# Building Better Apps with Value

데릭

보리입니다 데스

바드

ppt 참고 민소네 블로그

보리

# Roadmap

Reference semantics

Immutability

Value semantics

Value types in practice

Mixing value types and reference types

참조 의미론
불변성
값 의미론
값 타입에 대한 연습
값타입과 참조타입을 섞어쓰기

참조 의미론에 대해 알아보고
그리고 불변성을 참조 의미론의 해결책으로써 알아봄
이를 위한 값 의미론과 스위프트에서 어떻게 사용되는지 보고
참조와 값타입의 mixing에 대해서 이야기할것

# Reference Semantics

참조 의미론
스위프트에서 참조 의미론이란 클래스를 정의하는 과정이다.

# A Temperature Class

```
class Temperature {
  var celsius: Double = 0
  var fahrenheit: Double {
    get { return celsius * 9 / 5 + 32 }
    set { celsius = (newValue - 32) * 5 / 9 }
  }
}
```

여기 간단한 Temperature 클래스가 있다.
celcius에 온도를 저장하고,
계산 속성인 fahrenheit(이하 화씨)를 갖는다.

간단한, 추상화된 버전의 온도 클래스임.

# Using Our Temperature Class

```
let home = House()
let temp = Temperature()
temp.fahrenheit = 75
home.thermostat.temperature = temp

temp.fahrenheit = 425
home.oven.temperature = temp
home.oven.bake()
```

이 온도 클래스를 활용한 간단한 예시를 보자

1. 하우스 인스턴스와 온도 인스턴스를 생성함
2. 온도 인스턴스를 화씨 75도로 맞춤
3. 하우스의 온도조절기의 온도 프로퍼티에 온도 인스턴스를 주입

이제 저녁시간이 와서 연어를 구워야 하니 다른 온도값이 필요

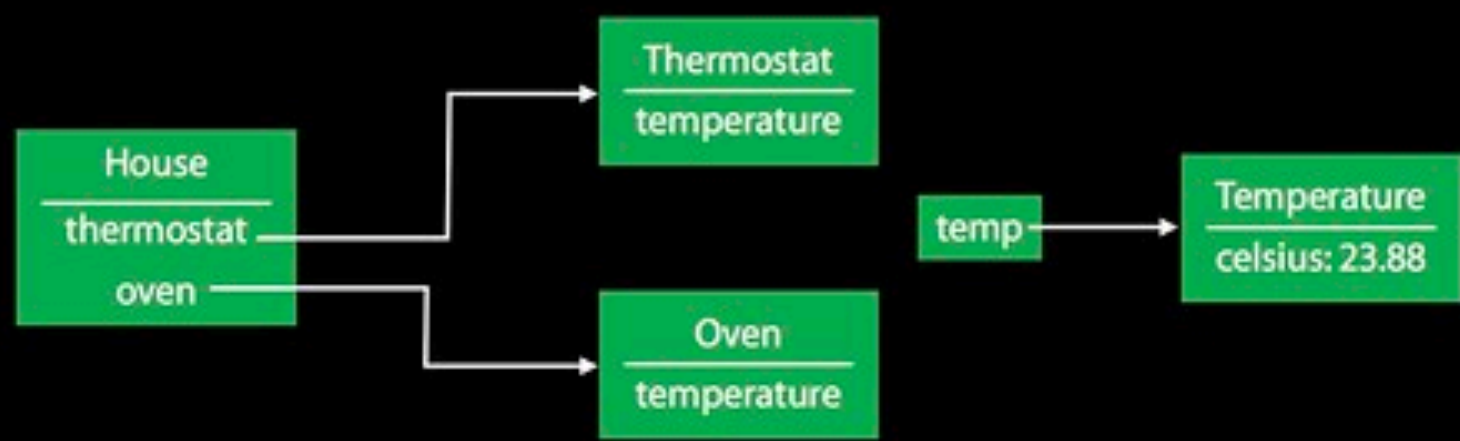
4. 온도 인스턴스를 425 화씨로 맞춤
5. 집의 오븐의 온도프로퍼티에 같은 온도 인스턴스 주입
6. 오븐 베이킹 시작

Why Is It So Hot in Here?

당연히 더워짐

# Unintended Sharing

```
let home = House()
let temp = Temperature()
temp.fahrenheit = 75
```

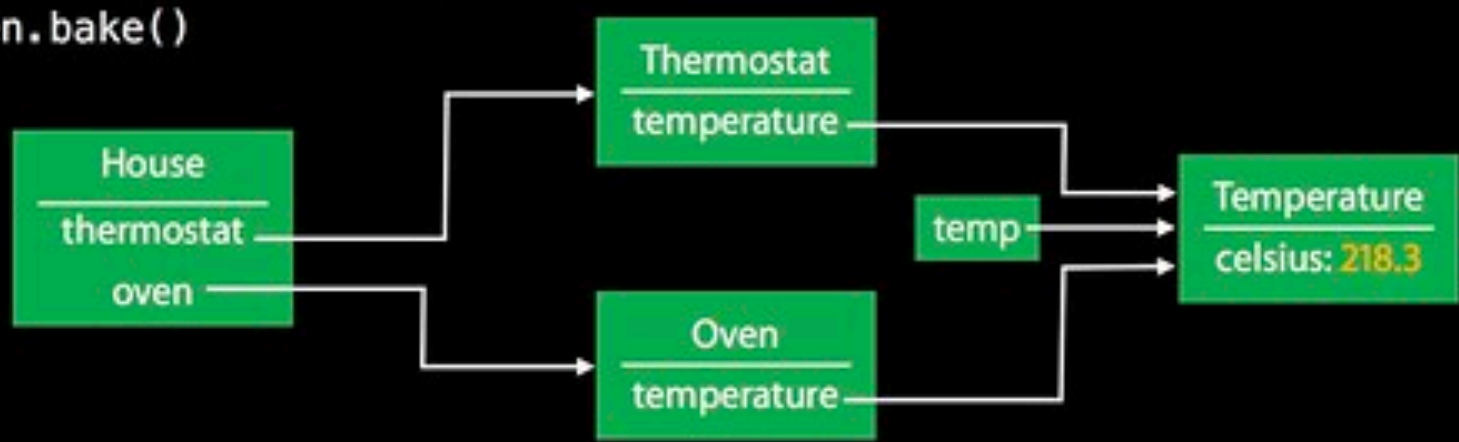



우리는 왜 이렇게 된지 안다.
바로 의도치 않은 공유를 한것.

이 객체 그래프를 보자
하우스에는 각각 온도조절기와 오븐이 있고,
이들은 모두 temperature를 가질수 있음

# Unintended Sharing

```
let home = House()
let temp = Temperature()
temp.fahrenheit = 75
home.thermostat.temperature = temp

temp.fahrenheit = 425
home.oven.temperature = temp
home.oven.bake()
```




다만 오븐의 온도를 조정하기 위해 화씨 425도로 바꾼 순간
이미 게임은 끝남. 연결해봤자 우리는 모두 잃었음(의미 없다는 얘기같음)

뭐가 틀린거지?

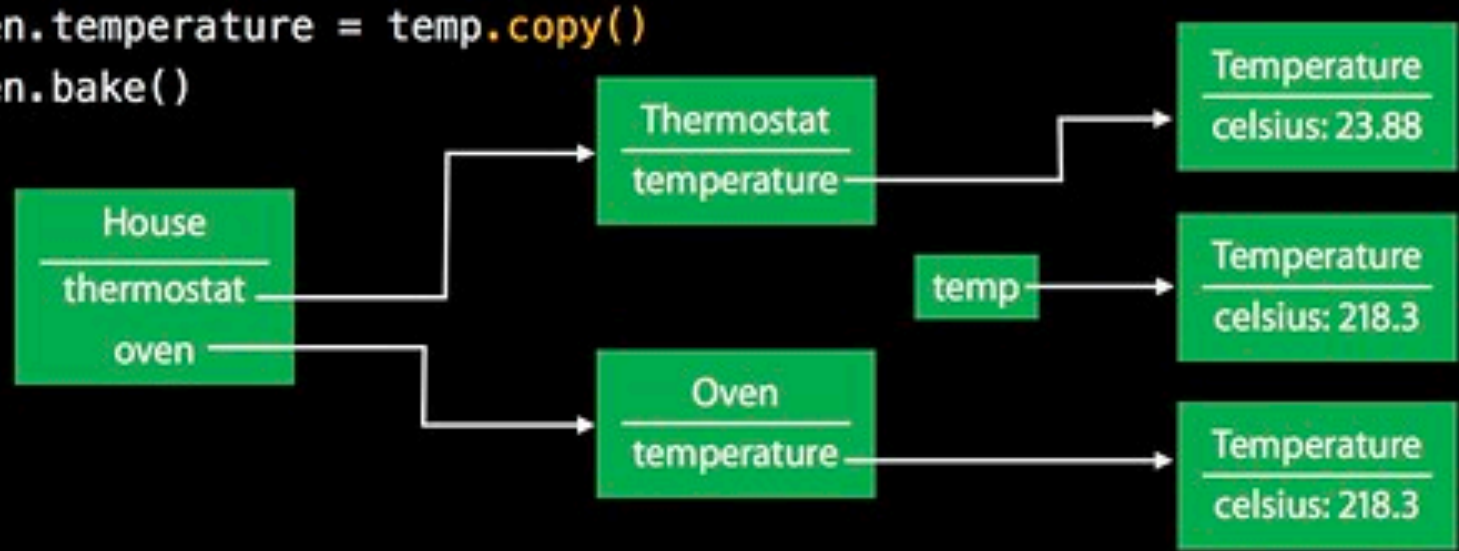
참조 의미론의 세상에서 공유를 막으려면 복사를 해야합니다.

# Copy When You Need It

오케이 한번 바꿔보자

# Manual Copying

```
let home = House()
let temp = Temperature()
temp.fahrenheit = 75
home.thermostat.temperature = temp.copy()

temp.fahrenheit = 425
home.oven.temperature = temp.copy()
home.oven.bake()
```




temp인스턴스를 만들고 온도조절기에서 사용할 온도인 75도 설정
그리고 그 인스턴스를 그대로 주입하는게 아니라 **복사**함

그 뒤 다시 화씨 425도로 바꾸고 이 또한 **복사**함

기술적으로 마지막 사본은 필요 없다고함
추가 사본을 만들기 위해 힙에 메모리를 할당하는 시간이 비효율적이라고 함.

# Defensive Copying

```
class Oven {
  var _temperature: Temperature = Temperature(celsius: 0)

  var temperature: Temperature {
    get { return _temperature }
    set { _temperature = newValue.copy() }
  }
}
```

**이해가 안되는 부분 1**

우리는 방어 복사로 이것을 참조하고,
여기서 우리는 필요한 것을 알고 있으므로 복사할 수 없지만,
만약에 지금 또는 언젠가 필요하면, 이 문제를 디버깅하는데 정말로 어렵습니다.
그래서 .copy로 언제든지 oven에서 temperature를 어딘가에 할당했다는 것을
너무 쉽게 잊습니다.
대신, oven에다 올바른 방법으로 구울 것입니다.

이 말이 의미하는게
1. 디버깅이 일단 어려워진다
2. 기존 temperature를 복사하므로 복사 temperature가 결국 어딘가에 할당되는
건데
이 할당 사실 자체를 잊기 쉽다.

이 두가지를 말하고 싶은건지 잘 모르겟음
특히 첫줄의 표현이...

우리가 필요로 하기 때문에가 아니라 우리가 필요로 하는 경우에 문제가 발생할
수 있다..?

# Defensive Copying

```
class Thermostat {
  var _temperature: Temperature = Temperature(celsius: 0)

  var temperature: Temperature {
    get { return _temperature }
    set { _temperature = newValue.copy() }
  }
}
```

Copy를 위해 여기에도 같은 작업을 해준다.

이제 아주 많은 boilplate가 있고 이를 복사하여 붙여넣는다.

그리고 곧 언어 특징에 대해 문을 두드리고 질문한다,.

# Defensive Copying in Cocoa and Objective-C

Cocoa[Touch] requires copying throughout

- **`NSCopying`** codifies copying an object
- **`NSString`**, **`NSArray`**, **`NSDictionary`**, **`NSURLRequest`**, etc. all require copying

Defensive copying pervades Cocoa[Touch] and Objective-C

- **`NSDictionary`** calls **`-copy`** on its keys
- Property **`copy`** attribute provides defensive copying on assignment

먼저 NSCopying이라는 프로토콜을 알고 넘어가야 이해가능할듯.
NSCopying : 개체가 자신의 기능적 복사본을 제공하기 위해 채택하는 프로토콜

복사의 의미는 클래스마다 다를 수 있지만, 복사본은 만들어질 당시의 원본과 동일한 값을 가진 기능적으로 독립적인 객체
NSObject를 상속하는 모든 객체에 대해 정의되며 copy()로 쓰면됨.
이게 방어적 복사라고하는듯.

NSCopying은 객체의 복사본을 코드화 한다.
이는 NSString, NSArray등 NSObject를 상속받는 애들은 모두 알고있따.
---
이제 복사가 필요한 시스템이 있다고 보자. NSDictionary는 dict에 있는 key를 복사함

# Is Immutability the Answer?

참조 의미론이 이렇게 있고, 여기에 약간의 변화(mutation)이 있을 때 문제가 발생할 수 있음.

불변 데이터 구조와 참조 의미론의 세상으로 이동해야 할지도 모른다(?)

# Eliminating Mutation

Functional programming languages have reference semantics with immutability

Eliminates many problems caused by reference semantics with mutation

- No worries about unintended side effects

Several notable disadvantages

- Can lead to awkward interfaces
- Does not map efficiently to the machine model

함수형 프로그래밍언어는 항상 불변성과 참조의미론을 가져옴
mutation(변형)이 있는 참조의미론이 발생시키는 많은 문제를 제거할 수 있음.
> 예상하지 못하는 사이드 이펙트를 걱정할 필요 없음.

몇가지 단점
- 어색한 인터페이스로 이어질 수 있음.
- 기계모델의 매핑에 효율적이지 않을 수 있음 ->> 불변의 알고리즈

# An Immutable Temperature Class

```
class Temperature {
  let celsius: Double = 0
  var fahrenheit: Double { return celsius * 9 / 5 + 32 }

  init(celsius: Double) { self.celsius = celsius }
  init(fahrenheit: Double) { self.celsius = (fahrenheit - 32) * 5 / 9 }
}
```

불변의 알고리즘에서 효율적인 수준까지 대응하는것은 어렵다.

temperature 클래스를 불변하게 만들어 안전하게 만들고자 한다.
- 저장 프로퍼티인 celsius를 var에서 let으로 변경
- fahrenheit 계산프로퍼티에서 set을 제거

이제 이 temperature클래스는 상태를 변경할 수 없음.

# Awkward Immutable Interfaces

With mutability

```
home.oven.temperature.fahrenheit += 10.0
```

Without mutability

```
let temp = home.oven.temperature
home.oven.temperature = Temperature(fahrenheit: temp.fahrenheit + 10.0)
```

우리는 temperature를 불변하게 만듬.
가변적인것과 불변적인것의 차이를 보자.

위의 가변 인터페이스는 직관적임. 누가봐도 화씨를 10도 올리는 행위임.
그러나 불변 인터페이스를 갖는 경우 아래처럼
1. 현재 오븐의 온도를 저장
2. 오븐에 새로운 온도 인스턴스를 생성하여 주입.

이러한 동작은 어색함. 코드가 더 생겼고, 힙에 새로운 객체를 할당하는 낭비를 저지름

아예 새로운 오븐 인스턴스를 할당했기 때문에 불변성 수용이 없음.

이렇게 불변 참조를 사용하면 온도를 바꿀때 마다
새로운 집, 새로운 오븐, 새로운 온도를 만들어 넣어야만 함.

어색함 이자체가.

# Sieve of Eratosthenes

```
func primes(n: Int) -> [Int] {
  var numbers = [Int](2..<n)
  for i in 0..<n-2 {
    guard let prime = numbers[i] where prime > 0 else { continue }
    for multiple in stride(from: 2 * prime-2, to: n-2, by: prime) {
      numbers[multiple] = 0
    }
  }
  return numbers.filter { $0 > 0 }
}
```

에라토스 테네스의 체라는 고대 알고리즘을 예시로 들어보자.
2부터 n까지 소수가 담긴 배열을 구할 예정.
루프를 돌면서 차례대로 모든 배수들을 제거함.
결과중 지워지지 않은 부분이 소수임.

# Sieve of Eratosthenes

| | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 3 | 0 | 5 | 0 | 7 | 0 | 0 | 0 | 11 | 0 | 13 | 0 | 0 | 0 | 17 | 0 | 19 | 0 |

```
func primes(n: Int) -> [Int] {
  var numbers = [Int](2..<n)
  for i in 0..<n-2 {
    guard let prime = numbers[i] where prime > 0 else { continue }
    for multiple in stride(from: 2 * prime-2, to: n-2, by: prime) {
      numbers[multiple] = 0
    }
  }
  return numbers.filter { $0 > 0 }
}
```

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 5 | 7 | 11 | 13 | 17 | 19 |

기본적으로 배열을 변경하는걸 전제로하는 알고리즘이다.
물론 변경없이도 가능은 하다.

# Functional Sieve of Eratosthenes

Haskell:

```
primes = sieve [2..]
sieve [] = []
sieve (p : xs) = p : sieve [x | x <- xs, x `mod` p > 0]
```

Swift:

```
func sieve(numbers: [Int]) -> [Int] {
  if numbers.isEmpty { return [] }
  let p = numbers[0]
  return [p] + sieve(numbers[1..<numbers.count].filter { $0 % p > 0 })
}
```

Haskell이라는 언어와 이를 스위프트로 변경한 코드
haskell은 함수형 언어임.

# Functional Sieve of Eratosthenes

Swift:

```
func sieve(numbers: [Int]) -> [Int] {
  if numbers.isEmpty { return [] }
  let p = numbers[0]
  return [p] + sieve(numbers[1..<numbers.count].filter { $0 % p > 0 })
}
```

2에서 20까지를 여기서는 예시로 듬
배열의 인덱스 0자리를 분리,
filter를 통해서 2의 배수를 모조리 제거한 나머지만 배열로 만들어 다시 재귀적으로
그 다음 값을 구하도록 만듬.

앞의 코드와는 성능적으로 다르기 때문에 같은 알고리즘이라고 볼 수 없음.

앞에껀 2의 배수에 0을 할당해서 6이 0이되어도
3의 배수를 볼 때 6이 있던 자리를 한번 더 보기 때문.
각 인덱스 자리별로 확인하는 작업이 현저하게 줄어듬.

# Immutability in Cocoa[Touch]

Cocoa[Touch] has a number of immutable classes

- `NSDate`, `NSURL`, `UIImage`, `NSNumber`, etc.
- Improved safety (no need to use copy)

Downsides to immutability

```
NSURL *url = [[NSURL alloc] initWithString: NSHomeDirectory()];
NSString *component;
while ((component = getNextSubdir()) {
  url = [url URLByAppendingPathComponent: component];
}
```

Cocoa 관련 프레임워크에서 불변성을 우리는 많이 볼 수 있음
NSDate, NSURL등등...

이것들은 모두 불변타입임. > 이 불변 타입을 갖는것은 안전성을 향상시킴.
왜? 복사에 대해서 딱히 걱정안해도되니까. 복사를 걱정안한다는건 의도치 않은 부작용을 갖는, 의도치않은 공유를 걱정안한다는 의미

[불변의 단점]
불변타입으로 작업할 때 단점을 볼 수 있음.
예시의 옵젝씨 코드. Home디렉토리부터 일부 디렉토리에 도달하는 연속 경로 요소를 추가로 시작하는 NSURL을 만들것.

참조 의미론세상에서는 변경없이 하기를 원함. 그래서 NSURL을 만듬(얘는 불변이니)
그러나 이 코드는 url에 매번 새로운 NSURL 객체를 만들어 넣어줌.
또 매 루프마다 모든 문자열을 복사함

# Immutability in Cocoa[Touch]

Cocoa[Touch] has a number of immutable classes

- `NSDate`, `NSURL`, `UIImage`, `NSNumber`, etc.
- Improved safety (no need to use copy)

Downsides to immutability

```
NSArray<NSString *> *array = [NSArray arrayWithObject: NSHomeDirectory()];
NSString *component;
while ((component = getNextSubdir()) {
  array = [array arrayByAddingObject: component];
}
url = [NSURL fileURLWithPathComponents: array];
```

이제 NSURL 대신에 NSArray로 변경했음.
요소들을 NSArray에 모으고 나서 fileURLWithPathComponents를 사용해야함.

# Thoughtful Mutability in Cocoa[Touch]

## You'd miss it if it were gone

Cocoa[Touch] has a number of immutable classes

- `NSDate`, `NSURL`, `UIImage`, `NSNumber`, etc.
- Improved safety (no need to use copy)

Thoughtful mutability

```
NSMutableArray<NSString *> *array = [NSMutableArray array];
[array addObject: NSHomeDirectory()];
NSString *component;
while ((component = getNextSubdir()) {
  [array addObject: component];
}
url = [NSURL fileURLWithPathComponents: array];
```

Mutablearray 즉 가변 어레이에 쭉 더해놓고 마지막에 불변으로 바꾸는 방식이 좋다.

불변성은 좋고, 참조 의미론 세상을 추론하기 더 쉽게 만들어준다.
하지만 우리는 완전히 불변으로 할 수는없으며 하려고하면 미칠수도있음.

데릭

# Value Semantics

Value Semantics는 Reference Semantics와 다르게 접근한다.

# Variables Are Logically Distinct

## Integers are value types

Mutating one variable of some value type will never affect a different variable

```
var a: Int = 17
var b = a
assert(a == b)

b += 25

print("a = \(a), b = \(b)") // a = 17, b = 42
```

공유되는 것이 문제!
간단한 아이디어
- 두 변수를 생성
- 위의 두 변수의 값은 논리적으로 별개이다.
- 정수 A를 정수 B에다 복사한다.
- copy 처리
- b += 25

# Variables Are Logically Distinct

## CGPoints are value types

Mutating one variable of some value type will never affect a different variable

```
var a: CGPoint = CGPoint(x: 3, y: 5)
var b = a
assert(a == b)
b.x = 17
print("a = \(a), b = \(b)") // a = (x = 3, y = 5), b = (x = 17, y = 5)
```

CGPoint 예제
- b에 a copy
- b가 변하여도 a는 아무런 영향 받지 않음
- Value Semantics는 기본적인 타입인 숫자와 숫자를 포함하는 구조체, 그리고 확장을 하는 것

# Variables Are Logically Distinct

## Strings are value types

Mutating one variable of some value type will never affect a different variable

```
var a: String = "Hello"
var b = a
assert(a == b)
b.extend(" WWDC!")
print("a = \(a), b = \(b)") // a = Hello, b = Hello WWDC!
```

Swift에서 문자열은 값타입
A를 만들고 b에 a를 할당하는 것이 a에는 영향이 없다 → 이게 값 타입이기 때문?

# Variables Are Logically Distinct

## Arrays are value types

Mutating one variable of some value type will never affect a different variable

```
var a: [Int] = [1, 2, 3, 4, 5]
var b = a
assert(a == b)
b[2] = 17
print("a = \(a), b = \(b)") // a = [1, 2, 3, 4, 5], b = [1, 2, 17, 4, 5]
```

배열은 String처럼 동작하지 않나?
- 완전히 다른 값이다.
- 무엇이? 메모리 주소값이?
- copy on write

# Variables Are Logically Distinct

## Dictionaries are value types

Mutating one variable of some value type will never affect a different variable

```
var a: [Int : String] = [1 : "uno", 2 : "dos"]
var b = a
assert(a == b)

b[2] = "due"

print("a = \(a), b = \(b)") // a = [1 : "uno", 2 : "dos"],
                            // b = [1 : "uno", 2 : "due"]
```

- Value Semantics – Collection
  Dictionary 에 value semantics인 것을 넣고 다시 value semantics를 얻는다?

# Value Types Compose

All of Swift's "fundamental" types are value types

- `Int`, `Double`, `String`, ...

All of Swift's collections are value types

- `Array`, `Set`, `Dictionary`, ...

Swift tuples, structs, and enums that contain value types are value types

Value Semantics에서는 모든 추상화를 쉽게 만들 수 있다?
- Swift에서 모든 기본 타입은 – integer, double, string, character, 기타 등등 – 모두 값 타입이기 때문
- 기본 타입은 두 변수가 논리적으로 구분된다. ===
- Array, Set, Dictionary로 만들어지는 모든 컬렉션은 값 타입으로 주어지면 값 타입이다.
- tuple, struct, enum의 언어 추상은 값 타입을 넣었을 때, 값 타입으로 얻습니다

# Value Types Are Distinguished by Value

Equality is established by value of a variable

- Not its identity
- Not how we arrived at the value

```
var a: Int = 5
var b: Int = 2 + 3
assert(a == b)
```

- Value Type에서 중요한 부분 두개의 값, 값 타입의 두 변수가 같을 때의 개념
  두 변수는 같은 값을 가진다.
  identity는 상관 없다는 것이 중요하다
  많은 copy 값을 가질 수 있기 때문
  저장된 실제 값이 중요하다
  값을 얻는 방법이 중요한 것은 아니다
  위의 a, b의 결과 값은 같다.

# Value Types Are Distinguished by Value

Equality is established by value of a variable

- Not its identity
- Not how we arrived at the value

```
var a: CGPoint = CGPoint(x: 3, y: 5)
var b: CGPoint = CGPoint(x: 1, y: 3)
b.x += 2
b.y += 2
assert(a == b)
```

- CGPoint도 결과 값이 같다.

# Value Types Are Distinguished by Value

Equality is established by value of a variable

- Not its identity
- Not how we arrived at the value

```
var a: String = "Hello WWDC!"
var b: String = "Hello"
b += " "
b += "WWDC!"
assert(a == b)
```

- 같은 값임을 나타내는 것은 문자열을 어떻게 얻었는지나 identity가 중요한 것이 아니다.

# Value Types Are Distinguished by Value

Equality is established by value of a variable

- Not its identity
- Not how we arrived at the value

```
var a: [Int] = [1, 2, 3]
var b: [Int] = [3, 2, 1].sort(<)
assert(a == b)
```

- 두 배열은 같은 값을 가지므로 두 배열은 같다.

# Equatable

## Value types should implement Equatable

```
protocol Equatable {
    /// Reflexive - `x == x` is `true`
    /// Symmetric - `x == y` then `y == x`
    /// Transitive - `x == y` and `y == z` then `x == z`
   func ==(lhs: Self, rhs: Self) -> Bool
}

var a = …
var b = a
assert(a == b)
assert(b == a)
var c = b
assert(c == a)
```

- 값 타입을 만들 때, Equatable protocol에 적합한지가 매우 중요하다
  모든 값 타입은 Equatable이어야 하기 때문
  비교하기 위한 연산자(==)가 있음을 의미함
  반사적이고(reflexive), 대칭적이고(symmetric), 과도적이(transitive) 될 필요가 있다

# Implementing Equatable

```
protocol Equatable {
    /// Reflexive - `x == x` is `true`
    /// Symmetric - `x == y` then `y == x`
    /// Transitive - `x == y` and `y == z` then `x == z`
   func ==(lhs: Self, rhs: Self) -> Bool
}

extension CGPoint: Equatable { }

func ==(lhs: CGPoint, rhs: CGPoint) -> Bool {
  return lhs.x == rhs.x && lhs.y == rhs.y
}
```

- CGPoint extension에 Equatable protocol을 채택하여 연산자(==)를 구현하고, 다른 값 타입을 구성할 때, 연산자(==)를 사용해야 한다.

# Value Semantics Temperature

```
struct Temperature: Equatable {
  var celsius: Double = 0
  var fahrenheit: Double {
    get { return celsius * 9 / 5 + 32 }
    set { celsius = (newValue - 32) * 5 / 9 }
  }
}

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

- Temperature class를 structure로 만들었다.
  celsius는 var로 변경하여 Value Semantics가 되었다.
  연산자(==)를 채탁하여 사용한다.

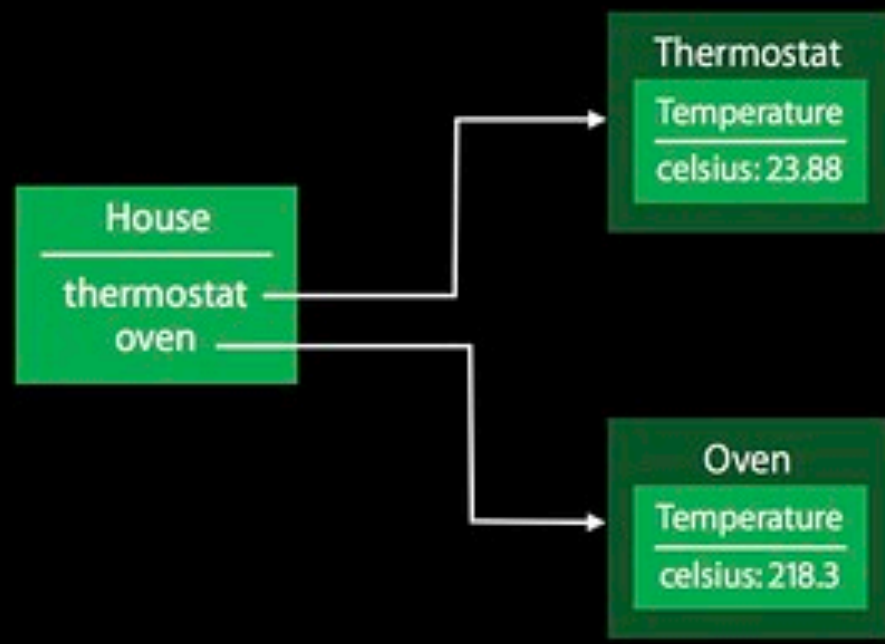
# Using Value Semantics Temperature

```
let home = House()
let temp = Temperature()
temp.fahrenheit = 75
home.thermostat.temperature = temp

temp.fahrenheit = 425
home.oven.temperature = temp
home.oven.bake()
```

- home, tepm를 생성한다
  temperature에 화씨 75도를 할당한다
  컴파일러는 여기에서 멈춘다??
  temp 프로퍼티를 변경하는데 temp 프로퍼티는 let(상수)이다!
  - 변경할 수 없음

# Using Value Semantics Temperature

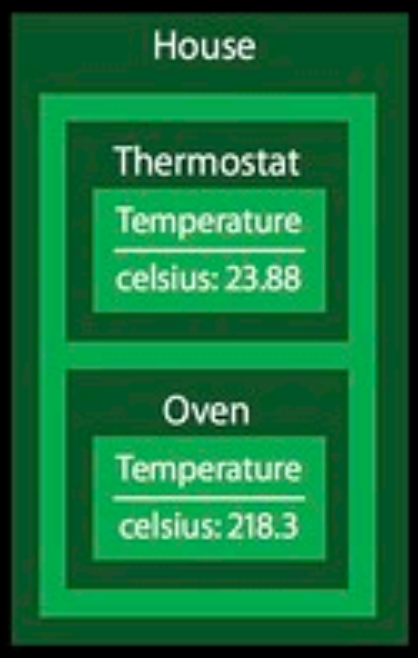
```
let home = House()
var temp = Temperature()
temp.fahrenheit = 75
home.thermostat.temperature = temp

temp.fahrenheit = 425
home.oven.temperature = temp
home.oven.bake()
```




- 상수 temp 프로퍼티의 값을 변경하기 위해 var 변수로 변경하여 사용
  house는 oven의 thermostat를 가리킨다
  thermostat와 oven 둘 다 각자의 temperature를 가진다.
  - 절대 공유되지 않음
  - structure로 inlining 되도록 하여 메모리 사용 성능을 높였다.

# Using Value Semantics ~~Temperature~~ Everywhere

```
var home = House()
var temp = Temperature()
temp.fahrenheit = 75
home.thermostat.temperature = temp

temp.fahrenheit = 425
home.oven.temperature = temp
home.oven.bake()
```




- house, thermostat, oven 모두 structure
  Value Semantics

# Mutation When You Want It

## But not when you don't

`let` means "the value will never change"

```
let numbers = [1, 2, 3, 4, 5]
```

`var` means you can update the value without affecting any other values

```
var strings = [String]()
for x in numbers {
  strings.append(String(x))
}
```

- Swift에서 let은 값이 변경되지 않는다를 의미한다.

# Freedom from Race Conditions

```
var numbers = [1, 2, 3, 4, 5]
scheduler.processNumbersAsynchronously(numbers)
for i in 0..<numbers.count { numbers[i] = numbers[i] * i }
scheduler.processNumbersAsynchronously(numbers)
```

- 스레드 경계를 가로질러 지나가는 값 타입을 사용할 때(변수가 여러 쓰레드에서 사용됨을 의미), 이들 타입은 race condition으로부터 해방된다.

  Reference Semantics에서 위의 상황은 Race Condition이다.
  그러나 ! Value Semantics에서는 논리적인 copy를 얻어 같은 배열 끼리 부딪히지 않는다.

# Copies Are Cheap

## Constant time

Copying a low-level, fundamental type is constant time

- `Int`, `Double`, etc.

Copying a struct, enum, or tuple of value types is constant time

- `CGPoint`, etc.

Extensible data structures use copy-on-write

- Copying involves a fixed number of reference-counting operations
- `String`, `Array`, `Set`, `Dictionary`, etc.

- Value Semantics의 중요 부분중 하나는 복사 비용이 싸다는 것이다.
Constant Time이 싸다는 것을 의미한다.
기본 타입이 있을 때, low-level인 integer, double, float 등의 타입의 복사 비용은 싸다. → 여러 바이트를 복사하는 것이다.
CGPoint는 두 개의 CGFloat으로 구성되어 있다. struct, enum, tuple은 field에 고정된 숫자가 있고, Constant Time에 각각을 복사한다. 모든 것은 Constant Time에 복사한다.
고정 길이인 것에 좋다. 확장 가능한 String, Array, Set, Dictionary 등은 Swift에서 Copy-write으로 다룬다.
- 저렴하게 copy를 만든다.
copy 값을 얻기 위해 copy-write은 Reference-counting operation의 고정된 숫자 일부이다. 변경하는 곳에서 var로 변경하고, copy를 만들고 copy로 작업을 한다. 해당 값들은 값을 공유하고 있지만 논리적 공유가 아니므로, 여전히 다른 값이다.

바드

# Value Types in Practice

## Conceptualize an Example

앞에서는 값 타입을 참조 타입과 비교하여 어떻게 작업하는지 알아보았다
이번에는 값 유형을 사용하여 실제 예제를 만들어보자

# A Diagram Made of Value Types

Circle

Polygon

Diagram

두개의 다른 값타입인 간단한 Diagram을 예제로 만들어보자
Circle과 Polygon을 만들어 볼것임

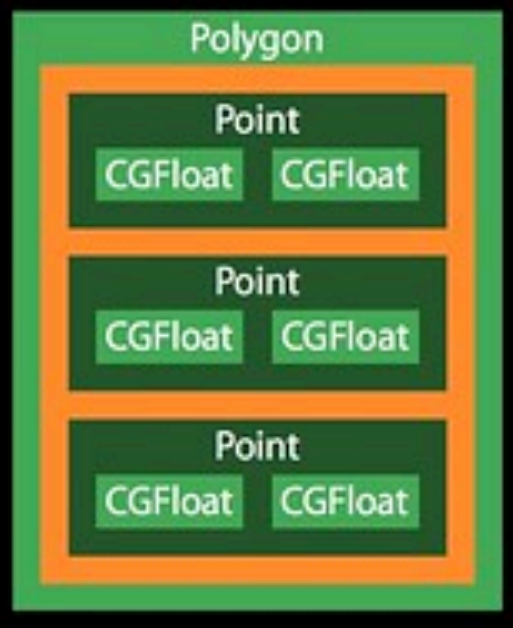
# Circle

```
struct Circle: Equatable {
    var center: CGPoint
    var radius: Double

    init(center: CGPoint, radius: Double) {
        self.center = center
        self.radius = radius
    }
}

func ==(lhs: Circle, rhs: Circle) {
  return lhs.center == rhs.center && lhs.radius == rhs.radius
}
```

center, radius 변수를 가지고 있음
두개의 값 타입은 Swift Standard Library임
= 연산자, == 연산자 그리고 ===연산자로 비교를 하도록 구현을 해볼 것임
standard Library를 만든 후 Library에서 나온 간단한 타입으로 구성한 이후 사용해야 함

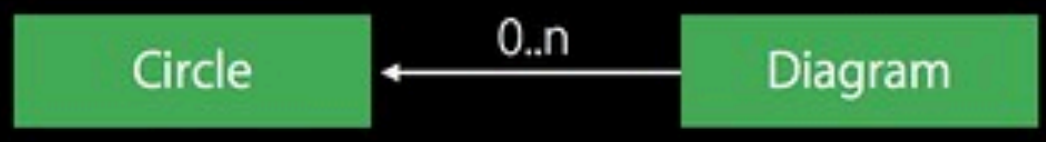
# Polygon

```
struct Polygon: Equatable {
    var corners: [CGPoint] = []
}

func ==(lhs: Polygon, rhs: Polygon) {
  return lhs.corners == rhs.corners
}
```




Polygon은 corners라는 CGPoint 배열이 있고 값 타입임
배열이 값 타입이므로 == 연산자를 통해 비교할 수 있음

# Diagram Contains Circles



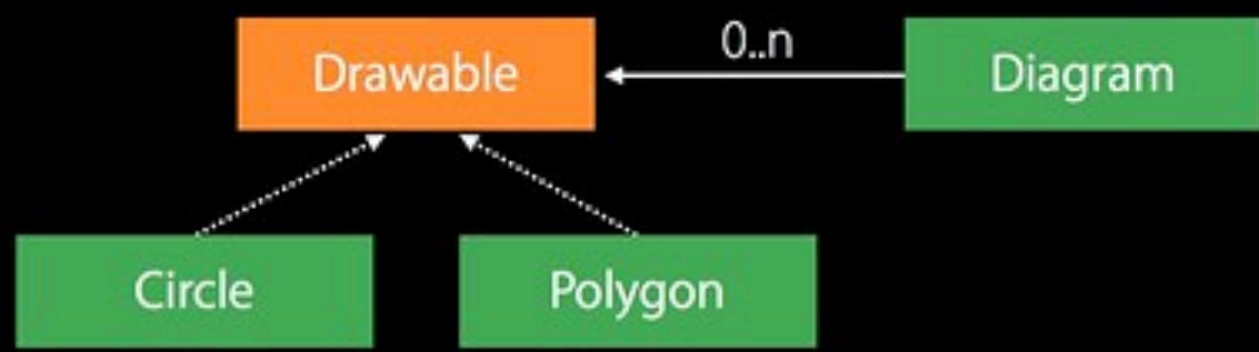

이제 Diagram에 Circle과 Polygon을 넣어 볼 것임

# Diagram Contains Polygons

Diagram에 두 타입의 배열을 넣어주고 싶음

# Diagram Contains Polygons

## Protocols can abstract over value types




두 개의 타입을 포함하는 배열을 만들기 위해서는 Protocol을 사용해줘야 됨
Drawable이라는 Protocol을 만들어서 Circle과 Polygon을 Drawble을 채택시켜 줌

# The Drawable Protocol

```
protocol Drawable {
  func draw()
}

extension Polygon: Drawable {
  func draw() {
    let ctx = UIGraphicsGetCurrentContext()
    CGContextMoveToPoint(ctx, corners.last!.x corners.last!.y)
    for point in corners {
      CGContextAddLineToPoint(ctx, point.x, point.y)
    }
    CGContextClosePath(ctx)
    CGContextStrokePath(ctx)
  }
}
```

Drawable Protocol은 draw 메서드를 가지고 있음
Polygon의 extension을 활용해 draw 메서드를 구현해주고,
Core Graphics를 호출해 polygon을 그림

# The Drawable Protocol

```
protocol Drawable {
  func draw()
}

extension Circle: Drawable {
  func draw() {
    let arc = CGPathCreateMutable()
    CGPathAddArc(arc, nil, center.x, center.y, radius, 0, 2 * π, true)
    CGContextAddPath(ctx, arc)
    CGContextStrokePath(ctx)
  }
}
```

Circle도 Polygon과 똑같이 extension으로 구현을 해줄 것

# Creating the Diagram

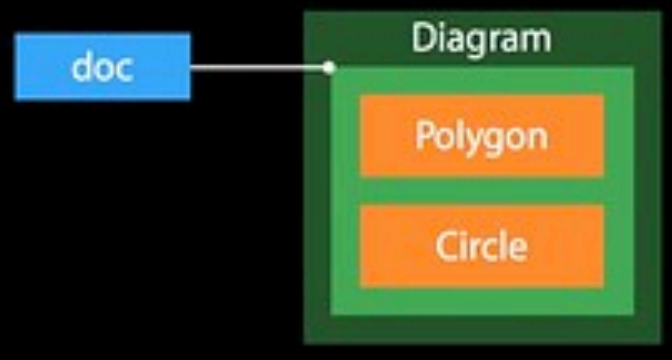
```
struct Diagram {
  var items: [Drawable] = []

  mutating func addItem(item: Drawable) {
    items.append(item)
  }

  func draw() {
    for item in items {
      item.draw()
    }
  }
}
```

Diargram으로 돌아와서 items는 Drawable의 배열을 가지고 있음
items에 값을 더해주는 메서드를 만들어줌
이는 자기자신을 변경하기 때문에 mutating 키워드를 표시해주어야 함

draw 메서드는 items 배열을 반복하고 각 item에서 draw 메서드를 호출함

# Adding Items

```
var doc = Diagram()
doc.addItem(Polygon())
doc.addItem(Circle())
```

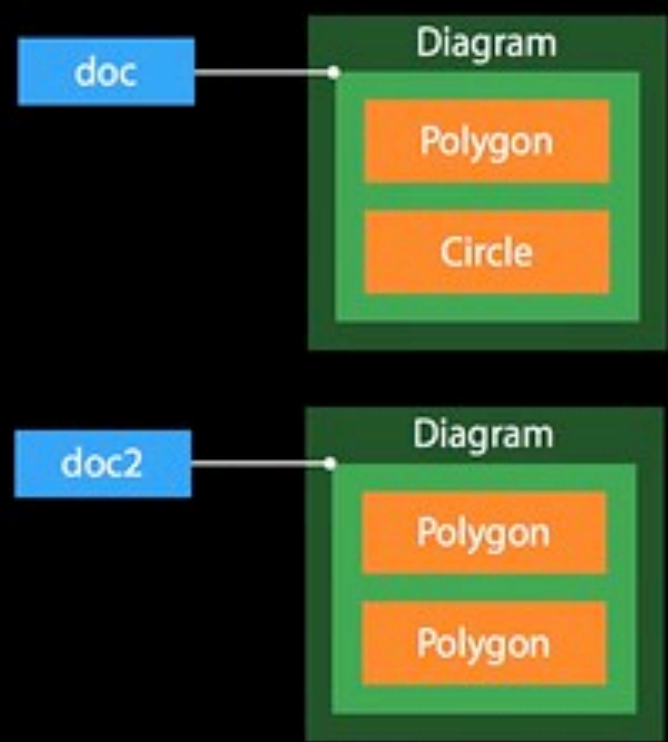



위에서 doc이라는 Diagram 변수를 만들어주고
doc에 Polygon을 추가해주고
그 다음 Circle을 추가해줌
이제 doc의 item 배열은 두개의 Drawble을 가지고 있는 배열이 됐음

# Copied on Assignment

## Heterogeneous arrays have value semantics, too!

```
var doc = Diagram()
doc.addItem(Polygon())
doc.addItem(Circle())
var doc2 = doc
doc2.items[1] = Polygon(corners: points)
```




이번에는 doc과 같은 doc2를 변수로 생성해줌
-> 논리적으로 완전히 새로운 인스턴스
doc2는 doc으로부터 논리적으로 완전히 분리가 됨
이제 doc2를 변경해도 doc에 아무런 영향을 끼치지 않음
doc2의 Circle을 Polygon으로 변경함
배열은 값 의미론을 가지며 heterogeneous임(value semantic이 정확히 뭐지)
배열 내부에 Polygon과 Cicle을 값으로 가지고 있음

# Making Diagram Equatable

```
extension Diagram: Equatable { }

func ==(lhs: Diagram, rhs: Diagram) {
  return lhs.items == rhs.items
}
```

```
error: binary operator '==' cannot be
applied to two [Drawable] operands
```

우리는 Diagram 구조체를 Equtable을 채택하도록 만들 것임
하지만 구현해도 양쪽의 값이 같은지에 대한 == 연산자가 없다고 말함
이에 대해서는 Protocol Oriented Programming 영상을 보면 됨
우리는 일단 value semantics에 대해 초점을 맞출 것임

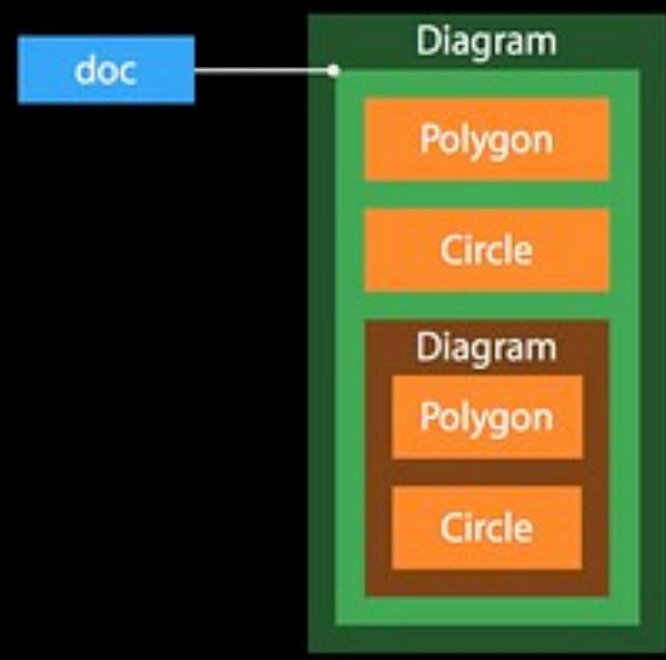
# If It Quacks Like a Duck…

```
protocol Drawable {
  func draw()
}

struct Diagram: Drawable {
  var items: [Drawable] = []

  func draw() { … }
}
```

Drawable은 draw라는 메서드를 가지고 있으며 Diagram도 draw 메서드가 있음
Diagram을 Drawble을 채택시켜봄
그러면 Diagram은 오리처럼 꽥꽥대고 실제로 오리임? -> 유머임?

# Diagram as a Drawable

```
var doc = Diagram()
doc.addItem(Polygon())
doc.addItem(Circle())
doc.addItem(doc)

func draw() {
  for item in items {
    item.draw()
  }
}
```




여기서부터 흥미로움
새로운 Diagram을 만들 수 있고 기존 Diagram에 추가할 수도 있음
세 개의 다른 타입이지만, 모두 배열 내부에 포함이 되었음

draw 메서드를 살펴보자
참조 의미론이었다면, draw 메서드는 무한 재귀였을 것임
Diagram에서 draw메서드를 호출하면 items 배열에서 다시 draw 메서드를 호출하고 무한 재귀가 일어날 것

하지만 우리는 값 타입을 사용하고 있음
Diagram에 doc을 추가해도 이는 값 타입이기 때문에 인스턴스가 완전히 나뉘고 구분이 됨
따라서 무한 재귀가 없고 두개의 polygon과 두개의 circle을 가지게 됨

# Mixing Value Types and Reference Types

단지 값 타입으로 객체 트리를 만드는 방법에 대해 이야기를 했으니
값 타입과 참조 타입을 섞는 방법을 이야기해보자

# Reference Types Often Contain Value Types

Value types generally used for "primitive" data of objects

```
class Button : Control {
  var label: String
  var enabled: Bool
  // …
}
```

objective-C에서는 참조 타입 안쪽에 매번 primitive data 타입을 넣는데 사용이 됨 -> ???

In objective-C you're used to putting primitive data types inside your reference types all the time

this is how we build things in objective-C

# A Value Type Can Contain a Reference

Copies of the value type will share the reference

```
struct ButtonWrapper {
  var button: Button
}
```

Maintaining value semantics requires special considerations

- How do we cope with mutation of the referenced object?
- How does the reference identity affect equality?

값 타입을 만든다면 참조가 안에 있어도 값 의미론을 여전히 유지하는지 확인하길 원함
확인하기 위해서는 두개의 다른 값이 참조가 있어 같은 것을 가르키고 있다는 것을 어떻게 확인하나

# Immutable References

```
struct Image : Drawable {
  var topLeft: CGPoint
  var image: UIImage
}
```

불변 클래스인 UIImage로 간단한 예시를 보자
Drawable을 채택하고 있고 UIImage에 참조를 가지고 있음

# Immutable References

```
struct Image : Drawable {
  var topLeft: CGPoint
  var image: UIImage
}
```

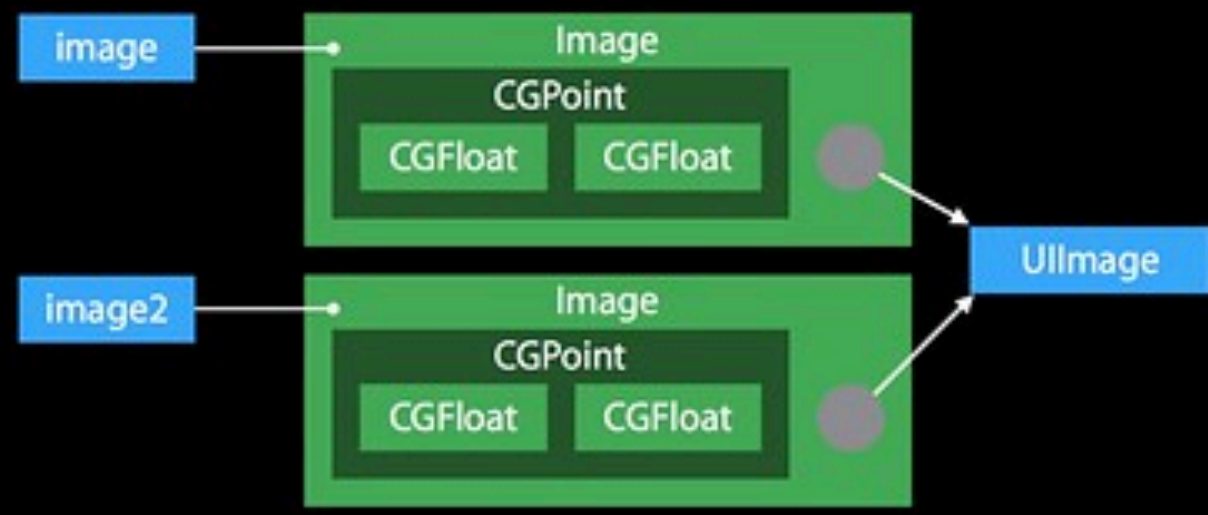
```
var image = Image(topLeft: CGPoint(x: 0, y: 0),
                  image: UIImage(imageNamed:"San Francisco")!)
```

샌프라시스코의 이미지를 가지고 Image 인스턴스를 만들어줌

# Immutable References Are Okay!

## Mutation of the referenced object does not occur

```
struct Image : Drawable {
  var topLeft: CGPoint
  var image: UIImage
}
```



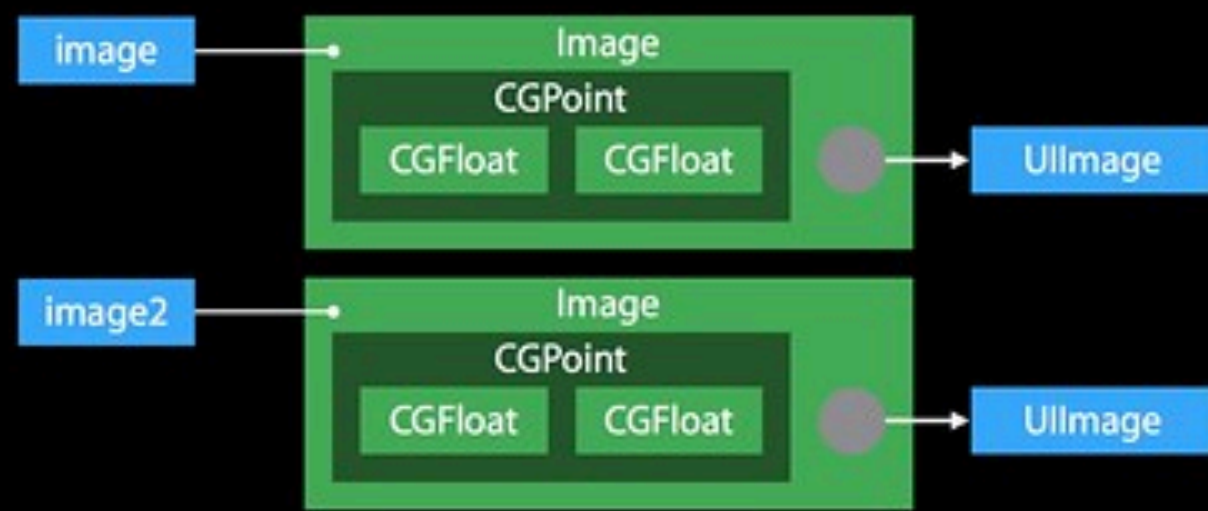

```
var image = Image(topLeft: CGPoint(x: 0, y: 0),
                  image: UIImage(imageNamed:"San Francisco”)!)
var image2 = image
```

그리고 image2를 만든다면, image와 image2는 같은 객체를 가리키고 있을 것이라는 문제가 될 것 같지만 문제가 되지 않음
UIImage는 불변이기에 image가 변경된 것에 대해 image2는 걱정할 필요없음

# Immutable References and Equatable

## Reference identity is not enough

```
struct Image : Drawable {
  var topLeft: CGPoint
  var image: UIImage
}
```



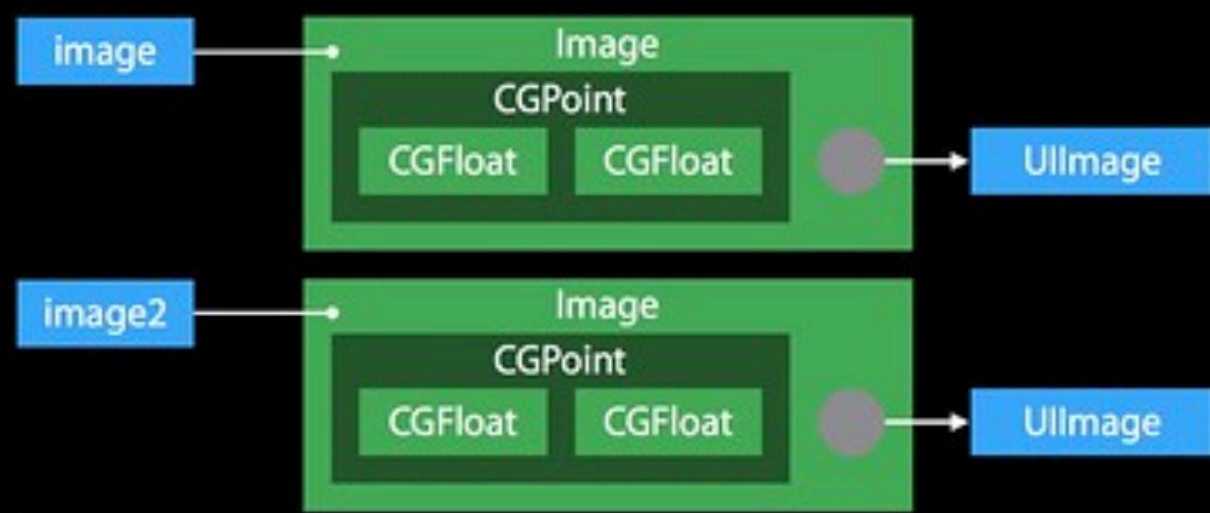

```
extension Image : Equatable { }
func ==(lhs: Image, rhs: Image) -> Bool {
  return lhs.topLeft == rhs.topLeft && lhs.image === rhs.image
}
```

확실히 비교하기 위해 참조가 같은 것을 가르키고 있는지 === 연산자를 통해 확인해 볼 수 있음
현재 예제에서는 괜찮겠지만, 같은 기본 bitmap을 사용한 두 개의 UIImage를 만들 때 발생하는 것도 생각해줘야 함
이 경우에는 참조를 비교한 이후에는 비교하지 않을 것임
그러면 두 이미지는 같지 않다고 잘못 말할 것임

# Immutable References and Equatable

## Use deep equality comparisons

```
struct Image : Drawable {
  var topLeft: CGPoint
  var image: UIImage
}
```




```
extension Image : Equatable { }
func ==(lhs: Image, rhs: Image) -> Bool {
  return lhs.topLeft == rhs.topLeft && lhs.image.isEqual(rhs.image)
}
```

대신 우리는 NSObject로부터 상속받은 isEqual 메서드를 사용하여
비교를 할 것이고 참조 타입이 같은 객체인지 아닌지 확인할 수 있음

# References to Mutable Objects

```
struct BezierPath: Drawable {
  var path = UIBezierPath()

  var isEmpty: Bool {
    return path.empty
  }

  func addLineToPoint(point: CGPoint) {
    path.addLineToPoint(point)
  }
}
```

이번에는 변경가능한 객체에 대해 이야기를 해보자
Drawable을 채택한 BeierPath가 있음
전체 구현은 변경가능한 참조 타입인 UIBezierPath로 구현되어 있음
읽을 경우에 empty를 수행할 때는 어떠한 변경도 하지 않기에 어떠한 인스턴스도 엉망이 되지 않음

# References to Mutable Objects

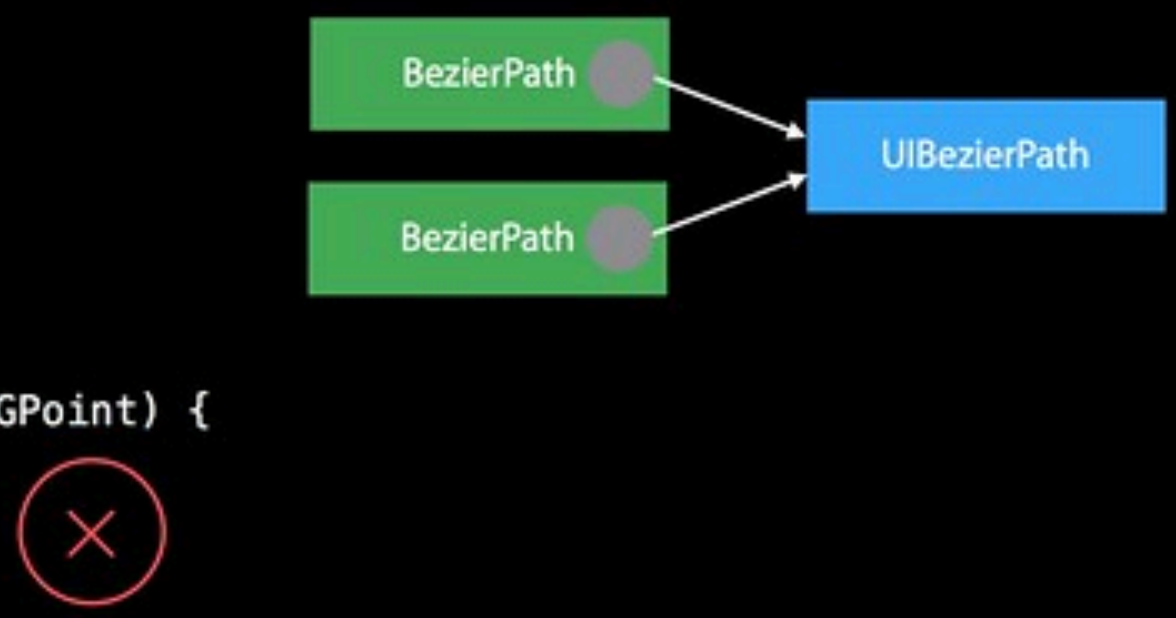
```
struct BezierPath: Drawable {
  var path = UIBezierPath()

  var isEmpty: Bool {
    return path.empty
  }

  func addLineToPoint(point: CGPoint) {
    path.addLineToPoint(point)
  }
}
```

하지만 addLineToPoint 메서드를 사용할 때, 두개의 BeizerPath를 가르키고 있다면, 문제가 생길 것임
심지어 mutating 키워드도 없는데 이는 path가 참조 유형이기 때문

# References to Mutable Objects

## Unexpected mutation

```
struct BezierPath: Drawable {
  var path = UIBezierPath()

  var isEmpty: Bool {
    return path.empty
  }

  func addLineToPoint(point: CGPoint) {
    path.addLineToPoint(point)
  }
}
```




BezierPath인스턴스가 두개이고 하나의 인스턴스에 대해 변경을 해주면 좋지 않은 상황이 일어날 것임
value semantic 또한 유지되지 않을 것임

# Copy-on-Write

Unrestricted mutation of referenced objects breaks value semantics

Separate non-mutating operations from mutating ones

- Non-mutating operations are always safe
- Mutating operations must first copy

이를 고치기 위해서는 Copy-on-write을 사용해 주면 됨
path를 작성하기 전에 복사본을 만들었는지 확인

# Copy-on-Write in Action

```
struct BezierPath: Drawable {
  private var _path = UIBezierPath()

  var pathForReading: UIBezierPath {
    return _path
  }

  var pathForWriting: UIBezierPath {
    mutating get {
      _path = _path.copy() as! UIBezierPath
      return _path
    }
  }
}
```

이를 하기 위해서 BezierPath의 새로운 방법 두가지를 소개함
1.
path 인스턴스를 private으로 만듬
그리고 읽기 전용 연산 프로퍼티를 구현하고 private 변수를 반환
자기 자신을 변경하기에 mutating 키워드 작성
기존 path의 새로운 복사본으로 갈아끼우도록 설정

# Copy-on-Write in Action

```
extension BezierPath {
  var isEmpty: Bool {
    return pathForReading.empty
  }

  func addLineToPoint(point: CGPoint) {
    pathForWriting.addLineToPoint(point)
  }
}
```

```
error: cannot read 'pathForWriting' because 'self' is not mutable
```

이제 읽기 복사본과 쓰기 복사본 방법이 있음
이를 반영하여 구현을 바꿈
isEmpty 메서드에서 읽기 복사본을 호출하고,
mutating 메서드에서 쓰기를 위한 path를 호출할 것
여기서 자기 자신을 바꾸기에 mutating 키워드 추가해줘야 됨

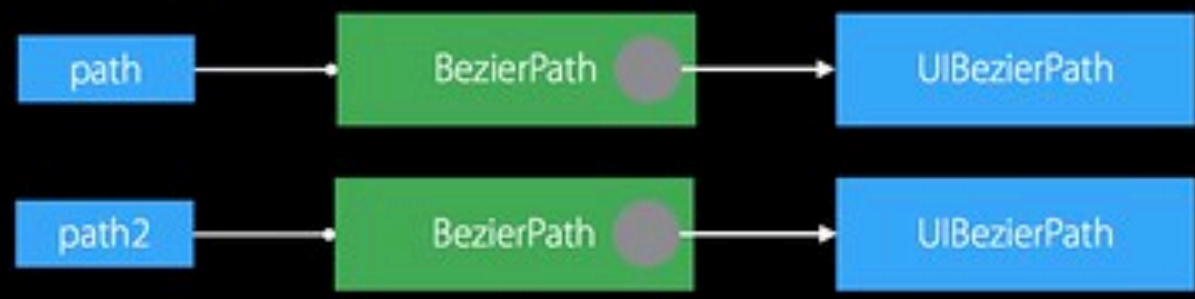
# Copy-on-Write in Action

```
extension BezierPath {
  var isEmpty: Bool {
    return pathForReading.empty
  }

  mutating func addLineToPoint(point: CGPoint) {
    pathForWriting.addLineToPoint(point)
  }
}
```

컴파일러의 도움을 받아 잘 해결할 수 있음

# Bezier Path

## Copy-on-write




```
var path = BezierPath()
var path2 = path
if path.empty { print("Path is empty") }
path.addLineToPoint(CGPoint(x: 10, y: 20))
```

다이어그램에서 path를 살펴보면 path2는 path로 만듬
물론 path2로 부터 읽을 수 있음
path2를 쓰려고 할 때, 새로운 BezierPath를 만든 이후에, path2는 변경이 일어났음을 인지하지 못함
path2 is none the wiser that a mutation has happened
so I’m not going to introduce some unexpected mutation behind pass back or path2

# Forming a Path from a Polygon

## Copies every time through the loop!

```
extension Polygon {
  var path: BezierPath {
    var result = BezierPath()
    result.moveToPoint(corners.last!)
    for point in corners {
      result.addLineToPoint(point)
    }
    return result
  }
}
```

Polygon 타입이 있고 BezierPath를 돌려줄 메서드르 추가하여 extension으로 구현
BezierPath를 만들고 point에 선을 추가해줌
단점은 매번 호출로 addLineToPoint 메서드가 복사된다는 것
수행할 수도 있고 안그럴 수도 있음

# Forming a Path from a Polygon

## Use the mutable reference type (carefully)

```
extension Polygon {
  var path: BezierPath {
    var result = UIBezierPath()
    result.moveToPoint(corners.last!)
    for point in corners {
      result.addLineToPoint(point)
    }
    return BezierPath(path: result)
  }
}
```

BezierPath 인스턴스 대신 UIBezierPath의 인스턴스를 만들어줌
변경 가능한 참조 타입이 있는 경우, 해당 참조 타입의 인스턴스를 만들어서
반복문을 돌린 후
값 타입의 인스턴스에 반환을 해줌

# Uniquely Referenced Swift Objects

```
struct MyWrapper {
  var _object: SomeSwiftObject
  var objectForWriting: SomeSwiftObject {
    mutating get {
      if !isUniquelyReferencedNonObjC(&_object)) {
         _object = _object.copy()
      }
      return _object
    }
  }
}
```

The standard library value types uses this throughout

Swift에는 유일하게 참조되었는지 아는 훌륭한 기능이 있고 이 이점을 활용할 수 있음
BezierPath에서 봤던 것과 유사한 구조이고, 유일하게 참조된 프로퍼티가 있다는 사실을 사용할 수 있고 뭔가 유일하게 참조됐다는 사실을 암
참조 타입이 유일하게 참조되었다는 것을 안다면 복사본 만드는 것을 피할 수 있음
Standard Library는 이 특징을 곳곳에 사용하고 이를 이용하여 성능 최적화를 많이 함

# Mixing Value Types and Reference Types

Maintaining value semantics requires special considerations

Copy-on-write enables efficient value semantics when wrapping Swift reference types

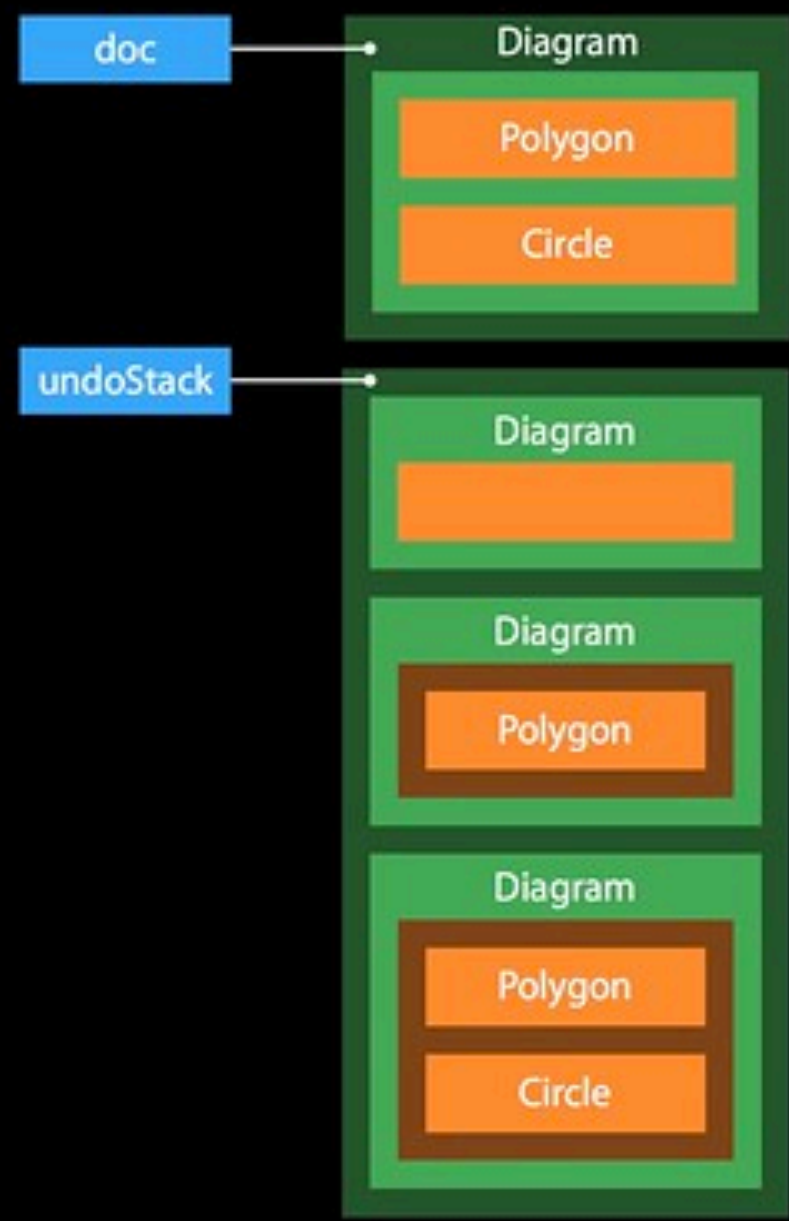
값타입과 참조타입을 섞을 때
Copy on write으로 변경할 수 있는 타입으로 참조타입이 있다는 사실에도 불구하고 value semantics을 유지하는지 확인하고 싶어함

# Implementing Undo with Value Types

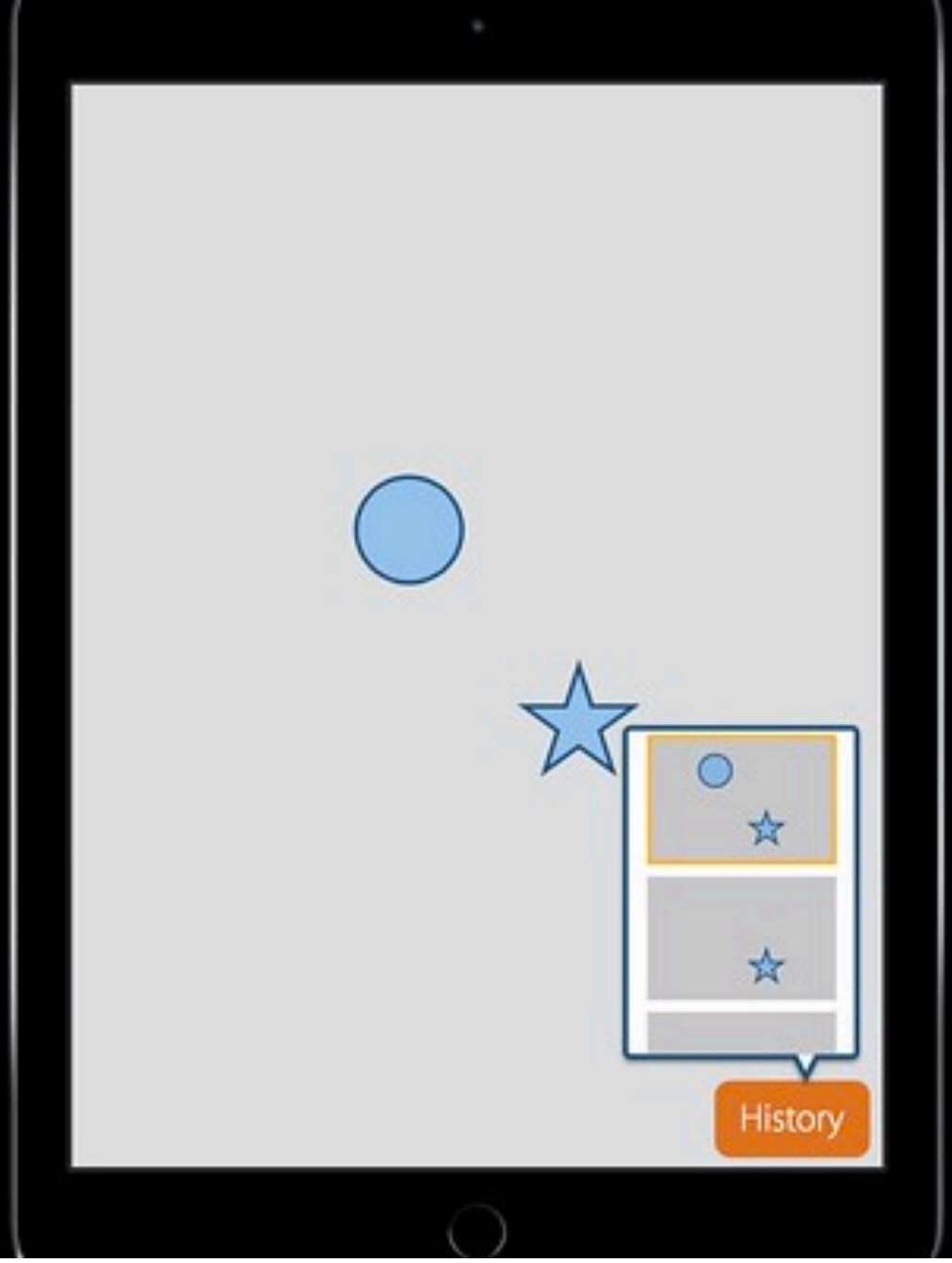
값 타입으로 모델을 구현하고 undoStack을 구현하는 기능을 보자

# Undo

```
var doc = Diagram()
var undoStack: [Diagram] = []
undoStack.append(doc)
doc.addItem(Polygon())
undoStack.append(doc)
doc.addItem(Circle())
undoStack.append(doc)
```




doc이라는 Diagram을 만들고,
undoStack이라는 Diagram 배열을 만들어 줄것임
그리고 나서 doc에 변경을 주고 다시 undoStack에 doc을 추가해줌
이제 undoStack은 세 개의 다른 Diagram 인스턴스가 있음
이들은 세 개의 다른 값이고 같은 것을 참조하지 않음




이것으로 멋진 기능을 구현할 수 있음
History 버튼이 있고 undoStack을 통해 이전 Diagram 상태를 확인할 수 있음
각 탭을 통해 이전으로 돌아갈 수 있음

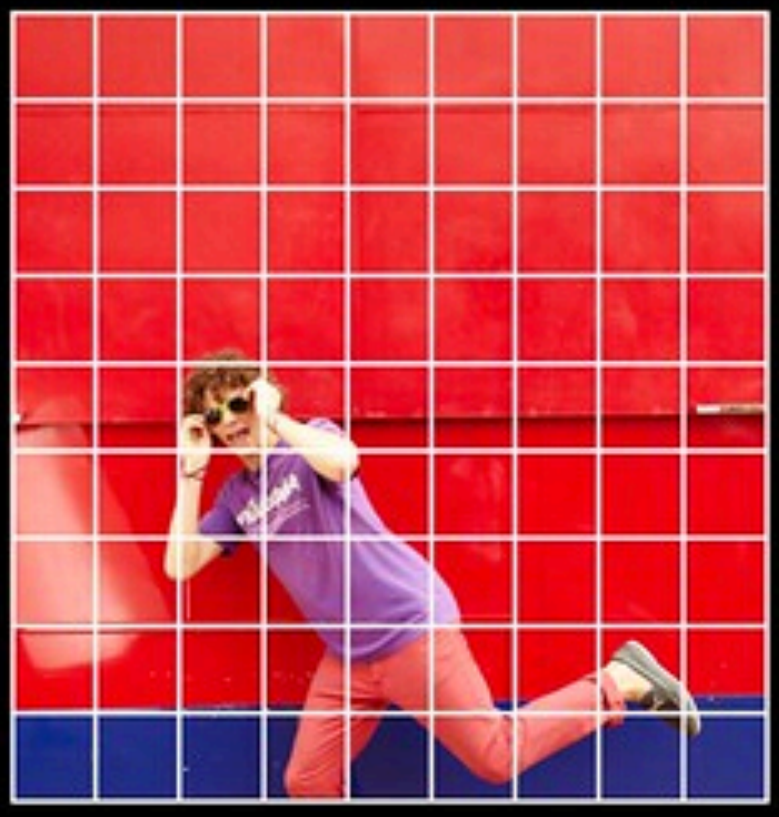
undo에 프로퍼티나 어떤 것을 추가한 배열을 유지하지 않아도 됨
단지 이전 인스턴스로 돌아가서 다시 그린 것일 뿐

# Value Semantics

## Photoshop uses value semantics

Every action results in a doc instance

Efficient because of copy-on-write



Parent, Sean.
*Value Semantics and Concept-based Polymorphism.*
C++ Now!, 2012

이는 매우 강력한 기능으로 포토샵에서 이 기능을 광범위하게 활용함
포토샵에서 이미지를 열면 이미지를 여러개의 작은 탕일로 나눠줌

# Value Semantics

## Photoshop uses value semantics

Every action results in a doc instance

Efficient because of copy-on-write

Parent, Sean.
*Value Semantics and Concept-based Polymorphism.*
C++ Now!, 2012

이 이미지의 셔츠 색깔을 바꾸면 셔츠가 포함된 타일인 다이어그램의 두 인스턴스에서 복사됨
두개의 다른 document가 있다고 하더라고 값이 변경된 타일에 대한 데이터만 소비가 됨

# Summary

Reference semantics and unexpected mutation

Value semantics solve these problems

Expressiveness of mutability, safety of immutability

PPT 페이지 명시할 것 피피티 페이지 링크 걸수 있음
상세한 것은 각 페이지에 주석으로 달기

| | 궁금한 점 | 이해 안된 점 |
|---|---|---|
| **보리** | | |
| | 1. 기술적으로 사본이 필요없다는 의미..?<br>2. defensive Copying 이라는 말이 우리가 의도하지 않는 공유를 미리 막기 위한 복사라는 의미인가<br>3. 언어 레벨로 내려간다는것의 의미 | 에라토스 테네스의 체 예시에서 재귀를 돌리면 좋다는건 알겠는데 이게 참조 의미론이랑 무슨의미인지 모르겠... |
| | | |
| | | |

PPT 페이지 명시할 것 피피티 페이지 링크 걸수 있음
상세한 것은 각 페이지에 주석으로 달기

| | 궁금한 점 | 이해 안된 점 |
|---|---|---|
| **데릭** | 이 예제가 이해가질 않아요 | Eratosthenes의 Sieve 는 소수를 계산하는 고대 알고리즘 |
| | Haskell이란? 좀 더 얘기해보고 싶다 | |
| | String 타입 A를 만들고 b에 a를 할당하는 것이 a에는 영향이 없다 → 이게 값 타입이기 때문? | String이 구조체인 것은 알겠는데 String은 heap영역에 할당된다. 왜 그런걸까?<br>가변적인 것이기 때문에 heap에 할당된다. |
| | UIImage는 Immutable이기 때문이다.<br>image 값이 변하여도 image2 값에 영향을 안받는다? | 왜? 장난하나? |

PPT 페이지 명시할 것 피피티 페이지 링크 걸수 있음
상세한 것은 각 페이지에 주석으로 달기

| | 궁금한 점 | 이해 안된 점 |
|---|---|---|
| **바드** | value semantic<br>(값 의미론?)<br>이 정확히 뭐죠?<br>참고할만한 블로그1<br>참고할만한 블로그2 | |
| | Bitmap | |
| | Copy on write | |
| | | |