IS 543 Fall 2023

Mobile Platform Development

Project 2 Q&A

Property Wrappers

Dr. Stephen W. Liddle
782 TNRB, office hours Tu 9:30-10:30am, Th 2-3pm
is543@byu.edu
801-422-8792 (rings on my cell)
Zoom: https://bit.ly/liddlezoom

Today

- How was the forum with Dr. Wilcox?
- Project 2 Q&A
- Property wrappers
  We've been using them, but let's see what's going on behind the scenes

Project 2 Q&A

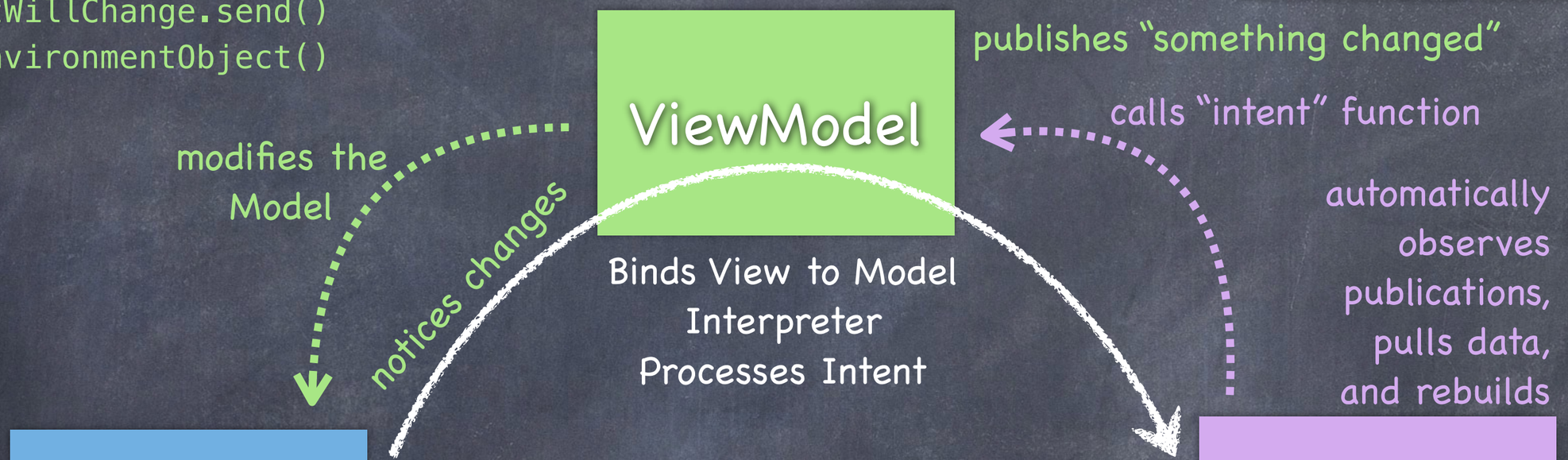
- Any questions you'd like to discuss together as a class?
  (If we end early, I'll stick around and answer individual questions too)

@Observable

- The `@Observable` macro does a lot for us:
  Makes the ViewModel class inherit from `ObservableObject`
  Ensures that all `var`s in the ViewModel are marked as `@Published`, which is a property wrapper
  Ensures that references to the ViewModel are marked with `@ObservedObject`
- (It's a little confusing sometimes: `@` marks macros and property wrappers)

MVVM

Review (Day 2)




```
@Observable
ObservableObject
@Published
objectWillChange.send()
.environmentObject()
```

Model

UI Independent
Data + Logic
"The Truth"

```
@ObservedObject
@Binding
.onReceive
@EnvironmentObject
```

View

Reflects the Model
Stateless
Declared
Reactive

Property Wrappers

- A property wrapper:
  - Is written as an attribute on a property `var`, i.e. `@Something`
  - Is a `struct`
  - Encapsulates some template behavior applied to the wrapped `var`
- Examples:
  - Making a `var` live in the heap so it can be shared among values (`@State`)
  - Making a `var` publish its changes to observers (`@Published`)
  - Causing a `View` to redraw when a published change is detected (`@ObservedObject`)
- "Syntactic sugar"
  - The property wrapper feature of Swift adds "syntactic sugar" to make these `struct`s easy to create and use

Property Wrappers

- Property wrapper syntactic sugar

```
@Published var game: ConcentrationGame<String>? = createGame()
```

… is really just this struct …

```
struct Published {
    var wrappedValue: ConcentrationGame<String>?
    var projectedValue: Publisher<ConcentrationGame<String>?, Never>
}
```

… and Swift (approximately) makes these vars available to you …

```
var _game: Published = Published(wrappedValue: createGame())
var game: ConcentrationGame<String>? {
    get { _game.wrappedValue }
    set { _game.wrappedValue = newValue }
}
```

But wait! There's more. There's also another var inside property wrapper structs …

You access this var using $, e.g. $game

Its type is up to the property wrapper

(e.g. Published's type is a Publisher<ConcentrationGame<String>?, Never>)

Property Wrappers

- Why property wrappers?

  As the name suggests: so the wrapper `struct` can do something on get/set of the `wrappedValue`

- `@Published`

  When `wrappedValue` is set, `Published` does two things:

  - Publishes the change through its `projectedValue` (e.g. `$game`) which is a `Publisher`
  - Also invokes `objectWillChange.send()` in its enclosing `ObservableObject`

  Let's look at the actions and projected values of some other property wrappers we know...

Property Wrappers

- `@State`

  The `wrappedValue` is: anything (but almost always a value type, since reference types are already stored on the heap)

  What it does: stores the `wrappedValue` in the heap; when it changes, invalidates the `View` (causing the `View` to redraw)

  Projected value (i.e `$`): a `Binding` (to that value in the heap)

- `@ObservedObject`

  The `wrappedValue` is: anything that implements the `ObservableObject` protocol (e.g. view models)

  What it does: invalidates the `View` when `wrappedValue` does `objectWillChange.send()`

  Projected value (i.e `$`): a `Binding` (to the `var`s of the `wrappedValue`, which is a view model)

- `@Binding`

  The `wrappedValue` is: a value that is bound to something else

  What it does: gets/sets the value of the `wrappedValue` from some other source

  What it does: when the bound-to value changes, it invalidates the `View`

  Projected value (i.e `$`): a `Binding` (`self`; i.e. the `Binding` itself)

Property Wrappers

- Where do we use `Bindings`?
  EVERYWHERE!
  Getting text out of a TextField, the choice out of a `Picker`, etc.
  Using a `Toggle` or other state-modifying UI element
  Finding out which item in a `NavigationStack` was chosen
  Finding out whether we're being targeted with a drag (the `isTargeted` argument to `onDrop`)
  Binding our gesture state to the `.updating` function of a gesture
  Knowing about (or causing) a modally presented `View`'s dismissal
  In general, breaking our `Views` into smaller pieces (and sharing data with them)
  And so many more places

  `Bindings` are all about having a single source of the truth!
  We don't ever want to have state stored in, say, our view model and also in `@State` in our `View`
  Instead, we would use a `@Binding` to the desired `var` in our view model
  Nor do we want two different `@State vars` in two different `Views` to store the same thing
  Instead, one of the two `@State vars` would want to be a `@Binding`

Property Wrappers

- Where do we use Bindings?

  Sharing @State (or an @ObservedObject's vars) with other Views

```
struct MyView: View {
    @State var myString = "Hello"
    var body: View {
        OtherView(sharedText: $myString)
    }
}
struct OtherView: View {
    @Binding var sharedText: String
    var body: View {
        Text(sharedText)
    }
}
```

  OtherView's body is a Text whose String is always the value of myString in MyView

  OtherView's sharedText is bound to MyView's myString

Special Binding Cases

- Binding to a constant value

  You can create a Binding to a constant value with Binding.constant(value)

  E.g. OtherView(sharedText: .constant("Howdy")) will always show Howdy in OtherView

- Computed Binding

  You can even create your own "computed Binding"

  We won't get into the details here, but check out Binding(get:, set:)

Another Property Wrapper Type

- `@EnvironmentObject`

  Same as `@ObservedObject`, but passed to a `View` in a different way

  ```
  let myView = MyView().environmentObject(theViewModel)
  ```

  vs.

  ```
  let myView = MyView(viewModel: theViewModel)
  ```

  Inside the `View`:

  ```
  @EnvironmentObject var viewModel: ViewModelClass
  ```

  vs.

  ```
  @ObservedObject let viewModel = ViewModelClass()
  ```

  Otherwise the code inside the `Views` would be the same

  Biggest difference between the two?
  Environment objects are visible to all `Views` in your `body` (except modally presented ones)
  So it is sometimes used when a number of `Views` are going to share the same view model
  When presenting modally (more on that later), you will want to use `@EnvironmentObject`
  Can only use one `@EnvironmentObject` wrapper per `ObservableObject` type per `View`

Another Property Wrapper Type

- `@EnvironmentObject`

  The wrappedValue is: `ObservableObject` obtained via `.environmentObject()` sent to the `View`
  What it does: invalidates the `View` when `wrappedValue` performs `objectWillChange.send()`
  Projected value (i.e `$`): a `Binding` (to the `var`s of the `wrappedValue`, which is a view model)

- Best practices?

  See this Hacking with Swift article — TL;DR:

  Use `@State` for simple properties that belong to a single view (should be marked private)
  Use `@ObservedObject` for complex properties that might belong to several views — the default technique for reference types
  Use `@StateObject` once for each observable object you use, in whichever part of your code is responsible for creating it
  Use `@EnvironmentObject` for properties that were created elsewhere in the app (shared data)

  => `@ObservedObject` is your go-to default and covers most cases; `@EnvironmentObject` is useful when many Views in the view hierarchy need to share the same View Model

Yet Another Property Wrapper Type

- `@Environment`

  Unrelated to `@EnvironmentObject` — totally different thing (!!)

  Property wrappers can have yet more variables than `wrappedValue` and `projectedValue`
  They are just normal `structs`

  You can pass values to set these other vars using `()` when you use the property wrapper
  E.g. `@Environment(\.colorScheme) var colorScheme`
  In `Environment`'s case, the value that you're passing (e.g. `\.colorScheme`) is a key path
  It specifies which instance variable to look at in an `EnvironmentValues` struct
  See the documentation of `EnvironmentValues` for what's available (there are many)

  Notice that the `wrappedValue`'s type is internal to the `Environment` property wrapper
  Its type will depend on which key path you're asking for
  In our example above, the `wrappedValue`'s type will be `ColorScheme`
  `ColorScheme` is an enum with values `.dark` and `.light`
  So this is how you know whether your `View` is drawing in dark mode or light mode right now

Yet Another Property Wrapper Type

- `@Environment`
  The wrappedValue is: the value of some `var` in `EnvironmentValues`
  What it does: sets/gets a value of some `var` in `EnvironmentValues`
  Projected value (i.e `$`): none

Publisher

- The "light" explanation
  We could talk in greater detail about `Publisher`s later, but let's start with a basic understanding
- What is a `Publisher`?
  It is an object that emits values and possibly a failure object if it fails while doing so
  `Publisher<Output, Failure>`
  `Output` is the type of the thing this `Publisher` publishes
  `Failure` is the type of the thing it communicates if it fails while trying to publish
  It doesn't care what `Output` or `Failure` are (though `Failure` must implement `Error`)
  If the `Publisher` does not deal with errors, the `Failure` can be `Never`
- What can we do with a `Publisher`?
  Listen to it (subscribe to get its values and find out when it finishes publishing and why)
  Transform its values on the fly
  Shuttle its values off to somewhere else
  And so much more!

Publisher

- Listening (subscribing) to a Publisher

  There are many ways to do this, but here are a couple of simple yet powerful ways

  You can simply execute a closure whenever a Publisher publishes

```
cancellable = myPublisher.sink(
    receiveCompletion: { result in ... }, // result is a Completion<Failure> enum
    receiveValue: { thingThePublisherPublishes in ... }
)
```

  If the Publisher's Failure is Never, then you can leave out the receiveCompletion above

  Note that .sink returns something (which we assign to cancellable here)
  The returned thing implements the Cancellable protocol
  Very often we will type erase this to AnyCancellable (just like with AnyTransition)
  What is its purpose?
    a) you can send .cancel() to it to stop listening to that publisher
    b) it keeps the .sink subscriber alive

  Always keep this var somewhere that will stick around as long as you want the .sink to!

Publisher

- Listening (subscribing) to a `Publisher`

  A `View` can listen to a `Publisher` too

```
.onReceive(publisher) { thingThePublisherPublishes in
    // do whatever you want with thingThePublisherPublishes
}
```

  Note that `.onReceive` will automatically invalidate your `View`, causing a redraw

Publisher

- Where do Publishers come from?
  - $ in front of vars marked @Published
  - URLSession's dataTaskPublisher (publishes the Data obtained from a URL)
  - Timer's publish(every:) (periodically publishes the current date and time as a Date)
  - NotificationCenter's publisher(for:) (publishes notifications when system events happen)
- Other stuff we can do with a Publisher
  - There's more you can do with a Publisher
  - But we'll give a couple of examples in our demo so you get a flavor of it

Coming Up...

- More Project 2 Q&A, plus miscellaneous topics as we have time
  - There is much we haven't talked about yet
  - Let me know if there are topics you're especially interested in