Follow Up Notes - **Lectures 6 and 7**

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. Memory management and IB
2. MVC revisted
3. Best places to save and load

Question 1

I'm confused about memory management and IB. What I should be initializing? What is IB initializing? What should I retain and release?

A: Throughout the week and in grading assignment 2, there have been several questions about when you need to alloc/init objects yourself and when IB does it for you. The interplay between IB and the code you write is non-obvious, but you'll be fine if you follow these two simple rules:

1. If an instance variable is declared as an IBOutlet in your class, you do **not** need to instantiate it. The xib loader will alloc/init the instance variable for you.

2. If an instance variable is declared as an IBOutlet in your class, you **do** need to release it. The xib loader won't do it for you, so it needs to be released along with your other instance variables in your dealloc override.

I know there's a strong urge to alloc/init all your instance variables, but if they're prefixed with IBOutlet, you must resist! Bugs resulting from not following these rules manifest in non-obvious ways. The first problem several people have run into is, "I call [shapeVeiw setNeeds-Display] but nothing happens". In this case, you have a shapeView reference in your controller:

```
@interface Controller : NSObject {
    IBOutlet PolygonView *shapeView;
    ...
}
```

which you then initialize in the awakeFromNib method:

```
- (void)awakeFromNib {
    shapeView = [[PolygonView alloc] init];     // incorrect!
    ...
}
```

The problem here is that IB has already alloc/init'ed the shapeView, and (here' s the crucial part) added it as a subview of the window. So when you alloc/init a new instance, not only are you leaking memory (since you didn't release the shapeView created by the xib loader), but any changes you make to the shape aren't rendered because you don't have a reference to the shapeView that's actually being displayed on screen.

These bugs are subtle and hard to catch, so be sure to follow the two memory management rules above, and you'll be in the clear.

Question 2

Lots of MVC-related questions over the course of the week:

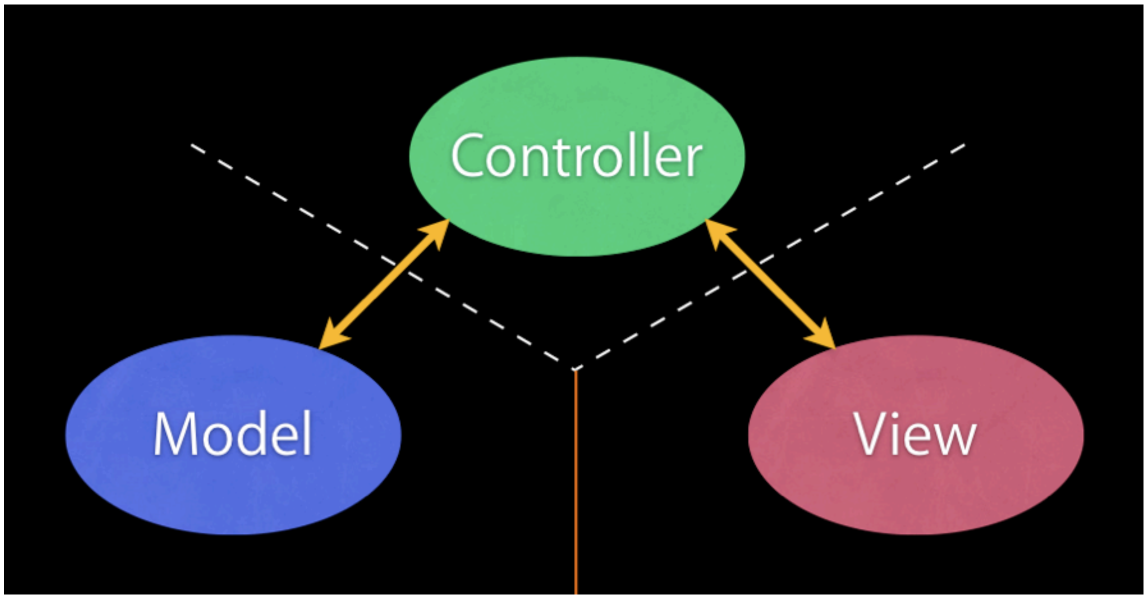
How should I give the view access to the PolygonShape's numberOfSides?

Give the view a reference to the PolygonShape object, and either hook it up as an IBOutlet that points to the same PolygonShape as the Controller or have the Controller pass the view a reference to the model object on application startup.

Do not, and I repeat: **DO NOT** add a numbeOfSides integer to the view class. This is bad, bad, bad and blatantly breaks the MVC paradigm. You may be thinking, why make such a fuss over adding a little variable? The problem lies in the fact that you're duplicating state and must manually remember to sync the two versions of numberOfSides (in the model and view classes). Almost certainly you'll forget to update both somewhere and end up with some very nasty and difficult bugs.

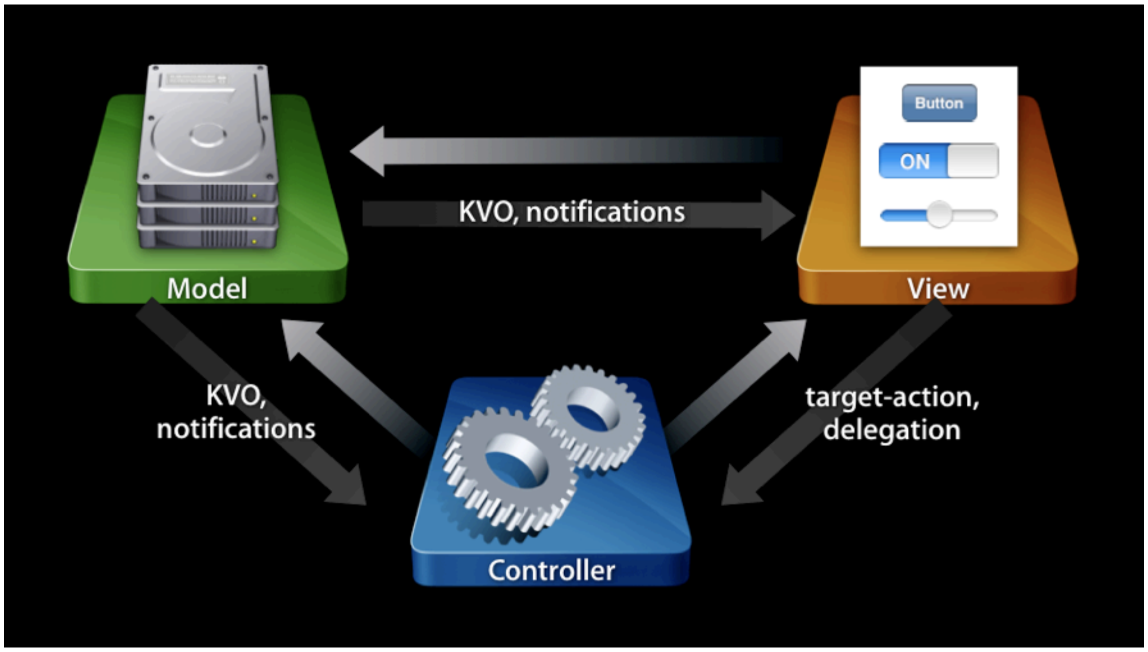
I thought a View couldn't have a reference to a Model class?

You're probably thinking of this picture from the first lecture on MVC (lecture 4):



From this picture, you could understandably assume that the model and view must communicate through the controller. It turns out that's not actually the case. (We were trying to keep things nice and simple for those who haven't been exposed to MVC before).

Instead, you actually have this fully-connected relationship between your models, views, and controllers (lecture 7):



Now, before you go wild mixing and matching your models, views, and controllers, it's crucial to understand what exactly is represented by each of these arrows:

Controller $\longrightarrow$ **View** : the controller can have a reference to the view and it's type so it can add/remove/update the UI as required. For example, here's the relationship in code:

```
@interface Controller : NSObject {
      PolygonView* polygonView;
    ...
}
```

Controller $\longrightarrow$ **Model** : the controller also has a reference to the model and its type so it can update the state of the object as required (if an action method is invoked on the controller by the view, a timer callback fires, a network transfer is complete, etc). Again, here's the reference code:

```
@interface Controller : NSObject {
      PolygonShape* polyShape;  // modify as needed
    ...
}
```

View → **Model** : the view can actually have a reference to the model and its type, but it **can not** update the model state. That is strictly the job of the controller. By having a direct reference to the model, the view can easily query the model object for its state information (such as numberOfSides) to determine what to render onscreen:

```
@interface PolygonView : UIView {
      PolygonShape* polyShape;  // read-only, no updating!
    ...
}
```

Up to this point, the relationships described all include explicit references to objects. The next set of relationships use callbacks to communicate between two objects. Here's a brief run-down of the 4 main callback mechanisms you'll use in Obj C (with much more info to come later in the course):

• **target/action** - Specify an object and method to be called on that object when a certain event occurs.
• **delegation** - Set your object as another object's delegate so your object can be notified when an event takes place or when your object needs to provide information to the caller object.
• **notification** - Register to be notified when a certain system-wide notification is broadcast using NSNotificationCenter
• **key-value observing** (KVO) - Register to be notified when a single property of another object changes

These methods are all slightly different, but the common thread is that they all involve one class communicating with another class **without knowing the type of the class**. This is key for making your code reusable, extensible, modular, etc.

Getting back to our arrows, the remaining three all use some combination of the callback mechanisms just described:

View → **Controller** : views communicate with controllers usually through either target/action or delegation. Controllers first tell the view they want to respond to an event or that they want to be a delegate for the class. The view then happily calls back on the controller at the appropriate times. Here's an idea of what the view could look like with both target/action and delegation (notice how the types are ids):

```
@protocol PolygonViewDelegate

- (void)polygonView:(PolygonView*)polygonView
willRotateWithDegrees:(float)degrees;
- (void)polygonView:(PolygonView*)polygonView
didRotateWithDegrees:(float)degrees;

@end
```

```
@interface PolygonView : UIView {
        id tappedTarget;
        SEL tappedAction;

        id<PolygonViewDelegate> delegate;
        ...
}
```

Model ⟶ **View** : (same as the next relationship)

Model ⟶ **Controller** : If you come from a Java background, you may be familiar with the Observer pattern where your model class has a list of objects it notifies when the state changes. In Obj C the Observer pattern is used, but the model object isn't responsible for keeping track of the objects interested in the state change. That duty falls on NSNotificationCenter or the KVO system. Instead of giving a code example here, I'll point you to the Apple docs which can explain these mechanisms much better than I can:

- NSNotificationCenter example usage - http://developer.apple.com/iphone/library/documentation/Cocoa/Conceptual/Notifications/Articles/Registering.html#//apple_ref/doc/uid/20000723

- KVO example usage - http://developer.apple.com/iphone/library/documentation/Cocoa/Conceptual/KeyValueObserving/Concepts/KVOBasics.html#//apple_ref/doc/uid/20002252-BAJEAIEE

Again, note that the model class does not have a typed reference to the controller or view classes. There are no Controller or PolygonView instance variables in the model, and that's exactly why the model class is the most reusable piece of code in your project. Because of the loose coupling, they can be passed around and used pretty much anywhere.

Geez, will you stop yappin' already and move on to the next question?

A: Yes.

Question 3

Where should I save and load data from NSUserDefaults?

A: We should break this question into two distinct chunks: loading and saving.

- **Loading:** In HelloPoly, you can load the data in either the app delegate's applicationDidFinishLaunching: or the controller's awakeFromNib method. In my larger projects, I usually instantiate my model objects in applicationDidFinishLaunching,: so I tend to read the defaults from there.

•**Saving:** When saving, you basically have two options: save on state change or save when the application's closing. Most students have been inclined to save only once, when the app is closing since that seems to be the more elegant solution. And in fact, you just need to override the applicationWillTerminate: method in your app delegate to hook in to the shut-down process for your app.

But, the problem is that applicationWillTerminate: is not guaranteed to be called on 100% of shutdowns. The phone OS can kill your process immediately if it needs memory badly enough or if the phone crashes hard, it can totally bypass the delegate method. In these cases, your user loses all their progress and has to go back to the previous starting state.

Saving incrementally avoids these scenarios completely. If you save every time the interface is updated, you're guaranteed to have the most up-to-date state every time you launch the app. It may not at first seem like an ideal solution, but it's actually not all that bad to use.

That's all folks

I was thinking about doing a question on the application life-cycle and explaining the entire init process with and without IB. I'm not sure if that's something people are having issues with, but if you'd find it helpful, let me know, and I'll add it to the next follow-up.

Hope you found the info helpful!