Extracted from:

iOS 8 SDK Development

Creating iPhone and iPad Apps with Swift

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

iOS 8 SDK Development

Creating iPhone and iPad Apps with Swift

Chris Adamson
Janie Clayton

edited by Rebecca Gulick

iOS 8 SDK Development

Creating iPhone and iPad Apps with Swift

Chris Adamson
Janie Clayton

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina

The team that produced this book includes:

Rebecca Gulick (editor)
Potomac Indexing, LLC (indexer)
Liz Welch (copyeditor)
Dave Thomas (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-941222-64-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—March 2015

Strings

The other major framework we import by default in iOS projects is *Foundation*, which provides fundamental data types for common concerns like dates and times, regular expressions, file I/O, and so on. We've already used two classes from Foundation: the NSURL and NSURLRequest that we used to populate the UIWebView.

Foundation also provides strings (as the NSString class) and collections (NSArray and NSDictionary), and in Objective-C, we would work with these as we would with any other class. However, in Swift, the language has taken more responsibility for strings and collections, and we can do a lot of common tasks without having to explicitly call methods on object instances. This makes Swift a lot easier to just get in and use.

Let's start with the String. We can create a string as we do with any other variable or constant, assigning a value with var or let. The value to assign on is enclosed in straight quotes and can include any Unicode characters.

```
let myConstantString = "iPhone"
var myVariableString = "iPad"
```

We can also build strings by using the concatenation operator, +:

```
var shoppingList = "I need to buy an " +
        myConstantString + " and an " + myVariableString
```

Strings defined with var are mutable, so we can append to them:

```
shoppingList += ", and maybe an Apple TV"
```

It's also possible to build up strings by performing in-line evaluations of expressions in the form \*(expression)*, like this:

```
var shoppingListCountString = "This list has \(1 + 1 + 1) items"
```

One rather surprising fact about Swift strings is that that they are *pass-by value*, rather than *pass-by reference*. This means that a method that takes a string as an argument gets the contents of the string, rather than a reference to a string object. It's a subtle difference but it means, among other things, that a function or method can count on the value of a string not being changed by code running at the same time and with a shared reference to the string object. In Objective-C, developers typically copied strings they received from callers just to prevent such problems. It's another way that Swift eliminates entire categories of subtle bugs.

Collections

As with strings, Swift moves support for the most common collections directly into the language, which makes them significantly easier to work with than the Objective-C approach of calling methods on objects.

Arrays

Swift provides direct language support for two essential collections: *arrays* and *dictionaries*. Arrays, as in most languages, are ordered lists of objects.

```
var musicGenres = ["Pop", "Rock", "Jazz", "Hip-hop", "Classical"]
```

Because of type inference, Swift knows this is an array of Strings. We could make that explicit by making the declaration var musicGenres: [String].

We can access array members by a zero-based index, inside square braces. We can also get a sub-array by using the *range operator*, where ..< includes the first index but not the second, and ... includes both the first and last index.

```
let pop = musicGenres[0]
let popRock = musicGenres[0..<2]
let popRockJazz = musicGenres [0...2]
```

If the array was declared with the var keyword, then it is mutable and we can add to it with the append() method, or change values in-place.

```
musicGenres.append("J-Pop")
musicGenres[1] = "Rock and Roll"
```

Arrays also have several self-descriptive methods for mutating their contents, such as insert(), removeAtIndex(), removeFirst(), and removeLast().

What do you suppose happens if we try to add something that isn't a String to musicGenres? The following produces a build error:

```
musicGenres += 2.99
```

Since the original contents of musicGenres were all strings, Swift inferred that it was an array of Strings, effectively making the declaration var musicGenres: Array<String>, where the type to the right of the colon explicitly declares what musicGenres is: an Array of Strings. Since 2.99 is a Double, it can't be added. If we really needed to do something like this, we could instead use the special type Any in our declaration, like this:

```
var musicGenres2 : Array<Any> = ["Pop", "Rock", "Jazz", "Hip-hop", "Classical"]
musicGenres2.append(2.99)
```

Any supports, well, pretty much anything: strings, numeric types, objects, and so forth. If we know we're dealing solely in objects—keeping in mind that String and collections are *not* objects in Swift—then we could use the somewhat more restrictive type AnyObject.

Swift also allows us to declare an array with the more concise syntax [Any], which is functionally identical to Array<Any>.

Dictionaries

Swift also provides *dictionaries*, which map from one object to another. These are commonly used in "lookup"-style scenarios. As with arrays, we can create a dictionary by assigning some values into it (by putting *key-value* pairs in square braces like an array, each pair separated by commas), and let Swift figure out the types.

```
var planetaryMass = [
        "Mercury"       : 3.301E+23,
        "Venus"         : 4.867E+24,
        "Earth"         : 5.972E+24,
        "Mars"          : 6.417E+23,
        "Jupiter"       : 1.899E+27,
        "Saturn"        : 5.685E+26,
        "Uranus"        : 8.682E+25,
        "Neptune"       : 1.024E+26,
]
```

In this example, Swift will infer the declaration var planetaryMass :Dictionary <String, Double> (although we gave it help by using scientific notation for the large numeric values, without which it might have inferred the value type to be Int).

As with arrays, there's a more compact way to write this declaration: [String:Double], though for now we'll use the more verbose form for clarity.

As with arrays, we use square braces to access members of the dictionary by name. The simplest use of this syntax is to add a member to the dictionary.

```
planetaryMass["Pluto"] = 1.471E+22
```

Of course, Pluto isn't a planet anymore, so this example is purely hypothetical. Anyway, we can also use square braces to look up a value by its key…or can we? Consider the following:

```
println ("Earth's mass is \(planetaryMass["Earth"]) kg")
```

This code won't even compile. But why not? The answer is a little tricky…

Optionals

To see what the problem is when we fetch a dictionary member by name, let's imagine if we added the following line:

```
var mass = planetaryMass["Gallifrey"]
```

Considering that "Gallifrey" is a fictional planet (and was, for a time, erased from history even within that fiction), it's not in our dictionary, so there's no valid answer here. So what value should be returned for a value that doesn't exist? Double is a numeric type, not an object, so we can't just have it be nil as a means of saying "no object." Can it be 0? No, 0 is a perfectly good value for a floating-point number. So what do we do here?

Swift uses a concept called *optionals* which encapsulate both knowing whether or not there *is* a value, and if so, what the value is. Dictionaries return optionals, so planetaryMass can return nil when there is no value for a key.

We make a type into an optional by adding a ? character to the type. Then we expose the optional to an if statement; if the optional has no value, this will evaluate to false. So here's a safe way to print a value from the dictionary:

```
let mass : Double? = planetaryMass["Earth"]
if mass != nil {
        println ("Earth's mass is \(mass) kg")
} else {
        println ("No such planet")
}
```

The only problem with this is that, well, optionals can be a little burdensome. In this case, the println() output is

```
Earth's mass is Optional(5.972e+24) kg
```

Ick. The println() puts that "Optional" stuff around the value. How do we get rid of it? Swift gives us an expedient way to work around cases like this: we can try to assign an optional to its non-optional base type inside an if statement, and the if evaluates to true or false based on whether the assignment works:

```
if let unwrappedMass : Double = planetaryMass["Earth"] {
        println ("Earth's mass is \(unwrappedMass) kg")
} else {
        println ("No such planet")
}
```

This prints Earth's mass is 5.972e+24 kg, without the Optional(...) stuff, because unwrappedMass is a real Double, and not a Double? (that is to say, a Double optional).

Converting an optional to its base type is called *unwrapping*. We can do this carefully, as with the if let... construction. But that can be burdensome if we have to nest a bunch of if statements, just to get at the underlying types of some optional variables or properties.

Swift also provides the as operator for casting between classes, and as? for an optional cast that may or may not succeed. So the following two if statements are equally valid ways to unwrap an optional:

```
if let unwrappedMass : Double = planetaryMass["Earth"]
if let unwrappedMass = planetaryMass["Earth"] as? Double
```

For cases where we "just know" that the value isn't nil, we can accelerate the unwrapping with the ! operator. So in our earlier example of getting Earth's mass, we can unwrap the optional within the println(), without an if test, by using the ! operator.

```
let optionalMass : Double? = planetaryMass["Earth"]
if optionalMass != nil {
        println ("Earth's mass is \(optionalMass!) kg")
} else {
        println ("No such planet")
}
```

This unwraps optionalMass within the println(), so we don't get the Optional(...) junk in our output.

Doing a fast unwrap is great, but the problem with the ! operator is that whole part about assuming the optional isn't nil. If we were to foolishly write the else case like this:

```
println ("Failed to get planet's mass: \(optionalMass!)")
```

then we would crash on the second line with `unexpectedly found nil while unwrapping an Optional value`. So much for what we "just know," huh?

We've actually seen the `!` character much earlier in this chapter, back when we dragged over a connection to create the `twitterWebView` property. When we apply the `!` to a type, it becomes an *implicitly unwrapped optional*, meaning we can unwrap its value without the `!` operator. In other words, we can just refer to `twitterWebView` and not have to write `twitterWebView!` or a bunch of `if let unwrappedWebView : UIWebView = twitterWebView` code every time we want to touch it. The unwrapping is implicit, hence the name.

Cool, right? But since it's still an optional, there is no guarantee that it even *has* a value. So it's inherently unsafe and should only be used when we really know the variable or property will always have a value when we reference it. Since Xcode is responsible for connecting our storyboard elements to our code, it can confidently use the implicitly unwrapped optional instead of the more burdensome optional type.

In fact, we'll see this a lot in the iOS code we're going to call. On http://devforums.apple.com, Apple's engineers have explained that implicitly unwrapped optionals are needed when bridging to old Objective-C code, where `nil` is always a possible value for Objective-C types, and they couldn't immediately audit every method in all the frameworks to guarantee that a given parameter can *never* be `nil`. Yet, on the other hand, if every parameter were a full-blown optional, we'd be writing lots of defensive "if not `nil`" code. So, for now, it's a trade-off between safety and usability, and most of Apple's methods currently work with implicitly unwrapped optionals. That said, Apple engineers are indeed auditing the iOS frameworks, and some implicitly unwrapped optional parameters and return types are changing to either plain types or full-blown optionals with each new release of Xcode. We can plan on this being an ongoing process for a while.

Failable Initializers

Xcode 6.1 and iOS 8.1 introduce a new use of optionals, the *failable initializer*. These are used for cases where we call an initializer and get *nothing* back.

We saw this earlier when we created an `NSURL` from a string. If the string is garbage, the `NSURL` class now reserves the right to give us back `nil`, rather than a useless object. With a failable initializer, the initializer returns `NSURL?`, which is an optional type. This explains why we had to put a `!` on the returned `url` object when we passed it to the `NSURLRequest` initializer.