

iOS Dev Accelerator

Week1 Day3

- Inheritance
- Auto Layout Review
- Auto Sizing Cells
- UIActivityIndicator
- UINavigationController Review
- Supporting Dynamic Type

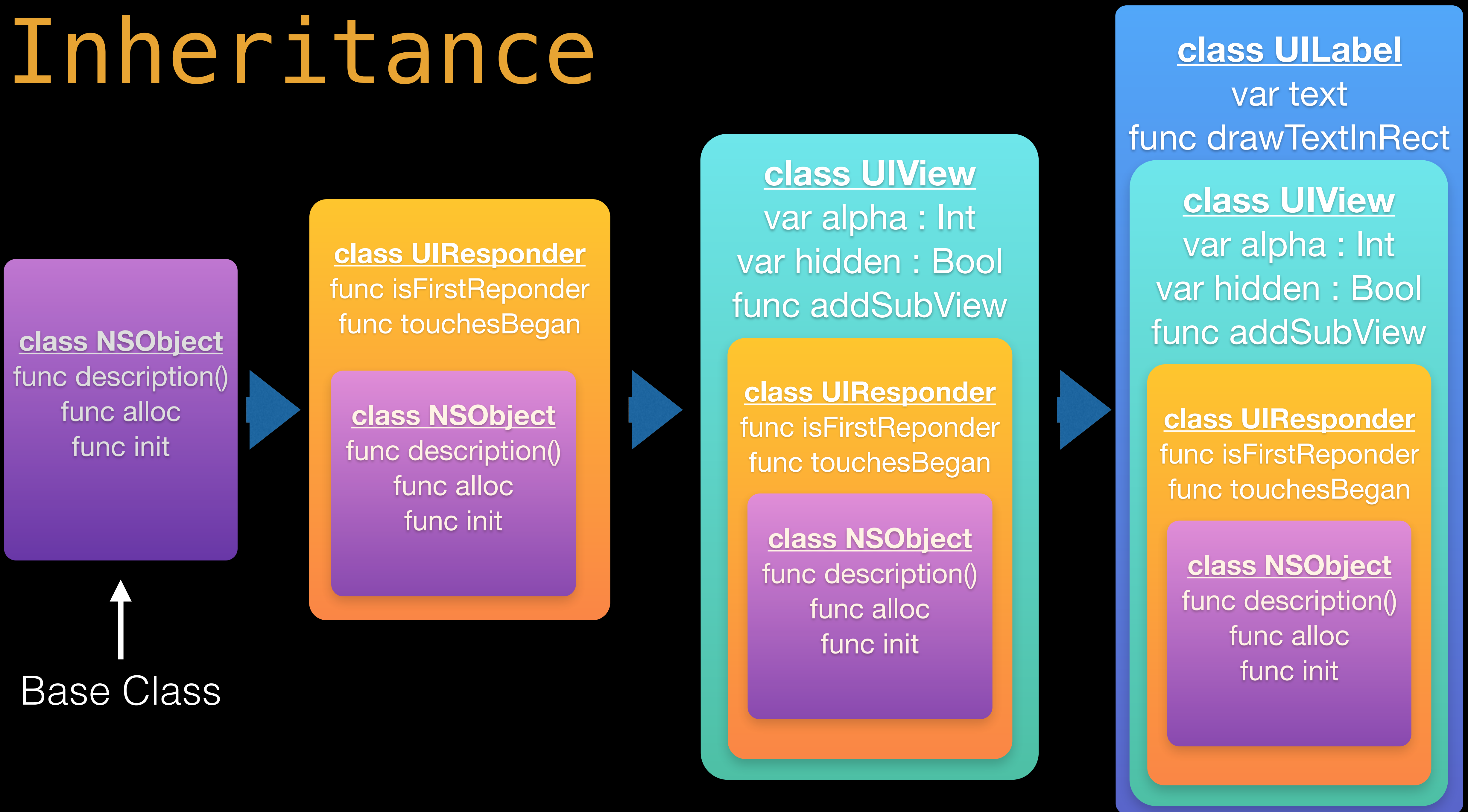
Previous Day Review

Inheritance

Inheritance in Swift

- Inheritance is one of the foundations of object oriented programming, and its no different in Swift.
- A class inherits methods and properties from its super class.
- Not only can a subclass access the properties and methods of its super class, it can also override those methods and properties to refine or modify their behavior.
- Xcode provides support for helping you override by checking to make sure the types, parameters, and return values match up.

Inheritance



Base class

- Any class that does not inherit from another class is known as a base class.
- In Objective-C that class is NSObject. Swift has no universal base class like NSObject
- So any class you create yourself without a super class automatically becomes a base class for you to build upon

Overriding in Swift

- Swift provides the override keyword to help you show clear intent when you are overriding, since accidental overriding can cause hard to diagnose bugs
- Anytime you override a method or property, you can access the super class's version of that method or property simply by accessing it on the variable 'super'

Inheritance

Auto Layout is Important

- With all these damn screen sizes and orientations now days, autolayout is the only way to go for properly laying out your interface.
- Apple chose to support iPhone 4s with iOS 9, now we gotta deal with that screen size for another year. Great.
- Auto Layout is extremely flexible and feature rich, Apple has been working on it for a long time. You can even animate autolayout changes!

Auto Layout Review Fact #1

- **Auto Layout is a constraint-based layout system. Constraints are the fundamental building block of Auto Layout**
- Constraints express rules for the layout of elements in your interface.
- When all of your constraints are considered, there should only be ONE possible layout. If there is more than one, Auto Layout complains because how the heck is going to know which one to pick.

Auto Layout Review Fact #2

- **Constraints are always attached to attribute(s).**
- An attribute is one of left, right, top, bottom, leading, trailing, width, height, centerX, centerY, and baseline.
- So every view has each one of these attributes.



Auto Layout Review Fact #3

- **Constraints take the form of the mathematical equation $y = mx + b$.**
- That translates to `firstItem.attribute = secondItem.attribute * multiplier + constant`.
- Constant : the physical size or offset, in points, of the constraint
- Multiplier: Often times its kept at its default of 1. But it can be useful to help create ratios. Like the firstItems width attribute should be equal to the secondItem's width attribute multiplied by 0.5.

Auto Layout Super Important Fact

- **To make autolayout happy, you must tell it 2 things about every object:**
 - 1. The location of the object**
 - 2. The size of the object***

*Unless the object has an intrinsic content size (like labels!)

Auto Layout Review Fact #4

- **Most constraints have both a firstItem and secondItem in their equation, but width and height constraints often only have firstItem.**
- For example, if you wanted to make a view have a width of 40 points. You would create a constraint with this equation:
 - $\text{view.width} = 0 * 1.0 + 40$
 - theres a 0 where normally there would be a secondItem.

Auto Layout Review Fact #5

- **Constraints are just objects. The class is called NSLayoutConstraint.**
- Every UIView has a method called constraints() that returns an array of all constraints held by that view. It not very common to use this though.
- Later in the course we will create an app without any storyboards and nib/xibs. And we will get plenty of practice creating constraints in code.
- You can even create outlets to constraints from the storyboard.

Demo

Hugging vs Resistance

- Every view that has an intrinsic content size, has attributes which define how constraints are built to show that size
- The first one is called content hugging priority, which dictates how much the view resists growing due to constraints.
- The second one is called compression resistance, which dictates how much the view resists shrinking due to constraints.
- Both of these categories have 2 values, one for horizontal and one for vertical.
- Whats nice is Xcode will usually tell you if you need to adjust those numbers.

Automatic TableView Row Height

Row Height

- Right now our table view gets its row height straight from the storyboard.
- There's a datasource method we could implement, that requests the row height for every row that is displayed
- Prior to iOS 8, having dynamic cell height was a bit of a pain. You had to calculate the size of your text labels based on the font type, font size, and how many characters you had. You would do this in the datasource method mentioned in the previous bullet

New way

- With iOS8, it is much easier:
 - Make sure you are using auto layout in your cell
 - Ensure the elements that are going to be dynamic and dictating the height don't have fixed height constraints
 - Modify the settings of the elements (for label set lines to 0, for text view disable scrolling)
 - Give your table view an estimatedRowHeight and then set its rowHeight to UITableViewAutomatic Dimensions

Demo

UIActivityIndicator

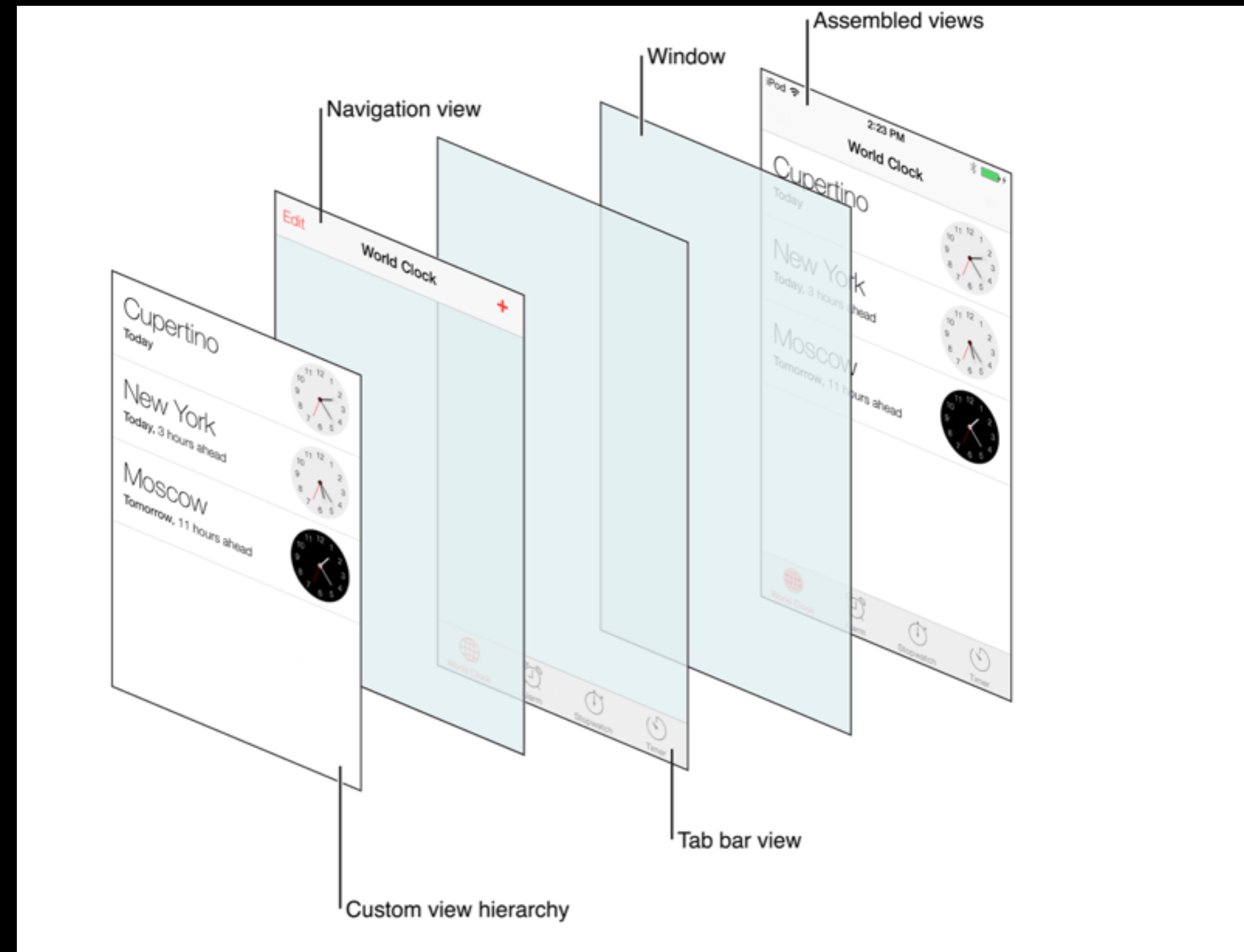
UIActivity Indicator

- An activity indicator shows that a task or process is progressing.
- An activity indicator does 2 things:
 - Spins while a task is progressing and disappears when task is complete
 - Doesn't allow user interaction
- Activity indicator assures the user their task or process hasn't stalled!

UIActivityIndicator Setup

- Drag it out from storyboard, place it in your view hierarchy.
- Should be placed at the bottom of the hierarchy, show it shows up on top
- Give it constraints just any other view (probably center of screen)
- Create an outlet for it
- Call `startAnimating` on it when you want to start showing progress to user, and `stopAnimating` when the task is complete
- It is your responsibility to hide/remove the indicator.

Demo



Navigation Controller Review

2 types of View Controllers

- Conceptually, there are two flavors of view controllers:
 - Content View Controllers: Present your app's content. Used to populate views with data from the model and respond to user actions.
 - Container view controllers: Used to manage content view controllers.
- **Container view controllers are the parent, and content view controllers are the children.**

Navigation Controller Review Fact #1

- **A navigation controller is a container view controller**
- The navigation controller is the parent, all of the view controllers its managing in its stack are considered its children.

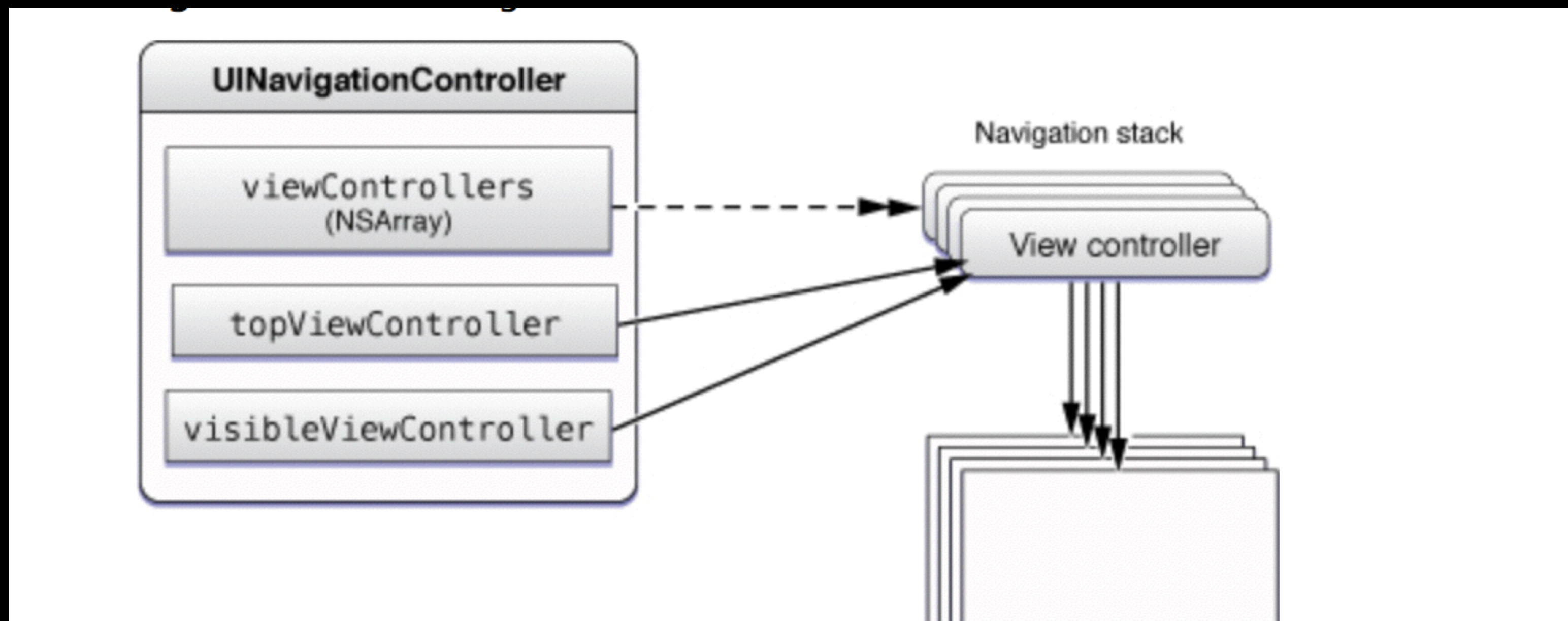
Navigation Controller Review Fact #2

- **Even though navigation controller is a container view controller, it does have a visual representation on screen. This comes in the form of the nav bar, and optionally a toolbar at the bottom.**
- So when you are looking at a view controller that is embedded in a navigation controller, you are looking at two view controllers on the screen at the same time.



Navigation Controller Review Fact #3

- A navigation controller keeps strong leashes on all view controllers its managing. So even if you don't see a view controller on screen (because you pushed some onto of it), its still alive in memory.
- When a view controller is dismissed, aka popped, from the navigation controller, it will likely be destroyed because nothing else has a strong leash on it.



Navigation Controller Review Fact #4

- **Every view controller has a property called navigationController.**
- This property is reference to the navigation controller the view controller is currently in, **IF it is in a navigation controller. This property will of course be nil if there is no navigation controller.**
- So you should always check if it exists first, before interacting with it (optional chaining to the rescue!)

Navigation Controller Review Fact #5

- **UINavigationController provides the content that the navigation bar displays. It is a wrapper object that manages the buttons and views to display in a navigation bar, and each view controller has its own.**
- The managing navigation controller uses the navigation items of the topmost two view controllers to populate the navigation bar with content.
- The navigation bar keeps a stack of all the items, in the exact same order as the navigation controller keeps track of its child content view controllers.
- Each view controller has a property that points to its corresponding navigation item
- The navigation bar has 3 positions for an item to fill: left, center, and right

hiding properties on the nav controller

- `hideBarsOnTap`
- `hideBarsOnSwipe`
- `hidesBarsWhenVerticallyCompact`
- `hidesBarsWhenKeyboardAppears`
- `barHideOnTapGestureRecognizer`
- `barHideOnSwipeGestureRecognizer`

Demo

Access Control

- Access Control is a universal programming concept that allows developers to restrict access to parts of their code from code in other source code files or modules
- It allows you to hide implementation details of your code, and specify a designated interface through which your code should be interacted with

Access Control in Swift

- In swift, you can assign specific access levels to individual types (classes, structures, enumerations)
- You can also assign specific access levels to properties, methods, inits, and subscripts of those types.
- You can also restrict protocols, global constants/variables/functions (not very common)

Access Control in Swift

- Swift provides default access control levels to all of your code, which usually work well for your typical app projects
- If you are writing a regular, single-target app, you don't actually need to specify access controls at all if you don't want to.
- But its still good to know how it works!

Swift Modules and Source Files

- Swift's access control model is based around on the concept of modules and source files
- A **module** is a single unit of code distribution. Like an app or a framework.
- A **source file** is a single Swift file within a module. It is a single file within an app or framework

Access Levels

- Swift provides 3 different access levels for your code:
- **Public** : enables entities to be used within any source file from their defining module (app or framework), and also in a source file from another module that imports the defining module.
- **Internal** : enables entities to be used within any source file from their defining module, but NOT in any source file outside of that module. This is the default access level, and it works great for single target apps!
- **Private** : restricts use of an entity to its own defining source file. Use this to hide implementation details of a specific piece of functionality.

Singleton Pattern



There can only be one

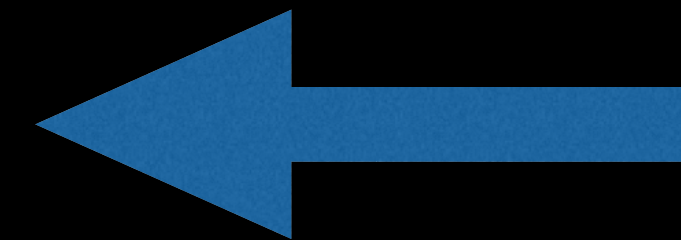
Singleton Pattern

- “A Singleton class returns the same instance no matter how many times an application requests it”
- You obtain the global instance from a singleton class through a factory method (class/type method)
- Apple uses the singleton pattern a lot in Cocoa programming
- Some examples are `UIApplication.sharedApplication()`, and `NSBundle.mainBundle()`
- In our app, it makes sense to have a singleton of our `TwitterService` class so we don't have to worry about passing a reference to them, and also so we know there will only ever be one we will use, since it carries important state (the account)

Singleton in swift

- There wasn't a best practice until recently
- Access Control!
- Thread Safety!

```
class Singleton {  
    static let sharedInstance = Singleton()  
    private init() {}  
}
```



Awesome

Singleton Pattern

- So why do we need a singleton?
- Well, if we didn't use a singleton, we would need to pass a reference around to our twitter service object. Our app only has a few view controllers and it would still be annoying. Think how crappy that would be if we had a much larger corporate sized app.

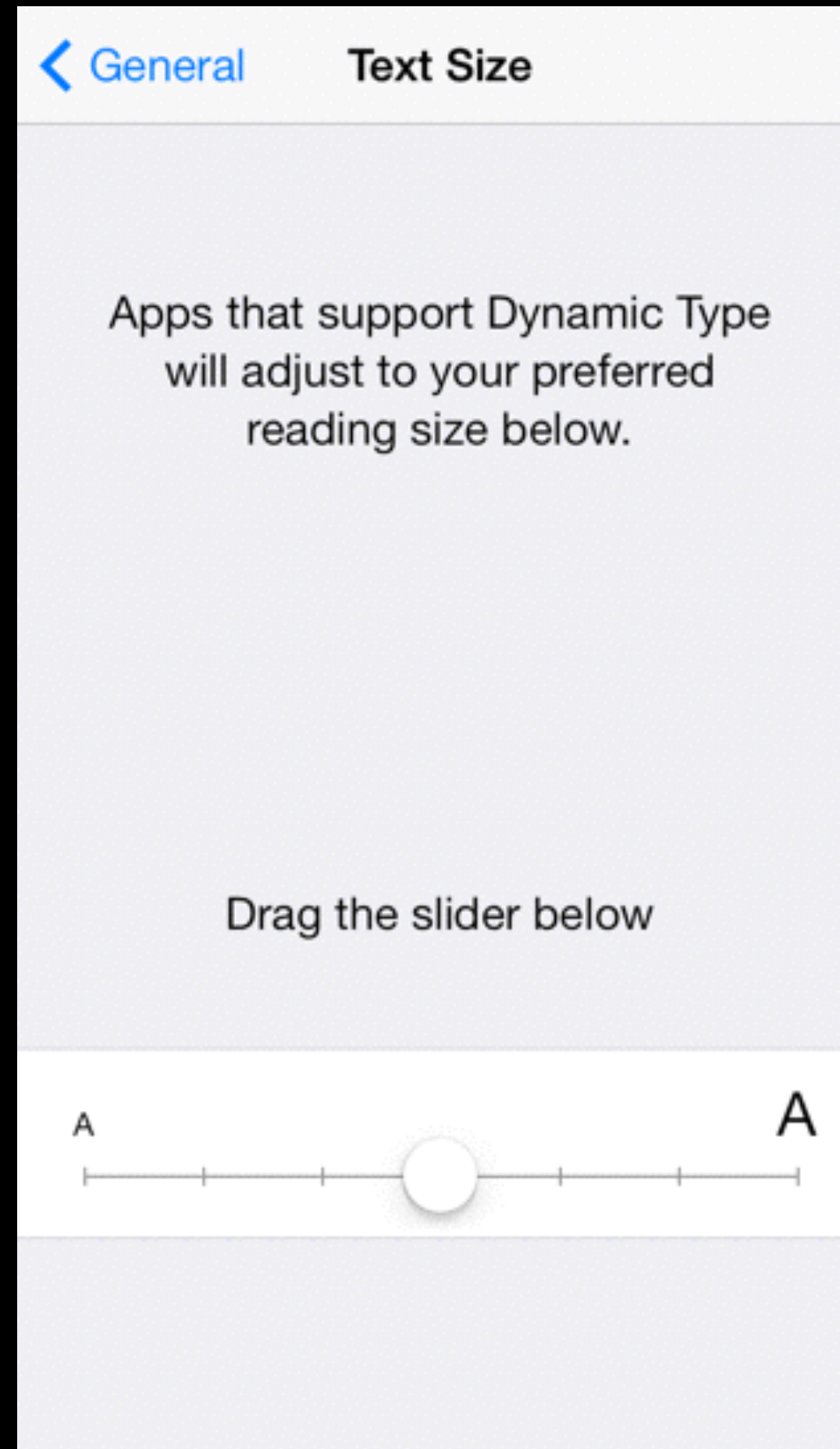
Supporting Dynamic Type

Sourced from useyourloaf.com

Dynamic Type

- Dynamic Type, introduced with iOS 7, provides iOS device users with the ability to specify the preferred universal text size
- Unfortunately that means more work for us app developers, as we have to explicitly support this feature with code. It's doesn't just automatically work out of the box for all apps.

User experience of changing the text size



Settings > General > Accessibility > Larger Text

Dynamic Type

- When the user makes a change to that slider, it changes a universal variable called `preferredContentSizeCategory`
- You can access this variable as a property on the `UIApplication.sharedApplication()` singleton, although you don't really need to.

Dynamic Type and Text Styles

- In order for interface object to participate in dynamic type, it must have its font set to a 'Text Style'
- The available text styles are:
 - UIFontTextStyleHeadline
 - UIFontTextStyleBody
 - UIFontTextStyleFootnote
 - UIFontTextStyleCaption1
 - UIFontTextStyleCaption2

Headline

Subhead

Body

Caption 1

Caption 2

Footnote

Headline

Subhead

Body

Caption 1

Caption 2

Footnote

Headline

Subhead

Body

Caption 1

Caption 2

Footnote

Setting the Text Styles

- So how do you give interface objects, like labels, a text style?
- 2 ways:
 - In Storyboard
 - Programmatically by setting the object's font property with the class method `preferredFontForTextStyle(style:)` on the `UIFont` class

Demo

Detecting when user changes text size

- Whenever the user drags that slider to change the size of the text, our app needs to react appropriately.
- We can do this by listening to a notification that is sent out anytime the user makes that change.
- We will do this using a system/class called NotificationCenter. Later in the course we learn more about this concept extensively.

Detecting when user changes text size

```
NSNotificationCenter.defaultCenter().addObserver(self, selector: "updateLabels", name:
    UIContentSizeCategoryDidChangeNotification, object: nil)
```

Signing up to listen for the notification. viewDidLoad() is a great place for this


```
deinit {
    NSNotificationCenter.defaultCenter().removeObserver(self, name:
        UIContentSizeCategoryDidChangeNotification, object: nil)
}
```

Everytime you sign up for a notification, you MUST also remove yourself as the observer when you are about to die. deinit is a great place for this

Detecting when user changes text size

Finally, we need to implement the selector (aka method) that is going to fire whenever we receive the notification. In it, we just need to reset the interface objects' font to their text styles:

```
NSNotificationCenter.defaultCenter().addObserver(self, selector: "updateLabels", name: UIContentSizeCategoryDidChangeNotification, object: nil)
```



```
func updateLabels() {  
  
    titleLabel.font = UIFont.preferredFontForTextStyle(UIFontTextStyleHeadline)  
    bodyLabel.font = UIFont.preferredFontForTextStyle(UIFontTextStyleBody)  
    captionLabel.font = UIFont.preferredFontForTextStyle(UIFontTextStyleCaption1)  
  
}
```

Demo