Stanford CS193p

Developing Applications for iOS

Spring 2020

Lecture 5

Today

- Swift Demo Warmup!
  - Access Control
- @ViewBuilder
  - What exactly is that argument to ZStack, ForEach, GeometryReader, etc.?
- Shape
  - What if I want to draw my own View rather than construct it from other Views?
- Animation
  - Mobile app UIs look pretty bad without animation.
  - Luckily, in SwiftUI, animation (almost) comes for free!
- ViewModifier
  - What exactly are functions like foregroundColor, font, padding, etc. doing?

@ViewBuilder

- @ViewBuilder

  Based on a general technology added to Swift to support "list-oriented syntax".
  It's a simple mechanism for supporting a more convenient syntax for lists of Views.

  Developers can apply it to any of their functions that return something that conforms to View.
  If applied, the function will still return something that conforms to View
  But it will do so by interpreting the contents as a list of Views and combine them into one.

  That one View that it combines it into might be a TupleView (for two to ten Views).
  Or it could be a _ConditionalContent View (when there's an if-else in there).
  Or it could even be EmptyView (if there's nothing at all in there; weird, but allowed).
  And it can be any combination of the above (if's inside other if's, etc.).

  Note that some of this is not yet fully public API (like _ConditionalContent).
  But we don't actually care what View it creates for us when it combines the Views in the list.
  It's always just some View as far as we're concerned.

@ViewBuilder

- @ViewBuilder

  Any `func` or read-only computed `var` can be marked with `@ViewBuilder`.
  If so marked, the contents of that `func` or `var` will be interpreted as a list of `Views`.

  For example, if we wanted to factor out the `Views` we use to make the front of a `Card` …

```
@ViewBuilder
func front(of card: Card) -> some View {
    RoundedRectangle(cornerRadius: 10)
    RoundedRectangle(cornerRadius: 10).stroke()
    Text(card.content)
}
```

  And it would be legal to put simple `if-else`'s to control which `Views` are included in the list.
  (But this is just the front of our card, so we don't need any `if`s.)

  The above would return a `TupleView<RoundedRectangle,RoundedRectangle,Text>`.

@ViewBuilder

- @ViewBuilder

  You can also use @ViewBuilder to mark a parameter that returns a View.

  For example, we know GeometryReader allows you to use @ViewBuilder syntax.
  Here's approximately how GeometryReader is declared inside SwiftUI …

```
struct GeometryReader<Content> where Content: View {
    init(@ViewBuilder content: @escaping (GeometryProxy) -> Content) { . . . }
}
```

  The content parameter is just a function that returns a View.
  Now all users of GeometryReader get to use the list syntax to express the Views to be sized.

  ZStack, HStack, VStack, ForEach, Group all do this same thing.
  We could have done this in our Grid except that we don't really know how to extract the Views.
  That is something "private" that SwiftUI View combiners know how to do.
  Probably will be made public down the road.

@ViewBuilder

- @ViewBuilder
  - Just to reiterate …
  - The contents of a `@ViewBuilder` is a list of `Views`.
  - Only.
  - It's not arbitrary code.
  - The `if-else` statements in there are just used to choose `Views` to include in the list.
  - So you can't declare variables.
  - Or just do random code.
  - It can only be a (conditional) list of `Views`.
  - Nothing more can be in there!

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()`.
  They 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.
  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 that "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 `Views` via their `ViewModifiers`.

  So what's a `ViewModifier`?

ViewModifier

- ViewModifier

  You know all those little functions that modified our Views (like aspectRatio and padding)?
  They are (probably) 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 …

  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.

```
protocol ViewModifier {
    associatedtype Content // this is a protocol's version of a "don't care"
    func body(content: Content) -> some View {
        return some View that represents a modification of content
    }
}
```

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

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)
            }
        }
    }
}
```

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))
      }
  }
  ```

Demo

- Cardify
  - Let's go implement this in our Memorize application!
  - That demo will be all we have time for today.
  - We'll start the next lecture talking about how we animate using `ViewModifiers`.