Stanford CS193p

Developing Applications for iOS

Spring 2023

Lecture 12

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.

Persistence

- UserDefaults
  Simple. Limited (Property Lists only). Small.
- Codable/JSON
  Clean way to turn almost any data structure into an interoperable/storable format.
- UIDocument
  Integrates the Files app and "user perceived documents" into your application.
  This is really the way to do things when you have a true document like EmojiArt has.
  Is UIKit-based (no SwiftUI interface to it yet) so UIKit compatibility code is required.
  Probably not something you'll use for your final project (since this is a SwiftUI course).
- Core Data
  Powerful. Object-Oriented. Elegant SwiftUI integration.

Persistence

- Cloud Kit
  - Storing data into a database in the cloud (i.e. on the network).
  - That data thus appears on all of the user's devices.
  - Also has its own "networked `UserDefaults`-like thing".
  - And plays nicely with Core Data (so that your data in Core Data can appear on all devices).
  - We'll go over the basics of this via slides (insufficient time to do a demo though, sorry!).
  - An ambitious final project API to choose, but doable.
- `FileManager/URL/Data`
  - Storing things in the Unix file system that underlies iOS.
  - We'll demo this by storing EmojiArt documents in the file system instead of `UserDefaults`.

File System

- Your application sees iOS file system like a normal Unix filesystem
  - It starts at /.
  - There are file protections, of course, like normal Unix, so you can't see everything.
  - In fact, you can only read and write in your application's "sandbox".
- Why sandbox?
  - Security (so no one else can damage your application)
  - Privacy (so no other applications can view your application's data)
  - Cleanup (when you delete an application, everything it has ever written goes with it)
- So what's in this "sandbox"?
  - Application directory — Your executable, .jpgs, etc.; not writeable.
  - Documents directory — Permanent storage created by and always visible to the user.
  - Application Support directory — Permanent storage not seen directly by the user.
  - Caches directory — Store temporary files here (this is not backed up).
  - Other directories (see documentation) ...

File System

- Getting a path to these special sandbox directories

  `FileManager` (along with `URL`) is what you use to find out about what's in the file system.

  You can, for example, find the URL to these special system directories like this ...

```
let url: URL = FileManager.default.url(
    for directory: FileManager.SearchPathDirectory.documentDirectory, // for example
    in domainMask: .userDomainMask  // always .userDomainMask on iOS
    appropriateFor: nil, // only meaningful for "replace" file operations
    create: true // whether to create the system directory if it doesn't already exist
)
```

- Examples of `SearchPathDirectory` values

  `.documentDirectory`, `.applicationSupportDirectory`, `.cachesDirectory`, etc.

URL

- Building on top of these system paths

  URL methods:

  ```
  func appendingPathComponent(String) -> URL
  func appendingPathExtension(String) -> URL  // e.g. "jpg"
  ```

- Finding out about what's at the other end of a URL

  ```
  var isFileURL: Bool  // is this a file URL (whether file exists or not) or something else?
  func resourceValues(for keys: [URLResourceKey]) throws -> [URLResourceKey:Any]?
  ```

  Example keys: `.creationDateKey`, `.isDirectoryKey`, `.fileSizeKey`

File System

- Data

  Reading binary data from a URL …

  ```
  init(contentsOf: URL, options: Data.ReadingOptions) throws
  ```

  The options are almost always `[]`.

  Notice that this function `throws`.

  Writing binary data to a URL …

  ```
  func write(to url: URL, options: Data.WritingOptions) throws -> Bool
  ```

  The `options` can be things like `.atomic` (write to tmp file, then swap) or `.withoutOverwriting`.

  Notice that this function `throws`.

File System

- `FileManager`

  Provides utility operations.

  e.g., `fileExists(atPath: String) -> Bool`

  Can also create and enumerate directories; move, copy, delete files; etc.

  Thread safe (as long as a given instance is only ever used in one thread).

  Also has a `delegate` you can set which will have functions called on it when things happen.

  And plenty more. Check out the documentation. And check out the demo ...

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 `Arrays` 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.

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()
```

… 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`.

- `@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`.
  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: anything (but almost certainly a 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).
- `@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` (a ViewModel)).
- `@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` (`self`; i.e. the `Binding` itself)

Property Wrappers

- Where do we use `Binding`s?
  - 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.

  `Binding`s 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`.

Property Wrappers

- Where do we use Bindings?

  Sharing @State (or an @ObservedObject's vars) with other Views.

```
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)
    }
}
```

  OtherView's body is a Text whose String is always the value of myString in MyView.
  OtherView's sharedText is bound to MyView's myString.

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

  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 (except modally presented ones).

  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` (a ViewModel)).

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`
  The `wrappedValue` is: the value of some `var` in `EnvironmentValues`.
  What it does: gets/sets a value of some `var` in `EnvironmentValues`.
  Projected value (i.e. $): none.