Stanford CS193p

Developing Applications for iOS

Spring 2020

Lecture 8

Today

- `UserDefaults`
  Lightweight persistent store
- Gestures
  Getting input from the user into your app

Persistence

- Storing Data Permanently
  - There are numerous ways to make data "persist" in iOS.
  - In the filesystem (`FileManager`). Hopefully we'll get to this later in the quarter.
  - In a SQL database (`CoreData` for OOP access or even direct SQL calls).
  - iCloud (interoperates with both of the above).
  - `CloudKit` (a database in the cloud).
  - Many third-party options as well.
  - One of the simplest to use (but only for lightweight data) is `UserDefaults`.
- `UserDefaults`
  - Think of it as sort of a "persistent dictionary".
  - Really only should be used for "user preferences" or other lightweight bits of information.
  - (You would never use it to store "documents" like we're going to do in EmojiArt — just a demo!).
  - It is limited in the data types it can store.

UserDefaults

- ## Data Types in UserDefaults

  UserDefaults is an "ancient" API.

  It far predates SwiftUI or even the Swift language.

  As a result, its API is a little strange for those used to functional programming.

  But if you "squint" at its API, you can integrate it into the Swift way of doing things.

- ## Property Lists

  UserDefaults can only store what is called a Property List.

  This is not a protocol or a struct or anything tangible or Swift-like.

  Property List simply a "concept".

  It is any combination of String, Int, Bool, floating point, Date, Data, Array or Dictionary.

  If you want to store anything else, you have to convert it into a Property List.

  A powerful way to do this is using the Codable protocol in Swift.

  Codable converts structs into Data objects (and Data is a Property List).

  We'll let the demo cover how to do that.

UserDefaults

- The `Any` type

  The API for `UserDefaults` is strange because it is pre-Swift.
  It has a lot of uses of the type `Any` (which basically means "untyped").

  Swift is a strongly-typed language.
  It does not like `Any` but supports it for backwards compatibility.
  … with things like `UserDefaults`.

  But we're going to try to ignore `Any` and still understand `UserDefaults`.

UserDefaults

- Using UserDefaults

  First you need an instance of UserDefaults. Most often we do this …

  ```
  let defaults = UserDefaults.standard
  ```

- Storing Data

  To store something …

  ```
  defaults.set(object, forKey: “SomeKey”) // object must be a Property List
  ```

  For convenience, there are also a number of functions like this …

  ```
  defaults.setDouble(37.5, forKey: “MyDouble”)
  ```

  See? Looks like a dictionary.

UserDefaults

- Retrieving Data

  To retrieve something …

```
let i: Int = defaults.integer(forKey: “MyInteger”)
let b: Data = defaults.data(forKey: “MyData”)
let u: URL = defaults.url(forKey: “MyURL”)
let strings: [String] = defaults.stringArray(forKey: “MyString”)
```

  etc.

  Retrieving `Array`s of anything but `String` is more complicated …

```
let a = array(forKey: “MyArray”)
```

  … will return an `Array<Any>`.
  At this point you would have to use the `as` operator in Swift to “type cast” the `Array` elements.
  We’ll stop there! But your reading assignment does describe how to use `as`.
  Hopefully you can use one of the above before needing `Array<Any>`.
  `Codable` might help you avoid this by using `data(forKey:)` instead.

Gestures

- Getting Input from the User

  SwiftUI has powerful primitives for recognizing "gestures" made by the user's fingers.
  This is called multitouch (since multiple fingers can be involved at the same time).

  SwiftUI pretty much takes care of "recognizing" when multitouch gestures are happening.

  All you have to do is handle the gestures.
  In other words, decide what to do when the user drags or pinches or taps.

Gestures

- Making your Views Recognize Gestures

  To cause your `View` to start recognizing a certain gesture, you use the `.gesture` `View` modifier.

  ```
  myView.gesture(theGesture) // theGesture must implement the Gesture protocol
  ```

- Creating a Gesture

  Usually `theGesture` will be created by some `func` or computed `var` you create.

  Or perhaps by a local `var` inside the `body` `var` of your `View`.

  Example …

  ```
  var theGesture: some Gesture {
      return TapGesture(count: 2)
  }
  ```

  This happens to be a "double tap" gesture (because of the `count: 2`).

  SwiftUI will recognize this `TapGesture` now, but it won't do anything about it …

Gestures

- Handling the Recognition of a Discrete Gesture

  So how do we "do something" about a recognized gesture?
  It depends on whether the gesture is discrete or not.

  `TapGesture` is a discrete gesture.
  It happens "all at once" and just does one thing when it is recognized.
  `LongPressGesture` can be treated as a discrete gesture as well.

  To "do something" when a discrete gesture is recognized, we use `.onEnded { }`.

```
var theGesture: some Gesture {
    return TapGesture(count: 2)
        .onEnded { /* do something */ }
}
```

  Discrete gestures also have "convenience versions" that you're already aware of.

```
myView.onTapGesture(count: Int) { /* do something */ }
myView.onLongPressGesture(...) { /* do something */ }
```

Gestures

- Non-Discrete Gestures

  Other gestures are not discrete.

  In those, you handle not just the fact that the gesture was recognized ...

  You also handle the gesture while it is in the process of happening (fingers are moving).

  Examples: `DragGesture`, `MagnificationGesture`, `RotationGesture`.

  `LongPressGesture` can also be treated as non-discrete (i.e. fingers down and fingers up).

Gestures

- Handling Non-Discrete Gestures

  You still get to do something when a non-discrete gesture ends.

```
var theGesture: some Gesture {
    DragGesture(...)
        .onEnded { value in /* do something */ }
}
```

  Note though that its `.onEnded` passes you a `value`.
  That `value` tells you the state of the `DragGesture` when it ended.

  What that `value` is varies from gesture to gesture.
  For a `DragGesture`, it's a `struct` with things like the start and end location of the fingers.
  For a `MagnificationGesture` it's the scale of the magnification (how far the fingers spread out).
  For a `RotationGesture` it's the `Angle` of the rotation (like the fingers were turning a dial).

Gestures

- Handling Non-Discrete Gestures

  But during a non-discrete gesture, you'll also get a chance to do something while it's happening.
  Every time something changes (fingers move), you will get a chance to update some state.

  This state is stored in a specially marked `var` …
  `@GestureState var myGestureState: MyGestureStateType =` <starting value>
  This can be a variable of any type you want.

  You can store whatever information you need for your `View` to update during the gesture.
  For example, during a drag, maybe you store how far the finger has moved.

  This `var` will always return to <starting value> when the gesture ends.

  While the gesture is happening, though, you are given an opportunity to change this state …

Gestures

- Handling Non-Discrete Gestures

  Your (only) opportunity to change your `@GestureState var` …

```
var theGesture: some Gesture {
    DragGesture(...)
        .updating($myGestureState) { value, myGestureState, transaction in
            myGestureState = /* usually something related to value */
        }
        .onEnded { value in /* do something */ }
}
```

  This `.updating` will cause the closure you pass to it to be called when the fingers move.

  Note the `$` in front of your `@GestureState var`. Don't forget this.

  The `value` argument to your closure is the same as with `.onEnded` (the state of the fingers).

  The `myGestureState` argument to your closure is essentially your `@GestureState`.
  This is the only place you can change your `@GestureState`!

  We're going to ignore the `transaction` argument (it has to do with animation).

Gestures

- Handling Non-Discrete Gestures

  There's a simpler version of `.updating` if you don't need to track any special `@GestureState`.
  It's called `.onChanged`.

```
var theGesture: some Gesture {
    DragGesture(...)
        .onChanged { value in
            /* do something with value (which is the state of the fingers) */
        }
        .onEnded { value in /* do something */ }
}
```

  This makes sense only if what you're doing is related directly to the actual finger positions.
  Maybe you're drag-to-selecting or letting the user use their finger like a pen?
  But usually you aren't interested in absolute finger position like this: you want relative position.
  (How far the fingers moved from when they first went down.)
  In that case you want to use `.updating()`.

Gestures

- Handling Non-Discrete Gestures

  In summary, when handling a non-discrete gesture while the fingers are moving ...
  Collect any information you need to draw your View during the gesture into a `@GestureState`.
  Add `.updating` to your gesture.
  In `.updating`, use the `value` that is passed to you to update your `@GestureState`.
  Understand that when the gesture ends (fingers lift), your `@GestureState` will reset.
  In other words, `@GestureState` is only meaningful while the fingers are down and moving.
  The rest of the time, your `View` has to know how to draw itself, obviously.
  So you might well have to modify some other state in `.onEnded` to make that be true.
  As usual, all best understood via demo!

Demo

- EmojiArt
  - JSON/Codable
  - UserDefaults
  - OptionalImage
  - Animatable Font Size
  - Gestures