< Previous: Setting up (/read/12/1/setting-up)

Next: Fixing Project 10: NSCoding > (/read/12/3/fixing-project-10-nscoding)

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.

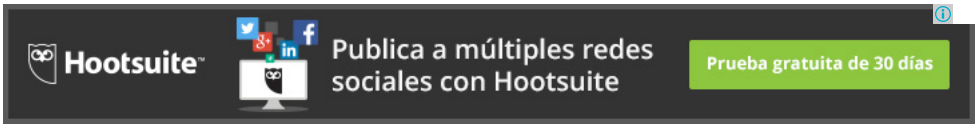
For example, let's say we want to read the array we saved earlier with the key name `SavedArray`. Here's how to do that with the `nil` coalescing operator:

```
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]()
```



< Previous: Setting up (/read/12/1/setting-up)

Next: Fixing Project 10: NSCoding > (/read/12/3/fixing-project-10-nscoding)