Extracted from:

iOS 8 SDK Development

Creating iPhone and iPad Apps with Swift

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

iOS 8 SDK Development

Creating iPhone and iPad Apps with Swift

Chris Adamson
Janie Clayton

edited by Rebecca Gulick

iOS 8 SDK Development

Creating iPhone and iPad Apps with Swift

Chris Adamson
Janie Clayton

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina

The team that produced this book includes:

Rebecca Gulick (editor)
Potomac Indexing, LLC (indexer)
Liz Welch (copyeditor)
Dave Thomas (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

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

Printed in the United States of America.
ISBN-13: 978-1-941222-64-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—March 2015

Do-It-Yourself Concurrency

Actually, our app isn't as fast as we might like. Try scrolling the table. The scrolling is still choppy. This has been the case since way back in *Custom Table Cells, on page ?*, where we started fetching the avatar images from their URLs. So let's think about what's causing the problem and whether we can fix it.

When the table asks us for a cell—in `tableView(cellForRowAtIndexPath:)`—we can easily set all the labels with strings from the `ParsedTweet`, but what we have for the avatar image is an `NSURL`. So we stop and load the data for that URL, make a new `UIImage` from it, and assign that to our custom cell's `UIImageView`. This has to happen for each cell. Moreover, we can only work on one cell at a time. As a new cell comes into view, we have to wait to download the image data, and only when we have it can we continue on to the next cell. It makes swiping quickly through the table impossible.

So, we're blocking the UIKit queue on a slow network access. "Hey, wait a minute," we say, "isn't that exactly what concurrency is supposed to fix? And isn't it exactly why the Social framework does the Twitter API call on a different queue?" Exactly. And that means to fix our problem, we should do what Apple does: *get our network stuff off the main queue.*

They Don't Call It "Blocking" the Main Queue for Nothing

Lest anyone think the issue of keeping long-running tasks off the main queue is an academic problem…well, do we have a story for you.

Years ago, one of the authors of this book was working at a company with a product that worked with video. For a demo, we had to show that the application could copy this video to an analog video tape recorder (VTR). Our solution was to connect the output of the video card to the VTR, and to use an RS-232 cable to send "record" and "stop" commands to the VTR. It seemed easy: to copy the video, we start the VTR recording and play the video from the PC, and then stop the VTR when the video's done. Easy peasy.

Except that the guy who wrote this didn't know how threads work in Java, which is what the application was written in. And desktop Java works almost exactly like UIKit: there's a main thread with an endless loop that looks for events like keypresses and mouse clicks, sends them to any code that handles the event, and repaints the window.

So when the user clicked the Record button, the code to play the video and start recording on the VTR was called…on the main thread. And that code effectively said "Wait here until the video is done," which meant that the window didn't update and no further events were processed until the video was done playing.

Some of these videos were 15 minutes long. The application couldn't do any repainting or event-handling during this time, so if you covered up the window and then foregrounded it, it wouldn't repaint. On Windows, dragging the mouse over the window would leave a trail of unerased mouse crud. Clicking a button did nothing. It was a disaster.

And this is pretty much where your author got to learn about threads, and had to completely rewrite this part of the program so that all of the video stuff happened on another thread, freeing up the main thread to immediately get back to work processing events and repainting, and then having the video thread put UI work back on the main thread only when ready.

And if you're still not convinced? Try plopping an NSThread.sleepUntilDate(NSDate(timeIntervalSinceNow:900.0)) as the first line of one of the button handlers. This will block the main queue for 900 seconds, or 15 minutes, during which time the button won't return to its untapped state, rotation events will be ignored, and the user will basically be blocked out of the app. *That's* what we're trying to avoid!

Moving Work Off the Main Queue

When tableView(cellForRowAtIndexPath:) needs an avatar, it does a slow NSURL load, makes an image from it, and sets it on the UIImageView. Only the last of these steps needs to be on the main queue, and the first shouldn't be. So we need a recipe to move work *off* the main queue.

dispatch_async() comes to our rescue again. Recall that it takes two parameters: the queue to put work on, and a closure with the tasks we want performed. What we need now is a different value for that first parameter, one that isn't the main queue, but just some other queue. For this, there's the GCD function dispatch_get_global_queue(), which takes a constant that indicates the priority of the system-provided queue we want. We're not picky, so we can use QOS_CLASS_DEFAULT to let GCD pick an ordinary background queue for us.

If you're an iOS 7 programmer, you might be wondering about that queue name. Well, iOS 8 introduces new "quality of service" constants for GCD queue priorities, which are meant to better express programmer intent. Unfortunately, they're not currently searchable in the Xcode documentation viewer, and are only visible in a C header file. The following table shows the new constants, and their older equivalents, which we'd have to use for code running on iOS 7 or earlier.

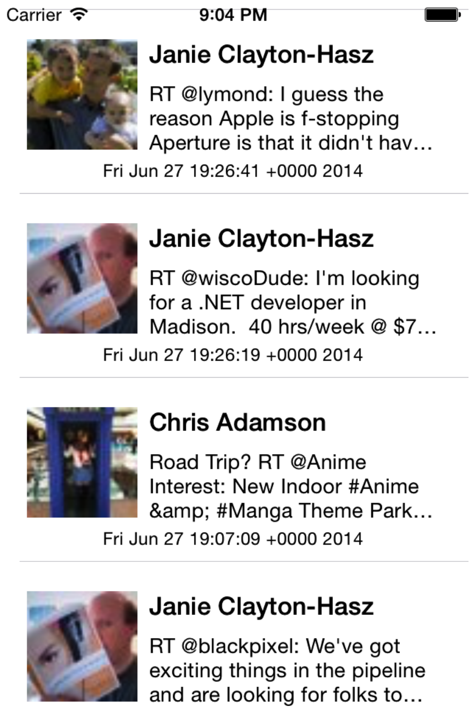
iOS 8 QOS Constant	iOS 7 Equivalent
QOS_CLASS_USER_INITIATED	DISPATCH_QUEUE_PRIORITY_HIGH
QOS_CLASS_DEFAULT	DISPATCH_QUEUE_PRIORITY_DEFAULT
QOS_CLASS_UTILITY	DISPATCH_QUEUE_PRIORITY_LOW
QOS_CLASS_BACKGROUND	DISPATCH_QUEUE_PRIORITY_BACKGROUND

Anyway, now we have the pieces we need. In tableView(cellForRowAtIndexPath:), find the if parsedTweet.userAvatarURL != nil block that sets the image, and replace it with the following version:

Concurrency/PragmaticTweets-7-2/PragmaticTweets/ViewController.swift

```
Line 1  dispatch_async(dispatch_get_global_queue(
     -    QOS_CLASS_DEFAULT, 0),
     -    {
     -      if let imageData = NSData (contentsOfURL:
     5        parsedTweet.userAvatarURL!) {
     -      let avatarImage = UIImage(data: imageData)
     -      dispatch_async(dispatch_get_main_queue(),
     -      {
     -        cell.avatarImageView.image = avatarImage
    10      })
     -    }
     -  })
```

Lines 1–12 are one big dispatch_async() call. The difference here is that we want to get work *off* the main queue, so on lines 1–2, we use the GCD function dispatch_get_global_queue() with the constant QOS_CLASS_DEFAULT to let GCD pick an ordinary background queue for us. That background queue gets the closure that runs from lines 3–12. This closure contains the "get a UIImage from an NSURL" logic from before, and then sets that image on the UIImageView. But since updating the image view has to happen on the main queue, we use a second dispatch_async() (lines 7–10) to wrap the UIKit work with a closure and put it back on the main queue.


Carrier 9:04 PM

Janie Clayton-Hasz
RT @lymond: I guess the reason Apple is f-stopping Aperture is that it didn't hav...
Fri Jun 27 19:26:41 +0000 2014

Janie Clayton-Hasz
RT @wiscoDude: I'm looking for a .NET developer in Madison. 40 hrs/week @ $7...
Fri Jun 27 19:26:19 +0000 2014

Chris Adamson
Road Trip? RT @Anime Interest: New Indoor #Anime &amp; #Manga Theme Park...
Fri Jun 27 19:07:09 +0000 2014

Janie Clayton-Hasz
RT @blackpixel: We've got exciting things in the pipeline and are looking for folks to...

And it's great! Now our table scrolls nice and fast, not blocking on the image loading at all!

There's just one more problem. Look at the figure. Every single one of the images is wrong: Chris is Janie, Janie is Chris, and *iOS Recipes* co-author Matt Drance is Janie, too.

Race Conditions

What's happened? A *race condition*, actually. When a cell goes offscreen and is queued for reuse, it will eventually get dequeued and filled with new data. *But the closure that fills in the image doesn't know that.* In this case, there was some cell for one of Matt's tweets that went off screen, dequeued, and repopulated with one of Janie's tweets (for the first row in the figure), but then the closure finished and filled in the image with Matt's picture. This doesn't happen often—we had to request 200 tweets, plus simulate poor network conditions to get the screenshot—but it *is* a bug, and if there's any way to make it happen in development, it's for sure going to hit someone in the real world.

The fix is to figure out when a closure has taken too long. How do we know that? Well, if the problem is that the cell has already filled in the contents from a different tweet, we can look to see if the parsedTweet that the closure started with has the same data that's displayed by the cell now. So here's the new contents for the if parsedTweet.userAvatarURL != nil block:

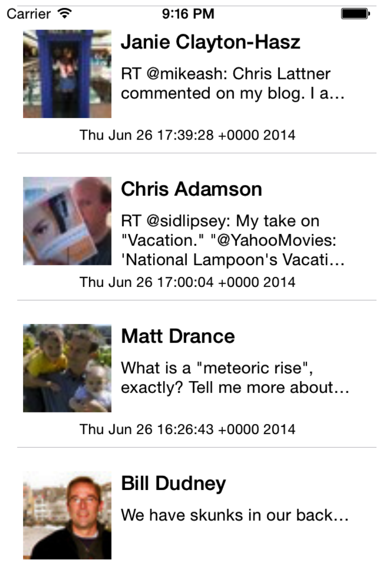
Concurrency/PragmaticTweets-7-2/PragmaticTweets/ViewController.swift

```
Line 1  cell.avatarImageView.image = nil
     -  dispatch_async(dispatch_get_global_queue(
     -    QOS_CLASS_DEFAULT, 0),
     -      {
     5        if let imageData = NSData (contentsOfURL:
     -          parsedTweet.userAvatarURL!) {
     -        let avatarImage = UIImage(data: imageData)
     -        dispatch_async(dispatch_get_main_queue(),
     -          {
    10            if cell.userNameLabel.text == parsedTweet.userName {
     -              cell.avatarImageView.image = avatarImage
     -            } else {
     -              println ("oops, wrong cell, never mind")
     -            }
    15          })
     -        }
     -    })
```

We start by clearing out the possibly wrong image, on line 1. The big change is inside the closure that runs on the main queue (lines 8–15): it looks to see if the text already set on the name label matches the userName of the ParsedTweet that the closure captured at the moment the closure was created. If it does, then this image belongs with this cell. If not, then the cell the closure was downloading an image for has already been reused and no longer matches, so the closure can just bail.

The else block is optional of course, but it's interesting to play with our network conditions and see how often the log message pops up in good conditions versus bad (for a way to reproduce this, see the sidebar on page 10). Suffice to say that if we hadn't fixed this, our Edge and 3G users would be *really unhappy*.

Now the race condition is fixed. If the image data comes in too late to use, we just don't use it. And we've once again been reminded of the promise and the hazards of working asynchronously. The figure shows our snappy and accurate app:



Janie Clayton-Hasz
RT @mikeash: Chris Lattner commented on my blog. I a...
Thu Jun 26 17:39:28 +0000 2014

Chris Adamson
RT @sidlipsey: My take on "Vacation." "@YahooMovies: 'National Lampoon's Vacati...
Thu Jun 26 17:00:04 +0000 2014

Matt Drance
What is a "meteoric rise", exactly? Tell me more about...
Thu Jun 26 16:26:43 +0000 2014

Bill Dudney
We have skunks in our back...

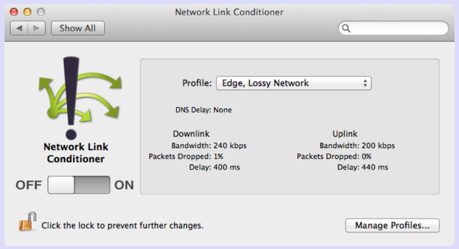
So we have a recipe for getting work onto and off of the main thread: just call dispatch_async(), with the work to be done as a closure. For the queue, we use dispatch_get_main_queue() to put work on the main queue, or dispatch_get_global_queue() to get a system queue that can get our work off the main queue. Either way, we're exploiting concurrency, the ability of the system to do many things at once, and now we're smarter about how to let the main queue keep doing its event-dispatching and repainting thing, while we do ours.

Joe asks: Can I Slow Down the Simulator Long Enough to See the Cells Get the Wrong Image?

If your Internet connection is really good, you may load the image data too fast to see the wrong-cells bug. This shows off one disadvantage of working with the simulator: its performance is unrealistically good, particularly for networking tasks. A Mac Pro with Gigabit Ethernet is going to get a web service response a lot more quickly than an iPhone with one bar of 3G coverage out in the woods somewhere.

Fortunately, a Mac can simulate lousy network conditions for this kind of testing. From the Xcode menu, select Open Developer Tool→More Developer Tools to be taken to Apple's Xcode downloads page. After asking for a developer ID and password, the page shows optional downloads for Xcode. Look for the latest version of the Hardware IO Tools For Xcode, download it, and double-click the Network Link Conditioner.prefPane to install it.

This adds a pane to the Mac's System Preferences called Network Link Conditioner, which adjusts the performance of the Mac's current networking device (Ethernet, AirPort, etc.) to resemble real-world conditions an iOS device might face, from Wi-Fi with good connectivity to the outdated Edge network experiencing packet loss.



Profile: Edge, Lossy Network

DNS Delay: None

```
Downlink                        Uplink
   Bandwidth: 240 kbps             Bandwidth: 200 kbps
Packets Dropped: 1%             Packets Dropped: 0%
        Delay: 400 ms                   Delay: 440 ms
```

Click the lock to prevent further changes.

Keep in mind, however, that the Network Link Conditioner degrades *all* network traffic on the Mac, not just the iOS Simulator application. So if we forget to turn it off when we're done testing, it will make everything we do seem like we're getting one bar in the middle of nowhere.