ATLS 4120/5120: Mobile Application Development
Week 6: Data Persistence

Data Persistence
In most apps users change or add data, and we need that data to be persistent.
Our model objects hold data and should support data persistence so you can write objects to a file and then read them back in.
The most common ways to handle data persistence in iOS are:

- Property lists
- Object archives
- SQLite3 (iOS's embedded relational database)
- Core Data (Apple's persistence framework)

We'll be using plists today but the book covers all 4. We'll also be looking at the Realm framework and Firebase next semester.

Property Lists
A property list is a simple data file in XML that stores item types and values. They use a key to retrieve the value, just as a Dictionary does. Property lists can have Boolean, Data, Date, Number, and String node types to hold individual pieces of data, as well as Arrays or Dictionaries to store collections of nodes.
All of our apps have an Info.plist file in Supporting Files.
Property lists can be created using the Property List Editor application (*/Developer/Applications/Utilities/Property List Editor.app*) or directly in Xcode.
We are going to use a property list to store our app's data.

You can use an Array or Dictionary to hold your data and the NSArray and NSDictionary classes to write the data to a property list.

- NSArray and NSDictionary have methods that allow you to access the contents of a plist which you can then load and cast to an array or dictionary
  - write(toFile:atomically:) or write(to:atomically:) writes the data to a plist
  - init(contentsOfFile:) or init(contentsOf:) initialize a dictionary with the data in a plist
- Swift Array and Dictionary types have no way to load data from external sources.

Only certain objects can be stored in property lists and then written to a file.

- Array or NSArray
- Dictionary or NSDictionary
- NSData
- String or NSString
- (and the mutable versions of the above)
- NSNumber
- Date or NSDate

If you can build your data model from just these objects, you can use property lists to save and load your data. This is ok for simple data models. Otherwise use another method.

Sandbox
Your app sees the iOS file system like a normal UNIX file system
Every app gets its own /Documents directory which is referred to as its sandbox

Your app can only read and write from that directory for the following reasons:

- 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 its ever written goes with it)

To find your sandbox n the Finder go into your home directory and go to Library/Developer/CoreSimulator/Devices/*Device UDID*/data/Containers/Data/Application
(The Library option is hidden so if you don't see the Library folder in your home directory Go | Go to Folder | Library hold down the alt key, or hold the Option key Go | Library)
Each app has it's own folder (names are the globally unique identifiers(GUIDs) generated by xcode
Each app has subdirectories

- Documents-app sandbox to store its data
- Library-user preferences settings
- Tmp-temp files (not backed up into iTunes)

The same file structure exists on devices

NSSearchPathForDirectoriesInDomain() is a C function that will locate a directory.

- Retrieve the path to the Documents directory to read and write data files
- The FileManager class enables you to perform many generic file-system operations
  - Check to see if a file exists
  - Manipulate files (move, copy, delete, etc)

Communication
In iOS there are four common patterns for objects to communicate

1. Target-Action: a single object calls a single method when a single event occurs
   - ie buttons
2. Delegation: an object responds to numerous methods to modify or add behavior
   - text fields, table views, etc that have delegate methods
3. Notification: Register an object to be notified when an event occurs
   - Sets up how to handle when an event fires
4. Key-Value Observing (KVO): register to be one of many objects notified when single property of another object changes.
   - used for archiving

Notifications
A notification is a callback mechanism that can inform multiple objects when an event occurs.

- NotificationCenter manages the notification process
- Objects register for the notifications they're interested in
- Notification senders post notifications to a notification center
- The notification center notifies any objects registered for that notification

Data Persistence
We'll add on to our Favorites app and use a plist to make our data persistent.
ViewController.swift
Define a constant for our data file

```
let filename = "favs.plist"
```

Now we write a method that will return the path to a given file.

```
    func docFilePath(_ filename: String) -> String?{
        //locate the documents directory
        let path =
NSSearchPathForDirectoriesInDomains(FileManager.SearchPathDirectory.document
Directory, FileManager.SearchPathDomainMask.allDomainsMask, true)
        let dir = path[0] as NSString //document directory
      //creates the full path to our data file
        return dir.appendingPathComponent(filename)
    }
```

FileManager.SearchPathDirectory.documentDirectory can be shortened to .documentDirectory (value from the SearchPathDirectory enum)

FileManager.SearchPathDomainMask.allDomainsMask can be shortened to .allDomainsMask (value from the SearchPathDomainMask enum). On iOS this maps to the running app's sandbox.

On iOS, there is always only one Documents directory for each application, so we can safely assume that exactly one object will be returned and can be accessed in index 0 of the returned array.

Update viewDidLoad so we see if favs.plist exists and if it does we use that.

```
    override func viewDidLoad() {
        let filePath = docFilePath(filename) //path to data file

        //if the data file exists, use it
        if FileManager.default.fileExists(atPath: filePath!){
            let path = filePath
            //load the data of the plist file into a dictionary
            let dataDictionary = NSDictionary(contentsOfFile: path!) as!
[String:String]
            //load favorite book
            if dataDictionary.keys.contains("book") {
                user.favBook = dataDictionary["book"]
                bookLabel.text=user.favBook
            }
            //load favorite author
            if dataDictionary.keys.contains("author") {
                user.favAuthor = dataDictionary["author"]
                authorLabel.text=user.favAuthor
            }
        }

        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a
nib.
    }
```

Then our application needs to save its data before the application is terminated or sent to the background, so we'll use the UIApplicationWillResignActiveNotification notification. This notification is posted whenever an app is no longer the one with which the user is interacting. This includes when the user quits the application and (in iOS 4 and later) when the application is pushed to the background. (add after the if statement in viewDidLoad())

```
        //application instance
        let app = UIApplication.shared
        //subscribe to the UIApplicationWillResignActiveNotification 
notification
        NotificationCenter.default.addObserver(self, selector: 
#selector(UIApplicationDelegate.applicationWillResignActive(_:)), name: 
NSNotification.Name(rawValue: "UIApplicationWillResignActiveNotification"), 
object: app)
```

Now we'll create the notification method applicationWillResignActive (_:)

```
    //called when the UIApplicationWillResignActiveNotification notification 
is posted
    //all notification methods take a single NSNotification instance as 
their argument
    func applicationWillResignActive(_ notification: Notification){
        let filePath = docFilePath(filename)
        let data = NSMutableDictionary()
        //adds
        if user.favBook != nil{
            data.setValue(user.favBook, forKey: "book")
            
        }
        if user.favAuthor != nil{
            data.setValue(user.favAuthor, forKey: "author")
        }
        //write the contents of the array to our plist file
        data.write(toFile: filePath!, atomically: true)
    }
```

We need to use a NSMutableDictionary because the NS classes have the write(toFile:atomically:) methods. Mutable means it can be changed.
The atomically parameter tells the method to write the data to an auxiliary file, not to the specified location. Once it has successfully written the file, it will then copy that auxiliary file to the location specified. This is to ensure the integrity of the file in case of a crash.

To test the app fill in the text fields and tap Done to go back to the first view. Tap the home button, then exit the simulator and run it again, it should load your data. If you just exit the simulator without pressing the home button, that's the equivalent of forcibly quitting your application. In that case, you will never receive the notification that the application is terminating, and your data will not be saved. This is just a function of running in the simulator and is not needed on a device.

Note that in testing this functionality you might need to change your plist file name often, otherwise once the file is created it will always find the file in viewDidLoad() and use it. So if you want to start fresh, use a different file name for your plist.