

# Protocol Oriented Programming

데릭

보리입니다

바드

# Classes Are Awesome

- Encapsulation
- Access Control
- Abstraction
- Namespace
- Expressive Syntax
- Extensibility

## OOP, Class는 놀랍다

- 캡슐화: 관련 데이터들과 연산자를 그룹화 할 수 있다
  - 접근 제어: 외부로부터 코드 내부와 외부로 구분하는 벽을 세울 수 있다 -> 불변성 유지
  - 추상화: 윈도우, 커뮤니케이션 채널과 같은 관련 아이디어를 대표시킬 수 있음
  - 네임 스페이스: 충돌을 방지하는데 도움이 되는 네임 스페이스 제공
  - 놀라운 표현 구문들을 가지고 있음 -> 메서드 호출과 프로퍼티를 작성하고 함께 연결할 수 있음, subscript를 만들 수 있음, 연산 프로퍼티 가능
- 확장성이 있음

접근제어, 추상화, 네임 스페이스를 사용하면 복잡성을 관리할 수 있음

# Types Are Awesome

- Encapsulation
- Access Control
- Abstraction
- Namespace
- Expressive Syntax
- Extensibility

I can do all  
that with structs  
and enums.

타입이 짱임 -> Struct와 Enum으로 다 할 수 있음

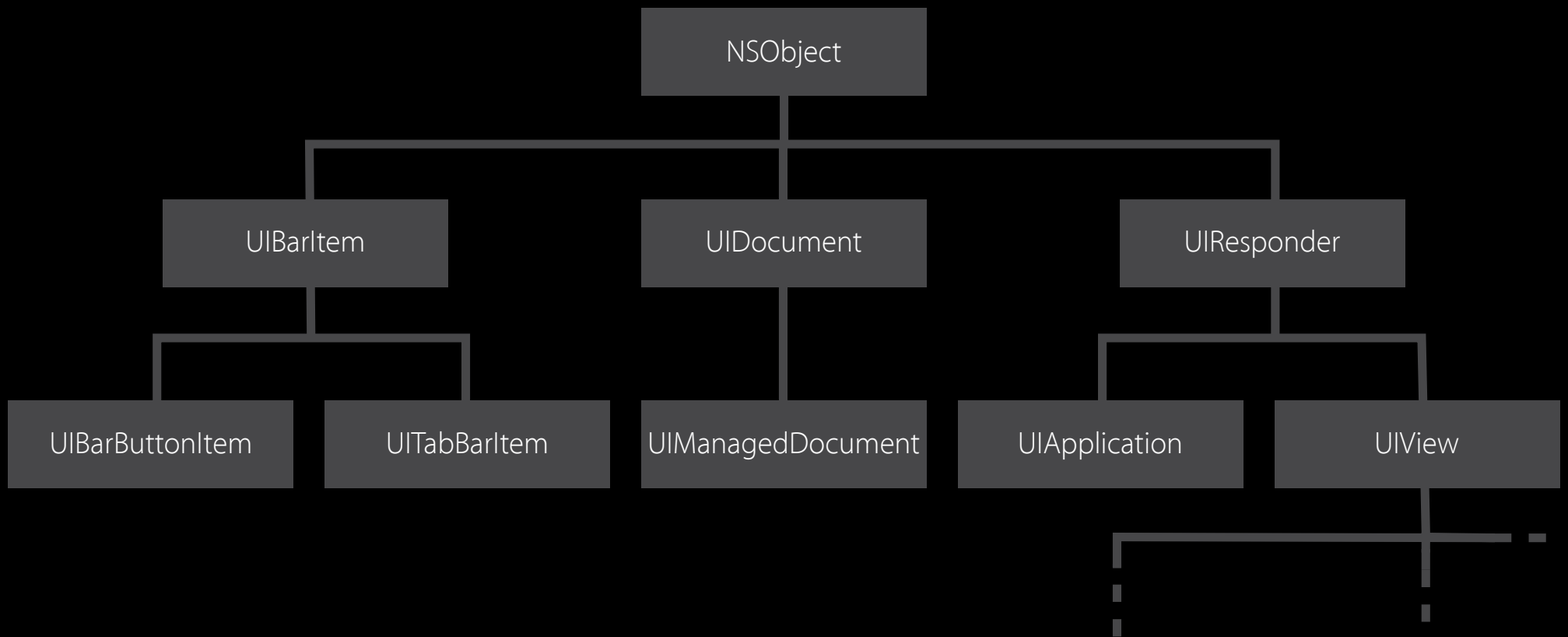
- Swift에서는 어떠한 타입이든 1급 객체(first class citizen)이기 때문에 이러한 모든 기능을 활용할 수 있음

그렇다면 객체 지향 프로그래밍의 핵심 기능은 무엇일까?

- 상속과 같이 클래스로만 수행할 수 있는 것에서 비롯되어야 함

# Classes Are Awesome

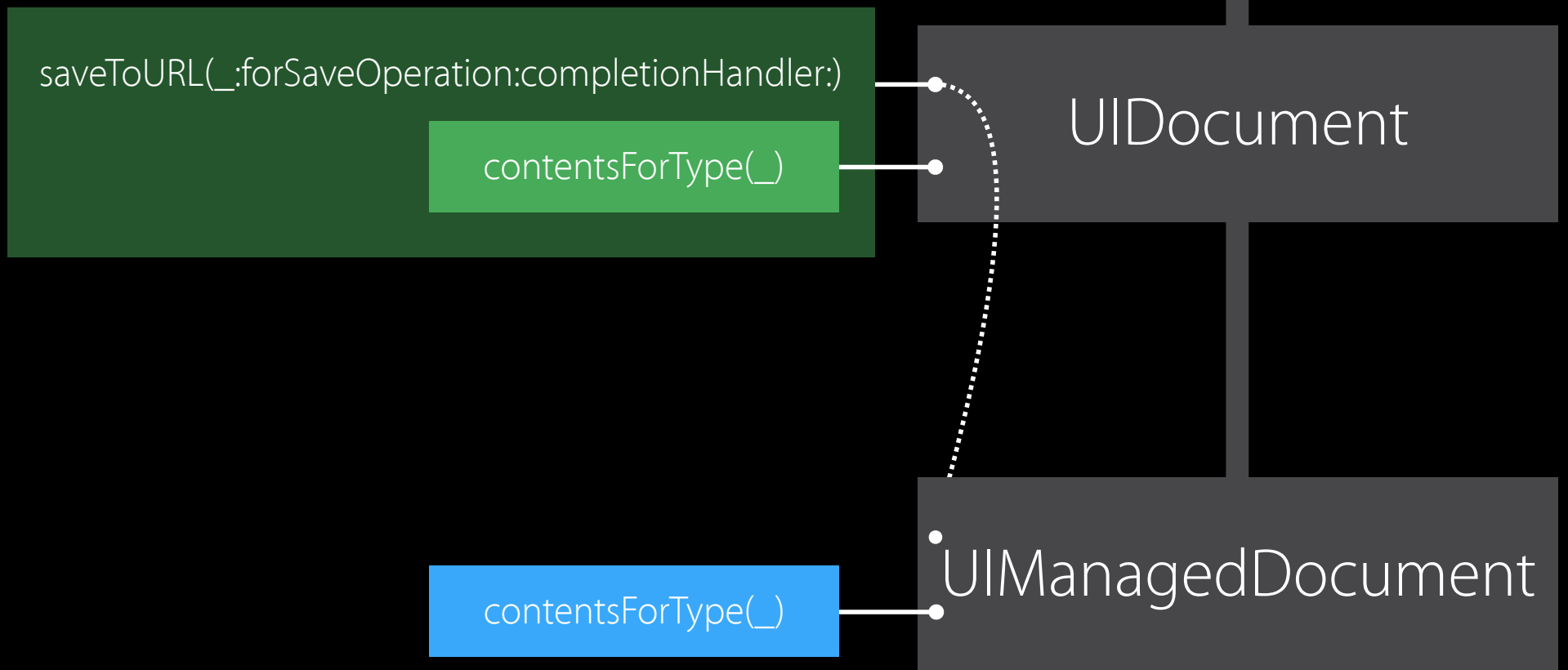
## Inheritance Hierarchies



이러한 구조를 통해 코드 공유와 세분화된 사용자 지정을 모두 가능하게 하는 방법에 대해 구체적으로 생각하게 됨

# Classes Are Awesome

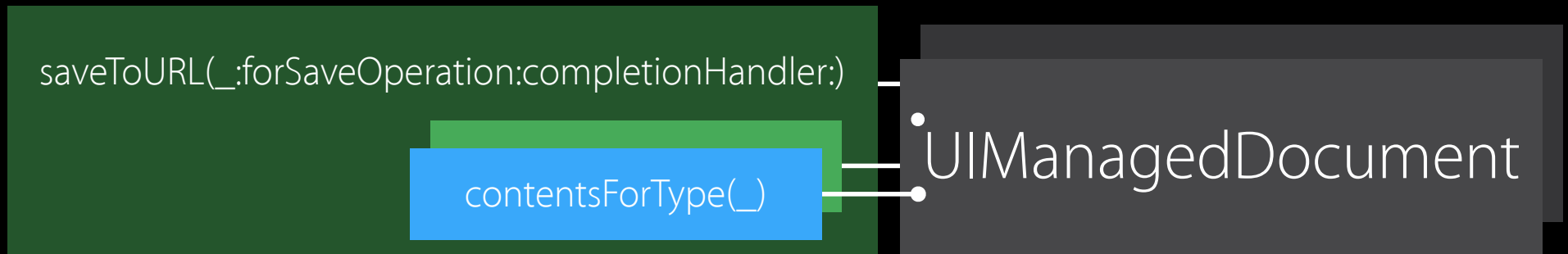
Customization points and reuse



슈퍼클래스는 복잡한 로직의 실질적인 메서드를 정의할 수 있고,  
서브 클래스는 슈퍼클래스가 수행하는 모든 작업을 무료로 가져옴  
단지 상속 할 뿐

# Classes Are Awesome

Customization points and reuse



진정한 마술은 서브클래스가 **override** 할 때 생김

- 이런 커스터마이징 한 것이 상속한 구현들과 함께 놓임
- 이것은 복잡한 로직들을 재사용 할 수 있음 -> 개방된 유연성 다양성을 가능케 함

ㅇㅋㅇㅋ 알겠음

근데 나는 구조체로 커스터마이징함

그렇다면 비용에 대해 얘기해보자

# 1. Implicit Sharing

## The sad story

Defensive Copying

Inefficiency

Race Conditions

Locks

More Inefficiency

Deadlock

Complexity

Bugs!

## 1. 객체가 공유되다 보니 많은 문제가 생김

- 너무 많은 복사가 생김 -> 코드 속도가 느려짐
- 디스패치 큐에서 무언가를 처리하고 스레드가 변경 가능한 상태를 공유하기 때문에 Race Condition이 발생하여 불변성을 보호하기 위해 lock을 해줘야 함 -> lock은 코드 속도를 더 느리게 하고 교착 상태로 이어질 수 있음
- 결국 이 모든 것은 복잡성이 추가되며 결국 버그가 되어버림

# This is not news.

@property(copy), coding conventions...

이를 처리하기 위해 수년동안 @property(copy) 및 Coding Convention과 같은 언어 기능의 조합들을 적용해 옴

여기서 @property란..?



# 1. Implicit Sharing

## NOTE

It is not safe to modify a mutable collection while enumerating through it. Some enumerators may currently allow enumeration of a collection that is modified, but this behavior is not guaranteed to be supported in the future.

One effect of implicit sharing on Cocoa

**Cocoa 문서에는 반복하는 동안 변경 가능한 컬렉션을 수정하는 것에 대한 경고가 있음**

**- class에 내재된 변경 가능한 상태의 암시적 공유 때문**

# 1. Implicit Sharing

## NOTE

It is not safe to modify a mutable collection while enumerating through it. Some enumerators may currently allow enumeration of a collection that is modified, but this behavior is not guaranteed to be supported in the future.

One effect of implicit sharing on Cocoa

하지만 Swift에는 적용되지 않음

- Swift 컬렉션은 모두 값 유형이기 때문에 반복하는 컬렉션과 수정하는 컬렉션이 서로 다름

## 2. Inheritance All Up In Your Business

One superclass — choose well!

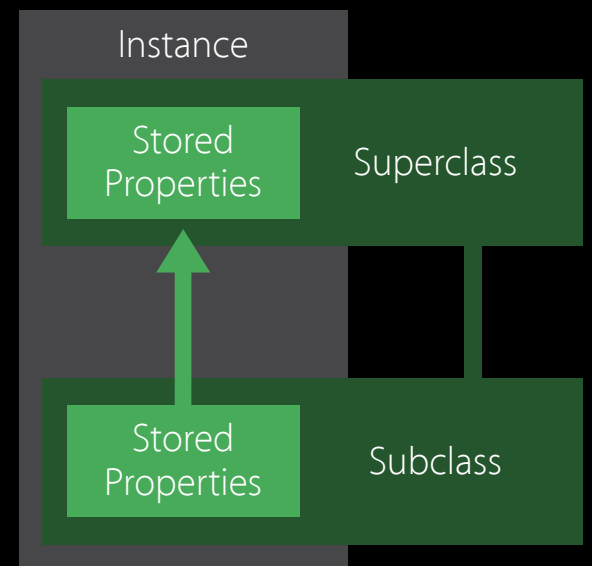
Single Inheritance weight gain

No retroactive modeling

Superclass may have stored properties

- You must accept them
- Initialization burden
- Don't break superclass invariants!

Know what/how to override (and when not to)



## 2. Class 상속이 너무 거슬림

- 일체형 -> 하나의 superClass를 가짐
  - 그렇다면 여러 추상화를 모델링해야 하는 경우 어떻게 해야 하나?
  - 컬렉션이 되어 직렬화 될 수 있을까? -> 컬렉션과 직렬화가 Class라면 그렇지 않음
- 클래스 상속은 단일 상속이기 때문에 관련될 수 있는 모든 것이 상속되면서 Class가 부풀려짐
- 원본은 수정하지 않고 재사용하여 확장시킬 수 없음
  - 나중에 어떤 확장이 아니라 Class를 정의하는 순간에 superClass를 선택해야 함
- superClass에 저장된 프로퍼티가 있는 경우
  - 어쩔 수 없이 받아드려야 함
  - 초기화 해줘야 함
  - superClass의 불변성을 깨뜨리지 말아야 함
    - superClass의 불변성을 깨뜨리지 않고 상호작용하는 방법도 이해해야 함
- 메서드가 어떤 식으로 override되고 체이닝 될지 알 수 없음 -> 그래서 delegate 패턴 사용하는 것임

# 3. Lost Type Relationships

```
class Ordered {  
    func precedes(other: Ordered) -> Bool { fatalError("implement me!") }  
}
```

```
class Number : Ordered {  
    var value: Double = 0  
    override func precedes(other: Ordered) -> Bool {  
        return value < other.value  
    }  
}
```

'Ordered' does not have a member  
named 'value'

3. Swift에는 추상 클래스가 없음 -> 그러다 보면 Ordered와 같은 Class의 구현이 매우 애매해짐
- Ordered의 하위 Class인 Number가 있음
  - 위에서 처럼 precedes의 함수의 파라미터 타입은 Ordered인데, Ordered 프로퍼티로 value가 있는지 알 수가 없음

### 3. Lost Type Relationships

```
class Ordered {  
    func precedes(other: Ordered) -> Bool { fatalError("implement me!") }  
}
```

```
class Label : Ordered { var text: String = "" ... }
```

```
class Number : Ordered {  
    var value: Double = 0  
    override func precedes(other: Ordered) -> Bool {  
        return value < (other as! Number).value  
    }  
}
```

- 올바른 타입에 도달하기 위해 다운 캐스팅을 해줘야 함
- 하지만 만약 other이 Label로 인식되었다면?
  - **Static type safety hole** 이란?

# as! ASubclass



A sign that a type relationship was lost  
Usually due to using classes for abstraction

코드에서 강제 다운캐스팅을 할 때 마다 일부 중요한 type 관계가 손실되었다는 신호

# A Better Abstraction Mechanism

Supports value types (and classes)

Supports static type relationships (and dynamic dispatch)

Non-monolithic

Supports retroactive modeling

Doesn't impose instance data on models

Doesn't impose initialization burdens on models

Makes clear what to implement

## 좀 더 좋은 추상화 메커니즘이 필요함

- 값 타입과 참조 타입 둘 다 지원
- 정적 타입 관계와 dynamic dispatch(동적 디스패치) 지원
- 단일적이지 않아야 함 -> 다루기 쉬워야 함
- 원본을 수정하지 않고 재활용 할 수 있어야 함(retroactive modeling)
- 모델에 인스턴스화된 데이터를 강요하지 않음
- 모델에서 초기화를 강요하지 않음
- 무엇을 구현할 지 확실히 함

## 이것이 Protocol이다

# Swift Is a Protocol-Oriented Programming Language

Swift는 객체 지향 프로그래밍에 훌륭하지만,  
For Loop 및 문자열 리터럴이 작동하는 방식, Standard Library에서  
generic에 대한 강조에 이르기까지  
Swift는 Protocol Oriented Programming 언어이다



# Starting Over with Protocols

```
class Ordered {  
    func precedes(other: Ordered) -> Bool { fatalError("implement me!") }  
}  
  
class Number : Ordered {  
    var value: Double = 0  
    override func precedes(other: Ordered) -> Bool {  
        return self.value < (other as! Number).value  
    }  
}
```

**해당 Orderd Class를 Protocol로 구현해보자**

# Starting Over with Protocols

error: protocol methods may not have bodies

```
protocol Ordered {  
    func precedes(other: Ordered) -> Bool { fatalError("implement me!") }  
}  
  
class Number : Ordered {  
    var value: Double = 0  
    override func precedes(other: Ordered) -> Bool {  
        return self.value < (other as! Number).value  
    }  
}
```

**Protocol에는 메서드 구현부가 필요 없음**

# Starting Over with Protocols

```
protocol Ordered {  
    func precedes(other: Ordered) -> Bool  
}  
  
class Number : Ordered {  
    var value: Double = 0  
    override func precedes(other: Ordered) -> Bool {  
        return self.value < (other as! Number).value  
    }  
}
```

error: method does not override any  
method from its superclass

기본 Class가 없기에 superClass도 없고 override도 필요없음  
또한 Class의 필요성이 없기에 Struct로 구현 가능

# Starting Over with Protocols

```
protocol Ordered {  
    func precedes(other: Ordered) -> Bool  
}  
  
struct Number : Ordered {  
    var value: Double = 0  
    func precedes(other: Ordered) -> Bool {  
        return self.value < (other as! Number).value  
    }  
}
```

현재 위 예제는 이전의 Class가 수행한 것과 동일한 역할을 하고 있음  
- 하지만 아직 static type safety hole을 해결하지 않음

# Starting Over with Protocols

```
protocol Ordered {  
    func precedes(other: Ordered) -> Bool  
}  
  
struct Number : Ordered {  
    var value: Double = 0  
    func precedes(other: Number) -> Bool {  
        return self.value < other.value  
    }  
}
```

protocol requires function 'precedes' with type '(Ordered) -> Bool'  
candidate has non-matching type '(Number) -> Bool'

강제 다운 캐스팅을 없애주고 other에 Number 타입을 넣어주면 에러 발생

# Starting Over with Protocols

```
protocol Ordered {  
    func precedes(other: Self) -> Bool  
}  
  
struct Number : Ordered {  
    var value: Double = 0  
    func precedes(other: Number) -> Bool {  
        return self.value < other.value  
    }  
}
```

"Self" requirement

Orderd Protocol 메서드에 파라미터 타입을 **Self**로 변경해주면 됨  
- 프로토콜을 준수한 type인 model type에 대한 표시자 -> **Self**

**Meta type에 대해 알아보자**

# Using Our Protocol

```
func binarySearch(sortedKeys: [Ordered], forKey k: Ordered) -> Int {  
    var lo = 0  
    var hi = sortedKeys.count  
    while hi > lo {  
        let mid = lo + (hi - lo) / 2  
        if sortedKeys[mid].precedes(k) { lo = mid + 1 }  
        else { hi = mid }  
    }  
    return lo  
}
```

**Ordered가 Class일 때의 함수**

– Protocol일 때 함수 파라미터로 **Self**를 추가해주기 전해도 작동은 했었음

# Using Our Protocol

```
func binarySearch(sortedKeys: [Ordered], forKey k: Ordered) -> Int {  
    var lo = 0  
    var hi = sortedKeys.count  
    while hi > lo {  
        let mid = lo + (hi - lo) / 2  
        if sortedKeys[mid].precedes(k) { lo = mid + 1 }  
        else { hi = mid }  
    }  
    return lo  
}
```

protocol 'Ordered' can only be used as a generic constraint because it has Self or associated type requirements

## Ordered가 Protocol일 때의 함수

- 여기서 [Ordered]는 이질적인 배열을 다룰 것이라는 주장
- 이 배열에서는 숫자와 레이블이 혼합되어 포함될 수 있음
- Ordered를 Protocol로 변경해주고 Self-requirement를 추가했으므로 컴파일러는 이것을 동질적인 배열로 만들도록 강제할 것임



# Using Our Protocol

```
func binarySearch<T : Ordered>(sortedKeys: [T], forKey k: T) -> Int {  
    var lo = 0  
    var hi = sortedKeys.count  
    while hi > lo {  
        let mid = lo + (hi - lo) / 2  
        if sortedKeys[mid].precedes(k) { lo = mid + 1 }  
        else { hi = mid }  
    }  
    return lo  
}
```

## Ordered가 Protocol일 때의 함수

- generic을 사용해 나는 단일 Ordered 타입 T의 동종 배열만 작업할 것이라고 말하고 있음
- 이는 배열을 동질화하는 것이 너무 제한적이거나, 기능이나, 유연성이 손실되는 것과 같다고 생각할 수 있음
- 하지만 Self 키워드를 Ordered 메서드에 파라미터로 넣음으로써 이질적인 경우를 처리할 일 자체가 없어졌음

# Two Worlds of Protocols

| Without Self Requirement                              | With Self Requirement                                   |
|---|---|
| <code>func precedes(other: Ordered) -&gt; Bool</code> | <code>func precedes(other: Self) -&gt; Bool</code>      |
| Usable as a type                                      | Only usable as a generic constraint                     |
| <code>func sort(inout a: [Ordered])</code>            | <code>func sort&lt;T : Ordered&gt;(inout a: [T])</code> |
| Think “heterogeneous”                                 | Think “homogeneous”                                     |
| Every model must deal with all others                 | Models are free from interaction                        |
| Dynamic dispatch                                      | Static dispatch   |
| Less optimizable                                      | More optimizable  |

| Self 요구사항이 없을 때   | Self 요구사항이 있을 때   |
|---|---|
| <code>func precedes(other: Ordered) -&gt; Bool</code>             | <code>Func precedes(other: Self) -&gt; Bool</code>                            |
| <b>type으로 사용 가능</b><br><code>func sort(inout a: [Ordered])</code> | <b>type으로 사용 불가능</b><br><code>func sort&lt;T: Sorted&gt;(inout a: [T])</code> |
| <b>이질적인 컬렉션 타입</b>  | <b>동질적인 컬렉션 타입</b>  |
| <b>모든 모델 타입간의 상호작용 가능</b>   | <b>다른 모델과 상호작용할 필요 없음</b>   |
| <b>동적 디스패치</b>  | <b>정적 디스패치</b>  |
| <b>부족한 최적화</b>  | <b>더 좋은 최적화</b>   |



클래스와 훨씬 덜 겹치는 다른 세계로 프로토콜이 이동됨

# A Primitive “Renderer”

```
struct Renderer {  
    func moveTo(p: CGPoint) { print("moveTo(\(p.x), \(p.y))") }  
  
    func lineTo(p: CGPoint) { print("lineTo(\(p.x), \(p.y))") }  
  
    func arcAt(center: CGPoint, radius: CGFloat,  
               startAngle: CGFloat, endAngle: CGFloat) {  
        print("arcAt(\(center), radius: \(radius),"  
              + " startAngle: \(startAngle), endAngle: \(endAngle))")  
    }  
}
```

**Protocol의 정적 측면이 어떻게 작동하는지 이해했지만,  
Protocol이 Class를 대체할 수 있는지 확신이 없음  
o ㅋ 예제하나 더 보여줌**

# Drawable

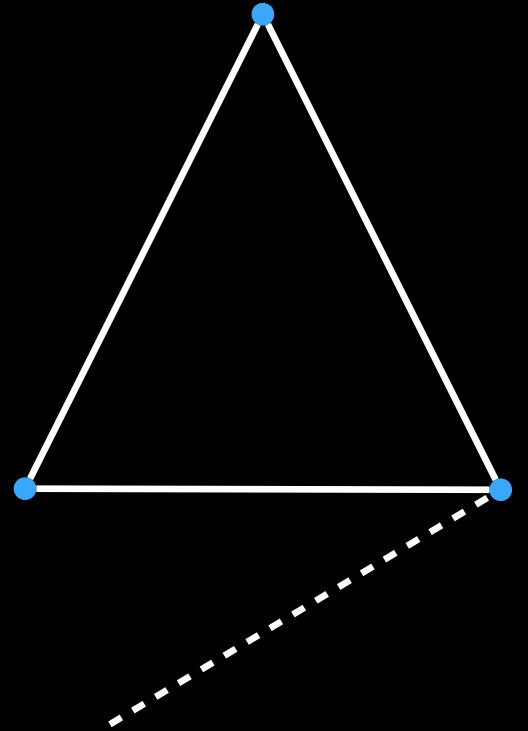
```
protocol Drawable {  
    func draw(renderer: Renderer)  
}
```

모든 그리기 요소에 대한 공통 인터페이스를 제공하기 위해 Drawable이라는 Protocol 구현

# Polygon



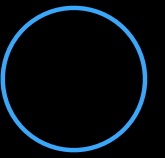
```
protocol Drawable {  
    func draw(renderer: Renderer)  
}  
  
struct Polygon : Drawable {  
    func draw(renderer: Renderer) {  
        renderer.moveTo(corners.last!)  
        for p in corners {  
            renderer.lineTo(p)  
        }  
    }  
    var corners: [CGPoint] = []  
}
```



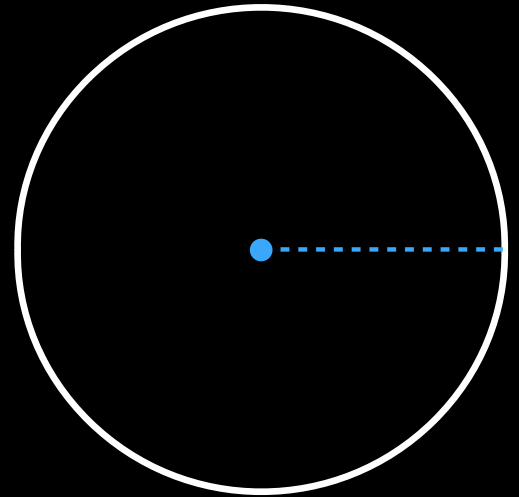
여기서 Polygon에 대해 가장 먼저 주목해야 할 점은 다른 값 타입으로 구현된 값 타입이라는 것

- corners는 CGPoint 배열을 포함하는 구조체
- 다각형을 그리기 위해 마지막 모서리로 이동한 다음 모든 모서리를 순환하면서 선을 그림

# Circle



```
protocol Drawable {  
    func draw(renderer: Renderer)  
}  
  
struct Circle : Drawable {  
    func draw(renderer: Renderer) {  
        renderer.arcAt(center, radius: radius,  
            startAngle: 0.0, endAngle: twoPi)  
    }  
    var center: CGPoint  
    var radius: CGFloat  
}
```



## Circle 또한 다른 값 타입으로 구현된 값 타입이라는 것

- 중심과 반지름을 프로퍼티로 가지고 있는 구조체
- 원을 그리기 위해 0에서 2파이까지 모든 방향을 쓸어넘기는 호를 그림

# Diagram

```
struct Diagram : Drawable {  
    func draw(renderer: Renderer) {  
        for f in elements {  
            f.draw(renderer)  
        }  
    }  
    var elements: [Drawable] = []  
}
```

**Drawable은 모두 값 타입으로 이루어져 있으므로 Diagram도 값 타입**

# Test It!

```
var circle = Circle(center:
    CGPoint(x: 187.5, y: 333.5),
    radius: 93.75)

var triangle = Polygon(corners: [
    CGPoint(x: 187.5, y: 427.25),
    CGPoint(x: 268.69, y: 286.625),
    CGPoint(x: 106.31, y: 286.625)])

var diagram = Diagram(elements: [circle, triangle])

diagram.draw(Renderer())
```

```
arcAt((187.5, 333.5),
    radius: 93.75,
    startAngle: 0.0,
    endAngle: 6.28)
```

```
moveTo(106.31, 286.625)
lineTo(187.5, 427.25)
lineTo(268.69, 286.625)
lineTo(106.31, 286.625)
```



# Test It!

```
var circle = Circle(center:
    CGPoint(x: 187.5,
    radius: 93.75)
```

```
var triangle = Poly
    CGPoint(x: 187.5,
    CGPoint(x: 268.6,
    CGPoint(x: 106.3
```

```
var diagram = Diag
```

```
diagram.draw(Renderer())
```

```
$ ./test
arcAt((187.5, 333.5),
    radius: 93.75, startAngle: 0.0,
    endAngle: 6.28318530717959)
moveTo(106.310118395209, 286.625)
lineTo(187.5, 427.25)
lineTo(268.689881604791, 286.625)
lineTo(106.310118395209, 286.625)
$
```



ㅇㅋㅇㅋ 알겠어 앱에서 실제로 사용할 거니까 CoreGraphics에서 사용할 수 있도록 다시 작성해볼게  
잠깐만 친구 프로토콜로 만들어봐!

# Renderer as a Protocol

```
protocol Renderer {  
    func moveTo(p: CGPoint)  
    func lineTo(p: CGPoint)  
    func arcAt(center: CGPoint, radius: CGFloat,  
               startAngle: CGFloat, endAngle: CGFloat)  
}
```

```
struct TestRenderer : Renderer {  
    func moveTo(p: CGPoint) { print("moveTo(\(p.x), \(p.y))" ) }  
    func lineTo(p: CGPoint) { print("lineTo(\(p.x), \(p.y))" ) }  
    func arcAt(center: CGPoint, radius: CGFloat,
```

**기존의 Struct에서 가지고 있던 메서드의 구현부를 없애주고,  
Test를 위해 만든 Struct에 Renderer를 채택  
TestRenderer 메서드 내부 기존의 Renderer Struct 메서드 내부 구  
현부로 재작성**

# Rendering with CoreGraphics

## Retroactive modeling

```
protocol Renderer {  
    func moveTo(p: CGPoint)  
    func lineTo(p: CGPoint)  
    func arcAt(center: CGPoint, radius: CGFloat,  
               startAngle: CGFloat, endAngle: CGFloat)  
}  
  
extension CGContext : Renderer {  
    func moveTo(p: CGPoint) { }  
    func lineTo(p: CGPoint) { }  
    func arcAt(center: CGPoint, radius: CGFloat,  
               startAngle: CGFloat, endAngle: CGFloat) { }  
}
```

나 CGContext에서 사용하고 싶어  
그러면 CGContext extension으로 구현하셈  
안에 구현부만 채워 넣어봐

# Rendering with CoreGraphics

## Retroactive modeling

```
extension CGContext : Renderer {  
    func moveTo(p: CGPoint) {  
        CGContextMoveToPoint(self, position.x, position.y)  
    }  
    func lineTo(p: CGPoint) {  
        CGContextAddLineToPoint(self, position.x, position.y)  
    }  
    func arcAt(center: CGPoint, radius: CGFloat,  
               startAngle: CGFloat, endAngle: CGFloat) {  
        let arc = CGPathCreateMutable()  
        CGPathAddArc(arc, nil, c.x, c.y, radius, startAngle, endAngle, true)  
        CGContextAddPath(self, arc)  
    }  
}
```

이게 되누..?

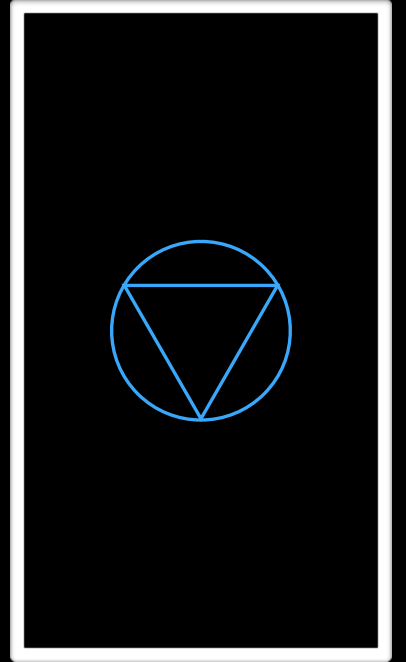
# Nested Diagram

```
var circle = Circle(center: CGPoint(x: 187.5, y: 333.5), radius: 93.75)
```

```
var triangle = Polygon(corners: [  
    CGPoint(x: 187.5, y: 427.25),  
    CGPoint(x: 268.69, y: 286.625),  
    CGPoint(x: 106.31, y: 286.625)])
```

```
var diagram = Diagram(elements: [circle, triangle])
```

```
diagram.elements.append(diagram)
```



`diagram.elements.append(diagram)`

이 함수가 무한 재귀로 가지 않는 이유를 알고 싶다면

Building Better Apps with Value Types in Swift 볼 것

# Protocols and Generics for Testability

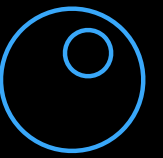
So much better than mocks

Disciplined decoupling is a beautiful thing.

**Protocol과 사물을 더 많이 분리할수록 모든 것이 더 테스트 가능해진다  
는 것을 발견**

- 이런 종류의 테스트는 mock으로 얻는 것과 비슷하지만 훨씬 더 좋음
- 모의 객체는 본질적으로 깨지기 쉬움
- 그 취약성 때문에 Swift의 강력한 정적 타입 시스템과 잘 어울리지 않음
- Protocol은 우리가 사용할 수 있는 원칙적인 인터페이스를 제공함
- 이 인터페이스는 언어에 의해 시행되지만, 여전히 필요한 모든 도구를 연결할 수 있는 연결고리를 제공함

# Bubble



```
struct Bubble : Drawable {  
    func draw(r: Renderer) {  
        r.arcAt(center, radius: radius, startAngle: 0, endAngle: twoPi)  
        r.arcAt(highlightCenter, radius: highlightRadius,  
                startAngle: 0, endAngle: twoPi)  
    }  
}  
  
struct Circle : Drawable {  
    func draw(r: Renderer) {  
        r.arcAt(center, radius: radius, startAngle: 0.0, endAngle: twoPi)  
    }  
}
```

이번엔 Bubble을 그려볼 것임

- Circle 구조체와 중복되는 코드가 있음
- Protocol을 사용하면 중복되는 코드를 줄여줄 수 있음

# Adding a Circle Primitive

```
protocol Renderer {  
    func moveTo(p: CGPoint)  
    func lineTo(p: CGPoint)  
    func circleAt(center: CGPoint, radius: CGFloat)  
    func arcAt(  
        center: CGPoint, radius: CGFloat, startAngle: CGFloat, endAngle: CGFloat)  
}
```

이렇게 Protocol 내부에 구현해줘도 다시 쓸 때 마다 구현을 또해줘야 되잖아



# Protocol Extensions

NEW

```
protocol Renderer {  
    func moveTo(p: CGPoint)  
    func lineTo(p: CGPoint)  
    func circleAt(center: CGPoint, radius: CGFloat)  
    func arcAt(  
        center: CGPoint, radius: CGFloat, startAngle: CGFloat, endAngle: CGFloat)  
}
```

```
extension Renderer {
```

```
    func circleAt(center: CGPoint, radius: CGFloat) {  
        arcAt(center, radius: radius, startAngle: 0, endAngle: twoPi)  
    }
```

```
}
```

Shared implementation



Protocol extension은 어파가 팔아먹음?

여기에다가 구현을 미리 해놔라

- Renderer의 모든 모델 간에 공유되는 구현이 구현된 것

# Protocol Extensions

Requirements create customization points

```
protocol Renderer {  
    func moveTo(p: CGPoint)  
    func lineTo(p: CGPoint)  
    func circleAt(center: CGPoint, radius: CGFloat)  
    func arcAt(  
        center: CGPoint, radius: CGFloat, startAngle: CGFloat, endAngle: CGFloat)  
}  
  
extension Renderer {  
    func circleAt(center: CGPoint, radius: CGFloat) { ... }  
    func rectangleAt(edges: CGRect) { ... }  
}
```

Renderer의 extension으로 circleAt()과 rectangleAt()을 구현해 주자

- circleAt()은 프로토콜 내부에 있지만,
- rectangleAt()은 extension에서만 구현되어있음


# Protocol Extensions

Requirements create customization points

```
extension Renderer {  
    func circleAt(center: CGPoint, radius: CGFloat) { ... }  
    func rectangleAt(edges: CGRect) { ... }  
}
```

```
extension TestRenderer : Renderer {  
    func circleAt(center: CGPoint, radius: CGFloat) { ... }  
    func rectangleAt(edges: CGRect) { ... }  
}
```

```
let r = TestRenderer()  
r.circleAt(origin, radius: 1);  
r.rectangleAt(edges);
```



**TestRenderer 인스턴스를 만들면 TestRenderer extension에서 구현한 메서드가 호출될 것**

# Protocol Extensions

Requirements create customization points

```
extension Renderer {  
    func circleAt(center: CGPoint, radius: CGFloat) { ... }  
    func rectangleAt(edges: CGRect) { ... }  
}
```

```
extension TestRenderer : Renderer {  
    func circleAt(center: CGPoint, radius: CGFloat) { ... }  
    func rectangleAt(edges: CGRect) { ... }  
}
```

```
let r: Renderer = TestRenderer()  
r.circleAt(origin, radius: 1);  
r.rectangleAt(edges);
```

하지만 TestRenderer 인스턴스의 타입을 Renderer로 명시해준다면?

- circleAt()은 Protocol 내부 요구사항이기에 TestRenderer의 circleAt()이 호출됨
- rectangleAt()은 Protocol 내부 요구사항이 아니기에 Renderer의 rectangleAt()이 호출됨

# More Protocol Extension Tricks

Scenes from the standard library and beyond

# More Protocol Extension Tricks

## Constrained extensions

```
extension CollectionType {  
    public func indexOf(element: Generator.Element) -> Index? {  
        for i in self.indices {  
            if self[i] == element {  
                return i  
            }  
        }  
        return nil  
    }  
}
```

binary operator '==' cannot be applied to  
two Generator.Element operands

이번엔 indexOf 메서드를 살펴보겠음 -> deprecated됨  
firstIndex(of:) 메서드를 사용

# More Protocol Extension Tricks

NEW

## Constrained extensions

```
extension CollectionType where Generator.Element : Equatable {  
    public func indexOf(element: Generator.Element) -> Index? {  
        for i in self.indices {  
            if self[i] == element {  
                return i  
            }  
        }  
        return nil  
    }  
}
```

**Collection의 element가 Equatable일 때 확장이 적용된다고 명시해줘야 비교를 허용할 수 있음**

# More Protocol Extension Tricks

## Retroactive adaptation

```
protocol Ordered {  
    func precedes(other: Self) -> Bool  
}  
func binarySearch<T : Ordered>(sortedKeys: [T], forKey k: T) -> Int { ... }
```

cannot invoke 'binarySearch' with an argument list of type '([Int], forKey: Int)'

```
let position = binarySearch([2, 3, 5, 7], forKey: 5)
```

제한된 확장의 간단한 예를 보았으므로 `binarySearch` 메서드를 다시 보자



# More Protocol Extension Tricks

## Retroactive adaptation

```
protocol Ordered {
    func precedes(other: Self) -> Bool
}

func binarySearch<T : Ordered>(sortedKeys: [T], forKey k: T) -> Int { ... }

extension Int : Ordered {
    func precedes(other: Int) -> Bool { return self < other }
}

extension String : Ordered {
    func precedes(other: String) -> Bool { return self < other }
}

let position = binarySearch(["2", "3", "5", "7"], forKey: "5")
```

**binarySearch 메서드에 Int를 사용하려면 Int extension에 Ordered를 채택해주고 구현해주면 됨**

**그러면 string일 때는? 또 extension으로 똑같이 적어줄꺼임?**

# More Protocol Extension Tricks

## Retroactive adaptation

```
protocol Ordered {
    func precedes(other: Self) -> Bool
}

func binarySearch<T : Ordered>(sortedKeys: [T], forKey k: T) -> Int { ... }

extension Comparable {
    func precedes(other: Self) -> Bool { return self < other }
}

extension Int : Ordered {}
extension String : Ordered {}

let position = binarySearch(["2", "3", "5", "7"], forKey: "5")
```

Comparable extension에 구현을 하면됨 0 0

# More Protocol Extension Tricks

## Retroactive adaptation

```
protocol Ordered {
  func precedes(other: Self) -> Bool
}

func binarySearch<T : Ordered>(sortedKeys: [T], forKey k: T) -> Int { ... }

extension Comparable {
  func precedes(other: Self) -> Bool { return self < other }
}

extension Int : Ordered {}
extension String : Ordered {}

let truth = 3.14.precedes(98.6)    // Compiles

let position = binarySearch([2.0, 3.0, 5.0, 7.0], forKey: 5.0)
```

cannot invoke 'binarySearch' with an argument list of type '([Double], forKey: Double)'

**Double에는 Ordered를 채택하지 않고 그냥 쓸래  
그러면 binarySearch에서는 할 수가 없잖아?**

# More Protocol Extension Tricks

## Retroactive adaptation

```
protocol Ordered {  
    func precedes(other: Self) -> Bool  
}  
  
func binarySearch<T : Ordered>(sortedKeys: [T], forKey k: T) -> Int { ... }  
  
extension Ordered where Self : Comparable {  
    func precedes(other: Self) -> Bool { return self < other }  
}  
  
extension Int : Ordered {}  
extension String : Ordered {}  
let truth = 3.14.precedes(98.6)
```

'Double' does not have a member named 'precedes'

그러면 제한된 확장을 사용하여 선택적으로 사용하자  
Double extension은 제거해줬으니 Double에서 직접 precedes 메서드는  
사용 못함

# Make All Value Types Equatable

```
func == (lhs: Polygon, rhs: Polygon) -> Bool {  
    return lhs.corners == rhs.corners  
}  
extension Polygon : Equatable {}  
  
func == (lhs: Circle, rhs: Circle) -> Bool {  
    return lhs.center == rhs.center  
        && lhs.radius == rhs.radius  
}  
extension Circle : Equatable {}
```

모든 값 타입은 동일하게 만들어라

```
struct Diagram : Drawable {  
    func draw(renderer: Renderer) { ... }  
    var elements: [Drawable] = []  
}
```

```
func == (lhs: Diagram, rhs: Diagram) -> Bool {  
    return lhs.elements == rhs.elements  
}
```

binary operator '==' cannot be applied to two [Drawable] operands.

```
func == (lhs: Diagram, rhs: Diagram) -> Bool {  
    return lhs.elements.count == rhs.elements.count  
        && !zip(lhs.elements, rhs.elements).contains { $0 != $1 }  
}
```

binary operator '!=' cannot be applied to two Drawable operands.

'==' 연산자 사용 불가!

```
struct Diagram : Drawable {  
    func draw(renderer: Renderer) { ... }  
    var elements: [Drawable] = []  
}  
  
func == (lhs: Diagram, rhs: Diagram) -> Bool {  
    return lhs.elements.count == rhs.elements.count  
        && !zip(lhs.elements, rhs.elements).contains { $0 != $1 }  
}  
  
protocol Drawable : Equatable {  
    func draw()  
}
```



## Protocol Equatable을 만들어서 == 연산자를 만든다!

a Self-requirement puts Drawable squarely in the homogenous, statically dispatched world. But Diagram really needs a heterogeneous array of Drawables. So we can put polygons and circles in the same Diagram.

So Drawable has to stay in the heterogeneous, dynamically dispatched world.

# Bridge-Building

```
struct Diagram : Drawable {  
    func draw(renderer: Renderer) { ... }  
    var elements: [Drawable] = []  
}  
  
func == (lhs: Diagram, rhs: Diagram) -> Bool {  
    return lhs.elements.count == rhs.elements.count  
        && !zip(lhs.elements, rhs.elements).contains { !$0.isEqualTo($1) }  
}  
  
protocol Drawable {  
    func isEqualTo(other: Drawable) -> Bool  
    func draw()  
}
```

```
extension Drawable where Self: Equatable {  
    func isEqualTo(other: Drawable) -> Bool {  
        if let other as? Self {  
            return self == other  
        }  
        return false  
    }  
}
```



# When to Use Classes

They do have their place...

You *want* implicit sharing when

- Copying or comparing instances doesn't make sense (e.g., Window)
- Instance lifetime is tied to external effects (e.g., TemporaryFile)
- Instances are just “sinks”—write-only conduits to external state (e.g., CGContext)

```
final class StringRenderer : Renderer {  
    var result: String  
    ...  
}
```

Still a protocol!

**Class를 사용해야 할 때 -> 그들도 사용이 되어야 할 때가 있음**

예를 들어 값 유형의 기본연산이 의미가 없는경우와 같이 암시적 공유를 원할때

- 인스턴스를 복사하거나 비교하는 것이 말이 안될 때
- 인스턴스의 수명이 외부 효과와 관련된 경우
  - 이 중 일부는 값이 컴파일러에 의해 자유롭게 생성 및 소멸되며 가능한 최적화하려고 하기 때문
  - 이 안정적인 정체성을 갖는 것은 참조 유형이므로 외부 엔티티에 해당하는 것을 만들려는 경우 참조 유형으로 만드는 것이 좋음 -> 외부 엔티티 firebase
- 추상화의 인스턴스가 Sinks인 경우 -> 다른 객체 또는 메서드에서 들어오는 이벤트를 수신하도록 설계된 클래스(프로토콜) 또는 함수 ex) Delegate

# When to Use Classes

They do have their place...

Don't fight the system

- If a framework expects you to subclass, or to pass an object, do!

On the other hand, be circumspect

- Nothing in software should grow too large
- When factoring something out of a class, consider a non-class

## 시스템과 싸우지 마라

- 만약 프레임워크가 상속하길 원한다면 상속해라

## 하지만 신중해라

- 프로그램의 어떠한 것도 너무 커지면 안됨
- 리팩토링 할 때는 클래스를 사용안할 수 있는 방향으로 생각해라

# Summary

Protocols > Superclasses

Protocol extensions = magic (almost)

Go see the value types talk on Friday!

Eat your vegetables

Be like Crusty...