Coding Standards – IOS

Table of Contents

Change History

Date	Ver	Change Description	Prepared By	Reviewed By	Approved By
26-08-16	1.0	First Release	ArunJoseph	Kumaran	Harihara G
07-02-2017	1.1	Updated the document ID	Deepa	Deepa	Harihara.G
01-02-19	1.2	Updated the document ID	Document Controller	MR	CISO

1. Correctness

Consider warnings to be errors. This rule informs many stylistic decisions such as not to use the ++ or --operators, C-style for loops, or strings as selectors.

2. Naming

Use descriptive names with camel case for classes, methods, variables, etc. Type names (classes, structures, enumerations and protocols) should be capitalized, while method names and variables should start with a lower case letter.

Preferred

```
privatelet maximumWidgetCount =100
class WidgetContainer {
        var widgetButton: UIButton   let widgetHeightPercentage =0.85
}
```

Not Preferred

```
let MAX_WIDGET_COUNT =100
class app_widgetContainer {
        var wBut: UIButton
        let wHeightPct =0.85
}
```

Abbreviations and acronyms should generally be avoided. Following the API Design Guidelines, abbreviations and initialisms that appear in all uppercase should be uniformly uppercase or lowercase. Examples:

Preferred

```
let urlString: URLString let userID: UserID
```

Not Preferred

```
let uRLString: UrlString let userId: UserId
```

For functions and init methods, prefer named parameters for all arguments unless the context is very clear. Include external parameter names if it makes function calls more readable.

```
funcdateFromString(dateString: String) -> NSDate
funcconvertPointAt(columncolumn: Int, row: Int) -> CGPoint
functimedAction(afterDelaydelay: NSTimeInterval, performaction: SKAction) -> SKAction!
// would be called like this: dateFromString("2014-03-
14") convertPointAt(column: 42, row: 13)
timedAction(afterDelay: 1.0, perform: someOtherAction)
```

For methods, follow the standard Apple convention of referring to the first parameter in the method name:

```
class Counter {
        funccombineWith(otherCounter: Counter, options: Dictionary?) { ... }
        funcincrementBy(amount: Int) { ... }
}
```

2.1 Protocols

Following Apple's API Design Guidelines, protocols names that describe what something is should be a noun.

Examples: Collection, WidgetFactory. Protocols names that describe an ability should end in -ing, -able, or -ible. Examples: Equatable, Resizing.

2.2 Enumerations

Following Apple's API Design Guidelines for Swift 3, use lowerCamelCase for enumeration values.

```
enum Shape {
        case rectangle
        case square
        case rightTriangle
        case equilateralTriangle
}
```

2.3 Prose

When referring to functions in prose (tutorials, books, comments) include the required parameter names from the caller's perspective or _ for unnamed parameters.

Examples:

Call convertPointAt(column:row:) from your own init implementation.

If you call dateFromString(_:) make sure that you provide a string with the format "yyyy-MM-dd".

If you call timedAction(afterDelay:perform:) from viewDidLoad() remember to provide an adjusted delay value and an action to perform.

You shouldn't call the data source method tableView(_:cellForRowAtIndexPath:) directly.

This is the same as the #selector syntax. When in doubt, look at how Xcode lists the method in the jump bar – our style here matches that.

2.4 Class Prefixes

Swift types are automatically namespaced by the module that contains them and you should not add a class prefix such as RW. If two names from different modules collide you can disambiguate by prefixing the type name with the module name. However, only specify the module name when there is possibility for confusion which should be rare.

```
importSomeModule

let myClass = MyModule.UsefulClass()
```

2.5 Selectors

Selectors are Obj-C methods that act as handlers for many Cocoa and Cocoa Touch APIs. Prior to Swift 2.2, they were specified using type unsafe strings. This now causes a compiler warning. The "Fix it" button replaces these strings with the **fully qualified** type safe selector. Often, however, you can use context to shorten the expression. This is the preferred style.

Preferred

```
let sel = #selector(viewDidLoad)
```

Not Preferred

```
let sel = #selector(ViewController.viewDidLoad)
```

2.6 Generics

Generic type parameters should be descriptive, upper camel case names. When a type name doesn't have a meaningful relationship or role, use a traditional single uppercase letter such as T, U, or V.

Preferred

```
struct Stack<Element> { ... }
funcwriteTo<Target: OutputStream>(inouttarget: Target)
funcmax<T: Comparable>(x: T, _y: T) -> T
```

Not Preferred

```
struct Stack<T> { ... }
funcwriteTo<target: OutputStream>(inoutt: target)
funcmax<Thing: Comparable>(x: Thing, _y: Thing) -> Thing
```

2.7 Language

Use US English spelling to match Apple's API.

Preferred

```
let color ="red"
```

Not Preferred

```
let colour ="red"
```

3. Code Organization

Use extensions to organize your code into logical blocks of functionality. Each extension should be set off with a // MARK: - comment to keep things well-organized.

3.1 Protocol Conformance

In particular, 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: - UITableViewDataSourceextension 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
}
```

Since the compiler does not allow you to re-declare protocol conformance in a derived class, it is not always required to replicate the extension groups of the base class. This is especially true if the derived class is a terminal class and a small number of methods are being overriden. When to preserve the extension groups is left to the discretion of the author.

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

3.2 Unused Code

Unused (dead) code, including Xcode template code and placeholder comments should be removed. An exception is when your tutorial or book instructs the user to use the commented code.

Aspirational methods not directly associated with the tutorial whose implementation simply calls the super class should also be removed. This includes any empty/unused UIApplicationDelegate methods.

Preferred

```
overridefuncdidReceiveMemoryWarning() {
            super.didReceiveMemoryWarning()   // Dispose of any resources that can be
recreated.
}
overridefuncnumberOfSectionsInTableView(tableView: UITableView) ->Int {
            // #warning Incomplete implementation, return the number of sections
return1
}
overridefunctableView(tableView: UITableView, numberOfRowsInSectionsection: Int) ->Int {
            // #warning Incomplete implementation, return the number of
            rows return Database.contacts.count
}
```

Not Preferred

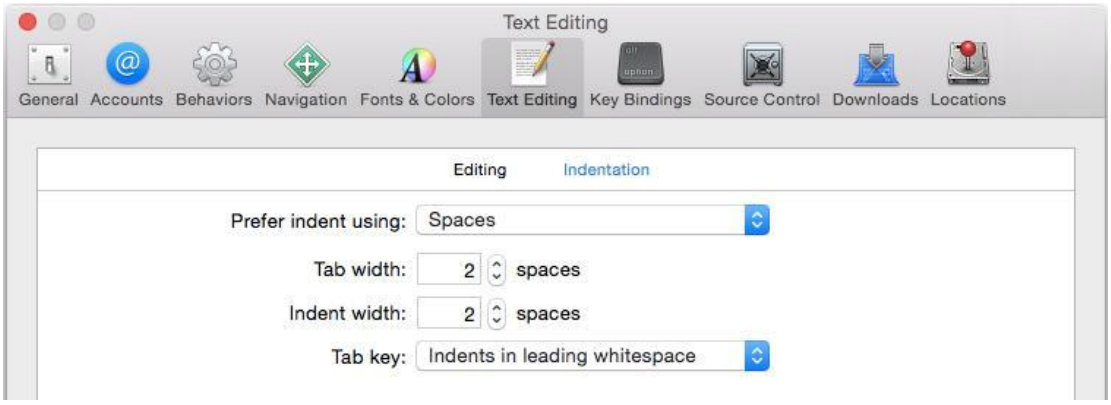
```
overridefunctableView(tableView: UITableView, numberOfRowsInSectionsection: Int) ->Int
            { return Database.contacts.count
}
```

3.3 Minimal Imports

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

4. Spacing

- Indent using 2 spaces rather than tabs to conserve space and help prevent line wrapping. Be sure to set this preference in Xcode and in the Project settings as shown below:

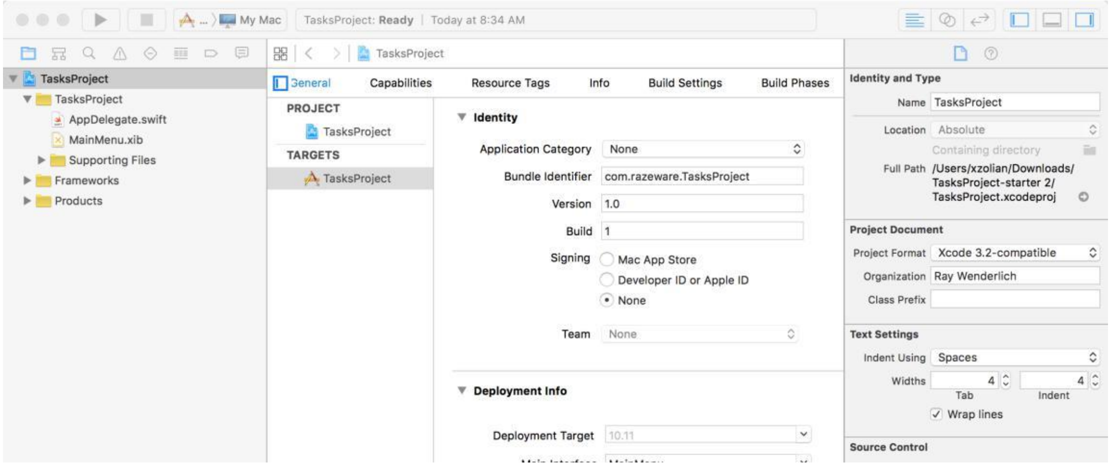


Prefer indent using: Spaces

Tab width: 2 spaces

Indent width: 2 spaces

Tab key: Indents in leading whitespace



Application Category: None

Bundle Identifier: com.razeware.TasksProject

Version: 1.0

Build: 1

Signing: ○ Mac App Store ○ Developer ID or Apple ID ● None

Team: None

Deployment Target: 10.11

Name: TasksProject

Location: Absolute

Containing directory

Full Path: /Users/xzolian/Downloads/TasksProject-starter 2/TasksProject.xcodeproj

Project Format: Xcode 3.2-compatible

Organization: Ray Wenderlich

Class Prefix:

Indent Using: Spaces

Widths: 4 Tab 4 Indent

☑ Wrap lines

- Method braces and other braces (if/else/switch/while etc.) always open on the same line as the statement but close on a new line.

- Tip: You can re-indent by selecting some code (or ⌘A to select all) and then Control-I (or Editor\Structure\Re-Indent in the menu). Some of the Xcode template code will have 4-space tabs hard coded, so this is a good way to fix that.

Preferred

```
if user.isHappy {
            // Do something
} else {
            // Do something else
}
```

Not Preferred

```
if user.isHappy
{
            // Do something
}
else {
            // Do something else
}
```

- There should be exactly one blank line between methods to aid in visual clarity and organization. Whitespace within methods should separate functionality, but having too many sections in a method often means you should refactor into several methods.

- Colons always have no space on the left and one space on the right. Exceptions are the ternary operator ? :and empty dictionary [:].

Preferred

```
class TestDatabase: Database {
            var data: [String: CGFloat] = ["A": 1.2, "B": 3.2]
}
```

Not Preferred

```
class TestDatabase : Database {
            var data :[String:CGFloat] = ["A":1.2, "B":3.2]
}
```

5. Comments

When they are needed, use comments to explain **why** a particular piece of code does something. Comments must be kept up-to-date or deleted.

Avoid block comments inline with code, as the code should be as self-documenting as possible. *Exception: This does not apply to those comments used to generate documentation.*

6. Classes and Structures

Which one to use?

Remember, structs have value semantics. Use structs for things that do not have an identity. An array that contains [a, b, c] is really the same as another array that contains [a, b, c] and they are completely interchangeable. It doesn't matter whether you use the first array or the second, because they represent the exact same thing. That's why arrays are structs.

Classes have reference semantics. Use classes for things that do have an identity or a specific life cycle. You would model a person as a class because two person objects are two different things. Just because

two people have the same name and birthdate, doesn't mean they are the same person. But the person's birthdate would be a struct because a date of 3 March 1950 is the same as any other date object for 3 March 1950. The date itself doesn't have an identity.

Sometimes, things should be structs but need to conform to AnyObject or are historically modeled as classes already (NSDate, NSSet). Try to follow these guidelines as closely as possible.

Example Definition

Here's an example of a well-styled class definition:

```
class Circle: Shape {
    var x: Int, y: Int
    var radius: Double
    var diameter: Double {
        get {
            return radius *2
        }
        set {
            radius = newValue /2
        }
    }
    init(x: Int, y: Int, radius: Double) {
        self.x = x
        self.y = y
        self.radius = radius
    }
    convenienceinit(x: Int, y: Int, diameter: Double) {
        self.init(x: x, y: y, radius: diameter /2)
    }
    funcdescribe() ->String {
        return"I am a circle at \(centerString()) with an area of \(computeArea())"
    }
    overridefunccomputeArea() ->Double {
        return M_PI * radius * radius
    }
    privatefunccenterString() ->String {
        return"(\(x),\(y))"
    }
}
```

The example above demonstrates the following style guidelines:

Specify types for properties, variables, constants, argument declarations and other statements with a space after the colon but not before, e.g. x: Int, and Circle: Shape.

Define multiple variables and structures on a single line if they share a common purpose / context.

Indent getter and setter definitions and property observers.Don't add modifiers such as internal when they're already the default. Similarly, don't repeat the access modifier when overriding a method.

6.1 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 when required to differentiate between property names and arguments in initializers, and when referencing properties in closure expressions (as required by the compiler):

```
class BoardLocation {
        let row: Int, column: Int
        init(row: Int, column: Int) {
                self.row = row
                self.column = column
                let closure = {
                        print(self.row)
                }
        }
}
```

6.2 Computed Properties

For conciseness, if a computed property is read-only, omit the get clause. The get clause is required only when a set clause is provided.

Preferred

```
var diameter: Double {
        return radius *2
}
```

Not Preferred

```
var diameter: Double {
        get {
                return radius *2
        }
}
```

6.3 Final

Mark classes final when inheritance is not intended.

Example:

```
// Turn any generic type into a reference type using this Box
class. finalclass Box<T> {
        let value: T    init(_ value: T) {
                self.value= value
        }
}
```

7. Function Declarations

Keep short function declarations on one line including the opening brace:

```
funcreticulateSplines(spline: [Double]) ->Bool {
        // reticulate code goes here
}
```

For functions with long signatures, add line breaks at appropriate points and add an extra indent on subsequent lines:

```
funcreticulateSplines(spline: [Double], adjustmentFactor: Double,     translateConstant: Int,
comment: String) ->Bool {
        // reticulate code goes here
}
```

8. Closure Expressions

Use trailing closure syntax only if there's a single closure expression parameter at the end of the argument list. Give the closure parameters descriptive names.

Preferred

```
UIView.animateWithDuration(1.0) {
        self.myView.alpha =0
}
UIView.animateWithDuration(1.0,
        animations: {
                self.myView.alpha =0
        },
        completion: { finished in
                self.myView.removeFromSuperview()
        }
)
```

Not Preferred

```
UIView.animateWithDuration(1.0, animations: {
        self.myView.alpha =0
})

UIView.animateWithDuration(1.0,
        animations: {
                self.myView.alpha =0
        }) { f in
                self.myView.removeFromSuperview()
}
```

For single-expression closures where the context is clear, use implicit returns:

```
attendeeList.sort {
        a, b in  a > b
}
```

Chained methods using trailing closures should be clear and easy to read in context. Decisions on spacing, line breaks, and when to use named versus anonymous arguments is left to the discretion of the author.

Examples:

```
let value = numbers.map { $0*2 }.filter { $0%3==0 }.indexOf(90)
let value = numbers
        .map {$0*2}
        .filter {$0>50}
        .map {$0+10}
```

9. 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.

9.1 Constants

Constants are defined using the let keyword, and variables with the var keyword. Always use let instead of varif 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 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 {
            staticlet e  =2.718281828459045235360287
            staticlet pi =3.141592653589793238462643
}

radius * Math.pi *2// circumference
```

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 pi =3.141592653589793238462643

radius * pi *2// is pi instance data or a global constant?
```

9.2 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.

9.3 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

```
iflet 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 oractualLabel.

Preferred

```
var subview: UIView?
var volume: Double?  // later on...

iflet subview = subview, volume = volume {
            // do something with unwrapped subview and volume
}
```

Not Preferred

```
var optionalSubview: UIView?
var volume: Double?

iflet unwrappedSubview = optionalSubview {
            iflet realVolume = volume {
                        // do something with unwrappedSubview and realVolume
            }
}
```

9.4 Struct Initializers

Use the native Swift struct initializers rather than the legacy CGGeometry constructors.

Preferred

```
let bounds = CGRect(x: 40, y: 20, width: 120, height: 80)
let centerPoint = CGPoint(x: 96, y: 42)
```

Not Preferred

```
let bounds = CGRectMake(40, 20, 120, 80)
let centerPoint = CGPointMake(96, 42)
```

Prefer the struct-scope constants CGRect.infinite, CGRect.null, etc. over global constants CGRectInfinite,CGRectNull, etc. For existing variables, you can use the shorter .zero.

9.5 Lazy Initialization

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

```
lazyvar locationManager: CLLocationManager =self.makeLocationManager()

privatefuncmakeLocationManager() -> 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.

9.6 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 asCGFloat 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 guideline means picking descriptive names is even more important than before.

9.7 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>
```

10. Functions vs Methods

Free functions, which aren't attached to a class or type, should be used sparingly. When possible, prefer to use a method instead of a free function. This aids in readability and discoverability.

Free functions are most appropriate when they aren't associated with any particular type or instance.

Preferred

```
let sorted = items.mergeSort()              // easily discoverable
rocket.launch()                             // clearly acts on the model
```

Not Preferred

```
let sorted = mergeSort(items)  // hard to discover l
aunch(&rocket)
```

Free Function Exceptions

```
let tuples =zip(a, b)  // feels natural as a free function (symmetry)
let value =max(x,y,z)            // another free function that feels natural
```

11. Memory Management

Code (even non-production, tutorial demo code) should not create reference cycles. Analyze your object graph and prevent strong cycles with weak and unowned references. Alternatively, use value types (struct, enum) to prevent cycles altogether.

11.1 Extending object lifetime

Extend object lifetime using the [weak self] and guard let strongSelf = self else { return } idiom. [weak self] is preferred to [unowned self] where it is not immediately obvious that self outlives the closure. Explicitly extending lifetime is preferred to optional unwrapping.

Preferred

```
resource.request().onComplete { [weakself] response
            in guardlet strongSelf =selfelse { return }
            let model = strongSelf.updateModel(response)
            strongSelf.updateUI(model)
}
```

Not Preferred

```
// might crash if self is released before response returns resource.request().onComplete {
[unownedself] response in
            let model =self.updateModel(response)
            self.updateUI(model)
}
```

Not Preferred

```
// deallocate could happen between updating the model and updating
UI resource.request().onComplete { [weakself] response in
            let model =self?.updateModel(response)
            self?.updateUI(model)
}
```

12 Access Control

Full access control annotation in tutorials can distract from the main topic and is not required. Using privateappropriately, however, adds clarity and promotes encapsulation. Use private as the leading property specifier. The only things that should come before access control are the static specifier or attributes such as @IBAction and@IBOutlet.

Preferred

```
class TimeMachine {    privatedynamiclazyvar fluxCapacitor = FluxCapacitor() }
```

Not Preferred

```
class TimeMachine {    lazydynamicprivatevar fluxCapacitor = FluxCapacitor() }
```

13. Control Flow

Prefer the for-in style of for loop over the while-condition-increment style.

Preferred

```
for _ in0..<3 {
            print("Hello three times")
}
```

```
for (index, person) in attendeeList.enumerate() {
        print("\(person) is at position #\(index)")
}

for index in0.stride(to: items.count, by: 2) {
        print(index)
}
for index in (0...3).reverse() {
        print(index)
}
```

Not Preferred

```
var i =0
while i <3 {
        print("Hello three times")   i +=1
}
var i =0
while i < attendeeList.count {
        let person = attendeeList[i]
        print("\(person) is at position #\(i)")
        i +=1
}
```

14. Golden Path

When coding with conditionals, the left hand margin of the code should be the "golden" or "happy" path. That is, don't nest if statements. Multiple return statements are OK. The guard statement is built for this.

Preferred

```
funccomputeFFT(context: Context?, inputData: InputData?) throws -> Frequencies
        { guardlet context = context else { throw FFTError.noContext }
        guardlet inputData = inputData else { throw FFTError.noInputData }

        // use context and input to compute the
        frequencies return frequencies
}
```

Not Preferred

```
funccomputeFFT(context: Context?, inputData: InputData?) throws -> Frequencies
        { iflet context = context {
                iflet inputData = inputData {
                        // use context and input to compute the frequencies
                        return frequencies
                }
                else {
                        throw FFTError.noInputData
                }
        }
        else {
                throw FFTError.noContext
        }
}
```

When multiple optionals are unwrapped either with guard or if let, minimize nesting by using the compound version when possible. Example:

Preferred

```
guardlet number1 = number1, number2 = number2, number3 = number3 else
{ fatalError("impossible") }
// do something with numbers
```

Not Preferred

```
iflet number1 = number1 {
        iflet number2 = number2 {
                iflet number3 = number3 {
                        // do something with numbers
                }
                else {
                        fatalError("impossible")
                }
        }
        else {
                fatalError("impossible")
        }
}
else {
        fatalError("impossible")
}
```

14.1 Failing Guards

Guard statements are required to exit in some way. Generally, this should be simple one line statement such asreturn, throw, break, continue, and fatalError(). Large code blocks should be avoided. If cleanup code is required for multiple exit points, consider using a defer block to avoid cleanup code duplication.

15. Semicolons

Swift does not require a semicolon after each statement in your code. They are only required if you wish to combine multiple statements on a single line.

Do not write multiple statements on a single line separated with semicolons.

The only exception to this rule is the for-conditional-increment construct, which requires semicolons. However, alternative for-in constructs should be used where possible.

Preferred

```
let swift ="not a scripting language"
```

Not Preferred

```
let swift ="not a scripting language";
```

NOTE: Swift is very different from JavaScript, where omitting semicolons is generally considered unsafe

16. Parentheses

Parentheses around conditionals are not required and should be omitted.

Preferred:

```
if name =="Hello" {
            print("World")
}
```

Not Preferred:

```
if (name =="Hello") {
            print("World")
}
```

Courtesy: The standards have been taken from Raywenderlich Swift Style Guide