Extracted from:

iOS 10 SDK Development

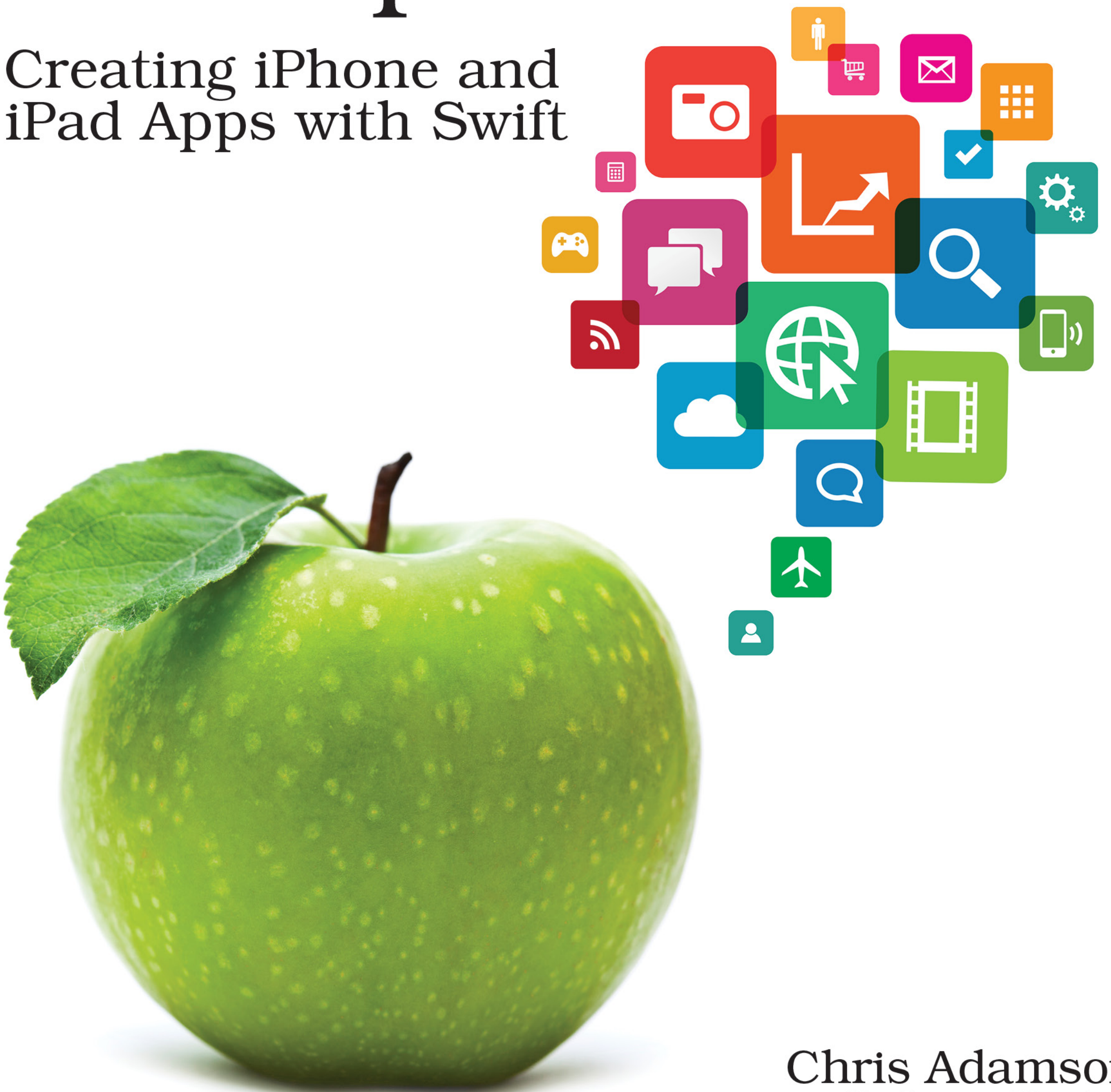
Creating iPhone and iPad Apps with Swift

The Pragmatic Bookshelf
Raleigh, North Carolina

The Pragmatic Programmers

iOS 10 SDK Development

Creating iPhone and iPad Apps with Swift



Chris Adamson
with Janie Clayton
edited by Rebecca Gulick

iOS 10 SDK Development

Creating iPhone and iPad Apps with Swift

Chris Adamson
with Janie Clayton

The Pragmatic Bookshelf
Raleigh, North Carolina

Pragmatic Bookshelf

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Executive Editor: Susannah Davidson Pfalzer
Development Editor: Rebecca Gulick
Indexing: Potomac Indexing, LLC
Copy Editor: Nicole Abramowitz
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-210-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—March 2017

Building Lightweight Structures

If we want to get away from object-oriented programming and try something different, we have to free ourselves of classes. In the next few sections, we'll do just that, and see that we're not losing anything in the transition.

To make a clean start, close this playground and create a new playground called StructsPlayground.

Let's think about the IOSDevice that we created as a class: it had some simple properties for the device name and dimensions, and some methods that operated on those properties. If it mostly serves as a container for data, if we don't care about inheritance, and if the data is small and not difficult to copy around in memory, then it's the kind of thing that functional programmers would tell us doesn't need to be a class.

So what's the alternative? In Swift, we have *structures*, which are lighter containers for properties. Let's remake IOSDevice as a struct to see how they work. Delete the default "Hello playground" line and define the IODevice structure as follows:

stylishswift/StructsPlayground.playground/Contents.swift
```
struct IOSDevice {
    var name : String
    var screenHeight : Double
    var screenWidth : Double
}
```

This is a lot like the beginning of our old class: it's just the property names and their types. One thing has changed, though: we can define these properties as non-optional types, and we don't get an error message about how "IOSDevice has no initializers." That's because the struct gets an initializer for free: just pass in all the values, labeled by their property names in the structure. That means we can create an iPhone7 like this:

stylishswift/StructsPlayground.playground/Contents.swift
```
let iPhone7 = IOSDevice(name: "iPhone 7", screenHeight: 138.1,
                        screenWidth: 67.0)
```

This shows an IOSDevice in the results pane, which means we've successfully created an instance of the IOSDevice structure. That was easy!

"But," the critic says, "you can't really *do* anything with it, can you?" Well, sure.

If this were C, our next step would probably be to write some global functions that work with this struct, either taking it as a parameter or returning it as a result. And the difference would be that the functions would receive copies of all the members of the structure, not just a reference to an object in memory (that some other part of the code might also be using, unbeknownst to us).

But still, Swift can do a lot better than just making us write a bunch of global functions.

Extensions

Swift gives us the ability to attach code to arbitrary types: structures, classes, enumerations, and even numeric types. The bits of code are called *extensions*, and they're delightfully powerful. Let's use them to beef up our IOSDevice.

To extend a type, we just write extension and the type we are extending, and then in curly braces we put code for methods or computed properties. This goes *outside* the struct's curly braces. So we can give the IOSDevice structure the screenArea computed property that the class had like this:

stylishswift/StructsPlayground.playground/Contents.swift

```
extension IOSDevice {
    var screenArea : Double {
        get {
            return screenWidth * screenHeight
        }
    }
}
```

Now just call that with iPhone7.screenArea on a new line, and we'll see 9,252.7 (or perhaps 9252.699…) in the results pane.

The fact that we write the code as an extension outside the type's definition implies something very powerful: *we can provide extensions for anything*. We're not limited to extending the abilities of our own classes and structures; we can extend classes in UIKit, basic types in Swift, basically any named type. As a rather absurd example, we can add methods to Swift's Int type:

stylishswift/StructsPlayground.playground/Contents.swift

```
extension Int {
    func addOne() -> Int {
        return self + 1
    }
}
```

And then we would call this with 41.addOne() to get 42.

Joe asks:

Why Is the Keyword func When It's Not Really a Function?

The keyword func is so named because Swift does indeed have honest-to-goodness functions: executable segments of code that take parameters and can return a value, but that aren't attached to an instance of anything. We've been using these already: the print() function is a global function that we've used to log messages to the Xcode console pane.

Defining a function is just like creating a method, just outside the scope of a class. Putting it inside the class makes it a method. Really, Swift methods are like a special case of functions: being inside a class, they pick up the stuff inside the class and are able to access its properties and other methods.

In fact, both functions and methods are a special case of the even more general-purpose concept of "closures," but we'll hold off talking about them until Chapter 7, *Handling Asynchronicity with Closures*, on page ?.

Extensions and Protocols

We don't have to put all our code to extend a given type into one extension block; it's OK to use several. This is helpful when we split up our code more purposefully. For example, let's get back our nice description string to log the name and dimensions of an IOSDevice. When we were writing a class, we implemented the CustomStringConvertible protocol. With a struct, we just provide another extension that conforms to the protocol.

stylishswift/StructsPlayground.playground/Contents.swift

```
extension IOSDevice : CustomStringConvertible {
    var description: String {
        return "\(name), \(screenHeight) x \(screenWidth)"
    }
}
```

Notice that for a read-only computed property, we can omit the get {...} and just provide the code to compute the property in curly braces right after the variable declaration.

Once you write this extension, the IOSDevice gets a nicer representation in the results pane, just like before.

In fact, extensions can be used to extend protocols themselves: an extension can declare new functions, methods, and properties to implement, and can even provide default implementations. Used in this way, it's called a *protocol* *extension*, and gives us another way to provide object-oriented traits like abstraction and extensibility to simpler types, without classes.

Listing Possibilities with Enumerations

Structures are familiar to old C programmers, and they were available in Objective-C but were so limited that they were often ignored in favor of classes. Swift extensions tilt the balance back toward structs, as the language does with another old C type: *enumerations*. The enum is a type that enumerates all its possible values. It's nice for times when you want to know there are a small number of valid values for something, like the suits of playing cards, positions in a team sport, and so forth.

Start a new playground called EnumsPlayground, and delete the "Hello playground" line. We're going to use this playground to rethink our IOSDevice.

So far, whether class or struct, we've assumed our IOSDevice is a touchscreen device like an iPhone, iPad, or iPod touch. But that's not necessarily so, is it? The Apple TV is technically an iOS device, and we currently have no way to account for its lack of a screen, short of turning screenHeight and screenWidth into optionals (which will be a hassle for callers), or using 0.0 flag values, which would just be ugly. Surely, we can do better.

Swift's enumerations give us an elegant solution to this problem. We can define a ScreenType enumeration to indicate what kind of screen the device has. Currently that would be "Retina" or "none," and we can extend it if, say, the iPhone 9 employs a pop-up hologram or something.

Define our ScreenType enumeration like this:

stylishswift/EnumsPlayground.playground/Contents.swift
```
enum ScreenType {
    case none
    case retina (screenHeight: Double, screenWidth: Double)
}
```

The different values for the enumeration are marked off as separate cases, kind of like a switch statement. What's really interesting here is the retina case. The two Doubles in parentheses are called *associated values*, and only exist when a given ScreenType is retina. The none case has no associated values, and some other case might have completely different associated values; maybe a hypothetical case crystalBall would have a radius: Double for its associated value.

Now let's create a new struct that can use this enum to represent its display, or lack thereof:

stylishswift/EnumsPlayground.playground/Contents.swift
```swift
struct IOSDevice {
    var name : String
    var screenType : ScreenType
}
```

That's easy enough; the enum acts like a new type, just like a String or Int. Now let's create some instances of this:

stylishswift/EnumsPlayground.playground/Contents.swift
```swift
Line 1 let iPhone7 = IOSDevice(name: "iPhone 7",
     2                         screenType: ScreenType.retina(
     3                             screenHeight: 138.1, screenWidth: 67.0))
     4 let appleTV4thGen = IOSDevice(name: "Apple TV (4th Gen)",
     5                               screenType: ScreenType.none)
```

Notice that just like with the struct, we automatically pick up the syntax for populating the associated values of the ScreenType.retina case; we just label and provide a value for each one, comma-separated, in parentheses (see line 3).

Concise Swift

Swift likes concision, and many things that are redundant can be omitted. For example, the screenType variable in these initializers can only be of type ScreenType, so it's legal to omit the type and just write the value with the leading dot character. So we could create the appleTV4thGen like this:

stylishswift/EnumsPlayground.playground/Contents.swift
```swift
let appleTV4thGen = IOSDevice(name: "Apple TV (4th Gen)",
                              screenType: .none)
```

As opportunities for omitting syntax occur throughout the book, we'll generally spell it out the long way first, mention what can be left out, and use the concise version from then on.

Using Associated Values in Enumerations

As before, the results pane evaluates the two instances of our new structure as just IOSDevice, because we no longer have a CustomStringConvertible implementation to provide a pretty string for them. We can provide one with an extension, and in the process we'll see how to use the associated values we provided for the screenHeight and screenWidth.

When we work with enumerations, we almost always need to use a switch statement to pick apart the possible cases. switch and enum go together perfectly, since they make it clear that we are walking through each possible value of the enum with case statements. In fact, Swift requires that the switch be exhaustive, meaning that it handles every possible case of the enum. There are only two in ScreenType, so it's easy here; with lots of cases, we could use default at the end of the switch to deal with otherwise-unhandled cases.

So to implement the description method, we need to return a string. It will include the name of the device and, only if it has a .retina screen type, the dimensions of the screen. Here's how we can do that with a switch:

stylishswift/EnumsPlayground.playground/Contents.swift

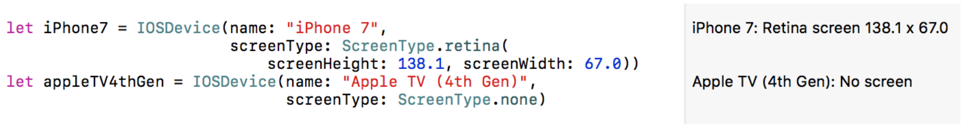
```
Line 1  extension IOSDevice : CustomStringConvertible {
     -      var description : String {
     -          var screenDescription: String
     -          switch screenType {
     5          case .none:
     -              screenDescription = "No screen"
     -          case .retina (let screenHeight, let screenWidth):
     -              screenDescription = "Retina screen " +
     -                  "\(screenHeight) x \(screenWidth)"
    10          }
     -          return "\(name): \(screenDescription)"
     -      }
     -  }
```

We begin on line 1 with an extension that says we are going to make IOSDevice conform to CustomStringConvertible. This means defining a description computed variable of type String.

Our description should provide a different string based on whether or not we have a screen. We declare the screenDescription on line 3. Notice that this is not an optional, yet we haven't provided a value for it; Swift lets us get away with this if it can tell that we are providing a value in all cases before the value is read. We start the switch on line 4, and the .none case that starts on line 5 is easy: we can just set screenDescription to "No Screen".

The interesting part is on line 7, which starts the .retina case. We use the let keyword in parentheses to receive the associated values as local variables. If we didn't care about one or more of these values, we could use the underscore character _ instead of let and the local variable name to say "I don't need this value." But in this case, we want both the screenHeight and screenWidth as local variables, so we can build a screenDescription that shows the dimensions.

Once this is written, the playground will immediately rebuild and rerun our playground, and the IOSDevice, now that it conforms to CustomStringConvertible, will pretty-print nice descriptions for our two devices in the evaluation pane:



```
let iPhone7 = IOSDevice(name: "iPhone 7",
                        screenType: ScreenType.retina(
                            screenHeight: 138.1, screenWidth: 67.0))
let appleTV4thGen = IOSDevice(name: "Apple TV (4th Gen)",
                              screenType: ScreenType.none)
```

```
iPhone 7: Retina screen 138.1 x 67.0

Apple TV (4th Gen): No screen
```

Swift's So Functional!

So, between structs, enums, protocols, and especially extensions, we can replicate most of the power of object-oriented programming, without needing to use classes and the usual practices of maintaining and mutating state. Swift isn't the most pure functional programming language by a long shot, but functional programming (FP) fans have found much to like in it.

Keep in mind that many of the iOS frameworks are very object-oriented in nature—they were written for use with Objective-C, after all—so much of the code we write will be of an OO style by necessity. Having said that, when we see an option to do things with a lighter touch, we'll try to do so.

Optionals Are Enumerations!

Here's a nifty little implementation detail that sometimes turns out to be useful: optionals are actually enumerations! An optional type is an enum with two cases, called .none and .some. The .none case is where the optional is nil, whereas .some has an associated value: the unwrapped value of the optional.

Some developers use this as a means of performing logic on the optional, particularly if we want to do something in the nil case. Rather than doing an if let and then testing the value against some other logic, we can put the logic in a switch like this:

stylishswift/EnumsPlayground.playground/Contents.swift
```
let optionalString : String? = "iPhone7"
switch optionalString {
case .none:
    print ("nil!")
case .some(let value):
    print ("some! \(value)")
}
```