Stanford CS193p

Developing Applications for iOS

Spring 2023

Lecture 12

Today

- Persistence
  - Storing data that persists between launchings of your application (part one)
  - `FileManager`/`URL`/`Data` (accessing the Unix file system)
  - JSON Encoding/Decoding + `Codable`
  - `UserDefaults`
- EmojiArt
  - Saving our EmojiArt document to the File System
  - `try`ing things that can `throw`
  - Using `UserDefaults` to make our Palettes persist
- Property Wrappers
  - All those `@` things.

Persistence

- Storing Data Permanently
  - There are numerous ways to make data "persist" in iOS ...
  - In the filesystem (`FileManager` and `URL` and `Data`).
  - 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.
  - `UserDefaults` (only for lightweight data like user preferences).

Archiving

- Codable Mechanism

  Essentially a way to gather all the vars of an object into a persistable blob (like a JSON Data).
  It's a great way to make an arbitrary struct be persistable (into the file system or wherever).

  For it to work, the struct you want to persist must implement the Codable protocol.
  For structs which contain only other Codables, Swift will implement the Codable for you.
  For enums too (but only if Codable associated data).
  If not all your vars are Codable (or you're a class) you can write custom code to be Codable.
  But that is beyond the scope of this introductory course.

  A lot of standard types already implement Codable …
  String, Bool, Int, Double, Float, URL
  Optional
  Array, Dictionary, Set, Data
  Date, DateComponents, DateInterval, Calendar
  CGFloat, AffineTransform, CGPoint, CGSize, CGRect, CGVector

Archiving

- Codable Mechanism

  Once your struct (et. al.) is all Codable, you can convert it to JSON (a standard format).

```
let object: MyType = … // MyType must conform to Codable
let jsonData: Data = try JSONEncoder().encode(object)
```

  Note that this encode throws. More on that in a moment.

  JSON is a text format and you can make a JSON String out of your jsonData like this …

```
let jsonString = String(data: jsonData, encoding: .utf8) // JSON is always utf8
```

  You could also just write your jsonData out to a file with a URL you got from FileManager …

```
try jsonData.write(to: url) // note that this throws
```

  To get your object graph back from the jsonData …

```
if let myObject: MyType = try? JSONDecoder().decode(MyType.self, from: jsonData) {
    // do something with myObject, which we just decoded from jsonData
}
```

Archiving

- Codable Example

  What does it look like to make something Codable?

```
struct MyType: Codable {
    var someDate: Date
    var someString: String
    var other: SomeOtherType // SomeOtherType has to be Codable too!
}
```

  If your vars are all also Codable (like the standard types all are), then you're done!

  The JSON for this might look something like …

```
{
    "someDate" : "2017-11-05T16:30:00Z",
    "someString" : "Hello",
    "other" : <whatever SomeOtherType looks like in JSON>
}
```

UserDefaults

- Using UserDefaults

  UserDefaults is a very simple, persistent, dictionary-like thing.

  To use it, you need an instance of UserDefaults. Most often we use this one …

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

- Storing Data

  To store something …

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

  A Property List is a concept (not a protocol or concrete type)

  String, Int, Double, [Property List], Dictionary<String, Property List> are all Property Lists

  And Data is a Property List, so any Codable can be JSON'ed up and put in UserDefaults

  The forKey: strings are shared by your entire app, so choose good, non-conflicting names.

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: “MyStrings”)
```

  etc.

  Retrieving a Dictionary or an Array 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! If you must, your reading assignment does describe how to use as.
  But hopefully you won’t ever need an Array<Any>.

  Codable might help you avoid this by using data(forKey:) instead.

Error Handling

- You must `try`

  Some functions throw errors at you when they fail.
  This is instead of, for example, returning some sort of error code.
  This is actually nice because it makes our code much cleaner.
  We're not forced to do ugly things like return a tuple with success and a possible error.
  We'll talk more about error throwing next week, but let's cover some basics now …

  Functions that are capable of throwing an error at you are marked with the keyword `throws`.
  e.g. `func write(to url: URL, options: ... ) throws` is a function in `Data`

  When you call a function that can `throw`, you must "`try`" it …
  e.g. `try write(to: url)`

  There are different ways to `try` depending on how you want to handle a thrown error …

Error Handling

- Choosing not to handle an error thrown at you

  There are three ways to `try` which don't handle an error thrown at you ...

  `try?` completely ignores any error that is thrown and returns `nil` if an error is thrown

```
let imageData = try? Data(contentsOf: url) // imageData = nil if error thrown
try? write(to: url) // ignores any error thrown
```

  `try!` crashes your program if an error is thrown

```
try! data.write(to: url) // if the disk was full, your application crashes
```

  `try` inside a function that is itself marked as `throws` (this rethrows any error that is thrown)

```
func foo() throws {
    try somethingThatThrows()
}
```

  In this last scenario, any code that calls `foo()` must do so with `try` (e.g. `try foo()`).

Error Handling

- Actually handling a thrown error

  Sometimes you actually want to find out what error was thrown at you, of course.

  You do this by wrapping a `do { } catch` around your code that is going to `try` something.

```
do {
    try functionThatThrows()
} catch let error {
    // handle the thrown error here
    // error can be anything that implements the Error protocol (localizedDescription)
}
```

  The "`let error`" can be left off if you want (Swift assumes it if you leave it off).
  Or you can change it to "`let foo`" in which case the `error` var inside the `catch` will be `foo`.
  You can also catch specific errors with multiple `catch` clauses on a single `do { }`.
  We'll talk about that more next week.

Demo

- EmojiArt
  - Saving our EmojiArt document to the File System
  - `try`ing things that can `throw`
  - Using `UserDefaults` to make our `Palettes` persist