Swift Coding Conventions

1. Basic conventions

The English language only. It is important to write your code in English.

*Recommended:*

let value = "value string"

*Not Recommended:*

let valor = "cadena de valor"

1. Brackets

*Recommended:*

func sayHello() {

print("Hello")

}

*Not Recommended:*

func sayHi()

{

print("Hi")

}

This is made simply to save space.

1. Semicolons

Semicolon is not required in SWIFT. It can be used to have many statements in one line, but we don’t recommend doing so. Each line should be a separate statement to keep your code clean and easy to read.

*Recommended:*

let anyString = "This is random string"

*Not Recommended:*

let anyString = "This is random string";

1. Naming

Classes names should start with the Uppercase letter:

class Device {}

//not class device { }

Functions must be opposite:

*Recommended:*

func sayHello() {

print("Hello")

}

*Not Recommended:*

func SayHi() {

print("Hi")

}

1. Let and Var keywords

The word Let means that variable will be *immutable*. The keyword Var is opposite and represents *mutable* variable.

***Hint:*** *Here is*[*Apple Developer Documentation*](https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html)*where you can read more about****Let vs Var***

Keep in mind let helps the compiler with optimization process, so use let whenever it is possible. You can declare all variables with the let keyword and the compiler will tell when you are trying to modify an immutable variable.

1. If Statements and Forced Unwrapping with Optional Binding

Let's assume we have an *Optional*:

var anyString : NSString?

[Optional binding](https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html) helps you to find out if *Optional* stores a value.

*Recommended:*

if let anyString = anyString {

//this code will be executed when anyString contains a value

} else {

//this code will be executed when anyString does NOT contain a value

}

This code is much safer than the next one. It is more stable and less likely to cause any runtime crashes

*Not Recommended:*

if anyString != nil {

//this code will be executed when anyString contains a value

} else {

//this code will be executed when anyString does NOT contain a value

}

1. External parameters

Remember to use External parameters. They make your code much easier to read. For example:

func saveDimensions(height: Int, width: Int)

saveDimensions(3,4)

It might be understandable when you write the code, because you have the hints   
https://s3.amazonaws.com/appsfoundation/assets/swift-code-convention-and-guidelines/hints-in-xcode-swift.pngBut when you read it later it will look like:

saveDimensions(3, 4)

After a while you won’t have any idea what that means. But using the external parameters changes that:

func saveDimensions(height height: Int, width width: Int) …

saveDimensions(height: 3, width: 4)

***Hint:*** *In Swift 1.x adding # symbol to the parameter (#height) makes the internal and external name to be the same. So you don’t have to write the parameter name twice.*

*In Swift 2.0 # symbol is removed, so we have duplicated the names for the external and internal parameters*

Another advice is to use external parameters with different names, like:

func combinedString(head head: String, append tail: String) -> String {

return head.stringByAppendingString(tail)

}

print(combinedString(head: "This is ", append: "full text."))

1. Enumerations

Use Uppercase camel style for enumerations:

enum Color {

case Red

case Green

case Blue

}

1. What to use Class or Structure?

[Classes and Structures](https://developer.apple.com/library/mac/documentation/Swift/Conceptual/Swift_Programming_Language/ClassesAndStructures.html#//apple_ref/doc/uid/TP40014097-CH13-XID_144) in SWIFT have some similarities, but you should know the difference and when to use each one. The Structure is a Value Type. That means that structures are copied each time you pass it. The Class instead has reference counting, which provides many references to a class instance, because it is a Reference Type.

**If you don’t need to use Class functionality, then you should use Structure instead.** Changing of data may cause some bugs. Value types are more straightforward to use, when it comes to changing data and using it on multiple threads.

Structure data modification example:

struct Device {

var version: Float = 1.2

}

var a = Device()

var b = a

a.version = 2.0

print("a.version = \(a.version) b.version \(b.version)")

//Result a.version = 2.0 b.version 1.2

Class data modification example:

class Device {

var version: Float = 1.2

}

var a = Device()

var b = a

a.version = 2.0

print("a.version = \(a.version), b.version \(b.version)")

//Result : a.version = 2.0, b.version 2.0

Try to use structures over classes whenever it is possible.

1. Self

Avoid using the word self because it is not required in SWIFT for object properties or methods.

*Recommended:*

class Device {

var version : Float

init() {

version = 1.0

}

func incrementVersion() {

version++

}

func publishVersion() {

incrementVersion()

print("published")

}

}

*Not Recommended:*

class Device {

var version : Float

init() {

self.version = 1.0

}

func incrementVersion() {

self.version++

}

func publishVersion() {

self.incrementVersion()

print("published")

}

}

Use self when an argument name conflicts with a property one:

extension Device {

func changeVersion (version : Float) {

self.version = version

}

}

1. Code grouping

A good manner of coding is to group a code into the logical parts. The //MARK*:* in SWIFT is equivalent to the #pragma mark in Objective-C. [Read more about *MARK* in Objective-C.](https://www.appsfoundation.com/post/objective-c-code-convention-1)

Example of well commented code with logical parts:

*Recommended:*

// MARK: - Lifecycle

override func viewDidLoad() {

super.viewDidLoad()

}

override func didReceiveMemoryWarning() {

super.didReceiveMemoryWarning()

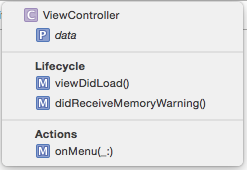
}

// MARK: - Actions

@IBAction func onMenu(sender: AnyObject) {

print("Open Menu")

}

In this case the drop down menu of the Source Editor is logically sorted and easy to understand. It saves your time and code is easy even for other developer:  


*Not Recommended:*

override func viewDidLoad() {

super.viewDidLoad()

}

override func didReceiveMemoryWarning() {

super.didReceiveMemoryWarning()

}

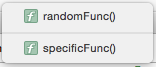
@IBAction func onMenu(sender: AnyObject) {

print("Open Menu")

}

As a result the drop down menu is hard to read and understand:  


There are different ways to use // MARK:

* // MARK:  
  Adds horizontal line to the drop down box of the source code editor  
  
* // MARK: some text  
  Adds “some text” to the drop down box of the source code editor  
  
* // MARK: - some text  
  Adds “some text” and horizontal line to the drop down box of the source code editor  
  

Remember to use // MARK*:* when you have protocol conformance methods

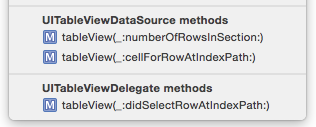
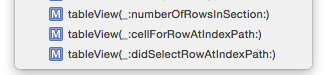
// MARK: - UITableViewDataSource methods

func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {

return 2

}

*Recommended:*

*  
Not Recommended:  
*