**# About ME (자소서 기반 +@)**

**# 과제 (기술면접 포함)**

1. **알고리즘 문제 – alter-date-format**
2. **설계 문제 – is-available-time**
3. **프로젝트 – shop-list-filter**

**# 하고싶은 말**

**# 과제**

**– 1. 알고리즘 문제 – alter-date-format**

**-. Make를 왜 썼는지? 이게 뭔지 설명. 왜 플레이그라운드나 프로젝트를 안쓰고 터미널로 했는지?**

-. 테스트가 빠르고 간편하다.

-. 매번 컴파일 해줘야 하는 불편함이 있었지만

-. make라는 파일 관리 유틸리티로 Makefile에 명령어를 써놓고 단축키워드로 작동시켜서

시간절약 + 단순반복 시간 최소화 할 수 있게 됐다.

**-. Type casting이란?**

**-. 옵셔널이란? 옵셔널 바인딩이란?**

**-. Guard 문을 쓰는 이유? If let과의 차이?**

-. 이후에도 계속 가드문으로 선언해준 변수 명을 사용할 수 있어 편리해 선호한다.

**-. 아쉬운점 / 보완할 점**

**– 2. 설계 문제 – is-available-time**

**-. 왜 이렇게 설계했는지?**

1. 매개변수인 시간패턴의 자료형을 뭘로 받을지부터 생각

-. 시간 패턴이 주어진 날짜와 비교하여 가능한지 불가능한지 2가지로 체크를 해야하기 때문에 1, 0으로 가능/불가능을 나눌 수 있는 비트 자료형을 생각했다.

2. 맨 처음은 1로 고정하여 비트 개수를 고정하고, 그 다음의 7개의 자리수는 요일을, 뒤의 24개의 자리수는 시간으로 표현하면 UInt32 내에서 해결가능해서 UInt32 자료형을 선택했다.

3. 이용여부를 판단하고 싶은 비교 시각을 입력하지 않을 경우, 현재 시각 기준으로 비교하라고 요구사항에 적혀 있어 시간은 옵셔널 Date로 타입을 설정했다.

4. 예시에서 다양한 시간패턴을 입력할 수 있었기 때문에 이후 매개변수를 UInt32의 배열로 변경

5. UInt32를 문자열 형태로 변환해 Date에서 요일과 날짜를 뽑아 해당 인덱스가 1인지 체크했다.

-. 비트시프트가 아닌 문자열 형태로 변환한 이유는 비트시프트할 경우, 자리수를 지키기 위해 만든 맨 앞의 1이 사라지며 자리수가 변경되어 제대로 확인이 불가능.

또 가능하다고 하더라도 인덱스 계산을 다시 해야돼서 덜 직관적이라고 판단.

문자열의 경우, 원래 시간 패턴의 변화 없이 + 요일이나 시간의 인덱스 숫자 체크하면 돼서 직관적.

6. 이후 테스트 코드 작성. 테스트코드의 매개변수는 이용자가 편하게 입력할 수 있도록 문자열로 받을 수 있게 처리.

>. 다른 해결방법은 이런게 있었는데 각

**-. 어느 상황에서 필요할까요?**

-. 쇼핑몰과 연락해야 하는 상황일 때. 지그재그는 쇼핑몰 플랫폼이기 때문에 실사용 유저와 쇼핑몰 모두 관리해야 하는데, CS 관련 사항을(배송문제 등)을 쇼핑몰에 전달하거나, 개별 쇼핑몰에 연락해야 하는 상황일 때 필요할 듯.

-. But, 지그재그는 플랫폼이고 고객 관리(배송 등)는 각 쇼핑몰이 알아서 하고, 쇼핑몰에서 연락이 되지 않을 경우 최후의 수단으로 지그재그에서 처리하는 거로 지그재그 앱 내 QA에서 봤다. 따라서 CS로 그렇게 많이 연락을 할 것 같지는 않다.

-. 쇼핑몰에 전달하는 내용은 주로 메일을 통하거나 다른 관리자용 버전 웹으로 공지를 띄우거나 하지 않을까 생각.

**-. 아쉬운점 / 보완할 점**

-. 더 좋은 방법은 없을까 계속 고민하다가 시간을 많이 써서 테스트 코드를 작성하는데 시간을 많이 못썼다. 때문에 Input 받는 문자열의 형태나 헬퍼 메서드를 깔끔하게 정리하지 못했는데 그 부분이 아쉽다.

-. 설계관점에서 아쉬운 / 보완점?

-. 아쉬운 점은 아니지만 느낀점을 얘기하자면, 원래 처음에는 2차원 배열을 생각했었다. 시간와 요일을 각각 행, 열로 만들어서 주어진 시간과 비교할까 생각했는데 이렇게 되면 시간복잡도가 O(n^2)으로 너무 오래 걸릴 것 같아서 다시 생각했다.

-. 이렇게 생각하고, 개선할 수 있는 문제를 줘서 감사하다. (마지막 하고싶은말과 중복 ㅠㅜ)

**– 3. 프로젝트 – shop-list-filter**

-. 과제수행서에 `디테일한 부분까지 신경쓰는 제 장점을 가장 잘 보여줄 수 있는 과제라고 생각했고, 필터 기능을 구현해보고 싶어 선택하게 되었습니다.` 이렇게 적었는데,

**-. 지그재그에서는 그 장점이 어디에서 드러났는지?**

**-. 이 장점을 드러내는 다른 사례는 뭐가 있는지?**

**-. 필터 기능은 왜 구현해보고 싶었는지?**

**-. MVC 패턴에 대해 설명**

-. Model-View-ViewController. 뷰 컨트롤러가 가운데에서 모델과 뷰와 소통하며 모델의 데이터를 뷰에 전달하고(Notification 사용), 뷰의 변화를 감지(Delegate)하여 모델을 변화시킨다.

-. 이때, 모델과 뷰는 서로에게 독립적이어서 뷰 컨트롤러가 모든 걸 컨트롤 해야 한다.

-. But, iOS의 MVC는 컨트롤러에 뷰의 라이프사이클이 존재하기 때문에 뷰와 뷰컨트롤러는 떨어질 수 없는 관계. 그래서 뷰컨트롤러가 너무 비대해지는 문제가 생긴다. 프로젝트가 커질수록 관리가 어렵다.

**-. MVVM 패턴에 대해 설명**

**-. MVVM 패턴을 써본적 있는지?**

-. “지그재그에 있는걸 보고” 최근에 간단한 예제 프로젝트를 만들어 봤다.

**-. AutoLayout 개념 설명**

-. Auto Layout이란 기존의 Frame-Based Layout과 다른 View들 간의 관계를 이용하여 View의 위치와 크기를 자동으로 결정하는 Layout System입니다.

-. 제약조건을 이용해서 뷰의 위치를 지정하는 것. 즉, 두 뷰 사이의 관계를 제약 조건이라는 것을 이용해서 뷰의 크기와 위치를 지정하는 것. 오토레이아웃 덕분에 여러 해상도의 지원이 가능해짐.

-. Item1.atrribute = 비율(multiplier) \* item2.attribute + 간격(constant)

**-. AutoLayout을 코드로 작성할 수 있는지?**

-. 할 수 있슴당. 팀프로젝트 할 때, 루트뷰인 탭바컨트롤러에 뷰를 배치해야해서 코드로 작성했었다. 당시에는 스냅킷을 이용하지 않았지만, 이번 개인프로젝트 Week-it을 하면서 간단하게나마 스냅킷도 써봤다.

**-. FrameBase의 장, 단점**

-. 장점: 코드의 복붙이 가능해서 생성, 수정이 편하다.

-. 단점: 하나의 뷰가 변하면 그와 연관된 뷰들도 다시 코드를 재작성해줘야 한다.

**-. AutoLayout 코드와 스토리보드 중 뭘 더 선호하는지? 각각의 장단점은?**

-. 스토리보드

-. 장점: 시각적으로 볼 수 있고, 만드는게 편하다.

-. 단점: 애니메이션 등을 적용해야 할 때나 제약조건을 변경해야 할 때, 스토리보드를 이용했다면 조건들을 하나하나 IBOutlet으로 끌어와서 결국은 코드로 변경해줘야 하기 때문에 매우 불편하다.

-> 그런 경우에는 코드로 작성하는게 훨씬 나은 것 같다. 결국 어떤 상황이냐에 따라 다른것 같다.

**-. 킹피셔 말고 기본 프레임워크로 이미지를 가져오고 캐싱할 수 있는지? (URLSession)**

-. 할 수 있다. 사실 URLSession을 쓸지 킹피셔를 쓸지 고민 하다가 좋은 라이브러리를 찾아서 사용하는 것도 개발자의 능력 중 하나라고 생각해서 이용했다. “수레 바퀴를 두번 만들지 말라.” 캐싱이 쉬운 라이브러리가 있는데 굳이 복잡하게 갈 이유가 없슴메!

**-. DataResponse / DataService(JSON ferch)를 만든 이유?**

-. 실제 구현시에는 서버에서 정보를 받아올텐데 최대한 실제 개발때와 같은 구조를 만들고 싶었다.

-. DataResponse에서 제네릭을 이용했는데, 제네릭이 뭔지

-. JSON Decoder를 이용했는데, 이를 이용하지 않고도 JSON을 파싱할 수 있는지

-. JSON Serialization을 이용해서 Foundation 객체로 변환하고, Dictionary의 키값을 불러와서 기존에 만들어 둔 인스턴스의 프로퍼티에 대입

**-. Codable 이란?**

-. Decodable & Encodable을 준수하는 프로토콜.

-. 코더블을 채택하면 Class, Struct, Enum 타입을 JSON으로 serialize/deserialize 할 수 있음

-. 디코딩 과정 : JSON -> Data 변환 -> 인스턴스로 Decode

**-. 테이블뷰 안에 콜렉션 뷰를 넣는 UI 구조인데 델리게이트를 썼네요. 델리게이트가 뭔가요?**

**-. 스토리보드와 xib, frame base 중에 뭘 선호하는지?**

-. 시각적으로 화면 구성을 볼 수 있는 스토리보드와 xib를 선호한다. 스토리보드는 앱 flow를 볼 수 있어서 좋고, xib는 조각 단위 화면, Ex. 테이블뷰, 콜렉션뷰 셀을 만들 때 재사용하기 좋다.

**-. Xib파일을 썼던데 왜 썼는지? 어떨때 쓰는지? 스토리보드와의 차이점은?**

-. 명확히 하고자

-. & 내가 할 수 있는 영역들을 최대한 보여주고자했다.

**-. Realm을 사용한 이유?**

**-. Realm 어디까지 해봤니? 마이그레이션 등등.**

**-. FilterView – 테이블뷰 안에 컬렉션 뷰를 넣는 UI에서 컬렉션뷰의 delegate를 Custom Protocol로 구현한 이유?**

-. 테이블뷰 셀 안에 컬렉션뷰의 Delegate & DataSource를 지정할 수도 있었겠지만, 뷰가 모델에 직접 액세서 할 수 없어야 하는 MVC를 위반. 사실 이전에는 이렇게 구현했었는데 뷰컨트롤러에 대한 정보를 알기 어려워서 또 다른 문제가 생기더라.

-. 따라서 컨트롤러가 뷰를 관리하도록 델리게이트를 구현

**-. Delegate에 대해 설명**

**# About ME**

**- Eat it**

-. Crash Report 본 적 있냐. 왜 크러쉬 났고, 어떻게 해결했는지

-. 본적 있음. 메인 뷰에 해당 주의 포스트들을 뿌려주는데 Realm에서 데이터를 불러와서 15개의 빈 자리가 있는 배열 중 알맞은 자리에 데이터를 집어 넣는 함수에서 에러가 났다. 계산식을 월이 변경되는 걸 고려하지 않고 짜서 8월에 뭔가를 입력하면 앱이 아예 죽었다.

-. 업데이트를 위해 테스트 하다가 문제를 발견했고, 피드백 메일이 와서 확인 & 크래쉬 리포트를 보며 몇명이나 문제를 겪고있는지 확인.

-. 다행히 7월의 마지막날에 수정했다.

-. 해결 방법: 매달의 마지막 날짜 가져와서 달이 바뀐 경우와 아닌 경우의 계산식을 다르게 짰다.

**- 기타 질문**

-. 왜 나를 뽑아야 하는지 어필해 보세요. 자신만의 강점이 뭔가요?

-. 나는 행동하는 사람. 주어진 것만 해내는게 아니라 계속에서 더 나은 것을 갈망하고, 찾고 발전시키는 사람.

-. 충분히 생각하고 신중하게 조심히 행동하는 사람도 중요하다. 하지만 스타트업이라는 환경에서는 빠르게 만들고 빠르게 피드백을 받아 더 나은 것을 만들어내는 것이 더욱 중요하다는 것을 깨달았다. 어차피 세상에 없던 것을 만들기 때문에 실제 유저들 말고는 피드백을 줄 사람이 없기 때문이다. 생각은 모두 추측일 뿐이다.

-. 나는 기획자다. 개발도 곧 기획이라고 생각한다. 큰 구조를 설계하고, 일정 안에 완성될 수 있도록 스케쥴을 짜고.

-. 지그재그 유저로서 바라는 점이 있다면?

-. 지그재그를 얼마나 자주 이용하고, 얼마나 자주 구매하는지?

-. 좋아하는 앱이 있는지? 이유는 뭔지?

-. 아이디어스

: 핸드메이드 마켓 플랫폼. 작가 후원하기를 통해 정말 작가를 위하고자 하는 + 평소 좋아하는 작가에게 조금이라도 금액을 지불하려고 하는 구매자들의 심리를 잘 파악했다고 생각.

: 덕분에 작가들이 스스로 앱을 홍보하는 선순환이 일어나며 매우 적절한 타겟에서 광고가 되는 효과.

: 그리고 쌈지길에 오프라인 매장을 열며 온오프를 활용해 핸드메이드 시장을 넓히는데 크게 일조.

: 단지 앱 서비스 하나가 아니라 플랫폼이기에 가능한 순기능이 아닐까 라는 생각이 들었다.

**-. TDD 방법론 (Test-Driven-Development, 테스트 주도 개발)**

-. 테스트 주도적인 개발로 소스코드 작업 전에 테스트 코드를 먼저 작성해 소스 수정에 대한 부담을 덜고 디버깅 시간을 줄일 수 있음

-. 장점: 소스코드의 품질이 높다. 재설계 및 디버깅 시간 절감

-. 단점: 단기적 코드일 경우 생산성 떨어짐. 실제 코드보다 테스트 케이스가 더 커질 수 있음

-. <https://minieetea.com/2014/01/archives/1327>

-. <https://academy.realm.io/kr/posts/ios-tdd-test-driven-development/>

-. TDD는 패러다임이고, 그 방법들 중에 UI Test, Unit Test 등등이 있음.

**-. CI, 젠킨스**

**# 하고싶은 말**

**\*\* 질문**

-. What is Next?

**\*\*마무리**

-. 아직 부족한게 참 많지만 그래도 이번 과제를 통해 스스로 많이 성장했다고 생각합니다. 어떻게 하면 더 효율적인, 좋은 코드, 구조를 짤 수 있을까에 대한 고민을 많이 하게 됐습니다.

사실 개발자로 구직활동을 한 건 지그재그가 처음인데, 처음부터 이렇게 좋은 곳에서 기회를 얻게 돼 눈이 많이 높아질 것 같네요 헤헤

시간 내주셔서 감사했습니다. 꼭 또 뵐 수 있었으면 좋겠습니다.

-. 코드 총 몇줄인지?

ㅇ week-it 2700줄

**줄**

-. 유연하지 못함. (시간 – 30분단위, 분, 초 단위 불가) & 같은 이유로 추가적인 걸 붙이기 어려움

-. 사람이 쓰기 어려움. 컨버팅이 필요함

-. Int를 쓰면 된다.

-. 튜플로 처리.

-. 왜 이렇게 했어요?

-. 클라이언트에서는 비트를 안써요. -> 이런걸 쓸 수 있다는 걸 보여주고 싶었다.

**ㅁ RxSwift (통신 방법입니다. Entity 사이에서 통신하는 방법)**

-. Reactive X 패러다임 자체를 MS가 제시함. (MVVM도 MS가 제시)

-. 비동기가 반응형이라는 거야?

ㄴ 놉! 비동기에 반응형을 더하자!

ㄴ 콜백을 받아서 진행해야 되는거 아니야??? 라는 생각. 끝을 알아야 한다는 생각을 버려! 왜 알아야해????

ㄴ 비동기 처리를 값만 바뀌어!라고 정해놓으면, 값만 바뀌었을 때 이걸 처리하라고만 정해놓으면, 알아서 반응해서 메서드를 실행함.

**\*\* 비동기가 핫해진 이유.**

-. 네트워크가 보편화돼서. 예전에는 네트워크가 없어서 동기로 처리해도 문제가 되지 않았음.

-. 연산이 오래걸리는 작업을 할 수가 없었기 때문에 동기로 다 오케이였다.

-. 현재는 지금은 그래픽은 그래픽대로, 연산은 연산대로 진행해야 한다.

>> 비동기로 처리해야 될 일이 잦아지면서 비동기가 핫해졌다.

**ㅁ 구현에 대한 이야기**

-. RxSwift는 신호를 줄때, 뭐에 담아서 준다고. 이벤트에 담아서 준다고?

-. 무슨 이벤트가 났는데, 값이 이걸로 바뀌었어.

-. 옵저버블(Type)이 신호기. / 릴레이 – 신호를 받으면서 동기에 – 서브젝트(신호를 받고 쏘는 애)

-. Next, error, complete

-. 기존의 쓰레드 이용이 마음에 안들었던 애플.

-. 프로그램은 쓰레드에 대해서는 알 필요가 없다. 프로그램은 운영체제에 이거 어싱크 작업이야. 라고 전해주기만 하면 운영체제가 해당 쓰레드에서 돌려준다.

-. 순발력은 2순위고, 정답을 맞추는게 1순위. 답을 해라

-. 처음 낸게 베스트 정답은 아닐 수 있다.

\*\* 특정 쇼핑몰을 노출시키거나 할 때!!!!!!!

-. 쇼핑몰들의 한정 특가.

-. 광고 노출시간 이런거.

\*\* 별표별표!!!!

-. 7개월만에 진행한거다!! 라는 게 가장 중요하다!!!!

-. Hash Table의 구성요소

-. Key에는 Hash 함수가 있고, Value에는 뭐가 있다.

-. 해시 함수의 조건

ㄴ 1. 일관된 함수

ㄴ 2.

-. 충돌이 뭐고, 그럴땐 어떻게 풀어야 하고.

ㄴ 백업 플랜.

+ 혼자 성장하는데 한계가 있다. 일하면서 함께 배우고 싶다.

**디자인 패턴**

왜 디자인 패턴을 적용할까요?

-> 프로그램의 유지보수를 쉽게

-> 단위 테스트를 할 수 있게 하기 위해서.

\*\* MVC

-. 모델, 뷰, 컨트롤러

-. 모델: 데이터에 관한 로직을 담는 부분

-. 뷰: 사용자에게 보여지는 화면

-. 컨트롤러: 모델과 뷰 간의 동작을 관리

예시. 계산기

사용자가 숫자를 입력 -> 뷰가 변경

컨트롤러가 뷰에 입력된 값을 모델로 전달

모델에서 값이 계산되면, 그 값을 컨트롤러가 뷰에 전달하여 화면을 갱신.

-> 장점: 프로그램을 구성하는 코드를 역할에 따라 부분적으로 나눌 수 있어서 프로그램이 커져도 유지보수가 상대적으로 쉬움.

즉, UI를 바꾸고 싶다면 뷰를, 계산 로직을 바꾸고 싶다면 모델 부분만 바꾸면 됨.

모델과 뷰는 서로에게 independent. 즉, 서로에게 직접적으로 접근 불가.

컨트롤러는 모델, 뷰와 접근 가능.

1.모델 -> 컨트롤러로 접근?

: 가능하지만, MVC 패턴에 어긋남. 모델은 데이터의 값을 바꾸고 관리하는 것만 해야하기 때문.

그래서 모델 값이 바뀌었을때 컨트롤러에게 알리기 위해서는 Notification을 사용.

모델 값이 바뀌었다는 사실을 알리면(Notification을 주면) 컨트롤러가 이를 듣고 모델이 접근.

2.뷰에서 컨트롤러로 접근?

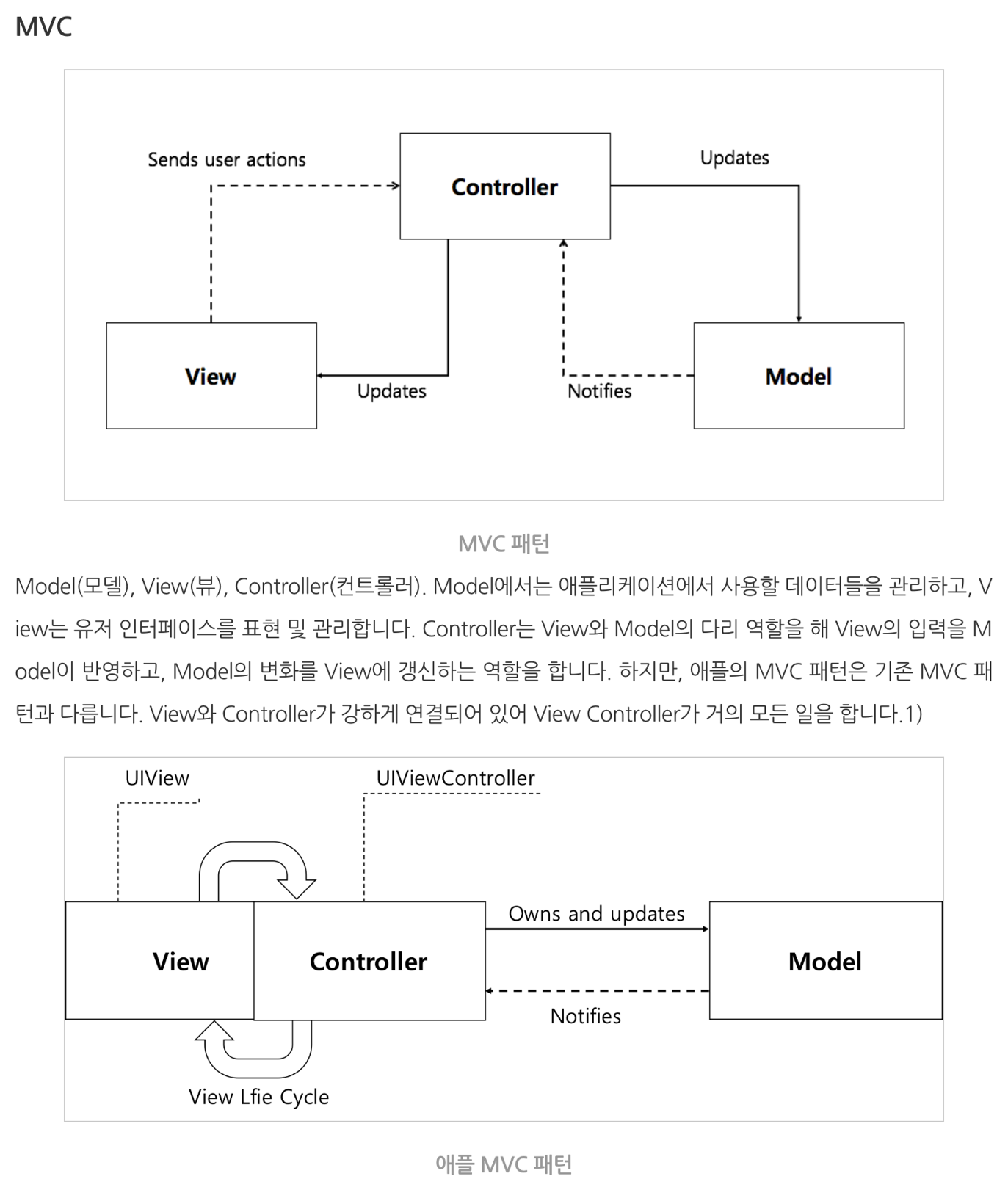
유저가 뷰에 값을 입력하거나 터치했을 때, Delegate, Datasource를 이용해서 뷰의 변화를 컨트롤러에 전달. 프로그램에서 해야할 일을 델리게이트를 통해 컨트롤러에게 위임하는 것.

=> 장/단점

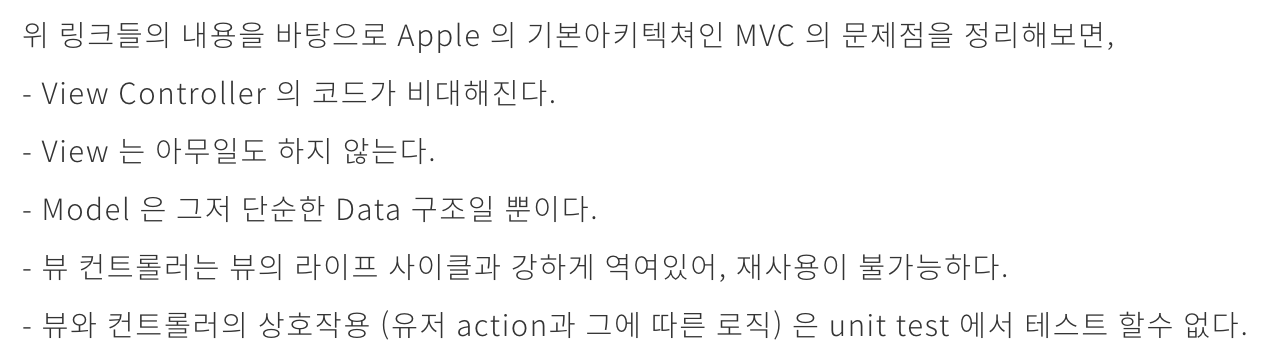
장점: 역할분담을 고려한 구조를 빠르게 구현할 수 있음. 즉, 생산성이 매우 좋음.

단점: iOS의 MVC는 모델뷰컨트롤러의 분리가 완전하지 않아서, 실제로는 뷰의 생명주기에 관한 코드가 컨트롤러에 모두 위치, 네트워크 통신도 모두 컨트롤러에 위치하기 때문에 컨트롤러 크기가 지나치게 커지는 경향이 있음.

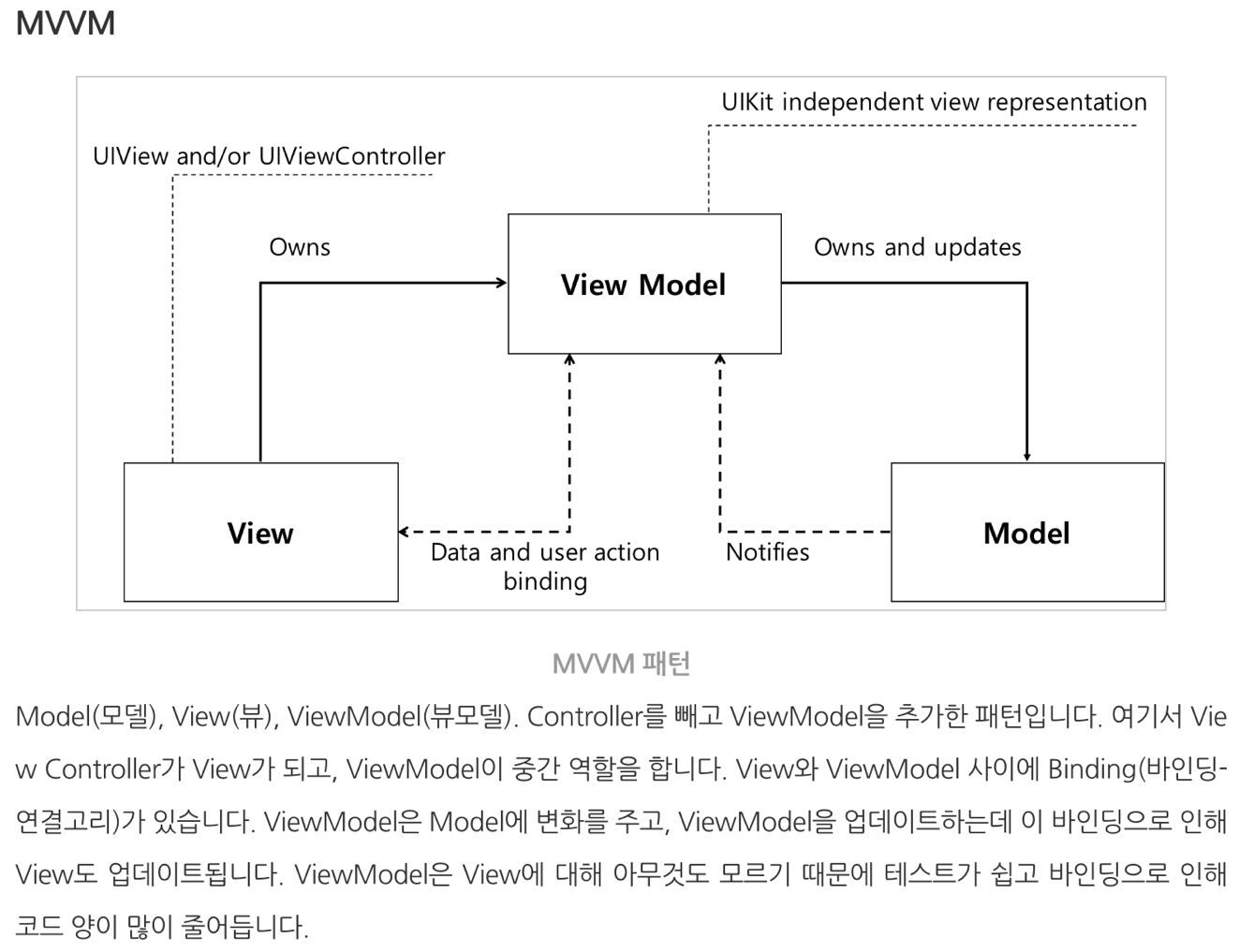
-> 이를 극복하기 위해 MVP, MVVM 등의 패턴을 사용하기도.



1) View Controller에서는 Controller가 View의 life cycle(라이프 사이클)에 관여하기 때문에 View와 Controller를 분리하기 어렵습니다. 개발자들 사이에서는 Massive View Controllers라고도 불립니다. 앱을 테스트할 때, Model은 따로 분리되어 테스트를 할 수 있어도 View와 Controller는 강하게 연결되어 있기 때문에 각각 테스트하기 어렵습니다.



\*\*MVM - Model-View-ViewModel



Model - View - ViewModel

모델은 단순히 데이터를 표현. 데이터를 의미

뷰는 보여주는 것들. But, “준비된 데이터”만 표시. 뷰모델에 의해 처리된 데이터, 즉 단순하게 만들어진 문자열을 가져와서 보여주기만 하는 역할

뷰모델은 그 외의 모든 것을 처리. View와 모델의 연결 역할.

뷰에 보여줄 데이터를 모델에서 가져와서 가공한다.

모델을 변경하는 것.

비동기 네트워크를 처리하는 것 등등.

-. UIKit 코드와 결합되어 있지 않음. UI 코드에 영향을 주지 않음. -> 단위테스트가 보다 쉬움

**Delegate**

참고1. 주영민 강사님 강의자료 - Application Life Cycle 중.

\*\* Protocol

-. 프로토콜은 원하는 작업이나 기능이 구현되도록 메서드, 프로퍼티 등으로 요구사항의 청사진(명세서)를 정의

-. 클래스, 구조체, 열거형은 프로토콜을 채택하면 프로토콜에서 요구한 사항에 대해 구현해야 함

-. 프로토콜을 통해 공통적인 작업을 강제할 수 있으며, 해당 프로토콜을 채택한 사람이 구현한 메소드에 대한 정보도 알 수 있음.

-. 프로토콜을 추상 클래스처럼 사용할 수 있음. 프로토콜을 채택한다는 건 그 타입이기도 하다는 뜻.

class Dog: Runable { }

class Horse: Runable { }

func racing(animals: [Runable]) -> Runable { }

// 프로토콜 타입으로 사용 가능.

let winner: Runable = racing(animals: [Dog(), Horse()])

\*\* Delegate

-. 클래스나 구조체에서 일부분의 할 일을 다른 인스턴스에게 대신하게 하는 디자인 패턴

-. Delegate - 뷰가 받은 이벤트나 상태를 컨트롤러에 전달해주기 위해 주로 사용 (하위 -> 상위)

-. DataSource - 컨트롤러를 통해 View 구성에 필요한 데이터를 받는 용도로도 사용 (상위 -> 하위)

// Delegate 선언부

protocol CustomViewDelegate {

    func viewFrameChanged() { }

}

class CustomView: UIView {

    var delegate: CustomViewDelegate? // delegate 프로퍼티 생성

    func layoutSubview() {

        delegate?.viewFrameChanged()  // delegate 인스턴스의 메서드 실행

        // 현재 CustomView 입장에서는 delegate 인스턴스가 존재하는지는 모름.

        // But, 뷰컨트롤러에서 custom.delegate = self 를 선언하며 delegate instance 에 값을 할당했다면

        // 컨트롤러는 프로토콜을 채택했으며, 메서드를 구현한 것.

        // 델리게이트 메서드를 사용해서 메서드 실행 및 리턴값을 받아와서 사용한다.

    }}

// Delegate 구현부

class ViewController: CustomDelegate { // 프로토콜 채택

    override func viewDidLoad() {

        super.viewDidLoad()

        let custom = CustomView() // CustomViewDelegate를 가진 CustomView 생성

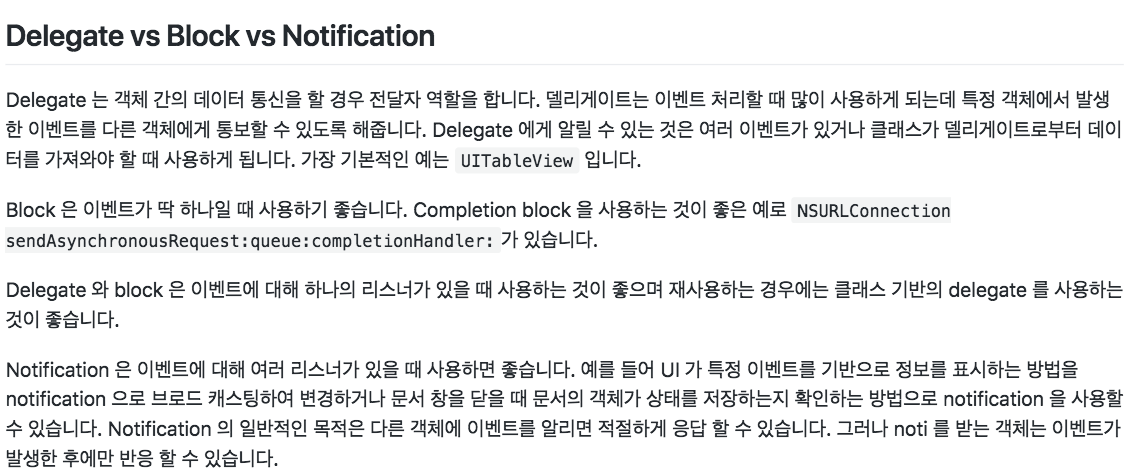
        custom.delegate = self // 커스텀 뷰의 Delegate가 ViewController 라는 것을 알려줌

    }

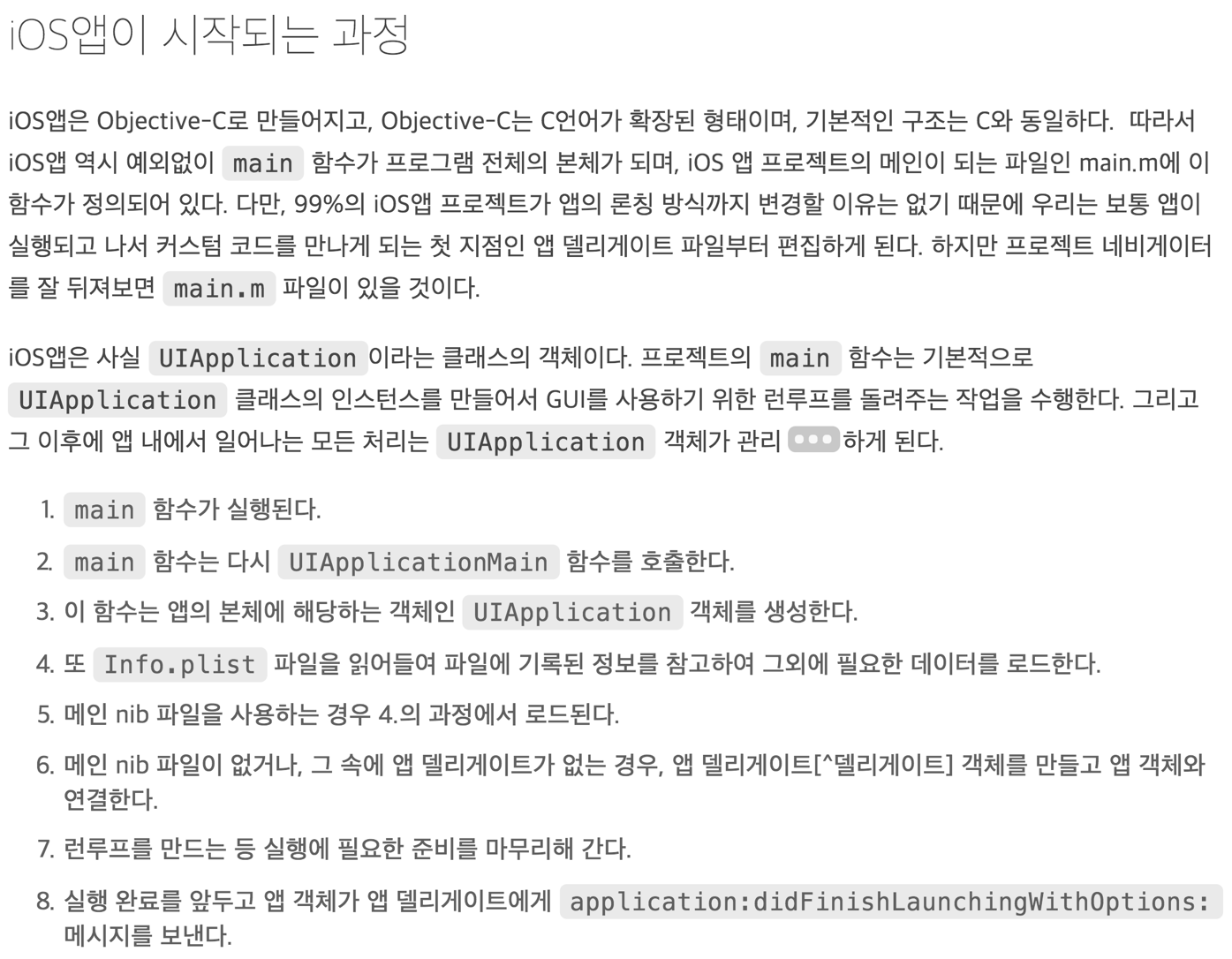
    func viewFrameChanged() { // 채택한 프로토콜 구현

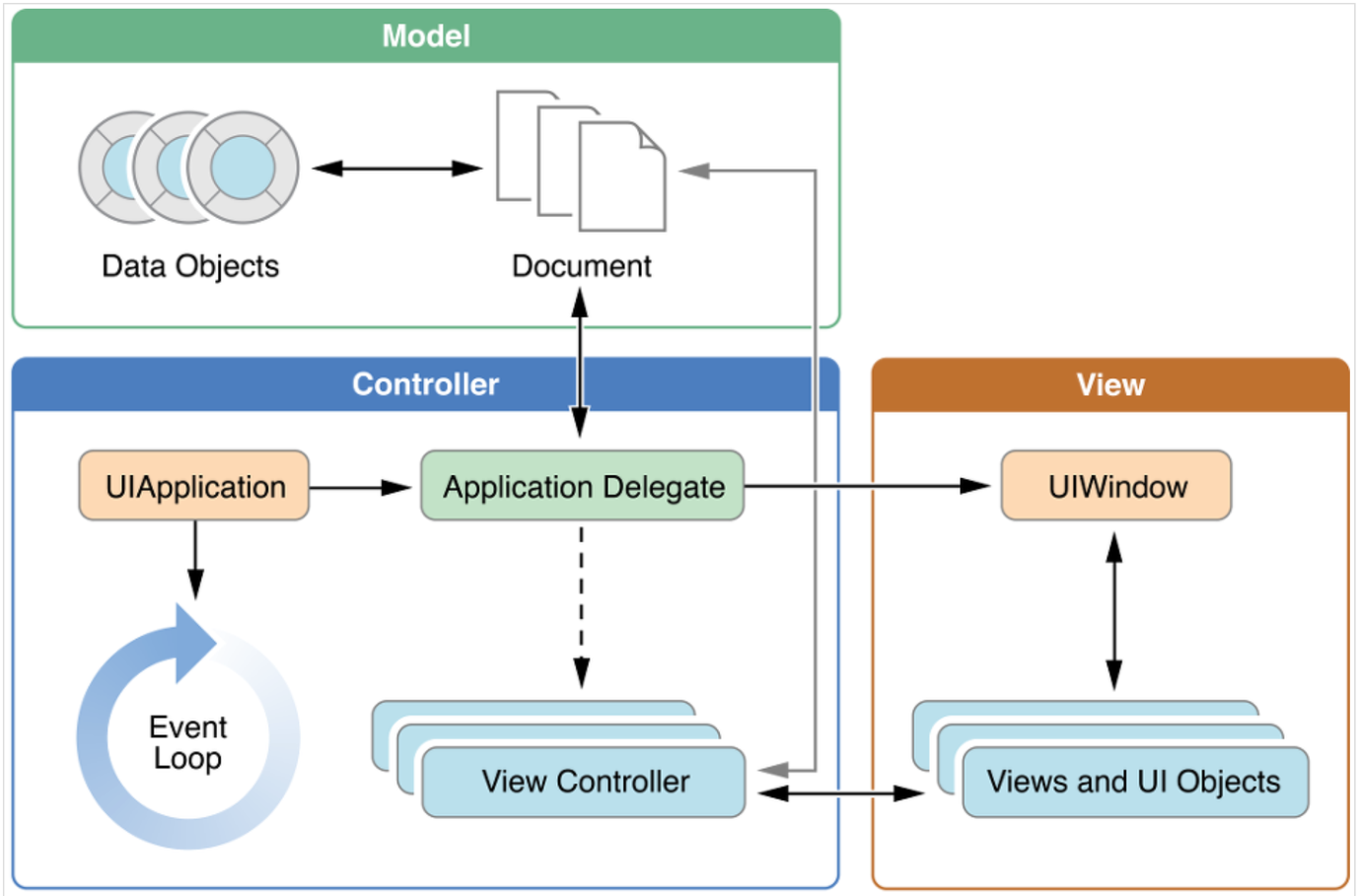
    } )

참고2.

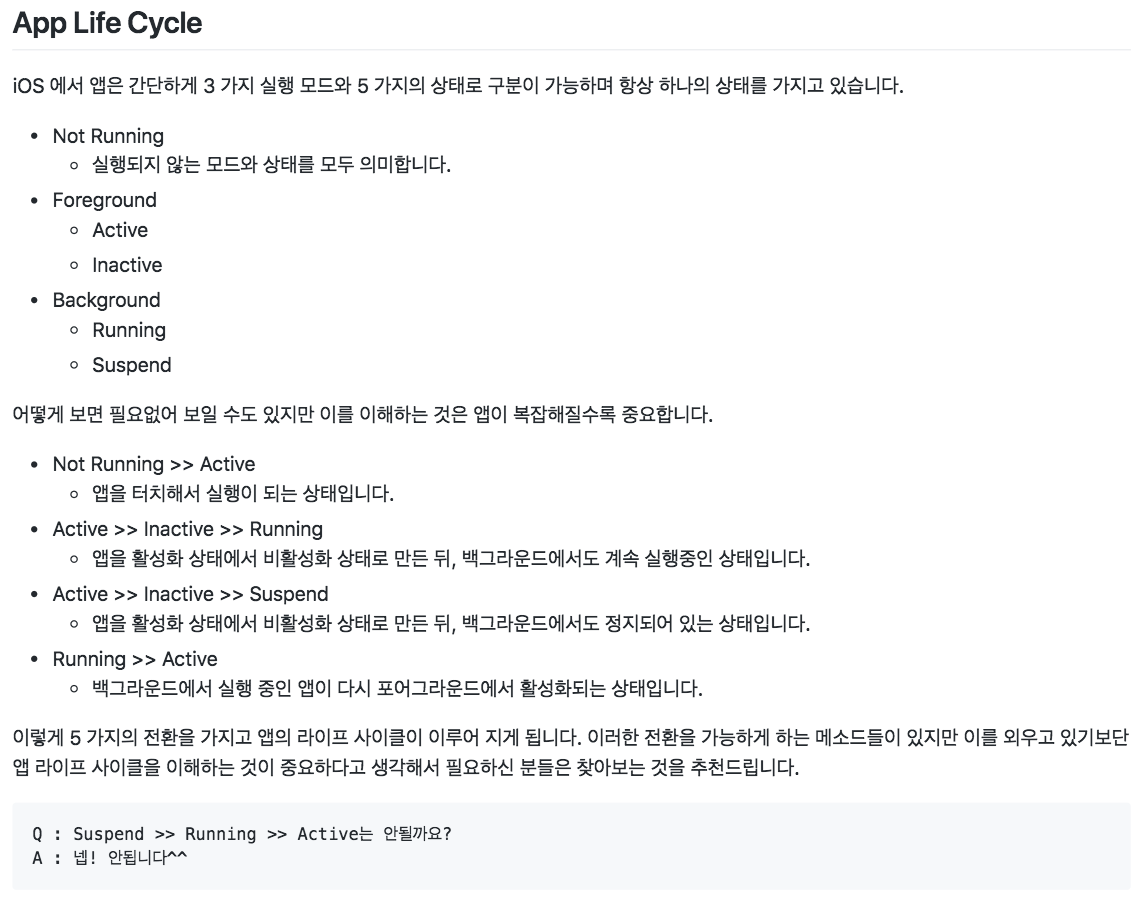


**Life Cycle**

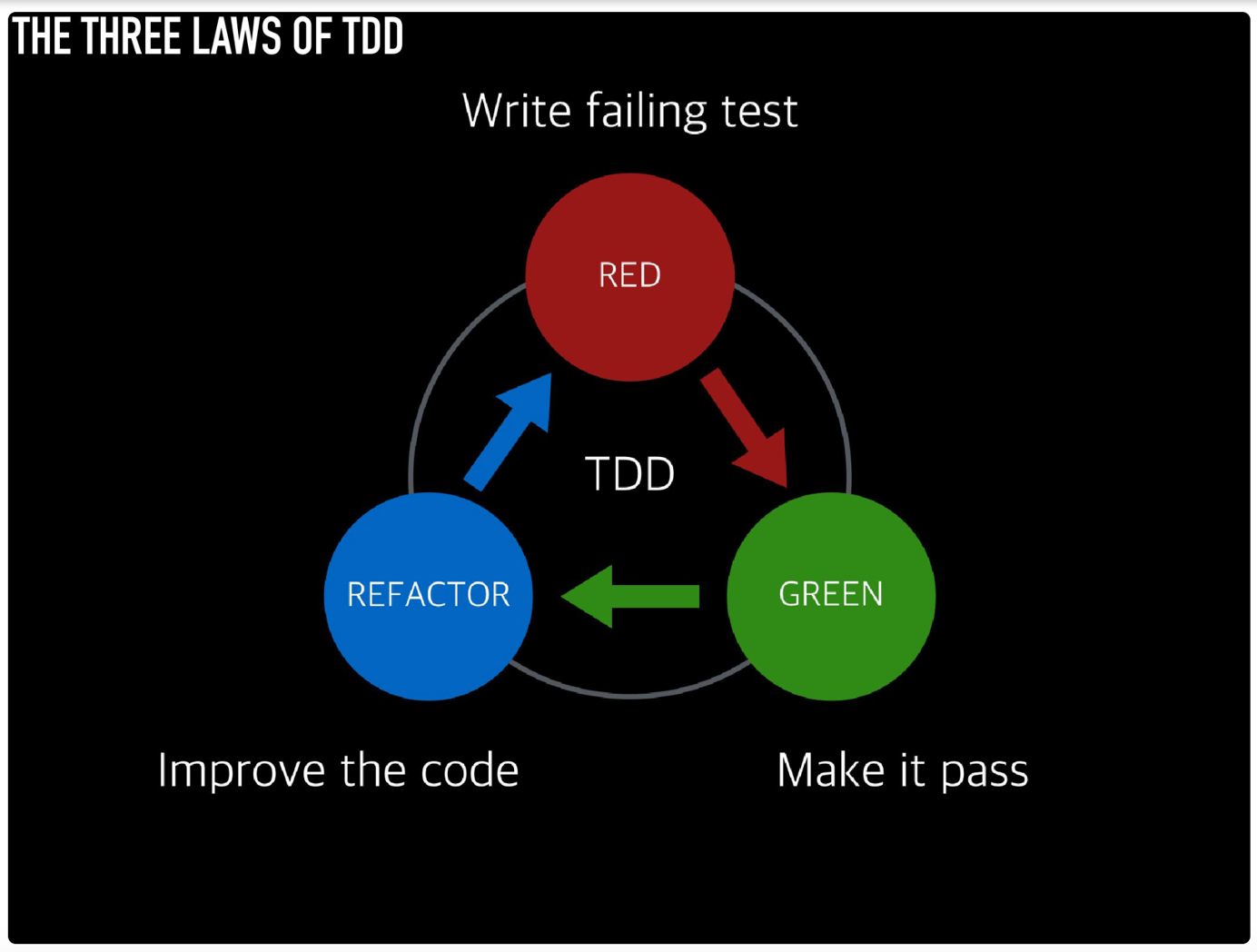




\*\* 뷰컨트롤러 생명주기



**TDD**

****

**장점**

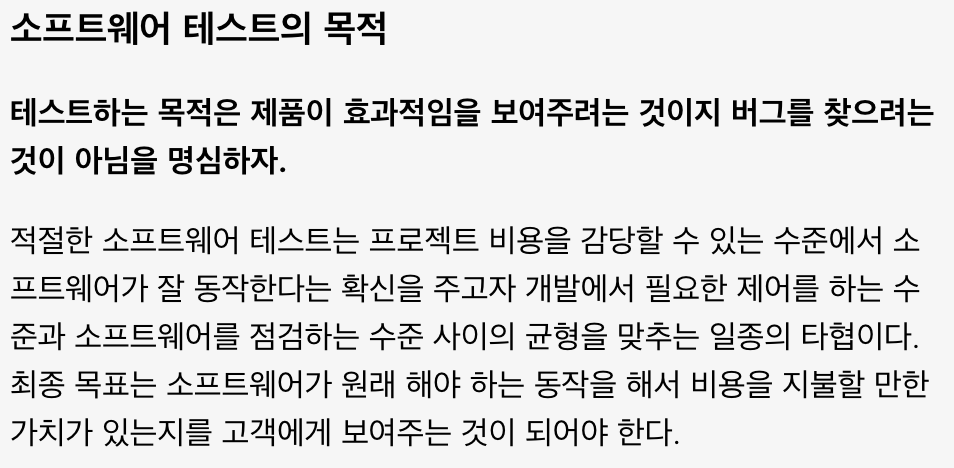
소스코드 품질이 향상되고, 필요한 코드만 작성하므로 사용하지 않는 코드를 짜지 않게 됩니다. 테스트를 통과한 코드만 존재하므로 디버깅 시간도 절약됩니다. 모듈화 디자인, 유지보수 용이성, 리팩토링 용이성과 테스트 코드 문서화라는 장점도 있습니다. 그리고 제가 가장 중요하게 생각하는 TDD의 장점은 **자신감** 을 가질 수 있다는 점입니다.

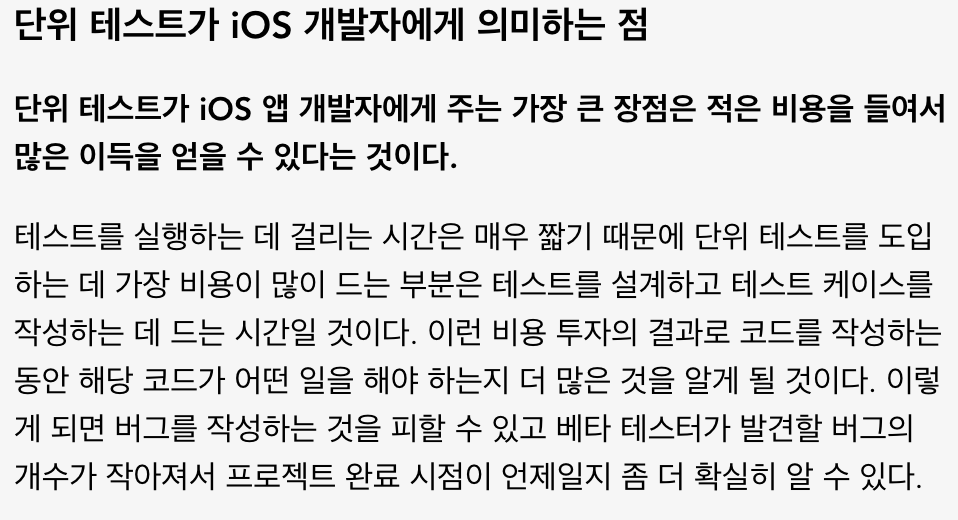
>> Kent : 프로그래머들은 그들이 작성한 코드가 잘 동작하는 것에 확신을 가져야 마땅하며, TDD가 그렇게 만드는 방법 중 하나(유일하지는 않은)이죠.

출처: [http://jinson.tistory.com/entry/한글화-프로젝트-TDD는-죽었는가](http://jinson.tistory.com/entry/%ED%95%9C%EA%B8%80%ED%99%94-%ED%94%84%EB%A1%9C%EC%A0%9D%ED%8A%B8-TDD%EB%8A%94-%EC%A3%BD%EC%97%88%EB%8A%94%EA%B0%80) [지앤선의 책 사랑]

**단점**

단점은 프로토타이핑을 빠른 시간 안에 마치고 결과를 보여줘야 하는 상황에는 부적합하며, 익숙해지는데 시간이 걸린다는 점입니다.





<https://minieetea.com/2014/01/archives/1327>

참고3. <http://junho85.pe.kr/977>

\*\* 왜 TDD가 필요한가?

clean code networks - 작동하는 클린코드를 위함.

1. 작동한다.

        -. 작동하지 않는 코드는 의미가 없다.

        -. “작동하는지”에 포인트를 두기 위해 올바르게 작동하는 경우에는 어떤 결과를 내는가를 테스트 코드로 명확히 적는다.

1. 클린코드

        -. 클린코드를 적게 된다. 테스트 코드가 있으면, step by step으로 점진적 변화를 주기 때문에 유연한 클린코드가 나올 수 있게 된다. 의존관계를 만드는데 어쩔 수 없이 유연하게 프로그램을 짜게 됨.

1. 유연함

        -. 테스트 코드를 먼저 작성하면 의존관계, 여러가지 복잡한 것들이 일찍 정리가 되고, 일찍 발견이 됨.

        -. 거대한 소프트웨어를 고치는 것 vs. Unit Test(매우 작은) 걸 고치는 걸 비교하면 저비용으로 위험부담을 낮출 수 있는 효율적인 방법.

\*\* TDD의 핵심

-. 테스트가 개발을 주도하는 것.

-. 테스트를 수동으로 하는게 아니라 자동으로 돌아가는 코드를 만드는 것

-. Unit Test이다.

    ㄴ 올바르게 작동하는지 check.

    ㄴ 올바르다면 어떻게 될건지를 테스트 코드로 작성하는 것.

    ㄴ 뭘 만들어야 하는지를 구체화 하고 명확히 하는 것.

-. 결과가 어떻게 나올것인지.

결과를 명확하게 알고 적어놓고.

결과를 내는 과정을 명확하게 알고 적고. 하나의 스펙을 의미하는 것인데, 문서로만 쓰는게 아니라 코드로 적는 것.

**ARC**

-. MRC(Manual) : 사용자가 직접, 수동으로 메모리를 관리하는 방식

-. 객체 참조하고 싶을 때 retain, 풀 때 release

-. 인스턴스 생성 - alloc

-. 카운트 0 - dealloc

참고2. 유튜브 한글링 강의

-. 클래스는 Heap에 저장 -> ARC에 의해 자동으로 관리

-. ARC (Automatic Reference Counting)

-. 참조 횟수를 세가면서 관리하는 기법. 포인터를 계속 탐지하고 있다가 참조횟수가 0이 되면 놓아버림.

-. 가비지 콜렉터와는 다른데, 표시 휩쓸고 청소하는게 가비지 콜렉터인데 얘는 매우 간헐적이고 불규칙적 & 예측 불가

-. 보통은 ARC에 의해 힙 메모리에 대해선 생각하지 않아도 되지만 ARC에 영향을 줄 수 있는 애들이 있는데 아래 3가지. 변수 선언할 때 사용하는 키워드

1.strong - default. normal reference counting. 스트롱 포인터는 힙 안에 있는 걸 강제로 힙에 머물도록 함. 포인터가 더이상 그걸 가르키지 않을 때까지.

2.weak

weak 포인터가 있다는 말은 힙에 있는 것에 아무도 관심이 없다면, 힙에서 제거할 수 있다는 의미. 그리고 nil로 설정됨. 그래서 weak은 오직 옵셔널 타입의 참조 포인터에서만 적용.

-. 예. Outlet >> Why? 뷰 계층에선, ex. UILabel의 상위 뷰는 UILabel에 대해 strong 포인터를 갖고 있어서 outlet을 따로 strong에 넣어놓지 않아도 뷰 계층에 따라 알아서 힙에 보관이 됨. but, 더이상 UILabel이 뷰의 일부가 아니라면(상위 뷰가 UILabel을 제거한다면) outlet이 nil이 될것.

3.unowned

-. 참조 횟수를 세지 말라(추적하지 말라)는 의미.

-. unowned가 있는데, 참조를 나중에 하고, 참조한 것이 힙 밖으로 버려지게 되면, 더이상 스트롱 포인터가 존재하지 않기 때문에 앱 크래쉬 남. 메모리 참조 에러.

-. 왜 필요할까?

> 객체들 간의 메모리 순환을 부수기 위해서. (weak을 쓸 수도 있음. weak을 더 자주 쓰기도)

Vs. weak?

> 옵셔널이냐 아니냐의 차이

**< 총 정리 >**

왜 메모리를 관리해야 할까?

> 메모리가 과부하되면 워닝단계에서는 백그라운드에서 돌아가는 앱들 중 가장 사용하지 않은 최초의 앱들이 종료되고, 그 이상 넘어가면 실행시키는 앱이 종료되기 때문.

iOS의 메모리 관리 방법.

하나의 객체가 생성되어 메모리에 올라가게 되면, 그 객체를 참조하는 횟수를 표시한다.

ARC 란?

카운팅이 0이 될 경우, 더이상 사용되지 않는 메모리로 인식하고 메모리에서 자동으로 해제시켜 주는 것이 ARC이다.

java의 가비지 컬렉터와는 다른건데, 가비지 컬렉터는 불규칙적인 사이클로 체크하고, 카운팅이 0인 모든 프로퍼티를 휩쓸어 없애는 반면

ARC는 카운팅이 0이 되는 순간에 해당 인스턴스를 메모리에서 제거해주기 때문이다.

> 가장 큰 차이점은

ARC는 GC (Garbage Collector)와는 다르게 런타임이 아닌 **컴파일 단에서 처리**된다.

**GC는 런타임에 메모리를 검사하기 때문에 앱 퍼포먼스에 악영향을 준다.**

그러나  ARC는 개발자가 직접 코딩하던 retain/release를 컴파일러가 자동으로 코드에 삽입시키므로, 동작시에는 MRC와 동일하여 성능 저하를 유발하지 않는다.

ARC 이전에는? MRC

수동으로 retain 해서 참조를 넣고, release로 참조를 해제하며 수동으로 카운팅 관리.

문제가 너무 많음.

1. 앱의 비정상 종료 원인 중 많은 부분이 메모리 문제. 메모리 관리는 애플의 앱 승인 거부(Rejection)의 대다수 원인 중 하나.
2. 많은 개발자들이 수동적인 (retain/release) 메모리 관리로 힘들어함.
3. retain/release 로 코드 복잡도가 증가.

매우 불편하다는 의견을 수렴. WWDC 2011, iOS 5 부터 ARC를 쓸 수 있게 되었다.

ARC로 인해 발생할 수 있는 문제: 강한 순환 참조 -> 메모리 leak이 나기 때문. / strong 키워드가 default.

이 문제를 방지하기 위해서 weak / unowned를 사용. 둘의 차이는 weak - optional이냐, unowned 값이 있냐의 차이.

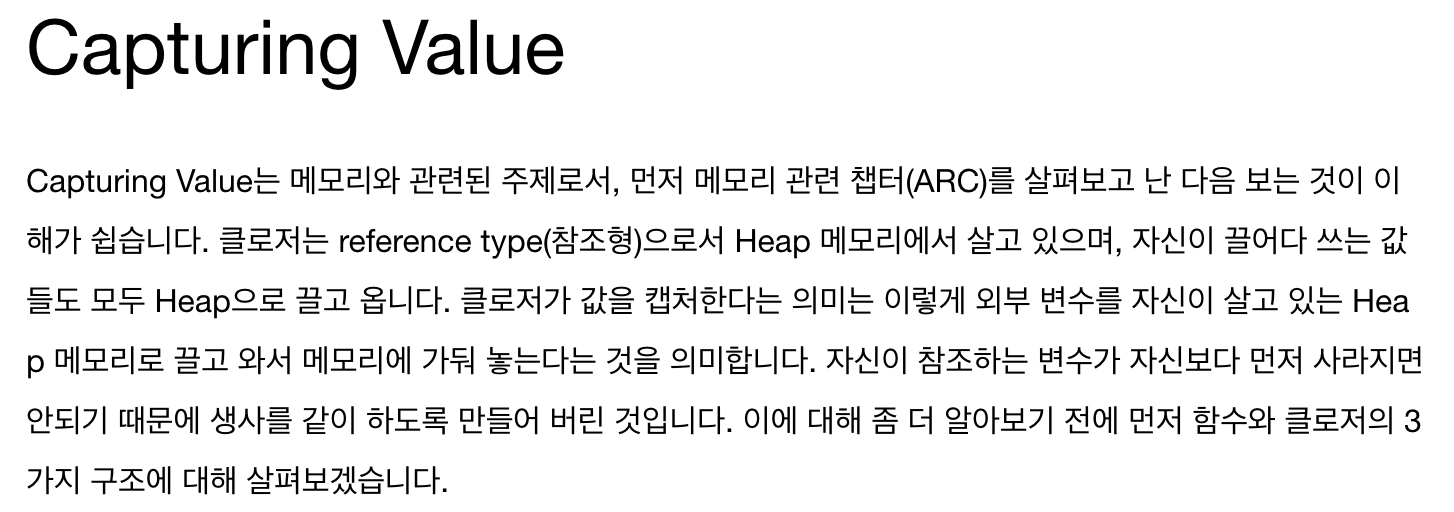
weak, unowned는 약한 참조로, 카운팅 횟수를 증가시키지 않고, 참조하는 객체가 해제되면 자동으로 해제된다.

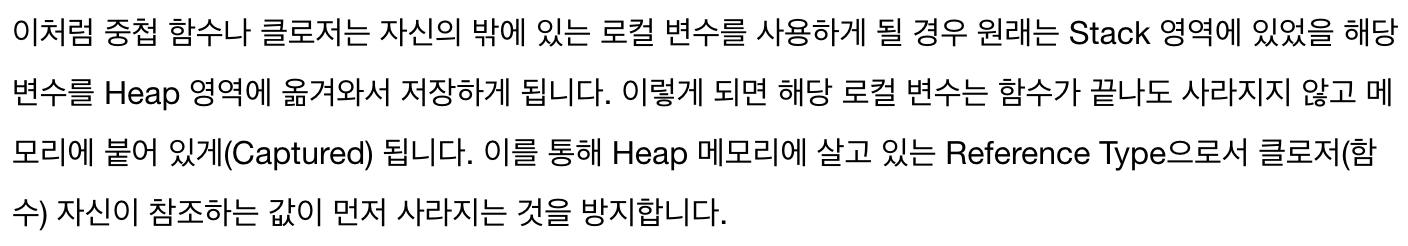
Q : weak는 언제 dealloc 될까요?

A : 마지막 강한 참조가 더 이상 객체를 가리키지 않으면 객체는 할당이 해제되고 모든 약한 참조는 dealloc 됩니다.

**Closure**

\*\* Capture





-. Heap 영역: 할당해야 할 메모리의 크기를 프로그램이 실행되는 동안(런타임) 결정해야 하는 경우(런 타임때) 유용하게 사용되는 공간 (메모리구조 참고: <http://dsnight.tistory.com/50>)

Q. 왜 동적할당을 사용할까?

> 사용자의 요구에 맞게 메모리를 할당해 주고, 다 쓰고나면 버려서 메모리 낭비를 줄이기 위해.

> 즉, 필요한만큼만 메모리를 할당해서 메모리를 효율적으로 사용하기 위해.

내가 생각하는 동적할당 하는 이유는 다음과 같다.

1. 만약 지역변수를 사용한다면 그 함수가 끝나면 해당 변수가 사라진다. 하지만 동적할당을 하면 그런 것을 신경쓰지 않고 포인터만 넘겨주면 공간을 사용가능하다.

2. 동적할당은 내가 원할 때 해제해서 메모리를 더 능률적으로 사용할 수 있다. 만약, 지역변수를 사용한다면 굳이 그 변수가 필요하지 않음에도 해당 함수가 끝날 때까지 그 변수의 스택 영역은 잡혀있다.

■ 동적 할당의 필요성 ■

우리가 프로그램을 코딩할 때에 주로 사용하는 변수나 상수 등의 형태는 컴파일 과정을 거치면서 정적 기억장소에 할당된다.

이렇게 되었을 경우 **우리가 입력의 수를 알 수 없을 경우** 큰 메모리 영역을 미리 할당해야 하는데, 이것은 **메모리를 낭비**하는 것이다.

이 때문에 프로그램이 돌아가는 중 **실시간으로 메모리 영역을 할당하고 반환하는 과정을 거치는 동적 할당**이 필요한 것이다.

**Zigzag**

- 모르면 구글에 물어보지. 외운거는 의미 없음. 면접이 중요. 과제식 면접을 준다고함.

특정 기간에 문제를 해결하는 능력( 생소한 라이브러리 혹은 면접자가 경험해보지 않은 라이브러리로 간단한 프로그램 작성 )

→ api를 참조하여 sorting만 쓰면 되는데 그걸 못하는 사람도 있다고함

신기술 도입에 대해서 굉장히 좋아함.

항상 새로운것을 추구하는 개발자를 원하고 빠르게 적응하는 개발자 중요.

nodejs로의 전환을 하는 중.

Ios 를 swift로 바꾸는 등 활발하게 새로운 기술로 바꾸는 중.

예상치 못한문제는 다 있지만, 그래도 가치있는 일이다. 기술적 검토는 중요함

<스타트업에서 개발팀과 다른팀과의 충돌이 생겼을 때 >

-개발팀이 기획, 운영 분업 또는 의견 충돌이 생겼을 때 engineer의 처세 및 대화법

Zigzag - 잘 말해야함. 개발팀이 뚜렷한 의견을 가지고 있어야 함. 전체적인 그림(서비스)에 대한 명확한 이해가 필요함.

기획팀과의 트러블보다는 자유로운 소통문화를 지향하면서 정방향에 가치를 만드는 것이 중요함.

⇒ 문제가 생기지 않는 프레임을 만드는 노력이 중요.