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

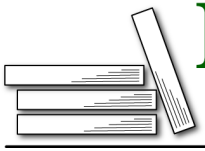
Chris Adamson
with Janie Clayton
edited by Rebecca Gulick

iOS 10 SDK Development

Creating iPhone and iPad Apps with Swift

Chris Adamson
with Janie Clayton

The Pragmatic Bookshelf
Raleigh, North Carolina



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:

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

CHAPTER 7

Handling Asynchronicity with Closures

Our player UI can play and pause audio, update the button label based on whether audio is playing or paused, and show a title. And in the last chapter, we exposed that functionality to unit testing, to make sure it keeps working. But there's one part of the UI we still haven't implemented: the label that shows the current playback time.

Thing is, time can be a real challenge for software. It's easy to write a series of instructions and have them executed in order. It's harder when things naturally happen at unpredictable times that we have to respond to, or when we want something to happen in the future, or if we have to respond to something and then do something.

We've seen two of iOS's older approaches to this: Timers to do work in the future (possibly repeatedly), and key-value observing (KVO) to respond to changes in supported properties. But both require somewhat clunky schemes to call back to designated objects, with special conventions for method names or parameter lists. It would be nice if there were something cleaner, so we could just say, "Every half-second, *do this*," or "When something special happens, *do that*."

Lucky for us, this cleaner approach—a Swift type that itself contains executable code—already exists, and we're going to put it to work in this chapter.

Understanding Closures

So let's think about the time display for our media player. Whenever media is playing, we want to periodically get the current playback time, and show that in the label as minutes and seconds…maybe hours, too, for those podcasts that won't wrap it up already. (You know who you are!)

In *Using a Timer*, on page ?, we learned how to use the Timer for our asynchronous tests, and it seems like that would work here, too. We could create a timer to periodically check on the player, get its current time, and update the label. That's fine, of course, although maybe a little wasteful if it keeps running when playback is paused, and there's extra code to write if we have to create a new timer when we start playing and destroy it when we pause.

Thinking about it, though, we didn't need a timer to change the Play/Pause button: that was based on an event we could observe from the player itself with KVO. So it's reasonable to think that AVPlayer could offer something appropriate for a playback time display.

If we look in the AVPlayer documentation, we find there's a discussion called "Timed State Observations," which says:

> KVO works well for general state observations, but isn't intended for observing continuously changing state like the player's time. AVPlayer provides two methods to observe time changes:

See? Just what we need! The section goes on to explain there are two methods to add these kinds of time observers—one for continuous observation, and another just for specific times, like reaching the end of the playing item. The first is what we need, so follow the link to the documentation for addPeriodicTimeObserver(forInterval:queue:using:). Now let's look at the declaration to see how we call it:

```
func addPeriodicTimeObserver(forInterval interval: CMTime,
  queue: DispatchQueue?,
  using block: @escaping (CMTime) -> Void) -> Any
```

What…the…heck?

OK, let's step back. This takes three arguments, and the first two are easy enough to understand: a CMTime with external name forInterval, and an optional DispatchQueue (whatever that is!) called queue. And the return type is an Any, so that's fine.

Obviously, the weird part is that third parameter, with external name using and internal name block. What's weird is its type: @escaping (CMTime) -> Void.

Set aside the @escaping for a moment, and consider what's left: (CMTime) -> Void. With the types on both sides of the arrow, that looks like a function or method declaration, right? Parameter types on the left, return type on the right?

That's pretty much what it is, in fact. This is the syntax for a *closure*, a self-contained block of functionality. Closures can take arguments, do work, and return a value…just like the functions and methods we're already used to.

But it's not that closures are a variation on functions; in fact, it's the other way around. Swift functions and methods are just special cases of closures! A closure is just some code represented as an object, and thus a function is a closure with a name, and then a method is a function associated with an instance of some type.

But closures are important because, as a Swift type, they can also be passed as parameters, stored in variables, and returned by functions and methods. We can pretty much do the same things with closures as we already do with Ints, Strings, and objects.

And using it as a type is what addPeriodicTimeObserver() is offering: we pass in a closure to be executed periodically—say, once every half-second—and the code in that closure gets repeatedly executed on that schedule. We don't have to use some special method name and parameter list like KVO's observeValue(forKeyPath:of:change:object:), and as a bonus, AVPlayer only calls this method when the media is playing.

Closures are perfect for our time label, so let's see how to use them.

Coding with Closures

To try out closures, we are going to call addPeriodicTimeObserver(forInterval:queue:using:), passing in a closure to call repeatedly when our podcast is playing. There's a little housekeeping we have to do for this approach: the docs say that the return value is an object of type Any that we will eventually provide to removeTimeObserver() to stop our updating. So, with the other properties near the top of ViewController.swift, add a property where we can hold on to this object. It'll need to be an optional, since we won't actually create it until long after init() is done.

closures/PragmaticPodcasts-7-1/PragmaticPodcasts/ViewController.swift

```
private var playerPeriodicObserver : Any?
```

We already cleaned up the player's KVO observer for the Play/Pause button in deinit(), so let's clean up this playerPeriodicObserver there, too, by adding the following:

closures/PragmaticPodcasts-7-1/PragmaticPodcasts/ViewController.swift

```
if let oldObserver = playerPeriodicObserver {
  player?.removeTimeObserver(oldObserver)
}
```

Notice that since playerPeriodicObserver is an optional, and removeTimeObserver() takes a non-optional parameter, we carefully unwrap with an if let.

A Simple Closure

Now we're ready to add the periodic observer. We'll do that in set(url:), where we currently create the player and set up the observer. For the moment, let's just log a message in the closure, before we worry about updating the UI.

closures/PragmaticPodcasts-7-1/PragmaticPodcasts/ViewController.swift

```
Line 1 let interval = CMTime(seconds: 0.25, preferredTimescale: 1000)
  2 playerPeriodicObserver =
  3   player?.addPeriodicTimeObserver(forInterval: interval,
  4                                   queue: nil,
  5                                   using:
  6     { currentTime in
  7       print("current time \(currentTime.seconds)")
  8   })
```

Because addPeriodicTimeObserver() wants a CMTime to indicate how often we want our closure to run, we create one on line 2. Without getting too deeply into the Core Media framework, the idea of a CMTime instance is that it uses a timescale to represent how accurately it's keeping time. We don't need it to be super-accurate for a UI display, so we'll just update every quarter-second, keeping track of time in 1000ths of a second.

Lines 2-8 are one big call to addPeriodicTimeObserver(). Line 3 specifies the 0.25-second interval we just created. For the queue on line 4, the docs say we can pass nil for the default behavior, so that's what we'll do for now.

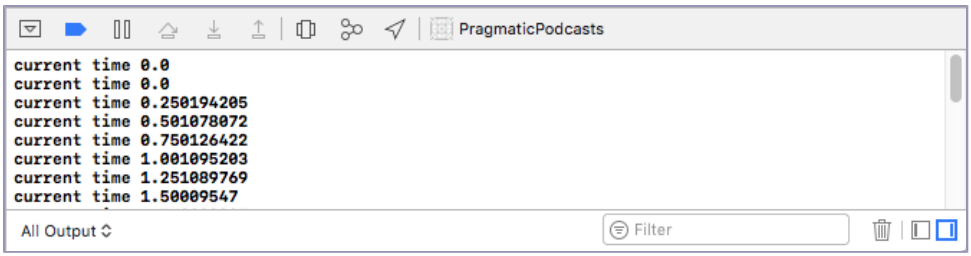
Finally, we have the using parameter on line 5. This takes our closure, which runs from lines 6 to 8. To write a closure, we use the syntax:

```
{ paramName1, paramName2, ... -> returnType in code... }
```

Simply put, the contents of a closure are a list of parameters, the arrow with a return type (omitted if none), the in keyword, and then executable code, all inside curly braces. We can choose whatever names we like for the parameters; in this case, the actual type of currentTime was defined as CMTime back in addPeriodicTimeObserver()'s declaration of its own using parameter.

So the closure receives a single parameter that we've called currentTime. To keep things simple, we'll just print() it, in seconds, on line 7.

Run the app, and click the Play button. In the console area at the bottom of the Xcode window—bring it up with ⇧⌘C or View > Debug Area > Activate Console, if it doesn't appear automatically—you'll see the log messages appear every 0.25 seconds or so as shown in the figure at the top of the next page. Hit Pause, and they'll stop, and then resume when you tap Play again.

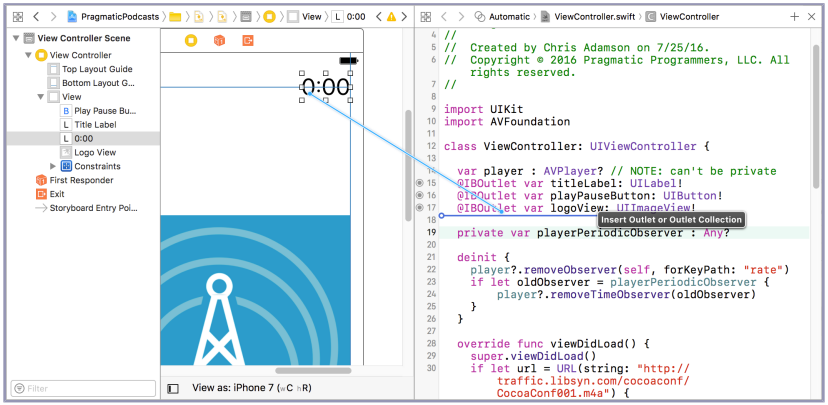


So, we're off and running, literally. We now have a simple block of code that will be called every 0.25 seconds when the podcast episode is playing. As a bonus, there's far less boilerplate than we had from setting up callback methods for KVO or Timers. Another advantage in Swift is that a closure can be created pretty much anywhere—in free functions, or methods on enums or structs, for example, whereas the callback approaches we saw earlier only work with full-blown objects.

Updating the Label from the Closure

Now we're ready to have our closure actually update the label with the current playback time. First things first, though: we don't currently have an outlet to the label, and we need one in order to change its text from code. We'll wire up a connection just like we did with the other UI elements.

Switch to Main.storyboard and select the 0:00 label. Bring up the Assistant Editor with the "two rings" toolbar button, or ⌥⌘↩. Make sure that ViewController.swift comes up as the Automatic selection in the right pane, and then Control-drag from the 0:00 label in the storyboard to the properties in the code. When you end the drag, a pop-up appears to fill in the details; give it the name timeLabel, and make sure the connection is "outlet," the type is UILabel, and the storage is "strong," and then click Connect.



Now we're ready to populate this label. Switch back to the Standard Editor (⌘↩) and return to ViewController.swift. Go down to the closure in set(url:). We could write all our label-updating code inside the closure, but we're already indented pretty far, so putting a bunch of code here is going to be kind of ugly. Instead, replace the print() line with the following method call:

closures/PragmaticPodcasts-7-1/PragmaticPodcasts/ViewController.swift

```
self.updateTimeLabel(currentTime)
```

For the moment, this is going to bring up an error because we haven't written the updateTimeLabel() method yet. But, more importantly, notice how we use self here. The closure has access to any variables currently in scope when the closure is created. Since self is available anywhere in the class, the closure can see it. Other variables local to set(url:), like url or interval, could be called too, if they were useful inside the closure. We call this *capturing* the variable.

Capture and Escape

The idea of a closure "capturing" a variable also explains the @escaping we saw back in the definition of addPeriodicTimeObserver(). This keyword is a signal that the closure will be held on to by the method or function receiving the closure, which in turn means that variables referenced by the closure will live on past the lifespan of the function call that receives the closure—addPeriodicTimeObserver() in this case—even a local variable that would otherwise disappear.

There's a corresponding @noescape that means variables captured by the closure *won't* be used after the function call that takes the closure. This lets the compiler make certain optimizations that aren't possible if the variable is going to hang around.

@escaping is by far the more common scenario, and it has an important side effect. When we refer to properties or methods from inside the closure, we explicitly have to use self, as we do here, to acknowledge that we know we're capturing self. Forgetting self in a closure is an easy mistake to make, but it's also easy to correct: you'll see an error telling you that you need to add self to "make capture semantics explicit."

Call to method 'updateTimeLabel' in closure requires explicit 'self.' to make capture semantics explicit

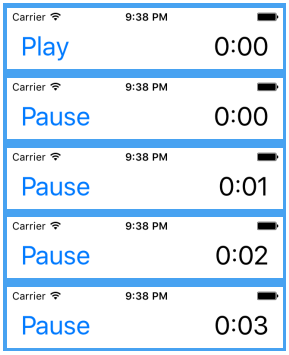
Now let's get this label to update its text by writing the missing updateTimeLabel() method. There's nothing closure-y about this; it's just some math and string formatting:

closures/PragmaticPodcasts-7-1/PragmaticPodcasts/ViewController.swift

```
Line 1  private func updateTimeLabel(_ currentTime: CMTime) {
2     let totalSeconds = currentTime.seconds
3     let minutes = Int(totalSeconds / 60)
4     let seconds = Int(totalSeconds.truncatingRemainder(dividingBy: 60))
5     let secondsString = seconds >= 10 ? "\(seconds)" : "0\(seconds)"
6     timeLabel.text = "\(minutes):\(secondsString)"
7   }
```

To format the string, we convert the CMTime into a total number of seconds, and then divvy that into minutes and seconds. The minutes are easy (just divide by 60), but the seconds are a little more obscure: Swift 3 eliminates the modulo operator (%) seen in many other languages, and instead requires us to use a method called truncatingRemainder(), as seen on line 4. With minutes and seconds computed, we figure out if the seconds need a leading "0" (line 5), and then set timeLabel's text to a colon-separated string.

And that's it! Run the app again, tap Play, and watch as the time counter counts up along with our playback.



We know from our earlier log statements that it doesn't bother updating when we're paused, and if we had a slider to skip around the podcast, the label would stay updated, since it's getting a new currentTime every quarter-second.