Extracted from:

iOS 9 SDK Development

Creating iPhone and iPad Apps with Swift

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

iOS 9 SDK Development

Creating iPhone and iPad Apps with Swift

Chris Adamson
with Janie Clayton

edited by Rebecca Gulick

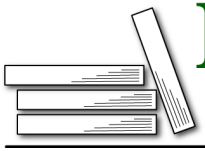
iOS 9 SDK Development

Creating iPhone and iPad Apps with Swift

Chris Adamson with Janie Clayton

The Pragmatic Bookshelf

Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Rebecca Gulick (editor)
Potomac Indexing, LLC (index)
Liz Welch (copyedit)
Dave Thomas (layout)
Janet Furlow (producer)

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: 978-1-68050-132-2
Encoded using the finest acid-free high-entropy binary digits.
Book version: P2.0—August 2016

CHAPTER 3

Swift with Style

In the previous chapter, we explored the basics of Swift: the type system, control flow, optionals, and so on. And, assuming Swift isn't your first programming language, you've probably guessed the next step is combining these simple pieces together into more complex, more capable, and more interesting constructs. While that is what we're going to do, it's not as straightforward as you might think.

Swift is a remarkably flexible language, one that takes its inspiration from a number of different sources. It's true to both the object-oriented nature of Objective-C and to new ideas about design, elegance, and maintainability in functional programming languages. You can write Swift like Objective-C, like C, like Java, or even like Haskell, and it will still work.

Since there's no one right way to write Swift, we will be making choices about how we want to organize our code. In this chapter, we're going to look at what Swift offers us for building bigger data structures, and how our choices will affect the evolution of our apps as we write and rewrite them. If the one hammer in your toolbox when you started this book was the good ol' class, let's discover what we can do by taking lightweight types like structures and enumerations and extending them with custom functionality.

Creating Classes

Many programmers—professionals and students, hobbyists and cowboy coders—have grown up in the mind-set of *object-oriented programming*. As Janie once said on the NSBrief podcast, "I didn't think I was learning object-oriented programming. I thought I was learning programming…like that was the only way to do it."

And it's not like anyone's wrong to learn OO! It's the dominant paradigm for a good reason: it has proven over the decades to be a good way to write applications. Whole languages are built around the concepts of OO: it's nigh-impossible to break out of the OO paradigm in Java, and Objective-C has OO in its very name, after all!

So let's see how Swift supports object-oriented programming. The heart and soul of OO is to create *classes*, collections of common behavior from which we will create individual instances called *objects*. We'll begin by creating a new playground called ClassesPlayground, and deleting the "Hello, playground" line as usual.

In the last chapter's collections examples, we used arrays, sets, and dictionaries to represent various models of iOS devices. But it's not easy or elegant to collect much more than a name that way, and there are lots of things we want in an iOS device model. So we will create a class to represent iOS devices.

We'll start by tracking a device's model name and its physical dimensions: width and height. Type the following into the playground:

stylishswift/ClassesPlayground.playground/Contents.swift

```
class IOSDevice {
        var name : String
        var screenHeight : Double
        var screenWidth : Double
}
```

In Swift, we declare a class with the class keyword, followed by the class name. If we were subclassing some other class, we would have a colon and the name of the superclass, like class MyClass : MySuperclass, but we don't need that for this simple class.

Next, we have *properties*, the variables or constants associated with an object instance. In this case, we are creating three variables: name, screenHeight, and screenWidth.

There's just one problem: this code produces an error. We need to start thinking about how our properties work.

Properties

The error flag tells us "Class IOSDevice has no initializers," and the red-circle instant-fix icon offers three problems and solutions. The problem for each is that there is no initial value for these properties. Before accepting the instant fix, let's consider what the problem is.

The properties we have defined are not optionals, so, by definition, they must have values. The tricky implication of that is that they must *always* have values. The value can change, but it can't be absent: that's what optionals are for.

We have a couple of options. We could accept the instant-fix suggestions and assign default values for each. That would give us declarations like

```
var name : String = ""
var screenHeight : Double = 0.0
var screenWidth : Double = 0.0
```

That's one solution, as long as we're OK with the default values. But here they don't quite make sense because we probably never want an iOS device with an empty string for a name.

Plan B: we can make everything optionals. To do this, we append the optional type ? to the properties.

```
var name : String?
var screenHeight : Double?
var screenWidth : Double?
```

Again, no more error, so that's good. Problem now is that any code that wants to access these properties has to do the if let dance from the last chapter to safely unwrap the optionals. And again, do we ever want the device name to be nil? That seems kind of useless.

Fortunately, we have another alternative: Swift's rule is that all properties must be initialized *by the end of every initializer*. So we can write an *initializer* to take initial values for these properties, and since that will be the only way to create an IOSDevice, we can know that these values will always be populated.

So rewrite the class like this:

stylishswift/ClassesPlayground.playground/Contents.swift

```
Line 1  class IOSDevice {
        -   var name : String
        -   var screenHeight : Double
        -   var screenWidth : Double
        5
        -   init (name: String, screenHeight: Double, screenWidth: Double) {
        -     self.name = name
        -     self.screenHeight = screenHeight
        -     self.screenWidth = screenWidth
        10  }
        -   }
```

The initializer runs from lines 6 to 10. The first line is the important one, as it starts with init and then takes a name and type for each of the parameters to be provided to the initializer code. In the initializer itself, we just use the self keyword to assign the properties to these arguments.

To create an instance of IOSDevice, we call the initializer by the name of the class, and provide these arguments by name. Create the constant iPhone6 after the class's closing brace, as follows (note that a line break has been added to suit the book's formatting; it's OK to write this all on one line).

stylishswift/ClassesPlayground.playground/Contents.swift

```
let iPhone6 = IOSDevice(name: "iPhone 6",
  screenHeight: 138.1, screenWidth: 67.0)
```

Congratulations! You've instantiated your first custom object, as the "IOSDevice" in the results pane indicates. Notice that the names of the arguments to the initializer are used as labels in actually calling the initializer. This helps us keep track of which argument is which, something that can be a problem in other languages when you call things that have lots of arguments.

Computed Properties

The three properties we've added to our class are *stored properties*, meaning that Swift creates the in-memory storage for the String and the two Doubles. We access these properties on an instance with dot syntax, like iPhone6.name.

Swift also has another kind of property, the *computed property*, which is a property that doesn't need storage because it can be produced by other means.

Right now we have a screenWidth and a screenHeight. Obviously, it would be easy to get the screen's area by just multiplying those two together. Instead of making the caller do that math, we can have IOSDevice expose it as a computed property. Back inside the class's curly braces—just after the other variables and before the init() is the customary place for it—add the following:

stylishswift/ClassesPlayground.playground/Contents.swift

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

Back at the bottom of the file, after creating the iPhone6 constant, fetch the computed property by calling it with the same dot syntax as with a stored property:

```
iPhone6.screenArea
```

The results pane shows the computed area, 9,252.7 (or possibly 9252.699...).

With only a get block, the screenArea is a read-only computed property. We could also provide a set, but that doesn't really make sense in this case.

It's also possible for stored properties to run arbitrary code; instead of computing values, we can give stored properties willSet and didSet blocks to run immediately before or after setting the property's value. We'll use this approach later on in the book.

Methods

Speaking of running arbitrary code, one other thing we expect classes to do is to let us, you know, *do stuff*. In object-oriented languages, classes have *methods* that instruct the class to perform some function. Of course, Swift makes this straightforward.

Let's take our web radio player from the first chapter and add that to our IOSDevice. After all, real iOS devices are used for playing music all the time, right? We'll start by adding the import statement to bring in the audio-video APIs, and the special code we used to let the playground keep playing. Add the following at the top of the file, below the existing import UIKit line:

stylishswift/ClassesPlayground.playground/Contents.swift

```
import AVFoundation
import XCPlayground
XCPlaygroundPage.currentPage.needsIndefiniteExecution = true
```

We need our IOSDevice to have an AVPlayer we can start and stop, so add that as a property after the existing name, screenHeight, and screenWidth:

stylishswift/ClassesPlayground.playground/Contents.swift

```
private var audioPlayer : AVPlayer?
```

Notice that this property is an optional type, AVPlayer?, since it will be nil until it is needed.

Now, let's add a method to the class. We do this with the func keyword, followed by the method name, a list of arguments, and a return type. Add this playAudio() method somewhere inside the class's curly braces, ideally after the init's closing brace, since we usually write our initializers first and our methods next.

stylishswift/ClassesPlayground.playground/Contents.swift

```
func playAudioWithURL(url: NSURL) -> Void {
  audioPlayer = AVPlayer(URL: url)
  audioPlayer!.play()
}
```

Like the init, the parentheses contain the parameters to the method and their types. By convention, we often imply the first parameter type in the name of the method. This is because when we call the method, we do *not* label the first parameter, but we do use labels for any other parameters. For example, if playAudioWithURL() also took a rate argument, we would call it like playAudioWithURL(someURL, rate: 1.0). Compared to some languages, the labeled parameters may seem chatty or verbose, but, in practice, they make the code more readable by exposing what each value is there for.

After the parameters, the return type is indicated by the -> arrow. In this case, the method returns nothing, so we return Void. (In fact, when we return Void we can omit the arrow and the return.) The rest of the method is the two lines of code we used in the first chapter to create the AVPlayer and start playing.

Now let's call it and start playing music. Put the following at the bottom of the file, after where we create the iPhone6 instance.

stylishswift/ClassesPlayground.playground/Contents.swift

```
if let url = NSURL(string: "http://armitunes.com:8010/listen.pls") {
  iPhone6.playAudioWithURL(url)
}
```

The first line attempts to create an NSURL out of the provided string. We use an if let because, if our string is garbage, what we get back from the initializer could be nil. This is because the NSURL provides a *failable initializer*, one that reserves the right to return nil instead of a new object. It's denoted this way in the documentation with the keyword init?, where the ? clues us in to the fact that optionals are in play.

Wrapping this in an if let means that we will only enter the curly-braced region if the initialization succeeds and assigns the value to the local variable url. This is the proper practice for failable initializers and gets around the bad practice we used in the first chapter when we just force-unwrapped the NSURL? optional with the ! operator.

And once we're safely inside the if let, we call the playAudioWithURL() method that we just wrote, and the music starts playing. If we wanted to write a proper stopAudio() method, that would look like this:

stylishswift/ClassesPlayground.playground/Contents.swift

```
func stopAudio() -> Void {
  if let audioPlayer = audioPlayer {
    audioPlayer.pause()
  }
  audioPlayer = nil
}
```

Again, we use an if let to safely unwrap the audioPlayer optional, and only if that succeeds do we pause() it. Then we can set audioPlayer back to nil.

Turn That Music Down

Remember that any change to the playground text will cause the contents to be rebuilt and rerun, which means that any change we make from here out will restart the audio. It's funny the first few times, but it gets annoying.

If you want to turn it off, just comment out the call to playAudioWithURL(). Swift uses the same comment syntax as all C-derived languages (Objective-C, C#, Java, etc.). That means you can either put // on the start of a line to turn it into a comment, or surround a whole range of lines with a starting /* and a closing */.