Reading Assignment II: More Swift

Objective

The goal of our first week of reading assignments was to start to get a handle on this new language you must learn called Swift. Now we'll move on to learn yet more about it.

Most of you have not had experience with Objective-C, but don't worry about that. Nothing in the Swift documentation really assumes that. However, if you have never programmed in C (or C++ or any other variant), then Swift might be extremely new to you (but hopefully still not too steep a hill to climb to learn).

Materials

- All of the reading comes from this Swift Programming Language document.

Sections to Read

To better utilize your valuable time and to emphasize important concepts, the sections in the reading have been broken down into four categories:

Red sections are VERY IMPORTANT and might be more difficult to understand. Read these carefully.

Yellow sections are important, but won't be as difficult to understand.

Green sections are important, but cover very basic, simple stuff (much of it just like C).

Grayed-out sections are not required reading (this week). They may be in future weeks.

Don't gloss over reading any NOTE text (inside gray boxes)—many of those things are quite important.

If there is a link to another section in the text, you don't have to follow that link unless what it links to is also part of this reading assignment.

In the **Language Guide** area, read the following sections in the following chapters:

The Basics

Numeric Literals
Numeric Type Conversion
Type Aliases
Tuples

Basic Operators

Nil Coalescing Operator

Strings and Characters

In Initializing an Empty String, do not ignore the "initializer syntax" reference now
In Strings Are Value Types, the NSString reference in NOTE will make sense this week
Unicode
Unicode Representations of Strings

There's not anything in particular about Unicode that you'll have to know for assignments in this class, but it's a good thing to know in general.

Collection Types

In Arrays, the discussion of initializers should now make sense to you
Dictionaries

Control Flow

In Conditional Statements, read Tuples, Value Bindings & Where (skipped last week)

Functions

In Function Parameters and Return Values, read Functions with Multiple Return Values
In Function Parameters and Return Values, read Optional Tuple Return Types
Function Parameter Names

Closures

We will revisit the following two topics in a few weeks when it will really be important, but go ahead and read them now just to finish off the first 7 chapters of the reference document. They are not difficult concepts, per se, but until you start using this idea, it may seem a bit abstract. You won't need to capture values in your closures in the first 3 assignments of this quarter, so don't sweat it too much.

Capturing Values
Closures are Reference Types

Enumerations

Read this entire chapter.

Since enumerations are quite a bit more powerful in Swift than in many other languages, I will be doing an extensive demonstration of enumerations in lecture 3, but will not be covering raw values (whose use is not really strongly encouraged anyway—there are some circumstances where it is "just the right thing to use", but lots of other circumstances where people are just reverting to their "old concept" of enumerations).

Classes and Structures

Classes and Structures are very similar in Swift (initializers, functions, properties, etc.) but there are important differences (value vs. reference type, inheritance, etc.) and this chapter outlines them. Read carefully.

Structures and Enumerations are Value Types
Classes are Reference Types
Choosing Between Classes and Structures
Assignment and Copy Behavior for Strings, Arrays, and Dictionaries

Properties

In Stored Properties, Lazy Stored Properties (skipped last week)
Property Observers
Global and Local Variables
Type Properties

Methods

Instance Methods (especially anything about Parameter Names)
Type Methods

Subscripts

Subscript Syntax (this is an optional read—pretty cool though)
Subscript Usage
Subscript Options (again, optional, but cool)

Inheritance

Defining a Base Class
Subclassing
Overriding
Preventing Overrides

Initialization

It is tough to really understand the entirety of this topic without reading this whole chapter, but it's a lot to read when combined with all of the above reading, so I'm focusing on the basic stuff. By the way, `UIViewController`'s initializer situation is pretty complicated, so I've added a detailed addendum about that below.

Setting Initial Values for Stored Properties
Customizing Initialization
Default Initializers
Initializer Delegation for Value Types
Class Inheritance and Initialization
Failable Initializers
Required Initializers
Setting a Default Property Value with a Closure or Function

(This last one is very cool, but pay close attention to the NOTE box in the middle.)

NOTE: You should not need a `UIViewController` initializer for assignment 1 or assignment 2 (and hopefully not for any assignment all quarter long!). So unless you disagree with that, you can skip the next page!

But … in the interests of full disclosure … here is the bare minimum you need to know if you for some reason feel you absolutely must have an initializer in your `UIViewController` subclass (again, hopefully never).

`UIViewController` has two initializers and both (or **neither**) should be implemented in a subclass …

```
override init(nibName nibNameOrNil: String?, bundle nibBundleOrNil: NSBundle?) {
    super.init(nibName: nibNameOrNil, bundle: nibBundleOrNil)
    <your initialization code here>
}
```

and

```
required init(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)
    <your initialization code here>
}
```

Don't forget the `override` and `required` keywords. Obviously you'd likely want to factor your initialization code into some other method you can call from both of these.

But if you can avoid implementing these (which you almost always can), please do. It's an annoying historical artifact (IMHO). Most `UIViewController` initialization occurs in the following View Controller Lifecycle method (which we will talk about in lecture):

```
override func viewDidLoad() {
    super.viewDidLoad()
    <your initialization code here>
}
```

When this is called, all of your outlets have been connected, but your Controller's View is not on-screen yet, so this is a great place to do all your one-time initialization. I would recommend you always design your `UIViewController` subclass so that initialization can occur here instead of futzing with the (somewhat arcane) initializers of `UIViewController`. Don't forget the strategy of making a property be an *implicitly unwrapped optional* if you have to (and initialize it in `viewDidLoad`). This is how `UIViewController` handles outlets (although it initializes those *just before* `viewDidLoad` is called, not in `viewDidLoad` itself).

Optional Chaining

Reading this entire chapter is (no pun intended) optional. It's a very cool way to make things very concise (and very readable too), but if the whole concept of Optionals has not fully sunk in for you yet, you might find this chapter a little bit too much right now.

Type Casting

While Swift itself is extremely type safe, iOS has grown up with a history which is not quite so strict. Thus, there are numerous occasions while working with iOS APIs where you have a pointer to an object and you are going to want to know what class it is and/or you will want to cast it to be a certain class (if possible). Swift provides ways to do this safely and this chapter discusses that.

Entire Chapter (although I don't think we will use the type Any this quarter)

Nested Types

Entire Chapter

Generics

For now, all you need to know about Generics is how to use them. It's very straightforward. For example, `Array<Double>` (array of `Double`) or `Dictionary<String, Int>` (dictionary whose keys are `String` and whose values are `Int`). You can read this chapter if you want to understand more about it, but it's a pretty powerful mechanism and you have a lot on your plate right now, so this chapter is optional at this point.

Access Control

Again, you have a lot to read, so I'm not going to make you read this chapter right now, but the executive summary is that we are going to start putting the keyword `private` in front of all of the API we write unless we truly intend for (and are prepared to support) other code in our application to call the method or property in question. When you are learning to develop, always imagine you are working in a team of programmers who might want to be using the code you are writing. Access control lets you properly express the "level of support" a given method or property is going to receive in the future.

Advanced Operators

Overflow Operators

It's possible you might want to use these if you do the error-reporting extra credit in assignment 2 and decide to report arithmetic overflow errors too.