ackee
edu

BI-IOS

Igor Rosocha, Jakub Olejník, Rostislav Babáček

Lecture 4 - More SwiftUI

Human Interface Guidelines

Offer invaluable information on how to design your app's interface, navigate content, and manage interactions in your app

Reading and following these guidelines is essential

https://developer.apple.com/design/human-interface-guidelines/platforms/overview

Good resources to start with

https://developer.apple.com/ios/planning/
https://developer.apple.com/ipados/planning/
https://developer.apple.com/macos/planning/
https://developer.apple.com/tvos/planning/
https://developer.apple.com/watchos/planning/

Important shortcuts

Build: ⌘ + B

Run: ⌘ + R

Test: ⌘ + U

Stop: ⌘ + .

Clean (the build folder): ⌘ + ⇧ (+ ⌥) + K

Important shortcuts

Open Quickly: ⌘ + ⇧ + O

Show/Hide Navigator: ⌘ + 0

Show/Hide Utilities: ⌘ + ⌥ + 0

Show/Hide Debug Area: ⌘ + ⇧ + Y

Show/hide completions: ctrl + Space

Show/hide preview: ⌘ + ⌥ + Enter

`ScrollView`

Allows to create scrolling containers of views

Can be either vertical or horizontal

Automatically sizes itself to fit the content that is placed inside it

Content width affects scrollability ❗

`ForEach`

Looping over a sequence to create views

It’s a view struct, which means you can return it directly from your view body

Requires identifier to identify each of the items

`LazyVStack` / `LazyHStack`

View that doesn’t create items until it needs to render them onscreen

View will remain in memory

Automatically has a flexible preferred width ❗

`LazyVGrid` / `LazyHGrid`

Grid layouts with a fair amount of flexibility

Number of GridItems specify the number of columns/rows

Columns fit the width/height of the screen, height/width is specified by the views itself

Questions?

- ❑ `ScrollView`
- ❑ `ForEach`
- ❑ `LazyVStack` / `LazyHStack`
- ❑ `LazyVGrid` / `LazyHGrid`

Live coding! 🎉

Why structs?

All the views are trivial structs and are almost free to create - no other part holds the reference

Structs are simpler and faster than classes

SwiftUI encourages us to move to a more functional design approach

Essential protocols

Equatable

Hashable

Identifiable

Comparable

Equatable

Values conforming to the Equatable protocol can be evaluated for equality

Compiler can automatically synthesize conformance for structures with Equatable properties

Lets a value be found in a collection and matched in a switch statement

Equatable

```
struct Pet: Equatable {
    let genus: String
    let species: String
}

🐶 == 🐶 // true
🐶 == 🐈 // false
```

Equatable

❌ Not necessary!

```
extension Pet: Equatable {
    static func == (lhs: Binomen, rhs: Binomen) -> Bool {
        return lhs.genus == rhs.genus &&
                lhs.species == rhs.species
    }
}

🐶 == 🐶 // true
🐶 == 🦙 // false
```

Hashable

A type that provides an integer hash value

Any type that conforms to Hashable must also conform to Equatable

Mandatory for Set items and Dictionary keys

Conforming is often just as easy as adding Hashable to your struct conformance

Hashable

```
struct iPad: Hashable {
    var serialNumber: String
    var capacity: Int
}

func hash(into hasher: inout Hasher) {
    hasher.combine(serialNumber)
}
```

Identifiable

Values of types adopting the Identifiable protocol provide a stable identifier for the entities they represent

```
protocol Identifiable {
    associatedtype ID: Hashable
    var id: ID { get }
}
```

Identifiable

```
struct Post: Identifiable {
    let id: String
    let username: String
    let likes: Int
    let description: String
}
```

Comparable

Allows for values to be considered less than or greater than other values

You can get away with only implementing the `<` operator

Comparable

```
struct Post: Comparable {
    let username: String
    let likes: Int
    let description: String

    static func < (lhs: Post, rhs: Post) -> Bool {
        lhs.likes < rhs.likes
    }
}
```

`List`

A container that presents rows of data arranged in a single column

A scrollable list of data that user can interact with

Has some predefined styles and separators

Has native pull-to-refresh since iOS 15 🎉

Questions?

- ❑ Equatable
- ❑ Hashable
- ❑ Identifiable
- ❑ Comparable
- ❑ List

Let’s code! 🤓

Property wrappers

A type that wraps a given value in order to attach additional logic to it

Encapsulation of ”template” behavior applied to the vars they wrap

Logic is triggered every time that value is modified

Has mandatory stored property called `wrappedValue`

Optional `projectedValue` accessible via `$`

Property wrappers

```
@propertyWrapper struct Uppercased {
    var wrappedValue: String {
        didSet { wrappedValue = wrappedValue.uppercased() }
    }

    init(wrappedValue: String) {
        self.wrappedValue = wrappedValue.uppercased()
    }
}
```

Property wrappers

```
@Uppercased var serialNumber: String = "unique-serial-number"

var _serialNumber: Uppercased = Uppercased(wrappedValue: "unique-serial-number")

var serialNumber: String {
  get { _serialNumber.wrappedValue }
  set { _serialNumber.wrappedValue = newValue }
}
```

Property wrappers

```
struct Device {
    @Uppercased var serialNumber: String
    var capacity: Int
}

// UNIQUE-SERIAL-NUMBER
var iPad = Device(serialNumber: "unique-serial-number", capacity: 128)

// NEW-SERIAL-NUMBER
iPad.serialNumber = "new-serial-number"
```

Questions?

- Property wrappers

Thank you very much,
see you next week!