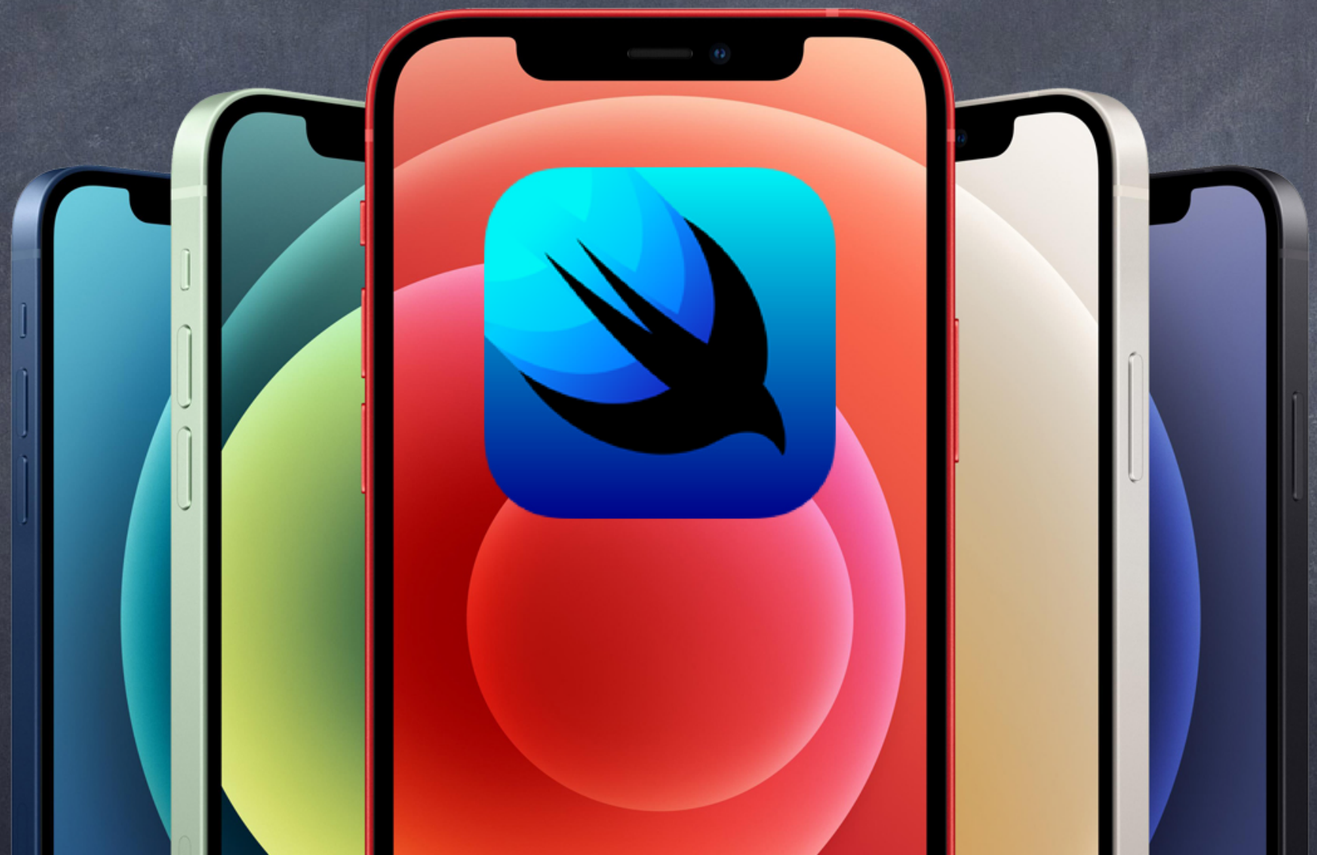
Stanford CS193p

Developing Applications for iOS

Spring 2023

Lecture 7



Today

- Demo Interlude
  - Separating a View into its own Swift file (demo: CardView).
  - Dealing with constants in your Swift code.
- Shape
  - Drawing your own custom shape.
  - Demo: Drawing (but not yet animating) a "pie-shaped" countdown timer on our cards.
- Animation
  - How does it work?
- ViewModifier
  - What exactly are functions like `foregroundColor`, `font`, `padding`, etc. doing?

Shape

- Shape
  - Shape is a protocol that inherits from View.
  - In other words, all Shapes are also Views.
  - Examples of Shapes already in SwiftUI: RoundedRectangle, Circle, Capsule, etc.

Shape

- Shape

  By default, Shapes draw themselves by filling with the current foreground color.
  But we've already seen that this can be changed with `.stroke()` and `.fill()`.
  This modifiers return a `View` that draws the `Shape` in the specified way (by stroking or filling).

  The arguments to `stroke` and `fill` are pretty interesting.
  In our demo, it looked like the argument to `fill` was a `Color` (e.g. `Color.white`).
  But that's not quite the case ...

  ```
  func fill<S>(_ whatToFillWith: S) -> View where S: ShapeStyle
  ```

  This is a generic function (similar to, but different than, a generic type).
  `S` is a don't care (but since there's a `where`, it becomes a "care a little bit").
  `S` can be anything that implements the `ShapeStyle` protocol.
  The `ShapeStyle` protocol turns a `Shape` into a `View` by applying some styling to it.
  Examples of such things: `Color`, `ImagePaint`, `AngularGradient`, `LinearGradient`.

Shape

- Shape

  But what if you want to create your own Shape?

  The Shape protocol (by extension) implements View's body var for you.
  But it introduces its own func that you are required to implement …

```
func path(in rect: CGRect) -> Path {
    return a Path
}
```

  In here you will create and return a Path that draws anything you want.
  Path has a ton of functions to support drawing (check out its documentation).
  It can add lines, arcs, bezier curves, etc., together to make a shape.

  This is best shown via demo.
  So we're going to add a "timer countdown pie" to our CardView (unanimated for now) …

Animation

- Animation

  Animation is very important in a mobile UI.
  That's why SwiftUI makes it so easy to do.

  One way to do animation is by animating a `Shape`.
  We'll show you this a bit later when we get our `Pie` moving.

  The other way to do animation is to animate `View`s via their `ViewModifier`s.
  So what's a `ViewModifier`?

ViewModifier

- ViewModifier

  Probably best learned by example.

  Let's say we wanted to create a modifier that would "card-ify" another `View`.
  In other words, it would take that `View` and put it on a card like in our Memorize game.
  It would work with any `View` whatsoever (not just our `Text("👻")`).
  What would such a modifier look like?

ViewModifier

- Cardify ViewModifier

```
Text("👻").modifier(Cardify(isFaceUp: true)) // eventually .cardify(isFaceUp: true)

struct Cardify: ViewModifier {
    var isFaceUp: Bool
    func body(content: Content) -> some View {
        ZStack {
            if isFaceUp {
                RoundedRectangle(cornerRadius: 10).fill(Color.white)
                RoundedRectangle(cornerRadius: 10).stroke()
                content
            } else {
                RoundedRectangle(cornerRadius: 10)
            }
        }
    }
}
```

ViewModifier

- Cardify ViewModifier

```
Text("👻").modifier(Cardify(isFaceUp: true)) // eventually .cardify(isFaceUp: true)

struct Cardify: ViewModifier {
    var isFaceUp: Bool
    func body(content: Content) -> some View {
        ZStack {
            if isFaceUp {
                RoundedRectangle(cornerRadius: 10).fill(Color.white)
                RoundedRectangle(cornerRadius: 10).stroke()
                content
            } else {
                RoundedRectangle(cornerRadius: 10)
            }
        }
    }
}
```

.modifier() returns a View that displays this

ViewModifier

- ViewModifier

  How do we get from …

  ```
  Text("👻").modifier(Cardify(isFaceUp: true))
  ```

  … to …

  ```
  Text("👻").cardify(isFaceUp: true)
  ```

  ?

  Easy …

  ```
  extension View {
      func cardify(isFaceUp: Bool) -> some View {
          return self.modifier(Cardify(isFaceUp: isFaceUp))
      }
  }
  ```

protocol

- What is a protocol used for?

  One of the most powerful uses of protocols is to facilitate code sharing.
  Implementation can be added to a protocol by creating an extension to it.

  This is how Views get foregroundColor and font and all their other modifiers!

  This is also how functions like filter and firstIndex(where:) get implemented.

  An extension can also add a default implementation for a func or var in the protocol.
  (That's how ObservableObjects get objectWillChange for free.)

  Adding extensions to protocols is key to protocol-oriented programming in Swift.

ViewModifier

- ViewModifier

  You know all those little functions that modified our Views (like aspectRatio and padding)?
  They are (likely) turning right around and calling a function in View called `modifier`.

  e.g. `.aspectRatio(2/3)` is likely something like `.modifier(AspectModifier(2/3))`
  `AspectModifier` can be anything that conforms to the `ViewModifier` protocol …

ViewModifier

- ViewModifier

  The ViewModifier protocol has one function in it.

  This function's only job is to create a new View based on the thing passed to it.

  Conceptually, this protocol is sort of like this …

```
protocol ViewModifier {
    func body(content: Content) -> some View { // Content is a "care a little bit" View
        return some View that almost certainly contains content
    }
}
```

  When we call .modifier on a View, the content passed to this body function is that View.

```
aView.modifier(MyViewModifier(arguments: …))
```

  MyViewModifier implements ViewModifier and aView will be passed to its body via content.

protocol

- What is a `protocol` used for?

  Adding filter to `Array`, `Range`, `String`, `Dictionary`, et. al. ...

  ```
  filter(_ isIncluded: (Element) -> Bool) -> Array<Element>
  ```

  This function was written once by Apple.

  And yet it works on `Array` and `Range` and `String` and `Dictionary` and others!

  `filter` was added to the Foundation library as an `extension` to the `Sequence` protocol.

protocol

- View

Somewhere in SwiftUI there's something (sort of) like this going on …

```
protocol View {
    var body: some View
}
```

And there's also something like this in SwiftUI …

```
extension View {
    func foregroundColor(_ color: Color) -> some View { /* implementation */ }
    func font(_ font: Font?) -> some View { /* implementation */ }
    func blur(radius: CGFloat, opaque: Bool) -> some View { /* implementation */ }
    … and many more …
}
```

The first part constrains something like CardView to have to provide that body var.

The second part gives many gains to CardView as a reward for doing so!

Obviously the above is a great oversimplification, but conceptually that's what's happening.

protocol

- Generics + Protocols

  Consider the protocol we already know called Identifiable …

```
protocol Identifiable {
    var id: ID { get }
}
```

  The type ID here is a "don't care" for Identifiable.

  Yes, protocols can be generic (i.e. have "don't cares") too!

  i.e. you can use any type of thing as the identifier to make something Identifiable.

  protocols sometimes declare their "don't care" types in a different way than a struct does …

```
protocol Identifiable {
    associatedtype ID
    var id: ID { get }
}
```

protocol

- Generics + Protocols

```
protocol Identifiable {
    associatedtype ID
    var id: ID { get }
}
```

As we learned in demo earlier, this type `ID` also has to be `Hashable`.
(Luckily `String`, the type we used for `ID` in `MemoryGame.Card` is `Hashable`.)
That's because we want to be able to look up these `Identifiable` things in hash tables.
It'd be kind of useless for something to be `Identifiable` but not "lookupable".

protocol

- Generics + Protocols

  So how do we enforce that (i.e. that `ID` is some type of thing that can be hashed)?

  By simply constraining the `ID` "don't care" to also "behave like" a `Hashable`.

  `Hashable` is just yet another `protocol` we'll look at in a minute.

```
protocol Identifiable {
    associatedtype ID where ID: Hashable
    var id: ID { get }
}
```

  Or, more simply ...

```
protocol Identifiable {
    associatedtype ID: Hashable
    var id: ID { get }
}
```

  We've just turned `ID` from a "don't care" into a "we care a little bit".

protocol

- some

  This keyword can be used with a protocol to pass things opaquely in or out of a function/var.
  Opaquely means that you know the thing conforms to the protocol, but nothing more about it.
  By in or out we mean a protocol can be used with some as a parameter type or return value.