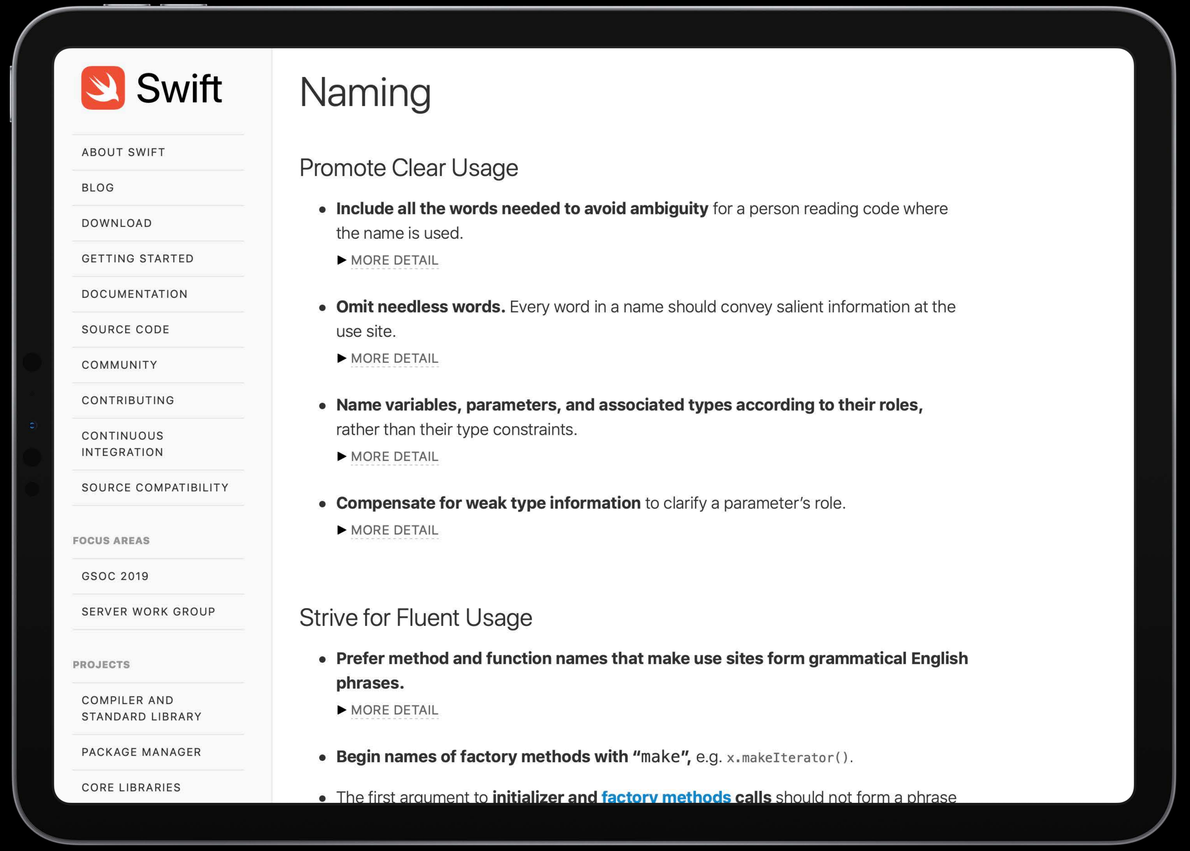
#WWDC19

Modern Swift API Design

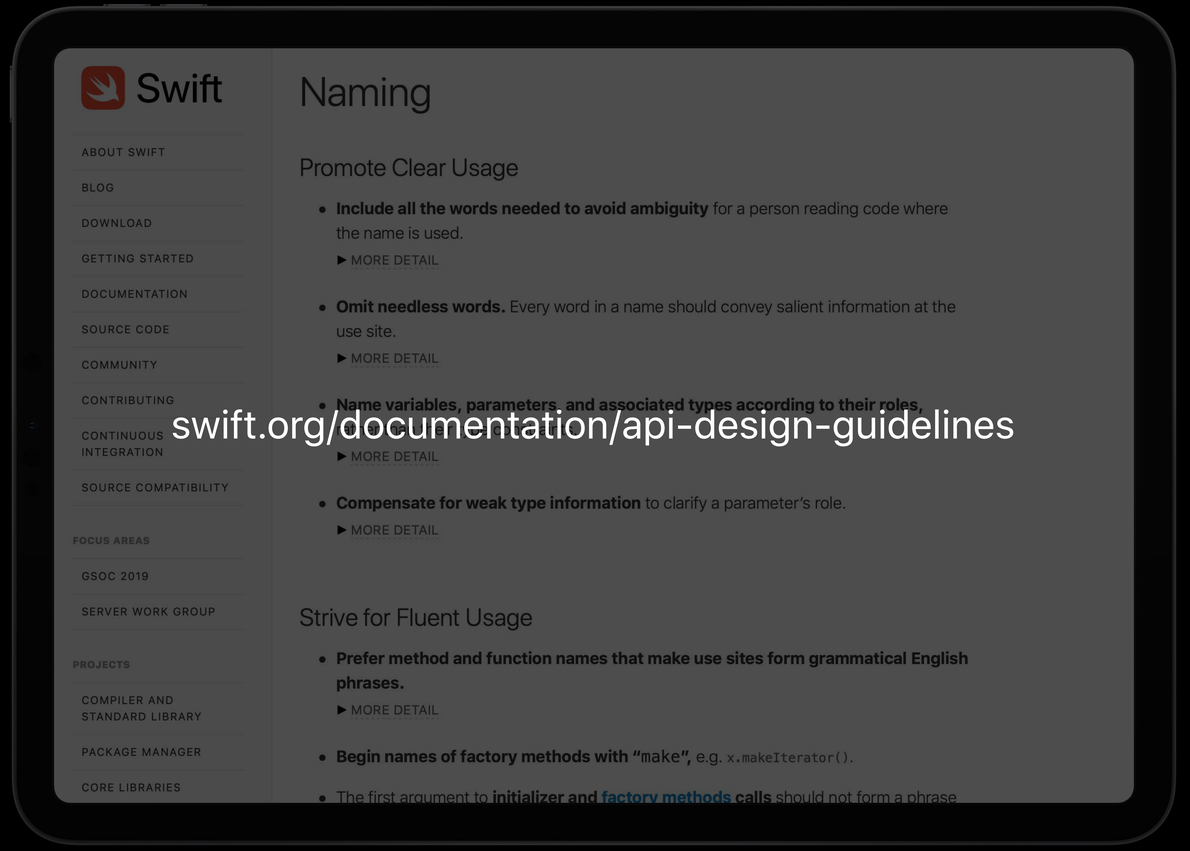
Ben Cohen, Swift Team
Doug Gregor, Swift Team

© 2019 Apple Inc. All rights reserved. Redistribution or public display not permitted without written permission from Apple.

API Design Guidelines



API Design Guidelines



swift.org/documentation/api-design-guidelines

Clarity at the point of use

No Prefixes in Swift-only Frameworks

No Prefixes in Swift-only Frameworks

C and Objective-C symbols are global

No Prefixes in Swift-only Frameworks

C and Objective-C symbols are global

Swift’s module system allows disambiguation

No Prefixes in Swift-only Frameworks

C and Objective-C symbols are global

Swift's module system allows disambiguation

Remember — each source file brings its imports into the same namespace

Values and references

Protocols and generics

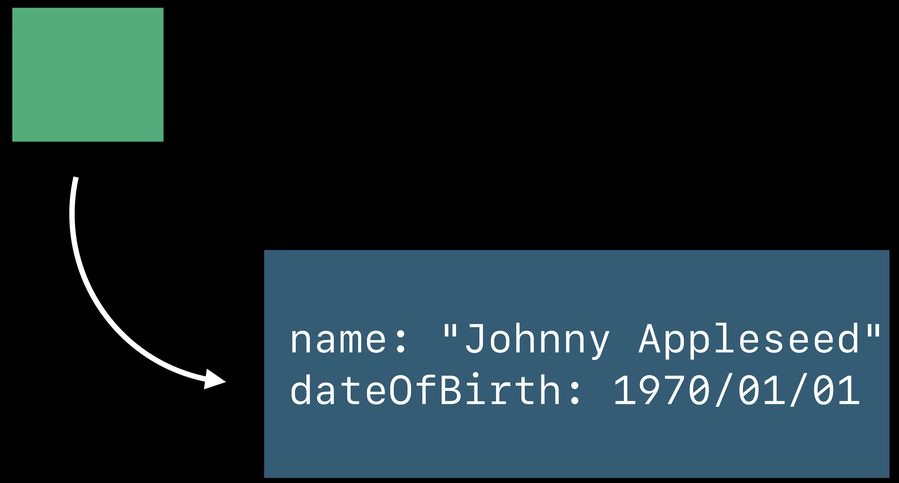
Key path member lookup

Property wrappers

Values and References

Classes — Reference Types

```
class User {
  var name: String
  var dateOfBirth: Date
}
```

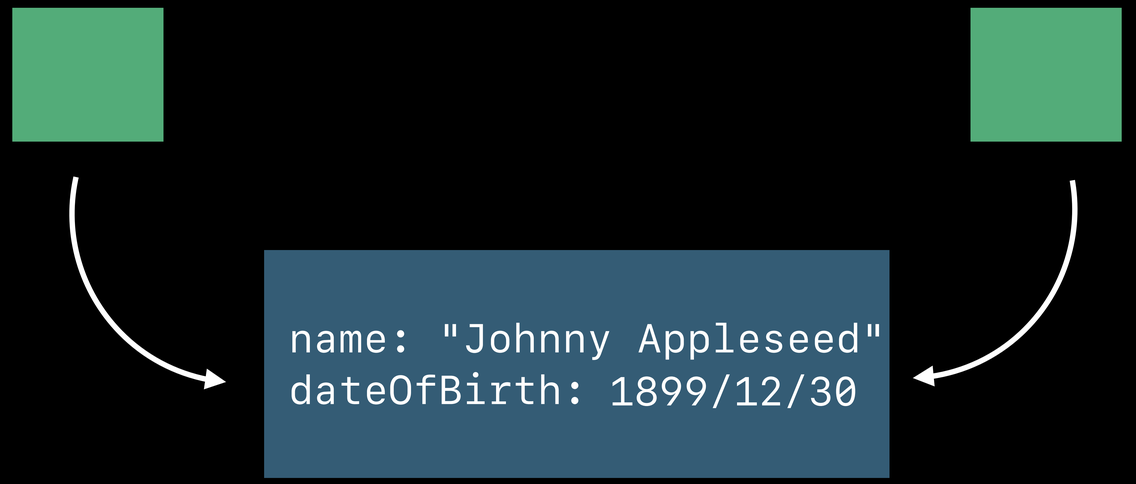


Classes — Reference Types

```
class User {
  var name: String
  var dateOfBirth: Date
}
```

Classes — Reference Types

```
class User {
  var name: String
  var dateOfBirth: Date
}
```



Structs and Enums — Value Types

```
struct User {
  var name: String
  var dateOfBirth: Date
}
```

```
name: "Johnny Appleseed"
dateOfBirth: 1970/01/01
```

Structs and Enums — Value Types

```
struct User {
  var name: String
  var dateOfBirth: Date
}
```

```
name: "Johnny Appleseed"
dateOfBirth: 1970/01/01
```

```
name: "Johnny Appleseed"
dateOfBirth: 1970/01/01
```

Structs and Enums — Value Types

```
struct User {
  var name: String
  var dateOfBirth: Date
}
```

```
name: "Johnny Appleseed"
dateOfBirth: 1970/01/01
```

```
name: "Johnny Appleseed"
dateOfBirth: 1899/12/30
```

Choosing — Reference or Value?

Choosing — Reference or Value?

Prefer structs over classes

- Only choose classes when reference semantics are important

Choosing — Reference or Value?

Prefer structs over classes

- Only choose classes when reference semantics are important

Classes can make a good choice when

Choosing — Reference or Value?

Prefer structs over classes

- Only choose classes when reference semantics are important

Classes can make a good choice when

- You need a reference counting and deinitialization

Choosing — Reference or Value?

Prefer structs over classes

- Only choose classes when reference semantics are important

Classes can make a good choice when

- You need a reference counting and deinitialization
- The value is held centrally and shared

Choosing — Reference or Value?

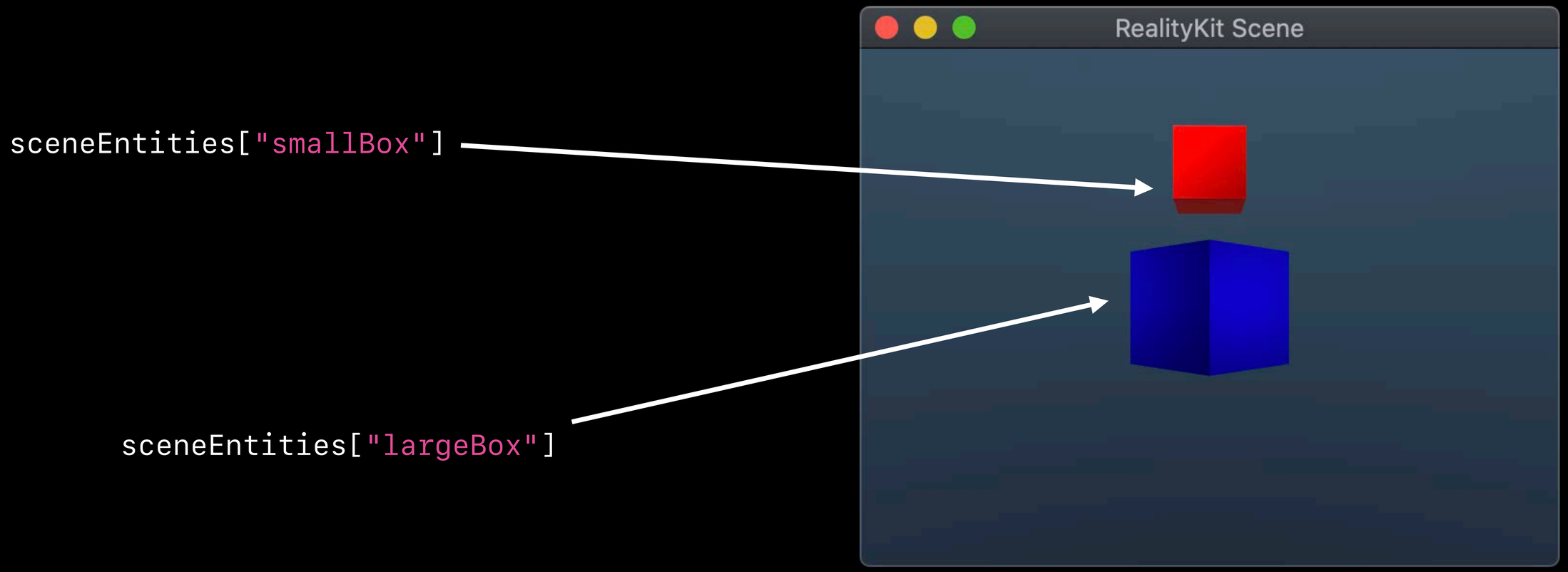
Prefer structs over classes

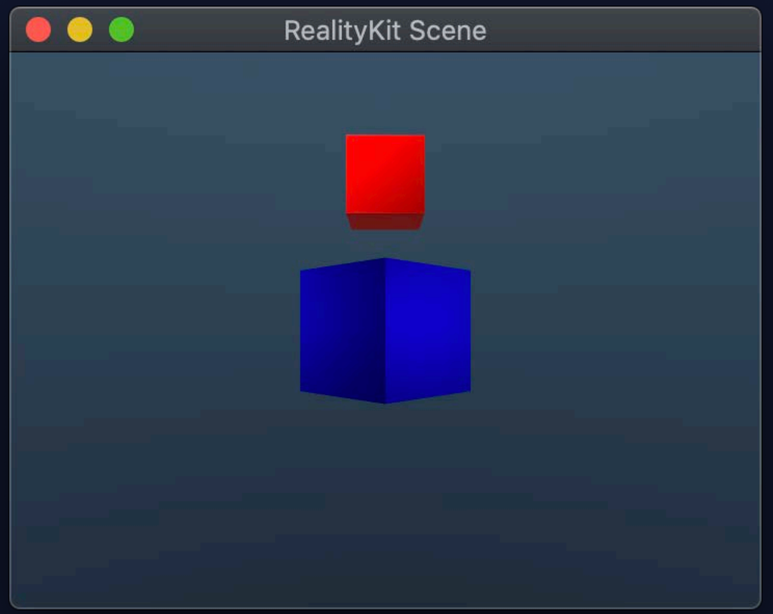
- Only choose classes when reference semantics are important

Classes can make a good choice when

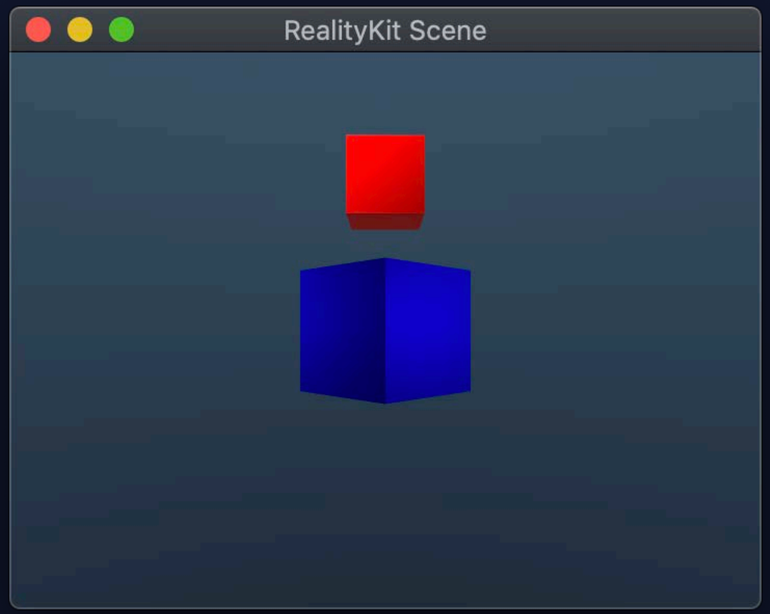
- You need a reference counting and deinitialization
- The value is held centrally and shared
- Where there is a separate notion of “identity” from “equality”

RealityKit — Scenes and Entities

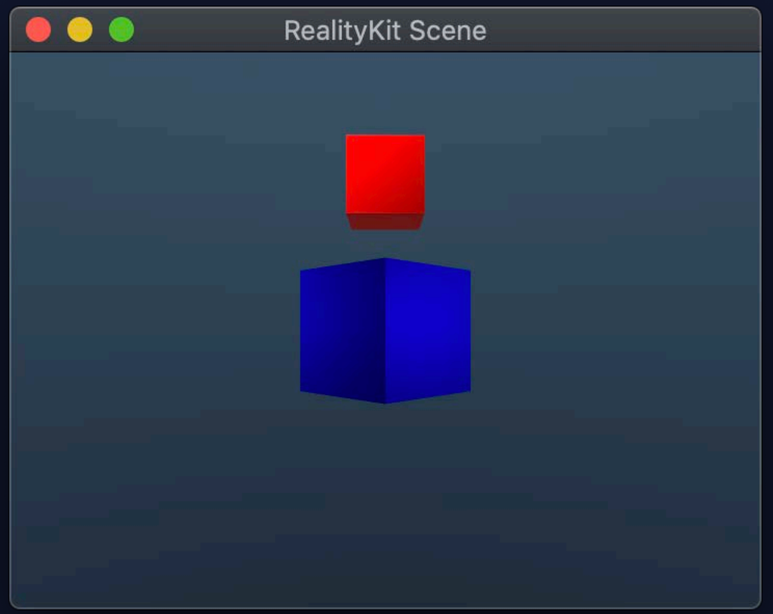



RealityKit Scene

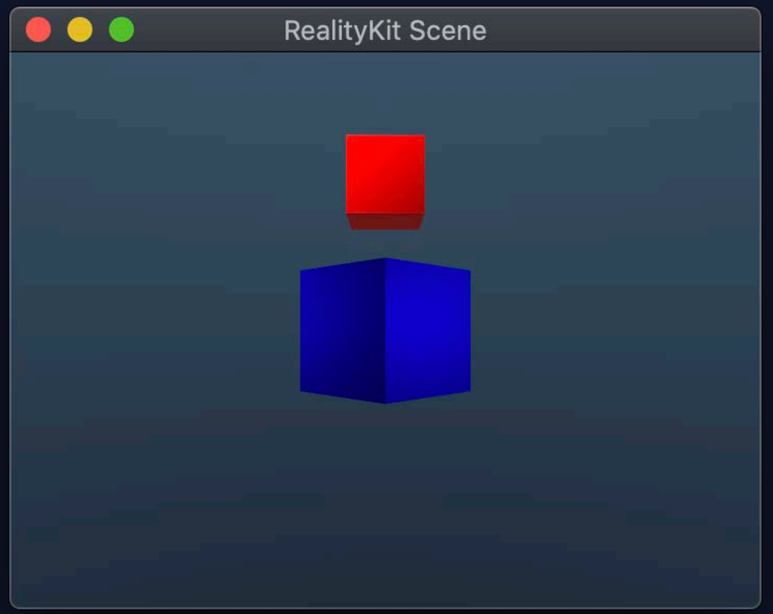
```
var material = SimpleMaterial(color: .blue, roughness: 0.3, isMetallic: true)
let largeBox = ModelEntity(mesh: .generateBox(size: 0.5), materials: [material])
let smallBox = ModelEntity(mesh: .generateBox(size: 0.3), materials: [material])
```



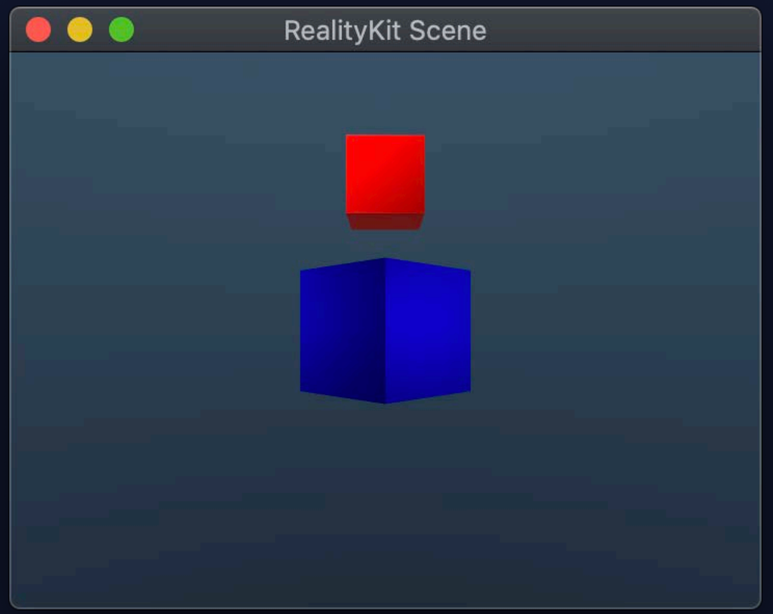
```swift
var material = SimpleMaterial(color: .blue, roughness: 0.3, isMetallic: true)
let largeBox = ModelEntity(mesh: .generateBox(size: 0.5), materials: [material])
let smallBox = ModelEntity(mesh: .generateBox(size: 0.3), materials: [material])


let anchor = AnchorEntity()
anchor.children.append(largeBox)
anchor.children.append(smallBox)
arView.scene.anchors.append(anchor)
```



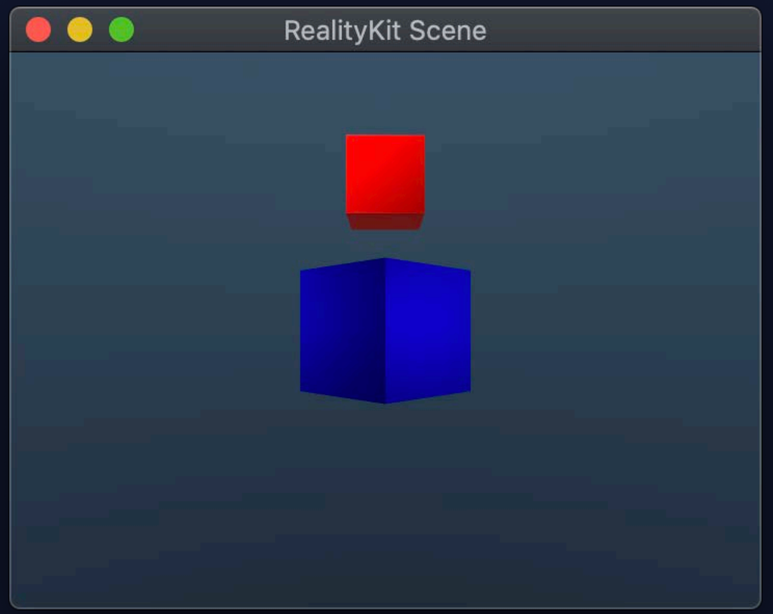
```swift
var material = SimpleMaterial(color: .blue, roughness: 0.3, isMetallic: true)
let largeBox = ModelEntity(mesh: .generateBox(size: 0.5), materials: [material])
let smallBox = ModelEntity(mesh: .generateBox(size: 0.3), materials: [material])


let anchor = AnchorEntity()
anchor.children.append(largeBox)
anchor.children.append(smallBox)
arView.scene.anchors.append(anchor)

smallBox.position.y += 0.6
```



```
var material = SimpleMaterial(color: .blue, roughness: 0.3, isMetallic: true)
let largeBox = ModelEntity(mesh: .generateBox(size: 0.5), materials: [material])
let smallBox = ModelEntity(mesh: .generateBox(size: 0.3), materials: [material])


let anchor = AnchorEntity()
anchor.children.append(largeBox)
anchor.children.append(smallBox)
arView.scene.anchors.append(anchor)

smallBox.position.y += 0.6
largeBox.orientation = .init(angle: .pi/4, axis: [0,1,0])
```



```
var material = SimpleMaterial(color: .blue, roughness: 0.3, isMetallic: true)
let largeBox = ModelEntity(mesh: .generateBox(size: 0.5), materials: [material])
material.tintColor = .red
let smallBox = ModelEntity(mesh: .generateBox(size: 0.3), materials: [material])

let anchor = AnchorEntity()
anchor.children.append(largeBox)
anchor.children.append(smallBox)
arView.scene.anchors.append(anchor)


smallBox.position.y += 0.6
largeBox.orientation = .init(angle: .pi/4, axis: [0,1,0])
```

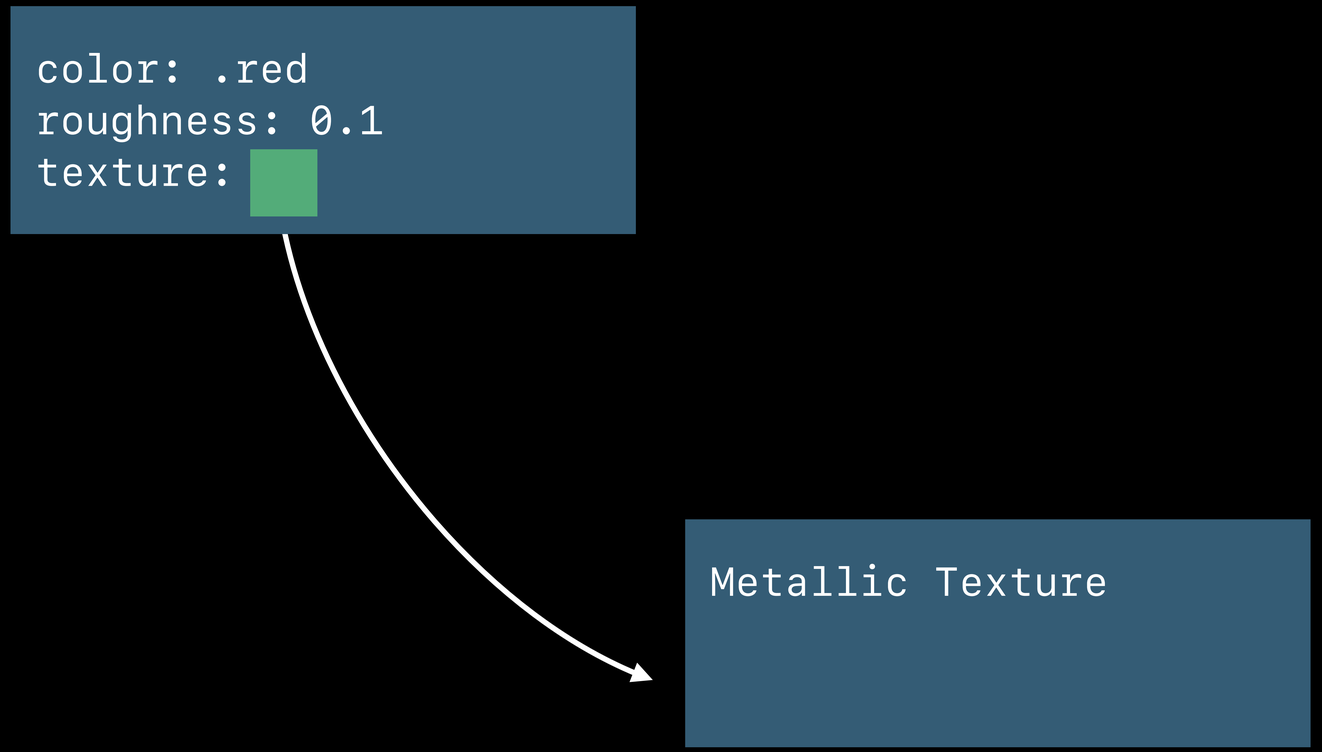


```
struct Material {
  public var roughness: Float
  public var color: Color
  public var texture: Texture
}
```

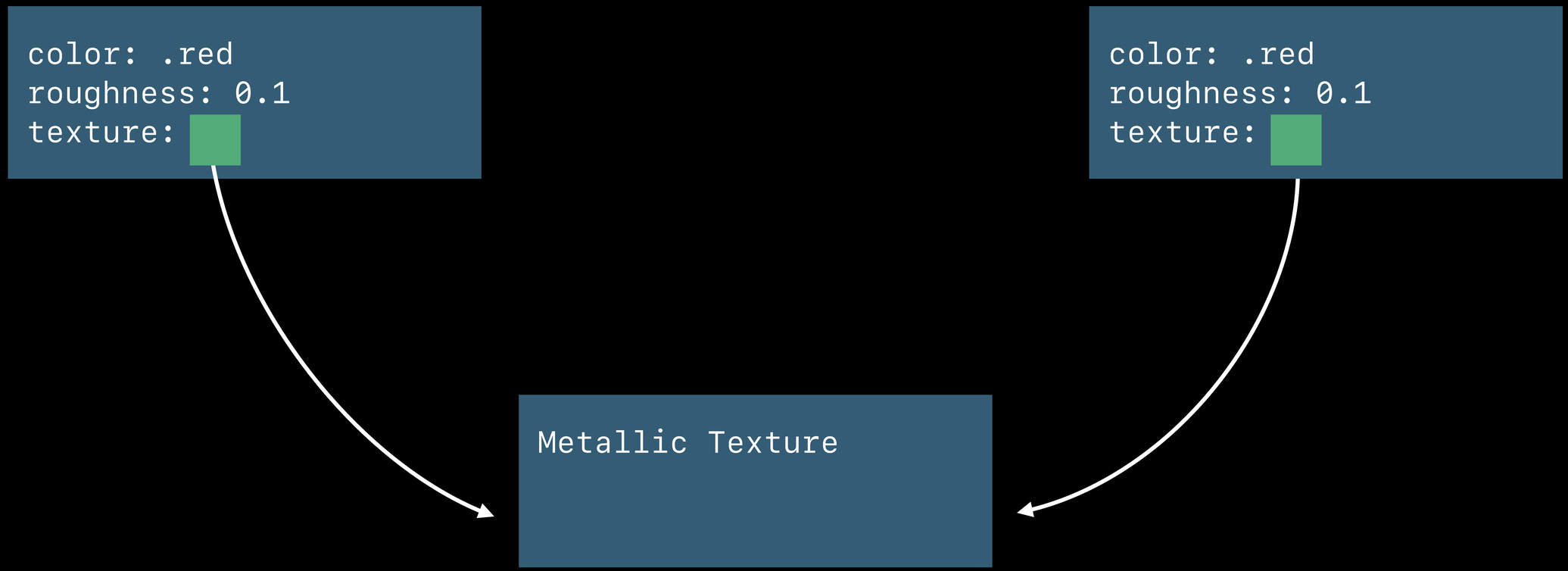
```
struct Material {
  public var roughness: Float
  public var color: Color
  public var texture: Texture
}
```

```
struct Material {
  public var roughness: Float
  public var color: Color
  public var texture: Texture
}


class Texture { … }
```

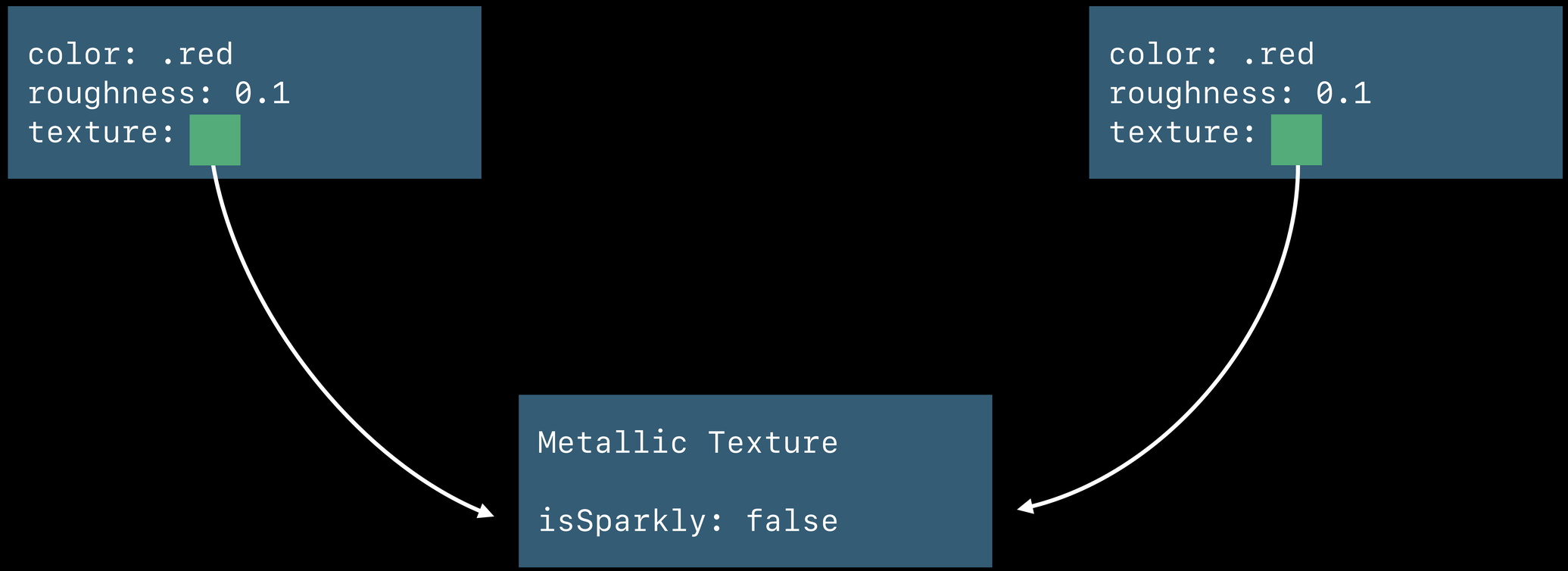
Value Types Make Copies of References



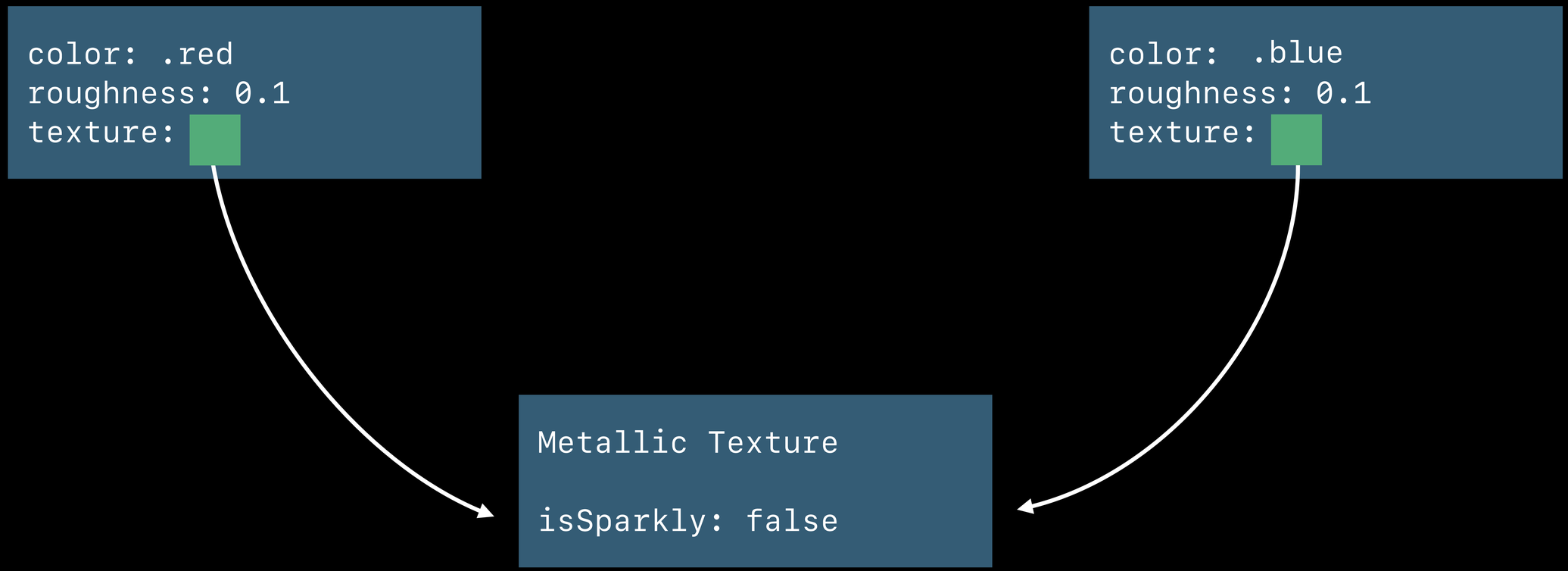
Value Types Make Copies of References



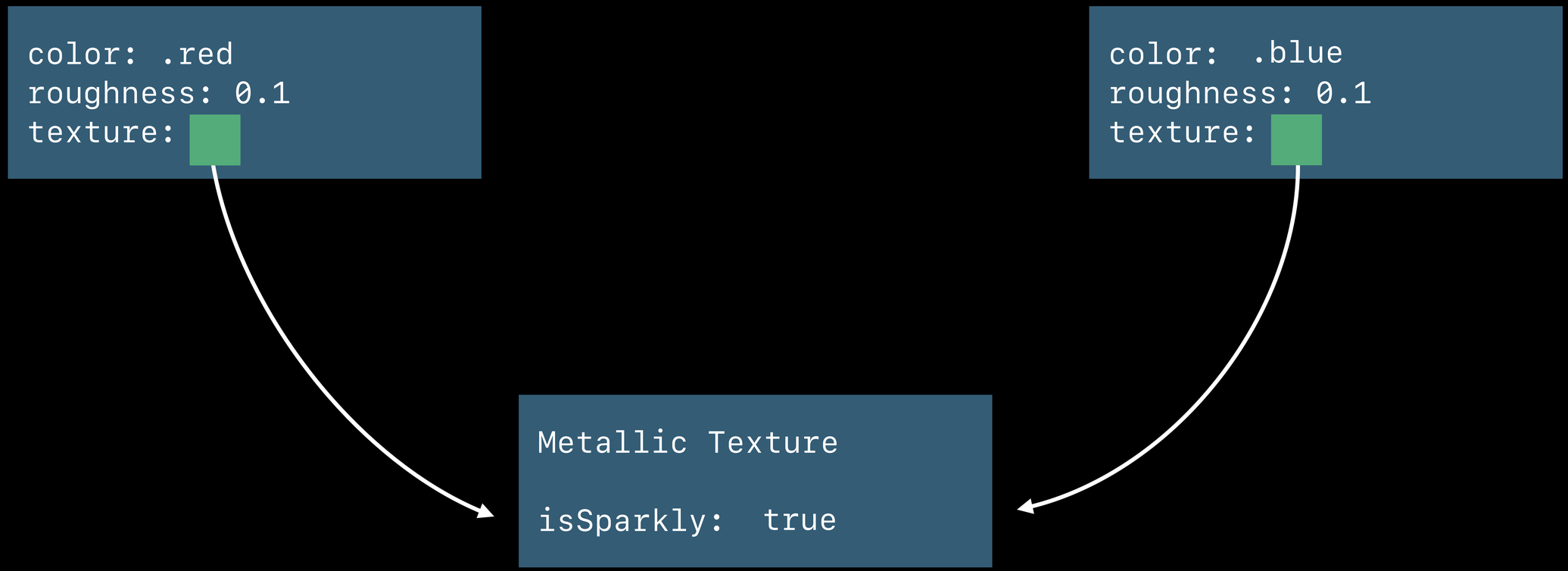
Value Types Make Copies of References



Value Types Make Copies of References



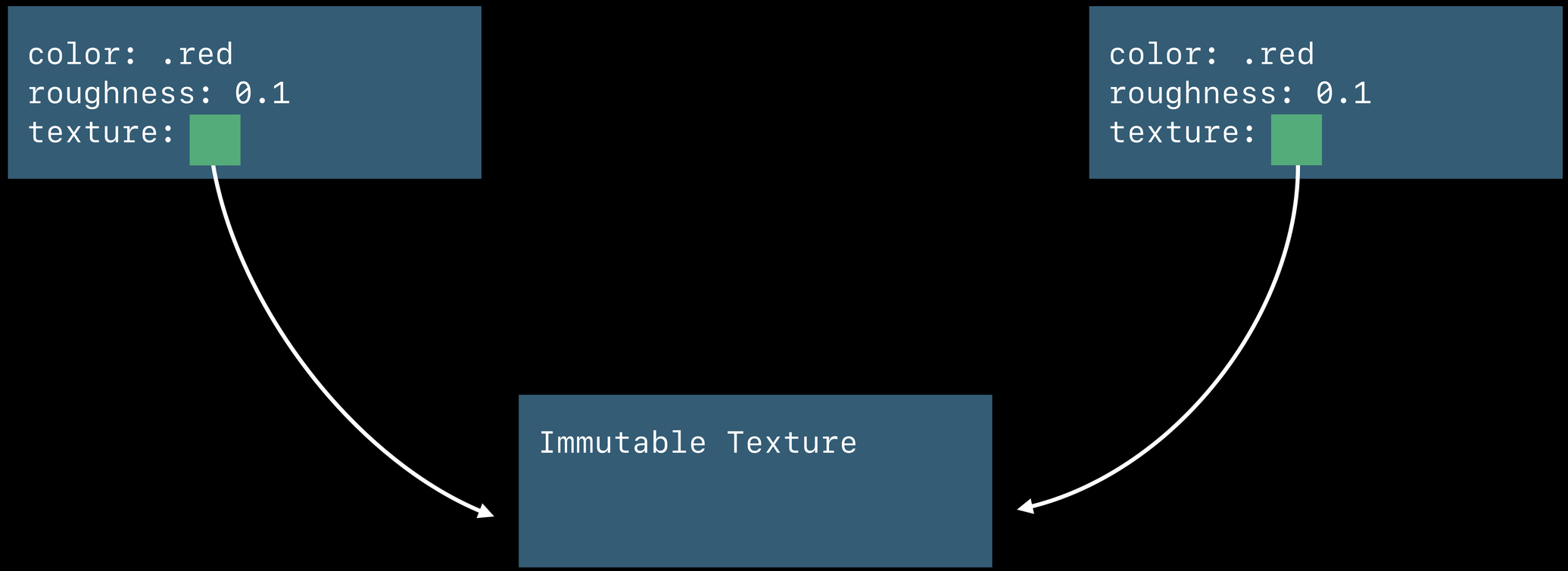
Value Types Make Copies of References



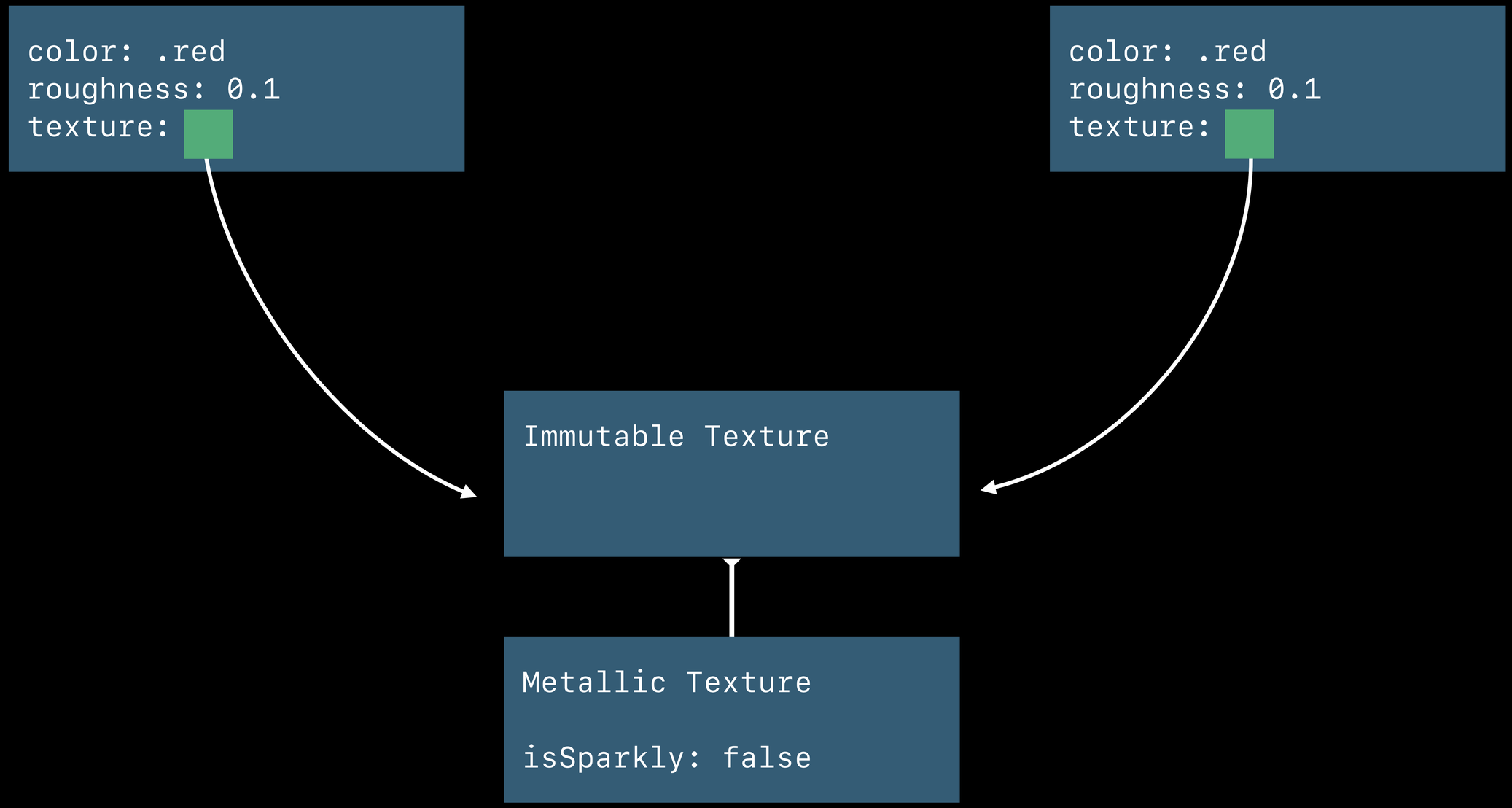
Value and Reference Types

Value and Reference Semantics

Value Types Make Copies of References



Value Types Make Copies of References



```
struct Material {
  public var roughness: Float
  public var color: Color

  public var texture: Texture




}
```

```
struct Material {
  public var roughness: Float
  public var color: Color

  private var _texture: Texture




}
```

```
struct Material {
  public var roughness: Float
  public var color: Color

  private var _texture: Texture

  public var texture {
    get { _texture }
    set { _texture = Texture(copying: newValue) }
  }
}
```

```
struct Material {
  public var roughness: Float
  public var color: Color

  private var _texture: Texture

  public var texture {
    get { _texture }
    set { _texture = Texture(copying: newValue) }
  }
}
```

```
struct Material {
  public var roughness: Float
  public var color: Color

  private var _texture: Texture




}
```

```
struct Material {
  public var roughness: Float
  public var color: Color

  private var _texture: Texture

  public var isSparkly: Bool {
    get { _texture.isSparkly }
    set {


    }
  }
}
```

```
struct Material {
  public var roughness: Float
  public var color: Color

  private var _texture: Texture

  public var isSparkly: Bool {
    get { _texture.isSparkly }
    set {


    }
  }
}
```

```
struct Material {
  public var roughness: Float
  public var color: Color

  private var _texture: Texture

  public var isSparkly: Bool {
    get { _texture.isSparkly }
    set {
      if !isKnownUniquelyReferenced(&_texture) { _texture = Texture(copying: _texture)  }
      _texture.isSparkly = newValue
    }
  }
}
```

```
struct Material {
  public var roughness: Float
  public var color: Color

  private var _texture: Texture

  public var isSparkly: Bool {
    get { _texture.isSparkly }
    set {
      if !isKnownUniquelyReferenced(&_texture) { _texture = Texture(copying: _texture)  }
      _texture.isSparkly = newValue
    }
  }
}
```

```
struct Material {
  public var roughness: Float
  public var color: Color

  private var _texture: Texture

  public var isSparkly: Bool {
    get { _texture.isSparkly }
    set {
      if !isKnownUniquelyReferenced(&_texture) { _texture = Texture(copying: _texture)  }
      _texture.isSparkly = newValue
    }
  }
}
```

Protocols and Generics

```
// Protocols Can Apply to Value Types
```

```
// Protocols Can Apply to Value Types

class NSOrderedSet: Sequence

struct Array: Sequence

struct CGPoint: Equatable

enum Optional: Equatable where Wrapped: Equatable
```

Don’t Literally Start With a Protocol

Don’t Literally Start With a Protocol

Start with concrete use cases

Don’t Literally Start With a Protocol

Start with concrete use cases

Discover a need for generic code

Don’t Literally Start With a Protocol

Start with concrete use cases

Discover a need for generic code

Try to compose solutions from existing protocols first

Don’t Literally Start With a Protocol

Start with concrete use cases

Discover a need for generic code

Try to compose solutions from existing protocols first

Consider a generic type instead of a protocol

```
protocol GeometricVector {




}
```

```
protocol GeometricVector {
    static func dot(_ a: Self, _ b: Self) -> Scalar

    var length: Scalar { get }

    func distance(to other: Self) -> Scalar
}
```

```
protocol GeometricVector : SIMD {
    static func dot(_ a: Self, _ b: Self) -> Scalar

    var length: Scalar { get }

    func distance(to other: Self) -> Scalar
}
```

```
protocol GeometricVector : SIMD where Scalar: FloatingPoint {
    static func dot(_ a: Self, _ b: Self) -> Scalar

    var length: Scalar { get }

    func distance(to other: Self) -> Scalar
}
```

```
extension GeometricVector {
  static func dot(_ a: Self, _ b: Self) -> Scalar {
    (a * b).sum()
  }

  var length: Scalar {
    Self.dot(self, self).squareRoot()
  }

  func distance(to other: Self) -> Scalar {
    (self - other).length
  }
}
```

```
extension SIMD2: GeometricVector where Scalar: FloatingPoint { }

extension SIMD3: GeometricVector where Scalar: FloatingPoint { }

extension SIMD4: GeometricVector where Scalar: FloatingPoint { }
```

```
extension SIMD2: GeometricVector where Scalar: FloatingPoint { }

extension SIMD3: GeometricVector where Scalar: FloatingPoint { }

extension SIMD4: GeometricVector where Scalar: FloatingPoint { }

// sure why not ¯\_(ツ)_/¯
extension SIMD64: GeometricVector where Scalar: FloatingPoint { }
```

```
extension GeometricVector {
  static func dot(_ a: Self, _ b: Self) -> Scalar {
    (a * b).sum()
  }

  var length: Scalar {
    Self.dot(self, self).squareRoot()
  }

  func distance(to other: Self) -> Scalar {
    (self - other).length
  }
}
```

```swift
extension SIMD where Scalar: FloatingPoint {
  static func dot(_ a: Self, _ b: Self) -> Scalar {
    (a * b).sum()
  }

  var length: Scalar {
    Self.dot(self, self).squareRoot()
  }

  func distance(to other: Self) -> Scalar {
    (self - other).length
  }
}
```

```
protocol GeometricVector: SIMD
```

```
struct GeometricVector<Storage: SIMD> where Storage.Scalar: FloatingPoint {
  typealias Scalar = Storage.Scalar

  var value: Storage

  init(_ value: Storage) { self.value = value }
}
```

```
struct GeometricVector<Storage: SIMD> where Storage.Scalar: FloatingPoint {
  typealias Scalar = Storage.Scalar

  var value: Storage

  init(_ value: Storage) { self.value = value }
}
```

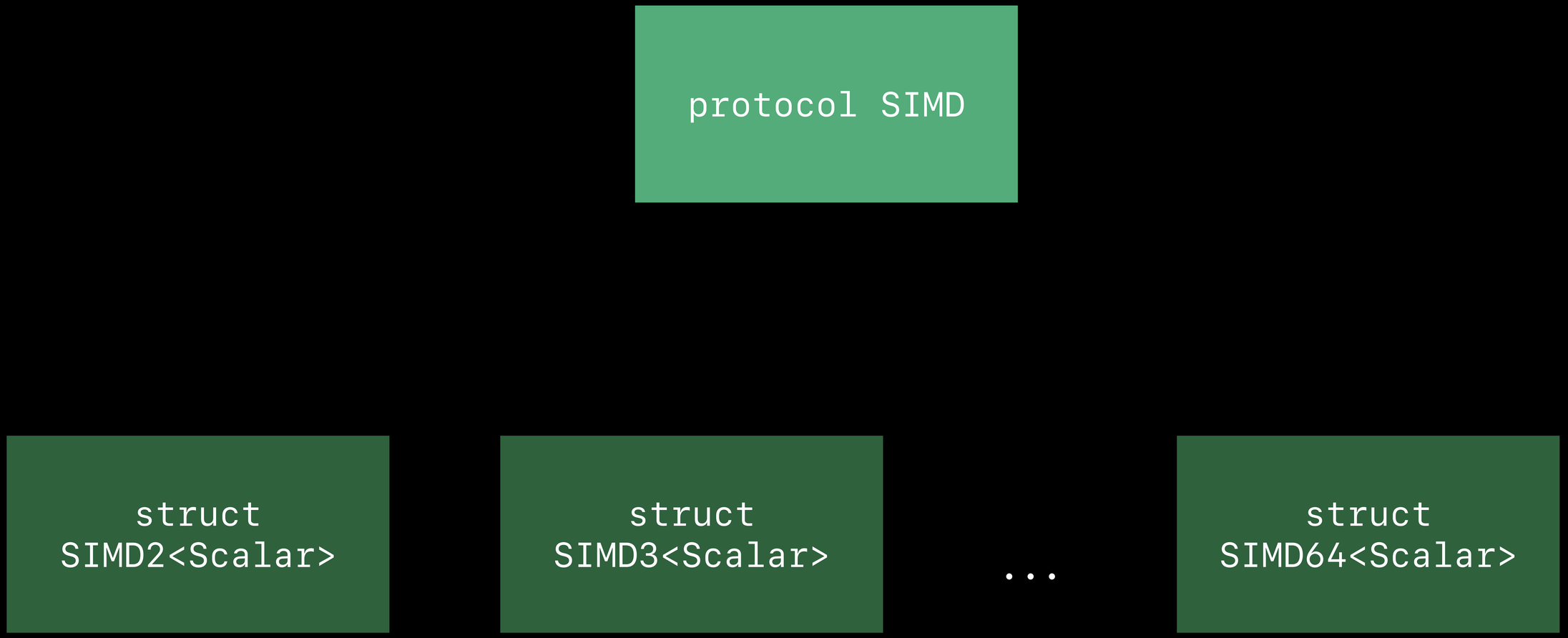
```
extension GeometricVector {
  static func + (a: Self, b: Self) -> Self {
    Self(a.value + b.value)
  }

  static func - (a: Self, b: Self) -> Self {
    Self(a.value - b.value)
  }

  static func * (a: Self, b: Scalar) -> Self {
    Self(a.value * b)
  }
}
```

```
extension GeometricVector {
  static func + (a: Self, b: Self) -> Self {
    Self(a.value + b.value)
  }

  static func - (a: Self, b: Self) -> Self {
    Self(a.value - b.value)
  }

  static func * (a: Self, b: Scalar) -> Self {
    Self(a.value * b)
  }
}
```

```swift
extension GeometricVector {
  static func + (a: Self, b: Self) -> Self {
    Self(a.value + b.value)
  }

  static func - (a: Self, b: Self) -> Self {
    Self(a.value - b.value)
  }

  static func * (a: Self, b: Scalar) -> Self {
    Self(a.value * b)
  }
}
```

```
extension GeometricVector {
  static func dot(_ a: Self, _ b: Self) -> Scalar {
    (a.value * b.value).sum()
  }

  var length: Scalar {
    Self.dot(self, self).squareRoot()
  }

  func distance(to other: Self) -> Scalar {
    (self - other).length
  }
}
```

SIMD Protocol and Structs



```
infix operator × : MultiplicationPrecedence

extension SIMD3 where Scalar: FloatingPoint {
  /// Cross product for two vectors
  static func × (_ a: Self, _ b: Self) -> Self {
    Self(
      a.y*b.z - a.z*b.y,
      a.z*b.x - a.x*b.z,
      a.x*b.y - a.y*b.x
    )
  }
}
```

```swift
extension GeometricVector {
  /// Cross product for two vectors
  static func ✕ <T>(_ a: Self, _ b: Self) -> Self where Storage == SIMD3<T> {
    Self([
      a.value.y*b.value.z - a.value.z*b.value.y,
      a.value.z*b.value.x - a.value.x*b.value.z,
      a.value.x*b.value.y - a.value.y*b.value.x,
    ])
  }
}
```

```swift
extension GeometricVector {
  /// Cross product for two vectors
  static func ✕ <T>(_ a: Self, _ b: Self) -> Self where Storage == SIMD3<T> {
    Self([
      a.value.y*b.value.z - a.value.z*b.value.y,
      a.value.z*b.value.x - a.value.x*b.value.z,
      a.value.x*b.value.y - a.value.y*b.value.x,
    ])
  }
}
```

Key Path Member Lookup

Swift Evolution: SE-0252

```swift
@dynamicMemberLookup
struct GeometricVector<Storage: SIMD> where Storage.Scalar: FloatingPoint {
  var value: Storage

  init(_ value: Value) { self.value = value }

  subscript(dynamicMember keyPath: KeyPath<Storage, Scalar>) -> Scalar {
    value[keyPath: keyPath]
  }
}
```

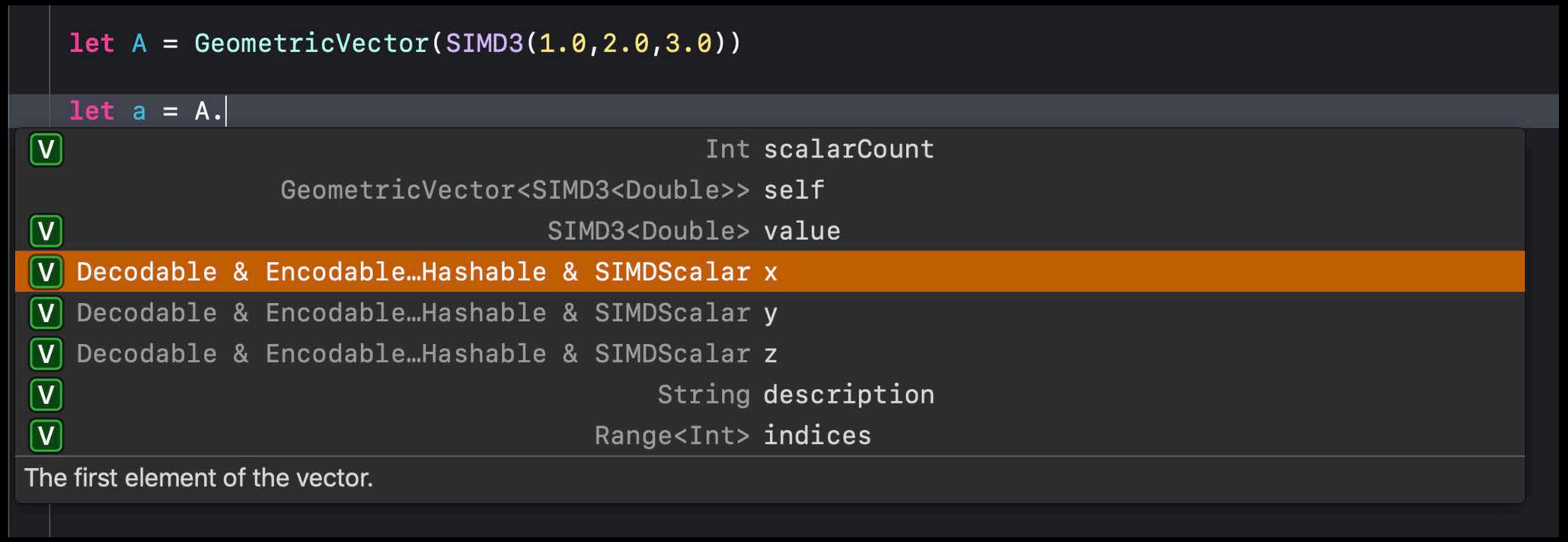
```
@dynamicMemberLookup
struct GeometricVector<Storage: SIMD> where Storage.Scalar: FloatingPoint {
  var value: Storage

  init(_ value: Value) { self.value = value }

  subscript(dynamicMember keyPath: KeyPath<Storage, Scalar>) -> Scalar {
    value[keyPath: keyPath]
  }
}
```

```swift
@dynamicMemberLookup
struct GeometricVector<Storage: SIMD> where Storage.Scalar: FloatingPoint {
  var value: Storage

  init(_ value: Value) { self.value = value }

  subscript(dynamicMember keyPath: KeyPath<Storage, Scalar>) -> Scalar {
    value[keyPath: keyPath]
  }
}
```

```swift
@dynamicMemberLookup
struct GeometricVector<Storage: SIMD> where Storage.Scalar: FloatingPoint {
  var value: Storage

  init(_ value: Value) { self.value = value }

  subscript(dynamicMember keyPath: KeyPath<Storage, Scalar>) -> Scalar {
    value[keyPath: keyPath]
  }
}
```

```
@dynamicMemberLookup
struct GeometricVector<Storage: SIMD> where Storage.Scalar: FloatingPoint {
  var value: Storage

  init(_ value: Value) { self.value = value }

  subscript(dynamicMember keyPath: KeyPath<Storage, Scalar>) -> Scalar {
    value[keyPath: keyPath]
  }
}
```

SIMD Properties Now Available on GeometricVector



```swift
extension GeometricVector {
  /// Cross product for two vectors
  static func ✕ <T>(_ a: Self, _ b: Self) -> Self where Storage == SIMD3<T> {
    Self([
      a.value.y*b.value.z - a.value.z*b.value.y,
      a.value.z*b.value.x - a.value.x*b.value.z,
      a.value.x*b.value.y - a.value.y*b.value.x,
    ])
  }
}
```

```swift
extension GeometricVector {
  /// Cross product for two vectors
  static func ✕ <T>(_ a: Self, _ b: Self) -> Self where Storage == SIMD3<T> {
    Self([
      a.y*b.z - a.z*b.y,
      a.z*b.x - a.x*b.z,
      a.x*b.y - a.y*b.x,
    ])
  }
}
```

```
struct Material {
  public var roughness: Float
  public var color: Color

  private var _texture: Texture

  public var isSparkly: Bool {
    get { _texture.isSparkly }
    set {
      if !isUniquelyReferenced(&_texture) { _texture = Texture(copying: _texture)  }
      _texture.isSparkly = newValue
    }
  }
}
```

```
struct Material {
  public var roughness: Float
  public var color: Color

  private var _texture: Texture









}
```

```
@dynamicMemberLookup
struct Material {
  public var roughness: Float
  public var color: Color

  private var _texture: Texture



}
```

```
@dynamicMemberLookup
struct Material {
  public var roughness: Float
  public var color: Color

  private var _texture: Texture

  public subscript<T>(dynamicMember keyPath: ReferenceWritableKeyPath<Texture, T>) -> T {




  }
}
```

```
@dynamicMemberLookup
struct Material {
  public var roughness: Float
  public var color: Color

  private var _texture: Texture

  public subscript<T>(dynamicMember keyPath: ReferenceWritableKeyPath<Texture, T>) -> T {




  }
}
```

```
@dynamicMemberLookup
struct Material {
  public var roughness: Float
  public var color: Color

  private var _texture: Texture

  public subscript<T>(dynamicMember keyPath: ReferenceWritableKeyPath<Texture, T>) -> T {




  }
}
```

```
@dynamicMemberLookup
struct Material {
  public var roughness: Float
  public var color: Color

  private var _texture: Texture

  public subscript<T>(dynamicMember keyPath: ReferenceWritableKeyPath<Texture, T>) -> T {
    get { _texture[keyPath: keyPath] }
    set {
      if !isKnownUniquelyReferenced(&_texture) { _texture = Texture(copying: _texture)  }
      _texture[keyPath: keyPath] = newValue
    }
  }
}
```

```
@dynamicMemberLookup
struct Material {
  public var roughness: Float
  public var color: Color

  private var _texture: Texture

  public subscript<T>(dynamicMember keyPath: ReferenceWritableKeyPath<Texture, T>) -> T {
    get { _texture[keyPath: keyPath] }
    set {
      if !isKnownUniquelyReferenced(&_texture) { _texture = Texture(copying: _texture)  }
      _texture[keyPath: keyPath] = newValue
    }
  }
}
```

Property Wrappers

Swift Evolution: SE-0258

Doug Gregor, Swift Team

```
// Wrapping a Stored Property

public struct MyType {
  var imageStorage: UIImage? = nil

  public var image: UIImage {
    mutating get {
      if imageStorage == nil {
        imageStorage = loadDefaultImage()
      }
      return imageStorage!
    }
    set {
      imageStorage = newValue
    }
  }
}
```

```
// Wrapping a Stored Property

public struct MyType {
  var imageStorage: UIImage? = nil

  public var image: UIImage {
    mutating get {
      if imageStorage == nil {
        imageStorage = loadDefaultImage()
      }
      return imageStorage!
    }
    set {
      imageStorage = newValue
    }
  }
}
```

```
// Wrapping a Stored Property

public struct MyType {
  var imageStorage: UIImage? = nil

  public var image: UIImage {
    mutating get {
      if imageStorage == nil {
        imageStorage = loadDefaultImage()
      }
      return imageStorage!
    }
    set {
      imageStorage = newValue
    }
  }
}
```

```
// Wrapping a Stored Property

public struct MyType {
  var imageStorage: UIImage? = nil

  public var image: UIImage {
    mutating get {
      if imageStorage == nil {
        imageStorage = loadDefaultImage()
      }
      return imageStorage!
    }
    set {
      imageStorage = newValue
    }
  }
}
```

```
// Wrapping a Stored Property

public struct MyType {


  public lazy var image: UIImage = loadDefaultImage()









}
```

```
// Wrapping Another Stored Property

public struct MyType {
  var textStorage: String? = nil

  public var text: String {
    get {
      guard let value = textStorage else {
        fatalError("text has not yet been set!")
      }
      return value
    }
    set {
      textStorage = newValue
    }
  }
}
```

Property Wrappers

Capture backing storage property and access policy for re-use

```
public struct MyType {
  @LateInitialized public var text: String
}
```

Property Wrappers

Capture backing storage property and access policy for re-use

```
public struct MyType {
  @LateInitialized public var text: String
}
```

Provides similar benefits to the built-in `lazy`

- Eliminates boilerplate
- Documents semantics at the point of definition

```
// Implementing a Property Wrapper
@propertyWrapper
public struct LateInitialized<Value> {
  private var storage: Value?

  public init() {
    storage = nil
  }

  public var value: Value {
    get {
      guard let value = storage else {
        fatalError("value has not yet been set!")
      }
      return value
    }
    set {
      storage = newValue
    }
  }
```

```
// Implementing a Property Wrapper
@propertyWrapper
public struct LateInitialized<Value> {
  private var storage: Value?

  public init() {
    storage = nil
  }

  public var value: Value {
    get {
      guard let value = storage else {
        fatalError("value has not yet been set!")
      }
      return value
    }
    set {
      storage = newValue
    }
  }
```

```swift
// Implementing a Property Wrapper
@propertyWrapper
public struct LateInitialized<Value> {
  private var storage: Value?

  public init() {
    storage = nil
  }

  public var value: Value {
    get {
      guard let value = storage else {
        fatalError("value has not yet been set!")
      }
      return value
    }
    set {
      storage = newValue
    }
  }
```

```
// Implementing a Property Wrapper
@propertyWrapper
public struct LateInitialized<Value> {
  private var storage: Value?

  public init() {
    storage = nil
  }

  public var value: Value {
    get {
      guard let value = storage else {
        fatalError("value has not yet been set!")
      }
      return value
    }
    set {
      storage = newValue
    }
  }
```

```swift
@propertyWrapper
public struct LateInitialized<Value> {
  private var storage: Value?

  public init() {
    storage = nil
  }

  public var value: Value {
    get {
      guard let value = storage else {
        fatalError("value has not yet been set!")
      }
      return value
    }
    set {
      storage = newValue
    }
  }
}
```

Using Property Wrappers

Uses of property wrappers expand into a stored property and a computed property

```
public struct MyType {
  @LateInitialized public var text: String
}
```

Using Property Wrappers

Uses of property wrappers expand into a stored property and a computed property

```
public struct MyType {
  @LateInitialized public var text: String

  // Compiler-synthesized code…
  var $text: LateInitialized<String> = LateInitialized<String>()

  public var text: String {
    get { $text.value }

    set { $text.value = newValue }
  }
}
```

Using Property Wrappers

Uses of property wrappers expand into a stored property and a computed property

```
public struct MyType {
  @LateInitialized public var text: String

  // Compiler-synthesized code…
  var $text: LateInitialized<String> = LateInitialized<String>()

  public var text: String {
    get { $text.value }

    set { $text.value = newValue }
  }
}
```

Using Property Wrappers

Uses of property wrappers expand into a stored property and a computed property

```
public struct MyType {
  @LateInitialized public var text: String

  // Compiler-synthesized code…
  var $text: LateInitialized<String> = LateInitialized<String>()

  public var text: String {
    get { $text.value }

    set { $text.value = newValue }
  }
}
```

Using Property Wrappers

Uses of property wrappers expand into a stored property and a computed property

```
public struct MyType {
  @LateInitialized public var text: String

  // Compiler-synthesized code…
  var $text: LateInitialized<String> = LateInitialized<String>()

  public var text: String {
    get { $text.value }

    set { $text.value = newValue }
  }
}
```

Using Property Wrappers

Uses of property wrappers expand into a stored property and a computed property

```
public struct MyType {
  @LateInitialized public var text: String

  // Compiler-synthesized code…
  var $text: LateInitialized<String> = LateInitialized<String>()

  public var text: String {
    get { $text.value }

    set { $text.value = newValue }
  }
}
```

```swift
// Defensive Copying

@propertyWrapper
public struct DefensiveCopying<Value: NSCopying> {
  private var storage: Value

  public init(initialValue value: Value) {
    storage = value.copy() as! Value
  }

  public var value: Value {
    get { storage }
    set {
      storage = newValue.copy() as! Value
    }
  }
}
```

```
// Defensive Copying

@propertyWrapper
public struct DefensiveCopying<Value: NSCopying> {
  private var storage: Value

  public init(initialValue value: Value) {
    storage = value.copy() as! Value
  }

  public var value: Value {
    get { storage }
    set {
      storage = newValue.copy() as! Value
    }
  }
}
```

```
// Defensive Copying

@propertyWrapper
public struct DefensiveCopying<Value: NSCopying> {
  private var storage: Value

  public init(initialValue value: Value) {
    storage = value.copy() as! Value
  }

  public var value: Value {
    get { storage }
    set {
      storage = newValue.copy() as! Value
    }
  }
}
```

```swift
// Defensive Copying

@propertyWrapper
public struct DefensiveCopying<Value: NSCopying> {
  private var storage: Value

  public init(initialValue value: Value) {
    storage = value.copy() as! Value
  }

  public var value: Value {
    get { storage }
    set {
      storage = newValue.copy() as! Value
    }
  }
}
```

Using the `DefensiveCopying` Property Wrapper

`@DefensiveCopying` variables can be initialized in their declaration:

```
public struct MyType {
  @DefensiveCopying public var path: UIBezierPath = UIBezierPath()
}
```

Using the `DefensiveCopying` Property Wrapper

`@DefensiveCopying` variables can be initialized in their declaration:

```
public struct MyType {
  @DefensiveCopying public var path: UIBezierPath = UIBezierPath()
}
```

Using the `DefensiveCopying` Property Wrapper

`@DefensiveCopying` variables can be initialized in their declaration:

```
public struct MyType {
  @DefensiveCopying public var path: UIBezierPath = UIBezierPath()

  // Compiler-synthesized code…
  var $path: DefensiveCopying<UIBezierPath> =
      DefensiveCopying(initialValue: UIBezierPath())

  public var path: UIBezierPath {
    get { $path.value }

    set { $path.value = newValue }
  }
}
```

Using the `DefensiveCopying` Property Wrapper

`@DefensiveCopying` variables can be initialized in their declaration:

```
public struct MyType {
  @DefensiveCopying public var path: UIBezierPath = UIBezierPath()

  // Compiler-synthesized code…
  var $path: DefensiveCopying<UIBezierPath> =
      DefensiveCopying(initialValue: UIBezierPath())

  public var path: UIBezierPath {
    get { $path.value }

    set { $path.value = newValue }
  }
}
```

Using the `DefensiveCopying` Property Wrapper

```
extension DefensiveCopying {
  public init(withoutCopying value: Value) {
    storage = value
  }
}
```

Using the `DefensiveCopying` Property Wrapper

```
extension DefensiveCopying {
  public init(withoutCopying value: Value) {
    storage = value
  }
}
```

Initializing the backing storage property:

```
public struct MyType {
  @DefensiveCopying public var path: UIBezierPath

  public init() {
    $path = DefensiveCopying(withoutCopying: UIBezierPath())
  }
}
```

Using the `DefensiveCopying` Property Wrapper

```
extension DefensiveCopying {
  public init(withoutCopying value: Value) {
    storage = value
  }
}
```

Initializing the backing storage property:

```
public struct MyType {
  @DefensiveCopying(withoutCopying: UIBezierPath())
  public var path: UIBezierPath
}
```

API Design with Property Wrappers

Property wrappers describe the policy behind your data access

Lots of API revolves around data access

API Design with Property Wrappers

Property wrappers describe the policy behind your data access

Lots of API revolves around data access

```
@UserDefault(key: "BOOSTER_IGNITED", defaultValue: false)
static var isBoosterIgnited: Bool
```

API Design with Property Wrappers

Property wrappers describe the policy behind your data access

Lots of API revolves around data access

```
@UserDefault(key: "BOOSTER_IGNITED", defaultValue: false)
static var isBoosterIgnited: Bool

@ThreadSpecific var localPool: MemoryPool
```

API Design with Property Wrappers

Property wrappers describe the policy behind your data access

Lots of API revolves around data access

```
@UserDefault(key: "BOOSTER_IGNITED", defaultValue: false)
static var isBoosterIgnited: Bool

@ThreadSpecific var localPool: MemoryPool

@Option(shorthand: "m", documentation: "Minimum value", defaultValue: 0)
var minimum: Int
```

Property Wrappers in SwiftUI

View data dependencies are expressed using property wrappers

```
struct SlideViewer: View {
  @State private var isEditing = false
  @Binding var slide: Slide









}
```

Property Wrappers in SwiftUI

View data dependencies are expressed using property wrappers

```
struct SlideViewer: View {
  @State private var isEditing = false
  @Binding var slide: Slide

  var body: some View {
    VStack {
      Text("Slide #\(slide.number)")
      if isEditing {
        TextField($slide.title)
      }
      // …
    }
  }
}
```

Property Wrappers in SwiftUI

View data dependencies are expressed using property wrappers

```
struct SlideViewer: View {
  @State private var isEditing = false
  @Binding var slide: Slide

  var body: some View {
    VStack {
      Text("Slide #\(slide.number)")
      if isEditing {
        TextField($slide.title)
      }
      // …
    }
  }
}
```

```
// Property Wrappers & Key Path Member Lookup

@propertyWrapper
public struct Binding<Value> {
  public var value: Value {
    get { … }
    nonmutating set { … }
  }



}
```

```swift
// Property Wrappers & Key Path Member Lookup

@propertyWrapper @dynamicMemberLookup
public struct Binding<Value> {
  public var value: Value {
    get { … }
    nonmutating set { … }
  }

  public subscript<Property>(dynamicMember keyPath: WritableKeyPath<Value, Property>)
      -> Binding<Property> { … }
}
```

```
// Property Wrappers & Key Path Member Lookup

@propertyWrapper @dynamicMemberLookup
public struct Binding<Value> {
  public var value: Value {
    get { … }
    nonmutating set { … }
  }

  public subscript<Property>(dynamicMember keyPath: WritableKeyPath<Value, Property>)
      -> Binding<Property> { … }
}
```

```
// Property Wrappers & Key Path Member Lookup

@propertyWrapper @dynamicMemberLookup
public struct Binding<Value> {
  public var value: Value {
    get { … }
    nonmutating set { … }
  }

  public subscript<Property>(dynamicMember keyPath: WritableKeyPath<Value, Property>)
      -> Binding<Property> { … }
}
```

```
// Property Wrappers & Key Path Member Lookup

@propertyWrapper @dynamicMemberLookup
public struct Binding<Value> {
  public var value: Value {
    get { … }
    nonmutating set { … }
  }

  public subscript<Property>(dynamicMember keyPath: WritableKeyPath<Value, Property>)
      -> Binding<Property> { … }
}

@Binding var slide: Slide

slide       // Slide instance
slide.title // String instance
```

```
// Property Wrappers & Key Path Member Lookup

@propertyWrapper @dynamicMemberLookup
public struct Binding<Value> {
  public var value: Value {
    get { … }
    nonmutating set { … }
  }

  public subscript<Property>(dynamicMember keyPath: WritableKeyPath<Value, Property>)
      -> Binding<Property> { … }
}

@Binding var slide: Slide

slide       // Slide instance       $slide       // Binding<Slide> instance
slide.title // String instance
```

```
// Property Wrappers & Key Path Member Lookup

@propertyWrapper @dynamicMemberLookup
public struct Binding<Value> {
  public var value: Value {
    get { … }
    nonmutating set { … }
  }

  public subscript<Property>(dynamicMember keyPath: WritableKeyPath<Value, Property>)
      -> Binding<Property> { … }
}

@Binding var slide: Slide

slide       // Slide instance       $slide       // Binding<Slide> instance
slide.title // String instance      $slide.title
```

```
// Property Wrappers & Key Path Member Lookup

@propertyWrapper @dynamicMemberLookup
public struct Binding<Value> {
  public var value: Value {
    get { … }
    nonmutating set { … }
  }

  public subscript<Property>(dynamicMember keyPath: WritableKeyPath<Value, Property>)
      -> Binding<Property> { … }
}

@Binding var slide: Slide

slide       // Slide instance       $slide       // Binding<Slide> instance
slide.title // String instance      $slide[dynamicMember: \Slide.title]
```

```
// Property Wrappers & Key Path Member Lookup

@propertyWrapper @dynamicMemberLookup
public struct Binding<Value> {
  public var value: Value {
    get { … }
    nonmutating set { … }
  }

  public subscript<Property>(dynamicMember keyPath: WritableKeyPath<Value, Property>)
      -> Binding<Property> { … }
}

@Binding var slide: Slide

slide       // Slide instance       $slide       // Binding<Slide> instance
slide.title // String instance      $slide.title // Binding<String> instance
```

```
// Property Wrappers & Key Path Member Lookup

@propertyWrapper @dynamicMemberLookup
public struct Binding<Value> {
  public var value: Value {
    get { … }
    nonmutating set { … }
  }

  public subscript<Property>(dynamicMember keyPath: WritableKeyPath<Value, Property>)
      -> Binding<Property> { … }
}

@Binding var slide: Slide

slide       // Slide instance       $slide       // Binding<Slide> instance
slide.title // String instance      $slide.title // Binding<String> instance
```

Summary

Value semantics vs. reference semantics

Use protocols for code reuse — not classification

Property wrappers reuse computed property definitions

More Information

developer.apple.com/wwdc19/415

Swift Open Hours	Thursday, 3:00
Swift Open Hours	Friday, 9:00


WWDC19