

# [WWDC16] Understanding Swift Performance

## ❖ 배경

Swift에는 다양한 코드 재사용 및 역동성을 위한 다양한 퍼스트 클래스 유형과 다양한 추상화 메커니즘이 존재한다. 그렇다면 작업에 적합한 도구를 선택하려면 무엇을 고려하는게 좋을까?

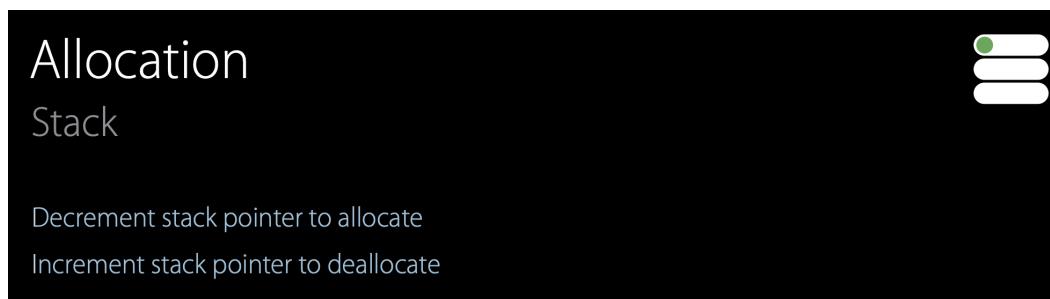
## ❖ 해결방안

Swift의 다양한 추상화 메커니즘의 모델링 의미를 고려한다. 세부적으로 성능은 다음의 세 가지 요소로 그 척도를 가늠할 수 있다.

- 성능 영향 요인

- Memory Allocation

- ▼ Stack



- Stack의 끝에서만 추가(push) 및 제거(pop)
- 따라서, Stack Pointer 증감을 통한 할당 및 해제
  - Stack Pointer를 약간만 줄여 공간을 함으로써 메모리 할당하는 구조
    - 매우 빠른 속도

- ▼ 작동방식

```
// 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

```
// 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

## ▼ Heap

# Allocation Heap

Advanced data structure  
 Search for unused block of memory to allocate  
 Reinsert block of memory to deallocate  
 Thread safety overhead

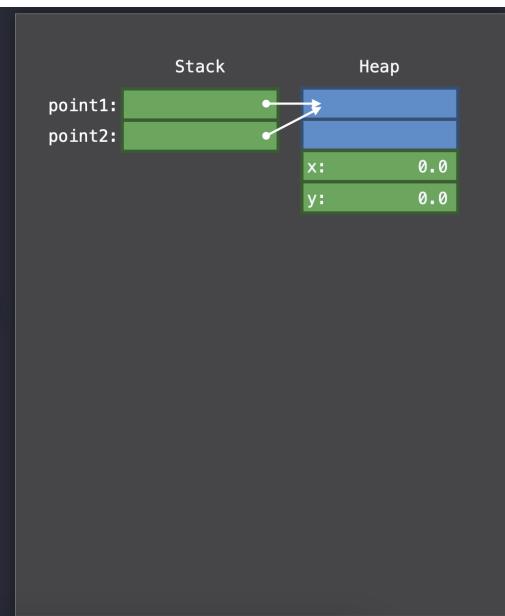
- Advanced Data Structure
  - Stack과 달리 Dynamic Lifetime
- Stack보단 비효율적
  - 메모리 할당을 위해 적절한 크기의 사용하지 않은 블럭을 찾기 위해 Heap Data Structure 검색
  - 메모리 할당 해제를 위해 해당 메모리를 적절한 위치에 재삽입
  - 이는 Stack에서 정수 할당보다 복잡한 작업이나 그렇게 큰 비용은 X
  - 오히려, **무결성 보호**를 위한 lock 또는 기타 동기화 메커니즘이 큰 비용 초래 → Thread Safety Overhead

▼ 작동방식

```
// 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`
```

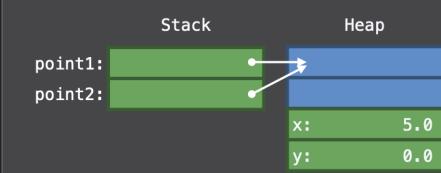


```

// 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`
```

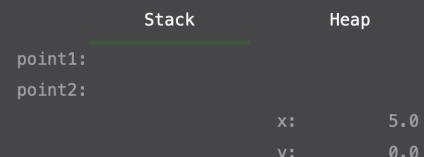


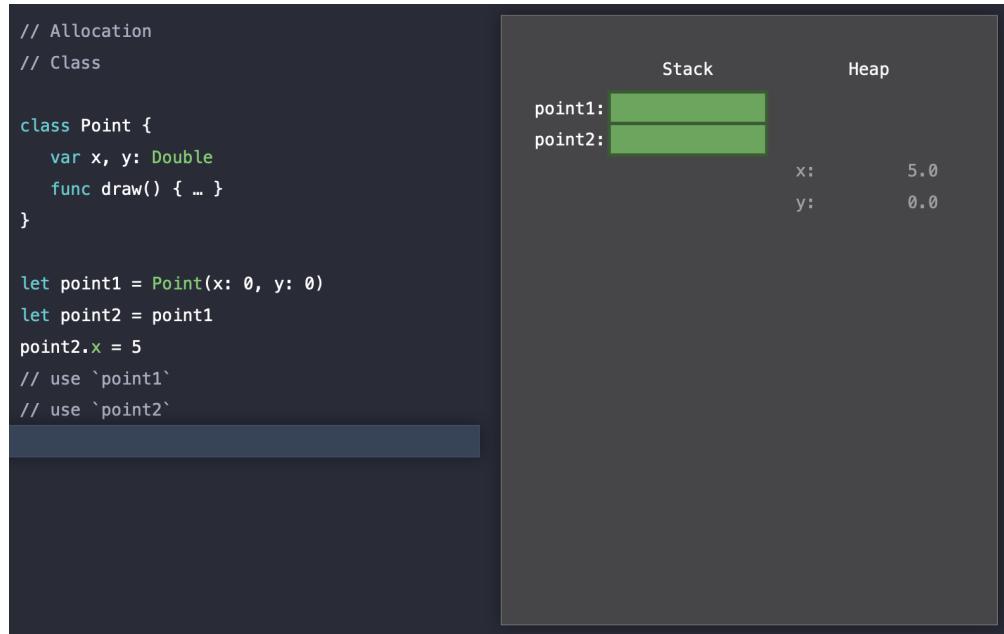
```

// 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`
```

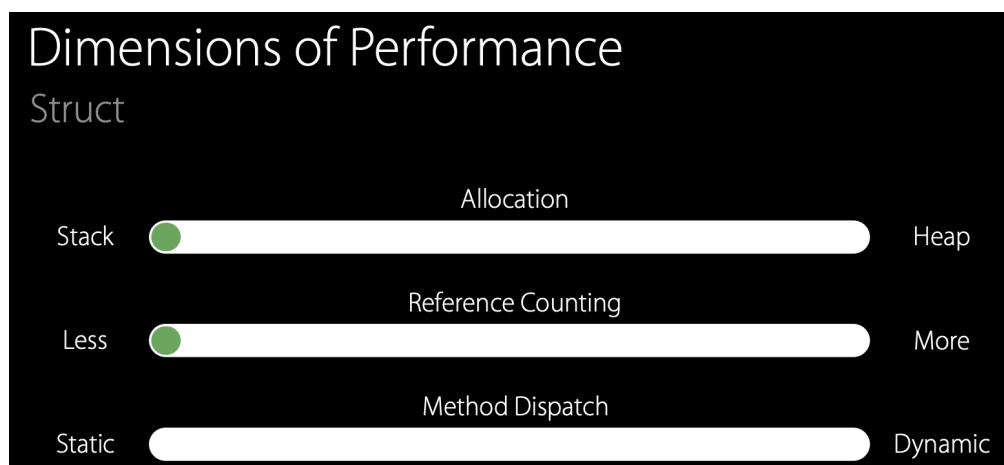




- Reference Counting

- ▼ No

- Struct



- No Reference Counting

- ▼ 작동방식

```
// Reference Counting
// Struct

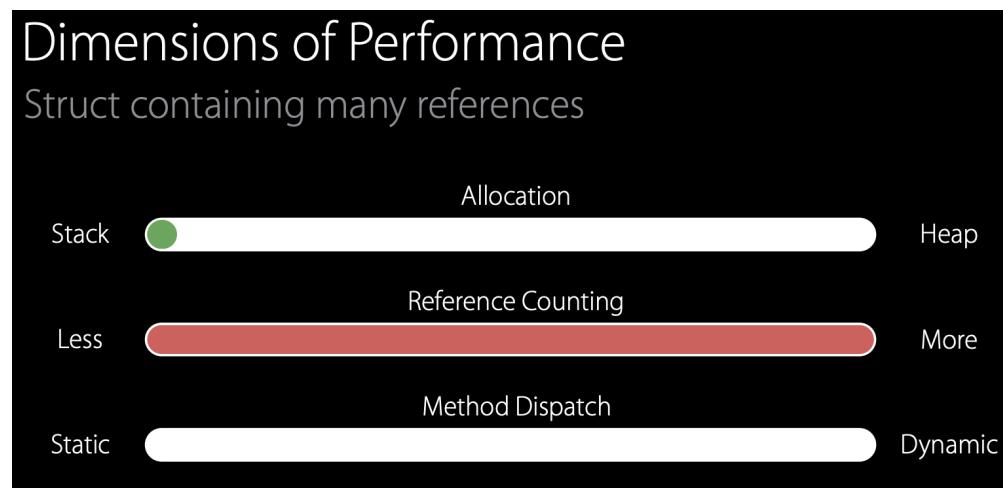
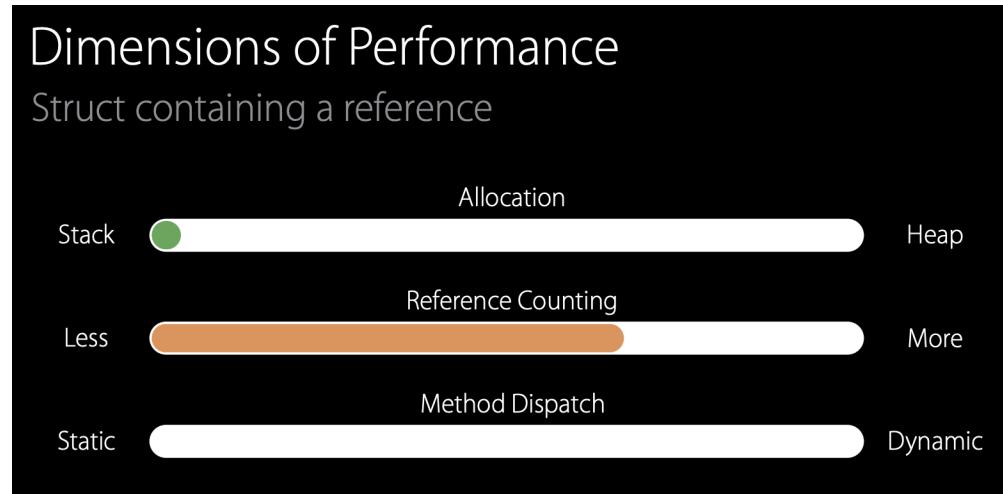
struct Point {
    var x, y: Double
    func draw() { ... }
}

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

Stack

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

- Struct containing references



- struct 내부 reference 타입 프로퍼티에 비례하여 ref count 비용 발생

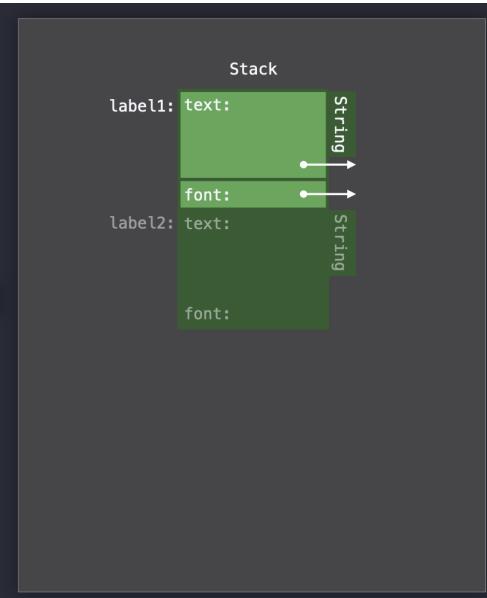
- 따라서, reference 타입의 프로퍼티를 value 타입으로 변경하는 Modeling Technique로 Reference Counting 비용을 줄일 수 있도록 노력

▼ 작동방식

```
// 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`
```

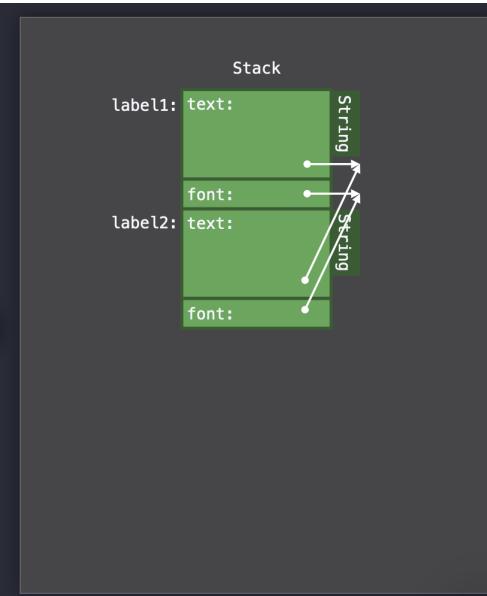


- String은 Value Semantics지만, 내부 storage로 class타입 가짐
- UIFont는 애초에 class 타입

```
// 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`
```

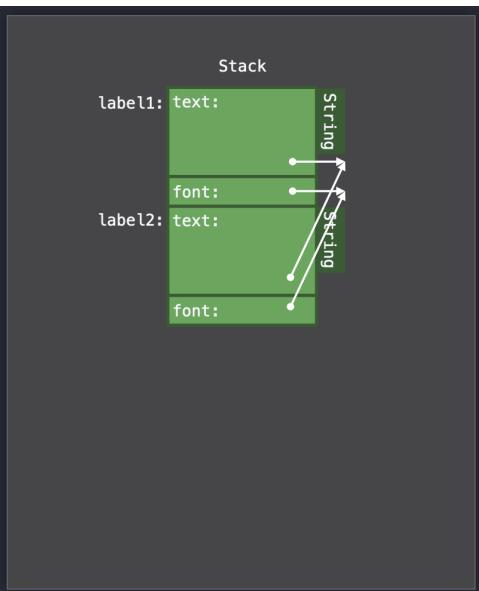


```

// 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)

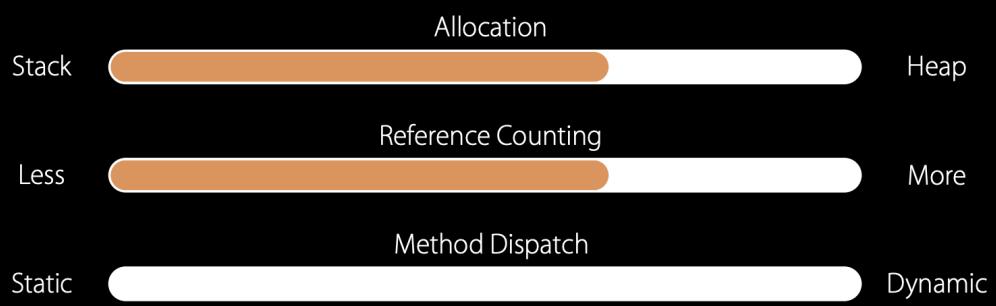
```



▼ Yes

## Dimensions of Performance

### Class



## Reference Counting



There's more to reference counting than incrementing, decrementing

- Indirection
- Thread safety overhead

- 증가 및 감소 실행을 위한 몇 가지 간접 참조 존재

- Heap Allocation과 마찬가지로 Thread Safety Overhead

- 여러 스레드의 힙 인스턴스에 참조를 추가하거나 제거할 수 있기 때문

- 스레드 안전성을 고려 필요
- 실제로 참조 수를 Atomically(=Thread Safe) 증가 및 감소

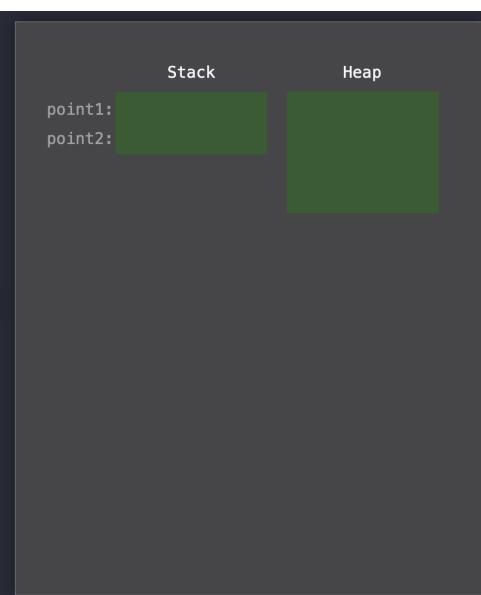
### ▼ 작동방식

- Class

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

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

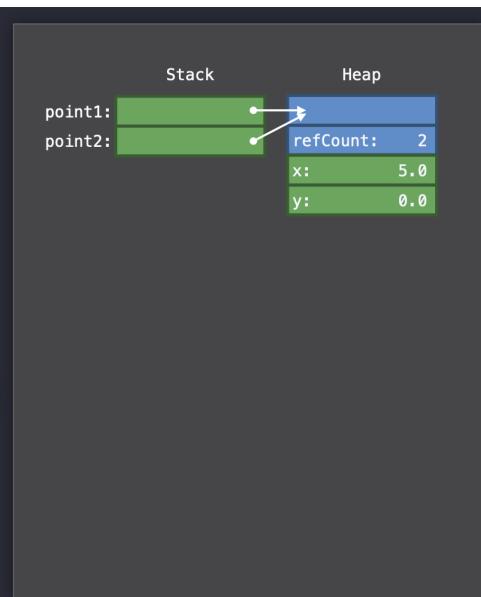
let point1 = Point(x: 0, y: 0)
let point2 = point1
retain(point2)
point2.x = 5
// use `point1`
release(point1)
// use `point2`
release(point2)
```



```
// 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)
```



```

// 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)

```



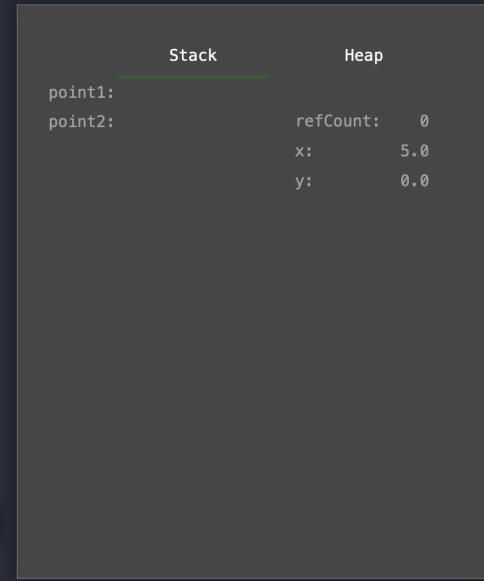
```

// 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)

```



- Method Dispatch(메서드 호출)

- ▼ Static(컴파일 시 호출)

## Method Dispatch

### Static

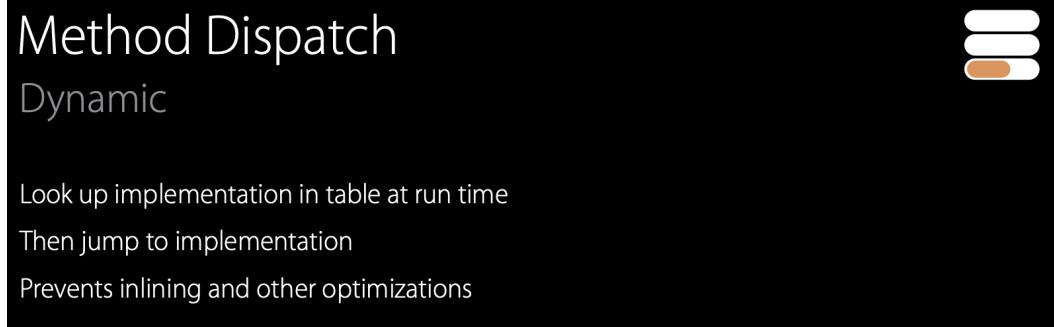
Jump directly to implementation at run time  
Candidate for inlining and other optimizations

- 런타임에 구현부로 바로 이동 가능
- 컴파일러가 실제 구현에 대한 가시성을 가짐
- 즉, 메소드 인라이닝(Method Inlining) 등과 같이 코드 최적화 가능

### 메소드 인라이닝

→ 메소드를 호출 할 때 해당 메소드로 이동하지 않고 메소드의 결과값을 바로 반환하여 성능을 향상 시키는 것입니다.

#### ▼ Dynamic(런타임 시 호출)



- 어떤 구현으로 이동할지 컴파일 타임에서 알 수 없음
- 따라서, 런타임에 실제 구현 조회한 다음 해당 구현으로 jump
- 동적 디스패치 자체로는 indirection은 한번이므로 비용이 많이 들지 않음
- 그러나, 컴파일러의 가시성을 차단하므로 정적 디스패치에서의 최적화 효과는 기대 할 수 없어 비용을 높게 측정
- **Polymorphism(다형성)**과 같은 강력한 기능을 가능케 함

#### • Polymorphism

#### ▼ Class

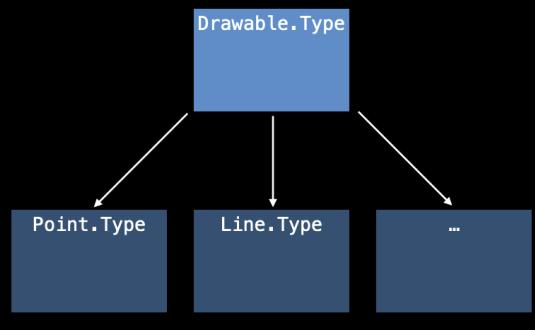
## Inheritance-Based Polymorphism

```
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()
}
```



- 컴파일러는 class에 대한 타입 정보를 가지고 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()
}
```



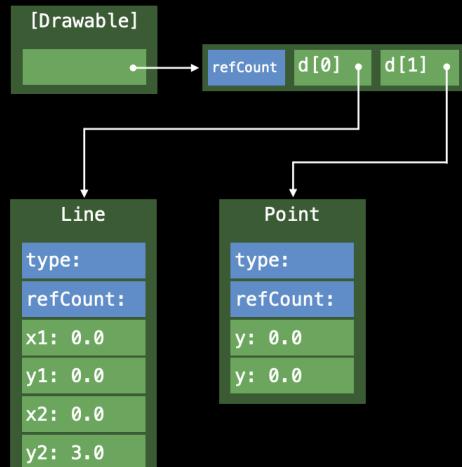
# Polymorphism Through Reference Semantics

```
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()
}
```



- 이 정보에는 class의 가상 함수에 대한 포인터를 가진 **가상 메서드 테이블(V-Table)**이 포함
- 이 테이블은 클래스의 정적 메모리에 저장

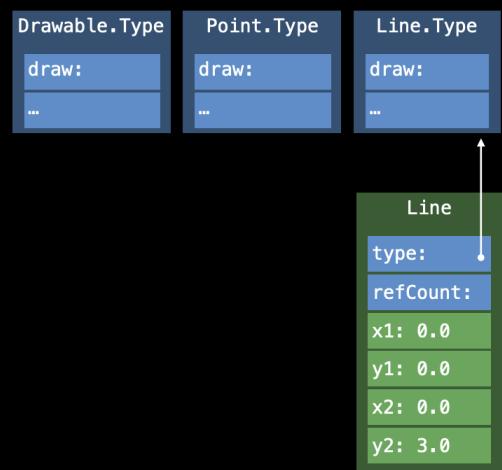
# Polymorphism Through V-Table Dispatch

```
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()
}
```



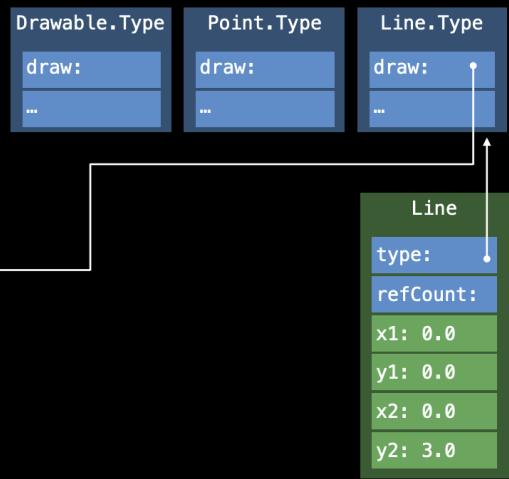
# Polymorphism Through V-Table Dispatch

```
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)
}
```



- 실제 코드에서 특정 가상 함수를 호출할 때, 컴파일러는 해당 클래스의 타입 정보를 통해 V-Table에서 올바른 구현을 찾아 호출
  - `d.type.vtable.draw(d)`
  - 즉, 구현을 찾기 위해 V-Table을 통해 조회하는 것을 엿볼 수 있음
- 이를 통해 다형성을 구현
- 그렇다고, 모든 클래스에 동적 디스패치가 필요한 것은 X
  - 클래스를 subclass로 분류하지 않으려는 경우, **final class**로 표시하여 의도 전달 가능

## ▼ Protocol

- 프로토콜은 V-Table이 존재하지 않음에도 다형성을 어떻게 구현할까?

# No Inheritance Relationship

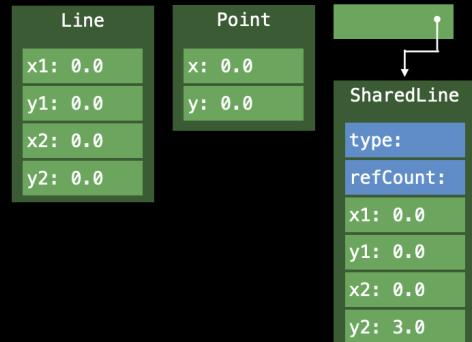
Dynamic dispatch without a V-Table

```
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()
}
```

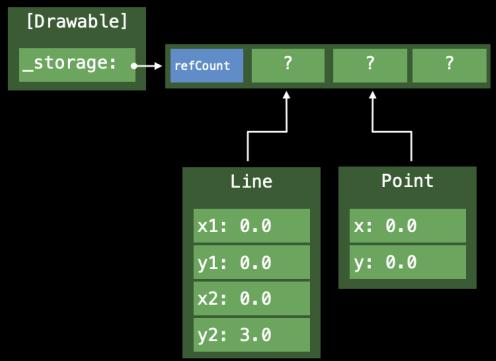


```
protocol Drawable { func draw() }

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()
}
```



- Existential Container

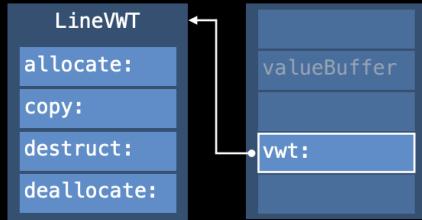
# The Existential Container

## Boxing values of protocol types

Inline Value Buffer: currently 3 words

Large values stored on heap

Reference to Value Witness Table



# The Existential Container

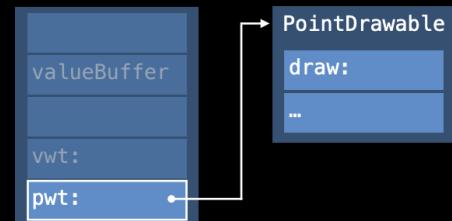
## Boxing values of protocol types

Inline Value Buffer: currently 3 words

Large values stored on heap

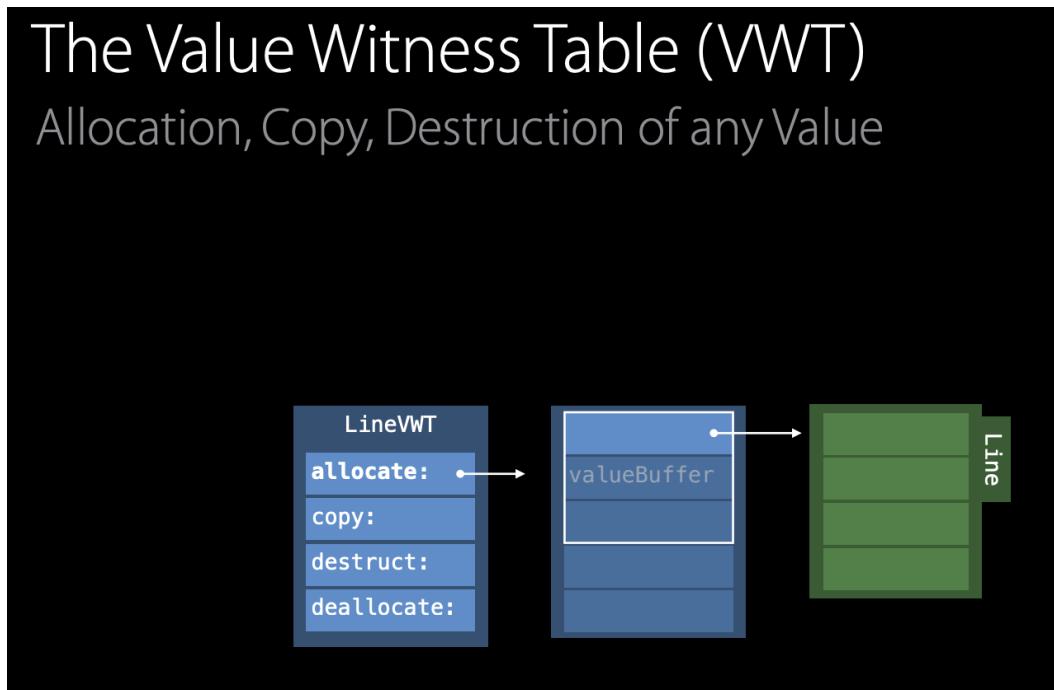
Reference to Value Witness Table

Reference to Protocol Witness Table



- Existential Container는 프로토콜 타입을 나타내는 런타임 객체
- 프로토콜을 채택하는 모든 타입이 해당 프로토콜을 채택한 것처럼 보이게끔 만들
- Existential Container는 값을 저장하고, 해당 값을 사용하는 코드를 작성하기 위해 VWT와 PWT와 함께 사용
- Inline Value Buffer → 3 words
  - 32bit CPU → 1 word = 32bit
  - 64bit CPU → 1 word = 64bit

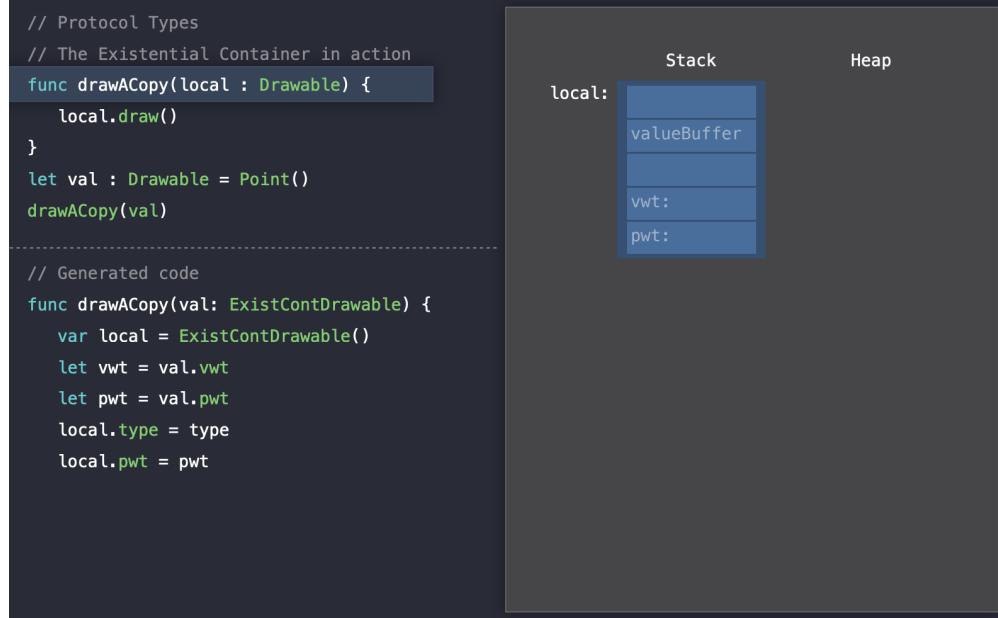
- 3 words 초과시 heap에 저장
  - VWT 참조
  - PWT 참조
- **Value Witness Table(VWT)**



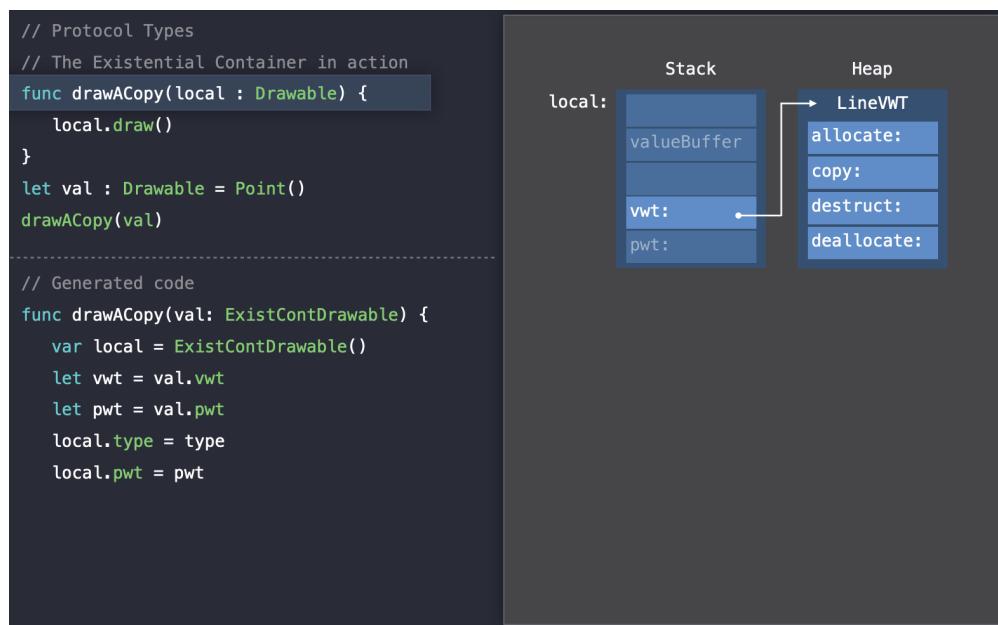
- VWT는 타입이 저장된 메모리 영역의 레이아웃을 나타냄
  - 이를 통해 Swift 컴파일러는 메모리를 효율적으로 사용하고, 코드를 최적화 가능
  - Protocol을 구현하는 타입마다 존재
  - Existential Container 내 메타데이터 영역의 VWT에 포인터를 가지고 있어서 copy 가 되더라도 VWT 데이터를 계속 가져갈 수 있음
- **Protocol Witness Table(PWT)**
- PWT는 프로토콜이 구현되는 방식을 설명하는 테이블
  - 프로토콜의 각 요구사항에 대한 구현체를 가리키는 **함수 포인터**를 저장
  - 즉, 프로토콜을 채택한 타입이 프로토콜의 요구사항을 충족시키기 위해 구현한 **함수**를 PWT에 등록
  - 그러면 Existential Container는 해당 PWT를 가리키고, 이를 통해 **프로토콜을 구현 한 함수를 호출 가능**

▼ 작동방식

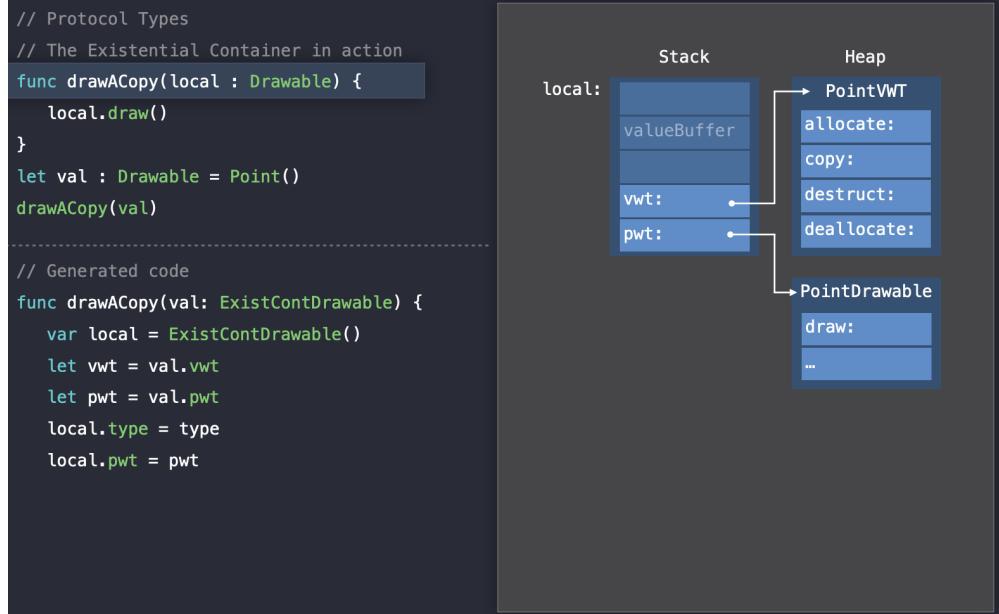
- Existential Container 생성



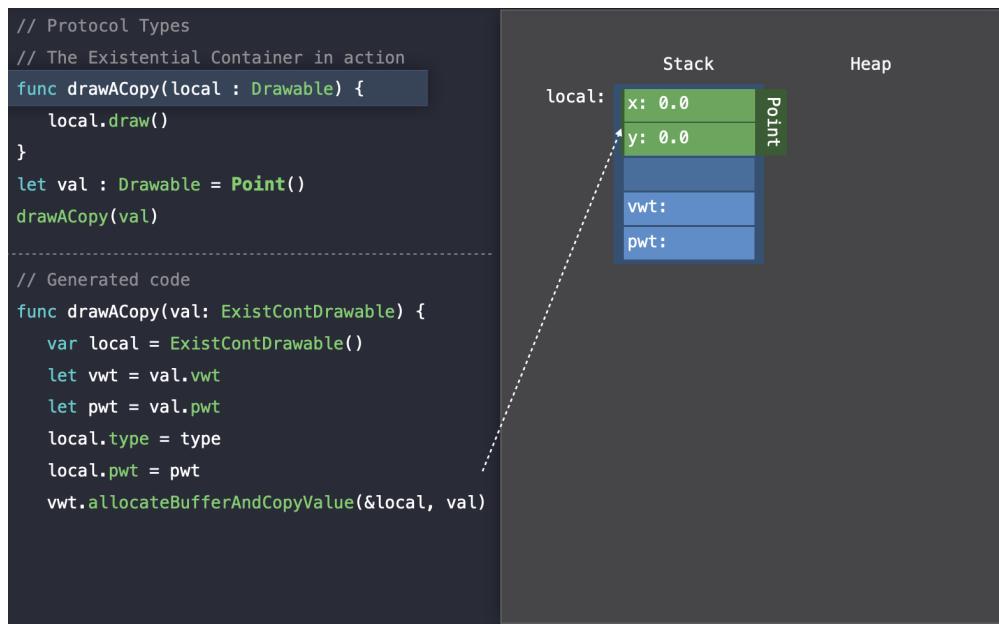
- VWT가 메모리 레이아웃을 정의



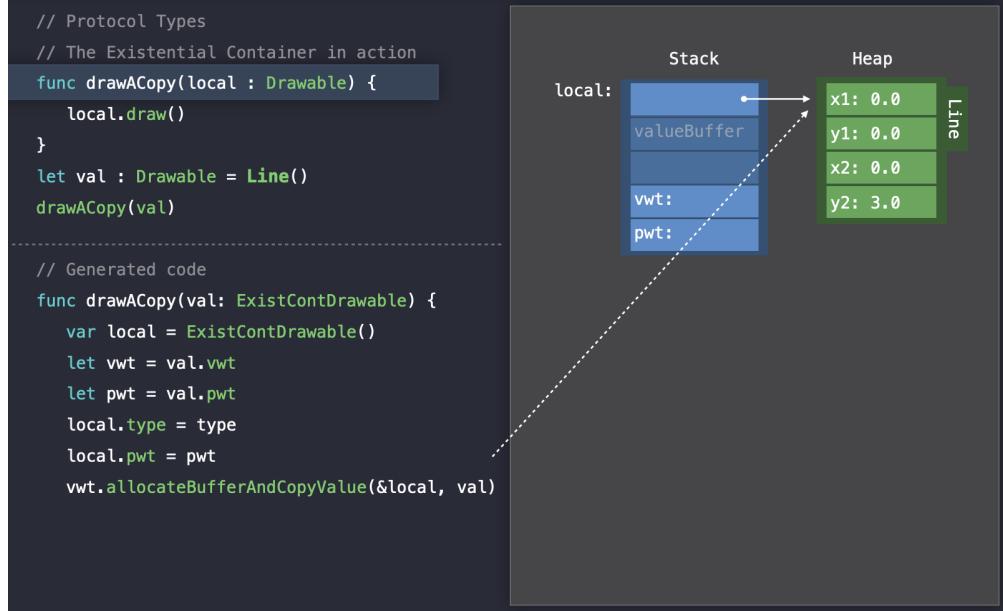
- PWT가 프로토콜을 구현하는 방식을 정의



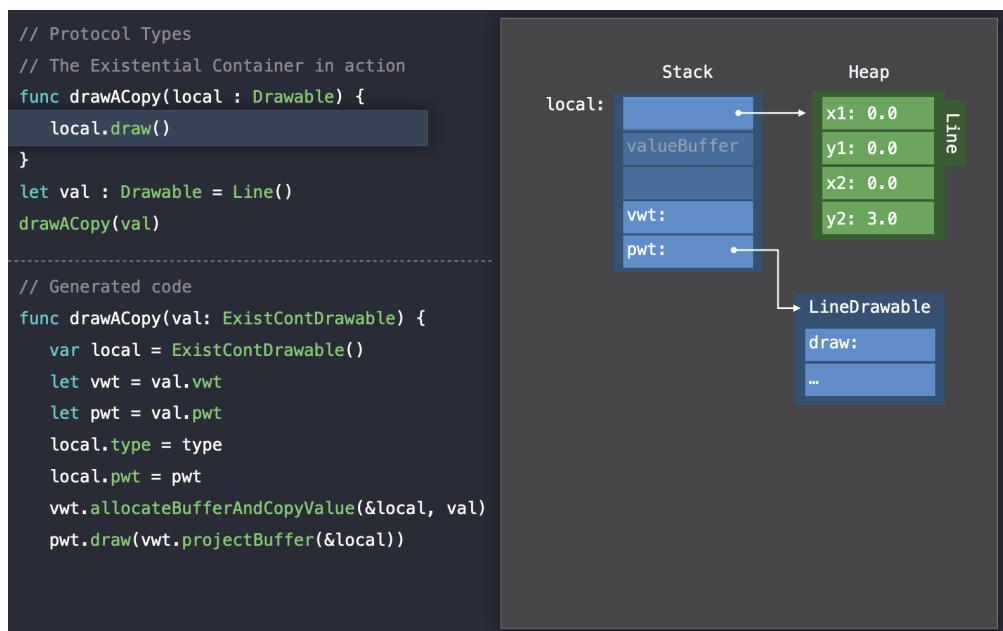
- VWT를 통해 Value Buffer에 들어갈 수 있는지 구분
- 3 words 이내인 Point 타입 인스턴스는 Existential Container에 값 모두 저장



- 3 words를 초과한 Line 타입 인스턴스는 Heap 할당하여 값 저장
- Existential Container에 해당 레퍼런스 저장



- PWT를 통해 method dispatch



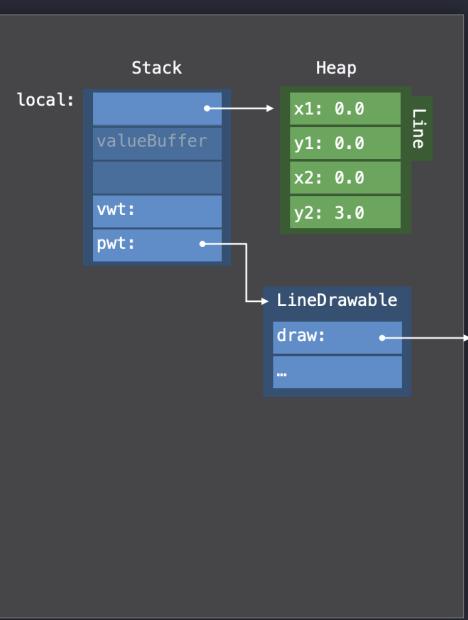
```

// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
    local.draw()
}

let val : Drawable = Line()
drawACopy(val)

-----
// 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))
}

```



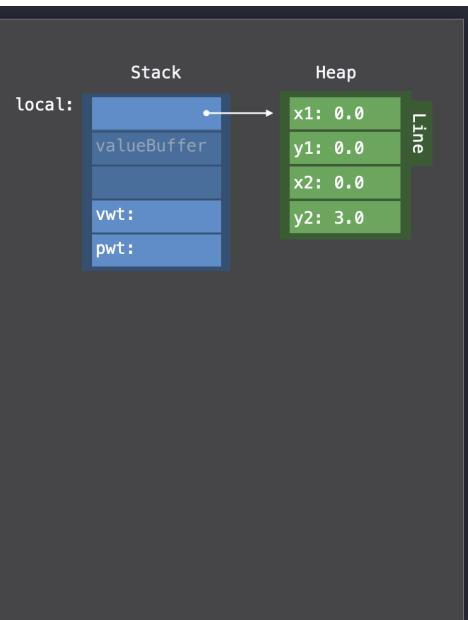
```

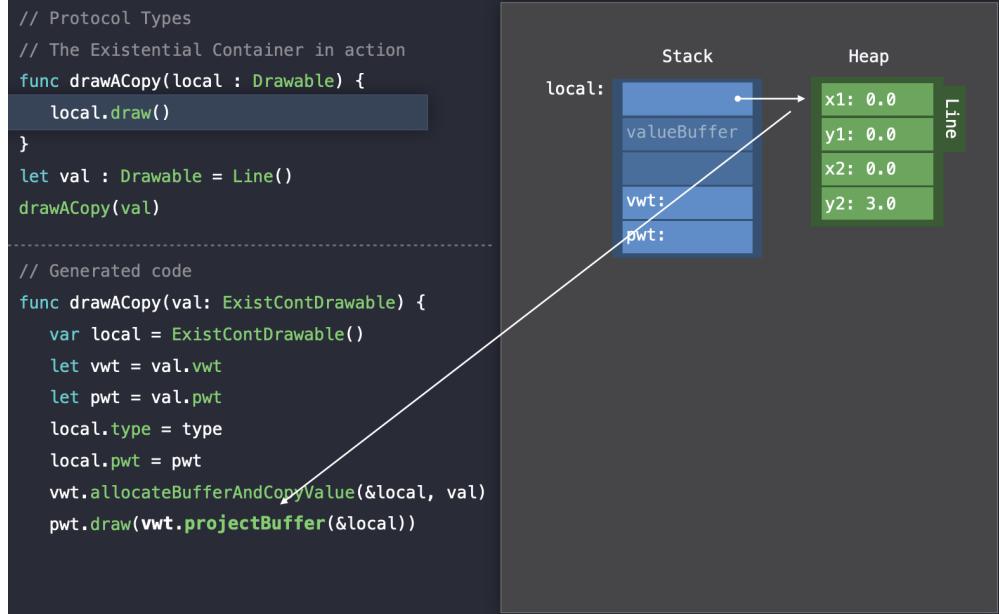
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
    local.draw()
}

let val : Drawable = Line()
drawACopy(val)

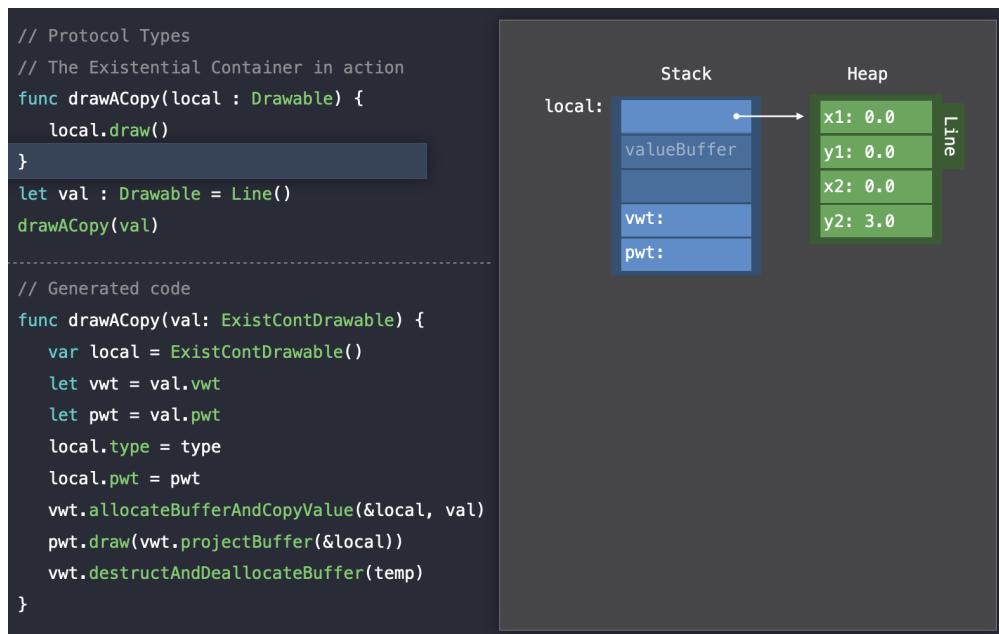
-----
// Generated code
func drawACopy(val: ExistContDrawable) {
    var local = ExistContDrawable()
    let vwt = val.vwt
    let pwt = val.pwt
    local.type = type
    local.pwt = pwt
    vwt.allocateBufferAndCopyValue(&local, val)
    pwt.draw(vwt.projectBuffer(&local))
}

```





- 할당 해제



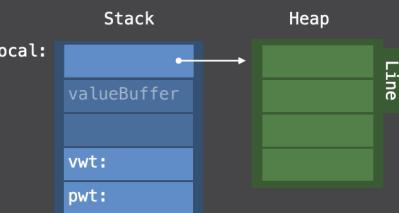
```

// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
    local.draw()
}

let val : Drawable = Line()
drawACopy(val)

-----
// 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)
}

```



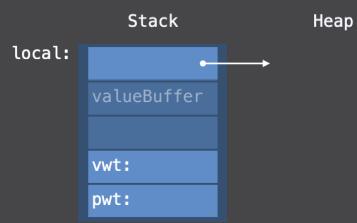
```

// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
    local.draw()
}

let val : Drawable = Line()
drawACopy(val)

-----
// 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)
}

```



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

let val : Drawable = Line()
drawACopy(val)

-----
// 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)
}
```

지역 변수가 복사되는 방식과 프로토콜 유형의 값에 대해 메서드 디스패치가 작동하는 방식을 다루어 보았으니 **Stored Property**에 대해서도 알아보자!

- **Protocol Type Stored Properties**

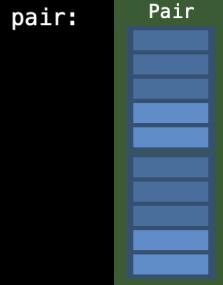
- ▼ 일반적인 경우

- 두 개의 Existential Container를 둘러싸는 struct의 inline에 저장

# 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



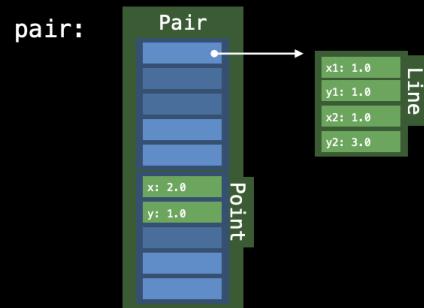
- 초기화

# 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

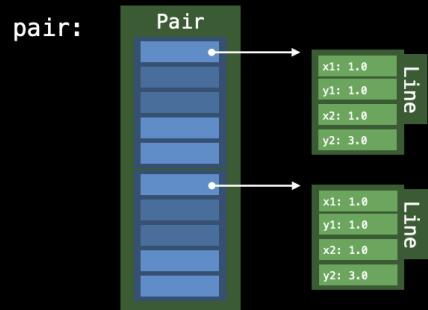


- 동적 다형성 지원

# 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



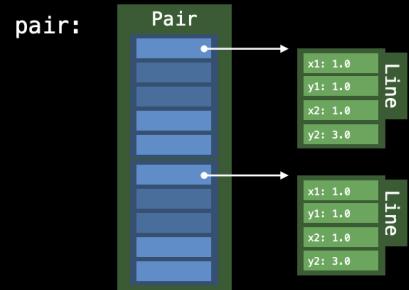
## ▼ Expensive Copies of Large Values

### ▼ 문제상황

- Large Value 생성

## Expensive Copies of Large Values

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



- 두 개의 heap 할당 발생

- Copy로 인하여 heap 할당에 많은 비용 발생

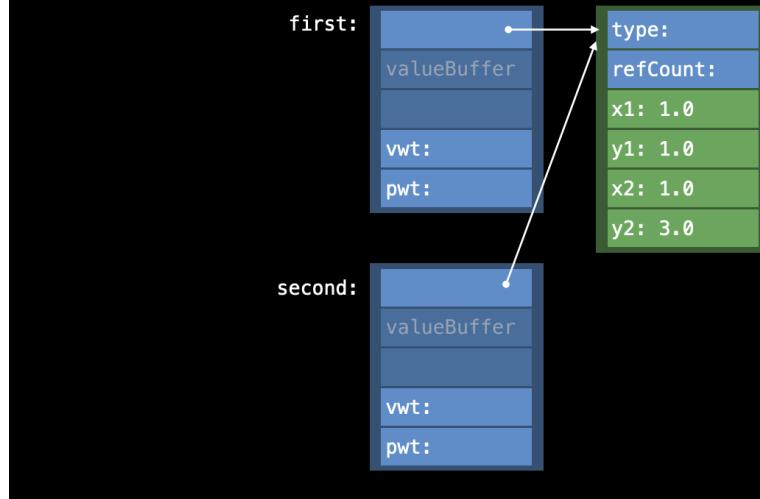
# Expensive Copies of Large Values



## ▼ 해결방안: Indirect Storage

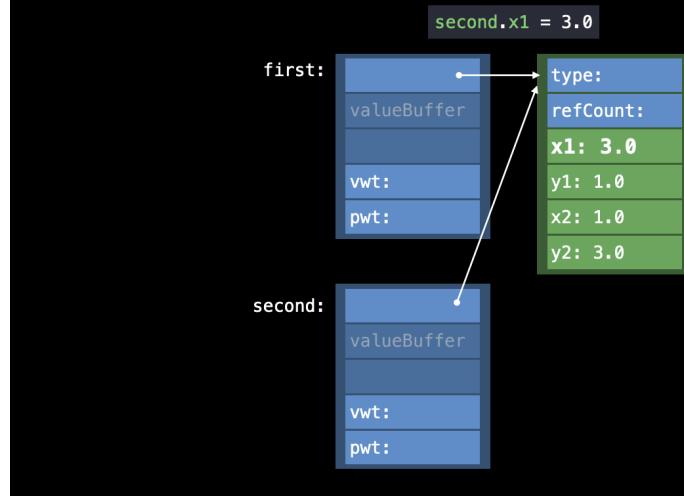
- 이를 해결하기 위해 Indirect Storage 적용
- reference는 1 word로 value buffer에 fit하므로, 같은 값을 참조하도록 변경
- 추가된 비용은 refCount 뿐이므로 비용 절감 효과
  - 비용: Heap 할당 → ref Count 증가
- 실제 existential container에서 넘어갈 때 생성된 heap이 아니라 LineStorage에 대한 레퍼런스만 저장

## References Fit in the Value Buffer



- 그러나, 상태 공유 문제 발생

## References Fit in the Value Buffer



**Expensive Copies of Large Values**에서 참조를 통해 비용을 절감하였으나, 우리가 원하는 **value semantics**를 구현하기 위해서는 어떤 조치를 취해야 할까?

▼ 해결방안: **Indirect Storage with Copy On Write(COW)**

- Line 구조체 내부에 직접 저장소를 구현하는 대신 LineStorage라는 클래스를 만들어서 이 저장소를 참조하도록 함

## 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 = ...
    }
}
```

## 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 = ...
    }
}
```

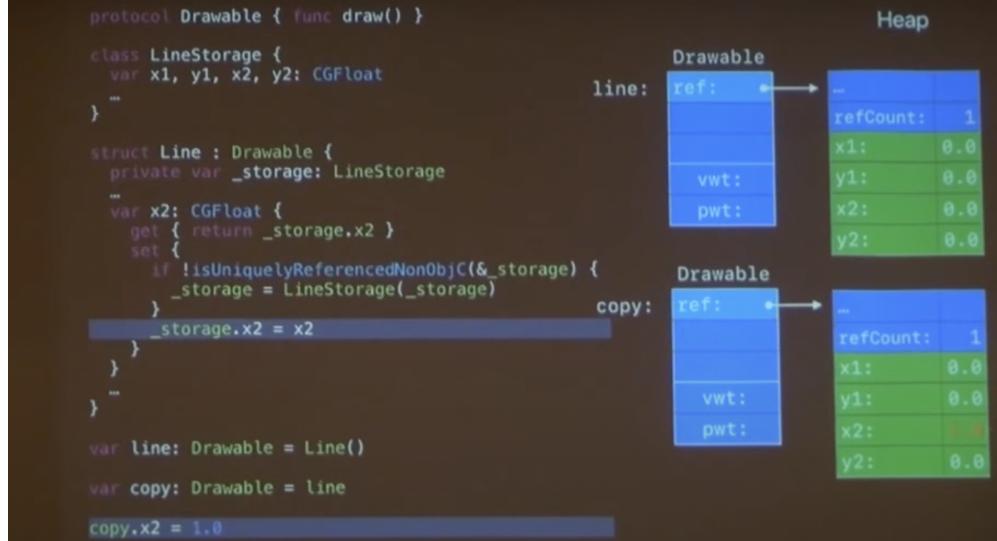
- 이로써, 값을 읽는 경우 해당 스토리지 내부의 값을 읽으면 되고, 값을 수정하려는 경우 먼저 참조 횟수를 확인한 다음, 1보다 큰 경우 라인 스토리지의 복사본을 만들어 이를 변경하도록 함

# Indirect Storage with Copy-on-Write

## Implement copy-on-write

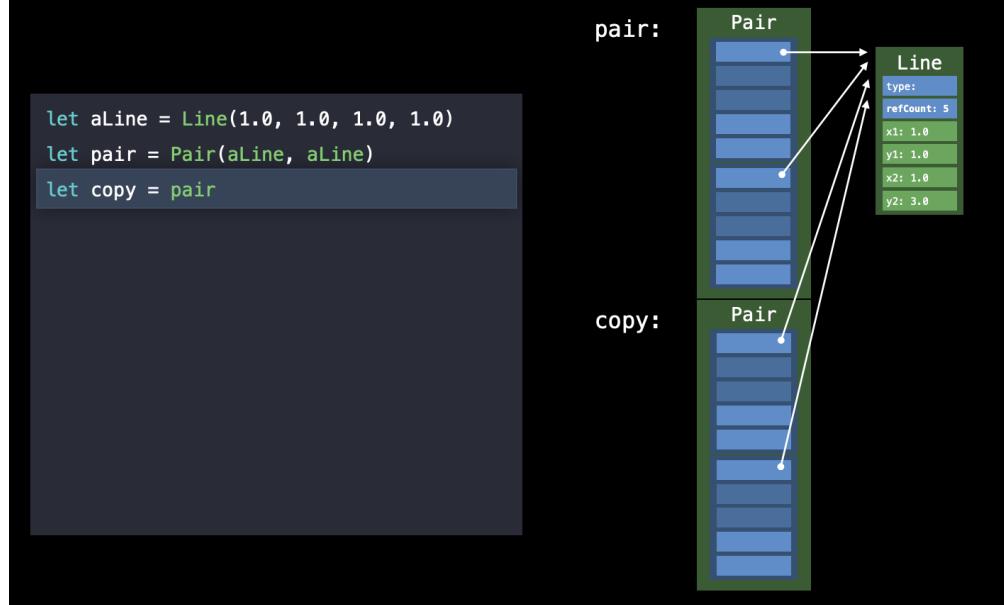
```
class LineStorage { var x1, y1, x2, y2: Double }
struct Line : Drawable {
    var storage : LineStorage
    init() { storage = LineStorage(Point(), Point()) }
    func draw() { ... }
    mutating func move() {
        if !isUniquelyReferencedNonObjC(&storage) {
            storage = LineStorage(storage)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

## Copy-on-Write



- 이 방법을 사용하면 참조만 복사되고, 참조 횟수 증가하므로, 힙 할당보다 훨씬 저렴하게 작동

# Copy Using Indirect Storage



## Protocol Type 중간 정리

1. Dynamic Polymorphism 제공
2. PWT, VWT, Existential Container를 통해 프로토콜과 함께 값 유형을 사용할 수 있으며, 프로토콜 유형의 배열 내부에 다양한 타입 제공 가능
3. Large Value일 경우 Heap Allocation 발생 → Indirect Storage & COW를 통해 해결 가능

- Generic

▼ Static Polymorphism(=Parametric Polymorphism) ( $\leftrightarrow$  Dynamic Polymorphism)

# 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)
```

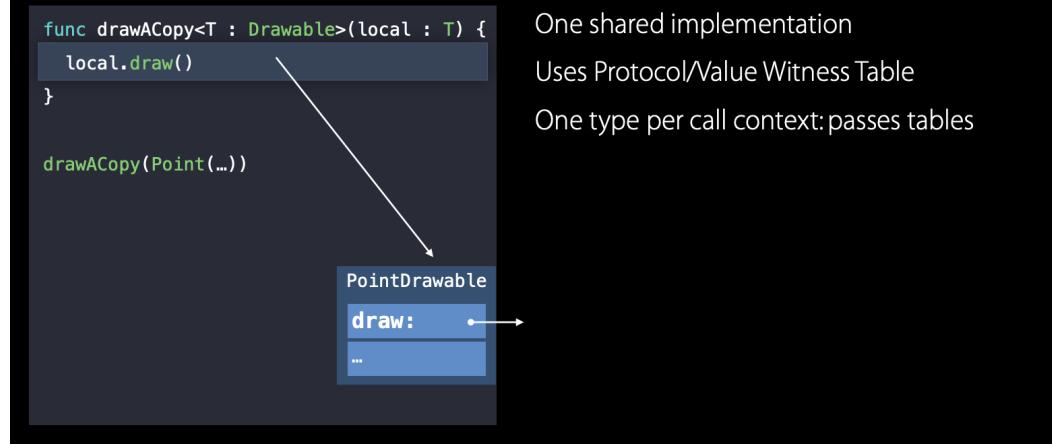
Static polymorphism

One type per call context

Type substituted down the call chain

- 호출 컨텍스트당 하나의 타입을 사용
  - 즉, 메서드가 일반 매개변수를 가지고 있을 때, 이 매개변수는 호출되는 컨텍스트에서 사용되는 타입으로 바인딩
  - 이는 곧, Parametric Polymorphism라고도 불리며, 매개변수를 따라 call chain 아래로 대체되는 것을 의미
  - 컴파일 시점에 호출하는 곳마다 타입이 정해져 있음
  - 런타임 시점에 변경 X
  - 특수화(specialization) 가능 의미
- ▼ 작동방식

# Implementation of Generic Methods



- Swift 내부에서는 VWT와 PWT를 사용하여 프로토콜 타입과 값 타입 모두에 대한 하나의 공유 구현(shared implementation)을 가짐
- 하지만, 호출 컨텍스트당 하나의 타입을 가지므로, **Existential Container** 존재 X
- 대신, 호출 사이트에서 사용되는 Point와 같은 타입의 VWT PWT을 모두 추가 인수로 전달
- 그리면 함수를 실행하는 동안, Swift는 VWT을 사용하여 변수에 필요한 버퍼를 heap에 할당하고 할당 소스에서 대상으로 복사
- 마찬가지로, 메서드를 실행할 때도 전달된 PWT을 사용하고, 테이블에서 고정 오프셋의 메서드를 찾아 구현부로 이동

**Existential Container가 없는데, Swift는 로컬 매개변수에 필요한 메모리를 어떻게 할당할까?**

# Storage of Local Variables

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

Value Buffer: currently 3 words  
Small values stored inline

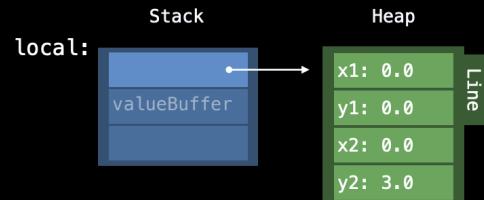


- Swift는 valueBuffer라는 스택에 메모리를 할당하여 로컬 변수에 필요한 메모리를 할당
- valueBuffer는 작은 메모리 크기의 값에 적합

# 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



- 큰 값은 다시 힙에 저장되며, 로컬 existential container 내부에 해당 메모리에 대한 포인터를 저장

## ▼ Specialization

- Static Polymorphism을 통해 컴파일러가 최적화 수행 가능

- 예를 들어, Point와 Line 클래스가 있고 Point를 인자로 받는 drawACopy 함수가 있다면, Swift는 호출 사이트에서 사용되는 타입별로 버전을 생성하여 해당 유형에 특정한 함수 버전을 생성

## Specialization of Generics

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

Static polymorphism: uses type at call-site

## Specialization of Generics

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

Static polymorphism: uses type at call-site  
Creates type-specific version of method

## 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

- 이를 통해 빠른 코드 실행이 가능
- 또한, 코드 크기를 줄일 수 있음
  - 사용할 수 없는 정적 타이핑 정보가 공격적인 컴파일러 최적화를 가능하게 하기 때문

## 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

## 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

## 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

- Specialization 가능 요건

# When Does Specialization Happen?

Infer type at call-site  
Definition must be available

```
main.swift
struct Point { ... }
let point = Point()
drawACopy(point)
```

- 해당 호출 사이트에서 사용되는 유형을 유추할 수 있어야 함
- 이를 위해서는 로컬 변수를 확인하고 초기화로 돌아가서 해당 유형을 확인
- 또한, 해당 함수를 Specialize하기 위해서는 사용 가능한 Generic 함수와 함께 해당 함수를 사용하는 모든 타입에 대한 정의가 필요

# Whole Module Optimization

Increases optimization opportunity

```
Module A
Point.swift
struct Point {
    func draw() {}
}

UsePoint.swift
let point = Point()
drawACopy(point)
```

- Whole Module Optimization을 통해 범위를 늘리는 것도 방법

이를 통해 PWT와 VWT를 사용하여 Unspecialization 코드가 작동하는 방식과 컴파일러가 제네릭 함수의 타입별 버전을 생성하는 코드를 Specialization 방법을 살펴봄



## 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

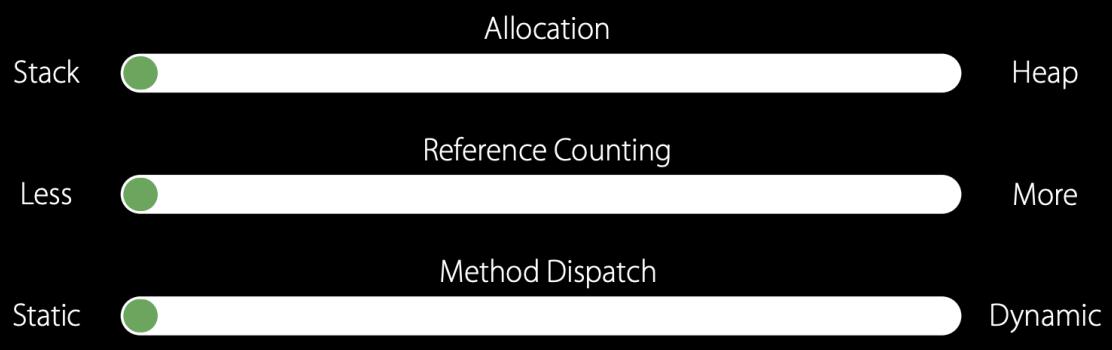
Use indirect storage to deal with large values

- 프로그래밍에서 엔터티를 추상화할 때, dynamic runtime type requirements가 적은 경우 → static type 검사가 활성화되도록 value type을 사용하는 것이 좋음
- struct
  - value type을 사용하면 상태 공유가 없어지며, 최적화 가능한 코드를 얻을 수 있음 → Value Semantics
- class
  - identity
  - OOP style polymorphism
- Generics
  - static polymorphism
- Protocol
  - 필요한 경우에는 Dynamic Polymorphism을 위해 프로토콜 타입과 값을 결합하여 사용 가능

### ▼ Performance 비교

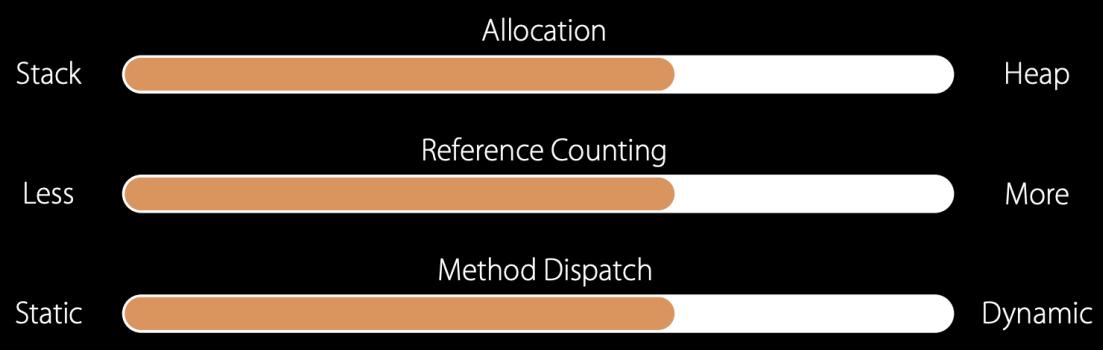
## Dimensions of Performance

### Struct



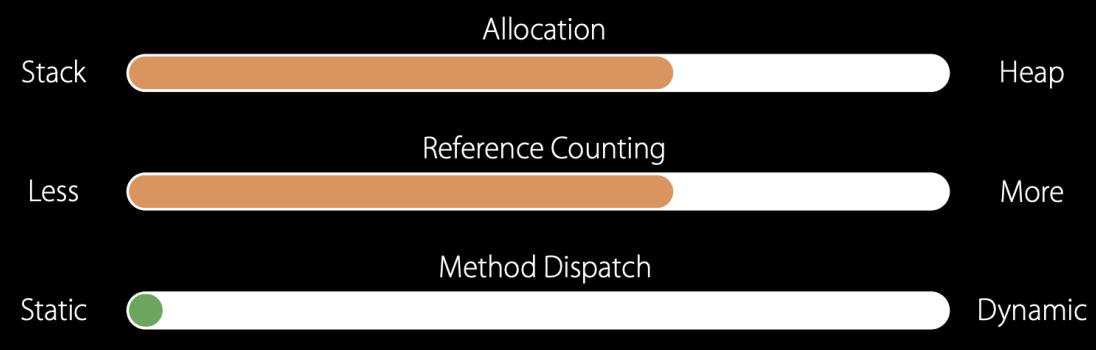
## Dimensions of Performance

### Class



## Dimensions of Performance

### Final Class



## Protocol Type—Small Value



Fits in Value Buffer: no heap allocation

No reference counting

Dynamic dispatch through Protocol Witness Table

## Protocol Type—Large Value

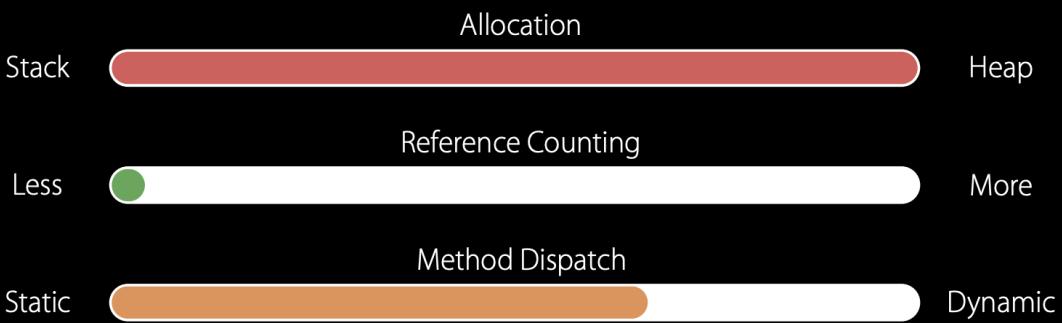


Heap allocation

Reference counting if value contains references

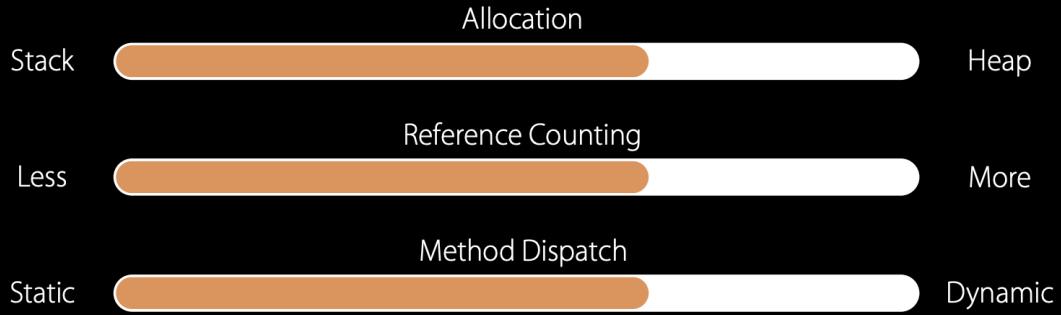
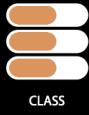
## Protocol Type—Large Value

Expensive heap allocation on copying



## Protocol Type—Indirect Storage

Trade off reference counting for heap allocation



## Specialized Generics—Struct Type



Performance characteristics like struct types

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

- struct를 포함한 Specialized Generics

- struct 타입을 사용하는 것과 동일한 성능
    - 기본적으로 구조체 측면에서 이 함수를 작성한 것과 같기 때문
  - struct 타입의 값을 복사할 때 heap 할당 필요 X
  - struct에 참조가 포함되어 있지 않으면 ref count X
  - 컴파일러 최적화를 가능하게 하고 런타임을 줄이는 static method dispatch

## Specialized Generics—Class Type



Performance characteristics like class types

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

- class 타입을 사용하면 class와 유사한 특성을 가짐
  - 인스턴스 생성시 heap 할당
  - 값 전달을 위한 reference counting
  - V-Table을 통한 Dynamic method dispatch

## Unspecialized Generics—Small Value



No heap allocation: value fits in Value Buffer

No reference counting

Dynamic dispatch through Protocol Witness Table

- small value을 포함하는 Unspecialized Generics
  - small value이 스택에 할당된 valueBuffer에 맞기 때문에 로컬 변수에 필요한 heap 할당 X
  - 값에 참조가 포함되어 있지 않으면 ref count X
  - 그러나 PWT을 사용하여 모든 임시적 호출 사이트에서 하나의 구현을 공유

# Unspecialized Generics—Large Value



Heap allocation (use indirect storage as a workaround)

Reference counting if value contains references

Dynamic dispatch through Protocol Witness Table

- large value을 포함하는 Unspecialized Generics

- large value는 스택에 할당된 valueBuffer에 맞지 않기 때문에 로컬 변수에 필요한 heap 할당
  - Indirect Storage을 통해 해결 가능
- large value에 참조가 포함되어 있으면 ref count 존재
- Dynamic dispatch
  - 즉, 코드 전체에서 하나의 일반 구현을 공유



## 참고 자료

- Understanding Swift Performance - Apple Developer

### Understanding Swift Performance - WWDC16 - Videos - Apple Developer

In this advanced session, find out how structs, classes, protocols, and generics are implemented in Swift. Learn about their relative...

apple <https://developer.apple.com/videos/play/wwdc2016/416/>

Choosing the Right Abstraction Mechanism



Modeling      Performance

- 스위프트 성능 이해하기 - 유용하

### 스위프트 성능 이해하기 - 유용하

발표자료 <http://www.slideshare.net/YongHaYoo/ss-63881606>

youtube <https://www.youtube.com/watch?v=z1Gf6EosaUQ>

