Swift Style Guide

The standard template of ".swiftlint.yml" file

.swiftlint.yml

```
disabled_rules: #
  - colon
  - comma
  - control_statement
  - vertical_whitespace
  - trailing_whitespace
  - trailing_newline
  - identifier_name
  - function_parameter_count
opt_in_rules: #
  - empty_count
  - missing_docs
  # :
  # swiftlint rules
included: #  linting  `--path`
  - {swiftlint}
excluded: #  linting   `included`
  #- Example/Carthage
  - {swiftlint}

#
#
force_cast: warning #
force_try:
  severity: warning #
#
#
line_length: 160
#
type_body_length:
  - 300 # warning
  - 400 # error
function_body_length:
  - 80 # warning
  - 120 # error
cyclomatic_complexity:
  - 30
#
file_length:
  warning: 500
  error: 1200
# /
```

```
#
type_name:
  min_length: 4 #
  max_length: #
    warning: 40
    error: 50
  excluded: iPhone #
identifier_name:
  min_length: #
    error: 4 #
  excluded: #
    - id
    - URL
    - GlobalAPIKey
```

```
reporter: "xcode" #  (xcode, json, csv, checkstyle, junit, html, emoji)
```

Swift Style Guide

> *Raywenderlich Swift Style Guide*

Treat Warnings as Errors

“Treat Warnings as Errors”Warnning Free

Correctness

`#selector` `Selector("selectorStringName")`

Naming

“”Apple API Design Guidelines:

- “”“”
- “”
- “camel case” (not snake case)
- methodclassprotocoletc
- weak type`weakSelf`
- `make`
- methods
  - methodverb“-ed, -ing”
  - methodnounformX
  - booleanassertions
  - protocolsprotocol`what something` isprotocolnoun
  - protocolsprotocol`a kind of capability`protocol-able-ible
- termssurprise expertsconfuse beginners
- abbreviations
-
- methodsdelegates
- closuretuplelabeling
- default parameters

Class PrefixesClass

SwiftMS

```
import SomeModule

let myClass = MyModule.UsefulClass()
```

DelegatesDelegates

delegate methodsparameterdelegate source UIKit

Preferred:

```
func namePickerView(_ namePickerView: NamePickerView, didSelectName name: String)
func namePickerViewShouldReload(_ namePickerView: NamePickerView) -> Bool
```

Not Preferred:

```
func didSelectName(namePicker: NamePickerViewController, name: String)
func namePickerShouldReload() -> Bool
```

Use Type Inferred ContextSwift

Use compiler inferred context to write shorter, clear code. (Also see Type Inference.)

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

upper camel case `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<T> { ... }
func write<target: OutputStream>(to target: inout target)
func swap<Thing>(_ a: inout Thing, _ b: inout Thing)
```

Language

US EnglishAppleAPI

Preferred:

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

Not Preferred:

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

Code Organization

extensionsextension `// MARK-`

Protocol Conformance Protocol

modelprotocolprotocol methodsextensionmethodsprotocol

Preferred:

```
class MyViewController: UIViewController {
  // class stuff here
}

// MARK: - UITableViewDataSource
extension MyViewController: UITableViewDataSource {
  // table view data source methods
}

// MARK: - UIScrollViewDelegate
extension MyViewController: UIScrollViewDelegate {
  // scroll view delegate methods
}
```

Not Preferred:

```
class MyViewController: UIViewController, UITableViewDataSource, UIScrollViewDelegate {
  // all methods
}
```

UIKit view controllersclassmethodslifecycle, custom accessors, IBActionclass extensions

Unused Code

Unused code, Xcode template codeplaceholder comments

methodsAspirational methodssuperclassmethodsXcodeUIApplicationDelegatemethods

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
}

override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
  // #warning Incomplete implementation, return the number of rows
  return Database.contacts.count
}
```

Minimal Importsimport

impoortmodules`Foundation`import `UIKit`import `UIKit`import `Foundation`

Preferred:

```
import UIKit
var view: UIView
var deviceModels: [String]
```

Preferred:

```
import Foundation
var deviceModels: [String]
```

Not Preferred:

```
import UIKit
import Foundation
var view: UIView
var deviceModels: [String]
```

Not Preferred:

```
import UIKit
var deviceModels: [String]
```

Spacing

- IndentsSpacestabs
- indent4spaceindent2spaces
- braces(`if`/`else`/`switch`/`while` etc.)

Preferred:

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

Not Preferred:

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

- Colons `:`, always have no space on the left and one space on the right. Exceptions are the ternary operator `? :`, empty dictionary `[:]` and `#selector`syntax `addTarget(_:action:)`.

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]
}
```

- 160spaceindent

Comments

`self-documenting`**PulicOpen**methodclass

C-style(`/* ... */`)`/////`

Classes and Structures

Which one to use?classstruct

, structsvalue semantics structidentity[abc][abc]SwiftArraystructDictionaryString

Classes reference semantics. classidentityPersonclasspersonperson195033195033identitySwiftDatestruct

Objective CstructclassNSDate, NSSet

Example definitionClass

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
  }

  convenience init(x: Int, y: Int, diameter: Double) {
    self.init(x: x, y: y, radius: diameter / 2)
  }

  override func area() -> Double {
    return Double.pi * radius * radius
  }
}

extension Circle: CustomStringConvertible {
  var description: String {
    return "center = \(centerString) area = \(area())"
  }
  private var centerString: 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.
- Organize extra functionality (e.g. printing) in extensions.
- Hide non-shared, implementation details such as `centerString` inside the extension using `private` access control.

Use of Self

`selfself@escaping` closuresinitializers

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
  }
}
```

Finalfinal

“final”finalBox“”

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

Function Declarations

+{:

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

+label:

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

Don't use `(Void)` to represent the lack of an input; simply use `()`. Use `Void` instead of `()` for closure and function outputs.

Preferred:

```
func updateConstraints() -> Void {
  // magic happens here
}

typealias CompletionHandler = (result) -> Void
```

Not Preferred:

```
func updateConstraints() -> () {
  // magic happens here
}

typealias CompletionHandler = (result) -> ()
```

Function Calls

:

```
let success = reticulateSplines(splines)
```

+label:

```
let success = reticulateSplines(
  spline: splines,
  adjustmentFactor: 1.3,
  translateConstant: 2,
  comment: "normalize the display")
```

Closure Expressions

closure expression parameter`trailing closure syntax``trailing closure`

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()
}
```

closuressingle-expression,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 }.index(of: 90)

let value = numbers
  .map {$0 * 2}
  .filter {$0 > 50}
  .map {$0 + 10}
```

Types

Always use Swift's native types and expressions 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)"                         // String
```

Less 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 drawing code, use `CGFloat` if it makes the code more succinct by avoiding too many conversions.

Constants

constants`let` keywordvariables`var` keyword

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.
Objective-C

Optionals

`nil`?optional

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()`. Prefer optional binding to implicitly unwrapped optionals in most other cases.

`optional chains`:

```
textContainer?.textLabel?.setNeedsDisplay()
```

`optional binding`variable`nil`:

```
if let textContainer = 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 whenever possible 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
}

// another example
UIView.animate(withDuration: 2.0) { [weak self] in
  guard let self = self else { return }
  self.alpha = 1.0
}
```

Not Preferred:

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

if let unwrappedSubview = optionalSubview {
  if let realVolume = volume {
    // do something with unwrappedSubview and realVolume
  }
}

// another example
UIView.animate(withDuration: 2.0) { [weak self] in
  guard let strongSelf = self else { return }
  strongSelf.alpha = 1.0
}
```

Lazy Initialization

Consider using lazy initialization for finer grained 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 = makeLocationManager()

private func makeLocationManager() -> CLLocationManager {
  let manager = CLLocationManager()
  manager.desiredAccuracy = kCLLocationAccuracyBest
  manager.delegate = self
  manager.requestAlwaysAuthorization()
  return manager
}
```

Notes:

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

Type Inference

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

Preferred:

```
let message = "Click the button"
let currentBounds = computeViewBounds()
var names = ["Mic", "Sam", "Christine"]
let maximumWidth: CGFloat = 106.5
```

Not Preferred:

```
let message: String = "Click the button"
let currentBounds: CGRect = computeViewBounds()
var 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.

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

FunctionsMethods“”FunctionsFree functionsnot attached to a class or typeMethodsMember methodsattached to a class or typeMethodsFunctionsMethodsFunctions

Free functionsmethod swizzlerfree function

Preferred

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

Not Preferred

```
let sorted = mergeSort(items)  // hard to discover
launch(&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
```

Memory Management

object graph`weak` and `unowned`

Objective C`class`Swift`struct``enum``struct``class`

Extending object lifetime

`[weak self]``guard let self = self else {return}``[weak self]` to `[unowned self]`Apple`[unowned self]``self`closure

`[weak self]``self`optional chaining

Preferred

```
resource.request().onComplete { [weak self] response in
  guard let self = self else {
    return
  }
  let model = self.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)
}
```

Not Preferred

```
// deallocate 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

`privatefileprivateprivatefileprivate`

`openpublic internal`

`@IBAction@IBOutlet@discardableResult`

Preferred:

```
private let message = "Great Scott!"

class TimeMachine {
  private dynamic lazy var fluxCapacitor = FluxCapacitor()
}
```

Not Preferred:

```
fileprivate let message = "Great Scott!"

class TimeMachine {
  lazy dynamic private var fluxCapacitor = FluxCapacitor()
}
```

Control Flow

`for-inwhile-condition-increment`

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
}
```

Ternary Operator

`?:`review

Preferred:

```
let value = 5
result = value != 0 ? x : y

let isHorizontal = true
result = isHorizontal ? x : y
```

Not Preferred:

```
result = a > b ? x = c > d ? c : d : y
```

Golden Path

conditional codes`if``guard`

Preferred:

```
func computeFFT(context: Context?, inputData: InputData?) throws -> Frequencies {

  guard let context = context else {
    throw FFTError.noContext
  }
  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 {
      // use context and input to compute the frequencies

      return frequencies
    } else {
      throw FFTError.noInputData
    }
  } else {
    throw FFTError.noContext
  }
}
```

`guard``if let`optionals`guard`if`else`

Preferred:

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

Not Preferred:

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

Failing Guards

Guard`return``throw``break``continue``fatalError()`code blocks`defer`

Semicolons

`;`Objective C

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

Parentheses

Parentheses.

Preferred:

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

Not Preferred:

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

In larger expressions, optional parentheses can sometimes make code read more clearly.

Preferred:

```
let playerMark = (player == current ? "X" : "O")
```

Multi-line String LiteralsString

`"""`stringpython

Preferred:

```
let message = """
  You cannot charge the flux \
  capacitor with a 9V battery.
  You must use a super-charger \
  which costs 10 credits. You currently \
  have \(credits) credits available.
  """
```

Not Preferred:

```
let message = """You cannot charge the flux \
  capacitor with a 9V battery.
  You must use a super-charger \
  which costs 10 credits. You currently \
  have \(credits) credits available.
  """
```

Not Preferred:

```
let message = "You cannot charge the flux " +
  "capacitor with a 9V battery.\n" +
  "You must use a super-charger " +
  "which costs 10 credits. You currently " +
  "have \(credits) credits available."
```

No Emoji

emojiemojicode

Organization and Bundle Identifier

Bundle Identifier `com.raywenderlich.{AppName}`, `AppName` .

Copyright Statement

Copyright:

```
/// Copyright (c) 2019 {} LLC
///
/// Permission is hereby granted, free of charge, to any person obtaining a copy
/// of this software and associated documentation files (the "Software"), to deal
/// in the Software without restriction, including without limitation the rights
/// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
/// copies of the Software, and to permit persons to whom the Software is
/// furnished to do so, subject to the following conditions:
///
/// The above copyright notice and this permission notice shall be included in
/// all copies or substantial portions of the Software.
///
/// Notwithstanding the foregoing, you may not use, copy, modify, merge, publish,
/// distribute, sublicense, create a derivative work, and/or sell copies of the
/// Software in any work that is designed, intended, or marketed for pedagogical or
/// instructional purposes related to programming, coding, application development,
/// or information technology.  Permission for such use, copying, modification,
/// merger, publication, distribution, sublicensing, creation of derivative works,
/// or sale is expressly withheld.
///
/// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
/// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
/// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
/// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
/// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
/// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
/// THE SOFTWARE.
```

References

- The Swift API Design Guidelines
- The Swift Programming Language
- Using Swift with Cocoa and Objective-C
- Swift Standard Library Reference