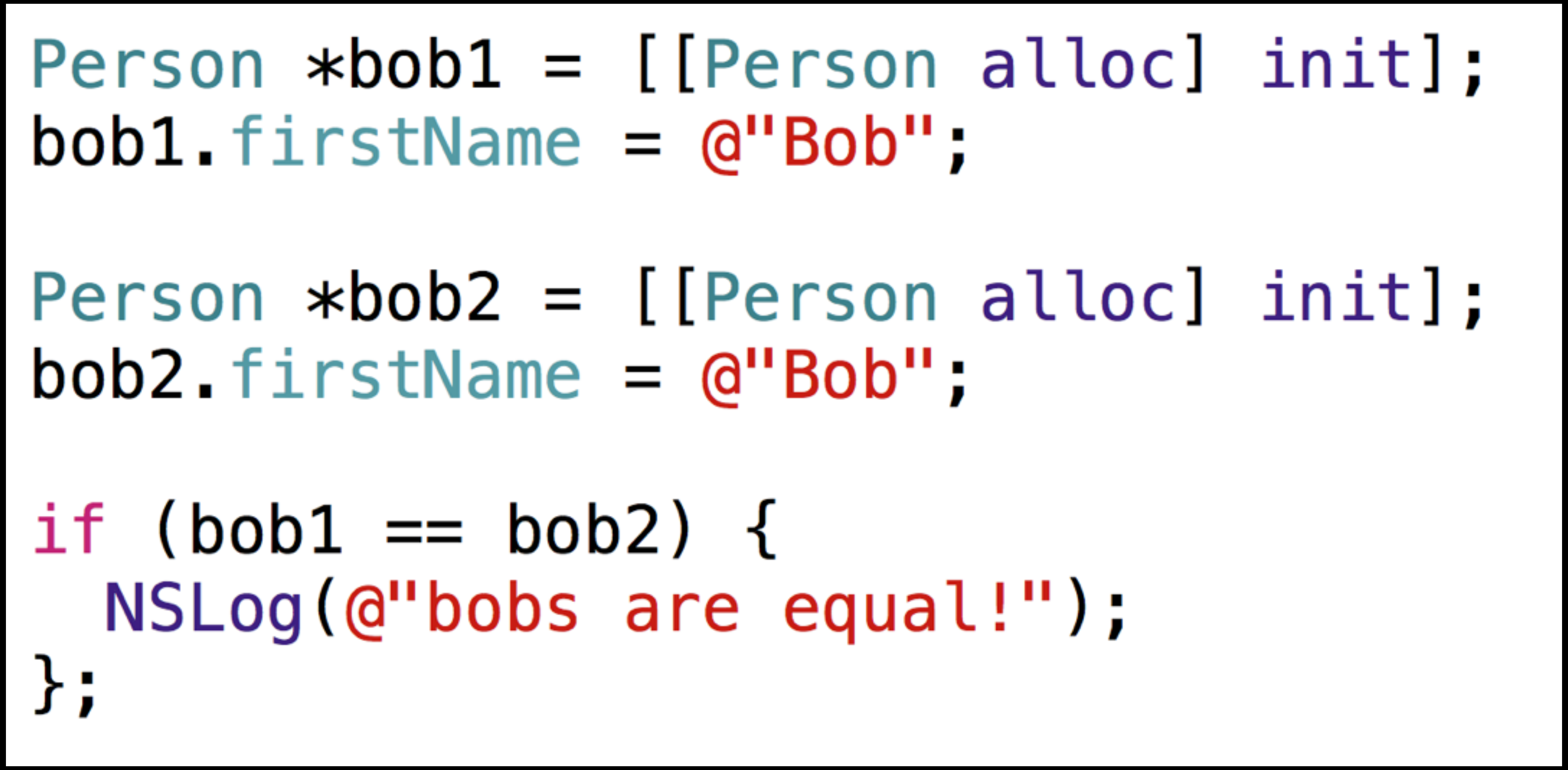
iOS Dev Accelerator
Week 5 Day 3

- Object Equality
- PFSubclassing
- MKMapViewDelegate
- Broadcast Pattern & Notification Center
- App state
- Local & Push Notifications

Object Equality

*sourced from NSHipster

POP QUIZ:



```
Person *bob1 = [[Person alloc] init];
bob1.firstName = @"Bob";

Person *bob2 = [[Person alloc] init];
bob2.firstName = @"Bob";

if (bob1 == bob2) {
  NSLog(@"bobs are equal!");
};
```

Will “bobs are equal” print?????

The answer is no!

….for now

Equality & Identity

- First, we need to distinguish the difference between equality and identify.
- Two objects are said to be equal to one another if they share a common set of properties.
- Yet even if they are equal, two objects can still be thought to be distinct, each with their own identity.
- In programming, an object’s identity is tied to its memory address.

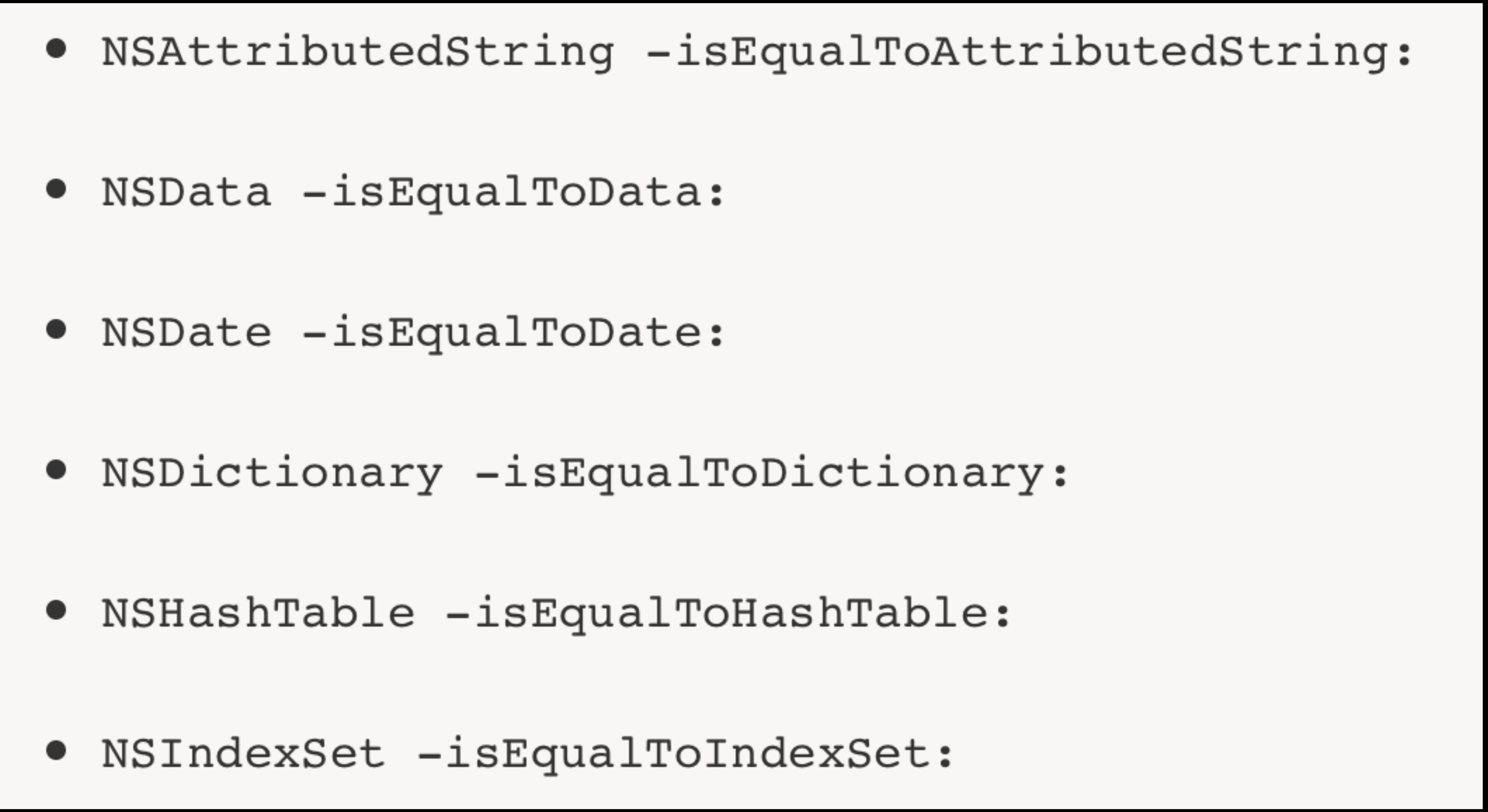
NSObject and Equality

- By default, an NSObject uses identity to test for object equality.
- When you use the == operator to compare two objects, it simply compares their pointer values, so its testing the identity.
- NSObject also has a method called isEqual:, which by default simply runs == under the hood.
- So two NSObjects by default are considered to be equal if they point to the same memory address

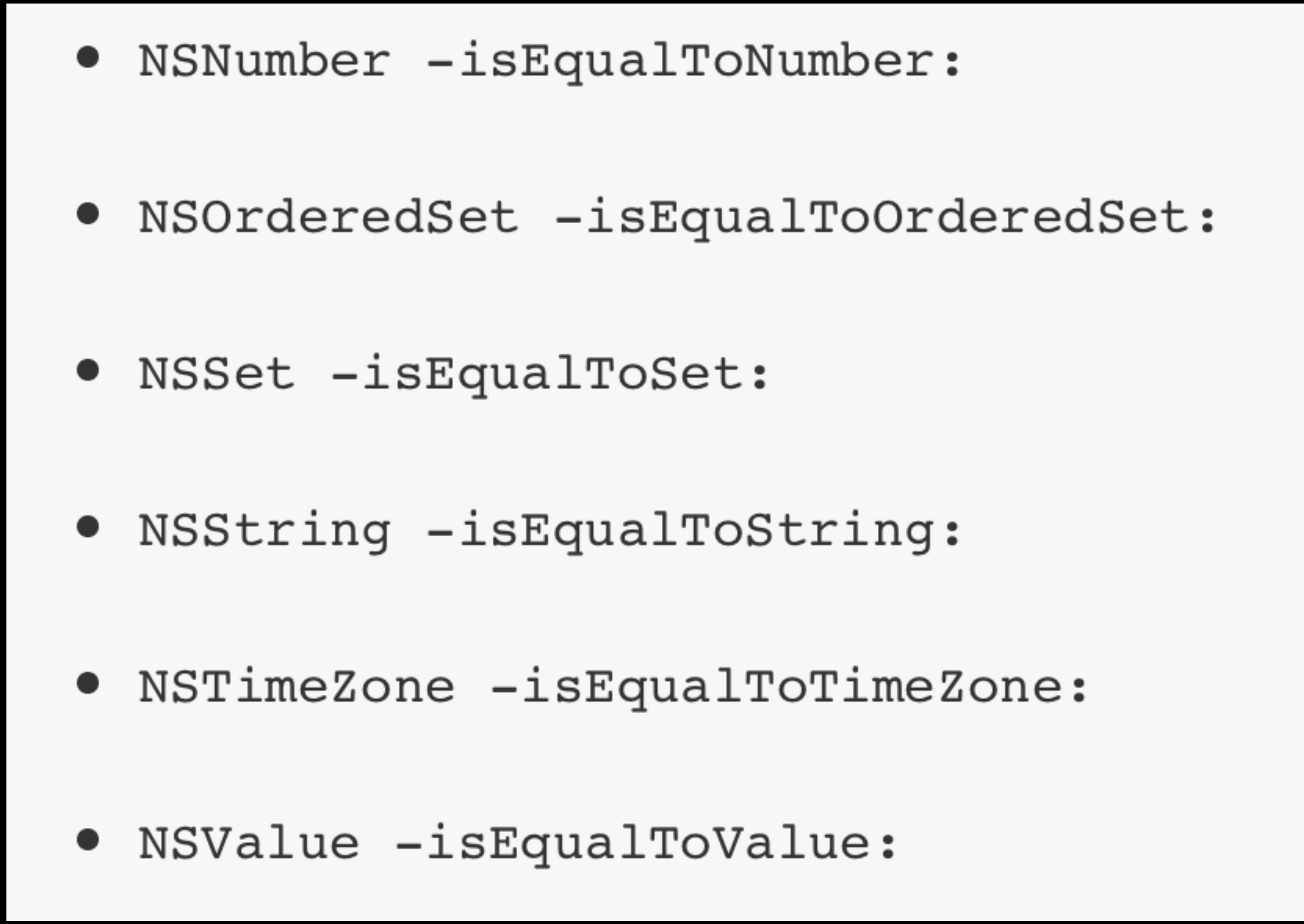
Subclasses and Equality

- Certain subclasses of NSObject provide their own version of isEqual, which does more than just an identity test.
- Typically the format of this method is isEqualTo<ClassName>:
- These methods do a meaningful value comparison.
- For example, an array will loop through all of its values and all the values in the array its being compared against, and will only return true if each pair of objects are equal.

isEquals



- NSAttributedString -isEqualToAttributedString:
- NSData -isEqualToData:
- NSDate -isEqualToDate:
- NSDictionary -isEqualToDictionary:
- NSHashTable -isEqualToHashTable:
- NSIndexSet -isEqualToIndexSet:



- NSNumber -isEqualToNumber:
- NSOrderedSet -isEqualToOrderedSet:
- NSSet -isEqualToSet:
- NSString -isEqualToString:
- NSTimeZone -isEqualToTimeZone:
- NSValue -isEqualToValue:

Whenever you are making comparisons with these classes, you are encouraged to use these methods

Implementing your own Equality checks

- First you need to ask yourself the question, what does it mean for two of my own objects to be equal? (aka identity vs equality)
- Sometimes you simply want the default behavior (identity check)
- If you want meaningful comparisons, you will want to write your own custom isEqualTo<ClassName> and then override isEqual

PFSubclassing

PFObjects

- So far when using Parse in our lectures, we have worked exclusively with PFObjects
- This works fine, but there’s another way we can work with our objects
- We can subclass PFObject, which make our PFObjects feel more like regular model objects (and makes it so we can work with regular properties, instead of the subscripting!)

PFObjects Subclass steps

1. Make a class that subclasses PFObject
2. Make the class conform to the PFSubclassing protocol
3. Implement the class method parseClassName on your class.
4. Import PFObject+Subclass.h in your .m file
5. Call [YourClass registerSubclass] in your app delegate before you call setApplicationID:clientKey:

Changes because of Subclass

- Instead of using [PFObject objectWithClassName:] to create a new object just use [YourClass object]
- Anywhere you were doing the subscripting to set a property, just use regular dot notation now (woohoo!)
- Any queries you perform on that class now pass you an array of your custom subclass instead of PFObjects (woohoo!)

Demo

MKMapViewDelegate

MKMapViewDelegate

- MKMapViewDelegate is a protocol full of optional methods that lets you receive many different types of map related updates
- MKMapView is a highly asynchronous object, since it needs to constantly query for map data
- A map view's delegate is also responsible for annotating the map view as well.

MKMapViewDelegate

- MKMapViewDelegate’s methods are broken down into these categories:
    - Responding to map positions changes
    - Loading Map Data
    - Tracking User Location
    - Managing Annotation Views
    - Managing the Display of Overlays

Docs

Region Monitoring

Region Monitoring

- “A geographical region is an area defined by a circle of a specified radius around a known point on the Earth’s surface.”
- Apps can use region monitoring to be notified when a user crosses specific boundary.
- “In iOS, regions associated with your app are tracked at all times, included when the app isn't running”
- If it detects a region crossing, your app is relaunched in the background.
- Regions are considered a shared resource on the device, so any app can only have a maximum of 20 regions registered for monitoring at a time.

Region Monitoring Availability

- Reasons why region monitoring may not be available to the user:
  - hardware not available
  - user denied app authorization for region monitoring
  - user disabled location services in the settings app
  - the user disabled background app refresh in the settings app
  - the device is in airplane mode
- First ask the CLLocationManager class if region monitoring is available with the method isMonitoringAvailableForClass: and pass in the class for CLCircularRegion

Setting up a Monitor

- In iOS7+ you define geographical regions using the CLCircularRegion class.
- Each region created should include both the data that defines the desired geographical area and a unique identifier string.
- To register a region, call startMonitoringForRegion: method on your CLLocationManager object.
- By default, every time a user's current location crosses a boundary region, the system generates an appropriate region event for your app.
- 2 methods: locationManager:didEnterRegion: and locationManager:didExitRegion:

Demo

Broadcast Pattern & Notification Center

Broadcast Pattern

- Normally one object fires a method on another object. This is how information gets passed in your app.
- But this requires that one object knows who the receiver of the method is, and what methods that receiver implements.
- This effectively couples the two objects together that otherwise may come from independent areas of your app.
- To avoid this scenario, a broadcast model can be introduced using the notification pattern.
- An object posts a notification, which is dispatched to the appropriate observers through NSNotificationCenter.

Notification Pattern

- The notification pattern is used to pass around information related to the occurrence of events.
- A notification encapsulates information about an event, such as a network connection closing or a button being pressed.
- Objects that need to know about an event register with the notification center that it wants to be notified when that event takes place. This object is considered an observer.

Notification Pattern vs Delegation

- Notification pattern is similar to delegation, except for:
  - Any number of objects can receive a notification, where delegation is is a one-to-one relationship.
  - An object may receive any message from the notification center, not just the predefined delegate methods.
  - The object posting the notification does not have to know the observer exists.

Notification Center

- A notification center manages the sending and receiving of notifications.
- Each program, or app, has a default notification center you can access with NSNotificationCenter.defaultCenter(). Almost always you will be posting notifications to this center.
- A notification center delivers notifications synchronously. To send them asynchronously you will have to use a notification queue.

Notification Center Pros vs cons

Pros

- Super easy to implement
- Easily allow multiple objects to respond to one event
- Easily pass objects with notifications

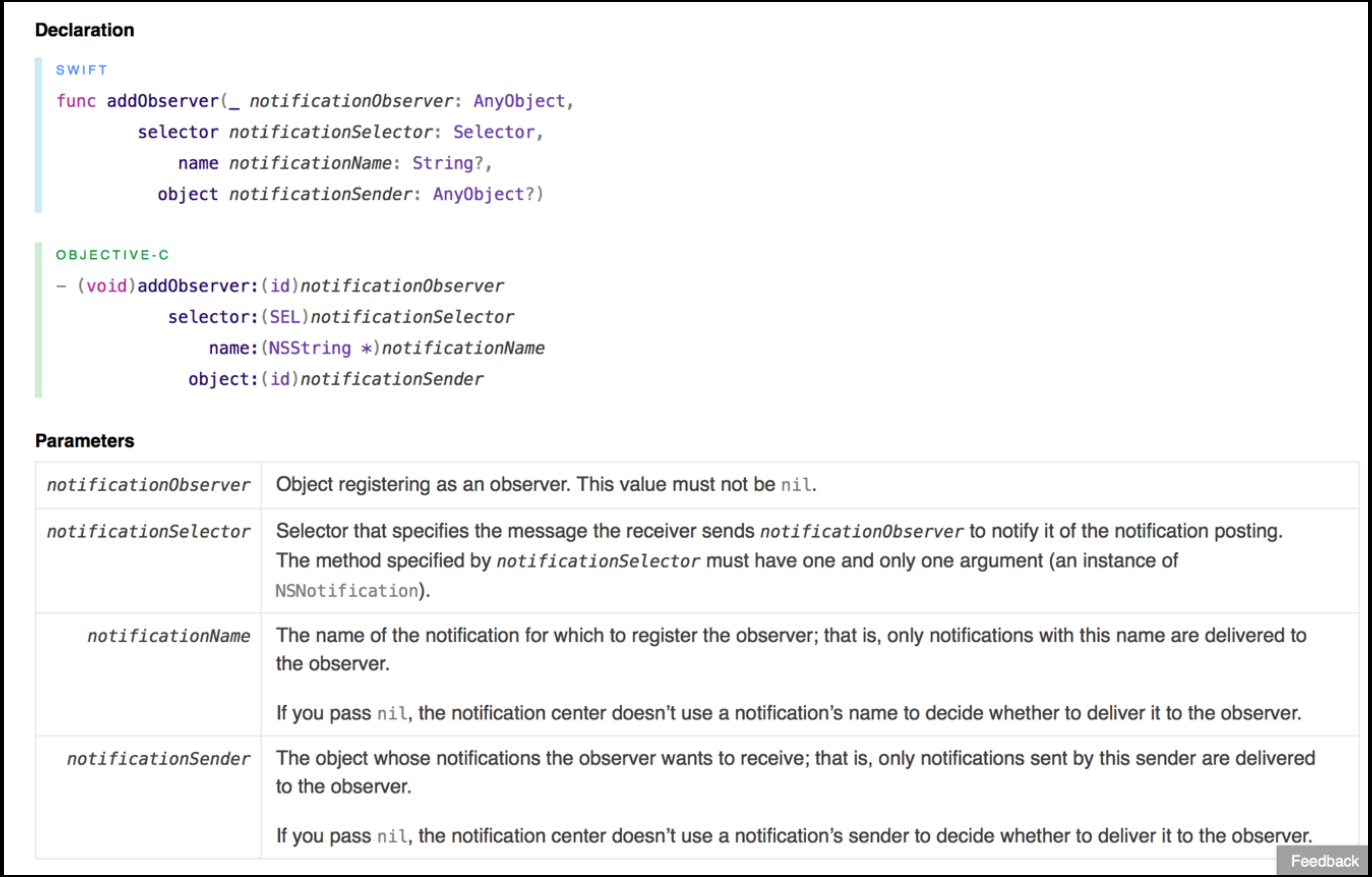
Cons

- No compile time Xcode checks to make sure everything is being handled correctly
- You must remember to remove objects as observers before they are dealloc'd, otherwise crash
- Hard to debug because the flow of the application becomes less apparent.

Registering for Notifications

- You register an object to receive a notification by invoking the notification center method addObserver:selector:name:object
- Name and object are both optional.
- If you only specify an object, the observer will get all notifications containing that object aka were sent by this object
- If you only specify a name, the observer will receive that notification every time its posted, regardless of the object associated with it.

Registering for Notifications



Declaration

SWIFT

```
func addObserver(_ notificationObserver: AnyObject,
        selector notificationSelector: Selector,
            name notificationName: String?,
          object notificationSender: AnyObject?)
```

OBJECTIVE-C

```
- (void)addObserver:(id)notificationObserver
           selector:(SEL)notificationSelector
               name:(NSString *)notificationName
             object:(id)notificationSender
```

Parameters

notificationObserver	Object registering as an observer. This value must not be `nil`.
notificationSelector	Selector that specifies the message the receiver sends *notificationObserver* to notify it of the notification posting. The method specified by *notificationSelector* must have one and only one argument (an instance of `NSNotification`).
notificationName	The name of the notification for which to register the observer; that is, only notifications with this name are delivered to the observer. If you pass `nil`, the notification center doesn't use a notification's name to decide whether to deliver it to the observer.
notificationSender	The object whose notifications the observer wants to receive; that is, only notifications sent by this sender are delivered to the observer. If you pass `nil`, the notification center doesn't use a notification's sender to decide whether to deliver it to the observer.

Unregistering for Notifications

- Before an object that is observing is deallocated, **it must tell the notification center to stop sending it notifications.**
- If you don't unregister, the next notification gets sent to a nonexistent object and your app will crash :(
- The notification center keeps weak references to all of the observers, so it doesn’t ‘own’ the observers. So it doesn't care if those observers are still alive or not. It just keeps firing notifications until they unregister.

Unregistering for Notifications

- 2 methods for unregistering:
  - removeObserver: - removes the passed in object from all notifications
  - removeObserver:name:object: - removes only the matching entries from the notification center’s dispatch table.

Posting Notifications

- Posting notifications is as simple as calling postNotificationName:Object: or postNotificationName:object:userInfo: on the default notification center.
- Notifications are immutable, so once they are created they cannot be modified.
- **If the observer will need some object related to the event, you can pass that object in the userInfo dictionary.**

Demo

Receiving the notifications

- When you register for a notification you pass in a selector, aka a method, that you want to fire when that notification is received.
- This selector can only have at most one parameter, which is the notification object itself that was posted.
- This notification has properties: to the object that originally fired the notification, the name of the notification, and the userInfo object.

Receiving the notifications

Signing up to be the observer

```
- (void)viewDidLoad {
    [super viewDidLoad];

    [[NSNotificationCenter defaultCenter] addObserver:self selector:@selector(reminderAdded:)
        name:@"REMINDER_ADDED" object:nil];
```

The method that will be called when we observe that notification

```
-(void)reminderAdded:(NSNotification *)notification {

    NSDictionary *userInfo = notification.userInfo;
    id postingObject = notification.object;
    NSString *notificationName = notification.name;
```

Demo

App States

App states

- Any at given time, your app is in one of 5 states:
  - **Not Running:** the app has not been launched, or was running and was terminated by the system.
  - **Inactive:** The app is running in the foreground but it is currently not receiving events. This is usually brief and during transitions.
  - **Active:** The app is running in the foreground and receiving events. This is the normal mode for apps.
  - **Background:** The app is in the background and executing code. Apps briefly enter this mode on their way to being suspended. However, an app that requests extra executing time may remain in this state for an extra period of time. Also, an app being launched directly into the background enters this state instead of inactive.
  - **Suspended:** The app is in the background but not executing code. The system moves apps to this state automatically and does not notify them before doing so. While suspended, an app remains in memory but does not execute code. It’s basically frozen in time. When a low-memory condition occurs, apps taking up large amounts of memory are terminated.

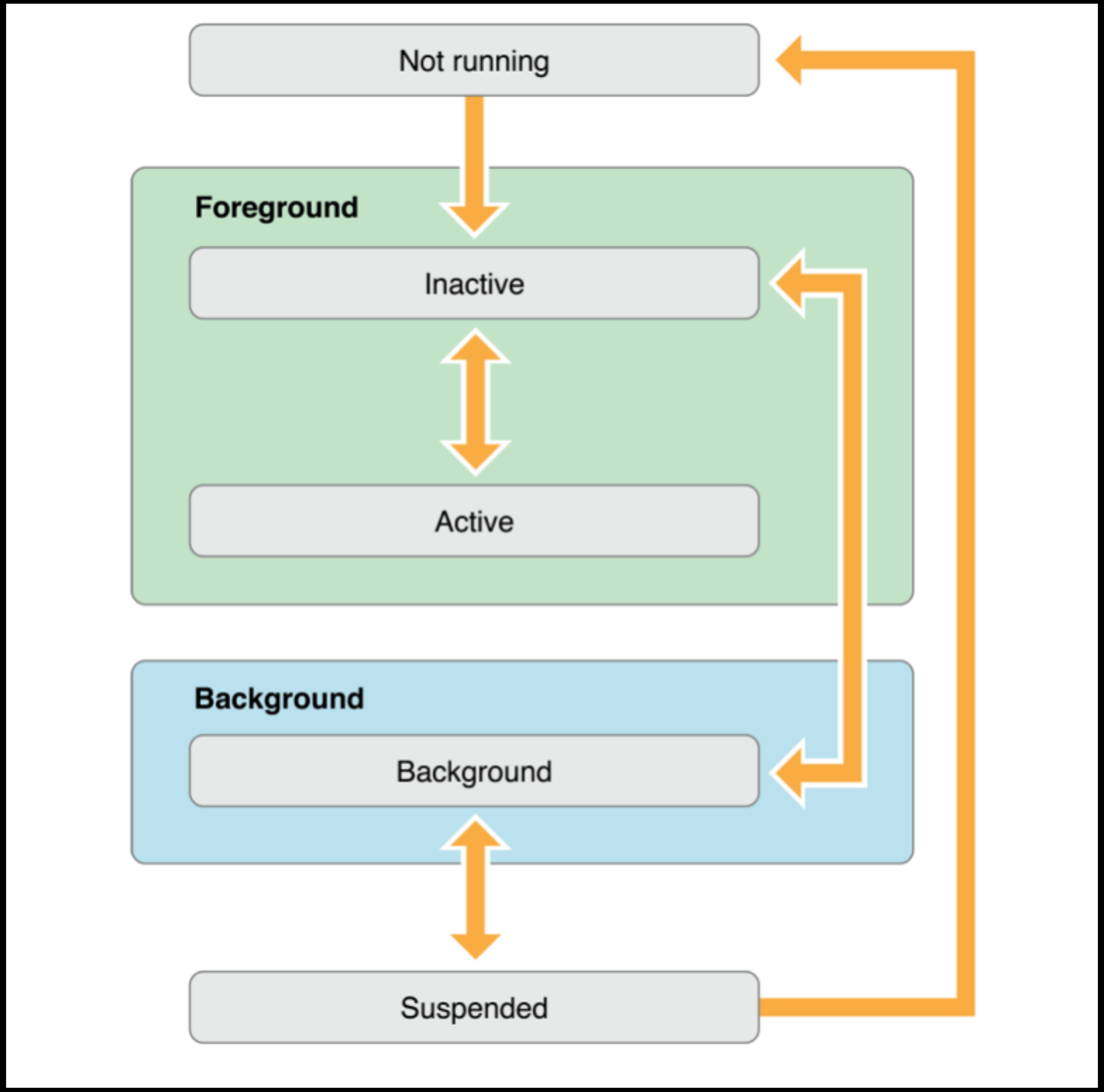
App state behaviors

- Apps behave different in the background and foreground.
- The OS purposely limits what your app can do while its in the background, because it always gives resource priority to the foreground app.
- Your app is always notified when it transitions from background to foreground and vice versa.

Apples App State Guidelines

1. Required - Respond appropriately to state transitions in your app.
2. Required - When moving to background specifically, make sure your app adjusts its behavior accordingly.
3. Recommended - Register for notifications that report system changes that your app needs. If your app is suspended, the system queues up these changes and then gives them to you when your app resumes.
4. Optional - If your app actually does work in the background, you need to ask the system for the appropriate permissions to do that work.

App states diagram



App state transitions

- Most state transitions are accompanied by a corresponding call to the methods of your app delegate object:
- application:willFinishLaunchingWithOptions: - This method is your apps first chance to execute code at launch time.
- application:DidFinishLaunchingWithOptions: - This method allows you to perform any final initialization before your app is displayed to the user.
- applicationDidBecomeActive: - Lets your app know that it is about to become the foreground app.

App state transitions cont.

- applicationWillResignActive: - Lets you know your app is transitioning away from being the foreground app.
- applicationDidEnterBackground: - Lets you know your app is now running in the background and may be suspended at any time.
- applicationWillEnterForeground: - Lets you know your app is moving out of the background and back into the foreground, but it is not yet active.
- applicationWillTerminate: - Lets you know your app is being terminated. This method is not called if your app is already suspended.

Things to do at Launch time

- Check contents of the launch options dictionary about why the app was launched, and respond appropriately. If its a regular launch from the user you usually don't care, but if the app was launched by the system for some background task, you definitely care.
- Initialize the app's critical data structures.
- Prepare your app for display if its going to be displayed.
- Remember to keep the willFinishLaunching and didFinishLaunching methods minimal to not slow down launch speeds.
- The system kills any app that takes longer than 5 seconds to launch.
- When launching straight into the background, there shouldn't be much to do except respond to the event that caused the launch.

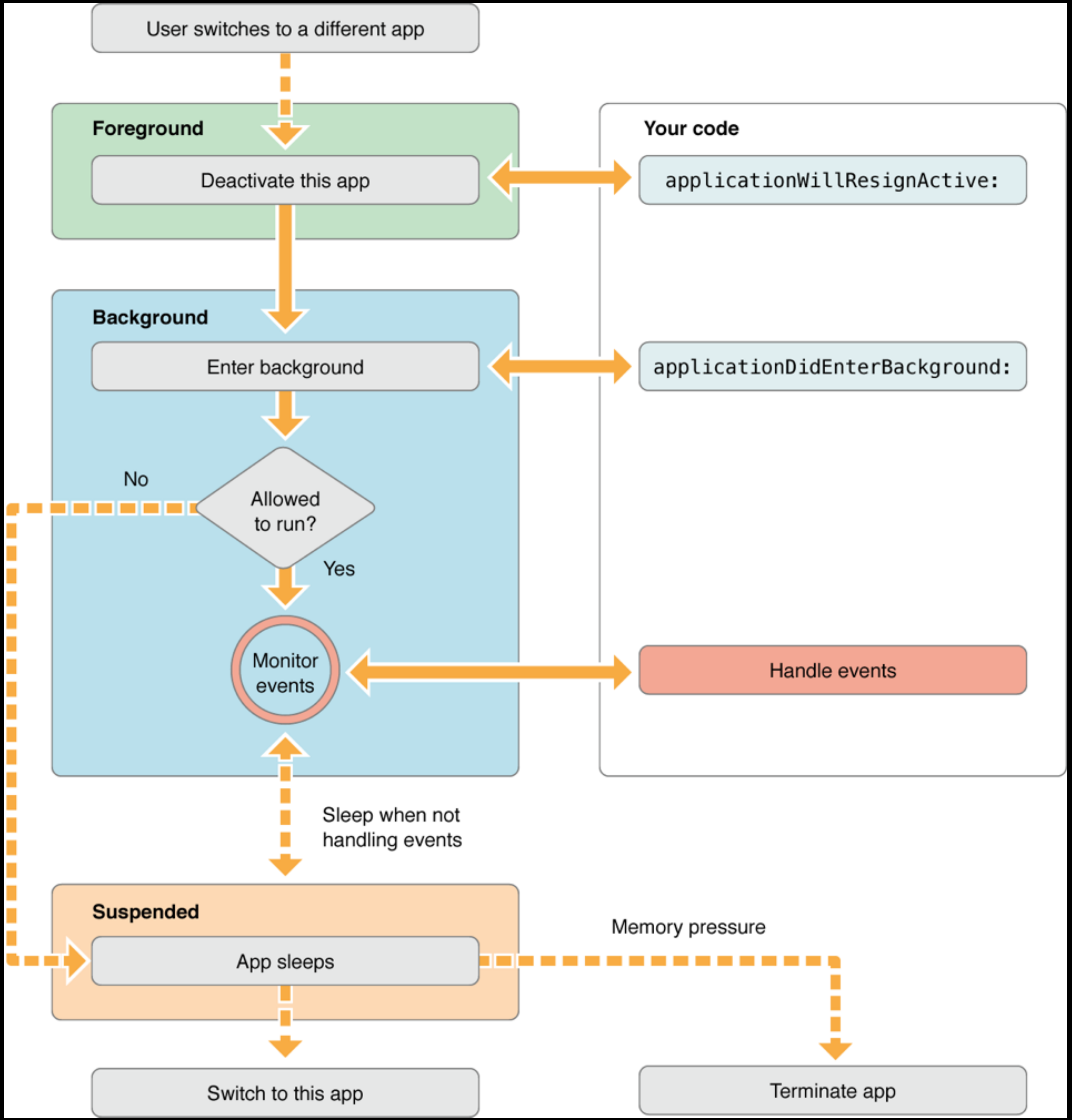
Responding to interruptions

- When an alert based interruption occurs (phone call,text message, system message) your app temporarily moves into the inactive state so the system can send a prompt to a user.
- The app remains in this state until the user dismisses the prompt.
- After the prompt dismisses, your app is either returned to active or move to background state.
- In response to this workflow, your app should do the following things in applicationWillResignActive if applicable to your app:
    - Stop timers or periodic tasks
    - Stop running queries
    - Don’t initiate any new tasks.
    - Pause move playback (except when over airplay)
    - Pause the game
    - suspend any non critical operation or dispatch queues.

Moving to the Background

- When user presses home button, sleep/wake button, or launches another app, the foreground app is transitioned to the inactive state and then background state.
- first applicationWillResignActive is called, and then applicationDidEnterBackground.
- Most apps move to suspended state after applicationDidEnterBackground returns or shortly there after.
- Apps that request specific background tasks may continue to run for a while longer.
- New in iOS8 : Location apps are automatically relaunched after the user kills it!

Moving to the Background



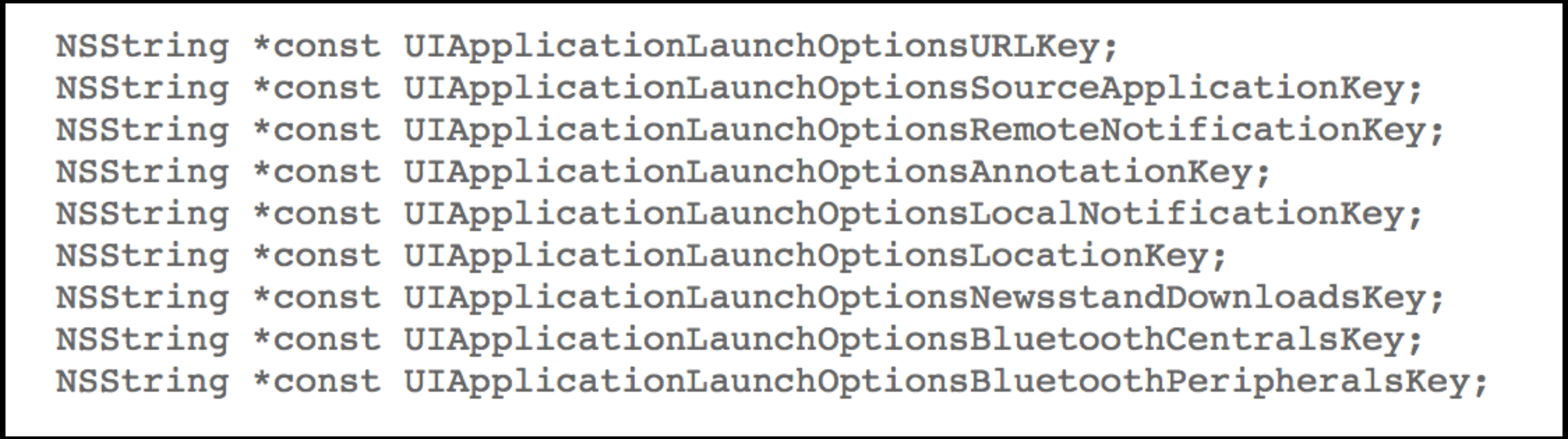
What to do when moving to background

- Say cheese: When the applicationDidEnterBackground method returns, the system takes a picture of your app’s user interface and uses the image for transitions. So clear out any sensitive data.
- Save User Data: all your unsaved data should be saved to disk, since your app could be quietly killed while in the suspended state.
- Free up memory: apps are killed based on how much memory they are taking up, so release as much memory as possible! (big images especially)

DidFinishLaunching Dictionary

- This dictionary will usually be empty if the user launches the app on their own.

Possible Keys:

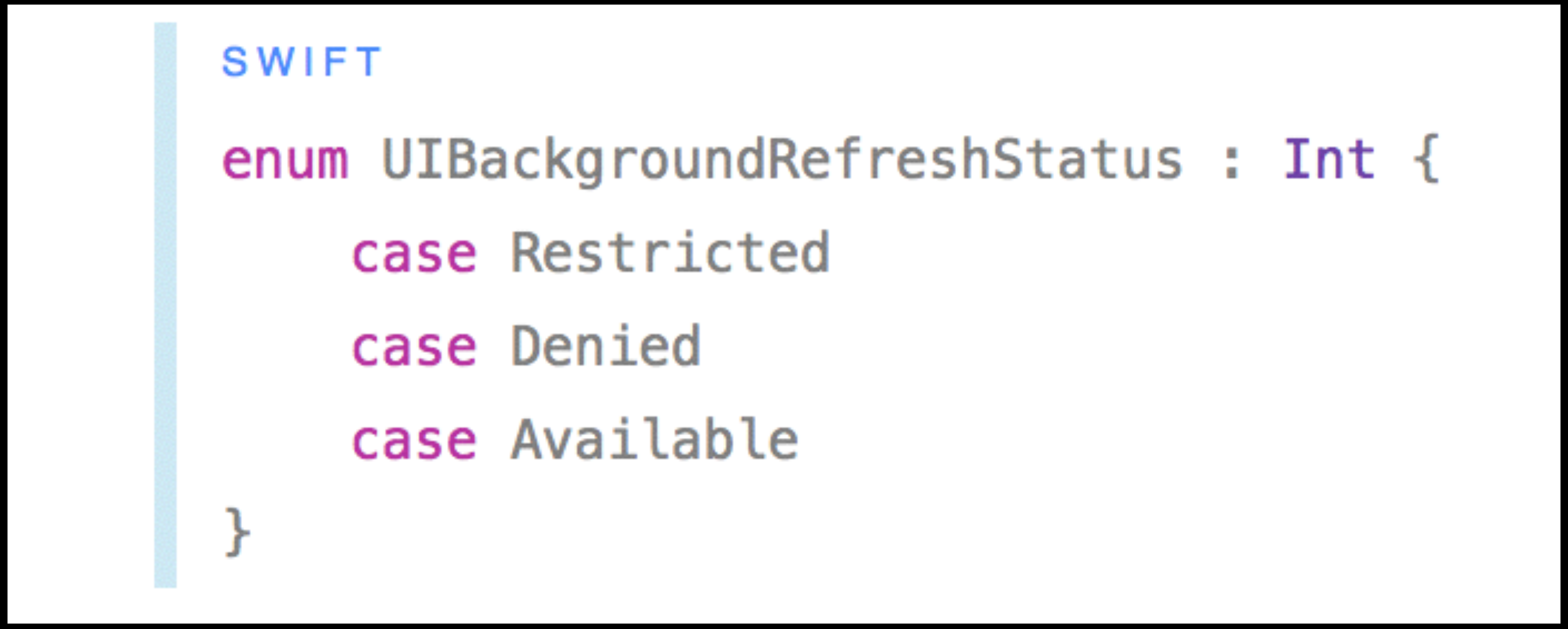


```
NSString *const UIApplicationLaunchOptionsURLKey;
NSString *const UIApplicationLaunchOptionsSourceApplicationKey;
NSString *const UIApplicationLaunchOptionsRemoteNotificationKey;
NSString *const UIApplicationLaunchOptionsAnnotationKey;
NSString *const UIApplicationLaunchOptionsLocalNotificationKey;
NSString *const UIApplicationLaunchOptionsLocationKey;
NSString *const UIApplicationLaunchOptionsNewsstandDownloadsKey;
NSString *const UIApplicationLaunchOptionsBluetoothCentralsKey;
NSString *const UIApplicationLaunchOptionsBluetoothPeripheralsKey;
```

- When an app is launched in the background, it still loads code like normal. It will load your initial view controller into memory (which runs view did load!)

`UIApplication.sharedApplication().backgroundRefreshStatus`

- Every app either has the ability to be launched into the background so it can perform background tasks or it doesn't.
- Check this this property to see if this is available, and warn the user if it isn't but your app replies on this.
- Users can disable background refresh on specific apps in Settings>General>Background Refresh


SWIFT

```swift
enum UIBackgroundRefreshStatus : Int {
    case Restricted
    case Denied
    case Available
}
```

App state behaviors

- Apps behave different in the background and foreground.
- The OS purposely limits what your app can do while its in the background, because it always gives resource priority to the foreground app.
- Your app is always notified when it transitions from background to foreground and vice versa.

Background/Terminated & Location services

- Your app can be launched in the background if it has signed up to receive certain location events.
- Region monitoring counts as one of these types of events
- In order to allow your app to be relaunched in the background, you need 2 keys in your plist:
  - Required device capabilities = [“location-services”]
  - Required background modes = [“App registers for location updates”]

Demo

Local & Push Notifications

Notifications

- “Local and push notifications are ways for an application that isn’t running in the foreground to let its users know it has information for them”
- Local and push look and sound the same.
- Can be displayed as an alert message and/or badge icon.
- Can play a sound.
- Not related to NSNotificationCenter!

Push vs Local

- Local notifications are scheduled by an app and delivered on the same app. Everything is done locally.
- Push notifications are sent by your server to the Apple Notification service, which pushes it to the device(s).
- While they appear the exact same to the user, they appear different to your app.

Launching with Notifications

- If your app is in the foreground, you will receive either application:didRecieveRemoteNotification: or application:didRecieveLocalNotification: in the app delegate.
- In that case, the system does not automatically show a popup or indication of a notification, **it is completely up to you notify the user of the event.**
- If your app is not in the foreground or not running, the system will display either a notification or a badge by your app icon. When the user launches the app, you need to check the launch dictionary for that information.

Local notifications

- Suited for time based or location based behaviors.
- Local notifications are instances of UILocalNotification
- 3 Properties:
  - Scheduled Time: Known as the fire date. Can set the time zone as well. Can also request it be rescheduled to repeat at regular intervals.
  - Notification Type: The alert message, the title of action button, the icon badge number, and a sound to play.
  - Custom Data: dictionary of custom data
- Each app limited to 64 scheduled local notifications.

Local notifications work flow

1. Create an instance of UILocalNotification
2. Set the fireDate property.
3. Set the alertBody (message) property, alertAction property(title of button or slider), applicationIconBadgeNumber property, and soundName property.
4. Optionally set any custom data you want with userInfo property
5. Schedule the delivery by calling scheduleLocalNotification: on UIApplication. Or you can fire it immediately by calling presentLocalNotificationNow:

- You can cancel local notifications with cancelLocalNotifcation: or cancel all with cancelAllLocalNotifications:

Reacting to a Notification when your app is not in the foreground.

1. The system presents the notification, displaying the alert, badge, and/or playing the sound.
2. As a result, the user taps the action button of the alert, or taps the applications icon with the badge.
3. If the user tapped the action button, the app is launched and the app calls its delegate's application:DidFinishLaunchingWithOptions: method. It passes in the notification payload in the info dictionary.

Reacting to a Notification when your app is in the foreground.

1. The application calls its delegate application:didReceiveRemoteNotification: method or application:didReceiveLocalNotification method and passes in the notification payload.
2. Remember its up to you notify the user of the event, the system wont play any sounds or show the popup alert.

Demo

Map Overlays

Map Overlays

- “Overlays offer a way to layer content over an arbitrary region of the map”
- Overlays are usually defined by multiple coordinates, which is different from the single point of an annotation.
- Overlays can be contiguous or noncontiguous lines, rectangles, circles, and other shapes.
- Those shapes can then be filled and stroked with color.

Overlay overview

- Getting an overlay onscreen involves two different objects working together:
    - An overlay object, which is an instance of a class that conforms to the MKOverlay protocol. This object manages the data points of the overlay.
    - An overlay renderer, which is a subclass of MKOverlayRenderer that does the actual drawing of the overlay onto the map.

Overlay workflow

1. Create an overlay object and add it to the map view
2. return an overlay renderer in the delegate method mapView:rendererForOverlay:

Overlay objects

- Overlay objects are typically small data objects that simply store the points that define what the overlay should represent and other attributes.
- MapKit defines several concrete objects you can use for your overlay objects if you want to display a standard shaped overlay.
- Otherwise, you can use any class as an overlay object as long as it conforms to the MKOverlay protocol.
- The map view keeps a reference to all overlay objects and uses the data contained in those objects to figure out when it should be displaying those overlays.

Overlay objects

- Concrete objects: MKCircle, MKPolygon, MKPolyline
- MKTileOverlay: use if your overlay can be represented by a series of bitmap tiles
- Subclass MKShape or MKMultiPoint for custom shapes
- Use your own objects with the MKOverlay protocol

`OverlayRenderer`

- MapKit provides many standard overlay renderers for standard shapes.
- You don’t add the renderer directly to the map, instead the delegate object provides an overlay renderer when the map view asks for one.

OverlayRenderer Objects

- Standard shapes: MKCircleRenderer, MKPolygonRenderer, MKPolylineRenderer.
- Tiled: MKTileOverlayRenderer
- Custom shapes : MKOverlayPathRenderer
- For the most customization, subclass MKOverlayRenderer and implement your custom drawing code.

Demo