Project 12: NSUserDefaults, NSCoding and more

Overview

`Brief`: Learn how to save user settings and data for later use.

`Learn`: NSUserDefaults, NSCoding, NSKeyedArchiver.

- Setting up
- Reading and writing basics: NSUserDefaults
- Fixing Project 10: NSCoding
- Wrap up

Setting up

This is our fourth technique project, and we're going to go back to `Project 10` and fix its glaring bug: all the names and faces you add to the app don't get saved, which makes the app rather pointless!

We're going to fix this using a new class called `NSUserDefaults` and a new protocol called `NSCoding`. We'll also be using the class `NSKeyedUnarchiver` that you just met in `Project 11`, along with its counterpart: `NSKeyedArchiver`. Putting all these together, we're going to update `Project 10` so that it saves its people array whenever anything is changed, then loads when the app runs.

We're going to be modifying `Project 10`, so if you want to preserve the old code take a copy now and call it `Project 12`.

Reading and writing basics: NSUserDefaults

You can use `NSUserDefaults` to store any basic data type for as long as the app is installed. You can write basic types such as `Bool`, `Float`, `Double`, `Int`, `String`, or `NSURL`, but you can also write more complex types such as arrays, dictionaries and `NSDate` – and even `NSData` values.

When you write data to `NSUserDefaults`, it automatically gets loaded when your app runs so that you can read it back again. This makes using it really easy, but you need to know that it's a bad idea to store lots of data in there because it will slow loading of your app. If you think your saved data would take up more than say `100KB`, `NSUserDefaults` is almost certainly the wrong choice.

Before we get into modifying `Project 12`, we're going to do a little bit of test coding first to try out what `NSUserDefaults` lets us do. You might find it useful to create a fresh `Single View Application` project just so you can test out the code.

To get started with `NSUserDefaults`, you create a new instance of the class like this:

```
let defaults = NSUserDefaults.standardUserDefaults()
```

Once that's done, it's easy to set a variety of values – you just need to give each one a unique key so you can reference it later. These values nearly always have no meaning outside of what you use them for, so just make sure the key names are memorable.

Here are some examples:

```
let defaults = NSUserDefaults.standardUserDefaults()
defaults.setInteger(25, forKey: "Age")
defaults.setBool(true, forKey: "UseTouchID")
defaults.setDouble(M_PI, forKey: "Pi")
```

In older versions of `iOS`, you needed to tell `iOS` when it was a good time to save the defaults data to disk, but this isn't needed (or even recommended!) any more.

After that, you should use the `setObject()` to set strings, arrays, dictionaries and dates. Now, here's a curiosity that's worth explaining briefly: in `Swift`, strings, arrays and dictionaries are all structs, not objects. But `NSUserDefaults` was written for `NSString` and friends – all of whom are `100%` interchangeable with `Swift` their equivalents – which is why this code works.

Using `setObject()` is just the same as using other data types:

```
defaults.setObject("Paul Hudson", forKey: "Name")
defaults.setObject(NSDate(), forKey: "LastRun")
```

Even if you're trying to save complex types such as arrays and dictionaries, `NSUserDefaults` laps it up:

```
let array = ["Hello", "World"]
defaults.setObject(array, forKey: "SavedArray")

let dict = ["Name": "Paul", "Country": "UK"]
defaults.setObject(dict, forKey: “SavedDict")
```

That's enough about writing for now; let's take a look at reading.

When you're reading values from `NSUserDefaults` you need to check the return type carefully to ensure you know what you're getting. Here's what you need to know:

- `integerForKey()` returns an integer if the key existed, or `0` if not.
- `boolForKey()` returns a boolean if the key existed, or `false` if not.
- `floatForKey()` returns a float if the key existed, or `0.0` if not.
- `doubleForKey()` returns a double if the key existed, or `0.0` if not.
- `objectForKey()` returns `AnyObject?` so you need to conditionally typecast it to your data type.

Knowing the return values are important, because if you use `boolForKey()` and get back `“false”`, does that mean the key didn't exist, or did it perhaps exist and you just set it to be false?

It's `objectForKey()` that will cause you the most bother, because you get an optional object back. You're faced with two options, one of which isn't smart so you realistically have only one option!

Your options:

- Use `as` to typecast your object to the data type it should be. This worked in `Xcode 6.2` or earlier.
- Use `as!` to force typecast your object to the data type it should be. This is available from `Xcode 6.3` or later.
- Use `as?` to optionally typecast your object to the type it should be.
- If you use `as/as!` and `objectForKey()` returned nil, you'll get a crash, so I really don't recommend it unless you're absolutely sure. But

equally, using `as?` is annoying because you then have to unwrap the optional or create a default value.

There is a solution here, and it has the catchy name of the `nil coalescing operator`, and it looks like `??`. This does two things at once: if the object on the left is optional and exists, it gets unwrapped into a non-optional value; if it does not exist, it uses the value on the right instead. This means we can use `objectForKey()` and `as?` to get an optional object, then use `??` to either unwrap the object or set a default value, all in one line.

For example, let's say we want to read the array we saved earlier with the key name `SavedArray`. Here's how to do that with the `nil coalescing operator`:

```
let array = defaults.objectForKey("SavedArray") as?
[String] ?? [String]()
```

So, if `SavedArray` exists and is a string array, it will be placed into the array constant. If it doesn't exist (or if it does exist and isn't a string array), then array gets set to be a new string array.

This technique also works for dictionaries, but obviously you need to typecast it correctly. To read the dictionary we saved earlier, we'd use this:

```
let dict = defaults.objectForKey("SavedDict") as? [String:
String] ?? [String: String]()
```

Fixing Project 10: NSCoding

You've just learned all the core basics of working with `NSUserDefaults`, but we're just getting started. You see, above and beyond integers, dates, strings, arrays and so on, you can also save any kind of data inside `NSUserDefaults` as long as you follow some rules.

What happens is simple: you use the `archivedDataWithRootObject()` method of `NSKeyedArchiver`, which turns an object graph into an `NSData` object, then write that to `NSUserDefaults` as if it were any other object. If you were wondering, “`object graph`” means “`your object, plus any objects it refers to, plus any objects those objects refer to, and so on`”.

The rules are very simple:

1. All your data types must be one of the following: boolean, integer, float, double, string, array, dictionary, NSDate, or a class that fits rule 2.
2. If your data type is a class, it must conform to the `NSCoding` protocol, which is used for archiving object graphs.
3. If your data type is an array or dictionary, all the keys and values must match `rule 1` or `rule 2`.

Many of `Apple's` own classes support `NSCoding`, including but not limited to: `UIColor`, `UIImage`, `UIView`, `UILabel`, `UIImageView`, `UITableView`, `SKSpriteNode` and many more. But your own classes do not, at least not by default. If we want to save the people array to `NSUserDefault` we'll need to conform to the `NSCoding` protocol.

The first step is to modify your `Person` class to this:

```
class Person: NSObject, NSCoding {
```

When we were working on this code in `Project 10`, there were two outstanding questions:

- Why do we need a class here when a struct will do just as well? (And in fact better, because structs come with a default initializer!)
- Why do we need to inherit from `NSObject`?

It's time for the answers to become clear. You see, working with `NSCoding` requires you to use objects, or, in the case of strings, arrays and dictionaries, structs that are interchangeable with objects. If we made the `Person` class into a struct, we couldn't use it with `NSCoding`.

The reason we inherit from `NSObject` is again because it's required to use `NSCoding` – although cunningly `Swift` won't mention that to you, your app will just crash.

Once you conform to the `NSCoding` protocol, you'll get compiler errors because the protocol requires you to implement two methods: a new initializer and `encodeWithCoder()`.

We need to write some more code to fix the problems, and although the code is very similar to what you've already seen in `NSUserDefaults`, it has two new things you need to know about.

First, you'll be using a new class called `NSCoder`. This is responsible for both `encoding (writing)` and `decoding (reading)` your data so that it can be used with `NSUserDefaults`.

Second, the new initializer must be declared with the required keyword. This means "`if anyone tries to subclass this class, they are required to implement this method`". An alternative to using required is to declare that your class can never be subclassed, known as a final class, in which case you don't need required because subclassing isn't possible. We'll be using required here.

Add these two methods to the `Person` class:

```
required init(coder aDecoder: NSCoder) {
    name = aDecoder.decodeObjectForKey("name") as! String
    image = aDecoder.decodeObjectForKey("image") as! String
}

func encodeWithCoder(aCoder: NSCoder) {
    aCoder.encodeObject(name, forKey: "name")
    aCoder.encodeObject(image, forKey: "image")
}
```

The initializer is used when loading objects of this class, and `encodeWithCoder()` is used when saving. The code is very similar to using `NSUserDefaults`, but I'm typecasting both values using as because I saved the data so I know what I'm loading.

With those changes, the `Person` class now conforms to `NSCoding`, so we can go back to `ViewController.swift` and add code to load and save the people array.

Let's start with writing, because once you understand that the reading code will make much more sense. As I said earlier, you can write `NSData` objects to `NSUserDefaults`, but we don't currently have an `NSData` object – we just have an array.

Fortunately, the `archivedDataWithRootObject()` method of `NSKeyedArchiver` turns an object graph into an `NSData` object using those `NSCoding` methods we just added to our class. Because we make changes to the array by adding people or by renaming them, let's create a single `save()` method we can use anywhere that's needed:

```
func save() {
    let savedData =
NSKeyedArchiver.archivedDataWithRootObject(people)
    let defaults = NSUserDefaults.standardUserDefaults()
    defaults.setObject(savedData, forKey: "people")
}
```

So: `line 1` is what converts our array into an `NSData` object, then `lines 2` and `3` save that data object to `NSUserDefaults`. You now just need to call that `save()` method when we change a person's name or when we import a new picture.

You need to modify our collection view's `didSelectItemAtIndexPath` method so that you call `self.save()` just after calling `self.collectionView.reloadData()`. Both times the self is required because we're inside a closure. You then need to modify the image picker's `didFinishPickingMediaWithInfo` method so that it calls save() just before the end of the method.

And that's it – we only change the data in two places, and both now have a call to `save()`.

Finally, we need to load the array back from disk when the app runs, so add this code to `viewDidLoad()`:

```
let defaults = NSUserDefaults.standardUserDefaults()

if let savedPeople = defaults.objectForKey("people") as?
NSData {
    people =
NSKeyedUnarchiver.unarchiveObjectWithData(savedPeople) as!
[Person]
}
```

This code is effectively the `save()` method in reverse: we use the `objectForKey()` method to pull out an optional `NSData`, using `if/let` and `as?` to unwrap it. We then give that to the `unarchiveObjectWithData()` method of `NSKeyedUnarchiver` to convert it back to an object graph – i.e., our array of `Person` objects.

Wrap up

You will use `NSUserDefaults` in your projects. That isn't some sort of command, just a statement of inevitability. If you want to save any user settings, or if you want to save program settings, it's just the best place for it. And I hope you'll agree it is (continuing a trend!) easy to use and flexible, particularly when your own classes conform to `NSCoding`.

One proviso you ought to be aware of: please don't consider `NSUserDefaults` to be safe, because it isn't. If you have user information that is private, you should consider writing to the keychain instead – something we'll look at in `Project 28`.