Building Apps with SwiftUI

CMSC 436

MVB Model

Recall the MVVM/MVB Model

- Model-View-*ViewModel/Binder*
- We'll call it *Binder*

Everything done in Swift (this wasn't true with UIKit)

Structure of an App

```
import SwiftUI

@main                           // Attribute
struct FooApp : App {           // "App" is a protocol
   var body : some Scene {      // "Scene" is a protocol
      WindowGroup {             // provided concrete Scene
         ContentView()              // our View
      }
   }
}
```

This is the *minimal app*

`some Scene` creates an *opaque type*
You need this for SwiftUI, but otherwise you hopefully can ignore it

A `Scene` is a top-level UI element (like a window)

Structure of a View

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Hello, world!")
            .padding()
    }
}
```

This is the auto-generated “Hello, world!” view

The auto-generated one will be called `ContentView`
Has a hook to *preview* the `View`

The View Protocol

Only a single requirement: **body** property.

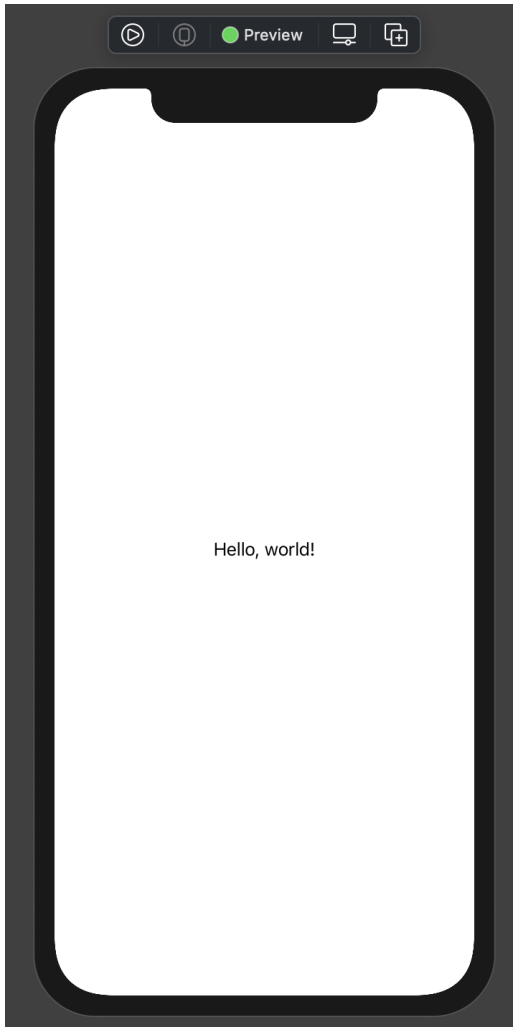
```
@available(iOS 13.0, *)
public protocol View {
    /// The type fo view representing the body of this view
    associatedtype Body : View

    /// The content and behavior of the view
    @ViewBuilder var body: Self.Body { get }
}
```

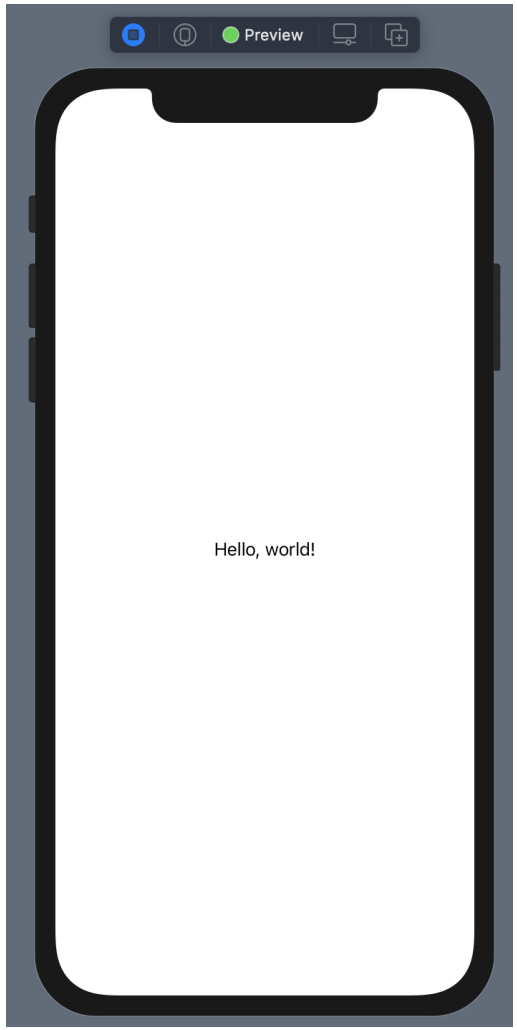
This is the auto-generated “Hello, world!” view

The auto-generated one will be called `ContentView`
Has a hook to *preview* the `View`

Previewing in Xcode

You don't always need the simulator!





The left-most button of the toolbar can start a *Live Preview*

More Complex Views

`HStack` and `VStack`

- "stacks" of horizontally- or vertically-distributed sub-`Views`
- tuple-like lists of sub-`Views`
- can be things like `Text`, `Button`, `HStack`, ...

`ForEach`

- like a `for` loop
- creates a tuple-like list of `Views`
- sequence elements must have an `.id` parameter
- we can use `ForEach([1,2,3],id:\.self)`

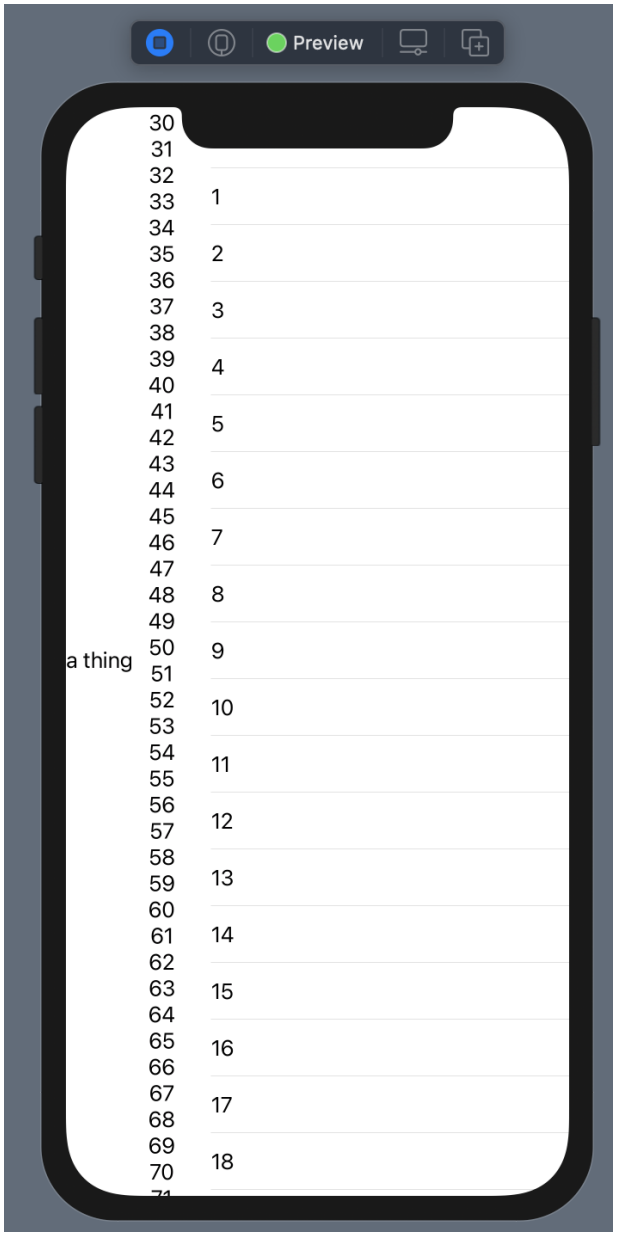
`List`

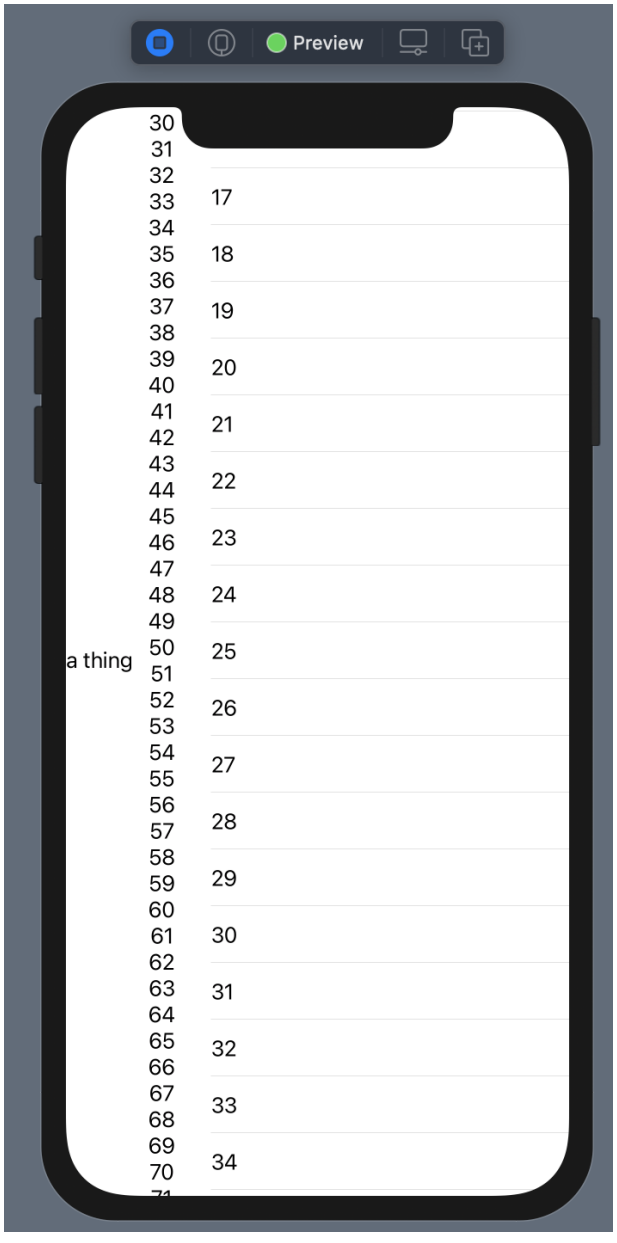
- similar to `ForEach`, but alternative to `VStack`
- scrollable

Dynamic View Example

```
struct ContentView: View {
    var body: some View {
        HStack {
            Text("a thing")
            VStack {
                ForEach(1...10,id:\.self) {
                    i in Text("\(i)")
                }
            }
            List(1...10,id:\.self) {
                i in Text("\(i)")
            }
        }
    }
}
```

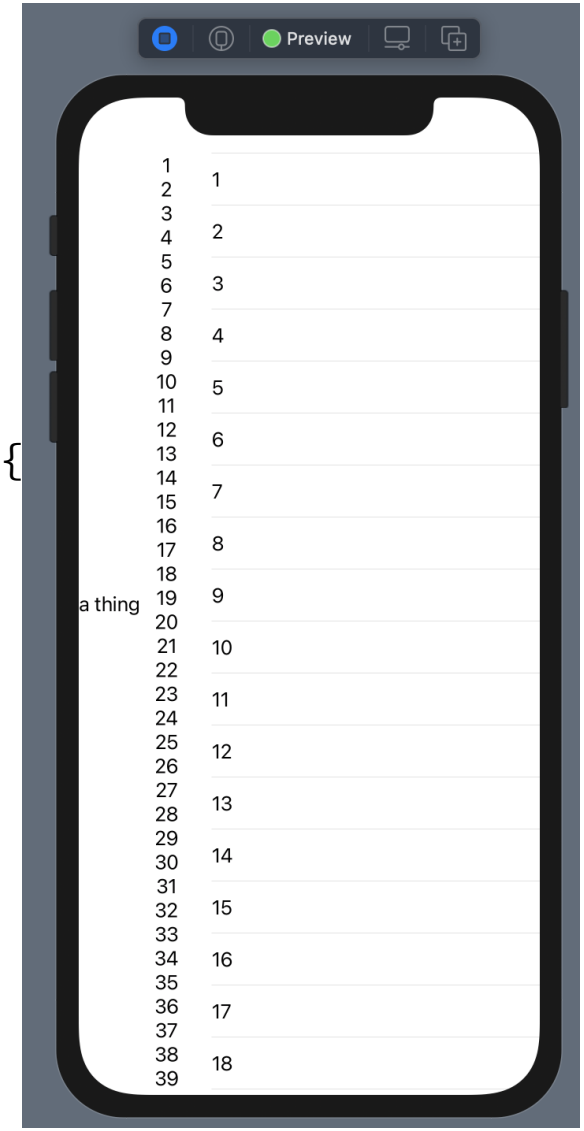
Turn It Up To 100!





Scrollin' Scrollin' Scrollin'

```
struct ContentView: View {
    var body: some View {
        HStack {
            Text("a thing")
            ScrollView {
                VStack {
                    ForEach(1...100,id:\.self) {
                        i in Text("\(i)")
                    }
                }
            }
            List(1...100,id:\.self) {
                i in Text("\(i)")
            }
        }
    }
}
```



Split Things Up

```
struct MyScroll: View {
   private var numRange: ClosedRange<Int>

   init(_ r:ClosedRange<Int>) {
      numRange = r
   }

   var body: some View {
      ScrollView { VStack {
            ForEach(numRange,id:\.self) { i in Text("\(i)") }
      } }
   }
}

struct ContentView: View {
   var body: some View {
      HStack {
         Text("a thing")
         MyScroll(1...100)
         List(1...100,id:\.self) { i in Text("\(i)") }
      }
   }
}
```

The Model

Data and *Logic*

What we're keeping in memory, and how it's used

```
import Foundation

class Values: ObservableObject {
    @Published var maxVal: Int = 5

    func setMax(_ v: Int) {
        if v > 1 {
            maxVal = v
        }
    }
}
```

`ObservableObject` tells Swift that we can build Bindings

`@Published` tells Swift that objects referencing this property should update when it changes

Using Model Bindings

```
struct MyScroll: View {
   private var maxVal: Int

   init(_ v:Int) {
      maxVal = r
   }

   var body: some View {
      ScrollView { VStack {
            ForEach(1...maxVal,id:\.self) { i in Text("\(i)") }
      } }
   }
}

struct ContentView: View {
   @EnvironmentObject var vals: Values

   var body: some View {
      HStack {
         Text("a thing")
         MyScroll(vals.maxVal)
         List(1...100,id:\.self) { i in Text("\(i)") }
      }
   }
}
```

Wiring the Model In

When you create a `ContentView`, you have to call its `environmentObject()` method (typically by the `App`):

- This takes any `ObservableObject` defined with `@StateObject` in app.
- Actual object type must match a property with `@EnvironmentObject` attribute from within the view
- `Views` can pass these to other `Views`
- Also done by `ContentView_Previews`

How Does the Model get Created?

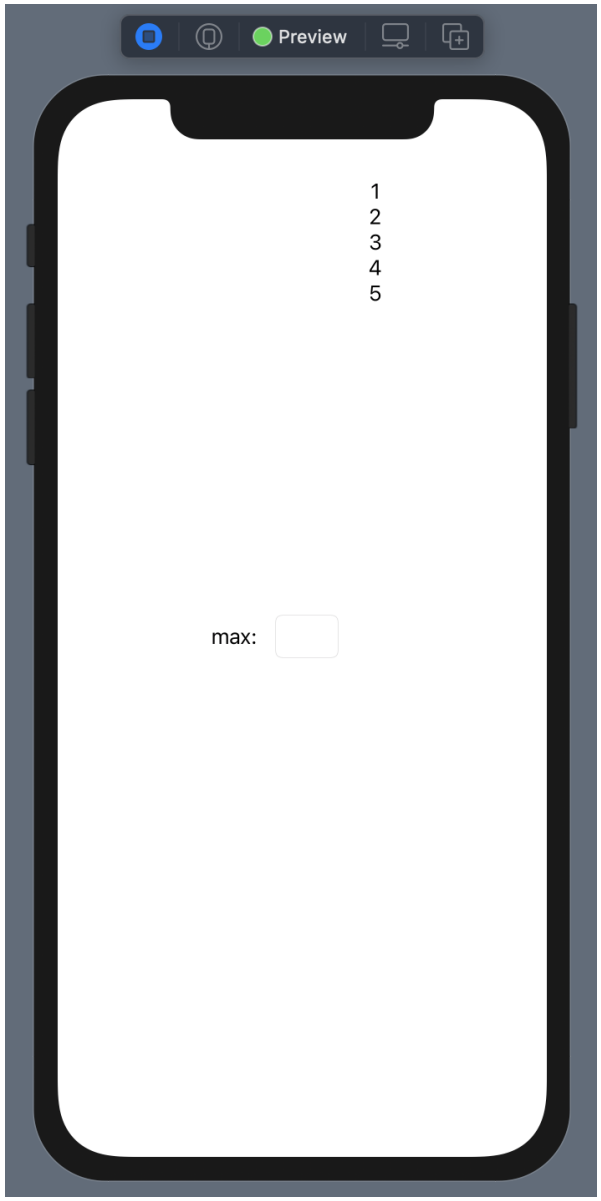
```
@main
struct FooApp: App {
    @StateObject var vals: Values = Values()

    var body: some Scene {
        WindowGroup {
            ContentView().environmentObject(vals)
        }
    }
}
```

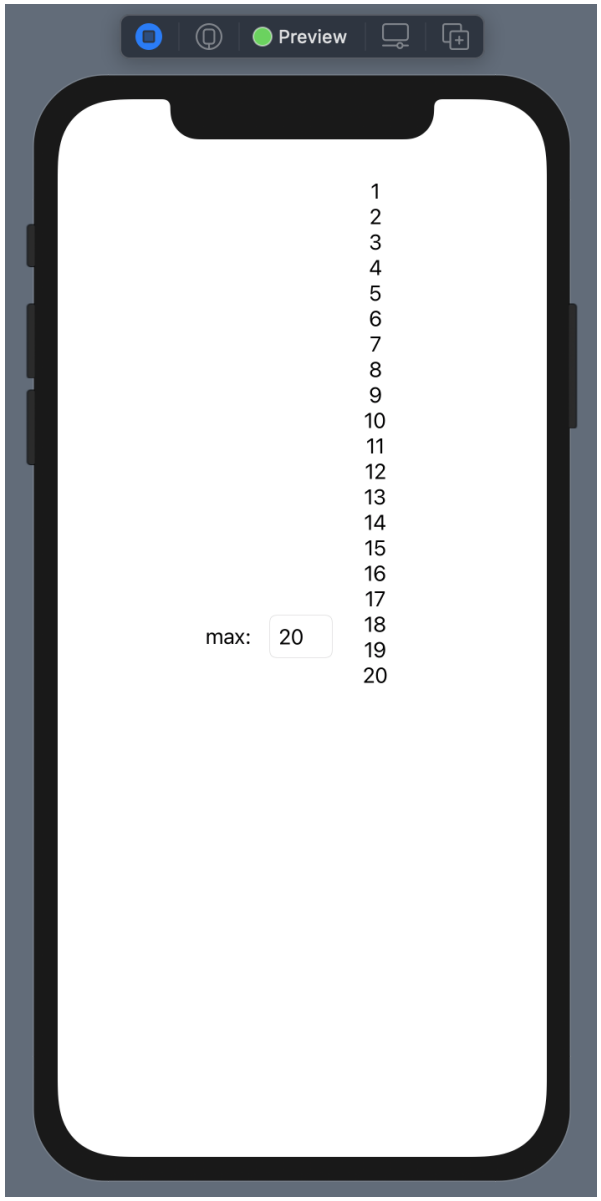
`@StateObject` creates the UI side of Bindings, and informs Swift that this object *owns* the Model instance

A More Dynamic Interface

```
struct ContentView: View {
    @EnvironmentObject var vals: Values
    @State var mvText: String = String()

    var body: some View {
        HStack {
            Text("max:").frame(width:50,height:20)//.padding()
            TextField("",text:$mvText) {
                _ in if let i = Int(mvText) { vals.setMax(i) }
            }
            .frame(width: 50, height: 20, alignment: .center)
            .textFieldStyle(RoundedBorderTextFieldStyle())
            MyScroll(vals.maxVal).padding()
        }
    }
}
```

Our Dynamic View in Action





Bindings Summary

Two types of bindings:

- environmental bindings
  - Defined as `ObservableObjects` with `@Published` properties.
  - Used with `@EnvironmentObject` from (possibly multiple) views.
  - Wired by calling `.environmentObject()` on `ContentView`.
  - must also use `.environmentObject()` to pass a model instance to the preview version of `ContentView` (still in ContentView.swift).
- single-view bindings
  - defined using `@State` in view
  - referenced inside view without `$`
  - use `$` syntax to pass access to another view (this is the "projected value" of property wrappers)
  - only works with value objects