Stanford CS193p

Developing Applications for iPhone 4, iPod Touch, & iPad

Fall 2010

Today

- Finish Demo from last lecture
  - After a quick review of what we did and a little bit more on drawing
- What we've done so far
  - Created a custom `UIView` called `FaceView`
  - Set `FaceView` up to delegate its data (its smileyness) to some other object
- What we'll do today
  - Add our Model (an `int`!) to our Controller. Add a property to set our Model's value.
  - Add outlets for our FaceView and a `UISlider` and add an action for our `UISlider`
  - Implement our Controller
- Watch for how we ...
  - Use proper property syntax to manage the memory of our `IBOutlets`
  - Update our Model using a private property to protect its integrity and to keep our UI in sync
  - Use our custom view's `delegate` property

Today

- Under the hood of "View-based Application" template in Xcode
  - What is actually going on there?
- Application Lifecycle
  - Especially `application:didFinishLaunchingWithOptions:`
- View Controller Lifecycle
  - `initWithNibName:bundle:` vs `loadView`
  - View appearance and disappearance methods
  - Other methods
- Controllers of Controllers
  - `UINavigationController` in detail (others next week)
- Demo
  - Quick look at `HappinessAppDelegate.m`
  - Create a new "Window-based app" called Psychologist which will reuse the Happiness MVC
  - `UINavigationController`

View-based Application

- What files does this template in Xcode create for us?
  Assuming the name of our application is `Happiness` …
  `main.m`
  `HappinessViewController.[mh]`
  `HappinessViewController.xib`
  `MainWindow.xib`
  `Happiness-Info.plist`
  `HappinessAppDelegate.[mh]`
- `main.m`
  Basically just the C entry point function `int main(int argc, char *argv[])`
  Calls `UIApplicationMain` which creates a `UIApplication` object and starts the run loop
  Also creates a catch-all autorelease pool (we'll talk about autorelease pools in a few slides)
- `HappinessViewController.[mh]` and `.xib`
  You know what these are by now!

View-based Application

- What files does this template in Xcode create for us?

  Assuming the name of our application is `Happiness` …

  `main.m`

  `HappinessViewController.[mh]`

  `HappinessViewController.xib`

  `MainWindow.xib`

  `Happiness-Info.plist`

  `HappinessAppDelegate.[mh]`

- `MainWindow.xib`

  Contains a `UIWindow` (top of the view hierarchy) for things to be installed in
  Can be (usually is) customizable per platform (we'll talk about that next week).
  Contains the Application Delegate (just an `NSObject` with its class set to `HappinessAppDelegate`)
  Application Delegate also has a couple of outlets wired up (notably to `HappinessViewController`).

- `Happiness-Info.plist`

  A variety of application configuration properties. We'

View-based Application

- What files does this template in Xcode create for us?

  Assuming the name of our application is `Happiness` …

  `main.m`

  `HappinessViewController.[mh]`

  `HappinessViewController.xib`

  `MainWindow.xib`

  `Happiness-Info.plist`

  `HappinessAppDelegate.[mh]`

- `HappinessAppDelegate.[mh]`

  Has an instance variable for the `UIWindow` in `MainWindow.xib` called `window`.

  Has an instance variable for `HappinessViewController` called `viewController`.

  Has stubs for a lot of `applicationDid`This and `applicationWill`DoThat.

  Most importantly `application:didFinishLaunchingWithOptions:`.

  This is the method where `HappinessViewController`'s `view` is added to the `UIWindow`.

  Also where the `UIWindow` is made visible … `[window makeKeyAndVisible]`

  We will be modifying this method today to create a controller of controllers.

Application Delegate

- HappinessAppDelegate.h (header file for application delegate)

```
@interface HappinessAppDelegate : NSObject <UIApplicationDelegate>
{
    UIWindow *window;
    HappinessViewController *viewController;
}

@property (nonatomic, retain) IBOutlet UIWindow *window;
@property (nonatomic, retain) IBOutlet HappinessViewController *viewController;

@end
```

This object implements the `UIApplicationDelegate` protocol.
The `applicationDid`DoThis and `applicationWill`DoThat methods.

Application Delegate

- HappinessAppDelegate.h (header file for application delegate)

```
@interface HappinessAppDelegate : NSObject <UIApplicationDelegate>
{
    UIWindow *window;
    HappinessViewController *viewController;
}

@property (nonatomic, retain) IBOutlet UIWindow *window;
@property (nonatomic, retain) IBOutlet HappinessViewController *viewController;

@end
```

Instance variable/property/outlet pointing to the top-level view for this application (UIWindow). This is hooked up in MainWindow.xib.

Application Delegate

- HappinessAppDelegate.h (header file for application delegate)

```
@interface HappinessAppDelegate : NSObject <UIApplicationDelegate>
{
    UIWindow *window;
    HappinessViewController *viewController;
}

@property (nonatomic, retain) IBOutlet UIWindow *window;
@property (nonatomic, retain) IBOutlet HappinessViewController *viewController;

@end
```

Instance variable/property/outlet pointing to HappinessViewController (our Controller).
Also hooked up in MainWindow.xib.

Application Delegate

- Method called in the application delegate when ready to run

```
- (BOOL)application:(UIApplication *)application
        didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // Override point for customization after application launch.

    // Add the view controller's view to the window and display.
    [window addSubview:viewController.view];
    [window makeKeyAndVisible];

    return YES;
}
```

This instance variable points to the instance of our Controller HappinessViewController

Application Delegate

- Method called in the application delegate when ready to run

```
- (BOOL)application:(UIApplication *)application
        didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // Override point for customization after application launch.

    // Add the view controller's view to the window and display.
    [window addSubview:viewController.view];
    [window makeKeyAndVisible];

    return YES;
}
```

Note the `view` property in `UIViewController`.
This is the top-level of the view hierarchy in its `.xib` file.
(which is `HappinessViewController.xib` in this case)

Application Delegate

- Method called in the application delegate when ready to run

```
- (BOOL)application:(UIApplication *)application
        didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // Override point for customization after application launch.

    // Add the view controller's view to the window and display.
    [window addSubview:viewController.view];
    [window makeKeyAndVisible];

    return YES;
}
```

We simply add that `view` as a `subview` of the top-level `UIWindow`.

Application Lifecycle

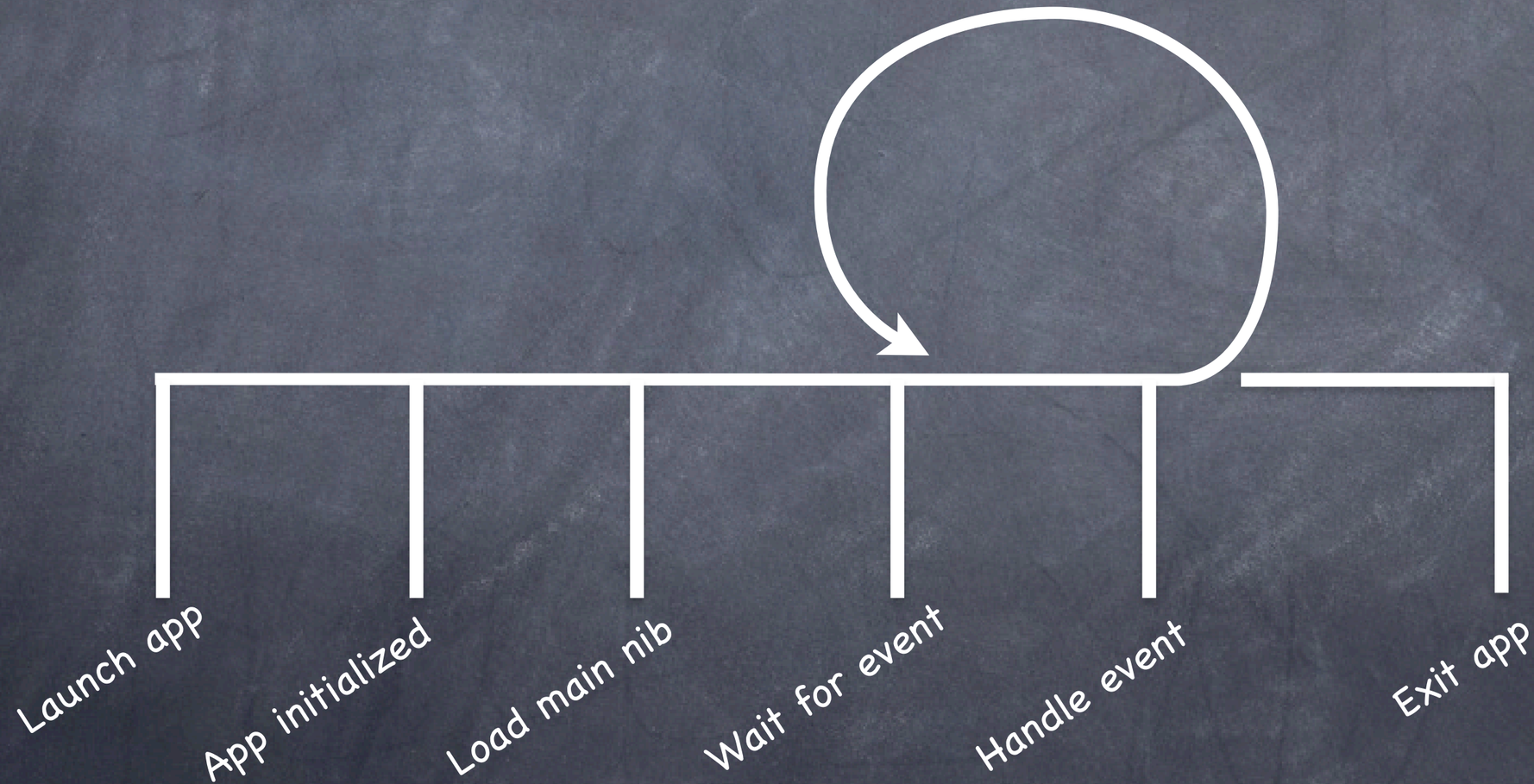
- After `application:didFinishLaunchingWithOptions:`, then what?

  Application enters a "run loop" repeatedly doing the following …

  An `autorelease` pool is created (more on this in a moment)
  Application waits for events (touch, timed event, I/O, etc.)
  Events are dispatched through `UIKit` objects and often on to your objects (via `delegates`, etc.)
  When all is done, the screen is updated (appropriate `drawRect:` methods are called)
  The `autorelease` pool is drained

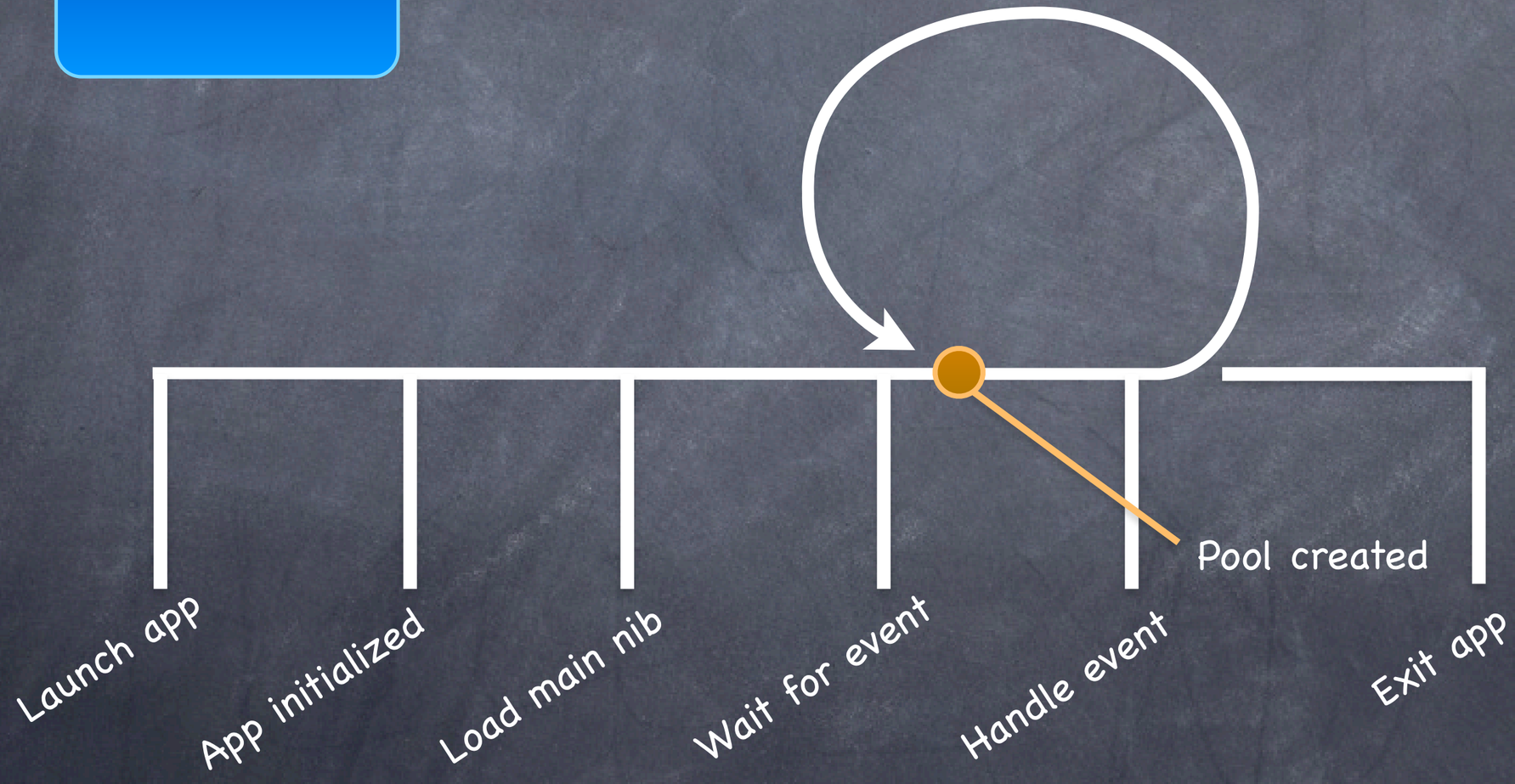
  Rinse, repeat.

Autorelease Pools



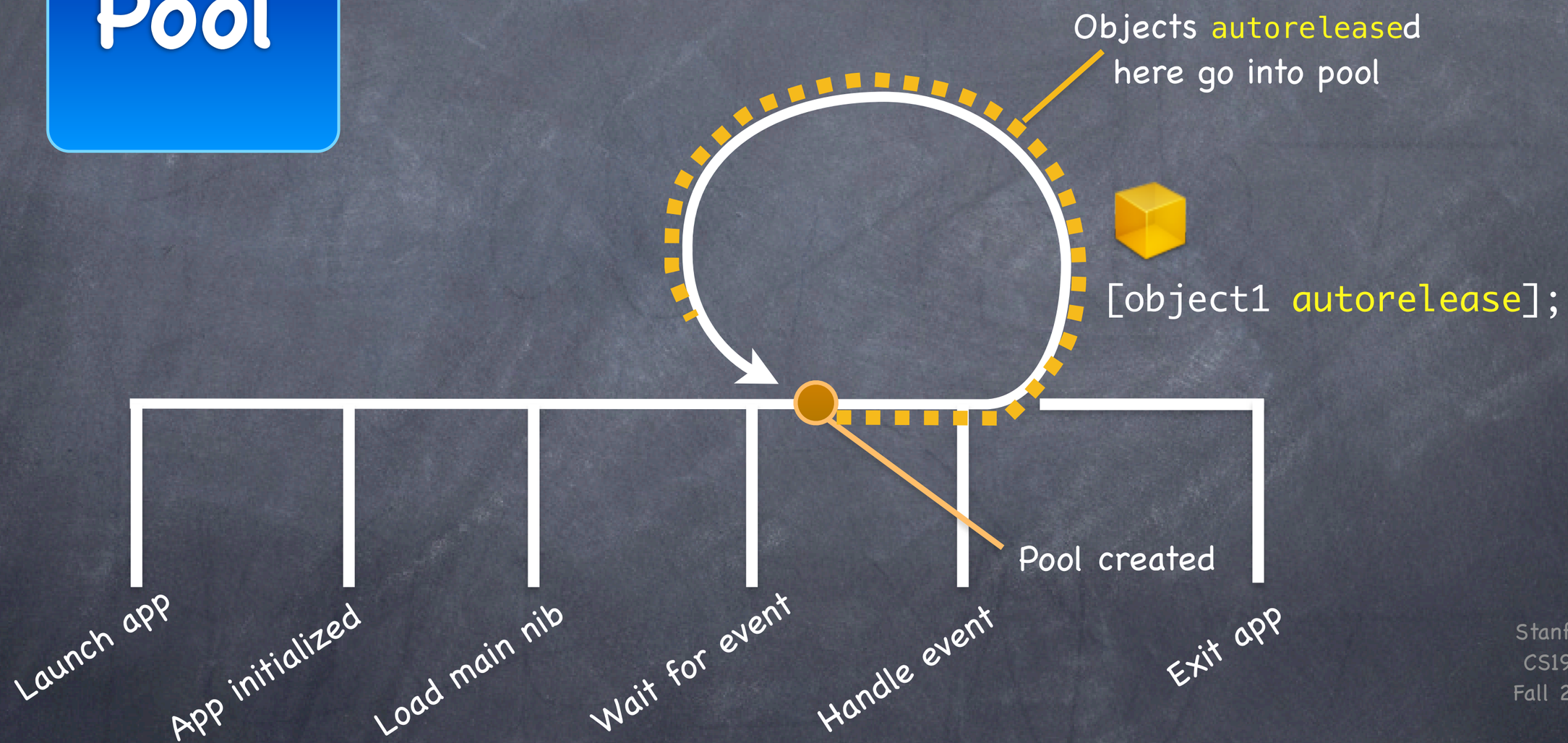
Autorelease Pools

Pool



Autorelease Pools

Pool



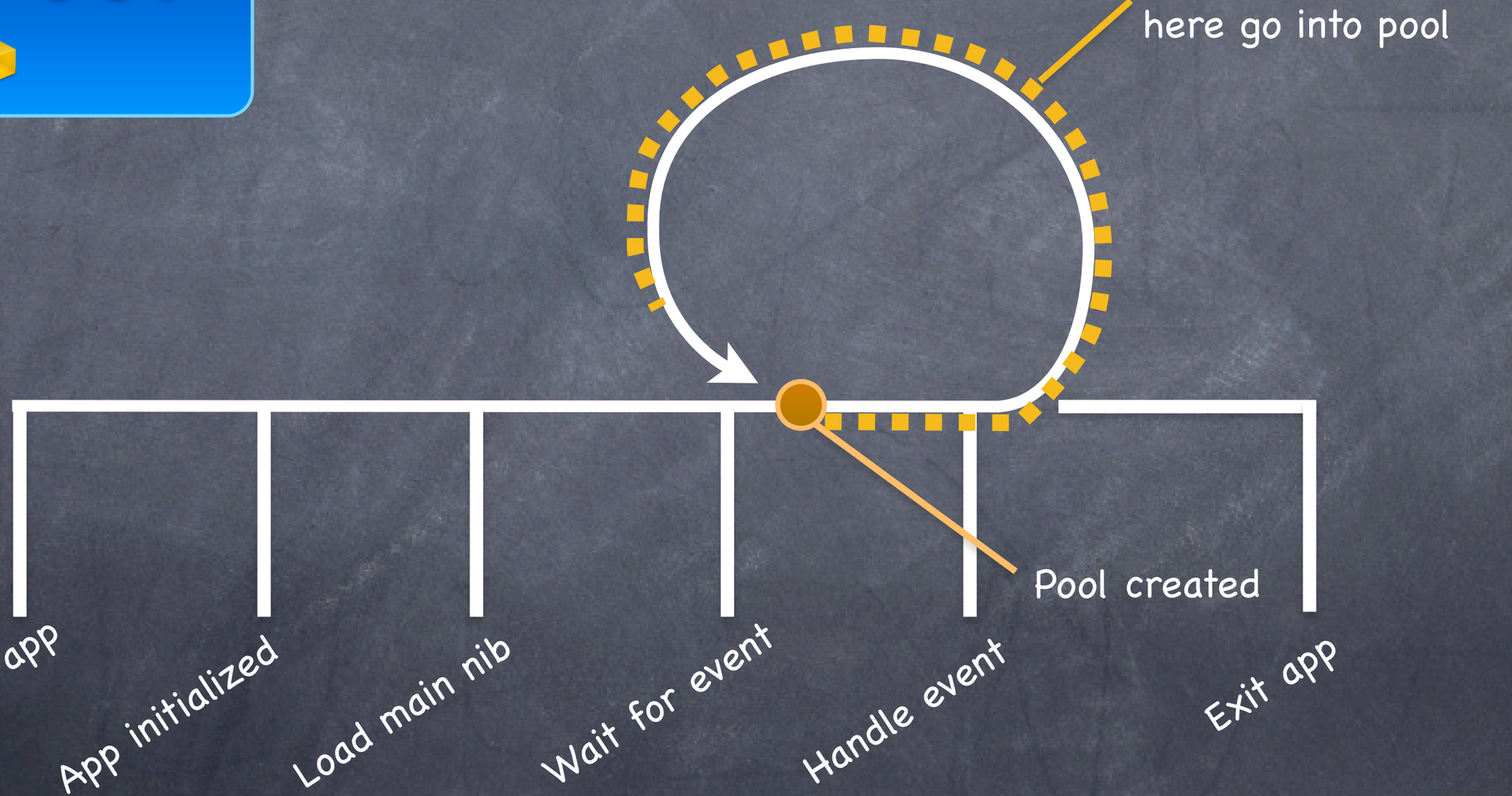
Autorelease Pools

Autorelease Pools

Autorelease Pools

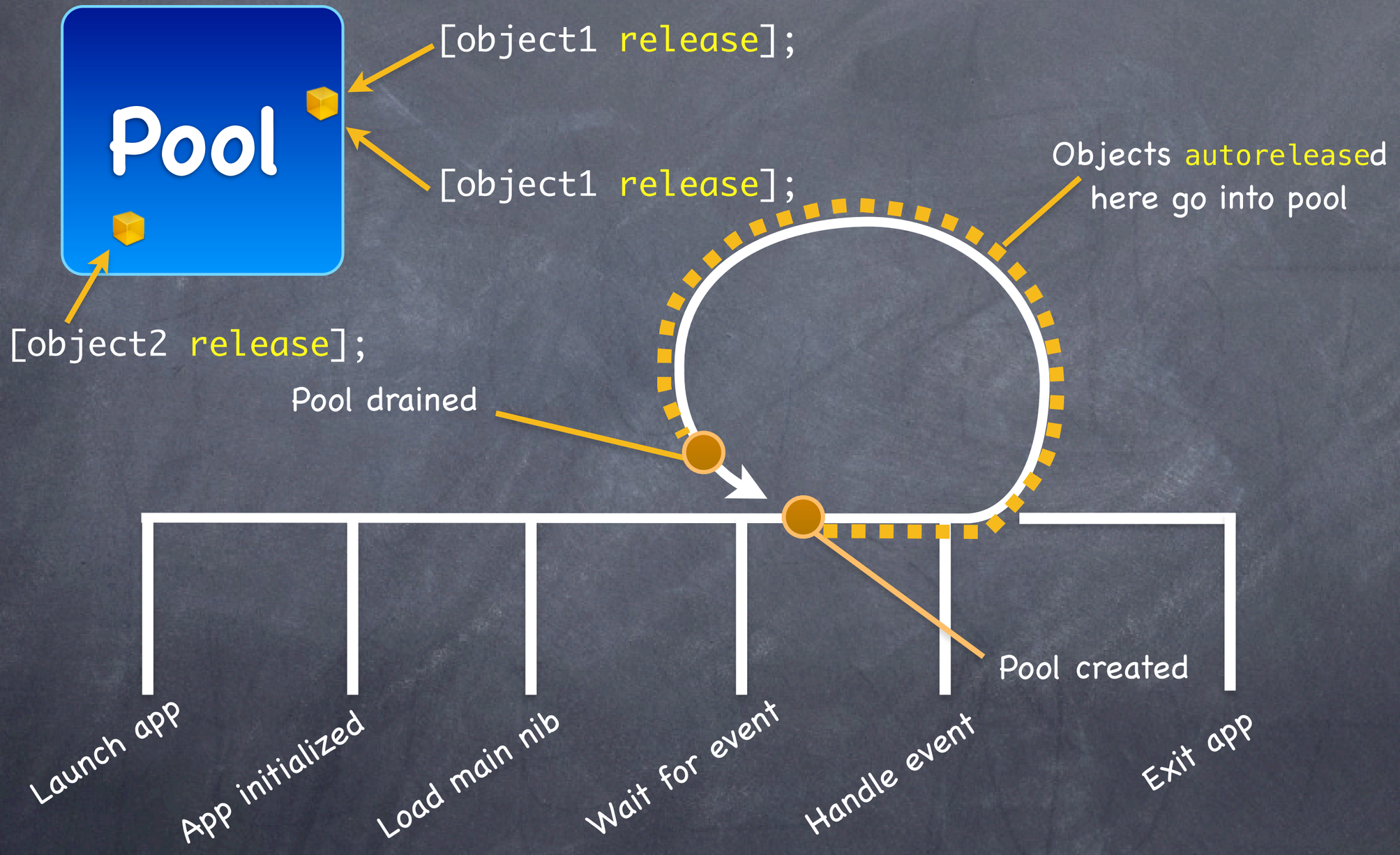
Autorelease Pools


```
[object1 autorelease];
(again)
```



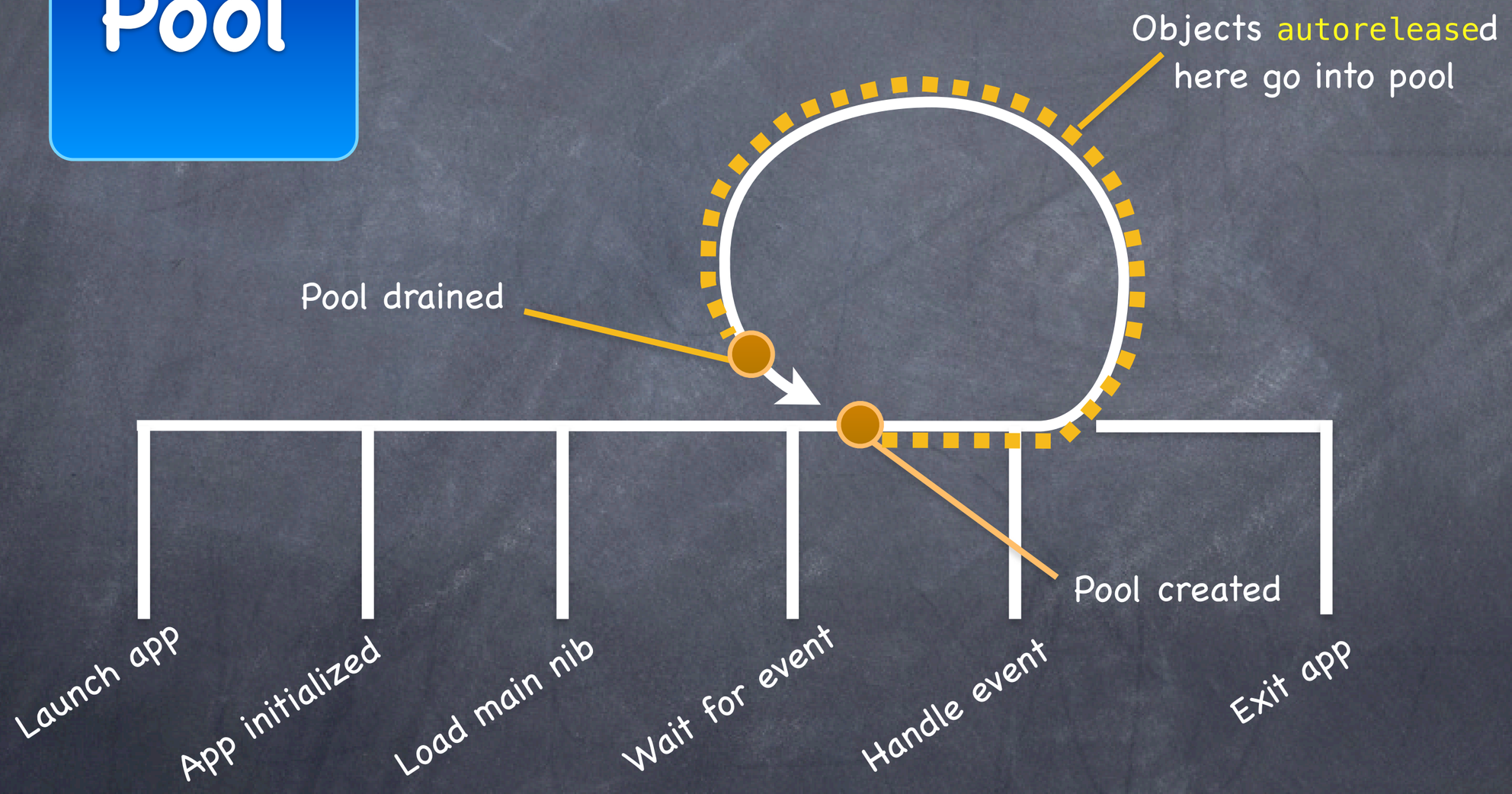
Autorelease Pools

Autorelease Pools



Autorelease Pools

Pool



Window-based Application

- What is the Window-based application template in Xcode?

```
@interface PsychologistAppDelegate : NSObject <UIApplicationDelegate> {
    UIWindow *window;
}
@property (nonatomic, retain) IBOutlet UIWindow *window;
@end


- (BOOL)application:(UIApplication *)application
        didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // Override point for customization after application launch.
    [window makeKeyAndVisible];
    return YES;
}
```

We have to create our own view controller(s) here.
And then add a controller's view to the window's view hierarchy.

Application Delegate

- Many methods in the `UIApplication` object's delegate protocol
  - `- (void)application:didFinishLaunchingWithOptions:(NSDictionary *)launchOptions;`
  - `- (void)applicationWillResignActive:`
  - `- (void)applicationDidBecomeActive:`
  - `- (void)applicationDidEnterBackground:`
  - `- (void)applicationWillEnterForeground:`
  - `- (BOOL)application:handleOpenURL:(NSURL *)url;`
  - `- (void)applicationDidReceiveMemoryWarning:`
  - `- (void)application:didReceiveLocalNotification:(UILocalNotification *)notification;`
  - `- (void)application:didReceiveRemoteNotification:(NSDictionary *)userInfo;`
  - `- (void)applicationWillTerminate:`
- We'll cover some of these as the quarter progresses

View Controller

- You've probably got a pretty good handle on the basics of this
  Class is `UIViewController`. It's your Controller in an MVC grouping.
- VERY important property in `UIViewController`
  `@property (retain) UIView *view;`
  This is a pointer to the top-level `UIView` in the Controller's View (in MVC terms)
- View Controllers have a "lifecycle" from creation to destruction
  Your subclass gets opportunities to participate in that lifecycle by overriding methods

View Controller

- The lifecycle starts with `alloc` and initialization of course

  `- (id)initWithNibName:(NSString *)nibName bundle:(NSBundle *)aBundle;`

  This is `UIViewController`'s designated initializer.

  The `UIViewController` tries to get its `view` from the specified `.xib` file called `nibName`.
  If `nibName` is `nil`, it uses the name of the class as the `nibName` (`HappinessViewController.xib`).

  The bundle allows you to specify one of a number of different `.xib` files (localization).
  We'll cover `NSBundle` later in the course when we talk about localization.
  Passing `nil` for `aBundle` basically means "look in the `Resources` folder from Xcode."

  Initializing `UIViewController` with `init` is very common, it means `nibName` is `nil` & `aBundle` is `nil`.

- Can I build a `UIViewController`'s `view` in code (i.e. w/o a `.xib`)?

  Yes.

  If no `.xib` is found using mechanism above, `UIViewController` will call `- (void)loadView` on itself.
  `loadView`'s implementation MUST set the `view` property in the `UIViewController`.
  Don't implement `loadView` AND specify a `.xib` file (it's undefined what this would mean).

View Controller

- After the `UIViewController` is initialized, `viewDidLoad` is called

  `- (void)viewDidLoad;`

  We learned about his in the last lecture.
  This is an exceptionally good place to put a lot of setup code.
  But be careful because the geometry of your view (its `bounds`) is not set yet.
  If you need to initialize something based on the geometry of the view, use the next method …

- Just before the view appears on screen, you get notified

  `- (void)viewWillAppear:(BOOL)animated;`

  When this is called, your `bounds` has been set (via your `frame` by your `superview` or some such).
  Your `view` will probably only get "loaded" once, but it might appear and disappear a lot.
  So don't put something in this method that really wants to be in `viewDidLoad`.
  Otherwise, you might be doing something over and over unnecessarily.
  Use this to optimize performance by waiting until this method (i.e. just before view appears)
  to kick off an expensive operation (might have to put up a spinning "loading" icon though).

View Controller

- And you get notified when you will disappear off screen too

```
- (void)viewWillDisappear:(BOOL)animated
{
    [super viewWillDisappear:animated];  // call this in all the viewWill/Did methods
    // let's be nice to the user and remember the scroll position they were at ...
    [self rememberScrollPosition];  // we'll have to implement this
    // do some other clean up now that we've been removed from the screen
    [self saveDataToPermanentStore];
    // but be careful not to do anything time-consuming here, or app will be sluggish
    // maybe even kick off a thread to do what needs doing here
}
```

- There are "did" versions of both of these methods too

```
- (void)viewDidAppear:(BOOL)animated;
- (void)viewDidDisappear:(BOOL)animated;
```

View Controller

- You already know about `viewDidUnload`

  Called in low-memory situations.

  Be sure to release your outlets (or other data tied to the `view` and its `subviews`) here.

  ```
  - (void)viewDidUnload;
  ```

View Controller

- Reacting to device rotation

```
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)anOrientation
{
    return (anOrientation == UIInterfaceOrientationPortrait) ||
           (anOrientation == UIInterfaceOrientationPortraitUpsideDown);
}
```

The default is to only allow `UIInterfaceOrientationPortrait`.

This `UIViewController`'s `view` is allowed to flip around if the device is turned upside down.

There is also `UIInterfaceOrientationLandscapeLeft` and `Right`.

It is certainly nice to return `YES` from this method for as many as possible orientations.

But make sure that your `view` can draw itself "wide and not-tall" as well as "tall and not-wide" if you are going to return `YES` for the landscape orientations.

View Controller

- When rotation actually happens

```
- (void)willRotateToInterfaceOrientation:(UIInterfaceOrientation)anOrientation
                                duration:(NSTimeInterval)seconds;

- (void)didRotateFromInterfaceOrientation:(UIInterfaceOrientation)anOrientation;

@property UIInterfaceOrientation interfaceOrientation;
```

The property will have the current orientation when each of the above is called.

Stop doing anything expensive (e.g. an animation maybe?) in `will` and resume it in `did`.

The best way to handle rotations is to design your `view` to layout its `subviews` properly (i.e. set their `frame`s) no matter what the aspect ratio of the `view` is.

Interface Builder can let you set "struts and springs" to help with layout flexibility.

Or the `UIView` method `layoutSubviews` can be overridden to do this (outside this course's scope).

Check out how the Apple-provided Calculator app reacts to landscape!

Controller of Controllers

- Special View Controllers that manage a collection of other MVCs
- `UINavigationController`
  Manages a hierarchical flow of MVCs and presents them like a "stack of cards"
  Very, very, very commonly used on the iPhone
- `UITabBarController`
  Manages a group of independent MVCs selected using tabs on the bottom of the screen
- `UISplitViewController`
  Side-by-side, master->detail arrangement of two MVCs
  iPad only

UINavigationController

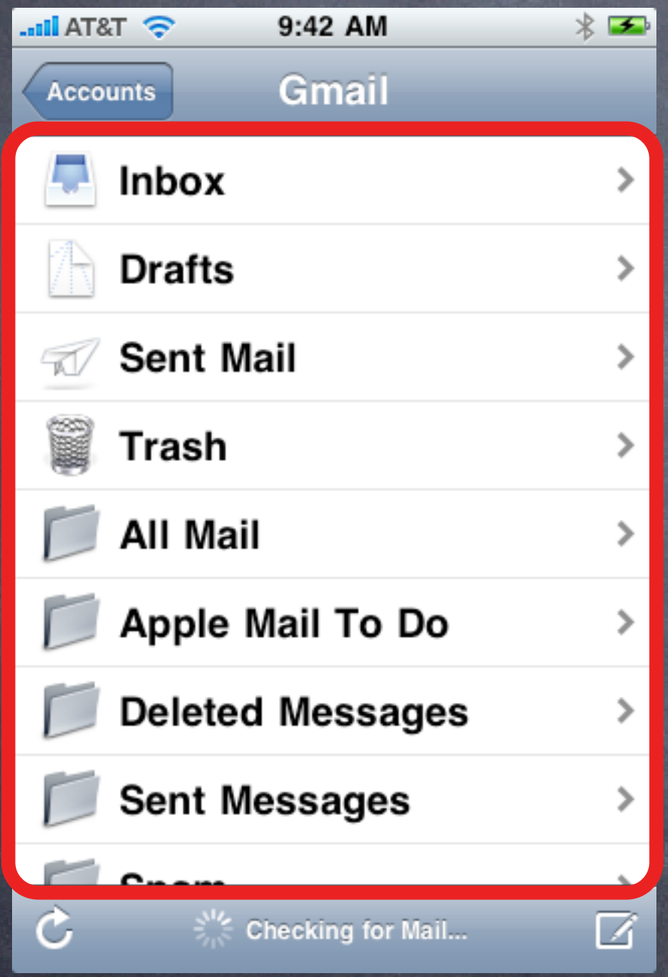
- Create with alloc/init
- Put the UINavigationController's view on screen

  Remember that UINavigationController is itself a UIViewController, so it has a view property.
  We simply call addSubview: to add the navigation controller's view to the view hierarchy.
  We almost always do this to the window in application:didFinishLaunchingWithOptions:.
  But we might do it in other places on an iPad where there's more screen real estate.
  E.g., we might put a UINavigationController into a UISplitView (next week).
  Then we'll have a controller of controllers inside a controller of controllers!

```
- (BOOL)application:(UIApplication *)application
        didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    UINavigationController *myNavController = [[UINavigationController alloc] init];
    [window addSubview:myNavController.view];
    [window makeKeyAndVisible];
    return YES;
}
```

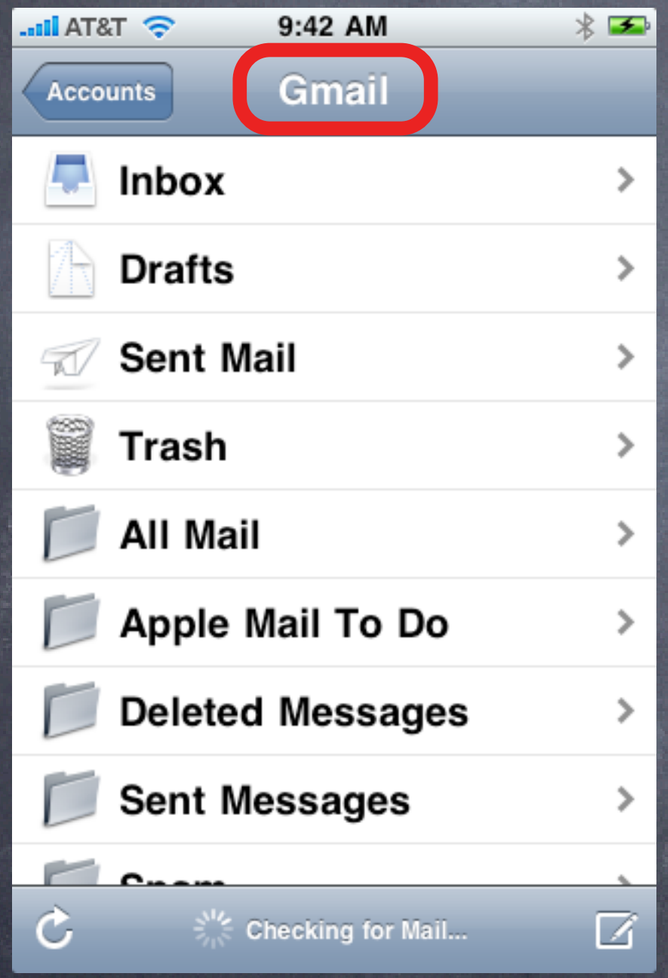
Something missing here

UINavigationController



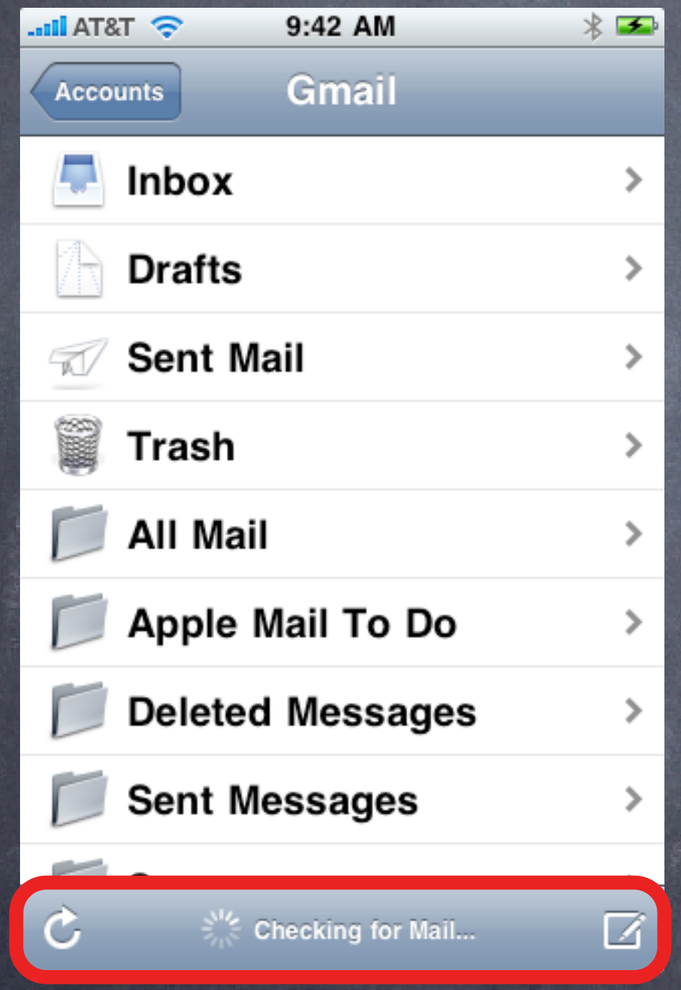
- `UIView` obtained from the `view` property of the `UIViewController` which is on top of the "stack of cards"

UINavigationController



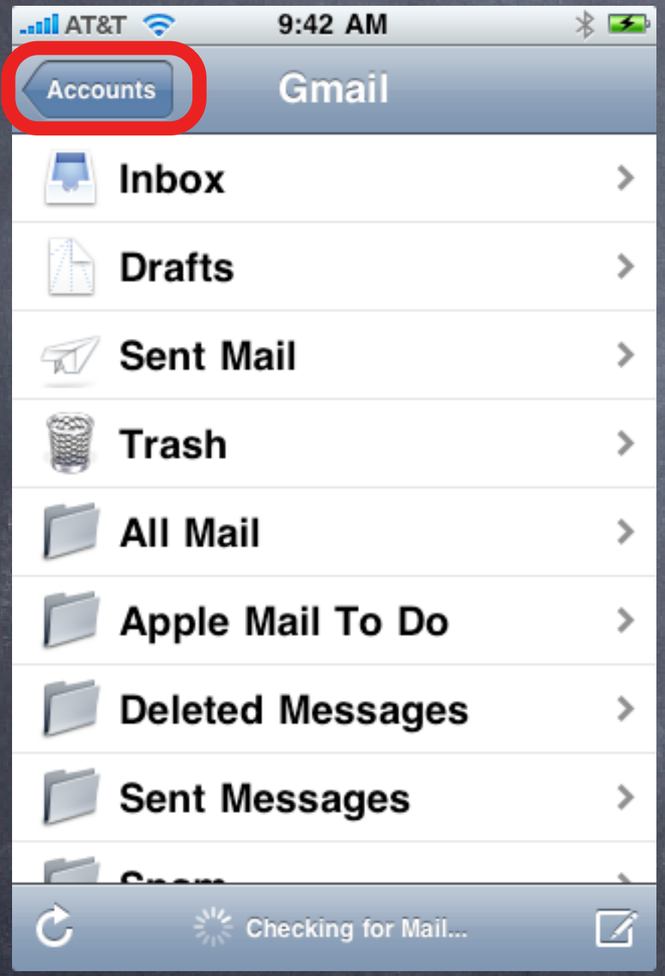
- `UIView` obtained from the `view` property of the `UIViewController` which is on top of the "stack of cards"
- `NSString` obtained from the `title` property of the `UIViewController` which is on top of the "stack of cards"

UINavigationController



- `UIView` obtained from the `view` property of the `UIViewController` which is on top of the "stack of cards"
- `NSString` obtained from the `title` property of the `UIViewController` which is on top of the "stack of cards"
- An `NSArray` of `UIBarButtonItems` obtained from the `toolbarItems` property of the `UIViewController` which is on top of the "stack of cards"

UINavigationController



- `UIView` obtained from the `view` property of the `UIViewController` which is on top of the "stack of cards"
- `NSString` obtained from the `title` property of the `UIViewController` which is on top of the "stack of cards"
- An `NSArray` of `UIBarButtonItems` obtained from the `toolbarItems` property of the `UIViewController` which is on top of the "stack of cards"
- `NSString` obtained from the `title` property of the next `UIViewController` down in the stack of cards. It is being displayed on a button provided by the navigation controller which, when touched, will cause the next `UIViewController` down in the stack of cards to move to the top of the stack (i.e. become visible). This is a "back" button.

UINavigationController

- How do we "push" a UIViewController onto the "stack of cards"?

```
- (void)pushViewController:(UIViewController *)vc animated:(BOOL)animated;
```

vc's view will appear on-screen inside the middle area of the navigation controller's UI.
Note that since the navigation controller has some UI of its own, vc's view will be squished.
So you must make sure that the "springs and struts" of vc's view are set properly.
This is what was "missing" from application:didFinishLaunchingWithOptions: two slides ago
We want to push one to get started before we put the UINavigationController's view on screen

- UIViewControllers know the UINavigationController they're in

So it's easy to push the next one on the stack from the one currently on the stack

```
- (IBAction)someAction:(UIButton *)sender
{
    UIViewController *vcToPush = ...;
    [self.navigationController pushViewController:vcToPush animated:YES];
}
```

This property in UIViewController returns the UINavigationController it is in (if any, else nil).

animated: is YES except the very 1st push before the UINavigationController is on screen

UINavigationController

- When does a pushed MVC come off the stack?

  Usually because the user presses the "back" button (shown on a previous slide).
  But it can happen programmatically as well with this `UINavigationController` instance method

  ```
  - (void)popViewControllerAnimated:(BOOL)animated;
  ```

  This does the same thing as clicking the back button.
  Somewhat rare to call this method. Usually we want the user in control of navigating the stack.
  But you might do it if some action the user takes in a view makes it irrelevant to be on screen.

- Example

  Let's say we push an MVC which displays a database record and has a delete button w/this action:

  ```
  - (IBAction)deleteCurrentRecord:(UIButton *)sender
  {
      // delete the record we are displaying
      // we just deleted the record we are displaying!
      // so it does not make sense to be on screen anymore, so pop
      [self.navigationController popViewControllerAnimated:YES];
  }
  ```

UINavigationController

- How you pass data when you push is important

  Do NOT use global variables (your AppDelegate is basically a global variable, by the way).
  Do NOT let the pushed Controller have a pointer back to the pushing Controller (violates MVC).
  Think of the pushed MVC construction as a "View" (in the MVC sense) of the pusher Controller.
  Actually, probably even more strict. Set up the pushee and let it do its thing on its own.
  If you absolutely must talk back to the pushee, use delegation (pusher sets itself as delegate)

- Example

```
- (IBAction)someAction:(UIButton *)sender
{
    // this action is going to cause another MVC's Controller to get pushed
    PusheeViewController *pushee = [[PusheeViewController alloc] init];
    pushee.someProperty = self.someValueThePusherKnows;
    pushee.someOtherProperty = self.someOtherValueThePusherKnows;
    pushee.delegate = self;  // self is the pusher and it implements the right protocol
    [self.navigationController pushViewController:pushee animated:YES];
    // now we're done (we'll be pushed off screen) until we get popped back
    // in the meantime, perhaps we will be informed of something as pushee's delegate
}
```

Next Time

- iPad
  How to take advantage of its platform-specific features.
- Universal Applications
  How to write a single app that will run on all platforms.
- Gestures
  Handling touch input (swipes, pans, pinches, etc.)

Demo

- HappinessAppDelegate.m
  We'll take a quick look at what the View-based Application made for us.
- New application: Psychologist
  Asks questions then comes up with a diagnosis.
  We'll create it using Window-based Application (i.e. Xcode will not create a Controller for us).
- A new UIViewController which will push the Happiness MVC
  Thus we'll reuse the Happiness MVC, including FaceView
- Create a UINavigationController & add it to our view hierarchy
  We'll do this in PsychologistAppDelegate.m's application:didFinishLaunchingWithOptions:
- Watch for ...
  Window-based Application instead of View-based Application
  Construct application's appearance in application:didFinishLaunchingWithOptions: method
  Properly initializing, then pushing a separate MVC construction onto the screen