## Swift Best Practices

#### Naming

As per the “[Swift Programming Language](https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/index.html)” type names should be [upper camel case](https://en.wikipedia.org/wiki/Studly_caps) (example: “VideoPlayerViewController”).

Variables and constants should be lower camel case (example “videoPlayer”).

Do not use any form of [Hungarian notation](https://en.wikipedia.org/wiki/Hungarian_notation) (e.g. k for constants, m for methods), instead use short concise names and use Xcode's type Quick Help (⌥ + click) to discover a variable's type. Similarly do not use [SNAKE\_CASE](https://en.wikipedia.org/wiki/Snake_case).

It also applies to enum values, which should be lowercase (as defined by "[0006-apply-api-guidelines-to-the-standard-library](https://github.com/apple/swift-evolution/blob/master/proposals/0006-apply-api-guidelines-to-the-standard-library.md)"):

enum Planet {

case mercury, venus, earth, mars, jupiter, saturn, uranus, neptune

}

#### Types

Always use Swift's native types when available. Swift offers bridging to Objective-C so you can still use the full set of methods as needed.

Preferred:

let width = 120.0 // Double

let widthString = (width as NSNumber).stringValue // String

Not Preferred:

let width: NSNumber = 120.0 // NSNumber

let widthString: NSString = width.stringValue // NSString

In Sprite Kit code, use CGFloat if it makes the code more succinct by avoiding too many conversions.

#### Constants

Constants are defined using the let keyword, and variables with the var keyword. Always use let instead of var if the value of the variable will not change.

Tip: A good technique is to define everything using let and only change it to var if the compiler complains!

You can define constants on a type rather than on an instance of that type using type properties. To declare a type property as a constant simply use static let. Type properties declared in this way are generally preferred over global constants because they are easier to distinguish from instance properties. Example:

Preferred:

enum Math {

static let e = 2.718281828459045235360287

static let root2 = 1.41421356237309504880168872

}

let hypotenuse = side \* Math.root2

Note: The advantage of using a case-less enumeration is that it can't accidentally be instantiated and works as a pure namespace.

Not Preferred:

let e = 2.718281828459045235360287 // pollutes global namespace

let root2 = 1.41421356237309504880168872

let hypotenuse = side \* root2 // what is root2?

#### Static Methods and Variable Type Properties

Static methods and type properties work similarly to global functions and global variables and should be used sparingly. They are useful when functionality is scoped to a particular type or when Objective-C interoperability is required.

#### Optionals

Declare variables and function return types as optional with ? where a nil value is acceptable.

Use implicitly unwrapped types declared with ! only for instance variables that you know will be initialized later before use, such as subviews that will be set up in viewDidLoad.

When accessing an optional value, use optional chaining if the value is only accessed once or if there are many optionals in the chain:

self.textContainer?.textLabel?.setNeedsDisplay()

Use optional binding when it's more convenient to unwrap once and perform multiple operations:

if let textContainer = self.textContainer {

// do many things with textContainer

}

When naming optional variables and properties, avoid naming them like optionalString or maybeView since their optional-ness is already in the type declaration.

For optional binding, shadow the original name when appropriate rather than using names like unwrappedView or actualLabel.

Preferred:

var subview: UIView?

var volume: Double?

// later on...

if let subview = subview, let volume = volume {

// do something with unwrapped subview and volume

}

Not Preferred:

var optionalSubview: UIView?

var volume: Double?

if let unwrappedSubview = optionalSubview {

if let realVolume = volume {

// do something with unwrappedSubview and realVolume

}

}

#### Lazy Initialization

Consider using lazy initialization for finer grain control over object lifetime. This is especially true for UIViewController that loads views lazily. You can either use a closure that is immediately called { }() or call a private factory method. Example:

lazy var locationManager: CLLocationManager = self.makeLocationManager()

private func makeLocationManager() -> CLLocationManager {

let manager = CLLocationManager()

manager.desiredAccuracy = kCLLocationAccuracyBest

manager.delegate = self

manager.requestAlwaysAuthorization()

return manager

}

Notes:

* [unowned self] is not required here. A retain cycle is not created.
* Location manager has a side-effect for popping up UI to ask the user for permission so fine grain control makes sense here.

#### Type Inference

Prefer compact code and let the compiler infer the type for constants or variables of single instances. Type inference is also appropriate for small (non-empty) arrays and dictionaries. When required, specify the specific type such as CGFloat or Int16.

Preferred:

let message = "Click the button"

let currentBounds = computeViewBounds()

var names = ["Mic", "Sam", "Christine"]

let maximumWidth: CGFloat = 106.5

Not Preferred:

let message: String = "Click the button"

let currentBounds: CGRect = computeViewBounds()

let names = [String]()

#### Type Annotation for Empty Arrays and Dictionaries

For empty arrays and dictionaries, use type annotation. (For an array or dictionary assigned to a large, multi-line literal, use type annotation.)

Preferred:

var names: [String] = []

var lookup: [String: Int] = [:]

Not Preferred:

var names = [String]()

var lookup = [String: Int]()

NOTE: Following this means picking descriptive names is even more important than before.

#### Syntactic Sugar

Prefer the shortcut versions of type declarations over the full generics syntax.

Preferred:

var deviceModels: [String]

var employees: [Int: String]

var faxNumber: Int?

Not Preferred:

var deviceModels: Array<String>

var employees: Dictionary<Int, String>

var faxNumber: Optional<Int>

#### Control Flow

Prefer the for-in style of for loop

Preferred:

for \_ in 0..<3 {

print("Hello three times")

}

for (index, person) in attendeeList.enumerated() {

print("\(person) is at position #\(index)")

}

for index in stride(from: 0, to: items.count, by: 2) {

print(index)

}

for index in (0...3).reversed() {

print(index)

}

#### Singletons

Singletons are simple in Swift:

class ControversyManager {

static let shared = ControversyManager()

}

The Swift runtime will make sure that the singleton is created and accessed in a thread-safe manner.

Singletons should generally just be accessed via "shared" static property unless you have a compelling reason to name it otherwise. Do not use static functions or global functions to access your singleton.

(Because singletons are so easy in Swift and because consistent naming saves you so much time you will have even more time to complain about how singletons are an anti-pattern and should be avoided at all costs. Your fellow developers will thank you.)

#### Protocol Conformance

When adding protocol conformance to a model, prefer adding a separate extension for the protocol methods. This keeps the related methods grouped together with the protocol and can simplify instructions to add a protocol to a class with its associated methods.

Preferred:

class MyViewController: UIViewController {

// class stuff here

}

// MARK: - UITableViewDataSource

extension MyViewController: UITableViewDataSource {

// table view data source methods

}

// MARK: - UIScrollViewDelegate

extension MyViewController: UIScrollViewDelegate {

// scroll view delegate methods

}

Not Preferred:

class MyViewController: UIViewController, UITableViewDataSource, UIScrollViewDelegate {

// all methods

}

For UIKit view controllers, consider grouping lifecycle, custom accessors, and IBAction in separate class extensions.

Preferred:

class MyViewController: UIViewController {

...

}

// MARK: - Animation Methods

extension MyViewController {

func animateSaveButton() {

...

}

func fanimateCustomActivityIndicator() {

...

}

}

// MARK: - Validation Methods

extension MyViewController {

func validatePersonalInfo() -> Bool {

...

}

func validatePaymentInfo() -> Bool {

...

}

}

If an extension contains a lot of code, consider separating that extension into a separate .swift file.

#### Minimal Imports

Keep imports minimal. For example, don't import UIKit when importing Foundation will suffice.

#### Early Returns & Guards

When possible, use guard statements to handle early returns or other exits (e.g. fatal errors or thrown errors).

Prefer:

guard let safeValue = criticalValue else {

fatalError("criticalValue cannot be nil here")

}

someNecessaryOperation(safeValue)

to:

if let safeValue = criticalValue {

someNecessaryOperation(safeValue)

} else {

fatalError("criticalValue cannot be nil here")

}

or:

if criticalValue == nil {

fatalError("criticalValue cannot be nil here")

}

someNecessaryOperation(criticalValue!)

This flattens code otherwise tucked into an if let block, and keeps early exits near their relevant condition instead of down in an else block.

#### Use of Self

For conciseness, avoid using self since Swift does not require it to access an object's properties or invoke its methods.

Use self only when required by the compiler (in @escaping closures, or in initializers to disambiguate properties from arguments). In other words, if it compiles without self then omit it.

#### Closures

Try using closures as and when possible. Closure expressions are a way to write inline closures in a brief, focused syntax. Closure expressions provide several syntax optimizations for writing closures in a shortened form without loss of clarity or intent.

Closure expression syntax has the following general form:

{ (parameters) -> return type in

    statements

}

Example:

reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in

return s1 > s2

})

Inferring Type From Context:

reversedNames = names.sorted(by: { s1, s2 in return s1 > s2 } )

Implicit Returns from Single-Expression Closures

reversedNames = names.sorted(by: { s1, s2 in s1 > s2 } )

Shorthand Argument Names

reversedNames = names.sorted(by: { $0 > $1 } )

Here, $0 and $1 refer to the closure’s first and second String arguments.

For a detailed explanation, please refer:  <https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Closures.html>

#### CoreData

All fetch requests should be constructed as:

let fetchRequest : NSFetchRequest<Item> = Item.fetchRequest()

The array that holds the result value should be of type [Item] and **not** NSArray

NSFetchRequest is now generic. NSFetchedResultsController is generic too. Therefore, when declaring variables, you have to use a generic declaration, e.g.:

var fetchedResultsController: NSFetchedResultsController<Item>?

## References

<https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html#//apple_ref/doc/uid/TP40014097-CH5-ID309>

<https://github.com/raywenderlich/swift-style-guide>

<https://github.com/schwa/Swift-Community-Best-Practices>

<https://www.toptal.com/swift/tips-and-practices>