Stanford CS193p

Developing Applications for iOS

Spring 2023

Lecture 14

Color vs. UIColor

- Color

  What this symbol means in SwiftUI varies by context.
  Is a color-specifier, e.g., `.foregroundColor(Color.green)`.
  Can also act like a `ShapeStyle`, e.g., `.fill(Color.blue)`.
  Can also act like a `View`, e.g., `Color.white` can appear wherever a `View` can appear.
  Due to this multifaceted role, its API is limited mostly to creation/comparison.

- UIColor

  Is used to manipulate colors.
  Also has many more built-in colors than `Color`, including "system-related" colors.
  Can be interrogated and can convert between color spaces.
  For example, you can get the RGBA values from a `UIColor`.
  Once you have desired `UIColor`, employ `Color(uiColor:)` to use it in one of the roles above.

Image vs. UIImage

- Image
  Primarily serves as a View.
  Is not a type for vars that hold an image (i.e. a jpeg or gif or some such). That's UIImage.
  Access images in your Assets.xcassets (in Xcode) by name using Image(_ name: String).
  Also, many, many system images available via Image(systemName:).
  Search all these system images in the SF Symbols app (available at developer.apple.com/design).
  While you're there, study the Human Interface Guidelines. A must for AppStore submission.
  You can control the size of system images with .imageScale() View modifier.
  System images are also very useful as masks (for gradients, for example).
- UIImage
  Is the type for actually creating/manipulating images and storing in vars.
  Very powerful representation of an image.
  Multiple file formats, transformation primitives, animated images, etc.
  Once you have the UIImage you want, use Image(uiImage:) to display it.

Multithreading

- Don't Block my UI!
  - It is never okay for a mobile application UI to be unresponsive to the user.
  - But sometimes you need to do things that might take more than a few milliseconds.
  - Most notably … accessing the network.
  - Another example … doing some very CPU-intensive analysis, e.g. machine learning.
  - How do we keep our UI responsive when these long-lived tasks are going on?
  - We execute them on a different "thread of execution" than the UI is executing on.

Multithreading

- Threads

  Most modern operating systems (including iOS) let you specify a "thread of execution" to use.
  These threads appear to be all executing their code simultaneously.
  And they might, in fact, be (in a multi-core (like iOS devices) or multi-processor computer).
  But they might just be switching back and forth between them really quickly.
  With some of them getting higher priority to run than others.
  You as a programmer can't tell the difference (and you don't want to know).

- Queues

  The challenge is to make multithreaded code authorable, readable and understandable.
  This is because time adds a "fourth dimension" to our code.
  Swift manages this complexity this using queues.
  A queue is just a bunch of blocks of code, lined up, waiting for a thread to execute them.
  You don't worry about the threads in Swift, you are concerned only with queues.
  The system takes care of providing/allocating threads to execute code off these queues.

Multithreading

- Queues and Closures

  We specify the blocks of code waiting in a queue using closures (aka functions as arguments).
  The core multithreading API is very simple: plop a closure onto a queue.
  We'll see this soon, but first, let's talk about queues ...

- Main Queue

  The most important queue in all the world (of iOS) is called the main queue.
  It is the queue that has all the blocks of code that might muck with the UI.
  Any time we want to do something in the UI, we must use this queue.
  It is an unequivocal error to do UI (directly or indirectly) off the main queue.

  The system uses a single thread to process all the blocks of code from the main queue.
  So it can also be used to synchronize.
  (e.g. to protect data structures by only mutating them on the main queue)

Multithreading

- Background Queues

  There are also a bunch of available “background queues.”
  These are where we do any long-lived, non-UI tasks.

  The system has a bunch of threads available to execute code on these background queues.
  So often they’ll be running “in parallel” (aka simultaneously).
  And they’ll also be running in parallel with the main UI queue.
  You can influence the priority of these background queues.
  You do this by specifying a “quality of service” you desire for that queue.

  But the main queue will always be higher priority than any of the background queues.

Multithreading

- GCD

  The base API for doing all this queue stuff is called GCD (Grand Central Dispatch).

  It has a number of different functions in it, but there are two fundamental tasks …

  1. getting access to a queue
  2. plopping a block of code on a queue

Multithreading

Creating a Queue

There are numerous ways to create a queue, but we're only going to look at two ...

```
DispatchQueue.main                    // the queue where all UI code must be posted

DispatchQueue.global(qos: QoS) // a non-UI queue with a certain quality of service
qos (quality of service) is one of the following ...
   .userInteractive   // do this fast, the UI depends on it!
   .userInitiated     // the user just asked to do this, so do it now
   .utility           // this needs to happen, but the user didn't just ask for it
   .background        // maintenance tasks (cleanups, etc.)
```

Multithreading

- Plopping a Closure onto a Queue

  There are two basic ways to add a closure to a queue …

  `let queue = DispatchQueue.main` or `DispatchQueue.global(qos:)`

  `queue.async { /* code to execute on queue */ }`

  `queue.sync { /* code to execute on queue */ }`

  The second one blocks waiting for that closure to be picked off by its queue and completed.
  Thus you would never call `.sync` in UI code.
  Because it would block the UI.
  You might call it on a background queue (to wait for some UI to finish, for example).
  But even that is rare.

  So we almost always use `.async`.
  Always remember that `.async` will execute that closure "sometime later".
  (Whenever that closure gets to the front of the queue you plopped it onto.)
  Your code needs to be tolerant of that.

  There are also functions to have the queue wait for a delay interval before executing.

Multithreading

Nesting

The beauty of this API is when you end up nesting.

For example …

```
DispatchQueue(global: .userInitiated).async {
    // do something that might take a long time
    // this will not block the UI because it is happening off the main queue
    // once this long-time thing is done, it might require a change to the UI
    // but we can't do UI here because this code is executing off the main queue
    // no problem, we just plop a closure with the UI code we want onto the main queue
     DispatchQueue.main.async {
         // UI code can go here! we're on the main queue!
     }
}
```

This almost makes asynchronous code look synchronous. But it's still not.

So be careful to think through what happens if long-time things take a really long time.

The world might have changed quite a bit during that time.

Multithreading

Asynchronous API

You will do `DispatchQueue.main.async { }` often when programming asynchronously in iOS.

However, you won't do `DispatchQueue.global(qos:)` as much as you might think.
That's because a lot of asynchronous iOS API is at a higher level.
Numerous iOS functions automatically do their work on one of these `global` queues.
Example: `URLSession` (fetches data from a URL and calls you back when it's got it).

But beware!
Often you'll give something like `URLSession` a "call me when it's done" closure as an argument.
And it may well execute your closure on these `global` queues as well!
In which case, if you want to do UI in response, you'll need `DispatchQueue.main.async { }`.

We're going to fetch some data from the internet directly instead of using `URLSession`.
But that's just so you can see this "nesting" of GCD going on.
In the real world, you'd use `URLSession` to fetch data from the internet.