Contents

1 Swift UI Intro 1/10

Definition

SwiftUI → a Swift Framework with Declarative Syntax

- Display views on screen
- Handle user interaction
- Manage state and dataflow

Everything is a "View", similar to Java Interface
Below is a starter app.

```
import SwiftUI
struct ContentView: View {
    var body: some View {
        // Horizontal Stack
        // Moreover, this will automatically change the order
        // In different languages, i.e. Hebrew and Arabic
        HStack {
            // Text
            Text("I Like to ride my bike")

            // images
            // this is the "Share" icon
            Image(systemName: "square.and.arrow.up")
        }
        // Vertical Stack
        VStack {}
    }
}

#Preview {
    ContentView()
}
```

Moreover, we can also change the alignments of code:

```
VStack(alignment: .leading) {
    Image(systemName: "person.circle")
    Text("blah blah")
}
```

Where in general, it is always centered, but we can specify to be left justified via ".leading"

Other parameters mentioned:

– Spacing → Change horizontal or vertical Spacing

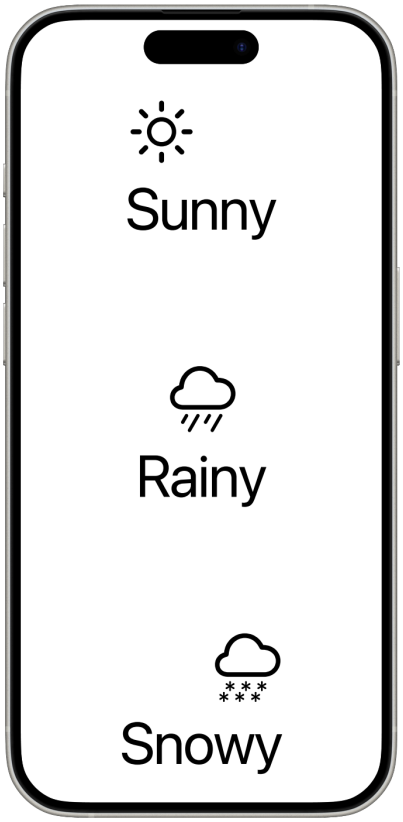


Figure 1: VStack example, .leading, .center, .trailing, respectively

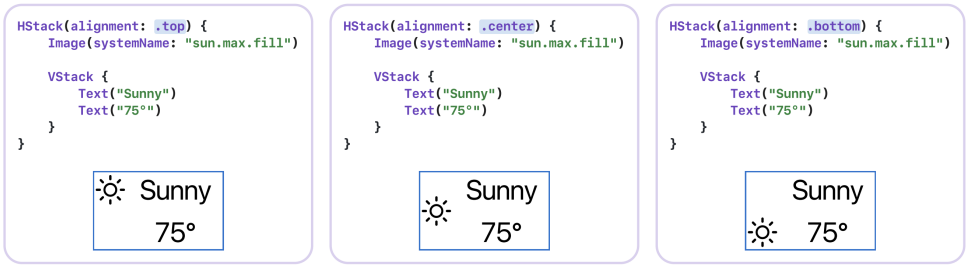


Figure 2: HStack example

2 Intro to Modifiers 1/13

```
Text("View with a background")
    .background {
        Color.blue
    }
Text("View with a border")
    .border(Color.blue)

Text("Padded with border")
    .padding(16)
    .border(Color.blue)
```

".background { ... }" is an example of a view modifier!

Remark 1. It's important to note that the order **matters**. One example is having border before padding, creating the padding outside

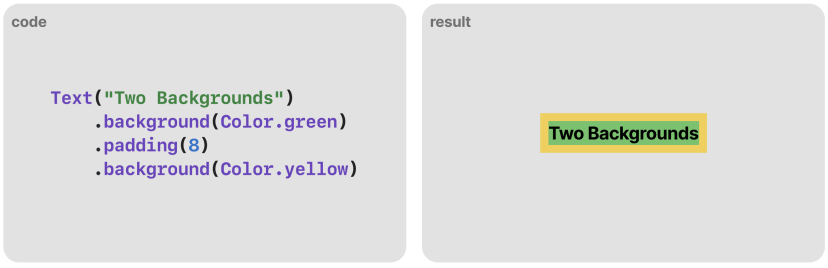


Figure 3: Example of ordering mattering

But how are they implemented?

```
Text("Padding")
.padding(16)

extension View {
    func padding(_ length: CGFloat) -> some View
}
```

Modifiers are instance methods on View that return a new, modified view.

Environment Modifiers, rather than wrapping view with new appearance, wraps and changes data, seen below

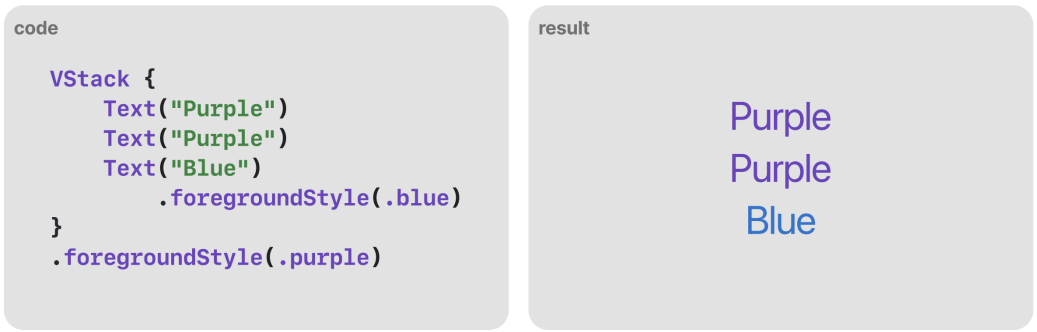


Figure 4: Environment Modifiers

Cannot use on a var, for example, as `.foregroundStyle(.blue)` because it has to be a part of a view, for example, VStack.

Environment is sort of a metadata of our view, that trickles down into the rest. So it would only apply with the ones with `foregroundStyle` after it

3 Swift UI Grab Bag 1/15

This lecture is a general grab bag of Swift UI tools, syntax, and semantics

3.1 Fonts and Font sizes

We can change text sizes and fonts with .font(. . .)

```
Text("Large Title")
    .font(.largeTitle)
```

```
Text("Title)
    .font(.title)
```

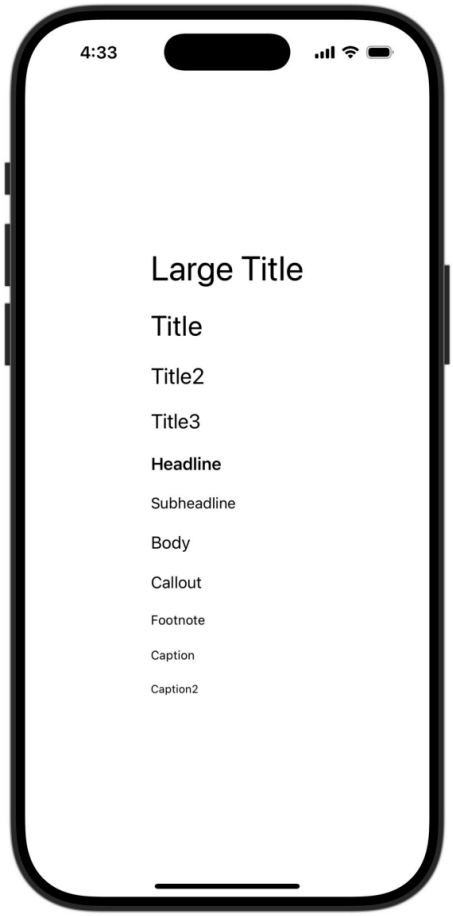


Figure 5: Here is an example with a bunch of sizes

Preset text styles are highly adaptable as there is Dynamic Type, Bold Text, etc.

As such, it can easily change font size based on system settings, making it highly accessible

Further modifications that can be done:

- `.fontWeight(\dots)`
- `.fontDesign(\dots)`
- `.lineLimit(\dots)`
- `.strikethrough()`
- `.underline()`

And there is much more you can do with text, such as markdown support, localization, custom text layout, and text addition, to name a few.

3.2 Colors in SwiftUI

Like Text, SwiftUI's Color provides many defaults and these defaults are also super dynamic with light and dark mode, or increase constrast.

Figure 6: Here is an example of light and dark mode

There is a notion of greedy vs. polite views! For example, colors has no intrinsic value and as such, take up as much space as possible. While text has a given font size, so it only takes up that amount of space.

3.3 SwiftUI Safe Area

The default range that the view stays within, for example, between the dynamic island and home indicator.

Sometimes, we want to ignore this safe area, for example, `.ignoresSafeArea()`, or `.background(_:ignoresSafeArea:)`

3.4 Images

There are symbols and there are images And these symbols react to font size changes as well!

With custom images, Xcode generates static properties to access

Example

Here is an example of Image:

```
Image(.oldWell)
    .resizeable()
    .scaledToFit() // matches shortest
    // .scaledToFill() // matches longest
    .frame(width: 300, height: 200) // constrain height
       and width
    .clipShape(.circle) // can be cropped into a shape,
       for ex. circle
```

4 Introduction to Swift Syntax 1/24

What is the difference between Let and Var?

Definition

var allows user to change the variable

let allows user to set a constant, immutable variable

Argument labels vs. parameter names A parameter can have different external and internal names which allows for better readability

```
func divide (_ a: Double, by b: Double) -> Double {
    return a / b
}

let result = divide(4.0, by: 2.0)
```

Structs vs. classes There are two ways to encapsulate data, struct & classes. Where structs are value types and classes are reference type

Definition

Value Types: directly holds the data

Reference Types: has a reference, or a pointer, to data

4.1 Value Semantics

An example with Class :

```
class BurritoBowl {\dots}

let recentOrder = BurritoBowl(meat: .chicken)

let currentOrder = recentOrder // points to recentOrder
currentOrder.meat = .steak // modifies shared instance

assert(recentOrder.meat == .chicken) // this fails, as
    classes are reference
```

Remark 2. To note, you may notice that recentOrder and currentOrder are defined with let keyword, yet you are able modify the meat type. As current order is a reference but you can still change the values inside that memory address.

An example with Struct :

```
struct BurritoBowl {\dots}

let recentOrder = BurritoBowl(meat: .chicken)

var currentOrder = recentOrder // points to recentOrder
currentOrder.meat = .steak

assert(recentOrder.meat == .chicken) // this succeeds
```

Remark 3. We used var in this example with struct, as now it is ALL the data from the struct, thus you must declare a var in order to change the meat type

Finally, the theme of the lecture was that Swift makes it pretty hard to write buggy code.

4.2 Brief intro to optionals

```
func findMax(in numbers: [Int]) -> Int {
    var max = -Int.max

    for numbers in numbers {
        if number > max {
            max = number
        }
    }
    return max
}
```

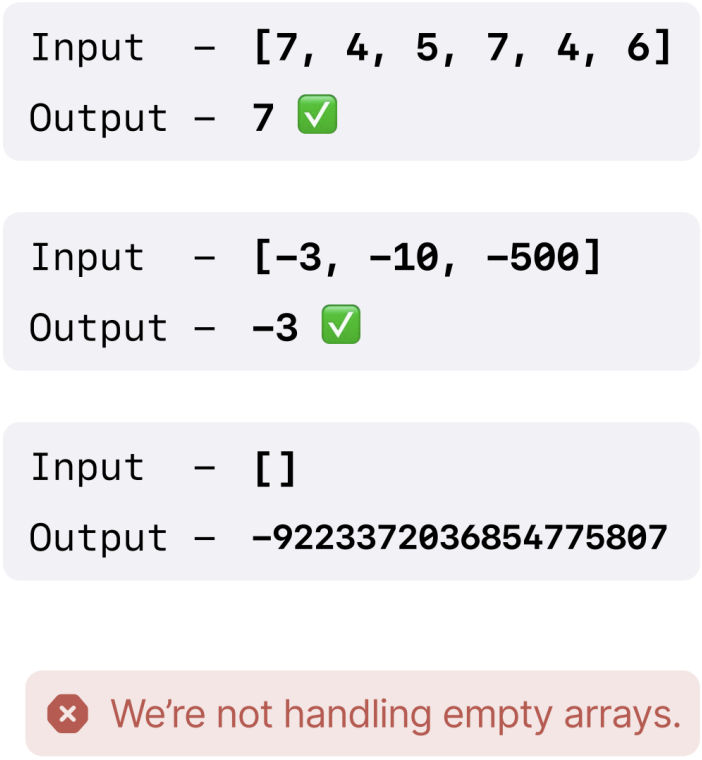


Figure 7: The outputs for this code

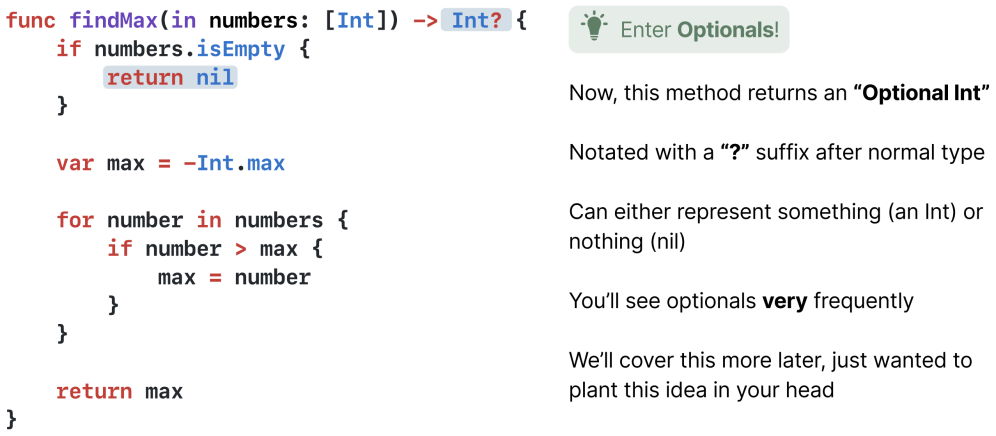


Figure 8: Enter optionals!!!

5 Building Interactive UIs 1/27

stuff about Binding and @State, missed this :(, but pretty much similar to React states and stuff

Using @State before a private variable

Using $ before a @State variable to use Binding

6 Bindings & Computed Properties 1/29

6.1 Reusable Subviews

Apps very freq. have repeated views w/ varying ContentView In SwiftUI, declare a new struct that conforms to the view

6.2 @Bindings

Modifying a parent's state

Definition

@Binding turns the child's property into a proxy to the parent's State Thus, writing to the **@Binding** updates the parent **@State** variable

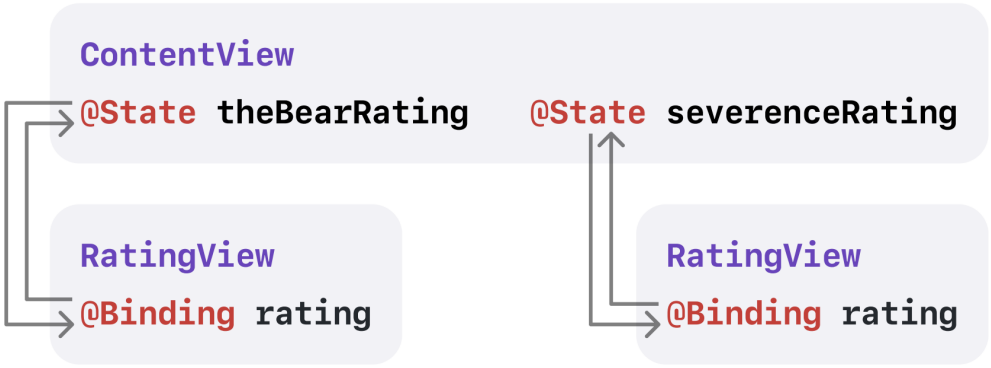


Figure 9: Here, we see an example of using @Binding, conceptually similar to a pointer in C

6.3 SwiftUI User Input

SwiftUI allows for user inputs and for example can be done as shown:

```
struct TextfieldExample: View {
    @State private var username: String = ""

    var body: some View {
```

```
            TextField("Enter username", text = $username)
        }
    }
```

6.4 Computed Properties

Consider the Rectangle struct with these attributes: width, height, area.

Instead of making area a store property we can make it a computer one as shown below:

```swift
struct Rectangle {
    var width: Double
    var height: Double

    var area: Double {
      width * height // implicit return
  }
}

var myRect = Rectangle(width: 3, height: 3)
print(square.area) // 9

myRect.width = 5
print(square.area) // 15
```

Meaning that they are always updated and consistent!

7 List & forEach

7.1 Range-based List

For range-based lists it can be defined as this:

```
List(0 ..< 100) { i in
    Text("\(i)")
}
```

This simply loops through range and request views for each element
This closure is conceptually the same as this method:

```
func generateRow(for i: Int) -> some View {
    return Text("\(i)")
}
```

7.2 KeyPaths

KeyPaths are an instruction of how to find a property.

Example

For example, **Person.age** locates the age property of Person

With a keyPath and an instance, you can read the value from the property

Moreover, we can use \.self to create a unique identifier for each row of a list.

For example,

```
let rows = ["A", "B", "C"]

List(rows, id: \.self) { row in
    Text(row)
}
```

7.3 List Unique IDs

When the array changes, SwiftUI will redraw the view. And because it has unique IDs, it can compare the before and after lists to properly animate the update...

We can also animate the following modifications:

- Insertions
- Deletions
- Relocations

7.4 Custom Data Types

Very often, we display lists of more complex data, thus we can create a custom struct to encapsulate that data...

If we want an id for the List, then we would use UUID, passing \ .id to List to refer to the UUID property

```
struct Course {
    let id = UUID()
    let code: String
    let room: String
}

struct ContentView: View {
    let courses: [Course] = [...]

    var body: some View {
        List(courses, id: \.id) { course in
            Text(course.code)
        }
    }
}
```

Figure 10: An example of custom struct using UUID

The Identifiable protocol is able to make a struct conform to indentifiable which allows not using the id property or keypaths. We can see that here:

```
struct Course: Identifiable {
    let id = UUID()
    let code: String
    let room: String
}

struct ContentView: View {
    let courses: [Course] = []

    var body: some View {
        List(courses) { course in
            Text(course.code)
        }
    }
}
```

7.5 List + Binding

Let us start combining some concepts:

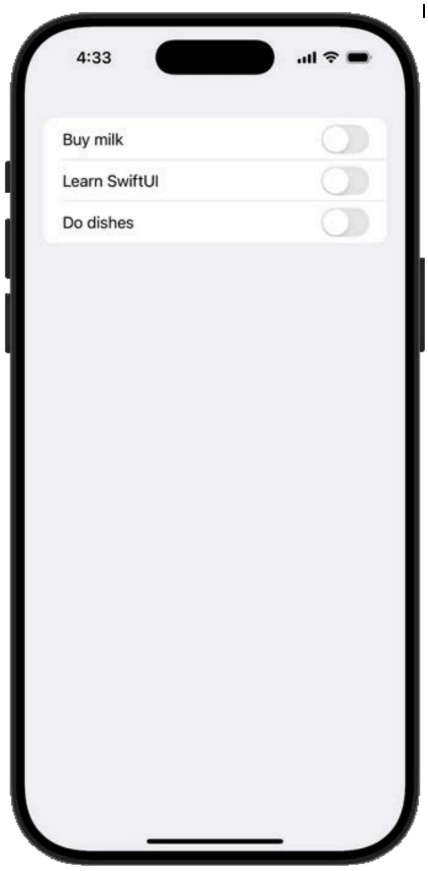
To start an interactive list, we can use binding!

Here is some example code:

```
struct SimpleTodoList: View {
    @State private var todos: [TodoItems] = [...]

    var body: some View {
        List($todos) { $todo in
            Toggle(todo.title, isOn: $todo.isCompleted)
        }
    }
}
```

The resulting output will be:



7.6 forEach

The real driver

The iterative List initializers that have been used are actually delegating to another view type – forEach