iOS Foundations II
Day 8

- Homework Review
- Data Persistence!
- Saying goodbye forever

Data Persistence

	Core Data	NSKeyedArchiver	NSUserDefaults
Entity Modeling	Yes	No	No
Querying	Yes	No	No
Speed	Fast	Slow	Slow
Serialization Format	SQLite, XML, or NSData	NSData	Binary Plist
Migrations	Automatic	Manual	Manual
Undo Manager	Automatic	Manual	Manual

Data Persistence

	Core Data	NSKeyedArchiver	NSUserDefaults
Persists State	Yes	Yes	Yes
Pain in the Ass	Yes	No	No

NSUserDefaults

- “NSUserDefaults allows an app to customize its behavior based on user preferences”
- Think of it as an automatically persisting plist that is easily modified in code.
- Use the standardUserDefaults class method to return the shared defaults object.
- Setting values inside of it is as easy as these methods:
  - setBool:ForKey:
  - setObject:ForKey:
  - setInteger:ForKey:

NSUserDefaults

- Each app has its own database of user preferences
- Used to store and retrieve an object
- Objects must be NSCoding-compliant
- Primitives may be stored as-is (Float, Int, Bool, String, etc.)
- Try to follow Apple's recommendation of only saving small 'settings' related data in the user defaults.

NSUserDefaults Workflow

- **To save to it:**

1. Get a reference to the standard user defaults singleton
2. set a value with key on the user defaults
3. tell it to ‘synchronize’ aka save

- **To read from it:**

1. Get a reference to the standard user defaults singleton
2. Try to get a reference to a value stored inside of it with a specific key (just like a dictionary)
3. See if the reference you got back contains something, if it does, then that means there was a value there already.

NSUserDefaults examples

To save data:

```
let userInfo = ["Name" : "Brad", "token" : "jfh1234"]

NSUserDefaults.standardUserDefaults().setObject(userInfo, forKey: "userInfo")
NSUserDefaults.standardUserDefaults().synchronize()
```

To load data

```
    if let userInfo = NSUserDefaults.standardUserDefaults().objectForKey("userInfo") as?
    [String : String] {
        println(userInfo["Name"]) // prints Brad
    }
```

Demo

NSKeyedArchiver

NSKeyedArchiver

- NSKeyedArchiver/Unarchiver serializes NSCoding compliant classes to and from a data representation.
- **Classes you want to serialize with NSKeyedArchiver must conform to the NSCoding protocol.**
- Once you have done that, it is as simple as calling archive and unarchive on NSKeyedArchiver/Unarchiver to load your object graph
- The amazing part of NSKeyedArchiver is that your object graph is saved and loaded as your custom model types. You don't have to recreate all of your model objects when you load them, like we had to do with the plist.

NSCoding Protocol

- The NSCoding protocol is a very simple protocol, it only has 2 methods:
  - initWithCoder()
  - encodeWithCoder()
- Your class that conforms to NSCoding must also inherit from NSObject. See the sample code on the git repo.
- The implementation of these two methods is very much just boilerplate code, as you will see in the next slide.

NSCoding Protocol

```
//first required method is the init with coder, this is used internally by
    NSKeyedUnarchiver to load your objects from the archived data
required init(coder aDecoder: NSCoder) {
    self.firstName = aDecoder.decodeObjectForKey("firstName") as String
    self.lastName = aDecoder.decodeObjectForKey("lastName") as String
    if let decodedImage = aDecoder.decodeObjectForKey("image") as? UIImage {
        self.image   = decodedImage
    }
}

//the other required method, used to encode your objects into the archive file by
    NSKeyedArchiver
func encodeWithCoder(aCoder: NSCoder) {
    aCoder.encodeObject(self.firstName, forKey: "firstName")
    aCoder.encodeObject(self.lastName, forKey: "lastName")
    if self.image != nil {
        aCoder.encodeObject(self.image!, forKey: "image")
    }
}
```

Saving/Loading from disk

- When you use NSKeyedArchiver and NSKeyedUnarchiver, you are saving an archive file to disk. To do this, you will need a path that you want to save to.
- Each app has a separate ‘sandbox’ (or directory) for storing data. iOS keeps the apps separated form each other and the OS for security reasons.
- The sandbox contains a number of different sub directories, and the one we are allowed to write to is called the documents directory.

Getting the path

- We can use the function ‘NSSearchPathForDirectoriesInDomains() to get a path to the documents directory.
- It takes 3 parameters:

1. An enum for which directory you are looking for (we will use .DocumentsDirectory since we want the documents directory!)
2. The domain mask (will always use .UserDomainMask)
3. expandTilde Boolean (will always use true)

Example of a save

```
func saveToArchive() {
    //get path to documents directory
    let documentsPath = NSSearchPathForDirectoriesInDomains(.DocumentDirectory, .
        UserDomainMask, true)[0] as String
    //archive
    NSKeyedArchiver.archiveRootObject(self.people, toFile: documentsPath + "/
        archive")
}
```

Example of a load

```
func loadFromArchive() {

    //get path to your app's documents directory in its sandbox
    let documentsPath = NSSearchPathForDirectoriesInDomains(.DocumentDirectory, .
        UserDomainMask, true)[0] as String
    //attempt to unarchive your object graph
    if let peopleFromArchive = NSKeyedUnarchiver.unarchiveObjectWithFile
        (documentsPath + "/archive") as? [Person] {
        //stored the data we just unarchived into this proper   ty
        self.people = peopleFromArchive
        //this is great, it loaded our stuff
    }
}
```

Demo

Finale