Follow Up Notes - **Lectures 4 and 5**

Howdy

These follow-ups are intended to address some of the questions people had in class in a bit more detail. Right now you're forming the foundation of all your future work on iPhone dev, and it's crucial to understand these ideas in full so you can write solid code down the line.

Here's a list of topics students had questions about during lecture this week:

1. Properties
2. Memory management
3. Proper MVC
4. IB & XCode interaction
5. UI creation in IB vs in code

Question 1

How do properties work again?

A: In summary, a few high points:

• (assign) vs (copy) vs (retain)

People have had questions about when to use (assign) vs (copy) vs (retain). The getters generated by @synthesize for these properties don't do anything other than return the value. The setters generated for these properties with @synthesize work like so:

- (assign) = simply assigns the value and doesn't modify the object's ref count
- (copy) = creates a new object that's been retained (ref count of 1), releases old object
- (retain) = release the old object, assigns old object to input object, retains input object (slide 35 of lecture 3)

There are a few rules of thumb you can use when deciding which attribute to use. Of course, these are exceptions, so be sure to think carefully about which attribute to apply:

- Use (assign) for primitive C types (int, float, double, char, etc)
- Use (copy) for NSStrings
- Use (retain) for NSObject and subclasses

• (nonatomic)

When browsing sample code in the iphone docs, you're bound to see the (nonatomic) attribute used prolifically. By default, the getters and setters generated by @synthesize are thread-safe, meaning only one thread is allowed to execute your method at a time.

The advantage here is that you avoid race-conditions but at the cost of speed (since there's a lock and unlock every time a getter or setter is called). If you aren't using threads, you can get a performance boost if you just use the (nonatomic) attribute which doesn't apply the thread-safe mechanism to your getters and setters when @synthesize is called.

• For more on properties, check out the official docs:
http://developer.apple.com/DOCUMENTATION/Cocoa/Conceptual/ObjectiveC/Articles/ocProperties.html

Question 2

My object has a reference count of 2, but it should only be 1. What's going on?

A: There are a few rules you can take to the bank that'll help you with you memory management:

• An NSArray retains any object added to it. When the NSArray is dealloc'ed, all the elements in the array are released. Same goes for any other collection in Obj-C (like NSDictionary).

• Alloc/init creates an object and sets the ref count to 1, so you **don't** need to retain after you alloc/init.

• Methods like [NSArray array] and [NSDate date] create an object for you with a ref count of 1, but they also **autorelease** it as well. If you need it as a temporary local variable, you don't need to do anything else. If you're assigning the result of [NSArray array] to an instance variable, though, you **must** retain it.

• Assume that if any method you call returns an object, that object has had autorelease called on it

• Writing dealloc for your classes is easy: just find all the object instance variables and call release on them

For more detailed memory mgmt info, see:
http://developer.apple.com/iphone/library/documentation/Cocoa/Conceptual/MemoryMgmt/MemoryMgmt.html

Question 3

My view can't have a reference to a model class b/c that would be bad MVC, right?

A: Not quite. It's actually perfectly fine for a view class to contain a reference to model class. It is **not**, however, ok for a model to a have a reference to a view class. The breakdown works like this:

• Models don't have access to controllers or views.

• Controllers have read/write access to both models and views. In fact, controllers are specifically in charge of updating model and view state.

• Views have read/write access to controllers but they can also have read-only access to models. The view is in charge of presenting the application state, and since the state is stored in the model, it's completely valid MVC for the view to read state information from the model object directly. However, the view should **not** be updating the model state; that's the responsibility of the controller.

Question 4

How exactly do IB and XCode work together again?

A: There's been a bit of confusion over how the pieces fit together between IB and XCode, and understandably so. It took me quite a while to understand the relationship. Here are some tips for figuring out how the two play nice together:

• Everything you do design in IB, all the views, labels, spinners, sliders, colors, fonts, **everything** is written to an XML file. In sharp contrast to Visual Studio, there are 0 lines of code generated when you design your UI using IB. Instead, all the details you specified in IB are bundled up into XML, and outputted as a .xib file. To reiterate, no source code or bytecode has been generated at this point.

• After you've finishing designing your hybrid flashlight fart app in IB (a sure hit), then you switch back to XCode and hit Build and Go. The UI shows up perfectly, but how? The .xib file was loaded at runtime, and due to the extremely dynamic and introspective nature of Obj-C, all the objects and classes you specified in the xib are generated by the xib loader. In the HelloPoly assignments, this is exactly how the Controller object is being instantiated.

• Note that the xib loader is taking care of instantiating the Controller object, which means the xib loader will release it. Since you didn't create it, you don't have to release it. However, by overwriting the dealloc method of Controller, you can still properly release Controller instance variables you created or retained during the Controller lifetime. Controller's dealloc method will be called when the xib loader releases it.

Question 5

Why use IB when I can just create the UI programatically?

A: Here's the thing: once you learn how to use IB, you can churn out an interface in no time flat. As such, IB is a fantastic prototyping tool if you need speed. It's also great if you're experimenting and want to see what different layouts look like. Pretty much, it works, and it's fast.

However, IB has its limitations. If you need to add or remove UI elements on the fly, programmatically creating your UI is the way to go. In fact, for me at least, that is the single biggest reason to go hack it in code. My rule of thumb is if the interface is static and unchanging, use IB. If it's dynamically adding and removing UI elements, do it by hand in code.

That's all folks

That was extraordinarily long, and you can count on future follow-ups being much shorter and succinct. I just think it's important at this early stage to get the fundamentals down pat before moving on to the more advanced stuff.