

INTERMEDIATE IOS

CSCI 4448/5448: OBJECT-ORIENTED ANALYSIS & DESIGN

LECTURE 17 — 03/08/2011

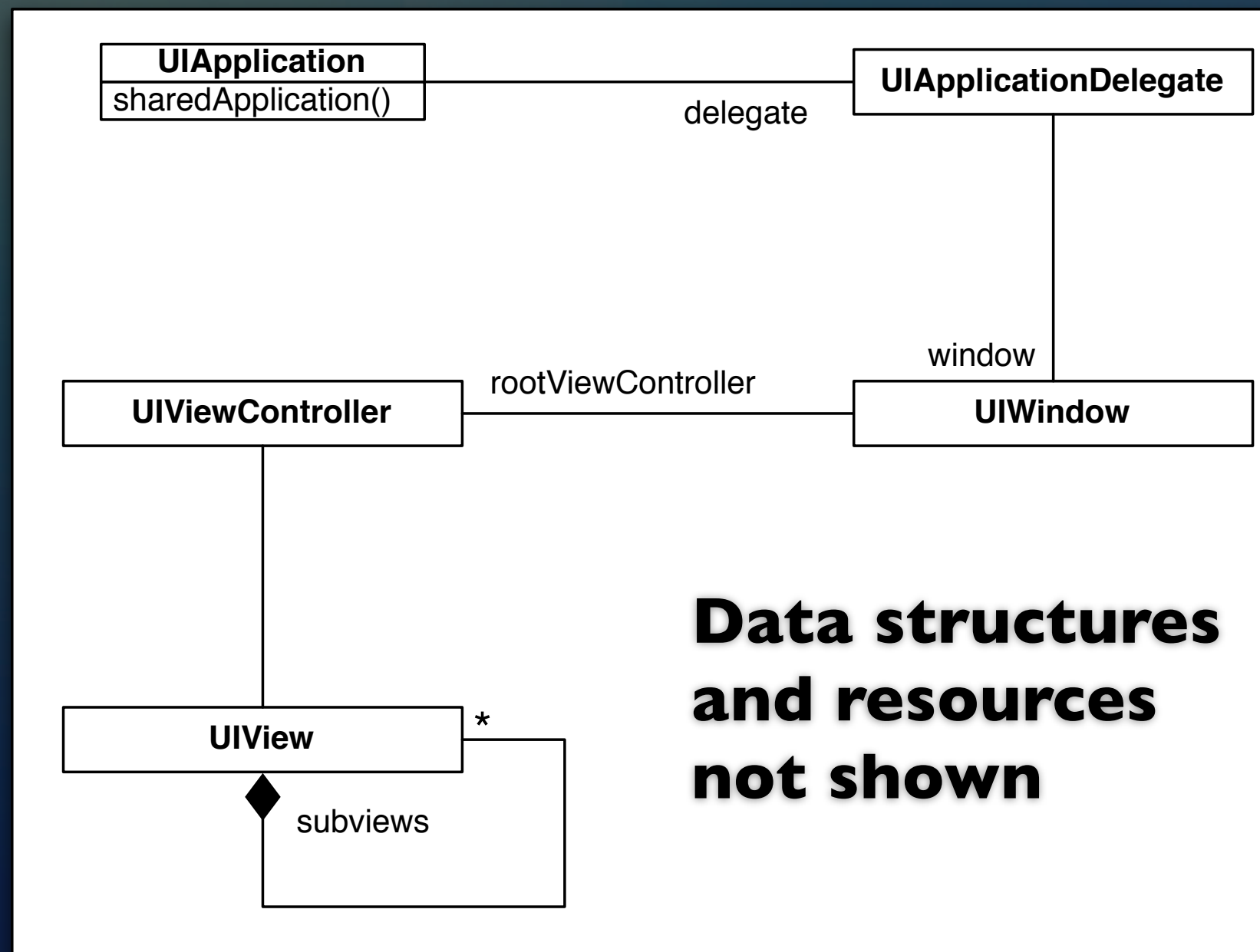
Goals of the Lecture

- 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
 - 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 (III)

- View controllers can be instantiated and activated via the use of .xib files (as we saw last time) 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

With respect to orientation

- Back in lecture 13, I had a problem in which my view would not change orientation when I rotated the iPhone simulator
- I blame myself for not appeasing the demo gods AND (more importantly) I forgot to edit a method in my view controller
- Each view controller can override the following method
 - **shouldAutorotateToInterfaceOrientation:**
- Its default code keeps views in a portrait orientation

Default Code

```
57  /*
58  - (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)interfaceOrientation {
59      // Return YES for supported orientations
60      // return (interfaceOrientation == UIInterfaceOrientationPortrait;
61  }
62  */
63  |
```

- ❖ The default code is commented out in the view controller template; this shows you the default behavior
 - ❖ this code will only return YES (true) for **UIInterfaceOrientationPortrait**
- DEMO** ❖ and thus the view stays in Portrait orientation

Say “yes” to changes in orientation

```
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)interfaceOrientation {  
    return YES;  
}
```

- In order to rotate to any orientation, we override the default behavior and return YES to any request
- This is an example of delegation. When we rotate the device, the application asks the current view controller “SHOULD we autorotate to this new orientation”
 - Our change above says “YES!”
 - For most apps, the autorotation will work just fine

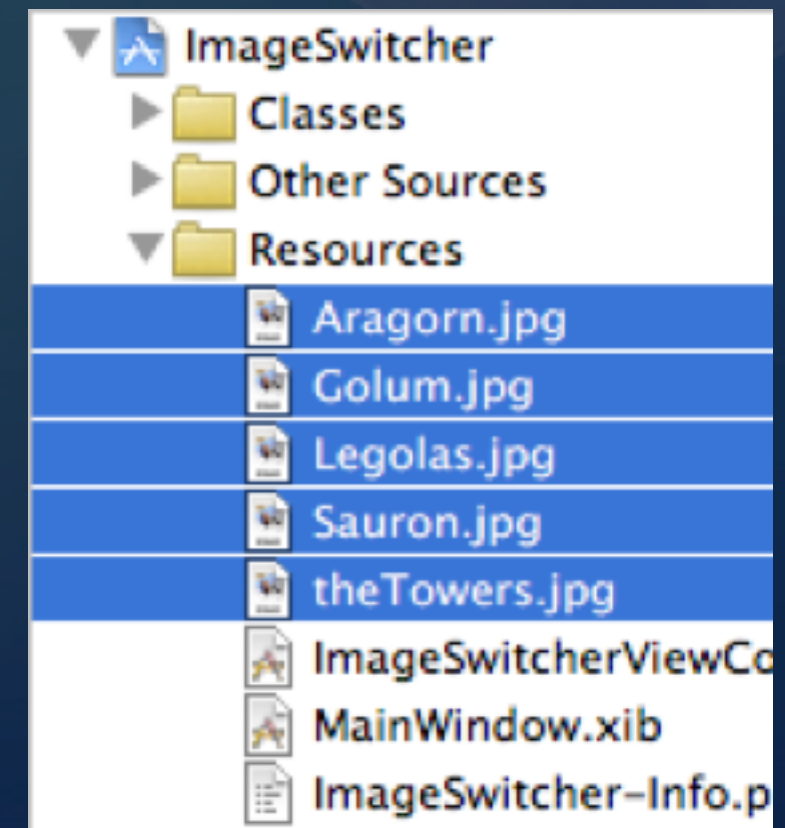
DEMO

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

Step 1: 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



Step 2: Edit View Controller's XIB File

- Edit **ImageSwitcherViewController.xib** in Interface Builder
 - 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”
 - Use the outline view of IB's window to select the views

Step 3: Add Outlets in .h file

- We need to add outlets and properties for the controls, plus add two variables that will help us manage the views

```
9  #import <UIKit/UIKit.h>
10
11  @interface ImageSwitcherViewController : UIViewController {
12      IBOutlet UIPageControl *pages;
13      IBOutlet UIImageView *one;
14      IBOutlet UIImageView *two;
15      UIImageView *front, *back;
16  }
17
18  @property (nonatomic, retain) UIPageControl *pages;
19  @property (nonatomic, retain) UIImageView *one;
20  @property (nonatomic, retain) UIImageView *two;
21
22  @end
23
```

Step 4: Wire up the controls

- Back in Interface Builder
 - connect the outlets to the various controls
 - UIImageView “one” should connect to the view with tag “0”
 - UIImageView “two” should connect to the view with tag “1”
- Save your work and head back to XCode

Step 5: Write the Code

- Don't forget to synthesize your properties!
- We then 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)

```
15 - (void)viewDidLoad {  
16     [super viewDidLoad];  
17  
18     [one setImage:[UIImage imageNamed:@"Aragorn.jpg"]];  
19  
20     front = one;  
21     back  = two;  
22  
23     [one setHidden:NO];  
24     [two setHidden:YES];  
25  
26     [pages addTarget:self  
27         action:@selector(pageTurning:)  
28         forControlEvents:UIControlEventValueChanged];  
29 }
```

viewDidLoad (III)

- Note the call (**addTarget:action:forControlEvents:**) that links up the page control with our method called **pageTurning:**
 - We used an Objective-C function to create a reference to the method
 - **@selector(<methodname>)**
 - creates a reference to the specified method signature that can then be resolved at run time
 - **@selector(pageTurning:)**
 - ensures that the page control can call the correct method

Step 6: 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
 - Swap the visibilities of the two image views
 - And update our pointers

```

30
31 - (void) pageTurning: (UIPageControl *) pager {
32     NSInteger nextPage = [pager currentPage];
33     switch (nextPage) {
34         case 0:
35         [back setImage:[UIImage imageNamed:@"Aragorn.jpg"]];
36         break;
37         case 1:
38         [back setImage:[UIImage imageNamed:@"Gollum.jpg"]];
39         break;
40         case 2:
41         [back setImage:[UIImage imageNamed:@"Legolas.jpg"]];
42         break;
43         case 3:
44         [back setImage:[UIImage imageNamed:@"Sauron.jpg"]];
45         break;
46         case 4:
47         [back setImage:[UIImage imageNamed:@"theTowers.jpg"]];
48         break;
49         default:
50         break;
51     }
52
53     [front setHidden:YES];
54     [back setHidden:NO];
55
56     if (front.tag == 0) {
57         front = two;
58         back = one;
59     } else {
60         front = one;
61         back = two;
62     }
63
64 }

```

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 two simple flip animations
 - One animation will flip the front image view out
 - The second will flip the back image in
- 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

Animation Template

- Animations on Views are started with a call to **beginAnimations:context:** which is a class method on **UIView**; Animations are committed with a call to **commitAnimations**, another class method on **UIView**
 - Calling this puts the animation request on a queue; the animation begins executing on a separate thread once the event handler returns
- In between, you configure the specific animation and make the change to the affected view
 - in this case, our changes are the calls to **setHidden:** on front and back

```
[UIView beginAnimations:@"flipping view" context:nil];  
[UIView setAnimationDuration:0.5];  
[UIView setAnimationCurve:UIViewAnimationCurveEaseInOut];  
[UIView setAnimationTransition:UIViewAnimationTransitionFlipFromLeft forView:front cache:YES];  
  
[front setHidden:YES];  
  
[UIView commitAnimations];  
  
[UIView beginAnimations:@"flipping view" context:nil];  
[UIView setAnimationDuration:0.5];  
[UIView setAnimationCurve:UIViewAnimationCurveEaseInOut];  
[UIView setAnimationTransition:UIViewAnimationTransitionFlipFromLeft forView:back cache:YES];  
  
[back setHidden:NO];  
  
[UIView commitAnimations];
```

We set up a flip transition for the front image view and ask that by the end of the transition, it be hidden from view

We add that animation to the queue and then we configure our second transition, another flip, on the back image view, asking that it be visible by the end

By specifying two Left transitions in a row, we get a nice flipping effect

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

Image Switcher Polished

- The animation example above is similar to what we saw in Android BUT
 - As of iOS 4.x the “beginAnimations/commitAnimations” methods are deprecated
 - I created a more polished version of image switcher that uses the new “block style” animation methods... plus
 - I added support for swiping and matching the direction of the animation with whether we were moving forward or backward through the images

DEMO

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 a button
 - The label will state which screen we are looking at
 - The button will take us to one of 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
- Select Cocoa Touch Class on the left and UIViewController subclass on the right
 - Make sure all checkboxes below are NOT clicked
- Click Next and name this file ScreenOne.m
 - Make sure the checkbox to create a header file IS clicked

Step 3: Create Label and Button

- At a high-level, we will
 - override the **viewDidLoad** method to
 - programmatically create a Screen One label and a Go To Screen Two button
 - 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

```
[goToScreenTwo addTarget:self action:@selector(handleClick:) forControlEvents:UIControlEventTouchUpInside];
```

- We need to import ScreenTwo.h in order to instantiate it
- Our method then locates the window via the application delegate and sets a new root view controller
 - Setting a new root view controller causes the previous one to be released and its views/subviews removed

handleClick: A lot going on in 2 lines

- Line one gets the application delegate

```
- (void) handleClick: (id) sender {  
    ViewSwitcherAppDelegate *delegate = (ViewSwitcherAppDelegate *)[UIApplication sharedApplication] delegate];  
    [[delegate window] setRootViewController:[[[ScreenOne alloc] initWithNibName:nil bundle:nil] autorelease]];  
}
```

- Line two
 - allocates a view controller (code from ScreenThree.m)
 - autoreleases it
 - gets the window from the delegate and sets the root view controller

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)

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)


```

- (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 **UITableView**

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 Due on Friday
- Homework 6 Assigned on Friday
- Lecture 19: Advanced Android
- Lecture 20: Advanced iOS