**iOS Style Guide**

Creators: Jitendra Kumar

**Correctness**

~~Strive to~~ **Always** make your code compile without warnings.

**General Naming Conventions**

Descriptive and consistent naming makes software easier to read and understand. Use the Swift naming conventions described in the API Design Guidelines. Some key takeaways include:

* Striving for **clarity** at the call site
* Prioritizing **clarity** over brevity
* Using **camel case** (not snake case)
* Using:
  + **uppercase** for **types** (and protocols),
  + **lowercase** for **everything else** including all needed words while omitting needless words
* Using **names based on roles**, not types
  + **Preferred**: var counter = 42
  + ~~Not Preferred~~: var myInt = 42
* Sometimes compensating for weak type information
* Striving for **fluent usage**
  + func item(for indexPath: IndexPath)
  + print(self.item(for: indexPath))
* Beginning factory methods with **make**

|  |
| --- |
| func makeDidSelectRowAtindexPathBlock() -> DidSelectRowAtIndexPathBlock {     return { (indexPath) in        print(“Selected item at \(indexPath)”)     }  } |

* Naming methods for their side effects
* Verb methods follow the -ed, -ing rule for the non-mutating version noun methods follow the formX rule for the mutating version
* Bool types should read like assertions
  + isValid
  + didSucceed
* Using precedent for names
* Choosing good parameter names that serve as documentation
* Labeling closure and tuple parameters – custom handlers
* typealias closures
* Taking advantage of default parameters instead of overloading

**~~Self~~**

When referring to instance variables, avoid use self when it’s not needed.

**Preferred:**

|  |
| --- |
| class ViewController {      private let label = UILabel()      override func viewDidLoad() {          super.viewDidLoad()          view.backgroundColor = .red // self not necessary for property references          fetchData { model // self not necessary for method calls             self.label.text = model.title // self necessary since it’s captured          }      }      func fetchData() {          // make API calls      }  } |

~~Not Preferred:~~

|  |
| --- |
| class ViewController {      override func viewDidLoad() {          super.viewDidLoad()          self.view.backgroundColor = .red          self.fetchData { model            }      }  } |

**Prose**

When referring to methods in prose, being unambiguous is critical. To refer to a method name, use the simplest form possible.

**Formula (writing** UIGestureRecognizer.addTarget**):**

1. Write the method name with no parameters.
   1. Example: func addTarget
2. Write the method name with argument labels.
   1. Example: func addTarget(\_:action:)
3. Write the full method name with argument labels and types.
   1. Example: func addTarget(\_: Any?, action: Selector?)

\*\***Pro Tip**: You can use Xcode's jump bar to lookup methods with argument labels. Methods in Xcode jump bar

**Use Type Inferred Context**

When possible, leave out the type declaration.

**Preferred:**

|  |
| --- |
| let selector = #selector(viewDidLoad)  view.backgroundColor = .red  let toView = context.view(forKey: .to)  let view = UIView(frame: .zero) |

~~Not Preferred:~~

|  |
| --- |
| let selector = #selector(ViewController.viewDidLoad)  view.backgroundColor = UIColor.red let toView = context.view(forKey: UITransitionContextViewKey.to) let view = UIView(frame: CGRect.zero) |

**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> { ... }  func write<Target: OutputStream>(to target: inout Target)  func swap<T>(\_ a: inout T, \_ b: inout T) |

**Not Preferred**

|  |
| --- |
| struct Stack<A> { ... }  func write<B: OutputStream>(to target: inout B) |

**Spelling**

Use english spelling to match Apple’s API docs as closely as possible.

**Preferred:**

let color = "gray"

~~Not Preferred:~~

let color = "grey"

**Protocol Conformance**

When adding protocol conformance, group methods with similar functionality into their own extensions and keep everything else in the main declaration.

**Preferred**

|  |
| --- |
| class MyViewController: UIViewController {     // class stuff here       // MARK: - UITableViewDataSource     func tableView(\_ tableView: UITableView,                    numberOfRowsInSection section: Int) -> Int {     }     // MARK: - UITableViewDelgate     func tableView(\_ tableView: UITableView,                    didSelectRowAt indexPath: IndexPath) {       }  }  // MARK: Loading  extension MyViewController {     func showLoadingDialog() {     }     func hideLoadingDialog() {    }  } |

~~Not Preferred:~~

|  |
| --- |
| class MyViewController: UIViewController, UITableViewDataSource, UITableViewDelegate {     // class stuff here  }    // MARK: - UITableViewDataSource  extension MyViewController: UITableViewDataSource {     // table view data source methods  }    // MARK: - UIScrollViewDelegate  extension MyViewController: UIScrollViewDelegate {     // scroll view delegate methods  } |

**\*\*Note**: 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 overridden. When to preserve the extension groups is left to the discretion of the author.

**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 superclass should also be removed. This includes any empty/unused UIApplicationDelegate methods.

**Delete dead code**

Remove code that’s not being used and never merge code that’s incomplete.

**Preferred:**

|  |
| --- |
| override func tableView(\_ tableView: UITableView,                          numberOfRowsInSection section: Int) -> Int {     return Database.contacts.count  } |

~~Not Preferred:~~

|  |
| --- |
| override func didReceiveMemoryWarning() {     super.didReceiveMemoryWarning()     // Dispose of any resources that can be recreated.  }  override func numberOfSections(in tableView: UITableView) -> Int {     // #warning Incomplete implementation, return the number of sections return 1  } |

**Minimal Imports**

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

**Spaces vs. tabs**

Always use tabs over spaces (EOM). If you prefer to use spaces, make sure to press the spacebar **4**  times instead of 2.

**Brackets**

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.

**\*\*Pro-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 happy  } else {     // Do something not happy  } |

~~Not Preferred:~~

|  |
| --- |
| if user.isHappy {      // Do something happy  }  else {      // Do something not happy  } |

**White Space**

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**

Colons always have no space on the left and one space on the right. Exceptions are the ternary operator ? :, empty dictionary [:] and #selector syntax for unnamed parameters (\_:).

**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]  } |

**Line Wrapping**

Long lines should be wrapped at around 70 characters. A hard limit is intentionally not specified.

**Indentation**

If the method has more than 2 parameters, indent. Otherwise keep it at one line. An exception can be made for really long method names.

**Indentation for Method Declarations**

Add a new line after every parameter.

**Preferred:**

|  |
| --- |
| func showCustomBorderFor(edge: RectEdge,                           color: UIColor,                           thickness: CGFloat = Dimens.borderHeight) {      // show border  } |

~~Not Preferred:~~

|  |
| --- |
| func showCustomBorderFor(edge: RectEdge, color: UIColor, thickness: CGFloat = Dimens.borderHeight) {      // show border  } |

**Indentation for Method Calls**

Add a new line after the opening parenthesis

**Preferred:**

|  |
| --- |
| self.showCustomBorderFor(     edge: .left,     color: Colors.gray5,     thickness: 1.0  ) |

~~Not Preferred:~~

|  |
| --- |
| self.showCustomBorderFor(edge: .left, color: Colors.gray5, thickness: 1.0) |

**Classes and Structures**

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.

**Enums**

* Use enums over constants when there is an exhaustive property for the idea you’re trying to represent.
* When you as a client have an intention of using an Enum outside of this class, declare it outside of the class / struct / enum that declares it. Otherwise, make it a private enum.
  + An exception to the rule is if you’re declaring an enum for a concept that has a relatively simple name:
    - RatingFilter.Option
    - RatingFilter.Mode

**Preferred**

|  |
| --- |
| // declaration  enum NavigationBarTheme {     case appColorBackground     ...  }  class NavigationItem {  }  // usage  let theme = NavigationBarTheme.appColorBackground |

~~Not Preferred~~

|  |
| --- |
| // declaration  class NavigationItem {     enum NavigationBarTheme {        case appColorBackground        ...     } }  // usage  let theme = NavigationItem.NavigationBarTheme.appColorBackground |

**Access Control**

Don't add modifiers such as internal when they're already the default. Similarly, don't repeat the access modifier when overriding a method.

**\*\*Pro tip**: Hide non-shared, implementation details inside the extension using private access control.

**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     }  } |

**Closure Expressions**

* When passing a closure as a parameter, make sure to include the parameter name if there is more than 1 closure being passed to a function call.
* 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.animate(withDuration: 1.0) {     self.myView.alpha = 0  }  UIView.animate(     withDuration: 1.0,     animations: {        self.myView.alpha = 0     },     completion: { finished in        self.myView.removeFromSuperview()     }  ) |

~~Not Preferred:~~

|  |
| --- |
| UIView.animate(withDuration: 1.0, animations: {     self.myView.alpha = 0  })    UIView.animate(     withDuration: 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 }

attendeeList.sort { $0 > $1 }

**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 |

**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.

**\*\*Pro 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**

* Don’t use force unwrap
* 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

textContainer.doSomething()

textContainer.doSomethingElse()

}

* 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()  let maximumWidth: CGFloat = 106.5 |

~~Not Preferred:~~

|  |
| --- |
| let message: String = "Click the button"  let currentBounds: CGRect = computeViewBounds() |

**Type Annotation Arrays and Dictionaries**

For array and dictionary literals, use type annotation.

**Preferred:**

|  |
| --- |
| var names: [String] = [     "rob",     "jason",     "nop",     "gene",  ]  var lookup: [String: String] = [     "gene": "san mateo",     "rob": "menlo park",     "jason": "oakland",     "nop": "san francisco",  ] |

~~Not Preferred:~~

|  |
| --- |
| var names = [String]()  var lookup = [     "gene": "san mateo",     "rob": "menlo park",     "jason": "oakland",     "nop": "san francisco",  ] |

\*\***Note**: Following this guideline 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> |

**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.

**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.

**Weak Self**

Use `self` instead of weakSelf

Extend object lifetime using the [weak self] and guard let `self` = 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 { [weak self] response in     guard let `self` = self else {        Return     }     let model = strongSelf.updateModel(response)     self.updateUI(model)  } |

~~Not Preferred~~

|  |
| --- |
| // might crash if self is released before response returns  resource.request().onComplete { [unowned self] response in     let model = self.updateModel(response)     self.updateUI(model)  }  // deinit could happen between updating the model and updating UI  resource.request().onComplete { [weak self] response in     let model = self?.updateModel(response)     self?.updateUI(model)  } |

**Access Control**

Use fileprivate when necessary.

Full access control annotation in tutorials can distract from the main topic and is not required. Using private and fileprivate appropriately, however, adds clarity and promotes encapsulation. Prefer private to fileprivate when possible. Using extensions may require you to use fileprivate.

Only explicitly use open, public, and internal when you require a full access control specification.

Use access control as the leading property specifier. The only things that should come before access control are the static specifier or attributes such as @IBAction, @IBOutlet and @discardableResult.

**Preferred:**

|  |
| --- |
| private let message = "Great Scott!"  class TimeMachine {     fileprivate dynamic lazy var fluxCapacitor = FluxCapacitor()  } |

~~Not Preferred:~~

|  |
| --- |
| fileprivate let message = "Great Scott!"  class TimeMachine {     lazy dynamic fileprivate var fluxCapacitor = FluxCapacitor()  } |

**Control Flow**

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

**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)  } |

~~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  } |

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:**

|  |
| --- |
| func computeFFT(context: Context?, inputData: InputData?) throws -> Frequencies {     guard let context = context else { throw FFTError.noContext        return     }     guard let inputData = inputData else {        throw FFTError.noInputData }     // use context and input to compute the frequencies     return frequencies  } |

~~Not Preferred:~~

|  |
| --- |
| func computeFFT(context: Context?, inputData: InputData?) throws -> Frequencies {     if let context = context {        if let inputData = inputData {           ... // do something after cascading all the way to the right side |

**Failing Guards**

Guard statements are required to exit in some way. Generally, this should be simple one line statement such as return, 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.

**Semicolons**

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

**Parameter Ordering**

The ordering of parameters should be:

1. Value Types
2. Class Types
3. Value Types with Default Arguments
4. Class Types with Default Arguments
5. Primary Completion Blocks
6. Secondary Completion Blocks

|  |
| --- |
| func methodDeclaration(     rawTypes: Int,     objectTypes: Object,     rawTypesWithDefaultParameters: Int = 0,     objectTypesWithDefaultParameters: Object? = nil,     primaryCompletionBlocks: PrimaryCompletionBlock,     secondaryCompletionBlocks: SecondaryCompletionBlock) { |

File Structure Layout

The layout of a file should be:

1. Views
2. Other Properties
3. Computed Properties
4. Initializer
5. Actions
6. Highest Order Superclass Overrides
7. Other Superclass Overrides
8. Protocol Conformance

\*\*Note: The location of helper functions depends on what the purpose of the helper function is. For example: a method like func layoutConstraints() should be placed next to the initializer because the init method is the stage of a view’s lifecycle in which constraints are supposed to be applied.