CS 47

Beginning iPhone Application Development

Week 3: Navigating Through View Controllers

Download This Keynote

- This keynote presentation is available on the class website under the Week 3 syllabus entry

http://cs47.fieldman.org

Agenda

- Navigation Controllers
- TabBar Controllers
- Modal Controllers
- Autoresizing Views

iPad

It's a big iPod Touch

Review

- MVC: Why do we need Controllers?
- They hold all of the application-specific logic, binding the views and model.
- Remember the basic rules:
  - Each screen performs one general task
  - One view controller per screen

Problem

- How do we navigate between screens (view controllers)?
  - UINavigationController
  - UITabBarController
  - presentModalViewController:animated:

Alternative

- You don't have to use these classes to navigation between screens
- You can implement your own transitions (fade, etc), but you need to create your own view controller management flow
- Scary

UINavigationController

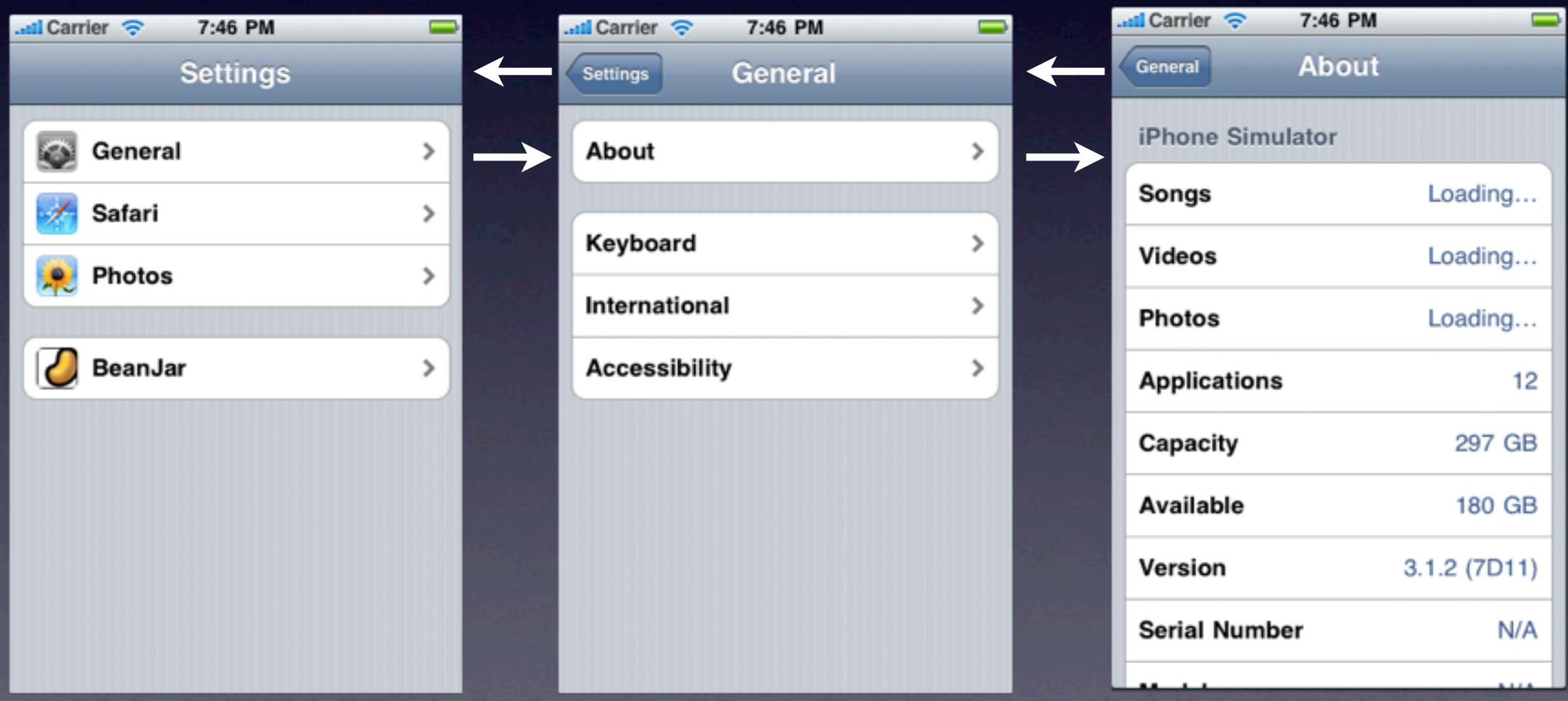
- Manages left/right screen swipe animation
- Manages navigation bar along the top
- Manages toolbar along the bottom
- Manages lifecycle of child view controllers

UINavigationController

- Uses a stack data structure to organize view controllers
- Push new views onto the stack
- Pop views off of the stack when you are done with them
- UINavigationController shows the user whichever controller is “on top”

UINavigationController

Typical flow:



UINavigationController

- In the previous example, each screen is a separate view controller
- When a new controller is pushed onto the navigation stack, it animates in from the right
- When a controller is popped from the navigation stack, it animates out to the right
- View controllers in the stack are retained by the navigation controller

UINavigationController

- The message to push/pop a view controller onto/from the navigation stack is usually called from the current view controller

- How does a view controller access the navigation controller that contains it?

```
UINavigationController *myNavController = self.navigationController;
```

- Will be nil if not joined to a nav controller

UINavigationController

- What do the push/pop calls look like?

```
UINavigationController *myNavController = self.navigationController;

/* Push new controller onto the stack */
UIViewController *vc = [[[UIViewController alloc] init] autorelease];
[myNavController pushViewController:vc animated:YES];

/* Pop controller off of the stack */
[myNavController popViewControllerAnimated:YES];

/* Pop to a certain controller */
UIViewController *someVC = [SomeViewControllerClass sharedInstance];
[myNavController popToViewController:someVC animated:YES];

/* Pop to the root view */
[myNavController popToRootViewControllerAnimated:YES];
```

UINavigationController

- Pressing the back button in the navigation bar performs an implicit `popViewControllerAnimated:`
- Or you can call this explicitly in response to an event

UINavigationController

- Uses the `navigationItem` property of the UIViewController to control how the navigation bar is rendered by the UINavigationController
- The navigationItem property is automatically allocated when accessed, e.g.:

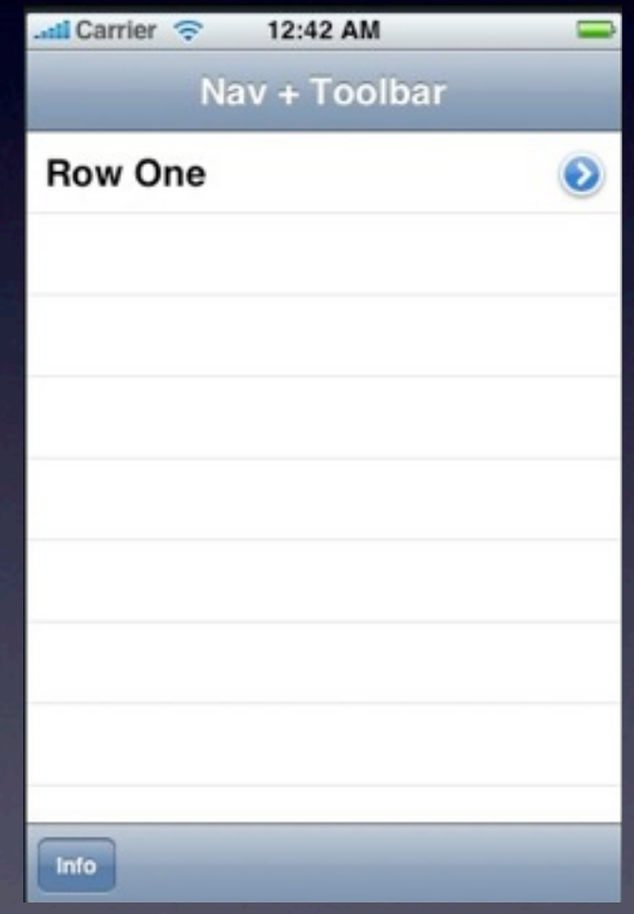
```
self.navigationItem.prompt = @”Hello”;
```

UINavigationController

- The `navigationItem` property controls: Title, prompt, left and right buttons, back buttons, and custom titleViews.
- You don't have to use the navigationItem member if you don't need special navigation behavior.
- At a minimum you only need to assign the `title` property of your view controller, and the UINavigationController will auto-generate the navigationItem.

UINavigationController

- Just like `navigationItem`, view controllers can assign an array to their `toolbarItems` member (an array of UIBarButtonItems)
- This will cause the UINavigationController to render a small toolbar at the bottom of the view



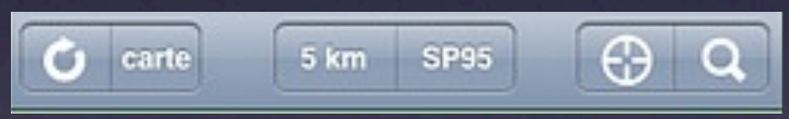

```
UIBarButtonItem *bbi = [[[UIBarButtonItem alloc]
                         initWithTitle:@”Info”
                                 style:UIBarButtonItemStyleBordered
                                target:self action:@selector(info:)]
                        autorelease];
self.toolBarItems = [NSArray arrayWithObject:bbi];
```

UINavigationController

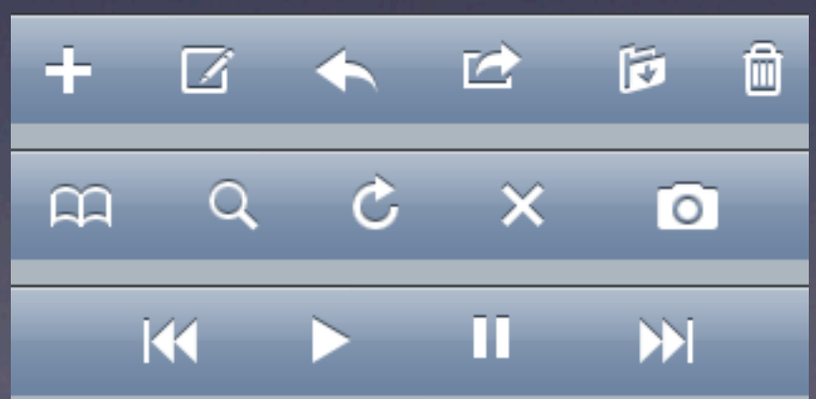
- What can a UIBarButtonItem look like?

Standard Toolbar Buttons

Custom Views



Standard System Icons



UINavigationController

- The navigation bar and the toolbar can be independently programmatically hidden or shown

```
self.navigationController.navigationBarHidden = YES;
self.navigationController.toolBarHidden       = YES;
```

- The space between the navigation bar and the toolbar is where the currently displayed view controller's `view` is rendered

UINavigationController

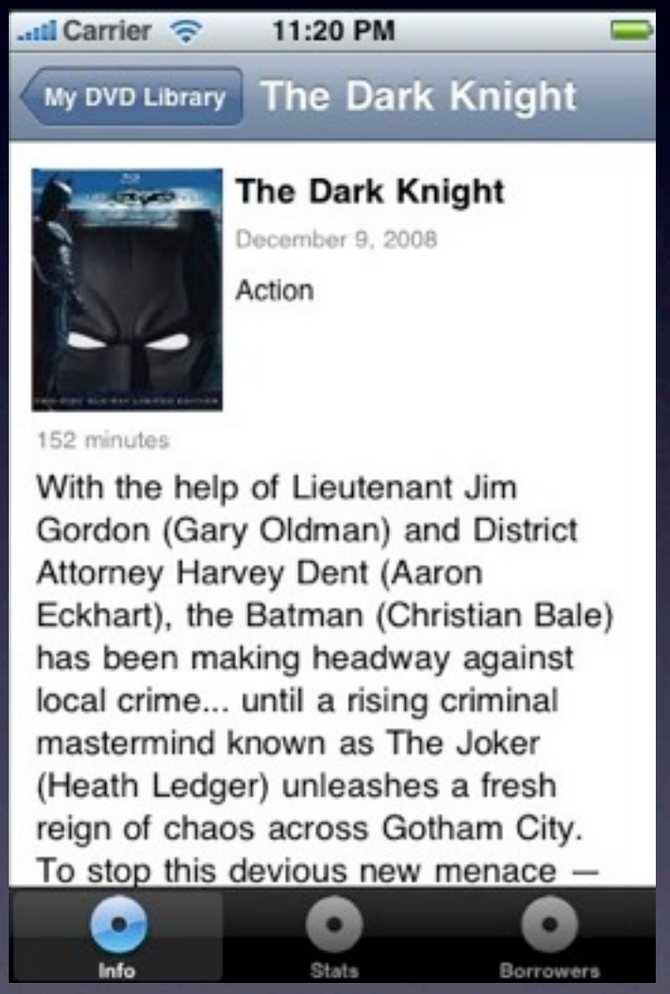
- Worth noting that there is a UINavigationControllerDelegate protocol
- UINavigationController informs its delegate when it will (and did) show a view controller

UITabBarController

- Manages transition between sibling views
- Manages tab bar along the bottom of the screen
- Manages lifecycle of child view controllers

UITabBarController

- Stores child controllers in a flat, sibling structure
- Tab bar along the bottom allows selection between children

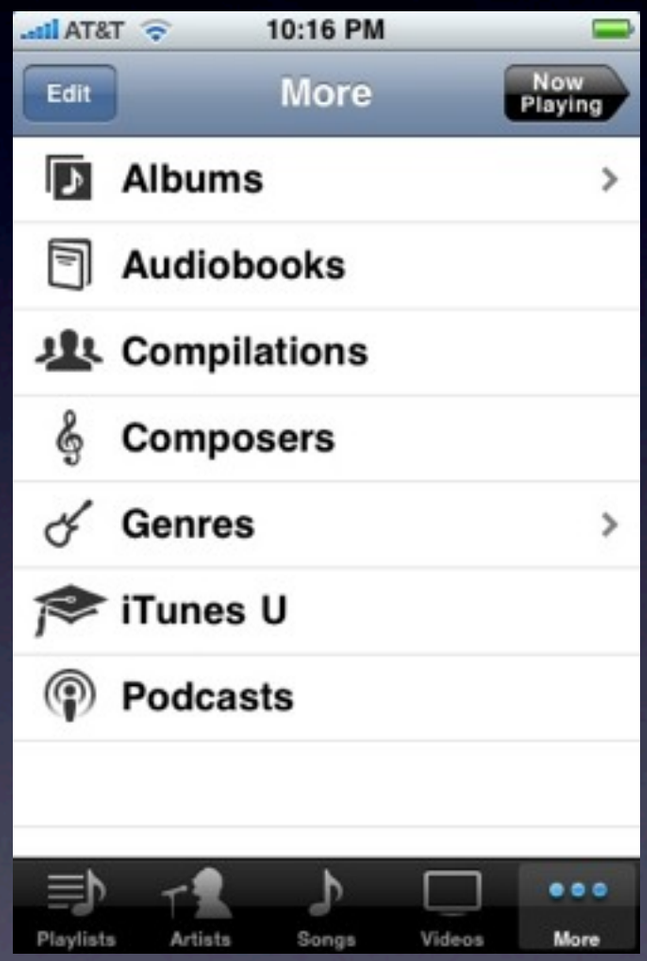


UITabBarController

- Each tab represents a single child view controller
- No animations when switching between views (immediate transition)
- Even though only one view controller is “visible” at a time, all child view controllers are retained and remain live in memory

UITabBarController

- Assign controllers by setting the `viewControllers` array
- Typical tab bars can support up to five tabs on the screen. If you need to manage more than five controllers, the tab bar will automatically show the "More..." tab
- You can allow the user to customize the tabs with the `customizableViewControllers` property (to reconfigure tab layout)



UITabBarController

- When you reassign the viewControllers property, or call `setViewControllers:animated:`, the tab bar controller will release all of the view controllers it was handling
- So it is important to note that you can't simply add a view controller to an existing tab bar... all of the view controllers are released, and then reassigned

UITabBarController

- How do you configure the tab buttons?
- Similar to the UINavigationController
- Every UIViewController has a `tabBarItem` variable that controls what its tab looks like
- You can initialize this value with a system value, or a custom title+image
- You can assign a red badge value to a tab

UITabBarController

- Rarely used, but useful to know: you can access the parent UITabBarController of a view controller through the `tabBarController` property

```
UITabBarController *tbc = self.tabBarController;
```

- Will be nil if the view controller is not part of a tab bar controller

UITabBarController

- The UITabBarController also has a delegate protocol to be notified when tabs are selected or customized

Modal View Controllers

- What if we just want to pop a view onto the screen outside of the standard navigation flow?

`presentModalViewController:animated:`

```
UIViewController *newVC = ...;
[self presentModalViewController:newVC animated:YES];
```

- The view controller is displayed above whatever else is on the screen

Modal View Controllers

- A view controller can access the parent that summoned it with

self.parentViewController

and dismiss itself with

```
[self.parentViewController dismissModalViewControllerAnimated:YES];
```

Modal View Controllers

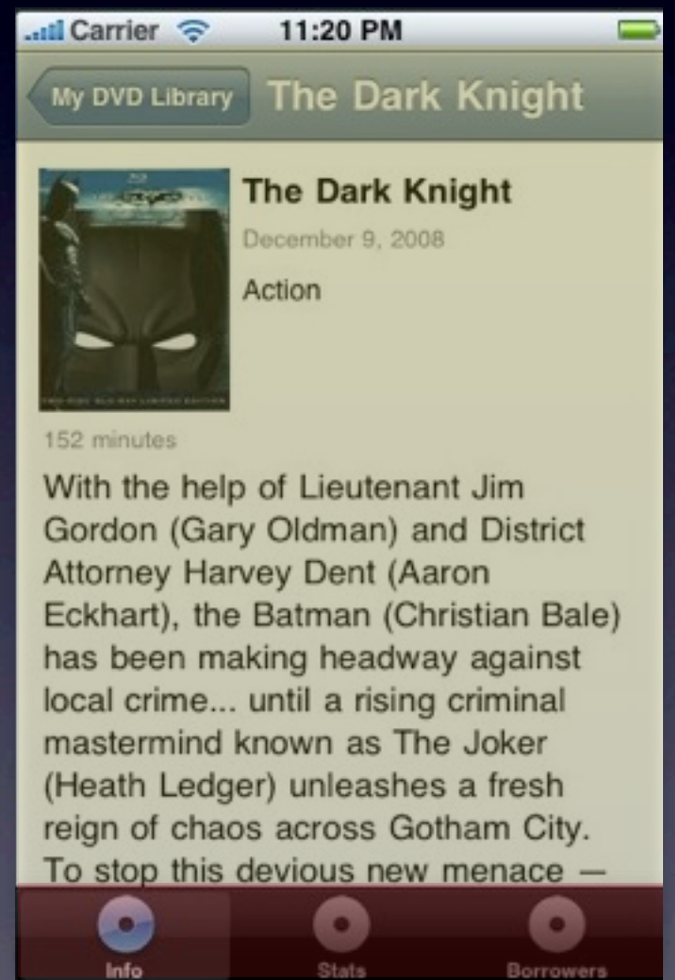
- Modal view controllers are summoned outside of the navigation flow, and are not bound by the current hierarchy's view bounds
- They take up the entire screen unless dismissed
- You can summon a UINavigationController or UITabBarController if you want

View Hierarchy

- Remember that UINavigationController and UITabBarController are just subclasses of the UIViewController class
- So you can embed tab bar controllers inside of navigation controllers, and vice-versa, just like any other view controller

View Hierarchy

- Remember: these view controller management subclasses split the screen into two areas:
  - The area reserved for their navigation/tab/toolbar views
  - The area remaining for the active child view controller



View Hierarchy

- Think about code reuse. We want our view controller subclasses to be flexible in how they are able to display in a view hierarchy
- Since we can embed these controllers to any level inside each other, it's feasible that we could have a very small space to display our actual content
- We also need to handle device rotation

Rotation

- UINavigationController and UITabBarController automatically support rotation for their implicit views (toolbar, tabbar, navbar)
- BUT: they will not rotate unless their current child UIViewController supports rotation

Rotation

- So how we support rotate? Override the `shouldAutorotateToInterfaceOrientation:` message

```
- (void) shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)toInterfaceOrientation
{
    return YES;
    // this allows any kind of rotation
    // optionally, return YES/NO based on the desired orientation
}
```

- Default implementation only returns YES for the standard portrait mode

Rotation

- You can also override certain messages to catch the fact that a rotation will or has occured

```
willAnimateRotationToInterfaceOrientation:duration:
willRotateToInterfaceOrientation:duration:
didRotateFromInterfaceOrientation:
```

Rotation

- When you rotate, the underlying coordinate system changes, along with the available space to display views
- e.g. a view that displayed in a 320x400 area in portrait mode would display in a 480x240 area in landscape mode

Resizing Views

- Ok, so how do we handle rotation, as well as the potential to display child view controllers in variable amounts of embedded navigation?
- Two ways: Automatic and manual

Resizing Views

- Automatic way: Autoresizing!
- We must enable the `autoresizesSubviews` property of the UIViewController's `view` properly

```
self.view.autoresizesSubviews = YES;
```

- Remember: all our view controller's content is under the `view` property, so this allows that top-level view to resize child elements

Resizing Views

- Set the `autoresizingMask` property of the child views you want to resize

```
myTableView.autoresizingMask = UIViewAutoresizingFlexibleWidth |
                               UIViewAutoresizingFlexibleHeight;
```

- Now myTableView will scale its width and height dynamically if its parent view changes size

Resizing Views

- Sometimes you have a complex UI that has many floating components
- You need to intercept the `willRotateToInterfaceOrientation:duration:` message and handle all of the child view restructuring manually