Stanford CS193p

Developing Applications for iOS

Spring 2023

Lecture 15

Today

- App and Scene
  We have mostly ignored Emoji_ArtApp/MemorizeApp.swift so far, but it's time to take a look!
- Document Architecture
  How to really handle "document-oriented" applications like EmojiArt.
- Components of a "document-oriented" application
  DocumentGroup in your App's body
  Document Type (e.g. a .emojiart document)
  FileDocument or ReferenceFileDocument
  FileWrapper
- Undo
- Notifications
- Demo

App Architecture

- App Protocol

  You'll have only one `struct` in your application that conforms to the `App` protocol.
  You'll pretty much always mark it with `@main`.

  It has a `var body`.
  But its `var body` is not some View or some App, it is `some Scene` ...

App Architecture

- Scene Protocol

  There are three main types of Scenes that you'll use in your App's var body …

  ```
  WindowGroup { return aTopLevelView }
  DocumentGroup(newDocument:) { config in ... return aTopLevelView }
  DocumentGroup(viewing:) { config in ... return aTopLevelView }
  ```

  These are a little bit like a ForEach for Scenes.

  But we're obviously not ForEaching over a Collection here.

  Instead, each is created by "new window" (on Mac or iPad).

App Architecture

- WindowGroup

  WindowGroup is the basic, non-document-oriented Scene-building Scene …

```
struct MemorizeApp: App {
    @StateObject var game = EmojiMemoryGame()
    var body: some Scene {
        WindowGroup {
            EmojiMemoryGameView(game: game)
        }
    }
}
```

  Note that we are sharing the same ViewModel amongst all the Scenes that might be created.
  If that's not what we want, then maybe @StateObject wants to be in EmojiMemoryGameView.

Document Architecture

- DocumentGroup

  DocumentGroup is the document-oriented Scene-building Scene.

  Here's what it looks like for an editable document (versus a document we can only view) ...

```
struct EmojiArtApp: App {
    var body: some Scene {
        DocumentGroup(newDocument: { EmojiArtDocument() }) { config in
            EmojiArtDocumentView(document: config.document)
        }
    }
}
```

  Note that here we are definitely not using an `@StateObject` for our ViewModel.
  Each time a new document is created or opened, SwiftUI creates a new ViewModel for it.
  The `config` argument contains the ViewModel to use (and the `fileURL` to the document too).
  In EmojiArt, `config.document` will be something of type `EmojiArtDocument`.

  The `newDocument:` argument is the function used to create a new, "blank" ViewModel.

Document Architecture

- DocumentGroup

  DocumentGroup causes a powerful document-choosing UI to be added to your application.
  The user can open, create, rename, and organize all of their documents inside your app.

```
struct EmojiArtApp: App {
    var body: some Scene {
        DocumentGroup(newDocument: { EmojiArtDocument() }) { config in
            EmojiArtDocumentView(document: config.document)
        }
    }
}
```

  For this code to work, though, your ViewModel must conform to ReferenceFileDocument.
  And you must implement Undo in your application.

Document Architecture

- FileDocument

  This protocol gets/puts the contents of a document from/to a file.

  Creating your document from a file …

```
init(configuration: ReadConfiguration) throws {
    if let data = configuration.file.regularFileContents {
        // initialize yourself from the Data in that file
    } else {
        throw CocoaError(.fileReadCorruptFile) // should never happen
    }
}
```

  Writing your document out to a file …

```
func fileWrapper(configuration: WriteConfiguration) throws -> FileWrapper {
    FileWrapper(regularFileWithContents: /* myself as a Data */)
}
```

  These are the "regular file" versions. You can store your document as a directory of files instead.

Document Architecture

- ReferenceFileDocument

  Almost identical to FileDocument. It inherits ObservableObject. So it is ViewModel only.

  The only difference is that writing is done on a background thread via a "snapshot" ...

```
func snapshot(contentType: UTType) throws -> Snapshot {
    return /* my Model converted (possibly) to some other type, like a Data probably */
}
```

  The Snapshot type is a "don't care", but usually it's a Data.
  UTType is a Uniform Type Identifier. We'll talk about that soon.

  Writing your document out to a file ...

```
func fileWrapper(
    snapshot: Snapshot,
    configuration: WriteConfiguration
) throws -> FileWrapper {
    FileWrapper(regularFileWithContents: /* snapshot converted to a Data */)
}
```

Document Architecture

- UTType (Uniform Type Identifier)

  Obviously there needs to be a way to clearly express what type of file you can open/edit.
  This is done with Uniform Type Identifiers.

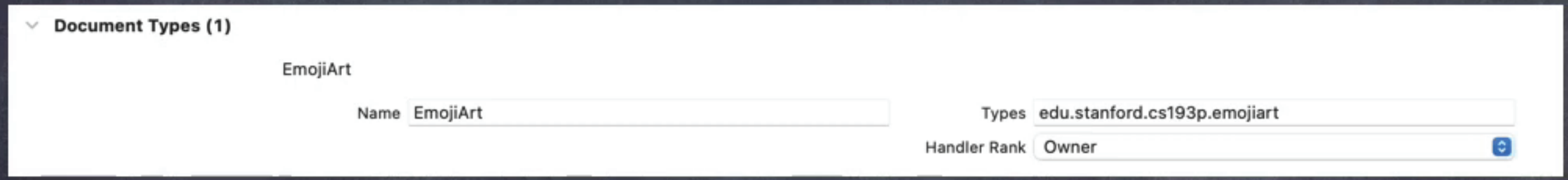
  For standard sorts of files (text files, image files, etc.), these are well known.

  If you have a custom document (like EmojiArt), you need to invent a unique one.
  We give it a unique identifier using reverse DNS notation, e.g. edu.stanford.cs193p.emojiart.

  You describe this custom identifier in your Project settings under the Info tab.
  Add an entry under Exported Type Identifiers (for your own custom types).
  Or add an entry under Imported Type Identifiers (for public/other app types).

Document Architecture

- Document Types
  - Once you've defined the UTType, you can say that you are the "Owner" of that type.
  - This is also done in the same place in Project Settings.
  - This is also where you say what the file extension for this type of document is.



Document Architecture

- Document Types
  - You'll also want to tell iOS that you're okay with people opening these with the Files app.
  - Go to the Info tab in your project.
  - Right click and choose Add Row.
  - Scroll down in the options to Supports Document Browser and set the value to YES.

Document Architecture

- Specifying the Types in your Code

  You also need to declare the UTTypes you can read and write in your code too.

  First add any custom type as a static on UTType …

```
extension UTType {
    static let emojiart = UTType(exportedAs: “edu.stanford.cs193p.emojiart”)
}
```

  This makes a new symbol UTType.emojiart to represent your custom UTType.
  If you can handle a standard type, there'll already be a static in UTType for it.

  Then specify the types you can handle in your {Reference}FileDocument object …

```
static var readableContentTypes: [UTType] { [UTType.emojiart] }
static var writeableContentTypes: [UTType] { [UTType.emojiart] }
```

  Now you can open/edit/create documents of your custom type!

Undo

- Undo

  If you use `ReferenceFileDocument`, you must implement Undo.
  This is how SwiftUI knows that you've changed the document so it can autosave it.
  It also provides a nice feature (undo) for your users!
  This is all done with something called an `UndoManager`.

  The actual "making things undoable" code is usually done by your ViewModel.
  Most often when an Intent function is called.

  Your `View`, however, is the one who has the `UndoManager` at hand.
  That's because it is part of the `View`'s environment.
  `@Environment(\.undoManager) var undoManager`
  So you will have to pass that along onto your ViewModel when you invoke an Intent function.
  This is another of those reasons we formalize Intent in our MVVM, by the way.

Undo

- Undo

  You make an operation undoable simply by registering an Undo for it.
  Here's the function in UndoManager to do that …

```
func registerUndo(withTarget: self, howToUndo: (target) -> Void)
```

  Oftentimes it is quite easy in your ViewModel to do undo if your Model is a struct …

```
func undoablyPerform(with undoManager: UndoManager?, doit: () -> Void) {
    let oldModel = model // save the model so we can undo back to it
    doit()
    undoManager?.registerUndo(withTarget: self) { myself in
        myself.model = oldModel
    }
}
```

  This snapshots the Model and does the undoable thing (i.e. the `doit` function).
  Then it registers an undo to go back to the old Model.

  Just wrap `undoablyPerform(with:) { }` around anything that changes your Model!

Undo

- Undo

  Undos can also be named so that a menu item would say "Undo Paste" for example.

```
func undoablyPerform(operation: String, with undoManager: UndoManager?, ...) {
    let oldModel = model // save the model so we can undo back to it
    doit()
    undoManager?.registerUndo(withTarget: self) { myself in
        myself.model = oldModel
    }
    undoManager?.setActionName(operation)
}
```

  You might want to undoablyPerform the line of code `myself.model = oldModel` too.

  Then you get redo as well!

Notification

- Notifications

  Allows asynchronous notification throughout your application.
  Note that this is different from `UNNotification` which is notifying the user of things.
  The system posts many of these that you can subscribe to (or you can post your own).
  Examples: database changes, keyboard coming and going, system settings changing, etc.

  You will see most of them as `statics` on `Notification.Name` (there are 100's).
  Some also live as `statics` in other places (e.g. `UserDefaults.didChangeNotification`).

  Note that many of these notifications are "the old way" to get certain kinds of info.
  Instead use `@Environment` to get a value out of `EnvironmentValues`.
  In other words, look in `EnvironmentValues` for what you want before `Notification.Name`.

  You can make your own name with `Notification.Name(String)`.

  So how do you sign up to receive these notifications?

Notification

- NotificationCenter

  In a View, use the .onReceive view modifier to receive notifications …

```
.onReceive(NotificationCenter.default.publisher(for: name) { notification in ... }
```

  Elsewhere, we need to register a closure to be called when a notification goes out.
  We do this with the addObserver function in NotificationCenter.default …

```
var observer: NSObjectProtocol?  // a cookie to later "stop listening" with
observer = NotificationCenter.default.addObserver(
    forName: Notification.Name,  // the name of the thing to get notified about
     object: nil,                // the broadcaster (or nil for "anyone")
      queue: .main               // the queue on which to dispatch the closure
) { (notification: Notification) -> Void in  // closure executed when broadcasts occur
    let info: Any? = notification.userInfo
    // info is usually a dictionary of notification-specific information
}

NotificationCenter.default.removeObserver(observer) // to stop listening
```

Notification

- Posting a Notification

  Send out your own notifications like this …

```
NotificationCenter.default.post(
        name: Notification.Name,          // name to notify under
      object: Any?,                       // who is sending this notification (usually self)
    userInfo: [any Hashable:Any]? = nil   // any info you want to pass to listeners
)
```

  Doing this will cause closures registered under your `Notification.Name` to be executed.

  Note the "any" arguments in the `userInfo`.
  This will require the receiver to use the `as` or `is` keywords in Swift to actually use.