Pace University - PCBT

Advanced iOS Class Spring 2015

Optionals

- This lecture will provide a background and context for using optionals within your Swift iOS Apps.
- We will discuss the best practices in doing so.
- The lecture should be accompanied by an XCode exercise creating a project implementing the topics.

What does this mean?

- Value of optional type 'Int?' not unwrapped; did you mean to use '!' or '?'?

You need nil so you need optionals

- There are places where things just don't exist yet, and we need a way to show it. That is what optionals are for. Optionals change any type into a type with a flag for the nil state and a value. If the optional is nil, there is no value, if the optional is non-nil there is a value.
- We declare an optional like this:
- var pizza:Pizza?= Pizza()
- var numberOfPizzas:Int? = nil
- var numberOfPizzas:Int?

Unwrap before use

- Assigning to an optional is just like assigning to anything else.
- numberOfPizzas = 5
- However getting data is more difficult…
- Optional values have two parts, one for nil and one for the value. If we retrieve a value from numberOfPizzas like this:
- var myNumberOfPizzas:Int = numberOfPizzas

You get this…

- Value of optional type 'Int?' not unwrapped; did you mean to use '!' or '?'?
- Here we need to use the force unwrap operator ! to get an integer value out of the optional
- var myNumberOfPizzas:Int = numberOfPizzas!

Always check for nil

- If we run now while myNumberOfPizzas is nil, we get a run time error
- fatal error: Can't unwrap Optional.None
- We cannot unwrap an optional that is nil. In order to prevent this, first check an optional for nil, then get its value using the force unwrap operator !

```
var myNumberOfPizzas:Int =  0

if numberOfPizzas{

   myNumberOfPizzas = numberOfPizzas!

}
```

Implicit optionals are still optional

For ease of use, Swift allows you to declare a implicitly unwrapped optional. For example:

```
var numberOfPizzas:Int! = 1
```

Makes a implicitly unwrapped optional with a value of 1. Implicitly unwrapped optionals aren't force unwrapped. Use them the same way as non-optionals, and don't force unwrap when accessed.

Optionals are not optional for API's

Objective C classes, as pointers, expect a nil value. As almost everything in the API's and frameworks is an Objective-C class, almost everything you will work with is an optional. The compiler will remind you of this very often when you try to assign something to a non-optional value that is an optional value. As you check help screens and documentation you will see definitions of API like this:

```
func prepareForSegue(_ segue: UIStoryboardSegue!,sender
sender: AnyObject!)
```

Optionals are not optional for API's

The definition tells you segue is an implicitly unwrapped UIStoryboardSegue and sender is a implicitly unwrapped AnyObject. For the most part, optionals in the API are implicitly unwrapped, so rule 5 above applies often when working with frameworks and API's. Read these definitions carefully. For example, while in Swift alone you can switch around NSString and String rather easily, many places that an NSString is a property in Objective-C requires String! in Swift. For example in UILabel, the text property's definition is:

```
func prepareForSegue(_ segue: UIStoryboardSegue!,sender
sender: AnyObject!)
```

Outlets are weak, weak is optional

If using outlets, anything that is an outlet is weak. Prior to Xcode 6 Beta 4 If you have in your code

@IBOutlet var myButton:UIButton

@IBOutlet var myButton:UIButton!

Reference Cycles

All outlets are implicitly weak optionals. Weak references are those that can be destroyed by ARC to prevent reference cycles. Weak references need to have a state of nil for when they do not have a reference so ARC can dispose of them. Thus weak references must be optionals.

Why would you create a "Implicitly Unwrapped Optionals"

- vs creating just a regular variable or constant.
- let someString:String! = "this is the string";
- let someString:String = "this is the string";
- Consider the case of an object that may have nil properties while it's being constructed and configured, but is immutable and non-nil afterwards (NSImage is often treated this way, though in its case it's still useful to mutate sometimes). Implicitly unwrapped optionals would clean up its code a good deal, with relatively low loss of safety (as long as the one guarantee held, it would be safe).

Links

- http://www.drewag.me/posts/what-is-an-optional-in-swift
- http://www.drewag.me/posts/uses-for-implicitly-unwrapped-optionals-in-swift

End of Lecture

rpascazio@pace.edu