Protocol Oriented Programming

Swift - iOS 앱 개발 심화과정

Protocol Oriented Programming

목표

프로토콜 지향 프로그래밍의 대두와

그 의미를 짚어보고

간단한 POP 코딩을 경험해보자.

Protocol in Objective-C

단지 기능의 청사진의 역할을 수행

주로 Delegate, DataSource 등으로 이용

기본 구현 (Default Implementation) 불가

- 카테고리 적용 불가

Protocol in Swift

Objective-C의 프로토콜 기능은 기본

기본 구현(Default Implementation) 가능

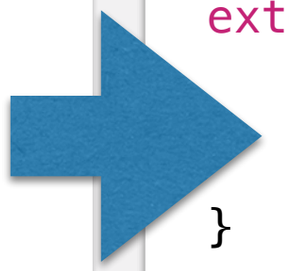
- Protocol + Extension = Protocol extension
- 특정 타입이 할 일 지정 + 구현을 한 방에!

Protocol extension

Protocol default implementation

프로토콜을 따르겠다고 선언만 하면 어디서든 해당 프로토콜의 기능을 사용 가능

```
protocol LayoutDrawable {
    func drawSomeLayout()
}

class MyView: UIView, LayoutDrawable {
    func drawSomeLayout() {
        // Draw some layout...
    }
}
```



```
protocol LayoutDrawable {
    func drawSomeLayout()
}

class MyView: UIView, LayoutDrawable {
}

extension LayoutDrawable {
    func drawSomeLayout() {
        // Draw some layout...
    }
}


extension UIView: LayoutDrawable { }
extension SKNote: LayoutDrawable { }
```

Protocol을 만들게 된 배경

상속의 한계

- 서로 다른 클래스에서 상속받은 클래스는 동일한 기능을 구현하기 위해 중복코드 발생
    - Protocol default implimentation으로 해결

카테고리의 한계

- 프로퍼티 추가 불가
- 오직 클래스에만 적용 가능
- 기존 메서드를 자신도 모르게 오버라이드 가능
    - Protocol default implimentation으로 해결

참조타입의 한계

- 동적 할당과 참조 카운팅에 많은 자원 소모
    - Value Type으로 해결

Protocol을 만들게 된 배경

class
Person

class
Bird

struct
Frog

struct
Turtle

struct
Fish

protocol
Runable

protocol
Flyable

protocol
Swimable

protocol
Talkable

Protocol을 만들게 된 배경

class Person	protocol Runable	protocol Talkable	protocol Swimable
class Bird	protocol Flyable		
struct Frog	protocol Runable	protocol Swimable	
struct Turtle	protocol Swimable		
struct Fish	protocol Swimable		

Protocol을 만들게 된 배경

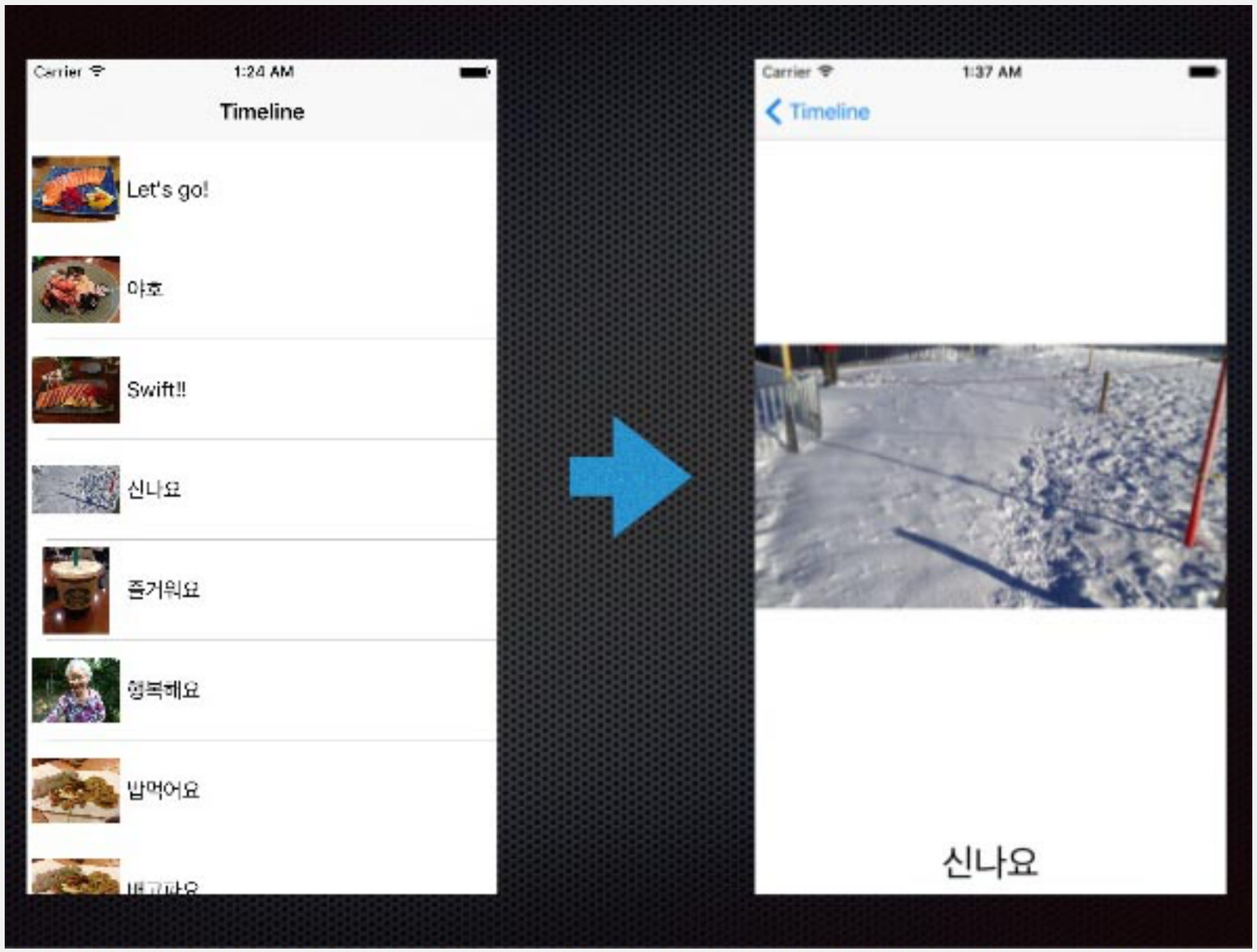
class Person	protocol Runable	protocol Talkable	protocol Swimable
class Bird	protocol Flyable		
struct Frog	protocol Runable	protocol Swimable	
struct Turtle	protocol Swimable		
struct Fish	protocol Swimable		

Default Implementation (Extension)

POP 적용예제

“테이블 뷰 형식의 타임라인 만들기”

타임라인을 나타낼 수 있는 테이블 뷰 컨트롤러 구성하기



POP 적용예제

기획 1 - 타임라인을 나타낼 수 있는 테이블 뷰 컨트롤러 구성하기

TimelineTableViewCell

```
class TimelineTableViewCell: UITableViewCell {
    var mediaImageView: UIImageView
    var note: UILabel
    var content: NSDictionary
}

class DetailViewController: UIViewController {
    var mediaImageView: UIImageView
    var note: UILabel
    var content: NSDictionary
}

class TimelineTableViewController: UITableViewController {
    var contents: [NSDictionary]
}
```

POP 적용예제

기획 2 - 사진첩처럼 볼 수 있는 모두 추가

모델에 프로토콜 중심 프로그래밍과 밸류 타입 적용하기

Table View Cell

Collection View Cell

```
class Content {
    var URLString: String
    var note: String
}
```

```
struct Content {
    var URLString: String
    var note: String
}
```

POP 적용예제

Model Property

```
class TimelineTableViewController: UITableViewController {
    var contents: [Content]
    // ...
}

class TimelineCollectionViewController: UITableViewController {
    var contents: [Content]
    // ...
}
```

```
class TimelineTableViewController: UITableViewController, ContainContents {
    var contents: [Content]
    // ...
}

class TimelineCollectionViewController: UITableViewController,
ContainContents {
    var contents: [Content]
    // ...
}

protocol ContainContents {
    var contents: [Content] { get }
}
```

POP 적용예제

Model Property

```
class TimelineTableViewController: UITableViewController, ContainContents { ... }

class TimelineCollectionViewController: UITableViewController, ContainContents { ... }

protocol ContainContents {
    var contents: [Content] { get }
}

class TimelineContentObject {
    static let shared = TimelineContentObject()
    var contents: [Content] = [Content]()
}

extension ContainContents {
    var contents: [Content] {
        return TimelineContentObject.shared.contents
    }
}
```

POP 적용예제

View with POP & Value

Table View Cell

Collection View Cell

Detail View Controller’s view

POP 적용예제

view에 프로토콜 중심 프로그래밍과 밸류 타입 적용하기

- 프로토콜 명시하기

```
protocol MediaContainer: class {
    var content: Content? { get set }
    var media: UIImageView { get }
    var note: UILabel { get set }

    func contentChanged()
}

extension MediaContainer {
    func contentChanged() {
        // Update view...
    }
}
```

POP 적용예제

view에 프로토콜 중심 프로그래밍과 밸류 타입 적용하기

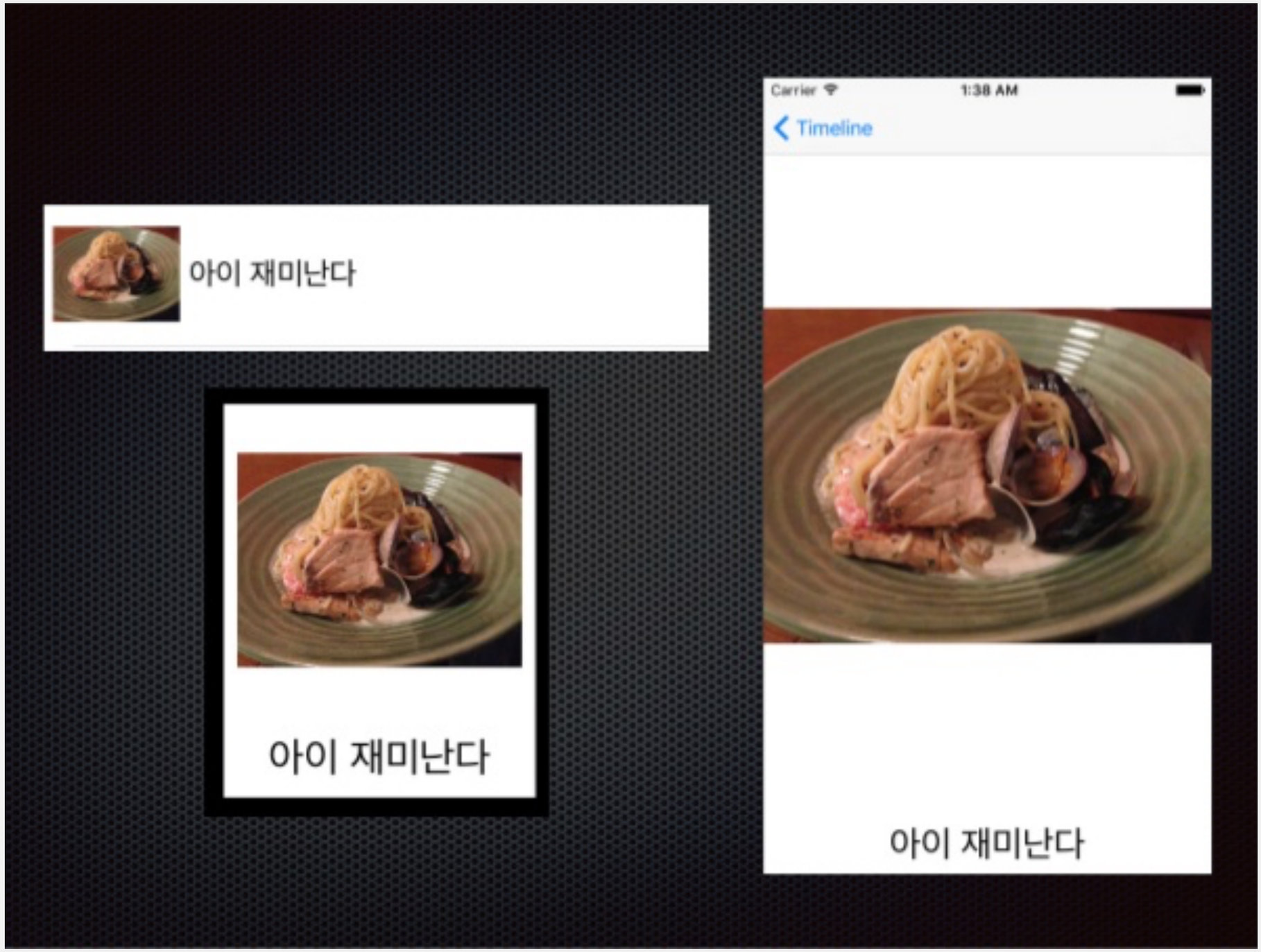
- 같은 기능과 역할을 수행하도록 기능 블럭을 가져와서 사용할 수 있음

```
class TimelineTableViewCell:
UITableViewCell,
MediaContainer {
    var media: UIImageView
    var note: UILabel
    var content: Content? {
        didSet {
            contentChanged()
        }
    }
}

class TimelineCollectionViewCell:
UICollectionViewCell,
MediaContainer {
    var media: UIImageView
    var note: UILabel
    var content: Content? {
        didSet {
            contentChanged()
        }
    }
}
```

```
class DetailViewController:
UIViewController,
MediaContainer {
    var media: UIImageView
    var note: UILabel
    var content: Content? {
        didSet {
            contentChanged()
        }
    }
}
```

POP 적용예제

view에 프로토콜 중심 프로그래밍과 밸류 타입 적용하기



POP 적용예제

Controller with POP & Value

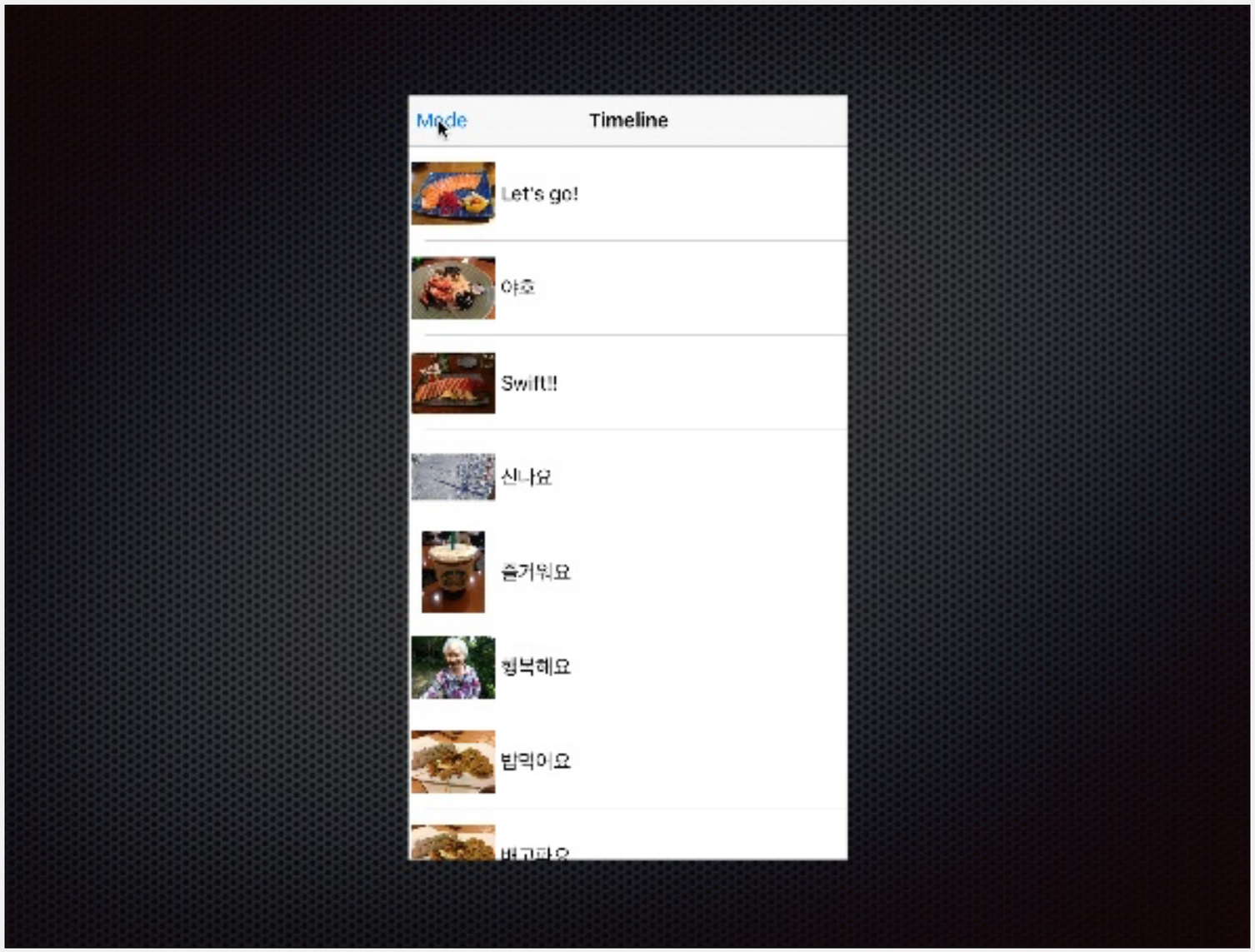
Controller에 프로토콜 중심 프로그래밍과 밸류 타입 적용하기

- tableViewController와 collectionViewController의 공통 기능인 다음화면 보여주기 기능을 뽑아 프로토콜로 만들기

```
protocol CanShowDetailView {
    func showDetailView(withContent content: Content)
    var navigationController: UINavigationController? { get }
}

extension CanShowDetailView {
    func showDetailView(withContent content: Content) {
        // Show detail view...
    }
}

class TimelineTableViewController: UITableViewController,
ContainContents, CanShowDetailView {
    // ...
}

class TimelineCollectionViewController: UITableViewController,
ContainContents, CanShowDetailView {
    // ...
}
```

POP 적용예제

Controller with POP & Value



POP 적용예제

기획 3 - 영상을 볼 수 있는 모드 추가

Refactoring하기

Table View Cell

Image or Video

Collection View Cell

Detail View Controller’s view

POP 적용예제

Refactoring하기

- 셀의 imageView 부분을 image와 video 모두 표현 가능하도록 변경
- var media에 들어가는 타입을 기존 UIImageView에서 프로토콜로 변경

```
protocol ContentPresentable: class, Layout {
    var frame: CGRect { get set }
    var canPresentContent: Bool { get }
}

extension ContentPresentable {
    var canPresentContent: Bool {
        return true
    }
}

extension UIImageView: ContentPresentable { }
extension AVPlayerLayer: ContentPresentable { }
```

POP 적용예제

container 프로토콜 변경

```
protocol MediaContainer: class {
    var content: Content? { get set }

    var media: UIImageView { get }

    var note: UILabel { get set }



    func contentChanged()
}

extension MediaContainer {
    func contentChanged() {
        // Update view...
    }
}
```

```
struct Content {




    var URLString: String
    var note: String
}
```

POP 적용예제

container 프로토콜 변경

```
protocol MediaContainer: class {
    var content: Content? { get set }

    var media: ContentPresentable { get }

    var note: UILabel { get set }

    var videoLayer: AVPlayerLayer { get }
    var mediaImageView: UIImageView { get }

    func contentChanged()
}

extension MediaContainer {
    func contentChanged() {
        // Update view...
    }
}
```

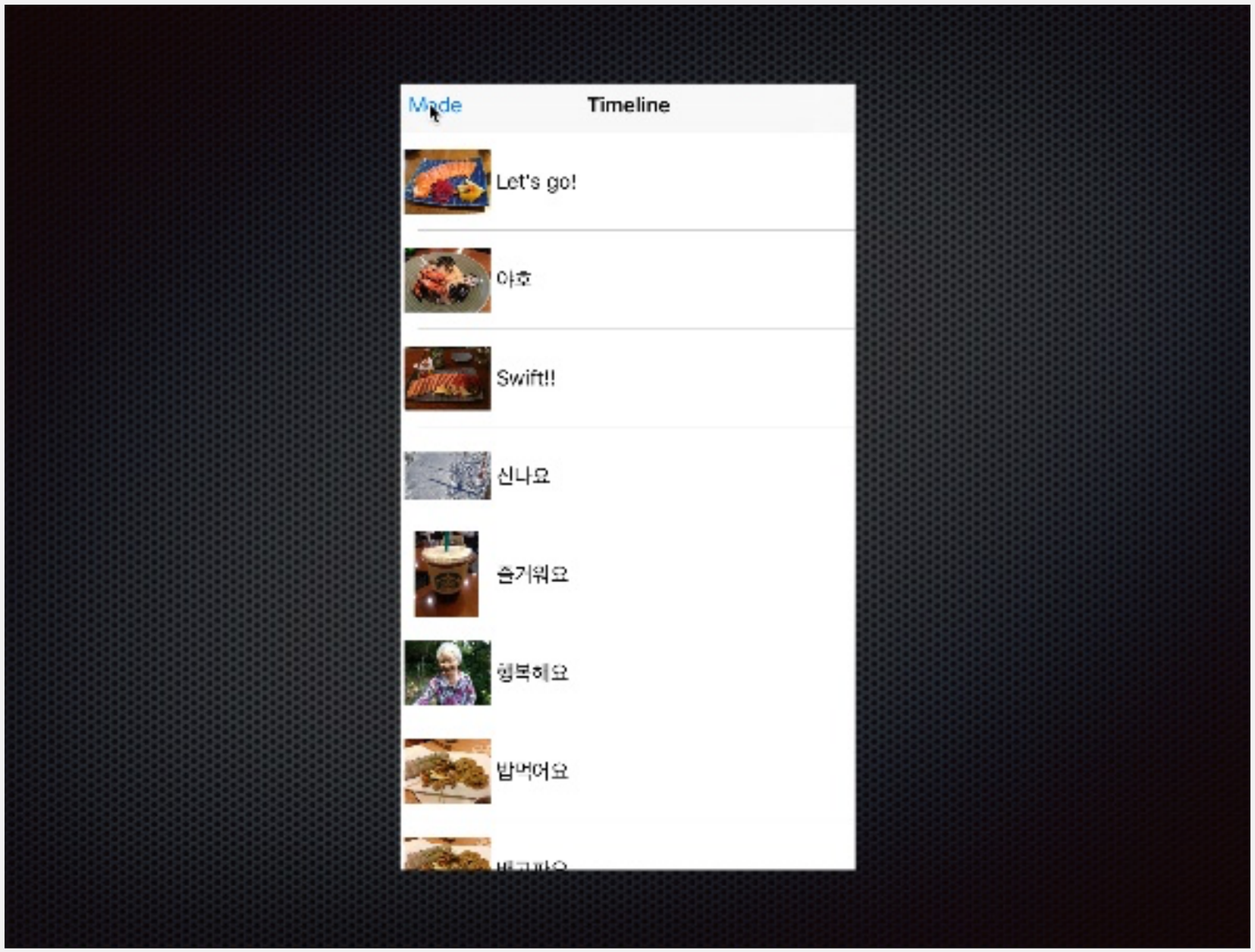
```
struct Content {

    enum MediaType {
        case image, video
    }

    var type: Content.MediaContainer

    var URLString: String
    var note: String
}
```

POP 적용예제

extension에 computed property 추가

```
extension MediaContainer {
    func contentChange() {
        // Update view...
    }

    var media: ContentPresentable {
        switch content!.type {
        case .image:
            return mediaImageView
        case .video:
            return videoLayer
        }
    }
}
```

POP 적용예제



Swift에서 POP의 장점

범용적인 사용

- 클래스, 구조체, 열거형 등등 모든 타입에 적용 가능
- 제네릭과 결합하면 더욱 파급적인 효과 (Type safe & Flexible code)
    - 하나의 타입으로 지정 가능

상속의 한계 극복

- 특정 상속 체계에 종속되지 않음
- 프레임워크에 종속적이지 않게 재활용 가능

Swift에서 POP의 장점

적은 시스템 비용

- Reference type cost > Value type cost

용이한 테스트

- GUI 코드 없이도 수월한 테스트

Swift에서 POP의 한계

Objective-C 프로토콜에 Swift Extension을 붙여도 Protocol default implementation이 구현되지 않음

자주 사용되는 Delegate, DataSource 등 프레임워크 프로토콜에 기본 구현 불가

결론

Value Type을 사용하여 성능상의 이득을 취하자

Protocol + Extension + Generic은 환상의 조합이다

이제 상속을 통한 수직 확장이 아닌 Protocol과 Extension을 통한 수평 확장과 기능추가를 고민해 볼 때

관련자료

Protocol-Oriented Programming in Swift (#WWDC15, 408)

- https://developer.apple.com/videos/play/wwdc2015/408/

Building Better Apps with Value Types in Swift (#WWDC15, 414)

- https://developer.apple.com/videos/play/wwdc2015/414/

Protocol and Value Oriented Programming in UIKit Apps (#WWDC16, 419)

- https://developer.apple.com/videos/play/wwdc2016/419/

LET'SWIFT 2016 Session : 스위프트 퍼포먼스 이해하기 - 유용하 님

- https://news.realm.io/kr/news/letswift-swift-performance/