene?

> UIKit는 UIWindowScene 객체를 사용하는 앱 UI의 각 인스턴스를 관리합니다. **Scene에는 UI의 하나의 인스턴스를 나타내는 windows와 view controllers가 들어있습니다.** 또한 각 **scene에 해당하는 UIWindowSceneDelegate 객체**를 가지고 있고, 이 객체는 **UIKit와 앱 간의 상호 작용을 조정**하는 데 사용합니다. Scene들은 같은 메모리와 앱 프로세스 공간을 공유하면서 서로 동시에 실행됩니다. **결과적으로 하나의 앱은 여러 scene과 scene delegate 객체를 동시에 활성화**할 수 있습니다.
> (Scenes - Apple Developer Document 참고)

UI의 상태를 알 수 있는 UILifeCycle에 대한 역할을 `SceneDelegate` 가 하게 됐죠! 역할이 분리된 대신 `AppDelegate` 에서 Scene Session을 통해서 scene에 대한 정보를 업데이트 받는데요! 그럼 Scene Session은 뭘까요??

Scene Session?

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



Q : 앱이 Inactive 상태가 되는 시나리오를 설명하시오.

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

Method 순서는

Not running 상태에서 앱을 실행하고 foreground 직전에 applicationDidFinishLaunching이 호출 되고 foreground 진입시 sceneWillEnterForeground가 호출되며 액티브 상태.

유저가 백그라운드 진입했을 때 sceneDidEnterBackground 가 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.

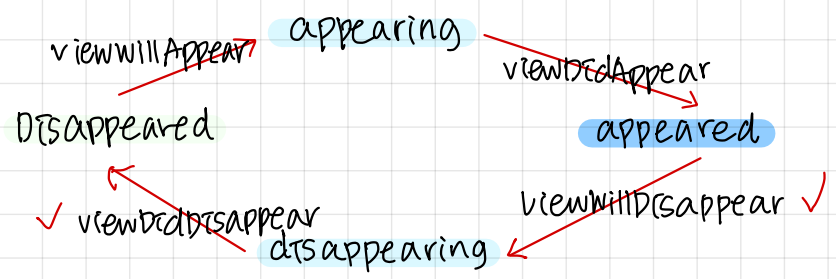
⑤ **suspended state**: remains in memory but does not execute any code.
When a low memory condition occurs, the system may purge suspended apps without notice to make more space for the foreground app

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.

- loadViewIfNeeded: 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 obj 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, dismiss, covered, or hidden



iOS13부터 AppDelegate가 하는 일?

> 이전에는 앱이 foreground에 들어가거나 background로 이동할 때 앱의 상태를 업데이트하는 등의 앱의 주요 생명 주기 이벤트를 관리했었지만 더이상 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 서브스와 같이 실행시 요구되는 모든 서비스를 등록하는것.**
> **입니다.**
> (UIApplicationDelegate - Apple Developer Document 참고)

1. Protocol / Delegate

protocol : 선언만 되고 구현되지 않은 method.
기능을 정의하고 method 구현해야 됨.

delegate : 대표, 위임의 뜻, 해당 객체가 모든 일을 처리하는것이 아니라 처리해야하는 일 중 일부를 다른 객체로 넘기는 것.
textField.delegate = self
- 대리자가 누구인지 알려주는 과정 → "textField 둬바라지는 내가!"
너에게 이벤트가 발생하면 프로토콜에 따라 너에게 응답을 줄게

2. 동기 / 비동기 작업 차이
동기 : 작업이 완료 될 때까지 기다린다.

비동기 : 백그라운드에서 작업이 완료되면 알림을 받을수 있다.

3. strong, weak, read only, copy 의 차이는?
- strong : 참조 카운트가 증가될 것이고 reference (참조)는 그 객체의 생명주기를 통해 유지 될 수 있음.
- weak : 객체에 포인터를 가르키고 있지만 RC를 올리지 X.
부모 자식 관계. 부모는 자식에게 strong ref.
자식은 부모에게 weak ref.
- read only : 초기에 속성 세팅할 수 있지만 변경할 수 X
- copy : 객체가 생성 될 때 객체의 값을 복사한다는 의미 (value)

4. CompletionHandler
작업이 끝났을 때 API 호출하고 싶을 때 사용 (closure)
NSData?, 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 사각형이고,
- UIView의 Frame은 그 뷰를 담고 있는 parent와 연관된 position (x,y) 와 size (width, height)로 표현 사각형.

6. 함수란?
어떤 작업을 수행하기 위해 일련의 명령을 모을 수 있게 해준다. 코드상에서 사용 가능하며 / 중복되는 명령을 찾는 다면, 함수가 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 자원 공유 용이
: 공통된 객체를 여러개 생성 상황 (cache, threadpool ...)
- 단점 : Singleton instance가 너무 많은 일을 하거나 데이터 공유↑, 다른 class instance간 결합도 높아져 "개방-폐쇄" 원칙 위배
: 수정과 테스트가 어려워짐

8. Observer 패턴이란?
오브젝트의 상태변화를 다른 obj에게 알려준다.

9. 디자인 Pattern ?
MVC : Model - 도메인 데이터의 역할이나 데이터를 조작하는 레이어를 접근하는 데이터.
: View - 표현하는 레이어 (GUI)의 역할. UI
: Controller - Model과 View의 중개인, View에서 실행된 사용자 행동에 반응하여 Model을 바꾸고 View에 갱신하는 역할.
(presenter viewModel)

MVVM : View와 그 상태의 표현이 독립적인 UI다.
ViewModel은 Model에서 바뀐 것을 부르고 갱신된 model로 그 자신을 갱신.

10. JSON / PLIST의 한계?
- obj 생성하고 디스크에 serialize
- 느림
- serialize / deserialize
- thread safe X

11. SQLite의 한계?
- 관계형 DB - table 마다 scheme 정의
- 손수 query 작성. → 오브젝이 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 다중 thread / 경로.
- concurrency : 한번에 여러 작업을 실행시키는 것.
- queue : FIFO 방식으로 obj 관리하는 경량의 자료 구조.
- sync vs async

16. Grand Central Dispatch (GCD)
두에서 thread를 관리하면서 동시적으로 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 전역사용가능

17. guard의 장점
- 파라미터 구조의 코드 피할 수 있음 (if let)
- break, return을 사용하여 초기에 함수 빠져나가게.

18. method Swizzling
#Selector (method) mapping 변경 → 런타임시 다른 것으로 대체

19. nonescaping / escaping closure.
~~함수와 클로저 정의~~ → 실행 → 함수 return.
정의 → 실행 → async → escaped.
stored

20. weak, unowned ?
- unowned는 nil이 될수 X, optinal X
- instance가 메모리 해제된 후 절대 사용되지 않을 거라고 확실시 사용

21. ARC ?
- 컴파일 타임의 자동 메모리 관리 기능
- strong ref가 0이 될 때만 obj을 메모리에서 해제.

22. CollectionView | Tableview
- Tableview : 여러 항목들의 list을 보여주고, 수직적 방법 수직 스크롤도 제한
- CollectionView : 여러항목을 보여주지만 여러 가로줄+조우.

23. REST, HTTP, JSON ?
- HTTP는 app protocol이며 규칙의 집합 Server - client 데이터 전송에 사용.
- REST : 일관성 있게, 규칙, Web API 쉽게 만들수 있도록.
- JSON : 직관적이고 사람이 읽게 시스템간 데이터 전송.

24. Delegation이 해결할 수 있는 문제.
- obj 객이는 것을 막는다.
- 동작이나 모습 수정.

25. Down casting
- as : upcasting
- as? : 실패시 nil
- as ! : 강제

26. IB outlet | IBAction? outlet
- IBOutlet : interface builder is needed to associate properties in your app with components in IB.
- IBAction : Interface Builder ACAion 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 rived)
  - 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. it-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 = gati.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 모드와 상태

② Foreground
- active
- inactive

③ Background
- running
- suspended

not running → active : app 실행 상태

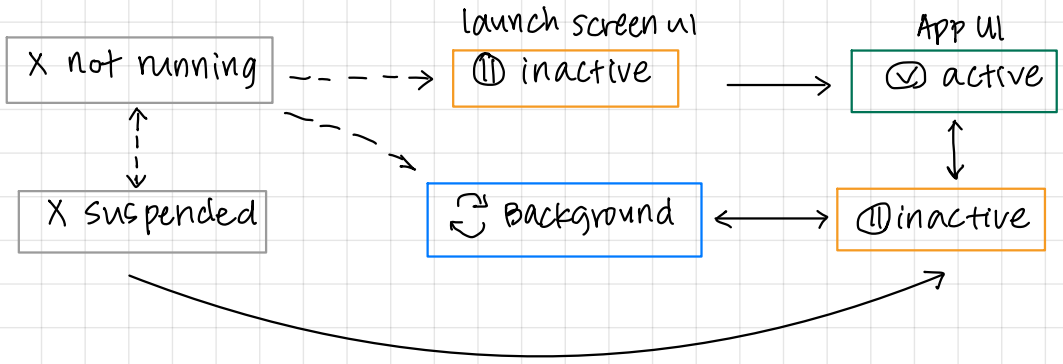
active → inactive → running : app 활성화 상태에서 비활성화 상태로 만든뒤 백그라운드에서도 계속 실행중인 상태.

active → inactive → suspend : app 활성화 상태에서 비활성화 상태로 만든뒤 백그라운드에서도 정지되어있는 상태.

running → active : 백그라운드에서 실행 중인 app이 foreground에서 활성화.

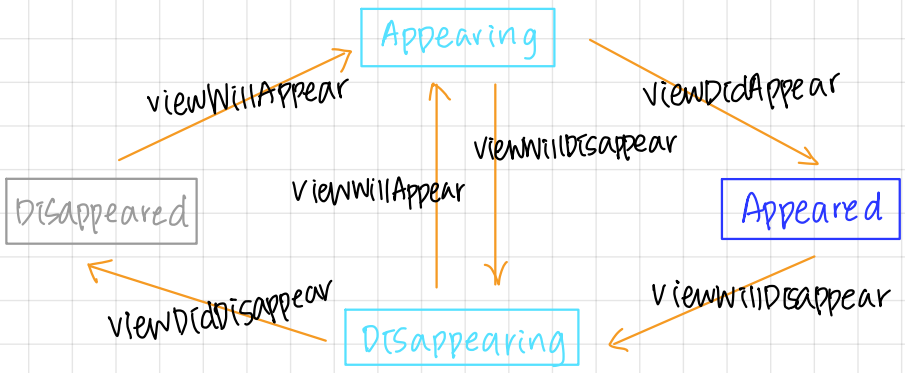
App and Environment : manage life-cycle events and your app's UI scenes, and get info about traits and the environment in which your app runs

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.



* ViewDidAppear
쓰는곳 + data 사용시 저장.

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

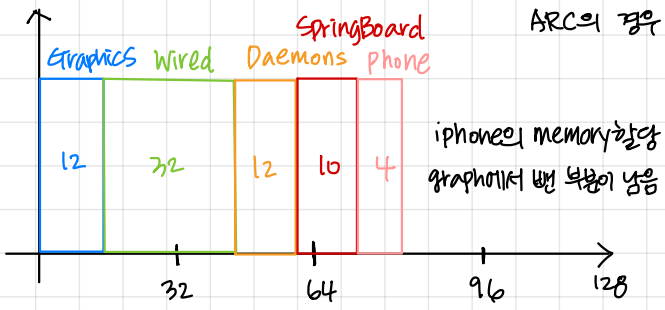
---

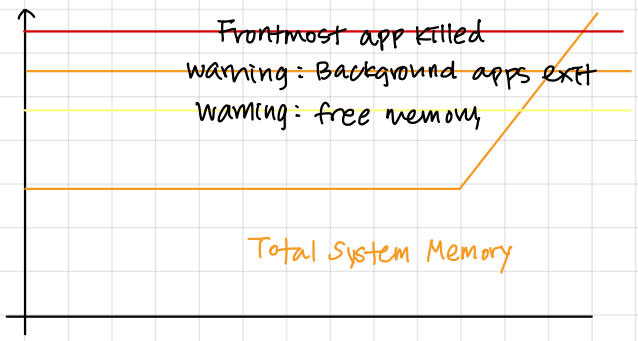
3. Delegate vs. Block vs. Notification.

- Delegate는 객체간의 데이터 통신을 할 경우 전달과 역할을 합니다. - event 처리 : 특정 객체에서 발생한 event나 data를 다른 객체에게 통보.
  Delegate에게 알릴 수 있는 것은 여러 event가 있거나 class가 delegate로 부터 데이터를 가져와야 할 때 사용. ex) UITableView.
- Block은 event가 하나일 때 사용하기 좋습니다. Completion block 사용예 ex) NSURLConnection sendAsynchronousRequest : queue : completionHandler
- Notification은 event에 대해 여러 listener 있을 때 사용. ex) 내가 특정 event를 기반으로 정보를 업데이트 하는 방법을 not.으로 broadcasting 하여 변경하거나 옵저 창을 닫을 때 옵저의 객체가 저장되는거 확인

---

4. Memory management : 기존 MRC의 경우 retain/release/autorelease (obj-c)
ARC의 경우 compiler가 알아서 처리 (Swift)





* 면접 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 사각형
- Bound : 자체 좌표계 (0,0)을 기준으로 위치 (x,y) 및 크기 (너비, 높이)로 표현되는 사각형.

FAQS

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

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

Struct vs Class

Struct

- call by value (할당 / parameter 전달시 value copy)
- stack memory 영역에 할당 (속도가 빠름)
  - scope based lifetime : 컴파일타임에 compiler가 언제 메모리를 할당/해제할지 정확히 알고 있음.
  - data locality : CPU Cache에 올림
- 상속 불가능 (protocol은 사용 가능)
- NSData로 Serialize 불가능
- Codable 프로토콜을 이용하여 손쉬운 JSON ↔ struct 변환 가능.
- 항상 새로운 변수 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 allocation하며 reference counting을 통해 dealloc 필요함.
  - memory fragmentation 등의 overhead 존재.
- 상속 가능
- NSData Serialize 가능
- Codable 사용 불가능
- runtime에 type casting을 통해서 class instance에 따라 여러 동작이 가능

RULE

STRUCT : 상속이 필요 X, 모델의 사이즈 ↓, JSON 필드와 1:1 mapping
CLASS : Serialize해서 전송해야하거나 파일로 저장, obj-c 사용시.

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 : Queue 에 추가된 작업들이 각 순서에 맞게 시작되고 순서에 해당되는 작업이 끝나야만 그 다음 작업 시작.
  - 직렬 · 동기 : Queue 작업들의 시작 및 종료 순서가 보장되고 해당 Queue 안 작업이 끝나야 Queue 밖의 작업 실행 (일반함수)
  - 직렬 · 비동기 : Queue 작업의 순서 보장, Queue 밖의 작업은 순서 X

    0 < 10 0 < 1000
    DispatchQueue.sync → temp=11 or 1001
    DispatchQueue.async → temp=1

(동시)
- Concurrent : Queue 에 추가된 작업들이 각 순서에 맞게 작업이 시작, 앞 순서의 작업이 실행 종료여부와 상관없이 새 작업이 시작되고 동시에 여러 작업이 실행중인 상태.

  DispatchQueue.global().async → 1
  DispatchQueue.global().sync → 991

  다른 연산이 끝나기 전에 다음 연산이 시작되어 누적이 되지 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 연산1 / 동시간을 결정하는 것이 아니라 task가 어떤식으로 핸들링하는지가 관건.

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 기다렸다가 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
High
- .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
low
- .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 func1 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.

reference type

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.

```swift
class Person {
    let name: String
    init(name: String) {
        self.name = name
        print("\(name) is being initialized")
    }
    deinit {
        print("\(name) is being deinitialized")
    }
}
```

```swift
var reference1: Person?
var reference2: Person?
var reference3: Person?
```

optional
→ automatically initialized with a value nil
so don't reference a Person instance

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

```swift
reference1 = Person(name: "John Appleseed")
// 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:

```swift
reference2 = reference1
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:

```swift
reference1 = nil
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:

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

Strong Reference Cycles Between Class Instances

In the examples above, ARC is able to track the number of references to the new Person instance you create and to deallocate that Person instance when it's no longer needed.

However, it's possible to write code in which an instance of a class never gets to a point where it has zero strong references. This can happen if two class instances hold a strong reference to each other, such that each instance keeps the other alive. This is known as a strong reference cycle.

```swift
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { print("\(name) is being deinitialized") }
}

class Apartment {
    let unit: String
    init(unit: String) { self.unit = unit }
    var tenant: Person?
    deinit { print("Apartment \(unit) is being deinitialized") }
}
```

```swift
var john: Person?
var unit4A: Apartment?
```

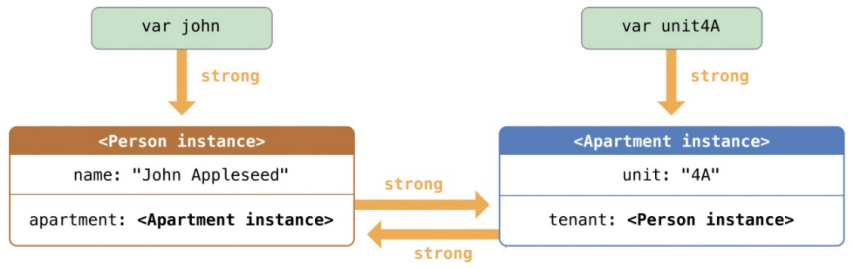
You can now create a specific `Person` instance and `Apartment` instance and assign these new instances to the `john` and `unit4A` variables:

```swift
john = Person(name: "John Appleseed")
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:

```swift
john!.apartment = unit4A
unit4A!.tenant = john
```

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

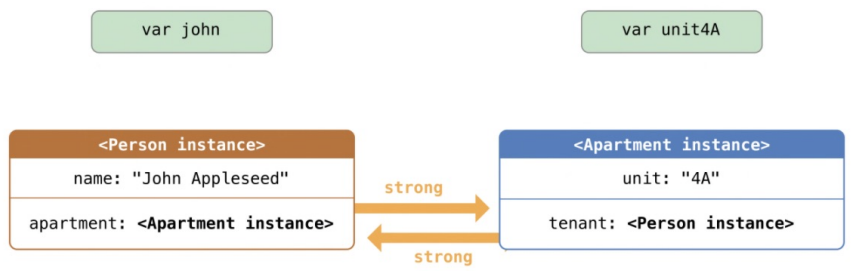


Unfortunately, linking these two instances creates a strong reference cycle between them. The `Person` instance now has a strong reference to the `Apartment` instance, and the `Apartment` instance has a strong reference to the `Person` instance. Therefore, when you break the strong references held by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

```swift
john = nil
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.

Resolving Strong Reference Cycles Between Class Instances

#2

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.

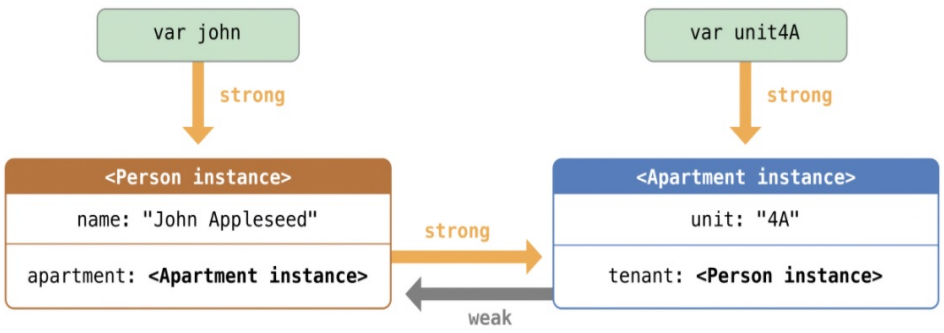
Because a weak reference doesn't keep a strong hold on the instance it refers to, it's possible for that instance to be deallocated while the weak reference is still referring to it. Therefore, ARC automatically sets a weak reference to nil when the instance that it refers to is deallocated. And, because 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.

```swift
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { print("\(name) is being deinitialized") }
}

class Apartment {
    let unit: String
    init(unit: String) { self.unit = unit }
    weak var tenant: Person?
    deinit { print("Apartment \(unit) is being deinitialized") }
}
```

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

```swift
var john: Person?
var unit4A: Apartment?

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

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




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


```swift
unit4A = nil
// 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.

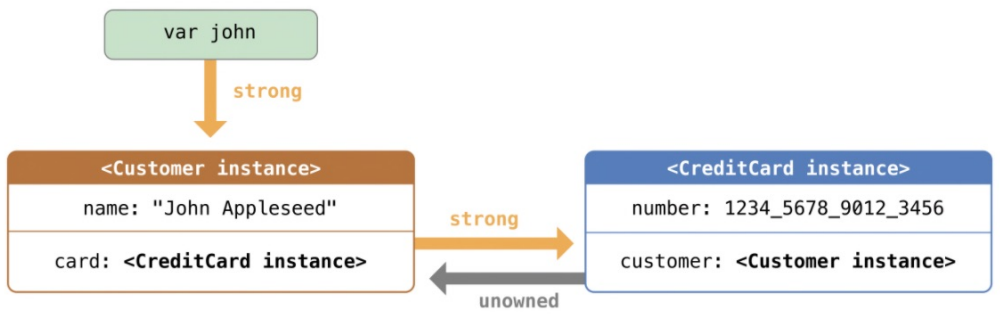
Furthermore, a new CreditCard instance can only be created by passing a number value and a customer instance to a custom CreditCard initializer. This ensures that a CreditCard instance always has a customer instance associated with it when the CreditCard instance is created.

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

```swift
class Customer {
    let name: String
    var card: CreditCard?
    init(name: String) {
        self.name = name
    }
    deinit { print("\(name) is being deinitialized") }
}

class CreditCard {
    let number: UInt64
    unowned let customer: Customer
    init(number: UInt64, customer: Customer) {
        self.number = number
        self.customer = customer
    }
    deinit { print("Card #\(number) is being deinitialized") }
}
```

not optional nor var

```swift
var john: Customer?
john = Customer(name: "John Appleseed")
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:

```swift
john = nil
// Prints "John Appleseed is being deinitialized"
// Prints "Card #1234567890123456 is being deinitialized"
```

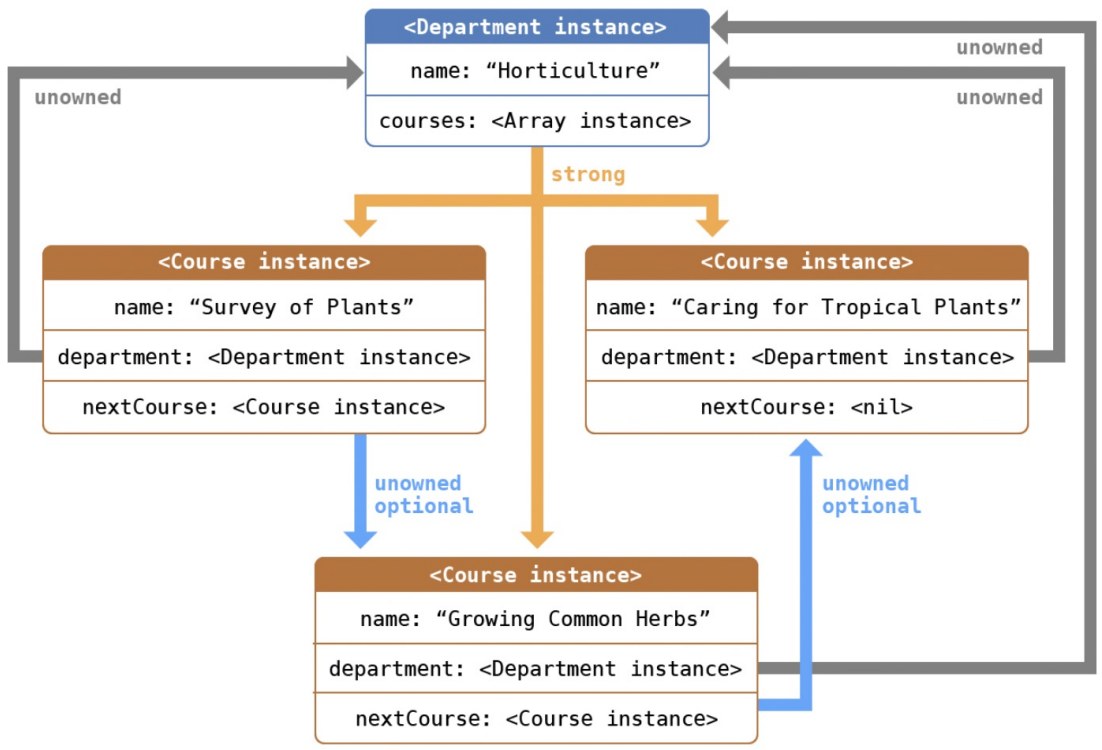
#3

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.

```
class Department {
    var name: String
    var courses: [Course]
    init(name: String) {
        self.name = name
        self.courses = []
    }
}

class Course {
    var name: String
    unowned var department: Department
    unowned var nextCourse: Course?
    init(name: String, in department: Department) {
        self.name = name
        self.department = department
        self.nextCourse = nil
    }
}
```

```
let department = Department(name: "Horticulture")

let intro = Course(name: "Survey of Plants", in: department)
let intermediate = Course(name: "Growing Common Herbs", in: department)
let advanced = Course(name: "Caring for Tropical Plants", in: department)

intro.nextCourse = intermediate
intermediate.nextCourse = advanced
department.courses = [intro, intermediate, advanced]
```



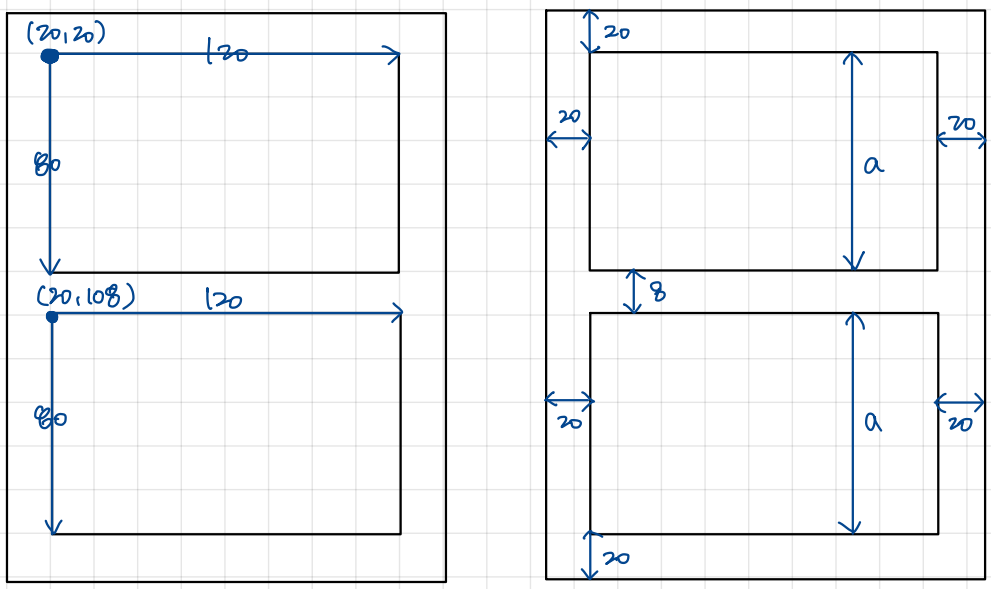
AutoLayout

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
   - The app supports internationalization — depends on language / region. (길이, 포맷 레이아웃)
   - 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

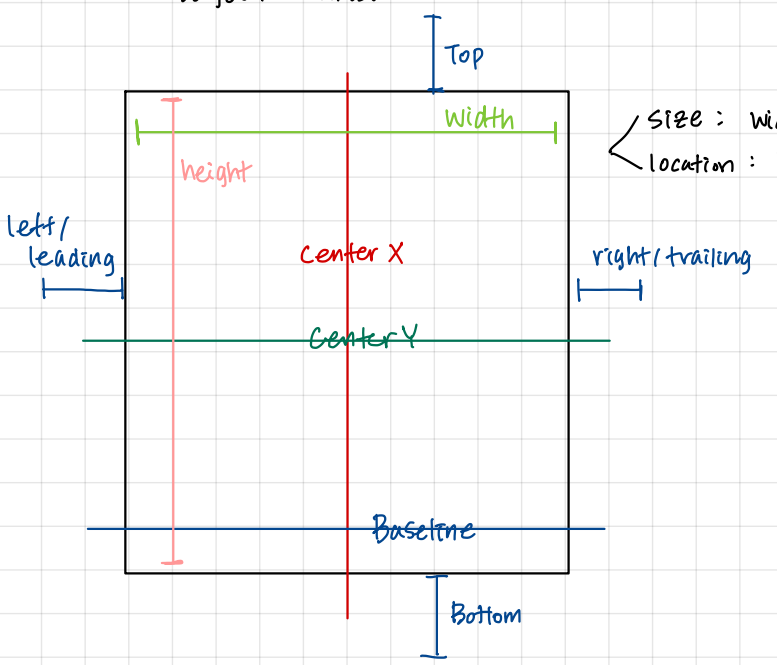
- axis (UIStackView only) : stackView's orientation { vertical / horizontal }
- orientation (NSstackView only) : stackview's orientation { vertical / horizontal }
- distribution : the layout of the views along the axis
- alignment : the layout of the views perpendicular to the stackview's axis
- spacing : spacing between adjacent views.

#1

3 Anatomy of a constraints

RedView.Leading = 1.0 × BlueView.trailing + 8.0
(Item1 / Attribute1 / relationship / Multiplier / Item2 / Attribute2 / Constants)

1. Auto Layout Attributes



size : width, height
location : leading, left, top

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, Width] — size — can be assigned constant values or combined with other height and width attr. (nonnegative)

[Top, Bottom, Baseline] — value increases as you move down the screen — attr can be combined only with centerY, Top, Bottom, Baseline attr.

[leading, Trailing] — value increases as you move towards the trailing edge. For left-right layout directions, the values increase as you move to the right. For right-left layout direction, the values increase as you move to left — can be combined only w/ leading, trailing, centerX attr.

[left, right] — the values increase as you move to the right — can be combined only w/ l, r, centerX (l, r은 쓰는 것보다 leading/trailing 사용하기. due to adaption to view's reading direc. (language setting))

[CenterX, CenterY] — Based on the other attr in the equation — CenterX can be combined w/ CenterX, leading, trailing, r, l. CenterY can be combined w/ centerY, top, bottom, Baseline.

Closure

* retain cycle 에 생기는거 + how to resolve → combine 에서는?

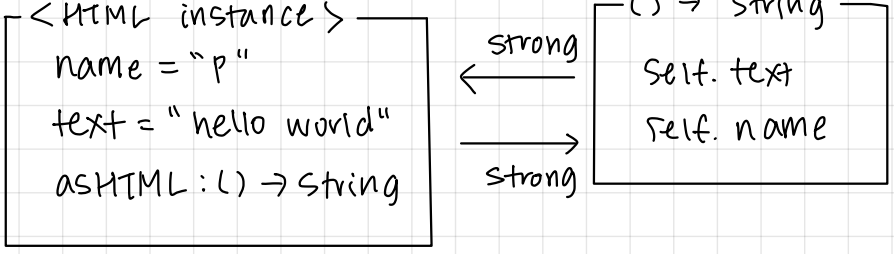
Syntax :

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

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

Closure type : (String) → ()

expression : { name in ...



만약 paragraph variable이 nil이 된다고 하더라도 HTMLElement, closure은 같이 deallocate 되지 X

@escaping vs. @nonescaping

① @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 @escaping : when passing a closure as the func arg, the closure is being preserve 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: @escaping ((Int) → Void)) {
    var sum : Int = 0
    for val in array { sum += val }
    self.completionHandler = handler
}
func doSometing() {
    self.getSumOf(array: [...]) {[weak self] (sum) in
        print(sum)
    }
}
```

Asynchronous Execution example

```
func getSumOf(array: [Int],
              handler: @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)
    }
}
```

@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 밖으로 나가서 반환 되어도 불려질 수 있도록 저장 가능 (비동기 처리 유용)

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

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

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 아까 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 @escaping을 통해 completion을 탈출시켜 외부에 저장해두고 원할 때 불러와 값을 변경할 수 있습니다.

만약 noescape함수에 클로져를 외부에 저장하려고 하면 아래와 같이 에러가 납니다.

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

    completionHandlers.append(closure)   // Converting non-escaping parameter 'closure' to generic parameter 'Eleme
}
```

file private

*Something

*privacy.

TableView ← dataSource
delegate

custom cell – deque the cell
: tableview.dequeReusableCell
(withIdentifier:) as! YourCell

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 detailedViewController = segue.
                                 destinationViewController
                                 as! DetailViewController
detailViewController.index = index
```

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

class SwipeTableController : UITableController, SwipeTableViewCellDelegate

CategoryViewController : SwipeTableViewController

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

★ viewDidload viewWillAppear 부분

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 : 몇개의 list member를 뺄것인가?
Categories? count ?? 1

CellForRowAt → UITableViewCell : Insert in the certain row

```
let cell = super.tableview(tableView, cellForRowAt : indexPath)
if let category = categories?[indexPath.row]{
```

Cell 재활용하는 방법

Tableview Delegate Method :
- didSelectRowAt ⇒ performSegue(withIdentifier : "goToItems", sender : self)
  스토리보드에 Configured 안되었을 경우
- Prepare(for segue : UIStoryBoardSegue, sender : Any?)

```
let destinationVC = segue.destination as! TodoListController

if let indexPath = tableView.indexPathForSelectedRow {
    destinationVC.selectedCategories = categories?[indexPath.row]
}
```

★ 왜 이거 쓰는지 알아야!

TodoListViewController : SwipeTableViewController

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

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

SwipeTableViewController : UITableViewController, SwipeTableViewCellDelegate {

```
@override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell{
    let cell = tableview.dequeueReusableCell(withIdentifier : "cell", for: indexPath) as! SwipeTableCell
    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

#2

AppDelegate : applicationWillTerminate self.saveContext()

AppDelegate 파일에서 data를 저장하는 cycle이 필요 (applicationWillTerminate)

```
lazy var persistentContainer : NSPersistentContainer = {
    let container = NSPersistentContainer ( name : "DataModel")

func saveContext () {
    let context = persistentContainer. viewContext
    if context. hasChanges {
        do {
            try context.save()  ...
```

CategoryViewController : UITableViewController

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

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

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

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

    func loadItems ( with request : NSFetchRequest < 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")
        } tableview. reloadData()
```

```
extension TodoListViewController : UISearchBarDelegate {
    func searchBarSearchButtonClicked( _ searchBar : UISearchBar){
        let request : NSFetchRequest < Items > = Items. fetchRequest()
        let predicate = NSPredicate ( format : " item CONTAINS %@ , searchBar. text! )
        request. predicate = predicate
        let sortDes = NSSortDescriptor ( key : " item " , ascending : true )
        request. sortDescriptor = [SortDes]
        loadItems ( with : request)
    }
    func searchBar ( _ searchBar : UISearchBar, ' textDidChange searchText : String) {
        if searchBar. text?. count == 0 {
            loadItems()

            DispatchQueue. main. async {
                searchBar. resignFirstResponder()
            }
        }
    }
```

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 iOS10 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 / data type

ex)
```
let items = List<Item>()
var parentCategory = LinkingObjects(fromType: Category.self,
                                    properties: "items")
```

#3 Creating and saving objects

Coredata : persistentContainer

```
let appDelegate = UIApplication.shared.delegate as! AppD
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

fetching data

CoreData :

```
let request: NSFetchRequest<Category> = Category.fetchRequest()
do{
    categories = try context.fetch(request)
```

OR

```
let cateP = NSPredicate(format: "parentCategory.name MATCHES %@",
                        selectedCategories!.name!)
```

Realm :

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

Pros & Cons

#3

#1 Database Usage

Realm이 더 쉬움. - 변경에 대해서 바로바로 save가 되어 데이터 손실 위험 ↓

CoreData의 경우 API에 대해서 더 깊게 알아야함

- 변경이 있을 때마다 계속 save를 해주어야 함

#2 Creation of the scheme

CoreData의 경우 XCode가 클래스를 만듦 - visual representation

one-to-many, many-to-many에 유리.

Realm의 경우 ① objc dynamic var

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

#3 The app volume

CoreData의 경우 앱의 크기가 커지지 않지만

Realm의 경우 13MB 정도 더 커지게 됨.

#4 Cross-Platform Support

Realm은 iOS 뿐만 아니라 Java, Swift, objective-C 도 있어서 logic이 뱅하지 않음 (일치성)

#5 Work Speed

Realm은 자체의 engine을 사용하기 때문에 → 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
  - access members of a collection
    - a variable or constant, containing collection
    - 2 square bracket
    - a key or index inside the bracket.
- can define initializers to set up their initial state init()
  - default initializer
  - memberwise initializer - only available in Struct
  - Failable initializer
    - ↳ returns nil / value optional 쓸 때 사용.

```
init?(value: ___) {
    guard condition else {
        return nil
    }
}
```

  - Designated Initializer

```
struct Rectangle {
    var width = 0
    var height = 0
}
let square
    = Rectangle(width = 10,
                height = 10)
```

default값 0/x

init()을 사용하지 않아도 저절로 initialize 가능

```
class Circle {
    var radius : Double
    var circumference : Double {
        2* .pi * radius
    }
    init(radius : Double) {
        self.radius = radius
    }
}
```

- 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 deinitialized, 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.
  → deadlock
- mostly structs scenario: when properties of a struct are mostly value type
- Inheritance X

Struct는 stack의 영역에 메모리를 저장해서 속도가 빠르고 Struct를 복사하면 새로운 Struct가 생기는데

class는 Heap영역에 메모리를 저장해 속도가 느리고, 복사를 하더라도 reference 복사 → 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,

When to use Classes

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

DEF: 자동 레퍼런스 카운팅으로서 자동으로 메모리를 관리해주는 방식을 말한다. 참조 카운팅이 0이 될 때만 메모리에서 해제한다.

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

시점: 컴파일 시점에 동작.

2. Weak, Strong, Unowned

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

- Weak: 객체 소유하지 않고 주소 값만을 가지고 있는 포인터 개념. 참조하지만 weak 메모리를 해제시킬수 있는 권한은 다른 클래스에. 값 지정시 리테인이 발생 X → release X. 언제 메모리가 해제될지 알수 X. 해제 될 경우 자동으로 reference가 nil로 초기화를 해준다. MUST BE OPTIONAL
  - retain cycle에 의해 메모리가 누수되는 문제를 막기위해 사용, Ios Framework에서 대표적으로 delegate patterns

- Unowned: nil이 될수 X, MUST NOT OPIONAL, 해제된 메모리 영역을 참조하지 않는다는 확실한 경우에만 사용.
  - non-optional type으로 선언, 객체가 ARC에 의해 메모리가 해제 되더라도, 해당 객체 값을 존재하는 것으로 인지하며, 해당 객체에 액세스 할 경우 런타임 오류. lifecycle 명확하고 개발자에 의해 제어 기능이 명확한 경우, weak optional type 대신 사용해 간결하게.

3. 클로저 블럭내에 [weak self] in 의 역할과 이유는?

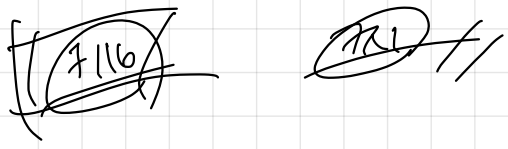
- [weak self]의 역할: ARC가 property 갯수를 counting하지 않도록 하여 순환참조가 일어나지 않도록 만드는 역할. 해제될때 property value = nil로 만들어 순환 참조 X.

4. ☆ Escaping Closure Closure 쓰는 방법 completion block 등등

- method parameter로 전달받은 closure를 method의 life cycle 내에서 실행하여 끝내지 않고, method scope의 외부에 전달할 때는 해당 closure를 escape. 해당 환경이 끝난 이후에도 closure는 메모리 어딘가 저장되어야 하며, closure안에 사용된 outer object (self)에 weak와 같은 reference type을 사용해야 할수 있음을 주의.

5. Casting

- as: type 변환의 성공 보장. compile time에 가능/불가능 여부 알수 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.

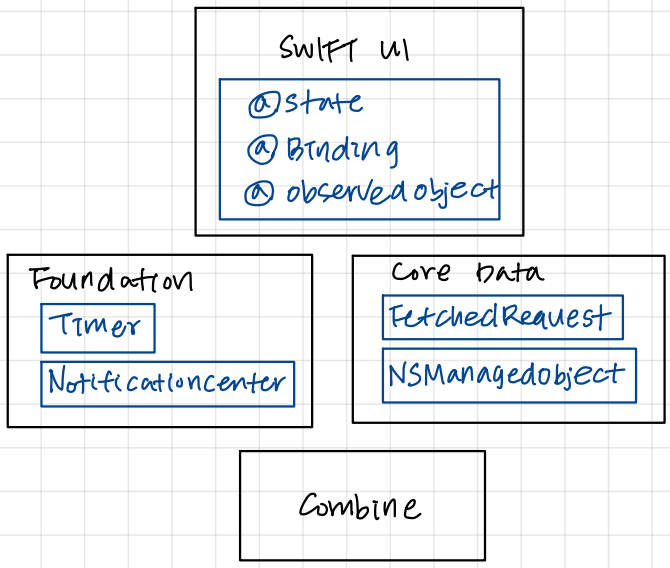
1. NotificationCenter: executes a piece of code any time an event of interest happens, such as when the user changes the orientation of the device or when the software keyboard shows or hide on the screen.
State에 따라서 code를 실행할 수 있도록 하는 장치

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 turn a multitude of tasks concurrently in different queues with different priorities.

   Concurrency = asynchronization.

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. TableView)
  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)
  하지만 Box layout system (web)을 이용하여 UI 보여줌.

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.

SWIFT UI

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

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 UI view에서 적용하길 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 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 & NS operation Queues (FIFO) : manages threads for us.

thread : think it as highway

main thread	background thread
- speedy - UI	- loaded tasks (timely)

Serial : 4 3 2 1 → 1번이 끝나야 2번 시작, 2번 끝나야 3번 시작.

- predictible executable order
- prevents race conditions

MAIN QUEUE

Concurrent : 1 2 3 4 → 1번부터 시작해 2번, 3번, 4번 순서대로 진행

하지만 task가 누가 빨리 끝날지는 "unpredictible"

- faster
- unpredictable order

BACKGROUND QUEUE

ex) Background에서 API를 가지고 와 tableview에 (reload) populate 하는 경우 →

```
DispatchQueue.main.async {
    self.tableView.reloadData()
}
```

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

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

async : 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.

메소드 (method)

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

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

instance 변수인 self.name, self.email 사용

@classmethod #클래스 메소드

```
def number_of_users(cls):
    print("총 유저수는: {}".format(cls.count))
```

class변수인 cls.count 사용

@staticmethod #정적메소드 → 아무런 변수 사용 X

```
def is_valid_email(email_address):
    return "@" in email_address
```

어떤 속성을 다루지 않고, 단지 기능(행동)적인 역할만하는 메소드를 정의할 때.

절차지향프로그래밍과 객체 지향 프로그래밍의 차이

절차 - ① 필요한 데이터를 관련 있는 함수와 묶어서 관리하기 힘들다. + ② 명령어들을 순서대로 실행하는 것.

객체 - 서로 관련있는 데이터와 함수를 객체로 묶어서 사용가능. + 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.
- 캡슐화(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

Swift UI

1. Identifiable (protocol): 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 : @ observed object 에 이어 Swift UI 2.0 에서 추가 됨.

ⓐobservedobject의 경우 view가 reloading / refresh가 될 경우에 data persistence X → reloading.
그래서 viewModel이 data를 holding 하고 있을 경우에는 StateObject 사용.

-첫번째로 data instance를 initialize 할 경우 StateObject를 사용 (CREATION)
2nd view로 passing을 하면서 subview에서 사용할 경우 Obsenved object 사용. (PASSING TO 2ND VIEW)

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

기타 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",
      :
}
```