Understanding Swift Performance

데릭

보리입니다 데스

바드

보리 Method Dispatch 전까지 - 99 슬라이드 , 영상 00:00 ~ 18:00

데릭 209 페이지
D2Coding으로 할 것

보리 | | | |

struct

class

enum

inheritance

generics

protocols

스위프트는 코드 재사용 및 역동성을 위한 다양한 퍼스트 클래스 유형과 매커니즘을 가지고 있다.

올바른 추상화 매커니즘을 선택하기 위해서는 Performance(성능영향)에 대해서 이해하는것이 더 나은 솔루션을 찾는데 도움이 될 수 있다.

스위프트의 추상화 매커니즘의 성능 영향을 이해하는 가장 좋은 방법은
기본 구현을 이해하는 것이다.

Agenda

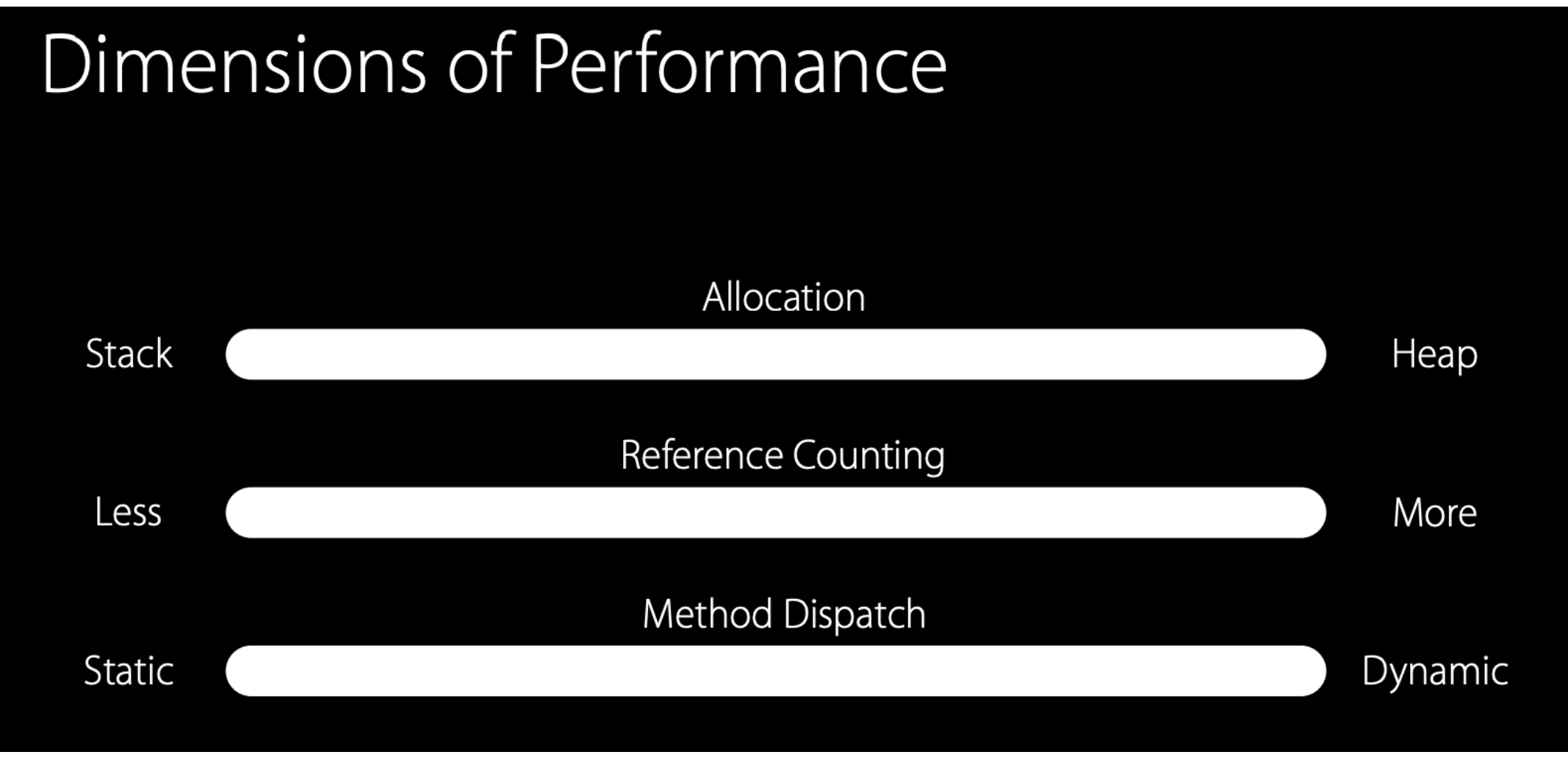
Allocation

Reference counting

Method dispatch

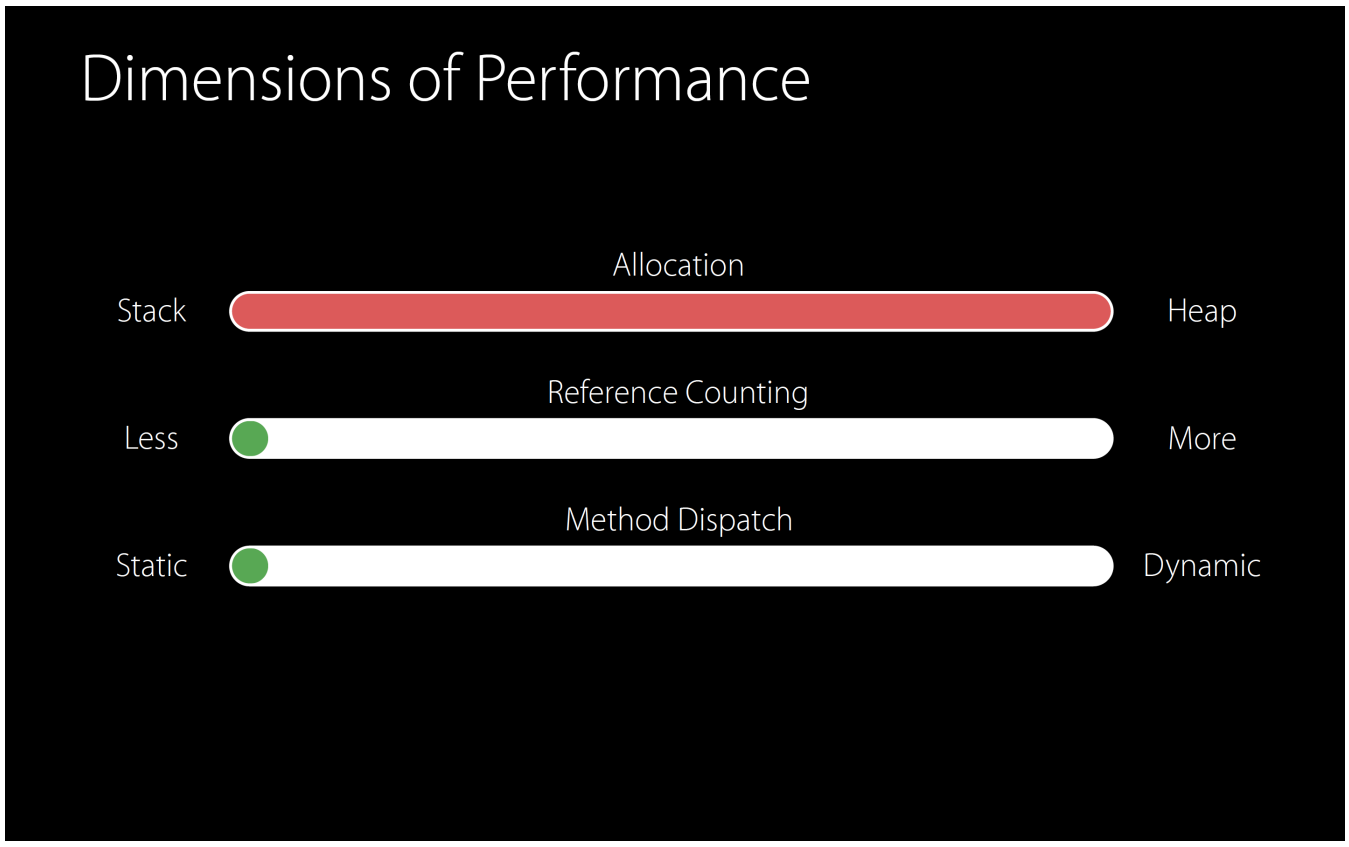
다양한 추상화 매커니즘 옵션을 평가할 때 고려할 사항에 대해 보자

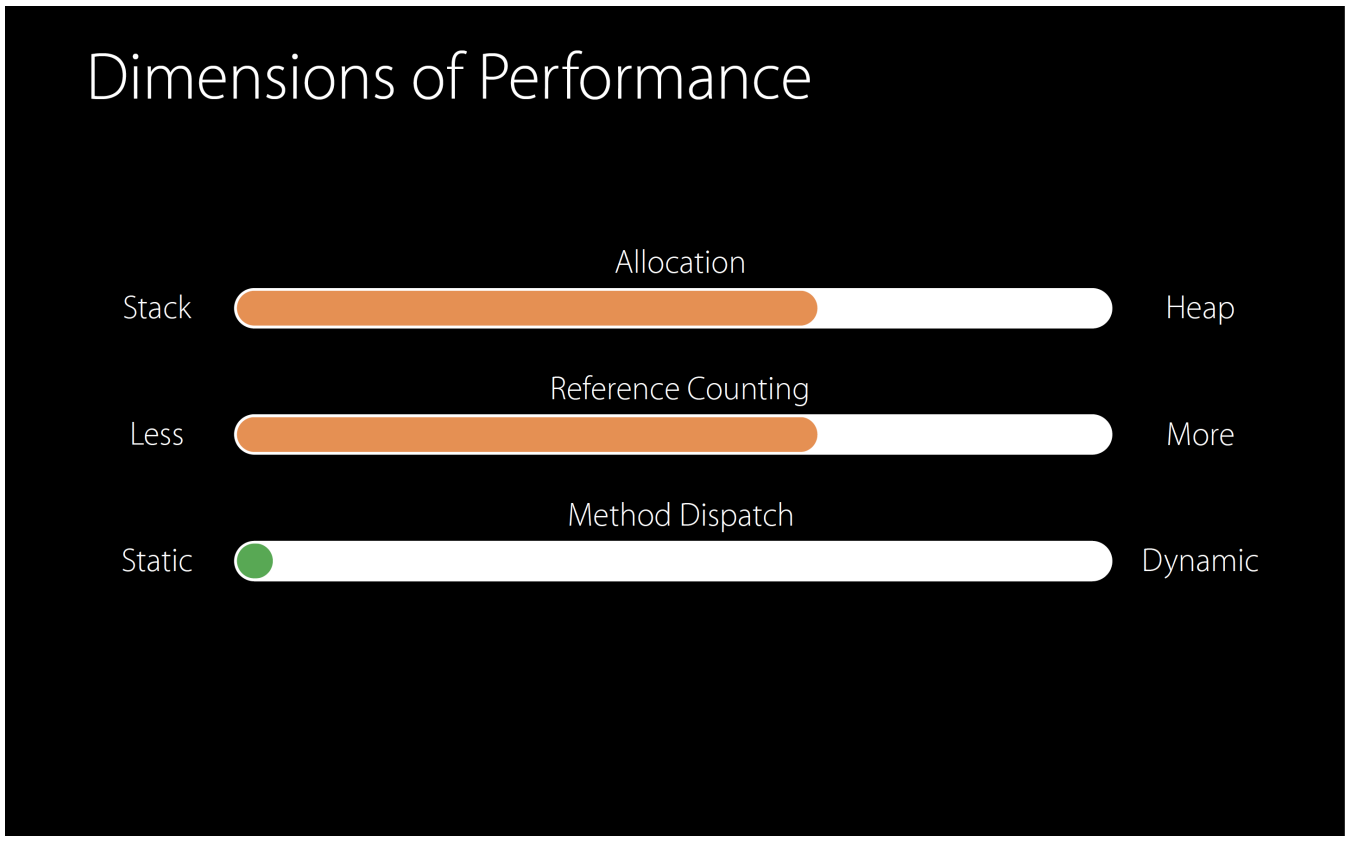
이들 각각에 대한 구조체와 객체를 사용할 때에 대한 오버헤드를 다음과 같은 멘탈 모델을 고려해볼 예정



추상화를 구축하고 매커니즘을 선택할 때
"내 인스턴스가 스택이나 할당되는가?"
"이 인스턴스를 전달할 때 참조 카운팅 오버헤드가 얼마나 될까?"
라고 자문해야 한다.

또한 "이 인스턴스에서 메서드를 호출하면 어떻게 디스패치될까?(정적, 동적)"
바른 swift 코드를 작성하려면 활용되지 않는 역동성이나 런타임시의 비용을
지불하지 않아야 한다.





우리는 더 나은 성능을 위해 3가지 차원의 트레이드오프를 고려해보아야 한다.

이제 각자의 차원(Allocation, ReferenceCounting, Method Dispatch)에 대해서 알아보자

Allocation

Stack

Decrement stack pointer to allocate

Increment stack pointer to deallocate

스위프트는 사용자를 대신하여 자동으로 메모리를 할당 및 해제함.

스택은 간단한 데이터구조
스택의 끝으로 push하고 끝에서 pop 가능

끝에서만 추가하거나 제거할 수 있기에 스택 끝에 포인터를 유지함으로 써 스택을 구현하거나 푸시팝을 구현할 수 있다.

스택 끝에 있는 포인터를 스택포인터라 함.

함수를 호출할 때 공간을 만들기 위한 스택포인터를 감소시키는것만으로 필요한 메모리를 할당할 수 있다.

함수가 끝나면 스택포인터를 이 함수의 호출 위치 전으로 증가시켜 간단히 할당해제할 수 있다.

스택할당은 정말 빠르다. 정수를 할당하는 비용

Allocation

Heap

Advanced data structure

Search for unused block of memory to allocate

Reinsert block of memory to deallocate

Thread safety overhead

힙은 스택과는 대조되게 효율성이 떨어진다.

힙을 사용하면 스택에서 동적수명으로 메모리를할 당 할 수 잇다. 스택은 못함.

다만 이를 구현하려면 고급 데이터 구조가 필요하다.
힙에 메모리를 할당하려면 힙 데이터 구조를 검색하여 적절한 크기의 "사용되지 않는 블록"을 찾아야 한다.

또한 작업이 끝난 뒤 할당을 해제 하려면 해당 메모리를 다시 적절한 위치에 삽입해야 한다.

이 작업은 스택이 정수를 할당하는 비용을 쓰는것과 다르게 많은것들이 관련되어있다. 다만 이것은 힙의 비용에 엄청 영향을 끼치는게 아님

여러 스레드가 동시에 힙에 메모리를 할당할 수 있으므로 잠금이나 동기화 매커니즘을 사용하여 무결성을 보호해야함. **이는 큰 비용임**

코드를 추적하며 swift가 우리를 대신해 뭘 하는지 보도록 하자.

```swift
// Allocation
// Struct

struct Point {
   var x, y: Double
   func draw() { … }
}

let point1 = Point(x: 0, y: 0)
var point2 = point1
point2.x = 5
// use `point1`
// use `point2`
```

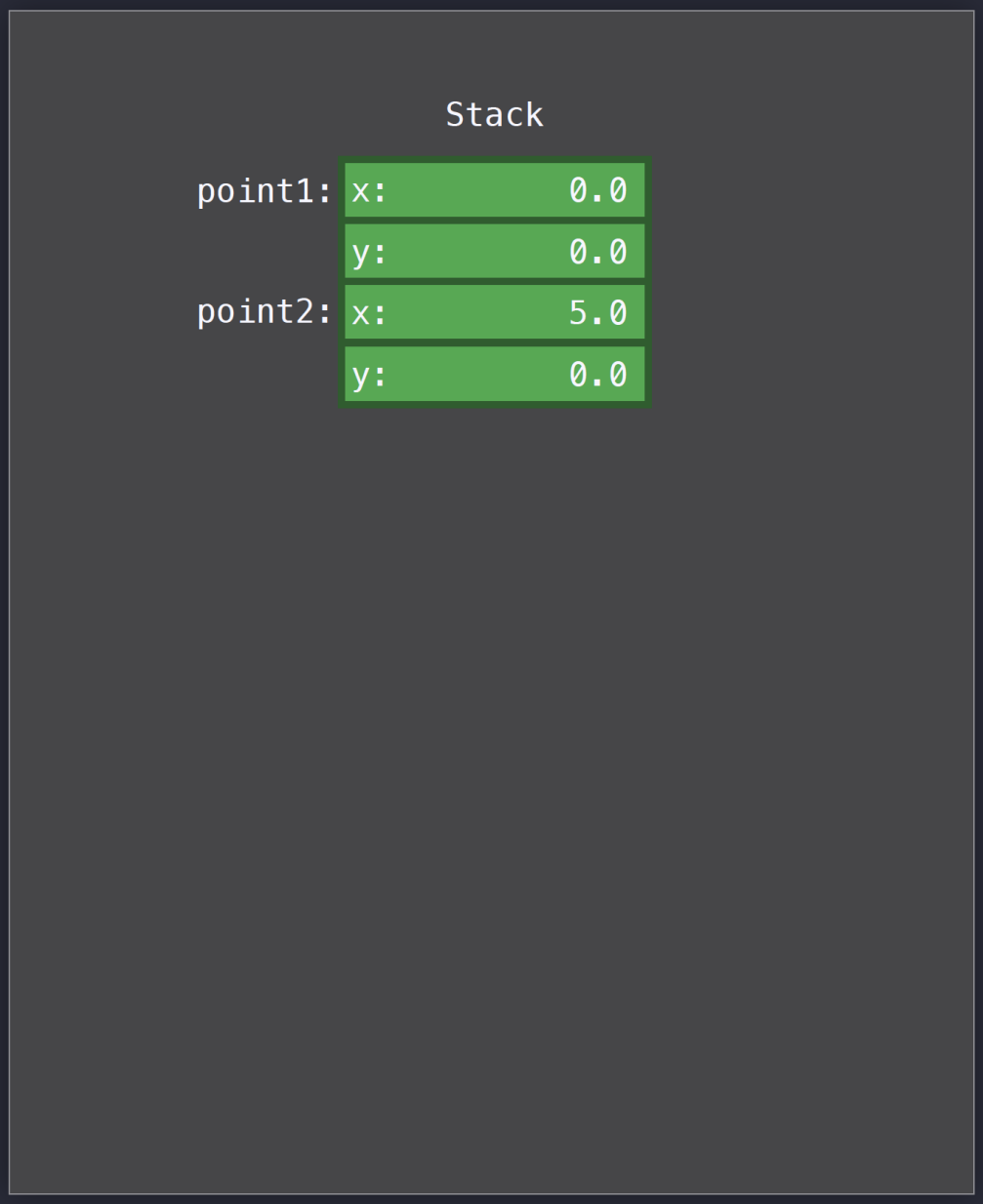
저장 프로퍼티인 x 및 y 가 있는 Point 구조체가 있다.
여기에는 draw 메서드도 있다.
[0,0]으로 point1을 구성하고 point2에 이를 할당하여 복사본을 만들고
point2.x에 5를 할당한다.

그 뒤 각 구조체를 사용한다.

point는 구조체이기 때문에 x 및 y 속성은 스택의 줄에 저장된다.

[0,0]인 point1을 구성할 때 이미 스택에 할당된 메모리를 초기화 하기만 하면됨.

```swift
// Allocation
// Struct

struct Point {
    var x, y: Double
    func draw() { … }
}

let point1 = Point(x: 0, y: 0)
var point2 = point1
point2.x = 5
// use `point1`
// use `point2`
```



point2를 할당할 때도 이미 복사본을 만들고 스택에 초기화할 뿐이다.
값 의미론을따라 point2의 x에 5를 할당해도 이미 point1과 point2는 별개의 인스턴스이므로 x값은 다르다

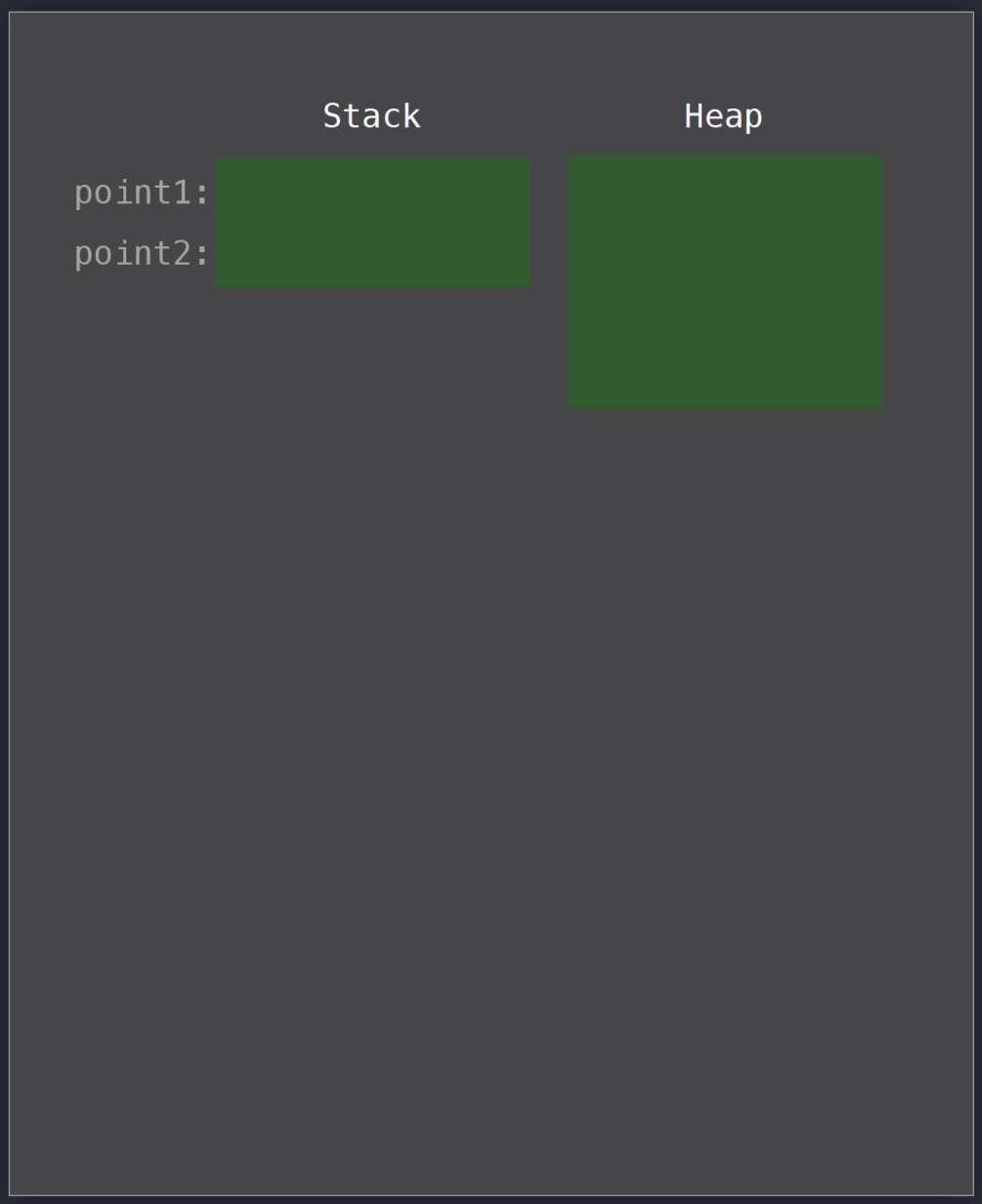
```swift
// Allocation
// Struct

struct Point {
   var x, y: Double
   func draw() { … }
}

let point1 = Point(x: 0, y: 0)
var point2 = point1
point2.x = 5
// use `point1`
// use `point2`
```

Stack

point1:	x:	0.0
	y:	0.0
point2:	x:	5.0
	y:	0.0

point1과 point2를 각각 사용한다.
사용이 끝나면 함수를 입력했을 때의 위치로 돌아가 스택포인터를 증가시켜 point1과 point2에 대한 메모리 할당을 간단히 해제할 수 있다.

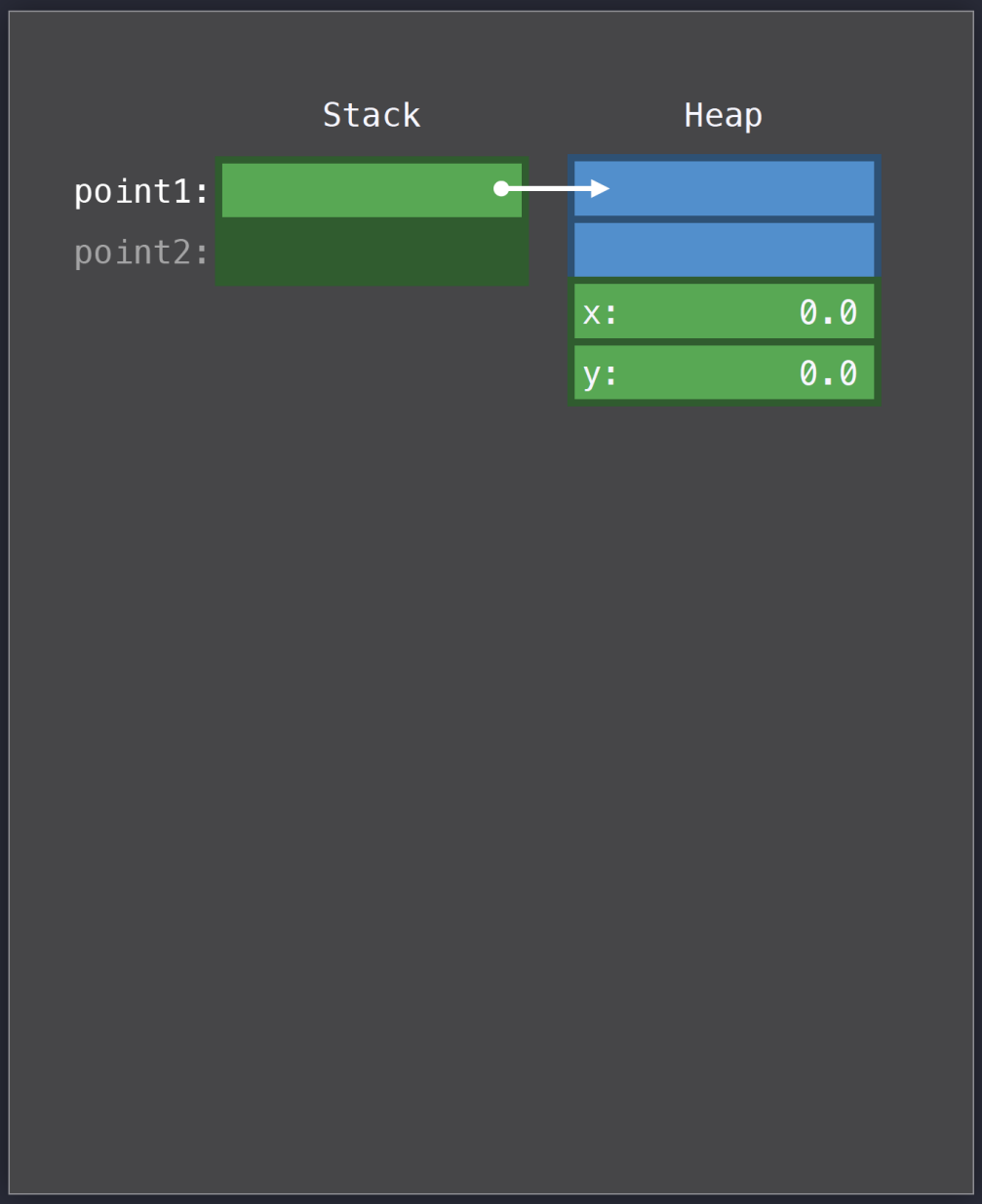
```swift
// Allocation
// Class

class Point {
    var x, y: Double
    func draw() { … }
}

let point1 = Point(x: 0, y: 0)
let point2 = point1
point2.x = 5
// use `point1`
// use `point2`
```



이제 완전 동일하지만 class인 코드와 대조해보자

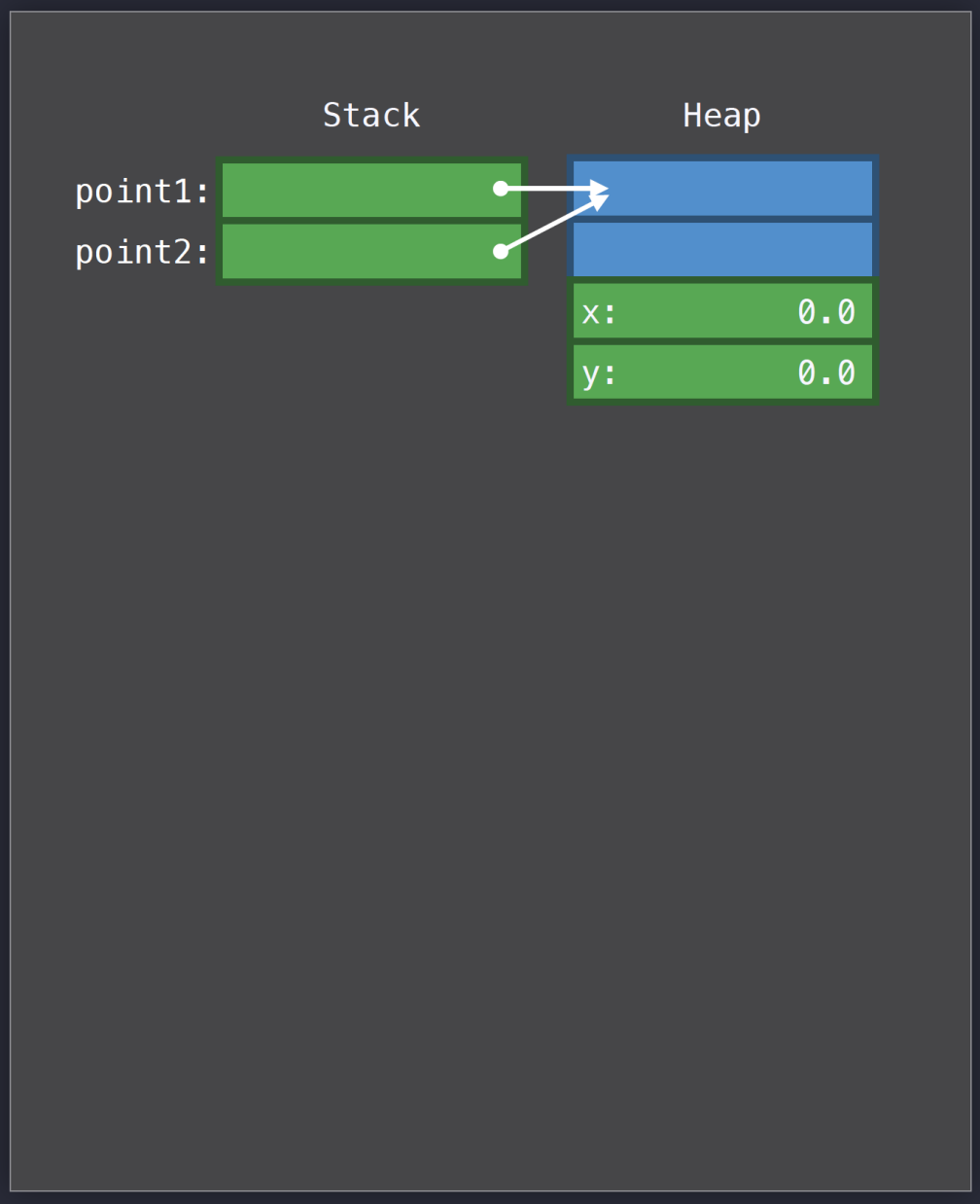
함수에 진입하면 이전과 마찬가지로 스택에 메모리를 할당한다.
다만 point프로퍼티값을 직접 저장하는대신 point값들에 대한 참조 메모리를 할당한다.
[0,0]point를 구성할 때는 힙을 잠그고(Lock) 해당 자료구조에서 적절한 크기의 사용되지 않는 메모리 블록을 검색한다.

```swift
// Allocation
// Class

class Point {
    var x, y: Double
    func draw() { … }
}

let point1 = Point(x: 0, y: 0)
let point2 = point1
point2.x = 5
// use `point1`
// use `point2`
```



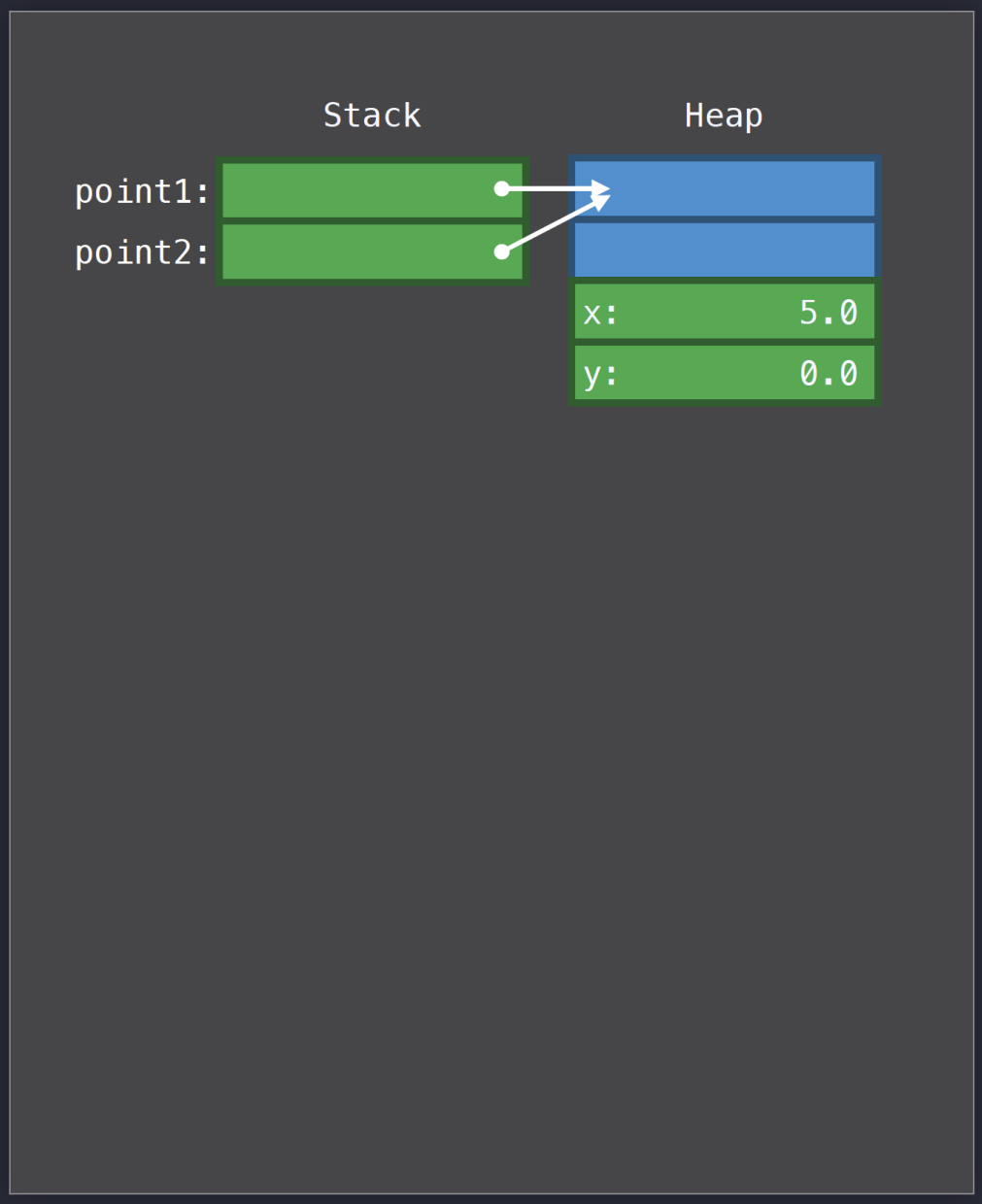
그다음 x가 0이고 y가 0인 메모리를 초기화 하고 힙의 해당 메모리에 대한 메모리주소로 point1을 초기화 한다. 힙에 할당할 때 클래스 포인트에 4word의 스토리지를 할당했다. 이것은 우리 포인트가 구조체 일 때 두 word를 할당하던것과는 대조적이다. 이것은 이제 Point가 클래스이고 x 및 y 에 대해 저장된 것 외에 **Swift가 우리를 대신하여 관리를 위한 두 word를 더 할당하기 때문.** 힙다이어 그램에서는 파란 영역

```swift
// Allocation
// Class

class Point {
    var x, y: Double
    func draw() { … }
}

let point1 = Point(x: 0, y: 0)
let point2 = point1
point2.x = 5
// use `point1`
// use `point2`
```



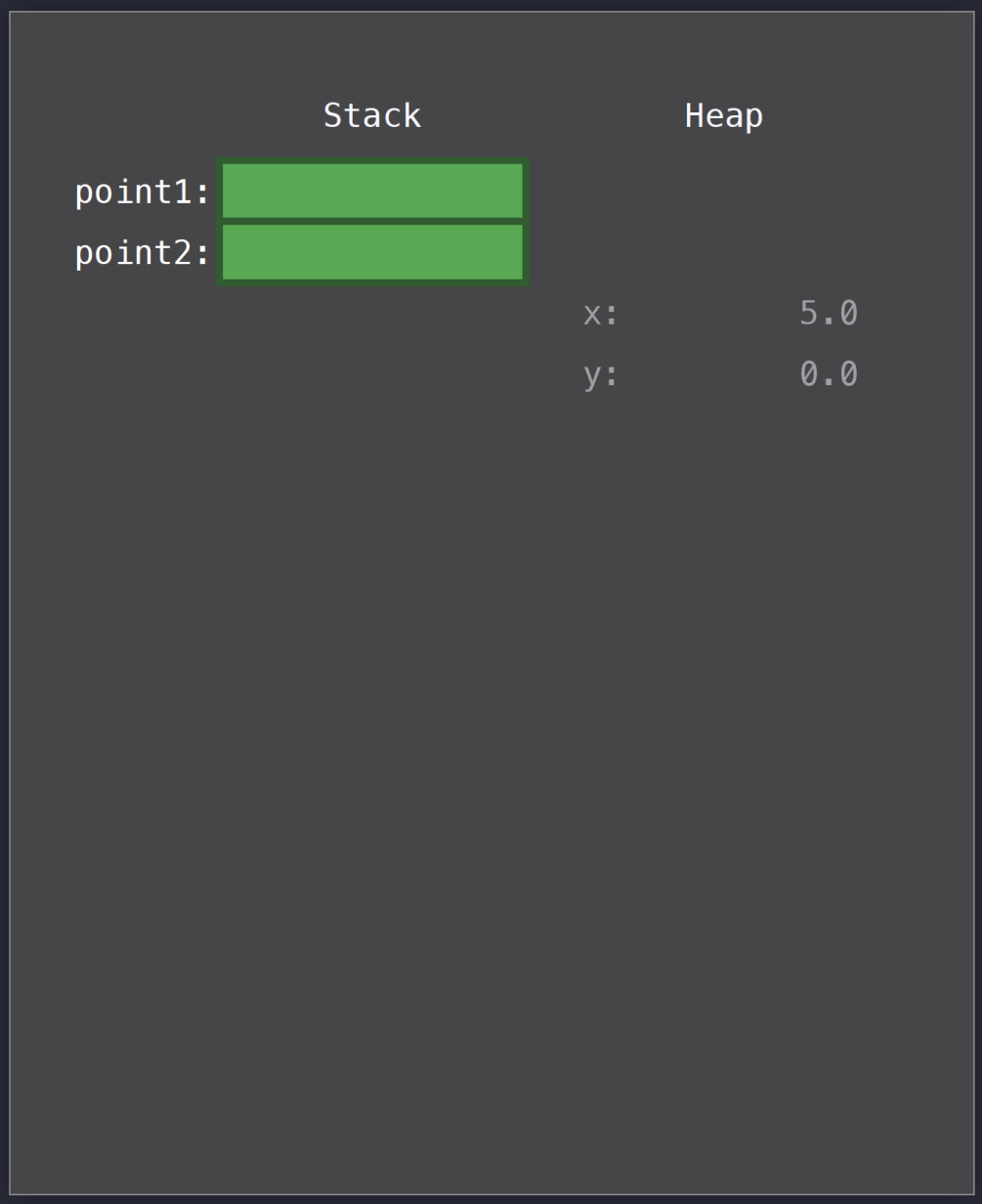
point2를 할당할 때에는 point1이 구조체였을 때 복사한것처럼 내용을 복사하는게 아니라 참조를 복사한다.
따라서 point1과 point2는 실제로 힙에 위치한 동일한 point 인스턴스를 참조한다.

```swift
// Allocation
// Class

class Point {
   var x, y: Double
   func draw() { … }
}

let point1 = Point(x: 0, y: 0)
let point2 = point1
point2.x = 5
// use `point1`
// use `point2`
```



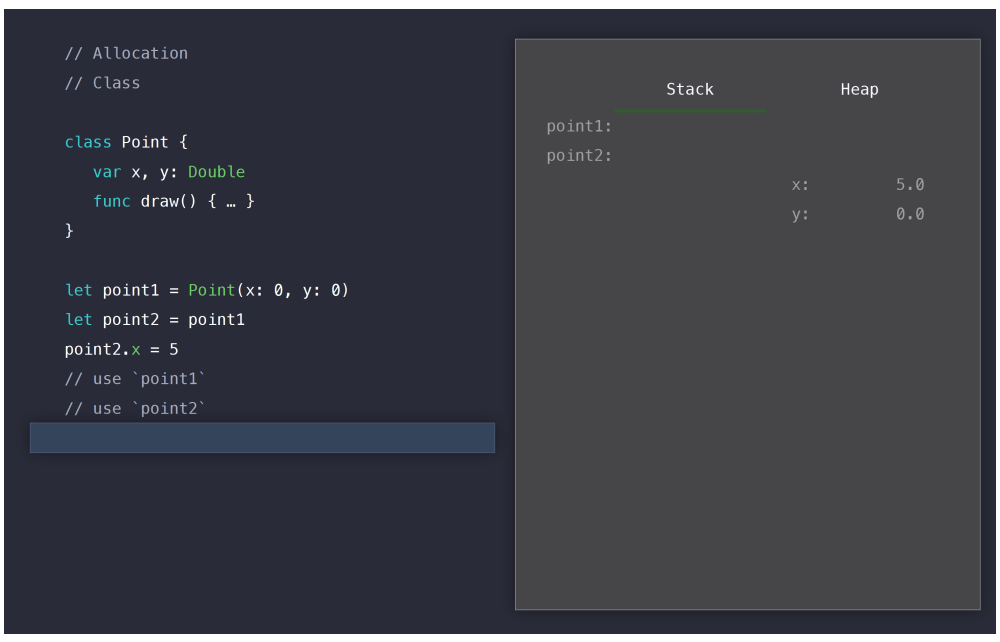
이제 point2의 x에 5를 할당하면 point값 두가지 모두 5가 된다. 이게 바로 참조 의미론이며 의도하지않는 공유(unintended sharing state)로 이어질 수 있다.

```
// Allocation
// Class

class Point {
   var x, y: Double
   func draw() { … }
}

let point1 = Point(x: 0, y: 0)
let point2 = point1
point2.x = 5
// use `point1`
// use `point2`
```

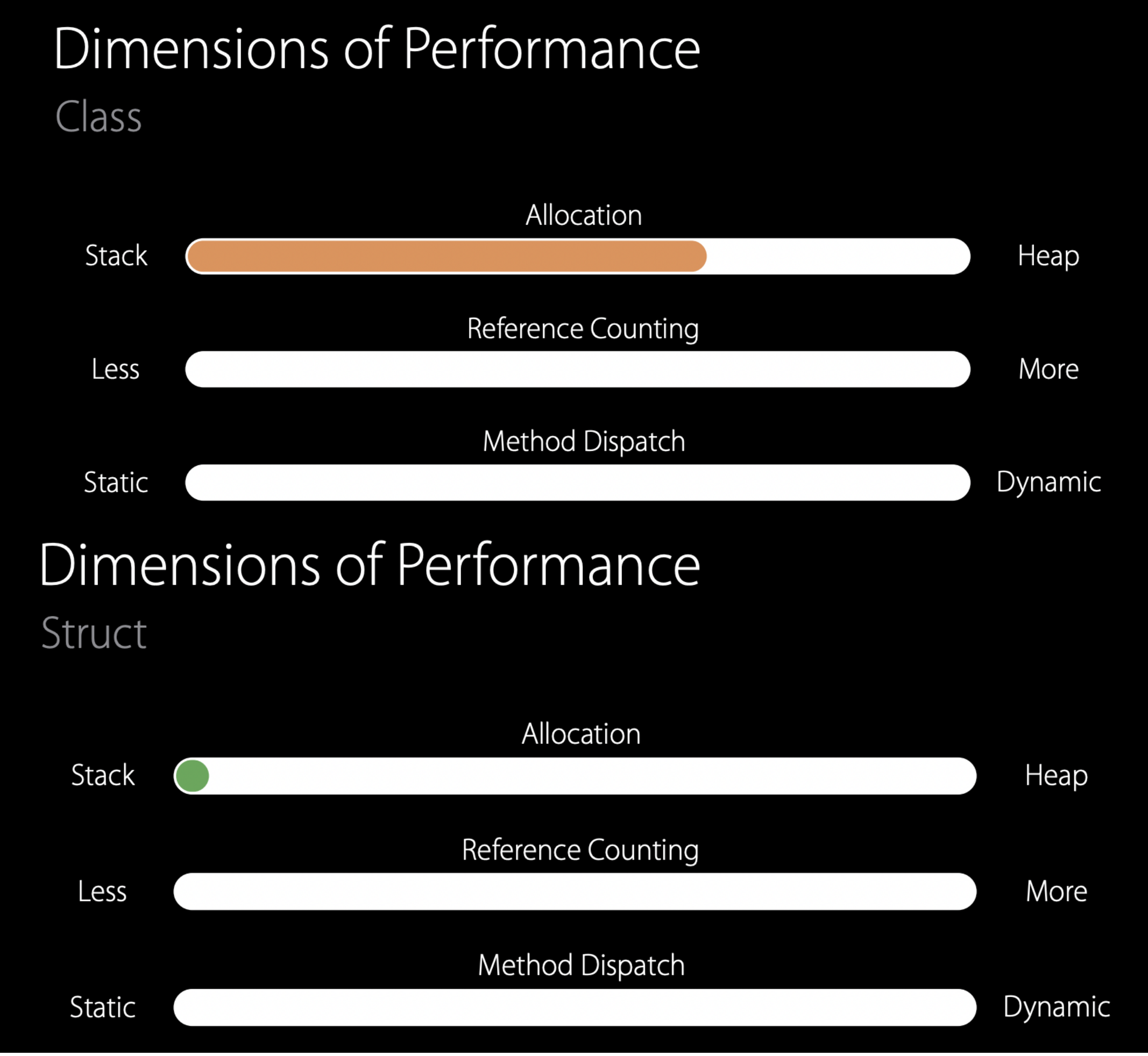


그 뒤 point1과 point2를 사용하고나서 해제를 해보자

swift는 힙을 Lock한 뒤 사용하지 않는 블록을 적절한 위치로 돌려 놓으며 메모리 할당을 해제한다.



그런 다음에 스택이 pop 된다.



클래스는 힙 할당이 필요하기 때문에 구조체를 구성하는것 보다 더 많은 비용이 듬을 확인할 수 있었다.
클래스는 힙에 할당되고 reference sementic 체계를 갖기 때문에 클래스에는 ID간접 저장과 같은 몇가지 강력한 특성이 있다.

다만 추상화 단계에서 이런 특성이 필요하지 않는다면 구조체를 사용하는것이 좋다.
그리고 구조체는 클래스 처럼 의도하지 않는 상태 공유에 취약하지 않다.

```swift
// Modeling Techniques: Allocation

enum Color { case blue, green, gray }
enum Orientation { case left, right }
enum Tail { case none, tail, bubble }



func makeBalloon(_ color: Color, orientation: Orientation, tail: Tail) -> UIImage {
    …
}
```

이를 적용하기 위해 메시징 응용 프로그램의 예시를 들어보자. 이것은 기본적으로 뷰 레이어에서 가져온 것이다.

유저들이 문자를 보내면 그 문자 뒤에 풍선 이미지를 그리고 싶다.

makeBalloon 함수는 이 이미지를 생성하는 것이며 이 함수는 다른 풍선의 configuration 또는 전체 configuration space를 지원함.

위와 같이 왼쪽 거품꼬리 회색풍선과 오른쪽 방향의 꼬리 풍선이 있을 수 있다.

이제 makeBalloon 함수는 할당 시작 및 사용자 스크롤 중에 자주 호출되기 때문에 정말 빨라야 함.

```swift
// Modeling Techniques: Allocation

enum Color { case blue, green, gray }
enum Orientation { case left, right }
enum Tail { case none, tail, bubble }

var cache = [String : UIImage]()

func makeBalloon(_ color: Color, orientation: Orientation, tail: Tail) -> UIImage {
    let key = "\(color):\(orientation):\(tail)"
    if let image = cache[key] {
        return image
    }
    …
}
```

그래서 다음과 같은 cache 레이어를 추가한다.
따라서 주어진 구성에 대해서 풍선 이미지를 두번 이상 생성할 필요가 없어졌다.
단지 한번 생성한 이미지라면 캐시에서 꺼내 쓰면됨.
캐시 작업을 위해 색상 방향 및 tail을 문자열인 키로 serialize 했다.

다만 몇가지 문제가 존재.

문자열은 키에 대해 강력한 타입이 아님.
예를들어 color라고 configuration을 정했지만 color에 강아지 이름을 넣을 수도 있다. 이는 별로 안전한 방법이 아님.

String은 그리고 힙에 간접적으로 해당 문자의 내용을 저장한다

따라서 makeBalloon 함수를 호출할 때마다 캐시 hit이 있더라도 힙 할당이 발생한다.
(키를 만드는 과정에서의 힙 할당)

```swift
// Modeling Techniques: Allocation

enum Color { case blue, green, gray }
enum Orientation { case left, right }
enum Tail { case none, tail, bubble }

var cache = [Attributes : UIImage]()

struct Attributes : Hashable {
    var color: Color
    var orientation: Orientation
    var tail: Tail
}

func makeBalloon(_ color: Color, orientation: Orientation, tail: Tail) -> UIImage {
    let key = Attributes(color: color, orientation: orientation, tail: tail)
    if let image = cache[key] {
        return image
    }
    …
}
```

이를 구조체를 이용하여 개선 해보자

우리는 구조체를 사용하여 색상, 방향, 구성공간을 나타낼 수 있다.
이는 String으로 이 구성공간을 나타내는 훨씬 안전한 방법이다.

구조체는 또한 Swift의 일급 객체이기 때문에 Dictionary에서 키로 사용이 가능하다.

이제 makeBalloon함수를 호출할 때 캐시 Hit이 있는 경우 이 구조체를 구성하는데 힙 할당이 필요하지 않기 때문에 할당 오버헤드가 없다(String은 길어지면 내부 내용을 힙에 저장하기 때문에 이 할당 오버헤드가 존재)

또한 이 구조체는 스택에 할당되므로 훨씬 안전하고 빠를것임.

Reference Counting

다음은 참조 카운팅의 차원에 대해 알아보자

할당에 대해 예시를 들 때 힙할당에 대해 세부사항을 이야기하지 않았다.
swift는 힙에 할당된 메모리를 언제 해제하는게 안전할지 어떻게 알까

swift는 힙의 모든 인스턴스에 대한 총 참조수를 유지한다

그리고 그 참조수를 인스턴스 자체에 유지한다.
인스턴스에 참조를 추가하거나 참조를 제거하면 해당 참조 카운트가 증가하거나 감소한다.
그 수가 0에 도달하면 swift 는 아무도 더이상 힙에 존재하는 해당 인스턴스를 가리키지 않는걸 알고 그때 해제하는게 안전하다고 안다.

참조카운팅 차원에서 염두에 두어야할 핵심은 매우 빈번한 작업이며 정수를 증감하는 것보다 더 많은것이 있다는 점이다.

Reference Counting

There's more to reference counting than incrementing, decrementing

- Indirection
- Thread safety overhead

첫번째는 증가 및 감소를 실행하기 위한 몇가지 수준의 간접 참조가 존재한다.

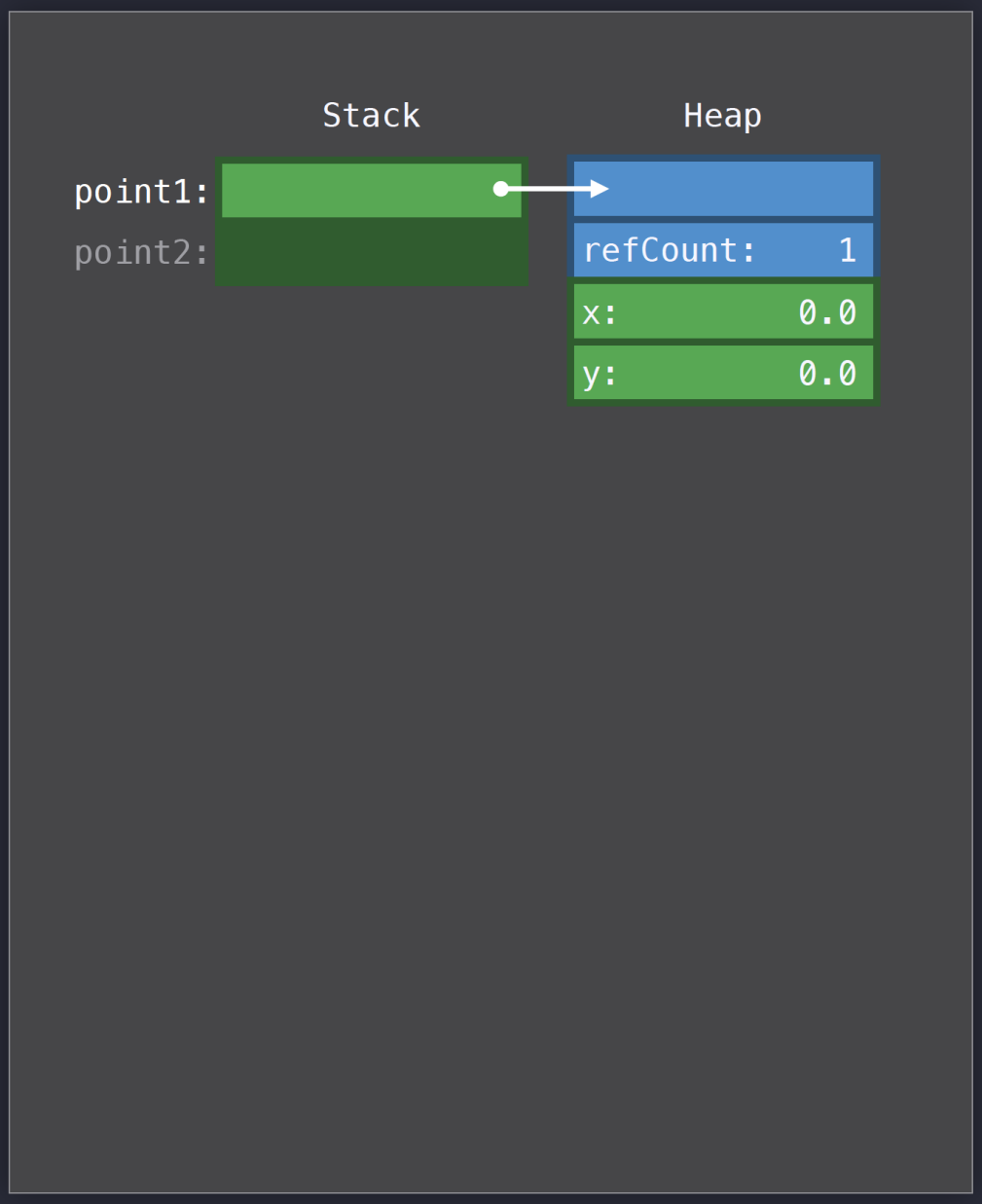
두번째는 힙할당과 마찬가지로 여러 스레드가 힙 인스턴스에 참조를 동시에 추가하거나 제거할 수 있기 때문에 thread safety를 고려해야한다는점이다.
때문에 참조 카운트는 원자적으로 증가 및 감소되어야 한다.

```swift
// Reference Counting
// Class (generated code)

class Point {
    var refCount: Int
    var x, y: Double
    func draw() { … }
}

let point1 = Point(x: 0, y: 0)
let point2 = point1
retain(point2)
point2.x = 5
// use `point1`
release(point1)
// use `point2`
release(point2)
```

Allocation에서 보았던 point클래스를 다시 보자.
앞에서 보았던 코드에서 refCount라는 추가 프로퍼티와 retain 및 release를 추가하였다.

refCount는 참조 횟수를 나타낼것이며
retain은 참조카운트를 원자적으로 증가
release는 참조 카운트를 원자적으로 감소시킬 것이다.

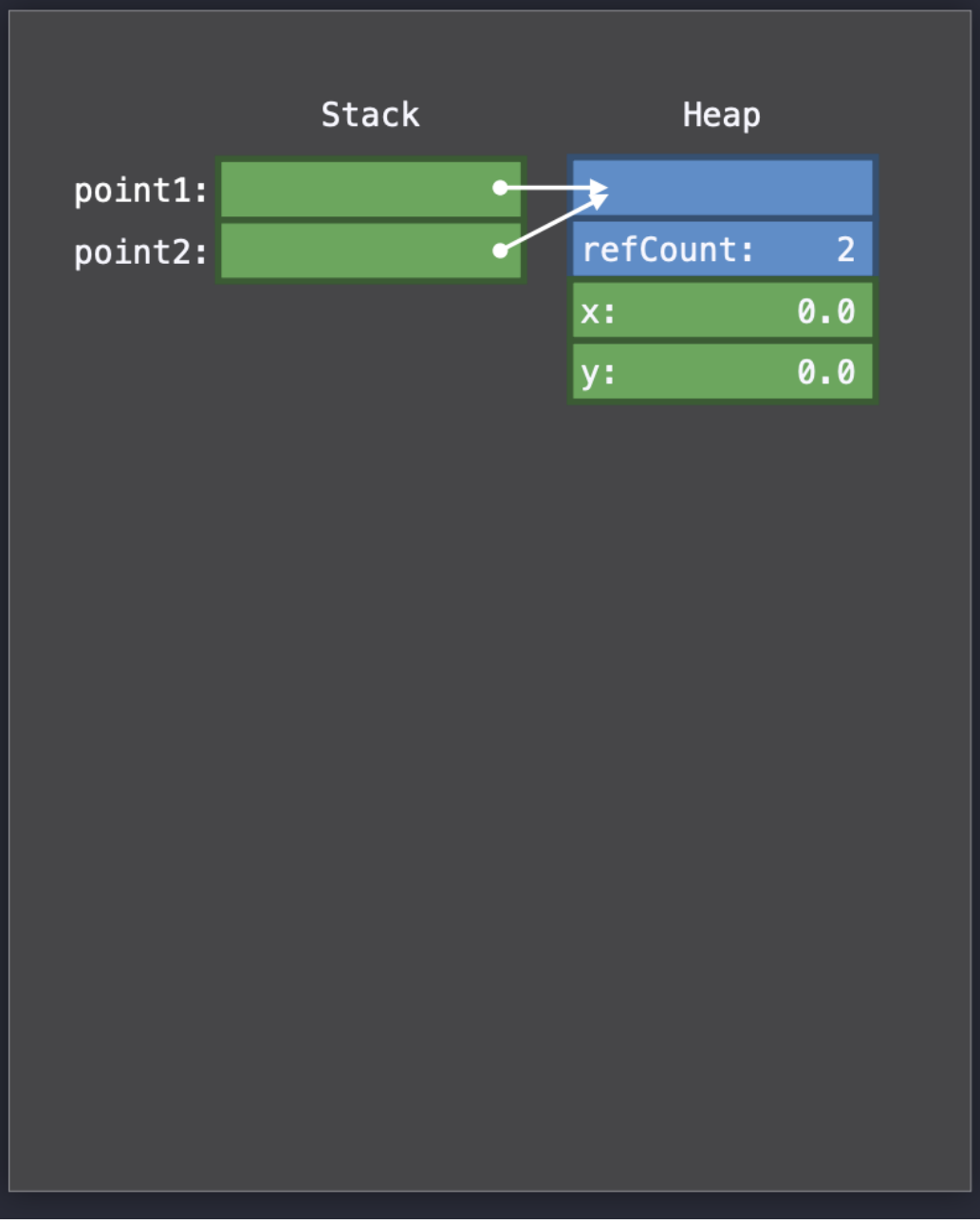
```swift
// Reference Counting
// Class (generated code)

class Point {
    var refCount: Int
    var x, y: Double
    func draw() { … }
}

let point1 = Point(x: 0, y: 0)
let point2 = point1
retain(point2)
point2.x = 5
// use `point1`
release(point1)
// use `point2`
release(point2)
```



참조 과정을 추적해보면

1. point를 힙에 구성한다.
2.해당 point(힙에있는)에 대한 참조가 stack의 point1이 존재하기 때문에 refCount가 1로 초기화 된다.

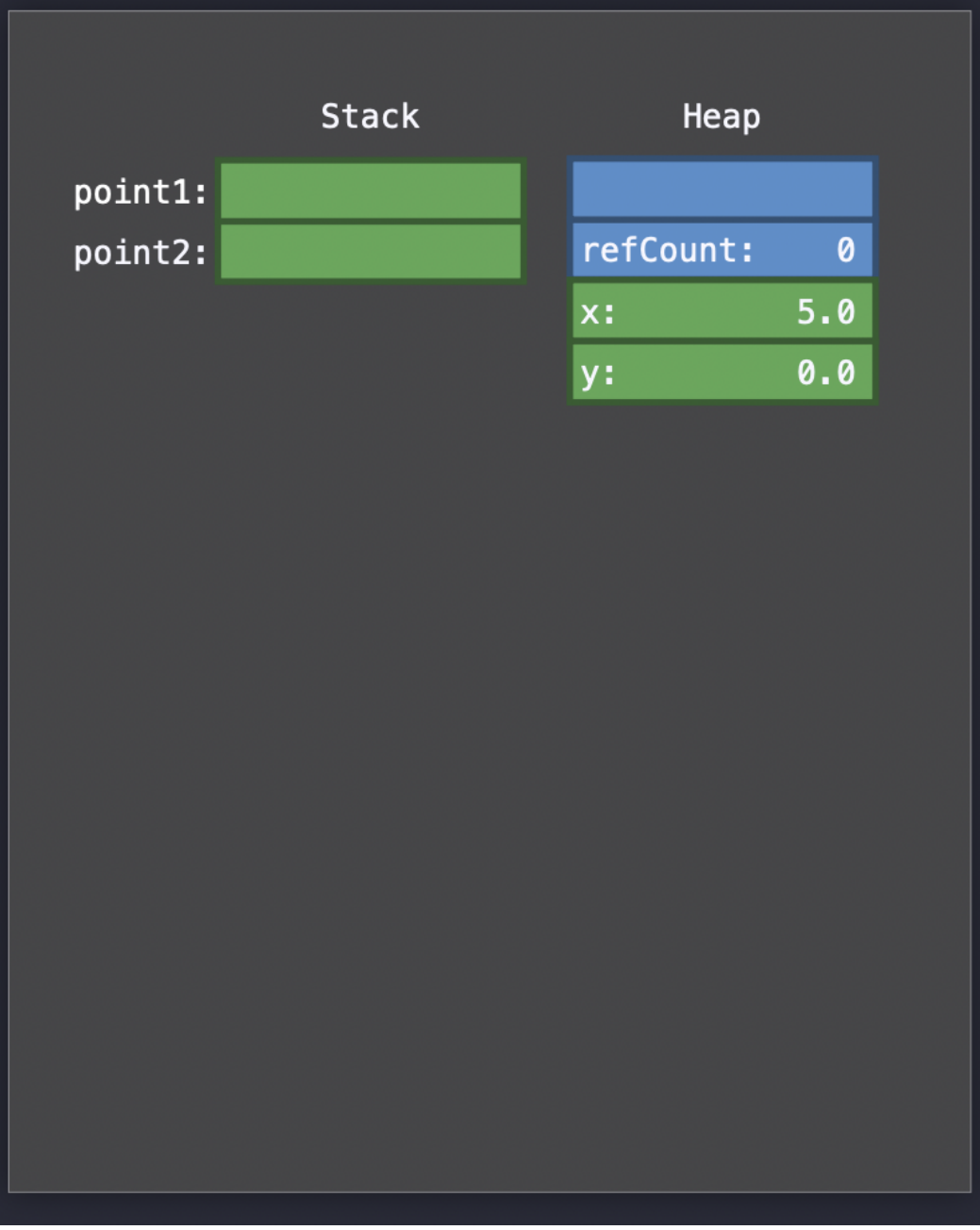
```swift
// Reference Counting
// Class (generated code)

class Point {
    var refCount: Int
    var x, y: Double
    func draw() { … }
}

let point1 = Point(x: 0, y: 0)
let point2 = point1
retain(point2)
point2.x = 5
// use `point1`
release(point1)
// use `point2`
release(point2)
```



point2에 point1의 참조 주소를 할당한다.

그러면 heap에 존재하는 Point 인스턴스를 point1과 point2가 참조하고 있으므로 refCount를 증가시키는 retain 작업을 진행한다.

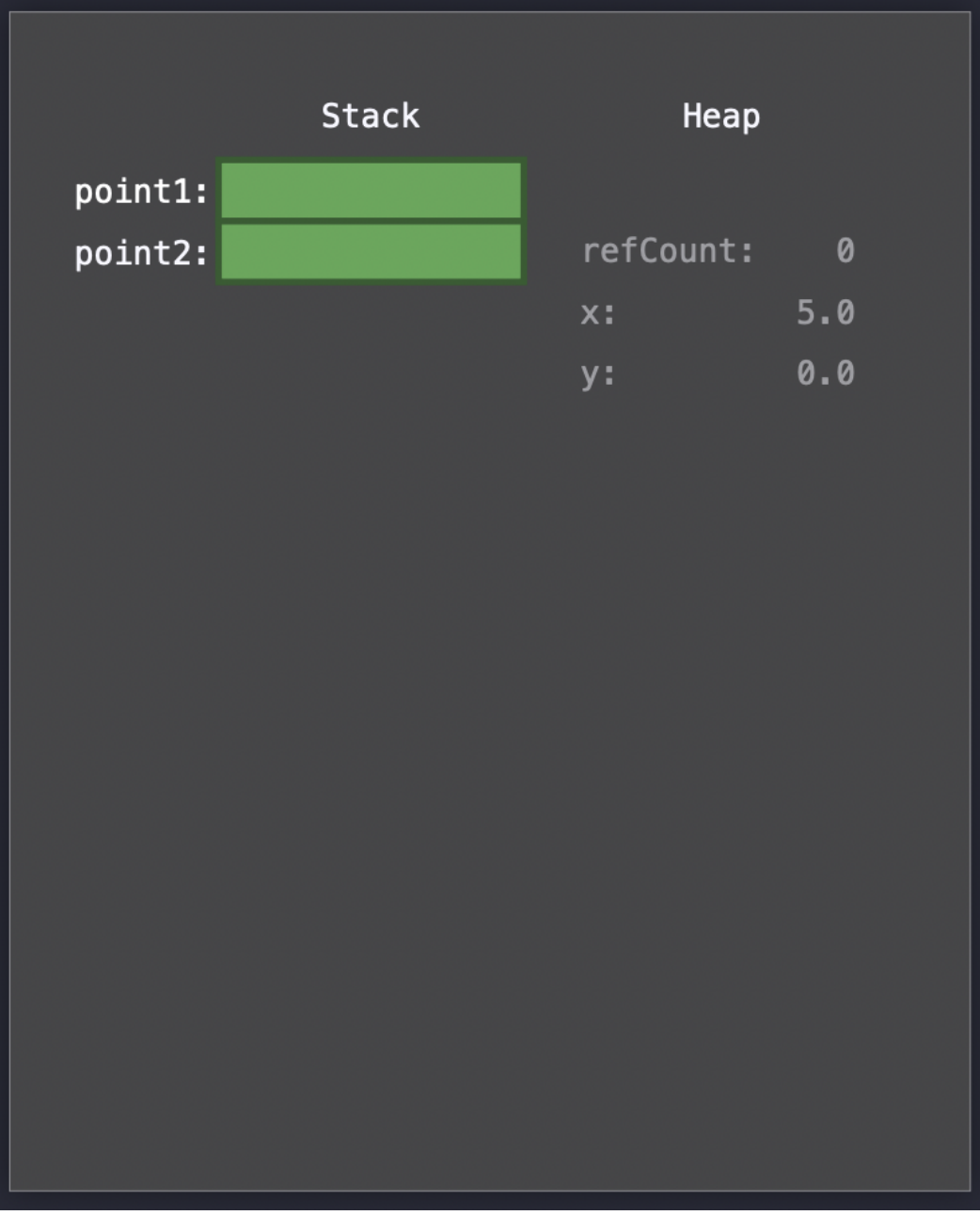
```swift
// Reference Counting
// Class (generated code)

class Point {
    var refCount: Int
    var x, y: Double
    func draw() { … }
}

let point1 = Point(x: 0, y: 0)
let point2 = point1
retain(point2)
point2.x = 5
// use `point1`
release(point1)
// use `point2`
release(point2)
```



point1을 사용하고 release
point2를 사용하고 release하여

최종적으로는 인스턴스의 refCount가 0이된다.

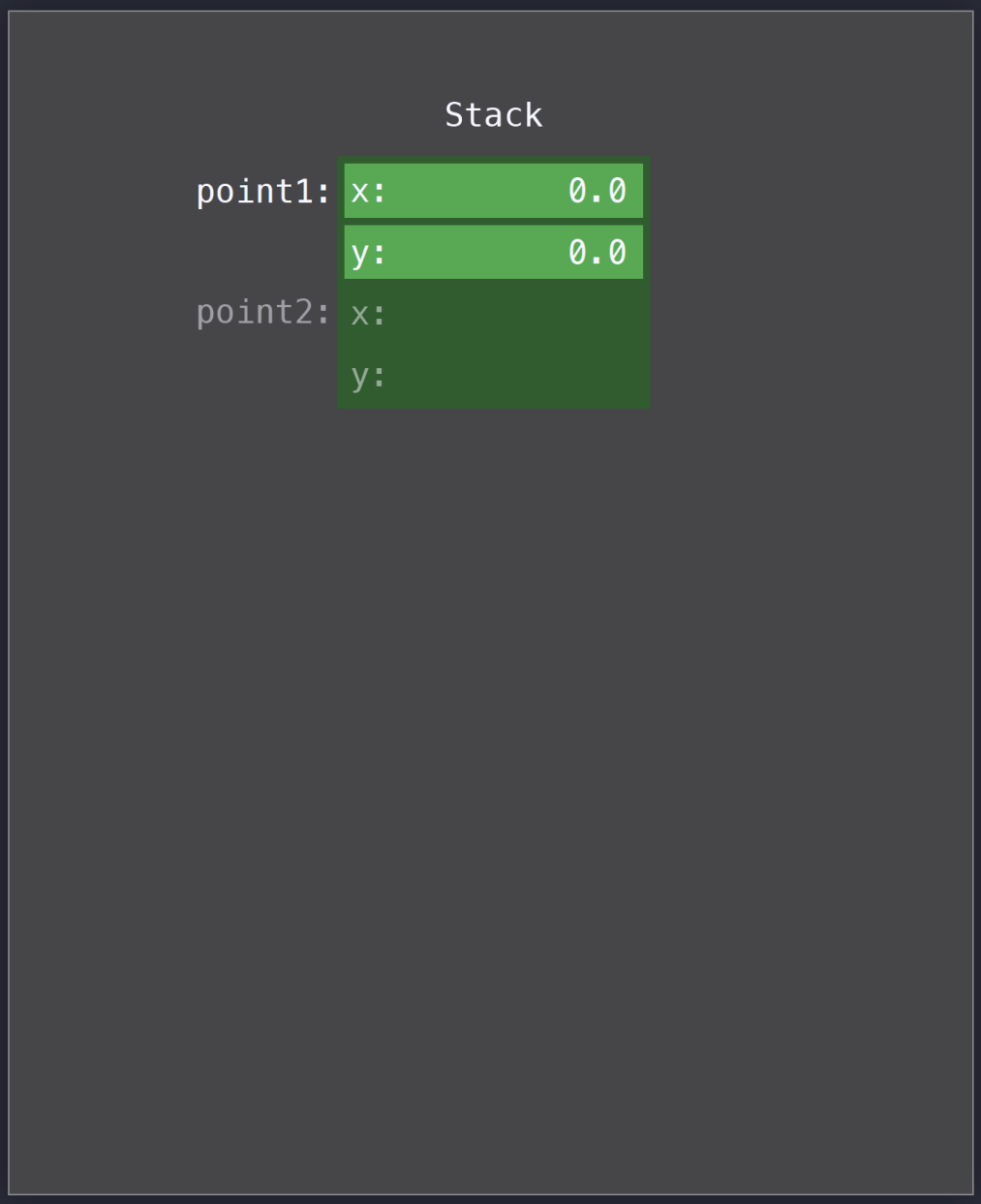
```swift
// Reference Counting
// Class (generated code)

class Point {
    var refCount: Int
    var x, y: Double
    func draw() { … }
}

let point1 = Point(x: 0, y: 0)
let point2 = point1
retain(point2)
point2.x = 5
// use `point1`
release(point1)
// use `point2`
release(point2)
```



point2의 참조를 해제하는 시점에서 힙의Point 인스턴스를 사용하는 참조가 없어진다.

이때 Swift는 힙을 lock하고 메모리 블록을 반환하는게 안전하다는 것을 알게 된다.

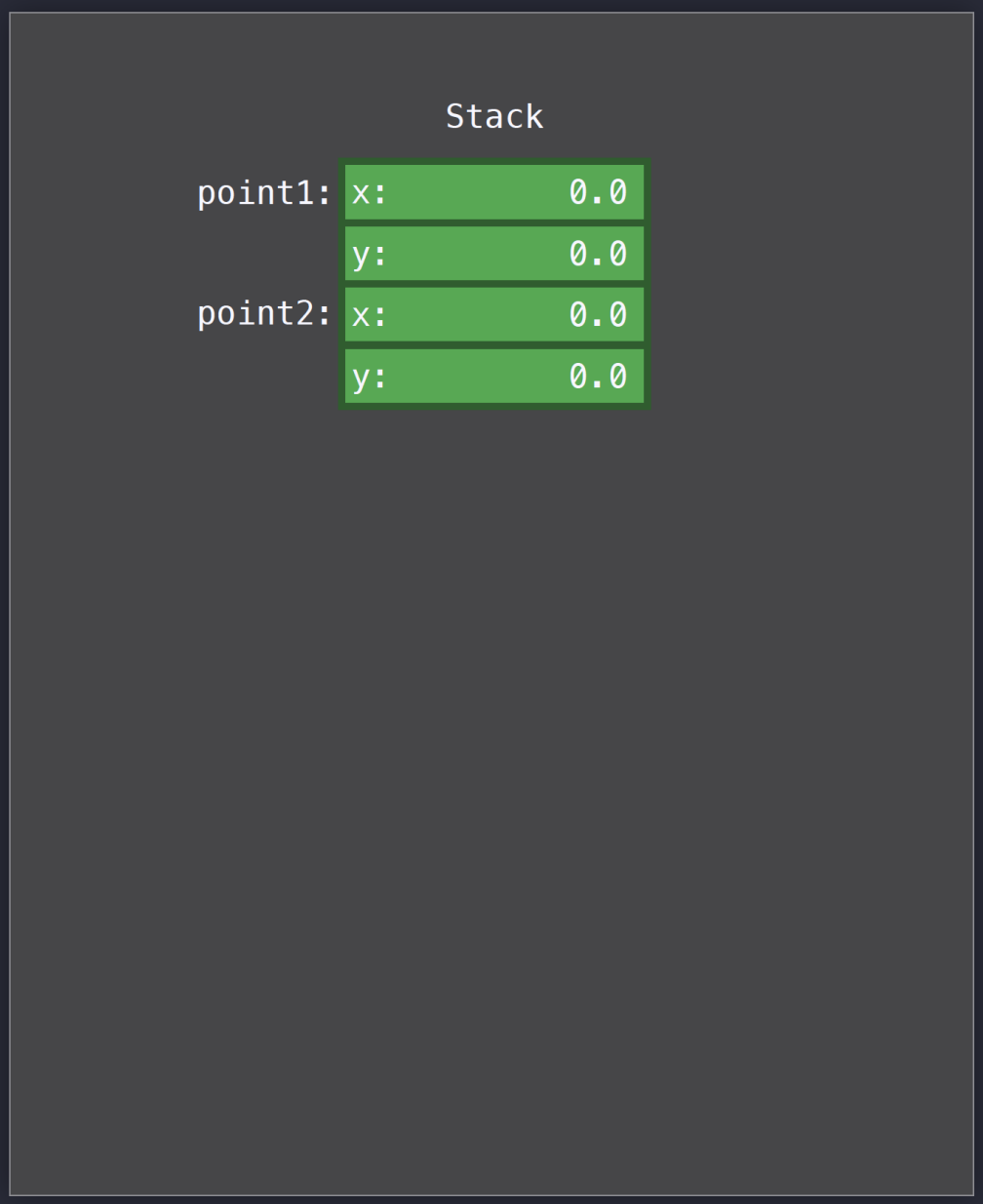
```swift
// Reference Counting
// Struct

struct Point {
    var x, y: Double
    func draw() { … }
}


let point1 = Point(x: 0, y: 0)
let point2 = point1
// use `point1`
// use `point2`
```



구조체는 어떨까?
구조체와 관련된 참조 카운팅이 존재할까?

앞에서 보았던 Point구조체를 구성할 때는 관련된 힙 할당이 존재하지 않았다.

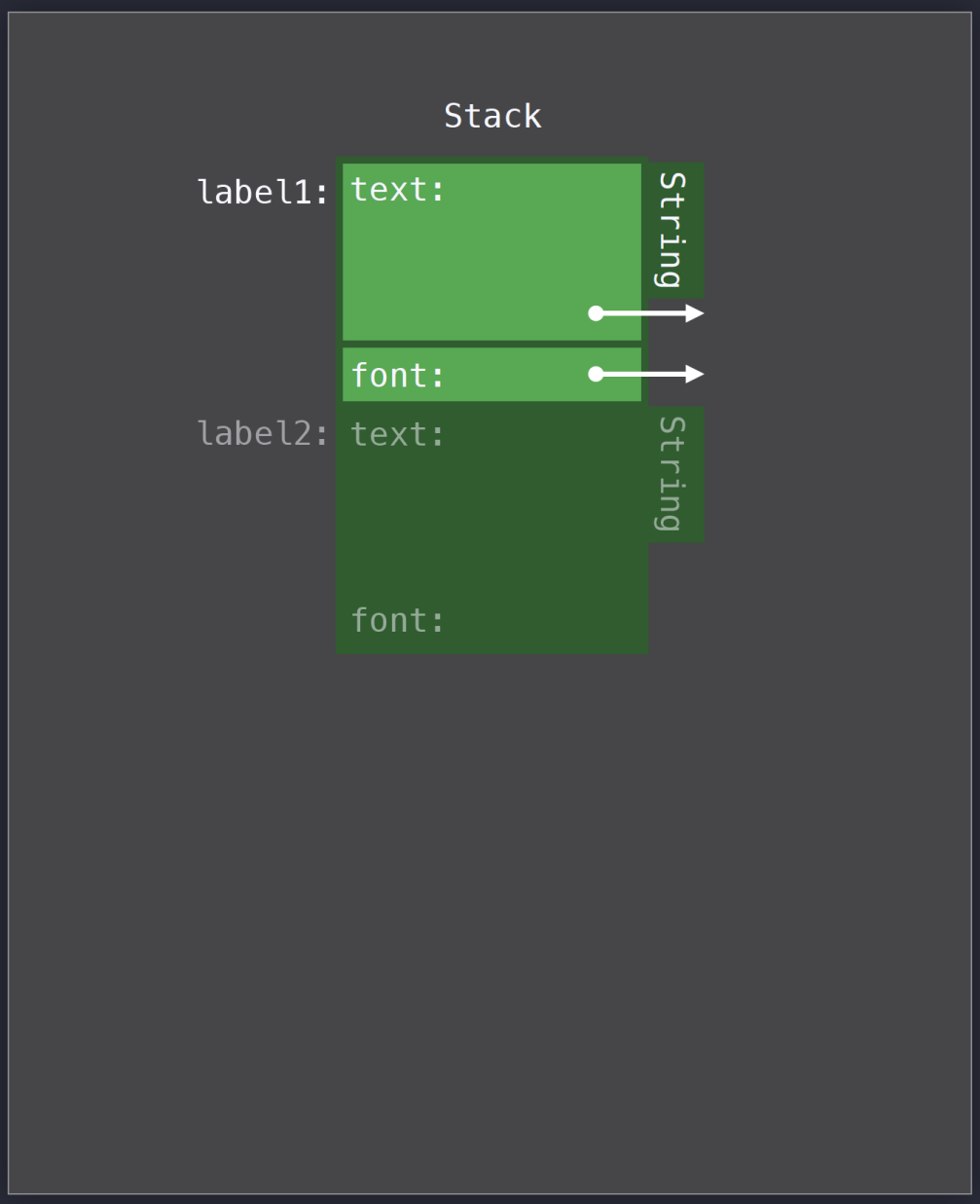
```swift
// Reference Counting
// Struct

struct Point {
    var x, y: Double
    func draw() { … }
}


let point1 = Point(x: 0, y: 0)
let point2 = point1
// use `point1`
// use `point2`
```



이는 구조체를 복사할 때도 없었다.

따라서 Point 구조체에 관한 참조 카운팅에 관련된 오버헤드(release, retain)는 존재하지 않는다.

```swift
// Reference Counting
// Struct containing references

struct Label {
    var text: String
    var font: UIFont
    func draw() { … }
}

let label1 = Label(text: "Hi", font: font)
let label2 = label1
// use `label1`
// use `label2`
```



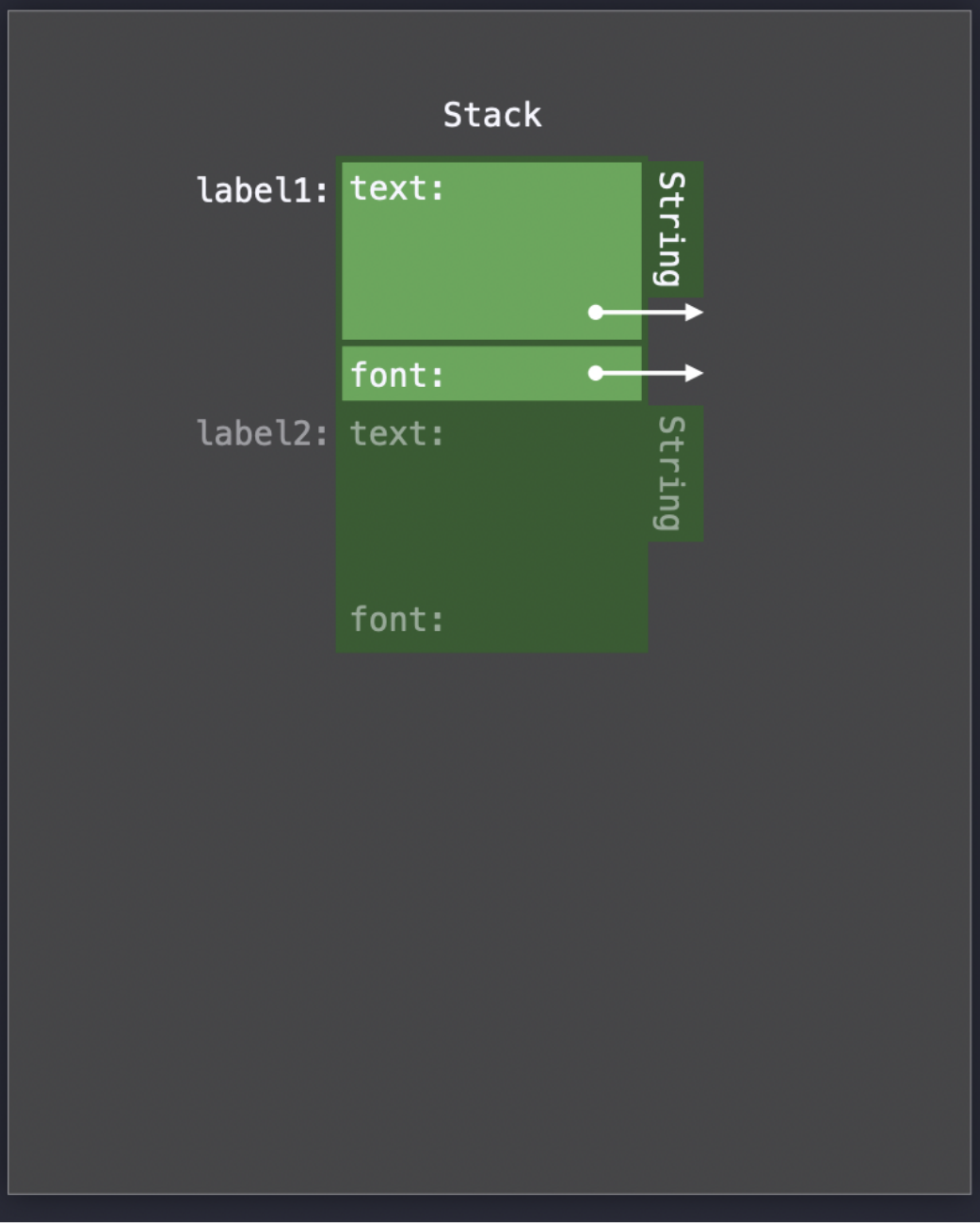
다만 더 복잡한 구조체는 어떠할까.

Label 구조체는 내부에 String 타입의 text와 UIFont타입의 font가 존재한다.

앞에서 얘기한것 처럼 String은 실제로 힙에 문자의 내용을 저장한다. 따라서 참조 카운트가 필요한다

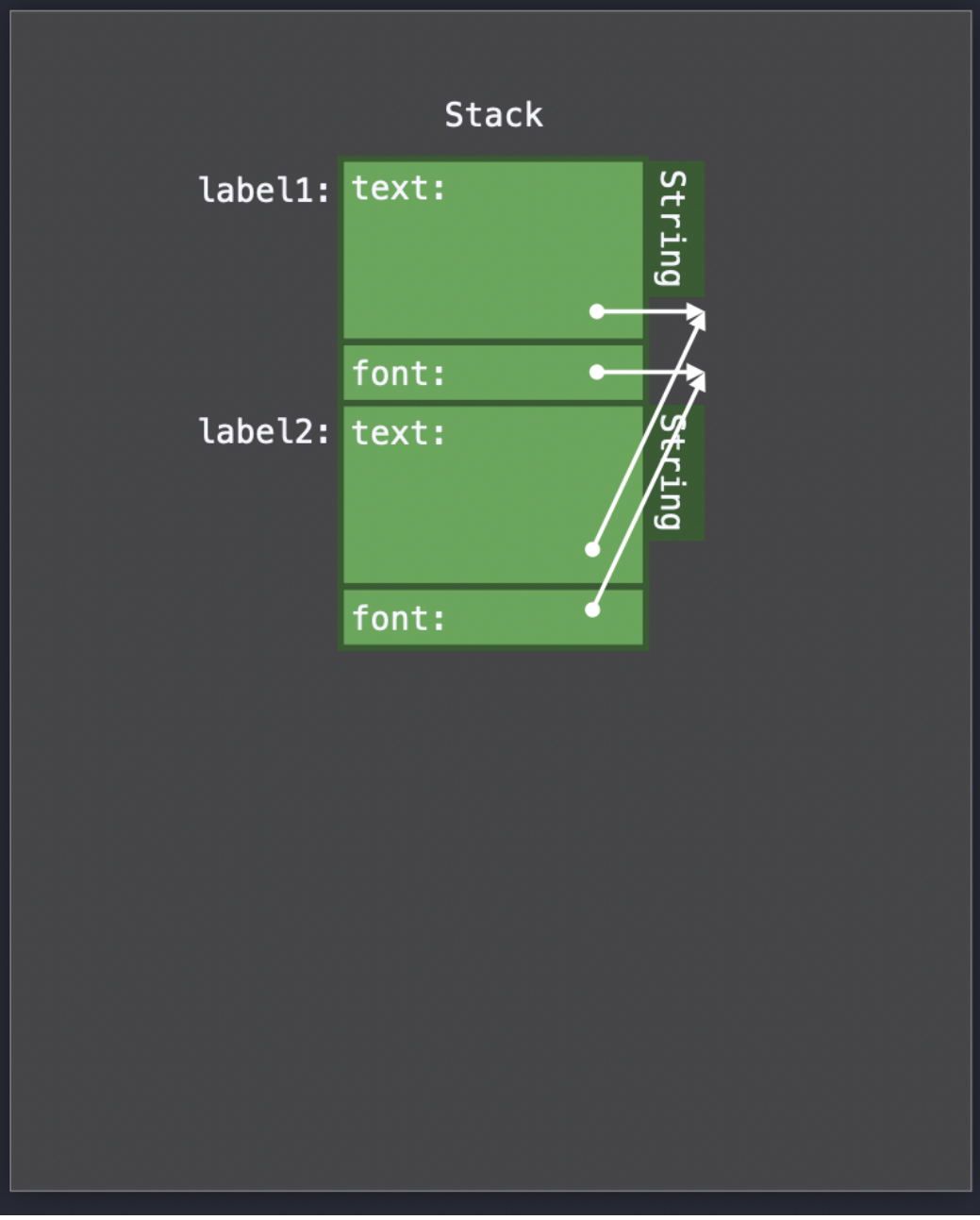
UIFont도 클래스이므로 참조 카운트 관련 작업이 필요하다.

```swift
// Reference Counting
// Struct containing references

struct Label {
    var text: String
    var font: UIFont
    func draw() { … }
}

let label1 = Label(text: "Hi", font: font)
let label2 = label1
// use `label1`
// use `label2`
```



따라서 메모리 표현을 보면 다음과 같이 Label에 두개의 참조가 있는걸 볼 수 있다.

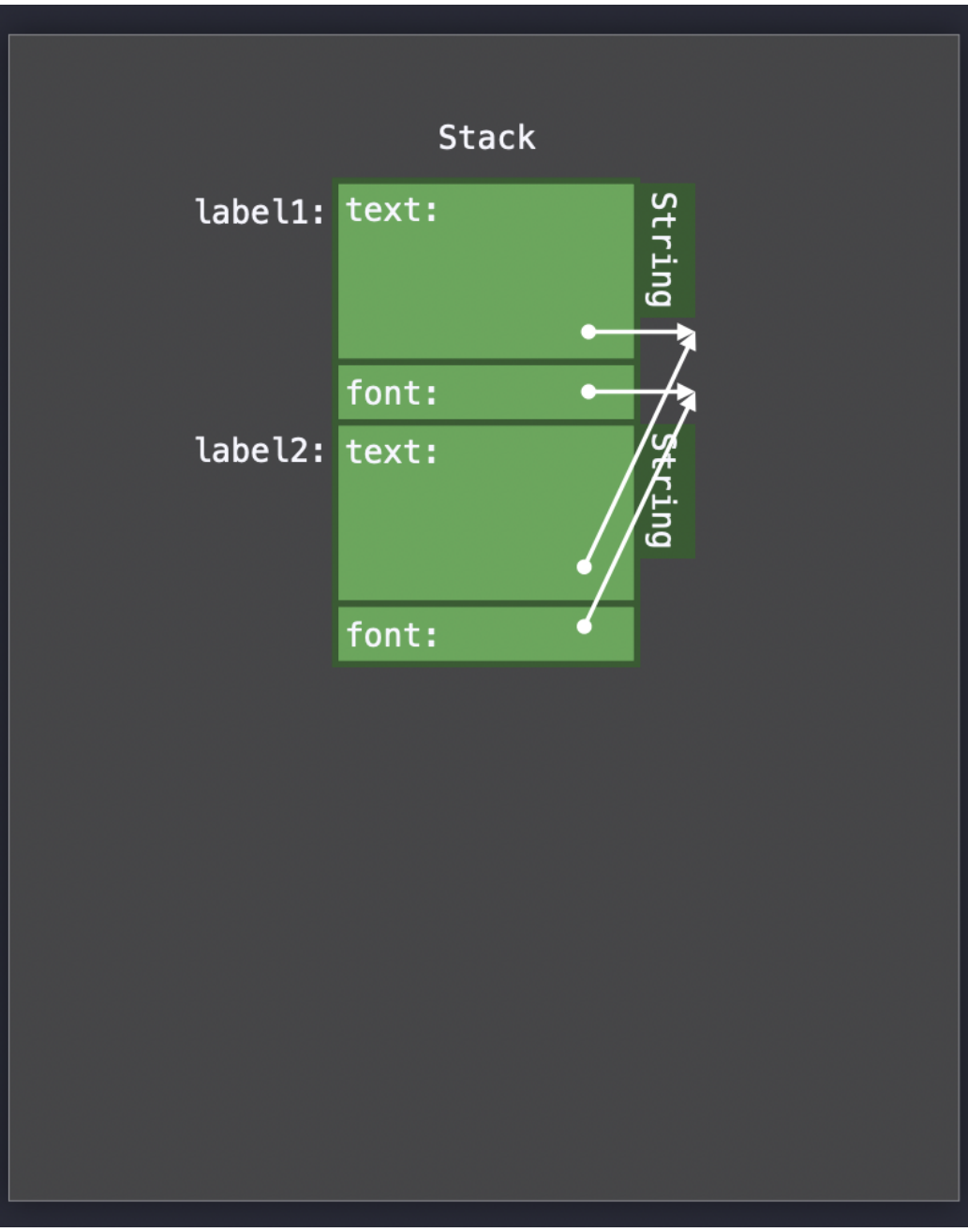
```swift
// Reference Counting
// Struct containing references

struct Label {
    var text: String
    var font: UIFont
    func draw() { … }
}

let label1 = Label(text: "Hi", font: font)
let label2 = label1
// use `label1`
// use `label2`
```



그리고 복사본을 만들 때 실제로 두개의 참조를 더 추가한다.

하나는text저장소, 다른 하나는 font에 추가한다.

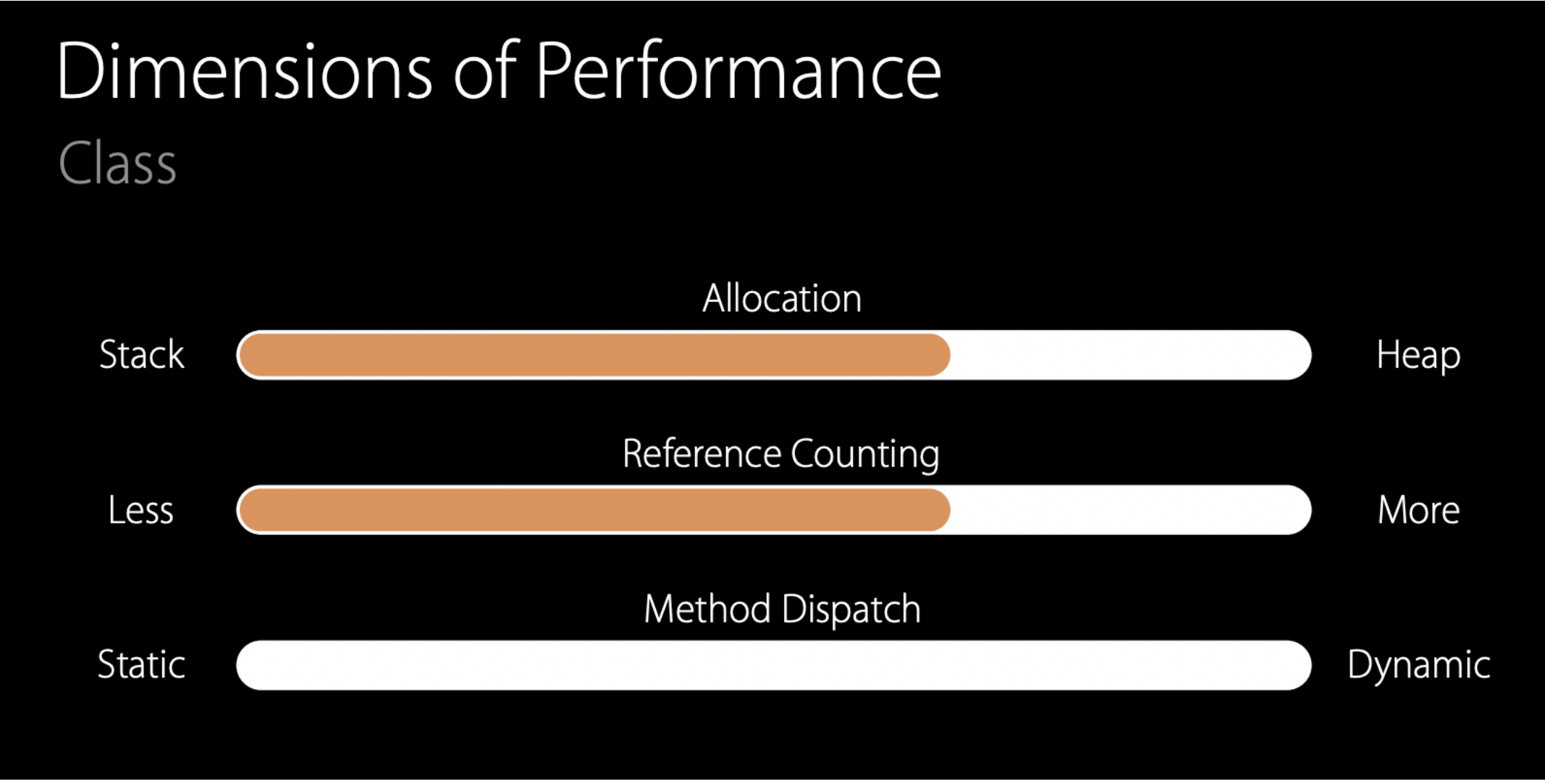
```swift
// Reference Counting
// Struct containing references
   (generated code)
struct Label {
   var text: String
   var font: UIFont
   func draw() { … }
}

let label1 = Label(text: "Hi", font: font)
let label2 = label1
retain(label2.text._storage)
retain(label2.font)
// use `label1`
release(label1.text._storage)
release(label1.font)
// use `label2`
release(label2.text._storage)
release(label2.font)
```



따라서 구조체 내부에 참조 카운팅 작업이 필요한 속성이 존재하고 이를 복사하는 과정을 거칠 경우 위와 같이 retain과 release작업을 각각 해주게 된다.

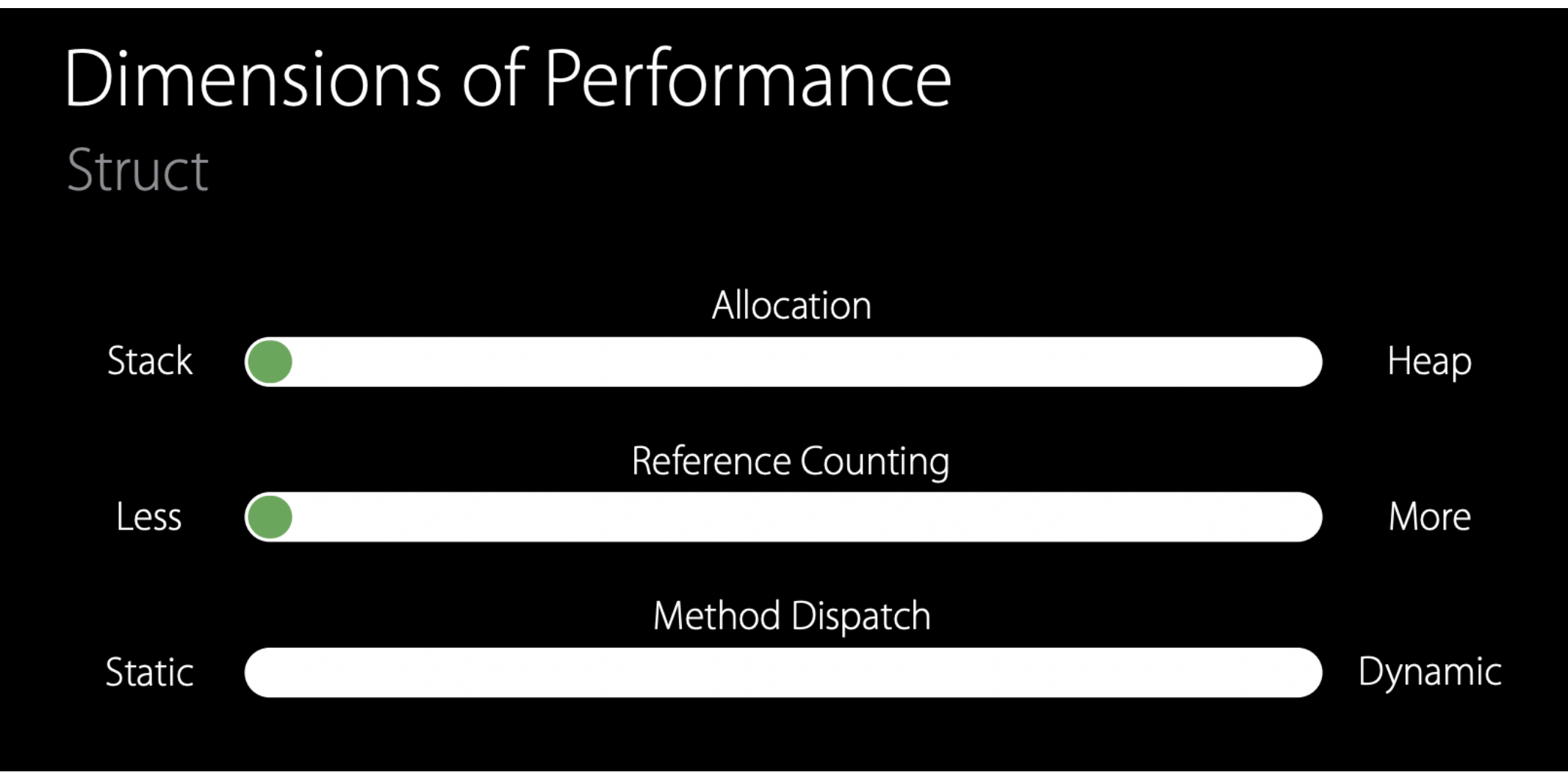
따라서 Label이 실제로 클래스가 갖는 참조 카운팅 오버헤드의 두배를 발생시킬 수 있다.

기존에 클래스는 클래스 자체를 복사햇는데 여기서는 내부 속성수 만큼 참조해야하니까 더 많이 참조가 들어간다는 이야기같음

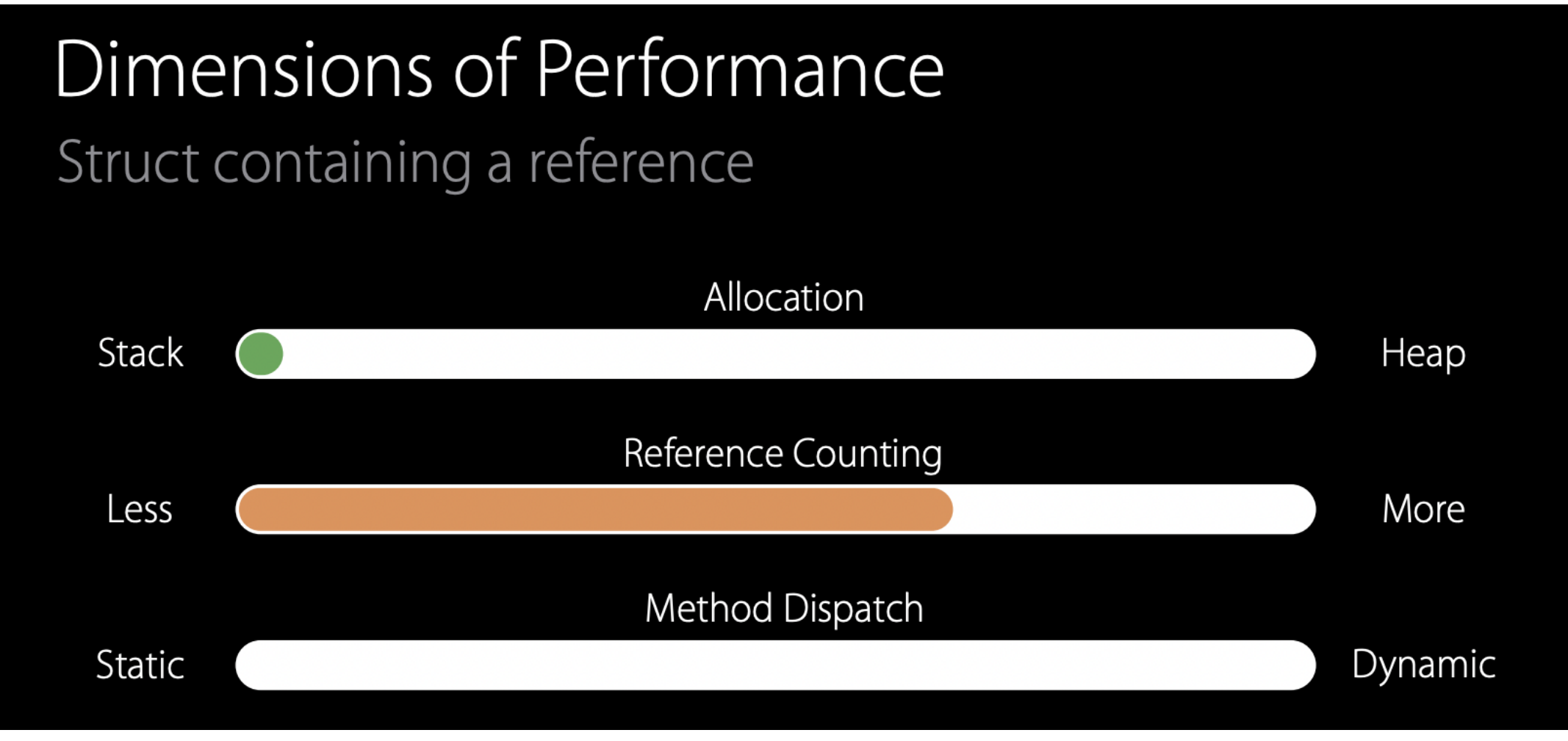


요약 해보자.

클래스는 힙에 할당되기 때문에 Swift는 해당 힙 할당의 수명을 관리할 의무가 있다.
그리고 이를 참조카운팅으로 한다.
다만 참조 카운팅 작업은 빈번하고 원자성을 지켜야 하기에 이것은 사소하지않다.



다시 보아도 이것은 구조체를 사용하라는 의미와 같다.

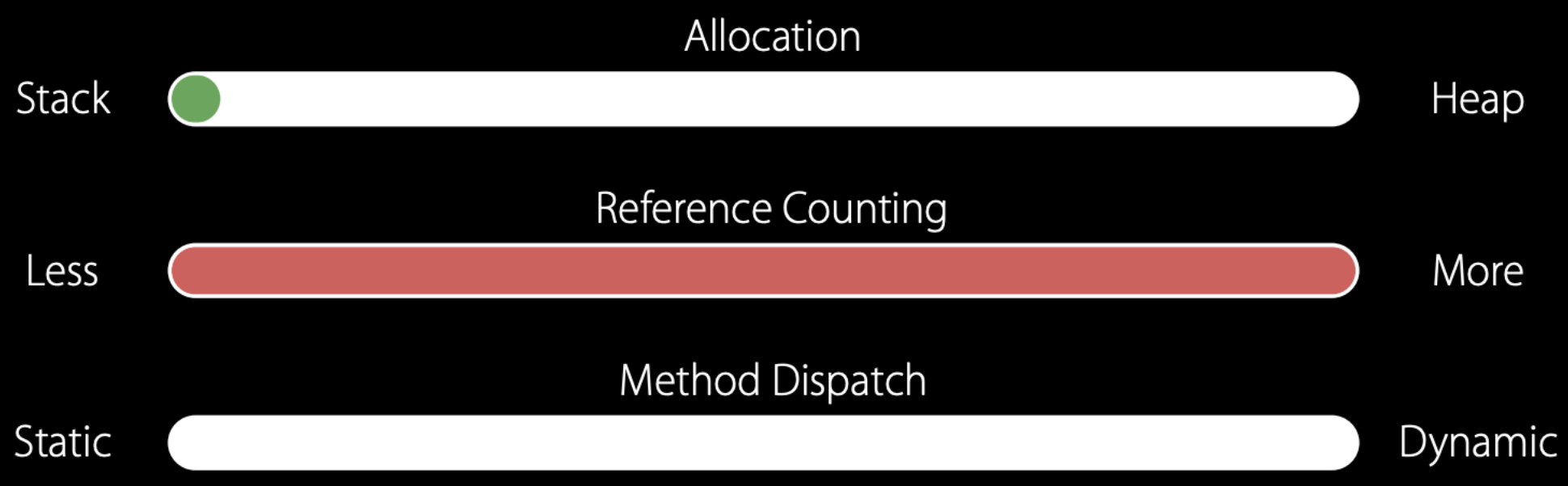


다만 이러한 이야기는 구조체 내부에 참조가 포함되지 않은 경우만이다.

구조체 내부에 참조가 포함된 경우 참조 계산 오버헤드를 지불해야한다.
즉 구조체는 포함하고 있는 참조의 수만큼 오버헤드를 지불하는 것이다

Dimensions of Performance

Struct containing many references



참조가 두개가 넘어가면 오히려 클래스보다 계산 오버헤드가 더 많이 들어갈 수 있다.

```swift
// Modeling Techniques: Reference Counting

struct Attachment {
   let fileURL: URL
   let uuid: String
   let mimeType: String

   init?(fileURL: URL, uuid: String, mimeType: String) {
      guard mimeType.isMimeType
      else { return nil }

      self.fileURL = fileURL
      self.uuid = uuid
      self.mimeType = mimeType
   }
}
```

이제 실제 예시를 보자

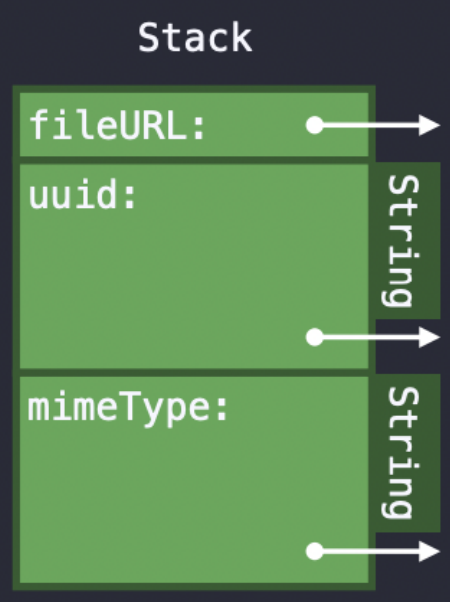
위 코드에 이론적인 메시징 응용프로그램에서 가져온 예제에 어떻게 연결하여 적용하는지 보도록 하자

사용자들은 단순히 문자메시지를 보내는것 뿐만 아니라 이미지와 같은 첨부파일을 보내고 싶어한다고 해보자

그래서 응용프로그램의 모델 객체인 Attachment에는 첨부파일 속성이 있다. 이 첨부파일 에 대한 디스크내 데이터 경로를 나타낼 fileURL 속성이 존재한다.
그리고 식별자인 uuid와 jpg, png등을 구별할 mimType이 존재한다.

이 예제에서 중요하지않은 유일한 코드는 실패 가능한 이니셜라이저라는것.
모든 mimeType을 지원하지 않기 때문에 mimeType이 이 애플리케이션에 지원되는 타입인지 확인해야함 지원되지 않는 경우 작업을 중단함(guard문)

```
// Modeling Techniques: Reference Counting

struct Attachment {
    let fileURL: URL
    let uuid: String
    let mimeType: String

    init?(fileURL: URL, uuid: String, mimeType: String) {
        guard mimeType.isMimeType
        else { return nil }

        self.fileURL = fileURL
        self.uuid = uuid
        self.mimeType = mimeType
    }
}
```

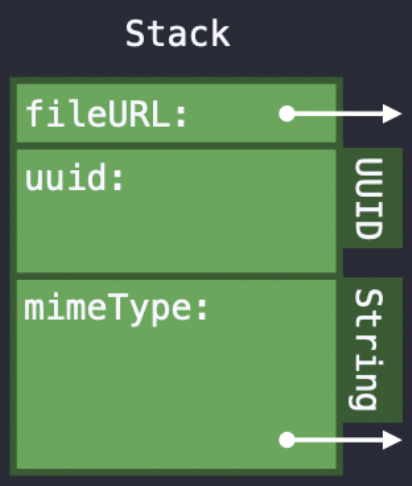


당장 구조체 내부의 속성이 모두 참조 타입이다. 따라서 각 구조체가 힙 할당에 대한 참조가 내포되어 있기 때문에 이를 전달할 때 3가지 프로퍼티 모두 참조 카운팅 오버헤드를 발생시킨다.

이제 이를 개선해보자

```
// Modeling Techniques: Reference Counting

struct Attachment {
    let fileURL: URL
    let uuid: UUID
    let mimeType: String

    init?(fileURL: URL, uuid: UUID, mimeType: String) {
        guard mimeType.isMimeType
        else { return nil }

        self.fileURL = fileURL
        self.uuid = uuid
        self.mimeType = mimeType
    }
}
```

NEW



What's New in Foundation for Swift Mission Tuesday 4:00PM

먼저 UUID는 잘 정의된 식별자이다 이는 128비트로 무작위 생성된다.
다만 String으로 되어있으므로 아무값이나 들어갈 수 있다.

Foundation의 UUID는 값타입이므로 구조체 속성을 UUID 타입으로 변경한다.

문자열을 사용함으로써 발생하는 참조 카운팅 오버헤드를 값 타입인 uuid로 대체하면서 지불해야할 오버헤드를 제거할 수 있다.

또한 uuid만 넣어야 하므로 더 엄격한 안전이 확보된다.

```swift
// Modeling Techniques: Reference Counting

struct Attachment {
    let fileURL: URL
    let uuid: UUID
    let mimeType: String

    init?(fileURL: URL, uuid: UUID, mimeType: String) {
        guard mimeType.isMimeType
        else { return nil }

        self.fileURL = fileURL
        self.uuid = uuid
        self.mimeType = mimeType
    }
}
```

```swift
extension String {
    var isMimeType: Bool {
        switch self {
        case "image/jpeg":
            return true
        case "image/png":
            return true
        case "image/gif":
            return true
        default:
            return false
        }
    }
}
```

그 다음 mimeType이다.

현재는 String의 확장을 통해 mimeType임을 검사하고 있다. 이는 JPG, PNG, GIF의 세트만 지원하고 있고 이는 닫힌 조합이다(closed Set)

swift는 고정된 집합을 표현하기 위한 훌륭한 추상화 매커니즘인 열거형을 지원한다.

```swift
// Modeling Techniques: Reference Counting

struct Attachment {
    let fileURL: URL
    let uuid: UUID
    let mimeType: MimeType

    init?(fileURL: URL, uuid: UUID, mimeType: String) {
        guard let mimeType = MimeType(rawValue: mimeType)
        else { return nil }

        self.fileURL = fileURL
        self.uuid = uuid
        self.mimeType = mimeType
    }
}
```

```swift
enum MimeType {
    init?(rawValue: String) {
        switch rawValue {
        case "image/jpeg":
            self = .jpeg
        case "image/png":
            self = .png
        case "image/gif":
            self = .gif
        default:
            return nil
        }
    }
    case jpeg, png, gif
}
```

switch문을 실패가능한 이니셜라이저 내부에 넣고 적절한 열거형의 case로 매핑한다.

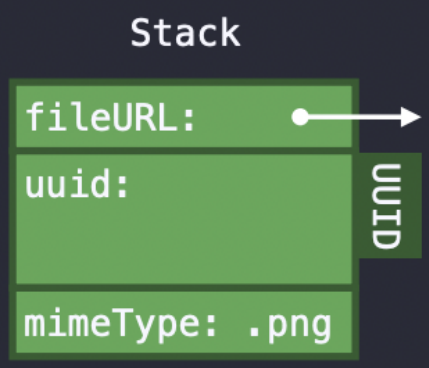
이렇게 되며 MimeType은 열거형이 되며 더 많은 타입안전성을 갖으며 힙에 간접적으로 저장할 필요가 없기에 성능이 향상된다.(열거형도 값타입이므로)

```swift
// Modeling Techniques: Reference Counting

struct Attachment {
    let fileURL: URL
    let uuid: UUID
    let mimeType: MimeType

    init?(fileURL: URL, uuid: UUID, mimeType: String) {
        guard let mimeType = MimeType(rawValue: mimeType)
        else { return nil }

        self.fileURL = fileURL
        self.uuid = uuid
        self.mimeType = mimeType
    }
}
```

```swift
enum MimeType : String {
    case jpeg = "image/jpeg"
    case png = "image/png"
    case gif = "image/gif"
}
```

더 간결하게 사용하기 위해서는 rawString을 사용하는 enum을 사용할 수 있다.
이것은 더 강력하고 동일한 성능을 가지며 작성이 훨씬 편리하다.

```
// Modeling Techniques: Reference Counting

struct Attachment {
    let fileURL: URL
    let uuid: UUID
    let mimeType: MimeType

    init?(fileURL: URL, uuid: UUID, mimeType: String) {
        guard let mimeType = MimeType(rawValue: mimeType)
        else { return nil }

        self.fileURL = fileURL
        self.uuid = uuid
        self.mimeType = mimeType
    }
}
```



기존의 참조 카운팅 오버헤드가 3군데나 발생했었으나 이를 적절한 값타입으로 대체하면서 더 안전해지고 오버헤드를 발생시키지 않는 타입으로 바꿀 수 있었다.

uuid, mimeType는 참조 카운트 되거나 힙할당이 될 필요가 없기 때문이고 이들이 값타입이기 때문에 가능

바드

Method Dispatch

성능의 마지막 차원인 메서드 디스패치로 넘어가보자
메서드를 런타임시 호출할 때 Swift는 올바른 impelmentation을 실행해야 함

Method Dispatch

Static

Jump directly to implementation at run time

Candidate for inlining and other optimizations

컴파일 타임에 실행할 구현을 결정할 수 있는 경우 정적 디스패치라고 함
런타임 시 올바른 구현으로 바로 이동할 수 있음
컴파일러가 실제로 어떤 구현이 실행될 것인가에 대해 가시성을 가질 수 있기 때문에 쿨함
Inlining 같은 것들을 포함하여 매우 최적화할 수 있음
이것은 동적 디스패치와 대조됨

Method Dispatch

Dynamic

Look up implementation in table at run time

Then jump to implementation

Prevents inlining and other optimizations

동적 디스패치는 어떤 구현으로 이동할것인지 컴파일 시간에 결정할 수 없음
런타임시 구현할 곳을 찾아서 바로 실행함
동적 디스패치 자체로는 정적 디스패치보다 비용이 많이 들지 않음
한가지 수준의 간접 참조가 있음
참조 카운팅 및 힙 할당과 같은 스레드 동기화 오버헤드는 없음
컴파일러는 정적 디스패치에 대해 최적화를 수행할 수 있지만, 컴파일러의 가시성을 차단하므로 최적화를 할 수 없음

```swift
// Method Dispatch
// Struct (inlining)

struct Point {
   var x, y: Double
   func draw() {
      // Point.draw implementation
   }
}

func drawAPoint(_ param: Point) {
   param.draw()
}

let point = Point(x: 0, y: 0)
drawAPoint(point)
```

앞에서 언급한 inlining에 대해서 알아보자
구조체 Point는 x, y 프로퍼티가 있고 draw 메서드가 있음
drawAPoint는 Point를 가져와서 draw를 호출함

```swift
// Method Dispatch
// Struct (inlining)

struct Point {
   var x, y: Double
   func draw() {
      // Point.draw implementation
   }
}

func drawAPoint(_ param: Point) {
   param.draw()
}

let point = Point(x: 0, y: 0)
drawAPoint(point)
```

point프로퍼티는 0, 0에 Point를 구성하고 그 point를 drawAPoint에 전달함
drawAPoint 메서드와 point.draw 메서드는 모두 정적으로 dispatch함

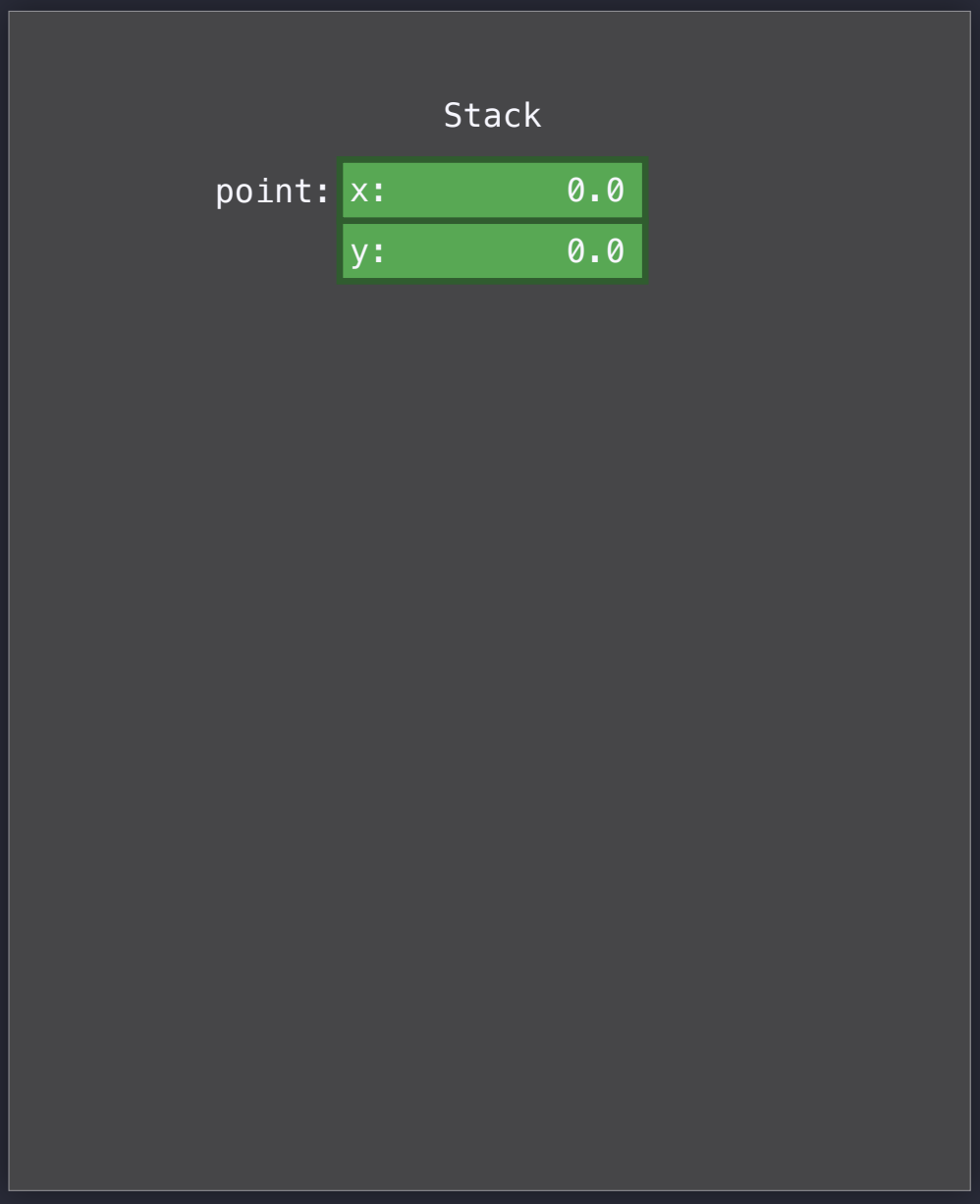
```swift
// Method Dispatch
// Struct (inlining)

struct Point {
    var x, y: Double
    func draw() {
        // Point.draw implementation
    }
}

func drawAPoint(_ param: Point) {
    param.draw()
}

let point = Point(x: 0, y: 0)
point.draw()
```

이 뜻은 컴파일러가 어떤 구현이 실행될 지 알고 있음
실제로 drawAPoint 디스패치를 가져오고 그것을 drawAPoint의 구현부인 point.draw()로 대체한다는 것임

```swift
// Method Dispatch
// Struct (inlining)

struct Point {
    var x, y: Double
    func draw() {
        // Point.draw implementation
    }
}

func drawAPoint(_ param: Point) {
    param.draw()
}

let point = Point(x: 0, y: 0)
// Point.draw implementation
```

그러면 point.draw() 메서드를 사용함
이는 정적 디스패치이므로 point.draw의 실제 구현부로 대체할 수 있음

```swift
// Method Dispatch
// Struct (inlining)

struct Point {
    var x, y: Double
    func draw() {
        // Point.draw implementation
    }
}

func drawAPoint(_ param: Point) {
    param.draw()
}

let point = Point(x: 0, y: 0)
// Point.draw implementation
```



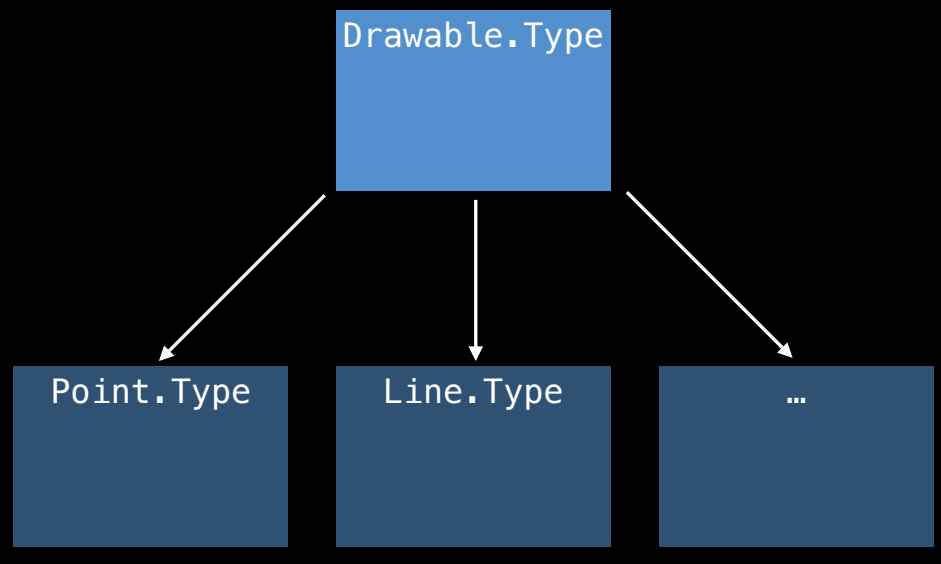
런타임시 이 코드를 실행하면 그냥 Point를 만든 다음 함수 구현부를 실행함
두 가지 정적 디스패치 오버헤드와 call stack의 설정 및 해제 관련 두 가지 오버헤드가 필요하지 않았음
이것은 왜 정적 디스패치가 동적 디스패치보다 빠른지 알려줌

단일 동적 디스패치와 비교하여 단일 정적 디스패치가 큰 차이점을 가지지는 않음
하지만 전체 정적 디스패치 체인은 컴파일러가 전체 체인을 통해 가시성을 가지게 됨
반면, 동적 디스패치 체인은 더 높은 레벨의 모든 추론단계에서 차단될 것
따라서 컴파일러는 call stack 오버헤드가 없는 단일 구현으로 정적 메서드 디스패치 체인을 축소할 수 있음

We didn’t need those two – the overhead of those two static dispatches and the associated setting up of the call stack and tearing it down

Inheritance-Based Polymorphism

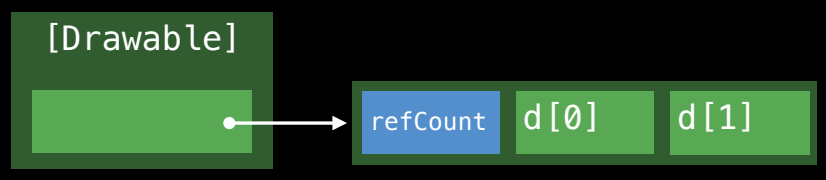
```
class Drawable { func draw() {} }

class Point : Drawable {
   var x, y: Double
   override func draw() { … }
}
class Line : Drawable {
   var x1, y1, x2, y2: Double
   override func draw() { … }
}

var drawables: [Drawable]
for d in drawables {
   d.draw()
}
```



그러면 왜 우리는 동적 디스패치를 가지고 있나?
그 이유중 하나는 다형성 같은 강력한 것들을 가능하게 하기때문
위의 object-oriented program로 구현된 Drawble 추상클래스를 보면
고유한 커스텀 구현으로 draw 메서드를 재정의하여 Point 하위클래스와 Line
하위클래스를 정의할 수 있음

Polymorphism Through Reference Semantics

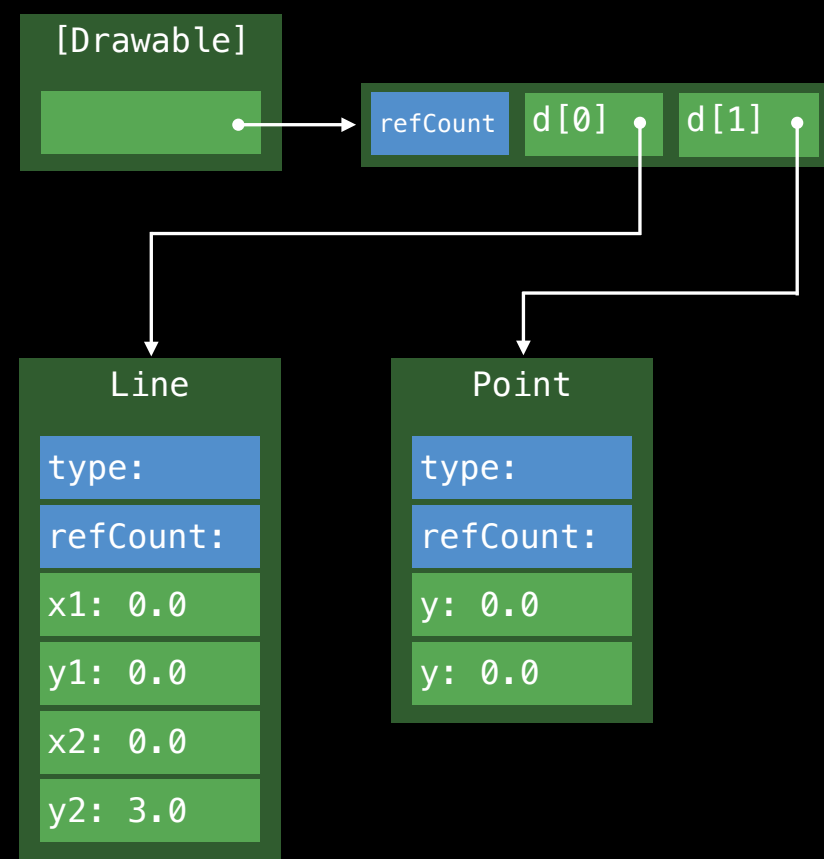
```
class Drawable { func draw() {} }

class Point : Drawable {
   var x, y: Double
   override func draw() { … }
}
class Line : Drawable {
   var x1, y1, x2, y2: Double
   override func draw() { … }
}

var drawables: [Drawable]
for d in drawables {
   d.draw()
}
```



그리고 Drawble 배열을 Line을 포함할 수도 있고, Point를 포함할수도 있는
다형성으로 생성하고 있음
그리고 각자에 대한 draw를 호출할 수 있음
어떻게 가능한가?

Polymorphism Through Reference Semantics

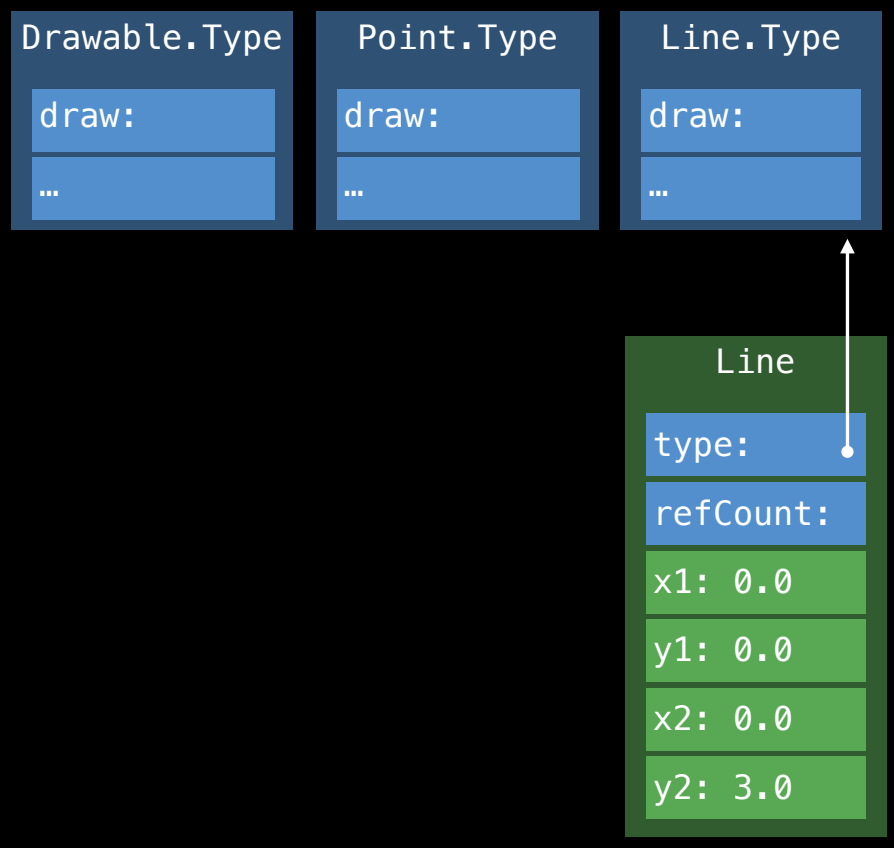
```
class Drawable { func draw() {} }

class Point : Drawable {
    var x, y: Double
    override func draw() { … }
}
class Line : Drawable {
    var x1, y1, x2, y2: Double
    override func draw() { … }
}

var drawables: [Drawable]
for d in drawables {
    d.draw()
}
```



Drawable, Point, Line이 모두 클래스이기에 가능함
우리는 이들의 배열을 만들 수 있고 그들은 참조로 저장하기에 크기가 모두 동일함
그리고 각자에 대한 draw 메서드를 호출함

Polymorphism Through V-Table Dispatch

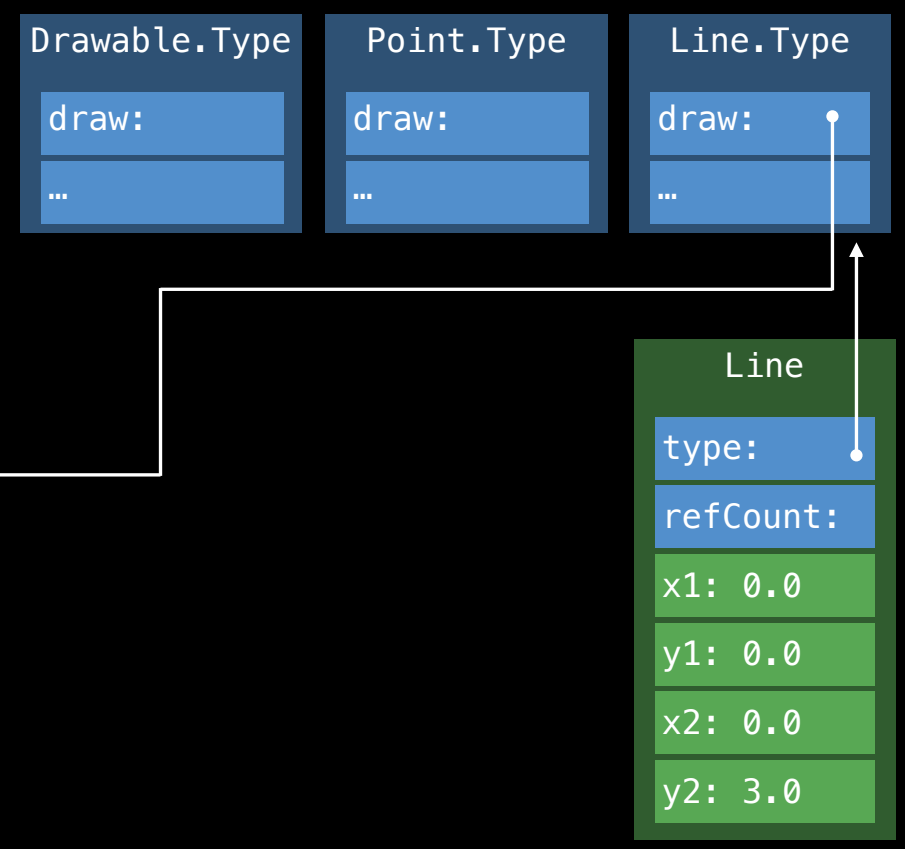
```
class Drawable { func draw() {} }

class Point : Drawable {
   var x, y: Double
   override func draw() { … }
}
class Line : Drawable {
   var x1, y1, x2, y2: Double
   override func draw() { … }
}

var drawables: [Drawable]
for d in drawables {
   d.draw()
}
```



여기서 우리는 컴파일러가 컴파일 타임에 실행할 올바른 구현을 결정할 수 없는 이유를 이해할 수 있고 직관을 가질 수 있음
여기서 d.draw는 Point가 될 수도 있고, Line이 될수도 있음
이들은 서로 다른 코드 경로를 가지고 있다
이 중 어떤 것을 호출할지 어떻게 결정하는가?

Polymorphism Through V-Table Dispatch

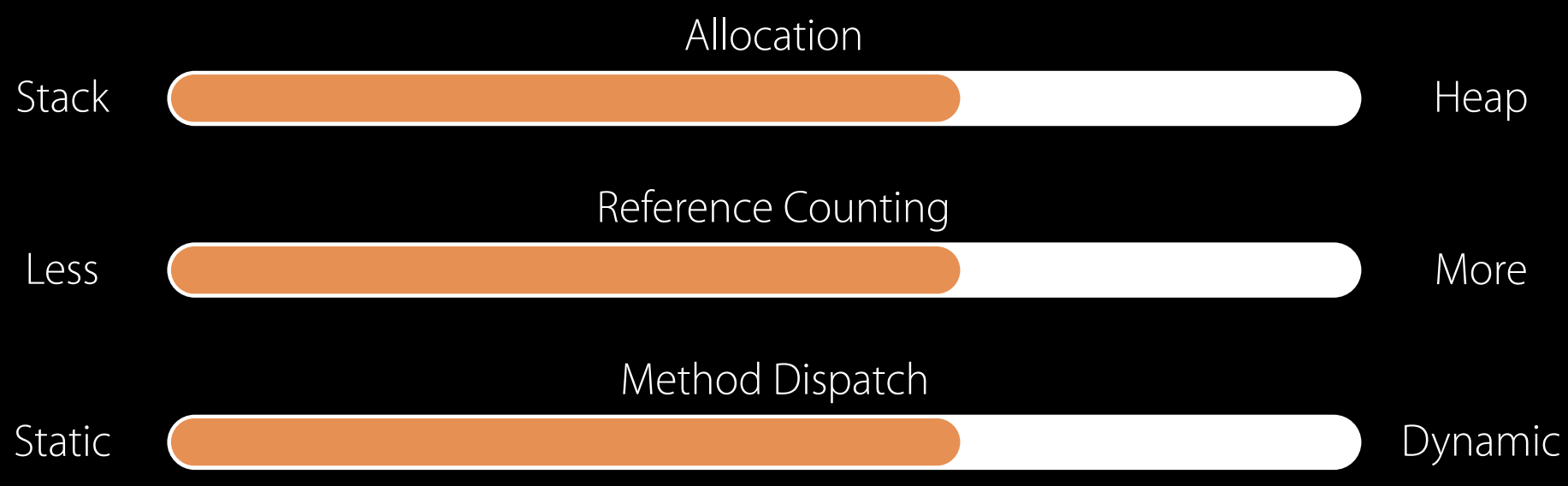
```
class Drawable { func draw() {} }

class Point : Drawable {
    var x, y: Double
    override func draw() { … }
}
class Line : Drawable {
    var x1, y1, x2, y2: Double
    override func draw(_ self: Line) { … }
}

var drawables: [Drawable]
for d in drawables {
    d.type.vtable.draw(d)
}
```



컴파일러는 다른 영역에 해당 클래스의 타입 정보에 대한 포인터를 클래스를 추가하고, 그것은 정적 메모리에 저장됨
그리고 draw 메서드를 호출할 때 컴파일러가 실제로 우리를 대신해서 생성하는 것은 가상 메서드 테이블이라고 불리는 것(타입 및 정적 메모리에 있는 것)을 조회하는 것
이 가상 메서드 테이블에는 실행할 올바른 구현에 대한 포인터가 포함되어 있음
-> 어떤 구현사항을 실행해야 할지 메타타입이 알고 있으며 이를 컴파일러가 조회
따라서 이 d.draw를 컴파일러가 우리를 대신하여 수행하는 작업으로 변경하면 실제로 실행할 올바른 draw 구현을 찾기 위해 가상 메서드 테이블을 검색하는 것을 볼 수 있음
그 다음 실제 인스턴스를 암시적 self 파라미터로 전달함

Dimensions of Performance

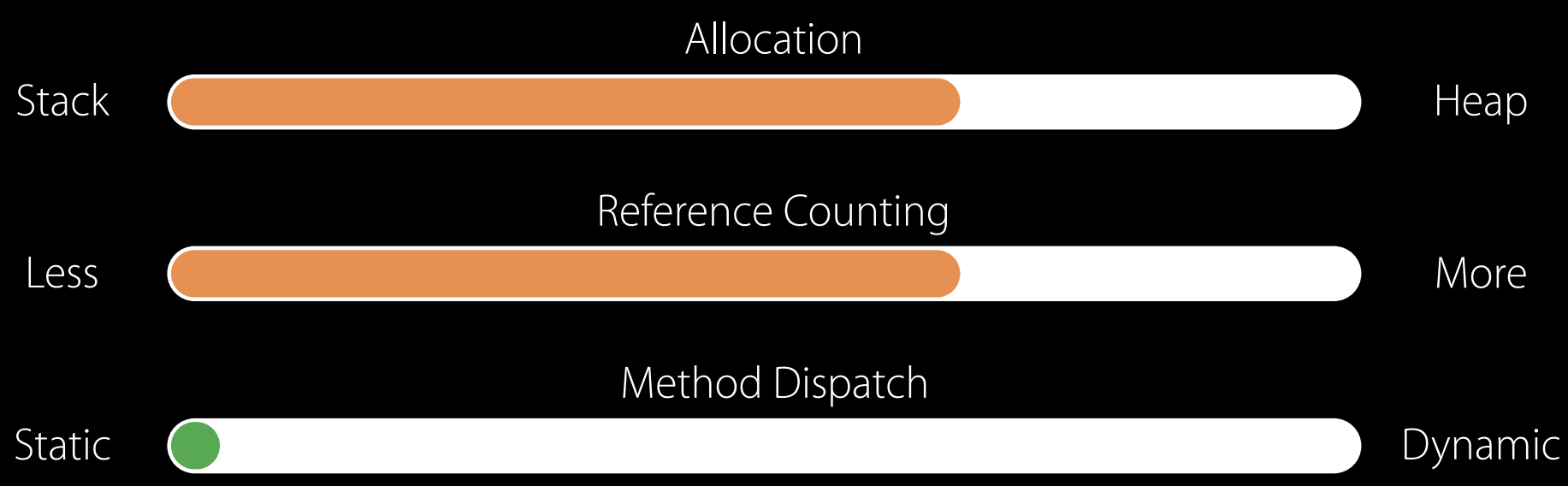
Class



클래스는 기본적으로 메서드를 동적으로 디스패치함
이것은 그 자체로 큰차이를 만들지 않지만, 메서드 체인 및 다른 것들과 관련되었을 때 inlining과 같은 최적화를 방지할 수 있고 더해질 수 있음

Dimensions of Performance

Final Class

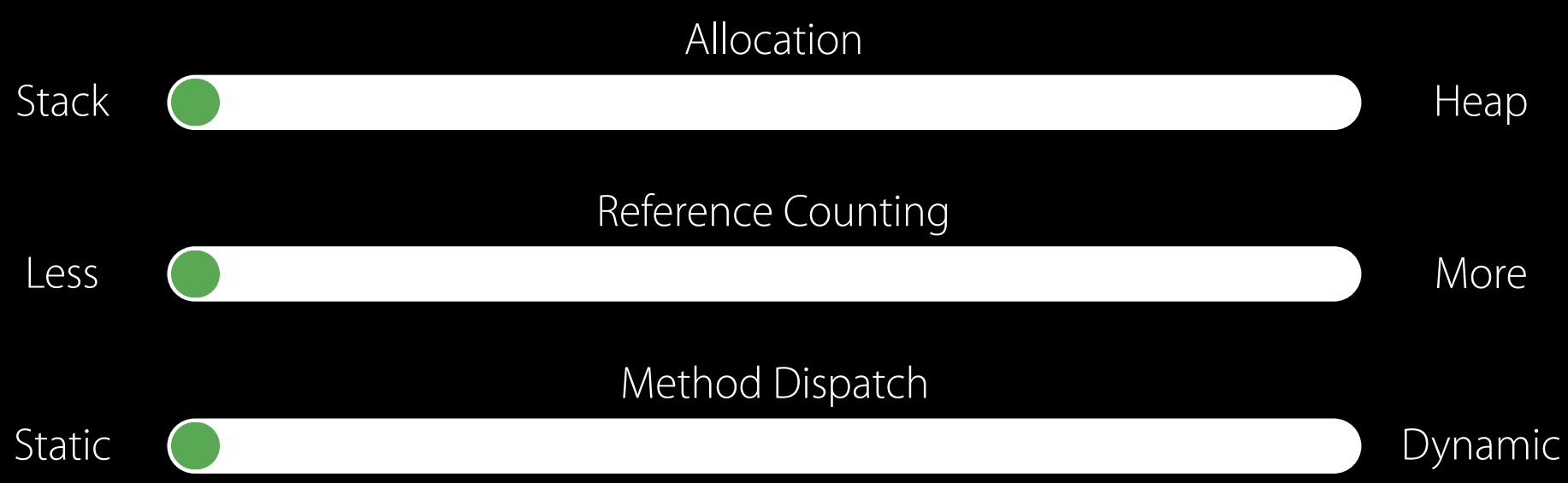


Optimizing Swift Performance WWDC 2015

모든 클래스가 동적 디스패치가 필요한 것은 아님
클래스를 하위 클래스를 만들 필요가 없다면 final 키워드를 표시하여 의도를 전달할 수 있음
또한 컴파일러가 앱에서 클래스를 상속하지 않을 증거와 근거ㅇ가 있다면, 다이나믹 디스패치에서 정적 디스패치로 전환함
이에 대해 더 알고 싶다면 작년의 Optimizing Swift Performance를 봐라

Dimensions of Performance

Struct



그렇다면 이제 무엇이 남았는가?
이 강연의 전반부에서 자기 자신에게 질문을 해야하는 것이 있음
Swift를 읽고 쓸 때, 이 인스턴스가 스택에 할당될 지, 힙에 할당될 지
이 인스턴스를 전달할 때 오버헤드가 포함된 참조가 얼마나 발생할 지
이 인스턴에서 메서드를 호출하면 해달 메서드가 정적 또는 동적으로 디스패치 되는지
만약 우리가 필요없는 역동성을 위해 비용을 지불한다면 성능이 저하될 것

Swift를 처음 사용하거나 Objective C에서 Swift로 변환된 코드에서 작업한다면 현재보다 구조체를 더 많이 사용할 수 있음

구조체를 사용하여 다형성 코드를 작성하는 방법은 무엇인가?
정답은 Protocol-Oriented-Programming임

Protocol Types

Arnold Schwaighofer Swift Performance Engineer

프로토콜 타입 및 프로토콜 타입과 함께 Generic 코드를 구현하는 방법에 대해 알아보자
우리는 프로토콜 타입의 변수가 저장되고 복사되는 방식과 메서드 디스패치가 작동하는 방법에 대해 알아볼 것

Protocol Types

Polymorphism without inheritance or reference semantics

```
protocol Drawable { func draw() }

struct Point : Drawable {
   var x, y: Double
   func draw() { … }
}
struct Line : Drawable {
   var x1, y1, x2, y2: Double
   func draw() { … }
}

var drawables: [Drawable]
for d in drawables {
   d.draw()
}
```

Drawable이라는 기본 추상 클래스 대신 draw 메서드를 선언한 Drawable 프로토콜을 가지고 있음
그리고 이 Drawable을 채택한 구조체 Point, Line이 있음

Protocol Types

Polymorphism without inheritance or reference semantics

```
protocol Drawable { func draw() }

struct Point : Drawable {
   var x, y: Double
   func draw() { … }
}
struct Line : Drawable {
   var x1, y1, x2, y2: Double
   func draw() { … }
}

var drawables: [Drawable]
for d in drawables {
   d.draw()
}
```

```
class SharedLine : Drawable {
   var x1, y1, x2, y2: Double
   func draw() { … }
}
```

우리는 프로토콜을 준수하는 SharedLine이라는 클래스도 가질 수 있음
그러나 클래스로 인해 reference semantic이 의도치 않은 공유를 발생시킬 수 있으니 이 클래스는 제외할 것임

Protocol Types

Polymorphism without inheritance or reference semantics

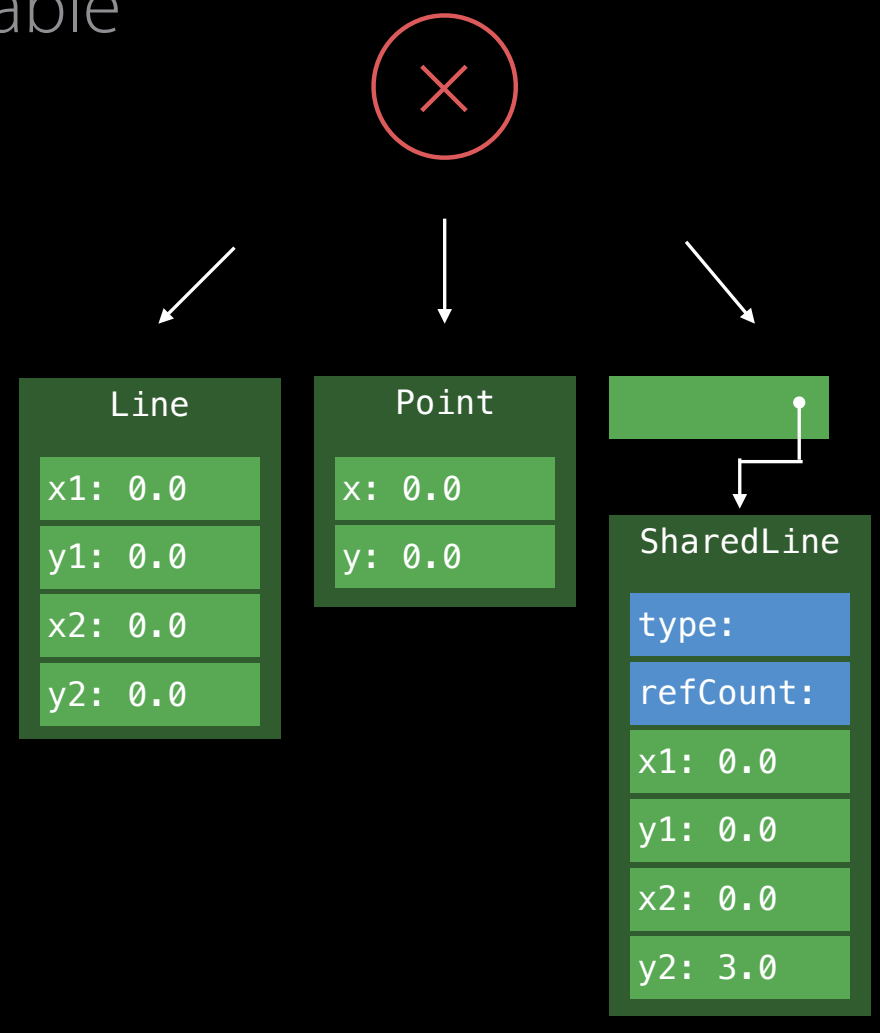
```
protocol Drawable { func draw() }

struct Point : Drawable {
   var x, y: Double
   func draw() { … }
}
struct Line : Drawable {
   var x1, y1, x2, y2: Double
   func draw() { … }
}

var drawables: [Drawable]
for d in drawables {
   d.draw()
}
```

우리 프로그램은 여전히 다형성임
drawables 배열에 Line, Point 타입의 값이 모두 저장할 수 있음
하지만 이전과 비교하여 한가지 다른점이 있음

No Inheritance Relationship

Dynamic dispatch without a V-Table

```
protocol Drawable { func draw() }

struct Point : Drawable {
    var x, y: Double
    func draw() { … }
}
struct Line : Drawable {
    var x1, y1, x2, y2: Double
    func draw() { … }
}

var drawables: [Drawable]
for d in drawables {
    d.draw()
}
```

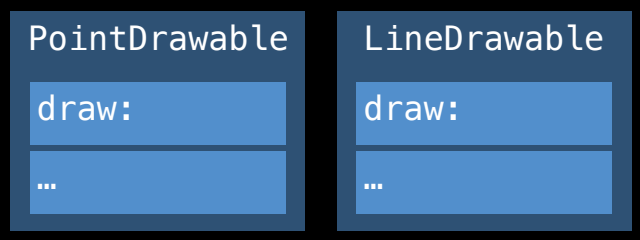


값 타입인 구조체 Line, Point는 가상 메서드 테이블디스패치를 수행하는데
필요한 공통 상속 관계를 가지고 있지 않음
그렇다면 Swift는 어떻게 올바른 메서드로 디스패치하나

The Protocol Witness Table (PWT)

Dynamic dispatch without a V-Table

```
protocol Drawable {
    func draw()
}
struct Point : Drawable {
    func draw() { … }
}
struct Line : Drawable {
    func draw() { … }
}
```

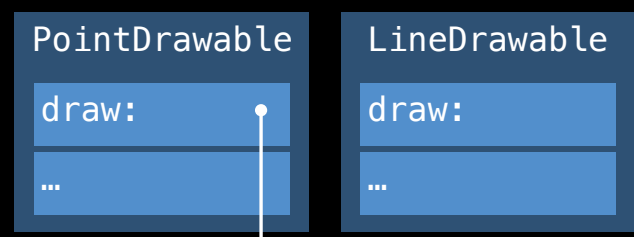


프로토콜은 Protocol Witness Table이라는 테이블 기반 메커니즘
앱에서 프로토콜로 구현한 타입은 이러한 테이블을 하나씩 가짐

The Protocol Witness Table (PWT)

Dynamic dispatch without a V-Table

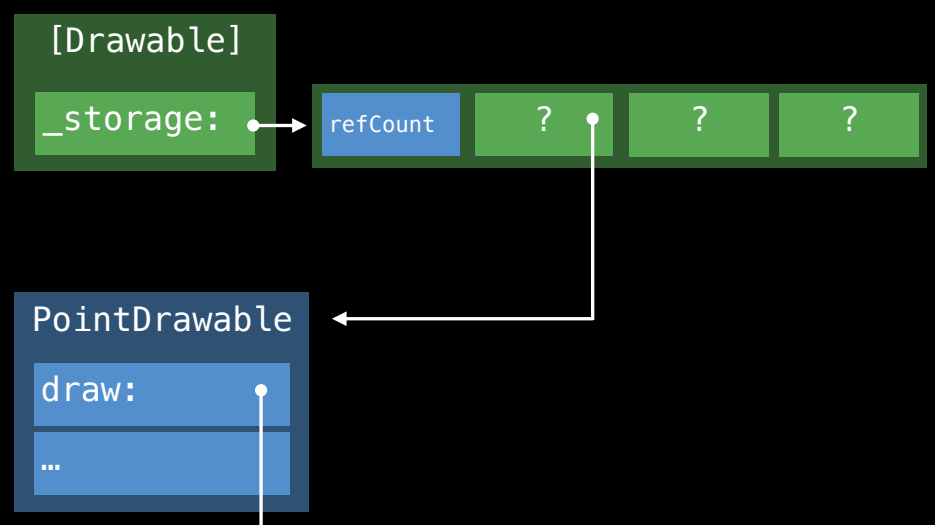
```
protocol Drawable {
    func draw()
}
struct Point : Drawable {
    func draw() { … }
}
struct Line : Drawable {
    func draw() { … }
}
```



해당 테이블 전체는 타입 구현에 연결되어 있음
이 방법으로 메서드를 찾는 방법을 알게 되었음

How to Look Up the Protocol Witness Table?

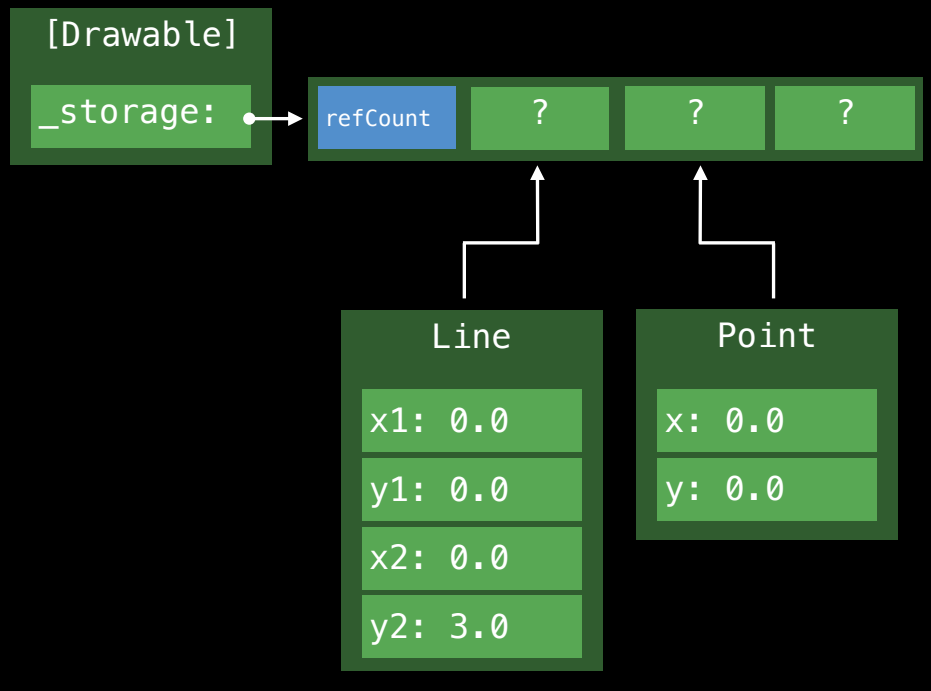
```
protocol Drawable { func draw() }

struct Point : Drawable {
    var x, y: Double
    func draw() { … }
}
struct Line : Drawable {
    var x1, y1, x2, y2: Double
    func draw() { … }
}

var drawables: [Drawable]
for d in drawables {
    d.draw()
}
```



그렇다면 배열에서는 어떻게 찾나?

How to Store Values Uniformly?

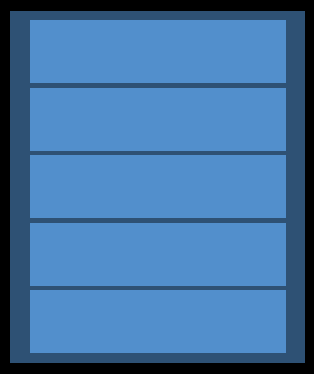
```
protocol Drawable { func draw() }

struct Point : Drawable {
    var x, y: Double
    func draw() { … }
}
struct Line : Drawable {
    var x1, y1, x2, y2: Double
    func draw() { … }
}

var drawables: [Drawable]
for d in drawables {
    d.draw()
}
```



값 타입인 Line과 Point가 있음
Line은 4 word가 필요하고, Point는 2 word가 필요함
이들은 같은 사이즈를 가지고 있지 않음
하지만 우리 배열은 요소들을 고정된 offset에 동일하게 저장할려고 함

The Existential Container

Boxing values of protocol types

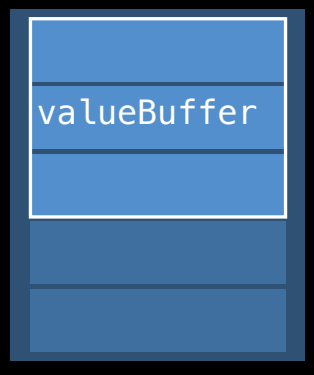


Swift는 특별한 저장소인 Existential Container를 사용함

The Existential Container

Boxing values of protocol types

Inline Value Buffer: currently 3 words

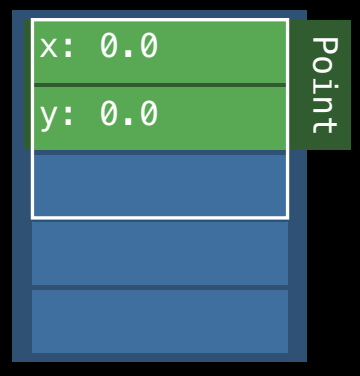


처음 세 word는 valueBuffer용으로 예약되어있음

The Existential Container

Boxing values of protocol types

Inline Value Buffer: currently 3 words



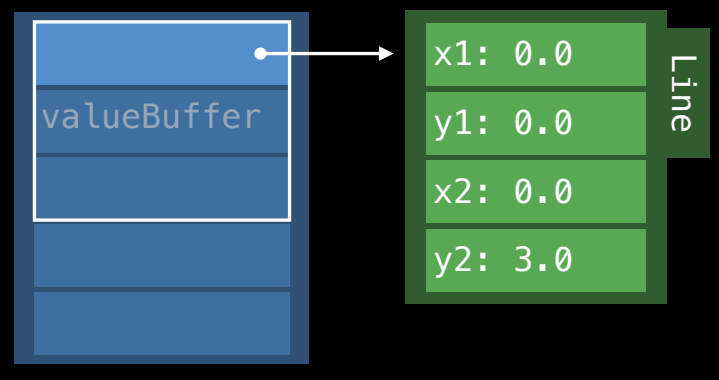
Point는 두 word만 필요해 valueBuffer에 맞음
그렇다면 4 word가 필요하 Line은?

The Existential Container

Boxing values of protocol types

Inline Value Buffer: currently 3 words

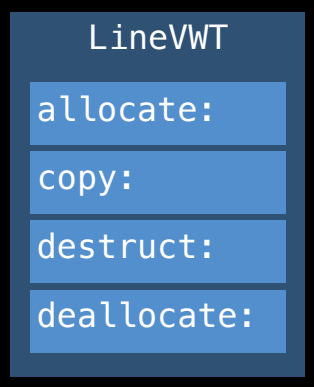
Large values stored on heap



이러한 경우 Swift는 힙에 메모리를 할당하고 해당 메모리에 대한 포인터를
Existential Container에 저장함
word 수의 차이에 따라 관리하는 방법이 다른 것을 알게 됨
그렇다면 이 차이를 어떻게 관리하나
다시 테이블 기반 메커니즘

The Value Witness Table (VWT)

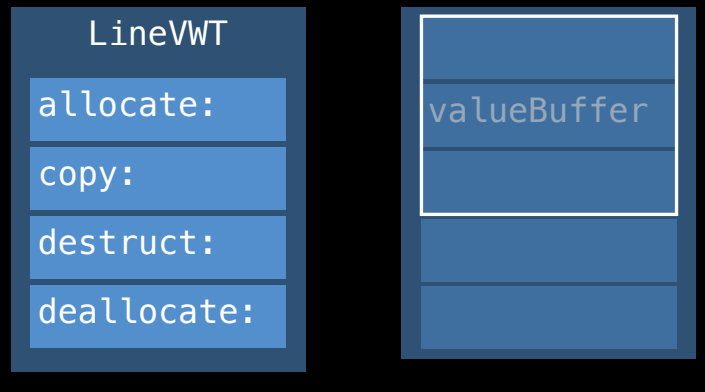
Allocation, Copy, Destruction of any Value



이 경우 Value Witness Table이라고 부름
Value Witness Table은 value 수명을 관리하며 타입별로 이러한 테이블 중 하나를 가짐
지역 변수의 lifetime 경우 어떻게 작동하는지 보자

The Value Witness Table (VWT)

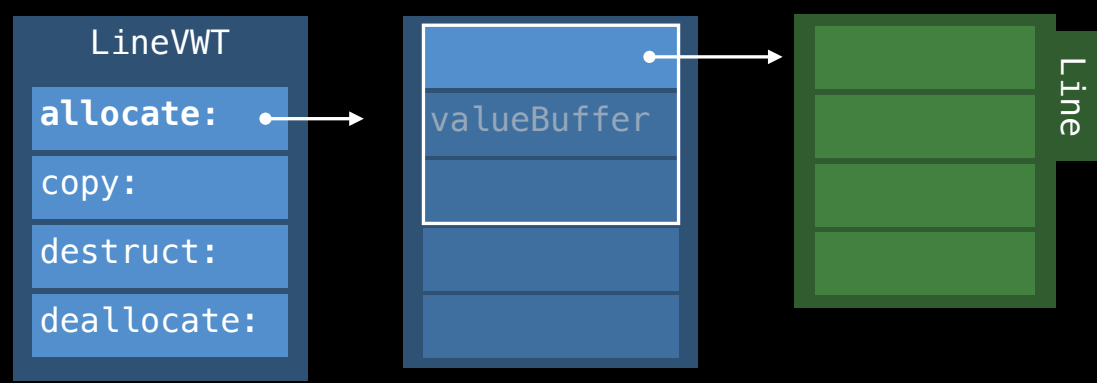
Allocation, Copy, Destruction of any Value



프로토콜 타입의 지역변수의 lifetime이 시작될 때
Swift는 해당 테이블 내부에서 allocate 함수를 호출함
이 함수는 이제 Line Value Witness Table을 가지고 있기 때문에,
힙에 메모리를 할당하고 existential container의 valueBuffer안에 해당 메모리에 대한 포인터를 저장함

The Value Witness Table (VWT)

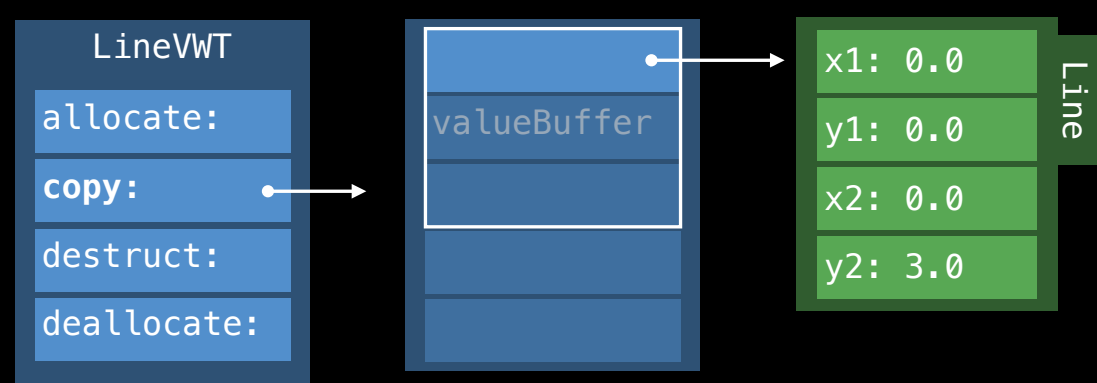
Allocation, Copy, Destruction of any Value



다음으로 Swift는 지역 변수를 초기화하는 allocate의 원본에서 existential container로 값을 복사함

The Value Witness Table (VWT)

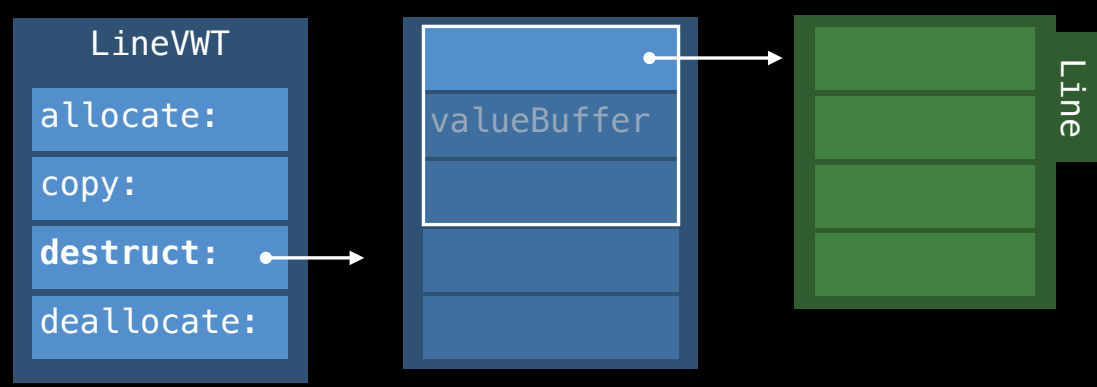
Allocation, Copy, Destruction of any Value



Line은 여기 있으므로 Value Witness Table의 복사본 항목은 다음과 같음

The Value Witness Table (VWT)

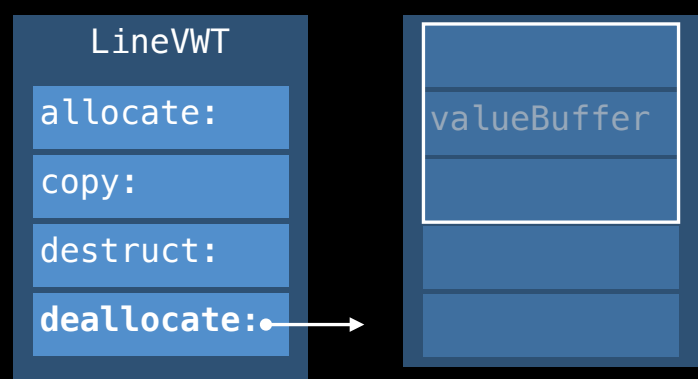
Allocation, Copy, Destruction of any Value



이제 지역 변수의 lifetime이 다했을 때임
Swift는 destruct라는 함수를 호출함
그러면 해당 타입이 포함되어있는 참조 카운트를 감소시킴

The Value Witness Table (VWT)

Allocation, Copy, Destruction of any Value



Line은 이제 아무것도 가지고 있지 않음
힙에 할당된 우리의 값은 할당 해제가 될 것임
다른 값들에 대해 Swift가 다루는 방법에 대해 알아보았음
이제 지역 변수의 lifetime이 다했을 때임
Swift는 destruct라는 함수를 호출함
그러면 해당 타입이 포함되어있는 참조 카운트를 감소시킴
하지만 여전히 어떻게 테이블에 접근하는지 의문임

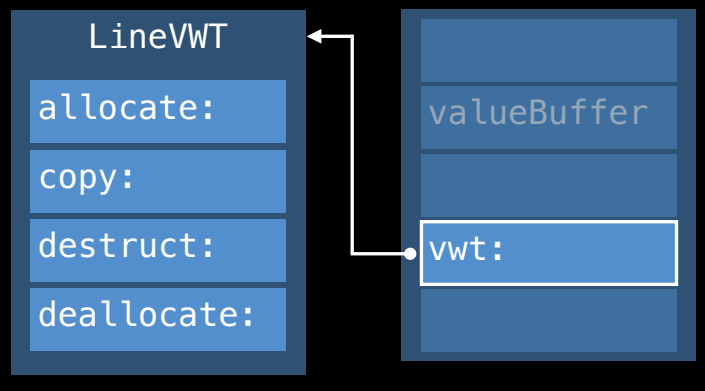
The Existential Container

Boxing values of protocol types

Inline Value Buffer: currently 3 words

Large values stored on heap

Reference to Value Witness Table



Value Witness Table은 참조
existential containter에는 value witness table에 대한 참조가 있음

The Existential Container

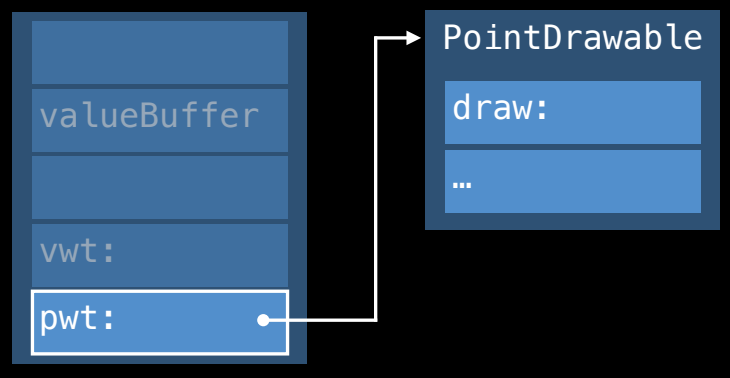
Boxing values of protocol types

Inline Value Buffer: currently 3 words

Large values stored on heap

Reference to Value Witness Table

Reference to Protocol Witness Table



마지막으로 Protocol Witness Table에 도달하는 방법은 무엇인가?
existential container에서 참조가 됨
다음은 예시를 보자

```swift
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
   local.draw()
}
let val : Drawable = Point()
drawACopy(val)
```

프로토콜 타입 Drawable을 매개변수로 받고 draw 메서드를 실행하는 함수,
지역변수가 있고 Point로 초기화함
그리고 이 지역변수를 drawACopy 함수 호출의 파라미터로 전달

```swift
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
   local.draw()
}
let val : Drawable = Point()
drawACopy(val)
```

```swift
// Generated code
struct ExistContDrawable {
   var valueBuffer: (Int, Int, Int)
   var vwt: ValueWitnessTable
   var pwt: DrawableProtocolWitnessTable
}
```

Swift 컴파일러가 생성하는 코드를 설명하기 위해 아래쪽에 pseudocode notation(의사코드 표기법)으로 Swift를 사용할 것임
Existential Containter를 위해 valueBuffer, vwt, pwt로 구성된 ExistContDrawable 구조체를 만들어줌

```swift
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
    local.draw()
}
let val : Drawable = Point()
drawACopy(val)
```

```swift
// Generated code
func drawACopy(val: ExistContDrawable) {
```

drawACopy 함수 호출이 실행되면 인자를 받아 함수에 전달함
generated code에서 Swift가 해당 함수의 인자로 Existential Container를 전달하는 것을 볼 수 있음

```
// Protocol Types
// The Existentiator in action
func drawACopy(local : Drawable) {
    local.draw()
}
let val : Drawable = Point()
drawACopy(val)
```

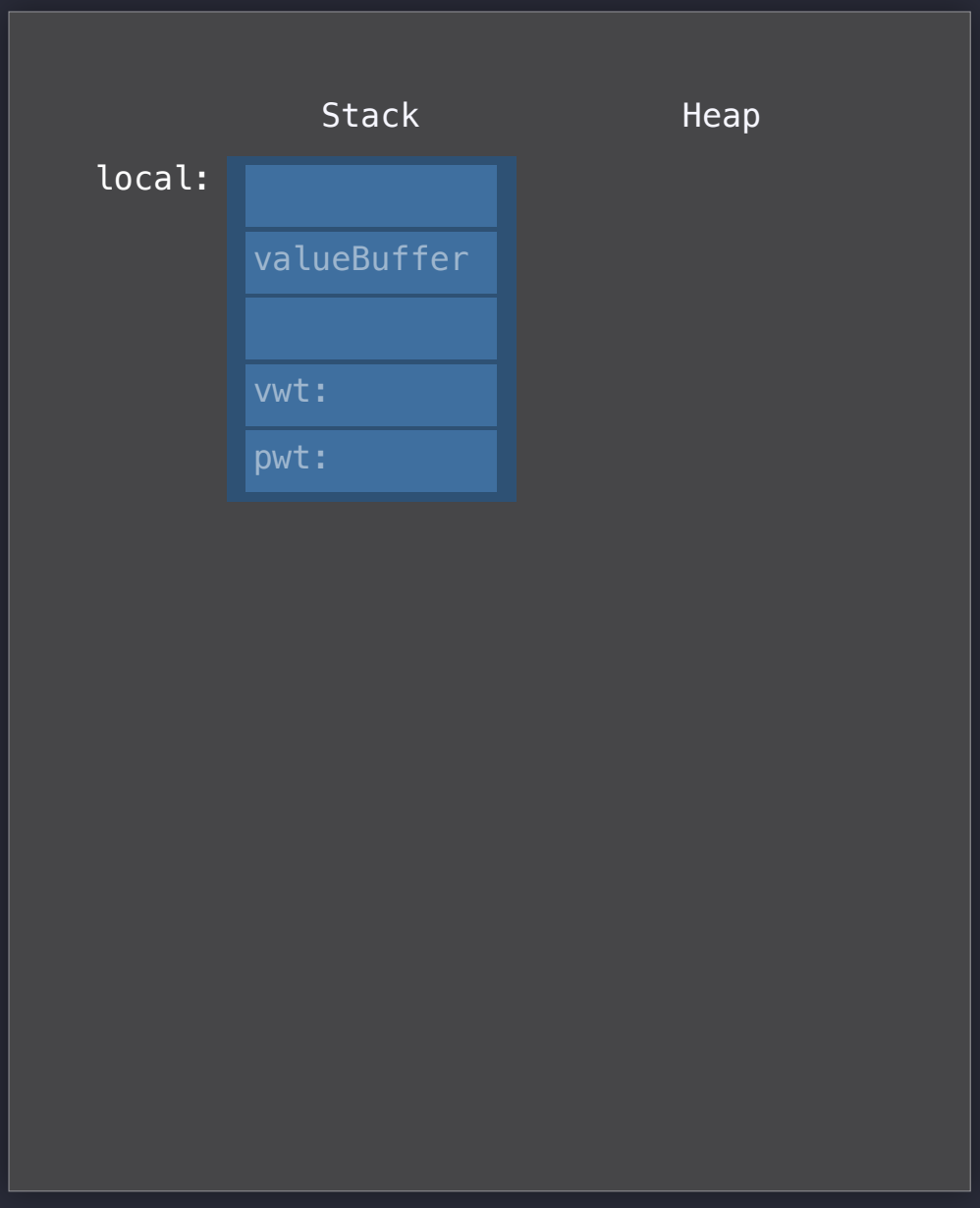
```
let local = val
```

```
// Generated code
func drawACopy(val: ExistContDrawable) {
```

함수 실행이 시작되면, 지역변수를 만들고 파라미터를 할당함

```swift
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
    local.draw()
}
let val : Drawable = Point()
drawACopy(val)
```

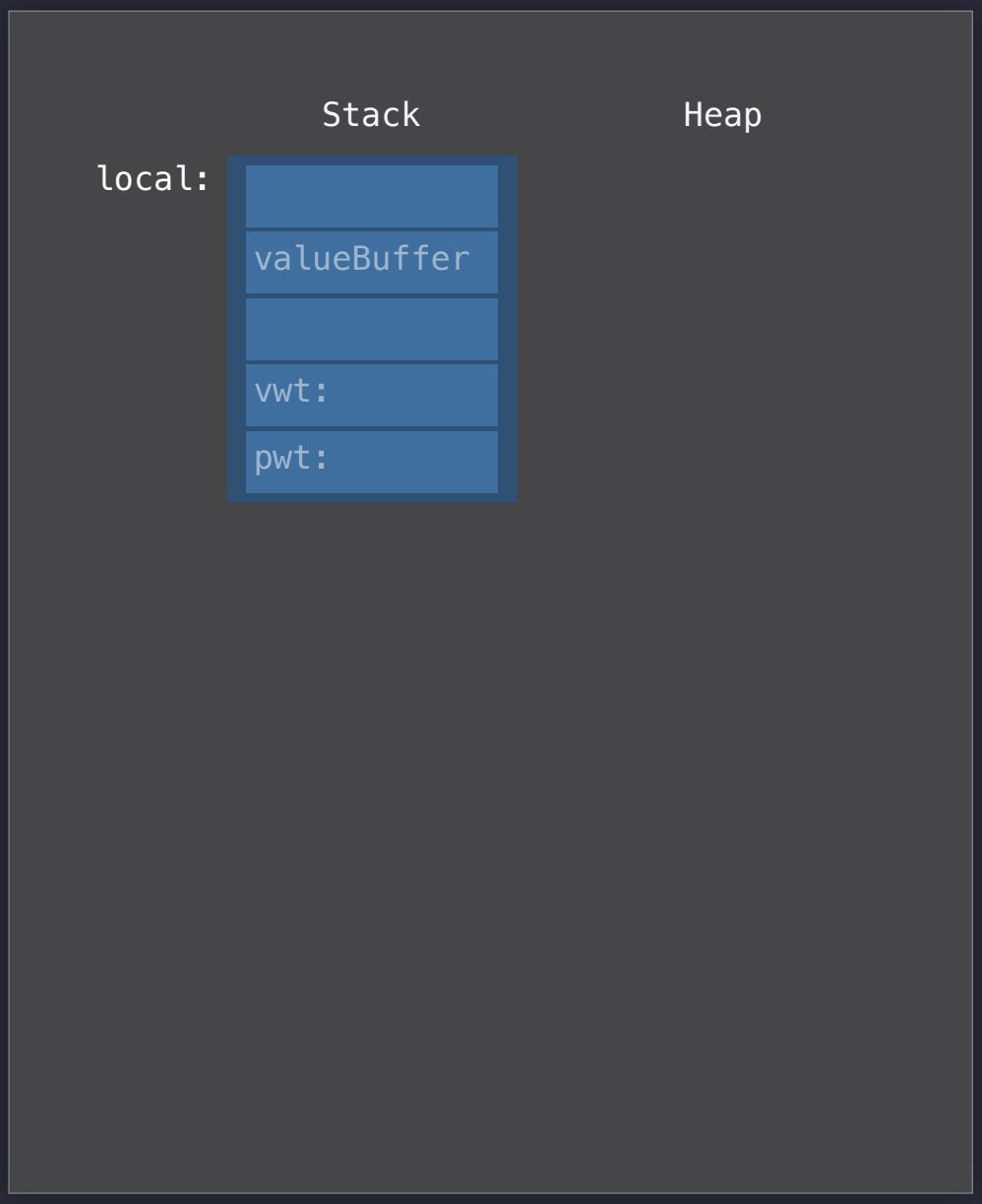
```swift
// Generated code
func drawACopy(val: ExistContDrawable) {
    var local = ExistContDrawable()
```



generated code에서 Swift는 스택에 Existential Container를 할당함

```swift
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
    local.draw()
}
let val : Drawable = Point()
drawACopy(val)
```

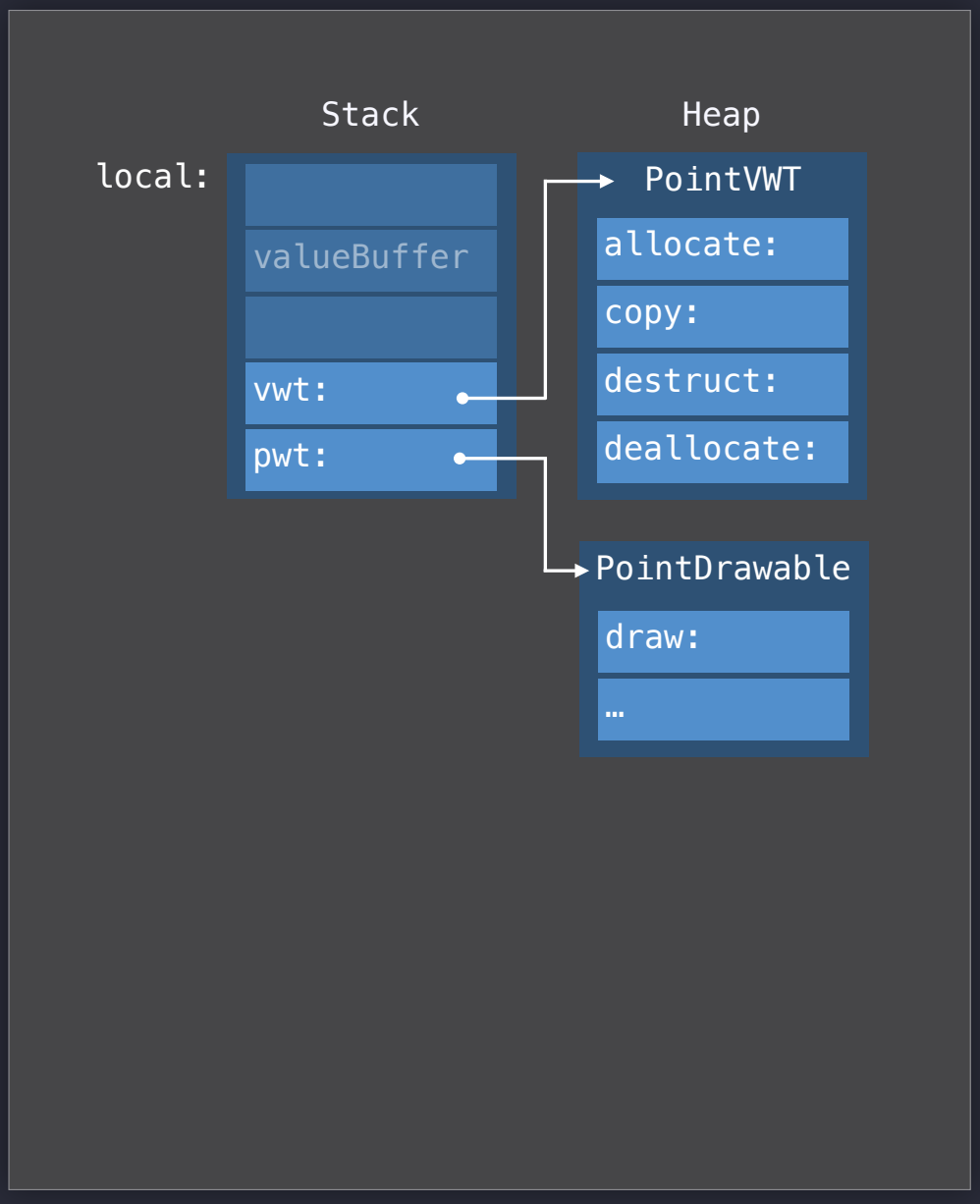
```swift
// Generated code
func drawACopy(val: ExistContDrawable) {
    var local = ExistContDrawable()
    let vwt = val.vwt
    let pwt = val.pwt
    local.type = type
    local.pwt = pwt
```



다음 인자로 들어온 Existential Container에서 value witness table과 protocol witness table을 읽고 local existential container 영역을 초기화

```swift
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
    local.draw()
}
let val : Drawable = Point()
drawACopy(val)
```

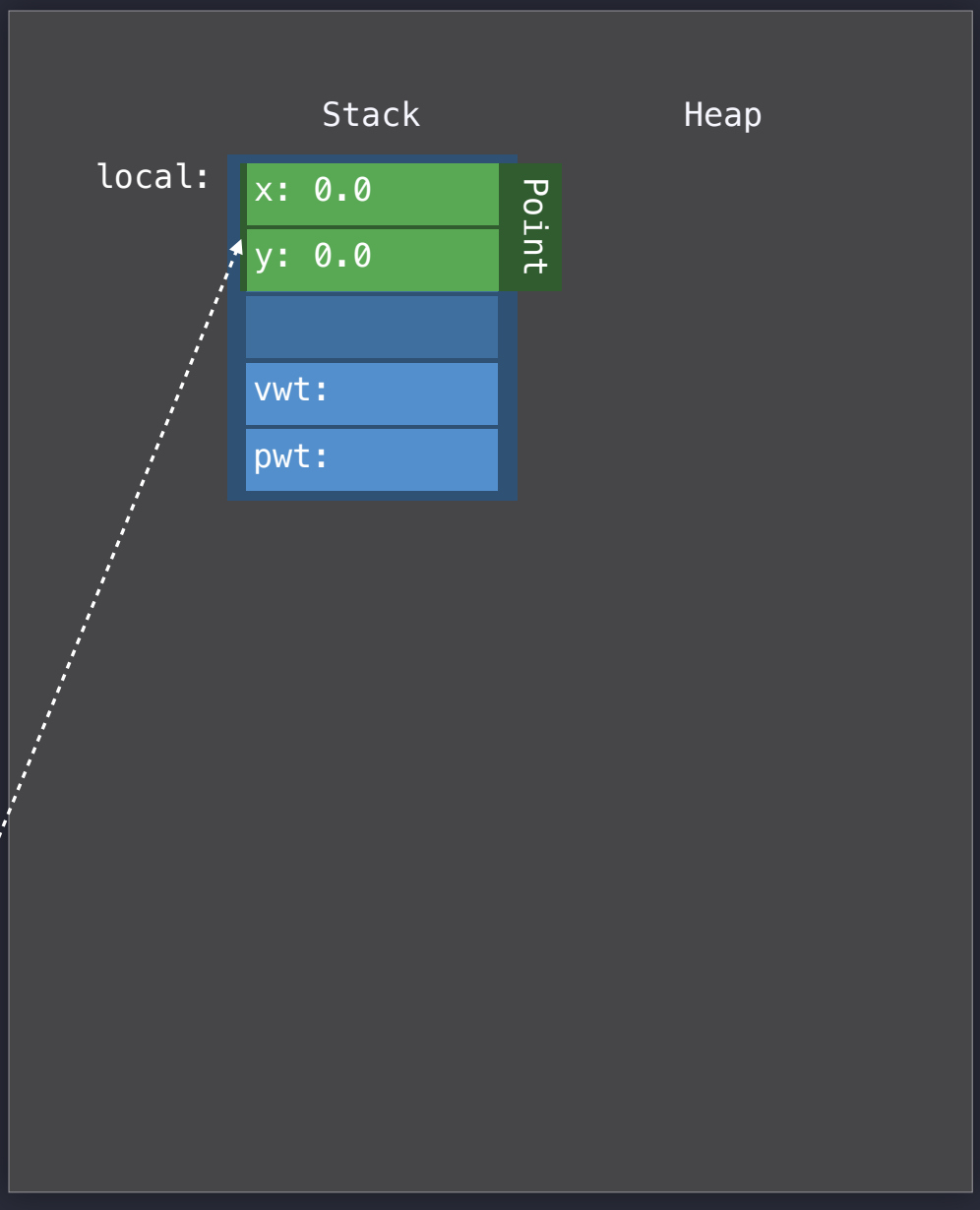
```swift
// Generated code
func drawACopy(val: ExistContDrawable) {
    var local = ExistContDrawable()
    let vwt = val.vwt
    let pwt = val.pwt
    local.type = type
    local.pwt = pwt
```



그 다음 필요하면 value witness 함수를 호출하고 buffer에 할당함
그리고 값을 복사
이 예제에서는 동적 힙 할당이 필요 없었음

```swift
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
    local.draw()
}
let val : Drawable = Point()
drawACopy(val)
```

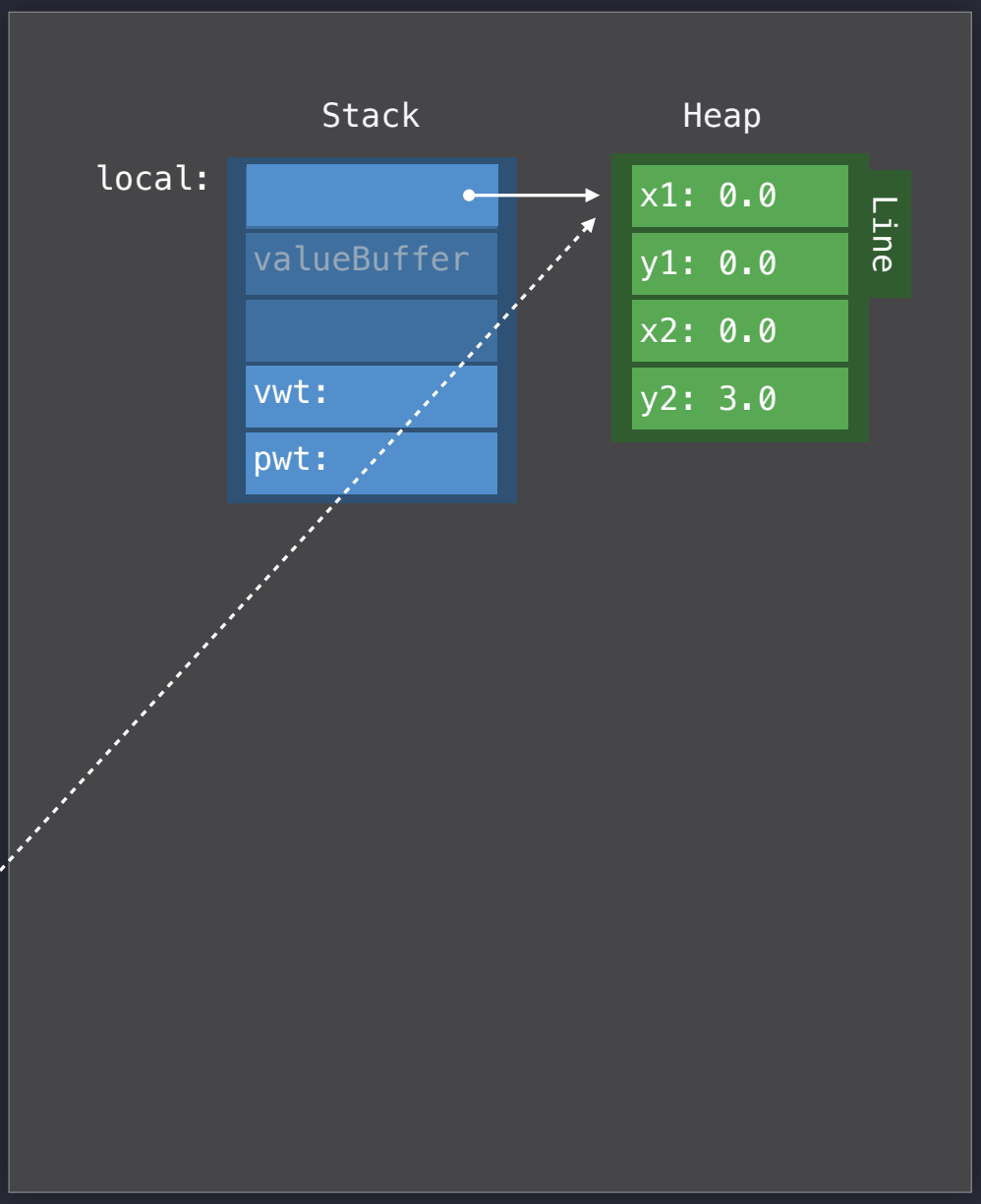
```swift
// Generated code
func drawACopy(val: ExistContDrawable) {
    var local = ExistContDrawable()
    let vwt = val.vwt
    let pwt = val.pwt
    local.type = type
    local.pwt = pwt
    vwt.allocateBufferAndCopyValue(&local, val)
```



이 함수는 인자의 값을 local existential container의 valueBuffer로 복사함

```swift
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
    local.draw()
}
let val : Drawable = Line()
drawACopy(val)
```

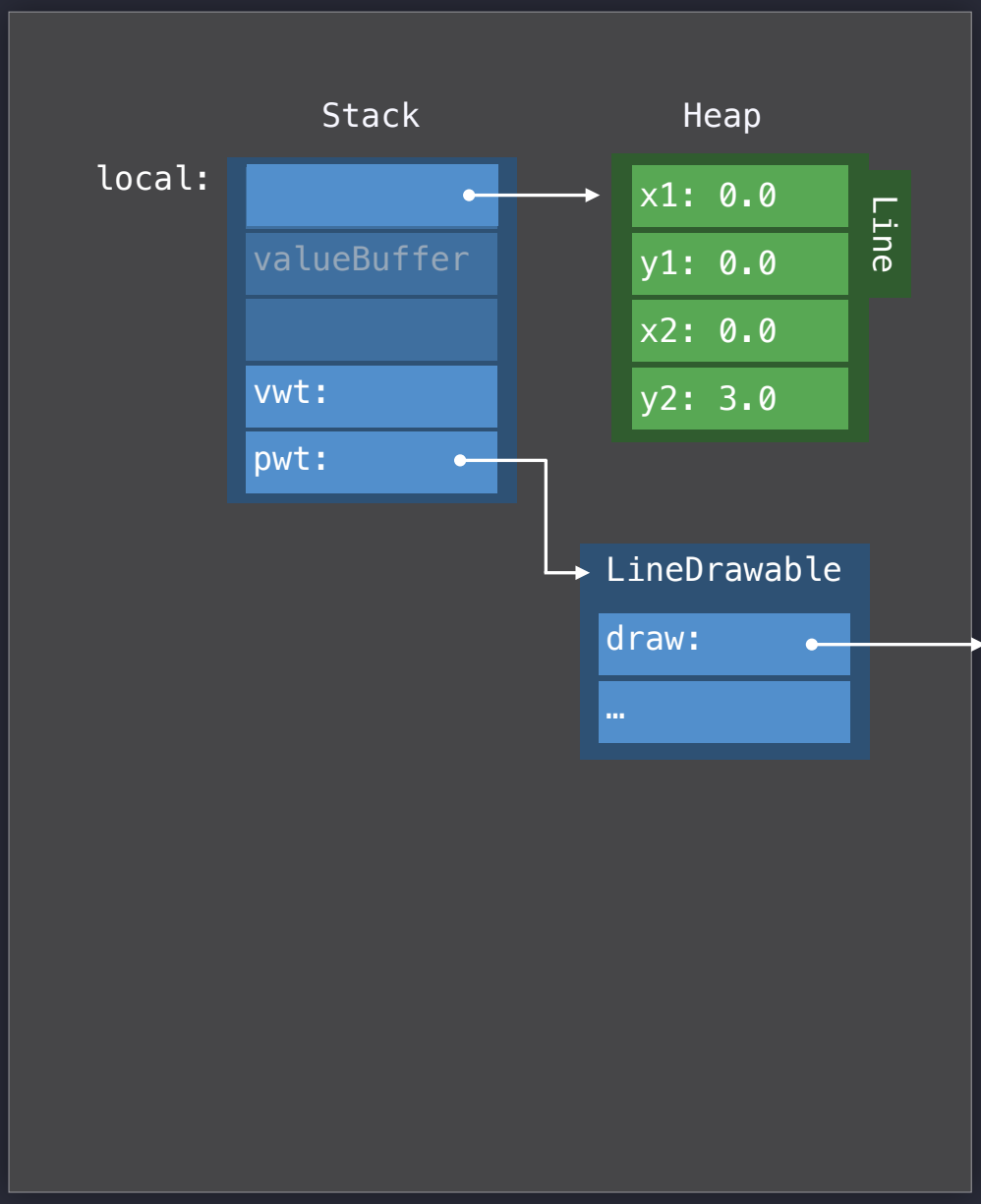
```swift
// Generated code
func drawACopy(val: ExistContDrawable) {
    var local = ExistContDrawable()
    let vwt = val.vwt
    let pwt = val.pwt
    local.type = type
    local.pwt = pwt
    vwt.allocateBufferAndCopyValue(&local, val)
```



만약 Line을 전달받으면, 이 함수는 버퍼를 할당하고 값을 복사할 것

```swift
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
    local.draw()
}
let val : Drawable = Line()
drawACopy(val)
```

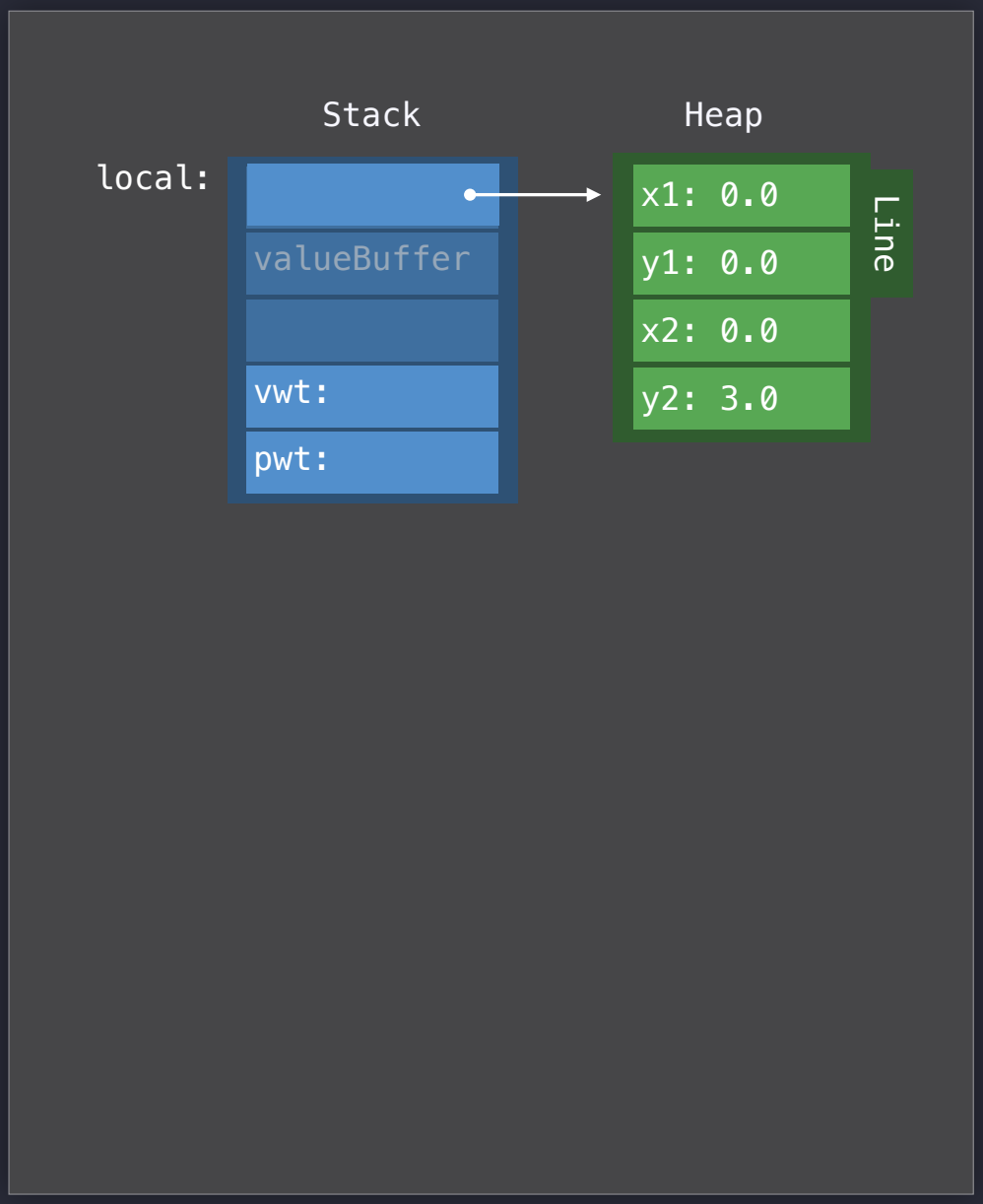
```swift
// Generated code
func drawACopy(val: ExistContDrawable) {
    var local = ExistContDrawable()
    let vwt = val.vwt
    let pwt = val.pwt
    local.type = type
    local.pwt = pwt
    vwt.allocateBufferAndCopyValue(&local, val)
    pwt.draw(vwt.projectBuffer(&local))
```



다음으로 draw 메서드가 실행되고 Swift는 existential container 내부에 protocol witness table을 찾고, 고정된 offset에서 draw 메서드를 찾고 구현부로 이동

```swift
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
    local.draw()
}
let val : Drawable = Line()
drawACopy(val)
```

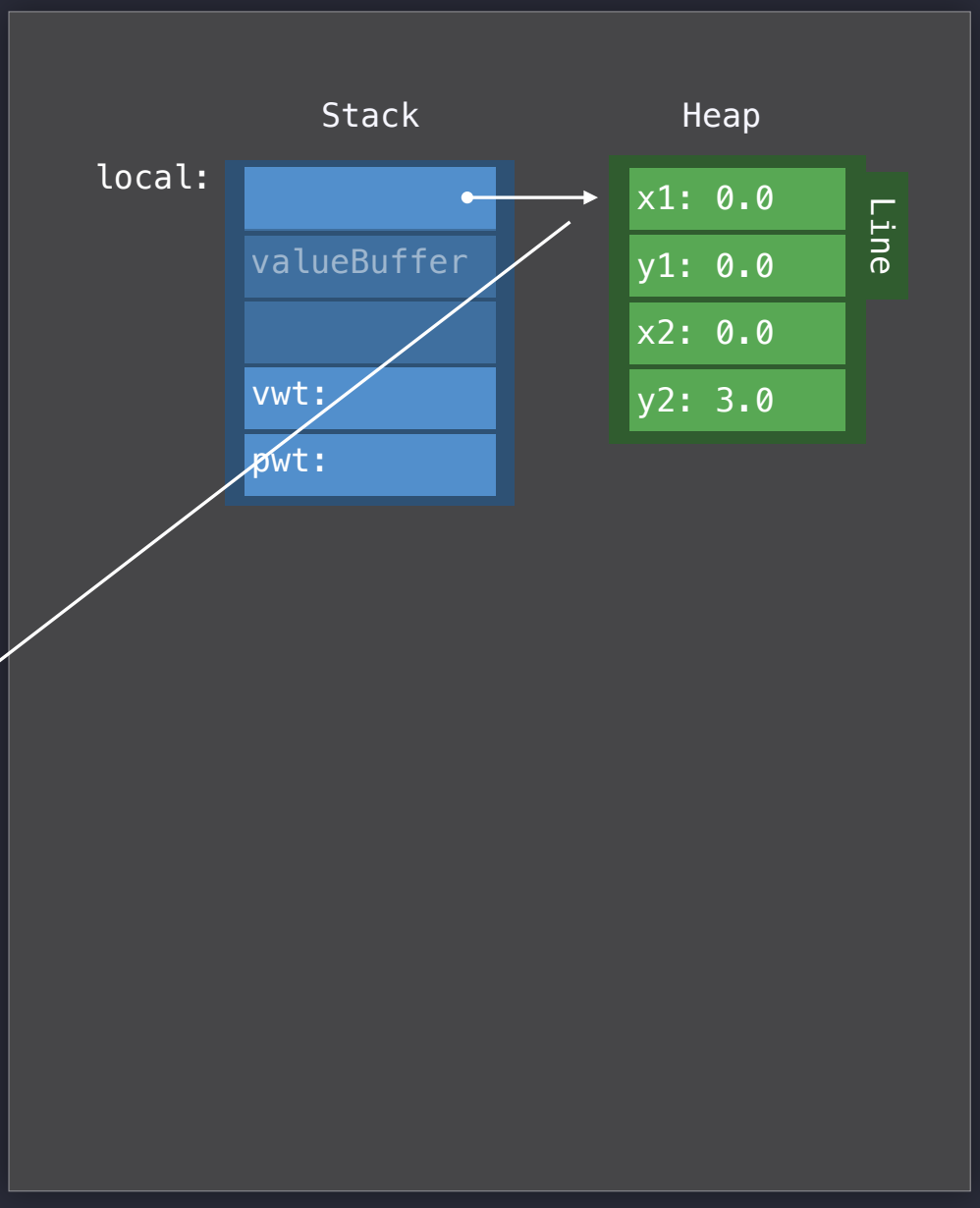
```swift
// Generated code
func drawACopy(val: ExistContDrawable) {
    var local = ExistContDrawable()
    let vwt = val.vwt
    let pwt = val.pwt
    local.type = type
    local.pwt = pwt
    vwt.allocateBufferAndCopyValue(&local, val)
    pwt.draw(vwt.projectBuffer(&local))
```



여기에 또다른 value witness가 있음 -> projectBuffer
왜 여기 있나?
draw 메서드는 입력으로 값 주소를 기대함

```swift
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
    local.draw()
}
let val : Drawable = Line()
drawACopy(val)
```

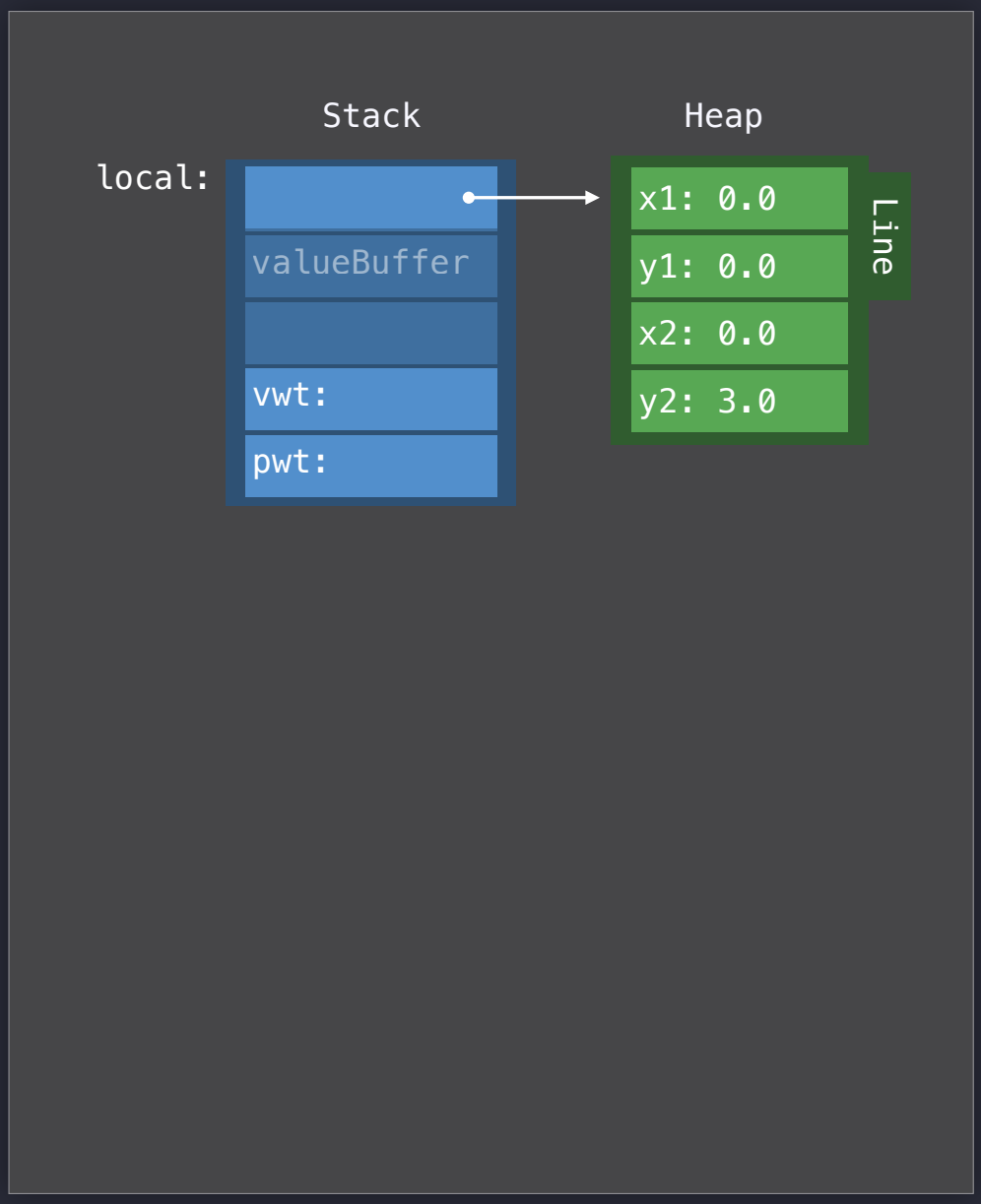
```swift
// Generated code
func drawACopy(val: ExistContDrawable) {
    var local = ExistContDrawable()
    let vwt = val.vwt
    let pwt = val.pwt
    local.type = type
    local.pwt = pwt
    vwt.allocateBufferAndCopyValue(&local, val)
    pwt.draw(vwt.projectBuffer(&local))
```



이 주소 값은 인라인 버퍼에 맞는 작은 값이라면 exisetential container에서 시작할 것이고,
인라인 버퍼보다 큰 값이라면 힙에 할당된 메모리의 주소에서 시작할 것임

```swift
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
    local.draw()
}
let val : Drawable = Line()
drawACopy(val)
```

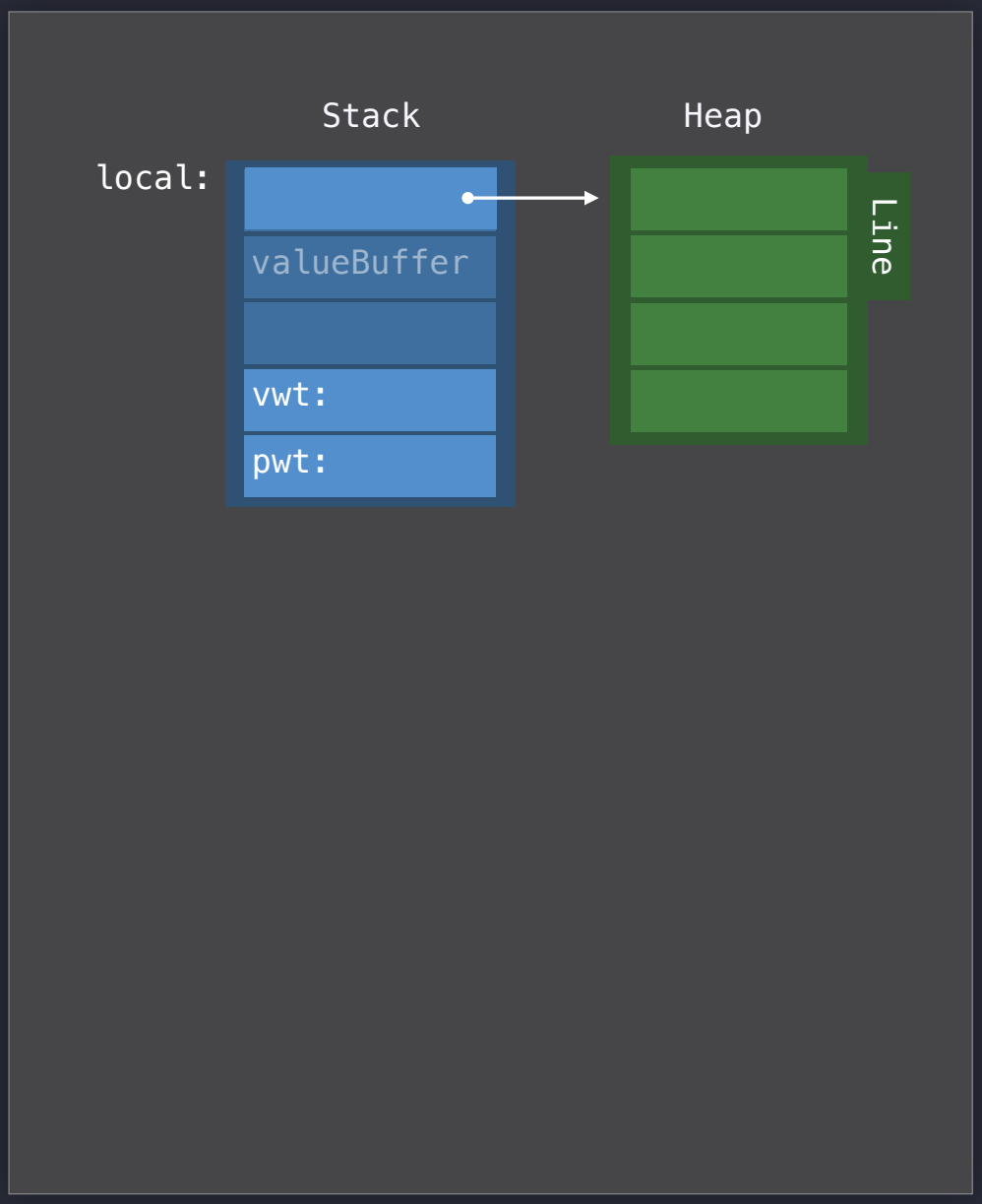
```swift
// Generated code
func drawACopy(val: ExistContDrawable) {
    var local = ExistContDrawable()
    let vwt = val.vwt
    let pwt = val.pwt
    local.type = type
    local.pwt = pwt
    vwt.allocateBufferAndCopyValue(&local, val)
    pwt.draw(vwt.projectBuffer(&local))
    vwt.destructAndDeallocateBuffer(temp)
}
```



draw 메서드가 실행되고 완료되면, 함수의 끝에 도달하게 됨

```swift
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
    local.draw()
}
let val : Drawable = Line()
drawACopy(val)
```

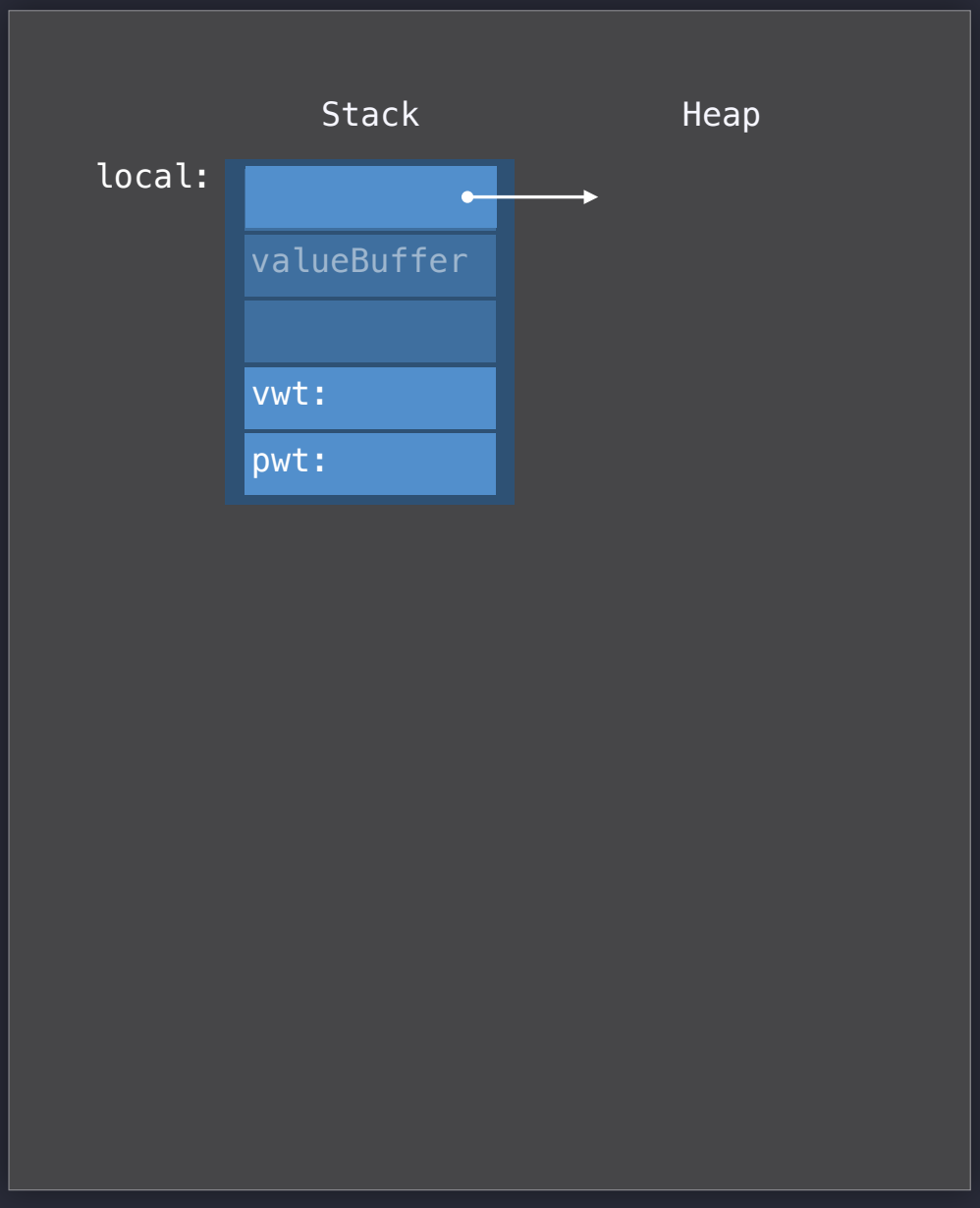
```swift
// Generated code
func drawACopy(val: ExistContDrawable) {
    var local = ExistContDrawable()
    let vwt = val.vwt
    let pwt = val.pwt
    local.type = type
    local.pwt = pwt
    vwt.allocateBufferAndCopyValue(&local, val)
    pwt.draw(vwt.projectBuffer(&local))
    vwt.destructAndDeallocateBuffer(temp)
}
```



매개변수에 대해 생성된 지역변수가 매개변수의 범위를 벗어나게 됨
이제 Swift는 값을 파괴하기 위해 value witness 함수를 호출함
만약 값에 참조가 있다면 참조 카운팅이 감소할 것이고 버퍼가 할당되있었다면, 버퍼가 할당 해제될 것임

```swift
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
    local.draw()
}
let val : Drawable = Line()
drawACopy(val)
```

```swift
// Generated code
func drawACopy(val: ExistContDrawable) {
    var local = ExistContDrawable()
    let vwt = val.vwt
    let pwt = val.pwt
    local.type = type
    local.pwt = pwt
    vwt.allocateBufferAndCopyValue(&local, val)
    pwt.draw(vwt.projectBuffer(&local))
    vwt.destructAndDeallocateBuffer(temp)
}
```



함수 실행이 완료되고 스택이 제거되면서 스택에 생성되었던 local existential container가 제거됨
이 작업에서 주목하고자 하는 한 가지는 Line, Point와 같은 값 타입을 프로토콜과 결합하여 동적 동작, 동적 다형성을 얻을 수 있다는 것
Drawable 프로토콜 배열에 Line과 Point를 저장할 수 있었음

역동성이 필요하다면 이 방법은 이전에 본 예제처럼 클래스를 사용하여 구현하는 것보다 비용이 괜찮은 좋은 선택
왜냐하면 클래스는 V-Table을 통해 참조 카운팅 추가 오버헤드가 있기 때문

여기까지 우리는 지역 변수가 복사되는 방식과 프로토콜 타입 값에 대해 메서드 디스패치가 작동하는 방식을 살펴봄

Protocol Type Stored Properties

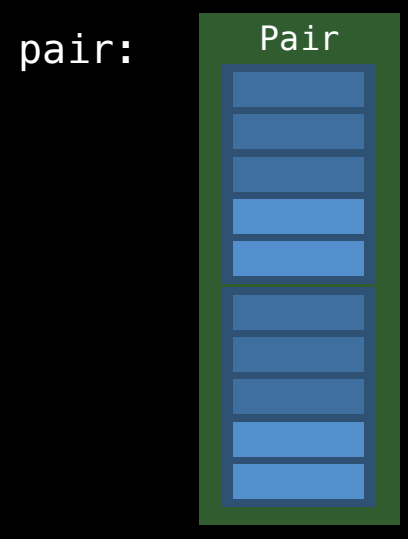
```
struct Pair {
   init(_ f: Drawable, _ s: Drawable) {
      first = f ; second = s
   }
   var first: Drawable
   var second: Drawable
}
```

이번에는 저장 프로퍼티에 대해 볼 것임
Drawable 프로토콜 타입의 first와 second 프로퍼티를 가진 Pair라는 구조체가 있음
Swift는 두 저장 프로퍼티를 어떻게 저장하나

Protocol Type Stored Properties

```
struct Pair {
   init(_ f: Drawable, _ s: Drawable) {
      first = f ; second = s
   }
   var first: Drawable
   var second: Drawable
}

var pair = Pair(Line(), Point())
```

Existential Container inline



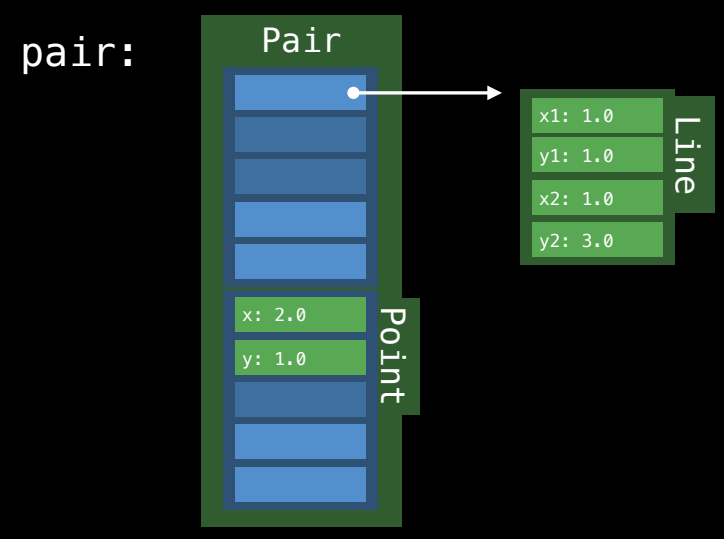
Pair를 할달하면, Swift는 인라인에 해당 Pair를 저장하는데 필요한 두개의 existential container를 저장할 것

Protocol Type Stored Properties

```
struct Pair {
   init(_ f: Drawable, _ s: Drawable) {
      first = f ; second = s
   }
   var first: Drawable
   var second: Drawable
}

var pair = Pair(Line(), Point())
```

Existential Container inline

Large values on the heap

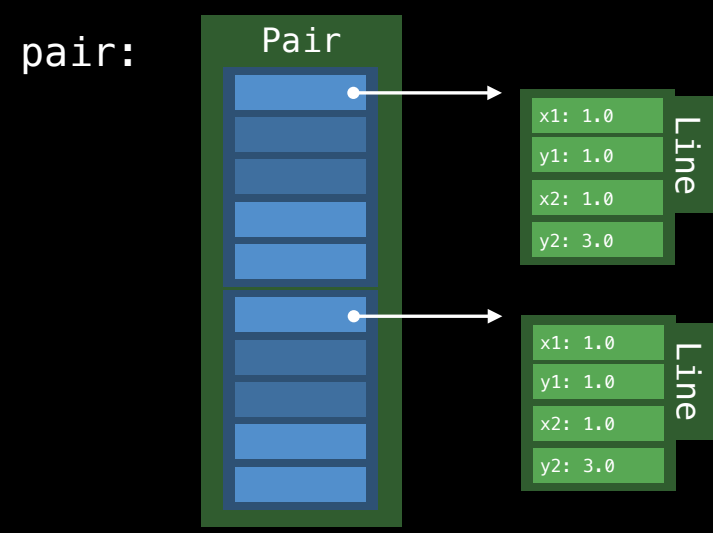


Line은 이전에 본 것 처럼 힙에 buffer를 할당함
Point는 인라인 valueBuffer에 맞으니 existential container에 저장할 수 있음

Protocol Type Stored Properties

```
struct Pair {
   init(_ f: Drawable, _ s: Drawable) {
      first = f ; second = s
   }
   var first: Drawable
   var second: Drawable
}

var pair = Pair(Line(), Point())
pair.second = Line()
```

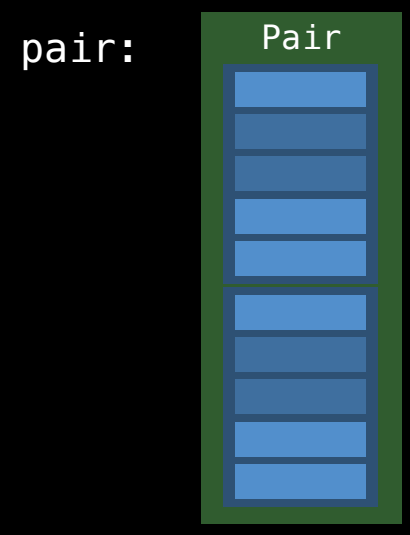
Supports dynamic polymorphism



두번째 파라미터에 다시 Line을 저장하려고 함
이제 두개의 힙 할당을 만들어냄
힙 할당 비용을 설명하기 위해 다른 프로그램을 살펴보겠음

Expensive Copies of Large Values

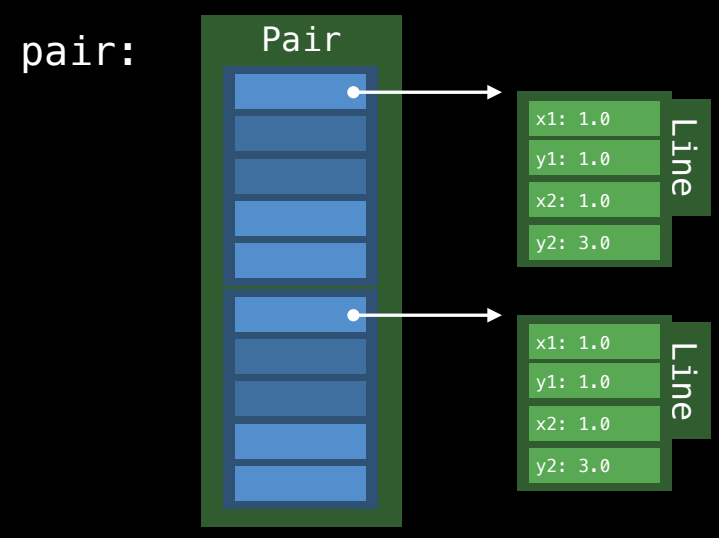
```
let aLine = Line(1.0, 1.0, 1.0, 3.0)
let pair = Pair(aLine, aLine)
let copy = pair
```



이번에는 Line을 만들고 Pair를 만들어서 Pair를 Line으로 초기화함

Expensive Copies of Large Values

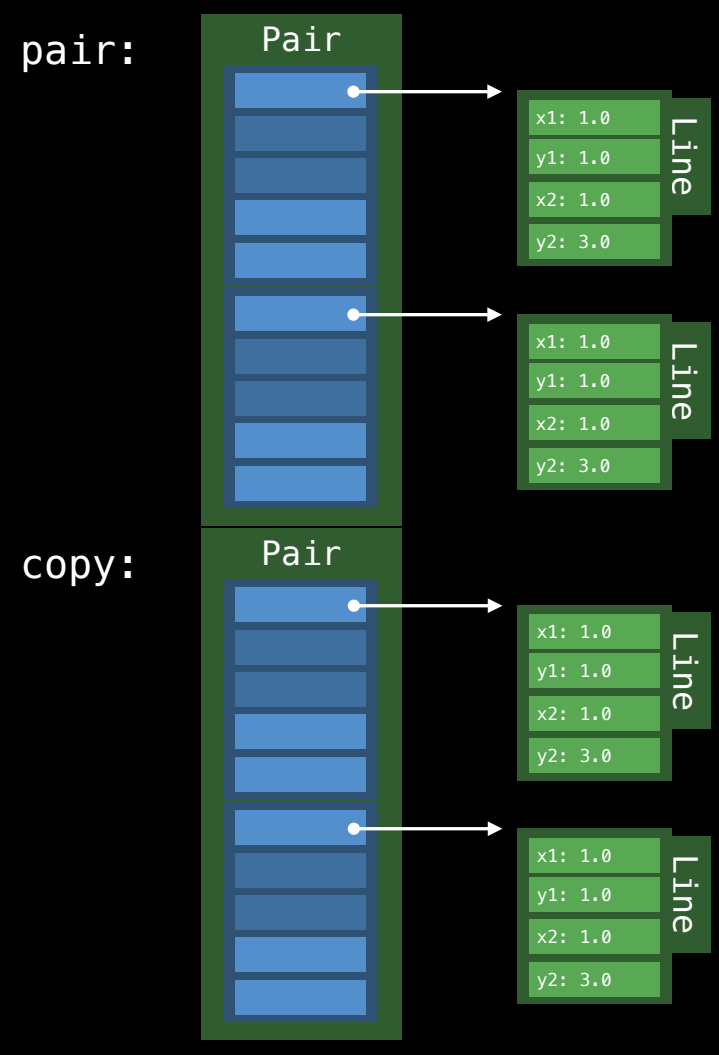
```
let aLine = Line(1.0, 1.0, 1.0, 3.0)
let pair = Pair(aLine, aLine)
let copy = pair
```



이로 인해 두개의 힙 할당이 생겼음

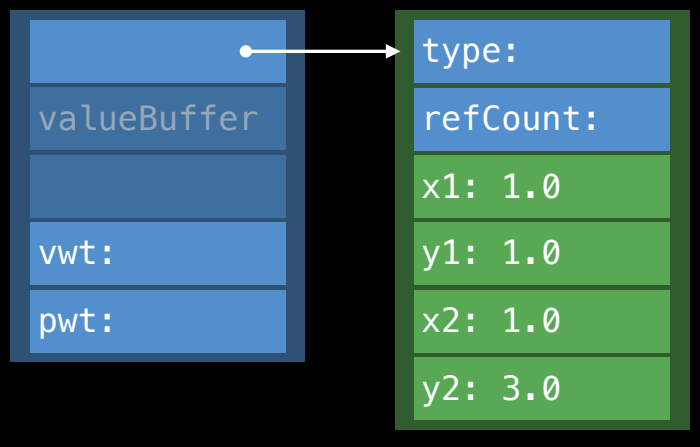
Expensive Copies of Large Values

```
let aLine = Line(1.0, 1.0, 1.0, 3.0)
let pair = Pair(aLine, aLine)
let copy = pair
```



그리고 다시 해당 복사본을 다시 생성함
두개의 existential container가 스택에 있고 두개의 힙 할당이 있음
이전에 힙 할당은 비용이 비싸다고 했음
여기서 무엇을 할 수 있을까?

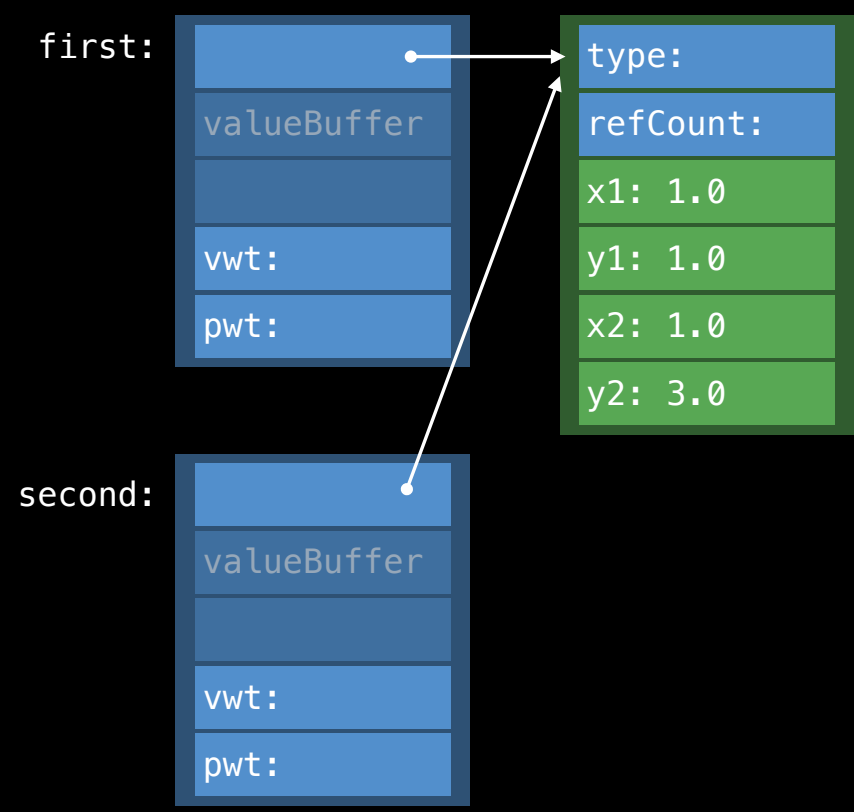
References Fit in the Value Buffer



existential container는 세개의 words를 넣을 수 있음
그리고 참조는 기본적으로 한 word이기에 이 세개의 words는 들어감

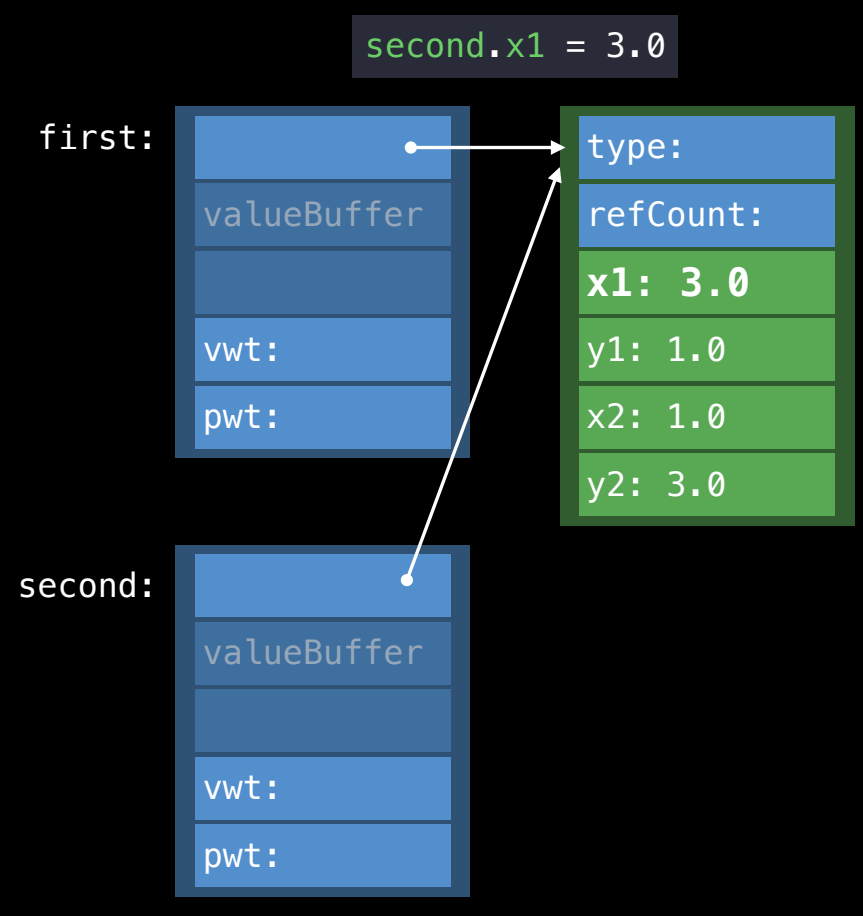
만약 Line을 클래스로 구현했다면 클래스는 reference semantic이므로 참조에 의해 저장되었을 것이고, 이 참조는 valuBuffer에 맞았을 것

References Fit in the Value Buffer



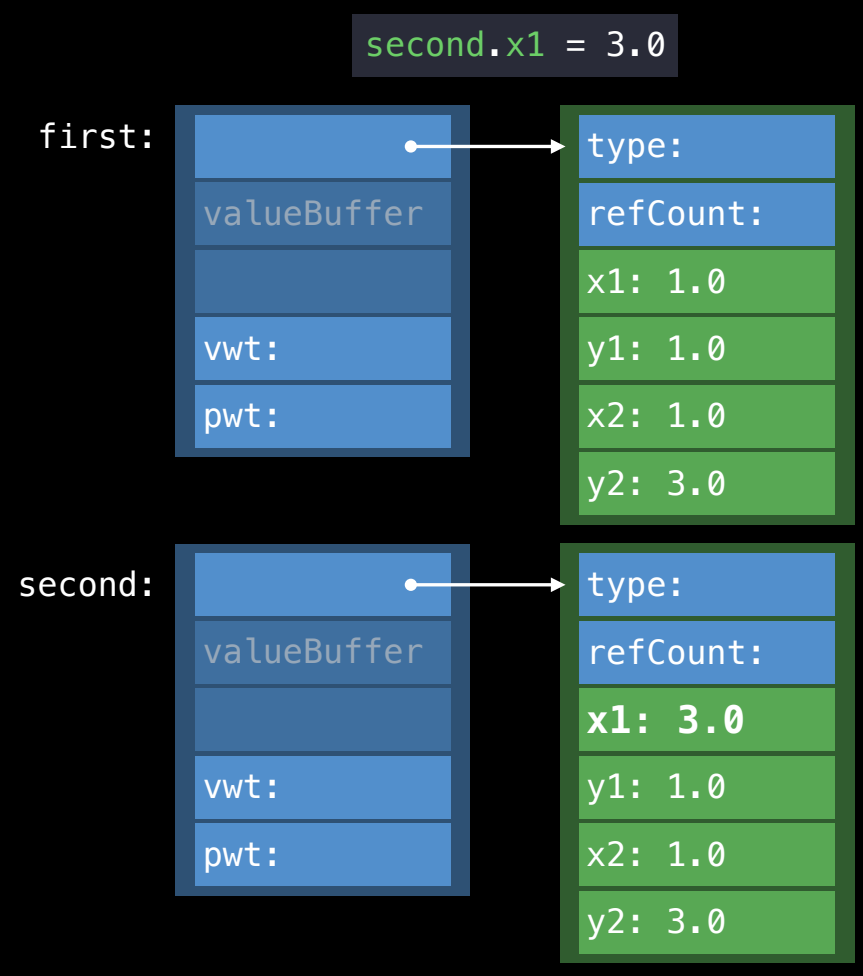
Pair의 첫번째 참조를 두번째 필드로 복사할 때 참조만 복사되고 우리가 지불했을 유일한 것은 참조 카운트 증가일 것

References Fit in the Value Buffer



하지만, reference semantic은 의도치 않은 공유 상태를 일으킴
Pair의 두번째 필드에서 x1에 특정 값을 저장한다면 첫번째 필드도 관찰 후 변경할 것임
이것은 우리가 원하지 않는 것
우리가 원하는 것은 value semantic
여기서 CoW 기술을 사용하면 됨

References Fit in the Value Buffer



이전에 클래스로 사용되었을 때, 우리는 참조 카운트를 확인하였음
동일한 인스턴스에 대해 하나 이상의 참조가 있으면 해당 참조 카운트는 1보다 클 것
이러한 상황일 경우 인스턴스를 쓰기전에 인스턴스를 복사하고 해당 복사본에 씀
이로써 상태가 분리됨

Indirect Storage with Copy-on-Write

Use a reference type for storage

```
class LineStorage { var x1, y1,  x2, y2: Double }
struct Line : Drawable {
   var storage : LineStorage
   init() { storage = LineStorage(Point(), Point()) }
   func draw() { … }
   mutating func move() {
      if !isUniquelyReferencedNonObjc(&storage) {
         storage = LineStorage(storage)
      }
      storage.start = ...
   }
}
```

Line 내부에 Storage를 구현하는 대신 Line 구조체의 모든 필드를 포함하는 LineStorage라는 클래스를 만듬
그런 다음 Line 구조체가 이 Storage를 참조
그리고 값을 읽고 싶을 때 storage 내부의 값을 읽음

Indirect Storage with Copy-on-Write

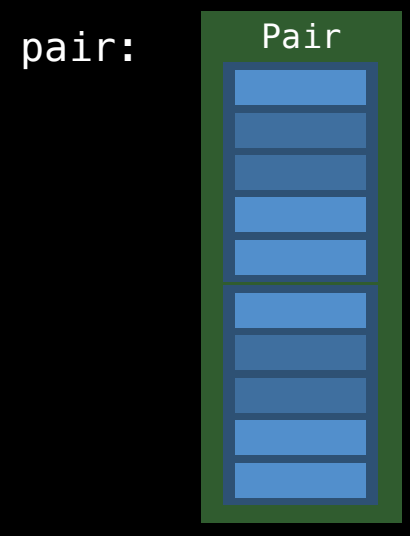
Implement copy-on-write

```
class LineStorage { var x1, y1,  x2, y2: Double }
struct Line : Drawable {
   var storage : LineStorage
   init() { storage = LineStorage(Point(), Point()) }
   func draw() { … }
   mutating func move() {
      if !isUniquelyReferencedNonObjc(&storage) {
         storage = LineStorage(storage)
      }
      storage.start = ...
   }
}
```

하지만, 값을 수정하거나 변경하려면, 참조 카운트를 확인함
1보다 크면 isUniquelyRefereced가 호출됨
이것은 참조 카운팅을 확인하는 녀석임
1보다 동일하거나 같으면 LineStorage의 복사본을 만들고 변경을 함

Copy Using Indirect Storage

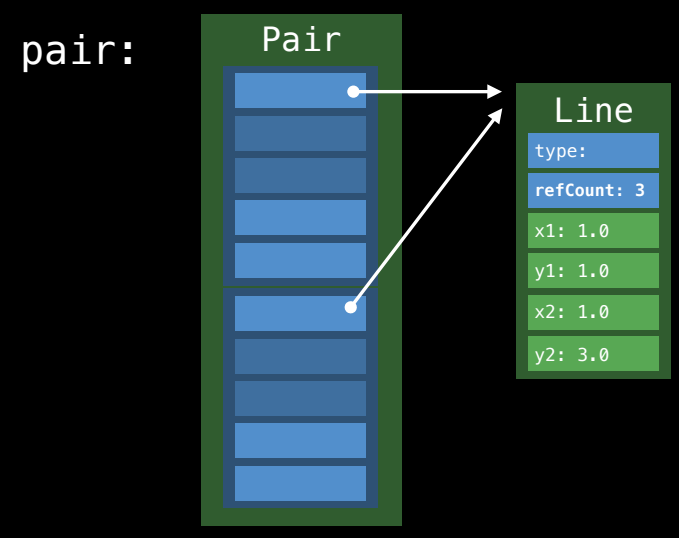
```
let aLine = Line(1.0, 1.0, 1.0, 1.0)
let pair = Pair(aLine, aLine)
let copy = pair
```



구조체와 클래스를 결합하여 CoW를 사용하는 간접 storage를 사용하는 방법에 대해 알아봄
이번에는 간접 저장을 사용할 때 어떤 일이 발생하는지 볼 것임
Line을 생성하고 이것은 lineStorage 객체를 힙에 생성할 것임
그런 다음 Line을 사용하여 Pair를 초기화함
이번에는 lineStorage에 대한 참조만 복사될 것임

Copy Using Indirect Storage

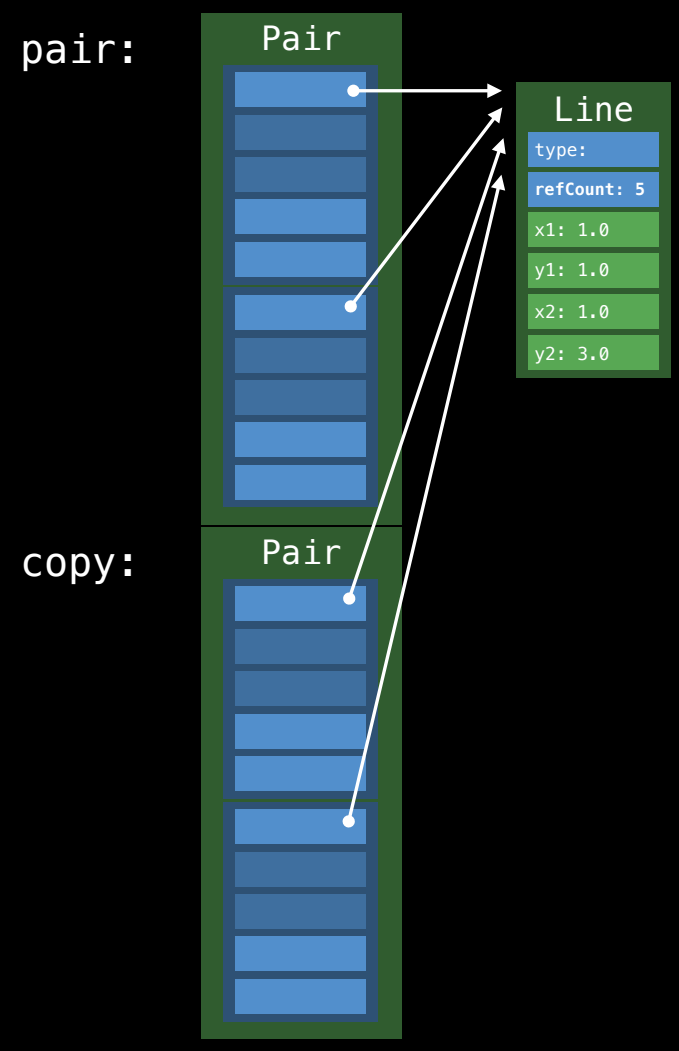
```
let aLine = Line(1.0, 1.0, 1.0, 1.0)
let pair = Pair(aLine, aLine)
let copy = pair
```



구조체와 클래스를 결합하여 CoW를 사용하는 간접 storage를 사용하는 방법에 대해 알아봄
이번에는 간접 저장을 사용할 때 어떤 일이 발생하는지 볼 것임
Line을 생성하고 이것은 lineStorage 객체를 힙에 생성할 것임
그런 다음 Line을 사용하여 Pair를 초기화함
이번에는 lineStorage에 대한 참조만 복사될 것임

Copy Using Indirect Storage

```
let aLine = Line(1.0, 1.0, 1.0, 1.0)
let pair = Pair(aLine, aLine)
let copy = pair
```



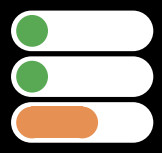
여기서 Line을 다시 복사하면 참조만 복사되고 참조 카운트가 증가할 것
이것은 힙 할당보다 훨씬 저렴함

Performance of Protocol Types

```
func drawACopy(val: ExistContDrawable) {
   var local = ExistContDrawable()
   let vwt = val.vwt
   let pwt = val.pwt
   local.type = type
   local.pwt = pwt
   vwt.allocateBufferAndCopyValue(&local, val)
   pwt.draw(vwt.projectBuffer(&local))
   vwt.destructAndDeallocateBuffer(temp)
}
```

우리는 프로토콜 타입의 변수가 복사 및 저장되는 방식과 메서드 디스패치가 작동하는 방식을 살펴봄
성능에 어떤 의미가 있는지 살펴보자

Protocol Type—Small Value



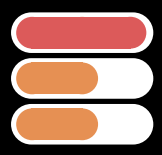
Fits in Value Buffer: no heap allocation

No reference counting

Dynamic dispatch through Protocol Witness Table

Existential container의 인라인 vlaueBuffer에 들어갈 수 있는 작은 값을 포함하는 프로토콜 타입은 힙 할당이 없음
구조체에 참조가 포함되어 있지 않고, 참조 카운팅도 없음
이것은 빠른 코드임
value witness 및 protocol witness table을 통한 간접 방식이기에 동적 디스패치의 모든 기능을 얻을 수 있어서 다형성 동작이 가능

Protocol Type—Large Value



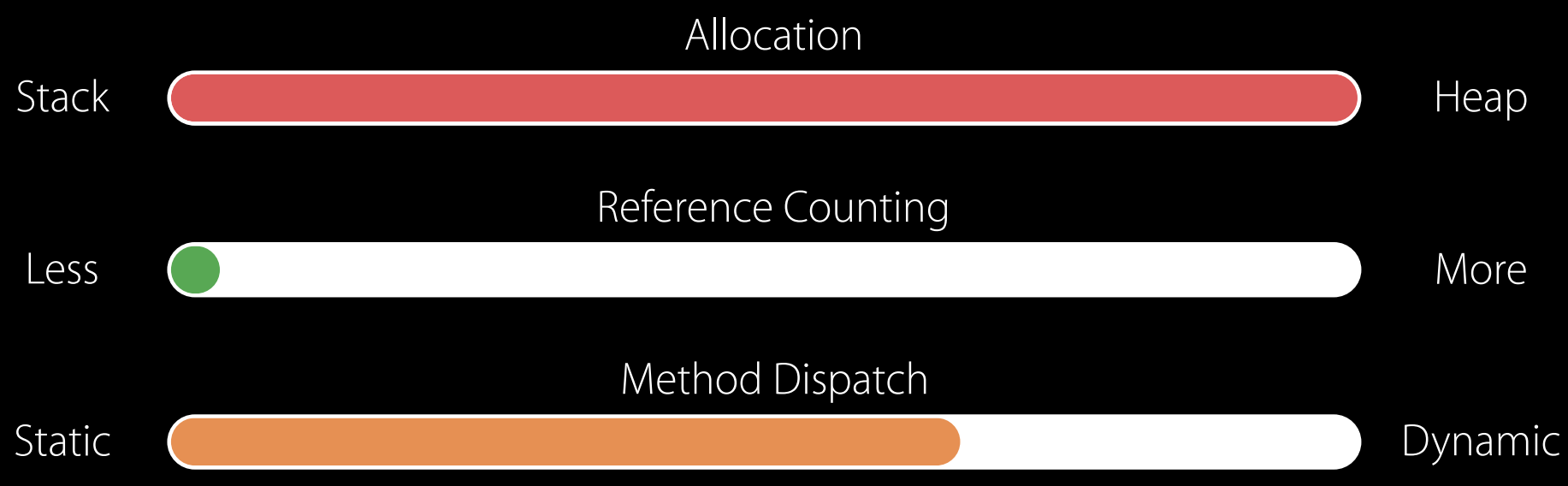
Heap allocation

Reference counting if value contains references

프로토콜 타입의 변수를 초기화하거나 할당할 때마다 힙 할당이 발생
큰 값 구조체에 참조가 포함된 경우 잠재적인 참조 카운팅도 발생함

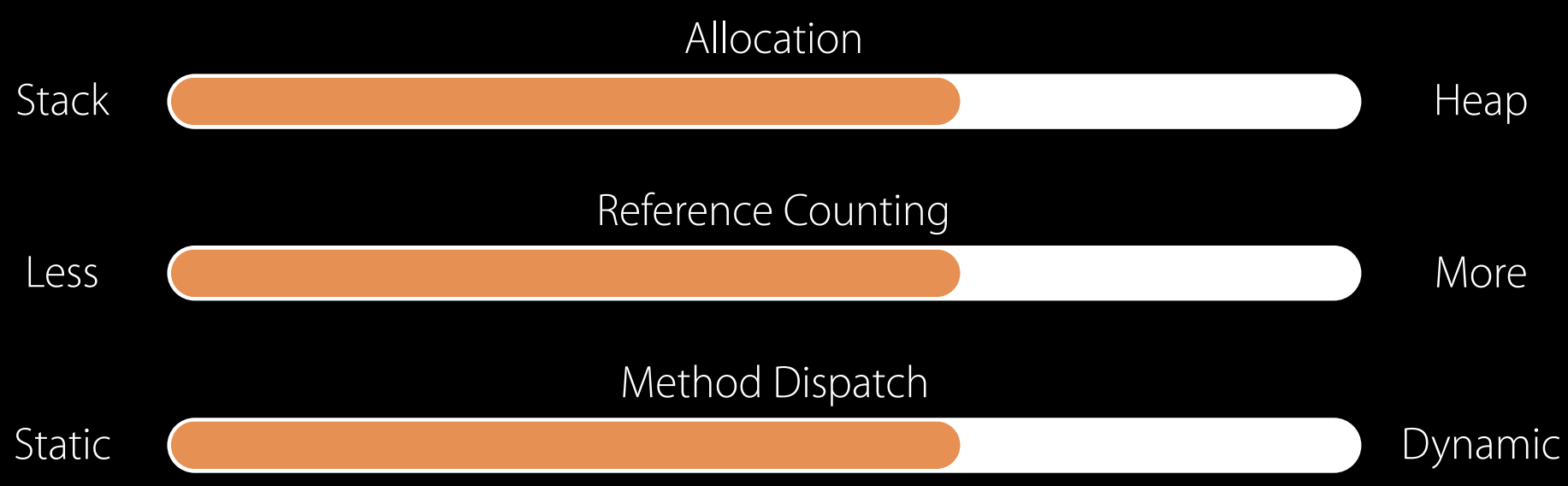
Protocol Type—Large Value

Expensive heap allocation on copying



Protocol Type—Indirect Storage

Trade off reference counting for heap allocation



CoW와 같은 함계 간접 storage를 사용하여 값비싼 힙 할당을 대신할 수 있는 기술을 보여줌
이것은 클래스를 사용하는 것과 유리하게 비교됨

클래스에는 참조 카운팅도 발생함
그리고 초기화 될때 할당 됨

Summary—Protocol Types

Dynamic polymorphism

Indirection through Witness Tables and Existential Container

Copying of large values causes heap allocation

동적 형태의 다형성을 제공함
프로토콜을 값 타입과 함께 사용할 수 있으며, 같은 프로토콜 타입의 형태 배열에 저장할 수 있었음
-> existential container 내부에 있는 protocol, value witness table을 통해 가능함
큰 값을 할당하면 힙 할당이 발생
-> 간접 storage 및 CoW를 사용해 구조체를 구현해 이 문제를 해결할 수 있음

데릭

Generic Code

- 이 강연의 마지막 부분에서 제네릭 타입의 변수가 저장되고 복사되는 방식과 메서드 디스패치가 이러한 변수와 함께 작동하는 방식을 살펴볼 것이다.

```swift
// Drawing a copy using a generic method
protocol Drawable {
   func draw()
}
func drawACopy<T: Drawable>(local : T) {
  local.draw()
}

let line = Line()
drawACopy(line)
// ...
let point = Point()
drawACopy(point)
```

- 따라서 이번에는 generic 코드를 사용하여 구현된 애플리케이션으로 돌아가보자.
- 이제 drawACopy 메서드는 Drawable이라는 generic 매개변수 제약 조건을 취하고 있으며 나머지 부분은 이전
과 동일하다.
- 이것은 프로토콜 타입과 비교할 때 무엇이 다를까?

Generic Code

```
func foo<T: Drawable>(local : T) {
  bar(local)
}
func bar<T: Drawable>(local: T) { … }

let point = Point()
foo(point)
```

```
foo<T = Point>(point)
   bar<T = Point>(local)
```

Static polymorphism

One type per call context

Type substituted down the call chain

- generic 코드는 매개변수 다형성이라고도 하는, 보다 정적 형태의 다형성을 지원한다.
  - 호출 컨텍스트당 하나의 타입.

  Drawable 제약 조건을 가진 T라는 generic 매개변수를 취하는 함수 foo가 있다. 그리고 이 함
  는 해당 매개변수를 함수 bar에 전달한다.

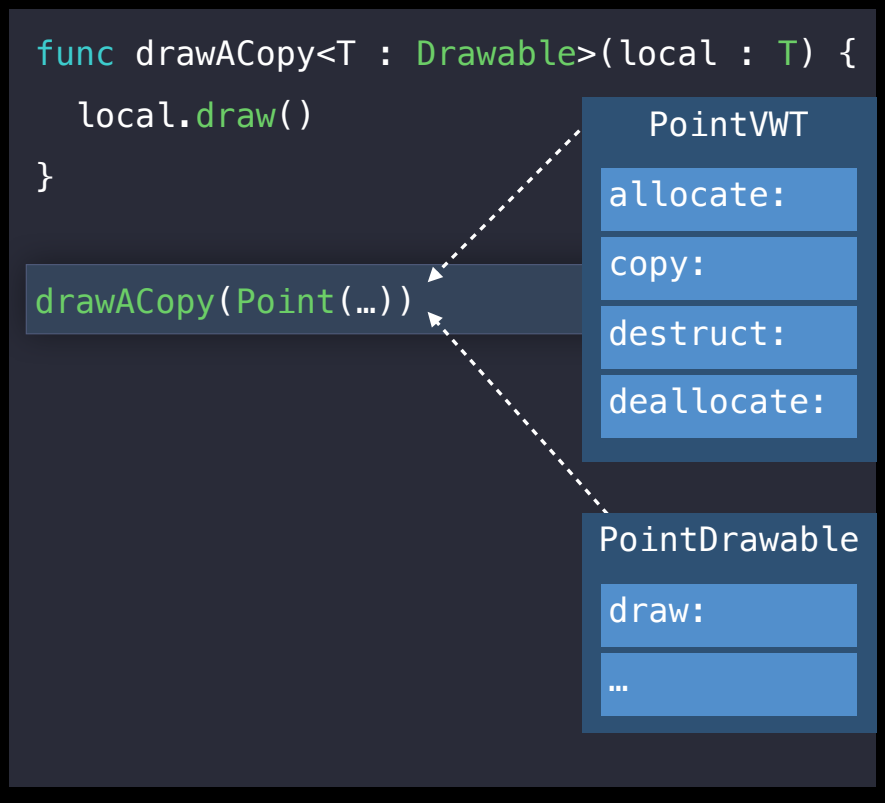
  이 bar 함수는 다시 generic 매개변수 T를 사용한다.

  그런 다음 프로그램은 Point를 만들고 이 point를 foo 함수에 전달한다.

  이 함수가 실행될 때, Swift는 제네릭 타입 T를 이 호출 측에서 사용되는 타입(이 경우에
  는 Point)에 바인딩한다.

  foo 함수가 이 바인딩으로 실행되고 bar의 함수 호출에 도달하면 이 로컬 변수는 방금 찾은
  입, 즉 Point를 갖는다.

Implementation of Generic Methods



One shared implementation

Uses Protocol/Value Witness Table

One type per call context: passes tables

- generic 코드는 매개변수 다형성이라고도 하는, 보다 정적 형태의 다형성을 지원한다.
  - 호출 컨텍스트당 하나의 타입.

  Drawable 제약 조건을 가진 T라는 generic 매개변수를 취하는 함수 foo가 있다. 그리고 이 함
  는 해당 매개변수를 함수 bar에 전달한다.

  이 bar 함수는 다시 generic 매개변수 T를 사용한다.

  그런 다음 프로그램은 Point를 만들고 이 point를 foo 함수에 전달한다.

  이 함수가 실행될 때, Swift는 제네릭 타입 T를 이 호출 측에서 사용되는 타입(이 경우에
  는 Point)에 바인딩한다.

  foo 함수가 이 바인딩으로 실행되고 bar의 함수 호출에 도달하면 이 로컬 변수는 방금 찾은
  입, 즉 Point를 갖는다.

  대신, 이 호출에서 사용된 Point 타입의 value witness table과 protocol witness table을 함수에
  한 추가 인자로 전달할 수 있다.

  따라서 이 경우 Point 와 Line에 대한 value witness table이 전달 되었음을 알 수 있다.

Implementation of Generic Methods

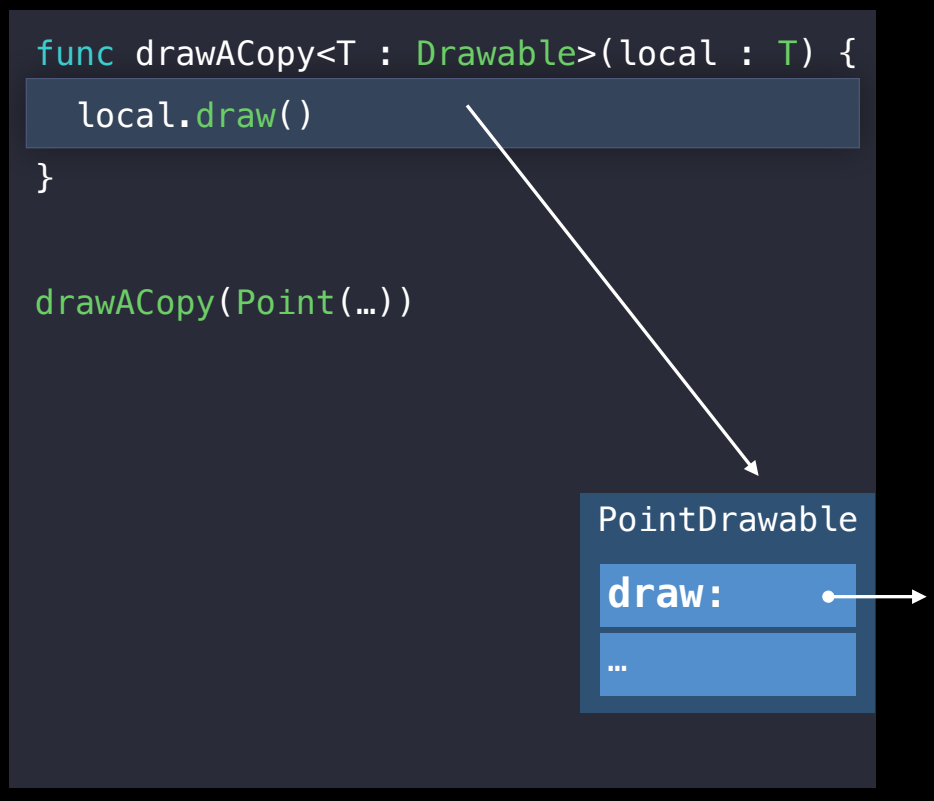
One shared implementation

Uses Protocol/Value Witness Table

One type per call context: passes tables

- 그런 다음 해당 함수를 실행하는 동안, 매개변수에 대한 로컬 변수를 생성할 때 Swift는 value witness table을 사용하여 잠재적으로 힙에 필요한 버퍼를 할당하고 할당 소스(source of assignment)에서 목적지로 복사를 실행한다.

Implementation of Generic Methods



One shared implementation

Uses Protocol/Value Witness Table

One type per call context: passes tables

- 그리고 로컬 매개변수에 대해 draw 메서드를 실행할 때와 유사하게, 전달된 protocol witness table을 사용하고 테이블 내의 고정 오프셋의 draw 메서드를 찾아 구현으로 이동한다.

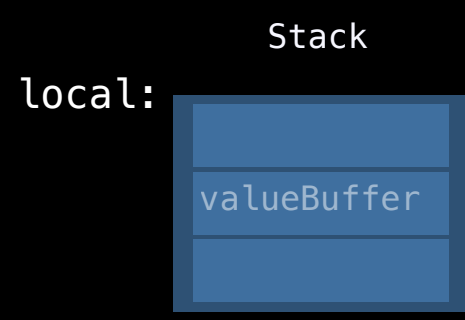
  여기에는 existential container가 없다고 방금 말했다.

  그렇다면 Swift는 이 지역 매개변수를 위해 생성된 지역 변수에 대해 필요한 메모리를 어떻
  할당할까?

Storage of Local Variables

```
func drawACopy<T : Drawable>(local : T) {
  local.draw()
}

drawACopy(Point(…))
```

Value Buffer: currently 3 words



- 바로, 스택에 valueBuffer를 할당한다.

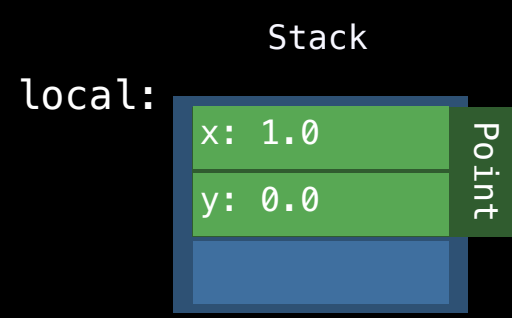
  다시 말하지만, 이 valueBuffer는 3 word이다.

Storage of Local Variables

```
func drawACopy<T : Drawable>(local : T) {
  local.draw()
}

drawACopy(Point(…))
```

Value Buffer: currently 3 words

Small values stored inline



- Point와 같은 작은 값은 valueBuffer에 맞는다.

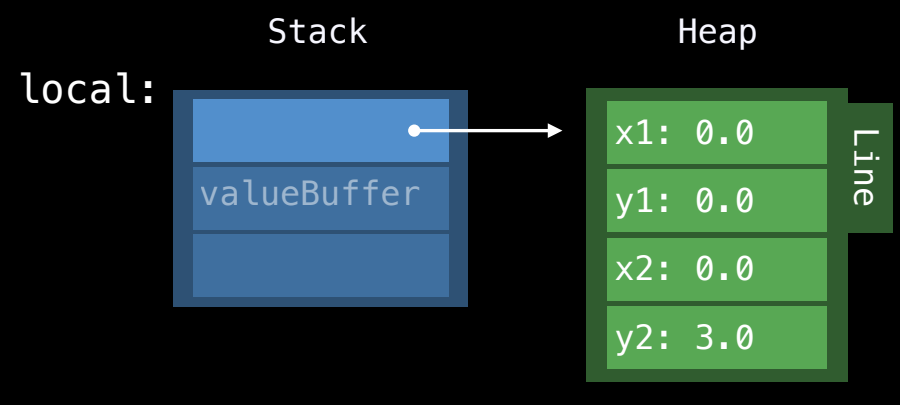
Storage of Local Variables

```
func drawACopy<T : Drawable>(local : T) {
  local.draw()
}

drawACopy(Line(…))
```

Value Buffer: currently 3 words

Small values stored inline

Large values stored on heap



- Line과 같은 큰 값은 다시 힙에 저장되고 local existential container 내부에 해당 메모리에 대한 포인터를 저장한다.

  그리고 이 모든 것은 value witness table 사용을 통해 관리된다.

  이제 "여기에서 프로토콜 타입을 사용하지 않았을 수 있었을까" 라는 질문을 던질 수 있다.

  이 정적 형태의 다형성은 제네릭의 특수화라고 하는 컴파일러 최적화를 가능하게 한다.

Faster?

Specialization of Generics

```
func drawACopy<T : Drawable>(local : T) {
  local.draw()
}

drawACopy(Point(…))
```

Static polymorphism: uses type at call-site

- 다시 말하지만, 여기에 generic 매개변수를 사용하는 함수 drawACopy가 있으며 해당 메서드
  호출하면서 함수에 Point를 전달한다.

  그리고 우리는 정적 다형성을 가지고 있으므로 호출 site에 하나의 타입이 존재한다.

  Swift는 해당 타입을 사용하여 함수의 generic 매개변수를 대체하고 해당 타입에 고유한 해
  함수 버전을 생성한다.

  따라서 여기에 Point 타입의 매개변수를 사용하는 Point 함수 drawACopy가 있으며 해당 함수
  부의 코드는 다시 해당 타입에 대해 고유하다.

  그리고 Kyle이 보여준 것처럼 이것은 정말 빠른 코드가 될 수 있다.

Specialization of Generics

```
func drawACopyOfAPoint(local : Point) {
  local.draw()
}
func drawACopyOfALine(local : Line) {
  local.draw()
}

drawACopyOfAPoint(Point(…))
drawACopyOfALine(Line(…))
```

Static polymorphism: uses type at call-site

Creates type-specific version of method

Version per type in use

- Swift는 프로그램의 호출 사이트에서 사용되는 타입별로 버전을 생성한다.

  따라서 Point와 Line에 대해 drawACopy함수를 호출하면 해당 함수의 두 가지 버전으로 맞춤화하고 생성한다.

  잠깐! 이것은 코드 크기를 많이 증가시킬 가능성이 있다.

  그러나 사용할 수 없는 정적 타이핑 정보는 적극적인 컴파일러 최적화를 가능하게 하기 때문에 Swift는 실제로 잠재적인 방향으로 코드 크기를 줄일 수 있다.

Specialization of Generics

```
func drawACopyOfAPoint(local : Point) {
  local.draw()
}
func drawACopyOfALine(local : Line) {
  local.draw()
}

let local = Point()
local.draw()
drawACopyOfALine(Line(…))
```

Static polymorphism: uses type at call-site

Creates type-specific version of method

Version per type in use

Can be more compact after optimization

- Point 메서드 함수의 drawACopy를 인라인한다.

Specialization of Generics

```
func drawACopyOfAPoint(local : Point) {
  local.draw()
}
func drawACopyOfALine(local : Line) {
  local.draw()
}

Point().draw()

drawACopyOfALine(Line(…))
```

Static polymorphism: uses type at call-site

Creates type-specific version of method

Version per type in use

Can be more compact after optimization

- 그런 다음 코드에 더 많은 컨텍스트가 있으므로 코드를 추가로 최적화한다.

  그래서 함수 호출은 기본적으로 이 한 줄로 줄어들 수 있고 Kyle이 우리에게 보여줬듯이 이것은 draw 구현으로 훨씬 더 줄일 수 있다.

Specialization of Generics

```
Point().draw()

Line().draw()
```

Static polymorphism: uses type at call-site

Creates type-specific version of method

Version per type in use

Can be more compact after optimization

- 이제 Point 메서드의 drawACopy가 더 이상 참조되지 않으므로 컴파일러는 이를 제거 하고 Line 예제에 대해 유사한 최적화를 수행한다.

  따라서 이 컴파일러 최적화가 반드시 코드 크기를 증가시키는 것은 아니다.

  일어날 수있는 것일뿐 반드시 그런 것은 아니다.

  specialization이 어떻게 작동하는지 살펴 보았지만 한 가지 질문은 "언제 발생합니까?"이다

When Does Specialization Happen?

Infer type at call-site

Definition must be available

main.swift

```
struct Point { … }
let point = Point()
drawACopy(point)
```

- Point를 정의한 다음 해당 타입의 지역 변수를 생성하였다.

  point는 이를 Point로 초기화한 다음 해당 Point를 drawACopy함수에 대한 인자로 전달한다.

  이제 이 코드를 specialization하기 위해 Swift는 이 호출 site에서 타입을 유추 할 수 있어야 한다.

  로컬 변수를 보고, 초기화로 돌아가 Point로 초기화되었는지 확인할 수 있기 때문에 그렇게 할 수 있다.

  또한 Swift는 specialization 중에 사용되는 타입과 그 자체로 사용 가능한 generic 함수에 대한 정의가 있어야 한다.

  이번에도 마찬가지이다.

  모두 하나의 파일에 정의되어 있다.

  이것은 전체 모듈 최적화(Whole Module Optimization)가 최적화 기회를 크게 향상시킬 수 있는 곳이다.

Whole Module Optimization

Increases optimization opportunity

Point.swift

```
struct Point {
  func draw() {}
}
```

UsePoint.swift

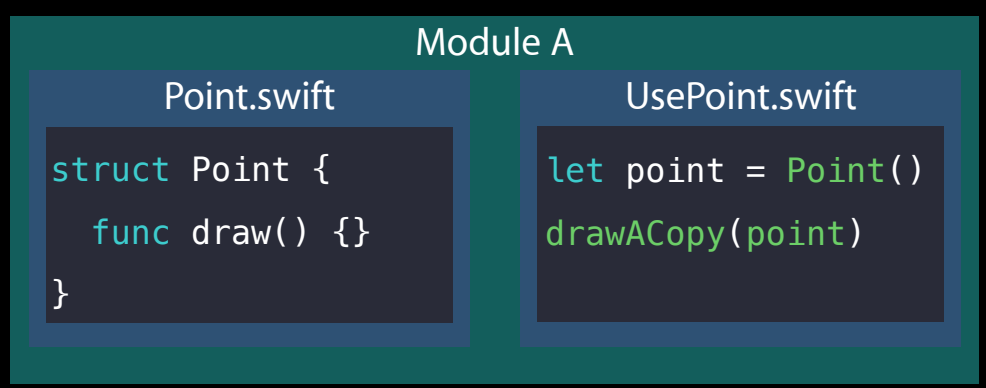
```
let point = Point()
drawACopy(point)
```

- Point 정의를 별도의 파일로 옮겼다고 가정해 보도록 하겠다.

  이제 이 두 파일을 별도로 컴파일하면, UsePoint 파일을 컴파일하려고 할 때 컴파일러가 두 파일을 별도로 컴파일했기 때문에 Point 정의를 더 이상 사용할 수 없다.

Whole Module Optimization

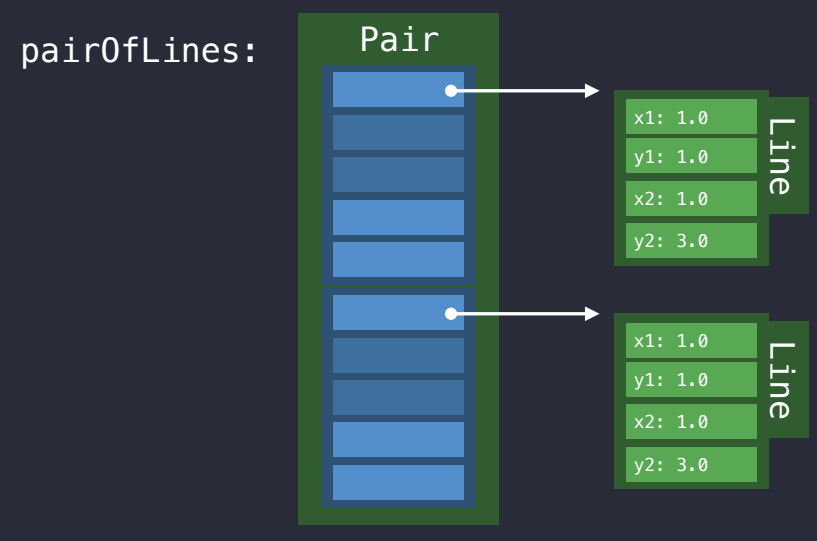
Increases optimization opportunity



- 그러나 전체 모듈 최적화를 통해 컴파일러는 두 파일을 하나의 단위로 함께 컴파일하고 Point 파일 정의에 대한 통찰력을 갖게 되며 최적화가 발생할 수 있다.

  이렇게 하면 최적화 기회가 크게 향상되므로 이제 Xcode 8부터는 전체 모듈 최적화가 기본적으로 활성화된다.

```swift
// Pairs in our program
struct Pair {
   init(_ f: Drawable, _ s: Drawable) {
      first = f ; second = s
   }
   var first: Drawable
   var second: Drawable
}

let pairOfLines = Pair(Line(), Line())

// ...

let pairOfPoint = Pair(Point(), Point())
```



- 이제 한 쌍의 `Line`에 대한 storage에는 두 개의 힙 할당이 필요 하다는 것을 기억해라. 이 프로그램을 볼 때 여기에서 제네릭 타입을 사용할 수 있다는 것을 깨달았다.

```swift
// Pairs in our program using generic types
struct Pair<T : Drawable> {
   init(_ f: T, _ s: T) {
      first = f ; second = s
   }
   var first: T
   var second: T
}

let pairOfLines = Pair(Line(), Line())
// ...
let pairOfPoint = Pair(Point(), Point())
```

- 따라서 쌍을 제네릭으로 정의한 다음 해당 제네릭 타입의 first 및 second 프로퍼티가 이 제네릭 타입을 갖도록 하면 컴파일러는 실제로 동일한 타입의 쌍만 생성 하도록 강제할 수 있다.

  이렇게 하면 나중에 프로그램에서 한 쌍의 Line 묶음에 Point를 저장할 수 없게 된다.

  이것이 우리가 원했던 것이다. 그러나 이것은 성능이 더 좋을까 나쁠까?

Generic Stored Properties

```
struct Pair<T: Drawable> {
    init(_ f: T, _ s: T) {
        first = f ; second = s
    }
    var first: T
    var second: T
}

var pair = Pair(Line(), Line())
pair.first = Point()
```

Type does not change at runtime

Storage inline

- 이번에는 저장 프로퍼티가 generic 타입이다.

  런타임에 타입을 변경할 수 없다고 말한 것을 기억하자.

  generated code가 의미하는 바는 Swift가 enclosing 타입의 저장소를 인라인으로 할당할 수
  다는 것이다.

  따라서 Line 쌍을 만들 때 Line에 대한 메모리는 실제로 enclosing 쌍의 인라인으로 할당된다

  추가 힙 할당이 필요하지 않다.

Performance of Generic Code

Unspecialized

```
func drawACopy<T : Drawable>(local : T) {
  local.draw()
}

drawACopy(Point(…))
```

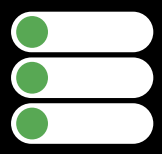
Specialized

```
func drawACopyOfAPoint(local : Point) {
  local.draw()
}
func drawACopyOfALine(local : Line) {
  local.draw()
}

drawACopyOfAPoint(Point(…))
drawACopyOfALine(Line(…))
```

- 나중에 다른 타입의 값을 저장프로퍼티에 저장할 수도 없다.

  value witness 및 protocol witness table을 사용하여 specialized되지 않은 코드가 작동하는 방식과 컴파일러가 generic 함수의 타입별 버전을 생성하는 코드를 specialize할 수 있는 방법을 살폈다.

Specialized Generics—Struct Type



Performance characteristics like struct types

- No heap allocation on copying
- No reference counting
- Static method dispatch

- 이 경우, 방금 보았듯이 generated code는 본질적으로 이 함수를 구조체로 작성한 것과 같기 때문에 구조체 타입을 사용하는 것과 동일한 성능 특성을 갖는다.

  구조체 타입의 값을 복사할 때 힙 할당이 필요하지 않다.

  구조체에 참조가 포함되어 있지 않으면 참조 카운팅이 없다.

  또한 컴파일러 최적화를 추가로 가능하게 하고 런타임 실행 시간을 줄이는 정적 메서드 디스패치를 가지고 있다.

Specialized Generics—Class Type

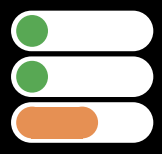
Performance characteristics like class types

- Heap allocation on creating an instance
- Reference counting
- Dynamic method dispatch through V-Table

- 이를 클래스 타입과 비교해보면 클래스 타입을 사용하면 힙 할당 및 인스턴스 생성, 값 전달을 위한 참조 카운팅 , V-Table을 통한 동적 디스패치같은 클래스와 유사한 특성을 갖게 된다.

  이제 작은 값을 포함하는 특수화되지 않은 generic 코드를 살펴보자.

Unspecialized Generics—Small Value



No heap allocation: value fits in Value Buffer

No reference counting

Dynamic dispatch through Protocol Witness Table

- 스택에 할당된 valueBuffer에 작은 값이 들어맞기 때문에 지역 변수에 힙 할당이 필요하지 않다.

  값에 참조가 포함되지 않은 경우 참조 카운팅도 없다.

  그러나 witness table을 사용하여 모든 잠재적 호출 site에서 하나의 구현을 공유하게 된다.

Unspecialized Generics—Large Value

Heap allocation (use indirect storage as a workaround)

Reference counting if value contains references

Dynamic dispatch through Protocol Witness Table

- 큰 값과 generic 코드를 사용하는 경우 힙 할당이 발생한다.
- 그러나 이전에 간접 저장 기술을 해결 방법으로써 사용 하는 것을 보여줬었다.
- 큰 값에 참조가 포함된 경우 참조 카운팅이 생기고, 동적 디스패치의 힘을 얻는다. 즉, 코드 전체에서 하나의 일반 구현을 공유할 수 있다.
- 그래서 우리는 오늘 구조체와 클래스의 성능 특성이 어떻게 생겼는지, generic 코드가 어떻게 작동하는지, 프로토콜 타입이 어떻게 작동하는지 보았다.
- 이것에서 무엇을 얻을 수 있을까?

Summary

Choose fitting abstraction with the least dynamic runtime type requirements

- struct types: value semantics
- class types: identity or OOP style polymorphism
- Generics: static polymorphism
- Protocol types: dynamic polymorphism

Use indirect storage to deal with large values

- 동적 런타임 타입 요구 사항이 가장 적은, 엔티티에 적합한 추상화를 선택하자.
- 이렇게 하면 정적 타입 검사가 가능하고 컴파일러는 컴파일 타임에 프로그램이 올바른지 확인할 수 있다.
- 또한 컴파일러는 코드를 최적화하기 위한 더 많은 정보를 가지고 있으므로 더 빠른 코드를 얻을 수 있다.
- 따라서 구조체 및 열거형과 같은 값 타입을 사용하여 엔티티를 표현할 수 있다면 value semantic 체계를 얻을 수 있으며 이는 의도하지 않은 상태 공유가 없고 최적화 가능한 코드를 얻을 수 있다.
- 엔티티에 있어, 또는 객체 지향 프레임워크로 작업을 해야해서 클래스를 사용해야 하는 경우 Kyle은 참조 카운팅 비용을 줄이는 몇 가지 기술을 보여주었었다.
- 프로그램 일부를 보다 정적인 형태의 다형성을 사용하여 표현할 수 있는 경우 generic 코드를 값 타입과 결합할 수 있으며 매우 빠른 코드를 얻을 수 있으면서도 해당 코드에 대한 구현도 공유할 수 있다.
- 그리고 Drawable 프로토콜 타입 예제의 배열과 같이 동적 다형성이 필요한 경우 프로토콜 타입을 값 타입과 결합하여, 클래스를 사용하는 것보다 비교적 빠른 코드를 얻을 수 있으면서도 여전히 value semantic을 유지할 수 있다.
- 그리고 프로토콜 타입 또는 제네릭 타입 내부에서 큰 값을 복사하기 때문에 힙 할당에 문제가 발생하는 경우 해당 기술, 즉 Copy on Write와 간접 저장을 사용하여 이 문제를 해결하는 방법을 보여줬었다.

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

	궁금한 점	이해 안된 점
보리		

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

	궁금한 점	이해 안된 점
데릭		

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

	궁금한 점	이해 안된 점
바드	Call stack이란?	