Developer Tools #WWDC16

Understanding Swift Performance

Session 416

Kyle Macomber Software Engineer
Arnold Schwaighofer Swift Performance Engineer

struct

class

enum

struct

class

enum

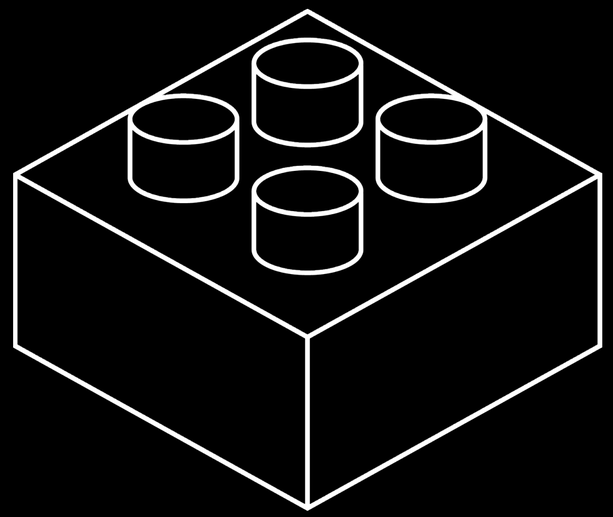
inheritance

generics

protocols

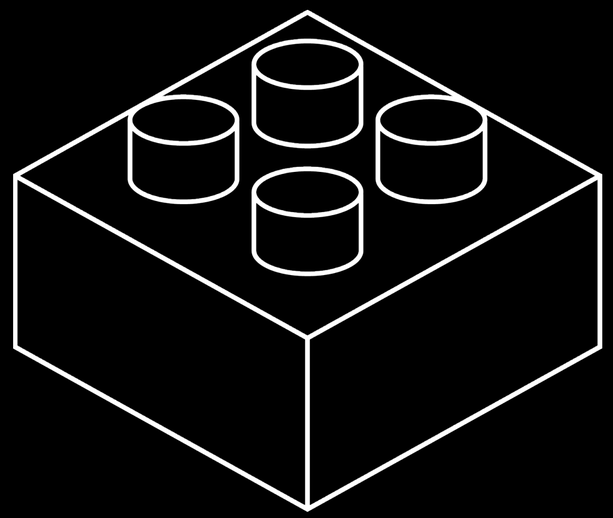
Choosing the Right Abstraction Mechanism

Choosing the Right Abstraction Mechanism

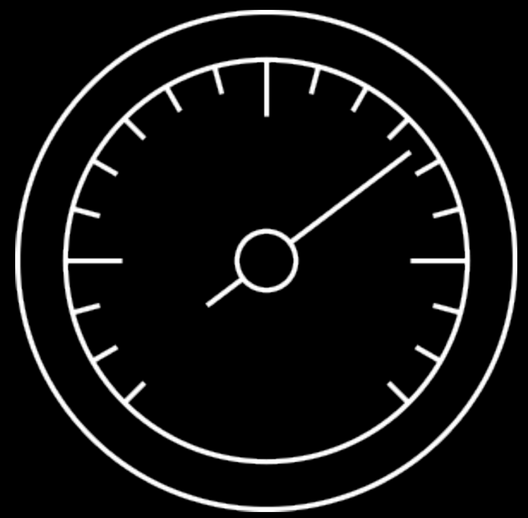


Modeling

Choosing the Right Abstraction Mechanism



Modeling



Performance

Choosing the Right Abstraction Mechanism

Modeling

Protocol and Value Oriented Programming in UIKit Apps	Presidio	Friday 4:00PM
Protocol-Oriented Programming in Swift		WWDC 2015
Building Better Apps with Value Types in Swift		WWDC 2015

Understand the implementation to understand performance

Agenda

Agenda

Allocation

Reference counting

Method dispatch

Agenda

Allocation

Reference counting

Method dispatch

Protocol types

Generic code

Dimensions of Performance

Dimensions of Performance

Dimensions of Performance

Dimensions of Performance

Dimensions of Performance

Allocation

Stack — Heap

Reference Counting

Less — More

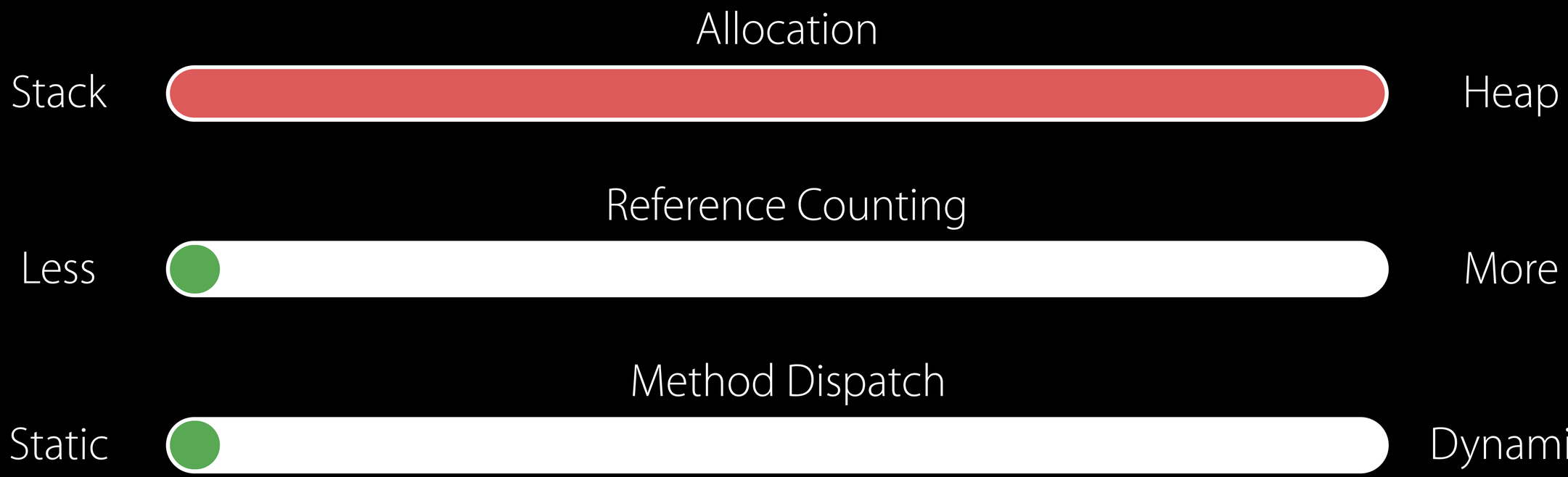
Method Dispatch

Static — Dynamic

Dimensions of Performance

Dimensions of Performance

Dimensions of Performance

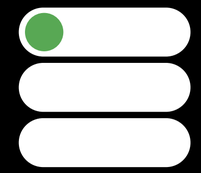


Dimensions of Performance

Allocation

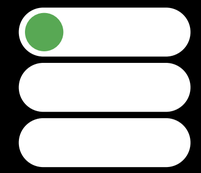
Allocation

Stack



Allocation

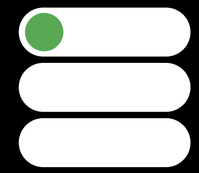
Stack



Decrement stack pointer to allocate

Allocation

Stack

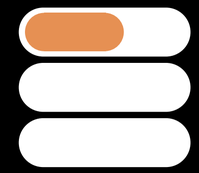


Decrement stack pointer to allocate

Increment stack pointer to deallocate

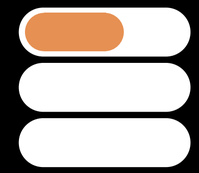
Allocation

Heap



Allocation

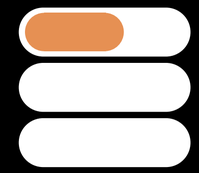
Heap



Advanced data structure

Allocation

Heap

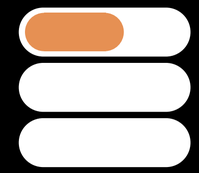


Advanced data structure

Search for unused block of memory to allocate

Allocation

Heap



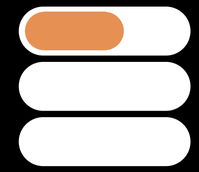
Advanced data structure

Search for unused block of memory to allocate

Reinsert block of memory to deallocate

Allocation

Heap



Advanced data structure

Search for unused block of memory to allocate

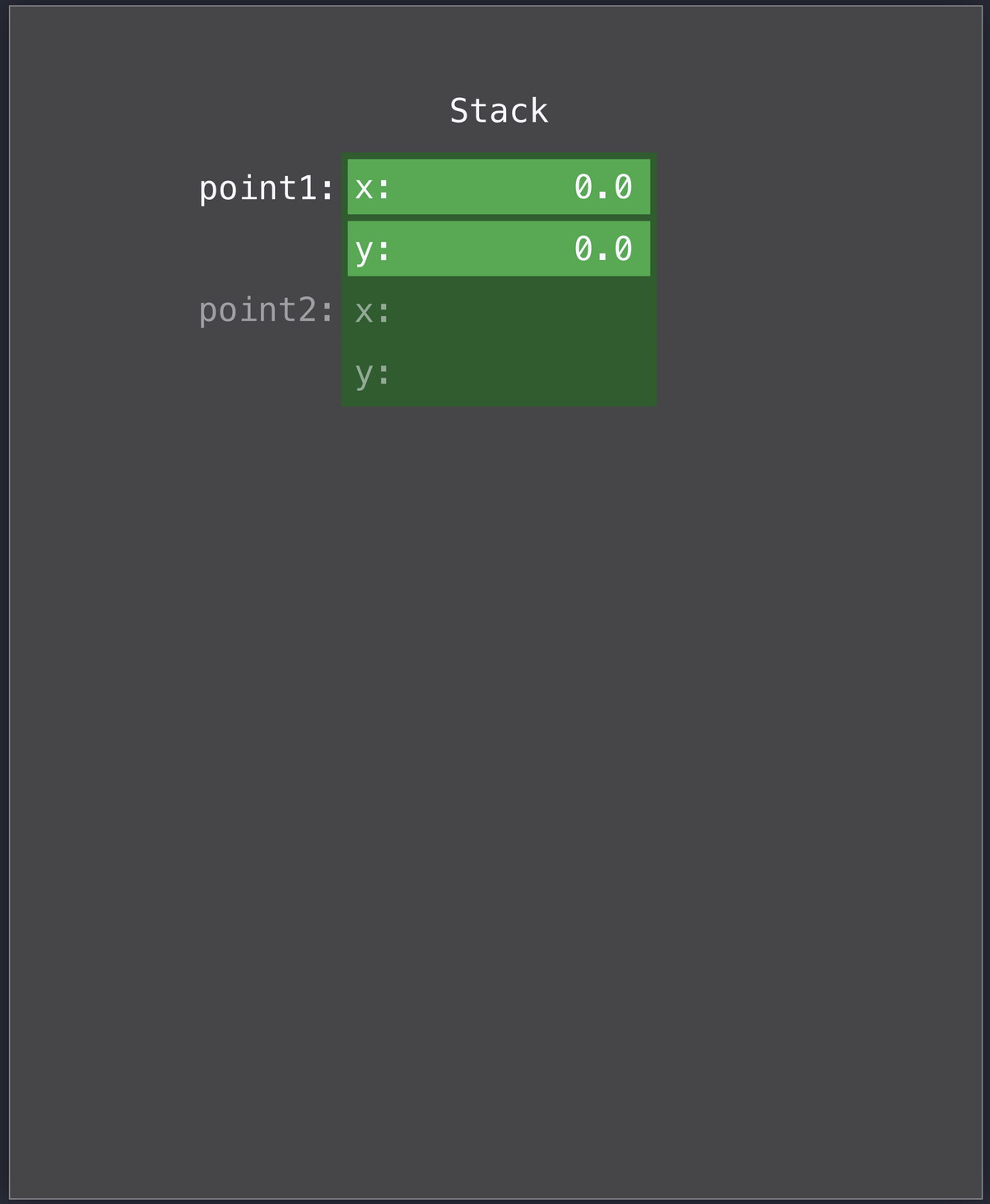
Reinsert block of memory to deallocate

Thread safety overhead

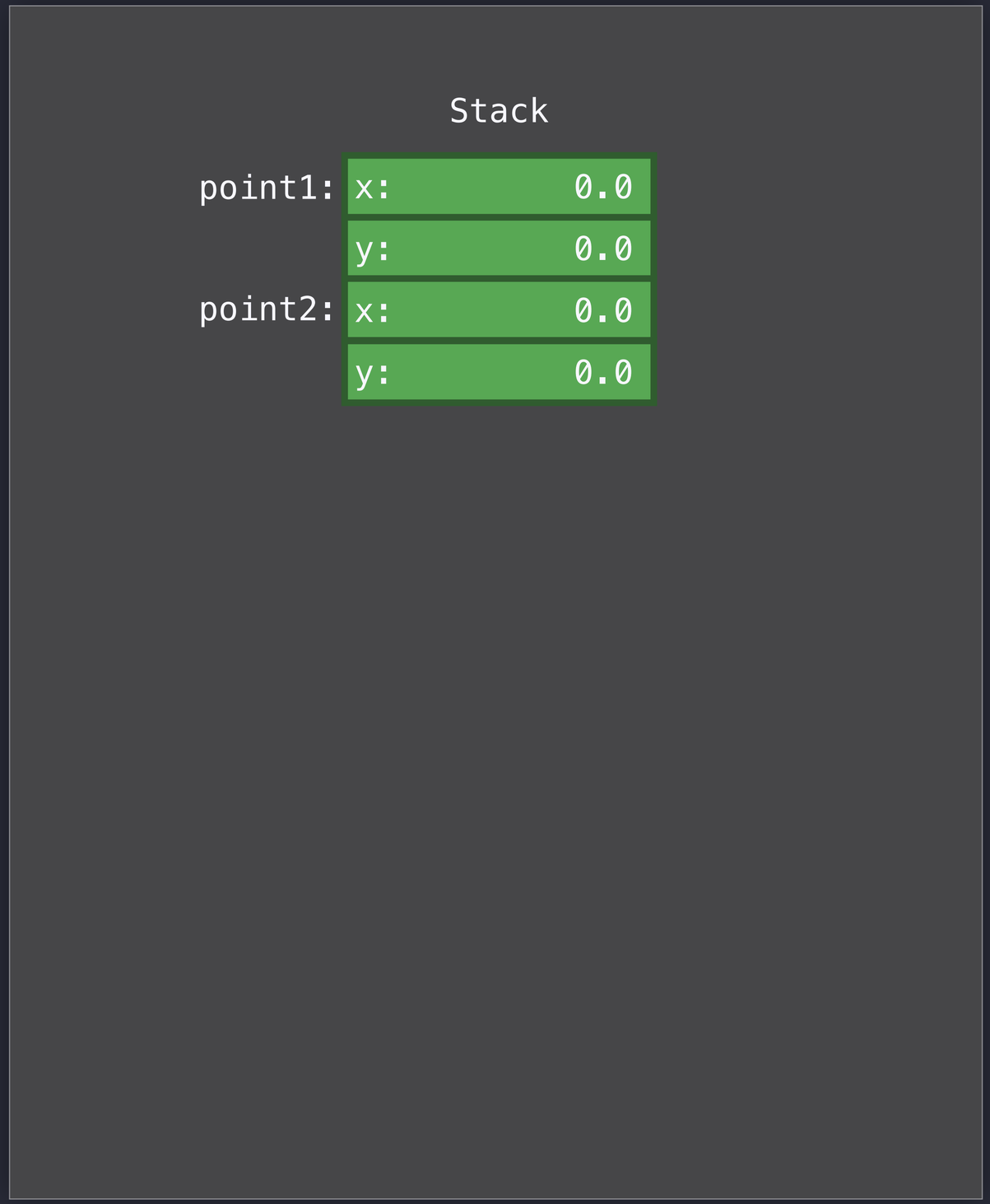
```
// Allocation
// Struct

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`
```

```swift
// Allocation
// Struct

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

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

```
// 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`
```

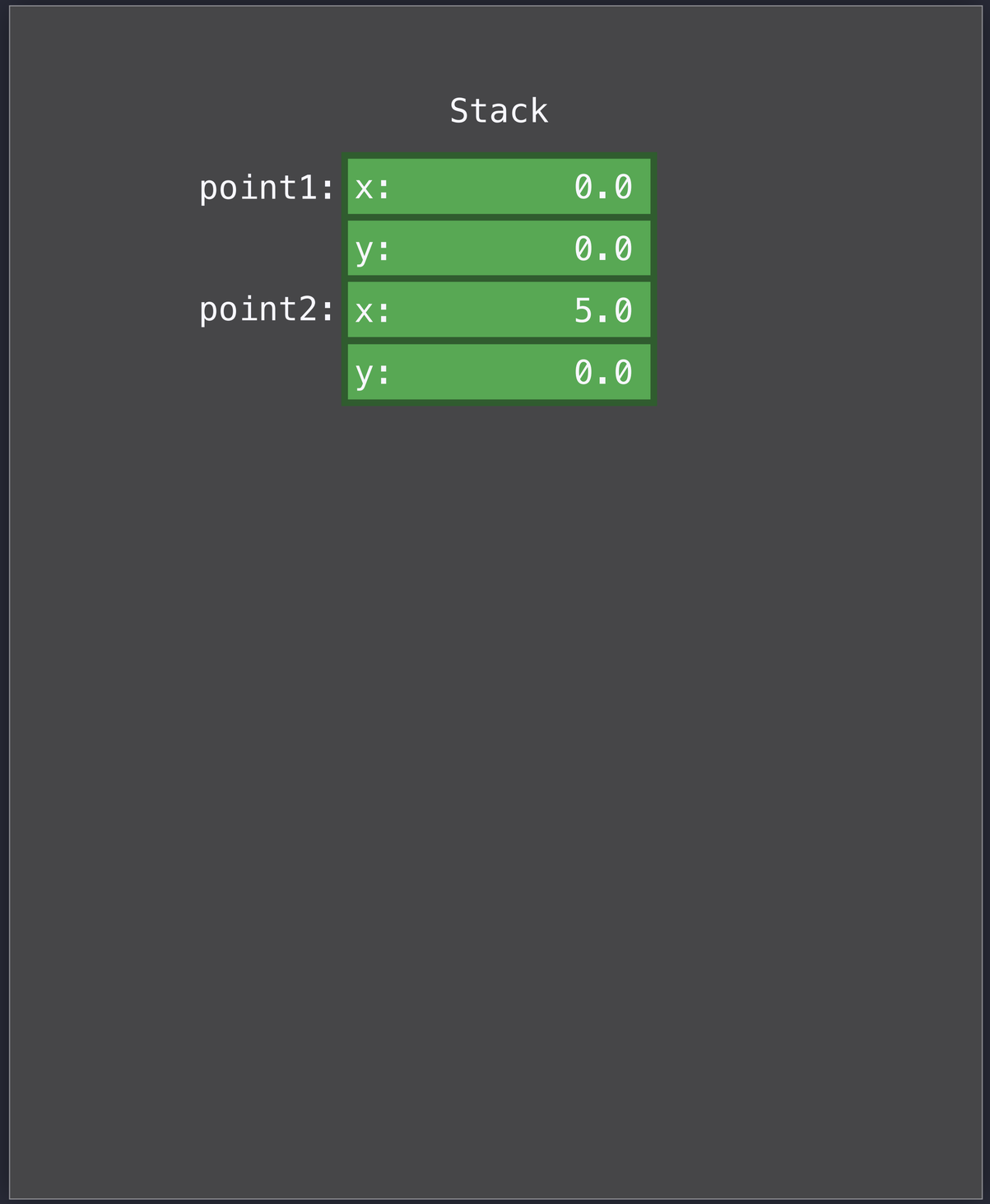


```
// Allocation
// Struct

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`
```

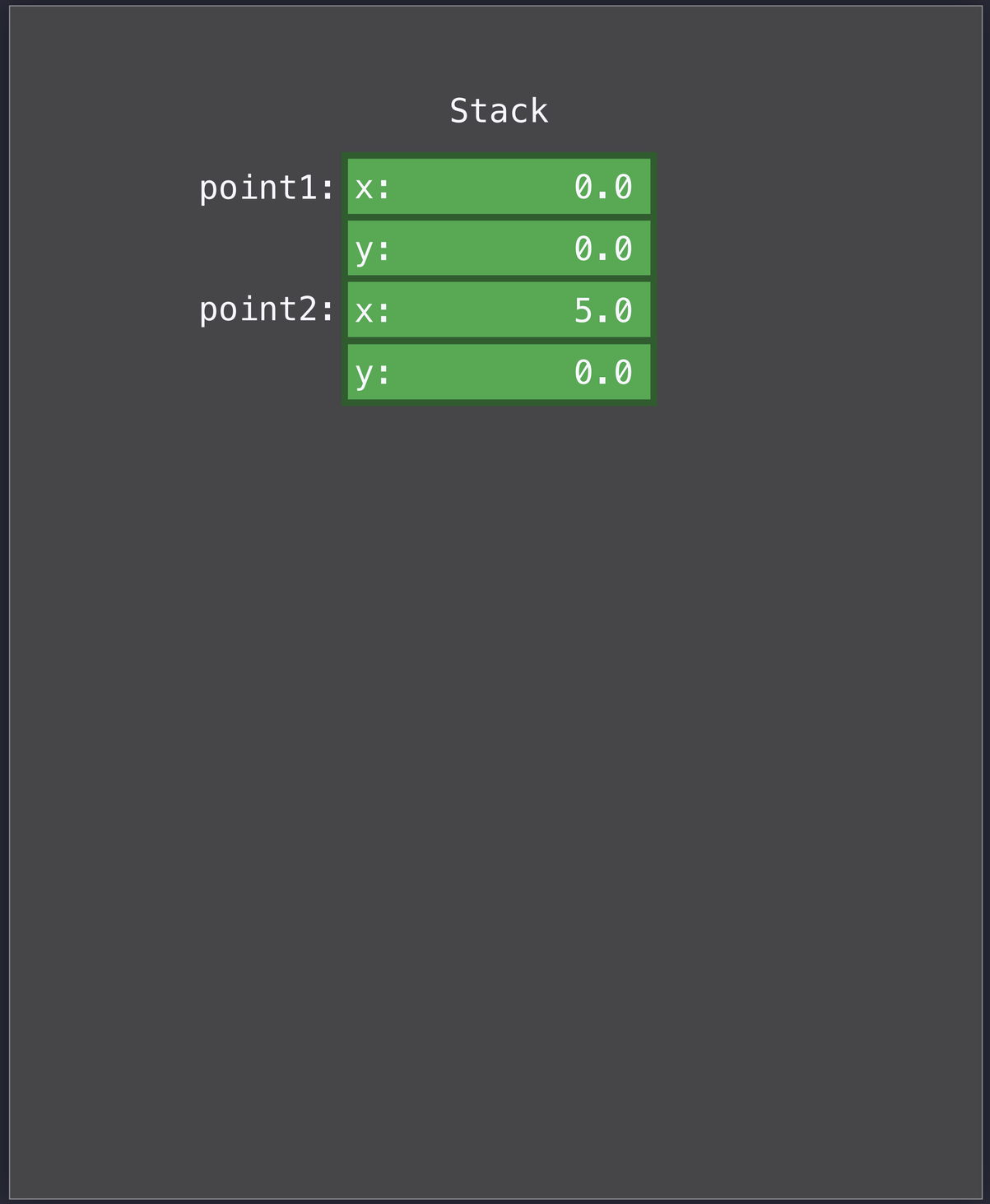


```
// Allocation
// Struct

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`
```

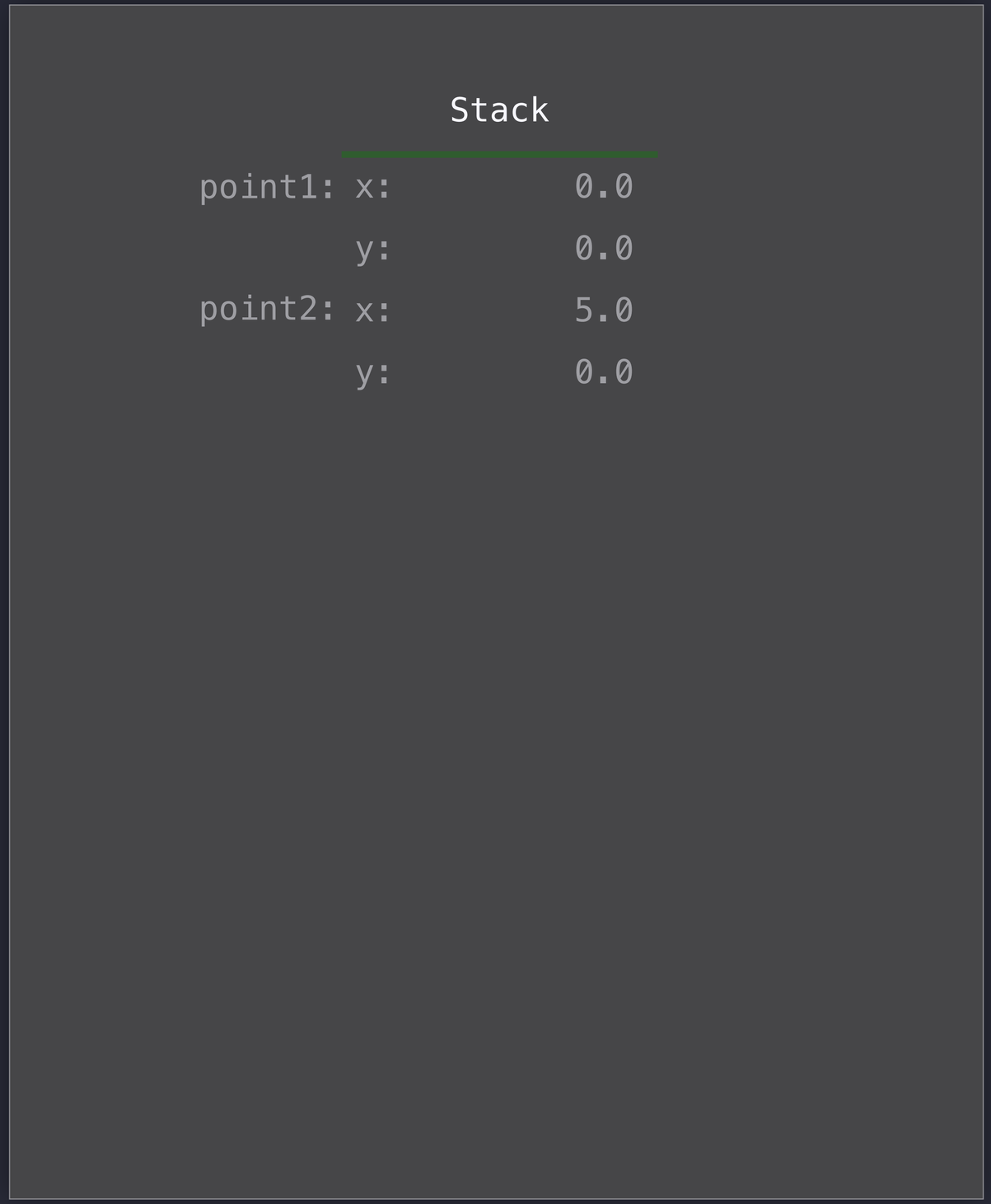
```
// Allocation
// Struct

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`
```



```swift
// Allocation
// Struct

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

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



```swift
// Allocation
// Struct

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

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

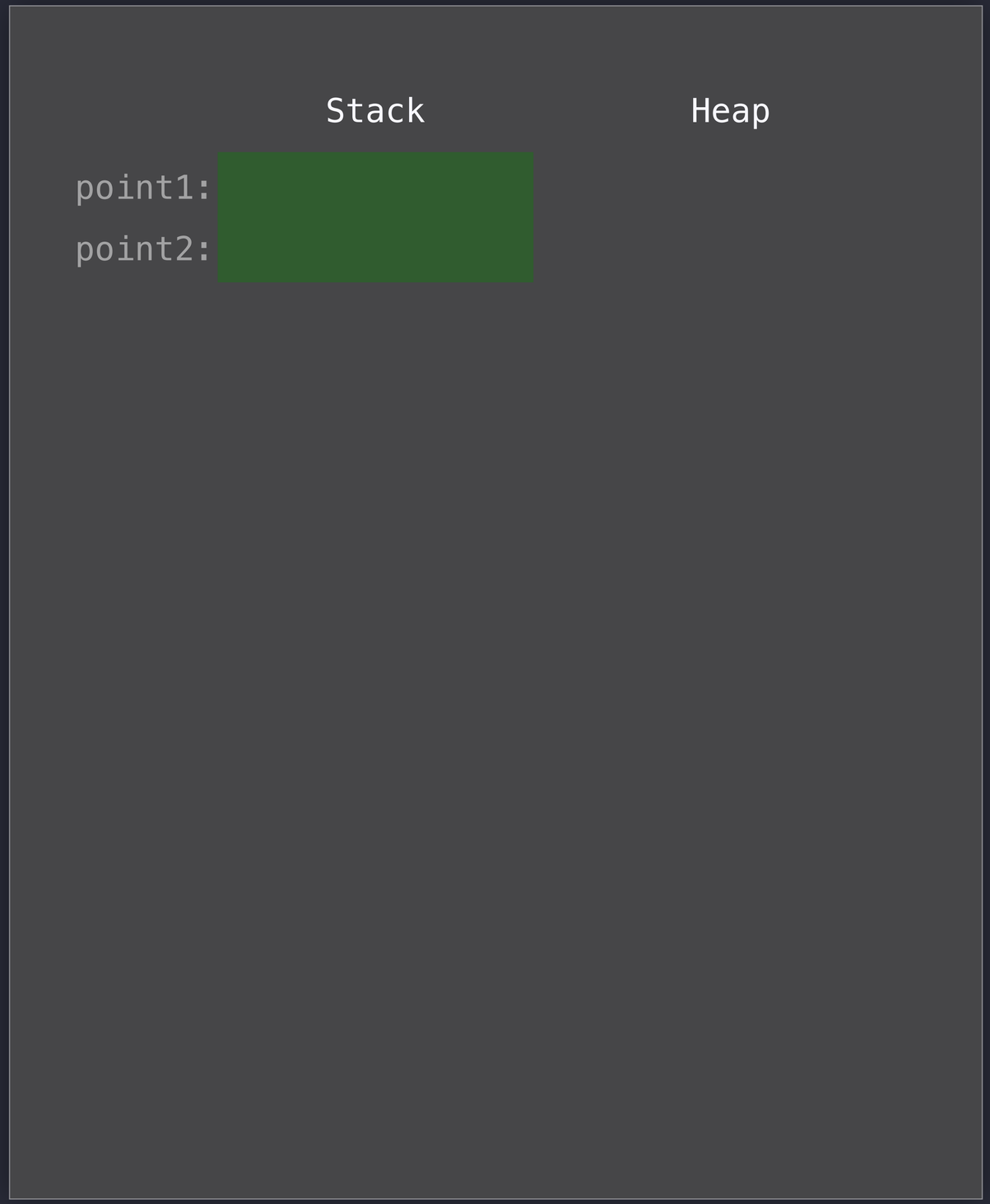


```swift
// Allocation
// Struct

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

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

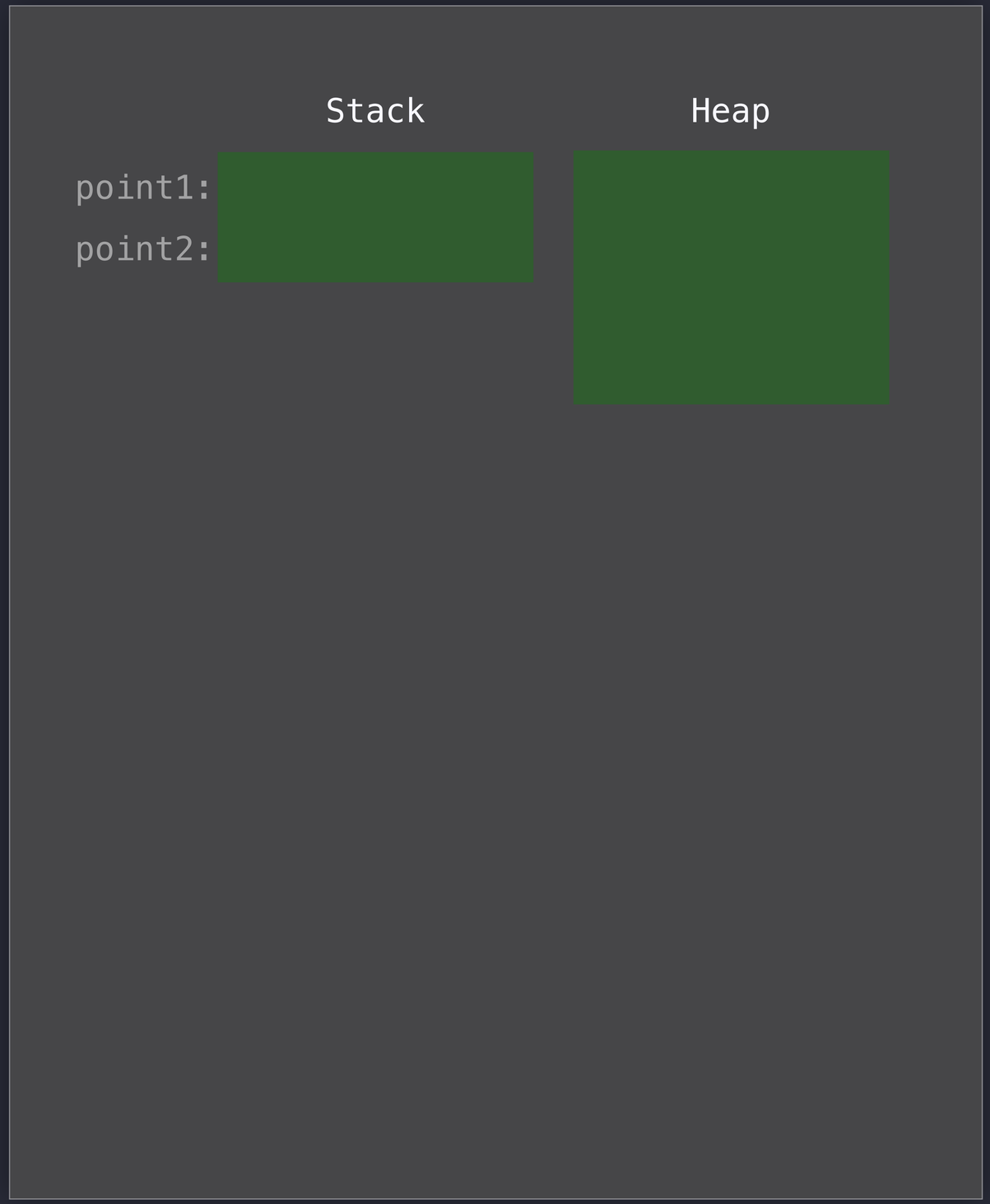
```swift
// Allocation
// Class

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

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

```
// 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`
```

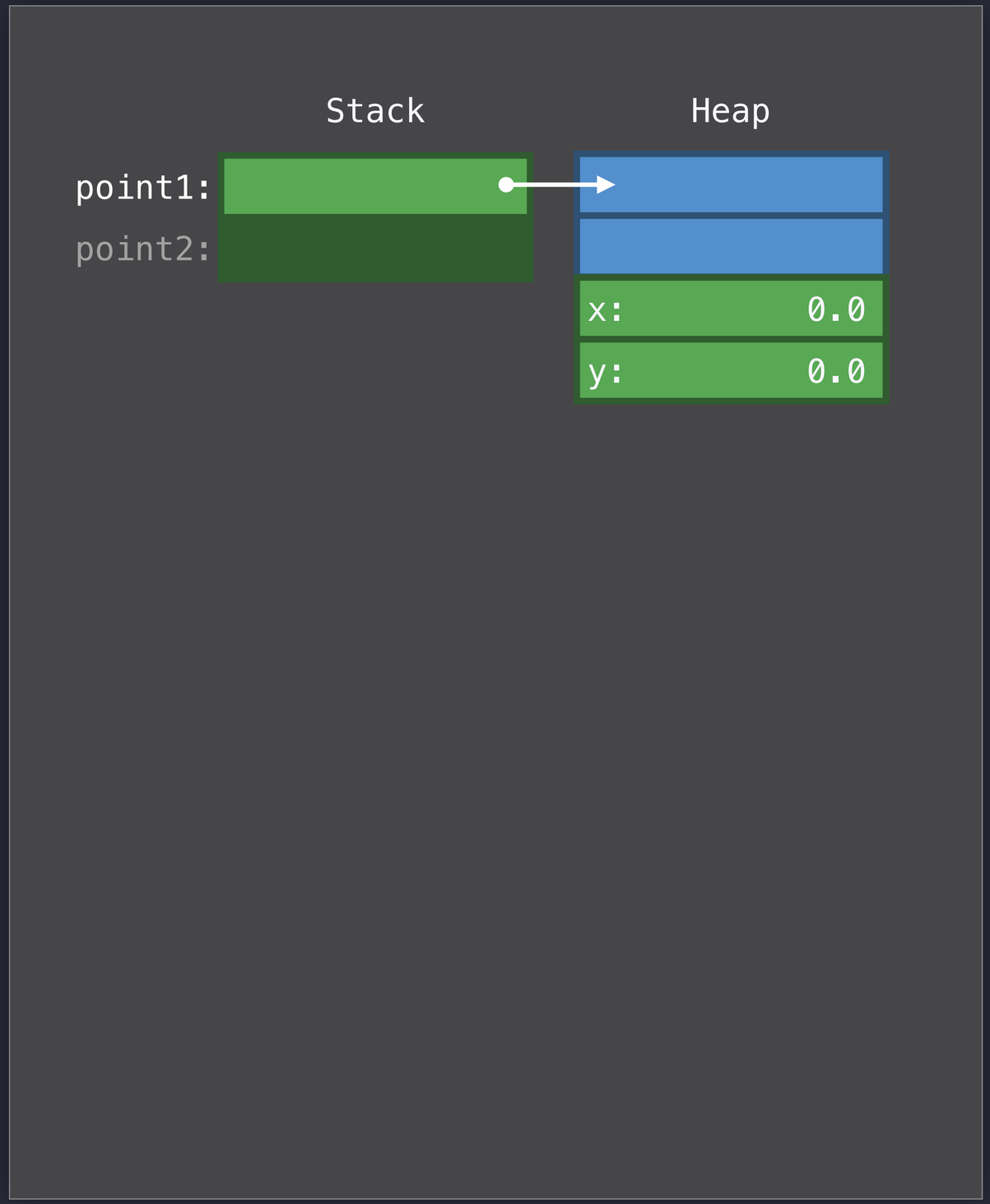
```swift
// Allocation
// Class

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

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



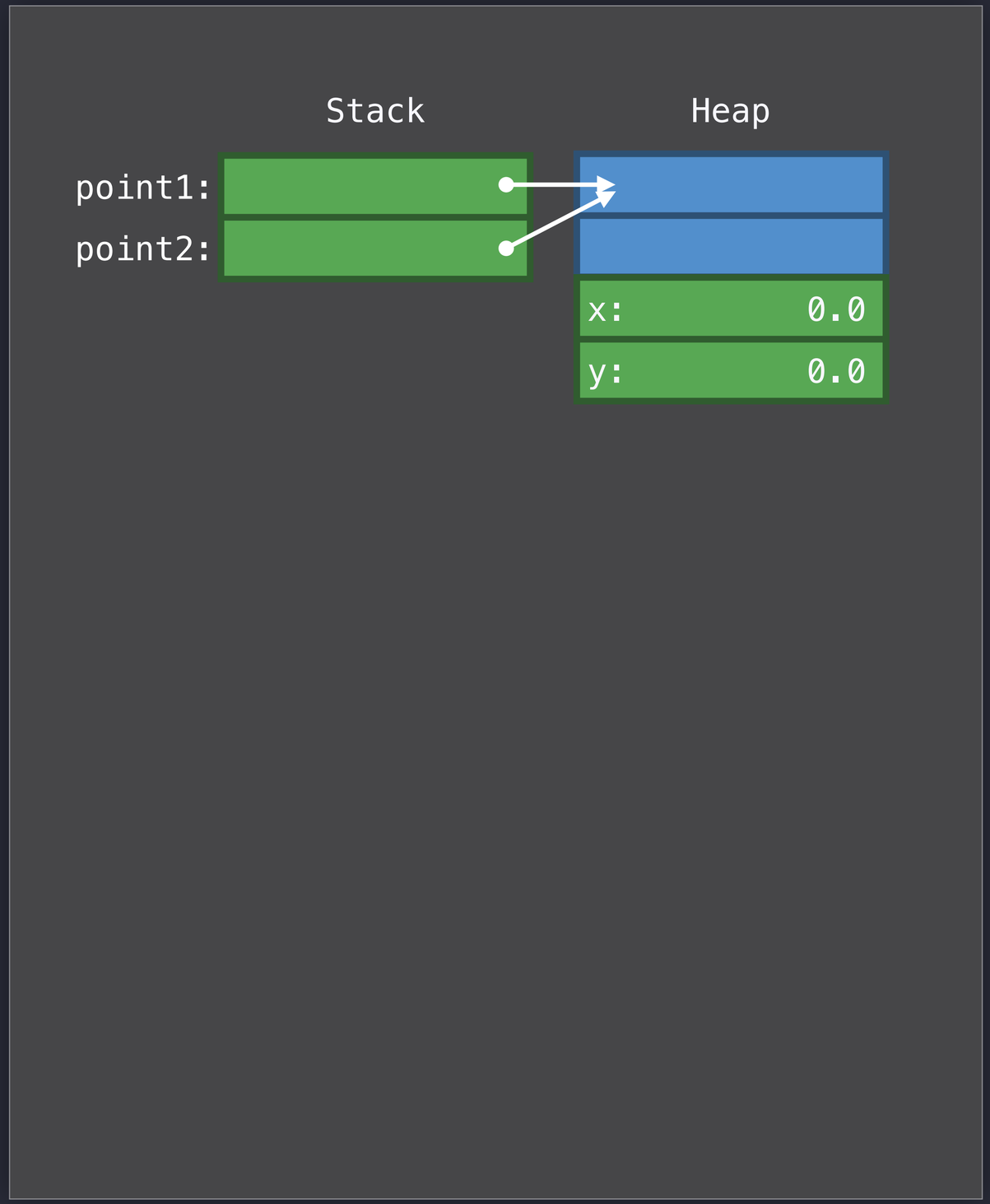
```swift
// Allocation
// Class

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

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



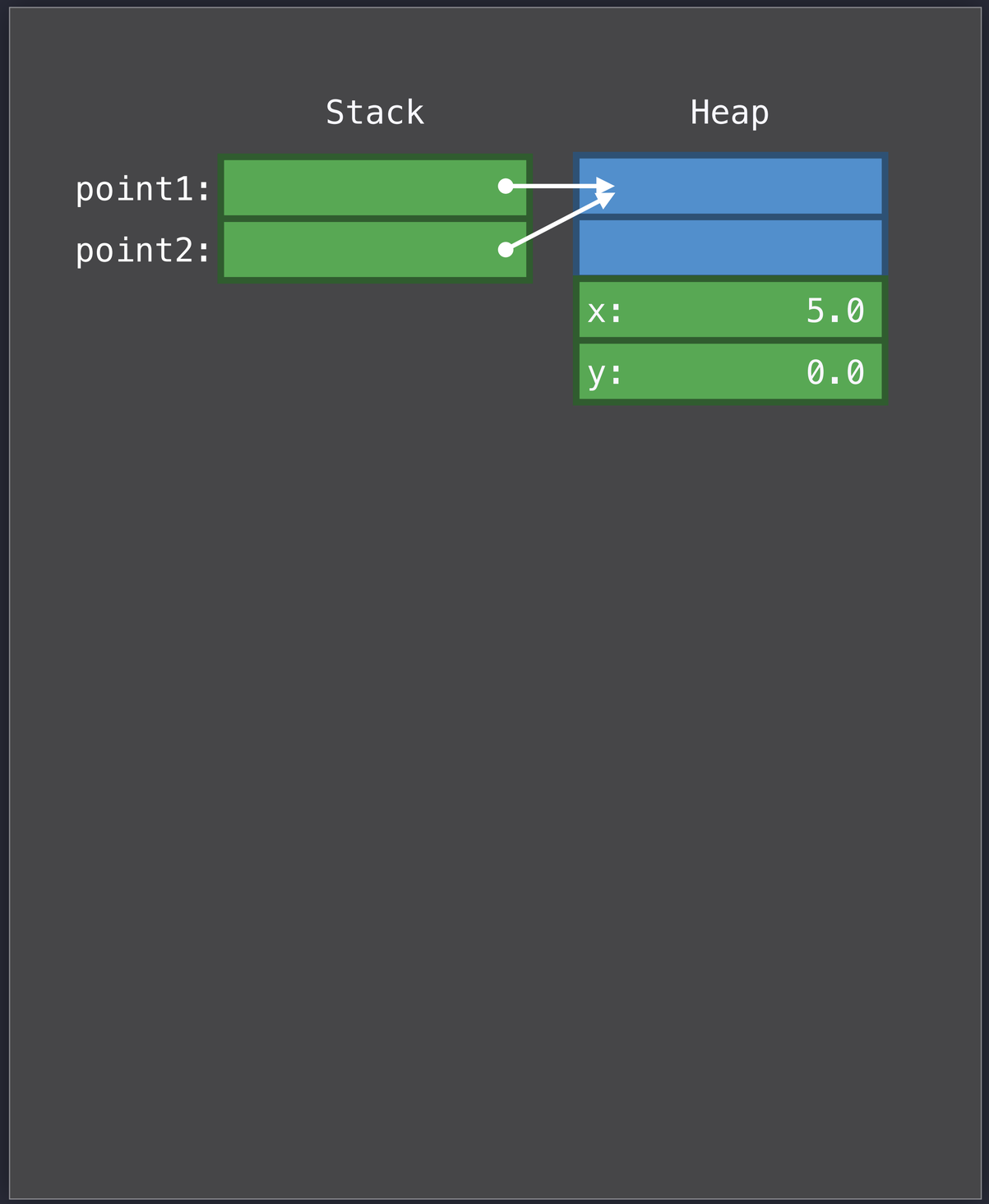
```swift
// Allocation
// Class

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

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



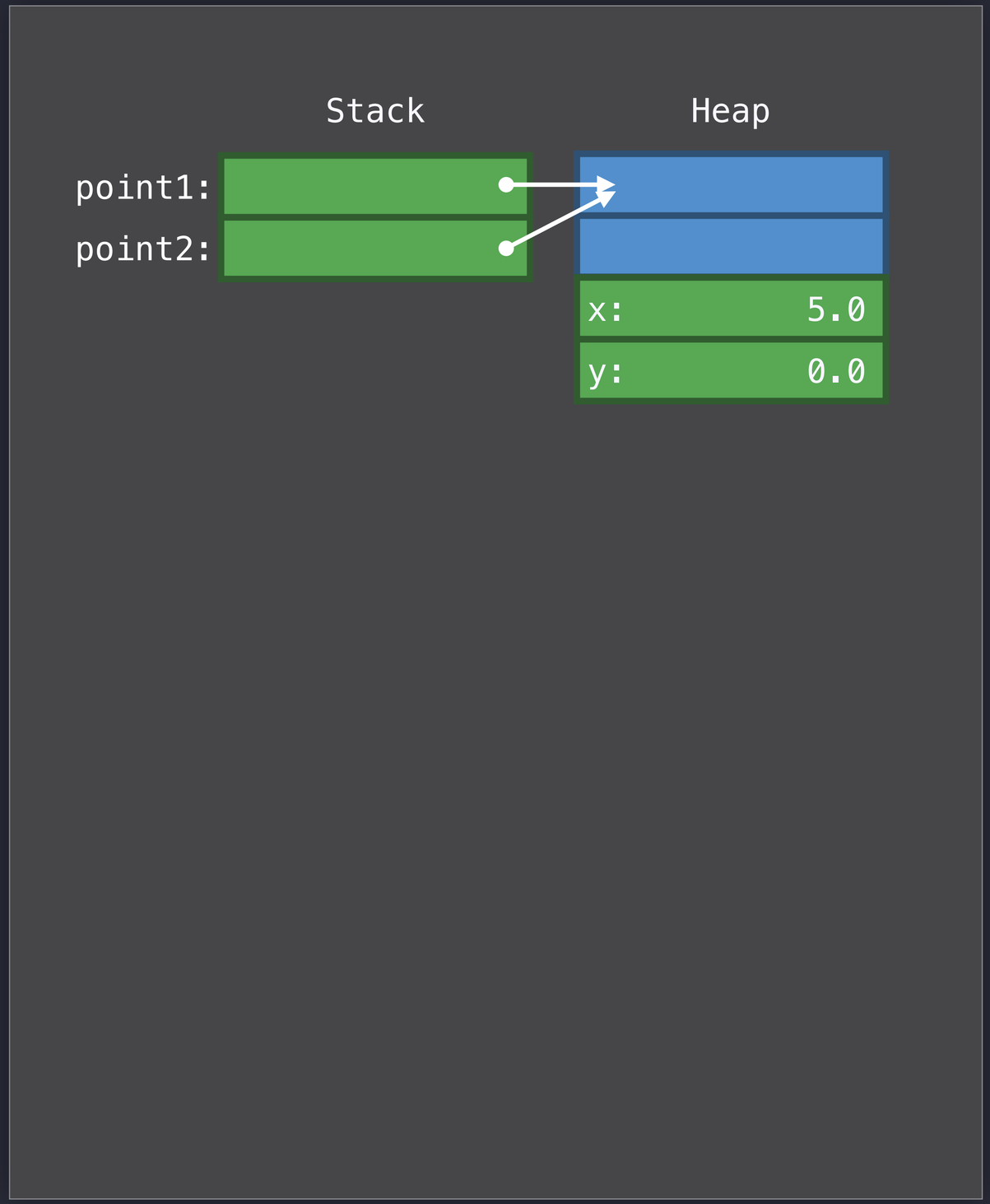
```
// Allocation
// Class

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`
```



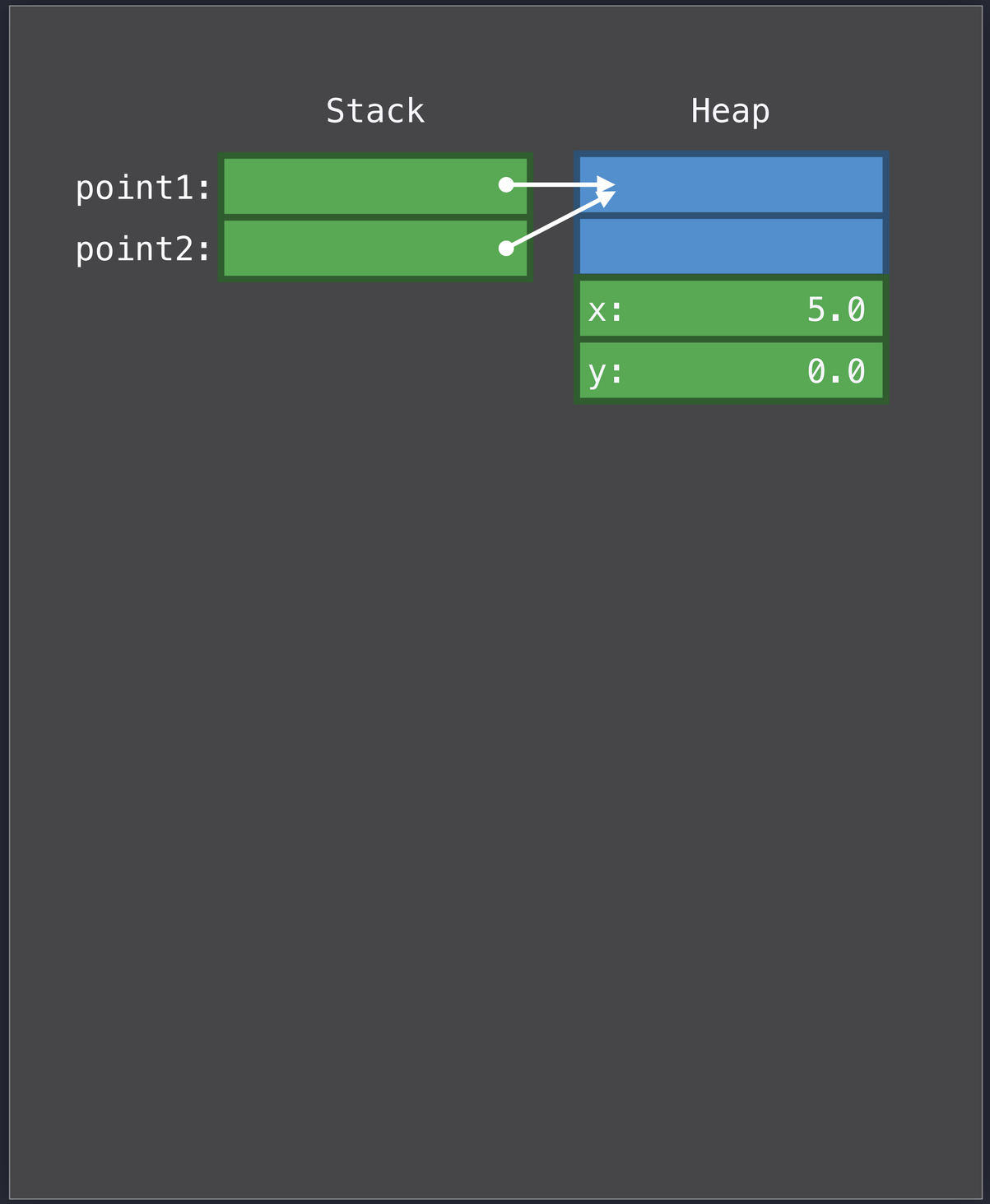
```
// Allocation
// Class

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`
```



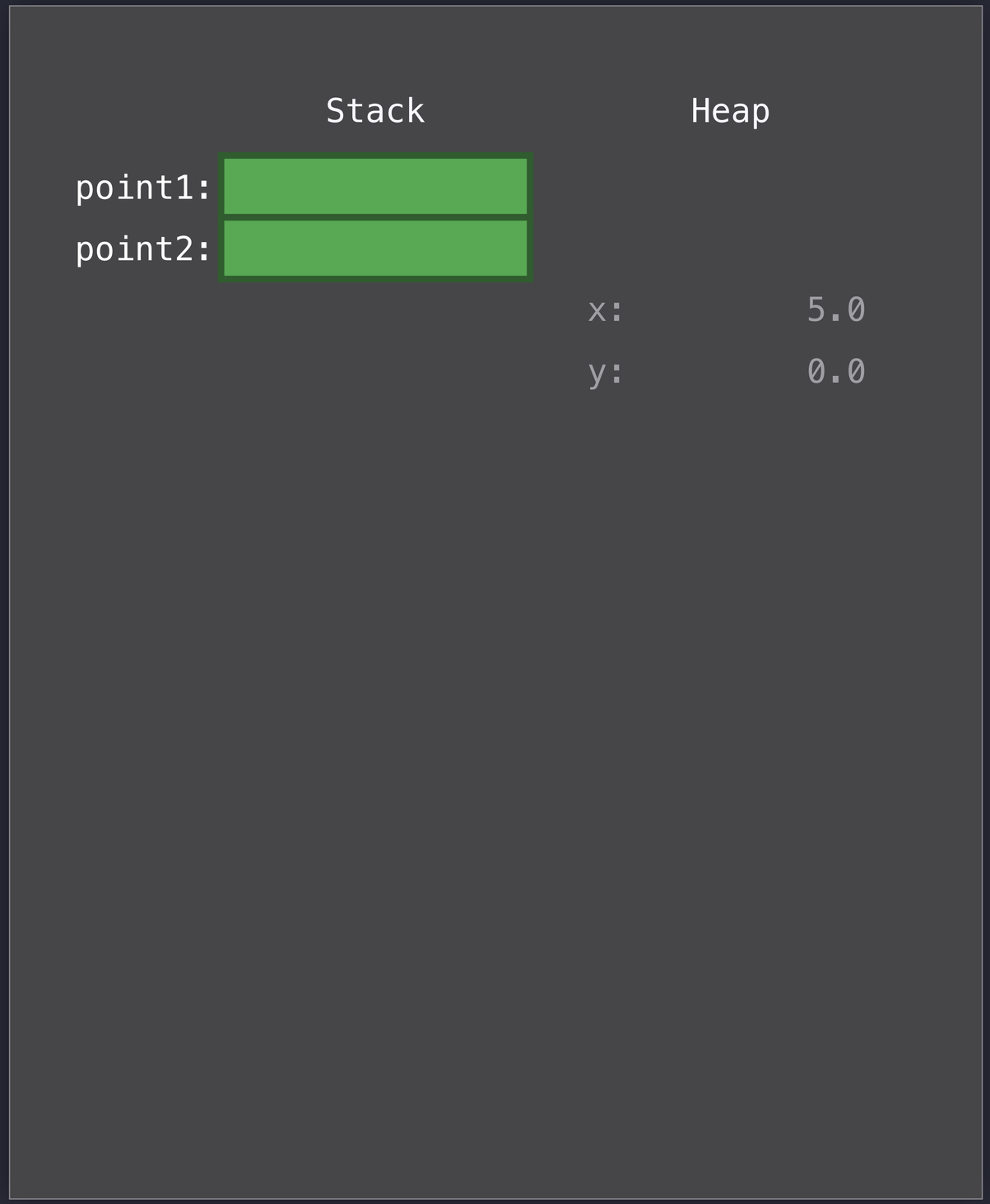
```
// Allocation
// Class

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`
```



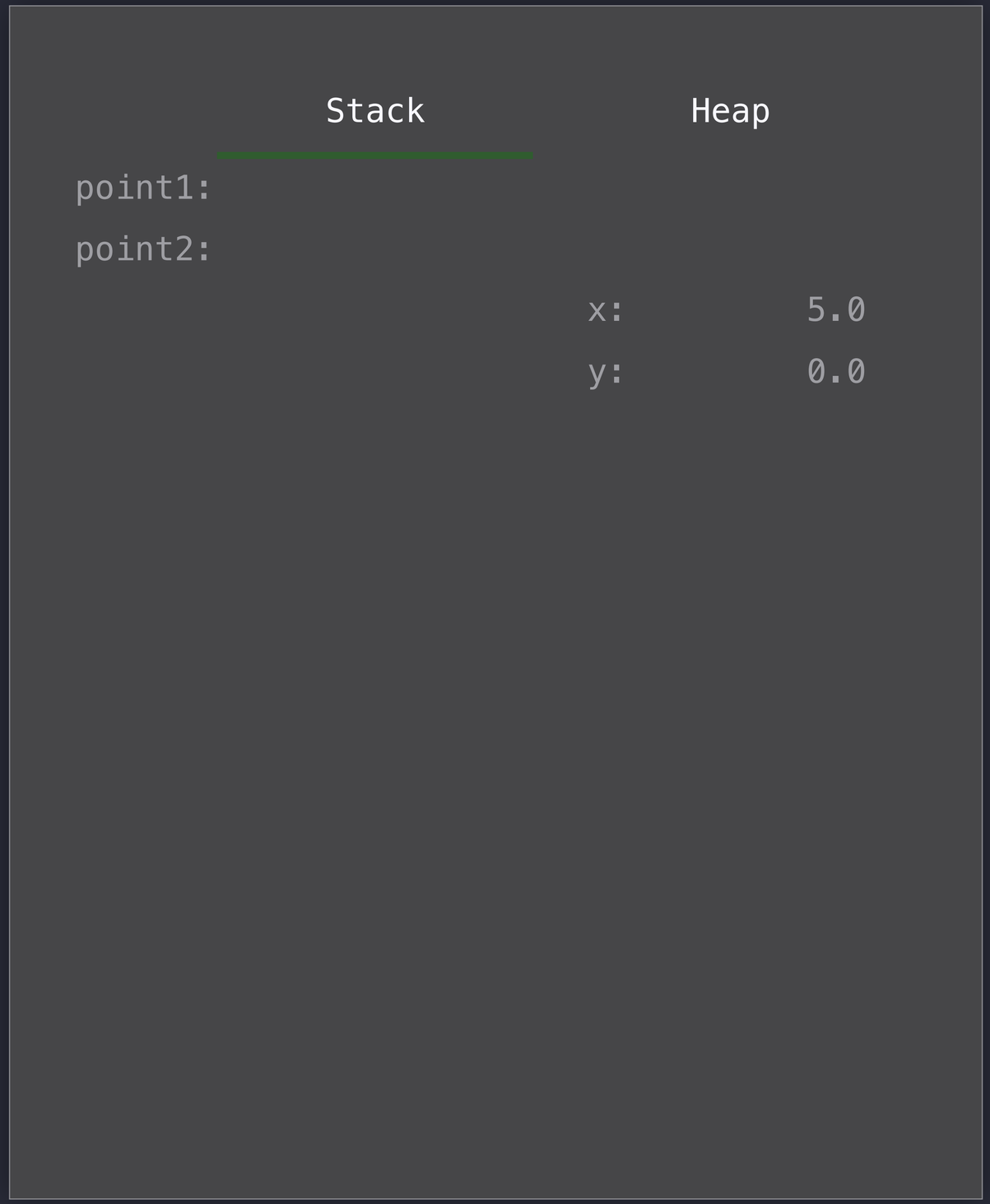
```swift
// Allocation
// Class

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

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



```
// 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`
```



Dimensions of Performance

Class

Allocation

Stack  Heap

Reference Counting

Less  More

Method Dispatch

Static  Dynamic

Dimensions of Performance

Struct

Allocation

Stack — Heap

Reference Counting

Less — More

Method Dispatch

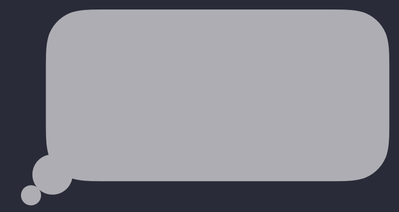
Static — Dynamic

```
// Modeling Techniques: Allocation

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



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

```swift
// Modeling Techniques: Allocation

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



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

```swift
// Modeling Techniques: Allocation

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



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

```
// Modeling Techniques: Allocation

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


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



```
// Modeling Techniques: Allocation

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

var cache = [String : UIImage]()

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

```
// Modeling Techniques: Allocation

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

var cache = [String : UIImage]()

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

```
// Modeling Techniques: Allocation

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

var cache = [String : UIImage]()
```

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

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

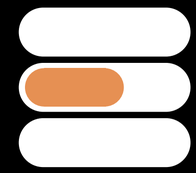
```
// Modeling Techniques: Allocation

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

var cache = [Attributes : UIImage]()
```

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

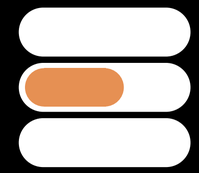
```
func makeBalloon(_ color: Color, orientation: Orientation, tail: Tail) -> UIImage {
    let key = Attributes(color: color, orientation: orientation, tail: tail)
    if let image = cache[key] {
        return image
    }
    …
}
```

Reference Counting

Reference Counting

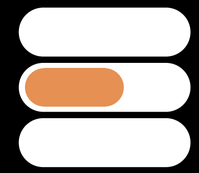


Reference Counting



There’s more to reference counting than incrementing, decrementing

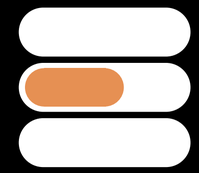
Reference Counting



There’s more to reference counting than incrementing, decrementing

- Indirection

Reference Counting



There’s more to reference counting than incrementing, decrementing

- Indirection
- Thread safety overhead

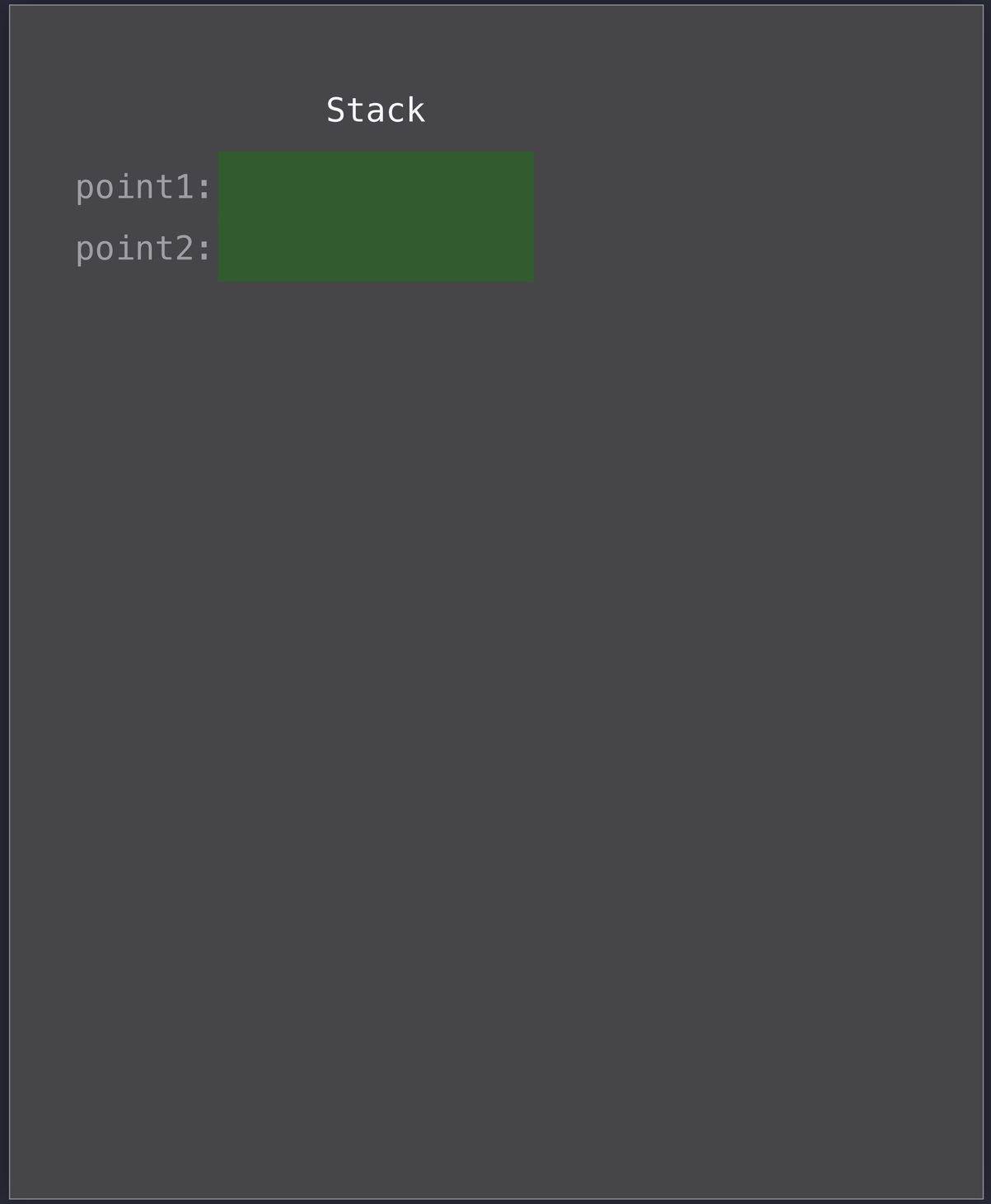
```
// Reference Counting
// Class

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
// 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`
```

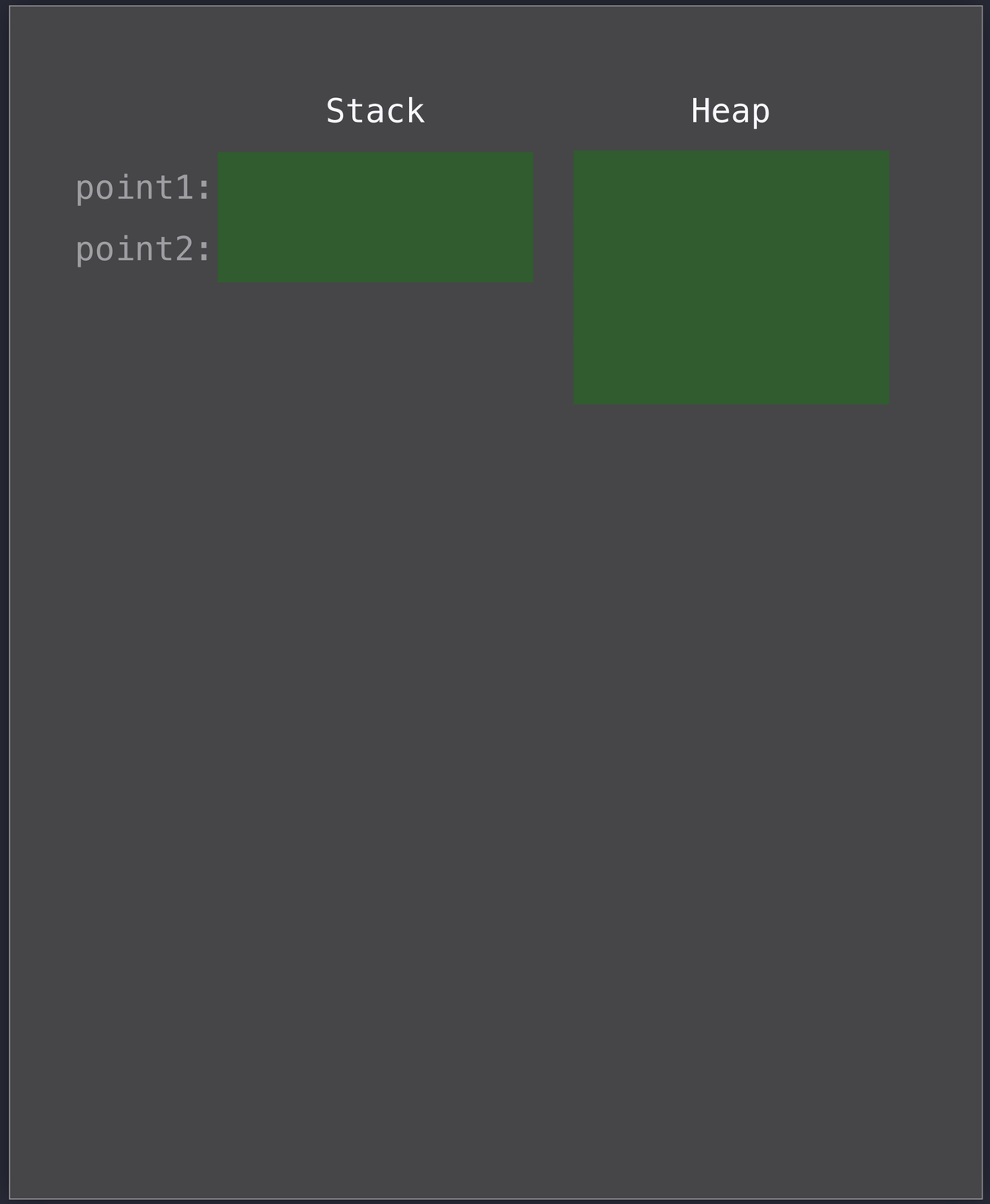
```
// Reference Counting
// Class (generated code)

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)
```

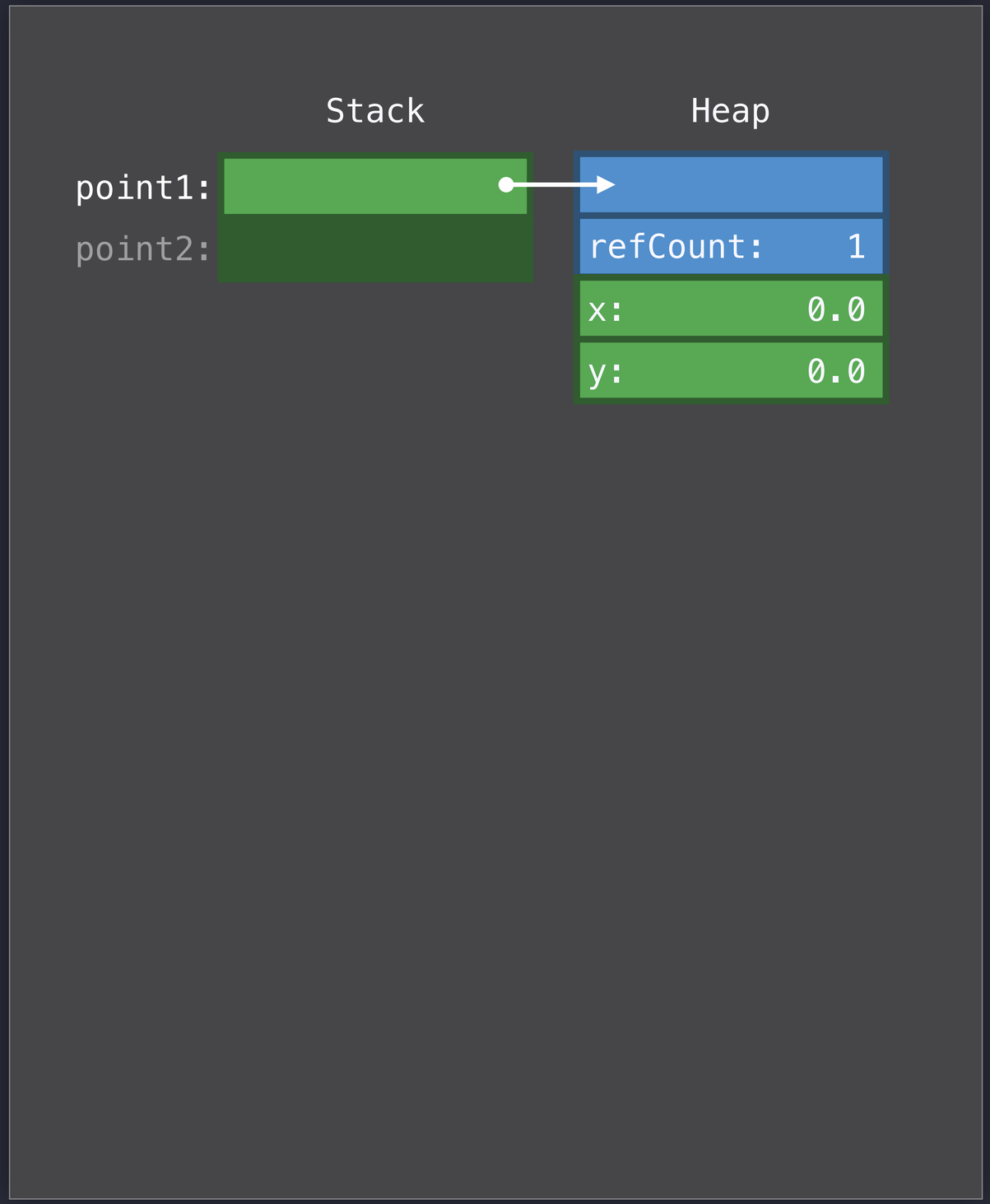
```
// Reference Counting
// Class (generated code)

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)
```



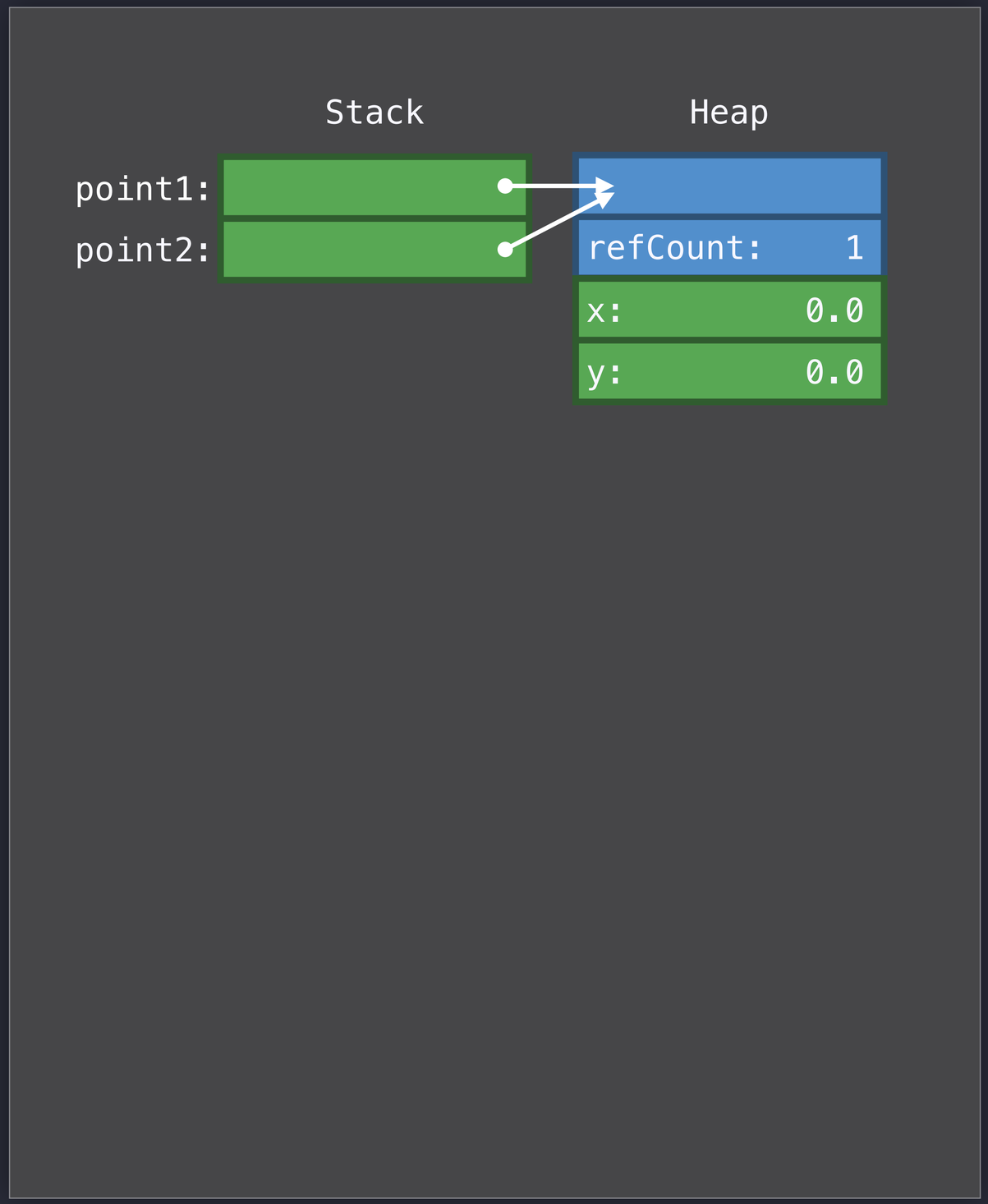
```
// Reference Counting
// Class (generated code)

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)
```



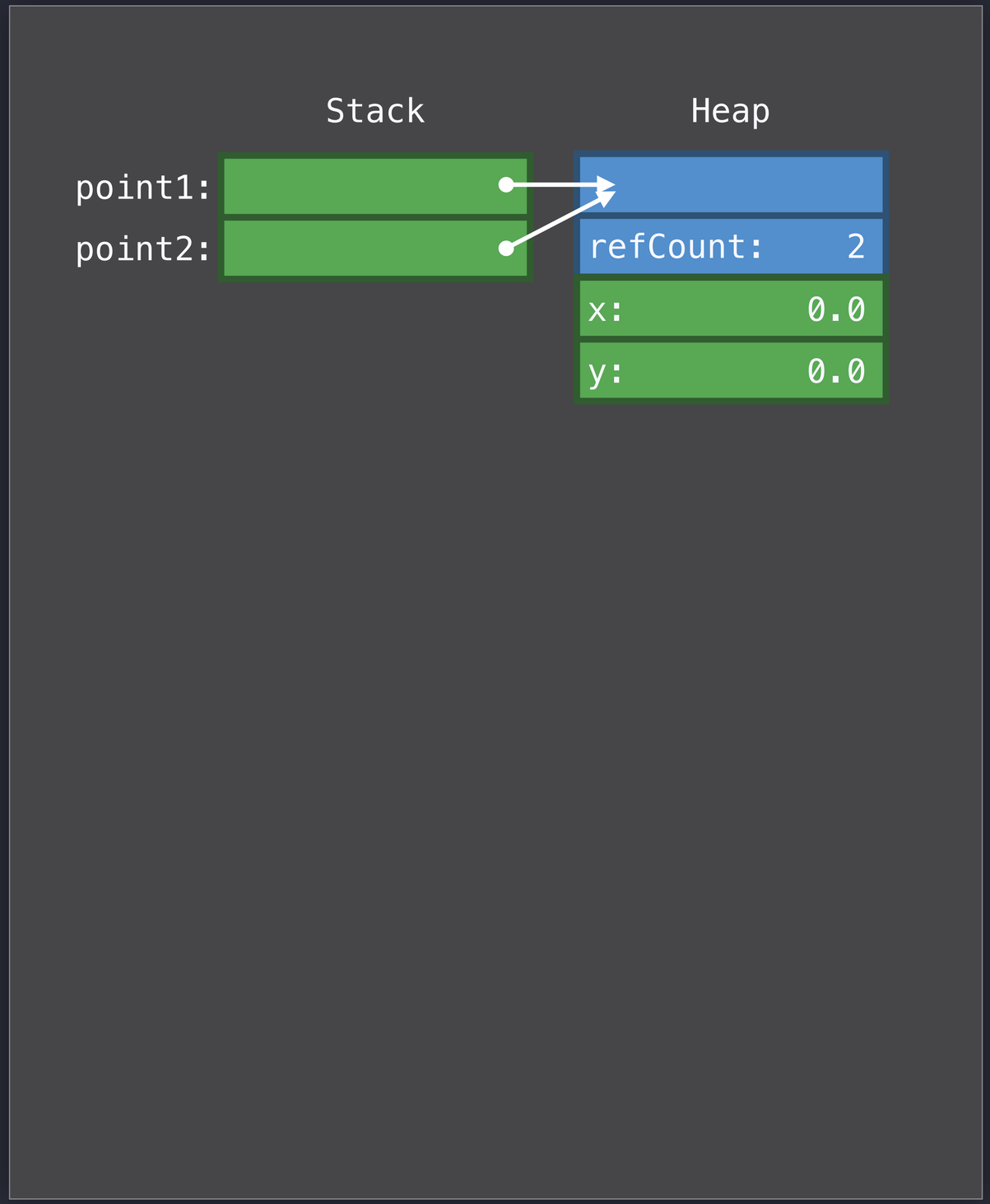
```
// Reference Counting
// Class (generated code)

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)
```



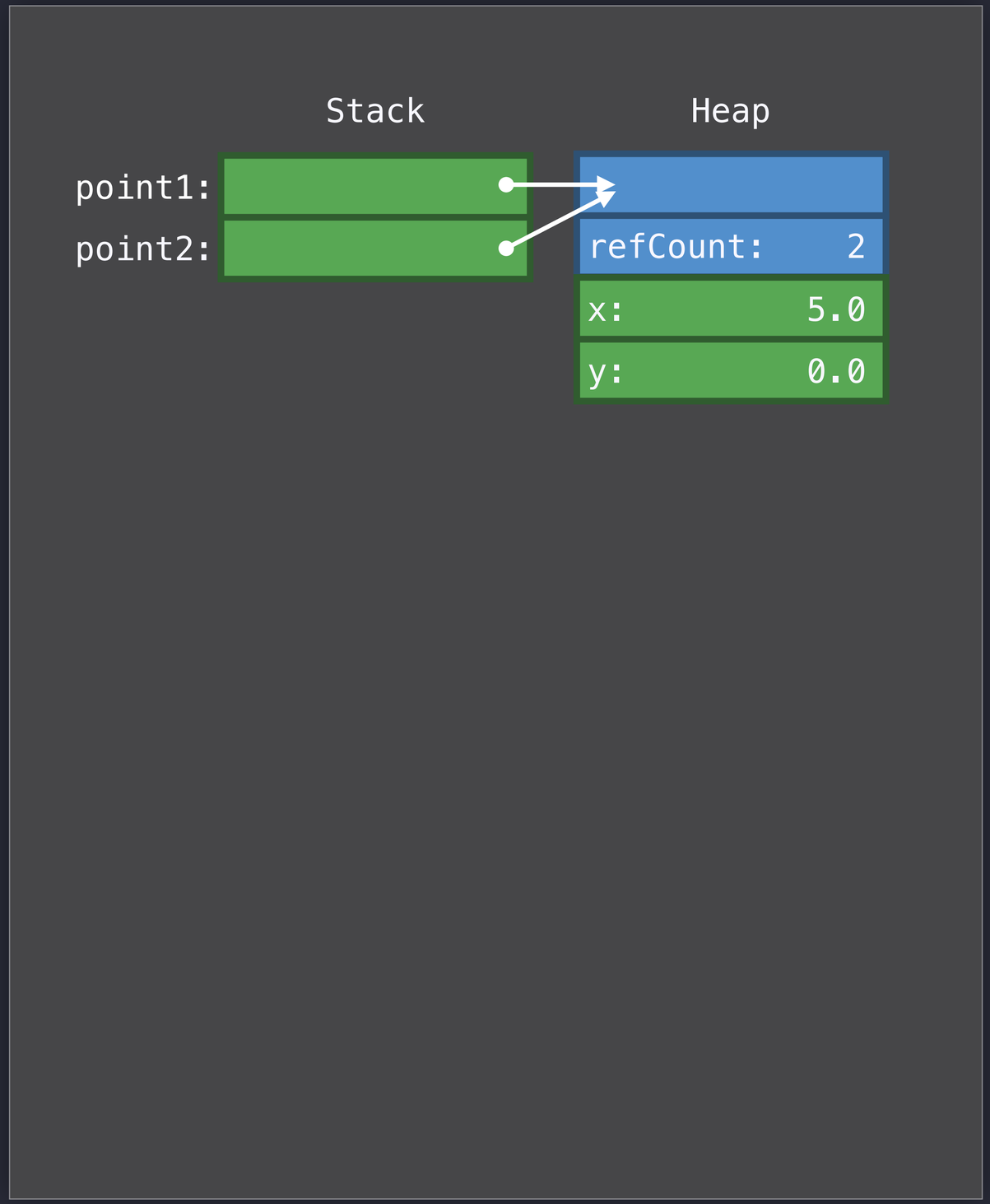
```
// Reference Counting
// Class (generated code)

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)
```



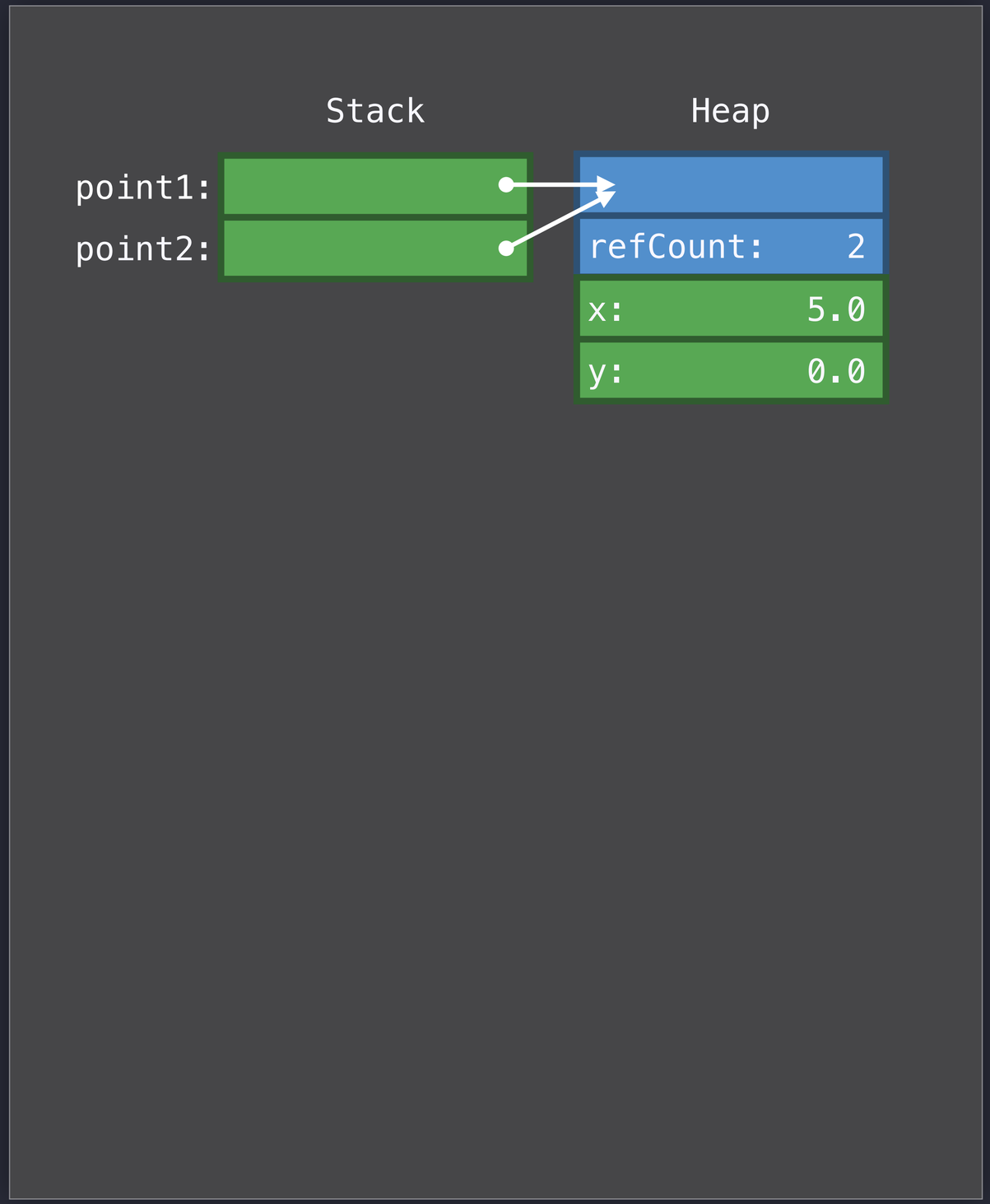
```
// Reference Counting
// Class (generated code)

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)
```



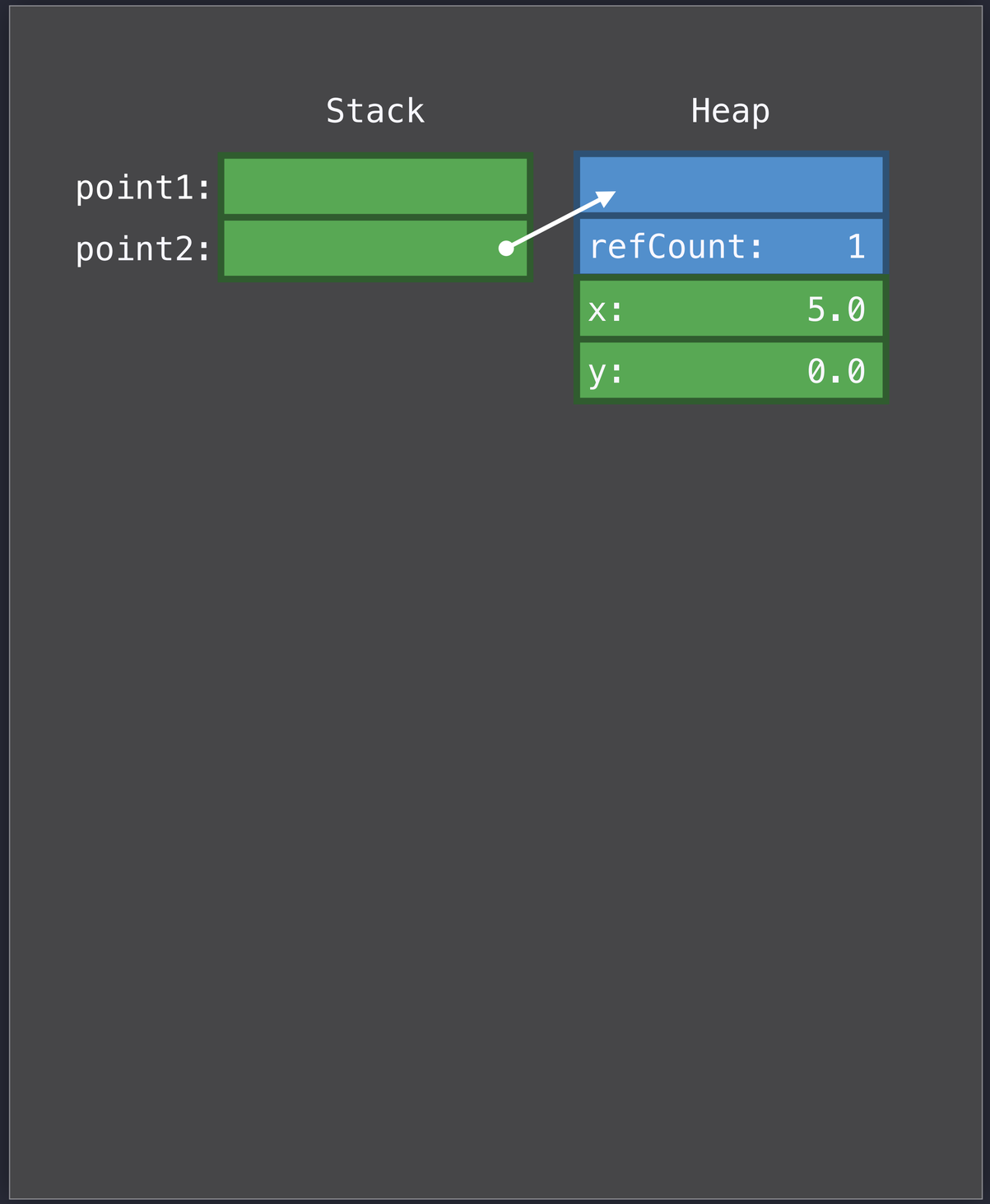
```
// Reference Counting
// Class (generated code)

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)
```



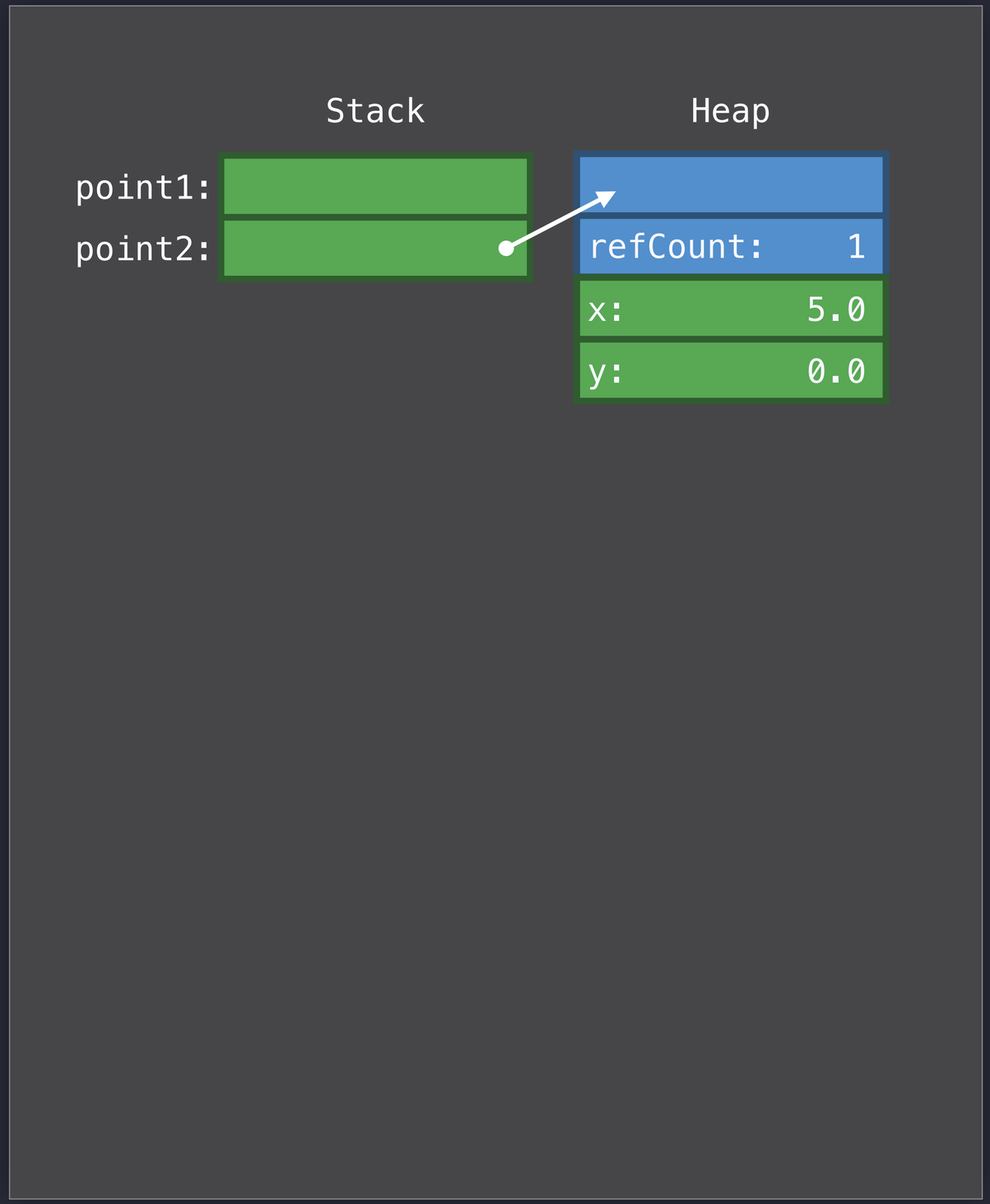
```
// Reference Counting
// Class (generated code)

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)
```



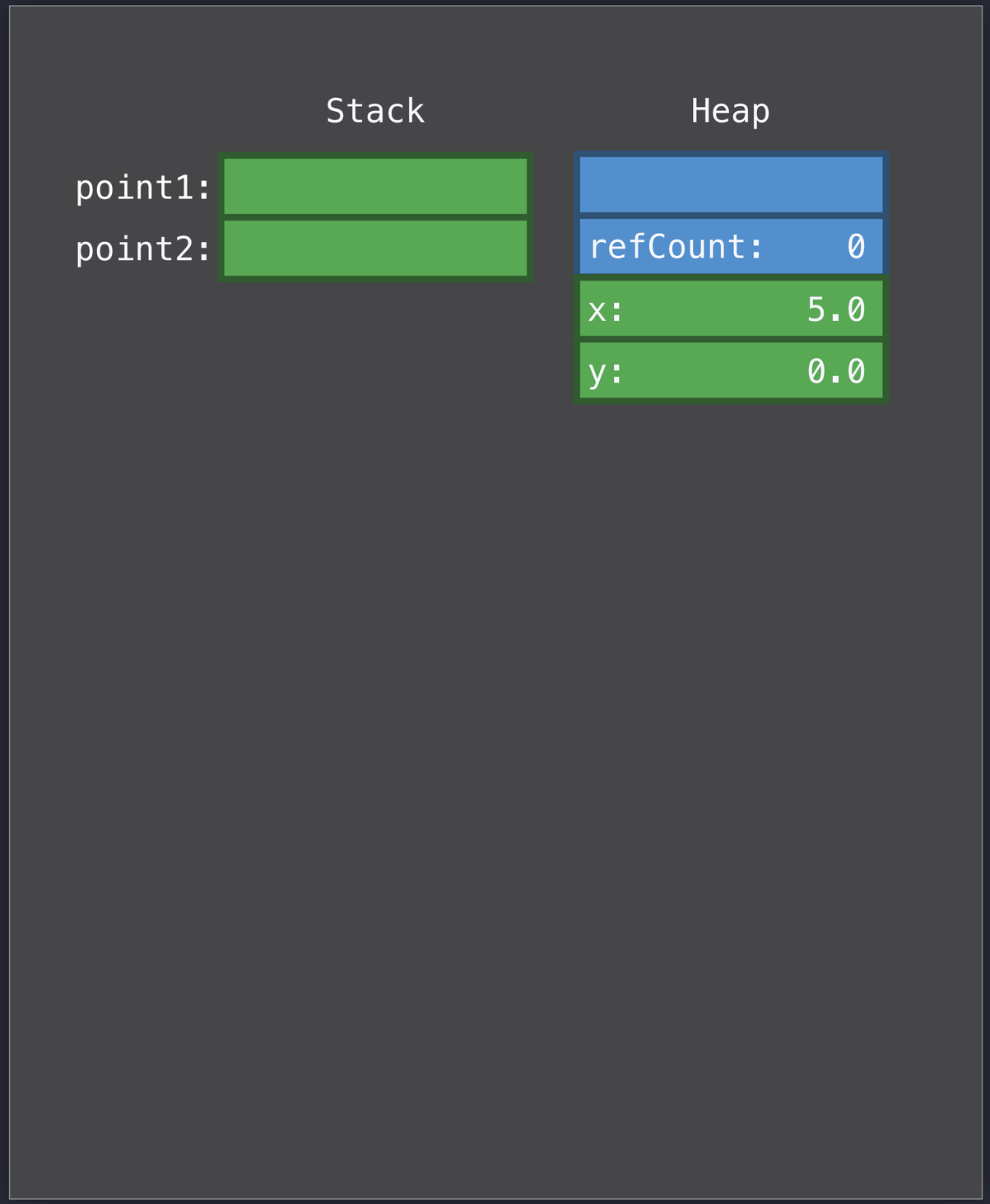
```
// Reference Counting
// Class (generated code)

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)
```



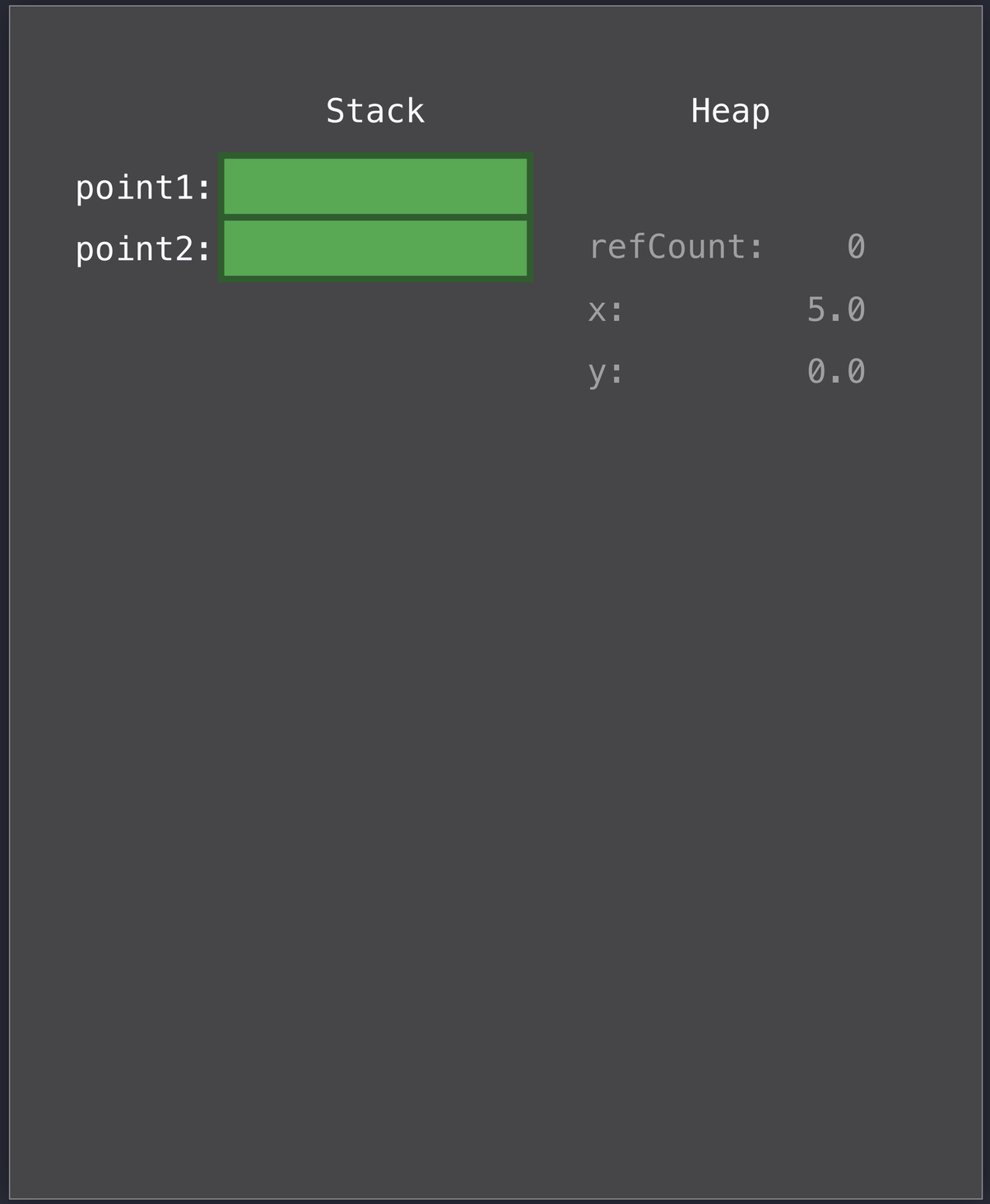
```
// Reference Counting
// Class (generated code)

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)
```



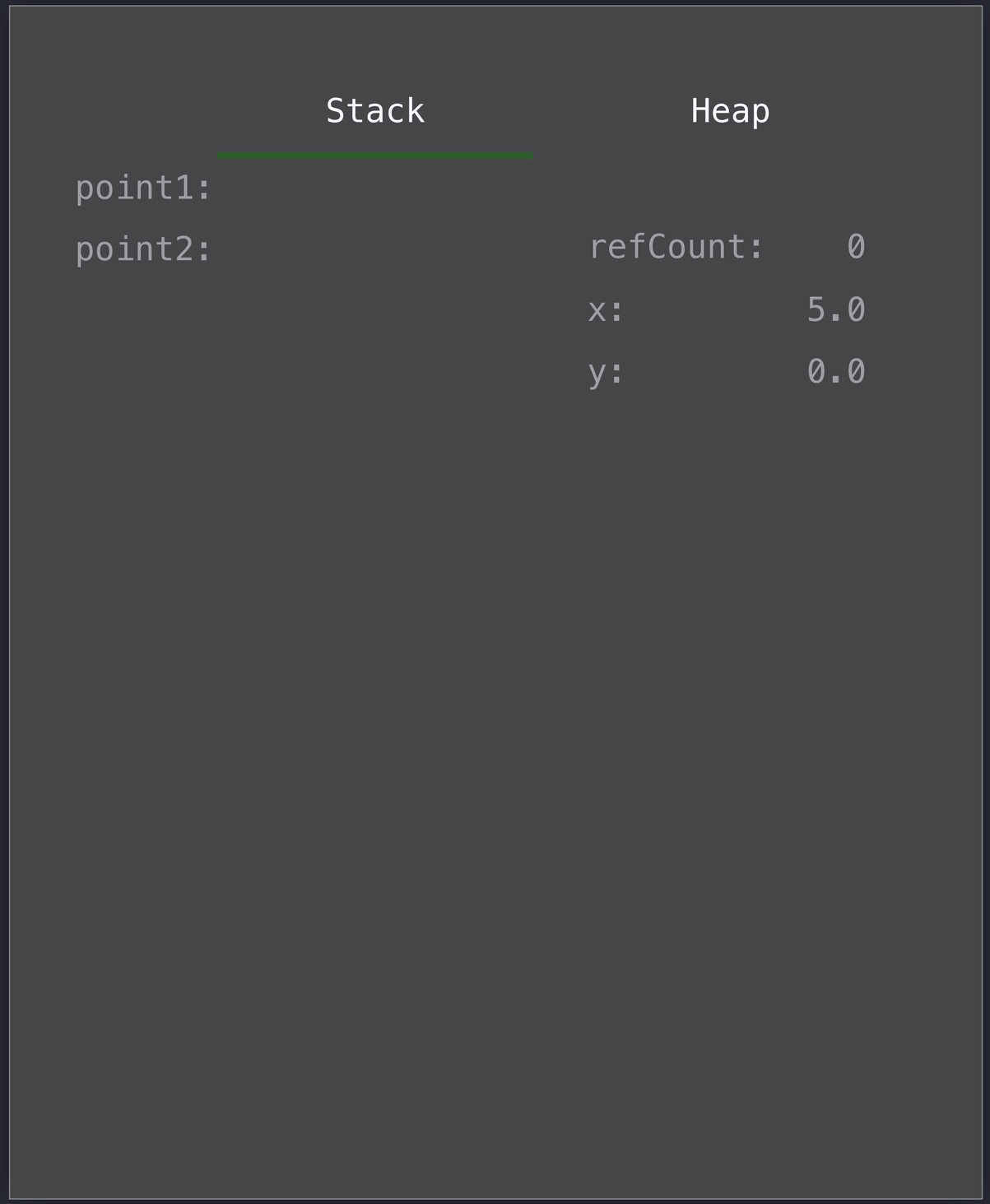
```
// Reference Counting
// Class (generated code)

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)
```



```
// 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)
```

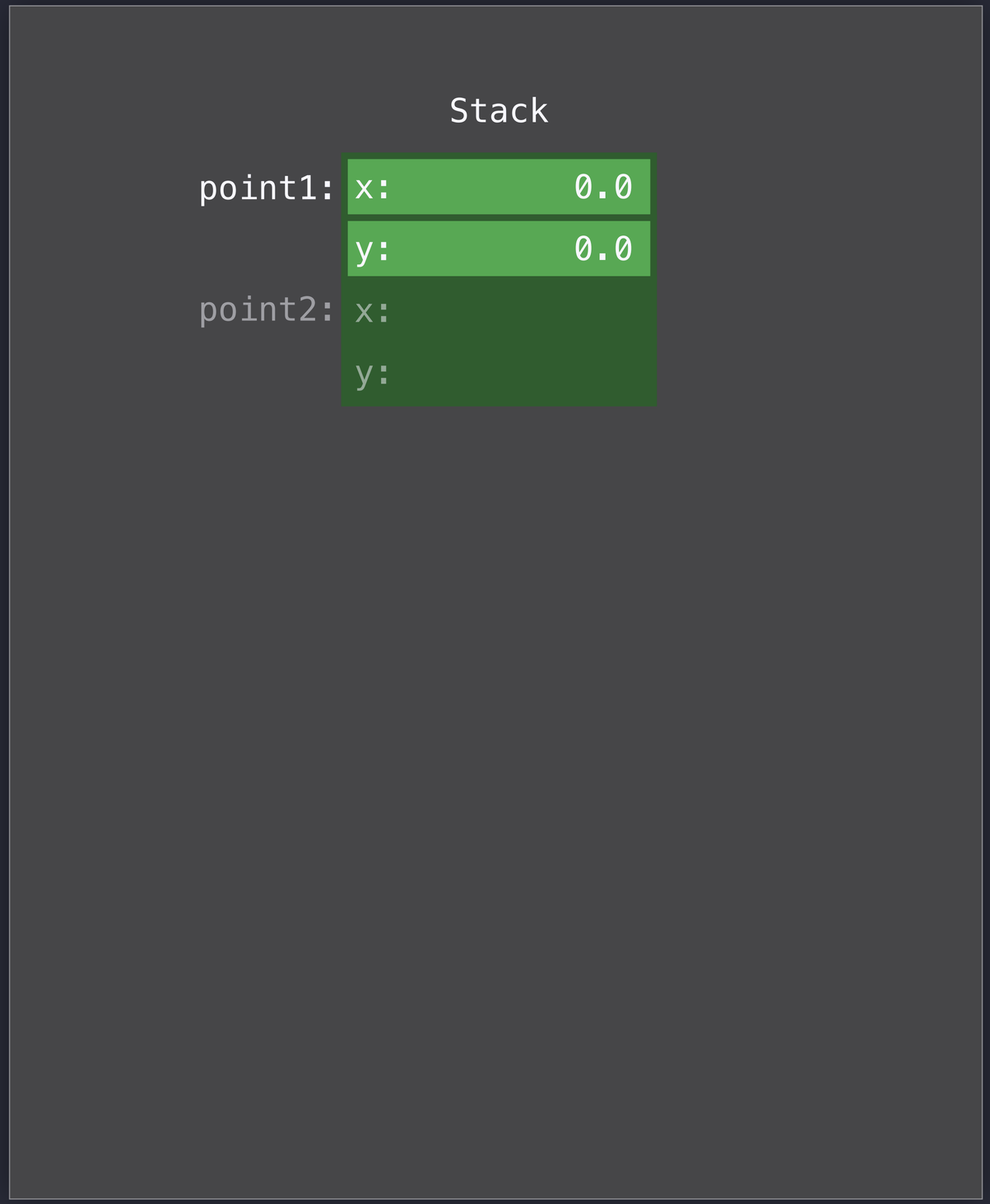


```
// Reference Counting
// Class (generated code)

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)
```

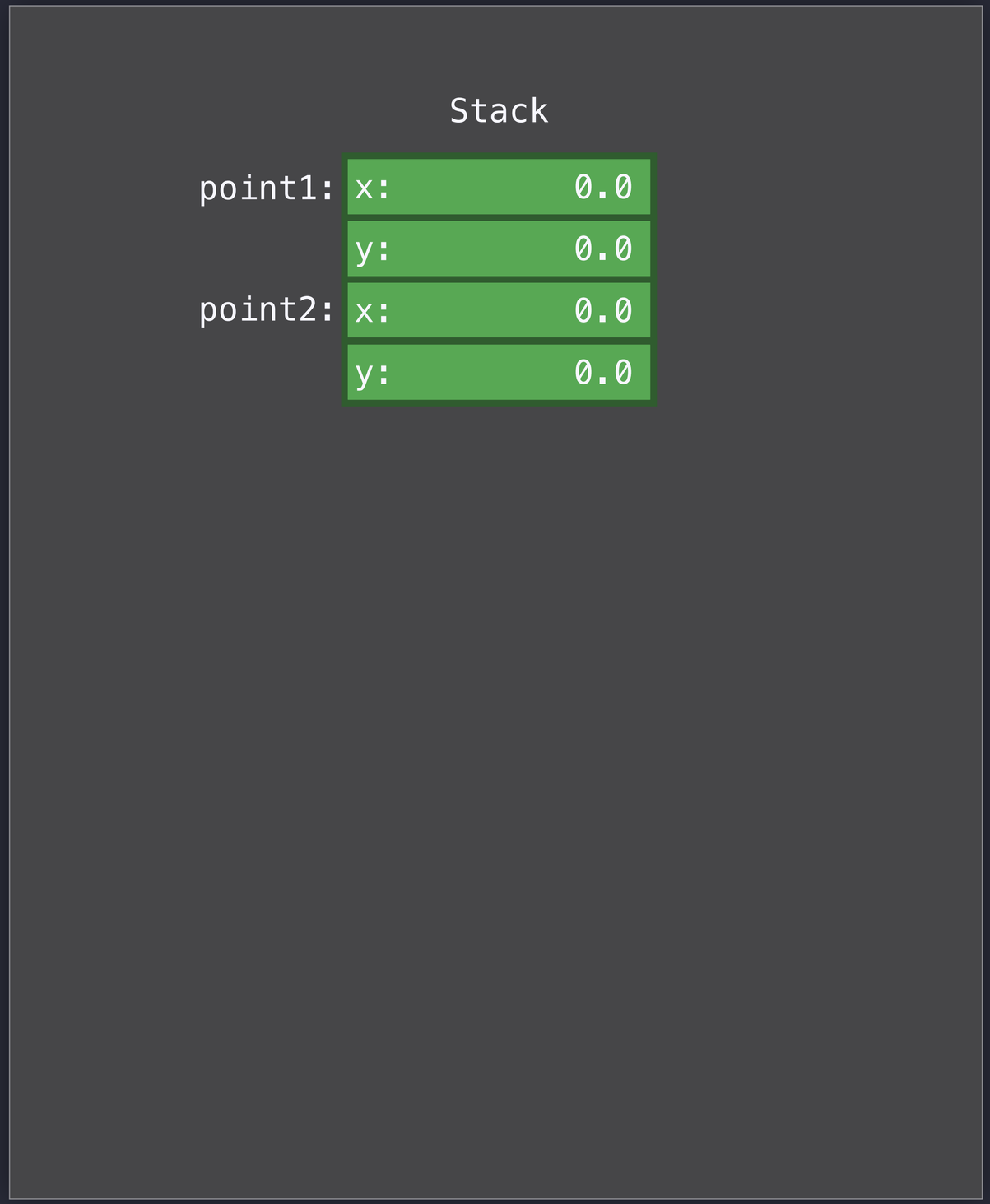


```
// Reference Counting
// Struct

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


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

```
// 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`
```

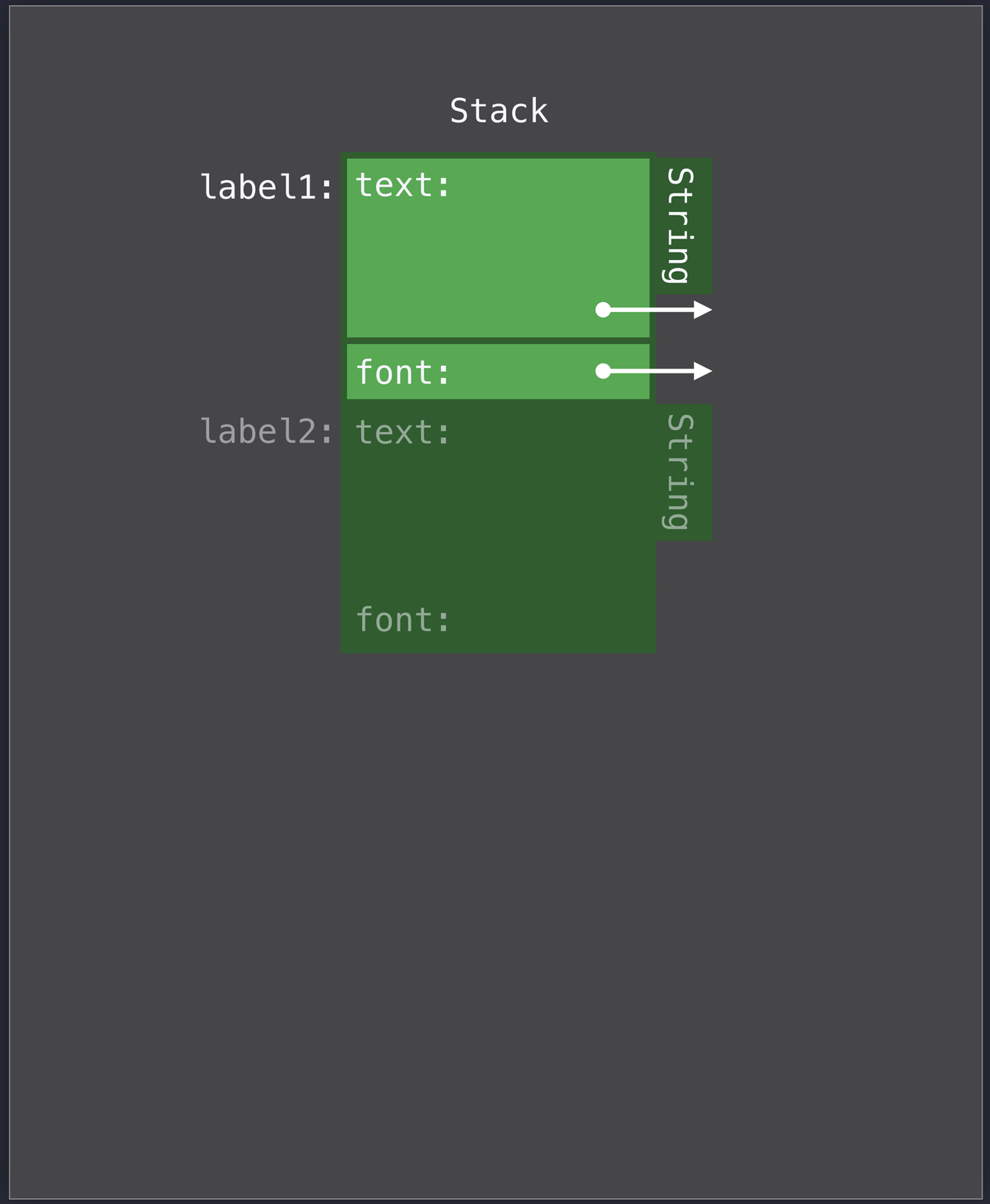


```swift
// Reference Counting
// Struct

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


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

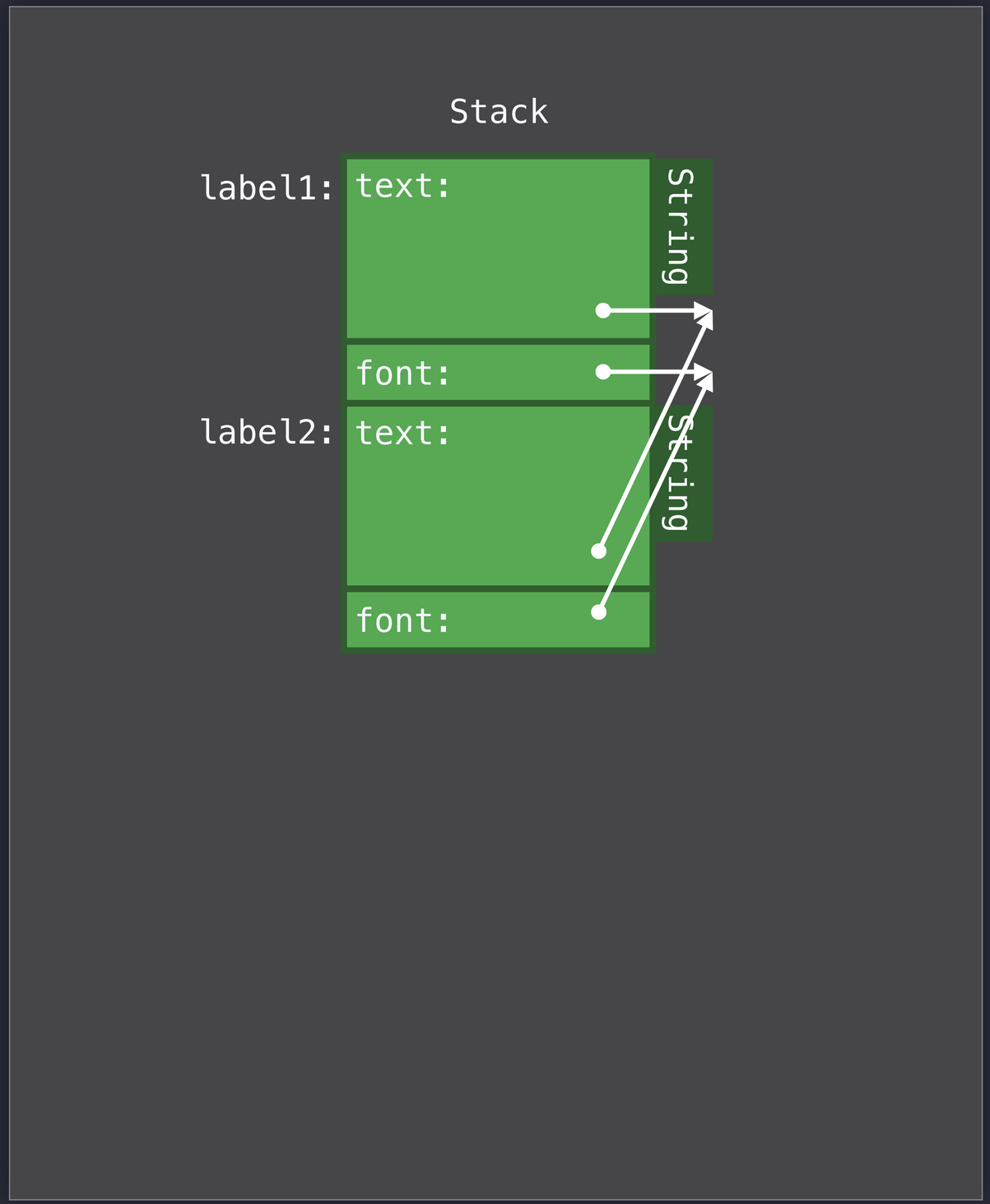


```
// Reference Counting
// Struct containing references

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

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

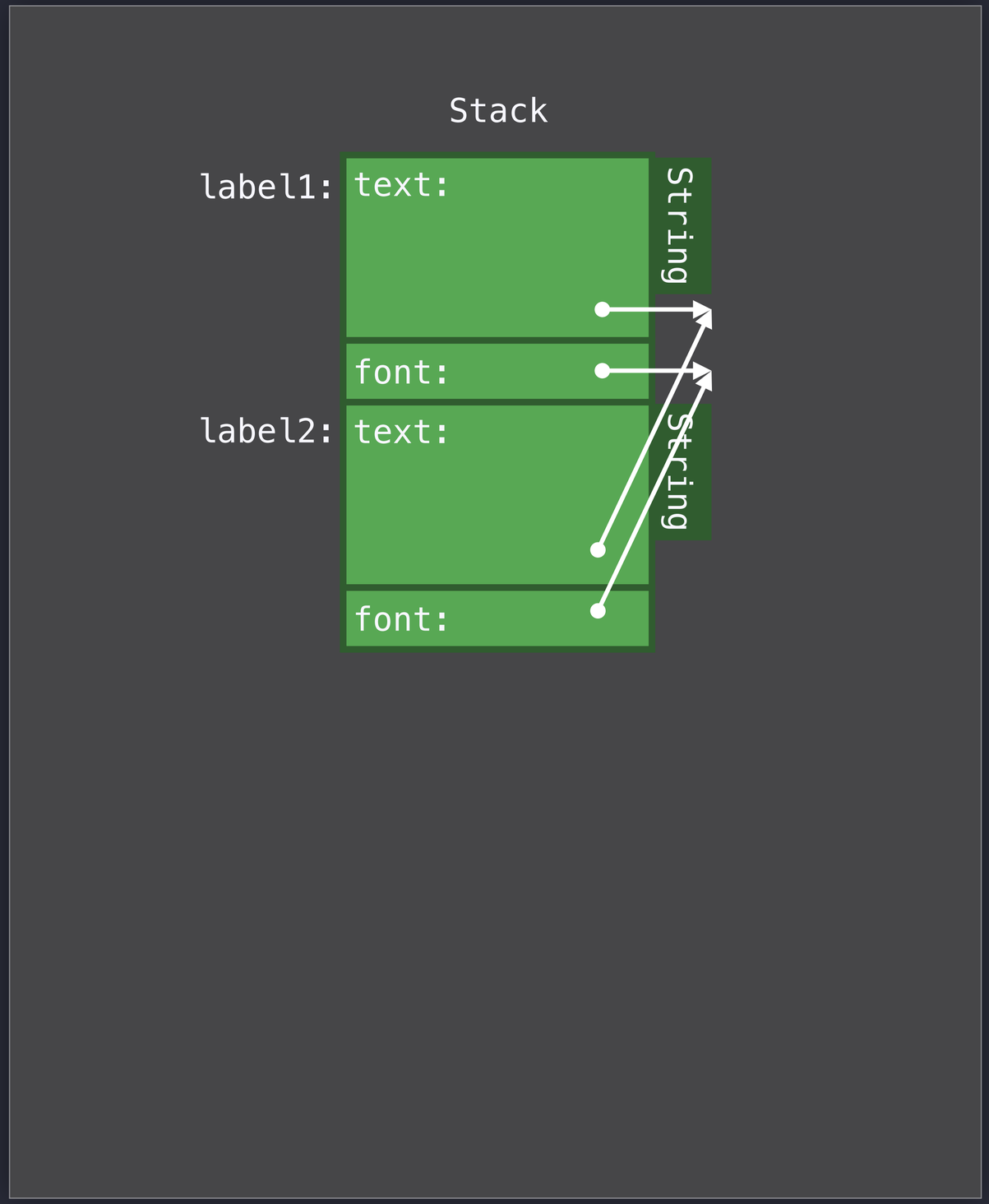
```swift
// Reference Counting
// Struct containing references

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

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



```
// 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)
```



Dimensions of Performance

Class

Allocation

Stack — Heap

Reference Counting

Less — More

Method Dispatch

Static — Dynamic

Dimensions of Performance

Struct

Allocation

Stack — Heap

Reference Counting

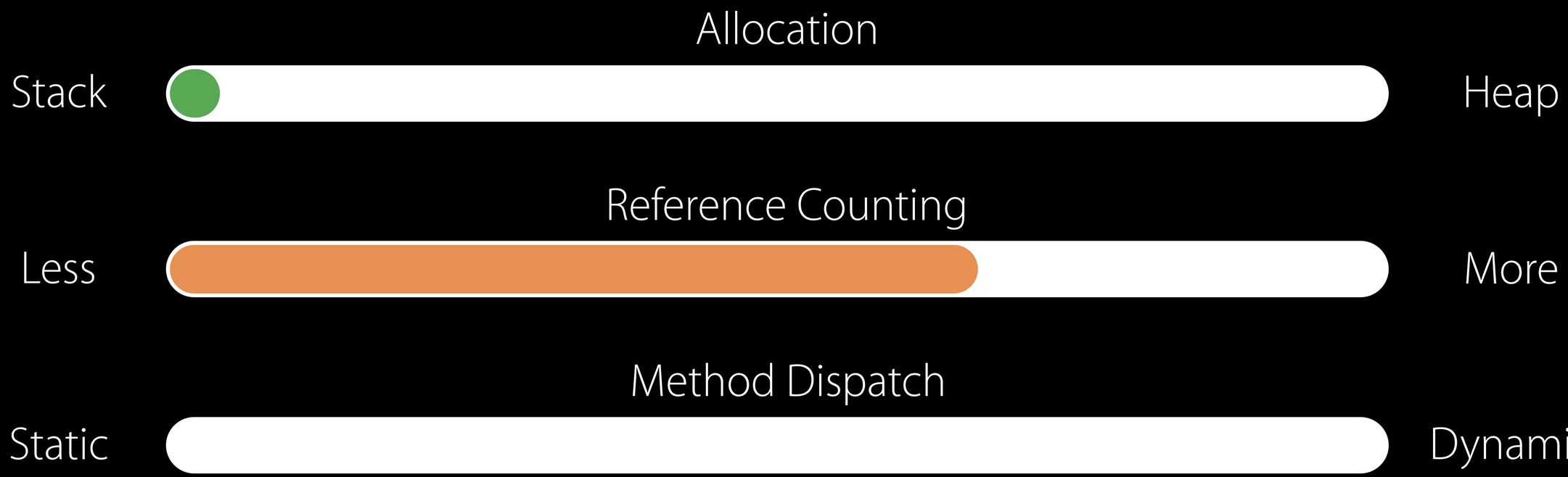
Less — More

Method Dispatch

Static — Dynamic

Dimensions of Performance

Struct containing a reference



Dimensions of Performance

Struct containing many references

Allocation

Stack — Heap

Reference Counting

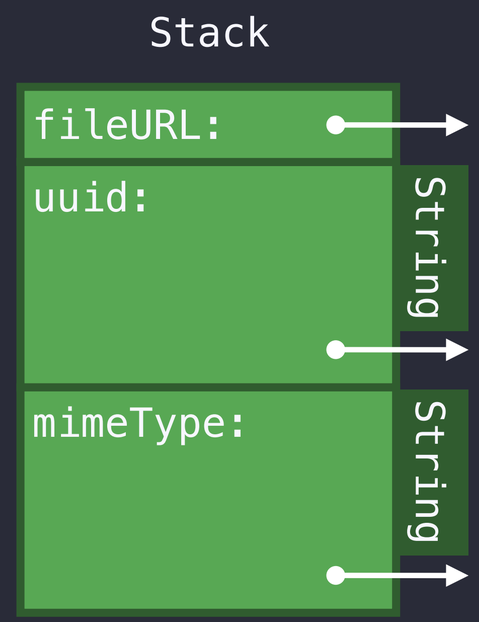
Less — More

Method Dispatch

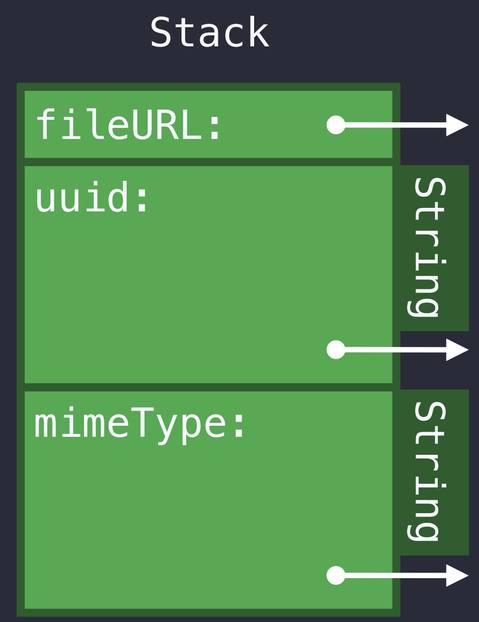
Static — Dynamic

```swift
// Modeling Techniques: Reference Counting

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

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

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

```
// Modeling Techniques: Reference Counting

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

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

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

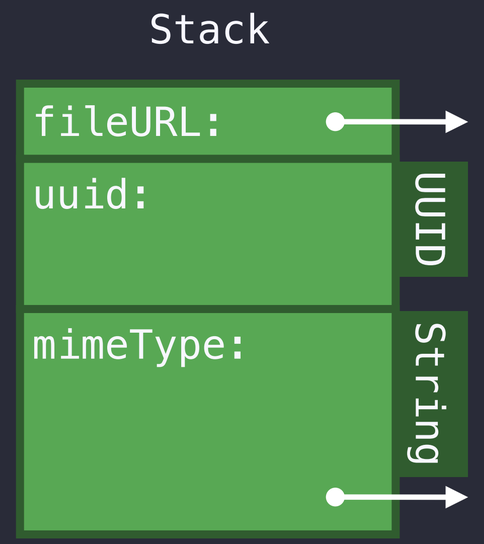


```swift
// Modeling Techniques: Reference Counting

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

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

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

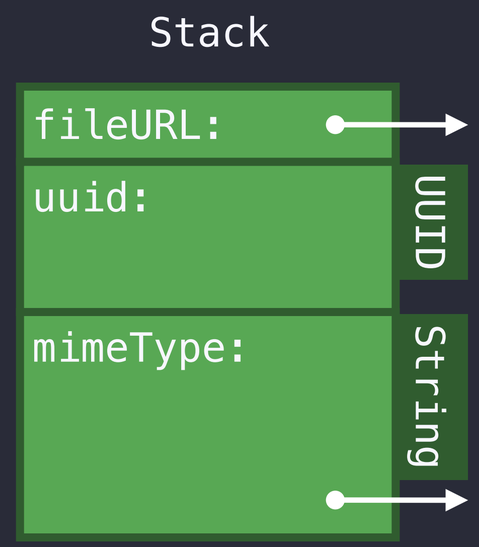


```
// Modeling Techniques: Reference Counting

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

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

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

NEW



```
// Modeling Techniques: Reference Counting

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

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

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



```
// Modeling Techniques: Reference Counting

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

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

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

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

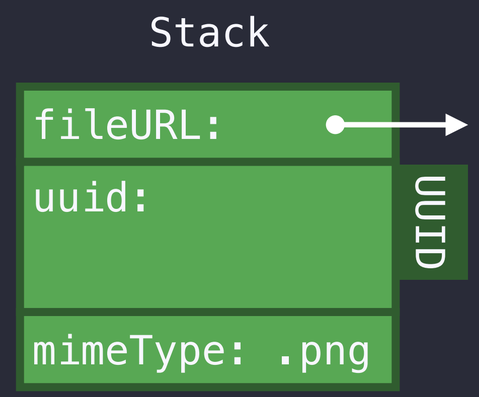
```
// Modeling Techniques: Reference Counting

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

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

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

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

```
// Modeling Techniques: Reference Counting

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

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

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

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

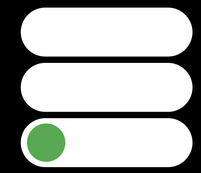
```swift
// Modeling Techniques: Reference Counting

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

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

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



Method Dispatch

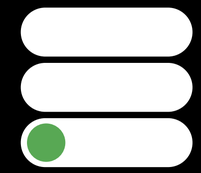
Method Dispatch

Static



Method Dispatch

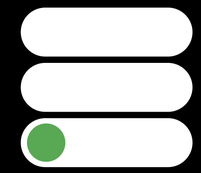
Static



Jump directly to implementation at run time

Method Dispatch

Static

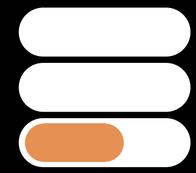


Jump directly to implementation at run time

Candidate for inlining and other optimizations

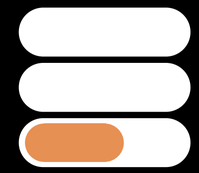
Method Dispatch

Dynamic



Method Dispatch

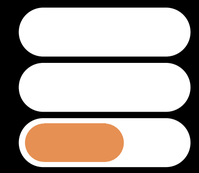
Dynamic



Look up implementation in table at run time

Method Dispatch

Dynamic

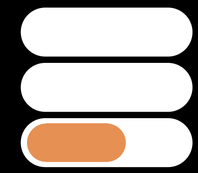


Look up implementation in table at run time

Then jump to implementation

Method Dispatch

Dynamic



Look up implementation in table at run time

Then jump to implementation

Prevents inlining and other optimizations

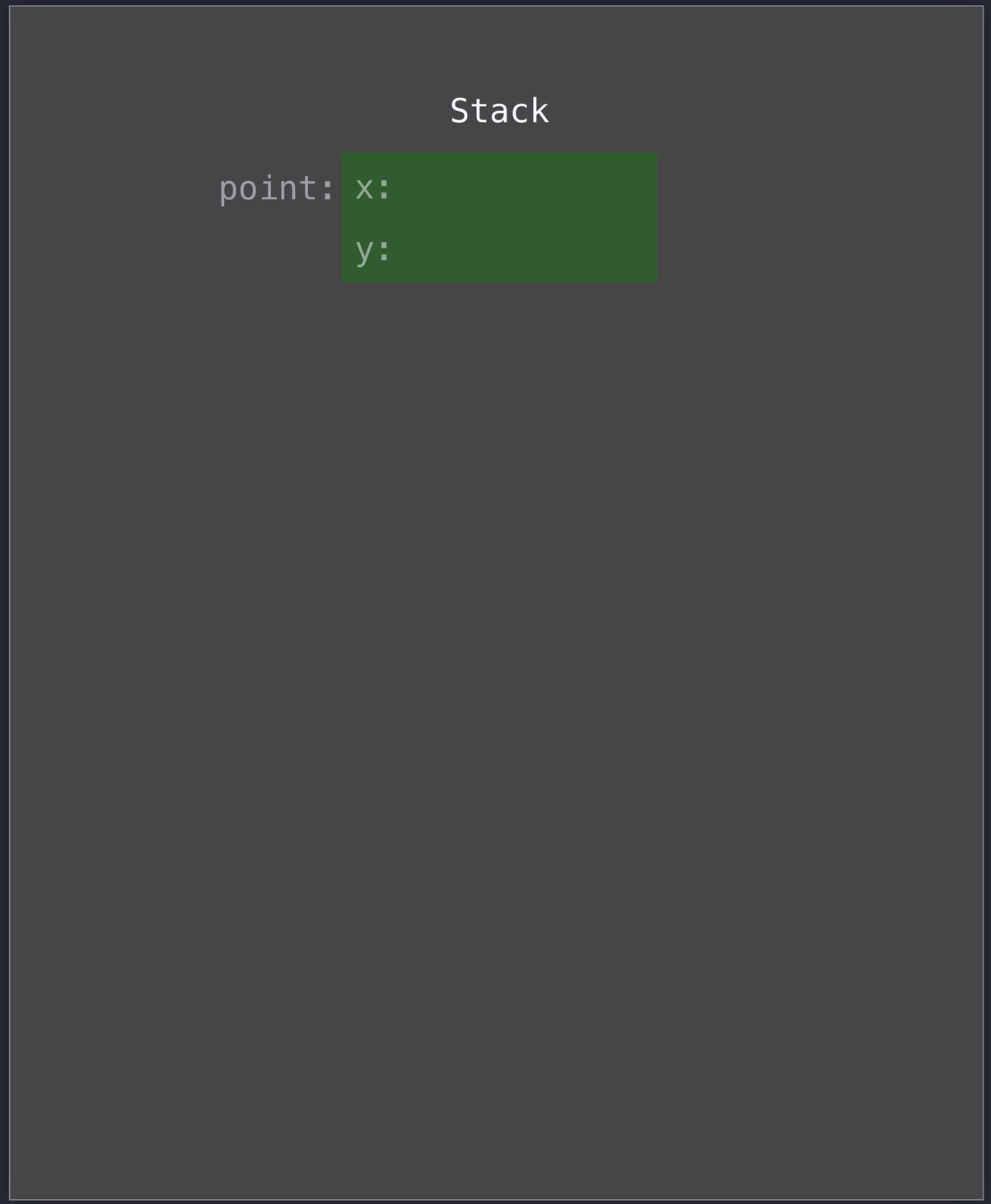
```
// Method Dispatch
// Struct (inlining)

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

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

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

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

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

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

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

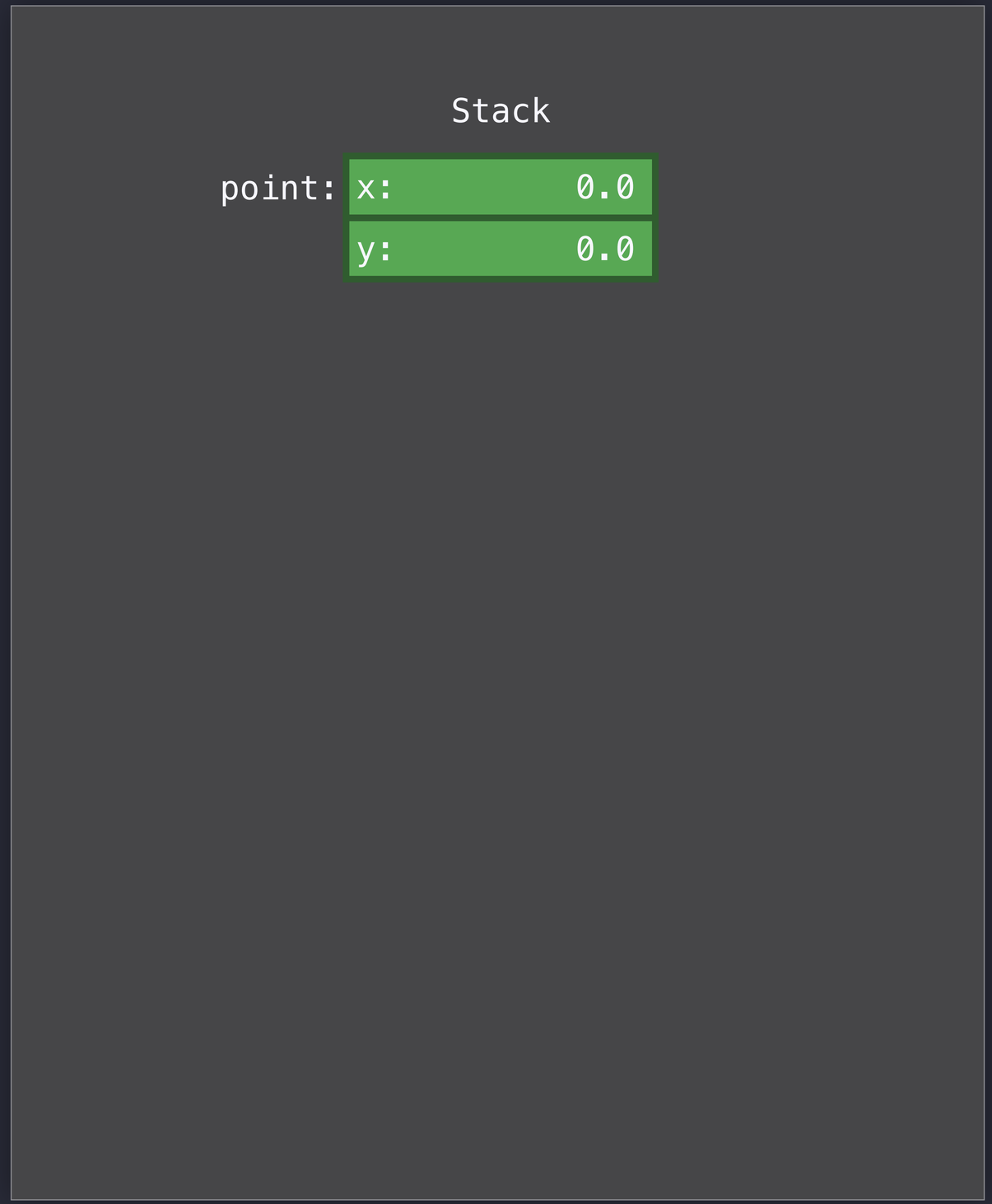
```
// Method Dispatch
// Struct (inlining)

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

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

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

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

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

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

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

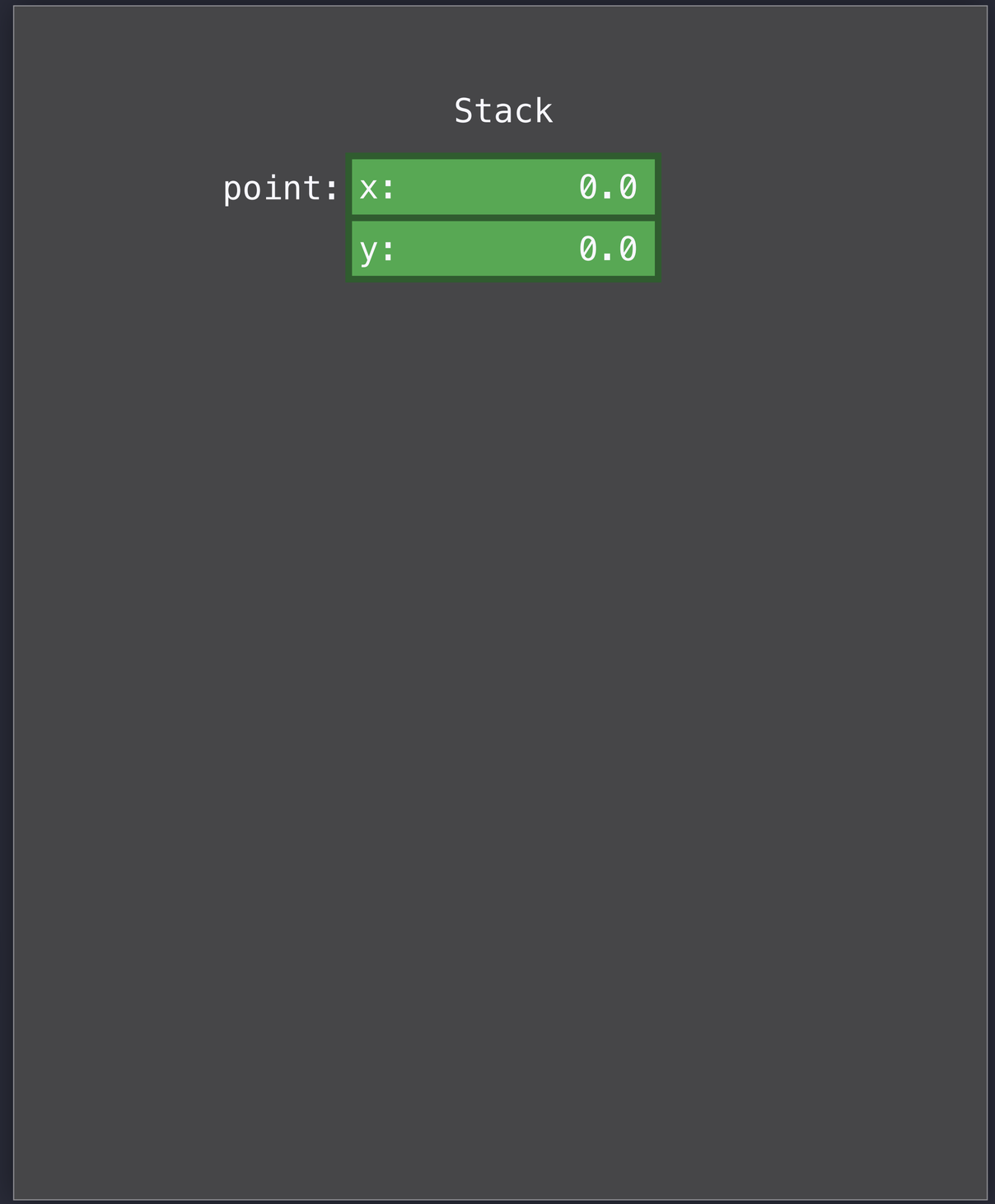
```swift
// Method Dispatch
// Struct (inlining)

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

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

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



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

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

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

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



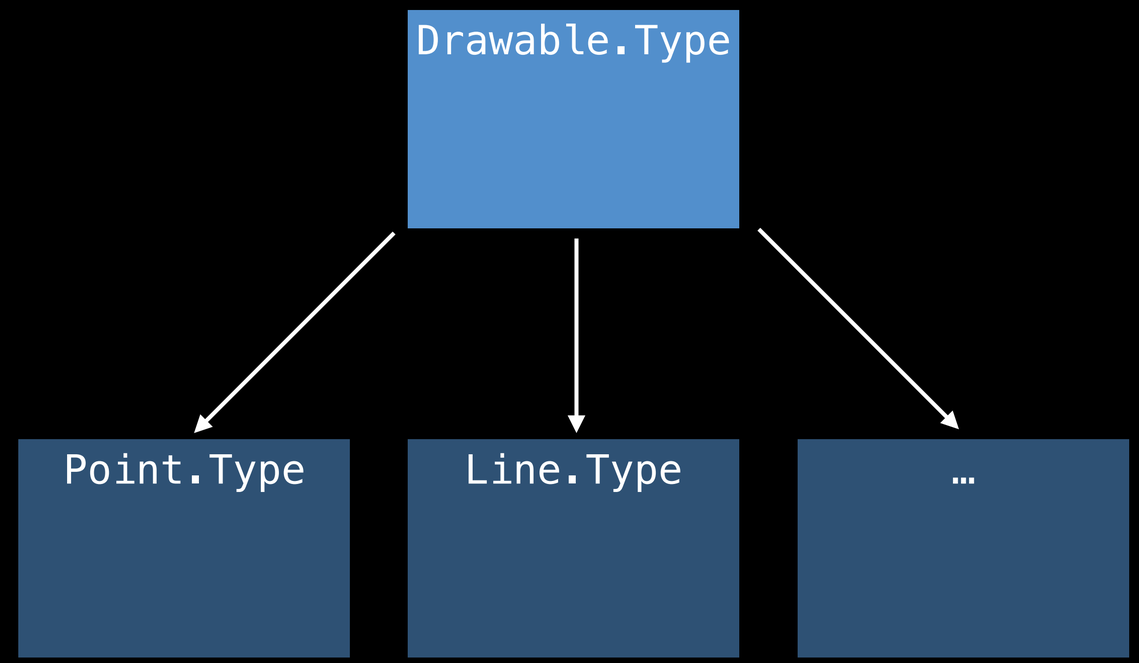
```
// Method Dispatch
// Struct (inlining)

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

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

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



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

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

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

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

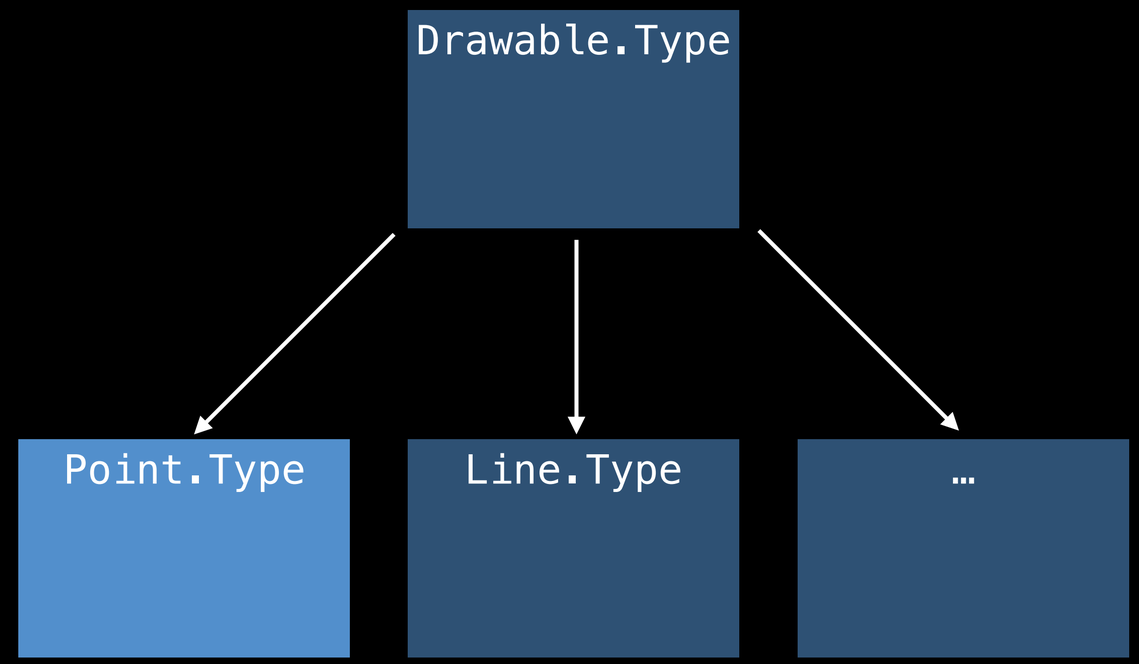
Inheritance-Based Polymorphism

```
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()
}
```



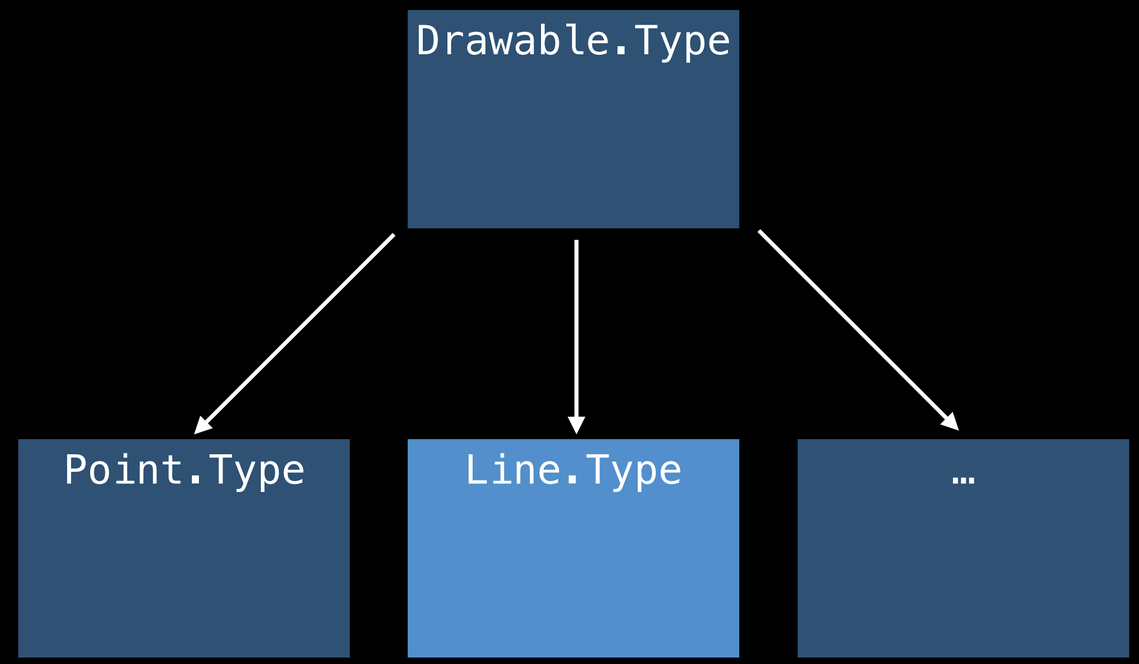
Inheritance-Based Polymorphism

```
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()
}
```



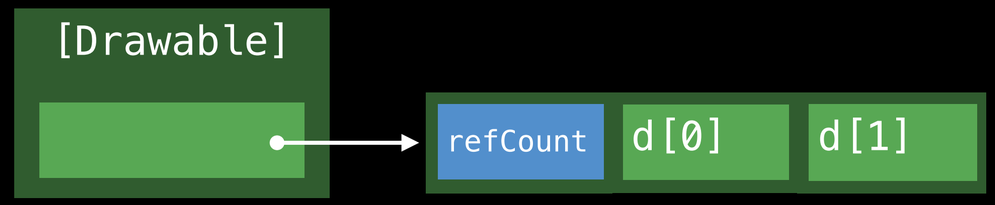
Inheritance-Based Polymorphism

```
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()
}
```



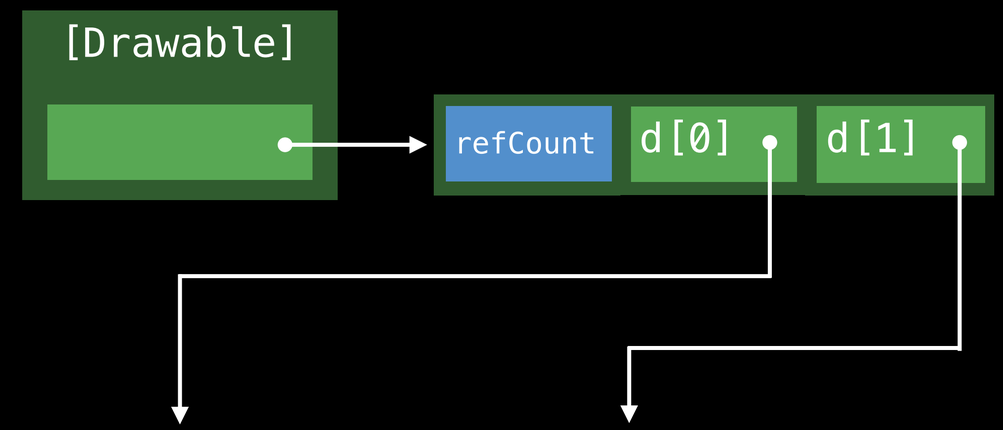
Polymorphism Through Reference Semantics

```
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()
}
```



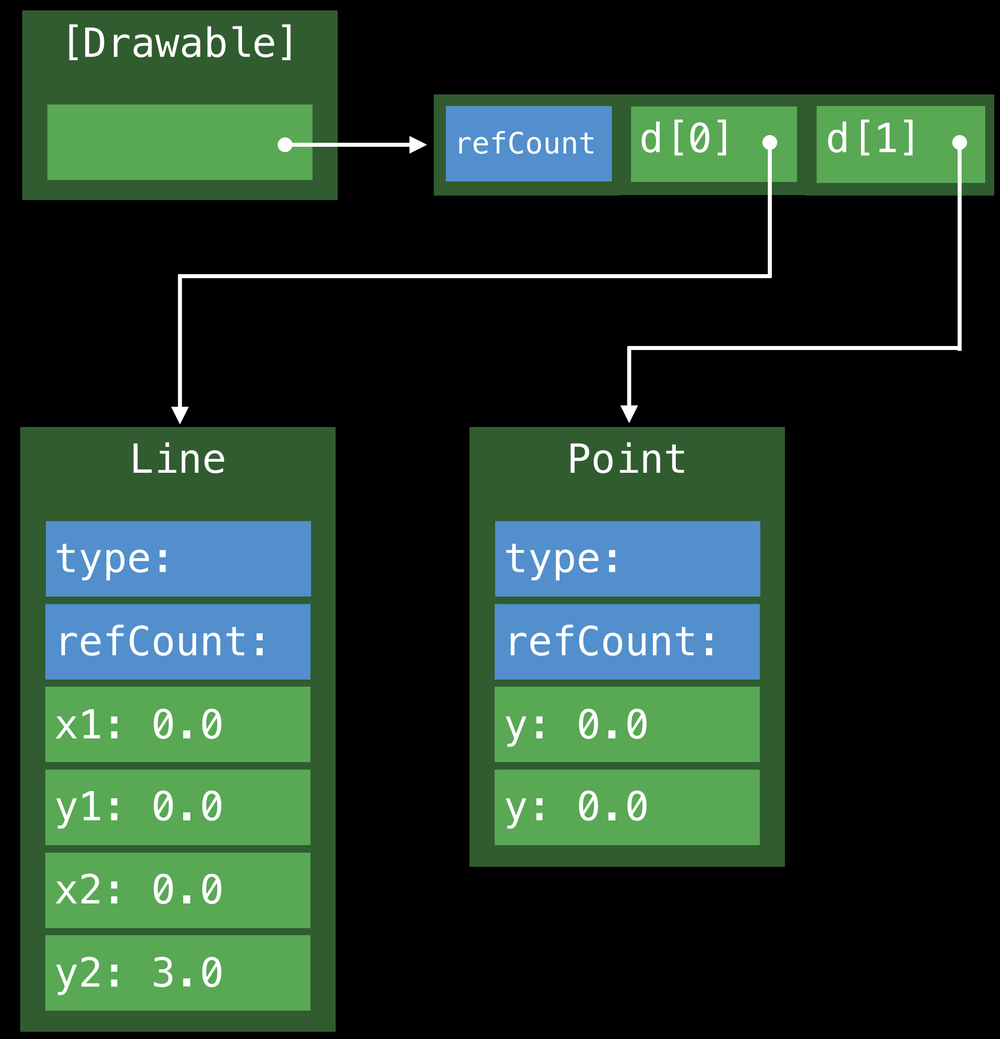
Polymorphism Through Reference Semantics

```
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()
}
```



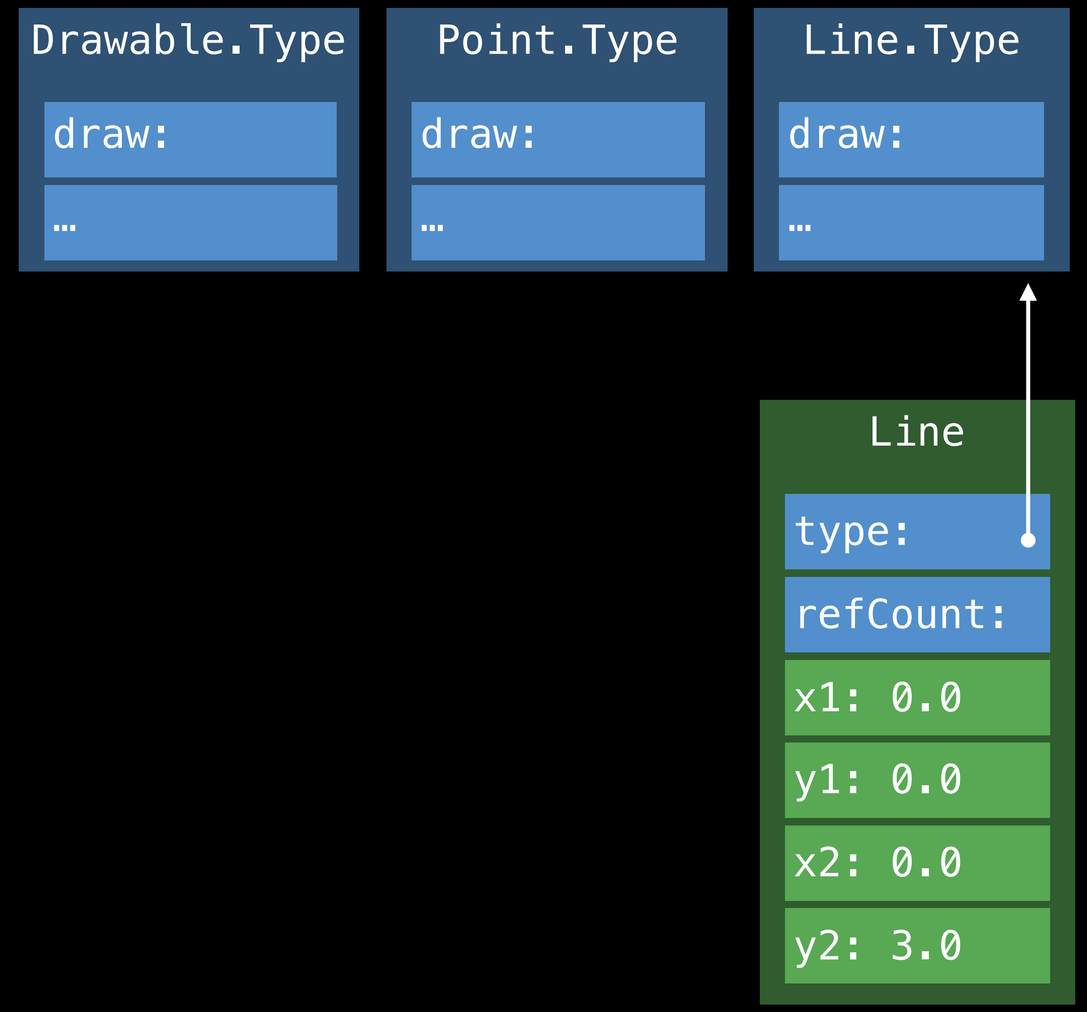
Polymorphism Through Reference Semantics

```
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()
}
```



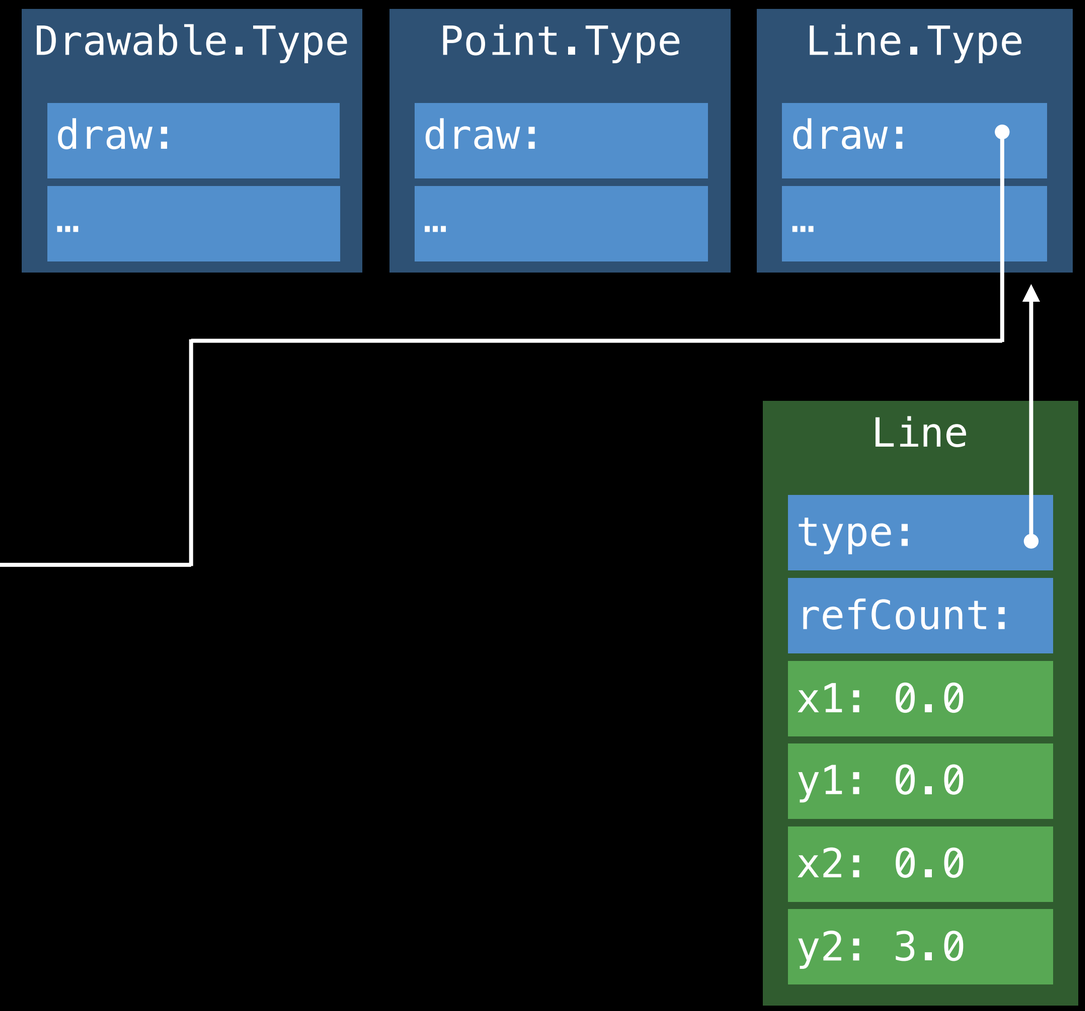
Polymorphism Through V-Table Dispatch

```
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() { … }
}

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)
}
```



Dimensions of Performance

Class

Allocation

Stack — Heap

Reference Counting

Less — More

Method Dispatch

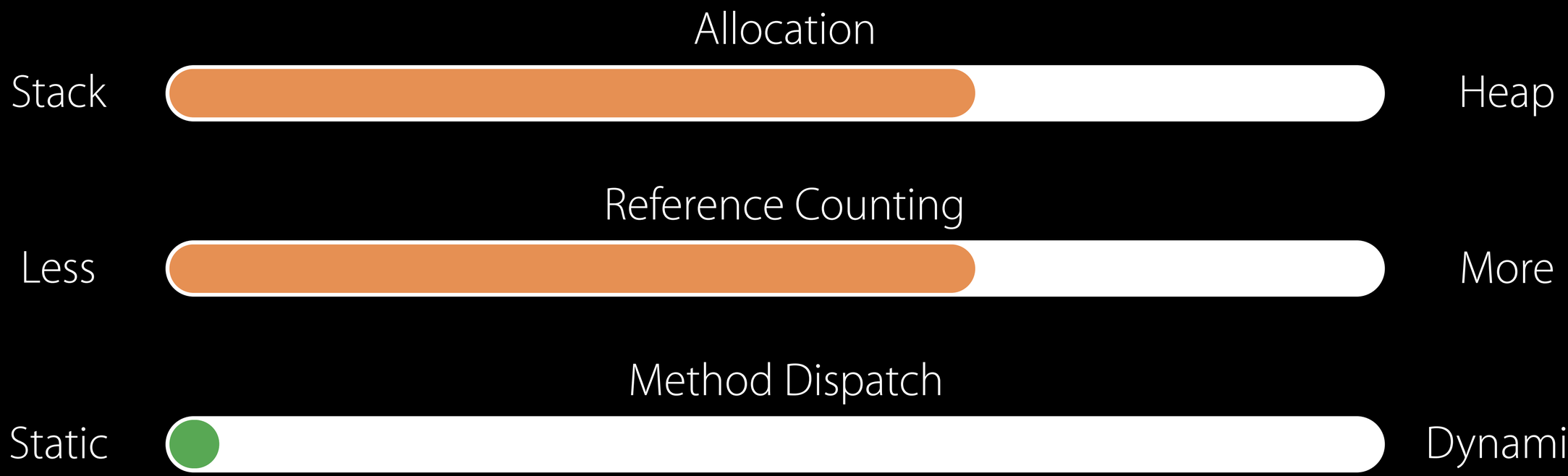
Static — Dynamic

Dimensions of Performance

Final Class

Dimensions of Performance

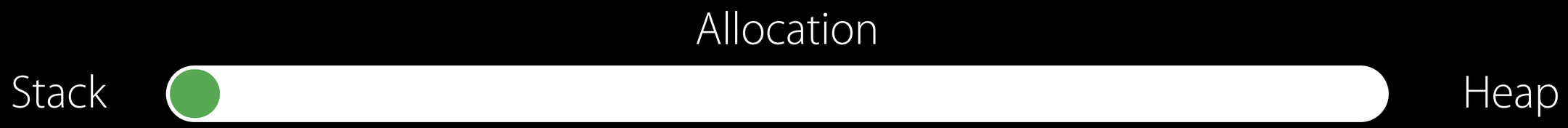
Final Class



Optimizing Swift Performance WWDC 2015

Dimensions of Performance

Struct



Protocol Types

Arnold Schwaighofer Swift Performance Engineer

Protocol Types

Polymorphism without inheritance or reference semantics

```
protocol Drawable { func draw() }

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

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

Protocol Types

Polymorphism without inheritance or reference semantics

```
protocol Drawable { func draw() }

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

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

Protocol Types

Polymorphism without inheritance or reference semantics

```
protocol Drawable { func draw() }

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

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

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

Protocol Types

Polymorphism without inheritance or reference semantics

```
protocol Drawable { func draw() }

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

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

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

Protocol Types

Polymorphism without inheritance or reference semantics

```
protocol Drawable { func draw() }

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

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

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

Protocol Types

Polymorphism without inheritance or reference semantics

```
protocol Drawable { func draw() }

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

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

Protocol Types

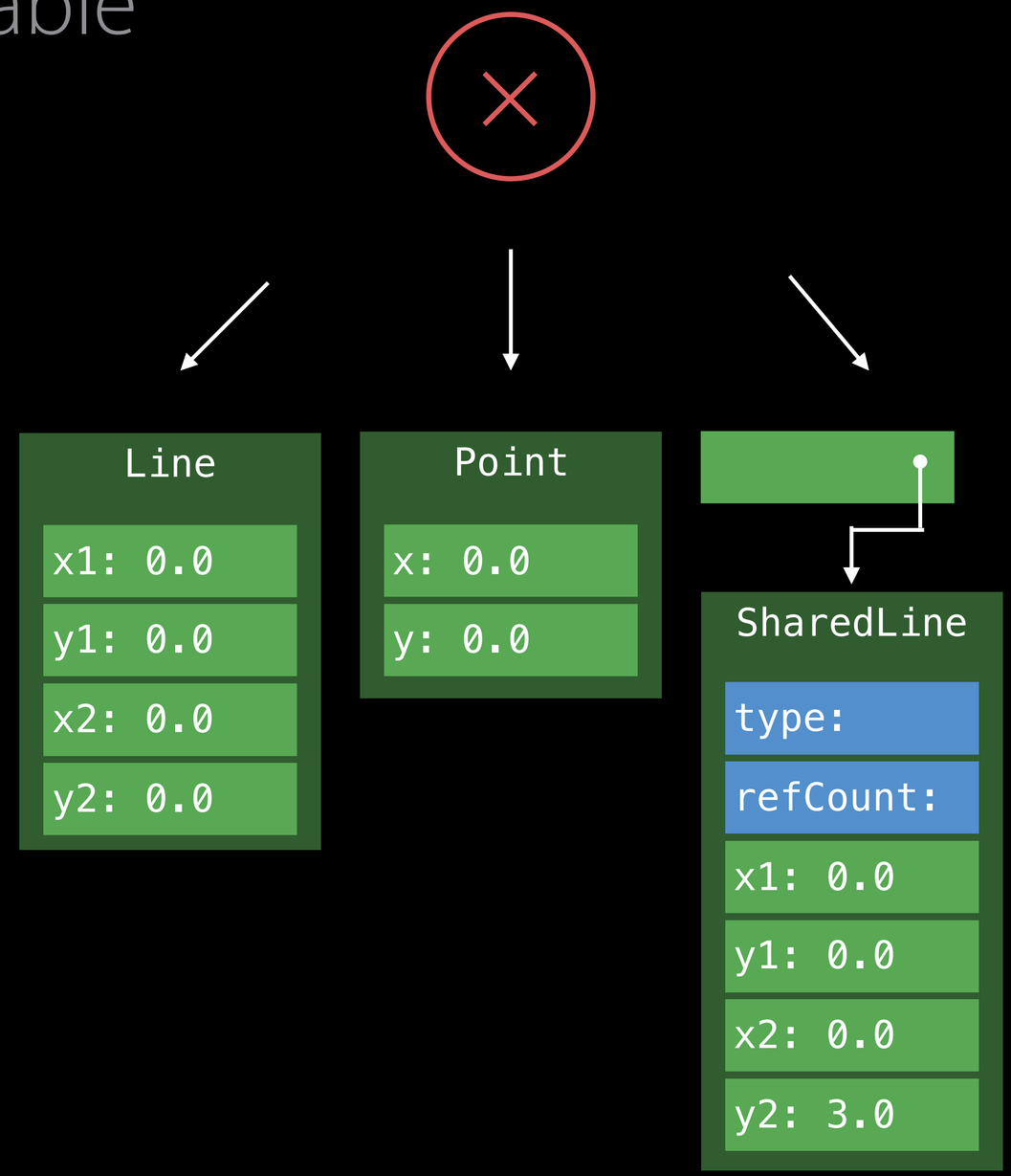
Polymorphism without inheritance or reference semantics

```
protocol Drawable { func draw() }

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

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

No Inheritance Relationship

Dynamic dispatch without a V-Table

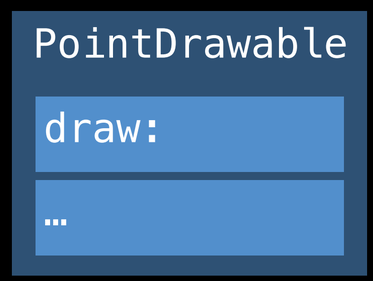
```
protocol Drawable { func draw() }

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()
}
```



The Protocol Witness Table (PWT)

Dynamic dispatch without a V-Table

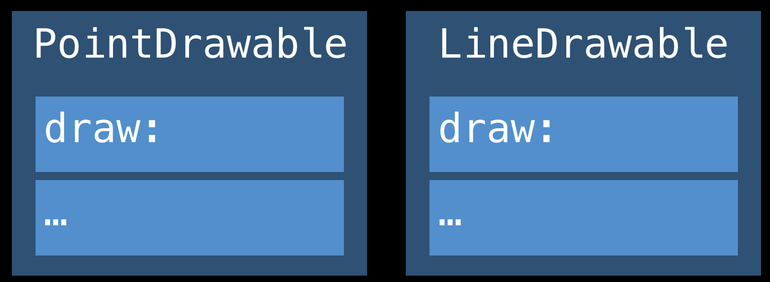
```
protocol Drawable {
    func draw()
}
struct Point : Drawable {
    func draw() { … }
}
struct Line : Drawable {
    func draw() { … }
}
```



The Protocol Witness Table (PWT)

Dynamic dispatch without a V-Table

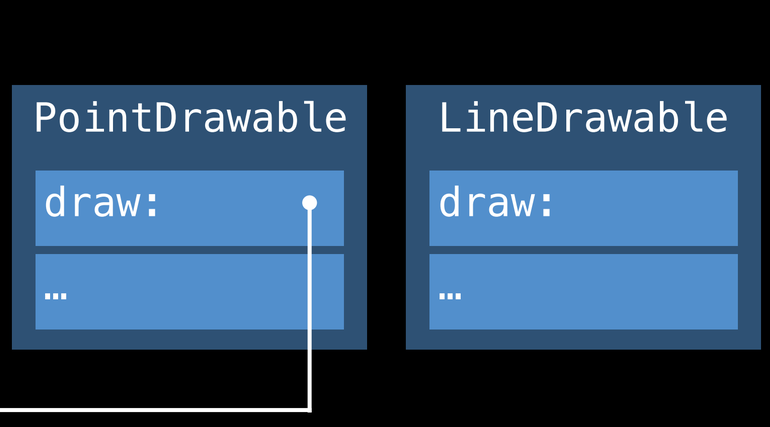
```
protocol Drawable {
   func draw()
}
struct Point : Drawable {
   func draw() { … }
}
struct Line : Drawable {
   func draw() { … }
}
```



The Protocol Witness Table (PWT)

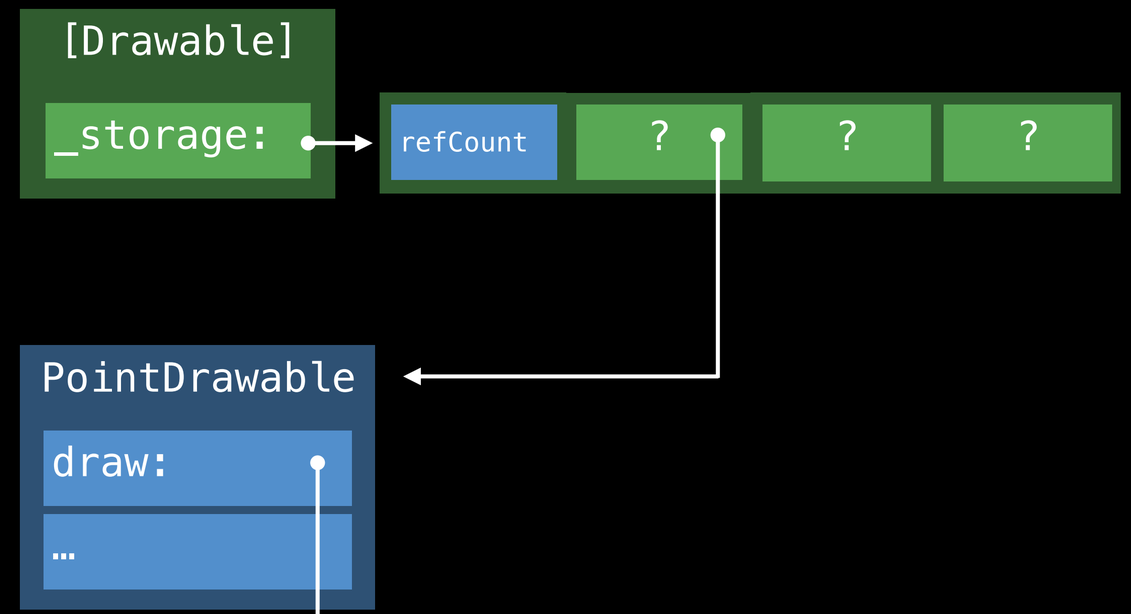
Dynamic dispatch without a V-Table

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



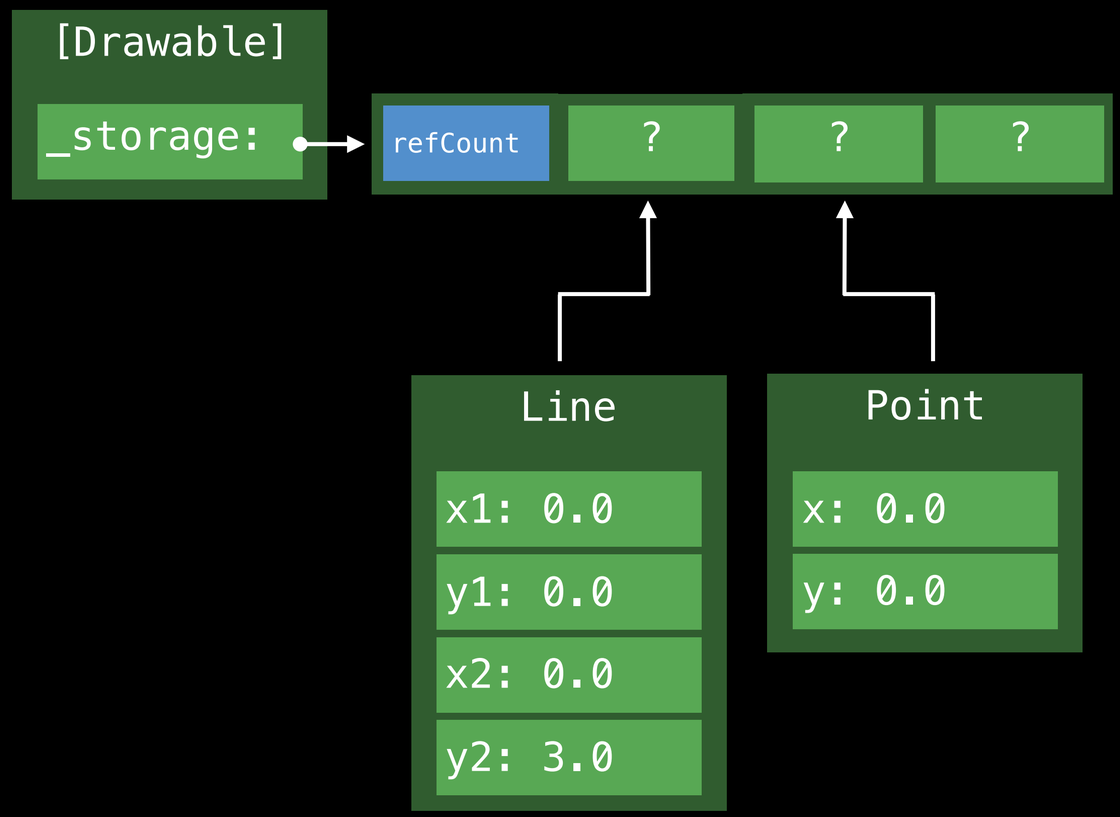
How to Look Up the Protocol Witness Table?

```
protocol Drawable { func draw() }

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

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



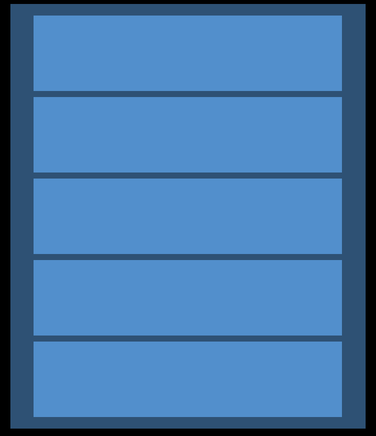
How to Store Values Uniformly?

```
protocol Drawable { func draw() }

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

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

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



The Existential Container

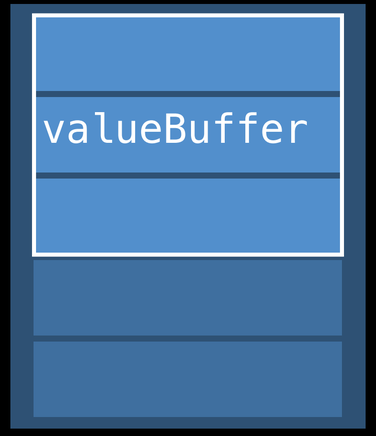
Boxing values of protocol types



The Existential Container

Boxing values of protocol types

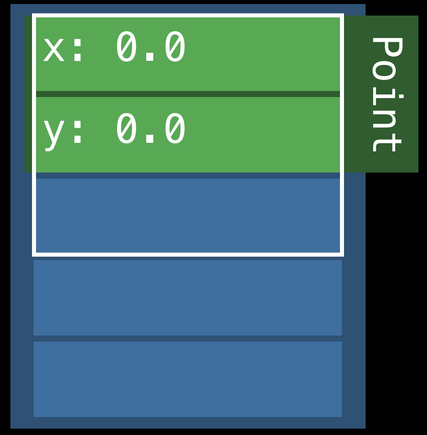
Inline Value Buffer: currently 3 words



The Existential Container

Boxing values of protocol types

Inline Value Buffer: currently 3 words

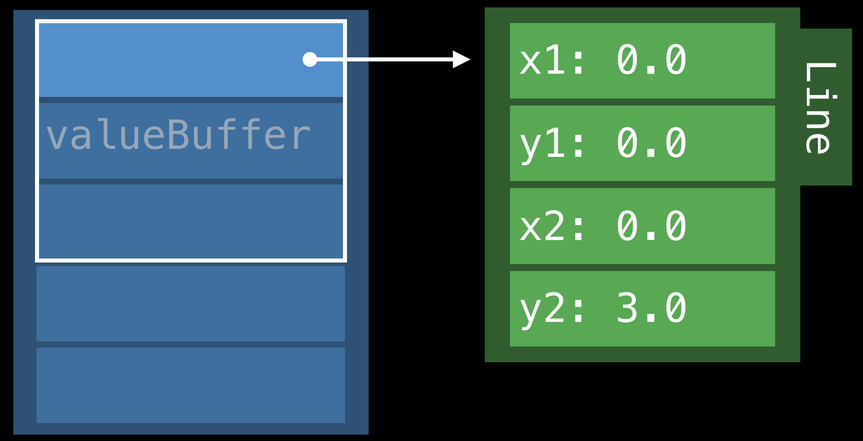


The Existential Container

Boxing values of protocol types

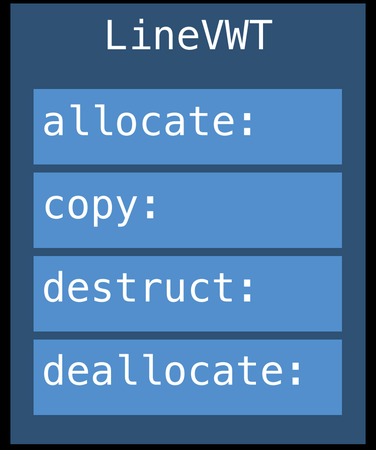
Inline Value Buffer: currently 3 words

Large values stored on heap



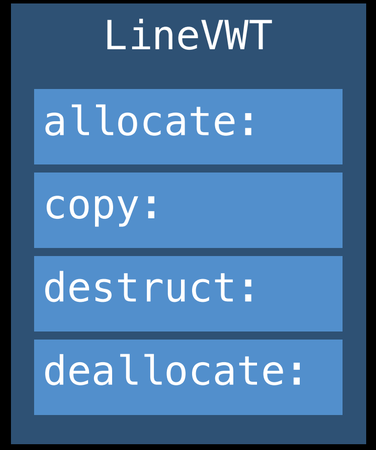
The Value Witness Table (VWT)

Allocation, Copy, Destruction of any Value



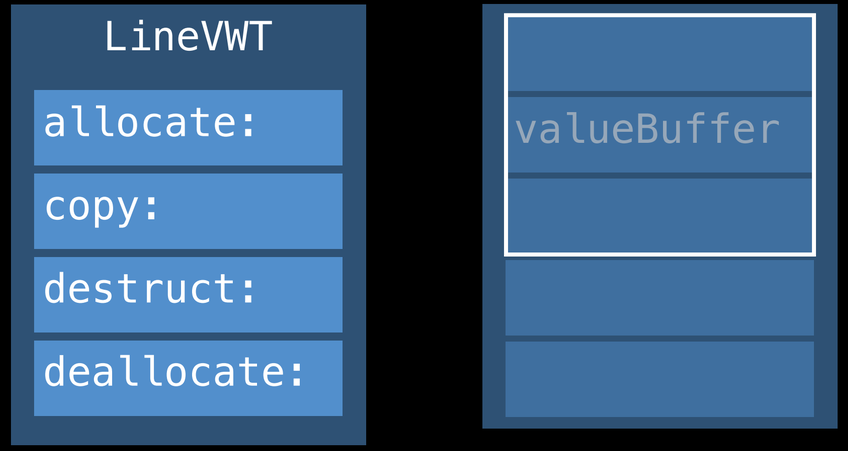
The Value Witness Table (VWT)

Allocation, Copy, Destruction of any Value



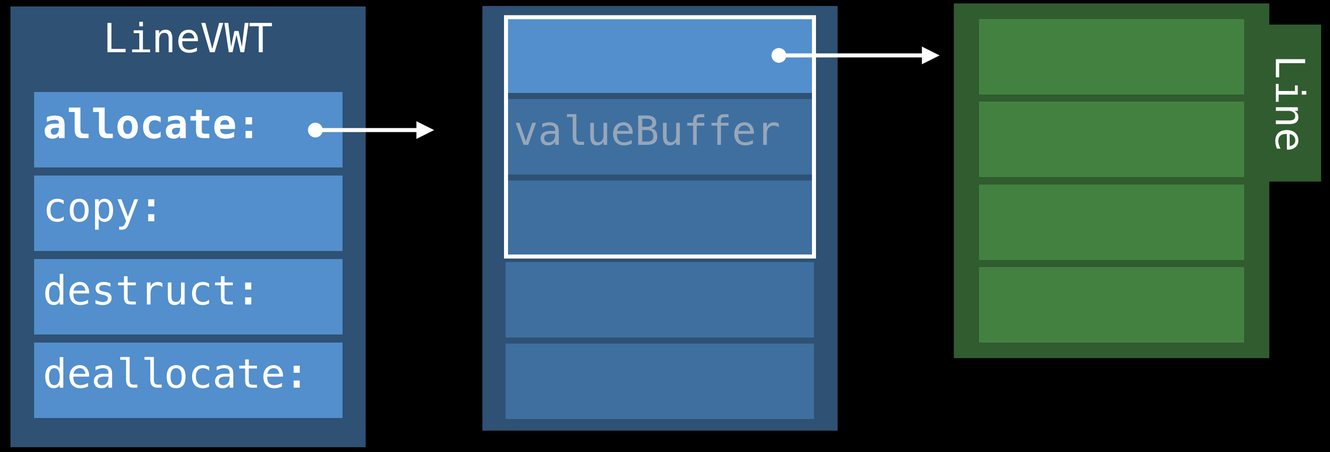
The Value Witness Table (VWT)

Allocation, Copy, Destruction of any Value



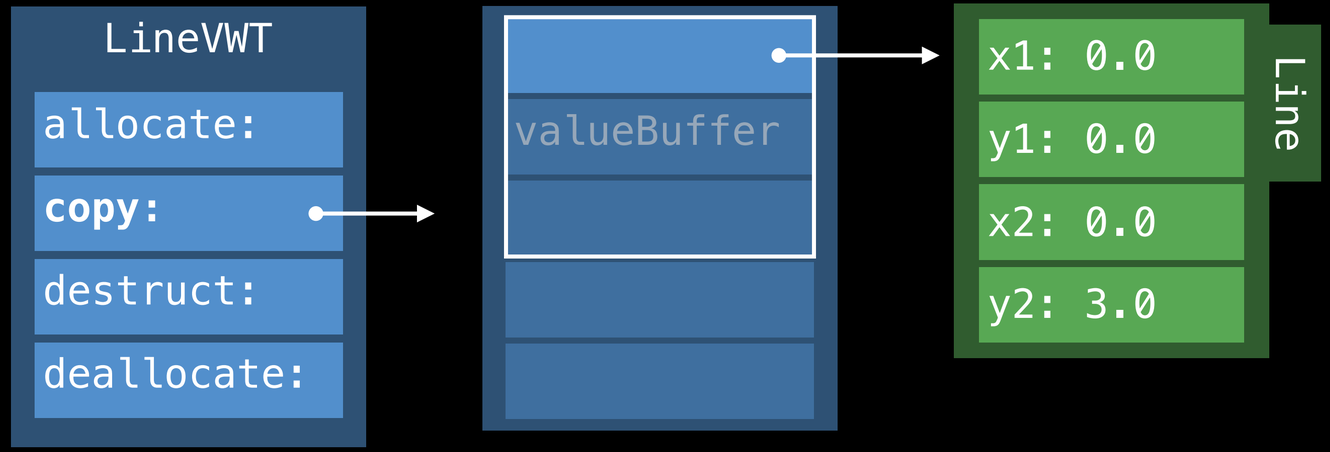
The Value Witness Table (VWT)

Allocation, Copy, Destruction of any Value



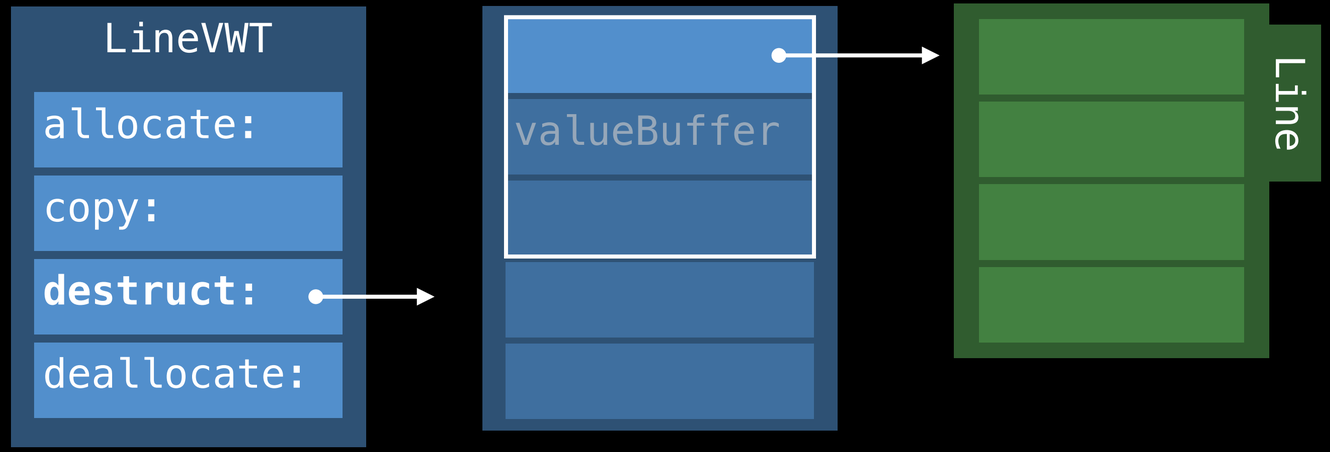
The Value Witness Table (VWT)

Allocation, Copy, Destruction of any Value



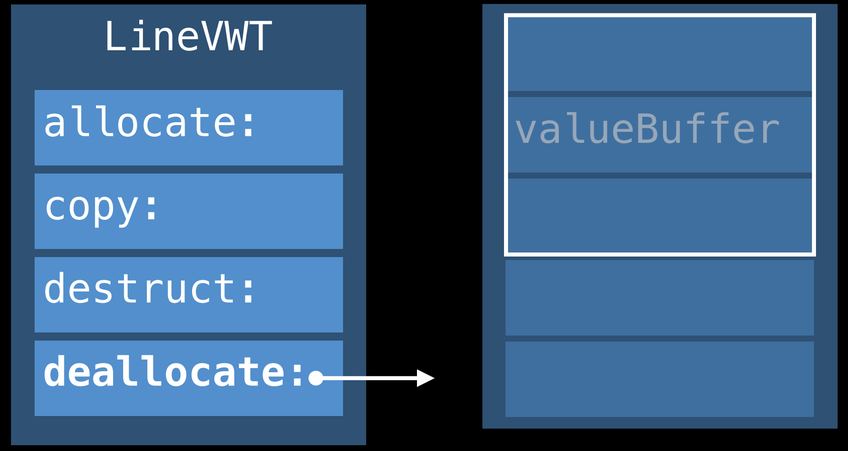
The Value Witness Table (VWT)

Allocation, Copy, Destruction of any Value



The Value Witness Table (VWT)

Allocation, Copy, Destruction of any Value



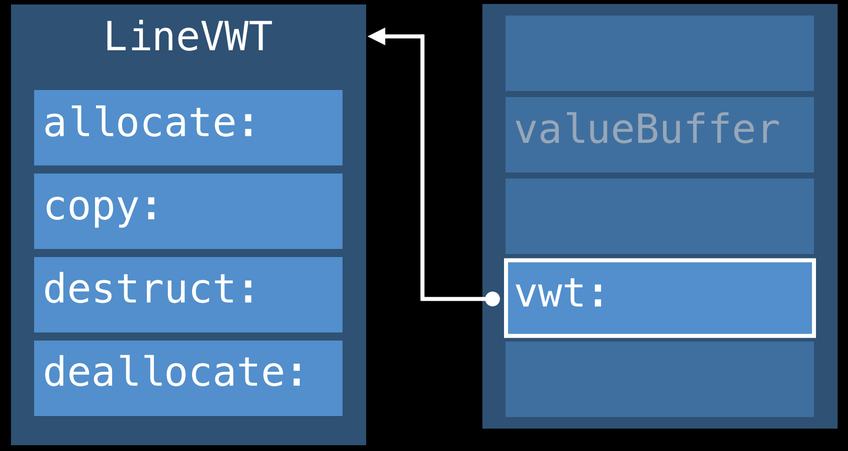
The Existential Container

Boxing values of protocol types

Inline Value Buffer: currently 3 words

Large values stored on heap

Reference to Value Witness Table



The Existential Container

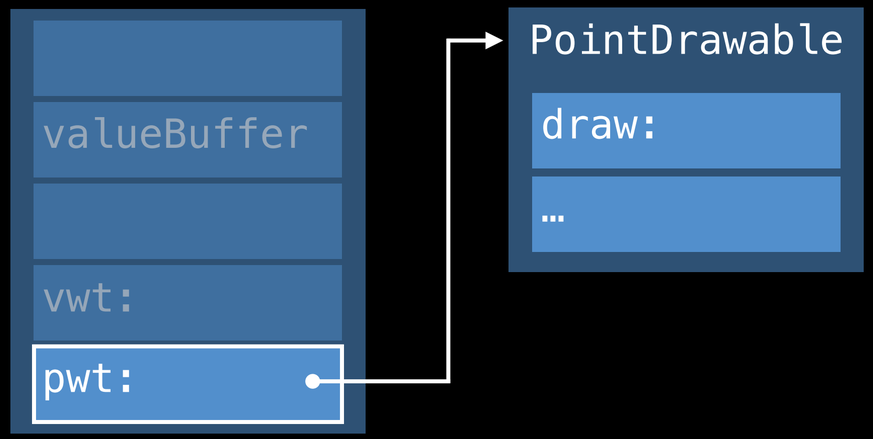
Boxing values of protocol types

Inline Value Buffer: currently 3 words

Large values stored on heap

Reference to Value Witness Table

Reference to Protocol Witness Table



```
// 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) {
```

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

```
let local = val
```

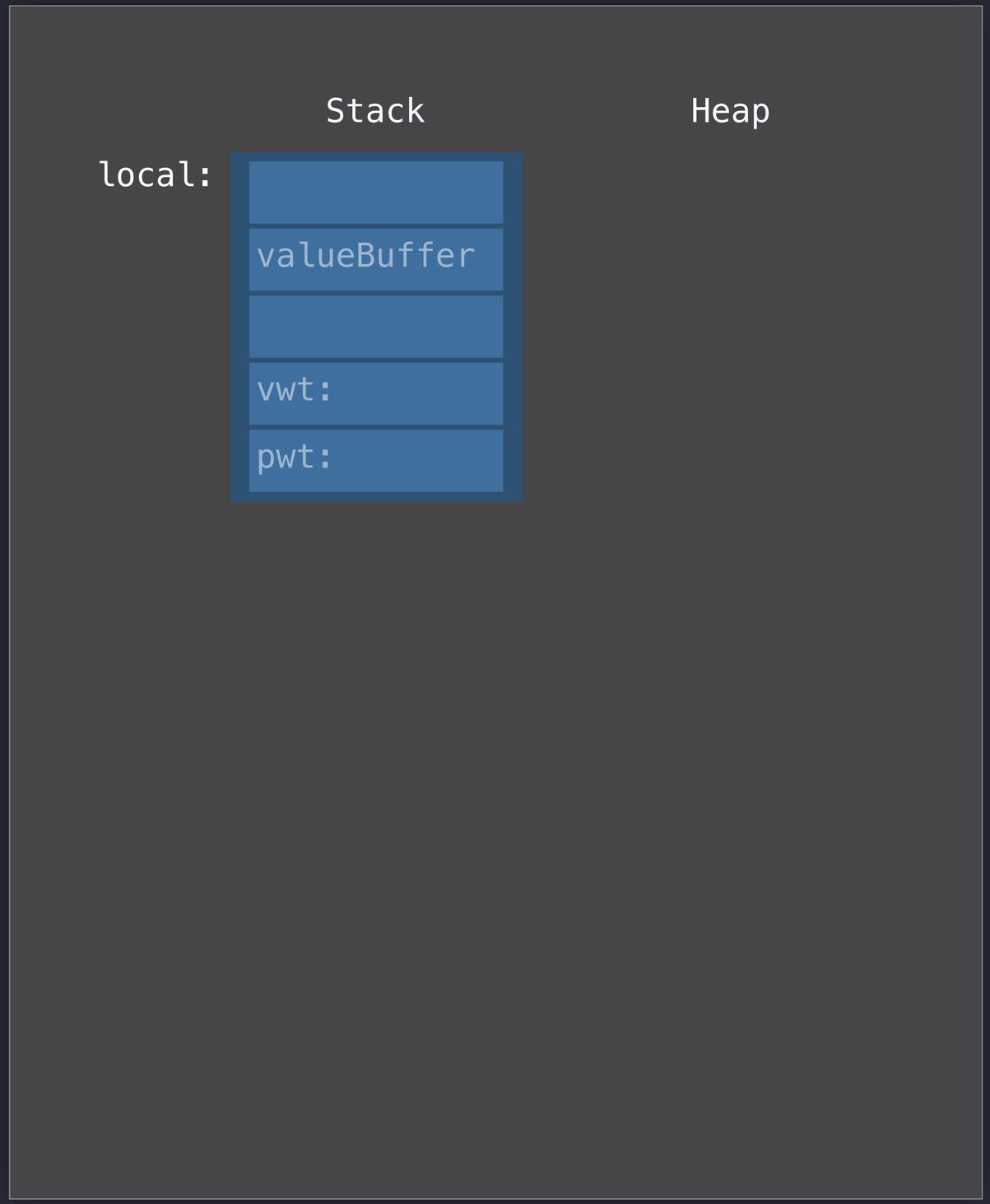
```
// Generated code
func drawACopy(val: ExistContDrawable) {
```

```
// 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) {
```

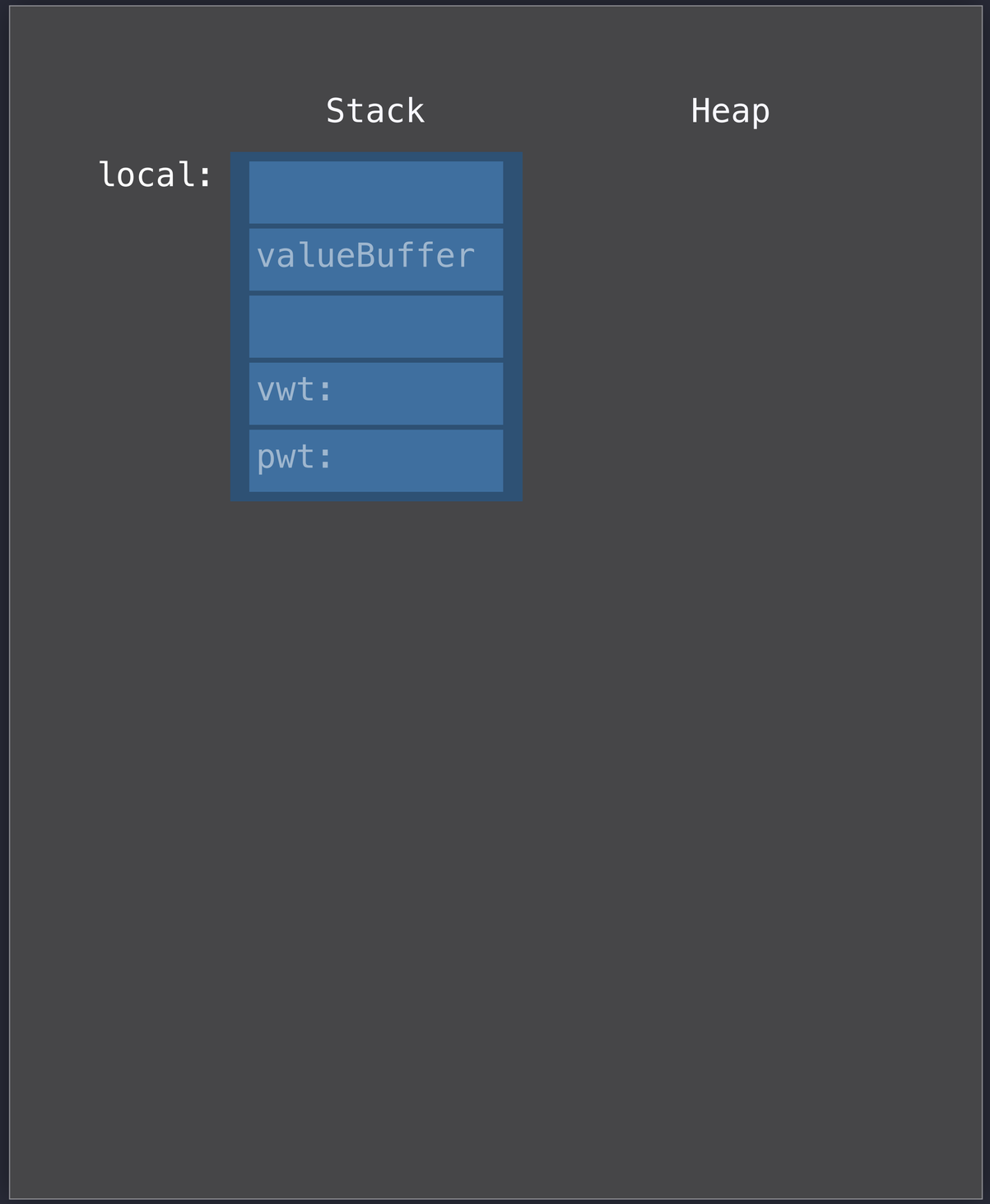
```swift
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
    local.draw()
}
let val : Drawable = Point()
drawACopy(val)
```

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



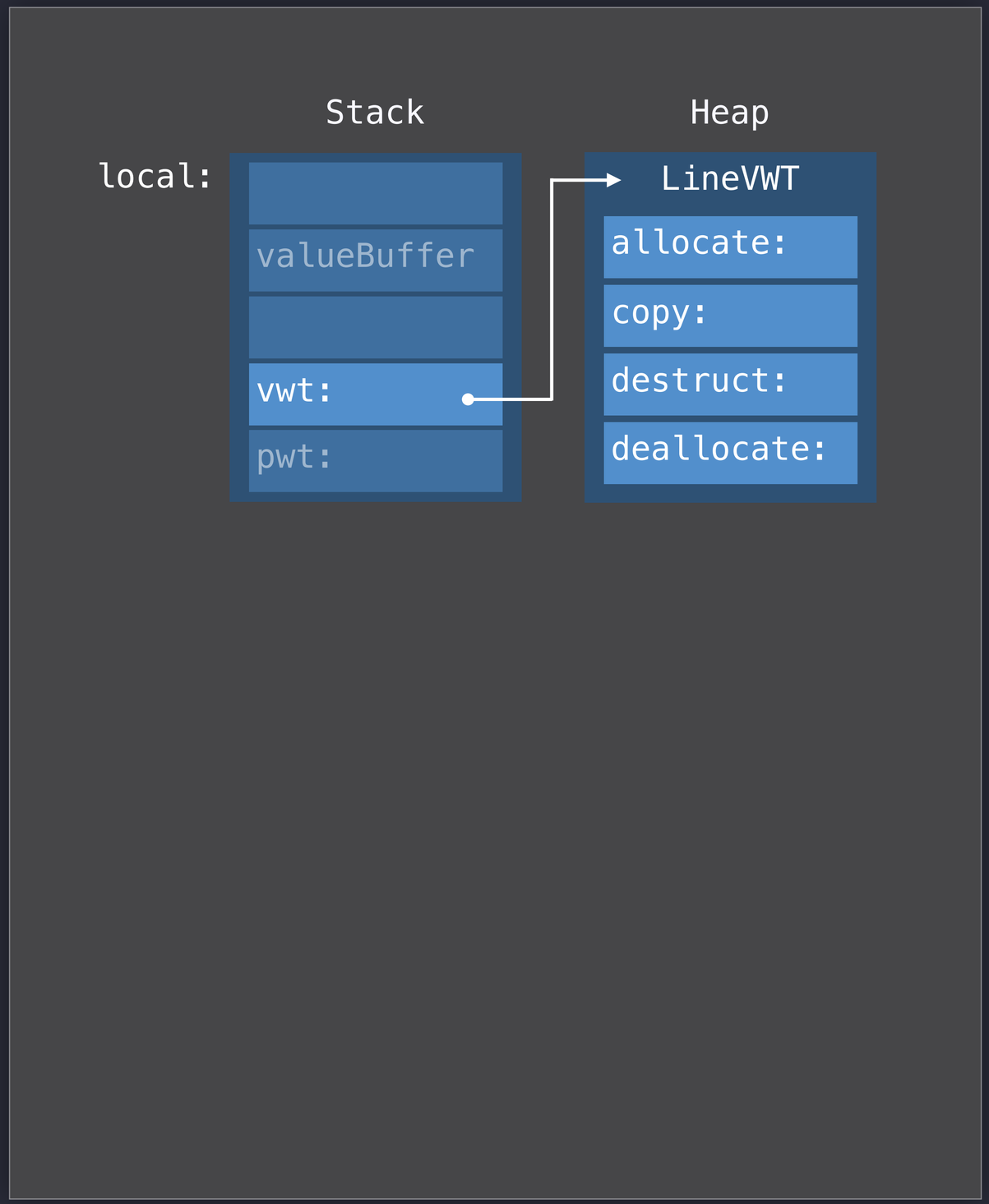
```
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
    local.draw()
}
let val : Drawable = Point()
drawACopy(val)
```

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



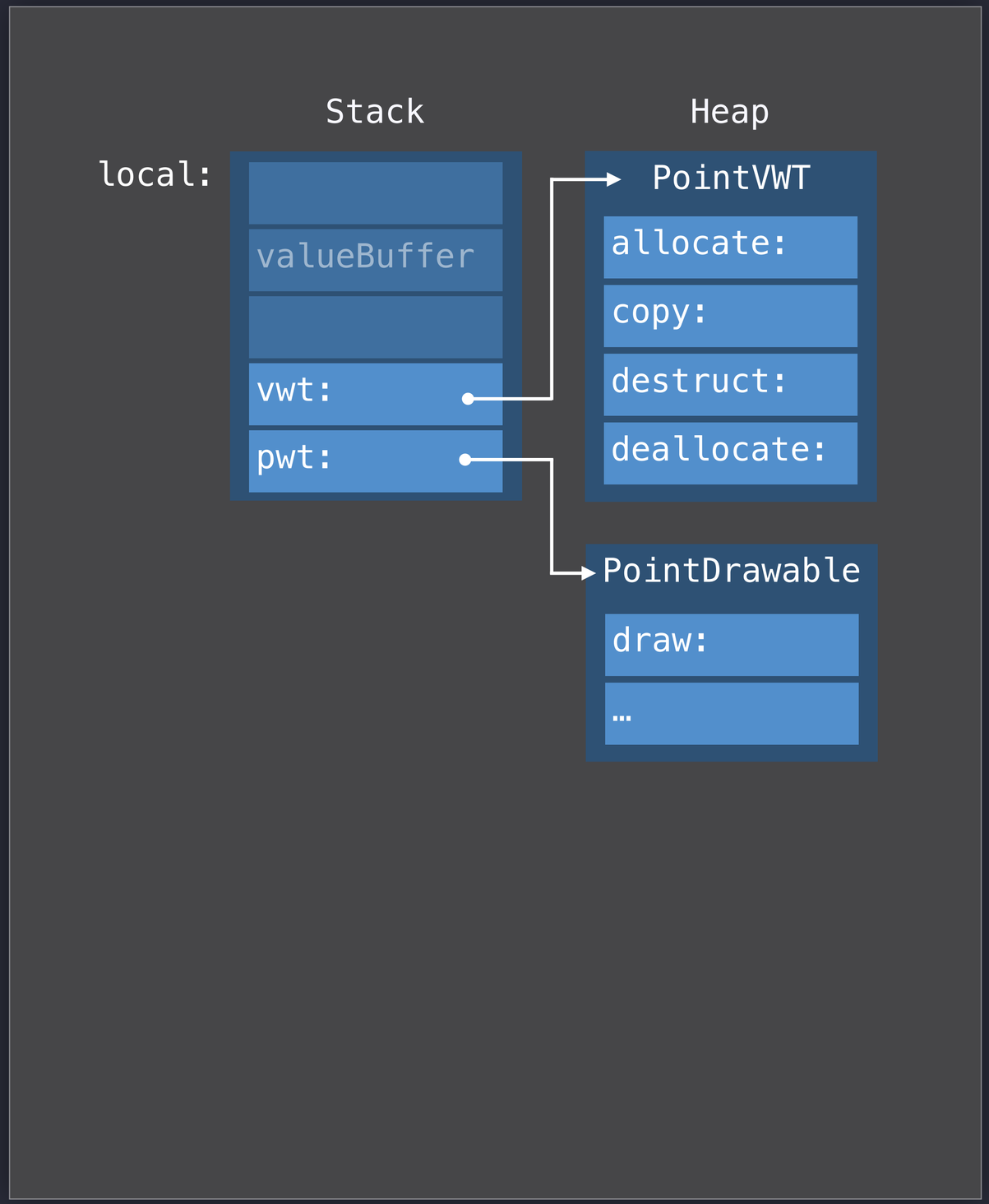
```
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
    local.draw()
}
let val : Drawable = Point()
drawACopy(val)
```

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



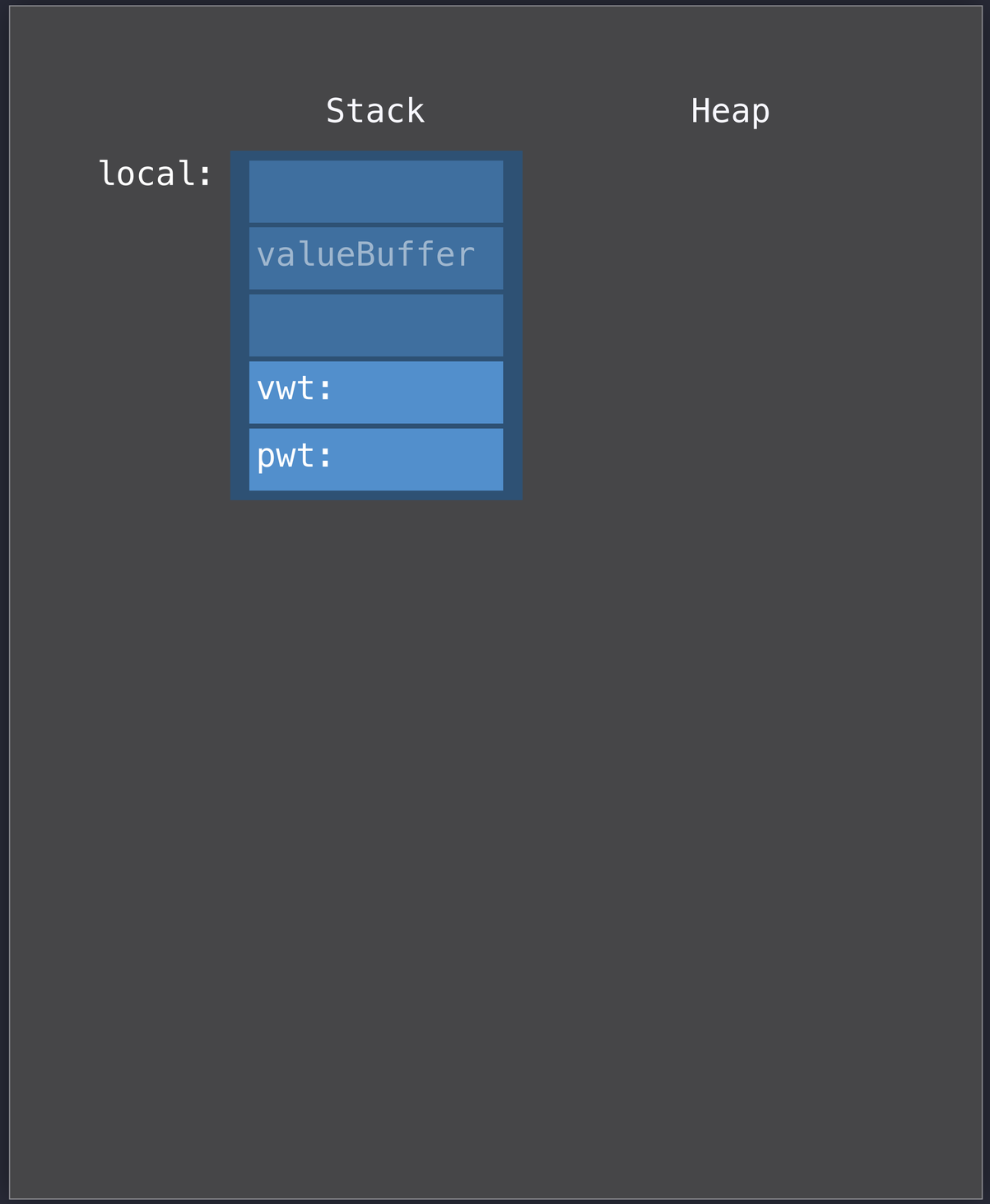
```
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
    local.draw()
}
let val : Drawable = Point()
drawACopy(val)
```

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



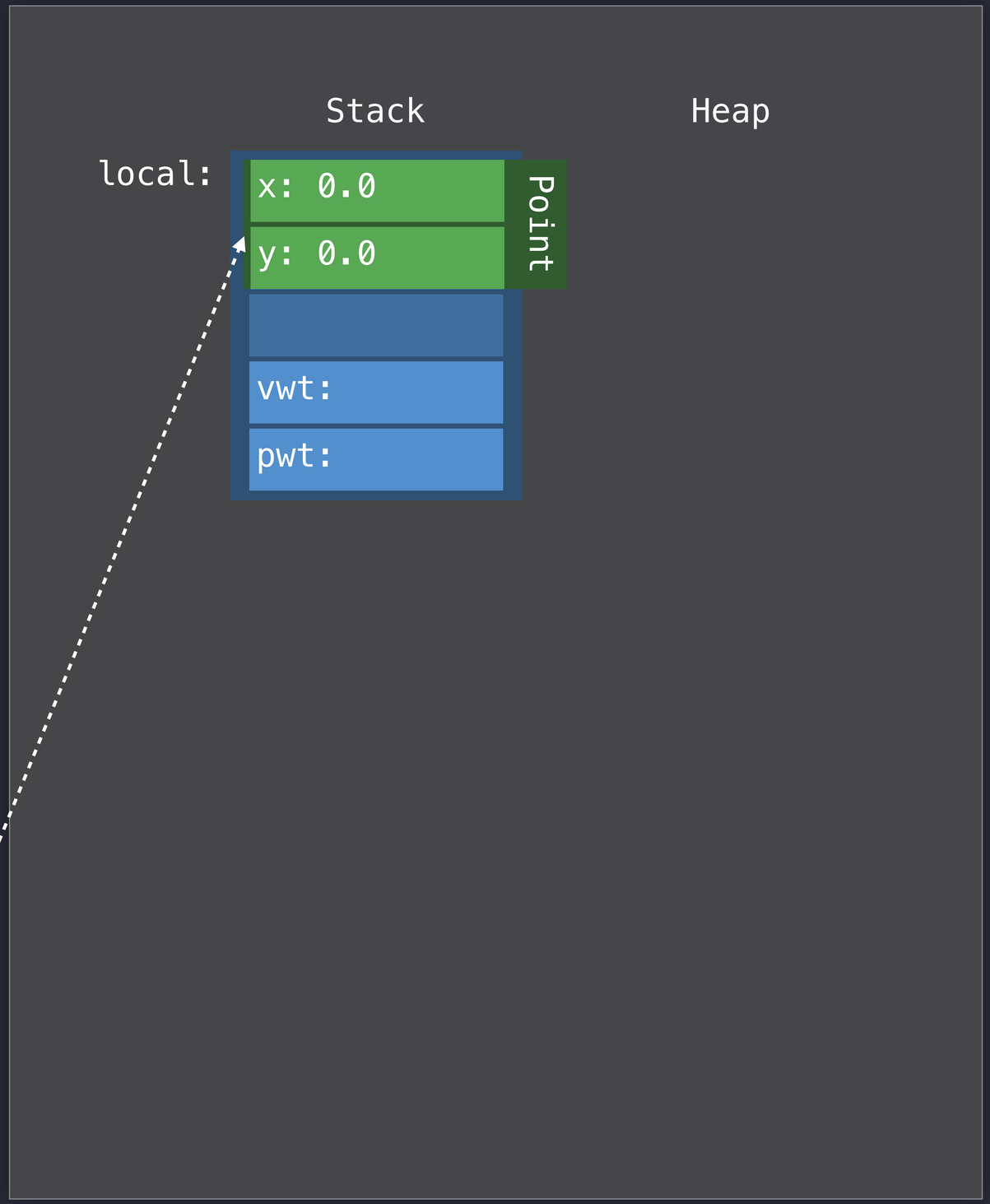
```swift
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
    local.draw()
}
let val : Drawable = Point()
drawACopy(val)
```

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



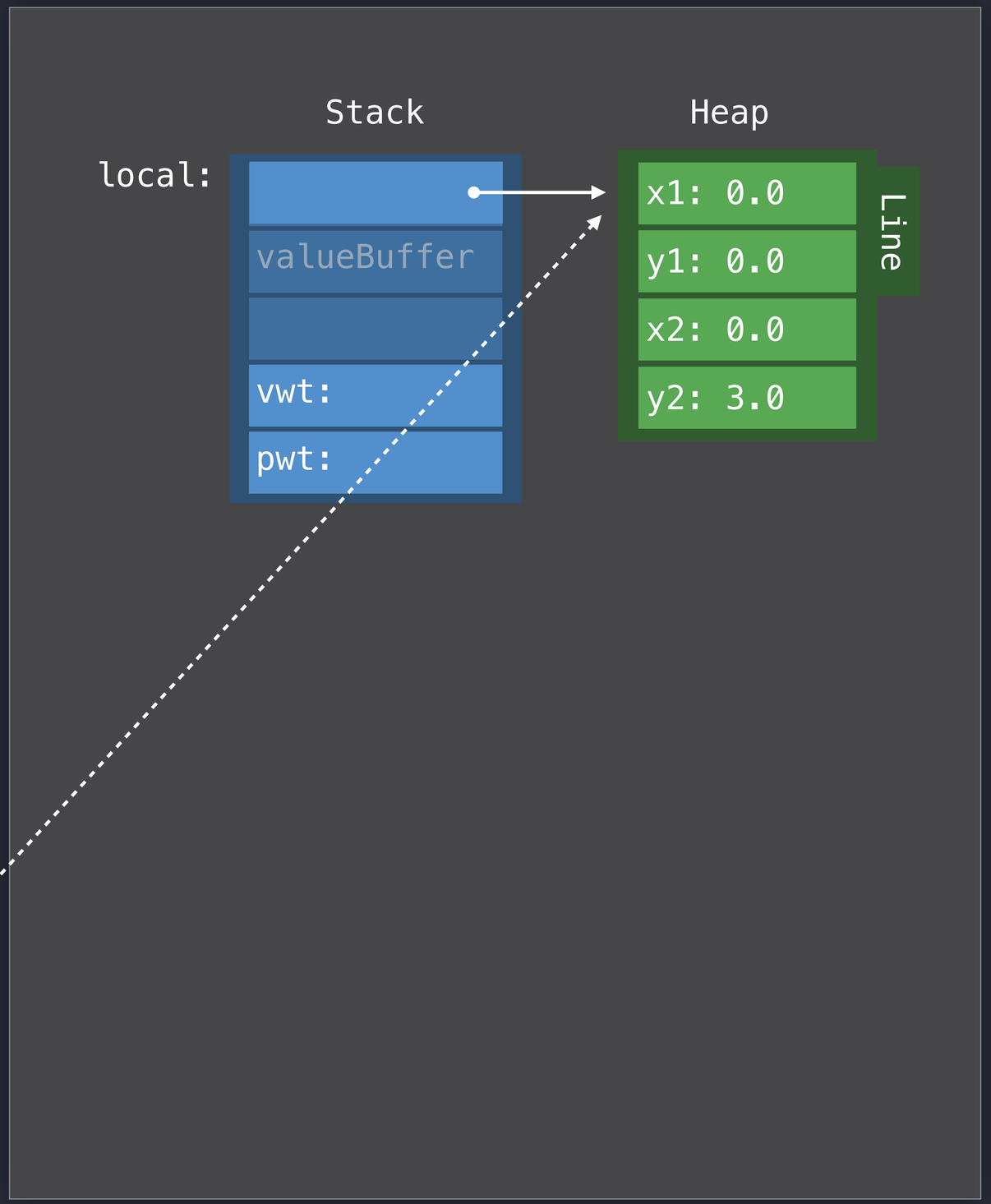
```swift
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
    local.draw()
}
let val : Drawable = Point()
drawACopy(val)
```

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



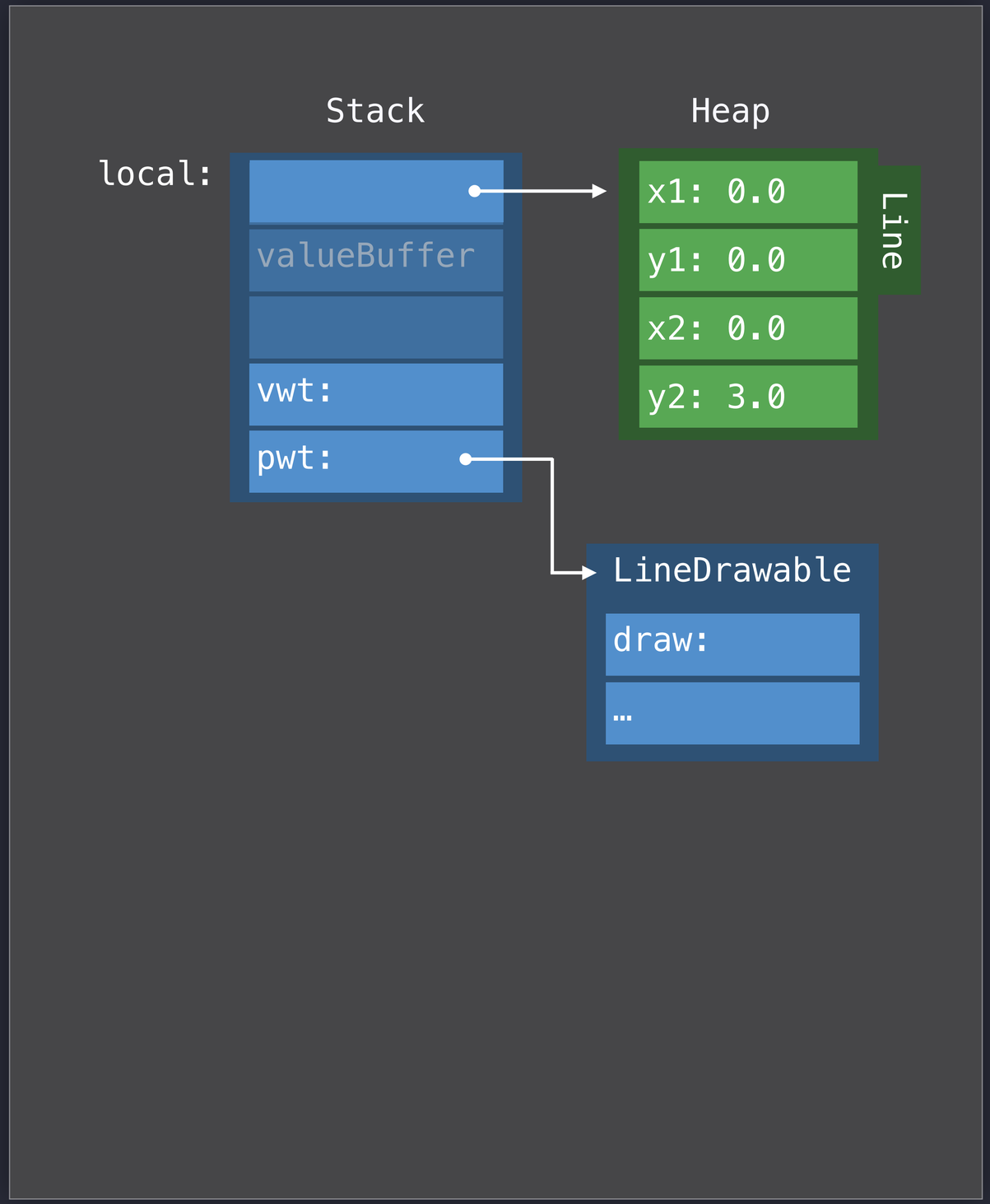
```swift
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
    local.draw()
}
let val : Drawable = Line()
drawACopy(val)
```

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



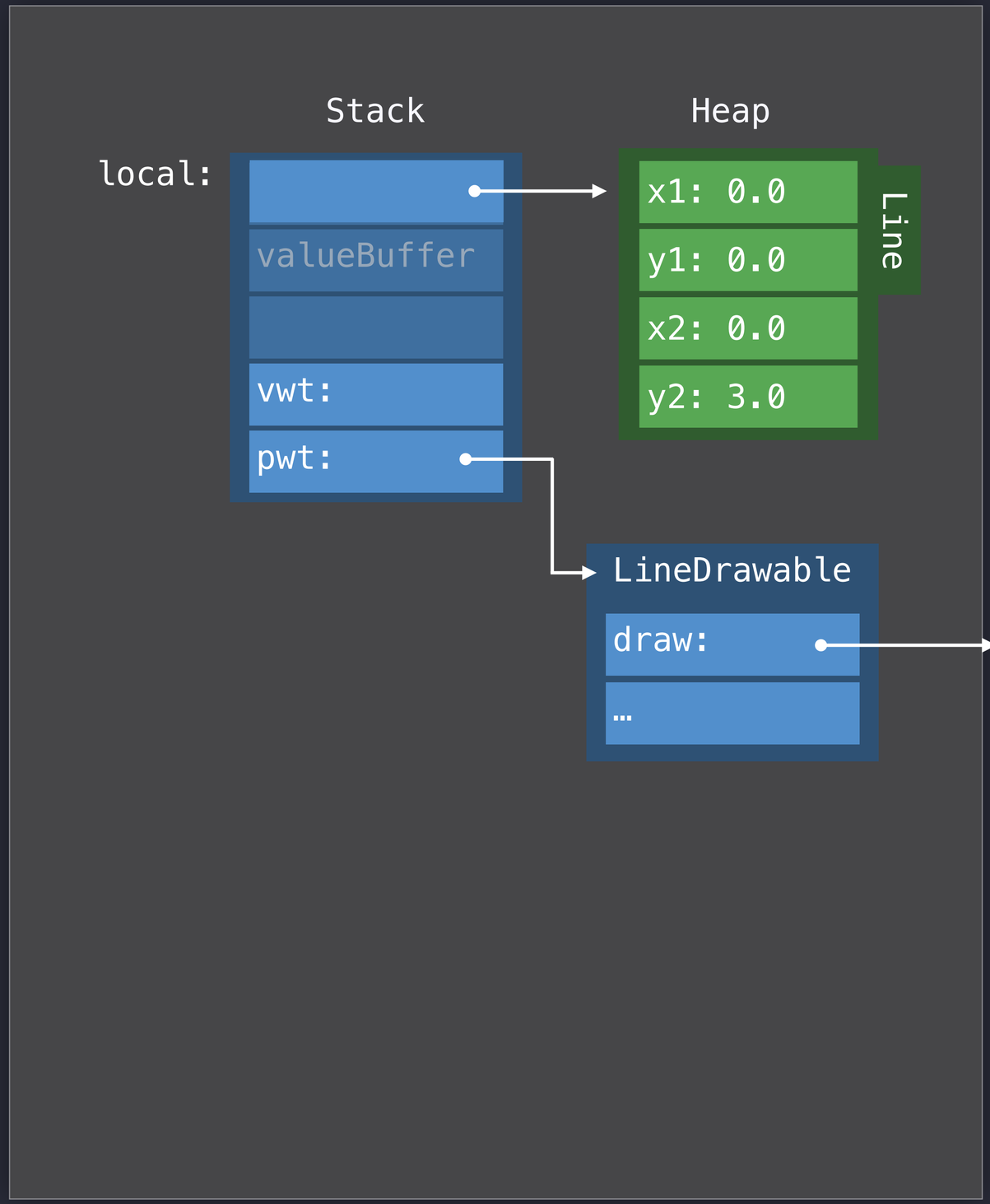
```
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
    local.draw()
}
let val : Drawable = Line()
drawACopy(val)
```

```
// 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))
```



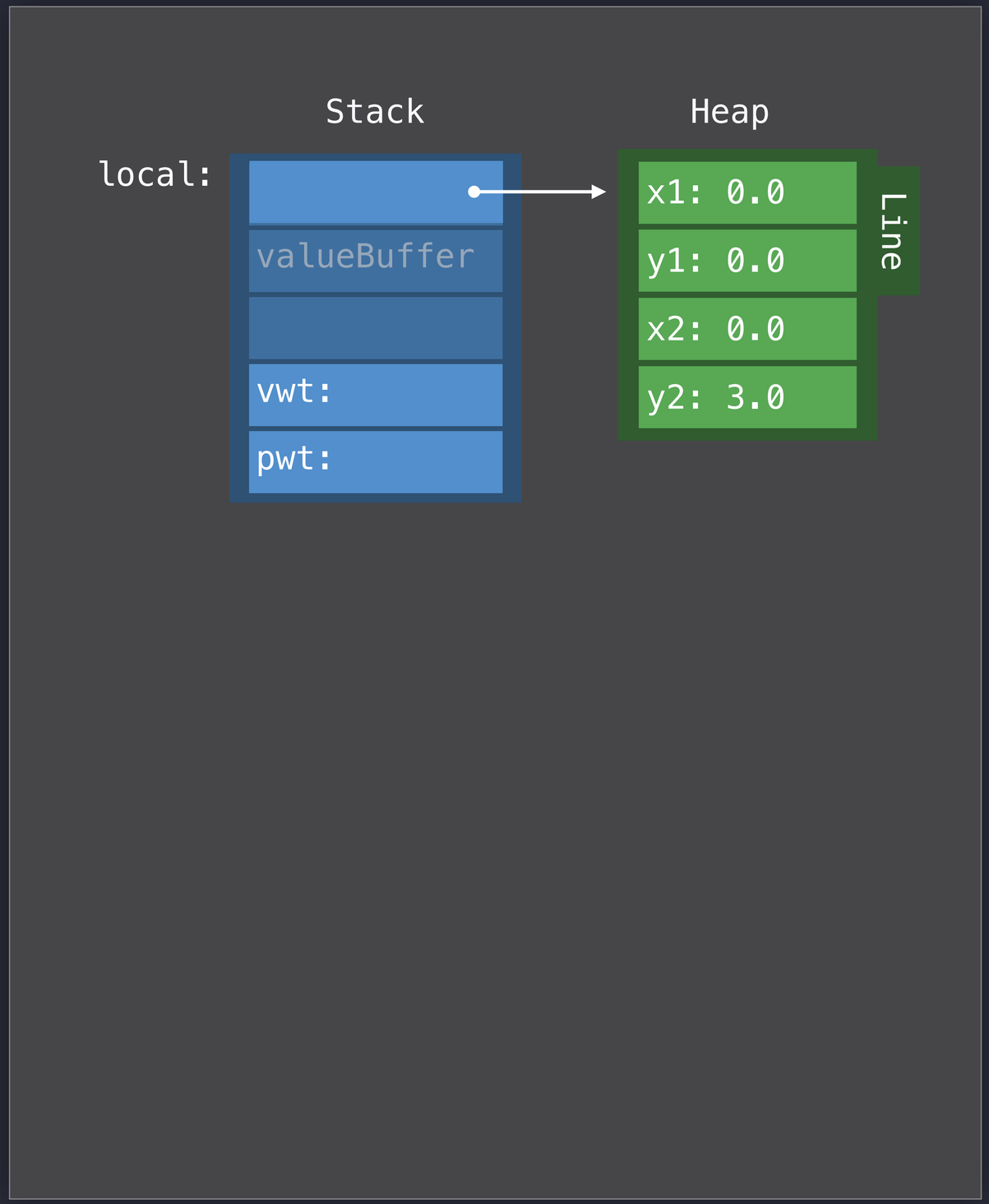
```
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
    local.draw()
}
let val : Drawable = Line()
drawACopy(val)
```

```
// 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))
```



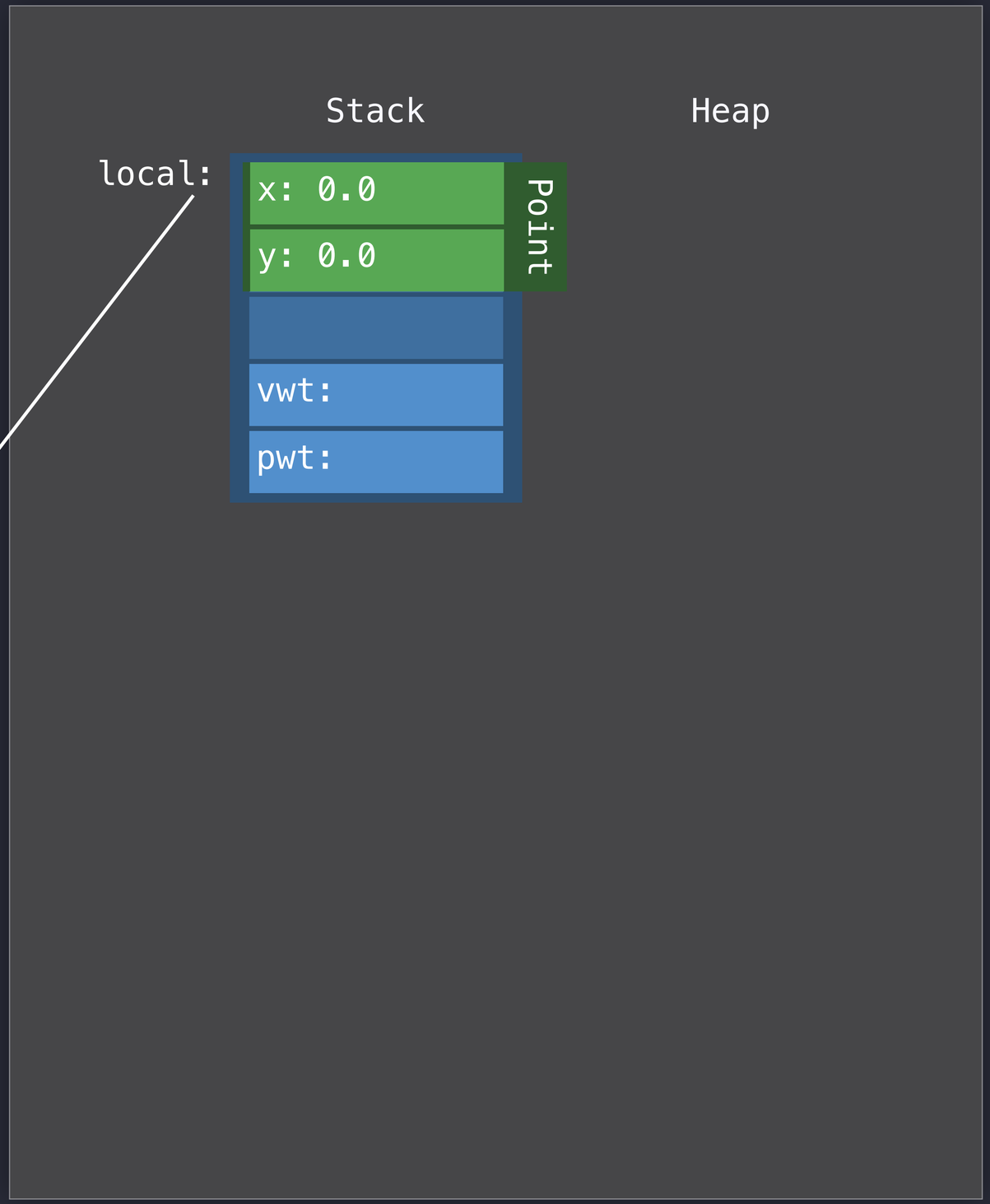
```
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
    local.draw()
}
let val : Drawable = Line()
drawACopy(val)
```

```
// 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))
```



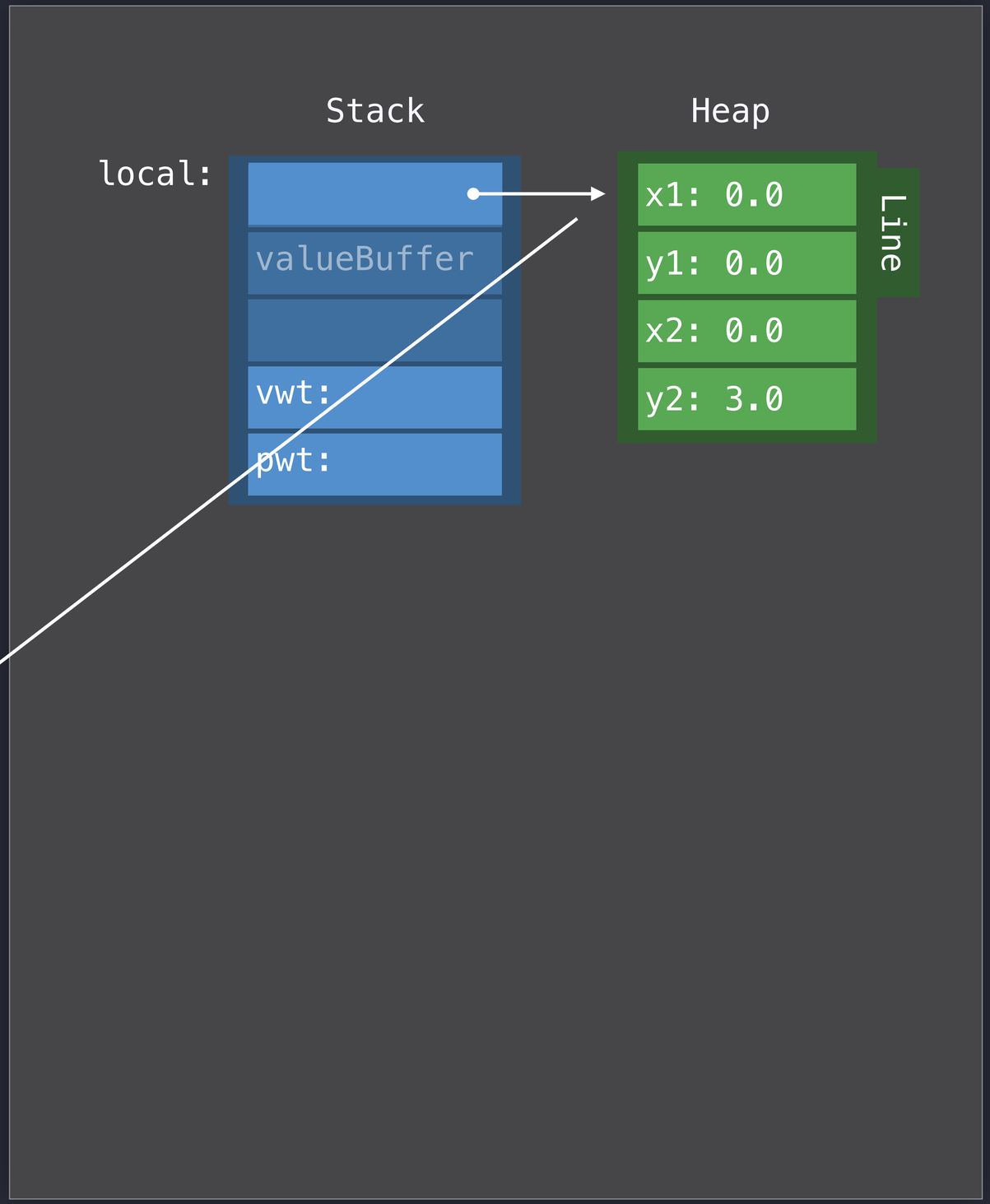
```
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
    local.draw()
}
let val : Drawable = Point()
drawACopy(val)
```

```
// 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))
```



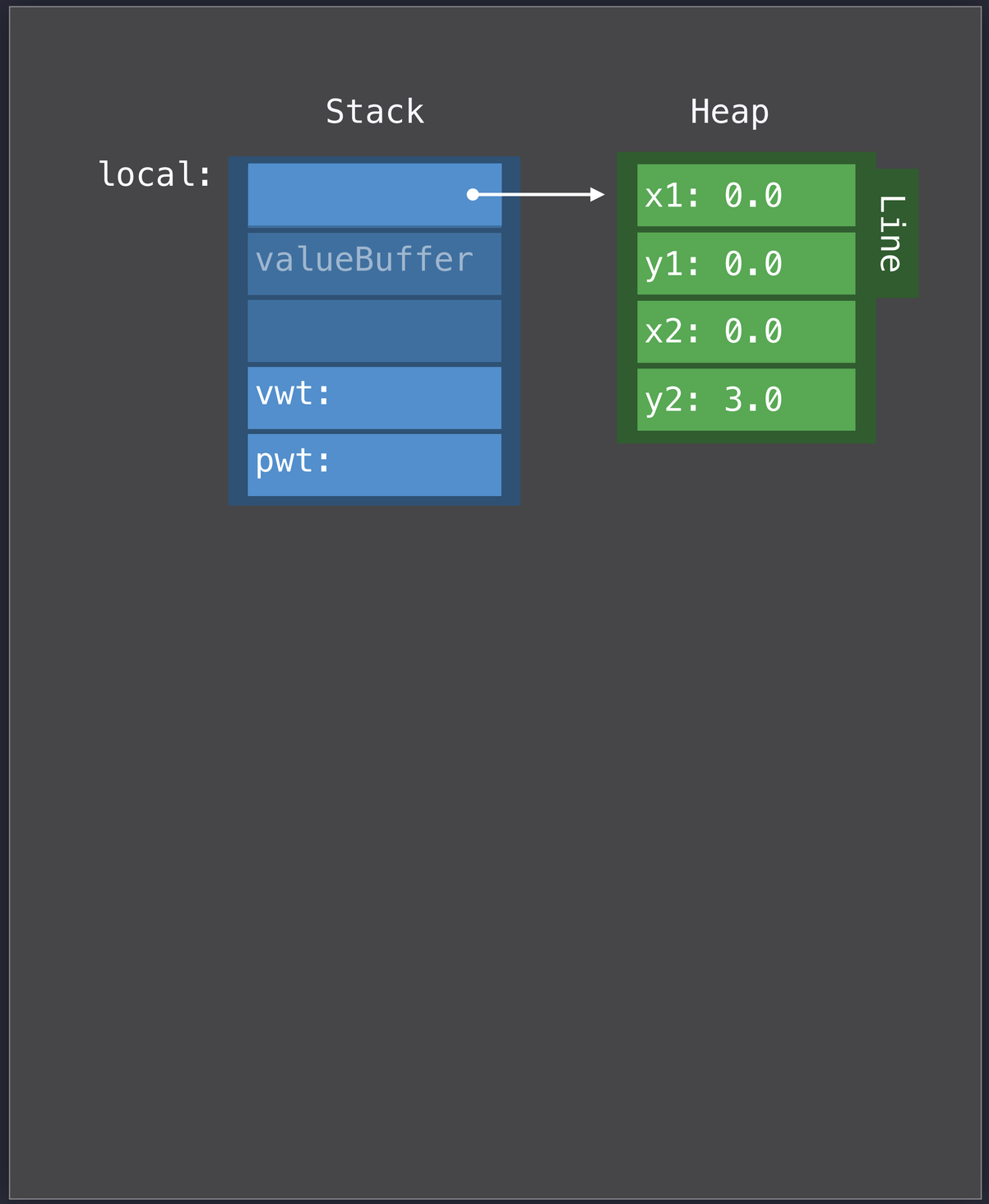
```swift
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
    local.draw()
}
let val : Drawable = Line()
drawACopy(val)
```

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



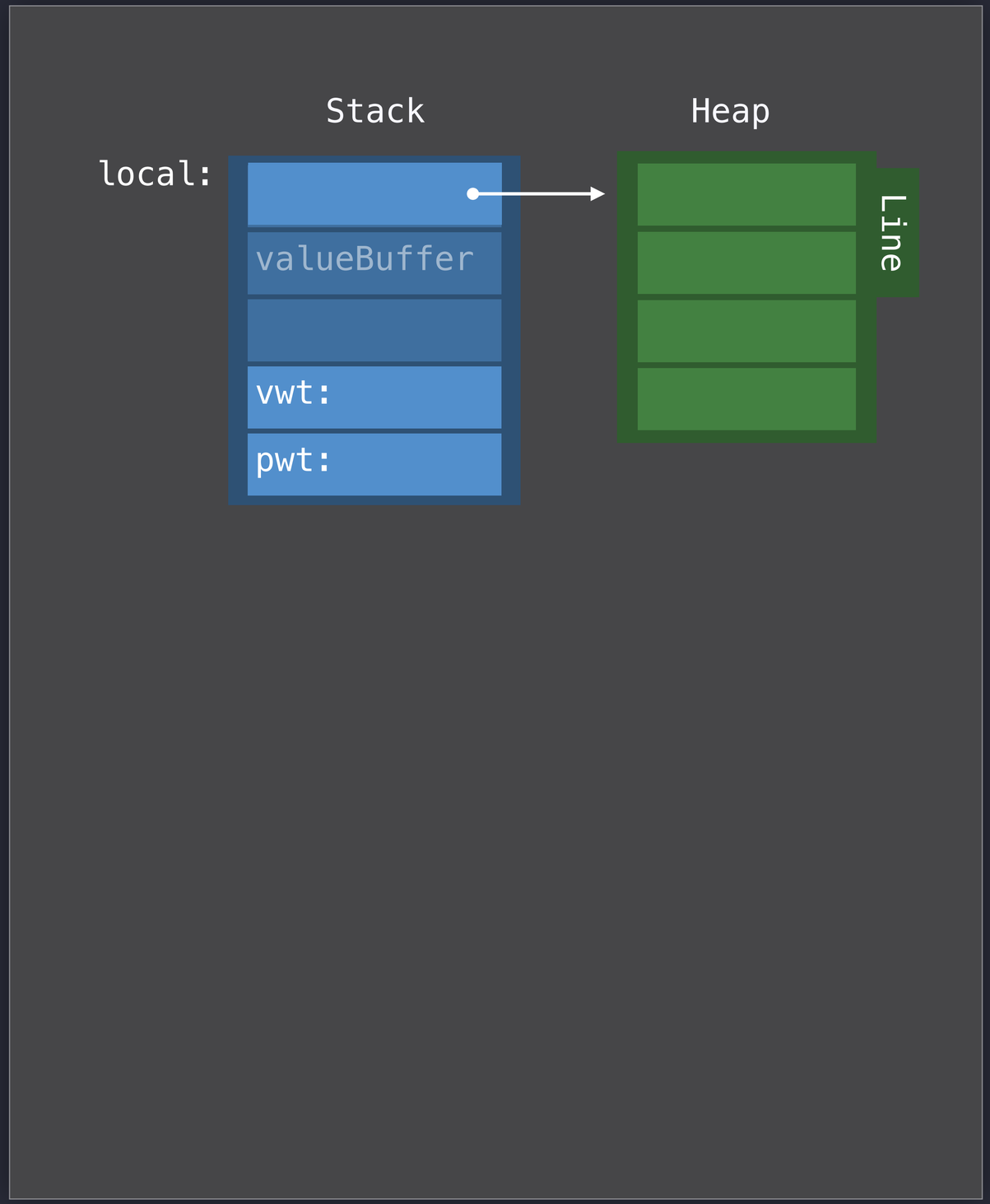
```
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
   local.draw()
}
let val : Drawable = Line()
drawACopy(val)
```

```
// 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)
}
```



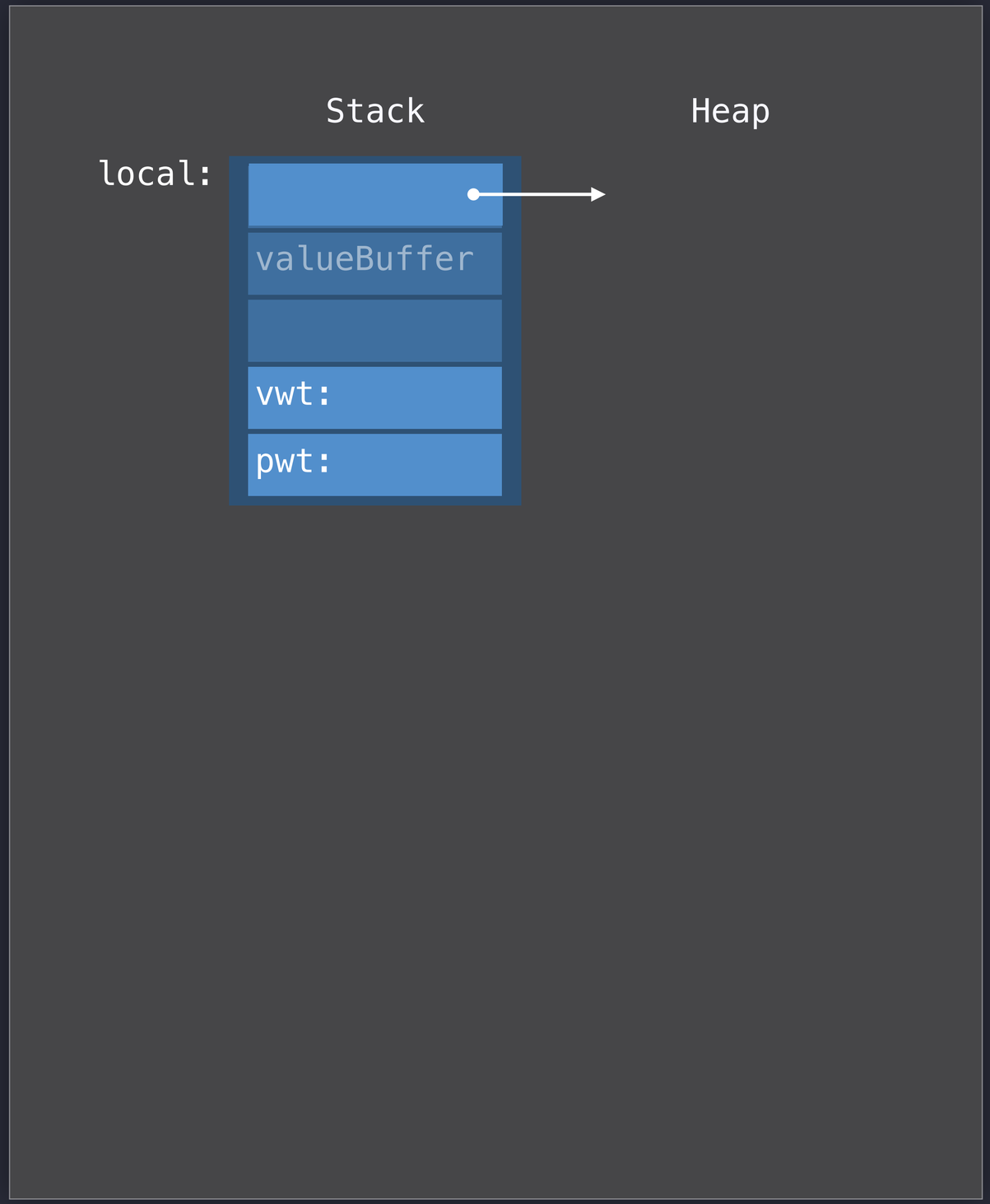
```
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
   local.draw()
}
let val : Drawable = Line()
drawACopy(val)
```

```
// 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)
}
```



```
// 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 Type Stored Properties

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

Protocol Type Stored Properties

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

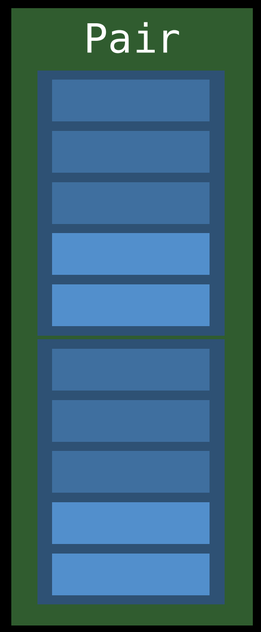
Existential Container inline

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

pair:

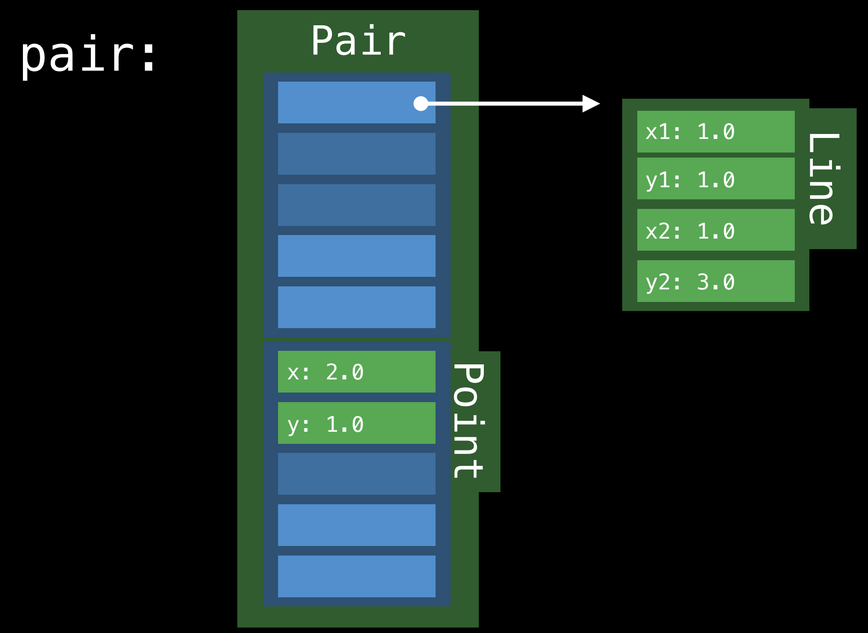


Protocol Type Stored Properties

```
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

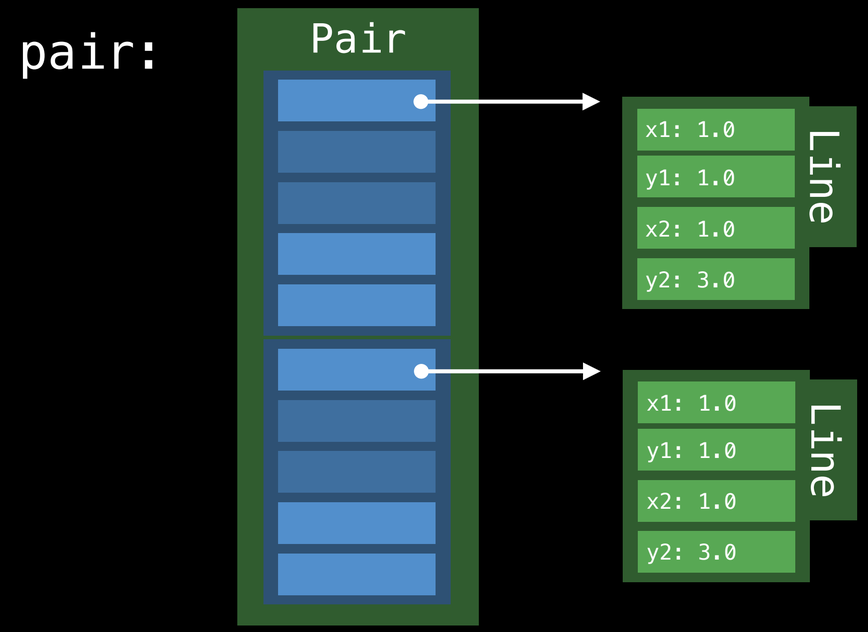
Large values on the heap



Protocol Type Stored Properties

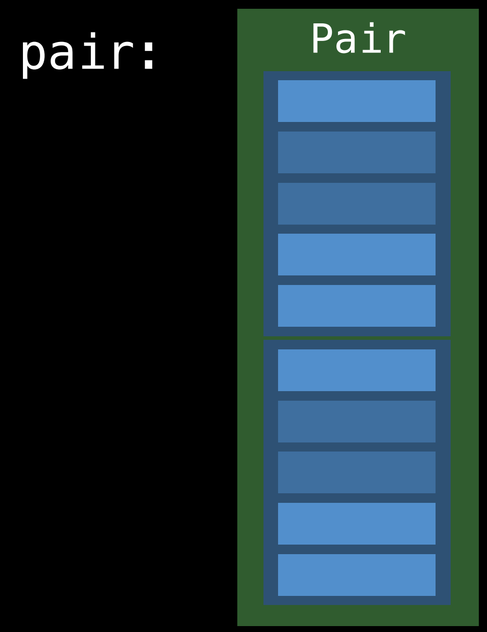
```
struct Pair {
   init(_ f: Drawable, _ s: Drawable) {
      first = f ; second = s
   }
   var first: Drawable
   var second: Drawable
}

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

Supports dynamic polymorphism



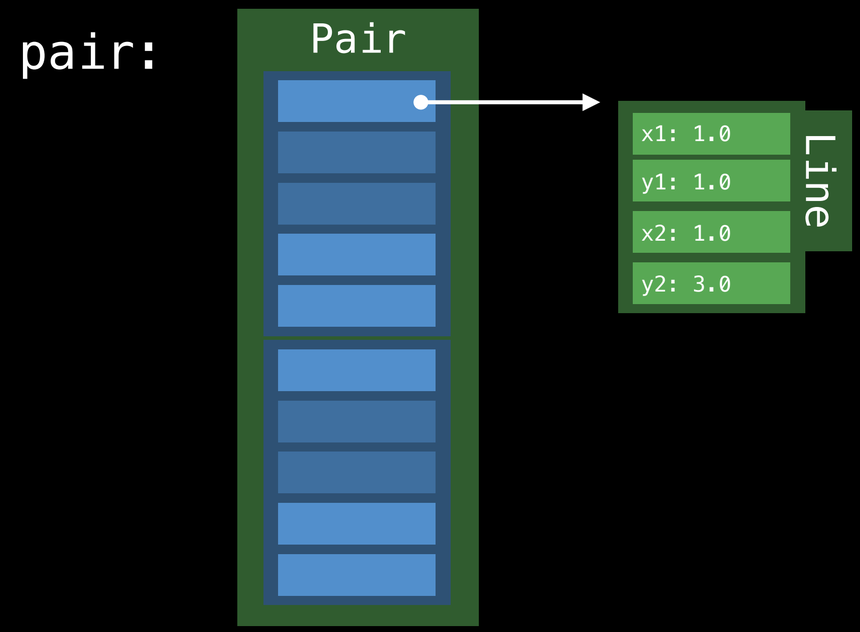
Expensive Copies of Large Values

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



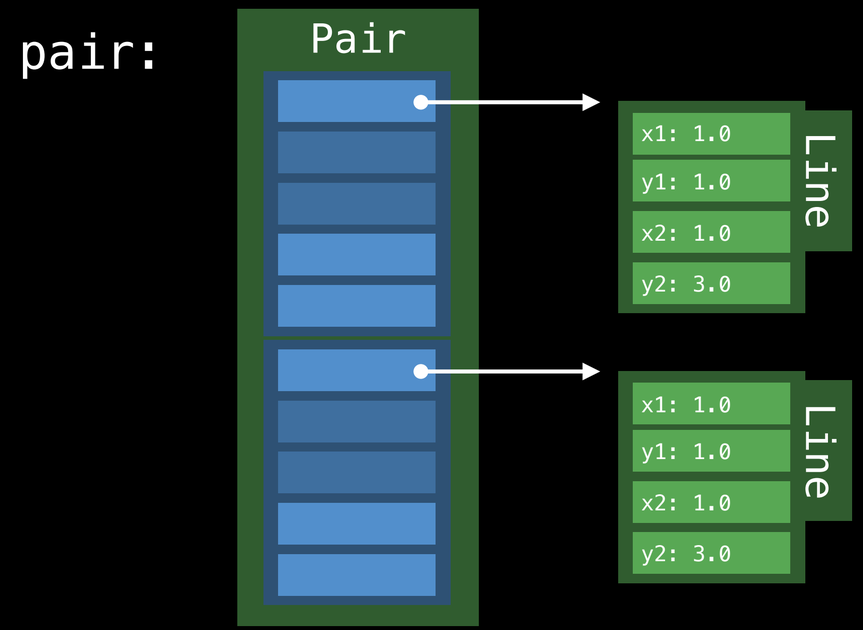
Expensive Copies of Large Values

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



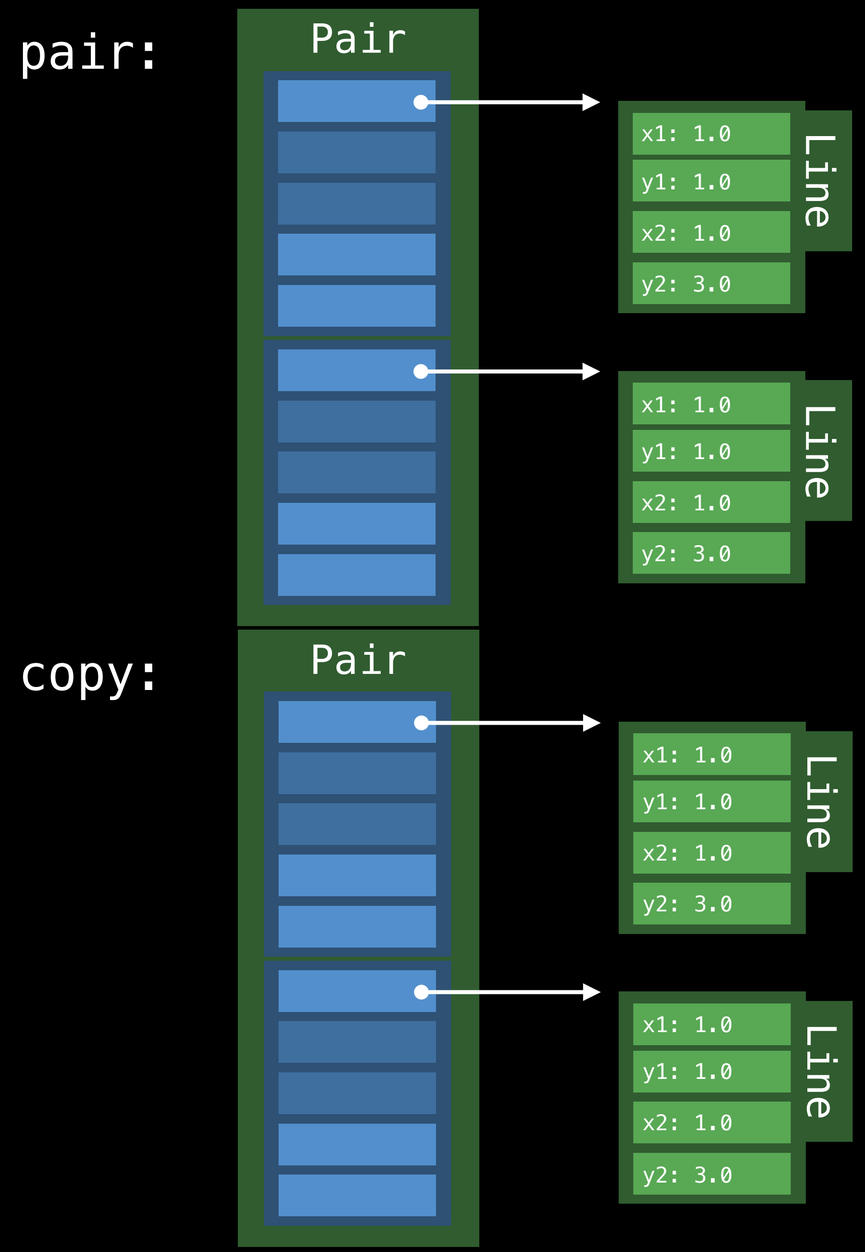
Expensive Copies of Large Values

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

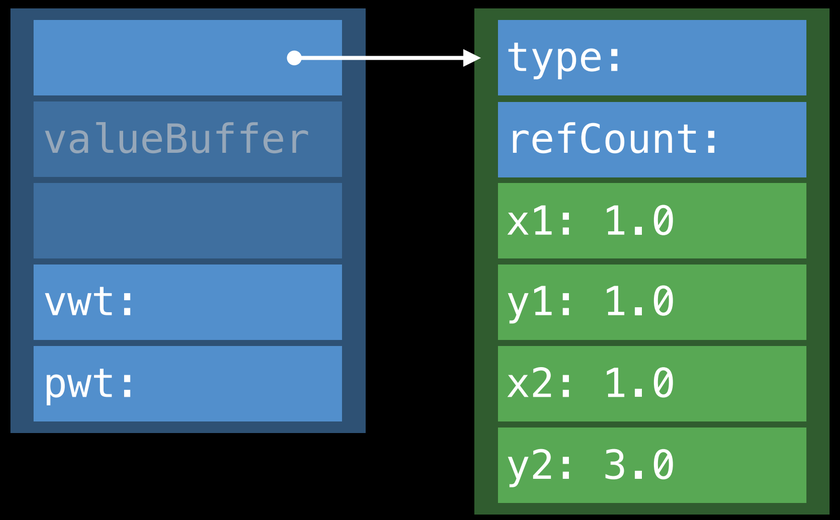


Expensive Copies of Large Values

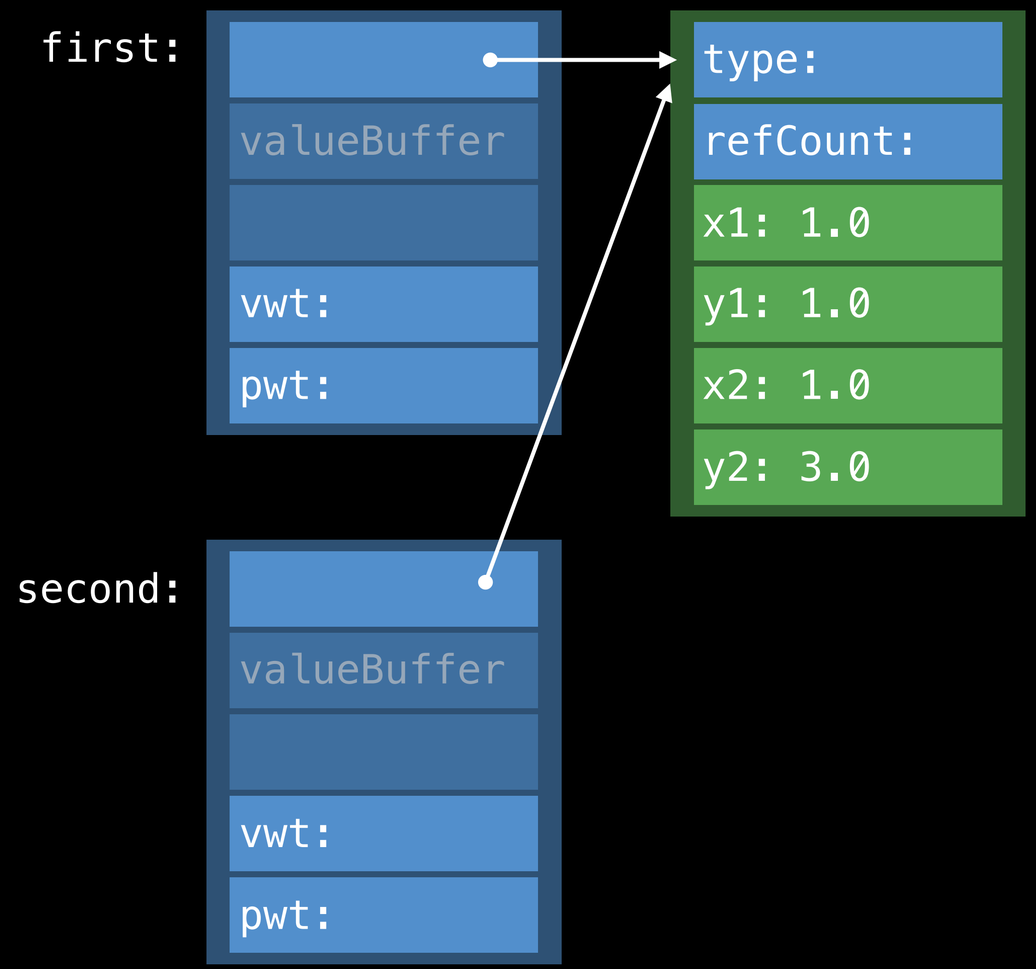
```
let aLine = Line(1.0, 1.0, 1.0, 3.0)
let pair = Pair(aLine, aLine)
let copy = pair
```



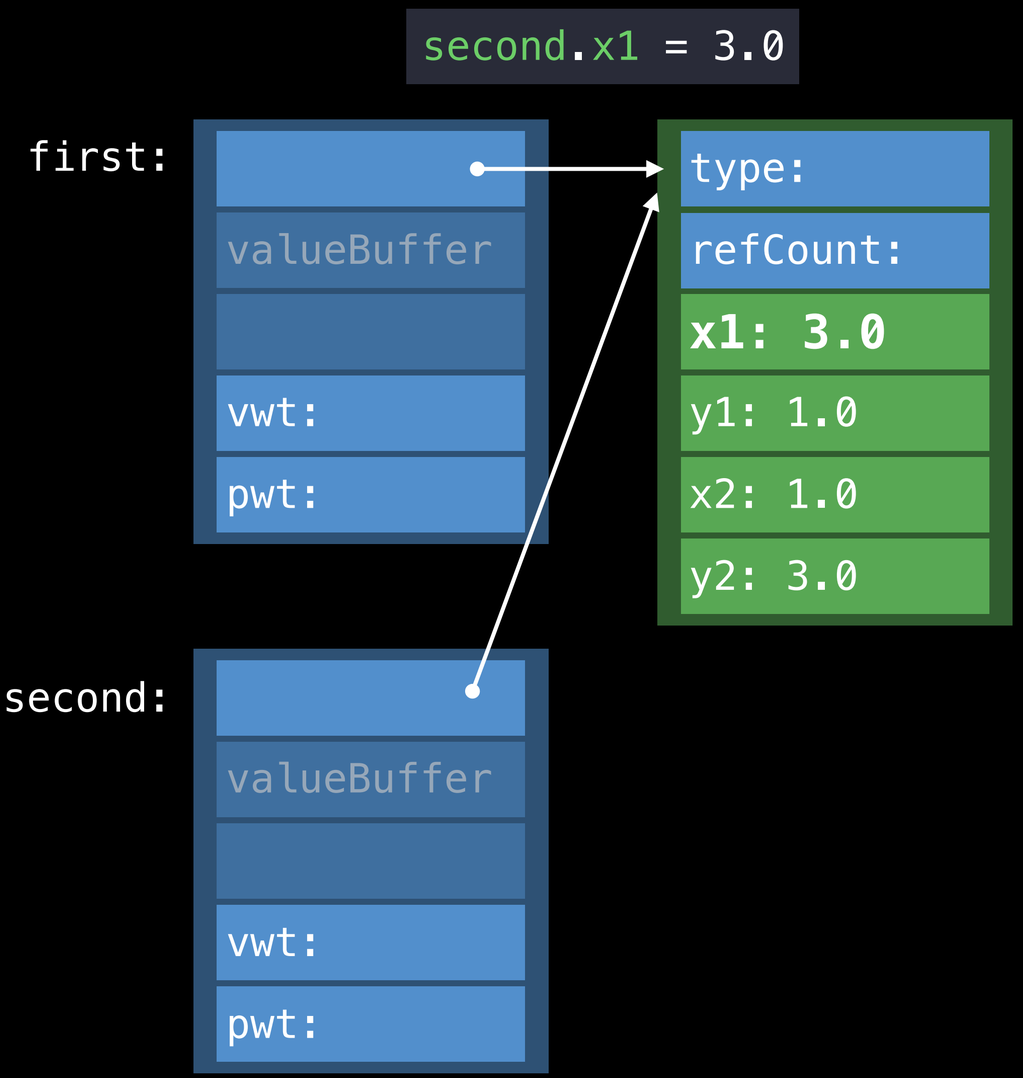
References Fit in the Value Buffer



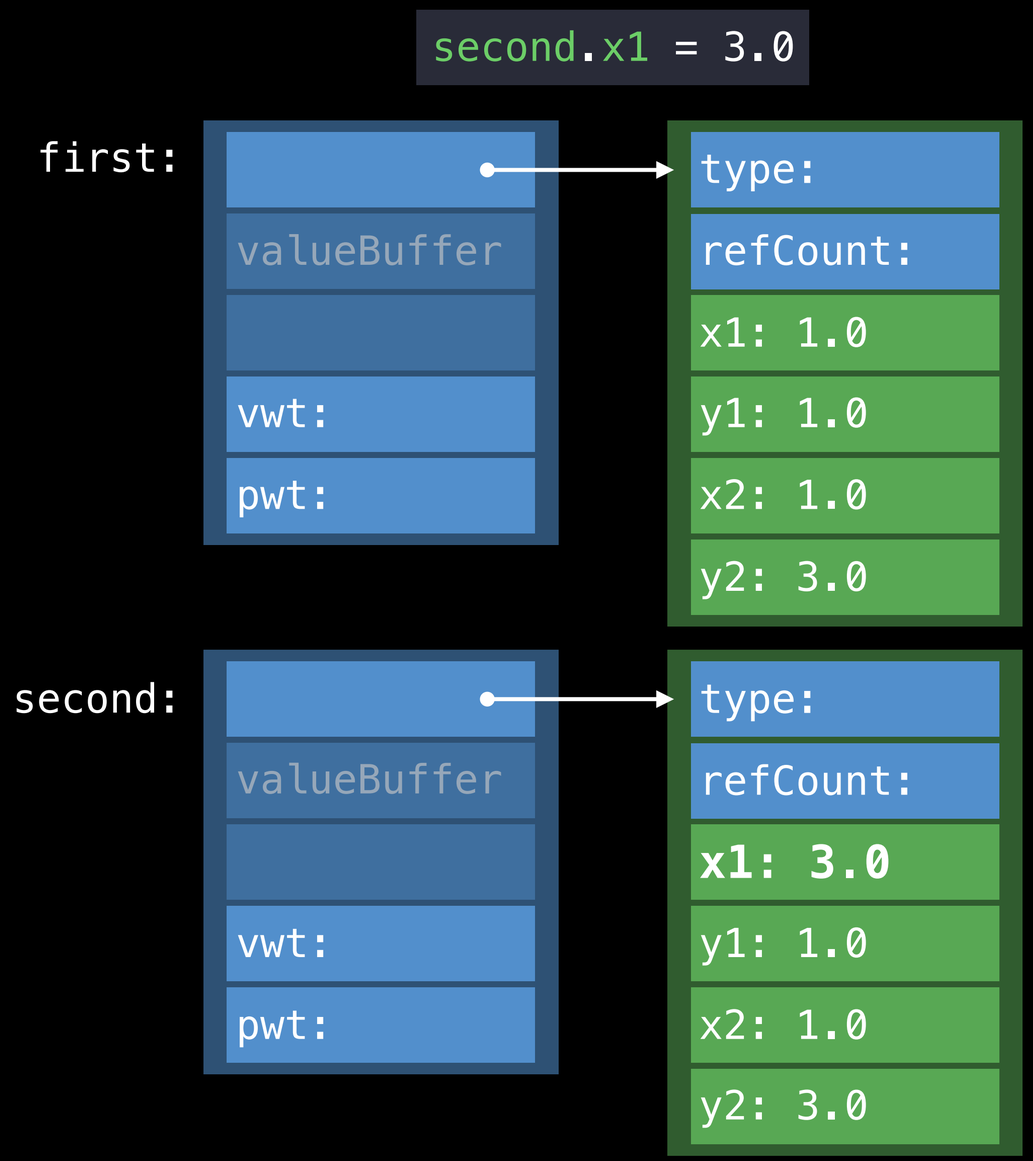
References Fit in the Value Buffer



References Fit in the Value Buffer



References Fit in the Value Buffer



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 = ...
   }
}
```

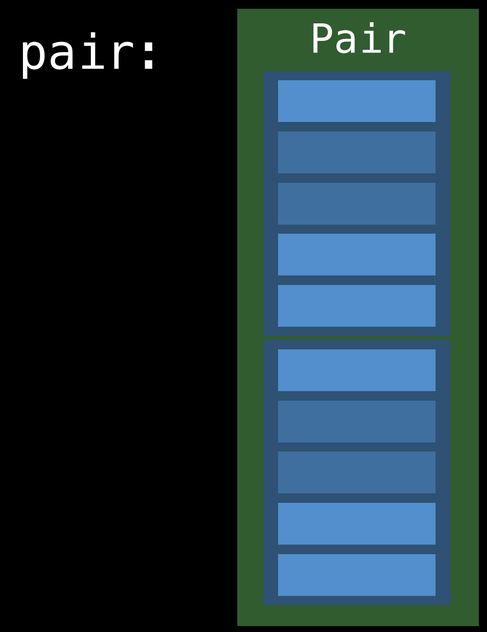
Indirect Storage with Copy-on-Write

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 = ...
   }
}
```

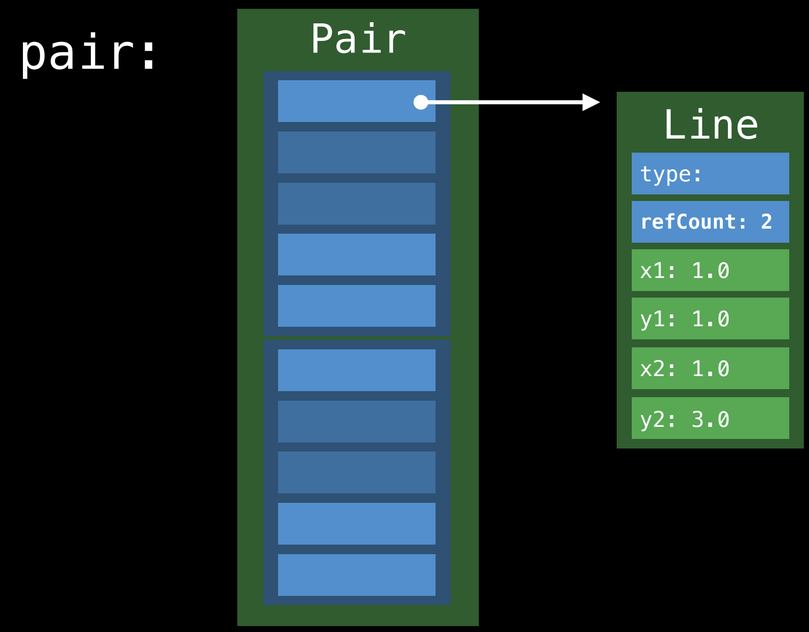
Copy Using Indirect Storage

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



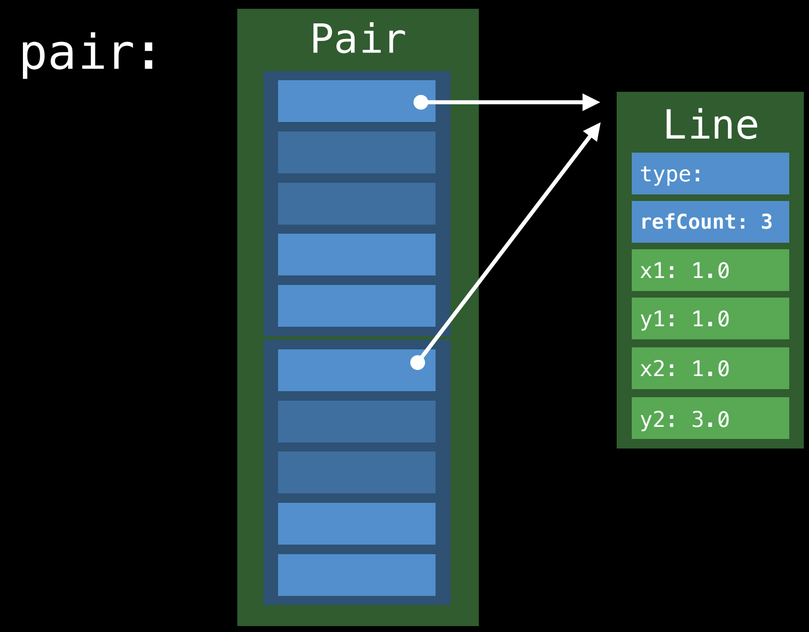
Copy Using Indirect Storage

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



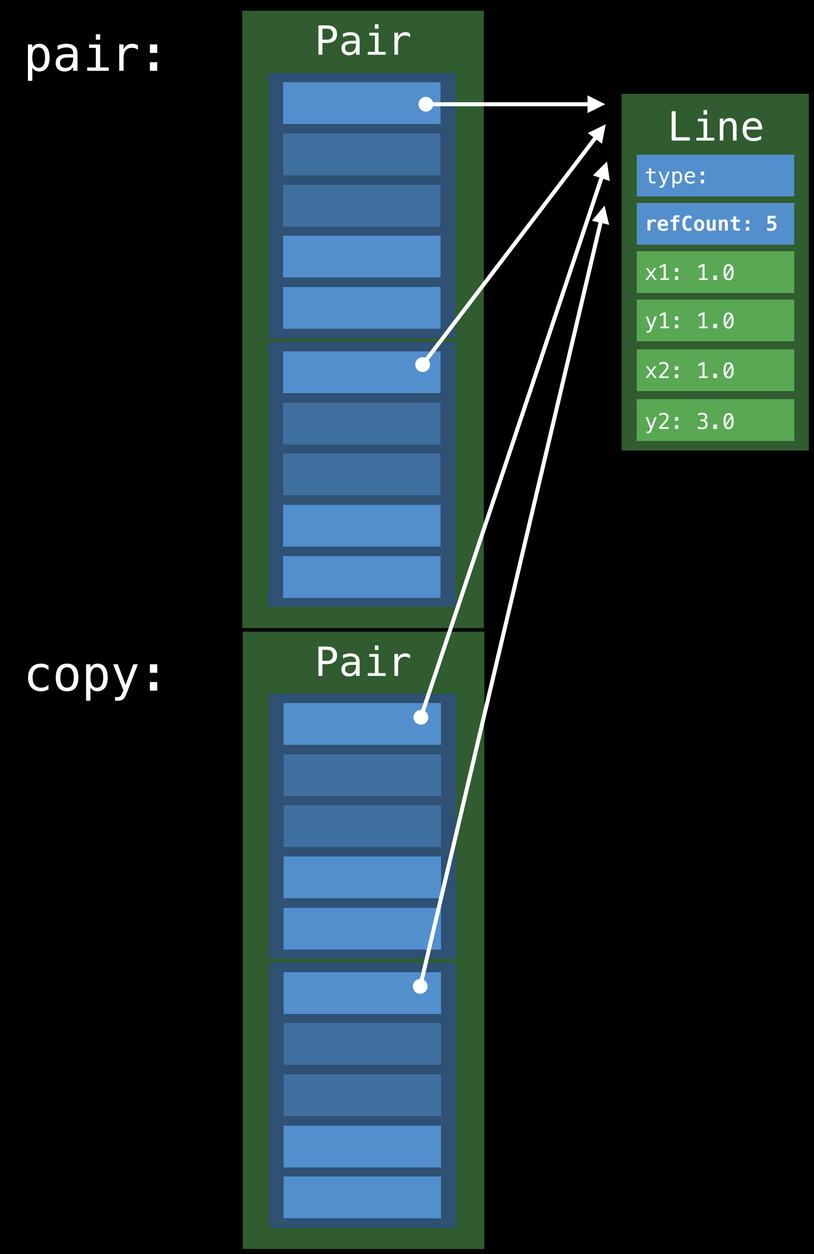
Copy Using Indirect Storage

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



Copy Using Indirect Storage

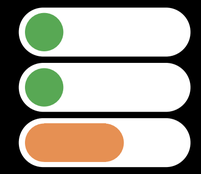
```
let aLine = Line(1.0, 1.0, 1.0, 1.0)
let pair = Pair(aLine, aLine)
let copy = pair
```



Performance of Protocol Types

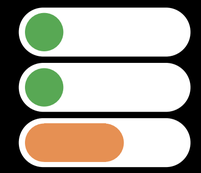
```
func drawACopy(val: ExistContDrawable) {
   var local = ExistContDrawable()
   let vwt = val.vwt
   let pwt = val.pwt
   local.type = type
   local.pwt = pwt
   vwt.allocateBufferAndCopyValue(&local, val)
   pwt.draw(vwt.projectBuffer(&local))
   vwt.destructAndDeallocateBuffer(temp)
}
```

Protocol Type—Small Value



Fits in Value Buffer: no heap allocation

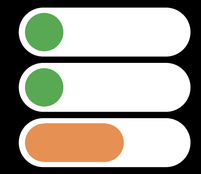
Protocol Type—Small Value



Fits in Value Buffer: no heap allocation

No reference counting

Protocol Type—Small Value

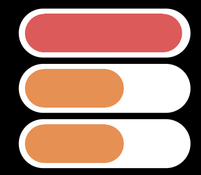


Fits in Value Buffer: no heap allocation

No reference counting

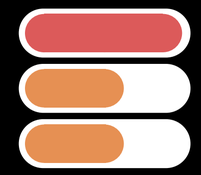
Dynamic dispatch through Protocol Witness Table

Protocol Type—Large Value



Heap allocation

Protocol Type—Large Value

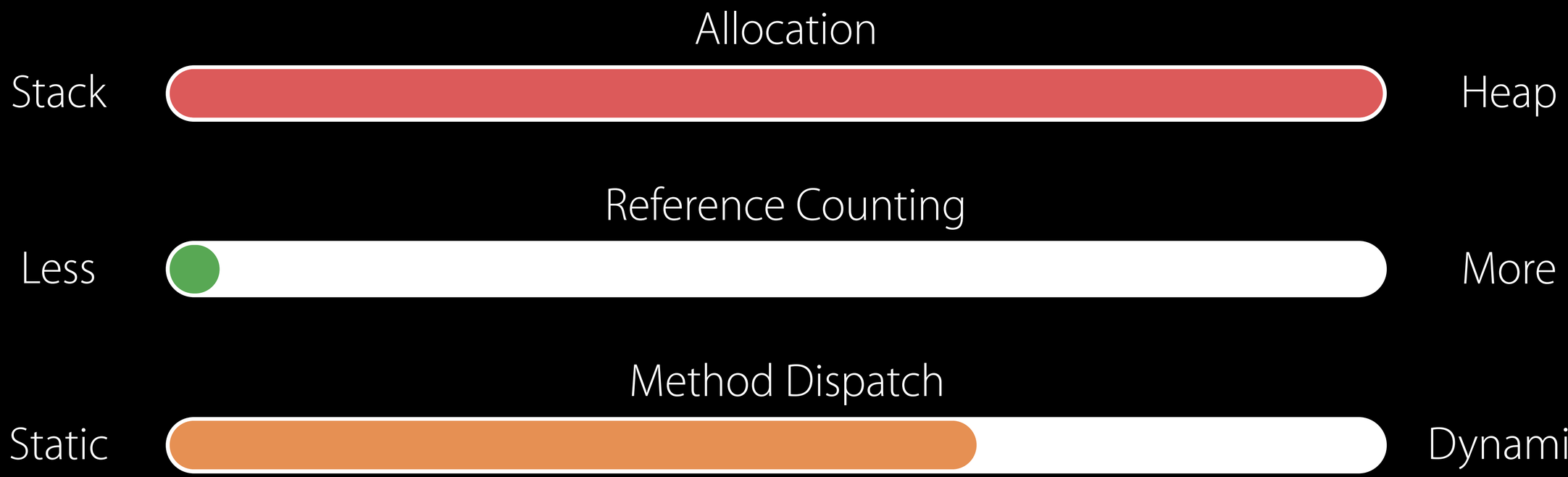


Heap allocation

Reference counting if value contains references

Protocol Type—Large Value

Expensive heap allocation on copying

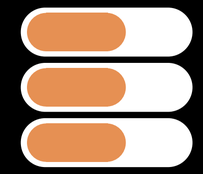


Protocol Type—Indirect Storage

Trade off reference counting for heap allocation

Protocol Type—Indirect Storage

Trade off reference counting for heap allocation



Allocation

Stack — Heap

Reference Counting

Less — More

Method Dispatch

Static — Dynamic

Summary—Protocol Types

Dynamic polymorphism

Indirection through Witness Tables and Existential Container

Copying of large values causes heap allocation

```
// Drawing a copy
protocol Drawable {
   func draw()
}
func drawACopy(local : Drawable) {
  local.draw()
}
```

```swift
// Drawing a copy
protocol Drawable {
   func draw()
}
func drawACopy(local : Drawable) {
  local.draw()
}

let line = Line()
drawACopy(line)
```

```swift
// Drawing a copy
protocol Drawable {
   func draw()
}
func drawACopy(local : Drawable) {
  local.draw()
}

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

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)
```

```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)
```

Generic Code

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

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

Static polymorphism

One type per call context

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)
```

Static polymorphism

One type per call context

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

Implementation of Generic Methods

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

drawACopy(Point(…))
```

Implementation of Generic Methods

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

drawACopy(Point(…))
```

One shared implementation

Implementation of Generic Methods

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

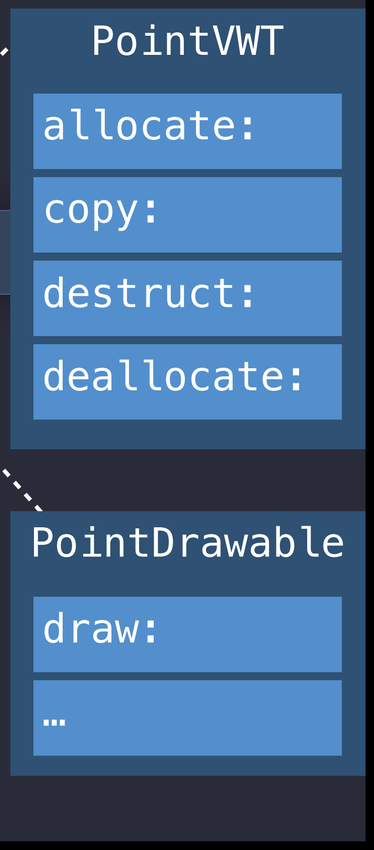
One shared implementation

Uses Protocol/Value Witness Table

One type per call context

Implementation of Generic Methods

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

drawACopy(Point(…))
```



One shared implementation

Uses Protocol/Value Witness Table

One type per call context: passes tables

Implementation of Generic Methods

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

drawACopy(Point(…))
```

One shared implementation

Uses Protocol/Value Witness Table

One type per call context: passes tables

Implementation of Generic Methods

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

drawACopy(Point(…))
```

```
let local = Point()
```

PointVWT

allocate:

copy:

destruct

deallocate

One shared implementation

Uses Protocol/Value Witness Table

One type per call context: passes tables

Implementation of Generic Methods

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


drawACopy(Point(…))
```

PointDrawable

draw:

…

One shared implementation

Uses Protocol/Value Witness Table

One type per call context: passes tables

Implementation of Generic Methods

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


drawACopy(Point(…))
```

One shared implementation

Uses Protocol/Value Witness Table

One type per call context: passes tables

Storage of Local Variables

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


drawACopy(Point(…))
```

Storage of Local Variables

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

drawACopy(Point(…))
```

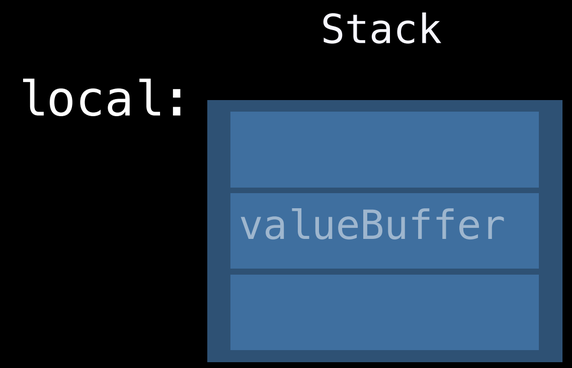
```
let local = Point()
```

Value Buffer: currently 3 words

Storage of Local Variables

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

drawACopy(Point(…))
```

Value Buffer: currently 3 words

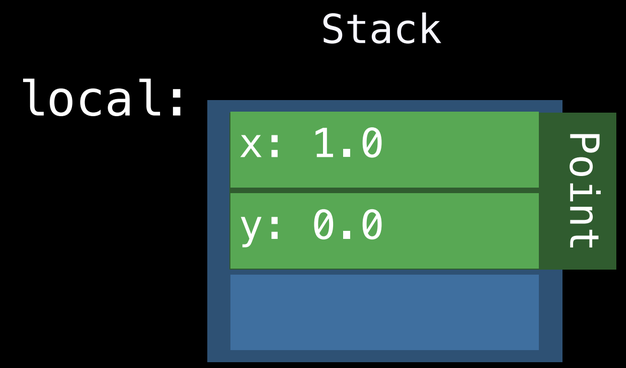


Storage of Local Variables

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

drawACopy(Point(…))
```

Value Buffer: currently 3 words

Small values stored inline



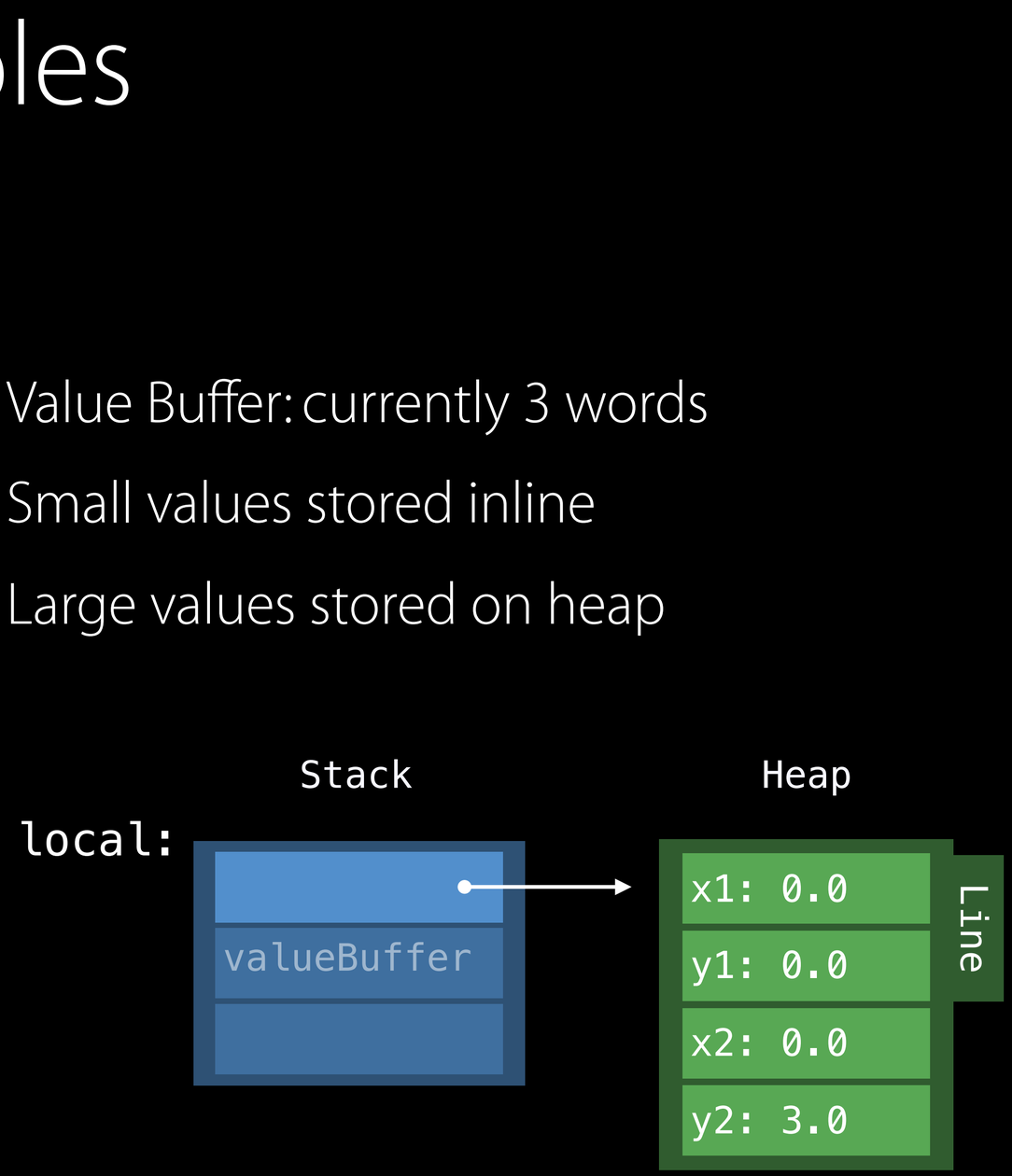
Storage of Local Variables

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

drawACopy(Line(…))
```

Value Buffer: currently 3 words

Small values stored inline

Large values stored on heap



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

Specialization of Generics

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

drawACopy(Point(…))
```

Static polymorphism: uses type at call-site

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()
}

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

When Does Specialization Happen?

main.swift

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

When Does Specialization Happen?

Infer type at call-site

main.swift

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

When Does Specialization Happen?

Infer type at call-site

Definition must be available

main.swift

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

Whole Module Optimization

Increases optimization opportunity

Point.swift

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

UsePoint.swift

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

Whole Module Optimization

Increases optimization opportunity

Module A

Point.swift

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

UsePoint.swift

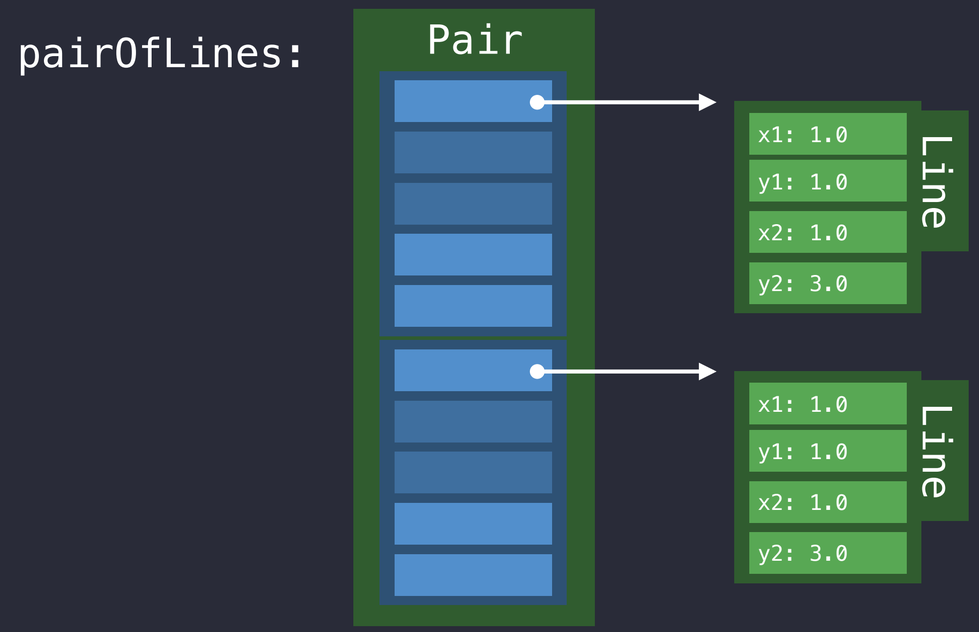
```
let point = Point()
drawACopy(point)
```

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

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

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

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

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

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

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



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

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

Generic Stored Properties

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

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

Generic Stored Properties

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

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

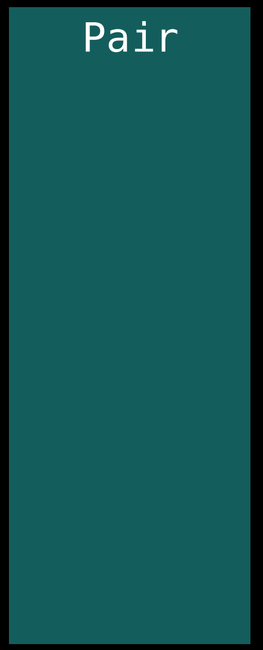
Type does not change at runtime

Generic Stored Properties

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

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

Type does not change at runtime

Storage inline

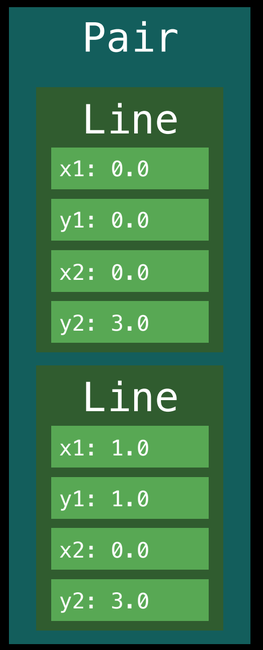


Generic Stored Properties

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

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

Type does not change at runtime

Storage inline

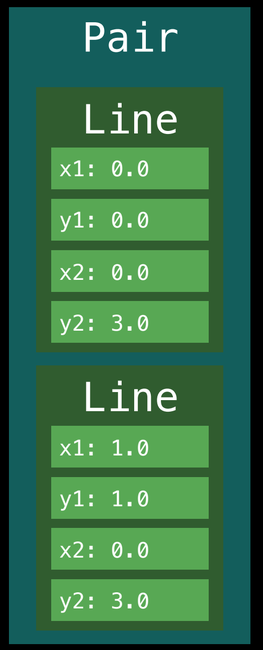


Generic Stored Properties

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

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

Type does not change at runtime

Storage inline



Performance of Generic Code

Unspecialized

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


drawACopy(Point(…))
```

Performance of Generic Code

Unspecialized

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

drawACopy(Point(…))
```

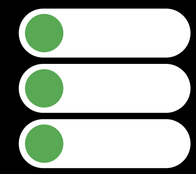
PointVWT

- **allocate:**
- **copy:**
- destruct:
- deallocate:

Specialized

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

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

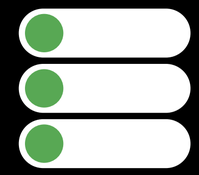
Specialized Generics—Struct Type



Performance characteristics like struct types

- No heap allocation on copying

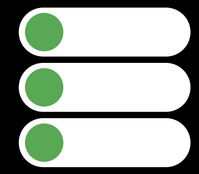
Specialized Generics—Struct Type



Performance characteristics like struct types

- No heap allocation on copying
- No reference counting

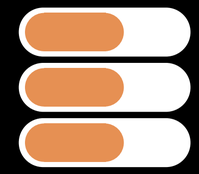
Specialized Generics—Struct Type



Performance characteristics like struct types

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

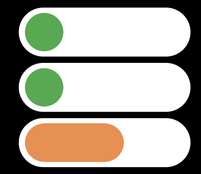
Specialized Generics—Class Type



Performance characteristics like class types

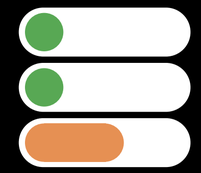
- Heap allocation on creating an instance
- Reference counting
- Dynamic method dispatch through V-Table

Unspecialized Generics—Small Value



No heap allocation: value fits in Value Buffer

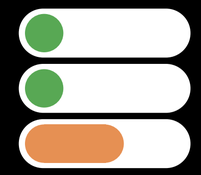
Unspecialized Generics—Small Value



No heap allocation: value fits in Value Buffer

No reference counting

Unspecialized Generics—Small Value

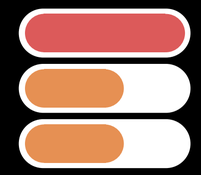


No heap allocation: value fits in Value Buffer

No reference counting

Dynamic dispatch through Protocol Witness Table

Unspecialized Generics—Large Value



Heap allocation (use indirect storage as a workaround)

Reference counting if value contains references

Dynamic dispatch through Protocol Witness Table

Summary

Choose fitting abstraction with the least dynamic runtime type requirements

Summary

Choose fitting abstraction with the least dynamic runtime type requirements

- struct types: value semantics

Summary

Choose fitting abstraction with the least dynamic runtime type requirements

- struct types: value semantics
- class types: identity or OOP style polymorphism

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

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

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

Related Sessions

What's New in Swift	Presidio	Tuesday 9:00AM
What's New in Foundation for Swift	Mission	Tuesday 4:00 PM
Protocol And Value Oriented Programming in UIKit Apps	Presidio	Friday 4:00 PM
Protocol-Oriented Programming in Swift		WWDC 2015
Building Better Apps with Value Types in Swift		WWDC 2015
Optimizing Swift Performance		WWDC 2015

Labs

Swift Open Hours	Dev Tools Lab A	Friday 12:00PM

WWDC16