iOS Development

Lecture 2
iOS SDK and UIKit

Ing. Simone Cirani
email: simone.cirani@unipr.it
http://www.tlc.unipr.it/cirani

Lecture Summary

- iOS operating system
- iOS SDK
- Tools of the trade
- Model-View-Controller
- MVC interaction patterns
- View Controllers
- View Controller lifecycle
- UIColor, UIFont, NSAttributedString
- UIKit views and controls: UILabel, UIButton, UISlider, UISwitch, UITextField, UITextView
- NSNotificationCenter, keyboard notifications
- Auto Layout
- DEMO

iOS

- iOS is Apple's mobile operating system, shipping with iPhone, iPod Touch, and iPad devices
- First released in 2007
- Current version is iOS7
  - released on September 2013
  - runs on iPhone 4/4s/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

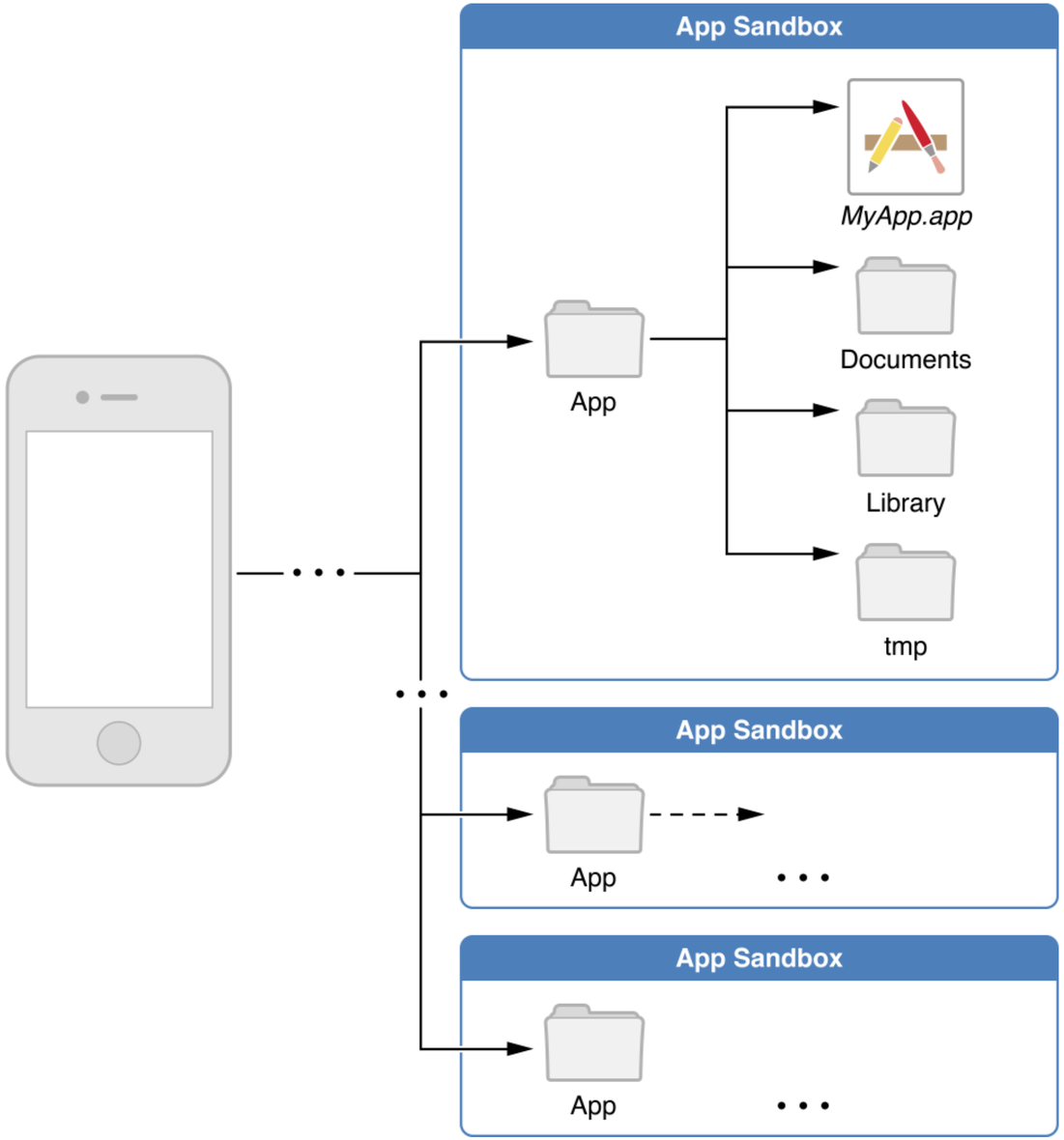
Multi-threading in iOS

- 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

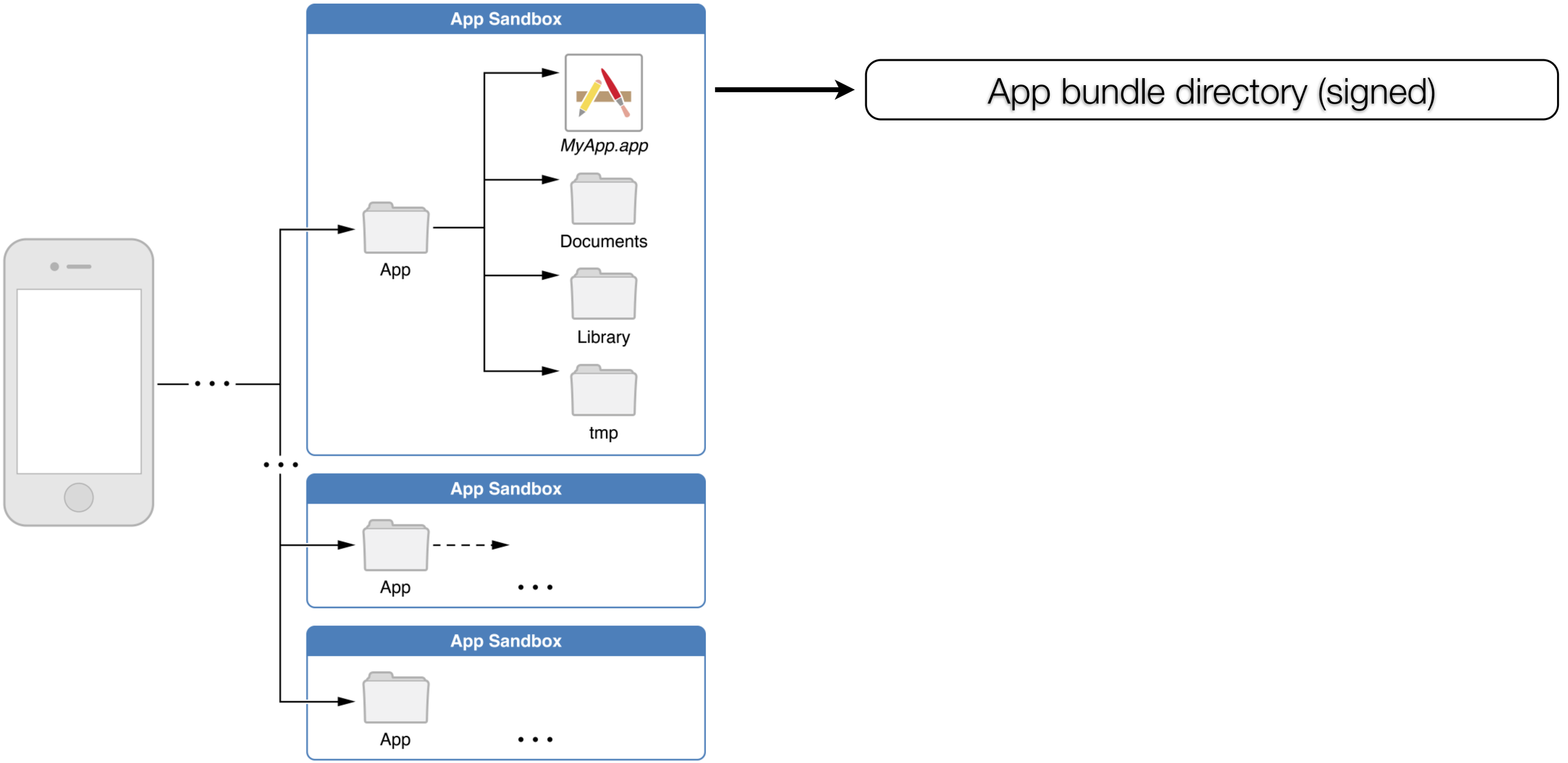
- In some cases, apps may ask the OS for background execution, but this requires a proper management of the app states

App sandbox

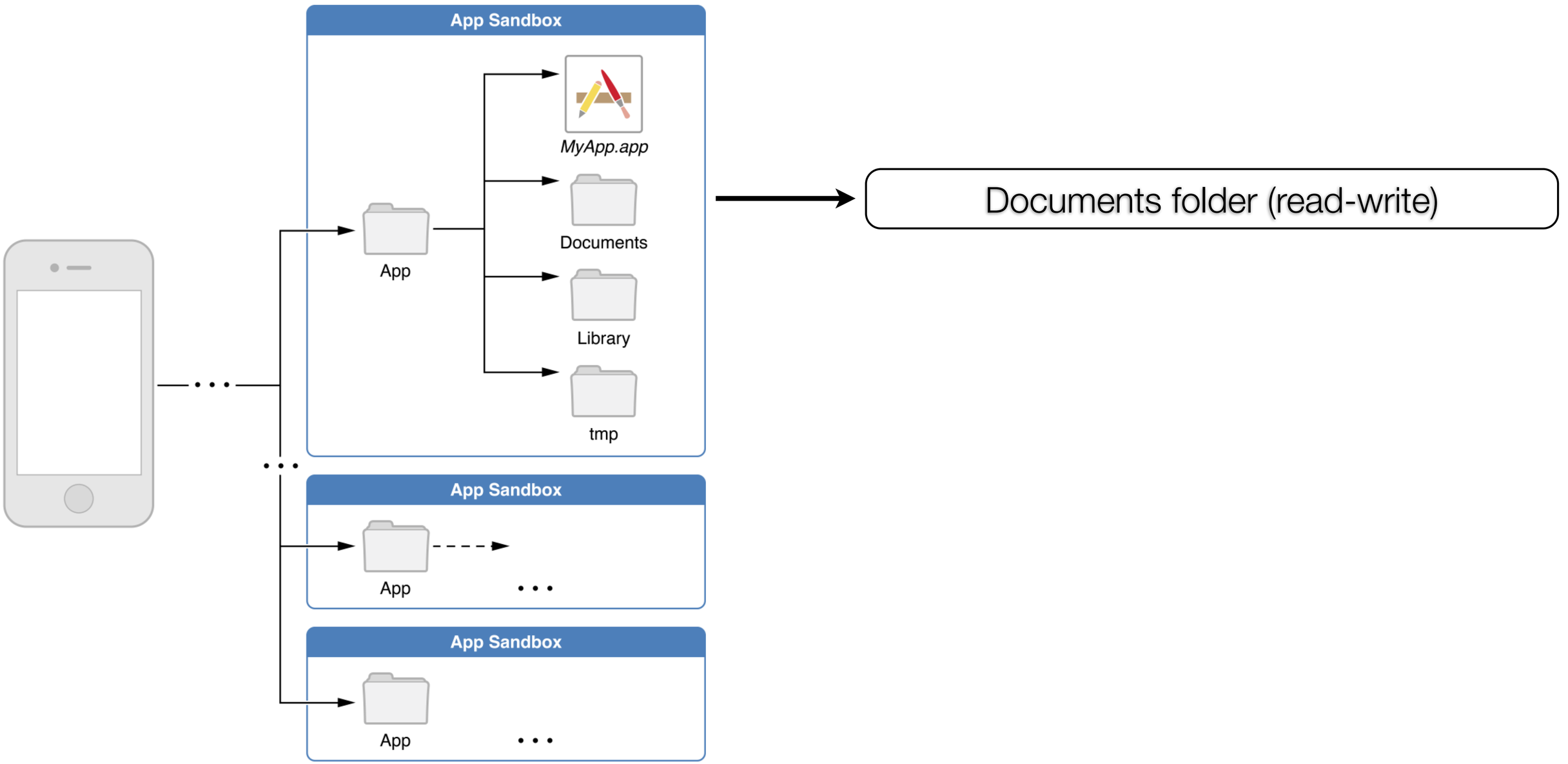
- For security reasons, iOS places each app (including its preferences and data) in a sandbox at install time
- A sandbox provides controls that limit the app's access to files, preferences, network resources, hardware, ...
- The system installs each app in its own sandbox directory, which can be seen 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**



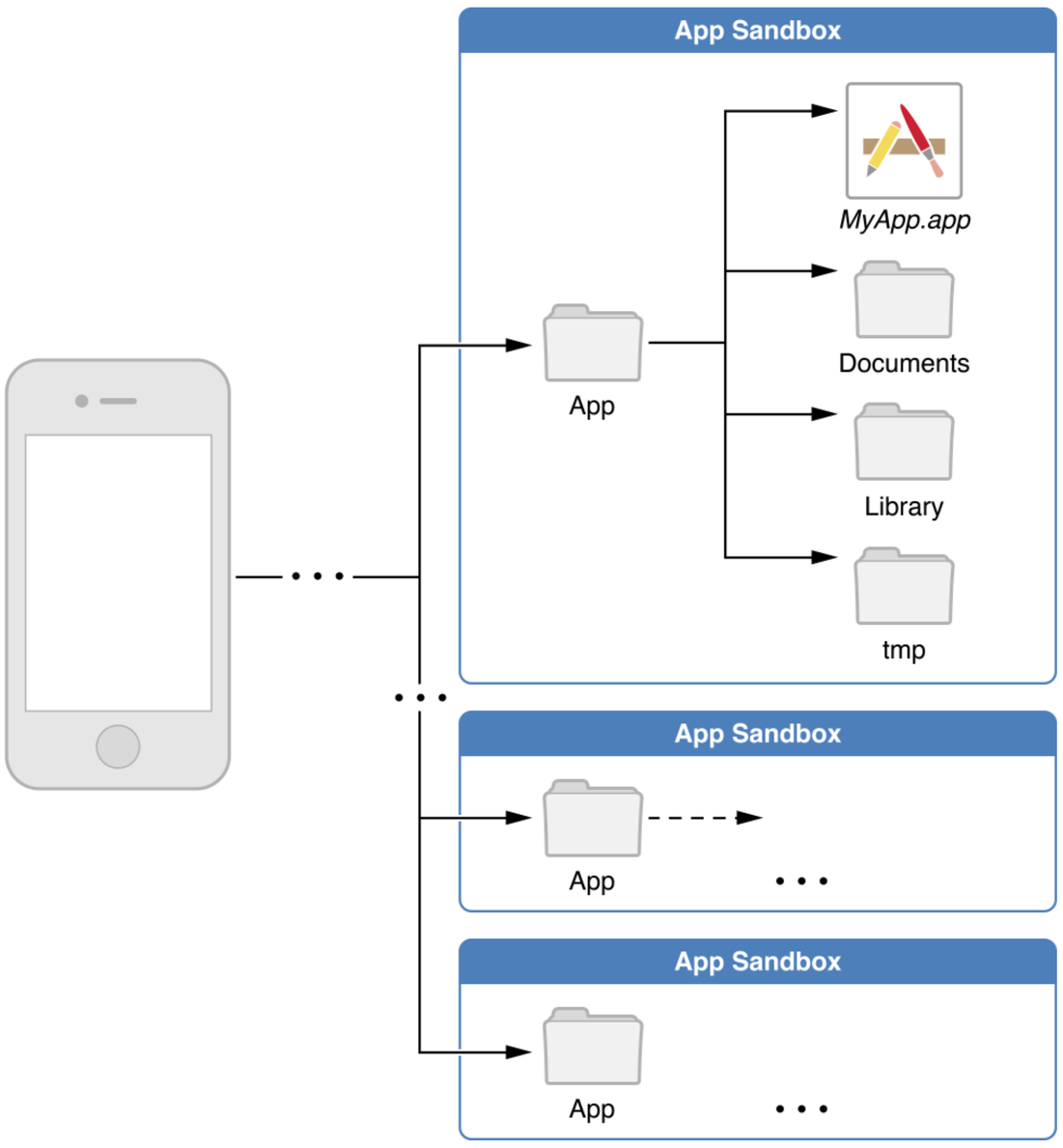
App sandbox


App Sandbox
MyApp.app
Documents
App
Library
tmp
App Sandbox
App
App Sandbox
App
App bundle directory (signed)

App sandbox

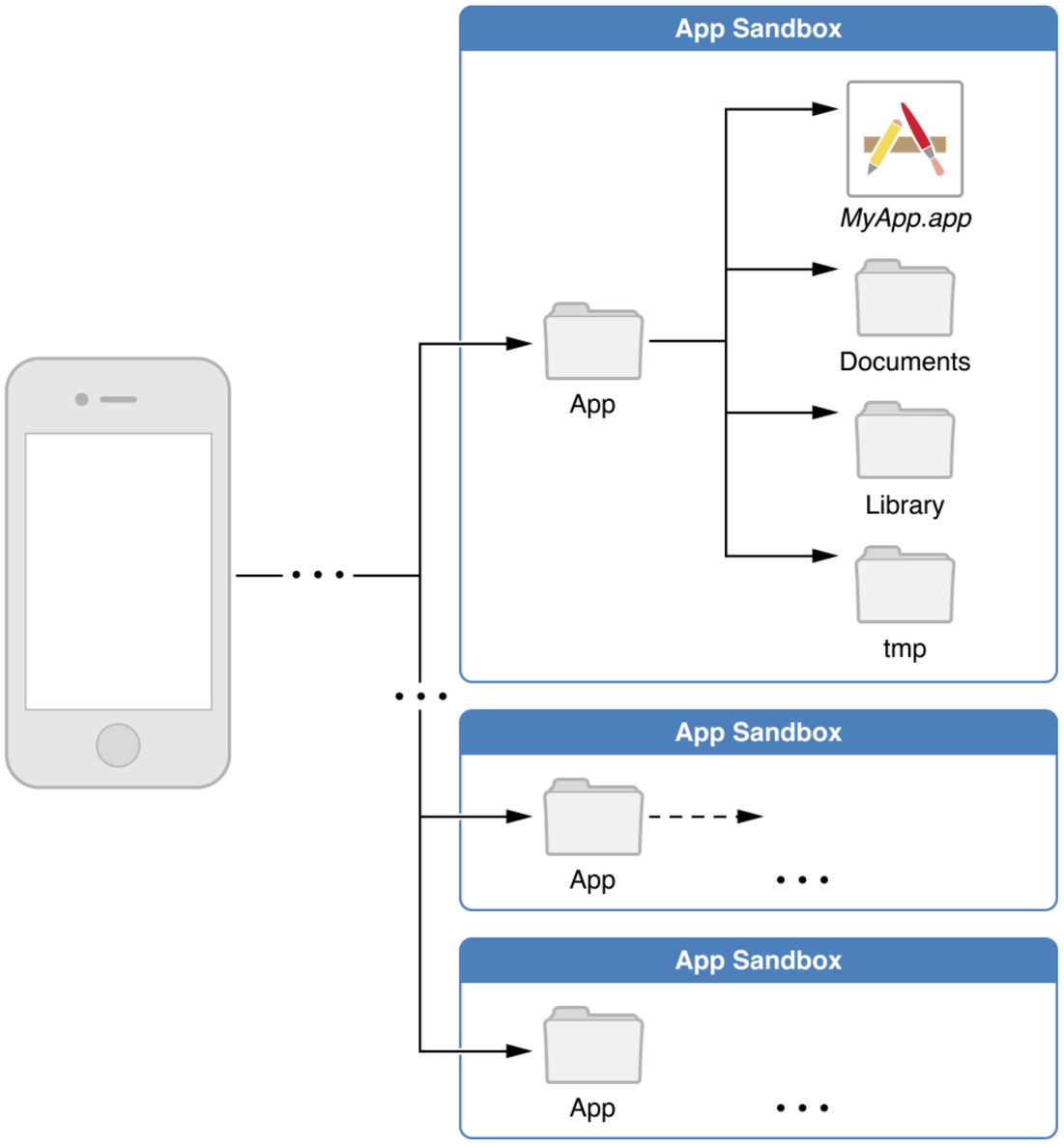

App Sandbox
MyApp.app
Documents
App
Library
tmp
App Sandbox
App
App Sandbox
App
Documents folder (read-write)

App sandbox



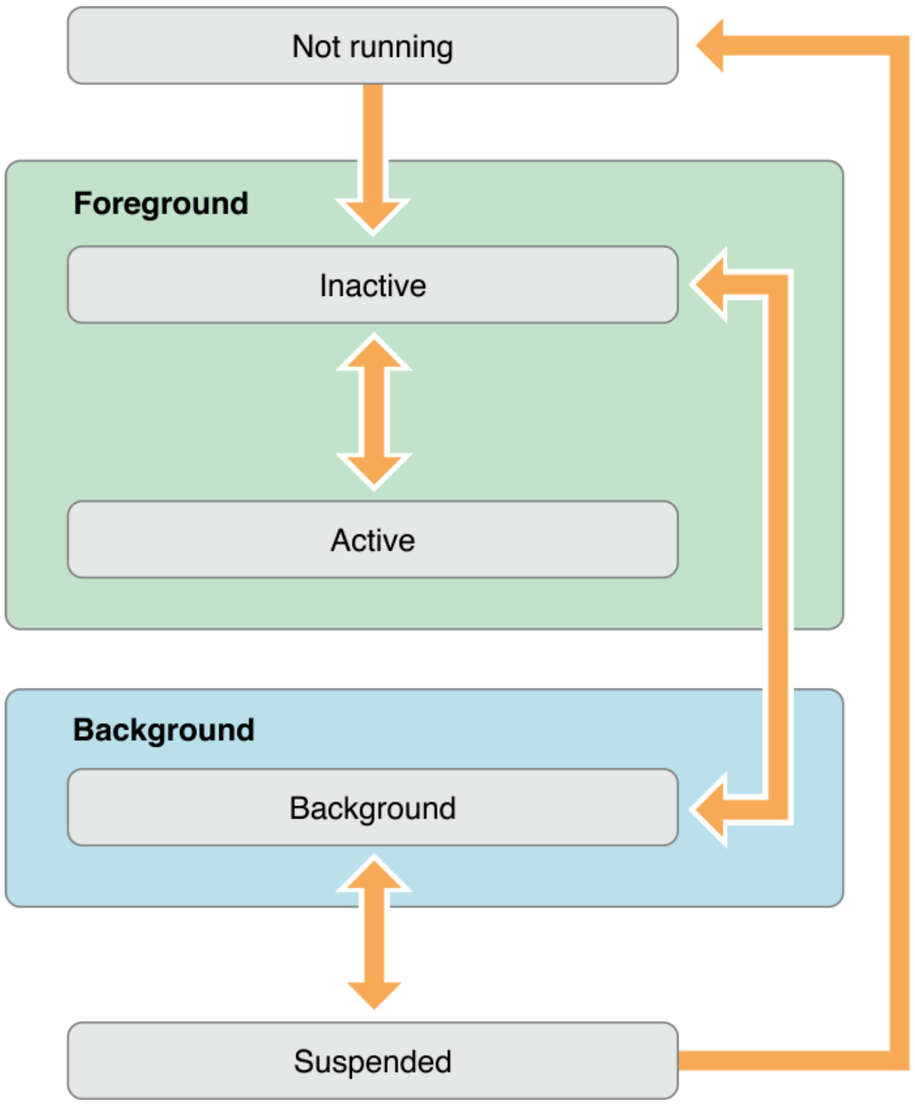
User-specific files, backed up by iTunes

App sandbox



Temporary files folder (read-write); contents are erased when the app exits

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)
- Each application has a *delegate* (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

App Life Cycle

1. App launched
2. App initialized
3. Load root view controller
4. Wait for event
5. Handle event
6. App terminates

App Life Cycle

1. App launched
2. App initialized
3. Load root view controller
4. Wait for event
5. Handle event
6. App terminates

Event loop - objects are destroyed (`dealloc` gets called) when the event loop completes its execution

UIApplicationDelegate 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

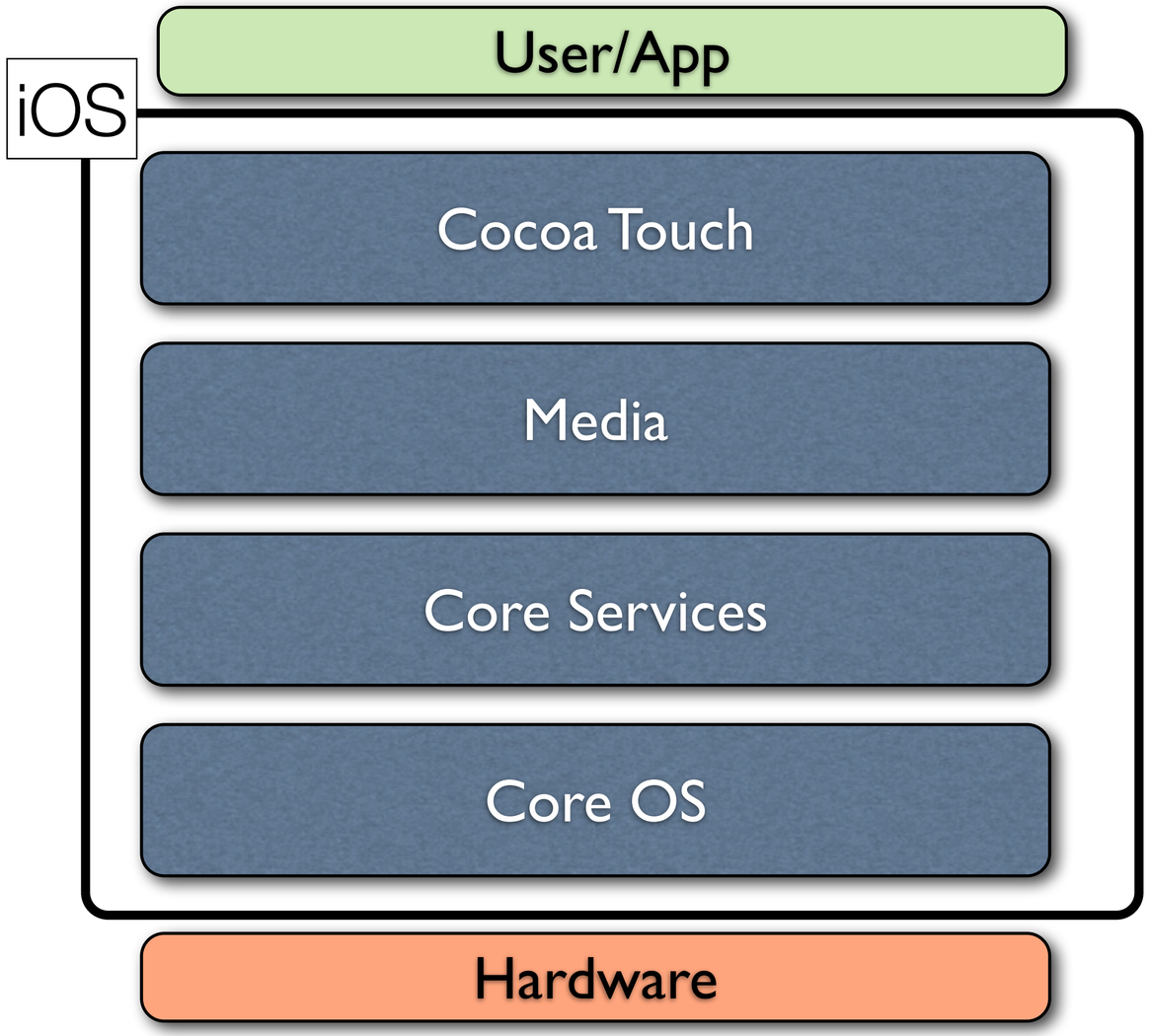
`applicationWillResignActive:` Lets you know that your app is transitioning away from being the foreground app; use this method to put your app into a dormant state

`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

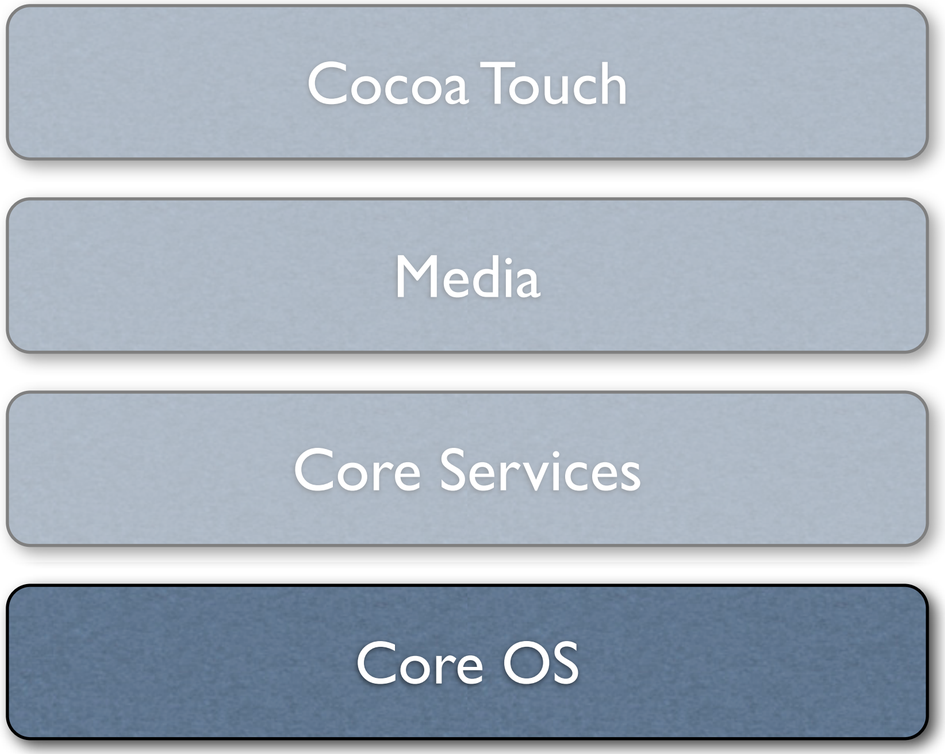
`applicationWillTerminate:` Lets you know that your app is being terminated; this method is not called if app is suspended

iOS Layers



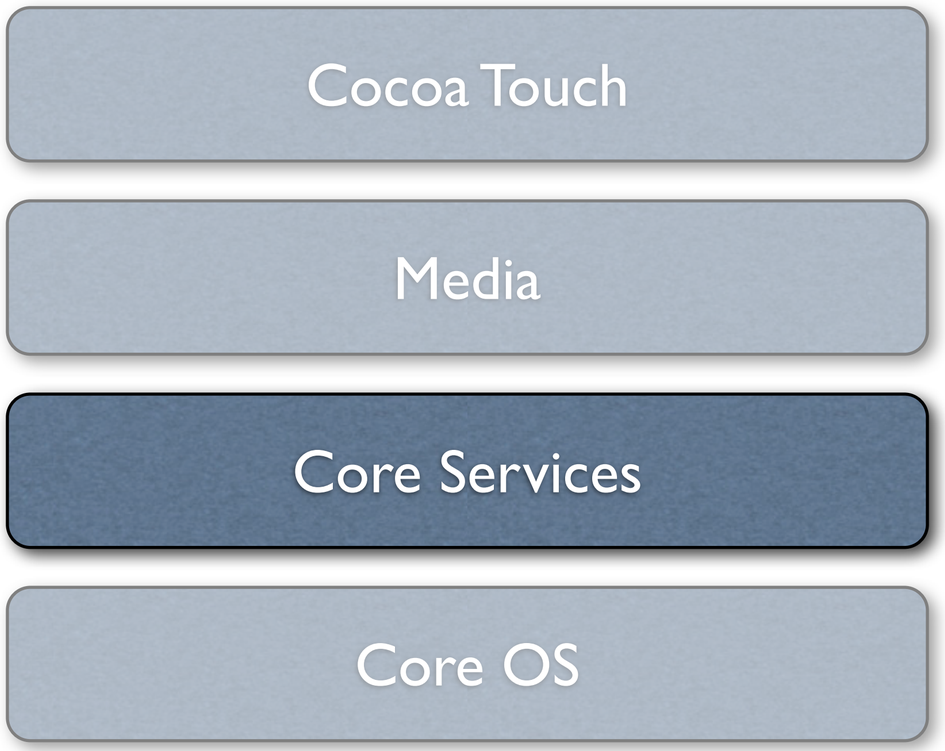
- The iOS architecture is layered
- iOS acts as an intermediary between the underlying hardware and the apps
- Apps communicate with the hardware through a set of well-defined system interfaces
- Lower layers contain fundamental services and technologies
- Higher-level layers build upon the lower layers 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



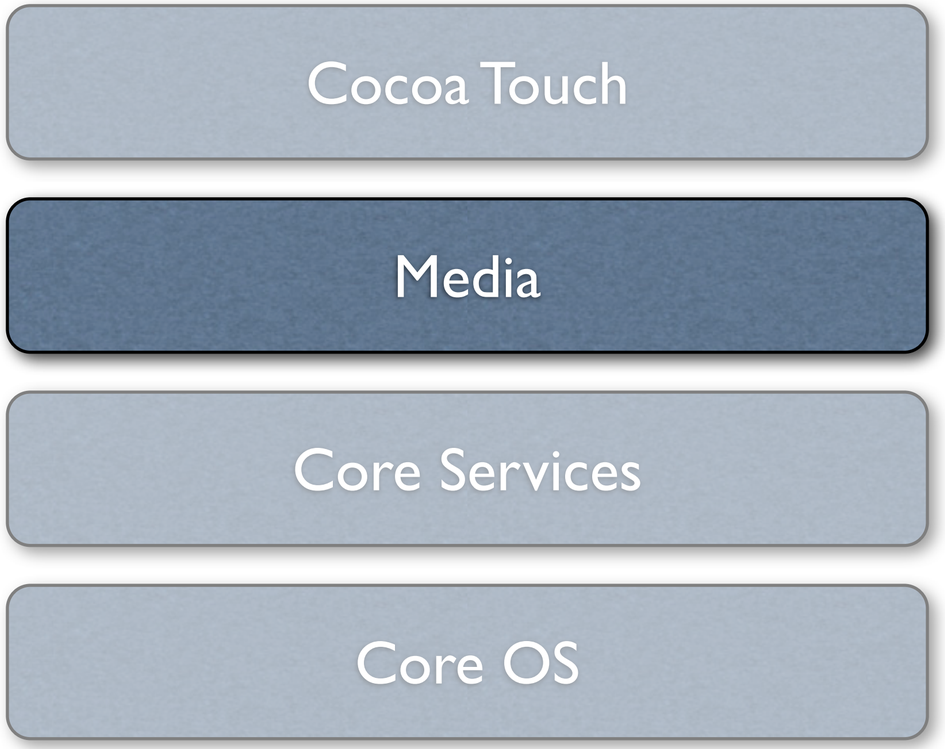
- Low-level features
- Main frameworks:
  - **Accelerate Framework**: vector and matrix math, digital signal processing, large number handling, and image processing
  - **Core Bluetooth Framework**
  - **Security Framework**: support for symmetric encryption, hash-based message authentication codes (HMACs), 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



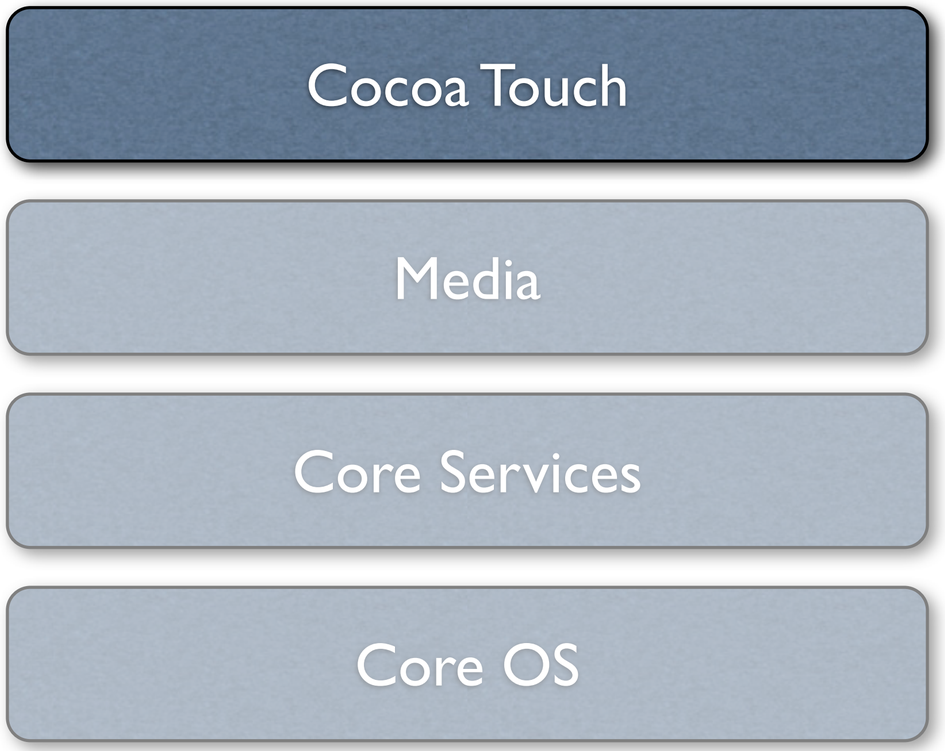
- Fundamental system services for apps
- Main frameworks:
  - **CFNetwork Framework**: BSD sockets, TLS/SSL connections, DNS resolution, HTTP/HTTPS connections
  - **Core Data Framework**
  - **Core Foundation Framework**: (C library) collections, strings, date and time, threads
  - **Core Location Framework**: 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

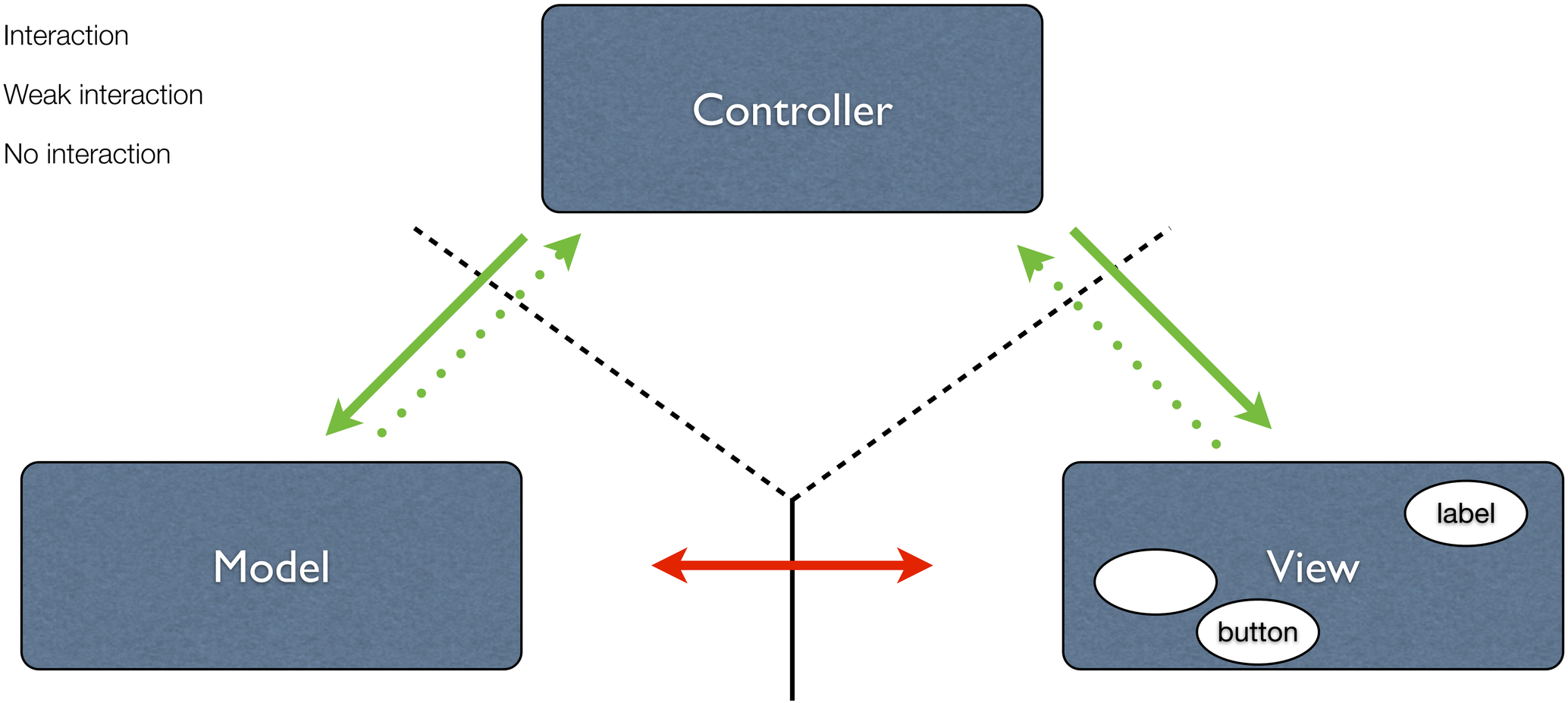
- MVC stands for Model-View-Controller
- An application's code can belong either to the model, to the view, or to the controller
- The **Model** is the representation of the data that will be used in the application; the model is independent from the View, since it does not know how data will be displayed (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

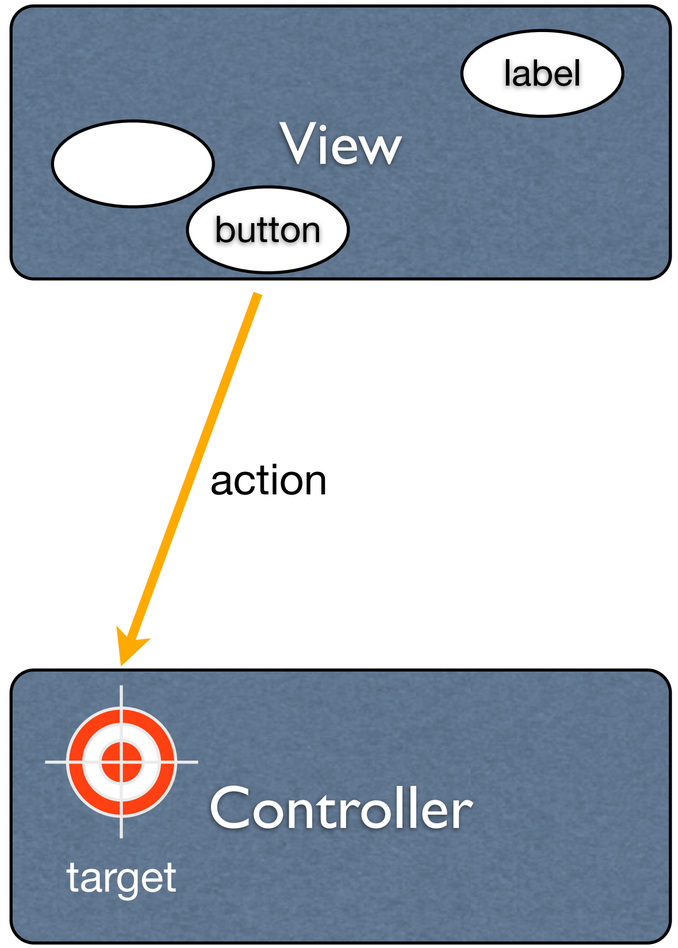


Model-View-Controller interactions

- The Model has direct interaction with neither the View (obviously) nor the Controller, since it 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

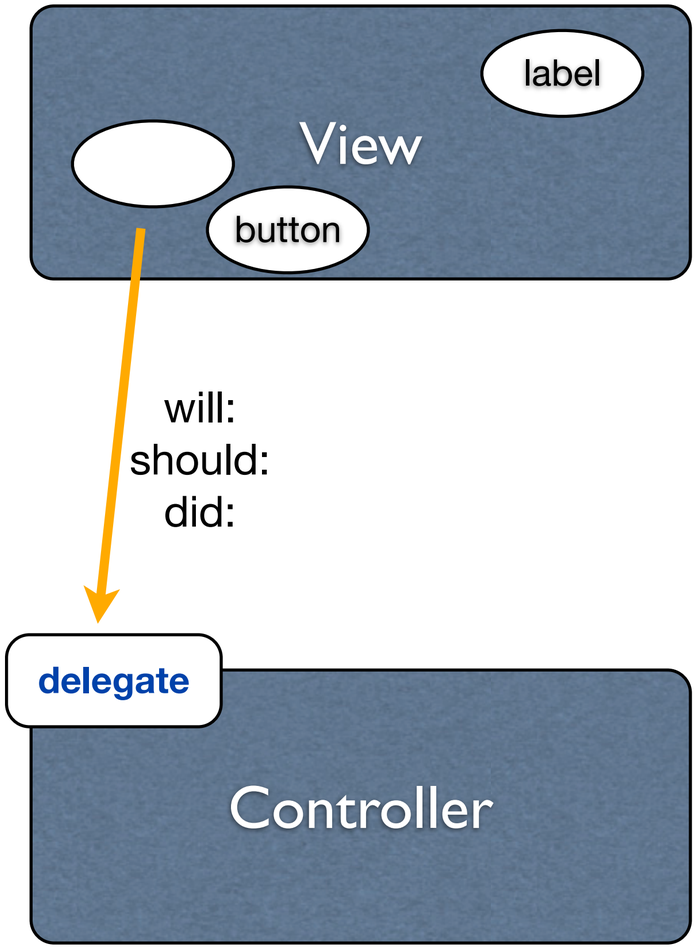
View-to-Controller interactions

- 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



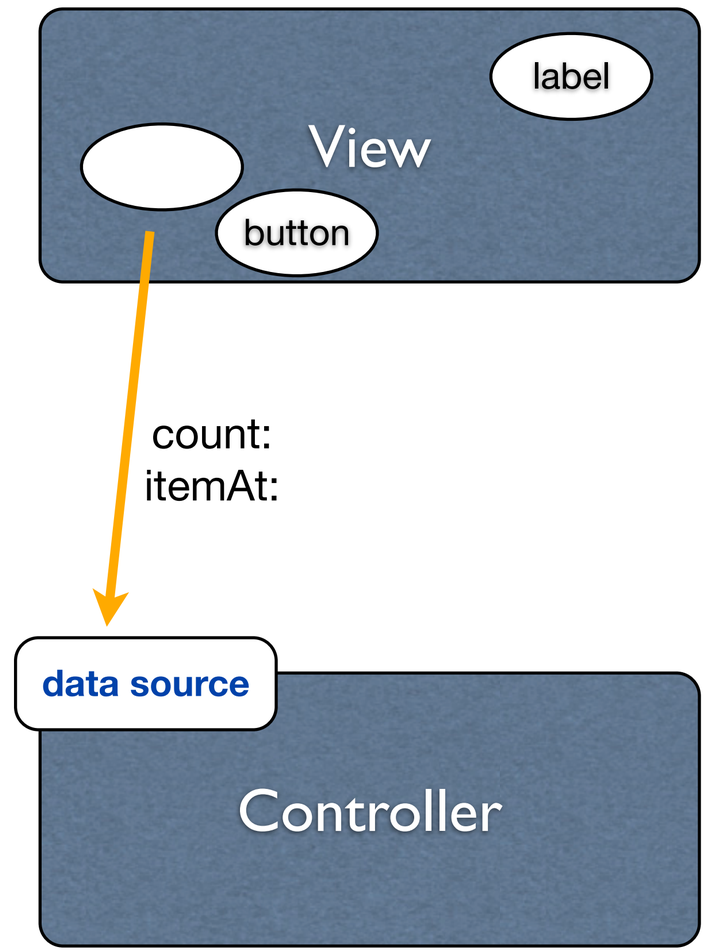
View-to-Controller interactions

- 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



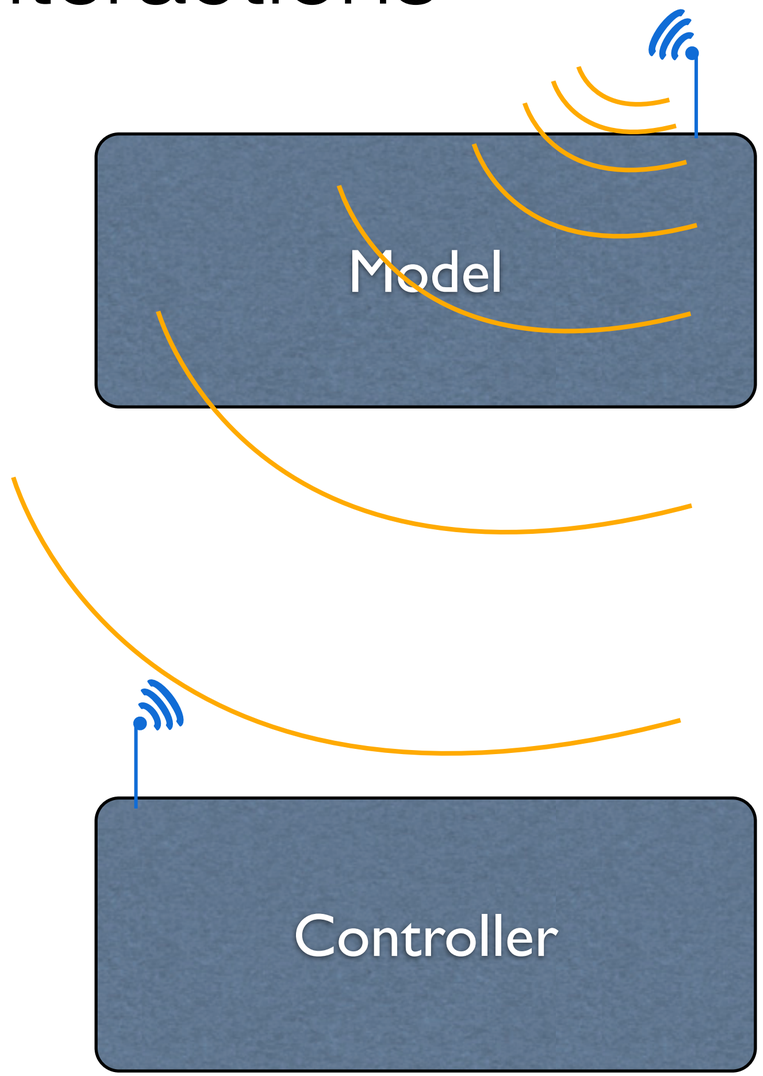
View-to-Controller interactions

- 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

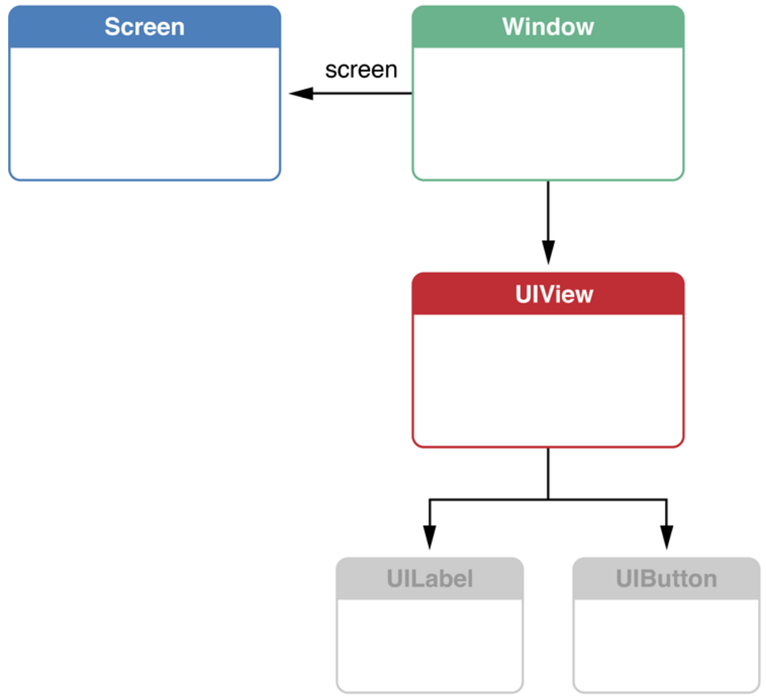
- The Controller's job is to retrieve and format data from the Model so that it can be displayed in the View
- Most of the work when developing apps is done within the Controller(s)
- Complex applications require several MVC to come into play, for instance when an event on a view causes another view 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)

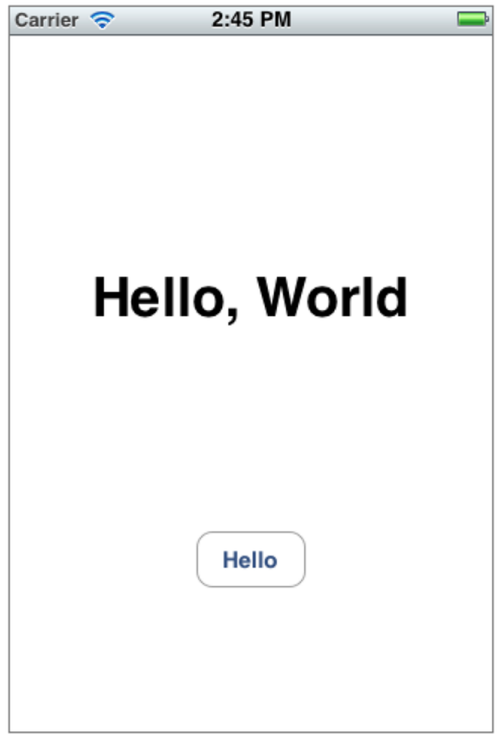
View Controllers

- 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





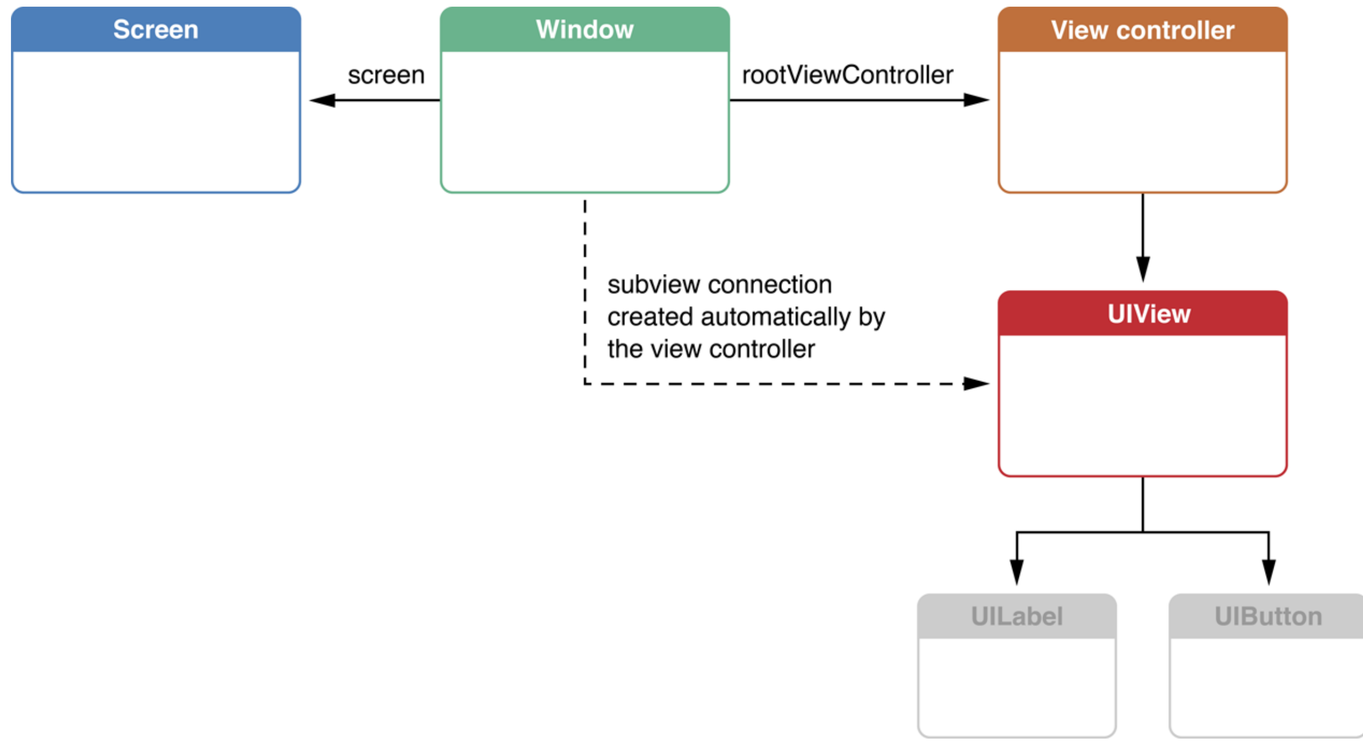
Views

- 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)

View Controller Lifecycle

- View controllers receive messages whenever certain events related to the view controller's lifecycle occur

- Every view controller is a subclass of the `UIViewController` class which defines a set of methods that will be executed as the view controller receives lifecycle messages

- Subclasses of `UIViewController` might override such methods to provide specific behavior (the implementation of the superclass must also be called through `super`)

View Controller Lifecycle

1. The first step in the lifecycle is the creation; view controllers can be created
   - through *storyboards*
   - programmatically
2. Next, the outlets of the view controller are set
3. View controllers (their managed views) may appear and disappear from the screen
4. Low-memory notification

- Anytime one of these events occurs, the system sends a message to the view controller

viewDidLoad

- After the view controller has been created and the outlets set, `viewDidLoad` gets invoked
- `viewDidLoad` is where most of the initialization of the view controller can be performed, after the outlets have been set (safe)
- `viewDidLoad` is called only once in the life of a view controller, after it has been created
- At this point all outlets have been set, but bounds property of the view is not set, so it is not safe to perform geometry-based settings in `viewDidLoad`

```
- (void)viewDidLoad{
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.
    // ...
}
```

viewWillAppear

- When the view is about to appear on the screen, `viewWillAppear` gets invoked
- The argument tells whether the view is appearing through an animation or instantly
- This method gets invoked as many times as the view appears on the screen, so it could be invoked multiple times: DO NOT PUT CODE THAT SHOULD BE EXECUTED JUST ONCE, USE `viewDidLoad` TO DO THAT!
- Typically, `viewWillAppear` is used to perform:
  - operations that are related to changes that occurred while the view was not on screen
  - long-running operations that could be unnecessary if the view is never displayed or could block the rendering of the view if executed in `viewDidLoad` (possibly in a separate thread)

```
- (void)viewWillAppear:(BOOL)animated{
    [super viewWillAppear:animated];
    // ...
}
```

viewWillDisappear

- When the view is about to go off the screen, `viewWillDisappear` gets invoked
- The argument tells whether the view is disappearing through an animation or instantly
- This method gets invoked as many times as the view disappears from the screen, so it could be invoked multiple times
- Typically, `viewWillDisappear` is used to:
  - save view state for later retrieval (e.g. scroll position in a scrollable view)
  - cleanup resources (memory and processing) that are not necessary and could be brought back up the next time the view appears

```
- (void)viewWillDisappear:(BOOL)animated{
    [super viewWillDisappear:animated];
    // ...
}
```

viewDidAppear and viewDidDisappear

- When the view has appeared on the screen, `viewDidAppear` gets invoked
- The argument tells whether the view has appeared through an animation or instantly
- `viewDidAppear` invoked as many times as the view has come up on the screen, so it could be invoked multiple times
- Conversely, `viewDidDisappear` gets invoked when the view has disappeared from the screen

```
- (void)viewDidAppear:(BOOL)animated{
    [super viewDidAppear:animated];
    // ...
}
```

```
- (void)viewDidDisappear:(BOOL)animated{
    [super viewDidDisappear:animated];
    // ...
}
```

viewWillLayoutSubviews and viewDidLayoutSubviews

- These methods are invoked when the subviews of the view are about to be or have just been laid out
- Between the execution of `viewWillLayoutSubviews` and `viewDidLayoutSubviews`, **autolayout** is performed
- Geometry-related code can be performed here

```
- (void)viewWillLayoutSubviews{
    [super viewWillLayoutSubviews];
    // ...
}
```

```
- (void)viewDidLayoutSubviews{
    [super viewDidLayoutSubviews];
    // ...
}
```

Autorotation

- The view controller is responsible to handle device rotation (must be set in the project's settings file `Info.plist`)
- If the view controller's `shouldAutorotate` method returns YES, then the view contents should rotate

```
- (BOOL)shouldAutorotate{
    return YES;
}
```

- If so, the supported orientations are defined as return value of the `supportedInterfaceOrientations` method
- `supportedInterfaceOrientations` returns a `UIInterfaceOrientationMask`

```
- (NSUInteger)supportedInterfaceOrientations{
    return UIInterfaceOrientationMaskPortrait;
}
```

Autorotation

- Some methods are invoked to notify the view controller about rotation events

```
- (void)willRotateToInterfaceOrientation:(UIInterfaceOrientation)toInterfaceOrientation
                                duration:(NSTimeInterval)duration;


- (void)willAnimateRotationToInterfaceOrientation:(UIInterfaceOrientation)toInterfaceOrientation
                                         duration:(NSTimeInterval)duration;


- (void)didRotateFromInterfaceOrientation:(UIInterfaceOrientation)fromInterfaceOrientation;
```

didReceiveMemoryWarning

- When memory is low, the view controller is notified and the `didReceiveMemoryWarning` method gets invoked
- Should this happen, all unnecessary (and big) resources (in the heap) should be released; to do this, `strong` pointers should be set to `nil`
- Resources allocated in the memory released at this point should be re-creatable
- Good code should avoid releasing big resources at this point, but should do this well in advance

```
- (void)didReceiveMemoryWarning{
    [super didReceiveMemoryWarning];
    // ...
}
```

awakeFromNib and initialization of view controllers

- `init` is not invoked on objects instantiated by the storyboard
- `awakeFromNib` is invoked by any object instantiated by the storyboard (views, subviews, view controllers, ...) before outlets are set (before `viewDidLoad`)
- `awakeFromNib` should have initialization code that could not be executed anywhere else
- UIViewController's designated initializer and awakeFromNib should have the same initialization code:

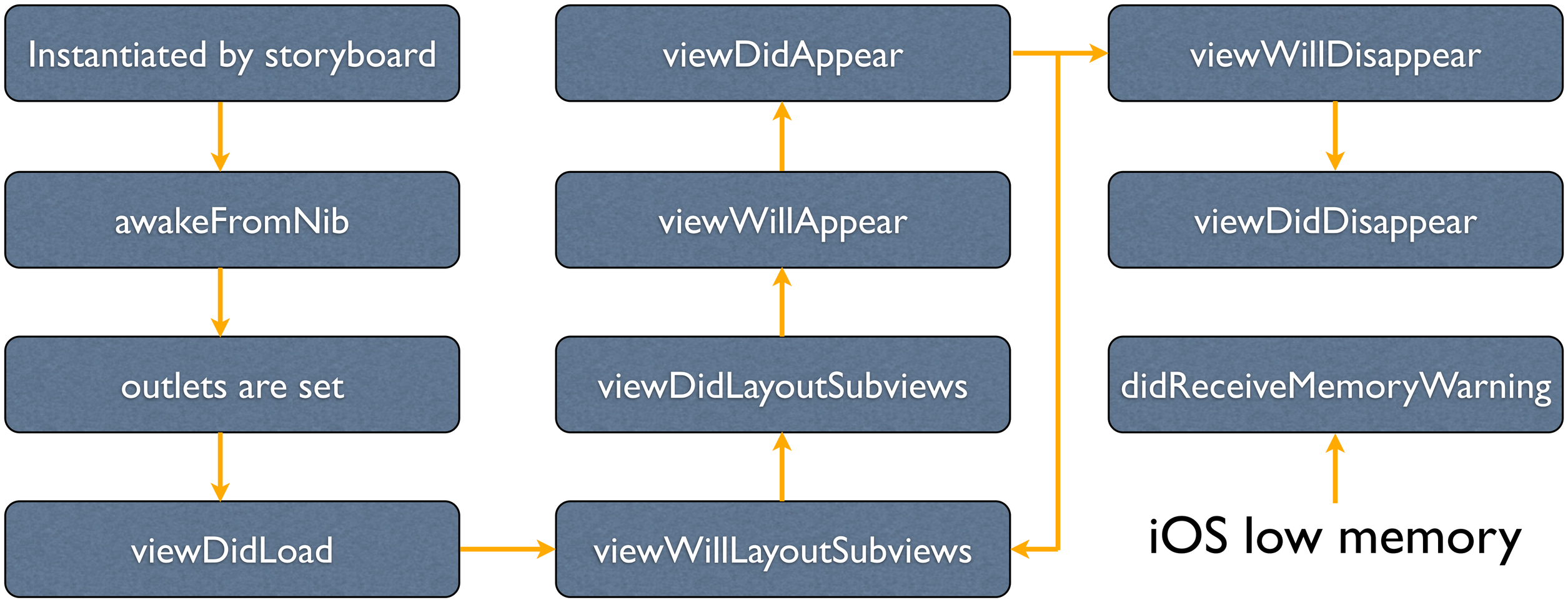
```
- (void)setup{...}

- (void)awakeFromNib{
    [self setup];
}

- (instancetype)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)nibBundleOrNil{
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
    [self setup];
    return self;
}
```

- `viewDidLoad` is preferrable

View Controller Lifecycle



UIColor

- `UIColor` class represents a color which can be initialized from:
  - RGB (red/gree/blue)
  - HSB (hue/saturation/brightness)
  - from images (patterns)
- Colors handle transparency through an `alpha` property, ranging from 0 to 1
- System colors are provided (`blackColor`, `redColor`, `greenColor`, ...)

Working with fonts

- Fonts can make applications great to see and might hugely improve user experience
- Choosing the right fonts and font sizes is extremely important to make applications good-looking and enjoyable for users
- `UIFont` class represents fonts; the best way to get a font is to ask the OS for the preferred font for a certain style of text (e.g. `UIFontTextStyleHeadline`, `UIFontTextStyleBody`, ...)

```
UIFont *font = [UIFont preferredFontForTextStyle:UIFontTextStyleHeadline];
```

- System fonts are used for buttons, not for content

NSAttributedString

- `NSAttributedString` object manages character strings and associated sets of attributes (for example, font and kerning) that apply to individual characters or ranges of characters in the string
- Attributes (such as color, stroke width, ...) apply to portions (ranges) of the string
- Key/value pairs (dictionaries of attributes) are assigned to a certain range of a string
- Attributed strings allow developers to render styled text
- The UIKit framework adds methods to `NSAttributedString` to support the drawing of styled strings and to compute the size and metrics of a string prior to drawing
- `NSAttributedString` is used to render fonts on the screen
- `NSAttributedString` is not a subclass of `NSString`; it is possible however to get a `NSString` from a `NSAttributedString` with the `string` method

NSAttributedString

- **NSAttributedString** are typically created from a simple (unattributed) **NSString**, an existing **NSAttributedString**, or a **NSString** and a **NSDictionary** of attributes

```
- (id)initWithString:(NSString *)aString

- (id)initWithAttributedString:(NSAttributedString *)attributedString

- (id)initWithString:(NSString *)aString attributes:(NSDictionary *)attributes
```

NSAttributedString

- **NSAttributedString** are typically created from a simple (unattributed) **NSString**, an existing **NSAttributedString**, or a **NSString** and a **NSDictionary** of attributes

```
- (id)initWithString:(NSString *)aString

- (id)initWithAttributedString:(NSAttributedString *)attributedString

- (id)initWithString:(NSString *)aString attributes:(NSDictionary *)attributes
```

- A mutable version of **NSAttributedString**, named **NSMutableAttributedString**, allows you to change dynamically the characters and attributes of the string at runtime through the methods:

```
- (void)addAttribute:(NSString *)name value:(id)value range:(NSRange)aRange

- (void)removeAttribute:(NSString *)name range:(NSRange)aRange

- (void)setAttributes:(NSDictionary *)attributes range:(NSRange)aRange
```

NSAttributedString

- Attributes that can be set are:

Attribute	Value type
NSFontAttributeName	UIFont*
NSForegroundColorAttributeName	UIColor*
NSBackgroundColorAttributeName	UIColor*
NSStrokeWidthAttributeName	NSNumber*
NSStrokeColorAttributeName	UIColor*

UIKit Views

- The `UIView` class defines a rectangular area on the screen and the interfaces for managing the content in that area
- Views can embed other views and create sophisticated visual hierarchies, creating a parent-child relationship between the view and its subviews
- The geometry of a view is defined by some properties:
  - `frame`: origin and dimensions of the view in the coordinate system of its superview
  - `bounds`: the internal dimensions of the view as it sees them
  - `center`: the coordinates of the center point of the rectangular area covered by the view
- All UI elements inherit from `UIView`

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

UIKit Views

- The following are the most commonly used `UIView` properties:

Property	Value type	Description
`frame`	`CGRect`	Frame rectangle describing the view's location and size in the superview's coordinate system
`bounds`	`CGRect`	Bounds rectangle describing the view's location and size in its own coordinate system
`center`	`CGPoint`	Center of the frame
`backgroundColor`	`UIColor*`	Background color; defaults to `nil` (transparent)
`alpha`	`CGFloat`	0.0 means transparent and 1.0 means opaque
`hidden`	`BOOL`	YES means the view is invisible, NO means visible
`userInteractionEnabled`	`BOOL`	NO means user events are ignored

UILabel

- `UILabel` objects are used to display static text on a fixed (by you) number of lines
- `UILabel` has the following properties to work with the text to display:

Property	Value type	Description
`text`	`NSString*`	Text being displayed
`font`	`UIFont*`	Font of the text
`textColor`	`UIColor*`	Color of the text
`textAlignment`	`NSTextAlignment`	Alignment of the text (NSTextAlignmentLeft, NSTextAlignmentRight, NSTextAlignmentCenter...)
`attributedText`	`NSAttributedString*`	Styled text being displayed
`numberOfLines`	`NSInteger`	Maximum number of lines to use

UIKit Controls

- A control is a communication tool between a user and an app
- Controls convey a particular action or intention to the app through user interaction
- Controls can be used to manipulate content, provide user input, navigate within an app, or execute other pre-defined actions
- The `UIControl` class (subclass of `UIView`) is the base class for all controls
- `UIControl` is never used, but its subclasses, such as `UIButton` and `UISlider`, are used instead
- Typical controls that can be used in iOS:
  - **Buttons**
  - **Date Pickers**
  - **Page Controls**
  - **Segmented Controls**
  - **Text Fields**
  - **Sliders**
  - **Steppers**
  - **Switches**

Control states

- A **control state** describes the current interactive state of a control:
  - normal (enabled but not selected or highlighted)
  - selected (e.g. in UISegmentedControl) ⟶ 

  - disabled
  - highlighted (when a touch enters and exits during tracking and when there is a touch up event)
- The control state changes as the user interacts with the control
- Specific behavior and appearance can be specified for each control state

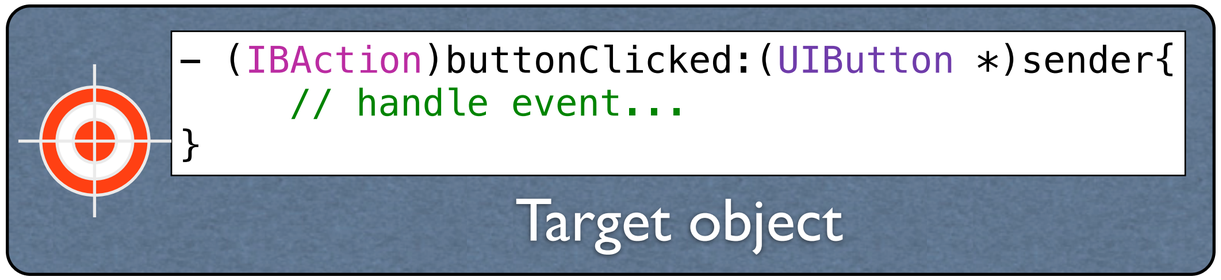
Control events

- **Control events** represent the ways (physical gestures) that users can make on controls
- Typical control events:
  - `UIControlEventTouchDown`: touch down inside a control
  - `UIControlEventTouchDownRepeat`: repeated touch down
  - `UIControlEventTouchDragInside`: a finger is dragged inside the bounds of the control
  - `UIControlEventTouchDragOutside`: a finger is dragged outside the bounds of the control
  - `UIControlEventTouchDragEnter`: a finger is dragged into the bounds of the control
  - `UIControlEventTouchDragExit`: a finger is dragged from within a control to outside its bounds
  - `UIControlEventTouchUpInside`: a finger is lifted when inside the bounds of the control (typical for `UIButton`)
  - `UIControlEventTouchUpOutside`: a finger is lifted when outside the bounds of the control
  - `UIControlEventTouchCancel`: system event canceling the current touches for the control
  - `UIControlEventValueChanged`: touch dragging or otherwise manipulating a control, causing a series of different values

Target-action

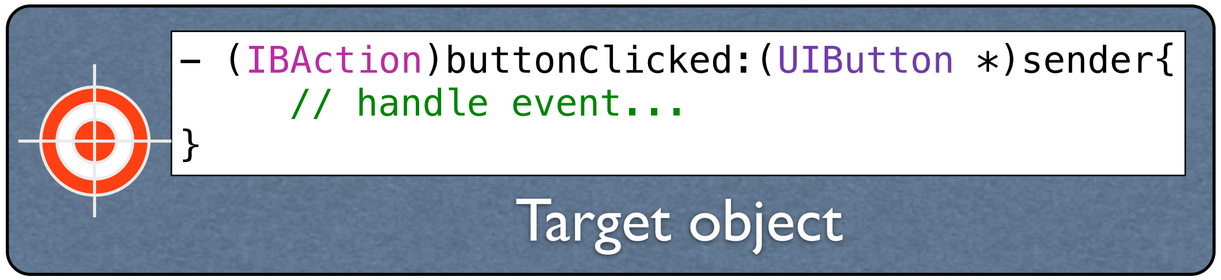
- The **target-action** mechanism is a model for configuring a control to send an action message to a target object after a specific control event

Button



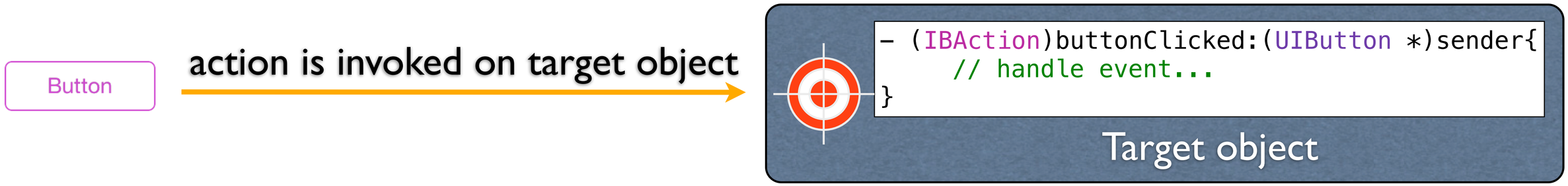
Target-action

- The **target-action** mechanism is a model for configuring a control to send an action message to a target object after a specific control event



Target-action

- The **target-action** mechanism is a model for configuring a control to send an action message to a target object after a specific control event



Target-action binding

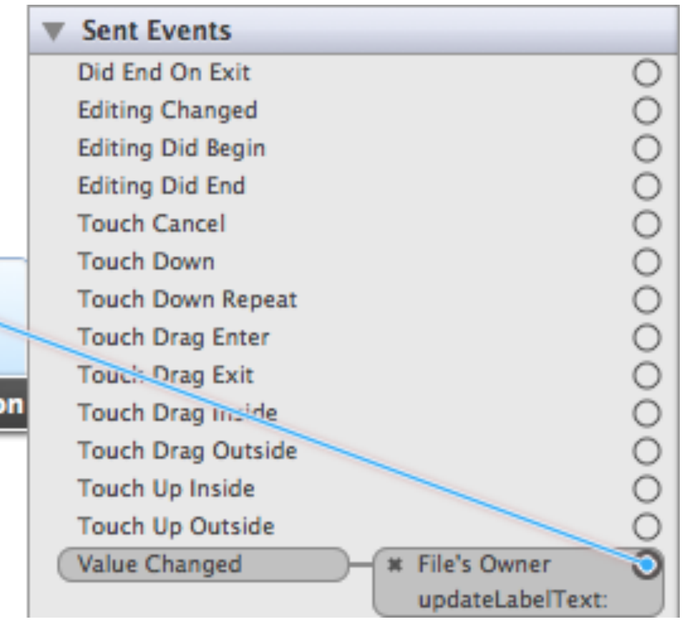
- Binding a target-action to a control event:

1. programmatically

```
[self.mySlider addTarget:self action:@selector(myAction:) forControlEvents:UIControlEventValueChanged];
```

2. with the Connection Inspector in Interface Builder to Control-drag the slider's Value Changed event to the action method in the target file

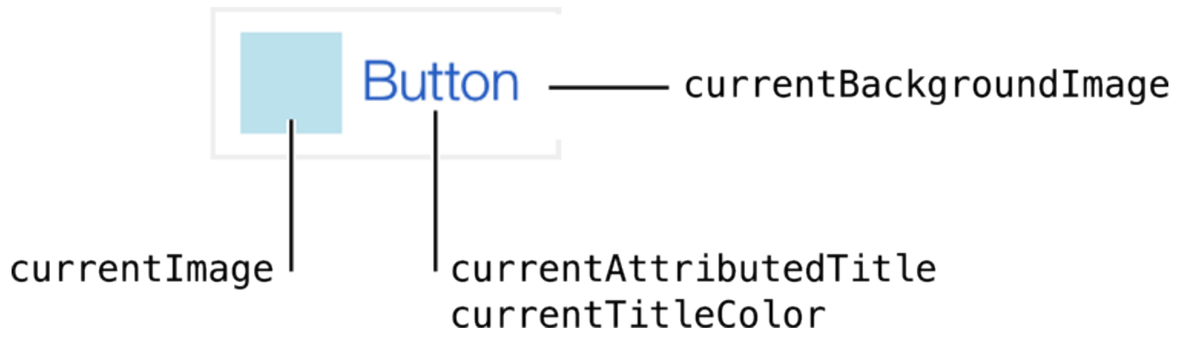
```
- (IBAction)updateLabelText:(UISlider *)
    sender {
}
```



3. Control-click the slider in Interface Builder, and drag its Value Changed event to the target object in your Storyboard and select the appropriate action from the list of actions available for the target

UIButton

- Buttons let a user initiate behavior with a tap
- Buttons display textual or image content
- When the users taps on a button, it changes its state to highlighted and its appearance accordingly
- If an action has been bound for a certain event, a button will trigger the execution of that action each time the event occurs
- The current appearance of button can be retrieved with some read-only properties



UIButton

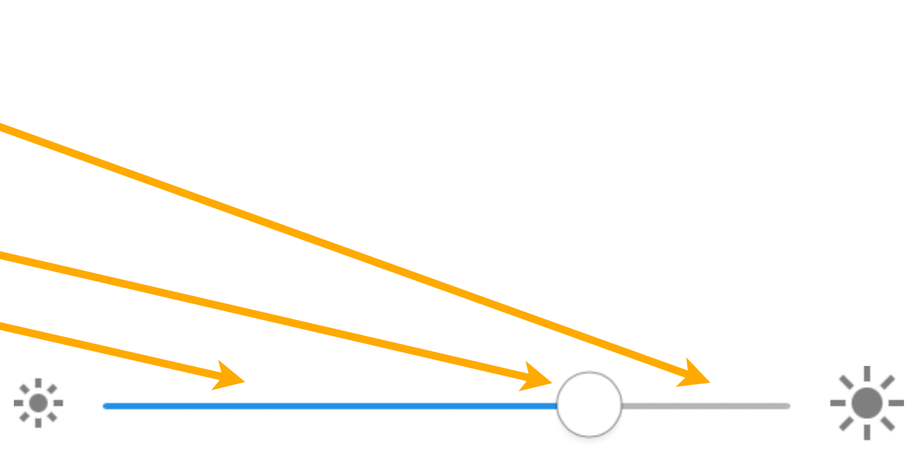
- It is possible to set the title, image, and background image of a button for each state

1. in the attributes inspector
2. programmatically

```
- (void)setTitle:(NSString *)title forState:(UIControlState)state
- (void)setAttributedTitle:(NSAttributedString *)title forState:(UIControlState)state
- (void)setImage:(UIImage *)image forState:(UIControlState)state
- (void)setBackgroundImage:(UIImage *)image forState:(UIControlState)state
```

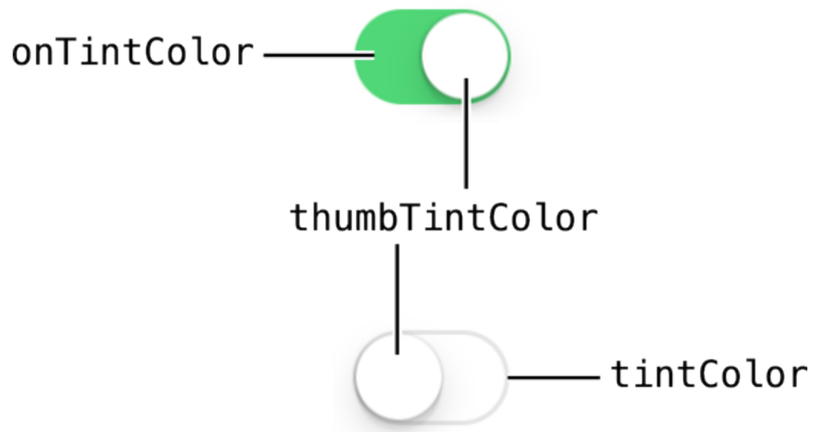
UISlider

- Sliders enable users to interactively modify some adjustable value in an app
- It is possible to configure a minimum, maximum, and current value for your slider with the properties `minimumValue`, `maximumValue`, and `value`, respectively
- Default values are minimum = 0, maximum = 1, and current value = 0.5
- It is possible to change the tint of the slider with the properties
  - `maximumTrackTintColor`
  - `thumbTintColor`
  - `minimumTrackTintColor`



UISwitch

- A switch lets the user turn an option on and off
- The appearance of a switch can be customized by setting the appropriate properties



- The boolean property **on** is used to set the off/on state of the switch

UITextField

- Text fields allows the user to input a single line of text into an app
- You can set and retrieve the value of the input text with the `text` and `attributedText` properties
- The `placeholder` and `attributedPlaceholder` properties are used to set a placeholder text in the field
- Text can also be styled using `font`, `textAlignment`, `textColor` properties
- The clear button on the right is displayed by default; it is possible to configure whether it should be present or not by using the property `clearButtonMode` with a `UITextFieldViewMode`
- The keyboard style and layout can also be set (`UITextInputTraits`)

Keyboard management

- Tapping a `UITextField` causes the keyboard to appear
- The keyboard slides in from the bottom of the screen and covers a certain area of the view
- The keyboard becomes the first responder
- It is necessary to handle the appearance of the keyboard properly:
  - it might be necessary to slide the view content up so that it does not get covered by the keyboard (more on this later)
  - it is necessary to make the keyboard disappear when done using it (tapping “done” won;t make it go away)
- The trick is to set the background view’s class to be UIControl instead of UIView so that it can receive tap events; when the background view is tapped, a target-action method is invoked and that’s where the keyboard can be dismissed by using:

```
[textField resignFirstResponder];
```

UITextView

- More powerful way to display text:
  - multiple lines, editable and selectable, scrollable
- Set the text through two properties:
  - `text`: `NSString` for normal text; properties that can affect the text style are `font` (`UIFont`), `textColor` (`UIColor`), and `textAlignment` (`NSTextAlignment`)
  - `attributedText`: `NSAttributedString` for styled text
- Efficient way to manipulate text:
  - `textStorage`: `NSTextStorage` allows you to change the styled text and it will automatically update the view (since iOS7)

UITextView

- `UITextView` has many following notable properties:

Property	Value type	Description
`text`	`NSString*`	Text being displayed
`font`	`UIFont*`	Font of the text
`textColor`	`UIColor*`	Color of the text
`textAlignment`	`NSTextAlignment`	Alignment of the text (NSTextAlignmentLeft, NSTextAlignmentRight, NSTextAlignmentCenter...)
`attributedText`	`NSAttributedString*`	Styled text being displayed
`textStorage`	`NSTextStorage`	Efficient text manipulation
`editable`	`BOOL`	Whether the receiver is editable
`selectable`	`BOOL`	Whether the receiver is selectable
`selectedRange`	`NSRange`	Current selection range in the text view

UITextViewDelegate

- `UITextViewDelegate` is a protocol that defines a set of optional methods can be used to receive editing-related messages for a certain `UITextView`
- It is possible to set a delegate for a `UITextView` object with the `delegate` property
- Protocol methods:

```
- (BOOL)textViewShouldBeginEditing:(UITextView *)textView
- (BOOL)textViewShouldEndEditing:(UITextView *)textView
- (void)textViewDidBeginEditing:(UITextView *)textView
- (void)textViewDidChange:(UITextView *)textView
- (void)textViewDidChangeSelection:(UITextView *)textView
- (void)textViewDidEndEditing:(UITextView *)textView
```

Notifications

- iOS provides another way for object interaction, other then message passing
- Notifications are the standard way by which the model can notify the controller of certain event
- **`NSNotificationCenter`** is a class that provides a mechanism for broadcasting information within a program
- A reference to a **`NSNotificationCenter`** instance is retrieved in the following way:

```
NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
```

- Each program has its own **`NSNotificationCenter`**, so it does not need to be created
- Objects can register with a notification center to receive notifications and upon receiving such notification can execute a particular method
- It is important to unregister for notifications when no longer needed to avoid possible crashes
- Typically, register when view appears and unregister when view disappears; if notifications should be received when the view is off screen, unregister in **`dealloc`** (rare)

Registering for Notifications

- Objects can register with the **`NSNotificationCenter`** for certain notifications with the method:

object that is listening for notifications

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

Registering for Notifications

- Objects can register with the **`NSNotificationCenter`** for certain notifications with the method:

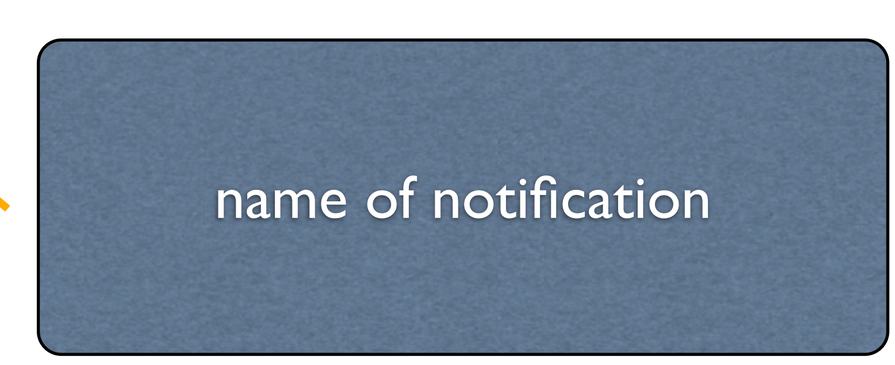
```
- (void)addObserver:(id)notificationObserver
           selector:(SEL)notificationSelector
               name:(NSString *)notificationName
             object:(id)notificationSender
```

selector to execute when receiving the notification

Registering for Notifications

- Objects can register with the **`NSNotificationCenter`** for certain notifications with the method:

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



Registering for Notifications

- Objects can register with the **`NSNotificationCenter`** for certain notifications with the method:

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

source of notifications (nil if any object)

Registering for Notifications

- For instance:

```
[[NSNotificationCenter defaultCenter] addObserver:self
                                         selector:@selector(aMethod)
                                             name:UIKeyboardWillShowNotification
                                           object:nil];
```

Unregistering for Notifications

- When an objects is no longer interested in notifications, it can unregister with the `NSNotificationCenter` with the method:

```
- (void)removeObserver:(id)notificationObserver
                  name:(NSString *)notificationName
                object:(id)notificationSender
```

- For instance:

```
[[NSNotificationCenter defaultCenter] removeObserver:self
                                                name:UIKeyboardWillShowNotification
                                              object:nil];
```

Generating Notifications

- An object that needs to generate a notification can post the notification to the `NSNotificationCenter` with the method:

```
- (void)postNotificationName:(NSString *)notificationName
                      object:(id)notificationSender
                    userInfo:(NSDictionary *)userInfo
```

- For instance:

```
[[NSNotificationCenter defaultCenter] postNotificationName:@"MyNotification"
                                                    object:self
                                                  userInfo:nil];
```

Keyboard notifications

- When the system shows or hides the keyboard, it posts several keyboard notifications
- Notifications contain information about the keyboard, such as its size
- Registering for these notifications is the only way to get some types of information about the keyboard
- System notifications for keyboard-related events (names are similar to delegate methods, but these are notifications, which implies a different communication paradigm):
  - `UIKeyboardWillShowNotification`
  - `UIKeyboardDidShowNotification`
  - `UIKeyboardWillHideNotification`
  - `UIKeyboardDidHideNotification`
- The selectors related to these events should be responsible to move the view to make the content visible when the view keyboard appears, and to reposition it when it disappears

Hard-coded Layout

- In storyboard, we can add subviews to a view controller's view by dragging it a view object from the object palette

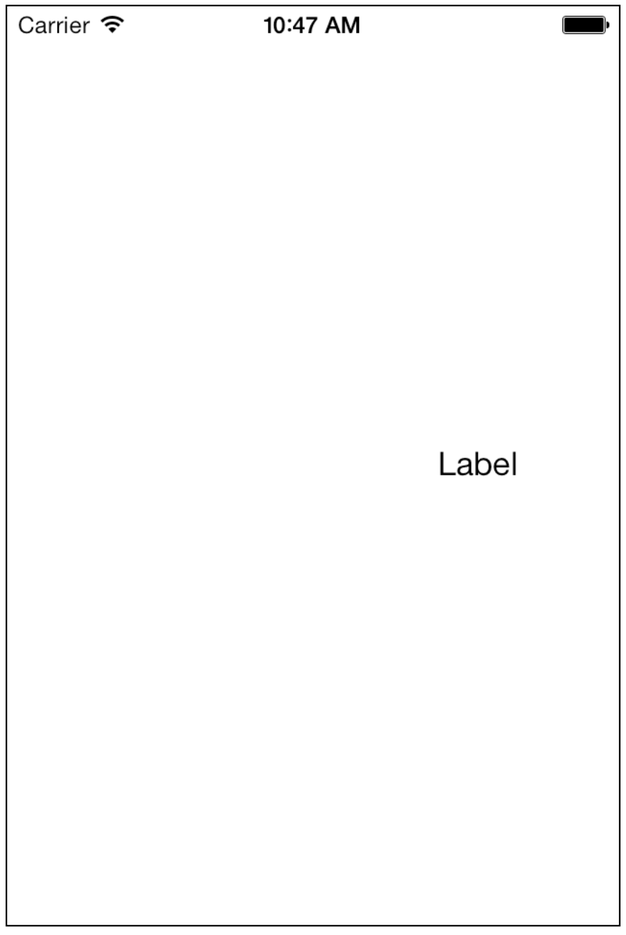
- Let's place it in a "random" place - also using the blue guidelines

- The label has a fixed (static) frame starting at point (226,229)

https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/AutolayoutPG

Hard-coded Layout

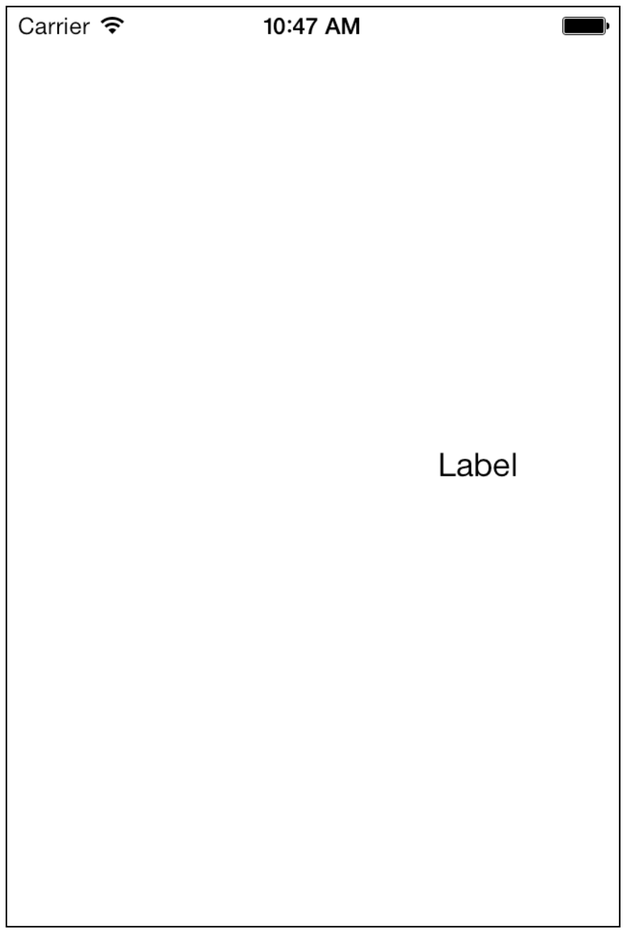
- Everything ok, the label is just where it was placed



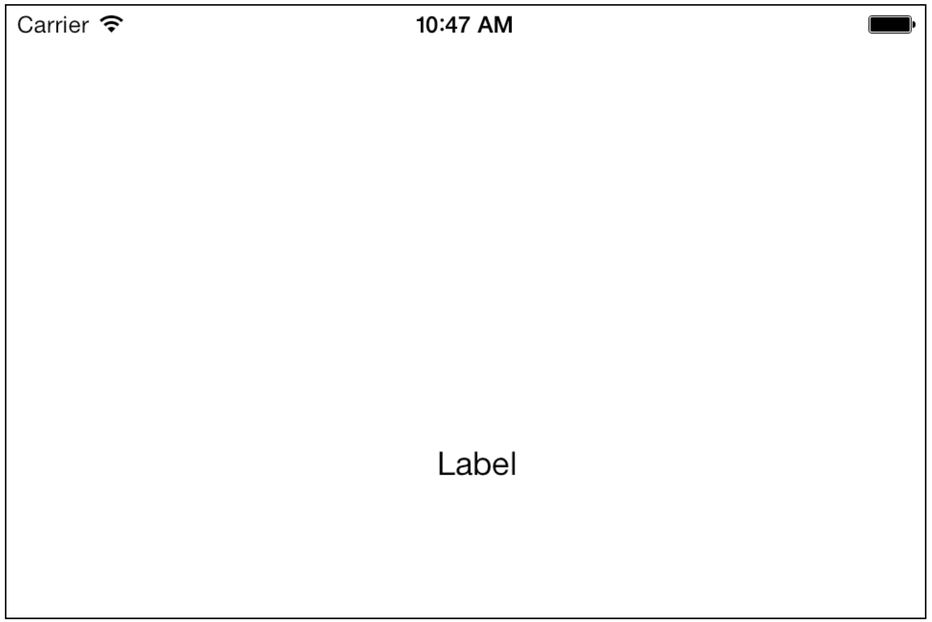
Portrait

Hard-coded Layout

- What happen when the device is rotated?



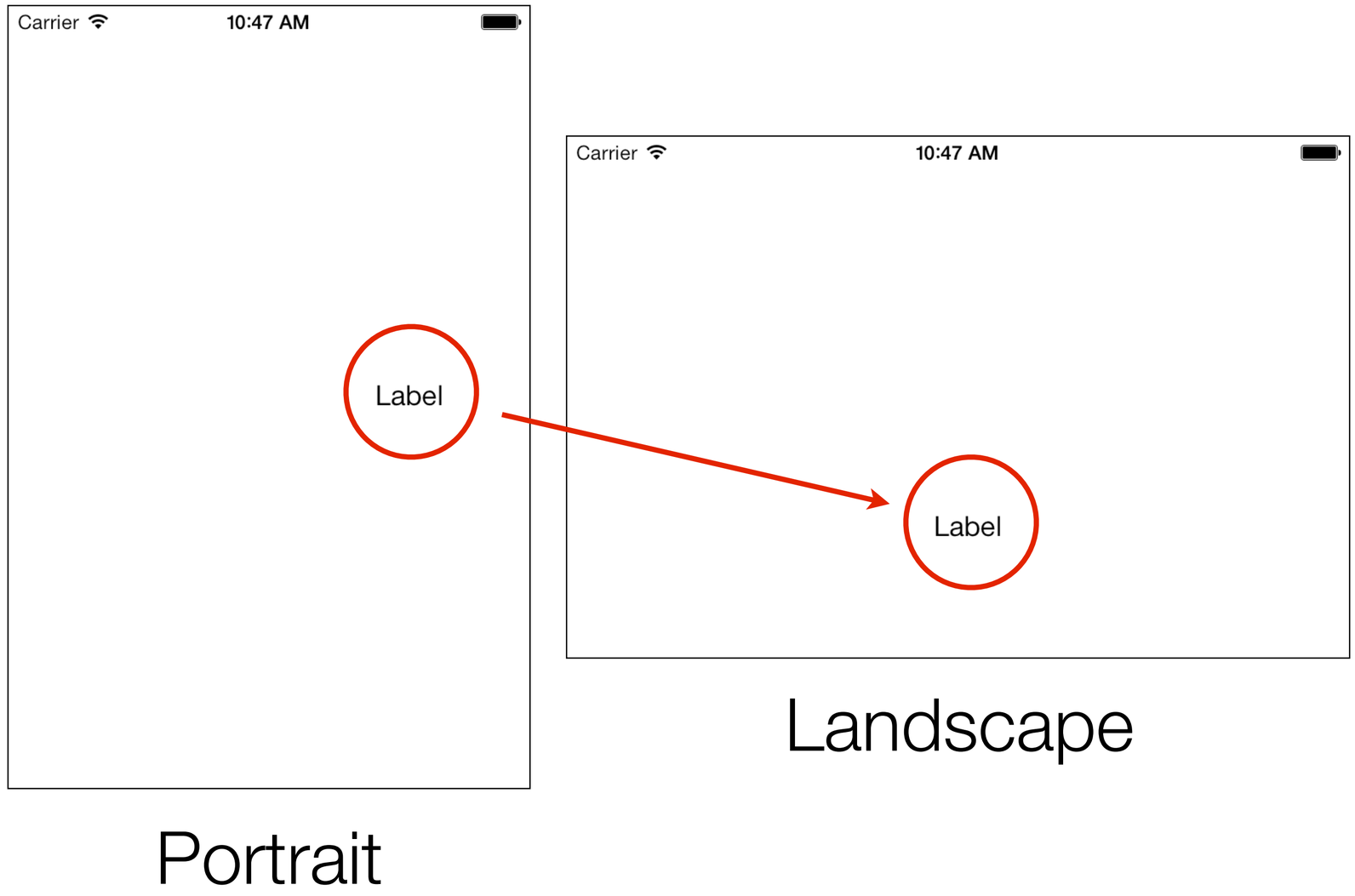
Portrait



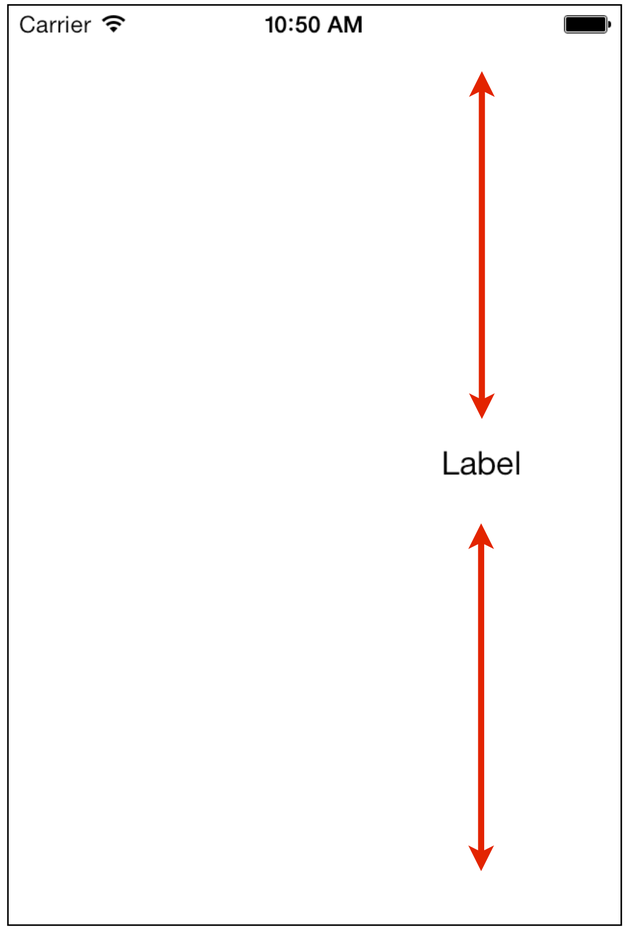
Landscape

Hard-coded Layout

- The label has a fixed (static) frame starting at point (226,229)
- When switching to landscape mode, the label maintains its original frame
- It would be best to have the UI adapt to the orientation automatically (e.g. stay at the center vertically)

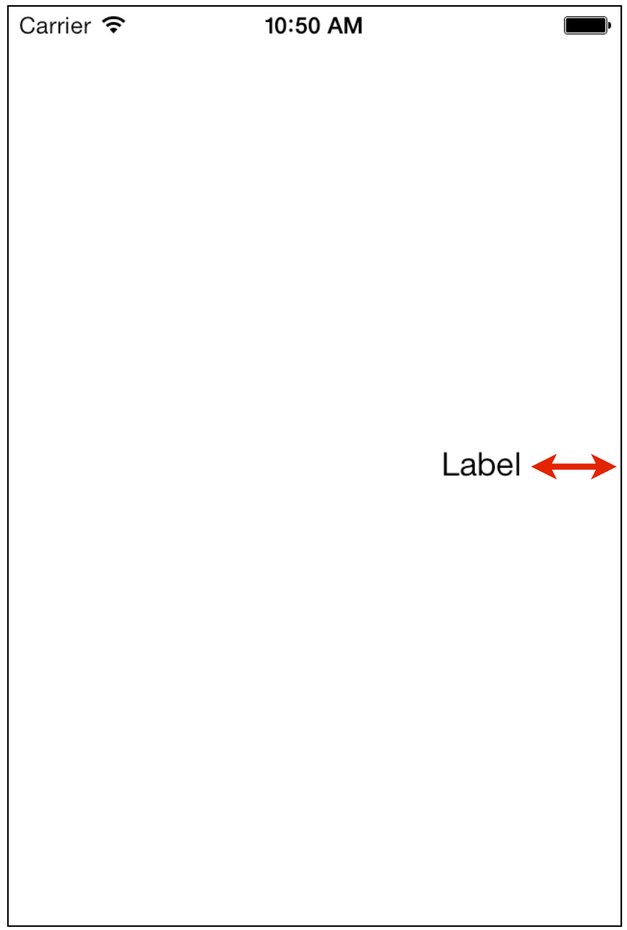


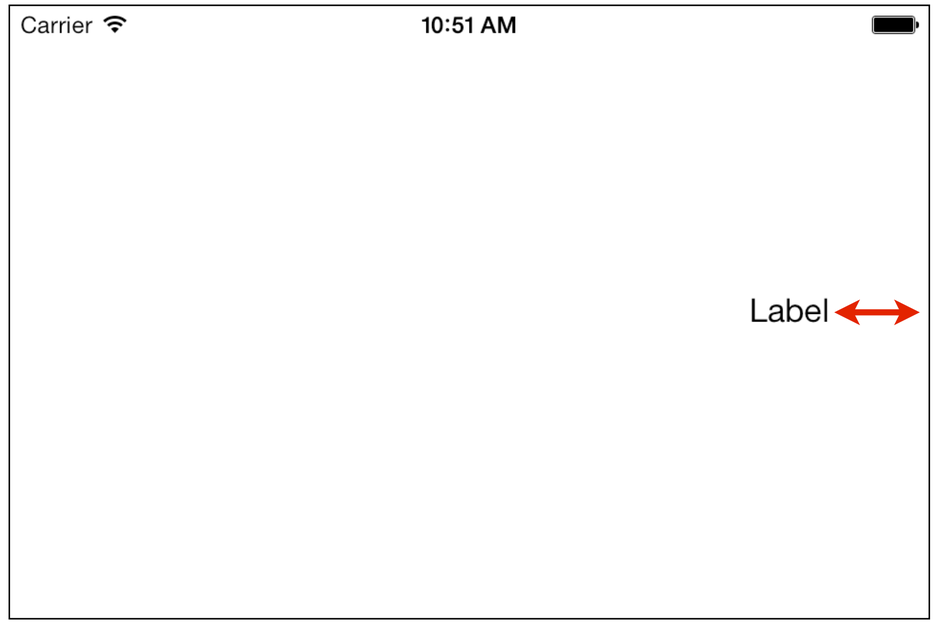
Auto Layout



- The label is vertically centered in the container view

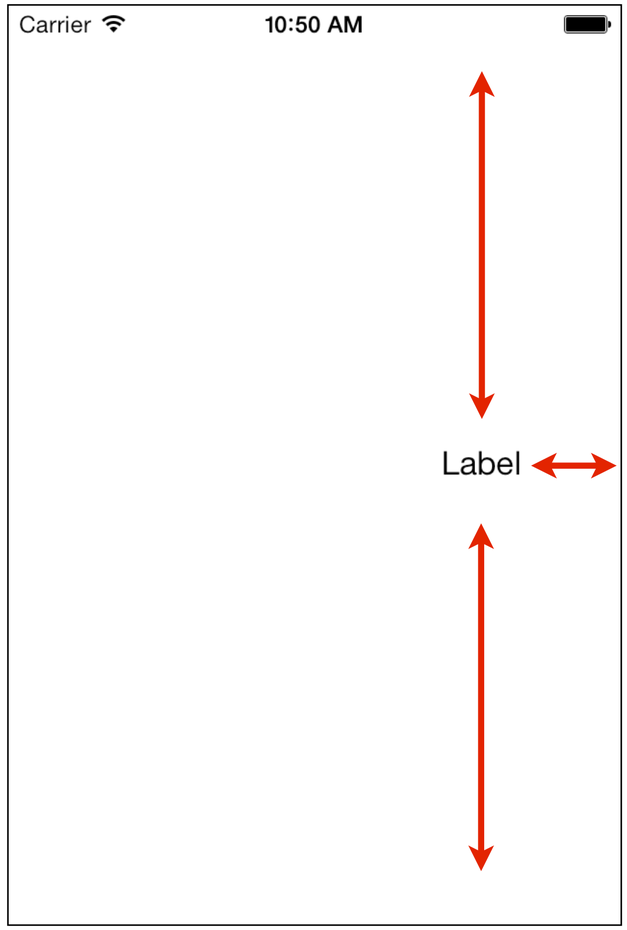
Auto Layout





- The label has a fixed trailing space (to the right edge of the container)

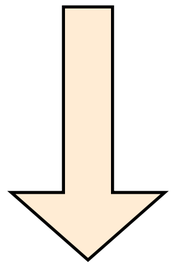
Auto Layout



- The label is vertically centered in the container view
- The label has a fixed trailing space (to the right edge of the container)

Auto Layout

- The label is vertically centered in the container view
- The label has a fixed trailing space (to the right edge of the container)



```
Control.centerY = Superview.centerY

Control.right = Superview.right - <padding>
```

- Auto Layout takes these human-readable rules defined and transforms them into **constraints**

Auto Layout

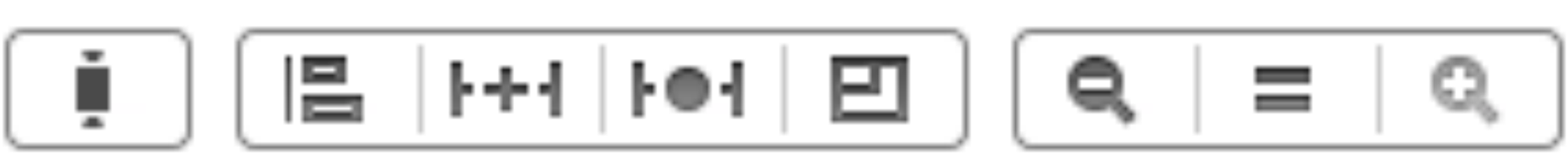
- Auto Layout is a constraint-based, descriptive layout system
- Auto Layout makes it easy to solve many complex layout problems automatically, without the need for manual view manipulation
- With Auto Layout, the layout with constraints is described, and frames are calculated automatically
- Constraints are defined as relationships between the view objects (allows to describe very complex relationships)
- Auto Layout keeps software less bug-prone: it avoids bugs that might be introduced if manual view manipulation were adopted (no code required!)
- Auto Layout makes it easy to deal with:
  - orientation (portrait vs. landscape orientation)
  - different screen sizes (iPhone 4 (3.5” screen) vs. iPhone 5 (4” screen))

Constraints

- A constraint is a sort of mathematical representation of a human-expressable statement
- *The left edge should be 20 points from the left edge of its containing view* translates to `button.left = (container.left + 20)`
- The constraint can be therefore represented by a formula: **y** = **m***x + **b**, where
  - **y** and **x** are attributes of views
  - **m** and **b** are floating point values
- An attribute is one of *left*, *right*, *top*, *bottom*, *leading*, *trailing*, *width*, *height*, *centerX*, *centerY*, and *baseline*

Using Auto Layout

- Auto Layout is managed visually in Xcode
- In storyboard, in the lower-right corner, are the controls for using Auto Layout



Using Auto Layout

- Auto Layout is managed visually in Xcode
- In storyboard, in the lower-right corner, are the controls for using Auto Layout

Switch between iPhone 3.5” and iPhone 4” screen sizes

Using Auto Layout

- Auto Layout is managed visually in Xcode
- In storyboard, in the lower-right corner, are the controls for using Auto Layout

Set alignment constraints: create alignment constraints, such as centering a view in its container, or aligning the left edges of two views

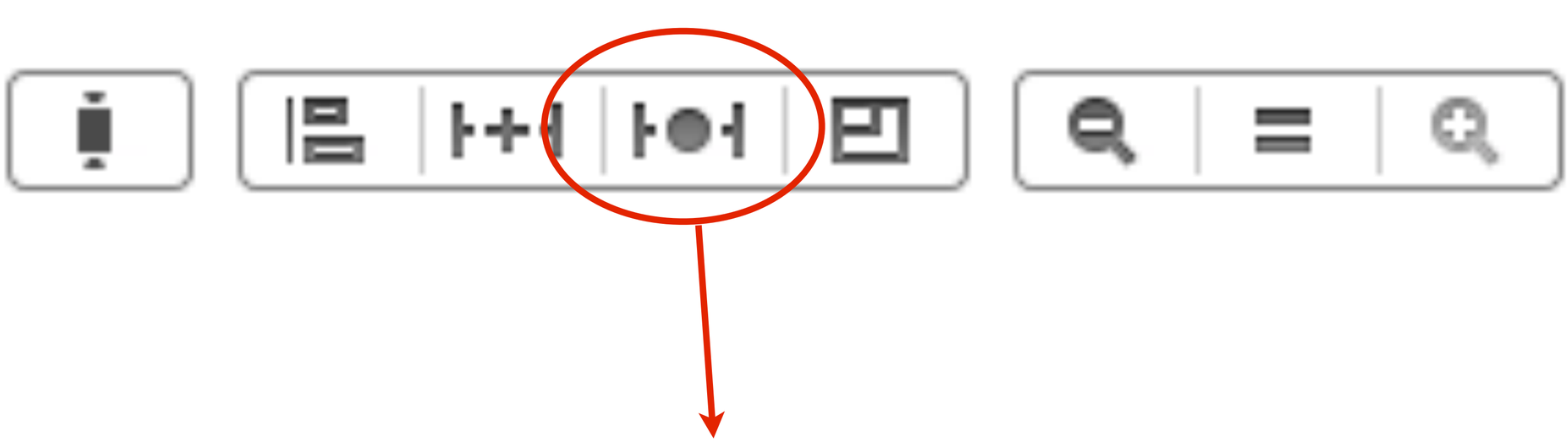
Using Auto Layout

- Auto Layout is managed visually in Xcode
- In storyboard, in the lower-right corner, are the controls for using Auto Layout

Set pin constraints: create spacing constraints, such as defining the height of a view, or specifying its horizontal distance from another view

Using Auto Layout

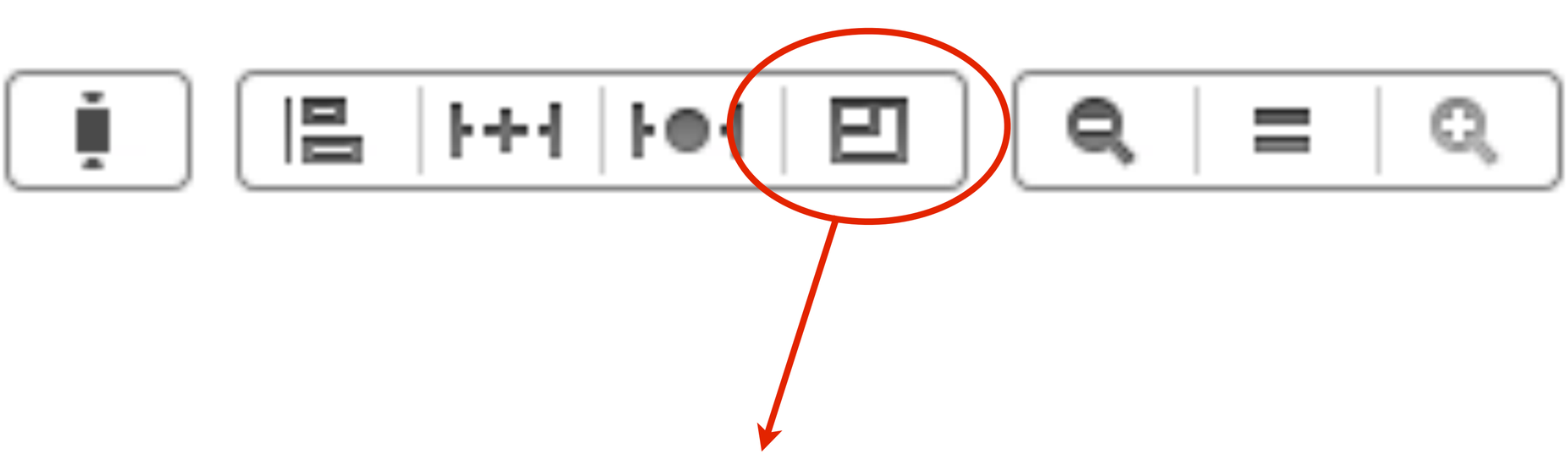
- Auto Layout is managed visually in Xcode
- In storyboard, in the lower-right corner, are the controls for using Auto Layout



Resolve Auto Layout issues: resolve layout issues by adding or resetting constraints based on suggestions

Using Auto Layout

- Auto Layout is managed visually in Xcode
- In storyboard, in the lower-right corner, are the controls for using Auto Layout



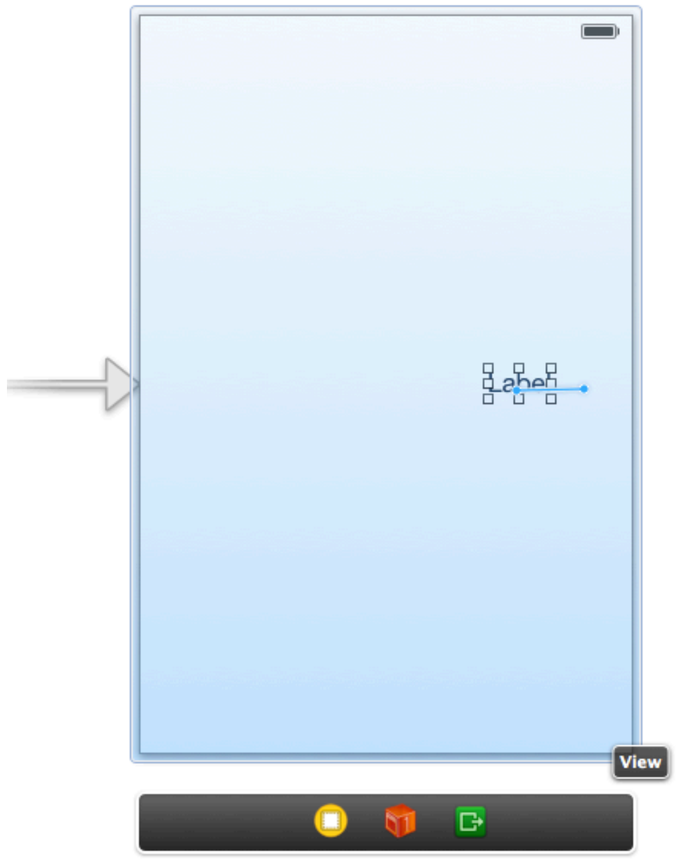
Resizing: specify how resizing affects constraints

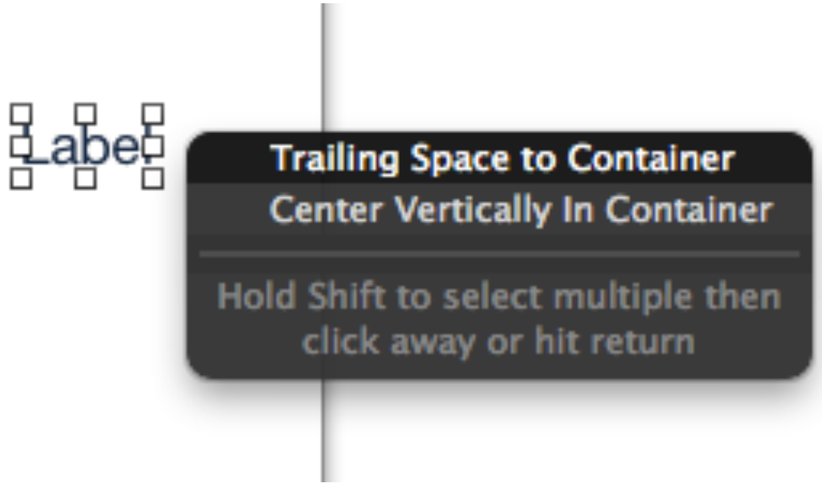
Adding constraints

- Constraints can be added in 3 ways:
  - Control-Drag
  - Align and Pin Menus
  - Adding Missing or Suggested Constraints

Adding Constraints with Control-Drag

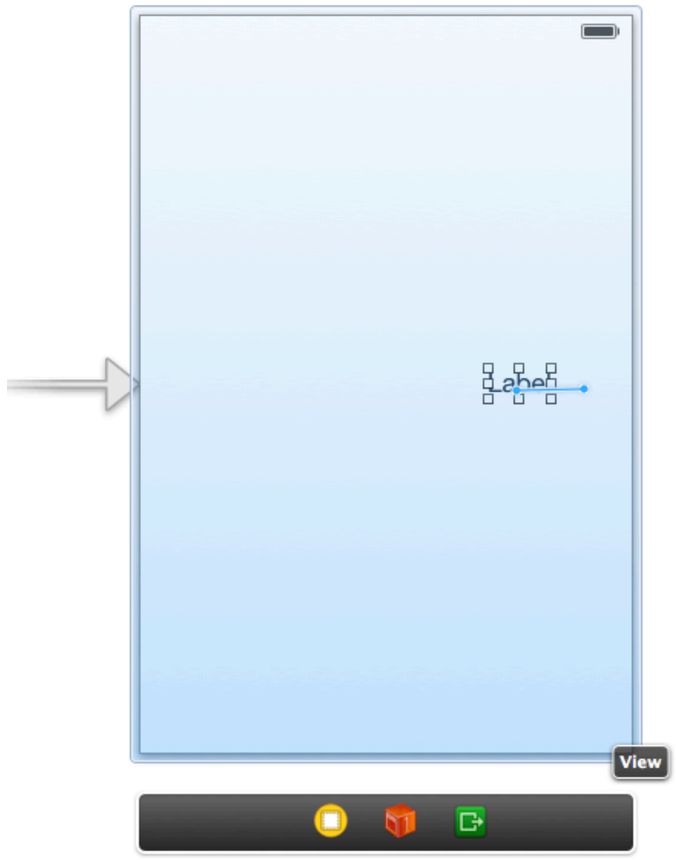
- Fastest way to add a single, specific constraint
- Hold down the Control key and dragging from a view on the canvas (or another view, for relative positioning)

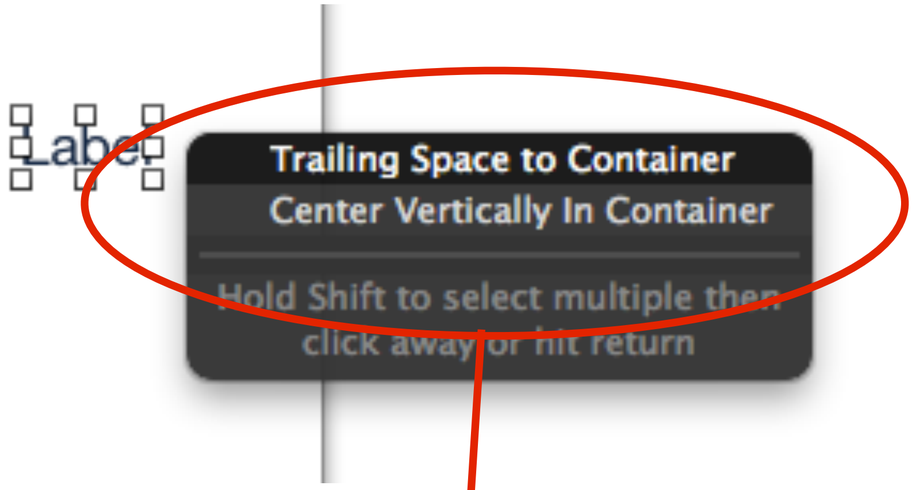




Adding Constraints with Control-Drag

- Fastest way to add a single, specific constraint
- Hold down the Control key and dragging from a view on the canvas (or another view, for relative positioning)

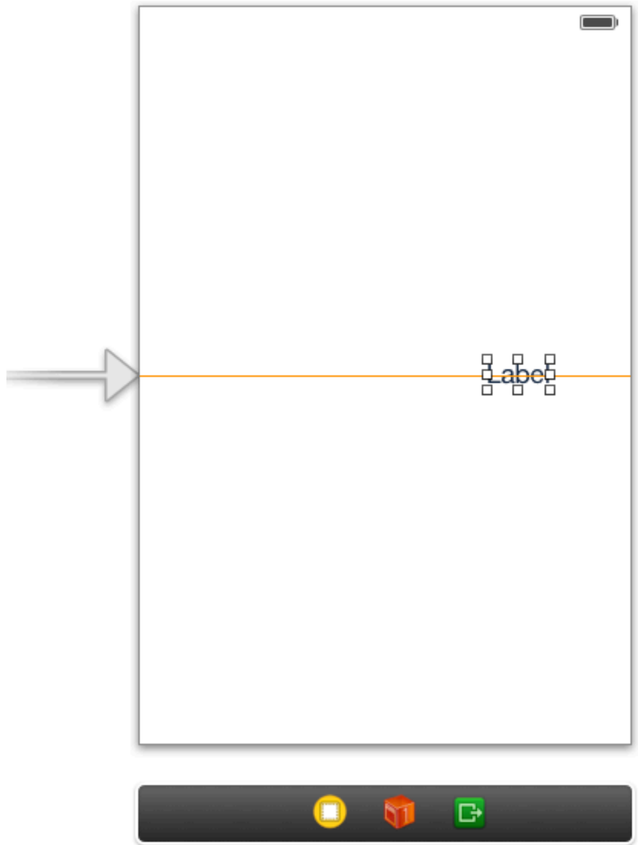




Auto Layout limits the possibilities of constraints appropriately, depending on the type of dragging performed

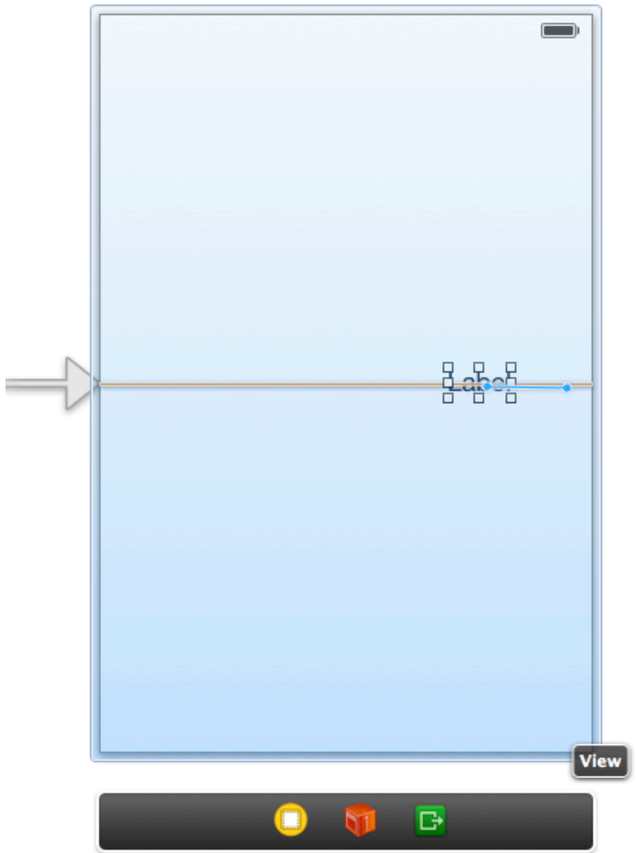
Adding Constraints with Control-Drag

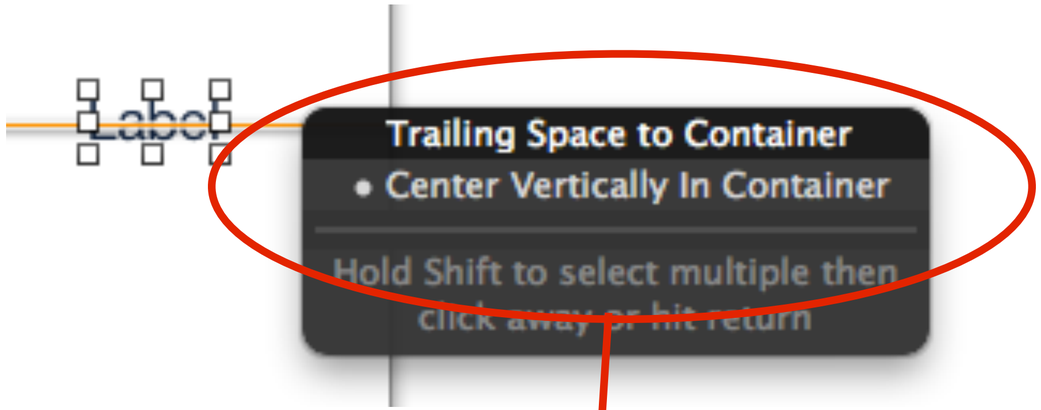
- Select **Center Vertically in Container**
- The constrained is added
- The yellow line is an warning: some constraint is missing (what about the horizontal position?)



Adding Constraints with Control-Drag

- Repeat the Ctrl-drag procedure
- Select **Trailing Space to Container**

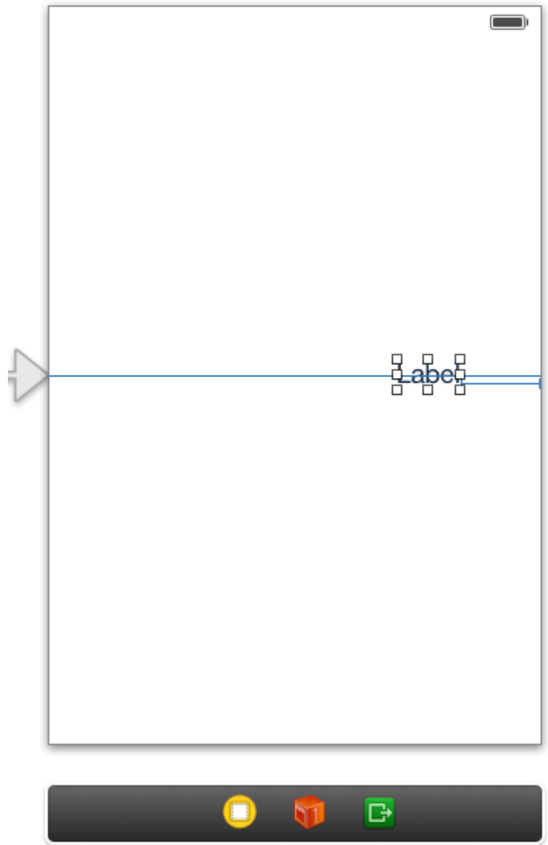




Note that one option is already selected

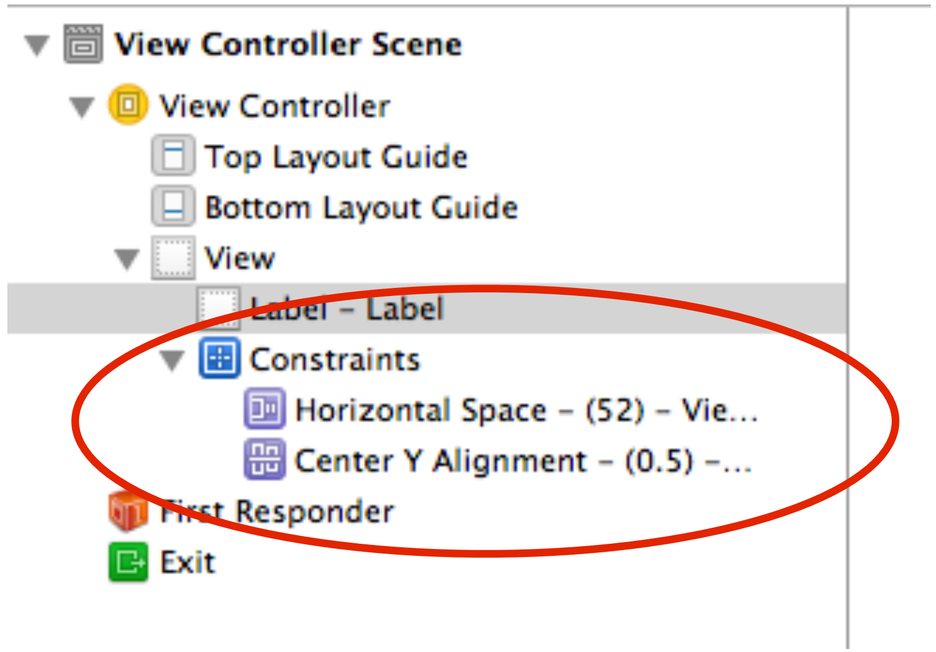
Adding Constraints with Control-Drag

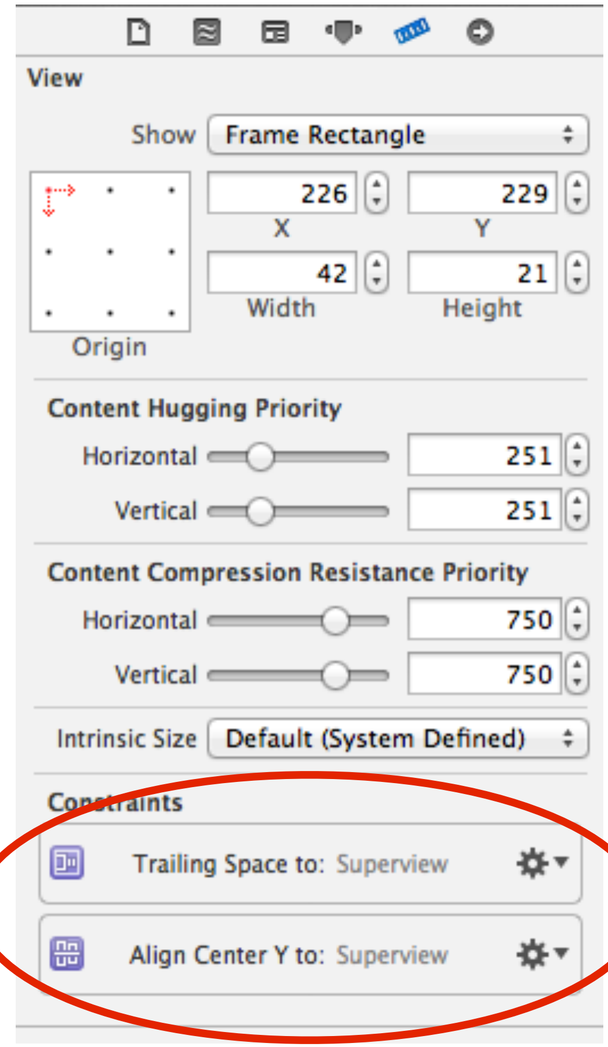
- The second constrained is added
- The yellow line has disappeared and has become blue: all good!
- A new line has appeared for the second constraint



Adding Constraints with Control-Drag

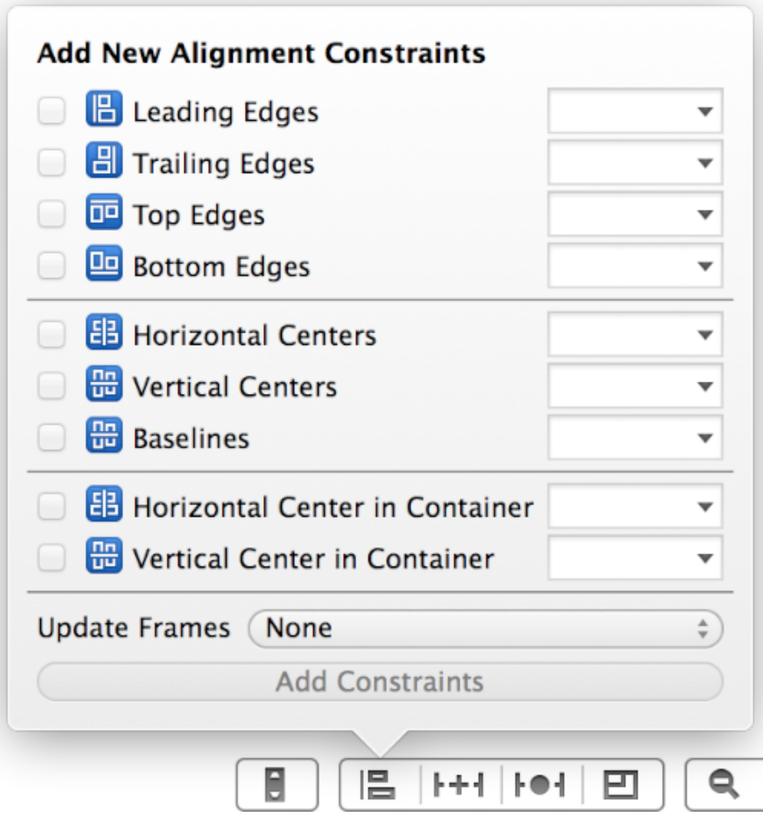
- Constraints are also visible:
  - in the Document Layout view in storyboard
  - in the Attributes inspector

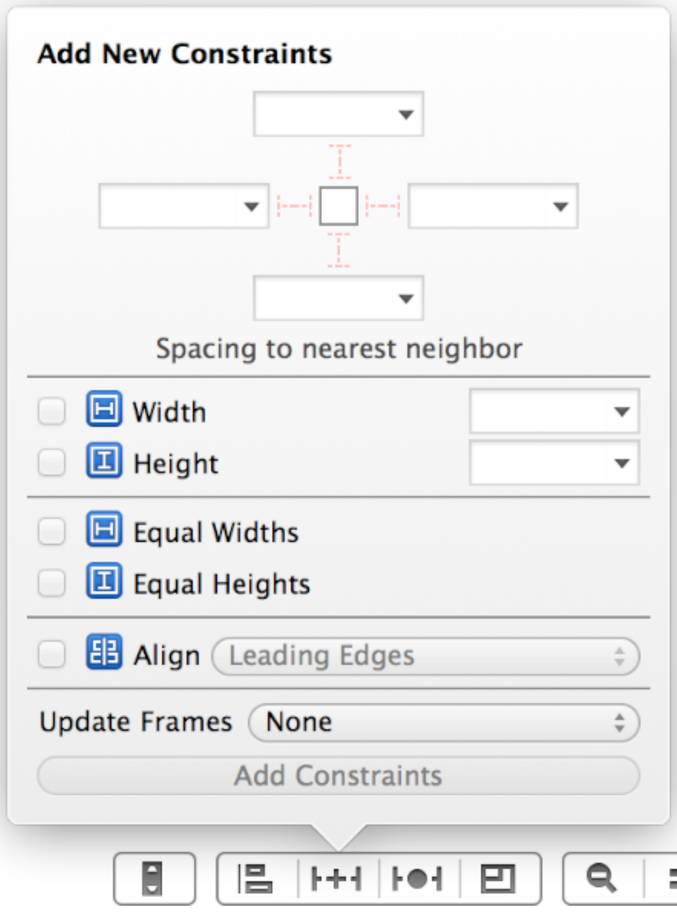




Adding constraints with align and pin

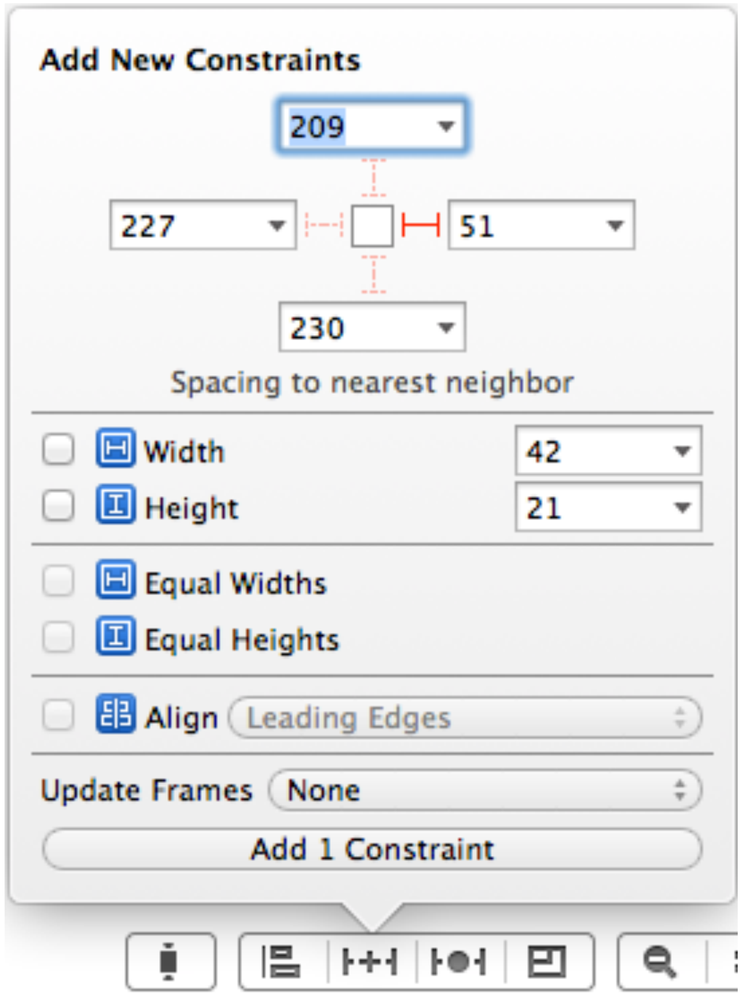
- Let's go back when only the first constraint was set
- In the Document Layout a red arrow appears (it means there are problems)





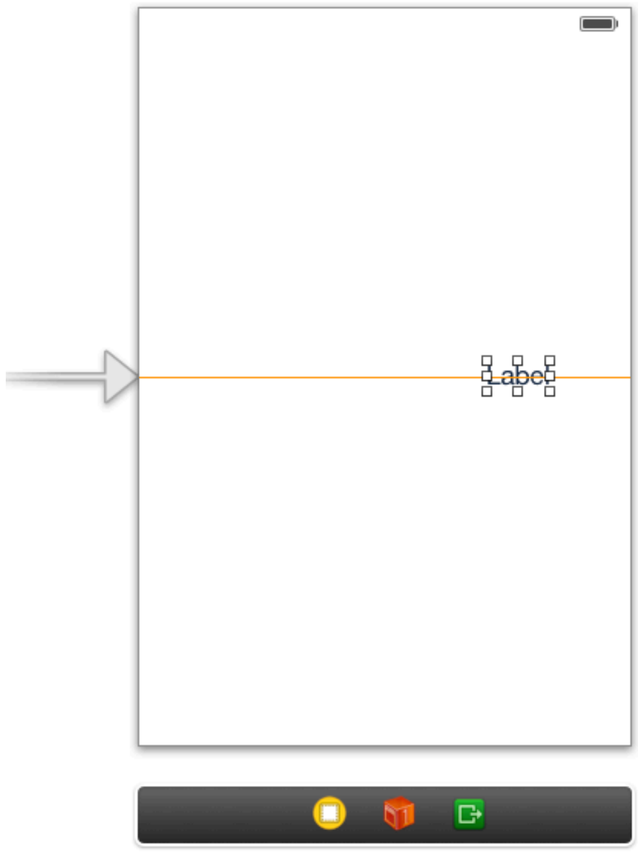
Adding constraints with align and pin

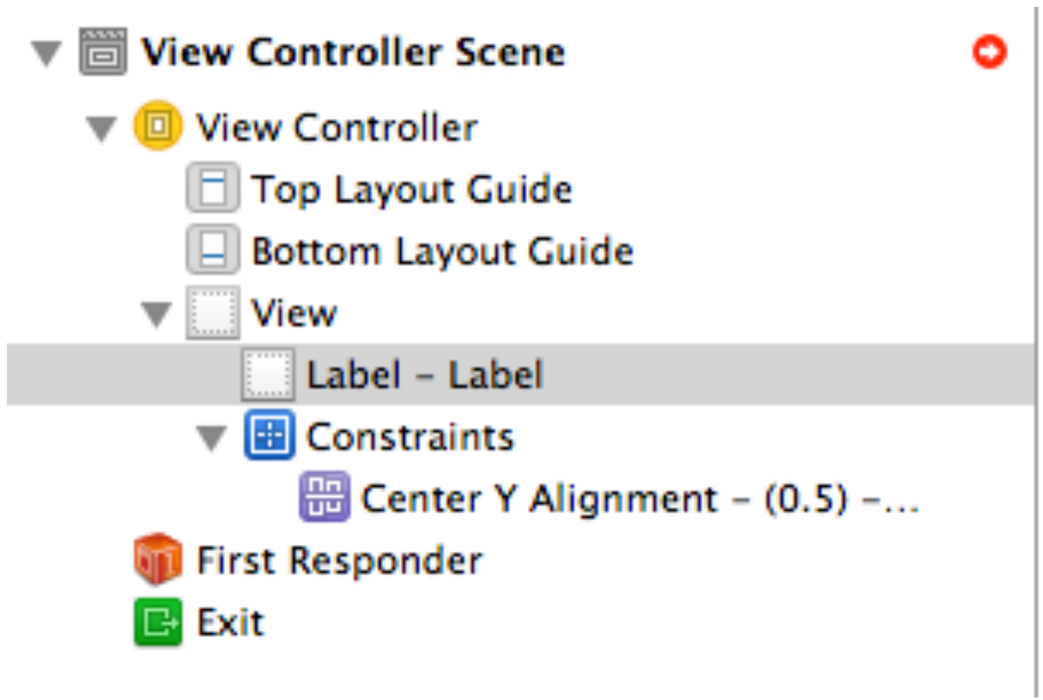
1. Select the checkbox next to the appropriate constraint:
   - to select a “Spacing to nearest neighbor” constraint, select the red constraint corresponding to the appropriate side of the element
2. Enter the constant value
3. Create the constraints by clicking the Add Constraints button adds the new constraints to the selected elements



Adding missing constraints

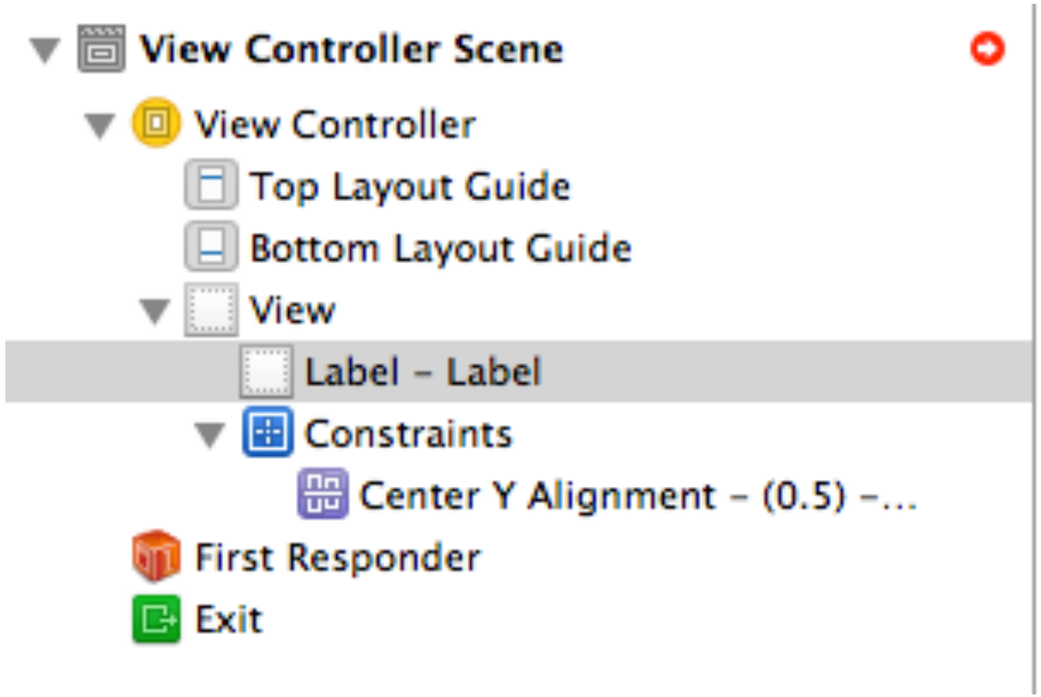
- Let's go back when only the first constraint was set
- In the Document Layout a red arrow appears (it means there are problems)

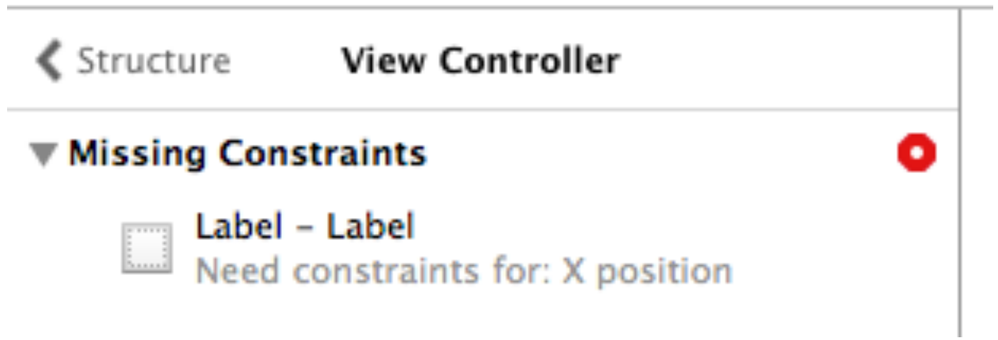




Adding missing constraints

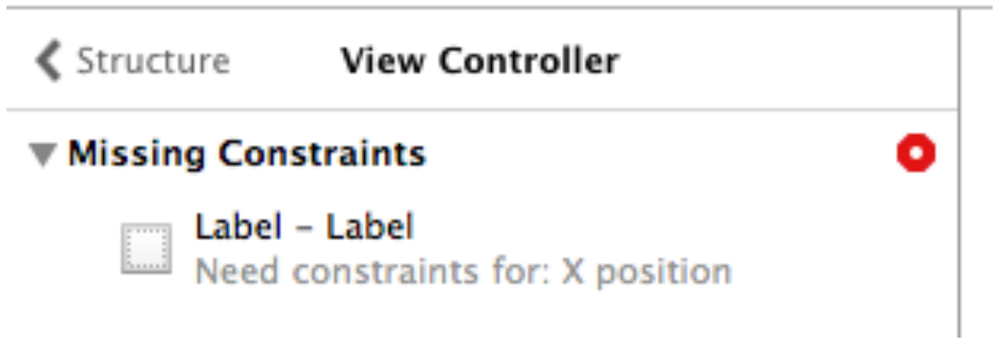
- By clicking on the arrow a new view appear, showing which elements have layout issues (only one in this case)



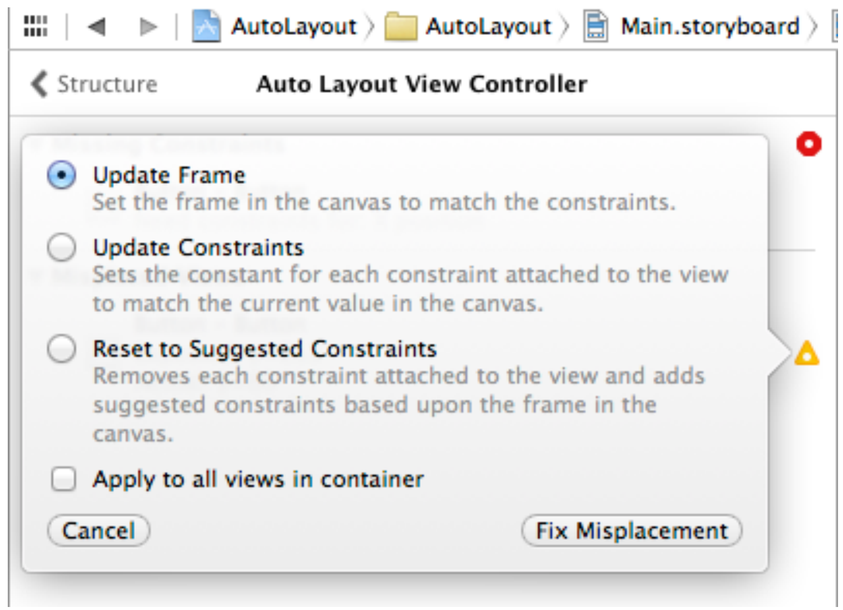


Adding missing constraints

- By clicking on the red circle, a popup appears
- Auto Layout is able to add the missing constraints automatically

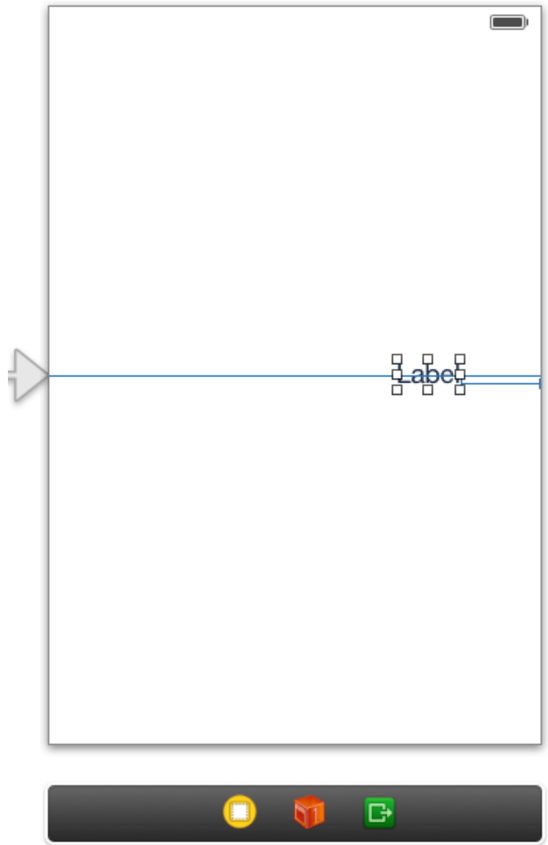


- If many solutions are available (not in this case), another popup shows all the possible actions that may be taken



Adding missing constraints

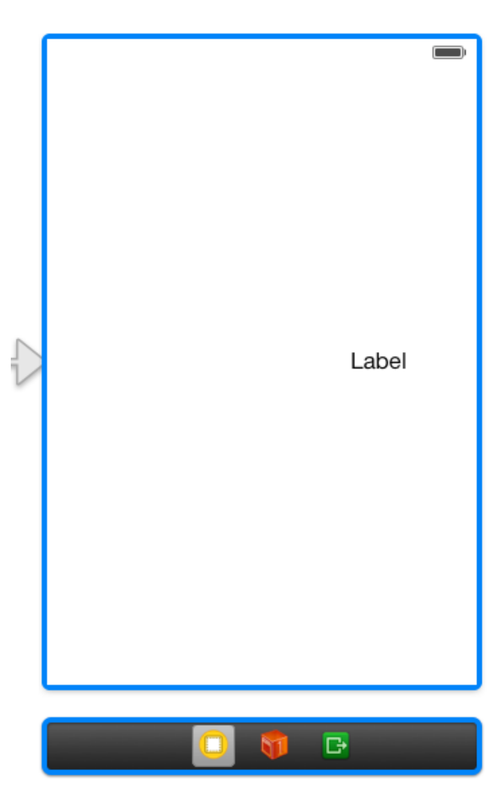
- By selecting Add Missing Constraints, Auto Layout creates the new constraints
- Since this was the only issue, all lines in the view become blue



Adding missing constraints

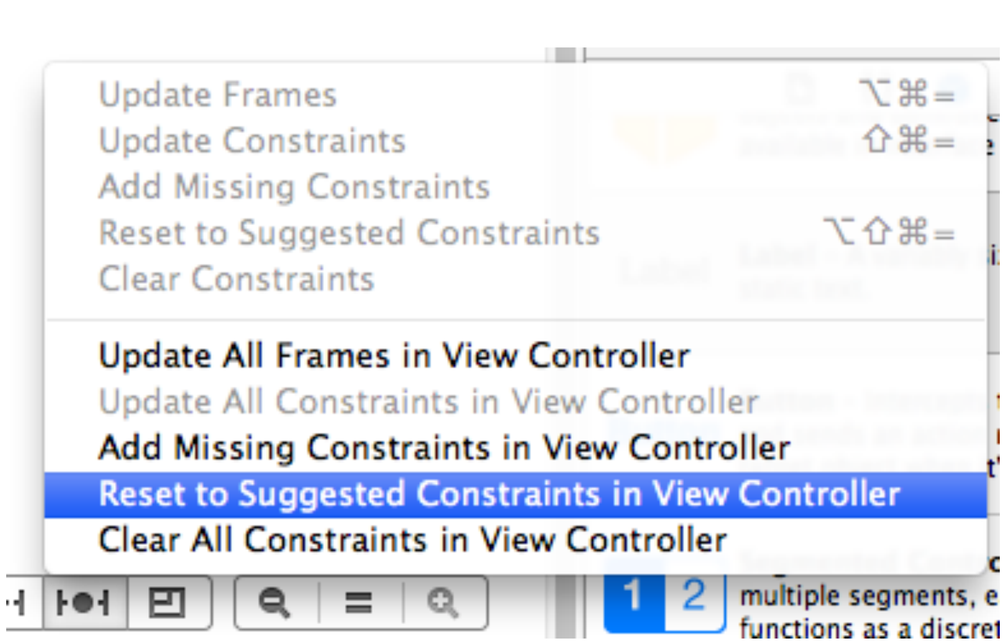
- If no constraints have been, it is possible to ask Auto Layout to set all the constraints for the view:

1. Select the View Controller, by clicking on the View Controller icon



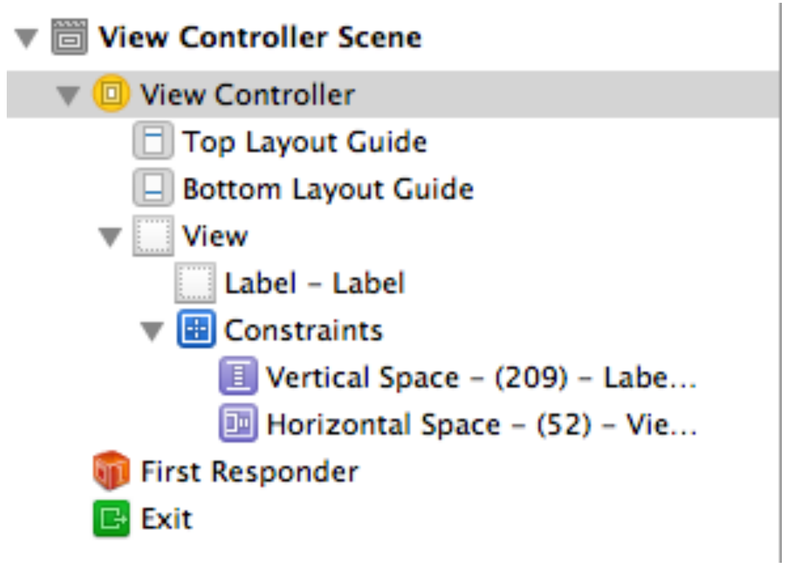
Adding missing constraints

- If no constraints have been, it is possible to ask Auto Layout to set all the constraints for the view:
  1. Select the View Controller, by clicking on the View Controller icon
  2. In the Resolve Issues menu, select **Reset to Suggested Constraints in View Controller**

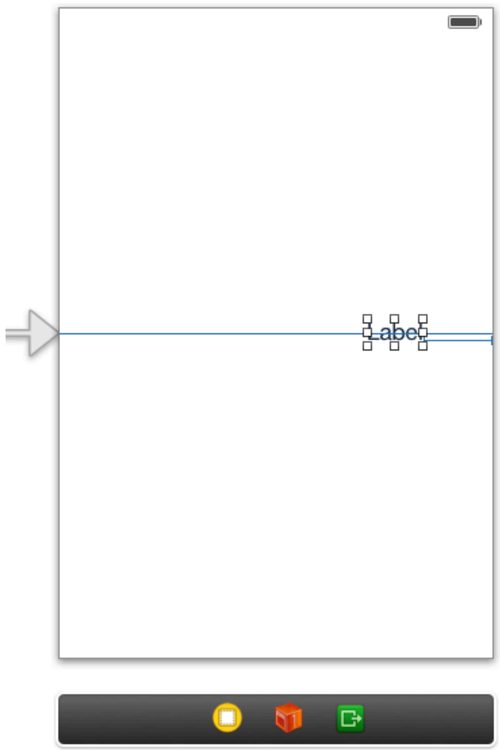


Adding missing constraints

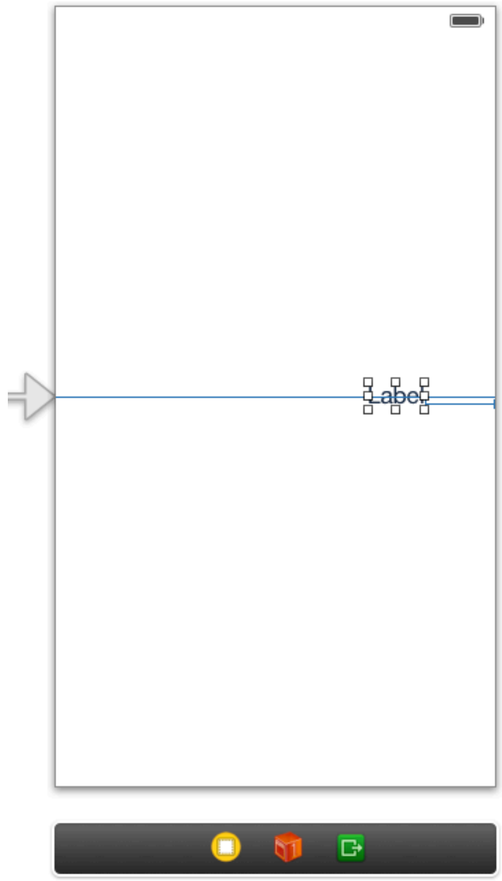
- If no constraints have been, it is possible to ask Auto Layout to set all the constraints for the view:
  1. Select the View Controller, by clicking on the View Controller icon
  2. In the Resolve Issues menu, select **Reset to Suggested Constraints in View Controller**
  3. The constraints are added (note that center vertically is not there!)



Checking constraints for different screen sizes

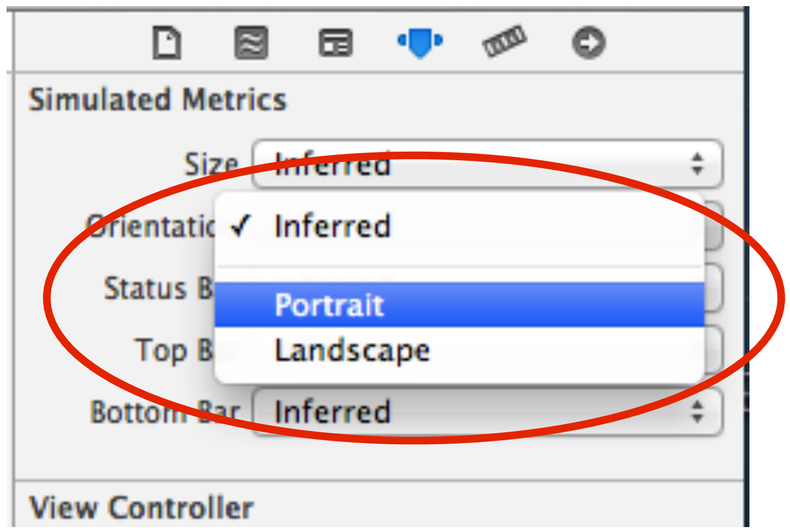


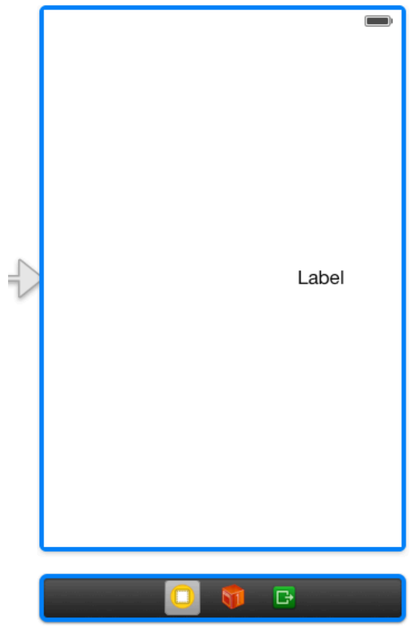
3.5” screen



4” screen

Checking constraints for different orientations





Portrait

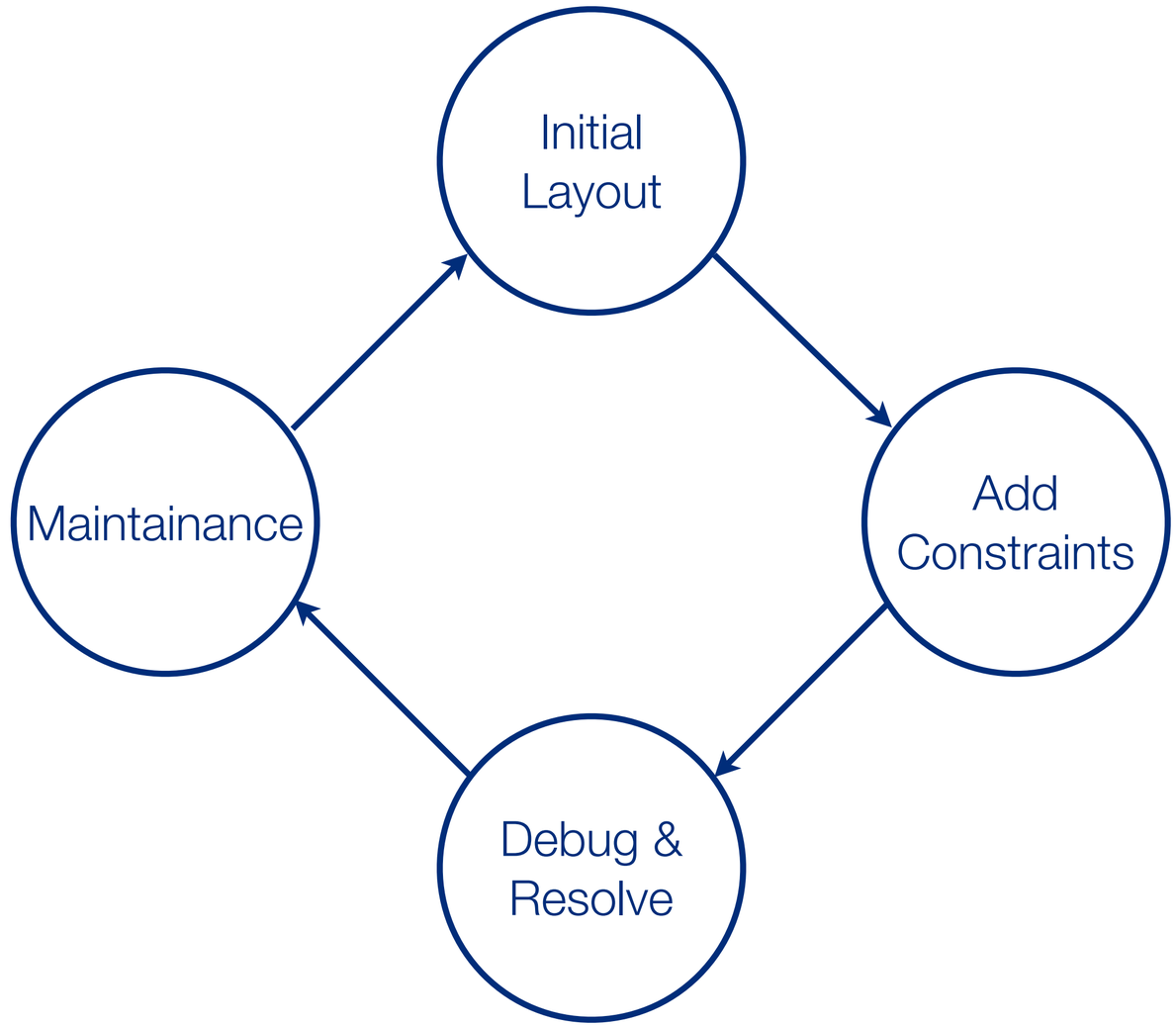
Landscape

Deleting and editing constraints

- Constraints can be selected and deleted or edited
- Delete by using “delete” on the keyboard
- Editing can be made by using the menus shown previously

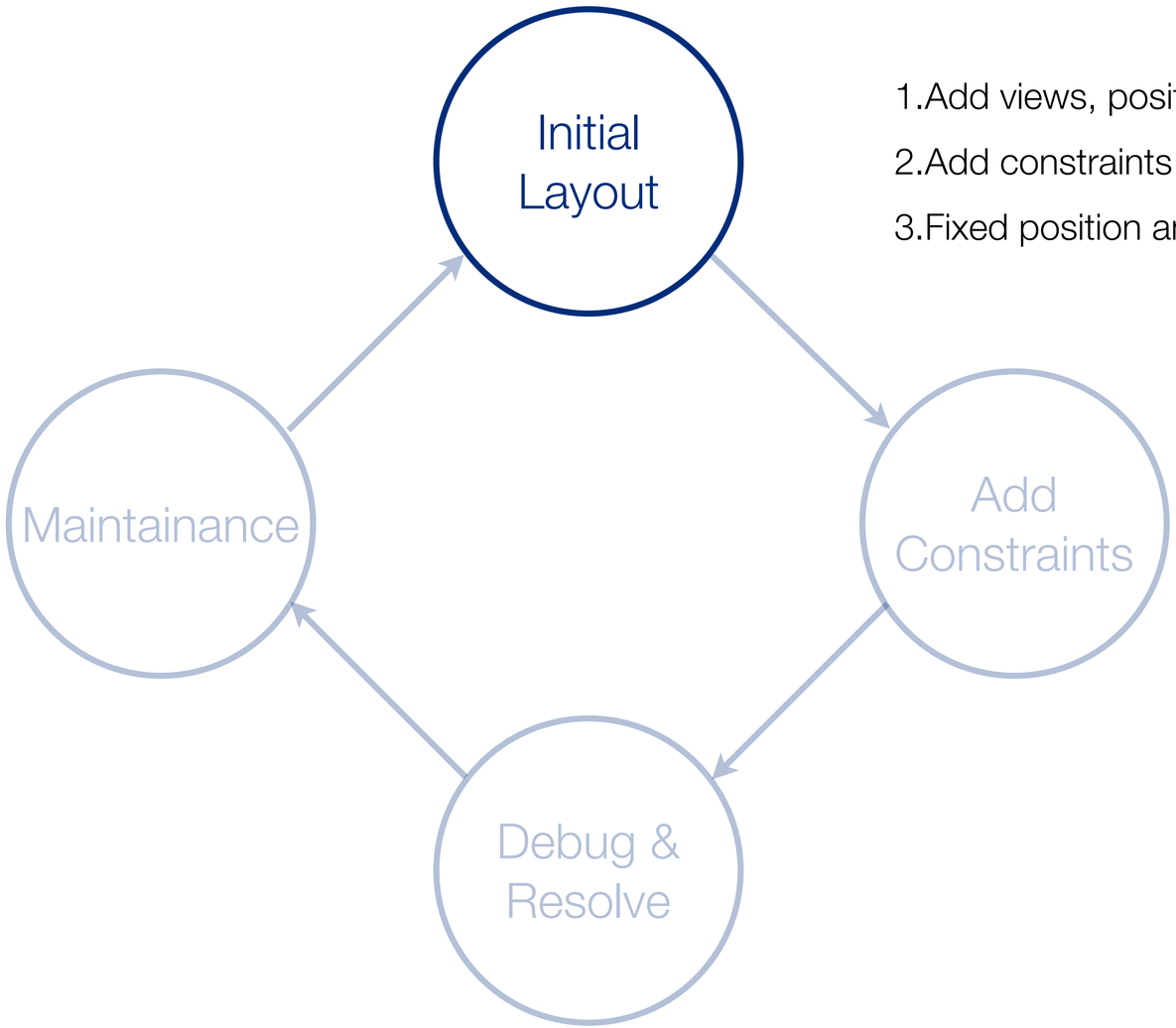
Auto Layout process

- When working with Auto Layout, typically an iterative process is performed



Auto Layout process

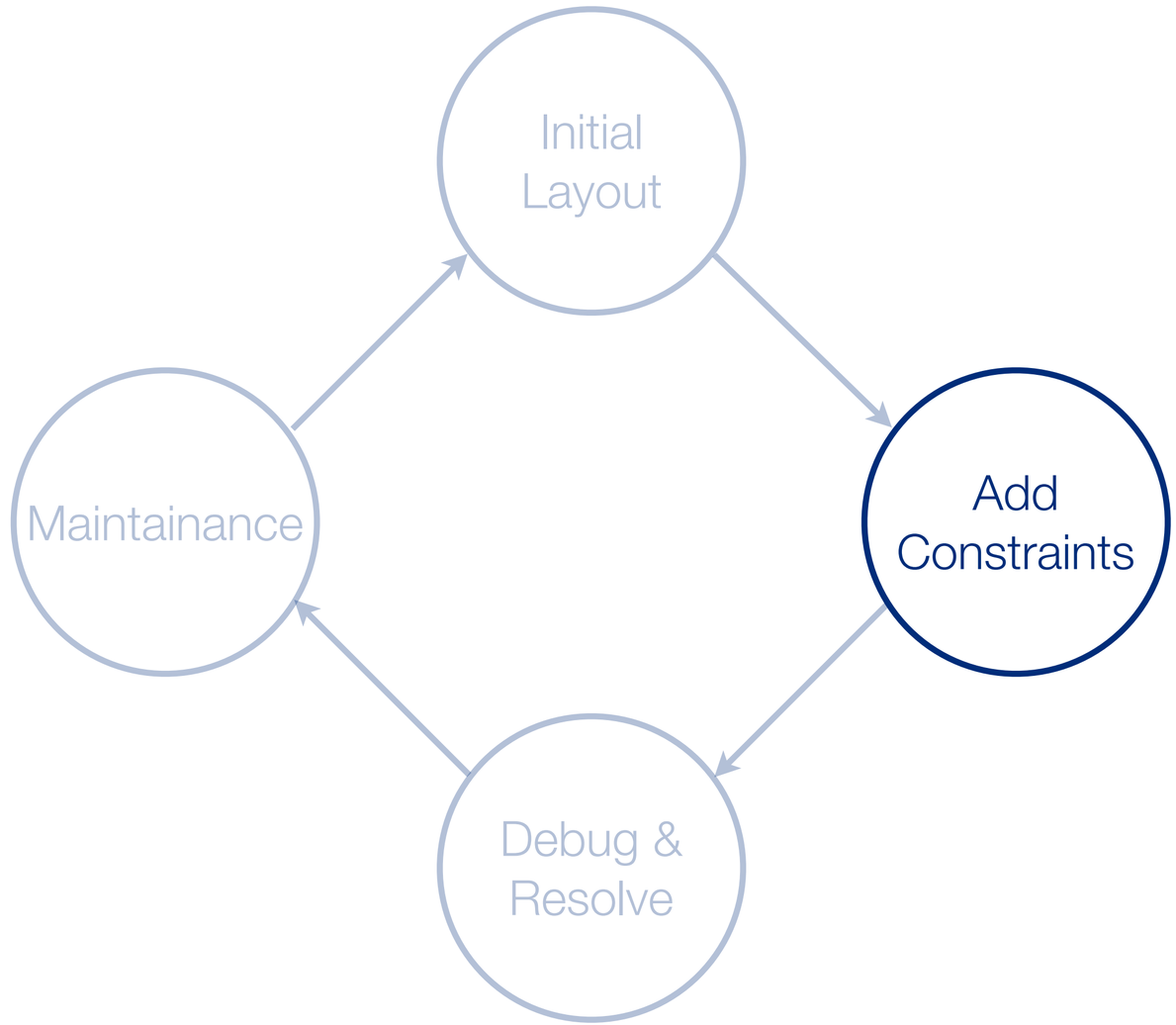
- When working with Auto Layout, typically an iterative process is performed



1. Add views, positioning, and resizing
2. Add constraints when you are ready
3. Fixed position and size

Auto Layout process

- When working with Auto Layout, typically an iterative process is performed

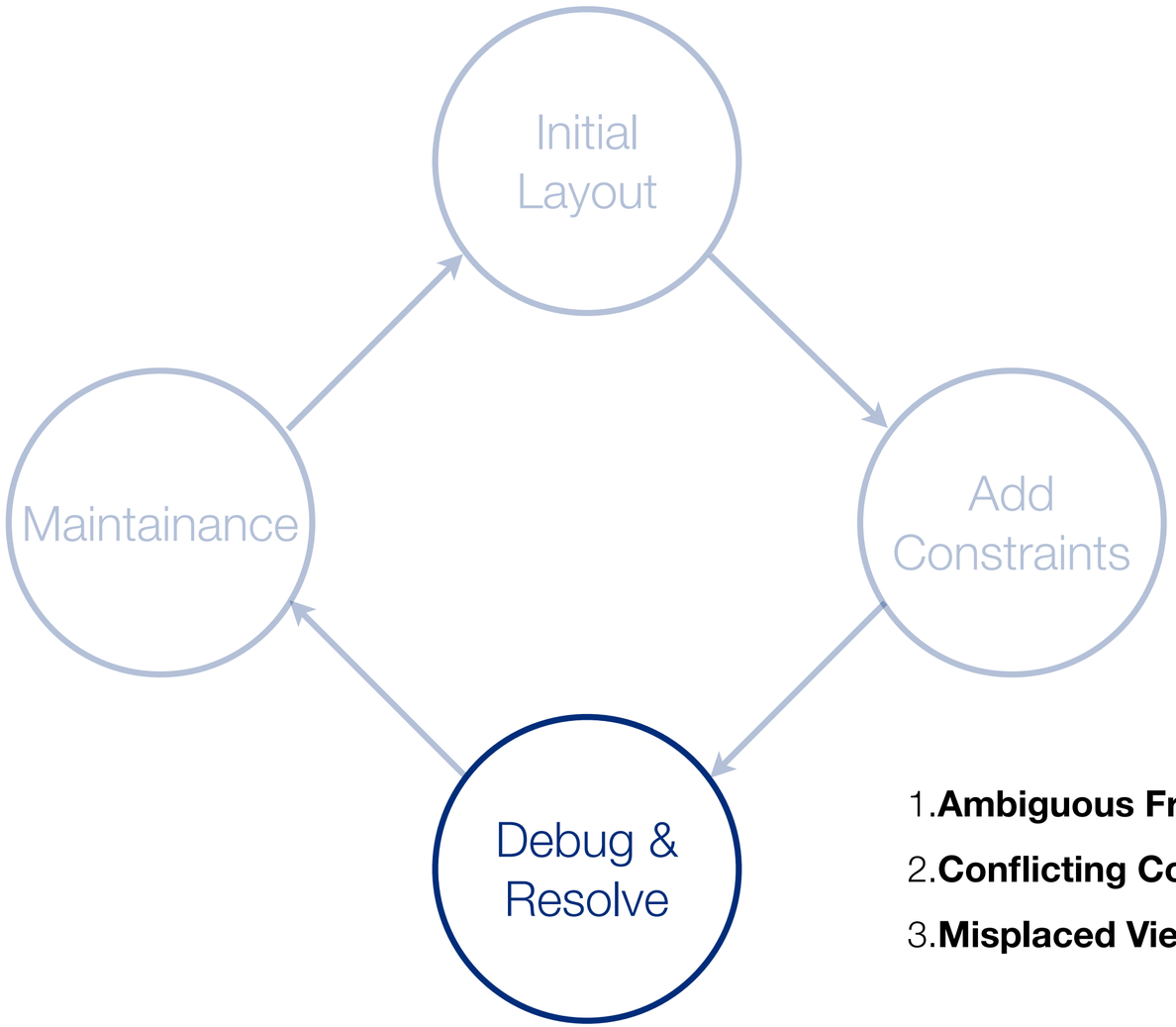


1.Check all changes keep things as they are supposed to be

1.Direct manipulation: Control drag between views
2.Auto Layout resolving menu
3.Constraint addition popovers

Auto Layout process

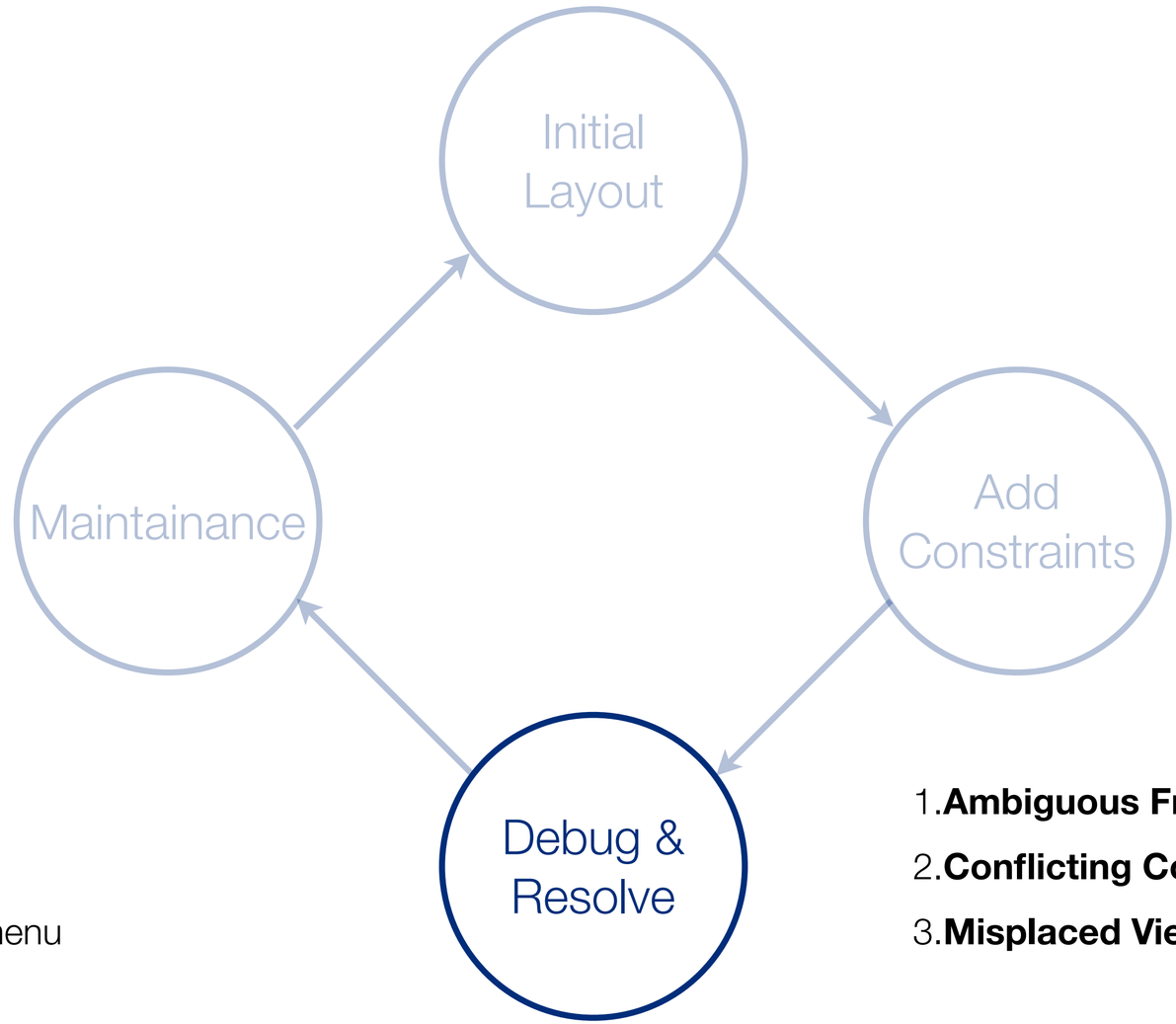
- When working with Auto Layout, typically an iterative process is performed



1. **Ambiguous Frames**: Not enough information
2. **Conflicting Constraints**: Too much information
3. **Misplaced Views**: Mismatched position or size

Auto Layout process

- When working with Auto Layout, typically an iterative process is performed

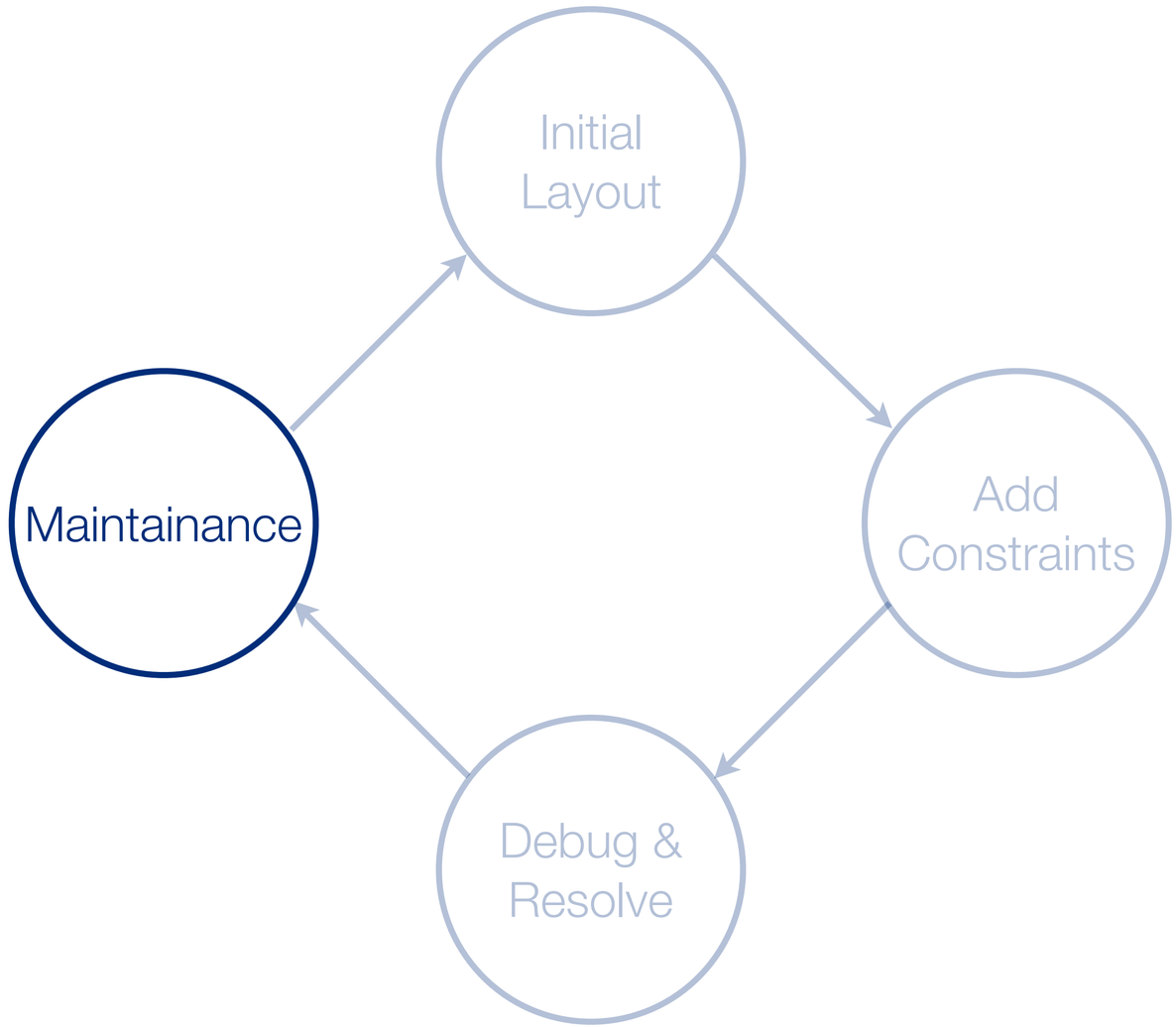


1. Canvas decorations
2. Xcode Issues Navigator
3. Quick fixes via the canvas resolving menu

1. **Ambiguous Frames**: Not enough information
2. **Conflicting Constraints**: Too much information
3. **Misplaced Views**: Mismatched position or size

Auto Layout process

- When working with Auto Layout, typically an iterative process is performed



iOS Development

Lecture 2
iOS SDK and UIKit

Ing. Simone Cirani
email: simone.cirani@unipr.it
http://www.tlc.unipr.it/cirani