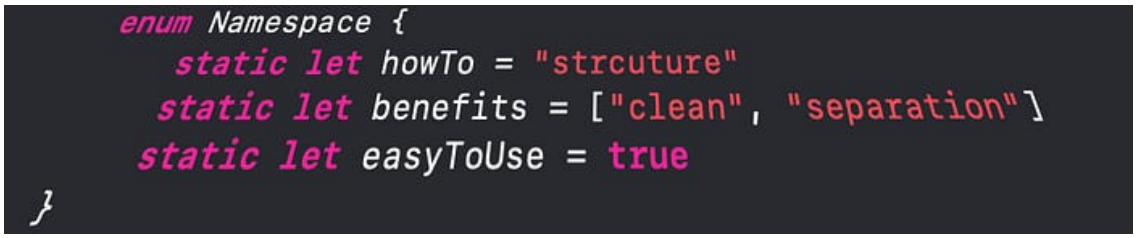


Structure constants in iOS with enums

Enums as namespaces in Swift

Sven Korset

Follow

Better Programming · ~4 min read · August 16, 2019 (Updated: December 11, 2021) · Free: No

Pieces of a scalable iOS app architecture

Finally, all views are finished and look exactly as required by the picky designers. Buttons are firing only at the first tap and no longer send 100 server requests, because the QA department always pushes a button like crazy. And everything else looks nice, too.

Of course, a small change comes five minutes before the end of the workday — all gaps should now be ten instead of eight pixels wide, and the debouncing should now be reduced from 500 to 300 milliseconds. In the project, 100 locations now must be found and adapted. And, of course, today! 😱

A wise programmer, of course, has no hard-coded values stored in the source code but only uses global constants. Two values exchanged, finished, evening saved. 🤓

There's no need to search 100 locations to see if the value "500" is really meant as a debouncing value or if that might not be a view

Only a central, no-brainer change.

How Do You Structure These Constants?

The most naive approach would be to simply write 1,000 `let` statements into a *const.swift* file. 🚨 Better still would be with their own prefix abbreviations in the name to prevent naming conflicts. 🚨🚨 Then you just have to find the right constant…

Is there not something better, e.g. *namespaces*? 🤔

> A namespace is a concept for separating code and preventing name conflicts.

Unfortunately, apart from modules, Swift itself has no namespaces. However, as Bart Jacobs explains nicely in "Namespaces in Swift" or Chung Duong in his post "Namespace in Swift", you can just use enums without cases. This comes closest to namespaces in Swift.

```swift
import UIKit

enum Const {
    enum Size {
        /// The default thickness of a separator.
        static let separator = CGFloat(1)
        /// The default gap size between two view elements.
        static let gap = CGFloat(8)
    }

    enum Duration {
        static let defaultDebounce = DispatchTimeInterval.milliseconds(300)
    }
}

let gapSize = Const.Size.gap
let debounceDuration = Const.Duration.defaultDebounce
```

namespace-const-allInOne.swift hosted with ❤️ by GitHub view r

Enums Are the Solution

Enums can be nested and expanded but without case definition not instantiated. That makes enums Swift's namespaces.

The declarations can even be further structured by moving each category into its own file. The root node of all constants then forms `Const`, e.g. in a file named *Const.swift*:

```swift
enum Const {
    // Specific constants will be defined in their extensions.
}
```

namespace-const.swift hosted with ❤️ by GitHub view r

Each additional category would then come as an extension in its own file, e.g. *Sizes.swift*:

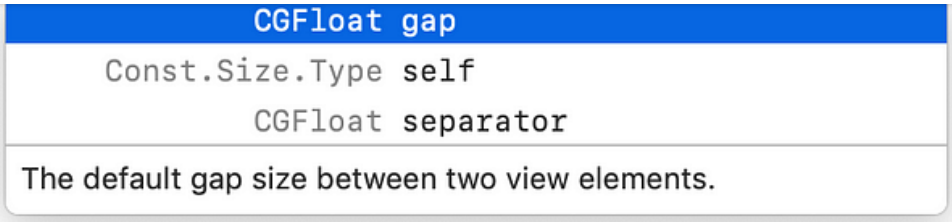
```swift
import UIKit

extension Const {
    enum Size {
        /// The default thickness of a separator.
        static let separator = CGFloat(1)
        /// The default gap size between two view elements.
        static let gap = CGFloat(8)
    }
}
```

namespace-sizes.swift hosted with ❤️ by GitHub view r

In this way, you bundle similar constants with each enum construct. This reduces potential merge conflicts. And by dividing them into different files, you keep the file size low and the overview high.

Access to a constant is then very easy as the autocompletion feature of Xcode only displays constants of this logical category.



Suboptimal Alternative

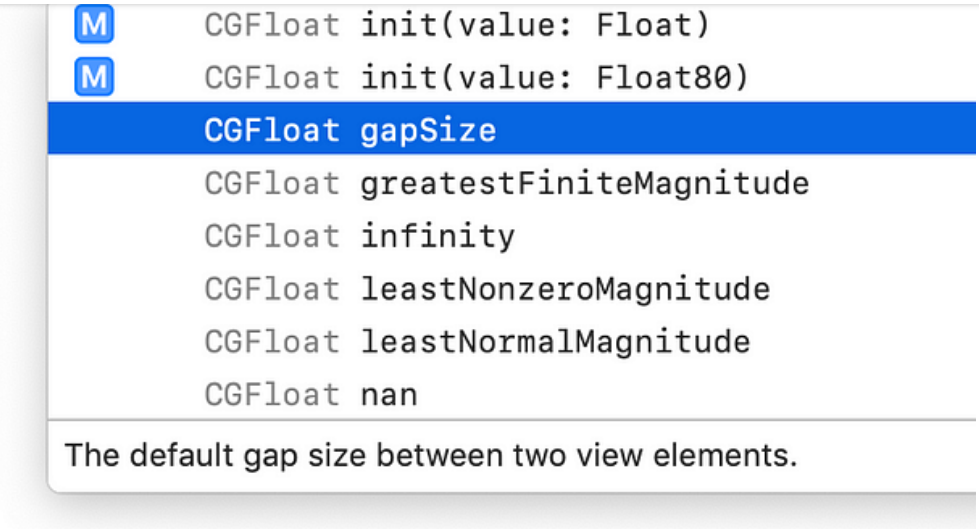
Of course, you could also write the constants in an extension of the associated return type and access the constant via the type instead of `Const.`.

```
1  extension CGFloat {
2      /// The default gap size between two view elements.
3      static let gapSize = CGFloat(8)
4  }
5
6  let gapSize: CGFloat = .gapSize
```

namespace-cgfloat.swift hosted with ❤️ by GitHub view r

That makes it shorter because you no longer need to write `Const.Size.`, but it may then come back to name collisions, e.g. if Apple introduces its own `gapSize` constant in *CGFloat*.

In addition, the overview suffers because it no longer distinguishes between self-defined constants and standard constants. The autocompletion feature then displays everything about the type.



Also, you have to know in advance what kind of a constant you are looking for.

Constants are then separated according to their type, e.g., a `static let defaultViewSize = CGSize(width: 100, height: 30)` would then be of type *CGSize* and not quite fit into the *CGFloat* extension anymore. However, it would still fit very well in the `Const.Size` extension. So, in this case, you have to know that `defaultViewSize` is of type *CGSize* but `gapSize` is of type *CGFloat*.

That's why I like the enum approach best and use it to structure constants in a scalable iOS app.

Namespaces for Non-Constants

If enums act as namespaces, could you also put all classes and other types in them? 🤔

```
4       class MyClass {
5       }
6   }
7
8   let obj = Namespace.MyClass()
```

namespace-class.swift hosted with ❤️ by GitHub view r

Of course it works, but there is a better way to do that: Modules. Simply outsource all classes and other types in frameworks. 😉

However, if you only want to bundle special types, it may make sense to use enums again. For example, in my DemoApp Project (DAP), I use a `Request` enum to bundle all requests and group the endpoints into other enums.

```
1   enum Request {}
2
3   extension Request {
4     enum SearchAutocompletion {
5       struct Query: ServerRequest { ... }
6     }
7   }
8
9   extension Request {
10    enum Authentication {
11      struct Login: ServerRequest { ... }
12      struct SignUp: ServerRequest { ... }
13      ...
14    }
15  }
16
17  let request = Request.SearchAutocompletion.Query()
18  serverWorker.send(request) { result in
19    // Do something with the result
20  }
```

namespace-request.swift hosted with ❤️ by GitHub view r

This has the advantage that autocompletion proposals match the endpoints (e.g., `Request.SearchAutocompletion`) and their requests (`Query`).

are located in their own framework.

Conclusion

As shown, one can simply refer to enums for better structuring of constants or types. It bundles affiliations and supports the autocompletion function of Xcode. No need for massive const files anymore.

This is a post from the "Pieces of a scalable iOS app architecture" series.

#swift #namespaces #ios #enum #programming