Follow Up Notes - **Lectures 2 and 3**

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. Why if (self = [super init]))?
2. id vs NSObject
3. Calling dealloc instead of release on an object
4. NSString memory management

Question 1

Why on earth do I have to check if (self = [super init]) in my init method?

A: Note the single '='. This line is simply checking that [super init] doesn't return nil. It's possible, although unlikely, for [super init] to return nil in the case the super class runs into some error during the init process.

There's also the possibility [super init] returns some object with a type **different** than the super class. Honestly, you'll almost never run across this case, but if you want a detailed and gory discussion about it, check out this wiki page: http://cocoadev.com/index.pl?FactoryMethod

Question 2

What's the difference between id and NSObject again?

A: There are a few notable differences:

1. id has no compile time type-checking. You can send it any message you want and it'll compile happily. NSObject, on the other hand, is type-checked and causes compiler warnings if you send it nonexistent messages.

2. id is **not** necessarily an NSObject. If you're coming from a Java background, you may be inclined to think that all objects in Obj-C inherit from NSObject. This is not the case. id can resolve to any Obj-C object, which might be an NSObject or it could be some other object that doesn't derive from NSObject.

3. An id declaration doesn't have a * in its declaration. NSObjects do. This is easy to miss. (id date; vs NSDate *date;)

For a more details on id vs NSObject, check out this nice explanation:
http://unixjunkie.blogspot.com/2008/03/id-vs-nsobject-vs-id.html

Question 3

What happens if I call dealloc on my object instead of release?

A: Think plague-of-locusts in your bytecode.

Imagine you have two classes A and B that retain some shared object C. A gets the bright idea to dealloc the object instead of releasing it. This invokes the dealloc method on C, which will release all the object's instance variables, many of which will now be freed themselves and turned into junk memory. So now if B tries to modify one of the instance variables in C, all hell will break loose. As if this wasn't bad enough, there's a another problem: the reference count on the shared object is still 2 after calling dealloc! Only a release call can decrement the reference count and ONLY when the reference count reaches 0 is the object memory actually freed.

One important thing to note here is that dealloc != free in C. Just calling dealloc by itself doesn't actually free the object in memory. Reference counting is king in Obj-C.

Question 4

How do I memory manage NSStrings?

A: First, I'll preface this with some of the general memory management rules we went over today:

1. if you alloc/init it, you release it
2. if you retain or copy it, you release it
3. if someone else gives you an object reference, you need to retain/release it yourself if you want to keep it around as an instance variable

Rule 3 also goes for factory methods or any other method that returns a freshly minted new object to you. In general, these methods are going to give you a reference to an autoreleased object, so if you assign it but don't retain it, it's going to be junk memory as soon as the current event loops ends (and the autorelease pool is emptied).

Now, what does this have to do NSString? Assume you have an instance variable of type NSString called valjean. There are two ways to initialize the string:

1. valjean = @"prisoner 24601";
2. valjean = [NSString stringWithFormat:@"prisoner %d", 24601];

The million dollar question: do you need to retain/release the string that's initialized using method 1? How about method 2? Remember valjean is an instance variable, so you want to keep it around for the lifetime of the class instance.

For statement 1, the correct answer is no, you don't need to retain/release the string. Statically allocated string are special, and retain/releases actually map to no-ops on the string, so you can call retain/release a million times on the string and nothing will happen.

Statement 2 is a different story. The answer is actually yes, you must retain/release the string. Note that this falls under the 3rd rule of memory management we mentioned above, that is, the NSString class method stringWithFormat formatted the string and returned a fresh new object to you, one that **you** are now responsible for retain/releasing if you want to keep it around as an instance variable.

The takeaway here is to be extra-careful and mindful when doing your memory management for NSString instance variables.

That's all folks

Alright, that's all for now. Of course, you won't really internalize this info until you bang your head against some memory bugs for a while, but hopefully this gives you a little forewarning on things to look out for.