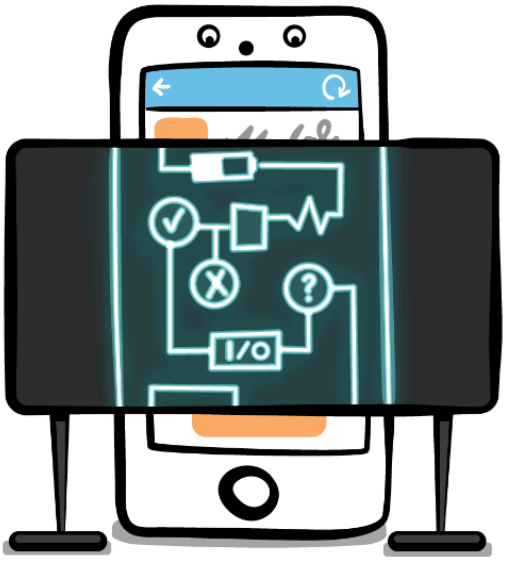
PRACTICAL INSTRUMENTS



HANDS-ON CHALLENGES

Practical Instruments

Luke Parham

Notice of Rights

Notice of Liability

Trademarks

Table of Contents: Overview

Table of Contents: Extended

Challenge #2: Image Caching

By Luke Parham

First things first, here's the link I promised:

Advanced Graphics and Animations for iOS Apps:

https://developer.apple.com/videos/play/wwdc2014/419/

And without further ado...

One thing you may have noticed, especially if you're on an old enough device, is that when you scroll back up, it oftentimes takes quite a while for existing images to appear in the cells. This is because we're actually doing this decode operation on every `UIImage` that gets set on our `AsyncImageViews`.

Since this operation is a bit time-consuming, we really shouldn't be so careless with it. Your challenge this time, is to implement a layer of caching for the decoded images so you do as little extra work as possible. To get started, go ahead and navigate to **AsyncImageView.swift**.

Adding a Global Cache

While this may be frowned upon by some people, the quickest and most convenient way to get a cache in place is to add a class level (or static) property to `AsyncImageView`.

The only trouble is that you can't actually add static properties to classes. Luckily, you can add them to `Structs`, which means you just need to get a little creative.

First, go ahead and add an extension to `AsyncImageView`:

```
extension AsyncImageView {
    struct Static {
        static var cache = NSCache<AnyObject, AnyObject>()
    }
```

```
}
```

Here, we just define a simple struct that has a static property of type `NSCache`. If you haven't used `NSCache` before, go ahead and check out the docs and/or the NSHipster article. The tl;dr is that `NSCache` is basically an `NSDictionary` that automatically clears itself out when your app receives a memory warning. Since these decoded jpegs are pretty big, and also disposeable, `NSCache` provides the perfect holder.

Next, we'll add our actual class property, which is really just a wrapper around the static struct property.

Add the following definition below the struct definition:

```
class var globalCache: NSCache<AnyObject, AnyObject> {
    get { return Static.cache }
    set { Static.cache = newValue }
}
```

Now you've got a static, globally accessible cache. Usually, shared global state is bad for a couple reasons, a big one being race conditions, but in this case `NSCache` is already thread-safe thanks to Apple, so you really don't have to worry too much about it.

Using the Cache

Alright, now that you've got your cache all set up, it's time to actually use it!

Using a cache is relatively straightforward. There are really only two scenarios where you care to look at it.

First, go to `decodedImage(_:)` and add the following lines right under the `guard`.

```
let cachedImage = AsyncImageView.globalCache.object(forKey: image)
if let cachedImage = cachedImage as? UIImage {
  return cachedImage
}
```

Here, you just start off the decode method by saying, "One moment, I may actually already have what you're looking for."

If you have already done the work, you return the decoded image and bail before you do any real work.

You may be saying, "Ok fine, but I don't remember ever putting anything in this cache..."

And you're right! That's the only other step.

Next, go to this line, where you create the decoded `UIImage`.

```
  let decodedImage = UIImage(cgImage: drawnImage)
```

And add the following line below it:

```
  AsyncImageView.globalCache.setObject(decodedImage, forKey: image)
```

Here, you're using the provided, encoded image as a key to the decompressed image. This way, any time you finish created a decompressed JPEG, you make sure to tuck it away in the cache just in case you need it later.