iOS 세미나 5

2023 와플스튜디오 루키 세미나

2023.11.26, 박신홍

오늘 배울 내용

- Authentication Overview
- ARC (Automatic Reference Counting)
- Protocol-oriented Programming
  - Protocols
  - Value Types
  - Generics
- Wrap-up

Authentication
vs Authorization

- Authentication (인증)
  - 당신이 누구인지 알리는 것
- Authorization (인가)
  - 인증된 유저가 어디까지 할 수 있는지 정하는 것

Authentication 방법
크게 두 가지

- Session-based authentication
- JWT authentication

JWT (Json Web Token)

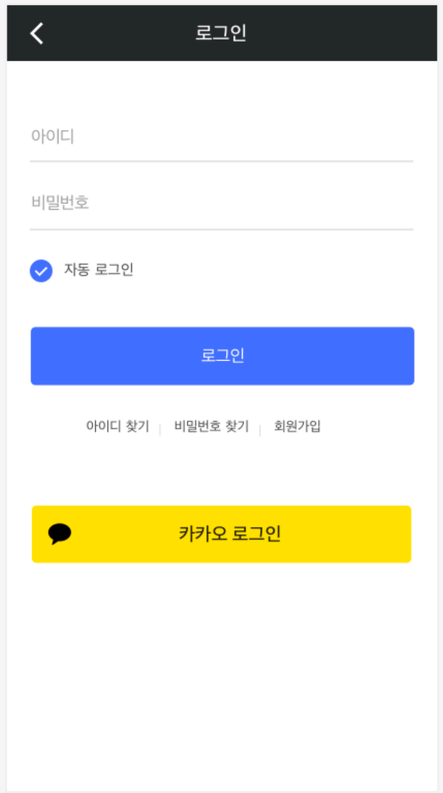
- 유저의 로그인 상태를 서버 DB에 저장하지 않고(Stateless), 안전하게 토큰을 생성할 수 있는 방법
- 토큰 자체에 간단한 정보를 포함시킬 수 있어서 편리함
- 실습: https://jwt.io/

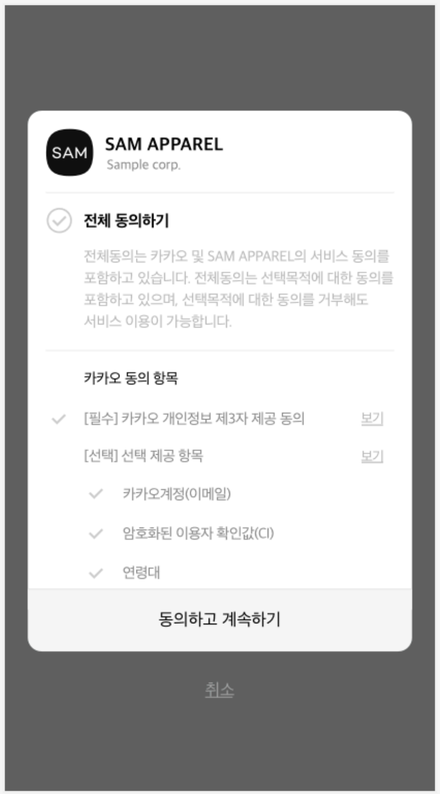
- 클라이언트 입장에서 구현 방법은 매우 간단!
- Retry?

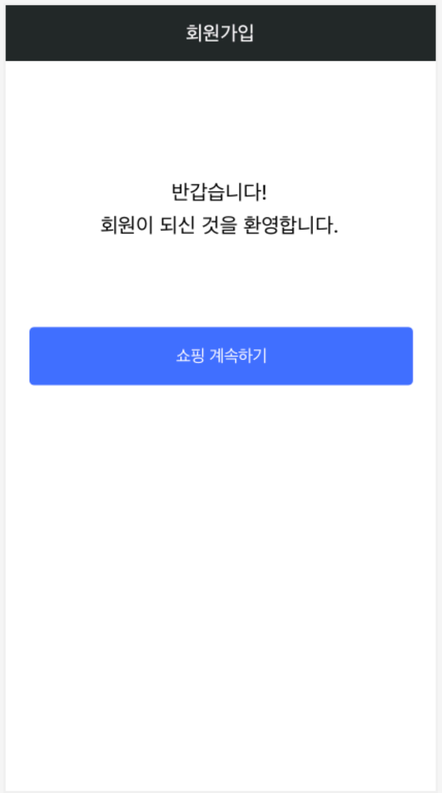
소셜로그인
OAuth 2.0

- 여러분이 자주 사용하는 카카오톡 로그인, 페이스북 로그인, 구글 로그인 등등..
- 대부분 OAuth 2.0이라고 불리는 표준 인증 프레임워크를 바탕으로 구현된 것

소셜로그인 예시







- 소셜로그인 서비스측에서 SDK를 제공하는 경우가 많음
- 공식 문서 참고

https://developers.kakao.com/docs/latest/ko/kakaologin/common

ARC

ARC
Automatic Reference Counting

- Swift의 Garbage Collection 메커니즘
- 아무도 객체를 참조하고 있지 않을 때 비로소 메모리에서 할당 해제하는 것
- 질문) var personB 앞에 weak을 붙이면?

```
class Person {
    let name: String
    init(name: String) {
        self.name = name
    }

    deinit {
        print("deinit")
    }
}

var personA: Person? = Person(name: "박신홍")
var personB = personA
print("personA setting to nil")
personA = nil  // reference count becomes 1
print("personB setting to nil")
personB = nil  // reference count becomes 0
print("personB set to nil")
// Person deallocated
```

```
personA setting to nil
personB setting to nil
deinit
personB set to nil
```

ARC
When to use weak?

- Retain Cycle을 피하기 위해

```
class Parent {
    var children: [Child] = []
    // ...
}

class Child {
    weak var parent: Parent?
    // ...
}
```

- 너무 오래 Retain하는 것을 피하기 위해

```
class ViewController: UIViewController {
    var networkRequest: NetworkRequest?

    func fetchData() {
        networkRequest?.fetchData { [weak self] result in
            self?.handleResult(result)
        }
    }

    func handleResult(_ result: Data) {
        // ...
    }
}
```

💡 Dealloc 해야 하는 타이밍에 하지 못하는 경우를 피하기 위해

Escaping closures

```swift
func performOperation(with closure: () -> Void) {
    print("Before closure")
    closure() // The closure is called within the function
    print("After closure")
}

performOperation {
    print("Inside closure")
}
```

```swift
func performAsyncOperation(with closure: @escaping () -> Void) {
    print("Before delay")
    DispatchQueue.main.async {
        closure() // The closure is called after a delay
    }
    print("After scheduling closure")
}

performAsyncOperation {
    print("Inside closure")
}
```

```swift
class ViewModel {
    var completionhandler: (() -> Void)? = nil

    func fetchData(completion: @escaping () -> Void) {
        completionhandler = completion
    }
}
```

Protocols, Value Types, Generics

Features of Classes

- Encapsulation
- Polymorphism
- Abstraction
- Inheritance
- Reference Semantics
- Data Identity
- Singleton

Limitations of Classes

- Encapsulation
- Polymorphism
- Abstraction
- Inheritance
- Reference Semantics
- Data Identity
- Singleton

- 상속으로 인해 너무 거대해진 서브클래스
- 요구 사항을 정확하게 반영하지 못하는 타입 관계
- **의도하지 않은 공유**
- **성능상 문제점**

Reference Semantics

Classes are reference types

```
class Point {
    init(x: Double, y: Double) { ... }
    var x: Double
    var y: Double
}

var p1 = Point(x: 1.0, y: 2.0)
var p2 = p1

p2.x = 3.0

print(p1.x) // prints 3.0
print(p2.x) // prints 3.0
```

Value Semantics

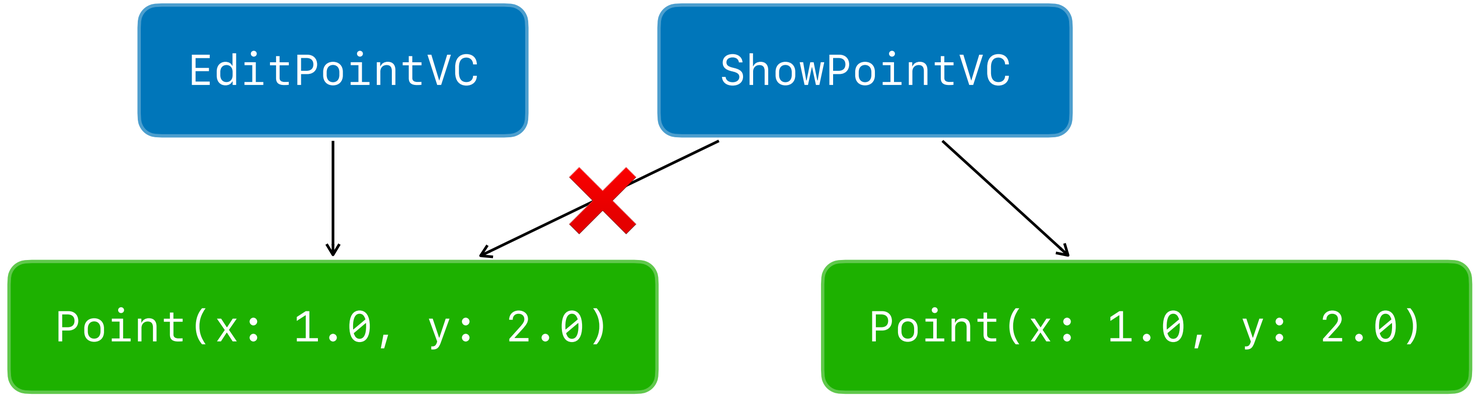
Structs (and enums) are value types

```
struct Point {
    var x: Double
    var y: Double
}

var p1 = Point(x: 1.0, y: 2.0)
var p2 = p1

p2.x = 3.0

print(p1.x) // prints 1.0 (not affected by `p2.x = 3.0`)
print(p2.x) // prints 3.0
```

Value Semantics

...removes unintended sharing

```
struct Point {
    var x: Double
    var y: Double
}
```




```
var p1 = Point(x: 1.0, y: 2.0)

let vc1 = EditPointViewController(point: p1)
let vc2 = ShowPointViewController(point: p1)
```

Value Semantics

...improves local reasoning

```
struct Point {
    var x: Double
    var y: Double
}

var p1 = Point(x: 1.0, y: 2.0)

let vc1 = EditPointViewController(point: p1)
let vc2 = ShowPointViewController(point: p1)

// no need to worry about changes to p1
print(p1)
```

Value Semantics

...improves local reasoning

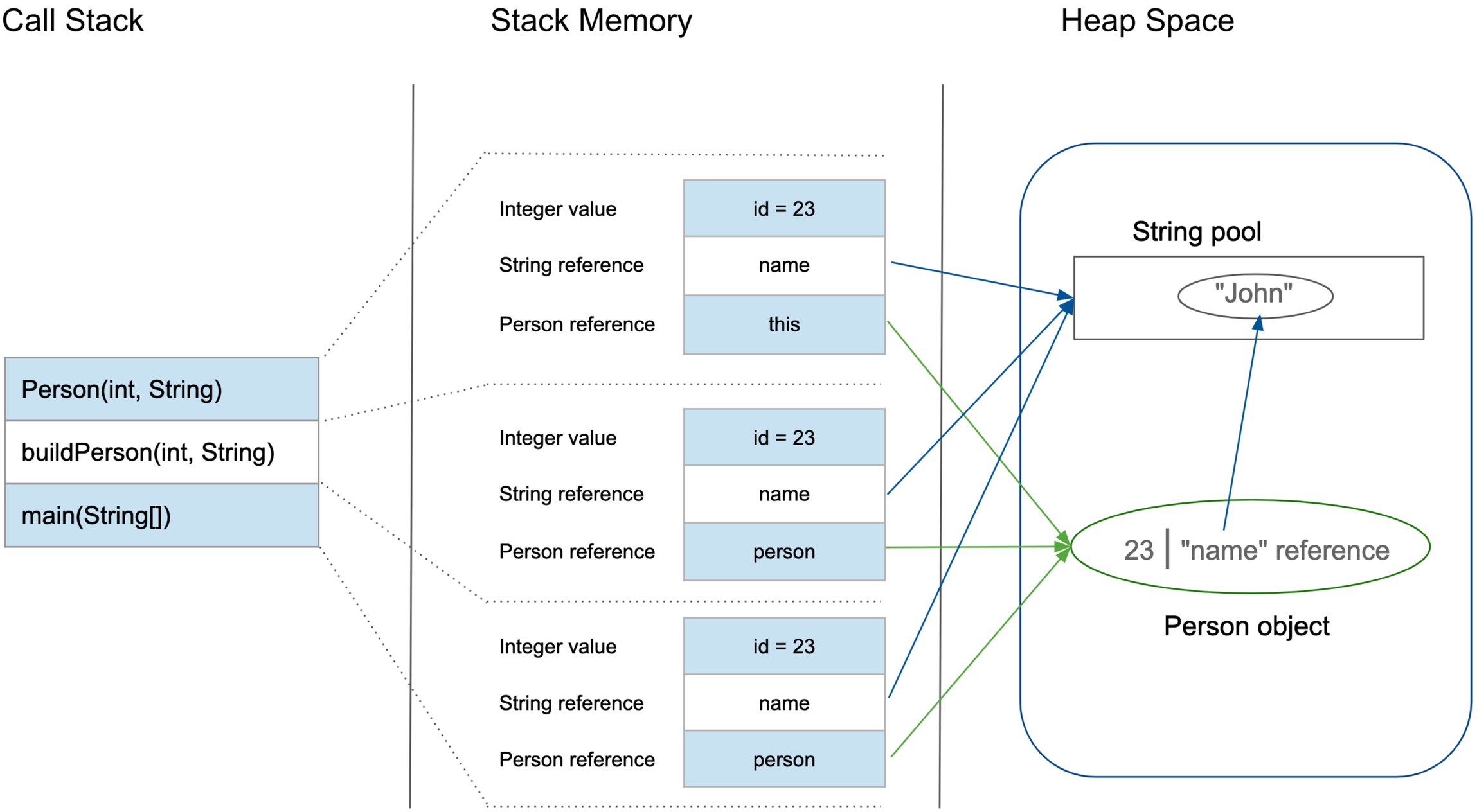
```
struct Point {
    var x: Double
    var y: Double
}
```

```
var p1 = Point(x: 1.0, y: 2.0)

// no need to worry about changes to p1
print(p1)
```

Local Scope 에만 집중할 수 있다.

Value Semantics 🤔

- **Copying cost** of value types?
- What about value types with **variable length**?
- Where and how do value types **allocated**?


Call Stack
Stack Memory
Heap Space
Person(int, String)
buildPerson(int, String)
main(String[])
Integer value
id = 23
String reference
name
Person reference
this
Integer value
id = 23
String reference
name
Person reference
person
Integer value
id = 23
String reference
name
Person reference
person
String pool
"John"
23 | "name" reference
Person object

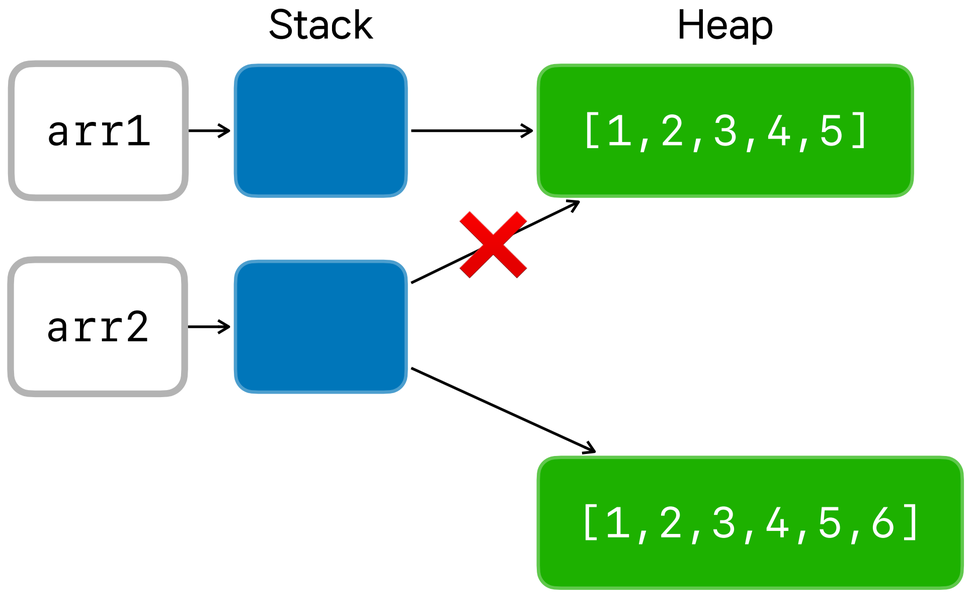
Value Semantics

Copying is cheap with copy-on-write

- Int
- Double
- String
- Struct
- Enum

- Array
- Dictionary
- Set

```
var arr1 = [1,2,3,4,5]

var arr2  = arr1

arr2.append(6)
```



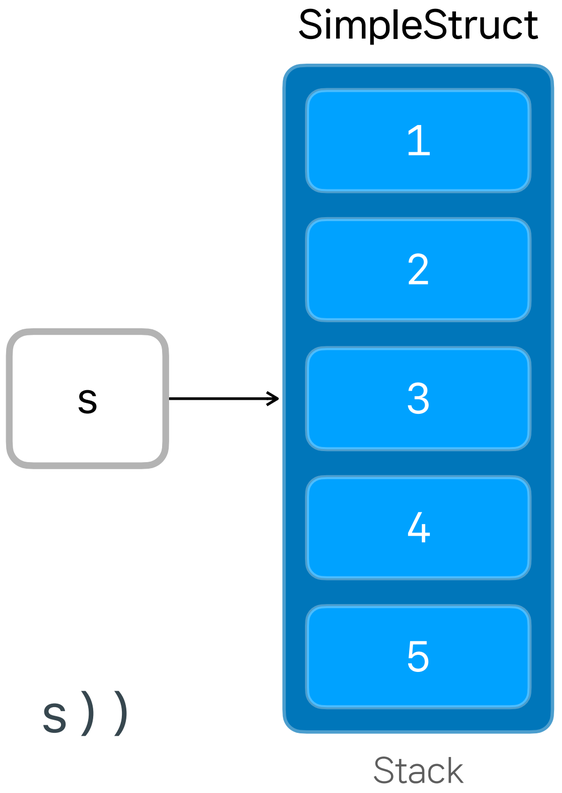
가변 길이 구조체는 힙에 저장되어
copy-on-write 방식으로 작동한다.

Value Semantics

Structs are stored in stack or heap?

- Int
- Double
- String
- Struct
- Enum
- Array
- Dictionary
- Set

```
struct SimpleStruct {
    let a = 1
    let b = 2
    let c = 3
    let d = 4
    let e = 5
}
var s = SimpleStruct()
print(MemoryLayout.size(ofValue: s))
// 40 (in bytes)
```

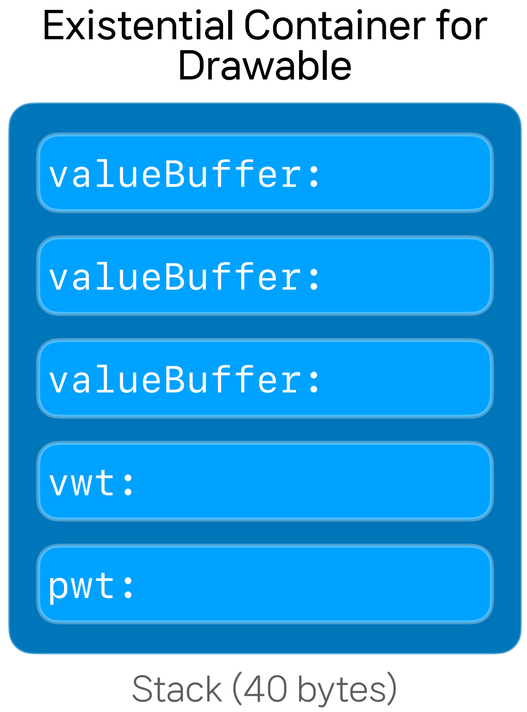


Value Semantics

Structs are stored in stack or heap?

- Int
- Double
- String
- Struct
- Enum
- Array
- Dictionary
- Set

```
protocol Drawable {
    func draw()
}

struct Point: Drawable {
    let x = 10
    let y = 20
    func draw() {}
}

struct Line: Drawable {
    let x1 = 10
    let y1 = 20
    let x2 = 30
    let y2 = 40
    func draw() {}
}

let drawables: [Drawable] = [Line(), Point()]
for d in drawables {
    d.draw()
}
```

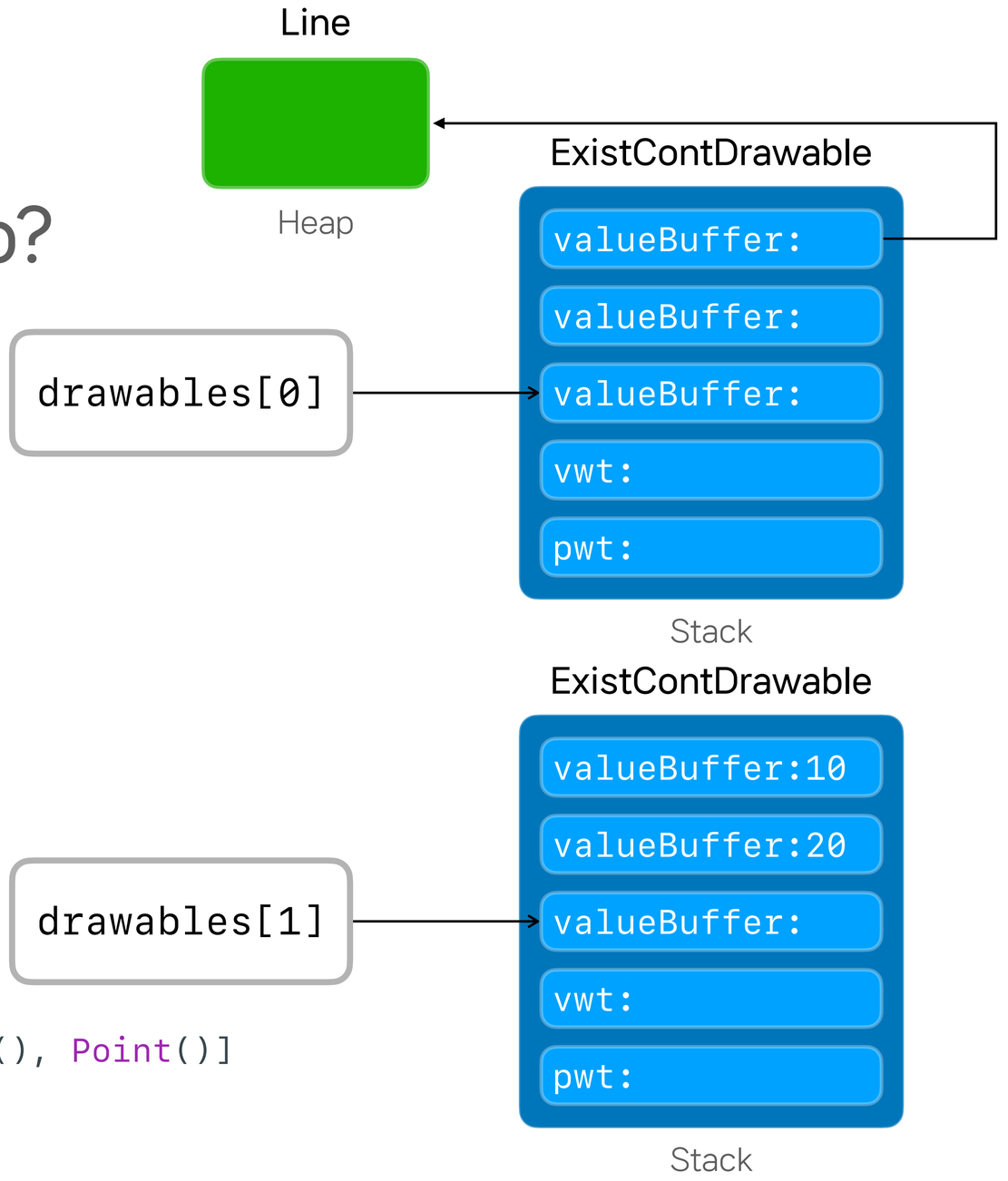


Value Semantics

Structs are stored in stack or heap?

- Int
- Double
- String
- Struct
- Enum
- Array
- Dictionary
- Set

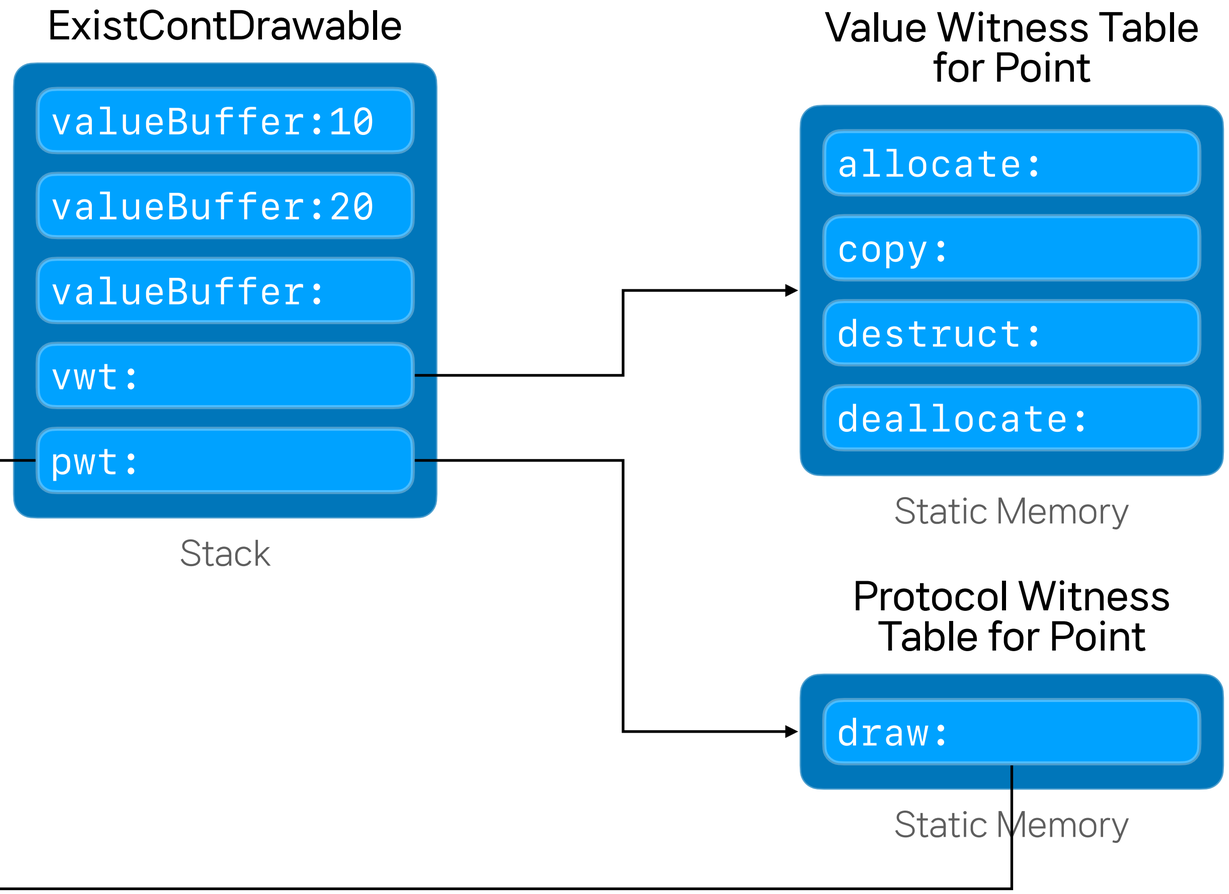
```
protocol Drawable {
    func draw()
}

struct Point: Drawable {
    let x = 10
    let y = 20
    func draw() {}
}

struct Line: Drawable {
    let x1 = 10
    let y1 = 20
    let x2 = 30
    let y2 = 40
    func draw() {}
}

let drawables: [Drawable] = [Line(), Point()]
for d in drawables {
    d.draw()
}
```



Value Semantics

Existential Containers are stored in stack

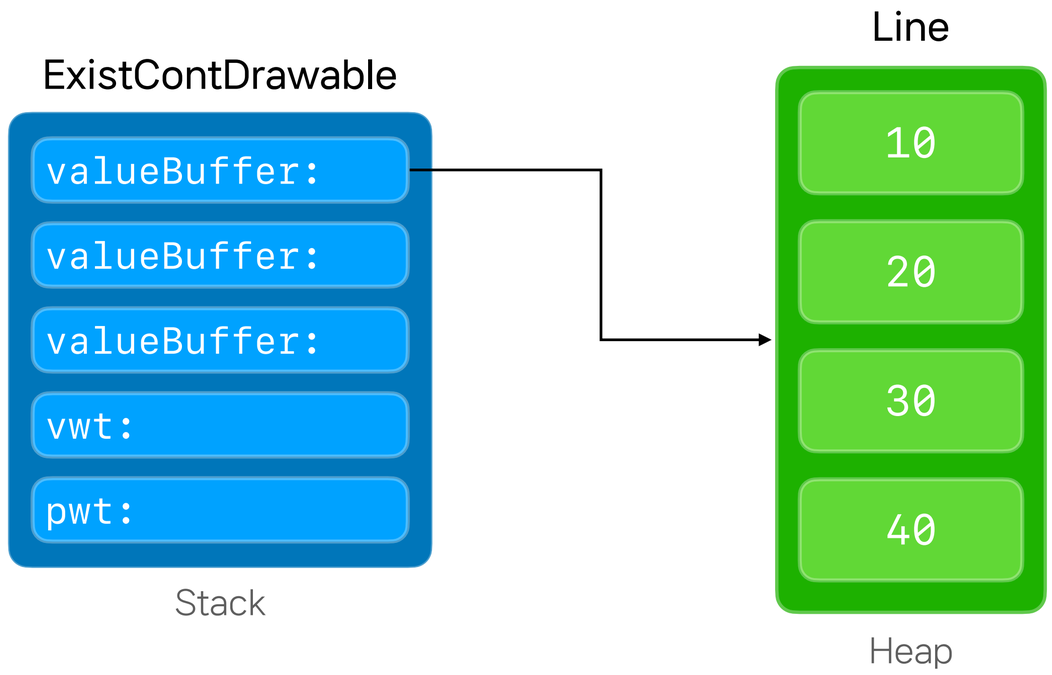
```
struct Point: Drawable {
    let x = 10
    let y = 20
    func draw() {}
}

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



Value Semantics

Large protocol types are stored in heap

```
struct Line: Drawable {
    let x1 = 10
    let y1 = 20
    let x2 = 30
    let y2 = 40
    func draw() {}
}

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



Performance Characteristics

... to bear in mind when programming

- Memory Allocation
- Reference Counting
- Method Dispatch

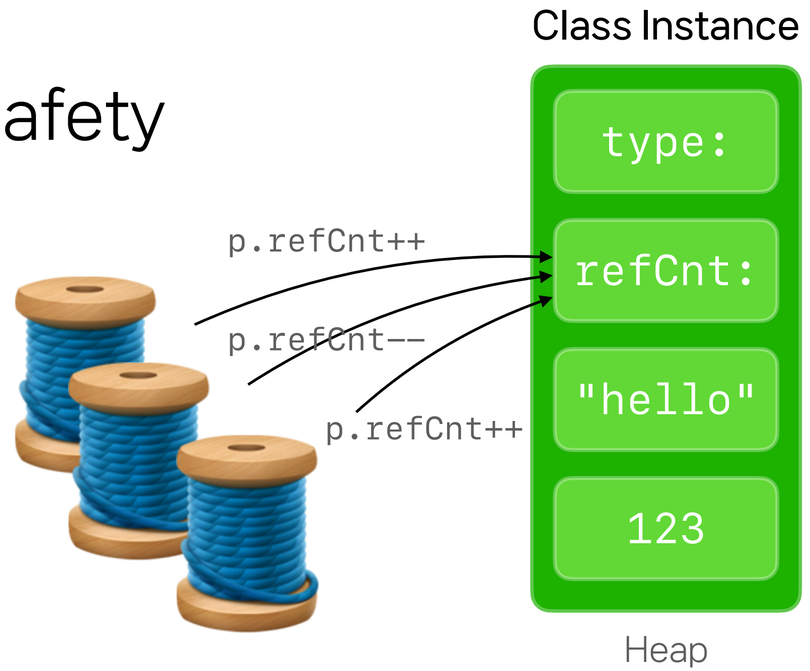
Performance Characteristics

- Memory Allocation
- Reference Counting
- Method Dispatch

Stack Allocation	Heap Allocation
⚡ Fast	🐢 Slow
Only requires incrementing / decrementing the stack pointer	Searching for empty memory blocks
	Optimizing for memory fragmentation
	Synchronizing for thread-safety

Performance Characteristics

- Memory Allocation
- Reference Counting
  - Synchronization overhead for thread-safety
- Method Dispatch



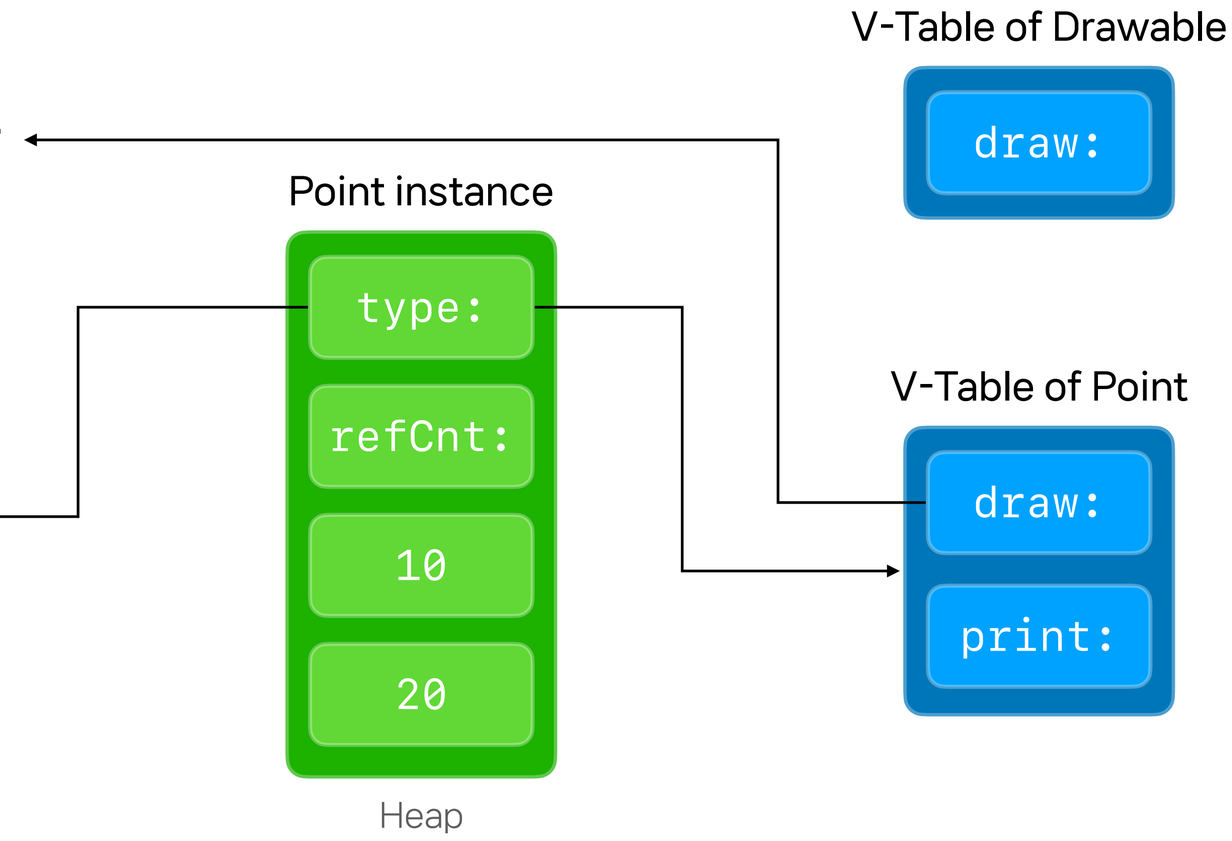
Performance Characteristics

- Memory Allocation
- Reference Counting
- Method Dispatch
  - Dynamic Method Dispatch 🎭
    - **`class`**: V-Table Dispatch
    - **`struct`**: Protocol Witness Table Dispatch
  - Static Method Dispatch ⚡

Performance Characteristics

Dynamic Method Dispatch (Class)

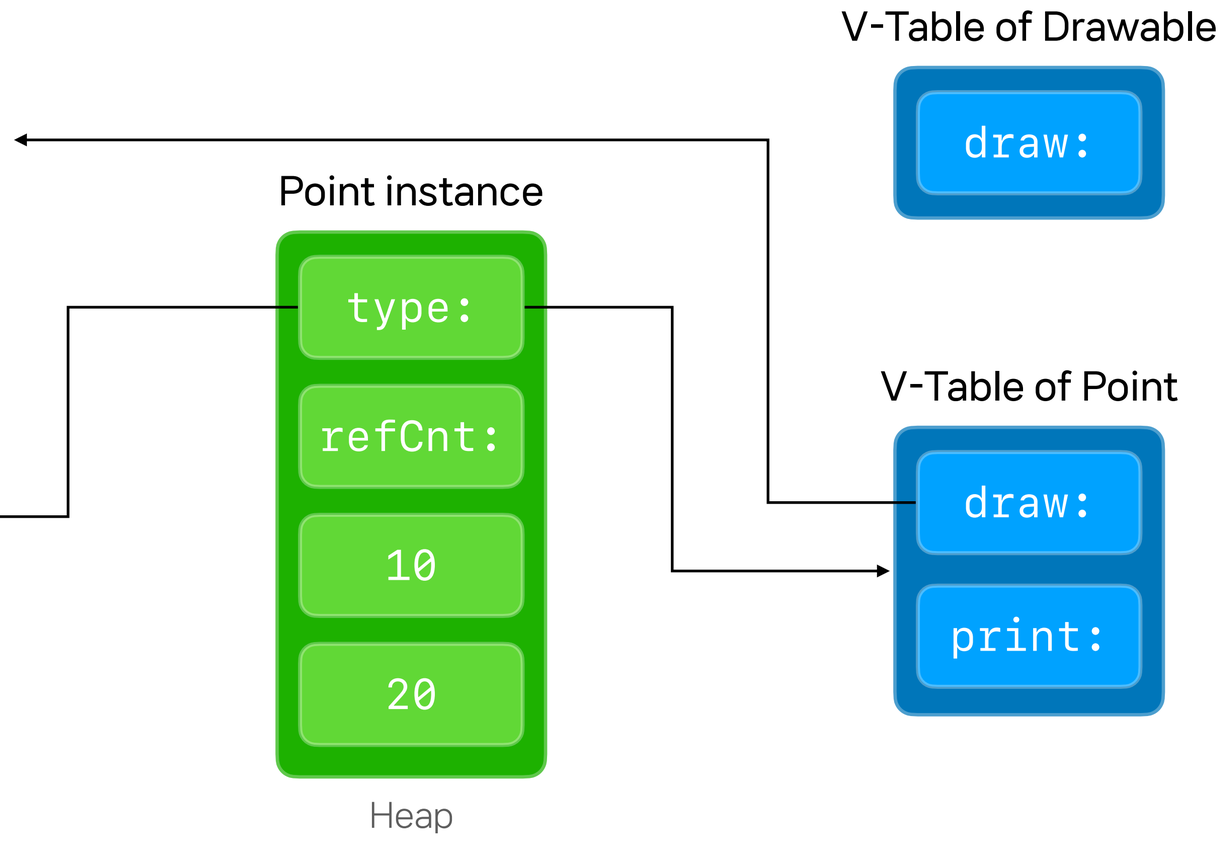
```
class Point: Drawable {
    let x1 = 10
    let y1 = 20
    override func draw() { }
    func print() { }
}

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



Performance Characteristics

Dynamic Method Dispatch (Class)

```
final class Point: Drawable {
    let x1 = 10
    let y1 = 20
    override func draw() { }
    func print() { }
}

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



Performance Characteristics

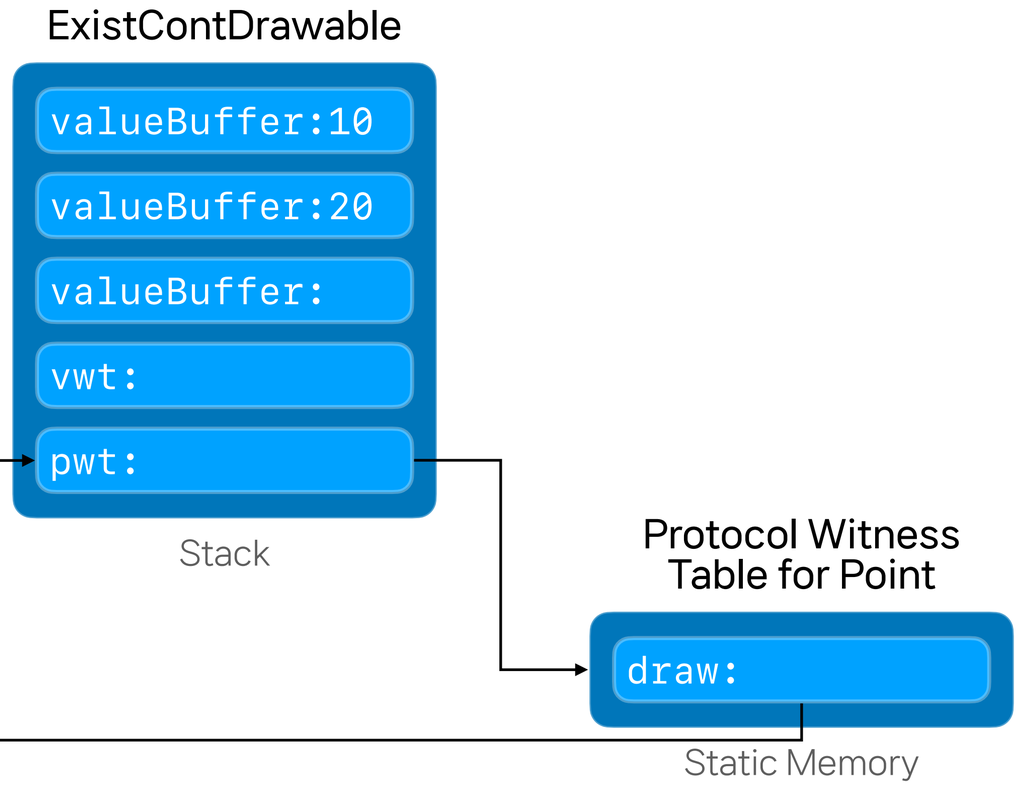
Static Method Dispatch (Class)

```
final class Point {
    let x1 = 10
    let y1 = 20
    func draw() { }
    func print() { }
}

let points: [Point]
for p in points {
    p.draw()
}
```

Performance Characteristics

Dynamic Method Dispatch (Struct)

```
struct Point: Drawable {
    let x = 10
    let y = 20
    func draw() {}
}

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



Performance Characteristics

Static Method Dispatch (Struct)

```
struct Line: Drawable {
    let x1 = 10
    let y1 = 20
    let x2 = 30
    let y2 = 40
    func draw() {
        // implementation of draw
     }
}
```

```
struct Line {
    let x1 = 10
    let y1 = 20
    let x2 = 30
    let y2 = 40
    func draw() {
        // implementation of draw
     }
}
```

```
let l = Line()
l.draw()
```

```
let l = Line()
// implementation of draw
```

Method Inlining ⚡

Performance Characteristics

Summary

- Memory Allocation
- Reference Counting
- Method Dispatch

Generics

Improve performance of protocol types

- Specialization of Generics
- Generic Stored Properties

Generics

Specialization of Generics

```
protocol Drawable { ... }
struct Point: Drawable { ... }
struct Line: Drawable { ... }
```

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

vs 

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

Generics

Specialization of Generics

```
protocol Drawable { ... }
struct Point: Drawable { ... }
struct Line: Drawable { ... }
```

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

is equivalent to ... 🐢

```
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)
}
```

Generics

Specialization of Generics

```
protocol Drawable { ... }
struct Point: Drawable { ... }
struct Line: Drawable { ... }
```

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

drawACopy(Point(...))

drawACopy(Line(...))
```

Specialization enables static dispatch

```
func drawACopyPoint(local: Point) {
    local.draw()
}

func drawACopyLine(local: Line) {
    local.draw()
}
```

Generics

Specialization of Generics

```
protocol Drawable { ... }
struct Point: Drawable { ... }
struct Line: Drawable { ... }
```

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

Method Inlining ⚡

```
drawACopy(Point(...))
drawACopy(Line(...))
```

```
Point(...).draw()
Line(...).draw()
```

```
func drawACopyPoint(local: Point) {
    local.draw()
}

func drawACopyLine(local: Line) {
    local.draw()
}
```

Generics

Generic Stored Properties

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

vs 

```
Pair(Line(), Point()) // ✅
GenericPair(Line(), Point()) // ❌
```

```
struct GenericPair<T: Drawable> {
    init(_ f: T, _ s: T) { ... }
    var first: T
    var second: T
}
```

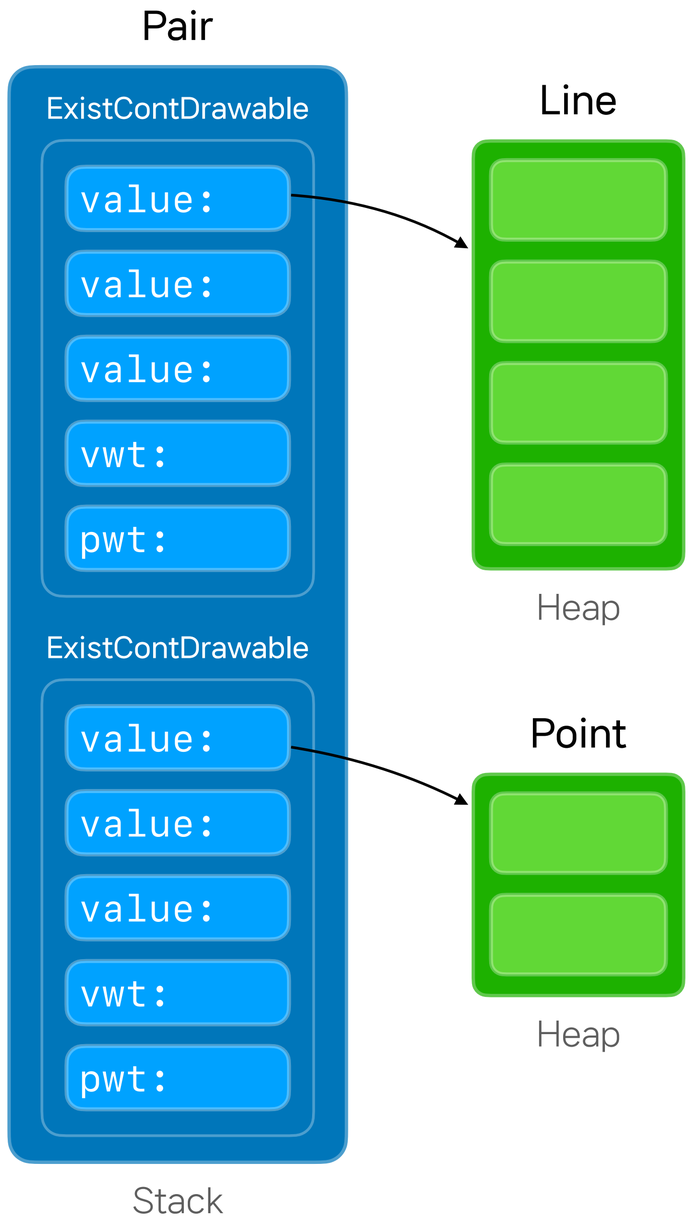
Generics

Non-generic example

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

```
Pair(Line(), Point()) // ✅
```

런타임에 어떤 크기의 타입이 들어올지 모르니
Existential Container를 사용하여 간접 참조 🐢



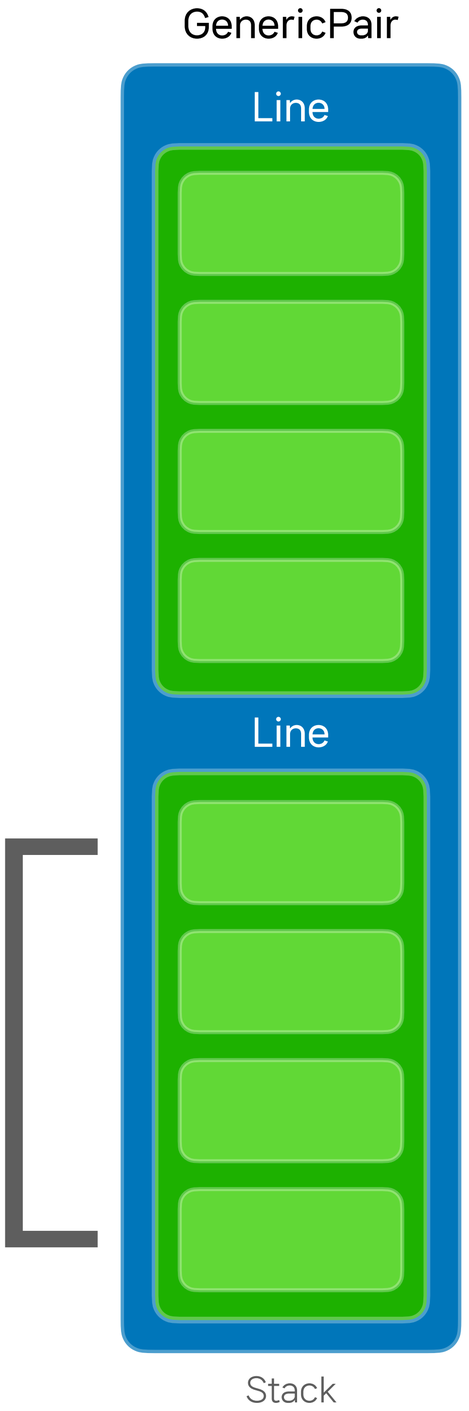
Generics

Generic Stored Properties - Static Polymorphism

```
struct GenericPair<T: Drawable> {
    init(_ f: T, _ s: T) { ... }
    var first: T
    var second: T
}

let gp = GenericPair(Line(), Line())
```

컴파일 타임에 타입이 정해지므로,
Existential Container가 필요하지 않음

No Heap Allocation ⚡
No Reference Counting ⚡
No Dynamic Dispatch ⚡



Conclusion

- Class와 Struct 중 어떤 것을 사용할 것인가?
  - 가능하면 Struct를 선호할 것
    - 이유1: Value Semantic을 따르므로 Local reasoning 향상
    - 이유2: 대부분의 경우 스택 메모리를 사용하므로 성능 관점에서 이득
  - Class는 꼭 필요한 경우에만 사용
    - 예1: 메모리 해제시 Combine 구독 해제가 필요한 경우
    - 예2: 객체가 Delegate로 기능하는 경우
- Generic를 잘 사용하자

앞으로 공부해보면 좋을 내용

- 고급 Swift
  - Opaque Type, Existential Any
  - AssociatedType, Generics
- 동시성 프로그래밍
  - Structured/Unstructured Task, Actors
- 테스트 작성, TDD
- SwiftUI
- CI / CD
  - 배포 및 테스트 자동화
- View/App LifeCycle
- Window의 개념

고생하셨어요