Developer Tools

#WWDC15

Building Better Apps with Value Types in Swift

Session 414

Doug Gregor Language Lawyer
Bill Dudney Arranger of Bits

Roadmap

Reference semantics

Immutability

Value semantics

Value types in practice

Mixing value types and reference types

Reference Semantics

A Temperature Class

```
class Temperature {
  var celsius: Double = 0
  var fahrenheit: Double {
    get { return celsius * 9 / 5 + 32 }
    set { celsius = (newValue - 32) * 5 / 9 }
  }
}
```

Using Our Temperature Class

```
let home = House()
let temp = Temperature()
temp.fahrenheit = 75
home.thermostat.temperature = temp
```

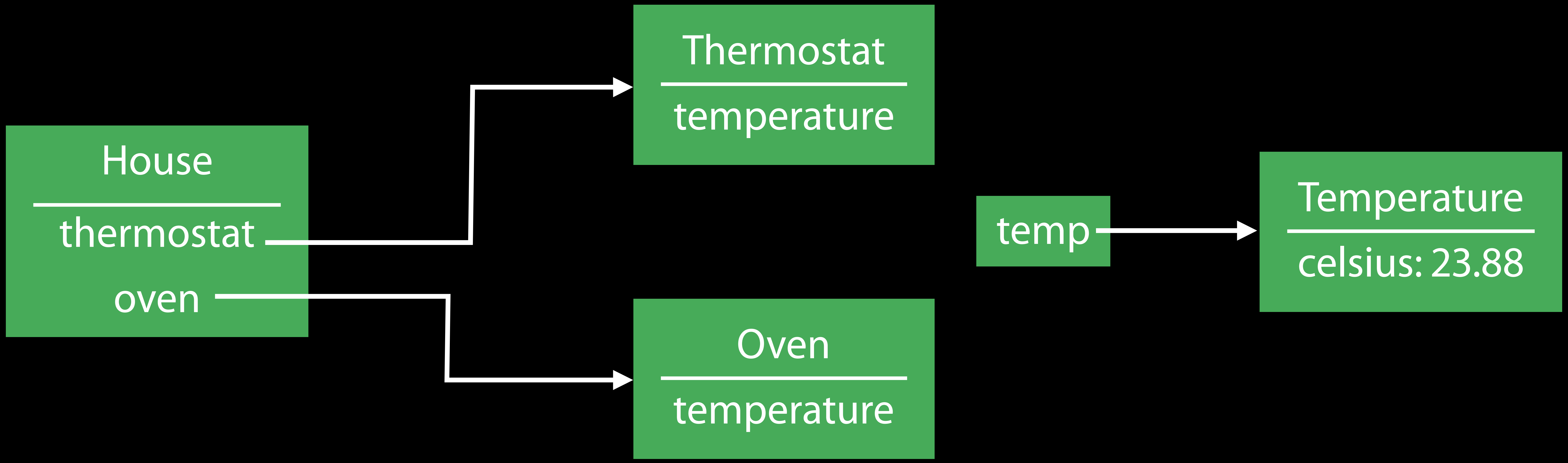
Using Our Temperature Class

```
let home = House()
let temp = Temperature()
temp.fahrenheit = 75
home.thermostat.temperature = temp

temp.fahrenheit = 425
home.oven.temperature = temp
home.oven.bake()
```

Why Is It So Hot in Here?

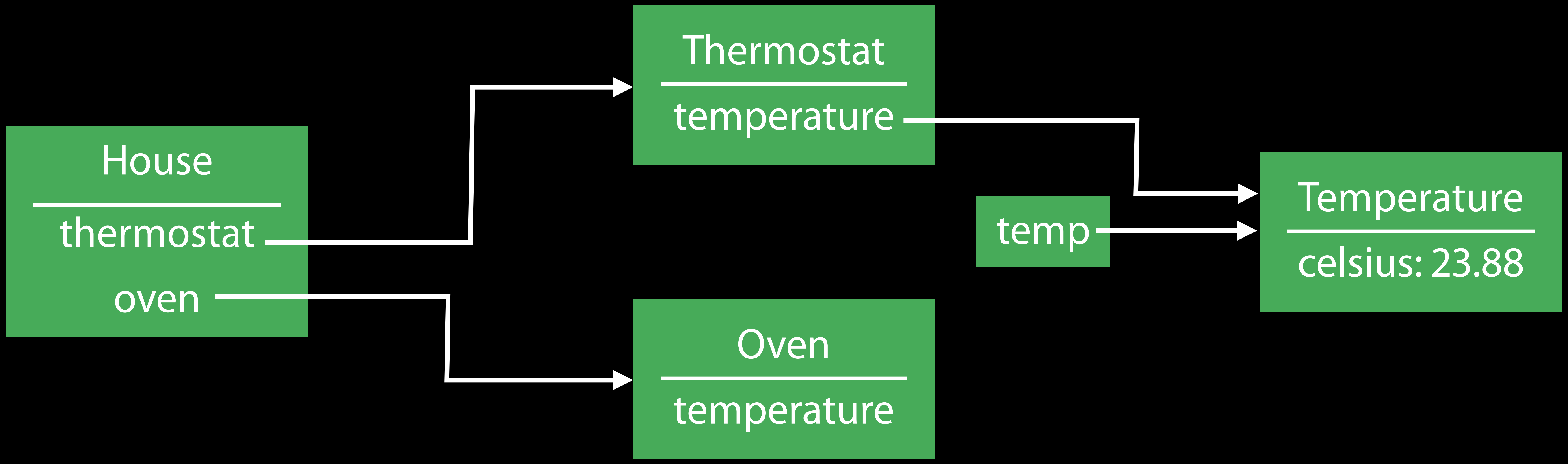
Unintended Sharing

```
let home = House()
let temp = Temperature()
temp.fahrenheit = 75
```



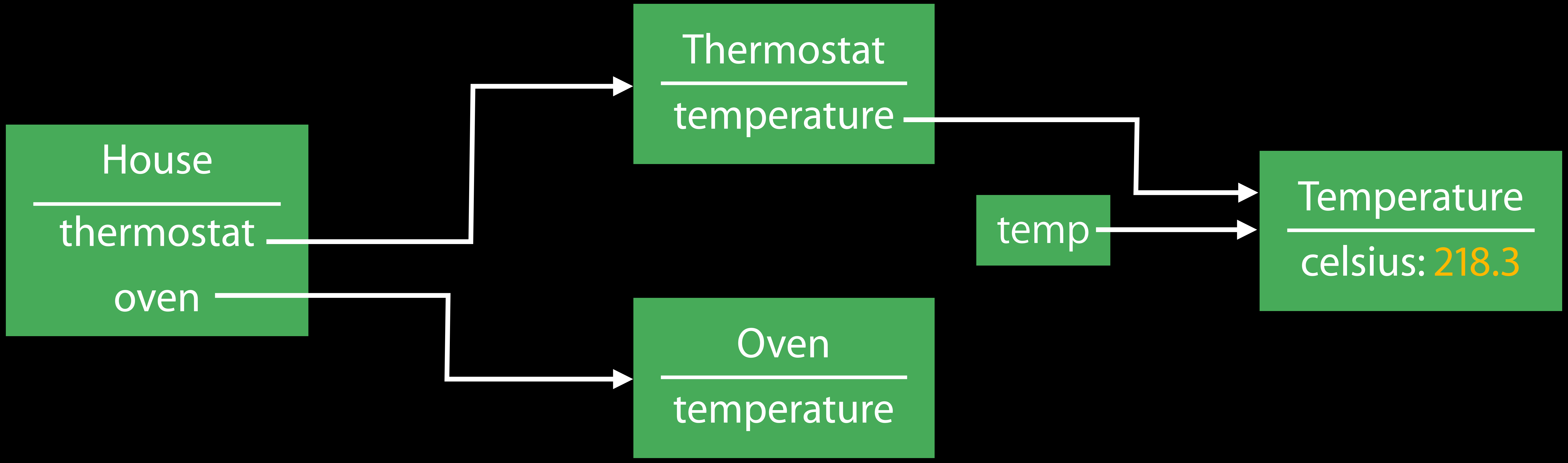
Unintended Sharing

```
let home = House()
let temp = Temperature()
temp.fahrenheit = 75
home.thermostat.temperature = temp
```



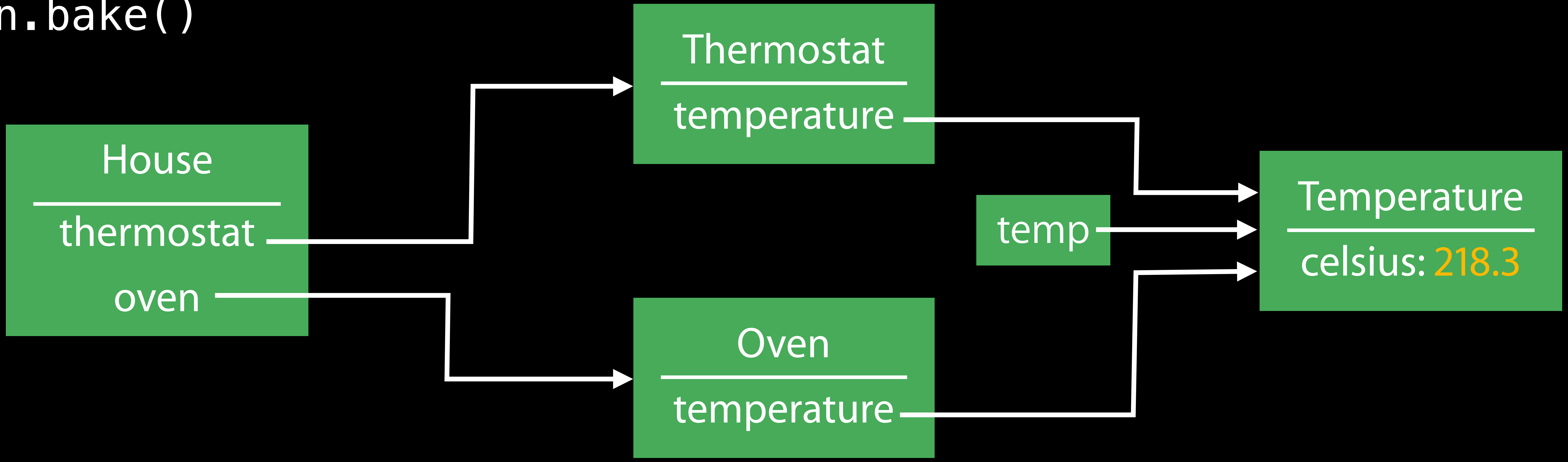
Unintended Sharing

```
let home = House()
let temp = Temperature()
temp.fahrenheit = 75
home.thermostat.temperature = temp

temp.fahrenheit = 425
```



Unintended Sharing

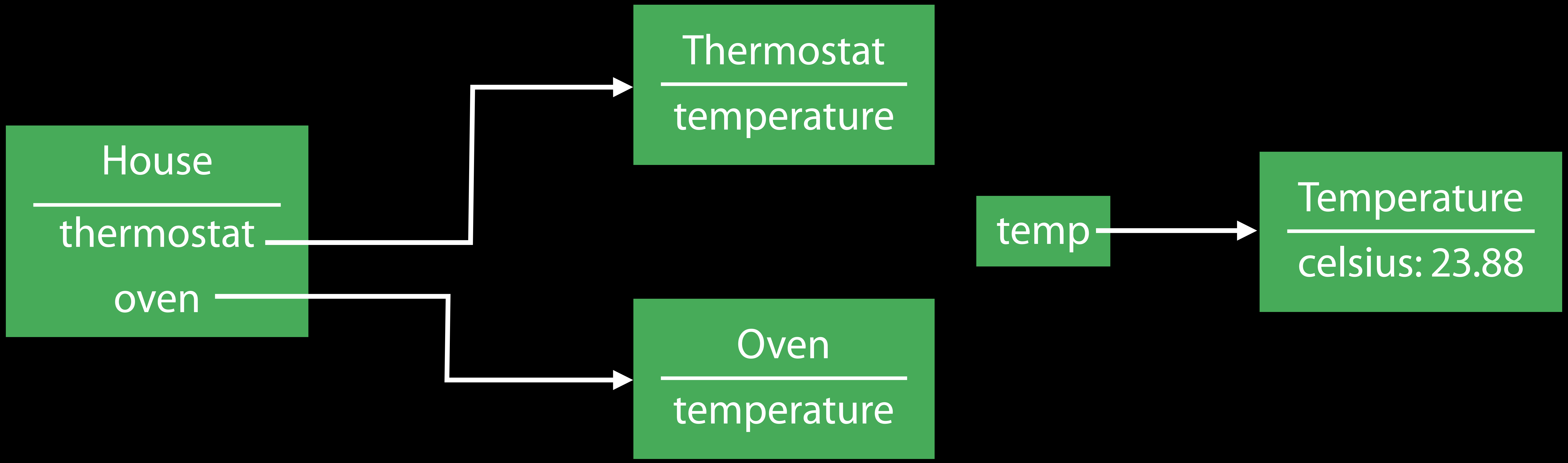
```
let home = House()
let temp = Temperature()
temp.fahrenheit = 75
home.thermostat.temperature = temp

temp.fahrenheit = 425
home.oven.temperature = temp
home.oven.bake()
```



Copy When You Need It

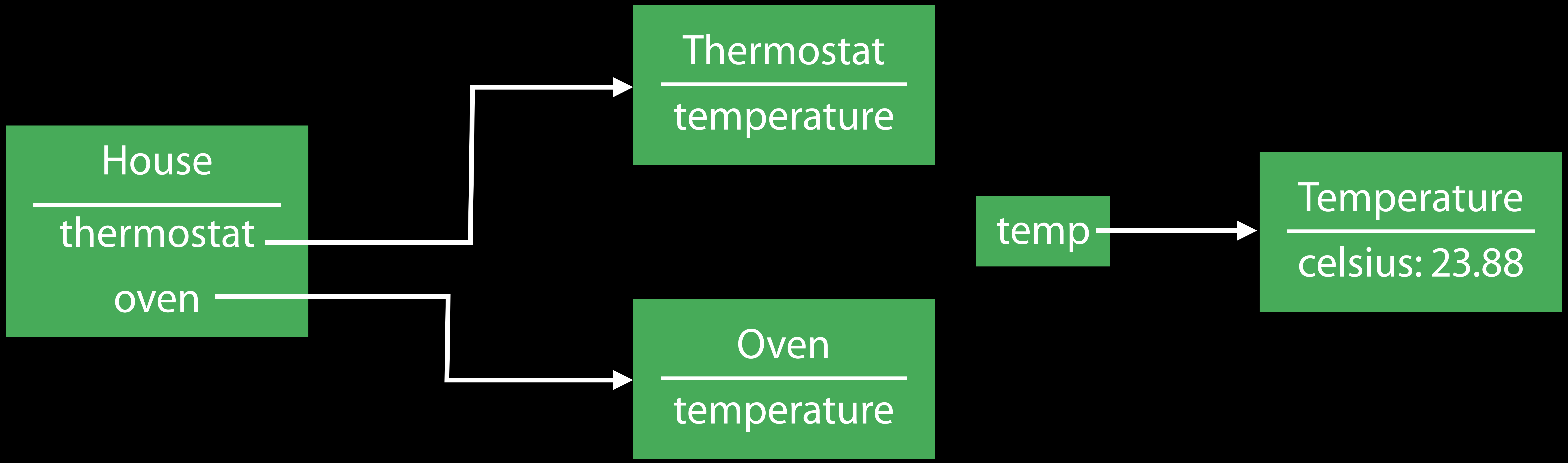
Manual Copying

```
let home = House()
let temp = Temperature()
temp.fahrenheit = 75
```



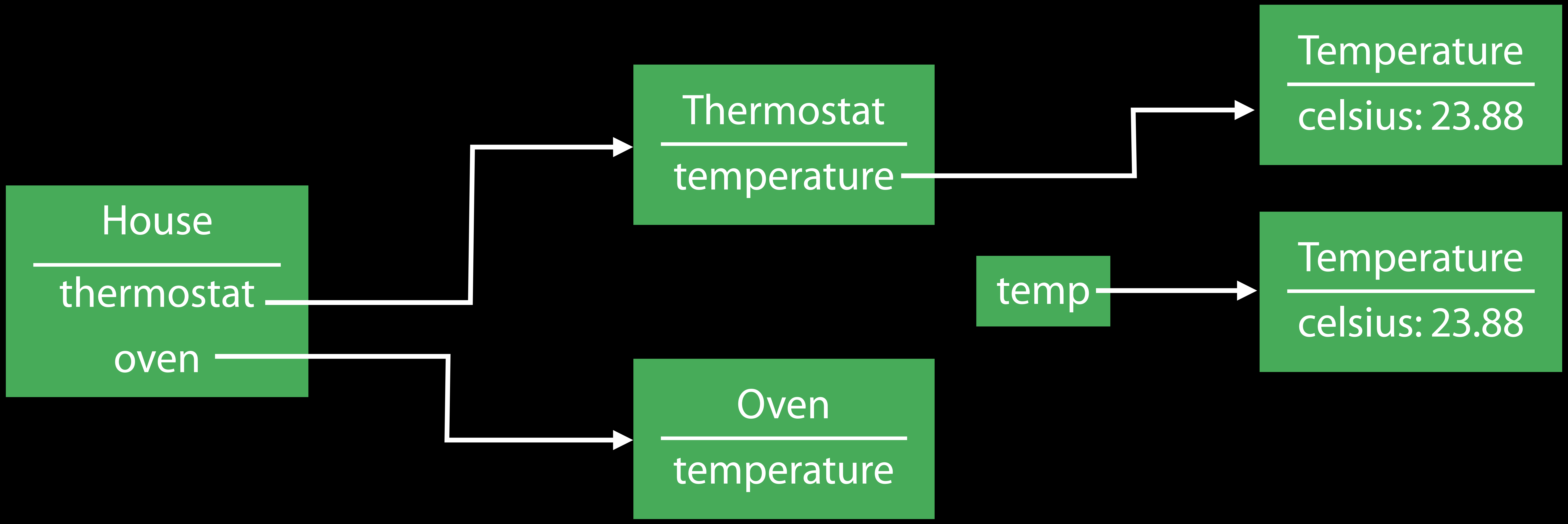
Manual Copying

```
let home = House()
let temp = Temperature()
temp.fahrenheit = 75
home.thermostat.temperature = temp.copy()
```



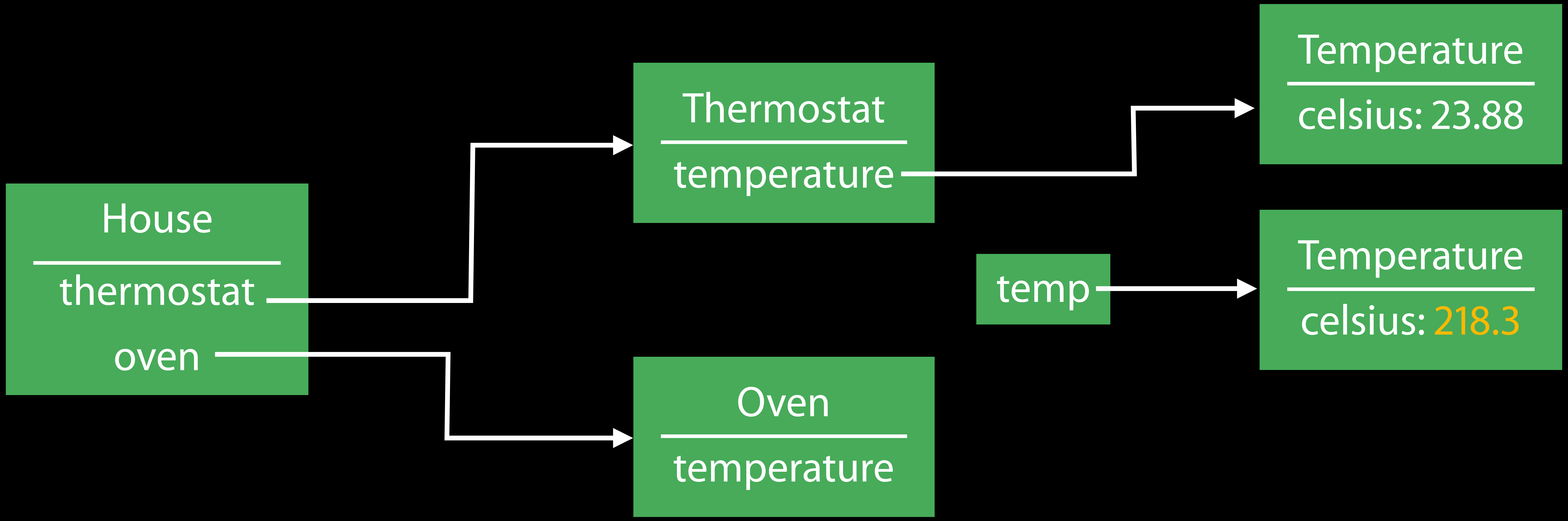
Manual Copying

```
let home = House()
let temp = Temperature()
temp.fahrenheit = 75
home.thermostat.temperature = temp.copy()
```



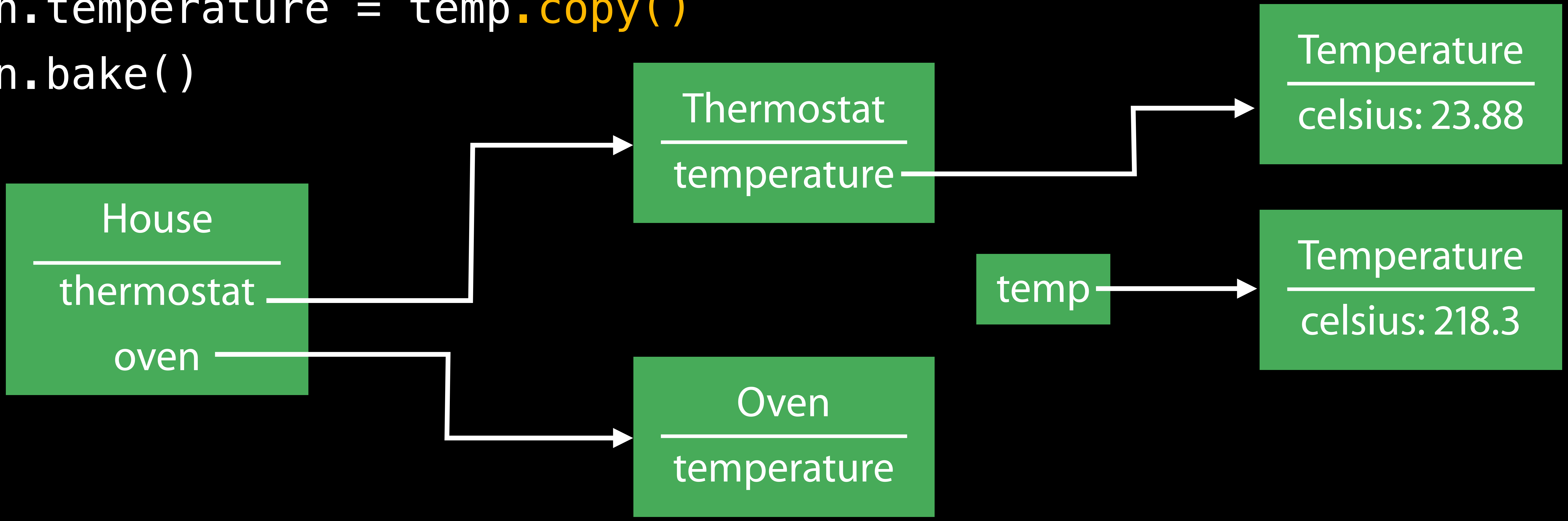
Manual Copying

```
let home = House()
let temp = Temperature()
temp.fahrenheit = 75
home.thermostat.temperature = temp.copy()

temp.fahrenheit = 425
```



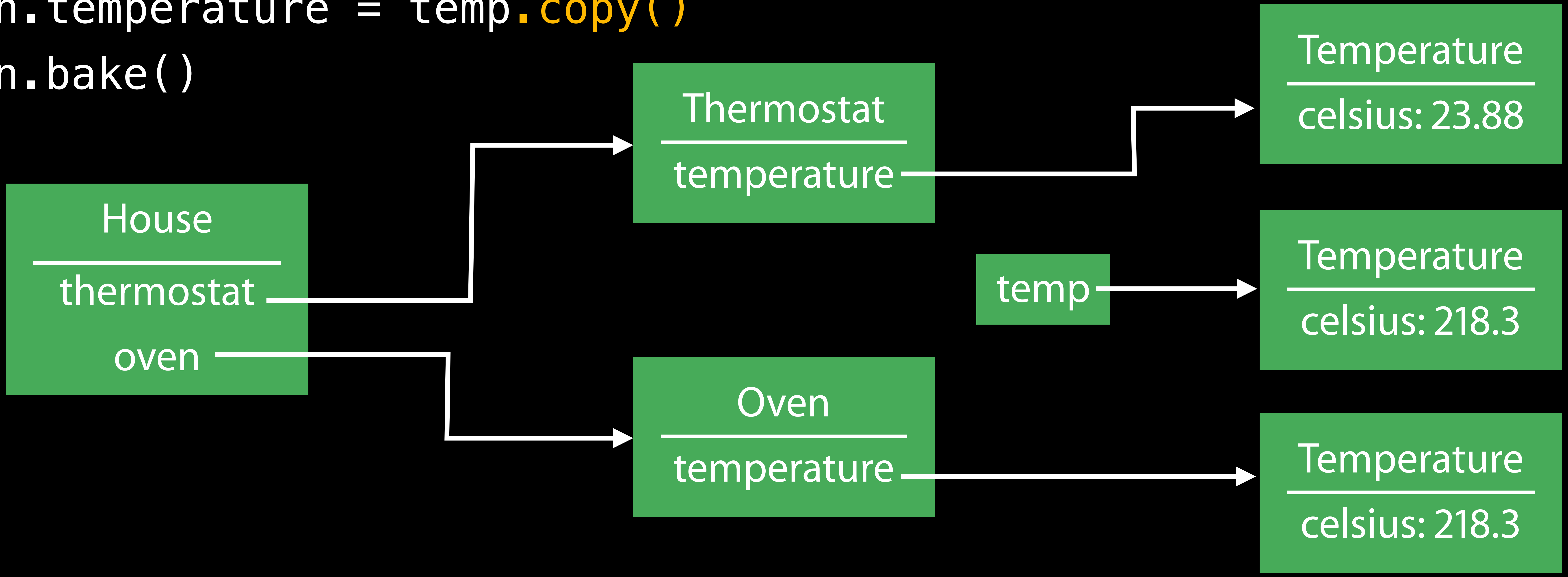
Manual Copying

```
let home = House()
let temp = Temperature()
temp.fahrenheit = 75
home.thermostat.temperature = temp.copy()

temp.fahrenheit = 425
home.oven.temperature = temp.copy()
home.oven.bake()
```



Manual Copying

```
let home = House()
let temp = Temperature()
temp.fahrenheit = 75
home.thermostat.temperature = temp.copy()

temp.fahrenheit = 425
home.oven.temperature = temp.copy()
home.oven.bake()
```



Defensive Copying

```
class Oven {
  var _temperature: Temperature = Temperature(celsius: 0)

  var temperature: Temperature {
    get { return _temperature }
    set { _temperature = newValue.copy() }
  }
}
```

Defensive Copying

```
class Thermostat {
  var _temperature: Temperature = Temperature(celsius: 0)

  var temperature: Temperature {
    get { return _temperature }
    set { _temperature = newValue.copy() }
  }
}
```

Copying in Cocoa[Touch] and Objective-C

Cocoa[Touch] requires copying throughout

- `NSCopying` codifies copying an object
- `NSString`, `NSArray`, `NSDictionary`, `NSURLRequest`, etc. all require copying

Defensive Copying in Cocoa and Objective-C

Cocoa[Touch] requires copying throughout

- `NSCopying` codifies copying an object
- `NSString`, `NSArray`, `NSDictionary`, `NSURLRequest`, etc. all require copying

Defensive copying pervades Cocoa[Touch] and Objective-C

- `NSDictionary` calls `-copy` on its keys
- Property `copy` attribute provides defensive copying on assignment

Defensive Copying in Cocoa and Objective-C

Cocoa[Touch] requires copying throughout

- `NSCopying` codifies copying an object
- `NSString`, `NSArray`, `NSDictionary`, `NSURLRequest`, etc. all require copying

Defensive copying pervades Cocoa[Touch] and Objective-C

- `NSDictionary` calls `-copy` on its keys
- Property `copy` attribute provides defensive copying on assignment

It's still not enough…bugs abound due to missed copies

Is Immutability the Answer?

Eliminating Mutation

Functional programming languages have reference semantics with immutability

Eliminates many problems caused by reference semantics with mutation

- No worries about unintended side effects

Eliminating Mutation

Functional programming languages have reference semantics with immutability

Eliminates many problems caused by reference semantics with mutation

- No worries about unintended side effects

Several notable disadvantages

- Can lead to awkward interfaces
- Does not map efficiently to the machine model

An Immutable Temperature Class

```
class Temperature {
  let celsius: Double = 0
  var fahrenheit: Double { return celsius * 9 / 5 + 32 }

  init(celsius: Double) { self.celsius = celsius }
  init(fahrenheit: Double) { self.celsius = (fahrenheit - 32) * 5 / 9 }
}
```

Awkward Immutable Interfaces

With mutability

```
home.oven.temperature.fahrenheit += 10.0
```

Awkward Immutable Interfaces

With mutability

```
home.oven.temperature.fahrenheit += 10.0
```

Without mutability

```
let temp = home.oven.temperature
home.oven.temperature = Temperature(fahrenheit: temp.fahrenheit + 10.0)
```

Awkward Immutable Interfaces

With mutability

```
home.oven.temperature.fahrenheit += 10.0
```

Without mutability

```
let temp = home.oven.temperature
home.oven.temperature = Temperature(fahrenheit: temp.fahrenheit + 10.0)
```

Sieve of Eratosthenes

```
func primes(n: Int) -> [Int] {
  var numbers = [Int](2..<n)
  for i in 0..<n-2 {
    guard let prime = numbers[i] where prime > 0 else { continue }
    for multiple in stride(from: 2 * prime-2, to: n-2, by: prime) {
      numbers[multiple] = 0
    }
  }
  return numbers.filter { $0 > 0 }
}
```

Sieve of Eratosthenes

```
func primes(n: Int) -> [Int] {
  var numbers = [Int](2..<n)
  for i in 0..<n-2 {
    guard let prime = numbers[i] where prime > 0 else { continue }
    for multiple in stride(from: 2 * prime-2, to: n-2, by: prime) {
      numbers[multiple] = 0
    }
  }
  return numbers.filter { $0 > 0 }
}
```

Sieve of Eratosthenes

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

```swift
func primes(n: Int) -> [Int] {
  var numbers = [Int](2..<n)
  for i in 0..<n-2 {
    guard let prime = numbers[i] where prime > 0 else { continue }
    for multiple in stride(from: 2 * prime-2, to: n-2, by: prime) {
      numbers[multiple] = 0
    }
  }
  return numbers.filter { $0 > 0 }
}
```

Sieve of Eratosthenes

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

```
func primes(n: Int) -> [Int] {
  var numbers = [Int](2..<n)
  for i in 0..<n-2 {
    guard let prime = numbers[i] where prime > 0 else { continue }
    for multiple in stride(from: 2 * prime-2, to: n-2, by: prime) {
      numbers[multiple] = 0
    }
  }
  return numbers.filter { $0 > 0 }
}
```

Sieve of Eratosthenes

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

```swift
func primes(n: Int) -> [Int] {
  var numbers = [Int](2..<n)
  for i in 0..<n-2 {
    guard let prime = numbers[i] where prime > 0 else { continue }
    for multiple in stride(from: 2 * prime-2, to: n-2, by: prime) {
      numbers[multiple] = 0
    }
  }
  return numbers.filter { $0 > 0 }
}
```

Sieve of Eratosthenes

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

```
func primes(n: Int) -> [Int] {
  var numbers = [Int](2..<n)
  for i in 0..<n-2 {
    guard let prime = numbers[i] where prime > 0 else { continue }
    for multiple in stride(from: 2 * prime-2, to: n-2, by: prime) {
      numbers[multiple] = 0
    }
  }
  return numbers.filter { $0 > 0 }
}
```

Sieve of Eratosthenes

2	3	0	5	0	7	0	9	0	11	0	13	0	15	0	17	0	19	0

```swift
func primes(n: Int) -> [Int] {
  var numbers = [Int](2..<n)
  for i in 0..<n-2 {
    guard let prime = numbers[i] where prime > 0 else { continue }
    for multiple in stride(from: 2 * prime-2, to: n-2, by: prime) {
      numbers[multiple] = 0
    }
  }
  return numbers.filter { $0 > 0 }
}
```

Sieve of Eratosthenes

2	3	0	5	0	7	0	9	0	11	0	13	0	15	0	17	0	19	0

```
func primes(n: Int) -> [Int] {
  var numbers = [Int](2..<n)
  for i in 0..<n-2 {
    guard let prime = numbers[i] where prime > 0 else { continue }
    for multiple in stride(from: 2 * prime-2, to: n-2, by: prime) {
      numbers[multiple] = 0
    }
  }
  return numbers.filter { $0 > 0 }
}
```

Sieve of Eratosthenes

2	3	0	5	0	7	0	9	0	11	0	13	0	15	0	17	0	19	0

```
func primes(n: Int) -> [Int] {
  var numbers = [Int](2..<n)
  for i in 0..<n-2 {
    guard let prime = numbers[i] where prime > 0 else { continue }
    for multiple in stride(from: 2 * prime-2, to: n-2, by: prime) {
      numbers[multiple] = 0
    }
  }
  return numbers.filter { $0 > 0 }
}
```

Sieve of Eratosthenes

2	3	0	5	0	7	0	0	0	11	0	13	0	0	0	17	0	19	0

```
func primes(n: Int) -> [Int] {
  var numbers = [Int](2..<n)
  for i in 0..<n-2 {
    guard let prime = numbers[i] where prime > 0 else { continue }
    for multiple in stride(from: 2 * prime-2, to: n-2, by: prime) {
      numbers[multiple] = 0
    }
  }
  return numbers.filter { $0 > 0 }
}
```

Sieve of Eratosthenes

2	3	0	5	0	7	0	0	0	11	0	13	0	0	0	17	0	19	0

```
func primes(n: Int) -> [Int] {
  var numbers = [Int](2..<n)
  for i in 0..<n-2 {
    guard let prime = numbers[i] where prime > 0 else { continue }
    for multiple in stride(from: 2 * prime-2, to: n-2, by: prime) {
      numbers[multiple] = 0
    }
  }
  return numbers.filter { $0 > 0 }
}
```

Sieve of Eratosthenes

2	3	0	5	0	7	0	0	0	11	0	13	0	0	0	17	0	19	0

```swift
func primes(n: Int) -> [Int] {
  var numbers = [Int](2..<n)
  for i in 0..<n-2 {
    guard let prime = numbers[i] where prime > 0 else { continue }
    for multiple in stride(from: 2 * prime-2, to: n-2, by: prime) {
      numbers[multiple] = 0
    }
  }
  return numbers.filter { $0 > 0 }
}
```

Sieve of Eratosthenes

2	3	0	5	0	7	0	0	0	11	0	13	0	0	0	17	0	19	0

```
func primes(n: Int) -> [Int] {
  var numbers = [Int](2..<n)
  for i in 0..<n-2 {
    guard let prime = numbers[i] where prime > 0 else { continue }
    for multiple in stride(from: 2 * prime-2, to: n-2, by: prime) {
      numbers[multiple] = 0
    }
  }
  return numbers.filter { $0 > 0 }
}
```

Sieve of Eratosthenes

2	3	0	5	0	7	0	0	0	11	0	13	0	0	0	17	0	19	0

```swift
func primes(n: Int) -> [Int] {
  var numbers = [Int](2..<n)
  for i in 0..<n-2 {
    guard let prime = numbers[i] where prime > 0 else { continue }
    for multiple in stride(from: 2 * prime-2, to: n-2, by: prime) {
      numbers[multiple] = 0
    }
  }
  return numbers.filter { $0 > 0 }
}
```

2	3	5	7	11	13	17	19

Functional Sieve of Eratosthenes

Haskell:

```
primes = sieve [2..]
sieve [] = []
sieve (p : xs) = p : sieve [x | x <- xs, x ‘mod‘ p > 0]
```

Functional Sieve of Eratosthenes

Haskell:

```
primes = sieve [2..]
sieve [] = []
sieve (p : xs) = p : sieve [x | x <- xs, x ‘mod‘ p > 0]
```

Swift:

```
func sieve(numbers: [Int]) -> [Int] {
  if numbers.isEmpty { return [] }
  let p = numbers[0]
  return [p] + sieve(numbers[1..<numbers.count].filter { $0 % p > 0 })
}
```

Functional Sieve of Eratosthenes

Swift:

```swift
func sieve(numbers: [Int]) -> [Int] {
  if numbers.isEmpty { return [] }
  let p = numbers[0]
  return [p] + sieve(numbers[1..<numbers.count].filter { $0 % p > 0 })
}
```

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Functional Sieve of Eratosthenes

Swift:

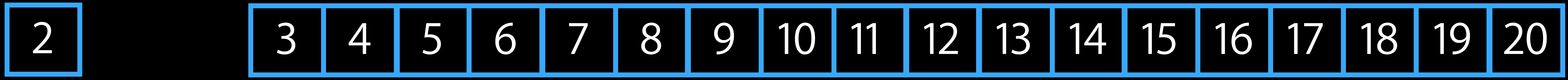
```
func sieve(numbers: [Int]) -> [Int] {
  if numbers.isEmpty { return [] }
  let p = numbers[0]
  return [p] + sieve(numbers[1..<numbers.count].filter { $0 % p > 0 })
}
```

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Functional Sieve of Eratosthenes

Swift:

```
func sieve(numbers: [Int]) -> [Int] {
  if numbers.isEmpty { return [] }
  let p = numbers[0]
  return [p] + sieve(numbers[1..<numbers.count].filter { $0 % p > 0 })
}
```



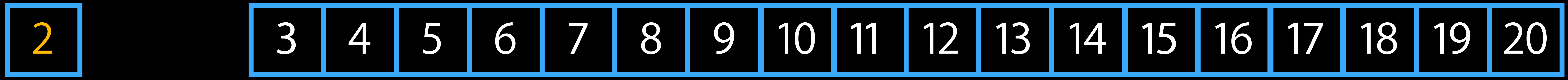
2

3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Functional Sieve of Eratosthenes

Swift:

```swift
func sieve(numbers: [Int]) -> [Int] {
  if numbers.isEmpty { return [] }
  let p = numbers[0]
  return [p] + sieve(numbers[1..<numbers.count].filter { $0 % p > 0 })
}
```



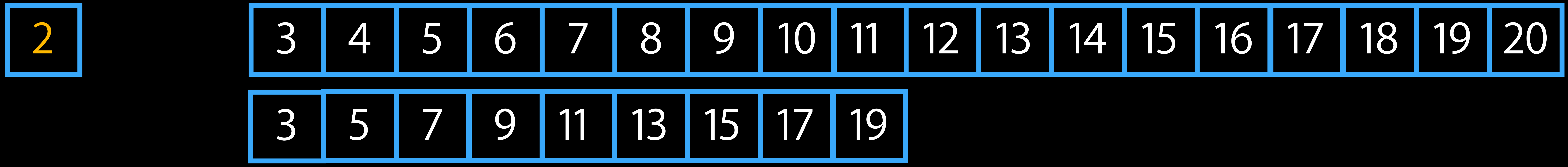
2

3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Functional Sieve of Eratosthenes

Swift:

```swift
func sieve(numbers: [Int]) -> [Int] {
  if numbers.isEmpty { return [] }
  let p = numbers[0]
  return [p] + sieve(numbers[1..<numbers.count].filter { $0 % p > 0 })
}
```



2

3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

3 5 7 9 11 13 15 17 19

Functional Sieve of Eratosthenes

Swift:

```swift
func sieve(numbers: [Int]) -> [Int] {
  if numbers.isEmpty { return [] }
  let p = numbers[0]
  return [p] + sieve(numbers[1..<numbers.count].filter { $0 % p > 0 })
}
```

2

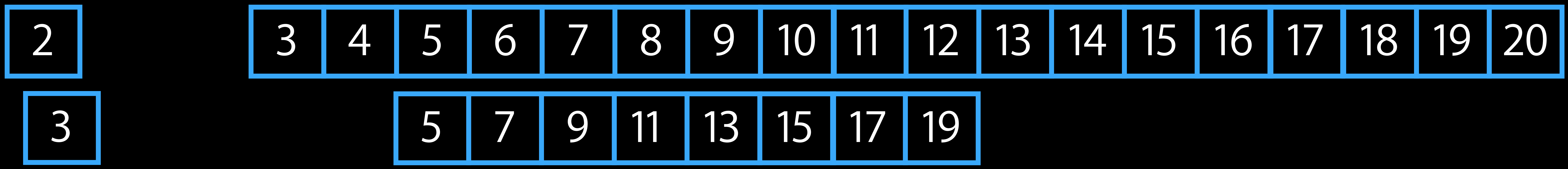
3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

3	5	7	9	11	13	15	17	19

Functional Sieve of Eratosthenes

Swift:

```
func sieve(numbers: [Int]) -> [Int] {
  if numbers.isEmpty { return [] }
  let p = numbers[0]
  return [p] + sieve(numbers[1..<numbers.count].filter { $0 % p > 0 })
}
```



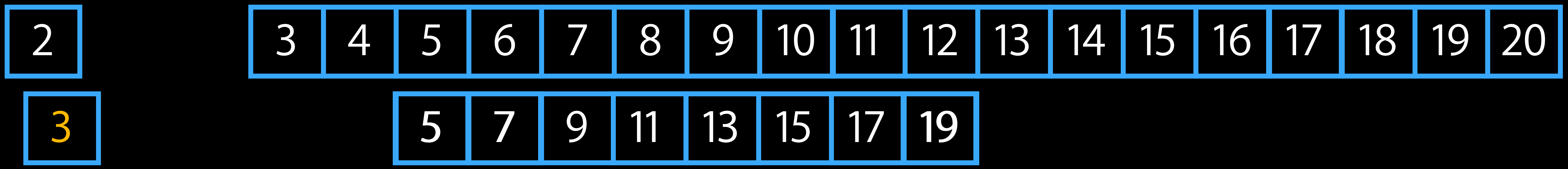
2   3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

3   5 7 9 11 13 15 17 19

Functional Sieve of Eratosthenes

Swift:

```
func sieve(numbers: [Int]) -> [Int] {
  if numbers.isEmpty { return [] }
  let p = numbers[0]
  return [p] + sieve(numbers[1..<numbers.count].filter { $0 % p > 0 })
}
```



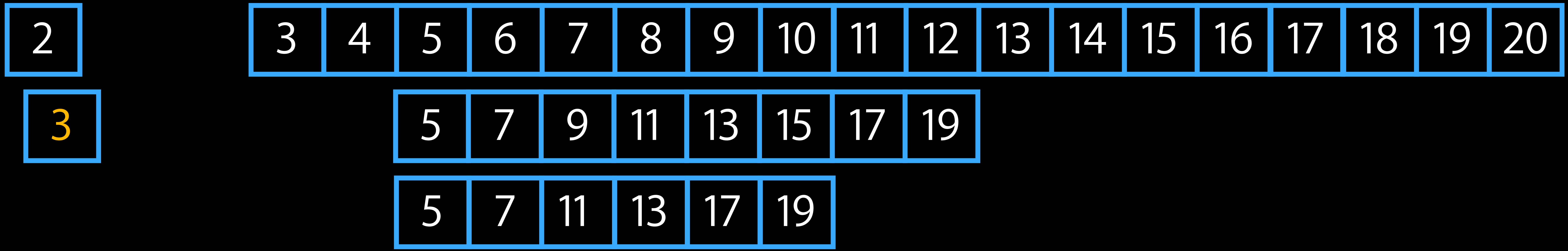
2 | 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

3 | 5 7 9 11 13 15 17 19

Functional Sieve of Eratosthenes

Swift:

```
func sieve(numbers: [Int]) -> [Int] {
  if numbers.isEmpty { return [] }
  let p = numbers[0]
  return [p] + sieve(numbers[1..<numbers.count].filter { $0 % p > 0 })
}
```



2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20

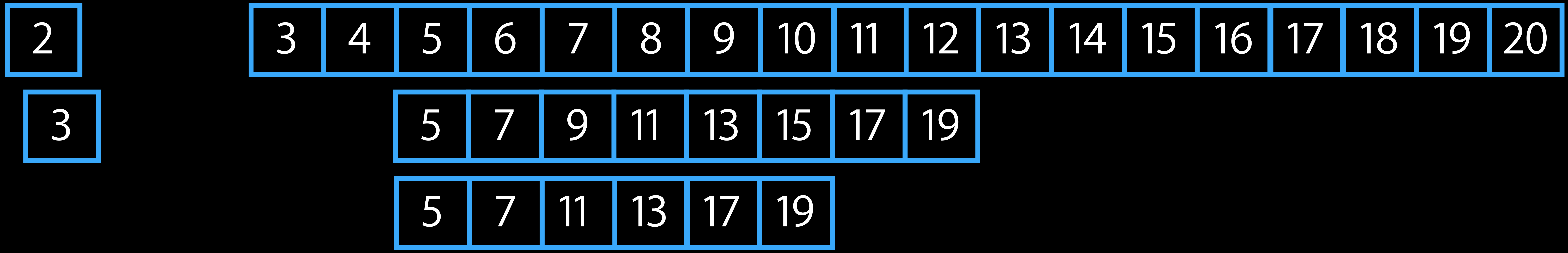
3 | 5 | 7 | 9 | 11 | 13 | 15 | 17 | 19

5 | 7 | 11 | 13 | 17 | 19

Functional Sieve of Eratosthenes

Swift:

```
func sieve(numbers: [Int]) -> [Int] {
  if numbers.isEmpty { return [] }
  let p = numbers[0]
  return [p] + sieve(numbers[1..<numbers.count].filter { $0 % p > 0 })
}
```



2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

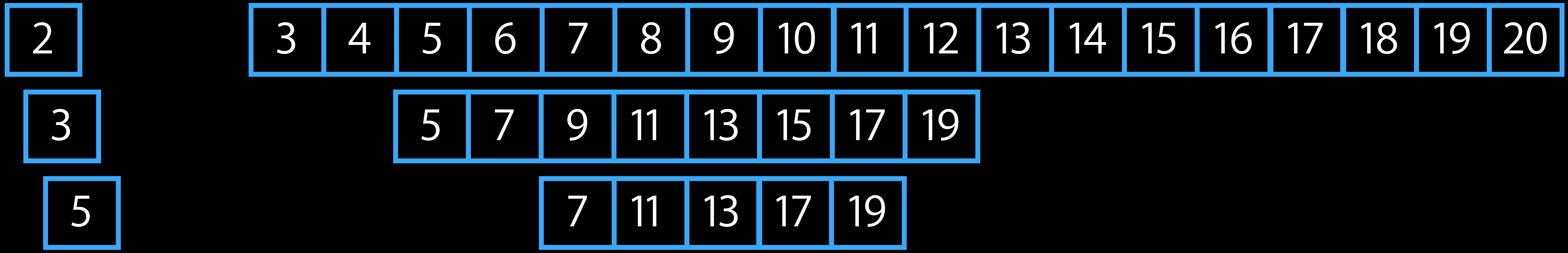
3 5 7 9 11 13 15 17 19

5 7 11 13 17 19

Functional Sieve of Eratosthenes

Swift:

```swift
func sieve(numbers: [Int]) -> [Int] {
  if numbers.isEmpty { return [] }
  let p = numbers[0]
  return [p] + sieve(numbers[1..<numbers.count].filter { $0 % p > 0 })
}
```



2 | 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

3 | 5 7 9 11 13 15 17 19

5 | 7 11 13 17 19

Functional Sieve Is Not the Real Sieve

Performance differences matter

Haskell:

```haskell
primes = sieve [2..]
sieve [] = []
sieve (p : xs) = p : sieve [x | x <- xs, x ‘mod‘ p > 0]
```

Swift:

```swift
func sieve(numbers: [Int]) -> [Int] {
  if numbers.isEmpty { return [] }
  let p = numbers[0]
  return [p] + sieve(numbers[1..<numbers.count].filter { $0 % p > 0 })
}
```

O'Neill, Melissa E. *The Genuine Sieve of Eratosthenes*.
Journal of Functional Programming, Vol. 19, No. 1. (2009), pp. 95-106

Functional Sieve Is Not the Real Sieve

Performance differences matter

Haskell:

```
primes = sieve [2..]
sieve [] = []
sieve (p : xs) = p : sieve [x | x <- xs, x ‘mod‘ p > 0]
```

Swift:

```
func sieve(numbers: [Int]) -> [Int] {
  if numbers.isEmpty { return [] }
  let p = numbers[0]
  return [p] + sieve(numbers[1..<numbers.count].filter { $0 % p > 0 })
}
```

O'Neill, Melissa E. *The Genuine Sieve of Eratosthenes*.
Journal of Functional Programming, Vol. 19, No. 1. (2009), pp. 95-106

Immutability in Cocoa[Touch]

Cocoa[Touch] has a number of immutable classes

- `NSDate`, `NSURL`, `UIImage`, `NSNumber`, etc.
- Improved safety (no need to use copy)

Immutability in Cocoa[Touch]

Cocoa[Touch] has a number of immutable classes

- `NSDate`, `NSURL`, `UIImage`, `NSNumber`, etc.
- Improved safety (no need to use copy)

Downsides to immutability

```
NSURL *url = [[NSURL alloc] initWithString: NSHomeDirectory()];
NSString *component;
while ((component = getNextSubdir()) {
  url = [url URLByAppendingPathComponent: component];
}
```

Immutability in Cocoa[Touch]

Cocoa[Touch] has a number of immutable classes

- `NSDate`, `NSURL`, `UIImage`, `NSNumber`, etc.
- Improved safety (no need to use copy)

Downsides to immutability

```
NSArray<NSString *> *array = [NSArray arrayWithObject: NSHomeDirectory()];
NSString *component;
while ((component = getNextSubdir()) {
  array = [array arrayByAddingObject: component];
}
url = [NSURL fileURLWithPathComponents: array];
```

Thoughtful Mutability in Cocoa[Touch]

You'd miss it if it were gone

Cocoa[Touch] has a number of immutable classes

- `NSDate`, `NSURL`, `UIImage`, `NSNumber`, etc.
- Improved safety (no need to use copy)

Thoughtful mutability

```
NSMutableArray<NSString *> *array = [NSMutableArray array];
[array addObject: NSHomeDirectory()];
NSString *component;
while ((component = getNextSubdir()) {
  [array addObject: component];
}
url = [NSURL fileURLWithPathComponents: array];
```

Value Semantics

Variables Are Logically Distinct

Integers are value types

Mutating one variable of some value type will never affect a different variable

```
var a: Int = 17
var b = a
assert(a == b)

b += 25

print("a = \(a), b = \(b)") // a = 17, b = 42
```

Variables Are Logically Distinct

CGPoints are value types

Mutating one variable of some value type will never affect a different variable

```
var a: CGPoint = CGPoint(x: 3, y: 5)
var b = a
assert(a == b)

b.x = 17

print("a = \(a), b = \(b)") // a = (x = 3, y = 5), b = (x = 17, y = 5)
```

Variables Are Logically Distinct

Strings are value types

Mutating one variable of some value type will never affect a different variable

```
var a: String = "Hello"
var b = a
assert(a == b)

b.extend(" WWDC!")

print("a = \(a), b = \(b)") // a = Hello, b = Hello WWDC!
```

Variables Are Logically Distinct

Arrays are value types

Mutating one variable of some value type will never affect a different variable

```
var a: [Int] = [1, 2, 3, 4, 5]
var b = a
assert(a == b)

b[2] = 17

print("a = \(a), b = \(b)") // a = [1, 2, 3, 4, 5], b = [1, 2, 17, 4, 5]
```

Variables Are Logically Distinct

Dictionaries are value types

Mutating one variable of some value type will never affect a different variable

```
var a: [Int : String] = [1 : "uno", 2 : "dos"]
var b = a
assert(a == b)

b[2] = "due"

print("a = \(a), b = \(b)") // a = [1 : "uno", 2 : "dos"],
                            // b = [1 : "uno", 2 : "due"]
```

Value Types Compose

All of Swift's "fundamental" types are value types

- `Int`, `Double`, `String`, …

Value Types Compose

All of Swift's "fundamental" types are value types

- `Int`, `Double`, `String`, …

All of Swift's collections are value types

- `Array`, `Set`, `Dictionary`, …

Value Types Compose

All of Swift's "fundamental" types are value types

- `Int`, `Double`, `String`, …

All of Swift's collections are value types

- `Array`, `Set`, `Dictionary`, …

Swift tuples, structs, and enums that contain value types are value types

Value Types Are Distinguished by Value

Equality is established by value of a variable

- Not its identity
- Not how we arrived at the value

```
var a: Int = 5
var b: Int = 2 + 3
assert(a == b)
```

Value Types Are Distinguished by Value

Equality is established by value of a variable

- Not its identity
- Not how we arrived at the value

```
var a: CGPoint = CGPoint(x: 3, y: 5)
var b: CGPoint = CGPoint(x: 1, y: 3)
b.x += 2
b.y += 2
assert(a == b)
```

Value Types Are Distinguished by Value

Equality is established by value of a variable

- Not its identity
- Not how we arrived at the value

```
var a: String = "Hello WWDC!"
var b: String = "Hello"
b += " "
b += "WWDC!"
assert(a == b)
```

Value Types Are Distinguished by Value

Equality is established by value of a variable

- Not its identity
- Not how we arrived at the value

```
var a: [Int] = [1, 2, 3]
var b: [Int] = [3, 2, 1].sort(<)
assert(a == b)
```

Equatable

Value types should implement Equatable

```
protocol Equatable {
    /// Reflexive - `x == x` is `true`
    /// Symmetric - `x == y` then `y == x`
    /// Transitive - `x == y` and `y == z` then `x == z`
   func ==(lhs: Self, rhs: Self) -> Bool
}
```

Equatable

Value types should implement Equatable

```
protocol Equatable {
    /// Reflexive - `x == x` is `true`
    /// Symmetric - `x == y` then `y == x`
    /// Transitive - `x == y` and `y == z` then `x == z`
   func ==(lhs: Self, rhs: Self) -> Bool
}

var a = …
var b = a
assert(a == b)
assert(b == a)
var c = b
assert(c == a)
```

Implementing Equatable

```
protocol Equatable {
    /// Reflexive - `x == x` is `true`
    /// Symmetric - `x == y` then `y == x`
    /// Transitive - `x == y` and `y == z` then `x == z`
   func ==(lhs: Self, rhs: Self) -> Bool
}

extension CGPoint: Equatable { }

func ==(lhs: CGPoint, rhs: CGPoint) -> Bool {
  return lhs.x == rhs.x && lhs.y == rhs.y
}
```

Value Semantics Temperature

```swift
struct Temperature: Equatable {
  var celsius: Double = 0
  var fahrenheit: Double {
    get { return celsius * 9 / 5 + 32 }
    set { celsius = (newValue - 32) * 5 / 9 }
  }
}

func ==(lhs: Temperature, rhs: Temperature) -> Bool {
  return lhs.celsius == rhs.celsius
}
```

Using Value Semantics Temperature

```
let home = House()
let temp = Temperature()
temp.fahrenheit = 75
home.thermostat.temperature = temp

temp.fahrenheit = 425
home.oven.temperature = temp
home.oven.bake()
```

Using Value Semantics Temperature

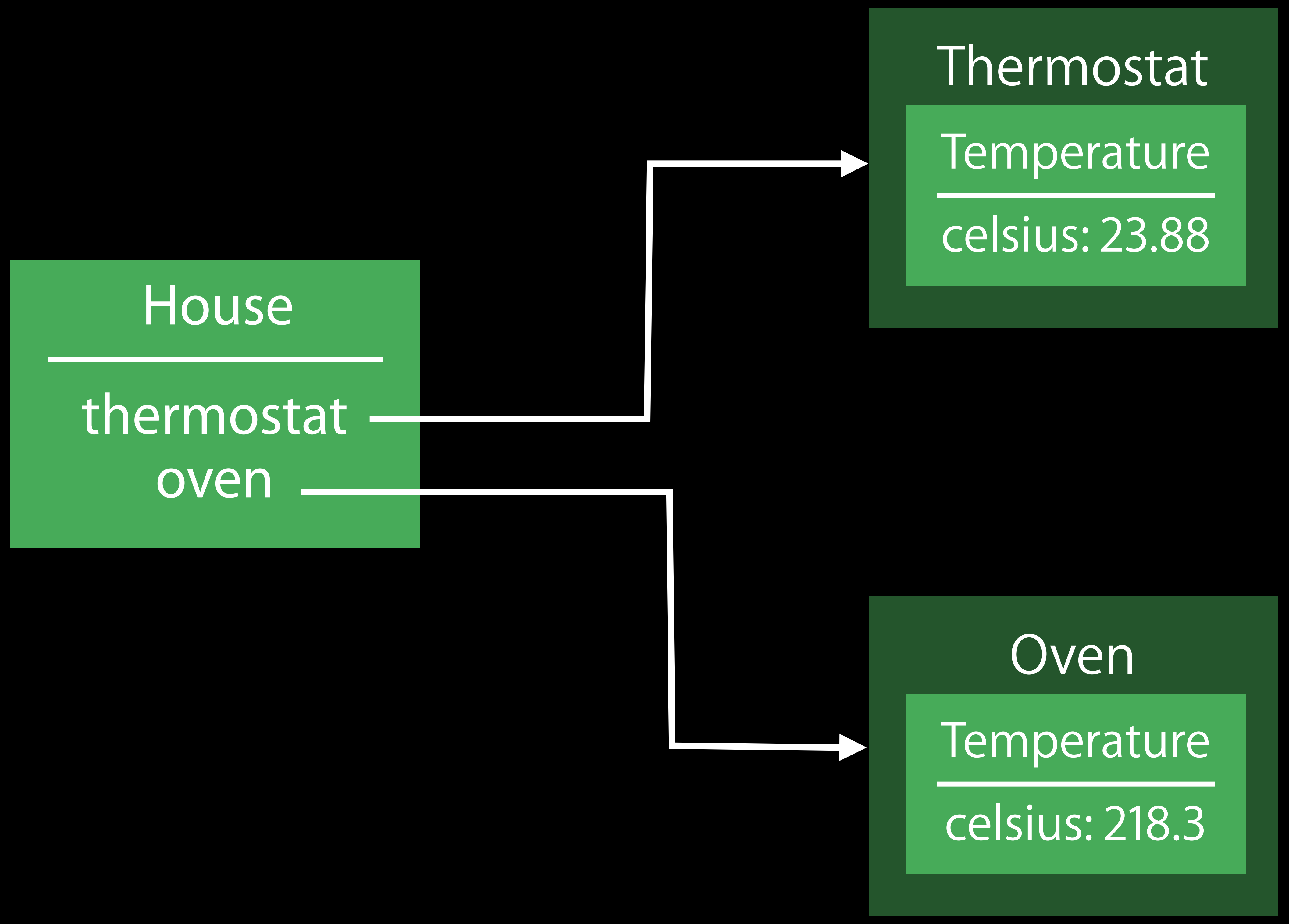
```
let home = House()
let temp = Temperature()
temp.fahrenheit = 75
home.thermostat.temperature = temp
       error: cannot assign to property: ‘temp’ is a ‘let’ constant

temp.fahrenheit = 425
home.oven.temperature = temp
home.oven.bake()
```

Using Value Semantics Temperature

```
let home = House()
var temp = Temperature()
temp.fahrenheit = 75
home.thermostat.temperature = temp

temp.fahrenheit = 425
home.oven.temperature = temp
home.oven.bake()
```

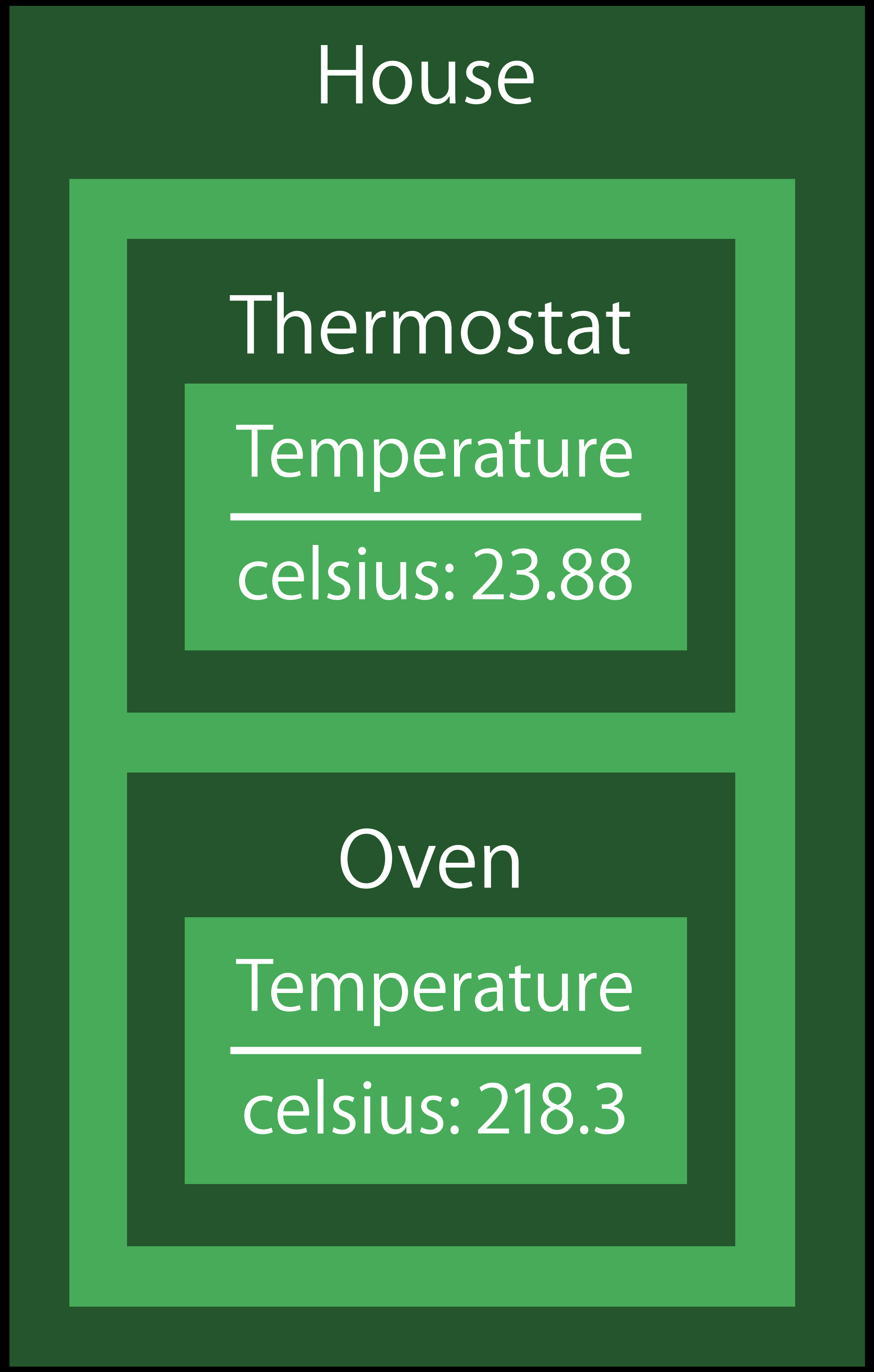
Using Value Semantics Temperature

```
let home = House()
var temp = Temperature()
temp.fahrenheit = 75
home.thermostat.temperature = temp

temp.fahrenheit = 425
home.oven.temperature = temp
home.oven.bake()
```



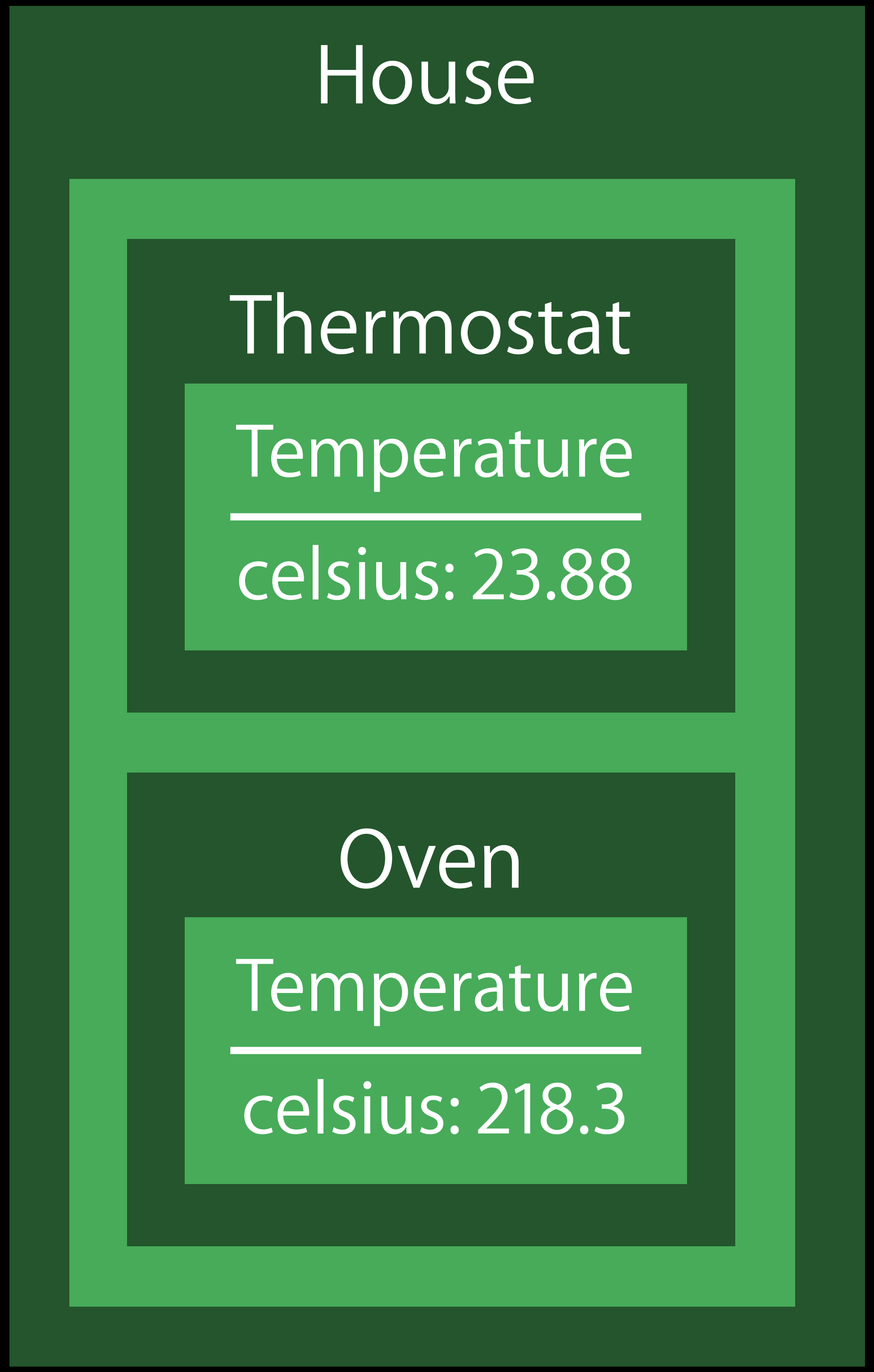
Using Value Semantics ~~Temperature~~ Everywhere

```
let home = House()
var temp = Temperature()
temp.fahrenheit = 75
home.thermostat.temperature = temp

temp.fahrenheit = 425
home.oven.temperature = temp
home.oven.bake()
```



Using Value Semantics ~~Temperature~~ Everywhere

```
var home = House()
var temp = Temperature()
temp.fahrenheit = 75
home.thermostat.temperature = temp

temp.fahrenheit = 425
home.oven.temperature = temp
home.oven.bake()
```



Mutation When You Want It

But not when you don’t

`let` means “the value will never change”

```
let numbers = [1, 2, 3, 4, 5]
```

`var` means you can update the value without affecting any other values

```
var strings = [String]()
for x in numbers {
  strings.append(String(x))
}
```

Freedom from Race Conditions

```
var numbers = [1, 2, 3, 4, 5]
scheduler.processNumbersAsynchronously(numbers)
for i in 0..<numbers.count { numbers[i] = numbers[i] * i }
scheduler.processNumbersAsynchronously(numbers)
```

Performance

What about all those copies?

```
var numbers = [1, 2, 3, 4, 5]
scheduler.processNumbersAsynchronously(numbers)
for i in 0..<numbers.count { numbers[i] = numbers[i] * i }
scheduler.processNumbersAsynchronously(numbers)
```

Copies Are Cheap

Copies Are Cheap

Constant time

Copies Are Cheap

Constant time

Copying a low-level, fundamental type is constant time

- `Int`, `Double`, etc.

Copies Are Cheap

Constant time

Copying a low-level, fundamental type is constant time

- `Int`, `Double`, etc.

Copying a struct, enum, or tuple of value types is constant time

- `CGPoint`, etc.

Copies Are Cheap

Constant time

Copying a low-level, fundamental type is constant time

- `Int`, `Double`, etc.

Copying a struct, enum, or tuple of value types is constant time

- `CGPoint`, etc.

Extensible data structures use copy-on-write

- Copying involves a fixed number of reference-counting operations
- `String`, `Array`, `Set`, `Dictionary`, etc.

Value Semantics Are Simple and Efficient

Different variables are logically distinct

Mutability when you want it

Copies are cheap

Value Types in Practice

Conceptualize an Example

A Diagram Made of Value Types

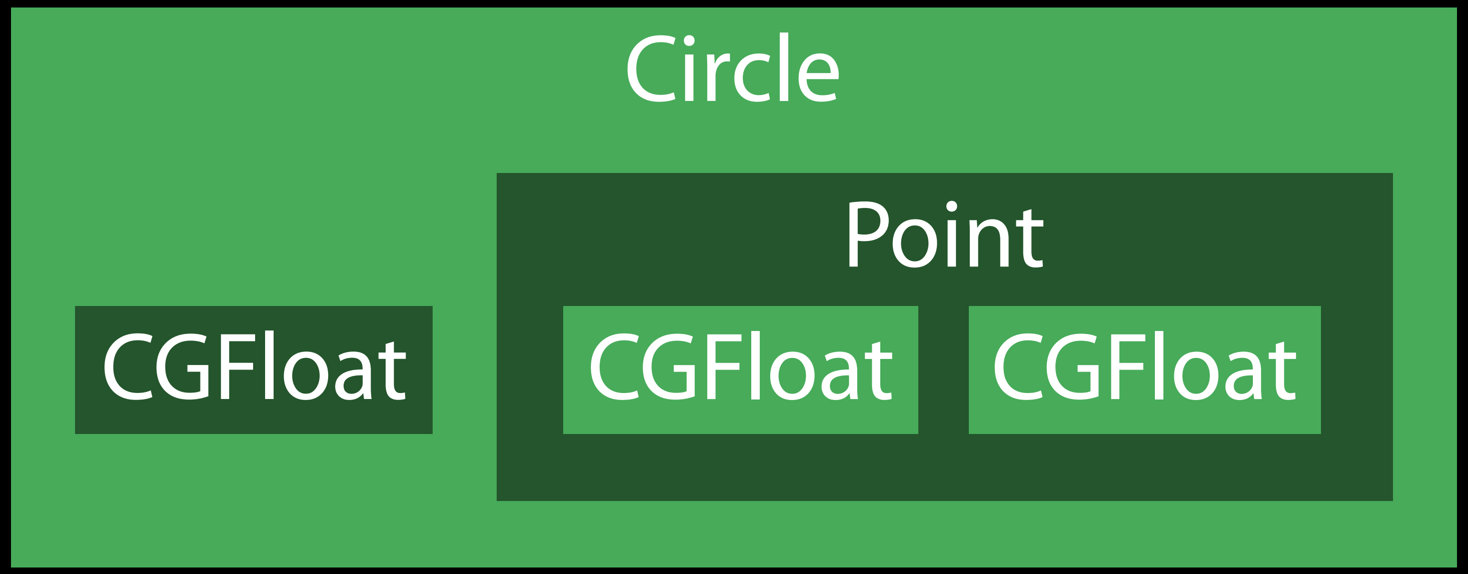
Circle

Polygon

Diagram

Circle

```
struct Circle: Equatable {
    var center: CGPoint
    var radius: Double

    init(center: CGPoint, radius: Double) {
        self.center = center
        self.radius = radius
    }
}

func ==(lhs: Circle, rhs: Circle) {
  return lhs.center == rhs.center && lhs.radius == rhs.radius
}
```



Polygon

```
struct Polygon: Equatable {
    var corners: [CGPoint] = []
}

func ==(lhs: Polygon, rhs: Polygon) {
  return lhs.corners == rhs.corners
}
```

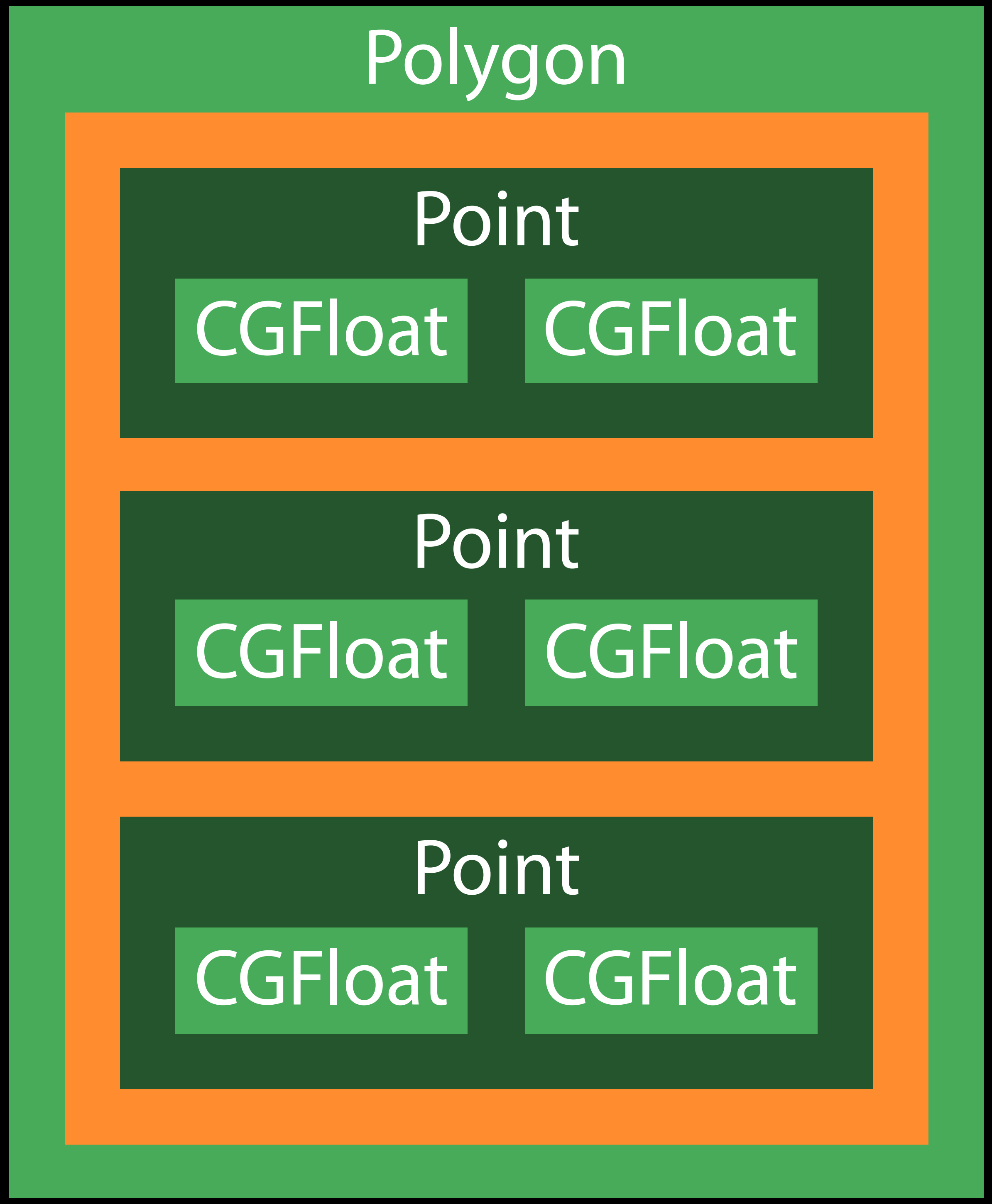


Diagram Contains Circles

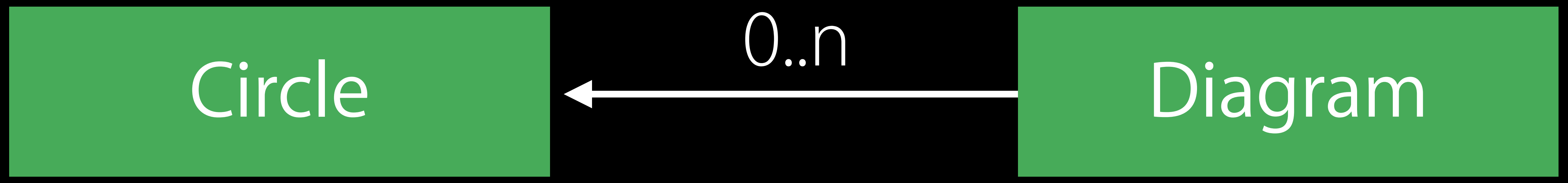
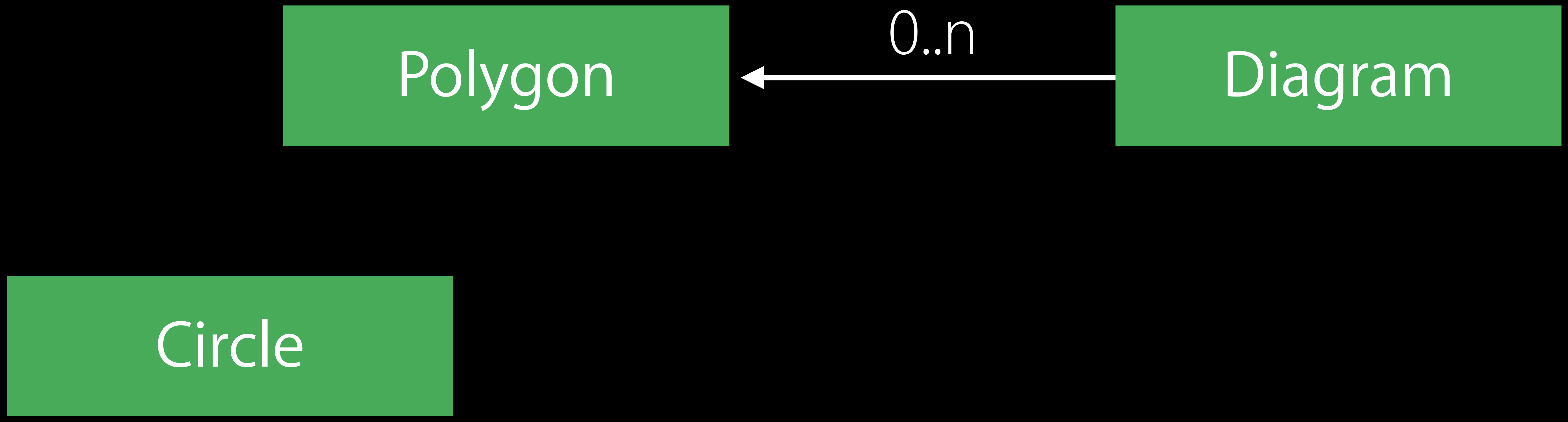


Diagram Contains Polygons



Polygon

0..n

Diagram

Circle

Diagram Contains Polygons

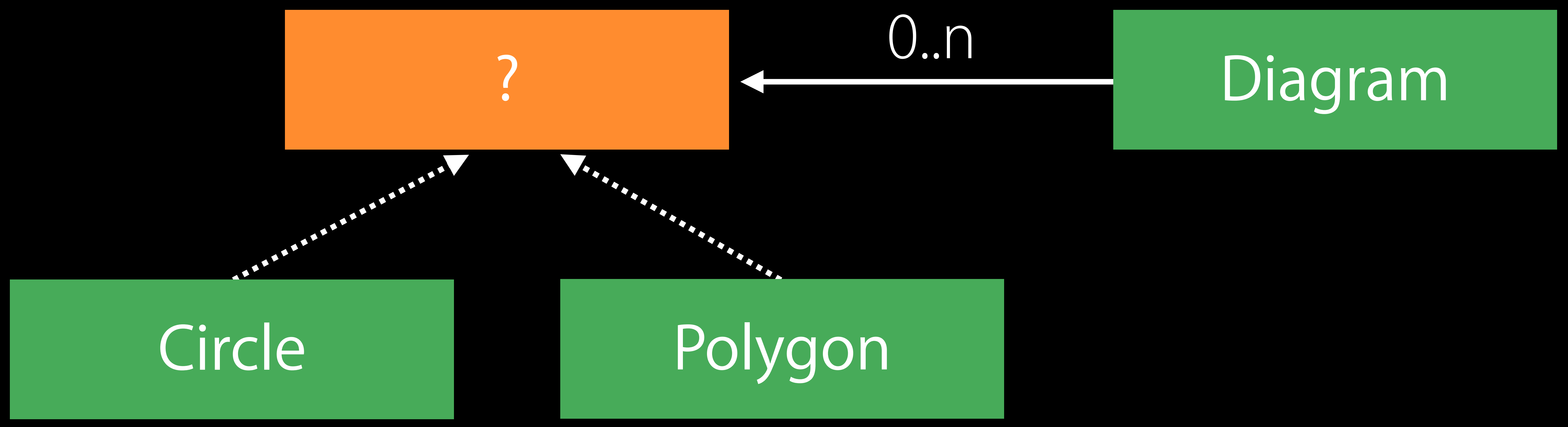


Diagram Contains Polygons

Protocols can abstract over value types

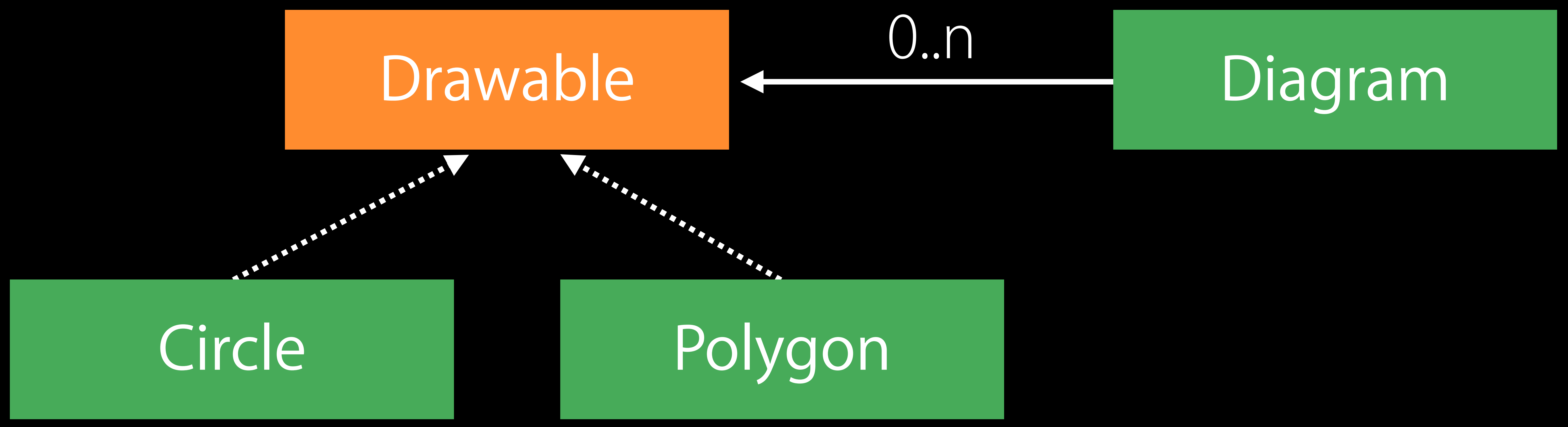
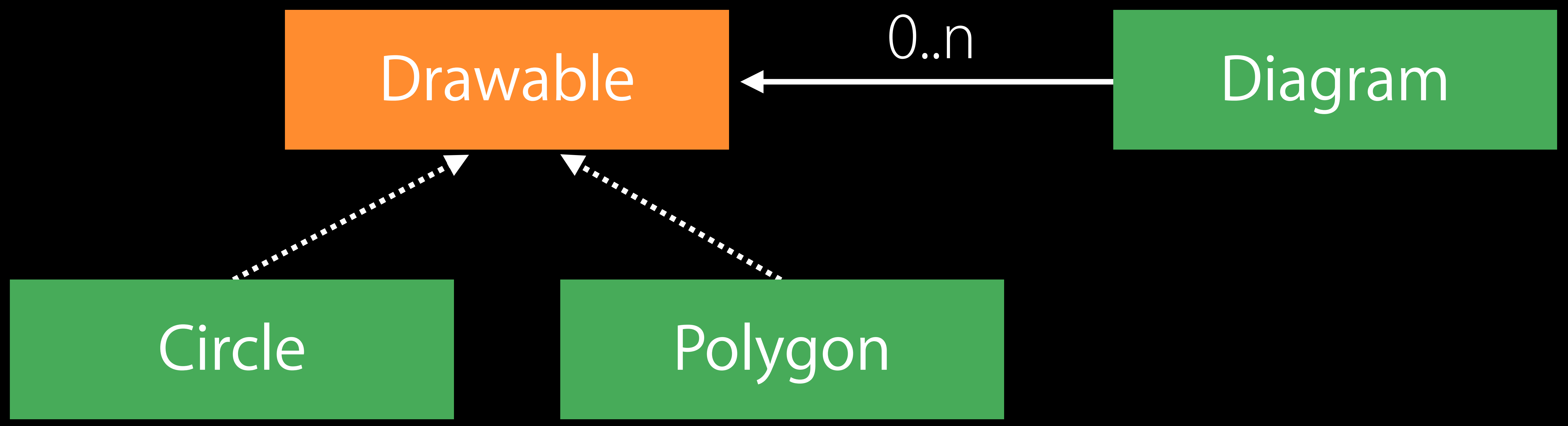


Diagram Contains Polygons

Protocols can abstract over value types



The Drawable Protocol

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

The Drawable Protocol

```
protocol Drawable {
  func draw()
}

extension Polygon: Drawable {
  func draw() {
    let ctx = UIGraphicsGetCurrentContext()
    CGContextMoveToPoint(ctx, corners.last!.x corners.last!.y)
    for point in corners {
      CGContextAddLineToPoint(ctx, point.x, point.y)
    }
    CGContextClosePath(ctx)
    CGContextStrokePath(ctx)
  }
}
```

The Drawable Protocol

```
protocol Drawable {
  func draw()
}

extension Circle: Drawable {
  func draw() {
    let arc = CGPathCreateMutable()
    CGPathAddArc(arc, nil, center.x, center.y, radius, 0, 2 * π, true)
    CGContextAddPath(ctx, arc)
    CGContextStrokePath(ctx)
  }
}
```

Creating the Diagram

```
struct Diagram {
  var items: [Drawable] = []









}
```

Creating the Diagram

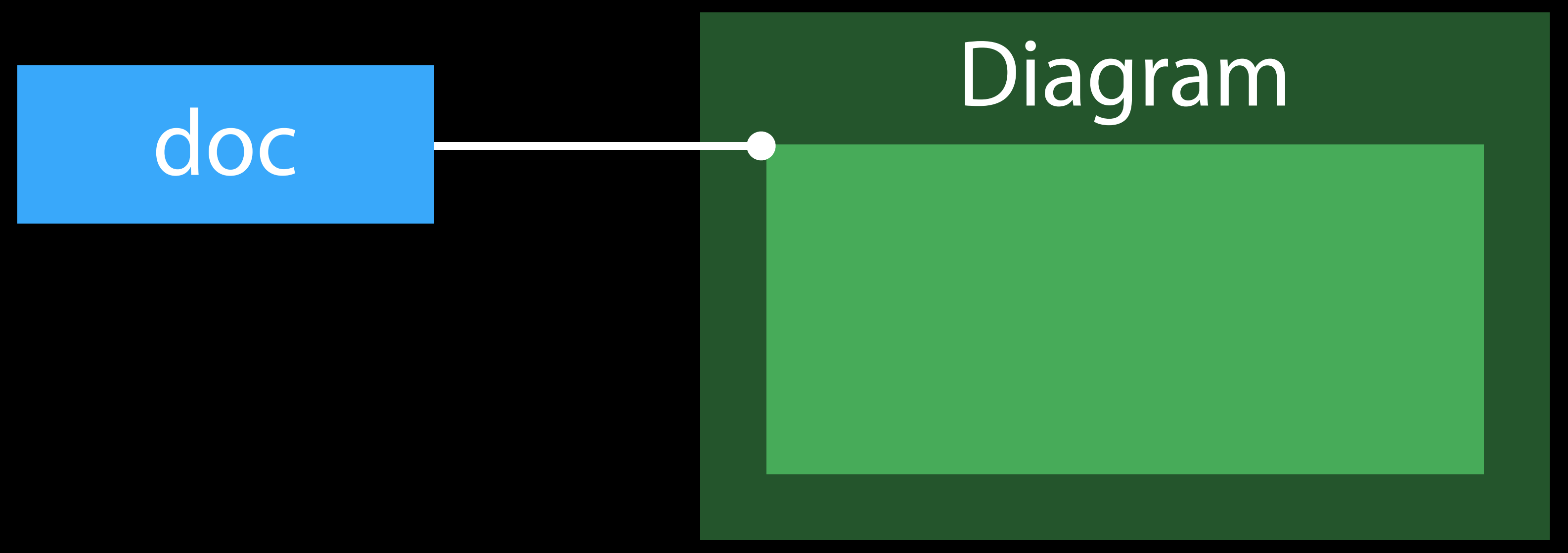
```
struct Diagram {
  var items: [Drawable] = []

  mutating func addItem(item: Drawable) {
    items.append(item)
  }






}
```

Creating the Diagram

```
struct Diagram {
  var items: [Drawable] = []

  mutating func addItem(item: Drawable) {
    items.append(item)
  }

  func draw() {
    for item in items {
      item.draw()
    }
  }
}
```

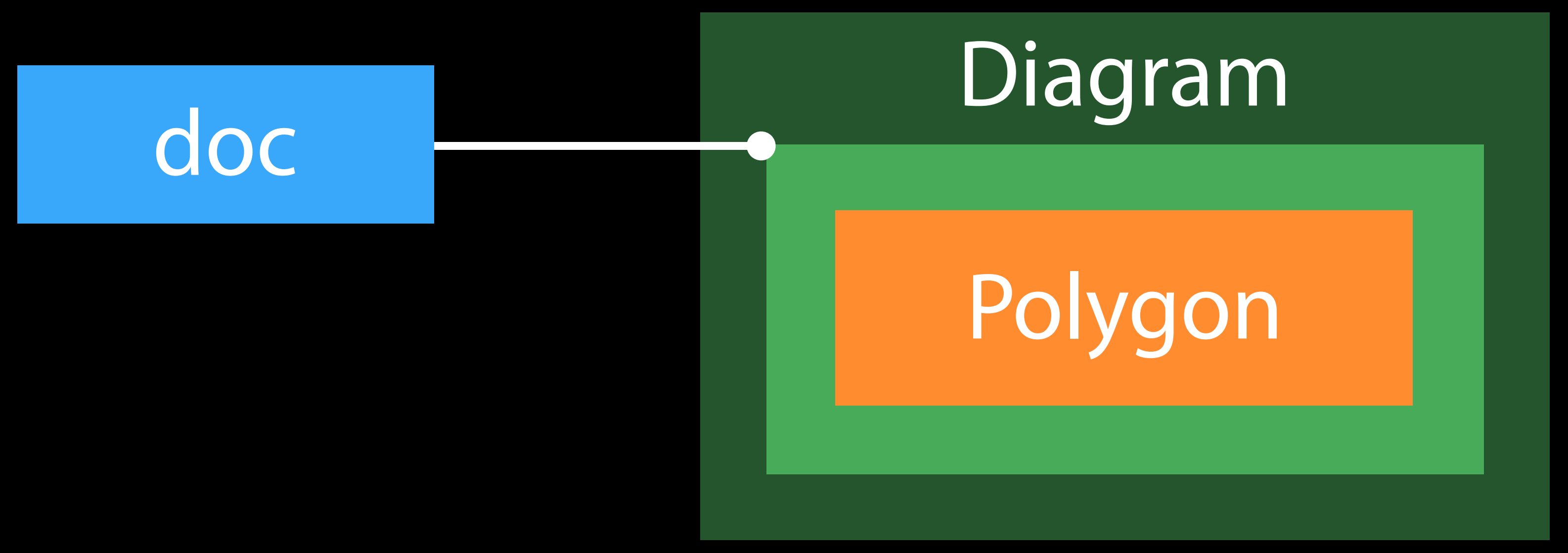
Adding Items

```
var doc = Diagram()
```



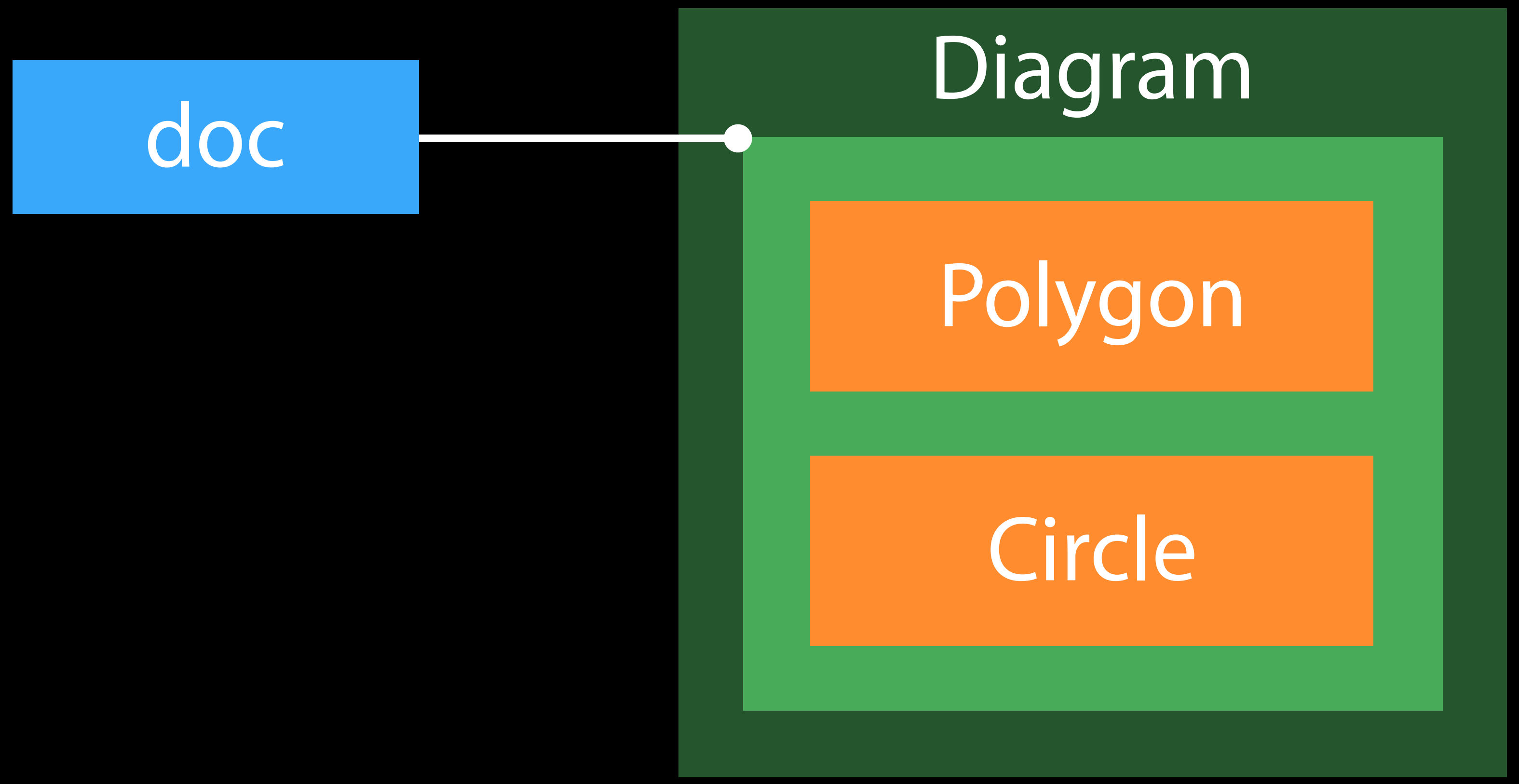
Adding Items

```
var doc = Diagram()
doc.addItem(Polygon())
```



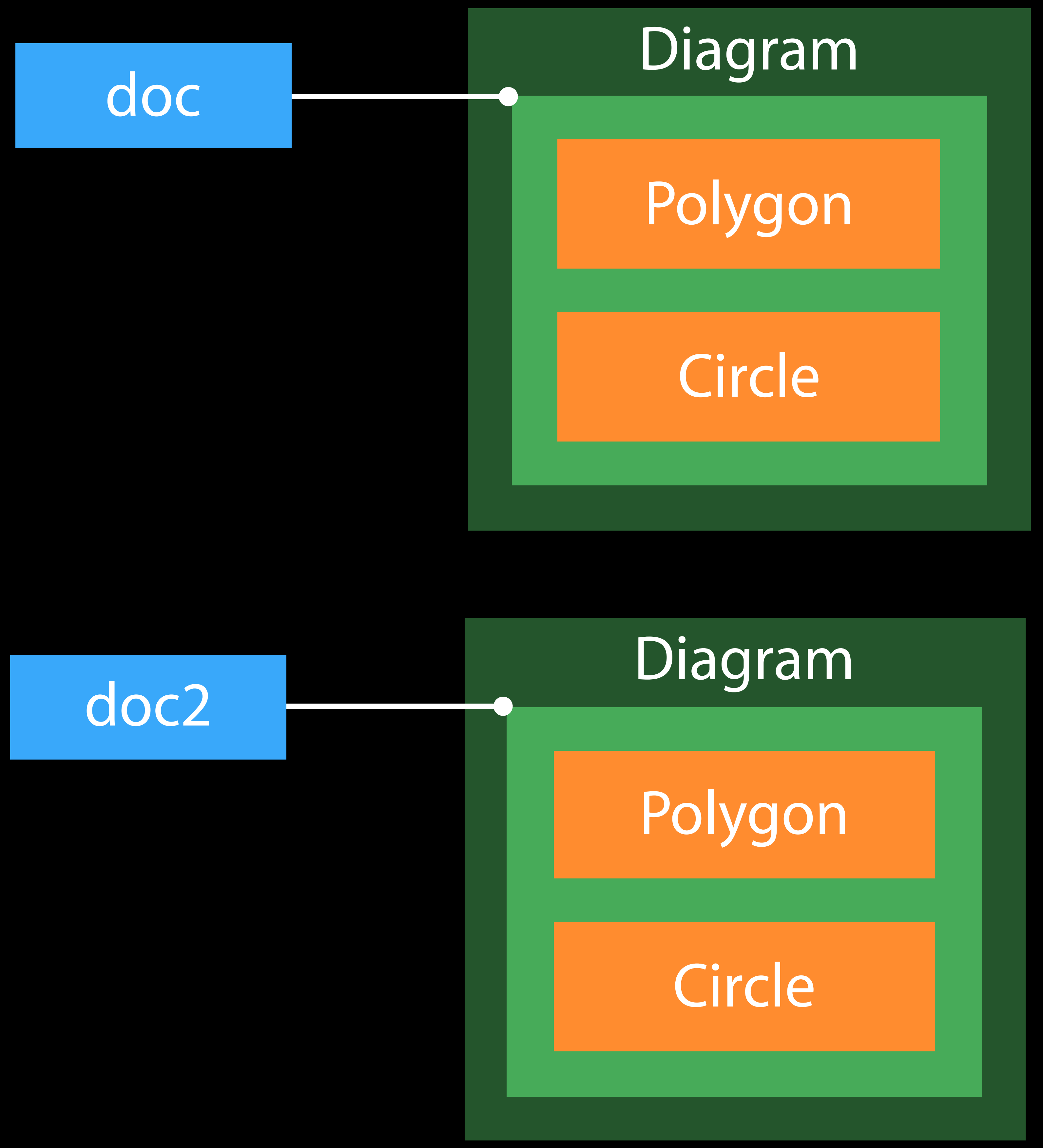
Adding Items

```
var doc = Diagram()
doc.addItem(Polygon())
doc.addItem(Circle())
```



Copied on Assignment

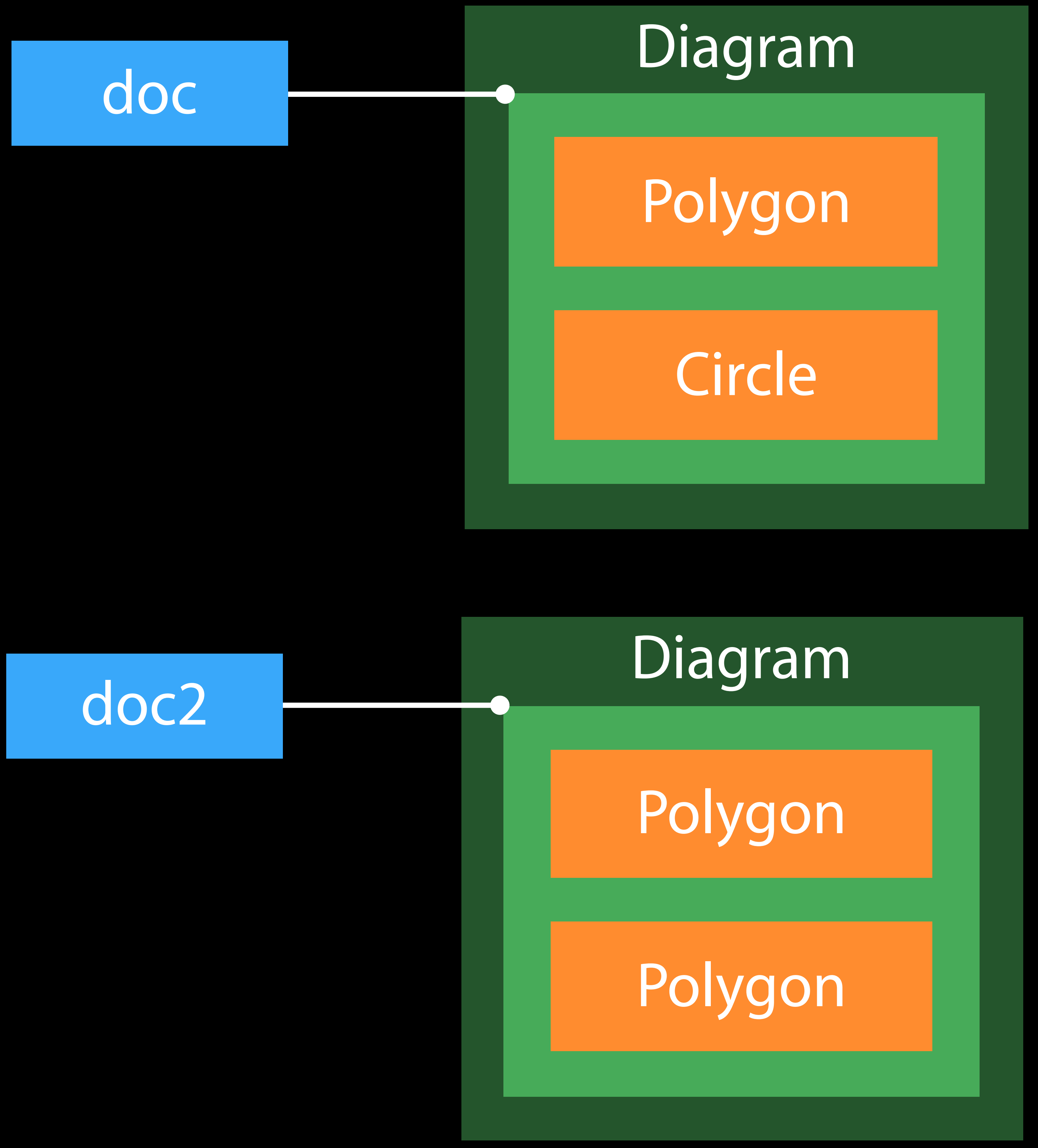
```
var doc = Diagram()
doc.addItem(Polygon())
doc.addItem(Circle())
var doc2 = doc
```



Copied on Assignment

Heterogeneous arrays have value semantics, too!

```
var doc = Diagram()
doc.addItem(Polygon())
doc.addItem(Circle())
var doc2 = doc
doc2.items[1] = Polygon(corners: points)
```



Making Diagram Equatable

```
extension Diagram: Equatable { }
func ==(lhs: Diagram, rhs: Diagram) {
  return lhs.items == rhs.items
}
```

Making Diagram Equatable

```
extension Diagram: Equatable { }

func ==(lhs: Diagram, rhs: Diagram) {
  return lhs.items == rhs.items
}
```

```
error: binary operator ‘==‘ cannot be
applied to two [Drawable] operands
```

Making Diagram Equatable

```
extension Diagram: Equatable { }

func ==(lhs: Diagram, rhs: Diagram) {
  return lhs.items == rhs.items
}
```

```
error: binary operator ‘==‘ cannot be
applied to two [Drawable] operands
```

If It Quacks Like a Duck…

```
protocol Drawable {
  func draw()
}

struct Diagram {
  var items: [Drawable] = []

  func draw() { … }
}
```

If It Quacks Like a Duck…

```
protocol Drawable {
  func draw()
}

struct Diagram: Drawable {
  var items: [Drawable] = []

  func draw() { … }
}
```

Diagram as a Drawable

```
var doc = Diagram()
doc.addItem(Polygon())
doc.addItem(Circle())
```

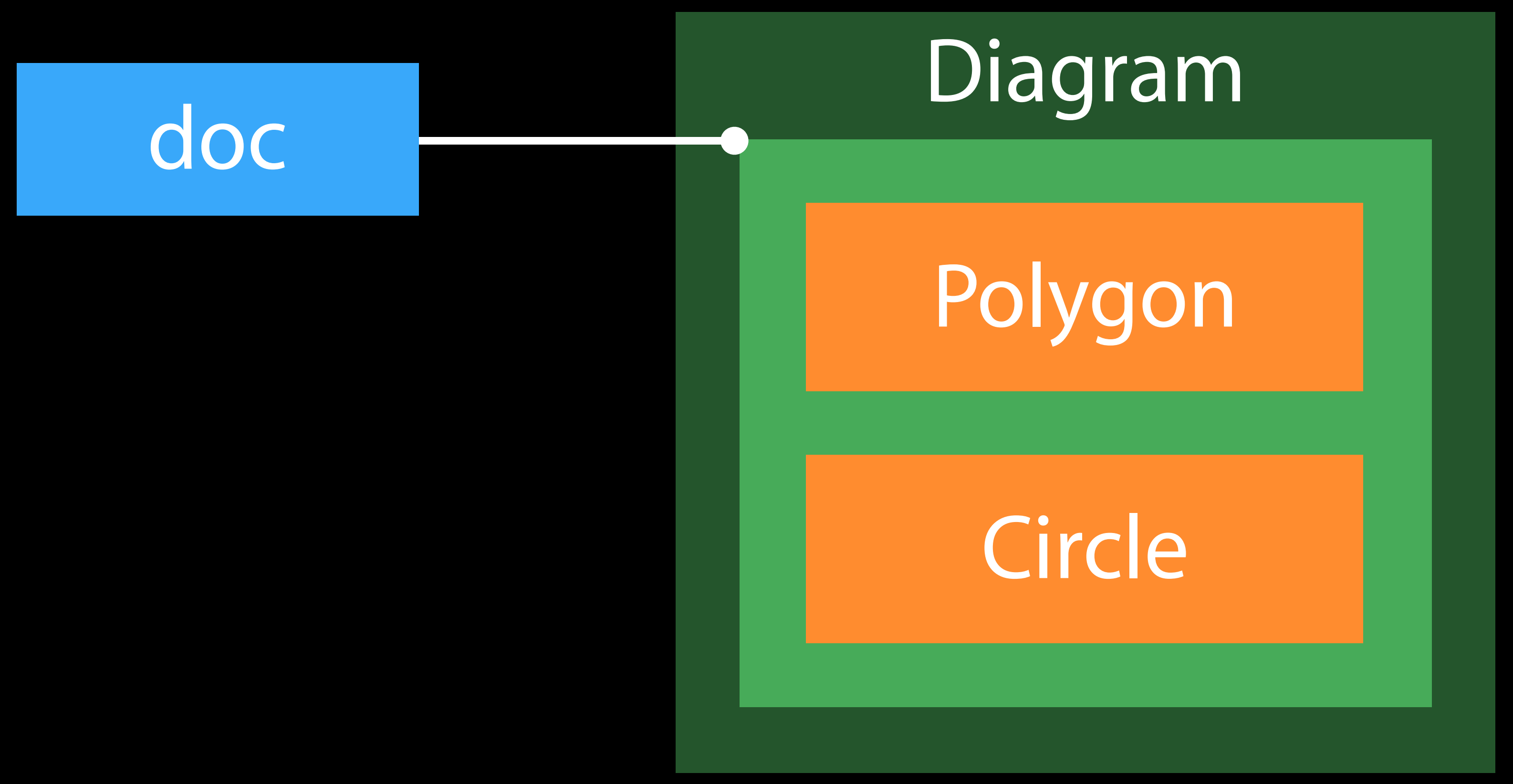


Diagram as a Drawable

```
var doc = Diagram()
doc.addItem(Polygon())
doc.addItem(Circle())
doc.addItem(Diagram())
```

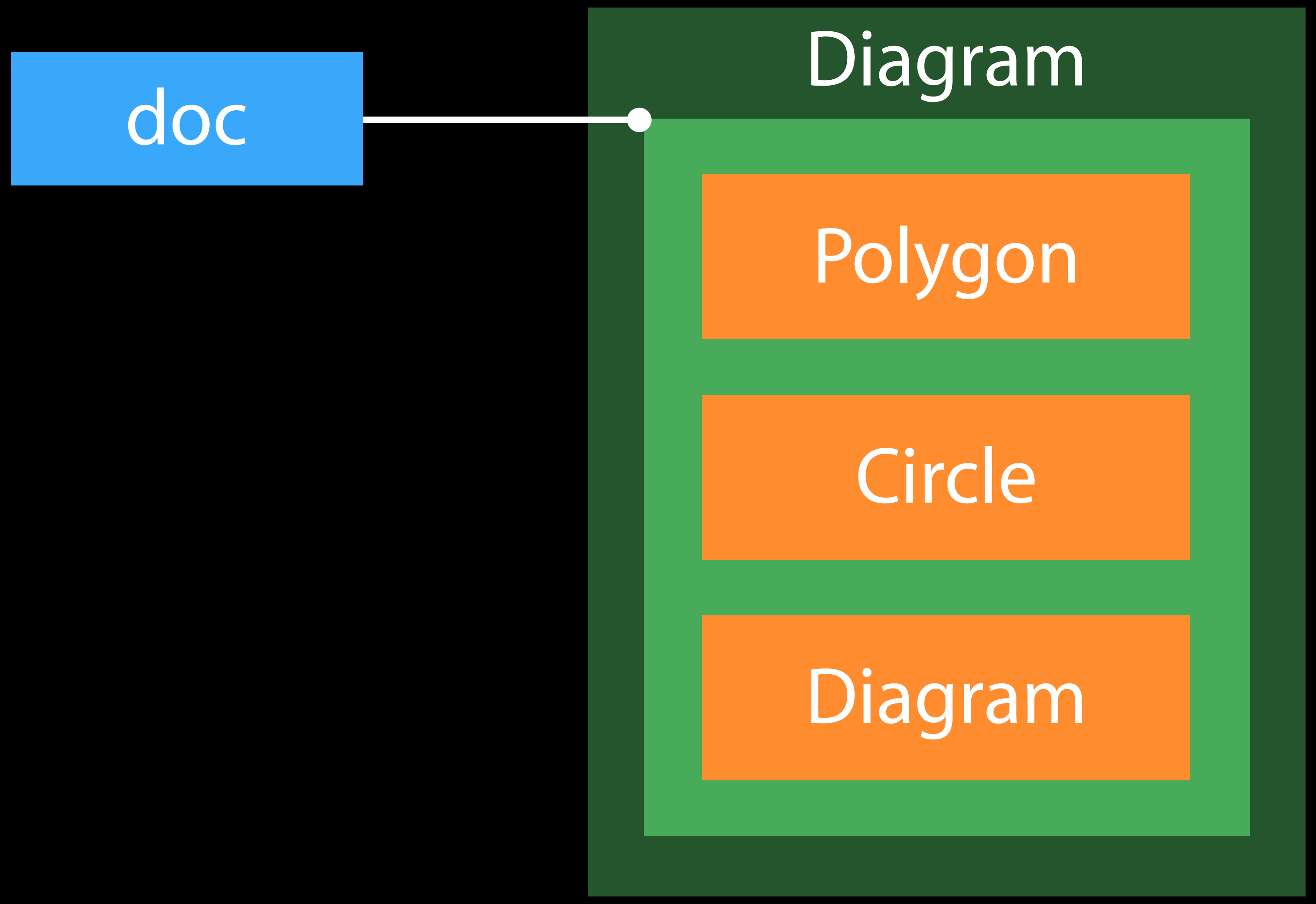


Diagram as a Drawable

```
var doc = Diagram()
doc.addItem(Polygon())
doc.addItem(Circle())
doc.addItem(doc)
```

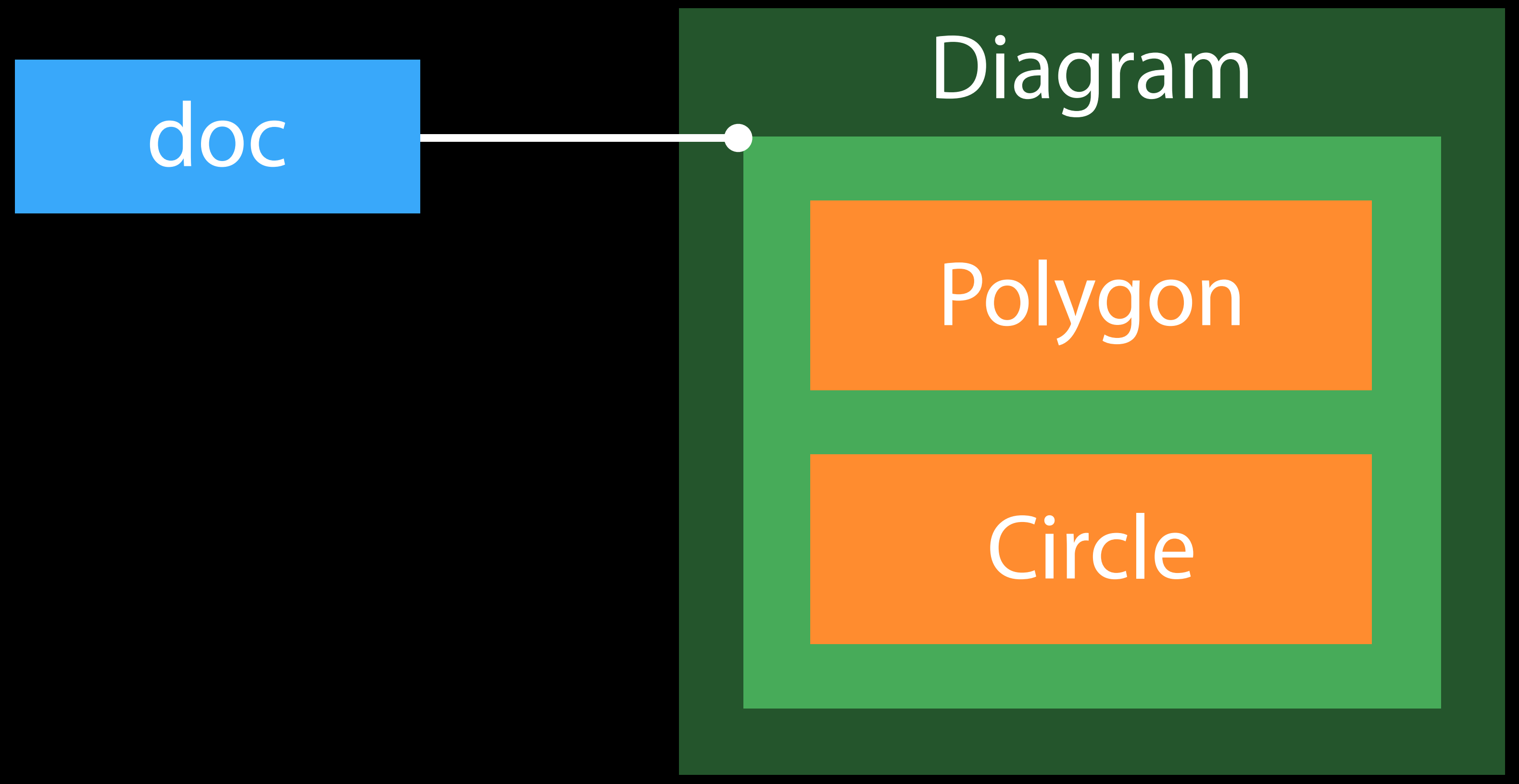


Diagram as a Drawable

```
var doc = Diagram()
doc.addItem(Polygon())
doc.addItem(Circle())
doc.addItem(doc)
```

```
func draw() {
  for item in items {
    item.draw()
  }
}
```

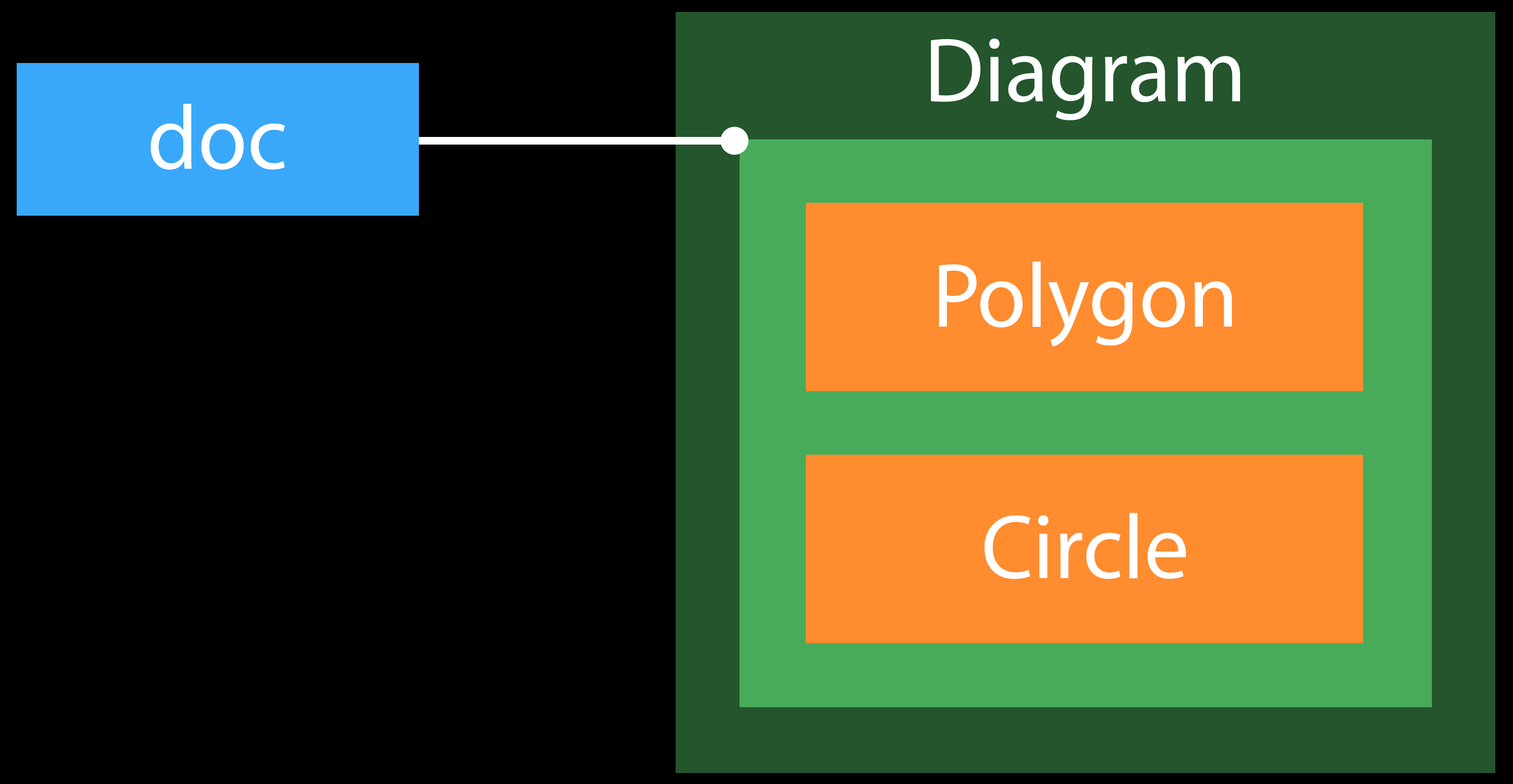
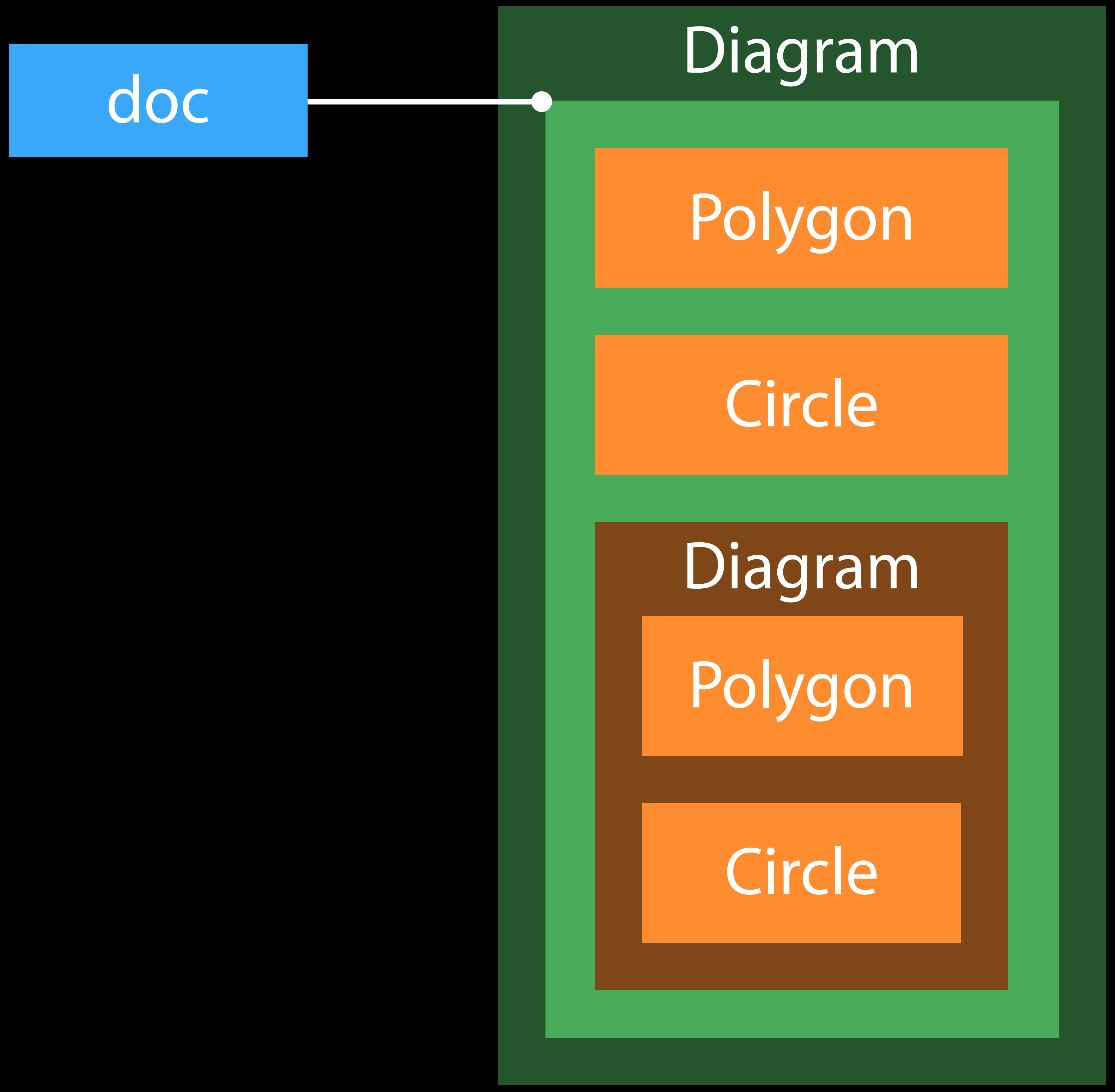


Diagram as a Drawable

```
var doc = Diagram()
doc.addItem(Polygon())
doc.addItem(Circle())
doc.addItem(doc)
```

```
func draw() {
  for item in items {
    item.draw()
  }
}
```



Mixing Value Types and Reference Types

Reference Types Often Contain Value Types

Value types generally used for “primitive” data of objects

```
class Button : Control {
  var label: String
  var enabled: Bool
  // …
}
```

A Value Type Can Contain a Reference

Copies of the value type will share the reference

```
struct ButtonWrapper {
  var button: Button
}
```

A Value Type Can Contain a Reference

Copies of the value type will share the reference

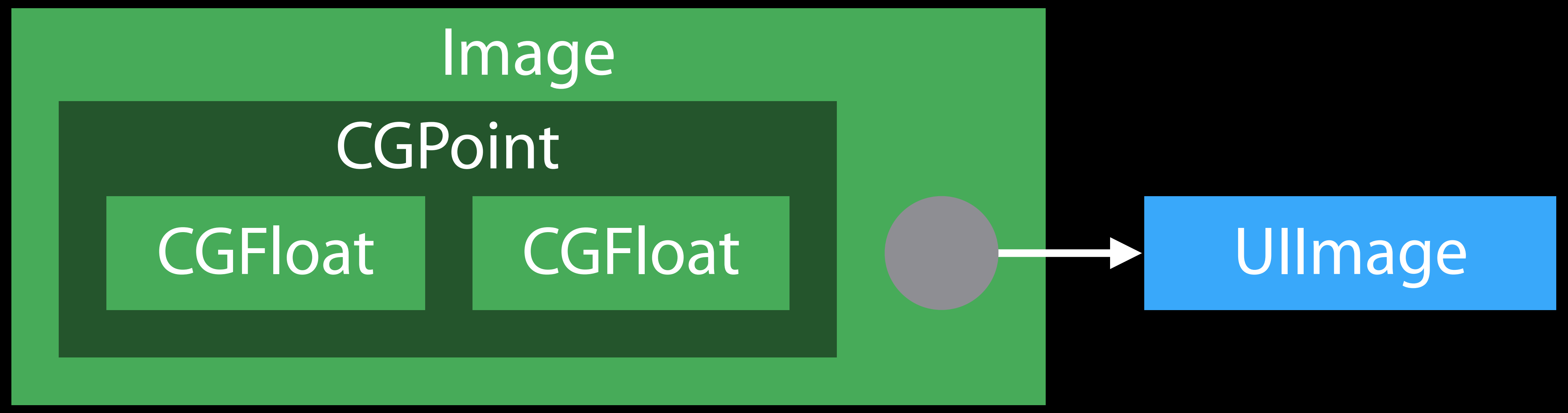
```
struct ButtonWrapper {
  var button: Button
}
```

Maintaining value semantics requires special considerations

- How do we cope with mutation of the referenced object?
- How does the reference identity affect equality?

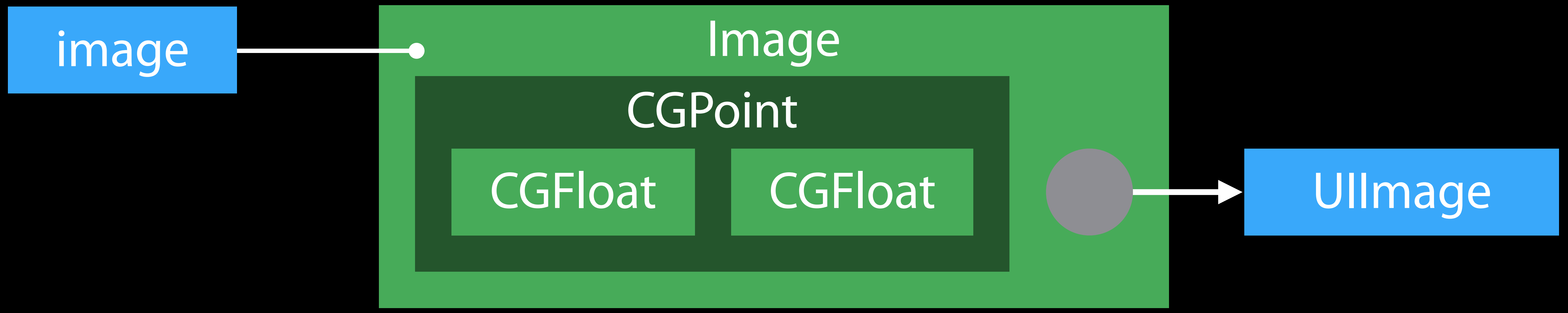
Immutable References

```
struct Image : Drawable {
  var topLeft: CGPoint
  var image: UIImage
}
```



Immutable References

```
struct Image : Drawable {
  var topLeft: CGPoint
  var image: UIImage
}
```

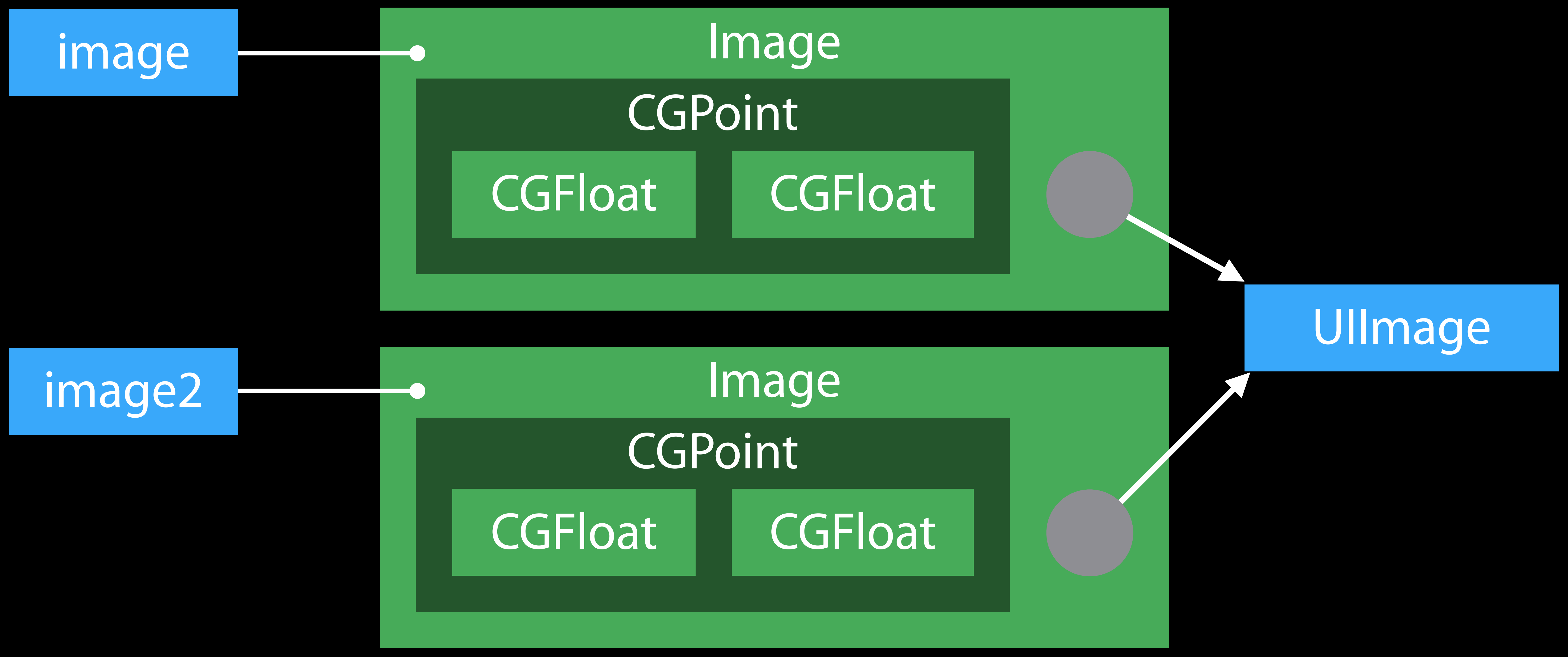


```
var image = Image(topLeft: CGPoint(x: 0, y: 0),
                  image: UIImage(imageNamed:"San Francisco")!)
```

Immutable References Are Okay!

Mutation of the referenced object does not occur

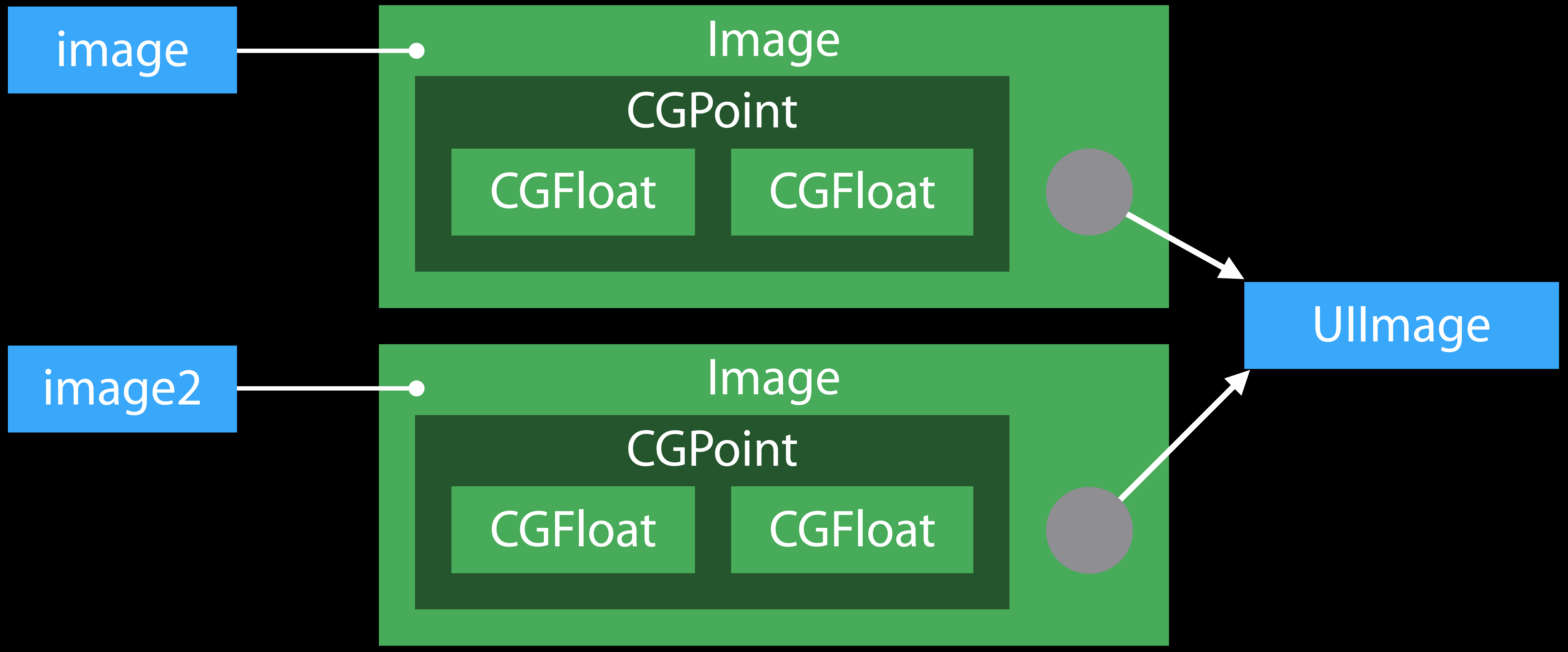
```
struct Image : Drawable {
  var topLeft: CGPoint
  var image: UIImage
}
```



```
var image = Image(topLeft: CGPoint(x: 0, y: 0),
                  image: UIImage(imageNamed:"San Francisco”)!)
var image2 = image
```

Immutable References and Equatable

```
struct Image : Drawable {
  var topLeft: CGPoint
  var image: UIImage
}
```

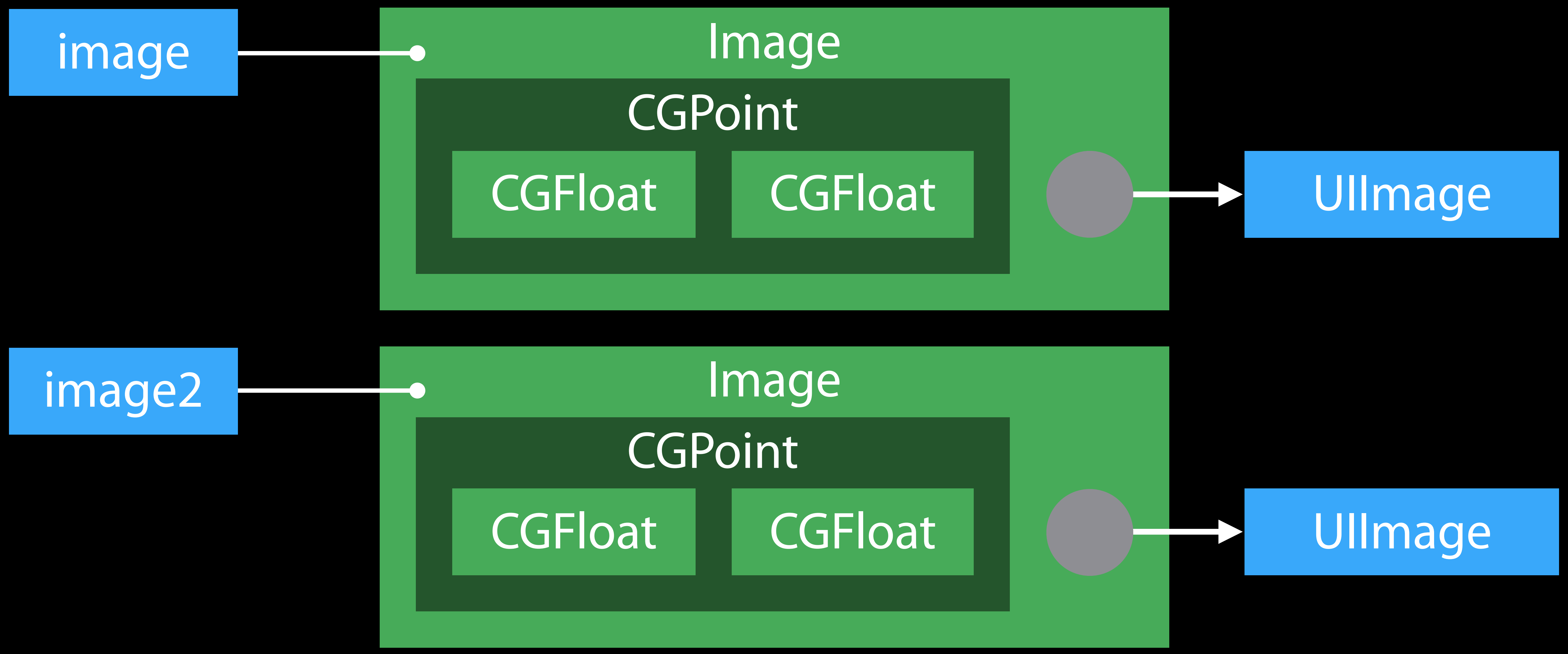


```
extension Image : Equatable { }
func ==(lhs: Image, rhs: Image) -> Bool {
  return lhs.topLeft == rhs.topLeft && lhs.image === rhs.image
}
```

Immutable References and Equatable

Reference identity is not enough

```
struct Image : Drawable {
  var topLeft: CGPoint
  var image: UIImage
}
```

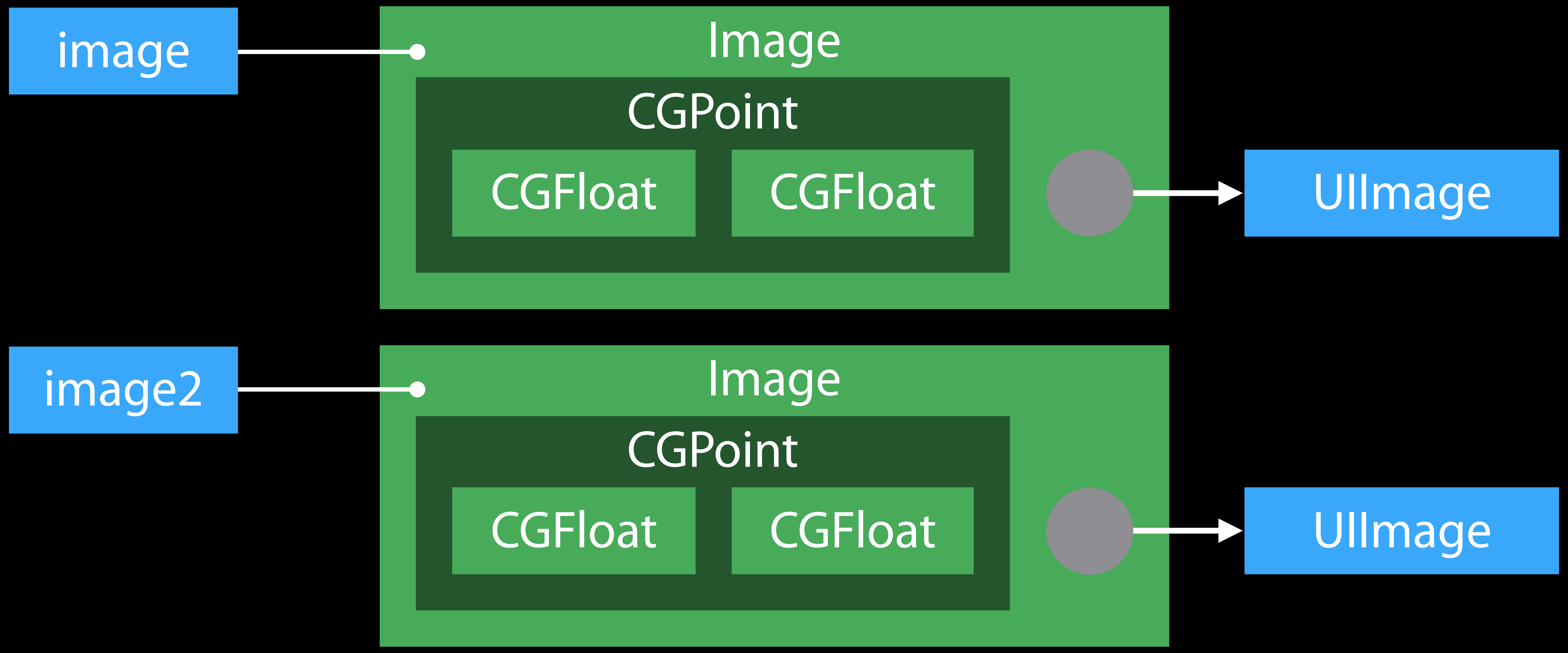


```
extension Image : Equatable { }
func ==(lhs: Image, rhs: Image) -> Bool {
  return lhs.topLeft == rhs.topLeft && lhs.image === rhs.image
}
```

Immutable References and Equatable

Use deep equality comparisons

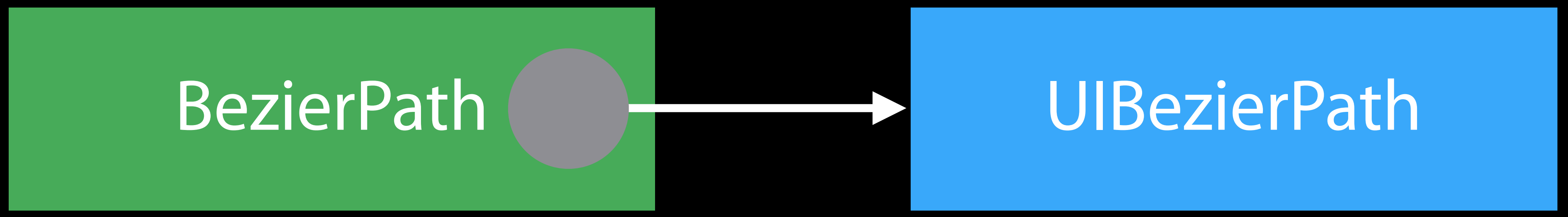
```
struct Image : Drawable {
  var topLeft: CGPoint
  var image: UIImage
}
```



```
extension Image : Equatable { }
func ==(lhs: Image, rhs: Image) -> Bool {
  return lhs.topLeft == rhs.topLeft && lhs.image.isEqual(rhs.image)
}
```

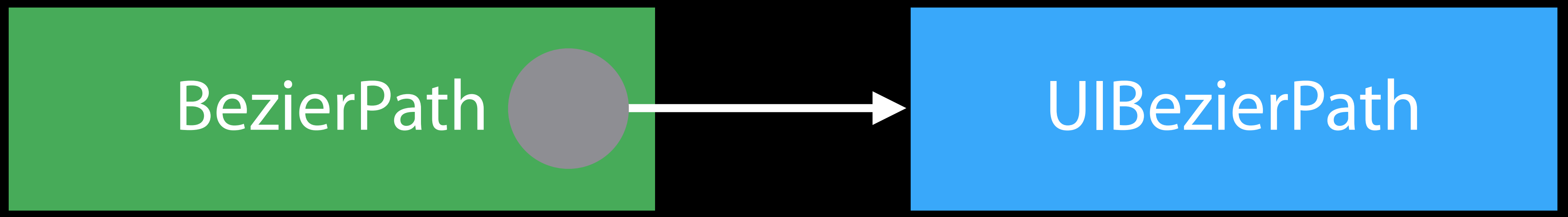
References to Mutable Objects

```
struct BezierPath: Drawable {
  var path = UIBezierPath()

  var isEmpty: Bool {
    return path.empty
  }

  func addLineToPoint(point: CGPoint) {
    path.addLineToPoint(point)
  }
}
```



References to Mutable Objects

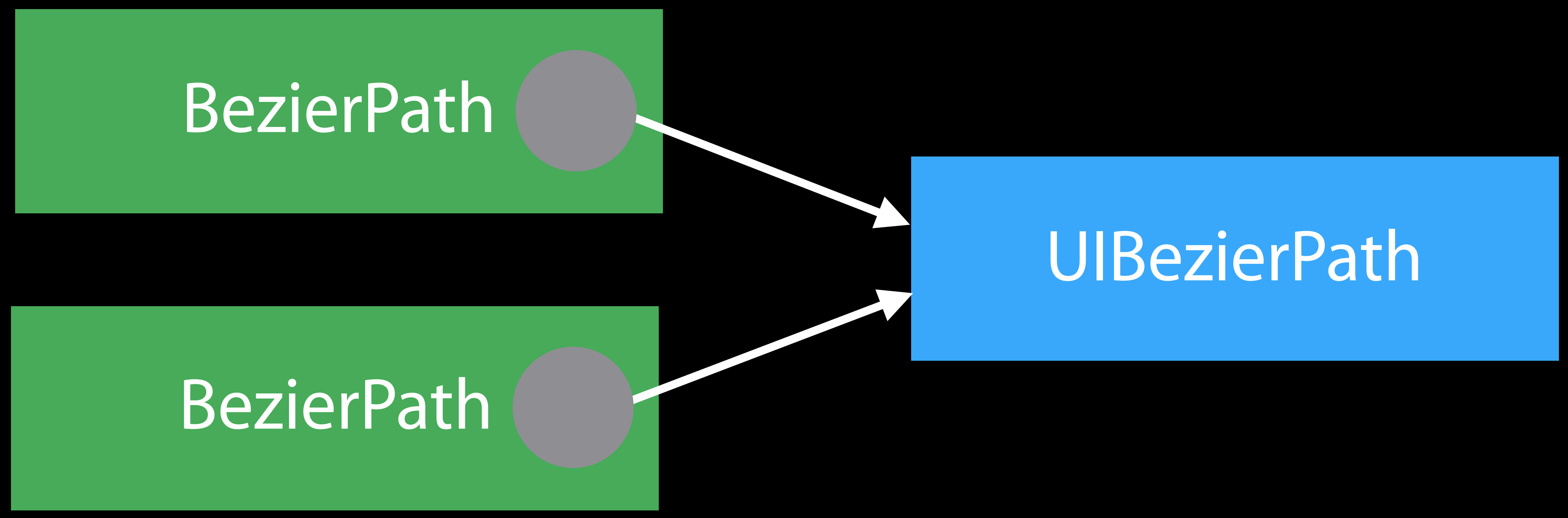
```
struct BezierPath: Drawable {
  var path = UIBezierPath()

  var isEmpty: Bool {
    return path.empty
  }

  func addLineToPoint(point: CGPoint) {
    path.addLineToPoint(point)
  }
}
```



References to Mutable Objects

Unexpected mutation

```
struct BezierPath: Drawable {
  var path = UIBezierPath()

  var isEmpty: Bool {
    return path.empty
  }

  func addLineToPoint(point: CGPoint) {
    path.addLineToPoint(point)
  }
}
```



Copy-on-Write

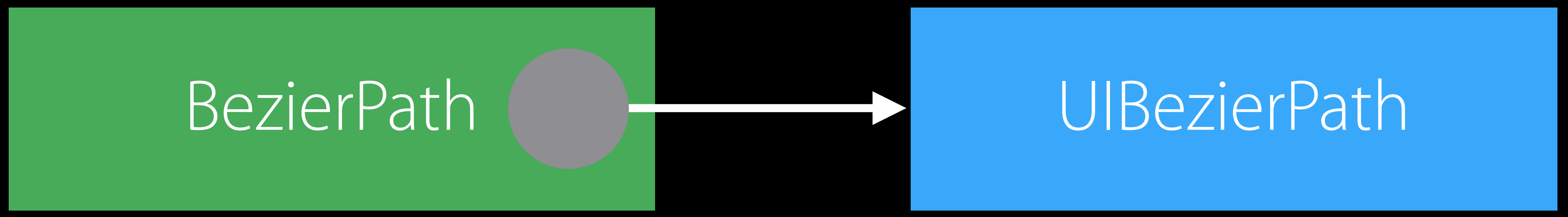
Unrestricted mutation of referenced objects breaks value semantics

Separate non-mutating operations from mutating ones

- Non-mutating operations are always safe
- Mutating operations must first copy

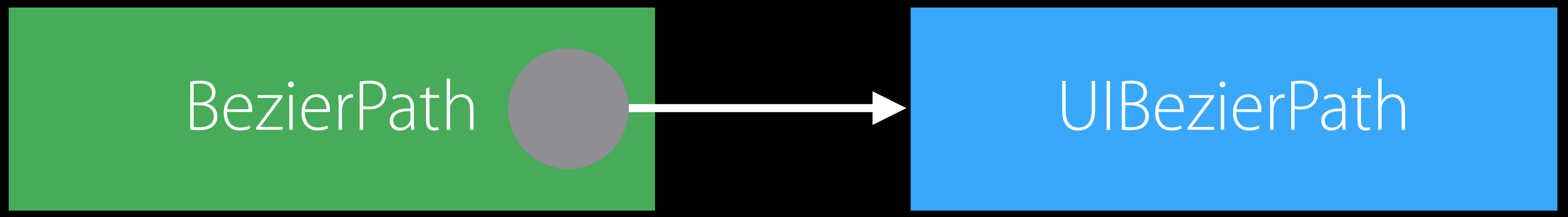
Copy-on-Write in Action

```
struct BezierPath: Drawable {
  private var _path = UIBezierPath()

  var pathForReading: UIBezierPath {
    return _path
  }






}
```



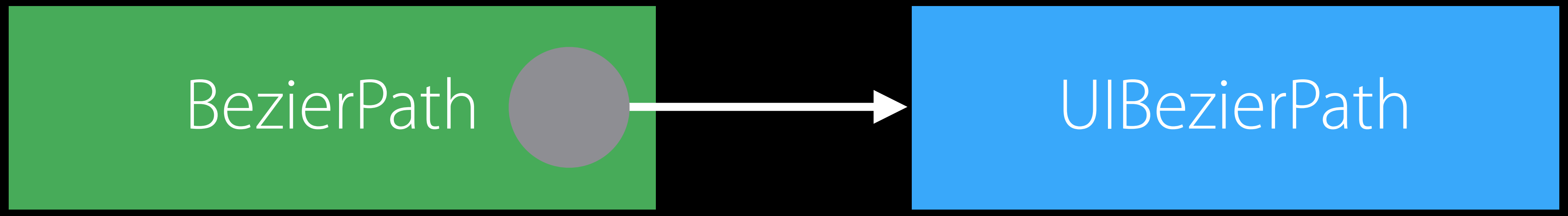
Copy-on-Write in Action

```
struct BezierPath: Drawable {
  private var _path = UIBezierPath()

  var pathForReading: UIBezierPath {
    return _path
  }

  var pathForWriting: UIBezierPath {
    mutating get {
      _path = _path.copy() as! UIBezierPath
      return _path
    }
  }
}
```



Copy-on-Write in Action

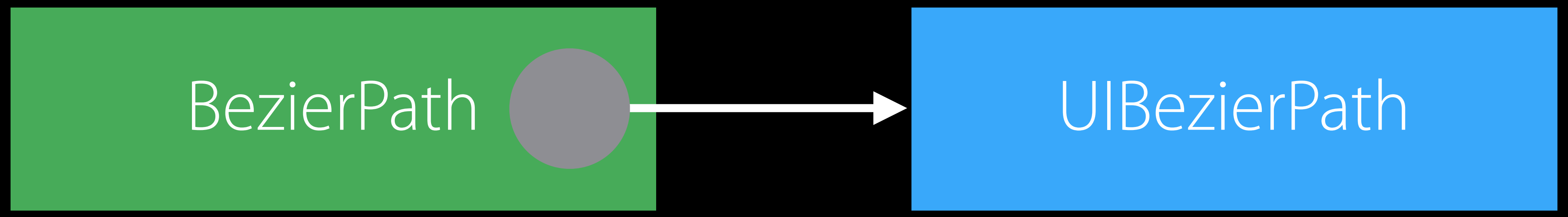
```
extension BezierPath {
  var isEmpty: Bool {
    return pathForReading.empty
  }

  func addLineToPoint(point: CGPoint) {
    pathForWriting.addLineToPoint(point)
  }
}
```



Copy-on-Write in Action

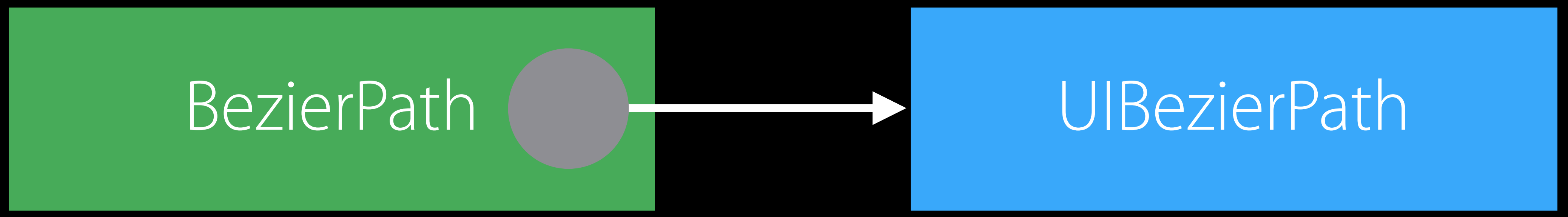
```
extension BezierPath {
  var isEmpty: Bool {
    return pathForReading.empty
  }

  func addLineToPoint(point: CGPoint) {
    pathForWriting.addLineToPoint(point)
  }
}
```

```
error: cannot read ‘pathForWriting’ because ‘self’ is not mutable
```



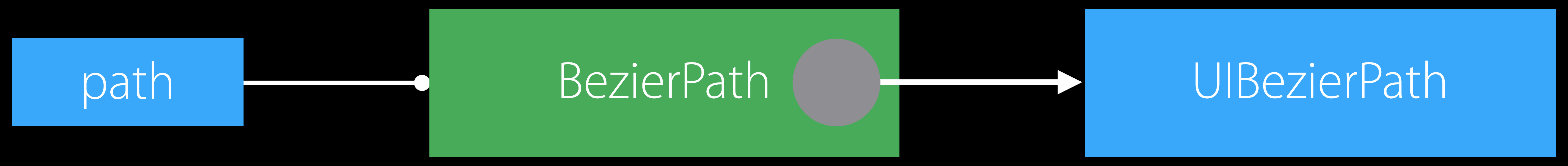
Copy-on-Write in Action

```
extension BezierPath {
  var isEmpty: Bool {
    return pathForReading.empty
  }

  mutating func addLineToPoint(point: CGPoint) {
    pathForWriting.addLineToPoint(point)
  }
}
```



Bezier Path

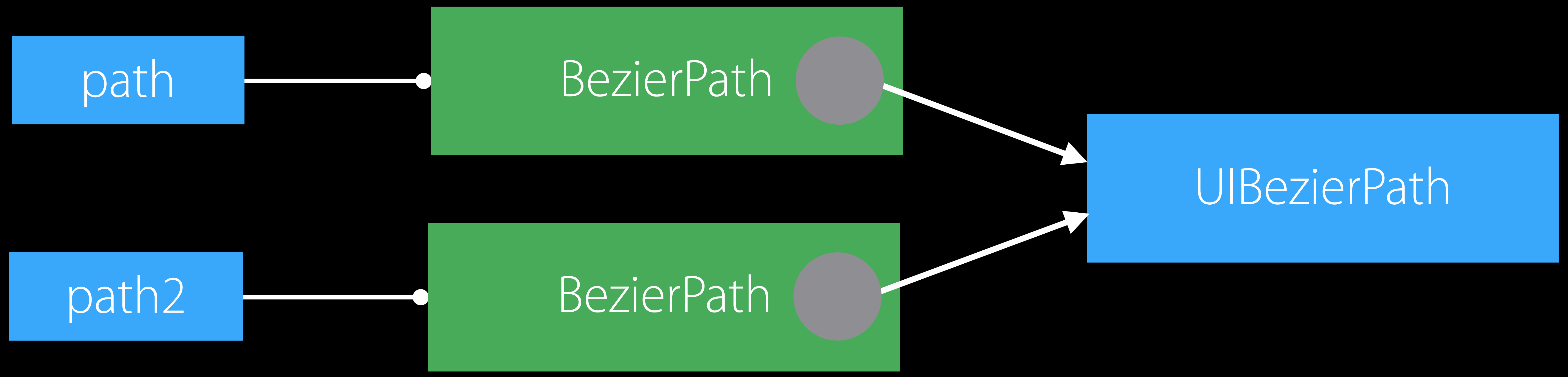
Copy-on-write



```
var path = BezierPath()
```

Bezier Path

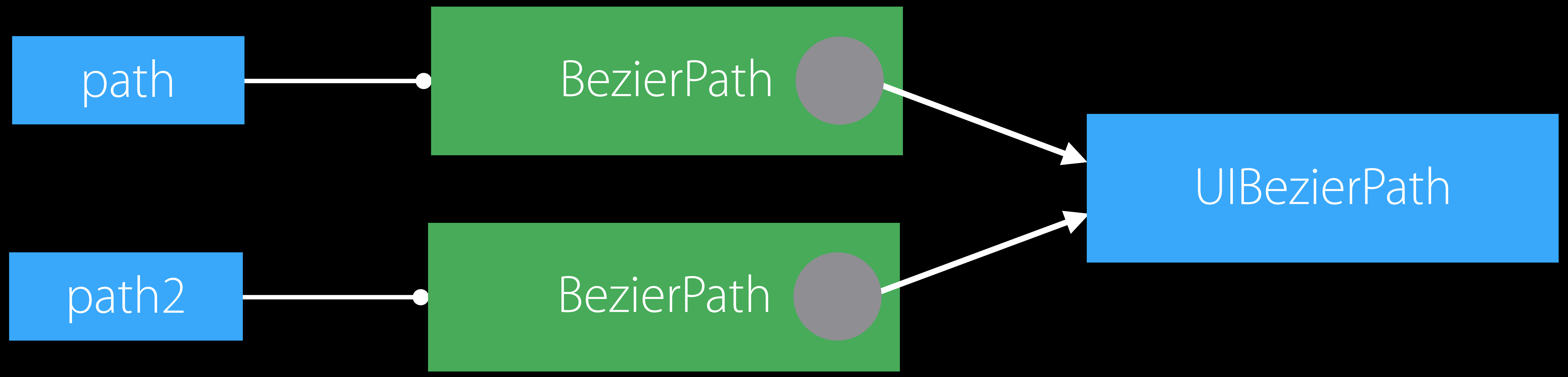
Copy-on-write



```
var path = BezierPath()
var path2 = path
```

Bezier Path

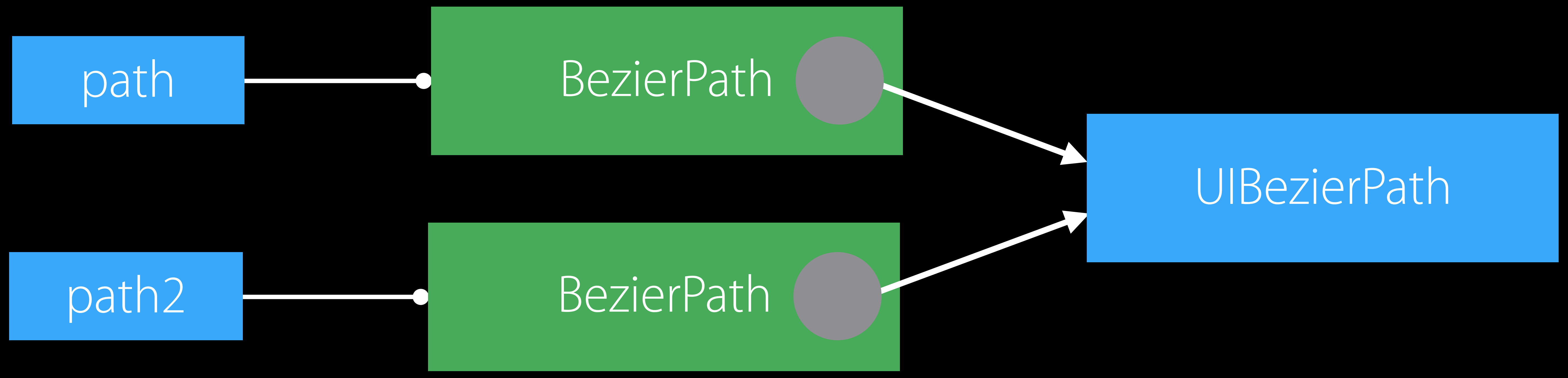
Copy-on-write



```
var path = BezierPath()
var path2 = path
if path.empty { print("Path is empty") }
```

Bezier Path

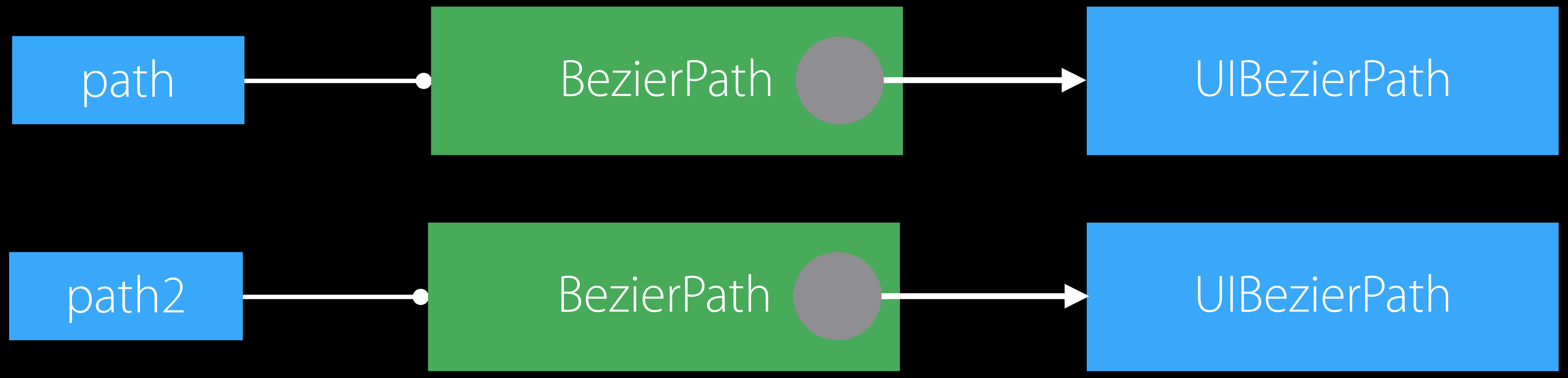
Copy-on-write



```
var path = BezierPath()
var path2 = path
if path.empty { print("Path is empty") }
path.addLineToPoint(CGPoint(x: 10, y: 20))
```

Bezier Path

Copy-on-write



```
var path = BezierPath()
var path2 = path
if path.empty { print("Path is empty") }
path.addLineToPoint(CGPoint(x: 10, y: 20))
```

Forming a Path from a Polygon

```
extension Polygon {
  var path: BezierPath {
    var result = BezierPath()
    result.moveToPoint(corners.last!)
    for point in corners {
      result.addLineToPoint(point)
    }
    return result
  }
}
```

Forming a Path from a Polygon

Copies every time through the loop!

```
extension Polygon {
  var path: BezierPath {
    var result = BezierPath()
    result.moveToPoint(corners.last!)
    for point in corners {
      result.addLineToPoint(point)
    }
    return result
  }
}
```

Forming a Path from a Polygon

Use the mutable reference type (carefully)

```
extension Polygon {
  var path: BezierPath {
    var result = UIBezierPath()
    result.moveToPoint(corners.last!)
    for point in corners {
      result.addLineToPoint(point)
    }
    return BezierPath(path: result)
  }
}
```

Uniquely Referenced Swift Objects

```
struct MyWrapper {
  var _object: SomeSwiftObject
  var objectForWriting: SomeSwiftObject {
    mutating get {

      _object = _object.copy()

      return _object
    }
  }
}
```

Uniquely Referenced Swift Objects

```
struct MyWrapper {
  var _object: SomeSwiftObject
  var objectForWriting: SomeSwiftObject {
    mutating get {
      if !isUniquelyReferencedNonObjC(&_object)) {
        _object = _object.copy()
      }
      return _object
    }
  }
}
```

Uniquely Referenced Swift Objects

```
struct MyWrapper {
  var _object: SomeSwiftObject
  var objectForWriting: SomeSwiftObject {
    mutating get {
     if !isUniquelyReferencedNonObjC(&_object)) {
        _object = _object.copy()
     }
     return _object
    }
  }
}
```

The standard library value types uses this throughout

Mixing Value Types and Reference Types

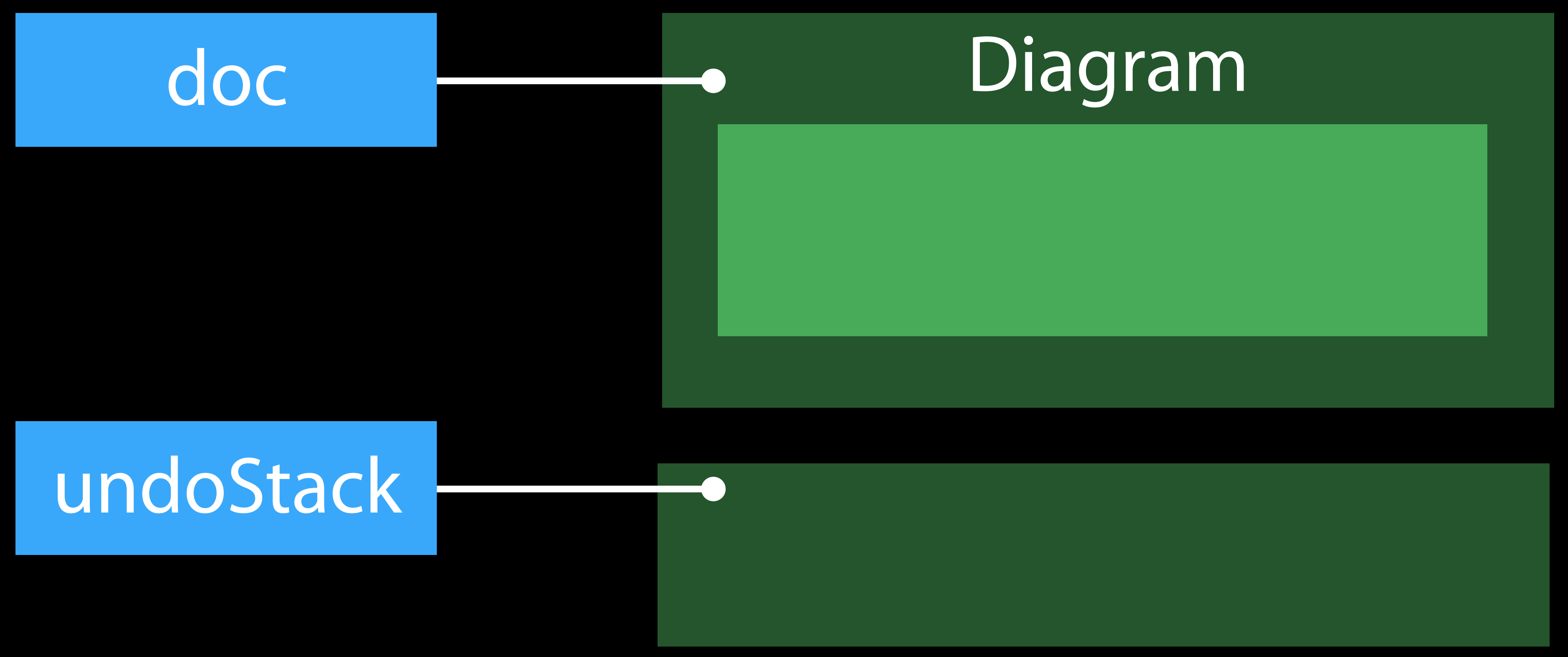
Maintaining value semantics requires special considerations

Copy-on-write enables efficient value semantics when wrapping Swift reference types

Implementing Undo with Value Types

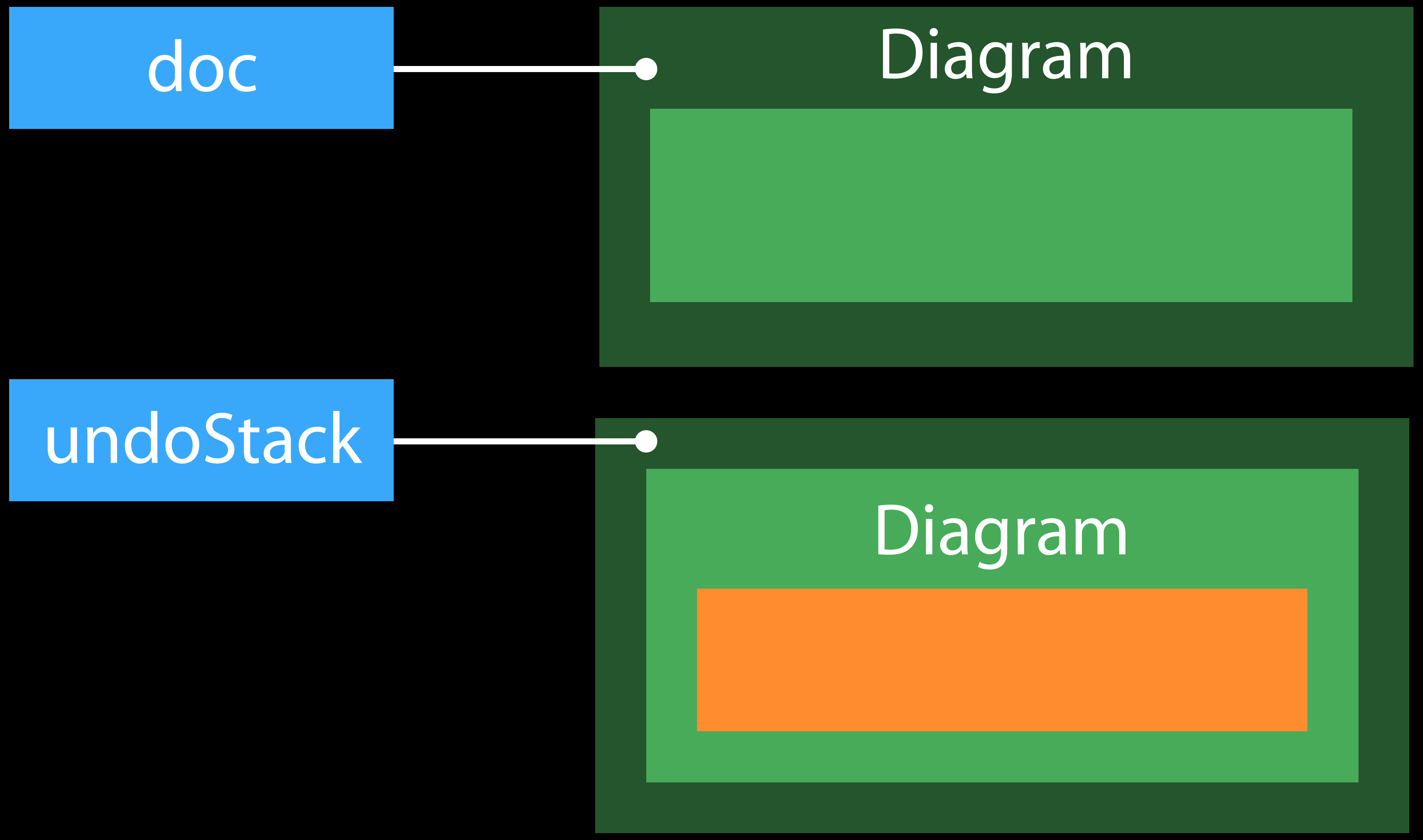
Undo

```
var doc = Diagram()
var undoStack: [Diagram] = []
```



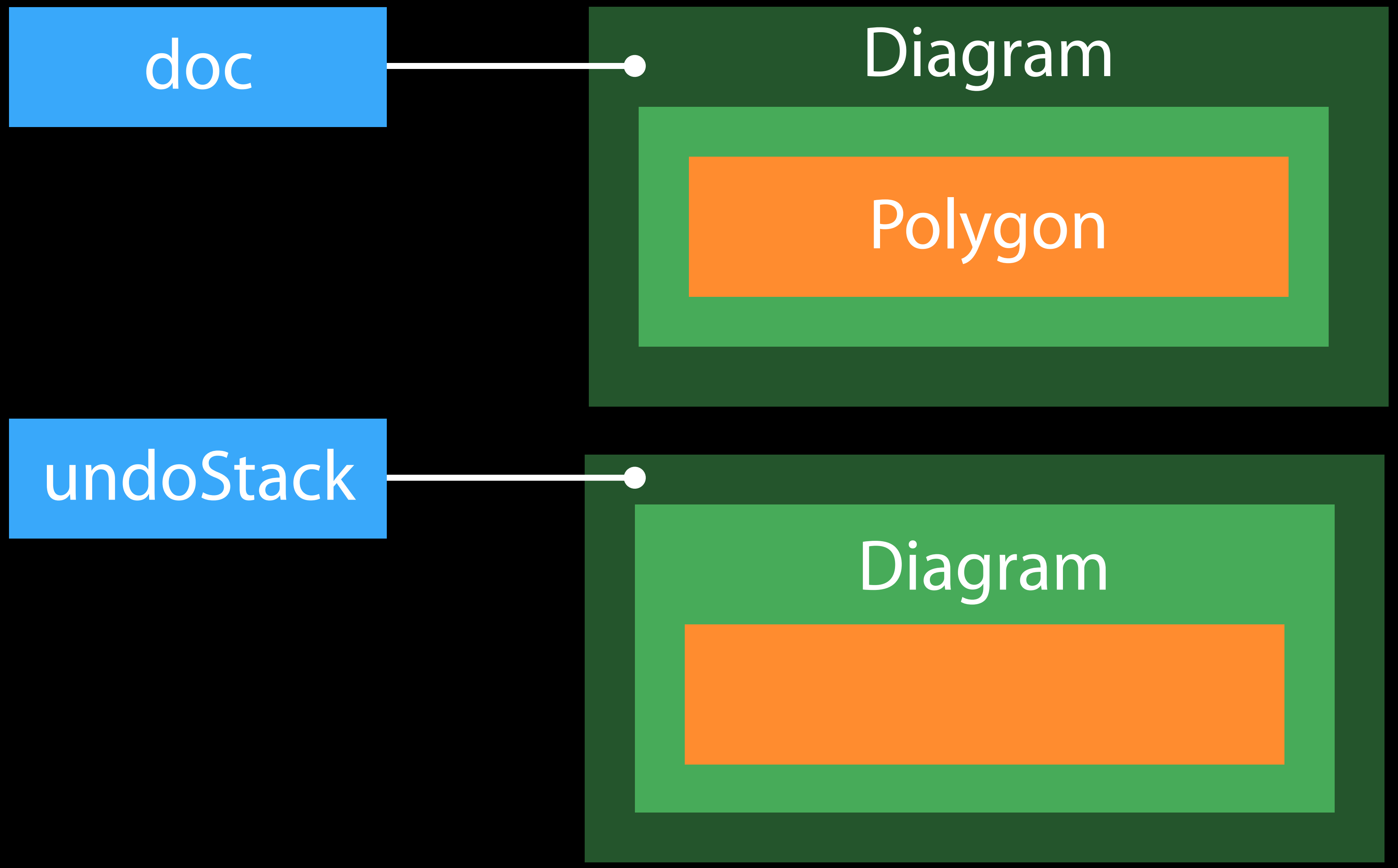
Undo

```
var doc = Diagram()
var undoStack: [Diagram] = []
undoStack.append(doc)
```



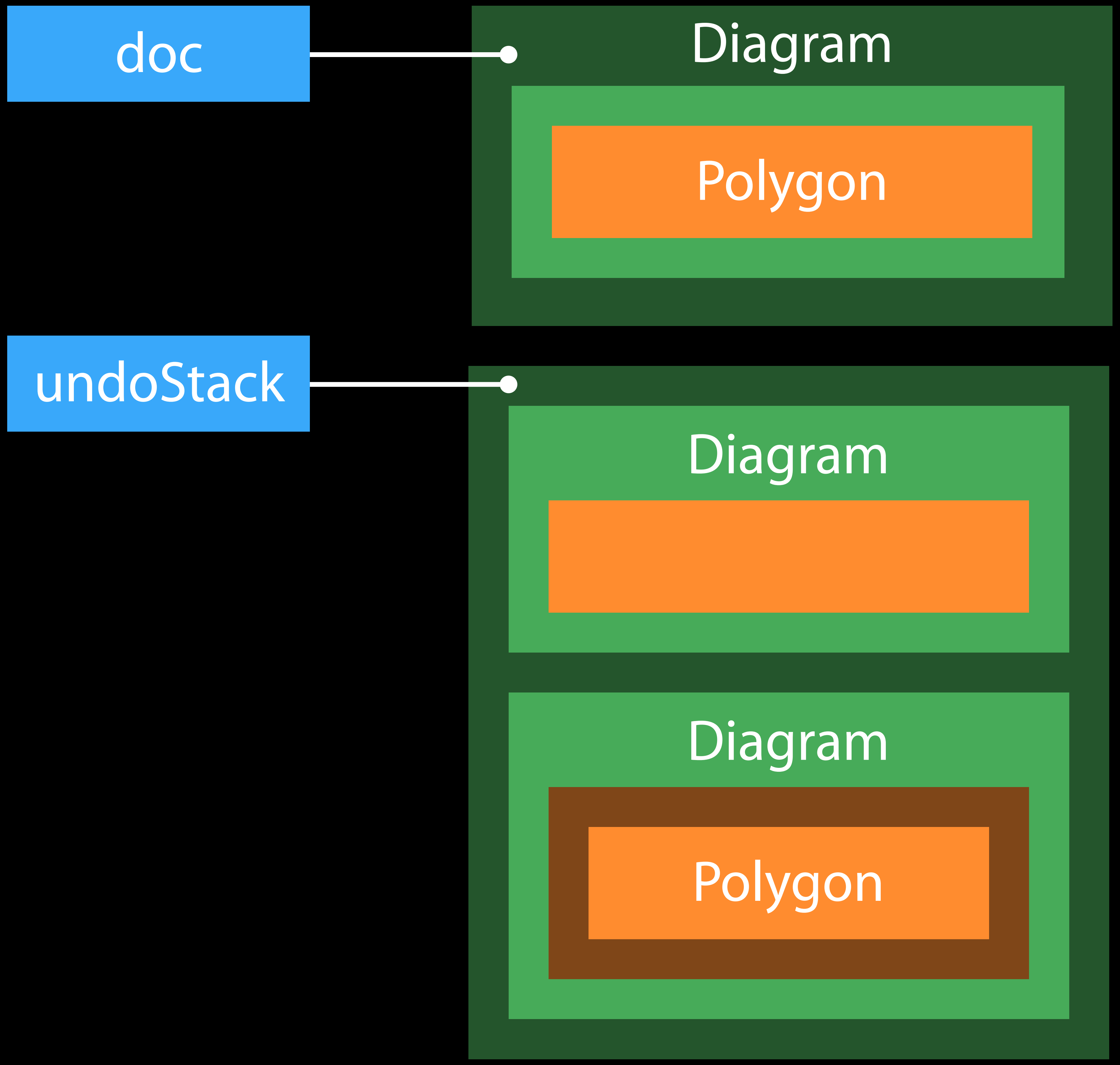
Undo

```
var doc = Diagram()
var undoStack: [Diagram] = []
undoStack.append(doc)
doc.addItem(Polygon())
```



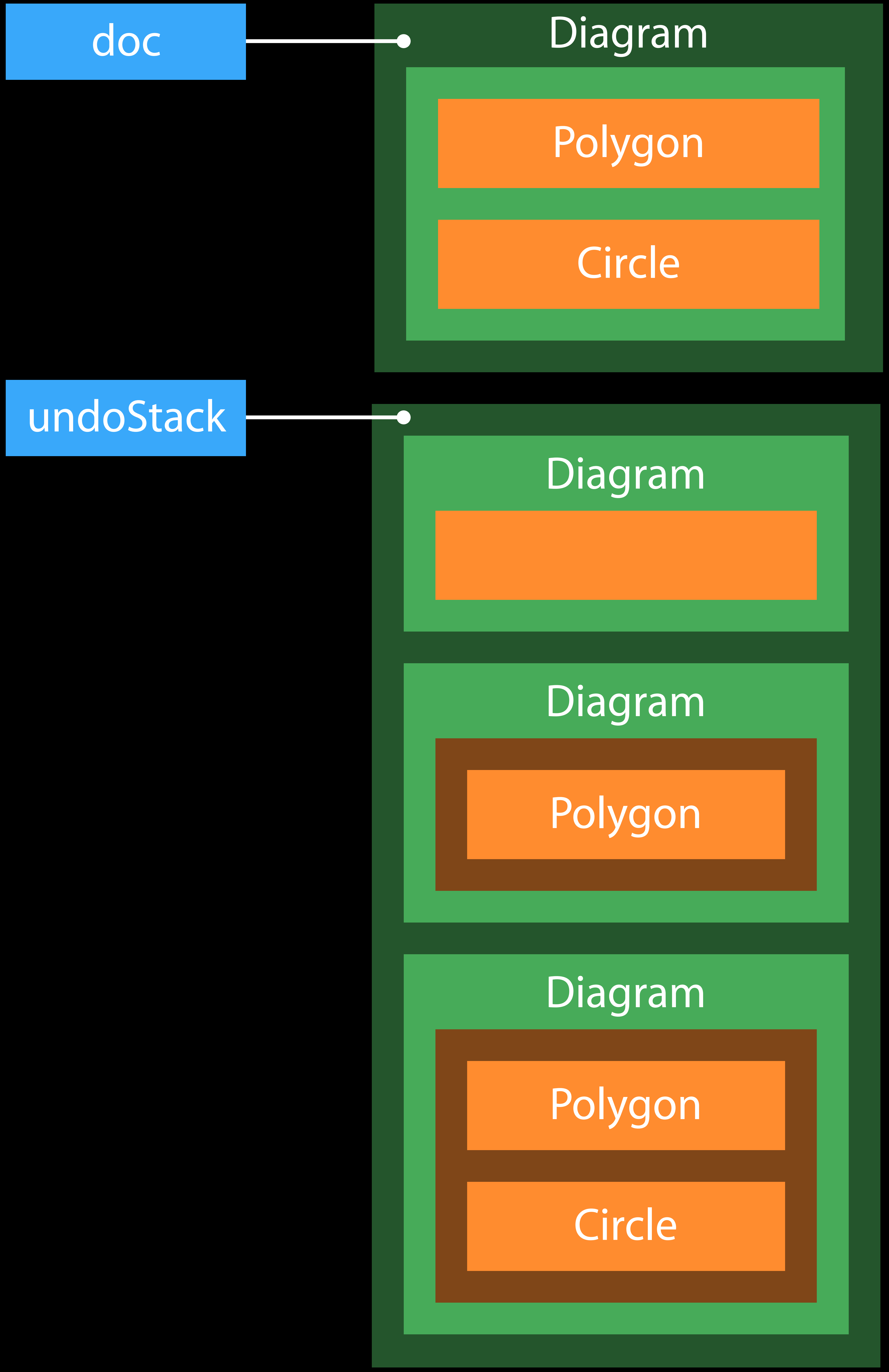
Undo

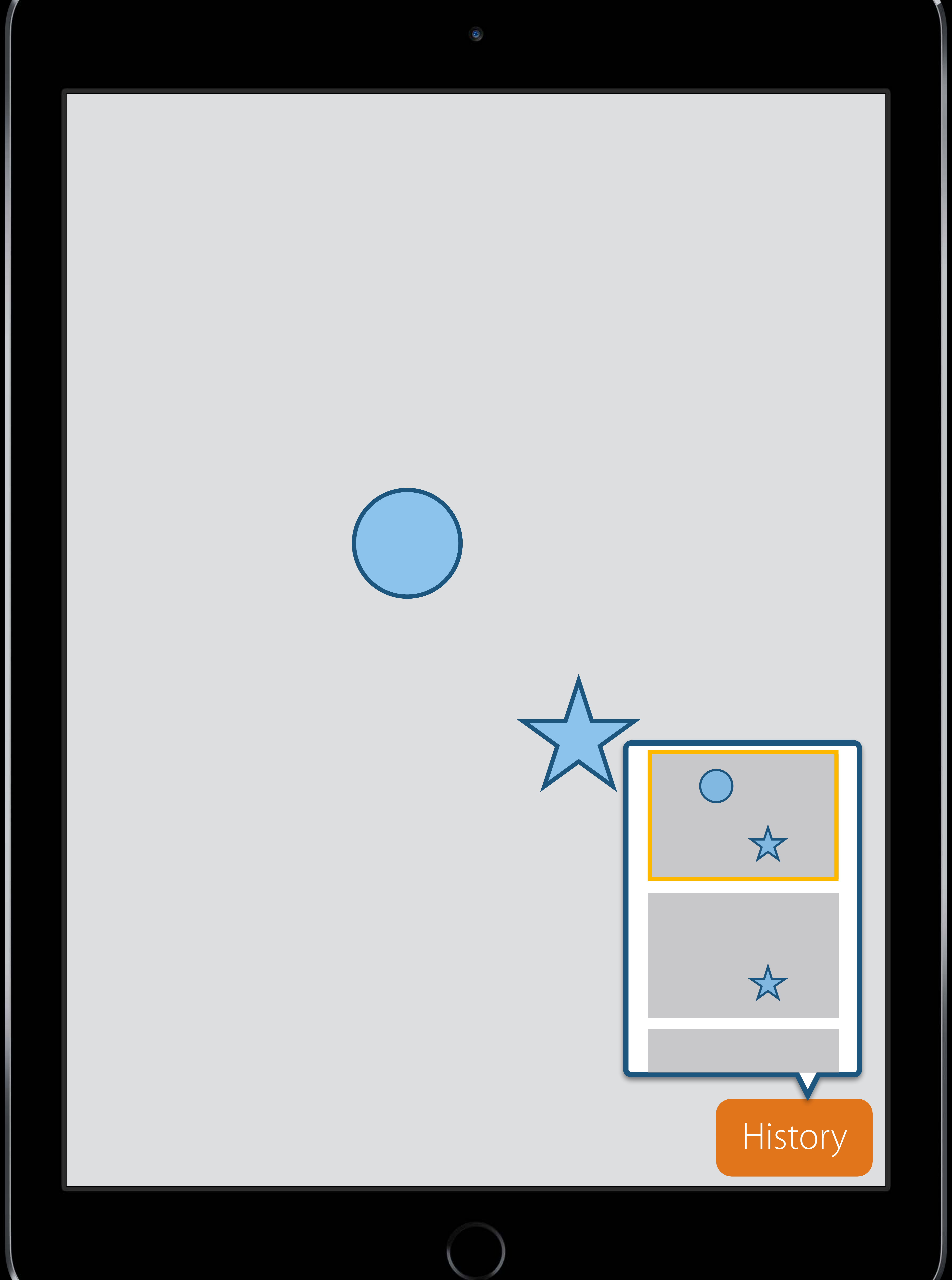
```
var doc = Diagram()
var undoStack: [Diagram] = []
undoStack.append(doc)
doc.addItem(Polygon())
undoStack.append(doc)
```

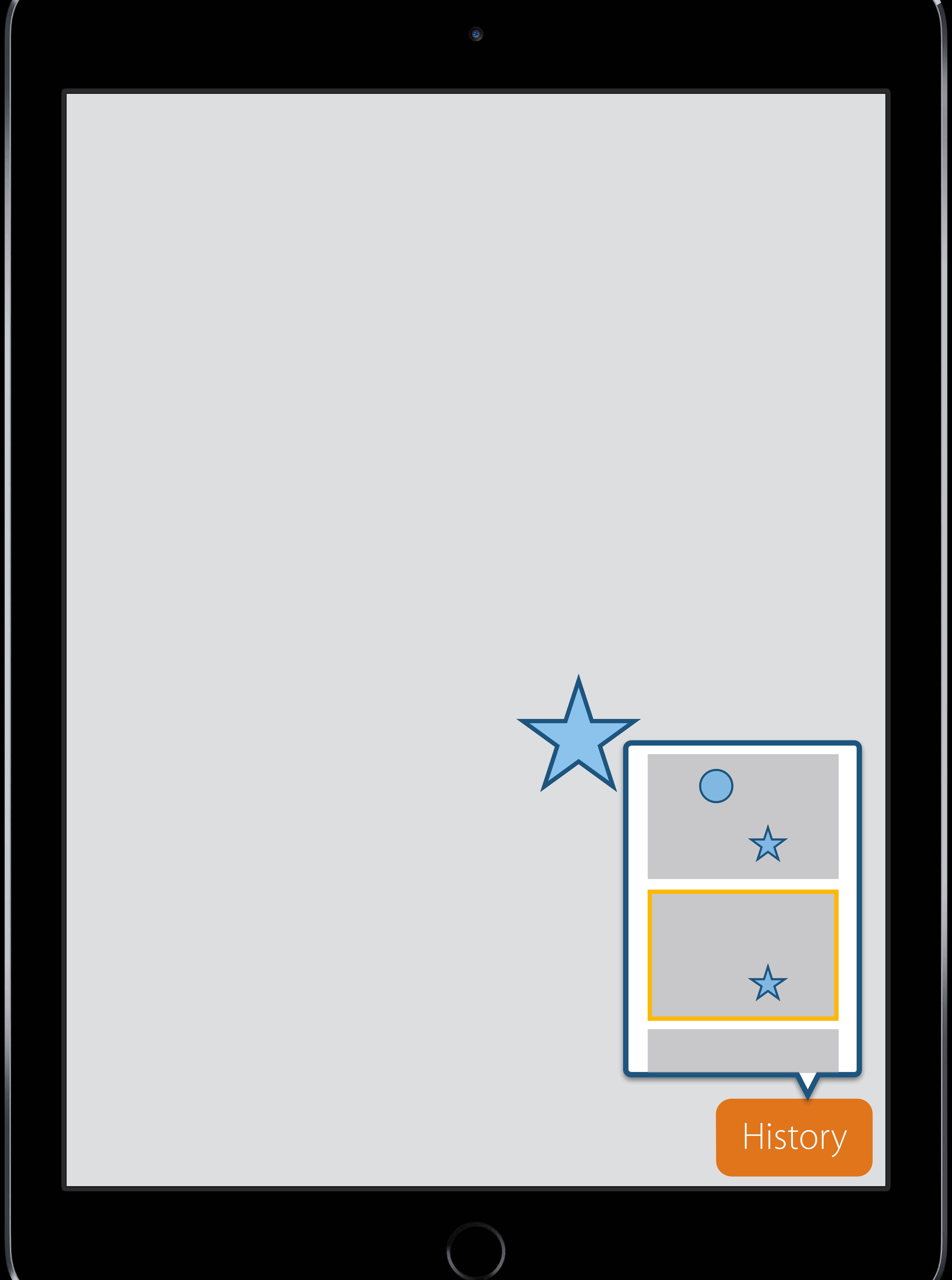


Undo

```
var doc = Diagram()
var undoStack: [Diagram] = []
undoStack.append(doc)
doc.addItem(Polygon())
undoStack.append(doc)
doc.addItem(Circle())
undoStack.append(doc)
```




History


History

Value Semantics

Photoshop uses value semantics

Every action results in a doc instance

Efficient because of copy-on-write

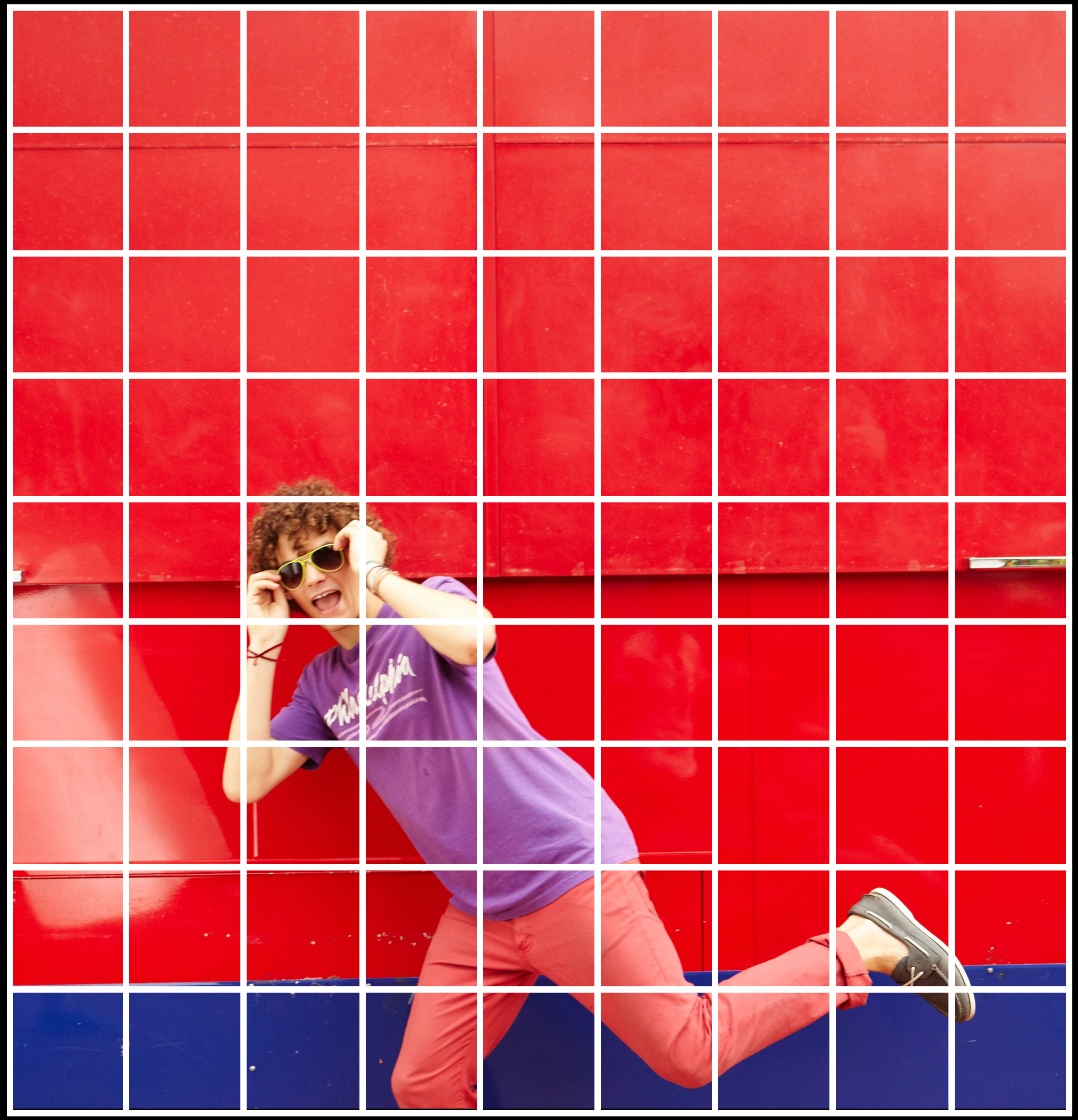
Parent, Sean.
Value Semantics and Concept-based Polymorphism.
C++ Now!, 2012

Value Semantics

Photoshop uses value semantics

Every action results in a doc instance

Efficient because of copy-on-write



Parent, Sean.
Value Semantics and Concept-based Polymorphism.
C++ Now!, 2012

Value Semantics

Photoshop uses value semantics

Every action results in a doc instance

Efficient because of copy-on-write

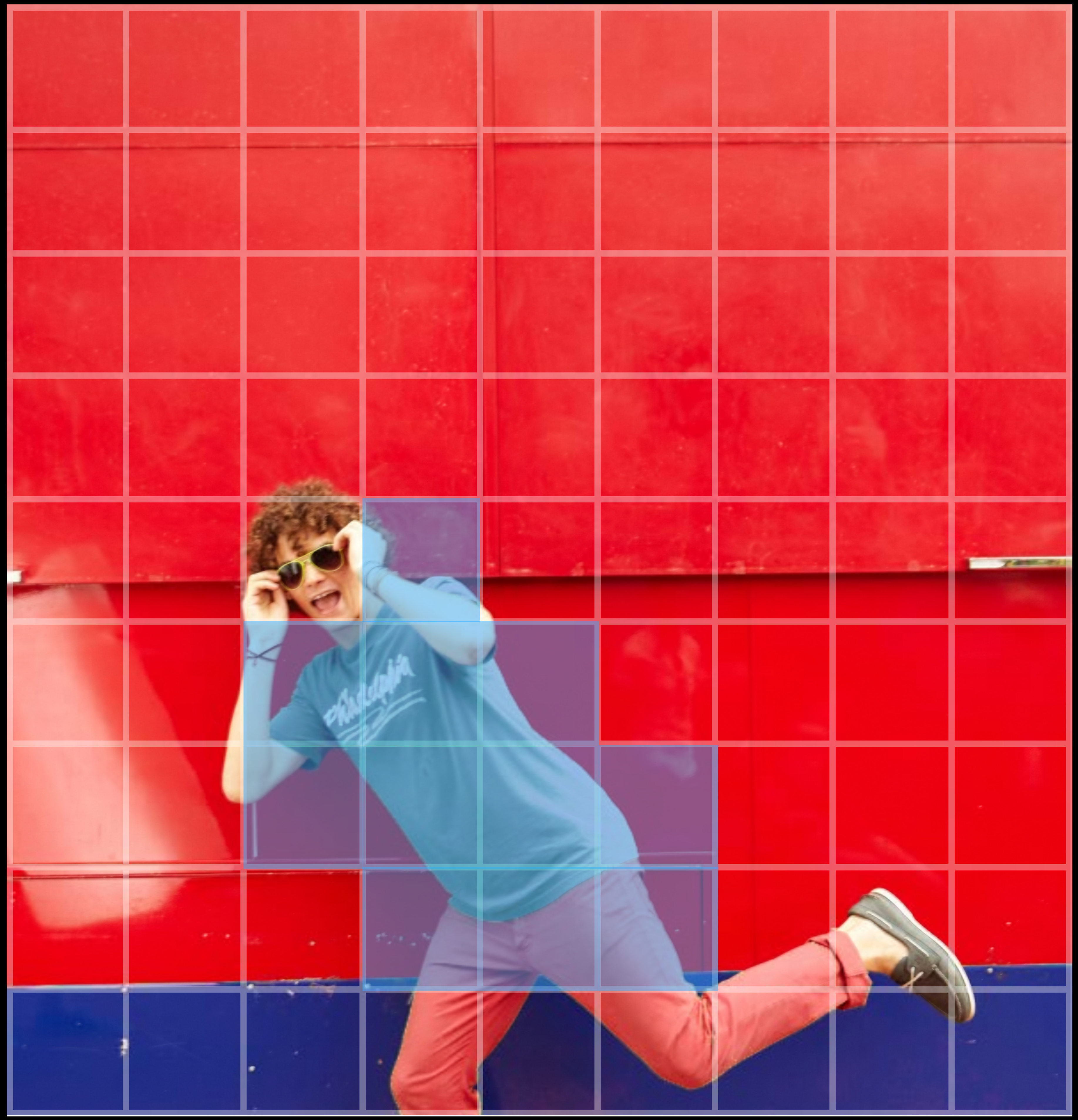
Parent, Sean.
Value Semantics and Concept-based Polymorphism.
C++ Now!, 2012

Value Semantics

Photoshop uses value semantics

Every action results in a doc instance

Efficient because of copy-on-write



Parent, Sean.
Value Semantics and Concept-based Polymorphism.
C++ Now!, 2012

Summary

Reference semantics and unexpected mutation

Value semantics solve these problems

Expressiveness of mutability, safety of immutability

Related Sessions

Protocol-Oriented Programming in Swift	Mission	Wednesday 2:30PM
Optimizing Swift Performance	Presidio	Thursday 9:00AM
Protocol-Oriented Programming in Swift (Repeat)	Pacific Heights	Friday 3:30PM

More Information

Swift Language Documentation
http://developer.apple.com/swift

Apple Developer Forums
http://developer.apple.com/forums

Stefan Lesser
Swift Evangelist
slesser@apple.com

WWDC15