



Mobile Application Development

Lecture 4
iOS SDK



This work is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-nc-sa/4.0/).

Lecture Summary

- iOS operating system
- iOS SDK
- Tools of the trade
- Model-View-Controller
- MVC interaction patterns
- View Controllers



iOS

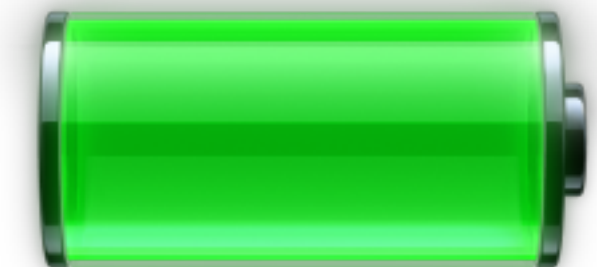
- iOS is Apple's mobile operating system, shipping with iPhone, iPod Touch, and iPad devices
- First released in 2007
- Current version is iOS8
 - released on September 2014
- This class covers iOS7
 - released on September 2014
 - runs on iPhone 4s/5/5c/5s, iPad 2/new iPad, iPod Touch 5th gen, iPad Mini
- iOS apps run in a UNIX-based system and have full support for threads, sockets, etc...

Memory management in iOS

- iOS uses a virtual memory system: each program has its own virtual address space
- iOS runs on constrained devices, in terms of available memory: memory is limited to the amount physical memory available
- iOS does not support paging to disk when memory gets full, so the virtual memory system releases memory if it needs more space
- Notifications of low-memory are sent to apps, so they can free memory

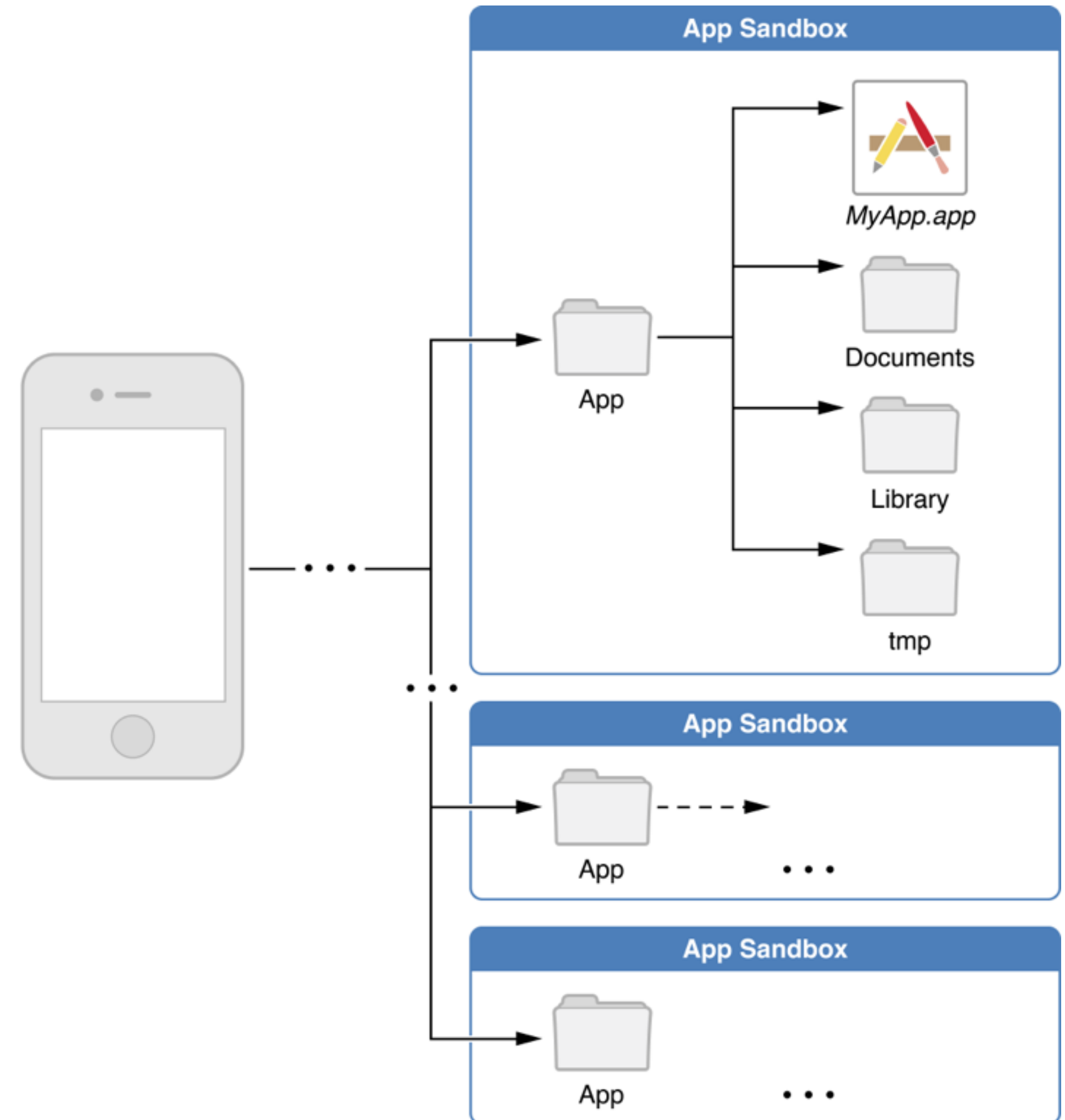
Multi-threading in iOS

- Since version 4, iOS allows applications to be run in the background even when they are not visible on the screen
- Most background apps reside in memory but do not actually execute any code
- Background apps are suspended by the system shortly after entering the background to preserve battery life
- In some cases, apps may ask the OS for background execution, but this requires a proper management of the app states

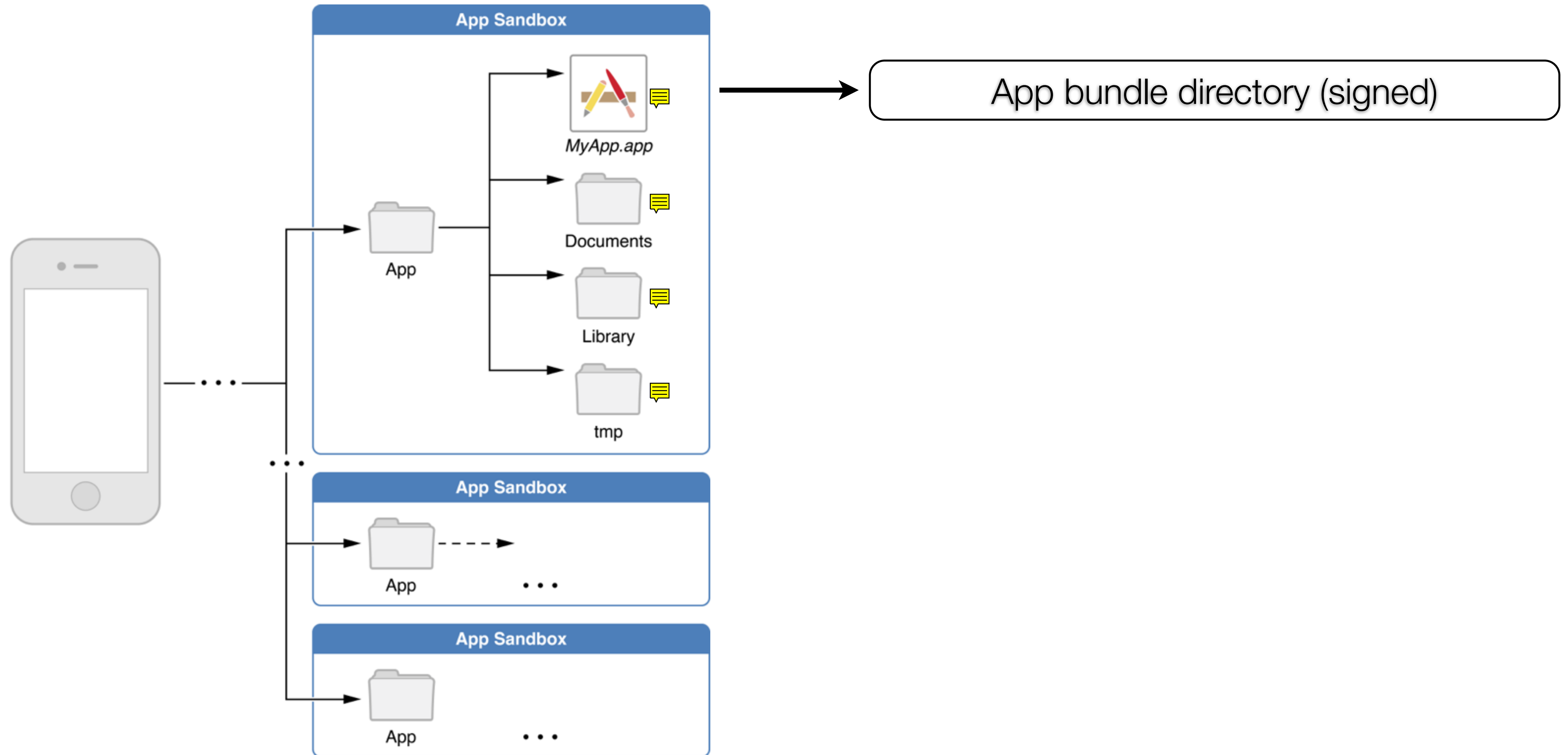


App sandbox

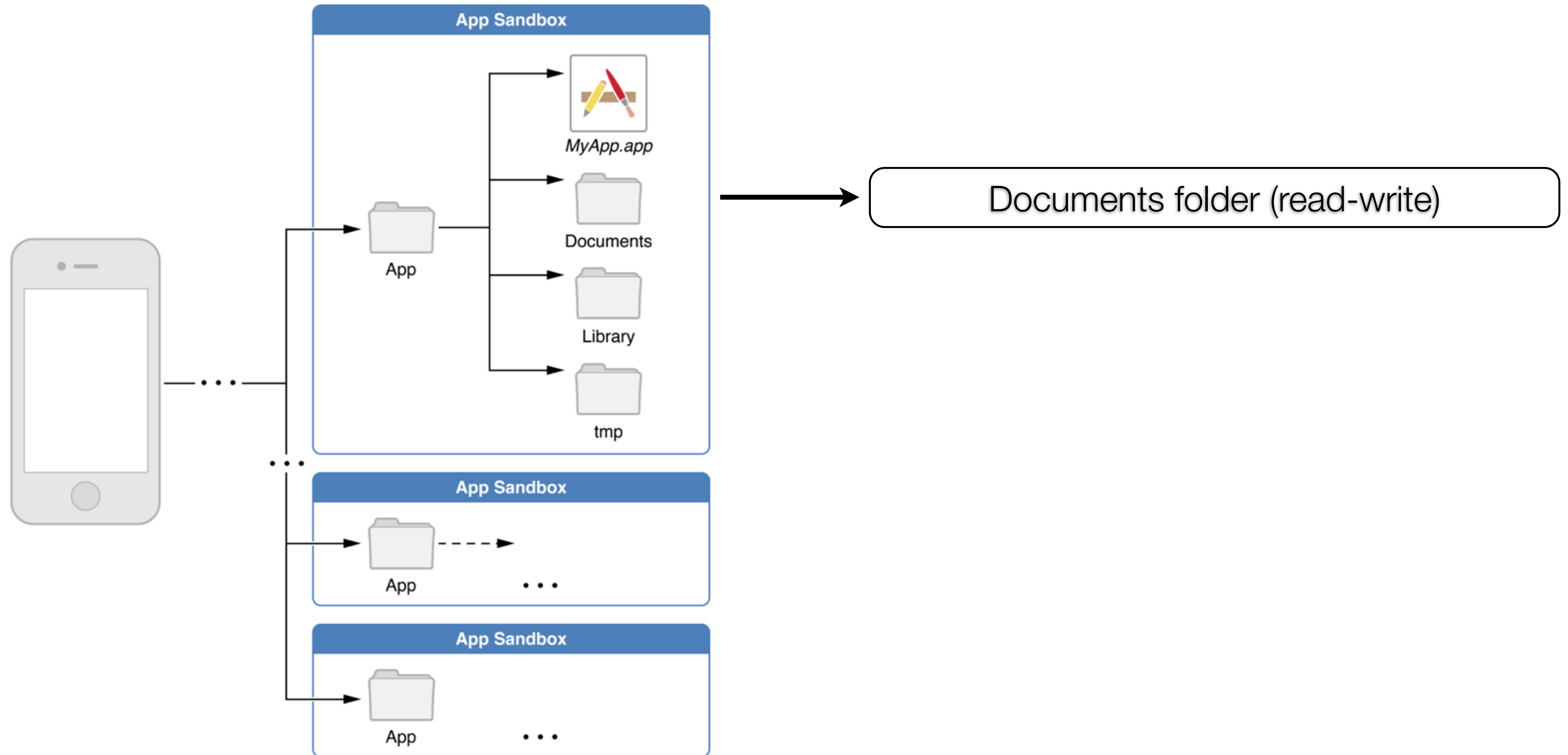
- For security reasons, iOS places each app (including its preferences and data) in a sandbox at install time
- A sandbox provides controls that limit the app's access to files, preferences, network resources, hardware, ...
- The system installs each app in its own sandbox directory, which can be seen as the home for the app and its data
- Each sandbox directory contains several well-known subdirectories for placing files
- **The sandbox only prevents the hijacked app from affecting other apps and other parts of the system, not the app itself**



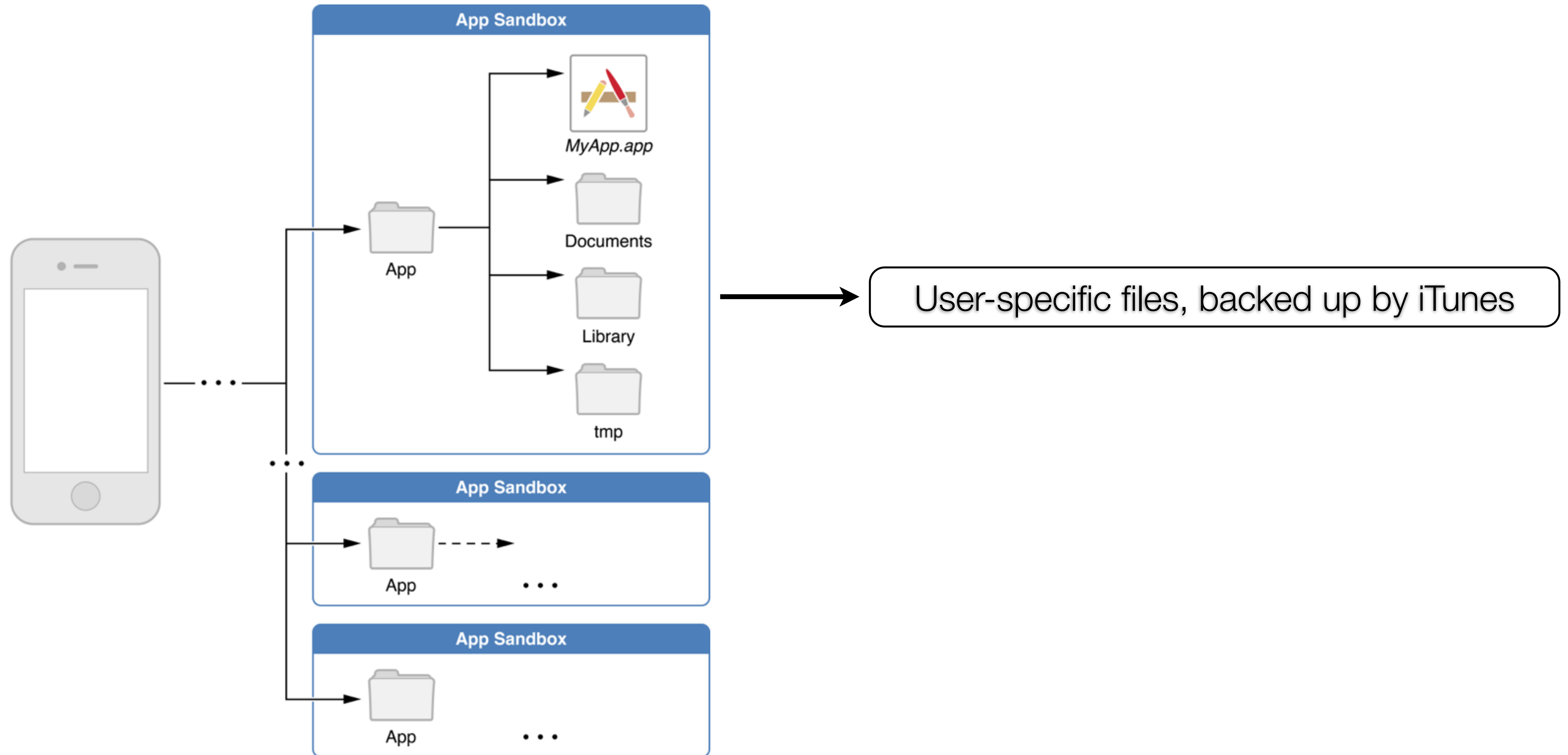
App sandbox



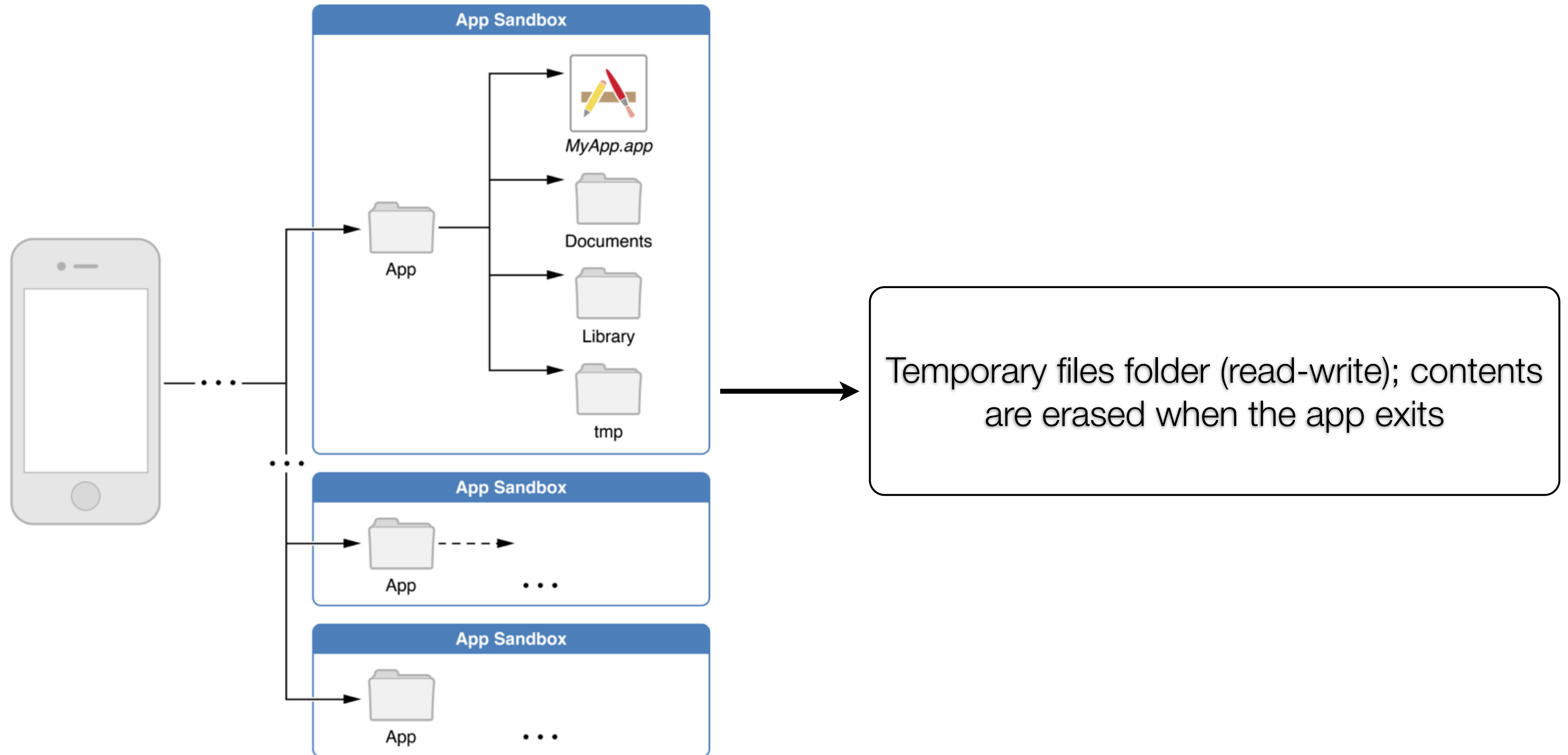
App sandbox



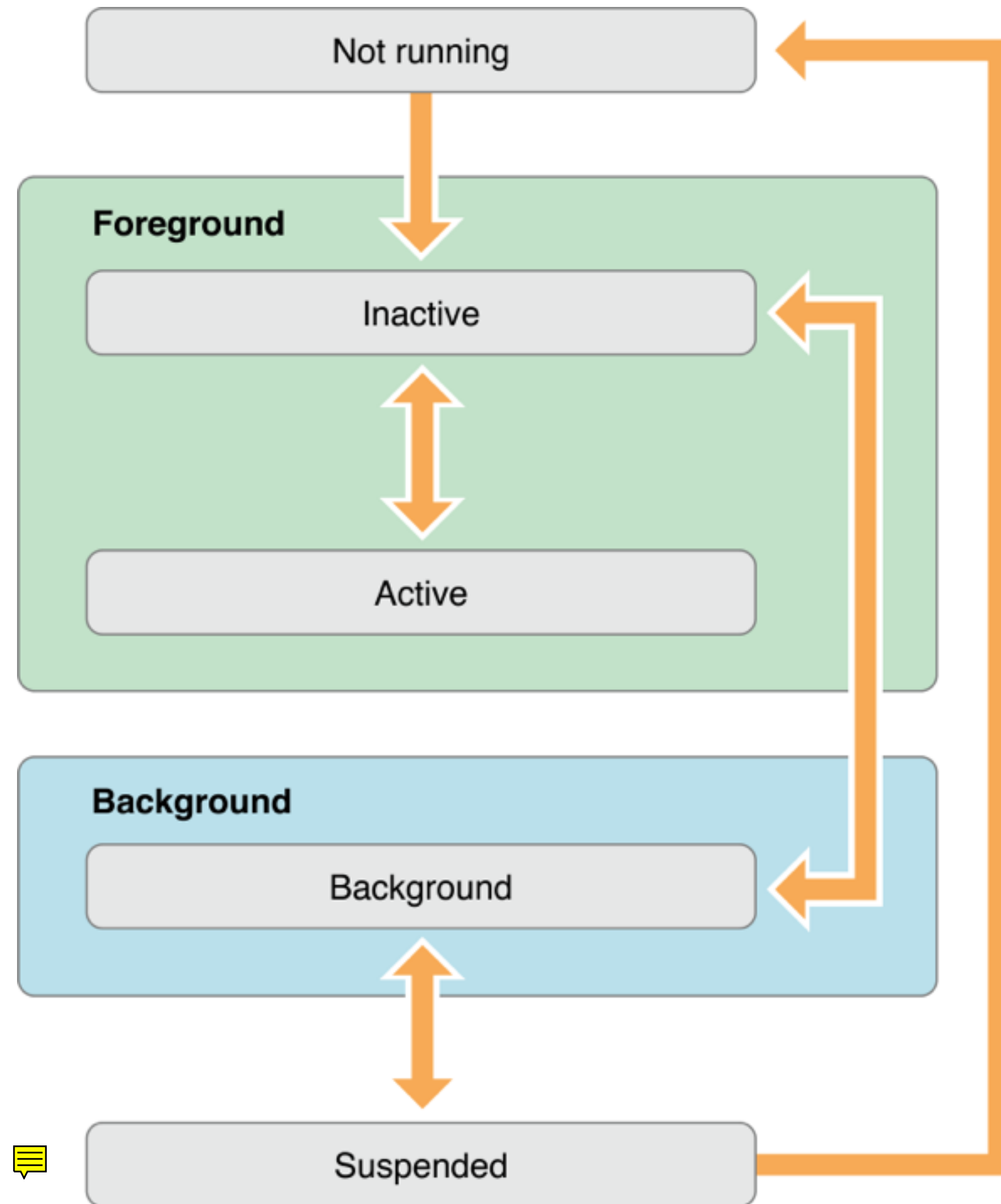
App sandbox



App sandbox

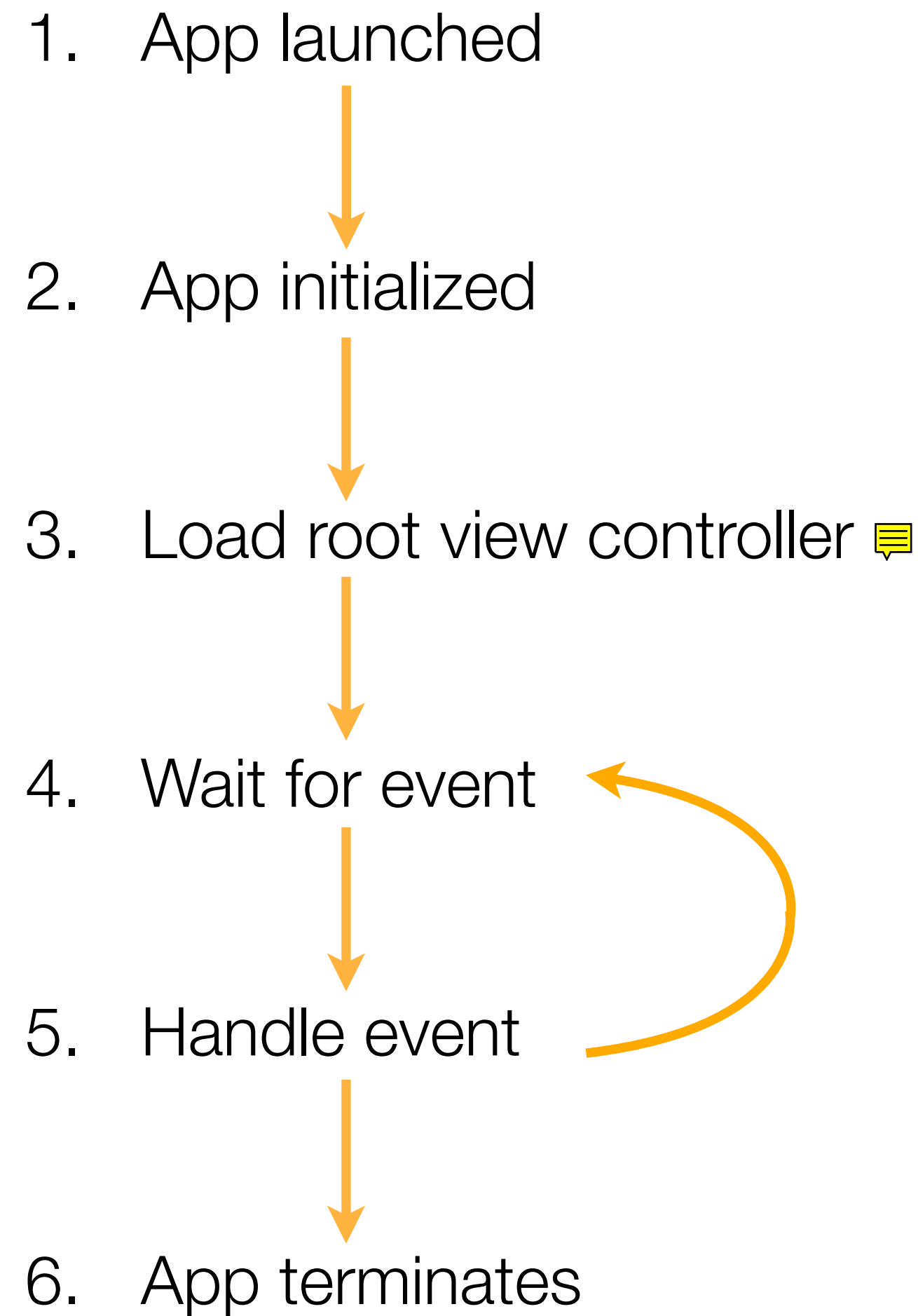


App Launch Cycle

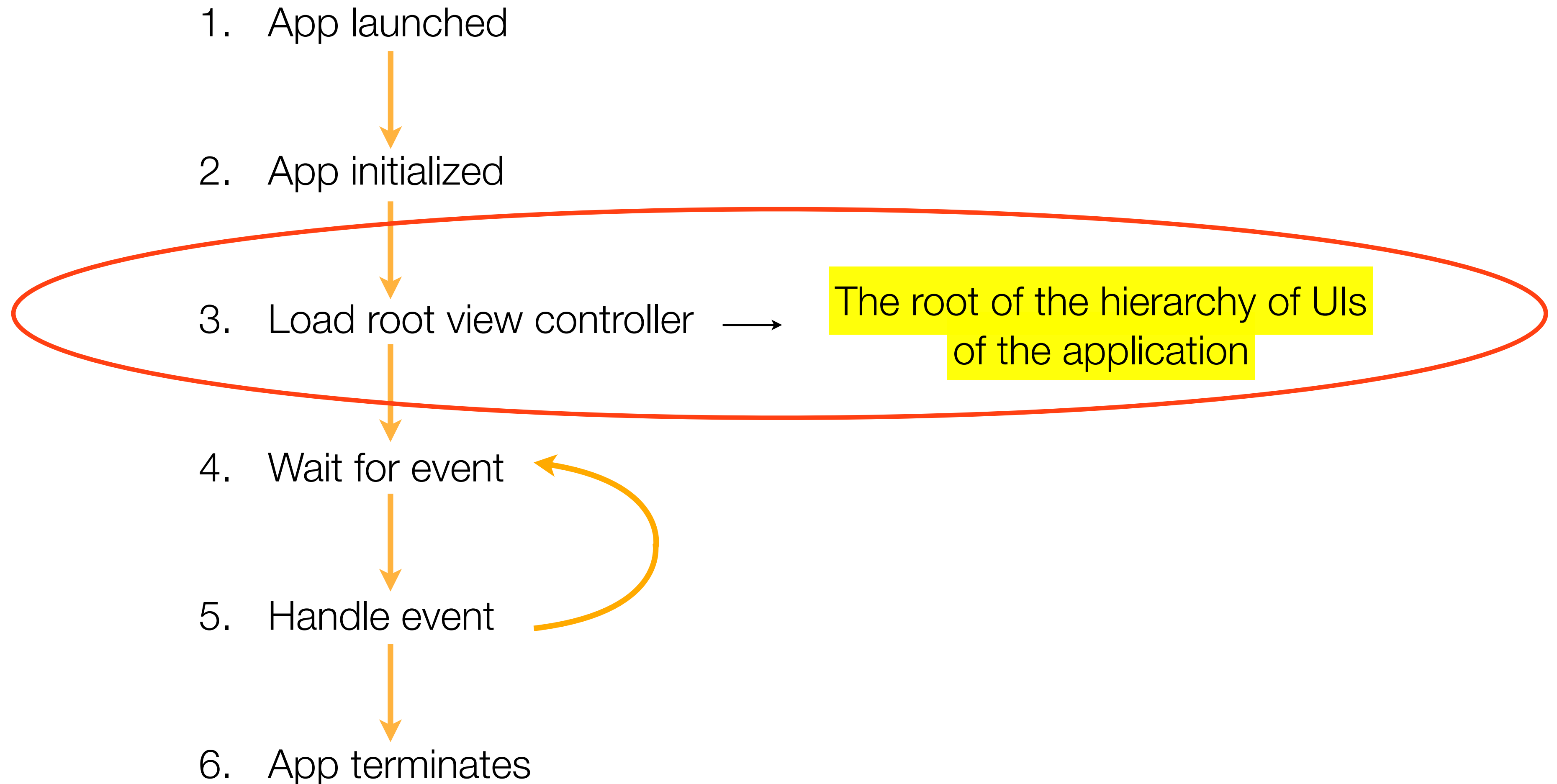


- When the app is launched it moves from the *not-running* state to the *active* or *background* state
- iOS creates a process and main thread for the app and calls the app's main function on that main thread (**main event loop**)
- The main event loop receives events from the operating system that are generated by user actions (e.g. UI-related events)

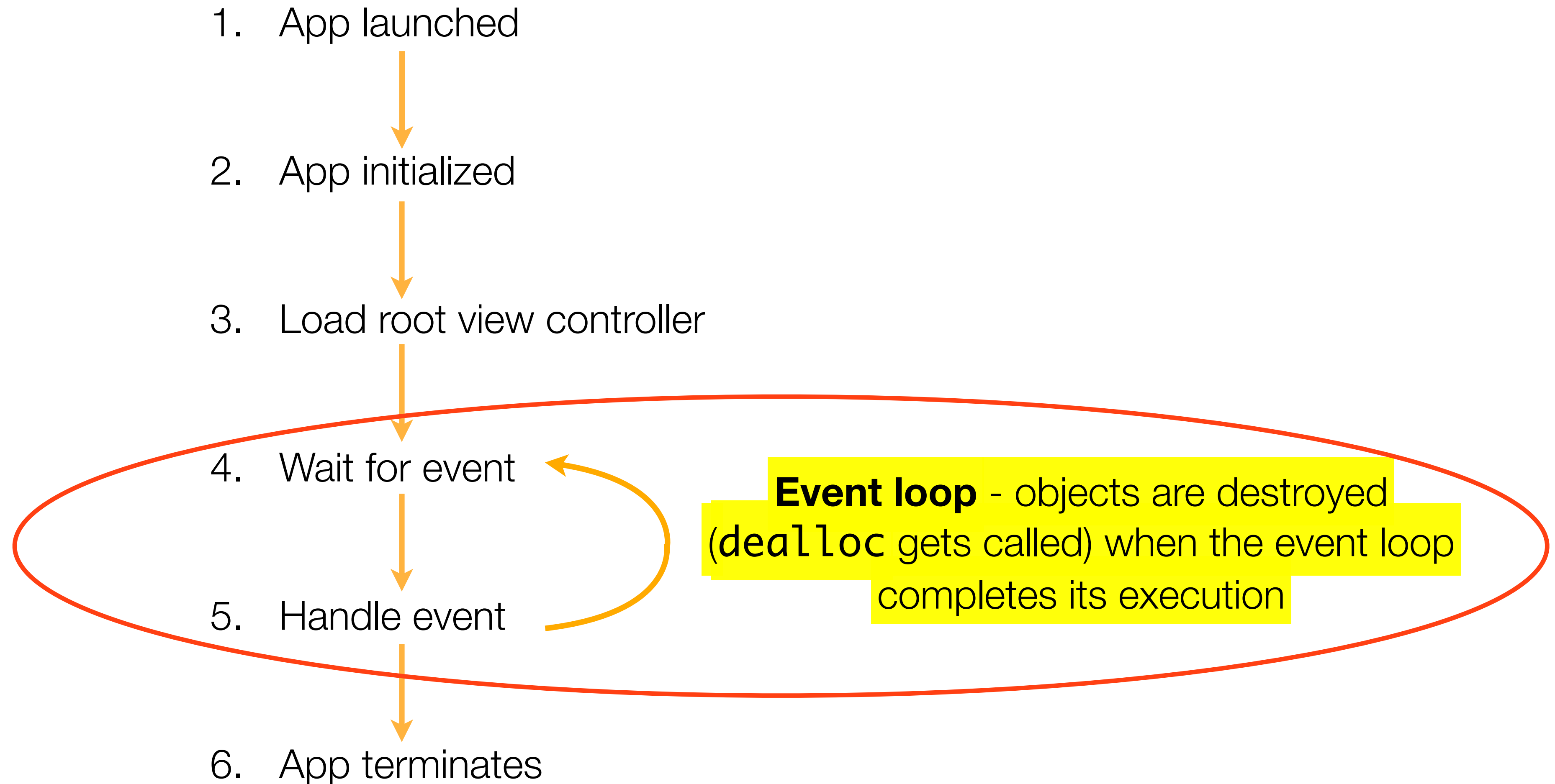
App Life Cycle



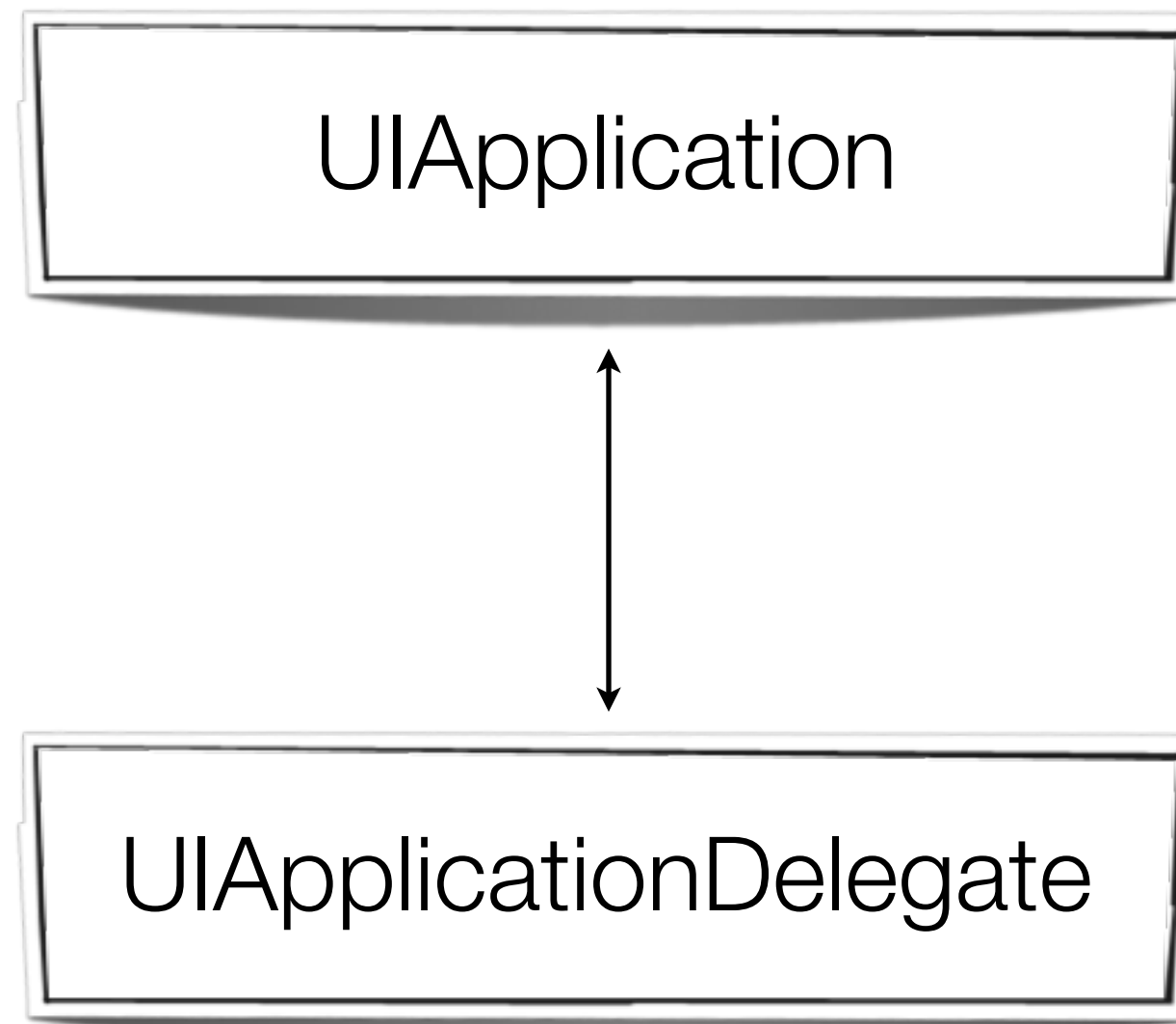
App Life Cycle



App Life Cycle



UIApplication and UIApplicationDelegate



- In iOS, the application is an instance of a class, called **UIApplication**
- The **UIApplication** is never touched directly, it is managed by the operating system
- Each application has a *delegate* object (conforming to the **UIApplicationDelegate** protocol) which receives messages when the app changes its state
- State transitions are accompanied by a corresponding call to the methods of the app delegate object
- These methods are a chance to respond to state changes in an appropriate way



UIApplicationDelegate protocol methods

`application:willFinishLaunchingWithOptions:` This method is the app's 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; use this method for any last minute preparation

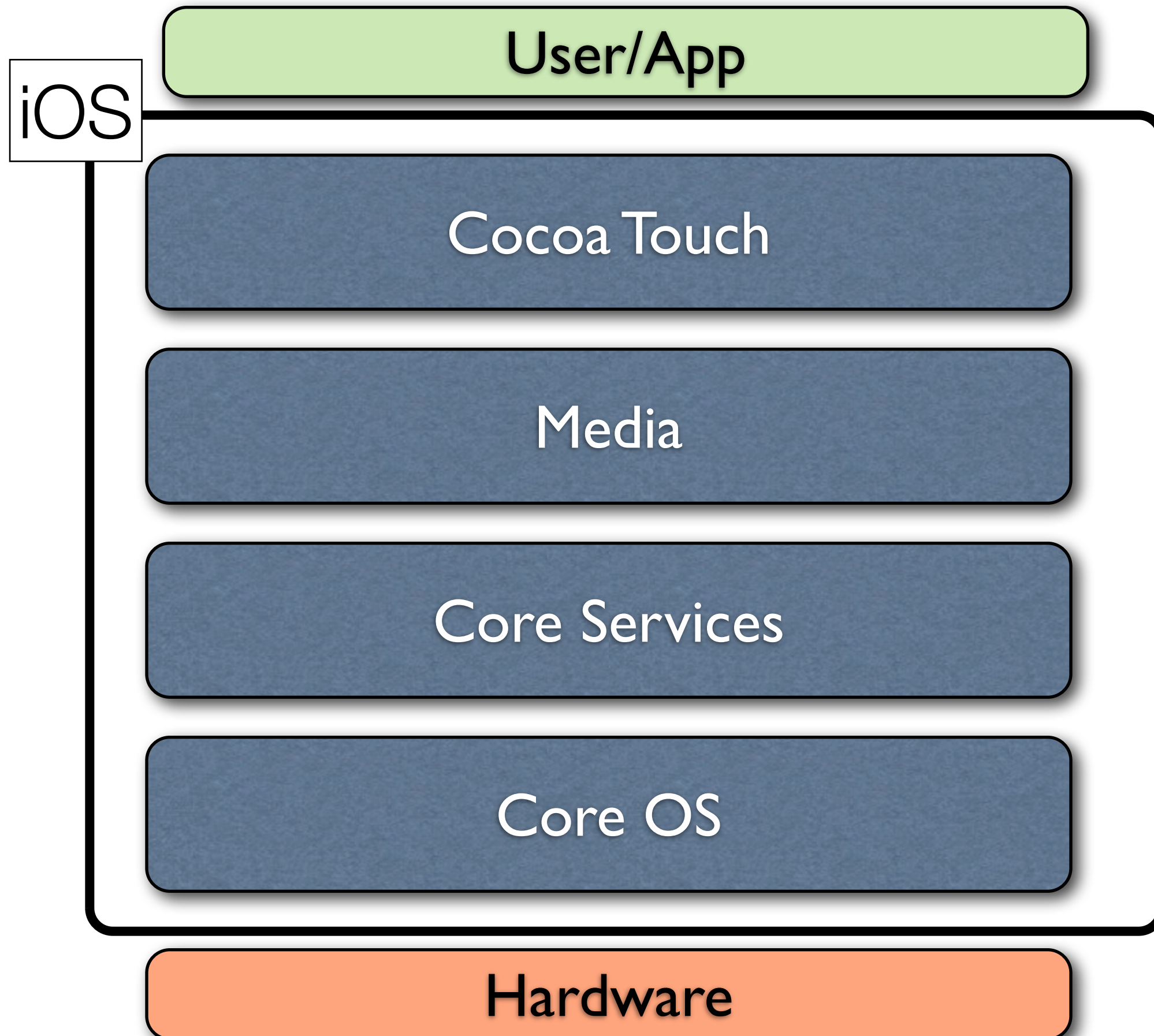
`applicationWillResignActive:` Lets you know that your app is transitioning away from being the foreground app; use this method to put your app into a dormant state

`applicationDidEnterBackground:` Lets you know that your app is now running in the background and may be suspended at any time

`applicationWillEnterForeground:` Lets you know that your app is moving out of the background and back into the foreground, but that it is not yet active

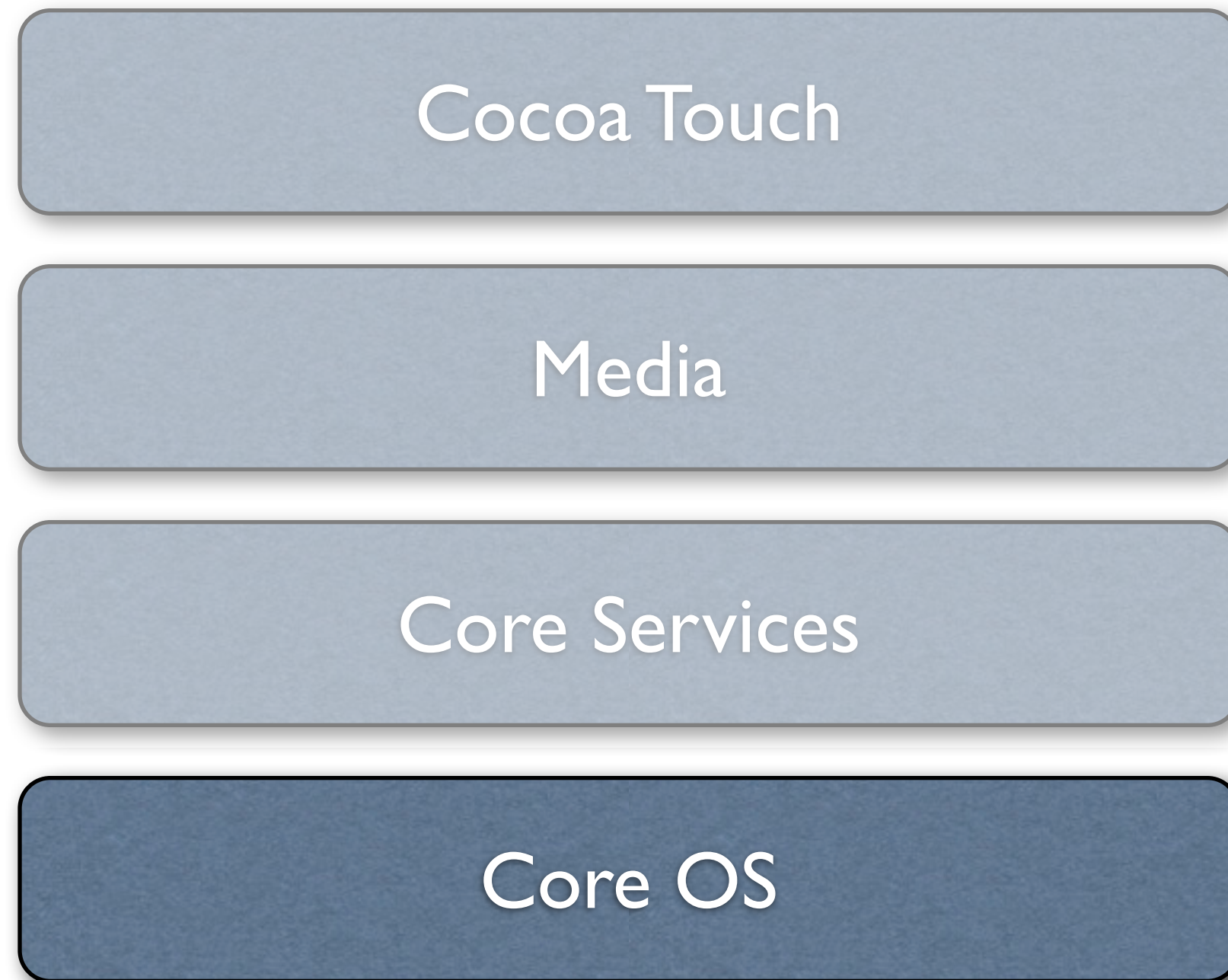
`applicationWillTerminate:` Lets you know that your app is being terminated; this method is not called if app is suspended

iOS Layers



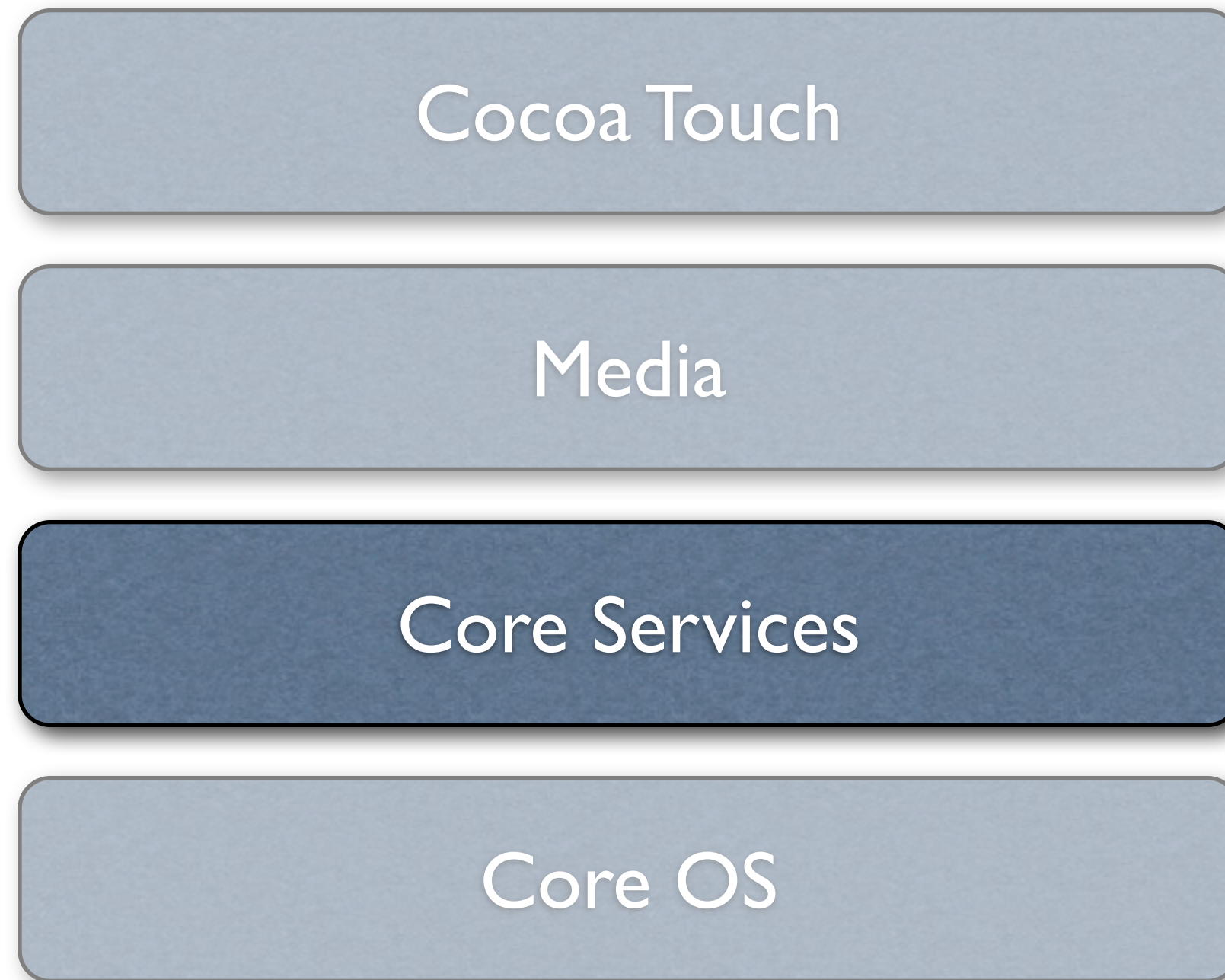
- The iOS architecture is layered
- iOS acts as an intermediary between the underlying hardware and the apps
- Apps communicate with the hardware through a set of well-defined system interfaces
- Lower layers contain fundamental services and technologies
- Higher-level layers build upon the lower layers and provide more sophisticated services and technologies
- iOS technologies are packaged as frameworks (especially **Foundation and UIKit frameworks**)
- A framework is a directory that contains a dynamic shared library and the resources (such as header files, images, and helper apps) needed to support that library

iOS Layers



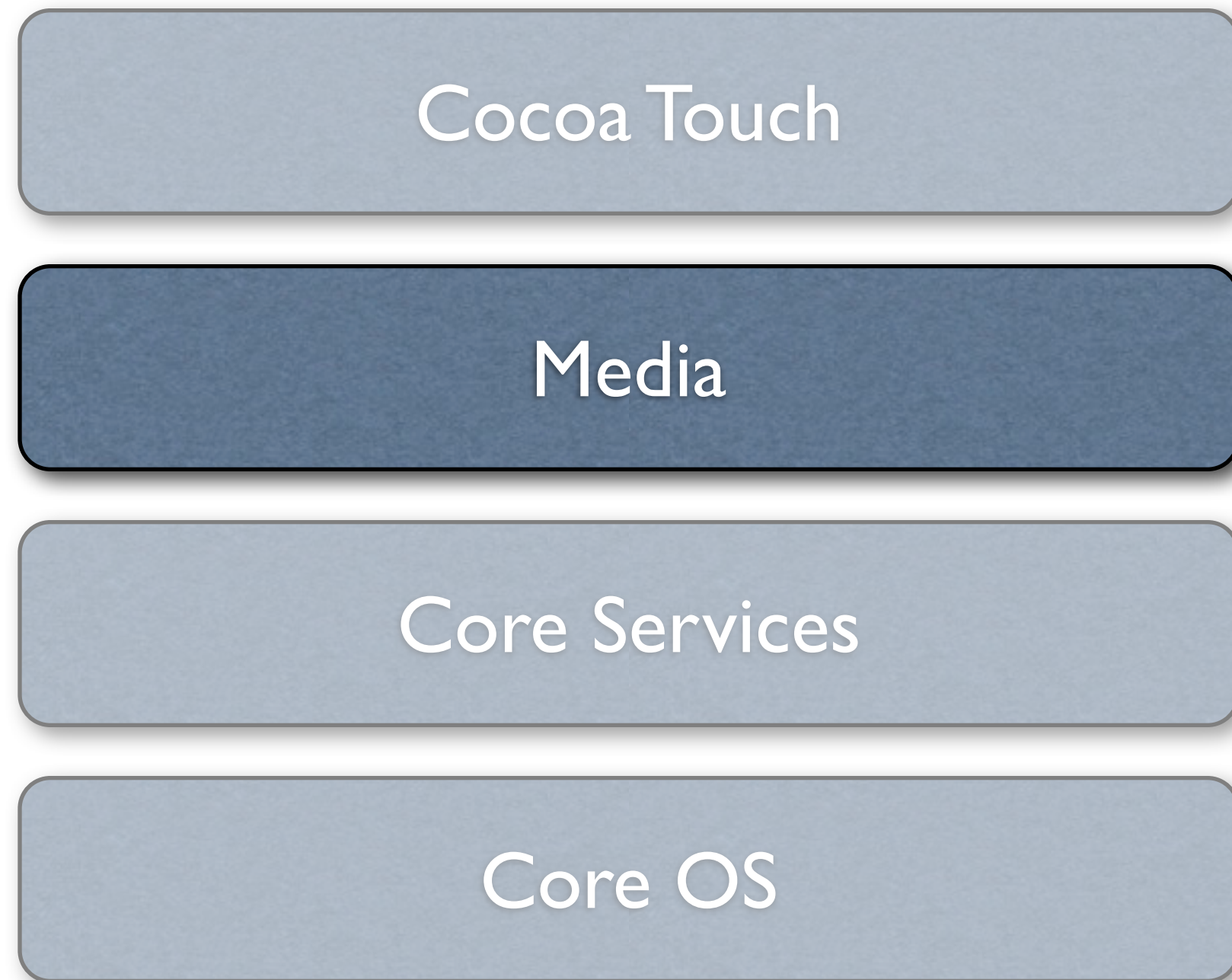
- Low-level features
- Main frameworks:
 - **Accelerate Framework:** vector and matrix math, digital signal processing, large number handling, and image processing
 - **Core Bluetooth Framework**
 - **Security Framework:** support for symmetric encryption, hash-based message authentication codes (HMACs), and digests
 - **System:** kernel environment, drivers, and low-level UNIX interfaces of the OS; Support for Concurrency (POSIX threads and Grand Central Dispatch), Networking (BSD sockets), File-system access, Standard I/O, Bonjour and DNS services, Locale information, Memory allocation, Math computations

iOS Layers



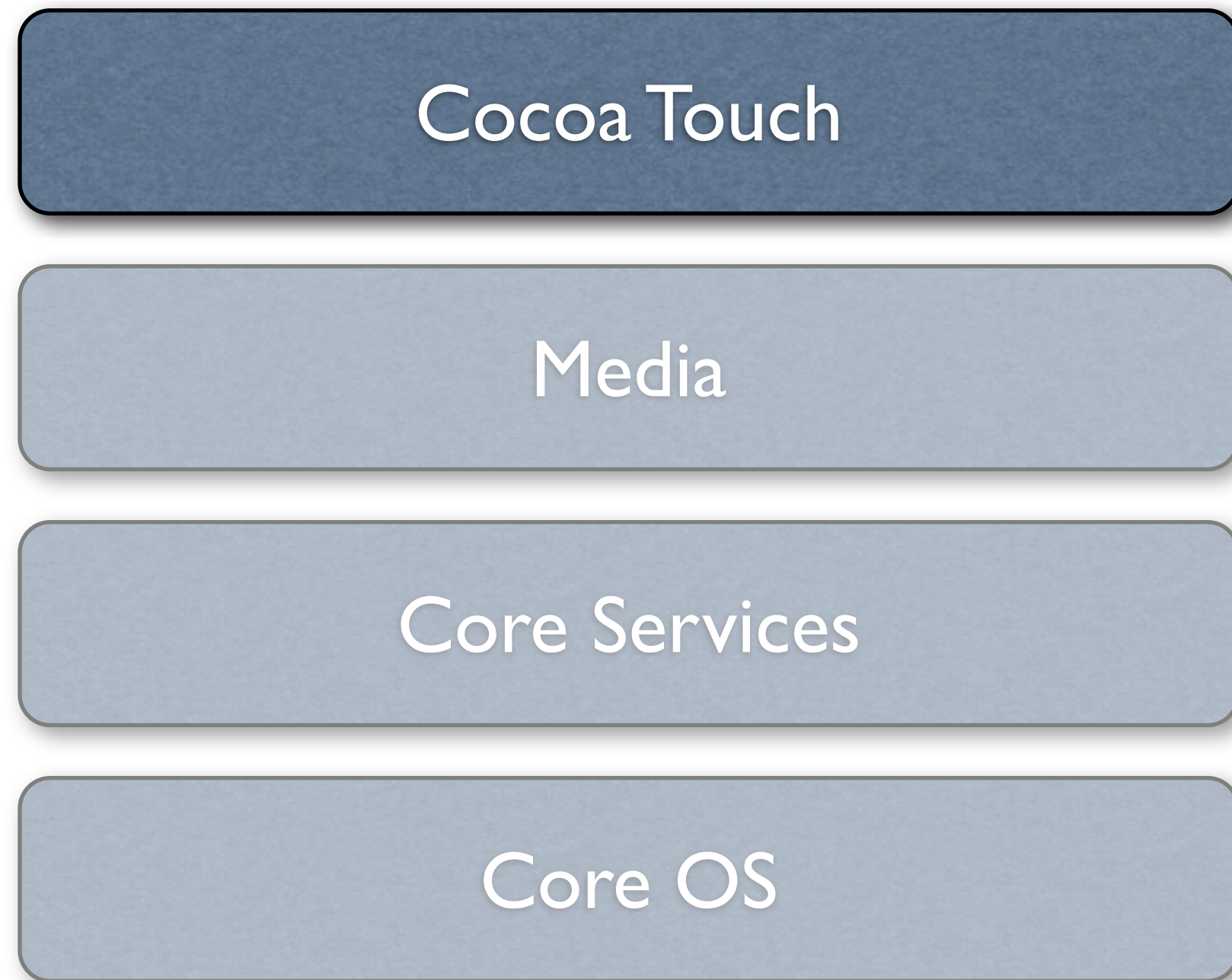
- Fundamental system services for apps
- Main frameworks:
 - **CFNetwork Framework**: BSD sockets, TLS/SSL connections, DNS resolution, HTTP/HTTPS connections
 - **Core Data Framework**
 - **Core Foundation Framework**: (C library) collections, strings, date and time, threads
 - **Core Location Framework**: provides location and heading information to apps
 - **Foundation Framework**: wraps Core Foundation in Objective-C types
 - **System Configuration Framework**: connectivity and reachability

iOS Layers



- Graphics, audio, and video technologies
- Main frameworks:
 - **AV Foundation Framework:** playing, recording, and managing audio and video content
 - **Media Player Framework:** high-level support for playing audio and video content
 - **Core Audio Frameworks:** native (low-level) support for handling audio
 - **Core Graphics Framework:** support for path-based drawing, antialiased rendering, gradients, images, colors
 - **Quartz Core Framework:** efficient view animations through Core Animation interfaces
 - **OpenGL ES Framework:** tools for drawing 2D and 3D content

iOS Layers



- Frameworks for building iOS apps
- Multitasking, touch-based input, push notifications, and many high-level system services
- Main frameworks:
 - **UIKit Framework:** construction and management of an application's user interface for iOS
 - **Map Kit Framework:** scrollable map to be incorporated into application user interfaces
 - **Game Kit Framework:** support for Game Center
 - **Address Book Framework:** standard system interfaces for managing contacts
 - **MessageUI Framework:** interfaces for composing email or SMS messages
 - **Event Kit Framework:** standard system interfaces for managing calendar events

UIKit Framework

- The UIKit framework provides the classes needed to construct and manage an application's user interface for iOS
- It provides an application object, event handling, drawing model, windows, views, and controls specifically designed for a touch screen interface
- UIKit provides:
 - Basic app management and infrastructure, including the app's main run loop
 - User interface management, including support for storyboards and nib files
 - A view controller model to encapsulate the contents of your user interface
 - Objects representing the standard system views and controls
 - Support for handling touch- and motion-based events

iOS SDK

- The iOS Software Development Kit (SDK) contains the tools and interfaces needed to develop, install, run, and test native apps
- Tools: **Xcode**
- Language: **Objective-C** (plus some C/C++)
- Libraries: **iOS frameworks**
- Documentations: **iOS Developer Library** (API reference, programming guides, release notes, tech notes, sample code...)

Xcode



- Xcode is the development environment used to create, test, debug, and tune apps
- The Xcode app contains all the other tools needed to build apps:
 - **Interface Builder**
 - **Debugger**
 - **Instruments**
 - **iOS Simulator**
- Xcode is used to write code which can be run on the simulator or a connected iDevice
- Instruments is used to analyze the app's behavior, such as monitoring memory allocation

Model-View-Controller

-  All iOS applications are built using the **MVC pattern**
- MVC is used to organize parts of code into clean and separate fields, according to the responsibilities and features of each of them
- This organization is extremely important because it provides a way to create applications that are easy to write, maintain, and debug
- Basically, the way that the iOS SDK is built, drives developers to build applications using MVC
- Understanding and enforcing MVC is 90% of the job when developing in iOS

Model-View-Controller

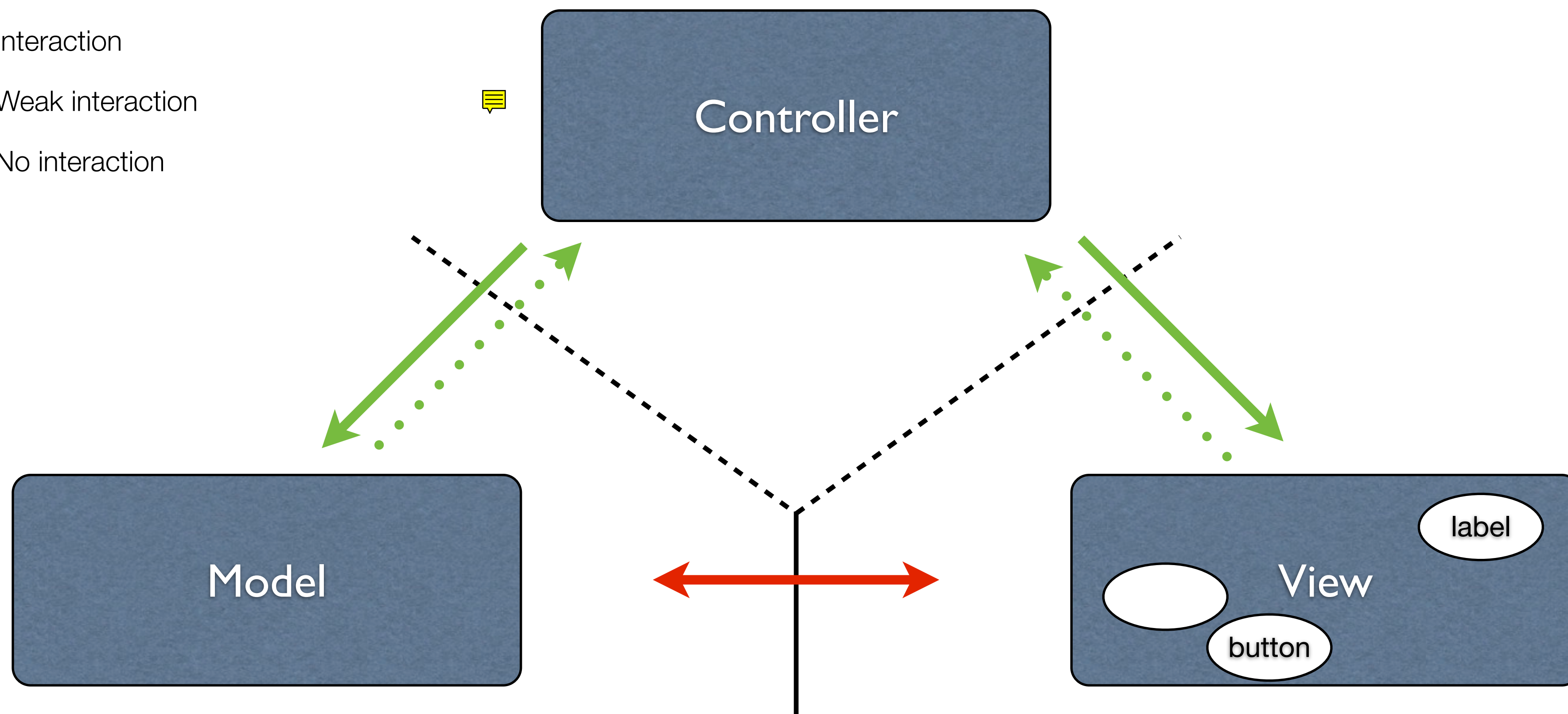
- MVC stands for Model-View-Controller
- An application's code can belong either to the model, to the view, or to the controller
-  The **Model** is the representation of the data that will be used in the application; the model is independent from the View, since it does not know how data will be displayed (e.g. iPod library)
-  The **View** is the user interface that will display the application's contents; the view is independent from the Model since it contains a bunch of graphical elements that can be used in any application (e.g. buttons, labels, sliders, ...)
- The **Controller** is the brain of the application: it manages how the data in the Model should be displayed in the View; it is highly dependent from the Model and the View since it needs to know which data it will handle and which graphical elements it will need to interact with
- The Controller coordinates and manages the application's UI logic

MVC interactions

 Interaction

 Weak interaction

 No interaction

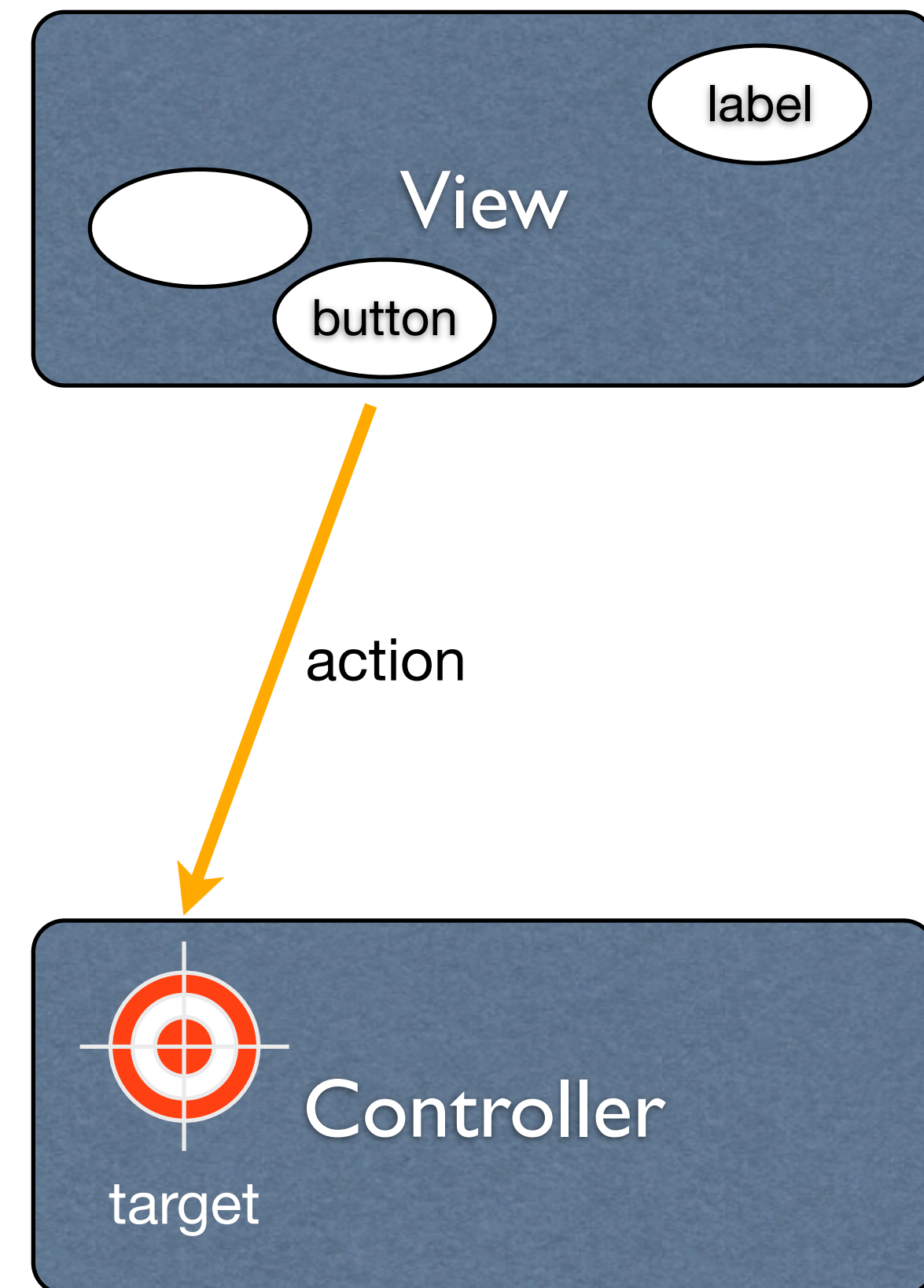


Model-View-Controller interactions

- The Model has direct interaction with neither the View (obviously) nor the Controller, since its responsibility is just to keep the data (e.g. a database)
- The Controller has direct interaction with the Model since it needs to retrieve and store the data
- The Controller has direct interaction with the View since it needs to update UI elements
- The Controller keeps a reference to UI elements it will use, called **outlets** (**IBOutlet**)
- The View has no direct interaction with neither the Model (obviously) nor the Controller, since its job is just to display UI elements on the screen

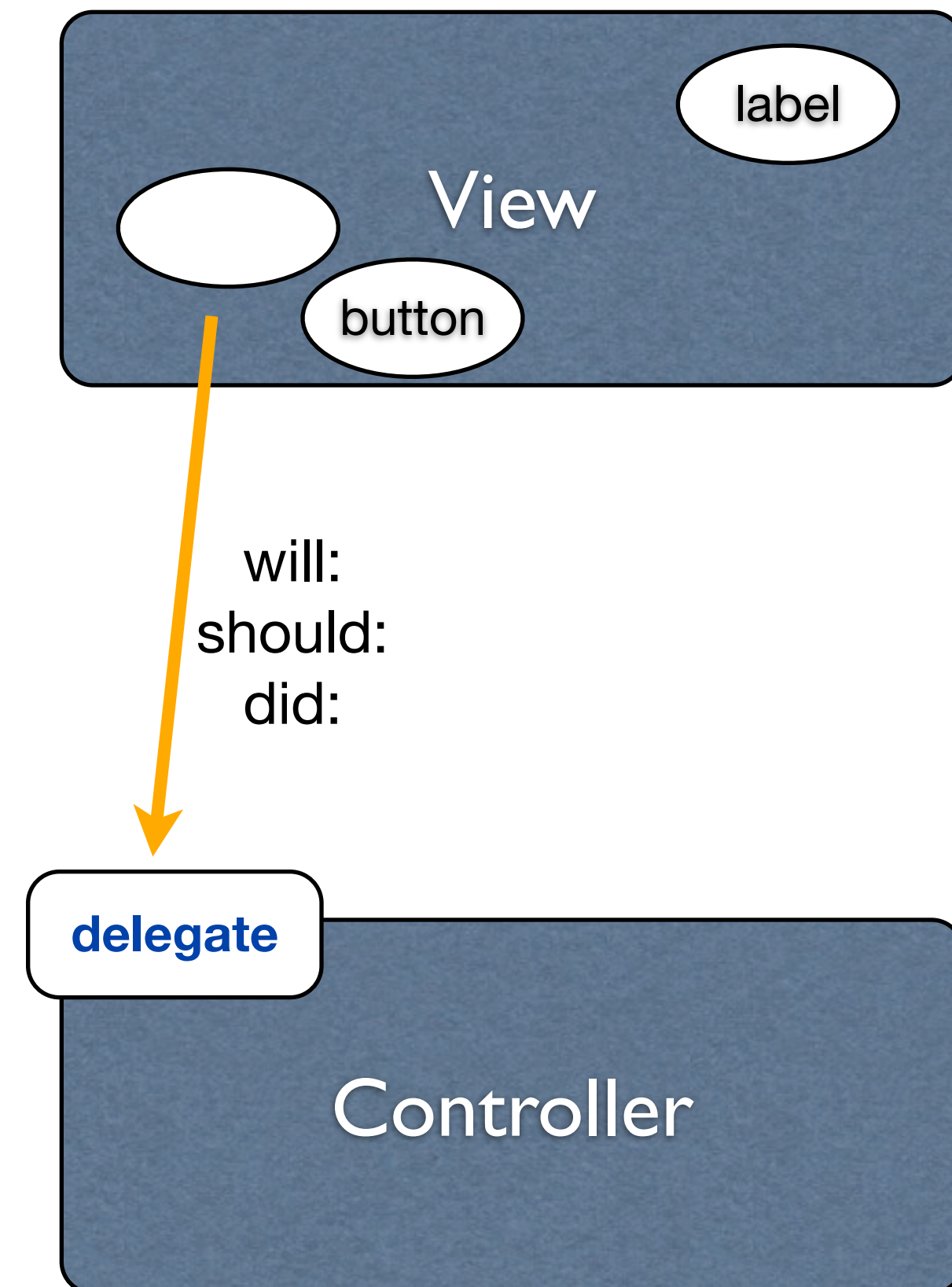
View-to-Controller interactions

- The View does not interact directly with the Controller, since View classes do not even know about the existence of the Controller
- However, the View should inform the Controller that certain events have occurred (e.g. a button has been clicked)
- The interaction between a View and its Controller occurs in a blind way, through **actions** (**IBAction**)
- The Controller can register to the View to be the **target** for an action; when the action is performed, the View will send it to the target
- This interaction model lets the View be totally independent from the Controller, yet allows the View to interact with the Controller



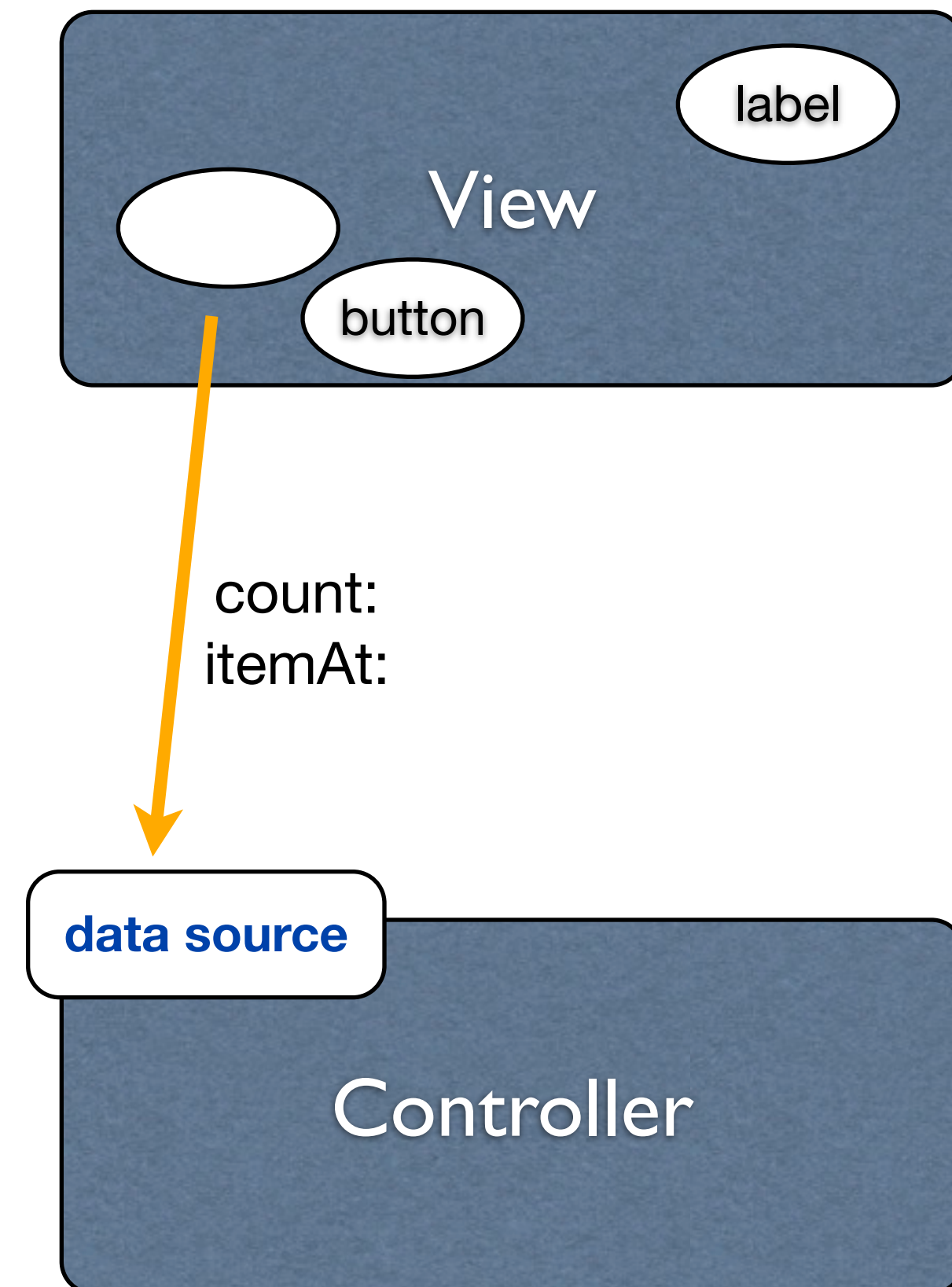
View-to-Controller interactions

- Some Views interact with the Controller, in order to coordinate (synchronize)
- When certain events **should**, **will**, or **did** occur (e.g. a list item was selected), the View must inform the Controller so that it can perform some operations
- This is called **delegation**: the Controller is the delegate, which means that the view passes the responsibility to the Controller to accomplish certain tasks
- There is a loose coupling between the View and the Controller
- Delegation is accomplished by using protocols



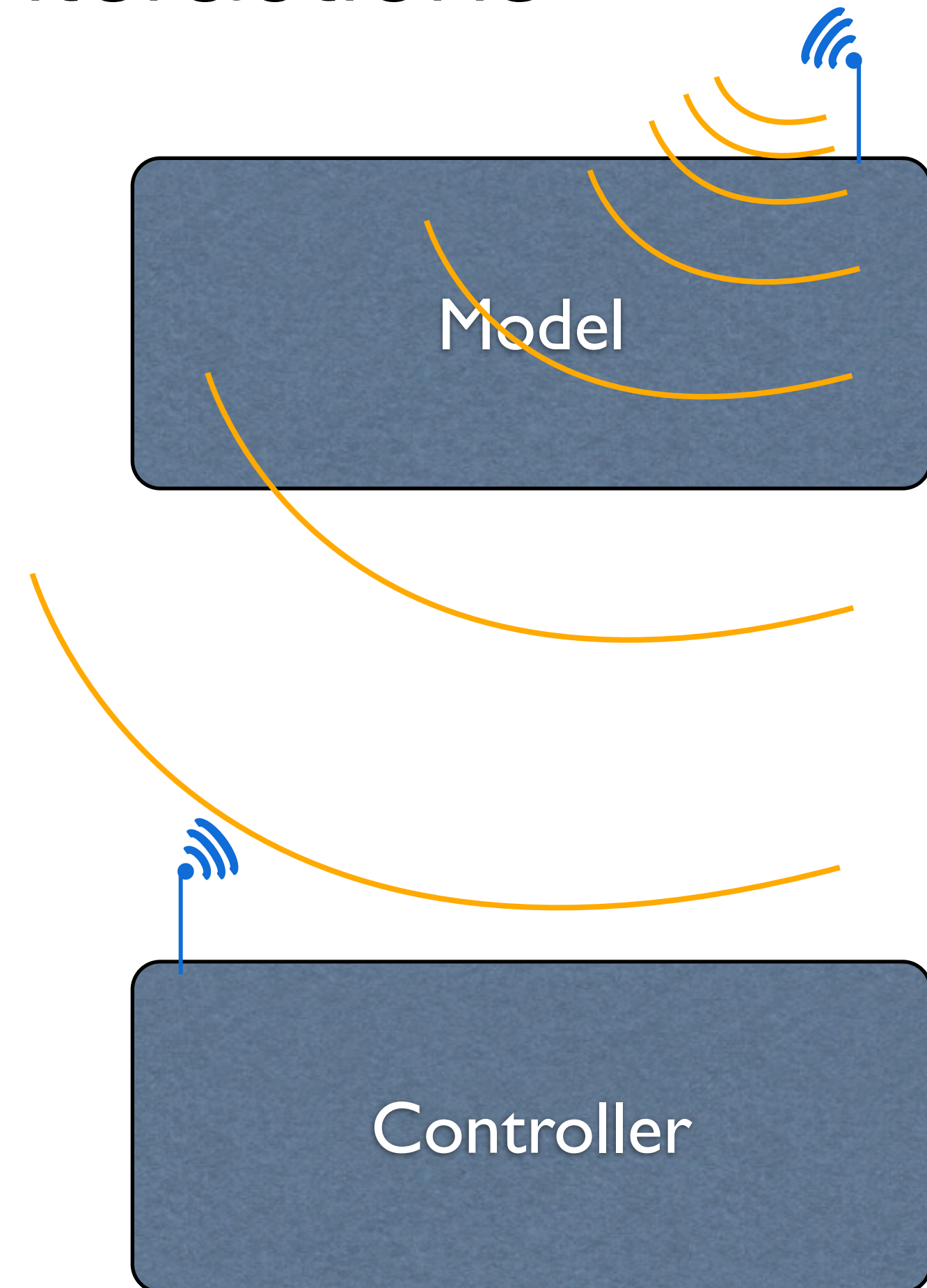
View-to-Controller interactions

- Some Views do not have enough information to be displayed directly (e.g. a list of elements)
- In general, Views do not own the data that they display, data belong to the Model
- The View needs the Controller to provide those data so that it can display them
- The Controller is a **data source** for the View
- Again, this is accomplished by using protocols
- Data source is indeed delegation, since the View is delegating the Controller to be the provider for the data to be displayed
- Data source is a protocol for providing data, delegate is a protocol for handling view-related events



Model-to-Controller interactions

- The Model cannot interact with the View, because it is UI-independent
- When data change, the Model should inform the Controller so that it can instruct the View to change what is being displayed
- The Model “broadcasts” the change event
- If the Controller is interested in the event, it will be notified
- This interaction occurs through **notifications** or **KVO (key-value observing)**



Model-View-Controller

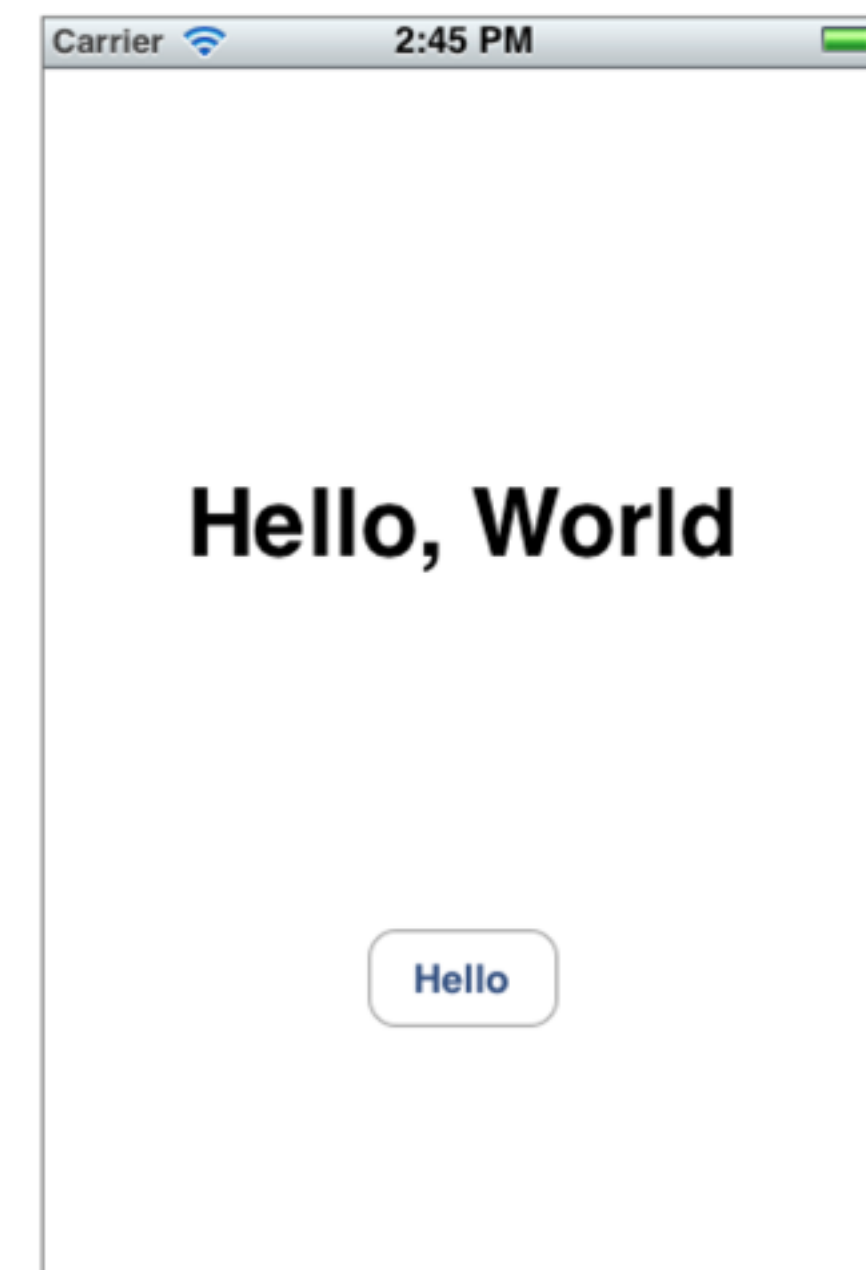
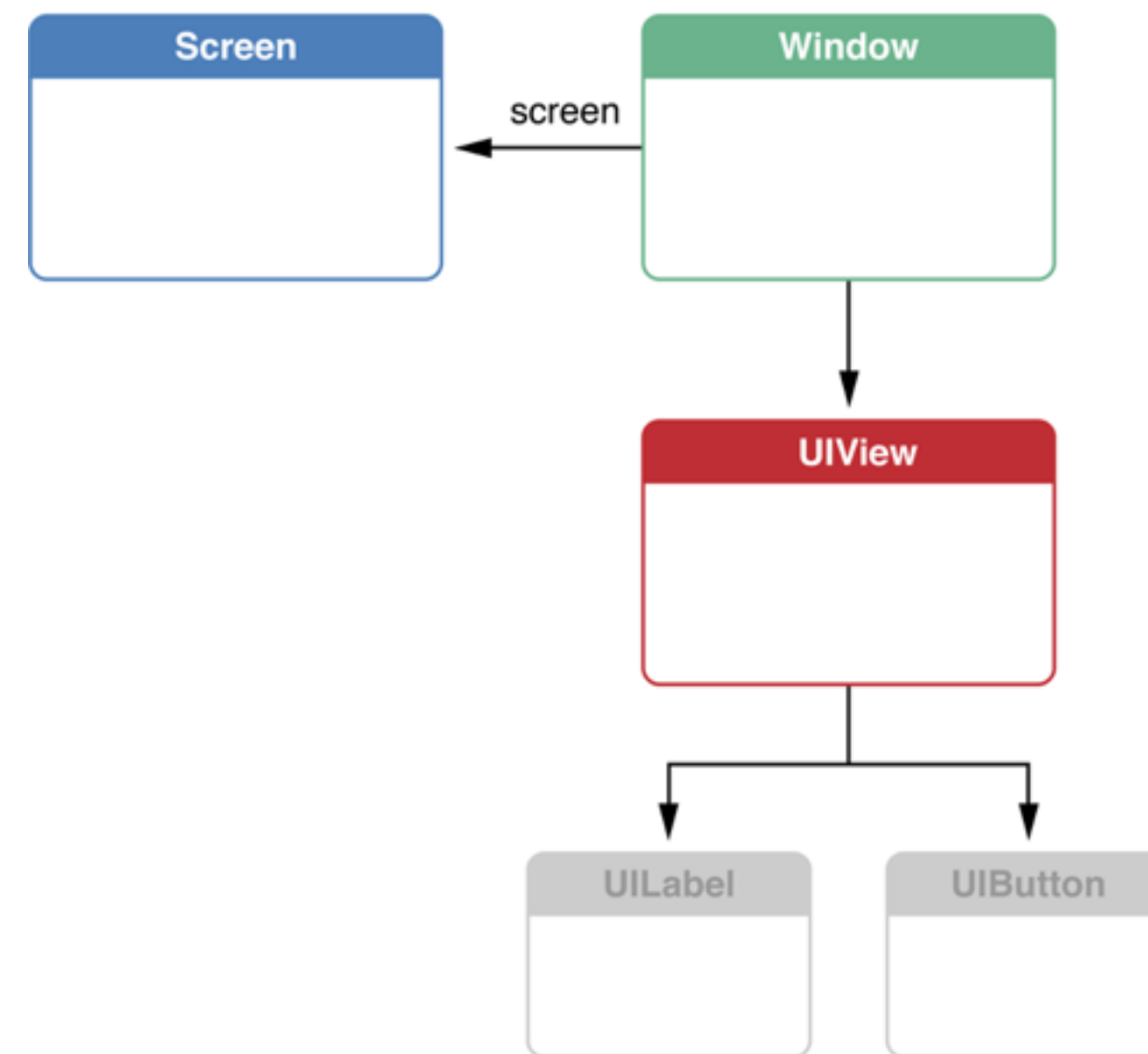
- The **Controller's** job is to **retrieve** and **format data** from the Model so that it can be displayed in the View
- **Most of the work when developing apps is done within the Controller(s)**
- **Complex applications require several MVC to come into play**, for instance when an event on a view causes another view to be displayed (typically, this is done by a **Controller interacting with other Controllers**)
- **Some parts of a MVC are other MVCs** (e.g. the tabs of a tabbed view are separate MVCs)

View Controllers

- View controller objects provide the infrastructure for managing content and for coordinating the showing and hiding of it
- By having different view controller classes control separate portions of user interface, the implementation of the user interface is broken up into smaller and more manageable units
- View controller objects represent the Controller part of the application's MVC

User interface: Screen, Window, and View

- **UIScreen** identifies a physical screen connected to the device
- **UIWindow** provides drawing support for the screen
- **UIView** objects perform the drawing; these objects are attached to the window and draw their contents when the window asks them to



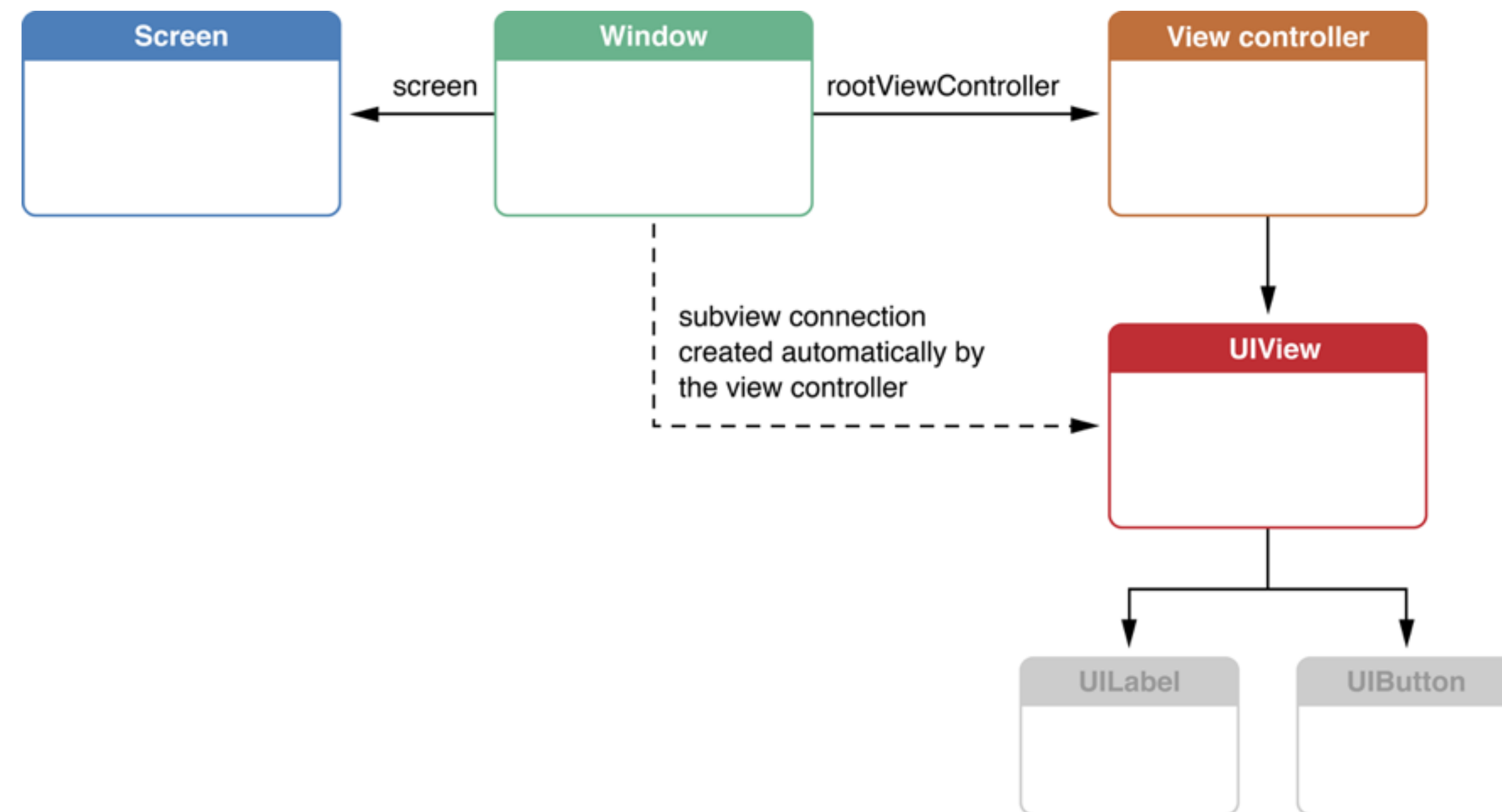
Views

- A view represents a user interface element; each view covers a specific area; within that area, it displays contents or responds to user events
- Views can be nested in a view hierarchy; subviews are positioned and drawn relative to their superview
- Views can animate their property values; animation are crucial to allow users understand changes in the user interface
- Views typically communicate with the controller through target/action, delegation, and data source patterns
- Complex apps are composed of many views, which can be grouped in hierarchies and animated
- Views that respond to user interaction are called **controls** (**UIControl**): **UIButton**s and **UISlider**s are controls

https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/UIKitUICatalog/index.html#//apple_ref/doc/uid/TP40012857

View Controllers

- A view controller organizes and controls a view
- A view controller is a controller in the application's MVC
- View controllers are subclasses of the **UIViewController** class
- View controllers also have specific tasks iOS expects them to perform, which are defined in the **UIViewController** class
- Normally, a view controller is attached to a window and automatically adds its view as a subview of the window



View Controllers

- View controller must carefully load views in order to optimize resource usage
- A view controller should only load a view when the view is needed and it can also release the view under certain conditions (low memory)
- View controller coordinate actions occurring in its connected views
- Because of their generality (which is required for reusability), view objects are agnostic on their meaning in the application and typically send messages to their controller; view controllers, instead, are required to understand and react to certain events that occur in the views they manage

Views and View Controllers

- Every view is controlled by only one view controller
- A view controller has a **view** property; when a view is assigned to the **view** property, the view controller owns the view
- Subviews might be controlled by different view controllers: several view controllers might be involved in managing portions of a complex view
- Each view controller interacts with a subset of the app's data: they are responsible for displaying specific content and should know nothing about data other than what they show (e.g. Mail app)



Mobile Application Development

Lecture 4
iOS SDK



This work is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-nc-sa/4.0/).