IMMUTABILITY

PERKS AND QUIRKS

JON-TAIT BEASON

@BUGKRUSHA

GLOWFORGE

WHAT DOES THIS FUNCTION DO?

```swift
func createThumbnail(path: UIBezierPath) -> UIImage {
    // ...
}
```

WORLD OF REFERENCES

```swift
var path = UIBezierPath() /// Shared
func createThumbnail(path: UIBezierPath) -> UIImage {
    // ...
}

func apply(scale: CGFloat to path: UIBezierPath) {
  /// Is this really your path?
  path.apply(CGAffineTransform(scaleX: 0.25, y: 0.25))
}
```

MAKING THE PARAMETER IMMUTABLE

```
// Gets an independent copy of `path`
func createThumbnail(path: CGPath) -> UIImage {
    // ...
}
```

REFERENCE TYPES CAN BE CONVENIENT

COMMON MUTABLE DATA STRUCTURE

```
class Person {
    var name: String
    var age: Int

    init(name: String, age: Int) {
        self.name = name
        self.age = age
    }
}

let jazbo = Person(name: "Jazbo", age: ha)
```

SWIFT & SAFETY

VALUE TYPES: OVER 90% OF TYPES IN STANDARD LIBRARY

- ENUMS
- STRUCTS
- COLLECTIONS

PURE VALUE TYPES

```swift
enum TransmissionType {
    case standard
    case automatic
}

struct Transmission {
    let type: TransmissionType
}

struct Car {
    let color: RGBA
    let transmission: Transmission

    /// ....
}
```

PURE VALUE TYPES

```
struct Car {
    let color: RGBA
    let transmission: Transmission

    /// ....
}

let orange = RGBA(red: 240, green: 83, blue: 5, alpha: 0.91)

let red = RGBA(red: 194, green: 0, blue: 0, alpha: 1)
let car = Car(color: orange, engine: Engine(type: .standard))

car.color = red // Not allowed
car.engine.type = .automatic // Not allowed
```

MIXING TYPES

```swift
struct SvgPathElement {
    let path: UIBezierPath
}

let rect = CGRect(x: 23, y: 45, width: 13, height: 24)
let element = SvgPathElement(path: UIBezierPath(roundedRect: rect, cornerRadius: 4))
let elementTwo = element

element.path = UIBezierPath(ovalIn: rect) // Not allowed

let timbuk: CGFloat = 2.0
let tu: CGFloat = 4.0
element.path.addLine(to: CGPoint(x: timbuk, y: tu)) // Allowed
```

IMMUTABILITY: PERKS

1. NO REFERENCING
2. SAFER AND EASIER TO UNDERSTAND

REFERENCING: THE ROOT OF ALL EVIL

1. `self`
2. A GLOBAL VARIABLE ACCESSIBLE FROM A FUNCTION
3. A PARAMETER PASSED IN TO A FUNCTION
4. A PARAMETER RETURNED FROM A FUNCTION
5. A LOCAL VARIABLE IN A FUNCTION BOUND TO ANY OF THE ABOVE

REFERENCE TYPES AND ROLES: MATRIX

$$\mathbf{a} = \begin{bmatrix} 1 & 8 \\ 4 & 5 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} 3 & 9 \\ 2 & 6 \end{bmatrix}$$

REFERENCE TYPES AND ROLES: MATRIX

```
class Matrix<T>  {
    var backing: [[T]]

    init(backing: [[T]]) {
        self.backing = backing
    }
}


func multiply(a: Matrix<Int>, b: Matrix<Int>, into c: Matrix<Int>) {
    ///...
}
```

REFERENCE TYPES AND ROLES: MATRIX

HERE IS HOW MULTIPLICATION WORKS...

$$\overset{a}{\begin{bmatrix} \boxed{1 \quad 8} \\ 4 \quad 5 \end{bmatrix}} \qquad \overset{b}{\begin{bmatrix} \boxed{\begin{matrix} 3 \\ 2 \end{matrix}} & \begin{matrix} 9 \\ 6 \end{matrix} \end{bmatrix}}$$

$$\overset{c}{\begin{bmatrix} 19 & \\ & \end{bmatrix}}$$

REFERENCE TYPES AND ROLES: MATRIX

SOURCE AND DESTINATION

```
func multiply(a: Matrix<Int>, b: Matrix<Int>, into c: Matrix<Int>) {
    ///...
}


let a = Matrix<Int>(backing: [[1, 8], [4, 5]])
let b = Matrix<Int>(backing: [[3, 9], [2, 6]])

let row = Array(repeating: 0, count: 2)
let c = Matrix<Int>(backing: [row, row])

multiply(a: a, b: b, into: a)
```

REFERENCE TYPES AND ROLES: MATRIX

```
multiply(a: a, b: b, into: a)
```

$$\begin{array}{c} a \\ \begin{bmatrix} \boxed{1 \quad 8} \\ 4 \quad 5 \end{bmatrix} \end{array} \qquad \begin{array}{c} b \\ \begin{bmatrix} \boxed{\begin{matrix} 3 \\ 2 \end{matrix}} & \begin{matrix} 9 \\ 6 \end{matrix} \end{bmatrix} \end{array}$$

$$\begin{array}{c} a \\ \begin{bmatrix} 19 & 8 \\ 4 & 5 \end{bmatrix} \end{array}$$

REFERENCE TYPES AND ROLES: MATRIX

WHAT ABOUT...

```
func multiply(a: Matrix<Int>, b: Matrix<Int>) -> Matrix<Int> {
    // …
}
```

REFERENCES & SUBCLASS POLYMORPHISM

```swift
class Person {
    var name: String
    var age: Int

    init(name: String, age: Int) {
        self.name = name
        self.age = age
    }
}

class Tutor: Person {
    override init(name: String, age: Int) {
        super.init(name: name, age: age)
    }
}
```

WHAT IS WRONG HERE?

```
class Department {
    func setTutorFor(person: Person, tutor: Tutor) {
        /// ...
    }
}

let t = Tutor(name: "Tee", age: 45)
department.setTutorFor(person: t, tutor: t)
```

IMMUTABILITY

SAFE, EASIER TO UNDERSTAND AND CHANGE

PASSING REFERENCE TYPES

```
class Stack<T: Comparable> {
    var list = [T]()

    var count: Int {
        return list.count
    }

    /// Add a new object to the stack.
    ///
    /// - Parameter value: Object to be added.
    func push(value: T) {
        list.append(value)
    }

    /// Remove and return the last object added.
    ///
    /// - Returns: Last object added or nil if list is empty.
    func pop() -> T? {
        return list.popLast()
    }
}
```

PASSING REFERENCE TYPES

```
let homePricesStack = Stack<Int>()
homePricesStack.push(value: 760000)
homePricesStack.push(value: 850000)
homePricesStack.push(value: 575000)
```

PASSING REFERENCE TYPES

```
func findMedian(stack: Stack<Int>) -> Int {
    stack.list.sort()

    let index = (stack.count - 1) / 2
    if stack.count % 2 == 0 {
        return (stack.list[index] + stack.list[index + 1]) / 2
    }

    return stack.list[index]
}
```

PASSING REFERENCE TYPES

```
homePricesStack.push(value: 760000)
homePricesStack.push(value: 850000)
homePricesStack.push(value: 575000)

func findMedian(stack: Stack<Int>) -> Int {
    stack.list.sort()

    let index = (stack.count - 1) / 2
    if stack.count % 2 == 0 {
        return (stack.list[index] + stack.list[index + 1]) / 2
    }

    return stack.list[index]
}

findMedian(stack: homePricesStack)
homePricesStack.pop()
```

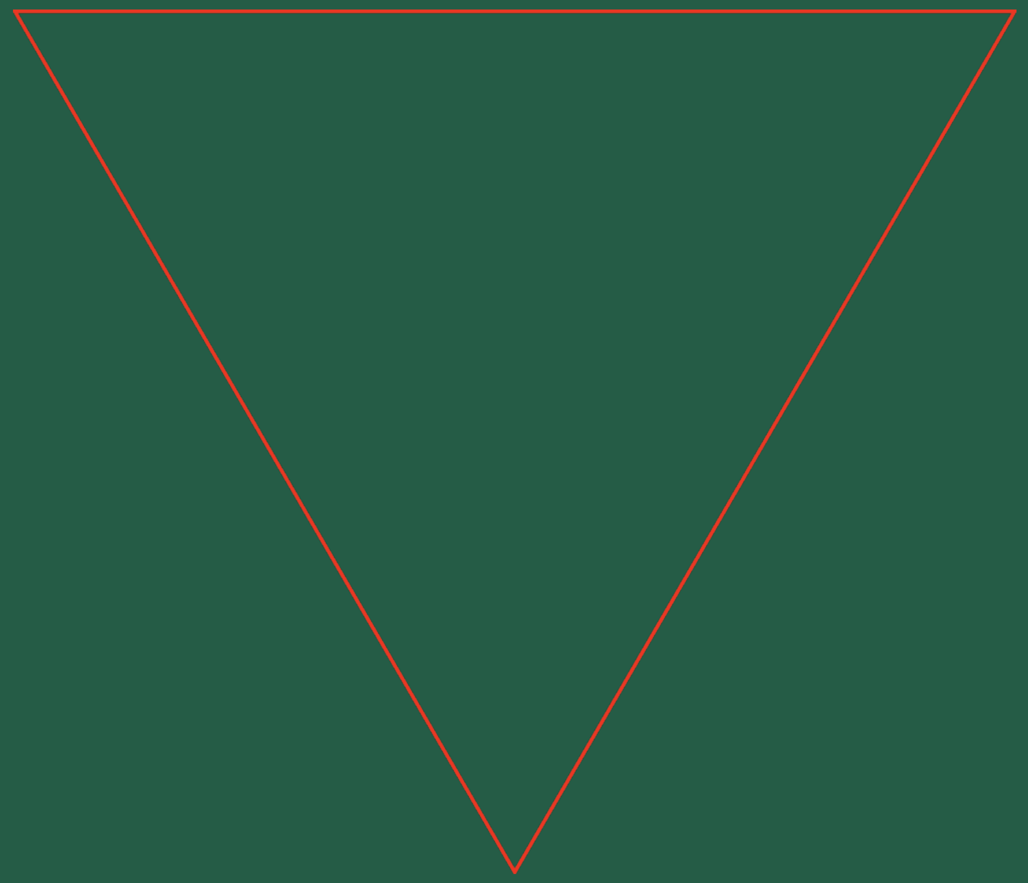
PASSING REFERENCE TYPES

LATENT BUGS

JUST WAITING TO HAPPEN

RETURNING REFERENCE TYPES

```
<?xml version="1.0" encoding="utf-8"?>
<svg version="1.1">
    <path d="M5,5h260L135,230"/>
</svg>
```



RETURNING REFERENCE TYPES

```swift
/// Converting the dpath from an svg file into a bezier path.
///
/// - Parameter dpath: dpath from svg file.
/// - Returns: Bezier path.
private func convertToBezierPath(dpath: String) -> UIBezierPath {
    let path = UIBezierPath()

    /// ....

    return path
}

func elementBezier() -> UIBezierPath {
    return convertToBezierPath(dpath: "M150 0 L75 200 L225 200 Z")
}
```

RETURNING REFERENCE TYPES

```swift
/// Accessible to functions in class
var path = UIBezierPath()

func elementBezier() -> UIBezierPath {
    if path.isEmpty {
        path = convertToBezierPath(dpath: "M150 0 L75 200 L225 200 Z")
    }

    return path
}
```

RETURNING REFERENCE TYPES

```swift
// Accessible to functions in class
var path = UIBezierPath()
func elementBezier() -> UIBezierPath {
    if path.isEmpty {
        path = convertToBezierPath(dpath: "M150 0 L75 200 L225 200 Z")
    }

    return path
}


func createThumbnail() -> UIImage {
    let path = elementBezier()
    path.apply(CGAffineTransform(scaleX: 0.25, y: 0.25))

    /// ...
}
```

IMMUTABLE CODE CAN BE TRUSTED

FOR YOUR REFERENCE

OBJECTS ARE DECLARED WITH VARIABLE NAMES DESCRIBING THEIR ROLE BUT ARE MANIPULATED BASED ON THEIR IDENTITIES

SO SHOULD WE DEFAULT TO VALUE TYPES?

NO.

DEMO

YO IT'S A ~~BUG~~ FEATURE!

THOUGHTFULLY CHOOSING VALUE TYPES

ENSURE YOUR SYSTEMS ARE SAFE, WELCOMING TO CHANGE, NEW ENGINEERS AND YOUR FUTURE SELVES

@BUGKRUSHA