

Swift Coding Standards

Using Swift

- These standards apply to Swift 5.0 and later.
- When using Swift in a Vokal project, use **Swift 5.0** or **higher**.
- Favor readability and clarity above fewer keystrokes.

When In Doubt

If questions aren't addressed here refer to the style guides of [raywenderlich.com with swift4.2](https://raywenderlich.com/swift4.2) and swift.org and [Swift API Design Guidelines](https://swift.org/doc/Language-Guidelines). If there are inconsistencies, our own standards take precedence. We're at the mercy of a cruel and capricious language unless you, the Vokal iOS Engineer, open a pull request.

- [Swift Coding Standards](#)
 - [Using Swift](#)
 - [When In Doubt](#)
- [Musts](#)
 - [Native Swift Types](#)
 - [Swift Collection Types](#)
 - [Class Prefixes](#)
 - [Optionals](#)
 - [Error Handling](#)
 - [Let vs. Var](#)
 - [Access Control](#)
 - [Spacing](#)
 - [Closures](#)
 - [Protocols](#)
 - [Arrays and Dictionaries](#)
 - [Constants](#)
 - [Function Parameters](#)
 - [Argument Labels](#)
 - [Semicolons](#)
 - [Typealiases](#)
 - [Flow Control](#)

- [Switch Statements](#)
- [Use Implicit Getters](#)
- [Loops](#)
- [Shoulds](#)
 - [Spacing](#)
 - [Usage of self](#)
 - [Loops](#)
 - [Closures](#)
 - [Trailing Closure Syntax](#)
 - [Argument Labels](#)
 - [Operator Overloading + Custom Operators](#)
 - [Tuples](#)
 - [Constants](#)
 - [Default Initializers](#)
 - [Classes vs Structs](#)
- [Tips & Tricks](#)

Musts

Native Swift Types

- Use Swift types whenever possible (`Array` , `Dictionary` , `Set` , `String` , etc.) as opposed to the `NS*` types from Objective-C. Many Objective-C types can be automatically converted to Swift types and vice versa. See [Working With Cocoa Data Types](#) for more details.

Incorrect

```
let pageLabelText = NSString(format: "%@/%@", currentPage, pageCount)
```

Correct

```
let pageLabelText = "\(currentPage)/\(pageCount)"
let alsoPageLabelText = currentPage + "/" + pageCount
```

Swift Collection Types

- Do not make `NSArray`, `NSDictionary`, and `NSSet` properties or variables. If you need to use a specific method only found on a Foundation collection, cast your Swift type in order to use that method.

Incorrect

```
var arrayOfJSONObjects: NSArray = NSArray()  
...  
let names: AnyObject? = arrayOfJSONObjects.value(forKeyPath: "name")
```

Correct

```
var arrayOfJSONObjects = [[String: AnyObject]]()  
...  
let names: AnyObject? = (arrayOfJSONObjects as NSArray).value(forKeyPath: "name")
```

- Consider if there is a Swiftier way to do what you're trying to do:

Swiftier Correct

```
var arrayOfJSONObjects = [[String: AnyObject]]()  
...  
let names: [String] = arrayOfJSONObjects.compactMap { object in  
    return object["name"] as? String  
}
```

Class Prefixes

- Do not add a class prefix. Swift types are automatically namespaced by the module that contains them. If two names from different modules collide you can disambiguate by prefixing the type name with the module name.

```
// SomeModule.swift  
public class UsefulClass {
```

```

    public class func helloWorld() {
        print("helloWorld from SomeModule")
    }
}
// MyApp.Swift
class UsefulClass {
    class func helloWorld() {
        print("helloWorld from MyApp")
    }
}
import SomeModule
let appClass = UsefulClass.helloWorld()
let moduleClass = SomeModule.UsefulClass.helloWorld()

```

Source: [RW – Swift Style Guide](#)

Optionals

Force Unwrapping

- Avoid force unwrapping optionals by using `!` or `as!` as this will cause your app to crash if the value you are trying to use is `nil`. Safely unwrap the optional first by using things like `guard let`, `if let`, `guard let as?`, `if let as?`, and optional chaining. A rare reason to force-unwrap would be if you have a value you expect to never be `nil` and you want your app to crash if the value actually is `nil` due to some implementation mistake. An example of this would be an `@IBOutlet` that accidentally gets disconnected. However, consider this an edge-case and rethink whether your own code could be refactored to not use force-unwrapping.

Incorrect unwrap

```

// URL init(string:) is a failable initializer and will crash at runtime with
// a force unwrap if initialization fails!
let url = URL(string: "http://www.example.com/")!
UIApplication.shared.open(url)

```

Correct unwrap

```

guard let url = URL(string: "http://www.example.com/") else {

```

```
        return
    }
    UIApplication.shared.open(url)
```

Incorrect downcast

```
// segue.destination is declared to be of type UIViewController, so forcing a
// downcast to type
// DetailViewController here will crash if the type is not DetailViewControll
// er at runtime!
let detailViewController = segue.destination as! DetailViewController
detailViewController.person = person
```

Correct downcast

```
guard let detailViewController = segue.destination as? DetailViewController e
lse {
    return
}
detailViewController.person = person
```

Incorrect optional chaining

```
// delegate is an optional so force unwrapping here will crash if delegate is
// actually nil at runtime!
delegate!.didSelectItem(item)
```

Correct optional chaining

```
delegate?.didSelectItem(item)
```

if let Pyramid of Doom

- Use multiple optional binding in an `if let` statement where possible to avoid the pyramid of doom:

Incorrect

```

if let id = jsonObject[Constants.id] as? Int {
    if let firstName = jsonObject[Constants.firstName] as? String {
        if let lastName = jsonObject[Constants.lastName] as? String {
            if let initials = jsonObject[Constants.initials] as? String {
                // Deep nesting
                let user = User(id: id, firstName: name, lastName: lastName,
initials: initials)
                // ...
            }
        }
    }
}

```

Correct

```

if
    let id = jsonObject[Constants.Id] as? Int,
    let firstName = jsonObject[Constants.firstName] as? String,
    let lastName = jsonObject[Constants.lastName] as? String,
    let initials = jsonObject[Constants.initials] as? String {
    // Flat
    let user = User(id: id, name: name, initials: initials)
    // ...
}

```

- If there are multiple unwrapped variables created, put each on its own line for readability (as in the example above).

Unwrapping Multiple Optionals

- When using `guard`, `if`, or `while` to unwrap multiple optionals, put each constant and/or variable onto its own line, followed by a `,` except for the last line, which should be followed by `else {` for `guard` (though this may be on its own line), or `{` for `if` and `while`.

Incorrect

```

guard let constantOne = valueOne,
    let constantTwo = valueTwo,
    let constantThree = valueThree

```

```

    else {
        return
    }
    if let constantOne = valueOne,
        let constantTwo = valueTwo,
        let constantThree = valueThree
    {
        // Code
    }
    guard let constantOne = valueOne, constantTwo = valueTwo, constantThree = valueThree else {
        return
    }
    if let constantOne = valueOne, let constantTwo = valueTwo, let constantThree = valueThree {
        // Code
    }

```

Correct

```

guard
    let constantOne = valueOne,
    let constantTwo = valueTwo,
    let constantThree = valueThree else {
    }
if
    let constantOne = valueOne,
    let constantTwo = valueTwo,
    let constantThree = valueThree {
        // Code
    }

```

- Put a line-break after `guard`, `if`, or `while` and list each constant or variable its own line.

Incorrect

```

guard let
    constantOne = valueOne,
    var variableOne = valueTwo,

```

```

        let constantTwo = valueThree else {
            return
        }
    if let constantOne = valueOne,
        var variableOne = valueTwo,
        var variableTwo = valueThree,
        var variableThree = valueFour,
        let constantTwo = valueFive {
        // Code
    }

```

Correct

```

    guard
        let constantOne = valueOne,
        let constantTwo = valueTwo,
        let constantThree = valueThree,
        var variableOne = valueFour,
        var variableTwo = valueFive,
        var variableThree = valueSix else {
        return
    }
    if
        let constantOne = valueOne,
        let constantTwo = valueTwo,
        let constantThree = valueThree,
        var variableOne = valueFour,
        var variableTwo = valueFive,
        var variableThree = valueSix {
        // Code
    }

```

Error Handling

Forced-try Expression

- **Avoid using the forced-try expression** `try!` as a way to ignore errors from throwing methods as this will crash your app if the error actually gets thrown. Safely handle errors using a `do` statement along with `try` and `catch`. A rare reason to use the forced-try expression is similar to force unwrapping optionals; you actually want the

app to crash (ideally during debugging before the app ships) to indicate an implementation error. An example of this would be loading a bundle resource that should always be there unless you forgot to include it or rename it.

Incorrect

```
// This will crash at runtime if there is an error parsing the JSON data!  
let json = try! JSONSerialization.jsonObject(with: data, options: .allowFragments)  
print(json)
```

Correct

```
do {  
    let json = try JSONSerialization.jsonObject(with: data, options: .allowFragments)  
    print(json)  
} catch {  
    print(error)  
}
```

Let vs. Var

- Whenever possible use `let` instead of `var`.
- Declare properties of an object or struct that shouldn't change over its lifetime with `let`.
- If the value of the property isn't known until creation, it can still be declared `let`: [assigning constant properties during initialization](#).
- Add **data type** while property define

Recommended

```
let maxBorderWidth: CGFloat = 1.0  
var userName: String = ""  
lazy var userInfoCache: [String: UserModel] = [:]
```

Discouraged

```
let maxBorderWidth = CGFloat(1.0)
var userName = ""
lazy var userInfoCache = [String: UserModel]()
```

Access Control

- Prefer `private` properties and methods whenever possible to encapsulate and limit access to internal object state.
- For private declarations at the top level of a file that are outside of a type, explicitly specify the declaration as `fileprivate`. This is functionally the same as marking these declarations `private`, but clarifies the scope:

Incorrect

```
import Foundation
// Top level declaration
private let foo = "bar"
struct Baz {
  ...
```

Correct

```
import Foundation
// Top level declaration
fileprivate let foo = "bar"
struct Baz {
  ...
```

- If you need to expose functionality to other modules, prefer `public` classes and class members whenever possible to ensure functionality is not accidentally overridden. Better to expose the class to `open` for subclassing when needed.

Spacing

- Open curly braces on the same line as the statement and close on a new line.
- Don't add empty lines after opening braces or before closing braces.
- Put `else` statements on the same line as the closing brace of the previous if block.
- Make all colons left-hugging (no space before but a space after) except when used with the ternary operator (a space both before and after).

Incorrect

```
class SomeClass : SomeSuperClass
{
    private let someString:String
    func someFunction(someParam :Int)
    {
        let dictionaryLiteral : [String : AnyObject] = ["foo" : "bar"]
        let ternary = (someParam > 10) ? "foo": "bar"
        if someParam > 10 { ... }
        else {
            ...
        } } }
}
```

Correct

```
class SomeClass: SomeSuperClass {
    private let someString: String
    func someFunction(someParam: Int) {
        let dictionaryLiteral: [String: AnyObject] = ["foo": "bar"]
        let ternary = (someParam > 10) ? "foo" : "bar"
        if someParam > 10 {
            ...
        } else {
            ...
        }
    }
}
```

Closures

Shorthand Argument Syntax

- Only use shorthand argument syntax for simple one-line closure implementations:

```
let doubled = [2, 3, 4].map { $0 * 2 } // [4, 6, 8]
```

- For all other cases, explicitly define the argument(s):

```
let names = ["George Washington", "Martha Washington", "Abe Lincoln"]
let emails: [String] = names.map { fullname in
    let dottedName = fullname.replacingOccurrences(of: " ", with: ".")
    return dottedName.lowercased() + "@whitehouse.gov"
}
```

Capture lists

- Use capture lists to [resolve strong reference cycles in closures](#):

```
UserAPI.registerUser(user) { [weak self] result in
    if result.success {
        self?.doSomethingWithResult(result)
    }
}
```

Protocols

Protocol Conformance

- When adding protocol conformance to a type, use 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 type with its associated methods.
- Use a `// MARK: - SomeDelegate` comment to keep things well organized.

Incorrect

```
class MyViewController: UIViewController, UITableViewDataSource, UIScrollView
Delegate {
```

```
// All methods
}
```

Correct

```
class MyViewController: UIViewController {
    ...
}
// MARK: - UITableViewDataSource
extension MyViewController: UITableViewDataSource {
    // Table view data source methods
}
// MARK: - UIScrollViewDelegate
extension MyViewController: UIScrollViewDelegate {
    // Scroll view delegate methods
}
```

Delegate Protocols


- Limit delegate protocols to classes only by adding `class` to the protocol's inheritance list (as discussed in [Class-Only Protocols](#)).
- If your protocol should have [optional methods](#), it must be declared with the `@objc` attribute.
- Declare protocol definitions near the class that uses the delegate, not the class that implements the delegate methods.
- If more than one class uses the same protocol, declare it in its own file.
- Use `weak` optional `var`s for delegate variables to avoid retain cycles.

```
//SomeTableViewCell.swift
protocol SomeTableViewCellDelegate: class {
    func cellButtonWasTapped(cell: SomeTableViewCell)
}
class SomeTableViewCell: UITableViewCell {
    weak var delegate: SomeTableViewCellDelegate?
    // ...
}
```

```
//SomeTableViewController.swift
class SomeTableViewController: UITableViewController {
    // ...
}
// MARK: - SomeTableCellDelegate
extension SomeTableViewController: SomeTableCellDelegate {
    func cellButtonWasTapped(cell: SomeTableCell) {
        // Implementation of cellbuttonwasTapped method
    }
}
```

Arrays and Dictionaries

Type Shorthand Syntax

- Use square bracket shorthand type syntax for Array and Dictionary as recommended by  in [Array Type Shorthand Syntax](#):

Incorrect

```
let users: Array<String>
let usersByName: Dictionary<String, User>
```

Correct

```
let users: [String]
let usersByName: [String: User]
```

Trailing Comma

- For array and dictionary literals, unless the literal is very short, split it into multiple lines, with the opening symbols on their own line, each item or key–value pair on its own line, and the closing symbol on its own line. Put a trailing comma after the last item or key–value pair to facilitate future insertion/editing. Xcode will handle alignment sanely.

Correct

```
let anArray = [
    object1,
    object2,
    object3,
]
let aDictionary = [
    "key1": value1,
    "key2": value2,
]
```

Incorrect

```
let anArray = [
    object1,
    object2,
    object3 //no trailing comma
]
let aDictionary = ["key1": value1, "key2": value2] //how can you even read th
at?!
```

Constants

- Define constants for unchanging pieces of data in the code. Some examples are `CGFloat` constants for cell heights, string constants for cell identifiers, key names (for KVC and dictionaries), or segue identifiers.
- Where possible, keep constants private to the file they are related to.
- File-level constants must be declared with `fileprivate let`.
- File-level constants must be capital camel-cased to indicate that they are named constants instead of properties.
- If the constant will be used outside of one file, `fileprivate` must be omitted.
- If the constant will be used outside of the module, it must be declared `public` (mostly useful for Pods or shared libraries).
- If the constant is declared within a class or struct, it must be declared `static` to

avoid declaring one constant per instance.

```
//SomeTableViewCell.swift
//not declared private since it is used in another file
let SomeTableViewCellIdentifier = "SomeTableViewCell"
class SomeTableViewCell: UITableViewCell {
    ...
}
```

```
//ATableViewController.swift
//declared fileprivate since it isn't used outside this file
fileprivate let RowHeight: CGFloat = 150.0
class ATableViewController: UITableViewController {
    ...
    private func configureTableView() {
        tableView.rowHeight = RowHeight
    }
    func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
        return tableView.dequeueReusableCellWithIdentifier(SomeTableViewCellIdentifier, forIndexPath: indexPath)
    }
}
```

Function Parameters

- Follow the Swift API Design Guidelines for proper handling of [function parameters](#) and [argument labels](#).

func move(from start: Point, to end: Point)

- **Choose parameter names to serve documentation.** Even though parameter names do not appear at a function or method's point of use, they play an important explanatory role.
 - Choose these names to make documentation easy to read. For example, these names make documentation read naturally:

Correct


```

/// Return an `Array` containing the elements of `self`
/// that satisfy `predicate`.
func filter(_ predicate: (Element) -> Bool) -> [Generator.Element]
/// Replace the given `subRange` of elements with `newElements`.
mutating func replaceRange(_ subRange: Range, with newElements: [E])

```

InCorrect

```

/// Return an `Array` containing the elements of `self`
/// that satisfy `includedInResult`.
func filter(_ includedInResult: (Element) -> Bool) -> [Generator.Element]
/// Replace the range of elements indicated by `r` with
/// the contents of `with`.
mutating func replaceRange(_ r: Range, with: [E])

```

These, however, make the documentation awkward and ungrammatical:

- Take advantage of **defaulted parameters** when it simplifies common uses. Any parameter with a single commonly-used value is a candidate for a default.
 - Default arguments improve readability by hiding irrelevant information. For example:

InCorrect

```

let order = lastName.compare(
  royalFamilyName, options: [], range: nil, locale: nil)

```

can become the much simpler:

```

let order = lastName.compare(royalFamilyName)

```

Default arguments are generally preferable to the use of method families, because they impose a lower cognitive burden on anyone trying to understand the API.

Correct

```
extension String {
    /// ...description...
    public func compare(
        _ other: String, options: CompareOptions = [],
        range: Range? = nil, locale: Locale? = nil
    ) -> Ordering
}
```

- **Prefer to locate parameters with defaults toward the end** of the parameter list. Parameters without defaults are usually more essential to the semantics of a method, and provide a stable initial pattern of use where methods are invoked.

Argument Labels

```
func move(from start: Point, to end: Point)
x.move(from: x, to: y)
```

- **Omit all labels when arguments can't be usefully distinguished**, e.g. `min(number1, number2)`, `zip(sequence1, sequence2)`.
- **In initializers that perform value preserving type conversions, omit the first argument label**, e.g. `Int64(someUInt32)`

The first argument should always be the source of the conversion.

```
extension String {
    // Convert `x` into its textual representation in the given radix
    init(_ x: BigInt, radix: Int = 10) ← Note the initial underscore
}
text = "The value is: "
text += String(veryLargeNumber)
text += " and in hexadecimal, it's"
text += String(veryLargeNumber, radix: 16)
```

In “narrowing” type conversions, though, a label that describes the narrowing is recommended.

```
extension UInt32 {
    /// Creates an instance having the specified `value`.
```

```

init(_ value: Int16)          ← Widening, so no label
/// Creates an instance having the lowest 32 bits of `source`.
init(truncating source: UInt64)
/// Creates an instance having the nearest representable
/// approximation of `valueToApproximate`.
init(saturating valueToApproximate: UInt64)
}

```

- A value preserving type conversion is a monomorphism, i.e. every difference in the value of the source results in a difference in the value of the result. For example, conversion from Int8 to Int64 is value preserving because every distinct Int8 value is converted to a distinct Int64 value. Conversion in the other direction, however, cannot be value preserving: Int64 has more possible values than can be represented in an Int8.

Note: the ability to retrieve the original value has no bearing on whether a conversion is value preserving.

- When the first argument forms part of a [prepositional phrase](#), give it an argument label. The argument label should normally begin at the [preposition](#), e.g. `x.removeBoxes(havingLength: 12)`.

An exception arises when the first two arguments represent parts of a single abstraction.

InCorrect

```

a.move(toX: b, y: c)
a.fade(fromRed: b, green: c, blue: d)

```

In such cases, begin the argument label after the preposition, to keep the abstraction clear.

Correct

```

a.moveTo(x: b, y: c)
a.fadeFrom(red: b, green: c, blue: d)

```

- Otherwise, if the first argument forms part of a grammatical phrase, omit its label, appending any preceding words to the base name, e.g. `x.addSubview(y)`

This guideline implies that if the first argument doesn't form part of a grammatical phrase, it should have a label.

Correct

```
view.dismiss(animated: false)
let text = words.split(maxSplits: 12)
let studentsByName = students.sorted(isOrderedBefore: Student.namePrecedes)
```

Note that it's important that the phrase convey the correct meaning. The following would be grammatical but would express the wrong thing.

InCorrect

```
view.dismiss(false)    Don't dismiss? Dismiss a Bool?
words.split(12)         Split the number 12?
```

Note also that arguments with default values can be omitted, and in that case do not form part of a grammatical phrase, so they should always have labels.

- Label all other arguments.

Semicolons

- Forget them. They're dead to us.
- Don't you dare put multiple statements on one line.

Typealiases

- Create `typealiases` to give semantic meaning to commonly used datatypes and closures.
- `typealias` is equivalent to `typedef` in C and should be used for making names for types.
- Where appropriate, nest `typealiases` inside parent types (classes, structs, etc.) to which they relate.

```
typealias IndexRange = Range<Int>
typealias JSONObject = [String: AnyObject]
typealias APICompletion = (jsonResult: [JSONObject]?, error: NSError?) -> Void
typealias BasicBlock = () -> Void
```

Flow Control

- For single conditional statements, do not use parentheses.
- Use parentheses around compound conditional statements for clarity or to make the order of operations explicit.
- When using optional booleans and optional `NSNumber`s that represent booleans, check for `true` or `false` rather than using the nil-coalescing operator:

Incorrect

```
if user.isCurrent?.boolValue ?? false {
    // isCurrent is true
} else {
    // isCurrent is nil or false
}
```

Correct

```
if user.isCurrent?.boolValue == true {
    // isCurrent is true
} else {
    // isCurrent is nil or false
}
```

Switch Statements

- `break` is not needed between `case` statements (they don't fall through by default)
- Use multiple values on a single `case` where it is appropriate:

```

var someCharacter: Character
...
switch someCharacter {
case "a", "e", "i", "o", "u":
    print("\(someCharacter) is a vowel")
...
}

```

- When pattern matching over an enum case with an associated value, use `case .CASENAME(let ...)` rather than `case let ...` syntax for value binding.

Incorrect

```

enum AnEnum {
    case foo
    case bar(String)
    case baz
}
let anEnumInstanceWithAssociatedValue = AnEnum.Bar("hello")
switch anEnumInstanceWithAssociatedValue {
    case .foo: print("Foo")
    // Incorrect
    case let .bar(barValue): print(barValue) // "hello"
    case .baz: print("Baz")
}

```


Correct

```

enum AnEnum {
    case foo
    case bar(String)
    case baz
}
let anEnumInstanceWithAssociatedValue = AnEnum.Bar("hello")
switch anEnumInstanceWithAssociatedValue {
    case .foo: print("Foo")
    // Correct
    case .bar(let barValue): print(barValue) // "hello"
    case .baz: print("Baz")
}

```

Use Implicit Getters

- When possible, omit the `get` keyword on read-only, computed properties and read-only subscripts as recommended by  under [Read-Only Computed Properties](#):

Incorrect

```
var someProperty: Int {  
    get {  
        return 4 * someOtherProperty  
    }  
}  
subscript(index: Int) -> T {  
    get {  
        return object[index]  
    }  
}
```

Correct

```
var someProperty: Int {  
    return 4 * someOtherProperty  
}  
subscript(index: Int) -> T {  
    return object[index]  
}
```

Loops

- Use the `enumerated()` function if you need to loop over a Sequence and use the index:

```
for (index, element) in someArray.enumerated() {  
    ...  
}
```

- Use `map` when transforming Arrays (`compactMap` for Arrays of Optionals or `flatMap`

for Arrays of Arrays):

```
let array = [1, 2, 3, 4, 5]
let stringArray = array.map { item in
    return "item \("\(item)"
}
let optionalArray: [Int?] = [1, nil, 3, 4, nil]
let nonOptionalArray = optionalArray.compactMap { item -> Int? in
    guard let item = item else {
        return nil
    }
    return item * 2
}
let arrayOfArrays = [array, nonOptionalArray]
let anotherStringArray = arrayOfArrays.flatMap { item in
    return "thing \("\(item)"
}
```

- If you are not performing a transform, or if there are side effects do not use `map/flatMap`; use a `for in` loop instead ([tips](#)).
- Avoid the use of `forEach` except for simple one line closures, similar to `makeObjectsPerformSelector:` in Objective-C.

Shoulds

Declaring Variables

- When declaring variables you should be verbose, but not too verbose.
- Avoid repeating type information. Once should be enough.
- Make the name descriptive.

Incorrect

```
var someDictionary: Dictionary = [String: String]() //Dictionary is redundant
var somePoint: CGPoint = CGPoint(x:100, y: 200) //CGPoint is repeated
var b = Bool(false) //b is not a descriptive name
```


Correct

```
var someArray = [String]()
var someArray: [String] = []
var someDictionary = [String: Int]()
var someDictionary: [String : Int] = [:]
var countOfCats: UInt32 = 12
var isMadeOfCheese = false
var somePoint = CGPoint(x:100, y: 200)
```

Optionals

guard let vs. if let

- Use `guard let` over `if let` where possible. This improves readability by [focusing on the happy-path](#) and can also reduce nesting by keeping your code flatter:

Incorrect

```
func openURL(string: String) {
    if let url = URL(string: string) {
        // Nested
        UIApplication.shared.open(url)
    }
}
```

Correct

```
func openURL(string: String) {
    guard let url = URL(string: string) else {
        return
    }
    // Flat
    UIApplication.shared.open(url)
}
```

- Since `guard let` needs to exit the current scope upon failure, `if let` is better suited for situations where you still need to move forward after failing to unwrap an optional:
-

```
if let anImage = UIImage(named: ImageNames.background) {
    imageView.image = anImage
}
// Do more configuration
// ...
```

Spacing

- Use a newline for each logical step when chaining 3 or more methods together in a fluent style:

Incorrect

```
func foo() -> Int {
    let nums: [Int] = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    return nums.map { $0 * 2 }.filter { $0 % 2 == 0 }.reduce(0, +)
}
```

Correct

```
func foo() -> Int {
    let nums: [Int] = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    return nums
        .map { $0 * 2 }
        .filter { $0 % 2 == 0 }
        .reduce(0, +)
}
```

- For `guard` statements, the `else {` may be placed on its own line after the last condition (rather than on the same line as the last condition). For a single-condition `guard`, if the `else {` is on its own line, the condition should be on its own line, too. If the `else` clause of a `guard` statement is a simple `return` statement, it may be all on one line.

Incorrect

```
func openURL(string: String) {
    guard let url = URL(string: string)
```

```

        else {
            return
        }
        // Flat
        UIApplication.shared.open(url)
    }

```

```

guard let constantOne = valueOne,
    let constantTwo = valueTwo,
    let constantThree = valueThree
else { return }

```

Correct (lots of choices here)

```

func openURL(string: String) {
    guard let url = URL(string: string) else {
        return
    }
    // Flat
    UIApplication.shared.open(url)
}

```

```

func openURL(string: String) {
    guard let url = URL(string: string) else { return }
    // Flat
    UIApplication.shared.open(url)
}

```

```

func openURL(string: String) {
    guard
        let url = URL(string: string)
    else {
        return
    }
    // Flat
    UIApplication.shared.open(url)
}

```

```

func openURL(string: String) {

```

```
guard
    let url = URL(string: string)
    else { return }
// Flat
UIApplication.shared.open(url)
}
```

```
guard
    let constantOne = valueOne,
    let constantTwo = valueTwo,
    let constantThree = valueThree else { return }
```

```
guard
    let constantOne = valueOne,
    let constantTwo = valueTwo,
    let constantThree = valueThree
    else { return }
```

- When grouping multiple `cases` in a `switch` statement, prefer putting each case on its own line, unless there are only a few cases and they are short.

Incorrect

```
switch something {
case .oneLongCase, .anotherLongCase, .thereAreMoreCases, .thisIsWayTooFarToTheRight:
    return true
case .sanity:
    return false
}
```

Correct

```
switch something {
case .oneLongCase,
    .anotherLongCase,
    .thereAreMoreCases,
    .thisIsInASanerPlace:
    return false
}
```

```
case .sanity:
    return true
}
```

Usage of self

- Except where necessary, avoid using `self.`. If you have a local variable that conflicts with a property name or are in a context where `self` is captured, you may need to use `self.`.

Loops

- If you have an Array of Arrays and want to loop over all contents, consider a `for in` loop using `joined(separator:)` instead of nested loops:

```
let arraysOfNames = [
    ["Moe", "Larry", "Curly"],
    ["Groucho", "Chico", "Harpo", "Zeppo"]
]
```

Recommended

```
for name in arraysOfNames.joined() {
    print("\(name) is an old-timey comedian")
}
```

Discouraged

```
for names in arraysOfNames {
    for name in names {
        print("\(name) is an old-timey comedian")
    }
}
```

Closures

- Avoid unnecessary parentheses around closure parameters.

Incorrect

```
functionWithAClosure { (result) in
    ...
}
```

```
functionWithAClosure { (result) -> Int in
    ...
}
```

Correct

```
functionWithAClosure { result in
    ...
}
```

```
functionWithAClosure { result -> Int in
    ...
}
```

```
functionWithAClosure { (result: String) in
    ...
}
```

Trailing Closure Syntax

- Use trailing closure syntax when the only or last argument to a function or method is a closure and there is only one closure parameter.

```
//a function that has a completion closure/block
func registerUser(user: User, completion: (Result) -> Void)
```

Correct

```
UserAPI.registerUser(user) { result in
    if result.success {
```



```

    ...
}
///
func executeAction(_ action: FXTutorialAction,
                  complete:((_ executeResult: Bool, _ interval: TimeInterval?) -> Void) {
    ...
}

```

Incorrect

```

typealias FXClosure_tutorialExecute = ((Bool, TimeInterval?) -> Void)
///
func executeAction(_ action: FXTutorialAction,
                  complete:((Bool, TimeInterval?) -> Void) {
    ...
}

```

Operator Overloading + Custom Operators

- The use of operator overloading and custom operators is **strongly discouraged** as this can hurt readability and potentially create a significant amount of confusion for other developers on a shared project. There are cases that it would be necessary (ex. overloading `==` to conform to `Equatable`). When writing a custom operator or overloading an existing one, the operator function should call another **explicitly named** function that performs that actual work. For more guidance on best practices on this matter, view the guidelines at the bottom of this [NSHipster article](#).

Tuples

- The word is pronounced like "tuh-ple"
- Rhymes with "couple" and "supple"
- Please **name** the members of your tuples when creating or decomposing tuples:

Correct


```
let foo = (something: "cats", somethingElse: 909_099)
let (something, somethingElse) = foo
func doSomething() -> (result: Bool, imageLink: Url?) {
    ...
}
```

InCorrect

```
let foo = ("cats", 909_099)
let (something, somethingElse) = foo
func doSomething() -> (Bool, Url?) {
    ...
}
```

Constants

- Declare constants with `static let` to ensure static storage.
- Prefer declaring constants in the scope in which they will be used rather than in a central shared constants file like **Constants.swift**.
- Be wary of large constants files as they can become unmanageable over time. Refactor related parts of the main constants file into separate files for that situation.
- Use `enums` to group related constants together in a namespace. The name of the `enum` should be singular, and each constant should be written using camelCase. (Using a `case`-less `enum` prevents useless instances from being created.)

```
enum SegueIdentifier {
    static let onboarding = "OnboardingSegue"
    static let login = "LoginSegue"
    static let logout = "LogoutSegue"
}
```

```
enum StoryboardIdentifier {
    static let main = "Main"
    static let onboarding = "Onboarding"
    static let settings = "Settings"
```

```
}  
print(SegueIdentifier.login) // "LoginSegue"
```

- Where appropriate, constants can also be grouped using an `enum` with a `rawValue` type that is relevant to the type you need to work with. An `enum` with a `rawValue` of type `String` will [implicitly assign](#) its `rawValue` from the name of the case if nothing is already explicitly defined for the `rawValue`. This can be useful when all the names of the cases match with the value of the constant. Be aware that if you use `enum` `cases` for constants in this way, you need to explicitly use `rawValue` every time you need to access the value of the constant:

```
enum UserJSONKeys: String {  
    case username  
    case email  
    case role  
    // Explicitly defined rawValue  
    case identifier = "id"  
    ...  
}  
print(UserJSONKeys.username.rawValue) // "username"  
print(UserJSONKeys.identifier.rawValue) // "id"  
guard let url = URL(string: "http://www.example.com") else {  
    return  
}  
let mutableURLRequest = NSMutableURLRequest(url: url)  
mutableURLRequest.HTTPMethod = HTTPMethods.POST.rawValue  
print(mutableURLRequest.httpMethod) // "POST"
```

Default Initializers

- Use [default initializers](#) where possible.

Classes vs Structs

- Most of your custom data types should be structs and enums.
- Some situations where you may want to use classes:
 - When you need the data sharing capabilities of reference types.
 - When you need deallocators to help free up any resources.

- When you need runtime class type checks.
- When you need Objective-C interoperability.
- When you need inheritance after considering all other possibilities like: protocols, protocol inheritance, protocol extensions with default implementations, generics.
- Refer to the [Swift Programming Language Guidelines](#) and [Choosing Between Structures and Classes](#) for detailed info on this topic.

Tips & Tricks

Simplify Xcode's Autocompletion Suggestions

- Xcode will try to be helpful when autocompleting closures for you by giving you the *full* type signature of the closure (input type(s) and return type). Simplify that information so that it's easier to read.
- Remove return types of `Void` and parentheses around single input parameters. This is especially relevant if the closure takes no input and returns no output.

What Xcode Autocompletion Suggests

```
UIView.animate(withDuration: 0.5) { () -> Void in
    ...
}
UIView.animate(withDuration: 0.5, animations: { () -> Void in
    ...
    }) { (complete) -> Void in
    ...
}
```

Simplified With Type Inference

```
UIView.animate(withDuration: 0.5) {
    //no need to specify type information for a no input, no output closure
}
//note the formatting of this example is further changed from the suggestion
for better readability
UIView.animate(withDuration: 0.5,
    animations: {
        ...
    },
```

```
completion: { complete in
    //the return type is inferred to be `Void` and `complete` does not ne
ed parens
    }
}
```

- [Swift Coding Standards](#)
 - [Using Swift](#)
 - [When In Doubt](#)
- [Musts](#)
 - [Native Swift Types](#)
 - [Swift Collection Types](#)
 - [Class Prefixes](#)
 - [Optionals](#)
 - [Error Handling](#)
 - [Let vs. Var](#)
 - [Access Control](#)
 - [Spacing](#)
 - [Closures](#)
 - [Protocols](#)
 - [Arrays and Dictionaries](#)
 - [Constants](#)
 - [Function Parameters](#)
 - [Argument Labels](#)
 - [Semicolons](#)
 - [Typealiases](#)
 - [Flow Control](#)
 - [Switch Statements](#)
 - [Use Implicit Getters](#)
 - [Loops](#)
- [Shoulds](#)
 - [Spacing](#)
 - [Usage of self](#)
 - [Loops](#)
 - [Closures](#)
 - [Trailing Closure Syntax](#)
 - [Argument Labels](#)
 - [Operator Overloading + Custom Operators](#)

- [Tuples](#)
- [Constants](#)
- [Default Initializers](#)
- [Classes vs Structs](#)
- [Tips & Tricks](#)