Stanford CS193p

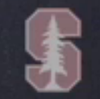
Developing Applications for iOS

Spring 2023

Lecture 14

Today

- Colors and Images
  - `Color` vs. `UIColor`
  - `Image` vs. `UIImage`
- Multithreaded Programming and Error Handling
  - Ensuring that my app is never "frozen", even for a moment
  - Cleaning up our APIs by throwing errors instead of returning them from functions
  - Demo: Implementing our own version of `AsyncImage`



Color vs. UIColor vs. CGColor

- 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 (RGB vs. HSB, etc.).
  - Once you have desired `UIColor`, employ `Color(uiColor:)` to use it in one of the roles above.
- CGColor
  - The fundamental color representation in the Core Graphics drawing system.
  - `Color` might be able to give a `CGColor` representation of itself (`color.cgColor` is optional).

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)`.
  - And of course you can create SF Symbol images using `Image(systemName:)`.
  - You can control the size of system images with `.imageScale()` `View` modifier.
  - System images also are affected by the `.font` 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.

Next Lecture

- Colors and Images
  - `Color` vs. `UIColor`
  - `Image` vs. `UIImage`
- Multithreaded Programming and Error Handling
  - Ensuring that my app is never "frozen", even for a moment
  - More about `throw`ing and `catch`ing errors
  - Demo: Implementing our own version of `AsyncImage`

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.
  Keeping track of what's running when and where can be tricky and arduous.

  In Swift, the system manages threads of execution behind the scenes for us!
  This makes it all a lot less tricky and arduous for us as Swift programmers.
  But we will have to give the system some hints about how it should do this ...

Multithreading

- Task

  In Swift, you can fire off some work asynchronously (i.e. "in parallel") using a `Task`.

```
let task = Task(priority: TaskPriority? = nil) {
    // do something that might take a long time
}
```

  The line of code above returns immediately.

  It will execute the closure "sometime later" (as soon as immediately, but never never).

  The returned `task` lets you `cancel`, `yield`, `sleep` or wait for the result `value` of the `Task`.

  Very often we just fire off a `Task` and never talk to the `Task` itself again.

```
Task {
    // do something that might take a long time
}
```

  With `Task`s you are telling the system "this might take a long time."

  So the system knows to work on it in parallel with other stuff (aka "asynchronously").

Multithreading

- The Pitfall of Multithreading

  This all sounds really great (and it is), but there is a big thing to worry about here ...
  Access to data structures.
  Imagine all these threads are all off simultaneous modifying the same `Array` in your ViewModel.
  They could all get very confused.
  Somehow access to data structures has to be synchronized.

  Swift does this with yet another data type: `actor`.
  An `actor` is a reference type (like `class` is), but does not have inheritance.
  So, other than its reference type semantics, it looks a lot like a `struct`.
  But an `actor`'s primary "talent" is that it synchronizes all access to its vars and funcs.

  If you are not doing multithreading, you'll never have to create an `actor`.
  If you are doing multithreading and UI, then there is a "main `actor`" to consider.
  We'll talk about that in a bit.

Multithreading

- actor

  How does an actor synchronize all data access across multiple threads of execution?
  First, only one of the actor's funcs (or computed vars) can be running at any given time.
  Second, all its funcs either run to completion or get suspended (and resumed at a later time).
  Since two funcs are never running at the same time, there're no conflicts over the actor's data.
  Voila! Synchronization!

  For this all to work, you have to let the system know about when suspensions can happen.
  So one thing you must do is mark any function that can be suspended with the keyword async.
  Suspendable functions are thus called "async functions".

  Functions that do things like access the network are almost always async.
  Because they are suspending while they wait for the network data to return.

  async is an advertisement that a suspension "could" happen in the func, not that it will.
  A very time-consuming function might mark itself async even if it can't really suspend.
  This would force callers to react to this ...

Multithreading

- async and await

  Okay, so we have some functions that are suspendable, aka async functions.

  There's another important rule ...
  When you call an async function, you must mark the call with the keyword await.
  And here is the rub: when a func executes a line of code with await, it gets suspended!
  In fact, the only time you can get suspended is if you await a call to an async function.
  And yes, that means any func that has an await in it must itself be marked async.
  (Since the very definition of an async function is that it can be suspended.)

  The whole reason await exists is for you to remind yourself you can be suspended there.
  Because you don't know how long you might be suspended at that await.
  It could be a very long time (or it could be zero time).
  And when you resume, other funcs in your actor might have been called in the meantime.
  So in the Array example, you might have to re-evaluate your assumptions about the Array.
  But at least there's no way the Array can be modified out from under you unless you await.
  (Because otherwise you are guaranteed to run to completion before anyone can interfere.)

Multithreading

- `await` inside a `Task`
  - There is a way, however, to call an `async` function without suspending yourself.
  - You do it inside a `Task`.
  - Putting any code in a `Task` of course means it will run "sometime later".
  - So your code cannot depend on what's being done in that `Task` in the rest of your `func`.
  - The code inside the `Task` might not execute until after your `func` is completely finished.

Multithreading

- async closures

  We haven't really talked about closures here.

  What happens if you pass a closure as an argument and the closure uses `await`?

  In that case the function you're passing the closure to must mark its argument with `async`.

  A couple of `View` modifiers take `async` closures as arguments ...

  `.task(async () -> Void)` // fires up a `Task` whose lifetime matches the `View`'s

  `.refreshable(async () -> Void)` // allows the user to "pull down to refresh" asynchronously

Multithreading

- The Main Actor

  The most important "data structure" we want to protect from multithreading is the UI.

  So there is an `actor` just for the UI.
  It's called the "main `actor`".
  All UI activity must be done on the main `actor`.

  All of your `View` functions and computed vars automatically run on the main `actor`.

  Your ViewModel, however, does not.
  Fear not!
  You can mark specific ViewModel `func`s with `@MainActor` to get them to run on the main `actor`.
  Or you can mark your entire ViewModel as `@MainActor` and all its `func`s will run there.
  If you do multithreading and don't do this, you'll get "purple" warnings in Xcode when you run.
  You must fix these or your UI will probably be unpredictably broken.

Error Handling

- Error Handling

  Error handling and async go together well.

  Why?

  Because when you are awaiting an async function to return its value, a problem might occur.
  For example, the async function fetches an image for you from the network.
  But there's a network error!
  In those scenarios, we want the suspended function to be able to resume asap.
  So we'll bail out of our function by throwing an error.

Error Handling

- Error Handling

  We know that functions that can suspend have to be marked `async`.
  And we know that functions that can `throw` errors have to be marked `throws`.
  We also know that when we call an `async` function we have to use the keyword `await`.
  And we call a function that `throws` errors, we have to use the keyword `try`.

  So the two mechanisms have a similar syntax and usage pattern.

Error Handling

- Throwing errors

  Let's look at what a function that `throws` might look like ...

```
func attendLecture() throws {
    if sleptIn {
        throw CS193pError.missedLecture
    }
    askQuestions() // not done if you sleptIn because throw exits attendLecture()
    . . .
}
enum CS193pError: Error {     // thrown things conform to the Error protocol
    case lateHomework(daysLate: Int)
    case missedLecture
    var localizedDescription: String { switch self { . . . } }
}
```

Error Handling

- Catching Specific Errors

  We know that we use `do { } catch { }` to catch errors.
  Sometimes we want to catch specific errors and handle them in specific ways.
  You do this by adding `catch` phrases to your `do { } catch { }` ...

```
do {
    try somethingThatThrows()
    print("hello") // will only happen if somethingThatThrows() does not throw anything
} catch CS193pError.lateHomework(let daysLate) {
    // notice how daysLate is grabbing some error-specific enum associated data
} catch let cs193pError where cs193pError is CS193pError {
    // handle all CS193pErrors other than .lateHomework (which is handled above)
} catch {
    // handle all other (non-CS193pErrors, since those are all handled above)
}
print("keep going") // will execute next after any caught error is handled
```