Swift Memory Layout

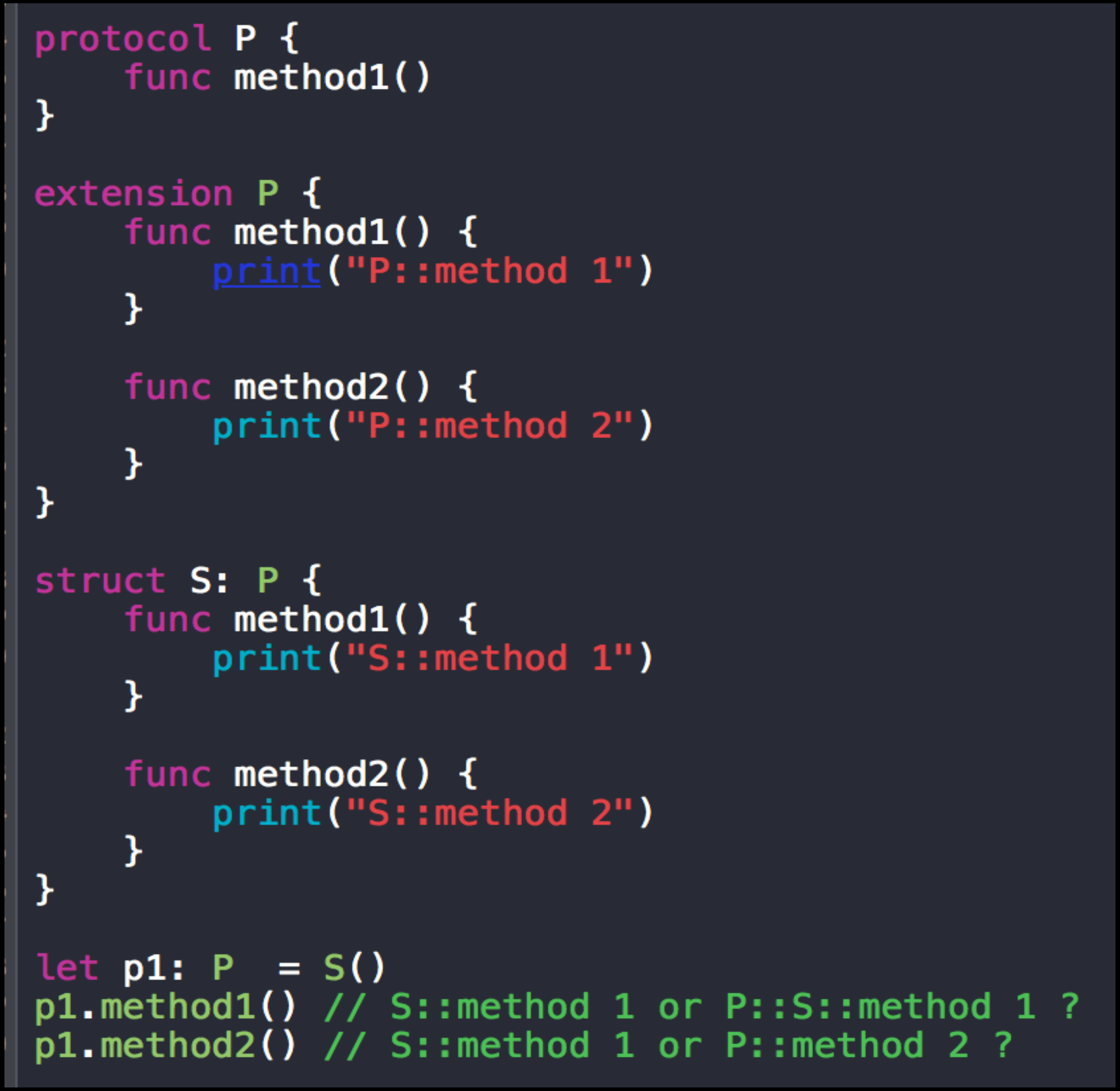
Fanglin Liu (fanglliu@cisco.com)

Cisco Spark for iOS Software Engineer

Why should we understand the implementation ?

- to understand the performance
  - C++ vs Swift vs Objective-C
- to avoid remembering the complex rules
  - C++ protocol extension method dispatch
- To write safer code
  - Prefer value type to reference type

Why should we understand the implementation ?



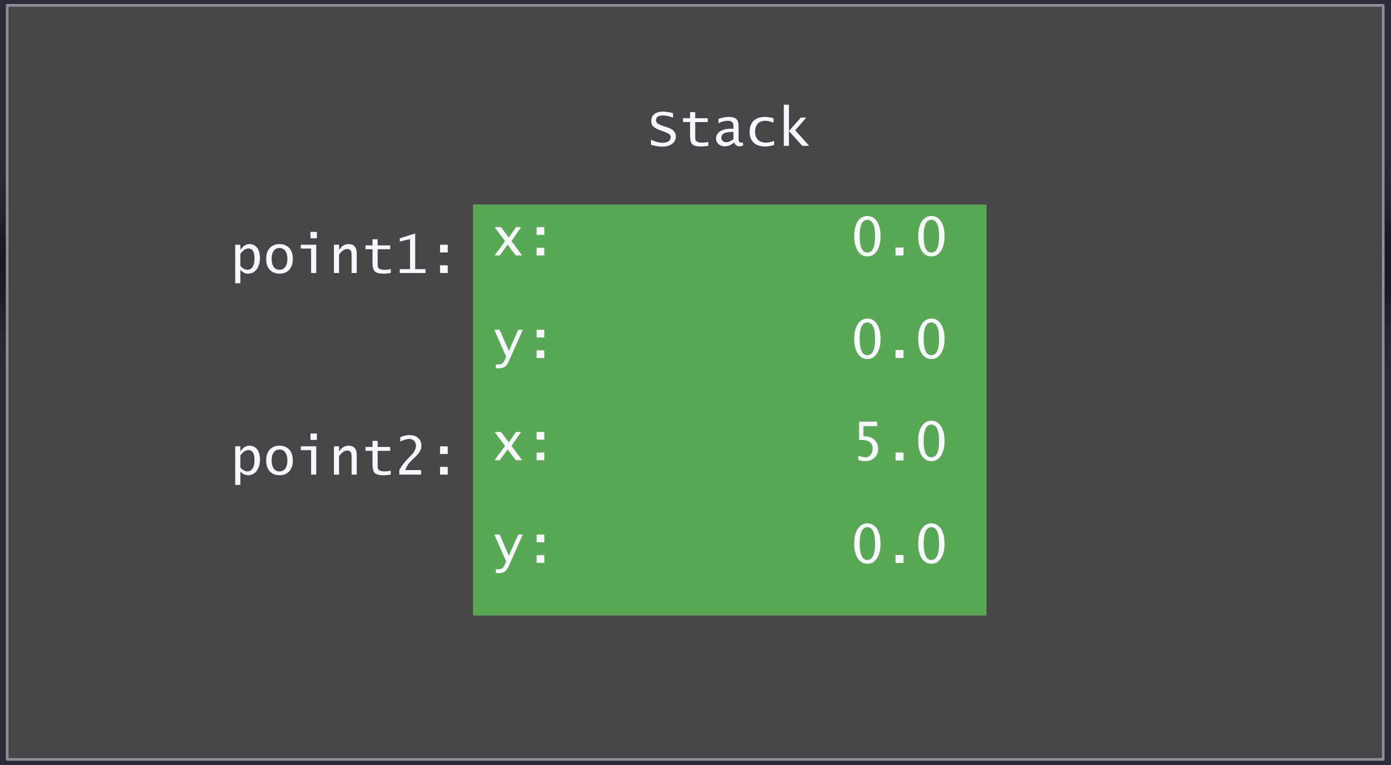
```
protocol P {
    func method1()
}

extension P {
    func method1() {
        print("P::method 1")
    }

    func method2() {
        print("P::method 2")
    }
}

struct S: P {
    func method1() {
        print("S::method 1")
    }

    func method2() {
        print("S::method 2")
    }
}

let p1: P  = S()
p1.method1() // S::method 1 or P::S::method 1 ?
p1.method2() // S::method 1 or P::method 2 ?
```

Program = data + method

Agenda

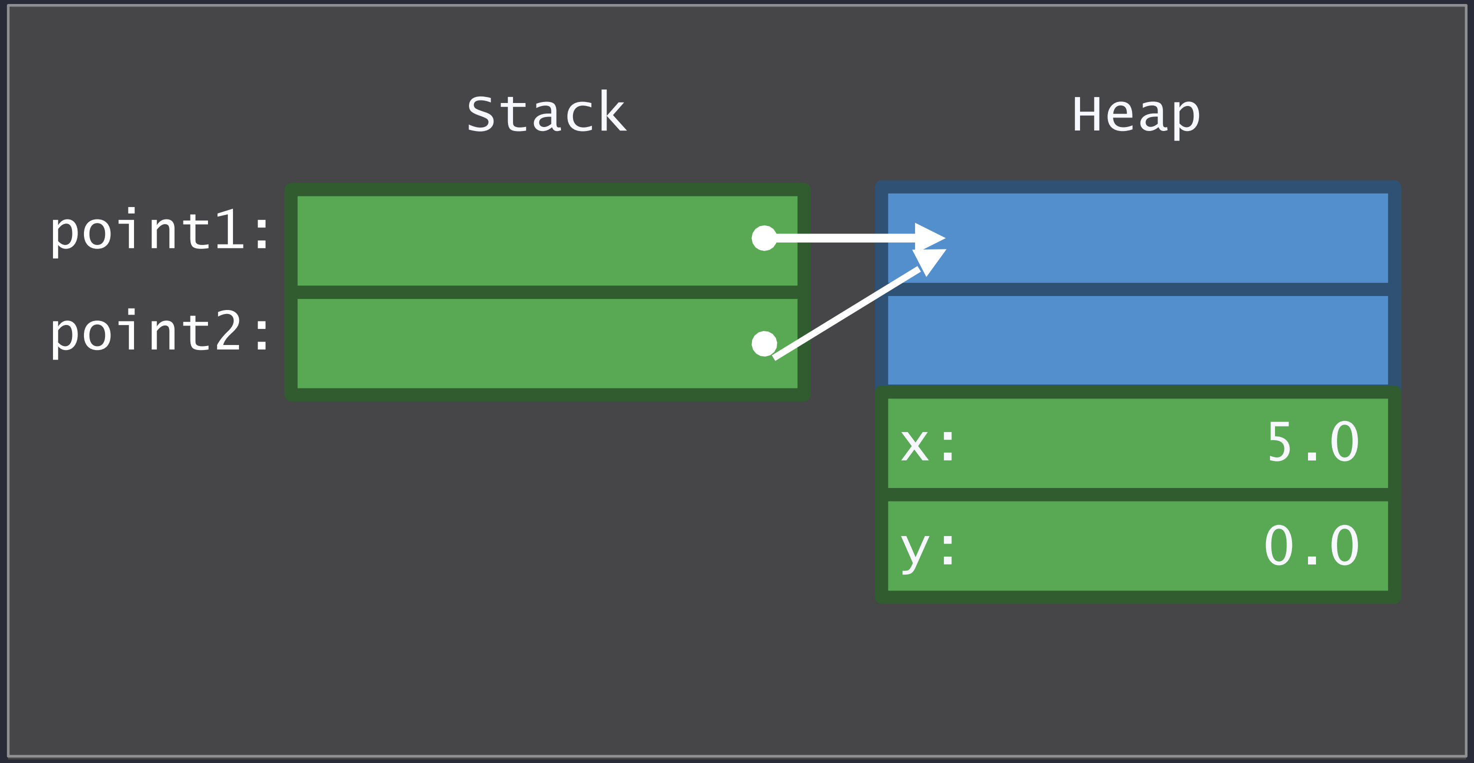
- Value Type vs Reference Type
    - Memory Allocation
    - Method dispatch
- Protocol types
- Generic code

Memory Allocation



Stack

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



Stack

Heap

point1:

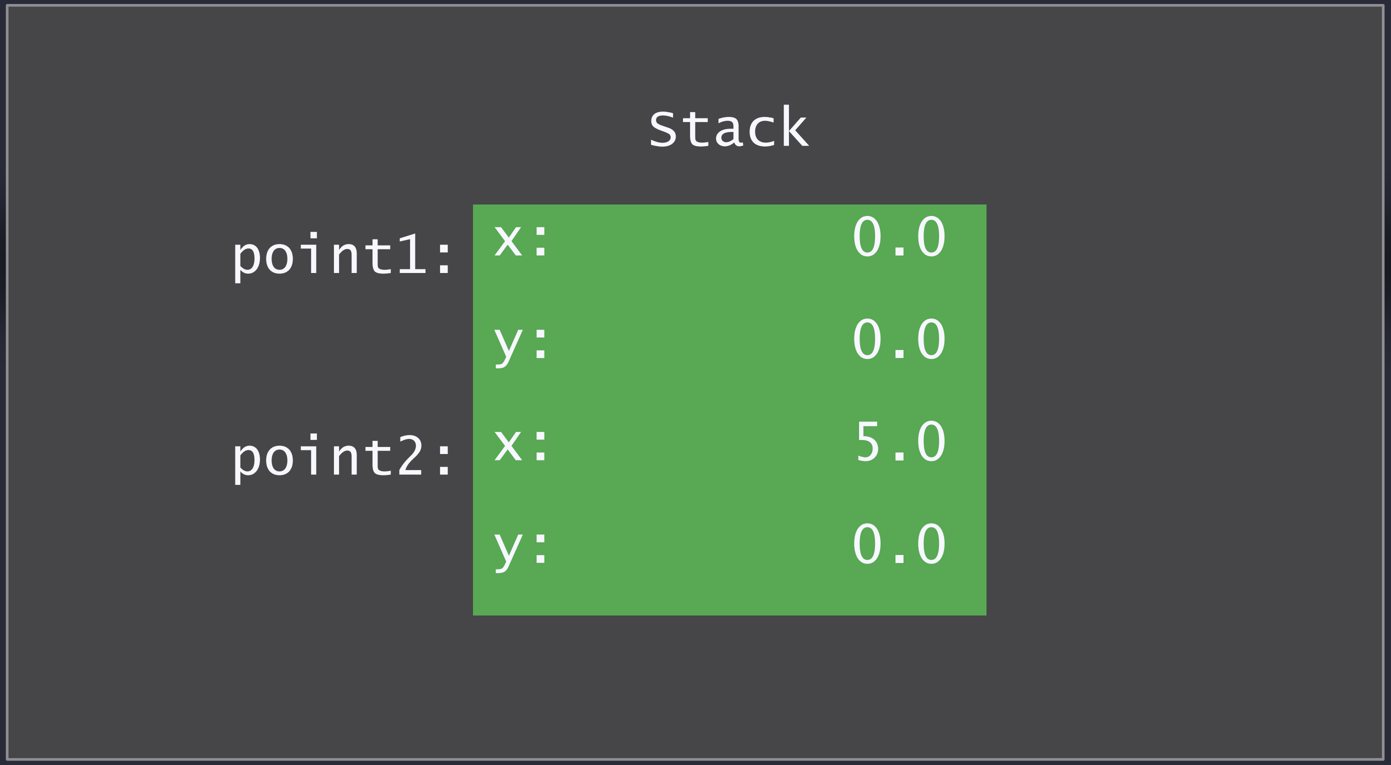
point2:

x: 5.0

y: 0.0

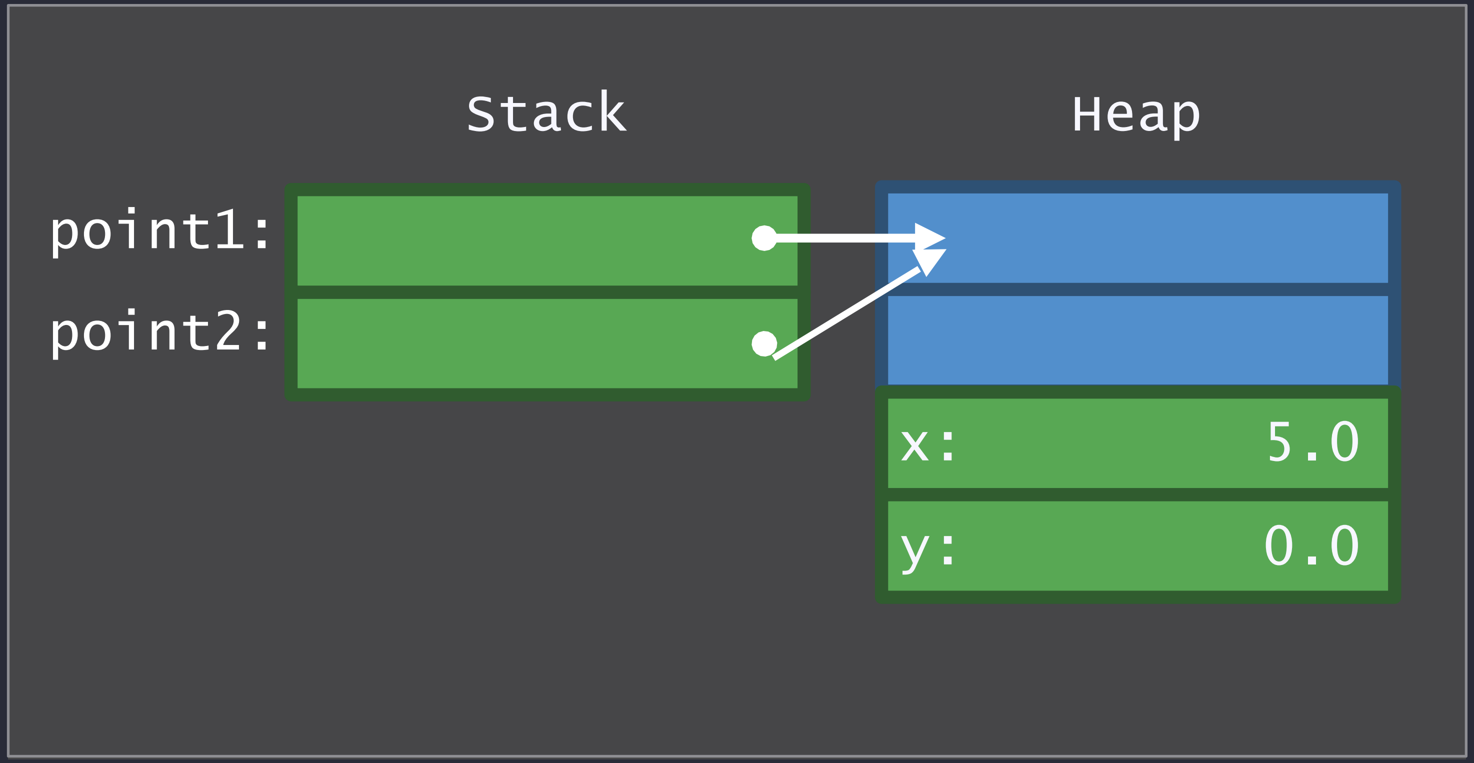
```
struct Point {
    var x, y: Double
    func draw() { … }
}

func foo(  ) {
    let point1 = Point(x: 0, y: 0)
    var point2 = point1
    point2.x = 5
}
```

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

func foo(  ) {
    let point1 = Point(x: 0, y: 0)
    var point2 = point1
    point2.x = 5
}
```



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

func foo(  ) {
    let point1 = Point(x: 0, y: 0)
    var point2 = point1
    point2.x = 5
}
```

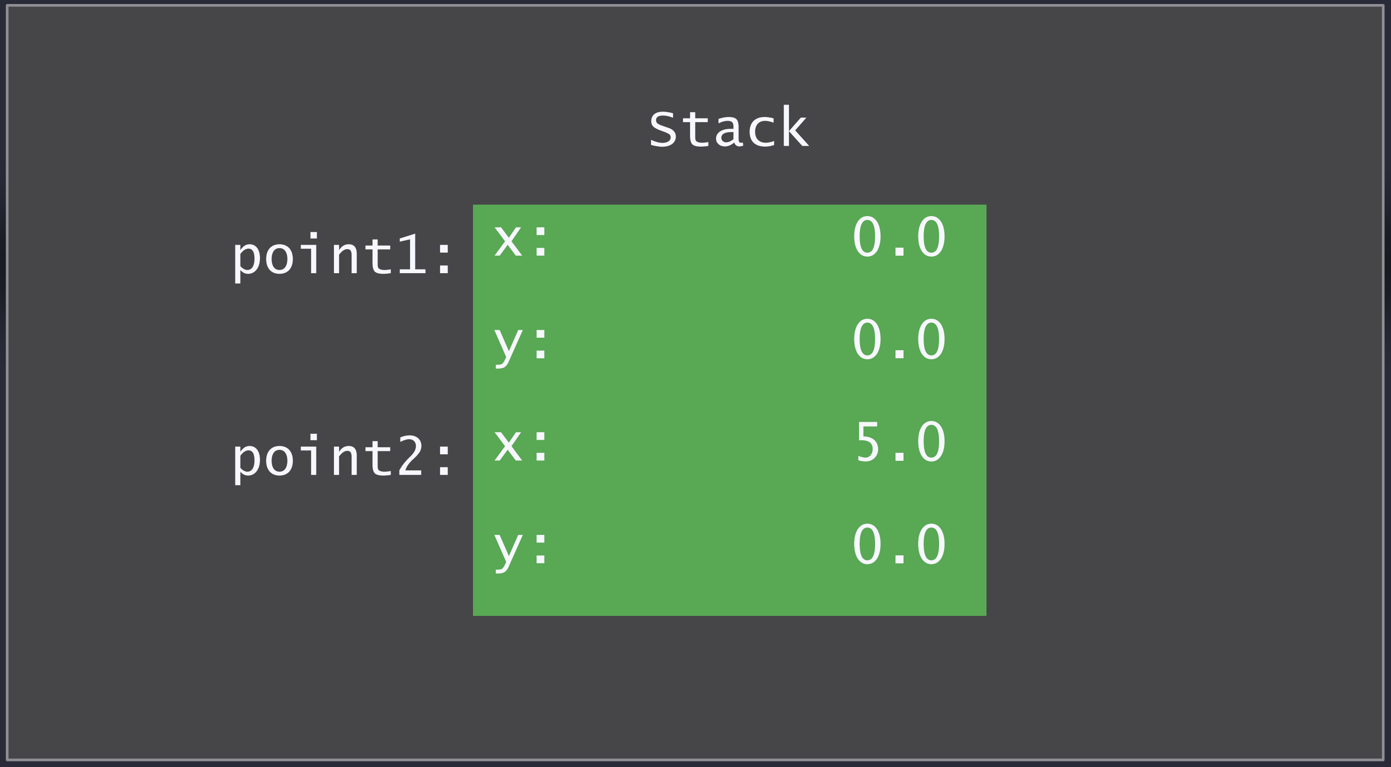
```
Let point1 = Point(x: 0, y: 0)

==== C++ programing language======

Point point1
Point1 = Point(0, 0)
Point point2 = point1
```



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

func foo(  ) {
    let point1 = Point(x: 0, y: 0)
    var point2 = point1
    point2.x = 5
}
```

```
Let point1 = Point(x: 0, y: 0)

==== C++ programing language======

Point * point1
Point1 = new Point(0, 0)
Point * point2 = point1
```

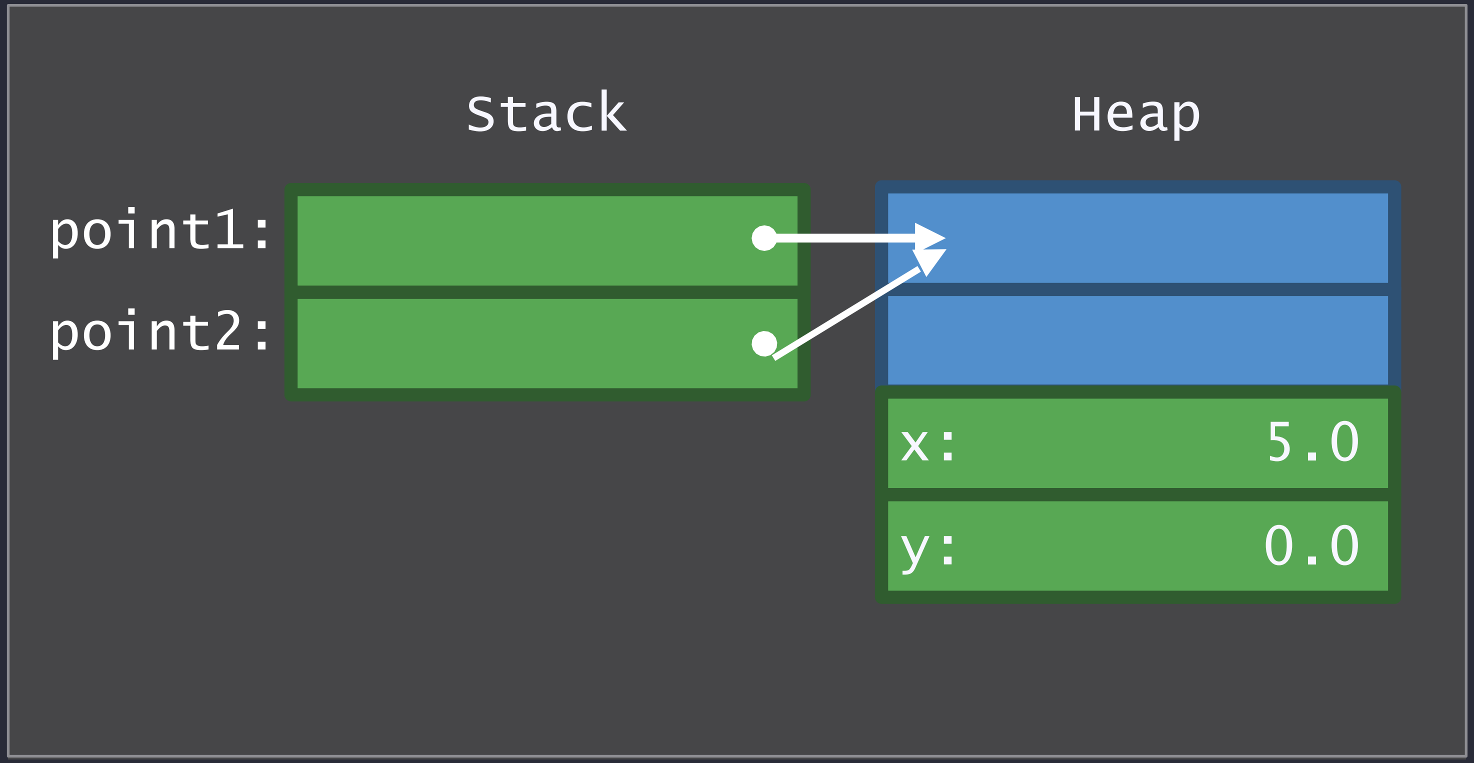
Allocation



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

func foo(  ) {
    let point1 = Point(x: 0, y: 0)
    var point2 = point1
    point2.x = 5
}
```

Stack

Decrement stack pointer to allocate Increment stack pointer to deallocate



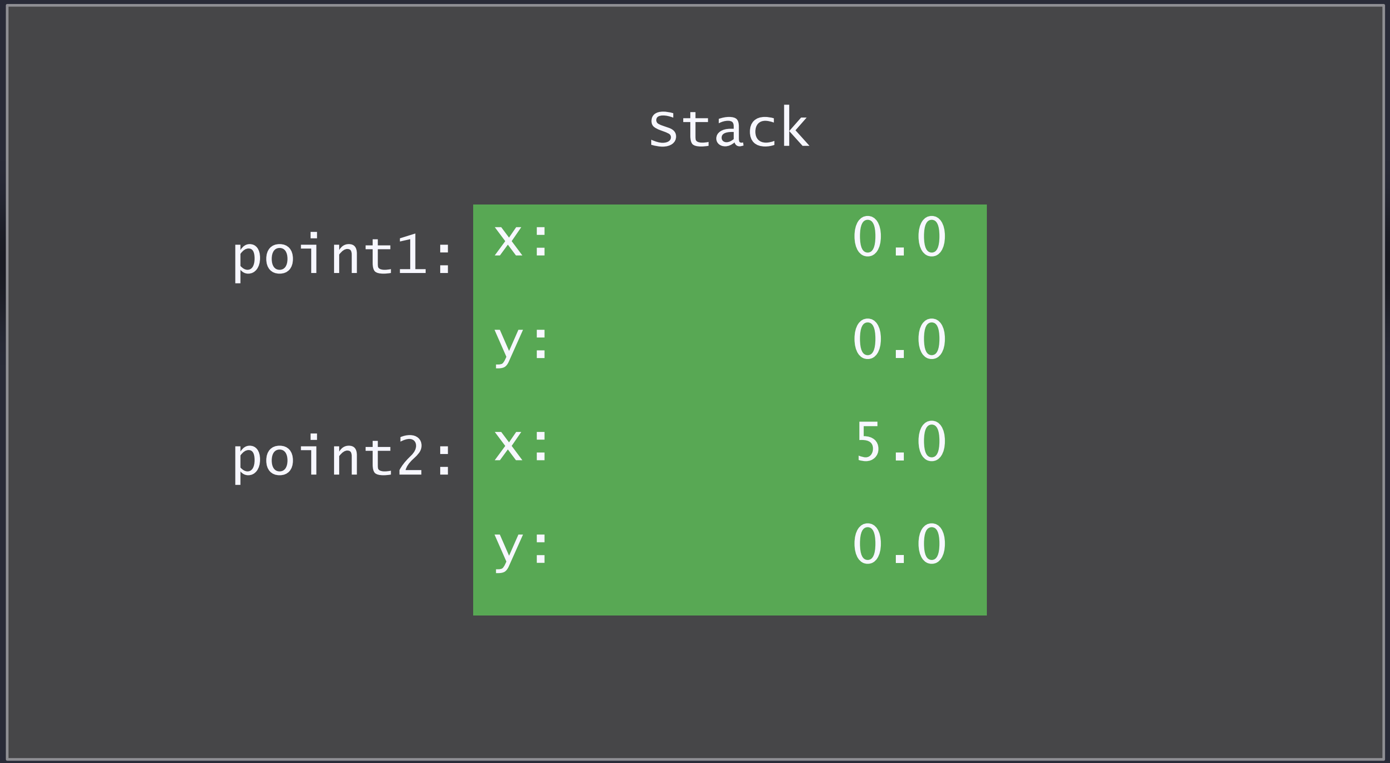
```
class Point {
    var x, y: Double
    func draw() { … }
}

func foo(  ) {
    let point1 = Point(x: 0, y: 0)
    var point2 = point1
    point2.x = 5
}
```

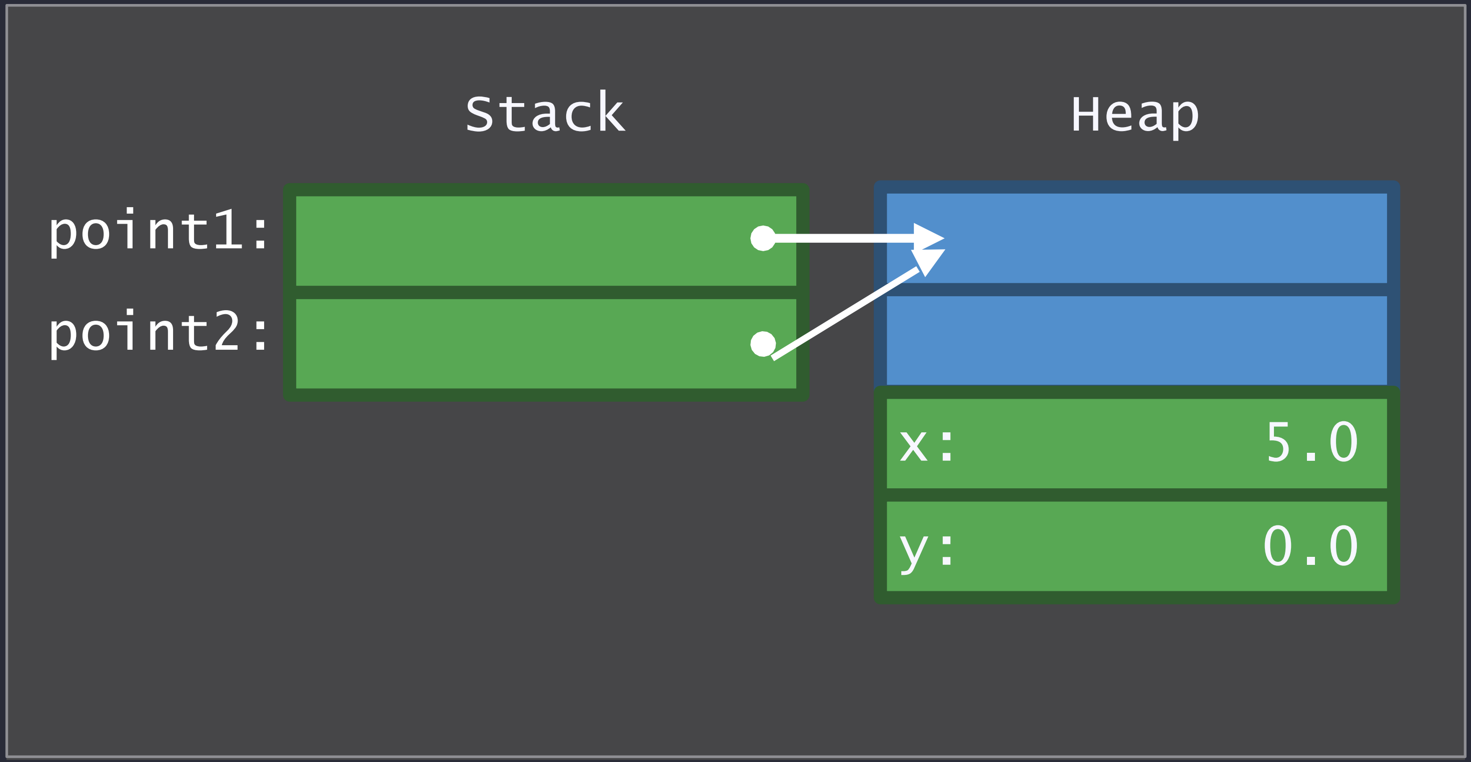
Heap:

Advanced data structure

Search for unused block of memory to allocate

Reinsert block of memory to deallocate Thread safety overhead





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

func foo( ) {
    let point1 = Point(x: 0, y: 0)
    var point2 = point1
    point2.x = 5
}
```

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

func foo( ) {
    let point1 = Point(x: 0, y: 0)
    var point2 = point1
    point2.x = 5
}
```

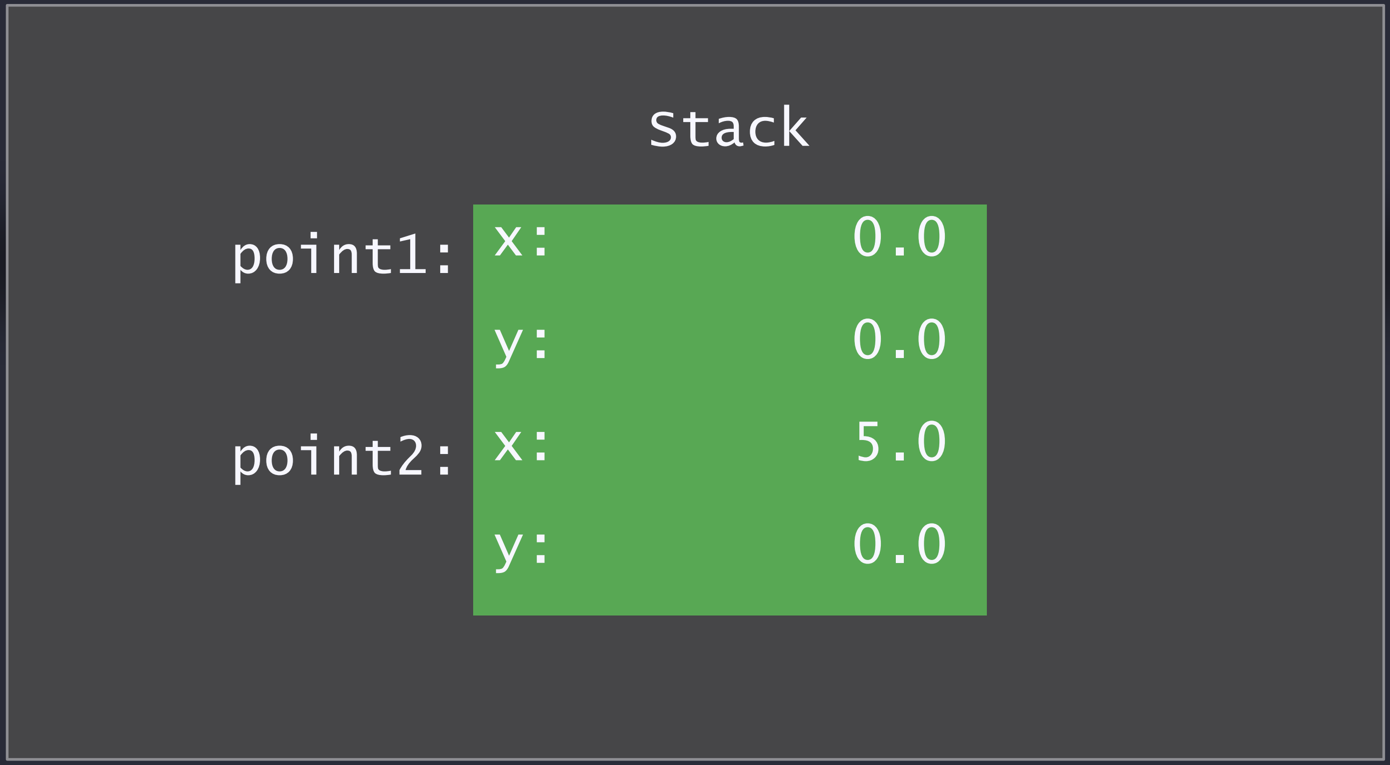
Reference count

NO

YES

Indirection

Thread safety overhead

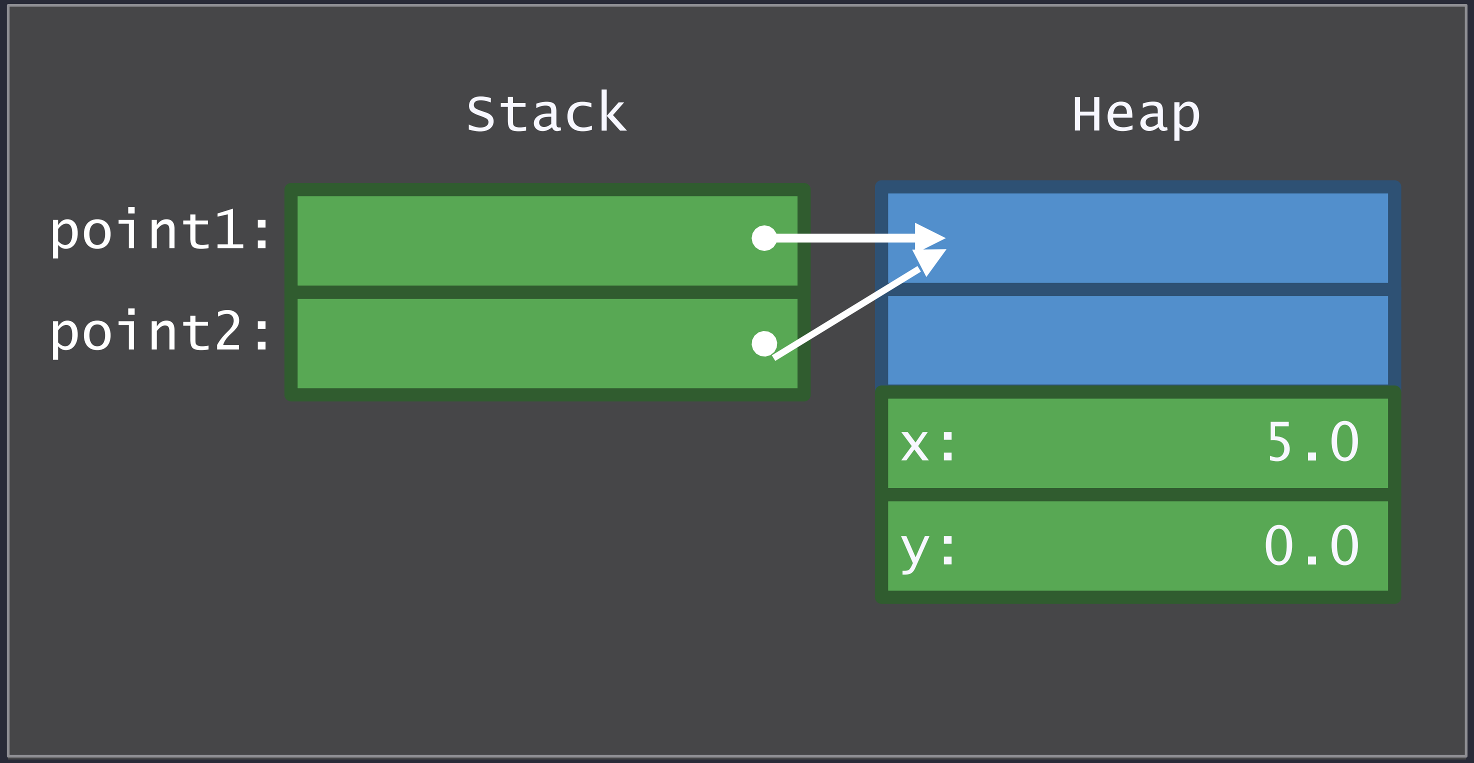


Stack

point1: x: 0.0
y: 0.0

point2: x: 5.0
y: 0.0

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

func foo(  ) {
    let point1 = Point(x: 0, y: 0)
    var point2 = point1
    point2.x = 5
}
```



Stack

Heap

point1:

point2:

x: 5.0

y: 0.0

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

func foo(  ) {
    let point1 = Point(x: 0, y: 0)
    var point2 = point1
    point2.x = 5
}
```

Multi-thread safety

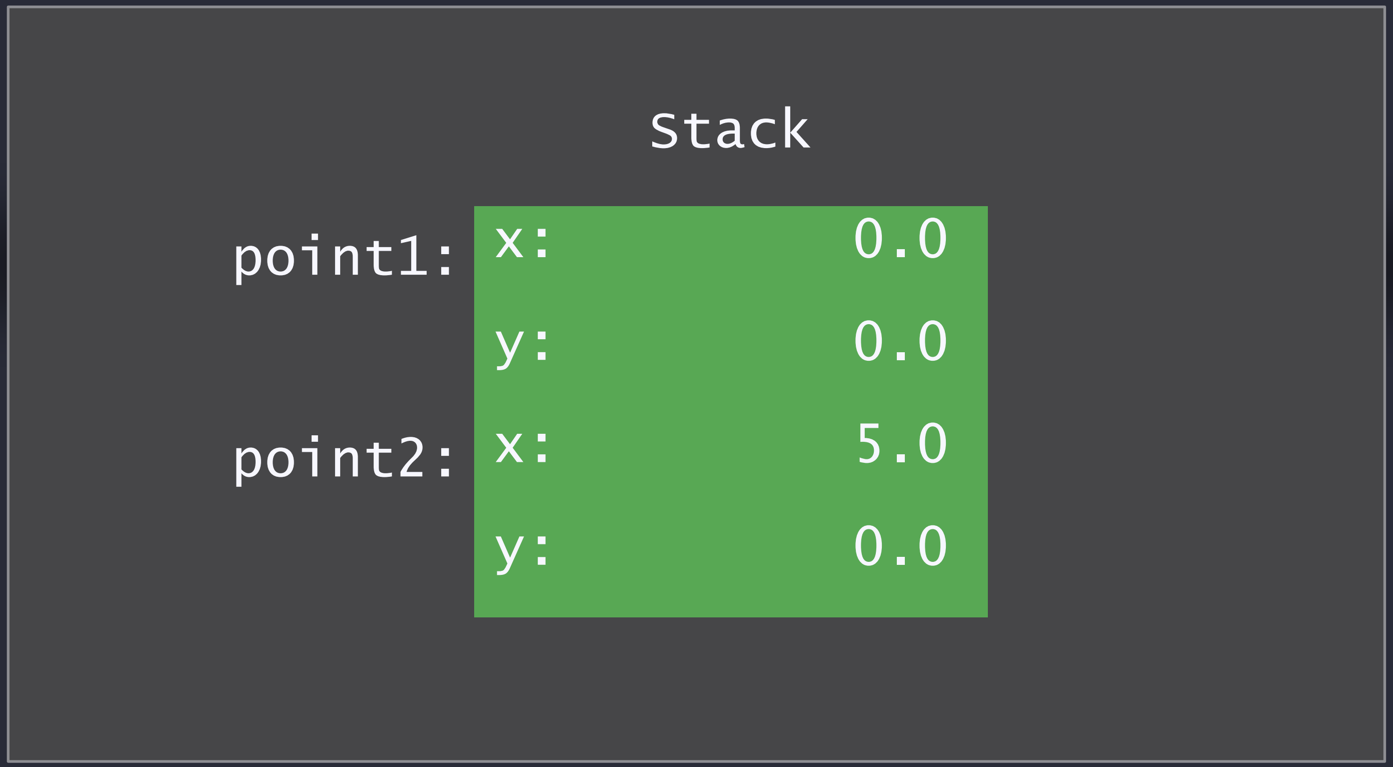
Parameters are always copied

Safe to change

Parameters are never copied

Unsafe to change

Problem with large value types



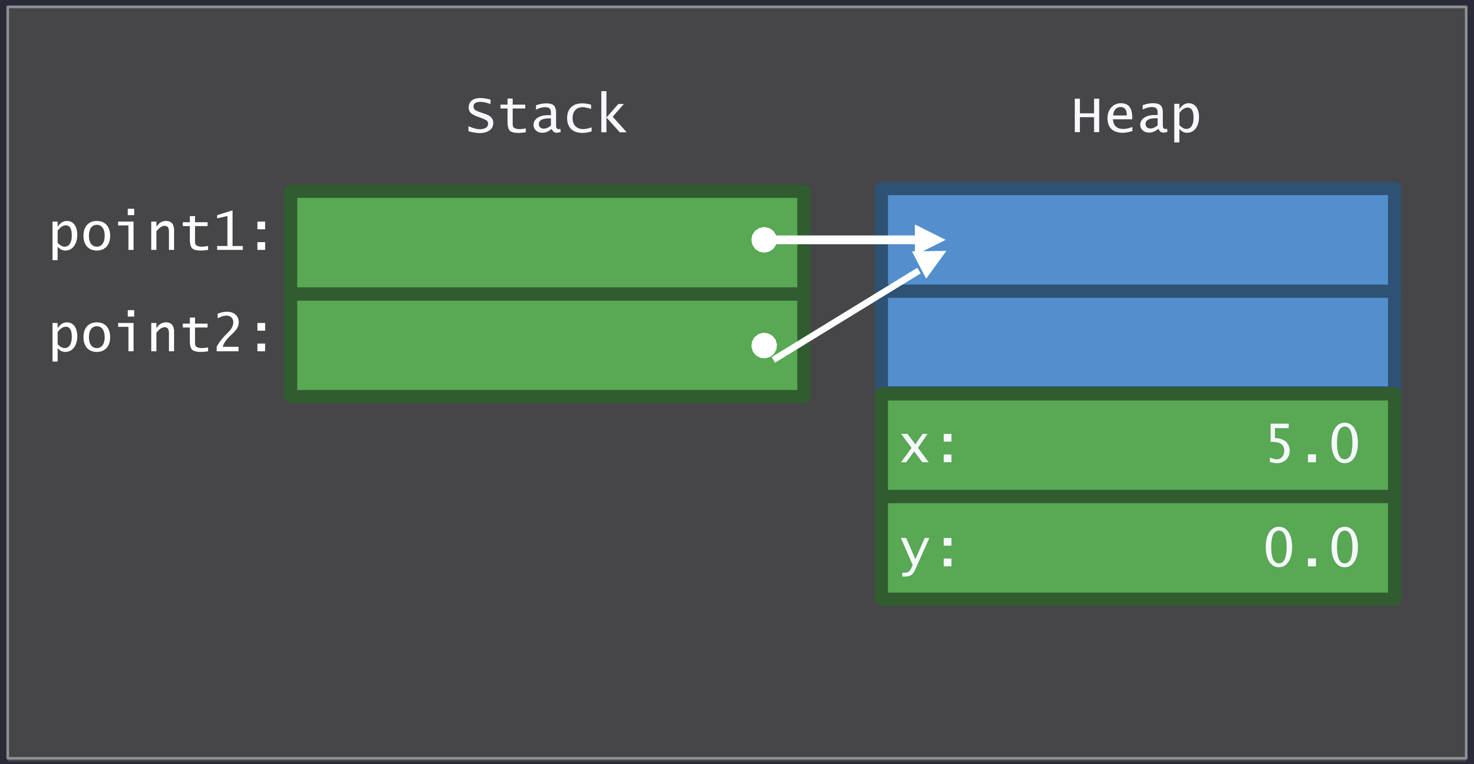
Stack

point1: x: 0.0

y: 0.0

point2: x: 5.0

y: 0.0



Stack

Heap

point1:

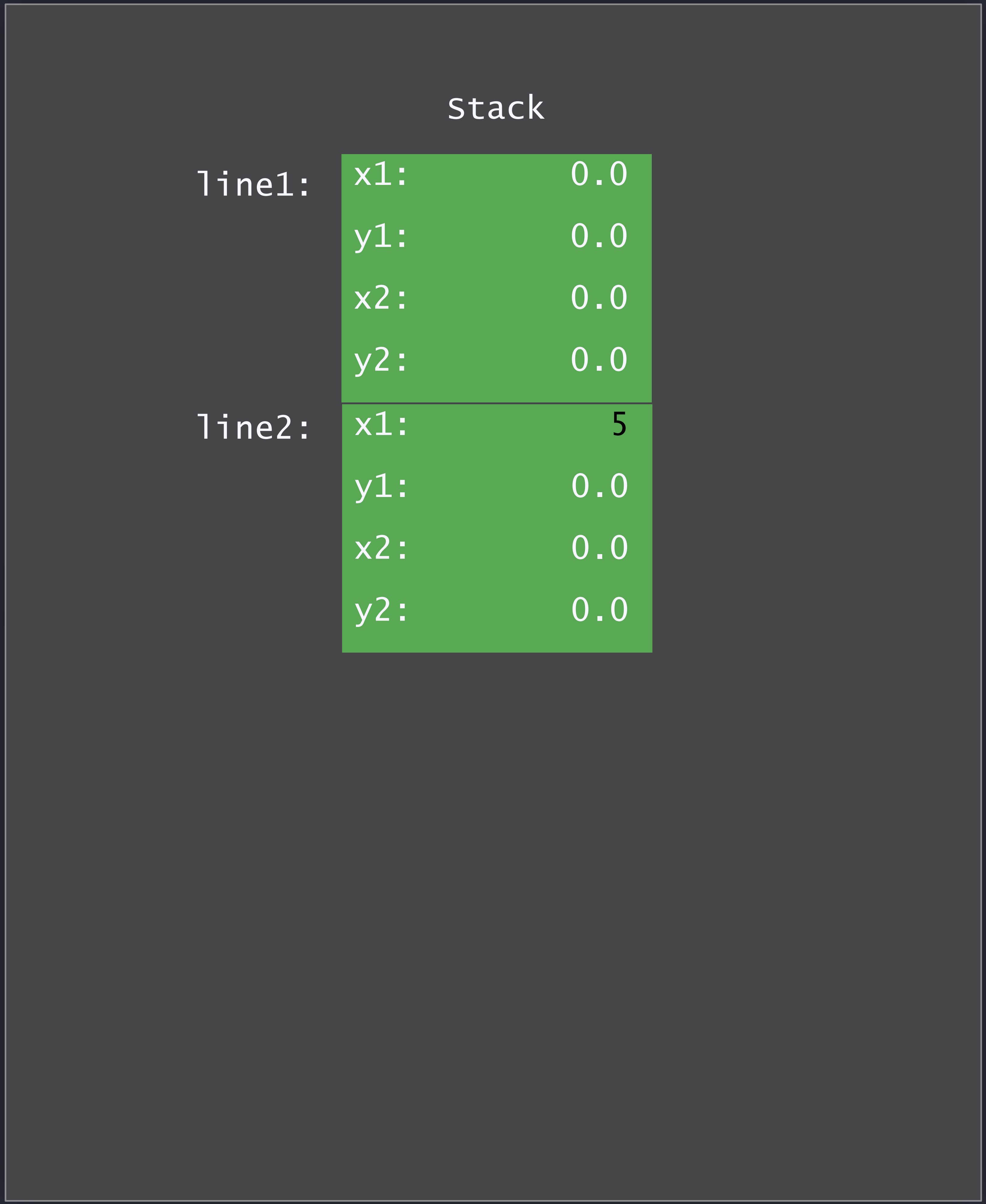
point2:

x: 5.0

y: 0.0

Problem with large value type

```
struct Line {
    var x1, y1, x2, y2: Double
    func draw() { … }
}

let line1  = Line(x1: 0, y1: 0, x2: 0, y2: 0)
let line2  = point1
line2.x1  = 5
// use `line1`
// use `line2`
```

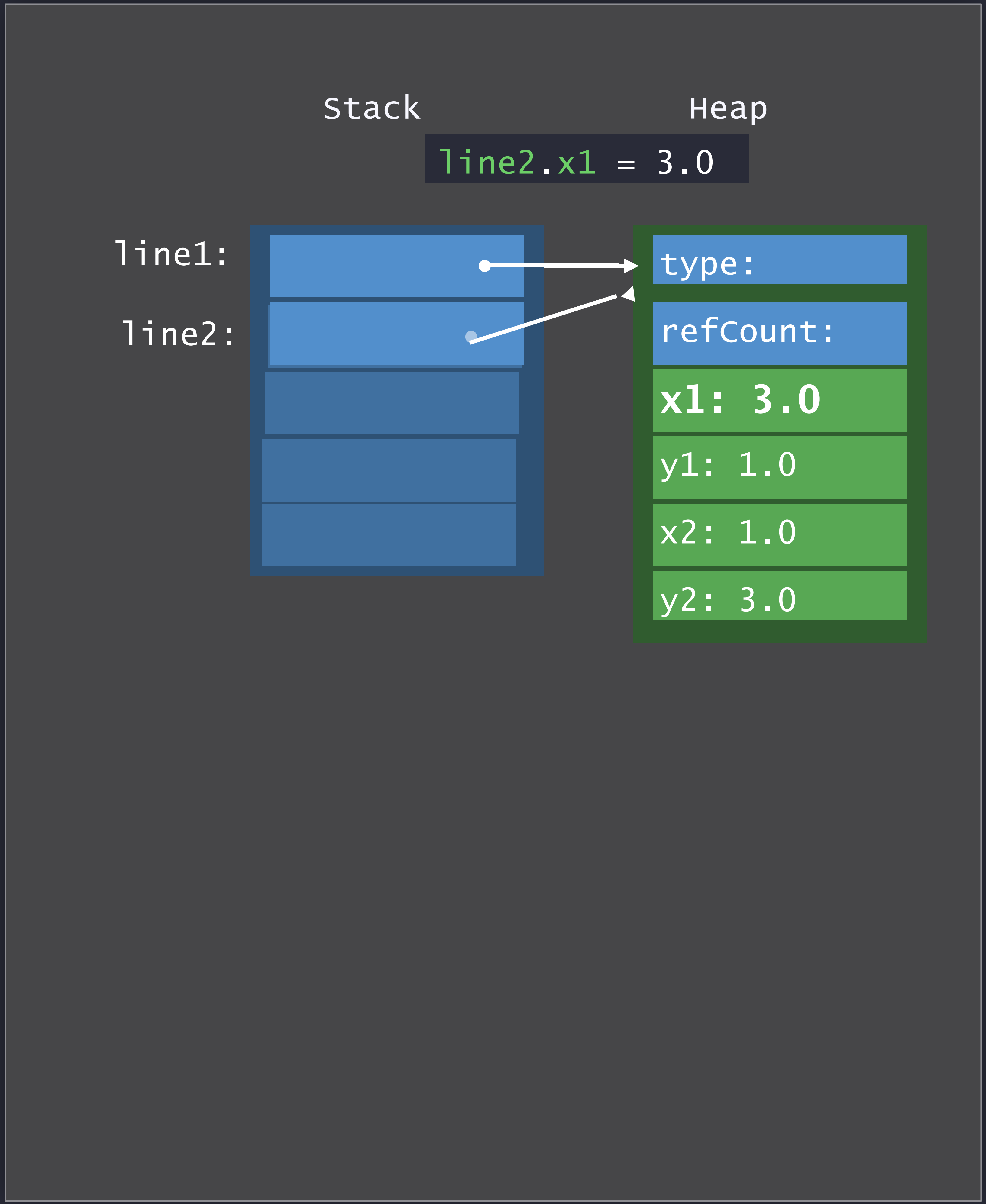


Stack

line1:	x1:	0.0
	y1:	0.0
	x2:	0.0
	y2:	0.0
line2:	x1:	5
	y1:	0.0
	x2:	0.0
	y2:	0.0

Problem with large value type

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

let line1  = Line(x1: 0, y1: 0, x2: 0, y2: 0)
let line2  = line1
line2.x1 = 3.0
// use `line1`
// use `line2`
```



Stack

Heap

line2.x1 = 3.0

line1:

line2:

type:

refCount:

x1: 3.0

y1: 1.0

x2: 1.0

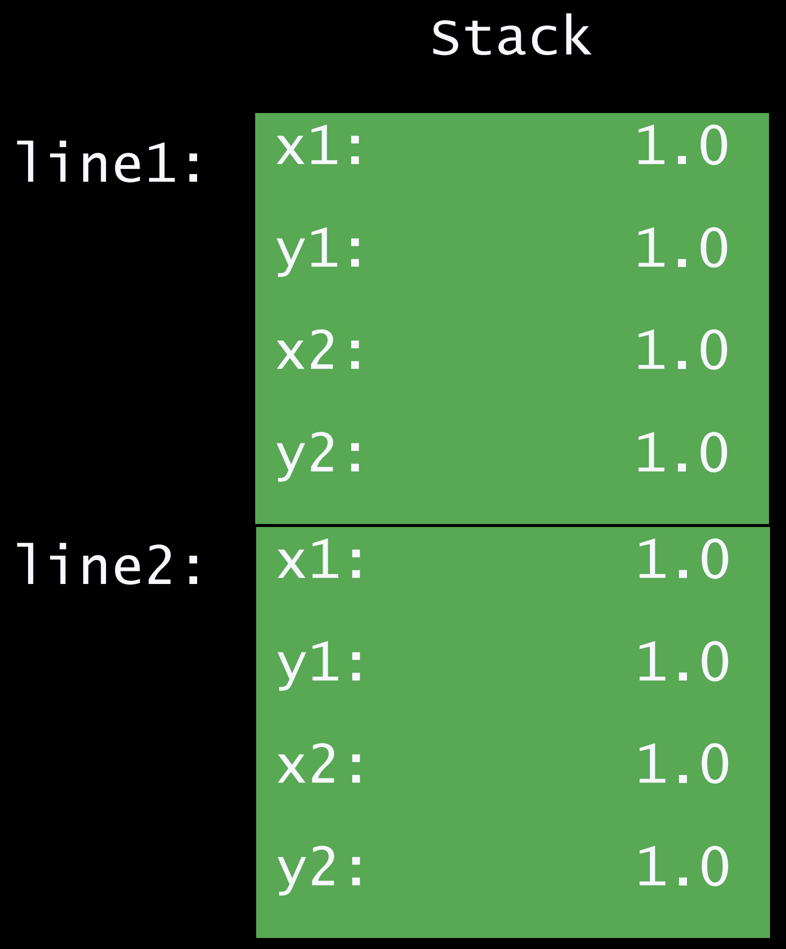
y2: 3.0

Copy-on-Write

Value type: Safe

Reference Type : Minimum memory usage

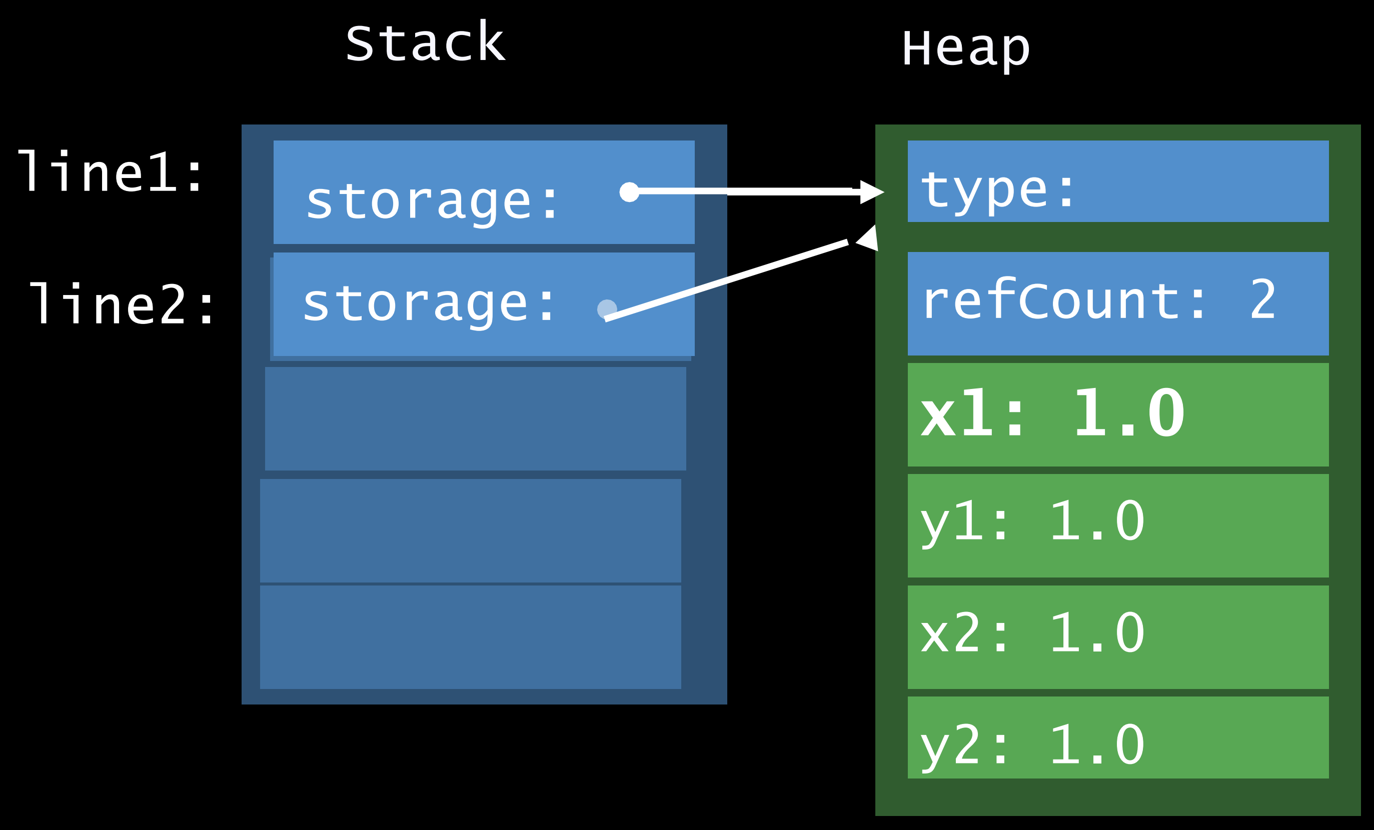
Copy-on-Write both read only



Stack

line1:
x1: 1.0
y1: 1.0
x2: 1.0
y2: 1.0

line2:
x1: 1.0
y1: 1.0
x2: 1.0
y2: 1.0



Stack

Heap

line1: storage:

line2: storage:

type:

refCount: 2

x1: 1.0

y1: 1.0

x2: 1.0

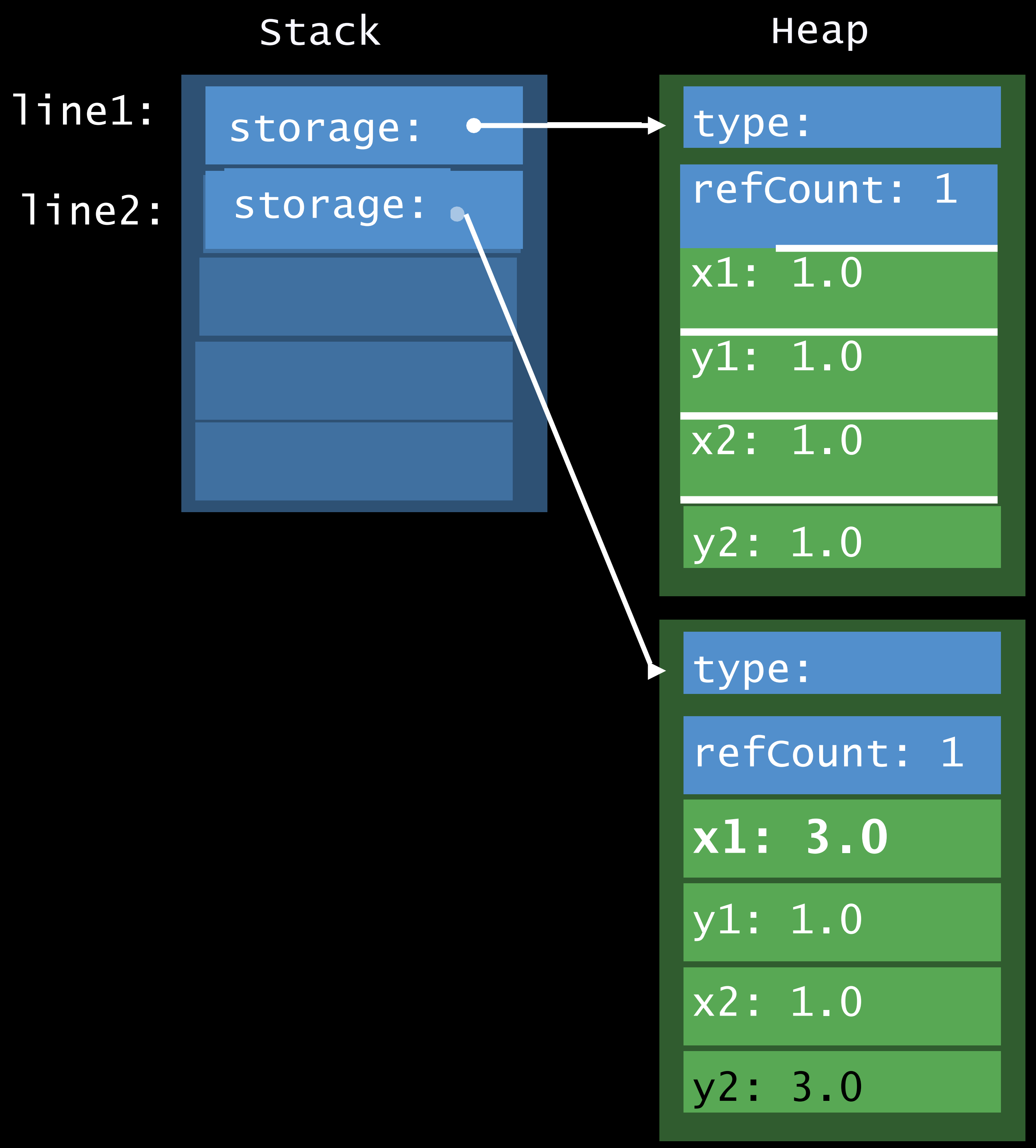
y2: 1.0

Copy-on-Write one read, one write

Stack

line1:	x1:	1.0
	y1:	1.0
	x2:	1.0
	y2:	1.0
line2:	x1:	1.0
	y1:	1.0
	x2:	1.0
	y2:	3.0

```
line2.storage.y2 = 3.0
```



```
line2.storage.y2 = 3.0
```

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 = ...
   }
}
```

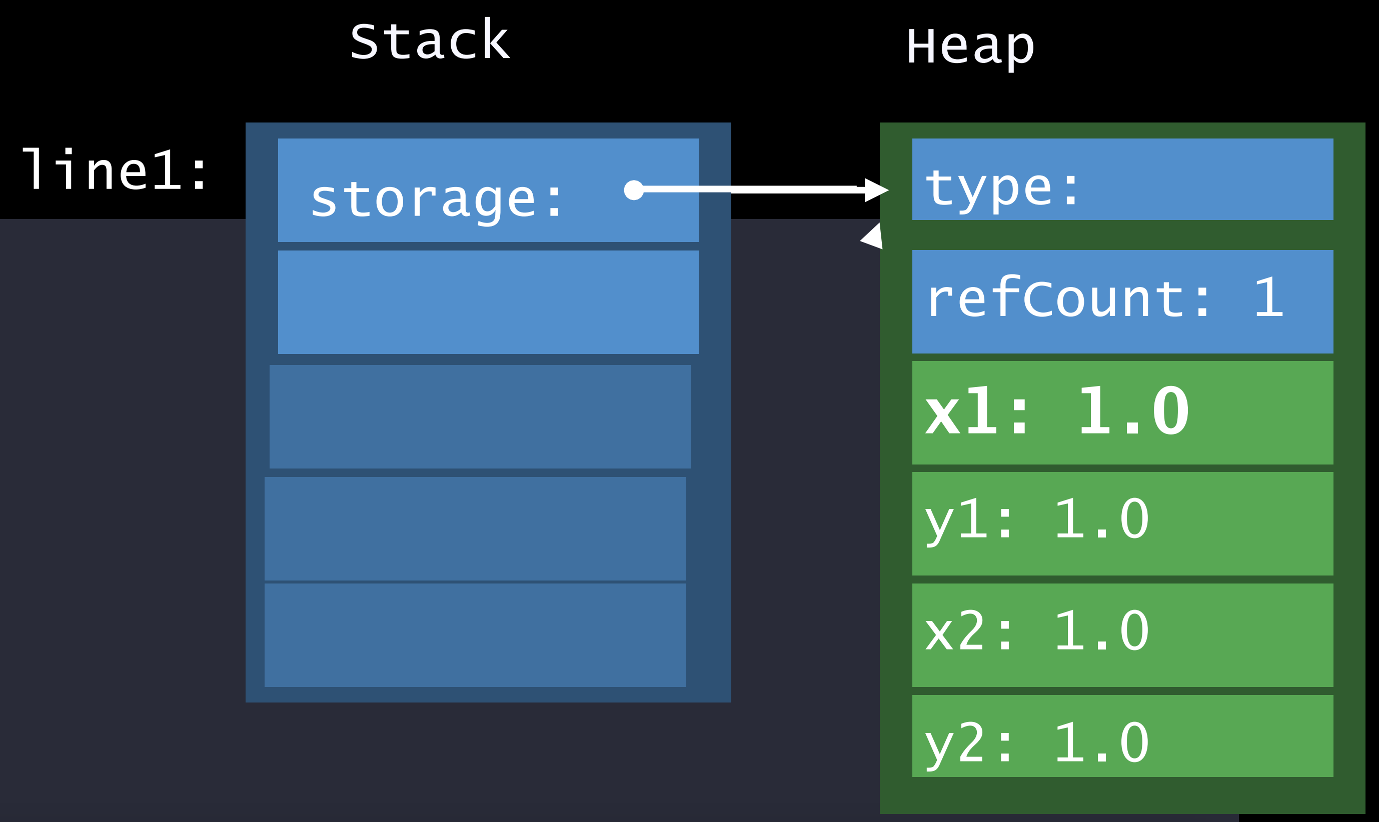
Indirect Storage with Copy-on-Write

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 = ...
   }
}
```

Indirect Storage with Copy-on-Write

Implement copy-on-write



Stack

Heap

line1:

storage:

type:

refCount: 1

x1: 1.0

y1: 1.0

x2: 1.0

y2: 1.0

```
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

Data structures in swift standard library

- String
- Array
- Dictionary

Method Dispatch

Method Dispatch

Static

Jump directly to implementation at run time

Candidate for inlining and other optimizations

Method Dispatch

Static

Jump directly to implementation at run time

Candidate for inlining and other optimizations

Dynamic

Look up implementation in table at run time

Then jump to implementation

Prevents inlining and other optimizations

Method Dispatch

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

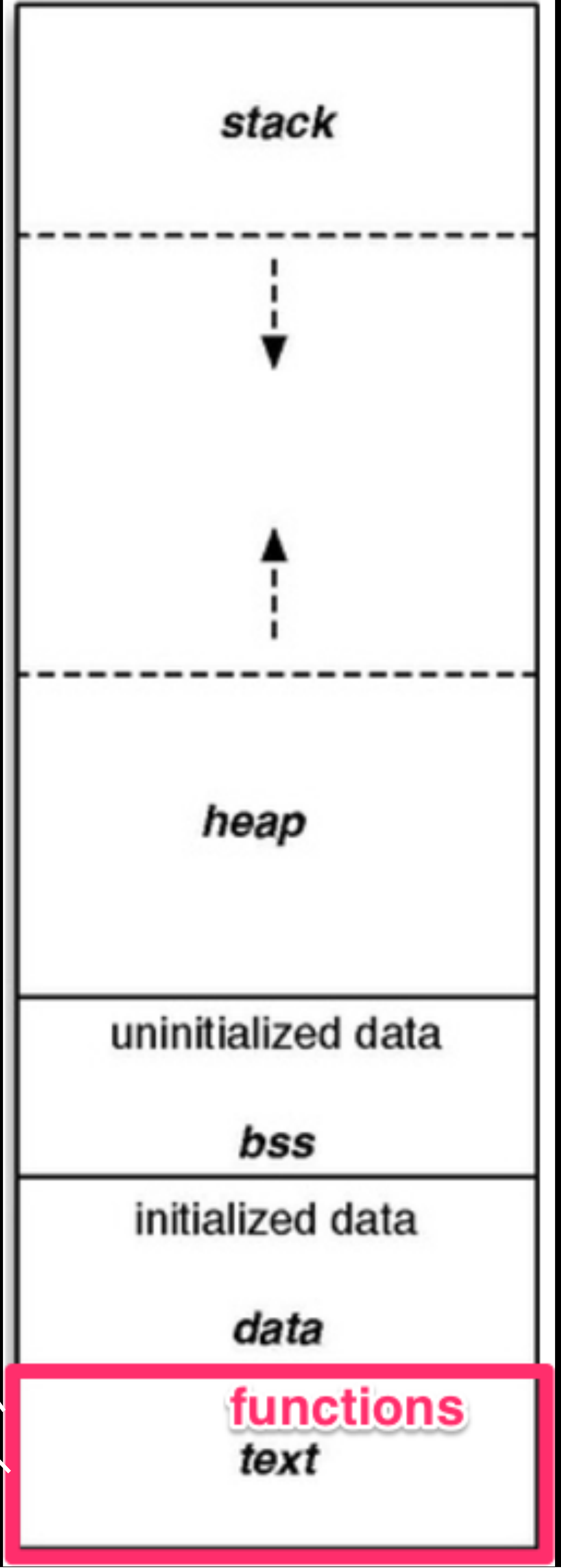
Point.draw()

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

Point.draw()

```
class Point3D: Point {
   var z: Double
   override func draw() { … }
}
```

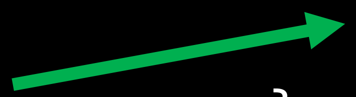
Point3D.draw()



Method Dispatch

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

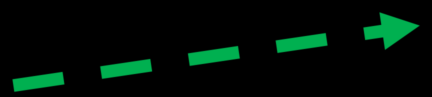
Point.draw()



```
func drawAPoint(_ point: Point) {
   point.draw()
}
```

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

Point.draw()



```
func drawAPoint(_ point: Point) {
   point.draw()
}
```

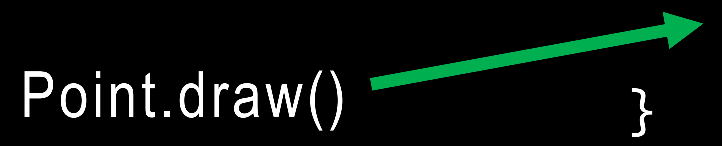
```
class Point3D: Point {
   var z: Double
   override func draw() { … }
}
```

Point3D.draw()

Method Dispatch

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

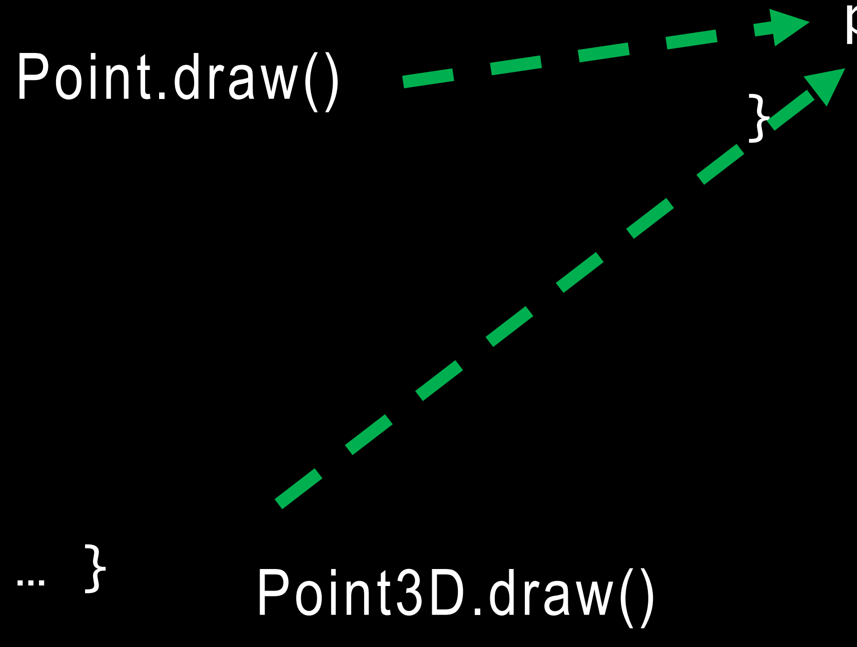
```
func drawAPoint(_ point: Point) {
    point.draw()
}
```



Point.draw()

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

class Point3D: Point {
    var z: Double
    override func draw() { … }
}
```

```
func drawAPoint(_ point: Point) {
    point.draw()
}
```



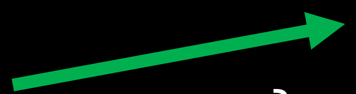
Point.draw()

Point3D.draw()

Method Dispatch

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

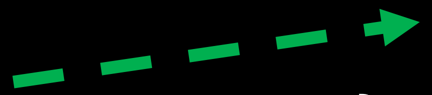
Point.draw()



```
func drawAPoint(_ point: Point) {
    point.draw()
}
```

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

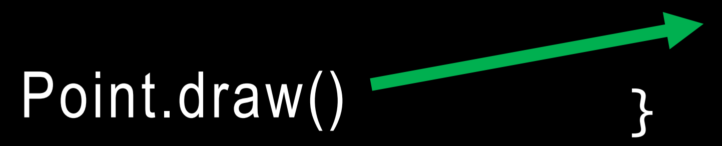
Point.draw()



```
func drawAPoint(_ point: Point) {
    point.draw()
}
```

Method Dispatch

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

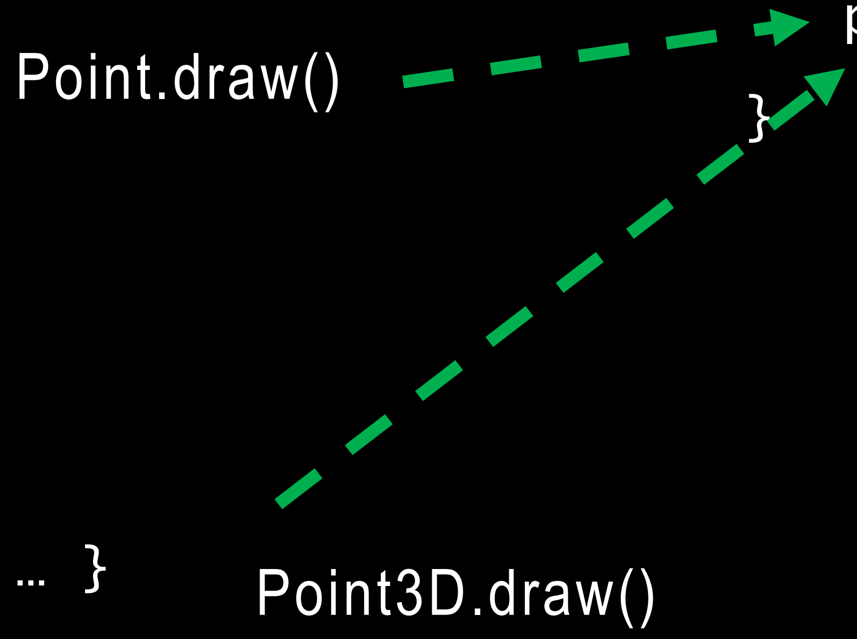


Point.draw()

```
func drawAPoint(_ point: Point) {
    point.draw()
}
```

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

class Point3D: Point {
    var z: Double
    override func draw() { … }
}
```

```
func drawAPoint(_ point: Point) {
    point.draw()
}
```

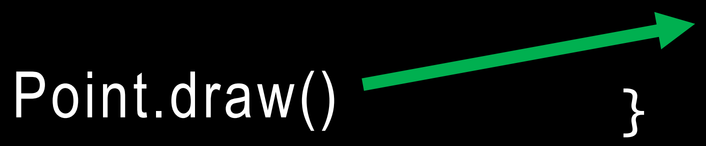


Point.draw()

Point3D.draw()

Method Dispatch (inline)

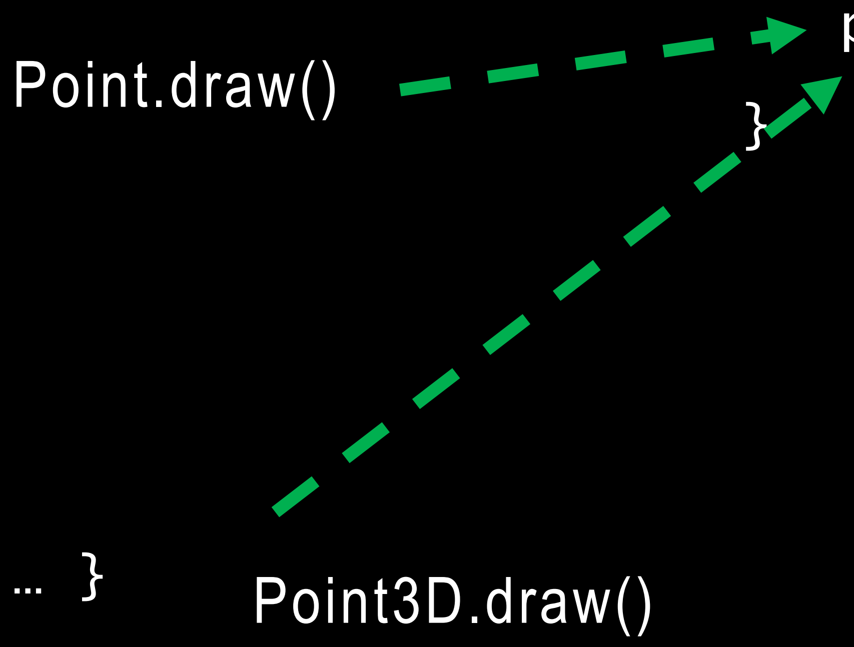
```
struct Point {
    var x, y: Double
    func draw() { … }
}
```



Point.draw()

```
func drawAPoint(_ point: Point) {
    point.draw()
}
```

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

class Point3D: Point {
    var z: Double
    override func draw() { … }
}
```



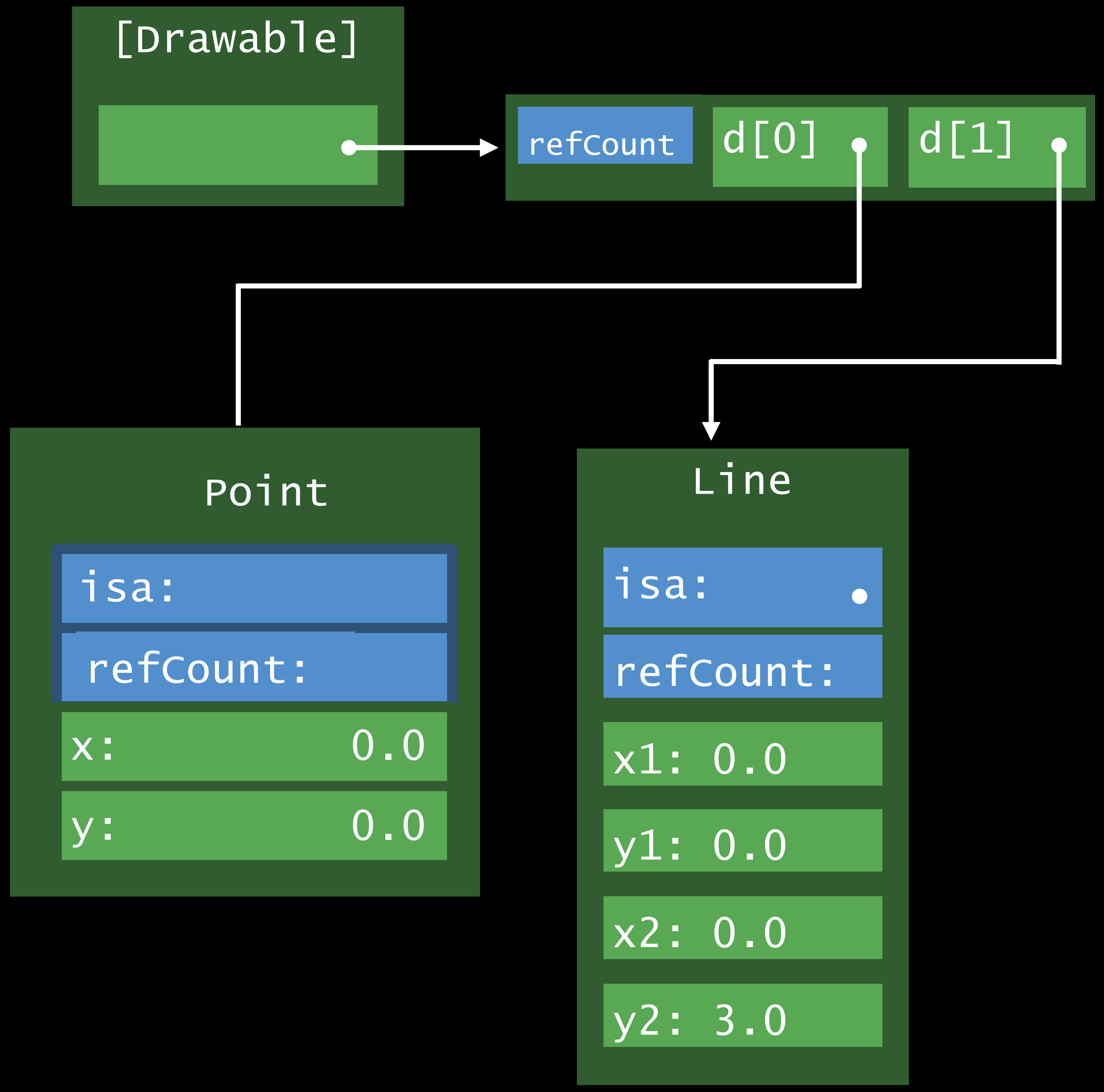
Point.draw()

Point3D.draw()

```
func drawAPoint(_ point: Point) {
    point.draw()
}
```

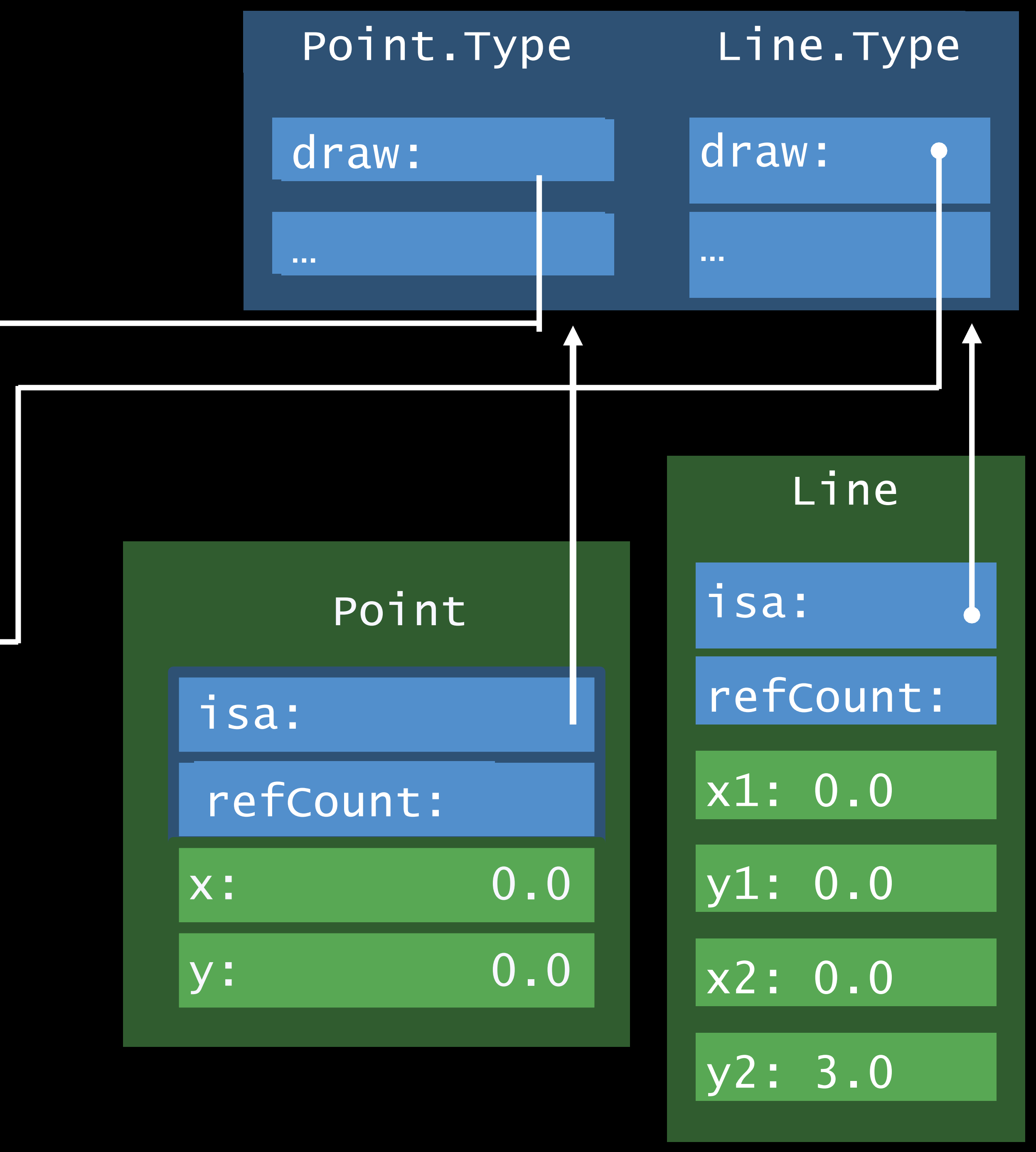
Polymorphism Through Reference Semantics

```
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()
}
```



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)
}
```



C++ vs Swift vs Objective-C

Dynamic Dispatch

Implementation principle

- C++ & Swift: vtable, no need to delegate to super class
- Objective-C: selector string pointer to function implementation map. Delegate to super class map

Performance

- Swift ~= C++ > Objective-C

Protocol Types

Protocol Types

```
protocol Drawable { func draw() }

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

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

Value type or reference type ???

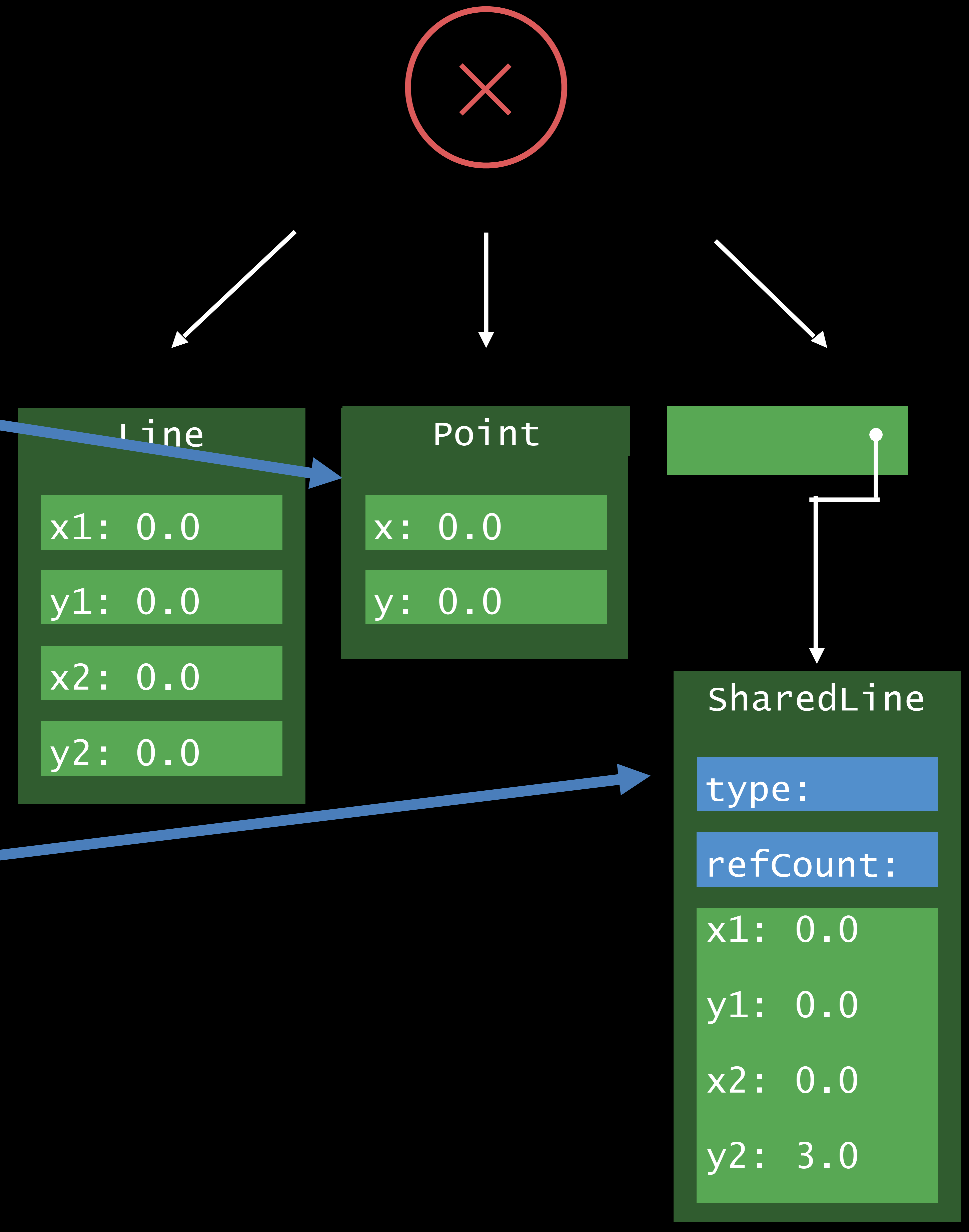
```
var d: Drawable = Point(x: 0, y: 0)
 d  = SharedLine (x1: 0, y1: 0, x2: 0, y2: 0)
 d.draw()
```

Program = data + method

Two Problems

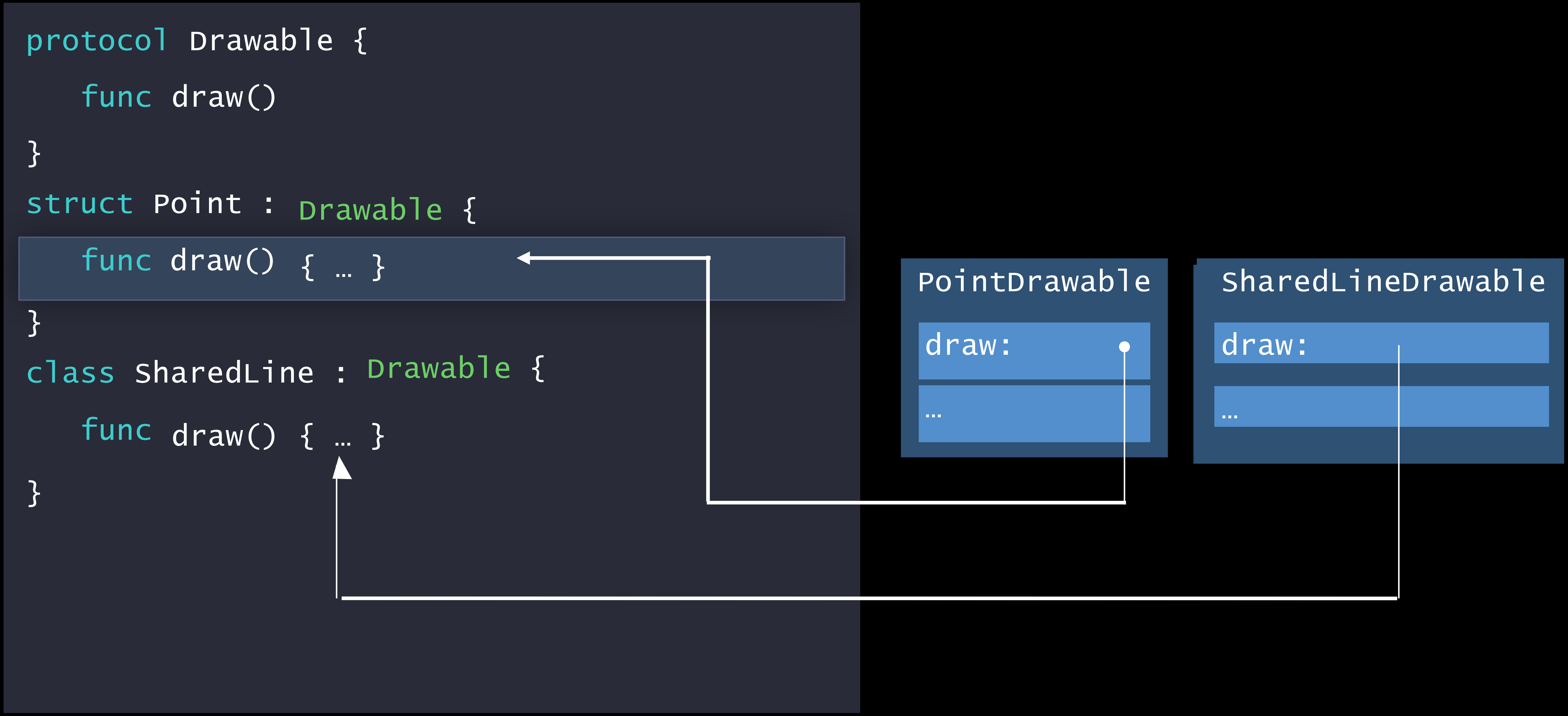
store data & dispatch method uniformly

```
protocol Drawable { func draw() }

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

var d: Drawable = Point(x: 0, y: 0)
 d = SharedLine (x1: 0, y1: 0, x2: 0, y2: 0)
 d.draw()
```



The Protocol Witness Table (PWT)

Dynamic dispatch without a V-Table

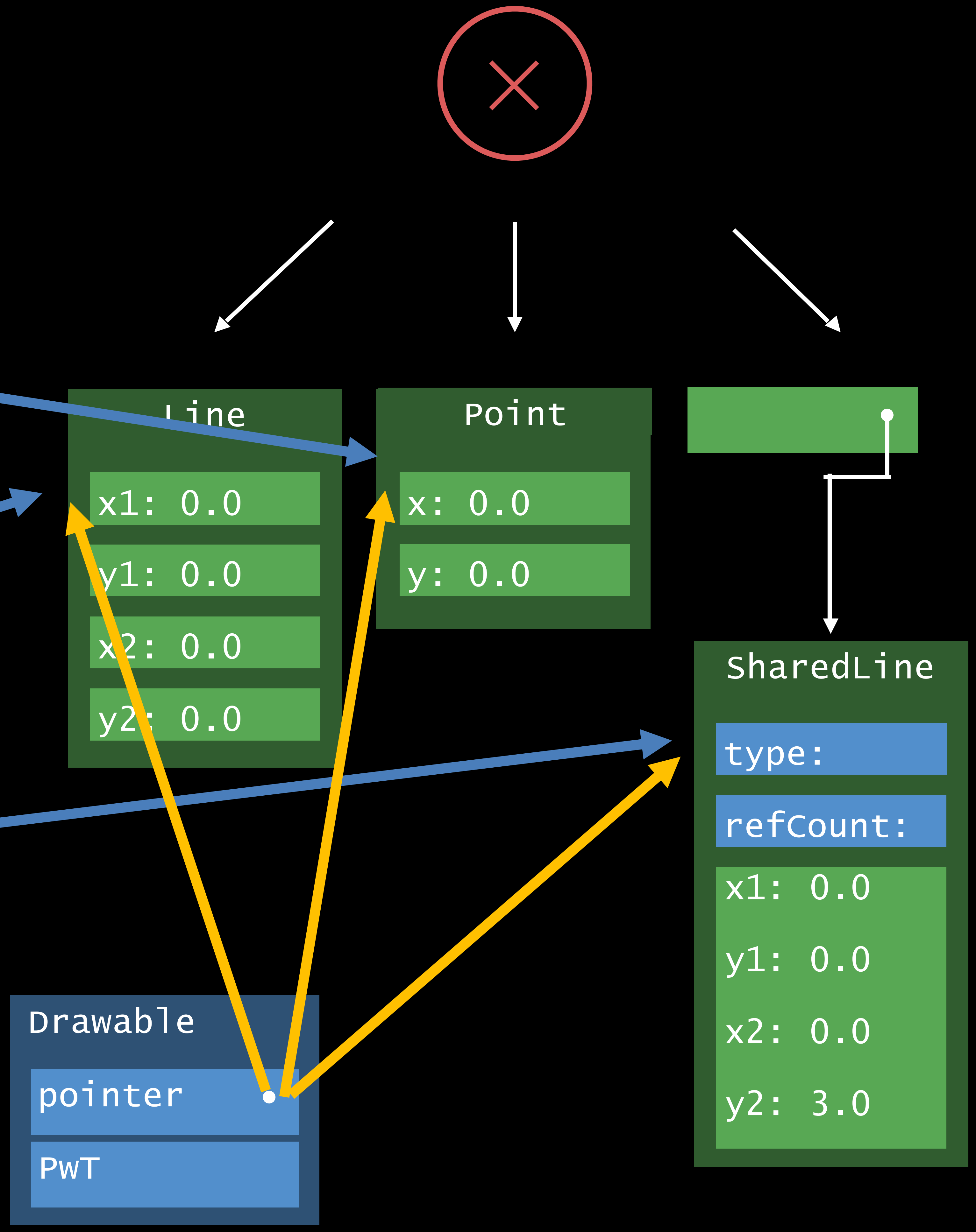


```
protocol Drawable {
    func draw()
}
struct Point : Drawable {
    func draw() { … }
}
class SharedLine : Drawable {
    func draw() { … }
}
```

Two Problems

store data & dispatch method uniformly

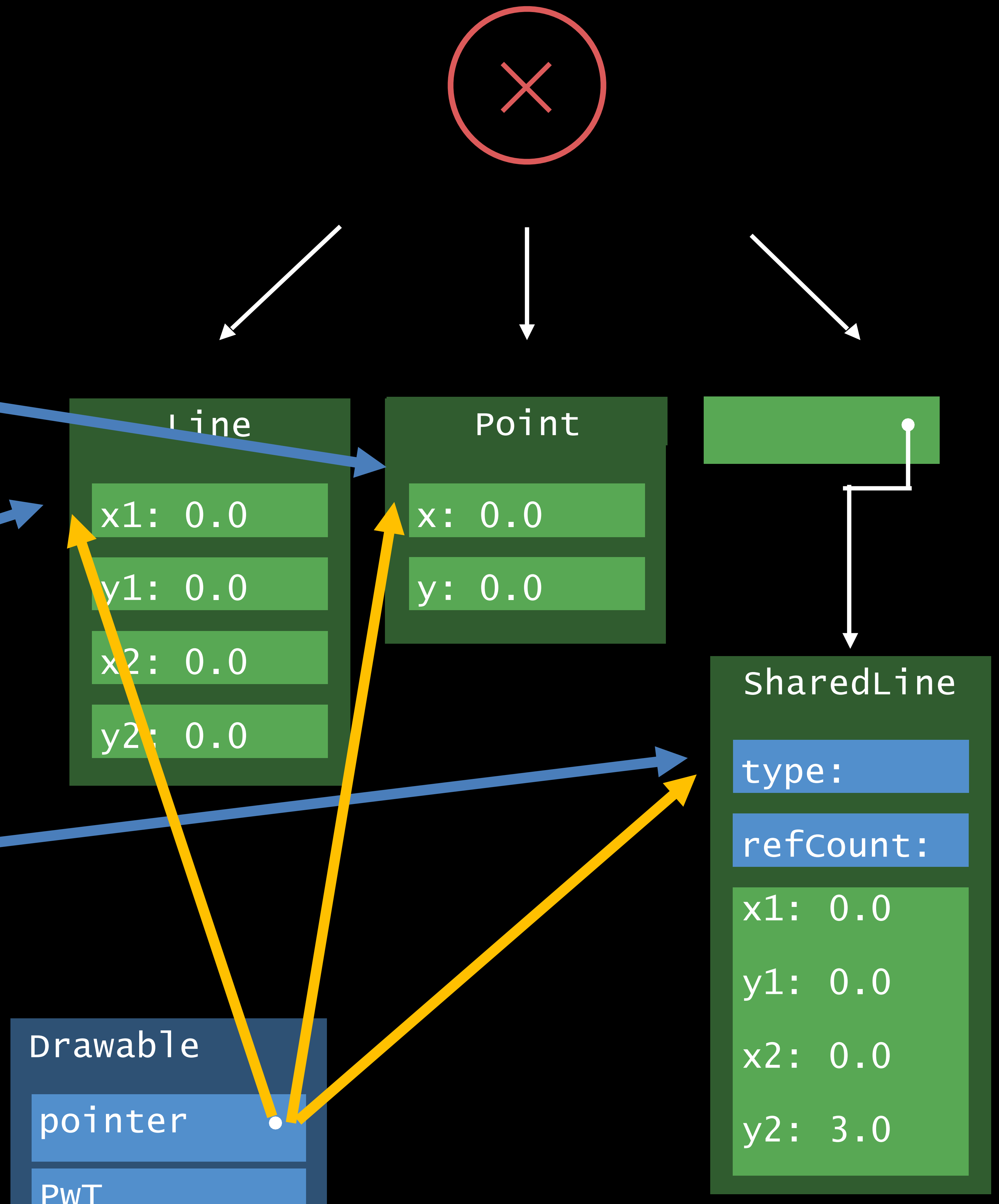
```
protocol Drawable { func draw() }

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

var d: Drawable = Point(x: 0, y: 0)
 d  = SharedLine (x1: 0, y1: 0, x2: 0, y2: 0)
 d.draw()
```



Two Problems

store data & dispatch method 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() { … }
}
class SharedLine : Drawable {
   var x1, y1, x2, y2: Double
   func draw() { … }
}

var d: Drawable = Point(x: 0, y: 0)
 d = SharedLine (x1: 0, y1: 0, x2: 0, y2: 0)
 d.draw()
 var d2: Drawable = d
```



Value type. Copy?

Two Problems

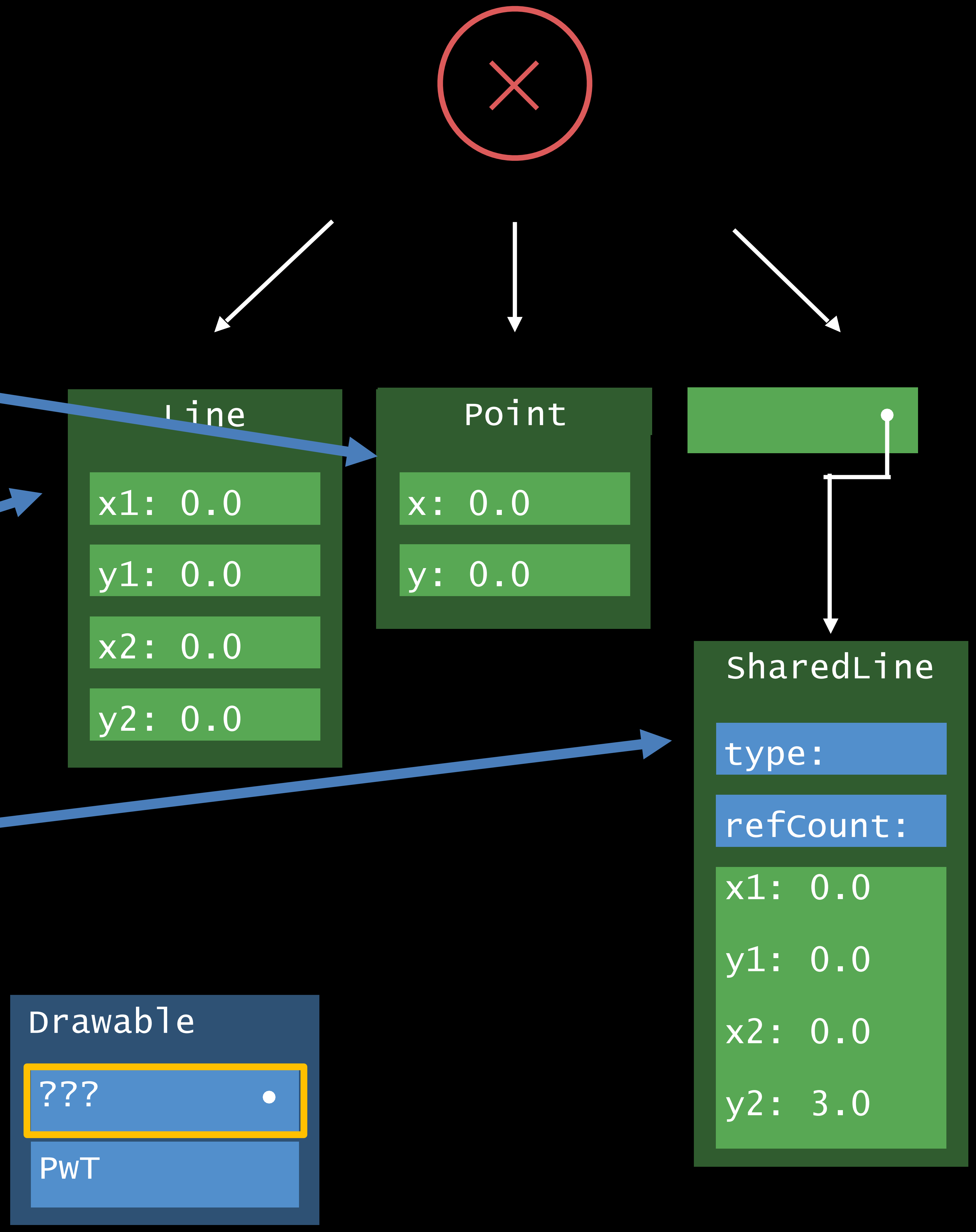
store data & dispatch method uniformly

```
protocol Drawable { func draw() }

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

var d: Drawable = Point(x: 0, y: 0)
 d  = SharedLine (x1: 0, y1: 0, x2: 0, y2: 0)
 d.draw()

var d2: Drawable
```

draw(self: Point)

draw(self: Line)

draw(self: SharedLine)



Line

x1: 0.0

y1: 0.0

x2: 0.0

y2: 0.0

Point

x: 0.0

y: 0.0

SharedLine

type:

refCount:

x1: 0.0

y1: 0.0

x2: 0.0

y2: 3.0

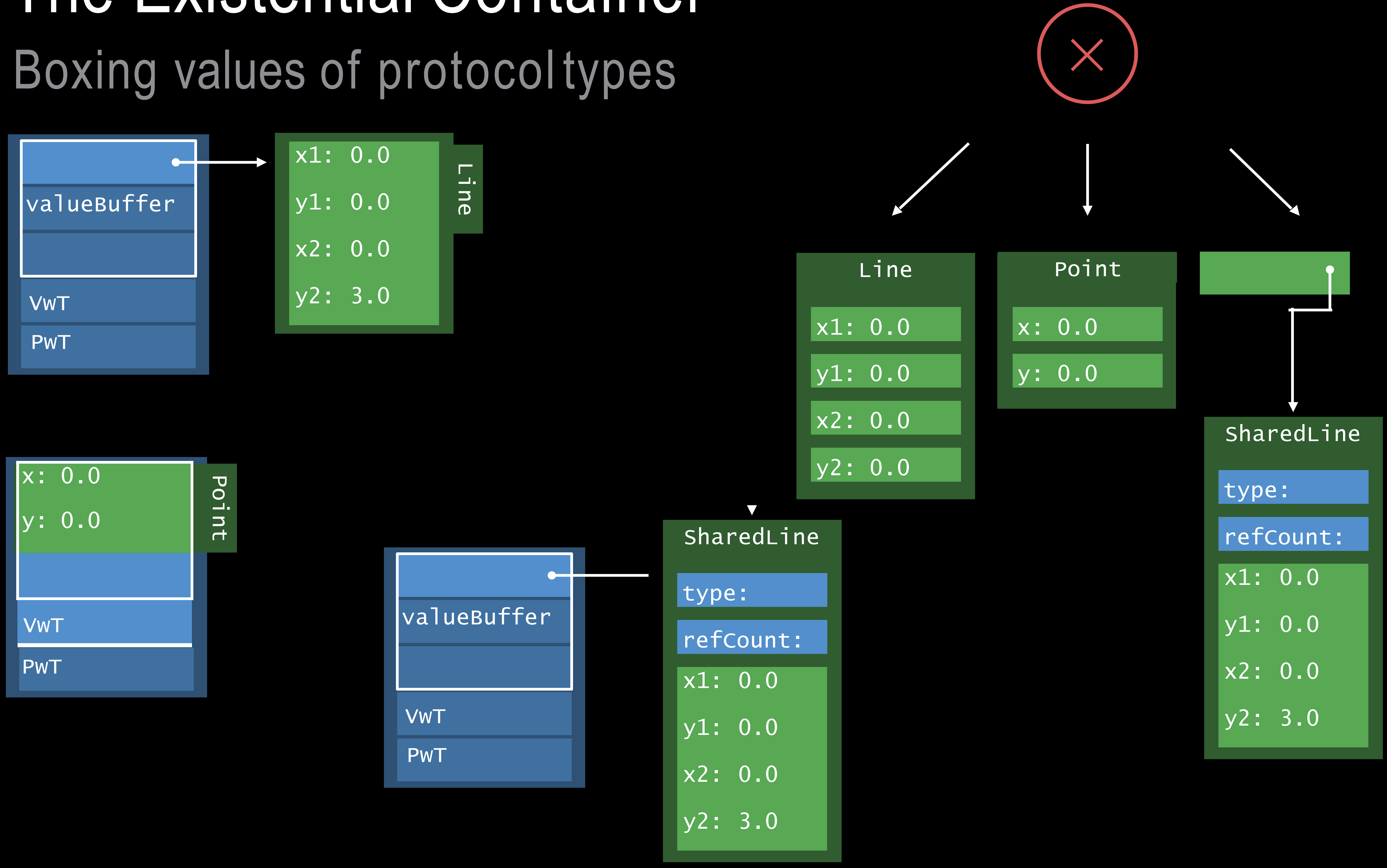
Drawable

???

PWT

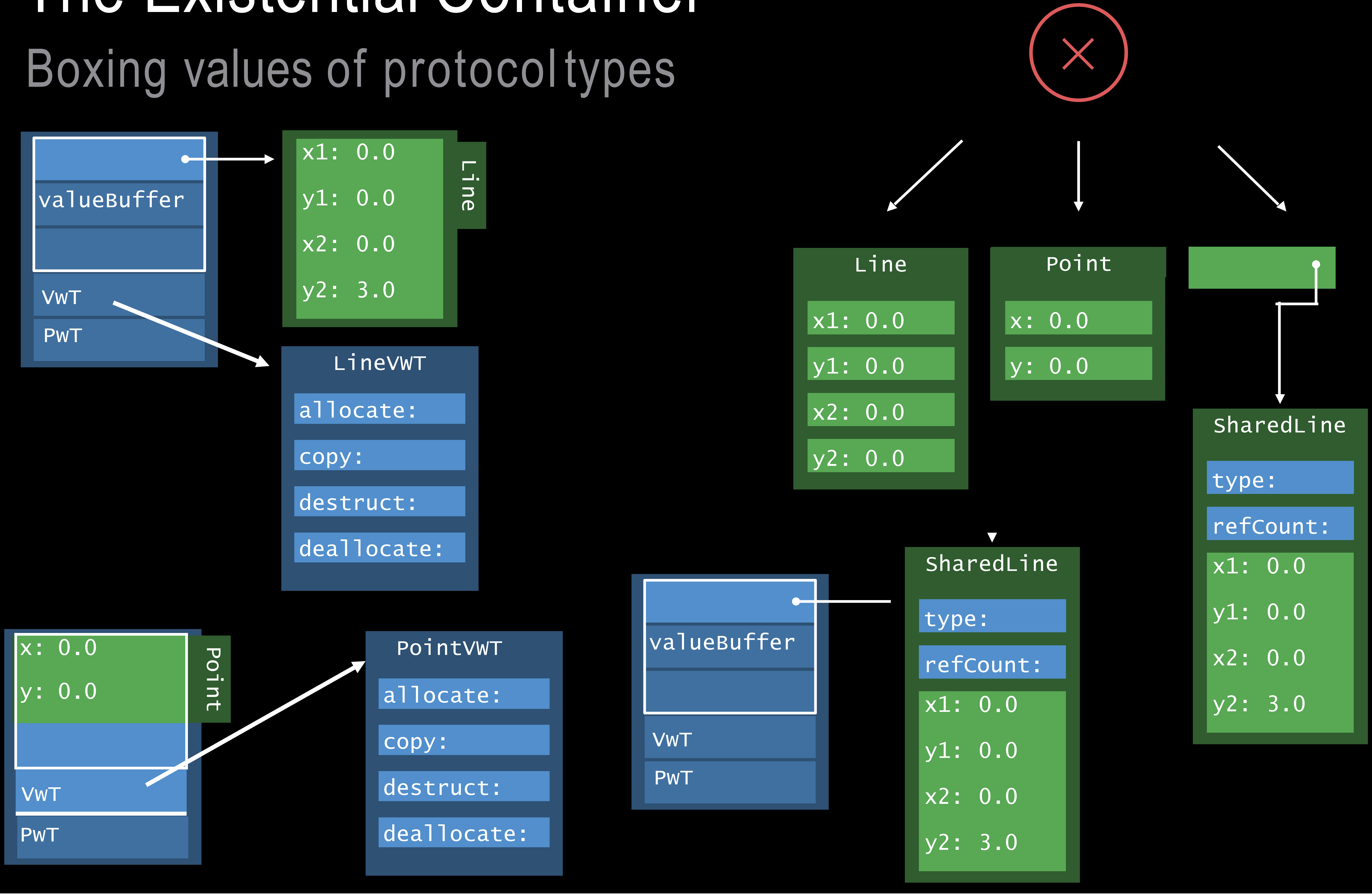
The Existential Container

Boxing values of protocol types



The Existential Container

Boxing values of protocol types



valueBuffer

VWT

PWT

x1: 0.0

y1: 0.0

x2: 0.0

y2: 3.0

Line

LineVWT

allocate:

copy:

destruct:

deallocate:

x: 0.0

y: 0.0

Point

VWT

PWT

PointVWT

allocate:

copy:

destruct:

deallocate:

valueBuffer

VWT

PWT

SharedLine

type:

refCount:

x1: 0.0

y1: 0.0

x2: 0.0

y2: 3.0

Line

x1: 0.0

y1: 0.0

x2: 0.0

y2: 0.0

Point

x: 0.0

y: 0.0

SharedLine

type:

refCount:

x1: 0.0

y1: 0.0

x2: 0.0

y2: 3.0

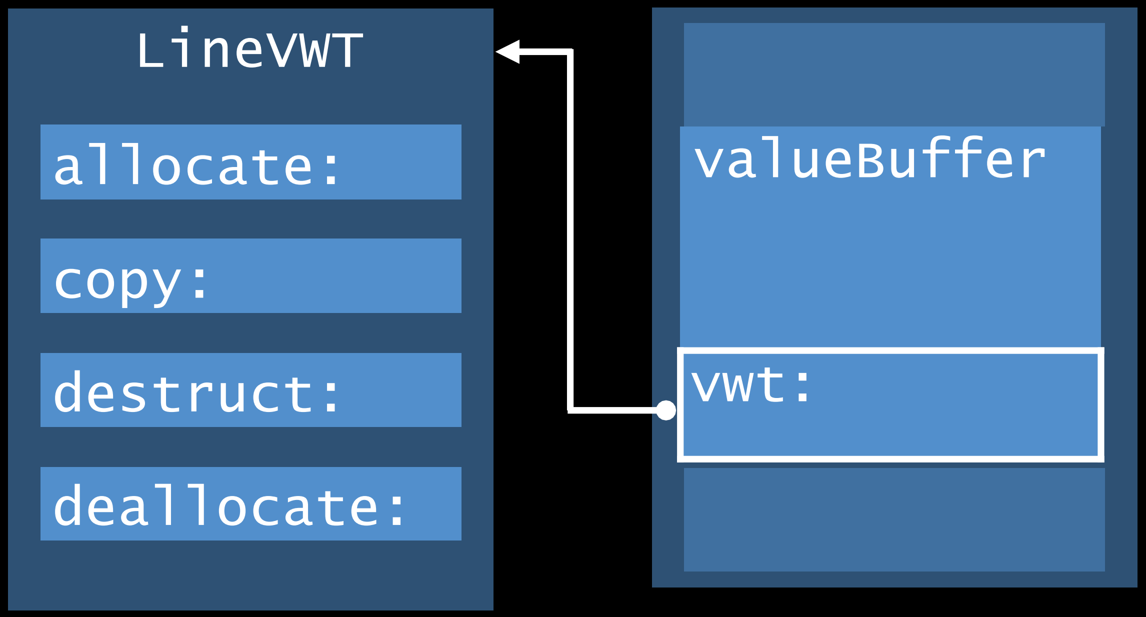
The Existential Container

Boxing values of protocol types

Inline Value Buffer: currently 3 words

Large values stored on heap

Reference to Value Witness Table



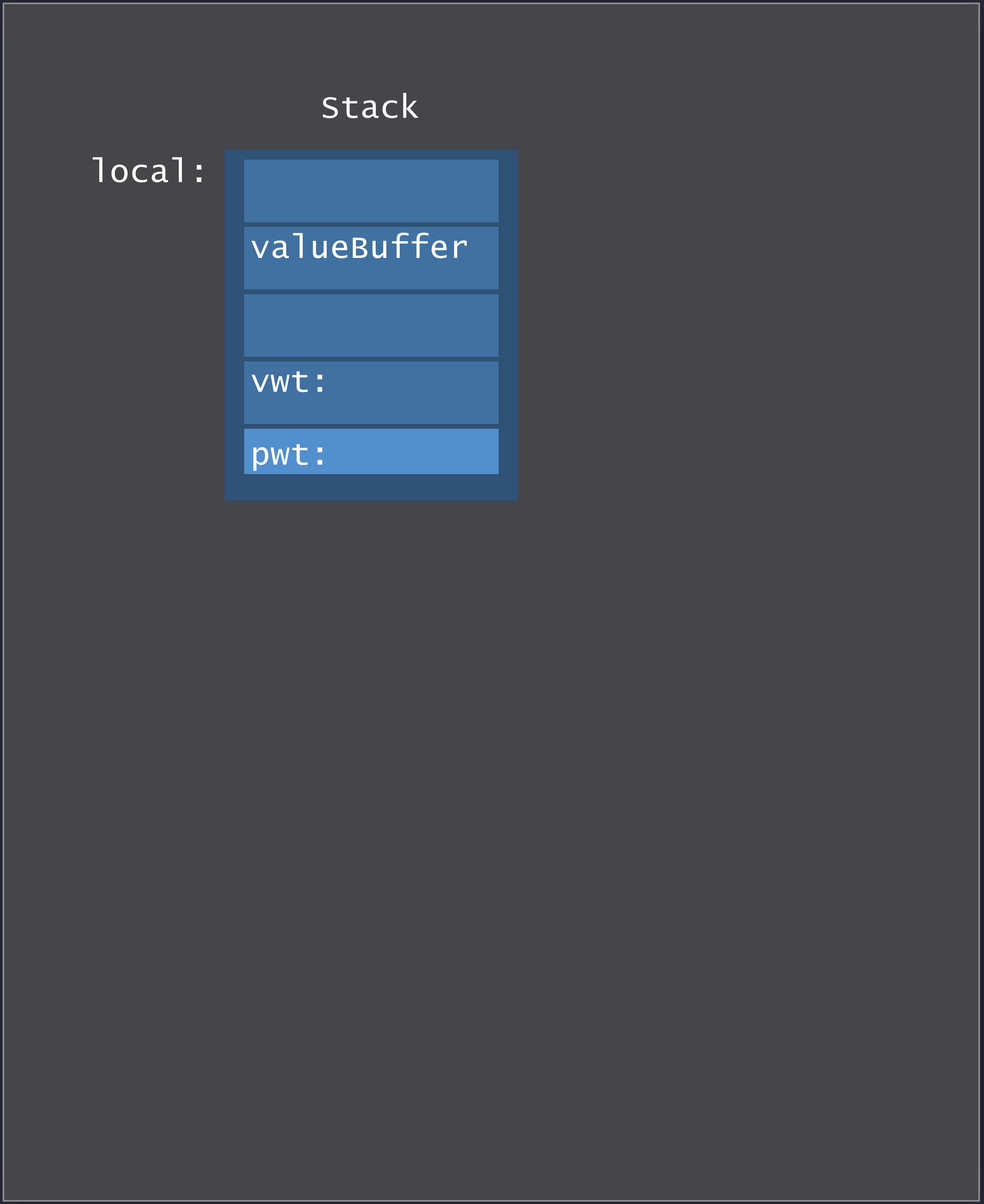
```swift
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
   local.draw()
}
let val : Drawable = Point()
drawACopy(val)
```

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

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

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

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



Stack

local:

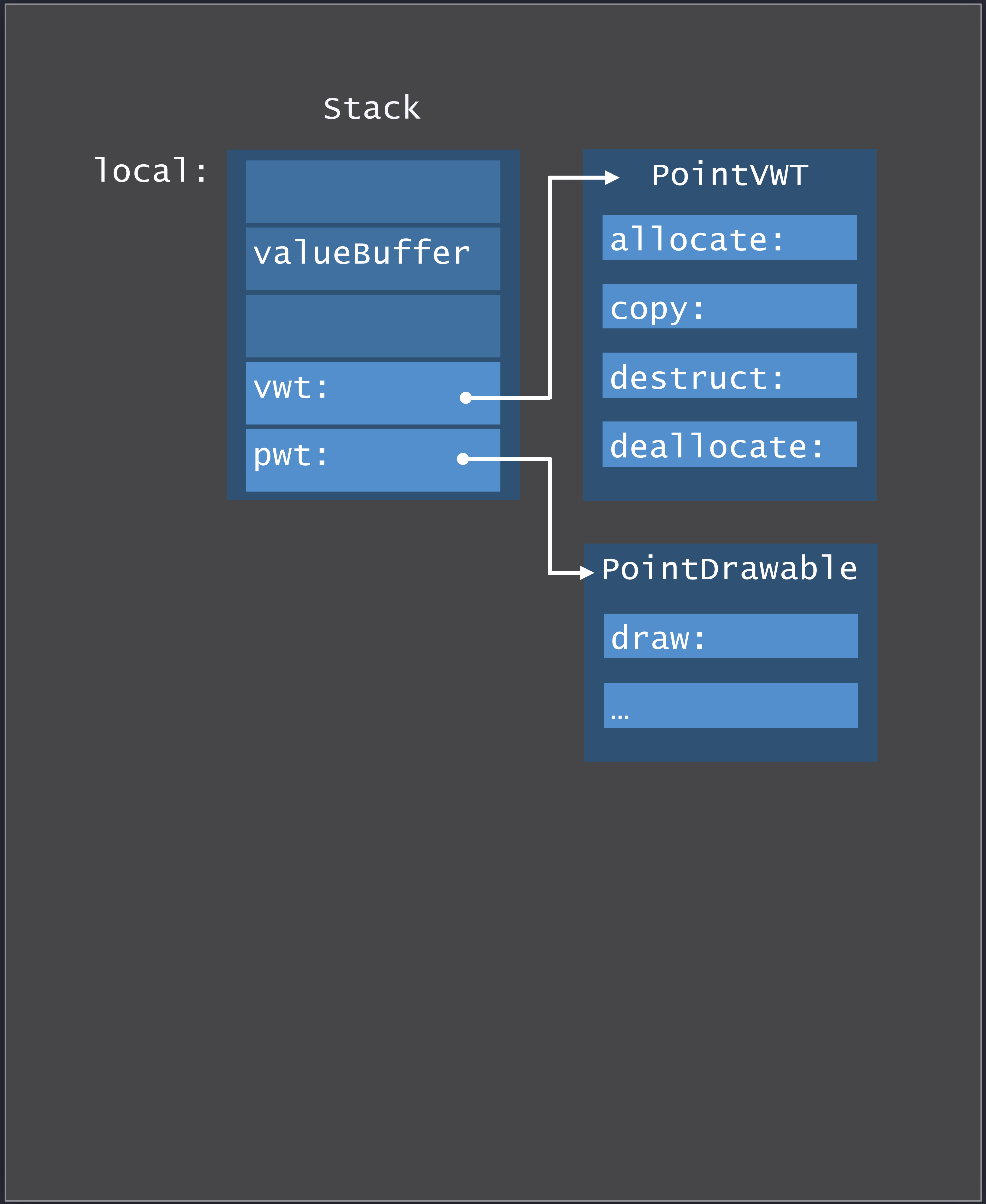
valueBuffer

vwt:

pwt:

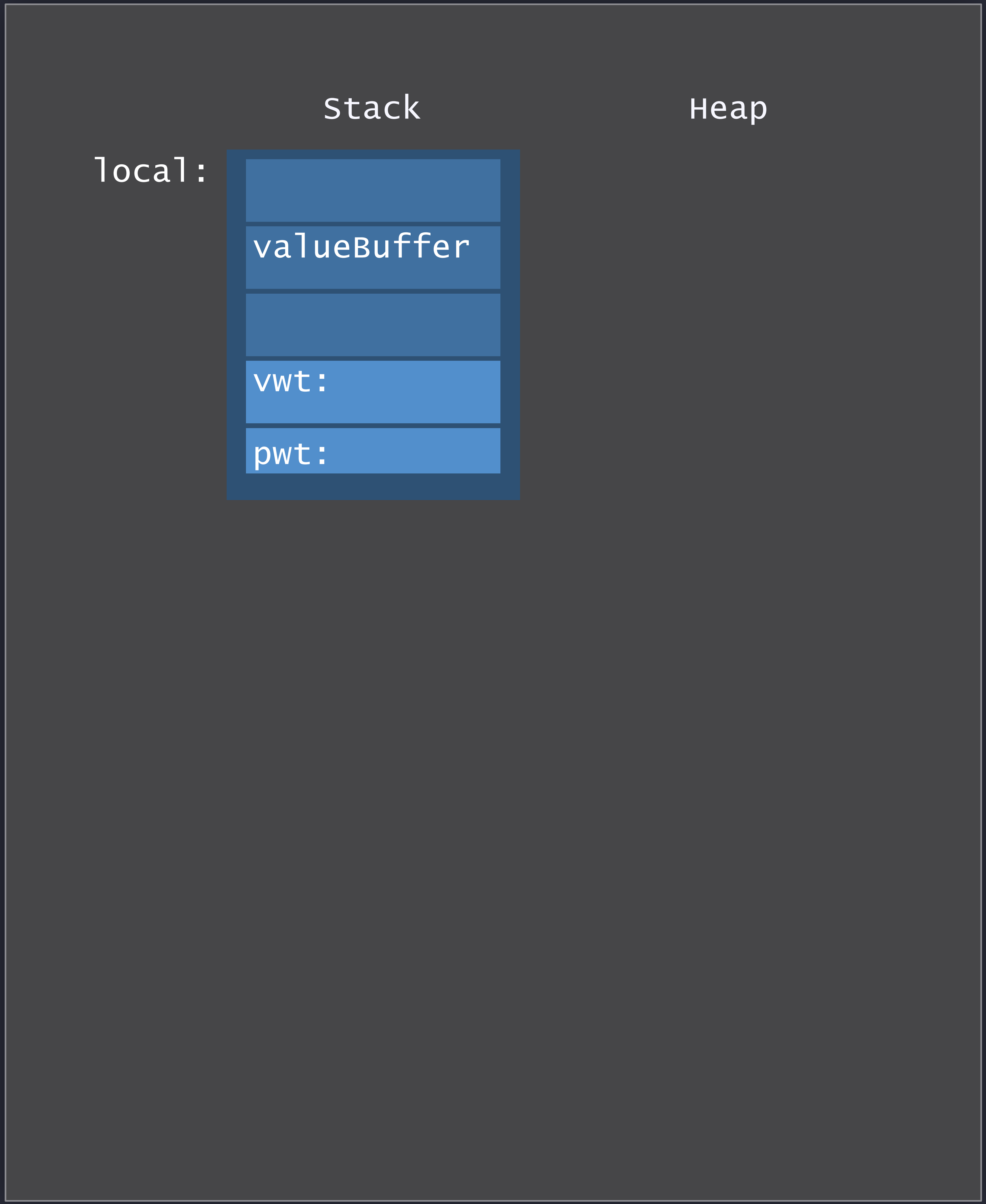
```
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
    local.draw()
}
let val : Drawable = Point()
drawACopy(val)
```

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



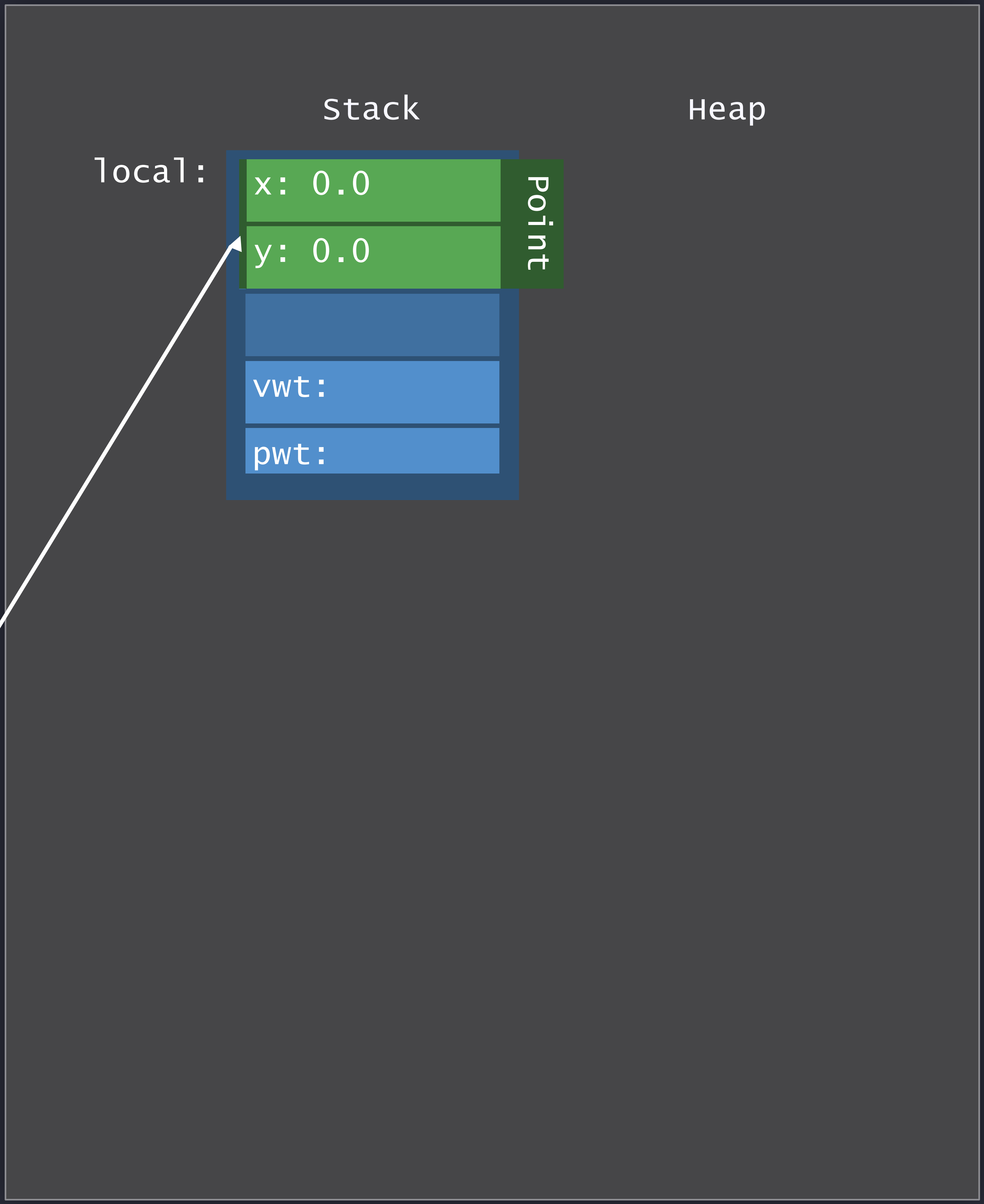
```
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
   local.draw()
}
let val : Drawable = Point()
drawACopy(val)
```

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



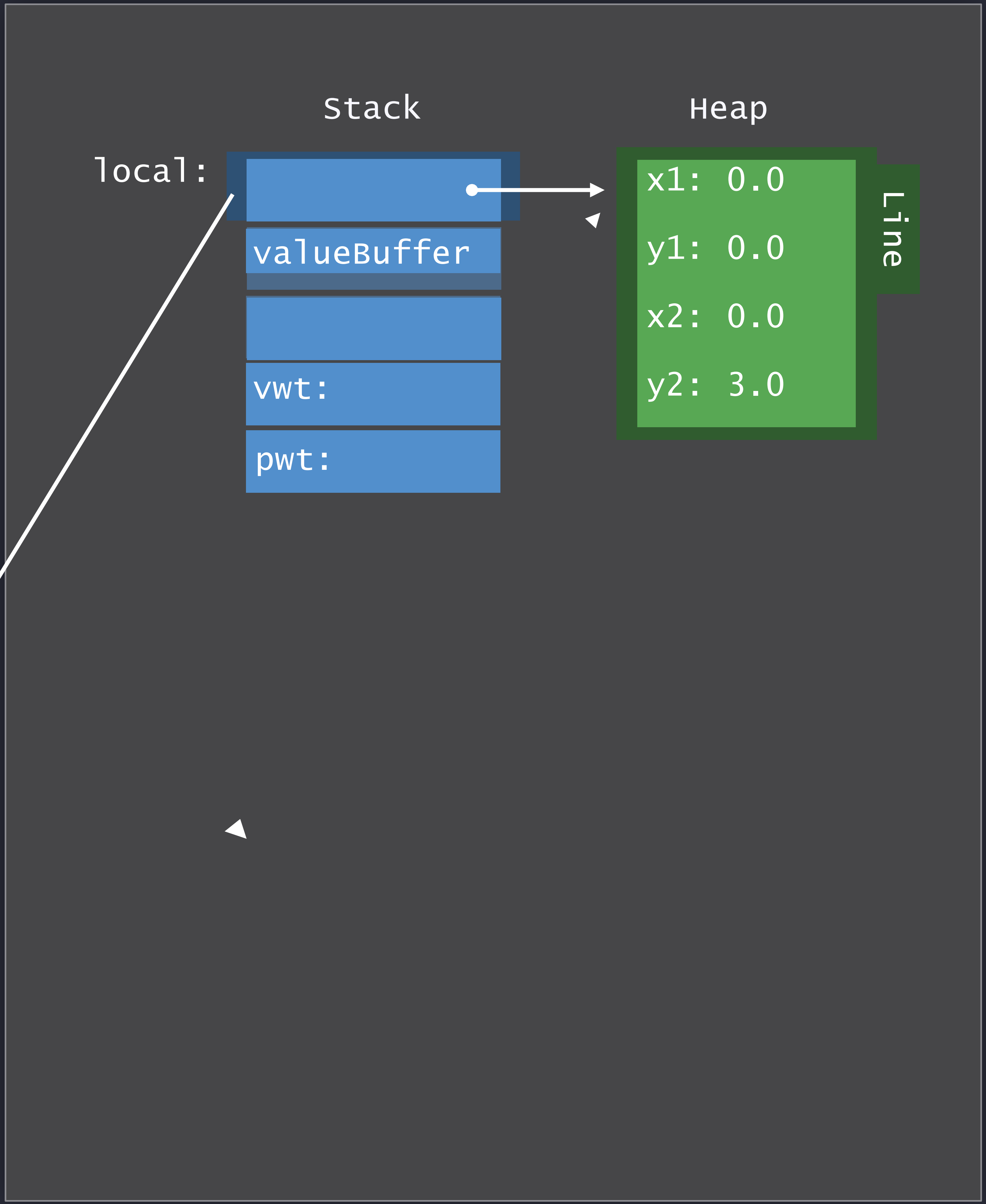
```
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
    local.draw()
}
let val : Drawable = Point()
drawACopy(val)
```

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



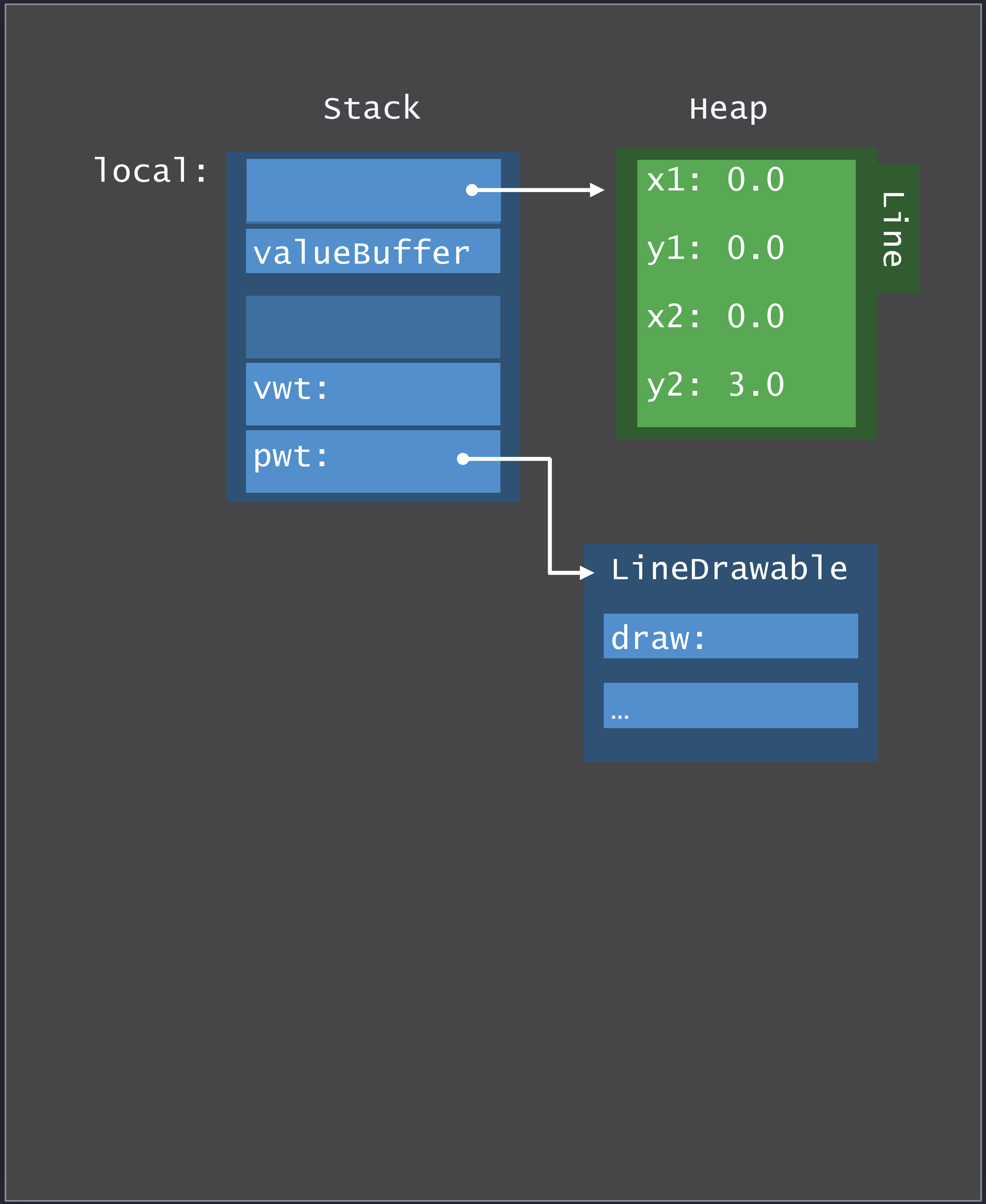
```
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
    local.draw()
}
let val : Drawable = Line()
drawACopy(val)
```

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



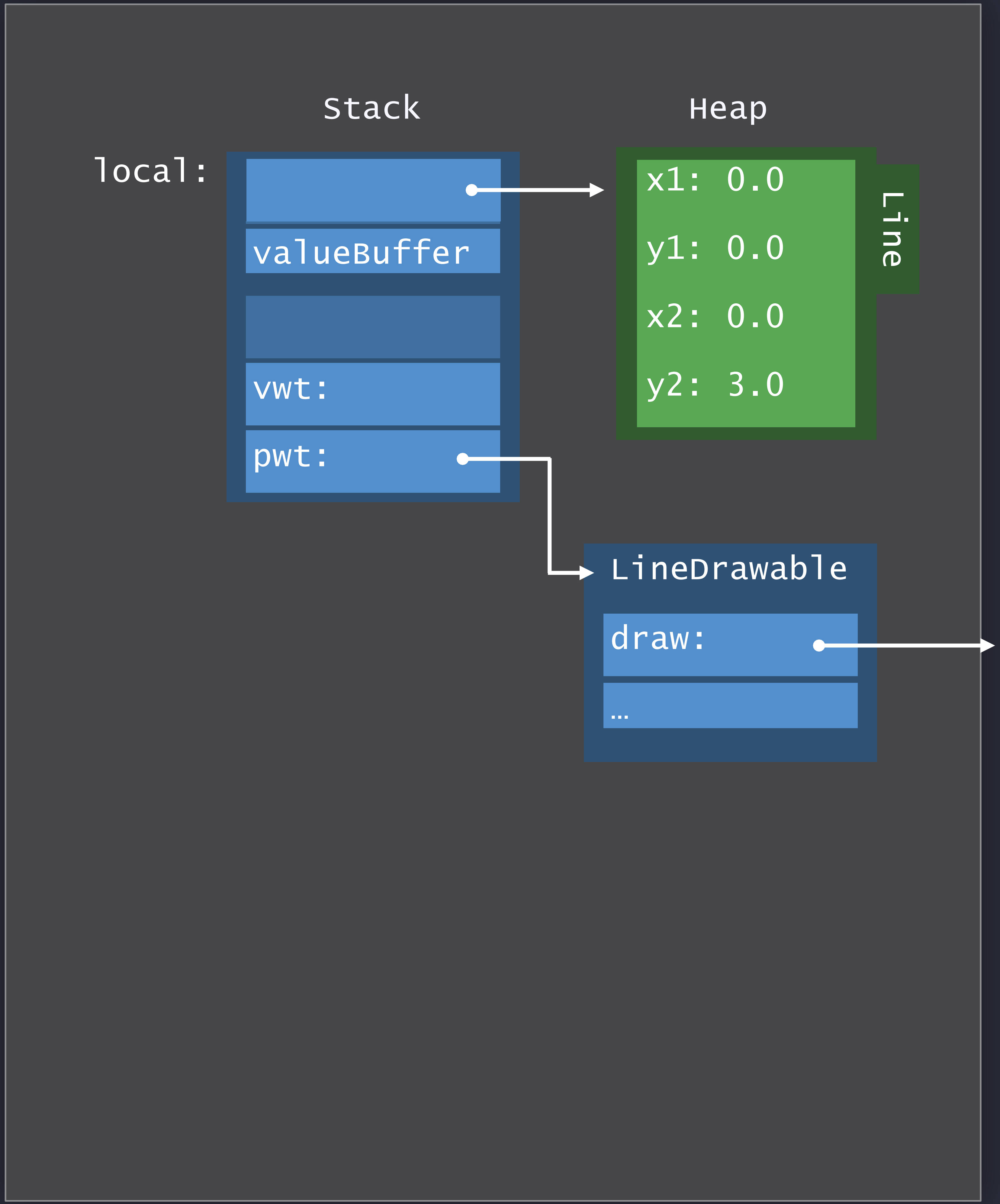
```
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
    local.draw()
}
let val : Drawable = Line()
drawACopy(val)
```

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



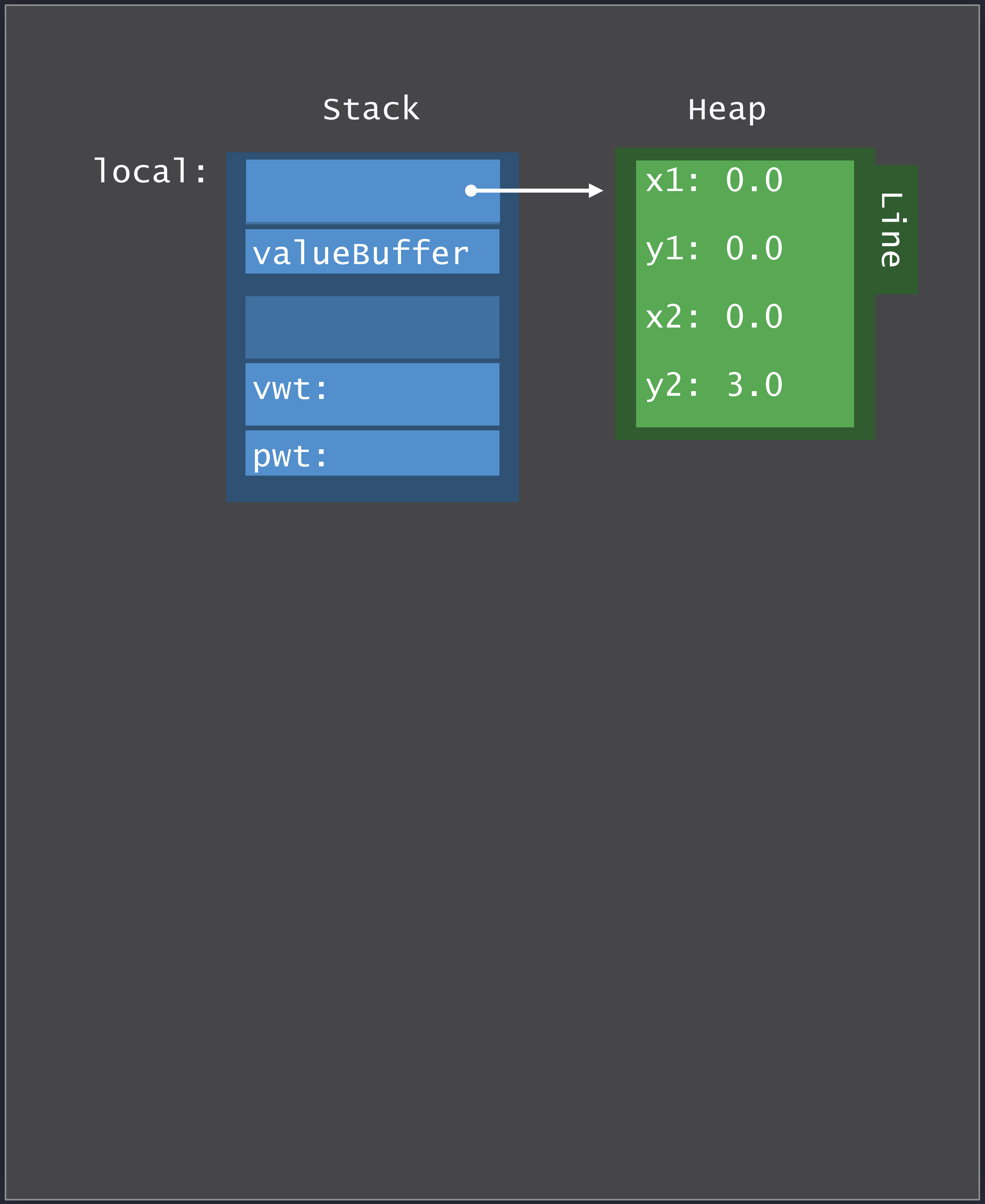
```
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
    local.draw()
}
let val : Drawable = Line()
drawACopy(val)
```

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



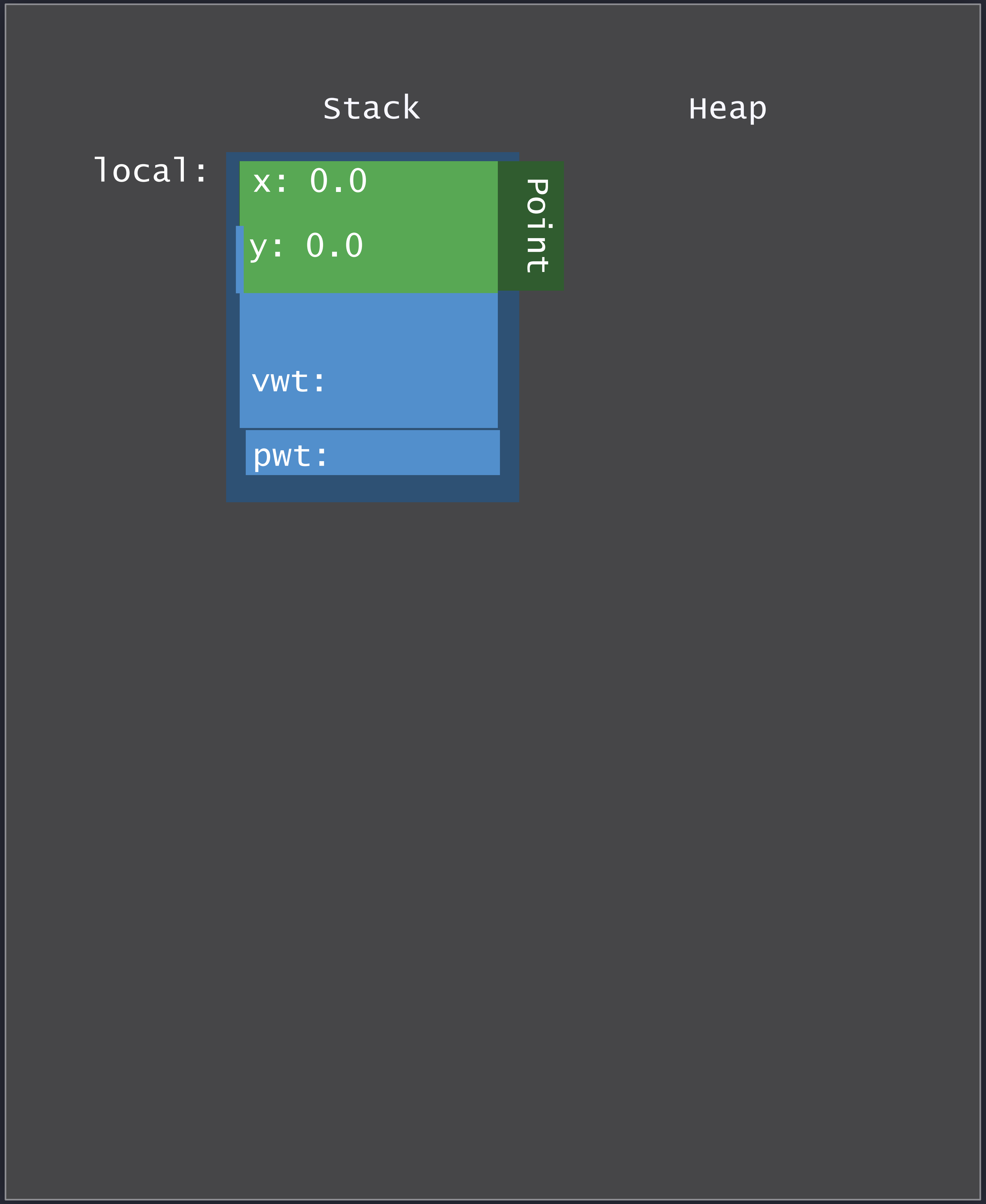
```
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
    local.draw()
}
let val : Drawable = Line()
drawACopy(val)
```

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



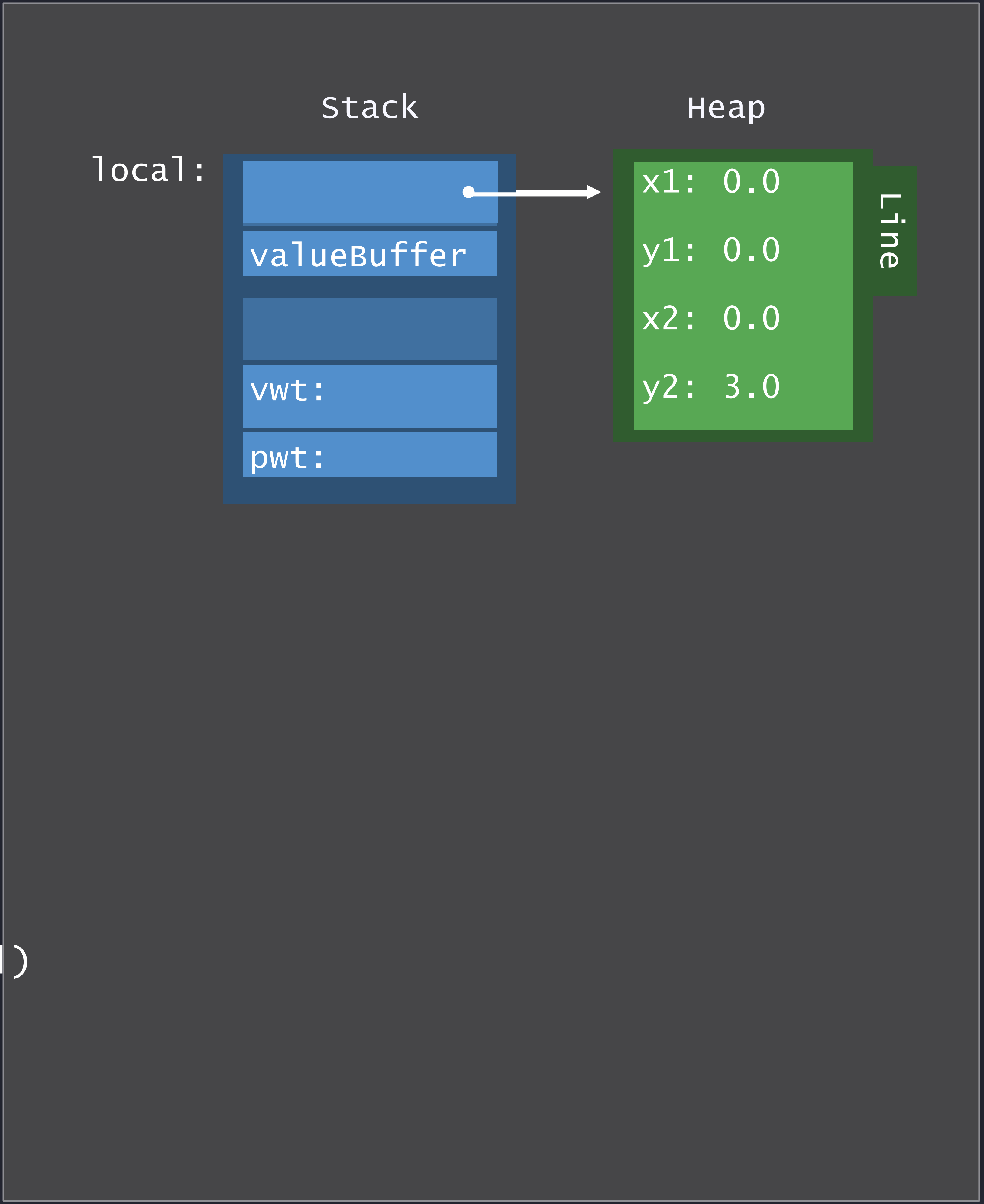
```
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
    local.draw()
}
let val : Drawable = Point()
drawACopy(val)
```

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



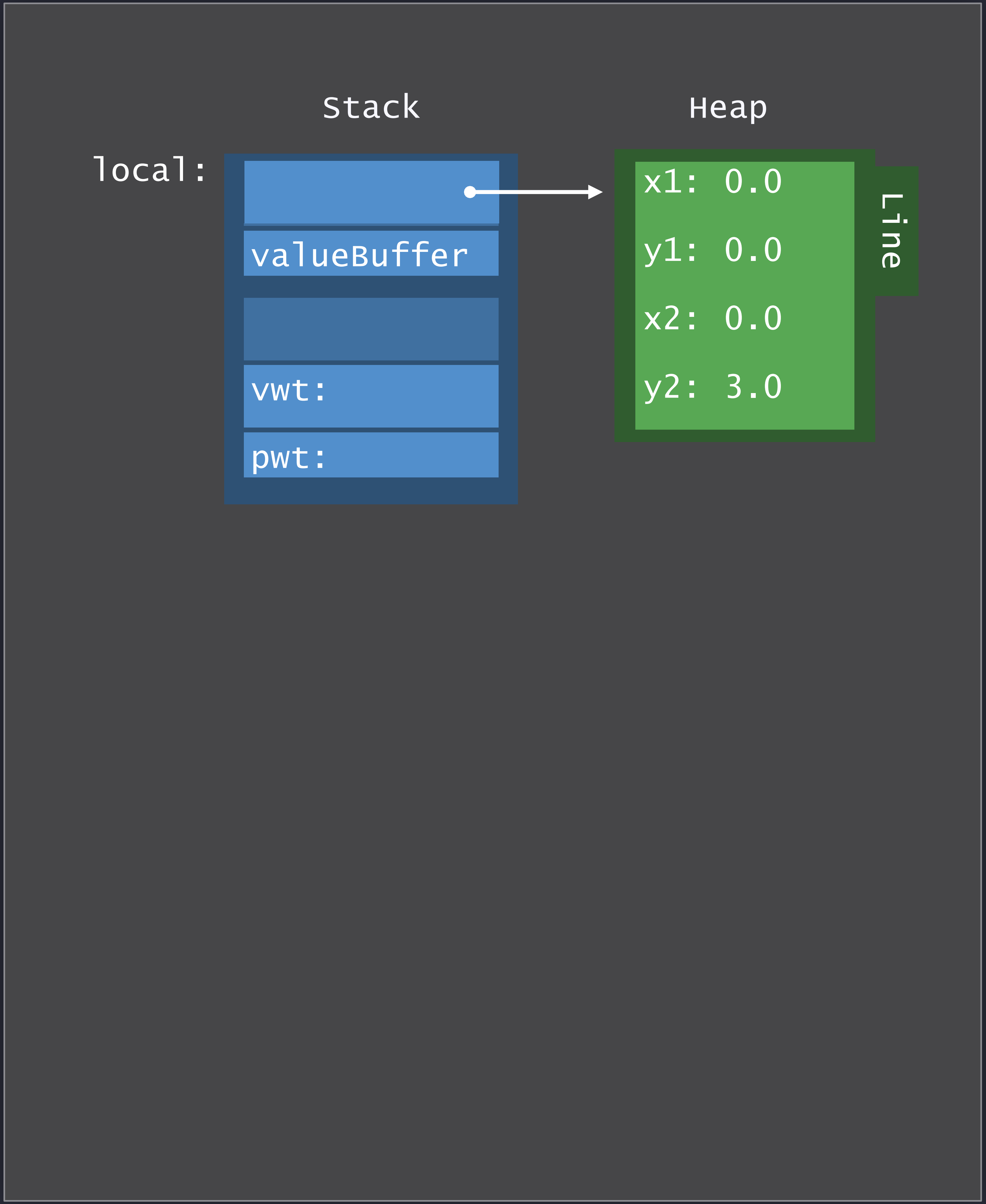
```
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
    local.draw()
}
let val : Drawable = Line()
drawACopy(val)
```

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



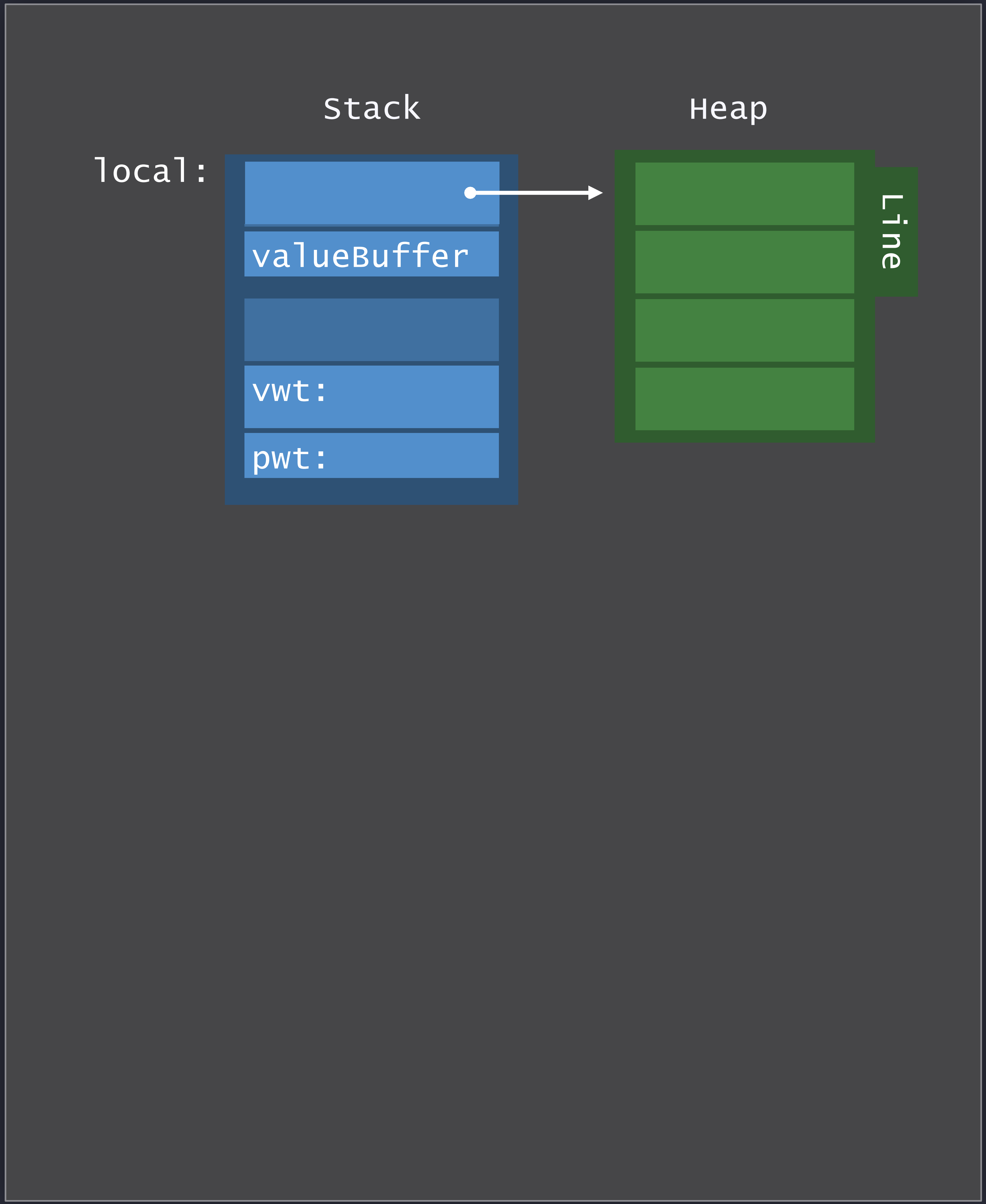
```
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
   local.draw()
}
let val : Drawable = Line()
drawACopy(val)
```

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



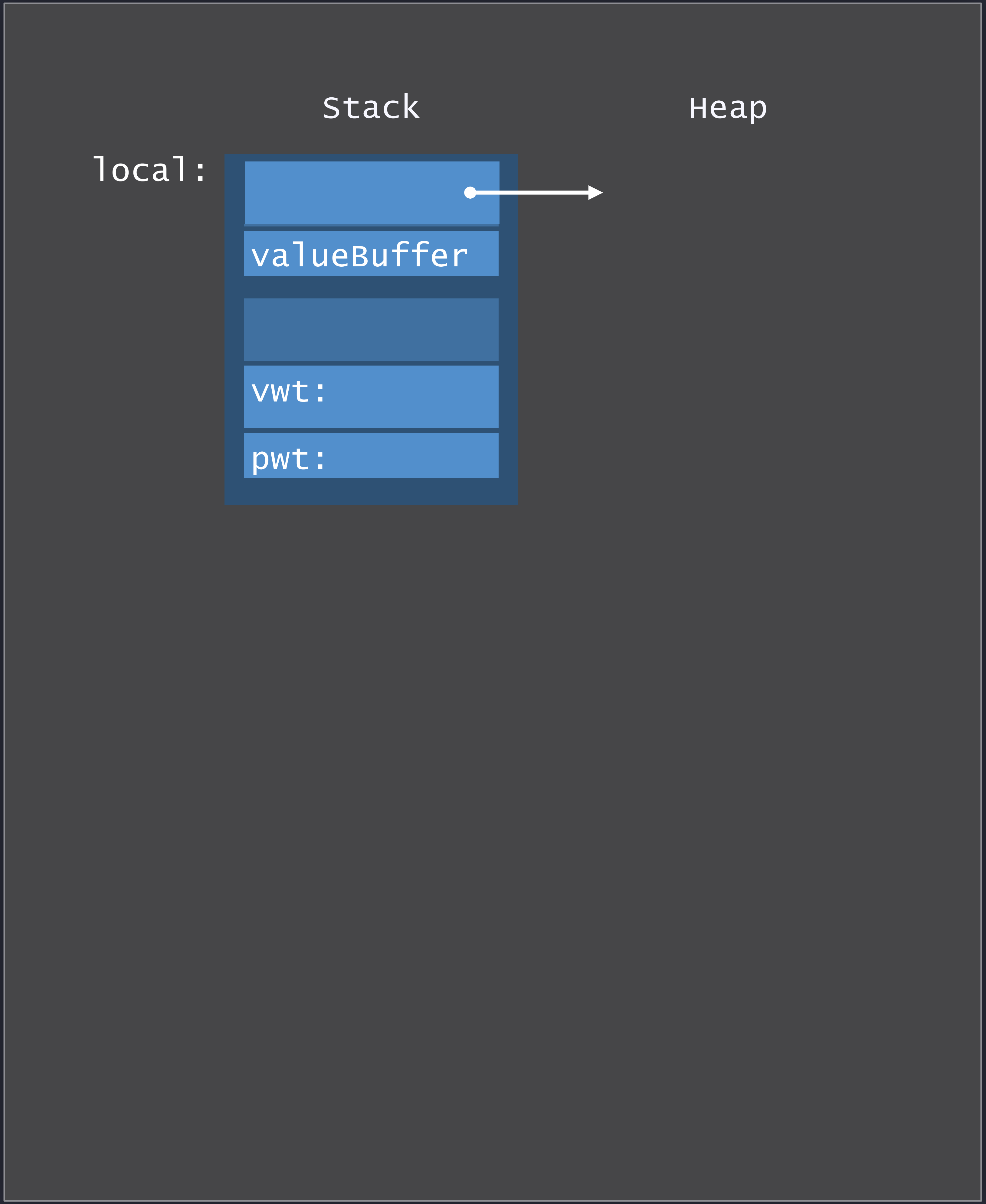
```
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
   local.draw()
}
let val : Drawable = Line()
drawACopy(val)
```

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

Performance of Protocol Types

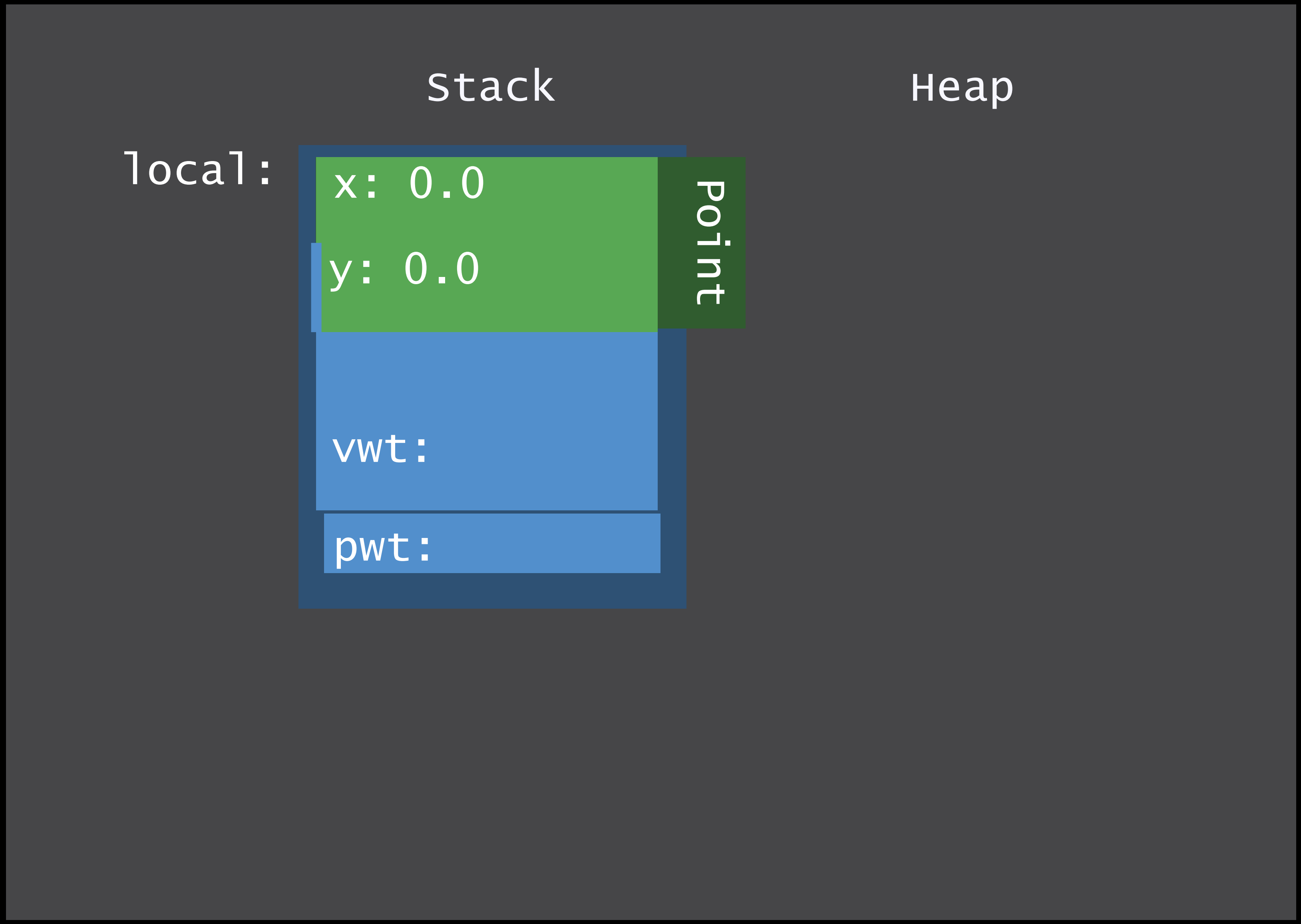
```
func drawACopy(val: ExistContDrawable) {
    var local = ExistContDrawable()
    let vwt = val.vwt
    let pwt = val.pwt
    local.type = type
    local.pwt = pwt
    vwt.allocateBufferAndCopyValue(&local, val)
    pwt.draw(vwt.projectBuffer(&local))
    vwt.destructAndDeallocateBuffer(temp)
}
```

Protocol Type—Small Value

Fits in Value Buffer: no heap allocation

No reference counting

Dynamic dispatch through Protocol Witness Table

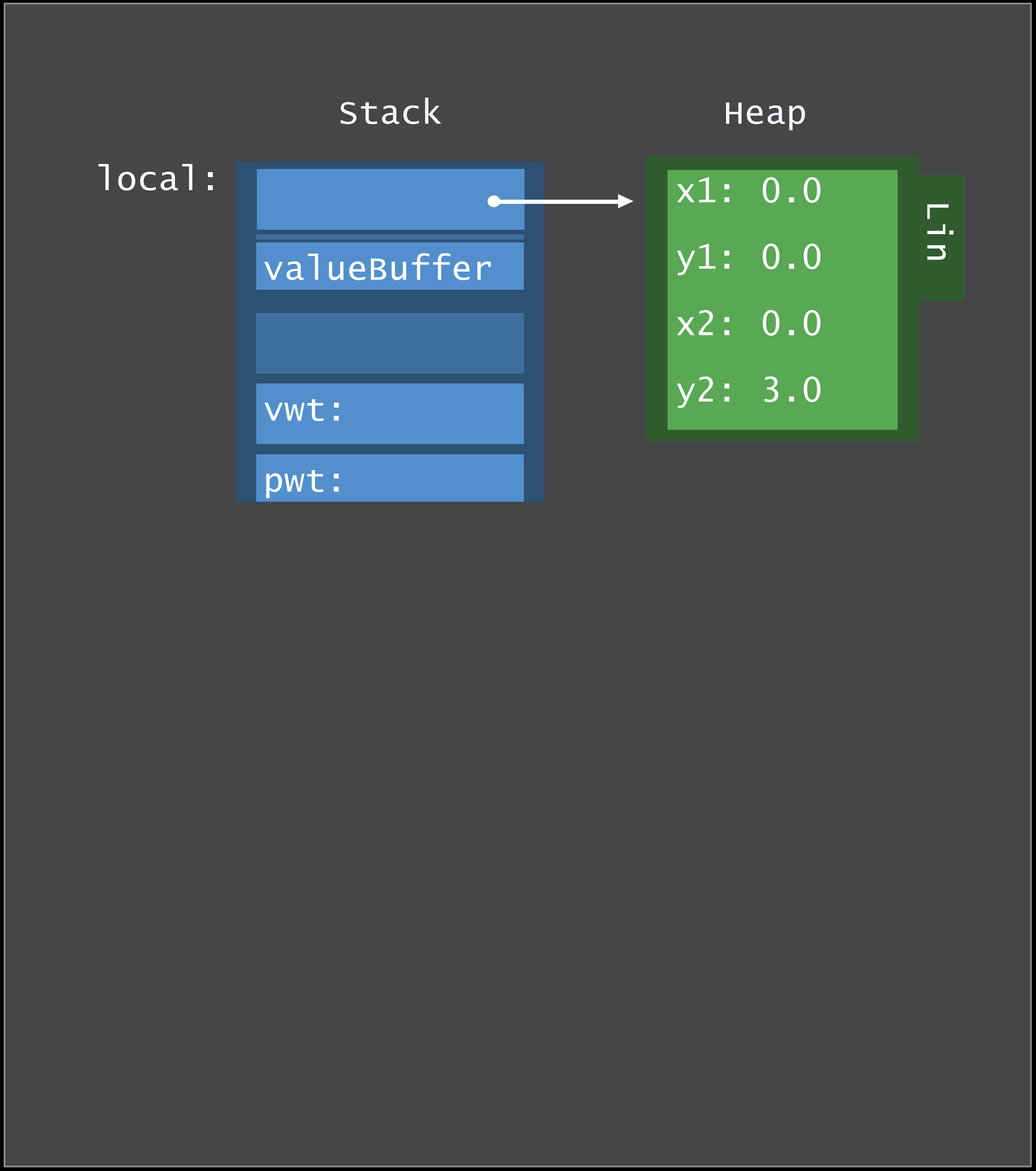


Protocol Type—Large Value

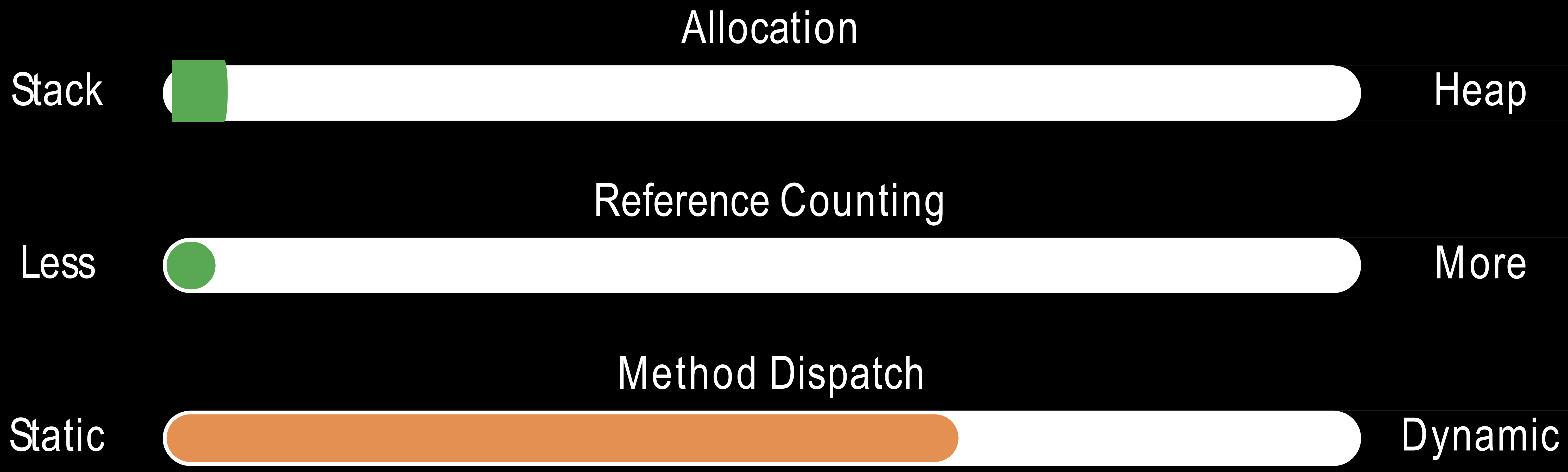
Heap allocation

No reference counting

Dynamic dispatch through Protocol Witness Table



Protocol Type—Small Value



Allocation

Stack Heap

Reference Counting

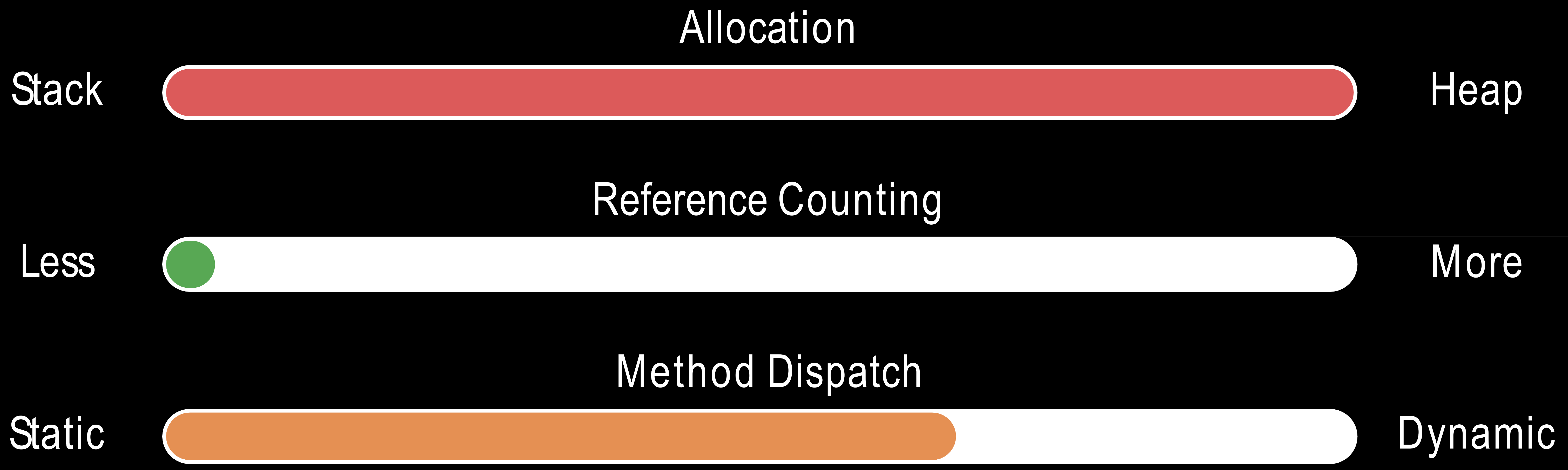
Less More

Method Dispatch

Static Dynamic

Protocol Type—Large Value

Expensive heap allocation on copying

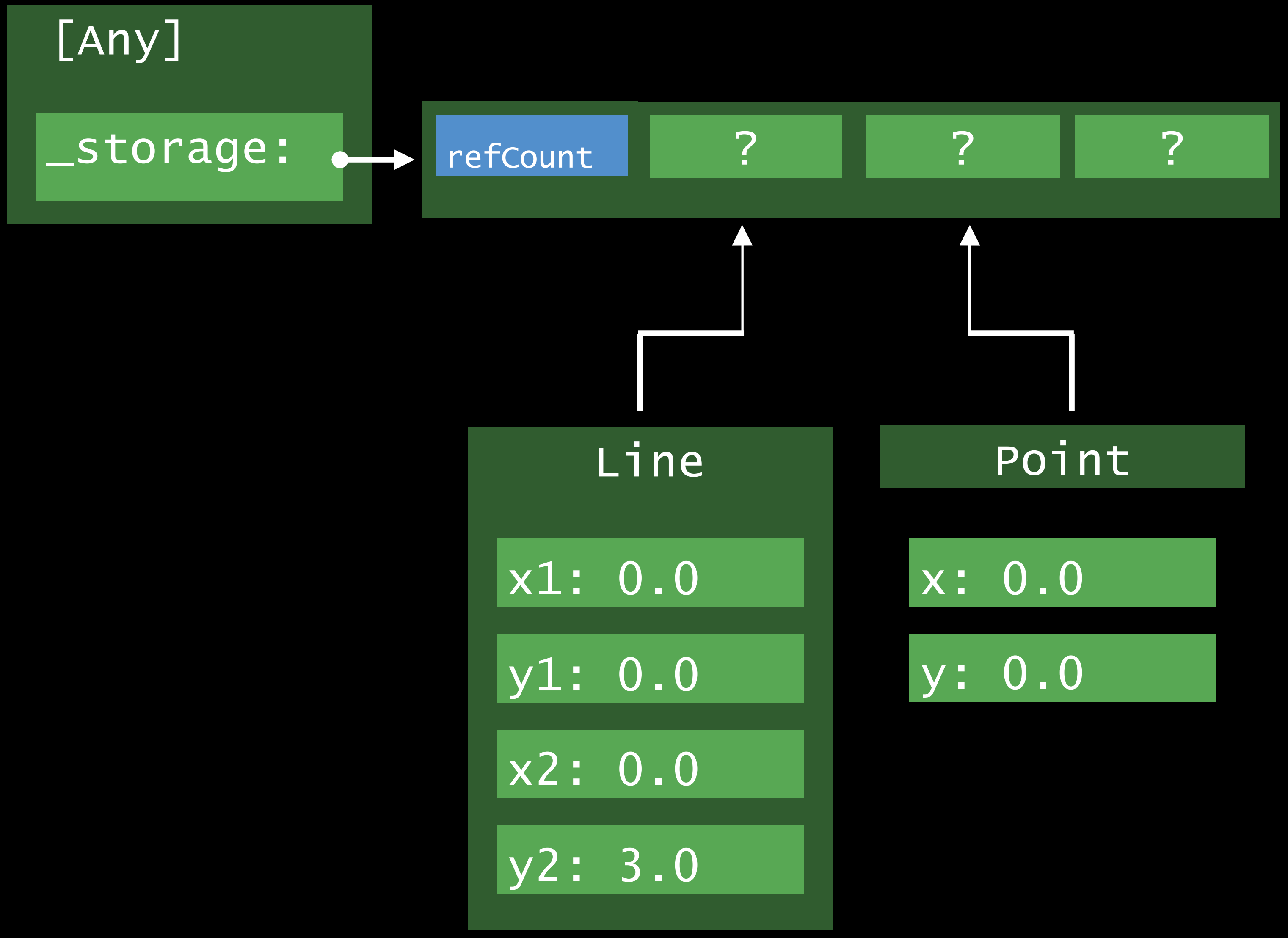


Question 1

```
var anys: [Any]
```

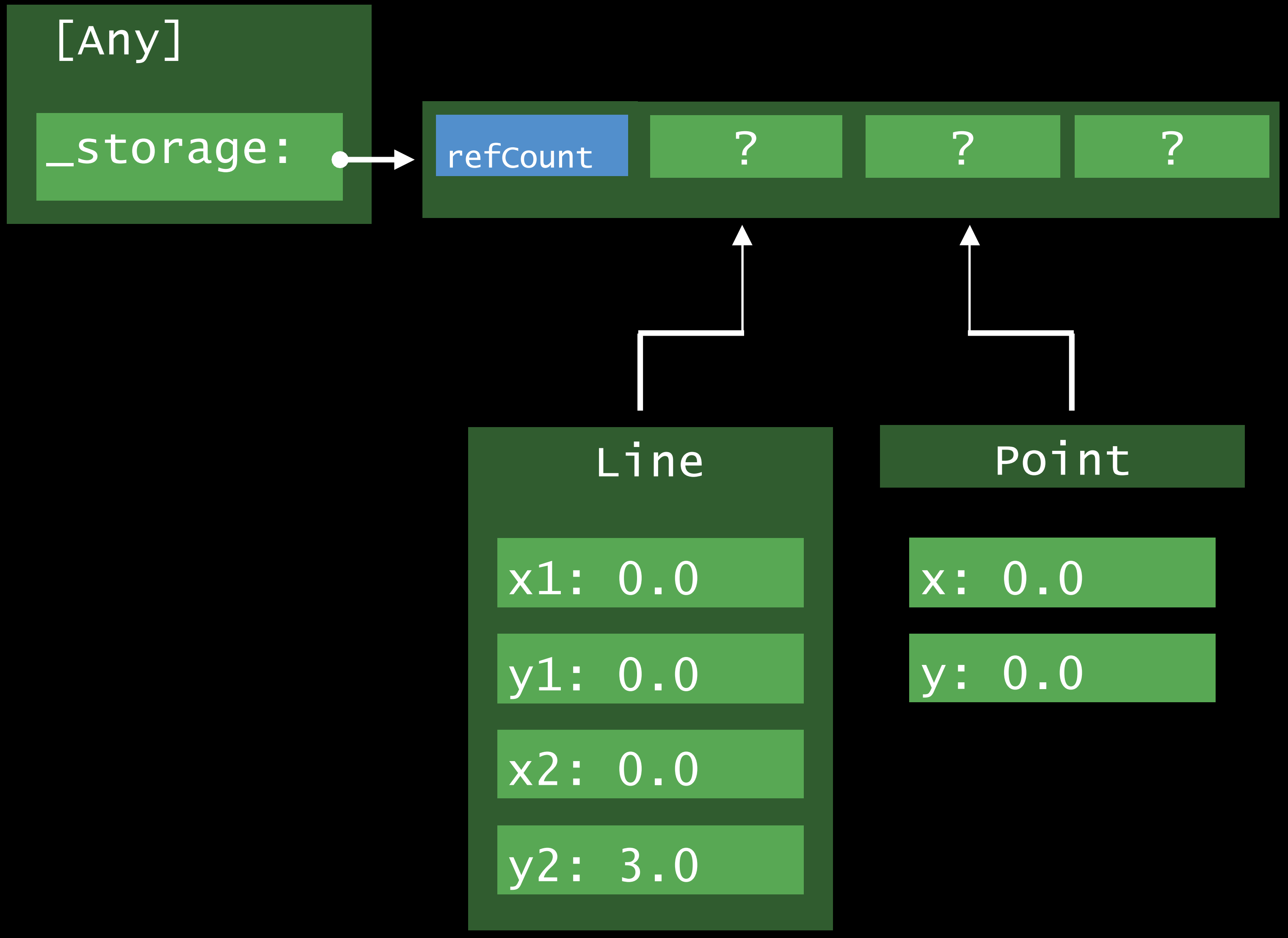
Question 1

```
var anys: [Any]
```



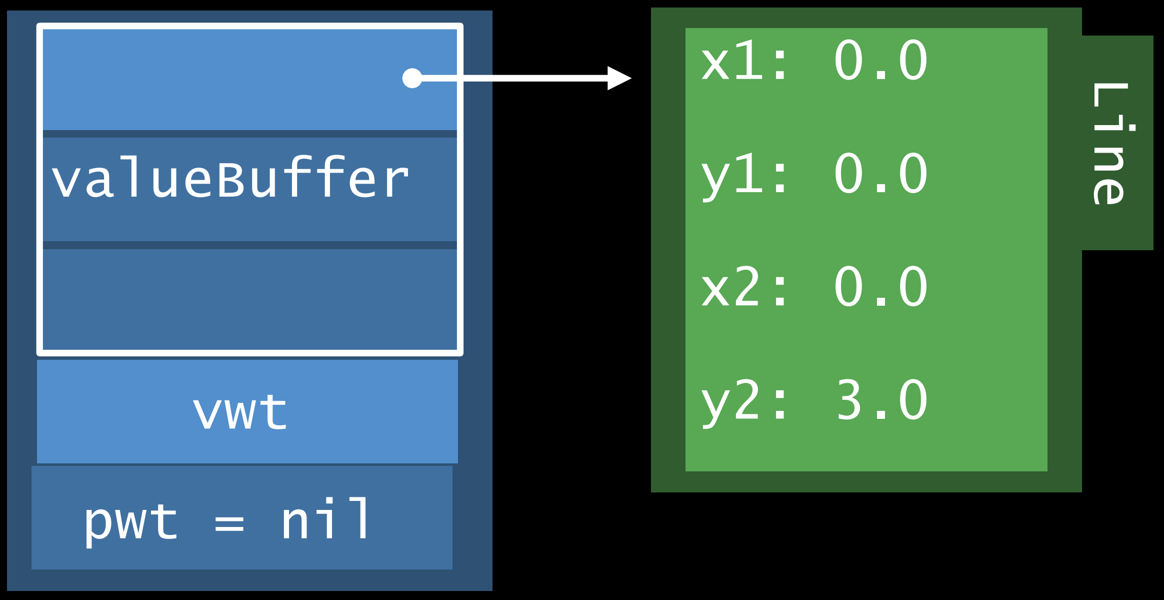
Question 1

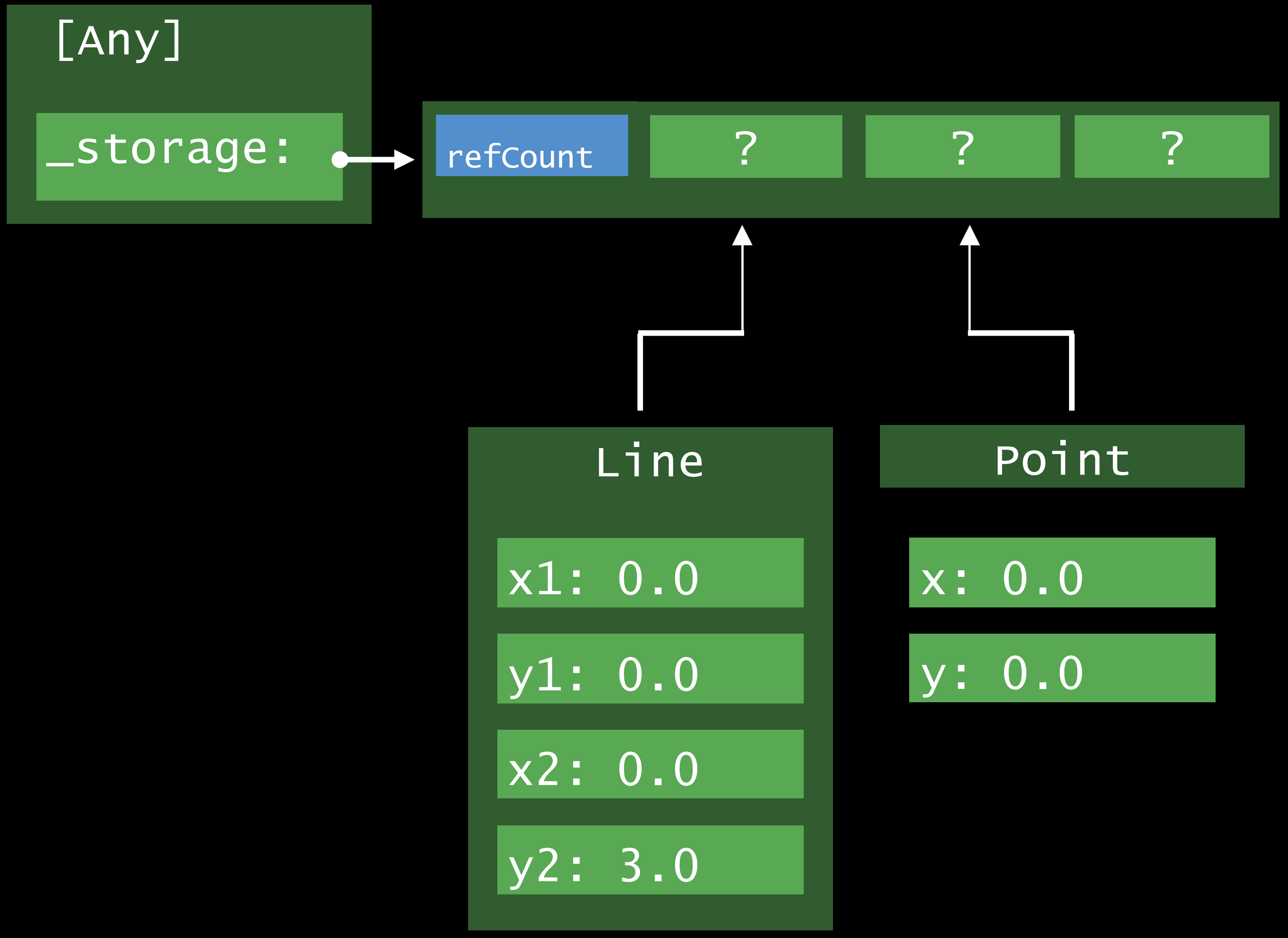
```
var anys: [Any]
Any = Protocol< >
```



Question 1

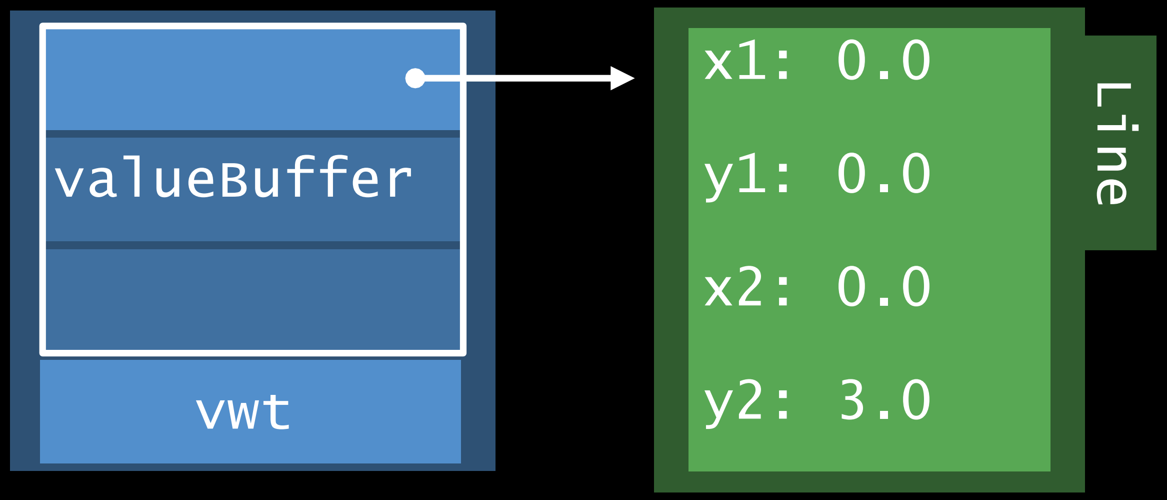
```
var anys: [Any]
Any = Protocol< >
```

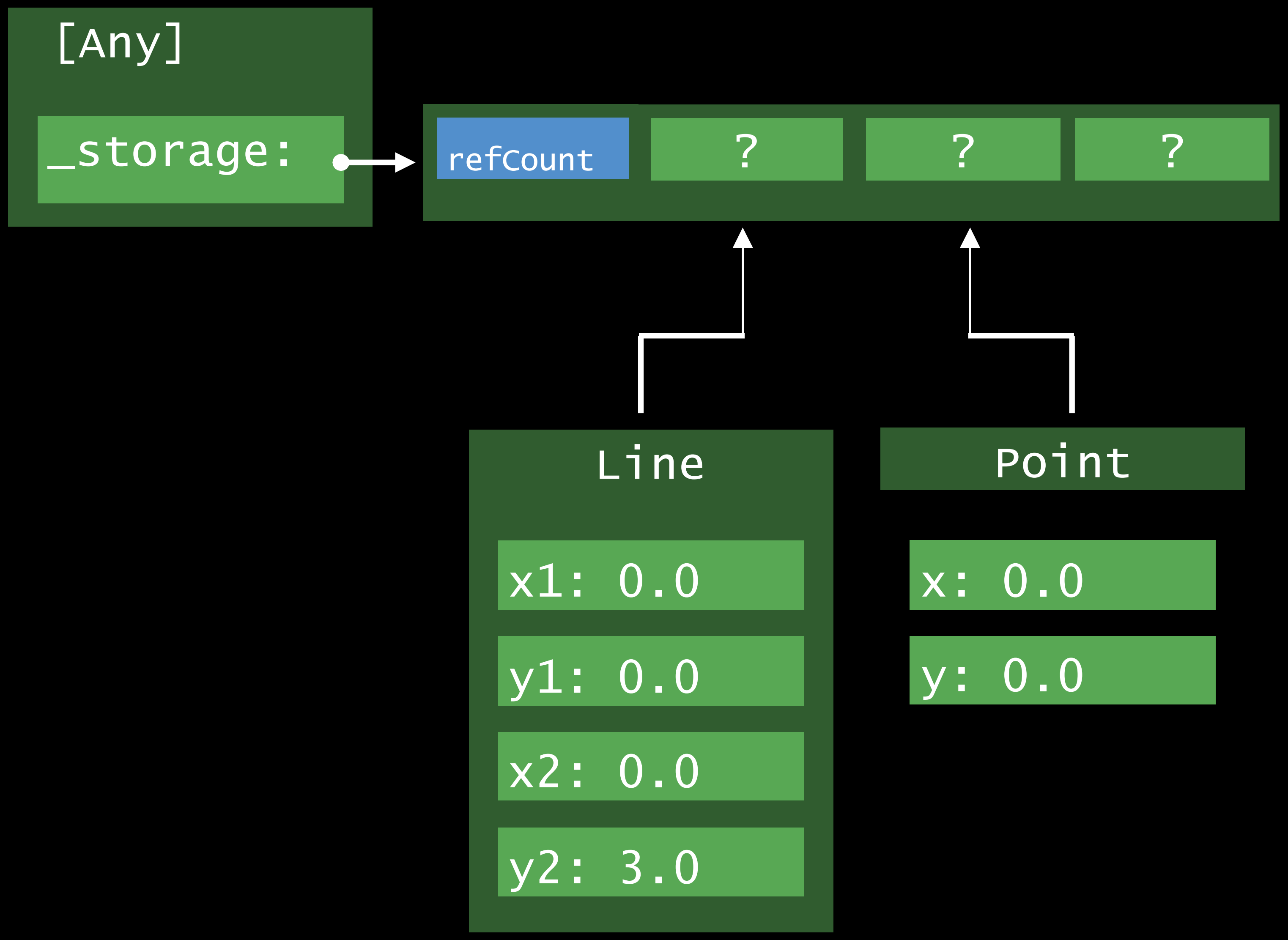




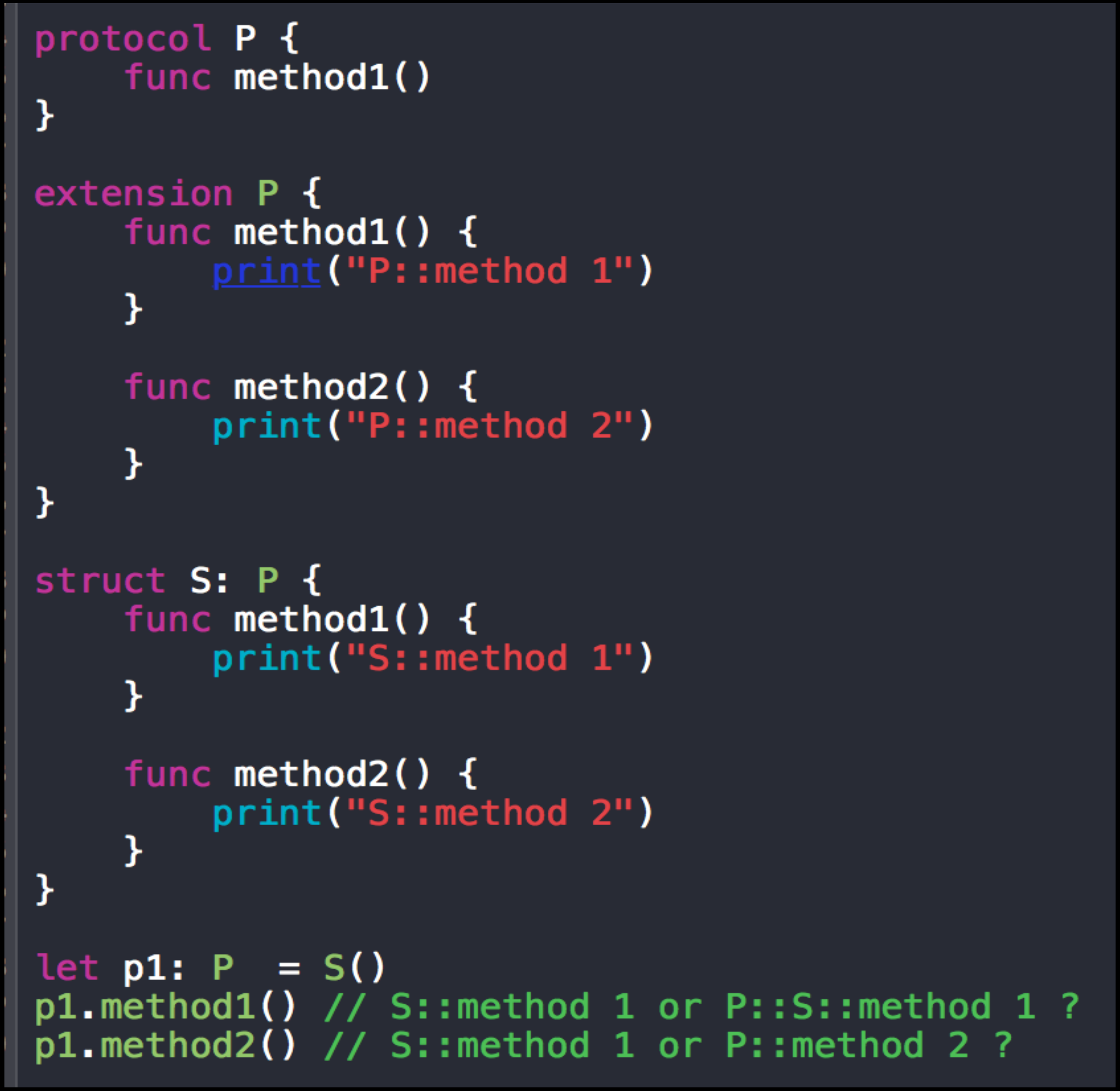
Question 1

```
var anys: [Any]
Any = Protocol< >
```



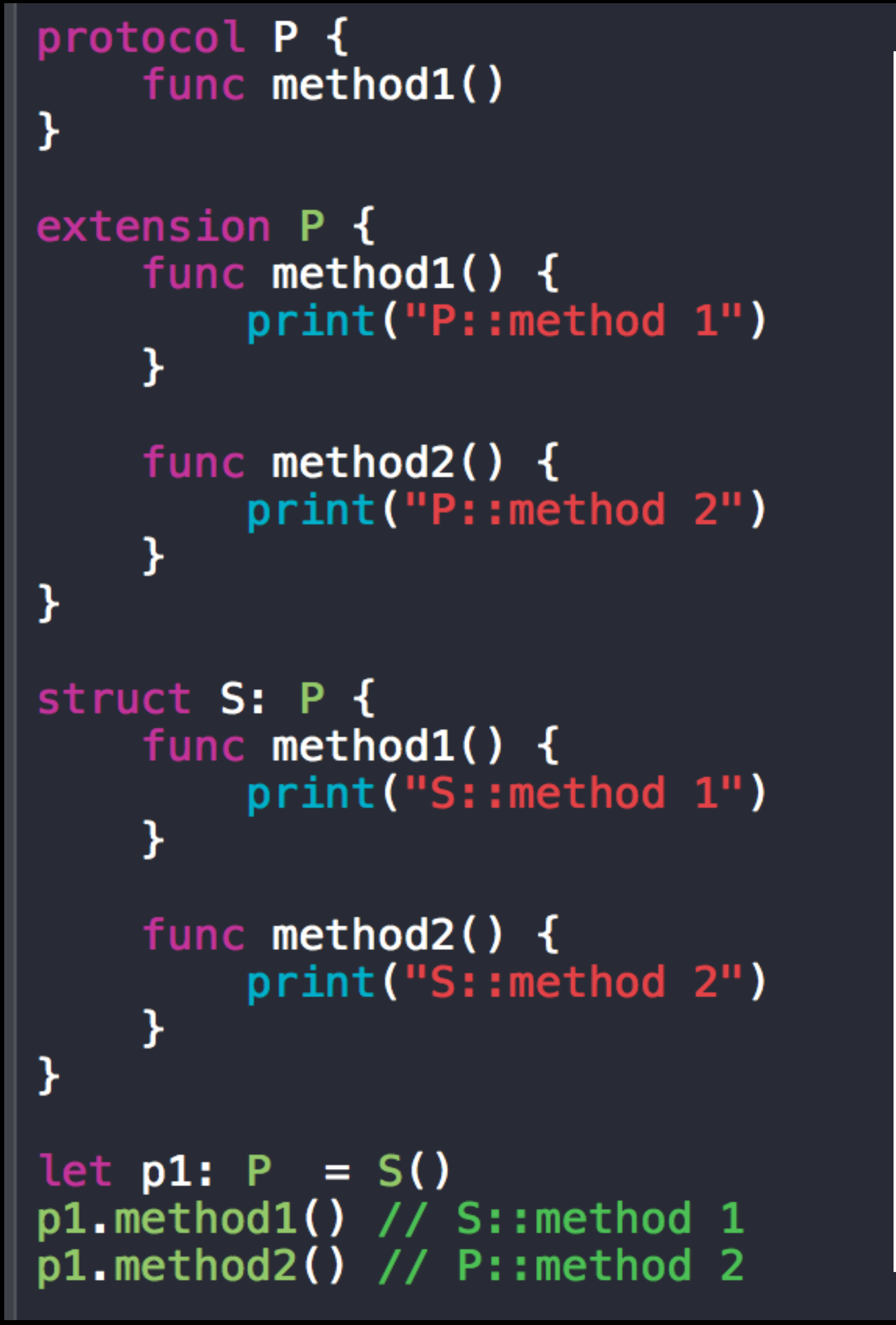


Question 2

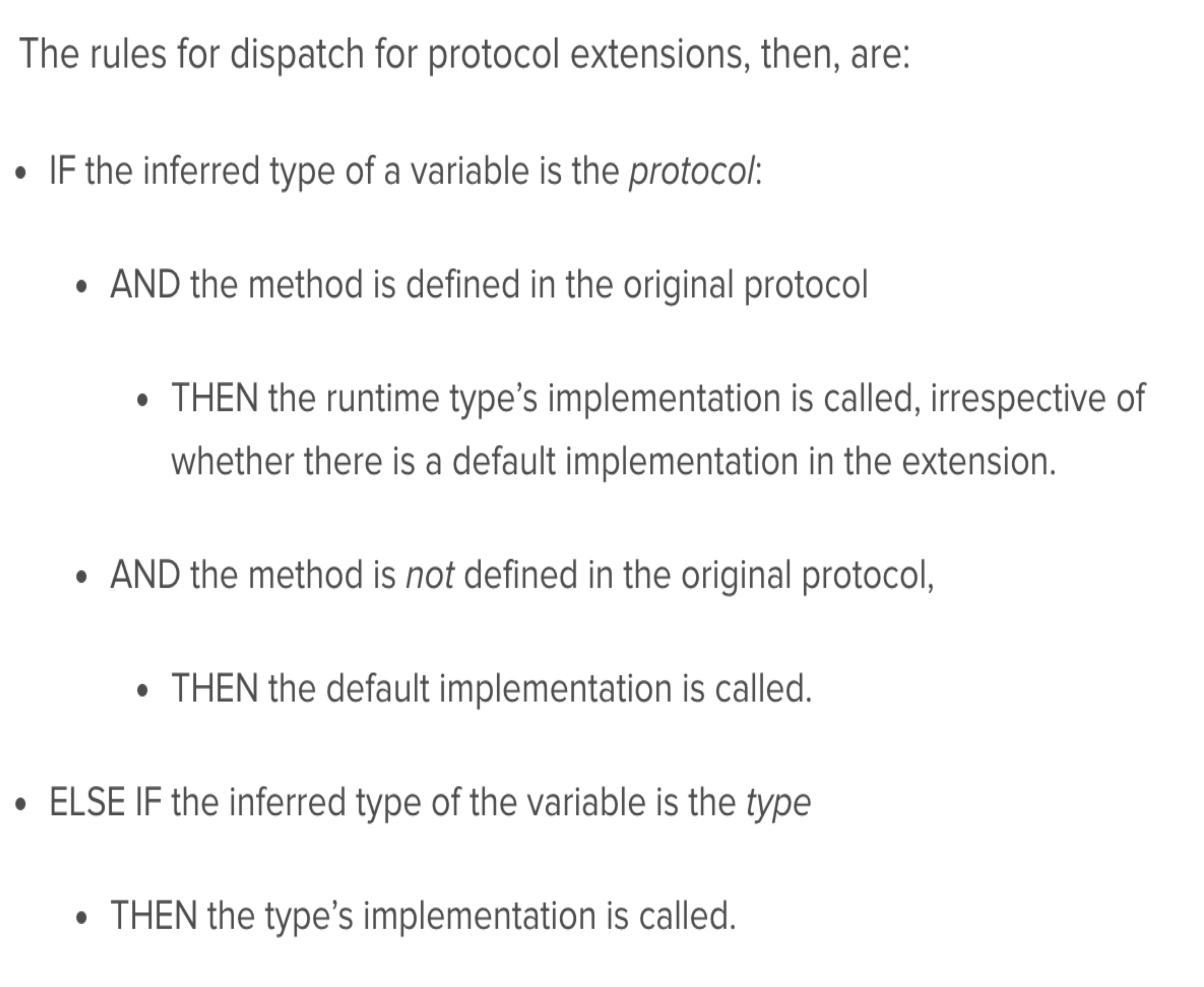


```
protocol P {
    func method1()
}

extension P {
    func method1() {
        print("P::method 1")
    }

    func method2() {
        print("P::method 2")
    }
}

struct S: P {
    func method1() {
        print("S::method 1")
    }

    func method2() {
        print("S::method 2")
    }
}

let p1: P  = S()
p1.method1() // S::method 1 or P::S::method 1 ?
p1.method2() // S::method 1 or P::method 2 ?
```

Question 2



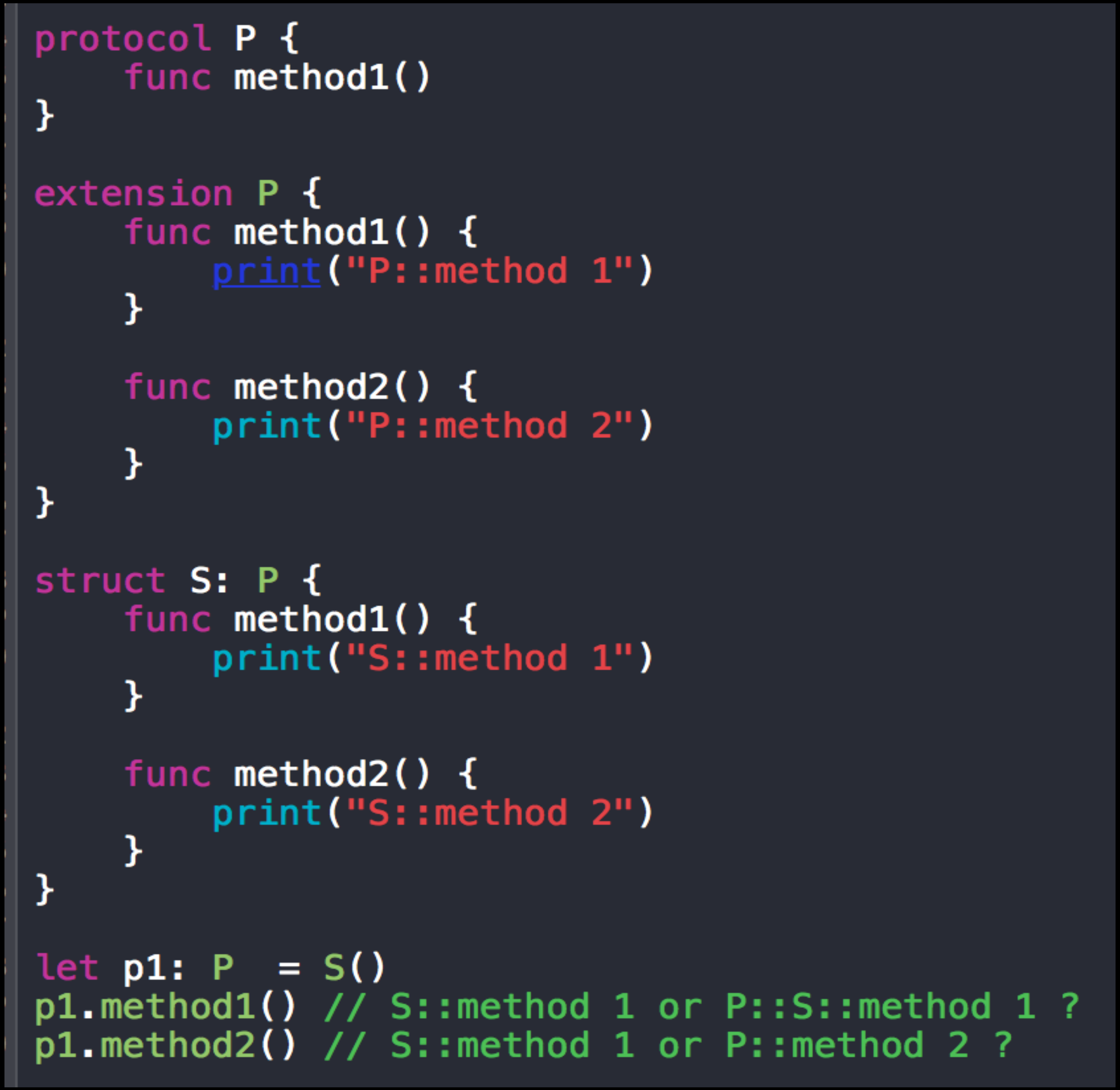
```
protocol P {
    func method1()
}

extension P {
    func method1() {
        print("P::method 1")
    }

    func method2() {
        print("P::method 2")
    }
}

struct S: P {
    func method1() {
        print("S::method 1")
    }

    func method2() {
        print("S::method 2")
    }
}

let p1: P  = S()
p1.method1() // S::method 1
p1.method2() // P::method 2
```



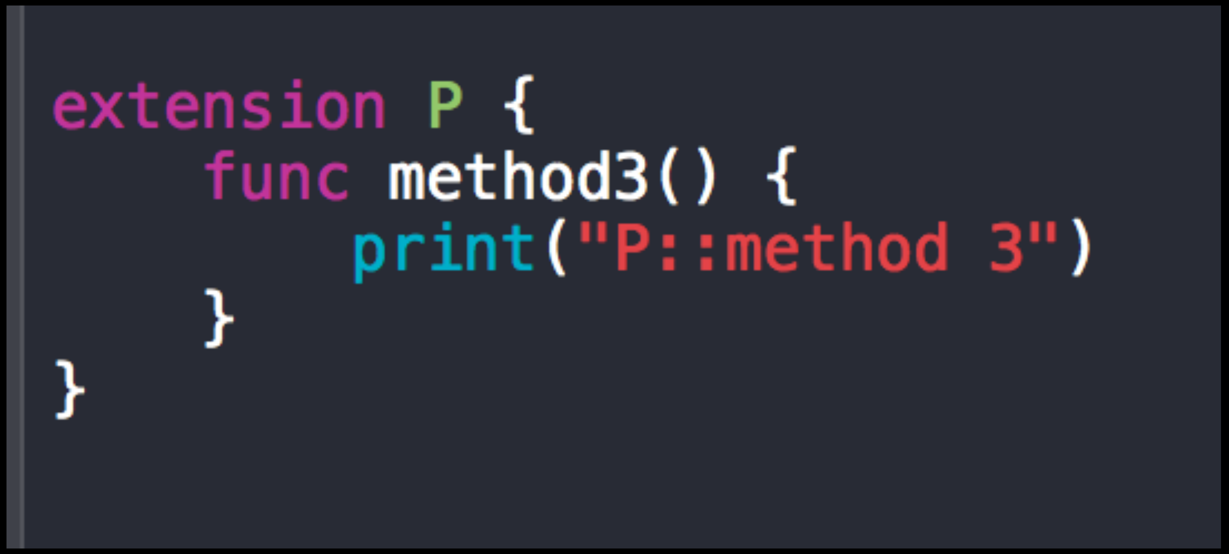
The rules for dispatch for protocol extensions, then, are:

- IF the inferred type of a variable is the *protocol*:
  - AND the method is defined in the original protocol
    - THEN the runtime type's implementation is called, irrespective of whether there is a default implementation in the extension.
  - AND the method is *not* defined in the original protocol,
    - THEN the default implementation is called.
- ELSE IF the inferred type of the variable is the *type*
  - THEN the type's implementation is called.

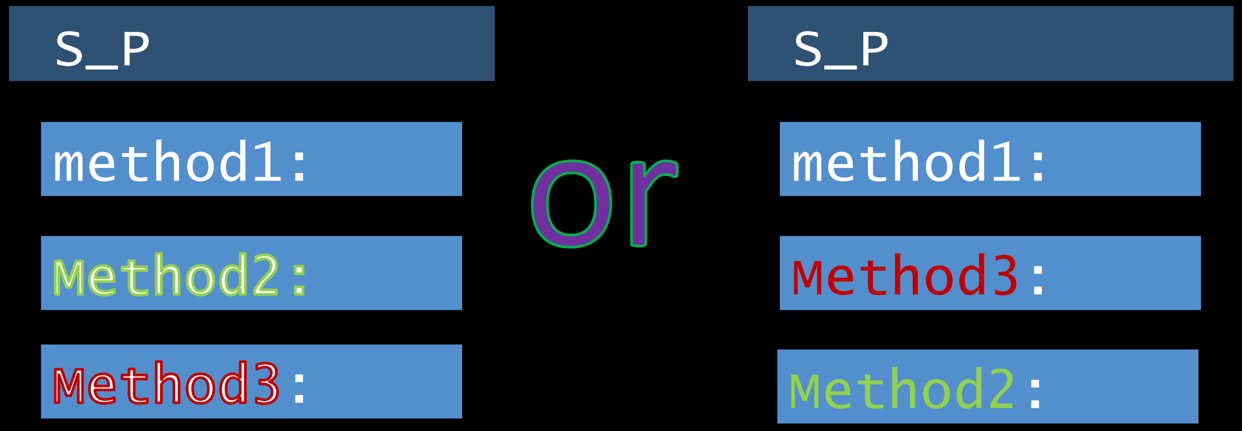
Question 2 static dispatch or dynamic dispatch



```
protocol P {
    func method1()
}

extension P {
    func method1() {
        print("P::method 1")
    }

    func method2() {
        print("P::method 2")
    }
}

struct S: P {
    func method1() {
        print("S::method 1")
    }

    func method2() {
        print("S::method 2")
    }
}

let p1: P  = S()
p1.method1() // S::method 1 or P::S::method 1 ?
p1.method2() // S::method 1 or P::method 2 ?
```



```
extension P {
    func method3() {
        print("P::method 3")
    }
}
```



S_P

method1:

Method2:

Method3:

or

S_P

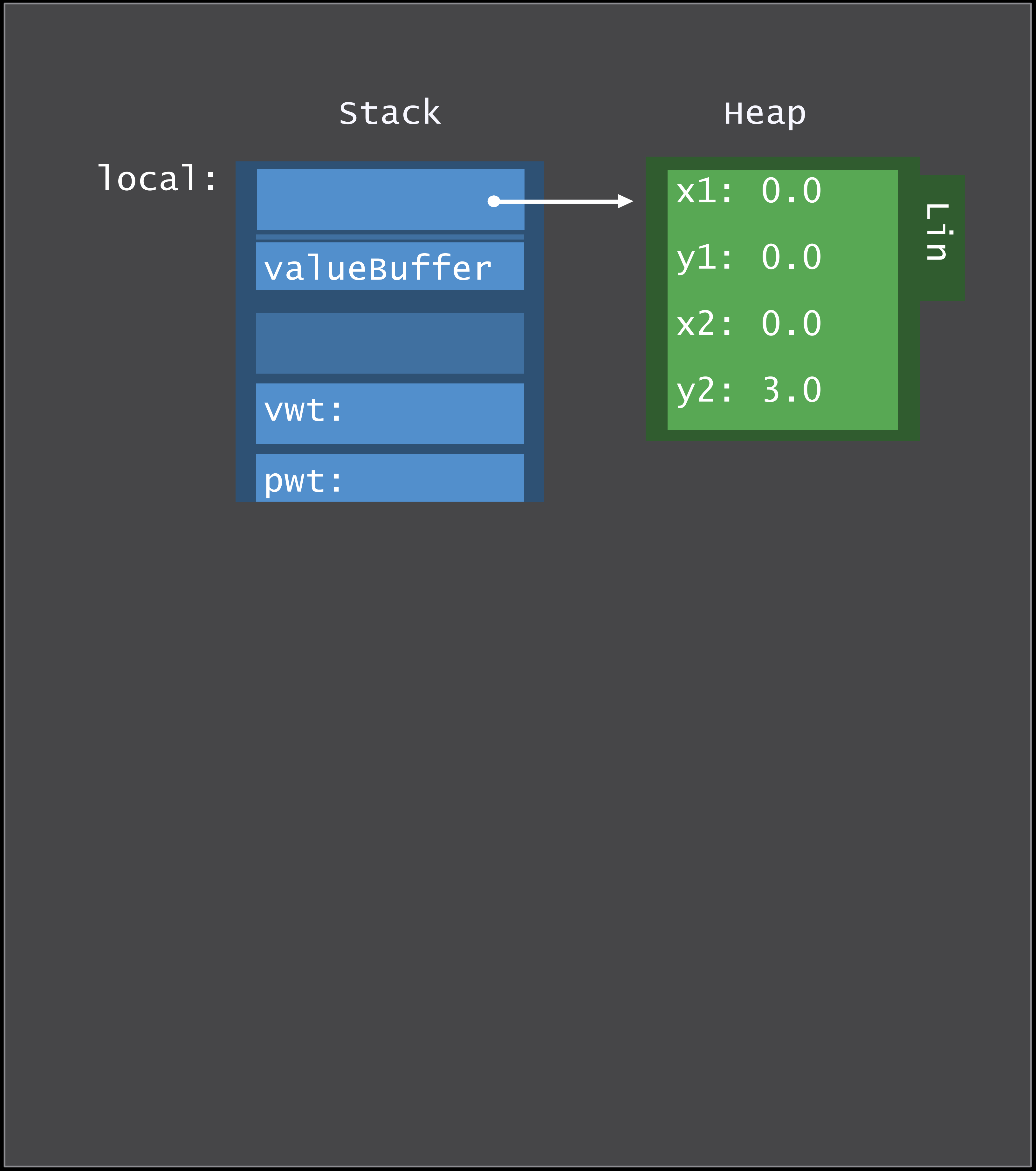
method1:

Method3:

Method2:

Summary—Protocol Types

1. Indirection through Witness Tables and Existential Container
2. Dynamic dispatch through Protocol Witness Table
3. Access value through Value Witness Table
4. Copying of large values causes heap allocation



Generic Code

```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)
```

Implementation of Generic Methods

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

drawACopy(Point(…))
```

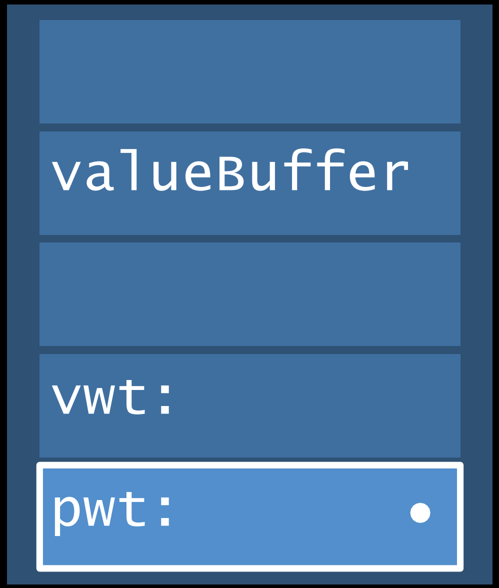
Implementation of Generic Methods

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

drawACopy(Point(…))
```

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

drawACopy(Point(…))
```

One shared implementation

Uses Protocol/Value Witness Table



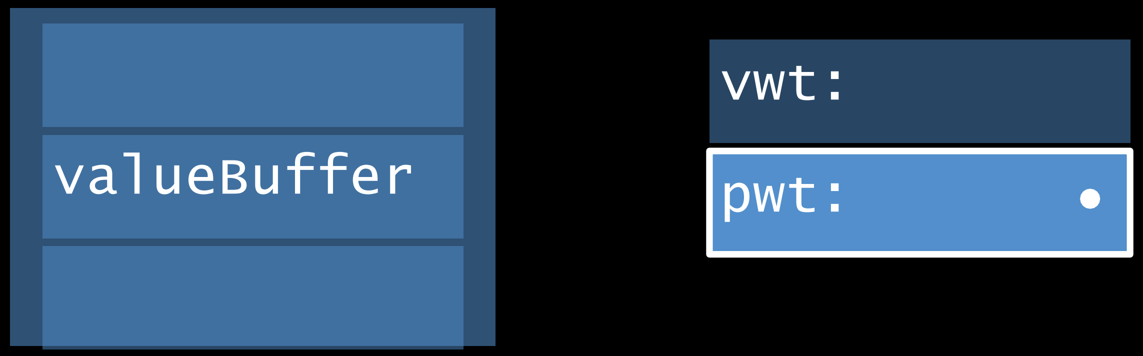
Implementation of Generic Methods

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

drawACopy(Point(…))
```

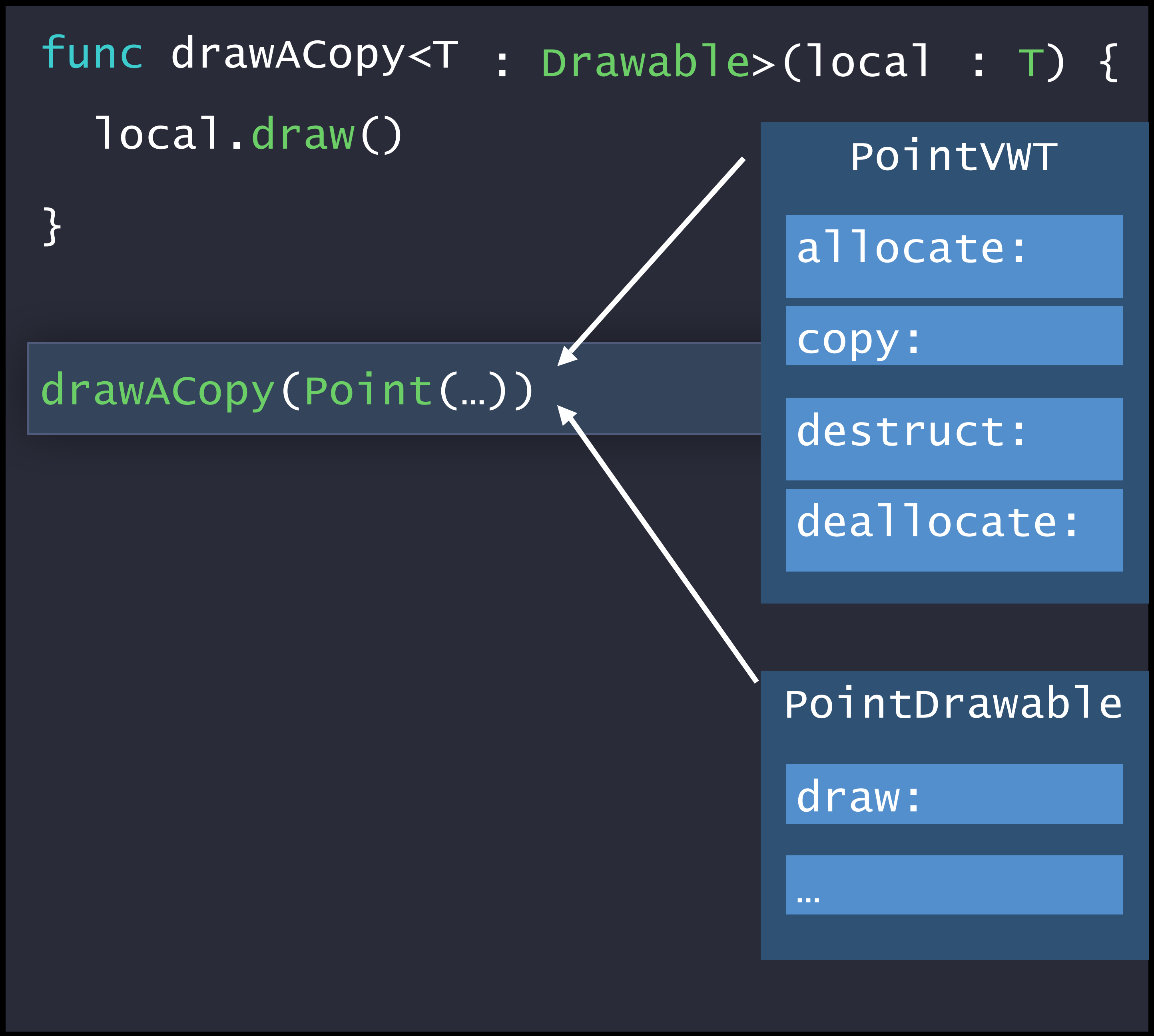
```
func drawACopy(local : Drawable) {
  local.draw()
}

drawACopy(Point(…))
```

One shared implementation

Uses Protocol/Value Witness Table



Implementation of GenericMethods



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

```
drawACopy(Point(…))
```

PointVWT

allocate:

copy:

destruct:

deallocate:

PointDrawable

draw:

…

One shared implementation

Uses Protocol/Value Witness Table

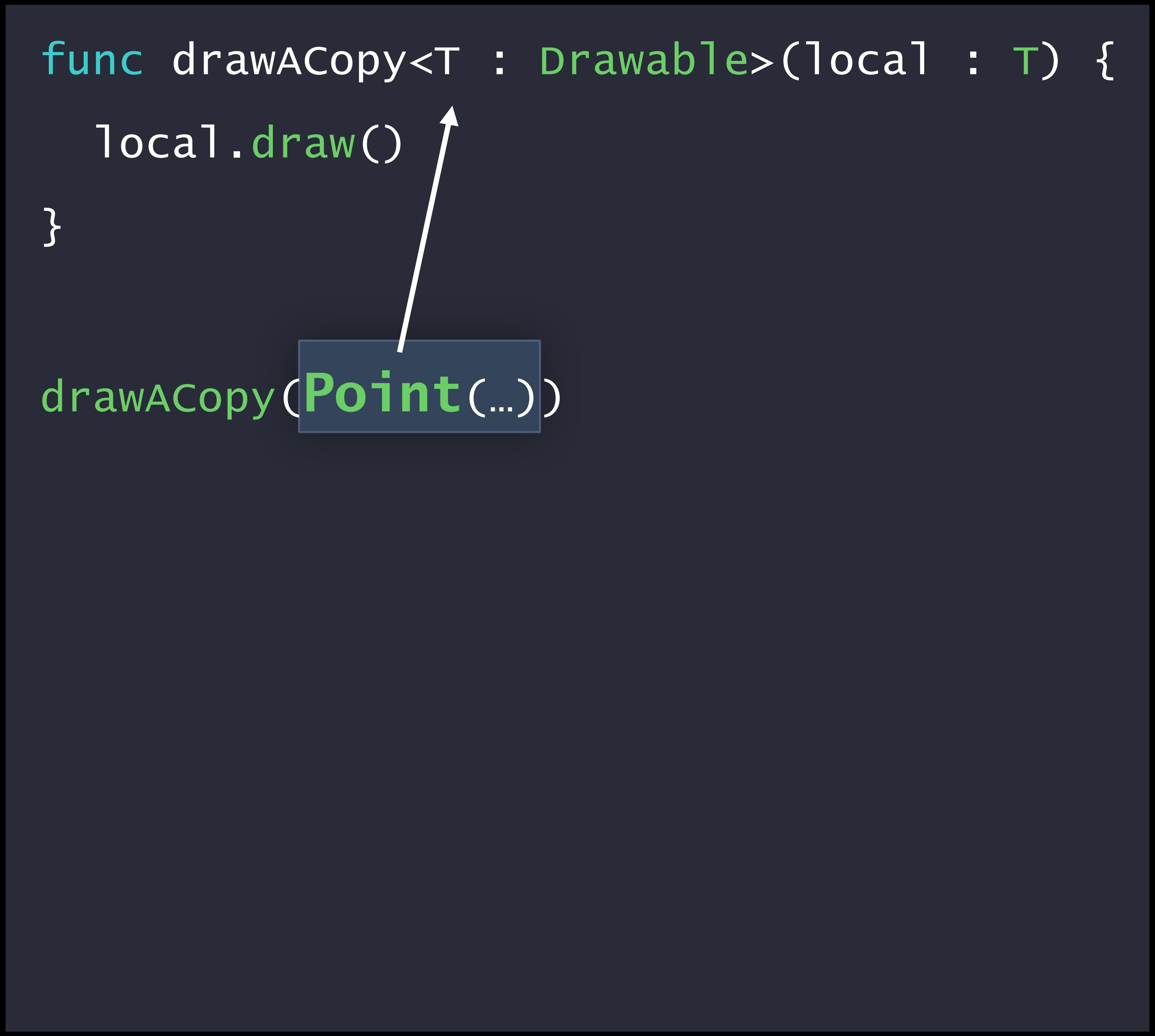
One type per call context: passes tables

Faster?

Specialization of Generics

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


drawACopy(Point(…))
```

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

Specialization of Generics

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

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 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

When Does Specialization Happen?

Infer type at call-site

Definition must be available

main.swift

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

Whole Module Optimization

Increases optimization opportunity

Point.swift

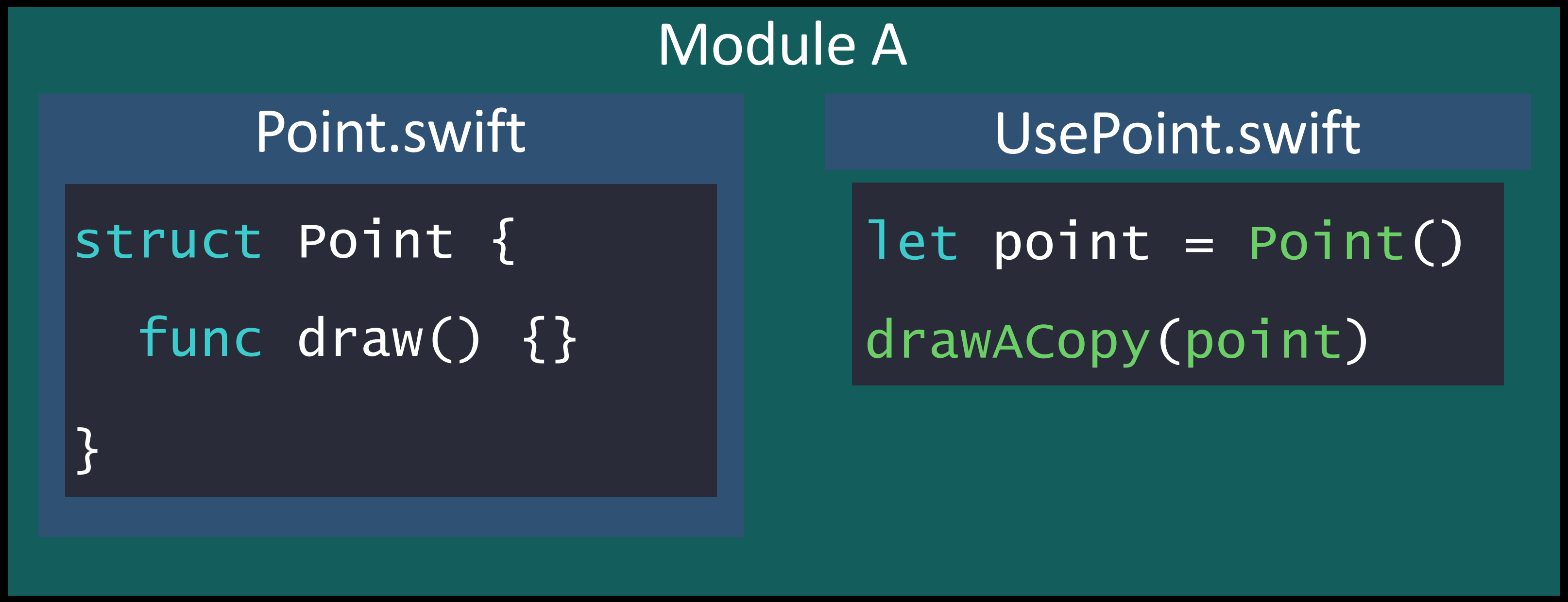
```swift
struct Point {
  func draw() {}
}
```

UsePoint.swift

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

Whole Module Optimization

Increases optimization opportunity



Module A

Point.swift

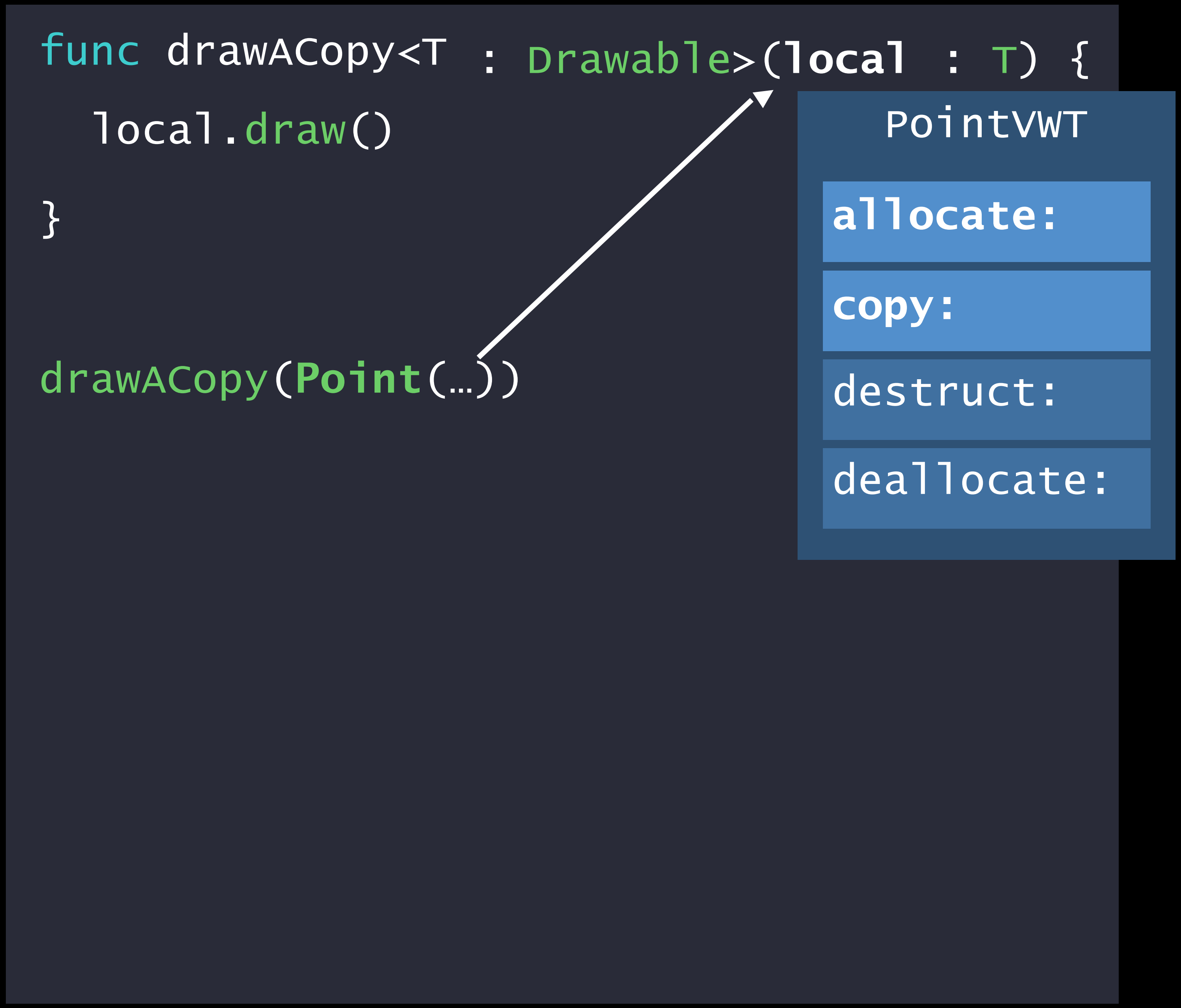
```
struct Point {
  func draw() {}
}
```

UsePoint.swift

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

Performance of Generic Code

Unspecialized



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

drawACopy(Point(…))
```

PointVWT

allocate:

copy:

destruct:

deallocate:

Specialized

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

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

Swift Generics vs C++ template

Is there any problem with this code?

```
func add<T>(a : T, b : T) -> T {
   return a + b

}
```

Swift Generics vs C++ template

```
 func add<T>(a : T, b : T) -> T {
    return a + b
/*No '+' candidates produce the expected
contextual result type 'T'*/

 }
```

```
Template <typename T>

T add (T a, T b) {

  return a + b

}
```

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

  local.draw()

}
```

Swift Generics vs C++ template

```
func add<T>(a : T, b : T) -> T {
   return a + b
/*No '+' candidates produce the expected
contextual result type 'T'*/
 }
```

```
Template <typename T>

T add (T a, T b) {

  return a + b

}
```

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

  local.draw()

}
```

1. C++ template always have a specialization for each type
2. C++ template has no shared implementation version
3. C++ template has explicit specialization
4. C++ template has template meta-programming
5. C++ template is more complex

Question

```
func drawACopy<T>(local : T) {
    …
}


drawACopy(Point(…))
```

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


drawACopy(Point(…))
```

Question

```
func drawACopy<T>(local : T) {
    …
}

drawACopy(Point(…))
```

```
func drawACopy<T : Any>(local : T) {
    …
}

drawACopy(Point(…))
```

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

drawACopy(Point(…))
```

Question

```swift
func drawACopy<T>(local : T) {
    …
}


drawACopy(Point(…))
```

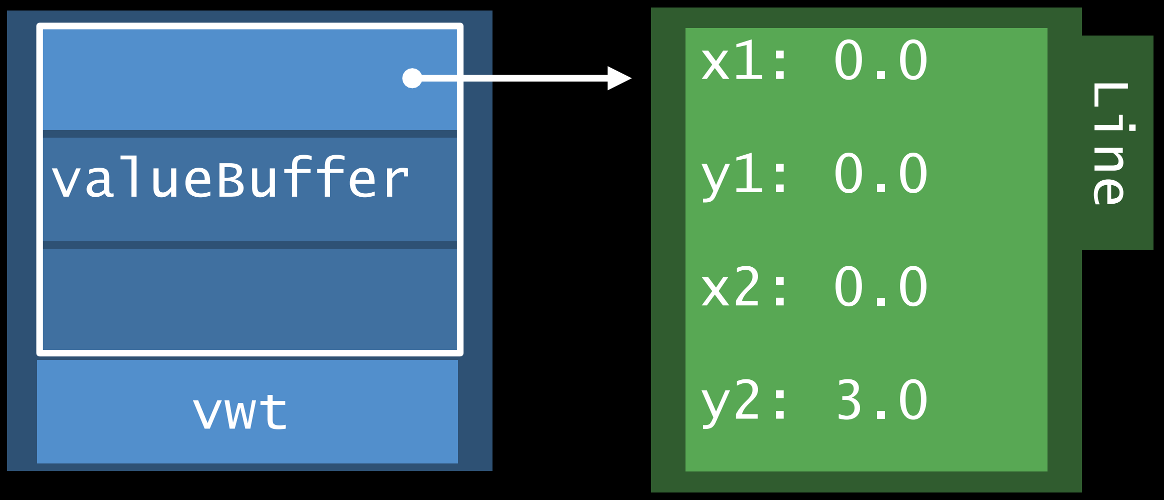
```swift
func drawACopy<T : Any>(local : T) {
    …
}


drawACopy(Point(…))
```

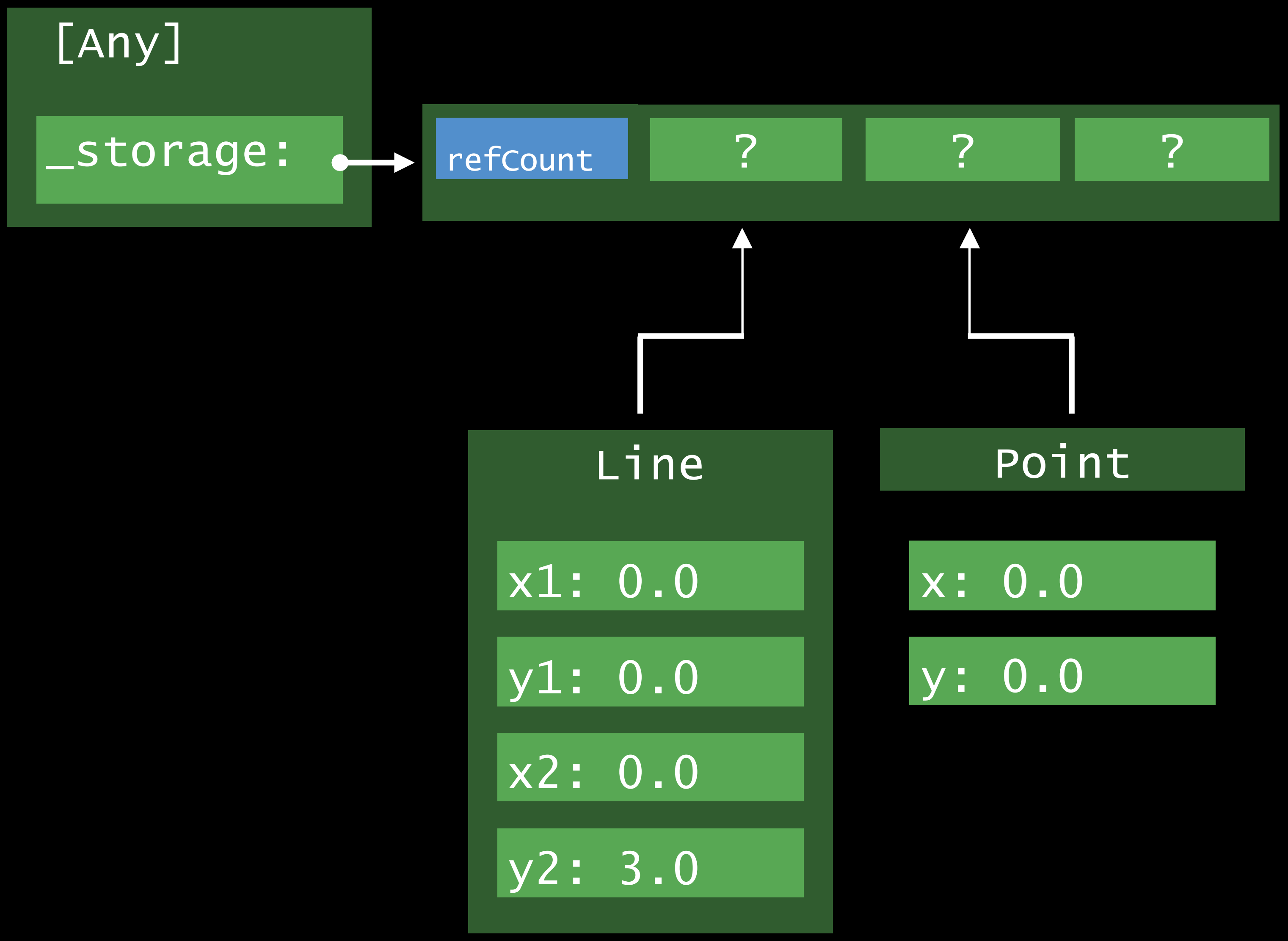
```swift
func drawACopy<T: Drawable>(local : T) {
    …
}


drawACopy(Point(…))
```

```swift
func drawACopy(local : Any) {
    …
}


drawACopy(Point(…))
```

Question

```
var anys: [Any]
Any = Protocol< >
```





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------Copy-on-Write

Performance: Swift ~= C++ > Objective-C