Stanford CS193p

Developing Applications for iOS
Winter 2015

Today

- Application Lifecycle
  - Notifications
  - AppDelegate
  - Info.plist
  - Capabilities
  - AirDrop Demo
- Core Motion
  - Gyro, Accelerometer, etc.
  - Bouncer Demo

NSNotification

- Notifications

  The "radio station" from the MVC slides. For Model (or global) to Controller communication.

- NSNotificationCenter

  Get the default "notification center" via `NSNotificationCenter.defaultCenter()`

  Then send it the following message if you want to "listen to a radio station" ...

```
func addObserverForName(String, // the name of the radio station
                  object: AnyObject?, // the broadcaster (or nil for "anyone")
                   queue: NSOperationQueue?) // the queue to execute the closure on
{ (notification: NSNotification) -> Void in
    let info: [NSObject:AnyObject]? = notification.userInfo
    // info is a dictionary of notification-specific information
}
```

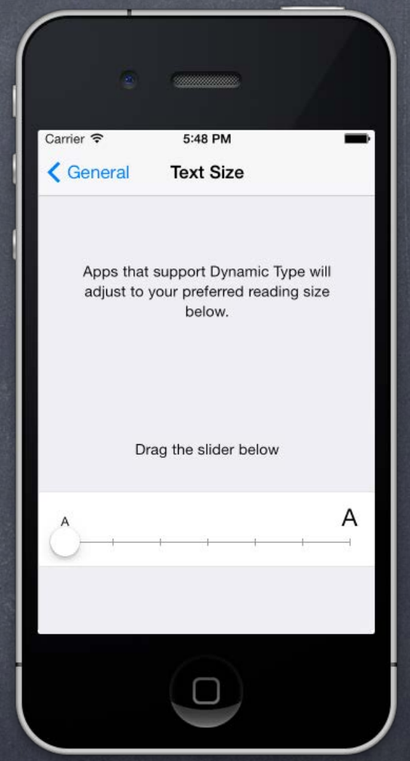
NSNotification

- Example

  Watching for changes in the size of preferred fonts (user can change this in Settings) …

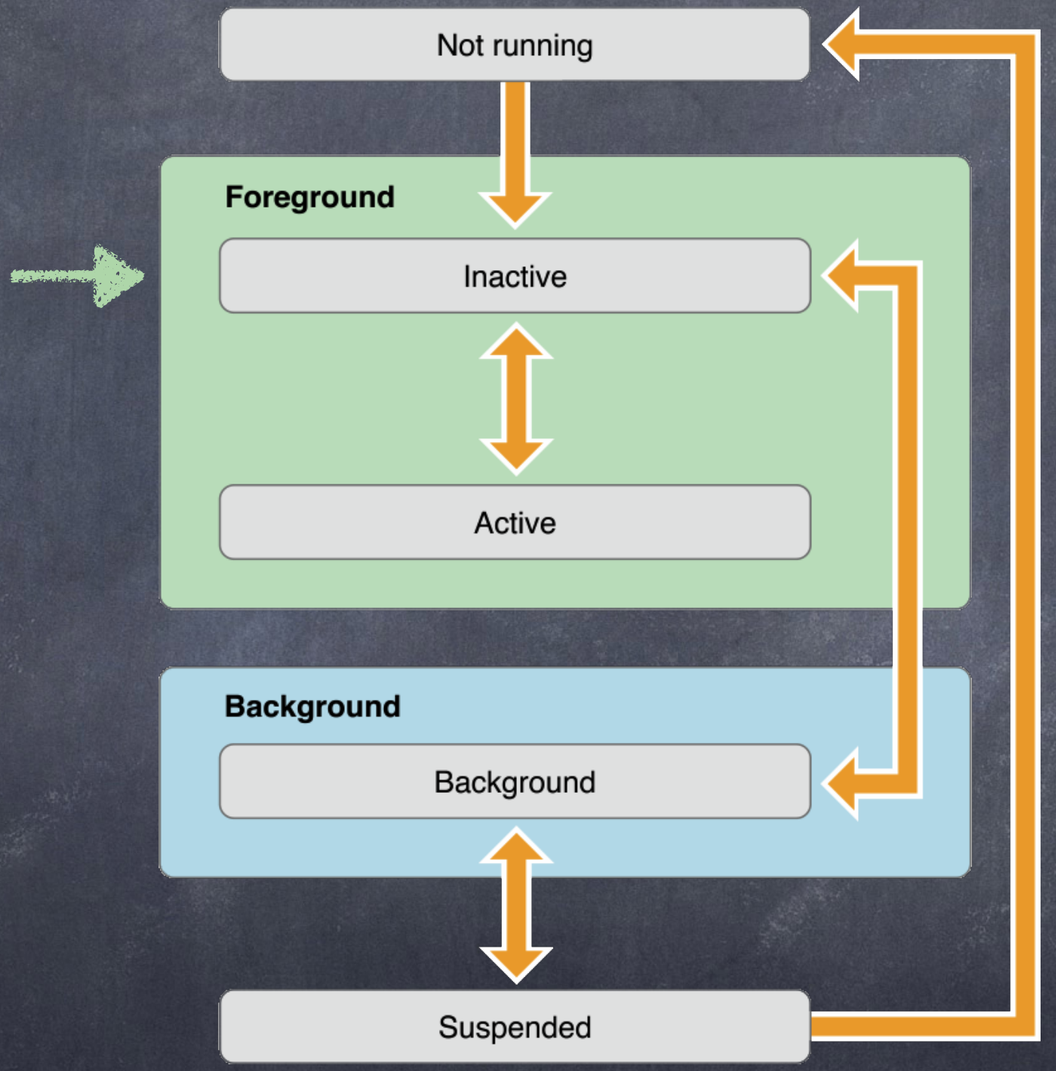
```
let center = NSNotificationCenter.defaultCenter()

center.addObserverForName(UIContentSizeCategoryDidChangeNotification
                  object: UIApplication.sharedApplication(),
                   queue: NSOperationQueue.mainQueue())
{ notification in
    // re-set the fonts of objects using preferred fonts
    // or look at the size category and do something with it …
    let c = notification.userInfo?[UIContentSizeCategoryNewValueKey]
    // c might be UIContentSizeCategorySmall, for example
}
```



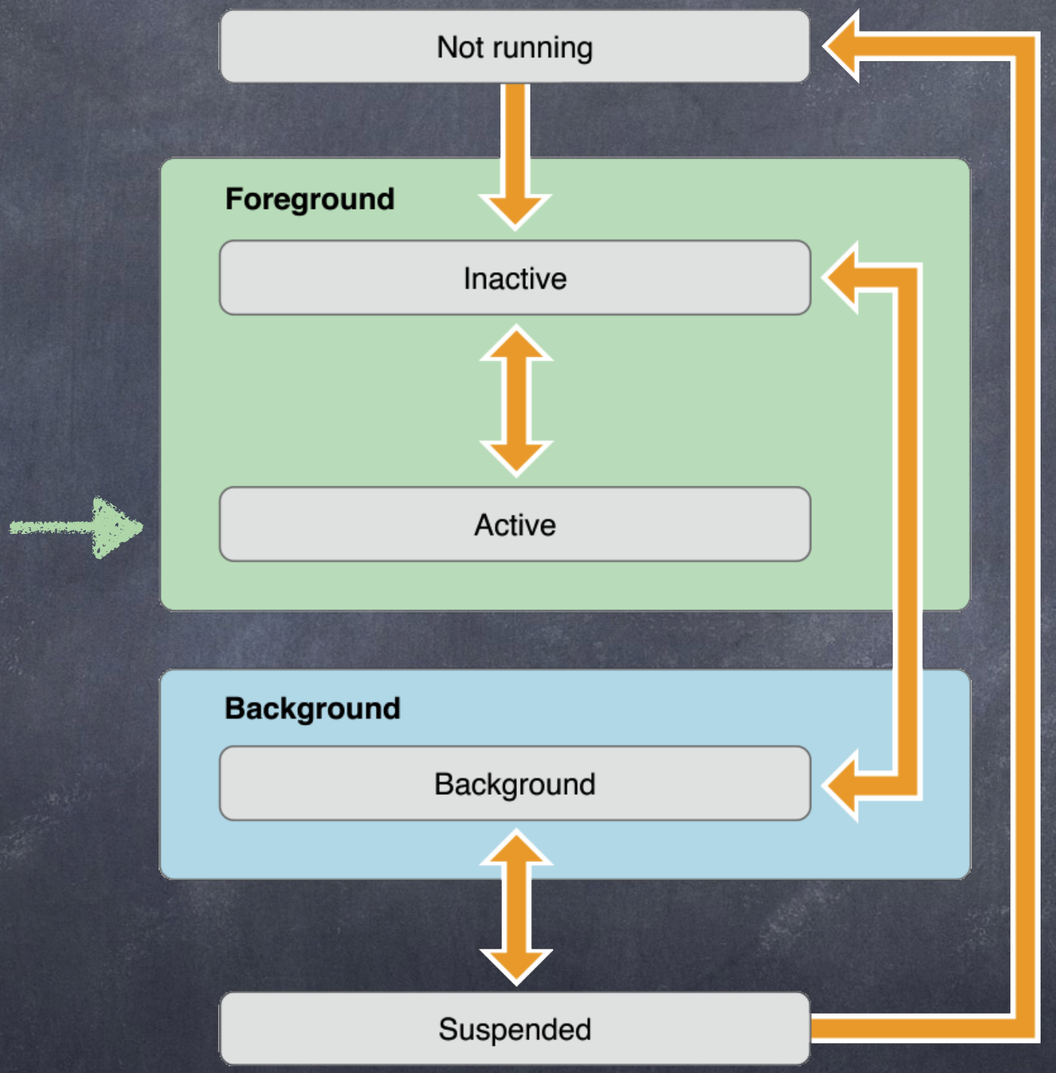
Application Lifecycle

Running your code, but no UI events.

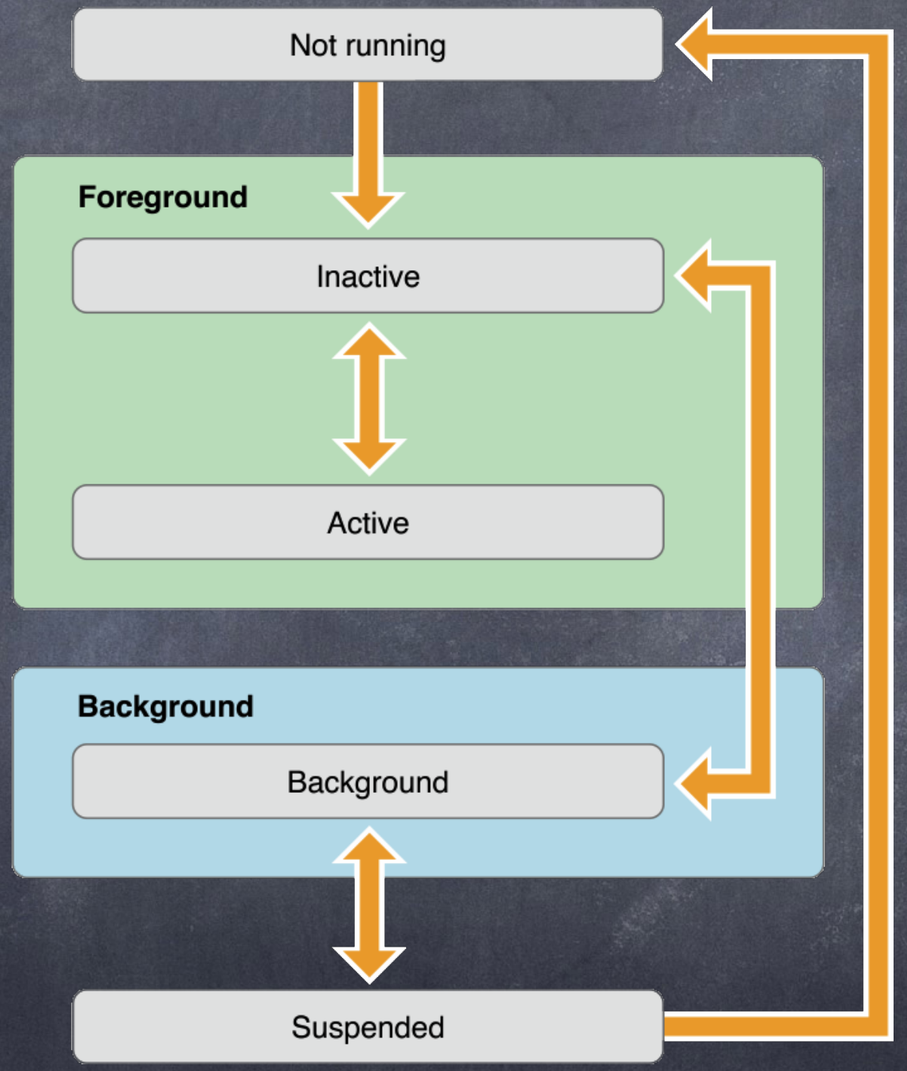


Application Lifecycle

Running your code, receiving and processing UI events.

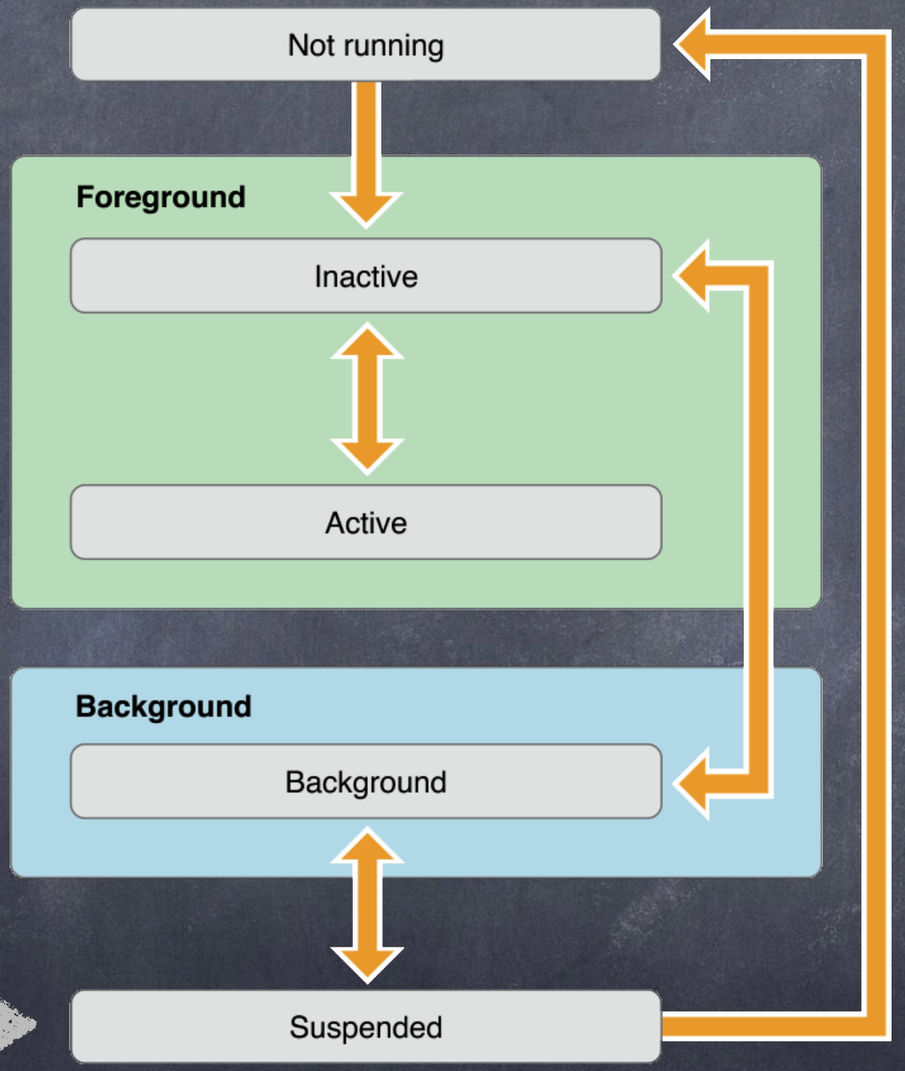


Application Lifecycle



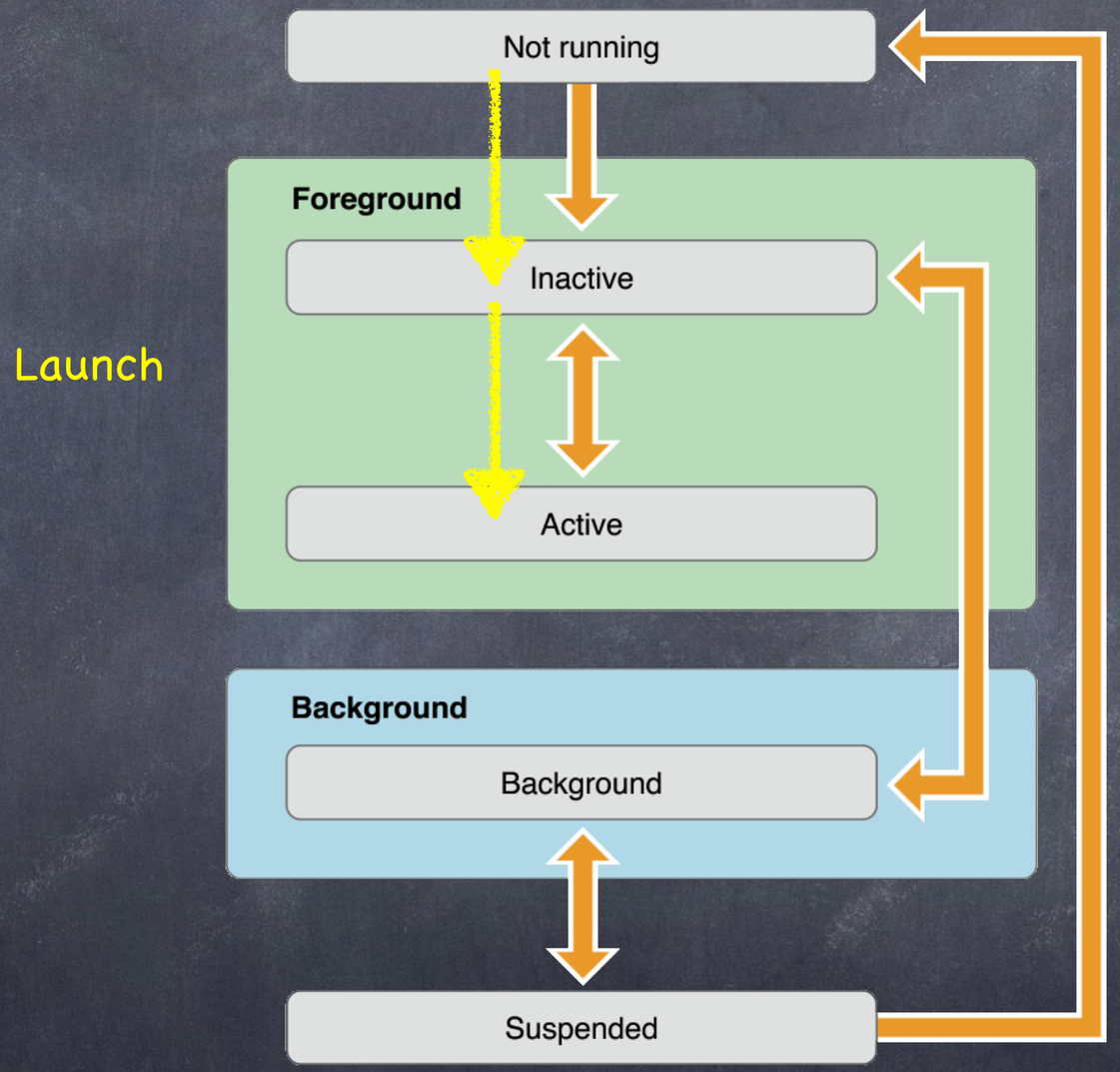
Running your code for a limited time, no UI events.

Application Lifecycle



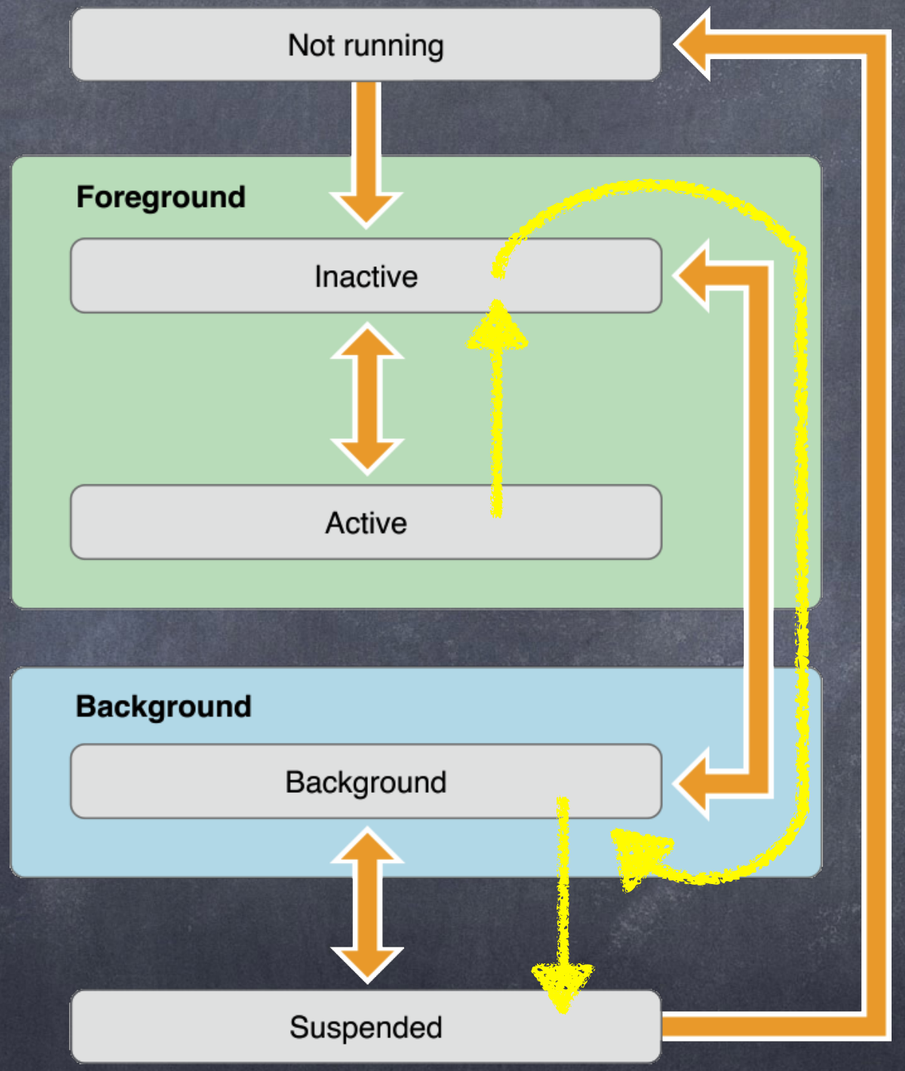
Your code not running.
You could be killed.

Application Lifecycle

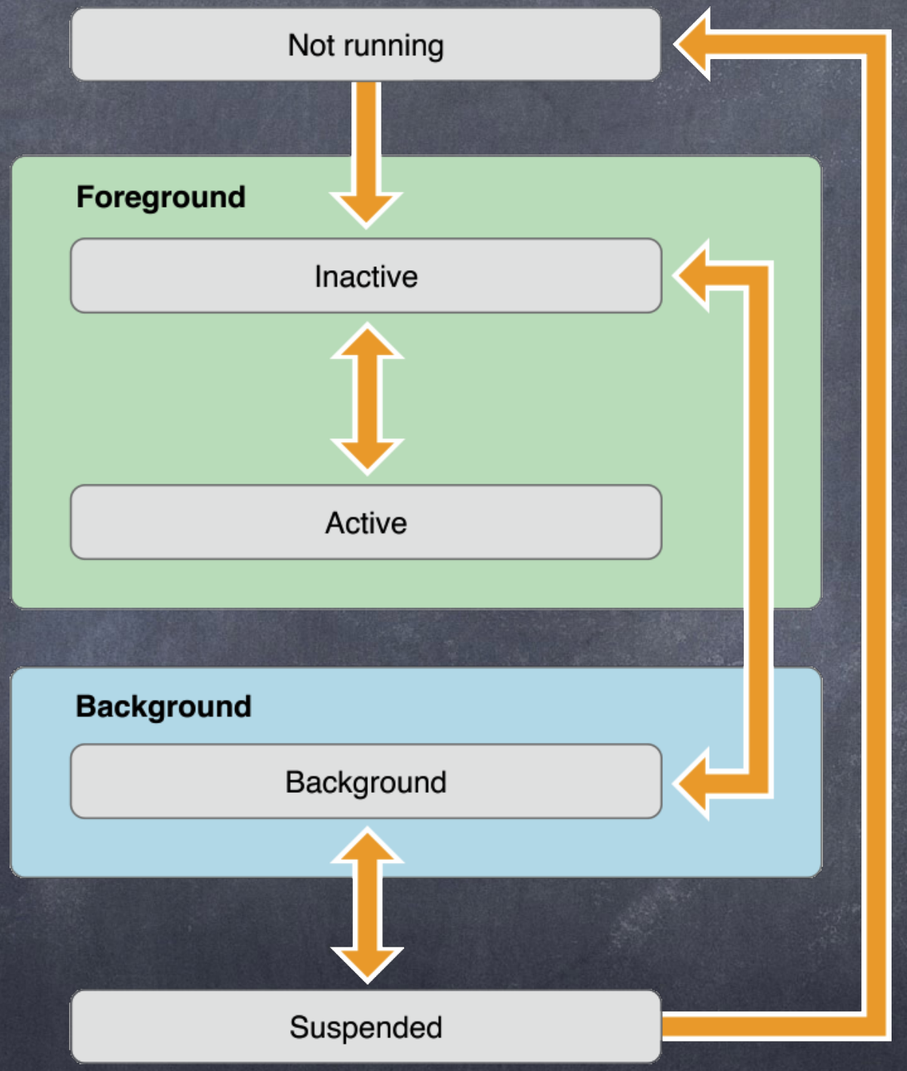


Application Lifecycle

Switch to another application



Application Lifecycle



Killed
(notice no code runs between suspended and killed)

Application Lifecycle

Your AppDelegate will receive …

```
func application(UIApplication,
    didFinishLaunchingWithOptions: [NSObject:AnyObject])
```

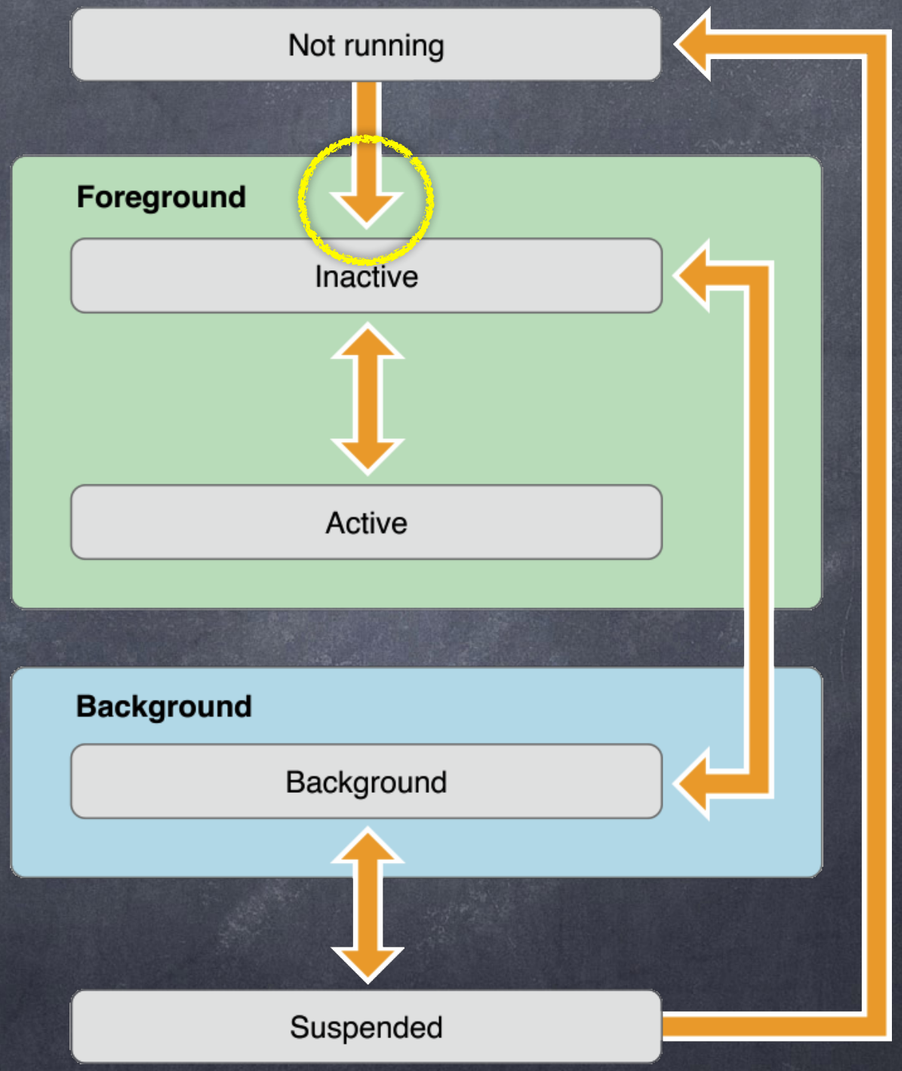
… and you can observe …

```
UIApplicationDidFinishLaunchingNotification
```

The passed dictionary (also in `notification.userInfo`) tells you why your application was launched.

Some examples …

- Someone wants you to open a URL
- You entered a certain place in the world
- You are continuing an activity started on another device
- A notification arrived for you (push or local)
- Bluetooth attached device wants to interact with you



Application Lifecycle

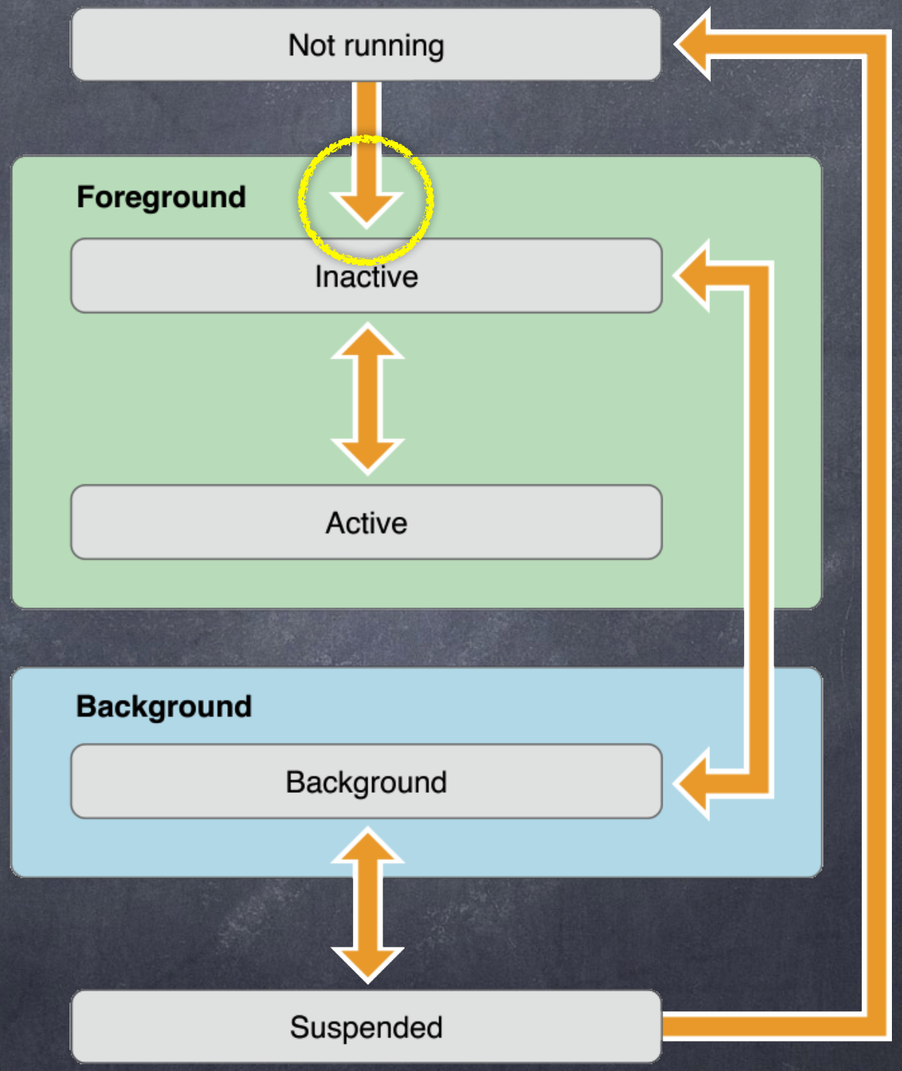
Your AppDelegate will receive …

```
func application(UIApplication,
    didFinishLaunchingWithOptions: [NSObject:AnyObject])
```

… and you can observe …

```
UIApplicationDidFinishLaunchingNotification
```

It used to be that you would build your UI here.
For example, you'd instantiate a split view controller and put a navigation controller inside, then push your actual content view controller.
But nowadays we use storyboards for all that.
So often you do not implement this method at all.



Application Lifecycle

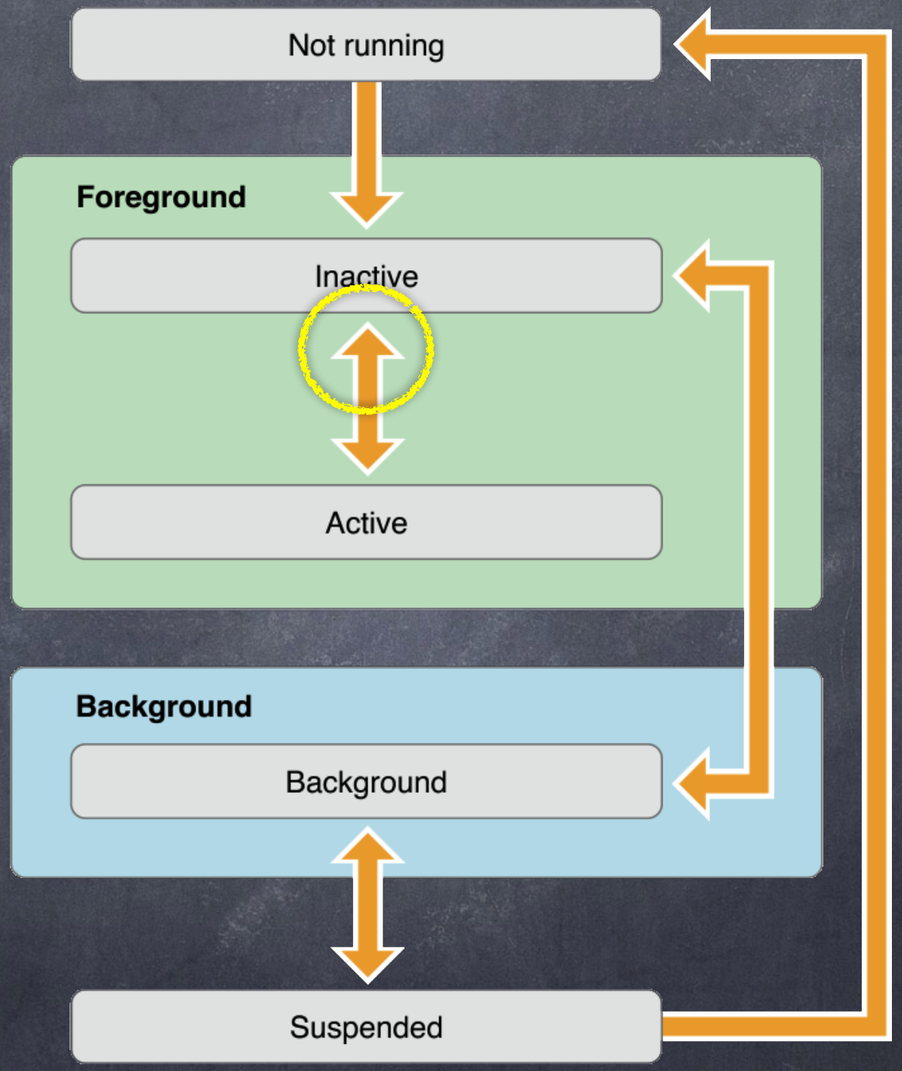
Your AppDelegate will receive …

```
func applicationWillResignActive(UIApplication)
```

… and you can observe …

```
UIApplicationWillResignActiveNotification
```

You will want to "pause" your UI here.
For example, Breakout would want to stop the bouncing ball.
This might happen because a phone call comes in.
Or you might be on your way to the background.



Application Lifecycle

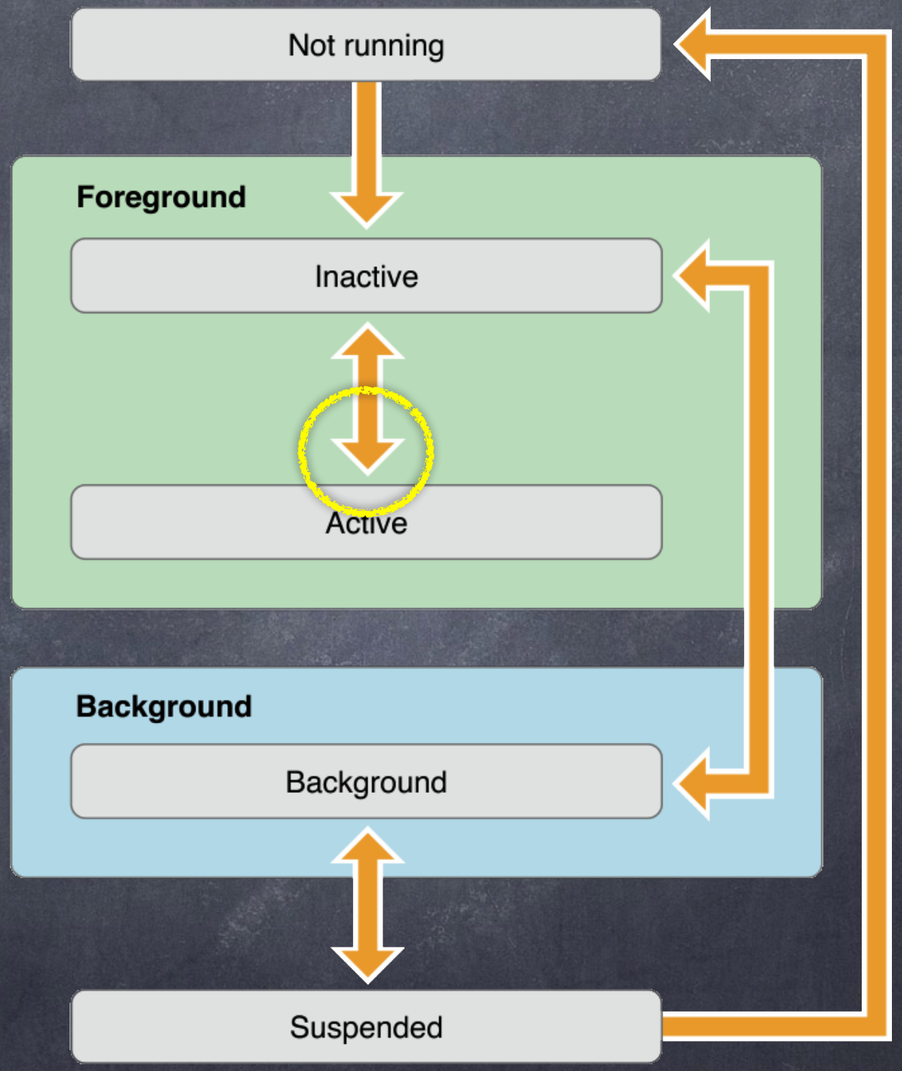
Your AppDelegate will receive …

```
func applicationDidBecomeActive(UIApplication)
```

… and you can observe …

```
UIApplicationDidBecomeActiveNotification
```

If you have "paused" your UI previously
here's where you would reactivate things.



Application Lifecycle

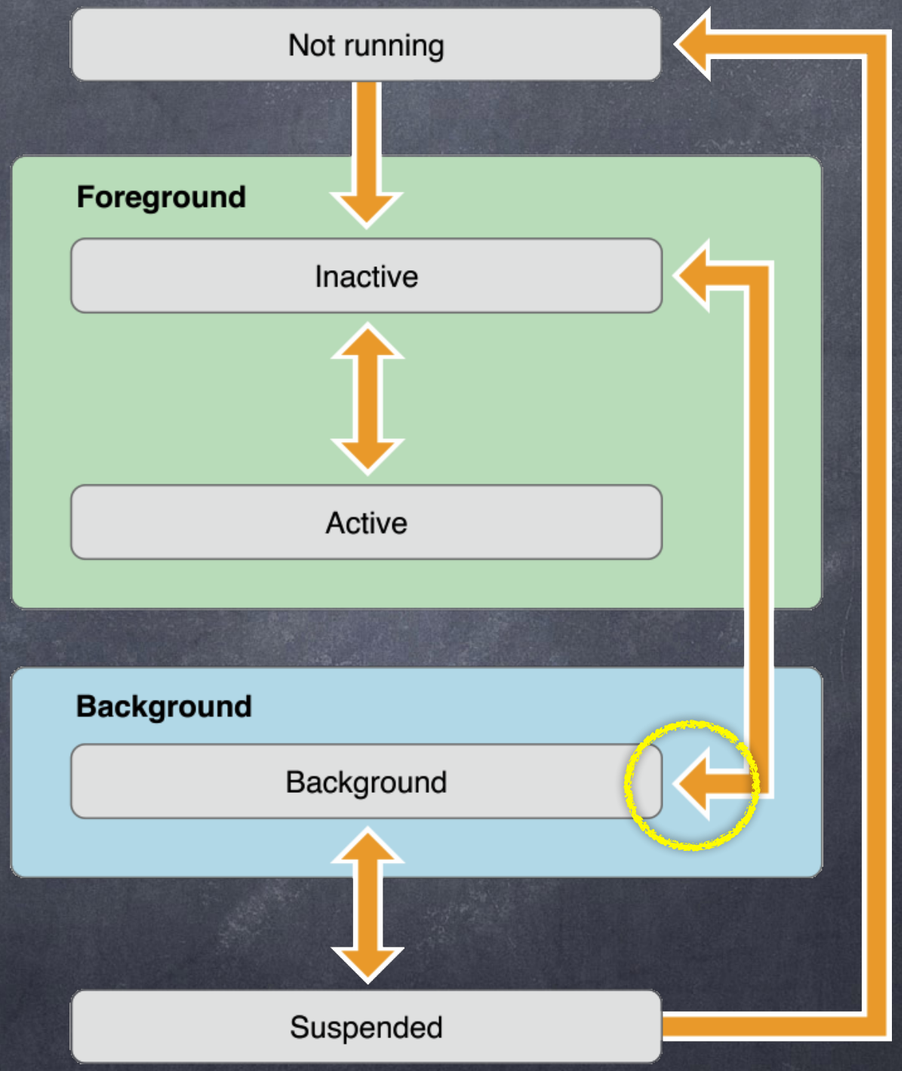
Your AppDelegate will receive …

```
func applicationDidEnterBackground(UIApplication)
```

… and you can observe …

```
UIApplicationDidEnterBackgroundNotification
```

Here you want to (quickly) batten down the hatches.
You only get to run for 30s or so.
You can request more time, but don't abuse this
(or the system will start killing you instead).
Prepare yourself to be eventually killed here
(probably won't happen, but be ready anyway).



Application Lifecycle

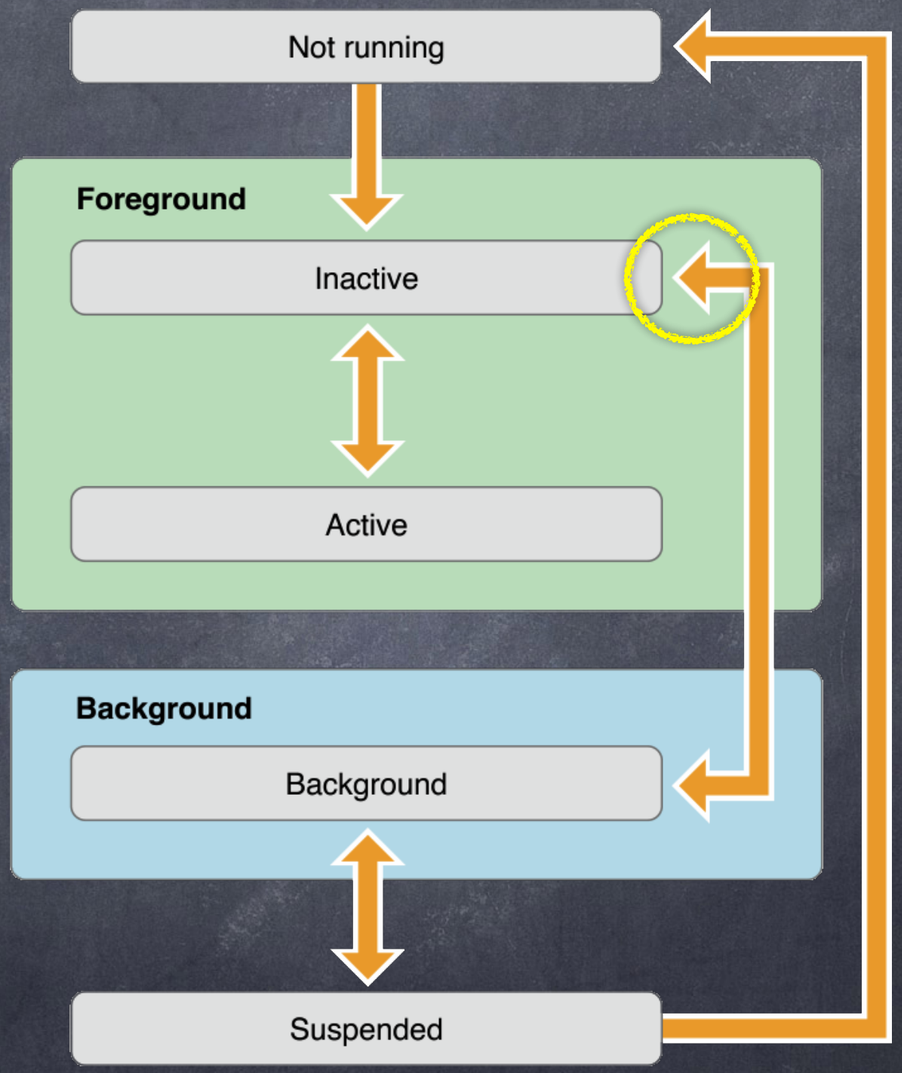
Your AppDelegate will receive …

```
func applicationWillEnterForeground(UIApplication)
```

… and you can observe …

```
UIApplicationWillEnterForegroundNotification
```

Whew! You were not killed from background state!
Time to un-batten the hatches.
Maybe undo what you did in DidEnterBackground.
You will likely soon be made Active.



UIApplicationDelegate

- Other `AppDelegate` items of interest …
  - Local Notifications (set timers to go off at certain times … will wake your application if needed).
  - State Restoration (saving the state of your UI so that you can restore it even if you are killed).
  - Data Protection (files can be set to be protected when a user's device's screen is locked).
  - Open URL (in Xcode's Info tab of Project Settings, you can register for certain URLs).
  - Background Fetching (you can fetch and receive results while in the background).

UIApplication

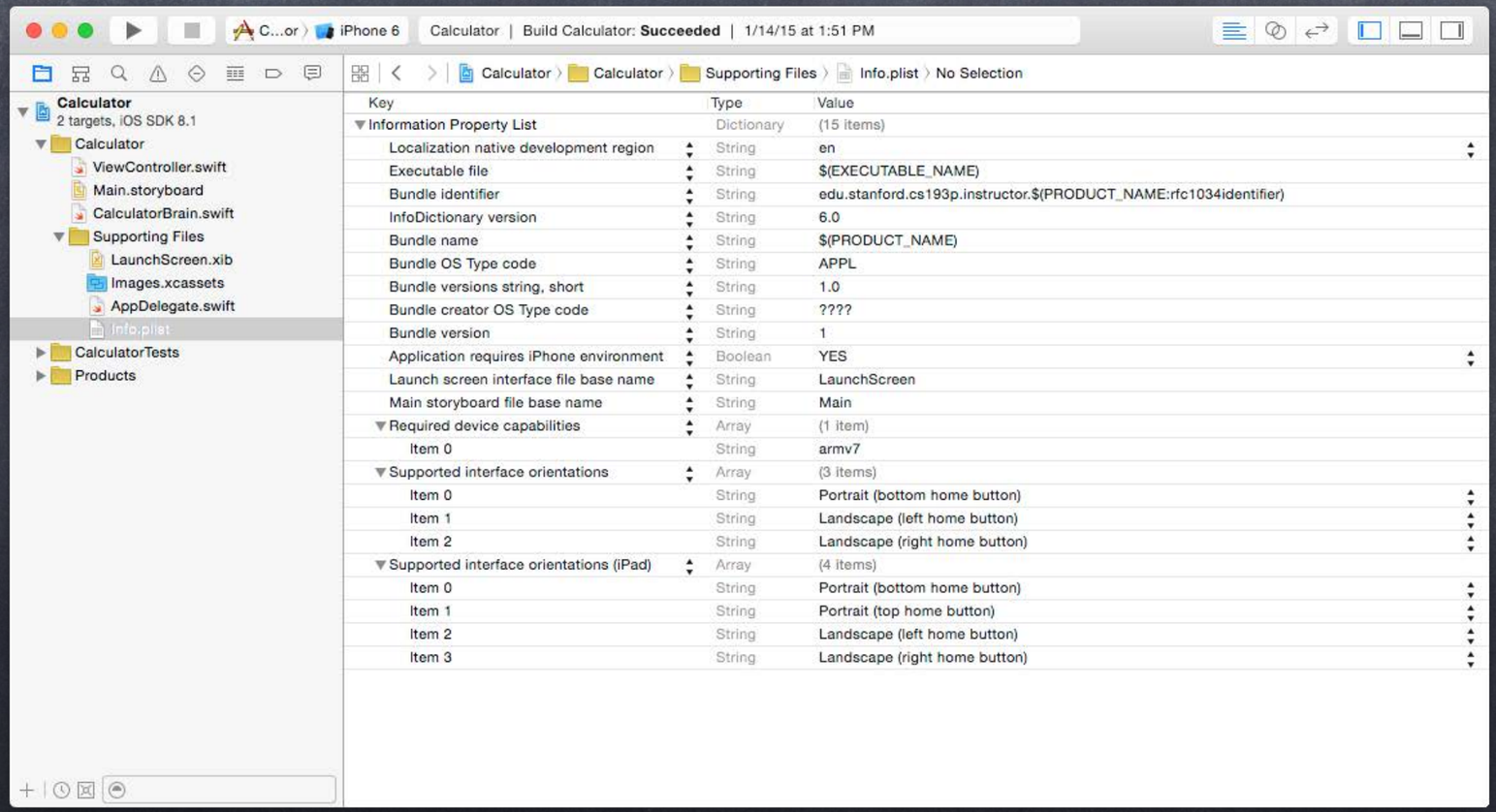
- Shared instance
  - There is a single `UIApplication` instance in your application
  - `let myApp = UIApplication.sharedApplication()`
  - It manages all global behavior
  - You never need to subclass it
  - It delegates everything you need to be involved in to its `UIApplicationDelegate`
  - However, it does have some useful functionality …
- Opening a URL in another application
  - `func openURL(NSURL)`
  - `func canOpenURL(NSURL) -> Bool`
- Registering or Scheduling Notifications (Push or Local)
  - `func (un)registerForRemoteNotifications()`
  - `func scheduleLocalNotification(UILocalNotification)`
  - `func registerUserNotificationSettings(UIUserNotificationSettings)` // permit for badges, etc.

UIApplication

- Setting the fetch interval for background fetching
  - You must set this if you want background fetching to work …
  - `func setMinimumBackgroundFetchInterval(NSTimeInterval)`
  - Usually you will set this to `UIApplicationBackgroundFetchIntervalMinimum`
- Asking for more time when backgrounded
  - `func backgroundTaskWithExpirationHandler(handler: () -> Void) -> UIBackgroundTaskIdentifier`
  - Do NOT forget to call `endBackgroundTask(UIBackgroundTaskIdentifier)` when you're done!
- Turning on the "network in use" spinner (status bar upper left)
  - `var networkActivityIndicatorVisible: Bool` // unfortunately just a `Bool`, be careful
- Finding out about things
  - `var backgroundTimeRemaining: NSTimeInterval { get }` // until you are suspended
  - `var preferredContentSizeCategory: String { get }` // big fonts or small fonts
  - `var applicationState: UIApplicationState { get }` // foreground, background, active

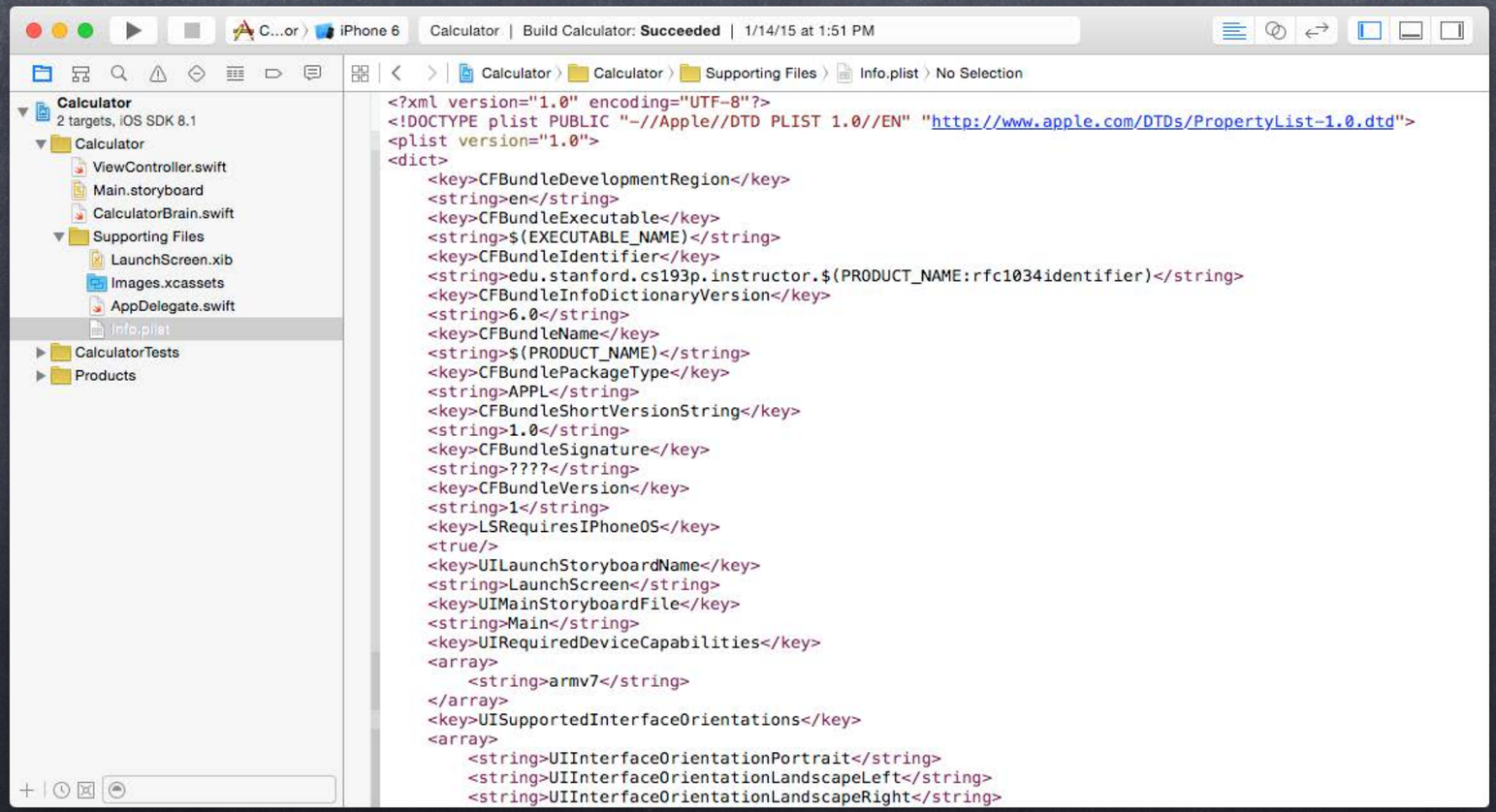
Info.plist

- Many of your application's settings are in Info.plist
  - You can edit this file (in Xcode's property list editor) by clicking on Info.plist



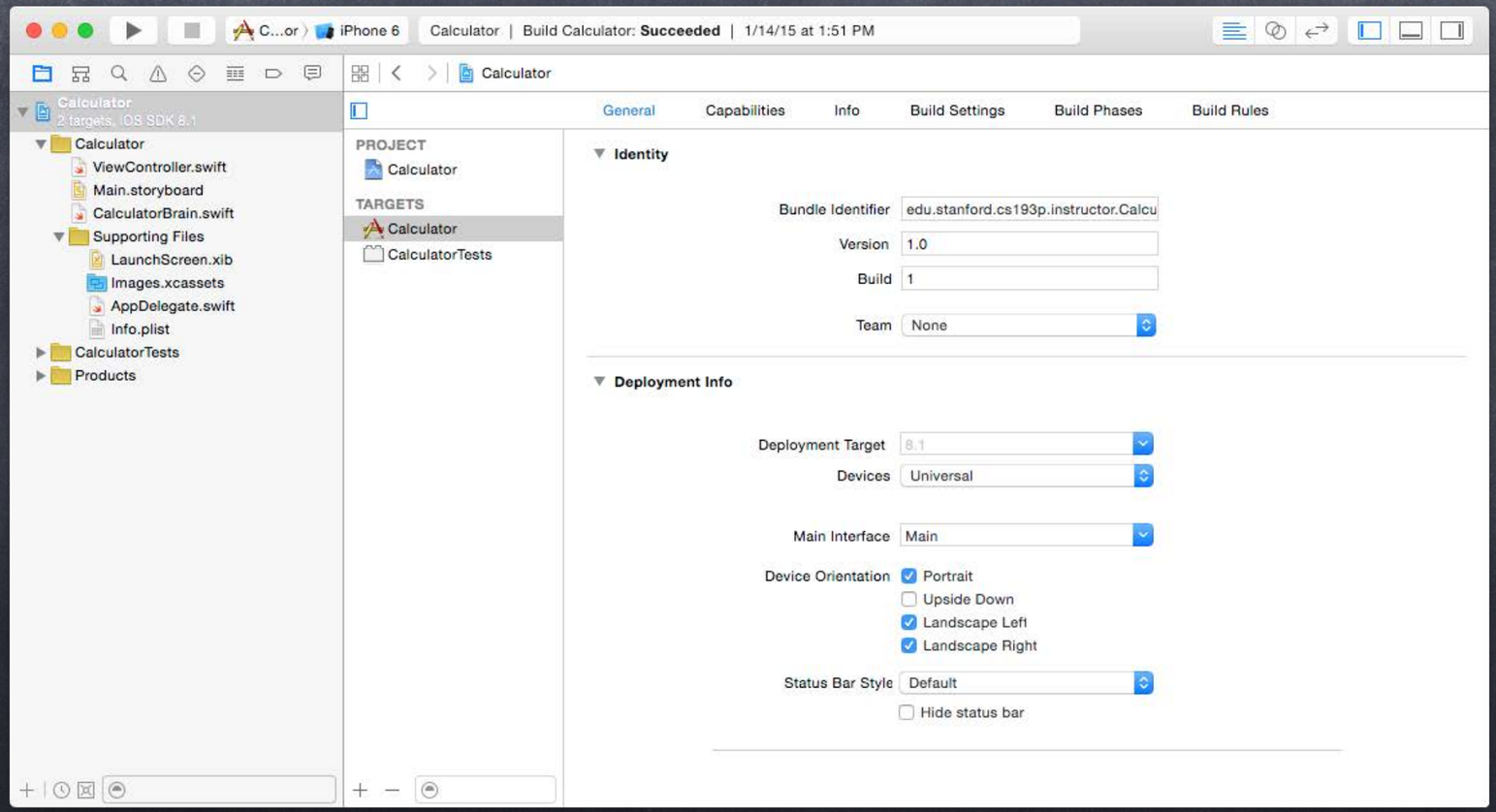
Info.plist

- Many of your application's settings are in Info.plist
  - You can edit this file (in Xcode's property list editor) by clicking on Info.plist
  - Or you can even edit it as raw XML!



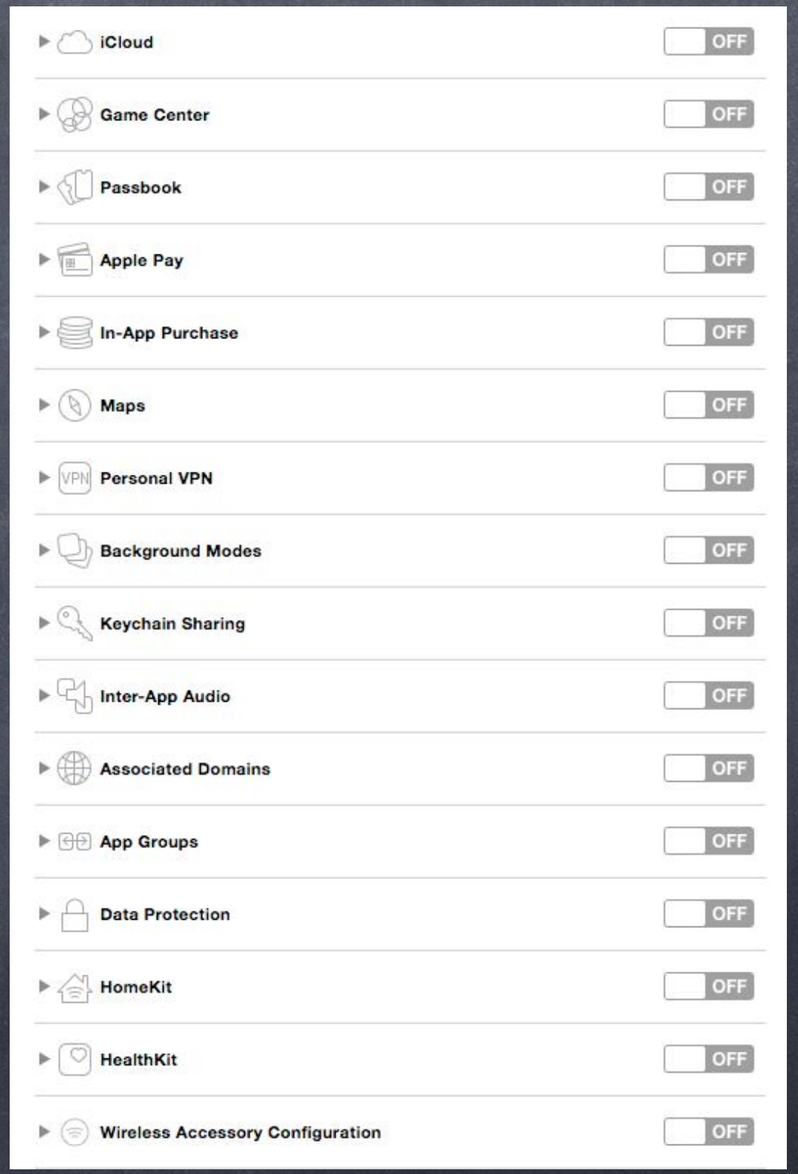
Info.plist

- Many of your application's settings are in Info.plist
  - You can edit this file (in Xcode's property list editor) by clicking on Info.plist
  - Or you can even edit it as raw XML!
  - But usually you edit Info.plist settings by clicking on your project in the Navigator …



Capabilities

- Some features require enabling
  - These are server and interoperability features
  - Like iCloud, Game Center, etc.
- Switch on in Capabilities tab
  - Inside your Project Settings
- Not enough time to cover these!
  - But check them out!
  - Many require full Developer Program membership
  - Familiarize yourself with their existence



AirDrop Demo

- AirDrop
  - Simple mechanism for file transfer
  - Requires our iOS app to register itself to open certain URLs
  - For our demo, we'll do "`gpx` files" (which store GPS location information in xml format)
  - We'll edit `Info.plist` directly to do this (for expediency), but we'll see Project Settings too
  - Once we've done that, we'll implement `AppDelegate` `openURL` method
  - That's all we need to do!
  - On Wednesday, we'll actually do something with `gpx` files that are AirDropped in

Core Motion

- API to access motion sensing hardware on your device
- Primary inputs: Accelerometer, Gyro, Magnetometer

  Not all devices have all inputs (e.g. only iPhone4/5/6 and 4th G iPod Touch and iPad 2+ have a gyro)

- Class used to get this input is `CMMotionManager`

  Use only one instance per application (else performance hit)

  It is a "global resource," so getting one via a class method somewhere is okay

- Usage

  1. Check to see what hardware is available
  2. Start the sampling going and poll the motion manager for the latest sample it has

  ... or ...

  1. Check to see what hardware is available
  2. Set the rate at which you want data to be reported from the hardware
  3. Register a closure (and a queue to run it on) to call each time a sample is taken

Core Motion

- Checking availability of hardware sensors

  ```
  var {accelerometer,gyro,magnetometer,deviceMotion}Available: Bool
  ```

  The "device motion" is a combination of all available (accelerometer, magnetometer, gyro).

  We'll talk more about that in a couple of slides.

- Starting the hardware sensors collecting data

  You only need to do this if you are going to poll for data.

  ```
  func start{Accelerometer,Gyro,Magnetometer,DeviceMotion}Updates()
  ```

- Is the hardware currently collecting data?

  ```
  var {accelerometer,gyro,magnetometer,deviceMotion}Active: Bool
  ```

- Stop the hardware collecting data

  It is a performance hit to be collecting data, so stop during times you don't need the data.

  ```
  func stop{Accelerometer,Gyro,Magnetometer,DeviceMotion}Updates()
  ```

Core Motion

- Checking the data (polling not recommended, more later)

```
var accelerometerData: CMAccelerometerData
```

CMAccelerometerData object provides `var acceleration: CMAcceleration`

```
struct CMAcceleration {
    var x: Double  // in g (9.8 m/s/s)
    var y: Double  // in g
    var z: Double  // in g
}
```

This raw data includes acceleration due to gravity

So, if the device were laid flat, `z` would be `1.0` and `x` and `y` would be `0.0`

Core Motion

- Checking the data (polling not recommended, more later)

```
var gyroData: CMGyroData
```

CMGyroData object provides `var rotationRate: CMRotationRate`

```
struct CMRotationRate {
    var x: Double  // in radians/s
    var y: Double  // in radians/s
    var z: Double  // in radians/s
}
```

Sign of the rotation data follows right hand rule

The data above will be biased

Core Motion

- Checking the data (polling not recommended, more later)

```
var magnetometerData: CMMagnetometerData
CMMagnetometerData object provides var magneticField: CMMagneticField
struct CMMagneticField {
    var x: Double  // in microteslas
    var y: Double  // in microteslas
    var z: Double  // in microteslas
}
```

The data above will be biased

CMDeviceMotion

- Acceleration Data in CMDeviceMotion

```
var gravity: CMAcceleration
var userAcceleration: CMAcceleration // gravity factored out using gyro
```

- Rotation Data in CMDeviceMotion

```
var rotationRate: CMRotationRate // bias removed from raw data using accelerometer
var attitude: CMAttitude         // device's attitude (orientation) in 3D space
class CMAttitude: NSObject       // roll, pitch and yaw are in radians
    var roll: Double    // around longitudinal axis passing through top/bottom
    var pitch: Double   // around lateral axis passing through sides
    var yaw: Double     // around axis with origin at CofG and ⊥ to screen directed down
}
// other mathematical representations of the device's attitude also available
```

CMDeviceMotion

- Magnetic Field Data in CMDeviceMotion

```
var magneticField: CMCalibratedMagneticField
struct CMCalibratedMagneticField {
    var field: CMMagneticField
    var accuracy: CMMagneticFieldCalibrationAccuracy
}
```

accuracy can be …

- CMMagneticFieldCalibrationAccuracyUncalibrated
- CMMagneticFieldCalibrationAccuracyLow
- CMMagneticFieldCalibrationAccuracyMedium
- CMMagneticFieldCalibrationAccuracyHigh

Core Motion

- Registering a block to receive Accelerometer data

```
func startAccelerometerUpdatesToQueue(queue: NSOperationQueue!,
                                      withHandler: CMAccelerometerHandler)
typealias CMAccelerationHandler = (CMAccelerometerData!, NSError!) -> Void
```

queue can be an NSOperationQueue() you create or NSOperation.mainQueue (or currentQueue)

- Registering a block to receive Gyro data

```
func startGyroUpdatesToQueue(queue: NSOperationQueue!,
                             withHandler: CMGyroHandler)
typealias CMGyroHandler = (CMAGyroData!, NSError!) -> Void
```

- Registering a block to receive Magnetometer data

Core Motion

- Registering a block to receive DeviceMotion data

```
func startDeviceMotionUpdatesToQueue(queue: NSOperationQueue!,
                                     withHandler: CMDeviceMotionHandler)
typealias CMDeviceMotionHandler = (CMDeviceMotion!, NSError!) -> Void
```

queue can be an NSOperationQueue() you create or NSOperation.mainQueue (or currentQueue)

Errors … CMErrorDeviceRequiresMovement
CMErrorTrueNorthNotAvailable
CMErrorMotionActivityNotAvailable
CMErrorMotionActivityNotAuthorized

Core Motion

- Setting the rate at which your block gets executed

```
var accelerometerUpdateInterval: NSTimeInterval
var gyroUpdateInterval: NSTimeInterval
var magnetometerUpdateInterval: NSTimeInterval
var deviceMotionUpdateInterval: NSTimeInterval
```

- It is okay to add multiple handler blocks

  Even though you are only allowed one `CMMotionManager`

  However, each of the blocks will receive the data at the same rate (as set above)

  (Multiple objects are allowed to poll at the same time as well, of course.)

Demo

- Bouncer
  - Use real gravity as gravity in a `UIDynamicAnimator`
  - We'll get the gravity information from CoreMotion