Stanford CS193p

Developing Applications for iOS

Spring 2023

Lecture 7

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))
      }
  }
  ```

protocol

- A `protocol` is sort of a "stripped-down" `struct`/`class`
  - It has functions and vars, but no implementation (or storage)!
  - Declaring a `protocol` looks very similar to `struct` or `class` (just w/o implementation) ...

```
protocol Moveable {
    func move(by: Int)
    var hasMoved: Bool { get }
    var distanceFromStart: Int { get set }
}
```

  - See? No implementation.
  - The `{ }` on the `var`s just say whether it's read only or a `var` whose value can also be set.

protocol

- A `protocol` is sort of a "stripped-down" `struct`/`class`

  It has `functions` and `vars`, but no implementation (or storage)!

  Declaring a `protocol` looks very similar to `struct` or `class` (just w/o implementation) …

```
protocol Moveable {
    func move(by: Int)
    var hasMoved: Bool { get }
    var distanceFromStart: Int { get set }
}
```

  … now any other type can claim to implement `Moveable` …

```
struct PortableThing: Moveable {
    // must implement move(by:), hasMoved and distanceFromStart here
}
```

protocol

- A `protocol` is sort of a "stripped-down" `struct`/`class`

  It has `functions` and `vars`, but no implementation (or storage)!

  Declaring a `protocol` looks very similar to `struct` or `class` (just w/o implementation) ...

```
protocol Moveable {
    func move(by: Int)
    var hasMoved: Bool { get }
    var distanceFromStart: Int { get set }
}
```

  ... and this is also legal (this is called "protocol inheritance") ...

```
protocol Vehicle: Moveable {
    var passengerCount: Int { get set }
}
class Car: Vehicle {
    // must implement move(by:), hasMoved, distanceFromStart and passengerCount here
}
```

protocol

- A `protocol` is sort of a "stripped-down" `struct`/`class`

  It has functions and vars, but no implementation (or storage)!

  Declaring a `protocol` looks very similar to `struct` or `class` (just w/o implementation) ...

```
protocol Moveable {
    func move(by: Int)
    var hasMoved: Bool { get }
    var distanceFromStart: Int { get set }
}
```

  ... and you can claim to implement multiple protocols ...

```
class Car: Vehicle, Impoundable, Leasable {
    // must implement move(by:), hasMoved, distanceFromStart and passengerCount here
    // and must implement any funcs/vars in Impoundable and Leasable too
}
```

protocol

- A protocol is a type

  Most protocols can be used anywhere any other type can be used.

  For example, it can be the type of an argument to a function or of any variable …

```
var m: Moveable
var car: Car = new Car(type: “Tesla”)
var portable: PortableThing = PortableThing()
m = car // perfectly legal
m = portable // perfectly legal
```

  … even though Car is a completely different class or struct than PortableThing.

  Both implement the Moveable protocol and so can be assigned to a var of type Moveable.

  Note, though, that this will not work …

```
portable = car // NOT legal
```

  The var portable is of type PortableThing (not Moveable) and a Car is not a PortableThing.

  Swift enforces the type of the var at all times.

protocol extension

- Adding protocol implementation

  One way to think about protocols is constrains and gains …

```
struct Tesla: Vehicle {
    // Tesla is constrained to have to implement everything in Vehicle
    // but gains all the capabilities a Vehicle has too
}
```

  But how does a Vehicle "gain capabilities" if it has no implementation?

  You can add implementations to a protocol using an extension to the protocol …

```
extension Vehicle {
    func registerWithDMV() { /* implementation here */ }
}
```

  Now Teslas (and all other Vehicles) can be registered with the DMV.

  Adding extensions to protocols is at the heart of functional programming in Swift.

  The protocol View is the world's greatest example of this!

protocol extension

- Adding `protocol` implementation

  You can even add "default implementations" of the `protocol`'s own `func`s/`var`s …

```
protocol Moveable {
    func move(by: Int)
    var hasMoved: Bool { get }
    var distanceFromStart: Int { get set }
}
extension Moveable {
    var hasMoved: Bool { return distanceFromStart > 0 }
}
struct ChessPiece: Moveable {
    // only need to implement move(by:) and distanceFromStart here
    // don't have to implement hasMoved because there's a default implementation out there
    // would be allowed to implement hasMoved here if we wanted to, though
}
```

extension

- Adding code to a `struct` or `class` via an `extension`

  Of course, you can use an `extension` to add things to `structs` and `classes` too.

```
struct Boat {
    ...
}
extension Boat {
    func sailAroundTheWorld() { /* implementation */ }
}
```

  You can even make something conform to a `protocol` purely via an `extension` ...

```
extension Boat: Moveable {
    // implement move(by:) and distanceFromStart here
}
```

  Now `Boat` conforms to the `Moveable` protocol!

protocol

- Why protocols?
  - Why do we do all this protocol stuff?
  - It is a way for types (structs/classes/other protocols) to say what they are capable of.
  - And also for other code to demand certain behavior out of another type.
  - But neither side has to reveal what sort of struct or class they are.
  - This is what "functional programming" is all about.
  - It's about formalizing how data structures in our application function.
  - Even when we talk about vars in the context of protocols, we don't define how they're stored.
  - We focus on the functionality and hide the implementation details behind it.
  - It's the promise of encapsulation from OOP but taken to a higher level.
  - And this is even more powerful when we combine it with generics ...

Generics and Protocols

- Help!

  Some of you might be shivering a bit right now.
  You might be thinking, "how am I going to design systems using generics/`protocols`?!"
  This is, indeed, a powerful foundation for designing things.
  But functional programming does require some mastery that only comes with experience.
  The good news: you can do a lot in SwiftUI without having to master functional programming.
  But the more you use it, the more you'll want to be "grokking" it.
  That's why I explain to you up front what's going on.
  No one expects you to now be able to be adding `extensions` to `protocols` with generics!
  But eventually you will be able to.
  And in the meantime, you'll have at least some idea how SwiftUI is built.