Style guide & coding conventions for Swift projects

Merge pull request **#42** from minsOne/linkKR …

joshaber authored on Jul 7 latest commit b71dd2bee6

CONTRIBUTING.md	Add CONTRIBUTING	a year ago
LICENSE	Initial commit	a year ago
README.md	Merge pull request #42 from minsOne/linkKR	3 months ago

README.md

A guide to our Swift style and conventions.

This is an attempt to encourage patterns that accomplish the following goals (in rough priority order):

1. Increased rigor, and decreased likelihood of programmer error
2. Increased clarity of intent
3. Reduced verbosity
4. Fewer debates about aesthetics

If you have suggestions, please see our contribution guidelines, then open a pull request. ⚡

Whitespace

- Tabs, not spaces.
- End files with a newline.
- Make liberal use of vertical whitespace to divide code into logical chunks.
- Don't leave trailing whitespace.
  - Not even leading indentation on blank lines.

Prefer `let` -bindings over `var` -bindings wherever possible

Use `let foo = …` over `var foo = …` wherever possible (and when in doubt). Only use `var` if you absolutely have to (i.e. you *know* that the value might change, e.g. when using the `weak` storage modifier).

Rationale: The intent and meaning of both keywords is clear, but *let-by-default* results in safer and clearer code.

A `let` -binding guarantees and *clearly signals to the programmer* that its value is supposed to and will never change. Subsequent code can thus make stronger assumptions about its usage.

It becomes easier to reason about code. Had you used `var` while still making the assumption that the value never changed, you would have to manually check that.

Accordingly, whenever you see a `var` identifier being used, assume that it will change and ask yourself why.

Avoid Using Force-Unwrapping of Optionals

If you have an identifier `foo` of type `FooType?` or `FooType!`, don't force-unwrap it to get to the underlying value (`foo!`) if possible.

Instead, prefer this:

```
if let foo = foo {
    // Use unwrapped `foo` value in here
} else {
    // If appropriate, handle the case where the optional is nil
}
```

Alternatively, you might want to use Swift's Optional Chaining in some of these cases, such as:

```
// Call the function if `foo` is not nil. If `foo` is nil, ignore we ever tried to ma
foo?.callSomethingIfFooIsNotNil()
```

Rationale: Explicit `if let`-binding of optionals results in safer code. Force unwrapping is more prone to lead to runtime crashes.

Avoid Using Implicitly Unwrapped Optionals

Where possible, use `let foo: FooType?` instead of `let foo: FooType!` if `foo` may be nil (Note that in general, `?` can be used instead of `!`).

Rationale: Explicit optionals result in safer code. Implicitly unwrapped optionals have the potential of crashing at runtime.

Prefer implicit getters on read-only properties and subscripts

When possible, omit the `get` keyword on read-only computed properties and read-only subscripts.

So, write these:

```
var myGreatProperty: Int {
    return 4
}

subscript(index: Int) -> T {
    return objects[index]
}
```

… not these:

```
var myGreatProperty: Int {
    get {
        return 4
    }
}

subscript(index: Int) -> T {
    get {
        return objects[index]
    }
}
```

Rationale: The intent and meaning of the first version is clear, and results in less code.

Always specify access control explicitly for top-level definitions

Top-level functions, types, and variables should always have explicit access control specifiers:

```
public var whoopsGlobalState: Int
internal struct TheFez {}
private func doTheThings(things: [Thing]) {}
```

However, definitions within those can leave access control implicit, where appropriate:

```
internal struct TheFez {
    var owner: Person = Joshaber()
}
```

Rationale: It's rarely appropriate for top-level definitions to be specifically `internal`, and being explicit ensures that careful thought goes into that decision. Within a definition, reusing the same access control specifier is just duplicative, and the default is usually reasonable.

When specifying a type, always associate the colon with the identifier

When specifying the type of an identifier, always put the colon immediately after the identifier, followed by a space and then the type name.

```
class SmallBatchSustainableFairtrade: Coffee { ... }

let timeToCoffee: NSTimeInterval = 2

func makeCoffee(type: CoffeeType) -> Coffee { ... }
```

Rationale: The type specifier is saying something about the *identifier* so it should be positioned with it.

Also, when specifying the type of a dictionary, always put the colon immediately after the key type, followed by a space and then the value type.

```
let capitals: [Country: City] = [ Sweden: Stockholm ]
```

Only explicitly refer to `self` when required

When accessing properties or methods on `self`, leave the reference to `self` implicit by default:

```
private class History {
    var events: [Event]
    func rewrite() {
        events = []
    }
}
```

Only include the explicit keyword when required by the language—for example, in a closure, or when parameter names conflict:

```
extension History {
    init(events: [Event]) {
        self.events = events
    }

    var whenVictorious: () -> () {
        return {
            self.rewrite()
        }
    }
}
```

Rationale: This makes the capturing semantics of `self` stand out more in closures, and avoids verbosity elsewhere.

Prefer structs over classes

Unless you require functionality that can only be provided by a class (like identity or deinitializers), implement a struct instead.

Note that inheritance is (by itself) usually *not* a good reason to use classes, because polymorphism can be provided by protocols, and implementation reuse can be provided through composition.

For example, this class hierarchy:

```
class Vehicle {
    let numberOfWheels: Int

    init(numberOfWheels: Int) {
        self.numberOfWheels = numberOfWheels
    }

    func maximumTotalTirePressure(pressurePerWheel: Float) -> Float {
        return pressurePerWheel * Float(numberOfWheels)
    }
}

class Bicycle: Vehicle {
    init() {
        super.init(numberOfWheels: 2)
    }
}

class Car: Vehicle {
    init() {
        super.init(numberOfWheels: 4)
    }
}
```

could be refactored into these definitions:

```
protocol Vehicle {
    var numberOfWheels: Int { get }
}

func maximumTotalTirePressure(vehicle: Vehicle, pressurePerWheel: Float) -> Float {
    return pressurePerWheel * Float(vehicle.numberOfWheels)
}

struct Bicycle: Vehicle {
    let numberOfWheels = 2
}

struct Car: Vehicle {
    let numberOfWheels = 4
}
```

Rationale: Value types are simpler, easier to reason about, and behave as expected with the `let` keyword.

Make classes `final` by default

Classes should start as `final`, and only be changed to allow subclassing if a valid need for inheritance has been identified. Even in that case, as many definitions as possible *within* the class should be `final` as well, following the same rules.

Rationale: Composition is usually preferable to inheritance, and opting *in* to inheritance hopefully means that more thought will be put into the decision.

Omit type parameters where possible

Methods of parameterized types can omit type parameters on the receiving type when they're identical to the receiver's. For example:

```
struct Composite<T> {
    …
    func compose(other: Composite<T>) -> Composite<T> {
        return Composite<T>(self, other)
    }
}
```

could be rendered as:

```
struct Composite<T> {
    …
    func compose(other: Composite) -> Composite {
        return Composite(self, other)
    }
}
```

Rationale: Omitting redundant type parameters clarifies the intent, and makes it obvious by contrast when the returned type takes different type parameters.

Use whitespace around operator definitions

Use whitespace around operators when defining them. Instead of:

```swift
func <|(lhs: Int, rhs: Int) -> Int
func <|<<A>(lhs: A, rhs: A) -> A
```

write:

```swift
func <| (lhs: Int, rhs: Int) -> Int
func <|< <A>(lhs: A, rhs: A) -> A
```

Rationale: Operators consist of punctuation characters, which can make them difficult to read when immediately followed by the punctuation for a type or value parameter list. Adding whitespace separates the two more clearly.

Translations

- 中文版
- 日本語版
- 한국어판