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.

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()
```

Property Wrappers

- Property Wrapper Syntactic Sugar

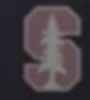
```
@Published var emojiArt: EmojiArt = EmojiArt()
```

… is really just this struct …

```
struct                {



}
```



Property Wrappers

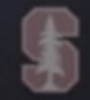
- Property Wrapper Syntactic Sugar

```
@Published var emojiArt: EmojiArt = EmojiArt()
```

… is really just this struct …

```
struct Published {


}
```



Property Wrappers

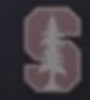
- Property Wrapper Syntactic Sugar

```
@Published var emojiArt: EmojiArt = EmojiArt()
```

… is really just this struct …

```
struct Published {
    var wrappedValue

}
```



Property Wrappers

- Property Wrapper Syntactic Sugar

```
@Published var emojiArt: EmojiArt = EmojiArt()
```

… is really just this struct …

```
struct Published {
    var wrappedValue: EmojiArt

}
```

Property Wrappers

- Property Wrapper Syntactic Sugar

```
@Published var emojiArt: EmojiArt = EmojiArt()
```

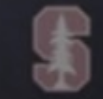
… is really just this struct …

```
struct Published {
    var wrappedValue: EmojiArt

}
```

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

```
var _emojiArt
```



Property Wrappers

- Property Wrapper Syntactic Sugar

```
@Published var emojiArt: EmojiArt = EmojiArt()
```

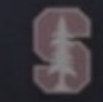
… is really just this struct …

```
struct Published {
    var wrappedValue: EmojiArt

}
```

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

```
var _emojiArt: Published
```



Property Wrappers

- Property Wrapper Syntactic Sugar

```
@Published var emojiArt: EmojiArt = EmojiArt()
```

… is really just this struct …

```
struct Published {
    var wrappedValue: EmojiArt

}
```

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

```
var _emojiArt: Published = Published(wrappedValue: EmojiArt())
```

Property Wrappers

- Property Wrapper Syntactic Sugar

```
@Published var emojiArt: EmojiArt = EmojiArt()
```

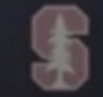
… is really just this struct …

```
struct Published {
    var wrappedValue: EmojiArt

}
```

… 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 }
}
```



Property Wrappers

- Property Wrapper Syntactic Sugar

```
@Published var emojiArt: EmojiArt = EmojiArt()
```

… is really just this struct …

```
struct Published {
    var wrappedValue: EmojiArt
    var projectedValue
}
```

… 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 …

Property Wrappers

- Property Wrapper Syntactic Sugar

```
@Published var emojiArt: EmojiArt = EmojiArt()
```

… is really just this struct …

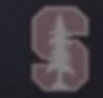
```
struct Published {
    var wrappedValue: EmojiArt
    var projectedValue
}
```

… 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`.



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>
}
```

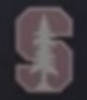
… and Swift (approximately) makes these vars available to you …

```
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`.

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)?

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`.
  We'll talk about what `Publishers` are later in the quarter.
  It also invokes `objectWillChange.send()` in its enclosing `ObservableObject`.

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`.
  We'll talk about what `Publishers` are later in the quarter.
  It also invokes `objectWillChange.send()` in its enclosing `ObservableObject`.

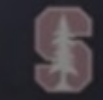
  Let's look at the actions and projected value of some other Property Wrappers we know ...

Property Wrappers

- `@State`

  The `wrappedValue` is: any value type.

  What it does: stores the `wrappedValue` in the heap; when it changes, invalidates the `View`.



Property Wrappers

- @State

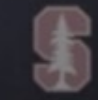
  The wrappedValue is: any value type.

  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).

  It's a bit tricky to initialize an @State in your View's init ...

```
@State private var foo: Int
init() {
    _foo = .init(initialValue: 5)
}
```

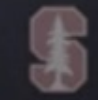


Property Wrappers

- `@StateObject`

  Like `@State`, but for `ObservableObjects` (aka ViewModels) instead of for value types.

  Behaves just like an `@ObservedObject var`, except ...

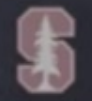


Property Wrappers

- `@StateObject`

  Like `@State`, but for `ObservableObjects` (aka ViewModels) instead of for value types.
  Behaves just like an `@ObservedObject var`, except ...

  An `@StateObject` is a "source of truth" (like `@State` is for `View`-local value types).
  Always create it with an initial value, e.g. `@StateObject var foo = SomeObservableObject()`.
  An `@ObservedObject` is not a source of truth (it's a reference to a source of truth).
  So you should never do `@ObservedObject var foo = SomeObservableObject()`.



Property Wrappers

- `@StateObject`

  Like `@State`, but for `ObservableObjects` (aka ViewModels) instead of for value types.
  Behaves just like an `@ObservedObject var`, except ...

  An `@StateObject` is a "source of truth" (like `@State` is for View-local value types).
  Always create it with an initial value, e.g. `@StateObject var foo = SomeObservableObject()`.
  An `@ObservedObject` is not a source of truth (it's a reference to a source of truth).
  So you should never do `@ObservedObject var foo = SomeObservableObject()`.

  Can be used in a View or in an App or in a Scene (which we'll cover later).

Property Wrappers

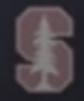
- `@StateObject`

  Like `@State`, but for `ObservableObjects` (aka ViewModels) instead of for value types.
  Behaves just like an `@ObservedObject var`, except ...

  An `@StateObject` is a "source of truth" (like `@State` is for `View`-local value types).
  Always create it with an initial value, e.g. `@StateObject var foo = SomeObservableObject()`.
  An `@ObservedObject` is *not* a source of truth (it's a *reference* to a source of truth).
  So you should *never* do `@ObservedObject var foo = SomeObservableObject()`.

  Can be used in a `View` or in an `App` or in a `Scene` (which we'll cover later).

  If in a `View`, the *lifetime* of the ViewModel will be tied to that of the `View`.
  In other words, when the `View` goes off-screen, the `@StateObject` will get tossed.
  Sometimes you want this.
  e.g. a `FetchedImageView` (we want to dump the image bits when `View` goes off screen)
  Most times, it's *not what you want*.
  e.g. anytime you share a ViewModel with any other `View` *that is not your child*

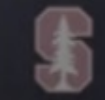


Property Wrappers

- `@StateObject` and `@ObservedObject`

  The `wrappedValue` is: `anything that implements the ObservableObject protocol` (ViewModels).

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

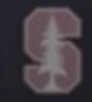


Property Wrappers

- `@StateObject` and `@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` ViewModel.

  The projected value `Binding` can be to any `var` expression, including subscripts.
  e.g. `$myViewModel.data.stuff[3]` is legal
  But be careful if you bind to something that can disappear out from under the `Binding`.
  For example, if we shortened the `stuff` array to only have 2 items in it.



Property Wrappers

- `@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`.

Property Wrappers

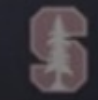
- @Binding

  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.

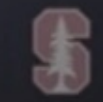
  What it does: when the bound-to value changes, it invalidates the View.

  Projected value (i.e. $): a Binding to the Binding itself



Property Wrappers

- Where do we use `Binding`s?
  - All over the freaking place ...



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.

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 to the other.

Property Wrappers

- Where do we use Bindings?

```
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)
        TextField("Shared", text: $sharedText)
    }
}
```

OtherView's sharedText is bound to MyView's myString.
Changing sharedText changes myString (and vice versa).

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`.

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

  Exactly the same as @ObservedObject, but passed to a View in a different way ...

Property Wrappers

- @EnvironmentObject

  Exactly the 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.

Property Wrappers

- @EnvironmentObject

  Exactly the 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
  ```

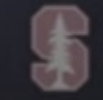
  Otherwise the code inside the Views would be the same.

  Biggest difference between the two?

  Environment objects are visible to all Views in your body (and thus to their body's Views).

  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.



Property Wrappers

- @EnvironmentObject

  Exactly the 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 (and thus to their body's Views).

  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.

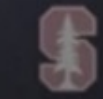
Property Wrappers

- @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 ViewModel.



Property Wrappers

- `@Environment`

  Unrelated to `@EnvironmentObject`. Totally different thing.

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`.

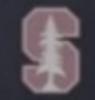
Property Wrappers

- `@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).



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`

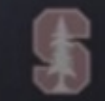
  Unrelated to `@EnvironmentObject`. Totally different thing.

  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.

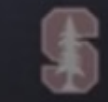
  ```
  view.environment(\.colorScheme, .dark)
  ```



Property Wrappers

- `@Environment`

  The `wrappedValue` is: the `value of some var in EnvironmentValues`.

  What it does: gets a value of some var in `EnvironmentValues`.

  Projected value (i.e. $): `none`.