Today

**Property Wrappers**
Finally we get to talk about what @State, @Published, @ObservedObject really are!
This is a very, very important topic.

**Publishers**
Just a very “light” treatment of this topic for now.

**Demo**
Publishers
Palette Chooser (@Binding)
Property Wrappers

Property Wrappers
All of these @Something statements are property wrappers.
A property wrapper is actually a struct.
These structs encapsulate some “template” behavior applied to the vars they wrap.

Examples …
Making a var live in the heap (@State)
Making a var publish its changes (@Published)
Causing a View to redraw when a published change is detected (@ObservedObject)

The property wrapper feature adds “syntactic sugar” to make these structs easy to create/use.
Property Wrappers

Property Wrapper Syntactic Sugar

@Published var emojiArt: EmojiArt = EmojiArt()
... is really just this struct ...

struct Published {
    var wrappedValue: EmojiArt
    var projectedValue: Publisher<EmojiArt, Never>
}
... and Swift (approximately) makes these vars available to you ...

var _emojiArt: Published = Published(wrappedValue: EmojiArt())
var emojiArt: EmojiArt {
    get { _emojiArt.wrappedValue }
    set { _emojiArt.wrappedValue = newValue }
}

But wait! There’s more. There’s also another var inside Property Wrapper structs ...

You access this var using $, e.g. $emojiArt.
Its type is up to the Property Wrapper. Published’s is a Publisher<EmojiArt, Never>. 
Property Wrappers

Why?
Because, of course, the Wrapper struct does something on set/get of the wrappedValue.

@Published
So what does Published do when its wrappedValue is set (i.e. changes)?
It publishes the change through its projectedValue ($emojiArt) which is a Publisher.
It also invokes objectWillChange.send() in its enclosing ObservableObject.

Let's look at the actions and projected value of some other Property Wrappers we know ...
Property Wrappers

@State
The wrappedValue is: anything (but almost certainly a value type).
What it does: stores the wrappedValue in the heap; when it changes, invalidates the View.
Projected value (i.e. $): a Binding (to that value in the heap).

@ObservedObject
The wrappedValue is: anything that implements the ObservableObject protocol (ViewModels).
What it does: invalidates the View when wrappedValue does objectWillChange.send().
Projected value (i.e. $): a Binding (to the vars of the wrappedValue (a ViewModel)).

@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?

All over the freaking place …
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 NavigationView 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 ViewModel and also in @State in our View.
Instead, we would use a @Binding to the desired var in our ViewModel.
Nor do we want two different @State vars in two different Views be storing 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.

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

```swift
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.*
Property Wrappers

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 go into detail here, but check out `Binding(get:, set:)`. 
Property Wrappers

@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 var 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 ViewModel.

When presenting modally (more on that later), you will want to use @EnvironmentObject.

Can only use one @EnvironmentObject wrapper per ObservableObject type per View.
Property Wrappers

@EnvironmentObject

The wrappedValue is: ObservableObject obtained via .environmentObject() sent to the View. What it does: invalidates the View when wrappedValue does objectWillChange.send(). Projected value (i.e. $): a Binding (to the vars of the wrappedValue (a ViewModel)).
Property Wrappers

@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.
Property Wrappers

@Environment
The wrappedValue is: the value of some var in EnvironmentValues. What it does: gets/sets a value of some var in EnvironmentValues. Projected value (i.e. $): none.
The “light” explanation
We’ll talk in much greater detail about Publishers later in the quarter. But let’s start with a basic understanding of them.

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 so many ways to do this, but here’s a couple of simple yet powerful ones ...

You can simply execute a closure whenever a Publisher publishes.

```swift
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!
Listening (subscribing) to a Publisher

A View can listen to a Publisher too.

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

.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

We’re not going to talk in detail today about what you can do with a Publisher. Instead, we’ll give a couple of examples in today’s demo so you get a flavor of it.
Demo

EmojiArt

Let's give better feedback when we're off loading our background image from the internet.

@Published's publisher to autosave.
.onReceive to automatically zoom to fit when our backgroundImage changes.

URLSession publisher to load image.
Add a Palette Chooser (@Binding)