Stanford CS193p

Developing Applications for iOS

Spring 2023

Lecture 3

Today

- MVVM
  - Design paradigm
- Swift Type System
  - `struct`
  - `class`
  - `protocol`
  - "don't care" type (aka generics)
  - `enum`
  - functions
- Back to the demo!
  - Apply MVVM to Memorize

Model and UI

- Separating "Logic and Data" from "UI"
  - SwiftUI is very serious about the separation of application logic & data from the UI
  - We call this logic and data our "Model"
  - It could be a `struct` or an SQL database or some machine learning code or many other things
    Or any combination of such things
  - The UI is basically just a "parametrizable" shell that the Model feeds and brings to life
    Think of the UI as a visual manifestation of the Model
  - The Model is where things like `isFaceUp` and `cardCount` would live (not in `@State` in the UI)
  - SwiftUI takes care of making sure the UI gets rebuilt when a Model change affects the UI

Model and UI

- Connecting the Model to the UI

  There are some choices about how to connect the Model to the UI …

  1. Rarely, the Model could just be an `@State` in a `View` (this is very minimal to no separation)
  2. The Model might only be accessible via a gatekeeper "View Model" `class` (full separation)
  3. There is a View Model `class`, but the Model is still directly accessible (partial separation)

  Mostly this choice depends on the complexity of the Model …

  A Model that is made up of SQL + `struct`(s) + something else will likely opt for #2

  A Model that is just a simple piece of data and little to no logic would likely opt for #1

  Something in-between might be interested in option #3

  We're going to talk now about #2 (full separation).

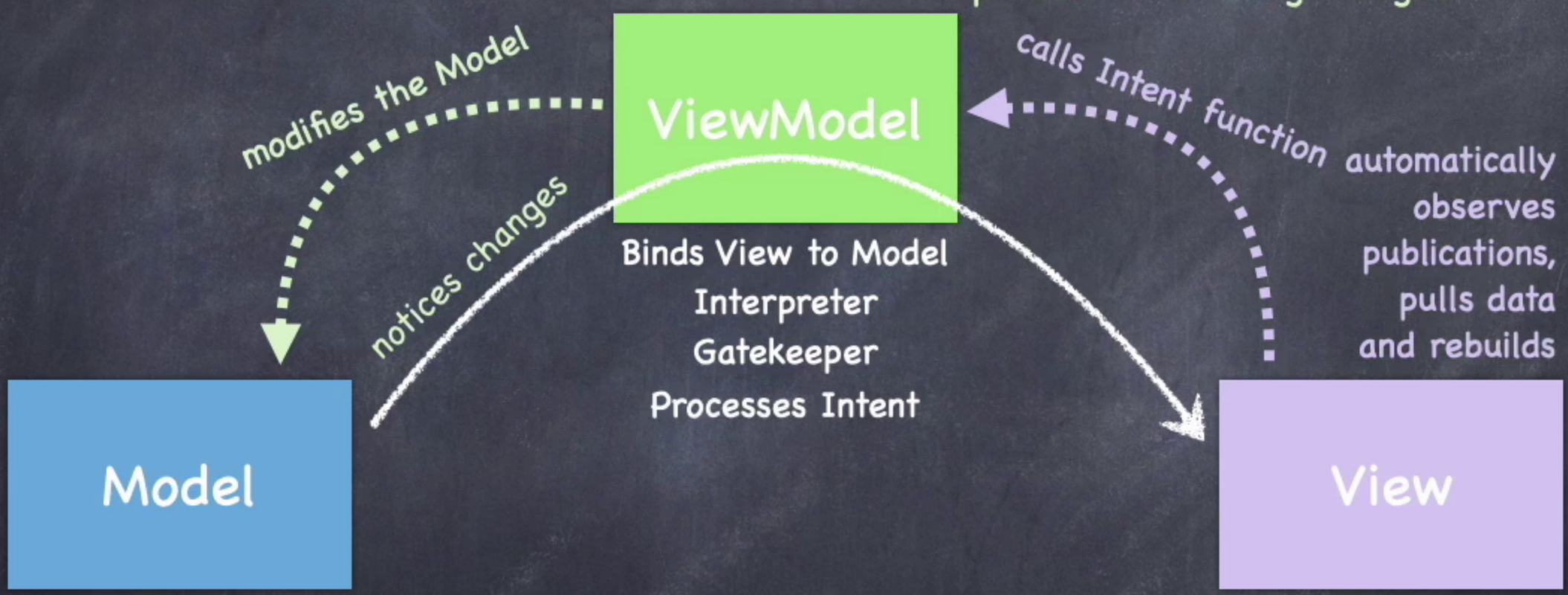
  We call this architecture that connects the Model to the UI in this way MVVM.

  Model-View-ViewModel

  This is the primary architecture for any reasonably complex SwiftUI application.

  You'll also quickly see how #3 (partial separation) is just a minor tweak to MVVM.

MVVM



UI Independent
Data + Logic
"The Truth"

Reflects the Model
Stateless
Declared
Reactive

What about the other direction?

Architecture

- MVVM
  Design paradigm
- Varieties of Types
  `struct`
  `class`
  "don't care" type (aka generics)
  `protocol`
  `enum`
  functions
- Back to the demo!
  Apply MVVM to Memorize

struct and class

- Both `struct` and `class` have …
  - … pretty much exactly the same syntax.
  - stored `var`s (the kind you are used to, i.e., stored in memory)
  - `var isFaceUp: Bool`

struct and class

- Both struct and class have …
  - … pretty much exactly the same syntax.
  - stored vars (the kind you are used to, i.e., stored in memory)
  - computed vars (i.e. those whose value is the result of evaluating some code)

```
var body: some View {
    return Text("Hello World")
}
```

struct and class

- Both struct and class have ...
  - ... pretty much exactly the same syntax.
  - stored vars (the kind you are used to, i.e., stored in memory)
  - computed vars (i.e. those whose value is the result of evaluating some code)
  - constant lets (i.e. vars whose values never change)

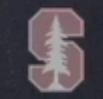
```
let defaultColor = Color.orange
. . .
CardView().foregroundColor(defaultColor)
```

struct and class

- Both struct and class have ...
  - ... pretty much exactly the same syntax.
  - stored vars (the kind you are used to, i.e., stored in memory)
  - computed vars (i.e. those whose value is the result of evaluating some code)
  - constant lets (i.e. vars whose values never change)
  - functions

```
func multiply(operand: Int, by: Int) -> Int {
    return operand * by
}
multiply(operand: 5, by: 6)
```

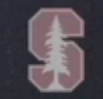
```
func multiply(_ operand: Int, by otherOperand: Int) -> Int {
    return operand * otherOperand
}
multiply(5, by: 6)
```



struct and class

- Both struct and class have …
  - … pretty much exactly the same syntax.
  - stored vars (the kind you are used to, i.e., stored in memory)
  - computed vars (i.e. those whose value is the result of evaluating some code)
  - constant lets (i.e. vars whose values never change)
  - functions
  - initializers (i.e. special functions that are called when creating a struct or class)

```
struct RoundedRectangle {
    init(cornerRadius: CGFloat) {
        // initialize this rectangle with that cornerRadius
    }
    init(cornerSize: CGSize) {
        // initialize this rectangle with that cornerSize
    }
}
```



struct and class

- Both struct and class have …
  - … pretty much exactly the same syntax.
  - stored vars (the kind you are used to, i.e., stored in memory)
  - computed vars (i.e. those whose value is the result of evaluating some code)
  - constant lets (i.e. vars whose values never change)
  - functions
  - initializers (i.e. special functions that are called when creating a struct or class)

```
struct MemoryGame {
    init(numberOfPairsOfCards: Int) {
        // create a game with that many pairs of cards
    }
}
```

struct and class

- Both struct and class have ...
  - ... pretty much exactly the same syntax.
  - stored vars (the kind you are used to, i.e., stored in memory)
  - computed vars (i.e. those whose value is the result of evaluating some code)
  - constant lets (i.e. vars whose values never change)
  - functions
  - initializers (i.e. special functions that are called when creating a struct or class)
  - So what's the difference between struct and class?

struct and class

struct

Value type

Copied when passed or assigned

Copy on write

Functional programming

No inheritance

"Free" init initializes ALL vars

Mutability is explicit (var vs let)

Your "go to" data structure

Everything you've seen so far is a struct
(except View which is a protocol)

class

Reference type

Passed around via pointers

Automatically reference counted

Object-oriented programming

Inheritance (single)

"Free" init initializes NO vars

Always mutable

Used in specific circumstances

The ViewModel in MVVM is always a class
(also, UIKit (old style iOS) is class-based)

Generics

- Sometimes we just don't care

  We may want to manipulate data structures that we are "type agnostic" about.
  In other words, we don't know what type something is and we don't care.

  But Swift is a strongly-typed language, so we don't use variables and such that are "untyped."

  So how do we specify the type of something when we don't care what type it is?

  We use a "don't care" type (we call this feature "generics") ...

Generics

- Example of a user of a "don't care" type: `Array`

  Awesome example of generics: `Array`.
  An `Array` contains a bunch of things and it doesn't care at all what type they are!

  But inside `Array`'s code, it has to have variables for the things it contains. They need types.
  And it needs types for the arguments to `Array` functions that do things like adding items to it.

  Enter ... GENERICS.

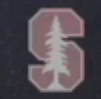
Generics

- How Array uses a "don't care" type

  Array's declaration looks something like this ...

```
struct Array<Element> {
    ...
    func append(_ element: Element) { ... }
}
```

  The type of the argument to append is Element. A "don't care" type.



Generics

- How Array uses a "don't care" type

  Array's declaration looks something like this ...

```
struct Array<Element> {
    ...
    func append(_ element: Element) { ... }
}
```

  The type of the argument to `append` is `Element`. A "don't care" type.

  Array's implementation of `append` knows nothing about that argument and it does not care.

  `Element` is not any known `struct` or `class` or `protocol`, it's just a placeholder for a type.

  The code for using an Array looks something like this ...

```
var a = Array<Int>()
a.append(5)
a.append(22)
```

  When someone uses Array, that's when `Element` gets determined (by `Array<Int>`).

Generics

- How Array uses a "don't care" type

  Array's declaration looks something like this ...

```
struct Array<Element> {
    ...
    func append(_ element: Element) { ... }
}
```

  Note that Array has to let the world know the names of all of its "don't care" types in its API.

  It does this with the < > notation on its struct declaration Array<Element> above.

  That's how users of Array know that they have to say what type Element actually is.

```
var a = Array<Int>()
```

  It is perfectly legal to have multiple "don't care" types in the above (e.g. <Element, Foo>)

Generics

- Type Parameter

  It is illustrative to refer to these types like `Element` in `Array` as a "don't care" type (since if you ask `Array` what type its elements are, it will say "I don't care")

  But its actual name is Type Parameter

  Other languages most of you may know (e.g. Java) have a similar feature
  However, Swift combines this with `protocols` to take it all to the next level
  We'll start to see this in the demo today and we'll talk about it more next week

  Speaking of `protocols` ...

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 vars 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 }
}
```

  Any type can now claim to implement Moveable …

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

  PortableThing now conforms to (aka "behaves like a") Moveable

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

- What is a `protocol` used for?

  A `protocol` is a type.

  So (with certain restrictions), it can be used in the normal places you might see a type.

  (especially with the addition of the keywords `some` and `any`)

  For example, it can be the type of a `var` or a `return` type (like `var body`'s return type).

  We are not going to talk about that sort of use right now.

  Since you've already seen it with `var body`, you probably get the general idea.

  But other than `var body`'s return type, this is actually not the most common use of a `protocol`.

  Instead, let's talk about the most powerful use of a `protocol` ...

protocol

- What is a `protocol` used for?

  Specifying the behavior of a `struct`, `class` or `enum`.

  ```
  struct ContentView: View
  ```

  Just by doing this, `ContentView` became a very powerful `struct`!
  Of course, `ContentView` did have to implement `var body` to participate in being a `View`, but still.

  We call this process "constrains and gains".
  A `protocol` can constrain another type (for example, a `View` has to implement `var body`).
  But a `protocol` can also supply huge gains (e.g. the 100's of functions a `View` gets for free).

  We'll see quite a variety of `protocols` in the coming weeks.
  Examples: `Identifiable`, `Hashable`, `Equatable`, `CustomStringConvertible`.
  And more specialized ones like `Animatable`.

protocol

- What is a `protocol` used for?

  Another use we'll see is turning "don't cares" into "somewhat cares".

  ```
  struct Array<Element> where Element: Equatable
  ```

  If `Array` were declared this way, then only things that are "equatable" could be put in Arrays.

  This is at the heart of "`protocol`-oriented programming".
  We'll be doing this in the demo today.

protocol

- More about Protocols in part two

  A `protocol` becomes massively more powerful via something called an `extension`.
  We'll cover how we combine `protocols` and `extensions` in "part two" of `protocols`.
  This will explain how the "gains" part of "constrains and gains" is implemented.

  We'll also learn more about using `protocols` as types in the same ways we use any other type.
  And how the `some` and `any` keywords help us do that.

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.

  It's also a way to add a lot of functionality (via extension) based on a protocol's primitives.
  This is what "functional (or protocol-oriented) 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.

Functions as Types

- **Functions are types**

  You can declare a variable (or parameter to a `func` or whatever) to be of type "function".
  The syntax for this includes the types of the arguments and return value.
  You can do this anywhere any other type is allowed.

  Examples ...

```
(Int, Int) -> Bool   // takes two Ints and returns a Bool
(Double) -> Void     // takes a Double and returns nothing
() -> Array<String> // takes no arguments and returns an Array of Strings
() -> Void           // takes no arguments and returns nothing (this is a common one)
```

  All of the above are just types. No different than `Bool` or `View` or `Array<Int>`. All are types.

```
var foo: (Double) -> Void // foo's type: "function that takes a Double, returns nothing"
func doSomething(what: () -> Bool) // what's type: "function, takes nothing, returns Bool"
```

Functions as Types

- Functions are types

```
var operation: (Double) -> Double
```

This is a var called `operation`.

It's type is "function that takes a Double and returns a Double".

Here's a simple function that takes a Double and returns a Double ...

```
func square(operand: Double) -> Double {
    return operand * operand
}

operation = square // just assigning a value to the operation var, nothing more
let result1 = operation(4) // result1 would equal 16
```

Note that we don't use argument labels (e.g. `operand:`) when executing function types.

```
operation = sqrt // sqrt is a built-in function which happens to take and return a Double
let result2 = operation(4) // result2 would be 2
```

We'll soon see an example of using a function type for a parameter to a function in our demo.

Functions as Types

- Closures

  It's so common to pass functions around that we are very often "inlining" them.

  We call such an inlined function a "closure" and there's special language support for it.
  We've already using these a lot (`@ViewBuilders` are closures, so is `onTapGesture`'s action).
  We'll peel back the layers on this in the demo and again later in the quarter.

  Remember that we are mostly doing "functional programming" in SwiftUI.
  As the very name implies, "functions as types" is a very important concept in Swift. Very.

Architecture

- MVVM
  Design paradigm
- Varieties of Types
  `struct`
  `class`
  `protocol` (part one)
  "don't care" type (aka generics)

  functions
- Back to the demo!
  Apply MVVM to Memorize