

CS 47

Beginning iPhone Application Development

Week 2: App Fundamentals

Office Hours Update

10-11am

Class Schedule

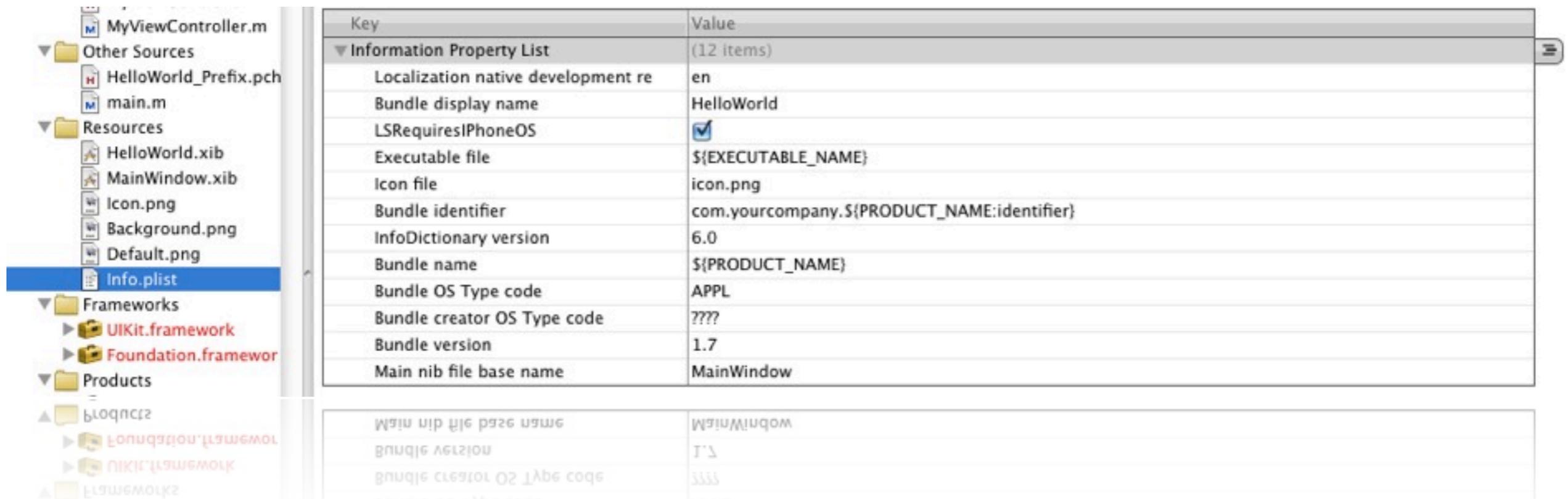
- **Reminder: No class on March 4**
- **Class is extended to March 25, but the last class may not be in this room due to finals. I will keep you updated.**

Agenda

- Basic app fundamentals
 - Structure, initialization
- MVC Framework
- UIView and UIViewController basics

App Fundamentals

- Every app has an info.plist (property list) file
- Contains many basic global settings of an application (e.g. version, name, etc)

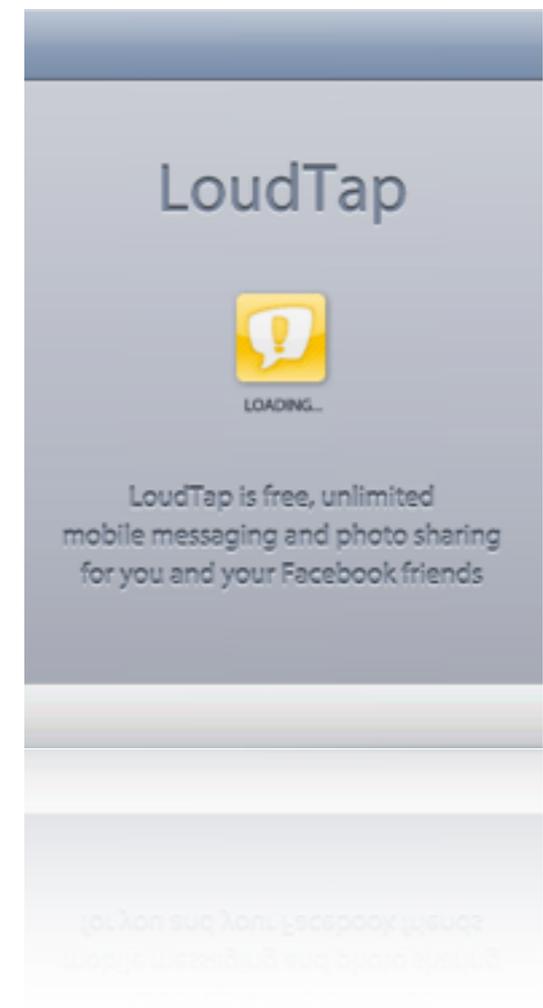
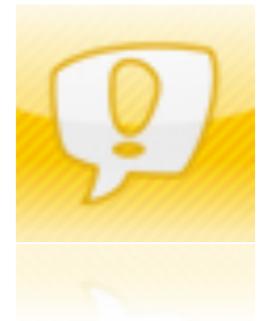


The screenshot shows the Xcode interface. On the left, the project browser displays the file structure, with the 'Info.plist' file selected in the 'Resources' folder. On the right, the 'Information Property List' table is visible, showing the following key-value pairs:

Key	Value
Information Property List	(12 items)
Localization native development region	en
Bundle display name	HelloWorld
LSRequiresiPhoneOS	<input checked="" type="checkbox"/>
Executable file	\${EXECUTABLE_NAME}
Icon file	icon.png
Bundle identifier	com.yourcompany.\${PRODUCT_NAME:identifier}
InfoDictionary version	6.0
Bundle name	\${PRODUCT_NAME}
Bundle OS Type code	APPL
Bundle creator OS Type code	????
Bundle version	1.7
Main nib file base name	MainWindow

App Fundamentals

- The bundle icon filename is defined in the info.plist file (normally icon.png). It must be a 57x57 PNG file
- The splash screen (when is shown before your application is finished loading) is stored in Default.png (case sensitive), and should be a 320x480 PNG file
- Both icon.png and Default.png should be in the resources group of your project



App Fundamentals

- Where does the app start?
- At a basic level, main! (Just like C)
 - main.m, you will never need to change this file.
- Calls UIApplicationMain



The screenshot shows the Xcode interface. On the left, the Project Navigator displays a project structure with folders 'Other Sources' and 'Resources'. The file 'main.m' is selected under 'Other Sources'. The main editor window shows the code for 'main.m' at line 78. The code is as follows:

```
#import <UIKit/UIKit.h>

int main(int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    int retVal = UIApplicationMain(argc, argv, nil, nil);
    [pool release];
    return retVal;
}
```

App Fundamentals

- UIApplicationMain
 - Sets up UIApplication and UIApplicationDelegate objects
 - nil params mean: use defaults (default delegate is determined by the info.plist default nib file).
 - Sets up main event loop

App Delegate

- The app delegate will be the first class you dive into
- It handles all application lifecycle-related callbacks
 - App starting, app ending, pausing due to the idle timer, receiving push notifications, etc.
- Remember: this is a delegate. It does not subclass UIApplication. It's an arbitrary NSObject-based class that conforms to the UIApplicationDelegate protocol

App Delegate

The entrance point that we care about:

- (void) applicationDidFinishLaunching:(UIApplication*)application

This is our only opportunity to initialize the application! After this function we turn over control to the event loop

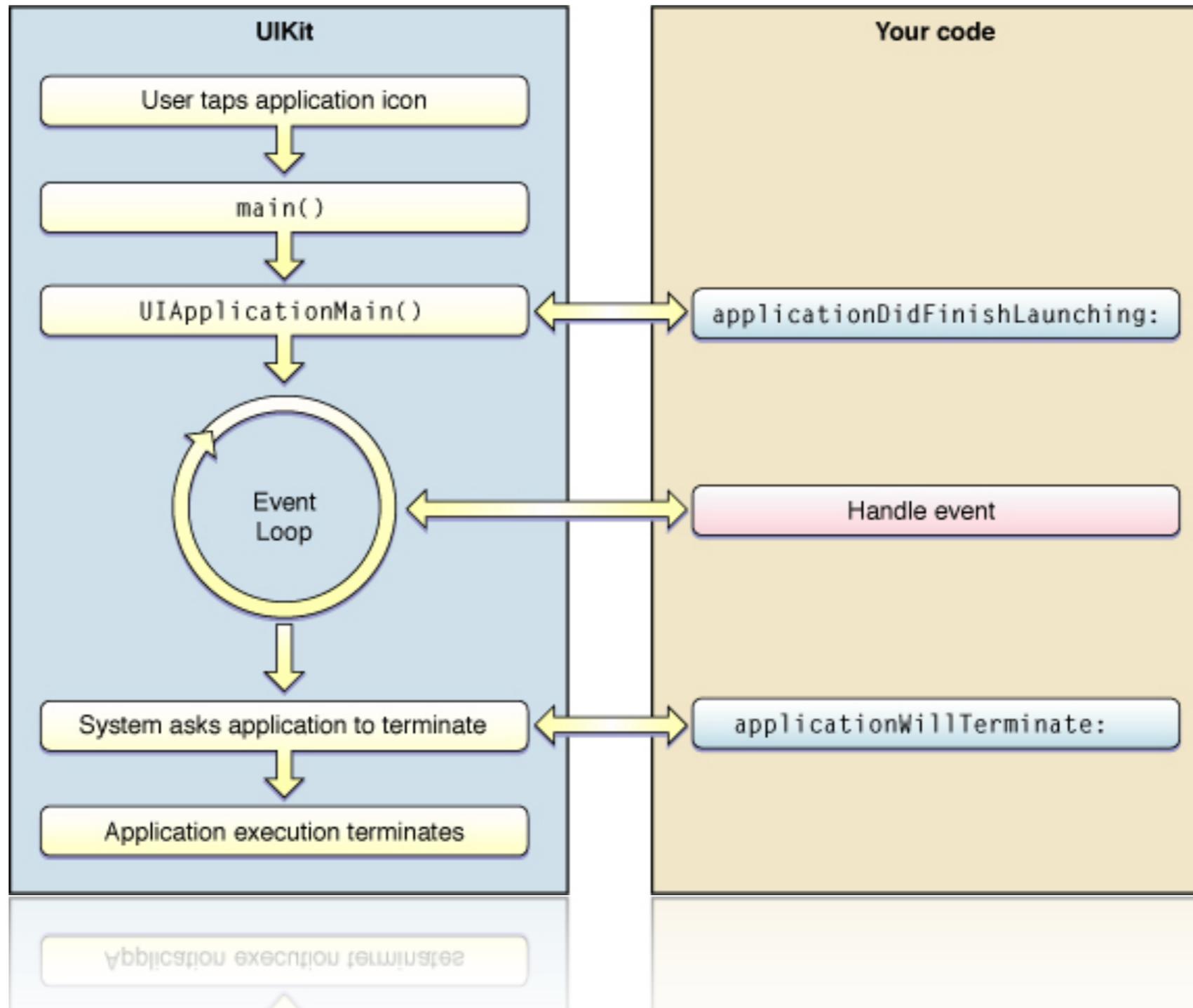
```
- (void)applicationDidFinishLaunching:(UIApplication *)application {  
  
    // Set up the view controller  
    MyViewController *aViewController = [[MyViewController alloc] initWithNibName:@"HelloWorld" bundle:[NSBundle mainBundle]];  
    self.myViewController = aViewController;  
    [aViewController release];  
  
    [[UIApplication sharedApplication] setStatusBarStyle:UIStatusBarStyleBlackOpaque];  
  
    // Add the view controller's view as a subview of the window  
    UIView *controllersView = [myViewController view];  
    [window addSubview:controllersView];  
    [window makeKeyAndVisible];  
}
```

App Delegate

- Our responsibilities in `applicationDidFinishLaunching`:
 - Initialize our data models
 - Setup the main `UIWindow`
 - Setup initial views and view controllers under the window (initial interaction hierarchy)
 - If our data model indicates that we are resuming a session, we need to rebuild the app state to where it was when interrupted.
- Do this as fast as possible or users will hate you

I Thought This Was Cool

Application Lifecycle Diagram



Event Loop

- The event loop looks something like this (behind the scenes):

```
while(wait for event) {  
  
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];  
    handle_event(event);  
    [pool drain];  
  
}
```

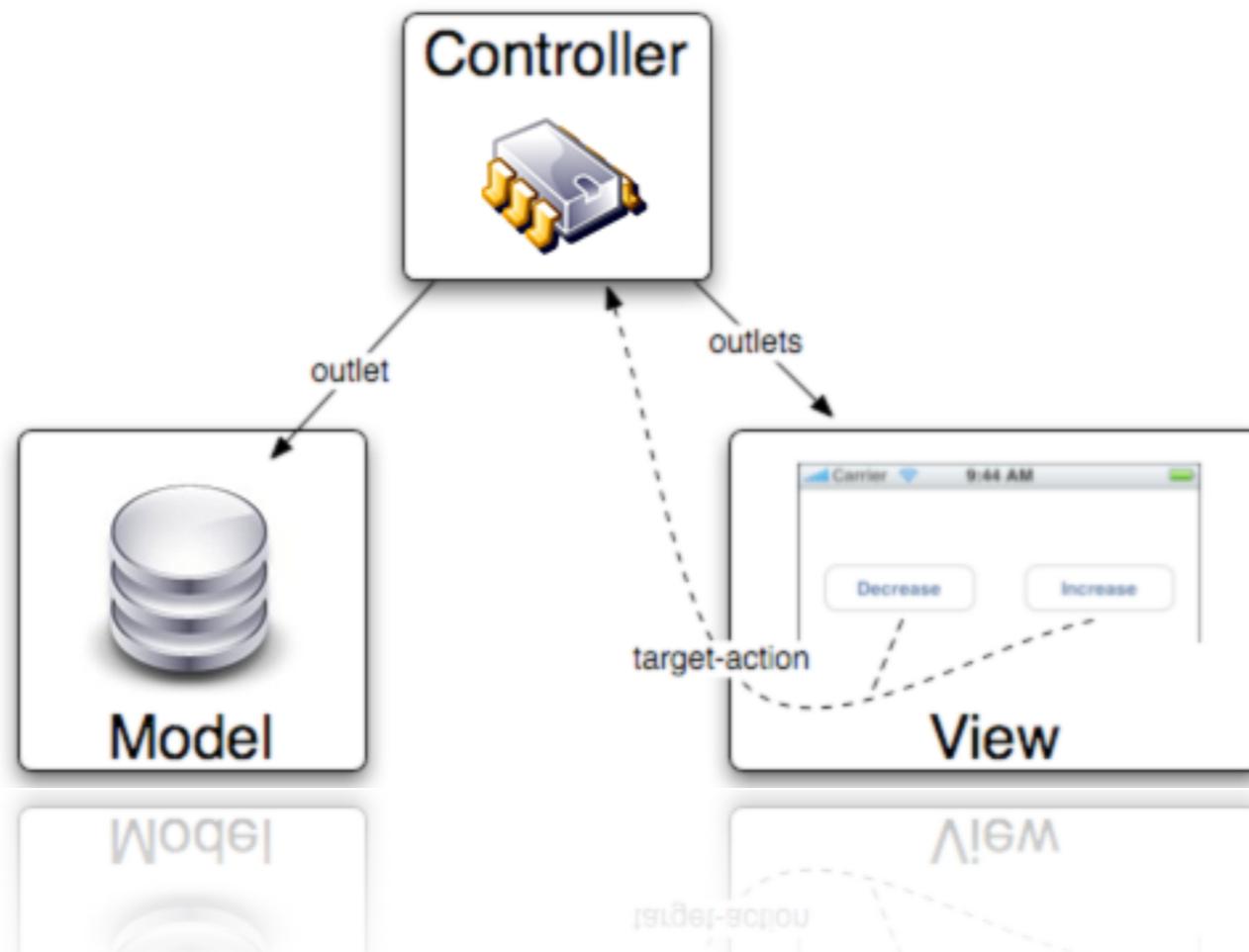
- This is why autorelease works - a new pool is created for each event, and drained when the event is handled

Event Loop

- Notice: events are handled one at a time
- You need to handle events quickly or your application will appear unresponsive
- Querying things that may not be available immediately needs to be done asynchronously or in another thread

MVC

- The iPhone SDK is based on the model-view-controller paradigm



MVC

- Why?
 - Reduces spaghetti code
 - Abstraction! Easier to compartmentalize areas of code in your mind
 - Separate reusable code (model, view) from specific-purpose code (controller)

Model

- Responsible for handling the organization of you application's data:
 - Storage (database or flat file)
 - Retrieval (from disk or remote system)
 - Analysis (basic data analytic functions)

Model

- Models can have very complex implementations (e.g. caching policies, complex databases, remote queries)
- But they should always have as simple an interface as possible for your application controllers

Abstraction and reusable code!

Views

- Represented by the UIView class
- Responsible for handling all I/O with the user
 - Displaying information to the user
 - Recognizing user input
- Views are controlled by (you guessed it) controllers! Controllers tell them what to display, and the views send input events back to the controllers

Views

- You should also strive for reusable code when implementing views
- Your views should display specific things in a specific manner, and have a good API to affect that display - don't pack too much functionality into a single view class!
- Implement callbacks for general events (e.g. touched, modified, etc)

Views

- In the iPhone SDK, UIViews create a view hierarchy
- There is a key window (ref: `makeKeyAndVisible`) that is the top-level UIView in the application
- UIViews are attached to a parent UIView with the `addSubview:` message
- UIViews can be removed with `removeFromSuperview:`

Views

- UIViews are displayed relative to their parent UIView
- Position, transparency, touch-enabled state, etc - all are relative to the parent UIView
 - e.g. if the parent UIView is set to $\alpha=0.5$ (half blend), then all child UIViews have their alphas halved

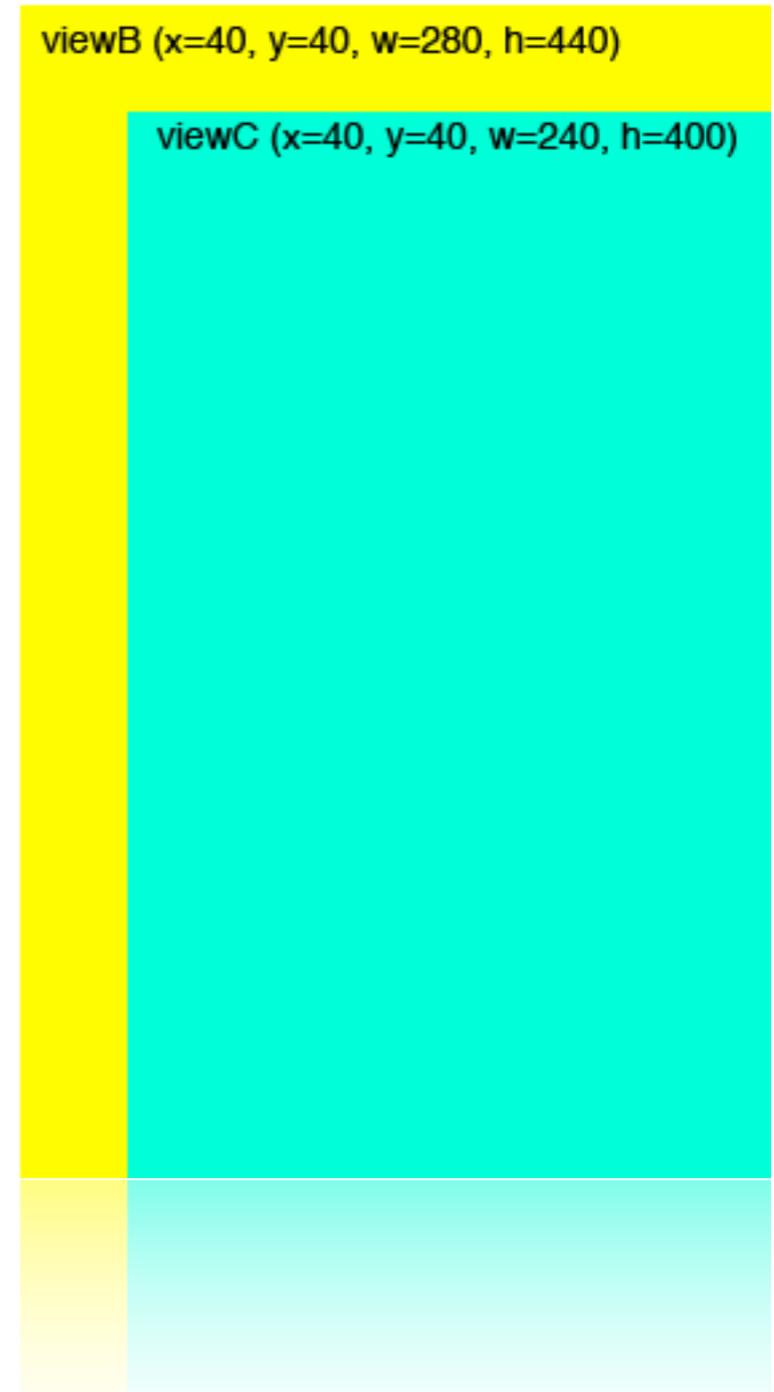
Views

- Positioning example:
- C is a subview of B
- B is a subview of A
- Notice B and C both have defined origins at 40,40
- But C is at absolute screen origin 80,80 since its origin is relative to B's
- Note that width/height are always absolute

viewA (x=0, y=0, w=320, h=480)

viewB (x=40, y=40, w=280, h=440)

viewC (x=40, y=40, w=240, h=400)



Views

- The view hierarchy is dynamic!
- The top-level UIWindow is always present, but its children are continuously swapped out as the user navigates around the application
- These hierarchy transitions are handled by UIViewControllers

Controllers

- The controller is the glue logic that ties the model and views together
 - Takes model info and displays it through views
 - Takes input from views and affects the model
 - This is where app-specific code goes! Controllers are generally not reusable
- Deals with other event-driven callbacks like timers or asynchronous data requests
- Controls the dynamic view hierarchy
- Controllers can have their own hierarchy (UINavigationController, UITabBarController, etc)

Controllers

- Think of a typical iPhone application that navigates through a sequence of distinct screens.
- Generally, every screen in an application is represented by its own subclassed `UIViewController`

Controllers

- One UIViewController per screen:
 - Try to focus on one UI interaction behavior per screen
 - Presenting data to the user in an expected and easy-to-use way
 - Link to other screens to get to other types of data/interactions

Controllers

- There is no standard mechanism for a `UIViewController` to interact with the model.
- Completely up to your discretion and heavily influenced by your model's implementation

Controllers

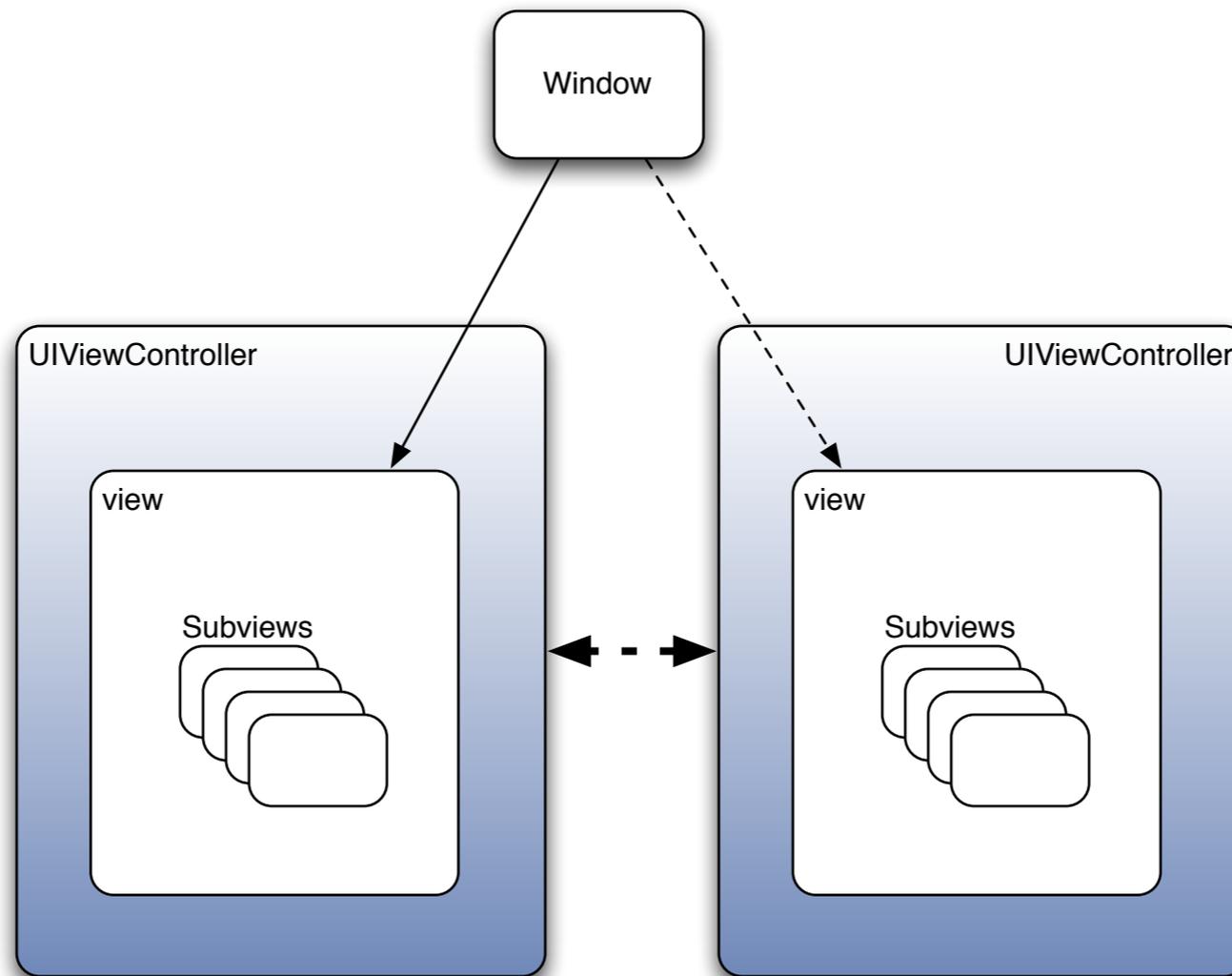
- Controller -> View interaction is also flexible, but there are some rules
- Every UIViewController has a `view` instance variable

```
@property (retain) UIView view;
```
- If you are subclassing from UIViewController, it is your responsibility to initialize this instance variable

Controllers

- The `loadView:` message is called if `view` is `nil` when accessed (`UIView *v = myViewController.view`)
- Usually, in the same place that you initialize the `view` instance member you also take the time initialize and add the entire `UIView` hierarchy of this controller to the `view`
- e.g. adding tables, buttons, labels, pickers, etc

Controllers



UITableViewController themselves are NOT part of the view hierarchy!

Remember: `[window addSubview:myViewController.view];`

Controllers

- What if you want to animate between screens?
Use special controllers designed to manage UIViewControllers
- You could use a UINavigationController, which animates left-right transitions and handles the navbar at the top of the screen
- Or a UITabBarController, which provides a selection of UIViewControllers in its tab at the bottom of the screen

Controllers

- Or you can implement your own transition style! Your only responsibility is removing the active `UIViewController`'s `view` from the hierarchy and bringing another one in.

Controllers

- Why else is it important to manage UIViewControllers?
- Remember that our view hierarchy only retains the UIViewController's `view` member (during `addSubview:`)
- It does not implicitly retain the entire UIViewController object!

Controllers

- Use UINavigationController manager classes to help you manage UINavigationController lifecycle
- UINavigationController and UITabBarController will retain your UIViewControllers until no longer needed
- Think about what this means: UIViewControllers are created and destroyed just like any other class

Controllers

- Since they can be instantiated and destroyed so often, you want to make sure your `UIViewController` subclasses can initialize and destruct quickly, without memory leaks
- Yes, you can have global view controllers, but it's not ideal.

Controllers

Global UIViewController (sharedInstance)

```
@interface MyViewController : UIViewController {  
}  
- (MyViewController*) sharedInstance;  
@end
```

```
@implementation MyViewController
```

```
- (MyViewController*) sharedInstance {  
    static MyViewController *singleton_instance = nil;  
    if (!singleton_instance) {  
        singleton_instance = [[MyViewController alloc] init];  
    }  
    return singleton_instance;  
}
```

Controllers

- Another note on UIViewController managers: they call various lifecycle messages as your controllers come in and out of view:
 - viewWillAppear:
 - viewDidAppear:
 - viewWillDisappear:
 - viewDidDisappear:

MVC Interaction

- Let's recap how components of an MVC system interact

MVC Interaction

Controller-Model

- As discussed before: This is easy... it's up to you!
- Generally the controller queries for information from the model (pull)
- The model may use a protocol to inform the controller that asynchronous info is ready (push)

MVC Interaction

Model-View

- This is tricky. There is a trade-off between reusable and succinct code.
- It may be very easy/fast to assign entire model objects to view instance variables in order to pass in much information at once, but this ties the view to the model
- Must be considered on a case-by-case basis, no general rules

MVC Interaction

Controller-View

- Controllers setup views and tell them what data to display
- Views tell controllers when user interaction occurs
 - Either through delegate methods, or target-action

MVC Interaction

Controller-View

- We know what delegates are. But what is target-action?
- Every UIView has a list of events that it can handle (UIControlEventXXX)
- When an event is detected by the view, it looks up an internal target-action table to see if anyone needs to be notified of that event

MVC Interaction

Controller-View

```
@implementation MyViewController
```

```
- (id) init {  
    if (self = [super init]) {  
        ...  
        myButton = [UIButton buttonWithType:UIButtonTypeCustom];  
        [myButton addTarget:self  
            action:@selector(buttonPressed:)  
            forControlEvents:UIControlEventTouchUpInside];  
        [self.view addSubview:myButton];  
        ...  
    }  
    return self;  
}  
  
- (void) buttonPressed:(id)sender {  
    NSLog(@"Button pressed!!");  
}  
  
@end
```

Demo Time

