Intermediate iOS

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 17 — 10/18/2011

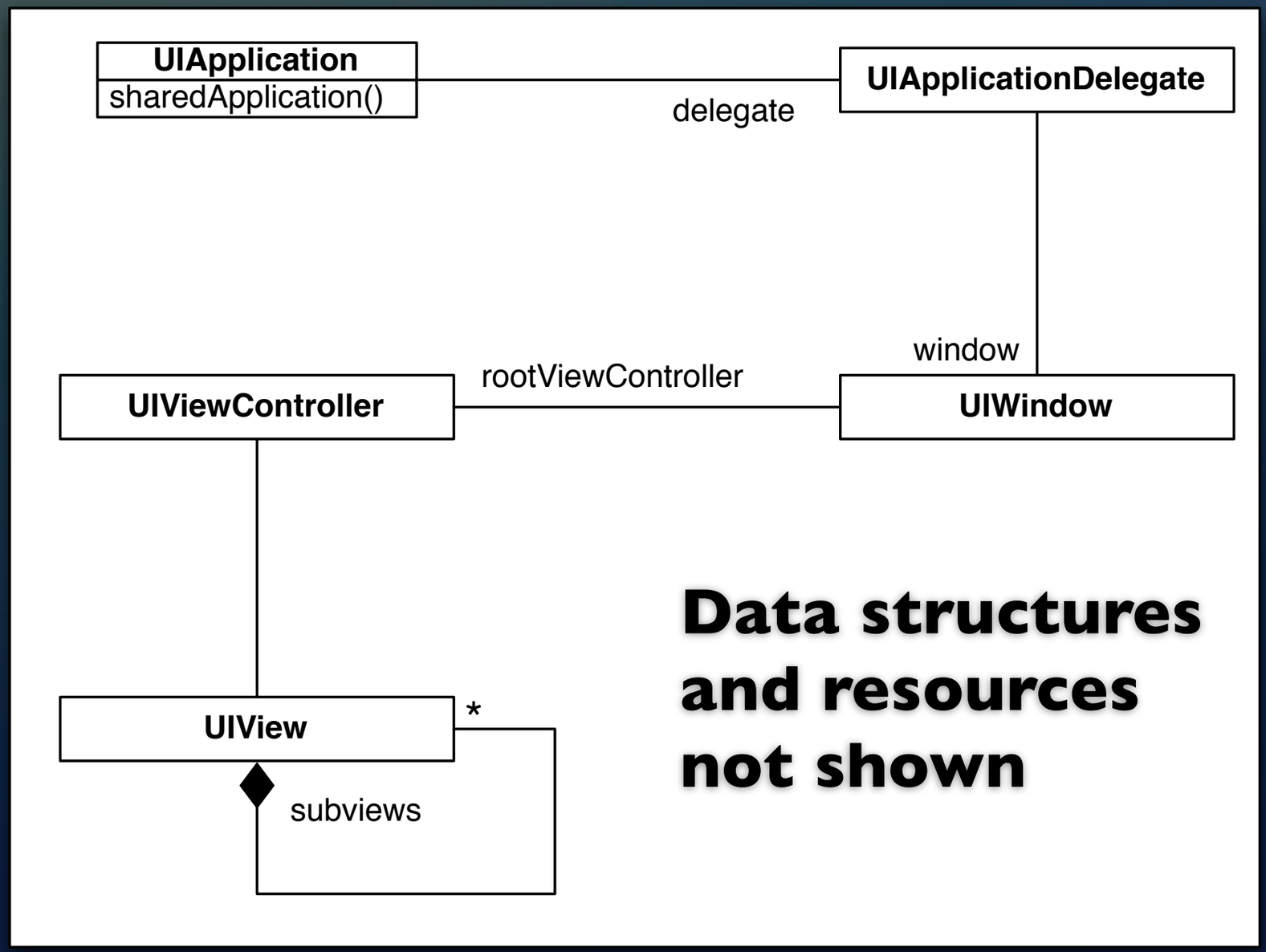
Goals of the Lecture

- Learn more about iOS
- In particular, focusing on the concept of views and their associated view controllers
- But also covering: autorelease, @selector, the use of Instruments to track allocations, gesture recognizers, animation, split view controllers & table view controllers!

iOS Fundamentals (I)

- Each iOS application has
  - one application delegate
  - one window
  - one or more view controllers
    - each view controller has one view that typically has many sub-views arranged in a tree structure
      - e.g. views contain panels contain lists contain items…

iOS Application Architecture



iOS Fundamentals (II)

- A window will have a "root" view controller
  - Some view controllers allow us to "push" a new view controller onto a stack (similar to Android's activity stack)
  - the new view controller's view is then displayed
    - When we "pop" that view controller off the stack, we return to the view of the previous view controller

iOS Fundamentals (III)

- At other times, we may switch the "root" view controller entirely
  - the new view is displayed and the previous view controller (and its view) is deallocated

iOS Fundamentals (IV)

- View controllers can be instantiated and activated via the use of .xib files (as we saw in Lecture 13) or they can be created programmatically
  - They, in turn, can create their view through the use of a .xib file or create their view programmatically
- We'll see examples of both in this lecture

iOS Fundamentals (IV)

- View controllers are very powerful
  - they handle the creation of views
  - they handle navigation among views and other view controllers
  - they help free up memory when a view is no longer being displayed
  - they handle the rotation of views when a device's orientation changes

Simple View-Based Application

- Image Switcher
  - View-Based Application Template
    - Two image views
    - One page controller
    - The two image views will work with the page controller to make it look like multiple images are available to display

Framework from Slide 4 Writ Large

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    self.window = [[[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]] autorelease];
    self.viewController = [[[ViewController alloc] initWithNibName:@"ViewController" bundle:nil] autorelease];
    self.window.rootViewController = self.viewController;
    [self.window makeKeyAndVisible];
    return YES;
}
```

Take a look at the default launch code in AppDelegate.m

The app delegate has a reference to a window (by creating one); it creates a view controller and assigns it as the window's rootViewController.

The View Controller's View is created with the .xib file loads

Step 1: Edit View Controller's XIB File

- Edit **ViewController.xib**
  - Change the view's background to black
  - Add two image views and one page controller
    - one image view directly on top of the other
    - the page controller should be "on top" of the two image views; it should be configured to have 5 pages
    - tag the image views as "0" and "1" (using attributes pane)
  - Use the outline view of the dock to select the views

Step 2: Generate outlets

- XCode can
  - auto-generate properties in the view controller
- and
  - generate the connections between the properties in the .h file and the widgets in the .xib file
- Enable the assistant editor by clicking the tuxedo icon
  - Then control drag from the .xib to the .h
  - XCode automatically picks the right .h file

Step 2: Continued

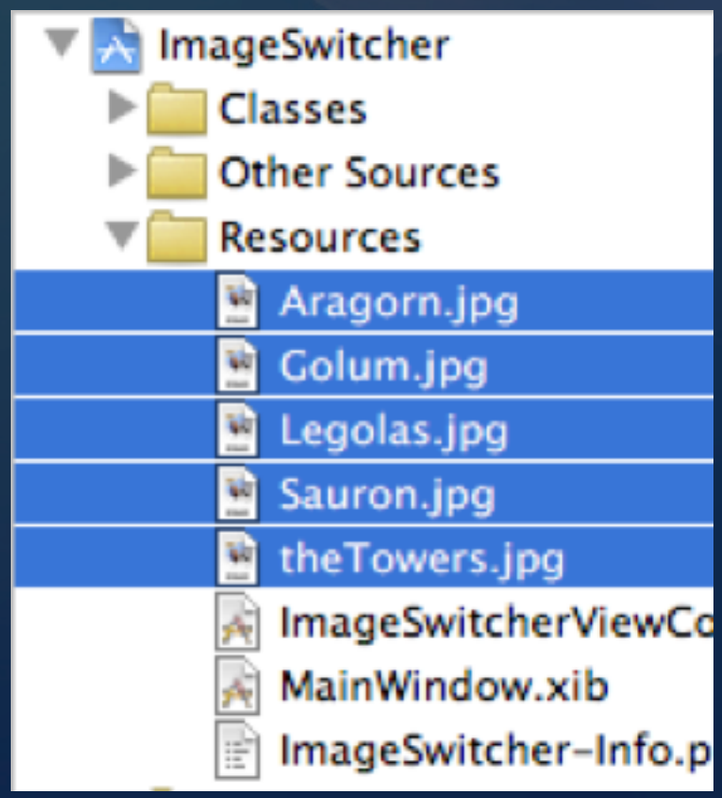
- When prompted name the outlets
  - imageOne (tag: 0), imageTwo (tag: 1), pages
- Switch to ViewController.m
  - You will see that XCode automatically
    - added an @synthesize for each property
    - added statements to dealloc and viewDidUnload to take care of releasing the properties at the appropriate time

Step 2: Continued

- To follow the guidelines I outlined in the previous lecture, we will want to
  - add a name for the instance variable
    - @synthesize pages = _pages;
  - update viewDidLoad to use properties
    - self.pages = nil;
  - update dealloc to use the new instance variable name
    - [_pages release];

Step 3: Add Images

- You can do this via Drag and Drop or Project ⇒ Add to Project…
  - Tell XCode to copy the files
- You want the images to end up in the Resources folder of the project view
  - So select all the image files, right click, and select "New Group from selection"
  - Rename group to "Resources"



Step 4: Write the Code

- We need to write code for two methods
  - The first is **viewDidLoad**; This is a view controller method that gets invoked just after it has created its view and just before that view gets displayed
    - This is your opportunity to initialize the view
  - The second is **pageTurning:**; this is a method we will create ourselves; we'll tell the page control to call this method when the user asks it to turn the page

viewDidLoad (I)

- In this method, we will
  - ask one image view to load an image
    - each image loaded will be cached automatically
  - hide the other image view
  - set up our variables "front" and "back"
  - tell the page control which method to invoke

viewDidLoad (II)

```
30  - (void)viewDidLoad {
31      [super viewDidLoad];
32
33      self.imageOne.image = [UIImage imageNamed:@"Aragorn.jpg"];
34
35      self.front = self.imageOne;
36      self.back = self.imageTwo;
37
38      [self.imageOne setHidden:NO];
39      [self.imageTwo setHidden:YES];
40
41      [self.pages addTarget:self
42                     action:@selector(pageTurning:)
43           forControlEvents:UIControlEventValueChanged];
44
45  }
```

WAIT A MINUTE!!!

- self.front? self.back? @selector(pageTurning:)
  - Where are these defined?
  - The answer: nowhere… yet!
- These two properties and the method will
  - help us swap between images, and
  - serve as the event handler when the page control is clicked

Where do we put them?

- They are thus features of the view controller that are not public; we want access to them in the .m file but we don't want other classes to know about them
- Solution: Objective-C 2.0 Class Extensions
  - A class extension allows us to reopen the class definition and add additional features that are only visible to the .m file
- If you define properties in the class extension, you still have to @synthesize them in the @implementation section

Syntax of Class Extension

- @interface ClassName ()
  - <property and method defs go here>
- @end

Don't forget to synthesize the front and back properties!

```
9  #import "ViewController.h"
10
11 @interface ViewController ()
12
13 @property (nonatomic, assign) UIImageView* front;
14 @property (nonatomic, assign) UIImageView* back;
15
16 - (void) pageTurning: (UIPageControl*) sender;
17
18 @end
```

Back to viewDidLoad (III)

```
30  - (void)viewDidLoad {
31      [super viewDidLoad];
32
33      self.imageOne.image = [UIImage imageNamed:@"Aragorn.jpg"];
34
35      self.front = self.imageOne;
36      self.back = self.imageTwo;
37
38      [self.imageOne setHidden:NO];
39      [self.imageTwo setHidden:YES];
40
41      [self.pages addTarget:self
42                     action:@selector(pageTurning:)
43           forControlEvents:UIControlEventValueChanged];
44
45  }
```

Discussion of Code (I)

- Having defined and synthesized the front and back properties, we can see that
  - they simply point at the image views and keep track of which one is visible (front) and which one is invisible

Discussion of Code (II)

- Having declared the pageTurning: method, we can see that the last thing viewDidLoad does is connect the UIPageControl to the pageTurning: method
  - The method call **addTarget:action:forControlEvents:** is invoked on self.pages (the UIPageControl)
  - It (essentially) says:
    - when your value changes invoke pageTurning: on "self" which is our ViewController object

This is equivalent to control dragging from the widget to the event hander in the XIB Editor

Discussion of Code (III)

- Note the use of
  - **@selector(pageTurning:)**
- to reference the method that needs to be invoked

- Every method is given an id at run-time
  - **@selector(method name)** returns that id and allows it to be invoked dynamically

Step 5: Implement switching images

- We will make sure we can change the images first
  - Then we'll add animation
- We will ask the page control which page we are turning to
  - We'll then load the appropriate image into the background image view and
  - swap the visibilities of the two image views and
  - update our pointers

```objc
- (void) pageTurning: (UIPageControl*) sender {
    NSInteger nextPage = [sender currentPage];
    switch (nextPage) {
        case 0:
            self.back.image = [UIImage imageNamed:@"Aragorn.jpg"];
            break;
        case 1:
            self.back.image = [UIImage imageNamed:@"Gollum.jpg"];
            break;
        case 2:
            self.back.image = [UIImage imageNamed:@"Legolas.jpg"];
            break;
        case 3:
            self.back.image = [UIImage imageNamed:@"Sauron.jpg"];
            break;
        case 4:
            self.back.image = [UIImage imageNamed:@"theTowers.jpg"];
            break;
        default:
            break;
    }

    [self.front setHidden:YES];
    [self.back setHidden:NO];

    if (self.front.tag == 0) {
        self.front = self.imageTwo;
        self.back = self.imageOne;
    } else {
        self.front = self.imageOne;
        self.back = self.imageTwo;
    }
}
```

Whenever this method gets invoked, we know that front points to the image currently displayed

we load the next image into back

then we hide the front and show the back

and then we swap our pointers

Step 7: Add the animation

- We'll add a simple flip animation when we turn between pages
- The style of animation that we will use is very similar to the "tweening" animation we saw for Android
- The only difference is that iOS animations are specified programmatically using Core Animation rather than using resources as we did in Android

Skinning the Cat

- There are many ways to do animations in iOS
  - We will use the new "block-style" animations introduced in iOS 4
- But first, in order to do this right, we'll need to keep track of which page we are on and then determine if we need to flip left or right
  - We'll add an integer property called current to keep track of the current page

```
UIViewAnimationOptions options;

if (self.current < nextPage) {
    options = UIViewAnimationOptionTransitionFlipFromLeft;
} else {
    options = UIViewAnimationOptionTransitionFlipFromRight;
}

[UIView transitionWithView:self.view
                  duration:1.0
                   options:options
                animations:^{
                    [self.front setHidden:YES];
                    [self.back setHidden:NO];
                }
                completion:^(BOOL finished) {
                    if (finished) {
                        UIImageView *temp = self.front;
                        self.front = self.back;
                        self.back = temp;

                        self.current = nextPage;
                    }
}];
```

We can now use the current page to determine which way to flip and call transitionWithView:duration:options:animations:completion: to animate the change. This code uses blocks to specify what happens during the animation and what happens after the animation is done

Step 8: Add Gesture Recognition

- Sometimes it's hard to click on the page control just right
  - It's more natural on a touch device to just swipe between images
- iOS makes it easy to detect a swipe gesture using its gesture recognizers

Game Plan

- First, we'll instantiate gesture recognizers in viewDidLoad: and add them to our root view
  - Our image views don't handle user input, so all touches and swipes will be directed to the root view of the view controller
- Second, we'll configure the gesture recognizers to call a method called handleSwipe:
  - We'll add handleSwipe to our class extension

Image Switcher Wrap Up

- Here we had a single view with three subviews
  - With some trickery, we made it look like our application had five images
    - with only one ever being displayed at a time
- We needed two image controls to enable the animation
- The 5 images are cached (only loaded once); **UIImage** is able to detect low memory situations and empty its cache as needed

Programmatic View Creation

- So far we have created views only via XIB files
  - Occasionally, you will be in situations where you need to create a view programmatically
    - To do this, you create a View Controller with no associated XIB file and then create the contents of your view in **viewDidLoad**;
      - View Controllers also have a method called **loadView**; leave it alone, its default behavior does just what we need

View Switcher

- Let's create an application with three view controllers
  - Each view controller will programmatically create a view that contains a label and 2 buttons
    - The label will state which screen we are looking at
    - The buttons will take us to the other screens
- To switch among the views, we will install the appropriate view controller as the application window's root view controller

Step 1: Create Window-Based iOS Application

- Call it View Switcher
  - This template contains only a single window and a single app delegate
    - No view or view controller is created by default
  - Our window has a white background by default, so that's what we see if we run the default project
  - Each screen will have a different color (red, green, blue) to distinguish our views from the window

Step 2: Create Screen One

- Select the Classes Folder and then invoke **⌘-N** to bring up the New File dialog
  - Indicate that you want an Objective-C class that is a subclass of NSObject; name it ScreenOne
    - Now change its supertype to be UIViewController
    - The default template of UIViewController contains a lot of code that can be confusing
      - better to start with NSObject's clean template

Step 3: Create Label and Buttons

- At a high-level, we will
  - override the **viewDidLoad** method to
    - programmatically create a Screen One label and "Go To Screen Two" and "Go To Screen Three" buttons
    - set the background to a nice shade of red
- Take a look at the source code for details

Step 4: Arrange Screen One Creation

- Now that we have created the ScreenOne view controller, we need to arrange for an instance of it to be created
  - To do this, we will override a method in our application delegate, **application:didFinishLaunchingWithOptions:**
- This method gets called after the application has launched but before the application's window appears
  - We need to import ScreenOne.h, instantiate an instance and set it as the root view controller

Step 5: Create Screen Two and Three

- These classes will be exactly the same as ScreenOne except for label/button names and the background color of the view
- See example code for details

Step 6: Wire up the Buttons

- Since we are not using Interface Builder to create our views, we have to wire our buttons to their event handlers programmatically
  - Just like we did with the UIPageController in the previous example
  - See sample code for details
- Now, we need to implement the button event handlers

Switching the root view controller (I)

- Our button event handler has to do the following
  - Get a handle to the app delegate
  - Use the app delegate to get a handle to our window
  - Instantiate an instance of the new view controller
  - Autorelease the new view controller (hold that thought)
  - Set the new view controller as the root view controller

Switching the root view controller (II)

```
- (void) goToScreenTwo:(UIButton*) sender {

    AppDelegate* delegate = (AppDelegate*)[UIApplication sharedApplication].delegate;

    ScreenTwo* screenTwo = [[ScreenTwo alloc] initWithNibName:nil bundle:nil];
    [screenTwo autorelease];

    delegate.window.rootViewController = screenTwo;

}
```

autorelease? (I)

- We have finally seen a situation that requires **autorelease**
  - This method is one of the memory management routines; here is why we need it
- If we don't use it, then
  - we create an instance of the view controller
    - retain count defaults to 1
  - we then pass it to the window, which retains it
    - retain count incremented to 2

autorelease? (II)

- And then?
  - We never see that object again and so we are unable to release it
  - When we finally set a new root view controller, the previous root controller gets released and now its retain count returns to 1
    - which means it never goes away: **memory leak!**

autorelease? (III)

- So, the question becomes how do we release the view controller after we create it, so that eventually its retain count will go to zero
- We can't release it **before** we pass it to window
  - If we do, its count goes to zero immediately and it gets deallocated and we end up passing a deallocated object to the window
- So, we autorelease it

autorelease? (IV)

- When you autorelease the view controller
- It gets added to the current autorelease pool, which is automatically created before the event handler is called
- It gets passed to the window: retain count == 2
- The event handler ends and the pool is flushed; When the pool is flushed, it releases all of the objects within it; retain count == 1
  - When the root view controller is updated, the previous root controller is released and deallocated

Tracking Memory

- We can run our app in a program called **Instruments** which allows us to track allocations (among other things)
  - We can then verify that our view controllers are being deallocated
    - We can then be confident that only one view controller is ever allocated in the ViewSwitcher application
- **Demo**

Split View Controller

- Let's take a look at a more complicated example
- A Split View Controller was added when the iPad came out to make it easy for an application to
  - have a list of items on the left
  - and a detail viewing space on the right
  - when an item in the list is selected, its details are displayed
  - the items are shown in a table when in landscape mode and in a pop-up list when in portrait mode

SplitView Template

- The default template for a split view application is configured, like all other templates, to work right away
- It displays a simple list of items "row 1, row 2, etc." and a detail view containing a label
  - when a row is selected, the label updates
  - (see next slide)

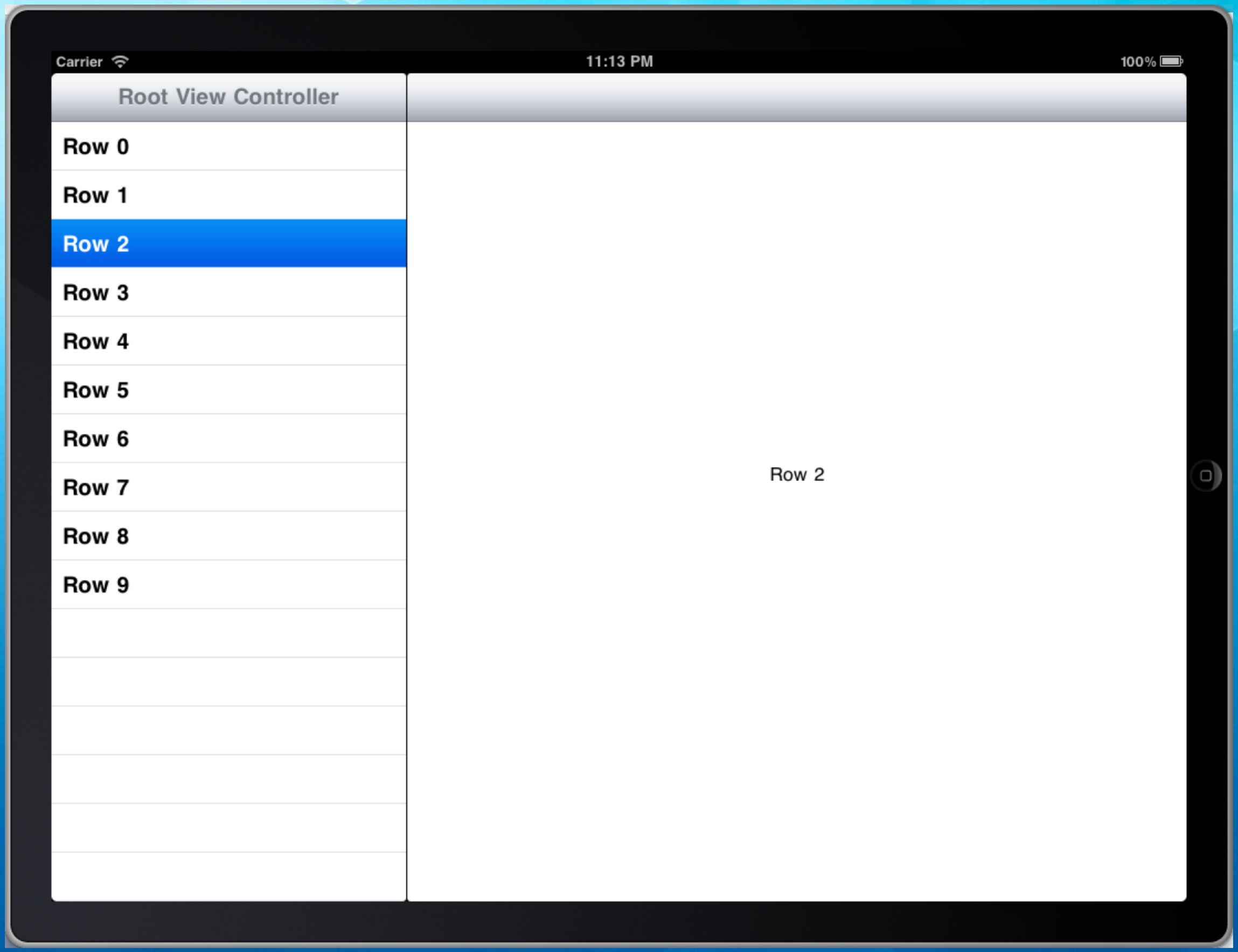

Carrier
11:13 PM
100%
Root View Controller
Row 0
Row 1
Row 2
Row 3
Row 4
Row 5
Row 6
Row 7
Row 8
Row 9
Row 2

Image Switcher Lives Again

- Let's explore the split-view template by recreating image switcher for the iPad; Create a Split View-based application and call it SplitViewer
  - This template comes with
    - A split view controller created in **MainWindow.xib**
    - Two view controllers: root and detail
      - root is a subclass of **UITableViewController**
      - detail is a **UIViewController** that implements two interfaces: **UISplitViewControllerDelegate** and **UIPopoverControllerDelegate**

Step One: Copy Images

- Drag and Drop the images from Image Switcher into the Resources folder of Split Viewer and copy them across
  - It's important that you drag and drop the images into the resources folder contained within the XCode window
  - If you copy the images to the SplitViewer folder in the Finder without copying them into the project, XCode won't be able to find them

Step 2: Prepare the Detail View

- We need to delete the label that is included in **DetailView.xib** by default
  - Replace it with an image view
  - Center the image, make it big, set its autosize constraints, etc.
  - Save your changes, add an outlet/property in the .h file and synthesize the property in the .m file
  - Go back to IB and connect the UIImageView to the property

Step 3: Init array of image names

- In the **viewDidLoad** method of the root view controller, we will create an array of image names
  - We will then use this array to populate the table
- We will also set the title of the navigation bar to "Lord of the Rings"

Populating a Table

- To populate a table, you implement three methods
  - **numberOfSectionsInTableView:**
    - return 1
  - **tableView:numberOfRowsInSection:**
    - return the size of the array
  - **tableView:cellForRowAtIndexPath:**
    - Very powerful, slightly complex code (see next slide)

```objc
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *CellIdentifier = @"CellIdentifier";

    // Dequeue or create a cell of the appropriate type.
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault reuseIdentifier:CellIdentifier] autorelease];
        cell.accessoryType = UITableViewCellAccessoryNone;
    }

    // Configure the cell.
    [[cell textLabel] setText:[images objectAtIndex:[indexPath row]]];
    return cell;
}
```

The above code is an iOS design pattern that ensures that you never allocate more table cells than you need

If a table cell scrolls off the top or bottom of a table, it becomes available to be used again; that is, the call to dequeueReusableCell... will return a pointer to a cell that is no longer visible on screen

You can then customize its contents based on the row it represents; it will then be displayed with the new content

A table with 1000s of rows may only have 10 cells allocated!

Standard approach to Table creation

- This approach of implementing a table by implementing "data source" methods is standard across many UI frameworks
  - Rather than create a table, you create its data source
  - The table asks you: "how many sections do I have" or "how many rows are in section 1" or "what cell should I display for row 6"
    - and you give it the answers
- This is delegation at work… no need to subclass **UITable**

Step 4: Handle a Selection

- Next we need to handle the selection of a name in the table
  - We implement the method **tableView:didSelectRowAtIndexPath:**
    - We are told the selected row
    - We use that to retrieve the image name
    - We append ".jpg" to the name and pass that modified name to the detail view by calling setDetailItem: on the detailViewController

Step 5: Update the Image View

- When the detail item has been updated, a customer "setter" is invoked on detail view controller
- In that setter, we call **configureView** and in that method, we can set the desired image on the image view in the same way we did in Image Switcher
- And with that we are done, the default template automatically takes care of creating, showing and hiding the pop-up control based on changes in orientation

Wrapping Up (I)

- Learned the fundamentals of view controllers
  - View-based Application template
  - Window-based Application template
    - Creating views and view controllers programmatically
    - Switching between view controllers
- Discussed autorelease, @selector
- Saw new widgets: UIImageView, UIPageControl

Wrapping Up (II)

- New View Controllers
  - UISplitViewController, UITableViewController
- Gesture Recognition
- Animation Support
- Allocation Tracking with Instruments

Coming Up Next

- Lecture 18: Review of Midterm
- Homework 5 Assigned on Friday
- Lecture 19: Advanced Android
- Lecture 20: Advanced iOS
  - or
- Lecture 18: Advanced Android
- Lecture 19: Review of Midterm (we'll see!)