Stanford CS193p

Developing Applications for iOS
Winter 2015

Today

- Camera
  - Trax Demo - Add an image to a waypoint
- Persistence
  - Archiving
  - SQLite
  - File System
  - Core Data
  - Trax Demo - Store a waypoint image added by the user in the file system
- Embed Segue
  - Putting an MVC's View as a subview of another MVC's View
  - Trax Demo - Show a "mini-map" of the waypoint when viewing its image

UIImagePickerController

- Modal view to get media from camera or photo library
  - i.e., you put it up with `presentViewController(animated:completion:)`
- Usage
  1. Create it & set its delegate (it can't do anything without its delegate)
  2. Configure it (source, kind of media, user edibility)
  3. Present it
  4. Respond to delegate methods when user is done/cancels picking the media
- What the user can do depends on the platform
  - Almost all devices have cameras, but some can record video, some can not
  - You can only offer camera or photo library on iPad (not both together at the same time)
  - As with all device-dependent API, we want to start by check what's available …

```
class func isSourceTypeAvailable(sourceType: UIImagePickerControllerSourceType) -> Bool
```

  - Source type is `.PhotoLibrary` or `.Camera` or `.SavedPhotosAlbum` (camera roll)

UIImagePickerController

- But don't forget that not every source type can give video

  So, you then want to check …

  ```
  class func availableMediaTypesForSourceType(UIImagePickerControllerSourceType) -> NSArray
  ```

  Depending on device, will return one or more of these …

  ```
  kUTTypeImage   // pretty much all sources provide this, hardly worth checking for even
  kUTTypeMovie   // audio and video together, only some sources provide this
  ```

These are declared in the MobileCoreServices framework.

```
import MobileCoreServices
```

UIImagePickerController

- But don't forget that not every source type can give video
  - So, you then want to check …
  - `class func availableMediaTypesForSourceType(UIImagePickerControllerSourceType) -> NSArray`
  - Depending on device, will return one or more of these …
  - `kUTTypeImage  // pretty much all sources provide this, hardly worth checking for even`
  - `kUTTypeMovie  // audio and video together, only some sources provide this`
- You can get even more specific about cameras
  - (Though usually this is not necessary)
  - `class func isCameraDeviceAvailable(UIImagePickerControllerCameraDevice) -> Bool`
    - `UIImagePickerControllerCameraDevice.Rear` or `.Front`
  - There are other camera-specific interrogations too, for example …
  - `class func isFlashAvailableForCameraDevice(UIImagePickerControllerCameraDevice) -> Bool`

UIImagePickerController

- Set the source and media type you want in the picker

  Example setup of a picker for capturing video (kUTTypeMovie) …

  (From here out, UIImagePickerController will be abbreviated UIIPC for space reasons.)

```
let picker = UIImagePickerController()
picker.delegate = self // self has to say it implements UINavigationControllerDelegate too
if UIIPC.isSourceTypeAvailable(.Camera) {
    picker.sourceType = .Camera
    if let availableTypes = UIIPC.availableMediaTypesForSourceType(.Camera) {
        if (availableTypes as NSArray).containsObject(kUTTypeMovie) {
            picker.mediaTypes = [kUTTypeMovie]
            // proceed to put the picker up
        }
    }
}
```

This is sort of goofy, but just roll with it.
It's a historical artifact not only of Objective-C,
but also of the way kUTTypeImage/Movie are declared.

UIImagePickerController

- Editability

  `var allowsEditing: Bool`

  If `true`, then the user will have opportunity to edit the image/video inside the picker.

  When your delegate is notified that the user is done, you'll get both raw and edited versions.

- Limiting Video Capture

```
var videoQuality: UIIPCQualityType
.TypeMedium                 // default
.TypeHigh
.Type640x480
.TypeLow
.TypeIFrame1280x720         // native on some devices
.TypeIFrame960x540          // native on some devices
var videoMaximumDuration: NSTimeInterval
```

UIImagePickerController

- Present the picker

```
presentViewController(picker, animated: true, completion: nil)
```

On iPad, if you are not offering Camera (just photo library), you must present with popover.
If you are offering the Camera on iPad, then full-screen is preferred.
Remember: on iPad, it's Camera OR Photo Library (not both at the same time).

- Delegate will be notified when user is done

```
func imagePickerController(UIIPC, didFinishPickingMediaWithInfo info: NSDictionary) {
    // extract image/movie data/metadata here from info, more on the next slide
    presentingViewController.dismissViewControllerAnimated(true, completion: nil)
}
```

- Also dismiss it when cancel happens

```
func imagePickerControllerDidCancel(UIIPC) {
    presentingViewController.dismissViewControllerAnimated(true, completion: nil)
}
```

UIImagePickerController

- What is in that `info` dictionary?

```
UIImagePickerControllerMediaType          // kUTTypeImage or kUTTypeMovie
UIImagePickerControllerOriginalImage      // UIImage
UIImagePickerControllerEditedImage        // UIImage
UIImagePickerControllerCropRect           // CGRect (in an NSValue)
UIImagePickerControllerMediaMetadata      // Dictionary info about the image
UIImagePickerControllerMediaURL           // NSURL edited video
UIImagePickerControllerReferenceURL       // NSURL original (unedited) video
```

- Saving taken images or video into the device's photo library
  - Check out `ALAssetsLibrary`.
  - Or you can use the file system (though much less likely, we'll demo this just for demo purposes).
- In general, much more sophisticated media capture is available
  - This `UIImagePickerController` API is pretty simple, but more powerful API exists.
  - Check out `AVCaptureDevice`.

UIImagePickerController

- Overlay View
  - `var cameraOverlayView: UIView`
  - Be sure to set this view's `frame` properly.
  - Camera is always full screen (on iPhone/iPod Touch anyway): `UIScreen`'s `bounds` property.
  - But if you use the built-in controls at the bottom, you might want your view to be smaller.
- Hiding the normal camera controls (at the bottom)
  - `var showsCameraControls: Bool`
  - Will leave a blank area at the bottom of the screen (camera's aspect 4:3, not same as screen's).
  - With no controls, you'll need an overlay view with a "take picture" (at least) button.
  - That button should send `takePicture()` to the picker.
  - Don't forget to `dismissModalViewController` when you are done taking pictures.
- You can zoom or translate the image while capturing
  - `var cameraViewTransform: CGAffineTransform`
  - For example, you might want to scale the image up to full screen (some of it will get clipped).

Demo

- Let user associate a photo with their added waypoint

  `UIImagePickerController`

Persistence

- Archiving
  - Very rarely used for persistence, but it is how storyboards are made persistent
- SQLite
  - Also rarely used unless you have a legacy SQL database you need to access
- File System
  - iOS has a Unix filesystem underneath it
  - You can read and write files into it with some restrictions
- Core Data
  - An object-oriented database
  - Primary way to store data in a sophisticated application
  - Hooks up rather easily to iCloud

Archiving

- There is a mechanism for making ANY object graph persistent
  - Not just graphs with `Array`, `Dictionary`, `NSDate`, etc. in them.
- For example, the view hierarchies you build in Xcode
  - Those are obviously graphs of very complicated objects.
- Requires all objects in the graph to implement `NSCoding` protocol

```
func encodeWithCoder(encoder: NSCoder)
init(coder: NSCoder)
```

- It is extremely unlikely you will use this in this course
  - Obviously we did not in the homework assignments.
  - But almost certainly not in your Final Project either.
  - There are other, simpler, (or more appropriate), persistence mechanisms.

SQLite

- SQL in a single file
  - Fast, low memory, reliable.
  - Open Source, comes bundled in iOS.
  - Not good for everything (e.g. not video or even serious sounds/images).
  - Not a server-based technology
    (not great at concurrency, but usually not a big deal on a phone).
  - Is used by Core Data (object-oriented database, more on that in a moment).

File System

- Accessing files in the Unix filesystem
  1. Get the root of a path into an NSURL
     "Documents" directory or "Caches" directory or ...
  2. Append path components to the URL
     The names of your files (and the directories they reside in)
  3. Write to/read from the files
     Usually done with `NSData` or property list components.
  4. Manage the filesystem with `NSFileManager`
     Create directories, enumerate files in directories, get file attributes, delete files, etc.

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.
- And you can only write inside your application's "sandbox"
- Why?
  - 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 bundle directory (binary, .storyboards, .jpgs, etc.). This directory is NOT writeable.
  - Documents directory. This is where you store permanent data created by the user.
  - Caches directory. Store temporary files here (this is not backed up by iTunes).
  - Other directories (check out `NSSearchPathDirectory` in the documentation).

File System

- How do you get a path to these special sandbox directories?

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

  You create an `NSFileManager` then find system directories ...

```
let fileManager = NSFileManager()
let urls: [NSURL] = fileManager.URLsForDirectory(NSSearchPathDirectory,
                                                 inDomain: NSUserDomainMask)
```

  There will only be one `NSURL` in the returned `Array` in iOS (different on Mac).

- Examples of `NSSearchPathDirectory` values

  `NSDocumentsDirectory`, `NSCachesDirectory`, `NSDocumentationDirectory`, etc.

  See documentation for more.

NSURL

- Building on top of these system paths

  NSURL methods:

  ```
  func URLByAppendingPathComponent(String) -> NSURL
  func URLByAppendingPathExtension(String) -> NSURL  // 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 resourceValuesForKeys([String], error: NSErrorPointer) -> [NSObject:AnyObject]?
  ```

  Example keys … NSURLContentAccessDateKey, NSURLIsDirectoryKey, NSURLFileSizeKey

File System

- NSData

  Reading/writing binary data to files

  ```
  init?(contentsOfURL: NSURL)
  func writeToURL(NSURL, atomically: Bool) -> Bool // atomically means "safe write"
  ```

File System

- NSFileManager
  - Provides utility operations
  - Check to see if files exist; create and enumerate directories; move, copy, delete files; etc.
  - Thread safe (as long as a given instance is only ever used in one thread)
  - Examples:

```
let manager = NSFileManager()
func createDirectoryAtURL(NSURL,
  withIntermediateDirectories: Bool,
                   attributes: [NSObject:AnyObject]?,   // permissions, etc.
                        error: NSErrorPointer) -> Bool
func isReadableFileAtPath(String) -> Bool
```

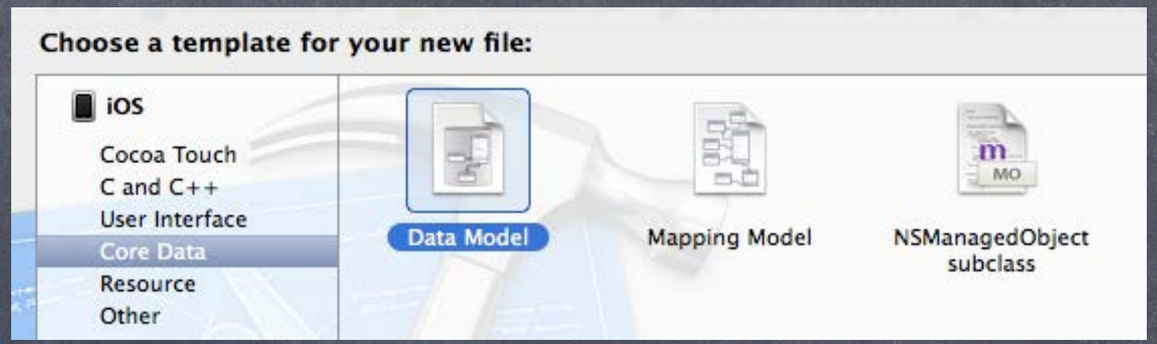
  - Also has a delegate with lots of "should" methods (to do an operation or proceed after an error)
  - And plenty more. Check out the documentation.

Core Data

- Where to store data?
  - Sometimes you need to store large amounts of data or query it in a sophisticated manner.
  - But we still want it to be object-oriented!
- Enter Core Data
  - Object-oriented database.
  - Very, very powerful framework in iOS.
- It's a way of creating an object graph backed by a database
  - Usually backed by SQL (but also can do XML or just in memory).
- How does it work?
  - Create a visual mapping (using Xcode tool) between database and objects.
  - Create and query for objects using object-oriented API.
  - Access the "columns in the database table" using `@NSManaged var`s on those objects.

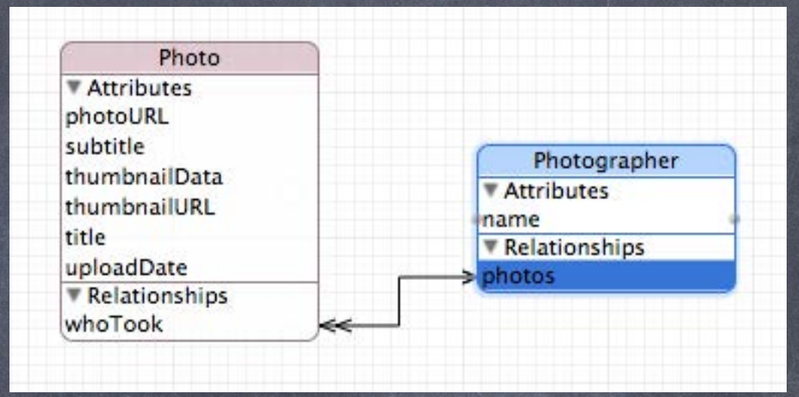
Core Data

- Visual Map
  - This is your database schema.
  - It is created with New File ...



Core Data

- Visual Map
  - This is your database schema.
  - It is created with New File ...
- Create Entities and Attributes
  - Which are similar to classes and properties



Core Data

- Visual Map
  - This is your database schema.
  - It is created with New File ...
- Create Entities and Attributes
  - Which are similar to classes and properties
- Get a `UIManagedObjectContext`
  - Can either get one by clicking the Use Core Data switch when create a new Project
    - (this will add a `managedObjectContext` var to your AppDelegate
  - Or you can use `UIManagedDocument` (which has a `managedObjectContext` var too)
- With a context, you can create and query database objects

```
func NSEntityDescription.insertNewObjectForEntityForName(String,
                        inManagedObjectContext: UIManagedObjectContext) -> NSManagedObject
let fetchedObjects = managedObjectContext.executeFetchRequest(NSFetchRequest)
```

Core Data

- Setting attributes on objects from the database

  You can set/get values with …

```
func setValue(AnObject!, forKey: String)
func valueForKey(String) -> AnyObject!
```

  … or you can create a subclass (usually with same name as Entity in database) …

```
class Photo: NSObject {
    @NSManaged var title: String
}
```

  … and simply set and get the property …

```
let photo = NSEntityDescription.insertNewObjectForEntityForName(“Photo”, inMan...)
photo.title = “My First Photo”
```

Core Data

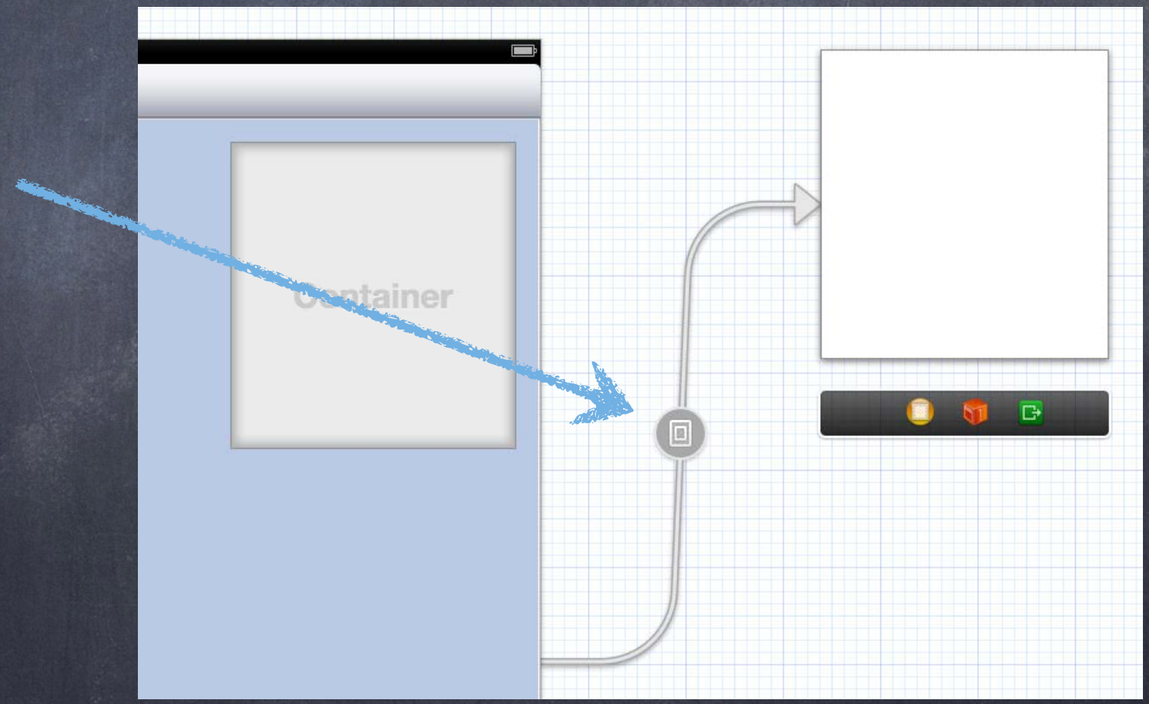
- Table View and Core Data
  - There is also a helper class for hooking up a Core Data database to a UITableView
  - NSFetchedResultsController
  - It takes an NSFetchRequest (same thing that is sent to executeFetchRequest)
    - and ensures that the table is always showing the results of that request
    - (even if the database changes out from under the table)
  - The NSFetchedResultsController can implement all of your UITableViewDatasource methods

Demo

- Store the user's waypoint photo in the file system
  - `NSFileManager`
  - `NSData's writeToURL(atomically:)`
  - `UIImageJPEGRepresentation()`

Embed Segues

- Putting a VC's `self.view` in another VC's view hierarchy!
  - This can be a very powerful encapsulation technique.
- Xcode makes this easy
  - Drag out a Container View from the object palette into the scene you want to embed it in.
  - Automatically sets up an "Embed Segue" from container VC to the contained VC.
- Embed Segue
  - Works just like other segues.
  - `prepareForSegue(sender:)`, et. al.



Embed Segues

- Putting a VC's `self.view` in another VC's view hierarchy!
  - This can be a very powerful encapsulation technique.
- Xcode makes this easy
  - Drag out a Container View from the object palette into the scene you want to embed it in.
  - Automatically sets up an "Embed Segue" from container VC to the contained VC.
- Embed Segue
  - Works just like other segues.
  - `prepareForSegue(sender:)`, et. al.
- View Loading Timing
  - Don't forget, though, that just like other segued-to VCs,
    the embedded VC's outlets are not set at the time `prepareForSegue(sender:)` is called.